Proceedings

2012 IEEE/ACM 45th International Symposium on Microarchitecture

MICRO-45

Proceedings

2012 IEEE/ACM 45th International Symposium on Microarchitecture

1-5 December 2012 / Vancouver, British Columbia, Canada

Los Alamitos, California

Washington • Tokyo

Copyright © 2012 by The Institute of Electrical and Electronics Engineers, Inc.

All rights reserved.

Copyright and Reprint Permissions: Abstracting is permitted with credit to the source. Libraries may photocopy beyond the limits of US copyright law, for private use of patrons, those articles in this volume that carry a code at the bottom of the first page, provided that the per-copy fee indicated in the code is paid through the Copyright Clearance Center, 222 Rosewood Drive, Danvers, MA 01923.

Other copying, reprint, or republication requests should be addressed to: IEEE Copyrights Manager, IEEE Service Center, 445 Hoes Lane, P.O. Box 133, Piscataway, NJ 08855-1331.

The papers in this book comprise the proceedings of the meeting mentioned on the cover and title page. They reflect the authors' opinions and, in the interests of timely dissemination, are published as presented and without change. Their inclusion in this publication does not necessarily constitute endorsement by the editors, the IEEE Computer Society, or the Institute of Electrical and Electronics Engineers, Inc.

IEEE Computer Society Order Number E4924
ISBN 978-0-7695-4924-8

Additional copies may be ordered from:

IEEE Computer Society
Customer Service Center
10662 Los Vaqueros Circle
P.O. Box 3014
Los Alamitos, CA 90720-1314
Tel: + 1 800 272 6657
Fax: + 1 714 821 4641
http://computer.org/cspress
csbooks@computer.org

IEEE Service Center
445 Hoes Lane
P.O. Box 1331
Piscataway, NJ 08855-1331
Tel: + 1 732 981 0060
Fax: + 1 732 981 9667
http://shop.ieee.org/store/
customer-service@ieee.org

IEEE Computer Society
Asia/Pacific Office
Watanabe Bldg., 1-4-2
Minami-Aoyama
Minato-ku, Tokyo 107-0062
JAPAN
Tel: + 81 3 3408 3118
Fax: + 81 3 3408 3553
tokyo.ofc@computer.org

Individual paper REPRINTS may be ordered at: <reprints@computer.org>

Editorial production by Bob Werner
Cover art production by Joseph Daigle / Studio Productions

IEEE Computer Society
***Conference Publishing Services* (CPS)**
http://www.computer.org/cps

2012 IEEE/ACM 45th Annual International Symposium on Microarchitecture

MICRO 2012

Table of Contents

Session IIA - GPUs and SIMD

Session IIB - Energy I

Session IIIA – Big Data

Session IIIB – Energy II

Session IVA – Memory Systems II

Session IVB – Interconnects

Session VA – Core Design

Session VB – Coherence and Consistency

Message from the General Chair

I would like to extend a warm welcome to all Micro-45 attendees to Vancouver, British Columbia. As Micro's first time in Canada, it is perhaps appropriate as a late fall conference that it is hosted in Canada's warmest city. The Vancouver metropolitan area is Canada's third largest by population, and Vancouver has consistently ranked near the top in worldwide "livable city" rankings for more than a decade. I am hopeful that first time visitors as well as those well acquainted with this great city and surrounding region take this opportunity to explore the many things it has to offer. We have planned an exciting banquet and excursion to the Vancouver Aquarium, Canada's largest and one of Vancouver's top attractions.

Micro continues its tradition as the premier forum for presentation and discussion of new ideas in microarchitecture, compilers, hardware/software interfaces, and design of advanced computing and communication systems. As the foundation of any successful conference is the program, I would like to thank first and foremost Program Chair Onur Mutlu. Onur worked tirelessly to establish the program committee, develop the review process, run the program committee meeting and develop the physical program. Two unique aspects this year proposed and developed by Onur are the Lightning Session and the Poster Session. This year we are also pleased to have three invited speakers: Charles Webb of IBM, Turner Whitted of Microsoft Research and this year's Bob Rau award recipient Josh Fisher.

I am also grateful to the rest of the organizing committee. The huge effort involved in putting on a Micro conference would not have been possible without their assistance. Finance Chair Tor Aamodt was responsible for raising corporate donations. Tutorials and Workshops Chair Derek Chiou supervised the administration of tutorials and workshops, which are an important component of the overall conference. This year we have five workshops and three tutorials during the weekend before the conference. Publication Chair Benjamin Lee handled the publication process. This year we are publishing both main conference and workshop papers online and providing them to attendees in electronic form, with printed proceedings available upon special order. Registration Chair Arrvindh Shriraman has been indispensable with setting up and supervising the online registration process, and physical registration will be assisted by some of his students. Student Travel Grant Chair Matt Farrens is running the travel grant process to control the distribution of travel grants. Finally, I would like to acknowledge Web and Submissions Chairs Chris Fallin, Justin Meza and Vivek Seshadri who maintained the paper submissions web site and the conference web site.

I would also like to thanks our corporate sponsors, whose valuable funding helped make the conference possible: AMD, ARM, HP, IBM Research, Intel, Microsoft and Qualcomm. The Steering Committee has provided helpful advice and guidance. Finally we are indebted to our society sponsors: ACM SIGMICRO and IEEE TC-uARCH for their continued interest and support.

Steve Melvin
General Chair, Micro-45

Message from the Program Chair

I am delighted to present you the technical program for Micro-45. This year's program consists of 40 technical papers and two keynote speeches. In addition to appearing in the mostly-online proceedings, the papers will be presented in three different forms during the conference: as a 100-second key idea presentation during the opening lightning (or, teaser) session, a full 25-minute length presentation during paper sessions, and as a poster in the poster session. We hope this format will allow the benefits of having parallel sessions while giving the authors an opportunity to reach out to most of the Micro attendees during the lightning and poster sessions, and allowing the attendees to delve into the presented works at different levels of depth.

The 40 papers you will be exposed to were selected from the 228 paper submissions. The Program Committee (PC), consisting of 44 distinguished experts, made this selection, closely aided by an External Review Committee (ERC) of 106 additional experts active in our field. Additional external reviewers aided both committees. In total, 248 reviewers wrote 1325 reviews, 815 of which were written by the Program Committee members. The average number of reviews per paper was 5.8. 216 submissions received at least 5 reviews, 159 received at least 6 reviews, and 44 received at least 7 reviews. I myself assigned all reviewers for each paper after careful examination of the paper, except for those 29 I had a conflict-of-interest with. The review process of the 26 submissions I had a conflict-of-interest with were completely handled by Scott Mahlke, and that of the 3 submissions both Scott and I had a conflict-of-interest with were completely handled by Derek Chiou. I thank both of them for their dedicated service.

The authors were given a chance to respond to the initial evaluations of the reviewers via the rebuttal process. After the rebuttal period ended on August 16, 2012, all reviewers were asked to revise their evaluations (both numerical scores and written comments) for each paper, taking into account the author response. Extensive online discussions and review revisions occurred during the 10 days before the PC meeting. At least 160 of the submissions were discussed online. External reviewers were fully involved in the entire review and discussion process leading up to the PC meeting. I requested additional expert reviews after rebuttal for some submissions where there were differences of opinion among the reviewers.

The Program Committee met for 1.5 days on August 25-26, 2012 at the Hilton Chicago O'Hare Airport to make the final selection of papers. All PC members were present for the entire first day of the meeting, and 42 out of the 44 members were present on the second day. In total, 82 submissions were discussed during the meeting, in a rank order of overall merit score that takes into account both pre-rebuttal and post-rebuttal overall merit scores of each reviewer, weighted by the reviewer's expertise and generosity, which I intend to disclose in another document. Any PC member who did not have a conflict-of-interest with a paper or an external reviewer who reviewed the paper had the choice to request the discussion of a submission during the meeting. Several controversial submissions received extensive discussions during the meeting, which was facilitated by the multi-day nature of the meeting. I sought consensus across the *entire PC* to make a decision for each paper, and resorted to a full vote across the *entire PC* to make the accept/reject decision only in cases where consensus was not reached. I am very grateful for the hard work, professionalism, fairness, and thorough reviews of the PC members during, before and after the PC meeting.

After the accept/reject decisions were made, with the help of the PC members, I assembled summary statements that provide a summary of the PC discussion for most of the papers that were discussed but rejected in the PC meeting. I hope this information provided more transparency to the PC discussion process and would be useful for the authors in improving their research.

Each accepted paper was shepherded by one PC member between September 20 and November 5, 2012. The authors of each paper provided the shepherd with a "summary of changes" document describing how

they would address the reviewer feedback. The shepherd PC member assembled and conveyed the feedback of the PC and the reviewers to the authors, and aided the authors in improving the paper.

This year, we introduced several new and different aspects into both the submission and review process as well as the final program. The main goals were to improve: 1) the quality of the decisions during the paper selection process, 2) the transparency of the paper selection process, 3) the level of involvement of the external reviewers in the selection process, 4) the authors' ability to respond to initial reviewer evaluations and potential questions that may come up after the rebuttal process, 5) the quality of the final program, and 6) the quality of interactions during the conference, especially in the presence of parallel sessions. I intend to describe the reasoning behind these new and different aspects, and detail the feedback we received on them in a longer experience report separately. Here, I briefly list some of them:

- The submission format of the papers was made the same as the final format to, most importantly, improve the fairness of paper selection process and to begin an attempt to standardize the submission format of Micro and other architecture conferences. I am delighted to see that this format was improved and adopted for the purposes of HPCA 2013 and ISCA 2013 submissions.
- The authors were given the option to upload an appendix in addition to the submission. Reviewers were not required to read the appendix; some of them did. An important purpose was to give the authors a chance to anticipate potential detail questions and satisfy the curiosity of the reviewers who wanted to dig deeper.
- The authors were given the option to upload a document summarizing any past reviews they may have received for a previous version of the submission, if any, to a past venue, along with a description of how they addressed those reviews. The purpose was to give the authors a chance to proactively address potential concerns that might appear after the rebuttal period.
- The Program Committee meeting spanned 1.5 days instead of the conventional 1-day meeting. This allowed the committee to deliberate more and reduced the pressure on the committee for making hasty decisions.
- An External Review Committee was employed to aid the Program Committee by augmenting the expertise of the PC members. ERC members committed to reviewing 3-6 submissions before the conference. Some ERC members reviewed as many as 7 submissions.
- All external reviewers were involved in the paper selection process, before and after the rebuttal until up to the Program Committee meeting. External reviewers were allowed to see rebuttals and other reviews, discuss the paper with other reviewers, and update their scores and reviews based on the rebuttal and the discussion.
- The rebuttal process was made more transparent to the authors. All initial numerical scores of reviewers were exposed to the authors during the rebuttal process. The review form explicitly allowed the reviewers to specify the three most important specific questions to be answered by the authors.
- The reviewers were required to read the author rebuttals and revise their reviews accordingly. They were asked to include "post-rebuttal comments" in their revised reviews. A large majority of reviewers updated their reviews as part of this process.
- Summary statements were provided to the authors of many rejected papers, if the paper was discussed during the PC meeting, describing the key points of the PC discussion of the paper.
- All accepted papers were shepherded to improve quality.
- A single-track lightning session will be the opening session of the conference (after the first keynote) to enable authors to get across key ideas of their papers to most of the conference attendees.
- A poster session will take place on the second day of the conference to enable the authors to discuss their work with the attendees and enable the attendees to have a technical networking session.

Clearly, this entire process has only been possible with the continuous hard work and participation of the Program Committee members, External Review Committee members, and the external reviewers who aided the PC and the ERC. I thank them gratefully. I especially thank the PC members who, in addition to thoroughly reviewing papers, caringly and diligently shepherded the accepted papers – I know the hard work they put in for this effort, as I was CC'ed in most of the correspondence between the authors and the shepherds. Special thanks to Scott Mahlke, who coordinated the review process of a considerably large number of (26) papers I had conflict-of-interest with.

Vivek Seshadri, the head Micro-45 submissions and web chair, was a key enabler of the entire submission and review process, as he tremendously helped with all aspects of it. Without his extensive help in running the submissions and review website, preparations for the PC meeting, and attendance, assistance and note-taking during the PC meeting, none of the above would have been easily possible. I would also like to thank two of my other PhD students, Chris Fallin and Justin Meza, who worked hard as the other submissions and web chairs.

I would like to especially thank Rich Belgard, Yale Patt, Tom Conte, and Wen-mei Hwu, who provided valuable advice in all steps and were always available. Many thanks to the other Steering Committee members, David Albonesi, Kemal Ebcioglu, Paolo Faroboschi, Scott Mahlke, Margaret Martonosi, Bill Mangione-Smith, and Milos Prvulovic for being helpful when needed and supportive. I am very much grateful to Pradip Bose, Ronny Ronen and Ben Zorn, whom I consulted with on many issues. Last year's Program Co-chairs, Andreas Moshovos and Milos Prvulovic, and the 2010 Program Chair Sanjay Patel provided feedback on their recent experience along with useful documents. Doug Burger, James Hoe, Trevor Mudge, Jim Smith, Guri Sohi, Per Stenstrom, and Chris Wilkerson also provided valuable feedback and opinions at various points in time, and I thank them for that. I am also very thankful to the General Chair Steve Melvin for being supportive and acting as a sounding board.

Finally, I would like to thank our keynote speakers, Charles Webb and Turner Whitted, for accepting my invitation, the Publications Chair Ben Lee and the Registration Chair Arrvindh Shriraman for putting up with my requests, and Laura McGee and Can Alkan for their help with meeting organization and the conference website.

The technical program of Micro-45 reflects a strong and thriving community effort, shaped by literally hundreds of contributors, including especially the submitting authors, PC and ERC members, external reviewers, invited speakers, and others acknowledged above (and yet others I may have unintentionally forgotten to acknowledge) whom have directly or indirectly affected the process. I would like to thank especially the submitting authors for the strong and diverse submissions that have allowed the review committees to choose from a strong set of technical papers.

I hope you enjoy Micro-45, and are looking forward to the technical program as much as I am. It was an honor for me to serve as the Program Chair. I would very much welcome any feedback you may have on anything related to Micro-45, especially on the new things we tried this year.

Onur Mutlu
Program Chair, Micro-45

Organizing Committee

General Chair
Stephen Melvin, Consultant

Program Chair
Onur Mutlu, Carnegie Mellon University

Finance Chair
Tor Aamodt, British Columbia

Tutorials and Workshops Chair
Derek Chiou, UT Austin

Publication Chair
Benjamin Lee, Duke University

Registration Chair
Arrvindh Shriraman, Simon Fraser University

Web and Submissions Chairs
Chris Fallin, Carnegie Mellon University
Justin Meza, Carnegie Mellon University
Vivek Seshadri, Carnegie Mellon University

Student Travel Grant Chair
Matt Farrens, UC Davis

Steering Committee
David Albonesi, Cornell
Richard Belgard, Consultant (Chair)
Tom Conte, Georgia Tech.
Kemal Ebcioglu, Global Supercomputing
Paolo Faraboschi, HP Labs
Wen-mei Hwu, University of Illinois
Scott Mahlke, University of Michigan
Margaret Martonosi, Princeton
Bill Mangione-Smith, IP Navigation Group
Yale Patt, UT Austin
Milos Prvulovic, Georgia Tech

Program Committee

Tor Aamodt, British Columbia
David Albonesi, Cornell
Krste Asanovic, Berkeley
Todd Austin, Michigan
Rajeev Balasubramonian, Utah
Richard Belgard, Consultant
Pradip Bose, IBM Research
David Brooks, Harvard
Douglas Carmean, Intel
Derek Chiou, UT Austin
Robert Colwell, DARPA
Tom Conte, Georgia Tech
Chita Das, Penn State
Michel Dubois, USC
Evelyn Duesterwald, IBM
Lieven Eeckhout, Ghent University
Boris Grot, EPFL
Nikos Hardavellas, Northwestern
James Hoe, CMU
Wen-mei Hwu, Illinois
Engin Ipek, Rochester
Daniel Jimenez, UT San Antonio
Hyesoon Kim, Georgia Tech
Konrad Lai, Intel
Gabriel Loh, AMD
Ahmed Louri, NSF/Arizona
Scott Mahlke, Michigan
Srilatha Manne, AMD
Andreas Moshovos, Toronto
Trevor Mudge, Michigan
Yale Patt, UT Austin
Milos Prvulovic, Georgia Tech
Moinuddin Qureshi, Georgia Tech
Ronny Ronen, Intel
Yanos Sazeides, Cyprus
Michael Schlansker, HP Labs
Andre Seznec, IRISA/INRIA
Michael Shebanow, Samsung
Burton Smith, Microsoft
Jared Smolens, Oracle
Viji Srinivasan, IBM
Chris Wilkerson, Intel
Yuanyuan Zhou, UCSD
Craig Zilles, Illinois

External Review Committee

Jung Ho Ahn, Seoul National University
Alaa Alameldeen, Intel
Murali Annavaram, USC
Leslie Barnes, AMD
Brad Beckmann, AMD
Ricardo Bianchini, Rutgers
Doug Burger, Microsoft
Alper Buyuktosunoglu, IBM
John Carter, IBM
Luis Ceze, Washington
Jichuan Chang, HP Labs
Zeshan Chishti, Intel
Sangyeun Cho, Pitt
Adrian Cristal, Barcelona Supercomputing Center and Microsoft
Reetuparna Das, Michigan
Ganesh Dasika, ARM
Nirav Dave, SRI
Srinivas Devadas, MIT
Eiman Ebrahimi, NVIDIA
Elmootazbellah Elnozahy, IBM
Mattan Erez, UT-Austin
Yoav Etsion, Technion
Babak Falsafi, EPFL
Matt Farrens, UC Davis
Kayvon Fatahalian, CMU
Krisztian Flautner, ARM
Kanad Ghose, Binghamton
Andy Glew, MIPS and comp-arch.net
Brian Gold, Oracle
Paul Gratz, TAMU
Rajiv Gupta, UC Riverside
Shantanu Gupta, Intel
Sudhanva Gurumurthi, Virginia/AMD
Dan Hammerstrom, DARPA
Maurice Herlihy, Brown
Glenn Hinton, Intel
Brian Hirano, Oracle
Jim Holt, Freescale
Michael Huang, Rochester
Jaehyuk Huh, KAIST
Hillery Hunter, IBM
Ravi Iyer, Intel
Norm Jouppi, HP Labs
Mahmut Kandemir, Penn State
Manolis Katevenis, FORTH and Univ of Crete
Steve Keckler, NVIDIA and UT-Austin
Omer Khan, UConn
John Kim, KAIST
Nam Sung Kim, Wisconsin
Vijaykrishnan Narayanan, Penn State
Avinash Kodi, Ohio
Christos Kozyrakis, Stanford
Eren Kursun, IBM
Jim Larus, Microsoft
James Laudon, Google
Chang Joo Lee, Intel
Benjamin C. Lee, Duke
Hsien-Hsin Lee, Georgia Tech
Jaejin Lee, SNU
Ruby Lee, Princeton
David Lie, University of Toronto
Shan Lu, Wisconsin
Milo Martin, Penn
Mojtaba Mehrara, NVIDIA
Stephan Meier, Apple
David Meisner, Facebook
Asit Kumar Mishra, Penn State
Subhasish Mitra, Stanford
Naveen Muralimanohar, HP Labs
Ravi Nair, IBM
Satish Narayanasamy, Michigan
Emre Ozer, ARM
Sanjay Patel, Illinois
Timothy Pinkston, USC
Brian Prasky, IBM
Partha Ranganathan, HP Labs
VJ Reddi, UT-Austin
Steve Reinhardt, AMD
Glenn Reinman, UCLA
Jose Renau, UCSC
Scott Rixner, Rice
Ali Saidi, ARM
Simha Sethumadhavan, Columbia
Xipeng Shen, William and Mary
Tim Sherwood, UCSB
Jim Smith, Wisconsin
Dan Sorin, Duke University
Per Stenstrom, Chalmers
Karin Strauss, MSR
Edward Suh, Cornell
Aater Suleman, Calxeda
Dam Sunwoo, ARM
Oliver Temam, INRIA
Radu Teodorescu, Ohio State
Josep Torrellas, Illinois
Osman Unsal, BSC

Mateo Valero, UPC
Thomas Wenisch, Michigan
Emmett Witchel, UT-Austin
Yuan Xie, Penn State
Sudhakar Yalamanchili, Georgia Tech
Qing Yang, Rhode Island
Sungjoo Yoo, Postech
Doe Hyun Yoon, HP Labs
Huiyang Zhou, NCSU
Ben Zorn, Microsoft Research

MICRO 2012 Reviewers

Yoshi Abe, CMU
Almutaz Adileh, EPFL
Berkin Akin, CMU
Haitham Akkary, AUB
Fuad Al Tabba, Oracle
Erik Altman, IBM
Amin Ansari, Illinois
Adria Armejach, BSC
David August, Princeton
Rachata Ausavarungnirun, CMU
Seungjae Baek, Pittsburgh
Ali Bakhoda, UBC
Ioana Baldini, IBM Research
Avram Bar-Cohen, DARPA
Christopher Batten, Cornell
Michela Becchi, Missouri
Oren Ben-Kiki, Intel
Keren Bergman, Columbia
Kerry Bernstein, DARPA
Valeria Bertacco, Michigan
Jesse Beu, Georgia Tech
Rishiraj Bheda, Georgia Tech
Hans Boehm, HP Labs
Michael Bond, Ohio State
Mary Brown, IBM
Francisco Cazorla, BSC
Gaurav Chadha, Michigan
Kevin Chang, CMU
Sai Charan, UC Riverside
Karam Chatha, Arizona State
Niladrish Chatterjee, Utah
Lizhong Chen, USC
Hsiang-yun Cheng, Penn State
Naveen Cherukuri, Intel
Trishul Chilimbi, Microsoft Research
Hyoun Kyu Cho, Michigan
Fred Chong, UC Santa Barbara
David Christie, AMD
John Chu, AMD
Eric Chung, Microsoft Research
Robert Cohn, Intel
Jason Cong, UCLA
Kypros Constantinides, Microsoft Research
Ayse Coskun, Boston University
Yigit Demir, Northwestern
Chen Ding, Rochester
Sang Do, USC
Ronald Dreslinski, Michigan
Ahmed El-Shafiey, UBC
Stijn Eyerman, Ghent
Chris Fallin, CMU
Min Feng, NEC Labs
Michael Ferdman, Stony Brook University
Wilson Fung, UBC
Ron Gabor, Intel
Siddharth Garg, Waterloo
Joseph Gebis, Oracle
Phillip Gibbons, Intel Labs
Boris Ginzburg, Intel
Eugene Gorbatov, Intel
R Govindarajan, Indian Institute of Science
Ed Grochowski, Intel
Anthony Gutierrez, Michigan
Faruk Guvenilir, UT-Austin
Sebastian Hack, Saarland University
Tim Harris, Oracle Labs
Milad Hashemi, UT-Austin
Mike Healy, IBM Research
Eric Hein, Georgia Tech
Mark Hill, Wisconsin
Ron Ho, Oracle
Sunpyo Hong, Georgia Tech
Chris Hughes, Intel
Ibrahim Hur, BSC
Canturk Isci, IBM Research
Ben Jaiyen, CMU
Aamer Jaleel, Intel
Gangwon Jo, Seoul National University
Lizy John, UT Austin
Ryan Johnson, Toronto
Jose Joao, UT-Austin
Adwait Jog, Penn State
Ulya Karpuzcu, Minnesota
Ramesh Karri, NYU Poly
Onur Kayiran, Penn State
Cansu Kaynak, EPFL
Samira Khan, CMU/Intel
Khubaib, UT-Austin
Changkyu Kim, Intel
Dong Wan Kim, UT Austin
E.J. Kim, Texas A&M
Jangwoo Kim, POSTECH
Jungrae Kim, UT Austin
Minjang Kim, Qualcomm
Yoongu Kim, CMU
Nevin Kirman, Intel
Marios Kleanthous, Cyprus
Amit Kumar, Intel

Pranith Kumar, Georgia Tech
Nagesh Lakshminarayana, Georgia Tech
Benjamin Lee, IBM Research
Chang Joo Lee, Intel
Donghyuk Lee, CMU
Jaekyu Lee, Georgia Tech
Joohwan Lee, Georgia Tech
Charles Lefurgy, IBM
Brian Leung, Intel
Ching-Kai Liang, Georgia Tech
Changhui Lin, UC Riverside
Jamie Liu, CMU
Pejman Lotfi-Kamran, EPFL
Danny Lynch, NVIDIA
Rakan Maddah, Pittsburgh
Ken Mai, CMU
Abhinandan Majumdar, Cornell
Dilan Manatunga, Georgia Tech
Bill Mangione-Smith, Phase Two LLC
Rajit Manohar, Cornell
Mehrtash Manoochehri, USC
Rami Melhem, Pittsburgh
Stephen Melvin, Zytek
Justin Meza, CMU
Rustam Miftakhutdinov, UT-Austin
Abdullah Muzahid, UT San Antonio
Nachi Nachiappan, Penn State
Veynu Narasiman, UT-Austin
Stephen Neuendorffer, Xilinx
Panagiota Nicolaou, Cyprus
Chrysostomos Nicopoulos, Cyprus
Mike O'Connor, AMD Research
Jungju Oh, Georgia Tech
Kunle Olukotun, Stanford
Oscar Palomar, BSC
Andreas Panteli, Cyprus
Michael Papamichael, CMU
Dong-kook Park, Intel
Sunjae Park, Georgia Tech
Yongjun Park, Michigan
Sudeep Pasricha, Colorado State
Bharath Pattabiraman, Northwestern
Gennady Pekhimenko, CMU
Fernando Pereira, UFMG
Keshav Pingali, UT Austin
Jason Poovey, Georgia Tech
Michael Powell, Intel
Andreas Prodromou, Cyprus
Joseph Pusdesris, Michigan
Shanxiang Qi, Illinois
Xuehai Qian, Illinois
Rodric Rabbah, IBM
Paul Racunas, Intel
Shlomo Raikin, Intel
Brian Railing, Georgia Tech
Bipin Rajendran, IBM
Lihu Rappoport, Intel
Minsoo Rhu, UT Austin
Efraim Rotem, Intel
Daniel Ben-Dayan Rubin, Intel
P Sadayappan, Ohio State
Juan Carlos Saez Alcaide,
Complutense University of Madrid
Suleyman Sair, Intel
Mehrzad Samadi, Michigan
Satya, CMU
Rob Schreiber, HP Labs
Naser Sedaghati, Ohio State
Sangmin Seo, Seoul National University
Vivek Seshadri, CMU
Jagdeep Shah, DARPA
Manjunath Shevgoor, Utah
Arrvindh Shriraman, Simon Fraser University
Jaewoong Sim, Georiga Tech
Inderpreet Singh, UBC
Anand Sivasubramaniam, Penn State
Michael Spear, Lehigh
Vilas Sridharan, AMD
Santhosh Srinath, NVIDIA
Sadagopan Srinivasan, Intel
Srikanth Srinivasan, Intel
Jared Stark, Intel
Lavanya Subramanian, CMU
Jinho Suh, Intel
Jakub Szefer, Princeton
Sudha Thiruvengadam, AMD
Mithuna Thottethodi, Purdue/AMD
Yingying Tian, UT San Antonio
Sasa Tomic, BSC
Francis Tseng, Intel
Dean Tullsen, UC San Diego
George Tziantzioulis, Northwestern
Aniruddha Udipi, ARM
Alexander Veidenbaum, UC Irvine
Xavier Vera, Intel
Carlos Villavieja, UT-Austin
Thomas Vogelsang, Rambus
Carl Waldspurger, Consultant
Yan Wang, UC Riverside
Yu Wang, CMU
Zhe Wang, UT San Antonio
Uri Weiser, Technion

Gabe Weisz, CMU
Wei Wu, Intel
Hongyi Xin, CMU
Adi Yoaz, Intel
HanBin Yoon, CMU
Alenka Zajic, Georgia Tech
Ayal Zaks, Intel
Tao Zhang, Penn State
Xiangyu Zhang, Purdue
Jishen Zhao, Penn State
Hongzhong Zheng, Rambus
Tianhao Zheng, UT Austin
Xiaoyun Zhu, VMware

2012 IEEE/ACM 45th Annual International Symposium on Microarchitecture

FPB: Fine-grained Power Budgeting to Improve Write Throughput of Multi-level Cell Phase Change Memory

Lei Jiang † Youtao Zhang § Bruce R. Childers§ Jun Yang †
† Electrical and Computer Engineering Department
§ Department of Computer Science
University of Pittsburgh
†{lej16,juy9}@pitt.edu §{zhangyt,childers}@cs.pitt.edu

Abstract

As a promising nonvolatile memory technology, Phase Change Memory (PCM) has many advantages over traditional DRAM. Multi-level Cell PCM (MLC) has the benefit of increased memory capacity with low fabrication cost. Due to high per-cell write power and long write latency, MLC PCM requires careful power management to ensure write reliability. Unfortunately, existing power management schemes applied to MLC PCM result in low write throughput and large performance degradation.

In this paper, we propose Fine-grained write Power Budgeting (FPB) for MLC PCM. We first identify two major problems for MLC write operations: (i) managing write power without consideration of the iterative write process used by MLC is overly pessimistic; (ii) a heavily written (hot) chip may block the memory from accepting further writes due to chip power restrictions, although most chips may be available. To address these problems, we propose two FPB schemes. First, FPB-IPM observes a global power budget and regulates power across write iterations according to the step-down power demand of each iteration. Second, FPB-GCP integrates a global charge pump on a DIMM to boost power for hot PCM chips while staying within the global power budget. Our experimental results show that these techniques achieve significant improvement on write throughput and system performance. Our schemes also interact positively with PCM effective read latency reduction techniques, such as write cancellation, write pausing and write truncation.

1. Introduction

Phase Change Memory (PCM) has emerged as a leading technology to alleviate the leakage and scalability problems of traditional DRAM [24]. With advantages over DRAM, such as near zero cell leakage, better scalability and comparable read speed, PCM is poised to replace a significant portion of DRAM in main memory [12, 19, 31]. A PCM cell uses different resistances to represent logic bits. Single-level cell PCM (SLC) differentiates between two resistance levels to store a bit (logic '0' or '1'). Due to the large resistance contrast between '0' and '1', intermediate levels can be used to store multiple bits per cell in multi-level cell PCM (MLC).

Although memory capacity is effectively increased with low per bit fabrication cost in MLC PCM, this technology has shorter write endurance, longer access latency, and larger write power than SLC PCM. Many schemes have been proposed to address some of these issues. In addition to schemes for SLC PCM [12, 18, 19, 25, 26, 31], Qureshi *et al.* proposed to transform MLC PCM pages to SLC pages for fast access [21]; Qureshi *et al.* proposed to pause MLC write operations and prioritize read operations to improve performance [20]; Jiang *et al.* proposed write truncation to reduce average MLC write time and use ECC to correct write errors [10]. Joshi *et al.* proposed an energy-efficient programming scheme for MLC PCM [11].

While past research has made significant strides, high PCM write power remains a major obstacle to improving throughput. For example, a recent study showed that the power provided by DDR3-1066×16 memory allows only 560 SLC PCM cells to be written in parallel [8], i.e., at most two 64B lines can be written simultaneously using Flip-n-Write [4]. Hay *et al.* proposed to track the available power budget and issue writes continuously as long as power demands can be satisfied [8]. Their heuristic works well for SLC PCM based main memory.

Unfortunately, applying the heuristic to MLC PCM results in low write throughput and large performance degradation: On average, we observed a 51% performance degradation over an ideal baseline without power limit. We identified two major problems for MLC PCM that limit throughput and performance for this heuristic.

The first problem is that allocating the same power budget for the entire duration of an MLC line write is often too pessimistic. A MLC PCM write is done in iterations, starting with a RESET pulse and followed by a varying number of SET pulses. The RESET pulse is short and of large magnitude while the SET pulse is long and of low magnitude. In addition, when writing one PCM line, most cells in the line require only a small number of SET pulses [10]. Allocating power according to the RESET power request and for the duration of the longest cell write is power inefficient.

The second problem is that one heavily written (hot) PCM chip may block the memory subsystem even though most memory chips are idle. This phenomenon arises because the power that each chip can provide is restricted by the area of its charge pump. When multiple writes compete for a single chip, some writes have to wait to avoid exceeding the charge pump's capability. Otherwise, cell writes become unreliable.

We propose two new fine-grained power budgeting (FPB) schemes to address these problems:

- **FPB-IPM** is a scheme that regulates write power on each write iteration in MLC PCM. Since writing one MLC line requires multiple iterations with step-down power requirements, FPB-IPM aims to (i) reclaim any *unused* write power after each iteration and (ii) reduce the maximum power requested in a write operation by splitting the first RESET iteration into several RESET iterations. By enabling more MLC line writes in parallel, FPB-IPM improves memory throughput.
- **FPB-GCP** is a scheme that mitigates power restrictions at chip level. Rather than enlarging the charge pump in an individual chip, FPB-GCP integrates a single global charge pump (GCP) on a DIMM. It dynamically pumps extra power to hot chips in the DIMM. Since GCP has lower effective power efficiency (i.e., the percentage of power that can be utilized for writes), we consider different cell mapping optimizations to maximize throughput.

1072-4451/12 $26.00 © 2012 IEEE
DOI 10.1109/MICRO.2012.10

IEEE computer society

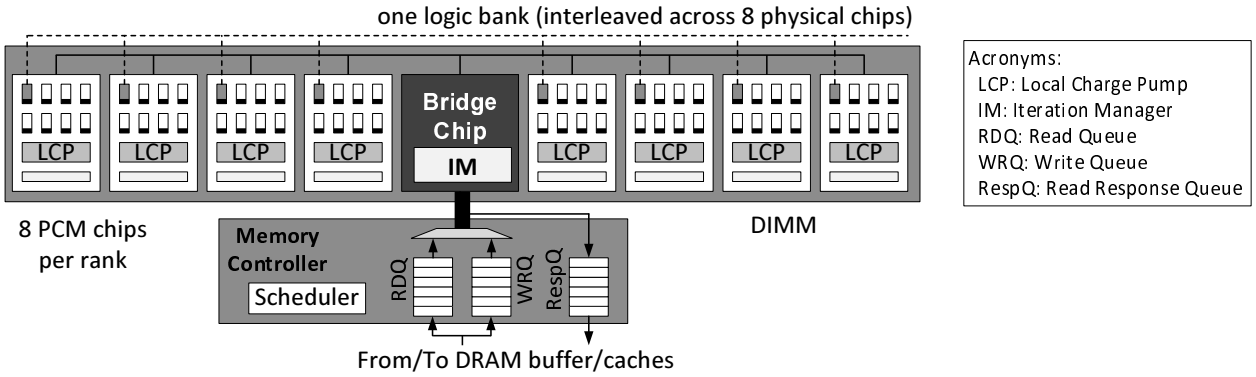


Figure 1: The baseline architecture of a MLC PCM-based memory subsystem (One DIMM).

We evaluate our proposed schemes and compare them to state-of-the-art PCM power management. Our results show that FPB-IPM and FPB-GCP are orthogonal designs that together successfully address the problems described above for MLC write operations. On average, FPB achieves a 3.4× throughput improvement and 76% performance improvement over state-of-the-art power budgeting.

The rest of the paper presents background and motivation for our designs in Section 2. FPB-IPM and FPB-GCP are described in Sections 3 and 4. We present our experimental methodology and analyze results in Section 5 and 6 respectively. Related work is discussed in Section 7. Section 8 concludes the paper.

2. Background and motivation

In this section, we first discuss a typical MLC PCM memory architecture and details of MLC write operations. Next, we motivate our designs by analyzing how simple power management heuristics behave for MLC PCM.

2.1. MLC PCM memory architecture

Our baseline architecture of a MLC PCM memory subsystem is shown in Figure 1. Similar to a traditional DRAM organization, a DIMM has eight memory chips (PCM) that are organized into eight logical banks. Due to non-deterministic MLC PCM write characteristics [10], we adopt the *universal memory interface* design proposed by Fang *et al.* [7]. In Figure 1, device control is performed collaboratively between the on-CPU memory controller and the on-DIMM bridge. The memory controller's scheduler issues requests in the read queue (RDQ) and write queue (WRQ) according to bus availability, bank availability, circuit timing constraints, and global DIMM and local chip power budgets. Completed read requests (from MLC PCM banks) wait in the read response queue (RespQ) until the bus or interconnect is available at which point the read data is sent back to the cores [7].

Memory interface: a *universal memory interface* [7] makes PCM timing and device-specific management issues transparent to memory controller. Instead of memory controller, a bridge chip on DIMM tracks the status of each DIMM and ongoing access operations. A new protocol is proposed to avoid conflicts on the shared data bus and to reply memory controller when requested data is ready. In this paper, we adopted this design to handle the communication between memory controller and bridge chip and leave PCM DIMM/chip management to bridge chip.

Different cell stripping methods: In this paper, we strip cells from one memory line across all chips in our baseline configuration, so that we can access all cells in one memory line in one round. There are two design alternatives:

- Stripping cells across half of the chips, and accessing one line in one round. Each chip handles twice as many cells and requires wider bus/peripherals. This is similar to chopping each chip into two sub-chips, or simply doubling the number of chips and using only half of them for one access. Our techniques can be applied to either case.
- Stripping cells across half of the chips and accessing one line in two rounds. Each chip handles the same number of cells as stripping cells across all chips. However, the read and write latency to memory array is doubled, which will harm system performance.

2.1.1. Non-deterministic MLC write MLC PCM devices widely adopt *program-and-verify* (P&V) [2, 15] to ensure programming (write) accuracy. For a given PCM line write, only a subset of cells in the line need to be changed [4, 31]. For these cells, the write circuit first injects a RESET pulse with large voltage magnitude to place them in similar states, and then injects a sequence of SET pulses with low voltage magnitudes. After each SET pulse, a read/verify operation is performed. A cell write is terminated when its target MLC resistance level is reached. The line write finishes when all cell writes are completed.

Due to process variations and material fluctuation [3, 14], non-determinism arises for MLC PCM writes. The cells comprising a MLC PCM line can take a varying number of iterations to finish (e.g., one cell might take a few iterations, while another may take the worst case number). Further complicating cell programming, the *same* cell may require a different number of iterations to finish for different write instances. Studies have shown that most cells finish in only a small number of iterations [20]. Jiang *et al.* proposed write truncation to speed up MLC write accesses [10].

To handle non-deterministic MLC PCM writes, it is beneficial to divide PCM device control between the memory controller and the bridge chip. Fang *et al.* evaluated the details of this division [7]. If an approach similar to DRAM is employed, i.e., the on-CPU memory controller does all device control, the memory controller may have to assume that all MLC write operations take the worst case number of iterations, which greatly degrades performance.

2.1.2. DIMM power budget PCM requires much higher per-cell write power than DRAM. Hay *et al.* calculated that the power provided by a typical DDR3-1066×16 DRAM memory allows up to 560 SLC PCM simultaneous cell writes. In comparison, a single DRAM refresh round can simultaneously write one 2KB row, or 16,384 DRAM cells.

The DIMM power budget is a critical parameter in a PCM memory subsystem as it restricts the number of simultaneous cell changes.

Figure 2 reports the average number of cell changes per PCM line write under different configurations for 2-bit MLC.[1]

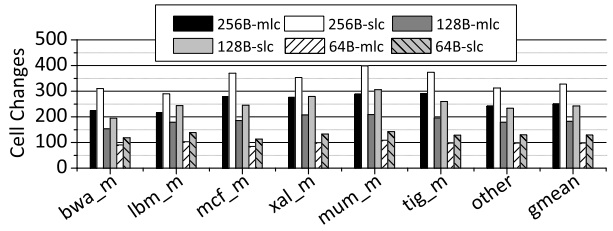


Figure 2: The cell changes under different settings.

According to Figure 2, 2-bit MLC tends to change a smaller number of cells than SLC. In addition, a larger line size results in more cell changes. In this paper, we assume the power budget per DIMM can support 560 MLC cell writes, which is the same number for SLC cell changes in previous work [8]. This represents a relaxed DIMM budget as MLC often needs more write power [11]. To explore configurations with different cell changes and thus power budget demands, we perform a wide design space exploration with different line sizes and power budgets. This also addresses the designs that use different write row buffer sizes at the device level [12].

Note that in future memory subsystems, the DIMM power budget is unlikely to increase significantly. First, PCM based main memory tends to be big to support large scale parallelizable workloads [19], which limits the budget available to a DIMM. Second, recent years have seen the need for low power DIMMs [13, 29].

2.1.3. Chip level power budget Another power restriction is the chip-level power budget. Since PCM writes require higher voltages than V_{dd}, PCM chips integrate CMOS-compatible charge pumps [6, 17] to supply required voltage and power. Studies have shown that the area of a charge pump is proportional to the maximum current that it can provide [17]:

$$A_{tot} = k \cdot \frac{N^2}{(N+1)\cdot V_{dd} - V_{out}} \frac{I_L}{f} \quad (1)$$

Here, A_{tot} is the total area overhead of the charge pump. k is a constant that depends on the process used to realize the capacitors. N indicates the number of stages in the charge pump. V_{dd} is the supply voltage and V_{out} is the target programming voltage. f denotes the charge pump's working frequency. I_L is the total write current.

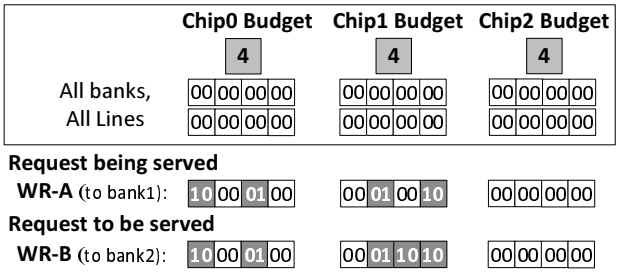


Figure 3: Writes blocked by chip level power budget (assuming three chips/bank for discussion purpose).

The write throughput of MLC PCM may be constrained by a chip power budget. In Figure 3, we assume (i) one bank spreads across three chips; (ii) the memory initially contains all 0s; (iii) the chip power budget can support 4 cell changes; (iv) the system is serving request WR-A when request WR-B arrives. They write to different banks and change 4 and 5 cells respectively (shown as shaded boxes with white font).

While these two writes change 9 cells in total and the DIMM power budget allows 12 cell changes, WR-B cannot be issued as the sum of cell changes for chip 1 is 5, which is larger than the chip's budget. If WR-B is issued, both writes may fail as there is not enough power for reliable programming.

A typical charge pump occupies 15% to 20% of a PCM chip's area [16]. Thus, it is undesirable to enlarge the charge pump to increase its maximum output current/power.

2.1.4. Our model In this paper, we adopt two-phase modeling [10, 21] for MLC PCM writes. For the DIMM power limit, we adopt the same one as past work [8]. To get the default chip power limit, we divide the DIMM power limit by 8 (i.e., eight chips per DIMM and the sum of the chip power limits equal the DIMM power limit). Our experiments consider an extensive design space. The results show that our schemes are independent of concrete model parameters. The designs are robust under a wide range of configurations.

2.2. Design motivation

To evaluate the impact of DIMM and chip power budgets for MLC PCM, Figure 4 compares several simple power management heuristics. The results are normalized to Ideal, which is a scheme that does not restrict power, i.e., a MLC write can be issued whenever a requested bank is idle.[1]

In Figure 4, DIMM-only is a case where only the DIMM power limit is enforced. There is no chip power budget, i.e., a MLC write can be issued whenever the DIMM has enough power to satisfy the write's power demand. DIMM-only adopts Hay et al.'s power management heuristic [8] to prevent the DIMM from drawing too much power. From the figure, the heuristic incurs 33% performance loss for MLC PCM, which is significantly worse than the small 2% loss for SLC PCM [8]. The reason for this discrepancy is DIMM-only does not consider MLC write iterations and it allocates the same power for the full duration of a complete line write. However, the maximum power demand happens only in the first iteration of the MLC PCM write: (i) RESET power is much larger than SET power; and, (ii) many MLC cells finish in a small number of iterations. Clearly this heuristic is overly pessimistic by budgeting the maximum write power for a line for the entire duration of the longest cell write.

Figure 4 also illustrates the impact of a PCM chip power budget. DIMM+chip uses the same heuristic as DIMM-only but it enforces both DIMM and chip power budgets. On average, there is a 51% performance loss. The increased loss over DIMM-only (i.e., the portion beyond DIMM-only's 33% loss) is due to the chip power budget. When several writes compete for a busy chip, some writes must wait to avoid exceeding the chip power budget, even though the DIMM power budget may not be exceeded. Violating the chip power budget leads to unreliable MLC writes.

To alleviate this problem for an individual chip, we tried three schemes. First, we tried to remove power competition at the chip level. PWL is an enhanced heuristic that adopts overhead-free near-perfect intra-line wear leveling. Since the lower order bits within a data block (words, double words, etc.) are more likely to be changed, intra-line wear leveling has been proposed to balance bit changes across all chips to *extend lifetime* [31]. We used intra-line wear-leveling to balance *write power requests* across chips. We assume

[1]The simulation framework and parameters are summarized in Section 5.

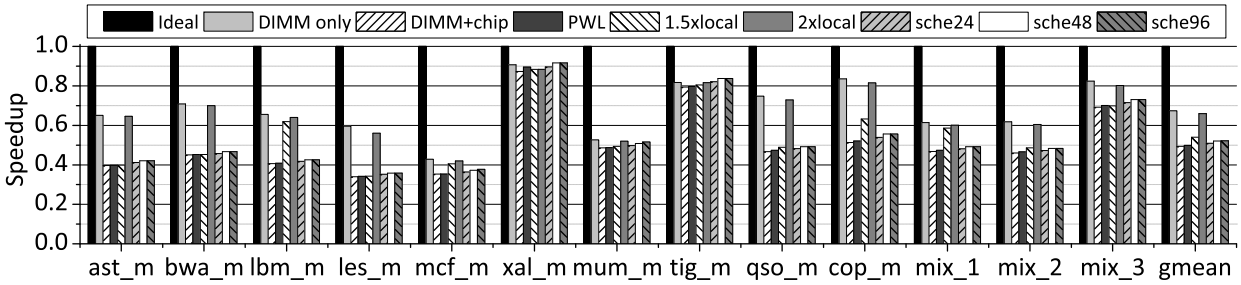


Figure 4: The performance under power restrictions for MLC PCM.

that each line is shifted by a random offset after every 8 to 100 writes and report the best results. From the figure, PWL achieves approximately a 2% improvement over DIMM+chip. We also tried different cell mapping schemes (i.e., cells are interleaved across chips) but observed similar small gains.

Second, we increased the chip's maximum power: 1.5xlocal and 2xlocal increased the chip's power budgets by 50% and 100%, respectively. From Figure 4, if the charge pump can provide 2× power, the performance loss relative to DIMM-only is negligible. Note, we have shown that the loss from Ideal to DIMM-only is due to iteration-oblivious power budgeting. The results show that the fluctuation in chip power demand is below 2× on average. However, for 50% more power, the loss is still significant, on average 20% loss. Increasing the maximum power is effective but has large area overhead.

Finally, we scheduled writes in the write queue (out-of-order) based on chip power availability. Sche-X is this scheme with an X-entry write queue. Figure 4 shows that a large write queue has little effect in mitigating performance loss.

To summarize, high MLC write power demand has a large performance impact. It cannot be resolved by state-of-the-art power management heuristics and/or simple adjustments at different levels.

3. FPB-IPM: iteration power management

In this section, we describe FPB-IPM, an iteration power management scheme for MLC PCM. For discussion purposes, we consider only the DIMM power budget in this section. The chip power budget is considered in the next section.

Figure 5 illustrates how FPB-IPM works. The scheme is token driven: in order for writes to proceed, there must be enough power tokens available to satisfy the number of bit changes required by a write. Each token represents the power for a single cell RESET. Assume that (i) two writes WR-A and WR-B arrive at the bridge chip and request to change 50 and 40 cells, respectively; and, (ii) the available DIMM power budget can support a RESET on 80 cells simultaneously, i.e., there are 80 available power tokens (APT).

Consider a simple per-write power management heuristic, as shown in Figure 5(a). This heuristic tracks APT with a counter, and releases a write only when there are enough unused tokens. Since WR-A arrives first and it requests fewer tokens than the DIMM's budget of 80 tokens, WR-A is served immediately. As a consequence, APT is reduced to 30 until WR-A finishes. In this case, WR-B stalls until WR-A returns its tokens. From Figure 5(a), the write throughput is low as the two writes do not overlap. However, some of the tokens allocated to WR-A are not actually used. For example, in the fourth iteration of WR-A, a SET is done to only 6 cells, and thus, only 3 tokens are used (SET power is half of RESET power). Nevertheless, WR-A holds all 50 tokens until the write is finished.

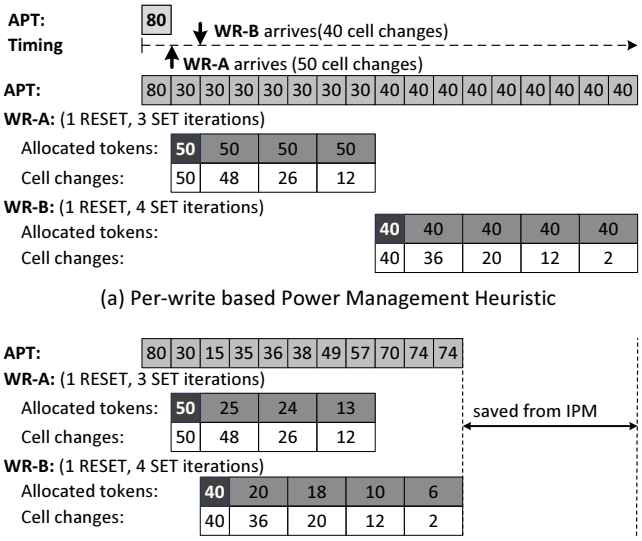


Figure 5: FPB-IPM: iteration power management (assuming SET power is half of RESET power and RESET pulse is half the length of SET pulse).

To resolve this problem, we designed FPB-IPM to reclaim unused power tokens as early as possible, which increases the number of simultaneous writes. Figure 5(b) illustrates our improved scheme. In this scheme, FPB-IPM first allocates power tokens to incoming write requests (e.g., WR-A) if there are enough ones. This is similar to the simple per-write management heuristic in Figure 5(a).

Next, after the first RESET iteration, FPB-IPM reclaims $((C-1)/C) \times PT_{RESET}$ tokens, where RESET_power = C×SET_power and PT_{RESET} is the number of tokens allocated in the first iteration. For example, half of the allocated tokens are reclaimed in write iteration 2, as shown in Figure 5(b). Because a MLC write operation finishes in a non-deterministic number of iterations, the number of cells that need to be written decreases after each SET iteration. The consumed write power also drops as the write operation proceeds. Thus, FPB-IPM also reclaims tokens after SET iterations. To reclaim unused tokens as early as possible, FPB-IPM dynamically adjusts the power token allocation on each iteration.

Starting from the 3rd iteration, FPB-IPM allocates write tokens based on cell changes in preceding iterations. In Figure 5(b), 24 tokens are allocated for the 3rd iteration of WR-A, which can SET 48 cells. This is enough tokens. Because the 2nd iteration changes 48 MLC cells, it is impossible to change more than 48 cells in the 3rd iteration and beyond.

3.1. Architecture enhancement

To enable iteration power management, FPB-IPM needs to know how many cells will be changed in each iteration. Hay *et al.* tracks SLC cell changes in the last-level cache [8]. However, this approach cannot be applied to FPB-IPM as MLC writes are non-deterministic and FPB-IPM regulates the power tokens at iteration granularity.

FPB-IPM integrates the power management logic in the bridge chip and includes two enhancements. One enhancement does a read before a write operation. The old data is compared with the new data to determine how many cells will be changed. This is slightly more expensive than *differential-write* [31] and *Flip-n-Write* [4] as these schemes perform the comparison inside the PCM chip. In FPB-IPM, the extra read increases bus contention within the DIMM. However, the read does not compete for the bus between the DIMM and the memory controller, which is a more precious resource in a multiple-DIMM memory subsystem. In the experiments, we model the cost of doing the full read before each write.

The other enhancement is each PCM chip reports the number of cells that finish after the verification operation in each write iteration. This helps FPB-IPM reclaim unused power tokens. The allocation for write iteration i, where $i >= 3$, is determined by the number of cell changes that remain after iteration $i-2$. This value can be computed during iteration $i-1$ using the information reported by the PCM devices at the end of iteration $i-2$. As a result, the allocation is available at the start of iteration i and the computation has no impact on write latency (overhead). For example, in Figure 5(b), 22 cells finished in the 2nd iteration of WR-A, which means 13 tokens are allocated in iteration 4 (i.e., 13=(2-1)/2×(48-22)).

3.2. Multi-RESET

By reclaiming unused power tokens after each iteration, the available power tokens accumulate fast. However, due to the large ratio between RESET and SET power, a write is often blocked because there are not enough tokens for the write's RESET iteration. If this iteration had a lower power demand, then the write would be more likely to go ahead without delay.

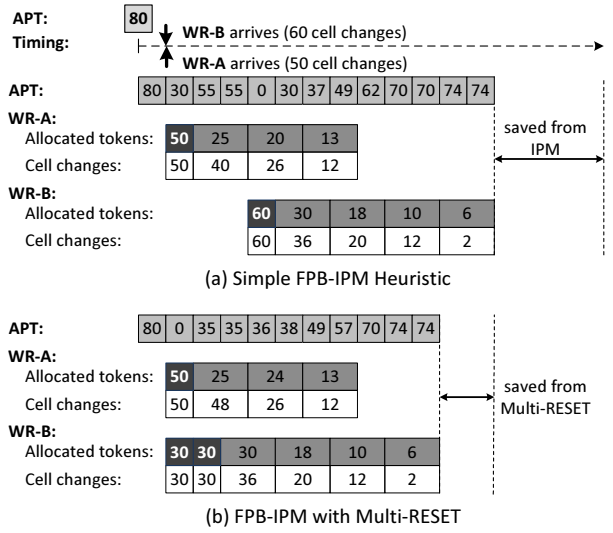


Figure 6: Multi-RESET reduces maximum power demand.

Based on this observation, we propose Multi-RESET, a technique that breaks a write's RESET iteration into several RESET iterations. Only a subset of cells are RESET in each iteration. After all cells are reset, the write does the normal SET iterations. By reducing the maximum power demand, Multi-RESET has the potential to enable more simultaneous writes. The disadvantage is increased write latency: if the RESET iteration is split into m RESET iterations, then the write latency increases by m-1 RESET iterations.

Figure 6 shows how Multi-RESET works. After issuing WR-A, the APT is 30. Since WR-B requests 60 power tokens, it has to wait until there are enough tokens (Figure 6(a)). By adopting Multi-RESET, WR-B splits the single, power-expensive RESET iteration into two less power-expensive iterations. Each iteration does a RESET for 30 cells. With this strategy, WR-B can be issued immediately. In this way, WR-A and WR-B have more overlap, resulting in improved write throughput (Figure 6(b)).

Implementing Multi-RESET requires that cells are grouped carefully. There are two approaches. One approach groups cells based on the cells to be changed. The other groups cells no matter if they are changed or not. The former tends to perform better while the latter has lower hardware overhead. In this paper, we choose the latter scheme and split cells from one chip into three groups. This requires a 2-bit control signal to a PCM chip to enable individual groups ('11' indicates all groups are RESET in one iteration).

Comparison. Multi-RESET shares similarity with *write pausing* [20], which pauses MLC writes to prioritize reads. Multi-RESET stalls the cells written in early RESET iterations until all cells to be changed are RESET. However, the design goal is different. Multi-RESET aims to lower the maximum power demand while *write pausing* aims to improve read performance. Due to the short latency pause after RESET, MLC resistance drift [30] can be ignored.

Multi-RESET also shares similarity with a *multi-round write* operation. If the DIMM has 560 power tokens, it is impossible to write a 512B line when half of all cells must be changed (i.e., 1024 cells). In this scenario, the line is written in two rounds and each round writes 512 cells. The difference is that *multi-round write* breaks one write into two non-overlapped writes, which doubles the write latency. Multi-RESET has much less latency overhead.

4. FPB-GCP: mitigating chip power restrictions by a global charge pump

In this section, we propose using a global charge pump (GCP) to mitigate performance loss due to a PCM chip's power budget. We present the architecture details and design trade-offs.

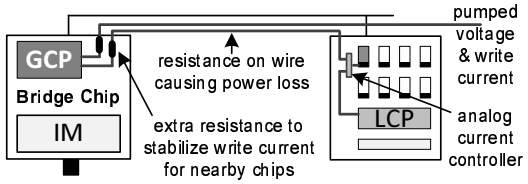


Figure 7: Integrating a global charge pump (GCP).

4.1. FPB-GCP Scheme

Section 2 describes how doubling the maximum power of the charge pumps in all chips on the DIMM can effectively eliminate the performance loss due to the chip power budget. This strategy incurs a large area overhead. Instead of making each local charge pump (LCP) in a PCM chip larger, we add a global charge pump (GCP)

into the bridge chip. As shown in Figure 7, the GCP resides in the bridge chip and uses a dedicated wire to supply the pumped voltage and write current to each PCM chip. Each bank segment (within a PCM chip) has an analog current controller to choose the write voltage from either the LCP or GCP (but not both). By default, the maximum power that the GCP can provide is set to the same power as one LCP.

While the GCP can provide extra power, the existing power budgets still need to be enforced: (i) the DIMM and chip power budgets must be obeyed and (ii) the DIMM and chip power budgets are not changed by introducing the GCP. In other words, the power that the GCP provides to one chip is actually "borrowed" from other chips.

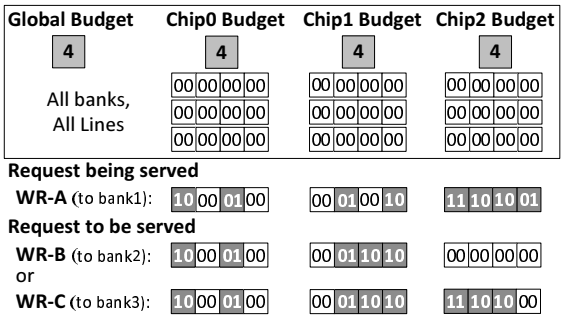


Figure 8: Schedule MLC PCM writes under FPB-GCP (assuming three chips/bank).

Figure 8 illustrates how FPB-GCP works. It has the same assumption as the earlier case in Figure 3. Now, the GCP has 4 power tokens. When WR-A is served, the available tokens are 2/2/0 for PCM chips 0/1/2 respectively. WR-B is chosen to be served next. Since it changes three cells for the 2nd segment (in chip 1), WR-B needs three tokens for chip 1. Given that only two tokens are available on chip 1, the write cannot be served using only the LCP. Thus, the GCP kicks in and injects extra power to write the segment in chip 1. Meanwhile, the LCP on chip 0 is used to write the first segment of WR-B: this segment asks for two tokens, which chip 0 has available.

In FPB-GCP, one segment uses either LCP or GCP, but not both. For example, it may still be impossible to serve WR-C and WR-A simultaneously because the GCP does not have enough tokens. Assume the GCP is used to write the 2nd segment of WR-C. Now, only one GCP token is available. Since WR-C changes three cells in its 3rd segment and chip 2 has no tokens available, the GCP is needed. However, WR-C cannot proceed as there are no enough GCP tokens.

In this example, the GCP might dynamically borrow tokens from chip 1 (has two available tokens). However, due to GCP power efficiency (discussed next), the two LCP tokens from chip 1 may correspond to only one GCP token. Thus, WR-A and WR-C still cannnot be served simultaneously.

4.2. Power efficiency

An important parameter for a charge pump is its power efficiency, i.e., what percentage of input power can be utilized to write cells. Since LCP and GCP use the same CMOS-compatible charge pumps [6, 17], they have the same power efficiency by themselves. However, the wire from the GCP to the write driver is much longer than the wire from the LCP. The pumped voltage and write current from the GCP needs to travel a long distance before it is consumed. While wide wires can be used to reduce wire resistance, the long distance will cause an inevitable power loss. Power loss is common even within one chip, e.g., Oh *et al.* observed around a 10% power loss within a PCM cell array [16]. To compensate for the loss, the GCP needs to output slightly larger current to ensure that the desired current can reach the farthest chip. This indicates a lower effective power efficiency. Given a limited number of pumping stages, the GCP may also need to add extra resistance to provide stable write current for nearby chips. A design alternative is to perform per-chip regulation to compensate different power losses between GCP and each chip, which tends to achieve better power efficiency at the cost of more complicated control logic. In addition, there is an efficiency loss from the pin to the write driver. Since the overall efficiency of the GCP depends on both technology and a combination these factors, the design of a highly power efficient GCP is beyond the scope of this paper.

To evaluate the effectiveness of GCP, we assume that the LCP has a 95% power efficiency, while the GCP has an effective power efficiency in the range [30%, 95%].

4.3. Cell mapping optimization

Due to low GCP power efficiency, the GCP wastes a non-negligible portion of input power. The more frequently the GCP is used, the more energy it wastes. Clearly, when two schemes have the same performance improvement, the one that uses the GCP less is preferred. In this section, we propose cell mapping optimizations to maximize throughput while minimizing GCP usage.

Our analysis shows that GCP usage is proportional to the imbalance of power demands at the chip level because the GCP "borrows" power tokens from the LCP. If all chips had exactly the same power demands, they would use up their power tokens at the same time, which leaves no tokens available to borrow. In practice, the imbalance exists due to memory access characteristics at the application level. For example, studies have shown that the lower-order bits of integer values are more likely to change. To minimize imbalance, and thus, the frequency of using the GCP, we study different mapping schemes that interleave cells across the chip.

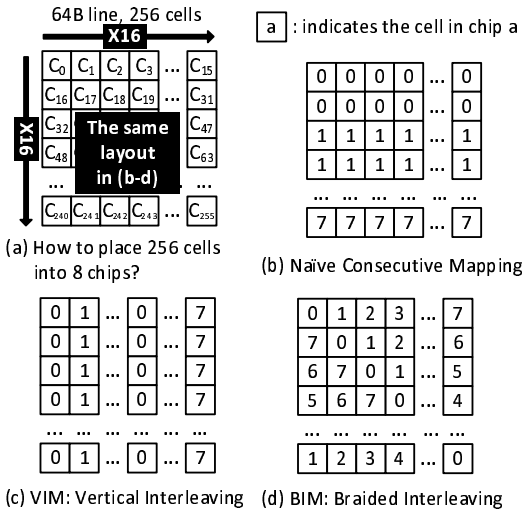


Figure 9: Different cell mapping schemes.

As shown in Figure 9(a), storing one 64B PCM line needs 256 2-bit MLC cells. We put 246 MLC cells into a 16×16 matrix layout

for illustration. The cell position keeps the same in Figure 9(b,c,d). A naïve mapping stores consecutive cells within one chip, e.g., the first 32 cells could be stored in chip 0 (Figure 9(b)).

For floating point (FP) programs, changing a FP value may lead to changing cells in one word (i.e., consecutively 16 logical cells), which incurs a request for more tokens from one chip. To distribute these changes, we propose Vertical Interleaving Mapping (VIM) that maps cells to chips as shown in Figure 9(c). The mapping function can be written as:

$$\mathsf{chip\_index} = \mathsf{cell\_index} \bmod 8 \tag{2}$$

For integer benchmark programs, the lower-orders bits in a word are more likely to change. To further balance cell changes, we propose Braided Interleaving Mapping (BIM) that distributes the lower-order cells from different words to different chips (Figure 9(d)). The mapping function is:

$$\mathsf{chip\_index} = (\mathsf{cell\_index} - \frac{\mathsf{cell\_index}}{16}) \bmod 8 \tag{3}$$

Cell mapping optimization overhead: BIM and VIM are static cell mapping optimization for distributing logic cells into different physical chips at the device level. On the contrary, intra-line wear leveling [31] periodically shifts a logical memory line by several bytes. Our cell mapping optimization is orthogonal to intra-line wear leveling: the inputs of BIM and VIM cell mapping functions are the outputs from intra-line wear leveling. Our schemes map logic MLC cell positions, instead of bit positions, to different chips. We observed that mapping bit positions by separating two consecutive bits in one cell would increase cell changes. BIM and VIM do not affect read procedure in PCM chip, since static cell mapping optimization does change row activation and wordline/bitline structure. Therefore, our schemes do not need any extra read/write dynamic energy. The cell mapping translation logic for 256 cells costs $87ps$ and $49\mu W$ at each access to PCM chip under $45nm$ technology.

5. Experimental methodology

5.1. Baseline configuration

To evaluate the effectiveness of our proposed schemes, we adopted the same simulation framework from [8] and compared our schemes to existing heuristics. The simulator is built as a PIN tool, which is used to collect long memory traces. Since our study focuses on memory subsystem performance and power characteristics, we used a memory trace-driven simulator (instead of a detailed pipeline simulator) to model accesses to and from MLC PCM main memory.

Our simulator faithfully models the entire memory hierarchy, including L1, L2 and DRAM last-level caches, the memory controller, and MLC PCM main memory. Several traces can be combined and interleaved by the simulator to create a multi-programmed workload. The simulator considers cache-to-cache and cache-to-memory bus contention, bank conflicts, and memory bus scheduling constraints. The memory controller gives higher priority to read requests. A write request is scheduled only when there is no read request. When the write queue is full, the memory controller schedules a write burst, which blocks any pending read requests until all the writes in the queue are finished. This strategy was also used by [8]. In addition to the normal bus and chip scheduling policies, writes can only be scheduled when there are enough available power tokens. We also consider the integration of our schemes with write cancellation, write pausing and write truncation.

CPU	8-core, 4GHz, single-issue, in-order
L1 I/D	private, I/D 32KB each/core, 64B line, 2-cycle hit
L2	private, 2MB/core, 4-way LRU, 64B line, write back 2-cycle tag, 5-cycle data hit, 16-cycle CPU to L2
DRAM L3	private, offchip, 32MB/core, 8-way LRU, write back 256B line, 50ns (200-cycle hit), 64-cycle CPU to L3
Memory Controller	onchip, 24-entry R/W queues, MC to bank 64-cycle scheduling reads first, issuing writes when there is no read, issuing write burst when W queue is full [8]
PCM Main Memory	4GB, the same line size as L3, single-rank 8 banks, MLC read 250ns (1000 cycles) RESET: 125ns (500 cycles), 300μA, 1.6V, 480μW SET: 250ns (1000 cycles), T_{off} [9] included, 150μA 1.2V, 90μW [12], MLC Write Model: 2-bit MLC[20, 10] '01': i/F1/F2 = 2/0.375/0.625, 8 iterations on average; '10': i/F1/F2 = 2/0.425/0.675, 6 iterations on average; '00': fixed 1 iteration; '11': fixed 2 iterations

Table 1: Baseline configuration

Our baseline configuration follows past work [10, 20]. There are eight cores in our CMP system. Each core is single-issue, in-order and can be operated at 4GHz. Our trace-driven simulation methodology limits the simulated cores to be in-order. Each core in the baseline has a 32MB private write-back DRAM cache to alleviate pressure on MLC PCM main memory bandwidth. The DRAM cache has a default 256B line size. We also examine 64B and 128B line sizes in a sensitivity study. The detailed parameters can be found in Table 1. The results showed that our techniques can obtain significant improvement on a wide range of baseline configurations.

We consider a main memory with a single 4GB MLC PCM DIMM. The 4GB PCM main memory is divided into 8 banks. A bank spreads across 8 PCM chips. Therefore, 8 banks share 8 PCM chips. The programming current of one chip is supplied by local charge pump.

We use the same DIMM power token number $\mathsf{PT_{DIMM}}$ as past work [8]. Let $\mathsf{E_{LCP}}$ and $\mathsf{E_{GCP}}$ represent the power efficiency of LCP and GCP, respectively. The following formula computes the maximum power tokens $\mathsf{PT_{LCP}}$ that each chip has:

$$\mathsf{PT_{LCP}} = \frac{\mathsf{PT_{DIMM}} \times \mathsf{E_{LCP}}}{8} \tag{4}$$

Assume the GCP borrows $\mathsf{Borrowed_i}$ tokens from each chip ($1 \le i \le 8$ and $0 \le \mathsf{Borrowed_i} \le \mathsf{PT_{LCP}}$). The following formula computes the power tokens that the GCP can provide:

$$\mathsf{PT_{GCP}} = \sum_{i=1}^{8} \frac{\mathsf{Borrowed_i}}{\mathsf{E_{LCP}}} \times \mathsf{E_{GCP}} \tag{5}$$

Thus, clearly, we have:

$$\mathsf{PT_{DIMM}} = \sum_{i=1}^{8} \frac{\mathsf{PT_{LCP}} - \mathsf{Borrowed_i}}{\mathsf{E_{LCP}}} + \frac{\mathsf{PT_{GCP}}}{\mathsf{E_{GCP}}} \tag{6}$$

5.2. Simulated workloads

We modeled a CMP that executes multi-programmed workloads. We chose a subset of programs from the SPEC2006, BioBench, MiBench and STREAM suites to construct workloads that exhibit different memory access characteristics. Table 2 lists the R/W-PKI (Read/Write accesses per thousand instructions) of each workload. We used SimPoint [27] to pickup representative phase. We simulate 1 billion instructions to obtain performance results.

Name	Description	RPKI	WPKI
ast_m	SPEC-CPU2006 (C), 8 C.astar	2.45	1.12
bwa_m	SPEC-CPU2006 (C), 8 C.bwaves	3.59	1.68
lbm_m	SPEC-CPU2006 (C), 8 C.lbm	3.63	1.82
les_m	SPEC-CPU2006 (C), 8 C.leslie3d	2.59	1.29
mcf_m	SPEC-CPU2006 (C), 8 C.mcf	4.74	2.29
xal_m	SPEC-CPU2006 (C), 8 C.xalancbmk	0.08	0.07
mum_m	BioBench (B), 8 B.mummer	10.8	4.16
tig_m	BioBench (B), 8 B.tigr	6.94	0.81
qso_m	MiBench (M), 8 M.qsort	0.51	0.47
cop_m	STREAM (S), 8 S.copy	0.57	0.42
mix_1	2S.add-2C.lbm-2C.xalan-2B.mummer	1.16	0.58
mix_2	2S.scale-2C.mcf-2C.xalan-2C.bwaves	0.94	0.61
mix_3	2S.triad-2B.tigr-2C.xalan-2C.leslie3d	0.96	0.58

Table 2: Simulated applications

For our results, we define *speedup* as:

$$\text{Speedup} = \frac{\text{CPI}_{\text{baseline}}}{\text{CPI}_{\text{tech}}} \quad (7)$$

where $\text{CPI}_{\text{baseline}}$ and CPI_{tech} are the CPIs of the baseline setting and the setting with scheme tech, respectively. This metric is also used by previous closely related research [10, 20].

Write burst: We adopted a write scheduling strategy from [8]. When write queue is 100% full, a write burst postponing all read requests is issued. And it is finished when the write queue is drained to be empty. The percentage of time in write burst of our baseline directly shapes the performance improvement achieved by our schemes. Figure 10 shows the percentage of write burst in the entire application simulation time for baseline. Since most of our simulated benchmarks are write intensity, the average percentage of time in write burst for our baseline is 52.2%, which is a strong motivation to improve heavily power constrained MLC PCM write throughput. Our result on write burst percentage is higher than that in [8] for several reasons: (i) MLC PCM has $\times 8$ long write latency than SLC PCM; (ii) compared to the baseline configuration in [8], the CPU frequency in our baseline is doubled; (iii) larger memory line size and chip level power restriction have more significant negative influence on write throughput than Flip-n-Write [4].

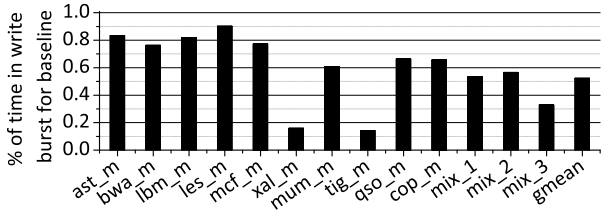


Figure 10: Percentage of execution cycles in write burst for baseline.

6. Experimental results

We implemented and compared the following schemes.

- Ideal: an ideal scheme that has an unlimited power budget.
- DIMM-only: a scheme that enforces only a DIMM power budget (PT_{DIMM}=560) [8]. The chip power is unrestricted.
- DIMM+chip: a scheme that enforces both DIMM and chip power budgets using Hay *et al.*'s technique [8]. To adopt this scheme, an oracle counter is introduced to provide the exact number of chip-level cell changes with no latency overhead. Here, $\text{PT}_{\text{LCP}} = \text{PT}_{\text{DIMM}} \times 0.95 / 8$.
- GCP-CL-E: a scheme that uses only FPB-GCP. The cell mapping, CL, may be NE (naïve mapping), VIM, or BIM. The GCP's power efficiency, E, ranges from 50% to 95% (0.5 to 0.95).
- GCP+IPM: a scheme that uses both FPB-GCP and FPB-IPM. By default, we use GCP-BIM-0.7 for the GCP. Multi-RESET (MR) is also evaluated.

Evaluation order and normalization. In Figure 4, we showed that the performance drop from Ideal to DIMM-only is due to iteration-oblivious budgeting, and the drop from DIMM-only to DIMM+chip is due to the chip power budget. In this section, we aim to restore these performance drops in reverse order. We first evaluate FPB-GCP with the goal to restore performance close to DIMM-only. Next, we add FPB-IPM with the goal to restore performance close to Ideal.

In this section, the speedup values are normalized to DIMM+chip.

6.1. Effectiveness of FPB-GCP

6.1.1. Performance improvement Figure 11 shows IPB-GCP's effectiveness for different GCP power efficiency values. We used the naïve cell mapping (NE) in this experiment. We compared GCP with DIMM-only as IPB-GCP aims to eliminate the chip power budget.

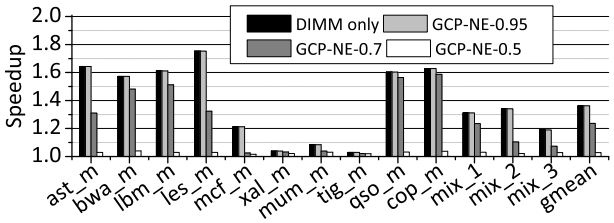


Figure 11: Speedup with different GCP power efficiencies.

From Figure 11, the GCP's power efficiency has a large performance impact. When $\text{E}_{\text{LCP}} = \text{E}_{\text{GCP}}$, GCP-NE-0.95 is 36.3% better than DIMM+chip. We found that GCP-NE-0.95 and DIMM-only have the same performance. The reason is, when $\text{E}_{\text{LCP}} = \text{E}_{\text{GCP}}$, there is no waste to let IPB-GCP borrow tokens from the LCPs (Equation 5).

In practice, the GCP is likely to be less efficient than the LCP. When E_{GCP}=70% (a typical value for an off-chip power supply), GCP-NE-0.7 improves performance by 23.7% over DIMM+chip. However, when the power efficiency is decreased further, its effectiveness diminishes. When E_{GCP}=50%, the GCP cannot help at all: on average, only 2.8% improvement was observed.

In Figure 11, some programs are less sensitive to chip power budget. There are three scenarios. (i) When write operations are intensive, e.g., *mcf* or *mum*, the bottleneck shifts to the DIMM power budget. IPB-GCP often cannot borrow enough tokens to help. (ii) When a program has few writes, e.g., *xal*, the writes have little performance impact. (iii) When a program has many more reads than writes, e.g., *tig*, the performance bottleneck shifts from the writes to the reads such that the chip power budget has a small impact.

6.1.2. Cell mapping optimization Figure 12 compares different cell mapping. In these results, the GCP has practical power efficiency values. When E_{GCP}=70%, VIM and BIM effectively mask the chip power budget; the performance loss versus DIMM-only is only 2% and 1.4%, respectively. VIM and BIM are comparably effective with BIM being slightly better. BIM better balances cell changes when a PCM line stores either FP or integer values.

More importantly, VIM and BIM make FPB-GCP effective for a 50% GCP power efficiency. These advanced cell mappings better

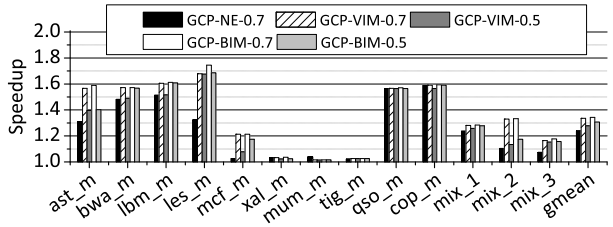


Figure 12: Speedup of cell mapping optimizations.

balance cell changes, which reduces how often the GCP needs to be employed. With fewer requests sent to the GCP, the advanced mappings relax the demands on the highly power-inefficient GCP.

6.1.3. GCP area overhead We next estimated the GCP's area overhead. Since the area of the charge pump is proportional to the maximum power that it can provide, we collected the maximal power tokens requested for GCP under different cell mappings and compared their area overheads.

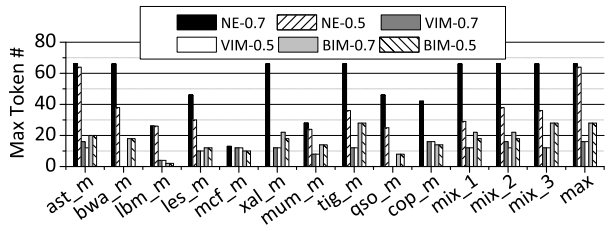


Figure 13: Maximum number of tokens requested by the GCP.

Figure 13 reports the maximum power tokens for each workload when $\mathsf{E}_{\mathsf{GCP}}$ is 70% and 50%. The maximum requested power tokens are 66, 16, and 28 for the naïve mapping, VIM, and BIM, respectively. Interestingly, with VIM, there is no request to the GCP for the *bwa* benchmark, which indicates VIM balanced the cell writes very well across the PCM chips.

Using the maximum power tokens requested, Table 3 estimates the area overheads under different schemes. As discussed in Section 2.2, 2xLocal can also mask the chip power budget. However, this scheme doubles the LCP area in each chip, i.e., 100% overhead. Using the GCP greatly reduces area overhead. For example, with VIM and 70% GCP power efficiency, the GCP overhead is only 4.1% of 2xLocal.

Scheme	Power Tokens	Overhead
Baseline (8 chips)	$70*8=560$	—
2×Local (8 chips)	$140*8=1120$	100%
GCP-NE-0.95	$66/0.95=70$	12.5%
GCP-NE-0.70	$64/0.70=92$	16.4%
GCP-VIM-0.95	$16/0.95=17$	3.1%
GCP-VIM-0.70	$16/0.70=23$	4.1%
GCP-BIM-0.95	$28/0.95=30$	5.4%
GCP-BIM-0.70	$28/0.7=40$	7.1%

Table 3: Charge pump overhead as measured by power tokens

We only compare the charge pump size. FPB-GCP also needs a dedicated pin for each PCM chip to inject the extra power from the GCP. A wire is needed on the DIMM to connect the GCP to each PCM chip as well.

6.1.4. GCP pin and packaging overhead To realize GCP, an extra pin is needed per PCM chip to deliver the high voltage and large current produced by the GCP on the DIMM. This overhead can be justified by the large performance improvement and small GCP size. In addition, the pin overhead is localized to the DIMM, rather between the memory controller and the CPU/PCM chips. Several recent designs use a similar approach. Raghavan *et al.* [23] use additional pins to inject extra power for concurrent computing in embedded systems. A similar external current supply interface has been implemented in a recent PCM chip prototype [5].

The wire overhead between the GCP and PCM chips on the DIMM is negligible. Although write throughput on a DIMM is significantly increased with our techniques, thermal dissipation is also increased from more simultaneous writes, which requires better thermal control.

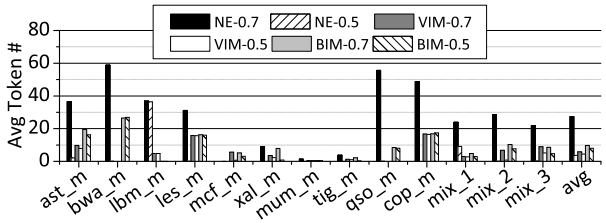


Figure 14: Averge power tokens requested by NE, VIM and BIM.

6.1.5. Minimize wasted energy When different cell mappings have similar performance improvement, the mapping that needs fewer power tokens from the GCP is preferred as it reduces energy waste on the wire. Figure 14 reports the average number of tokens requested per line write from the GCP. VIM and BIM greatly reduce the total number of tokens requested. And thus, on average, VIM and BIM reduce energy waste by 78.5% and 64.4% over the naïve mapping at 70% GCP power efficiency.

6.1.6. BIM overall effectiveness The last experiment considers BIM effectiveness as the GCP's power efficiency is decreased. Figure 15 reports speedup for three typical workloads. BIM helps preserve the performance benefit relative to DIMM+chip with very low GCP power efficiency. For example, in *mix*_1, BIM is still effective, although GCP power efficiency is as low as 20%.

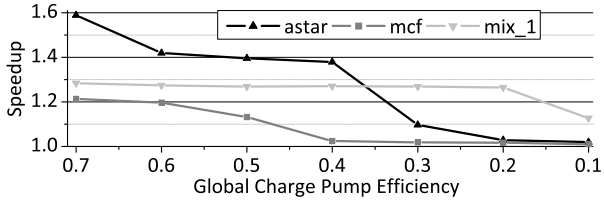


Figure 15: Speedup with BIM as GCP efficiency is decreased.

6.2. Effectiveness of FPB-IPM

6.2.1. Performance improvement We evaluated the effectiveness of FPB-IPM. The goal is, together with FPB-GCP, to make performance close to Ideal. Figure 16 reports the speedup achieved by IPM and Multi-RESET over DIMM+chip. GCP is used with BIM at 70% GCP power efficiency. Figure 16 also shows the performance improvement for GCP power efficiency values of 50% (gm0.5) and 30% (gm0.3).

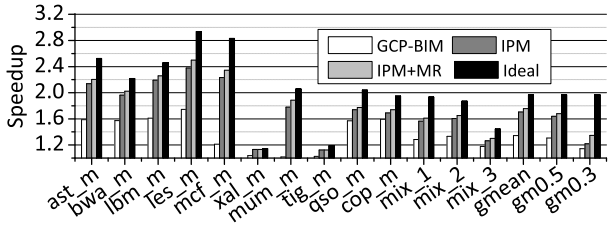


Figure 16: IPM and Multi-RESET Speedup.

On average, IPM improves performance by 26.9% over GCP-BIM. IPM+MR includes Multi-RESET that splits the first RESET iteration of a write into 3 new iterations, when there are not enough power tokens. IPM+MR has a 30.7% performance improvement over GCP-BIM and 75.6% improvement over DIMM+chip. This value is within 12.2% of Ideal, which has no power restrictions.

Also, from Figure 16, the overall performance improvement decreases with decreasing GCP power efficiency (compare gmean, gm0.5 and gm0.3). In addition, the improvement from IPM is stable from 70% GCP efficiency to 50% efficiency but drops at 30%. Multi-RESET tends to be more beneficial as efficiency decreases. For the benchmarks with a large number cell changes and a large WPKI, e.g., *mcf* and *mum*, IPM achieves significant improvements over GCP-BIM, indicating IPM makes better use of DIMM power for these benchmarks.

Intuitively, Multi-RESET increases the overlap among the long SET portions of multiple write requests. With a small available budget that cannot support all RESETs in one write, Multi-RESET adopts a greedy strategy to start a portion of RESETs in one write as early as possible. Without Multi-RESET, the many small pieces of available budget will be wasted for at least the current iteration. By fully utilizing these small available budget fragments, the power consuming RESETs in a write finish in multiple rounds. In this way, the processing time in every write burst can be reduced.

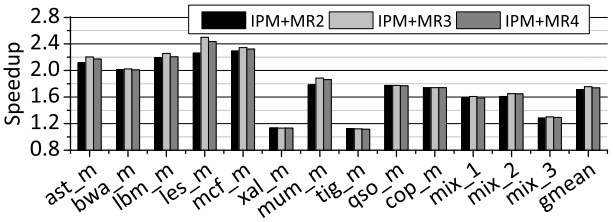


Figure 17: Multi-RESET iteration split limit.

6.2.2. Multi-RESET iteration count Multi-RESET introduces more RESET iterations. In turn, this lowers the maximum power demand but lengthens write latency. We examined how Multi-RESET should split the RESET iteration; i.e., how many new iterations should a single RESET be split into. Figure 17 reports performance when Mutli-RESET splits the first RESET iteration into 2, 3, or 4 new RESET iterations. As shown in Figure 17, the best improvement is achieved for 3 iterations. There is a 2% performance decrease at 4 iterations due to the longer write latency. Thus, we use 3 as the limit when applying Mutli-RESET.

6.3. Throughput improvement

As the performance improvement comes mainly from improved write throughput, we report overall throughput gains in Figure 18.

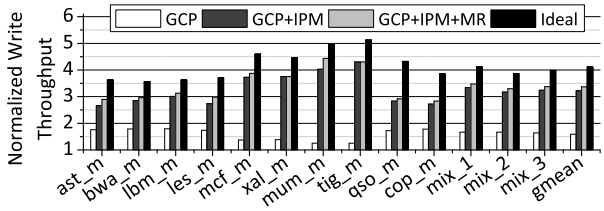


Figure 18: Write throughput improvement.

The results are normalized to DIMM+chip. From Figure 18, FPB achieves around 58.8% throughput improvement from GCP and 3.4× improvement when GCP, IPM and MR are applied. The write throughput obtained by GCP, IPM and MR is smaller than the Ideal (no power restrictions) write throughput by 22%. This throughput gap is due to the tight DIMM level power budget and the large memory line size.

6.4. Design space exploration

To evaluate the effectiveness of our proposed fine-grained power budgeting schemes under different settings, we did experiments in a wide design space with different memory line sizes, last-level cache capacities, number of entries in the write queue, and number of power tokens. We also integrated our FPB schemes with the state-of-the-art designs for MLC PCM: write cancellation, write pausing [20] and write truncation [10]. These methods are orthogonal to power budgeting. In the following design exploration, we use IPM+MR with BIM and E_{GCP}=70%. We abbreviated this combined scheme as FPB.

In the comparison, when studying the sensitivity of parameter X, each bar is normalized to DIMM+chip that has the same X value. Different bars show different X values.

6.4.1. Cache/memory line size Figure 19 compares the performance impact with different memory line sizes. We assume that the MLC PCM memory line size is the same as the last-level cache's line size. For 64B line size, Hay *et al.* observed that the existing DIMM power budget barely meets the demand for eight simultaneous line writes [8]. The improvement that FPB achieves is modest for 64B line size. For large line sizes (or large row buffer sizes), the number of line writes are reduced but each line write changes more cells, which creates contention for the power budget as writes are issued. From Figure 19, FPB achieves a larger improvement with bigger line sizes due to better utilization of the DIMM power budget. On average, FPB has a 41.3%, 61.8% and 75.6% improvement for 64B, 128B and 256B line sizes.

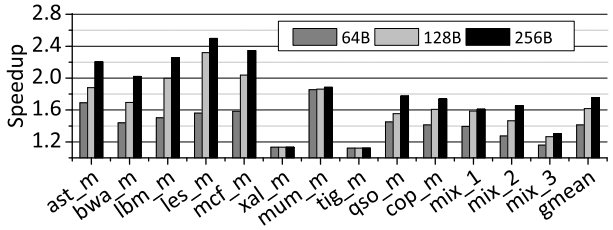


Figure 19: Speedup of FPB for different line sizes.

6.4.2. Last-level cache capacity Figure 20 compares performance for FPB under different last-level cache (LLC) capacities. With a small LLC, e.g., 8MB, there are a large number of memory accesses, which causes the system bottleneck to be main memory bandwidth.

Enforcing the DIMM and chip power budget with DIMM+chip results in even lower memory throughput and performance. On average, FPB achieves 39.9% improvement over DIMM+chip in this setting.

However, as LLC capacity is increased, the number of writes is reduced, yet each line write tends to have more cells to be changed. An improvement in the memory throughput exhibits large performance improvement. On average, FPB achieves 62.1% and 75.6% performance gains for 16MB and 32MB LLC capacities.

We also tried 128MB LLC size per core (1GB LLC for 8 cores). With a large LLC capacity, the offchip read and write traffic in the benchmarks is substantially reduced. Our power management schemes improve performance by 23.4% due to the short write burst time in this case. Most part of performance gain is achieved on streaming benchmarks, such as *qso* and *cop*.

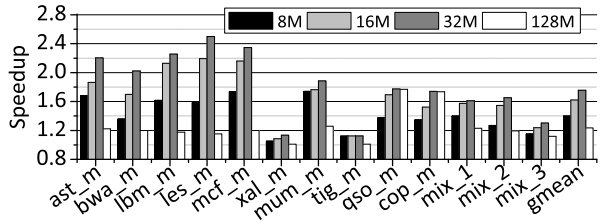


Figure 20: Speedup of FPB for different LLC capacities.

6.4.3. Number of write queue entries Figure 21 shows FPB's effectiveness for varying number of write queue entries. The writes in the queue are flushed when the queue is full. With more entries in the queue, the bursty flush tends to request more power tokens, which is sensitive to write throughput. On average, FPB improves performance by 75.6%, 85.2% and 88.1% for three write queues.

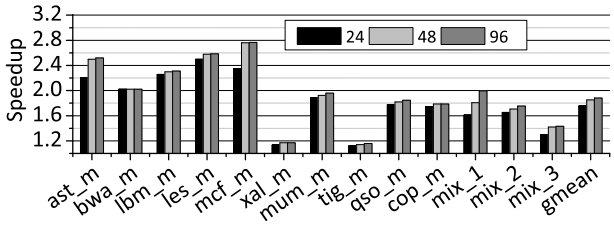


Figure 21: Speedup of FPB for different write queue sizes.

For benchmarks with large WPKI, such as *mum*, FPB has a large speedup. The overall performance improves significantly when the write entry count is increased from 24 to 48. It saturates at 48, and a 96-entry write queue does not exhibit notable improvement over a 48-entry queue.

6.4.4. Number of power tokens Figure 22 shows the performance impact of using 1/8 more or fewer power tokens. We chose this setting to study performance when the overall area change (increase or decrease) is about one LCP size, i.e., all eight chips each increase or decrease by 1/8 size.

From Figure 22, FPB does better with a tighter power budget. This phenomenon is due to FPB better using the power budget than DIMM+chip. If there is an abundant power budget, then wasting some tokens will not have a large performance impact and it is less critical to design advanced power budgeting schemes.

6.4.5. Integrating write pausing and write truncation Write cancellation, write pausing [20], and write truncation [10] are recent effective read latency reduction schemes for MLC PCM. Although they address different issues than FPB, we examined their compatibility with FPB.

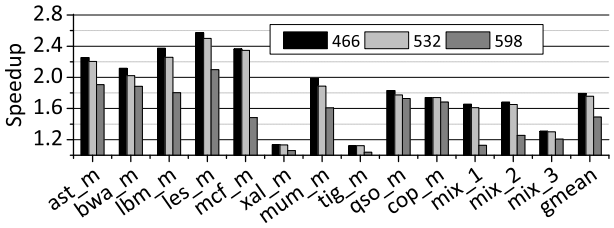


Figure 22: Speedup of FPM for different power token budgets.

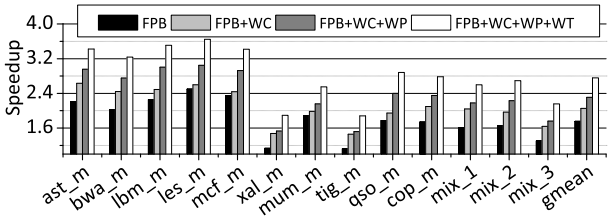


Figure 23: FPB with WC, WP and WT.

Figure 23 shows performance improvement when FPB is integrated with write cancellation (WC), write pausing (WP), and write truncation (WT). As WC needs a large write queue, we increased the entries in the read and write queues to 320 (40 R/W entries per bank, 8 banks). In our experiments, WC is always enabled with WP.

From Figure 23, we observe that FPB, WC, WT, and WT are orthogonal designs that target different performance opportunities. When all these designs are combined, on average, FPB+WP+WT achieves 175.8% improvement over DIMM+chip. This is a gain of 57% over FPB.

However, FPB+WP+WT has a smaller improvement over WP+WT. This happens because WC, WP and WT mitigate the importance of writes on performance, i.e., WC and WP move many writes off the critical path and WT reduces write latency. As discussed, FPB gains performance due to improved write throughput. Thus, when writes are less critical, the performance improvement from FPB is less.

7. Prior art

In addition to the related research discussed in the preceding sections, we discuss other related work in this section.

PCM power management. High write power is known as a major disadvantage of PCM. Schemes have been proposed to change only the cells that need to be changed [12, 31]. Cho *et al.* proposed *Flip-n-Write* that can pack two line writes with the power budget of writing the number of one line cells [4]. It has limited benefit for MLC PCM due to the additional states used in MLC PCM. Hay *et al.* proposed to track/estimate bit changes in the last-level cache and issue write operations as long as the DIMM power budget can be satisfied [8]. These power management schemes focus mainly on SLC PCM. The FPB schemes proposed in this paper address MLC PCM write by exploiting its characteristics.

To address write power in MLC PCM, Joshi *et al.* proposed a novel programming method that decreases write energy and latency by switching between two write algorithms: *single RESET multi-SET* and *single SET multi-RESET* [11]. The latter is often less re-

liable. Wang *et al.* proposed to reduce write energy by adopting different mappings between data values and resistance levels [28].

MLC PCM. MLC PCM can effectively reduce per bit fabrication cost. Schemes have been proposed to address its latency, write energy, and endurance issues [20, 21, 10]. MLC PCM differs from SLC PCM in that it has a non-negligible resistance drift problem. Zhang *et al.* proposed different encoding schemes to mitigate drift [30]. Awasthi *et al.* proposed lightweight scrubing operations to prevent soft errors [1].

Asymmetric write. The RESET and SET operations have asymmetric characteristics in terms of latency and power [24]. For SLC PCM, Qureshi *et al.* proposed to perform SET operations before the memory line is evicted from last-level cache [22]. When a write operation comes, only the short-latency RESET needs to be performed. Applying PRESET on MLC PCM indicates the adoption of *single SET multi-RESET* write scheme, which tends to increase the demand for power tokens.

Industry chip demonstration. Samsung recently demonstrated a 20*nm* PCM chip [5] that splits the conventional local charge pump into sub-pumps to reduce voltage drop and current consumption along long wires. The prototype adds an external high voltage and current supplement interface. Our scheme is orthogonal to this design as IPM can be applied at the sub-pump level, while GCP can reuse the current supplement interface.

8. Conclusions and Future Work

In this paper, we proposed FPB, fine-grained power budgeting, that applies two new power management strategies: FPB-IPM enables iteration power management to reclaim unused power tokens as early as possible at the DIMM level and FPB-GCP uses a global charge pump to mitigate power restrictions at the chip level. Our experimental results showed that FPB is effective and robust for a broad range of MLC PCM settings. On average, FPB improves performance by 76% and write throughput by 3.4× over previous power management technique [8].

In this paper, we used a range of power efficiencies to accommodate different device types and their associated parameters. For a more detailed evaluation and low level understanding of the circuit behavior, our future work will consider the circuit level design and conduct SPICE simulations.

9. Acknowledgments

We thank the anonymous reviewers for their constructive suggestions, and Prof. Moinuddin K. Qureshi for sheparding the paper. We also acknowledge the support from PCM@Pitt research group. This research is supported partially by National Science Foundation grants CNS CAREER-0747242 and CNS-1012070.

References

[1] M. Awasthi, *et al.*, "Efficient Scrub Mechanisms for Error-Prone Emerging Memories," in *HPCA*, 2012.

[2] F. Bedeschi, *et al.*, "A Bipolar-Selected Phase Change Memory Featuring Multi-Level Cell Storage," *JSSC*, 2009.

[3] M. Boniardi, *et al.*, "Impact of Material Composition on the Write Performance of Phase-Change Memory Device," in *IMW*, 2010.

[4] S. Cho and H. Lee, "Flip-N-Write: A Simple Deterministic Technique to Improve PRAM Write Performance, Energy and Endurance," in *MICRO*, 2009.

[5] Y. Choi, *et al.*, "A 20nm 1.8V 8Gb PRAM with 40MB/s program bandwidth," in *ISSCC*, 2012.

[6] J. Dickson, "On-chip High Voltage Generation in NMOS Integrated Circuits using an Improved Voltage Multiplier Technique," *JSSC*, 1976.

[7] K. Fang, *et al.*, "Memory Architecture for Integrating Emerging Memory Technologies," in *PACT*, 2011.

[8] A. Hay, *et al.*, "Preventing PCM Banks from Seizing Too Much Power," in *MICRO*, 2011.

[9] Y. Hwang, *et al.*, "MLC PRAM with SLC write-speed and robust read scheme," in *VLSIT*, 2010.

[10] L. Jiang, *et al.*, "Improving Write Operations in MLC Phase Change Memory," in *HPCA*, 2012.

[11] M. Joshi, *et al.*, "Mercury: A Fast and Energy-Efficient Multi-level Cell based Phase Change Memory System," in *HPCA*, 2011.

[12] B. C. Lee, *et al.*, "Architecting Phase Change Memory as a Scalable DRAM Alternative," in *ISCA*, 2009.

[13] K. T. Malladi, *et al.*, "Towards Energy-Proportional Datacenter Memory with Mobile DRAM," in *ISCA*, 2012.

[14] D. Mantegazza, *et al.*, "Statistical Analysis and Modeling of Programming and Retention in PCM Arrays," in *IEDM*, 2007.

[15] T. Nirschl, *et al.*, "Write Strategies for 2 and 4-bit Multi-Level Phase-Change Memory," in *IEDM*, 2007.

[16] H. Oh, *et al.*, "Enhanced Write Performance of a 64-Mb Phase-Change Random Access Memory," *JSSC*, 2006.

[17] G. Palumbo and D. Pappalardo, "Charge Pump Circuits: An Overview on Design Strategies and Topologies," *IEEE Circuits and Devices Magazine*, 2010.

[18] M. K. Qureshi, *et al.*, "Enhancing Lifetime and Security of PCM-based Main Memory with Start-Gap Wear Leveling," in *MICRO*, 2009.

[19] M. K. Qureshi, *et al.*, "Scalable High Performance Main Memory System Using Phase-Change Memory Technology," in *ISCA*, 2009.

[20] M. K. Qureshi, *et al.*, "Improving Read Performance of Phase Change Memories via Write Cancellation and Write Pausing," in *HPCA*, 2010.

[21] M. K. Qureshi, *et al.*, "Morphable Memory System: A Robust Architecture for Exploiting Multi-Level Phase Change Memories," in *ISCA*, 2010.

[22] M. K. Qureshi, *et al.*, "PreSET: Improving Read Write Performance of Phase Change Memories by Exploiting Asymmetry in Write Times," in *ISCA*, 2012.

[23] A. Raghavan, *et al.*, "Computational Sprinting," in *HPCA*, 2012.

[24] S. Raoux, *et al.*, "Phase-change Random Access Memory: A Scalable Technology," *IBM J. RES. & DEV.*, 2008.

[25] S. Schechter, *et al.*, "Use ECP, not ECC, for Hard Failures in Resistive Memories," in *ISCA*, 2010.

[26] N. H. Seong, *et al.*, "Security Refresh: Prevent Malicious Wear-out and Increase Durability for Phase-change Memory with Dynamically Randomized Address Mapping," in *ISCA*, 2010.

[27] T. Sherwood, *et al.*, "Automatically Characterizing Large Scale Program Behavior," in *ASPLOS*, 2002.

[28] J. Wang, *et al.*, "Energy-efficient Multi-Level Cell Phase-Change Memory System with Data Encoding," in *ICCD*, 2011.

[29] D. H. Yoon, *et al.*, "BOOM: Enabling Mobile Memory Based Low-Power Server DIMMs," in *ISCA*, 2012.

[30] W. Zhang and T. Li, "Helmet: A Resistance Drift Resilient Architecture for Multi-level Cell Phase Change Memory System," in *DSN*, 2011.

[31] P. Zhou, *et al.*, "A Durable and Energy Efficient Main Memory Using Phase Change Memory Technology," in *ISCA*, 2009.

2012 IEEE/ACM 45th Annual International Symposium on Microarchitecture

Leveraging Heterogeneity in DRAM Main Memories to Accelerate Critical Word Access *

Niladrish Chatterjee‡ Manjunath Shevgoor‡ Rajeev Balasubramonian‡ Al Davis‡
Zhen Fang§† Ramesh Illikkal∞ Ravi Iyer∞

‡University of Utah
{nil,shevgoor,rajeev,ald}@cs.utah.edu

§Nvidia Corporation
zfang@nvidia.com

∞Intel Labs
{ramesh.g.illikkal,ravishankar.iyer}@intel.com

Abstract

The DRAM main memory system in modern servers is largely homogeneous. In recent years, DRAM manufacturers have produced chips with vastly differing latency and energy characteristics. This provides the opportunity to build a heterogeneous main memory system where different parts of the address space can yield different latencies and energy per access. The limited prior work in this area has explored smart placement of pages with high activities. In this paper, we propose a novel alternative to exploit DRAM heterogeneity. We observe that the critical word in a cache line can be easily recognized beforehand and placed in a low-latency region of the main memory. Other non-critical words of the cache line can be placed in a low-energy region. We design an architecture that has low complexity and that can accelerate the transfer of the critical word by tens of cycles. For our benchmark suite, we show an average performance improvement of 12.9% and an accompanying memory energy reduction of 15%.

1. Introduction

The main memory system is a significant performance and energy bottleneck in modern high-performance platforms. Several trends threaten to amplify the memory bottleneck. It is increasingly evident that overall performance/cost and performance/watt are improved by using simpler cores [10]. This shifts the energy bottleneck from processor to memory. Simpler cores offer increased core count per die, placing greater pressure on memory bandwidth. Simpler cores also tend to be less effective in hiding memory latency. There is also a demand for higher memory capacities because of the data-intensive workloads that execute on many datacenter platforms. For example, the RAMCloud project argues that disks are incapable of fulfilling high query rates in future datacenters and disks must be replaced by large DRAM memories accessed via a network [31]. There is also a push towards in-memory databases within the database community. This trend will also increase the contribution of the memory system to overall system energy and performance.

To date, the main memory system has been largely homogeneous; the DIMMs that are employed in a server have uniform timing parameters and only exhibit minor variations. But it is well known that all data elements do not have a uniform impact on overall performance. Some data elements are more *critical* to performance than others. This argues for having a heterogeneous main memory design where critical words or cache lines are placed in low-latency portions of memory and non-critical elements are placed in low-energy portions. Little prior work has explored the use of different types of DRAM chips to build and exploit a heterogeneous memory system. This paper takes an important step in uncovering the potential of such a heterogeneous DRAM memory system.

The DRAM industry already produces chips with varying properties. Micron offers a Reduced Latency DRAM (RLDRAM) product that offers lower latency and lower capacity, and is targeted at high performance routers, switches, and network processing [7]. Micron also offers a Low Power DRAM (LPDRAM) product that offers lower energy and longer latencies and that has typically been employed in the mobile market segment [6]. Our work explores innovations that can exploit a main memory that includes regular DDR chips as well as RLDRAM and LPDRAM chips. To reduce complexity, we do not integrate different types of DRAM chips on the same DIMM or even the same channel. We assume that each DRAM chip type has its own DIMM and can be accessed via its own channel and memory controller. We propose a *critical-word-first (CWF)* optimization that organizes a single cache line across multiple channels. We observe that the first word in a cache line is typically most critical and benefits from being placed in a low-latency channel/DIMM. Other non-critical words in the cache line can be placed in a low-energy channel/DIMM, thus improving performance *and* reducing energy. The phenomenon of critical word regularity was first reported by Gieske [17] and was exploited in the same work to optimize accesses to the on-chip SRAM caches. Our results show that the CWF optimization yields an average 12.9% performance improvement and a 15% memory energy reduction, while incurring low implementation overheads.

Modern memory systems already incorporate a CWF optimization where the critical word is returned before the other words when servicing a cache line request (p. 372 [19], [13]). However, this only prioritizes the critical word by a few CPU cycles, while not yielding energy efficiency for the non-critical words. We show that our CWF optimization prioritizes the critical word by 70 CPU cycles on average because the critical word has its own independent channel with lower queuing delays, *and* it allows non-critical words to be placed in low-energy DRAM chips. It is therefore a novel and significant advancement over the state-of-the-art.

2. Background

In a modern server, the main memory system is comprised of multiple channels, each having one or more Dual Inline Memory Modules (DIMMs). Each DIMM consists of several DRAM chips which are grouped into ranks. All DRAM chips in a rank work in unison to service a cache line request. Each rank is itself partitioned into many banks and all these banks may be in operation at the same time, each servicing a different request, subject to various timing constraints. Ranks and banks on a single channel enable memory-level parallelism, but the accesses are finally serialized on the channel because

*This work was supported in parts by NSF grants CCF-0811249, CCF-0916436, NSF CAREER award CCF-0545959, and the University of Utah.

†Work done while at Intel Labs.

1072-4451/12 $26.00 © 2012 IEEE
DOI 10.1109/MICRO.2012.11

IEEE computer society

the data transfer has to utilize this shared resource. Each bank in a DRAM chip consists of 2D arrays of DRAM cells and a row buffer which stores the most recently accessed row of the bank. An activate operation fetches data from an entire row (on the order of 8 KB) into the row buffer. A column-read command transfers a cache line from the row buffer to the processor. If a subsequent access is to the same row, it yields a row buffer hit and can be completed at a lower delay and energy cost. Before activating a new row, a precharge operation must be performed. A close-page policy precharges a bank after every column-read command to prepare the bank for access to a new row. An open-page policy keeps the row open to service row buffer hits and precharges the bank only when a new row is encountered. We next examine the specific properties of different flavors of DRAM chips that can be used to construct memory systems.

2.1. DDR3

The most widely used DRAM variant in mid–to high-end computing is the Dual-Data-Rate (DDR3) module. DDR3 DRAM chips attempt to strike a balance between performance and density and are produced in very high volumes for desktop, laptop, and server market segments. We model a standard x8 (data output width of 8 bits), 800 MHz DDR3 part that has a pin data bandwidth of 1600MBps [28] and a capacity of 2Gb. Each DRAM chip has an 18 bit address and a 5 bit command bus. The address is provided in two cycles (row and column). Two primary determinants of the possible throughput of a DRAM chip are its bank count and bank-turnaround time. While a larger number of banks promotes better concurrency, the bank–turnaround time (tRC) determines the minimum time that needs to elapse before a new row in the bank can be activated. To maximize density, DDR3 chips are seldom designed with more than 8 banks, and DDR3 bank-turnaround is about 50 ns.

2.2. LPDRAM

Low Power DRAM or mobile DRAM chips are used as an alternative to DDR3 chips in mobile devices. Designed for low-power usage, LPDDR2 [29] has lower per pin bandwidth due to the lower operating frequency (400 MHz). The core densities and bank counts remain the same across LPDDR2 and DDR3. Due to lower operating voltages, the core latencies are increased by a small fraction leading to a bank-turnaround time of 60 ns. The lower voltages allow low active power consumption; the output drivers are also designed to consume less power. In addition to regular low-power modes, LPDDR2 supports many additional low-power modes like Temperature Compensated Self-Refresh (TCSR), Partial Array Self-Refresh (PASR), and Deep Power Down (DPD) modes, all suited for use in a mobile environment. While current LPDDR2 modules employ a 10-bit address/command bus that is dual-pumped, future low-power DRAM chips could employ DDR3-style 23-bit address/command buses.

2.3. RLDRAM

Reduced Latency DRAM (RLDRAM3 [30]) was designed as a deterministic latency DRAM module for use in high-speed applications such as network controllers [40]. While the pin-bandwidth of RLDRAM3 is comparable to DDR3, its core latencies are extremely small, due to the use of many small arrays. This sacrifice in density is justified by a bank-turnaround time (tRC) of 10–15 ns compared to 50 ns for DDR3, and a higher bank count (16 as opposed to 8 for DDR3). The maximum capacity offered by an RLDRAM3 chip currently (576 Mb) is several times less than the capacity of a DDR3 chip –however Micron roadmaps [1] show plans for 1Gb and 2Gb parts in the near future. In DDR3 devices, to limit the current draw, a timing window tFAW is defined, during which only 4 bank activations can be issued. RLDRAM does not have any such restrictions, favoring low latency over peak power guarantees. RLDRAM uses SRAM-style addressing –the entire address is provided with a single READ or WRITE command, instead of separate RAS and CAS commands (although, this could be modified in future RLDRAM chips). After a read or a write, the bank gets precharged automatically. Thus, effectively, an RLDRAM chip can only operate with a close-page policy.

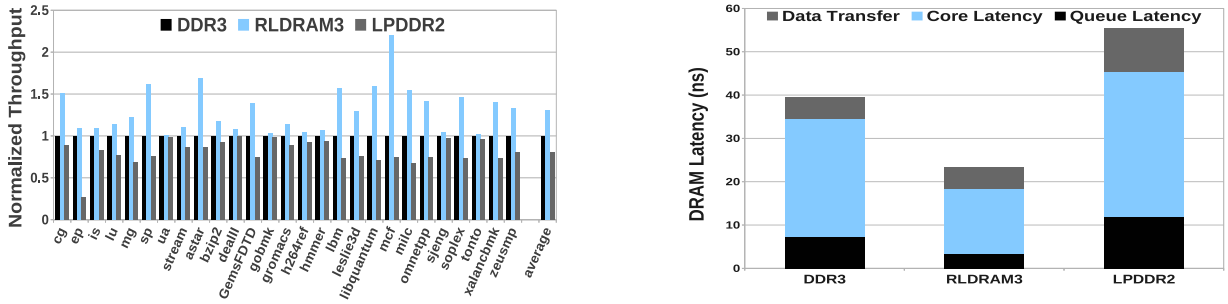
3. Motivational Data

Most high-end servers employ DDR3 parts because they provide the best combination of memory capacity and high performance. If capacity constraints are removed, the use of RLDRAM3 chip latencies can lead to a dramatic improvement in performance. Likewise, if DDR3 parts are replaced with LPDDR2 parts, performance is worsened, but memory energy is reduced. The graph in Figure 1.(a) shows the sensitivity of applications to these different DRAM properties, while still presenting a homogeneous main memory system. We show results for an 8-core system running programs from SPEC2k6 and NAS Parallel Benchmark (NPB) suites. We show results for homogeneous memory systems where the chips are all either RLDRAM3, DDR3, or LPDDR2. More methodology details are in Section 5.

RLDRAM3 outperforms DDR3 by 31% across the simulated workloads (Figure 1a) while LPDDR2 suffers a 13% reduction in throughput. The average main memory access time of RLDRAM3 is about 43% lower than that of DDR3. The breakdown of the latencies in Figure 1b shows that each memory read request spends substantially less time in the queue in a RLDRAM3 system (*Queue Latency*) compared to DDR3, and also requires less time to be serviced from the memory array (*Core Latency*). The low bank-turnaround time of RLDRAM3 (10ns, compared to 50ns for DDR3) leads to the low queuing delay. The lack of write-to-read turnaround delays, as well as the higher bank counts of RLDRAM3, also allow it to provide 24% higher sustained bandwidth compared to DDR3, even though the pin bandwidth of the RLDRAM3 system is the same as that of the DDR3 baseline. On the other hand, the LPDDR2 latency is 41% higher than DDR3. The higher latency of LPDDR2 is due to the higher bank-turnaround time, the slower arrays (due to lower voltages) as well as the lower data bus frequency.

However, RLDRAM3 offers much lower capacity. It is well–known that many applications require large amounts of main memory space [21, 31, 36, 42] and frequent page faults can lead to severe drop in performance –for example, Qureshi et al. quantify the page fault rates seen in several representative benchmarks as a function of DRAM capacity [36]. Thus, an RLDRAM3 system that has the same silicon area as a DDR3 system will have much lower capacity and potentially lower performance for applications with large memory footprints.

While RLDRAM3 is a clear winner in terms of access latencies, it incurs a power overhead. Figure 2 shows the power consumption of the 3 DRAM variants for different bus utilization values. Section 5 outlines the power calculation methodology. We see that at low utilization, the RLDRAM3 power consumption is much higher (due to the high background power), while at higher activity scenarios,



(a) Performance

(b) DRAM Latency Breakdown

Figure 1: Sensitivity of Applications to Different DRAM Flavors.

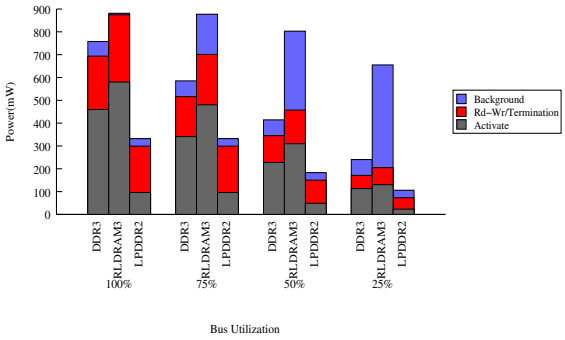


Figure 2: Power vs Bus Utilization(the RLDRAM3 part has a capacity of 512Mb while the DDR3 and the LPDDR2 parts have capacities of 2Gb)

the power consumptions are more comparable. The high power consumption of RLDRAM3 stems from its high background consumption. This is in spite of the fact that the RLDRAM3 part supports lower capacity. LPDDR2, by virtue of its lower operating voltage and frequency, as well as enhanced low-power modes, can yield substantial power savings over DDR3.

Thus, a single memory type cannot yield a design point that is favorable on all metrics. This motivates a heterogeneous memory architecture where selective use of different memory types can help improve performance, energy, and cost.

4. Exploiting Main Memory Heterogeneity

Instead of implementing a homogeneous memory system with DDR3 DIMMs, we implement a memory system where some channels access LPDDR2 DIMMs and some channels access RLDRAM3 DIMMs. We attempt this in a manner that improves performance and energy, at acceptable cost. The design of a heterogeneous DRAM main memory system is a new topic and even some of the basic implementation questions have not been addressed. Therefore, in Section 4.1, we start by describing one design from prior work, explain why a simpler design approach is required, and finally describe a heterogeneous memory architecture that is viable. Section 4.2 then integrates a critical word first optimization with this heterogeneous main memory.

4.1. Building a Heterogeneous Main Memory

Prior Work. Prior work attempted a heterogeneous DRAM design where three types of DIMMs were connected to a single channel [35]. The authors assume that one DIMM is optimized for low latency by using narrow arrays, a wider address bus, and small prefetch buffers. A second DIMM is optimized for low power by using LPDRAM techniques (partial array self refresh and temperature compensated self refresh), fewer banks, narrower banks, small prefetch buffers, and low voltage. The third DIMM is regular DDR and offers high bandwidth by using many wide banks. With an off-line profiling process, applications are classified as either latency or bandwidth sensitive, or a candidate for power optimizations. Accordingly, pages are placed in the low-latency or low-power DIMM, or striped across all three DIMMs if they are bandwidth-sensitive. The authors argue that such a design is meaningful because optimizing a DRAM device for one of the three properties (latency, energy, bandwidth) usually compromises the other two. While this prior work took an excellent first step in an important area, some implementation issues were not addressed. Some of the significant roadblocks that need to be overcome to successfully institute a heterogeneous memory system are listed below:

- Operating different types of DRAM on the same channel is extremely complex (if not impossible) because of the different operating frequencies and voltages as well as different commands and addressing modes.
- The complexity of the memory controller is extremely high as it has to simultaneously handle the different command timing requirements and states of the three kinds of DIMMs. Besides, the RLDRAM command set is completely different from that of DDR3 or LPDRAM.
- As we shall see in Section 4.1, using LPDRAM in a server-like setting requires some modifications to the interface. This introduces a few power overheads. The methodology in [35] assumes different latency and power properties for different DRAM chips while assuming no impact on the DRAM I/O interface, area or power.
- RLDRAM is power hungry and must be used sparingly in the design – simply provisioning one-third of the main memory with RLDRAM as done in [35], will lead to significant increase in memory system power consumption.

In this section, we first address the above issues to develop a scalable, cost-effective heterogeneous memory design that is implementable.

A Focus on Two Types of DIMMs. Unlike the prior work that integrates three types of DIMMs in a heterogeneous main memory (latency-optimized, power-optimized, and bandwidth-optimized), our design only integrates two types of DIMMs (performance-optimized and power-optimized). An evaluation of existing chips shows that a DIMM made with RLDRAM chips can not only offer lower latency, but also higher bandwidth. Compared to a DDR chip, RLDRAM has the same peak frequency, lower gap between successive requests (t_{RC}), and twice as many banks. Hence, we see little value in creating separate DRAM modules for latency and band-

width. The memory controller complexity is also increased by striping some pages across 3 DIMMs (for high bandwidth [35]) and some pages across a single DIMM. Therefore, in our work, we employ a high-performance DRAM module that is built with RLDRAM devices and that offers low latency and high bandwidth. We employ a second type of DRAM module that is built with LPDRAM devices and that has lower bandwidth, higher latency, and lower power. A third type of DRAM module can also be built out of regular DDR3 devices that represents an intermediate design point in terms of performance and power. To reduce cost and complexity, we implement DIMMs that are themselves homogeneous, i.e., a DIMM only has chips of one type.

Homogeneous DIMMs on a Channel. In the proposed system, each DIMM type has its own dedicated channel. The memory controller would have high complexity if a single channel controlled DIMMs of different types. In all of our discussions, for simplicity, we assume that only 1 or 2 DIMMs of the same type are attached to each channel. This is consistent with projections that future DDR generations will only support 1-2 DIMMs per high-speed channel. However, our proposals are equally applicable to configurations that can support more DIMMs or ranks per channel. Modern high-performance processors implement multiple memory channels. Our heterogeneous memory system is built by designating a subset of these channels as performance-optimized channels and a subset as power-optimized channels. In our designs, our performance-optimized DIMM/channel is representative of state-of-the-art RLDRAM technology and operates at high frequency. Our power-optimized DIMM/channel uses LPDRAM chips at lower frequency, an open-row policy (that is known to minimize energy), and an aggressive sleep-transition policy.

Adapting LPDRAM and RLDRAM for Server Memories. In our work, we rely on existing DRAM chips and do not introduce extensive chip optimizations of our own. We therefore restrict ourselves to modeling DRAM chips that closely resemble regular DDR3, LPDRAM, and RLDRAM properties. We allow a few modifications that can be easily integrated into future generations of these chips. For example, we assume that x8 and x9 RLDRAM chips can be supported (only x18 and x36 are available today) and that LPDRAM chips support ODT and DLL.

As described in Section 2, RLDRAM parts are designed for low latency response as well as high sustained bandwidth. In such environments, RLDRAM chips are soldered onto the motherboard, communicating one-on-one with the processor. LPDRAM modules, used in the embedded domain, are also soldered onto the system PCB. This is in contrast to a regular server/desktop environment where individual DDR chips form a DIMM, which is then plugged onto the DRAM channel on the motherboard. While using LPDRAM parts in a DIMM form factor, it is important that additional features be implemented to ensure signal timing and integrity, particularly when high-speed signalling is employed. RLDRAM parts already employ these features.

ODT : An important feature present in modern DDR memory is on-die-termination (ODT) for signals. The purpose of ODT is to match the terminating impedance on the transmission die to the impedance of the transmission line. A mismatch in the terminating impedance leads to signal reflections which act as a source of noise on the memory channel. RLDRAM devices, by virtue of their high-speed operating environments, already incorporate ODT for signal integrity. LPDRAM devices, typically operated in power-sensitive systems, use lower frequencies, and hence, LPDDR2 is not equipped with ODT. However, LPDDR3 chips will incorporate ODT resistors to increase signal integrity in high-frequency environments [32]. ODT resistors can be incorporated in the LPDDR2 chips with very little area overhead as they reside near the data pins [43]. The main overhead of introducing the termination resistors is the increase in static power which we model in the same way as in a DDR3 chip.

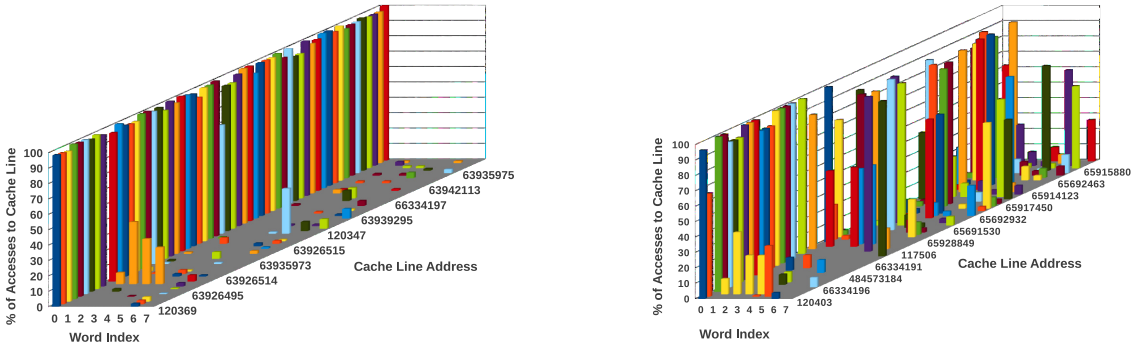
DLL : Modern DDRx SDRAM devices utilize an on-chip DLL (delay-locked loop) to synchronize the data signals and the source-synchronous clocking reference signal DQS with the global clock used by the memory controller. The data sent out on the data bus by a DIMM is synchronous with the DQS signal which in turn is used by the memory controller to sample the data. To make sure that the skew between the global clock fed to the DRAM chip and the DQS signal seen by the memory controller is not catastrophic, the on-chip DLL delays the DQS signal (and the data signals) enough so as to put them in phase with the global clock [19]. DDRx devices as well as RLDRAM 3 devices already use DLLs for this purpose while LPDRAM chips, due to their power constraints, do not have any such feature. We assume that DLLs can be integrated into the I/O circuitry of the LPDRAM chips at a power cost. In Section 5 we describe how we account for this increased power cost in our simulations.

Silicon Cost: It is well known that changes to a DRAM chip must be very sensitive to area and cost. A recent paper [43] pointed out that changes made to a DRAM chip are most disruptive to chip density when they are instituted in the bit-line sense-amplifier stripe, followed by the local word-line driver stripe, then in the column logic, and finally in the center stripe. To implement the DLLs and the ODTs, we need to introduce and modify circuits close to the I/O pads near the center stripe – we can therefore localize the changes to the least sensitive portion of the DRAM chip. The inclusion of ODTs in future generation LPDDRs (LPDDR3 [32]) demonstrates that adapting LPDRAM for higher frequency and long channel environments is indeed commercially viable.

We have thus instituted a low-complexity heterogeneous main memory by introducing the following features: (1) the use of only two types of DIMMs (performance-optimized vs. power-optimized) in one design, (2) the use of only one type of DIMM per channel, (3) the integration of DLL and ODT in LPDRAM chips to make them server-class. It should also be noted that the use of heterogeneity need not be exposed to application developers and does not pose programmer complexity. Similar to the non-uniformity introduced by row buffer hits/misses, memory latencies will differ based on whether heterogeneity is successfully exploited or not.

4.2. Accelerating Critical Words

4.2.1. Motivation. The few prior efforts [35, 37] that have considered non-uniform or heterogeneous memories have attempted to classify data at the granularity of cache lines or OS pages. Instead, we focus on another data criticality opportunity that has not been previously leveraged by heterogeneous memories. A cache line is fetched from memory when a given CPU word request results in on-chip cache misses. It is well known that the word requested by the CPU is more critical than the other words in that cache line. When transferring the cache line, it is re-ordered so that the critical word is sent before the other words, thus reducing access latency by a few cycles [19, 18]. We argue that the transfer of the critical word can



(a) leslie3d

(b) mcf

Figure 3: Distribution of critical words in highly accessed cache-lines

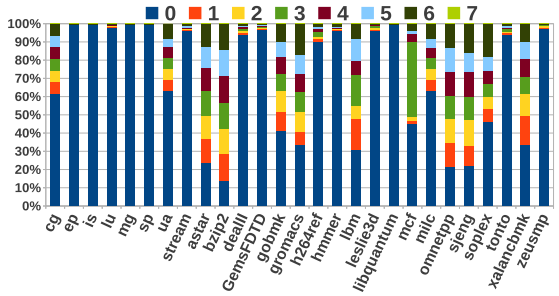


Figure 4: Distribution of Critical Words

be improved by tens of cycles if it is placed on a separate DIMM with lower device latencies and lower queuing delays. Since the other non-critical words in the cache line are not required urgently, they can be placed in low-power DRAM modules without incurring a performance penalty.

For such an optimization it is necessary that the critical word in a cache line remain fairly constant over a long period of time. The graphs in Figure 3 demonstrate such critical word regularity. We monitor DRAM accesses over a billion cycle period to identify the criticality of words in the cache. In Figure 3, we plot the percentage of accesses to different words, of the most accessed cache lines for two applications. The graphs show that for most cache lines, some words are more critical than others. For leslie3d (Figure 3.a), the most accessed critical word is word 0. On the other hand, in mcf (Figure 3.b), other words (i.e., 1 thru 7) are often the most frequently accessed critical words in the cache line. But clearly, in both cases, we see that within a cache line, there is a well-defined bias towards one or two words. We see the same pattern for all the simulated benchmarks.

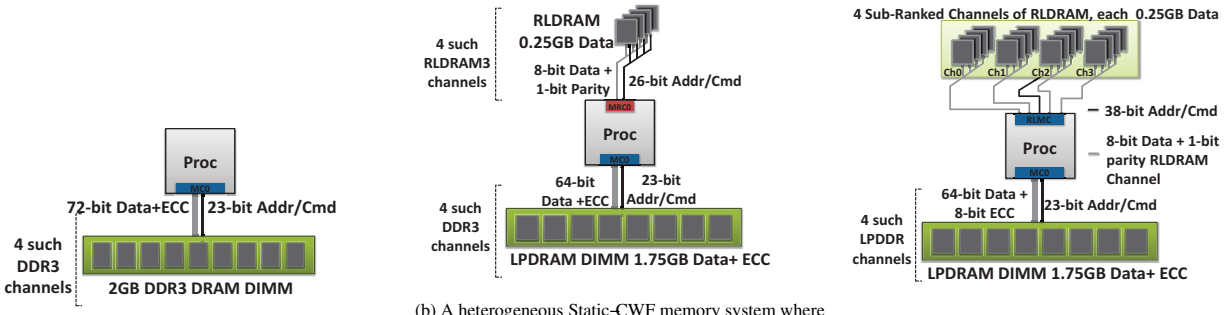
A critical word first (CWF) optimization is facilitated if the critical word for a cache line remains unchanged over the execution of a program. If every cache line has a different critical word, more book-keeping would be required to track the identity of the critical word. The implementation is simplified if the same word is always the critical word for all cache lines and is placed in a low-latency DIMM.

Figure 4 shows the distribution of critical words for our benchmark suite. For 21 of 27 programs, of the 8 words in a cache line, word-0 is critical in more than 50% of all cache line fetches. In 6 programs, there is no well-defined bias and all words have roughly an equal probability of being the critical word. The Appendix includes a description explaining these biases for a variety of programs. In summary, if the program is performing a sequential scan across an array of data, early words in a cache line have a higher probability of being critical than later words in the cache line, especially if the stride is small and the word alignment is favorable. Pointer chasing codes tend to exhibit less bias.

4.2.2. Data Placement. On average, across the entire suite, word-0 is critical for 67% of all cache line fetches. To keep the design simple, we therefore assume a static architecture where word-0 is always placed in a low-latency DIMM and words 1-7 are placed in a low-power DIMM. We later also describe a more complex and flexible data organization.

In our baseline system (Figure 5a), an entire cache line plus SECDED ECC is placed on a single DDR3 DIMM and striped across 9 chips, where each chip is x8 wide. The proposed design for a single channel is shown in Figure 5b. The 9-chip 72-wide DDR3 rank is replaced by an 8-chip 64-wide LPDRAM rank. The 8 chips store words 1-7 and the ECC codes for each cache line. Word-0 for each cache line is now stored in a separate low-latency DIMM (rank) made of RLDRAM chips, and controlled by its own independent channel and memory controller. It is important for the RLDRAM DIMM to have its own independent controller. Since the RLDRAM DIMM has lower queuing delays, the request for word-0 from the RLDRAM DIMM can be issued many cycles before the request for words 1-7 from the LPDRAM DIMM gets issued. This allows the critical word to arrive tens of cycles before the rest of the cache line, having an impact much higher than other CWF optimizations that have been considered by processors [19, 18]. Since RLDRAM has lower density, we assume that 4 RLDRAM chips are required to support the same capacity as one DDR3 or LPDRAM chip. In order to remain processor-pin-neutral, the RLDRAM channel is assumed to only be 8 bits wide (plus a parity bit, explained shortly). We have thus replaced a single conventional 72-bit channel/DIMM with a 64-bit low-energy channel/DIMM and 9-bit low-latency channel/DIMM. The 9 DDR chips that made up a rank have now been replaced by 8 low-power LPDDR chips and 4 low-latency RLDRAM chips. The low-power and the low-latency channels require their own independent 23-bit address/command bus and independent memory controllers on the processor. We will shortly propose optimizations to reduce these overheads.

In our CWF design, whenever there is an LLC miss, an MSHR entry is created and two separate memory requests are created, one in the LPDRAM memory controller and one in the RLDRAM mem-



(a) A baseline DDR3 system, with a 72-bit wide Data + ECC bus and a 23-bit wide address/control bus. The processor has four instances of this configuration.

(b) A heterogeneous Static-CWF memory system where word-0 is placed in a single-rank RLDRAM DIMM and other words are placed in a single-rank LPDRAM DIMM. The processor has four instances of this configuration.

(c) A low-cost heterogeneous Static-CWF memory system where the critical words of all 4 channels are aggregated on a 16-chip RLDRAM DIMM and other words are placed in four single-rank LPDRAM DIMMs.

Figure 5: A basic Static-CWF heterogeneous memory system and an optimized low-cost version of it.

ory controller. Both channels are completely independent and will access their respective DIMMs and return data many cycles later. In the common case, word-0 arrives first and is buffered at the MSHR. If it is the critical word, it is returned to the CPU. When the second part of the cache line arrives, the access is deemed complete. The caches are then populated, and the MSHR entry is freed. The added complexity is the support for buffering two parts of the cache line in the MSHR and the additional memory controller for the RLDRAM channel.

4.2.3. Handling ECC Checks. Memory systems are often required to provide ECC support that can detect and recover from a single bit error [38]. The ECC check can only happen after the entire cache line arrives. However, our CWF design wakes up a waiting instruction before confirming that the data is free of errors. Typically, the waiting instruction is at the head of the reorder buffer and will be retired soon after the critical word arrives and likely before the ECC code arrives. If the ECC-check flags an error, roll-back is often not possible because the instruction that consumed the critical word has already retired. We solve this problem as follows. A parity bit is attached to every byte stored in the RLDRAM DIMM. RLDRAM chips are already available in x18 and x36 configurations, i.e., they already are provisioned for additional bits that can be used for fault tolerance. When word-0 arrives at the processor, it is forwarded to the waiting instruction if there is no parity error. If there is a parity error, the data is forwarded only after the ECC code arrives and the error has been corrected. Parity checks can lead to silent data corruption if there are multi-bit errors. However, this is already beyond the scope of the baseline fault tolerance support which typically can only guarantee recovery from a single bit failure per 64-bit word. As in the baseline, our design will detect a 2-bit error on the 64-bit word when the ECC finally arrives. This general approach of lightweight error detection within RLDRAM and full-fledged error correction support within LPDRAM can also be extended to handle other fault tolerance solutions such as chipkill [9]. To summarize, the parity solution does not replace the SECDED scheme; it augments SECDED. So the error coverage is exactly the same as in the baseline SECDED. The only deviation is that a multi-bit error (not detected by parity) will cause an erroneous result to commit; this will be detected by the regular SECDED a few cycles later when the full cache line is received. The new model will also fail-stop and the point of failure will be precisely known.

4.2.4. Reducing Overheads. We propose two optimizations to improve upon the basic design described so far. First, we recognize that energy overhead is increased by striping word-0 across 4 chips on the RLDRAM DIMM. Instead, we place each word-0 and its parity bit on a single x9 RLDRAM chip. Hence, the RLDRAM DIMM is implemented as 4 ranks, each with one x9 chip. The address/command bus is now 26-bits wide to accommodate chip-select signals (see Figure 5b).

By having more narrow ranks, we not only reduce activation energy, we also increase rank and bank level parallelism and reduce queuing delays. While narrow ranks lead to a narrow row, that is not a problem because RLDRAM employs a close-page policy and does not yield row buffer hits anyway.

The second optimization targets the overhead of having an extra 26-bit wide address/command bus and memory controller for each critical-word RLDRAM channel. Our baseline has four 72-bit DDR3 channels. If our CWF proposal is applied to each channel, it produces four separate 9-bit RLDRAM channels, each with its own memory controller and 26-bit address/command bus. To reduce these overheads, we aggregate the four RLDRAM channels into a single memory controller and channel (see Figure 5c). The aggregated RLDRAM channel will now have a 36-bit data bus and a 38-bit address/command bus (note the extra signals for chip-select). This organization is similar to Rank-Subsetting [8] where one address/cmd bus is used to control many skinny data channels. The only change from our previous scheme is that the four 9-bit wide channels will all share the same address/command bus. It is safe to share the address/command bus among four data channels without creating contention. This is because the transfer of word-0 from a RLDRAM chip keeps the data bus busy for eight clock edges, but keeps the address bus busy for only two clock edges[1]. We can thus afford to increase the utilization of the address/command bus by a factor of four. The address/command bus now drives 16 chips on the RLDRAM DIMM, an already common scenario in modern channels.

With these optimizations in place, some of the initial overheads are reduced by a factor of four. Now, every access activates the same number of DRAM chips (9) as the baseline, keeping activation

[1]RLDRAM uses a close-page policy and does not require an explicit precharge on the address/cmd bus after every data access. Hence, the utilization of the data and address/cmd bus is in the ratio 4:1. We assume double-data-rate for the address/command bus, as is done for LPDRAM [29], and GDDR5 [5]

energy in check. We are introducing only one additional memory controller and only 42 (38 for the address/cmd bus and 4 for parity) new pins to control the RLDRAM channel. This is a small overhead given that modern processors have 1000+ pins.

4.2.5. Adaptive Placement. Design complexity is an important guiding metric in creating our static CWF architecture. We therefore assume that word-0 is always critical and place it in the RL-DRAM DIMM/channel. This makes it easy to locate and assemble the two parts of a cache line. As a potential study, we also consider a more complex organization where every cache line is allowed to designate one of its eight words as the critical word. We assume that the critical word on the previous fetch will also be the critical word on the next fetch. Hence, when a dirty line is evicted from cache, the line is re-organized and placed in DRAM memory such that the predicted critical word is placed in the low-latency DIMM. With such prediction, 79% of all critical words will be found in the low-latency DIMM, compared to the 67% success rate of the static CWF architecture. The critical word would need a 3-bit tag in cache and DRAM to identify itself as one of the eight words in the cache line. We present results for this model to showcase the additional room for improvement with an adaptive CWF optimization, but acknowledge the relatively high design complexity required to realize this added improvement.

4.2.6. Discussion. To summarize, our innovations allow us to enhance system performance and lower the memory system energy. To achieve this, we add a memory controller, one extra address/command bus and four extra data pins. On the processor, some support from the MSHR is required to handle the fragmented transfer of a cache line. The total memory capacity and DRAM error coverage is the same as in the baseline. The use of RLDRAM will lead to a higher purchase cost.

5. Methodology

Processor	
ISA	UltraSPARC III ISA
CMP size and Core Freq.	8-core, 3.2 GHz
Re-Order-Buffer	64 entry
Fetch, Dispatch, Execute, and Retire	Maximum 4 per cycle
Cache Hierarchy	
L1 I-cache	32KB/2-way, private, 1-cycle
L1 D-cache	32KB/2-way, private, 1-cycle
L2 Cache Coherence Protocol	4MB/64B/8-way, shared, 10-cycle Snooping MESI
DRAM Parameters	
DDR3	MT41J256M4 DDR3-1600 [28],
RLDRAM3	Micron MT44K32M18 [30]
LPDDR-2	Micron MT42L128M16D1 [29] (400MHz)
Baseline DRAM Configuration	4 72-bit Channels 1 DIMM/Channel (unbuffered, ECC) 1 Rank/DIMM, 9 devices/Rank
Total DRAM Capacity	8 GB
DRAM Bus Frequency	800MHz
DRAM Read Queue	48 entries per channel
DRAM Write Queue Size High/Low Watermarks	48 entries per channel 32/16

Table 1: Simulator parameters.

Simulator Details. We use the Wind River Simics [25, 4] simulation platform for our study. Table 1 details the salient features of the simulated processor and memory hierarchy. We model an out-of-order processor using Simics' *ooo-micro-arch* module and use a heavily modified *trans-staller* module for the DRAM simulator. The DRAM simulator is adapted from the USIMM simulation framework [12]. We also model a stride prefetcher. At the memory controller, demand requests are prioritized over prefetch requests in general, unless the prefetch requests exceed some age threshold, at which point they are promoted over demand requests.

Parameter	DRAM	RLDRAM3	LPDDR2
tRC	50ns	12ns	60ns
tRCD	13.5ns	–	18ns
tRL	13.5ns	10ns	18ns
tRP	13.5ns	–	18ns
tRAS	37ns	–	42ns
tRTRS	2 Bus Cycles	2 Bus Cycles	2 Bus Cycles
tFAW	40 ns	–	50ns
tWTR	7.5ns	0	7.5ns
tWL	6.5ns	11.25ns	6.5ns

Table 2: Timing Parameters [30, 29, 28, 23]

DRAM Simulator. The memory controller models a First-Ready-First-Come-First-Served (FR-FCFS) scheduling policy (for DDR3 and LPDRAM2) and models all DRAM, RLDRAM3 and LPDDR2 commands and timing parameters. We describe the most pertinent ones in Table 2. The DRAM device model and timing parameters were derived from Micron datasheets [28, 19, 29, 30]. DRAM address mapping parameters for our platform (i.e., number of rows/columns/banks) were adopted from Micron data sheets [28, 29, 30]. The open row address mapping policy from [19] is used in the baseline and LPDDR2. We use this address mapping scheme because this results in the best performing baseline on average when compared to other commonly used address interleaving schemes [44, 19]. For the RLDRAM modules, we use a close-page policy. Note that the queuing delays that we see in the baseline in our simulations (Figure 1.b) are lower than those observed in prior work such as [41] because we simulate more channels and larger last level cache capacities which reduce the traffic in each channel.

Workloads. Our techniques are evaluated with full system simulation of a wide array of memory-intensive benchmarks. We use multi-threaded workloads (each core running 1 thread) from the OpenMP NAS Parallel Benchmark [11] (cg, is, ep, lu, mg, sp) suite and the STREAM [3] benchmark. We also run multiprogrammed workloads from the SPEC CPU 2006 suite (astar, bzip2, dealII, gromacs, gobmk, hmmer, h264ref, lbm, leslie3d, libquantum, mcf, milc, omnetpp, soplex, sjeng, tonto, xalancbmk and zeusmp). Each of these single threaded workloads are run on a single core – so essentially each workload is comprised of 8 copies of the benchmark running on 8 cores. For multi-threaded applications, we start simulations at the beginning of the parallel-region/region-of-interest of the application. For the multiprogrammed SPEC benchmarks, we fast forward the simulation by 2 billion instructions on each core before taking measurements. We run the simulations for a total of 2 million DRAM read accesses after warming up each core for 5 million cycles. Two million DRAM read accesses correspond to roughly 540 million program instructions on average. For comparing the effectiveness of the proposed schemes, we use the total system throughput defined as $\sum_i (IPC^i_{shared}/IPC^i_{alone})$ where IPC^i_{shared} is the IPC of program i in a multi-core setting. IPC^i_{alone} is the IPC of program i on a stand-alone single-core system with the same memory system.

Power Modeling. We use the Micron DRAM power calculators [2] to model the power consumption of each DRAM chip. While the

DDR3 and RLDRAM3 calculators can directly be used to model the power of each of these variants of DRAM chips, we modify the DDR3 power calculator to serve as the power model for LPDDR2. We change the current values from those of DDR3 to the corresponding LPDDR2 values for all except the background currents (Idd3P and Idd3PS). This is to ensure that we do not artificially inflate the LPDDR2 power savings. As mentioned in Section 4.1 we assume that an LPDDR2 chip used on a DIMM will have a DLL. This DLL consumes power in idle modes. To account for this component of the idle power, we assume that an LPDDR2 chip consumes the same amount of current that a DDR3 chip does in idle state. We also assume that LPDRAM requires ODT and calculate the static power contribution of such termination resistors.

6. Evaluation

6.1. Critical Word Optimization

6.1.1. Performance Analysis. We evaluate the proposed critical word optimizations for three memory configurations. In the first two of these, an eighth of the DRAM capacity (meant for storing critical words) is constructed out of RLDRAM3 while the rest is built with DDR3 and LPDDR2 respectively. In the third configuration, DDR3 is used to store the critical word and the rest of the word is stored in LPDDR2. The three configurations are as follows.

- **RD** : 1GB RLDRAM3 and 7GB DDR3 (plus 1GB ECC)
- **RL** : 1GB RLDRAM3 and 7GB LPDDR2 (plus 1GB ECC)
- **DL** : 1GB DDR3 and 7GB LPDDR2 (plus 1GB ECC)

In Figure 6, we present the throughput of each of these systems normalized to an 8GB DDR3 baseline, while Figure 7 shows the average DRAM latency of the requested critical word. Each system represents a different point on the power–performance space, depending on the type of DRAM chip in use. We see in Figure 6 that RD has an average throughput improvement of 21% over DDR3 while RL has an average improvement of 12.9% over the baseline. The source of these improvements are the 30% and 22% reductions in critical word latency for RD and RL respectively. The overall performance degradation with the power optimized DL scheme is 9%.

Maximum improvements with the RLDRAM based configurations are observed for those benchmarks that have a large fraction of critical words placed in RLDRAM. Figure 8 shows the fraction of total critical word requests served by the faster RLDRAM module. Applications like *cg*, *lu*, *mg*, *sp*, *GemsFDTD*, *leslie3d*, and *libquantum* show significant improvements with RL and RD because most of the requested critical words for these benchmarks are to word 0 and hence are serviced with RLDRAM3 latency. In fact, for these benchmarks, RD and RL perform very similarly because the latency of the slower DRAM module rarely affects the system performance. For the same reason, DL is able to hide the latency of LPDDR2 to a large extent in these cases and suffers minimal performance degradation.

On the other hand, applications like *lbm*, *mcf*, *milc*, and *omnetpp* have a large fraction of critical words that are not the first word in the cache line and are hence serviced from the slower DRAM module (Figure 8). For these applications, RL performs nominally better than the baseline. In fact, with RL, bzip2 performs about 4% worse than the baseline because of the increased critical word latency.

To further understand the performance gains from the RL scheme, we collected statistics on the time gap between the first request for a cache-line and the subsequent access to the same cache-line, but to a different word. This is to determine if accesses to words 1 through 7 in the cache-line occur sufficiently late in the instruction window to tolerate the increased LPDDR latency. We compare the average gap between the first two accesses to a cache-line to the average LPDRAM access latency for all the benchmarks which have word-0 as the most frequently accessed critical word. We see that in the majority of the applications that benefit from our proposed architecture, this gap is greater than or very close to the LPDRAM latency for more than 82% of all accesses. This means that there is very little additional slowdown compared to an all DDR baseline for these applications. However, some applications such as *tonto* and *dealII*, which see reduced critical word latencies with both RD and RL, due to the large percentage of critical word 0 accesses, still experience marginal overall slowdown compared to the baseline. This is because in these applications, most of the second accesses to cache-lines (more than 75%) occur before the whole line is returned from the LPDDR.

All the results above are with a system that employs a stream prefetcher. In the absence of the prefetcher the performance gain with the RL system is 17.3%, simply because there is more opportunity for latency hiding with CWF.

We also perform an experiment where the critical words are assumed to be randomly mapped between the RLDRAM and LPDRAM systems (with the critical word being 7 times more likely to be found in the LPDRAM system compared to the RLDRAM system). This experiment is done to ensure that the intelligent data mapping contributes to the performance gains observed above. We see that with a random mapping, the average performance improvement is only 2.1% for RL (compared to the DDR3 baseline) with many applications showing severe performance degradation because a large percentage of their critical words are fetched from LPDRAM.

6.1.2. Analyzing RLDRAM+LPDDR performance. Since the RL scheme has comparable performance to the RD scheme and (as shown later) has significant energy advantages over RD, we choose it as our flagship configuration. We apply the adaptive critical word placement on it and present the results in the following graph, Figure 9.

The bar **RL OR** in Figure 9 represents system throughput with an oracular scheme, where we assume that every critical word is fetched from RLDRAM3. The performance obtained with RL-OR is the highest (28% improvement) that can be achieved by the RL configuration. Note that the performance thus obtained will be lower than an all-RLDRAM3 system represented by the last bar, RLDRAM3, in Figure 9. Note that in RL OR, only the critical word comes from the RLDRAM3 part, compared to the whole cache line in RLDRAM3. The other factor restricting the performance of the RL OR system is the limited address/command bus bandwidth. Even though the RLDRAM chips are organized into 4 independent data channels, the single address and command bus can become a bottleneck, especially for applications with high memory pressure such as *mcf*, *milc*, *lbm*.

The RL AD bar represents the performance of the adaptive critical word placement scheme. The RL AD scheme re-organizes the layout of the cache word to place the last used critical word in the RLDRAM3 part. This scheme performs better (15.7% improvement over baseline) than the RL scheme (12.9% improvement over the baseline) because it lowers the critical word latency of more requests compared to the RL scheme. *Mcf* is one of the applications that benefits from adaptive placement (a 20% improvement over baseline compared to 12% for RL). This is because in mcf, word 0 and word

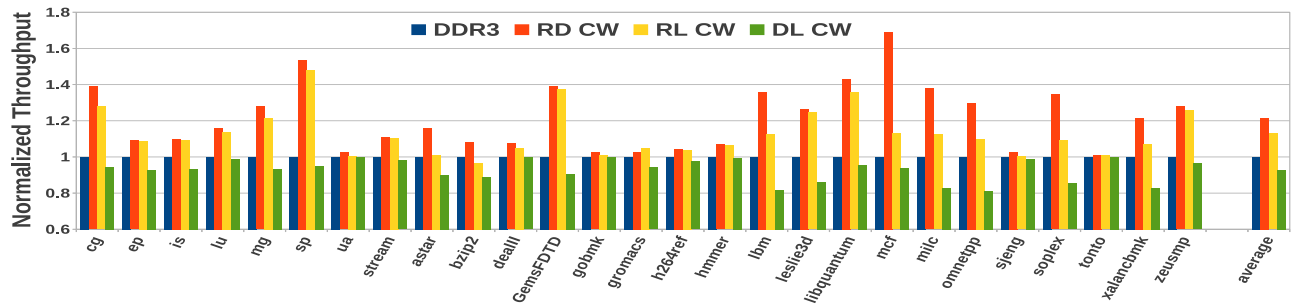


Figure 6: Performance of CW optimization

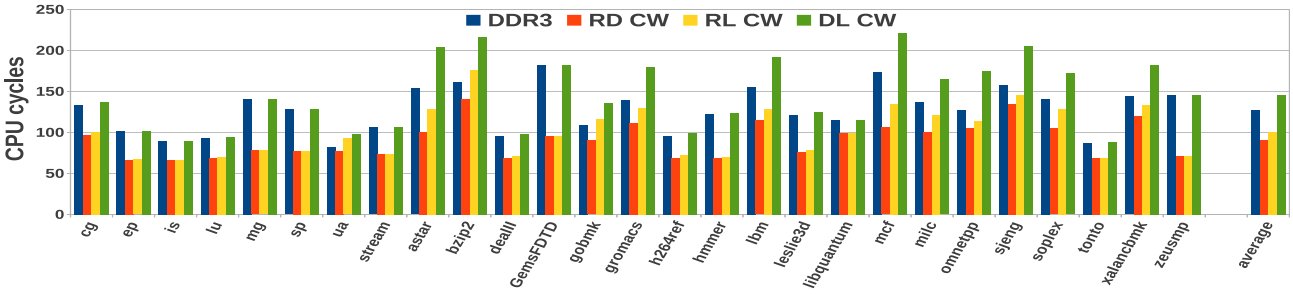


Figure 7: Critical Word Latency

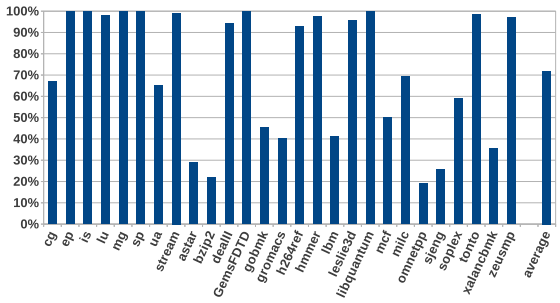


Figure 8: Percentage of Critical Word Accesses Served By RLDRAM3

3 are the most accessed critical words (see Figure 4). With adaptive placement, during a write to the lines which have word 3 as the critical word, the layout of the cache line is altered and subsequent reads to these lines have lower critical word latencies. The performance of the RL AD scheme is dependent on the write-traffic –i.e., unless a word is written to, its organization in the main memory is not altered. Therefore, not all applications that have words 1 through 7 as critical words can be accelerated with the RL AD scheme.

6.1.3. Energy Analysis. In this section, we analyze the energy consumption of the ensemble. The system energy consumption for the different configurations is shown in Figure 10.

Methodology: To calculate system energy, we assume the power consumption of the DRAM system in the baseline to be 25% of the entire system [27, 22]. We assume that one-third of the CPU power is constant (leakage + clock), while the rest scales linearly with CPU activity. We calculate the power consumption of DDR3, RLDRAM3 and LPDDR2 chips using activity factors from the simulator, which are then fed to the Micron DRAM power calculators [2].

As seen in Figure 2, the power consumption of a single RLDRAM3 chip is higher than that of a DDR3 chip at the bus utilization values that we see in our simulations (between 5% and 40%). In an RL system, we have 16 RLDRAM3 chips and 32 LPDDR2 chips compared to 36 DDR3 chips in the baseline. In the baseline, every cache line request requires the activation of 9 DDR3 chips, while in RL, 1 RLDRAM3 chip and 8 LPDDR2 chips are activated for each request. The high power consumption of RLDRAM3 is alleviated by the lower power consumption of the LPDDR2 chips. In our experiments we find that by only fetching one cache-word from the RLDRAM3 chip, its active power consumption is kept in check. Also, since each RLDRAM3 chip sees one-fourth of the activity seen by each DDR3 chip (because of the sub-ranking optimization), the I/O power of the RLDRAM3 chips is kept lower. The LPDDR2 channels, due to their lower frequency and lower operating voltage consume about 20% less power on average compared to the DDR3 system. The faster power-down entry and exit modes allow the LPDDR2 ranks to be put to sleep more frequently compared to the baseline. The overall memory power decreases by 1.9% and memory energy reduces by 15%.

We see that the overall system energy consumption drops by about 6% with RL while the DL scheme consumes about 13% lower energy compared to a DDR3 baseline. The RL scheme exhibits maximum energy savings for high-bandwidth applications with high critical word 0 accesses –e.g., mg, sp, GemsFDTD, leslie3d, and libquantum. High DRAM utilization bridges the gap between DDR3's power consumption and that of RLDRAM3 (Figure 2) to a certain extent. When such high-bandwidth applications can benefit from the critical word optimization, i.e., have a high number of critical word 0 accesses, the overall system energy is reduced. On the other hand, high bandwidth applications such as lbm, mcf, and milc, which show modest performance improvements with our schemes do not show a marked reduction in system energy consumption.

Applications like bzip2, dealII and gobmk, have low bandwidth requirements. As a result, the power consumption of the RLDRAM3 component is high enough that the DRAM power in the RL configuration exceeds the baseline DDR3 power consumption. Coupled with this are the marginal execution time reductions (increase in bzip2) for these applications with RL, resulting in overall system

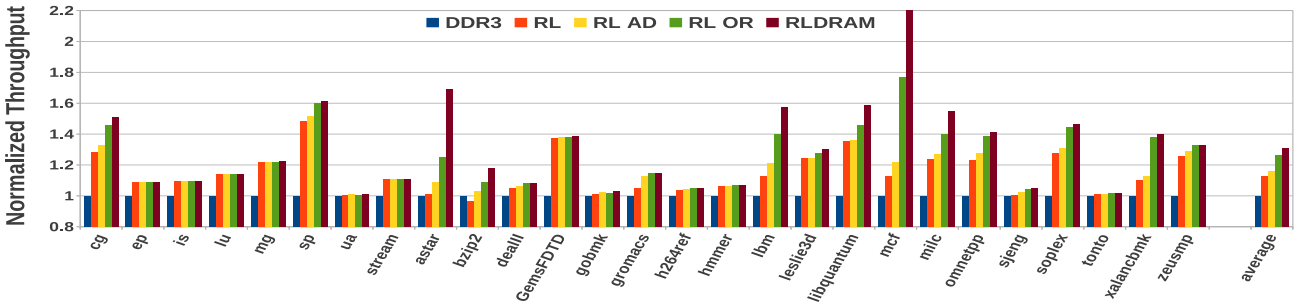


Figure 9: Performance of various RLDRAM3+LPDDR2 configurations

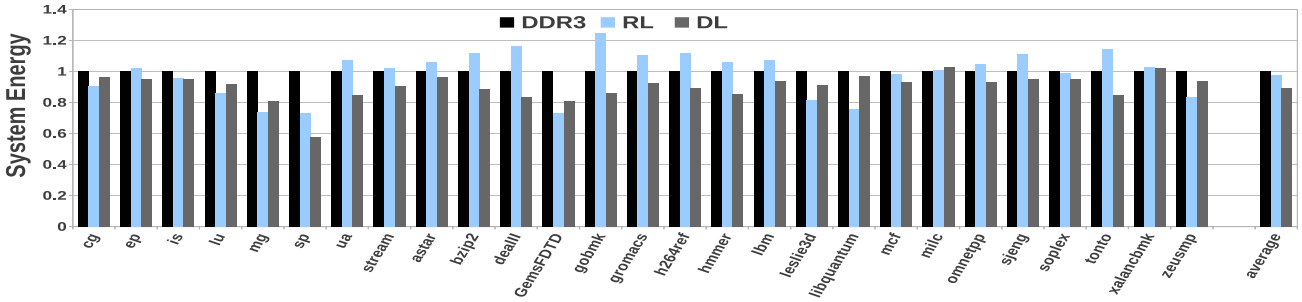


Figure 10: System Energy Normalized to DDR3 Baseline

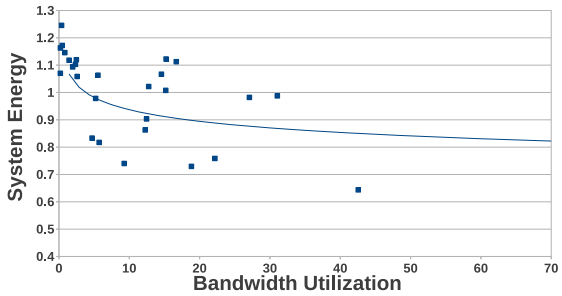


Figure 11: Bandwidth Utilization vs Energy Savings

energy consumption increases.

Figure 11 shows the energy savings obtained for the RL scheme as a function of bandwidth utilization (each point in the figure represents a different workload). As explained above, the energy savings are a function of two parameters – the critical word regularity and the bandwidth utilization. We notice in Figure 11 that in general, with increasing bandwidth utilization, the system energy savings are progressively greater. This is because the energy gap between RLDRAM and DDR3 shrinks at high utilization. This implies that in future systems constrained by bandwidth, our innovations can be expected to provide greater energy savings and performance improvements.

7. Alternate Heterogeneous Memory Design Options

7.1. Comparison to Page Placement Proposals

Most heterogeneous main memory proposals, including that of Phadke et al., place a group of cache-lines (usually belonging to the same physical page) in a single DRAM variant on a single channel. To compare the merits of such a strategy against our proposal we evaluate a heterogeneous memory system that consists of RLDRAM3 and LPDRAM2. An application is statically profiled and a fraction of its pages are placed in RLDRAM3 and the rest in LPDRAM2. In this experiment, to ensure that the memory configuration is practical, we assume that the system consists of four 72 bit wide channels. Three contain 2GB LPDDR2 DIMMs each while the fourth channel has only .5GB of RLDRAM3 memory. This design allows the RLDRAM3 system to enjoy the pin bandwidth of a conventional system and curbs the energy consumption of the RLDRAM3 system by restricting the number of chips. Hence the baseline configuration and the one described above are iso-pin-count and iso-chip-count. We do not consider the performance effects of the reduced RLDRAM3 capacity. The applications are profiled offline and the top 7.6% (.5GB/6.5GB) of the accessed pages wwere placed in RLDRAM3. We see that the performance varies widely, from a 9.3% loss to a 11.2% improvement, yielding an average improvement of about 8%. Those applications, most of whose working set fits in the RLDRAM memory, benefit from this organization showing higher benefits compared to our scheme. However, the loss in performance is due to many accesses to the LPDRAM – since the top 7.6% of pages only account for a maximum of 30% of all accesses for any of the programs. In addition, [39] demonstrates that for most applications, few cache-lines in a page make up the majority of all accesses to that page. Thus, allocating space on RLDRAM at a page granularity does not yield much benefit. It is also necessary to note that the improvements reported here will likely be less once page-fault effects due to the smaller overall capacity of the memory system are taken into consideration (our simulations are not long enough to consume even the reduced memory capacity). The power consumption of this system is decidedly better than our scheme, because

- there is less RLDRAM3 in the system compared to the proposed scheme, thus less background power.
- pages in LPDRAM have low access frequencies, thus the LPDRAM can stay in power down modes for longer, thereby reducing power consumption further.

7.2. Alternate LPDRAM Design

As noted in Section 9, Malladi et al. have looked at adapting LPDRAM for use in server memories [26]. Their analog simulations

show that even in the absence of On–Die–Termination, the "eye" of the read and write signals is wide enough to allow using unmodified LPDRAM chips (grouped into packages of four) to construct the memory channel. We simulate a similar system in our experiments and find that the LPDRAM power is further reduced in this case with very little loss in performance (due to deeper sleep modes) and thus the energy savings are boosted to 26.1%.

8. Cost Analysis

The total cost of ownership of a server system is an important design metric. The acquisition cost of a product is intrinsically tied to the volume of production. LPDRAM is already being used in high–volume markets (portable compute devices) which will help reduce its price. The higher cost–per–bit of the RLDRAM3 devices is kept in check in our design by using RLDRAM for only 1/8th of the total memory space. The CapEx involved will reduce if a case can be made for large–scale adoption of such devices. In the recent future, we expect non–volatile memories to relieve the DRAM system of its high density requirements. It can therefore be expected that DRAM products, their target markets, and their volumes will evolve accordingly in the future. Energy–optimized memory systems, like the one we propose in this work, will drive down the OpEx by reducing overall system–energy. We expect this to be a big motivation for the adoption of such a design in the data–center space and this will likely help drive up the volume of these specialty parts.

9. Related Work

Critical Word Regularity : Regularity in the critical word accesses was reported by Gieske et al. [17]. In this work the repeatable patterns in critical words to optimize the L2 cache's power and performance characteristics via a "critical–word–cache".

Heterogeneous Memory : A recent paper [35] proposes one possible application of heterogeneous DRAM based memories. The authors propose an offline analysis of the LLC miss–rate and average wait–time of an instruction at the head of the queue to determine if an application is latency or bandwidth sensitive. Section 3 already provides a qualitative comparison. Quantitatively, we observed that for a similar pin count and chip count as the baseline, this approach only yields an 8% improvement.

Dong et al. [15] discuss the possibility of having a two–tiered DRAM–based hierarchy where an on–chip dram is used in conjunction with traditional off–chip DDR DIMMs. However, in contrast to convention [24, 47, 20], the on–chip dram is not used as a large LLC, but instead considered a part of the main memory. The authors describe hardware accounting techniques to identify candidate pages to migrate from the on–chip dram to the main DRAM.

Memory design with LPDRAM : Very recently, two techniques to construct server memory entirely out of mobile DRAM parts have been proposed [45, 26]. In these papers, the authors aim to exploit the low background–power characteristics of LPDRAM and focus on solutions to overcome the lower bandwidth, lower channel capacity and ECC implementation problems of memory systems built with LPDDR. Our proposals are complementary to their designs. Specifically, we demonstrate that the addition of RLDRAM to such a LPDDR based design can provide significant performance benefits for most applications.

Hybrid DRAM+NVM Memory : Several papers have suggested integrating Phase Change Memory (PCM) technology in the main memory hierarchy [23, 48, 36, 46]. Considering the longer read and write latencies of PCM, it seems likely that DRAM cannot be completely substituted by PCM – rather, a combination of the two technologies would be the better choice. Consequently, the management of such a hybrid memory space has been studied by some recent papers. Proposals for managing such a hybrid system consist of placement of performance critical and frequently written pages in DRAM and the rest in PCM [37], write–frequency counter guided page placement to manage wear–levelling [14] and migration of data pages between PCM and DRAM based on access counts to mitigate background power consumption [33].

10. Conclusions

We show that modern memory systems can be enhanced by promoting heterogeneity in their construction. Existing production DRAM parts can be utilized to construct an efficient heterogeneous main memory system with low complexity with changes to the memory controller and channel organization. We also identify a novel pattern in the repeatability of critical word accesses at the DRAM level for a variety of workloads and achieve 12.9% performance improvement and 6% system energy reduction by exploiting the critical word regularity with a heterogeneous memory. Trends indicate that our techniques can be used in future bandwidth–hungry systems to achieve higher performance and energy savings.

It is also likely that in the future, 3D–stacked DRAM technology will become mainstream. The recently announced Hybrid Memory Cube (HMC) from Micron [34] is one embodiment of such 3D technology. Each HMC device consists of 2D DRAM dies that are stacked on top of one another and connected via high–bandwidth TSVs to a logic layer at the bottom. The HMC is touted as a solution for the scaling problems with current DDR technology by virtue of its high capacity, high bandwidth and low power consumption. If we witness widespread use of HMCs in the future, then we can think of integrating some kind of heterogeneity in a HMC–based memory system. There are two ways to enable a critical–data–first architecture with HMCs. In one possible variant, one could include dies with different latency/energy properties and the critical data could be returned in an earlier high–priority packet. In another implementation, one could imagine having a mix of high–power, high–performance and low–power, low–frequency HMCs. The high–speed signalling employed in the baseline HMC makes it power–hungry. Thus a critical data bit could be obtained from a high–frequency HMC and the rest of the data from a low–power HMC.

11. Acknowledgments

We thank our reviewers (especially our shepherd Derek Chiou) and members of the Utah Arch group for their suggestions to improve this work.

References

[1] "Micron RLDRAM Memory," http://www.micron.com/~/media/Documents/Products/Product%20Flyer/rldram_flyer.pdf.
[2] "Micron System Power Calculator," http://goo.gl/4dzK6.
[3] "STREAM – Sustainable Memory Bandwidth in High Performance Computers," http://www.cs.virginia.edu/stream/.
[4] "Wind River Simics Full System Simulator," http://www.windriver.com/products/simics/.
[5] "Quimonda GDDR5−White Paper," http://www.hwstation.net, 2007.
[6] "All You Need to Know About Mobile LPDRM," http://download.micron.com/pdf/flyers/mobile_lpdram_flyer.pdf, 2008.

[7] "RLDRAM3 Press Release," http://www.issi.com, 2011.
[8] J. Ahn *et al.*, "Future Scaling of Processor-Memory Interfaces," in *Proceedings of SC*, 2009.
[9] AMD Inc., "BIOS and Kernel Developer's Guide for AMD NPT Family 0Fh Processors."
[10] O. Azizi, A. Mahesri, B. Lee, S. Patel, and M. Horowitz, "Energy-Performance Tradeoffs in Processor Architecture and Circuit Design: A Marginal Cost Analysis," in *Proceedings of ISCA*, 2010.
[11] D. Bailey *et al.*, "The NAS Parallel Benchmarks," *International Journal of Supercomputer Applications*, vol. 5, no. 3, pp. 63–73, Fall 1994.
[12] N. Chatterjee, R. Balasubramonian, M. Shevgoor, S. Pugsley, A. Udipi, A. Shafiee, K. Sudan, M. Awasthi, and Z. Chishti, "USIMM: the Utah SImulated Memory Module," University of Utah, Tech. Rep., 2012, uUCS-002-12.
[13] V. Cuppu, B. Jacob, B. Davis, and T. Mudge, "A Performance Comparison of Contemporary DRAM Architectures," in *Proceedings of ISCA*, 1999.
[14] G. Dhiman, R. Ayoub, and T. Rosing, "PDRAM: A Hybrid PRAM and DRAM Main Memory System," in *Proceedings of DAC*, 2009.
[15] X. Dong, Y. Xie, N. Muralimanohar, and N. Jouppi, "Simple but Effective Heterogeneous Main Memory with On-Chip Memory Controller Support," in *Proceedings of SC*, 2010.
[16] K. Ganesan, J. Jo, and L. K. John, "Synthesizing Memory-Level Parallelism Aware Miniature Clones for SPEC CPU2006 and ImplantBench Workloads," in *Proceedings of ISPASS*, 2010.
[17] E. J. Gieske, "Critical Words Cache Memory : Exploiting Criticality Withing Primary Cache Miss Streams," Ph.D. dissertation, 2008.
[18] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 4th ed. Elsevier, 2007.
[19] B. Jacob, S. W. Ng, and D. T. Wang, *Memory Systems - Cache, DRAM, Disk*. Elsevier, 2008.
[20] X. Jiang, N. Madan, L. Zhao, M. Upton, R. Iyer, S. Makineni, D. Newell, Y. Solihin, and R. Balasubramonian, "CHOP: Integrating DRAM Caches for CMP Server Platforms," *IEEE Micro (Top Picks)*, Jan/Feb 2011.
[21] C. Kozyrakis, A. Kansal, S. Sankar, and K. Vaid, "Server Engineering Insights For Large-Scale Online Services," in *IEEE Micro*, 2010.
[22] J. Laudon, "UltraSPARC T1: A 32-Threaded CMP for Servers," 2006, invited talk, URL: http://www.cs.duke.edu.
[23] B. Lee, E. Ipek, O. Mutlu, and D. Burger, "Architecting Phase Change Memory as a Scalable DRAM Alternative," in *Proceedings of ISCA*, 2009.
[24] G. Loh and M. Hill, "Efficiently Enabling Conventional Block Sizes for Very Large Die-stacked DRAM Caches," in *In Proceedings of MICRO*, 2011.
[25] P. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner, "Simics: A Full System Simulation Platform," *IEEE Computer*, vol. 35(2), pp. 50–58, February 2002.
[26] K. T. Malladi, F. A. Nothaft, K. Periyathambi, B. C. Lee, C. Kozyrakis, and M. Horowitz, "Towards Energy-Proportional Datacenter Memory with Mobile DRAM," in *Proceedings of ISCA*, 2012.
[27] D. Meisner, B. Gold, and T. Wenisch, "PowerNap: Eliminating Server Idle Power," in *Proceedings of ASPLOS*, 2009.
[28] "Micron DDR3 SDRAM Part MT41J256M8," Micron Technology Inc., 2006.
[29] "Micron Mobile LPDDR2 Part MT42L128M16D1," Micron Technology Inc., 2010.
[30] "Micron RLDRAM 3 Part MT44K32M18," Micron Technology Inc., 2011.
[31] J. Ousterhout *et al.*, "The Case for RAMClouds: Scalable High-Performance Storage Entirely in DRAM," *SIGOPS Operating Systems Review*, vol. 43(4), 2009.
[32] J. P. Park, S. J. Rhee, S. B. Ko, Y. Jeong, K. S. Noh, Y. Son, J. Youn, Y. Chu, H. Cho, M. Kim, D. Yim, H. C. Kim, S. H. Jung, H. I. Choi, S. Yim, J. B. Kee, J. S. Choi, and K. Oh, "A 1.2V 30nm 1.6Gb/s/pin 4Gb LPDDR3 SDRAM with Input Skew Calibration and Enhanced Control Scheme," in *Proceedings of ISSCC*, 2012.
[33] Y. Park, D. Shin, S. Park, and K. Park, "Power-aware Memory Management For Hybrid Main Memory," in *Proceedings of ICNIT*, 2011.
[34] T. Pawlowski, "Hybrid Memory Cube (HMC)," in *HotChips*, 2011.
[35] S. Phadke and S. Narayanasamy, "MLP-aware Heterogeneous Main Memory," in *In Proceedings of DATE*, 2011.
[36] M. Qureshi, V. Srinivasan, and J. Rivers, "Scalable High Performance Main Memory System Using Phase-Change Memory Technology," in *Proceedings of ISCA*, 2009.
[37] L. Ramos, E. Gorbatov, and R. Bianchini, "Page Placement in Hybrid Memory Systems," in *Proceedings of ICS*, 2011.
[38] B. Schroeder *et al.*, "DRAM Errors in the Wild: A Large-Scale Field Study," in *Proceedings of SIGMETRICS*, 2009.
[39] K. Sudan, N. Chatterjee, D. Nellans, M. Awasthi, R. Balasubramonian, and A. Davis, "Micro-Pages: Increasing DRAM Efficiency with Locality-Aware Data Placement," in *Proceedings of ASPLOS-XV*, 2010.
[40] C. Toal, D. Burns, K. McLaughlin, S. Sezer, and S. O'Kane, "An rl-dram ii implementation of a 10gbps shared packet buffer for network processing," in *Proceedings of the 2nd NASA/ESA Conference on Adaptive Hardware and Systems*, 2007.
[41] A. N. Udipi, N. Muralimanohar, N. Chatterjee, R. Balasubramonian, A. Davis, and N. Jouppi, "Rethinking DRAM Design and Organization for Energy-Constrained Multi-Cores," in *Proceedings of ISCA*, 2010.
[42] VMWare Performance Team, "Ten Reasons Why Oracle Databases Run Best on VMware," http://blogs.vmware.com/performance/2007/11/ten-reasons-why.html, 2007.
[43] T. Vogelsang, "Understanding the Energy Consumption of Dynamic Random Access Memories," in *Proceedings of MICRO*, 2010.
[44] D. Wang *et al.*, "DRAMsim: A Memory-System Simulator," in *SIGARCH Computer Architecture News*, September 2005.
[45] D. H. Yoon, J. Chang, N. Muralimanohar, and P. Ranganathan, "BOOM: Enabling Mobile Memory Based Low-Power Server DIMMs," in *Proceedings of ISCA*, 2012.
[46] W. Zhang and T. Li, "Exploring Phase Change Memory and 3D Die-Stacking for Power/Thermal Friendly, Fast and Durable Memory Architectures," in *Proceedings of PACT*, 2009.
[47] Z. Zhang, Z. Zhu, and Z. Zhang, "Design and Optimization of Large Size and Low Overhead Off-chip Caches," *IEEE Transactions on Computer*, July 2004.
[48] P. Zhou, B. Zhao, J. Yang, and Y. Zhang, "A Durable and Energy Efficient Main Memory Using Phase Change Memory Technology," in *Proceedings of ISCA*, 2009.

A. Appendix: Explaining Critical Word Accesses in Programs

A profile of LLC filtered memory accesses of different applications shows that for almost all cache lines, one particular word in the cache line accounts for the vast majority of critical accesses to the cache line. The existence of such critical word regularity is not surprising, i.e., it is reasonable to expect that data in a region will be traversed in a similar order on multiple occasions. Such critical word regularity was reported and exploited by Gieske to optimize LLC performance [17].

Applications that stream through large data arrays will exhibit critical word accesses to words near the beginning of the cache line – most often to word 0 (at the DRAM level). Applications which have strided accesses with small stride lengths are most likely to have word 0 as the most critical word. For example, an application like hmmer has a dominant stride length of 0 for 90% of all accesses [16] – hence word 0 is the most popular critical word for hmmer. The STREAM benchmark has 4 parts, Copy, Scale, Sum and Triad. Each of these kernels uses unit strided accesses over large arrays, also results in word 0 being critical.

On the other hand, applications like mcf and xalancbmk spend most of their time in chasing pointers. For xalancbmk, 80% of misses are generated by two nested loops of pointer chasing, while in mcf, 70% of accesses are generated by random pointer chasing. These kinds of applications demonstrate a more uniform critical word distribution.

2012 IEEE/ACM 45th Annual International Symposium on Microarchitecture

Transactional Memory Architecture and Implementation for IBM System z

Christian Jacobi, Timothy Slegel
IBM Systems and Technology Group
Poughkeepsie, NY, USA
cjacobi@us.ibm.com, slegel@us.ibm.com

Dan Greiner
IBM Systems and Technology Group
San Jose, CA, USA
dgreiner@us.ibm.com

***Abstract*—We present the introduction of transactional memory into the next generation IBM System z CPU. We first describe the instruction-set architecture features, including requirements for enterprise-class software RAS. We then describe the implementation in the IBM zEnterprise EC12 (zEC12) microprocessor generation, focusing on how transactional memory can be embedded into the existing cache design and multiprocessor shared-memory infrastructure. We explain practical reasons behind our choices. The zEC12 system is available since September 2012.**

I. Introduction And Related Work

Over the last years, the number of CPU cores on a chip and the number of CPU cores connected to a shared memory have grown significantly to support growing workload capacity demand. For example, the IBM zEC12[1] enterprise server [1][2] supports operating system images with up to 101 CPUs. The increasing number of CPUs cooperating to process the same workloads puts significant burden on software scalability; for example, shared queues or data-structures protected by traditional semaphores become hot spots and lead to sub-linear n-way scaling curves. Traditionally this has been countered by implementing finer-grained locking in software, and with lower latency/higher bandwidth interconnects in hardware. Implementing fine-grained locking to improve software scalability can be very complicated and error-prone, and at today's CPU's frequency, the latency of hardware interconnects is limited by the physical dimension of the chips and systems, and by the speed of light.

In [3], Herlihy et al. introduced transactional memory: a group of instructions called a transaction is operating atomically and in isolation (called *serializability* in [3]) on a data structure in memory; the transaction executes optimistically without obtaining a lock, but may need to abort and retry if the operation conflicts with other operations on the same memory locations. The authors propose one implementation with a special transaction cache to hold pending transactional stores.

Various alternative hardware transactional memory designs have been proposed since [3]. In the Transactional Memory Coherence and Consistency (TCC) model [4], all stores performed during a transaction are buffered, and a request to store them out to memory is put on the bus at the end of the transaction. The bus arbitrates between multiple CPUs, and while one CPU is storing its stores, other CPUs are snooping the stores for conflicts and abort their transaction if necessary. A different approach is chosen in LogTM [5]: the transaction speculatively updates the memory but keeps the original value in a log and can restore the original memory content from the log in case of an abort. LogTM allows faster commit than TCC, and the less frequent aborts take longer than in TCC. LogTM uses a directory based eager conflict detection mechanism, where on a local cache miss other CPUs are informed of the transactional access so that they can detect potential conflicts and abort. Recent commercial implementations include Sun Microsystems' Rock [6] and IBM's BlueGene/Q processors [7]; Intel has announced transactional memory for their Haswell CPUs expected in 2013 [8]. A major difference of our architecture is that it supports *constrained transaction* which are guaranteed to eventually succeed.

In [9], architectural semantics for transactional memory have been studied. The authors propose to combine transactional hardware with a software layer that provides 2-phase commit, software handlers for transaction commit/abort, and closed and open nested transactions. In [10], the same group discusses virtualization of transactions to address overflows, interrupts, and other conditions with the help of the operating system. We have chosen to implement a pure hardware transactional system with closed nesting. In our design, each non-constrained transaction needs a fallback path, which then can also be invoked in circumstances where the transaction cannot complete, e.g. due to interrupts or transaction size overflows. Therefore the additional complexity of a software assist layer was not warranted for our implementation. Not relying on a software layer furthermore enables all components of the software stack including firmware, hypervisor, and operating system to exploit transactional memory.

There is a significant body of work on software-based and hybrid hardware/software transactional memory; since this is only remotely related to our work we refer to a transactional memory overview that discusses some of them [11].

The three use cases we considered during the definition of transactional memory are lock elision, lock-free data structures, and general code optimization. In lock elision [12], a data structure that is typically guarded by a lock is accessed with transactional memory operations without first obtaining the lock. If the transaction aborts due to conflicts with other CPUs, the program can obtain the lock as a fallback path (see figure 1). Every transaction must check that the lock is free to prevent concurrent operation of a transactional CPU and a CPU currently in the fallback path. This method also works with programs that have only been partially changed to use transactional memory, which is important for realistic introduction of transactional memory into large software products. In [12], lock elision is defined with special instructions; we embed the idea into the general transactional memory context. Lock free data structures have been studied extensively (see e.g. [13]). Transaction semantics provide a more powerful and easy-

[1]IBM, System z, z/OS, z/Architecture, zEnterprise, Blue Gene are trademarks of the International Business Machines Corporation.

1072-4451/12 $26.00 © 2012 IEEE
DOI 10.1109/MICRO.2012.12

IEEE computer society

```
        LHI      R0,0            *initialize retry count=0
loop    TBEGIN                   *begin transaction
        JNZ      abort           *go to abort code if CC!=0
        LT       R1,lock         *load&test the fallback lock
        JNZ      lckbzy          *branch if lock busy
        ...perform operation...
        TEND                     *end transaction
        ...
lckbzy  TABORT                   *abort if lock busy; this resumes after TBEGIN

abort   JO       fallback        *no retry if CC=3
        AHI      R0,1            *increment retry count
        CIJNL    R0,6,fallback   *give up after 6 attempts
        PPA      R0,TX           *random delay based on retry count
        ... potentially wait for lock to become free
        J        loop            *jump back to retry

fallback
        OBTAIN   lock            *using Compare&Swap
        ...perform operation...
        RELEASE  lock
        ...
```

Figure 1: Example Transaction

to-use foundation than for example a simple compare-and-swap instruction. Lastly, transactions provide a mechanism for code optimization, for example re-ordering code more aggressively, relying on the atomicity and isolation of memory accesses and the provided register-rollback for correctness [14].

During the definition of the transactional memory facility, it became clear very early that special care had to be taken for software testability and debug. Transactional memory poses interesting challenges [15]: for example, the non-transactional fallback path is rarely exercised which may lead to test coverage problems; when a program fails for example with an access exception inside a transaction, the memory and register state is rolled back due to transaction abort, which makes post-mortem analysis more difficult; and lastly, in our implementation interactive debugging is restricted by the fact that interrupts cause transactions to abort, and so for example, setting break-points inside a transaction would be impractical. We discussed many alternative solutions with IBM software teams in order to minimize hardware cost and complexity while meeting the requirements for efficient software development. In section 2, we describe the instruction set architecture for transactional memory, including features for software test and debug.

The design of a high-speed, scalable, and reliable multiprocessor shared memory protocol is very complex and time-consuming and represents a significant investment. The microprocessor design also evolves from generation to generation without a complete redesign. It was therefore clear from the beginning that transactional memory support must fit into a mostly unchanged SMP protocol, and that changes to the microprocessor core design should be minimized. At the same time the design must provide robust transactional memory performance to compete with the performance of obtaining idle locks; otherwise a compiler could not know at compile time whether to replace a lock with a transaction. We describe our implementation in section 3. In section 4 we evaluate the performance of the transactional memory implementation under a set of micro-benchmarks. The results show good scalability even for very high numbers of CPUs under realistic contention scenarios. We also provide some early results on real-world code. Section 5 summarizes the paper.

II. Instruction Set Architecture

A. New Instructions and General TX Operation

The Transactional Execution (TX) Facility provides 6 new instructions to the z/Architecture [16], as well as a few new control bits. This section gives an overview of the central features of the facility, further details can be found in [16].

Transactions are formed by pairs of *Transaction Begin* and *Transaction End* instructions (TBEGIN and TEND, respectively). Except as described below, either all or none of the instructions inside a transaction are executed (*atomicity*), and all operand accesses to memory are performed isolated (sometimes called serializable, block-concurrent, or also atomic), that is, other CPUs and the I/O subsystem cannot observe changes made by the transaction before it successfully ends, and the transaction cannot observe changes made by other CPUs or the I/O subsystem during the transaction.

Transactions may abort. There are various reasons for transactions to abort, including interrupts (e.g., page faults, divide by zero, or asynchronous interrupts like timer and I/O), exceeding the maximum nesting depth, overflow of the CPU's capability to track transactional accesses to memory (*footprint overflow*), or conflicts on accessed memory locations with other CPUs or the I/O subsystem that would cause an isolation violation if the transactional execution would continue. Privileged instructions that modify the control state of the CPU and some other complex instructions are not allowed inside a transaction and always lead to a transaction abort.

The architecture requires that the partial execution of the transaction before an abort was detected is isolated with respect to other CPUs and I/O (this is referred to as *opacity* in [17]). This

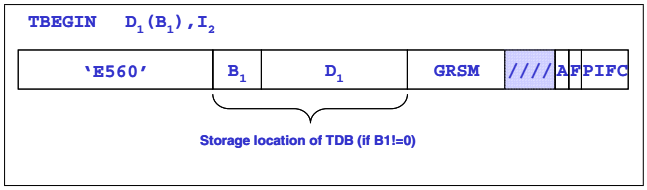


Figure 2: TBEGIN instruction text

was an important request from our software community since it prevents execution based on inconsistent data. For example, one transaction might pop the last element from a stack by updating the count of elements to 0 and by setting the stack pointer to NULL. If another transaction would first read the old non-zero element count, and then would proceed to access the stack pointer without NULL-checking, a page fault would result. Such situations are prevented by requiring that the entire execution is isolated even if the transaction aborts. Herlihy [3] proposed a *validate* instruction to prevent such *zombie transactions* [18]. We choose the stronger isolation since it is easier to use for software developers and does not require additional path length for the intermediate validations.

The architecture also requires isolation of transactions against non-transactional accesses done by other CPUs and I/O (*strong atomicity* in [19]). Again this was an important requirement from our software community to be able to mix transactional and traditional locking-based code in order to ease stepwise introduction of TM technology into existing programs.

Execution of TBEGIN sets the Condition Code (CC) to 0. If a transaction aborts, the *Instruction Address* in the *Program Status Word* (PSW) is restored to the instruction immediately after the TBEGIN, and a condition code is set to a non-zero value. A typical program will test the condition code after TBEGIN to either start the actual transaction processing (CC=0) or branch to the *abort handler* (CC!=0). The abort handler may return to retry the transaction, or it may perform the same functionality non-transactionally on a *fallback path*. An example is shown in figure 1.

Depending on whether the CPU considers the abort reason transient (e.g. another CPU conflicting) or permanent (e.g. a restricted instruction), the condition code is either set to 2 or 3. This allows the program a quick check on whether it should retry the transaction (with a threshold count) or immediately branch to a non-transactional fallback path. Depending on the program, certain clean-up is necessary before repeating the transaction, like restoring certain registers (see below).

Before repeating a transaction after a transient abort, it often makes sense to introduce a random delay that increases with the number of aborts, in order to prevent harmonic, repetitive, aborts due to conflicts between two CPUs (for example using random exponential back-off). The optimal delay distribution may depend on the particular abort reason, specifics of the design of the CPU generation, and details of the SMP configuration. In order to avoid changing the program to adjust the delays to these design parameters, the new *Perform Processor TX-Abort Assist* (PPA with function code TX) instruction is introduced. The program passes the current abort count to the instruction, which then performs a random delay optimal for the current configuration; that way software does not have to be adjusted for future machine generations or different configurations.

If the CPU is already in transactional execution mode when a TBEGIN is executed, a nested inner transaction is started, and the CPU increments the transaction nesting depth. The maximum supported nesting depth is 16. The TEND instruction closes a transaction by decrementing the current nesting depth; if the current nesting depth is 1, execution of TEND commits the transaction and the CPU leaves transactional execution mode. If a transaction abort happens on a nested transaction, the entire nest of transactions is aborted (*flattened nesting*), the nesting depth is set to 0, and execution continues at the instruction after the outermost TBEGIN.

The *Extract Transaction Nesting Depth* (ETND) instruction can be used to load the current nesting depth into a General Register (GR).

The support of nested transactions is important to software in certain cases. Compilers may produce code that calls sub-routines from within transactions, and if those sub-routines were themselves compiled to use transactions, nesting occurs. Sometimes it is important for a sub-routine to know whether it is being called transactionally, which can be done quickly with ETND by checking whether the depth is 0.

While the CPU is in transactional mode, stores performed by the CPU are not made visible to other CPUs or the I/O subsystem until the outermost TEND completes. If the transaction aborts, all stores done during the transaction are discarded. The exception are stores performed by the *Non-transactional Store* (NTSTG) instruction; these 8-byte stores are also isolated, that is, not visible to other CPUs and I/O while the transaction executes, but unlike normal stores they are committed to memory even in the case of transaction abort. The main use case for NTSTG is transactional debugging: a programmer/compiler can store intermediate results into memory and analyze the data even in the abort case to see which program path and data was observed prior to the abort (*breadcrumb debugging*). The architecture requires that the memory locations stored to by NTSTG do not overlap with other stores from the transaction.

The *Transaction Abort* (TABORT) instruction causes an immediate abort. The operand provides an abort reason code which is placed into the optional Transaction Diagnostic Block (see below). The least significant bit of the abort code determines whether the condition code is set to 2 or 3 to indicate transient versus permanent abort to the abort handler.

B. TBEGIN Control Fields

The TBEGIN instruction has a set of operand fields (see figure 2). The *General Register Save Mask* (GRSM) is an 8 bit field, each bit corresponding to an even/odd pair of the 16 General Registers (GRs). At the execution of the outermost TBEGIN, the pairs indicated with a '1' in the mask are saved, and are restored to their pre-TBEGIN content in the case of transaction abort. GR-pairs not indicated in the mask keep their current value in case of abort (this is an exception to the "all or nothing" atomicity rule, since modified state survives the abort). Saving only a subset of GRs during TBEGIN speeds up execution, and not restoring all GRs during abort provides information for debugging and analysis.

The IBM z/Architecture supports additional register sets, namely Access Registers (ARs) and Floating-Point Registers (FPRs), for which no save/restore mechanism is provided - it is up to software

to save necessary registers before entering a transaction, and to restore those registers in the abort handler.

Some software may expect ARs or FPRs to not be modified and thus does not provide save/restore in the abort handler. But when calling a sub-routine — potentially in a linked library — the sub-routine might inadvertently change one of those registers. Even worse, a sub-routine might for example use an AR as scratch space and later restore it. This is not observable if the transaction does not abort, but in the (rare) case of an abort while the register is modified, unpredictable results may occur due to modified register content after the abort. Such failures may be rare and extremely difficult to debug. To protect against such situations, the TBEGIN instruction provides AR and FPR modification control bits. Any instruction attempting to modify an AR or FPR, respectively, leads to a restricted-instruction abort if the respective control bit is '0'. For nested transactions, the effective control is the 'AND' of all control bits in the nest.

C. Interruption Filtering

When a program-exception condition is detected during normal program execution, an interruption into the operating system occurs. The PSW at which the exception is detected is stored as *program-old PSW*, and a *program-new PSW* pointing to the OS interrupt handler is loaded. The OS can then service the interrupt (e.g. page in memory from disk) and then return to program execution by loading the saved program-old PSW into the PSW. (The same concept is used for other interruptions like I/O, but we omit the details here.)

The transactional memory architecture provides control over whether certain exceptions detected during a transaction actually lead to a program interruption into the OS, or whether the interruption is *filtered*. In both cases the transaction is first aborted. If an interruption into the OS occurs, the program-old PSW will point to the instruction after the TBEGIN with a non-zero condition code. That way, when the OS returns control to the program, the program knows to execute the abort handler before potentially retrying the transaction. In the case of a filtered interruption, the condition code is also non-zero, and the program continues execution after the outermost TBEGIN without first trapping into the OS.

Exceptions are categorized into 4 groups. First, some exceptions can never occur during transactional execution, for example since they are related to specific instructions that are restricted in transactions. Second is a group of exceptions that is always considered a programming error and thus is always causing interruption into the OS; examples include undefined instruction op-codes in the instruction stream. Third is a group of exceptions related to accessing of memory, for example page fault exceptions. Fourth is a group of exceptions related to arithmetic and data, e.g. divide-by-zero or overflow exceptions. The third and fourth group of exceptions can be filtered under the control of the *Program Interruption Filtering Control* (PIFC) field of the TBEGIN instruction. Values 0 to 2 in this field correspond to no filtering, filtering of group 4 only, and filtering of groups 3 and 4, respectively. In a nested transaction, the effective PIFC is the highest value of all TBEGINs in the nest.

Interruption filtering is useful in many speculative program optimizations, e.g. by not performing null-pointer checks before accessing data, or not performing NaN or zero checks before performing computations. Instead these (depending on the program)

```
TBEGINC   *begin constrained transaction
...perform operation...
TEND      *end transaction
...
```

Figure 3: Constrained transaction example

rare conditions can be treated in the transaction abort handler, improving performance for the normal case and penalizing the rare case only. Of course null-pointer or NaN checking are only examples; once the infrastructure is available, the compiler can perform general if-then-else speculation and other optimization using the same concepts [14].

It is important for the program to adhere to certain rules when using interruption filtering. For example, a filtered page fault encountered during a transaction is not reported to the OS; if the abort handler does not access the same memory locations non-transactionally, the program may never trap the page fault into the OS, the page fault will never be resolved, and thus the transaction continues to fail every time it is executed. Exceptions related to instruction fetching are never filtered. The program-old PSW will indicate a transient abort condition code of 2, so that the program usually repeats the transaction immediately after the OS handled the interrupt. If instruction fetching exceptions were filtered, a page fault on an instruction page that is only used during transactional execution would never be resolved by the OS and always cause that transaction code to abort.

D. Constrained Transactions

Transactions started with TBEGIN are not assured to ever successfully complete with TEND, since they can experience an aborting condition at every attempted execution, e.g. due to repeating conflicts with other CPUs. This requires that the program supports a fallback path to perform the same operation non-transactionally, e.g. by using traditional locking schemes. This puts significant burden on the programming and software verification teams, especially where the fallback path is not automatically generated by a reliable compiler.

Many transactions operating on shared data structures are expected to be short, touch only few distinct memory locations, and use simple instructions only. For those transactions, the concept of *constrained transactions* is available; under normal conditions, the CPU assures that constrained transactions eventually end successfully, albeit without giving a strict limit on the number of necessary retries. A constrained transaction starts with a TBEGINC instruction and ends with a regular TEND. Implementing a task as constrained or non-constrained transaction typically results in very comparable performance, but constrained transactions simplify software development by removing the need for a fallback path.

A transaction initiated with TBEGINC must follow a list of programming constraints; otherwise the program takes a non-filterable constraint-violation interruption. The constraints include: the transaction can execute a maximum of 32 instructions, all instruction text must be within 256 consecutive bytes of memory; the transaction contains only forward-pointing relative branches (hence no loops or sub-routine calls); the transaction can access a maximum of 4 aligned octowords (32 bytes) of memory; and restriction of the instruction-set to exclude complex instructions

like decimal or floating-point operations. The constraints are chosen such that many common operations like double-linked list-insert/delete operations can be performed, including the very powerful concept of atomic compare-and-swap targeting up to 4 aligned octowords. At the same time the constraints were chosen conservatively such that future CPU implementations can assure transaction success without needing to adjust the constraints, since that would otherwise lead to software incompatibility.

TBEGINC mostly behaves like TBEGIN, except that the FPR control and the program interruption filtering fields do not exist and the controls are considered to be zero. On a transaction abort, the instruction address is set back directly to the TBEGINC instead to the instruction after, reflecting the immediate retry and absence of an abort path for constrained transactions.

Nested transactions are not allowed within constrained transactions, but if a TBEGINC occurs within a non-constrained transaction it is treated as opening a new non-constrained nesting level just like TBEGIN would. This can occur e.g. if a non-constrained transaction calls a sub-routine that uses a constrained transaction internally.

Since interruption filtering is implicitly off, all exceptions during a constrained transaction lead to an interruption into the OS. Eventual successful finishing of the transaction of course relies on the capability of the OS to page-in the at most 4 pages touched by any constrained transaction. The OS must also ensure time-slices long enough to allow the transaction to complete.

Figure 3 shows the constrained-transactional implementation of the code in figure 1, assuming that the constrained transactions does not interact with other locking-based code. No lock testing is shown therefore, but could, of course, be added if constrained transactions and lock-based code were mixed.

E. Debugging Features

Reliable software is essential for enterprise class computing, and transactional memory poses interesting challenges to how software debugging and testing is performed during the software development cycle and during field failure analysis. Significant effort was spent on the development of architectural features to support debugging and testing.

1) Transaction Diagnostic Block: The TBEGIN instruction has an optional address operand called the *Transaction Diagnostic Block (TDB) Address*. The TDB is not used during normal transaction processing, but if a transaction aborts and a TDB Address is specified on the outermost TBEGIN, detailed information about the abort is stored in the TDB. The TDB is 256 bytes in length, and its fields include: (i) Transaction Abort Code, indicating the detailed reason for the abort; (ii) Conflict Token, providing the address that caused a conflict with another CPU; this field cannot always be provided and there is a bit indicating the validity; (iii) Aborted-Transaction Instruction Address, indicating the IA at which the abort was detected; (iv) Exception information like Program Interruption Code and Translation Exception Address; (v) the content of all GRs at the time of abort; and (vi) CPU specific information not formally architected. The last item provides detailed CPU-generation dependent information on the details of why the transaction aborted, and which path the program took from the outermost TBEGIN to the abort IA.

It is expected that extracting the information and storing the TDB on transaction abort takes a number of CPU cycles, and thus only code in debug/test or with extremely low abort rates will enable TDBs on performance-sensitive transactions. During initial hardware validation of the transactional facility, the information in the TDB was invaluable for debugging test program and hardware/firmware problems. We expect similar usefulness for debugging application code.

A second copy of the TDB is stored into the processor prefix area (a memory area containing reserved locations specific for each CPU in the system) on every abort due to a program interruption; this is valuable for post-mortem failure analysis after a program ended abnormally, e.g. on an access exception.

2) PER: In z/Architecture, traditionally Program Event Recording provides a hardware mechanism to trigger a program interruption for certain events. The supported events include stores into a specified memory range, execution of instructions from a specified memory range, and branching into a specified memory range. This mechanism is used extensively for software debugging, for example in z/OS SLIP traps, or in GDB under Linux for setting break- or watch-points.

Detection of a PER event inside a transaction causes a transaction abort and a non-filterable interruption into the OS. Two new features are added to PER for transactional memory: (i) *PER Event Suppression* suppresses any PER event while running in transactional mode, and (ii) the new *PER TEND event* triggers on successful execution of an outermost TEND instruction.

For example, if a debugger is running in single instruction mode, a PER instruction-fetch event is enabled for the entire address range. PER event suppression can be used to avoid aborting every single transaction on the first instruction after the TBEGIN. This effectively makes entire transactions look like single ”big instructions” in the single-step mode.

Another use case of PER is monitoring for stores into a specific memory range for implementing watch-points. Without event suppression, a transaction modifying memory in the monitored range always aborts and eventually takes the fallback path. To enable debugging of the transactional code itself, event suppression can be enabled alongside the new PER TEND-event, which triggers at the ending of every transaction. The debugger can then check all active watch-points for whether the memory content changed and enter the interactive debugging mode in that case.

For constrained transactions, it is up to the OS to enable event suppression after a PER event caused a transaction abort, in order to enable the transaction to complete on the next retry. The OS can use PER TEND to disable event suppression after successful transaction completion.

3) Transaction Diagnostic Control: Most transactions will abort only infrequently, and the point of abort inside the transaction may be non-uniformly distributed. For example, certain instructions cause conflicts with other CPUs more frequently than other instructions. This creates unique debugging and testing challenges. The abort path and fallback path might be sparsely exercised leading to poor testing coverage. Also, the random distribution of the abort point may lead to unusual corner cases after the abort if residual state survives the abort (for example non-restored registers). This may lead to program failures that are very hard to reproduce and to debug.

In order to enhance the testing coverage of the abort path and to protect against untested corner cases, the Transaction Diagnostic Control is provided to force random aborts. At one setting, the CPU is instructed to often, randomly abort transactions at a random point. At a more aggressive setting, the CPU is instructed to abort every transaction at a random point but at latest before the outermost TEND instruction. The latter setting can be used to stress the reaching of the retry-threshold and force the non-transactional fallback path to be used. This more aggressive setting is treated like the less aggressive setting for constrained transactions. The Transaction Diagnostic Control can be enabled by the OS for testing specific programs.

III. Implementation

The main implementation components of the transactional memory facility are a transaction-backup register file for holding pre-transaction GR content, a cache directory to track the cache lines accessed during the transaction, a store cache to buffer stores until the transaction ends, and firmware routines to perform various complex functions. In this section we describe the detailed implementation.

A. System Background

The transactional execution facility is first implemented in the IBM zEC12 processor [1], the successor of the z196 processor described in [20]. The processor can decode 3 instructions per clock cycle; simple instructions are dispatched as single micro-ops, and more complex instructions are cracked into multiple micro-ops. The micro-ops are written into a unified issue queue, from where they can be issued out-of-order. Up to two fixed-point, one floating-point, two load/store, and two branch instructions can execute every cycle. A Global Completion Table (GCT) holds every micro-op. The GCT is written in-order at decode time, tracks the execution status of each micro-op, and completes instructions when all micro-ops of the oldest instruction group have successfully executed.

The L1 data cache is a 96KB 6-way associative cache with 256 byte cache-lines and 4 cycle use-latency, coupled to a private 1MB 8-way associative 2nd-level data cache with 7 cycles use-latency penalty for L1 misses. Both L1 and L2 caches are store-through. Six cores on each *CP chip* share a 48MB 3rd-level store-in cache, and six CP chips are connected to an off-chip 384MB 4th-level cache, packaged together on a glass-ceramic multi-chip module (MCM). Up to 4 MCMs can be connected to a coherent SMP system with up to 144 cores (not all cores are available to run customer workload).

Coherency is managed with a variant of the MESI protocol. Cache-lines can be owned read-only (shared) or exclusive; the L1 and L2 are store-through and thus do not contain dirty lines. The L3 and L4 caches are store-in and track dirty states. Each cache is inclusive of all its connected lower level caches.

Coherency requests are called *cross interrogates* (XI) and are sent hierarchically from higher-level to lower-level caches, and between the L4s. When one core misses the L1 and L2 and requests the cache line from its local L3, the L3 checks whether it owns the line, and if necessary sends an XI to the currently owning L2/L1 under that L3 to ensure coherency, before it returns the cache line to the requestor. If the request also misses the L3, the L3 sends a request to the L4 which enforces coherency by sending XIs to all necessary L3s under that L4, and to the neighboring L4s. Then the L4 responds to the requesting L3 which forwards the response to the L2/L1.

Note that due to the inclusivity rule of the cache hierarchy, sometimes cache lines are XI'ed from lower-level caches due to evictions on higher-level caches caused by associativity overflows from requests to other cache lines. We call those XIs *LRU XIs*.

Demote-XIs transition cache-ownership from exclusive into read-only state, and Exclusive-XIs transition cache-ownership from exclusive into invalid state. Demote- and Exclusive-XIs need a response back to the XI sender. The target cache can accept the XI, or send a reject response if it first needs to evict dirty data before accepting the XI. The L1/L2 are store through, but may reject demote- and exclusive XIs if they have stores in their store queues that need to be sent to L3 before downgrading the exclusive state. A rejected XI will be repeated by the sender. Read-only-XIs are sent to caches that own the line read-only; no response is needed for such XIs since they cannot be rejected. The details of the SMP protocol are very similar to those described for the IBM z10 in [21].

B. Transactional Instruction Execution

The instruction decode unit (IDU) keeps track of the current transaction nesting depth (TND, see figure 4). When the IDU receives a TBEGIN instruction, the nesting depth is incremented, and conversely decremented on TEND instructions. The nesting depth is written into the GCT for every dispatched instruction. When a TBEGIN or TEND is decoded on a speculative path that later gets flushed, the IDU's nesting depth is refreshed from the youngest GCT entry that is not flushed. The transactional state is also written into the issue queue for consumption by the execution units, mostly by the Load/Store Unit (LSU).

Similar to the nesting depth, the IDU/GCT collaboratively track the AR/FPR-modification masks through the transaction nest; the IDU can place an abort-request into the GCT when an AR/FPR-modifying instruction is decoded and the modification mask blocks that. When the instruction becomes next-to-complete, completion is blocked and the transaction aborts. Other restricted instructions are handled similarly, including TBEGIN if decoded while in a constrained transaction, or exceeding the maximum nesting depth.

An outermost TBEGIN is cracked into multiple micro-ops depending on the GR-Save-Mask; each micro-op will be executed by one of the two FXUs to save a pair of GRs into a special transaction-backup register file, that is used to later restore the GR content is case of a transaction abort. Also the TBEGIN spawns micro-ops to perform an accessibility-test for the TDB if one is specified; the address is saved in a special purpose register for later usage in the abort case. At the decoding of an outermost TBEGIN, the instruction address and the instruction text of the TBEGIN are also saved in special purpose registers for a potential abort processing later on.

TEND and NTSTG are single micro-op instructions; NTSTG is handled like a normal store except that it is marked as non-transactional in the issue queue so that the LSU can treat it appropriately. TEND is a no-op at execution time, the ending of the transaction is performed when TEND completes.

As mentioned, instructions that are within a transaction are marked as such in the issue queue, but otherwise execute mostly

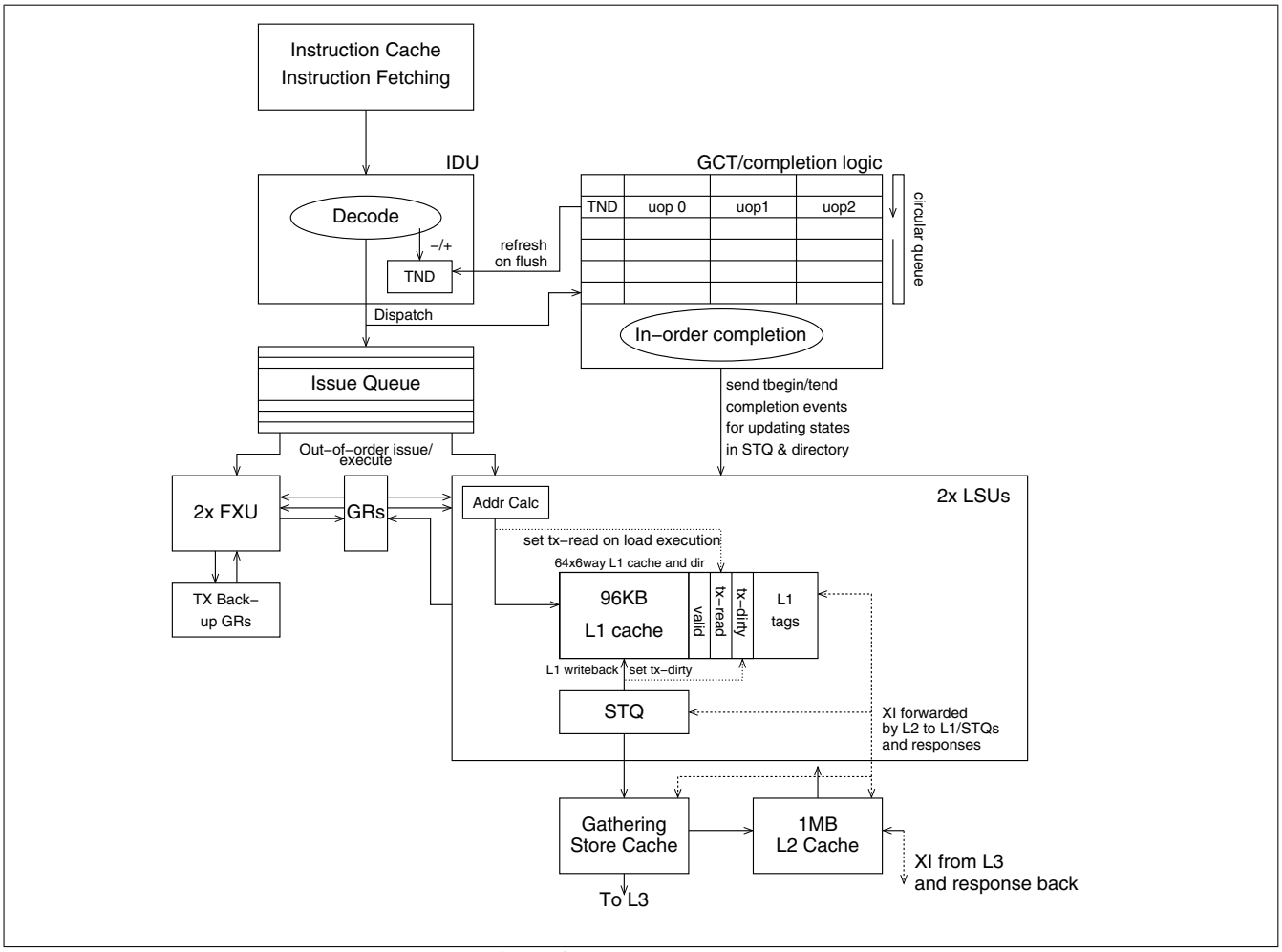


Figure 4: Block diagram of CPU

unchanged; the LSU performs isolation tracking as described in the next section.

Since decoding is in-order, and since the IDU keeps track of the current transactional state and writes it into the issue queue along with every instruction from the transaction, execution of TBEGIN, TEND, and instructions before, within, and after the transaction can be performed out-of-order. It is even possible (though unlikely) that TEND is executed first, then the entire transaction, and lastly the TBEGIN executes. Of course program order is restored through the GCT at completion time. The length of transactions is not limited by the size of the GCT, since GRs can be restored from the backup register file.

During execution, the PER events are filtered based on the Event Suppression Control, and a PER TEND event is detected if enabled. Similarly, while in transactional mode, a pseudo-random generator may be causing the random aborts as enabled by the Transaction Diagnostics Control.

C. Tracking for Transactional Isolation

The Load/Store Unit tracks cache lines that were accessed during transactional execution, and triggers an abort if an XI from another CPU (or an LRU-XI) conflicts with the footprint. If the conflicting XI is an exclusive or demote XI, the LSU rejects the XI back to the L3 in the hope of finishing the transaction before the L3 repeats the XI. This *stiff-arming* is very efficient in highly contended transactions. In order to prevent hangs when two CPUs stiff-arm each other, a XI-reject counter is implemented, which triggers a transaction abort when a threshold is met.

The L1 cache directory is traditionally implemented with SRAMs. For the transactional memory implementation, the valid bits (64 rows x 6 ways) of the directory have been moved into normal logic latches, and are supplemented with two more bits per cache line: the tx-read and tx-dirty bits.

The tx-read bits are reset when a new outermost TBEGIN is decoded (which is interlocked against a prior still pending transaction). The tx-read bit is set at execution time by every load instruction that is marked transactional in the issue queue. Note that this can lead to over-marking if speculative loads are executed, for example on a mispredicted branch path. The alternative of setting the tx-read bit at load completion time was too expensive for silicon area, since multiple loads can complete at the same time, requiring many read-ports on the load-queue.

Stores execute the same way as in non-transactional mode, but a transaction mark is placed in the store queue (STQ) entry of the store instruction. At writeback time, when the data from the STQ is written into the L1, the tx-dirty bit in the L1-directory is set for the written cache line. Store writeback into the L1 occurs only after the store instruction has completed, and at most one store is written back per cycle. Before completion and writeback, loads can access the data from the STQ by means of store-forwarding; after write-back, the CPU can access the speculatively updated data in the L1. If the transaction ends successfully, the tx-dirty bits of all cache-lines are cleared, and also the tx-marks of not yet written stores are cleared in the STQ, effectively turning the pending stores into normal stores.

On a transaction abort, all pending transactional stores are invalidated from the STQ, even those already completed. All cache lines that were modified by the transaction in the L1, that is, have the tx-dirty bit on, have their valid bits turned off, effectively removing them from the L1 cache instantaneously.

As described in Section 2, the architecture requires that before completing a new instruction we ensure that isolation of the transaction read- and write-set is maintained. This is ensured by stalling instruction completion at appropriate times when XIs are pending; we allow speculative out-of-order execution, optimistically assuming that the pending XIs are to different addresses and not actually cause a transaction conflict. This design fits very naturally with the XI-vs-completion interlocks that are implemented on prior systems to ensure the strong memory ordering that the architecture requires [22].

When the L1 receives an XI, it accesses the directory to check validity of the XI'ed address in the L1, and if the tx-read bit is active on the XI'ed line and the XI is not rejected, the LSU triggers an abort. When a cache line with active tx-read bit is LRU'ed from the L1, a special LRU-extension vector remembers for each of the 64 rows of the L1 that a tx-read line existed on that row. Since no precise address tracking exists for the LRU extensions, any non-rejected XI that hits a valid extension row the LSU triggers an abort. Providing the LRU-extension effectively increases the read footprint capability from the L1-size to the L2-size and associativity, provided no conflicts with other CPUs against the non-precise LRU-extension tracking causes aborts; section 4 contains statistical analysis of the effectiveness of the LRU extension.

The store footprint is limited by the store cache size (next section) and thus implicitly by the L2 size and associativity. No LRU-extension action needs to be performed when a tx-dirty cache line is LRU'ed from the L1.

D. Store Cache

In prior systems, since the L1 and L2 are store-through caches, every store instruction causes an L3 store access; with now 6 cores per L3 and further improved performance of each core, the store rate for the L3 (and to a lesser extent for the L2) becomes problematic for certain workloads. In order to avoid store queuing delays a gathering store cache had to be added, that combines stores to neighboring addresses before sending them to the L3.

For transactional memory performance, it is acceptable to kill every tx-dirty cache line from the L1 on transaction aborts, because the L2 cache is very close (7 cycles L1 miss penalty) to bring back the clean lines. It would however be unacceptable for performance (and silicon area for tracking) to have transactional stores write the L2 before the transaction ends and then kill all dirty L2 cache lines on abort (or even worse on the shared L3).

The two problems of store bandwidth and transactional memory store handling can both be addressed with the gathering store cache. The cache is a circular queue of 64 entries, each entry holding 128 bytes of data with byte-precise valid bits. In non-transactional operation, when a store is received from the LSU, the store cache checks whether an entry exists for the same address, and if so gathers the new store into the existing entry. If no entry exists, a new entry is written into the queue, and if the number of free entries falls under a threshold, the oldest entries are written back to the L2 and L3 caches.

When a new outermost transaction begins, all existing entries in the store cache are marked *closed* so that no new stores can be gathered into them, and eviction of those entries to L2 and L3 is started. From that point on, the transactional stores coming out of the LSU STQ allocate new entries, or gather into existing transactional entries. The writeback of those stores into L2 and L3 is blocked, until the transaction ends successfully; at that point subsequent (post-transaction) stores can continue to gather into existing entries, until the next transaction closes those entries again.

The store cache is queried on every exclusive or demote XI, and causes an XI reject if the XI compares to any active entry. If the core is not completing further instructions while continuously rejecting XIs, the transaction is aborted at a certain threshold to avoid hangs.

The LSU requests a transaction abort when the store cache overflows. The LSU detects this condition when it tries to send a new store that cannot merge into an existing entry, and the entire store cache is filled with stores from the current transaction. The store cache is managed as a subset of the L2: while transactionally dirty lines can be evicted from the L1, they have to stay resident in the L2 throughout the transaction. The maximum store footprint is thus limited to the store cache size of 64 x 128 bytes, and it is also limited by the associativity of the L2. Since the L2 is 8-way associative and has 512 rows, it is typically large enough to not cause transaction aborts.

If a transaction aborts, the store cache is notified and all entries holding transactional data are invalidated. The store cache also has a mark per doubleword (8 bytes) whether the entry was written by a NTSTG instruction - those doublewords stay valid across transaction aborts.

E. Millicode-implemented functions

Traditionally, IBM mainframe server processors contain a layer of firmware called millicode which performs complex functions like certain CISC instructions, interruption handling, system synchronization, and RAS. Firmware resides in a restricted area of main memory that customer programs cannot access. When hardware detects a situation that needs to invoke millicode, the instruction fetching unit switches into *millicode mode* and starts fetching at the appropriate location in the millicode memory area.

For transactional memory, millicode is involved in various complex situations. Every transaction abort invokes a dedicated millicode sub-routine to perform the necessary abort steps. The

transaction-abort millicode starts by reading special-purpose registers (SPRs) holding the hardware-internal abort reason, potential exception reasons, and the aborted instruction address, which millicode then uses to store a TDB if one is specified. The TBEGIN instruction text is loaded from an SPR to obtain the GR-save-mask, which is needed for millicode to know which GRs to restore. The CPU supports a special millicode-only instruction to read out the backup-GRs and copy them into the main GRs. The TBEGIN instruction address is also loaded from an SPR to set the new instruction address in the PSW to continue execution after the TBEGIN once the millicode abort sub-routine finishes. That PSW may later be saved as program-old PSW in case the abort is caused by a non-filtered program interruption.

The TABORT instruction is millicode implemented; when the IDU decodes TABORT, it instructs the instruction fetch unit to branch into TABORT's millicode, from which millicode branches into the common abort sub-routine.

The Extract Transaction Nesting Depth (ETND) instruction is also millicoded, since it is not performance critical; millicode loads the current nesting depth out of a special hardware register and places it into a GR.

The PPA instruction is millicoded; it performs the optimal delay based on the current abort count provided by software as an operand to PPA, and also based on other hardware internal state.

For constrained transactions, millicode keeps track of the number of aborts. The counter is reset to 0 on successful TEND completion, or if an interruption into the OS occurs (since it is not known if or when the OS will return to the program). Depending on the current abort count, millicode can invoke certain mechanisms to improve the chance of success for the subsequent transaction retry. The mechanisms involve, for example, successively increasing random delays between retries, and reducing the amount of speculative execution to avoid encountering aborts caused by speculative accesses to data that the transaction is not actually using. As a last resort, millicode can broadcast to other CPUs to stop all conflicting work, retry the local transaction, before releasing the other CPUs to continue normal processing. Multiple CPUs must be coordinated to not cause deadlocks, so some serialization between millicode instances on different CPUs is required.

IV. Performance Evaluation

We conducted a set of experiments to measure the performance of transactional memory in comparison to lock-based concurrency, over a range of realistic and artificial conditions. We used micro-benchmarks for these experiments since application-level transactional memory exploitation is still in development. The benchmarks use different pools of shared variables ranging from a single variable to 10k variables, each on a separate cache line. Each CPU repeatedly picks either 1 or 4 random variables from the pool and increments the chosen variable(s). If the pool consists of only 1 variable, we use 4 consecutive cache lines for the tests that update 4 variables.

We use both coarse and fine grained locking for comparison with transactional memory. For coarse-grained locking, we use a single lock for the entire pool. For fine-grained locking, we define a lock for each variable, each lock sitting on a separate cache line. In both cases we use a simple mutex algorithm, which first tests the lock to be empty and spins if necessary, then uses compare-and-swap to set the lock, which starts over if not successful; the unlock uses a simple store to unset the lock.

For non-constrained transactions, we use the code from figure 1; for the fallback lock we use the single coarse-grained lock in all experiments. The constrained transaction code from figure 3 does not need fallback locks.

Each CPU independently picks random variables and performs the incrementing on the shared variables. We use the *Store Clock Fast* instruction to measure the time between each lock/tbegin and unlock/tend, but exclude the overhead such as random number generation from the results. The overhead is significant for small numbers of CPUs since the path length for lock/update/unlock is very short compared to computing four random variables. From the measured times we compute the system throughput as the quotient of the number of CPUs divided by the average time per update. All results are normalized to a throughput of 100 for 2 CPUs concurrently updating a single variable from a pool of 1 variable.

Contention for most objects is relatively low in typical commercial applications, and if an object is accessed once it likely will be accessed again by the same CPU. As a result, many lock obtain/release operations are performed with L1-cache hits and thus are very fast. It was important in the design of transactional memory that starting and ending a transaction has similar overhead as locking and releasing a lock that is in the L1-cache; otherwise any performance gain from better behavior on contended locks could have been eroded by non-contended locks. Our experiments cover this case by having only a single CPU participate, and by setting the pool size to a single cache line. In that experiment, transactions outperform locks by 30%. This is mostly due to the longer path length of the lock and release code. The results also show that the overhead of testing the lock in the non-constrained transaction (see figure 1) is insignificant since the branch is perfectly predictable in this case; the performance difference between constrained and non-constrained transactions is 0.4%.

One major drawback of lock-based methods is the complexity involved with fine-grained locking. For example, in the case of updating 4 random variables, the programmer would have to ensure that locks are acquired in a certain order to prevent deadlocks, which in practice can be very hard, e.g. when the objects involved in an operation are not all known a priori. Thus one major use case of transactions is to allow fine grained concurrency in cases where fine-grained locking is hard to achieve. Figure 5(a) shows the performance of updating 4 random variables from a large pool (1k and 10k entries), using transactions versus using a coarse lock. For small numbers of CPUs, the performance grows slightly as CPUs are added since the entire pool does not fit into a single CPU's cache, and some cache miss penalty can be hidden under the lock-waiting. But as expected, coarse grain locking leads to very poor throughput when the number of CPUs grows further (note the step-functions as the number of CPUs crosses the chip and MCM boundaries). In contrast, transactions scale very well. Even at 100 CPUs, the performance is not limited by the concurrency, but by the cache miss penalty that almost every iteration incurs when accessing a cache line that was previously accessed by another CPU: at 100 CPUs, the throughput with TBEGINC is 99.8% of the throughput without any locking scheme.

While not particularly interesting for real-world commercial applications, we also studied the performance of transactions versus

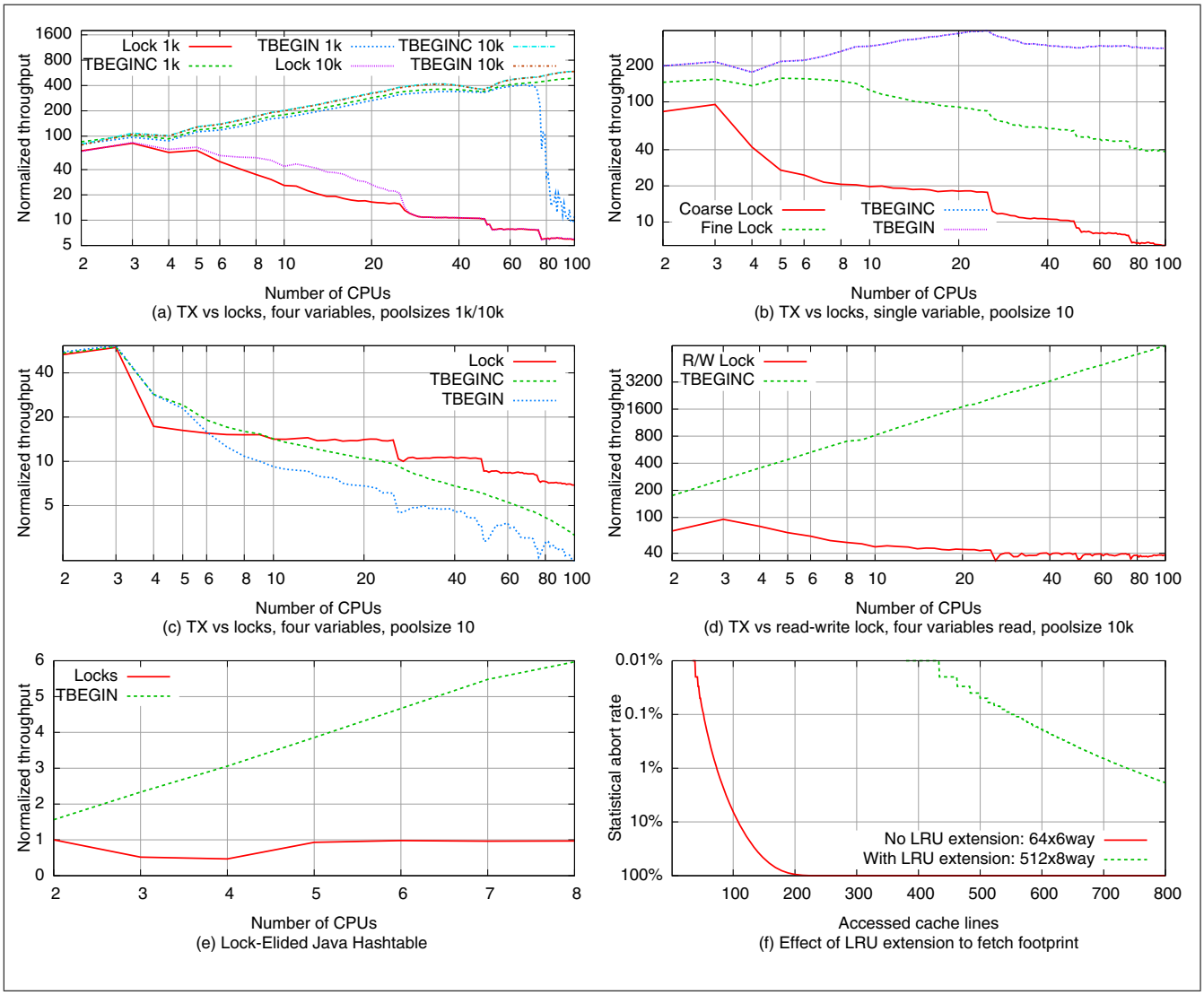


Figure 5: Performance Results

locks for very high contention cases. As can be seen in figure 5(a), when using a pool of 1000 lines, the throughput using TBEGIN drops steeply after the number of CPUs reaches a threshold, but still exceeds the locking performance. Figure 5(b) shows the performance of updating a single variable from a very small pool of only 10 variables. As expected, coarse grain locks yield very poor throughput. The throughput is better with fine grained locks, although it does not grow much with the number of CPUs and declines with more than 10 CPUs. In contrast, with transactions, the throughput grows up to 24 CPUs (the size of the MCM node in the tested system) and holds steady beyond, and transactions out-perform locks across the entire CPU range. Figure 5(c) shows updating of 4 variables from a pool of 10. With up to 6 CPUs, transactions behave slightly better than a coarse grained lock. But as the number of CPUs and such the contention grows further, locks perform better, not dropping as steeply as transactions. The reason for the difference between single-variable and four-variable updates is that for the latter case a CPU must receive all 4 lines into the L1 cache before it can commit the transaction; after the first line is received, the increment instruction for that variable can execute out-of-order. The transaction then becomes subject to conflicts on that cache line while the CPU is waiting for the other cache lines. This leads to a high abort rate, which means that many cache line transfers in the system are wasted. In contrast, as soon as a CPU obtains a lock, that CPU is guaranteed to finish the update on all 4 lines and thus less cache bandwidth is wasted.

It is interesting to note that under extreme contention, constrained transactions behave better than non-constrained transactions. This is because the CPU turns off speculative fetching after a certain number of aborts for constrained transactions, preserving some cache bandwidth which helps throughput. We did not implement this feature for performance reasons, but in order to guarantee eventual success for constrained transactions.

The above test cases all update shared variables. Another impor-

tant case is reading of shared variables. Traditionally a read-write lock is used if updates are relatively rare. This allows shared access to the variables by multiple CPUs while no updates are in progress. Typical implementations of read-write locks require updating of the lock-word every time a reader enters or leaves its critical section, in order to keep track of how many readers are in-flight. The update of the read-count causes the lock-word to be transferred between CPUs, which limits the throughput significantly (see figure 5(d)). Transactions avoid this problem since they only need to check the write-count to be 0, without updating the read-count. If a writer enters during the transaction, the writer's update of the write-count causes all reader transactions to abort, but as long as no writers appear, all CPUs can share the read/write count cache line. This leads to almost linear performance improvement with the number of CPUs.

Figure 5(e) shows the performance of a more real-world example. The IBM Java team has prototyped an optimization in the IBM Testarossa JIT to automatically elide locks used for Java synchronized sections. This optimization has been shown to transparently improve the scalability of widely used standard data structures such as `java/util/hashtable`. Multiple software threads run under z/OS, accessing the hash table for reading and writing. As can be seen in figure 5(e), the performance using locks is flat, whereas the performance grows almost linearly with the number of threads using transactions.

In another experiment (not shown in figure 5), the Java team has implemented the `ConcurrentLinkedQueue` using constrained transactions. The throughput using transactions exceeds locks by a factor of 2. In [23], the IBM XL C/C++ team compares a subset of the STAMP benchmarks using pthread locks and transactions. Depending on the benchmark application, transactional execution improves performance by factors between 1.2 and 7.

As described in section 3.3, the L1 cache employs a LRU-extension scheme to enhance the supported fetch footprint beyond the L1 cache size. Figure 5(f) shows the statistical abort rate (%) from associativity conflicts with n=1...800 accesses to random congruence classes. As can be seen, the abort rate for large transactions is significantly reduced when the footprint limitation is moved from the L1 cache to the L2 cache, as is done by the L1 cache LRU extension scheme. Of course, very large and long transactions may suffer from other abort reasons like conflicts with other CPUs, LRU evictions from higher level caches, or asynchronous interrupts. These effects limit the practical transaction size. Learning over time will show exactly how to best tailor transactions, but we feel that with the LRU extension, the read footprint will not be a limitation.

V. Summary

We have described the instruction-set architecture and implementation of the transactional memory feature of the latest mainframe server processor in the IBM zEC12 system. Special focus was put on software test and debug, as well as the introduction of transactional memory support into an existing SMP and microprocessor design. The transactional memory feature is defined so that integration into existing large-scale software products can be done without a complete software redesign. For example we have shown how transactions and locks can co-exist by eliding locks using transactions. The introduction of constrained transactions eases the explitation of transactional memory be removing the need for a lock-based fallback path.

We have evaluated the performance on a set of micro-benchmarks, and under realistic contention, the performance of the transactional memory system meets our expectations and clearly exceeds the performance of traditional lock based methods, in some cases significantly. Preliminary performance experiments with examples like a parallel hash table are very promising and show almost linear n-way scalability.

The IBM compiler development teams are involved in ongoing development for support of transactional memory in various programming languages. In [24], the support of transactional memory in IBM's XL C/C++ compiler is described. The operating systems and middleware development groups are aggressively identifying opportunities to improve the scaling of hot-spots. Transactional memory is a very promising new tool for improving parallel software scalability, that is driving innovation in both hardware and software design now and in the future.

References

[1] http://www-03.ibm.com/press/us/en/pressrelease/38653.wss

[2] K. Shum, IBM zNext – The 3rd Generation High Frequency Microprocessor Chip, HotChips, 2012

[3] M. Herlihy, J. Eliot, B. Moss, Transactional Memory: Architectural Support for Lock-Free Data Structures, ISCA, 1993

[4] L. Hammond et al., Transactional Memory Coherence Consistency, ISCA, 2004

[5] K. Moore, J. Bobba, M. Moravan, M. Hill and D. Wood, LogTM Log-Based Transactional Memory, HPCA, 2006

[6] S. Chaudhry et al., Rock: A high-performance Sparc CMT processor. IEEE Micro, 29(2):6-16, 2009

[7] R. Haring et al., The IBM Blue Gene/Q Compute Chip. IEEE Micro 32(2): 48-60 (2012)

[8] http://software.intel.com/en-us/blogs/2012/02/07/transactional-synchronization-in-haswell/

[9] A. McDonald et al., Architectural Semantics for Practical Transactional Memory, ISCA, 2006

[10] J. Chung et al., Tradeoffs in Transactional Memory Virtualization, ASPLOS, 2006

[11] T. Harris et al., Transactional Memory: An Overview, IEEE Micro 27:3, May 2007

[12] R. Rajwar, J. Goodman. Speculative lock elision: enabling highly concurrent multithreaded execution. MICRO, 2001

[13] M. Herlihy. A methodology for implementing highly concurrent data objects. In ACM Transactions on Programming Languages and Systems 15:5, 1993

[14] N. Neelakantam, R. Rajwar, S. Srinivas, U. Srinivasan, C. Zilles. Hardware Atomicity for Reliable Software Speculation. ISCA, 2007

[15] Y. Lev, M. Moir. Debugging with transactional memory. TRANSACT '06: 1st Workshop on Languages, Compilers, and Hardware Support for Transactional Computing, 2006.

[16] z/Architecture Principles of Operation, IBM Publication SA22-7832-09, 2012

[17] R. Guerraoui, M. Kapalka. On the correctness of transactional memory. Symposium on Principles and Practice of Parallel Programming (PpoPP), 2008

[18] D. Dice, O. Shalev, N. Shavit, Transactional locking II, DISC 06: 20th International Symposium on Distributed Computing, 2006

[19] C. Blundell, E. C. Lewis, and M. Martin. Deconstructing transactions: The subtleties of atomicity. WDDD 05: 4th Annual Workshop on Duplicating, Deconstructing, and Debunking, 2005

[20] F. Busaba, M. Blake, B. Curran, M. Fee, C. Jacobi, P.-K. Mak, B. Prasky, C. Walters, IBM zEnterprise 196 microprocessor and cache subsystem, IBM Journal of Research and Development, Vol 56:1.2, 2012

[21] P. Mak, C. Walters, G. Strait, IBM System z10 processor cache subsystem microarchitecture, IBM Journal of Research and Development, Vol 53:1, 2009

[22] K. Choy, J. Navarro, C.-L. Shum, A. Tsai, Method, System, and Computer Program Product for Cross-invalidation Handling in a Multi-level Private Cache, US Patent Application 20090240889

[23] M. Mitran, V. Vokhshori, Evaluating the zEC12 Transactional Execution Facility, IBM Systems Magazine, 2012

[24] M. Mitran, V. Vokhshori, IBM XL C/C++ Maximizes zEC12's Transactional Execution Capabilities, IBM Systems Magazine, 2012

2012 IEEE/ACM 45th Annual International Symposium on Microarchitecture

Warped-DMR: Light-weight Error Detection for GPGPU

Hyeran Jeon Murali Annavaram
University of Southern California
{hyeranje,annavara}@usc.edu

Abstract

General purpose graphics processing units (GPGPUs) are feature rich GPUs that provide general purpose computing ability with massive number of parallel threads. The massive parallelism combined with programmability made GPGPUs the most attractive choice in supercomputing centers. Unsurprisingly, most of the GPGPU-based studies have been focusing on performance improvement leveraging GPGPU's high degree of parallelism. However, for many scientific applications that commonly run on supercomputers, program correctness is as important as performance. Few soft or hard errors could lead to corrupt results and can potentially waste days or even months of computing effort. In this research we exploit unique architectural characteristics of GPGPUs to propose a light weight error detection method, called Warped Dual Modular Redundancy *(Warped-DMR). Warped-DMR detects errors in computation by relying on opportunistic spatial and temporal dual-modular execution of code. Warped-DMR is light weight because it exploits the underutilized parallelism in GPGPU computing for error detection. Error detection spans both within a warp as well as between warps, called* intra-warp *and* inter-warp DMR*, respectively. Warped-DMR achieves 96% error coverage while incurring a worst-case 16% performance overhead without extra execution units or programmer's effort.*

1. Introduction

Recently GPU architectures have been enhanced with several microarchitectural features that allow GPUs to be used not only for graphics applications but also for general purpose computing. Nowhere else is this trend more visible than in super-computing centers which are adopting GPUs as processing engines to achieve massive parallel execution. Named GPGPU, the new GPU variant is known to derive superior performance over multi-core CPUs by allowing thousands of concurrent threads to run efficiently within a limited power budget. Many application developers have been attracted to this new powerful and relatively cheap parallel execution paradigm. Significant effort has been expended to port applications to GPGPU paradigm in order to achieve better performance by efficiently leveraging abundant parallel computational resources.

GPGPUs now run business-critical applications, long running scientific codes, and financial software. These new application domains demand strict program correctness [7]. A few erroneous computations or a corrupt value could have severe negative repercussions. CMOS technology scaling, while provided power and performance benefits, is also leading to significant number of reliability concerns. GPGPU will be vulnerable to soft/hard error and the vulnerability is predicted to grow exponentially [6]. Since GPGPUs evolved from GPUs, the primary focus of GPGPU design has been to increase parallel performance. In particular, reliability is considered as a secondary issue in GPU computations since traditional graphics applications have been shown to be inherently fault tolerant [7]. However, to support business critical application domains on GPGPUs, there is a need to provide architectural support for at least error detection. As a first step, error detection can translate the most harmful silent data corruption (SDCs) errors to detectable but unrecoverable errors (DUEs). In fact, commercial GPGPU designers have already started addressing reliability concerns. Recently NVIDIA's Fermi GPGPU added ECC for the memory components [16].

GPGPUs have hundreds of hardware thread contexts today and in the near future they will have thousands of contexts. Each thread context contains a relatively simple processor pipeline with very minimal resources to support speculation, if any. Hence the vast majority of the chip area is dedicated to execution units, such as ALUs. In the presence of hundreds (or even thousands) of thread contexts, even a tiny probability of a logic error in each thread context adds up to an exponentially high probability of errors at the chip level. Recognizing this concern, several researchers have been focusing on improving reliability of GPGPU computation. They are mostly software approaches [6] [22]. Software approaches can be more flexible but demand programmers to re-write their applications with focus on fault tolerance, which is quite undesirable given that writing a GPGPU application itself is non trivial [11]. Compilers could reduce some of the burden on the programmer by automatically providing redundant code execution [22]. However, the error coverage is limited by the granularity of compiler's code insertion and has the *hidden error* problem: even though each line of code is executed twice for verification purpose, if the two instances are executed on the same processing core, some hardware defects, such as stuck-at faults, cannot be detected. Note that software does not decide which thread is mapped to which core in current GPGPU architectures. Furthermore, in software approaches the results from redundant execution are mostly compared at the end of the program execution. Hence faults are likely to be discovered too late to take quick corrective action.

In this paper we propose a low-overhead hardware approach for detecting computation errors in GPGPUs. The approach uses dual modular redundancy (DMR) [13] but opportunistically switches between spatial and temporal redundancy to improve error coverage while reducing the error detection overhead. We call this approach Warped-DMR. In this paper, we assume that only execution units are vulnerable. Memory is assumed to be protected by ECC, as is done in many industrial GPU designs [16]. Hence, for the memory operations, we only verify the address computations and assume that the loaded data is always error free.

1.1. Exploiting Opportunity

Before presenting the details of Warped-DMR, we first present motivating data that shows the utilization of thread contexts on a GPGPU. Figure 1 shows the execution time breakdown in terms of the number of active threads for a subset of benchmarks selected from NVIDIA CUDA SDK [4], Parboil Benchmark Suite [5], and ERCBench [1]. These results were generated by simulating a modern NVIDIA-style GPGPU architecture using *GPGPU-Sim* [3]. More details of the simulated architecture and benchmarks are provided later in Section 5. NVIDIA GPGPU executes instructions in a batch of threads unit

1072-4451/12 $26.00 © 2012 IEEE
DOI 10.1109/MICRO.2012.13

IEEE computer society

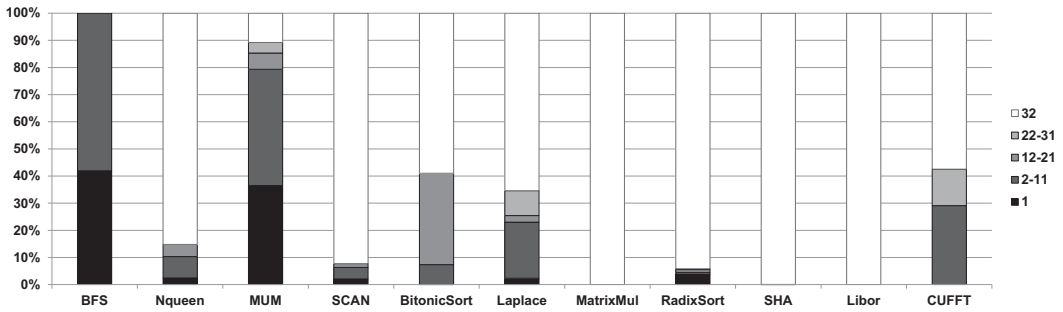


Figure 1: Execution time breakdown with respect to the number of active threads

(a.k.a. *Warp*) which consists of 32 threads. Each color in the bar chart denotes the fraction of cycles that the corresponding number of threads are actively executing the code. As can be seen, majority of applications do not have 32 active threads all the time. For example, over 40% of BFS instructions are executed by only single thread. In other words, the remaining 31 threads within the warp are idle while the single active thread is executing instructions. The processing cores associated with the idle threads remain unused during the idle period. The reasons for this underutilization are described later in Section 2.2.

The underutilization of GPGPU resources provides us an opportunity to provide error detection capability by exploiting the underutilized resources without incurring performance overheads. In this paper we present Warped-DMR, which consists of two techniques for error detection.

(1) *Intra-warp DMR:* Our first error detection method is called *intra-warp DMR*. Intra-warp DMR simply uses the inactive threads to verify execution of active threads in the same warp by using dual modular spatial redundancy. The underutilized or idle cores are used as computational checkers for a subset of active threads. Hence the overhead of intra-warp DMR is nearly zero, with the exception of a negligible area overhead needed to compare the computation results and duplicate the input data.

(2) *Inter-warp DMR:* The second error detection approach is called *inter-warp DMR*. When a warp is fully utilized, all 32 threads are active in a warp, and hence there is no opportunity for intra-warp DMR. In this scenario we use dual modular temporal redundancy. A duplicated execution of each fully utilized warp is scheduled for execution later whenever the associated execution unit becomes idle. A special purpose *Replay Queue*(ReplayQ) is used for the purpose of buffering the duplicate execution warps. We also shuffle the execution of redundant threads onto different cores, compared to the original thread-to-core assignment, to reduce the hidden error problem. Inter-warp DMR when combined with the ReplayQ mechanism reduces the need for unnecessary stalls in the pipeline to execute redundant instructions, thus significantly lowering the performance overhead of temporal redundancy.

Simulation results on several GPGPU applications shows that intra-warp and inter-warp DMR complement each other to provide 96.43% error coverage with 16% worst case performance overhead.

The remainder of this paper is organized as follows. Section 2 provides background on the design of contemporary GPGPUs and error detection methods. Section 3 describes Warped-DMR. Section 4 describes the architectural modification for supporting Warped-DMR. Section 5 shows our evaluation methodology and results. Section 6 discuss related work and we conclude in Section 7.

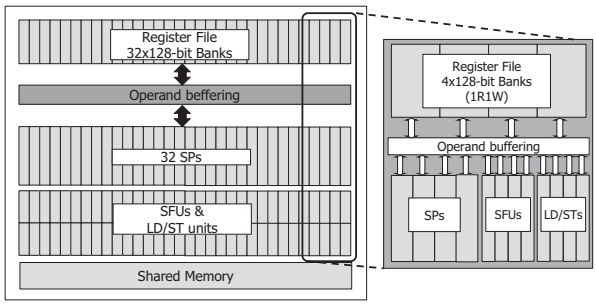


Figure 2: GPGPU Chip architecture and a SIMT cluster(modified after borrowing from [8])

2. Background

2.1. GPGPU Architecture

The GPGPU architecture varies depending on vendors and models. In this paper, we use the basic architecture of NVIDIA's Fermi [16] as our GPGPU model. A GPGPU consists of a scalable number of *streaming multiprocessor*(SM)s, each comprising of *shader processor*(SP) cores for arithmetic operations, *LD/ST units* for memory operations, *Special Function Units* (SFUs) for instructions such as sine, cosine, and square root, several register file banks, and shared memory. In this paper, we assume that each SM has 32 SPs, 32 register file banks and 64KB of shared memory.

Fig 2 shows the internals of one SM. The shared memory is accessible by all the SPs within a SM and part of this memory is configurable as an L1 cache. Each SM schedules threads in a unit of thread group of 32 threads, called a Warp. The threads within a warp execute the same code in a lock step manner. They all share one *program counter*(PC) but access different data operands. Such an execution approach is called *Single Instruction Multiple Threads*(SIMT) execution. Each thread in a warp may use a different register bank within a SM to access its data operands. Each individual thread execution is referred to as *SIMT lane*.

Within each SM, we further assume that four SIMT lanes make a *SIMT cluster* as in [8]. Thus each warp has eight SIMT clusters. As shown in Fig 2, each SIMT cluster has 4 SPs and 4 banks of register files. Each entry of a register bank is 128-bit wide and contains four 32-bit registers, each associated to one SIMT lane [8]. As each entry of the register bank consists of 4 registers having the same name but associated with 4 different threads, loading an entry from a register bank can feed all 4 SIMT lanes at once. Most common instructions that read 2 operands, write 1 result (*2R1W*), as well as the special instruction like MULADD that read 3 operands, write 1 result (*3R1W*) can access the four register banks to read their input operands and write output data concurrently without any register port

stalls most of the time. However, if an instruction fetches operands from the same bank, the operands cannot be fetched concurrently. To handle bank conflicts, GPGPUs use operand buffering logic that hides the latency of multi-cycle register fetch.

2.2. Underutilization of GPGPU Resources

Underutilization of GPGPU's computational resources can be due to two reasons (1) underutilization within homogeneous execution units and (2) underutilization among heterogeneous execution units.

Underutilization of homogeneous units: Underutilization within homogeneous execution units is caused by lock-step execution in GPGPUs. All 32 threads in a warp share a single PC. Whenever a branch instruction is encountered, some of the threads within the warp may take the branch while others may not depending on the data operands. Due to a single PC-constraint, threads with not-taken branch are executed first followed by threads with taken branches (or vice-versa). While the not-taken path instructions are executed, the core assigned to the taken path threads are idled. This is called the branch divergence problem.

To execute the divergent instructions, GPGPU hardware scheduler uses an *active mask* which consists of 32 bits indicating the active state of each thread within a warp. At each cycle, the threads whose active bit is set to '1' are allowed to execute the issued instruction, while the threads whose active bit is set to '0' wait. We call the thread as an *active thread* if it has '1' in the corresponding bit of the active mask and as an *inactive thread* otherwise. Note that there is one active mask per each warp.

A simple example of a branch divergence is illustrated in Figure 3. Let us assume that an if-then-else statement(shown in Figure 3(a)) is executed by a warp of two threads. When both threads reach the conditional branch instruction and if the condition is true for both threads then the two threads are concurrently executed (shown in Figure 3.(b)). However, if the two threads take different branch paths then only one thread can execute at a time (shown in Figure 3.(c)). In this example, the utilization of the system while executing the if-else statement becomes only 75% since among 8 cycles (2 cores $\times$ 4 cycles each), 6 cycles are actually used for the execution. Underutilization is even worse in real applications as shown in Figure 1: ranging from 7% in BFS to up to 77% in Bitonic Sort.

Underutilization of heterogeneous units: The underutilization among heterogeneous execution units is caused by the limitations in the scheduler feeding three different execution units. GPGPUs have three different types of execution units: SPs, LD/ST units, and SFUs. All three different types of execution units are fed by a single warp scheduler and an instruction dispatcher unit [16]. Hence, during any given cycle, only one instruction can be issued to one of the three execution units which leads to idle units. Heterogeneous unit underutilization has not been considered as severe as the underutilization caused by control divergence. However, if a code segment executes the same type instructions in a burst fashion then the scheduler will schedule instructions to just one type of execution unit while the rest two execution units remain idle.

Some state-of-the-art GPGPUs such as NVIDIA Fermi and Kepler [17] have multiple schedulers per SM. For example, Fermi has two schedulers in a SM which can issue instructions concurrently. The two schedulers share LD/ST units and SFUs while having their own SPs. Hence, instructions can be simultaneously issued to two different type execution units among three if the two schedulers issue different type operations. Even in this case there is still an underutilization of heterogeneous units since not all three execution units are used, but the degree of underutilization is decreased. Furthermore, due to several scheduling issues such as data dependency among the instructions, schedulers are not likely to be able to issue instructions to all the execution units.

3. Warped-DMR

Warped-DMR exploits the two types of underutilized resources to execute code redundantly and opportunistically. Different execution strategy is used for each of them.

3.1. Intra-warp DMR

To detect errors in execution units, intra-warp DMR relies on DMR execution approach. In traditional DMR there are as many *verification cores* as the number of *monitored cores*. A verification core executes the same instruction stream of the associated monitored core and the two execution outputs are compared. Error is detected if the results on the two cores differ. Every single instruction is thus executed twice providing 100% error coverage. However, the area or performance overhead of DMR exceeds 100% since at least one verification core should be added for each monitored core.

Intra-warp DMR uses the cores idled by underutilization within homogeneous execution units to execute the code redundantly, instead of adding extra cores for verification purpose. Whenever a partially utilized warp is scheduled, the operands of an active thread within the warp are forwarded to an inactive thread. The inactive thread thus can DMR an active thread's execution. The execution results of the inactive thread and the active thread are compared at the end of execution. If the two results are not identical, the hardware scheduler will be notified of an error occurrence. The necessary microarchitectural support for intra-warp DMR are discussed in Section 4.

Since the focus of this work is to detect errors, error handling is out of scope of this paper. But one can use simple techniques that allow the scheduler to either re-schedule the warp (in case of transient errors) or to stop running the program and raise an exception to the system (in case of a permanent fault).

3.2. Inter-warp DMR

Intra-warp DMR is an opportunistic approach that exploits idle cores. But when a warp utilizes all the cores, intra-warp DMR is unable to provide error detection coverage. To handle this case we present the second error detection method, called inter-warp DMR. Inter-warp DMR exploits resource underutilization caused due to heterogeneous execution units. As mentioned earlier, NVIDIA GPGPU uses SPs for arithmetic operations, LD/ST units for memory instructions, and SFUs for complex GPGPU operations such *sine* and *cosine*. In any given cycle the instruction issue logic issues instructions to only one of the three execution units. Hence, when an instruction is issued to SFUs or LD/ST units, the SPs may become idle. If different instruction types are issued in an interleaved manner, then each instruction's verification is done at the following cycle of the original execution. For instance, if an arithmetic instruction is followed by a LD/ST instruction, then the arithmetic instruction will be redundantly executed in the next cycle on SPs when the primary LD/ST instruction is being executed.

Figure 4 shows a simplified execution of a code segment which has several interleaved *add* and *load* instructions. As the two instructions use different type of execution units(*add* uses SPs and *load* executes on LD/ST units), whenever an instruction is issued onto the corresponding execution units, the other type of execution units become

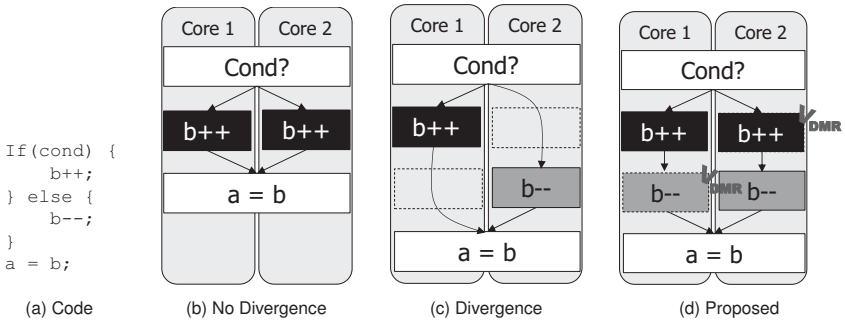


Figure 3: Example of underutilization of homogeneous units and Intra-Warp DMR

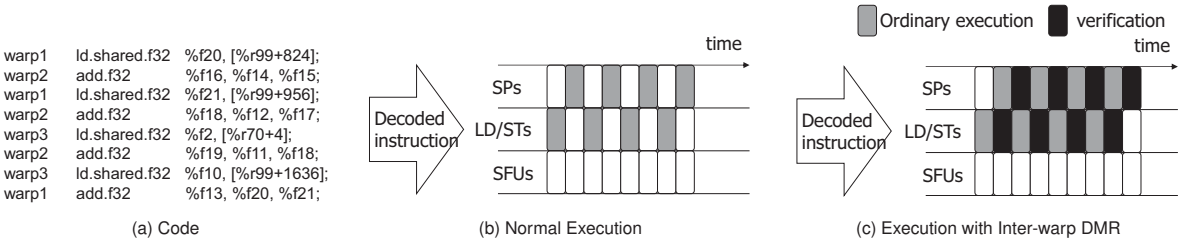


Figure 4: Example of underutilization of heterogeneous units and Inter-Warp DMR

idle. If both units take only one cycle to execute, inter-warp DMR allows the *add* and *load* instructions to be DMRed one cycle later than the original execution cycle on the associated execution units, without the need for stealing many cycles from regular program execution. Figure 4(c) depicts the operation of inter-warp DMR which only adds one extra cycle at the end of the eight cycle execution. Inter-warp DMR does not interfere with the execution scheduling of the primary *add* and *load* instructions.

Even with inter-warp DMR, there are scenarios when it is not possible to completely eliminate the overhead of DMR. In the example shown above, instructions that require different types of execution units are interleaved. But when the same type of instructions are scheduled for several cycles in a row, a new microarchitectural structure called ReplayQ is used to buffer the unverified instructions so that the instructions can be dequeued and re-executed whenever the corresponding execution unit becomes available. Note that we do not allow an instruction to consume unverified instruction results that are still buffered in the ReplayQ. Hence, whenever there is a RAW dependency on an unverified result, the dependent instruction is forced to wait and the ReplayQ gives priority to verify the source instruction.

During intra-warp DMR, the original code and verification code are guaranteed to be executed on different SIMT lanes. However, during inter-warp DMR, no such guarantee can be provided by default since contemporary GPGPUs may use core affinity that assigns a thread to the same core when redundantly executed. If an execution is DMRed on the same core, hardware defect on the core cannot be detected. For example, if core *i* has stuck-at-zero error, the result of the verification and original execution both will be 0, which leads to a hidden error. To avoid such hidden errors, inter-warp DMR associates a verification thread to a different SIMT lane than the original SIMT lane. We call this approach *Lane Shuffling*. Lane shuffling is operated within a SIMT cluster to minimize the wiring overhead. The microarchitectural enhancements for inter-warp DMR

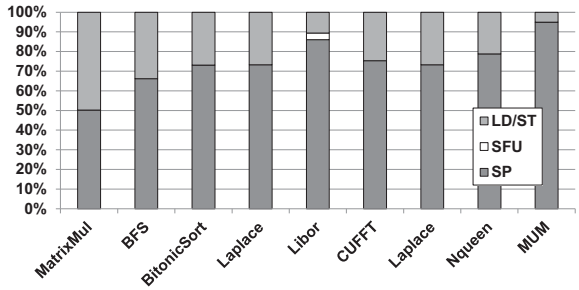


Figure 5: Execution time breakdown with respect to the instruction type

are discussed in Section 4.

3.3. Error Coverage

The theoretical error checking coverage of intra-warp DMR is 100% when the number of the active threads is less than half of the warp size. In this scenario every active thread's execution can be verified by at least one of the inactive threads. If the active thread count is greater than half of the warp size, the coverage is $\frac{\#inactive\_threads * 100}{\#active\_threads}\%$. The overhead of intra-warp DMR is almost zero as verification is done on the existing idle cores concurrently with the active threads. Only minimal hardware logic is added for register forwarding and results comparison.

The theoretical error checking coverage of inter-warp DMR is 100% as each fully occupied warp's execution is re-executed a few cycles later. The best case execution overhead of inter-warp DMR is zero as the redundant execution is done only when the corresponding execution unit is idle. In reality, due to the capacity of the ReplayQ and the unbalanced instruction distribution(see Figure 5 for instruction type distribution), there will be some overhead. Our results show that the worst case overhead is 16%, which is well below the theoretical overhead of 100%.

Priority	MUX0	MUX1	MUX2	MUX3
1st	0	1	2	3
2nd	1	0	3	2
3rd	2	3	0	1
4th	3	2	1	0

Table 1: Priority table of RFU MUXs

3.4. Advantages of Warped-DMR

Warped-DMR verifies the computations at individual execution unit level(i.e. SP). The DMR can be done at a coarser granularity, such as at the entire SM level by duplicating a thread block[1] onto two different SMs or at the chip level by invoking two copies of a kernel function onto two GPGPUs. The coarser method might be simpler to implement. However, the finer method allows for more aggressive error detection. For example, when there is a faulty SP, a SM-level or a chip-level error checking cannot isolate which core has the defect. Hence, the only option to fix the problem is to disable the entire SM even though the remaining 31 SPs in the SM as well as the other logic blocks including scheduler, dispatcher, and the local memory are fault-free. Similarly, when using chip level checking, one has to disable an entire GPGPU chip even with just one failed SP. With Warped-DMR we can monitor the reliability at the granularity of a SP. In the previous examples, we can still use the SM even though a SP has a defect by using a core re-routing approach as suggested in [23].

It is also worth noting that the static power consumption of GPGPUs is nearly 60% of the total power consumption. To reduce static power consumption [9] showed that it is best to provide power gating at the SM level. Idle SM periods can be long and hence they can be completely turned off . But providing power gating at the SP level does not provide enough benefits. While SPs are idle for a significant fraction of the time, the idleness is finely interspersed with periods of activity. The latency of power gating outweighs the benefits of turning off idle SPs. Warped-DMR is thus an ideal choice for repurposing idle SPs to provide reliability since power gating idle SPs is not beneficial.

4. Architectural support for Warped-DMR

4.1. Register Forwarding Unit

For intra-warp DMR each inactive thread that is going to verify the computation should be able to either access an active thread's register file or get the active thread's register values using data forwarding. As adding an extra port to the register file is expensive, we added a *Register Forwarding Unit*(RFU) at the end of each register bank. RFU consists of four 4 32-bit input MUXs as can be seen in Figure 6. A 128-bit entry of a register bank is divided into 4 32-bit data and forwarded to all the 4 MUXs.

To enable intra-warp DMR the four MUXs in the RFU pair active threads with inactive threads based on a priority. The priority configuration of the 4 MUXs within a SIMT cluster is shown in Table 1. Each column indicates the priority ordering for each MUX. As a first priority every MUX delivers the input data to its associated SIMT lane if that SIMT lane's active mask is set. MUX0 provides input data to SIMT lane 0, MUX1 to SIMT lane 1 and so on. If a SIMT lane's active mask is reset, then that SIMT lane is idle and can be used for DMR. Hence, every idle SIMT lane looks for an active SIMT lane whose computation can be redundantly executed on the idle SIMT lane. To find an active SIMT lane which can be redundantly executed on an idle SIMT lane, each MUX looks for the active SIMT lane according to the priority listed in the table. For instance, if SIMT lane 0 is idle, then MUX0 looks at SIMT lane 1 to see if it is active as lane 1 is the 2nd priority for MUX0. If so, then the inputs from SIMT lane 1 are then simply directed by MUX0 to run on SIMT lane 0. If SIMT lane 1 is also inactive then SIMT lane 2 active mask bit is checked followed by SIMT lane 3 active mask to find an active thread. As can be seen from Table 1, each MUX runs through a different priority sequence to allow uniform pairing possibilities between active and idle SIMT lanes. In this algorithm, if there is only one active lane, the lane is redundantly executed on the rest three idle lanes, which results in more than dual modular redundancy. We simply allow such a scenario to occur rather than to add additional hardware logic in the MUX to prevent this scenario since it does not lower the error coverage.

In Figure 6, the bold lines inside of RFU illustrates a simple example of an intra-warp DMR when an active mask for an instruction is 4'b0011. As each bit of an active mask indicates each corresponding thread's activeness, 4'b0011 means that thread 0 and 1 are active and thread 2 and 3 are inactive for the instruction. Thread 2 and 3 will perform DMR for the execution of thread 0 and 1 according to intra-warp DMR assignment from Table 1.

We implemented a RFU design and a 128-bit comparator by using *Synopsis Design Compiler v.Y-2006.06-SP4* [21]. The respective area overhead is $390\mu m^2$ and $622\mu m^2$ and the timing overhead is 0.08ns and 0.068ns. The timing overhead of the MUX is thus less than 0.06% compared to a typical cycle period(1.25ns) of GPGPU of 40nm technology and 800MHz core clock. [2]

4.2. Thread-Core Mapping

Since the register forwarding is limited to within a SIMT cluster of just four SPs, in intra-warp DMR the verification and monitored core are restricted to be within the same SIMT cluster. This limitation minimizes wire delays and complex routing paths. With this mapping restriction, however, some SIMT clusters might not be able to use intra-warp DMR if all the SIMT lanes within a cluster are fully utilized, even when some SIMT lanes across clusters are idle. Based on preliminary experiments we found that many applications are likely to have unbalanced active thread distribution within a warp.

To improve the availability of idle SPs within a SIMT cluster, we modified the thread to core affinity scheduling algorithm. There is only sparse documentation on how current GPGPUs assign threads to SPs within a warp. It is believed that the threads are mapped to cores in order: for example, thread 0 is always executed on core 0, thread 1 is mapped to core 1 and so on. Our modified scheduling algorithm assigns threads to SIMT clusters in a round-robin fashion. Thus thread 0 is assigned to cluster 0, thread 1 is assigned cluster 1 and so on. We show later in our results section that this simple scheduler change increased error detection opportunities by 9.6% compared to the default in order mapping of threads to core.

4.3. ReplayQ

As discussed earlier, inter-warp DMR relies on re-executing an instruction at a later time if the corresponding execution units are not free. Instead of stalling program execution, inter-warp DMR buffers

[1] A thread block in NVIDIA CUDA programming is a logical partition of a program. Any thread can communicate with other threads only when they are in the same thread block. A thread block is launched onto a SM.

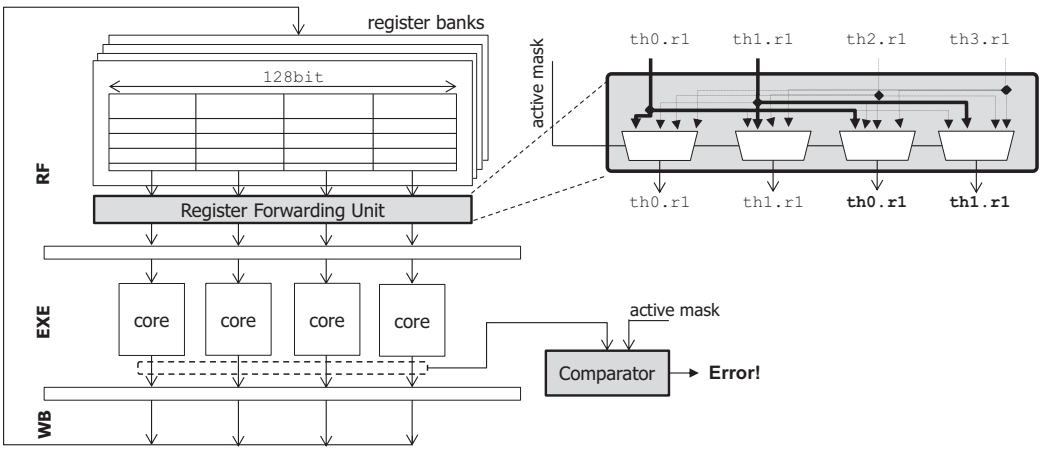


Figure 6: Register Forwarding Unit and Comparator for Intra-Warp DMR

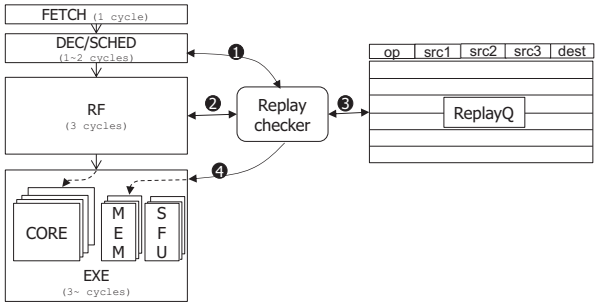


Figure 7: ReplayQ and Replay Checker for Inter-Warp DMR

unverified instructions into a ReplayQ whenever the corresponding execution unit is not available. A *Replay Checker* engine is designed to manage the ReplayQ. There is one Replay Checker and ReplayQ per SM.

Figure 7 shows how the Replay Checker works within the context of current GPGPU pipeline. A GPGPU pipeline in the figure has the following stages: instruction fetch stage (FETCH), decode & schedule stage (DEC/SCHED), register fetch stage (RF), and execution stage (EXE). The write back stage (WB) is omitted in the figure for simplicity. The latency of each pipeline stage is also shown in the figure. The stages having multiple cycle latency consist of multiple sub-stages. For example, RF is comprising of RF0, RF1, and RF2. These latencies reflect the pipeline latencies of current GPGPUs that we modeled [8].

If active mask of the instruction in the first *RF* stage is all active, the Replay Checker is activated. If the active mask has some idle slots intra-warp DMR will verify the warp's execution. During intra-warp DMR execution Replay Checker and ReplayQ do not play any role in managing the warp's redundant execution. Once Replay Checker is active it compares the instruction type of the warp in *RF*(❷) and that of the warp in $DEC/SCHED$ stage(❶). If the instruction type is different, then the Replay Checker creates a DMR copy of the *RF* instruction to be co-executed with the instruction in $DEC/SCHED$ (❹). DMR copy consists of the values of the input operands and opcode. Note that even though an instruction takes several cycles of the execution, the next instruction can be issued at the following cycle to the execution unit as the EXE stage itself is super-pipelined. If *RF* and $DEC/SCHED$ instruction type are the same then the instruction type in *RF* stage (a two bit value indicating SP, LD/ST or SFU instruction) is compared against all the queued entries in the ReplayQ (❸). The instruction decoder would have already marked each instruction based on its execution resource demand into either SP, LD/ST or SFU instruction type. In our experiments the maximum size of ReplayQ is 10 entries. Hence, 10 two bit XORs are used for this comparison. If any instruction in the ReplayQ has different type than the instruction in *RF* then the Replay Checker dequeues that instruction and pairs it with the instruction in *RF* stage (❹) for co-execution in the next cycle. When multiple ReplayQ instructions are available for co-execution then one instruction is picked at random. The instruction in *RF* stage is then enqueued in the ReplayQ. When an instruction *RF* is enqueued into ReplayQ it simply implies that the instruction that is one cycle behind *RF* is going to use the same type of execution units as the instruction in *RF*. Hence, there will be no opportunity to verify the *RF* instruction in the next cycle following its execution cycle. Hence, that instruction needs to be buffered for future verification.

Also to distinguish a DMR execution from an original execution, a single *dmr* bit is added. *dmr* is set by Replay Checker when creating a DMR copy. By using this value, RFU can apply the lane shuffling only to the fully utilized DMR executions.

If there is no instruction in the ReplayQ whose instruction type is different than the instruction in the *RF* stage, the Replay Checker checks if the ReplayQ is full. If the ReplayQ has empty slots, the *RF* instruction is enqueued to the ReplayQ (❸). If the ReplayQ is full, a stall cycle is inserted into the pipeline immediately after the instruction in *RF* finishes the first *EXE* stage and then the instruction is re-executed by using the operand values that are still available in the pipeline. This eager re-execution reduces unnecessary register reads but adds one cycle performance penalty. Note that this penalty is applicable only in the rare case that ReplayQ has no instruction that is different than *RF* stage instruction *and* ReplayQ is full.

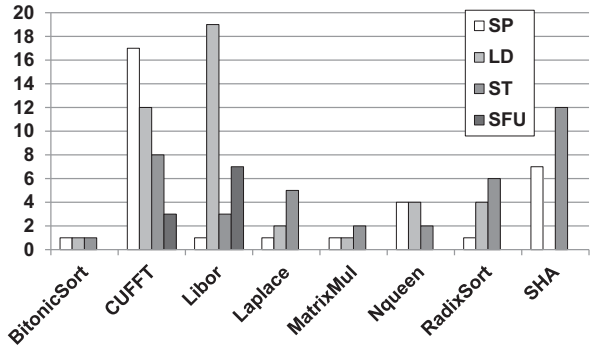
Whenever a new instruction is scheduled in the pipeline which is going to consume (RAW dependency) data from an unverified instruction that is buffered in the ReplayQ then the Replay Checker stalls the pipeline and executes the verification of the source instruction before allowing the consumer instruction to execute.

Algorithm 1 shows the pseudo code of the Inter-Warp DMR with ReplayQ.

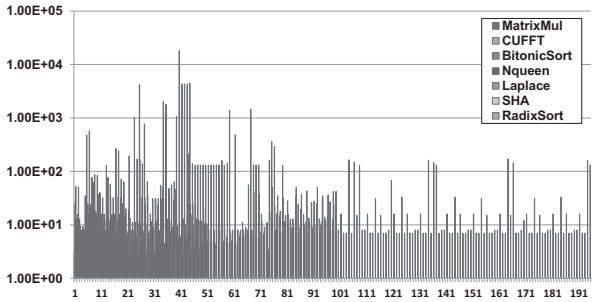
4.3.1. Components and Effective Size of ReplayQ Since ReplayQ buffers instructions only when there is no available resource, it is

Algorithm 1 Inter-Warp DMR with ReplayQ

i_{rf} := *instruction in RF stage*
i_{dec} := *instruction in DEC/SCHED stage*
op_{rf} := *instruction type of* i_{rf}
op_{dec} := *instruction type of* i_{dec}
if $op_{rf} \neq op_{dec}$ **then**
  Coexecute DMR of i_{rf} *with* i_{dec} *execution*
else
  if $\exists i_{rq} : i_{rq} \in ReplayQ$ *and instruction type of* $i_{rq} \neq op_{rf}$ **then**
    Dequeue i_{rq} *from ReplayQ*
    Enqueue i_{rf} *to ReplayQ*
    Coexecute DMR of i_{rq} *with* i_{rf} *execution*
  else
    if *ReplayQ is full* **then**
      Insert a Stall cycle
      DMR i_{rf} *one cycle later the original execution*
    else
      Enqueue i_{rf} *to ReplayQ*
    end if
  end if
end if



(a) Instruction type switching distances within 1000 cycles



(b) RAW dependency distances of the registers of warp1 thread 32(warp0 thread1 for SHA)

Figure 8: Two key factors to determine effective ReplayQ size

OS	Ubuntu Linux kernel v2.6.38
CPU	Intel Core i7(quad core) @ 2.67 GHz
Compiler	nvcc-2.3 / gcc-4.3.4

Table 2: Experimental Environment

Parameter	Value
Execution Model	In-order
Execution Width	32 wide SIMT
Warp Size	32
# Threads/Core	1024
Register Size	64 KB
# Register Banks	32
# Core(SP)s/Multiprocessor(SM)	32
# SMs	30

Table 3: Simulation Parameters

critical to quantify how often such a scenario occurs in GPGPUs. ReplayQ also stalls the pipeline whenever there is RAW dependency on an unverified instruction.

Figure 8(a) shows the average cycle distance before an instruction type is switched to another. In most of the applications, normally less than 6 instructions of the same type are consecutively issued. CUFFT, Libor, and SHA have longer distances between different instruction types but it is also bounded to a maximum of 20. Hence, the ReplayQ only needs to buffer 20 instructions in the worst case, but an average size of 6 will suffice for most applications.

Figure 8(b) shows the number of cycles between when a register is written to the time when that register is read by another instruction. In this figure we only show the RAW dependency distance for warp1 thread 32. But the data is quite similar for all warps and all threads within each warp. The RAW dependency distance is at least 8 cycles and almost half of the registers have greater than 100 cycles of distance and some have even longer than 1000 cycles of distance. Hence, the RAW dependency related pipeline stalls are likely to be just a few in Warped-DMR.

Each entry of the ReplayQ should maintain opcode and the original execution result as well as the source register values. The original execution result is for verifying the original execution. Each SM has one ReplayQ which covers all the SIMT Clusters. Each entry of a ReplayQ contains 32 *lanes* $\times$ 3 *operands*(each instruction can have up to 3 operands)$\times$4 *bytes* for the source register values, 32 *lanes* $\times$ 4 *bytes* for the original execution result and $2 \sim 4$ *bytes* for the opcode so total of $514 \sim 516$ *bytes*. Therefore, the ReplayQ size with 10 entries is around 5KB. This is only 4% of the register file size which is assumed to have 128KB in [8].

5. Evaluation

5.1. Settings and Workloads

We used *GPGPU-Sim v3.0.2* [3] to evaluate the proposed Warped-DMR approach. The simulation environment is described in Table 2 and the simulation parameters are set as listed in Table 3. The simulation parameters model our baseline GPGPU architecture as illustrated in Fig 2. A GPGPU chip has 30 SMs and each SM is comprising of 32 SIMT lanes. 4 SIMT lanes build a SIMT cluster which consists of 4 register banks, 4 SPs, 4 LD/ST units and 4 SFUs.

For the workloads, we used several applications from NVIDIA CUDA SDK [4], Parboil Benchmark Suite [5], and ERCBench [1].

Category	Benchmark	Parameter
Scientific	Laplace transform	$gridDim = 25 \times 4$, $blockDim = 32 \times 4$
	Mummer	$input files$: $NC\_003997.20k.fna$ and $NC\_003997\_q25bp.50k.fna$
	FFT	$gridDim = 32$, $blockDim = 25$
Linear Algebra/Primitives	BFS	$input file$: $graph65536.txt$, $gridDim = 256$, $blockDim = 256$
	Matrix Multiply	$gridDim = 8 \times 5$, $blockDim = 16 \times 16$
	Scan Array	$gridDim = 10000$, $blockDim = 256$
Financial	Libor	$gridDim = 64$, $blockDim = 64$
Compression/Encryption	SHA	$directmode$, $inputsize$: 99614720, $gridDim = 1539$, $blockDim = 64$
Sorting	Radix Sort	$–n = 4194304$ $–iterations = 1$ $–keysonly$
	Bitonic Sort	$gridDim = 1$, $blockDim = 512$
AI/Simulation	NQueen	$gridDim = 256$, $blockDim = 96$

Table 4: Workloads

As mentioned earlier, our main target applications are those needing strict accuracy such as scientific computing or financial applications. Hence, we excluded some applications that are inherently fault tolerant, such as graphics applications. We picked 6 categories of applications: scientific computing, linear algebra/primitives, financial, compression/encryption, sorting, and AI/simulation. The applications that are included in the 6 categories are listed in Table 4.

5.2. Error Coverage and Overhead

Figure 9(a) shows the percentage of executed instructions covered by Warped-DMR. We compare three different implementations. The baseline implementation is the 4 SIMT lane cluster with no enhanced thread-core mapping. The second bar shows the impact of increasing the cluster size to 8 SIMT lanes and allowing register forwarding within a larger cluster size. The last bar shows the results using the enhanced thread-core mapping as stated in the Section 4.2. Warped-DMR with enhanced thread mapping provide an average of 96.43% error coverage compared to 91.91% error coverage with a more hardware intensive 8 SIMT lane cluster. The gaps in error coverage are primarily due to intra-warp DMR when the number of idle cores is fewer than the number of active cores. For instance, BFS is almost exclusively covered by only intra-warp DMR as all the warps are underutilized. The utilization of all the warps in BFS is less than 50% as illustrated in Figure 1. Hence, every single active thread's execution can be verified by inactive threads without any ReplayQ involvement. Such applications also have negligible performance overhead (almost zero)as seen in Figure 9(b) while the error coverage is 100%. CUFFT derived the lowest error coverage, 90.167%, as most of the underutilized warps' utilization is greater than 80%, implying only 25% of active threads' executions can be verified.

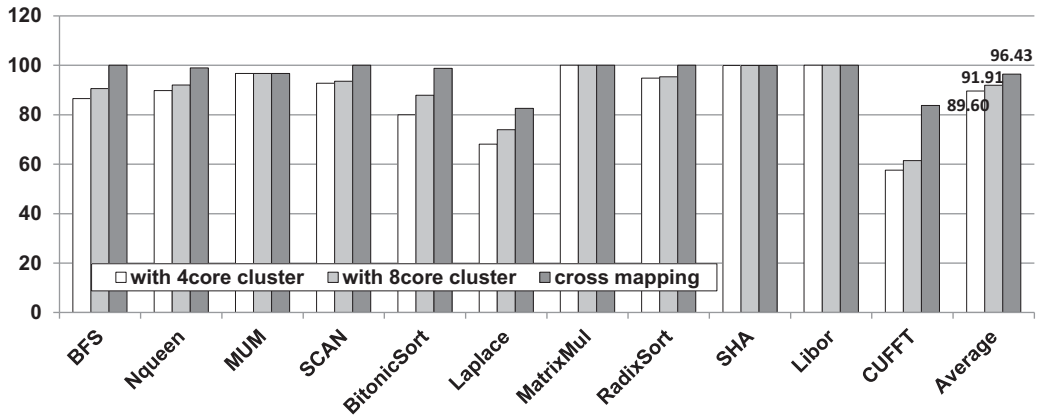
Applications that are well parallelized like Libor, MatrixMul, and SHA are mostly covered by inter-warp DMR as most of the warps are fully utilized. In such applications, the error coverage is almost 100% but the performance overhead is higher than the other applications as shown in Figure 9(b). There are four bars per benchmark in Figure 9(b). Each bar is normalized to the kernel execution cycles of the base machine with zero error detection support. Using the data presented in Section 4.3.1, we varied the ReplayQ from 0 to 10 entries. As the ReplayQ size increased to a maximum of just 10 entries the average performance overhead reduced to 16%. In some applications that are mostly covered by inter-warp DMR such as MatrixMul, performance overhead without ReplayQ exceeds 70%. However, by using 10 entries of ReplayQ, the overhead drops to 18%.

5.3. Comparison with SW approach

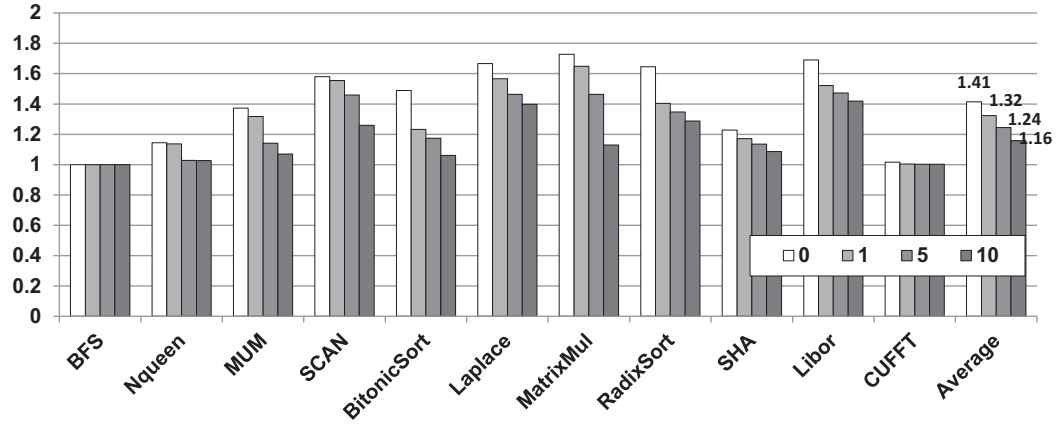
In this section, we compare Warped-DMR with other approaches. We modified applications to provide error detection capabilities through R-Naive and R-Thread approaches as described in [6]. R-Naive simply invokes the kernel function twice and then compares the output data. As each kernel should use the same input data, the memory copy between CPU and GPGPU also should be duplicated. It is relatively simple to implement but has more than 100% time overhead [6]. R-Thread duplicates the thread blocks within a kernel and then compares the output of the original and the duplicated thread block. The duplicated thread block refers to the original thread block's input data by modifying index calculations. If there is any SM that is not used for the kernel execution, the redundant thread block can be executed on it simultaneously with the original thread blocks. Otherwise, R-Thread also can have quite high execution overhead as the execution time for redundant thread blocks cannot be hidden. R-Thread does not call expensive CUDA APIs redundantly like R-Naive but it still requires twice as much data transfer from GPGPU to CPU resulting in significant data transfer overhead. R-Scatter in [6] is not used in this experiment as it is for VLIW architecture and it was shown to be inferior to R-Thread. We also implemented a dual modular temporal redundancy(DMTR) which verifies every single instruction in the following cycle. It is a simplified version of SRT with 1 cycle of slack time [19].

Figure 10 shows the execution time of the four different error detection approaches and the original execution without error detection. The execution time includes data transfer time between CPU and GPGPU as well as kernel execution time. As the data transfer between CPU and GPGPU is not included in the simulation, the data transfer time was measured by using CUDA Timer API. As R-Naive should call kernel twice, the data transfer time also becomes twice the original. In R-Thread, twice the size of original output should be copied back to the CPU as the output of redundant and original thread block are compared on the CPU side. On the other hand, both DMTR and Warped-DMR have the same data transfer time as the original execution as they compare the execution results on the GPGPU.

Of the four approaches, R-Naive took the longest time for execution. It is mainly due to two individual kernel executions and the twice the data transfer time. R-Thread can reduce the execution overhead in the presence of idle resources at granularity of a SM. For instance, Bitonic Sort uses fewer than 30 SMs and hence idle SMs are always available. However, in the other applications which use all the SMs for the original execution, R-Thread has no room to hide the execution time of the redundant thread blocks. Also, R-Thread trans-



(a) Error coverage with respect to the SIMT cluster organization and Thread to Core mapping



(b) Normalized Kernel simulation cycles with respect to the ReplayQ size

Figure 9: Error coverage and Overhead of Warped-DMR

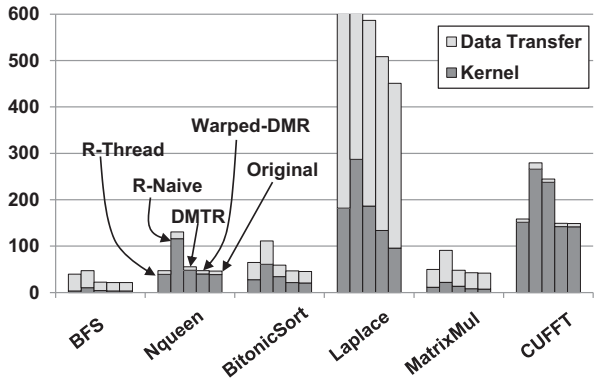


Figure 10: Execution times of different approaches

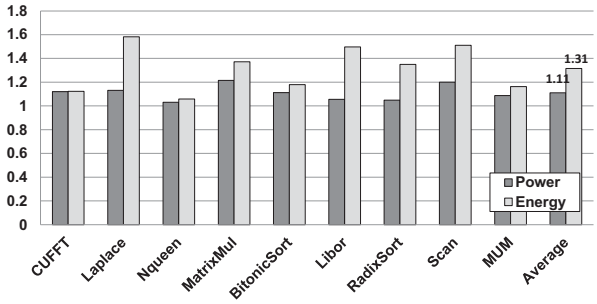


Figure 11: Normalized Power Consumption

fers twice the output data to the CPU. Therefore, R-Thread suffered the second longest execution time in many of the tested applications. Warped-DMR derived the best performance among the four different approaches due to opportunistic error detection. In some applications like Laplace, Warped-DMR has 20% longer kernel execution time compared to the original execution. However, when the data transfer time is included, the overall overhead is reduced to 13%.

5.4. Power Consumption

We measured power and energy consumption by using an analytical model [9]. In [9], the power consumption of processing components are estimated based on the following equation:

$$RP_{comp} = MaxPower_{comp} \times AccessRate_{comp} \quad (1)$$

$$AccessRate_{comp} = \frac{\sum \#Insts\_accessing\_comp\ per\ SM}{Exec\_cycles \div Ints\_sched\_interval} \quad (2)$$

The total power consumption consists of runtime and idle power. The runtime power is again comprising of runtime power of SMs and Memory. Each SM's runtime power is aggregated power consump-

tion of several processing components(SPs, SFUs, caches, shared memory, register file, and fetch/decode/schedule unit) and a constant factor. Each component's runtime power(RP_{comp}) can be calculated by multiplying the access rate of the component($AccessRate_{comp}$) by a power parameter($MaxPower_{comp}$) as shown in Equation 1. $AccessRate_{comp}$ is estimated by dividing the total number of instructions accessing the component(#$Insts\_accessing\_comp$) by the execution cycles($Exec\_cycles$) over the instruction scheduling interval($Ints\_sched\_interval$).

We assume that the architecture dependent parameters(i.e. $MaxPower_{comp}$, idle power, and constant factor) are the same as in [9]. By using the total simulation cycles and the number of executed instructions acquired from our simulation, we estimated the power consumption of each application with and without Warped-DMR. Memory components such as caches and shared memory are not included in the estimation since Warped-DMR only doubles the address calculations for the memory operations and the redundant executions are always conducted on already loaded data. ReplayQ is assumed to have 10 entries.

As shown in Figure 11, the power consumption of Warped-DMR is 11% higher and the overall energy consumption is 31% higher than the baseline with zero error detection capability. Some applications such as Laplace consume the worst case 60% more energy due to timing overhead as shown in Figure 9. Energy consumption is calculated by multiplying the power consumption of each application by its execution time. The execution time is calculated by multiplying the number of simulated cycles by the cycle duration. We assume that the cycle duration is 1.25 ns.

6. Related Work

In this section we describe the most relevant prior work on GPGPU reliability and DMR execution. [6] proposed three software approaches: R-Naive, R-Scatter, and R-Thread. R-Naive simply invokes memory API and kernel function twice to create a software-centric DMR execution within a GPGPU. R-Scatter tries to exploit underutilized VLIW lanes for redundant execution by duplicating kernel code. R-Thread doubles the thread block count for a kernel and uses the newly added thread blocks do the redundant execution. We compared our work with the R-Naive and R-thread approaches and showed that Warped-DMR significantly reduces the overhead of DMR execution.

[14] checks computations, control flow, and data flow of a GPGPU program by inserting signature collector code. After the kernel execution, the collected signatures are compared with statically generated signatures. However, this approach only checks the computation after the kernel code is complete, which can be much later than when the error was first encountered.

[22] is a systematic approach which uses a guardian process that intercepts the crash event and restarts the program using checkpoints. They also instrumented the source code, duplicated non-loop code and inserted range checking code for loops. This approach also relies on extensive software instrumentation and large checkpoints to support redundant execution. Many of these reliability studies for GPGPUs are software approaches. Software approaches are always more flexible compared to hardware approaches. However, the error coverage can be limited by the granularity of programmers (or compiler's) code insertion. Compared to those approaches, Warped-DMR can check 96.43% of all instructions without any programmer's effort.

[15] suggested a sampling DMR in which DMR is conducted only for a short period of time within each epoch rather than doing it for entire execution time. Using this approach the authors state that permanent errors can be eventually detected even though transient errors might be missed. Warped-DMR takes advantage of GPU-specific microarchitectural features to provide high coverage for both transient and permanent faults.

[19] proposed a Simultaneous and Redundantly Threaded (SRT) processor design. Instead of replicating hardware resources, they used thread level replication. Trailing thread redundantly executes the same program copy that the leading thread executes. The hardware resources are shared between the trailing and the leading threads. [13] proposed a Chip-level Redundantly Threaded (CRT) processor which explicitly disables core-affinity to make sure that two threads execute on different cores. This approach essentially exploits the performance advantage of SRT's loose synchronization as well as the high fault coverage of lockstepping method [20]. [10] proposed a method to reduce the performance overhead of SRT by preventing the trailing thread from redundantly fetching register data. The key idea is to reuse the already fetched data for the trailing thread.

Compared to the hardware based DMR or RMT approaches, Warped-DMR has some domain-specific advantages. Warped-DMR checks every single instruction (but in less than 4% of cases it checks only partial number of inputs). This approach not only detects permanent errors but most transient errors can also be detected. Also, unlike [20], Warped-DMR does not use an entire core just for DMR. Instead, we utilize the idle periods of cores for DMR.

Due to DMR's high area overhead, some self-checking schemes also have been studied. One of the most popular self-checking schemes is residue checking [18] [12]. Instead of duplicating entire execution units, residue checking adds residue operator units which require much less area than the entire execution units. An error in the original operator unit is detected by comparing the residue of the original computation result and the output of the residue operation which is executed on the residue operator unit. Residue checking has small area footprint but residue checking is only applicable for some simple arithmetic operations(it cannot be used for exponent calculations [12]). Warped-DMR can detect errors in any arithmetic operation supported on a GPGPU, including complex operations implemented in an SFU.

7. Conclusion

As GPGPUs play critical role in high performance computing today, reliability should be treated as a first class citizen alongside performance. In this paper, we proposed Warped-DMR a hardware approach to detect computation errors in GPGPUs. We presented the reasons for underutilization of resources in GPGPU applications and then presented inter-warp and intra-warp DMR to exploit the idle resources for error detection. Intra-warp DMR checks the active threads' execution by using idle cores from underutilized warps. For the fully utilized warps, inter-warp DMR verifies computation by using temporal DMR whenever the corresponding execution unit becomes idle. A simple ReplayQ microarchitecture design is used for maintaining instructions in case the corresponding execution unit is not idle for several cycles. To prevent an instruction from being executed and verified on the same core, which may lead to hidden errors, we designed a register forwarding/lane shuffling logic. We presented a detailed state space exploration and showed that Warped-DMR provides 96.43% error coverage with 16% performance overhead.

Acknowledgement

We would like to thank the anonymous reviewers for their valuable comments. This work was supported by DARPA-PERFECT-HR0011-

12-2-0020 and NSF grants NSF-1219186, NSF-CAREER-0954211, NSF-0834798.

References

[1] "Ercbench." [Online]. Available: http://ercbench.ece.wisc.edu/

[2] "Geforce 400 series." [Online]. Available: http://en.wikipedia.org/wiki/GeForce_400_Series

[3] "Gpgpu-sim." [Online]. Available: http://www.ece.ubc.ca/ aamodt/gpgpu-sim/

[4] "Nvidia cuda sdk 2.3." [Online]. Available: http://developer.nvidia.com/cuda-toolkit-23-downloads

[5] "Parboil benchmark suite." [Online]. Available: http://impact.crhc.illinois.edu/parboil.php

[6] M. Dimitrov, M. Mantor, and H. Zhou, "Understanding software approaches for gpgpu reliability," in *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, March 2009, pp. 94–104.

[7] X. Fu, N. Goswami, and T. Li, "Analyzing soft-error vulnerability on gpgpu microarchitecture," in *Proceedings of the 2011 IEEE International Symposium on Workload Characterization*, November 2011, pp. 226–235.

[8] M. Gebhart, D. R. Johnson, D. Tarjan, S. W. Keckler, W. J. Dally, E. Lindholm, and K. Skadron, "Energy-efficient mechanisms for managing thread context in throughput processors," in *Proceedings of the 38th annual International Symposium on Computer Architecture*, Jun 2011, pp. 235–246.

[9] S. Hong and H. Kim, "An integrated gpu power and performance model," in *Proceedings of the 37th annual International Symposium on Computer Architecture*, Jun 2010, pp. 280–289.

[10] S. Kumar and A. Aggarwal, "Reducing resource redundancy for concurrent error detection techniques in high performance microprocessors," in *Proceedings of the 12th International Symposium on High Performance Computer Architecture*, February 2006, pp. 212–221.

[11] V. W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, R. Singhal, and P. Dubey, "Debunking the 100x gpu vs. cpu myth: an evaluation of throughput computing on cpu and gpu," in *Proceedings of the 37th annual International Symposium on Computer Architecture*, Jun 2010, pp. 451–460.

[12] D. Lipetz and E. Schewarz, "Self checking in current floating-point units," in *Proceedings of the IEEE 20th Symposium on Computer Arithmetic*, July 2011, pp. 73–76.

[13] S. S. Mukherjee, M. Kontz, and S. K. Reinhardt, "Detailed design and evaluation of redundant multithreading alternatives," in *Proceedings of the 29th annual International Symposium on Computer Architecture*, May 2002, pp. 99–110.

[14] R. Nathan and D. J. Sorin, "Argus-g: A low-cost error detection scheme for gpgpus," in *Workshop on Resilient Architectures*, December 2010.

[15] S. Nomura, M. D. Sinclair, C.-H. Ho, V. Govindaraju, and M. de Kruijf, "Sampling + dmr: Practical and low-overhead permanent fault detection," in *Proceedings of the 38th annual International Symposium on Computer Architecture*, Jun 2011, pp. 201–212.

[16] NVIDIA, "Fermi white paper v1.1." Available: http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf

[17] NVIDIA, "Nvidia geforce gtx 680 white paper v1.0." Available: http://www.geforce.com/Active/en_US/en_US/pdf/GeForce-GTX-680-Whitepaper-FINAL.pdf

[18] N. Ohkubo, T. Kawashimo, M. Suzuki, Y. Suzuki, J. Kikuchi, M. Tokoro, R. Yamagata, E. Kamada, T. Yamashita, T. Shimizu, T. Hashimoto, and T. Isobe, "A fault-detecting 400 mhz floating-point unit for a massively-parallel computer," in *International Solid-State Circuits Conference*, February 1999, pp. 368–369.

[19] S. K. Reinhardt and S. S. Mukherjee, "Transient fault detection via simultaneous multithreading," in *Proceedings of the 27th annual International Symposium on Computer Architecture*, Jun 2000, pp. 25–36.

[20] T. J. Slegel, R. M. A. III, M. A. Check, B. C. Giamei, B. W. Krumm, C. A. Krygowski, W. H. Li, J. S. Liptay, and J. D. MacDougall, "Ibm's s/390 g5 microprocessor design," in *IEEE MICRO*, March 1999, pp. 12–23.

[21] Synopsis, "Design compiler user guide," 2010. Available: http://acms.ucsd.edu/info/documents/dc/dcug.pdf

[22] K. S. Yim, C. Pham, M. Saleheen, Z. Kalbarczyk, and R. K. Iyer, "Hauberk: Lightweight silent data corruption error detector for gpgpu," in *Proceedings of 25th IEEE International Parallel & Distributed Processing Symposium*, May 2011, pp. 287–300.

[23] L. Zhang, Y. Han, Q. Xuz, and X. Li, "Defect tolerance in homogeneous manycore processors using core-level redundancy with unified topology," in *Proceedings of the Conference on Design, Automation and Test in Europe*, March 2008, pp. 891–896.

2012 IEEE/ACM 45th Annual International Symposium on Microarchitecture

The performance vulnerability of architectural and non-architectural arrays to permanent faults

Damien Hardy*†, Isidoros Sideris*, Nikolas Ladas*, Yiannakis Sazeides*
*University of Cyprus †University of Rennes 1, IRISA, France

Abstract

This paper presents a first-order analytical model for determining the performance degradation caused by permanently faulty cells in architectural and non-architectural arrays. We refer to this degradation as the performance vulnerability factor (PVF).

The study assumes a future where cache blocks with faulty cells are disabled resulting in less cache capacity and extra misses while faulty predictor cells are still used but cause additional mispredictions.

For a given program run, random probability of permanent cell failure, and processor configuration, the model can rapidly provide the expected PVF as well as lower and upper PVF probability distribution bounds for an individual array or array combination.

The model is used to predict the PVF for the three predictors and the last level cache, used in this study, for a wide range of cell failure rates. The analysis reveals that for cell failure rate of up to 1.5e-6 the expected PVF is very small. For higher failure rates the expected PVF grows noticeably mostly due to the extra misses in the last level cache. The expected PVF of the predictors remains small even at high failure rates but the PVF distribution reveals cases of significant performance degradation with a non-negligible probability.

These results suggest that designers of future processors can leverage trade-offs between PVF and reliability to sustain area, performance and energy scaling. The paper demonstrates this approach by exploring the implications of different cell size on yield and PVF.

1. Introduction

For the past 50 years, technological advances have enabled continuous miniaturization of circuits and wires. The increasing device density offers designers the opportunity to place more functionality per unit area and in recent years has allowed the integration of large caches and many cores into the same chip. Unfortunately, the scaling of device area has been accompanied by at least two negative consequences: a slowdown of voltage scaling and frequency, due to slower scaling of leakage current as compared to area scaling [34], and a shift to probabilistic design and less reliable silicon primitives due to variations [6, 9].

Technology trends suggest that in tomorrow's computing world failures will become commonplace due to many and/or frequent causes such as: static [6] and dynamic [9] variations, latent-faults due to limited burn-in [35], higher temperatures and accelerated wear-out [33], near subthreshold operation [36], etc. A recently published resilience roadmap [24] underlines the expected increase of probability of failure (*pfail*) with scaling. Table 1 shows the pfail predicted in [24] for inverters, latches and SRAM cells due to random dopant fluctuations as a function of technology node (the trends for negative-bias-temperature-instability are similar). The table shows that the pfail of all types of devices increases dramatically with scaling. However, the table clearly indicates that not all types of devices are equally vulnerable; SRAM cells that are usually aggressively sized are distinctively more problematic.

A variety of well known approaches, such as column/row sparing and error correcting codes [29], have been adopted to provide fault-free chips by mitigating manufacturing faults and static parametric variations while preserving an important abstraction: *performance-invariability*. This is defined to be the expectation that two identical chips, or even two cores within a chip, when each is operating stand-alone under identical conditions have identical cycle by cycle behavior and, therefore, identical performance.

Tech Node	Inverter	Latch	SRAM
45nm	≈0	≈0	6.1e-13
32nm	≈0	1.8e-44	7.3e-09
22nm	≈0	5.5e-18	1.5e-06
16nm	2.4e-58	5.4e-10	5.5e-05
12nm	1.2e-39	3.6e-07	2.6e-04

Table 1: Predicted probability of failure (pfail) for different types of circuits vs technology node [24]

The central hypothesis of this paper is that the abstraction of performance-invariability is unrealistic to preserve for future processor chips due to the non-scalable cost of existing techniques to mitigate the mismatch between the scaling rates of area, voltage and static and dynamic variations. Replacing a chip and coarse grain disabling (e.g. core, cache bank/ways) may be acceptable when variability phenomena are rare but with increasing static and dynamic variations finer disabling and deconfiguration (e.g. individual functional units, cache blocks) will become economically necessary. We are going, therefore, to enter an era of *performance-variability*: functionally correct chips with variable degraded performance, intra-chip and across-chips, from the time when parts are shipped. The performance-variability will be due to lower resource capacity caused by the disabling of faulty cache blocks or units [32, 21, 36], timing misspeculation in the datapath due to timing variations [13], and extra mispredictions caused by faulty predictor cells [20].

In this paper, we propose a first-order analytical model for understanding the implications on performance of permanently faulty cells in architectural and non-architectural arrays. The model for a given program execution, micro-architectural configuration, and probability of cell failure, provides rapidly the *Performance Vulnerability Factor (PVF)*. PVF is a direct measure of the performance degradation due to permanent faults. In particular, the model can determine the expected PVF as well as the PVF probability distribution bounds for caches and prediction arrays without using an arbitrary number of random fault-maps.

PVF is analogous to the Architectural Vulnerability Factor (AVF) [23] proposed for assessing the vulnerability of architectural structures to soft-errors. PVF like AVF can be used by designers/researchers to appreciate and compare vulnerability of different structures and perform reliability driven trade-offs. However, PVF is complementary to AVF as it measures performance degradation due to permanent errors.

Virtually all previous micro-architectural work aiming to assess the performance implications of permanently faulty cells relies on simulations with random fault-maps, assumes faulty blocks are disabled [32, 21, 4, 1], and focuses on architectural arrays such as caches. These studies are, therefore, limited by the fault-maps they use that may not be representative for the average and distributed performance. Moreover, they are incomplete by ignoring faults in non-architectural

1072-4451/12 $26.00 © 2012 IEEE
DOI 10.1109/MICRO.2012.14

IEEE computer society

arrays, such as predictors, that do not affect correctness but can degrade performance. A narrow understanding of the consequences of permanently faulty cells can lead to a processor fault-reliability strategy that neither addresses key reliability challenges nor leverages reliability driven trade-offs.

A recent relevant study [27] proposes a method that given a cache, random probability of permanent cell failure, and an address trace, can calculate *without any fault-maps* the expected miss ratio when blocks with permanent faults are disabled. The model proposed in our paper, augments the method in [27], to predictors, can produce for caches and predictors the miss-rate and misprediction distribution bounds respectively, and provides the expected performance and performance distribution bounds for individual or combination of faulty arrays.

The paper establishes that for three predictors (a return address stack, a gshare direction predictor and an indirect jump predictor) and the last level cache used in this study, the proposed model assumptions are valid. The model is used to predict, for the same arrays and processor, the expected PVF and PVF distribution bounds at different technology nodes using projected cell failure rates due to random dopant fluctuations.

The experimental analysis shows that for cell pfail of up to 1.5e-6 the expected PVF is very small for any array or array combination. For higher pfail the PVF grows considerably mainly due to the extra misses in the last level cache. The expected PVF of all predictors remains small even at high pfail, but the PVF distribution reveals cases of significant performance degradation with small but noticeable probability. These results suggest that in the future designers can leverage trade-offs between PVF and reliability to improve efficiency. The paper demonstrates this approach by investigating the impact of different cell size on yield and PVF.

The remainder of the paper is organized as follows. Section 2 introduces the notion of PVF. The model that determines PVF is presented in Section 3. Experimental results are given in Section 4. Section 5 reviews related work and Section 6 concludes the paper and gives direction for future work.

2. Performance Vulnerability Factor - PVF

The *Performance Vulnerability Factor (PVF)* is a measure of the performance degradation that a processor will experience for a given benchmark run due to permanently faulty cells in its arrays. PVF takes values in the range $[0,1)$, with zero meaning no vulnerability at all (100% of performance) and for values near 1 almost zero performance. To define PVF, we rely on the notion of *Computation Capacity* (CC) as follows:

$$PVF = 1 - CC \tag{1}$$

where the computation capacity [5] is the fraction of the original pristine machine's performance that is still available given current hardware conditions and defined in our work as:

$$CC = \frac{C_{base}}{C_{base} + overhead * \#failures} \tag{2}$$

where C_{base} corresponds to the number of cycles of a fault-free run of a program by the microarchitecture under study, $\#failures$ is the number of failures, e.g. misses or mispredictions, resulting from permanently faulty cells and, *overhead* represents the mean increase in cycles of the overall execution time per failure.

For mathematical convenience, we introduce the notation ETV (Execution Time Vulnerability) which represents the normalized execution cycle increase of a program due to faults in arrays and defined as:

$$ETV = penalty * \#failures \tag{3}$$

where $penalty = \frac{overhead}{C_{base}}$ and corresponds to the normalized overhead of a single failure to the overall cycle count of a fault free run. By substituting ETV in Eq. 2 the computation capacity becomes:

$$CC = \frac{1}{1 + ETV} \tag{4}$$

PVF and expected PVF (noted $\overline{PVF}$) can be determined directly from ETV and $\overline{ETV}$ by using Eq. 1 and 4.

In the following section we describe a model that can determine the ETV, $\overline{ETV}$ and ETV distribution bounds for caches and prediction arrays.

3. ETV for non-architectural and architectural arrays

This section presents a first-order analytical model for predicting the ETV of an individual or combination of non-architectural and architectural arrays. First we present the model assumptions and then introduce the basic model. We then present how to use the model for non-architectural and architectural arrays. In particular, we focus on how to obtain the mean penalty per failure and the number of failures for each array. After, we describe a method to determine lower and upper ETV probability distribution bounds. At the end of the section, we discuss model limitations, extensions and uses.

3.1. Model Fault Assumptions

The model assumes that permanently faulty SRAM cells locations are random, each cell has equal probability of failure, and broken cells behave as stuck-at faults (the stuck-at assumption is only needed for predictors). The random behavior aims to capture some major causes of uncorrelated faults, such as line edge roughness and random dopant fluctuation. Also, the granularity of spatial correlation is large and within a chip the failures can be treated as uncorrelated [10].

The cells in processor can be divided into the following categories:

- can be faulty and not disabled - e.g. predictor bits
- can be faulty and must be disabled - e.g. cache block data bits
- cannot be faulty (fault free or replaced with a spare) - e.g. PC bits, block disable bit

The third category affects correctness and yield. Our model focuses on the first two categories, which affect performance. Based on the above cell classification, faulty entries in prediction arrays are used normally and can lead to extra mispredictions. In caches, on the other hand, blocks with faults are disabled and thus reduce the array capacity and lead to extra misses. For caches, an entry (i.e. a cache block) with at least one permanent fault is considered as faulty. Faulty cache blocks are assumed to be detected with post-manufacturing and boot-time tests, ECC, and built in self tests. Cache block disabling has been proposed for commercial processors in [22].

The model does not address the effect of rarely occurring transient errors, e.g. due to particle strikes because their effects are rare and short lived. In fact, they will cause at most a handful of mispredictions in prediction arrays and a repair when captured by ECC in caches.

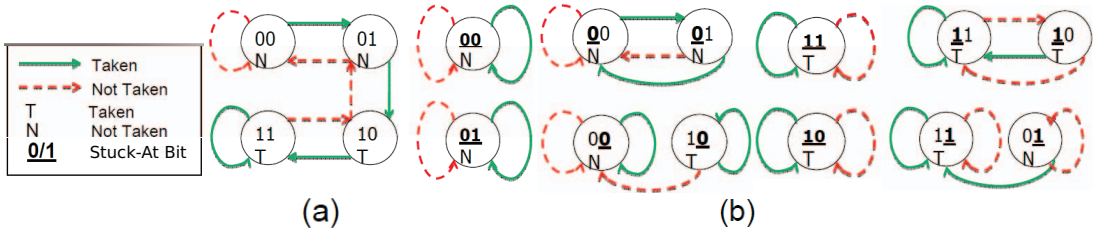


Figure 1: Operation of a 2-bit saturating counter used for prediction (a) Fault Free, (b) With Faults

3.2. Basic Model

At the core of determining PVF is the computation of ETV (Eq. 3) which can be obtained for a given program run by:

$$ETV = penalty * \#failures = \sum_{i=1}^{u} \sum_{j=1}^{M_i} P_{ij} \quad (5)$$

where u is the number of arrays with faults, M_i is the number of failures caused by faults in array i, and P_{ij} is the penalty for the jth failure in the program from array i. M_i corresponds to additional mispredictions (faulty-mispredictions) in case the unit i is a predictor, and additional cache misses (faulty-misses) in case i is a cache. Eq. 5 can be simplified to:

$$ETV = \sum_{i=1}^{u} ETV_i = \sum_{i=1}^{u} M_i \overline{P_i} \quad (6)$$

where ETV_i is the ETV of unit i and $\overline{P_i}$ is the mean normalized penalty for each failure in array i. Finally, the expected ETV_i of unit i can be obtained from:

$$\overline{ETV_i} = \overline{M_i} * \overline{P_i} \quad (7)$$

This is the same equation as Eq. 6 except $\overline{M_i}$ that represents the expected number of faulty-mispredictions or faulty-misses due to unit i.

The above model assumes that ETV is additive, i.e. we determine the ETV_i of each array i separately and simply sum them together to obtain the overall ETV. We discuss in Section 3.5 why this can be accurate.

Next we describe how to determine for non-architectural and architectural arrays: (i) $\overline{P_i}$, (ii) M_i when given a fault-map, (iii) $\overline{M_i}$ when given a random cell pfail but without fault-maps, and (iv) distribution bounds of ETV when given a random cell pfail without fault-maps.

3.3. ETV for non-architectural arrays

The focus of this part is on the performance implications of *permanently faulty SRAM cells* in non-architectural arrays. Today's high-end processors employ several non-architectural arrays for improving performance. Such arrays are used to predict: next instruction line, direction for conditional branches, target for branches (especially for return and indirect branches), addresses for prefetching, etc. In addition, non-architectural arrays are used to guide the updating of predictors (hysteresis bits).

Although faults can occur in any of these arrays, the performance implications of a faulty cell vary depending on how the value in a faulty cell is used in a pipeline and, in particular, on how a faulty cell influences performance. For example, a faulty cell in a conditional predictor, a return-address-stack and an indirect-jump predictor can cause a misprediction and a pipeline flush and, therefore, have a large misprediction penalty. In contrast, a faulty line-predictor cell has a smaller misprediction penalty because a line-predictor is usually corrected by a more accurate predictor within one or two cycles.

A faulty cell can have different implications depending on its semantic functionality. For example, a stuck-at fault in an 1-bit hysteresis, used to guide replacement on a misprediction, can lead to two behaviors: always replace or never update. Both behaviors can degrade performance but the second can be more grave. Fig. 1 illustrates how different combinations of stuck-at fault(s) at different bit positions transform the operation of a 2-bit saturating counter used for direction prediction. It is interesting to observe that such faults mainly result in always taken or always not-taken behavior.

Another key parameter that influences the vulnerability of a non-architectural array to faults is the distribution of accesses. A permanent fault in a frequently accessed entry is likely to cause more degradation than a fault in an infrequently accessed entry. Also, a program with many "hot" entries has higher probability to have a faulty hot entry. However, for the same number of accesses the more the hot entries the lower the impact from each.

The bias of the values stored in a non-architectural array also influences the performance with faults. For instance, an entry that contains bits that are highly biased will experience worst case degradation when at least one of these bits is stuck-at in the opposite value of the bias.

The above discussion reveals that performance is not equally vulnerable to faults across non-architectural arrays and even within an array. Furthermore, there is variation due to differences in the dynamic behavior between programs, such as the access distribution, instruction mix, predictability and misprediction penalty. For example, it is obvious that a program with or with low branch predictability will not suffer significantly from faults in the predictors.

Below we define the basic analytical model for determining the performance impact of permanently faulty cells in non-architectural arrays.

3.3.1. Model for non-architectural arrays

According to Eq. 6, the ETV_i, of a non-architectural array i, can be obtained from M_i and $\overline{P_i}$. $\overline{P_i}$ can be obtained by linear regression of the additional cycles and additional mispredictions from runs that inject randomly stuck-at faults in unit i[1]. It is known from previous work [15] that misprediction penalty is sensitive to the number of mispredictions, but our approximation for $\overline{P_i}$ is found to be accurate for most benchmarks and non-architectural arrays we considered in this study. If more accuracy is needed future work can consider models that correlate penalty to other parameters [15].

The M_i for a non-architectural array i and a program run, can be obtained using the *access-map* and *bias-maps* of the array. An *access-map* represents the number of correct predictions from each entry

[1]We used ten runs for each of the following fractions of faulty bits in unit i: 0.5, 2 and 8%

	Bias0	Bias1
0	30	12
1	6	47
2	23	30
3	15	15

	Fmap0	Fmap1
0	0	0
1	0	1
2	1	0
3	0	0

Figure 2: M_i computation example for a 4 entry 1bit/entry predictor. The predictor has a stuck-at-1 bit in entry 1 resulting in 6 extra mispredictions (Bias0 map) and a stuck-at-0 bit in entry 2 resulting in 30 extra mispredictions (Bias1 map).

during a fault free run of the program. The access map is further broken into *bias-maps*, *bias0* and *bias1*, that indicate for each bit in the array, during the fault free run, how many times it is part of a correct prediction with value 0 and 1 respectively.

Next we explain how to use the access and bias maps to obtain the M_i when given a fault-map, and to determine the expected $\overline{M_i}$ and $\overline{ETV_i}$ when given a cell pfail.

3.3.2. M_i from a fault-map

We can obtain M_i for a specific fault-map, that indicates whether a bit is faulty and its stuck-at value, by taking effectively the dot product of the fault-map and the bias-maps. This is illustrated in Figure 2 for a single-bit per entry predictor. For a 2-bit saturating predictor, this approach works as well because, as shown in Figure 1, a stuck-at bit in such a predictor, can lead to always taken or always not-taken predictions.

We limit the fault-maps used for non-architectural arrays to at most one faulty-bit per entry to keep the bias-maps simple, as multiple faulty bits in an entry are very unlikely for the pfail range considered in this work.

3.3.3. Expected ETV_i and M_i for a given cell pfail

The expected ETV_i, for a benchmark running on a processor with faulty cells in a non-architectural array i, can be obtained from Eq. 7. The term $\overline{M_i}$ represents the expected number of mispredictions due to faults in unit i. $\overline{M_i}$ can be obtained probabilistically for a given cell pfail without using fault-maps as follows:

$$\overline{M_i} = \frac{\sum_{j=1}^{entries_i} access\_map_j}{2} * \overline{pfail\_entry_i} \tag{8}$$

where $entries_i$ represents the total number of entries in predictor array i, $access\_map_j$ is the number of correct predictions of entry j during a fault free run, and $\overline{pfail\_entry_i}$ is the expected probability of having a faulty entry determined as follows for an n bit entry:

$$1-(1-pfail)^n \tag{9}$$

The halving in Eq. 8 captures the probability of a fault being stuck-at 0 or 1.

The non-architectural PVF model is valid as long as the access-map and bias-maps of an array, for a given program run, are virtually insensitive to pipelining effects, i.e. the number of correct predictions is insensitive to pipelining effects. Fortunately, this is the case for prediction arrays that are not speculatively updated or when they are speculatively updated they are repaired from wrong path effects [18, 31].

Fault Map: 1 0 3 2

	LRU+3 (MRU)	LRU+2	LRU+1	LRU
SET 0	120	100	110	60
SET 1	150	140	110	55
SET 2	180	134	80	50
SET 3	220	200	100	30

Figure 3: M computation example for a 4 set 4 ways LRU cache. $M = 454$ resulting from 1 faulty block in set 0, 3 faulty blocks in set 2 and 2 faulty block in set 3.

3.4. ETV for architectural arrays

Nowadays, caches take most of the real-estate in processors and contain numerous SRAM cells. As explained before, we assume that blocks that contain permanently faulty cells are disabled and thus reduce the cache capacity.

Architectural arrays vulnerability, similar to non-architectural, depends on the distribution of hit accesses across sets and within sets and cache miss latency. Consequently, the PVF for architectural arrays can vary across programs and across sets, and, it depends on the dynamic program behavior. The model detailed hereafter is proposed to assess the performance impact of permanently faulty SRAM cells in architectural arrays.

3.4.1. Notations and Assumptions

A cache configuration is defined by the number of sets s, ways per set n, and block size in bits k. The model is defined for the LRU replacement policy, the generalization of the model to other replacement policies is left for future work.

3.4.2. Model for architectural arrays

The ETV_i, caused by faulty blocks in cache i, can be obtained using Eq. 6. For caches, M_i corresponds to the number of additional misses caused by faulty blocks, and $\overline{P_i}$ is the average normalized penalty per additional miss. $\overline{P_i}$ can be determined by linear regression using additional cycles and additional misses from handful runs[2].

The number of additional misses M_i for a program run is determined by using the *access-map* of the cache during a fault free run of the program. An *access-map* represents the number of hits per set and per LRU position during a fault free run. In the access-map, a row corresponds to a set and each column corresponds to a position in the LRU stack.

Next we present how to use cache access maps to obtain the M_i when given a cache fault-map, and to determine the expected $\overline{M_i}$ and expected $\overline{ETV_i}$ when given a cell pfail.

3.4.3. M_i from a fault-map

Given an access-map, we can obtain M_i for a specific fault-map of cache i, which indicates the number of faulty blocks w per set, by simply adding the accesses to the last w columns as illustrated in Figure 3 and suggested by [25]. It can be noticed that, thanks to the LRU replacement policy, the exact position of the faulty-blocks are not needed because the LRU stack is reduced by the number of faulty blocks in each set.

3.4.4. Expected ETV_i and M_i for a given cell pfail

The expected ETV_i, for a benchmark running on a cache with faulty cache blocks, can be obtained from Eq. 7. The term $\overline{M_i}$ represents the expected number of additional misses due to faulty blocks. $\overline{M_i}$

[2]We used n (associativity of the cache) runs by considering 1 way faulty in each set, 2 ways faulty etc.

can be obtained analytically for a given cell pfail without using fault-maps as proposed in [27][3]. To determine $\overline{M_i}$, we first compute the probability of a cache block failure p_{bf} with Eq. 9 for k bits. Then, the probability pe_i for i faulty ways in a set can be determined based on the well-known binomial probability:

$$pe_i = \binom{n}{i} p_{bf}^{i} (1 - p_{bf})^{n-i} \tag{10}$$

which can provide, for every value of $i[0 \ldots n]$, the probability of having i faulty ways. This distribution provides insight about how likely it is to have a given number of faulty ways in a set and, can be used to obtain the expected number of additional misses. The expectation of a random variable $X = x_1, x_2, \ldots, x_n$ for which each possible value has probability $P = p_1, p_2, \ldots, p_n$ is given by:

$$E[X] = \sum_{i=1}^{n} x_i * p_i \tag{11}$$

In our case, the random variable X corresponds to the total number of additional misses in a cache set with faults, x_i corresponds to the total number of additional misses when there are i faulty ways in a set, noted hereafter $m_{i,set}$, and p_i the probability of having i faulty ways in a set. $m_{i,set}$ is simply determined by adding the last $n-(i-1)$ columns of the access map of the set:

$$m_{i,set} = \sum_{j=n-(i-1)}^{n} access\_map[set][j] \tag{12}$$

Therefore, the expectation of the number of additional cache misses $\overline{M_{set}}$ for a given set can be expressed as follows:

$$\overline{M_{set}} = \sum_{i=1}^{n} m_{i,set} * pe_i \tag{13}$$

Finally, we sum this expectation for each set to get the expected number of additional misses $\overline{M_i}$.

We note that the ETV model requires that the access-map and the number of cache hits per set and LRU position, for a given program run, to be rather insensitive to pipelining effects. Our empirical analysis presented in Section 4 reveals that this assumption is true.

3.5. Why are ETV_is additive?

Eq. 6 assumes that ETV_i for each array i is additive when computing the overall ETV. It is known from previous work [30, 14] that, the penalty of different units can overlap, thus, how can this additive claim be accurate?

The proposed method for the penalty estimation of an array i considers the overall execution increase due to additional misses or mispredictions while it captures the overlapping and interactions with other pipelining effects and events as well as the non-constant main memory latency. Specifically, the penalty $\overline{P_i}$, for an array is obtained by activating faults while in the pipeline there can be other concurrent misses, mispredicts, stalls, memory accesses etc.

The simple summation of ETV_is is found to be accurate for the benchmarks, faulty arrays, and microarchitecture used in this study. If more accuracy is needed, future work can consider the detail interaction between units [30, 14].

[3]The model used in this section to estimate $\overline{M_i}$ is from [27] and is presented here for completeness.

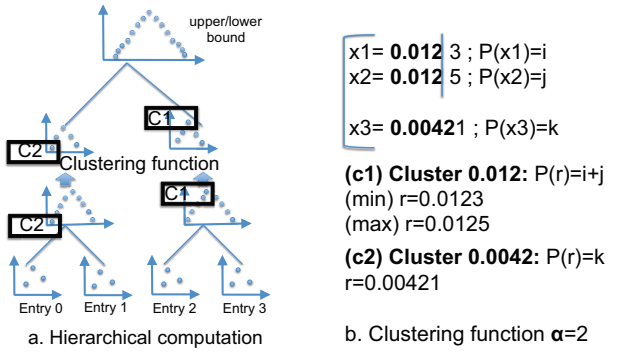


Figure 4: Hierarchical distribution computation and clustering function example.

3.6. ETV probability distribution bounds

To draw more insight about the PVF, we introduce an analysis that determines the lower and upper PVF probability distribution bounds for an individual or any combination of arrays for a given cell $pfail$. These bounds are computed in two steps: (i) estimation of a discrete ETV probability distribution per entry for each unit i, (ii) combining the multiple discrete ETV probability distributions by using a heuristic to determine a lower bound and an upper bound of the distribution. Although, not discussed further, the heuristic can be trivially used to produce the M_i probability distribution bounds by using Eq. 6.

Estimation of a discrete probability distribution. For predictors, a discrete ETV probability distribution is first determined per entry. For each entry, this distribution is determined by the sum of probabilities that can cause a given ETV by considering the following cases:

- $ETV = 0$ which corresponds to a fault free entry. The corresponding probability is $1 - \overline{pfail\_entry_i}$
- ETV_{bias0}[4] which corresponds to the ETV caused by a stuck-at-1 bit in the entry. The corresponding probability is $\overline{pfail\_entry_i}/2$. The ETV_{bias0} can be obtained using Eq. 6.
- ETV_{bias1} which corresponds to the ETV caused by a stuck-at-0 bit in the entry. The corresponding probability is $\overline{pfail\_entry_i}/2$.

For a cache, the discrete ETV probability distribution is determined at the granularity of a set. We obtain at most $associativity + 1$ different ETVs that correspond to 0 up to all blocks faulty in a set with the probability for each case given by pe_i.

Combining multiple discrete probability distributions. To determine the resulting ETV probability distribution of an array or combination of arrays, we need to combine together all the previously computed distributions. It is well known that exhaustive combining is infeasible due to a combinatorial explosion [16]. To avoid this problem we propose a parametric (α) heuristic to determine a lower bound and an upper bound of the ETV distribution.

A hierarchical approach [16] is used based on a binary tree representation as illustrated in Figure 4.a. Each leaf represents the distribution of an entry for predictors or the distribution of a set in a cache. The root represents the resulting upper/lower distribution bound and each intermediate node an intermediate joint distribution.

[4]For arrays that predict multi-bit values, like the return address-stack, ETV is determined for each bit and the probability is further divided by the number of bits per entry. This approach is similarly used for ETV_{bias1}

To avoid the combinatorial explosion, after obtaining each intermediate distribution (at each intermediate node), we use a clustering approach by summing probabilities when ETV values are close enough.

This strategy uses a clustering function as illustrated in Figure 4.b. It starts with a truncation of each value (i.e. ETV) after the first α non-zero digits[5]. Equal resulting values are then grouped together and their respective probability are summed. The value r representative of each group is the minimal/maximal value of the group before the truncation. This ensures (see the proof in appendix) a lower/upper bound ETV probability distribution.

3.7. How many runs are needed to determine the input values of the analytical model?

In this section, we summarize the number of runs needed by the analytical model to determine the expected PVF and the PVF distribution bounds. We like to note that the PVF analysis assumes a fixed microarchitecture. Therefore, for each distinct microarchitecture a separate PVF analysis is required.

For each array under study the model needs information provided by a single run without faults for each program that is analyzed. This fault-free run is performed to collect the *Cbase*, program baseline performance, and the access-maps of each array. The access-maps obtained for predictors and caches are then used to compute the expected PVF and PVF distribution bounds.

The mean penalty, $\overline{P_i}$, for a unit i is obtained per program by running it on the given microarchitecture with increasing number of faults in the unit i. Typically k_i (=30 in this study[6]) runs for each predictor and as many runs as the number of ways, w_i, in cache i are sufficient to observe how the overall execution increases as a function of additional faulty mispredictions and misses.

Therefore, for one benchmark the total number of runs needed to determine the input values for the analytical model is given by:

$$1 + \sum_{i=1}^{\#arch\_units} w_i + \sum_{i=1}^{\#non\_arch\_units} k_i$$

So assuming 26 benchmarks, 3 predictors and an 8 way cache we need to perform 2574 simulations to completely characterize the microarchitecture.

The model key strengths is that once the characterization is complete, the model does not require performing new simulations when changing cell pfail. It can very fast and accurately obtain the expected PVF (few seconds) and the PVF distributions (1 minute on a current machine) for different pfails.

To put in perspective, to produce the model results using random fault maps and simulations will require exponential to the number of entries and sets runs for each unique pfail. The benefits of our analytical model as compared to current practice depend on the number of distinct pfail to consider and the number of fault-maps to simulate. The proportion of runs needed between our model and current practice for a benchmark is:

$$\frac{1 + \sum_{i=1}^{\#arch\_units} w_i + \sum_{i=1}^{\#non\_arch\_units} k_i}{\#pfails * \#faultmaps}$$

The denominator is the number of runs for current practice and the numerator is the number of runs needed to determine the input values of the model (100 is an upper bound in our study). Our cost is thus a small number of runs whereas a random fault map methodology may require huge amount of runs to obtain the expected PVF and its distribution.

3.8. Model limitations

The model is a first-order approximation and can underestimate the PVF in some cases. In particular, for prediction arrays when multiple bits of the same entry happen to be faulty and their combined mispredictions is more than their individual contributions, then the expected PVF is underestimated. For arrays with 1-bit entries or 2-bit saturating counters this is not a problem since the faulty mispredictions are determined uniquely by the faulty bits, the faulty value and the bias (see Fig. 1) and, therefore, M_i can be estimated at the granularity of an entry. For arrays that predict multi-bit values, like the return address-stack and the indirect jump predictor, this is not a problem as long as the fault-maps are random and the number of faults is small enough that render very unlikely to have more than one faulty bit in a frequently accessed entry. These assumptions are reasonable for the configuration and parameters used in this study (in particular for pfail $\leq$0.001).

It is possible that faults in the prediction arrays and caches can result in a significant increase in the cache accesses and misses that may not be captured by our model. We have not observed such behavior for the pfails considered in this study.

All these limitations are directions for future model improvements if higher model accuracy is required.

3.9. Model Extensions

The model, in its current version, is defined for three predictors and a cache of a single core. The model, however, should be applicable to other arrays as long as we can derive a method to obtain the number of extra events due to faults and their corresponding penalties. For non-arrays, like functional units, we believe that the model can be extended based on the probability of timing violations [13, 19]. Finally, for multi-cores the impact on performance of faults in shared resources, such as caches and interconnect, will need to be modeled while also considering the interactions between benchmarks.

With the currently modeled structures, we observe that the performance degradation depends on the penalty and on the number of accesses to a structure. The more accessed an entry is and the higher the penalty of an extra miss or misprediction, the higher the performance degradation when the entry is faulty. We believe that this observation will also hold for other structures.

3.10. Model Uses

The model, once derived, can be used to explore processor behavior with different cell pfail. This can be helpful to forecast how processor performance may be affected by faults in the future. Additionally, this information can be useful to explore the use of different cell sizes that enable a trade-off between area and PVF. Another use of the model is to determine which arrays have significant PVF and make design decisions to reduce their PVF, for example through a protection mechanism, using larger cells, or even by selecting a different array organization. The PVF distributions bounds can help establish how a population of chips is affected due to faults in predictors and caches. Such binning provides an indication about

[5] α is a parameter that determines the precision and allows to trade-off accuracy for computation time.

[6] k_i is determined empirically. Furthermore, k_i has been validated by a sensitivity analysis over the penalty values, which reveals that in most of the cases there is a range of values around the determined penalties that provides the same PVF.

Parameter description	Setting
Pipeline depth	15 stages
Fetch/Decode/Issue/Commit	up to 4/4/6/4 instr. per cycle
Line Predictor	4096 entries
RAS	16 entries (31 bits per entry) x 2 (fetch and commit)
Indirect Jump Predictor	512 entries (31 bits per entry)
Branch Predictor	8 KB gshare (32768 entries - 2 bits per entry, 15 bits history)
Branch Resolution	In-order
Issue Queue/Reorder buffer	40 INT entries, 20 FP entries/128 entries
Functional Units	4 INT ALUs, 4 INT mult/div, 1 FP ALUs, 1 FP mult/div
L1 instr./data cache	64 KB, 2-way, 64 B blocks, 1-cycle, LRU / 64 KB, 2-way, 64 B blocks, 3-cycle, LRU
L2 unified cache	2 MB, 8-way, 64 B blocks, 12-cycle hit latency, 255 cycles miss latency, LRU

Table 2: Processor Configuration

Benchmark	Baseline	Conditional	Returns	Indirect	L2 cache	Accuracy			Overheads			
	IPC	branches (M)		jumps	MPKI	Gshare	RAS	Ijump	Gshare	RAS	Ijump	L2 cache
ammp00	1.54	8.8	21K	0	1.19	0.97	1.00	0	20	25	0	58
applu00	0.52	0.6	100	94	22.04	0.99	0.99	0.68	33	41	0	51
apsi00	2.35	3.5	58K	0	0.82	0.99	1.00	0	25	29	0	187
art00	1.82	8.6	110	0	0.25	0.99	0.99	0	45	27	0	70
bzip00	1.29	14.4	353K	0	0.82	0.95	1.00	0	17	18	0	62
crafty00	1.90	8.7	1.09M	209K	0.17	0.96	0.99	0.66	24	24	25	66
eon00	1.26	7.0	2.04M	571K	0.01	0.99	1.00	0.76	14	16	13	48
equake00	0.64	1.7	1.06M	0	12.73	0.97	1.00	0	39	28	0	55
facerec00	1.58	6.7	166K	0	4.22	0.98	1.00	0	23	24	0	43
fma3d00	1.43	16.3	1.43M	278K	0.04	0.96	1.00	0.83	19	25	25	76
galgel00	2.42	5.8	0	0	0.24	0.99	0	0	29	0	0	55
gap00	1.56	9.5	2.05M	1.53M	1.01	0.99	0.99	0.77	23	25	26	38
gcc00	1.17	6.9	478K	213K	4.02	0.97	1.00	0.59	17	27	34	74
gzip00	1.54	7.0	1.05M	13	0.17	0.94	1.00	0.77	20	20	0	94
lucas00	0.70	1.3	0	0	13.39	1.00	0	0	9	0	0	40
mcf00	0.13	20.4	3.33M	0	86.42	0.96	1.00	0	58	68	0	60
mesa00	1.73	5.9	1.19M	547K	0.20	0.97	1.00	0.99	29	25	27	35
mgrid00	0.82	0.4	327	154	10.58	0.99	0.99	0.90	16	0	0	66
parser00	1.19	12.3	1.99M	240	1.37	0.96	0.99	0.90	20	21	0	82
perlbmk00	1.28	9.2	2.11M	1.54M	0.19	0.99	1.00	0.77	21	16	23	48
sixtrack00	1.96	2.3	128	24K	0.29	0.99	1.00	0.80	28	34	38	50
swim00	0.40	2.3	66	46	26.36	0.99	1.00	0.57	24	23	0	45
twolf00	1.06	10.6	705K	0	0.25	0.90	1.00	0	20	24	0	86
vortex00	1.86	10.8	2.06M	79K	0.34	0.99	0.99	0.95	23	24	23	81
vpr00	0.72	9.7	647K	49	6.13	0.94	1.00	0.93	23	31	0	72
wupwise00	1.78	8.8	652K	18	2.79	0.99	1.00	0	25	17	0	85

Table 3: Benchmark Characteristics

how many chips will be affected by failures and up to what extent. Finally, the PVF analysis can provide a first order timing analysis for systems with performance constraints such as real time systems.

The above types of studies are facilitated by the proposed methodology, since being analytical it allows for quick exploration, without long micro-architectural simulation or fault maps generation.

4. Experimental Results

4.1. Experimental setup

In our experiments, the validated cycle accurate simulator *sim-alpha* [12] is used. We have extended it to measure the performance implications of faults in three prediction arrays: a return address stack, a gshare direction predictor and an indirect jump predictor; and the L2 cache of a high performance out-of-order superscalar processor. Key parameters of the processor configuration are summarized in Table 2. For L2 cache, we consider blocks comprised of 64 bytes for data, 11 bits for its ECC, 25 bits for the tag, 3 control bits for valid, disable and dirty states and 7 bits for the tag ECC. The experiments are conducted using SPEC CPU2000 benchmarks. The applications characteristics are summarized in Table 3 that also shows the estimated overhead per failure for each benchmark and structure. An in-house SimPoint [17]-like tool is used to select the regions to simulate and run them for 100M committed instructions.

Two types of experimental results are reported: *simulation based* (Section 4.2.1) and *model-based* (Section 4.2.2).

The simulation based results are used to validate the accuracy of our model. The validation compares the values obtained by simulations against the values predicted by the model presented in Sections 3.3.1 and 3.4.2. The validation runs used 1000 fault maps. A *fault-map* represents the location of the faulty entries. Each fault-map represents a processor with faults and contains the locations of faults in the prediction arrays and the L2 cache. The validation fault-maps are randomly generated with cell probability failure of 1e-3 (lower probabilities have also been used to validate the model). For prediction arrays, unlike architectural arrays, it is not sufficient to know that a cell is faulty but we also need to know what is the faulty value. Therefore, for the prediction arrays, each fault-map is paired with a value-map that contains, for each fault location, a randomly generated fault value. For each fault map, the PVF of the predictions arrays and the L2 cache are estimated for each of the benchmarks using the model. If the PVF is 1% or more for both the prediction arrays and the L2 cache, a detailed performance simulation is performed. Out of the 26000 possible runs, only 1847 are predicted to produce PVF

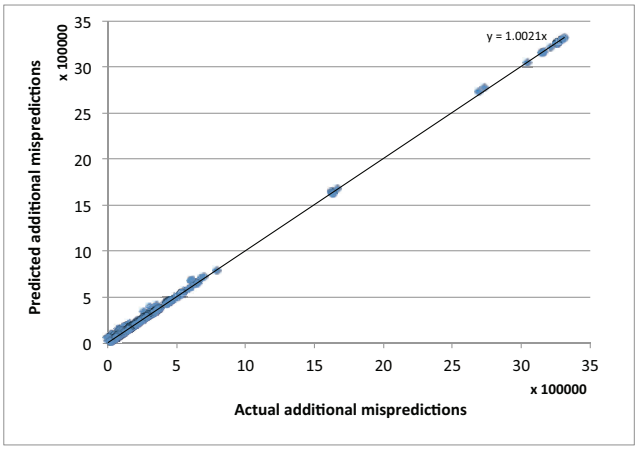


Figure 5: Actual vs. Predicted number of additional mispredictions with pfail=0.001.

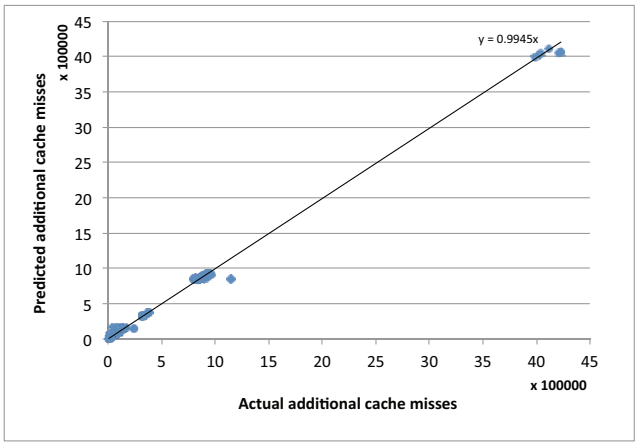


Figure 6: Actual vs. Predicted number of additional L2 misses with pfail=0.001.

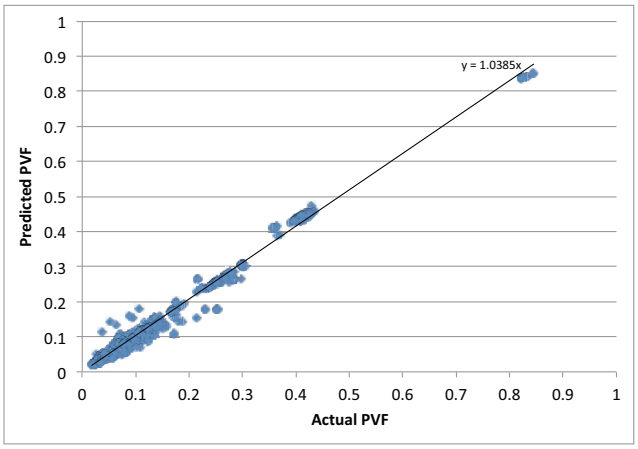


Figure 7: Actual vs. Predicted PVF with pfail=0.001.

more than 1% for both type of arrays.

For the model based results, we determine the expected PVF and the distribution bounds. The pfails values in [24] for technologies ranging from 32nm to 12nm (7.3e-09 to 2.6e-04) are used in this study. To account for aging phenomena and lower voltage, we use cell pfail of 1.0e-03, which also falls in the range considered by many recent studies [11, 1, 4, 36, 26, 2]. Finally, we explore for the L2 cache the impact of more robust but larger cells on the expected PVF and yield.

4.2. Experimental results

4.2.1. Simulation based results

The simulation based results are used to validate the model. Each point in Figures 5 and 6 shows the additional mispredictions (respectively additional misses) predicted from the model and the actual obtained through simulation for different benchmarks and fault-maps when cell pfail is 0.001. On each figure, the linear regression of all points and its corresponding equation is shown. We can observe from Figure 5 that the prediction model for additional mispredictions is very accurate (3.79% error on average) with virtually all predicted and actual values being equal. Figure 6 shows the same trends for the additional misses (3.34% error in average) except for two cases which are underestimated. We further analyzed these cases and found that for both cases the underestimation comes from the interaction with the faulty predictors which increase significantly the number of accesses to the L2 cache. We have validated this hypothesis by using only the cache fault-map and found that the predicted and actual values match in that case.

Figure 7 compares the PVF predicted from the model and the actual obtained through simulations. We observe again that the proposed model is quite accurate (7.15% error in average) with most of the predictions being close to the actual experimental outcome. However, there is more deviation in Figure 7 as compared to Figures 5 and 6. Considering that the additional misprediction and misses are predicted very accurately, the deviation is attributed to the variability in the penalties which are not captured by the proposed approach.

Overall, the results suggest that the model is quite accurate and also validate the model assumption that the ETV of different prediction arrays and the L2 cache are additive.

4.2.2. Model based results

This set of experiments estimates the expected PVF for cell pfails that correspond to different technology nodes [24]. Results are shown in Table 4. In this table, the first two columns show the technology node and the corresponding pfail, the next six columns show the maximal expected PVF in at least one benchmark for gshare, ras, ijump, all the predictors together, L2 cache and the global expected PVF. The last column shows the expected PVF for a composite benchmark which treats the consecutive benchmark execution as a single benchmark.

As shown in Table 4, the expected performance degradation from permanent faults in predictors is small and at most 2% when pfail equals to 1e-03. For the L2 cache, the performance degradation observed at current technology nodes is small, but, at pfail=2.6e-04 and pfail=1e-03 the maximal expected PVF is close to 30% and 84% respectively. Further analysis, not shown, reveals one benchmark, art00, experiences large PVF. The composite expected PVF for the different benchmarks at the two highest pfails is low but still significant at 2% and 14% respectively. Our model, therefore, can be useful to estimate at which point the performance degradation due to permanent faults can be tolerated and when it starts to become problematic and needs to be addressed with fault tolerance techniques.

The lower PVF of the predictors as compared to the L2 cache is mainly due to the size and the organization of each structure: number of total bits and number of bits per entry. Specifically, for the same cell pfail the expected number of faulty bits in the L2 cache will be $L2\_size\_in\_bits/Predictor\_size\_in\_bits$ times more. For example,

Technology	pfail [24]	Max Expected PVF						Composite Bench.
		Gshare	RAS	ijump	Predictors	L2 cache	**global**	Expected PVF
32nm	7.3e-09	0.00001%	0.00001%	0.00001%	0.00002%	0.00007%	0.00007%	0.00003%
22nm	1.5e-06	0.00106%	0.00213%	0.00111%	0.00347%	0.01441%	0.01533%	0.00621%
16nm	5.5e-05	0.03874%	0.07792%	0.04055%	0.12697%	1.70053%	1.73797%	0.25661%
12nm	2.6e-04	0.18286%	0.36615%	0.19083%	0.59581%	29.96965%	30.05939%	1.95337%
-	1.0e-03	0.69939%	1.37861%	0.72203%	2.23256%	83.71776%	83.73641%	14.60214%

Table 4: Expected Performance Vulnerability Factors (PVFs) for different technology nodes

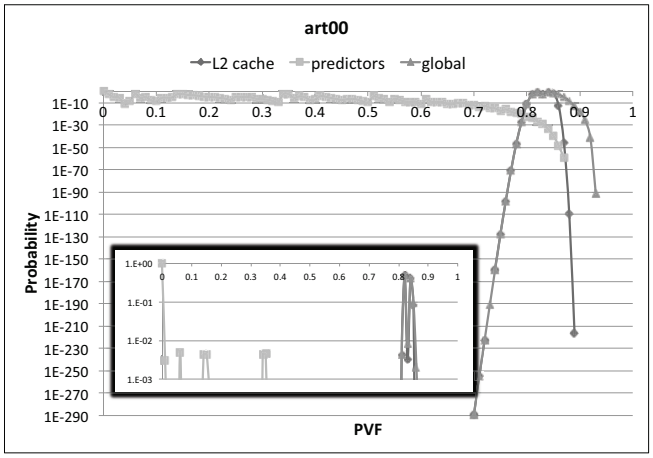


Figure 8: art00: average of the lower and upper distribution bounds. pfail=0.001 ; $\alpha = 2$.

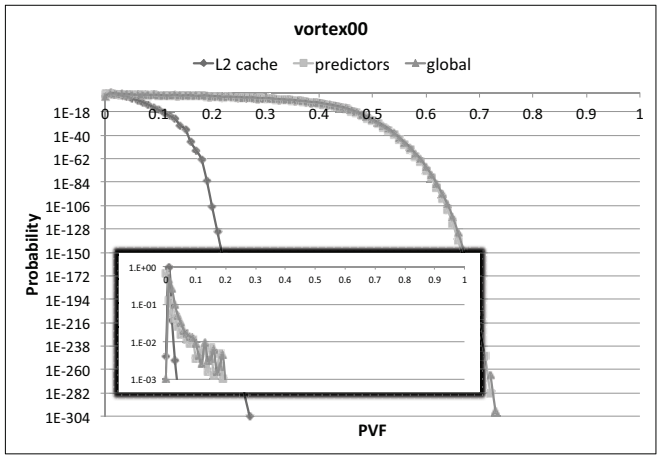


Figure 9: vortex00: average of the lower and upper distribution bounds. pfail=0.001; $\alpha = 2$.

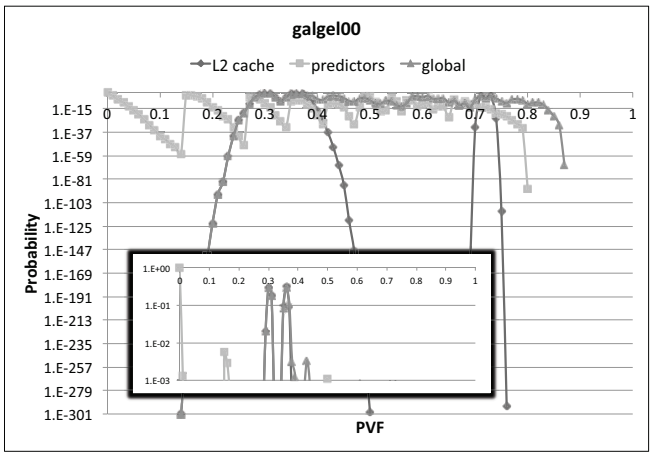


Figure 10: galgel00: average of the lower and upper distribution bounds. pfail=0.001; $\alpha = 2$.

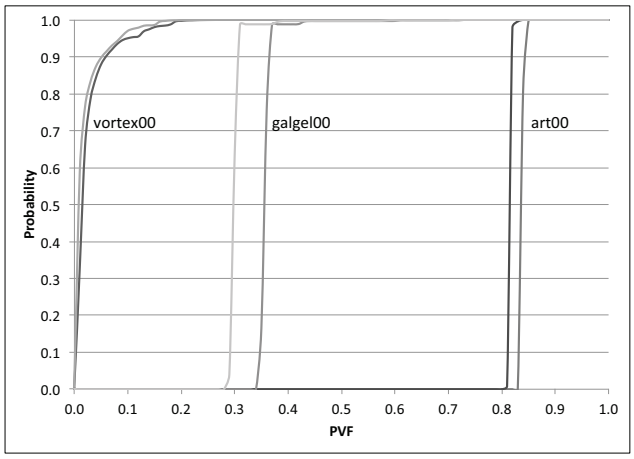


Figure 11: Cumulative distribution of the lower and upper distribution bounds for all units together. pfail=0.001 ; $\alpha = 2$.

for pfail=1.0e-3 the L2 cache used in this study will experience about 18000 faulty bits whereas the gshare predictor 65. Furthermore, the impact of a faulty bit in the L2 is significantly more pronounced since the size of a block is typically much larger than a predictor entry. We observe that the performance degradation due to a structure depends on the penalty and on the number of accesses to that structure. The more accessed an entry is and the higher its penalty, the higher the performance degradation when the entry is faulty.

PVF distribution bounds: To draw more insight about the expected PVF and to show that the range of possible PVF values can be quite distant from the expectation, we determine the lower and upper bounds of the PVF distributions. For reading ease, we show only three benchmarks representative of the different cases we observed.

Figures 8, 9, and 10 present the average of the two distribution bounds for the last level cache, the predictors and all units together. Figure 11 shows the cumulative distribution of the two bounds for the three benchmarks when all units are considered together to highlight the small distance between the bounds (i.e. small error).

The first observation is that the combined distribution of all units is dominated by the distribution of the unit that has the highest expected PVF (the cache for art00 and galgel00, the predictors for vortex00).

Moreover, the distribution results help to reveal the shortcoming of analysis based only on expected values and limited number of random fault maps: they cannot reveal the shape of the distribution. For instance, even if the highest PVF probability is close to the expectation, there is a significant probability to suffer a much higher PVF. In terms of population of processors, this will mean that a significant number will experience a PVF higher than the expectation. For example, with vortex00, 1 per 1000 processors will experience a PVF near 0.2 while the expected PVF is only 0.02.

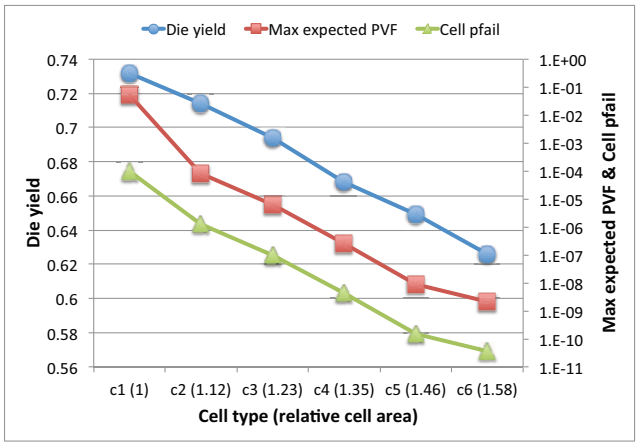


Figure 12: Die yield vs. Max Expected PVF for different L2 cell size (32nm).

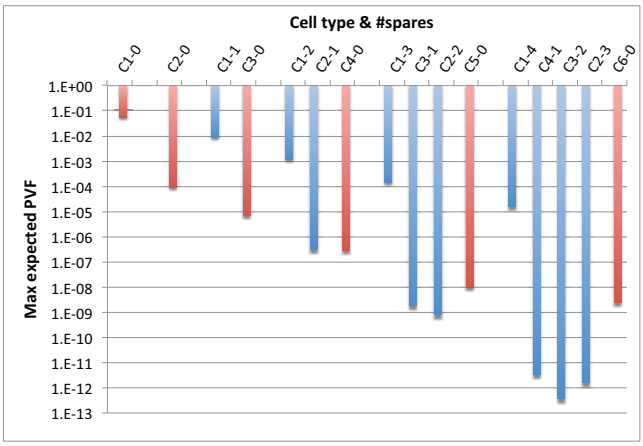


Figure 13: Max Expected PVF for different L2 cell size (x) and spare blocks (s). The configurations Cx-s are sorted in decreasing order according to their corresponding estimated die area.

The main cause for this difficult to detect behavior, when using fault maps, is that accesses are not evenly distributed across entries and, in general, it will require many fault-maps to produce a representative distribution.

Finally, the cumulative distributions in Figure 11 show that the bounds are accurate for all benchmarks with a small error, 0.027 on average and up to 0.07, when α is set to 2. In terms of computation time, computing the two bounds for all units together takes 1 minute per benchmark on average, when running on a typical desktop.

4.2.3. Case study: design trade-offs using PVF

As shown above, for the same cell pfail the L2 cache has higher PVF than the predictors. An approach that can reduce the L2 PVF is to use more robust cells. On the other hand, robust cells require more die area. This is especially true for large size L2 cache which occupies a significant part of the total processor's area and thus will lower the die yield. To assess this trade-off, we use six different cells sizes with their corresponding pfail from [37] for 32nm technology. Figure 12 shows the die yield[7], the maximal expected PVF for the different L2 cell size and their corresponding pfail. The results show that by choosing a less robust cell (c2 in this case) instead of the most robust one (c6), the PVF remain low (1.0e-4 vs. 1.0e-9) with a significant improvement in die yield (0.72 vs. 0.63). Furthermore, the cell with higher yield (c1) results in the largest PVF (close to 0.05). This first order analysis helps identify that c2 can be a good compromise.

Another approach to reduce the PVF is to use spare blocks (for instance s per set). The notion of spares can be easily incorporated in our model by changing Equation 10 as follows:

$$pe_i = \binom{n+s}{i+s} p_{bf}^{i+s} (1-p_{bf})^{n-i}$$

which provides, for every value of i $[1...n]$, the probability of having $i+s$ faulty blocks. To assess the benefits of sparing, Figure 13 shows the expected maximal PVF when using different cell size and number of spare blocks. Each configuration is noted Cx-s where x is the cell type and s is the number of spare blocks per set. In the figure, the different configurations are sorted in decreasing order according to their estimated area/yield. As shown in the figure, using spares with the less robust cell (c1) is not a good solution because it decreases significantly the die yield to achieve the same PVF as c2 without spares (PVF(C1-3)$\sim$PVF(C2-0)). Nevertheless depending on the targeted PVF threshold, it can be better to use relatively robust cells with some spares to maximize the die yield instead of using only the most robust cells. C2-2, for instance, gives a lower PVF and a better die yield as compared to C5-0 and C6-0.

The two cases studies highlight how the rapid exploration of PVF and area trade-offs can help designers configure and optimize their array designs.

[7] $die\ yield = wafer\ yield * (1 + \frac{die\ area * defects\ per\ unit\ area}{\alpha})^{-\alpha}$, where $wafer\ yield = 1$, $defects\ per\ unit\ area = 0.4$, $\alpha = 4$ and the die area for the processor is estimated by using hotfloorplan [28] in our experiments.

5. Related Work

Previous work investigating the implications of permanently faulty cells in processor arrays focused on caches and considered yield and performance analysis.

Yield analysis provides the probability distribution in terms of number of faulty blocks expected in a cache given a specific cache configuration and a random probability for permanently faulty cells. Such analysis is usually based on binomial probability and helps determine the expected fraction of fault-free caches and number of spares that may be required by the caches with faults [2, 26, 36, 4, 1, 11]. The cell pfail depends on several parameters including technology node, failure sources, such as static process-variations and below Vcc-min operation, operating conditions and fault-model.

Performance analysis is useful to assess the performance of a processor that operates with disabled faulty blocks that have not been replaced with spares. Sohi [32] studied the impact on miss-rate of a cache organization with randomly disabled portions, such as ways and sets. The aim of [32] is to improve yield without noticeable miss-rate increase. Related research performed by Pour and Hill [25] also studied the impact of manufacturing faults on cache miss-rate. The work by [25] quantified analytically the expected miss-rate implications of different fault scenarios using a single run through an address trace. This approach aims to eliminate the need for long simulations but uses large number of random-fault maps. They estimate the expected miss ratio for a fixed number of faults by generating all the possible distributions of these faults over the cache sets. In [27], a methodology is proposed to estimate the expected miss rate and its expected distribution by using pfail without the need to generate fault

maps. Our work shares similarities to [27] but, as highlighted in the introduction, also some key differences.

A number of recent studies consider the performance implications with disabled cache blocks assuming random fault-maps [21, 4, 1]. Our model does not rely on fault-maps but rather on probability analysis.

An earlier work is concerned with the testing and validation of mechanisms aiming to enhance performance [7]. This underlines the importance of mitigating faults in prediction arrays. However, this work does not evaluate the performance implications of faults. Bower *et al.* [8] investigate the performance effects of up to 8 permanently faulty entries in a branch history table. Their conclusion is that it is not worthwhile to protect this table against hard faults as performance degradation is negligible. Also, Makris *et al.* [3] evaluated the effect of a single fault in the most frequently accessed entry of a conditional branch predictor. Our paper considers the performance impact more rigorously using an analytical approach.

6. Conclusions and Future work

This work proposes a model for predicting PVF: the expected performance degradation in the presence of permanently faulty cells in architectural and non-architectural arrays. The model for a given program execution, micro-architectural configuration, and cell pfail, provides rapidly the PVF. PVF can be used by designers/researchers to evaluate and compare vulnerability of different structures and perform reliability driven trade-offs.

The model assumptions are validated and shown to be correct by comparing the predicted values of our model against actual values obtained by simulations with many fault-maps.

Predictions using the model reveal that the expected PVF for predictors is small even with high pfail. However, the PVF distribution reveals cases where processors in a given population will experience a significant performance degradation. Consequently, future processor reliability strategies may need to consider predictors. For the last-level cache, the PVF becomes increasingly prominent with technology scaling. This suggests that last level cache PVF mitigation techniques will become essential for future processors.

Design trade-off analysis using the model reveals that choosing appropriately the cell size in an array can help maintain PVF low with a small impact on die yield.

Future work, will extend and validate the proposed PVF model for other non-architectural arrays and architectural arrays as well as to multi-cores. We also plan to investigate low-cost detection and repair schemes for both architectural and non-architectural arrays to ensure a given PVF bound for all processors in a population. Developing an integrated AVF-PVF analysis will help measure the interactions between the two types of vulnerability when making design decisions.

Acknowledgements

The research leading to this paper is supported by the European Commission FP7 project "Energy-conscious 3D Server-on-Chip for Green Cloud Services (Project No:247779 "EuroCloud")". Damien Hardy was also supported by a mobility grant by HiPEAC (FP7 Network of Excellence). We like to thank the reviewers for their critique and comments that helped improve significantly the quality of this manuscript. The last author likes to acknowledge Veerle Desmet, Babak Falsafi, Emre Özer, Ronny Ronen, and André Seznec for their encouragement to pursue this line of work.

References

[1] J. Abella, J. Carretero, P. Chaparro, X. Vera, and A. González, "Low vccmin fault-tolerant cache with highly predictable performance," in *MICRO42*, 2009, pp. 111–121.

[2] A. Agarwal, B. C. Paul, H. Mahmoodi, A. Datta, and K. Roy, "A process-tolerant cache architecture for improved yield in nanoscale technologies," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 13, no. 1, pp. 27–38, Jan. 2005.

[3] S. Almukhaizim, T. Verdel, and Y. Makris, "Cost-effective graceful degradation in speculative processor subsystems: The branch prediction case," *Computer Design*, p. 194, 2003.

[4] A. Ansari, S. Gupta, S. Feng, and S. Mahlke, "Zerehcache: armoring cache architectures in high defect density technologies," in *MICRO42*, 2009, pp. 100–110.

[5] M. D. Beaudry, "Performance-related reliability measures for computing systems," *IEEE Trans. Comput.*, vol. 27, no. 6, pp. 540–547, Jun. 1978.

[6] S. Borkar, T. Karnik, S. Narendra, J. Tschanz, A. Keshavarzi, and V. De, "Parameter variations and impact on circuits and microarchitecture," in *DAC40*, Jun. 2003, pp. 338–342.

[7] P. Bose, "Testing for function and performance: Towards anintegrated processor validation methodology," *J. Electron. Test.*, vol. 16, no. 1-2, pp. 29–48, 2000.

[8] F. A. Bower, P. G. Shealy, S. Ozev, and D. J. Sorin, "Tolerating hard faults in microprocessor array structures." in *DSN34*, Jun. 2004, pp. 51–60.

[9] K. Bowman, J. Tschanz, C. Wilkerson, S.-L. Lu, T. Karnik, V. De, and S. Borkar, "Circuit techniques for dynamic variation tolerance," in *DAC46*. New York, NY, USA: ACM, 2009, pp. 4–7.

[10] L. Cheng, P. Gupta, C. J. Spanos, K. Qian, and L. He, "Physically justifiable die-level modeling of spatial variation in view of systematic across wafer variability," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 30, no. 3, pp. 388–401, 2011.

[11] Z. Chishti, A. R. Alameldeen, C. Wilkerson, W. Wu, and S.-L. Lu, "Improving cache lifetime reliability at ultra-low voltages," in *MICRO42*, 2009, pp. 89–99.

[12] R. Desikan, D. Burger, S. Keckler, and T. Austin, "Sim-alpha: a validated execution driven Alpha 21264 simulator," CS Dept., University of Texas at Austin, Tech. Rep. TR-01-23, 2001.

[13] D. Ernst, N. S. Kim, S. Das, S. Pant, R. Rao, T. Pham, C. Ziesler, D. Blaauw, T. Austin, K. Flautner, and T. Mudge, "Razor: A low-power pipeline based on circuit-level timing speculation," in *Proceedings of the 36th International Symposium on Microarchitecture*, Dec. 2003, pp. 7–18.

[14] S. Eyerman, K. Hoste, and L. Eeckhout, "Mechanistic-empirical processor performance modeling for constructing cpi stacks on real hardware," in *Performance Analysis of Systems and Software (ISPASS), 2011 IEEE International Symposium on*, april 2011, pp. 216 –226.

[15] S. Eyerman, L. Eeckhout, and J. E. Smith, "Characterizing the branch misprediction penalty," in *Proceedings of the 2006 IEEE International Symposium on Performance Analysis of Systems and Software*, Mar. 2006.

[16] D. Fass, "Approximation of discrete multivariate probability distributions: Recursive and hierarchical approaches," 2005.

[17] G. Hamerly, E. Perelman, J. Lau, and B. Calder, "Simpoint 3.0: Faster and more flexible program analysis," in *Journal of Instruction Level Parallelism*, 2005.

[18] E. Hao, P.-Y. Chang, and Y. N. Patt, "The effect of speculatively updating branch history on branch prediction accuracy, revisited," in *MICRO27*, Nov. 1994, pp. 228–232.

[19] E. Krimer, P. Chiang, and M. Erez, "Lane decoupling for improving the timing-error resiliency of wide-simd architectures," in *ISCA*, 2012, pp. 237–248.

[20] N. Ladas, Y. Sazeides, and V. Desmet, "Performance Implications of Faults in Prediction Arrays," in *2nd HiPEAC Workshop on Design for Reliability*, 2010.

[21] H. Lee, S. Cho, and B. R. Childers, "Performance of graceful degradation for cache faults," in *IEEE Computer Society Symposium on VLSI*, Mar. 2007, pp. 409–415.

[22] C. McNairy and J. Mayfield, "Montecito error protection and mitigation," in *HPCRI '05: 1st Workshop on High Performance Computing Reliability Issues, in conjunction with HPCA '05*, 2005.

[23] S. S. Mukherjee, C. Weaver, J. S. Emer, S. K. Reinhardt, and T. M. Austin, "A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor," in *MICRO36*, Dec. 2003, pp. 29–42.
[24] S. R. Nassif, N. Mehta, and Y. Cao, "A resilience roadmap," in *DATE*, 2010, pp. 1011–1016.
[25] A. F. Pour and M. D. Hill, "Performance implications of tolerating cache faults," *IEEE Transactions on Computers*, vol. 42, no. 3, pp. 257–267, Mar. 1993.
[26] D. Roberts, N. S. Kim, and T. N. Mudge, "On-chip cache device scaling limits and effective fault repair techniques in future nanoscale technology," *Microprocessors and Microsystems - Embedded Hardware Design*, vol. 32, no. 5-6, pp. 244–253, May 2008.
[27] D. Sánchez, Y. Sazeides, J. L. Aragón, and J. M. Garcia, "An analytical model for the calculation of the expected miss ratio in faulty caches," in *IOLTS*, 2011, pp. 252–257.
[28] K. Sankaranarayanan, S. Velusamy, M. Stan, C. L, and K. Skadron, "A case for thermal-aware floorplanning at the microarchitectural level," *Journal of ILP*, vol. 7, 2005.
[29] D. P. Siewiorek, R. S. Swarz, and A. K. Peters, *Reliable computer systems (3rd ed.): design and evaluation*. Ltd, 1998.
[30] L. J. Simonson and L. He, "Micro-architecture performance estimation by formula." in *SAMOS'05*, 2005, pp. 192–201.
[31] K. Skadron, P. S. Ahuja, M. Martonosi, and D. W. Clark, "Improving prediction for procedure returns with return-address-stack repair mechanisms," in *MICRO31*, Nov. 1998, pp. 259–271.
[32] G. S. Sohi, "Cache memory organization to enhance the yield of high performance VLSI processors," *IEEE Transactions on Computers*, vol. 38, no. 4, pp. 484–492, Apr. 1989.
[33] J. Srinivasan, S. V. Adve, P. Bose, and J. A. Rivers, "Lifetime reliability: Toward an architectural solution," *IEEE Micro*, vol. 25, no. 3, pp. 70–80, May 2005.
[34] Y. Taur, "CMOS design near to the Limit of Scaling," *IBM Journal of Research and Development*, vol. 46, no. 2/3, pp. 213–222, Mar./May 2002.
[35] A. Vassighi, O. Semenov, M. Sachdev, S. Member, A. Keshavarzi, and C. Hawkins, "Cmos ic technology scaling and its impact on burn-in," *IEEE Trans. on Devices and Materials Reliability*, vol. 4, 2004.
[36] C. Wilkerson, H. Gao, A. R. Alameldeen, Z. Chishti, M. Khellah, and S.-L. Lu, "Trading off cache capacity for reliability to enable low voltage operation," in *ISCA35*, Jun. 2008, pp. 203–214.
[37] S.-T. Zhou, S. Katariya, H. Ghasemi, S. Draper, and N. S. Kim, "Minimizing total area of low-voltage sram arrays through joint optimization of cell size, redundancy, and ecc," in *Computer Design (ICCD), 2010 IEEE International Conference on*, oct. 2010, pp. 112 –117.

Appendix

Sketch of proof: correctness of the discrete distribution lower/upper bounds.

Let's consider the min clustering function (similar reasoning can be applied to the max clustering function).

1 Let's assume a discrete distribution P. By applying the clustering function on P, we obtain distribution P'.
By construction of the min clustering function, the following property (pa) is ensured:

$$pa : \forall X, P'(x < X) \geq P(x < X)$$

The correctness is ensured for the first step of the algorithm.

2 Let's assume two distributions P1 and P2 and their corresponding P'_1 and P'_2 distributions resulting from the min clustering function. The next step of the algorithm consist of combining P'_1 with P'_2, noted $P'_{1,2}$ and we want to be sure that the following condition (ca):

$$ca : \forall Z, P'_{1,2}(x < Z) \geq P_{1,2}(x < Z)$$

is always valid to ensure the correctness of this step.
With property *pa*, we have:

$$\forall Z, P'_1(x < Z) \geq P_1(x < Z)$$

$$\forall Z, P'_2(y < Z) \geq P_2(y < Z)$$

Thus,

$$\forall (x+y) < Z,$$
$$P'_1(x < Z-y) \geq P_1(x < Z-y)$$
$$\wedge$$
$$P'_2(y < Z-x) \geq P_2(y < Z-x)$$

Thus,

$$\forall (x+y) < Z,$$
$$P'_1(x < Z-y) * P'_2(y < Z-x) \geq P_1(x < Z-y) * P_2(y < Z-x)$$

And the combination of distributions is a multiplication of probabilities thus, ca is verified.

3 By induction we obtain a safe distribution bound.

■

2012 IEEE/ACM 45th Annual International Symposium on Microarchitecture

NoCAlert: An On-Line and Real-Time Fault Detection Mechanism for Network-on-Chip Architectures

Andreas Prodromou[1], Andreas Panteli[1], Chrysostomos Nicopoulos[1], and Yiannakis Sazeides[2]

{prodromou.andreas, panteli.andreas, nicopoulos}@ucy.ac.cy, yanos@cs.ucy.ac.cy
[1]Department of Electrical and Computer Engineering, University of Cyprus
[2]Department of Computer Science, University of Cyprus

Abstract

*The widespread proliferation of the Chip Multi-Processor (CMP) paradigm has cemented the criticality of the on-chip interconnection fabric. The Network-on-Chip (NoC) is becoming increasingly susceptible to emerging reliability threats. As technology feature sizes diminish into the nanoscale regime, reliability and process variability artifacts within the NoC start to become prominent. The need to detect the occurrence of faults at run-time is steadily becoming imperative. In this work, we propose **NoCAlert**, a comprehensive on-line and real-time fault detection mechanism that demonstrates 0% false negatives within the interconnect, for the fault model and stimulus set used in this study. Based on the concept of invariance checking, NoCAlert employs a group of lightweight micro-checker modules that collectively implement real-time hardware assertions. The checkers operate seamlessly and concurrently with normal NoC operation, thus eliminating the need for periodic, or triggered-based, self-testing. More importantly, 97% of the faults are detected instantaneously. Extensive cycle-accurate simulations in a 64-node CMP demonstrate the efficacy of the proposed technique. Finally, hardware synthesis results using commercial 65 nm technology libraries indicate minimal area and power overhead of 3% and less than 1%, respectively, and negligible impact on the router's critical path.*

1. Introduction

Diminutive technology feature sizes have enabled microprocessors with billions of transistors on a single chip die [1]. This unprecedented abundance of on-chip resources, coupled with thinning Instruction-Level Parallelism (ILP), have urged designers to switch their attention to another computational archetype: the *Chip Multi-Processor* (CMP) [2]. The presence of multiple on-chip processing entities has precipitated a shift from computation-centric to communication-centric micro-architectures. As a result, the on-chip interconnection fabric is fast becoming a mission-critical component. Packet-based Networks-on-Chip (NoC) are widely viewed as the de facto communication medium of future multi-/many-core CPUs, primarily due to their inherent scalability attributes and modular nature [3].

However, the march towards CMPs with tens – or even hundreds – of processing cores has been marred by the emergence of an ominous threat: waning *reliability* [4]. The extreme downscaling trends of CMOS technology have rendered transistors more susceptible to both permanent and transient faults. Moreover, digital circuits are increasingly affected by growing process variability artifacts [5] and accelerated aging effects [6, 5], all of which are consequences of dwindling feature sizes. Just like any on-chip component, the interconnection backbone is also affected by decreasing reliability [7]. In fact, a single fault in the on-chip network may paralyze an otherwise healthy CMP. Faults within the NoC may result in such show-stopping predicaments as network disconnections, network-level deadlocks, protocol-level (cache coherence) deadlocks, lost packets, and severely degraded on-chip communication performance [8].

Architects and designers have proposed a multitude of techniques, mechanisms, and design modifications to increase the fault tolerance and reliability of the NoC. However, the vast majority of the related work found in the literature concentrates on fault *prevention* (improving durability/fault-tolerance, prolonging lifetime, etc.) [9] and/or *recovery* (redundancy, reconfiguration, adaptation, etc.) [10, 11]. The equally important aspect of ***fault detection*** has not been adequately addressed.

Traditionally, fault detection is undertaken by Built-In Self-Test (BIST) mechanisms that predominantly assume a disruption in the system's operation. The BIST process may be executed by the manufacturer prior to shipment, or it may constitute part of system boot-up [12, 13]. Runtime BIST is also possible, but system operation is (partially) halted while the module-under-test is examined [10, 14]. BIST usually entails the use of predefined test vectors, patterns, or routines, which tend to be pure overhead. Regardless, *detecting faults at run-time* is rapidly becoming a necessity, in light of the aforementioned decline in reliability. When BIST or BIST-like methodologies are employed within the context of on-line (run-time) testing, the process is usually triggered periodically [14]. Choosing the length of the period between two consecutive test sessions is certainly non-trivial: if testing is conducted too frequently, the impact on performance will be more pronounced, due to excessive interruptions; if testing is rarely performed, then faults may go unnoticed for a prolonged period of time [15]. Furthermore, periodic testing often implies the use of checkpointing, which adds further overhead (both in terms of performance and hardware/storage/power).

Near-instantaneous fault detection may be achieved in the datapath of the interconnect through the use of *error detecting codes*. Simple parity checks – or more elaborate coding – will detect (and may even correct) errors affecting the *contents* of in-flight packets [16]. While this methodology guarantees protection of the message contents, faults within the *control logic* of the NoC may still wreak havoc with the operation of the entire CMP. Hence, what is needed to guarantee functional correctness within the NoC – and, by extension, within the CMP – is to *protect the NoC's control logic* (assuming that the flit *contents* are protected by error-correcting codes). This thesis statement marks the central theme of our work.

Realizing the significance of accurate and timely run-time detection of faults within the NoC's control logic, we hereby propose **a comprehensive on-line fault detection mechanism**, aptly called ***NoCAlert***, which provides full fault coverage for all on-chip network control logic components and achieves *instantaneous* detection of any erroneous behavior. Depending on the application's criticality, instantaneous detection may be of paramount significance. The NoCAlert mechanism is based on the notion of ***invariance checking***, whereby the system is continuously checked for illegal outputs as a result of upsets (permanent, transient, or intermittent). An *illegal output* is defined here as an operational decision that violates

1072-4451/12 $26.00 © 2012 IEEE
DOI 10.1109/MICRO.2012.15

IEEE computer society

the functional correctness rule(s) of a particular component. The underlying principle of this technique is inspired by prior efforts to protect the microprocessor by using invariances [17]. NoCAlert comprises several checker micro-modules distributed throughout the NoC router, which seamlessly and concurrently monitor all NoC modules for illegal activity. The checkers never interfere with – or interrupt – the operation of the NoC and they provide real-time on-line fault detection. In essence, NoCAlert implements an all-encompassing collection of extremely lightweight ***real-time hardware assertions*** that can detect illegal outputs within the NoC's control logic.

In particular, the main contributions of this work are:

1. The development of **a comprehensive on-line and real-time fault detection mechanism** for the control logic of the NoC of multi-core CMPs. The proposed NoCAlert checker modules operate *seamlessly and concurrently with normal NoC operation*, thus obviating the need for testing epochs and periodic triggering of testing sessions that may interrupt/impede normal system operation.
2. The NoCAlert protective blanket ensures **0% false negatives within the interconnect** for the fault model (single-bit transient) and stimulus set used in this study, with **97% of the faults detected instantaneously** (i.e., in the same cycle as the fault occurrence). This attribute allows for ultra-fast response by a potential fault recovery scheme and/or re-configuration mechanism. NoCAlert is intended to be used in conjunction with fault recovery techniques.
3. We demonstrate that by using checkers that solely detect *illegal* outputs (outputs that cannot be produced by any input) for all NoC control components, we observe 0% false negatives for the entire network using the fault model of this study. This empirical observation leads to an interesting hypothetical corollary about a NoC router's control components: if a unit produces a faulty but *legal* output, which does not lead to subsequent invariance violations, it is always benign as far as the overall NoC operation in concerned.
4. The entire NoCAlert scheme is **extremely lightweight in terms of all salient design metrics**. Hardware synthesis results using commercial 65 nm standard-cell libraries indicate minimal area and power overhead of 3% and less than 1%, respectively. More importantly, the critical path of the router is shown to be negligibly affected (around 1%), rendering the proposed mechanism transparent to normal operation. Our analysis clearly indicates that checkers used to detect only illegal outputs have significantly lower hardware cost, as compared to the cost of the unit they check; i.e., the complexity of determining whether an output is illegal – given an input – is much simpler than producing the output.
5. The NoCAlert framework is evaluated by injecting faults **in all possible locations** (according to the employed fault model) within the NoC of a 64-node CMP arranged in an 8×8 mesh. Extensive simulations were run in a cycle-accurate NoC evaluation framework. The results and ensuing analysis corroborate the efficacy of the NoCAlert mechanism.
6. Through a detailed experimental comparison, NoCAlert is shown to outperform ForEVeR [15], a recently proposed state-of-the-art fault detection and recovery framework. NoCAlert provides more than $100\times$ reduction in fault detection latency, with no loss in detection accuracy, and without the need to rely on a secondary, fault-free checker network for detection purposes.

To the best of our knowledge, this work constitutes the first attempt to utilize real-time hardware-based assertion checkers to ensure 0% false negatives within the on-chip interconnection network of CMPs.

The rest of the paper is organized as follows: Section 2 discusses related prior work in fault-tolerant NoCs. Section 3 introduces the idea of invariance checking within the on-chip network, while Section 4 delves into the description, implementation, and analysis of the proposed NoCAlert mechanism. Section 5 presents the employed evaluation framework, the various experiments, and accompanying analysis. Finally, Section 6 concludes the paper.

2. Related Work

In general, research in the field of fault tolerance revolves around two fundamental axes: (1) Fault *Detection*, and (2) Fault *Recovery/Protection/Isolation*. While the focus is often slanted more toward the latter, both axes are essential in delivering a robust system. Research in the field of *NoC* reliability naturally falls into two main categories: (a) Inter-router faults (i.e., faults within the links interconnecting the various switches), and (b) Intra-router faults (i.e., faults within the switches themselves). The following sub-sections will concentrate on these two categories.

2.1. Inter-Router Faults

Disabled inter-router links in the network reduce connectivity. Reduced connectivity may, in turn, lead to network deadlocks and – depending on the routing algorithm used – may lead to halted network operation. Broken network links imply reduced path diversity, creation of hotspots, and network delay due to back-pressure effects. Default Backup Paths (DBP) [18] were proposed as a means to maintain connectivity in the presence of faults. In [11], all the physical links are doubled in order to enhance NoC connectivity. Naturally, the presence of fully disabled links predominantly affects the routing algorithm within the network routers.

The assumption of *fully* disabling a parallel multi-bit inter-router link is overly pessimistic. In reality, each parallel link (ranging from 32 up to 256 wires) is individually driven. Whenever a wire fails, the rest can still function properly. Hence, in a real-world scenario, a fault within the links will give rise to *partially faulty links*. It is this realization that has led researchers to look into Error-Correcting Codes (ECC) [19, 20], utilizing redundant wires/bits. For online detection and diagnosis purposes, these codes are very effective. Retransmission mechanisms are typically required to co-operate with ECC schemes [19]. Researchers have devised methodologies to transfer flits through these partially faulty links through shifting and multi-cycle transmissions [21], and by using spare wires [20].

2.2. Intra-Router Faults

This sub-section presents prior work regarding fault-tolerant routing and architectural redundancy schemes.

Fault-Tolerant Routing in NoCs: Most NoC fault-tolerant routing algorithms are inspired from seminal work conducted in the domain of large-scale, multi-computer interconnection networks [22, 23]. NoCs are characterized by severely limited on-chip resources, scarce energy budgets, and the imperative need for ultra-high performance. As such, the fault-tolerant routing algorithms proposed for NoCs must account for these salient attributes. Universal Logic-Based Distributed Routing (uLBDR) [24] aims to eliminate fault-susceptible routing tables. Stochastic routing algorithms [25] have been employed to bypass faulty links in the network. Dynamic reconfigurable routing algorithms [26] determine forbidden turns at

run-time to avoid deadlocks while bypassing faulty components. Deflection routing [27, 28] is another technique that favors routing resilience, and it has been employed as a fault-tolerant routing mechanism [28]. Distributed [29] and multi-path [30] routing strategies aim to evenly spread network traffic over a faulty network topology without deadlocks. Finally, the concept of exploration/scouting packets [31, 32] has also been used to identify faulty nodes ahead of regular data packets.

Architectural Techniques to Tackle Datapath and Control Logic Faults: Besides the multitude of routing algorithms designed to provide uninterrupted network functionality in the presence of faults, a lot of research has addressed fault-tolerance in the critical components comprising the datapath and control logic of NoC routers.

The Row-Column (RoCo) Decoupled router [33] provides extensive fault tolerance and graceful degradation by decomposing the router into two independent modules and by employing resource sharing. Bulletproof [9] proposes various online repair and recovery capabilities and investigates protection at various levels, ranging from system-level to arbitrary partitions of the design.

Fault-tolerant techniques provide effective *recovery* mechanisms that ensure correct functionality in the presence of faults. Of course, the basic assumption is that the fault must first be *detected*. While most of the techniques considered in this Section so far *assume* the presence of fault detection capability and concentrate on the recovery aspects, others have also tackled the non-trivial facet of detecting the faults in the first place.

The proposed mechanism in [13] broadcasts test vectors within the NoC *during boot-up only* and detects faults by examining the responses of the various router components. To accommodate run-time occurrence of faults, the work in [12] is also capable of generating *on-line* test vectors that are broadcast in the network. Test vector results are then evaluated by neighboring routers. However, in this scheme, the entire network's operation is halted for testing purposes. In order to mitigate the performance degradation caused by testing interruptions, the token-based mechanism in [14] interrupts only a small portion of the network at any given time. The Allocation Comparator of [19] performs on-line, real-time diagnosis by observing the occurrence of some invalid operations within the router arbiters as a result of transient faults. The Vicis router [10] employs ECC codes to detect *some* faults. Subsequently, specialized BIST testers located in each router are utilized for more extensive testing and fault localization [10]. Finally, the appropriate reconfiguration mechanism is triggered to combat the detected fault. Error-correcting codes have also been used in conjunction with a packet/flit counting technique to detect and diagnose permanent faults in the network [34].

The ***ForEVeR* framework** [15] was recently proposed, which complements the use of formal methods and runtime verification to ensure functional correctness in NoCs. While ForEVeR's goal is to protect against escaped design-time verification errors with a run-time technique, the scheme may also be used to provide robustness against run-time faults. Fault detection is achieved with the help of (a) an *additional* lightweight checker network that is *assumed to be 100% reliable*, (b) the Allocation Comparator from [19], and (c) an end-to-end checker. The checker network is used to alert destination nodes ahead of time about incoming flits. The destination node increases a flit counter upon a notification reception, and decreases the same counter upon flit reception. Time is separated in so called *epochs*, and at the end of each epoch the counter must have reached the value of zero at least once within the epoch interval. If not, a recovery mechanism is triggered, which delivers the in-flight data to the intended destination via the checker network. The use of timing intervals implies the non-trivial task of finding the optimal epoch duration to minimize false positives. In fact, if the epoch duration is not carefully chosen, the mechanism may give rise to false positives even in a fault-free environment. Moreover, the epoch duration is sensitive to the traffic injection rate, which hinders widespread applicability. More importantly, the use of an end-to-end, epoch-based scheme, such as ForEVeR, results in significantly delayed fault detection. Particular to ForEVeR, fault detection relies on ahead-of-time notifications sent through the checker network; hence, a *run-time* fault in the checker network would incapacitate fault detection. Finally, any faults that cause degradation in performance, but do not cause a functional error at the output (end-to-end delivery) will never be detected (i.e., only faults that cause functional errors are detected). A detailed quantitative comparison between NoCAlert and ForEVeR [15] will be presented in Section 5.

In summary, recovery and reconfiguration schemes rely on efficient, accurate, and quick-responding fault detection mechanisms. In the absence of such mechanisms, the efficacy of recovery is severely compromised. Inaccurate detection mechanisms can cause undue network/system performance degradation, while delayed detection will necessitate the presence and invocation of checkpointing mechanisms, which inevitably incur both hardware and performance overhead.

The NoCAlert mechanism proposed in this work ensures ***on-line and real-time fault detection*** within the NoC, and it guarantees ***0% false negatives*** under the employed fault model. Most importantly, the technique works concurrently with normal network operation (i.e., no testing interruptions) and is shown to be extremely lightweight. Moreover, **NoCAlert may be used to complement any other fault recovery scheme**, such as ForEVeR [15]. The recovery mechanism – aided by NoCAlert's instantaneous fault detection – may react much more rapidly (if deemed necessary), thus minimizing the effect on system-level performance.

3. Invariance Checking within the NoC

The NoCAlert mechanism is based on the concept of *invariance checking*. When checking for invariances, the system is continuously examined for illegal outputs as a result of some kind of perturbation (fault). As previously mentioned, the term *illegal output* is defined here as an operational output that is impossible to occur, based on the set of functional correctness rules of a given component. Thus, the term "invariance" describes a condition that cannot – by definition – vary. Consequently, an invariance violation is the breaking of fundamental rules within the context of a system component. Invariance is a general term that applies to every system governed by some rules within a specific context. Considering an adder circuit as an example, one derived invariance would be that the sum of two even numbers must always be even as well.

A well-known implementation of invariance checking is the use of *assertions* in software development [35]. Software assertions ensure that a forbidden state cannot be reached; if it is reached, a notification is issued.

In this work, we adopt the notion of invariance checking and apply it to all the modules of a NoC router's control logic to *detect* abnormalities in the network resulting from either transient, or permanent, faults. The assertions are implemented in *hardware* so as to provide near-instantaneous detection of anomalies.

The salient characteristic of invariance checking, in general, is the fact that only *functionally illegal* outputs are flagged as violations. In other words, a fault that causes the generation of an erroneous,

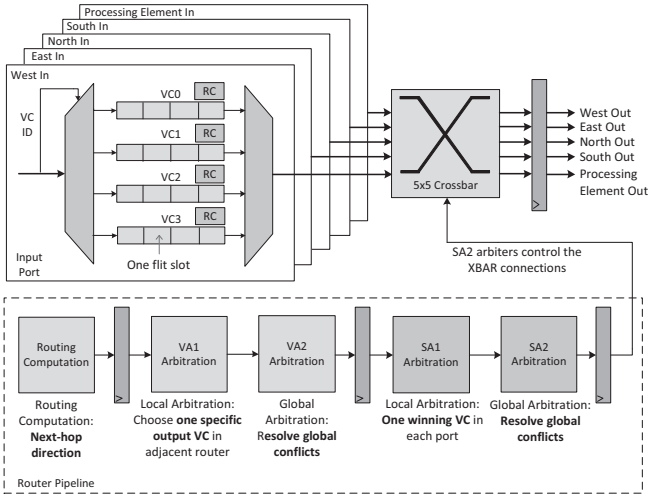


Figure 1: Overview of the router pipeline. The baseline router has five pipeline stages; namely, Routing Computation (RC), Virtual channel Allocation (VA), Switch Arbitration (SA), Crossbar (XBAR) traversal, and Link Traversal (LT).

yet functionally legal, output will *not* be identified as a breach of correctness.

Knowing this innate limitation of invariance checking, what we aim to explore in this work – among others – is how often, and under what conditions, such non-invariant faults could potentially lead to compromised network-level correctness. We will demonstrate empirically that non-invariant faults within the NoC routers, which do not cause any subsequent invariance violations (i.e., they are not caught by subsequent checkers), always prove to be innocuous at the system level, i.e., they do not cause network-level malfunction.

Before we proceed with the identification of invariant conditions (invariances) within the NoC, we first present – without loss of generality – a typical router micro-architecture [36], which forms the foundation of most router implementations discussed in the literature. It is important to note that this architecture is general enough to allow the proposed NoCAlert mechanism to be applicable to *any* router implementation. Later on in the paper, we will briefly show how NoCAlert can also be fitted to different router designs.

3.1. A Generic NoC Router Micro-architecture

Figure 1 presents a high-level, abstracted view of the baseline router micro-architecture assumed in this work. This generic input-buffered router design consists of five input/output ports. Four of them are used to communicate with the adjacent routers in the network (in the four cardinal directions of a 2D mesh) and the fifth port is used to communicate with the local processing element. Each input port has a number of Virtual Channels (VC) that support (potentially) the routing algorithm (e.g., adaptive) and, more importantly, the cache coherent protocol employed within the CMP. VCs are used to avoid protocol-level deadlocks in the network, as well as to enhance bandwidth utilization at the network level. A central crossbar (XBAR) facilitates the interconnection between the input and output ports, as shown in Figure 1. The main modules of the router's control logic are the Routing Computation (RC) unit, the Virtual channel Allocation (VA) unit, and the Switch Arbitration (SA) unit. The RC unit is responsible to compute the output direction that a packet must follow to get to the next hop, based on the destination information found in the header flit of each packet. The VA unit allocates a downstream VC to each packet. This is the VC that the packet will use in the adjacent router. Finally, the SA unit decides which flits traverse the crossbar in each cycle. The baseline router is assumed to be wormhole-switched (the predominant choice in *on-chip* networks) and to use credit-based flow control.

The employed router has a five-stage pipeline, with each stage corresponding to one of the major functional units within the router: RC, VA, SA, XBAR traversal, and Link Traversal (LT), as illustrated in Figure 1. The first two stages are executed only for the *header* flit of each packet (in order to set up the wormhole), while the remaining stages are executed for all flits. As can be seen in Figure 1, the VA and SA stages are further separated into *local* (intra-port) and *global* (inter-port) sub-stages. Local stages perform arbitration within a specific port, while the global stages resolve conflicts between the various ports. The data-path of the router comprises the input buffers and the XBAR switch. Each input port employs an input de-multiplexer and an output multiplexer to accommodate the sharing of one physical channel by multiple VCs. This organization implies that only one flit can arrive to, or leave from, an input port in each cycle. Furthermore, VCs may be atomic or non-atomic. Atomic VCs can only store the flits of a single packet at any given time. In other words, flits from two different packets cannot co-exist in the same VC.

3.2. Examples of On-Chip Network Invariances

This sub-section presents three representative examples of invariances found within the NoC. To aid understanding, the examples are depicted in Figure 2.

Assuming the 4×4 mesh network in Figure 2(a), let us identify one important invariance pertaining to the widely used XY routing algorithm. Routing algorithms, in general, forbid some turns to avoid deadlocks and/or livelocks in the network. The XY routing algorithm, in particular, first routes a packet along the X dimension until the intended destination's X-coordinate has been reached, and then along the Y dimension until the destination node has been reached. Suppose the origin of the Cartesian system is the bottom left router, and assume that a packet is injected in router (1,1) with destination (1,3). Upon reaching router (1,2), a fault in the RC unit of said router forwards the packet to the East output port, toward router (2,2). This action constitutes an *invariance violation*, since a packet arriving from the Y dimension (North or South input ports) may not make a turn to the X dimension (East or West output ports) under XY routing. Such an invariance violation, if caught, indicates a malfunctioning RC unit.

As described in Section 3.1, there are five pipeline stages in the baseline router. Under normal operation, those pipeline stages should be executed in the correct order. To maintain the pipeline functionality, each VC keeps its own functional state. Figure 2(b) shows an example of a VC's status table. In this example, a header flit is present at the head of the queue and is waiting for the VA stage (VC allocation) to be executed. Since the "VA done" field in the status table is set to 0, an output VC has not yet been allocated to the specific packet. A malfunctioning SA arbiter, however, sends an active grant success signal to the VC, thus violating the correct pipeline order (SA success before VA is complete). This invariance violation identifies erroneous behavior within the SA module.

Finally, Figure 2(c) illustrates a router's input port with four VCs. Suppose that the flit at the head of the VC0 queue is ready to proceed to the XBAR stage, as indicated by the active "Read Signal" at VC0. However, a fault leads to the simultaneous assertion of the VC1 read signal in the same clock cycle. As can be seen in the figure, only one flit from each input port may depart in a single clock cycle, due to the

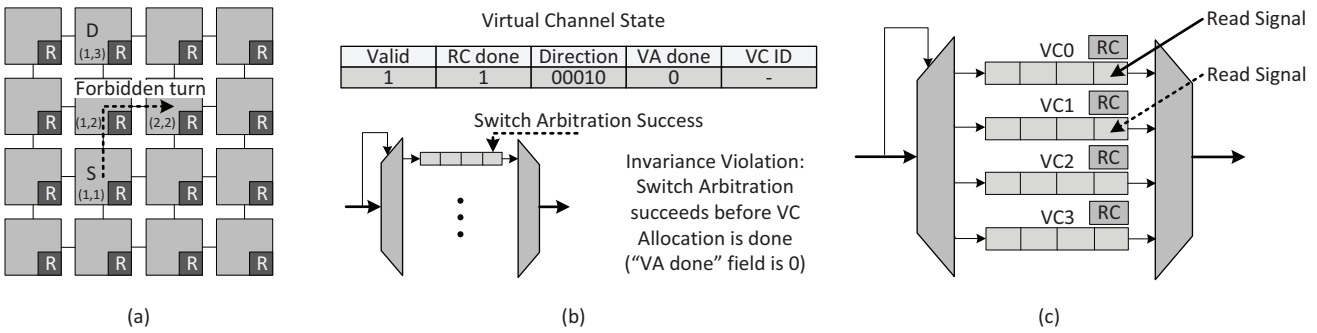


Figure 2: Examples of NoC invariances. The first example (a) illustrates the case whereby a malfunctioning XY routing computation unit attempts to route a packet in a forbidden direction (S: Source node; D: Destination node). The second example (b) demonstrates an invariance violation that occurs upon receiving an SA success signal before the VA stage is complete. Finally, the third example (c) illustrates the erroneous case of having more than one active read signals in the same router port in the same cycle.

presence of the multiplexer. The presence of two concurrently active read signals in the same input port indicates an invariance violation.

3.3. Identifying Invariances within the NoC Router

In order to identify a component's invariances, one must carefully examine the operation of said component within the context of its governing functional rules. In general, it is not always obvious that a range of values will never appear under normal operation. In the case of NoC routers, invariance identification is possible, because of the inherent modularity of the constituent modules. Each router module is usually responsible for a very specific task. For example, the RC unit is only tasked with the determination of the output direction[1] of a particular incoming packet. An arbiter grants one out of a number of requests, and the crossbar module is responsible for interconnecting input and output ports.

In this work, invariances were constructed by observing the operation and behavior of each functional module. Specifically, the list of invariances is constructed using a bottom-up approach. The NoC router design is implemented in a modular and hierarchical manner; e.g., FIFO buffers $\rightarrow$ Arbiters $\rightarrow$ Input Port $\rightarrow$ Crossbar Switch $\rightarrow$ ... $\rightarrow$ Entire Router. The *algorithm* responsible for the *functional* operation of each module (e.g., the routing algorithm) is then exhaustively inspected to identify all *functional rules*. This analysis allows us to identify the functional rules of all components (which are not prohibitively many in a NoC), and, by extension, all *functionally illegal* outputs. Hence, the assertions are derived from each functional rule in the algorithm that describes the operation(s) of each module. This methodology is repeated for higher levels in the design hierarchy until the whole router is covered. Finally, end-to-end invariances at the network level (considered to be the highest level in the hierarchy) are also identified. To be able to follow the same procedure, designers must keep the design *modular*, so as to enable the decomposition of each module's operation.

By viewing the design *hierarchically* (not just locally), invariances manifesting in a coupled/combined manner are also covered. By gradually moving up the hierarchy (from individual modules to groups of modules), new assertions are derived from functional rules governing the higher levels of the hierarchy (e.g., rules that apply to the input-port-level).

The completeness of invariances depends on the completeness of the functional analysis of the design itself: a NoC consists of a set of functional rules. These rules are defined by the modules (and their interactions) within the routers. If the invariance checkers cover all functional rules, NoCAlert will detect *any illegal* behavior.

All identified invariances are listed in Table 1, which will be described in more detail in Section 4.

It should be stressed at this point that *our focus is on the control logic* of the on-chip network. The data-path is usually protected by the well-established and ubiquitous practice of augmenting the flit payload with *Error Detecting Codes (EDC)* [19, 10]. In fact, more elaborate codes can also be employed that can even correct some errors (bit flips) within the flit *contents*. Hence, our only assumption in this paper is that the *contents* of the flits/packets are protected by a simple error detecting code, which will alert the system of any undesired alteration in the message contents (the code usually provides coverage for both the payload and the network overhead bits). In its simplest guise, the EDC could be a single-bit parity check.

Protecting the message *contents*, however, is not enough to guarantee the functional correctness of the NoC. Erroneous behavior within the control logic can lead to catastrophic results, since the control logic is the coordinator of the network's operation. Control logic upsets may lead to flit drop, packet loss, network/protocol deadlocks, network livelocks, packet mixing (which is not detected by the EDC, since it involves the flits of different packets erroneously following the same wormhole), and degraded performance (at best).

It is precisely for this reason that we advocate the incorporation of the NoCAlert mechanism into the on-chip network. NoCAlert provides on-line and real-time detection of faults within the control logic of the entire NoC. The flow of packets – from injection to ejection – is seamlessly monitored for any digression from normalcy. The NoCAlert scheme acts as a guardian of NoC operation and casts a protective blanket over the entire interconnect.

4. NoCAlert: An On-Line, Hardware-Assertion-Based Fault Detection Mechanism for NoCs

The proposed NoCAlert utilizes the principle of invariance checking and implements it in the form of ***real-time hardware-based assertions***. The key idea is to have a simple *hardware* checker module for every NoC component. This checker module will take as inputs the inputs and outputs of the protected component and it will check whether any functional rule is broken during the component's operation. Checkers mostly perform simple comparisons and, hence, they comprise simple combinational circuits with very low complexity.

As previously mentioned, in order to deploy and integrate the checkers in the router design, all the invalid outputs of all the main modules of the router had to be identified through a comprehensive analysis of each module's operation. This detailed exploration of

[1] Some routing algorithms also provide the output VC, in addition to the output direction [36].

	Routing Computation (RC) Unit	
1	Illegal turn	Routing algorithms forbid some turns to prevent deadlocks in the network.
2	Invalid RC output direction	There are some invalid RC output directions. For example, if the router has five ports (numbered 1 to 5), value 6 is invalid [19].
3	Non-minimal routing (if required)	The RC unit's output direction must take the flit one step closer to its destination.
	Arbiter Modules (VA and SA Stages)	
4	Grant w/o request	It is not possible for a flit to win a grant without making a request.
5	Grant to nobody	The arbiter must always provide a winner when there is at least one client request.
6	1-hot grant vector	The arbiter's output vector must have at most one bit set to logic high.
7	Grant to occupied or full VC	A grant to an occupied or full VC (based on the neighbor's credits) is forbidden.
8	One-to-One VC assignment	An input VC must not be assigned to multiple output VCs.
9	One-to-One port assignment	An input port must not gain simultaneous access to multiple output ports.
10	VA agrees with RC	The output VC assigned by the VA unit must be in agreement with the result of the RC stage, as originally proposed in [19].
11	SA agrees with RC	The SA result must be in agreement with the result of the RC stage, as originally proposed in [19].
12	Intra-VA stage order	If a VC wins the VA2 arbitration stage, it must have also won the VA1 stage.
13	Intra-SA stage order	If a VC wins the SA2 arbitration stage, it must have also won the SA1 stage.
	Crossbar (XBAR)	
14	1-hot column control vector	At most one connection must be active in each column of a matrix-style XBAR in each clock cycle (to avoid flit mixing).
15	1-hot row control vector	At most one connection must be active in each row of a matrix-style XBAR in each clock cycle (to avoid unwanted multicasting).
16	# of Incoming flits equals # of Outgoing flits	During each clock cycle, the number of flits exiting the XBAR must be equal to the number of flits entering the XBAR.
	***Buffer State* (Note: Each VC buffer maintains its own state)**	
17	Consistent VC buffer state	The NoC router pipeline stages must be executed in the correct order.
18	Only header flits in free VC buffers	A VC buffer is free when it is not allocated to an in-flight packet. During this state, only a header flit may enter the buffers (i.e., a new packet creating a wormhole).
19	Invalid output VC value	At the end of the VA stage, the computed output VC of the packet is saved in order to extend and maintain the wormhole. The output VC value cannot be out of range [19].
20	Complete RC stage on a non-header flit	Routing computation is performed only on header flits. Thus, to make a transition from the RC to the VA stage, a header flit must be present at the head of the buffer.
21	Complete RC stage on an empty VC	A transition from the RC to the VA stage is forbidden if the buffer of the respective VC is empty.
22	Complete VA stage on a non-header flit	Virtual channel allocation is performed only on header flits. Thus, to make a transition from the VA to the SA stage, a header flit must be present at the head of the buffer.
23	Complete VA stage on an empty VC	A transition from the VA to the SA stage is forbidden if the buffer of the respective VC is empty.
24	Read from an empty buffer	A "read" signal cannot be issued to an empty VC buffer.
25	Write to a full buffer	A "write" signal cannot be issued to a full VC buffer.
26	Buffer atomicity violation (if required)	If the buffers are atomic, only flits from a single packet may reside in the buffer at any given time. Thus, a header flit cannot arrive at a non-free VC buffer.
27	Packet mixing in non-atomic buffer	If the buffers are non-atomic, a tail flit may only be followed by a header flit.
28	Packet flit-count violation	Typically, packets belonging to the same message class have the same length, i.e., the same number of flits. Thus, the number of a packet's flits arriving at a VC belonging to a specific message class must always be the same (equal to a pre-defined constant) [34].
	Port-Level Invariances	
29	Concurrent read from multiple VCs	Only one flit may leave a single input port in each clock cycle (due to multiplexer).
30	Concurrent write to multiple VCs	Only one flit may arrive at a single input port in each clock cycle (due to de-multiplexer).
31	Concurrent RC stage completion of multiple VCs	Since only one flit can arrive at an input port in a single clock cycle, only one VC may complete its RC stage in a single clock cycle in each input port [assuming that (a) atomic buffers are used, and (b) all VCs in a single port use the same routing algorithm].
	Network-Level Invariance	
32	End-to-End delivery violation	The destination address of all packets ejected from a node should equal that node's address.

Table 1: Complete list of the invariances associated with the baseline NoC router design of Figure 1. Note that Invariance 5 (shaded in grey) is innocuous if the fault causing it is *transient/intermittent* (leading only to momentary performance degradation analogous to a NOP instruction in a microprocessor), while it may prove catastrophic if the fault causing it is *permanent* (packets stuck in NoC buffers).

the router's micro-architecture identified a total of **32 invariances**, which are listed in Table 1. The invariances are categorized based on the router module they are associated with: the Routing Computation unit, Arbiters, Crossbar, VC State, Port-Level, and End-to-End.

This list of invariances *completely* characterizes the operational behavior of the router: *any forbidden behavior* (as dictated by the functional rules that govern the router's operation) will be captured by *at least one* of these 32 assertion checkers.

4.1. Ensuring Network Correctness Using Invariances

As NoCs are increasingly becoming more complex, the task of ensuring their functional correctness as a whole is becoming more daunting. Prior research [37, 15] has identified four main conditions that *ensure* functional correctness within the network: (1) No packets are dropped, (2) Delivery time is bounded, (3) No data corruption occurs, and (4) No new packet is generated within the network. If satisfied, these four conditions guarantee functional correctness [37, 15].

Following this guideline, the 32 invariances of Table 1 are categorized according to the aforementioned four general requirements, as illustrated in Figure 3. Each number in the diagram refers to the corresponding entry of Table 1. Even though the original categorization of [37, 15] was made at the *packet* level, we choose to operate at the *flit* level, since the smallest unit of flow control is the flit. By

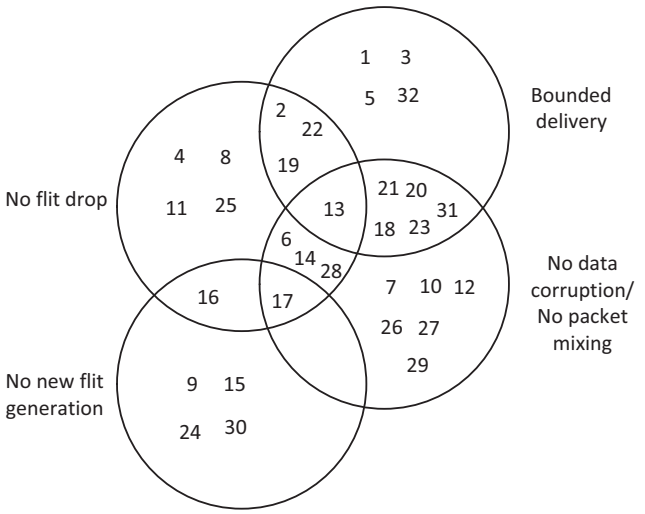


Figure 3: Using invariances to ensure functional correctness. The 32 invariances of Table 1 are categorized based on the 4 fundamental conditions that ensure functional correctness within the NoC [37, 15].

doing this transformation, we actually make the four requirements even stronger, because flits are sub-units of packets. For example, if an extra flit is generated in the network and becomes part of an existing packet, the flit-level rule will correctly identify this as an error, whereas the packet-level rule would not capture this anomaly. Note that operating at the flit level adds the additional requirement that *intra-packet* flit ordering is maintained by the network (a typical assumption in NoCs). Upsets causing such flit ordering violations also violate some of the fundamental invariances monitored by NoCAlert; thus, the proposed mechanism also safeguards against intra-packet flit order changes.

Bounded delivery implies the delivery of all flits to their intended destination within a finite amount of clock cycles. This rule specifies that no deadlock or livelock should occur in the network. *No flit drop* specifies that no flit should be lost during its traversal through the network. *No new flit generation* specifies that no new flits should be spontaneously generated within the NoC. Abnormal flit duplication is also included in this requirement. Finally, *no data corruption/packet mixing* specifies that there should be no collision of flits, and that no flit belonging to a packet should enter the wormhole of another packet (packet mixing). Even though the message contents are assumed to be protected by error-detecting codes, data corruption could still occur by packet mixing, which would escape the per-flit error-detecting codes.

Due to lack of space, only two of the 32 invariances of Table 1 will be described in detail. Specifically, two invariances will be analyzed, which can cause several types of errors. In particular, invariances 13 and 17 sit at the intersection of multiple categories in Figure 3 and may breach *three* out of the four functional correctness requirements.

As described in Section 3.1, the Switch Arbitration stage is further separated into the SA1 (local) and SA2 (global) arbitration stages. Invariance 13 (see Table 1) specifies that if a VC wins in the SA2 arbitration, it must also have won its SA1 arbitration. If a VC wins the SA2 stage without winning SA1, there is a possibility of being forwarded to a full VC in the adjacent router (since credits are evaluated in SA1), and, therefore, it will be dropped (*No flit drop* violation). Additionally, since the SA2 stage drives the crossbar switch,

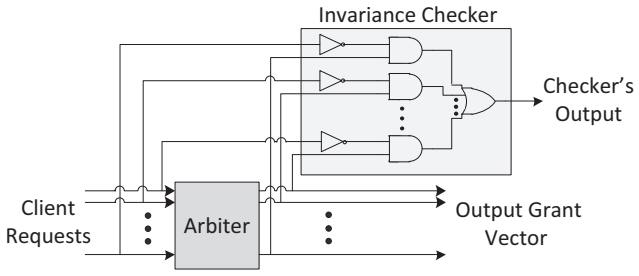


Figure 4: An example NoCAlert checker circuit. This checker constantly monitors an arbiter module at run-time to detect whether a grant signal has been issued at the arbiter's output without any requests at the arbiter's inputs. Note that the figure is not drawn to scale; the checker module is exaggerated for clarity. In reality, the invariance checkers are significantly smaller than the units they check.

a flit might be sent in a different direction than the one calculated by the RC unit. If the flit happens to be sent to an idle VC, this may lead to a deadlock due to "breaking up" of the packet (malfunctioning wormhole). Thus, a *Bounded delivery* rule violation will occur. Finally, if the flit is sent to an occupied VC, the *No packet mixing* rule will be breached.

Invariance 17 states that pipeline stages must be executed in the correct order. Suppose that the VA stage is executed before the RC stage. In this case, the flits of the packet might be forwarded to an occupied VC in the adjacent router (*No packet mixing* violation). Now suppose that the SA stage is executed before the VA stage. The flit will be forwarded to the adjacent router without correct VC ID information. Thus, the flit will be written to an arbitrary VC. If that VC is full, the flit will be dropped (*No flit drop* violation). Finally, suppose that the SA stage is executed before the RC stage, i.e., on an empty VC buffer. This will cause a flit to be forwarded to an adjacent router, but garbage information will be sent (since buffers employ pointers to maintain FIFO order, an "empty" buffer slot is not blank). Therefore, a new flit may be generated (*No new flit generation* violation).

4.2. Hardware Complexity of the NoCAlert Checkers

The NoCAlert fault detection mechanism consists of an array of distributed hardware checkers, which constantly and seamlessly monitor the modules comprising the control logic of the router. Each checker is a simple combinational circuit performing a specific check, according to the rules of the module being monitored.

An example checker circuit is shown in Figure 4. This checker is responsible to monitor an arbiter module and detect whether there is an active grant signal without any requests at the arbiter's inputs. As can be seen from the figure, only two logic gates are needed for each input/output of the arbiter, as well as an OR gate to combine all individual checks. Furthermore, the checker size grows *linearly* with the number of arbiter inputs, whereas the arbiter size grows in a polynomial fashion.

Invariance checking relies mostly on value comparison, which, in hardware terms, translates into simple combinational circuits consisting of inverters, AND, OR, and XOR gates. Therefore, the NoCAlert checkers provide a lightweight and holistic approach to runtime fault detection, as will be demonstrated through hardware synthesis results in Section 5.

4.3. Faults That Do Not Cause Invariance Violations

As previously mentioned, invariance checking only detects *illegal* outputs, not necessarily *incorrect* ones. Faults that give rise to *func-*

tionally legal outputs – based on the given input – will not be detected. A simple NoC example to illustrate this scenario is the RC unit's functionality. Suppose a packet enters a router from the East port and is destined to the West output port. Even under deterministic XY routing, a misdirection to the North output port will *not* constitute an invariance violation, since X-to-Y turns are allowed in XY routing. Moreover, *adaptive* routing algorithms – such as Duato's Protocol [38] – inherently allow more than one routing options to avoid congestion. It is clear that faults in the RC unit have a good chance of still returning a valid/legal output that does not violate any invariance.

The two elemental questions here are the following:

- If such non-invariant upsets cause some other functional/invariance violation *later on* in the network, will the fault be caught by one/some subsequent NoCAlert checkers?
- If these non-invariant upsets do *not* cause any other functional/invariance violation *later on* in the network (i.e., they are never caught by any NoCAlert checker), do they end up affecting the overall network correctness (as defined in Section 4.1 and [37, 15])?

An example relevant to the second question is when a packet requests VC1 of a specific output port, but a fault occurrence causes the grant of, say, VC2 of the same output port, which also happens to be available. If both of these VCs belong to the same protocol-level message class, then this fault does *not* cause an invariance violation and it is, in fact, *benign*, i.e., no functional error manifests itself at the network or system level later on.

The extensive simulations of Section 5 will answer these two important questions. It turns out (empirically) that all the non-invariant faults that end up causing a functional error later on are, indeed, successfully captured by subsequent NoCAlert checkers, whereas the non-invariant faults that do not cause any other invariance violation later on turn out to be benign, as far as overall network correctness is concerned.

4.4. Applicability of the NoCAlert Framework to Any Router Micro-architecture

Based on our exploration so far, it is clear that the invariance concept is closely related to the micro-architecture under test. Changes in the router's micro-architecture may result in subtle (or not so subtle) changes in the components' invariances. However, the underlying principles will still be the same: study each individual module and identify invariances, while gradually moving up to coarser granularities (e.g., port-level). There are many proposed router architectures [39, 33, 40, 41], with each one involving changes to the constituent modules, or the pipeline stages and associated flow. The inherent modularity of all router designs (a direct consequence of the router's parallel nature) allows the designer to fairly easily identify the new functional invariances.

This sub-section will briefly investigate the key changes to the invariances of the generic router model when some key router parameters are varied. The chosen variations are typical alterations observed in the literature. For example, the router design may forego the use of VCs, it may choose to employ non-atomic FIFO buffers, it may implement a speculative design (e.g., the VA and SA happening concurrently), and it may employ a more elaborate routing algorithm. By exploring how these changes will affect invariance checking, one may appreciate the flexibility and widespread applicability of the NoCAlert scheme.

In the absence of virtual channels in the design, the VA pipeline stage is eliminated. Hence, all the invariance checks pertaining to the VA stage in Table 1 may be removed. For example, invariances 29 and 30 in the table are no longer needed.

Non-atomic buffers allow the simultaneous storage of flits belonging to different packets (albeit without mixing), unlike the atomic buffers in the baseline architecture, which only allow the flits of a single packet to reside in the buffer at any given time. If non-atomic buffers are used, all invariances that forbid a new packet to arrive in an already-occupied VC buffer are discarded. At the same time, however, a new invariance is created (invariance 27 in Table 1). The mixing of flits from two different packets is still forbidden in non-atomic buffers. This means that an assertion should be raised if the flit following a tail flit is *not* a header flit of a new packet.

In speculative router designs [36], the VA and SA stages are executed in parallel. In this case, the SA may, in fact, finish before the VA stage. Thus, invariance 17 in Table 1 must be altered, so as not to raise an assertion if SA succeeds before VA is done.

The functional definition of a routing algorithm defines its invariances. Most routing algorithms have some turn restrictions in order to prevent network deadlocks and livelocks, as well as protocol deadlocks. Some routing algorithms also provide a specific output VC, in addition to the output direction. In all cases, the NoCAlert checkers are derived from these restrictions. For example, Duato's Protocol [38] dictates that "when making a turn from the East to the North, a packet must enter VC0." This statement immediately defines an assertion checker.

5. Experimental Evaluation

5.1. Evaluation Framework

The goal of the experimental evaluation is to thoroughly assess the *efficacy* and *efficiency* of the NoCAlert mechanism in a realistic environment. Our evaluation approach is double-faceted and consists of (a) extensive simulations in a cycle-accurate simulator, and (b) hardware evaluation based on a full Verilog implementation of NoCAlert and synthesis using 65 nm commercial standard-cell libraries.

For the former part, the cycle-accurate GARNET NoC simulator [42] is employed. GARNET models the packet-switched routers down to the micro-architectural level. The simulator was further extended with all the checker modules listed in Table 1 (see Section 4) and the fault injection framework to be described in Section 5.2.

Since the focus of this work is the *fault detection performance* of NoCAlert (and not the network/system performance), the use of synthetic traffic patterns in an 8×8 mesh suffices to accurately capture the salient characteristics of the design. Synthetic traffic patterns are typically more effective in stressing the router design to its limits and isolating the inherent attributes of the network itself. Hence, we employ synthetic (uniform random) traffic at various injection rates to ensure all router components are stressed over a range of traffic intensities.

The NoCAlert framework is also compared to ForEVeR [15], a recently proposed state-of-the-art fault detection and recovery framework (see Section 2). The ForEVeR mechanism was cycle-accurately implemented within GARNET with all three of its key fault-detecting techniques: the secondary checker network (including the counters and timers), the Allocation Comparator from [19], and the end-to-end checker.

Without loss of generality, the router architecture assumed in this evaluation is the baseline implementation described in Section 3.1. The router is five-stage pipelined (4 intra-router stages + 1 link traversal stage), with four 5-flit deep VCs per input port, and 128-bit inter-router links. Atomic VC buffers, wormhole switching, and

credit-based flow control are also assumed. The routing algorithm used is deterministic XY.

For the hardware evaluation part, we implemented the baseline NoC router augmented with the NoCAlert mechanism (all 32 invariance checker modules) in Verilog Hardware Description Language (HDL). The resulting design was synthesized using Synopsys Design Compiler and 65 nm commercial TSMC libraries at 1 V operating voltage and 1 GHz clock frequency. The results were used to perform detailed area/power/timing analysis and evaluate the overhead footprint of NoCAlert.

5.2. Fault Model and Fault Injection Framework

Throughout the evaluation, we assume the occurrence of *single* faults in the NoC mesh. Specifically, the simulator injects *single-bit, single-event transient faults* at different locations and at different instances (network states). The above-mentioned fault model is widely used in the literature and it was chosen as a proof-of-concept for NoCAlert. More elaborate fault models are left for future work. Even though we employ *transient* fault injections for the purposes of our simulations, the mechanism works with permanent failures in an identical manner. Effectively, the fault model used evaluates fault behavior for single-event upsets and single permanent faults. The difference is that the NoCAlert checkers will raise permanent/prolonged (rather than momentary) assertions upon the occurrence of a permanent/intermittent fault. In other words, since the NoCAlert checkers raise an exception upon an invariance violation, NoCAlert's performance/accuracy is orthogonal to whether the invariance is temporary or permanent; as soon as the invariance violation *commences*, NoCAlert will detect it. The reasoning is that a permanent fault, or an intermittent fault, will trigger the same checker as a transient fault, but the checker's flag will remain raised for more than one cycle (indicating an intermittent, or permanent, fault). Note that even if the erroneous value disappears after one clock cycle, the effects of that short "malfunction" perturbation may propagate through the network with unpredictable results.

Our fault model looks at the router micro-architecture at the **fine granularity of individual sub-components**. These sub-components comprise all the modules responsible for the router's control logic: individual RC units, control status tables, VC buffer status, arbiters in both VA and SA, and the crossbar control logic. Our only assumption is that the packet/flit *contents* are already protected by error-detecting codes (see Section 3.3), so the datapath of the router is also covered. Our model has the capability of *injecting single-bit faults at the inputs and the outputs of each individual module*. The fault injection framework is illustrated in Figure 5. By looking at the router micro-architecture at this fine granularity, we are able to inject single-bit faults at 205 different locations within a single 5-port NoC router. Taking into account corner and edge routers (which have fewer ports), the *total number of fault locations* is 11,808 in an 8×8 mesh network.

5.3. Experimental Methodology

One simulation run at a single traffic injection rate and one network state consists of 11,808 different simulations (to exhaustively inject faults in all possible locations of an 8×8 mesh, assuming the specific single-fault injection model used in this work). The traffic injection rate was varied from low to high (0.1–0.4 flits/node/cycle) in steps of 0.05 flits/node/cycle. Moreover, three different scenarios of fault injection instances were studied (fault injection at cycle 0, 32K, and 64K). Hence, 21 different scenarios were investigated (7 injection rates × 3 injection times), for a total of $21 \times 11,808 \approx 248$K fault-injection simulations.

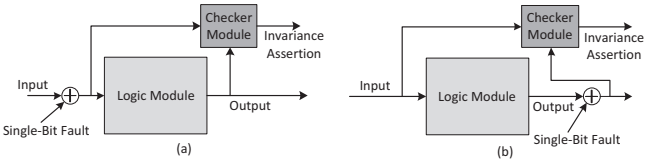


Figure 5: Abstract view of the employed fault injection framework. The evaluation framework used in this work has the capability of injecting single-bit faults at the inputs (a), or outputs (b), of each individual router module. The module granularity is very fine, which results in 11,808 possible fault locations in an 8×8 mesh network.

The exact same experiments were also run in a *fault-free* environment and detailed flit ejection logs were collected and compiled in a so called *Golden Reference* (GR) report. The GR is then used to ensure that no violations of the four network correctness rules of Section 4.1 and [37, 15] occur. Furthermore, the GR also detects any changes in the intra-packet flit order (as previously mentioned, such order violations constitute erroneous behavior). Since NoCAlert only captures faults that cause invariances, the GR is used to facilitate the investigation of the two key questions posed in Section 4.3. Moreover, by comparing the GR with the equivalent *under-fault* log report, we can study the effects of any fault occurrence on overall network correctness. This allows us to assess the *false positive* (assertions that prove benign) and *false negative* (undetected network correctness violations) performances of both NoCAlert and ForEVeR [15].

5.4. Simulation Results

As discussed in Section 5.1, simulation experiments were performed in an 8×8 2D mesh network using synthetic traffic patterns. In this sub-section, we present the results and an evaluation of NoCAlert's efficacy and efficiency in terms of several key metrics. Moreover, we conduct a quantitative comparison with the ForEVeR [15] framework.

We begin our exploration with NoCAlert's fault detection capabilities. It is important at this point to differentiate the *injected faults* from the *actual errors* manifesting themselves at the network-level (as defined in Section 4.1 and [37, 15]). NoCAlert's ultimate goal is to ensure that no *actual error at the network-level* escapes detection. Therefore, injected faults that do NOT cause a real functional error within the network are viewed as benign. Based on this crucial differentiation, we classify each of NoCAlert's detection outcomes into one of four main categories:

- **True Positive**: Event detected by NoCAlert when the injected fault causes an actual error at the network-level (network correctness violation).
- **False Positive**: Event detected by NoCAlert when the injected fault turns out to be benign.
- **True Negative**: Nothing detected by NoCAlert when the injected fault turns out to be benign.
- **False Negative**: Nothing detected by NoCAlert when the injected fault causes an actual error at the network-level (network correctness violation).

In order to identify which injected faults turned out to be malicious (i.e., they caused a network correctness violation), we used the Golden Reference (GR) log report described in Section 5.3.

Obviously, the most important metric when evaluating the performance of a *detection* mechanism is the occurrence of *False Negatives*, i.e., actual faults that evade the detection process.

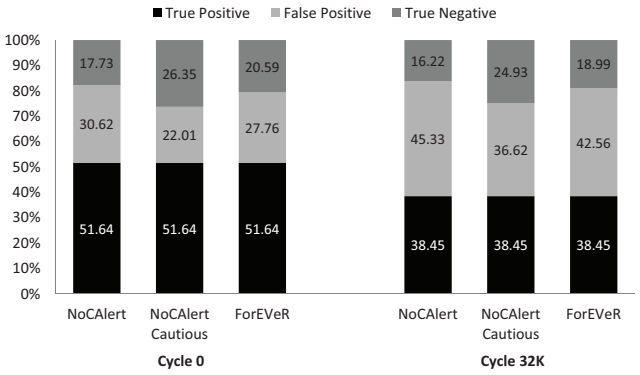


Figure 6: Fault coverage breakdown (over all injected faults) using synthetic (uniform random) traffic in an 8×8 mesh at two different fault injection instances (cycle 0 and cycle 32K). The "NoCAlert Cautious" bars refer to a system where Invariances 1 and 3 of Table 1 are considered low risk (see text for details).

Observation 1: Out of all the simulations we ran, NoCAlert registered *zero* false negatives. In other words, *all faults that violated network correctness were successfully captured* by NoCAlert. The same was true for ForEVeR [15]. Thus, both mechanisms exhibit the same fault detection *accuracy*.

Figure 6 presents a breakdown of the fault detection performance of both NoCAlert and ForEVeR at two different fault injection instances: cycle 0 and cycle 32K (the results for fault injection at cycles 32K and 64K are very similar; thus, only the 32K results are shown here for brevity). The results for cycle 0 are representative of an empty network, while the results for cycle 32K are representative of networks at steady-state (warmed up). Note that the true positive percentages are identical for NoCAlert and ForEVeR, since both mechanisms detected *all* network correctness violations (all of the injected faults that actually violated the network correctness). The notable difference is in the *False Positives*, where NoCAlert is slightly worse in both cases. This result is attributed to the real-time nature of NoCAlert, which raises assertions instantaneously. Instead, ForEVeR is epoch-based, which means that some benign faults simply "vanish" by the time the epoch expires. In general, the false positive percentages are higher for cycle 32K – as compared to cycle 0 – because violations are more likely to be masked by other traffic in a warmed up network. For example, in an empty network, an erroneous switch allocation request would propagate to the output uncontested (since there are no other packets competing for crossbar access). However, in a more congested environment, the erroneous request may lose the arbitration to another packet.

The false positive percentages may be markedly reduced if the *recovery* mechanism's reaction is guided by the checkers' *risk levels*. In other words, the NoCAlert checkers may be classified into different categories, based on their *risk levels*. Low-risk checkers would trigger a *delayed/deferred* response, in order to account for the high probability of a false positive. For instance, we made a very interesting observation regarding NoCAlert. In our conducted set of experiments, invariances 1 and 3 of Table 1 *never* led to network-level incorrectness when asserted *alone*, even though they might *theoretically* have led to a deadlock. These invariances are violated if the RC unit misdirects a header flit (possibly in a direction further away from the packet's destination). We noticed that many benign faults registered as false positives by NoCAlert were caused by those two invariances. In all those cases, the invariances were asserted by

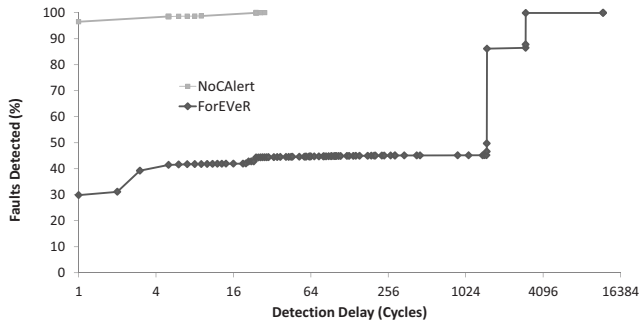


Figure 7: Cumulative fault-detection delay distribution for the true positive faults (The epoch duration in ForEVeR was set to 1,500 cycles; see text for details).

themselves (no other assertion was raised). Hence:

Observation 2: If the fault recovery mechanism connected to NoCAlert sees either Invariance 1 or Invariance 3 (Table 1) violated, without any other assertions raised, it could move into a "cautious" state, whereby the fault recovery mechanism is not triggered until there is further evidence later on that a deadlock actually occurred. If this strategy is followed, then NoCAlert's *False Positive* rates would drop to 22.01% and 36.62% for cycles 0 and 32K, respectively, as indicated by the "NoCAlert Cautious" bars in Figure 6.

Invariance 5 in Table 1 exhibits noteworthy behavior. Said invariance is violated whenever an arbiter produces an all-zero grant vector (i.e., no arbitration winner is declared), even though there was at least one active client request. If the fault is transient or intermittent, this fault would only result in brief performance degradation, similar to a NOP instruction in a microprocessor's pipeline. However, if the fault is permanent (i.e., the checker remains permanently asserted), the consequences could be quite dramatic. The fault may lead to network/protocol deadlocks.

Observation 3: Invariance 5 in Table 1 exhibits the unique characteristic of being benign (in terms of network correctness) under transient/intermittent faults, but malicious under permanent faults.

The real strength of NoCAlert is its fault detection *latency*. Figure 7 shows the cumulative fault detection delay distribution for both NoCAlert and ForEVeR. In this figure, only faults that resulted in true positives were evaluated. Notice that 97% of all faults are captured instantaneously (in the same cycle) by NoCAlert, 99% are captured within 9 clock cycles, and 100% are captured after 28 cycles. ForEVeR's epoch-based scheme takes significantly longer, with 99% of faults being captured after 3,000 cycles and 100% captured after 11,995 cycles. The epoch duration in ForEVeR was set to 1,500 cycles, which was the shortest period that did not yield excessive false positives *under the fault model employed in this work*. These results demonstrate that:

Observation 4: NoCAlert provides near-instantaneous fault detection with a staggering 97% of all true positive faults captured at the instance of injection (same cycle). The worst-case detection latency is only 28 cycles after fault injection. Moreover, NoCAlert achieves *several orders of magnitude* lower fault detection latency than ForEVeR [15].

In order to answer the critical questions posed in Section 4.3, we need to examine all injected faults that did not result in an invariance violation at the instance of fault injection. It turns out that 78% of those faults did not cause a subsequent invariance violation, and *all of them* turned out to be benign (no network correctness violation). The remaining 22% caused a subsequent invariance violation and

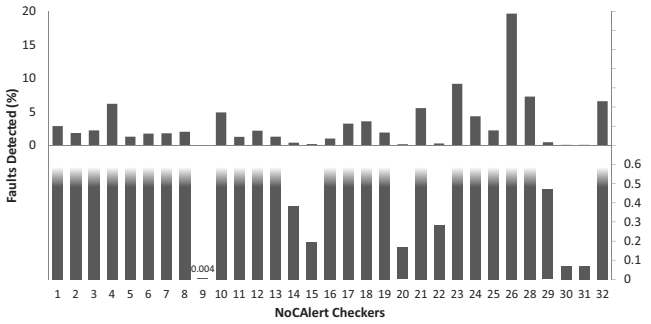


Figure 8: Percentage of invariance violations captured by each individual NoCAlert checker of Table 1 (over all experiments). The bottom part of the figure has a finer y-axis scale and focuses on the very low y-axis values of the top part. Invariance 27 is missing, because it is only applicable to non-atomic VC buffers.

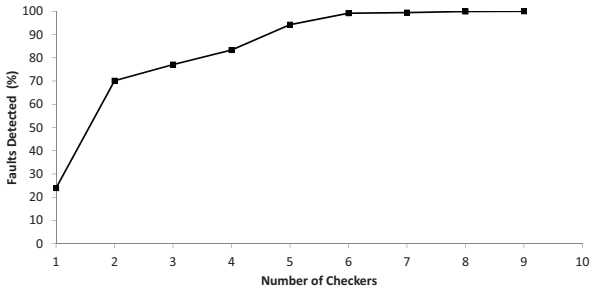


Figure 9: Cumulative distribution of invariance violations as a function of the number of simultaneously asserted checkers.

were successfully captured by NoCAlert.

Observation 5: Injected faults that do *not* cause any invariance violation in the network are *always* benign (i.e., they never cause any network correctness violation).

Figures 8 and 9 evaluate the behavior of the 32 invariance checkers. Specifically, Figure 8 shows the percentage of invariance violations caught by each individual checker over all experiments. Note that Invariance 27 is missing, because it refers to non-atomic buffers (we used atomic VC buffers in our simulations, as stated in Section 5.1). It should also be noted that all checkers detected invariances in the absence of any other checker assertions. This fact indicates that *no single checker is redundant*. Finally, Figure 9 shows the cumulative distribution of invariance violations as a function of the number of simultaneously asserted checkers. Most invariances were caught by two checkers, while the maximum number of checkers triggered due to a single invariance violation was 9.

5.5. Hardware Evaluation – Area/Power/Timing Overhead

As described in Section 5.1, a baseline NoC router augmented with the complete NoCAlert mechanism was implemented in Verilog HDL and synthesized using 65 nm commercial standard-cell libraries.

In order to assess the *scalability* of NoCAlert, we vary the number of VCs per port from two [41] to eight [39] and evaluate the NoCAlert percentage area and power overhead. The number of VCs per port is dictated by the employed routing algorithm (e.g., deterministic vs. adaptive) and/or the cache-coherence protocol (number of message classes). The **area** results are shown in Figure 10. To better appreciate the size of NoCAlert, we also implemented a design with Double Modular Redundancy (DMR) in the entire NoC control logic (designated as "DMR-CL" in the figure). DMR serves

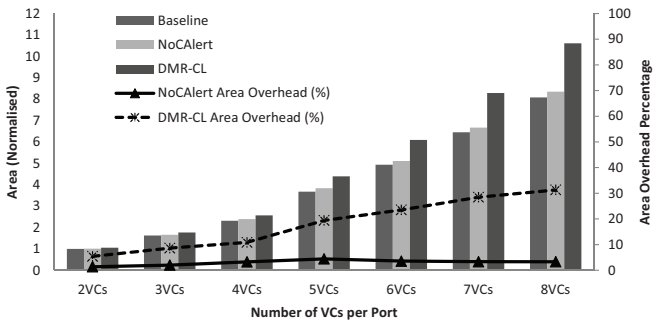


Figure 10: The NoCAlert area overhead as a function of the number of VCs per input port. A comparison with double modular redundancy in the control logic ("DMR-CL") is also presented.

as the most complete fault detection solution possible, albeit a very expensive one. Clearly, the NoCAlert area overhead is minimal and ranges from 1.38% to 4.42% (3%, on average) and the *percentage* overhead remains fairly constant as the number of VCs increase. On the other hand, the *percentage* area overhead of DMR increases linearly from 5.41% in the case of two VCs, up to 31.32% in the case of eight VCs per port.

The **power** results exhibit the same trends and are, thus, omitted for brevity. The absolute numbers, however, are much smaller for NoCAlert, since the checkers comprise purely combinational logic and have no power-hungry storage elements. Hence, the percentage power overhead ranges from 0.3% to 1.2% (0.7%, on average), i.e., it is negligible. The power numbers were extracted from the Synopsys Design Compiler power report, with switching activity set to 50% for all nets.

The final key design metric evaluated was the **critical path delay**, which sets the maximum possible operating frequency. Our synthesis results indicate minimal impact on the critical path of at most 3% and, on average, around 1%. This means that the proposed NoCAlert mechanism is, essentially, transparent to overall network operation.

These results corroborate the fact that NoC control logic checkers used to detect only illegal outputs have significantly lower hardware cost than the units they check.

6. Conclusions

This paper proposes NoCAlert, a comprehensive on-line and real-time fault detection mechanism that ensures 0% false negatives within the NoC, under the employed fault model. NoCAlert is based on the concept of *invariance checking*, whereby the outputs of the control logic modules of the on-chip network are constantly checked for *illegal* outputs, based on current inputs. By combining a collection of such micro-checker modules dispersed throughout the router's control logic modules, the proposed mechanism implements real-time hardware assertions. The checkers operate seamlessly and concurrently with normal NoC operation, thus obviating the need for periodic (epoch-based), or triggered-based, self-testing.

Extensive simulation results validate the efficacy of the NoCAlert mechanism and yield important insight as to the behavior of the network when non-invariant faults (that evade the checkers) occur. Specifically, non-invariant faults either cause some subsequent invariance violation (and are captured), or they prove benign at the network/system level. Hardware synthesis analysis using 65 nm commercial libraries demonstrates the extremely lightweight nature of NoCAlert in terms of area/power/timing overhead. Furthermore, a detailed comparison with a recently proposed framework [15] high-

lights higher than 100× improvements in detection latency, with no loss in detection accuracy and with much lower overall complexity.

In summary, this work demonstrates the potential for extremely accurate and near-instantaneous fault detection within the NoC using minimally intrusive hardware-based invariance checkers.

Acknowledgments

This work was supported by "EuroCloud, Project No 247779" of the European Commission 7th RTD Framework Programme – Information and Communication Technologies: Computing Systems, and by the Cyprus Research Promotion Foundation's Grant ΤΠΕ/ΠΛΗΡΟ/0609(ΒΙΕ)/09 (co-funded by the Republic of Cyprus and the European Regional Development Fund).

References

[1] S. Damaraju et al. A 22nm ia multi-cpu and gpu system-on-chip. In *Proc. of the International Solid-State Circuits Conference (ISSCC)*, 2012.

[2] K. Olukotun et al. The case for a single-chip multiprocessor. In *Proc. of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 1996.

[3] W.J. Dally and B. Towles. Route packets, not wires: on-chip interconnection networks. In *Proc. of the Design Automation Conference (DAC)*, 2001.

[4] S.R. Nassif, N. Mehta, and Yu Cao. A resilience roadmap. In *Proc. of the Design, Automation and Test in Europe Conference (DATE)*, 2010.

[5] S. Borkar. Microarchitecture and design challenges for gigascale integration. In *Proc. of the International Symposium on Microarchitecture (MICRO)*, 2004.

[6] E Wu et al. Interplay of voltage and temperature acceleration of oxide breakdown for ultra-thin gate oxides. In *International Journal of Solid-State Electronics*, November 2002.

[7] C. Nicopoulos et al. On the effects of process variation in network-on-chip architectures. *In IEEE Trans. on Dependable and Secure Computing*, July 2010.

[8] K. Aisopos, C.-H.O. Chen, and L.S. Peh. Enabling system-level modeling of variation-induced faults in networks-on-chips. In *Proc. of the Design Automation Conference (DAC)*, 2011.

[9] K. Constantinides et al. Bulletproof: a defect-tolerant cmp switch architecture. In *Proc. of the International Symposium on High-Performance Computer Architecture (HPCA)*, 2006.

[10] D. Fick et al. Vicis: A reliable network for unreliable silicon. In *Proc. of the Design Automation Conference (DAC)*, 2009.

[11] M.R. Kakoee, V. Bertacco, and L. Benini. Relinoc: A reliable network for priority-based on-chip communication. In *Proc. of the Design, Automation and Test in Europe Conference (DATE)*, 2011.

[12] A. Strano et al. Exploiting network-on-chip structural redundancy for a cooperative and scalable built-in self-test architecture. In *Proc. of the Design, Automation and Test in Europe Conference (DATE)*, 2011.

[13] M. Hosseinabady, A. Dalirsani, and Z. Navabi. Using the inter- and intra-switch regularity in noc switch testing. In *Proc. of the Design, Automation and Test in Europe Conference (DATE)*, 2007.

[14] M.R. Kakoee, V. Bertacco, and L. Benini. A distributed and topology-agnostic approach for on-line noc testing. In *Proc. of the International Symposium on Networks-on-Chip (NOCS)*, 2011.

[15] R. Parikh and V. Bertacco. Formally enhanced runtime verification to ensure noc functional correctness. In *Proc. of the International Symposium on Microarchitecture (MICRO)*, 2011.

[16] A. Kohler, G. Schley, and M. Radetzki. Fault tolerant network on chip switching with graceful performance degradation. *In IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, June 2010.

[17] Albert Meixner, Michael E. Bauer, and Daniel Sorin. Argus: Low-cost, comprehensive error detection in simple cores. In *Proc. of the International Symposium on Microarchitecture (MICRO)*, 2007.

[18] M. Koibuchi et al. A lightweight fault-tolerant mechanism for network-on-chip. *in Proc. of the International Symposium of Networks-on-Chip (NOCS)*, 2008.

[19] D. Park et al. Exploring fault-tolerant network-on-chip architectures. In *Proc. of the International Conference on Dependable Systems and Networks (DSN)*, 2006.

[20] S. Shamshiri, A. Ghofrani, and Kwang-Ting Cheng. End-to-end error correction and online diagnosis for on-chip networks. In *Proc. of the International Test Conference (ITC)*, 2011.

[21] M. Palesi, S. Kumar, and V. Catania. Leveraging partially faulty links usage for enhancing yield and performance in networks-on-chip. *In IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, March 2010.

[22] J. Duato. A theory of fault-tolerant routing in wormhole networks. *In IEEE Trans. on Parallel and Distributed Systems (TPDS)*, August 1997.

[23] W.J. Dally et al. The reliable router: A reliable and high-performance communication substrate for parallel computers. In *Proc. of the International Workshop on Parallel Computer Routing and Communication (PRCW)*, 1994.

[24] S. Rodrigo et al. Addressing manufacturing challenges with cost-efficient fault tolerant routing. In *Proc. of the International Symposium on Networks-on-Chip (NOCS)*, 2010.

[25] T. Dumitraş, S. Kerner, and R. Mărculescu. Towards on-chip fault-tolerant communication. In *Proc. of the Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2003.

[26] B. Fu et al. An abacus turn model for time/space-efficient reconfigurable routing. In *Proc. of the International Symposium on Computer Architecture (ISCA)*, 2011.

[27] T. Moscibroda and O. Mutlu. A case for bufferless routing in on-chip networks. In *Proc. of the International Symposium on Computer Architecture (ISCA)*, 2009.

[28] A. Kohler and M. Radetzki. Fault-tolerant architecture and deflection routing for degradable noc switches. In *Proc. of the International Symposium on Networks-on-Chip (NOCS)*, 2009.

[29] D. Fick et al. A highly resilient routing algorithm for fault-tolerant nocs. In *Proc. of the Design, Automation and Test in Europe Conference (DATE)*, 2009.

[30] S. Murali et al. A multi-path routing strategy with guaranteed in-order packet delivery and fault-tolerance for networks on chip. In *Proc. of the Design Automation Conference (DAC)*, 2006.

[31] K. Aisopos et al. Ariadne: Agnostic reconfiguration in a disconnected network environment. In *Proc. of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2011.

[32] V. Puente et al. Immunet: Dependable routing for interconnection networks with arbitrary topology. *In IEEE Trans. on Computers*, December 2008.

[33] J. Kim et al. A gracefully degrading and energy-efficient modular router architecture for on-chip networks. In *Proc. of the International Symposium on Computer Architecture (ISCA)*, 2006.

[34] A. Ghofrani et al. Comprehensive online defect diagnosis in on-chip networks. In *Proc. of the VLSI Test Symposium (VTS)*, 2012.

[35] C. A. R. Hoare. An axiomatic basis for computer programming. *In Communications of the ACM*, October 1969.

[36] L.S. Peh and W.J. Dally. A delay model and speculative architecture for pipelined routers. In *Proc. of the International Symposium on High-Performance Computer Architecture (HPCA)*, 2001.

[37] D. Borrione et al. A generic model for formally verifying noc communication architectures: A case study. In *Proc. of the International Symposium on Networks-on-Chip (NOCS)*, 2007.

[38] J. Duato, S. Yalamanchili, and L. Ni. *Interconnection Networks: An Engineering Approach*. IEEE Computer Society Press, Los Alamitos, CA, USA, 1st edition, 1997.

[39] J. Howard et al. A 48-core ia-32 processor in 45 nm cmos using on-die message-passing and dvfs for performance and power scaling. *In IEEE Journal of Solid-State Circuits*, Jan. 2011.

[40] A. Kumary et al. A 4.6tbits/s 3.6ghz single-cycle noc router with a novel switch allocator in 65nm cmos. In *Proc. of the International Conference on Computer Design, 2007. (ICCD)*, 2007.

[41] S.R. Vangal et al. An 80-tile sub-100-w teraflops processor in 65-nm cmos. *In IEEE Journal of Solid-State Circuits*, Jan. 2008.

[42] N. Agarwal et al. Garnet: A detailed on-chip network model inside a full-system simulator. In *Proc. of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2009.

2012 IEEE/ACM 45th Annual International Symposium on Microarchitecture

Cache-Conscious Wavefront Scheduling

Timothy G. Rogers[1] Mike O'Connor[2] Tor M. Aamodt[1]
[1]University of British Columbia [2]Advanced Micro Devices Inc. (AMD)
tgrogers@ece.ubc.ca, mike.oconnor@amd.com, aamodt@ece.ubc.ca

Abstract

This paper studies the effects of hardware thread scheduling on cache management in GPUs. We propose Cache-Conscious Wavefront Scheduling (CCWS), an adaptive hardware mechanism that makes use of a novel intra-wavefront locality detector to capture locality that is lost by other schedulers due to excessive contention for cache capacity. In contrast to improvements in the replacement policy that can better tolerate difficult access patterns, CCWS shapes the access pattern to avoid thrashing the shared L1. We show that CCWS can outperform any replacement scheme by evaluating against the Belady-optimal policy. Our evaluation demonstrates that cache efficiency and preservation of intra-wavefront locality become more important as GPU computing expands beyond use in high performance computing. At an estimated cost of 0.17% *total chip area, CCWS reduces the number of threads actively issued on a core when appropriate. This leads to an average 25% fewer L1 data cache misses which results in a harmonic mean 24% performance improvement over previously proposed scheduling policies across a diverse selection of cache-sensitive workloads.*

1. Introduction

Manycore accelerators, such as GPUs, enable efficient execution of parallel workloads, allowing continued performance improvement with each process node despite diminished voltage scaling [11]. Programming interfaces like OpenCL [24] and CUDA [33] require the user to define the behavior of a single scalar thread which can be run thousands of times across dozens of simple *single instruction multiple data* (SIMD) compute units (also known as shader cores). This type of architecture, sometimes referred to as *single instruction multiple thread* (SIMT) [28], allows the GPU's SIMD core to make progress on multiple threads using a single instruction by grouping them into wavefronts (or warps), thus amortizing the instruction fetch and decode overhead.

Each cycle, a hardware wavefront scheduler must decide which of the multiple active wavefronts execute next. Our work focuses on this decision. The goal of a wavefront scheduler is to ensure the execution pipeline is kept active in the presence of long latency operations. The inclusion of caches on GPUs [32] can reduce the latency of memory operations and act as a bandwidth filter, provided there is some locality in the access stream. Figure 1 presents the average number of hits and misses *per thousand instructions* (PKI) of *highly cache-sensitive* (HCS) and *moderately cache-sensitive* (MCS) benchmark access streams using an unbounded *level one data* (L1D) cache. The figure separates hits into two classes. We classify locality that occurs when data is initially referenced and re-referenced from the same wavefront as *intra-wavefront locality*. Locality resulting from data that is initially referenced by one wavefront and re-referenced by another is classified as *inter-wavefront locality*. Intra-wavefront locality is a combination of intra-thread locality [27] (where data is private to a single scalar thread) and inter-thread locality where data is shared among scalar threads in the same wavefront.

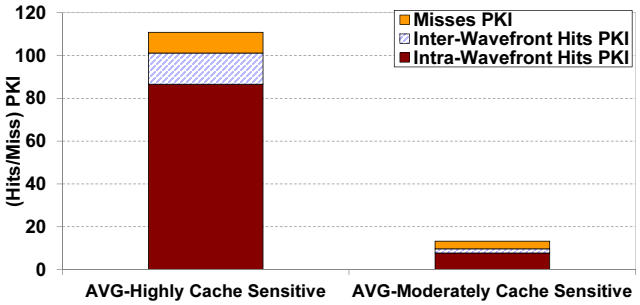


Figure 1: Average hits and misses *per thousand instructions* (PKI) using an unbounded L1 data cache (with 128B lines) on cache-sensitive benchmarks.

Figure 1 illustrates that the majority of data reuse observed in our HCS benchmarks comes from intra-wavefront locality.

To exploit this type of locality in HCS benchmarks, we introduce Cache-Conscious Wavefront Scheduling (CCWS). CCWS uses a novel *lost intra-wavefront locality detector* (LLD) that alerts the scheduler if its decisions are destroying intra-wavefront locality. Based on this feedback, the scheduler assigns intra-wavefront locality scores to each wavefront and ensures that those wavefronts losing intra-wavefront locality are given more exclusive access to the L1D cache.

Simple wavefront scheduling policies such as round-robin are oblivious to their effect on intra-wavefront locality, potentially touching data from enough threads to cause thrashing in the L1D. A two level scheduler such as that proposed by Narasiman et al. [31] exploits inter-wavefront locality while ensuring wavefronts reach long latency operations at different times by scheduling groups of wavefronts together. However, Figure 1 demonstrates that the HCS benchmarks we studied will benefit more from exploiting intra-wavefront locality than inter-wavefront locality. Existing schedulers do not take into account the effect issuing more wavefronts has on the intra-wavefront locality of those wavefronts that were previously scheduled. In the face of L1D thrashing, the round-robin nature of their techniques will cause the destruction of older wavefront's intra-wavefront locality.

Figure 2 illustrates the cache size sensitivity of our benchmarks (described in Section 4) when using a round-robin scheduler and the baseline system described in Section 4. Although all of these benchmarks are somewhat cache-sensitive, the HCS benchmarks plotted on the left in Figure 2 see 3× or more performance improvement with a much larger L1 data cache.

For GPU-like architectures to effectively address a wider range of workloads, it is critical that their performance on irregular workloads is improved. Recent work on the highly cache-sensitive Memcached (MEMC) [16] and BFS [30] has shown promising results running these commercially relevant irregular parallel workloads on GPUs. However, since current GPUs face many performance challenges running irregular applications, there are relatively few of them written. In this work we evaluate a set of irregular GPU applications and

1072-4451/12 $26.00 © 2012 IEEE
DOI 10.1109/MICRO.2012.16

IEEE computer society

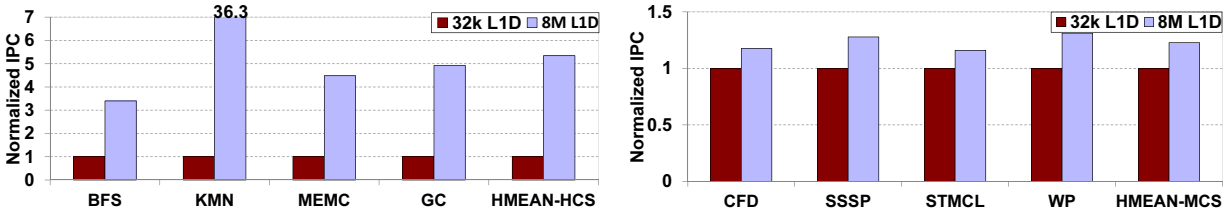


Figure 2: Performance using a loose round-robin scheduler at various L1D cache sizes for highly cache-sensitive (left) and moderately cache-sensitive benchmarks (right), normalized to a cache size of 32k. All caches are 8-way set-associative with 128B cache lines.

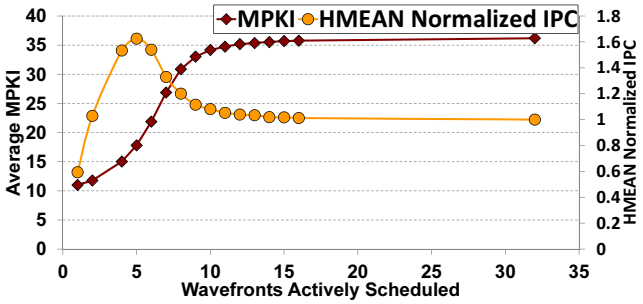


Figure 3: Average *misses per thousand instructions* (MPKI) and *harmonic mean* (HMEAN) performance improvement of HCS benchmarks with different levels of multithreading. *Instructions per cycle* (IPC) is normalized to 32 wavefronts.

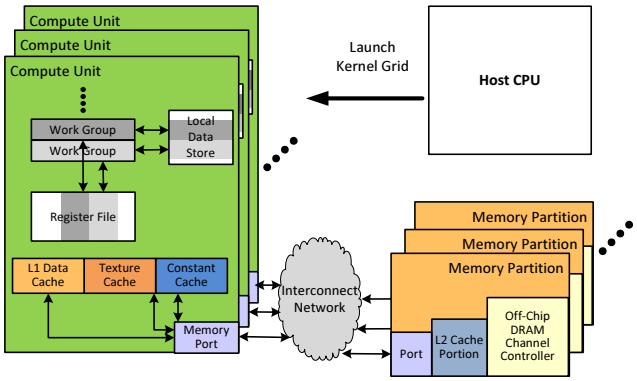


Figure 4: Overview of our GPU-like baseline accelerator architecture.

demonstrate their performance can be highly sensitive to the GPU's wavefront scheduling policy.

Figure 3 highlights the impact wavefront scheduling can have on preserving intra-wavefront locality. It shows the effect of statically limiting the number of wavefronts actively scheduled on a core. Peak throughput occurs at a multithreading value less than maximum concurrency, but greater than the peak cache performance point (which limits concurrency to a single wavefront). Although it may seem counterintuitive to limit the amount of multithreading in a GPU, our data demonstrates a trade-off between hiding long latency operations and creating more of them by destroying intra-wavefront locality.

Our work draws inspiration from cache replacement and insertion policies in that it attempts to predict when cache lines will be reused. However, cache way-management policies' decisions are made among a small set of blocks. A thread scheduler effectively chooses which blocks get inserted into the cache from a pool of potential memory accesses that can be much larger than the cache's associativity. Similar to how cache replacement policies effectively *predict* each line's re-reference interval [21], our proposed scheduler effectively *changes* the re-reference interval to reduce the number of interfering references between repeated accesses to high locality data. Unlike scheduling approaches for managing contention implemented in the operating system [39], our technique exploits fine-grained information available to the low-level hardware scheduler.

This paper makes the following contributions:

- It identifies intra and inter wavefront locality and quantifies the trade-off between maximizing intra-wavefront locality and concurrent multithreading.
- It proposes a novel *Cache-Conscious Wavefront Scheduling* (CCWS) mechanism which can be implemented with no changes to the cache replacement policy. CCWS uses a novel *lost intra-wavefront locality detector* (LLD) to update an adaptive locality scoring system and improves the performance of HCS workloads by 63% over existing wavefront schedulers.
- It demonstrates that CCWS reduces L1D cache misses more than the Belady-optimal replacement scheme.
- It demonstrates that CCWS can be tuned to trade-off power and performance. A power-tuned configuration of CCWS reduces energy-expensive L1D cache misses an additional 18% above the performance tuned configuration while still achieving a 49% increase in performance on HCS workloads.

The rest of this paper is organized as follows, Section 2 describes our baseline architecture, Section 3 describes our scheduling techniques including CCWS, Section 4 describes methodology, Section 5 describes our results, Section 6 describes related work, and Section 7 concludes.

2. Baseline Architecture

In this work we study modifications to the GPU-like accelerator architecture illustrated in Figure 4. The workloads we study are written in OpenCL [24] or CUDA [33]. Initially, an application begins execution on a host CPU and then a kernel is launched on the GPU. An OpenCL kernel is composed of "work items" which can be thought of as scalar threads. To facilitate communication, "work items" are collected into "work groups" which can communicate through local memory. Work items are grouped into wavefronts that execute in lock-step on a GPU core. The microarchitecture of the baseline GPU core with the extensions required to support CCWS is illustrated in Figure 5. The GPGPU-Sim 3.x Manual [1] describes the baseline pipeline in more detail.

Issuing a single wavefront memory instruction can generate up to W data cache accesses where W is the wavefront width. Modern GPUs attempt to reduce the number of memory accesses generated from each wavefront by coalescing the lane's memory requests into cache line sized chunks when there is spatial locality across the wavefront [33]. Applications with highly regular access patterns may generate as few as two memory requests that service all W lanes. Our baseline (L1D) cache evicts global data on writes and reserves cache lines on misses.

Our baseline architecture assigns workgroup sized chunks of threads to compute units. Each compute unit is able to schedule wavefronts from multiple workgroups. The number of workgroups assigned to each core is limited by the total number of threads on the core and the static resource usage of each workgroup [32].

This work focuses on the decision made by the *wavefront issue arbiter* (WIA) (❶ in Figure 5). An in-order scoreboard (❷) and decode unit (❸) control when each instruction in an instruction buffer (I-Buffer ❷) is ready to issue. The WIA decides which of these ready instructions issues next.

3. Wavefront Scheduling to Preserve Locality

This section describes our scheduling techniques, which take advantage of the insights mentioned in Section 1. First, Section 3.1 analyzes wavefront scheduling for locality preservation in an example workload with intra-wavefront locality. Next, Section 3.2 introduces *static wavefront limiting* (SWL) which gives high-level language programmers an interface to tune the level of multithreading. Finally, Section 3.3 describes *Cache-Conscious Wavefront Scheduling* (CCWS), an adaptive hardware scheduler that uses fine-grained memory system feedback to capture intra-wavefront locality.

3.1. A Code Example

Consider the inner loop of a graph processing workload presented in Example 1. The problem has been partitioned by having each scalar thread operate on all the edges of a single node. The adjoining edges of each node are stored sequentially in memory. This type of storage is common in many graph data structures including the highly space efficient compressed sparse rows [6] representation. This workload contains *intra-wavefront locality* resulting from *intra-thread locality* (data's initial reference and subsequent re-references come from the same scalar thread).

The inner loop of each scalar thread strides through attributes of its assigned node's edges sequentially. This sequential access stream has significant spatial locality that can be captured by a GPU's large cache line size (e.g. 128B). If the GPU was limited to just a single thread per compute unit, the memory loads inside the loop would hit in the L1D cache often. In realistic workloads, more than one thousand threads executing this loop will share the same L1D cache.

Example 1 Example graph algorithm kernel run by each scalar thread.

```
int node_degree = nodes[thread_id].degree;
int thread_first_edge = nodes[thread_id].starting_edge;
for ( int i = 0; i < node_degree; i++ ) {
    edge_attribtes = edges[thread_first_edge + i];
    int neighbour_node_id = edge_attributes.node;
    int edge_weight = edge_attributes.weight;
    ...
}
```

We find that if the working set of all the threads is small enough to be captured by the L1D, optimizing both cache efficiency and overall throughput is largely independent of the scheduler choice. In the other extreme, if only one wavefront's working set fits in the cache, optimizing misses would have each wavefront run to completion before starting another. Optimizing performance when the L1D is not large enough to capture all of the locality requires the wavefront scheduler to intelligently trade-off preserving intra-wavefront locality with concurrent multithreading.

If the scheduler had oracle information about the nature of the workload, it could limit the number of wavefronts actively scheduled to maximize performance. This observation motivates the introduction of *static wavefront limiting* (SWL) which allows a high-level programmer to specify a limit on the number of wavefronts actively scheduled per compute unit at kernel launch.

3.2. Static Wavefront Limiting (SWL)

Figure 3 shows the effect limiting the number of wavefronts actively scheduled on a compute unit has on cache performance and system throughput. Current programming API's such as CUDA and OpenCL allow the programmer to specify workgroup size. However, they allow as many wavefronts to run on each compute unit as shared core resources (e.g., registers, shared scratchpad memory) permit. Consequently, even if the programmer specifies small workgroups, multiple workgroups will run on the same compute unit if resources permit. As a result, the number of wavefronts/warps running at once may still be too large a working set for the L1D. For this reason, we propose *static wavefront limiting* (SWL) which is implemented as a minor extension to the wavefront scheduling logic where a register is used to determine how many wavefronts are actively issued, independent of workgroup size.

In SWL, the programmer must specify a limit on the number of wavefronts when launching the kernel. This technique is useful if the user knows the optimal number of wavefronts prior to launching the kernel, which could be determined by profiling.

In benchmarks that make use of work group level synchronization, SWL limits the number of wavefronts running until a barrier, allows the rest of the work-group to reach the barrier, then continues with the same multithreading constraints.

In Section 5 we demonstrate that the optimal number of wavefronts is different for different benchmarks. Moreover, we find this number changes in each benchmark when its input data is changed. This dependence on benchmark and input data makes an adaptive CCWS system desirable.

3.3. Cache-Conscious Wavefront Scheduling (CCWS)

This subsection first defines the goal and high level implementation of CCWS in Section 3.3.1. Next, Section 3.3.2 details how CCWS is applied to the baseline scheduling logic. Section 3.3.3 explains the *lost intra-wavefront locality detector* (LLD), followed by Section 3.3.4 which explains how our locality scoring system makes use of LLD information to determine which wavefronts can issue. Finally, Section 3.3.5 describes the locality score value assigned to a wavefront when lost locality is detected.

3.3.1. High-Level Description The goal of CCWS is to dynamically determine the number of wavefronts allowed to access the memory system and which wavefronts those should be. At a high level, CCWS is a wavefront scheduler that reacts to access level feedback (❹ in Figure 5) from the L1D cache and a *victim tag array* (VTA) at the memory stage. CCWS uses a dynamic locality scoring system to make scheduling decisions.

The intuition behind why our scoring system works can be explained by Figure 6. At a high level, each wavefront is given a score based on how much *intra-wavefront locality* it has lost. These scores change over time. Wavefronts with the largest scores fall to the bottom of a sorted stack (for example, W_2 at T_1), pushing wavefronts with smaller scores above a cutoff (W_3 at T_1) which prevents them from accessing the L1D. In effect, the locality scoring system reduces the number of accesses between data re-references from the

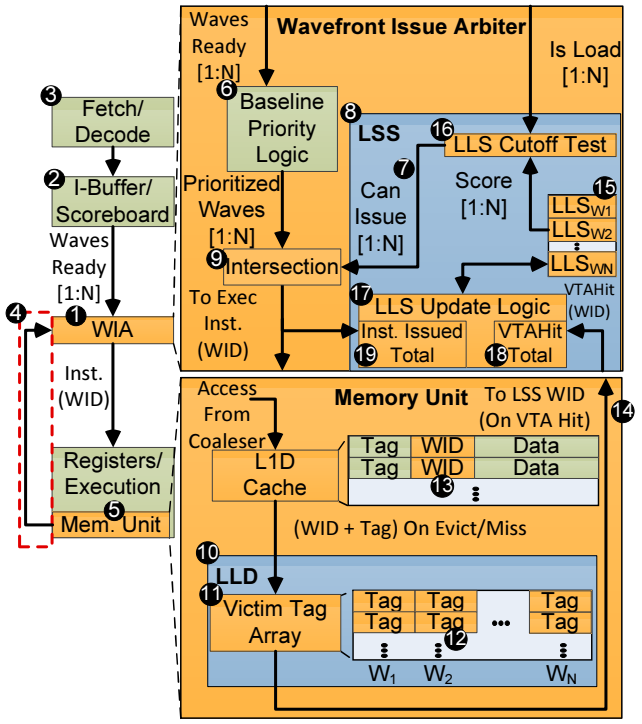


Figure 5: Modeled GPU core microarchitecture. N is the number of wavefront contexts stored on a core. LSS=*locality scoring system*, LLD=*lost intra-wavefront locality detector*, WID=*wavefront ID*, LLS=lost-locality score, VTA=*victim tag array*, I-Buffer=*instruction buffer*

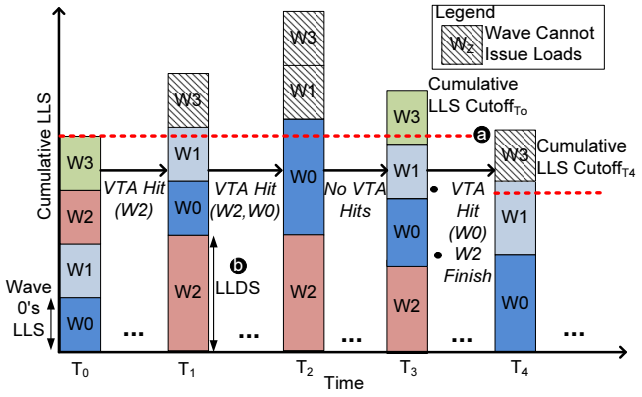


Figure 6: Locality scoring system operation example. LLS=*lost-locality score*, LLDS=*lost-locality detected score*

same wavefront by removing the accesses of other wavefronts. The following subsections describe CCWS in more detail.

3.3.2. Effect on Baseline Issue Logic Figure 5 shows the modifications to the baseline wavefront issue arbiter (❶) and memory unit (❺) required for CCWS. CCWS is implemented as an extension to the system's baseline wavefront prioritization logic (❻). This prioritization could be done in a greedy, round-robin or two level manner. CCWS operates by preventing loads that are predicted to interfere with intra-wavefront locality from issuing through a "Can Issue" bit vector (❼) output by the locality scoring system (❽). The intersection logic block (❾) selects the highest priority ready wavefront that has issue permission.

3.3.3. Lost Intra-Wavefront Locality Detector (LLD) To evaluate which wavefronts are losing intra-wavefront locality, we introduce the LLD unit (❿) which uses a *victim tag array* (VTA) (⓫). The VTA is a highly modified variation of a victim cache [23]. The sets of the VTA are sub-divided among the all the wavefront contexts supported on this core. This gives each wavefront its own small VTA (⓬). The VTA only stores cache tags and does not store line data. When a miss occurs and a line is reserved in the L1D cache, the *wavefront ID* (WID) of the wavefront reserving that line is written in addition to the tag (⓭). When that line is evicted from the cache, its tag information is written to that wavefront's portion of the VTA. Whenever there is a miss in the L1D cache, the VTA is probed. If the tag is found in that wavefront's portion of the VTA, the LLD sends a VTA hit signal to the locality scoring system (⓮). These signals inform the scoring system that a wavefront has missed on a cache line that may have been a hit if that wavefront had more exclusive access to the L1D cache.

3.3.4. Locality Scoring System Operation Figure 6 provides a visual example of the locality scoring system's operation. In this example, there are four wavefronts initially assigned to the compute unit. Time T_0 corresponds to the time these wavefronts are initially assigned to this core. Each segment of the stacked bar represents a score given to each wavefront to quantify the amount of intra-wavefront locality it has lost. We call these values *lost-locality score*s (LLS). At T_0 we assign each wavefront a constant base locality score. LLS values are stored in a max heap (⓯) inside the locality scoring system. A wavefront's LLS can increase when the LLD sends a VTA hit signal for this wavefront. The scores each decrease by one point every cycle until they reach the base locality score. The locality scoring system gives wavefronts losing the most intra-wavefront locality more exclusive L1D cache access by preventing the wavefronts with the smallest LLS from issuing load instructions. Wavefronts whose LLS falls above the cumulative LLS cutoff (ⓐ in Figure 6) in the sorted heap are prevented from issuing loads. The value of the cumulative LLS cutoff is defined as $NumActiveWaves \times BaseLocalityScore$, where $NumActiveWaves$ is the number of waves currently assigned to this core.

The LLS cutoff test block (⓰) takes in a bit vector from the instruction buffer indicating what wavefronts are attempting to issue loads. It also takes in a sorted list of LLSs, performs a prefix sum and clears the "Can Issue" bit for wavefronts attempting to issue loads whose LLS is above the cutoff. The locality scoring system is not on the critical path, can be pipelined and does not have to update the score cutoffs every compute unit cycle. In our example from Figure 6, between T_0 and T_1, W_2 has received a VTA hit and its score has been increased to the *lost-locality detected score* (LLDS), (ⓑ in Figure 6). Section 3.3.5 explains the LLDS in more detail. W_2's higher score has pushed W_3 above the cumulative LLS cutoff, clearing W_3's "Can Issue" bit if it attempts to issue a load instruction. From a microarchitecture perspective, LLSs are modified by the score update logic (⓱). The update logic block receives VTA hit Signals (with a WID) from the LLD which triggers a change to that wavefront's LLS. We limit the amount one wavefront can dominate the point system by capping each wavefront's score at LLDS, regardless of how many VTA hits it has received. Other methods of capping a wavefront's LLS were attempted and we found that limiting them to the LLDS simplified the point system and yielded the best results. In the example, between T_1 and T_2 both W_2 and W_0 have received VTA hits, pushing both W_3 and W_1 above the cutoff. Between T_2 and T_3,

no VTA hits have occurred and the scores for W_2 and W_0 have decreased enough to allow both W_1 and W_3 to issue loads again. This illustrates how the system naturally backs off thread throttling over time. Between time T_3 and T_4, W_2 finishes and W_0 has received a VTA hit to increase its score. This illustrates that when a wavefront is added or removed from the system, the cumulative LLS cutoff changes. Now that there are three wavefronts active, the LLS cutoff becomes 3× the base score. Having the LLS cutoff be a multiple of the number of active wavefronts ensures the locality scoring system maintains its sensitivity to lost-locality. If the LLS cutoff does not decrease when the number of wavefronts assigned to this core decreases, it takes a higher score per wavefront to push lower scores above the cutoff as the kernel ends. This results in the system taking more time to both constrain multithreading when locality is lost and back off thread limiting when there is no lost locality.

3.3.5. Determining the Lost-Locality Detected Score (LLDS) The value assigned to a wavefront's score on a VTA hit (the LLDS) is a function of the total number of VTA hits across all this compute unit's wavefronts (18) and all the instructions this compute unit has issued (19). This value is defined by Equation (1).

$$LLDS = \frac{VTAHits_{Total}}{InstIssued_{Total}} \cdot K_{THROTTLE} \cdot CumLLSCutoff \quad (1)$$

Using the fraction of total VTA hits divided by the number of instructions issued serves as an indication of how much locality is being lost on this core per instruction issued. A constant ($K_{THROTTLE}$) is applied to this fraction to tune how much throttling is applied when locality is lost. A larger constant favors less multithreading by pushing wavefronts above the cutoff value more quickly and for a longer period of time. Finding the optimal value of $K_{THROTTLE}$ is dependent on several factors including the number threads assigned to a core, the L1D cache size, relative memory latencies and locality in the workload. We intend for this constant to be set for a given chip configuration and not require any programmer or OS support. In our study, a single value for $K_{THROTTLE}$ used across all workloads captures 95.4% to 100% of the performance of any workload's optimal $K_{THROTTLE}$ value. This static value is determined experimentally and explored in more detail in Section 5.5. Like the LLS cutoff test, the lost-locality detected score can take several cycles to update and does not impact the critical path.

4. Experimental Methodology

We model the cache-conscious scheduling mechanisms as described in Section 3 in GPGPU-Sim [4] (version 3.1.0) using the configuration in Table 1. The Belady-optimal replacement policy [8], which chooses the line which is re-referenced furthest in the future for eviction, is evaluated using a custom *stand alone GPGPU-Sim cache simulator* (SAGCS). SAGCS is a trace based cache simulator that takes GPGPU-Sim cache access traces as input. Since SAGCS is not a performance simulator and only provides cache information, we do not present IPC results for the Belady-optimal replacement policy. To validate SAGCS, we verified the miss rate for LRU replacement using SAGCS and found that it was within 0.1% of the LRU miss rate reported using GPGPU-Sim. This small difference is a result of variability in the GPGPU-Sim memory system that SAGCS does not take into account.

We perform our evaluation using the high-performance computing GPU-enabled server workloads listed in Table 2 from Rodinia [9], Hetherington et al. [16] and Bakhoda et al. [4]. While

Table 1: GPGPU-Sim Configuration

# Compute Units	30
Wavefront Size	32
SIMD Pipeline Width	8
Number of Threads / Core	1024
Number of Registers / Core	16384
Shared Memory / Core	16KB
Constant Cache Size / Core	8KB
Texture Cache Size / Core	32KB, 64B line, 16-way assoc.
Number of Memory Channels	8
L1 Data Cache	32KB, 128B line, 8-way assoc. LRU
L2 Unified Cache	128k/Memory Channel, 128B line, 8-way assoc. LRU
Compute Core Clock	1300 MHz
Interconnect Clock	650 MHz
Memory Clock	800 MHz
DRAM request queue capacity	32
Memory Controller	out of order (FR-FCFS)
Branch Divergence Method	PDOM [12]
GDDR3 Memory Timing	t_{CL}=10 t_{RP}=10 t_{RC}=35 t_{RAS}=25 t_{RCD}=12 t_{RRD}=8
Memory Channel BW	8 (Bytes/Cycle)

Table 2: GPU Compute Benchmarks (CUDA and OpenCL)

Highly Cache Sensitive (HCS)			
Name	**Abbr.**	**Name**	**Abbr.**
BFS Graph Traversal [9]	BFS	Kmeans [9]	KMN
Memcached [16]	MEMC	Garbage Collection [5, 36]	GC
Moderately Cache Sensitive (MCS)			
Name	**Abbr.**	**Name**	**Abbr.**
Weather Prediction [9]	WP	Streamcluster [9]	STMCL
Single Source Shortest Path [4]	SSSP	CFD Solver [9]	CFD
Cache Insensitive (CI)			
Name	**Abbr.**	**Name**	**Abbr.**
Needleman-Wunsch [9]	NDL	Back Propagation [4]	BACKP
Speckle Red. Anisotropic Diff. [9]	SRAD	LU Decomposition [9]	LUD

the regularity of the HPC applications makes them particularly well suited for the GPU, they represent only one segment of the overall computing market [18] [17].

In addition to the cache-sensitive benchmarks introduced earlier, we also evaluate against a number of *cache-insensitive* (CI) benchmarks to ensure CCWS does not have a detrimental effect.

To make use of a larger input, the KMN benchmark was slightly modified to use global memory in place of both texture and constant memory.

All of our benchmarks run from beginning to end which takes between 14 million and 1 billion instructions.

4.1. GPU-enabled server workloads

This work uses two GPU-enabled server workloads. These benchmarks were ported to OpenCL from existing CPU implementations. They represent highly parallel code with irregular memory access patterns whose performance could be improved by running on the GPU.

Memcached-GPU (MEMC) Memcached is a key-value store and retrieval system. Memcached-GPU is described in detail by Hetherington et al. [16]. The application is stimulated with a representative portion of the Wikipedia access trace collected by Urdaneta et al. [37].

Tracing Garbage Collector (GC) Garbage collection is an important aspect of many server applications. Languages such as Java use system-controlled garbage collection to manage resources [2]. A version of the tracing mark-and-compact garbage collector presented in Barabash et al. [5] is created in OpenCL. The collector is stimulated with benchmarks provided by Spoonhower et al. [36].

5. Experimental Results

This section is structured as follows, Section 5.1 presents the performance of SWL, CCWS, other related wavefront schedulers and

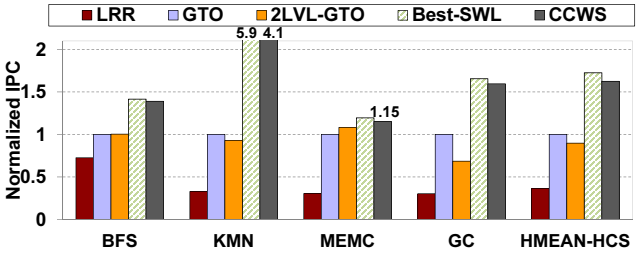


Figure 7: Performance of various schedulers and replacement policies for the highly cache-sensitive benchmarks. Normalized to the GTO scheduler.

the Belady-optimal replacement policy using the system presented in Section 4. The results for CCWS presented in Section 5.1 represent a design point that maximizes performance increase over area increase. The remainder of this section is devoted to exploring the sensitivity of our design and explaining the behaviour of our benchmarks.

5.1. Performance

The data in Figures 7, 8, 9 and 10 is collected using GPGPU-Sim for the following mechanisms:

LRR Loose round-robin scheduling. Wavefronts are prioritized for scheduling in round-robin order. However, if a wavefront cannot issue during its turn, the next wavefront in round-robin order is given the chance to issue.

GTO A greedy-then-oldest scheduler. GTO runs a single wavefront until it stalls then picks the oldest ready wavefront. The age of a wavefront is determined by the time it is assigned to the core. For wavefronts that are assigned to a core at the same time (i.e. they are in the same workgroup), wavefronts with the smallest threads IDs are prioritized. Other greedy schemes (such as greedy-then-round-robin and oldest-first) were implemented and GTO scheduling had the best results.

2LVL-GTO A two-level scheduler similar to that described by Narasiman et al. [31]. Their scheme subdivides wavefronts waiting to be scheduled on a core into *fetch groups* (FG) and executes from only one fetch group until all wavefronts in that group are stalled. Narasiman et al. used a fetch group size of 8 and a round-robin scheduling policy to select among wavefronts in a fetch group as well as among fetch groups. To provide a fair comparison against their scheduling technique in our simulator and on our workloads, all fetch group sizes were swept. We also explored alternate scheduling policies for intra-FG and inter-FG selection. We found using GTO for both of these policies was better than the algorithm they employed. A fetch group size of 2 using GTO for both intra-FG and inter-FG selection provides the best performance on our workloads and is what we present in our results. This disparity in optimal configuration can be explained by the nature of our workloads and our baseline architecture. Their core pipeline allows only one instruction from a given wavefront to be executing at a time. This means that a wavefront must wait for its previously issued instruction to complete execution before the wavefront can issue another instruction. This is different from our baseline which prevents a fetched instruction from issuing if the scoreboard detects a data hazard.

Best-SWL Static Wavefront Limiting as described in Section 3.2. All possible limitation values (32 to 1) were run and the best performing case is picked. The GTO policy is used to select between wavefronts. The wavefront value used for each benchmark is shown in Table 3.

CCWS Cache-Conscious Wavefront Scheduling described in Section 3.3 with the configuration parameters listed in Table 3. GTO wavefront prioritization logic is used.

The data for Belady-optimal replacement *misses per thousand instructions* (MPKI) presented in Figures 8 and 10 is generated with SAGCS:

<scheduler>-BEL Miss miss rate reported by SAGCS when using the Belady-optimal replacement policy. SAGCS is stimulated with L1D access streams generated by using GPGPU-Sim running the specified <scheduler>. Since SAGCS only reports misses, MPKI is calculated from the GPGPU-Sim instruction count.

Figure 7 shows that CCWS achieves a harmonic mean 63% performance improvement over a simple greedy wavefront scheduler and 72% over the 2LVL-GTO scheduler on HCS benchmarks. The GTO scheduler performs well because prioritizing older wavefronts allows them to capture intra-wavefront locality by giving them more exclusive access to the L1 data cache. The 2LVL-GTO scheduler performs slightly worse than the GTO scheduler because the 2LVL-GTO scheduler will not prioritize the oldest wavefronts every cycle. 2LVL-GTO only attempts to schedule the oldest FG intermittently, once the current FG is completely stalled. This allows loads from younger wavefronts, which would not have been prioritized in the GTO scheduler, to be injected into the access stream, causing older wavefront's data to be evicted.

CCWS and SWL provide further benefit over the GTO scheduler because these programs have a number of uncoalesced loads, touching many cache lines in relatively few memory instructions. Therefore, even restricting to just the oldest wavefronts still touches too much data to be contained by the L1D. The GTO, 2LVL-GTO, Best-SWL and CCWS schedulers see a greater disparity in the completion time of workgroups running on the same core compared to the LRR scheduler. Since all our workloads are homogeneous (at any given time only workgroups from one kernel launch will be assigned to each core) and involve synchronous kernel launches, the relative completion time of workgroups is not an issue. All that matters is when the whole kernel finishes. Moreover, the highly cache-sensitive workloads we study do not use any workgroup or global synchronization within a kernel launch, therefore older wavefronts are never stalled waiting for younger ones to complete.

Figure 7 also highlights the importance of scheduler choice even among simple schedulers like GTO and LRR. The LRR scheduler suffers from a 64% slowdown compared to GTO. Scheduling wavefronts with a lot of intra-wavefront locality in a RR fashion strides through too much data to be contained in the L1D. Best-SWL is able to slightly outperform CCWS on all the benchmarks. The CCWS configuration used here has been optimized to provide the highest performance per unit area. If the VTA cache is doubled in size, CCWS is able to slightly outperform Best-SWL on some workloads. CCWS is not able to consistently outperform Best-SWL because there is a start-up cost associated with detecting the loss of locality and a cool-down cost to back off the wavefront throttling. Adding to that, the execution time of these kernels is dominated by the code section that benefits from wavefront limiting. Therefore, providing the static scheme with oracle knowledge (through profiling) gives it an advantage over the adaptive CCWS scheme. Section 5.6 examines the shortcomings of the SWL under different run-time condi-

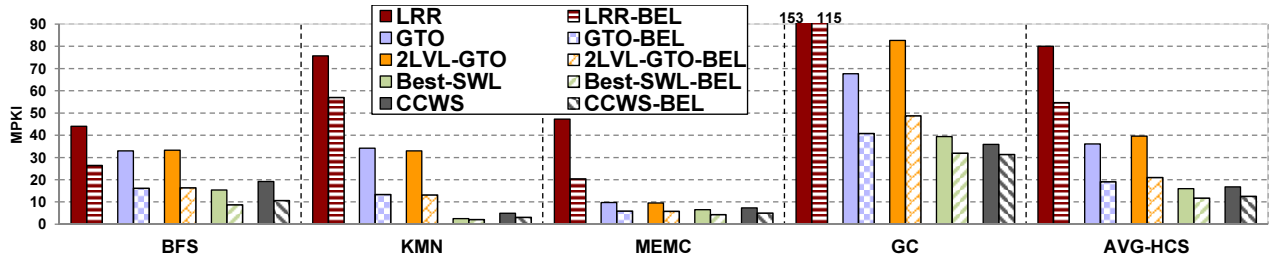


Figure 8: MPKI of various schedulers and replacement policies for the highly cache-sensitive benchmarks.

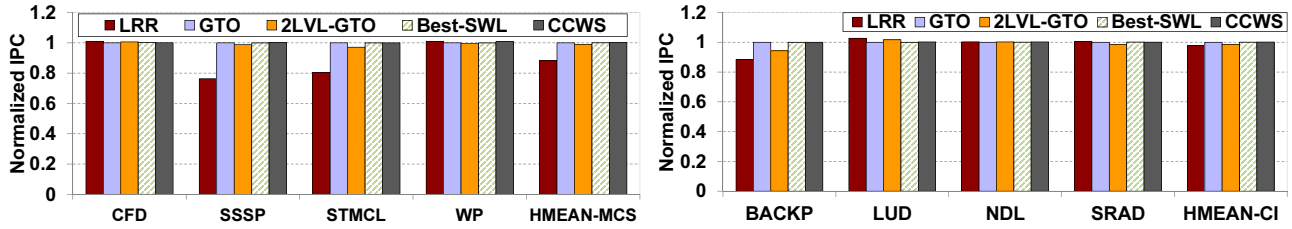


Figure 9: Performance of various schedulers and replacement policies for moderately cache-sensitive (left) and cache-insensitive (right) benchmarks. Normalized to the GTO scheduler.

tions.

Although not plotted here, it is worth mentioning the performance of the 2LVL-LRR scheduling configuration evaluated by Narasiman et al. On the HCS benchmarks the 2LVL-LRR scheduler is a harmonic mean 43% faster than the LRR scheduler, however this is still 47% slower than the GTO scheduler. Performing intra-FG and inter-FG scheduling in a round-robin fashion destroys the intra-wavefront locality of older wavefronts that is captured by the GTO scheduler. However, in comparison to the LRR scheduler, which cycles through 32 wavefronts in a round-robin fashion, cycling through smaller FG sized pools (each fetch group has 8 wavefronts in their configuration) will thrash the L1 data cache less.

Figure 8 illustrates that the reason for the performance advantage of the wavefront limiting schemes is a sharp decline in the number of L1D misses. This figure highlights the fact that no cache replacement policy can make up for a poor choice in wavefront scheduler, as even an oracle Belady-optimal policy on the LRR access stream is outperformed by all the schedulers. The insight here is that even optimal replacement cannot compensate for an access stream that strides through too much data, at least for the relatively low associativity L1 data caches we evaluated.. Furthermore, the miss rate of CCWS outperforms both GTO-BEL and 2LVL-GTO-BEL. This data suggests L1D cache hit rates are more sensitive to wavefront scheduling policy than cache replacement policy.

Figures 9 and 10 present the performance and MPKI of our MCS and CI benchmarks. The harmonic mean performance improvement of CCWS across both the highly and moderately cache-sensitive (HCS and MCS) benchmarks is 24%. In the majority of the MCS and CI workloads, the choice of wavefront scheduler makes little difference and CCWS does not degrade performance. There is no degradation because the MPKI for these benchmarks is much lower than the HCS applications, so there are few VTA hits compared to instructions issued. As a result the lost-locality detected score as defined by Equation (1) stays low and the thread throttling mechanism does not take effect.

5.2. Detailed Breakdown of Inter- and Intra-Wavefront Locality

Figure 11 breaks down L1D accesses into misses, inter-wavefront hits and intra-wavefront hits for all the schedulers evaluated in Section 5.1 on our HCS benchmarks. In addition, it quantifies the portion of intra-wavefront hits that are a result of intra-thread locality. It illustrates that the decrease in cache misses using CCWS and Best-SWL comes chiefly from an increase in intra-wavefront hits. Moreover, the bulk of these hits are a result of intra-thread locality. The exception to this rule is BFS, where 30% of intra-wavefront hits come from inter-thread locality and we see a 23% increase in inter-wavefront hits. An inspection of the code reveals that inter-thread sharing (which manifests itself as both intra-wavefront and inter-wavefront locality) occurs when nodes in the graph share neighbours. Limiting the number of wavefronts actively scheduled increases the hit rate of these accesses because it limits the amount of non-shared data in the cache, increasing the chance that these shared accesses hit.

Table 3: Configurations for Best-SWL (wavefronts actively scheduled) and CCWS variables used for performance data.

Best-SWL		CCWS Config	
Benchmark	Wavefronts Actively Scheduled	Name	Value
BFS	5	$K_{THROTTLE}$	8
KMN	4	Wavefront Base Score	100
MEMC	7	VTA Tag array	8-way
GC	4		16 entries per wavefront
All Others	32		(512 total entries)

Figure 12 explores the access stream of all the cache-sensitive benchmarks using SAGCS and an unbounded L1D. It shows that with the exception of SSSP, the MCS benchmarks have significantly less locality in the access stream. The larger amount of intra-wavefront locality in SSSP is consistent with the significant performance improvement we observe for CCWS at smaller cache sizes when the working set of all the threads does not fit in the L1D cache (see Figure 14).

5.3. Sensitivity to Victim Tag Array Size

Figure 13 shows the effect of varying the VTA size on performance. With a larger victim tag array the system is able to detect lost intra-wavefront locality occurring at further access distances. Increasing the size of the VTA keeps data with intra-wavefront locality in the VTA longer and causes wavefront limiting to be appropriately applied. However, if the VTA size is increased too much, the lost-locality detector's time sensitivity is diminished. The VTA will contain tags from data that was evicted from the L1 data cache so

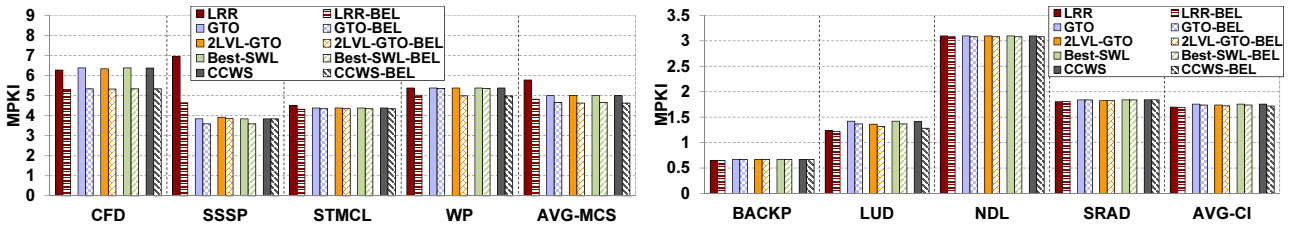


Figure 10: MPKI of various schedulers and replacement policies for moderately cache-sensitive (left) and cache-insensitive benchmarks (right).

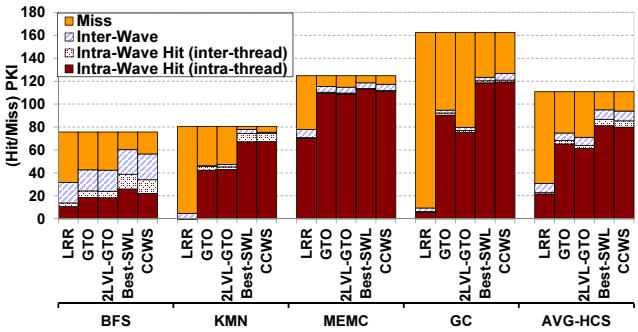


Figure 11: Breakdown of L1D misses, intra-wavefront locality hits (broken into intra-thread and inter-thread) and inter-wavefront locality hits per thousand instructions for highly cache-sensitive benchmarks. The configuration from Section 5.1 is used.

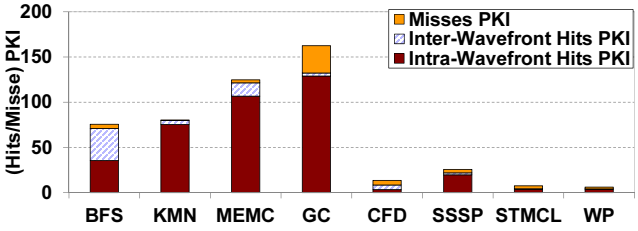


Figure 12: Breakdown of L1D misses, intra-wavefront locality Hits and inter-wavefront locality PKI using an unbounded L1 cache with 128 byte cache lines.

long ago that it would have been difficult to capture with changes to the scheduling policy. For example, at the 512 entry design point, each wavefront has a VTA that can track as much data as the entire L1D. In this configuration, a wavefront would need exclusive access to the L1 data cache to prevent all the detected loss of locality. The increase in detected lost-locality results in excessive wavefront constraining on some workloads. Based on this data, the best-performing configuration with 16 entries per wavefront is selected.

5.4. Sensitivity to Cache Size

Figure 14 shows the sensitivity of CCWS to the L1D size. As the cache size decreases, CCWS has a greater performance improvement relative to the GTO scheduler. This is because at small cache sizes it is even more desirable to limit multithreading to reduce cache footprint. In fact SSSP, which showed no performance gain at 32k shows a 35% speedup when the L1 cache is reduced to 8k. This is because SSSP has significant intra-wavefront locality but its footprint is small enough that it is contained by a 32k L1D. As the cache size increases, the effect of CCWS dwindles relative to the GTO scheduler because the working set of most wavefronts fit in a larger cache. At a large enough cache size, the choice of wavefront scheduler makes little difference.

At 128k per L1D, CCWS shows little benefit over the GTO scheduler. This is because the input to these benchmarks is small enough that 128k captures most of the intra-wavefront locality. Since we are collecting results on a performance simulator that runs several orders of magnitude slower than a real device, the input to our benchmarks is small enough that they finish in a reasonable amount of time. Figure 15 show the effect of increasing the size of the BFS input graph from the baseline 500k edges to 20M edges. As the input size increases, the performance of CCWS over the GTO scheduler also increases even at a 128k L1 cache size. We observe that simply increasing the capacity of the L1 cache only diminishes the performance impact of CCWS with small enough input sets. Hence, we believe CCWS will have an even greater impact on data sizes used in real workloads.

5.5. Sensitivity to $K_{THROTTLE}$ and Tuning for Power

Figure 16 shows the effect of varying $K_{THROTTLE}$ on L1D misses and performance. $K_{THROTTLE}$ is the constant used in Equation (1) to tune the score assigned to wavefronts when lost locality is detected (LLDS). At smaller $K_{THROTTLE}$ values, there is less throttling caused by the point system and more multithreading. At the smallest values of $K_{THROTTLE}$ multithreading is not constrained enough and performance suffers. As $K_{THROTTLE}$ increases, CCWS has a greater effect and the number of L1D misses falls across all the HCS benchmarks. In every HCS benchmark, except GC, performance peaks then falls as $K_{THROTTLE}$ increases. However, since a miss in the L1D cache can incur a significant power cost it may be desirable to use a higher $K_{THROTTLE}$ value to reduce L1D misses at the cost of some performance. For example, at $K_{THROTTLE} = 32$ there is an average 18% reduction in L1D misses over the chosen $K_{THROTTLE} = 8$ design point. $K_{THROTTLE} = 32$ still achieves a 46% performance improvement over the GTO scheduler.

Figure 16 also demonstrates that each benchmark has a different optimal $K_{THROTTLE}$ value. However, the difference in harmonic mean performance between choosing each benchmark's optimal $K_{THROTTLE}$ value and using a constant $K_{THROTTLE} = 8$ is $< 4\%$. For this reason, we do not pursue an online mechanism for determining the value of $K_{THROTTLE}$. If other HCS benchmarks have more variance in their intra-wavefront locality then such a system should be considered.

The value of $K_{THROTTLE}$ makes no difference in the CI benchmarks since there is little locality to lose and few VTA Hits are reported. In the MCS benchmarks there are relatively few L1D MPKI, which keeps the product of $K_{THROTTLE}$ and $\frac{VTAHits_{Total}}{InstIssued_{Total}}$ low. In the MCS benchmarks, CCWS performance matches GTO scheduler performance until $K_{THROTTLE} = 128$. At this point there is a harmonic mean 4% performance degradation due to excessive throttling. Since their performance is largely unchanged by the value of $K_{THROTTLE}$, we do not graph the MCS or CI benchmarks in Figure 16.

5.6. Static Wavefront Limiting Sensitivity

In Section 3 we noted that the optimal SWL limiting number was different for different benchmarks. We also indicated that this value

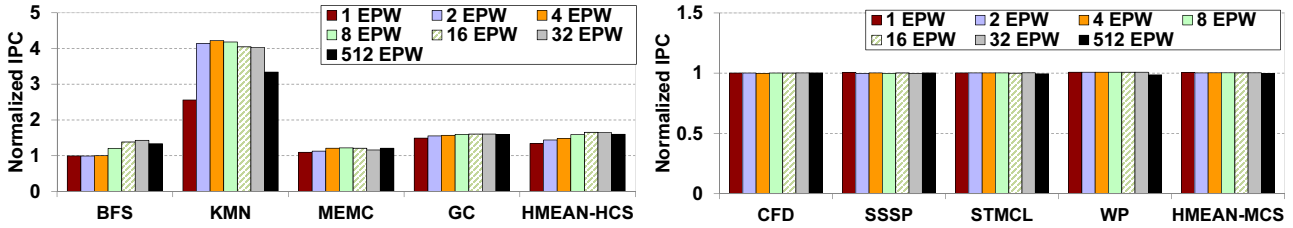


Figure 13: Performance of CCWS at various victim tag array sizes. Normalized to the GTO scheduler. EPW=Entries per Wavefront. EWP 1-4 are 1-4 set associative respectively. All other victim tag arrays are 8-way set associative.

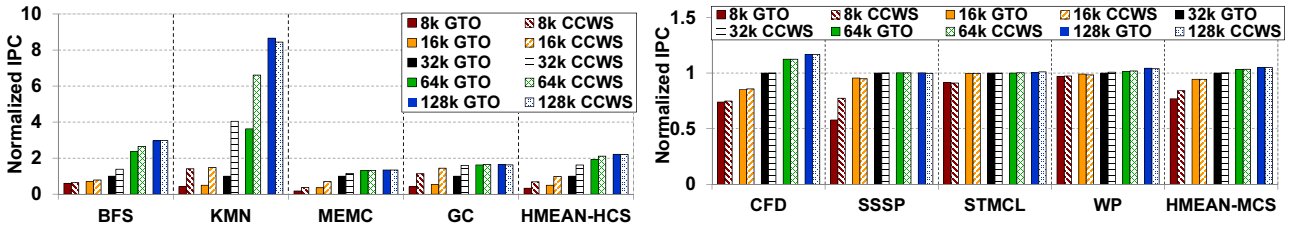


Figure 14: Performance of CCWS and GTO at various cache sizes. Normalized to the GTO scheduler with a 32k L1D. All caches are 8-way set associative. The VTA Size is 16 entries per wavefront for all instances of CCWS.

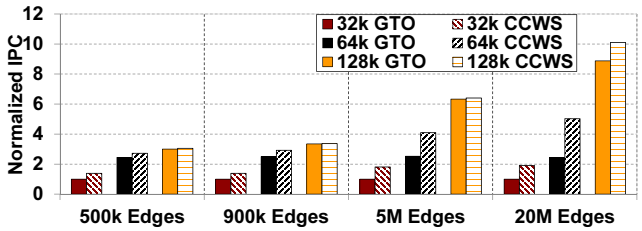


Figure 15: Performance of CCWS on BFS with different graph sizes when varying the L1D cache size and scheduler choice. Normalized to the GTO scheduler with a 32k L1D. The VTA size is 16 entries per wavefront for all instances of CCWS.

changes when running the same benchmark with different input sets. Figure 17 illustrates that peak performance for each of the HCS benchmarks occurs with different multithreading limits. This happens because each workload has a different working set and access stream characteristics. Furthermore, Figure 18 shows that for different input graphs on BFS, the values of the peak performance point are different. This variation happens because the working set size is input data dependent. Finding the optimal wavefront limiting number in SWL would require profiling of each instance of a particular workload, making the adaptive CCWS more practical.

SWL also suffers in programs that have phased execution. The larger and more diverse the application is, the less likely a single wavefront limiting value will capture peak performance. This type of phased behaviour is not abundant in the HCS workloads we study, but as the amount and type of code running on the GPU continues to grow so too will the importance of adaptive multithreading.

SWL is also sub-optimal in a multi-programmed GPU. If wavefronts from more than one type of kernel are assigned to the same compute unit, a per-kernel limiting number makes little sense. Even if there was no cache thrashing in either workload individually their combination may cause it to occur. CCWS will adapt to suit the needs of whatever wavefront combination is running on a compute unit and preserve their intra-wavefront locality. Since there will be no inter-wavefront locality among multi-programmed wavefronts, preservation of intra-wavefront locality becomes even more important.

5.7. Area Estimation

The major source of area overhead to support CCWS comes from the victim tag array. For the configuration used in Table 3 and a 48-bit virtual address space, we require 40 bits for each tag entry in our VTA. Using CACTI 5.3 [38], we estimate that this tag array would consume 0.026 mm^2 per core at 55nm or 0.78 mm^2 for the entire 30 core system. This represents 0.17% of GeForce GTX 285 area, which our system closely models with the exception that we also model data caches. There are a variety of smaller costs associated with our design that are difficult to quantify and as a result are not included in the above estimation. Adding an additional 5-bits to each L1D cache line for the WID costs 160 bytes per core. There are 32 lost-locality score values, each represented in 10 bits which are stored in a max heap. Also, there are two counter registers, one for the number of instructions issued and another for the total VTA hit signals. In addition, there is logic associated with the scoring system. Compared to the other logic in a compute unit, we do not expect this additional logic to be significant.

6. Related Work

This section summarizes and contrasts CCWS against prior scheduling and cache management work.

6.1. Thread Throttling to Improve Performance

Bakhoda et al. [4] present data for several GPU configurations, each with a different maximum number of workgroups (or CTAs) that can be concurrently assigned to a core. They observed that some workloads performed better when less workgroups were scheduled concurrently. The data they present is for a GPU without an L1 data cache, running a round-robin wavefront scheduling algorithm. They conclude that this increase in performance occurs because scheduling less concurrent workgroups on the GPU reduces contention for the interconnection network and DRAM memory system. In contrast, the goal of CCWS is use L1 data cache feedback to preserve locality by focusing on fine-grained, issue level wavefront scheduling, not coarse-grained workgroup assignment.

Guz et al. [15] use an analytical model to quantify the "performance valley" that exists when the number of threads sharing a

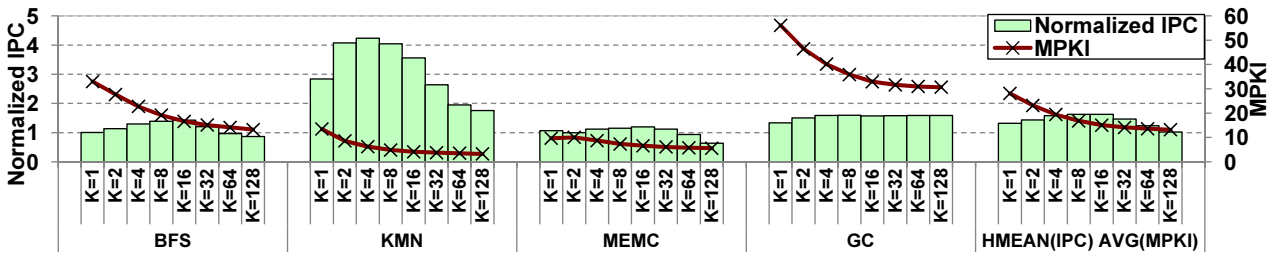


Figure 16: Performance of CCWS (normalized to the GTO scheduler) and MPKI of CCWS when varying $K_{THROTTLE}$.

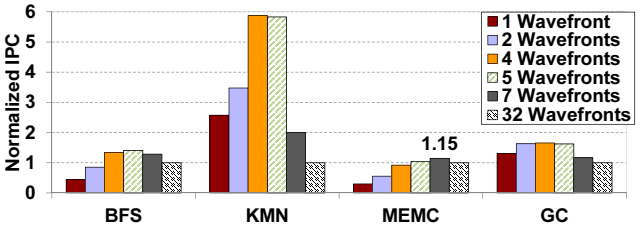


Figure 17: Performance of SWL at various multithreading limits. Normalized to 32 wavefronts.

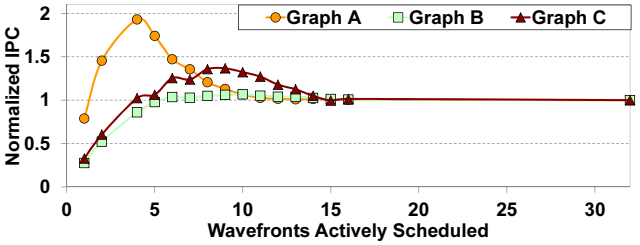


Figure 18: Performance of SWL with different multithreading values on BFS with different input graphs. Normalized to 32 wavefronts.

cache is increased. They show that increasing the thread count increases performance until the aggregate working set no longer fits in cache. Increasing threads beyond this point degrades performance until enough threads are present to hide the system's memory latency. In effect, CCWS dynamically detects when a workload has entered the machine's performance valley and scales down the number of threads sharing the cache to compensate.

Cheng et al. [10] introduce a thread throttling scheme to reduce memory latency in multi-threaded CPU systems. They propose an analytical model and memory task limit throttling mechanism to limit thread interference in the memory stage. Their model relies on a stream programming language which decomposes applications into separate tasks for computation and memory and their technique schedules tasks at this granularity.

6.2. Wavefront Scheduling Techniques

Lakshminarayana and Kim [25] explore numerous warp scheduling policies in the context of a GPU without hardware managed caches and show that, for applications that execute symmetric (balanced) dynamic instruction counts per warp, a fairness based warp and DRAM access scheduling policy improves performance. In contrast to our work, their study did not explore scheduling policies that improve performance by improving cache hit rates.

Fung et al. [12] explore the impact of wavefront scheduling policy on the effectiveness of their *Dynamic Warp Formation* (DWF) technique. DWF attempts to mitigate control flow divergence by dynamically creating new warps when scalar threads in the same wavefront take different paths on a branch instruction. They propose five schedulers and evaluate their effect on DWF. Fung and Aamodt [13] also propose three thread block prioritization schemes to compliment their *Thread Block Compaction* (TBC) technique. The prioritization schemes attempt to schedule threads within the same thread block (or workgroup) together. Their approach is similar to the two-level technique proposed by Narasiman et al. [31], except thread blocks are scheduled together instead of fetch groups. In contrast to both these works, CCWS explores the impact of scheduling on cache locality using existing control flow divergence mitigation techniques.

Gebhart and Johnson et al. [14] introduce the use of a two-level scheduler to improve energy efficiency. Experiments we run using their exact specification yielded mixed results. They note that the performance of their workloads increases less than 10% if a perfect cache is used instead of no cache at all. For this reason, they run all their simulations with a constant 400 cycle latency to global memory. As a result their scheme switches wavefronts out of the active pool whenever a compiler identified global or texture memory dependency is encountered. We find that obeying this constraint causes performance degradation because it does not take cache hits into account. However, if this demotion to the inactive pool is changed to just those operations causing a stall (i.e. those missing in cache) it's operation is similar to Narasiman's two level scheduler we evaluated in Section 5.

Meng et al. [29] introduce *Dynamic Warp Subdivision* (DWS) which splits wavefronts when some lanes hit in cache and some lanes do not. This scheme allows individual scalar threads that hit in cache to make progress even if some of their wavefront peers miss. DWS improves performance by allowing run-ahead threads to initiate their misses earlier and creates a pre-fetching effect for those left behind. DWS attempts to improve intra-wavefront locality by increasing the rate data is loaded into the cache. In contrast, CCWS attempts to load data from less threads at the same time to reduce thrashing.

Narasiman et al. [31] detail a two-level wavefront scheduler similar to that proposed in [14]. Their work focuses on improving performance by allowing groups of threads to reach the same long latency operation at different times. This helps ensure cache and row-buffer locality within a fetch group is maintained and the system is able to hide long latency operations by switching between fetch groups. In contrast, our work focuses on improving performance by adaptively limiting the amount of multithreading the system can maintain based on how much intra-wavefront locality is being lost.

6.3. Improving Cache Efficiency

There is a body of work attempting to increase cache hit rate by improving the replacement policy (e.g., [21] [34] among many others). All these attempt to exploit different heuristics of program behavior to predict a block's re-reference interval and mirror the Belady-optimal [8] policy as closely as possible. While CCWS also at-

tempts to maximize cache efficiency, it does so by shortening the re-reference interval rather than by predicting it. CCWS has to balance the shortening of the re-reference interval by limiting the number of eligible wavefronts while still maintaining sufficient multithreading to cover most of the memory and operation latencies. Other schemes attempt to manage interference among heterogeneous workloads [35, 19] but each thread in our workload has roughly similar characteristics. Recent work has explored the use of prefetching on GPUs [26]. However, prefetching cannot improve performance when an application is bandwidth limited whereas CCWS can help in such cases by reducing off-chip traffic.

Beckmann et al. [7] use victim tag information to detect locality lost due to excessive replication in the cache hierarchy and adapt the replication level accordingly. The LLD in CCWS differs from their technique in that it subdivides the victim tag array by wavefront ID and makes use of the this information to influence thread scheduling.

Concurrent to our work, Jaleel et al. [20] propose the CRUISE scheme which uses LLC utility information to make high level scheduling decisions in multi-programmed CMPs. Our work focuses on the first level cache in a massively multi-threaded environment and is applied at a much finer grain. Scheduling decisions made by CRUISE tie programs to cores, where CCWS makes issue level decisions on which bundle of threads should enter the execution pipeline next.

Agrawal et el. [3] present theoretical cache miss limits when scheduling streaming applications represented as directed graphs on uniprocessors. Their work shows that scheduling the graph by selecting partitions comes within a constant factor of the optimal scheduler when heuristics such as working set and data usage rates are known in advance.

Jia et al. [22] characterize GPU L1 cache locality in a current NVIDIA Tesla GPU and present a compile time algorithm to determine which loads should be cached by the L1D. In contrast to our work, which focuses on locality between different dynamic load instructions, their algorithm and taxonomy focus on locality across different threads in a single static instruction. Moreover, since their analysis is done at compile time they are unable to capture any locality with input data dependence.

7. Conclusion

This work introduces a new classification of locality for GPUs. We quantify the caching and performance effects of both intra- and inter-wavefront locality for workloads in massively multi-threaded environments.

To exploit the observation that intra-wavefront locality is of greatest importance on highly cache-sensitive workloads, this work introduces Cache-Conscious Wavefront Scheduling. CCWS is a novel technique to capitalize on the performance benefit of limiting the number of actively-scheduled wavefronts, thereby limiting L1 data cache thrashing and preserving intra-wavefront locality. Our simulated evaluation shows this technique results in a harmonic mean 63% improvement in throughput on highly cache-sensitive benchmarks, without impacting the performance of cache-insensitive workloads.

We demonstrate that on massively multi-threaded systems, optimizing the low level thread scheduler is of more importance than attempting to improve the cache replacement policy. Furthermore, any work evaluating cache replacement on massively multi-threaded systems should do so in the presence of an intelligent wavefront scheduler.

As more diverse applications are created to exploit irregular parallelism and the number of threads sharing a cache continues to increase on both GPUs and CMPs, so too will the importance of intelligent HW thread scheduling policies, like CCWS.

Acknowledgments

The authors would like to thank Wilson Fung, Hadi Jooybar, Inderpreet Singh, Tayler Hetherington, Ali Bakhoda, the reviewers and our shepherd Yale N. Patt for their insightful feedback. We also thank Rimon Tadros for his work on the garbage collector benchmark. This research was funded in part by a grant from Advanced Micro Devices Inc.

References

[1] T. M. Aamodt et al., *GPGPU-Sim 3.x Manual*, http://gpgpu-sim.org/manual/index.php5/GPGPU-Sim_3.x_Manual, University of British Columbia, 2012.

[2] O. Agesen, D. Detlefs, and J. E. Moss, "Garbage Collection and Local Variable Type-Precision and Liveness in Java Virtual Machines," in *Proc. of Prog. Lang. Design and Implementation (PLDI 1998)*, pp. 269–279.

[3] K. Agrawal, J. T. Fineman, J. Krage, C. E. Leiserson, and S. Toledo, "Cache-Conscious Scheduling of Streaming Applications," in *Proc. of Symp. on Parallelism in Algorithms and Architectures (SPAA 2012)*, pp. 236–245.

[4] A. Bakhoda, G. Yuan, W. Fung, H. Wong, and T. Aamodt, "Analyzing CUDA Workloads Using a Detailed GPU Simulator," in *Proc. of Int'l Symp. on Performance Analysis of Systems and Software (ISPASS 2009)*, pp. 163–174.

[5] K. Barabash and E. Petrank, "Tracing Garbage Collection on Highly Parallel Platforms," in *Proc. of Int'l Symp. on Memory Management (ISMM 2010)*, pp. 1–10.

[6] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. V. der Vorst, *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition*. SIAM, 1994.

[7] B. M. Beckmann, M. R. Marty, and D. A. Wood, "ASR: Adaptive Selective Replication for CMP Caches," in *Proc. of Int'l Symp. on Microarchitecture (MICRO 39)*, 2006, pp. 443–454.

[8] L. A. Belady, "A Study of Replacement Algorithms for a Virtual-Storage Computer," *IBM Systems Journal*, vol. 5, no. 2, pp. 78 –101, 1966.

[9] S. Che, M. Boyer, J. Meng, D. Tarjan, J. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A Benchmark Suite for Heterogeneous Computing," in *Proc. of Int'l Symp. on Workload Characterization (IISWC 2009)*, pp. 44–54.

[10] H.-Y. Cheng, C.-H. Lin, J. Li, and C.-L. Yang, "Memory Latency Reduction via Thread Throttling," in *Proc. of Int'l Symp. on Microarchitecture (MICRO 43)*, 2010, pp. 53–64.

[11] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger, "Dark Silicon and the End of Multicore Scaling," in *Proc. of Int'l Symp. on Computer Architecture (ISCA 2011)*, pp. 365–376.

[12] W. W. L. Fung, I. Sham, G. Yuan, and T. M. Aamodt, "Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow," in *Proc. of Int'l Symp. on Microarchitecture (MICRO 40)*, 2007, pp. 407–420.

[13] W. Fung and T. Aamodt, "Thread Block Compaction for Efficient SIMT Control Flow," in *Proc. of Int'l Symp. on High Performance Computer Architecture (HPCA 2011)*, pp. 25 –36.

[14] M. Gebhart, D. R. Johnson, D. Tarjan, S. W. Keckler, W. J. Dally, E. Lindholm, and K. Skadron, "Energy-Efficient Mechanisms for Managing Thread Context in Throughput Processors," in *Proc. of Int'l Symp. on Computer Architecture (ISCA 2011)*, pp. 235–246.

[15] Z. Guz, E. Bolotin, I. Keidar, A. Kolodny, A. Mendelson, and U. Weiser, "Many-Core vs. Many-Thread Machines: Stay Away From the Valley," *Computer Architecture Letters*, vol. 8, no. 1, pp. 25 –28, jan. 2009.

[16] T. H. Hetherington, T. G. Rogers, L. Hsu, M. O'Connor, and T. M. Aamodt, "Characterizing and Evaluating a Key-Value Store Application on Heterogeneous CPU-GPU Systems," in *Proc. of Int'l Symp. on Performance Analysis of Systems and Software (ISPASS 2012)*, pp. 88 –98.

[17] IDC, "HPC Server Market Declined 11.6% in 2009, Return to Growth Expected in 2010," Mar 2010.

[18] IDC, "Worldwide Server Market Rebounds Sharply in Fourth Quarter as Demand for Blades and x86 Systems Leads the Way," Feb 2010.

[19] A. Jaleel, W. Hasenplaugh, M. Qureshi, J. Sebot, S. Steely, Jr., and J. Emer, "Adaptive Insertion Policies for Managing Shared Caches," in *Proc. of Int'l Conf. on Parallel Architecture and Compiler Techniques (PACT 2008)*, pp. 208–219.

[20] A. Jaleel, H. H. Najaf-abadi, S. Subramaniam, S. C. Steely, and J. Emer, "CRUISE: Cache Replacement and Utility-Aware Scheduling," in *Proc. of Int'l Conf. on Architectural Support for Prog. Lang. and Operating Systems (ASPLOS 2012)*, pp. 249–260.

[21] A. Jaleel, K. B. Theobald, S. C. Steely, Jr., and J. Emer, "High Performance Cache Replacement Using Re-Reference Interval Prediction (RRIP)," in *Proc. of Int'l Symp. on Computer Architecture (ISCA 2010)*, pp. 60–71.

[22] W. Jia, K. A. Shaw, and M. Martonosi, "Characterizing and Improving the use of Demand-Fetched Caches in GPUs," in *Proc. of Int'l Conf. on Supercomputing (ICS 2012)*, pp. 15–24.

[23] N. P. Jouppi, "Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers," in *Proc. of Int'l Symp. on Computer Architecture (ISCA 1990)*, pp. 364–373.

[24] Khronos Group, "OpenCL," http://www.khronos.org/opencl/.

[25] N. B. Lakshminarayana and H. Kim, "Effect of Instruction Fetch and Memory Scheduling on GPU Performance," in *Workshop on Language, Compiler, and Architecture Support for GPGPU*, 2010.

[26] J. Lee, N. B. Lakshminarayana, H. Kim, and R. Vuduc, "Many-Thread Aware Prefetching Mechanisms for GPGPU Applications," in *Proc. of Int'l Symp. on Microarchitecture (MICRO 43)*, 2010, pp. 213–224.

[27] S. Lee, S.-J. Min, and R. Eigenmann, "OpenMP to GPGPU: A Compiler Framework for Automatic Translation and Optimization," in *Proc. of Symp. on Principles and Practice of Parallel Programming (PPoPP 2009)*, pp. 101–110.

[28] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "NVIDIA Tesla: A Unified Graphics and Computing Architecture," *Micro, IEEE*, vol. 28, no. 2, pp. 39–55, March-April 2008.

[29] J. Meng, D. Tarjan, and K. Skadron, "Dynamic Warp Subdivision for Integrated Branch and Memory Divergence Tolerance," in *Proc. of Int'l Symp. on Computer Architecture (ISCA 2010)*, pp. 235–246.

[30] D. Merrill, M. Garland, and A. Grimshaw, "Scalable GPU Graph Traversal," in *Proc. of Symp. on Principles and Practice of Parallel Programming (PPoPP 2012)*, pp. 117–128.

[31] V. Narasiman, M. Shebanow, C. J. Lee, R. Miftakhutdinov, O. Mutlu, and Y. N. Patt, "Improving GPU Performance via Large Warps and Two-Level Warp Scheduling," in *Proc. of Int'l Symp. on Microarchitecture (MICRO 44)*, 2011, pp. 308–317.

[32] *NVIDIA's Next Generation CUDA Compute Architecture: Fermi*, http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf , NVIDIA, 2009.

[33] *NVIDIA CUDA C Programming Guide v4.2*, http://developer.nvidia.com/nvidia-gpu-computing-documentation/, NVIDIA Corp., 2012.

[34] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer, "Adaptive Insertion Policies for High Performance Caching," in *Proc. of Int'l Symp. on Computer Architecture (ISCA 2007)*, pp. 381–391.

[35] M. K. Qureshi and Y. N. Patt, "Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches," in *Proc. of Int'l Symp. on Microarchitecture (MICRO 39)*, 2006, pp. 423–432.

[36] D. Spoonhower, G. Blelloch, and R. Harper, "Using Page Residency to Balance Tradeoffs in Tracing Garbage Collection," in *Proc. of Int'l Conf. on Virtual Execution Environments (VEE 2005)*, 2005, pp. 57–67.

[37] G. Urdaneta, G. Pierre, and M. van Steen, "Wikipedia Workload Analysis for Decentralized Hosting," *Elsevier Computer Networks*, vol. 53, no. 11, pp. 1830–1845, 2009.

[38] S. Wilton and N. Jouppi, "CACTI: An Enhanced Cache Access and Cycle Time Model," *Solid-State Circuits, IEEE Journal of*, vol. 31, no. 5, pp. 677–688, May 1996.

[39] S. Zhuravlev, S. Blagodurov, and A. Fedorova, "Addressing Shared Resource Contention in Multicore Processors via Scheduling," in *Proc. of Int'l Conf. on Architecture Support for Prog. Lang. and Operating Systems (ASPLOS 2010)*, pp. 129–142.

2012 IEEE/ACM 45th Annual International Symposium on Microarchitecture

Libra: Tailoring SIMD Execution using Heterogeneous Hardware and Dynamic Configurability

Yongjun Park Jason Jong Kyu Park Hyunchul Park* Scott Mahlke

Advanced Computer Architecture Laboratory
University of Michigan
Ann Arbor, MI, USA
{yjunpark, jasonjk, parkhc, mahlke}@umich.edu

Abstract

Mobile computing as exemplified by the smart phone has become an integral part of our daily lives. The next generation of these devices will be driven by providing an even richer user experience and compelling capabilities: higher definition multimedia, 3D graphics, augmented reality, games, and voice interfaces. To address these goals, the core computing capabilities of the smart phone must be scaled. However, the energy budgets are increasing at a much lower rate, requiring fundamental improvements in computing efficiency. SIMD accelerators offer the combination of high performance and low energy consumption through low control and interconnect overhead. However, SIMD accelerators are not a panacea. Many applications lack sufficient vector parallelism to effectively utilize a large number of SIMD lanes. Further, the use of symmetric hardware lanes leads to low utilization and high static power dissipation as SIMD width is scaled. To address these inefficiencies, this paper focuses on breaking two traditional rules of SIMD processing: homogeneity and static configuration. The Libra *accelerator increases SIMD utility by blurring the divide between vector and instruction parallelism to support efficient execution of a wider range of loops, and it increases hardware utilization through the use of heterogeneous hardware across the SIMD lanes. Experimental results show that the 32-lane Libra outperforms traditional SIMD accelerators by an average of 1.58x performance improvement due to higher loop coverage with 29% less energy consumption through heterogeneous hardware.*

1. Introduction

The mobile devices market, including cell phones, netbooks, and personal digital assistants, is one of the most highly competitive businesses. The computing platforms that go into these devices must provide ever increasing performance capabilities while maintaining low energy consumption in order to support advanced multimedia and signal processing applications. Application-specific integrated circuits (ASICs) are the most common solutions for meeting these requirements, performing the most compute-intensive kernels in a high performance but energy-efficient manner. However, several features push designers to a more flexible and programmable solution: supporting multiple applications or variations of applications, providing faster time-to-market, and enabling algorithmic changes after the hardware is constructed.

Processors that exploit instruction-level parallelism (ILP) provide the highest degree of computing flexibility. Modern smart phones employ a one GHz dual-issue superscalar ARM as an application processor. Higher performance digital signal processors are also available such as the 8-issue TI C6x. However, ILP processors have scalability limits including many-ported register files (RFs) and complex interconnects. Alternately, single-instruction multiple-data (SIMD) accelerators provide high efficiency because of their regular structure, ability to scale lanes, and low control logic overhead. They have long been used in the desktop space for high performance multimedia and graphics functionality. But, their combination of scalable performance, energy efficiency, and programmability make them ideal for mobile systems [24, 9, 15, 27].

In order to fully utilize the SIMD hardware, it is necessary for the programmer or compiler to extract sufficient data-level parallelism (DLP). Automatic loop vectorization is available in a variety of commercial compilers including offerings from Intel, IBM, and PGI. Classic scientific computing (regular structure, large trip count loops, and few data dependences) are naturally well-matched to SIMD accelerators. But, in many respects, the mobile terminal has become a general-purpose computer. Thus, like the desktop, only a small percentage of mobile applications look like classic scientific computing. The computation is not dominated by simple vectorizable loops, but by loops containing significant numbers of control and data dependences to handle the complexity of modern multimedia standards. As a result, applications have varying amounts of vector parallelism ranging from none to some to large amounts. The net effect is that SIMD hardware goes unused for a large fraction of application execution and thus cannot be counted on to provide significant performance gains.

A second but inter-related problem with SIMD computing is low hardware utilization even when vector loops are executed. The use of homogeneous hardware (e.g, identical lanes) is one of the best advantages of SIMD datapaths by reducing design cost and complexity. But, the utilization of the most complex components of a SIMD lane is often disproportionally lower than the simple components. For example, the H.264 video decoding application is dominated by simple integer operations (adds, subtracts, shifts) and an average of only 2.2% and 1.3% of the total dynamic instructions are multiplies and divides [8]. This is not an outlying data point, most multimedia and visual computing applications have small fractions of multiply, divide and other expensive operators. For 128-bit SIMD (4 lanes), such utilization rates may not matter, but as SIMD widths are scaled to increase performance to 1024 bits (32 lanes) or more, the problem becomes serious due to poor area utilization and high static power dissipation.

To attack these problems, we propose a customizable SIMD accelerator that is capable of tailoring its execution strategy to the running application, referred to as the *Libra*. Libra employs two key concepts, *heterogeneity* and *dynamic configurability*, to achieve broader applicability and better energy efficiency than traditional SIMD accelerators. Heterogeneity allows lanes to have different functionalities and better match functional capabilities with expected operator

*Currently with Programming Systems Lab, Intel Labs, Santa Clara, CA

1072-4451/12 $26.00 © 2012 IEEE
DOI 10.1109/MICRO.2012.17

IEEE computer society

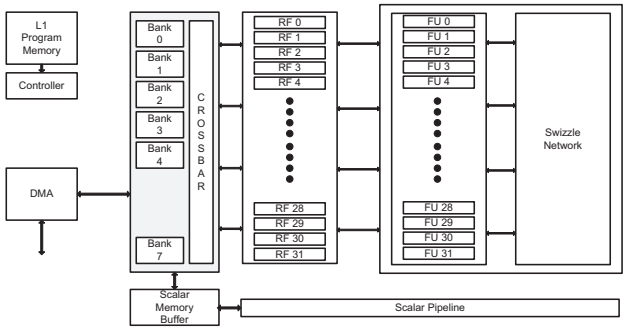


Figure 1: A traditional 32-lane SIMD accelerator.

distributions. Dynamic configurability enables lane resource to execute as a traditional SIMD processor, be re-purposed to behave as a clustered VLIW processor, or combinations in between. Dynamic configurability also enables efficient sharing of expensive resources between lanes (e.g., multipliers) by interleaving independent instructions with each lane's expensive instruction so as to hide resource contention. Libra consists of an array of simple processing elements (PEs) that are tightly interconnected by a scalar operand network. Groups of four PEs form PE groups that are normally driven by a single instruction stream. Each group can behave as a building block for a SIMD processor (e.g., PEs behave as SIMD lanes) or a VLIW processor (e.g., PEs behave as a cluster of function units). The compiler maps 1 or more loops to the Libra accelerator by combining and configuring clusters of PE groups to efficiently exploit the available DLP and ILP.

This paper offers the following three contributions:

- An in-depth analysis of the available ILP/DLP parallelism and its variability in three representative mobile application domains: computer vision applications, commercial media applications optimized in industry level, and game physics engine applications.
- The design of a unified loop accelerator that can effectively support future mobile applications with varying performance requirements and characteristics. To achieve this objective, we offer three key features:
  1. Scalability: Libra can meet high performance requirements by simply increasing the number of clusters, whereas most current accelerators suffer from poor scalability.
  2. Configurable performance: Libra can dynamically tune the ILP/DLP-support capability in order to successfully support ILP-intensive, DLP-intensive, and ILP/DLP-mixed applications, as well as tolerate performance degradation due to its heterogeneity.
  3. Energy efficiency: Simple hardware implementation achieves high energy-efficiency with competitive performance.
- A light-weight design and organization of a configurable processing element for supporting simple latency hiding techniques and sharing expensive resources.

2. Background and Motivation

In this section, we examine the limitations of traditional SIMD accelerators based on an analysis of various mobile applications. We first introduce the target benchmarks and the baseline architecture, and find two main sources of inefficiencies in SIMD accelerators. We then propose high-level solutions to overcome these challenges that facilitate designing efficient hardware and maximizing the utilization of existing resources.

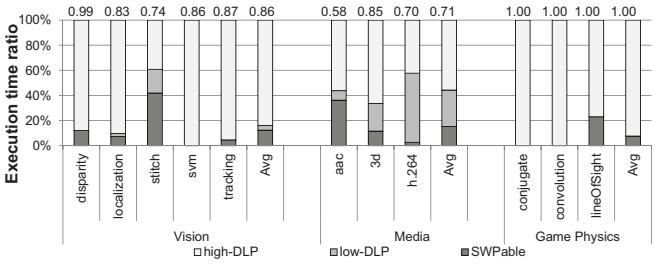


Figure 2: Loop categorization: The components of the bar indicate ratio of execution time in SWPable loops, low-DLP, and high-DLP SIMDizable loops. The ratio of loop execution time over total execution time is indicated as a number above each bar.

2.1. Benchmarks Overview

Three classes of mobile benchmarks are used for this application analysis that contain varying degrees of vector parallelism. The benchmarks consist of:

- Vision benchmark: We evaluated a subset of the SD-VBS benchmark suite [26] for mobile vision applications. As these benchmarks are not originally optimized for a specific target architecture, we manually modified these benchmarks to increase the opportunities for efficient execution with function inlining and loop unrolling. All the benchmarks are functionally verified on QCIF[1] input data sizes, which is widely used on mobile devices.
- Media benchmark: Three mobile media applications are selected: AAC decoder (MPEG4 audio decoding, low complexity profile), H.264 decoder (MPEG4 video decoding, baseline profile, qcif) [13], and 3D (3D graphics rendering) [3]. These benchmarks are optimized for DSPs in the production-quality level and a large portion of the loops have a high potential degree of ILP and are software pipelinable.
- Game physics benchmark: Three common kernels are extracted from mobile game applications [2]. First, lineOfSight plays an important role of separating visible objects and non-visible objects. Sound effects, collision detection and other functions involving linear equations often exploit convolution and the conjugate gradient method. The three kernels mostly consist of high DLP loops.

2.2. Baseline Architecture

A SIMD architecture that is based on SODA [15] is used as the baseline SIMD accelerator. This architecture has both SIMD and scalar datapaths. The SIMD pipeline consists of a multiple-lane datapath where each lane has an arithmetic unit working in parallel. Each datapath has two read-ports, one write-port, a 16 entry register file, and one ALU with a multiplier. The number of lanes in the SIMD pipeline can vary depending on the characteristics of the target applications. The SIMD Shuffle Network (SSN) is implemented to support intra-processor data movement. The scalar pipeline consists of one 32-bit datapath and supports the application's control code. The scalar pipeline also handles DMA (Direct Memory Access) transfers.

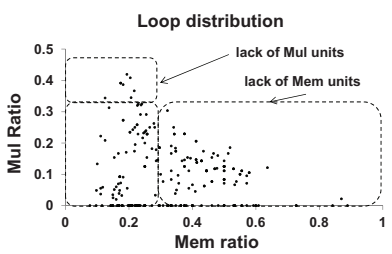
2.3. Limitations for Current SIMD Accelerators

2.3.1. Loop Characterization Applications typically have many compute intensive kernels that are in the form of nested loops.

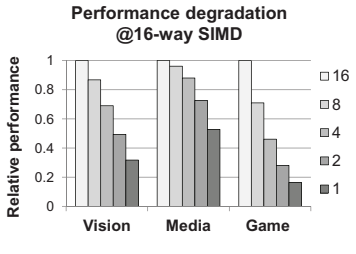
[1] We used QCIF (176x144) image size for uniformity of benchmarks, and the similar trend appears on higher resolution images.

	Vision	Media	Game	Total
Avg ratio(MEM)	0.44	0.26	0.27	0.32
Avg ratio(MUL)	0.15	0.10	0.22	0.16
ratio of MEM loop	0.93	0.36	0.33	0.54
ratio of Mul loop	0	0.04	0	0.01

(a)



(b)



(c)

Figure 3: Resource utilization: (a) average ratio of dynamic instruction count of expensive instructions and ratio of Mem/Mul dominant loops, (b) loop distribution over ratio of Mem/Mul, and (c) performance degradation on a SIMD at different number of Mem/Mul resources.

Among these kernels, we analyze the available ILP and DLP of the innermost loops and find the maximum natural vector width that is achievable. To extract the maximum degree of ILP, we found the *Software pipelinable* innermost loops: 1) counted loop, 2) no subroutine call, and 3) no multiple exits/backedges. Control flows inside the innermost loops are solved using if-conversion. Among the software pipelinable (SWPable) innermost loops, we also identify the *SIMDizable* innermost loops which can utilize DLP. We apply the conditions used by the Intel compiler [12] to determine if a loop is SIMDizable and the minimum iteration count is set to the maximum available SIMD width (natural SIMD width).

2.3.2. SIMD Width Variance over Loops Figure 2 shows the relative execution time of SWPable loops and SIMDizable loops to total execution time on a simple 1-issue ARM processor. As we use a 16-lane SIMD processor for this experiment, SIMDizable loops with natural SIMD width smaller than 16 are categorized into low-DLP loops. On average, there is a substantial amount of time (87%) spent on SWPable or SIMDizable loops as expected. An interesting question here is how many applications are not well-matched to a wide SIMD accelerator. Unfortunately, 4 of 11 applications are highly dependent on SWPable and low-DLP loops, which means that not all the lanes can be utilized. For example, traditional SIMD cannot decrease the execution time of an AAC application more than 60% of the total loop execution time because around 40% of the time is spent on SWPable loops. In general, the game physics benchmarks have high levels of data parallelism, vision benchmarks have modest data parallelism, and media benchmarks have low degrees of data parallelism. Results in Figure 2 confirm that a simple SIMD accelerator cannot effectively support the range of mobile applications. Even with a perfect support for DLP, SWPable and low-DLP loop execution result in low utilization of SIMD resources. Therefore, further consideration is required to fully utilize the SIMD resources on the execution of non-fully SIMDizable loops.

2.3.3. Resource Utilization Variance To maximize the total utilization of computation resources, the number of each resource should be decided based on the average fraction of dynamic instructions. While current CPUs solve these challenges by out-of-order execution of parallel instructions on multiple execution units, current SIMD architectures cannot solve this problem due to its homogeneous nature: the datapath of each SIMD lane has the same functionalities, even for expensive units such as memory and multiply units. These characteristics are unfavorable in terms of efficiency because not all execution units are active every cycle, and expensive units are much less utilized (an average of only 32% for a memory unit and 16% for a multiply unit (Figure 3(a))). A traditional solution for this problem is to turn off the unused resources by clock/input gating, but this solution does not eliminate leakage power. Power gating is unlikely a practical solution because idle periods for expensive units tend to be relatively short.

Another challenge is the diversity of instruction distribution across/inside applications. Even if we are somehow able to place a specific number of each execution unit based on average fraction, careful consideration is also required because the fraction varies greatly. In Figure 3(a), for example, the ratio of multiply instruction varies from 10% to 22% across three application domains. We also define a loop to be memory/multiplication dependent if the fraction of memory/multiplication instructions are more than 33% of the total instructions. Figure 3(b) shows a distribution of the loops according to the ratio of memory/multiply instructions. Based on Figure 3(a) and (b), more than 54% of the loops in the three benchmark sets highly depend on the memory instructions, and therefore, normal ALU functional units can be idle due to the memory operation bottleneck if only 33% of memory resources exist. On the contrary, multiplication is not the critical performance bottleneck if 33% of multiplication resources exist because only 1% of the loops are multiplication dependent. As a result, the high diversity in the instruction distribution will make most loops to not be effectively accelerated due to the lack of enough resources, or to waste resources due to the excess resources, if the SIMD accelerator simply allocate resources based on specific rules such as average fraction or one per four lanes.

2.4. Insights for the Traditional SIMD

Based on the application analysis, we found several fundamental sources of SIMD inefficiency. First, a traditional wide SIMD accelerator may be over-designed since the overall performance will be saturated at some point and limited by non-high-DLP loops where the SIMD accelerator is poorly utilized. Second, lane uniformity makes the SIMD datapath inefficient due to over-provisioning expensive resources. Third, the high variation in the resource requirements of loops makes the problem more difficult than simple sharing of expensive resources would accomplish. A central challenge here is how to decrease over-provided resources on traditional SIMD accelerators and to overcome the inflexibility in order to more effectively utilize the hardware.

3. Libra Architecture

3.1. Overview

The Libra accelerator presented here is a unified accelerator for mobile applications that allows flexible execution of loops by customizing the configuration adaptive to their key characteristics. The Libra

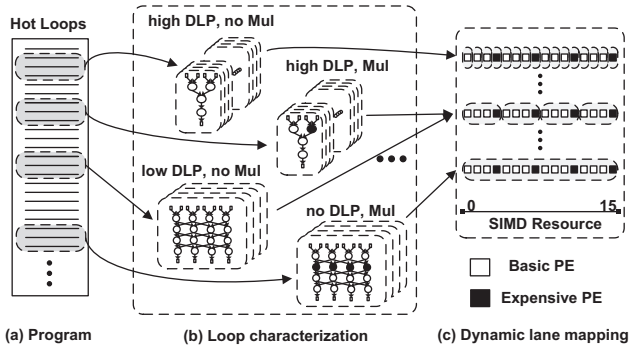


Figure 4: Mapping loops to Libra: (a) identify hot loops, (b) find the available DLP and resource requirement of each expensive operation, and (c) change the configuration based on the characteristics of each loop.

accelerator is based on traditional SIMD accelerators and has several important extensions for providing both high energy-efficiency and performance improvement. First, Libra is composed of a non-uniform lane structure for power efficiency: only a subset of lanes has expensive but infrequently used execution units. Furthermore, dynamic configurability of logical lanes helps Libra in executing a target loop in an efficient manner with high utilization. In Libra, a group of logical lanes is executed in a SIMD manner, where the logical lane is configured by a group of processing elements (PEs). DLP is exploited in the form of parallel execution of logical lanes, and ILP is exploited inside each logical lane in a way that each PE execute different operations. Therefore, Libra is able to flexibly tune the ILP/DLP-support capability by changing the logical lane configuration.

Figure 4 shows a conceptual view of the execution of Libra. First, several hot loops are identified as candidates to be accelerated utilizing the Libra architecture(Figure 4(a)). Second, software-pipelinable loops are selected, and the DLP availability is also determined as discussed in Section 2.3.1(Figure 4(b)). In this step, several additional key characteristics such as the amount of potential ILP in the loopbody and the ratio of expensive instructions are also considered. Finally, a best matched logical lane configuration for each loop is chosen by the compiler (Figure 4(c)). In Figure 4, we assume a 16-lane heterogeneous SIMD including 12 basic and 4 expensive PEs. Based on this, each PE constitutes one logical lane for full DLP support to execute high-DLP loops having only simple instructions, intermediate numbers of PEs form each logical lane for ILP/DLP hybrid execution to support low-DLP loops or expensive operation-intensive loops, and one large logical lane for full ILP execution is configured for non-DLP loops. Note that fully exploiting SIMD parallelism does not always outperform exploiting ILP on heterogeneous structures. Section 3.1.1 and 3.1.2 explain the core concept of Libra in detail with evidence of its effectiveness.

3.1.1. Heterogeneity Heterogeneous lane organization, based on average fraction of resource utilization, is required in order to enhance power efficiency: all the lanes support simple integer operations and only a subset of the lanes support expensive operations. When an expensive instruction is fetched, the accelerator stalls until this subset of lanes generates results for all lanes, then resumes execution. This structure delivers a high level of power efficiency due to the expensive resource removal, but significant performance degradation will occur when executing expensive operation-intensive code. Figure 3(c) illustrates the performance degradation as the number

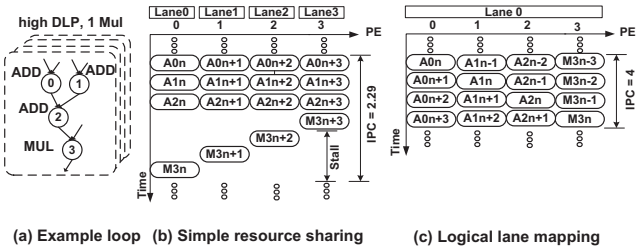


Figure 5: Dynamic configurability on a 4-lane heterogeneous SIMD (lane 3 has a multiplier): (a) a simple high-DLP loop with 1 multiply, (b) performance degradation due to stalls during multiply execution, (c) logical lane formation removes stalls by instruction pipelining.

of multiplier/memory units decreases on a 16-lane SIMD accelerator. Each bar shows the relative performance normalized to that of the homogeneous SIMD when each heterogeneous SIMD has specific number of expensive resources. From this graph, substantial amounts of performance degradation exist in vision and game benchmark because they are highly dependent on expensive operations and incur a number of stalls to handle these operations. However, media benchmarks are not highly affected by the proportion of these expensive resources because the performance is already constrained by low DLP.

3.1.2. Dynamic Configurability Dynamic configurability of lanes helps the heterogeneous SIMD accelerator in dealing with the aforementioned problems. One logical lane can consist of one PE for highly SIMDizable loops with no expensive instructions, and also consist of multiple PEs for non/low-SIMDizable loops or loops having expensive instructions. The resulting SIMD width is decided by the number of logical lanes and each logical lane executes the same instruction stream in lockstep. Inside a logical lane, ILP is exploited to use multiple lanes in parallel, and therefore it can efficiently distribute instructions between simple lanes and expensive lanes.

The effectiveness of dynamic lane mapping can be explained by the simple following performance equation. In the equation, we compare the total performance of the simple SIMD and the Libra SIMD by the metric of IPC (instruction per cycle). The IPC of SIMD can be calculated by the multiplication of IPC of one lane (IPC_{lane}) and the minimum of the number of PEs (N_{SIMD}) and the available degree of DLP (N_{DLP}) of the target loop (Equation (1)). Similarly, the IPC of Libra can be the multiplication of IPC of one logical lane ($IPC_{logical\_lane}$), consisting of m PEs, and the minimum of the number of logical lanes ($\frac{N_{SIMD}}{m}$) and the degree of DLP of the loop (Equation (2)). Therefore, when executing non/low-DLP loops, Libra can easily outperform the basic SIMD because it only requires better performance of a logical lane than that of a PE, and it is always true as a logical lane exploits ILP with multiple PEs inside(Equation (3)). Dynamic configurability is also able to address the performance degradation problem on the heterogeneous SIMD. When executing high-DLP loops, Libra outperforms SIMD when the IPC of a logical lane is higher than that of m PEs. Although the ILP performance is normally inferior to DLP performance because of its dependences and complexity, Libra can frequently be better due to the heterogeneity. Figure 5(a), (b) and (c) shows the superiority of Libra. Figure 5(b) and (c) show the execution of a simple high-DLP loop having a multiply instruction on both the simple SIMD and Libra which have one multiplier on the PE 3. In this example, the IPC of SIMD is less than the IPC of Libra when one large logical lane is configured due to a number of stalls.

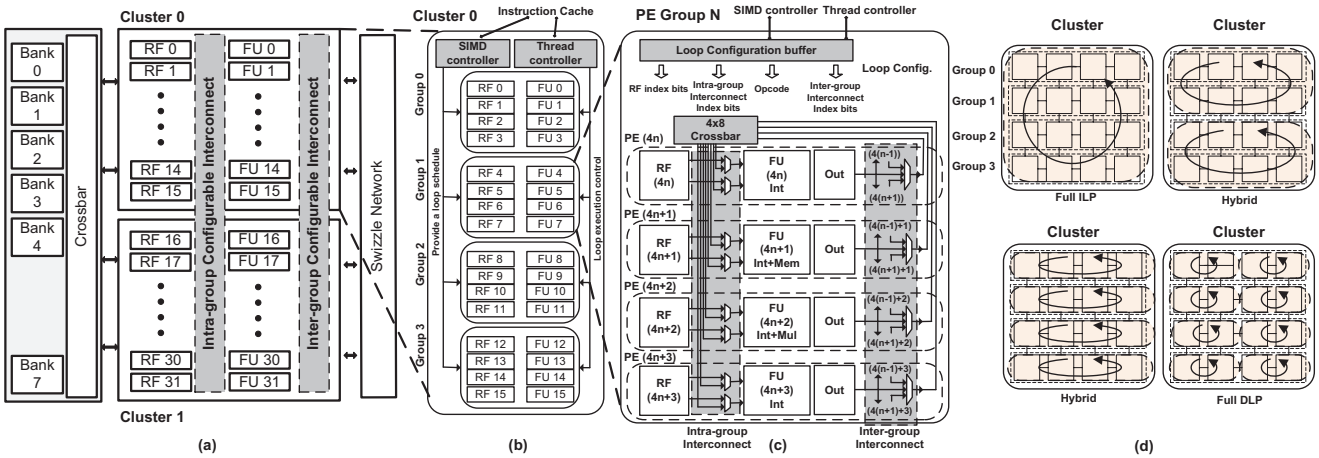


Figure 6: The 32-PE Libra architecture: (a) a 2-cluster Libra accelerator, (b) a cluster, (c) an example of a single PE group: PE 1 supports memory operation and PE 2 supports multiply operation, and (d) execution modes.

$$IPC_{SIMD} = \min(N_{SIMD}, N_{DLP}) \times IPC_{lane} \quad (1)$$

$$IPC_{Libra} = \min(\frac{N_{SIMD}}{m}, N_{DLP}) \times IPC_{logical\_lane} \quad (2)$$

$$IPC_{Libra} > IPC_{SIMD},$$

$$when \begin{cases} IPC_{logical\_lane} > IPC_{lane}, & \text{if } \frac{N_{SIMD}}{m} > N_{DLP} \\ IPC_{logical\_lane} > m \times IPC_{lane}, & \text{if } \frac{N_{SIMD}}{m} < N_{DLP} \end{cases} \quad (3)$$

3.2. Microarchitectural Details

The Libra architecture with eight PE groups (32 PEs) is shown in Figure 6(a). Differently from the traditional SIMD, the Libra datapath consists of 2 groups of clusters, which can be configured to create logical SIMD lanes of 2, 4, 8, and 16 PEs based on the loop characteristics. Each of the clusters is composed of 4 PE groups. The SIMD controller performs the role of managing the logical lane status to exploit SIMD parallelism, while the thread controller manages the ILP-exploiting method inside the logical lane. Each PE group contains 4 PEs. Each of the PEs has an FU and a register file, which can be thought as one lane of the traditional SIMD. Only one of the PEs in a PE group has a multiplier while another has a memory unit. Differently from the traditional SIMD, each PE group also has two kinds of reconfigurable interconnects inside and across PE groups in order to achieve flexible configuration of logical lanes.

Key features of Libra architectures are as follows:

Scalability: The resources are fully distributed including FUs, register files, and interconnections. PE groups have dense interconnections inside but each PE group is sparsely connected with neighbors. As a result, area and power costs increase approximately proportional to the number of resources, which makes Libra as scalable as a simple SIMD.

Polymorphic Lane Organization: PE groups can be aggregated to form a larger logical lane in order to exploit the existing ILP inside the loop body, or be split into multiple small logical lanes in order to exploit DLP over loop iterations.

Resource Sharing: In heterogeneity, the major challenge is how to determine the number of expensive resources and how to efficiently share them between logical lanes when necessary. To flexibly handle this, we place the expensive resources based on the average utilization and provide a sharing mechanism between them in two categories. A more detailed description is provided in Section 3.3.3.

Simple Multi-threading Mechanism: Even though a logical lane provides a number of parallel resources, efficient use of the available resources is limited due to the low ILP of the loopbody. Therefore, we extended the ILP into loop-level parallelism through modulo scheduling [20]. Modulo scheduling generally provides a decent performance improvement by parallelizing instructions over loop iterations and hiding long latency between back-to-back instructions. However, several Libra specific features, such as SIMD capability and fully-distributed nature, diminish the effectiveness of modulo scheduling. To compensate for this, simple static multi-threading with list scheduling is proposed in Section 3.4.

3.2.1. PE Group A detailed illustration of a single Libra PE group is provided in Figure 6(c). A PE group consists of four PEs each with a 32-bit FU and a 16-entry register file with 2-read/1-write ports (write ports can be added to support threading). Integer arithmetic operations are supported in all four FUs but multiply and memory operations are available in only one FU per PE group (PE1 for memory and PE 2 for multiplication in Figure 6(c)). The FUs inside are modified to connect with each other with a dense 4x8 full crossbar network for passing data between the FUs without writing back to the RF. This allows the PE groups to exploit ILP in a distributed nature. In order to retain scalability, the Libra architecture has a simple and fully distributed across-PE group interconnect. Only FUs are connected between the corresponding neighbors in adjacent PE groups. In addition to these components, a loop configuration buffer is added to store instructions for modulo/list scheduled loops. The buffer is a small SRAM that saves the configuration information including instructions, register addresses and interconnect index bits of the current loop. The interconnect between the loop buffer and SIMD/Thread controllers in the cluster is used to transfer instructions for executing loops. The hardware components and execution mechanism for SIMD/ILP support is explained in detail in Sections 3.3 and 3.4.

3.2.2. Cluster A cluster is a high-level basic unit that consists of four PE groups and several additional features for flexible loop execution support: the SIMD controller and the thread controller. The SIMD controller is a small controller to manage the logical lane organization inside the cluster, including the number of logical lanes and the SIMD width of memory transfer. It receives the information from the instruction cache. In addition, the SIMD controller also gets the configuration for one logical lane from the instruction cache

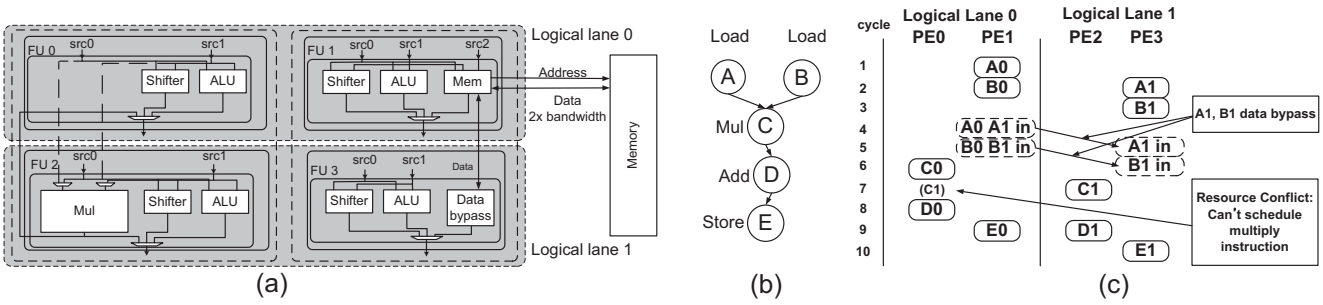


Figure 7: Resource sharing support: (a) hardware modification: PE 0 and 2 share the multiplier and PE 1 and 3 share the memory unit, (b) example loop body dataflow graph, and (c) actual schedule: 1-cycle difference between lanes for resource contention avoidance.

and transfers it to each PE group. A thread controller is responsible for executing loops. It also gets the information about which mode is selected from the instruction cache and orchestrates the loop execution. When modulo scheduling is selected, it just executes the loop sequentially, and, when multi-threading is selected, it executes the loop in the order of the thread sequence table. The information is statically set during compile time and is fetched from the instruction cache. Multiple clusters can execute one large loop or can execute multiple parallel loops separately.

3.2.3. Configuration Process Loop execution of Libra can be divided into two stages: configuration and execution. Configuration stage is forming logical lanes and sending configuration bits to all the loop buffers of each PE-group. For every loop, the instruction cache contains both logical lane organization information and configuration bits for one logical lane. The SIMD controller gets these information from the instruction cache and then sends the configuration bits to the loop buffers of the PE groups based on the logical lane configuration. The thread controller also gets the information about the execution mode and sequence table, if required, from the instruction cache. This process takes 3-5 cycles on average before the loop buffer receives the configuration bits for the first cycle and the time varies depending on the size of the logical lane. The thread controller starts the execution when the first cycle configuration is ready on all the loop buffers.

3.2.4. Memory Support The memory operation of the Libra system needs support for both scalar and SIMD memory access. For scalar memory access, the local memory has the same number of banks as the number of total memory units. For SIMD access, the local memory also needs to support contiguous access across all logical lanes in parallel. Therefore, for the 32-PE Libra system, a 64kB local memory is used, consisting of 8 memory banks where each bank is a 2-wide SIMD containing 1024 32-bit entries. As shown in Section 2.3.1, all memory transfers have the same strides over iterations in SIMDizable loops. Therefore, when several logical lanes execute the same instructions for SIMDized loops, a single address calculation followed by a wide memory operation is performed. The data is then distributed to different logical lanes. Multiple memory units inside a logical lane need to generate their own memory addresses. The SIMD width of each access and the number of different addresses are determined by the logical lane configuration, which is saved in the SIMD controller.

3.2.5. Communication with a Host Processor The Libra architecture is a co-processor similar to a GPU and interfaces with a host processor such as ARM using memory. The data transfer is performed through a standard AMBA bus along with a DMA.

3.3. Execution Model

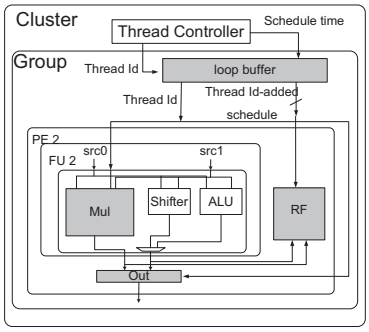
This section describes the three different execution modes of the Libra architecture, which are full ILP, hybrid, and full DLP modes. We first explain how each mode operates and then provide proof of how the three modes can effectively support different kinds of loops. The example provided assumes a four-PE group cluster as shown in Figure 6(d).

3.3.1. Full ILP Mode In this mode, the Libra architecture decides to use all the PEs as one large logical lane. The SIMD controller spreads different configuration informations into the loop buffer of each PE group. The execution mechanism is the same as the loop acceleration technique of common VLIW solutions but the performance might be slightly worse than previous solutions because the Libra architecture sacrifices both centralized resources and dense across-PE group interconnects. Applications which have a high proportion of non-SIMDizable loops mostly utilize this mode for acceleration.

3.3.2. Hybrid Mode When a loop is SIMDizable, a cluster has the possibility of either having several small logical lanes or forming a large logical lane. In this case, the Libra architecture may choose to use a hybrid mode with a cluster having at least two logical lanes, each having at least one PE group. With smaller logical lanes, the performance usually increases since SIMDization provides an opportunity to increase performance by the same amount as the degree of DLP. Also the routing overhead decreases with small logical lanes, further boosting performance. Figure 6(d) also has two examples of hybrid mode execution. The SIMD controller distributes the same configuration information and live values to the loop buffer and RFs of each logical lane. When a loop lacks sufficient level of DLP or has a moderate proportion of expensive resources, hybrid mode can achieve the best performance.

3.3.3. Full DLP Mode When a loop is highly data-parallel but has a low degree of ILP, the resources (PEs) cannot be effectively utilized because the degree of ILP in the loop cannot meet the minimum degree of the PE group. To compensate for the lack of ILP, the Libra architecture supports separation of PE groups, forming two smaller logical lanes. As a result, SIMD parallelism can make up for insufficient ILP in the loops (also in Figure 6(d)). Hence, a cluster has a total of eight logical lanes executing in lockstep. Distinct from loops with a small number of instructions, loops with unbalanced resource usage can also be well matched to a full DLP mode, unlike the hybrid mode. As mentioned in Section 2.3.3, the hybrid mode cannot fully utilize resources in a PE group since performance of loops with a high proportion of memory operations are constrained by the memory unit.

The major challenge in full DLP mode is determining how to



(a)

Cycle	Thread Id	Loop buffer address
0	0	0
1	0	1
2	1	0
3	1	1
4	0	3
5	0	4
6	0	5
7	1	3
8	1	4
9	1	5

(b)

Original Cycle	Original Configuration
0	A
1	B
2	NOP
3	C
4	D
5	E

(c)

Cycle	Thread-aware configuration
0	A0
1	B0
2	A1
3	B1
4	C0
5	D0
6	E0
7	C1
8	D1
9	E1

(d)

Figure 8: Multi-threading support & compiler support: (a) hardware modification: shaded components are modified, (b) sequence table in the thread controller, (c) loop buffer, and (d) final multi-threaded schedule.

share expensive resources between two small logical lanes in a PE group. The first category for resource sharing is expensive but infrequently used functionalities such as the multiply operation. As shown in Figure 3(a), the average ratio of multiply is as low as 16% and only 1% of loops are multiply-dominant, and therefore simple sharing between two half-PE groups does not incur performance degradation. The second category is frequently used functionalities such as memory operations as shown in Figure 3(a). These instructions are already a performance bottleneck and simple sharing cannot enhance the overall performance. Therefore, this shared resource should lead to double the performance in a lightweight manner.

We accomplish these requirements using simple hardware modifications as shown in Figure 7(a). One PE group is mapped into two small logical lanes with (PE 0, PE 1) and (PE 2, PE 3). Based on the application analysis, only PE2 supports multiply operations and PE 1 supports memory operations. To ensure that both logical lanes support all functionalities, PE 0 and PE 2 share the multiplier and PE 1 and PE 3 share the memory unit. To share the multiplier, PE 0 connects input and output ports to the multiplier of PE 2. A memory controller in PE 1 is shared with PE 3 in a different manner. When the memory controller receives a memory operation command, only PE1 communicates with the memory with double bandwidth and send/receives the data of PE 3 through a bypass logic.

To execute the same instructions in both logical lanes using the above modifications, the following processes are required:

- The compiler must not schedule multiply instructions in a row, because the multiplier needs a spare cycle after the cycle in which the multiply instruction is scheduled in order to handle the operation of the other logical lane. However, other instructions can be placed since they have no resource or writeback contention. Memory instructions can be scheduled without any restrictions as the hardware supports double bandwidth.
- The SIMD controller has the instruction configuration only for one logical lane. The controller transfers the same configuration into the loop buffer of both logical lanes with one-cycle difference to avoid resource contention.

Figure 7(b) is an example of a full DLP mode execution. For a simple dataflow graph of the loop body, the latency of the load and multiply operations are set to 4 and 2. Due to the small size and high memory dependent characteristic of the loop body, a full DLP mode is selected and each PE group is separated into two logical lanes. Identical schedules based on two PEs are transferred into the loop buffer in the PE group with one cycle difference between logical lane 0 and logical lane 1 (see Figure 7(c)). Different memory operations can execute in the same cycle as shown in cycle 2 but different multiply instructions cannot be scheduled at cycle 7 because logical lane 1 needs to use the multiplier in that cycle.

3.4. Improving ILP Performance

Although modulo scheduling has proven to be an effective solution to exploit ILP over loops, it is not always the best solution because 1) original iteration count is divided by DLP capability, and therefore, the smaller iteration count may not compensate for the prolog and epilog overheads even in moderate DLP loops [23] and 2) sparse interconnection between PEs and no centralized RFs make the quality of the schedule worse. As a result, we suggest supporting list scheduling [6] of the loop body as another option to exploit ILP. When either there is not much total ILP in the loop, or the hardware cannot benefit from increased ILP, list scheduling can outperform modulo scheduling since it does not incur the overhead of modulo scheduling: handling modulo information such as staging predicates.

The remaining problem of adapting list scheduling to hide idle cycles comes from long latency instructions such as multiply and memory operations. To solve this problem, we propose a simple multi-threading scheme with fast context switching. Assuming the Libra architecture supports two threads, a loop with large number of iterations is divided into two threads with identical loops with half number of iterations. The two threads are then executed on the same logical lane. To make the scheme simple, a switch of running threads is allowed only when all the PEs are idle. Each thread has its own register file space divided by the number of threads, similar to what a GPU does, and therefore no context change overhead exists. The schedule with multiple threads is statically decided at compile time. The multi-threading technique is simple but highly effective and is a realistic solution because of the following two reasons: 1) low register pressure: loops with small number of instructions have a small amount of data to save in the register file and list scheduling does not require additional register overhead, and 2) a high chance of hiding latency: this technique is applied only to SIMDizable loops executing on small logical lanes, thus increasing the probability that all FUs are idle.

Although multi-threading looks promising, the Libra architecture faces a number of challenges in reality. There are three essential challenges and we present the lightweight solutions incorporated in the Libra architecture:

Context Saving: The fully distributed nature of Libra allows temporal data to be saved in the register files as well as the output buffer

in order to directly transfer the data between FUs. As a result, the output buffer data of each thread should also be saved in addition to the register files. The register file is divided into the same number of threads. The parts are then addressed by the thread ID. However, the output buffer is originally a simple flip-flop without addressing support. Therefore, it is substituted by an n-entry register file addressed by thread ID(n: the number of threads supported). The output data can thus remain unchanged when another thread is executed.

Writeback Contention Avoidance: Handling multi-latency instructions is not a simple problem if the output data from a multi-latency instruction is generated when the other thread is executing. To solve this problem, multi-latency FUs need to save the thread ID when the input is issued and be connected to the output buffer (small register file) with an additional port addressed by the original input thread ID. Since only a single additional port is required for multiple FUs with the same latency, the overhead is negligible. For the Libra architecture, only two ports are added to the whole PE group to support a multiplier and a memory controller.

Code Bloat: Since multiple threads are scheduled at compile time, the loop buffer of each PE group needs to contain the entire schedule information of all threads for each cycle. This causes the code bloat problem, requiring an increased loop buffer size which incurs a power overhead. However, an important observation to point out is that the schedules of different threads are essentially the same, just with different execution times. We can, therefore, solve the problem by 1) saving the schedule configuration of only one thread and 2) adding a simple sequence table which contains a thread ID and the corresponding loop buffer address pointing to the actual schedule configuration. The thread controller contains the basic information for supporting multi-threading and the sequence table.

Figure 8 shows an illustration of the Libra architecture with an emphasis on modified features(shaded components) to support multi-threading, assuming that the architecture supports execution of two threads. The loop buffer contains configuration information for only one thread as shown in Figure 8(c). Therefore, its size is the same as when one thread is executed. The thread controller in the cluster has a tiny sequence table containing the actual thread ID and the address of the configuration saved in the loop buffer. Figure 8(b) depicts an example sequence table for two thread execution. Since two threads are executed in this example, the space of RF is divided by two and the output buffer is a 2-entry register file. By reading the sequence table from cycle 0 to cycle 9, the thread controller transfers the thread ID and loop buffer address for each cycle to the loop buffer. From this information, the loop buffer generates the final configuration by reading the appropriate configuration and adding a thread ID to the register file address (see Figure 8(d)). The multiplier gets the thread ID and has a separate data bus due to the multi-latency functionality. When the original configuration B has the multiply operation for FU 2, the result data from thread 0 and B configuration can be stored in the output buffer at cycle 2 without any writeback contention.

3.5. Decision Flow

In order to maximize the performance and resource utilization, the Libra architecture depends on an intelligent selection of the configuration between the number of logical lanes and the size of each logical lane. The system flow is shown in Figure 9. Applications run through a front-end compiler, producing a generic Intermediate Representation (IR), which is unscheduled and uses virtual registers. The compiler also has a high-level machine specific information, including the number of resources, size of register files, the size of a

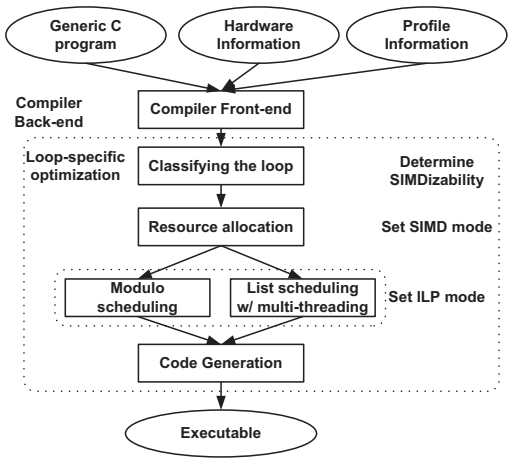


Figure 9: Decision flow of the Libra architecture.

cluster, and the number of supported micro-threads. In addition to this, the compiler needs to have profile information about the iteration counts of loops and memory alias information. Given the IR, hardware and profile information, the compiler categorizes loops into two basic types: SWPable and SIMDizable loops. The compiler then decides the logical lane configuration of a cluster for each loop (resource allocation). If a loop is not SIMDizable but only SWPable, the entire cluster is assigned to the loop. If a loop is proved as SIMDizable, the compiler finds the best configuration based on the provided information such as average iteration count, instruction and dependency information of the loop. Briefly speaking, the compiler tries to fully exploit SIMD parallelism by securing the maximum number of logical lanes without performance degradation due to the instruction imbalance. However, it also performs broad design space explorations by changing the number of logical lanes. This is because 1) sometimes the effectiveness of DLP is not clear when the divided trip count is small and the instruction number is not too small, and 2) the scheduler uses a heuristic way to generate the modulo schedule. After deciding the lane configuration, the compiler chooses the method to exploit ILP inside the logical lane. Finally, the compiler performs modulo scheduling or list scheduling. It then generates the final schedule and the configuration information.

4. Experiments

4.1. Experimental Setup

Target Architecture To evaluate the effectiveness of the Libra architecture, three example implementations with different sizes are used: 16 (one cluster, four PE groups), 32 (two clusters), and 64 (four clusters) PEs. Four FUs per cluster are able to perform load/store instructions to access the data memory with four-cycle latency while another four FUs support two-cycle pipelined multiply instructions. The Libra is compared against two other accelerators in our experiment. We generate 4(cluster)×4(PE), 8×4, and 16×4 heterogeneous VLIWs having the same organization of PEs as corresponding Libra architectures. The wide SIMD architecture as discussed in Section 2.2 is used and the number of SIMD resources can vary from 16 to 64, having the same heterogeneous FU structure.

Target Applications As discussed in Section 2.1, the evaluation is conducted for subsets of three domains. Max 20 top loops having a high execution time are selected for vision and game physics benchmarks, and 144 loop kernels, varying in size from 4 to 142

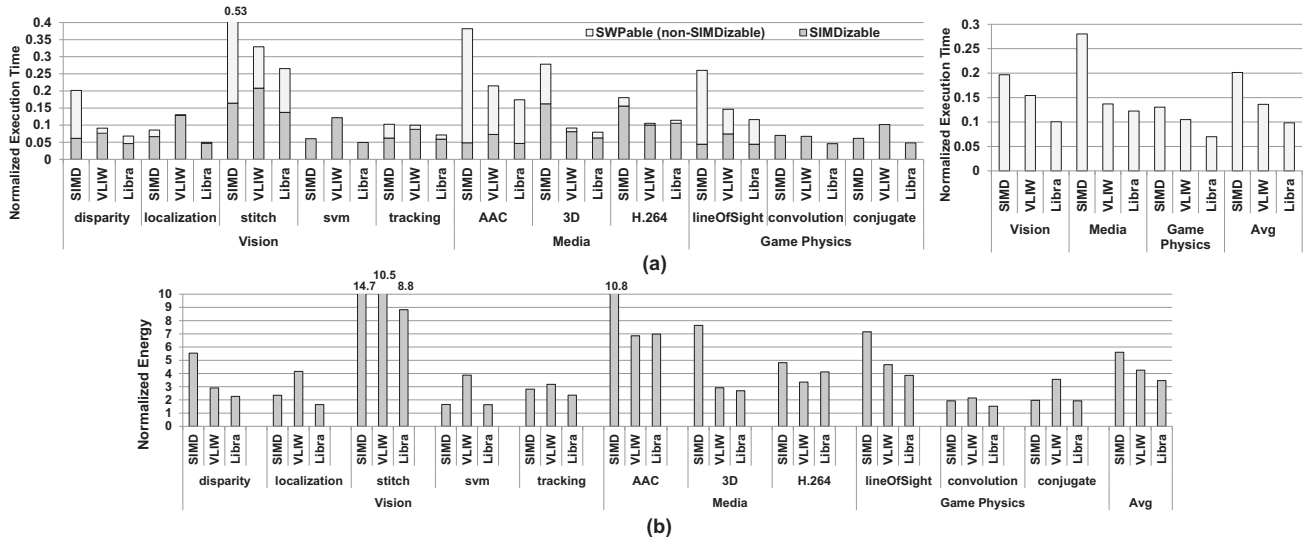


Figure 10: Performance/Energy comparison of 32-PE Libra/SIMD/VLIW architectures: (a) total loop execution time and (b) energy consumption. All the data are normalized to that of a simple in-order core.

operations, are extracted from the media benchmark because the ratio of execution time to the total execution time of the top 20 loops is too small. High number of loops in the media benchmarks and several major loops in the vision benchmarks have conditional statements, while the gaming benchmarks do not have them. In order to eliminate all internal branches, we applied if-conversion for these loops.

Compilation and Simulation The industrial tool chain developed by SAIT [5] is used for compilation and simulation of Libra. The IMPACT compiler [19] is used as the frontend compiler. Basic list scheduler [6], edge-centric modulo scheduling (EMS) [20]-based modulo scheduler, and simple loop-level SIMDization scheduler using a SODA-style [15] wide vector instruction set are implemented in the backend compiler. Based on the original modulo scheduler, we developed a scheduler that can support both flexible execution of Libra and list scheduling with static multi-threading technique. The performance is generated by the cycle-accurate code schedule of loops, accounting for the configuration overhead.

Performance Measurement For fair comparison, both list scheduling and modulo scheduling are applied and the better performing schedule is picked for the SIMD accelerator. For VLIW, loop unrolling is applied when a loopbody size is too small and its resources may not be fully utilized. Multi-threading technique of Libra is also not applied for a fair comparison of the performance of the three architectures. This issue is discussed in Section 4.6.

Power/Area Measurements All architectures are generated in RTL Verilog, synthesized with the Synopsys design compiler, and place-and-routed with the Cadence Encounter using IBM SOI 45nm regular Vt standard cell library in slow operating conditions with a 0.81V operating voltage. Synopsys PrimeTime PX is used to measure the power consumption based on the utilization. The Artisan Memory Compiler is used to determine the area and the power of the memory operation using a 0.81 Volts operating voltage. The target frequency of Libra is 500MHz[2] similar to the latest mobile GPUs.

[2]The FO4 delay of this process is about 13ps.

4.2. Performance/Energy Evaluation

We compared the performance of a 32-PE Libra architecture with identically sized VLIW (8×4) and SIMD(32-wide) architectures. Performance results are measured as the total loop execution time when each loop is scheduled by the method the target architecture supports. Figure 10(a) shows a plot comparing the performance of the three architectures normalized to the simple 1-issue inorder core. For individual benchmarks, the graph also indicates the fraction of two different loop categories: SIMDizable and SWPable loops.

For benchmarks with a high ratio of non-SIMDizable loops such as stitch, AAC, and lineOfSight, SIMD shows severe performance degradation, whereas VLIW and Libra show a fair performance improvement. Libra outperforms even VLIW because it can accelerate SIMDizable regions more efficiently. On the other hand, both the SIMD and Libra deliver a substantial performance improvement for benchmarks with mostly SIMDizable loops, while VLIW suffers. The Libra also shows better performance than SIMD because it effectively accelerates applications having low-SIMDizable loops (3D, H.264) and its ILP capability also helps Libra to adequately tolerate the lack of expensive resources for high-SIMDizable loops (convolution, conjugate). Overall, Libra shows the best performance in all benchmarks except H.264 benchmark. This is because of the slightly lower performance gain on SWPable regions due to its distributed nature. Among average result of each domain, performance gain of Libra is the highest on game physics. As a result, Libra shows a performance gain of 2.04x and 1.38x over SIMD and VLIW, respectively.

Despite using the same amount of computation resources, performance-only comparison may not be fair due to the different interconnection strategies among the architectures. An energy comparison may yield a better comparison considering both performance and hardware overhead. Figure 10(b) shows the energy consumption of three architectures and the results are also normalized to the 1-issue core. This graph shows a similar trend to Figure 10(a). On average, even though SIMD added extra logics for handling sharing resources (Figure 5(b)), VLIW shows 16% more power consumption because of bigger RFs and complex control logics, and Libra shows 20% more power consumption due to more interconnects and

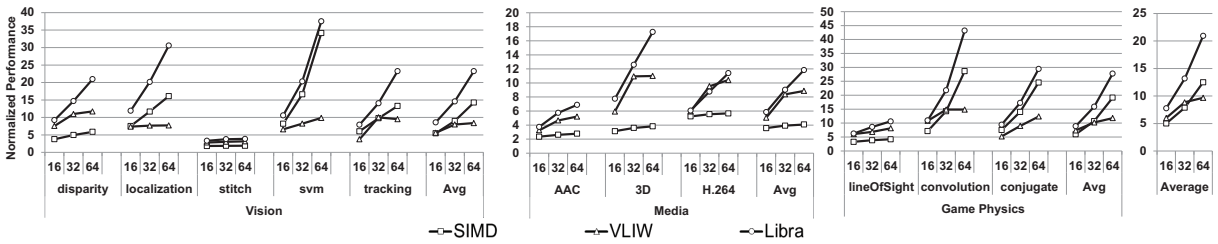


Figure 11: Scalability of Libra/SIMD/VLIW architectures: the Libra architecture is highly scalable for most of benchmarks, while SIMD and VLIW cannot be scalable for several benchmarks.

Libra-specific overhead such as a loop-buffer and a thread controller. Based on these power differences, the Libra saves 38% and 19% energy compared to SIMD and VLIW, respectively[3]. As a result, the Libra architecture shows a fair amount of performance improvement in addition to high energy efficiency by providing a more suitable acceleration scheme for each loop.

4.3. Scalability

Figure 11 shows the performance of each architecture normalized to a 1-issue core for different sizes across three benchmark domains. The number of PEs varying from 16 to 64 are shown on the X-axis. The results show high scalability of the Libra architecture in all benchmark domains.

In the vision and game domain benchmarks, applications are not specially optimized to the SIMD-style architecture, but the performance is highly scalable as the number of PEs increases because most loops are simple and highly SIMDizable. Only the stitch is barely scalable because the application is mostly sequential as the dominating loop has only a small number of iterations. In the media domain, the Libra accelerator performance also fairly increases as it scales to more PEs. Compared to other architectures, VLIW performance results are frequently saturated because modulo scheduling of a big size loopbody(often unrolled) on a large number of PEs is too complex to exploit ILP, while Libra solves this problem by scheduling a small loopbody in a small logical lane and applying the same schedule to multiple logical lanes. The SIMD results are also constrained by lack of expensive resources and program complexity. To summarize, the Libra architecture can increase its performance with larger resources when the application has enough total ILP/DLP parallelism.

4.4. From the Homogeneous SIMD to the Heterogeneous Libra

Section 4.2 and 4.3 evaluate three different architectures consisting of the same computation resources. The key question here is how much Libra surpasses the traditional SIMD architecture. To answer this question, we compared the performance and energy consumption of the heterogeneous Libra and the homogeneous SIMD. The heterogeneous Libra has a quarter of memory/multiply resources and the homogeneous SIMD has the same number of memory/multiply resources as the total number PEs. Figure 12 shows the average of relative performance and energy consumption of Libra over SIMD for different sizes. In terms of performance, Libra outperforms SIMD and the difference increases in proportion to the size (Figure 12(a)). This is because 1) the lack of expensive resources can be effectively compensated for by forming logical lanes and 2) the lane utilization of the traditional SIMD is lower for a larger size due to the program characteristics.

In terms of the energy consumption, Libra still shows similar results as its performance improvement because significantly less computational units can reduce the overall power overheads, and the result is better on larger size. For example, the 32-PE heterogeneous Libra consumes 11% more power than the same size homogeneous SIMD due to 12% power savings on FUs with 23% overheads (Figure 12(c)). On average, Libra shows 101%, 71%, and 56% energy consumption compared to the traditional SIMD.

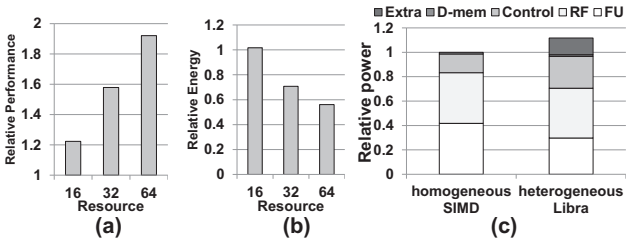


Figure 12: Performance/energy improvement of the heterogeneous Libra over the same sized homogeneous SIMD: (a) performance, (b) energy consumption, and (c) power breakdown with five categories: FU, RF, control logic, memory, and architecture specific additional logic.

4.5. Acceleration Mode Selection

Our experiments so far have focused on the overall performance of the Libra architecture compared to other architectures, showing considerable performance enhancement. In this section, we evaluate the effectiveness of flexible lane mapping to answer the question if Libra really needs to provide various intermediate sizes of logical lanes between SIMD and VLIW. Figure 13(a) shows the execution time distribution at different logical lane sizes for the three application domains on the 16, 32, and 64-PE Libra. On average, all available modes are used for considerable fraction of time and no dominating logical lane size exists, which proves the effectiveness of flexible lane mapping. Furthermore, the lane sizes are selected adaptive to the domain characteristics. For vision benchmarks, 2-PE small sized logical lane is dominant because most loops are small and memory operation dominant. In media benchmarks, large logical lanes are used for a high fraction of the execution because of lack of DLP. Game physics uses a 4-PE logical lane in substantial fraction to execute high-DLP loops with some ILP. Figure 13(b) compares the normalized performance of Libra to that when only one specific logical lane configuration is allowed to execute benchmarks. The results of this graph further prove the effectiveness of flexibility by showing that any fixed mode execution cannot win over the flexible execution.

[3]Figure 10(b) does not mean that a simple 1-issue core is 3x energy efficient than Libra because the performances are different. For a performance-equivalent comparison, Libra is much more efficient than the simple core.

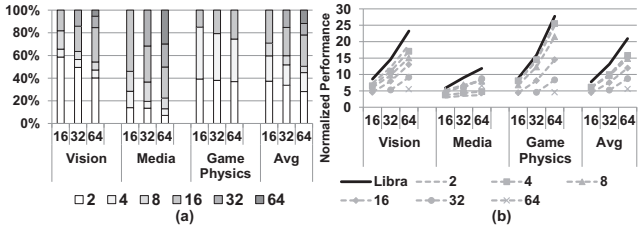


Figure 13: Mode selection: (a) execution time distribution at different logical lanes, (b) flexible vs. fixed execution.

4.6. Multi-threading Effectiveness

As discussed in Section 3.4, a simple multi-threading functionality is added to Libra. In this section, we evaluate the effectiveness of this functionality. Figure 14(a) shows the performance improvement on SIMDizable loops only, since this technique can be only applied to SIMDizable loops. On average, a performance gain of 12-16% is achieved, and this is up to 28% more effective in vision benchmarks because the majority of loops are small and multi-threading is most effective in small size logical lane mapping. Figure 14(b) shows the execution time distribution for different logical lane sizes when multi-threading is applied. Compared to Figure 13(a), a substantial amount of 2 and 4-PE logical lane execution is substituted with multi-threading. Overall, multi-threading is effective for small logical lanes when executing SIMDizable loops.

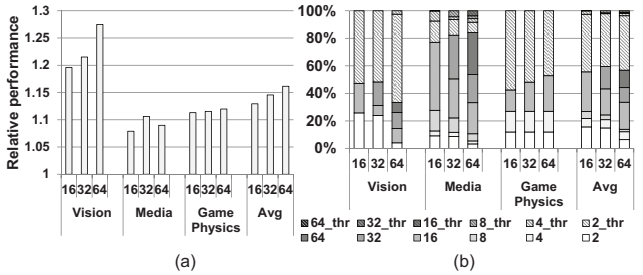
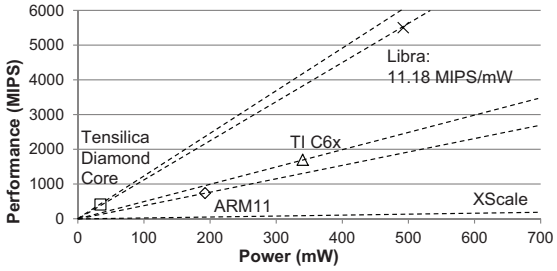


Figure 14: Multi-threading effectiveness: (a) performance improvement for SIMDizable loops, (b) execution time distribution at different logical lanes.

4.7. Power and Area Measurement

We measured the average power when the 32-PE Libra architecture executes the H.264 benchmark at 500 MHz. A power and an area consumption breakdown for various components that are part of the architecture are shown in Figure 15(b). Compared to the normal SIMD, the power consumption of the routing logic is larger due to its dynamic configurability, but FU power is smaller due to the smaller number of expensive units. A SIMD controller and four loop buffers, and a thread controller are added to a cluster. The power consumption of a SIMD controller and four loop buffers is substantial because the loop buffer is implemented as 64-entry wide two-port SRAM and the data is read every-cycle. In addition to this, the thread controller also consumes 0.7% of total power because the sequence table is a 256 entry 8 bit two-port SRAM. The total area of the 32-PE Libra architecture is 2.0 mm^2.

Based on the power and performance data, we compared the efficiency of Libra to other architectures using data shown in [11]. Based on Figure 15(a), the Libra architecture achieves 11.18 MIPs/mW and most of the other well-known solutions show lower efficiency. The Tensilica Diamond Core is slightly more efficient than the Libra architecture, but the actual performance is not enough to successfully execute compute-intensive media applications.



Component	Power(mW)	Ratio(%)	Area(um^2)	Ratio(%)
SIMD FUs	131.3	26.7%	341909	17.1%
SIMD RFs	180.2	36.6%	405963	20.3%
SIMD Pipeline + Routing + Scalar Pipeline	115.5	23.5%	117721	5.9%
Instruction Control (SIMD controller + Loop buffer)	56.0	11.4%	471984	23.6%
Thread controller	3.2	0.7%	37714	1.9%
D-mem (64kB)	5.9	1.2%	626550	31.3%
Total	492.2	100.0%	2001840	100.0%

(b)

Figure 15: (a) Power/Performance comparison, and (b) power and area breakdown of the 32-PE Libra architecture.

5. Related Works

Many previous works have focused on accelerators to address the challenges of improving computing efficiency. Some exploit only one type of parallelism and others introduce some flexibility to support more than one type of parallelism. Figure 16 compares and shows the major differences between Libra and prior works.

		ILP	DLP	Heterogenity	Configurable Performance	Scalability	Power Efficiency
DLP Accelerator	SIMD	No	High	No	No	High	High
	GPU	Low	High	Limited	No	High	Low
	Embedded GPU	Low	High	Limited	No	High	High
ILP Accelerator	ADRES	High	No	Yes	No	Low	High
DLP + ILP Accelerator	Imagine	High	High	Yes	No	High	Low
Flexible Accelerator	AnySP	Low	High	No	Limited	High	High
	SIMD-Morph	High	High	No	Limited	Low	High
	TRIPS, SCALE	High	High	Yes	Yes	High	Medium
	Libra	High	High	Yes	Yes	High	High

Figure 16: Comparison to prior work

Accelerators for multimedia usually focus on one type of parallelism without adaptive configuration. Conventional SIMD [9, 15] only supports DLP and misses the opportunity of improving performance with other form of parallelism. By Amdahl's law, low-DLP regions quickly become the bottleneck of applications. Conventional SIMD also wastes expensive resources due to imbalanced utilization. While the latest GPUs [18, 17] support the limited level of heterogeneity and embedded GPUs such as Qualcomm Adreno [4] and ARM Mali [1] are power-efficient, GPUs have the same fundamental weakness as other data-parallel accelerators.

ILP accelerators, such as ADRES [16], tackle the problem in another way by exploiting ILP with the help of modulo scheduling. Even though it has high scalability by providing distributed architecture, the throughput quickly saturates as the number of resources increases due to the scheduling difficulty as shown in PPA [21]. Hybrid accelerators such as the Stanford Imagine [7] use the VLIW-SIMD scheme but the fixed configuration frequently incurs a lack or waste of resources.

Recently, several architectures have tried to embrace flexibility in a conventional SIMD accelerator in order to support multiple application domains with different characteristics. AnySP [27] targets mobile applications such as 4G wireless communication and high-definition video coding. AnySP achieves the goal efficiently by simply chaining two SIMD lanes and supporting limited thread level parallelism, but underutilization in low-DLP loops is still inevitable due to the lack of general policy to support ILP. SIMD-Morph [10] employs subgraph matching to accelerate sequential code region. Despite their fair performance gain, their simple ILP/DLP mode transition policy cannot adaptively adjust the degree of ILP and DLP inside a specific code region. For example, it is impossible to fully utilize the SIMD-Morph for a low-DLP code region since an insufficient degree of DLP cannot be supplemented by ILP exploitation, while Libra can. In addition, they are still homogeneous SIMD, and therefore, cannot improve utilization and power efficiency.

TRIPS [25] and SCALE [14] are also similar to this work. TRIPS integrates ILP, DLP and TLP, and SCALE exploits both vector parallelism and TLP. They are targeting more the desktop/server space, and therefore, need expensive architectural features such as inter-cluster networks, additional multiple fetch units, and specialized caches for generality. However, Libra focuses on more efficient execution of loops with minimal hardware modifications.

Avoiding resource contention of expensive instructions by pipelined execution is also introduced in an instruction-systolic array architecture [22]. However, systolic execution may incur severe performance degradation on high number of PEs because of the pipelining delay, while Libra limits sharing only between two logical lanes in full DLP mode.

6. Conclusion

The popularity of mobile computing platforms has led to the development of feature-packed devices that support a wide range of software applications with high single-thread performance and power efficiency requirements. To efficiently achieve both objectives, SIMD-based architectures are currently proposed. However, the SIMD is not able to efficiently support a wide range of mobile applications due to several limiting factors: limited availability of high trip count vector loops and the homogeneous nature of the hardware. To enhance the applicability of SIMD and improve its inherent energy efficiency, we break two long-standing traditions of SIMD design: identical lanes and static configuration. The *Libra* accelerator adapts the SIMD lane resources to target application. The Libra architecture customizes the lane configuration based on the loop structure from many resource-constrained logical lanes for highly data-parallel loops, to a modest number of lanes with moderate resources, up to a single resource-rich logical lane that is effectively a multicluster VLIW. A 32-PE Libra system achieves an average 1.58x speedup over the traditional SIMD system, and the gain becomes higher as the number of PEs increases. Through a judicious mechanism to share expensive resources, Libra also achieves a 29% reduction in energy compared to the SIMD system. We believe that as industry requires higher performance with high energy efficiency, the proposed scalable architecture puts more resources to work in order to meet this demand.

7. Acknowledgments

Thanks to Gaurav Chadha, Anoushe Jamshidi, Dongsuk Jeon and Yoonmyung Lee for all their help and feedback. We also thank Krste Asanovic for shepherding this paper. This research is supported by Samsung Advanced Institute of Technology and the National Science Foundation under grants CCF-0916689 and CNS-0964478.

References

[1] ARM Mali Graphics Hardware - http://www.arm.com/products/multimedia/mali-graphics-hardware/.
[2] Cuda toolkit. - http://developer.nvidia.com/cuda-toolkit.
[3] Glbenchmark - http://www.glbenchmark.com/.
[4] Qualcomm Adreno - http://www.qualcomm.com/solutions/multimedia/graphics/.
[5] Samsung advanced institute of technology - http://www.sait.samsung.co.kr/.
[6] T. Adam, K. Chandy, and J. Dickson. A comparison of list schedules for parallel processing systems. *Communications of the ACM*, 17(12):685–690, Dec. 1974.
[7] J. H. Ahn et al. Evaluating the Imagine stream architecture. In *Proc. of the 31st Annual International Symposium on Computer Architecture*, pages 14–25, June 2004.
[8] M. Alvarez, E. Salami, A. Ramirez, and M. Valero. A performance characterization of high definition digital video decoding using h.264/avc. In *2005 IEEE International Symposium on Workload Characterization*, pages 24–33, Oct. 2005.
[9] H. Bluethgen, C. Grassmann, W. Raab, and U. Ramacher. A programmable platform for software-defined radio. In *Intl. Symposium on System-on-a-Chip*, pages 15–20, Nov. 2003.
[10] G. Dasika, M. Woh, S. Seo, N. Clark, T. Mudge, and S. Mahlke. Mighty-morphing power-simd. In *Proc. of the 2010 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, Oct. 2010.
[11] K. Fan, M. Kudlur, G. Dasika, and S. Mahlke. Bridging the computation gap between programmable processors and hardwired accelerators. In *Proc. of the 15th International Symposium on High-Performance Computer Architecture*, pages 313–322, Feb. 2009.
[12] Intel. Intel compiler, 2009. software.intel.com/en-us/intel-compilers/.
[13] H. Kalva. The H.264 video coding standard. *IEEE MultiMedia*, 13(4):86–90, 2006.
[14] R. Krashinsky, C. Batten, M. Hampton, S. Gerding, B. Pharris, J. Casper, and K. Asanovic. The vector-thread architecture. In *Proc. of the 31st Annual International Symposium on Computer Architecture*, 2004.
[15] Y. Lin et al. Soda: A low-power architecture for software radio. In *Proc. of the 33rd Annual International Symposium on Computer Architecture*, pages 89–101, June 2006.
[16] B. Mei et al. ADRES: An architecture with tightly coupled vliw processor and coarse-grained reconfigurable matrix. In *Proc. of the 2003 International Conference on Field Programmable Logic and Applications*, pages 61–70, Aug. 2003.
[17] NVIDIA. NVIDIA's Next Generation CUDA Compute Architecture: Fermi. http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf.
[18] NVIDIA. GeForce GTX 200 GPU architectural overview, 2008. http://www.nvidia.com/docs/IO/55506/GeForce_GTX_200_GPU_Technical_Brief.pdf.
[19] OpenIMPACT. The OpenIMPACT IA-64 compiler, 2005. http://gelato.uiuc.edu/.
[20] H. Park, K. Fan, S. Mahlke, T. Oh, H. Kim, and H. seok Kim. Edge-centric modulo scheduling for coarse-grained reconfigurable architectures. In *Proc. of the 17th International Conference on Parallel Architectures and Compilation Techniques*, pages 166–176, Oct. 2008.
[21] H. Park, Y. Park, and S. Mahlke. Polymorphic pipeline array: A flexible multicore accelerator with virtualized execution for mobile multimedia applications. In *Proc. of the 42nd Annual International Symposium on Microarchitecture*, pages 370–380, Dec. 2009.
[22] J. Park, H. Yang, G. Park, S. Kim, and C. C. Weems. An instruction-systolic programmable shader architecture for multi-threaded 3d graphics processing. *Journal of Parallel and Distributed Computing*, 70(11):1110–1118, 2010.
[23] B. R. Rau. Iterative modulo scheduling: An algorithm for software pipelining loops. In *Proc. of the 27th Annual International Symposium on Microarchitecture*, pages 63–74, Nov. 1994.
[24] R. M. Russell. The CRAY-1 computer system. *Communications of the ACM*, 21(1):63–72, Jan. 1978.
[25] K. Sankaralingam et al. Exploiting ILP, TLP, and DLP using polymorphism in the TRIPS architecture. In *Proc. of the 30th Annual International Symposium on Computer Architecture*, pages 422–433, June 2003.
[26] S. K. Venkata, I. Ahn, D. Jeon, A. Gupta, C. L. S. Garcia, S. Belongie, and M. B. Taylor. SD-VBS: The san diego vision benchmark suite. In *2009 IEEE International Symposium on Workload Characterization*, pages 55–64, Oct. 2009.
[27] M. Woh, S. Seo, S. Mahlke, T. Mudge, C. Chakrabarti, and K. Flautner. AnySP: Anytime Anywhere Anyway Signal Processing. In *Proc. of the 36th Annual International Symposium on Computer Architecture*, pages 128–139, June 2009.

2012 IEEE/ACM 45th Annual International Symposium on Microarchitecture

Unifying Primary Cache, Scratch, and Register File Memories in a Throughput Processor

Mark Gebhart[1,2] Stephen W. Keckler[1,2] Brucek Khailany[1] Ronny Krashinsky[1] William J. Dally[1,3]

[1]NVIDIA [2]The University of Texas at Austin [3]Stanford University
{mgebhart, skeckler, bkhailany, rkrashinsky, bdally}@nvidia.com

Abstract

Modern throughput processors such as GPUs employ thousands of threads to drive high-bandwidth, long-latency memory systems. These threads require substantial on-chip storage for registers, cache, and scratchpad memory. Existing designs hard-partition this local storage, fixing the capacities of these structures at design time. We evaluate modern GPU workloads and find that they have widely varying capacity needs across these different functions. Therefore, we propose a unified local memory which can dynamically change the partitioning among registers, cache, and scratchpad on a per-application basis. The tuning that this flexibility enables improves both performance and energy consumption, and broadens the scope of applications that can be efficiently executed on GPUs. Compared to a hard-partitioned design, we show that unified local memory provides a performance benefit as high as 71% along with an energy reduction up to 33%.

1. Introduction

Modern GPUs have emerged as an attractive platform for high performance computing. Oriented to throughput processing, GPUs are highly parallel with hundreds of cores and extremely high-bandwidth external memory systems. GPUs employ thousands of chip-resident threads to drive these parallel resources. With so many threads, register files are the largest on-chip memory resource in current GPUs. However, GPUs also provide both scratchpad memories and caches. These local resources provide low latency and high bandwidth access, as well as flexible scatter/gather addressing. In contrast to register files, scratchpad and cache memories allow threads to share data on chip, avoiding costly round trips through DRAM.

Although GPU architectures have traditionally focused primarily on throughput and latency hiding, data locality and reuse are becoming increasingly important with power-limited technology scaling. The energy spent communicating data within a chip rivals the energy spent on actual computation, and an off-chip memory transfer consumes orders of magnitude greater energy than an on-chip access. These trends have made on-chip local memories one of the most crucial resources for high performance throughput processing. As a result, in addition to their large and growing register files, future GPUs will likely benefit from even larger primary cache and scratchpad memories. However, these resources can not all grow arbitrarily large, as GPUs continue to be area-limited even as they become power limited.

Unfortunately a one-size-fits-all approach to sizing register file, scratchpad, and cache memories has proven difficult. To maximize performance, programmers carefully tune their applications to fit a given design, and many of these optimizations must be repeated for each new processor. Even after careful optimization, different programs stress the GPU resources in different ways. This situation is exacerbated as more applications are mapped to GPUs, especially irregular ones with diverse memory requirements.

In this work, we evaluate unified local memory with flexible partitioning of capacity across the register file, scratchpad (shared memory in NVIDIA terminology), and cache. When resources are unified, aggregate capacities can be allocated differently according to each application's needs. This design may at first seem fanciful, as register files have typically had very different requirements than other local memories, particularly in the context of CPUs. However in GPUs, register files are already large, highly banked, and built out of dense SRAM arrays, not unlike typical cache and scratchpad memories. Still, a remaining challenge for unification is that even GPU register files are very bandwidth constrained. For that reason, we build on prior work that employs a two-level warp scheduler and a software-controlled register file hierarchy [8, 9]. These techniques reduce accesses to the main register file by 60%, mitigating the potential overheads of moving to a unified design with shared bandwidth.

Unified memory potentially introduces several overheads. For applications that are already tuned for a fixed partitioning, the main overhead is greater bank access energy for the larger unified structure. Another potential drawback is that with more sharing, unified memory can lead to more bank conflicts. Our analysis shows that even for benchmarks that do not benefit from the unified memory design, the performance and energy overhead is less than 1%.

The unified memory design provides performance gains ranging from 4–71% for benchmarks that benefit from increasing the amount of one type of storage. In addition, DRAM accesses are reduced by up to 32% by making better use of on-chip storage. The combination of improved performance and fewer DRAM accesses reduces energy by up to 33%.

The rest of this paper is organized as follows. Section 2 describes our baseline GPU model. Section 3 characterizes the sensitivity to register file, shared memory, and cache capacity of modern GPU workloads. Section 4 proposes our unified memory microarchitecture. Sections 5 and 6 discuss our methodology and results. Sections 7 and 8 describe related work and conclusions.

2. Background

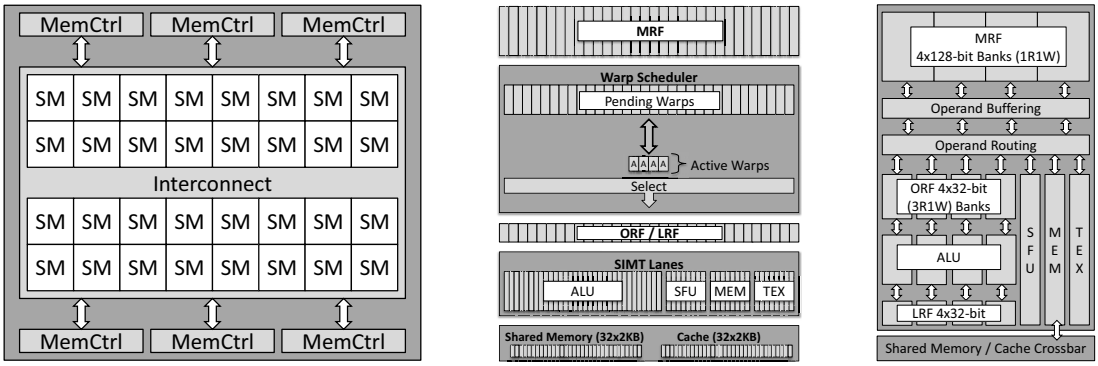
While GPUs are increasingly being used for compute applications, most design decisions were made to provide high graphics performance. Graphics applications have inherent parallelism, with memory access patterns that are hard to capture in a typical CPU L1 cache [7]. To tolerate DRAM latency and provide high performance, GPUs employ massive multithreading. Additionally, programmers can explicitly manage data movement into and out of low-latency on-chip scratchpad memory called *shared memory*.

Figure 1 shows the design of our baseline GPU, which is loosely modeled after NVIDIA's Fermi architecture. The figure represents a generic design point similar to those discussed in the literature [2, 16, 25], but is not intended to correspond directly to any existing

This research was funded in part by the U.S. Government. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government.

1072-4451/12 $26.00 © 2012 IEEE
DOI 10.1109/MICRO.2012.18

IEEE computer society



(a) Full chip (b) Streaming multiprocessor (c) 4-Wide SIMT lane detail

Figure 1: Baseline GPU architecture.

industrial product. The GPU consists of 32 streaming multiprocessors (SMs) and 6 high-bandwidth DRAM channels for a total of 256 bytes/cycle of DRAM bandwidth. Figure 1b shows an SM containing 32 SIMT (single-instruction, multiple thread) lanes that each execute up to one thread instruction per cycle. A group of 32 threads form an execution unit called a *warp*. The SIMT model executes all threads in a warp together using a common physical program counter. While the hardware supports control-flow divergence of threads within a warp, the SM operates most efficiently when all threads execute along a common control-flow path. Warps are grouped into larger units called co-operative thread arrays (CTAs) by the programmer. Threads in the same CTA execute on the same SM and can communicate through shared memory. A program may consist of one or more *kernels*, each consisting of one or more CTAs.

2.1. Baseline SM Architecture

In this work, we focus on the design of the SM shown in Figures 1b and 1c. The SM has up to 1024 resident threads, and a 32-entry, single-issue, in-order warp scheduler selects one warp per cycle to issue an instruction. Each SM provides 64KB of local scratchpad storage known as shared memory, 64KB of cache, and a 256KB register file. While these are large capacity structures compared to a uniprocessor, the SM provides on average only 256 bytes of registers, 64 bytes of data cache, and 64 bytes of shared memory per thread. Figure 1c provides a detailed microarchitectural illustration of a cluster of 4 SIMT lanes. A cluster is composed of 4 ALUs, 4 register banks, a special function unit (SFU), a memory unit (MEM), and a texture unit (TEX) shared between two clusters. Eight clusters form a complete 32-wide SM.

We leverage prior work which introduced a two-level warp scheduler and a software controlled register file hierarchy [8, 9]. The two-level warp scheduler divides the 32 warps present on an SM into an active set and an inactive set. Only warps in the active set are allowed to issue instructions, and warps are moved to the inactive set when they encounter a dependence on a long-latency operation. The software controlled register file hierarchy introduces two additional levels beyond the main register file (MRF): the operand register file (ORF) with 4 entries per thread, and a last result file (LRF) with a single entry per thread. Only active warps can allocate values in the ORF and LRF. When an active warp is descheduled, all of its live values must be in the MRF. The compiler controls all data movement between the MRF, ORF, and LRF. The result of these prior techniques is a reduction in the number of accesses to the MRF of 60%, without a performance loss, resulting in a significant savings in register file energy and MRF bandwidth.

Each MRF bank is 16 bytes wide with 4 bytes allocated to the same-named architectural register for threads in each of the 4 SIMT lanes in the cluster. Each bank has a capacity of 8KB, providing a total of 256KB of register file capacity per SM. Registers are interleaved across the register file banks to minimize bank conflicts. Instructions that access multiple values from the same bank incur a cycle of delay for each access beyond the first. The operand buffering between the MRF and the execution units represents interconnect and pipeline storage for operands that may be fetched from the MRF on different cycles. Stalls due to bank conflicts are rare and can be minimized with compiler techniques [27].

Each SM contains 64KB of cache and 64KB of shared memory. Each of these structures is composed of 32 2KB banks, and each bank supports one 4-byte read and one 4-byte write per cycle. The cache uses 128-byte cache lines which span all 32 banks, and only supports aligned accesses with 1 tag lookup per cycle. Shared memory supports scatter/gather reads and writes, subject to the limitation of one access per bank per cycle. Avoiding shared memory bank conflicts is a common optimization employed by programmers. The cache and shared memory banks are connected to the memory access units in the SM clusters through a crossbar.

2.2. Unified Cache and Shared Memory

Fermi has a unified cache and shared memory, providing programmers a limited choice of either a 16KB cache and a 48KB shared memory or a 48KB cache and a 16KB shared memory [16]. The memory configuration is controlled through a CUDA library function. Section 6.3 shows that a limited form of flexibility across shared memory and cache, like that found in Fermi, has benefits. However, a more flexible solution across all three types of storage (register file, cache, and shared memory) further improves both performance and energy consumption.

3. Application Characterization

In this section, we characterize modern GPU applications based on their usage of registers, shared memory, and cache. We begin with a large number of benchmarks and show that modern workloads fall into several different categories. Next, we explain in detail why some applications benefit from larger capacity in a given type of storage. Finally, we study the performance sensitivity of applications to storage capacity.

Table 1: Workload characteristics.

Workload	Registers per thread (no spills)	Registers per Thread (normalized dynamic instructions) 18	24	32	40	64	RF Size full occupancy no spills (KB)	Shared Memory (bytes / thread)	Cache Size (normalized DRAM Accesses) 0	64KB	256KB
Shared Memory Limited											
Needle [3]	18	1.02	1	1	1	1	72	264.1	0.85	1	1
sto [2]	33	1.18	1.08	1	1	1	132	127	3.95	1	1
lu [3]	20	1	1	1	1	1	80	96	1.94	1.46	1
Cache Limited											
GPU-mummer [3]	21	1.04	1	1	1	1	84	0	1.48	1.01	1
BFS [3]	9	1	1	1	1	1	36	0	1.46	1.13	1
Backprop [3]	17	1.02	1	1	1	1	68	2.125	1.56	1	1
MatrixMul [15]	17	1.04	1	1	1	1	68	8	4.77	1	1
Nbody [15]	23	1	1	1	1	1	92	0	3.52	1	1
VectorAdd [15]	9	1	1	1	1	1	36	0	3.88	1	1
srad [3]	18	1	1	1	1	1	72	24	1.22	1.20	1
Register Limited											
DGEMM [11]	57	1.42	1.23	1.01	1	1	228	66.5	1	1	1
PCR [26]	33	1.39	1.18	1.03	1	1	132	20	2.88	1.29	1
BicubicTexture [15]	33	1.18	1.10	1.05	1	1	132	0	1	1	1
hwt [3]	35	1.04	1.04	1.04	1	1	140	23	1	1	1
ray [2]	42	1.18	1.11	1.08	1.05	1	168	0	1.02	1.07	1
Balanced / Minimal Capacity Requirements											
Hotspot [3]	22	1.21	1	1	1	1	88	12	1.44	1	1
RecursiveGaussian [15]	23	1.02	1	1	1	1	92	2.125	1.04	1.03	1
Sad [17]	31	1.01	1	1	1	1	124	0	1.01	1.01	1
ScalarProd [15]	18	1.01	1	1	1	1	72	16	1	1	1
SGEMV [11]	14	1	1	1	1	1	56	4	1.01	1.01	1
SobolQRNG [15]	12	1	1	1	1	1	48	2	1	1	1
aes [2]	28	1.30	1.18	1	1	1	112	24	1	1	1
Dct8x8 [15]	26	1.16	1.10	1	1	1	104	0	1	1	1
DwtHaar1D [15]	14	1	1	1	1	1	56	8	1	1	1
lps [2]	15	1	1	1	1	1	60	19	1.48	1	1
nn [2]	13	1	1	1	1	1	52	0	20.81	1.07	1

3.1. Workload Characterization

We characterize these applications along three axes:

- Register usage: Two parameters are related to register file capacity: registers per thread and number of threads. Each thread is allocated registers for thread private values, with the same number of registers allocated for every thread in a kernel. Modern GPUs support a very large number of registers per thread. However, using more registers per thread results in fewer threads per SM, as the register file is shared across the SM. The compiler inserts spill and fill code when there are not enough registers available. We use the number of dynamic instructions executed as a metric to measure the overhead of register spills.
- Shared memory usage: Shared memory tradeoffs are controlled by the programmer, with each kernel specifying the total shared memory required per CTA along with the number of threads per CTA. The physical shared memory capacity available in an SM then dictates the maximum number of CTAs that can be mapped, if the register file capacity does not become a bottleneck first. While the programmer can often adjust shared memory requirements by changing an application's blocking pattern, we evaluate existing benchmarks that have fixed shared memory requirements per thread. Section 6.5 discusses tuning the shared memory requirements to exploit the unified design.
- Cacheable memory usage: The amount of spatial and temporal locality varies from application to application. Streaming applications mainly have spatial locality, but often have some degree of access redundancy which can be filtered by a small cache. Applications with cache blocking or a large number of register spills have higher temporal locality. The cache is a very scarce resource, and our baseline configuration has only 64 bytes on a per-thread basis. We use the number of DRAM accesses as a metric for the cache's effectiveness.

Table 1 presents an analysis of a range of CUDA applications according to the above criteria. Columns 2–8 show the per-thread register requirements, with column 2 showing the number of registers per thread required to eliminate spills. Columns 3–7 show the increase in dynamic instructions due to spill and fill code with different numbers of registers per thread. All of our surveyed benchmarks experience no spills when 64 registers per thread are available. Hand tuned programs tend to use more registers per thread than compiled programs as the programmer can block data into the register file for higher performance. `DGEMM`, `PCR`, and `BicubicTexture` all experience a large number of spills with a small number of registers per thread. Column 8 shows the register file capacity required to achieve full occupancy without experiencing register spills. The capacity required ranges from 36KB to 228KB. Column 9 shows the number of bytes of shared memory required per thread. Many applications need less than 20 bytes per thread, particularly when developed to fit the small shared memory capacities of early GPUs. `Needle` on the other hand, requires a large amount of shared memory. Columns 10–12 show the number of DRAM accesses for different capacity primary data caches. In general, as the cache capacity is increased, DRAM ac-

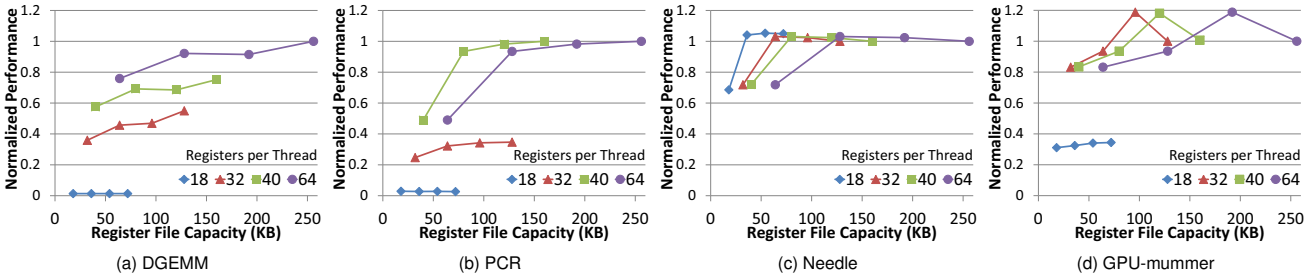


Figure 2: Performance as a function of register file capacity (with 64KB cache and unbounded shared memory), normalized to 64 registers per thread and 1,024 threads per SM.

cesses decrease. This decrease in DRAM traffic is due to the cache's ability to filter traffic and amplify bandwidth. The DRAM bandwidth demand can actually go up when using a cache, particularly when the cache line size exceeds the minimum DRAM fetch size. For example, `Needle` fetches unneeded data because only a fraction of the cache line is used after fetch.

Table 1 demonstrates that different applications place different stresses on the register file, shared memory, and cache structures. Many of the benchmarks fall into the balanced / minimal capacity requirements category as they were developed to fit the design of existing GPUs. As new emerging applications are ported to GPUs and applications are optimized to take advantage of our unified design, we expect to see more diversity in the memory requirements.

3.2. Application Case Studies

To provide greater insight into the advantages of the unified design, we discuss in detail the benchmarks which see a significant performance benefit from higher capacity in a given type of storage.

`Needle` implements the Needleman-Wunsch algorithm for DNA sequence alignment using dynamic programming [3]. The algorithm constructs a large (2048 by 2048 entry) matrix where each entry depends on its north, west, and north-west neighbor. The problem is broken into subblocks to make use of shared memory. The size of the subblock is a key parameter for this algorithm. Larger subblocks improve performance, but increase the shared memory requirements quadratically. Section 6.5 discusses the choice of blocking factor in more detail.

`LU` performs LU decomposition to solve a set of linear equations [3]. The kernel requires a moderate amount of registers but a high capacity shared memory. A large cache can exploit the reuse patterns as values in the input matrix are accessed repeatedly.

`GPU-mummer` implements DNA sequence alignment using graph traversal [3]. The algorithm consist of many parallel graph traversals across a large reference suffix tree. Each thread processes a single independent query. This workload does not use shared memory, as the working set size depends on the input. If the reference suffix tree is cached, a large performance gain is possible.

`BFS` is a breadth-first search of a graph with one million nodes [3]. It does not make use of shared memory and uses a small number of registers per thread. The application benefits from caching as the node and edge list is accessed repeatedly.

`SRAD` is an image processing application that relies on partial differential equations [3]. It uses a moderate number of registers and shared memory per thread, but benefits greatly from a large primary cache. Each output element is computed based on its four neighbors, allowing the cache to filter DRAM accesses.

`DGEMM` is an optimized double precision matrix multiplication kernel from the MAGMA library [11]. Two temporary matrices in shared memory capture subblock temporal locality. There is little performance benefit from caching. Each thread requires 57 registers per thread to eliminate spills, requiring a large register file.

`PCR` is a parallel cyclic reduction kernel that solves a tridiagonal linear system [26]. The algorithm uses shared memory to store temporary data and streams a large dataset from global memory. The large amount of communication between steps of the algorithm requires high bandwidth access to shared memory.

`RAY` performs ray-tracing with each thread responsible for rendering a single pixel; several levels of reflections and shadows are modeled. The kernel does not use shared memory but does require a large number of registers. A larger data cache is able to capture the environment, reducing the number of DRAM accesses.

3.3. Performance Sensitivity Study

Finally, we explore the performance sensitivity to the capacity of the register file, shared memory, and cache. We present limit studies which highlight the diverse memory requirements of modern workloads and the performance gains that can be achieved with larger storage structures. The details of our evaluation methodology are in Section 5. Because of the large number of benchmarks that we characterize in Table 1, we only present results for a subset of benchmarks which exhibit unique behaviors across the three different types of on-chip storage.

3.3.1. Register File Capacity: Register file capacity is a function of both the number of registers allocated to each thread and the number of concurrent threads. Performance is penalized when the number of registers per thread is small, which results in a large number of spills and fills. Likewise, applications that must tolerate DRAM accesses experience performance degradations when the number of concurrent threads is small.

Figure 2 illustrates the relationship between performance and register file capacity for four different types of applications. Each line in the graph shows performance with a different number of registers per thread. The performance penalty of spills can be seen by comparing the four lines. The points on a given line show performance for 256, 512, 768, and 1024 threads per SM. `DGEMM` requires both a large number of registers per thread and a large number of threads to maximize performance. These types of applications that require both a large number of registers per thread and a large number of concurrent threads stress the capacity of the register files found on current GPUs. `PCR` experiences a large number of spills with 18 or 32 register per thread and is less sensitive to thread count than `DGEMM`. There is no advantage to using more registers per thread than is necessary to eliminate spills. `Needle` is an example of an application that eliminates

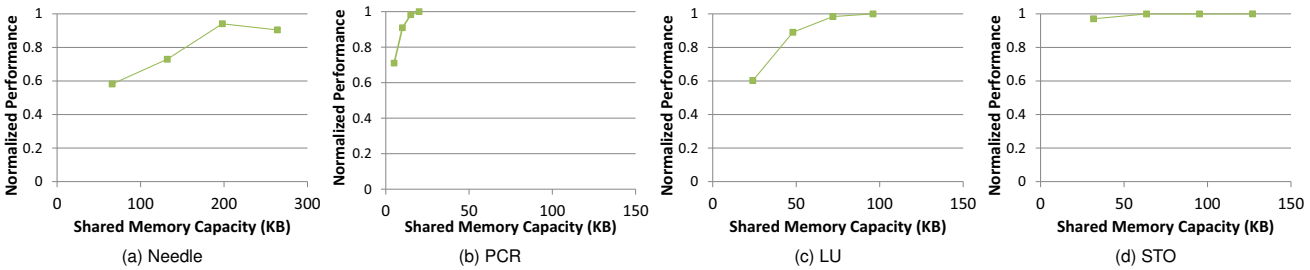


Figure 3: Performance versus shared memory capacity (with 64 registers per thread and 64KB of cache), normalized to 1,024 threads per SM. Note the wider x-axis on Needle.

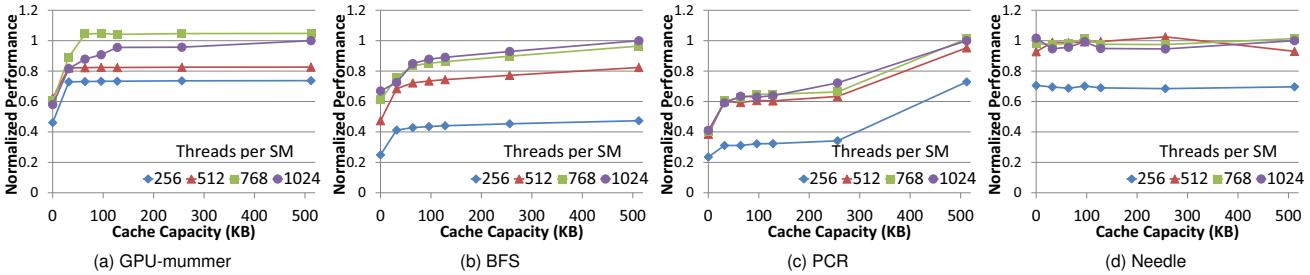


Figure 4: Performance as a function of cache capacity (with 64 registers per thread and unbounded shared memory), normalized to 512KB cache and 1,024 threads per SM.

spills even with as few as 18 registers per thread. Further, increasing thread count beyond 512 threads does not increase performance. DRAM latency tolerance is not important for this application, as it operates mainly out of shared memory. The spikes in performance in Figure 2d result from the interaction between the cache size and thread count. Changing the thread count can change performance due to interactions with the thread scheduler, especially when the larger number of threads are not needed to tolerate DRAM latency.

3.3.2. Shared Memory Capacity: Figure 3 shows the tradeoff in performance and shared memory capacity. Each point along a line shows an increasing number of threads per SM ranging from 256 to 1,024 in increments of 256. To isolate the effects of shared memory, these experiments use a large register file, eliminating register spills, and a 64KB cache. The application with the largest shared memory needs is `Needle`, which requires over 200KB. We discuss alternate blocking factors that can be used for `Needle` in Section 6.5. The shared memory usage of `PCR` is typical of today's applications. There is a large performance gain from maximizing thread count and even with the maximum number of threads per SM only 20KB of shared memory is required. `LU` is an example of an application that requires more shared memory than is present on today's GPUs and maximizing thread count improves performance. `STO` is an example where the application operates primarily out of shared memory, reducing the importance of running a large number of threads to tolerate DRAM latency. A small number of threads can still achieve high performance and minimizes the shared memory requirements.

3.3.3. Cache Capacity: Figure 4 shows the performance sensitivity to cache capacity. In these graphs, each line shows a different number of threads per SM (ranging from 256 to 1,024), and each point along a line shows performance with a different cache capacity. To isolate the effects of cache capacity, the register file is sized to eliminate spills and shared memory is unbounded. Running more threads per SM helps to tolerate latency from main memory access, but also reduces the amount of cache available on a per-thread basis. `BFS` and `PCR` benefit from having a large cache. In particular, `PCR` sees a large performance benefit moving from a 256KB to 512KB cache. `GPU-mummer` sees a performance benefit from caching, but it has a small working set for the input datasets we used. We expect a greater improvement with larger datasets. `Needle` is an example of an application that sees little performance benefit from caching as it operates mostly out of shared memory.

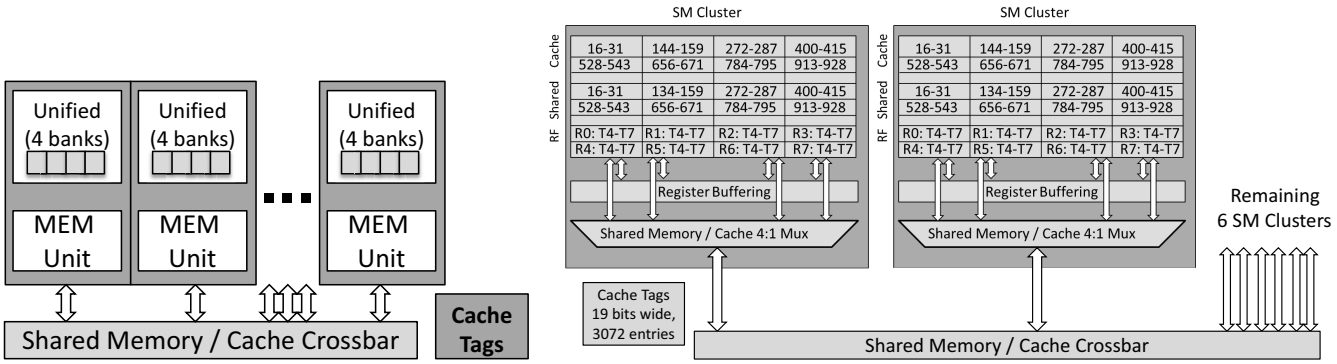
4. Microarchitecture

The characterization in Section 3 shows that modern GPU workloads have diverse local storage requirements and a single resource is often most critical to performance of a given application. We propose a *unified memory* architecture that aggregates these three types of storage and allows for a flexible allocation on a *per-kernel basis.*

4.1. Overview

Figures 5a and 6 compare the microarchitectures of the baseline design and our proposed unified architecture. The baseline design is structured as discussed in Section 2.1. In the unified design, all data storage is moved into the SM clusters. Effectively, the unified design merges together the 32 MRF banks, 32 shared memory banks, and 32 cache banks. Although we evaluate a range of capacities, the number of unified banks is always 32 per SM, to keep bandwidth constant. Each unified bank supports 1 read and 1 write per cycle, as do the banks in the baseline design. Also similar to the baseline design, the SM clusters in the unified design are connected by a crossbar to transfer data between the memory access units and other SM clusters.

As with the partitioned design, the cache tags are stored outside the SM clusters and 1 tag lookup can be processed per cycle. A 384KB unified design requires up to 7.125KB of tag storage compared with the baseline 64KB cache requiring 1.125KB. This overhead can be reduced by limited the maximum cache size in the unified design.



(a) Unified SM architecture, 3 of 8 SM clusters shown.

(b) Detailed address mapping: RF: thread ID / register ID, Shared Memory/Cache: address (bytes).

Figure 5: Proposed unified memory microarchitecture.

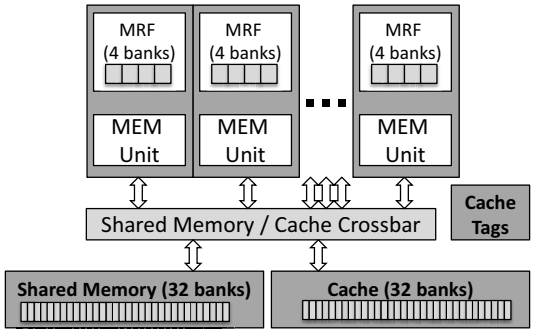


Figure 6: Baseline SM architecture, 3 of 8 SM clusters shown.

4.2. Unified Memory Bank Design

Each unified memory bank is 16 bytes wide with byte-enable support. Figure 5b shows how registers, cache, and shared memory are mapped across the banks. This figure shows 2 of the 8 SM clusters found in one SM. Threads from a single warp are mapped evenly across the 8 SM clusters, with 4 threads executing on each SM cluster. As all of a thread's register file entries are located in the same SM cluster where it executes, register file values are not communicated between SM clusters. The unified memory architecture does not change the register file bank mapping, bank widths, or register muxing in any way. The cache line size is 128 bytes in both the partitioned and unified designs. As shown in Figure 5b, the cache line is address-partitioned across 8 of the unified banks, 1 from each of the SM clusters. The shared memory address space is mapped across the banks in a similar manner. The unified memory design uses a smaller number of larger memory banks. The banks are sized such that the increase in bank size does not result in additional cycles required for bank access. The bank access is not on the processor's critical path, allowing for a larger memory bank.

When accessing cache or shared memory, only a single bank is used from each of the 8 clusters. This single bank routes its 16 bytes of data onto the crossbar, providing a peak shared memory or cache bandwidth of 128 bytes per cycle, identical to the baseline partitioned design. Compared with the partitioned design, the unified design adds one level of additional muxing for shared memory and cache accesses across SM clusters. Section 5 describes how we account for the extra wiring energy to access this multiplexor. This 4 to 1 mux is used to select which bank should access the crossbar and is only traversed for remote memory traffic, not for register file accesses. However, this single bank per cluster design is more restrictive than the partitioned design. To be bank-conflict free, a warp's shared memory accesses must coalesce to 8 banks rather than 32. A more aggressive design allows multiple banks in a single cluster to be accessed to increase the scatter / gather bandwidth. This enhanced design increases the complexity of the data muxing in a cluster, but still only allows 16 bytes per cluster to enter the crossbar. We simulated this design and found that it had an average performance improvement of 0.5%, compared to the simpler design. Our results in Section 6 assume the simpler design.

4.3. Arbitration Conflicts

In our baseline design, bank conflicts only occur within a single type of storage. With the unified design, accesses to the register file and cache or shared memory can conflict with each other. We refer to these conflicts as arbitration conflicts. One of the key enablers of the unified design is the software controlled register file hierarchy, which fetches most operands from the ORF or LRF and greatly reduces the required bandwidth to the MRF [9]. We model all conflicts and give priority to register access before cache or shared memory, but find that the performance impact of conflicts is small. Memory instructions fetch a small number of register operands and these operands often come from the LRF or ORF rather than the MRF, minimizing the number of arbitration conflicts. Our design uses a write through cache, eliminating bank accesses for evicting dirty data. The large number of threads can also tolerate some additional latency from conflicts without harming performance.

4.4. Managing Partitioning

Modern GPU workloads typically contain several different kernels, each of which may have different memory requirements. Before each kernel launch, the system can reconfigure the memory banks to change the memory partitioning. Because the register file and shared memory are not persistent across CTA boundaries, the only state that must be considered when repartitioning is the cache. As we use a write-through cache, the cache does not contain dirty data to evict. In the applications that we evaluated, the memory requirements across kernels were similar. Therefore, the results in Section 6 reflect choosing a single memory partitioning at the start of each benchmark and not reconfiguring the partitioning for each kernel.

4.5. Allocation Decisions

The unified memory architecture requires the programming system and hardware to determine the capacity of the register file, shared memory, and cache. We use the following automated algorithm to calculate the storage partitioning evaluated in Section 6:

- **Register File:** The compiler calculates how many registers per thread are required to avoid spills (Table 1, column 2).
- **Shared Memory:** The programmer specifies the amount of shared memory required per thread when constructing each kernel in the same manner as today's partitioned designs.
- **Thread count:** The hardware scheduler takes as input the number of registers per thread to avoid spills, the number of bytes of shared memory, and the overall capacity of the unified memory. The scheduler calculates the maximum number of threads by dividing the unified memory capacity by the per-thread register and shared memory requirements. Some applications see higher performance with fewer than the maximum number of threads, due to interactions with the thread scheduler and memory system. This phenomenon occurs both for the partitioned and unified design. Techniques like autotuning [24] can be used to automatically optimize thread count.
- **Cache:** Any remaining storage is allocated to the primary data cache.

5. Methodology

When possible, we used the default input sets and arguments distributed with the benchmarks described in Table 1, but we scaled down some of the workloads to make the simulation time tractable.

5.1. Simulation

We used Ocelot, a PTX dynamic compilation framework, to create execution and address traces [6]. We built a custom SM simulator that takes these traces as input and measures performance. We simulate execution using the SM parameters shown in Table 2. Our SM simulator runs the traces to completion, correctly modeling synchronization between threads in a CTA. We model execution for the full application running on a single SM and allocate 8 bytes per cycle of DRAM bandwidth making the simplifying assumption that the global DRAM bandwidth is evenly shared among all 32 SMs. Because the applications run each kernel many times across a large number of threads, modeling a single SM, rather than the full chip, simplifies simulation without sacrificing accuracy.

5.2. Energy Model

We assume a 32nm technology node for our energy evaluation using the parameters listed in Table 3 and focus on the following elements which are affected by our unified design:

- Bank Access Energy: Compared with the baseline partitioned design, the unified design uses a smaller number of larger banks, resulting in more energy per access to the main register file, shared memory, and cache. Table 4 shows dynamic read and write energy for SRAM banks of various sizes. These numbers are scaled using a combination of CACTI [13] and prior work that used synthesis for memory structures [8]. While the unified design increases bank access energy, especially for shared memory and cache accesses, Section 6 shows that this increase is small in comparison to total system energy.

Table 2: Simulation parameters.

Parameter	Value
SM Execution Width	32-wide SIMT
SM Execution Model	In-order
SM Register File Capacity	256 KB
SM MRF Bank Capacity	8 KB
SM Shared Memory Capacity	64 KB
SM Shared Memory Bandwidth	128 bytes/cycle
SM Cache Associativity	4-way
SM DRAM Bandwidth	8 bytes/cycle
ALU Latency	8 cycles
Special Function Latency	20 cycles
Shared Memory Latency	20 cycles
Texture Instruction Latency	400 cycles
DRAM Latency	400 cycles

Table 3: Energy parameters.

Parameter	Value
Technology node	32 nm
Frequency	1 GHz
Voltage	0.9 V
Wire capacitance	300 fF / mm
Wire energy	1.9 pJ / mm
Dynamic power per SM	1.9 W
Leakage power per SM	0.9 W
Leakage power per KB of SRAM	2.37 mW
DRAM energy	40 pJ / bit

- Wiring Energy: In the baseline design, the cache and shared memory banks are 4 bytes wide. In the unified design, the banks are 16 bytes wide. To simplify the crossbar, we stripe cache lines across banks in different SM clusters as described in Section 4. However, we still incur an overhead of a 4:1 multiplexer. Furthermore, for an equal capacity design, the area of an SM cluster will increase as cache and shared memory storage is moved into the clusters. This increase in area will increase the overhead of the crossbar that connects the clusters. As we have not implemented a detailed physical design, we model these overheads as 10% additional energy relative to the bank access energy for cache and shared memory reads and writes. We also account for an increase in cache tag lookup energy with this factor.
- SRAM Leakage: Each of the unified and partitioned designs use different amounts of SRAM for the main register file, shared memory, and cache. We use an estimate of 2.37 mW per KB of SRAM capacity from prior work to calculate leakage for each design [10].
- DRAM Energy: Our architecture reduces DRAM accesses by making better use of on-chip memory. We model each DRAM access as consuming 40 pJ/bit [22].

We use a high-level GPU power model to account for the core dynamic and leakage energy. We assume a modern GPU in 32nm process technology that consumes 130 watts and contains 32 SMs. The SMs consume 70% of the chip-wide energy, with the remaining 30% consumed by the memory system. Assuming that leakage is one third of the chip-wide power, each SM consumes 1.9 watts of dynamic power and 0.9 watts of leakage power. Except for bank access and DRAM energy, we assume that dynamic power for the SM is constant across the various configurations. We use the performance of the baseline 256/64/64 configuration to calculate SM dynamic power for each benchmark. We adjust leakage for each configuration using the SRAM leakage data of 2.37mW per KB of capacity. On the

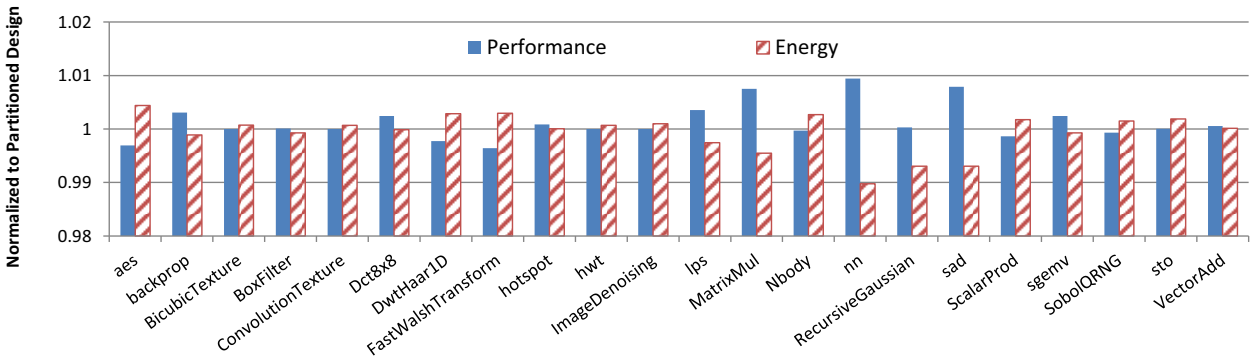


Figure 7: Performance (higher is better) and energy (lower is better) of unified design (384KB) normalized to an equal-capacity partitioned design for applications that do not benefit from unified storage (note the narrow range of the y-axis).

Table 4: Energy for 16-byte SRAM bank access (32nm) for unified and partitioned designs.

Structure	Bank Size	Read (pJ)	Write (pJ)
Partitioned			
256KB RF	8 KB	9.8	11.8
64KB Shared Memory	2 KB	3.9	5.1
64KB Cache	2 KB	3.9	5.1
Unified			
384KB Unified	12 KB	12.1	14.9

baseline design with 384KB of SRAM storage, 0.7 watts of leakage is from the core and 0.2 watts is from the SRAM. The SM and SRAM leakage energy is calculated for each design point based on performance. Design points with higher performance experience less leakage, since faster completion of the workload results in less time for transistors to leak.

6. Results

In this section, we evaluate the overheads and advantages of the unified memory design. We divide the benchmarks that we characterized in Section 3 into two sets: those that see no benefit from the unified design (Section 6.1) and those that benefit (Section 6.2).

6.1. Applications With No Benefit From Unified Memory

First, we evaluate the set of benchmarks that do not benefit from the unified design. These benchmarks are not able to make use of the additional capacity provided by the unified design to improve performance. However, the unified design does not harm performance or energy. Many of these benchmarks were tuned for the small capacity structures present on early GPUs and may benefit from the unified design if they were tuned for larger capacity structures.

Figure 7 shows the performance and energy improvements of a 384KB unified design normalized to an equal-capacity partitioned design. Each SM in this baseline partitioned design contains a 256KB register file, a 64KB shared memory, and a 64KB primary data cache as described in Section 2. The unified design only slightly changes performance and energy for these benchmarks, with the largest changes less than 1%. The slight changes in performance and energy are mainly due to (1) changes in bank conflicts resulting from changing the bank width from 4 bytes in the partitioned case to 16 bytes in the unified case, and (2) from bank conflicts associated with combining the register file with shared memory and the cache. These results show that the performance degradation due to an increase in bank conflicts is negligible.

One of the potential overheads of the unified design is an increase in memory bank conflicts, as each memory bank supports only one read and one write operation per cycle. Bank conflicts are due to accesses from the same instruction or different instructions mapped to the same bank. The inter-instruction conflicts depend on the exact scheduling policy and instruction pipeline used. To get an estimate of the potential increase in bank conflicts from unified memory, we rely on a simplified model where we only track conflicts within a single warp instruction. For each warp instruction, we count the bank accesses across the 32 threads in the warp. We then impose a performance penalty of 1 cycle for each access beyond the first for the bank that was accessed the most by that warp instruction. For example, if one bank was accessed three times and another bank was accessed twice the instruction would be delayed by 2 cycles. This model is likely pessimistic, as accesses from a single warp instruction can actually span different clock cycles due to the pipeline design. In the partitioned design, bank conflicts occur (1) in the register file when an instruction tries to read multiple registers mapped to the same bank, and (2) in the cache and shared memory when threads in the same warp access values that are mapped to the same bank. In the unified design, additional arbitration conflicts occur when an instruction tries to read or write a value from the cache or shared memory that is mapped into the same bank as its register operands.

Table 5 quantifies the potential increase in bank conflicts by showing how many accesses each warp instruction makes to the same memory bank. In both designs, the vast majority of warp instructions make one or fewer accesses to each memory bank. The unified design experiences a small increase (0.6 percentage points) in the number of warp instructions that access a bank multiple times. However, Figure 7 shows this increase in accesses leads to a negligible performance change. The key enabler that allows the unification of on-chip memory without excessive numbers of arbitration conflicts is the register file hierarchy, which dramatically reduces the number of accesses to the main register file [9].

Relative to the partitioned architecture, the unified memory design slightly increases bank access energy due to its smaller number of banks, each with higher capacity. However, bank access energy

Table 5: Breakdown of warp instructions based on the maximum number of accesses to a single bank for the unified and partitioned design, averaged across Figure 7 benchmarks.

	Maximum accesses to a single bank per instruction				
	<= 1	2	3	4	>4
Partitioned	97.0%	2.7%	0.09%	0.14%	0.03%
Unified	96.4%	3.4%	0.01%	0.02%	0.21%

makes up a small component of overall system energy. Figure 7 shows that the overall changes in energy are negligible. The largest increase in energy is 0.9% for `nn` and on average the energy of the unified design is 0.06% lower than that of the partitioned design. Much of the energy spent in the register file system, cache, and shared memory is for control and wiring rather than actual bank access. Additionally, the register file hierarchy reduces the number of accesses to the main register file, minimizing register file bank access energy for both the partitioned and unified designs. These results show that even though these benchmarks do not benefit from the unified design the overhead from our proposed design is negligible.

6.2. Applications That Benefit From Unified Memory

Next, we evaluate benchmarks that see significant improvements from the unified memory architecture. We have made no source code modifications to these benchmarks to tune them for the unified memory architecture. As the analysis in Section 3 shows, modern applications have a variety of requirements in on-chip storage needs and the unified memory architecture is able to adapt on a per-application basis with the most efficient partitioning of on-chip storage.

As described in Section 4.5, the allocation decisions are managed automatically by the compiler and hardware. Figure 8 shows how the 384KB of unified memory was configured for each of these benchmarks. The amount of storage devoted to the register file ranges from 36KB on `bfs` to 228KB on `dgemm`. One of the applications, `needle`, devotes 264KB to shared memory to allow a larger number of concurrent threads to execute. The remaining applications that make use of shared memory devote less than 100KB of their on-chip storage to it. The unified memory design allows larger primary caches, as any remaining storage not used for the register file or shared memory serves as cache.

Figure 9 shows the performance, energy, and DRAM traffic improvements for eight benchmarks that see significant improvements. The performance improvements range from 4.2% to 70.8% with an average performance improvement of 16.2%. These performance improvements are the result of a combination of having a larger register file, shared memory, or cache. In many cases, the larger capacity register file or shared memory allows more concurrent threads to run, which allows the SM to better tolerate DRAM latency.

All of the benchmarks, except for `DGEMM`, see a reduction in DRAM traffic ranging from 1% to 32%. The reduction in DRAM accesses is primarily the result of having higher capacity caches. As DRAM bandwidth is and will continue to be a precious resource, minimizing off-chip traffic is vital to improving efficiency. The performance improvements along with the reduction in DRAM accesses lead to a reduction in chip-wide energy. The energy savings range from 2.8% to 33% across these eight applications. These savings are significant for today's power limited devices.

6.3. Comparison to Limited Unified Memory

As discussed in Section 2.2, Fermi has a limited form of unified memory. The programmer can choose between either 16KB of shared memory and 48KB of cache or 48KB of shared memory and 16KB of cache per SM. Our unified design allows all three types of storage found in the SM to be unified. We evaluate an equal-capacity Fermi-like design which has a total of 384KB of storage divided into a 256KB register file and either 96KB of shared memory and 32KB of cache or 32KB of shared memory and 96KB of cache. Figure 10 shows the improvement in performance, energy, and DRAM accesses compared to the partitioned design.

Table 6: Performance and energy of three unified memory capacities normalized to the the baseline partitioned design.

	Performance (higher is better)			Energy (lower is better)		
Benchmark	128KB	256KB	384KB	128KB	256KB	384KB
bfs	1.03	1.08	1.12	0.91	0.89	0.88
dgemm	0.77	1.01	1.08	1.13	0.95	0.94
lu	0.96	1.07	1.07	1.00	0.91	0.89
GPU-mummer	0.96	1.04	1.04	0.97	0.95	0.97
pcr	0.77	1.04	1.06	1.33	0.92	0.93
ray	0.94	1.03	1.13	1.01	0.95	0.89
srad	1.00	1.08	1.09	0.94	0.86	0.89
needle	1.29	1.75	1.71	0.76	0.64	0.67
Average	0.97	1.14	1.16	0.98	0.87	0.87
Figure 7 Benchmarks (Average)	**0.99**	**1.00**	**1.00**	**0.93**	**0.96**	**1.00**

The Fermi-like design is able to improve performance for all of the benchmarks between 1%–20%. However, comparing Figures 9 and 10 shows that the unified design achieves higher performance for all but one benchmark. The Fermi-like design actually achieves higher performance on `GPU-mummer` because this benchmark is extremely sensitive to cache size and thread scheduling. The smaller capacity cache provided by the Fermi-like design results in slightly different cache behaviors that interact differently with the thread scheduler. Overall, the gains in energy-efficiency and DRAM traffic reduction are higher for the unified architecture than the Fermi-like limited flexibility design.

6.4. Capacity Sensitivity

Next, we explore the sensitivity of performance and energy to the capacity of the unified memory. Larger unified memory designs improve performance at the cost of increased SRAM leakage. Table 6 shows performance and energy for a range of different unified memory capacities. Performance is generally maximized with 384KB of unified memory. `Needle` sees slightly higher performance with 256KB due to the choice of thread count and the resulting scheduling decisions made by the thread scheduler. The performance of the benchmarks from Figure 7 is flat across the range of capacities as they do not see a speedup from the larger capacity designs.

As the capacity of unified memory is increased, SRAM leakage increases. However, a larger capacity design can also reduce overall leakage (higher performance reduces runtime) and DRAM energy. The benchmarks that do not benefit from unified memory see the lowest energy with the 128KB design, which minimizes SRAM leakage. The benchmarks that benefit from the unified design generally see the lowest energy with either the 256KB or 384KB design. The tradeoff between performance, area, and leakage must be carefully considered when deciding how much storage should be allocated per SM. Compared with the partitioned or limited flexibility designs, the unified architecture gives designers more freedom as each memory structure must be provisioned for the maximum requirements of any workload. By dynamically partitioning storage, the unified design allows the amount of storage to be set based on the aggregate storage requirements of workloads.

6.5. Tuning Applications for Unified Architecture

Many applications are tuned to fit into the storage requirements of either the register file, shared memory, or cache. A partitioned design

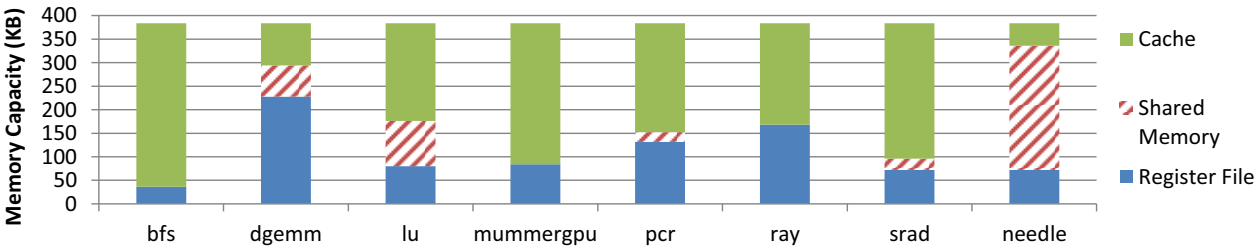


Figure 8: Partitioning of 384KB unified memory.

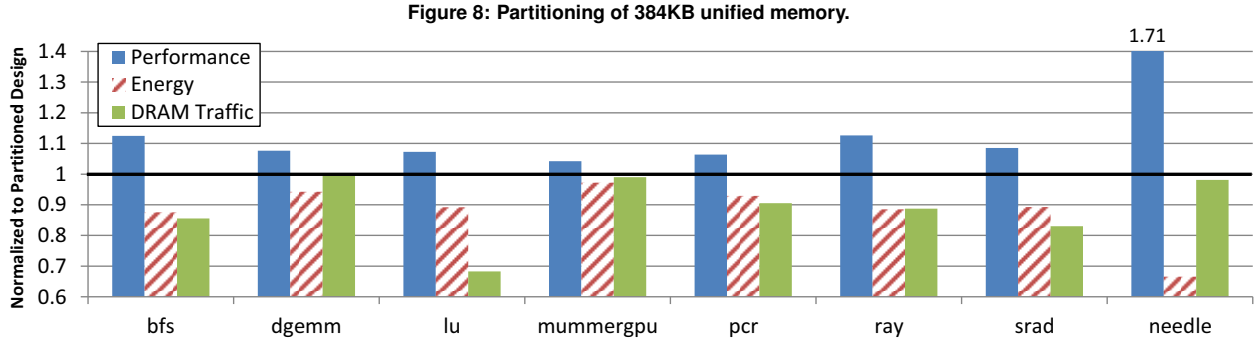


Figure 9: Performance (higher is better), energy (lower is better), and DRAM traffic (lower is better) of unified design (384KB) normalized to an equal-capacity partitioned design for applications that benefit from unified storage.

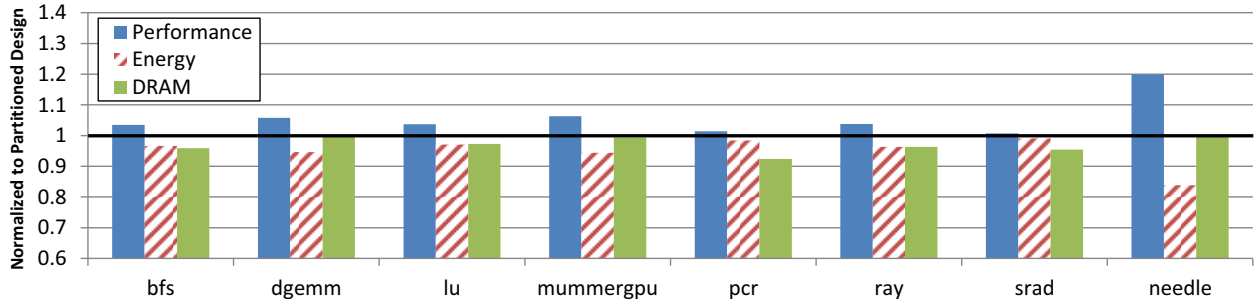


Figure 10: Performance (higher is better), energy (lower is better), and DRAM traffic (lower is better) of Fermi-like limited flexibility design (384KB) normalized to an equal-capacity partitioned design.

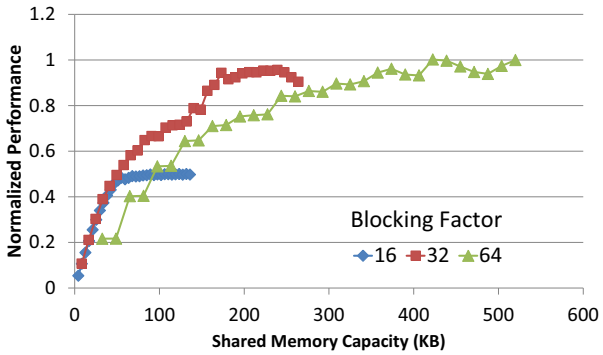


Figure 11: Performance of various blocking factors and shared memory requirements for needle.

forces applications to be tuned across the narrow capacity range of each structure. The unified architecture presents an opportunity to tune applications across the entire range of performance and unified memory capacity points. As a case study, Figure 11 shows performance as a function of shared memory capacity for three different shared memory blocking factors on `Needle`. Performance is normalized to the maximum shared memory capacity tested of 520KB which is required with a blocking factor of 64 and 1024 threads per SM. As the blocking factor is increased, the amount of shared memory required per thread increases. Each point along the lines represents increasing the number of concurrent threads from 32 to 1024 in increments of 32. When the amount of shared memory available is small, as found on prior generation GPUs, the blocking factor of 16 was used. The results discussed so far in this paper have used a blocking factor of 32, as this is the most efficient operating point when 64KB of shared memory is available. When more than 300KB of shared memory is available, a blocking factor of 64 provides slightly better performance and requires fewer concurrent threads than a blocking factor of 32. The unified design allows programmers the option of optimizing their applications over wider ranges of performance points and potentially utilizing more efficient algorithms.

7. Related Work

Several projects have considered reconfigurable memories that serve as either cache or scratchpad for designs other than GPUs, including Smart Memories [12], TRIPS [19], and the TI TMS320C62xx DSP [21]. Ranganathan et al. proposed a reconfigurable cache that could be divided into several partitions with each partition performing a different task [18]. Their work mainly focused on the cache design required for reconfigurability and provided a case study for using a cache partition for instruction reuse in a media processing

application. The ALP project proposed to reconfigure part of the L1 data cache to serve as a vector register file when performing vector processing [20]. Cook et al. proposed mechanisms for flexible partitioning between cache and shared memory in a multi-core CPU [4]. Albonesi proposed selective cache ways which allows a subset of the ways in a set associative cache to be disabled to save power [1].

Volkov identified applications that achieve better performance by using fewer threads as this allows more registers to be allocated to each thread [23]. Recent work uses cyclic reduction as a case study on the tradeoffs between allocating values to the register file versus shared memory along with balancing the number of registers per thread and the number of threads per SM [5]. Murthy et al. developed a model for optimal loop unrolling for GPGPU programs that considers the increase in register pressure versus the potential improvements from unrolling [14]. Our flexible storage system can relax the programming burden associated with the fixed capacity storage structures and accommodate diverse workloads.

8. Conclusion

Modern applications have varying requirements in register file, cache, and shared memory capacity. Traditional GPUs require the programmer to carefully tune their applications to account for the size of each of these structures. In this work, we propose a unified on-chip storage for the register file, cache, and shared memory. This flexible structure can adjust the storage partitioning on a per application basis, providing a performance improvement as high as 71% along with an energy reduction up to 33%. The overhead of the flexibility is small, with a minimal increase in bank conflicts and a small increase in bank access energy. These overheads are negligible in terms of system performance and energy, even for benchmarks that do not benefit from the unified design. We explore the sensitivity to unified memory capacity and find that many benchmarks achieve energy savings with smaller capacity unified memory. Future systems could exploit this fact by disabling unneeded memory. Our unified equal capacity design provides meaningful energy efficiency improvements for a significant number of today's benchmarks, which are tuned for partitioned designs. By making the processor's storage more flexible, we broaden the scope of applications that GPUs can efficiently execute. Future applications or application tuning can further improve efficiency by taking advantage of this new flexibility.

Acknowledgments

We thank the anonymous reviewers, Trevor Mudge, and the members of the NVIDIA Architecture Research Group for their comments. This research was funded in part by DARPA contract HR0011-10-9-0008 and NSF grant CCF-0936700.

References

[1] D. H. Albonesi, "Selective Cache Ways: On-Demand Cache Resource Allocation," in *International Symposium on Microarchitecture*, November 1999, pp. 248–259.

[2] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt, "Analyzing CUDA Workloads Using a Detailed GPU Simulator," in *International Symposium on Performance Analysis of Systems and Software*, April 2009, pp. 163–174.

[3] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. H. Lee, and K. Skadron, "Rodinia: A Benchmark Suite for Heterogeneous Computing," in *International Symposium on Workload Characterization*, October 2009, pp. 44–54.

[4] H. Cook, K. Asanovic, and D. A. Patterson, "Virtual Local Stores: Enabling Software-Managed Memory Hierarchies in Mainstream Computing Environments," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2009-131, September 2009.

[5] A. Davidson and J. D. Owens, "Register Packing for Cyclic Reduction: A Case Study," in *Workshop on General Purpose Processing on Graphics Processing Units*, March 2011, pp. 1–6.

[6] G. Diamos, A. Kerr, S. Yalamanchili, and N. Clark, "Ocelot: A Dynamic Compiler for Bulk-Synchronous Applications in Heterogeneous Systems," in *International Conference on Parallel Architectures and Compilation Techniques*, September 2010, pp. 353 – 364.

[7] K. Fatahalian and M. Houston, "A Closer Look at GPUs," *Communications of the ACM*, vol. 51, no. 10, pp. 50–57, October 2008.

[8] M. Gebhart, D. R. Johnson, D. Tarjan, S. W. Keckler, W. J. Dally, E. Lindholm, and K. Skadron, "Energy-efficient Mechanisms for Managing Thread Context in Throughput Processors," in *International Symposium on Computer Architecture*, June 2011, pp. 235–246.

[9] M. Gebhart, S. W. Keckler, and W. J. Dally, "A Compile-Time Managed Multi-Level Register File Hierarchy," in *International Symposium on Microarchitecture*, December 2011, pp. 465–476.

[10] X. Guo, E. Ipek, and T. Soyata, "Resistive Computation: Avoiding the Power Wall with Low-Leakage STT-MRAM Based Computing," in *International Symposium on Computer Architecture*, June 2010, pp. 371–382.

[11] "MAGMA: Matrix Algebra for GPU and Multicore Architectures," http://icl.eecs.utk.edu/magma.

[12] K. Mai, T. Paaske, N. Jayasena, R. Ho, W. J. Dally, and M. Horowitz, "Smart Memories: A Modular Reconfigurable Architecture," in *International Symposium on Computer Architecture*, June 2000, pp. 161–171.

[13] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, "CACTI 6.0: A Tool to Model Large Caches," HP Laboratories, Tech. Rep., April 2009.

[14] G. S. Murthy, M. Ravishankar, M. M. Baskaran, and P. Sadayappan, "Optimal Loop Unrolling for GPGPU Programs," in *International Symposium on Parallel and Distributed Processing*, April 2010, pp. 1–11.

[15] NVIDIA, "Compute Unified Device Architecture Programming Guide Version 2.0," http://developer.download.nvidia.com/compute/cuda/2_0/docs/NVIDIA_CUDA_Programming_Guide_2.0.pdf, June 2008.

[16] NVIDIA, "NVIDIA's Next Generation CUDA Compute Architecture: Fermi," http://nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf, 2009.

[17] "Parboil Benchmark Suite," http://impact.crhc.illinois.edu/parboil.php.

[18] P. Ranganathan, S. Adve, and N. P. Jouppi, "Reconfigurable Caches and their Application to Media Processing," in *International Symposium on Computer Architecture*, June 2000, pp. 214–224.

[19] K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, D. Burger, S. W. Keckler, and C. R. Moore, "Exploiting ILP, TLP, and DLP with the Polymorphous TRIPS Architecture," in *International Symposium on Computer Architecture*, June 2003, pp. 422–433.

[20] R. Sasanka, M.-L. Li, S. V. Adve, Y.-K. Chen, and E. Debes, "ALP: Efficient Support for All Levels of Parallelism for Complex Media Applications," *ACM Transactions on Architecture and Code Optimization*, vol. 4, no. 1, March 2007.

[21] Texas Instruments, "TMS320C6202/C6211 Peripherals Addendum to the TMS320C6201/C6701 Peripherals Reference Guide (SPRU290)," Tech. Rep., August 1998.

[22] T. Vogelsang, "Understanding the Energy Consumption of Dynamic Random Access Memories," in *International Symposium on Microarchitecture*, December 2010, pp. 363–374.

[23] V. Volkov, "Better Performance at Lower Occupancy," in *GPU Technology Conference*, September 2010.

[24] R. C. Whaley and J. J. Dongarra, "Automatically Tuned Linear Algebra Software," in *International Conference for High Performance Computing*, 1998, pp. 1–27.

[25] H. Wong, M.-M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos, "Demystifying GPU Microarchitecture through Microbenchmarking," in *International Symposium on Performance Analysis of Systems and Software*, March 2010, pp. 235–246.

[26] Y. Zhang, J. Cohen, and J. D. Owens, "Fast Tridiagonal Solvers on the GPU," in *Symposium on Principles and Practice of Parallel Programming*, January 2010, pp. 127–136.

[27] X. Zhuang and S. Pande, "Resolving Register Bank Conflicts for a Network Processor," in *International Conference on Parallel Architectures and Compilation Techniques*, September 2003, pp. 269–278.

2012 IEEE/ACM 45th Annual International Symposium on Microarchitecture

Kernel Weaver: Automatically Fusing Database Primitives for Efficient GPU Computation

Haicheng Wu
Georgia Institute of Technology
hwu36@gatech.edu

Gregory Diamos
NVIDIA Research
gdiamos@nvidia.com

Srihari Cadambi
NEC Laboratories America
cadambi@nec-labs.com

Sudhakar Yalamanchili
Georgia Institute of Technology
sudha@ece.gatech.edu

Abstract

Data warehousing applications represent an emerging application arena that requires the processing of relational queries and computations over massive amounts of data. Modern general purpose GPUs are high bandwidth architectures that potentially offer substantial improvements in throughput for these applications. However, there are significant challenges that arise due to the overheads of data movement through the memory hierarchy and between the GPU and host CPU. This paper proposes data movement optimizations to address these challenges.

Inspired in part by loop fusion optimizations in the scientific computing community, we propose kernel fusion as a basis for data movement optimizations. Kernel fusion fuses the code bodies of two GPU kernels to i) reduce data footprint to cut down data movement throughout GPU and CPU memory hierarchy, and ii) enlarge compiler optimization scope. We classify producer consumer dependences between compute kernels into three types, i) fine-grained thread-to-thread dependences, ii) medium-grained thread block dependences, and iii) coarse-grained kernel dependences. Based on this classification, we propose a compiler framework, Kernel Weaver, that can automatically fuse relational algebra operators thereby eliminating redundant data movement.

The experiments on NVIDIA Fermi platforms demonstrate that kernel fusion achieves 2.89x speedup in GPU computation and a 2.35x speedup in PCIe transfer time on average across the micro-benchmarks tested. We present key insights, lessons learned, measurements from our compiler implementation, and opportunities for further improvements.

1. Introduction

The arrival of big data [20] has energized the search of architectural and systems solutions to sift through massive volumes of data. The use of programmable GPUs has appeared as a potential vehicle for high throughput implementations of data warehousing applications with an order of magnitude or more performance improvement over traditional CPU-based implementations [36, 18]. This expectation is motivated by the fact that GPUs have demonstrated significant performance improvements for data intensive scientific applications such as molecular dynamics [2], physical simulations in science [32], options pricing in finance [34], and ray tracing in graphics [33]. It is also reflected in the emergence of accelerated cloud infrastructures for the Enterprise such as Amazon's EC-2 with GPU instances [40].

However, the application of GPUs to the acceleration of data warehousing applications that perform relational queries and computations over massive amounts of data is a relatively recent trend [14] and there are fundamental differences between such applications and compute-intensive high performance computing (HPC) applications. Relational algebra (RA) queries form substantial components of data warehousing applications and on the surface appear to exhibit significant data parallelism. Unfortunately, this parallelism is generally more unstructured and irregular than other domain specific operations, such as those common to dense linear algebra, complicating the design of efficient parallel implementations. RA operators also exhibit low operator density (operations per byte) making them very sensitive to and limited by the memory hierarchy and costs of data movement.

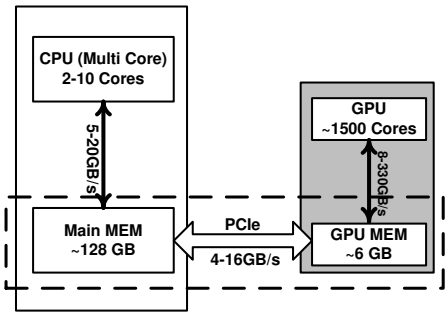


Figure 1: Memory hierarchy bottlenecks for GPU accelerators.

Overall, the nature and structure of RA queries make different demands on i) traversals through the memory hierarchy, ii) choice of logical and arithmetic operators, iii) control flow structure, and iv) data layouts. Consequently, there arise two fundamental issues. First there is a need for the efficient GPU implementations of RA primitives. Second, an issue that is fundamental to the current architecture of GPU-based systems is the set of limitations imposed by the CPU-GPU memory hierarchy, as shown in Figure 1. Internal to the GPU there exists a memory hierarchy that extends from GPU core registers, through on-chip shared memory, to off-chip global memory. However, the amount of memory directly attached to the GPUs (the off-chip global memory) is limited, forcing transfers from the next level which is the host memory that is accessed in most systems via PCIe channels. The peak bandwidth across PCIe can be up to an order of magnitude or more lower than GPU local memory bandwidth. Data warehousing applications must stage and move data throughout this hierarchy. He et al. observed that although GPU can bring 2-27x speedup compared with CPU if only considering computation time, 15-90% of the total execution time is spent on moving data between CPU and GPU when accelerating database applications [18]. Consequently there is a need for techniques to optimize the implementations of data warehousing applications considering both the GPU computation capabilities and system memory hierarchy limitations.

This paper addresses the challenge of optimizing data movement through the CPU-GPU memory hierarchy in the context of data warehousing applications (and hence their dominant primitives). Specifically, we propose and demonstrate the utility of **Kernel Weaver** as a framework for optimizing data movement. Kernel Weaver applies a cross-kernel optimization, *kernel fusion*, to GPU kernels. *Kernel fusion* is analogous to traditional loop fusion and its principal benefits are that it i) reduces transfers of intermediate data through the CPU-GPU memory hierarchy, ii) reduces the overall memory data footprint of a sequence of kernels in each level of the memory hierarchy, and iii) increases the textual scope, and hence benefits, of many existing compiler optimizations.

1072-4451/12 $26.00 © 2012 IEEE
DOI 10.1109/MICRO.2012.19

IEEE computer society

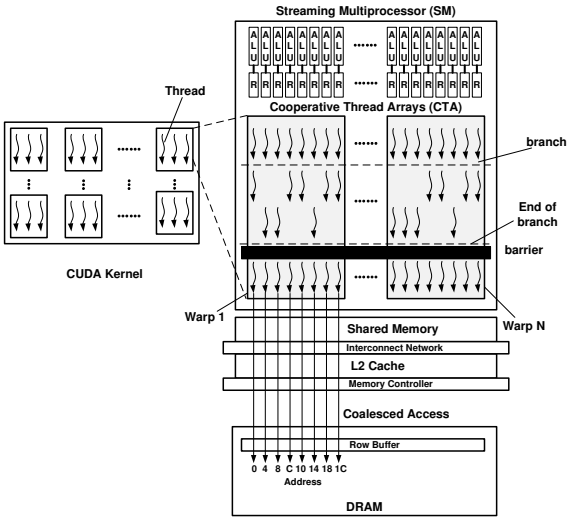


Figure 2: NVIDIA C2050 architecture and execution model.

This paper proposes the Kernel Weaver optimization framework and demonstrates the impact of kernel fusion for optimizing data movement in patterns of interacting operators found in the TPC-H benchmark suite. The goal of this paper is to provide insight into how, why, and when kernel fusion works with quantitative measurements from implementations targeted to NVDIA GPUs. This paper makes the following specific contributions:

- Introduction of the Kernel Weaver framework and algorithms for automated kernel fusion;
- Definition of basic dependences and general criteria for kernel fusion applicable across multiple application domains;
- Quantification of the impact of kernel fusion on different levels of the CPU-GPU memory hierarchy for a range of RA operators;
- Proposes and demonstrates the utility of compile-time data movement optimizations based on kernel fusion.

2. Background and Motivation

2.1. Programmable GPU

This paper uses NVIDIA devices and CUDA as the target platform. Figure 2 shows an abstraction of NVIDIA's GPU architecture and execution model. A CUDA program is composed of a series of multi-threaded kernels. Kernels are composed of a grid of parallel work-units called *Cooperative Thread Arrays* (CTAs) [37], that are mapped to Single Instruction Multiple Thread (SIMT) units called stream multiprocessors (SMs) where each thread has support for independent control flow. Different CTAs can execute in arbitrary order and synchronization between threads only exists within a CTA. Global memory is used to buffer data between CUDA kernels as well as to communicate between the CPU and GPU. Each SM has a shared scratch-pad memory with allocations for each CTA and can be used as a software controlled cache. Registers are privately owned by each thread to store immediately used values. CTAs execute in SIMD chunks called warps; hardware warp and thread scheduling hide memory and pipeline latencies. Effective utilization of the memory subsystem is also critical to achieving good performance.

CUDA and OpenCL are the dominant programming models in GPU computation. CUDA is dedicated to NVIDIA devices, and OpenCL is supported by NVIDIA, AMD and Intel GPUs. Terms used to describe GPU abstractions such as data parallel threads and shared scratch-pad memory typically vary depending on the specific programming model being considered. CUDA typically uses the terms *thread* and *shared memory*, and OpenCL typically uses *work item* and *local memory*. The CUDA terminology is adopted in this paper because Kernel Weaver is currently implemented based on it. However, the same concept and technology can also be applied to OpenCL and its supported devices.

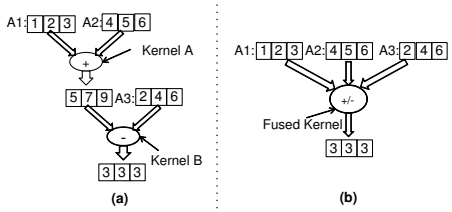


Figure 3: Example of kernel fusion.

2.2. Relational Algebra Operators

Relational algebra (RA) operators can express the high level semantics of an application in terms of a series of bulk operations on relations [1]. They are the building blocks of modern relational database systems. A relation is a set of tuples, each of which comprises of a list of n attributes. Some attributes are keys that are considered by the RA operator.

Table 1 lists the common RA operators and a few simple examples. In general, these operators perform simple tasks on a large amount of data. A typical data warehousing query consists of dozens of RA operators over massive data sets.

In addition to these operators, data warehousing applications perform arithmetic computations ranging from simple operators such as aggregation to more complex functions such as statistical operators used for example in forecasting or retail analytics. Further, operators such as SORT and UNIQUE are required to maintain certain order amongst data elements and thereby can introduce certain ordering constraints amongst relations.

2.3. Motivation

The idea of GPU kernel fusion comes from classic loop fusion optimization. Basically, kernel fusion reduces data flow between two kernels (via the memory system) by merging them into one larger kernel. Therefore, its benefits goes far beyond reduction in PCIe traffic. Figure 3 depicts an example of kernel fusion. Figure 3(a) shows two dependent kernels - one for addition and one for subtraction. After fusion, a single functionally equivalent new kernel (Figure 3(b)) is created. The new kernel directly reads in three inputs and produces the same result without generating any intermediate data.

Kernel Fusion has six benefits as listed below. The first four stem from creating a smaller data footprint through fusion since it is unnecessary to store temporary intermediate data in global memory after each kernel execution, while the other two relate to increasing the compiler's optimization scope.

Smaller Data Footprint results in the following benefits:

- Reduction in Memory Accesses: Fusing data dependent (producer-consumer) kernels enables storage of intermediate data in registers or GPU shared memory (or cache) instead of global memory.
- Temporal Data Locality: As in traditional loop fusion, access to common data structures across kernels expose and increase temporal data locality. For example, fusion can reduce array traversal overhead when the array is accessed in both kernels.

RA Operator	Description	Example
SET UNION	A binary operator that consumed two relations to produce a new relation consisting of tuples with keys that are present in at least one of the input relations.	x = {(2,b),(3,a),(4,a)}, y = {(0,a),(2,b)} UNION x y → {(0,a),(2,b),(3,a),(4,a)}
SET INTERSECTION	A binary operator that consumes two relations to produce a new relation consisting of tuples with keys that are present in both of the input relations.	x = {(2,b),(3,a),(4,a)}, y = {(0,a),(2,b)} INTERSECT x y → {(2,b)}
SET DIFFERENCE	A binary operator that consumes two relations to produce a new relation of tuples with keys that exist in one input relation and do not exist in the other input relation.	x = {(2,b),(3,a),(4,a)}, y = {(3,a),(4,a)} DIFFERENCE x y → {(2,b)}
CROSS PRODUCT	A binary operator that combines the attribute spaces of two relations to produce a new relation with tuples forming the set of all possible ordered sequences of attribute values from the input relations	x = {(3,a),(4,a)}, y = {(True)} PRODUCT x y → {(3,a,True),(4,a,True)}
JOIN	A binary operator that intersects on the key attribute and cross product of value attributes.	x = {(2,b),(3,a),(4,a)}, y = {(2,f),(3,c),(3,d)} JOIN x y → {(2,b,f),(3,a,c),(3,a,d)}
PROJECT	A unary operator that consumes one input relation to produce a new output relation. The output relation is formed from tuples of the input relation after removing a specific set of attributes.	x = {(2,False,b),(3,True,a),(4,True,a)} PROJECT [0,2] x → {(2,b),(3,a),(4,a)}
SELECT	A unary operator that consumes one input relation to produce a new output relation that consists of the set of tuples that satisfy a predicate equation. This equation is specified as a series of comparison operations on tuple attributes.	x = {(2,False,b),(3,True,a),(4,True,a)} SELECT (key==2) x → {(2,False,b)}

Table 1: The set of relational algebra operations. In the example, the 1st attribute is the "key". (Syntax: (x,y) – tuple of attributes; {(x1,y1),(x2,y2)} – relation; [0,2] – attribute index)

- Reduction in PCIe Traffic: Kernel fusion can cut down transfers of inter-kernel data across the PCIe interconnect.
- Larger Input Data: Since kernel fusion reduces intermediate data thereby freeing GPU memory, larger data sets can be processed on the GPU increasing throughput.

Larger Optimization Scope brings two benefits:

- Common Computation Elimination: When two kernels are fused, the common stages of computations are redundant and can be avoided.
- Improved Compiler Optimization Benefits: When two kernels are fused, the textual scope of many compiler optimizations are increased bringing greater benefits than when applied to each kernel individually.

These benefits are especially useful for data warehousing applications since RA operators are fine grained and exhibit low operation density, ops per byte transferred from memory. Fusion naturally improves operator density and hence performance. Figure 4 is a simple example comparing the GPU computation throughput of back-to-back SELECTs with and without kernel fusion. Inputs are randomly generated 32-bit integers, the x-axis is the problem size which fits GPU memory, and kernels were manually fused in this example. On average, fusing two SELECTs achieves **1.80x** larger throughput while fusing three kernels achieves **2.35x**. Fusing three SELECTs is better since more redundant data movement is avoided and larger code bodies are created for optimization.

Recently, Intel introduced the Sandy (and Ivy) Bridge architectures and AMD brought Fusion APUs to the market. Both designs put the CPU and GPU on the same die and remove the PCIe bus. In these systems, four out of the six benefits listed above still apply (excluding Reduction in PCIe Traffic and Larger Input Data). Thus, kernel fusion is still valuable.

While a programmer could perform a fusion transformation manually, database queries are typically supplied in a high level language like Datalog or SQL, from which lower-level operations are synthesized using a query planner and compiler. Automating this process

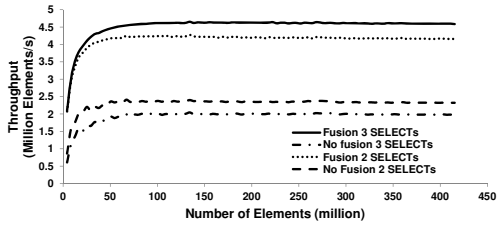


Figure 4: Performance Comparison between fused and independent SELECTs.

as a compilation transformation is necessary to make GPUs accessible and useful to the broader community of business analysts and database experts. Moreover, running kernel fusion dynamically in a just-in-time compiler (JIT) creates opportunities to leverage runtime information for more effective optimizations.

3. System Overview

Kernel Weaver is implemented as part of a domain-specific compilation and run-time system illustrated in Figure 5. The language front-end is based on the Datalog language [21]. Datalog is a declarative language used to express database queries and operations - in this case over large data sets. The output of the language front-end is a query plan that contains nodes corresponding to arithmetic, logical, and relational operators and their dependences. These are translated into an internal kernel intermediate representation which drives the Kernel Weaver transformation engine. Kernel Weaver operates on CUDA source implementations of RA operators stored in a primitive library to produce fused CUDA implementations from which *nvcc* is used to generate kernel code in NVIDIA's parallel thread execution (PTX) instruction set. The lightweight host runtime layer [10] picks up the fused PTX kernels and drives the Ocelot dynamic compilation and runtime infrastructure [11] which is responsible for the execution on the NVIDIA GPUs. Note that each RA operator may be implemented as several CUDA kernels so that fusing operators requires coordinated fusion of several CUDA kernels. While we tested all of

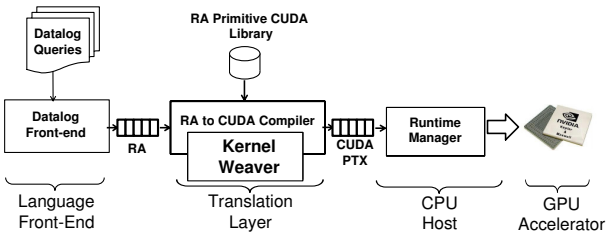


Figure 5: System diagram of Kernel Weaver

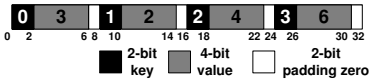


Figure 6: An example of a relation with four tuples, each compressed into 8-bits and packed into a single 32-bit word.

the examples in this paper, the language front-end needs further work before the full Datalog language can be executed on GPUs.

3.1. Kernel Representation

Kernel fusion is based on the multi-stage formulation of algorithms for the RA operators. Multi-stage algorithms are common to sorting [31], pattern matching [39], algebraic multi-grid solvers [5], or compression [17]. This formulation is popular for GPU algorithms in particular since it enables one to separate the structured components of the algorithm from the irregular or unstructured components. This can lead to good scaling and performance. Kernel fusion can now be explained as a process of weaving (mixing, fusing, and reorganizing) stages from different operators to generate new optimized operator implementations.

The high level description of the order and functionality of the stages will be referred as an *algorithm skeleton*. In this paper we use the algorithm skeletons developed by Diamos et al. [12] which have been evaluated to be 1.69-3.54x faster than those developed by He et al. [18]. Diamos et al. store relations as a densely packed array of tuples with strict weak-ordering. Figure 6 is an example of a 32-bit relation containing 4 tuples sorted according to the key attributes. The sorted form allows for efficient array partitioning and tuple lookup operations. In our compilation environment, skeletons for all of the RA operators are stored in the RA primitives library (see Figure 5) with CUDA implementations of each stage. The remainder of this section describes the basic structure, relevant details, and adaptations we have made of their implementation.

All RA operator skeletons are comprised of three major stages, partition, compute and gather. In the following we briefly describe the functionality of each stage using the implementation of a simple operator - SELECT (Figure 7) - as an example.

Partition: The input relations are partitioned into independent sections that are processed in parallel by different CTAs. For unary operators such as SELECT in Figure 7 the input relations can be evenly partitioned to balance the workload across CTAs. Binary operators such as JOIN and SET INTERSECTION are more complex in this stage since they need to partition both inputs and partitioning is based on a key value consequently producing unbalanced sizes of inputs to CTAs and resulting in unbalanced compute loads.

Compute: A function for each RA operator is applied to its partition of the inputs to generate independent results. Different RA operators are specialized to effectively utilize fine-grained data parallelism and the multi-level memory hierarchy of the GPU to maximize

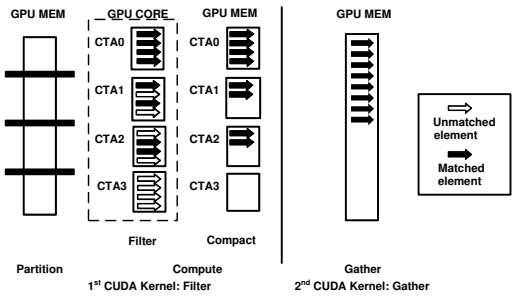


Figure 7: Example algorithm of SELECT

performance. For example, the SELECT in Figure 7 first *filters* every element in parallel and then leverages the shared memory to *compact* [6] the filtered result in preparation to create a contiguous output.

Gather: The results computed in individual partitions are gathered into a global dense sorted array of tuples by using a coalesced memory to memory copy, a common CUDA programming pattern [30].

Multi-stage RA operators are implemented as multiple CUDA kernels - typically one per stage. Kernel weaver fuses operators by interleaving stages and then fusing interleaved stages (their respective CUDA implementations) to produce a multi-stage implementation of the fused operator. Thus, fusion of CUDA kernels is necessary to realize operator fusion. A variety of alternative implementations can be used for the implementation of each stage and can be accommodated by the operator fusion process. Diamos et. al. [12], report performance results that are significantly better than any reported results in the literature. Consequently we use their multi-stage algorithms for RA operators in the demonstration of kernel weaver.

4. Automating Fusion

This section introduces the process of kernel fusion employed in Kernel Weaver. For simplicity, the initial description is based on each operator being implemented as a single data parallel kernel. Subsequently, we will describe higher performance multi-stage implementations of the RA operators. This section describes three main steps to fuse operators: (i) using compiler analysis to find all groups of operators that can be fused, (ii) selecting candidates to fuse, and (iii) performing fusion and generating code for the fused operators.

4.1. Criteria for Kernel Fusion

The simple idea is to take two kernels say with 4096 threads each, and produce a single kernel with 4096 threads, where each thread is the result of fusing two corresponding threads in the individual kernels. Clearly, the data flow between the two fused threads must be correctly preserved. The classification below can be understood from the perspective of preserving this simple model of kernel fusion. The first consideration is finding feasible combinations of data parallel kernels to fuse via compiler analysis, followed by the selection of the best options. Two types of criteria for fusion of candidate kernels are that they possess i) same kernel configuration (CTA dimensions and thread dimensions), and ii) producer-consumer dependence.

The first criteria is similar to loop fusion [23] that requires compatible loop headers (same iteration number, may need loop peeling to pre-transform the loop, etc.). Kernel fusion also requires compatibility between kernel parameters. The fused kernel will have the same kernel configuration as the candidates. The data parallel nature of RA operators make their implementation independent (with respect to correctness) of the kernel configuration. Thus, while too many

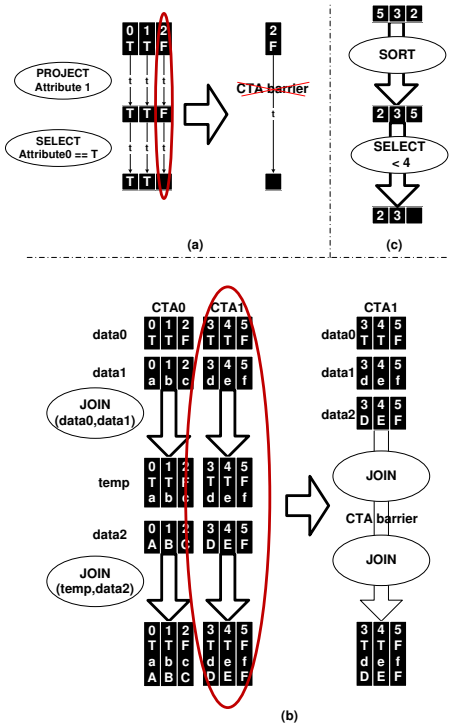


Figure 8: Example of three kinds of dependence: (a) thread dependence; (b) CTA dependence; (c) kernel dependence.

or too few CTAs or threads may lead to inefficient use of resources, fusion can be performed correctly if the kernel configurations are the same. This work tests a set of micro-benchmarks (see Section 5) with a wide range of combination of CTA dimensions and thread dimensions and picks one pair that works best in most cases.

The second criteria is due to the fact that the benefits listed in Section 2.3 are derived primarily from exploiting producer-consumer dependences. Data dependence analysis is necessary to find candidate kernels. Producer-consumer dependence between two data parallel kernels can be classified into three categories as shown in Figure 8: thread, CTA and kernel dependence.

In the first category, each thread of the consumer kernel only consumes data generated by a single thread from the producer kernel. Figure 8(a) illustrates such an example with tuples containing two attributes, e.g., (1,T). Dependences between producer and consumer kernels corresponding to unary RA operators such as SELECT and PROJECT, belong to this category because the operation on one input tuple is independent of the operation performed on any neighboring tuple. In this case, corresponding producer and consumer threads from each kernel can be fused without having to insert synchronization operations. This type of producer-consumer dependence between kernels is referred to as *thread dependence*.

The second category is wherein every CTA of the consumer kernel depends on the completion of a CTA of the producer kernel. Such dependences are referred to as *CTA dependences*. For example, this occurs between binary RA operators such as JOIN and SET INTERSECT that have a producer-consumer dependence. Consider, Figure 8(b) that illustrates a producer-consumer dependence between two JOIN operators. The first operator performs a JOIN operation across tuples from two input data sets, *data0*, and *data1*. Each CTA is provided a partition of input tuples, corresponding to some range of the key value used in the JOIN (in this example each tuple has a unique key value which is an integer). Thus, a thread in a CTA must compare the key values of tuples it is processing with the key values of tuples being processed by every other thread in the CTA, *and only within the CTA*. While such a partitioning of input tuples across CTAs produces unbalanced loads between CTAs, data dependences between threads are confined to remain within a CTA. The producer CTA writes its tuples to shared memory where a CTA from the consumer kernel can now pick it up. A barrier synchronization is necessary after the first JOIN operation before the second JOIN operation can start. The actual implementation is more involved, but for the purposes of this paper, corresponding threads from producer-consumer CTAs can be fused with appropriately placed barriers.

```
Input: a list of operators op
Output: a list of fusion candidate groups c
i = 0;
length = size of list op;
Topologically sort op;
while i ≠ length do
    class = classify dependence between op[i] and
            its predecessors and successors;
    if class == Kernel Dependence then
        delete op[i];
    end
    i = i + 1;
end
c = all connected subgraphs of op;
```

Algorithm 1: Searching for fusing candidates.

The third category is wherein the consumer kernel has to wait until the completion of all threads in the kernel, i.e., kernel fusion is not feasible. A typical example is where the producer kernel is a SORT operator (Figure 8(c)) because it behaves like a global barrier. The reasons are i) it cannot be launched until all inputs arrive; ii) SORT shuffles all data and the following consumer operators need to wait for its completion before being able to start streaming data. Such dependence is referred to as *kernel dependence*.

Note the three categories of dependences are from the perspective of being able to fuse kernels by fusing corresponding threads within producer-consumer kernels. Accordingly, the dependences are implicitly associated with the level of the memory hierarchy used to pass data. Fused threads across thread dependent kernels use the register file which is allocated by the thread. Fused threads across CTA dependent kernels use shared memory which is allocated by the CTA. Finally, according to the above classification, the kernels in an dependence graph that are candidates for kernel fusion only exhibit thread or CTA dependences with other kernels, and are bounded by operators with kernel dependences. Algorithm 1 formalizes the steps to find kernel fusion candidates. Its main idea is first removing operators causing kernel dependence from the graph and then finding the rest connected operators.

The output of the language front-end consists of RA operators and their associated variables. This information is used to construct an RA dependence graph like the one shown in Figure 9(b). The nodes in the graph represent RA operators and the directional edges identify nodes with the producer-consumer dependences. The large circle bounded by SORT operators contains operators satisfying the dependence requirement and are candidates for fusion. Instances supporting recursive queries (e.g. ancestor(a,c)←parent(a,b),ancestor(b,c)) may generate a dependence graph with enclosed loops. This work only

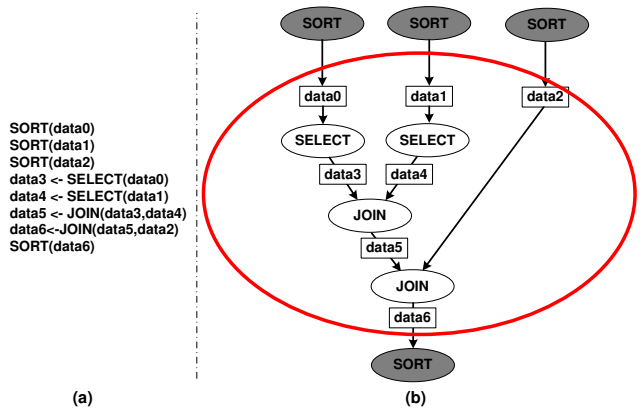


Figure 9: Example of constructing dependence graph: (a) database program; (b) dependence graph.

Input: a list of candidate operators op
Input: resource budget b
Output: a list of fusion groups f
$i = 0$;
$j = 0$;
$length$ = size of list op;
Topologically sort op;
while $i \neq length$ **do**
  add $op[i]$ to $f[j]$;
  $cost$ = resource usage estimation of $f[j]$;
  if $cost > b$ **then**
    delete $op[i]$ from $f[j]$;
    $j = j+1$;
  else
    $i = i + 1$;
  end
end

Algorithm 2: Choosing operators to fuse.

considers acyclic graphs although often loop unrolling and related known optimizations can create acyclic dependence graphs.

4.2. Choosing Operators to Fuse

Fusing all the kernels meeting above criteria may not be practical. The main constraint on fusion is resource constraints - pressure on limited registers and limited amount of shared memory available within each stream multiprocessor. Fusion choices must also be ordered based on dependences and performance impact. Accordingly we adopt the following heuristic and use Algorithm 2 to choose operators to fuse.

Figure 10 is an example that shows how the greedy heuristic of Algorithm 2 works. It starts from the candidates circled in Figure 9(b). Figure 10(a) first performs a topological sorting on the dependence graph to produce a list of operators. If operators execute in this order, all dependences will be honored. Starting from the top of the list, Figure 10(b) searches for the longest contiguous sequence of operators that can be fused, within resource constraints, i.e., fits within the shared memory and registers budgeted for each CTA (*data3* and *data4* become internal to the fused operator). In the example in Figure 10(c) the second JOIN cannot be added since the estimated shared memory resource usage is larger than the budget. The algorithm repeats the above process for the next not fused operator, the second JOIN, until no more operators can be fused. Resource

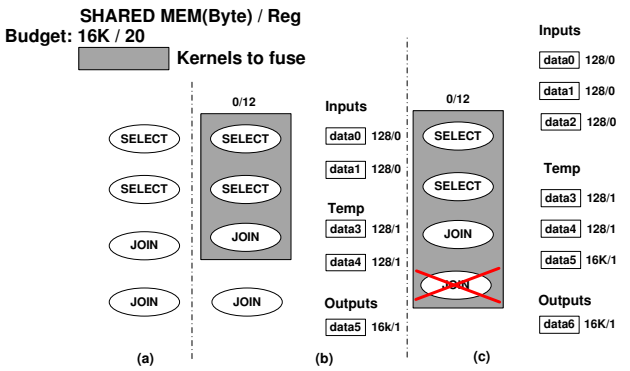


Figure 10: Example of choosing operators to fuse: (a) topologically sorted operators; (b) choose the first three operators to fuse; (c) refuse to fuse the fourth operator.

usage estimation is discussed in Section 4.3.3 after introducing code generation.

The intuition underlying the above method is that it is more important to fuse operators executed earlier than those executed later. The reason is that data warehousing applications normally process large amounts of data. After several filtering and reduction operators, the data set is reduced significantly. Resource permitting, fusing the first few operators in the dependency graph provides the most benefit.

4.3. Kernel Weaving and Fusion

Given the dependence graph and candidate operators to fuse, the final step is performing the fusion. Recall that each operator is implemented as a multi-stage algorithm with three stages - *partition*, *compute*, and *gather* - each of which is implemented as a CUDA data parallel kernel. The fused operator still has these three stages. At a high level, fusion is achieved by two main steps: (i) grouping the *partition*, *compute*, and *gather* stages of the operators together (which we also refer to as interleaving); (ii) fusing the individual stages. In other words, the *partition* stages of the candidate operators are fused together into a single data parallel kernel, which could be viewed as the *partition* stage for the newly fused operator. Similarly, the *compute* and *gather* stages are fused into a single fused *compute* and *gather* stage respectively. For example, when two operators are fused, the fused operator will have the multi-stage structure shown in Figure 11 where the two compute stages are fused into one data parallel kernel (the fused partition and fused gather stages similarly represent fusion of individual partition and gather stages). The fused computation stage performs the computation stage of the original operators in the order of their dependences. All intermediate data and data sizes are stored in the shared memory or registers. The fused operator may have multiple inputs and outputs.

The above fusion process includes code generation for the fused operators. Code generation takes as input a description of a topologically sorted set of operators to be fused and their associated variables, and produces CUDA code for the data parallel kernels that implement the fused operator. The CUDA code is generated by concatenating the instantiated algorithm skeleton code of each stage, and connecting the outputs of one stage to the inputs of the next stage. How to connect stages is discussed in the following sub sections. A variable table, which records and tracks the use of variables between stages, is needed to instantiate the skeleton. Figure 11 shows how the variable table tracks the variables that hold result data and result size of each computation stage.

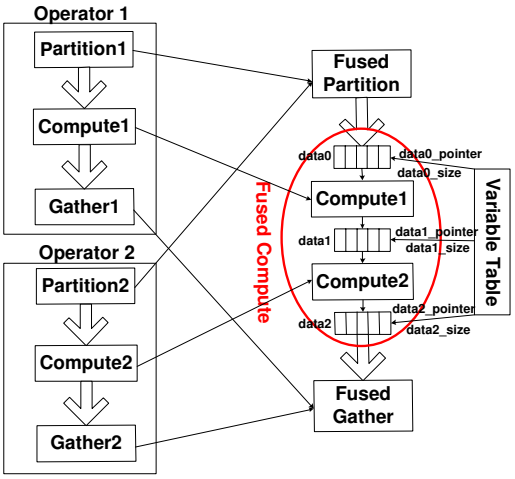


Figure 11: The structure of generated code (fusing two operators).

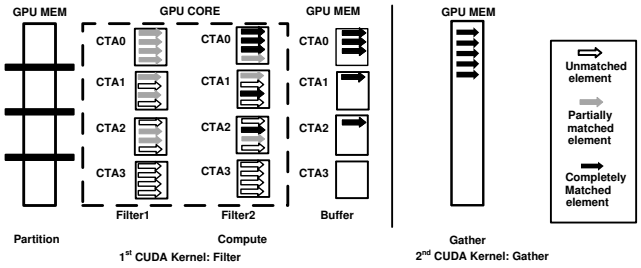


Figure 12: Example of fusing two SELECTs.

Fusing operators depends on whether thread dependence or CTA dependence exists between operators. We now describe in more detail how to fuse operators with thread and CTA dependences.

4.3.1. Fusing Thread Dependent Only Operators Unary operators SELECT and PROJECT exhibit thread dependence. The kernel configuration (threads/CTA and CTA grid dimensions) of both operators are equal. Therefore each thread in the producer operator is fused with a corresponding thread in the consumer operator.

The partition stage of the fused operator remains the same as that of the producer operator. The compute stage of the fused operator is a data parallel kernel with the same kernel configuration, where each thread is created as follows. Every thread first loads a tuple from its input partition into registers. The computation of corresponding producer and consumer threads are performed using these registers, i.e., SELECT or PROJECT in the correct order. These operators either discard data (SELECT) or discard attributes (PROJECT). The output of this sequence of operations is compacted into an output array. The gather stage accumulates all of the data from different threads in the fused compute stage into contiguous memory.

As shown in Figure 7, the computation stage of SELECT has two parts, filter and stream compaction. After kernel fusion, stream compaction is needed only when the SELECT result should be copied to GPU memory. Figure 12 is an example of fusing two back-to-back SELECTs together. Compared with Figure 7, only one filter operation is added. Moreover, the fused kernel only needs to read and write memory once rather than twice.

For PROJECT, its result tuple should be stored into a new register with a different data type since it contains less attributes. Thus, the operations after PROJECT have to use this new register instead.

4.3.2. Fusing CTA and Thread Dependent Operators Binary relational operators are CTA dependent . This change increases the

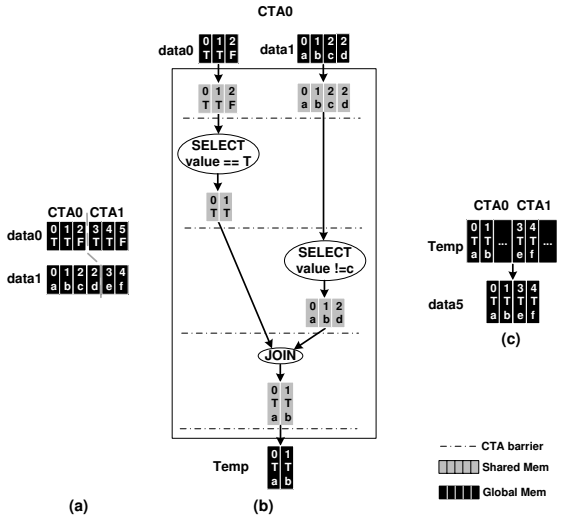


Figure 13: Example of Generated Code of Figure 10(b): (a) Partition two inputs; (b) Computation of one CTA; (c) Gather one output.

number of inputs of the fused operators and necessitates the following main distinction in code generation: (i) Use binary search to partition inputs; (ii) Use shared memory to support CTA dependence; (iii) Synchronize two operators having CTA dependence. Thus, code generation has to be extended to support the three differences. Figure 13 shows the generated code for the operators in Figure 9(b) and is used as example to explain the extensions.

Our approach is to maintain the independent operation of each CTA to be able to fuse corresponding CTAs from the producer and consumer operators. This is achieved in the *partition* stage by partitioning the input set by key values. Each CTA then receives a set of tuples corresponding to a specific range of key value pairs. This is achieved using binary search [3] and both inputs of each binary operator are partitioned across CTAs. For example, in Figure 13(a), *data0* is first evenly partitioned into two parts bounded by pivot tuples. Then, a binary search is used to lookup the tuples in *data1* corresponding to the *key* attributes of *data0* pivot tuples. The partitioned data sizes of the two inputs provided to each CTA thus may differ (e.g. *data1*). However, when fusing two binary operators (e.g., two JOIN operators), three inputs need to be partitioned and each operator may use different keys. For instance, one JOIN may use the first 2 attributes as a *key* and the other JOIN may only use the first attribute as *key*. In this case, the fused input stage will only use the first attribute as *key*. This preserves the independence of operation across CTAs.

Figure 13(b) is an example of the computation stage of one CTA. The other CTA works in exactly the same way but upon different data. In the beginning, each CTA first allocates a software controlled cache in shared memory for each input and then loads data into the cache (e.g. CTA0 loads in a portion of corresponding *data0* and *data1* divided as in Figure 13(a)) . Afterwards, a CTA-wise fused computation performs fused operations upon those cached data. Within a CTA, the generated code can perform all supported operations such as SELECT and JOIN. If two connected operators have CTA dependence (e.g. between SELECT and JOIN), the result data of the producer operator should be stored in a cache allocated in the shared memory, and the result size is stored in a register. To guarantee all threads within a CTA finish updating the cache, a CTA

barrier synchronization is needed after the producer operation. If two dependent operators only exhibit thread dependence, they only need to use register(s) to pass value(s) and no synchronization is necessary. For example, the first operator in Figure 13 to execute is SELECT and it has CTA dependence relationship with its consumer JOIN. Thus, SELECT has to store its result in shared memory rather than the register. The second SELECT is handled in the same way. Thus, the inputs of JOIN all reside in the shared memory before its execution. After JOIN, the result is dumped to GPU memory.

The gather stage (Figure 13(c)) is the same as in the thread dependent only cases which packs the useful results generated by two CTAs into an output array.

4.3.3. Resource Usage Code generation decides how many resource will be occupied. As shown in Figure 10(c), some resources are used to store input, output, and intermediate temporary data. Others are used inside the computation.

Fusing thread dependent operators stores intermediate data in the registers. The number of needed registers depends on the data type of the tuple which is provided by the database front-end. Fusing CTA dependent operators stores temporary data in the shared memory and temporary data size value in one register. Allocated shared memory size is a function of data type, input data size and operator type. For example, SET INTERSECT needs to allocate $\min(input1, input2)$ tuples for its output. The data variable and data size variable stored in registers are live until they are no longer needed.

Registers are also needed to perform partition, computation, and gather. The partition result, the beginning and the end position of all inputs, uses variables to pass to the computation stage. The liveness of the variables used inside each stage is the same as the scope of the stage. Thus, different variables of different stages can reuse the same registers. So, the register usage of a fused operator is not larger than the maximum of the register usage in each stage plus the registers used to pass values between stages. The registers used by each stage can be determined as long as the data types of all tuples are known.

4.4. Extensions

The preceding three sections discussed how code is generated for RA operators having producer-consumer dependence. This method can be extended to support other dependence or other operators.

The first extension is to support input dependence, i.e. operators shares the same inputs. The benefits of fusing these operators is that the input data shared by different operators only need to be loaded once, which is not as important as the case of producer-consumer dependence. Fusing operators having input dependence also increases the resource pressure. The modification to the above automation process is to detect input dependence when constructing the dependence graph. The code generation part can remain the same.

The second extension is to support simple arithmetic operations such as addition, subtraction, multiplication and division. These arithmetic operators are much simpler than RA operators. They have two inputs, but use even partitions to divide both inputs. The dependence between them belongs to thread dependence and can use registers to store computation results.

5. Experimental Evaluation

Table 2 shows our experimental infrastructure. We use TPC-H [9], a widely-used decision support benchmark suite, to quantify the speedups of kernel fusion in a practical situation. TPC-H comprises 22 queries with varying degrees of complexity. The queries analyze relations between customers, orders, suppliers and products using complex data types and multiple operators on large volumes of randomly generated data sets. Before showing results for actual TPC-H queries, we examine micro-benchmarks derived from the TPC-H queries.

CPU	2 quad-core Xeon E5520 @ 2.27GHz
Memory	48 GB
GPU	1 Tesla C2070 (6GB GDDR5 memory)
OS	Ubuntu 10.04 Server
GCC	4.4.3
NVCC	4.0

Table 2: Experimental Environment.

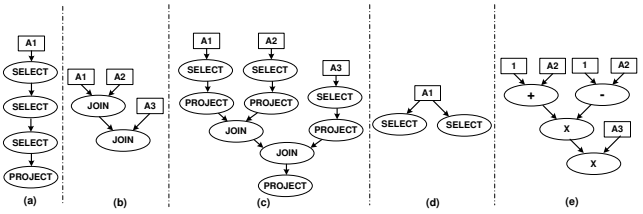


Figure 14: Common operator combinations to fuse.

5.1. Micro-benchmarks

We analyze TPC-H queries and identify commonly occurring combinations of operators that are potential candidates for fusion. From the 22 queries in TPC-H, Figure 14 illustrates some frequently occurring patterns of operators corresponding to different cases discussed in Section 4. In the figure, (a) is a sequence of back-to-back SELECT operators that perform filtering, for instance, of a date range. It only has thread dependence. (b) is a sequence of JOIN operations that creates a large table with multiple attributes, and exhibits CTA dependence. (c) corresponds to the JOIN of three small selected tables and has both thread and CTA dependence. (d) represents the case when different SELECT operators need to filter the same input data and has input dependence. (e) performs arithmetic computations such as $price \times (1 - discount) \times (1 + tax)$ which appears in several TPC-H queries. The PROJECTs in the figure discard their sources and only retain part of the result. The above patterns can be further combined to form larger patterns that can be fused. For example, (a) and (b) can be combined to form (c).

In the following experiments, the tuple used in patterns (a)–(d) are 16 bytes. (e) uses single precision floating point values.

5.1.1. Examples of Generated Code Figure 15 shows the generated fused computation stage code of Figure 14(a) (only two SELECTs shown for brevity). It performs two SELECTs and a PROJECT. The first two filters operate on the value in *data_reg*, and store the filter result in *match*, which is later used to determine if follow-up operations are needed. The result of PROJECT is written to a new register *project_reg* since its data type is smaller than *data_reg*. The last step, stream compaction, dumps the value stored in this new register to the GPU's global memory. The generated code may be less compact than manually written code, but compilers such as *nvcc* can optimize it to produce high quality binary code.

5.1.2. Small Inputs The micro-benchmarks listed in Figure 14 are first tested with small inputs that fit in GPU memory. The purpose of this is to isolate the benefits of kernel fusion from the effects of PCIe transfer. Figure 16 shows the speedup in the pure GPU execution time (no PCIe transfer) with kernel fusion. The input data are randomly

```
if(begin_input + id < end_input)
{
  data_reg = begin_input[id];                    Load Data To Reg
  {
    unsigned char key = extract(data_reg);
    if(comp(key, 64))
    match = true;
  }                                              Filter0
  {
    if(match)
    {
      unsigned char key =extract(data_reg);
      if(comp(key, 64))
      match = true;
    }
  }                                              Filter1
  {
    if(match)
    {
      project_reg = project(data_reg, 0);
    }
  }                                              PROJECT
}
{
  unsigned int max = 0;
  unsigned int output_id = exclusiveScan(match, max, 0);
  if(match)
    buffer0[output_id] = project_reg;            Stream
  __syncthreads();                               Compaction
  if(threadIdx.x < max)
    begin_output0[outputIndex0 + threadIdx.x] = buffer0[threadIdx.x];
  outputIndex0 += max;
}
```

Figure 15: Example of generated computation stage source code of Figure 14(a).

generated and then fed into the automatically generated fused code using the compilation flow of Figure 5. The baseline implementation for comparison directly uses the implementation from the primitive library without fusing. Similar to Figure 4, the performance data are averaged over a wide set of problem size (from 64 MB to 1 GB). On average, kernel fusion achieves a **2.89x** speedup. Cases (a) and (e) containing only thread dependence show the largest speedup, because they do not insert new synchronizations, and threads execute independently. Furthermore, (a) gets rid of three stream compaction stages and three gather stages after fusion. The speedup of case (d) is less than the rest because it has input dependences and can only benefit from loading fewer inputs. (b) and (c) have CTA dependences and need extra synchronizations which makes kernel fusion less beneficial than the thread dependence only cases. The speedup in case (c) is larger than (b) because (c) fuses some thread dependence operators. Considering the reported CPU and GPU computation performance difference [18, 12], the baseline GPU implementation should be 4x–40x faster than CPU and kernel fusion can further increase the GPU advantage.

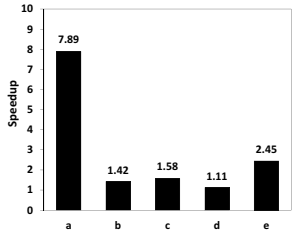


Figure 16: Speedup in execution time (Small Inputs).

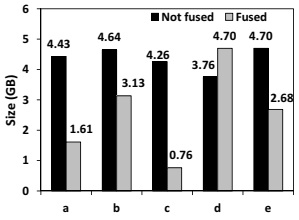


Figure 17: GPU global memory allocation reduction.

The next set of experiments examine the benefits claimed in Section 2.3, specifically the improvement in GPU global memory usage, total memory access cycles and compiler efficacy.

Figure 17 shows the GPU global memory allocated and used with and without kernel fusion. The additional memory without fusion is attributed to large intermediate results. In pattern (d) however, the fused operator uses a little more memory because the fused compute stage has to store two outputs in memory for future gather rather than one. Similarly, Figure 18 shows the data for GPU memory access cycles (collected using the *clock()* intrinsic). On average, fusion reduces the GPU global memory access time by 59%. Finally, Figure 19 quantifies the impact of the compiler. All micro-benchmarks are compiled with *-O3* and *-O0* flags, both with and without kernel fusion. The figure shows the speedups achieved by *-O3* compared to *-O0*. Clearly, kernel fusion enables the the compiler to perform better optimization.

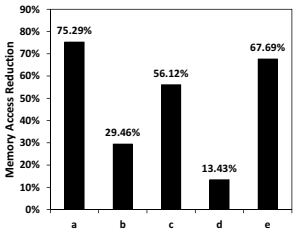


Figure 18: Reduced memory cycles with kernel fusion.

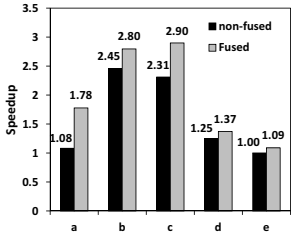


Figure 19: Comparison of compiler optimization impact.

When fusing two or three SELECTs together (e.g., pattern (a)), the second or third SELECT might have some idle threads because some data are not matched in the earlier SELECT. This might impact the overall performance. Figure 20 examines the performance sensitivity of kernel fusion to the selection ratio (percentage of data matching selection condition) with randomly generated 32-bit integers. The results shows fusing two 10% SELECTs produces (more idle threads) 1.28x speedup while fusing two 90% SELECTs (less idle threads) produces 2.01x speedup. Thus, it is fair to say that idle threads may impact the performance but do not negate the benefits of data movement reduction.

5.1.3. Large Inputs In this experiment, the program inputs are enlarged so that every operator has to move its result data back to host to make room for the next operator when kernel fusion is not used. But the problem size still fits the GPU memory when running fused kernels. This set of tests examines the effect of kernel fusion on reducing PCIe data traffic. The input data is generated on the CPU and the final results are sent back to the CPU. Figure 21 compares the execution time with and without kernel fusion over a wide range of problem sizes. In this figure, the execution time comprises two parts: GPU computation time and PCIe transfer time. On average, kernel fusion achieves **2.91x** speedup in the GPU computation time, **2.08x** speedup in PCIe data transfer, and **1.98x** speedup overall. Computation time speedup is similar to the small input case because performance scales with data size. The speedup of PCIe transfer dominates the overall speedup because it is the bottleneck for RA operators. The case (d)

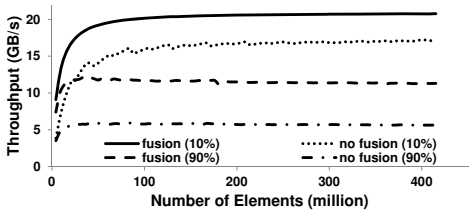


Figure 20: Sensitivity to selection ratio.

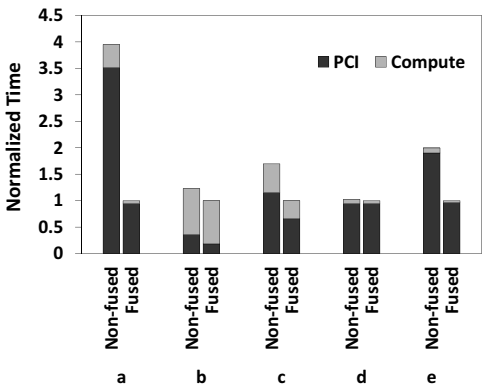


Figure 21: GPU execution time with and without kernel fusion (Large Inputs).

with input independence does not enjoy any benefit from reducing PCIe transfer because the fused version has the same data footprint as the baseline. Considering only the four producer-consumer cases, kernel fusion has **2.35x** speedup in PCIe and a **2.22x** speedup overall. If compared with CPU only systems, due to the large amount of time spent in PCIe shown in Figure 21, the computation performance gap between CPU and GPU would be reduced or GPU could even lag behind of CPU for these simple micro-benchmarks even with the help of kernel fusion. Other techniques discussed in the Related Work section are needed for a complete GPU-assisted database system.

5.1.4. Resource Usage Table 3 lists the GPU resource usage and occupancy (active warps / maximum active warps) of the individual operators and the fused patterns. Since resources are finite, utilizing too many resources per thread may decrease the occupancy. We obtain resource information from *ptxas* and occupancy from *CUDA_Occupancy_calculator*. The left four columns list the resources used by individual operators (e.g. 1 PROJECT needs 11 PTX registers and 0 byte shared memory), and the right four columns show the usage of each pattern after kernel fusion (e.g. fused pattern (a) uses 22 PTX registers and around 2.3K shared memory). The statistics indicate that kernel fusion in most cases increases the resource usage which is the same as the impact of loop fusion, and consequently may lower occupancy (pattern (b) – pattern(e)). Taking pattern (b) as an example, it requires 55 PTX registers and about 23K shared memory with fusion. However, if running two JOINs back-to-back sequentially, each JOIN only needs 47 PTX registers and 13K shared memory. Pattern (a) will use less shared memory after kernel fusion than a single SELECT because i) thread dependence does not use shared memory to store temporary results and ii) the data type of fused results array buffered in shared memory uses smaller data type since PROJECT removes some attributes.

5.2. Real Queries

In this section, we evaluate kernel fusion with two real queries from TPC-H benchmark suites, Q1 and Q21. Q1 represents arithmetic centric queries and Q21 represents relational centric queries. TPC-H queries are very complex (e.g. the 15 operators of Q1 maps to 107 kernels to execute). While the microbenchmarks were compiled and executed with the Datalog front-end, the query plans for the two TPC-H queries presented here were created manually. The additional language support required in the front-end to also automatically compile all Datalog TPC-H queries is being completed for open source distribution of the compiler.

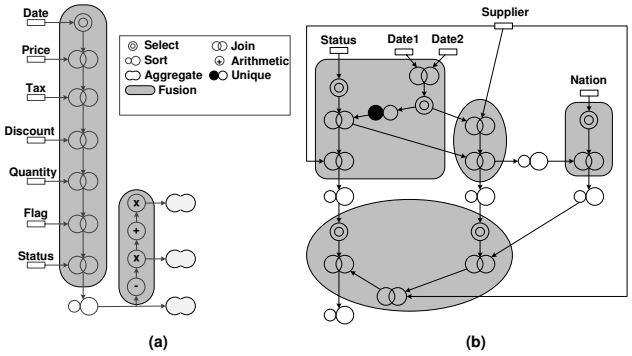


Figure 22: (a) Query plan for Q1; (b) Query plan for Q21.

Q1 calculates several price statistics for selected entries. Figure 22(a) is the query plan generated for Q1. There are (i) several JOINs and one SELECT to generate a large table from seven columns, ii) SORT by a different key, and iii) arithmetic calculations over several fields of the table. The first part of the query including one SELECT and six JOINs can be fused into one operator. All of the arithmetic computations performed as the final part of the query can be fused as well. The SORT operator causes kernel dependence and cannot be fused because it has to wait for the completion of the JOINs and arithmetic operations have to wait for the completion of the SORT.

From the TPC-H database, query Q21 identifies suppliers who were not able to ship required parts in a timely manner. Compared with Q1, Q21 has less arithmetic computation but many more relational operations. Figure 22(b) is its simplified query plan (simple operators such as PROJECTs are omitted for clarity). Just as with Q1, SORTs form a boundary for the application of kernel fusion since they can not be fused with their producers nor their consumers.

We tested two queries with 200 MB to 1 GB data and averaged the performance. For Q1, the most time consuming part is the SORT operator which takes around 71% of the total execution time, but cannot be optimized. However, fusion dramatically speeds up the other operators and contributes an overall **1.25x** speedup. Further study shows that when SORT is excluded, the remaining operators can be fused and fusion achieves a **3.18x** speedup due to the fusing of 6 JOINS and 1 SELECTs into a single kernel. For Q21, kernel fusion realizes a **1.22x** overall speedup, which is significant given the complexity of the operators.

The fused patterns for Q1 and Q21 are built based on JOIN operators (e.g. joining several columns together into a larger table and then performing different cross-field computations). These patterns appear very frequently in all 22 queries of TPC-H so that they can all get similar speedup from kernel fusion.

6. Discussion

The proposed framework, Kernel Weaver, opens a door to a class of new optimizations that can be applied in different situations. The following discusses three possible opportunities.

Different Domain: Instead of database applications, kernel fusion can also be used in other domains such as dense linear algebra. The requirement is that the application should use multi-stage algorithms and the stages are independent of each other so that they can be weaved into a new format. The classification of dependences used in this paper is still useful as a guide to fusion candidate selections.

Different Representation: In this paper, the optimization occurs at the CUDA source code level. However, the same technology can

	PTX Reg #	Shared Mem (Byte)	Occupancy (%)		PTX Reg #	Shared Mem (Byte)	Occupancy (%)
PROJECT	11	0	100	(a)	22	2308	88
SELECT	22	3848	88	(b)	55	23560	33
JOIN	47	13580	38	(c)	62	23048	17
+/-	10	0	100	(d)	30	4612	67
Multiply	13	0	100	(e)	27	0	75

Table 3: Resource usage and occupancy of individual (left) and fused (right) operators.

be applied to different representations, such as OpenCL [24], CUDA PTX or LLVM [25], as long as each stage of the operator can be represented. Thus, kernel fusion can be implemented as a module of a static compiler or a JIT compiler that optimizes the representations online.

Different Platform: Furthermore, kernel fusion can be considered as a general cross-kernel optimization that is not only restricted to GPU devices. The benefits of smaller data footprints and larger optimization scope still applies if the CPU program is optimized using kernel fusion. Thus, if using an execution model translator such as Ocelot [11], and a runtime manager such as Harmony [10] it is possible to execute fused kernels on both the CPU and GPU to fully utilize the available computation power.

Moreover, a more complicated fusion framework can use invariant analysis to reschedule operators and to fuse those which are not originally executed back-to-back. For example, if switching the order of SORT and SELECT of Figure 9(c) does not alter the final result, the switch brings more opportunity to optimize since SELECT can thus fuse with the operators before SORT.

7. Related Work

For decades, academia and industry have invested a great deal of effort in query optimization for traditional CPU-based relational database management systems (RDMS) [22, 29]. These query optimizations originated from different perspectives, considered different factors, and made different tradeoffs. Take the CPU cache as an example - the database system can choose among techniques such as cache prefetching, cache partitioning, cache compression, and so on to minimize cache misses and miss penalties [19]. This paper mainly uses shared memory to apply kernel fusion. Compared with the CPU cache, GPU shared memory is i) completely programmable which provides more flexibility, ii) accessed by a large number of threads which forces us to keep concurrency in mind. Thus, the optimization here differs quite a bit.

The idea of kernel fusion arises from loop fusion [23], a well studied loop optimization technique, which can reduce loop traversal overhead and improve certain types of data locality. It is also used in loop parallelization since it can aggregate a large loop body.

The most similar to our work is that of Sato et al. [35], who built a system to run general primitives, map, reduce and zipwith on GPUs with kernel fusion enabled. They fused the CPU code of the primitives and then inserted CUDA runtime library calls and other CUDA required language features to turn a C program into a CUDA program. Thus, they did not exploit the advantageous characteristics of GPUs, such as the multi-level memory hierarchy, that can improve performance. There are also some domain specific kernel fusion techniques targeting GPUs. Copperhead, developed by Catanzaro et al. [7], attempted to fuse a subset of Python primitives to reduce global synchronizations when accelerating them using GPUs. They classified dependence into *local* and *global* which are similar to the thread and kernel dependence of this paper, and fuse primitives having *local* dependence. Thus, they can only fuse a few simple primitives. Chakravarty et al. [8] noticed the benefits of kernel fusion in accelerating Haskell array operations with GPUs and listed it as their future work. On the CPU side, Lee et al. [27] propose a runtime framework, Thread Tailor, which uses fusion techniques albeit at a different level of granularity. Their framework partitions an application into a large number of threads and use a greedy heuristic to combine these small threads later based on their dependences.

There are also several ongoing projects using GPUs to accelerate database applications. In particular, He et al. [18] implement a complete GPU database system, GDB, which is also based on the GPU implementation of relational algebra operators. Further, other groups focus on designing algorithms to accelerate individual RA primitives [36, 26, 38, 28, 15, 16]. Similarly, Bakkum et al. [4] modified the virtual machine infrastructure of SQLite to use GPUs to execute SQLite opcodes (not RA primitives). All of these previous works achieve several factors of speedup in comparison with their CPU counterparts. However, none of them use any optimizations to further improve the overall performance of the database system on GPUs. Moreover, He et al. also point out that the PCIe transfer time may outweigh the speedup enabled by the GPUs and suggest the use of data compression techniques to reduce the amount of transfered data [13]. Our work differs in that we are seeking to discover and develop mainstream compiler passes that can automatically provide inter-kernel optimizations.

To further boost the performance of a GPU assisted database system, other techniques including but not limited to PCIe data compression [13], double buffer [41], and GPU aware query optimizer, are also important to reduce the PCIe hazard. These techniques are orthogonal to kernel fusion because they are independent of the contents transfered over PCIe and can be applied together with kernel fusion. As to larger systems having multiple GPUs or even spanning over multiple nodes, the runtime should have an intelligent scheduling module that can balance the work load of each device (CPU and GPU) and minimize the data movement over the interconnections [10].

8. Conclusion

This paper proposes a cross-kernel optimization framework, Kernel Weaver, that can apply kernel fusion optimization to improve the performance of relational algebra primitives used in data warehousing applications on GPUs. Kernel fusion aggregates larger body of code that can reuse as much data as possible. It can reduce the data traffic through the memory hierarchy caused by the I/O bound nature of database applications, and also enlarge the optimization scope.

To automate the process of kernel fusion, this paper first classifies the producer-consumer dependence between RA operators into three categories: thread, CTA and kernel dependence. Then, Kernel Weaver leverages the multi-stage algorithm design to weave stages from operators having thread and CTA dependence. The experiments shows that kernel fusion optimization brings **2.89x** speedup in GPU computation, **2.35x** speedup in PCI transfer on average across the micro-benchmarks tested. The same technique can be applied to different domain, different representation format and different devices.

Acknowledgements

This research was supported in part by the National Science Foundation under grants IIP-1032032 & CCF 0905459, by LogicBlox Corporation, and by equipment grants from NVIDIA Corporation. We also acknowledge the detailed and constructive comments of the reviewers.

References

[1] S. Abiteboul, R. Hull, and V. Vianu, *Foundations of databases*. Addison-Wesley, 1995, vol. 8.

[2] J. Anderson, C. Lorenz, and A. Travesset, "General purpose molecular dynamics simulations fully implemented on graphics processing units," *Journal of Computational Physics*, vol. 227, no. 10, pp. 5342–5359, 2008.

[3] R. Baeza-Yates, "A fast set intersection algorithm for sorted sequences," *Lecture Notes in Computer Science*, vol. 3109, pp. 400–408, 2004. Available: http://www.springerlink.com/content/yth9h90y94n10l7e

[4] P. Bakkum and K. Skadron, "Accelerating SQL database operations on a GPU with CUDA," in *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*. ACM, 2010, pp. 94–103.

[5] N. Bell, S. Dalton, and L. Olson, "Exposing fine-grained parallelism in algebraic multigrid methods," NVIDIA Corporation, NVIDIA Technical Report NVR-2011-002, Jun. 2011.

[6] M. Billeter, O. Olsson, and U. Assarsson, "Efficient stream compaction on wide simd many-core architectures," in *Proceedings of the Conference on High Performance Graphics 2009*, ser. HPG '09. New York, NY, USA: ACM, 2009, pp. 159–166. Available: http://doi.acm.org/10.1145/1572769.1572795

[7] B. Catanzaro, M. Garland, and K. Keutzer, "Copperhead: compiling an embedded data parallel language," in *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, ser. PPoPP '11. New York, NY, USA: ACM, 2011, pp. 47–56. Available: http://doi.acm.org/10.1145/1941553.1941562

[8] M. Chakravarty *et al.*, "Accelerating haskell array codes with multicore gpus," in *Proceedings of the Sixth Workshop on Declarative Aspects of Multicore Programming*. ACM, 2011, pp. 3–14.

[9] T. Council, "Tpc benchmark h, standard specification revision 1.3. 0," 1999.

[10] G. Diamos and S. Yalamanchili, "Harmony: an execution model and runtime for heterogeneous many core systems," in *Proceedings of the 17th international symposium on High performance distributed computing*. ACM, 2008, pp. 197–200.

[11] G. Diamos *et al.*, "Ocelot: A dynamic compiler for bulk-synchronous applications in heterogeneous systems," in *Proceedings of PACT '10*. ACM, 2010, pp. 353–364.

[12] G. Diamos *et al.*, "Efficient relational algebra algorithms and data structures for gpu," CERCS, Georgia Institute of Technology, Tech. Rep. GIT-CERCS-12-01, Feb. 2012.

[13] W. Fang, B. He, and Q. Luo, "Database compression on graphics processors," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 670–680, 2010.

[14] W. Fang *et al.*, "Frequent itemset mining on graphics processors," in *Proceedings of the Fifth International Workshop on Data Management on New Hardware*. ACM, 2009, pp. 34–42.

[15] N. Govindaraju *et al.*, "Gputerasort: high performance graphics coprocessor sorting for large database management," in *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*. ACM, 2006, pp. 325–336.

[16] N. Govindaraju *et al.*, "Fast computation of database operations using graphics processors," in *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*. ACM, 2004, pp. 215–226.

[17] I. Grebnov, "libbsc: A high performance data compression library," http://libbsc.com/default.aspx, November 2011.

[18] B. He *et al.*, "Relational query coprocessing on graphics processors," *ACM Transactions on Database Systems (TODS)*, vol. 34, no. 4, p. 21, 2009.

[19] B. He and Q. Luo, "Cache-oblivious databases: Limitations and opportunities," *ACM Transactions on Database Systems (TODS)*, vol. 33, no. 2, p. 8, 2008.

[20] T. Hetherington *et al.*, "Characterizing and evaluating a key-value store application on heterogeneous cpu-gpu systems," in *Proceedings of the 2012 IEEE International Symposium on Performance Analysis of Systems and Software*, April 2012.

[21] S. Huang, T. Green, and B. Loo, "Datalog and emerging applications: an interactive tutorial," in *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, 2011, pp. 1213–1216.

[22] M. Jarke and J. Koch, "Query optimization in database systems," *ACM Computing surveys (CsUR)*, vol. 16, no. 2, pp. 111–152, 1984.

[23] K. Kennedy and K. McKinley, "Maximizing loop parallelism and improving data locality via loop fusion and distribution," *Languages and Compilers for Parallel Computing*, pp. 301–320, 1994.

[24] Khronos OpenCL Working Group, *The OpenCL Specification, version 1.0.29*, 8 December 2008.

[25] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation," in *Proc. of the 2004 International Symposium on Code Generation and Optimization*, 2004, pp. 75–86.

[26] T. Lauer *et al.*, "Exploring graphics processing units as parallel coprocessors for online aggregation," in *Proceedings of the ACM 13th international workshop on Data warehousing and OLAP*. ACM, 2010, pp. 77–84.

[27] J. Lee *et al.*, "Thread Tailor : Dynamically Weaving Threads Together for Efficient , Adaptive Parallel Applications," in *Proc. of the 37th Annual International Symposium on Computer Architecture*, 2010.

[28] M. Lieberman, J. Sankaranarayanan, and H. Samet, "A fast similarity join algorithm using graphics processing units," in *Data Engineering, 2008. ICDE 2008. IEEE 24th International Conference on*. IEEE, 2008, pp. 1111–1120.

[29] M. Mannino, P. Chu, and T. Sager, "Statistical profile estimation in database systems," *ACM Computing Surveys (CSUR)*, vol. 20, no. 3, pp. 191–221, 1988.

[30] W. mei W. Hwu and D. Kirk, "Proven algorithmic techniques for many-core processors," http://impact.crhc.illinois.edu/gpucourses/courses/sslecture/lecture2-gather-scatter-2010.pdf, 2011.

[31] D. Merrill and A. Grimshaw, "Revisiting sorting for gpgpu stream architectures," University of Virginia, Department of Computer Science, Charlottesville, VA, USA, Tech. Rep. CS2010-03, 2010.

[32] J. Mosegaard and T. Sørensen, "Real-time deformation of detailed geometry based on mappings to a less detailed physical simulation on the gpu," in *Proceedings of Eurographics Workshop on Virtual Environments*, vol. 11, 2005, pp. 105–111.

[33] S. G. Parker *et al.*, "Optix: a general purpose ray tracing engine," *ACM Transactions on Graphics*, vol. 29, pp. 66:1–66:13, July 2010.

[34] V. Podlozhnyuk, "Black-scholes option pricing," *Part of CUDA SDK documentation*, 2007.

[35] S. Sato and H. Iwasaki, "A skeletal parallel framework with fusion optimizer for gpgpu programming," *Programming Languages and Systems*, pp. 79–94, 2009.

[36] P. Trancoso, D. Othonos, and A. Artemiou, "Data parallel acceleration of decision support queries using cell/be and gpus," in *Proceedings of the 6th ACM conference on Computing frontiers*. ACM, 2009, pp. 117–126.

[37] L. G. Valiant, "A bridging model for parallel computation," *Commun. ACM*, vol. 33, no. 8, pp. 103–111, 1990.

[38] P. Volk, D. Habich, and W. Lehner, "GPU-based speculative query processing for database operations," in *Proceedings of the 1st International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures*, 2010.

[39] P. D. Vouzis and N. V. Sahinidis, "Gpu-blast: using graphics processors to accelerate protein sequence alignment," *Bioinformatics*, vol. 27, no. 2, pp. 182–8, 2010. Available: http://www.ncbi.nlm.nih.gov/pubmed/21088027

[40] E. Walker, "Benchmarking amazon ec2 for high-performance scientific computing," *Usenix Login*, vol. 33, no. 5, pp. 18–23, 2008.

[41] H. Wu *et al.*, "Optimizing data warehousing applications for gpus using kernel fusion/fission," in *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2012 IEEE 26th International*. IEEE, 2012, pp. 2433–2442.

2012 IEEE/ACM 45th Annual International Symposium on Microarchitecture

KnightShift: Scaling the Energy Proportionality Wall Through Server-Level Heterogeneity

Daniel Wong Murali Annavaram
Ming Hsieh Department of Electrical Engineering
University of Southern California
Los Angeles, CA
{wongdani,annavara}@usc.edu

Abstract

Server energy proportionality has been improving over the past several years. Many components in a system, such as CPU, memory and disk, have been achieving good energy proportionality behavior. Using a wide range of server power data from the published SPECpower data we show that the overall system energy proportionality has reached 80%. We present two novel metrics, linear deviation and proportionality gap, that provide insights into accurately quantifying energy proportionality. Using these metrics we show that energy proportionality improvements are not uniform across various server utilization levels. In particular, the energy proportionality of even a highly proportional server suffers significantly at non-zero but low utilizations. We propose to tackle the lack of energy proportionality at low utilization using server-level heterogeneity. We present KnightShift, a server-level heterogenous server architecture that introduces an active low power mode, through the addition of a tightly-coupled compute node called the Knight, enabling two energy-efficient operating regions. We evaluated KnightShift against a variety of real-world datacenter workloads using a combination of prototyping and simulation, showing up to 75% energy savings with tail latency bounded by the latency of the Knight and up to 14% improvement to Performance per TCO dollar spent.

1. Introduction

Energy consumption of datacenter servers are a critical concern. Server operating energy costs comprise a significant fraction of the total operating cost of datacenters. However, many servers operate at low utilization and still consume significant energy due to the lack of ideal energy proportionality [6].

Server consolidation [7, 8] can boost utilization on some servers while allowing idle servers to be turned off, improving energy proportionality at the datacenter level. Unfortunately, server shutdown is not always possible due to data availability concerns and workload migration overheads. When server shutdown is impractical, as is the case in many industrial data centers [3, 26], system-level energy proportionality approaches must be explored.

Energy proportionality improvements of various server components [10, 30], such as CPUs and memory, has fueled the improvements of overall system efficiency. Energy proportionality of current systems has reached around 80%. While 80% seems reasonable, the primary concern today is that energy proportionality improvements have not been uniform across different utilizations. The problem of disproportionality is particularly acute at non-zero but low server utilization. Since no single component dominates server energy usage [31], holistic system-level approaches must be developed to improve energy proportionality particularly at low utilization regions.

Several system-level power saving approaches have focused on reducing the power consumption during idle periods [24]. Researchers then focused on increasing the length of idle periods by queueing requests [26] or by shifting I/O burden directly to disk and memory [2, 3]. However, as multicore servers becomes dominant, idle periods are virtually nonexistent [26, 34]. Even as idle periods become rare, servers still spend a significant fraction of their execution time operating at low utilization levels. Thus there is a critical need to develop active low-power modes that exploits low-utilization periods to continue improving server-level energy proportionality across the entire utilization range.

This paper addresses this critical need by proposing KnightShift, a server-level energy proportionality technique. This work makes the following contributions:

Metrics to Identify Disproportionality(§2): We propose metrics to evaluate energy proportionality and to identify sources of disproportionality. Using data from historical SPECpower [35] results of 291 servers, we show that commonly used metrics such as dynamic range are inappropriate due to the lack of linearity in energy consumption across different utilizations. We present a metric for measuring linearity of energy consumption across different utilizations. Using the linearity metric we show that the proportionality gap is much wider at lower utilization than at idle or higher utilization.

Energy Proportionality Trend Analysis(§3): From historical SPECpower data we show the existence of an energy proportionality wall due to the lack of improvements to the dynamic range and poor energy efficiency at low server utilization periods. Previous work (§4) only targeted improvements to the dynamic range by improving idle power. In order to continue improving energy proportionality, we must improve the linearity of the server's energy proportionality curve, especially at lower utilization where the majority of the proportionality gap exists.

KnightShift(§5): We present KnightShift, a server-level heterogeneous server architecture that introduces an active low power mode to exploit low-utilization periods. By fronting a high-power primary server with a low-power compute node, called the Knight, we enable two energy-efficient operating regions. We show that KnightShift effectively improves energy proportionality and linear deviation of servers in §6. We present evaluation results of KnightShift in §7 and explore TCO impact in §8.

2. Measuring Energy Proportionality

In order to understand energy proportionality trends, we must first quantify energy proportionality. Figure 1 illustrate the power usage of two servers over their operating utilization, called the *energy proportionality curve*. The dotted line shows the *ideal* energy proportionality curve of a server. The dashed line shows the *linear* energy proportionality curve by interpolating idle and peak power. The solid line shows the *actual* server energy proportionality curve. The data presented in this figure are obtained from measurements on real servers reported to SPECpower (more detailed analysis of SPECpower data is provided later).

1072-4451/12 $26.00 © 2012 IEEE
DOI 10.1109/MICRO.2012.20

IEEE computer society

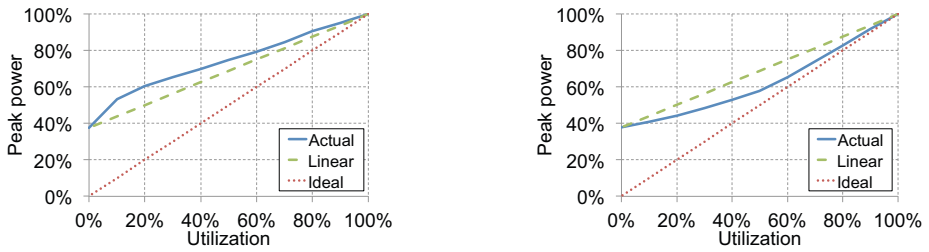


Figure 1: Energy Proportionality (EP) curve. The dotted, dashed, and solid lines shows the ideal, linear, and actual server EP curve, respectively.

Dynamic Range: The *dynamic range* (DR) metric is commonly used as a first order approximation for energy proportionality. The dynamic range of a server is given by:

$$DR = \frac{Power_{peak} - Power_{idle}}{Power_{peak}} \quad (1)$$

where $Power_{peak}$ is the peak power at 100% utilization and $Power_{idle}$ is the idle power at 0% utilization. In Figures 1a and 1b, the DR for both servers are the same at 60% . An ideal energy proportional system would have DR of 100%. DR only accounts for peak and idle power usage and does not account for power usage variations across different utilizations. Since most servers are rarely fully utilized or fully idle, DR is a poor measurement of the server's actual proportionality. For example, assume that both servers in Figure 1 consume 100W at peak power. If each server experiences utilization distribution similar to Google servers reported in [6], then Server A (on the left) would consume on average 28% more power (68.6W vs 52.6W) compared to Server B (on the right), even though they both have the same dynamic range.

Energy Proportionality: To accurately quantify energy proportionality, we must account for intermediate utilization power usage. The *energy proportionality* (EP) of a server (proposed in [29] and adapted for this paper) is given by:

$$EP = 1 - \frac{Area_{actual} - Area_{ideal}}{Area_{ideal}} \quad (2)$$

where $Area_{actual}$ and $Area_{ideal}$ is the area under the server's actual and ideal energy proportionality curve, respectively. Note that if $Area_{actual} = Area_{linear}$, then EP would equal DR. Therefore, DR is a good measurement of energy proportionality only if a server is *linearly energy proportional*, however, this is not the case in most servers. For example, the EP of Server A and B is 53% and 74%, respectively. Although the DR of both servers is 60%, their EP values *differ by over 20%*. Compared to DR, EP provides a more accurate metric in determining how energy efficient a server is. Energy proportionality is a function of the dynamic range and the linearity of the energy proportionality curve. Thus to accurately account for energy proportionality, one has to account for the amount of deviation from linearity within the server's energy proportionality curve.

Linear Deviation: We define *Linear Deviation* (LD) as a measure of the energy proportionality curve's linearity. Linear Deviation is given by:

$$LD = \frac{Area_{actual}}{Area_{linear}} - 1 \quad (3)$$

A server is considered *linearly energy proportional* if LD = 0, *superlinearly energy proportional* if LD $>$ 0, and *sublinearly energy proportional* if LD $<$ 0. Figure 1a and 1b shows a proportionality curve with superlinear and sublinear energy proportional system, respectively. Superlinear energy proportional servers have EP $<$ DR , while sublinear energy proportional servers have EP $>$ DR. This can be proven by equation 2 where $Area_{+LD} > Area_{linear} > Area_{-LD}$, therefore $EP_{+LD} < EP_{linear} < EP_{-LD}$, where $EP_{linear} = DR$.

Proportionality Gap: The *Proportionality Gap* (PG) is a measure of deviation between the server's actual energy proportionality and the ideal energy proportionality at individual utilization levels. PG allows us to quantify the disproportionality of servers at a finer granularity compared to EP to better pinpoint the causes of disproportionality. PG at utilization level x% is given by:

$$PG_{x\%} = \frac{Power_{actual@x\%} - Power_{ideal@x\%}}{Power_{peak}} \quad (4)$$

For an ideal energy proportional server, the PG for all utilization levels is 0. For superlinearly proportional systems, like Server A, PG is very large at zero utilization and it continues to grow for some time before it starts to shrink. For sublinearly proportional systems, like Server B, PG is very large at zero utilization but it continues to decrease with utilization.

3. Energy Proportionality Trends

To understand trends in energy proportionality, we analyze the submitted results of SPECpower [35] for 291 servers from November 2007 to December 2011. These servers are a representative mix of server configurations in current use. They feature servers with various vendors, form factors, and processors. The SPECpower benchmark evaluates the power and performance characteristics of servers by measuring the performance and power consumption of servers at each 10% utilization interval. These trends are shown in Figure 2 and are discussed below.

Dynamic Range: Figure 2a plots the dynamic range of servers along with the median trend line. Each data point corresponds to one server whose SPECpower results were posted on a given date. Overall, DR improved from about 50% to 80% from 2007 to 2009. From 2009 onward, DR stagnated at 80%. Although the best DR is 80%, half of new servers today still have DR less than 70%. Even in 2011, there are still new servers with DR less than 40%. We can surmise that achieving 100% dynamic range is very difficult due to energy disproportional and energy inefficient components such as power supplies, voltage converters, fans, chipsets, network components, memory, and disks.

Energy Proportionality: Figure 2b shows that EP trends are similar, but not identical, to DR trends. Clearly, EP has also stalled at around 80%. This *energy proportionality wall* is mainly due to the lack of DR improvement. Each server's EP data point is classified

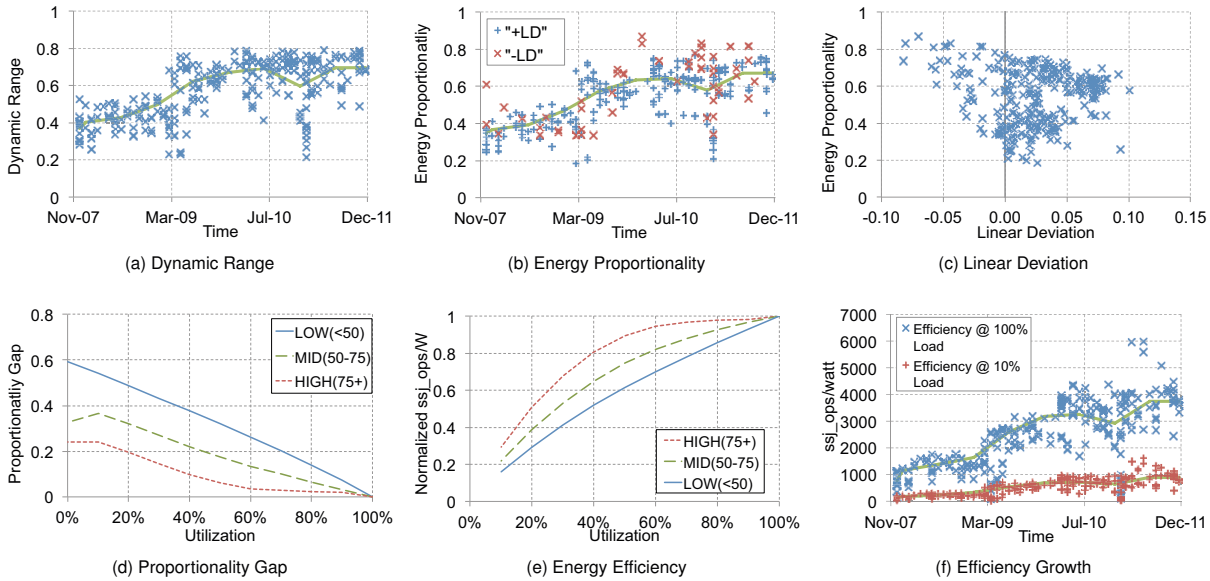


Figure 2: Energy Proportionality Trends

as either superlinear (+LD) or sublinear (-LD) proportional based on their SPECpower data. It is important to draw attention to a few data points where EP > 80%, although no servers have a DR above 80%. Recall that the only way a server can have EP > DR is for that server to have sublinear energy proportionality (-LD). A sublinearly energy proportional server consumes less power than a linearly proportional server. Hence, it can have higher EP than DR. Thus, those few servers with EP > 80% in Figure 2b have sublinear energy proportionality. Note that -LD does not always imply high EP. In particular, energy proportionality is affected by two components: dynamic range and the linear deviation. If DR can be improved, then LD improvements have a secondary impact on overall EP. But as DR improvements hit a wall, the only way to improve EP moving forward is to improve LD.

Linear Deviation: Figure 2c shows the relationship between linear deviation and energy proportionality. Unfortunately, the data shows that the majority of servers (at least 80%) are superlinearly proportional (+LD). Hence, there lies potential to improve energy proportionality in current servers by improving their linear deviation.

Proportionality Gap: Figure 2d shows the average proportionality gap of servers at various utilization levels. The curves, from top to bottom, represent servers with low EP (<50%), medium EP (50-75%), and high EP (>75%). Irrespective of the EP level the striking feature is that all servers suffer large proportionality gap at low utilizations. Furthermore, as EP increases, it becomes clear that the majority of the proportionality gap occurs at lower utilizations. As EP improves, *energy disproportionality at lower utilization will be the main obstacle to achieving perfectly energy proportional systems*. Unfortunately, due to limited information reported in SPECpower results, we cannot gain a clear insight as to the fundamental causes of proportionality gap at lower utilization and thus this question remains an open research problem.

Energy Efficiency: We defined energy efficiency as ssj_ops/Watt from the SPECpower data. Figure 2e shows that energy efficiency is strongly correlated with the proportionality gap. The curves, from top to bottom, represents servers with high, medium, and low EP, respectively. Due to the large proportionality gap at low utilization, server energy efficiency is about 30% of the peak efficiency even for servers with relatively high EP. Hence, even if the overall EP of a server improves over time, the energy efficiency will still suffer at low utilizations unless the proportionality gap at low utilizations is reduced. Otherwise, even the highest EP servers can run efficiently only at high utilizations. In order to improve energy efficiency, we should improve efficiency at lower utilization. Unfortunately, as Figure 2f shows, improvements to efficiency at higher utilization has outpaced improvements at lower utilization.

Overcoming the EP wall: In order to improve the energy efficiency of servers, we cannot solely rely on improvements to dynamic range, as has been the case in the past. Therefore, we cannot concentrate on energy efficiency improvements at peak and idle only. As dynamic range is now static, we must focus on improving the linear deviation. As shown previously, servers operate in two distinct energy proportional regions. Servers tend to be perfectly proportional at high utilization (>50%), while disproportional at low utilization (<50%). Therefore, in order to gain the most benefits, we must focus our efforts in the low utilization region. Furthermore, processors are no longer the major consumer of power in servers [31]. In order to reduce energy consumption, new server-level solutions that tackle proportionality gap at low utilizations are needed.

4. Related Work

Power and energy related issues in the context of large scale datacenters have become a growing concern for both commercial industries and the government. Barroso [6] showed that energy-proportionality is a chief concern since most enterprise servers operate at low average utilizations. These concerns have become the source of much active research in the energy proportional computing space. Numerous studies have examined energy efficiency approaches to servers in datacenter settings. These approaches can be classified along three dimensions: spatial granularity (Granularity), the period in which the low power mode is active (Period), and the ability for the low power mode to perform work (Active/Inactive). The granularity refers to whether the low power mode work at the cluster, server, or component level. The period in which the low power mode is active refers to the region of operation that the low power mode exploits. Low power modes can exploit either idle periods (0% utilization), or low

Granularity	Cluster-level		Server-level		Component-level	
Period	**Idle**	**Low Utilization**	**Idle**	**Low Utilization**	**Idle**	**Low Utilization**
Active Low power		Consolidation & Dynamic Cluster Resizing [7, 8]	Somniloquy [1] Barely-alive Servers [3]	KnightShift		DVFS MemScale [10] Heter. Cores [21, 14, 13, 5]
Inactive Low power			Shutdown PowerNap [24]		DRAM Self-refresh Core Parking Disk Spin down	

Table 1: Classification of Server Low-power modes

utilization periods. The ability to perform work refers to whether the low power mode allows the system to continue processing requests. For inactive low power modes, the system cannot processes requests. For active low power modes, the system can still process requests, possibly with lower capability and performance. For example, if a low power mode is an active low power mode and exploits low utilization periods, it means that the low power mode is activated during low utilization periods and can still perform work. Using the three dimensional classification Table 1 bins the most relevant prior work which we will briefly describe next .

Cluster-level techniques: Common techniques such as consolidation and dynamic cluster resizing [7, 8] concentrate workload to a group of servers to increase average server utilization and power off idle machines, improving efficiency and lowering total power usage. Although beneficial, these techniques are not suitable for many emerging workloads in today's datacenter settings. For direct-attached storage architectures or workloads with large distributed data sets, servers must remain powered on to keep data accessible. Furthermore, due to the large temporal granularity of these techniques, they cannot respond rapidly to unanticipated load as it could take minutes to migrate tasks with very large working sets. Under these circumstances, server consolidation is not a viable solution. Our proposed solution will allow significant energy savings even when servers are required to actively operate at low utilization.

Component-level techniques: Component-level energy saving techniques for CPU, memory, and disk covers both active and inactive low-power modes. Active low-power techniques improves the energy-proportionality of components by providing multiple operating efficiencies at different utilization levels. Heterogeneous cores [5, 13, 14, 21], such as Tegra 3 and ARM big.LITTLE, can switch to low-power efficient cores during low-utilization periods, while DVFS and MemScale [10] scales the frequency and power of components depending on utilization levels. Furthermore, inactive low-power techniques, such as DRAM self-refresh, core parking and disk spin down can improve idle power consumption of these components. Most dynamic range improvements seen to date are driven primarily by processor energy efficiency gains. But going forward, no single component dominates overall power usage [31], which may limit the potential of component-level techniques in the future.

Server-level techniques: Server-level techniques aim to put the entire server into a low-power mode. Previous techniques aimed to improve energy efficiency by increasing the dynamic range through lowering the idle power usage and extending the time a system stays in idle. PowerNap [24] exploits millisecond idle periods by rapidly transitioning to an inactive low-power state. DreamWeaver [26] extends PowerNap to queue requests, artificially creating and extending idle periods. Barely-alive servers [3] place the server in a low-power state, but extends idle periods by keeping memory active to process remote I/O requests. Similarly, Somniloquy [1] allows idle computers to supports certain application protocols, such as download and instant messaging. As the number of processors in servers increase, idle periods will become increasingly rare [26, 34]. Thus *active low-power modes that can efficiently operate at low-utilization levels will be the only practical server-level energy saving technique in the future*. Current literature lacks work that exploit low-utilization opportunities. As the data in Section 3 showed, it is critical to tackle the lack of energy efficiency during low-utilization periods. Our work, KnightShift, fills this important gap.

Low-power design: Wimpy nodes [4] aims to save power by running low-power energy-efficiency nodes. Wimpy clusters are limited to workloads that can tolerate higher latency, but may degrade QoS during traffic spikes, requiring over-provisioning [28, 15]. Heterogeneous clusters [9] of brawny and wimpy cores also suffers the same issues as consolidation and task migration. In KnightShift, we can dynamically switch modes to handle latency demands, without the overhead of consolidation due to a tightly-coupled disk subsystem.

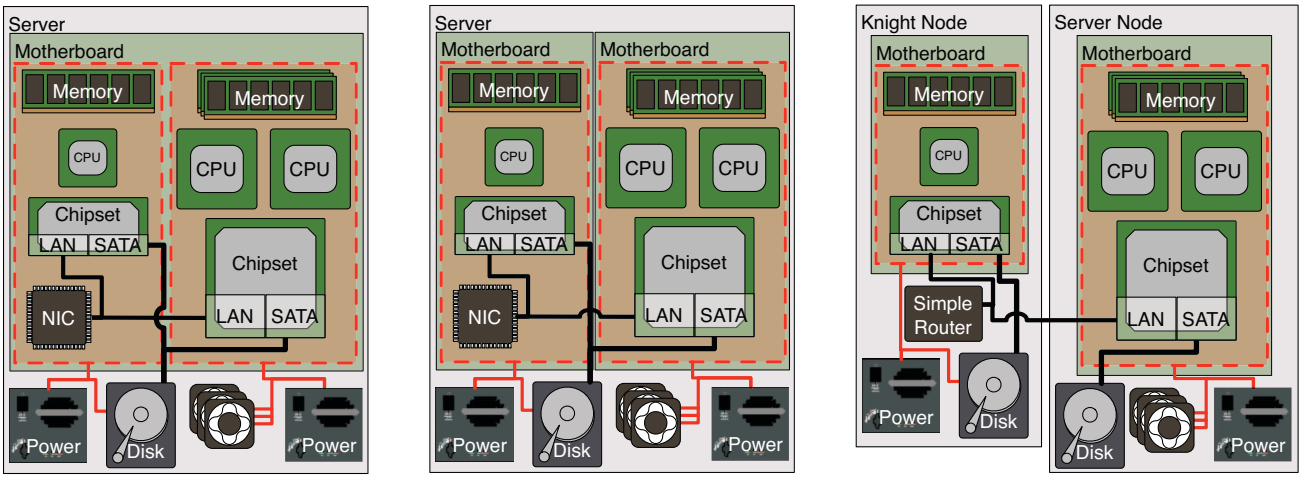
5. KnightShift

In this section we introduce KnightShift, a heterogenous server-level architecture to reduce the proportionality gap of servers at low utilization. KnightShift fronts a high-power primary server with a low-power compute node, called the Knight, enabling two energy-efficient operating regions. We define *Knight capability* as the fraction of throughput that Knight can provide compared to the primary serve. To the best of our knowledge, KnightShift is the first *server-level active low-power mode solution to exploit low-utilization periods*. The fundamental issues limiting energy proportionality have been lack of improvement to dynamic range and linear deviation. While previous techniques only targeted dynamic range, KnightShift extends previous techniques by also targeting linear deviation.

A KnightShift system consists of three components:

I **KnightShift hardware:** The KnightShift hardware consists of a low-power low-performance compute node, called the Knight, paired with a high-power high-performance server. Both the Knight and primary server can be independently powered on and off. Both the Knight and primary server share a common data disk and are able to communicate with one another through traditional network interface. In section 5.1, we will introduce three possible implementations of KnightShift.

Due to low-power demands we assume the Knight has less memory than the primary server. However, certain workloads require large memory-resident datasets, such as scale-out workloads [12], and cannot tolerate smaller memory. These workloads can still benefit from KnightShift by alternatively using low-power mobile memory [23], therefore still benefiting from overall reduced energy savings. Current server motherboards are typically not built to accommodate low-power mobile memory while a Knight can use such a memory type.



(a) Board-level (b) Server-level (c) Ensemble-level (Prototype)

Figure 3: Three proposed implementations of KnightShift. In a board-level implementation the primary server and Knight are integrated into the same motherboard. In a server-level implementation the Knight is a separate add-on board that attaches to SATA port, converting commodity servers into KnightShift systems. In an ensemble-level implementation, only commodity parts are used.

II **System software:** The system software enables several key functionalities required for KnightShift, such as disk sharing, network configuration and remote wakeup of compute nodes. Most operating systems already support the required system software functionality. In section 5.1 we will describe the specifics of system software required to support KnightShift.

III **KnightShift runtime:** The KnightShift runtime is the new software layer that is built specifically for the purpose of operating KnightShift. This runtime layer monitors utilization, makes mode switching decisions, redirect requests between the Knight and the primary server, and coordinates disk access rights to ensure data consistency. We will discuss this runtime in detail in section 5.2 and present our prototype implementation in section 7.1.1.

5.1. KnightShift Implementation Options

We propose three implementations of KnightShift as shown in Figure 3. The preferred choice for implementing KnightShift depends on the usage scenario and level of integration supported by system designers.

Board-level integrated KnightShift: Board-level integrated KnightShift integrates the primary server and Knight onto the same motherboard. Both Knight and primary server have independent memory, CPU, and chipsets. To allow each node to power on/off independently, the motherboard is separated into two power domains (designated by the dotted box). The Knight's power domain comprises of it's memory, CPU, chipset, ethernet, and disks. The Knight's power domain is always on but the primary server's power status is controlled by the Knight. The Knight is capable of remotely waking up the primary server. Existing technology such as wake-on-lan can be used to support remote wakeup. Using wake-on-lan, when the primary server is off, the Knight can send a "magic" packet to the primary server's network interface which in turn will wake up the primary server. All three proposed implementations use wake-on-lan for remote wakeup.

Networking is provided through sideband ethernet, allowing two devices to be exposed through a single physical port to external servers. Both the Knight and primary server would have their own IP address, but only the Knight's IP address would be publicly available. This allows the KnightShift server to appear as a single server on the network, eliminating additional network overheads to adopt KnightShift servers.

Disks are shared between the primary server and Knight through a shared SATA connection. Since SATA currently supports hot-plugging, the system designer can add switching logic to route SATA requests between the primary server and Knight.

Server-level integrated KnightShift: In a server-level integrated KnightShift configuration, the Knight resides on a separate independently powered motherboard. The only shared components between the Knight and primary server is the network and disk. We envision that this approach can be implemented by intercepting the SATA interface and building a Knight which can fit as a hard drive module within the primary server. Hard drive mounts are designed to fit various hard drive sizes. For example, 3.5inch drives comes in 19mm or 25.4mm heights. By using 19mm height drives or 2.5inch drives, we can integrate the Knight into the unused space on the 3.5inch mount. This approach is feasible even today as some potential Knight candidates are as small as credit cards [19].

Since the Knight remains on at all times, it is exposed to the outside world as the only server. Thus, the primary ethernet connection will be on the Knight board. The existing primary server's ethernet is then connected into the Knight board. Thus this approach requires one extra internal ethernet connection compared to board-level integration. This implementation allows us to convert any commodity server to a KnightShift-enabled server without using additional space.

Ensemble-level KnightShift: The ensemble-level implementation uses only commodity parts with no changes to hardware. By using a primary server and a Knight based on commodity computers (such as nettops), a KnightShift system can be implemented. This is the prototype that we will use to evaluate KnightShift in section 7.1. Disk sharing is fulfilled through NFS, with the Knight acting as the NFS server and the primary server mounting the NFS drive. This allows data to persist when the primary server is off. Since the Knight acts as the NFS server this approach requires the Knight to always be on. A router is used to network the Knight and primary server. To the outside world only the Knight's IP address is exposed. The primary

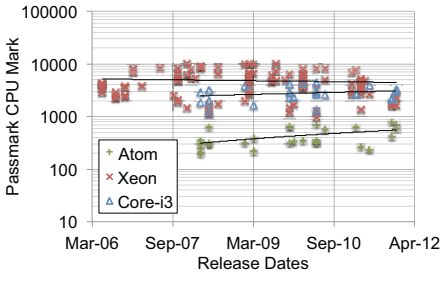


Figure 4: Performance trends of commercial systems.

server communicates to the outside world through the Knight.

Board-level implementation requires the least amount of space, but requires several modifications to the system design. The server-level implementation allows commodity servers to be KnightShift-enhanced, with minimal space requirements. The ensemble-level is the simplest to implement with commodity parts. But this solution can be expensive and may need additional rack space in a data center.

5.2. KnightShift Runtime

In the above section we presented three choices for implementing KnightShift and the basic system software needed for remote wakeup, disk and network sharing. The KnightShift functionality is implemented in a special purpose runtime layer called the KnightShift runtime. The runtime serves the following purposes: 1) Monitor server utilization, 2) Decide on when to switch between Knight and primary server using mode switching policies, 3) Ensure data consistency on shared disk data, 4) Coordinate mode switching, 5) Redirect requests to active node.

Monitoring server utilization and mode switching policies: An essential part of KnightShift is the ability to monitor the utilization of the primary server and Knight to make mode switching decisions. Server utilization monitoring can be carried out simply through the Linux kernel or through third-party libraries.

Whenever the primary server's utilization is low, the Knight will put the primary server to sleep and handle service requests. Whenever the Knight's utilization is too high, it does a remote wakeup of the primary server which then handles service requests. In this paper, our primary goal is to introduce the benefits of KnightShift and thus we use a simple switching policy to determine when to switch modes. In order to maximize power savings, we have to maximize the amount of time that we spend in the Knight. To do so, our simple switching policy aggressively switches into the Knight, and conservatively switches to the primary server. For example, if the Knight is 20% capable, the KnightShift runtime will switch to the Knight whenever the primary server utilization falls below 20%. KnightShift switches back to the primary server only when the Knight's utilization exceeds 100% for at least the amount of time it takes to switch between Knight and primary server, called the transition time.

By maximizing energy savings, we may negatively impact performance as we may stay in the Knight mode during periods where the Knight cannot handle the requests, causing increased response time. Although it is outside the scope of this work, by using a more balanced switching policy, through predicting high utilization periods or carefully chosen timeouts, KnightShift may provide a better balance between energy savings and performance [34].

Data consistency and coordinating mode switching: Recall that in all three implementation the Knight and primary server share the disk data needed for processing service requests. Hence, whenever mode switching is activated, the compute node that is shutting down must flush any buffered disk writes that are cached in memory back to disk and unmount the disk. This allows the complementary node to mount the disk and operate on consistent data. The KnightShift runtime enforces this consistency by coordinating disk access sequence between the two nodes. In section 7.1 we detail our prototype KnightShift system where coordination is carried out through a set of scripts communicating using message passing.

Redirecting requests: There are many ways to forward incoming requests to the active compute node. One approach is to run a simple load balancer software on the Knight, which would require the Knight to remain always on. We take this approach in our prototype KnightShift implementation in section 7.1. It would also be possible to use a hardware component which redirects requests.

5.3. Choice of Knights

We originally considered three options for the Knight: ARM, Atom, and Core i3-based systems. It is currently not feasible to use ARM based systems as a Knight because its capability level (<10%) is simply too low and does not provide ample opportunities to switch to Knight mode. With the emergence of server-class ARM processors, ARM may become a viable Knight option in the near future.

Figure 4 shows the performance growth of Atom and Core i3 as potential Knights compared to a Xeon based server as the primary server. The performance data was obtained from Passmark CPU Mark [18]. Most Atom based systems have one order of magnitude lower capability than a Xeon based server and in the best case they have 20% capability. The performance of Core i3 on the other hand, is within 50% of Xeon based server. Thus Atom and Core i3 can provide Knight capability of up to 20% and 50%, respectively. Although Core i3 based Knights use ~4x more power than Atom based Knights, Core i3 offers more opportunity for the Knight to handle requests from the primary server. In our prototype implementation we used only an Atom based Knight due to limited hardware budget.

Mixed ISA: In our current prototype, all the Knight choices run x86 ISA. Additionally, we ran a fully functional KnightShift prototype using x86+ARM and we didn't encounter any functional difficulties. Many popular applications, such as java, apache and mysql already have ARM binaries. As ARM becomes more powerful and prevalent, mixed ISA KnightShift systems may even become the norm. While the ARM based Knight ran perfectly well in terms of functionality, the latency overhead was too high. Hence we do not consider mixed ISA implementation in the rest of the paper.

6. A Case for Server-level Heterogeneity

In this section we show the potential benefits of KnightShift on top of current production systems. We selected all 291 servers from the SPECpower results and studied how various energy proportionality metrics are affected if that server was enhanced with a Knight. Recall that we define *Knight capability* as the fraction of throughput that the Knight can provide compared to the primary server. By assuming the Knight was created with the same technology as the primary server, the peak and idle power of the Knight, with capability C, can be obtained by theoretically scaling the power using the equation $Power_{Knight} = C^{1.7} * Power_{Primary}$ [5]. For example, if the primary server operates between 100W (idle power)-200W (peak power at 100% utilization), then a 50% capable Knight will operate from 31-62W. We assume the Knight is linearly proportional (LD=0) between its idle and peak power.

Figure 5 shows the effect of KnightShift on the energy proportionality curve, from Figure 1, with a 20% and 50% capable Knight. To

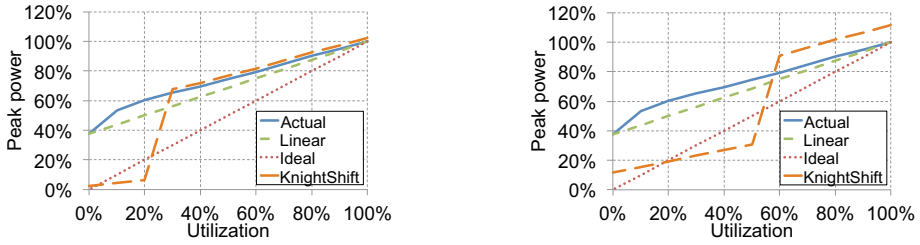


Figure 5: KnightShift enhanced energy proportionality curve

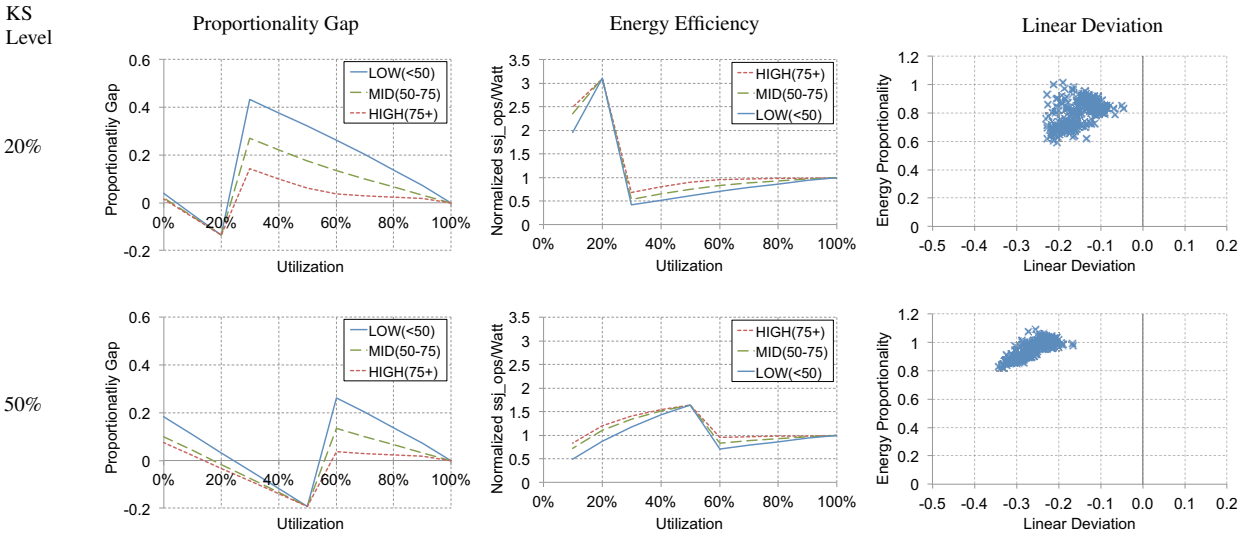


Figure 6: Effect of KnightShift on SPECpower commercial servers

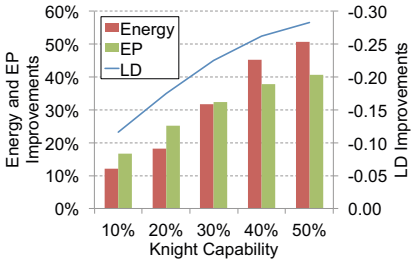


Figure 7: Server Energy, EP, LD improvement with KnightShift

generate this data, we assume that anytime the utilization is within the Knight's capability levels, the Knight will handle that requests. Otherwise, the primary server will handle the request. Note that in KnightShift, the Knight must remain on, which increases the peak power consumption of the server. The reason for this requirement was explained in the previous section. Even with the increase in peak power consumption, we still experience significant power savings because the servers spend the majority of the time in the low utilization regions. (more details will be presented in section 7). The primary server is shut down at low utilizations, allowing the Knight to handle all low utilization requests, significantly decreasing power consumption. Depending on the capability, energy savings vary. But in all cases, we shift the server to -LD domain, but with differing levels of -LD. It is interesting to note that at specific utilization levels, a KnightShift-enabled system can consume less power than an ideal energy proportional system, opening the possibility of servers operating with better efficiency than ideal energy proportionality. For instance, in Figure 5b when the server utilization is approximately between 20% and 50%, the overall power consumption is better than an ideal energy proportional server because the Knight uses less power than an ideal energy proportional server at that utilization.

Figure 6 shows the effect of KnightShift with a 20% and 50% capable Knight on proportionality gap, energy efficiency, and linear deviation (compare to Figure 2).

Proportionality Gap: At 20% capability, the proportionality gap of the KnightShift server is essentially eliminated at utilization below 20%. While in Knight mode, the proportionality gap is negative, meaning that the power used by the Knight at a specific utilization is lower than that of an ideal energy proportional server, as shown in Figure 5. At 50% capability, the proportionality gap is greatly reduced in the 0%-25% utilization range as compared to Figure 2d, while the proportionality gap is eliminated from 25%-50% utilization range. The reason for non-zero proportionality gap at the lower range is because of the power consumed by the Knight itself. As long as the proportionality gap exists at low utilization, KnightShift should benefit that server.

Energy Efficiency: The energy efficiency curves for the 20% and 50% Knight capabilities are shown in Figure 6. KnightShift enhances server's energy efficiency and allows them to run at or better than peak efficiency (great than 1 in the figure) even at lower utilization. The improvement is directly correlated with the reduction in proportionality gap. 20% capable Knights operate above peak efficiency between 0-25% utilization range. 50% capable Knights operate at above peak efficiency from 25%-50% utilization range, and

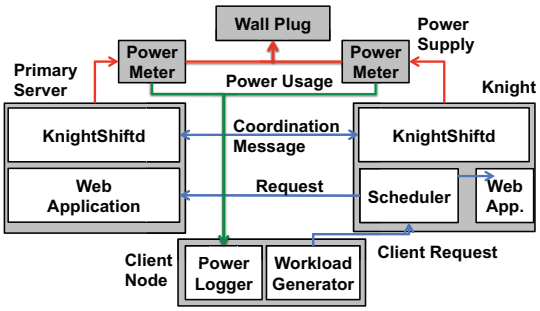


Figure 8: KnightShift Prototype setup

just below peak efficiency below 25% utilization. This data shows that KnightShift energy efficiency is substantially higher than the baseline shown in Figure 2e.

Linear Deviation: KnightShift effectively shifts all servers from +LD to -LD range. Improving LD is the only option to improve energy proportionality when dynamic range improvements are not feasible. With a 20% capable Knight, the lowest EP server amongst the 291 servers jumped from 20% to 60%. Thus KnightShift is able to improve the EP of servers by allowing commodity servers to exhibit -LD. For 50% capable Knights, we even see servers with EP > 1, indicating that KnightShift effectively closed the proportionality gap.

EP and Energy Savings: To evaluate energy savings, we assume server utilization distribution similar to Google datacenter servers in [6]. Figure 7 shows the average improvements to LD, EP and potential energy savings. For 20% Knights, we experience average EP improvements of 25%, average energy savings of 18% and average LD decreased by .175. As Knight capability increases, energy savings grows due to more opportunity to be in the Knight. For 50% capable Knights, we experience average energy savings of 51% and average EP improvement of 41%. By having KnightShift being configurable, vendors may pick a KnightShift implementation that is best suited for their performance and energy budget goals.

7. Evaluation

In this work we evaluate KnightShift using two approaches. First, we present a KnightShift prototype and run a real-world workload, WikiBench [20], to demonstrate feasibility and performance of KnightShift under realistic conditions. A prototype implementation, however, provides limited flexibility to change the hardware configuration parameters. Hence, we developed a queueing model based simulator that is validated against the prototype implementation. We then use the simulator to conduct a broad design space exploration using traces collected from USC's production datacenter.

7.1. Prototype Evaluation

7.1.1. Prototype Setup The KnightShift prototype is similar to Figure 3c. The exact experimental setup is shown in Figure 8 . In this setup the Knight is a Shuttle XS35 slim PC with a 1.8GHz Intel Atom D525, 1GB of ram, 500GB hard drive and operates from 15W-16.7W. At idle, the CPU and memory consumes 9W, with the disk and motherboard consuming 6W. The primary server is a Supermicro server with dual 2.13GHz 4-core Intel Xeon L5630, 36GB of ram, 500GB hard drive and consumes from 156W to 205W while active. Recall that we assume that it is reasonable for the Knight to have less memory than the primary server as the performance impact due to less memory is accounted for in the capacity measurement

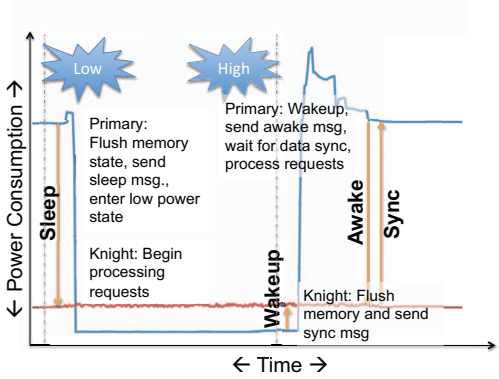


Figure 9: Coordination of KnightShift servers

of the Knight. For our particular setup, our Knight is not capable of supporting memory size larger than 1GB.

Using SPECpower results we determined that our primary server has an EP of 24%. By enhancing the primary server with a Knight, we improved the EP to 48%! Note that although our primary server has a relatively low EP, if we use a significantly higher 70% EP server, our prototype Knight can still provide EP improvements. KnightShift is not meant to compete directly with servers that are already highly proportional across all utilization levels. KnightShift improves EP of servers that have large proportionality gap at low utilizations.

The primary server can turn on/off in 20/10 seconds, respectively. During Knight mode, the primary server is placed in hibernate mode, where the system is shutdown except for the network interface. It is also possible to place the primary server in suspend mode quicker than shutting down, but at the cost of higher idle power. Both nodes run Ubuntu Linux with all power-saving features enabled (DVFS, drive spin-down, etc). We determined that our Knight is 15% capable compared to the primary server using throughput measurements from apachebench [16]. In other words, the Knight's throughput is 15% of the peak throughput of the primary server. Request generation and power measurement data collection are handled by a separate client node which is not part of the prototype. The power consumption of the primary server and Knight are measured using two *Watts Up? Pro* power meters with data logged to the client node.

KnightShift Runtime: In our prototype setup, both nodes shares data through NFS, with the Knight acting as the NFS server. In order to force data consistency, we require coordination between both nodes. Thus, we require runtime software for KnightShift support. This software will handle both utilization monitoring and coordination. While there are many options for implementation, here we only present one particular implementation used in our prototype.

To support and enforce the KnightShift functionality, both nodes run a daemon, called KnightShiftd, to support utilization monitoring and coordination messages. KnightShiftd is implemented as a set of scripts and acts as the control center for the KnightShift system. KnightShiftd monitors the utilization of the node it's running on and makes mode switching decisions. To support redirection of requests, the Knight runs a scheduler, which acts as a simple load balancer to forward the requests to the active node.

Communication between both nodes takes place through messages. Figure 9 highlights the processes of switching between nodes, which also enforces data consistency. Upon entering a low utilization period, the KnightShiftd daemon will detect the low utilization of the primary server and initiates a mode switch. The primary server will flush its memory state to ensure that the latest data is up to date in the

	Response Time		Energy
	Average	95th	Consumption(KWH)
Prototype			
Baseline	144ms	249ms	23.27
KnightShift	150ms	296ms	15.35
Improvement	-4%	-19%	34%
Simulation			
Baseline	1.00	1.66	23.27
KnightShift	1.12	2.00	15.11
Improvement	-12%	-21%	35%
Error	8%	2%	1%

Table 2: Energy consumption and response time of Wikibench using our KnightShift prototype and simulator.

disk. When this completes, KnightShiftd will send a *sleep* message to the Knight and begin to power down. The KnightShiftd daemon on the Knight system receives the *sleep* message from the primary server, which is an indication that the Knight should begin processing requests. The Knight will process low utilization requests until it reaches a high utilization region. At this point, the daemon on the Knight will send a *Wakeup* message (through wake-on-lan) to wake up the primary server. When the primary server has booted up, the daemon on the primary server gets ready to process requests. It will send an *awake* message to the Knight. At this point, the Knight will flush its data and send a *sync* message, indicating to the primary server that it can resume processing requests.

7.1.2. Prototype Results To verify the correctness of KnightShift and to evaluate KnightShift under realistic workloads, we cloned Wikipedia and benchmarked it using real-world Wikipedia request traces. Wikipedia consists of two main components, Mediawiki, the software wiki package written in PHP, and a backend mySQL database. For our clone, we used a publicly available database dump from January 2008, containing over 7 million articles. We replayed a single-day Wikipedia access trace [32], which follows a diurnal sinusoidal pattern, using WikiBench [20], a Wikipedia based web application benchmark. Detailed WikiBench workload utilization profile for this case study is presented in [33].

The first three rows of data of Table 2 show the energy consumption and the 95th percentile response time of our KnightShift prototype compared to the baseline primary server. Service Level Agreements (SLA), which sets per-request latency targets, are typically based on 95th percentile latency [25].

We define the baseline as a system where all requests are always handled by the primary server. KnightShift is able to achieve 34% energy savings with only 19% impact on 95th percentile response time. This latency impact is mainly due to the single-threaded performance of the Knight rather than penalties due to switching between the Knight and primary server (Note that the average response time only increased by 4%). When running Wikibench only on the Atom-based Knight, we experience 95th percentile response time of 323ms for successfully completed requests. Thus, *KnightShift's 95th percentile response time is bounded by that of the Knight*. By using higher single-threaded performing processors, such as Intel Core i3, we should expect to experience response time bounded by the response time of the Core i3.

7.2. Trace-based Evaluation

7.2.1. Trace-based Setup While a prototype implementation provides great confidence regarding the functional viability and realistic improvement results, it also limits our ability to alter some of the critical design space parameters, such as Knight capability level, Knight performance, and Knight transition time. In order to fully explore these variables, we present KnightSim, a trace-driven KnightShift system simulator validated against our prototype system. During simulation runs, KnightSim replays the utilization traces collected from our production datacenter on a modeled KnightShift system. KnightShift is modeled as a G/G/k queue, where the arrival rate is time-varying based on the utilization trace, the service rate is exponential with a mean of 1 second, and varying k servers modeling the capacity of the Knight and primary server. Because we do not have measured response time from out datacenter traces, we arbitrarily set the service rate to 1 second and report relative performance impact.

		Utilization		Δ Utililization	
Server	Type	$\bar{x}$	σ	$\bar{x}$	σ
aludra	stu. timeshare	3.87	3.12	0.59	0.84
email	email store	3.26	1.74	0.78	1.20
girtab	stu. timeshare	0.83	2.42	0.73	1.94
msg-mmp	email services	32.62	13.60	2.64	2.76
msg-mx	email services	19.23	7.41	1.69	2.30
msg-store	email store	11.05	5.88	2.39	2.72
nunki	stu. timeshare	4.86	10.85	1.98	4.50
scf	file server	5.47	4.19	1.15	1.65

Table 3: Datacenter trace workload characteristics

Datacenter Utilization Traces: In order to rigorously evaluate KnightShift under various workload patterns, we collected minute-granularity CPU and I/O utilization traces from our production datacenter over 9 days. The datacenter serves multiple tasks, such as e-mail store(email, msg-store1), e-mail services (msg-mmp, msg-mx), file server (scf), and student timeshare servers (aludra, nunki, girtab). Each task is assigned to a dedicated cluster, with the data spread across multiple servers. Selected servers within a cluster exhibit a behavior representative of each server within that cluster.

Table 3 shows the properties of each server workload along with its corresponding utilization and burstiness characteristics. Some of our servers (aludra, email, girtab, nunki, scf) run at less than 20% CPU utilization for nearly 90% of their total operational time [33]. These traces reaffirms prior studies that CPU utilization reaches neither 100% nor 0% for extended periods of time [6, 11, 27]. We also collected a second-granularity traces for a subset of these servers and found that there is a high correlation to minute-granularity. Thus, we use the minute-granularity data for the rest of the paper. The burstiness of the workload is characterized by $\sigma_{utilization}$, the standard deviation of the workload's utilization, and Δutilization, the change in utilization from sample to sample. $\sigma_{utilization}$ tells us how varied the utilization of the server is, while the Δutilization tells us how drastic the utilization changes from sample to sample. For example, nunki has a wide operating utilization range with large variation in utilization from sample to sample. More details of our datacenter traces are presented in [33].

Modeling Knight capability: Knight capability is modeled by varying the system capacity, k. For example, if we have a 10% Knight, then k = 10 in our G/G/k queueing model when operating in Knight mode. When the primary server becomes active then k = 100.

Scaling Power Consumption: To faithfully scale the power of the Knight as its capability changes, we assume simply that the power consumption of the CPU scales quadratically with performance. The quadratic assumption is based on historical data [5] which showed that power consumption increased in proportions to $performance^{1.7}$. We assume this is a reasonable assumption due to the fact that even if the Knight and primary server require similar infrastructure (such as same size memory), the Knight can tradeoff performance by us-

ing low-power components (such as low-power mobile memory), therefore, most components can scale.

Modeling Power: Our power model is based on our prototype system to allow us to compare and validate KnightSim. Through online instrumentation, we collect the utilization vs power data for both the Knight and primary server. We use this utilization-power data in our simulations; whenever a Knight is active at a given utilization we use the power consumption data collected from our prototype Knight. Similarly whenever the primary server is operating at a given utilization, we use the power consumption collected from the primary server in our prototype. It is also possible to generalize the power model and use a linear power model validated in [11].

In order to capture the energy penalty of transitioning to/from Knight, we conservatively model the transition power as a constant power during the entire wakeup period equal to the peak transition power. We determined empirically that the peak transition power for the primary server is 167W.

Arrival Rate and Latency Estimation: Our datacenter traces only have CPU and I/O utilization per second without individual request information. By assuming a mean service time of 1 second for each request, we can estimate a time-varying arrival rate through our utilization trace. For example, 50% utilization would correspond to an arrival rate of 50 requests per second. Through the simulated queueing model, we can obtain a relative average and 95th percentile latency of a KnightShift system compared to a baseline system.

Modeling Single-threaded Performance: We vary the queueing model's service time to model the performance difference of the Knight and primary server. We cannot infer single-threaded performance directly from processor frequency because single-threaded performance is based on frequency and the underlying architecture. Instead, we compare the 95th percentile latency of the Knight and primary server and scale the service time accordingly. For example, our primary server has tail latency of 249ms while our Knight has tail latency of 323ms as shown in section 7.1.2. As we do not have direct access to the datacenter servers, nor can we replicate the proprietary applications on our Knight, we cannot collect response times for the primary server and Knight for each individual workload. Therefore, in our model, we assume that all workloads experience similar performance slowdown due to the Knight similar to WikiBench, where the service time is increased by a factor of 1.3 compared to baseline.

Simulator Validation: We validated our trace-based emulation by collecting utilization traces from our WikiBench run and replayed the utilization traces through the trace emulator. In addition, we validated our power results against our prototype system by running a CPU and I/O load generator to match the utilization of the traces. Table 2 shows the results of our validation run. 95th percentile latency and energy consumption improvement results from KnightSim are all within 2% of our prototype system.

7.2.2. Sensitivity Analysis In this section we explore KnightShift's sensitivity to various parameters such as workload utilization patterns, Knight capability, and transition times.

Sensitivity to Workload patterns: We used KnightSim to simulate KnightShift running a variety of workload patterns by driving the queueing model with traffic patterns from Table 3. The energy and latency impact are shown in Table 4. Recall that our Atom-based Knight has a 95% response time that is 30% greater than the primary server, thus we consider any latency above 30% to be attributable to the KnightShift mechanism overhead. For workloads with low burstiness (aludra, email, msg-mmp, wikibench), we experience relatively low response time impact (<10%).

Trace	Energy Savings	95% Latency Impact
aludra	87.9%	40%
email	85.5%	37%
girtab	87.2%	49%
msg-mmp	-6.7%	7%
msg-mx	7.2%	254%
msg-store	34.5%	53%
nunki	67.7%	5989%
scf	77.5%	46%
wikibench	35.1%	21%

Table 4: Energy savings and latency impact wrt Baseline of a 15% Capable KnightShift system

For moderately bursty workloads (girtab, msg-store, scf), we experience latency impact within 25% of the Atom-based Knight. For these workloads, the majority of the latency impact occurs during the transition from the Knight to primary server when the Knight is handling requests that it cannot handle until the primary server is ready. These bursty behaviors tend to be periodic, thus it would be possible for KnightShift to learn day-to-day utilization patterns and proactively switch to the primary server to handle these high-utilization bursty periods, negating the high latency impact. This topic is outside the scope of the paper and will be explored in future work.

For very bursty workloads with high utilization (msg-mx, nunki), we experience the most latency impact, as expected. KnightShift does not handle scenarios where the workload switches quickly between very low and high utilization. In these scenarios, the workload may benefit from a higher capacity Knight.

Almost all workloads experience energy saving benefits from KnightShift with the exception of workloads with mostly high utilization periods. There are no benefits from using KnightShift for workloads that operate mostly at utilization above the capability of the Knight, hence such workloads don't need KnighShift support to begin with. For these cases, this may even lead to an energy penalty (msg-mmp) due to running the Knight alongside a heavily utilized primary server.

For most other workloads (aludra, email, girtab, scf, wikibench), we can experience an average of 75% energy savings with tail latency within 9% of the Atom-based server.

Sensitivity to Knight Capability: Figure 10 shows the effect of Knight capability levels on energy savings and 95th percentile response time. As Knight capability increases up to 50%, so does energy savings due to more opportunity for the system to stay in the Knight mode. Although the Knight uses more power at higher capability levels, increased energy savings from time spent in the Knight offset the Knight's higher power.

As Knight capability increases, up to a limit of around 50%, the primary server spends more time sleeping, resulting in latency converging to the 95th percentile latency of the Knight. At low Knight capability, especially for capability less than 20%, KnightShift thrashes; Knight cannot handle the tasks when switched to the Knight mode and these tasks endure long latency while waiting for the primary server to wakeup. Some workloads (msg-mx and nunki) experience latency penalties beginning at higher capability. These workloads do not experience latency impact at lower capabilities since KnightShift rarely switches to the Knight mode and hence the primary server handles nearly all the requests due to the high utilization demands. But when the Knight capability increases the system occasionally

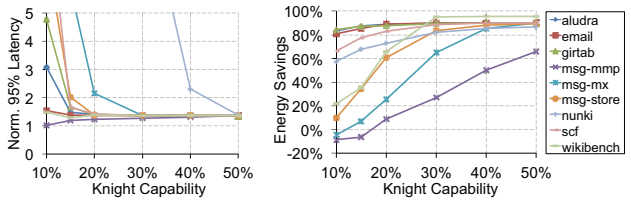


Figure 10: Effect of Capability on Latency and Energy

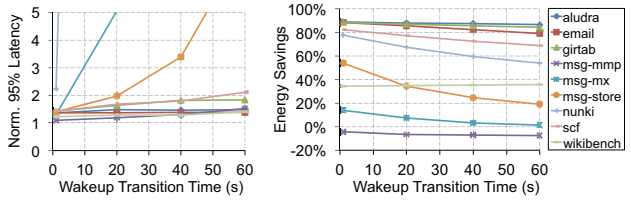


Figure 11: Effect of Wakeup transition time

switches to the Knight mode and the Knight is quickly saturated. Hence, the workload switches back to the primary server leading to latency penalties. KnightShift is in fact unnecessary for these workloads. Thus, for certain workloads with stringent QoS bounds KnightShift may not be an ideal solution.

Sensitivity to Transition Time: For brevity, we only present wakeup transition time. The effect of sleep transition time is similar. Figure 11 shows the effect of wakeup transition time on energy savings and 95th percentile response time. In general, as transition time increases, we experience less energy savings due to the primary server using power but not doing work, while the Knight is still handling requests it potentially cannot handle. This is reflected in an increase in 95th percentile latency as transition time increases.

Sensitivity to single-threaded performance: The tail latency of KnightShift is determined by the Knight. If the SLAs demand very tight latency slack (less than 20%), then it is best to use low-power processors, such as Core i3, as Knights instead of extremely low power Atom boards.

8. TCO

To study the effect of KnightShift on TCO of an entire datacenter, we use a publicly available cost model [17]. The model assumes an 8MW power budget where facility and IT capital costs are amortized over 15 and 3 years, respectively. The model breaks down TCO into server, networking, power distribution and cooling, power, and other infrastructure costs. We assume that KnightShift has no impact on rack density, with power budget as the sole limiting factor. In Table 5, we present our cost breakdown for our primary server and our Knight. We broke down cost into memory, storage, processor, and other system components. Other system components includes motherboard, chipset, network interface, fans, and other on-board components. A significant portion of the energy savings derive from other system components. This is due to the fact that many of these components are energy-disproportional, such as chipset, network interface, fans, and sensors. For example, the power consumption of motherboard components, such as chipset and network interface, are mostly constant with utilization. But with KnightShift, when we switch to the Knight, we could use a low-power mobile chipset (such as for Atom) rather than a higher power chipset (such as for Xeon) to save power. Performance is based on the SPECpower benchmark. An integrated version of KnightShift is expected to consume less power and have lower cost but we assume our prototype implementation of KnightShift to present worst-case TCO.

	Primary Server		Knight	
	Cost	Power(W)	Cost	Power(W)
Memory	$248	40	$20	3
Storage	$130	20	$70	18
Processor	$1102	70	$69	12
Other System Components	$350	75		
Total	$1830	205	$159	33
No. Servers	37361		34483	

Table 5: Cost breakdown of primary server and Knight based on prototype KnightShift system. Other system components include motherboard, chipset, network interface, fans, and other on-board components.

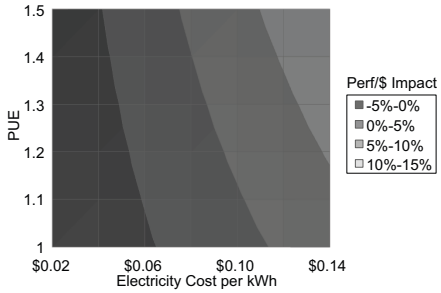


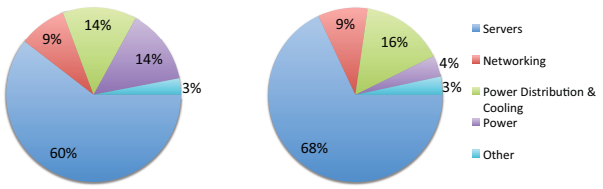
Figure 12: TCO breakdown across PUE and Energy Cost

We present TCO on a monthly basis as Performance per TCO Dollar spent (Perf/$), an important metric in TCO-conscious datacenters [22]. These results make the worst case assumption that both the Knight and primary server are always ON. Figure 12 shows the effect of PUE and electricity cost per kWh on Perf/$. There are two distinct regions, one where Perf/$ is improved, and one where Perf/$ is impacted. Due to the increased peak power usage of Knightshift and the fixed power budget of the datacenter, we suffer a decrease in total datacenter performance. Although there is a reduction in the number of servers due to peak power constraint, we do not always suffer any loss in Perf/$. In regions of higher electricity prices and higher PUE, it is easier to recoup the cost of KnightShift hardware due to more monetary savings per watt. For cases with high PUE and electricity cost, we experience up to 14% improvement to Perf/$. Only at very low electricity prices do we see a negative impact in Perf/$, due to the hardware cost outweighing the potential in energy savings. Note that even with PUE of 1, KnightShift can still provide Perf/$ advantages with electricity prices above $0.07 per kWh.

Figure 13 shows the TCO breakdown across server and infrastructure for PUE of 1.45 and electricity cost of $0.07. Although the total cost of servers is higher with KnightShift (68% total cost vs 60% in the baseline), the power budget improvements (from 14% to 4%), more than makes up for the difference, resulting in TCO savings of 11% monthly. Even by accounting for the lower number of servers, Perf/$/month improved by 4% compared to baseline.

9. Conclusion

Energy proportionality of computer systems has been increasing over the past few years. We introduce several metrics to analyze energy proportionality which shed light into why proportionality has not improved uniformly across all utilization levels. We show that servers exhibit significant proportionality gap at low utilizations. With the pervasiveness of multicores, servers in future will be rarely idle and hence energy saving techniques must now tackle the proportionality gap at low server utilization levels. We introduce KnightShift, a



(a) baseline (b) knightshift

Figure 13: TCO breakdown across servers and infrastructure

server-level heterogeneous architecture that fronts a primary server with a low-power compute node. By operating KnightShift at two levels of efficiency, we convert any server to exhibit sublinear energy proportionality, drastically improving energy proportionality. In our prototype KnightShift implementation with a 15% capable Atom-based Knight, we achieve a 2x improvement in energy proportionality (from 24% to 48%) due to improvements to both dynamic range and proportionality linearity. We demonstrated energy savings of 35% with latency bounded by the latency of the Knight using a real-world Wikipedia workload. In addition, we rigorously evaluated our prototype using various production datacenter traces and experience up to 75% energy savings with tail latency increase of about 9%. Through publicly available cost models, we also showed that KnightShift can improve performance per TCO dollar spent up to 14%. Our work hopes to motive future work in system-level active low-power modes that exploits low-utilization periods.

Acknowledgement

We would like to thank the anonymous reviewers for their valuable comments. We also thank Sabyasachi Ghosh and Mark Redekopp for their early contributions which inspired this work. This work was supported by DARPA-PERFECT-HR0011-12-2-0020 and NSF grants NSF-1219186, NSF-CAREER-0954211, NSF-0834798.

References

[1] Y. Agarwal *et al.*, "Somniloquy: augmenting network interfaces to reduce PC energy usage," in *NSDI'09: Proceedings of the 6th USENIX symposium on Networked systems design and implementation*, Apr. 2009.

[2] H. Amur and K. Schwan, "Achieving power-efficiency in clusters without distributed file system complexity," in *ISCA'10: Proceedings of the 2010 International Conference conference on Computer Architecture*, Jun. 2010, pp. 222–232.

[3] V. Anagnostopoulou *et al.*, "Energy conservation in datacenters through cluster memory management and barely-alive memory servers." in *WEED '09: Workshop on Energy-Efficient Design*, 2009.

[4] D. G. Andersen *et al.*, "FAWN: a fast array of wimpy nodes," in *SOSP '09: Proceedings of the 22nd Symposium on Operating Systems Principles*, Oct. 2009.

[5] M. Annavaram, E. Grochowski, and J. Shen, "Mitigating Amdahl's law through EPI throttling," in *ISCA'05: Proceedings of the 32nd international symposium on Computer Architecture*, 2005, pp. 298–309.

[6] L. A. Barroso and U. Holzle, "The case for energy-proportional computing," *Computer*, vol. 40, no. 12, pp. 33–37, dec 2007.

[7] J. S. Chase *et al.*, "Managing energy and server resources in hosting centers," in *SOSP '01: Proceedings of the 18th Symposium on Operating Systems Principles*, Dec. 2001.

[8] G. Chen *et al.*, "Energy-aware server provisioning and load dispatching for connection-intensive internet services," in *NSDI'08: Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, Apr. 2008.

[9] B.-G. Chun *et al.*, "An energy case for hybrid datacenters," *SIGOPS Operating Systems Review*, vol. 44, no. 1, Mar. 2010.

[10] Q. Deng *et al.*, "MemScale: active low-power modes for main memory," in *ASPLOS '11: Proceedings of the 16th International Conference on Architectural support for programming languages and operating systems*, Mar. 2011.

[11] X. Fan, W.-D. Weber, and L. A. Barroso, "Power provisioning for a warehouse-sized computer," in *ISCA'07: Proceedings of the 34th international symposium on Computer architecture*, 2007, pp. 13–23.

[12] M. Ferdman *et al.*, "Clearing the clouds: a study of emerging scale-out workloads on modern hardware," in *ASPLOS '12: Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems*, Mar. 2012, pp. 37–48.

[13] S. Ghiasi, "Aide de camp: asymmetric multi-core design for dynamic thermal management," Ph.D. dissertation, 2004.

[14] E. Grochowski *et al.*, "Best of both latency and throughput," in *Proceedings of International Conference on Computer Design*, 2004, pp. 236–243.

[15] U. Hölzle, "Brawny cores still beat wimpy cores, most of the time." *IEEE Micro*, 2010.

[16] http://httpd.apache.org/docs/2.0/programs/ab.html, "ab - apache http server benchmarking tool."

[17] http://perspectives.mvdirona.com, "Cost of power in large-scale data centers."

[18] http://www.cpubenchmark.net/, "Passmark cpu benchmark."

[19] http://www.fit-pc.com/web/fit pc/, "fit-pc2."

[20] http://www.wikibench.eu, "Wikibench."

[21] R. Kumar *et al.*, "Single-ISA Heterogeneous Multi-Core Architectures: The Potential for Processor Power Reduction," in *MICRO 36: Proceedings of the 36th International Symposium on Microarchitecture*, Dec. 2003, pp. 81–92.

[22] P. Lotfi-Kamran *et al.*, "Scale-out processors," in *ISCA '12: Proceedings of the 39th International Symposium on Computer Architecture*, Jun. 2012, pp. 500–511.

[23] K. T. Malladi *et al.*, "Towards energy-proportional datacenter memory with mobile DRAM," in *ISCA '12: Proceedings of the 39th International Symposium on Computer Architecture*, Jun. 2012, pp. 37–48.

[24] D. Meisner, B. T. Gold, and T. F. Wenisch, "PowerNap: eliminating server idle power," in *ASPLOS '09: Proceeding of the 14th International Conference on Architectural support for programming languages and operating systems*, Feb. 2009, pp. 205–216.

[25] D. Meisner *et al.*, "Power management of online data-intensive services," in *ISCA'11: Proceeding of the 38th international symposium on Computer architecture*, Jun. 2011, pp. 319–330.

[26] D. Meisner and T. F. Wenisch, "DreamWeaver: architectural support for deep sleep," in *ASPLOS '12: Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems*, Mar. 2012, pp. 313–324.

[27] P. Ranganathan *et al.*, "Ensemble-level Power Management for Dense Blade Servers," in *ISCA '06: Proceedings of the 33rd international symposium on Computer Architecture*, Jun. 2006, pp. 66–77.

[28] V. J. Reddi *et al.*, "Web search using mobile cores: quantifying and mitigating the price of efficiency," in *ISCA'10: Proceedings of the 37th international symposium on Computer architecture*, Jun. 2010, pp. 314–325.

[29] F. Ryckbosch, S. Polfliet, and L. Eeckhout, "Trends in Server Energy Proportionality," *Computer*, vol. 44, no. 9, pp. 69–72, 2011.

[30] G. Semeraro *et al.*, "Dynamic frequency and voltage control for a multiple clock domain microarchitecture," in *MICRO 35: Proceedings of the 35th international symposium on Microarchitecture*, Nov. 2002.

[31] D. Tsirogiannis, S. Harizopoulos, and M. A. Shah, "Analyzing the energy efficiency of a database server," in *SIGMOD '10: Proceedings of the 2010 International Conference on Management of Data*, Jun. 2010.

[32] G. Urdaneta, G. Pierre, and M. van Steen, "Wikipedia workload analysis for decentralized hosting," *Computer Networks: The International Journal of Computer and Telecommunications Networking*, vol. 53, no. 11, Jul. 2009.

[33] D. Wong and M. Annavaram, "Evaluating a prototype knightshift-enabled server," in *WEED '12: Workshop on Energy-Efficient Design*, 2012.

[34] D. Wong and M. Annavaram, "Scalable System-level Active Low-Power Mode with Bounded Latency," University of Southern California, Tech. Rep. CENG-2012-5, 2012.

[35] www.spec.org/power_ssj2008/, "Spec power_ssj2008."

2012 IEEE/ACM 45th Annual International Symposium on Microarchitecture

Rethinking DRAM Power Modes for Energy Proportionality

Krishna T. Malladi† Ian Shaeffer‡ Liji Gopalakrishnan‡
David Lo† Benjamin C. Lee§ Mark Horowitz†
Stanford University † Rambus Inc‡ Duke University§
{ktej, davidlo, horowitz}@stanford.edu †, {ians, lijig}@rambus.com‡, {benjamin.c.lee}@duke.edu

Abstract

We re-think DRAM power modes by modeling and characterizing inter-arrival times for memory requests to determine the properties an ideal power mode should have. This analysis indicates that even the most responsive of today's power modes are rarely used. Up to 88% of memory is spent idling in an active mode. This analysis indicates that power modes must have much shorter exit latencies than they have today. Wake-up latencies less than 100ns are ideal.

To address these challenges, we present MemBlaze, an architecture with DRAMs and links that are capable of fast powerup, which provides more opportunities to powerdown memories. By eliminating DRAM chip timing circuitry, a key contributor to powerup latency, and by shifting timing responsibility to the controller, MemBlaze permits data transfers immediately after wake-up and reduces energy per transfer by 50% with no performance impact.

Alternatively, in scenarios where DRAM timing circuitry must remain, we explore mechanisms to accommodate DRAMs that powerup with less than perfect interface timing. We present MemCorrect which detects timing errors while MemDrowsy lowers transfer rates and widens sampling margins to accommodate timing uncertainty in situations where the interface circuitry must recalibrate after exit from powerdown state. Combined, MemCorrect and MemDrowsy still reduce energy per transfer by 50% but incur modest (e.g., 10%) performance penalties.

1. Introduction

In an era of big data and datacenter computing, memory efficiency is imperative. More than 25% of datacenter energy can be attributed to memory and this fraction will only grow with demands for memory capacity [13, 24, 28].

Recent efforts to improve efficiency study memory that is active and transferring data. The resulting architectures focus on reducing energy per transfer. By tailoring DRAM page width, memory core energy is made proportional to the amount of data requested [2, 37, 41]. However, none of these architectures address a different source of inefficiency: idle memories kept in an active power mode.

One approach to address this problem is to use mobile-class DRAM [27] which have much lower active idle power. But using LPDDR2 requires a static decision to trade bandwidth for efficiency. Alternatively, we could use dynamic powerdown modes but controllers have difficulty invoking them. Transfers are separated by idle periods but they are often too short to justify powerdown.

Indeed, witness the sophistication and complexity of efforts in the compiler, operating system, and architecture to consolidate memory activity to a small number of active ranks [14, 15, 23]. By attempting to lengthen idle periods in other ranks, these approaches acknowledge the unwieldy nature of today's power modes and build systems to accommodate them.

In this paper, we present a fundamentally different approach. Instead of shaping memory activity to produce idleness suited to existing power modes, we re-think the power modes themselves. In an application-driven approach, we model and characterize inter-arrival times for memory requests to determine the properties an ideal power mode should have. This analysis indicates that power modes must have much shorter exit latencies than they have today.

To architect power modes with fast exits, we identify the key contributor to powerup latency: DRAM timing circuitry. The most efficient modes turn off delay-locked loops (DLLs) and clocks. But turning them on again requires expensive recalibration (e.g., 700+ns). Few applications have idle periods long enough to justify this latency. Thus, existing modes offer an unattractive energy-delay trade-off.

We improve this trade-off with a new I/O architecture that shifts timing circuitry from DRAMs to the controller while preserving high bandwidth. In this architecture, the first transfer after wake-up completes in a few nanoseconds. Such responsiveness is orders of magnitude faster than the exit latency of today's most efficient power mode, which must recalibrate timing after wake-up. We make the following contributions:

- **Understanding Power Mode Inefficiency.** Even the most responsive of today's power modes are rarely used. Up to 88% of memory time is spent idling in an active mode. Addressing limitations in existing DRAMs could improve energy efficiency by 40-50%.
- **Understanding Memory Activity.** We study memory activity and its implications for power mode design. A probabilistic analysis establishes a clear path from fast wake-up to attractive energy-delay trade-offs. A workload characterization indicates wake-up in $\leq$100ns is ideal.
- **Rethinking Power Modes.** We present *MemBlaze*, a DRAM I/O architecture that is capable of fast wake-up while ensuring high bandwidth. Alternatively, we propose two new mechanisms: *MemCorrect*, which detects timing errors, and *MemDrowsy*, which lowers transfer rates to widen timing margins. These architectures allow memory transfers immediately after wake-up.
- **Saving Energy.** MemBlaze reduces energy per transfer by up to 50% with negligible performance penalty since data transfers begin immediately after wake-up. If timing is less than perfect, a combination of MemCorrect and MemDrowsy provide similar energy savings with a 10% performance penalty incurred to correct timing errors.

2. Background and Motivation

Today's DRAM interfaces provide performance but dissipate high idle power. Moreover, these interfaces include power modes which are disconnected from architectural requirements. To address these challenges, we architect new DRAM interfaces for fast transitions between power modes.

2.1. DRAM Systems

Each DRAM device contains two-dimensional arrays of memory cells. Multiple devices comprise a rank and multiple ranks share a data bus. Figure 1 illustrates a memory system with four ranks that

1072-4451/12 $26.00 © 2012 IEEE
DOI 10.1109/MICRO.2012.21

IEEE computer society

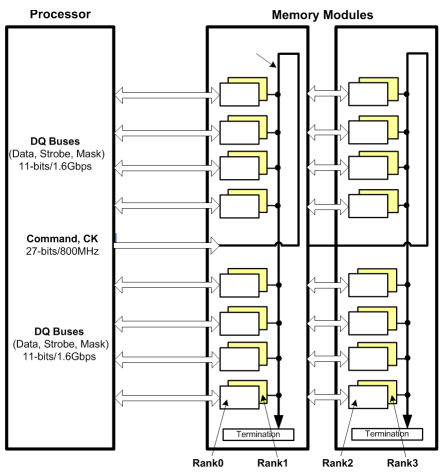


Figure 1: DDR3 DRAM Memory System.

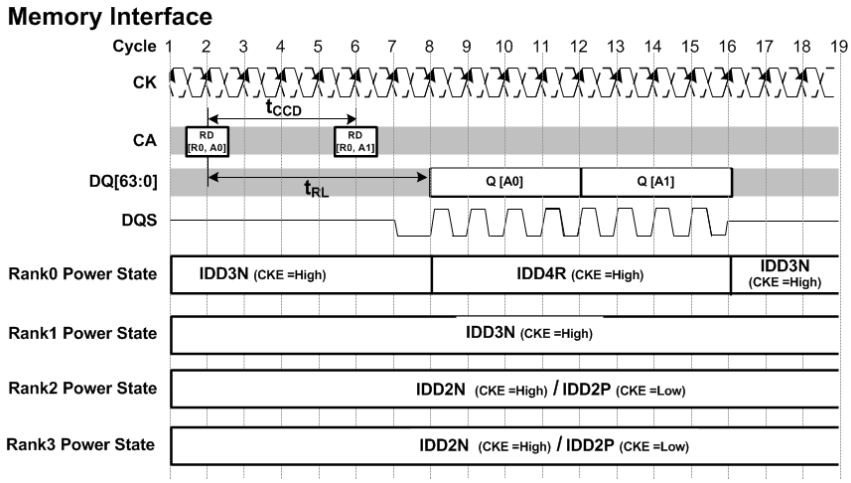


Figure 2: DDR3 DRAM Timing.

share a x64 channel. The number of channels and the interface's data rate determine system bandwidth.

Each channel is attached to a memory controller, which is integrated on the processor die. To activate a row, the controller issues a row access strobe (RAS) to enable word lines and buffers a row's data. To read and write, a column access strobe (CAS) transfers buffered data to IO interfaces. Prefetching 8 bits across a 64b wide channel produces 64B to fill a processor cache line.

2.2. DRAM Interfaces

The controller and DRAMs are connected by CA and DQ buses for control and data signals. To synchronize signals, the controller generates and forwards a clock (CK) to the DRAMs. Controller circuitry aligns this clock with command and enable signals. Because these signals have lower bandwidth and experience the same loading conditions and discontinuities en route to DRAMs, skew is not an issue. Thus, commands and writes are synchronized.

However, synchronizing reads is more difficult. During a read, data signals are generated by DRAMs (DQ) while clock signals are generated by the controller (CK). Originating on different dies, these signals are subject to different loading conditions and variations in process, voltage, and temperature. Under these conditions, the controller has difficulty using CK edges to sample DQ for arriving read data, especially at high frequencies and narrow data windows.

To facilitate read synchronization, DRAMs explicitly communicate data timing to the controller with a data strobe signal (DQS) that is aligned with the clock (CK) and bus data (DQ). The controller samples DQ on DQS edges as illustrated in Figure 2. Data is available some latency after receiving a read command (RD on CA produces Q on DQ after tRL).

DQS edges and data windows must align with the controller-generated clock. DRAMs ensure alignment in two ways. First, during initialization, DQS and CK are calibrated to eliminate any skew due to wire length while the controller specifies worst-case tolerance for timing differences (tDQSCK). Second, during operation, delay-locked loops (DLLs) dynamically adjust the DRAM clock delays to compensate for voltage and temperature variations and keep the position of the DQS at the controller constant to reduce timing uncertainty when sampling data at high frequencies.

2.3. DRAM Power Mode Limitations

Consider two scenarios in which DLLs affect efficiency. In the first, the DRAM is idling in an active power mode. In such an 'active-idle' state, the DLL and clocking power are a large fraction of the total power. For example, DDR3 active-idle current is 2× that of LPDDR2 and much of this difference is attributed to the interface [27].

Power Mode	DIMM Idle Power (W)	Exit Latency (ns)	Mechanism
Active idle	5.36	0	none
Precharge-idle	4.66	14	pages closed
Active powerdown	3.28	6	clock, I/O buffers, decode logic off
Fast exit powerdown	2.79	19.75	active powerdown + pages closed
Slow exit powerdown	1.60	24	fast exit powerdown + DLL frozen
Self Refresh	0.92	768	fast exit powerdown + DLL, CK off
Self Refresh + registers off	0.56	6700	self refresh + register PLLs off
Disabled	0	disk latency	DIMMs off

Table 1: Power Modes for a 4GB DDR3-x4-1333 RDIMM [5, 30]

In a second scenario, which we call 'idle-idle', the DRAM is in a powerdown mode. More efficient modes have higher powerup latencies (e.g., self-refresh in Table 1). While this state seems energy-efficient, the next reference pays the cost as the DRAM spends tDLLK=512 active memory cycles (768ns) powering up the interface. This is a lot of energy. In addition, applications slow down, as indicated in Figure 3(a). Thus, existing DRAM interfaces impose unattractive performance and power tradeoffs.

Static mechanisms to reduce interface power fare no better. We can configure the memory mode registers (MR) in the BIOS [30], eliminating DLLs but this imposes performance penalties. First, the peak data rate is halved as channel frequency must be lowered to ensure signal integrity. Furthermore, without DLLs, timing is less certain and controllers must assume worst-case margins (i.e., tDQSCK=10ns [30]). Conservative timing increases critical word latency, affecting application performance as shown in (Figure 3(b)).

Due to these punishing trade-offs, memory controllers invoke power modes conservatively. Modern controllers recommend a powerdown threshold no lower than 15 idle memory cycles [17]. Figure 4(a) shows the percent of time the DRAMs stay in each power state for this aggressive threshold (A), a moderate (M) threshold 10× larger, and a conservative (C) threshold 100 × larger. With such thresholds, up to 88% of memory time is in active-idle.

Potential for Efficiency. Suppose we were to address limitations in today's interfaces and power modes so that the most efficient

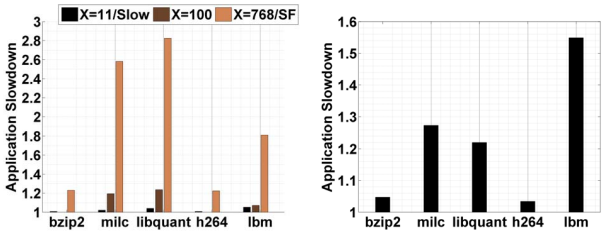


Figure 3: Performance sensitivity to (a) dynamic power down modes at different exit latencies and (b) static BIOS programming to disable DLLs.

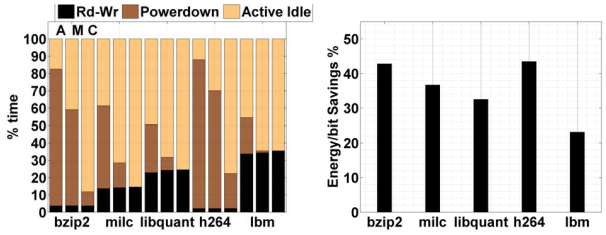


Figure 4: (a) Memory time breakdown with aggressive (A), moderate (M), and conservative (C) thresholds; (b) Potential efficiency from new power modes.

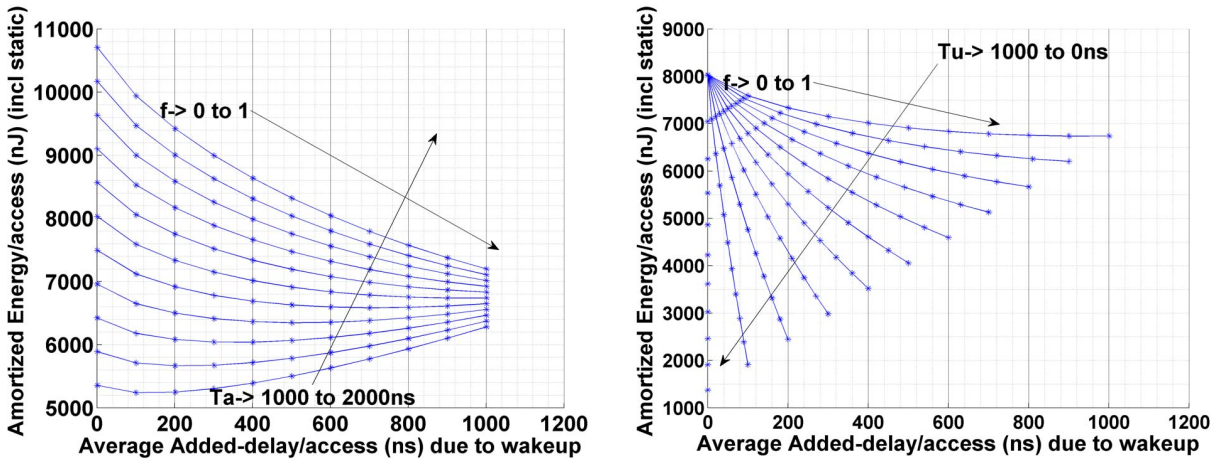


Figure 5: Probabilistic energy-delay trade-offs when powerup latency is exposed. E-D plots with different lines for different (a) memory request inter-arrival times and (b) varying powerup times. Each line's points sweep powerdown fraction f.

modes could be exploited. We would enter these modes aggressively (A) and leave them instantaneously upon the next access. Efficiency could improve by 40-50% (Figure 4(b)). Moreover, performance penalties would be negligible.

In this paper, we re-think energy-delay trade-offs with new DRAM architectures. We consider a high-performance system that requires sustained bandwidth; we cannot simply eliminate DLLs and operate at lower data rates. We present architectures that reduce power mode exit latencies and interface power by replacing DLLs with another synchronization mechanism or using existing DLLs differently.

3. Understanding Memory Activity

The precise benefits of fast exit power modes depend on the interaction between memory activity and the exit latency, which we study in two ways. First, we probabilistically model memory requests in order to understand fundamental energy-delay trends. Then, we precisely capture memory request inter-arrival times from emerging big data applications.

3.1. Probabilistic Energy-Delay Analysis

We model a stream of memory requests as a Poisson process. This analysis assumes that the time between requests follow an exponential distribution and these inter-arrival times are statistically independent, which roughly match our data. Histograms for memory inter-arrival times resemble exponential probability densities and the autocorrelation between inter-arrival times is nearly zero.

Let T_i be an exponentially distributed random variable for the idle time between two memory requests. The exponential distribution is parameterized by $1/T_a$ where T_a is the average inter-arrival time. Let P_d and P_u denote power dissipated in powerdown and powerup modes. The memory powers-down if idleness exceeds a threshold T_t. And it incurs a latency T_u when powering-up again.

Power-down is invoked with probability $f = P(T_i > T_t) = e^{-T_t/T_a}$. In this scenario, DRAM dissipates P_d for $T_i - T_t$ time while powered-down and dissipates P_u for $(T_t + T_u)$ time while powered-up. T_i is the only random variable; $\mathbb{E}[T_i] = T_a$.

$$\begin{aligned}\mathbb{E}[E] &= f\times\mathbb{E}\left[P_d(T_i - T_t) + P_uT_t + P_uT_u\right] + (1-f)\times\mathbb{E}\left[P_uT_i\right] \\ &= f\times\left[P_d(T_a - T_t) + P_uT_t + P_uT_u\right] + (1-f)\times\left[P_uT_a\right] \\ &= P_d\left[f(T_a - T_t)\right] + P_u\left[f(T_t + T_u) + (1-f)T_a\right]\end{aligned}$$

With this probabilistic formulation, the expectation for memory energy is given by $\mathbb{E}[E]$. And the expected impact on delay is $\mathbb{E}[\Delta D] = fT_u$, which conservatively assumes that powerup latency is exposed on the critical path.

Clearly, we would prefer to frequently use an efficient powerdown mode (i.e., large f and $P_d << P_u$). Energy falls as inter-arrival time increases beyond the threshold. Conversely, energy increases if powerup latency is large.

Energy-Delay Trade-offs. The relationship between $\mathbb{E}[E]$ and $\mathbb{E}[\Delta D]$ depends on average inter-arrival times (T_a), powerdown threshold (T_t), and powerup latency (T_u).

First, consider various inter-arrival times and powerdown thresholds. Each curve in Figure 5(a) plots trade-offs for a particular inter-arrival time T_a at $T_u = 1000$ns and points along a curve represent varying thresholds T_t. Short inter-arrival times ($T_a = 1000$ns) mean that the added energy costs to power back up are expensive than the savings by invoking powerdown. Thus both the energy and

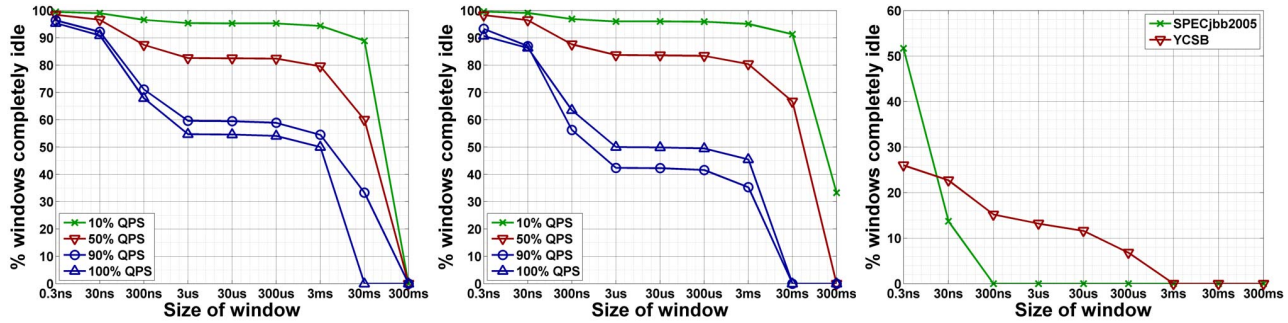


Figure 6: Activity graphs for (a-b) memcached at value sizes of 100B and 10KB. (c) SPECjbb2005 and YCSB.

delay increase with the powerdown fraction f. As inter-arrival times increase ($T_a \rightarrow$2000ns), the energy saving in powerdown offsets the overhead of wakeups.

But by implementing different thresholds for powerdown, memory controllers can explore steep and interesting trade-offs between energy and delay. Each curve in Figure 5(b) plots trade-offs for a particular powerup latency and clearly shows the cost of slow wakeups. At one end of the spectrum, zero latency powerup reduces energy with no delay penalty ($T_u = 0$ns). Waiting to go to powerdown only costs more energy, as the energies are higher for low values of f. In contrast, today's approach to disabling DLLs and clocks is expensive, producing a horizontal trend line ($T_u = 1000$ns).

Ideally, power modes reduce energy with little delay impact. This scenario would manifest as instantaneous powerup and produce vertical lines in Figure 5(b). In practice, today's efficient power modes require nearly 1000ns to powerup, producing nearly horizontal energy-delay trends in these figures. To close the gap between the ideal and practice, we re-think memory architecture to reduce powerup latency and accommodate practical inter-arrival statistics in real applications.

3.2. Characterizing Emerging Applications

The nature of computing has changed significantly in the last decade [11] and many emerging datacenter applications have memory behavior that is not well understood. While a prior study quantifies memory idleness in websearch [29], it does so for coarse, 100ms, time periods. At this granularity, which is many orders of magnitude larger than device access times (e.g., 100ns), understanding application requirements for power modes is difficult.

To study memory behavior at fine granularity, we use a custom simulation infrastructure with x86 instrumentation and a built-in scheduler [33] to benchmark a spectrum of real applications. From the spectrum of emerging data driven workloads, we characterize three representative workloads: memcached for distributed memory caching, Yahoo! Cloud Serving Benchmark (YCSB) for OLTP and data serving, and SPECjbb2005 for conventional enterprise computing.

Applications. Distributed memory caching is used by many popular websites (e.g., Facebook and Twitter) to improve query times while OLTP applications are popular in cloud computing. On the other hand, Java-based middleware servers are still popular in many enterprises.

Memcached is a popular open source distributed key-value store that caches data in DRAM for fast retrieval [32]. As the cache fills, evictions occur according to an LRU policy. Memcached hashes keys to distribute load and data. Memcached activity is a function of data popularity and query rate. We model popularity with a zipf distribution and use a large α parameter to create a long tail. We model query inter-arrival times with an exponential distribution. Such models are consistent with observed memcached queries in datacenters [34].

Yahoo! Cloud Serving Benchmark (YCSB) is a benchmark for online transaction processing that interfaces to cloud databases, such as Cassandra, BigTable, and HBase [4]. To characterize YCSB, we first populate a 6.2GB database. Next, we use the YCSB client model to generate a zipf distribution with operations that have a 95:5 read to write ratio, which is representative of modern, read-heavy applications.

SPEC Java Server Benchmark emulates a three-tier client/server system with an emphasis on the middle tier. SPECjbb performs work that is representative of business logic and object manipulation to simulate the middle tier. It exercises the Java Virtual Machine, Just-In-Time compiler, garbage collection, threads and some aspects of the OS [35].

Memory Activity. To help understand how applications will interact with low power modes, we use rank-level activity graphs to visualize memory behavior [29]. These graphs characterize bus activity using windows that define a period of time. We sweep a window over the timeline of application execution and count the number of completely idle windows for varying different window sizes. If applicable, this measurement is also taken across various application loads, which is measured in queries per second (QPS) relative to the system's peak load (denoted as %QPS).

At small value sizes (100B), memcached is CPU-bound as the CPU must cope with many small, incoming packets. At large value sizes (10KB), memcached saturates the network connection. With limited network bandwidth (e.g., 10Gb/s), memcached rarely stresses memory bandwidth (e.g., 80Gb/s).

Although memcached does not saturate memory bandwidth, we must determine whether the memory channel has uniformly low utilization or has bursty traffic with many periods of idleness. The former would make power mode design difficult but the latter would benefit from many existing DRAM power modes, and even full-system power modes.

Figure 6(a-b) shows rank-level activity graphs for memcached configured at 100B and 10KB. A large percentage of short windows (e.g., 100ns) are idle. At typical loads between 10-50%, 95% of the windows encounter completely idle memory. Moreover, these idle periods are long. Even as we increase window size towards microsecond granularities, 80-90% of these windows encounter idle memory. However, idleness is difficult to find as application load increases to 90-100% or when windows widen beyond μs granularities.

While memcached exhibits such idleness, conventional datacenter workloads in data serving and online-transaction processing have even fewer opportunities to exploit existing power-modes when run

at 100% QPS. Figure 6(c) illustrates few idle windows for SPECjbb and YCSB even at small windows. At lower utilizations that are typical in datacenters [29], the idle fractions could be higher but the opportunities are scarce beyond 1μs.

Clearly, with idle windows at the order of 1μs or less for emerging workloads, power modes that can transition in the order of 100ns are necessary. However, today's modes are insufficient to take advantage of the memory idle times in many workloads like memcached. They either have wakeup times of a few ns with consequently small energy savings, or they require nearly a μs to wakeup. Such power modes are not applicable to these applications and new modes are needed.

4. Architecting Effective Power Modes

Probabilistic analysis and workload characterization highlight the importance of fast wake-up for memory efficiency. We review high-speed interfaces to explain today's long wake-up times. Then, we propose several architectures with much shorter idle to active transitions but differ in how conservatively they enforce timing after wake-up.

4.1. High-Speed Interfaces

A reliable, high-speed interface performs two critical tasks. First, the interface converts a sequence of bits into a voltage waveform. Then, it drives that waveform on a wire with enough margin so that the receiver can distinguish between the voltages that represent ones and zeros. For high data rates, we engineer the wires as transmission lines and use termination to avoid reflections. Even so, loss in the wires and process variations cause high and low voltage levels to become distorted and mixed when they arrive at the receiver. Equalization cleans up the waveforms.

But getting the signal to the receiver is only half the battle. The other half is knowing when to sample the signal to get the correct value of the bit. At a data rate of 1.6Gb/s, the time window for each bit is only 625ps, and this time includes transitions from the previous bit and to the next bit. To build a reliable link, the interface needs to sample the bit in the middle of the stable region.

Analog circuits drive and receive the bits, and align the clock so that bits are sampled at the right time. These circuits use voltage and current references for their operation, and often use feedback to learn the right corrections (e.g. sample time or equalization) that optimize link operation.

Because they are turned off during powerdown, these circuits must re-learn their connection settings before the link can be operated again. Worse, this re-learning cannot begin until voltage and current references stabilize after powerup. Because analog circuits have lower bandwidth than digital ones, yet demand precision, μs settling latencies are typical.

One of the critical circuits in high-speed links is a delay locked loop (DLL), which uses feedback to align the phase (timing) of two clocks. In links, DLLs align the sample clock to data, or align data and strobes to the system clock. DLLs compensate for changes in timing that would otherwise occur from variations in process, voltage, and temperature (PVT). Since voltage and temperature are dynamic, DLLs continue to run after initial calibration to track and remove their effect [3, 21].

Interfaces that rapidly transition from idle to active mode apply several strategies, such as digitally storing "analog" feedback state, using simpler analog circuits that power off quickly, and designing bias networks that power off and on quickly. Applying these strategies allows DRAMs to wake-up quickly.

4.2. Fast Wake-up DRAMs

Existing link interfaces are generally symmetric: circuitry on both sides of the link need to be the same. But, symmetry is not optimal in a memory system that has a large number of DRAMs but usually a small number of controllers. Furthermore, because DRAM process technology is optimized for density, the speed of its transistors is much worse than that of transistors in a comparable logic process. Thus, we would rather shift link circuitry from DRAMs to the controller.

MemBlaze DRAMs. We present a post-DDR4 DRAM architecture with an asymmetric link interface that removes clock delay circuitry from DRAMs and places them on the memory controller. Because such circuitry determines wake-up latency in today's DRAMs, the system is capable of much faster power mode transitions. The controller and memory interfaces, which we have implemented in silicon, are shown in Figure 7.

Synchronization. In this architecture, DRAMs no longer have DLLs for timing adjustment. For arriving commands and writes, DRAMs simply sample inputs at the rising edge of link clocks, CK and DCK received from the controller. But synchronizing reads (i.e., data from DRAM to controller) requires special treatment. DRAMs no longer send data strobes along with data, which raises two new issues. The first is how the controller can learn the correct timing, and the second is that this timing may be different for each rank.

To address these challenges, the controller uses a clock and data recovery (CDR) circuit to update its clock, and thus update its sample points for data reads, based on a timing reference signal (TRS) received from DRAMs. The DRAMs time-multiplex the TRS on a pin used for error detection and correction (EDC). For every read and write, DRAMs calculate and transmit an 8-bit EDC to the controller. The remainder of the 32-bit EDC burst transmits DRAM clock information.

Thus, during normal rank operation, the EDC pin transmits correction codes interleaved with a toggling pattern that guarantees some minimum edge density. The controller tracks timing variations for each DRAM in a rank as long as that rank sees activity and communicates edges across the EDC pin. Activity on one rank provides no timing for other ranks.

Accommodating Idle Ranks. With regular accesses to a rank, the controller tracks rank timing. But gaps in activity produce gaps in phase updates. Because our interfaces rely on these updates, DRAMs specify the maximum amount between rank accesses. Ranks with longer idle periods incur a recalibration latency before further data transfers.

Alternatively, data-less pings can maintain timing when data is not needed from the memory core but toggling patterns are needed on the EDC pin for phase updates. The ping furnishes a toggling pattern without page activation or column access strobe. In this scenario, the system uses EDC signals for timing and ignores DQ signals.

Recalibration or data-less pings are small overheads that are rarely incurred. But when pings do occur, they coincide with periods of low channel utilization and thus do not interfere with normal traffic.

Fast Wake-up Protocol. Because MemBlaze DRAMs do not have DLLs, the critical latency during wake-up shifts from clock delay circuitry to the datapath. MemBlaze defines an extra control pin (DCKE) to enable the data clock domain, quickly powering the datapath, data clock buffering, and data I/O circuitry (shaded blocks in Figure 7). DCKE observes timing constraints to avoid a latency penalty.

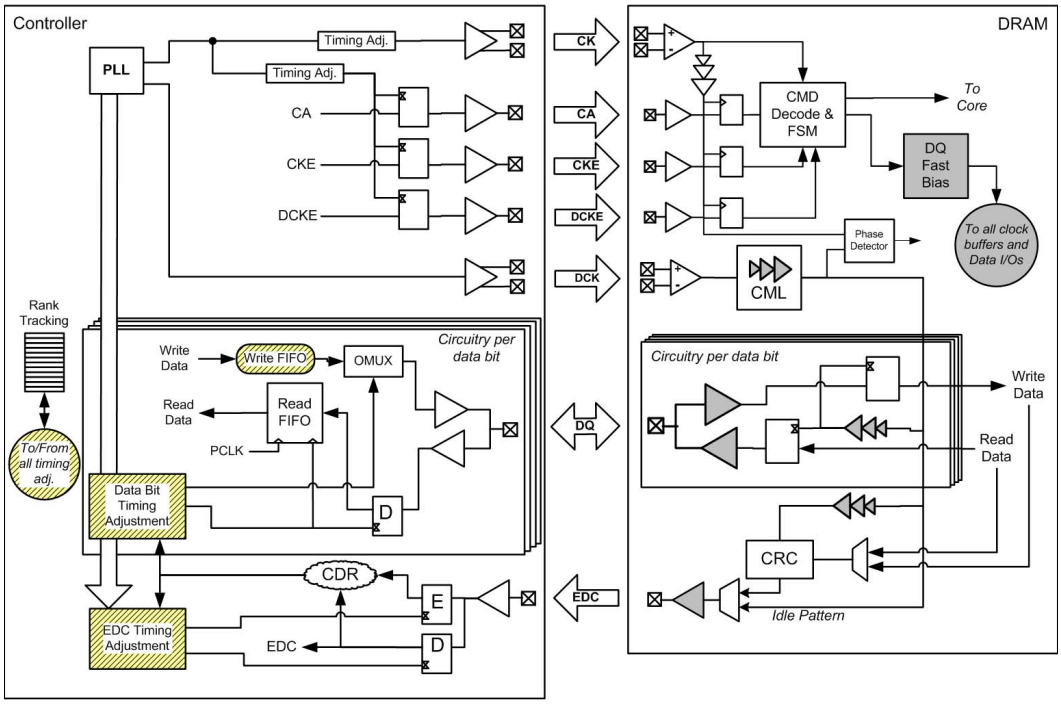


Figure 7: Proposed architecture introduces clock data recovery (CDR) circuitry to the controller, which uses the timing reference signal (TRS) transmitted across error detection and correction (EDC) pins.

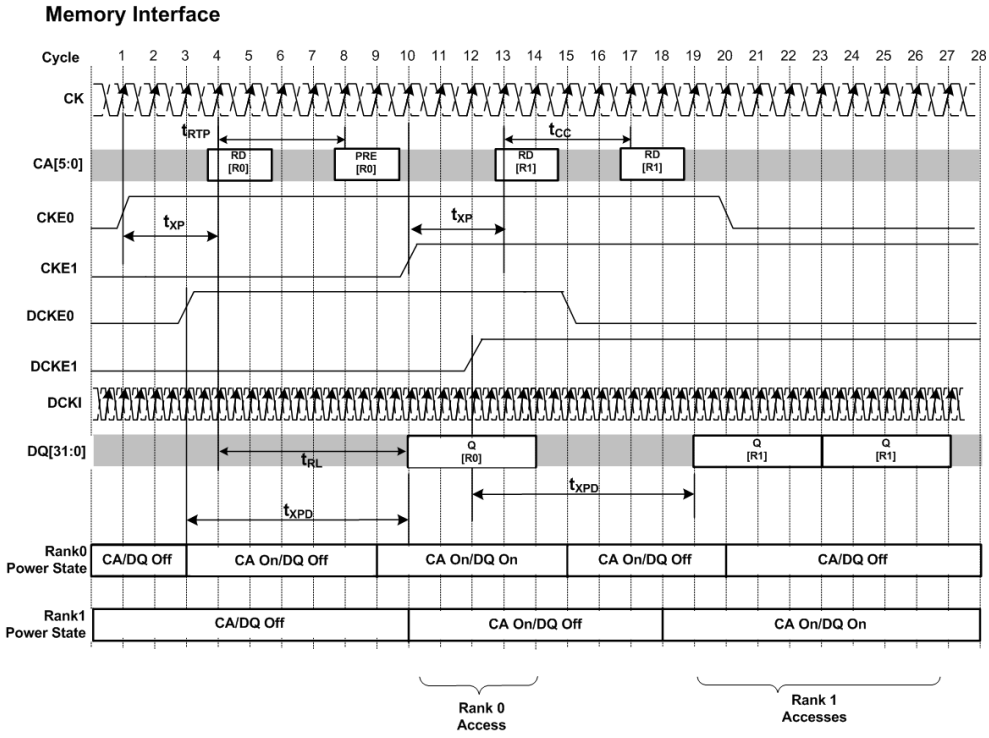


Figure 8: Timing diagram illustrates separate power management for command (CA) and data (DQ) paths.

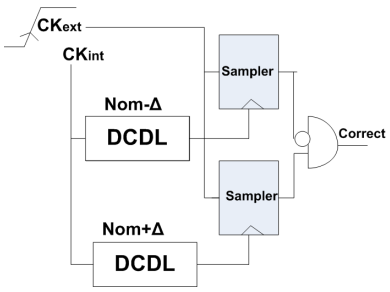


Figure 9: MemCorrect Error Detection with Digitally Controlled Delay Lines (DCDL) to sample CK_{ext} using delayed versions of nominal CK_{int}.

Figure 8 illustrates operation in a two-rank MemBlaze system. Initially, data and clock enables (DCKE, CKE) for both ranks are de-asserted, which powers-down command and data blocks. At cycle 1, CKE0 for rank 0 is asserted and the command block powers-up; the data block remains powered-down. Command receivers (CA) are awake in time to receive a read for rank 0. At cycle 3, DCKE0 is asserted and the data block powers-up to transmit read data. Exit latency for command and data blocks are tXP and tXPD.

Similarly, the second rank exits command standby at cycle 10 and exits data standby at cycle 12. Reads arrive at cycles 13 and 17, satisfying constraints on consecutive reads, which is denoted by tCC. Power-up does not affect read latency as long as DCKE is asserted early enough (i.e., tXPD before first read data). By separating command and data block enable signals, the DQ interface circuitry can remain powered-down even as precharge and refresh commands arrive.

Hiding Datapath Wake-up Latency. By default, datapath wake-up requires approximately 10ns. MemBlaze defines the DCKE pin to enable the data block early enough to avoid affecting latency and completely hide it under column access (CAS). Thus, we can quickly power the datapath only when needed and separate the powerups for command and data blocks. For fast DRAM interface wake-up, MemBlaze exploits a number of circuit innovations including common-mode logic (CML) clock trees and fast-bias circuitry to powerup links quickly. Further, if we leverage more insights from a recently implemented serial link interface that transitions from idle to active well under 10ns [40], we could simply use read or write commands to trigger datapath powerup eliminating the need for DCKE.

Timing, Datarate and Power. Both the MemBlaze Memory Controller and DRAM PHYs were taped out in a 28nm process and the chip was rigorously tested for functionality, correctness and the proposed fast wakeup speed in an industry-strength, serial-links laboratory[19]. In lieu of a DRAM core, the test chip pairs the new PHY blocks with test pattern generation and checking logic for emulating memory read and write transactions. The laboratory operation of the timing is demonstrated in Appendix §A.

The transmit eye diagram at the DQ pins had clear eyes with sufficient timing and voltage margins at 6.4Gbps. The architecture also reduces power in both active-standby and precharge-standby power modes. Compared to today's power modes, this provides the performance of fast-exit powerdown with the DLL-off efficiency.

Specifically, this matches deep power down mode's power at a reduced exit latency of 10ns making it useful for many different applications including emerging ones that have short idle periods. The power difference between the active idle mode and the most efficient powerdown mode is attributed to DLLs, clock tree, and

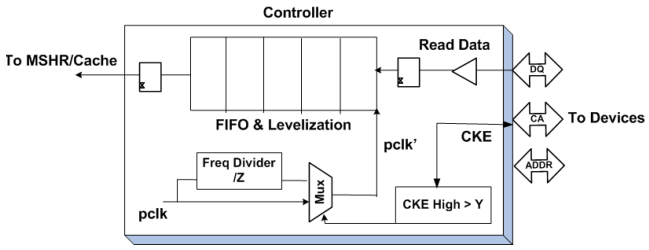


Figure 10: MemDrowsy Architecture.

command pins. By powering down these components, MemBlaze lowers power in all other states except during active reads/writes.

Although standby efficiency improves, dynamic burst power is largely unchanged. During bursts, DLL power savings are offset by new power costs in current-mode clock circuitry and injection locked-oscillator power costs.

4.3. Imperfect Wake-up Timing

MemBlaze provides an ideal solution and perfect timing upon power-mode wake-up. Alternatively, we propose two new mechanisms in which DRAMs retain their DLLs and turn them on/off very aggressively. Systems that would prefer not to modify the device interface (as MemBlaze does) and can tolerate modest performance degradation will benefit from such techniques. But such DRAMs face synchronization challenges, and we propose mechanisms to mitigate or avoid timing errors. Without changing the controller-device link interface and operating speed these systems retain the DDRx memory links. However, they greatly reduce DRAM idle power by simply changing the sampling rate/decision inside the controller using a few flops.

Reactive Memory Interfaces (MemCorrect). For interfaces that track timing less precisely, we introduce speculative DRAM data transfers immediately after waking up from deep powerdown modes. To guarantee that each memory transaction completes correctly, we architect error detectors. Our fast wake-up memory interface implements this error check on each transaction (Figure 9), which ensures that the clock transition is within a window ($\pm\Delta$) of its nominal location. This check handles cases when variations and drift during powerdown affect link operation. The error is communicated to the controller through a dedicated pin, *Correct*. If an error occurs, it is because the controller has issued a command too soon after powerup. The controller could simply wait a longer period of time before retrying the command or could send a timing calibrate command to expedite wake-up.

Errors are unlikely in systems with modest voltage and temperature variations. Most memory systems fit this description since boards are designed for tolerance against voltage fluctuations and temperatures vary slowly. Moreover, these variations have a modest effect on timing margins if ranks powerdown for short periods as drift is less likely to have accumulated to affect timing margins.

Drowsy Memory Interfaces (MemDrowsy). Rather than wait for calibration, a controller might begin transfers immediately but mitigate timing errors by halving the data rate for a certain period of time (Y) after wake-up. This slower rate more than doubles the timing margin of the link, greatly improving tolerance to small timing errors induced by VT variations.

Thus, MemDrowsy reduces the effective data rate and relaxes timing precision after wake-up, for reads that need a locked DLL. The

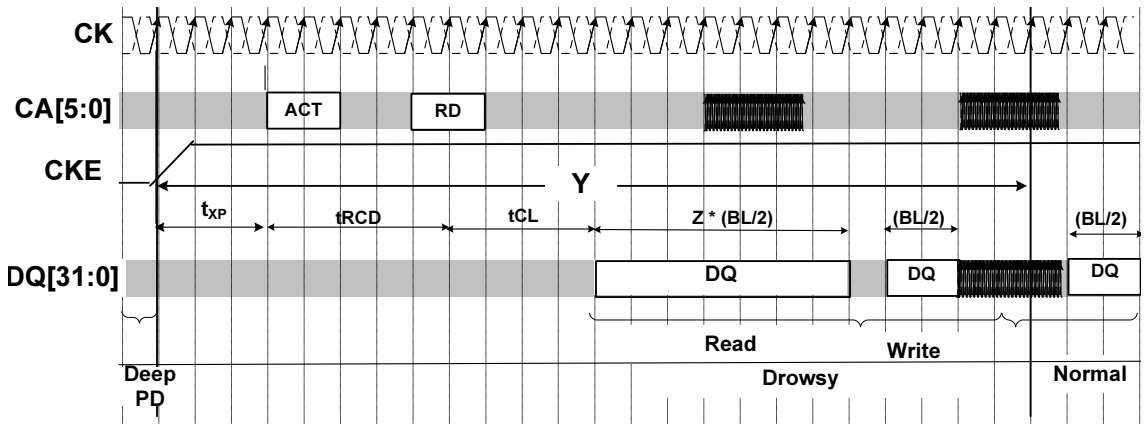


Figure 11: MemDrowsy Timing Diagram.

clock speed is still maintained at the full rate but the link effectively transmits each bit twice. This enables transmitting data while recalibrating and results in lengthening the valid data window. Of course, the controller must also shift the point at which data is sampled. After recalibration, the link operates at nominal data rates.

Figure 10 illustrates extensions to the memory controller. A rank is powered-down simply by disabling the clock (CKE low). Upon powerup, the clock is enabled (CKE high) and a timer starts. The clock operates at the nominal frequency f, providing a sufficient density of clock edges needed to facilitate timing feedback and recovery.

However, given timing uncertainty after wake-up, we use a frequency divider to reduce the rate at which we sample the incoming data; the drowsy rate is f/Z. A multiplexer chooses between sampling at the nominal clock rate or at a divided rate. During drowsy mode, the valid read window is lengthened by a factor of Z as illustrated in Figure 11.

Since the drowsy sampling period is an integral multiple of nominal sampling period, the controller clock is unchanged; it is simply sampled every Z-th cycle.[1] Sampling returns to the nominal frequency only after timing recalibration.

5. Evaluation

We evaluate system implications from two types of memory architectures. The first type, MemBlaze, provides perfect timing and synchronization after wake-up by eliminating expensive interface circuitry from the DRAMs. The second type, exemplified by MemCorrect and MemDrowsy, provides imperfect timing and requires corrective mechanisms.

MemBlaze provides the efficiency of powerdown without performance trade-offs. But MemCorrect and MemDrowsy's mechanisms to ensure correct timing can negatively affect performance. We quantify these effects.

5.1. Experimental Methodology

Simulators. We use an x86_64 execution-driven processor simulator based on a Pin front-end [26, 33]. We use eight out-of-order (OOO) cores at 3GHz matched with Intel's Nehalem microarchitecture and cache latencies as shown in Table 2.

The memory simulator is extended to model three architectures: MemBlaze, MemCorrect, and MemDrowsy. MemBlaze is implemented in silicon and chip measurements are used to configure the simulator. For the other memory architectures, the simulator draws timing estimates from JEDEC specifications and energy estimates from Intel's analyses [5, 16]. In general, DDR3 systems dissipate about 1-1.5W/GB on average [30] and about 2.5W/GB at peak [12]. We validate that our experiments produce numbers in this range. Other memory simulator parameters are described in Table 2.

Processor	Eight 3GHz x86 Out-of-Order cores
L1 cache	private, 8-way 32KB, cycle time = 1, 64B cache-lines
L2 cache	private, 8-way 256KB, cycle time = 7, 64B cache-lines
L3 cache	shared, 16-way 16MB, cycle time = 27, 64B cache-lines
Memory controller	Fast powerdown with threshold timer = 15 mem-cycles [17] Closed-page, FCFS scheduling
Main memory	32GB capacity, 2Gb x4 1333MT/s parts, single ranked 4GB-RDIMMs, four channels, 2DIMMs/channel [5, 16]

Table 2: Baseline System Simulation Parameters.

Classification	**Multi-Programmed (MP) Benchmarks**
High B/W (MP-HB)	433.milc, 436.cactusADM, 450.soplex, 459.GemsFDTD, 462.libquantum, 470.lbm, 471.omnetpp, 482.sphinx3
Med. B/W (MP-MB)	401.bzip2, 403.gcc, 434.zeusmp, 454.calculix, 464.h264ref 473.astar
Low B/W (MP-LB)	435.gromacs, 444.namd, 445.gobmk, 447.dealII, 456.hmmer, 458.sjeng, 465.tonto
Classification	**Multi-Threaded (MT) Benchmarks**
High B/W (MT-HB)	applu, art, canneal, streamcluster, swim, mgrid
Med. B/W (MT-MB)	apsi, blackscholes, equake
Low B/W (MT-LB)	ammp, fluidanimate, wupwise

Table 3: Benchmark Classification.

Workloads. We evaluate memory activity and the proposed architectures on datacenter workloads like memcached. [2] During evaluation, we fast-forward initialization phases and perform accurate simulations during the measurement phase by running for fixed number of instructions across power modes.

In addition, we evaluate a variety of multi-programmed (MP) SPEC CPU2006 as well as multi-threaded (MT) SPEC OMP2001 and PARSEC benchmarks, following prior memory studies [2, 37, 22, 7, 18]. Each core runs a copy of the program/thread depending on the benchmark and the number of application threads or processes are matched to the cores. We fast-forward 10 to 20 billion instructions to skip warm-up and initialization stages and focus on memory behavior in steady state for weighted IPC calculations. We classify MP and MT applications into 3 groups (HB, MB, LB) as shown in Table 3.

Metrics. For each memory architecture, we plot efficiency and performance. Efficiency is measured in energy per bit (mW/Gbps). In this metric, static and background power are amortized over useful data transfers. Performance penalties measure the impact on cycles

[1] MemDrowsy clock rates are unchanged, differentiating it from work in channel frequency scaling [5].

[2] 100b value denoted by *a* and 10KB by *b*

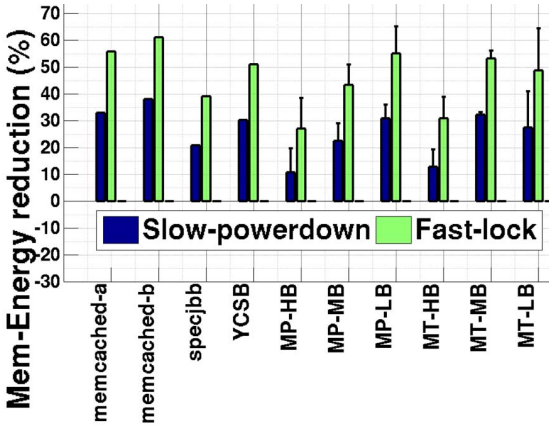


Figure 12: MemBlaze (fast-lock) energy savings relative to DDR3 DRAM baseline (fast-powerdown) and compared against DDR3 DRAM baseline (slow-powerdown).

per instruction (CPI). In each workload group, worst case performance penalty and best case energy savings are plotted on the top of each bar.

Energy savings are measured relative to baseline DDR3 DRAM that aggressively exploits fast-powerdown whenever encountering 15 idle memory cycles [17]. This low threshold gives an optimistic baseline. Realistic, high-performance systems would set the threshold an order of magnitude higher, which would only magnify both active-idle energy costs and our architectures' advantages.

5.2. MemBlaze

MemBlaze efficiency arises from two key features. First, it eliminates DRAMs' DLLs and clocks, thus eliminating long-latency DLL recalibration, which is on the critical path for today's mode exits. Capable of fast exits, MemBlaze can spend more time in powerdown and less time in active-idle.

Second, for any remaining time spent in active idle (i.e., neither bursting data nor in powerdown), MemBlaze consumes very little energy. With MemBlaze links that are capable of fast wake-up, DRAMs' data blocks are powered-on by DCKE precisely when they are needed and no earlier. Only the command blocks remain active, consuming a small fraction of the original active-idle power.

Given these advantages, MemBlaze energy savings are substantial. Even though silicon results indicated feasibility at much larger datarates, we conservatively use DDR3-1333 transfer rate in our simulations to make the comparison fair. Figure 12 compares savings from MemBlaze (fast-lock) and the most efficient power-mode in today's DRAMs (slow-powerdown). When DLLs in today's DRAMs are kept in a quiescent state, slow-power down improves efficiency by 22%. But this efficiency requires a performance trade-off. Exit latency is 24ns, which affects the critical word latency.

MemBlaze fast-lock improves efficiency by 43%. Even memory-intensive applications, like *433.milc* and *471.omnetpp*, dissipate 25-36% less energy. Applications that demand little bandwidth (LB) like *444.namd* consume 63% less energy. With compared against a baseline that uses power-modes more conservatively, these savings would increase by 2×.

Moreover, efficiency comes with better performance than the baseline since MemBlaze power mode exit latency is comparable to that of fast-powerdown in today's DRAMs; neither incur DLL-related wake-up latencies. Fast links powerup the datapath in 10ns. Because this latency is hidden by the command access, we reduce energy with no performance impact.

With attractive energy savings and no delay trade-off, MemBlaze is an order of magnitude better than approaches that aggressively power-off DLLs at run-time or modify the BIOS to disable DLLs at boot-time. These mechanisms all require large performance trade-offs since today's high-performance DRAMs rely on DLLs for timing.

5.3. MemCorrect

While MemBlaze provides perfect timing, other interfaces (including today's DRAMs with DLLs) may be susceptible to timing errors when aggressively exploiting power modes. MemCorrect provides circuitry to detect timing errors, allowing the system to speculate that the timing was correct. We assess performance and energy relative to the DDR3 DRAM baseline in Figure 13.

We evaluate MemCorrect based on the probability p of correct timing. In the best-case, p=100% and timing is never affected when using power-modes. And p=0% is the worst-case in which every wake-up requires a long-latency recalibration. Smaller values for p degrade performance. When p=50%, performance degrades by as much as 100%.

In practice, systems are more likely to encounter correct timing. Boards can be designed with decoupling capacitors to tolerate voltage fluctuations and thermal effects have long time constants. If timing is correct for 99% or 90% of transfers immediately following a wake-up, we incur modest performance penalties of 1% and 10%, respectively.

In exchange for the occasional delay, MemCorrect can exploit power-modes more aggressively. DRAMs with DLLs might bypass recalibration, start transfers immediately after wake-up, and detect errors as they occur. In such a system, MemCorrect energy savings are 38% and 30% when timing is correct for p =99% and p =90% of the transfers. However, if errors are too common, workloads encounter large penalties and low, or even negative, energy savings.

To increase the likelihood of correct timing, we might characterize phase sensitivity to temperature (T) and voltage (V) while the part is operating. During powerdown, we could store the current phase, voltage, and temperature. And changes in T and V during powerdown could be used to calculate a small correction to the last phase. Upon powerup, this correction is added to the phase. We draw lessons from processors in which canaries predict critical path delay across PVT corners and find frequencies that meet timing constraints [9].

5.4. MemDrowsy

If correct timing cannot be ensured at nominal data sampling rates, the system could operate in drowsy mode and reduce its sampling rate by a factor of Z for Y=768ns (tDLLk). In practice Z depends on timing margins at the DRAM interface. $Z = 2$ is realistic because existing LPDDR2 systems eliminate DLLs and transfer at half the data rate to ensure timing. We also assess sensitivity to more conservative margins ($Z = 4$, $Z = 8$).

Reducing the data sample rate more aggressively produces larger penalties in Figure 14(a); latency-sensitive *streamcluster* sees a 32% penalty when $Z = 8$. Less drowsy transfers have far more modest penalties, ranging from 1-4%.

MemDrowsy is also parameterized by how long the DRAM must operate in drowsy mode. In practice, this parameter is defined by the nominal wake-up latency. In other words, transfers are drowsy until the interface can ensure correct timing (e.g., current DRAM DLLs require 768ns). Performance is insensitive to the duration of drowsy operation as only the first few transfers after wake-up are slowed.

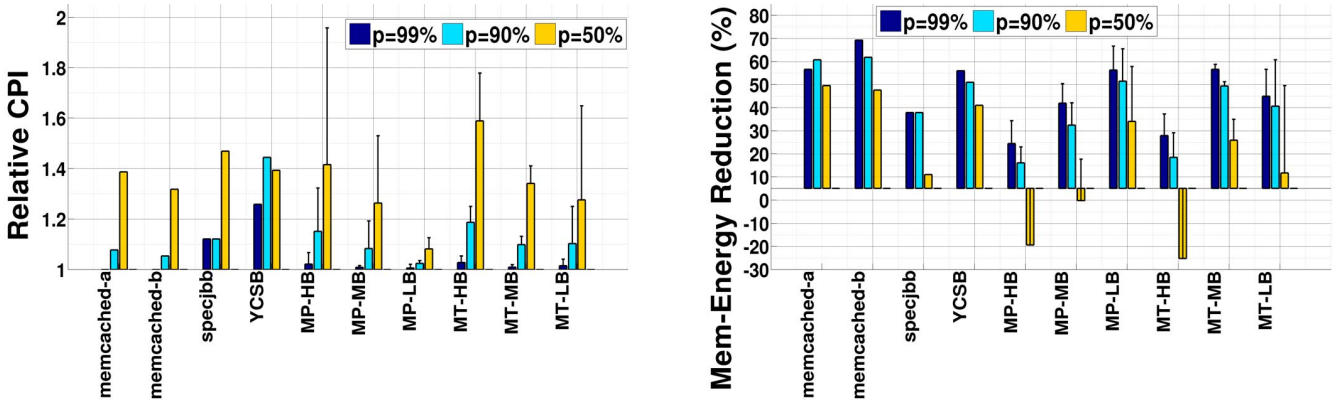


Figure 13: MemCorrect (a) performance as measured in Cycles per Instruction (CPI) and (b) energy relative to DDR3 DRAM baseline. The probability p of correct timing for transfer immediately after wakeup is varied. Plotted for MP, MT, datacenter benchmarks. Error bars represent ranges over mean value in the group.

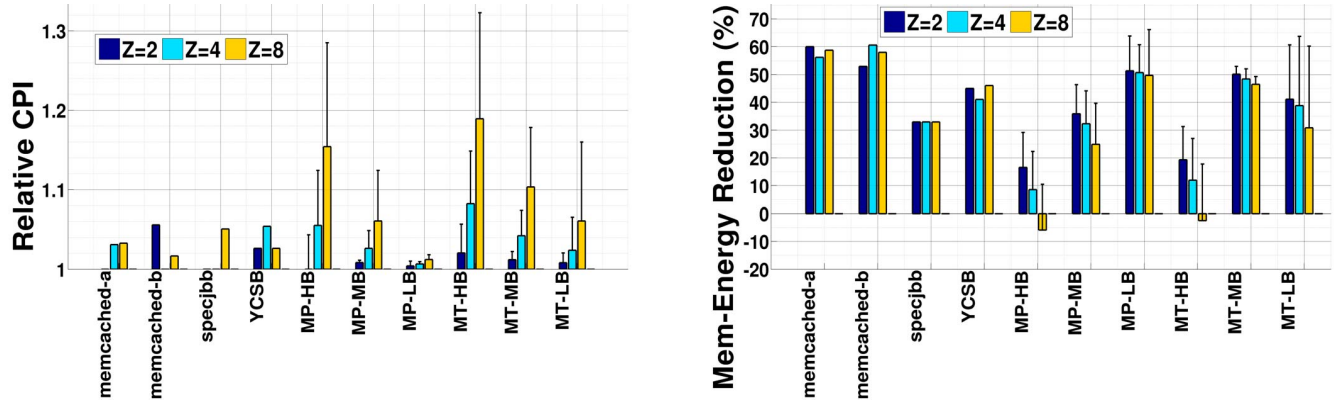


Figure 14: MemDrowsy (a) performance as measured in Cycles per Instruction (CPI) and (b) energy relative to DDR3 DRAM baseline. The drowsy rate reduction factor Z for transfer is varied by using Y=768ns. Plotted for MP, MT, datacenter benchmarks. Error bars represent ranges over mean value in the group.

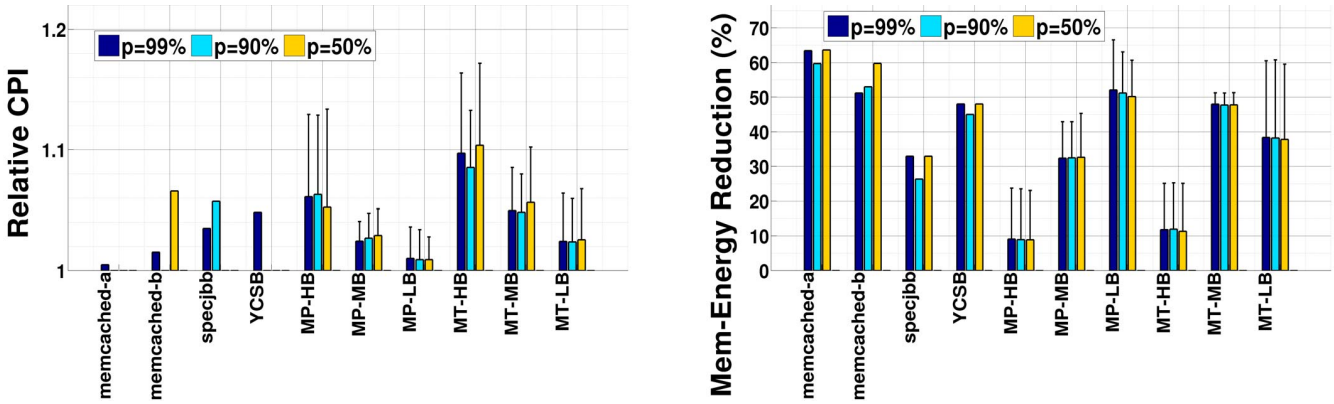


Figure 15: MemCorrect+MemDrowsy (a) performance as measured in Cycles per Instruction (CPI) and (b) energy relative to DDR3 DRAM baseline. The probability p of correct timing for transfer immediately after wakeup with $Z = 4$ is varied. Plotted for MP, MT, datacenter benchmarks. Error bars represent ranges over mean value in the group.

For these modest penalties, MemDrowsy achieves significant energy savings in Figure 14(b). Drowsy transfers allow applications to enter power modes more often with fewer penalties. Clearly, applications that demand memory bandwidth (HB) are more sensitive to drowsy operation. Indeed, average energy per transfer might increase due to larger termination energy from higher bus utilization and also the idle power during the extra cycles.

5.5. MemCorrect and MemDrowsy

Suppose MemCorrect detects a timing error for a transfer immediately following a wake-up. Instead of delaying the transfer for the nominal wake-up latency, the system invokes MemDrowsy and begins the transfer immediately at a slower rate. Clearly, performance and efficiency in MemCorrect+MemDrowsy will be better than either approach applied individually. Immediately after wake-up, transfers begin immediately either at the nominal or reduced data rate.

With MemCorrect+MemDrowsy, exploiting power modes and transferring data immediately after wake-up has performance penalties between 10-20%, as shown in Figure 15(a). In exchange, power modes are more often exploited and energy savings are more consistent.

When implemented alone, MemCorrect energy savings are very sensitive to the probability of correct timing after wake-up. In combination, however, MemCorrect+MemDrowsy is insensitive to timing risk, as shown in Figure 15(b). As MemDrowsy improves memory channel utilization, background power is amortized over more transfers.

MemBlaze promises large energy savings with an architecture that provides perfect timing information. Without such timing guarantees, however, MemCorrect+MemDrowsy provide the next best thing: comparable efficiency and modest ($<$10%) performance degradation for many applications.

6. Related Work

Much prior work reduces power in conventional server memory. Memory systems can statically set voltage and frequency at boot time, typically in the BIOS [7]. Frequency scaling reduces power but since the static power is amortized over few accesses at low utilizations, the energy per memory access is still expensive. The energy per access could also increase due to higher bus utilization from scaling [5].

Malladi et al. studied LPDDR2 in servers to trade bandwidth for reduced active-idle power [27]. In the current paper, we convert DDR3 active-idle time to time in efficient powerdown without affecting bandwidth. Lim et al. consider various grades of DDR memory [25, 24] while Kgil et al. consider memory-processor stacking [20]. In contrast, we propose changes to power-hungry DRAM interfaces.

Prior work also manages DRAM data placement, increasing access locality and creating opportunities to transition between power states [10, 23, 36, 8, 31, 1]. Prior work studies compiler strategies for cluster accesses by inserting NOPs or reordering to coalesce requests [6]. Also, pages might be redirected to particular DRAM ranks to create hot and cold memory spaces [14]. Memory controllers can throttle requests to manage power [15]. In contrast, our work improves powermode efficiency as multiple studies highlight increasing difficulty of finding usable idle times [7, 29].

We build upon detailed studies of Meisner et al. about subsystem characterization [29], and Ferdman et al. insights on scale out workloads [11]. We study memory bus activity at finer granularities, enabling the analysis and design of DRAM power modes.

Given wide accesses internal to DRAM, chips have been proposed to reduce the number of parts activated. One approach reduces access granularity through separately controlled parts (e.g.,chips, ranks, banks, etc.) to create smaller, independent memory spaces [2, 37, 39, 41]. However, reducing the size of the DRAM activated increases the number of peripheral circuits and degrades density [38].

7. Conclusion

In server memory, idle power can comprise 60-70% of the total. Memory ranks spend 45-60% of their time idle. The spectrum of memory architectures presented in this paper demonstrate new interfaces and architectures that address this problem. By eliminating or mitigating long-latency DLL wake-ups, these systems aggressively uses efficient powerdown states during short idle periods with negligible performance penalty.

Benefits are particularly pronounced for high-capacity, multi-rank systems with frequent idleness. We demonstrate energy savings of up to 68% in a four-rank memory system. While MemBlaze reduces idle power with no performance impact on the system, MemDrowsy and MemCorrect accomplish similar power savings with low penalties. A MemBlaze test chip has also been fabricated and demonstrated to function at a high datarate of 6.4Gbps while the exit latencies and idle power are verified with hardware measurements. Overall, we demonstrate possible techniques to build scalable, energy-proportional memory systems for the future.

8. Acknowledgments

We sincerely thank Yi Lu for helping us with lab measurements, James Tringali, Hongzhong Zheng, Jared Zerbe for their useful discussion. This project is supported in part by a Google Focused Research Award. Krishna Malladi is supported by Benchmark-Capital Stanford Graduate Fellowship. Benjamin Lee is supported in part by NSF grant CCF-1149252.

A. Laboratory Measurements

The fabricated test chip has been tested extensively and Figure 16 demonstrates the timing operation of the powermodes.

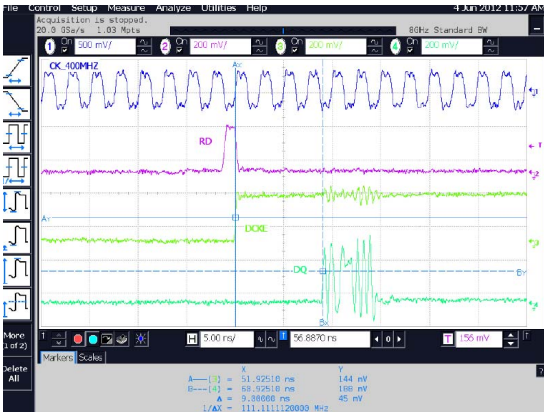


Figure 16: Timing operation for the MemBlaze test chip's fast lock link demonstrating DQ operation within 10ns from powermode wakeup.

References

[1] N. Aggarwal et al. Power-energycient DRAM speculation. In *High Performance Computer Architecture*, 2008.

[2] J. H. Ahn, N. P. Jouppi, C. Kozyrakis, J. Leverich, and R. S. Schreiber. Future scaling of processor-memory interfaces. In *SC*, 2009.

[3] Chih-Kong and K. Yang. Delay-locked loops - an overview. In *Phase Locking in High-Performance Systems. IEEE Press*, 2003.

[4] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *ACM Symposium on Cloud Computing*, 2010.

[5] H. David, O. Mutlu, et al. Memory power management via dynamic voltage/frequency scaling. In *ICAC*, 2011.

[6] V. Delaluz, M. Kandemir, N. Vijaykrishnan, A. Sivasubramaniam, and M. Irwin. DRAM energy management using software & hardware directed power mode control. In *High Performance Computer Architecture*, 2001.

[7] Q. Deng, D. Meisner, L. Ramos, T. F. Wenisch, and R. Bianchini. Memscale: Active low-power modes for main memory. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2011.

[8] B. Diniz, D. Guedes, and R. Bianchini. Limiting the power consumption of main memory. In *International Symposium on Computer Architecture*, 2007.

[9] D. Ernst, N. S. Kim, S. Das, S. Lee, D. Blaauw, T. Austin, T. Mudge, and K. Flautner. Razor: A low-power pipeline based on circuit-level timing speculation. In *International Symposium on Microarchitecture*, 2003.

[10] X. Fan, C. Ellis, and A. Lebeck. Memory controller policies for DRAM power management. In *International Symposium on Low Power Electronics and Design*, 2001.

[11] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi. Clearing the clouds: a study of emerging scale-out workloads on modern hardware. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2012.

[12] Hewlett-Packard. DDR3 memory technology. Technology brief TC100202TB, Hewlett-Packard, 2010.

[13] U. Hoelzle and L. Barroso. *The Datacenter as a Computer*. Morgan and Claypool, 2009.

[14] H. Huang, K. G. Shin, C. Lefurgy, and T. Keller. Improving energy efficiency by making DRAM less randomly accessed. In *International Symposium on Low Power Electronics and Design*, 2005.

[15] I. Hur and C. Lin. A comprehensive approach to DRAM power management. In *High Performance Computer Architecture*, 2008.

[16] Intel. Intel memory 3-sigma power analysis methodology. Data sheet, Intel.

[17] Intel. Intel xeon processor e3-1200 family datasheet. Data sheet, Intel, 2011.

[18] A. Jaleel, K. B. Theobald, S. C. S. Jr, and J. Emer. High performance cache replacement using re-reference interval prediction (RRRIP). In *International Symposium on Computer Architecture*, 2010.

[19] K. Kaviani et al. A 6.4-Gb/s near-ground single-ended transceiver for dual-rank dimm memory interface systems. In *International Solid-State Circuits Conference*, February 2013.

[20] T. Kgil et al. PicoServer: Using 3D stacking technology to enable a compact energy efficient chip multiprocessor. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2006.

[21] J. Kim, M. Horowitz, and G.-Y. Wei. Design of cmos adaptive-bandwidth plls/dlls: A general approach. In *IEEE Transactions on Circuits and Systems-II: Analog and Digital Signal Processing*, 2003.

[22] Y. Kim, M. Papamichael, O. Mutlu, and M. Harchol-Balter. Thread cluster memory scheduling: Exploiting differences in memory access behavior. In *International Symposium on Microarchitecture*, 2010.

[23] A. Lebeck, X. Fan, H. Zeng, , and C. Ellis. Power aware page allocation. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2000.

[24] K. Lim, J. Chang, T. Mudge, P. Ranganathan, S. K. Reinhardt, and T. F. Wenisch. Disaggregated memory for expansion and sharing in blade servers. In *International Symposium on Computer Architecture*, 2009.

[25] K. Lim, P. Ranganathan, J. Chang, C. Patel, T. Mudge, and S. Reinhardt. Understanding and designing new server architectures for emerging warehouse-computing environments. In *International Symposium on Computer Architecture*, 2008.

[26] C. Luk et al. Pin: Building customized program analysis tools with dynamic instrumentation. In *PLDI*, 2005.

[27] K. Malladi, F. Nothaft, K. Periyathambi, B. Lee, C. Kozyrakis, and M. Horowitz. Towards energy-proportional datacenter memory with mobile DRAM. In *International Symposium on Computer Architecture*, 2012.

[28] D. Meisner, B. Gold, and T. Wensich. PowerNap: Eliminating server idle power. In *International Symposium on Computer Architecture*, 2009.

[29] D. Meisner, C. M. Sadler, L. A. Barroso, W.-D. Weber, and T. F. Wenisch. Power management of online data-intensive services. In *International Symposium on Computer Architecture*, 2011.

[30] Micron. Micron 2Gb: x4, x8, x16 DDR3 SDRAM. Data Sheet MT41J128M16HA-125, Micron, 2010.

[31] V. Pandey, W. Jiang, Y. Zhou, and R. Bianchini. DMA-aware memory energy management. In *High Performance Computer Architecture*, 2006.

[32] P. Saab. Scaling memcached at Facebook. *Facebook Engineering Note*, 2008.

[33] D. Sanchez et al. The ZCache: Decoupling ways and associativity. In *International Symposium on Microarchitecture*, 2011.

[34] N. Sharma, S. Barker, D. Irwin, and P. Shenoy. Blink: Managing server clusters on intermittent power. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2011.

[35] K. Shiv et al. SPECjvm2008 performance characterization. In *SPEC Benchmark Workshop on Computer Performance Evaluation and Benchmarking*, 2009.

[36] M. Tolentino et al. Memory MISER: Improving main memory energy efficiency in servers. *IEEE Trans*, 2009.

[37] A. N. Udipi, N. Muralimanohar, N. Chatterjee, R. Balasubramonian, A. Davis, and N. P. Jouppi. Rethinking DRAM design and organization for energy-constrained multi-cores. In *International Symposium on Computer Architecture*, 2010.

[38] T. Vogelsang. Understanding the energy consumption of dynamic random access memories. In *International Symposium on Microarchitecture*, 2010.

[39] F. Ware and C. Hampel. Improving power and data efficiency with threaded memory modules. In *International Conference on Computer Design*, 2006.

[40] J. Zerbe, B. Daly, L. Luo, B. Stonecypher, W. D. Dettloff, J. C. Eble, T. Stone, J. Ren, B. S. Leibowitz, M. Bucher, P. Satarzadeh, Q. Lin, Y. Lu, and R. Kollipara. A 5gb/s link with matched source synchronous and common-mode clocking techniques. In *IEEE Journal of Solid-State Circuits*, 2011.

[41] H. Zheng, J. Lin, Z. Zhang, E. Gorbatov, H. David, and Z. Zhu. Mini-rank: Adaptive DRAM architecture for improving memory power efficiency. In *International Symposium on Microarchitecture*, 2008.

2012 IEEE/ACM 45th Annual International Symposium on Microarchitecture

CoScale: Coordinating CPU and Memory System DVFS in Server Systems

Qingyuan Deng David Meisner[†] Abhishek Bhattacharjee
Thomas F. Wenisch[‡] Ricardo Bianchini

Rutgers University [†]Facebook Inc. [‡]University of Michigan
{qdeng,abhib,ricardob}@cs.rutgers.edu meisner@fb.com twenisch@umich.edu

Abstract

Recent work has introduced memory system dynamic voltage and frequency scaling (DVFS), and has suggested that balanced scaling of both CPU and the memory system is the most promising approach for conserving energy in server systems. In this paper, we first demonstrate that CPU and memory system DVFS often conflict when performed independently by separate controllers. In response, we propose CoScale, the first method for effectively coordinating these mechanisms under performance constraints. CoScale relies on execution profiling of each core via (existing and new) performance counters, and models of core and memory performance and power consumption. CoScale explores the set of possible frequency settings in such a way that it efficiently minimizes the full-system energy consumption within the performance bound. Our results demonstrate that, by effectively coordinating CPU and memory power management, CoScale conserves a significant amount of system energy compared to existing approaches, while consistently remaining within the prescribed performance bounds. The results also show that CoScale conserves almost as much system energy as an offline, idealized approach.

1. Introduction

The processor has historically consumed the bulk of system power in servers, leading to a rich array of processor power management techniques, e.g. [16, 20, 37]. However, due to their success, and because of increasing memory capacity and bandwidth requirements in multicore servers, main memory energy consumption is increasing as a fraction of the total server energy [2, 24, 29, 39]. In response, many active and idle power management techniques have been proposed for main memory as well, e.g. [8, 10, 11, 12, 22, 34]. In light of these trends, servers are likely to provide separate power management capabilities for individual system components, with distinct control policies and actuation mechanisms. Our ability to maximize energy efficiency will hinge on the coordinated use of these various capabilities [31].

Prior work on the coordination of CPU power and thermal management across servers, blades, and racks has demonstrated the difficulty of coordinated management and the potential pitfalls of independent control [36]. Existing studies seeking to coordinate CPU DVFS and memory low-power modes have focused on *idle* low-power memory states [6, 13, 27]. While effective, these works ignore the possibility of using DVFS for the memory subsystem, which has recently been shown to provide greater energy savings [10]. As such, the coordination of *active* low-power modes for processors and memory in tandem remains an open problem.

In this paper, we propose CoScale, the first method for effectively coordinating CPU and memory subsystem DVFS under performance constraints. As we show, simply supporting separate processor and memory energy management techniques is insufficient, as independent control policies often conflict, leading to oscillations, unstable behavior, or sub-optimal power/performance trade-offs.

To see an example of such behavior, consider a scenario in which a chip multiprocessor's cores are stalled waiting for memory a significant fraction of the time. In this situation, the CPU power manager might predict that lowering voltage/frequency will improve energy efficiency while still keeping performance within a pre-selected performance degradation bound and effect the change. The lower core frequency would reduce traffic to the memory subsystem, which in turn could cause its (independent) power manager to lower the memory frequency. After this latter frequency change, the performance of the server as a whole may dip below the CPU power manager's projections, potentially violating the target performance bound. So, at its next opportunity, the CPU manager might start increasing the core frequency, inducing a similar response from the memory subsystem manager. Such oscillations waste energy. These unintended behaviors suggest that it is essential to coordinate power-performance management techniques across system components to ensure that the system is balanced to yield maximal energy savings.

To accomplish this coordinated control, we rely on execution profiling of core and memory access performance, using existing and new performance counters. Through counter readings and analytic models of core and memory performance and power consumption, we assess opportunities for per-core voltage and frequency scaling in a chip multiprocessor (CMP), voltage and frequency scaling of the on-chip memory controller (MC), and frequency scaling of memory channels and DRAM devices.

The fundamental innovation of CoScale is the way it efficiently searches the space of per-core and memory frequency settings (we set voltages according to the selected frequencies) in software. Essentially, our epoch-based policy estimates, via our performance counters and online models, the energy and performance cost/benefit of altering each component's (or set of components') DVFS state by one step, and iterates to greedily select a new frequency combination for cores and memory. The selected combination trades off core and memory scaling to minimize full-system energy while respecting a user-defined performance degradation bound. CoScale is implemented in the operating system (OS), so an epoch typically corresponds to an OS time quantum.

For comparison, we demonstrate the limitations of fully uncoordinated and semi-coordinated control (i.e., independent controllers that share a common estimate of target and achieved performance) of processor and memory DVFS. These strategies either violate the performance bound or oscillate wildly before settling into local minima. CoScale circumvents these problems by assessing processor and memory performance in tandem. In fact, CoScale provides energy savings close to an offline scheme that considers an exponential space of possible frequency combinations. We also quantify the benefits of CoScale versus CPU-only and memory-only DVFS policies.

1072-4451/12 $26.00 © 2012 IEEE
DOI 10.1109/MICRO.2012.22

IEEE computer society

Our results show that CoScale provides up to 24% *full-system* energy savings (16% on average) over a baseline scheme without DVFS, while staying within a 10% allowable performance degradation. Furthermore, we study CoScale's sensitivity to several parameters, including its effectiveness across performance bounds of 1%, 5%, 15%, and 20%. Our results demonstrate that CoScale meets the performance constraint while still saving energy in all cases.

2. Motivation and Related Work

Despite the advances in CPU power management, current servers remain non-energy-proportional, consuming a substantial fraction of peak power when completely idle [1]. To improve proportionality, researchers have recently proposed active low-power modes for main memory [7, 10]. CoScale takes a significant step in realizing effective server-wide power-performance tradeoffs using active low-power modes for both cores and memory. Next, we summarize some of the work on CPU and memory power management.

2.1. CPU Power Management

A large body of work has addressed the power consumption of CPUs. For example, studies have quantified the benefits of detecting periods of server idleness and rapidly transitioning cores into *idle low-power states* [30]. However, such states do not work well under moderate or high utilization. In contrast, processor *active low-power modes* provide better power-performance characteristics across a wide range of utilizations. Here, DVFS provides substantial power savings for small changes in voltage and frequency, in exchange for moderate performance loss. Processor DVFS is a well-studied technique [16, 20, 37] that is effective for a variety of workloads.

Processor DVFS techniques typically either rely on modeling or measurements (and feedback) to determine the next frequency to use. Invariably, these techniques assume that the memory subsystem will behave the same, regardless of the particular frequency chosen for the processor(s).

2.2. Memory Power Management

While CPUs have long been a focus of power optimizations, memory power management is now seeing renewed interest, e.g. [7, 9, 10, 38, 41]. As with processors, idle low-power states (e.g., precharge powerdown, self-refresh) have been extensively studied, e.g. [11, 22, 27, 28, 34]. However, past work has shown that active low-power modes are more successful at garnering energy savings for server workloads [9, 10, 31]. In particular, the memory bus is often underutilized for long periods, providing ample opportunities for memory power management.

To harness these opportunities, we recently proposed MemScale, a technique that leverages dynamic profiling, performance and power modeling, DVFS of the MC, and DFS of the memory channels and DRAM devices [10]. David *et al.* also studied memory DVFS [7]. In both these works, memory system scaling was done in the absence of core power management.

2.3. Integrated Approaches and CoScale

Researchers have only rarely considered coordinating management across components [6, 5, 13, 28, 36]. Raghavendra *et al.* considered how best to coordinate managers that operate at different granularities, but focused solely on processor power [36]. Much as we find, they showed that uncoordinated approaches can lead to destructive and unpredictable interactions among the managers' actions.

A few works have considered coordinated processor and memory power management for energy conservation [13, 27]. However, unlike these works, which assume only idle low-power states for memory, we concentrate on the more effective active low-power modes for memory (and processors). This difference is significant for two reasons: (1) Although the memory technology in these earlier studies (RDRAM) allowed per-memory-chip power management, modern technologies only allow management at a coarse grain (e.g., multi-chip memory ranks), complicating the use of idle low-power states; and (2) active memory low-power modes interact differently with the cores than idle memory low-power states. Moreover, these earlier works focused on single-core CPUs, which are easier to manage than CMPs. In a different vein, Chen *et al.* considered coordinated management of the processor and the memory for capping power consumption (rather than conserving energy), again assuming only idle low-power states [6]. Also assuming a power cap, Felter *et al.* proposed coordinated power shifting between the CPU and the memory by using a traffic throttling mechanism [14]. CoScale can be readily extended to cap power with appropriate changes to its decision algorithm and epoch length.

Perhaps the most similar work to CoScale is that of Li *et al.* [27], which also seeks to conserve CPU and memory energy subject to a performance bound. Their study investigates the combination of CPU microarchitectural adaptations (but could easily be extended to CPU DVFS) and memory idle low-power states, adapting the delay threshold before a memory device is transitioned to sleep. However, the study considers only a single-core CPU and a memory system with few low-power states. As such, their design is able to employ a policy that experimentally profiles each processor low-power configuration. The policy then profiles different combinations of processor and memory idle threshold configurations. It uses phase detection techniques and a history-based predictor to select the best state combination based on past measurements. Such a profiling-based approach is not viable for a large multicore with per-core and memory DVFS settings, due to the combinatorial explosion of possible states. Moreover, it is unclear how to extend their phase-based prediction for multi-programmed workloads; a proper configuration must be learned for each phase combination across all programs that may execute concurrently. CoScale's most fundamental advance is that it can optimize over a far larger combinatorial space. The large space is tractable because CoScale profiles performance at current settings and then uses simple models to predict power/performance at other settings.

3. CoScale

CoScale leverages three key mechanisms: core and memory subsystem DVFS, and a performance management scheme that keeps track of how much energy conservation has slowed down applications.

Core DVFS. We assume that each core can be voltage and frequency scaled independently of the other cores, as in [21, 40]. We also assume the shared L2 cache sits in a separate voltage domain that does not scale. A core DVFS transition takes a few 10's of microseconds.

Memory DVFS. Our memory DVFS method is based on MemScale [10], which dynamically adjusts MC, bus, and DIMM frequencies. Although it adjusts these frequencies together, we shall simply refer to adjusting the bus frequency. The DIMM clocks lock to the bus frequency (or a multiple thereof), while the MC frequency is fixed at double the bus frequency. Furthermore, MemScale adjusts the voltage

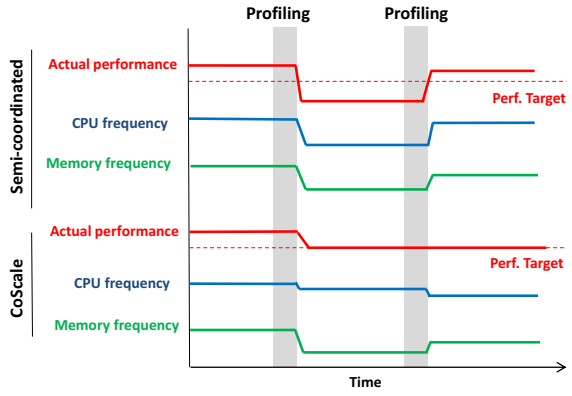


Figure 1: CoScale operation: Semi-coordinated oscillates, whereas CoScale scales frequencies more accurately.

of the MC (independently of core/cache voltage) and PLL/register in the DIMMs, based on the memory subsystem frequency.

Memory mode transition time is dominated by frequency re-calibration of the memory channels and DIMMs. The DIMM operating frequency may be reset while in the precharge powerdown or self-refresh state. We use precharge powerdown because its overhead is significantly lower than that of self-refresh. Most of the re-calibration latency is due to the DLL synchronization time, t_{DLLK} [32]—approximately 500 memory cycles.

Performance management. Similar to the approach initially proposed in [28] and later explored in [9, 10, 11, 34], our policy is based on the notion of program *slack*: the difference between a baseline execution and a target latency penalty that a system operator is willing to incur on a program to save energy. The basic idea is that energy management often necessitates running the target program with reduced core or memory subsystem performance. To constrain the impact of this performance loss, CoScale dictates that each executing program incurs no more than a pre-selected maximum slowdown γ, relative to its execution without energy management ($T_{MaxFreq}$). Thus, $Slack = T_{MaxFreq}(1+\gamma) - T_{Actual}$.

Overall operation. CoScale uses fixed-size epochs, typically matching an OS time quantum. Each epoch consists of a system profiling phase followed by the selection of core and memory subsystem frequencies that (1) minimize *full system* energy, while (2) maintaining performance within the target given by the accumulated slack from prior epochs.

In the system profiling phase, performance counters are read to construct application performance and energy estimates. By default, we profile for 300 μs, which we find to be sufficient to predict the resource requirements for the remainder of the epoch. Our default epoch length is 5 ms.

Based on the profiling phase, the OS selects and transitions to new core and/or memory bus frequencies using the algorithm described below. During a core transition, that core does not execute instructions; other cores can operate normally. To adjust the memory bus frequency, all memory accesses are temporarily halted, and PLLs and DLLs are resynchronized. Since the core and memory subsystem transition overheads are small (tens of microseconds) compared to our epoch size (milliseconds), the penalty is negligible.

The epoch executes to completion with the new voltages and frequencies. At the end of the epoch, CoScale again estimates the accumulated slack, by querying the performance counters and estimating what performance would have been achieved had the cores and the memory subsystem operated at maximum frequency. These estimates are then compared to achieved performance, with the difference used to update the accumulated slack and carried forward to calculate the target performance in the next epoch.

CoScale example. Figure 1 depicts an example of CoScale's behavior (bottom), compared to a policy that does not fully coordinate the processor and memory frequency selections (top). We refer to the latter policy as *semi-coordinated*, as it maintains a single performance slack (a mild form of coordination) that is shared by separate CPU and memory power state managers. As the figure illustrates, under semi-coordinated control, the CPU manager and the memory manager independently decide to scale down when they observe performance slack (performance above target). Unfortunately, because they are unaware of the cumulative effect of their decisions, they over-correct by scaling frequency too far down. For the same reason, in the following epoch, they over-react again by scaling frequency too far up. Such over-reactions continue in an oscillating manner. With CoScale, by modeling the joint effect of CPU and memory scaling, the appropriate frequency combination can be chosen to meet the precise performance target. Our control policy avoids both over-correction and oscillation.

3.1. CoScale's Frequency Selection Algorithm

When choosing a frequency for each core and a frequency for the memory bus, we have two goals. First, we wish to select a frequency combination that maximizes full-system energy savings. The energy-minimal combination is not necessarily that with the lowest frequencies; lowering frequency can increase energy consumption if the slowdown is too high. Our models explicitly account for the system-vs.-component energy balance. Fortunately, the cores and memory subsystem consume a large fraction of total system power, allowing CoScale to aggressively consume the performance slack. Second, we seek to observe the bound on allowable cycles per instruction (CPI) degradation for each running program.

Dynamically selecting the optimal frequency settings is challenging, since there are $M \times C^N$ possibilities, where M is the number of memory frequencies, C is the number of possible core frequencies, and N is the number of cores. M and C are typically on the order of 10, whereas N is in the range of 8-16 now but is growing fast. Thus, CoScale uses the greedy heuristic policy described in Figure 2.

Our gradient-descent heuristic iteratively estimates, via our online models, the marginal benefit (measured as $\Delta power/\Delta performance$) of altering either the frequency of the memory subsystem or that of various groups of cores by one step (we discuss core grouping in detail below). Initially, the algorithm estimates performance assuming all cores and memory are set to their highest possible frequencies (line 1 in the figure). It then iteratively considers frequency reductions, as long as some frequency can still be lowered without violating the performance slack (loop starting in line 2). When presented with a choice between next scaling down memory or a group of cores, the heuristic greedily selects the choice that will produce the highest marginal benefit (lines 3-12). If only memory or only cores can be scaled down, the available option is taken (line 13-19). Still in the main loop, the algorithm computes and records the full-system energy ratio (SER, Section 3.3) for the considered frequency configuration. When no more frequency reductions can be tried without violating the slack, the algorithm selects the configuration yielding the smallest SER (i.e., the best full-system energy savings) (line 21) and directs the hardware to transition frequencies (line 22).

```
1.  Estimate performance with each core and the memory subsystem at their highest frequencies
2.  While any component can be scaled down further without slack violation
3.    If both memory and at least one core can still scale down by 1 step
4.      If the memory frequency has changed since we last computed marginal_memory
5.        Compute marginal utility of lowering memory frequency as marginal_memory
6.      If any core frequency has changed since we last computed marginal_cores
7.        Compute marginal utility of lowering the frequency of core groups (per algorithm in Figure 3)
8.        Select the core group (group_best) with the largest utility (marginal_cores)
9.      If marginal_memory is greater than marginal_cores
10.       Scale down memory by 1 step
11.     Else
12.       Scale down cores in group_best by 1 step each
13.   Else if only memory can scale down
14.     Scale down memory by 1 step
15.   Else if only core groups can scale down
16.     If any core frequency has changed since we last computed marginal_cores
17.       Compute marginal utility of lowering the frequency of core groups (per algorithm in Figure 3)
18.       Select the core group (group_best) with largest marginal utility (marginal_cores)
19.     Scale down cores in group_best by 1 step each
20.   Compute and record the SER for the current combination of core and memory frequencies
21. Select the core and memory frequency combination with the smallest SER
22. Transition hardware to the new frequency combination
```

Figure 2: CoScale's greedy gradient-descent frequency selection algorithm.

```
1.  Scan the previous list of cores, removing any that may not scale down further or whose frequency has changed
2.  Re-insert cores with changed frequency, maintaining an ascending sort order by delta performance
3.  For group i from 1 to number of cores on the list
4.    Let delta power of the i-th group be equal to the sum of delta power from first to the i-th core
5.    Let delta performance be equal to delta performance of the i-th core
6.    Let marginal utility of i-th group be equal to delta power over delta performance just calculated
7.  Set the group with the largest marginal utility as the best group (group_best) and its utility as marginal_cores
```

Figure 3: Sub-algorithm to consider core frequency changes by group.

Changing the frequency of the memory subsystem impacts the performance of all cores. Thus, when we compute the $\Delta performance$ of lowering memory frequency, we choose the highest performance loss of any core. Similarly, when computing the $\Delta performance$ of lowering the frequencies of a group of cores, we consider the worst performance loss in the group. The $\Delta power$ in these cases is the power reduction that can be achieved by lowering the frequency of each core in the group.

An important aspect of the CoScale heuristic is that it considers lowering the frequency of cores in groups of 1, 2, 3, ..., N cores (lines 1-6 in Figure 3). The group formation algorithm maintains a list of cores that are eligible to scale down in frequency (i.e., they can be scaled down without slack violation), sorted in ascending order of $\Delta performance$. To avoid a potentially expensive sort operation on each invocation, the algorithm updates the existing sorted list by removing and then re-inserting only those cores whose frequency has changed (lines 1-2). N possible core groups are considered, forming groups greedily by first selecting the core that incurs the smallest delta performance from scaling (i.e., just the head of the list), then considering this core and the second core, then the third, and so on. This greedy group formation avoids combinatorial state space explosion, but, as we will show, it performs similarly to an offline method that considers all combinations. Considering transitions by group is needed to prevent CoScale from always lowering memory frequency first, because the memory subsystem at first tends to provide greater benefit than scaling any one core in isolation. Failing to consider group transitions may cause the heuristic to get stuck in local minima.

Our algorithm is run at the end of the profiling phase of each epoch (5ms by default). Because of core grouping, the complexity of our heuristic is $O(M + C \times N^2)$, which is exponentially better than that of the brute-force approach. Given our default simulation settings for M (10), C (10), and N (16), searching once per epoch has negligible overhead. Specifically, in all our experiments, searching takes less than 5 microseconds on a 2.4GHz Xeon machine. Our projections for larger core counts suggest that the algorithm could take 83 and 360 microseconds for 64 and 128 cores, respectively, in the worst case (4 microseconds in the best case). If one finds it necessary to hide these higher overheads, one can either increase the epoch length or dedicate a spare core to the algorithm.

3.2. Comparison with Other Policies

The key aspect of CoScale is the efficient way in which it searches the space of possible CPU and memory frequency settings. For comparison, we study five alternatives. The first, called "MemScale", represents the scenario in which the system uses only memory subsystem DVFS. The second alternative, called "CPUOnly", represents the scenario with CPU DVFS only. To be optimistic about this alternative, we assume that it considers all possible combinations of core frequencies and selects the best. In both MemScale and CPUOnly, the performance-aware energy management policy assumes that the behavior of the components that are not being managed will stay the same in the next epoch as in the profiling phase.

The third alternative, called "Uncoordinated", applies both MemScale and CPU DVFS, but in a completely independent fashion. In determining the performance slack available to it, the CPU power manager assumes that the memory subsystem will remain at the same frequency as in the previous epoch, and that it has accumulated no CPI degradation; the memory power manager makes the same assumptions about the cores. Hence, each manager believes that it alone influences the slack in each epoch, which is not the case. The fourth alternative, called "Semi-coordinated", increases the level of coordination slightly by allowing the CPU and memory power managers to share the same overall slack, i.e. each manager is aware of the past CPI degradation produced by the other. However, each

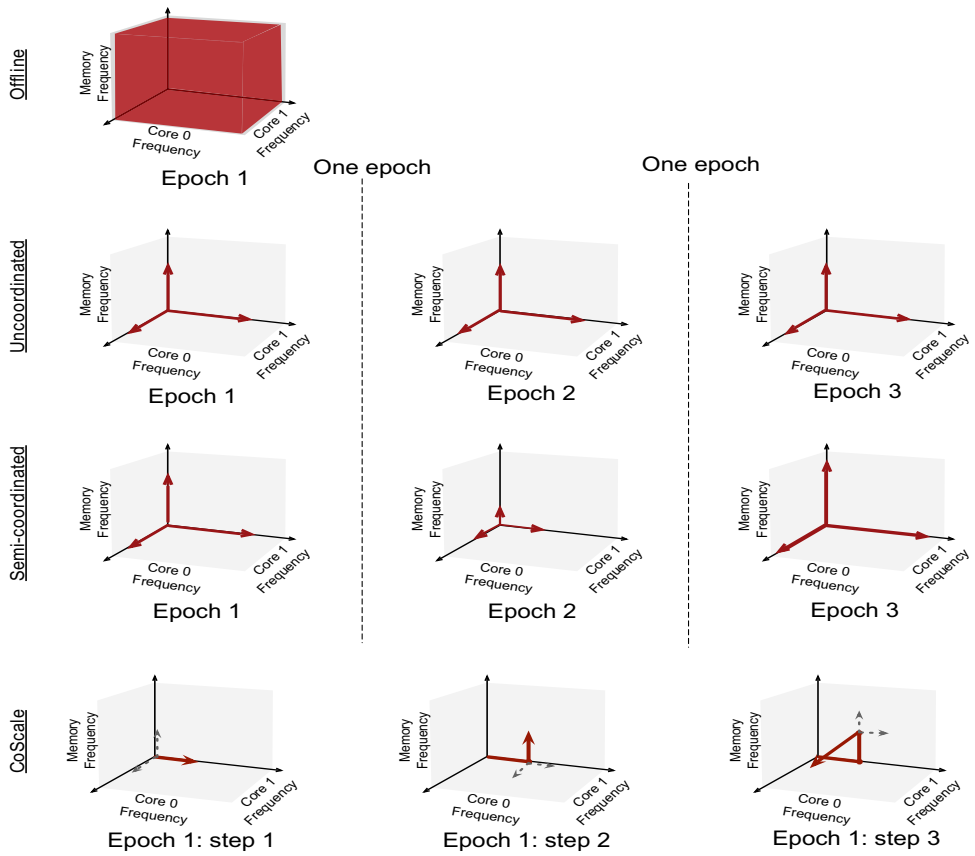


Figure 4: Search differences: CoScale searches the parameter space efficiently. Uncoordinated violates the performance bound and Semi-coordinated gets stuck in local minima.

manager still tries to consume the entire slack independently in each epoch (i.e., the two managers account for one another's past actions, but do not coordinate their estimate of future performance).

Finally, the fifth alternative, called "Offline", relies on a perfect offline performance trace for every epoch, and then selects the best frequency for each epoch by considering *all* possible core and memory frequency settings. As the number of possible settings is exponential, Offline is impractical and is studied simply as an upper bound on how well CoScale can do. However, Offline is not necessarily optimal, since it uses the same epoch-by-epoch greedy decision-making as CoScale (i.e., a hypothetical oracle might choose to accumulate slack in order to spend it in later epochs).

Figure 4 visualizes the difference between CoScale and other policies in terms of their search behaviors. For clarity, the figure considers only two cores (X and Y axes) and the memory (Z axis), forming a 3-D frequency space. The origin point is the highest frequency of each dimension; more distant points represent lower per-component frequencies. CPUOnly and MemScale search subsets of these three dimensions, so we do not illustrate them.

We can see from the figure that the Offline policy (top illustration) examines the entire space, thus always finding the best configuration. Under the Uncoordinated policy (second row), the CPU power manager tries to consume as much of the slack as possible with cores 0 and 1, while the memory power manager gets to consume the same slack. This repeats every epoch. Semi-coordinated (third row) behaves similarly in the first epoch. However, in the second epoch, to correct for the overshoot in the first epoch, each manager is restricted to a smaller search space. This restriction leads to over-correction in the third epoch, resulting in a much larger search space. The resulting oscillation may continue across many epochs. Finally, CoScale (bottom row) starts from the origin and greedily considers steps of memory frequency or (groups of) core frequency, selecting the move with the maximal marginal energy/performance benefit. From the figure, we can see that in step 1, CoScale scaled core 0 down by one frequency level; then it scaled the memory frequency down in step 2; and finally scaled core 1 down by two frequency levels in step 3. The search then terminates, because the performance model predicts that any further moves will violate the performance bound of at least one application. CoScale's greedy walk is shorter and produces better results than the other practical approaches.

Although CoScale provides no formal guarantees precluding oscillating behavior, this behavior is unlikely and occurs only when the profiling phases are consistently poor predictions of the rest of the epochs, or the performance models are inaccurate. On the other hand, the Semi-coordinated and Uncoordinated policies exhibit poor behavior due to their design limitations.

3.3. Implementation

We now describe the performance counters and performance/power models used by CoScale.

Performance counters. CoScale extends the performance modeling framework of MemScale [10] with additional performance counters

that allow it to estimate core power (in addition to memory power) and assess the degree to which a workload is instruction throughput vs. memory bound.

- **Instruction counts** – For each core, CoScale requires counters for *Total Instructions Committed* (`TIC`), *Total L1 Miss Stalls* (`TMS`), *Total L2 Accesses* (`TLA`), *Total L2 Misses* (`TLM`), and *Total L2 Miss Stalls* (`TLS`). CoScale uses these counters to estimate the fraction of CPI attributable to the core and memory, respectively. These counters allow the model to handle many core types (in-order, out-of-order, with or without prefetching), whereas MemScale's model (which required only `TIC` and `TMS`) supports only in-order cores without prefetching.
- **Memory subsystem performance** – CoScale reuses the same seven memory performance counters introduced by MemScale, which track memory queuing statistics and row buffer performance. We refer readers to [10] for details.
- **Power modeling** – To estimate core power, CoScale needs the L1 and L2 counters mentioned above and per-core sets of four *Core Activity Counters* (`CAC`) that track committed ALU instructions, FPU instructions, branch instructions, and load/store instructions. We reuse the memory power model from MemScale, which requires two counters per channel to track active vs. idle cycles and the number of page open/close events (details in [10]).

In total, CoScale requires eight additional counters per core beyond the requirements of MemScale (which requires two per core and nine per memory channel, all but five of which already exist in current Intel processors).

Performance model. Our model builds upon that proposed in [10], with two key enhancements: (1) we extend it to account for varying CPU frequencies, and (2) we generalize it to apply to cores with memory-level-parallelism (e.g., out-of-order cores or cores with prefetchers).

The performance model predicts the relationship between CPI, core frequency, and memory frequency, allowing it to determine the runtime and power/energy implications of changing core and memory performance. Given this model, the OS can set the frequencies to both maximize energy-efficiency and stay within the predefined limit for CPI loss.

CoScale models the rate of progress of an application in terms of CPI. The average CPI of a program is defined as:

$$E[\text{CPI}] = (E[TPI_{\text{CPU}}] + \alpha \cdot E[TPI_{\text{L2}}] + \beta \cdot E[TPI_{\text{Mem}}]) \cdot F_{\text{CPU}} \quad (1)$$

where $E[TPI_{\text{CPU}}]$ represents the average time that instructions spend on the CPU (including L1 cache hits), α is the fraction of instructions that access the L2 cache and stall the pipeline, $E[TPI_{\text{L2}}]$ is the average time that an L1-missing instruction spends accessing the L2 cache while the pipeline is stalled, β is the fraction of instructions that miss the L2 cache and stall the pipeline, $E[TPI_{\text{Mem}}]$ is the average time that an L2-missing instruction spends in memory while the pipeline is stalled, and F_{CPU} is the operating frequency of the core. The value of α can be calculated as the ratio of `TMS` and `TIC`, whereas β is the ratio of `TLS` and `TIC`.

The expected CPU time of each instruction ($E[TPI_{\text{CPU}}]$) depends on core frequency, but is insensitive to memory frequency. Since we keep the frequency (and supply voltage) of the L2 cache fixed, the expected time per L2 access that stalls the pipeline ($E[TPI_{\text{L2}}]$) does not change with either core or memory frequency (we neglect the secondary effect of small variations in L1 snoop time). The expected time per L2 miss that stalls the pipeline ($E[TPI_{\text{Mem}}]$) varies with memory frequency. We decompose the latter time as in [10]: $E[TPI_{\text{Mem}}] = \xi_{\text{bank}} \cdot (S_{\text{Bank}} + \xi_{\text{bus}} \cdot S_{\text{Bus}})$, where ξ_{bus} represents the average number of requests waiting for the bus; ξ_{bank} are requests waiting for the bank; S_{Bank} is the average time, excluding queueing delays, to access a bank (including precharge, row access and column read, etc); and S_{Bus} is the average data transfer (burst) time.

The above counters and model assume single-threaded applications, each running on a different core. To tackle multi-threaded applications, CoScale would require additional counters and a more sophisticated performance model (one that captures inter-thread interactions). To deal with context switching, CoScale can maintain the performance slack independently for each software thread.

Full-system energy model. Meeting the CPI loss target for a given workload does not necessarily maximize energy-efficiency. In other words, though additional performance degradation may be allowed, it may save more energy to run faster. To determine the best operating point, we construct a model to predict full-system energy usage as a function of the frequencies of the cores and memory subsystem.

For frequency f^i_{core} for core i and memory frequency f_{mem}, we define the *system energy ratio* (SER) as:

$$\text{SER}(f^1_{\text{core}}, ..., f^n_{\text{core}}, f_{\text{mem}}) = \frac{T_{f^1_{\text{core}},...,f^n_{\text{core}},f_{\text{Mem}}} \cdot P_{f^1_{\text{core}},...,f^n_{\text{core}},f_{\text{Mem}}}}{T_{\text{Base}} \cdot P_{\text{Base}}} \quad (2)$$

Here, T_{Base} and P_{Base} are time and average power at a nominal frequency (e.g., the maximum frequencies). $T_{f^1_{\text{core}},...,f^n_{\text{core}},\text{Mem}}$ is the time estimate for an epoch at frequencies $f^1_{\text{core}}, ..., f^n_{\text{core}}$ for the n cores and frequency f_{Mem} for the memory subsystem. This time estimate corresponds to the core with the highest CPI degradation compared to running at maximum frequency.

$$P_{f^1_{\text{core}},...,f^n_{\text{core}},f_{\text{Mem}}} = P_{\text{NonCoreL2OrMem}} + P_{\text{L2}} + P_{\text{Mem}}(f_{\text{Mem}}) + \sum_{i=1}^{n} P^i_{\text{Core}}(f^i_{\text{core}}). \quad (3)$$

In this formula, $P_{NonCoreL2OrMem}$ accounts for all system components other than the cores, the shared L2 cache, and the memory subsystem, and is assumed to be fixed. P_{L2} is the average power of the L2 cache and is computed from its leakage and number of accesses during the epoch. $P_{\text{Mem}}(f)$ is the average power of L2 misses and is calculated according to the model for memory power in [33]. We find that this average power does not vary significantly with core frequency (roughly 1-2% in our simulations); workload and memory bus frequency have a stronger impact. Thus, our power model assumes that core frequency does not affect memory power. $P^i_{\text{Core}}(f)$ is calculated based on the cores' activity factors using the same approach as prior work [3, 18]. We also find that the power of the cores is essentially insensitive to the memory frequency.

3.4. Hardware and Software Costs

We now consider CoScale's implementation cost. Core DVFS is widely available in commodity hardware, although each voltage domain may currently contain several cores. Though CPUs with multiple frequency domains are common, there have historically been few voltage domains; however, research has shown this is likely to change soon [21, 40].

Our design also may require enhancements to performance counters in some processors. Most processors already expose a set of counters to observe processing, caching and memory-related performance behaviors (e.g., row buffer hits/misses, row pre-charges).

Table 1: Workload descriptions.

Name	MPKI	WPKI	Applications (x4 each)			
ILP1	0.37	0.06	vortex	gcc	sixtrack	mesa
ILP3	0.27	0.07	sixtrack	mesa	perlbmk	crafty
ILP2	0.16	0.03	perlbmk	crafty	gzip	eon
ILP4	0.25	0.04	vortex	mesa	perlbmk	crafty
MID1	1.76	0.74	ammp	gap	wupwise	vpr
MID3	1.00	0.60	apsi	bzip2	ammp	gap
MID2	2.61	0.89	astar	parser	twolf	facerec
MID4	2.13	0.90	wupwise	vpr	astar	parser
MEM1	18.2	7.92	swim	applu	galgel	equake
MEM3	7.93	2.55	fma3d	mgrid	galgel	equake
MEM2	7.75	2.53	art	milc	mgrid	fma3d
MEM4	15.07	7.31	swim	applu	sphinx3	lucas
MIX1	2.93	2.56	applu	hmmer	gap	gzip
MIX3	2.55	0.80	equake	ammp	sjeng	crafty
MIX2	2.34	0.39	milc	gobmk	facerec	perlbmk
MIX4	2.35	1.38	swim	ammp	twolf	sixtrack

In fact, the latest Intel architecture exposes many MC counters for queues [25]. However, the existing counters may not conform precisely to the specifications required for our models.

When CoScale adjusts the frequency of a component, the component briefly suspends operation. However, as our policy operates at the granularity of multiple milliseconds, and transition latencies are in the tens of microseconds, the overheads are negligible. As mentioned above, the execution time of the search algorithm is not a major concern.

Existing DIMMs support multiple frequencies and can switch among them by transitioning to powerdown or self-refresh states [19], although this capability is typically not used by current servers. Integrated CMOS MCs can leverage existing DVFS technology. One needed change is for the MC to have separate voltage and frequency control from other processor components. In recent Intel architectures, this would require separating last-level cache and MC voltage control [17]. Although changing the voltage of DIMMs and DRAM peripheral circuitry is possible [23], there are no commercial devices with this capability.

4. Evaluation

We now present our methodology and results.

4.1. Methodology

Workloads. Table 1 describes the workload mixes we use. We construct the workloads by combining applications from the SPEC 2000 and SPEC 2006 suites. We use workloads exhibiting a range of compute and memory behavior, and group them into the same mixes as [10, 41]. The workload classes are: memory-intensive (MEM), compute-intensive (ILP), compute-memory balanced (MID), and mixed (MIX, one or two applications from each other class). The rightmost column of Table 1 lists the application composition of each workload; four copies of each application are executed to occupy all 16 cores.

We run the best 100M-instruction simulation point for each application (selected using Simpoints 3.0 [35]). A workload terminates when its slowest application has run 100M instructions. Table 1 lists the LLC misses per kilo-instruction (MPKI) and writebacks per kilo-instruction (WPKI). In terms of the workloads' running times, the memory-intensive workloads tend to run more slowly than the CPU-intensive ones. On average, the numbers of epochs are: 46 for MEM workloads, 32 for MIX, 15 for MID, and 10 for ILP.

Simulation infrastructure. Our evaluation uses a two-step simulation methodology. In the first step, we use M5 [4] to collect memory access traces (consisting of L1 cache misses and writebacks), and per-core activity counter traces. In the second step, we feed the memory traces into our detailed LLC/memory simulator of a 16-core CMP with a shared L2 cache (LLC), on-chip MC, memory channels, and DRAM devices. We also feed core activity traces, along with the run-time statistics from the L2 module, into McPAT [26] to dynamically estimate the CPU power. Overall, our infrastructure simulates in detail the aspects of cores, caches, MC, and memory devices that are relevant to our study, including memory device power and timing, and row buffer management.

Table 2: Main system settings.

Feature		Value
CPU cores		16 in-order, single thread, 4GHz Single IALU IMul FpALU FpMulDiv
L1 I/D cache (per core)		32KB, 4-way, 1 CPU cycle hit
L2 cache (shared)		16MB, 16-way, 30 CPU cycle hit
Cache block size		64 bytes
Memory configuration		4 DDR3 channels, 8 2GB ECC DIMMs
Time	tRCD, tRP, tCL	15ns, 15ns, 15ns
	tFAW	20 cycles
	tRTP	5 cycles
	tRAS	28 cycles
	tRRD	4 cycles
	Refresh period	64ms
Current	Row buffer read, write	250 mA, 250 mA
	Activation-precharge	120 mA
	Active standby	67 mA
	Active powerdown	45 mA
	Precharge standby	70 mA
	Precharge powerdown	45 mA
	Refresh	240 mA

Table 2 lists our default simulation settings. We simulate in-order cores with the Alpha ISA. Each core is allowed one outstanding LLC miss at a time. Like [10], we compensate for the lower memory traffic of these assumptions by simulating prefetching in Section 4.2.4. In the same section, we investigate an optimistic out-of-order design.

Table 2 also details the memory subsystem we simulate: 4 DDR3 channels, each of which populated with two registered, dual-ranked DIMMs with 18 DRAM chips each. Each DIMM also has a PLL device and 8 banks. Timing and power parameters are taken from Micron datasheets for 800 MHz devices [32].

Our simulated MC exploits bank interleaving and uses closed-page row buffer management, which outperforms open-page policies for multi-core CPUs [38]. Memory read requests (cache misses) are scheduled using FCFS, with reads given priority over writebacks until the writeback queue is half-full. More sophisticated memory scheduling is unnecessary for our single-issue workloads, as opportunities to increase bank hit rate via scheduling are rare.

We assume per-core DVFS, with 10 equally-spaced frequencies in the range 2.2-4.0 GHz. We assume a voltage range matching Intel's Sandybridge, from 0.65 V to 1.2 V, with voltage and frequency scaling proportionally, which matches the behavior we measured on an i7 CPU. We assume uncore components, such as the shared LLC, are always clocked at the nominal frequency and voltage.

As in [10], we scale MC frequency and voltage, but only frequency for the memory bus and DRAM chips. The on-chip 4-channel MC has the same voltage range as the cores, and its frequency is always double that of the memory bus. We assume that the memory bus and DRAM chips may be frequency-scaled from 800 MHz to 200 MHz, with steps of 66 MHz. We determine power at each frequency using Micron's calculator [32]. Transitions between bus frequencies are assumed to take 512 memory cycles plus 28 ns, which accounts for a DRAM state transition to fast-exit precharge powerdown and

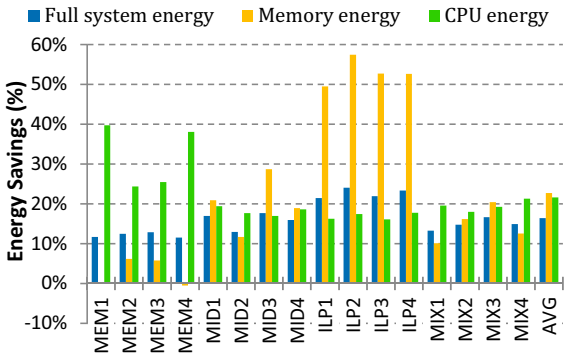


Figure 5: CoScale energy savings. CoScale conserves up to 24% of the full-system energy.

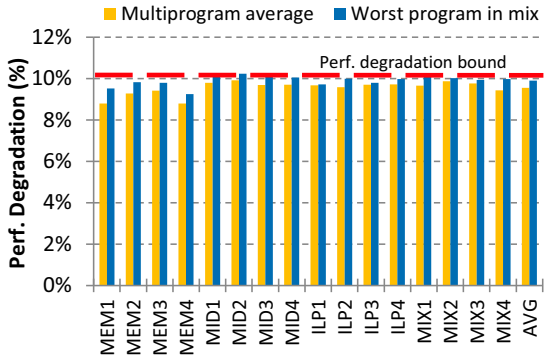


Figure 6: CoScale performance. CoScale never violates the 10% performance bound.

DLL re-locking [19, 10]. Some components' power draws also vary with utilization. Specifically, register and MC power scale linearly with utilization, whereas PLL power scales only with frequency and voltage. As a function of utilization, the PLL/register power ranges from 0.1 W to 0.5 W [10, 15, 17], whereas the MC power ranges from 4.5 W to 15 W.

We do not model power for non-CPU, non-memory system components in detail; rather, we assume these components contribute a fixed 10% of the total system power in the absence of energy management (we show the impact of varying this percentage in Section 4.2.4).

Under our baseline assumptions, at maximum frequencies, the CPU accounts for roughly 60%, the memory subsystem 30%, and other components 10% of system power.

4.2. Results

4.2.1. Energy and Performance We first evaluate CoScale with a maximum allowable performance degradation of 10%. We consider lower performance bounds in Section 4.2.4.

Figure 5 shows the full-system, memory, and CPU energy savings CoScale achieves for each workload, compared to a baseline without energy management (i.e., maximum frequencies). The memory energy savings range from -0.5% to 57% and the CPU energy savings range from 16% to 40%. As one would expect, the ILP workloads achieve the highest memory and lowest CPU energy savings, but still save at least 21% system energy.

The memory energy savings in the MID and MIX workloads are lower but still significant, whereas the CPU energy savings are somewhat higher (system energy savings of at least 13% for both workload classes). Note that CoScale is successful at picking the right energy saving "knob" in the MIX workloads. Specifically, it more aggressively conserves memory energy in MIX3, whereas it more aggressively conserves CPU energy in MIX1, MIX2, and MIX4.

The MEM workloads achieve the smallest memory and largest CPU energy savings (system energy savings of at least 12%), since their greater memory channel traffic reduces the opportunities for memory subsystem DVFS.

Figure 6 shows the average and maximum percent performance losses relative to the maximum-frequency baseline. The figure shows that CoScale never violates the performance bound. Moreover, CoScale translates nearly all the performance slack into energy savings, with an average performance loss of 9.6%, quite near the 10% target.

In summary, *CoScale conserves between 13% and 24% full-system energy for a wide range of workloads, always within the user-defined performance bounds.*

4.2.2. Dynamic Behavior To provide greater insight, we study an example of the dynamic behavior of CoScale in detail. Figure 7 plots the memory subsystem and core frequency (for milc in MIX2) selected by CoScale over time. For comparison, we also show the behavior of the Uncoordinated and Semi-coordinated policies.

Figure 7(a) shows that, in epoch two, CoScale reduces the core and memory frequencies to consume the available slack. In this phase, milc has low memory traffic needs, but the other applications in the mix preclude lowering the memory frequency further. Near epoch 10, another application's traffic spike results in a memory frequency increase, allowing a reduction of core frequency for milc. Near epoch 14, milc undergoes a phase change and becomes more memory-bound.

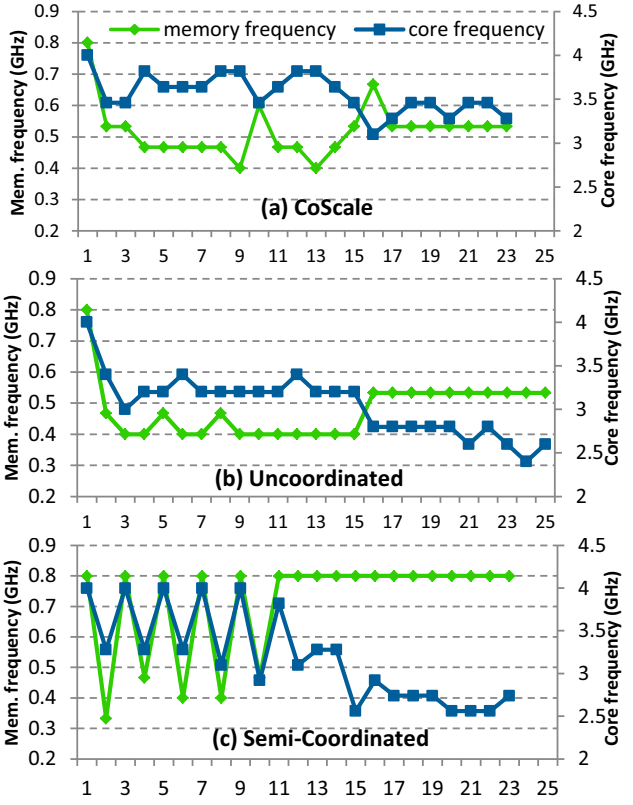


Figure 7: Timeline of the milc application in MIX2. Milc exhibits three phases. CoScale adjusts core and memory subsystem frequency precisely and rapidly in response to the phase changes. The other techniques do not.

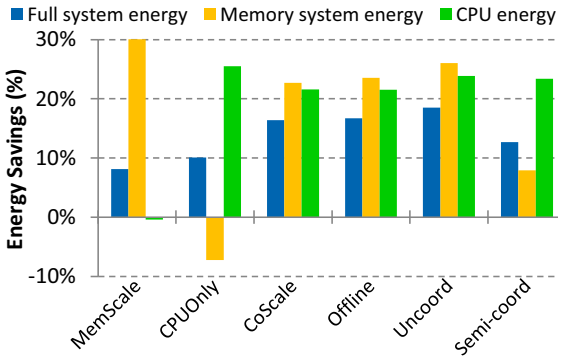


Figure 8: Energy savings. CoScale provides greater full-system energy savings than the practical policies.

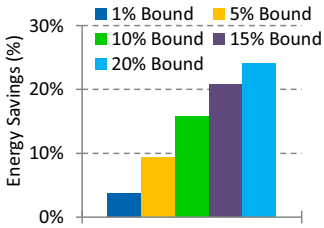


Figure 10: Impact of performance bound. Higher bound allows more savings without violations.

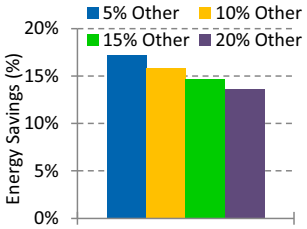


Figure 11: Impact of rest-of-system power. Savings still high for higher rest-of-system power.

As a result, CoScale increases the memory frequency, while reducing the core frequency.

Figure 7(b) shows a similar timeline for Uncoordinated. On the whole, the frequency transitions follow the same trend as in CoScale. However, both frequencies are markedly lower. Because there is no coordination, both CPU and memory power managers try to consume the same slack. These lower frequencies result in a longer running time (23 vs 25 epochs), violating the performance bound.

Figure 7(c) plots the timeline for Semi-coordinated. Initially, it incurs frequency oscillations until the traffic spike at epoch 10 causes memory frequency to become pegged at 800MHz. At that point, the CPU frequency for milc is also lowered considerably to consume all remaining slack. Unlike Uncoordinated, Semi-coordinated is successful in meeting the performance bound as slack estimation is coordinated among controllers. However, both the oscillations and the local minima selected after epoch 12 result in lower energy savings relative to CoScale. Altering the CPU and memory power managers to make their decisions half an epoch out of phase reduces oscillation, but the system gets stuck at local minima even sooner (around the 7th epoch). Making decisions an entire epoch out of phase produces similar behavior.

4.2.3. Energy and Performance Comparison Figure 8 contrasts average energy savings and Figure 9 contrasts average and worst-case performance degradation across polices. These results demonstrate that MemScale and CPUOnly are of limited use. Although they save considerable energy in the component they manage (MemScale conserves 30% memory energy, whereas CPUOnly conserves 26% CPU energy), gains are partially offset by higher energy consumption in the other component (longer runtime leads to higher background/leakage energy for the unmanaged component). These schemes save at most 10% full-system energy.

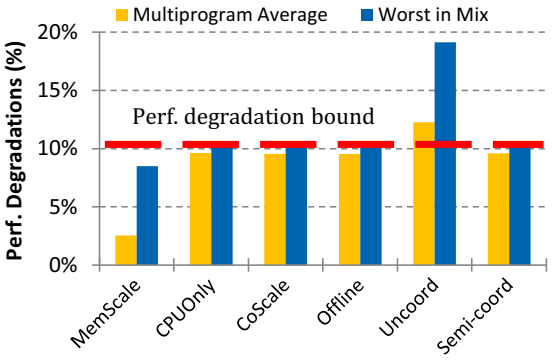


Figure 9: Performance. Uncoordinated is incapable of limiting performance degradation.

Uncoordinated conserves substantial memory and CPU energy, achieving the highest full-system energy savings of any scheme. Unfortunately, it is incapable of keeping the performance loss under the pre-defined 10% bound. In some cases, the performance degradation reaches 19%, nearly twice the bound. On the other hand, Semi-coordinated bounds performance well because the managers share the slack estimate. However, because of frequent oscillations and settling at sub-optimal local minima, Semi-coordinated consumes up to 8% more system energy (2.6% on average) than CoScale. Reducing oscillations by having the power managers make decisions out of phase does not improve results (0.3% lower savings with the same performance).

CoScale is more stable and effective than the other practical policies at conserving both memory and CPU energy, while staying within the performance bound. CoScale does almost as well as Offline. These results show that our heuristic for selecting frequencies is almost as effective as considering an exponential number of possibilities with prior knowledge of each workload's behavior.

4.2.4. Sensitivity Analysis To illustrate CoScale's behavior across different system and policy settings, we report on several sensitivity studies. In every case, we vary a single parameter at a time, leaving the others at their default values. Given the large number of potential experiments, we usually present results only for the MID workloads, which are sensitive to both memory and core performance.

Acceptable performance loss. In Figure 10, we vary the maximum *allowable* performance degradation, showing energy savings. Recall that our other experiments use a bound of 10%. As one would expect, 1% and 5% bounds produce lower energy savings, averaging 4% and 9%, respectively. Allowing 15% and 20% degradations saves more energy. In all cases, CoScale meets the configured bound, and provides greater percent energy savings than performance loss, even for tight performance bounds.

Rest-of-the-system power consumption. Figure 11 illustrates the effect of doubling and halving our assumption for non-memory, non-core power. When this power is doubled, CoScale still achieves 14% average full-system energy savings, whereas the savings increase to 17% when it is halved. In all cases performance remains within bounds (not shown).

Ratio of memory subsystem and CPU power. We also consider the effect of varying the ratio of memory subsystem to CPU power. Recall that, under our baseline power assumptions, CPU accounts for 60%, while memory accounts for 30% of total power at peak frequency (a CPU:Mem ration of 2:1). In Figure 12, we consider

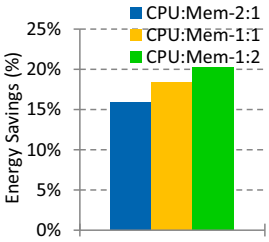


Figure 12: Impact of CPU:mem power, MID. Savings increase as memory power increases.

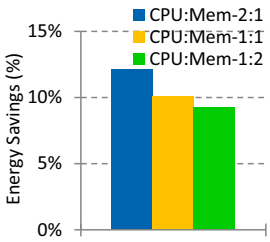


Figure 13: Impact of CPU:mem power, MEM. Savings decrease as memory power increases.

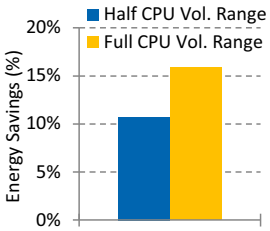


Figure 14: Impact of CPU voltage range. Smaller voltage ranges reduce energy savings.

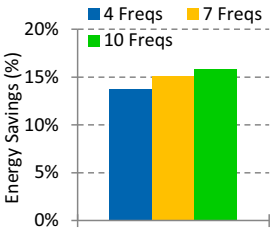


Figure 15: Impact of number of frequencies. Savings decrease little when fewer steps are avalaible.

1:1 and 1:2 ratios. CoScale achieves greater energy savings when the fraction of memory power is higher for the MID workloads. Interestingly, this trend is reversed for our MEM workloads (Figure 13), as most savings come from scaling the CPU.

CPU voltage range. We next consider the impact of a narrower CPU (and MC) voltage range, which reduces CoScale's ability to conserve core energy. Figure 14 shows results for a half-width range (0.95 1.2v) relative to our default assumption (0.65 1.2v). When the marginal utility of lowering CPU frequency decreases, CoScale scales the memory subsystem more aggressively and still achieves 11% full-system energy savings on average.

Number of available frequencies. By default, we assume 10 frequencies for both the CPU and the memory subsystem. Figure 15 shows results for 4 and 7 frequencies as well. As expected, the energy savings decrease as the granularity becomes coarser. However, CoScale adapts well, conserving only slightly less energy with fewer frequencies. With 4 frequencies the maximum performance loss is slightly lower than 10%, because the coarser granularity limits CoScale's ability to consume the slack precisely.

Prefetching. Next, we consider the impact of the increase in memory traffic that arises from prefetching. We implement a simple next-line prefetcher. This prefetcher is effective for these workloads, always decreasing the LLC miss rate. However, the prefetcher is not perfect; its accuracy ranges from 52% to 98% across our workloads. On average, it improves performance by almost 20% on MEM workloads, 8% on MIX, 4% on MID, and 1% for ILP. At the same time, it increases the memory traffic more than 33% on MEM, 20% on MID, 33% on MIX, and 13% on ILP. As one might expect, the higher memory traffic and instruction throughput result in higher memory and CPU power.

Figure 16 shows the full-system energy per instruction of three designs (Base+prefetching, Base+CoScale, and Base+prefetching+CoScale) normalized to our baseline (Base). We can see that the energy consumptions of Base+prefetching and Base are almost the same, except for the MEM workloads, since higher power and better performance roughly balance from an energy-efficiency perspective. Again except for MEM, the energy consumptions of Base+CoScale and Base+prefetching+CoScale are almost exactly the same, since average memory frequency is lower but CPU frequency is higher. For the MEM workloads, the performance improvement due to prefetching dominates the average power increase, so the average energy of Base+prefetching is 7% lower than Base. In addition, Base+prefetching+CoScale achieves 17% energy savings, compared to 12% from Base+CoScale. These results show that CoScale works well both with and without prefetching.

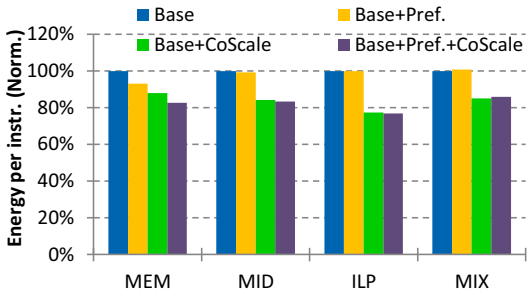


Figure 16: Impact of prefetching. CoScale works well with and without prefetching.

Out-of-Order. Although our trace-based methodology does not allow detailed out-of-order (OoO) modeling, we can approximate the latency hiding and additional memory pressure of OoO by emulating an instruction window during trace replay. We make the simplifying assumption that all memory operations within any 128-instruction window are independent, thereby modeling an upper bound on memory-level parallelism (MLP). Note that we still model a single-issue pipeline, hence, our instruction window creates MLP, but has no impact on instruction-level parallelism. Figure 17 compares the average CPI of the in-order and OoO designs, with and without CoScale, normalized to the in-order result. At one extreme, OoO drastically improves MEM, as memory stalls can frequently overlap. At the other extreme, ILP gains no benefit, since the infrequent L2 misses do not overlap frequently enough to impact performance. Note that, in the OoO+CoScale cases, performance remains within 10% of the OoO case; that is, CoScale is still maintaining the target degradation bound. Although we do not show these results in the figure, similar to the in-order case, Semi-coordinated on OoO meets the performance requirement, whereas Uncoordinated on OoO does not – Uncoordinated on OoO degrades performance by up to 16%, on a 10% performance loss bound.

Figure 18 shows average energy per instruction normalized to In-order. As we do not model any power overhead for OoO hardware structures (only the effects of higher instruction throughput and memory traffic), OoO always breaks even (ILP and MIX) or improves (MEM and MID) energy efficiency over In-order. Across the workloads, CoScale provides similar percent energy-efficiency gains for OoO as for In-order. The MEM case is the most interesting, as OoO has the largest impact on this workload. OoO increases memory bus utilization substantially (35% on average and up to 50%) and also results in far more queueing in the memory system (43% on average). The increased memory traffic balances with a reduced sensitivity

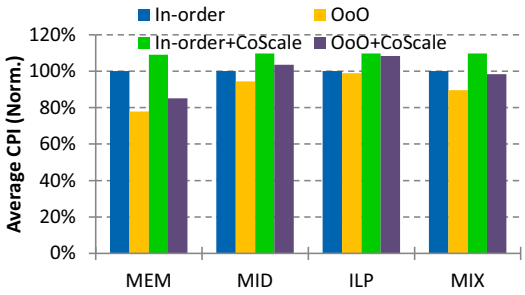


Figure 17: In-order vs OoO: performance. CoScale is within the performance bound in both in-order and OoO.

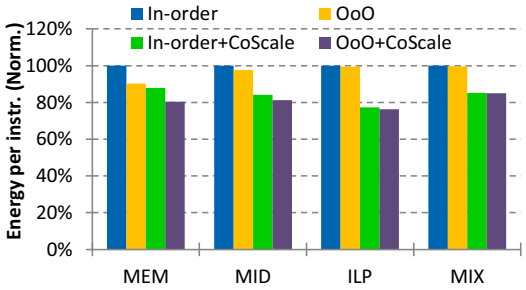


Figure 18: In-order vs OoO: energy. CoScale saves similar percent of energy in in-order and OoO.

to memory latency, and CoScale selects roughly the same memory frequencies under In-order and OoO. Interestingly, because of latency hiding, the MEM workload is more CPU-bound under OoO, and CoScale selects a slightly higher CPU frequency (5% higher on average). Again, we do not show results for Semi-coordinated and Uncoordinated on OoO in the figure, but their results are similar to those on an in-order design. Semi-coordinated on OoO causes frequency oscillation and leads to higher (up to 8%, and 4% on average) energy consumption than CoScale. Uncoordinated on OoO saves a little more energy (1% on average) than CoScale, but it violates the performance target significantly as mentioned above.

Summary. These sensitivity studies demonstrate that CoScale's performance modeling and control frameworks are robust—across the parameter space, CoScale always meets the target performance bound, while energy savings vary in line with expectations. Although the results in this subsection focused mostly on the MID workloads, we observed similar trends with the other workloads as well.

5. Conclusion

We proposed CoScale, a hardware-software approach for managing CPU and memory subsystem energy (via DVFS) in a coordinated fashion, under performance constraints. Our evaluation showed that CoScale conserves significant CPU, memory, and full-system energy, while staying within the performance bounds; that it is superior to four competing energy management techniques; and that it is robust over a wide parameter space. We conclude that CoScale's potential benefits far outweigh its small hardware costs.

Acknowledgements

This research was partially supported by Google and the National Science Foundation under grants #CCF-0916539, #CSR-0834403, and #CCF-0811320.

References

[1] L. A. Barroso and U. Hölzle. The Case for Energy-Proportional Computing. *IEEE Computer*, 40(12):33–37, 2007.

[2] L. A. Barroso and U. Hölzle. The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines. *Synthesis Lectures on Computer Architecture*, 2009.

[3] F. Bellosa. The Benefits of Event-Driven Energy Accounting in Power-Sensitive Systems. In *SIGOPS European Workshop '00*, 2000.

[4] N. Binkert, R. Dreslinski, L. Hsu, K. Lim, G. Saidi, and S. Reinhardt. The M5 Simulator: Modeling Networked Systems. *IEEE Micro*, 26(4), July 2006.

[5] R. Bitirgen, E. Ipek, and J. F. Martinez. Coordinated management of multiple interacting resources in chip multiprocessors: A machine learning approach. In *MICRO*, 2008.

[6] M. Chen, X. Wang, and X. Li. Coordinating Processor and Main Memory for Efficient Server Power Control. In *ICS*, 2011.

[7] H. David, C. Fallin, E. Gorbatov, U. Hanebutte, and O. Mutlu. Memory Power Management via Dynamic Voltage/Frequency Scaling. In *ICAC*, 2011.

[8] V. Delaluz, M. Kandemir, N. Vijaykrishnan, A. Sivasubramaniam, and M. J. Irwin. Hardware and Software Techniques for Controlling DRAM Power Modes. *IEEE Transactions on Computers*, 50(11), 2001.

[9] Q. Deng, D. Meisner, A. Bhattacharjee, T. F. Wenisch, and R. Bianchini. MultiScale: Memory System DVFS with Multiple Memory Controllers. In *ISLPED*, 2012.

[10] Q. Deng, D. Meisner, L. Ramos, T. F. Wenisch, and R. Bianchini. MemScale: Active Low-Power Modes for Main Memory. In *ASPLOS*, 2011.

[11] B. Diniz, D. Guedes, W. M. Jr, and R. Bianchini. Limiting the Power Consumption of Main Memory. *ISCA '07: International Symposium on Computer Architecture*, 2007.

[12] X. Fan, C. Ellis, and A. Lebeck. Memory Controller Policies for DRAM Power Management. In *ISLPED*, 2001.

[13] X. Fan, C. S. Ellis, and A. R. Lebeck. The Synergy between Power-aware Memory Systems and Processor Voltage Scaling. In *PACS*, 2003.

[14] W. Felter, K. Rajamani, T. Keller, and C. Rusu. A Performance-Conserving Approach for Reducing Peak Power Consumption in Server Systems. In *ICS*, 2005.

[15] E. Gorbatov, 2010. Personal communication.

[16] S. Herbert and D. Marculescu. Analysis of Dynamic Voltage/Frequency Scaling in Chip-Multiprocessors. In *ISLPED*, 2007.

[17] Intel. Intel ® Xeon ® Processor 5600 Series, 2010.

[18] C. Isci and M. Martonosi. Runtime Power Monitoring in High-End Processors: Methodology and Empirical Data. In *MICRO*, 2003.

[19] JEDEC. DDR3 SDRAM Standard, 2009.

[20] S. Kaxiras and M. Martonosi. Computer Architecture Techniques for Power-Efficiency. *Synthesis Lectures on Computer Architecture*, 2009.

[21] W. Kim, M. S. Gupta, G.-Y. Wei, and D. Brooks. System Level Analysis of Fast, Per-Core DVFS Using On-Chip Switching Regulators. In *HPCA*, 2008.

[22] A. R. Lebeck, X. Fan, H. Zeng, and C. Ellis. Power Aware Page Allocation. In *ASPLOS*, 2000.

[23] H.-W. Lee, K.-H. Kim, Y.-K. Choi, J.-H. Shon, N.-K. Park, K.-W. Kim, C. Kim, Y.-J. Choi, and B.-T. Chung. A 1.6V 1.4 Gb/s/pin Consumer DRAM with Self-Dynamic Voltage-Scaling Technique in 44nm CMOS Technology. In *ISSCC*, 2011.

[24] C. Lefurgy, K. Rajamani, F. Rawson, W. Felter, M. Kistler, and T. W. Keller. Energy Management for Commercial Servers. *IEEE Computer*, 36(12), December 2003.

[25] D. Levinthal. Performance Analysis Guide for Intel ® Core ™ i7 Processor and Intel ® Xeon ™ 5500 processors, 2009.

[26] S. Li, J. Ahn, R. Strong, J. Brockman, D. Tullsen, and N. Jouppi. McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures. In *MICRO*, 2009.

[27] X. Li, R. Gupta, S. Adve, and Y. Zhou. Cross-component energy management: Joint adaptation of processor and memory. In *ACM Trans. Archit. Code Optim.*, 2007.

[28] X. Li, Z. Li, F. M. David, P. Zhou, Y. Zhou, S. V. Adve, and S. Kumar. Performance-directed energy management for main memory and disks. In *ASPLOS*, 2004.

[29] K. Lim, J. Chang, T. Mudge, P. Ranganathan, S. K. Reinhardt, and T. F. Wenisch. Disaggregated Memory for Expansion and Sharing in Blade

Servers. In *ISCA*, 2009.
[30] D. Meisner, B. T. Gold, and T. F. Wenisch. PowerNap: Eliminating Server Idle Power. In *ASPLOS*, 2009.
[31] D. Meisner, C. M. Sadler, L. A. Barroso, W.-D. Weber, and T. F. Wenisch. Power Management of Online Data-Intensive Services. In *ISCA*, 2011.
[32] Micron. 1Gb: x4, x8, x16 DDR3 SDRAM, 2006.
[33] Micron. Calculating Memory System Power for DDR3, July 2007.
[34] V. Pandey, W. Jiang, Y. Zhou, and R. Bianchini. DMA-Aware Memory Energy Management. In *HPCA*, 2006.
[35] E. Perelman, G. Hamerly, M. V. Biesbrouck, T. Sherwood, and B. Calder. Using SimPoint for Accurate and Efficient Simulation Erez Perelman. In *SIGMETRICS*, 2003.
[36] R. Raghavendra, P. Ranganathan, V. Talwar, Z. Wang, and X. Zhu. No "Power" Struggles: Coordinated Multi-level Power Management for the Data Center. In *ASPLOS*, 2011.
[37] D. Snowdon, S. Ruocco, and G. Heiser. Power Management and Dynamic Voltage Scaling: Myths and Facts. In *Power Aware Real-time Computing*, 2005.
[38] K. Sudan, N. Chatterjee, D. Nellans, M. Awasthi, R. Balasubramonian, and A. Davis. Micro-Pages: Increasing DRAM Efficiency with Locality-Aware Data Placement. In *ASPLOS*, 2010.
[39] M. Ware, K. Rajamani, M. Floyd, B. Brock, J. Rubio, F. Rawson, and J. Carter. Architecting for Power Management: The IBM POWER7 Approach. In *HPCA*, 2010.
[40] G. Yan, Y. Li, Y. Han, X. Li, M. Guo, and X. Liang. AgileRegulator: A Hybrid Voltage Regulator Scheme Redeeming Dark Silicon for Power Efficiency in a Multicore Architecture. In *HPCA*, 2012.
[41] H. Zheng, J. Lin, Z. Zhang, and Z. Zhu. Decoupled DIMM: Building High-Bandwidth Memory System Using Low-Speed DRAM Devices. In *ISCA*, 2009.

2012 IEEE/ACM 45th Annual International Symposium on Microarchitecture

Predicting Performance Impact of DVFS for Realistic Memory Systems

Rustam Miftakhutdinov[†] Eiman Ebrahimi[‡] Yale N. Patt[†]
[†]The University of Texas at Austin [‡]Nvidia Corporation
{rustam,patt}@hps.utexas.edu ebrahimi@hps.utexas.edu

Abstract

Dynamic voltage and frequency scaling (DVFS) can make modern processors more power and energy efficient if we can accurately predict the effect of frequency scaling on processor performance. State-of-the-art DVFS performance predictors, however, fail to accurately predict performance when confronted with realistic memory systems. We propose CRIT+BW, the first DVFS performance predictor designed for realistic memory systems. In particular, CRIT+BW takes into account both variable memory access latency and performance effects of prefetching. When evaluated with a realistic memory system, DVFS realizes 65% of potential energy savings when using CRIT+BW, compared to less than 34% when using previously proposed DVFS performance predictors.

1. Introduction

Dynamic voltage and frequency scaling (DVFS) [1, 15] enables significant improvements in power and energy efficiency of modern processors. With DVFS support, a processor can alter its performance and power consumption on the fly by changing its frequency and supply voltage. This ability allows the processor to continuously adapt to dynamically changing application characteristics.

Exploiting the full potential of DVFS requires accurate performance and power prediction. If the processor can accurately predict what its performance and power consumption would be at any operating point, it can switch to the optimal operating point for any efficiency metric (e.g., energy or energy-delay-squared).

Existing DVFS performance predictors, however, fail to accurately predict performance under frequency scaling due to their unrealistic view of the off-chip memory system. Recently, two DVFS performance predictors have been proposed: *leading loads* [12, 20, 34][1] and *stall time* [12, 20]. Both assume a linear DVFS performance model, which, as we show in Section 3.2, does not model the performance effects of prefetching. In addition, leading loads was inspired by a simplified constant access latency view of memory and breaks down when confronted with a more realistic variable latency memory system. Figure 1 illustrates how the fraction of potential energy savings[2] actually realized by leading loads and stall time on memory-intensive workloads decreases as we increase the realism of the modeled memory system.

In this paper, we propose CRIT+BW, the first DVFS performance predictor for an out-of-order processor with a realistic DRAM system and a streaming prefetcher. We focus on the realism of the memory system because the effect of chip frequency scaling on performance depends largely on memory system behavior (as described in Section 2.2). Therefore, any DVFS performance predictor must be designed for and evaluated with a realistic memory system.

We develop CRIT+BW in two steps. First, we address variable memory access latency—a key characteristic of modern DRAM systems ignored by leading loads. To this end, we design CRIT, a DVFS performance predictor that accounts for variable memory access latency. The key idea is to predict the memory component of execution time by measuring the *critical path* through memory requests (hence the name "CRIT"). Second, we show that in the presence of prefetching, performance may be limited by achievable DRAM bandwidth—an effect ignored in the linear DVFS performance model used by leading loads and stall time. We develop a new *limited bandwidth* DVFS performance model that accounts for this effect and extend CRIT to use this performance model; CRIT+BW is the result ("BW" is shorthand for "bandwidth").

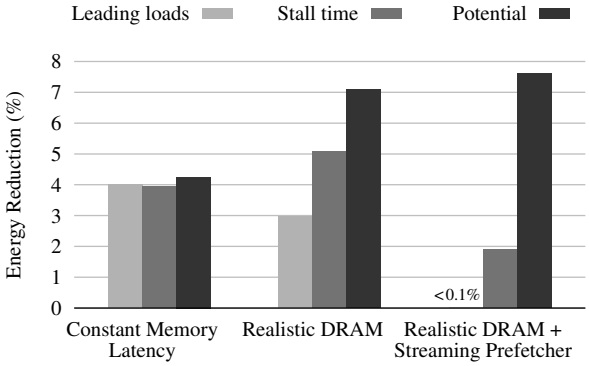


Figure 1: Energy savings realized by leading loads and stall time versus potential energy savings on 13 memory-intensive SPEC 2006 benchmarks

We evaluate CRIT+BW on an out-of-order processor capable of scaling the chip frequency from 1.5 GHz to 4.5 GHz, featuring a streaming prefetcher and a modern 800 MHz DDR3 SDRAM memory system. Across SPEC 2006, CRIT+BW realizes 65% of potential energy savings, compared to 34% for stall time and 12% for leading loads.

2. Background

2.1. Dynamic Voltage and Frequency Scaling

Dynamic voltage and frequency scaling (DVFS) [1, 15] helps increase power and energy efficiency of modern processors. DVFS does so by allowing the processor to switch between *operating points* (voltage/frequency combinations) at runtime. This capability gives rise to the problem of choosing the optimal operating point at runtime.

Traditionally, DVFS has been applied at the chip level only; recently, however, other DVFS domains have been proposed. David et al. [9] propose DVFS for off-chip memory and Intel's Westmere [23] supports multiple voltage/clock domains inside the chip. In this work, we focus on chip level DVFS.

2.2. DVFS Performance and Power Prediction

Estimating the performance impact of changing the chip's operating point is critical to choosing the optimal operating point. Which operating point is optimal depends on the chosen efficiency metric, e.g., energy or energy-delay-squared. All commonly used efficiency

[1]These three works propose very similar techniques. We use the name "leading loads" from Rountree et al. [34] for all three proposals.

[2]Section 4.3 describes the *dynamic optimal* DVFS policy used to calculate potential energy savings.

1072-4451/12 $26.00 © 2012 IEEE
DOI 10.1109/MICRO.2012.23

IEEE computer society

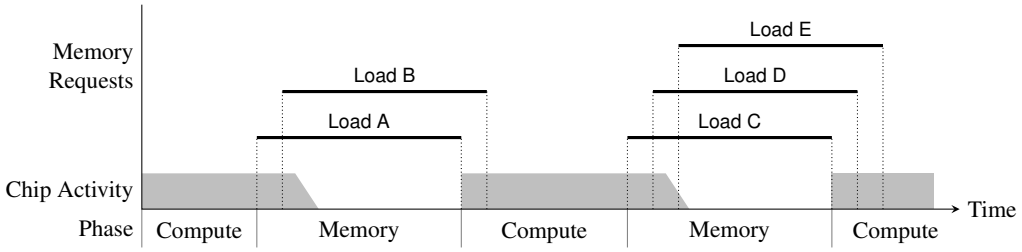


Figure 2: Two-phase abstract view of out-of-order execution used by leading loads

metrics are functions of execution time and power. Hence, choosing an optimal operating point requires a prediction of both performance and power at each of the available operating points. In this paper, we focus on performance prediction.

Predicting performance at different chip frequencies is made particularly difficult by the interaction between frequency scaling and the memory system's effect on performance. While memory latencies (as measured in seconds) are not affected by chip frequency scaling, they do scale with chip frequency in terms of processor cycles. The impact of these delay fluctuations on processor performance depends on the application, which further complicates DVFS performance prediction.

A DVFS performance predictor generally employs a *performance model* that predicts performance across a range of frequencies based on parameters measured at runtime. Specifically, the predictor measures these parameters during an execution interval and feeds them into the performance model to produce a performance estimate for every available frequency. These estimates, together with the corresponding power estimates, are then used to select the estimated optimal operating point for the next execution interval. Once the next execution interval ends, the process repeats.

Most published DVFS performance predictors [5–8, 10, 25, 29] rely on existing performance counters as inputs to their performance models. Many [5–8,25] use statistical regression analyses to correlate measured parameters with observed performance. An alternative approach is to design new hardware counters based on insight into the microarchitectural effects of frequency scaling, as done by the leading loads and stall time mechanisms described below.

2.2.1. Leading Loads. Leading loads [12, 20, 34] is a state-of-the-art DVFS performance predictor for out-of-order processors. The leading loads predictor was designed based on two simplifying assumptions about the memory system:

1. all memory requests have the same latency, and
2. after an instruction fetch or a data load misses in the last level cache and generates a memory request, the processor continues to execute but eventually runs out of ready instructions and stalls before the memory request returns.

Figure 2 shows the abstract view of execution implied by these assumptions. In this view, the out-of-order processor splits its time between two alternating phases: compute and memory. In the *compute* phase, the processor runs without generating any memory requests due to instruction fetches or data loads. As soon as the processor generates the first such memory request, the compute phase ends and the *memory* phase begins. At first, the out-of-order processor continues to execute instructions independent of the original memory request and may generate more memory requests. Eventually, however, the processor runs out of ready instructions and stalls. Since the processor generated the memory requests at roughly the same time, and the memory requests have the same latencies, they return data to the chip at about the same time as well. As soon as the first memory request returns, the memory phase ends and another compute phase begins.

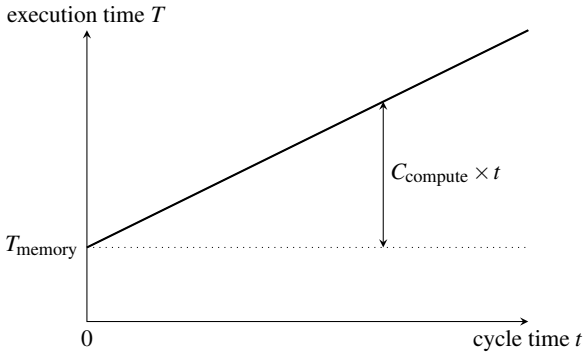


Figure 3: Linear DVFS performance model

This two-phase view of execution predicts a linear relationship between execution time T and chip cycle time t. To show this, we let

$$T = T_{\text{compute}} + T_{\text{memory}},$$

where T_{compute} denotes the total length of the compute phases and T_{memory} denotes the total length of the memory phases. As t changes due to DVFS, the number of cycles C_{compute} the chip spends in compute phases stays constant; hence

$$T_{\text{compute}}(t) = C_{\text{compute}} \times t.$$

Meanwhile, T_{memory} remains constant for every frequency. Thus, given measurements of C_{compute} and T_{memory} at any cycle time, we can predict execution time at any other cycle time:

$$T(t) = C_{\text{compute}} \times t + T_{\text{memory}}. \quad (1)$$

Figure 3 illustrates this linear model.

Leading loads introduces a hardware counter that continually accumulates T_{memory}. In each memory phase, the latency of the first memory request generated by a load is added to the counter; hence the name "leading loads." To estimate C_{compute}, leading loads employs existing performance counters to measure $T(t)$ and calculates

$$C_{\text{compute}} = \frac{T(t) - T_{\text{memory}}}{t}.$$

Note that, even though leading loads is derived from a simplified constant access latency view of memory, the mechanism can still be applied to more realistic memory systems.

2.2.2. Stall Time. Like leading loads, the *stall time* [12, 20] DVFS predictor uses the linear DVFS performance model. The key idea is

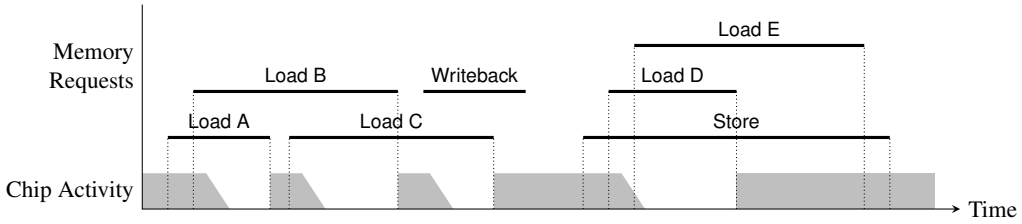


Figure 4: Abstract view of out-of-order processor execution with a variable latency memory system

simple: the time the processor spends unable to retire instructions due to an outstanding off-chip memory access should stay roughly constant as chip frequency is scaled (since this time depends largely on memory latency, which stays constant). The stall time predictor uses this time as the memory component of execution time (T_{memory} in Equation 1).

Unlike leading loads, the stall time predictor is not based on an abstract view of execution. The connection between retirement stalls due to memory accesses and T_{memory} is intuitive but not mathematically precise.

2.3. Realistic Memory System Architecture

Modern memory systems employ dynamic random access memory (DRAM) and streaming prefetching.

2.3.1. DRAM. In modern DRAM systems, contrary to leading loads' simplified constant access latency view of memory, memory request latency varies based on the addresses of the access stream [28]. Every memory address is statically mapped to one of several memory *banks* and one of many *rows* within its bank. Requests that map to different banks can be serviced in parallel, while those that map to the same bank have to be serviced serially. Among requests mapped to the same bank, requests that map to the same *row*, a 2–8 KB aligned block of memory, can be serviced faster than those that map to different rows. Memory request latencies also vary due to their wait time in the memory controller's queues.

2.3.2. Streaming Prefetcher. Streaming prefetchers are used in many commercial processors [2, 16, 24] and can greatly improve performance of memory intensive applications that stream through contiguous data arrays. Streaming prefetchers do so by detecting memory access streams and generating memory requests for data the processor will request further down stream. A well-performing streaming prefetcher significantly increases processor demand for memory bandwidth.

3. DVFS Performance Prediction on Realistic Memory Systems

We develop CRIT+BW, our DVFS performance predictor for realistic memory systems, in two steps.

First, we design CRIT, a DVFS performance predictor for a processor with a realistic DRAM system but no prefetcher. Like leading loads and stall time, CRIT measures the memory component of execution time within the confines of the linear DVFS performance model.

Second, we extend CRIT to account for performance effects of prefetching. We show that timely prefetching exposes the limiting effect of memory bandwidth on performance and develop a new DVFS performance model that accounts for this effect. The complete CRIT+BW predictor consists of the limited bandwidth DVFS performance model and hardware mechanisms that measure its parameters.

3.1. Realistic DRAM System with No Prefetching

Introducing a realistic DRAM system breaks the leading loads' abstract view of execution based on constant latency memory. Specifically, memory requests can now have very different latencies depending on whether they contend for DRAM banks and whether they map to the same row. Hence, the abstract view of processor execution relied on by leading loads becomes incorrect and, as we demonstrated in Figure 1, the predictor becomes ineffective.

Still, in the absence of prefetching, the other premise of leading loads (and stall time) still applies: after sending out a few instruction fetch or data load memory requests the processor eventually stalls. Figure 4 illustrates the abstract view of processor execution when memory latency is allowed to vary. Note that the processor eventually stalls under fetch and load memory requests.

This observation implies that execution time can still be modeled as the sum of a memory component whose latency remains constant under DVFS, and a compute component whose latency under DVFS changes in proportion to cycle time. Hence, the linear DVFS performance model (Equation 1) still applies in the case of a variable access latency memory system.

The introduction of variable memory access latencies, however, complicates the task of measuring the memory component T_{memory}. We must now calculate how execution time is affected by multiple memory requests with very different behaviors. Some of these requests are serialized (the first returns its data to the chip before the second one enters the memory controller). This serialization may be due to:

1. program dependencies (e.g., pointer chasing), or
2. limited core resources (e.g., if the out-of-order instruction window is too small to simultaneously contain both instructions corresponding to the two memory requests).

Other requests, however, overlap freely.

To estimate T_{memory} in this case, we recognize that in the linear DVFS performance model, T_{memory} is the limit of execution time as chip frequency approaches infinity (or, equivalently, as chip cycle time approaches zero). In that scenario, the execution time equals the length of the longest chain of dependent memory requests that stall the processor (i.e., data loads and instruction fetches). We refer to this chain as the *critical path* through the memory requests.

To calculate the critical path, we must know which memory requests are dependent (and remain serialized at all frequencies) and which are not. We observe that independent memory requests almost never serialize; the memory controller schedules independent requests as early as possible to overlap their latencies. Hence, we make the following assumption:

> If two memory requests are serialized (the first one completes before the second one starts), the second one depends on the first one.

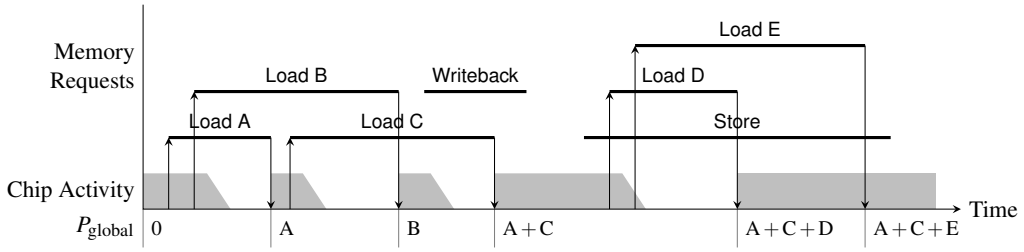


Figure 5: Critical path calculation example

3.1.1. Hardware Mechanism. We now describe CRIT, the hardware mechanism that uses the above assumption to estimate the critical path through load and fetch memory requests. CRIT maintains one global critical path counter P_{global} and, for each outstanding DRAM request i, a critical path timestamp P_i. Initially, the counter and timestamps are set to zero. When a request i enters the memory controller, the mechanism copies P_{global} into P_i. After some time ΔT the request completes its data transfer over the DRAM bus. At that time, if the request was generated by an instruction fetch or a data load, CRIT sets $P_{\text{global}} = \max(P_{\text{global}}, P_i + \Delta T)$. As such, after each fetch or load request i, CRIT updates P_{global} if request i is at the end of the new longest path through the memory requests.

Figure 5 illustrates how the mechanism works. We explain the example step by step:

1. At the beginning of the example, P_{global} is zero and the chip is in a compute phase.
2. Eventually, the chip incurs two load misses in the last level cache and generates two memory requests, labeled Load A and Load B. These misses make copies of P_{global}, which is still zero at that time.
3. Load A completes and returns data to the chip. Our mechanism adds the request's latency, denoted as A, to the request's copy of P_{global}. The sum represents the length of the critical path through Load A. Since the sum is greater than P_{global}, which is still zero at that time, the mechanism sets P_{global} to A.
4. Load A's data triggers more instructions in the chip, which generate the Load C request. Load C makes a copy of P_{global}, which now has the value A (the latency of Load A). Initializing the critical path timestamp of Load C with the value A captures the dependence between Load A and Load C: the latency of Load C will eventually be added to that of Load A.
5. Load B completes and ends up with B as its version of the critical path length. Since B is greater than A, B replaces A as the length of the global critical path.
6. Load C completes and computes its version of the critical path length as A + C. Again, since A + C > B, CRIT sets P_{global} to A + C. Note that A + C is indeed the length of the critical path through Load A, Load B, and Load C.
7. We ignore the writeback and the store because they do not cause a processor stall.
8. Finally, the chip generates requests Load D and Load E, which add their latencies to A + C and eventually result in P_{global} = A + C + E.

We can easily verify the example by tracing the longest path between dependent loads, which indeed turns out to be the path through Load A, Load C, and Load D. Note that, in this example, leading loads would incorrectly estimate T_{memory} as A + C + D.

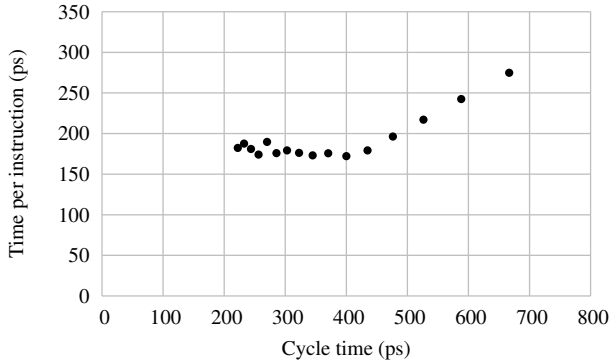


Figure 6: Time per instruction versus cycle time for `bwaves` with a streaming prefetcher enabled

3.2. Realistic DRAM System with Prefetching

Adding a prefetcher to the system changes the effect of DVFS on performance. Figure 6 shows time per instruction (TPI) for 100K retired instructions from `bwaves` at sixteen different cycle times (666–222 ps or 1.5–4.5 GHz) with a streaming prefetcher enabled. Note that these data points do not admit a linear approximation. This example is one of many where the linear performance model used by leading loads, stall time, and CRIT fails in the presence of prefetching.

The linear performance model fails due to the special nature of prefetching. Unlike demand memory requests, a prefetch request is issued in advance of the instruction that consumes the request's data. A prefetch request is *timely* if it fills the cache before the consumer instruction accesses the cache. Timely prefetches do not cause processor stalls; hence, their latencies do not affect execution time. Without stalls, however, the processor may generate prefetches at a high rate, exposing another performance limiter: the rate at which the memory system can satisfy memory requests (i.e., the memory bandwidth).

3.2.1. Limited Bandwidth Performance Model. We now describe a performance model, illustrated in Figure 7, that takes into account the performance limiting effect of finite memory bandwidth exposed by prefetching. This model splits the chip frequency range into two parts:

1. the low frequency range where the DRAM system can service memory requests at a higher rate than the chip generates them, and
2. the high frequency range where the DRAM system cannot service memory requests at the rate they are generated.

In the low frequency range, shown to the right of $t_{\text{crossover}}$ in Figure 7, the prefetcher runs ahead of the demand stream because the DRAM system can satisfy prefetch requests at the rate the prefetcher

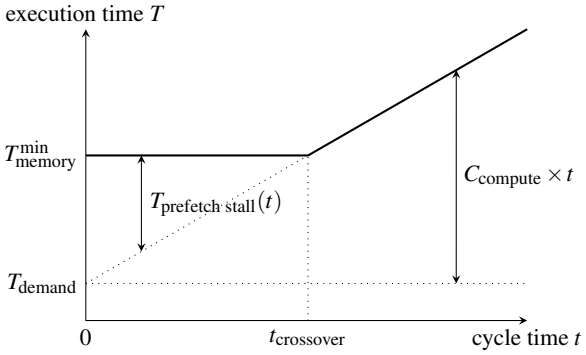


Figure 7: Limited bandwidth DVFS performance model

generates them. Hence, most prefetches are timely and instructions that use prefetched data result in cache hits. Execution time in this case is modeled by the original linear model, with only the non-prefetchable demand memory requests contributing to the memory component of the execution time, which we refer to as T_{demand}.

In the high frequency range, shown to the left of $t_{\text{crossover}}$ in Figure 7, the prefetcher fails to run ahead of the demand stream due to insufficient DRAM bandwidth. As the demand stream catches up to the prefetches, some demand requests stall the processor as they demand data that the prefetch requests have not yet brought into the cache. The delay due to these processor stalls is shown as $T_{\text{prefetch stall}}(t)$ in the figure.

Note that in the high frequency range the execution time is determined solely by $T^{\min}_{\text{memory}}$: the minimum time the DRAM system needs to satisfy all of the memory requests. Therefore, execution time does not depend on chip frequency in this case.

The limited bandwidth DVFS performance model shown in Figure 7 has three parameters:

1. the critical path through non-prefetchable demand memory requests T_{demand},
2. the number of cycles C_{compute} that the chip spends in the compute phase, and
3. the minimum time $T^{\min}_{\text{memory}}$ required by the DRAM system to satisfy the observed sequence of memory requests (both demands and prefetches).

Given the values of these parameters, we can estimate the execution time at any other cycle time t as follows:

$$T(t) = \max\left(T^{\min}_{\text{memory}},\ C_{\text{compute}} \times t + T_{\text{demand}}\right). \qquad (2)$$

3.2.2. Measuring Model Parameters. We now describe the hardware mechanisms to measure the parameters of the limited bandwidth DVFS performance model: T_{demand}, C_{compute}, and $T^{\min}_{\text{memory}}$. These mechanisms, together with the limited bandwidth DVFS performance model, comprise CRIT+BW—our complete DVFS performance predictor.

We measure T_{demand} in almost the same way as we measure T_{memory} in CRIT (Section 3.1): by calculating the critical path through memory requests. The only difference is that we exclude all prefetch requests and prefetchable demand requests from this calculation (just like we exclude stores and writebacks in CRIT). As shown in Figure 7, the extra chip stall time due to these prefetching-related requests, $T_{\text{prefetch stall}}(t)$, disappears at low frequencies. Therefore, this time is not a part of T_{demand}, which stays constant across frequencies.

To calculate C_{compute} we recognize that

$$T(t) = T_{\text{demand}} + C_{\text{compute}} \times t + T_{\text{prefetch stall}}(t).$$

We can solve this equation for C_{compute} if we can measure $T_{\text{prefetch stall}}(t)$. To this end, we introduce a new hardware counter that tracks the time the processor is stalled while only prefetch requests and prefetchable demand requests are outstanding. With $T_{\text{prefetch stall}}(t)$ now known, we have

$$C_{\text{compute}} = \frac{T(t) - T_{\text{demand}} - T_{\text{prefetch stall}}(t)}{t}.$$

Recall that $T^{\min}_{\text{memory}}$ is defined as the minimum time the DRAM system needs to satisfy all of the memory requests. We can calculate $T^{\min}_{\text{memory}}$ if we can measure the amount of slack $T_{\text{memory slack}}$ in the memory system, because

$$T^{\min}_{\text{memory}} = T(t) - T_{\text{memory slack}}(t). \qquad (3)$$

The description of the slack measurement hardware follows.

Whenever the memory controller schedules a DRAM command (e.g., "precharge" or "column access"), it must ensure that the command does not violate DRAM timing constraints. Hence, the memory controller can compute the *slack* of the DRAM command: how much earlier could the DRAM command have been scheduled without violating the DRAM timing constraints. The memory controller accumulates this slack separately for each DRAM bank and for the DRAM bus.

The presence of DRAM slack, however, does not always imply that the DRAM command could have been scheduled earlier. In fact, the slack may be due to the inability of the memory controller to schedule distant memory requests in parallel owing to the finite size of its scheduling window.

We account for this limitation when measuring slack in order to not overpredict the amount of reducible slack. To do this, we reset slack measurement every *slack measurement period*, which ends whenever the number of memory requests serviced within it reaches the size of the scheduling window. At the end of each slack measurement period, the memory controller finds the least slack among the banks and the bus. The memory controller adds the least slack amount to the global DRAM slack counter $T_{\text{memory slack}}$ and resets the bus and bank slack counters, starting a new period. From any cycle time t we can now calculate $T^{\min}_{\text{memory}}$ using Equation 3.

Storage Component	Quantity	Width	Bits
Global critical path counter P_{global}	1	32	32
Copy of P_{global} per memory request	32	32	1024
Global DRAM slack counter	1	32	32
DRAM bus slack counter	1	32	32
Per DRAM bank slack counters	8	16	128
Prefetch stall counter	1	32	32
Total bits			1280

Table 1: Hardware storage cost of CRIT+BW

3.3. Hardware Cost

Table 1 details the storage required by CRIT+BW. The additional storage is only 1280 bits. The mechanism does not add any structures or logic to the critical path of execution.

Frequency		Front end		OOO Core		All Caches			ICache	DCache	L2
Min	1.5 GHz	Uops/cycle	4	Uops/cycle	4	Line size	64 B	Size	32 KB	32 KB	1 MB
Max	4.5 GHz	Branches/cycle	2	Pipe depth	14	MSHRs	32	Assoc.	4	4	8
Step	100 MHz	BTB entries	4K	ROB size	128	Repl.	LRU	Cycles	3	3	18
		Predictor	hybrid[a]	RS size	48			Ports	1R/1W	2R/1W	1

DRAM Controller		Bus		DDR3 SDRAM [28]				Stream prefetcher [40]			
Policy	FR-FCFS [33]	Freq.	800 MHz	Chips	8×256 MB	Row size	8 KB	Streams	64	Distance	64
Window	32 requests	Width	8 B	Banks	8	CAS[b]	13.75 ns	Queue	128	Degree	4

[a] 64K-entry gshare + 64K-entry PAs + 64K-entry selector.
[b] CAS $= t_{\text{RP}} = t_{\text{RCD}} =$ CL; other modeled DDR3 constraints: CWL, $t_{\{\text{RC, RAS, RTP, BL, CCD, RRD, FAW, WTR, WR}\}}$.

Table 2: Simulated processor configuration

4. Methodology

We compare energy saved by CRIT+BW to that of the state-of-the-art (leading loads and stall time) and to three kinds of potential energy savings (computed using offline DVFS policies). Before presenting the results, we justify our choice of energy as the efficiency metric, describe our simulation methodology, explain how we compute potential energy savings, and discuss our choice of benchmarks.

4.1. Efficiency Metric

Our choice of efficiency metric is driven solely by the need to evaluate DVFS performance predictors. As such, the efficiency metric must be implementable by a simple DVFS controller (so that most of the benefit comes from DVFS performance prediction) and must allow comparisons to optimal results. Note that we are not evaluating the usefulness of DVFS itself.

We choose *energy* (or, equivalently,[3] *performance per watt*) by eliminating the other metrics from the set of the four commonly used ones: energy, energy delay product (EDP), energy delay-squared product (ED^{2}P), and execution time.

We eliminate EDP and ED^{2}P because they complicate DVFS performance predictor evaluation by 1) requiring another predictor in the DVFS controller, and 2) precluding comparisons to optimal results. Specifically, these metrics have the undesirable property that the optimal frequency for an execution interval depends on the behavior of the rest of execution. Therefore, the DVFS controller must keep track of past long-term application behavior and predict future long-term application behavior *in addition* to short-term DVFS performance prediction we are evaluating. The necessity of this additional prediction makes it hard to isolate the benefits of DVFS performance prediction in the results. This undesirable property also makes simulating an oracle DVFS controller infeasible, precluding comparisons to optimal results. Sazeides et al. [35] discuss these issues in greater detail.

We eliminate execution time as not applicable to chip-level DVFS. In this scenario, optimizing execution time does not require a performance prediction: the optimal frequency is simply the highest frequency.

Therefore, of the four common efficiency metrics, only energy is suitable for our evaluation.

4.2. Simulation Methodology

4.2.1. Timing Model. We use a cycle-accurate simulator of an x86 superscalar out-of-order processor. The simulator models port contention, queuing effects, and bank conflicts throughout the cache hierarchy and includes a detailed DDR3 SDRAM model. Table 2 lists the baseline processor configuration.

[3] Energy and performance per watt are equivalent in the sense that in any execution interval, the same operating point is optimal for both metrics.

Component	Parameter	Value	
		@1.5 GHz	@4.5 GHz
Chip	Static power (W)	9	28
	Peak dynamic power (W)	2	58
DRAM	Static power (W)	1	
	Precharge energy (pJ)	79	
	Activate energy (pJ)	46	
	Read energy (pJ)	1063	
	Write energy (pJ)	1071	
Other	Static power (W)	40	

Table 3: Power parameters

4.2.2. Power Model. We model three major system power components: chip power, DRAM power, and other power (fan, disk, etc.).

We model chip power using McPAT 0.8 [26] extended to support DVFS. Specifically, to generate power results for a specific chip frequency f, we:

1. run McPAT with a reference voltage V_0 and frequency f_0,
2. scale voltage using $V = \max(V_{\min}, \frac{f}{f_0} V_0)$,
3. scale reported dynamic power using $P = \frac{1}{2} C V^2 f$, and
4. scale reported static power linearly with voltage [3].

We model DRAM power using CACTI 6.5 [30] and use a constant static power as a proxy for the rest of system power.

Table 3 details the power parameters of the system.

4.2.3. DVFS Controller. Every 100K retired instructions, the DVFS controller chooses a chip frequency for the next 100K instructions.[4] Specifically, the controller chooses the frequency estimated to cause the least system energy consumption. To estimate energy consumption at a candidate frequency f while running at f_0, the controller:

1. obtains measurements of
   - execution time $T(f_0)$,
   - chip static power $P_{\text{chip static}}(f_0)$,
   - chip dynamic power $P_{\text{chip dynamic}}(f_0)$,
   - DRAM static power $P_{\text{DRAM static}}(f_0)$,
   - DRAM dynamic power $P_{\text{DRAM dynamic}}(f_0)$, and
   - other system power $P_{\text{other}}(f_0)$

[4] We chose 100K instructions because it is the smallest quantum for which the time to change chip voltage (as low as tens of nanoseconds [21, 22], translating to less than 1K instructions) can be neglected.

for the previous 100K instructions from hardware performance counters and power sensors,

2. obtains a prediction of execution time $T(f)$ for the next 100K instructions from the performance predictor (either leading loads, stall time, or CRIT+BW),
3. calculates chip dynamic energy $E_{\text{chip dynamic}}(f_0)$ and DRAM dynamic energy $E_{\text{DRAM dynamic}}(f_0)$ for the previous interval using $E = PT$,
4. calculates $E_{\text{chip dynamic}}(f)$ by scaling $E_{\text{chip dynamic}}(f_0)$ using $E = \frac{1}{2}CV^2$,
5. calculates $P_{\text{chip static}}(f) = \frac{V}{V_0}P_{\text{chip static}}(f_0)$ as in [3],
6. and finally calculates total estimated system energy

$$\begin{aligned} E(f) &= E_{\text{chip}}(f) + E_{\text{DRAM}}(f) + E_{\text{other}}(f) \\ &= P_{\text{chip static}}(f) \times T(f) + E_{\text{chip dynamic}}(f) + \\ &\quad P_{\text{DRAM static}}(f_0) \times T(f) + E_{\text{DRAM dynamic}}(f) + \\ &\quad P_{\text{other}}(f_0) \times T(f). \end{aligned}$$

To isolate the effect of DVFS performance predictor accuracy on energy savings, we do not simulate delays associated with switching between frequencies. Accounting for these delays requires an additional prediction of whether the benefits of switching outweigh the cost. If the accuracy of that prediction is low, it could hide the benefits of high performance prediction accuracy, and vice versa.

4.3. Offline Policies

We model three offline DVFS controller policies: *dynamic optimal*, *static optimal*, and *perfect memoryless*.

The *dynamic optimal* policy places a lower bound on energy consumption. We compute this bound as follows:

1. run the benchmark under study at each chip frequency,
2. for each interval, find the minimum consumed energy across all frequencies,
3. total the per-interval minimum energies.

The *static optimal* policy chooses the chip frequency that minimizes energy consumed by the benchmark under study, subject to the constraint that frequency must remain the same throughout the run. The difference between dynamic and static optimal results yields potential energy savings due to benchmark phase behavior.

The *perfect memoryless* policy simulates a perfect *memoryless* performance predictor. We call a predictor *memoryless* if it assumes that for each chip frequency, performance during the next interval equals performance during the last interval. This assumption makes sense for predictors that do not "remember" any state (other than the measurements from the last interval); hence the name "memoryless." Note that all predictors discussed in this paper are memoryless. For each execution interval, the perfect memoryless policy chooses the chip frequency that would minimize energy consumption during the *previous* interval.

The perfect memoryless policy provides a quasi-optimal[5] bound on energy saved by memoryless predictors. A large difference between dynamic optimal and perfect memoryless results indicates that a memoryless predictor cannot handle the frequency of phase changes in the benchmark under study. Getting the most energy savings out of such benchmarks may require "memoryful" predictors that can detect and predict application phases.[6] We leave such predictors to future work.

[5] We call this bound *quasi-optimal* because an imperfect memoryless predictor may actually save more energy than the perfect memoryless predictor if the optimal frequency for the previous interval does not remain optimal in the next interval.

4.4. Benchmarks

We simulate SPEC 2006 benchmarks compiled using the GNU Compiler Collection version 4.3.6 with the `-O3` option. We run each benchmark with the reference input set for 200M retired instructions selected using Pinpoints [32].

4.4.1. Benchmark Classification. To simplify the analysis of the results, we classify the benchmarks based on their memory intensity and the number of prefetch requests they trigger. We define a benchmark as *memory-intensive* if it generates more than 3 last level cache misses per thousand instructions (with no prefetching). We define a benchmark as *prefetch-heavy* if it triggers more than 5 prefetch requests per thousand instructions. The resulting benchmark classes are the same across all simulated frequencies.

5. Results

We show results for two configurations: with prefetching turned off and with a streaming prefetcher. In both cases, we show normalized energy reduction relative to the energy consumed at 3.7 GHz, the most energy-efficient static frequency across SPEC 2006 (which happens to be the same for both cases).

Before analyzing the results, we first explain their presentation using Figure 8 as an example. Note that, for each benchmark, the figure shows five bars within a wide box. The height of the box represents dynamic optimal energy reduction. Since no other DVFS policy can save more energy than dynamic optimal, we can use this box to bound the other five bars. The five bars inside the box represent energy reduction due to 1) leading loads, 2) stall time, 3) CRIT+BW, 4) optimal static DVFS policy, and 5) perfect memoryless DVFS policy. This plot design allows for easy comparisons of realized and potential gains for each benchmark and simplifies comparison of potential gains across benchmarks at the same time.

5.1. Realistic DRAM with No Prefetching

Figure 8 shows realized and potential energy savings across thirteen memory-intensive workloads. On average, CRIT+BW and stall time realize 5.5% and 5.1% out of potential 7.1% energy savings, whereas leading loads only realizes 3%. For completeness, Figure 9 shows energy savings for low memory intensity benchmarks (note the difference in scale).

The subpar energy savings by leading loads are due to its constant memory access latency approximation. As described in Section 2.2.1, leading loads accumulates the latency of the first load in each cluster of simultaneous memory requests to compute the memory component T_{memory} of total execution time T. It turns out that in such clusters, the leading load latency is usually *less* than that of the other requests. In fact, this is the case in all memory-intensive benchmarks except `libquantum` and `lbm`; in these eleven benchmarks the average leading load latency is only 74% of the average latency of the other memory requests. This discrepancy is due to the fact that the first memory request in a cluster is unlikely to contend with another request for a DRAM bank, whereas the later requests in the cluster

[6] Section 6.3 describes related work on phase prediction.

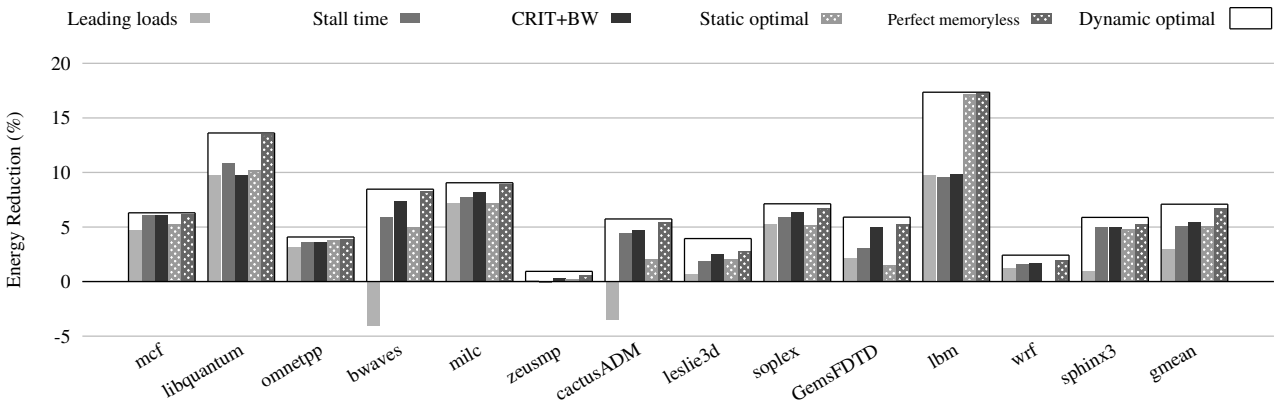


Figure 8: Realized and optimal energy savings for memory-intensive benchmarks (no prefetching)

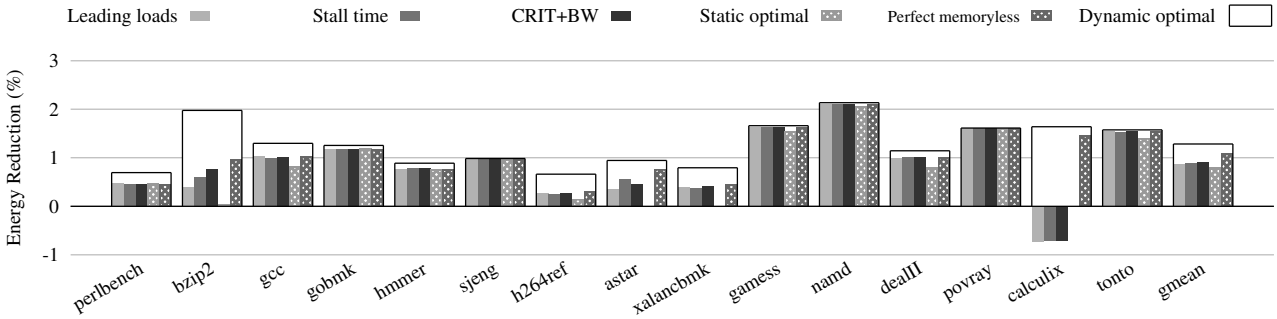


Figure 9: Realized and optimal energy savings for non-memory-intensive benchmarks (no prefetching)

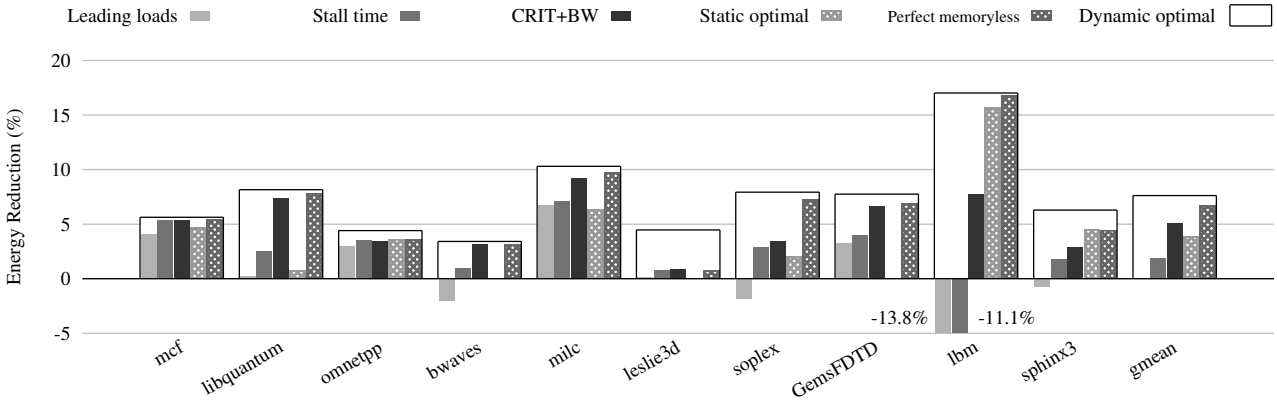


Figure 10: Realized and optimal energy savings for prefetch-heavy benchmarks

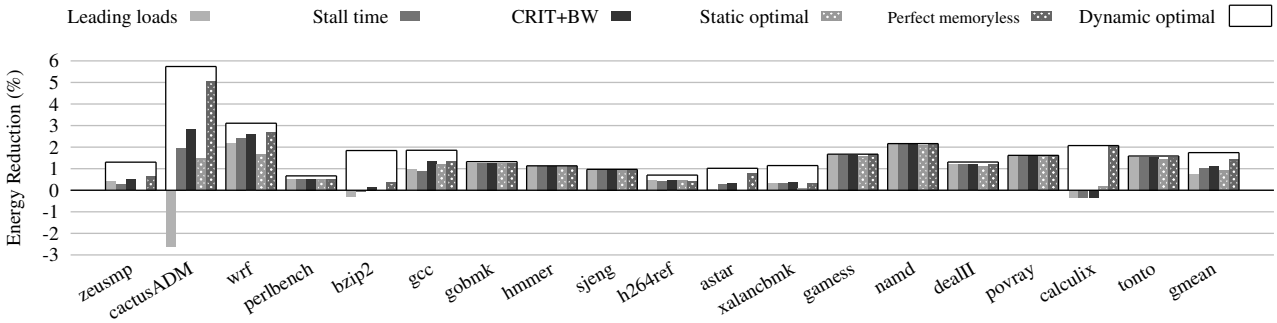


Figure 11: Realized and optimal energy savings for prefetch-light benchmarks

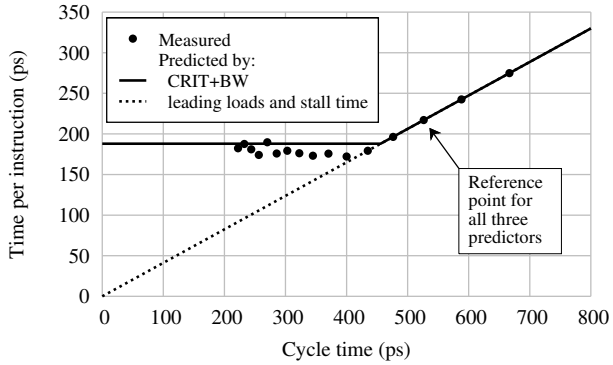


Figure 12: Measured and predicted TPI on `bwaves` with streaming prefetcher enabled

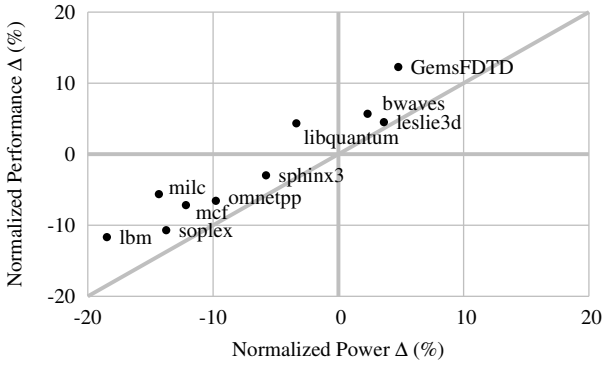


Figure 13: Performance delta versus power delta under DVFS with CRIT+BW for prefetch-heavy benchmarks

likely have to wait for the earlier ones to free up the DRAM banks. This underestimate of T_{memory} results in subpar energy savings, exemplified by `bwaves` and `cactusADM` on which leading loads actually consumes more energy than the baseline.

The fact that stall time beats leading loads supports our original argument that DVFS performance predictors must be designed for and evaluated with a realistic memory system. Both our experiments and prior work [12, 20] show that when evaluated with a constant access latency memory, leading loads saves more energy than stall time. Evaluation of the two predictors with a realistic DRAM system, however, shows this conclusion to be incorrect.

Note that CRIT+BW, the mechanism we derived from an abstract view of execution in Section 3, outperforms stall time, a mechanism based on a less precise view of execution, by a relatively small margin (5.5% vs. 5.1% energy saved). It is unclear, however, whether the approximations that make stall time work will hold in all configurations.

5.2. Realistic DRAM with Stream Prefetching

Figure 10 shows realized and potential energy reduction across ten prefetch-heavy benchmarks with a streaming prefetcher enabled. On average, CRIT+BW realizes 5% out of potential 7.6% energy savings, whereas stall time and leading loads only realize 1.8% and less than 0.1%, respectively. For completeness, Figure 11 shows energy savings for prefetch-light benchmarks (note the difference in scale).

5.2.1. Prediction Example. To provide insight into why CRIT+BW bests the competition on prefetch-heavy workloads, we analyze performance predictions generated by all three predictors for an interval of `bwaves`, the prefetch-heavy benchmark we use to motivate the limited bandwidth DVFS performance model in Section 3.2. Figure 12 contrasts the performance predictions generated by CRIT+BW, leading loads, and stall time. In particular, the figure shows:

1. sixteen thick dots representing measured time per instruction (TPI) at sixteen frequencies,
2. a dashed line showing TPI predicted by both leading loads and stall time while running at 1.9 GHz, and
3. a solid curve showing TPI predicted by CRIT+BW while running at 1.9 GHZ.

Note that all three predictions for low frequencies (right half of the figure) are identical. The reason lies in the highly streaming nature of `bwaves` that enables the prefetcher to eliminate *all* demand misses in the interval. Therefore, all three predictors estimate the memory component of execution time to be zero, predicting performance to scale proportionately with frequency.

On the other hand, predictions for higher frequencies diverge; CRIT+BW predicts that TPI saturates at 188 ps per instruction, whereas leading loads and stall time still predict performance to scale proportionally with frequency. Comparing the predictions to measured TPI demonstrates that accounting for limited memory bandwidth allows CRIT+BW to be more accurate than both leading loads and stall time.

Due to this prediction inaccuracy, a DVFS controller using either leading loads or stall time has to act on skewed estimates of energy consumption at high frequencies. Specifically, the controller may choose a high frequency and waste a lot of power for no performance benefit, losing out on potential energy savings.

5.2.2. Power and Performance Tradeoff. Figure 13 details how CRIT+BW trades off power and performance to reduce energy. The figure plots performance delta versus power delta (normalized to performance and power achieved at the baseline 3.7 GHz frequency). The diagonal line consists of points where performance and power deltas are equal, resulting in the same energy as the baseline.

CRIT+BW trades off power and performance differently across workloads. On `GemsFDTD`, `bwaves`, and `leslie`, CRIT+BW spends extra power for even more performance, while on `lbm`, `mcf`, `milc`, `omnetpp`, `soplex`, and `sphinx3` CRIT+BW allows performance to dip to save more power.

Note that CRIT+BW improves performance *and* saves power on `libquantum`. CRIT+BW does so by exploiting `libquantum`'s phase behavior. In some phases, CRIT+BW spends extra power for more performance; in others, it makes the opposite choice. On average, both performance and power consumption improve.

5.3. Analysis of `lbm`

With and without prefetching, `lbm` stands out due to its large potential energy savings which are not fully realized by CRIT+BW and the other predictors. The reasons, however, are different for each case.

Without prefetching, the reason lies in the peculiar nature of the benchmark. The majority (84%) of memory requests in `lbm` are stores and writebacks, which do not stall the processor. At high frequencies, however, the load memory requests are more likely to contend with these stores and writebacks for DRAM banks, taking more time to complete and thus violating the linear DVFS performance model assumption that T_{memory} stays the same across frequencies. This leads CRIT+BW (and the other predictors) to underestimate the performance effect of memory at high frequencies.

With prefetching, the reason lies in the details of memory request scheduling. At high frequencies, `lbm` floods the memory system with prefetches; this large number of memory requests allows the memory controller to make better scheduling decisions and reduce the number of row conflicts by up to 61%. The slack approach to estimating $T^{\min}_{\text{memory}}$ does not take this effect into account, resulting in an overestimate of $T^{\min}_{\text{memory}}$.

5.4. Summary

When evaluated on an out-of-order processor featuring a streaming prefetcher and a realistic DRAM system, CRIT+BW realizes 65% of dynamically optimal energy savings (75% of perfect memoryless energy savings) across all SPEC 2006 workloads, compared to only 34% (40%) for stall time and 12% (14%) for leading loads.

6. Related Work

To our knowledge, this paper is the first to propose a DVFS performance predictor designed to work with a realistic DRAM system. Specifically, our predictor addresses two characteristics of realistic DRAM systems which make DVFS performance prediction difficult: varying memory request latencies and prefetching, neither of which are considered by the state-of-the-art [12, 20, 34].

We have already compared our predictor to leading loads and stall time. Here we briefly discuss three major areas of related work: performance and power prediction for DVFS, analytical performance models, and phase prediction.

6.1. Performance and Power Prediction for DVFS

Most prior papers on DVFS performance and power prediction [5–8, 10, 25, 29] address the problem above the microarchitectural level and do not explore hardware modification. Hence, these approaches can only use already existing hardware performance counters as inputs to their performance and power models. These counters were not designed to predict the performance impact of DVFS and thus do not work well for that purpose. Hence, these papers resort to statistical [5–8, 25] and machine learning [10, 29] techniques.

In contrast, we design new hardware counters with the explicit goal of aiding DVFS performance prediction. This approach was introduced by leading loads [12, 20, 34] and stall time [12, 20] proposals and extended to power prediction by Spiliopoulos et al. [38].

The tradeoff between these two general approaches is as follows: statistical and machine learning techniques are easier to apply to complex prediction scenarios (e.g., per-core DVFS); however, our approach of designing new hardware counters builds on an understanding of the underlying microarchitectural effects that ensures robust predictions even for applications never seen before.

6.2. Analytical Performance Models

Traditional analytical performance models [4, 11, 13, 14, 19, 27, 31, 37, 39] have a different purpose than the commonly used linear DVFS performance model and our limited bandwidth DVFS performance model. Specifically, traditional analytical models are used to gain insight into the performance bottlenecks of modeled architectures and drive design space exploration. These models are evaluated offline and target only a first order performance estimate. A DVFS performance model, on the other hand, is an analytical performance model evaluated at runtime by the operating system or the hardware power management unit and has to be accurate to be useful.

6.3. Phase Prediction

Phase detection and prediction mechanisms [17, 18, 36, 42] can help improve DVFS performance prediction accuracy and hence the overall utility of DVFS. Specifically, a DVFS mechanism can benefit from phase prediction by triggering re-training of the DVFS performance predictor in the beginning of each phase, and switching to the predicted optimal operating point for the rest of the phase.

7. Conclusions

We have shown that a DVFS performance predictor must be designed with an accurate model of the memory system in mind.

We demonstrated quantitatively that previously proposed DVFS performance predictors, designed with an over-simplified view of the memory system (e.g., assuming a constant access latency or disregarding prefetching), generate inaccurate performance predictions and lose out on potential energy savings. In particular, we have shown that the commonly used linear DVFS performance model breaks down in the presence of prefetching because it does not account for finite memory bandwidth.

To address these problems, we have 1) developed the limited bandwidth DVFS performance model that takes memory bandwidth into account, and 2) proposed CRIT+BW, a low cost mechanism that accurately predicts the performance impact of frequency scaling in the presence of a realistic memory system, realizing 65% of the potential energy savings.

Acknowledgments

We thank Onur Mutlu, members of the HPS research group, our shepherd Lieven Eeckhout, and the anonymous reviewers for their comments and suggestions. We thank Rafael Ubal and other developers of Multi2Sim [41], from which we adapted the x86 functional model that drives our performance simulator. We gratefully acknowledge the support of the Cockrell Foundation and Intel Corporation.

References

[1] T. D. Burd and R. W. Brodersen, "Energy efficient CMOS microprocessor design," in *Proc. 28th Hawaii Int. Conf. Syst. Sci. (HICCS-28)*, vol. 1, Jan. 1995, pp. 288–297.

[2] M. Butler, L. Barnes, D. D. Sarma, and B. Gelinas, "Bulldozer: An approach to multithreaded compute performance," *IEEE Micro*, vol. 31, no. 2, pp. 6–15, Mar. 2011.

[3] J. A. Butts and G. S. Sohi, "A static power model for architects," in *Proc. 33rd ACM/IEEE Int. Symp. Microarchitecture (MICRO-33)*, Dec. 2000, pp. 191–201.

[4] X. E. Chen and T. M. Aamodt, "Hybrid analytical modeling of pending cache hits, data prefetching, and MSHRs," in *Proc. 41st ACM/IEEE Int. Symp. Microarchitecture (MICRO-41)*, Nov. 2008, pp. 59–70.

[5] K. Choi, R. Soma, and M. Pedram, "Fine-grained dynamic voltage and frequency scaling for precise energy and performance trade-off based on the ratio of off-chip access to on-chip computation times," in *Proc. Conf. Design, Automation, and Test in Europe (DATE 2004)*, vol. 1, Feb. 2004, pp. 4–9.

[6] G. Contreras and M. Martonosi, "Power prediction for Intel XScale processors using performance monitoring unit events," in *Proc. 2005 Int. Symp. Low Power Electron. and Design (ISLPED'05)*, Aug. 2005, pp. 221–226.

[7] M. Curtis-Maury, J. Dzierwa, C. D. Antonopoulos, and D. S. Nikolopoulos, "Online power-performance adaptation of multithreaded programs using hardware event-based prediction," in *Proc. 20th Int. Conf. Supercomputing (ICS'06)*, Cairns, Queensland, Australia, Jun. 2006, pp. 157–166.

[8] M. Curtis-Maury, A. Shah, F. Blagojevic, D. S. Nikolopoulos, B. R. de Supinski, and M. Schulz, "Prediction models for multi-dimensional power-performance optimization on many cores," in *Proc. 17th Int. Conf. Parallel Arch. and Compilation Techniques (PACT'08)*, Oct. 2008, pp. 250–259.

[9] H. David, C. Fallin, E. Gorbatov, U. R. Hanebutte, and O. Mutlu, "Memory power management via dynamic voltage/frequency scaling," in *Proc. 8th ACM Int. Conf. Autonomic Computing (ICAC 2011)*, Jun. 2011, pp. 31–40.

[10] G. Dhiman and T. S. Rosing, "Dynamic voltage frequency scaling for multi-tasking systems using online learning," in *Proc. 2007 Int. Symp. Low Power Electron. and Design (ISLPED'07)*, Aug. 2007, pp. 207–212.

[11] P. G. Emma and E. S. Davidson, "Characterization of branch and data dependencies in programs for evaluating pipeline performance," *IEEE Trans. Comput. (TOC)*, vol. C-36, no. 7, pp. 859–875, Jul. 1987.

[12] S. Eyerman and L. Eeckhout, "A counter architecture for online DVFS profitability estimation," *IEEE Trans. Comput. (TOC)*, vol. 59, pp. 1576–1583, Nov. 2010.

[13] S. Eyerman, L. Eeckhout, T. Karkhanis, and J. E. Smith, "A mechanistic performance model for superscalar out-of-order processors," *ACM Trans. Comput. Syst. (TOCS)*, vol. 27, pp. 3:1–3:37, May 2009.

[14] A. Hartstein and T. R. Puzak, "The optimum pipeline depth considering both power and performance," *ACM Trans. Arch. and Code Optimiz. (TACO)*, vol. 1, pp. 369–388, Dec. 2004.

[15] M. Horowitz, T. Indermaur, and R. Gonzalez, "Low-power digital design," in *IEEE Symp. Low Power Electron. (ISLPE'94) Digest of Tech. Papers*, Oct. 1994, pp. 8–11.

[16] *Intel 64 and IA-32 Architectures Optimization Reference Manual Version 026*, Intel Corporation, April 2012.

[17] C. Isci, G. Contreras, and M. Martonosi, "Live, runtime phase monitoring and prediction on real systems with application to dynamic power management," in *Proc. 39th ACM/IEEE Int. Symp. Microarchitecture (MICRO-39)*, Dec. 2006, pp. 359–370.

[18] C. Isci and M. Martonosi, "Phase characterization for power: Evaluating control-flow-based and event-counter-based techniques," in *Proc. 12th IEEE Int. Symp. High Perf. Comput. Arch. (HPCA-12)*, Feb. 2006, pp. 121–132.

[19] T. S. Karkhanis and J. E. Smith, "A first-order superscalar processor model," in *Proc. 31th Int. Symp. Comput. Arch. (ISCA 2004)*, Jun. 2004, pp. 338–349.

[20] G. Keramidas, V. Spiliopoulos, and S. Kaxiras, "Interval-based models for run-time DVFS orchestration in superscalar processors," in *Proc. ACM Int. Conf. Computing Frontiers (CF'10)*, May 2010, pp. 287–296.

[21] W. Kim, D. Brooks, and G.-Y. Wei, "A fully-integrated 3-level DC-DC converter for nanosecond-scale DVFS," *IEEE J. Solid-State Circuits (JSSC)*, vol. 47, no. 1, pp. 206–219, Jan. 2012.

[22] W. Kim, M. S. Gupta, G.-Y. Wei, and D. Brooks, "System level analysis of fast, per-core DVFS using on-chip switching regulators," in *Proc. 14th IEEE Int. Symp. High Perf. Comput. Arch. (HPCA-14)*, Feb. 2008, pp. 123–134.

[23] R. Kumar and G. Hinton, "A family of 45nm IA processors," in *2009 IEEE Int. Solid-State Circuits Conf. (ISSCC 2009) Digest Tech. Papers*, Feb. 2009, pp. 58–59.

[24] H. Q. Le, W. J. Starke, J. S. Fields, F. P. O'Connell, D. Q. Nguyen, B. J. Ronchetti, W. M. Sauer, E. M. Schwarz, and M. T. Vaden, "IBM POWER6 microarchitecture," *IBM J. of Research and Develop.*, vol. 51, no. 6, pp. 639–662, Nov. 2007.

[25] S. J. Lee, H.-K. Lee, and P.-C. Yew, "Runtime performance projection model for dynamic power management," in *Advances in Comput. Syst. Arch. 12th Asia-Pacific Conf. (ACSAC 2007) Proc.*, ser. Lecture Notes in Computer Science, Aug. 2007, vol. 4697, pp. 186–197.

[26] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures," in *Proc. 42nd ACM/IEEE Int. Symp. Microarchitecture (MICRO-42)*, Dec. 2009, pp. 469–480.

[27] P. Michaud, A. Seznec, and S. Jourdan, "Exploring instruction-fetch bandwidth requirement in wide-issue superscalar processors," in *Proc. 1999 Int. Conf. Parallel Arch. and Compilation Techniques (PACT'99)*, Oct. 1999, pp. 2–10.

[28] *MT41J512M4 DDR3 SDRAM Datasheet Rev. K*, Micron Technology, Inc., Apr. 2010, http://download.micron.com/pdf/datasheets/dram/ddr3/2Gb_DDR3_SDRAM.pdf.

[29] M. Moeng and R. Melhem, "Applying statistical machine learning to multicore voltage and frequency scaling," in *Proc. ACM Int. Conf. Computing Frontiers (CF'10)*, May 2010, pp. 277–286.

[30] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, "CACTI 6.0: A tool to model large caches," HP Laboratories, Tech. Rep. HPL-2009-85, Apr. 2009.

[31] D. B. Noonburg and J. P. Shen, "Theoretical modeling of superscalar processor performance," in *Proc. 27th ACM/IEEE Int. Symp. Microarchitecture (MICRO-27)*, Nov. 1994, pp. 52–62.

[32] H. Patil, R. S. Cohn, M. Charney, R. Kapoor, A. Sun, and A. Karunanidhi, "Pinpointing representative portions of large Intel Itanium programs with dynamic instrumentation," in *Proc. 37th ACM/IEEE Int. Symp. Microarchitecture (MICRO-37)*, Dec. 2004, pp. 81–92.

[33] S. Rixner, W. J. Dally, U. J. Kapasi, P. R. Mattson, and J. D. Owens, "Memory access scheduling," in *Proc. 27th Int. Symp. Comput. Arch. (ISCA 2000)*, Jun. 2000, pp. 128–138.

[34] B. Rountree, D. K. Lowenthal, M. Schulz, and B. R. de Supinski, "Practical performance prediction under dynamic voltage frequency scaling," in *2011 Int. Green Computing Conf. and Workshops (IGCC'11)*, Jul. 2011.

[35] Y. Sazeides, R. Kumar, D. M. Tullsen, and T. Constantinou, "The danger of interval-based power efficiency metrics: When worst is best," *Comp. Arch. Lett. (CAL)*, vol. 4, no. 1, Jan. 2005.

[36] T. Sherwood, S. Sair, and B. Calder, "Phase tracking and prediction," in *Proc. 30th Int. Symp. Comput. Arch. (ISCA 2003)*, San Diego, California, Jun. 2003, pp. 336–347.

[37] D. J. Sorin, V. S. Pai, S. V. Adve, M. K. Vemon, and D. A. Wood, "Analytic evaluation of shared-memory systems with ILP processors," in *Proc. 25th Int. Symp. Comput. Arch. (ISCA 1998)*, Jun. 1998, pp. 380–391.

[38] V. Spiliopoulos, S. Kaxiras, and G. Keramidas, "Green governors: A framework for continuously adaptive DVFS," in *2011 Int. Green Computing Conf. and Workshops (IGCC'11)*, Jul. 2011.

[39] E. Sprangle and D. Carmean, "Increasing processor performance by implementing deeper pipelines," in *Proc. 29th Int. Symp. Comput. Arch. (ISCA 2002)*, Jun. 2002, pp. 25–34.

[40] J. Tendler, J. S. Dodson, J. S. Fields Jr., L. Hung, and B. Sinharoy, "POWER4 system microarchitecture," *IBM J. of Research and Develop.*, vol. 46, pp. 5–25, Oct. 2001.

[41] R. Ubal, B. Jang, P. Mistry, D. Schaa, and D. Kaeli, "Multi2Sim: A simulation framework for CPU-GPU computing," in *Proc. 21st Int. Conf. Parallel Arch. and Compilation Techniques (PACT'12)*, Sep. 2012, pp. 335–344.

[42] F. Vandeputte, L. Eeckhout, and K. De Bosschere, "A detailed study on phase predictors," in *Proc. 11th Int. Euro-Par Conf. Parallel Process. (Euro-Par 2005)*, Aug. 2005, pp. 571–581.

2012 IEEE/ACM 45th Annual International Symposium on Microarchitecture

Vector Extensions for Decision Support DBMS Acceleration

Timothy Hayes Oscar Palomar Osman Unsal Adrian Cristal Mateo Valero
Barcelona Supercomputing Center
{first}.{last}@bsc.es

Abstract

Database management systems (DBMS) have become an essential tool for industry and research and are often a significant component of data centres. As a result of this criticality, efficient execution of DBMS engines has become an important area of investigation. This work takes a top-down approach to accelerating decision support systems (DSS) on x86-64 microprocessors using vector ISA extensions. In the first step, a leading DSS DBMS is analysed for potential data-level parallelism. We discuss why the existing multimedia SIMD extensions (SSE/AVX) are not suitable for capturing this parallelism and propose a complementary instruction set reminiscent of classical vector architectures. The instruction set is implemented using unintrusive modifications to a modern x86-64 microarchitecture tailored for DSS DBMS. The ISA and microarchitecture are evaluated using a cycle-accurate x86-64 microarchitectural simulator coupled with a highly-detailed memory simulator. We have found a single operator is responsible for 41% of total execution time for the TPC-H DSS benchmark. Our results show performance speedups between 1.94x and 4.56x for an implementation of this operator run with our proposed hardware modifications.

1. Introduction

Database management systems (DBMS) have become an essential tool for industry and research and are often a significant component of data centres. They can be used in a multitude of scenarios including online analytical processing, data mining, e-commerce and scientific analysis. As the amount of information to manage grows exponentially each year, there is a pressure on software and hardware developers to create data centres that can cope with the increasing requirements. At the same time, there is now also an additional demand to provide greener and more power-efficient data centres without compromising their performance [17].

Modern microprocessors often include SIMD multimedia extensions that can be used to exploit data-level parallelism (DLP) [31, 35, 10]. There has been some prior work [42, 40] using these features to accelerate database software, however these multimedia extensions tend to be very limited. They often lack the flexibility to describe non-trivial DLP found in DBMS software. There is generally an upper threshold of four to eight elements that can be operated on in parallel and code must be restructured and recompiled if this threshold should increase in the future. The extensions are typically targeted at multimedia and scientific applications with the bulk of support geared towards floating point operations. In contrast, integer code, e.g. DBMS software, has different characteristics and requirements from an ISA and this can be a limiting factor in exploiting the available parallelism. Furthermore, in typical SIMD implementations the data is expected to have high spatial locality which is typically found in multimedia applications but not as much in business-domain applications [5].

Vector architectures [13] are highly scalable and overcome the limitations imposed by superscalar designs [23] as well as SIMD multimedia extensions. They have the flexibility to represent complex DLP which the multimedia extensions lack. More importantly, they are energy-efficient [25, 24] and can be implemented in microprocessors using simple and efficient hardware [4]. DBMS architectures are known to be bottlenecked by memory access [6]; vector processors were traditionally used to tolerate long latencies to main memory and could be instrumental in optimising database software and reducing their bottlenecks. Vector architectures have traditionally been used for scientific applications abundant with floating-point code, however their applicability to business-domain, i.e. integer, applications has yet to be analysed. This paper makes this contribution as well as a study on how to optimise a vector architecture for this class of application.

Vectorwise [3] is a block at a time query engine based on MonetDB/x100 [7]. It is hardware-conscientious and highly optimised for modern superscalar microprocessors. Vectorwise researchers have identified the bottlenecks of previous database solutions and structured their own software to exploit the full capabilities of modern commodity hardware. Vectorwise algorithms are designed to take advantage of the instruction cache as well as to reduce branch misprediction penalties. Where possible, it uses blocking to optimise its algorithms for the data cache. Its functions are designed to be data-level parallel in order to expose independent loop iterations to the underlying microarchitecture. Vectorwise can store tables in a columnar fashion [9] meaning that columns of a table are stored as arrays in memory. When the algorithms access data like this, it can help to expose DLP and generate more regular memory access patterns.

Vectorwise has transformed a lot of potential DLP into instruction-level parallelism (ILP) in order to keep the processor fully utilised. While the performance is much better, optimising software by expressing DLP as ILP is not the most efficient nor scalable solution. Modern microprocessors found in servers are generally superscalar/out of order and can cope to some extent; the problem is that the hardware complexity and power consumption of finding more independent instructions this way increases quadratically [29] making this an unscalable solution and not suitable in the long term.

This work takes a top-down methodology and profiles Vectorwise looking for opportunities to use vector technology. From this, software bottlenecks caused by unscalable superscalar hardware structures are identified. A discussion is given as to why multimedia instruction sets such as SSE and AVX are insufficient to express the potential DLP in this application; consequently, new vector ISA extensions for x86-64 are proposed as a solution. These ISA extensions can be implemented using simple scalable hardware which has been evaluated using a cycle-accurate microprocessor simulator fused with a highly-detailed memory simulator. The major design decisions are compared against alternative options and evaluated quantitatively. Our experiments show that increasing the superscalar/out of order resources offers very little benefit to the existing scalar application and that the vector implementation achieves performance speedups between 1.94x and 4.56x for the most significant part of the DBMS.

1072-4451/12 $26.00 © 2012 IEEE
DOI 10.1109/MICRO.2012.24

IEEE computer society

This paper is structured as follows: Section 2 characterises the application domain and provides motivation for this work. Section 3 discusses the design and implementation of the vector extensions. Section 4 describes the experimental setup. Section 5 presents the results of various experiments related to the design space and performance. Section 6 compares and contrasts related work. Section 7 concludes this work and proposes research for the future.

2. Software Characterisation

Vectorwise v1.0 was used in conjunction with the TPC-H [39] benchmark to look for areas of interest. TPC-H is the standard benchmark to evaluate decision support systems. The benchmark defines the database tables and their relations (schemas); the values contained in the database; and the queries to be evaluated over the database. Unlike the SPEC CPU benchmark suite, it is not a set of individual applications but rather a set of queries that stress different aspects of a DSS DBMS implementation. The DBMS software has the freedom to store the database in its preferred way and evaluate the queries in a manner that it sees fit, therefore what is presented is one *particular* evaluation of Vectorwise.

Figure 1 shows the CPU time of 22 [1] queries executed on a database of 100 GB on an Intel Nehalem system with 16 GB of DDR3-1333 memory. The results show that a significant amount of time is spent in the hash join operation. If all the queries are evaluated together and their total execution time is accumulated, the hash joins account for 61% of this time. This has motivated us to focus our work on this operation, in particular the probe phase of the join which constitutes 67% of the time spent in hash joins and 41% of the total execution time. Although this work focuses on a particular algorithm, we expect many of our findings to be applicable to other aspects of the DBMS. This is due to the way that Vectorwise has been implemented as a column-oriented DBMS: blocking their algorithms and exposing DLP in their functions. The hash join algorithm of a row-oriented DBMS could be vectorised in a similar fashion, however this would entail transforming several unit-stride memory accesses to strided ones.

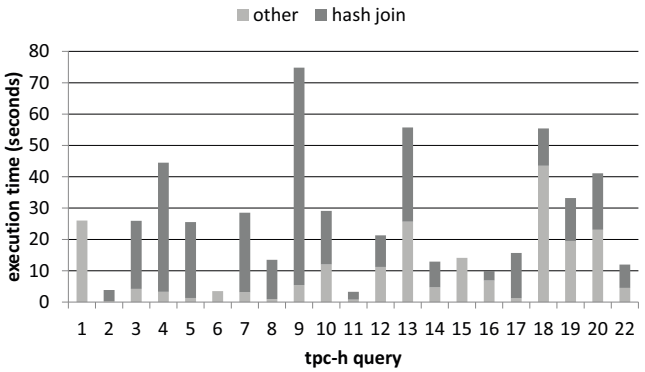


Figure 1: TPC-H 100 GB Breakdown

2.1. Hash Join Probing

Vectorwise's hash join probe works in several stages and rounds, illustrated in Figure 2. The ultimate goal is to match and join the keys from the left hand side (LHS) with those on the right hand side (RHS). A portion of the LHS, called a block, is processed together as a compromise between cache locality and function call overhead. First, the keys of the block are hashed to create indices into the hash table (HT) structure. The corresponding value in the HT is in turn an index into RHS. The values must be retrieved and compared against the keys from the LHS for matches. It is possible to hash to the same index leading to bucket collisions which must be handled appropriately; it can be seen in the example that keys 133 and 379 cause a collision. If the match fails, an auxiliary structure named collisions is checked. If there was a bucket collision, the entry will chain to the next colliding key otherwise the entry is empty implying there are no collisions and potentially no possible matches. For a more detailed explanation of the Vectorwise hash join implementation the reader is referred to [43, 37].

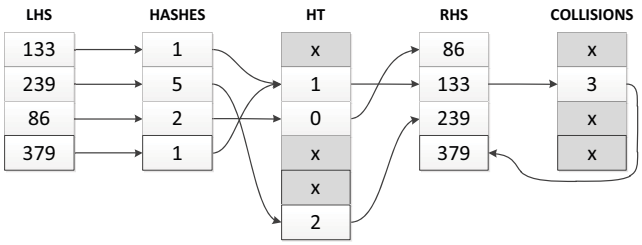


Figure 2: Hash Join Probe Phase

Vectorwise's hash probing also uses blocking to expose more independent operations to the compiler/microprocessor and amortise function-call overheads. Unlike other operations, it is harder to achieve data locality and cache residency. It is necessary that the hash table is fully built before it can be probed thus the entire right hand side must be evaluated before any work on the left hand side can begin. If the right hand side has many values, the hash table becomes large and reduces the opportunity to effectively use the cache. For example, the structures used in one hash join found in query 9 amount to 86 MB which is queried by over 600 M keys.

Each row of the join is independent with respect to another thus making the algorithm potentially data-level parallel. The structures are stored contiguously as arrays in memory, however due to the random access nature of the algorithm indexed memory operations are necessary i.e. gather/scatter. Mask registers can be used to optimise many of the operations. When keys from the LHS are matched with the RHS, they can be masked out of subsequent rounds and needn't proceed with further checks in the collisions table. A reordering instruction, e.g. compress, coupled with a programmable vector length can be used to reduce the amount of time resolving bucket collisions. Each round will reduce the number of candidates to check for collisions thus decreasing the time spent per pass.

3. Design and Implementation

x86-64 was chosen as the base ISA to build upon for several reasons. It is the leading ISA in the server market having roughly 60% of the market share. It is a very universal ISA with mature optimising compilers and toolchains. Vectorwise, although not exclusively written for x86-64, has several optimisations made for x86-64 and the Intel Xeon 5500 series [18]. x86-64 is also a large improvement over the archaic IA32 ISA; many improvements have been made e.g. the number of general purpose registers is doubled, and some legacy features have been removed such as memory segmenting.

[1] Query 21 was not run due to a larger memory requirement than what was available.

The baseline microarchitecture is not taken from any specific incarnation of x86-64. Instead, the features available from PTLsim [41], a cycle-accurate x86-64 simulator, are used. PTLsim models an aggressive superscalar out of order microarchitecture with instruction to μop translation; multistage pipelines; speculation and recovery; and a multi-tiered cache hierarchy. Values from Intel's Nehalem microarchitecture [19] are used as configuration parameters.

3.1. Instruction Set Proposal

We have designed the vector instruction set based on our analysis of the hash join probe algorithm discussed in Section 2. Instructions are classified and listed in Table 1.

class	instructions
memory	unit stride, indexed, prefetching
arithmetic	integer mul, add, sub
bitwise logical	and, or, not, xor, shift
comparison	not equal
initialisation	set all, clear all, iota
mask	set, clear, and, or, not, xor, popcount
permutative	compress
vector length	set, set MVL, get
memory fence	scalar-vector, vector-scalar, vector-vector

Table 1: Vector Instruction Listing

The ISA offers eight vector registers, although the vectorised functions utilise a maximum of six of these. Each vector register stores the same number of elements defined by the maximum vector length (MVL) constant. The actual number of elements that a given instruction operates on depends on the value of the vector length (VL) control register which is managed explicitly by get/set instructions. The ISA also provides an instruction that sets the VL to the MVL. Retrieving the MVL at runtime allows for transparent scaling of the microarchitecture; if the vectorised functions are written using loop strip mining, they may be able to take advantage of larger vector register lengths without the need of rewriting nor recompiling.

The ISA includes vectorised integer arithmetic and bitwise logical instructions. One of the source operands must be a vector register and the other may be another vector register or a scalar register. There is a class of initialisation instructions which can set a specified vector register to a scalar value. A very useful variant of this, known as iota [36], is also included which generates a vector of consecutive integers starting from a specified value; this is useful for dynamically generating indices into the hash join structures.

The majority of instructions can take an optional vector mask specified by one of the four available mask registers. Vector masks are updated in three ways: 1) with initialisation instructions (set all/clear all); 2) with comparison instructions that write their boolean results to mask registers; and 3) with mask-mask logical instructions. The ISA also includes a position manipulation instruction called compress; this condenses non-masked elements from one vector register contiguously into another vector register. To complement this, a mask population instruction is included which counts the number of set bits in a mask register. These instructions are useful for eliminating rows that have no potential entry in the hash table as well as shortening the vector length when checking candidate matches.

Both unit-stride and indexed memory access instructions are needed. Strided memory instructions are not necessary and are omitted from the ISA, however these would be useful if the baseline DBMS were row-oriented instead of column-oriented. Individual elements that comprise a vector load operation are assumed to be independent of one another; stores/scatters always write to unique locations and scatters with conflicting indices are left semantically undefined.

A major decision was choosing a weak consistency model between the vector memory instructions themselves. This way the execution order of vector loads/stores is not deterministic and ordering must be achieved through explicit fence instructions. Although this puts more pressure on the programmer, the vast majority of memory accesses are independent of one another and this allows for more aggressive scheduling in the microarchitecture as well as reduced hardware complexity due to the absence of memory aliasing checks. Weak ordering guarantees can also be found in Cray's NV-2 ISA [2].

The proposed ISA is reminiscent of classic vector ISAs used in supercomputers. This type of ISA is already known to be useful for scientific computing and multimedia processing [4, 13] as well as other areas [11] thus broadening the scope of applicability of our work. Additionally there has been related work [15, 27, 42] that shows DLP opportunities in DBMS software beyond hash join that could also be exploited with an ISA like this.

3.2. Design Decisions

3.2.1. Out of Order Execution One of the biggest design decisions made was to allow vector instructions to issue out of order. The work of [12] showed that by using register renaming and out of order execution additional performance can be gained. An out of order execution engine can begin memory operations early and utilise the memory ports much more efficiently hence hiding long memory latencies. Vectors are already tolerant of long memory latencies in their own right; combining them with an out of order core can further enhance this quality.

There are also drawbacks to an out of order microarchitecture. The structures used to achieve out of order execution don't scale well and are very power-hungry [29]. Fortunately a single vector instruction can represent a lot of work and reduce the need to scale these structures more than what already exists in current commodity out of order microprocessors. The decision to allow issuing vector instructions out of order affects many of the subsequent design decisions. In Section 5.1.1 the benefit of using the out of order mechanism is evaluated against a simpler decoupled design.

3.2.2. Cache Integration The block at a time processing technique used by Vectorwise is very conscientious of the cache hierarchy. As mentioned in Section 2, large structures like the hash tables often have trouble fitting in the cache hierarchy however many other structures can reside there comfortably. Of particular importance are the blocks which flow through various data operators storing intermediate results in cache-resident arrays. For this reason it is highly desirable to take advantage of the cache hierarchy when possible.

A solution to integrating vector support into an existing superscalar processor was proposed in [30]. Part of the work was integrating the vector units with the cache hierarchy. Their novel solution involved bypassing the level 1 data (L1D) cache altogether and going directly to the level 2 (L2) cache. The main motivation behind this was that adding the logic necessary to support vector loads at the L1D cache could compromise its access time as seen by the scalar units.

The L2 cache is always larger than the L1D cache so there is also the additional benefit of potentially having a larger working set. Because unit-stride loads and cache lines match quite well,

this solution could pull many elements from the cache at once and hide the additional latency incurred by the L2. Accessing the L2 directly introduces potential coherency problems with the L1D cache, a simple approach described in [30] is used in our design to resolve this. This involves adding an extra bit to each line to mark if its data is exclusively owned by the scalar units or the vector units. L1D bypassing was later used in Tarantula [11] which had a 4 MB banked L2 cache directly accessible by its vector memory units. For these reasons, this design is used in our experiments. In Section 5.1.3 the L2 bypassing is evaluated against an alternative approach that accesses the L1D cache directly instead.

3.3. Microarchitecture Implementation

It was desirable to reuse as much as possible from the base microarchitecture so the additions necessary to implement the vector ISA would be minimal. One of the key design decisions made was integrating the vector units into the core itself. The decode units had to be modified to incorporate the new vector ISA. The changes were minimal as the new instructions all have a fixed length and begin with the same prefix. The register rename tables had to be changed to accommodate the new vector/mask/vector length architectural registers. Two new physical register files were added in order to support vectors and masks; the vector length register is part of the existing integer register file.

New functional units and issue queues have been added. The original issue queues can handle up to four operands which are sufficient for the existing x86-64 instructions; the new vector instructions needed two extra operands on top of the existing four. This is due to two reasons: 1) the vector length register is allowed to be renamed and thus it is necessary to have it as an operand in the issue queue. 2) The destination register is also a source register. It is possible for the vector instruction to overwrite part, but not all, of the its destination register. This occurs when the VL is shorter than the MVL or the instruction masks out operations on some of the vector's elements. The scalar issue queues using four operands can coexist with the vector issue queues using six operands.

Misspeculation recovery piggybacks on the existing infrastructure of the out of order core. Vector registers are renamed using the same mechanism as scalar register and on a branch misprediction the register rename table is restored to a stable state before fetching from the correct path. This is similar to the approach used in [12]. Vector stores can generate their addresses when issued but don't modify the memory state until they are the oldest instruction ready to commit.

3.3.1. Fence Mechanism For vector memory fences it is necessary to mark the reorder buffer entry of the youngest store instruction, this way when a fence instruction is decoded it can be made dependent on this store provided it has not already committed. The fence should only proceed when the store instruction writes to the memory state, therefore it would be necessary to add a path from the commit stage of the pipeline to the issue queues. To simplify this, the fences are given their own issue queue which has a dedicated path from the commit logic. Memory instructions younger than the fence will in turn use this fence as a dependency. In Section 5.1.1 the custom fence mechanism is compared against a completely fenceless approach and in Section 5.1.2 against a more a naïve implementation.

3.3.2. Vector Memory Request File Vector memory requests had to be handled differently from all other operations. Scalar memory loads can be executed out of order but to achieve this a complex associative hardware structure called the load/store queue (LSQ) must be used. The LSQ detects memory aliases, i.e. loads and stores that go to the same address and may have incorrect behaviour when issued out of order. Vector memory operations are known to be data independent at the element level and in most cases are also data independent with respect to one another; vector memory aliasing is handled explicitly using fence instructions and therefore does not need transparent resolution in hardware. Using the LSQ would limit the number of in-flight vector memory operations in the microprocessor; additionally, such a structure would present a huge design problem for handling indexed memory operations which may have irregular access patterns.

It is also important to take advantage of the regular patterns found in vector memory operations. Unit-stride loads/stores access consecutive locations in memory and thus have a lot of spatial locality; it is therefore preferable to work with whole cache lines when possible. The LSQ as it exists does not take advantage of this locality and each entry refers to a single scalar value. For indexed memory operations with less spatial locality, it is important to reduce the penalties that may be incurred from transferring unnecessary data.

A structure called the Vector Memory Request File (VMRF) has been designed to manage vector memory instructions while avoiding the complex associative hardware found in the LSQ. Every vector memory instruction takes an entry in this structure. The VMRF translates vector memory operations into L2 cache accesses at the line level. It tracks all accesses that comprise a single vector memory instruction until these accesses complete and keeps enough metadata about the vector memory instructions to be able to recover from misspeculation. It also ensures that the writes to memory of a vector store will not start until it is the oldest instruction in the pipeline and can commit safely.

The VMRF has two structures to service vector load operations. The Load Table (LT) and the Cache Line Table (CLT). When a vector load (unit-stride or indexed) is issued, it takes a single entry in the LT. As the address generation unit generates cache line addresses, these are given individual entries in the CLT. An entry also includes metadata such as the physical registers used as well as a bit-field that denotes which bytes of the register/cache line are relevant. Each cycle the CLT selects a pending entry (round robin) to be sent to the L2. When the request completes, the CLT is freed and a bitmap inside the corresponding LT is updated. When the bitmap is all set, the LT can notify the reorder buffer entry of the load and be freed. We use indices to access each table and thus don't need associativity. The VMRF does not track RAWs and relies on explicit fence instructions to avoid hazards.

Two buses were added that connect the physical register file to the L2 cache: one for load requests and another for store requests. As an optimisation, the structure can handle partial cache line transfers. The cache line is broken into discrete sectors: the size of an L2 cache line divided by the width of the bus. To save bus cycles, only necessary sectors need to be sent. The VMRF tracks which bytes within the cache line are actually needed. Indexed operations benefit from this especially if the number of required bytes per cache line is small. Reorder buffer entries must contain an identifier into the VMRF; both loads and stores require this information when misspeculation recovery occurs. In these cases, the entry in the VMRF must be annulled and recycled.

It is quite controversial to add index memory instructions to an out of order microprocessor. There are often reservations about doing this, especially about compromising the latency of scalar memory instruc-

tion which could affect the performance of existing non-vectorised applications. We have made two important design choices to circumvent this from happening: 1) the LSQ is untouched by vector instructions and more importantly avoids the complexities that would arise when detecting aliasing between indexed memory operations. 2) We leave the interface between the functional units and the L1D cache untouched and instead bypass this structure and access the L2 cache directly instead.

4. Experimental Setup

4.1. Simulators

Experiments have been evaluated using PTLsim [41], a cycle-accurate x86-64 simulator. The experiments were conducted using the *classic* mode PTLsim i.e. where system calls are emulated. The simulator has been extended extensively to incorporate the new instruction set and additional microarchitectural changes. PTLsim uses a fixed latency memory model by default which does not model bandwidth and contention issues at all. It was felt that for a memory-intensive algorithm like hash join it is paramount to model the memory accurately. Consequently we have integrated DRAMSim2 [32], a cycle accurate memory system simulator, into PTLsim and replaced the default memory model. This also allows us to experiment with multiple memory controllers. Our results show large discrepancies between the default simplified model and more accurate model using DRAMSim2.

4.2. Default Parameters

This section lists the parameters of the baseline setup. In all the experiments that follow these parameters are used unless explicitly stated otherwise. The parameters of the scalar baseline are based on the Intel Nehalem microarchitecture [19], the same used to profile Vectorwise in Section 2.

parameter	value
fetch width	4
fetch queue	28
frontend width	4
frontend stages	7
dispatch width	4
writeback width	4
commit width	4
issue width per cluster/total	1/6
reorder buffer	128
issue queue	8 (per cluster)
load queue	48
store queue	32
outstanding l1d misses	10
outstanding l2 misses	16

Table 2: Superscalar and Out of Order Parameters

Table 2 lists the superscalar parameters as well as the sizes of various structures in the microarchitecture. Here *frontend* refers to decoding, renaming and structure allocation. In Nehalem the equivalent to an issue queue is the reservation station, a single structure with 36 entries shared by all the clusters. It is not possible to model clusters as well as a shared issue queue in PTLsim. To solve this, the reservation station is divided into six parts (there are six clusters) and each is given an extra two entries to compensate. We have measured the impact of larger issue queues with up to 32 entries per cluster and have found a difference in performance of only 3%. There is one cluster for loads; two clusters for stores; and three general purpose clusters. All functional units are fully pipelined as in Nehalem. For further details, the reader is referred to [19].

cache level	size	latency	line size	ways	sets
l1 instruction	32 KB	1	64	4	128
l1 data	32 KB	4	64	8	64
l2 unified	256 KB	10	64	8	512

Table 3: Cache Hierarchy Parameters

Table 3 shows the cache parameters. The hierarchy is inclusive and write through with respect to L1D $\rightarrow$ L2, but writeback with respect to L2 $\rightarrow$ memory. Although the Nehalem has a larger shared L3 cache, this is not included as the effects of a multiple cores are not modelled here.

parameter	value
type	DDR3-1333
clock	1.5 ns
transaction queue	64
command queue	256
policy	open page
row accesses	8
data bus	64 bits (JEDEC standard)
queue	per rank per bank
scheme	row:rank:bank:chan:col:burst
scheduling policy	rank then bank
banks	8
ranks	4
rows	32768
columns	2048
device width	4
burst length	64 bytes

Table 4: Memory System Parameters

Table 4 contains the parameters of the memory system. We model a memory system with both one and two memory controllers. The memory modules are DDR3-1333 (cycle time of 1.5 ns) and the CPU frequency is taken to be 2.67 GHz therefore the memory controllers are clocked once every four CPU cycles. The burst length is taken as 64 bytes as this coincides with the line sizes of the cache hierarchy. An open page policy is used, however using a closed page policy results in only marginally less overall performance. This may be important when considering energy consumption where the closed page policy can be more beneficial [21].

Table 5 shows the default configuration of the vector parameters. The number of physical vector registers has been made twice the amount of architectural vector registers. This is based on [12] that states for register renaming to be effective, there should be at minimum twice as many physical registers to architectural registers. We have noted that eight architectural vector registers is excessive for our workload. Six architectural registers would be more than sufficient meaning that the physical register file could be reduced to 12 entries. All experiments presented in the next section use a single vector lane (i.e. parallel lockedstepped pipelines used to operate on elements

parameter	value
maximum vector length (MVL)	64
physical vector registers	16
physical mask registers	8
vector load requests	12
vector store requests	8
L2 → vector register bus width	32 bytes
vector cache line requests	256
maximum datatype width	64 bits

Table 5: Vector Parameters

within a single vector instruction). Our experiments have shown that adding more lanes improves performance marginally as the algorithm is dominated by memory requests. Additionally, chaining (i.e. allowing some vector instructions to issue as soon as the first elements of an input operand are ready rather than waiting for all of the elements to be calculated first) does not exhibit any significant performance changes and has thus been disabled.

The L2 cache to vector register file bus width was chosen to be 32 bytes; the same as the bus width that connects the L1D cache to the L2 cache in Nehalem. The number of cache lines requests tracked by the VMRF has been made 256 entries; this number was chosen to allow at least two indexed memory operations in flight when the MVL is 128, however this structure can be reduced when the MVL is shorter. We measure that reducing the VMRF to 64 entries decreases performance at most by 0.5%. Three additional clusters have been added for vector support. One cluster is designated for vector memory operations and the remaining two can execute non-memory vector instructions. Each cluster requires one write and two read ports to the vector register file. A vector instruction must complete fully before another one can occupy the same functional unit.

4.3. Workload

The DBMS has been configured to use blocks of 1,024 elements. The relevant functions have been vectorised by hand using the proposed ISA extensions. The vectorised functions retain their semantics and minimal transformations are needed. This way a comparison against the original/scalar implementation is fair and representative. Both the scalar and vector versions have been compiled with GCC using the best measured optimisation level.

The proposed changes are evaluated using a partial run of a hash join probe found in query 9 of TPC-H labelled `tpch`. This join uses two keys to query a hash table of 32 MB, a conflict table of 18 MB and RHS indices of 36 MB (total of 86 MB). The LHS input is originally 600 M rows but reduced to 12 M in order to shorten the simulation times. This will be just as representative as the LHS input data is distributed in such a way that the selectivity of the query will remain fixed whether the LHS is complete or partial. We have observed that useful performance metrics such as the instructions per cycle and the cache miss ratio are invariant to the size of the LHS input.

Four extra synthetic datasets have been added in order to evaluate the algorithm in different scenarios. `l1r` and `l2r` are built such that the hash table, conflict table and RHS indices can be resident in the L1D and L2 caches respectively. The LHS input does not have temporal locality and is therefore not considered in this measurement. `2mb` is eight times the size of `l2r` thus allowing for a mixture of cache hits and misses in the experiment. `huge` has structures of an equal size to `tpch` but with a different selectivity (the LHS finds more matches in the RHS) leading to more work. In order to compare against `tpch` the LHS input is fixed at 12 M rows for all the datasets.

5. Results

This section presents the results of several key experiments used to evaluate the impact our vector extensions have on the hash join probe algorithm. We have chosen to present our results in terms of processor cycles or speedup in order to fairly compare the purely scalar simulations with the vector simulations. CPI/IPC metrics are not used because they don't translate well to something comparable with the baseline architecture. A single vector instruction is not comparable to a single scalar instruction. For example, `l1r` run with the scalar baseline commits 1,163 M instructions whereas the vectorised version run with a MVL of 64 commits only 51 M instructions of which 16 M are vector instructions. Even treating a single vector instruction as the number of scalar instructions equivalent to the MVL is not fair due to 1) the decrease in structural pressure and 2) the reduction of related bookkeeping scalar instructions such as loop constructs and conditionals.

5.1. Design Exploration

Figure 3 displays the results of various experiments related to the design and implementation space. These experiments are run with the vectorised binary with the parameters described in Section 4. *default* refers to the default configuration with out of order logic, customised fences and L1D cache bypassing. *decoupled* restricts the out of order capabilities of the vector issue queues and permits only the oldest instruction, i.e. at the head, in each queue to issue. The scalar issue queues are still fully out of order to isolate the impact of dynamic scheduling applied to vector instructions. Vector memory instructions can still issue speculatively thus the explicit fences are still necessary. *fenceless* restricts the out of order capabilities in the same way as *decoupled* and additionally limits the in-flight vector memory instructions to remove the necessity of fences entirely. *flush* replaces the custom fence mechanism with a less efficient alternative. *l1* forces vector memory instructions to go directly to the L1D cache instead of the default bypass mechanism. For clarity these results are presented with absolute numbers using simulated processor cycles of execution; accordingly the lower the value the better the result.

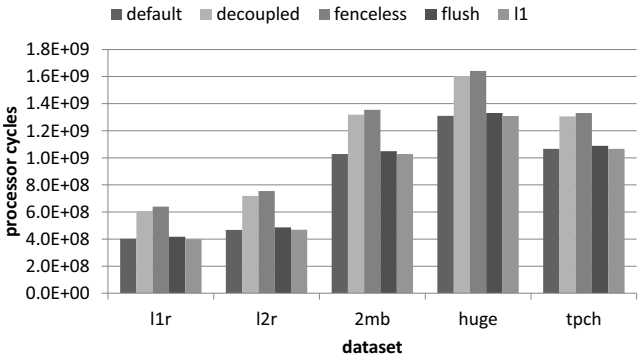


Figure 3: Design Space Exploration

5.1.1. Out of Order Logic Here the benefit of the out of order vector issue queues is quantified. It is immediately apparent that the out of order capabilities of *default* outperform both of the more restricted

configurations: *decoupled* and *fenceless*. Taking into account all the datasets, *default* takes 75% of the number of cycles of *decoupled* and 72% of *fenceless* or alternatively gives 1.34x and 1.39x speedups respectively. Although restricting the scheduling policy simplifies the processor, the out of order logic allows the instruction stream to execute more aggressively and start independent vector instructions earlier thus utilising the available execution units more efficiently. It must be restated that out of order support does add a lot of complexity to the microarchitecture however the proposed design attempts to piggyback on the existing out of order support and reuse as much as possible from the scalar core.

5.1.2. Custom Fences Next the benefit of the custom fence logic is evaluated. *flush* uses PTLsim's internal mechanism for creating a true instruction stream barrier. This is typically used to service hardware assists, i.e. special x86-64 instructions that cannot be decoded into μops. When the decoder encounters one of these instructions the processor stops fetching new instructions, it waits for the pipeline to drain completely (thus establishing a correct hardware state) before servicing the instruction. The semantics of our fence instructions are changed to evoke this behaviour.

The difference in performance is not as significant as we originally anticipated. On average the custom fence mechanism outperforms a full pipeline flush by 1.03x. This may be explained by the fact that each phase of the algorithm typically ends by storing the same data that will be used by a subsequent phase; this implies each phase begins with load instructions and finishes with store instructions. These loads are generally at the head of a dependency chain; if they stall then the rest of the instruction stream is stalled as well. The true difference between *flush* and *default* in this scenario is that the latter allows instructions to decode and dispatch but the former does not, however in both cases nothing will be able to issue until the fence has committed. The penalty of using *flush* may become more apparent with a longer frontend pipeline.

5.1.3. Level 1 Data Cache Bypass Here the benefit of L1D cache bypassing is measured. For the *l1* design, the VMRF goes directly to the L1D cache instead of the L2 cache in *default*. The benefit is that any data resident in the L1D cache can be transferred to a vector register in fewer cycles; the disadvantage is that on a cache miss an extra cycle is needed to request the missing data from the L2 cache. It also must be stated that in this evaluation the scalar access time to the L1D cache remains unchanged. A vector access takes the same latency as a scalar one although transfers to the vector register file are still restricted to 32 bytes per cycle. The reality is that a direct vector access to the L1D cache could compromise the overall cycle time as discussed in [30]; the *l1* design is thus more optimistic than it could be.

The results show that on average *l1* has a negligible speedup over *default*. This can be explained by the fact that, internally, the VMRF still needs to generate the same number of requests to the caches. When accessing the L2 cache, these are pipelined and the penalty is amortised. Additionally the majority of the workloads don't comfortably fit in the L1D cache which in turn has less outstanding misses available than its L2 parent. It can be concluded that going to the L2 cache in lieu of the L1D cache adds very little penalty and ensures that the existing scalar performance is not compromised.

5.2. Vector Scalability

It is desirable to have a large average vector length (AVL) as it is directly related to the scalability of the code. Figure 4 shows the trend of the AVL of `tpch` when increasing the MVL. The horizontal axis varies the MVL i.e. the number of elements that can be contained in a vector register. The vertical axis shows the AVL normalised to the MVL; this way the scalability of the algorithm is clearer.

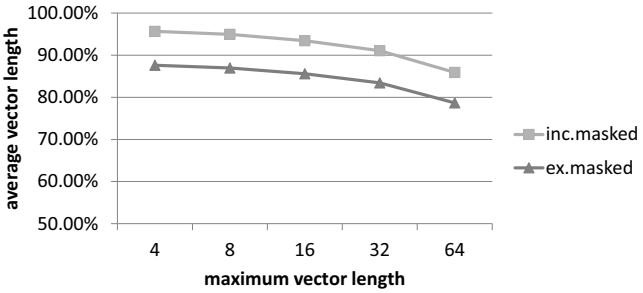


Figure 4: Average Vector Length Using `tpch`

The AVL is calculated by dividing the total number of elements processed by vector operations divided by the total number of vector instructions. It has two variants, the AVL including masked out elements (*inc.masked*) and another excluding these (*ex.masked*). The results show that the AVL degrades gradually with larger MVLs, however not too rapidly, therefore it is worth experimenting with large MVLs such as 64.

Figure 5 shows the performance benefits of increasing the MVL for all of the datasets. The horizontal axis doubles the MVL at each increment and is shown on a logarithmic scale. The vertical axis shows the speedup of the vectorised code over the scalar equivalent. The speedup shown for each line is relative to the scalar baseline run with that particular dataset. The vector solution is particularly good at describing the independence of individual operations, expressing them in a compact manner and scheduling them back to back. The scalar implementation still suffers from inter-instruction dependencies and stifles the potential of faster scheduling, especially with respect to memory instructions. The vector implementation reduces the number of instructions fetched, decoded, renamed, issued and committed as well as their occupancy in structures such as the fetch queue, issue queues and reorder buffer.

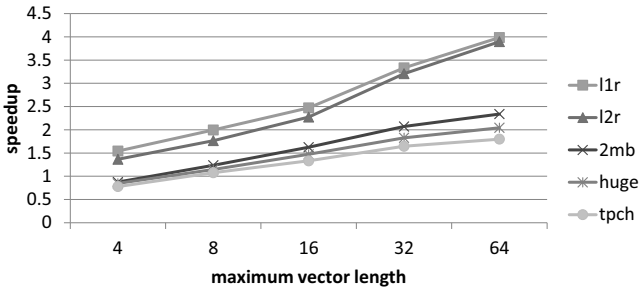


Figure 5: Vector Scalability

`l1r` and `l2r`, the two cache-resident datasets, see the greatest benefit of a larger MVL with speedups of ~4x in the best case. `2mb`, `huge` and `tpch`, the noncache-resident datasets, also scale with the MVL albeit more slowly. When increasing the MVL from 32 elements to 64, the average performance increase of the cache-resident datasets is 1.2x whereas for the noncache-resident datasets it is 1.11x.

Depending on the expected input size, it may be more economical to have a smaller MVL.

It is interesting to note that the noncache-resident datasets run with a MVL of four perform worse than their scalar equivalents. It is known in vector research that there is a break-even vector length below which the vectorised operation needs more time than the equivalent scalar operation [33]. This can be explained by the penalty of going directly to the L2 cache which isn't yet amortised with such a low MVL.

Increasing the size of the MVL is very significant to the performance speedups even while keeping the number of vector lanes the same. For the `l1r` experiments a MVL of four yields a speedup of 1.5x over the scalar baseline, however changing the MVL to 64 increases the performance to 4x with an equivalent number of lanes. We observe that the number of cycles reduced in `l1r` with a MVL of 64 is very close to the number of cycles that the frontend cannot dispatch due to full clusters with a MVL of four. With a larger MVL there are fewer instructions and although an instruction with a MVL of 64 has a higher latency than an instruction with a MVL of 8, the aggregate time is lower due to the vector startup penalty being paid less frequently. In general there are less structural hazards leading to higher throughput.

5.3. Memory Controller Saturation

Figure 6 shows the results of the vectorised code run with different memory configurations. Here the `tpch` dataset is measured and the vectorised algorithm's speedup is shown relative to its scalar baseline. The diagram plots three trends: *inf. bw*, *mc1* and *mc2*. *inf. bw* shows the relative performance when using PTLsim's default fixed latency memory model. This is configured at 150 cycles per memory request which is the average load memory latency of the scalar version reported by DRAMSim2. This model is considered to be infinite in bandwidth as it does not model contention, variable latencies, bandwidth nor any of the quirks found in a realistic memory system.

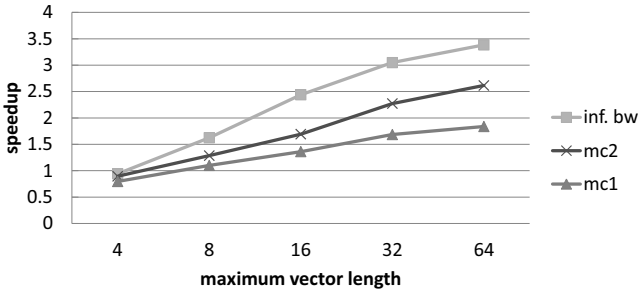


Figure 6: Impact of Available Bandwidth Using `tpch`

Comparatively, *mc1* shows the same experiment run using an accurate DRAM model. For a MVL of 64 *mc1* reports 1.84x performance over the scalar baseline whereas *inf. bw* discloses 3.4x, a massive discrepancy. The vector unit had been saturating the memory system with requests which in turn did not have enough bandwidth to sustain the requirements. *mc2* shows the same experiment run with an additional memory controller used to increase the available bandwidth; although it still falls short of the controversial *inf. bw* trend, it allows the speedup to increase to 2.61x.

It must be made clear that the effectiveness of the vector support comes from its ability to saturate the memory controller with requests. Figure 7 shows the effects of increasing the maximum number of outstanding last level cache misses by increasing the number of miss status holding registers (MSHRs). The results are shown as the speedup over the scalar version with one memory controller and the default number of MSHRs using the `tpch` dataset. Here *s-* and *v-* refer to the scalar and vector experiments respectively.

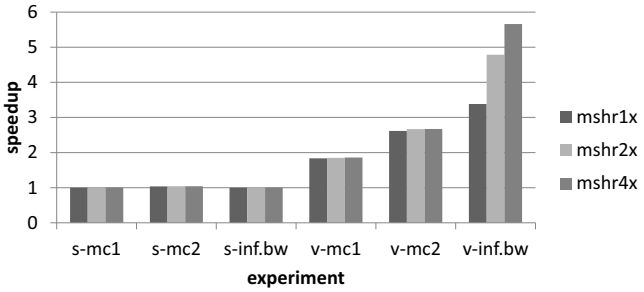


Figure 7: Memory Bandwidth and MSHRs Variation Using `tpch`

It can be seen that the scalar version does not show any speedup with the addition of a second memory controller nor with the infinite bandwidth memory model; *s-mc1*, *s-mc2* and *s-inf.bw* do not exceed 1.0x even when extra MSHRs are offered. The scalar version of the algorithm may be able take advantage of the available bandwidth in the system if it were able to generate its requests quicker. We have found that the scalar code uses about 3.5 GB/s of effective bandwidth out of DDR3-1333's maximum theoretical of 10 GB/s.

In contrast, it can be seen there is no performance gain for the vector version when the number of MSHRs is increased, but for different reasons. Clearly the vector version can generate a sufficient number of requests to main memory otherwise there would be no speedup shown for the infinite bandwidth memory model *v-inf.bw*. The simulations that model detailed memory controllers don't exhibit additional speedup with more MSHRs because the memory resources are already strained. The vector code with a single memory controller achieves 6.2 GB/s of effective bandwidth, however it is normal for an application to peak at around 70% of the maximum theoretical bandwidth. When operating close to the application's maximum sustainable bandwidth, latencies tend to increase exponentially, this is described in detail in [21, 38]. Seeing this plateau of available bandwidth motivated us to experiment with an additional memory controller.

5.4. Scalar Scalability

To illustrate that vectors are an appropriate solution to this problem, Figure 8 shows the effects of increasing the superscalar/out of order structures listed in Table 6. All the datasets are measured on the four configurations and the results are presented as the speedup over the baseline *ss1* configuration. It must stated that it is extremely unrealistic to presume these parameters can be scaled in such a way, however it makes for an interesting experiment and exposes the limitations of a purely scalar approach.

It can be seen that doubling the parameters once increases the performance 1.16x on average, which is a minor gain considering the resources required to achieve this speedup. Increasing the hardware structures 8x will increase the performance between 1.22x (for `l1r`) and 1.4x (for `2mb` and `huge`). This means in the best case, a huge out of order superscalar design can yield an extra 40% of benefit at

parameter	ss1	ss2	ss4	ss8
fetch queue	28	56	112	224
load queue	48	96	192	384
store queue	32	64	128	256
reorder buffer	128	256	512	1024
issue queue	48	96	192	384
outstanding l1d	10	20	40	80
outstanding l2	16	32	64	128
fetch width	4	8	16	32
frontend width	4	8	16	32
dispatch width	4	8	16	32
writeback width	4	8	16	32
commit width	4	8	16	32

Table 6: Scaled Superscalar and Out of Order Parameters

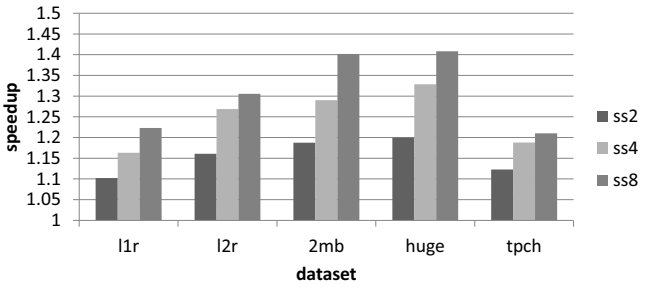


Figure 8: Scalar Scalability

best whereas the simpler vector model can increase performance past 400%, an order of magnitude in difference.

5.5. Software Prefetching

The work of [8] showed the potential of increasing hash join performance using software prefetching. This is a particularly appealing solution as it takes advantage of existing hardware found in commodity processors. x86-64, the baseline ISA in these experiments, includes a set of software prefetching instructions defined by the SSE standard. The scalar version of the hash join was modified using the group prefetching technique described in [8]. The vector code was also modified to use prefetching. To achieve this, a new instruction was added that allows the indexed load (gather) operation to be prefetched into the L2 cache.

Figure 9 shows the results of the experiments made with each dataset. *s-pre* is the scalar code with software prefetching enabled. *v-no-pre* is the default vectorised code without software prefetching modifications. *v-pre* is the vectorised code with software prefetching additions. All experiments are presented as the relative speedup over the scalar baseline without software prefetching for the particular dataset. It can be seen that *s-pre* improves the performance of the algorithm run with all datasets with an average speedup of 1.34x; this is quite a good speedup considering it requires no additional hardware. In contrast, *v-no-pre* achieves much better speedups (between 1.8x for `tpch` and 4.0x for `l1r` and `l2r`) hence showing that the vector approach has higher returns than a scalar version with prefetching. *v-pre* shows that for all of the datasets, prefetching combined with the vectorised code pushes the performance even more. In the case of `l2r`, performance exceeds 4.5x which helps illustrate that the performance gains of software prefetching can be complementary to the proposed vector additions. It is important to note that for `l1r` and `l2r` it is only the RHS that is cache-resident; the LHS is larger than the cache and prefetching helps reduce the effect of cold misses.

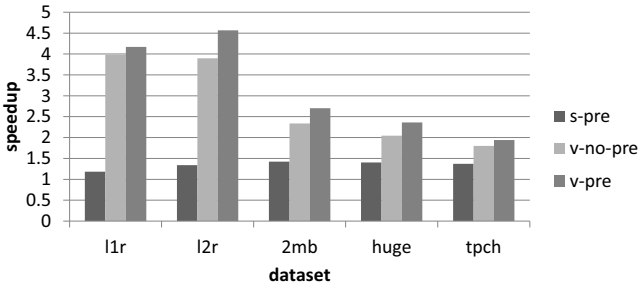


Figure 9: Software Prefetching Speedup

5.6. Existing SIMD Solutions

So far different aspects of the proposed hardware have been evaluated and compared against a scalar baseline. What is missing is a comparison against a hash join implementation that utilises the existing multimedia extensions found in commodity processors such as SSE and AVX. The problem is that no version can exist given the limitations of these instruction set extensions. Nevertheless we measured the difference in execution time between a purely scalar (but optimised) version of the code and a version with GCC's autovectorisation enabled. The functions were altered to expose alignment and absence of aliasing to help the autovectorisation algorithm. GCC was able to transform a small portion of the code to use SSE instructions, namely the part that hashes the LHS input. When both versions were run, the difference in execution time was less than 1%. This is mainly due to the fact that this particular part of the code is not the most dominant in the overall algorithm.

The work of [22] made extensive optimisations to, and an evaluation of, the hash join algorithm and concluded that for DLP to be attained effectively there must be efficient support for indexed memory operations. Intel introduced the LRBni instruction set [1] which was originally intended for the Larrabee [34] architecture. LRBni introduced SIMD registers wider than existing SSE/AVX registers that can contain eight 64-bit elements; it additionally offers gather and scatter support to and from these new registers. The catch is that the hardware was developed with graphical processing in mind and the indexed support was geared towards data structures with high spatial locality e.g. in-cache lookup tables.

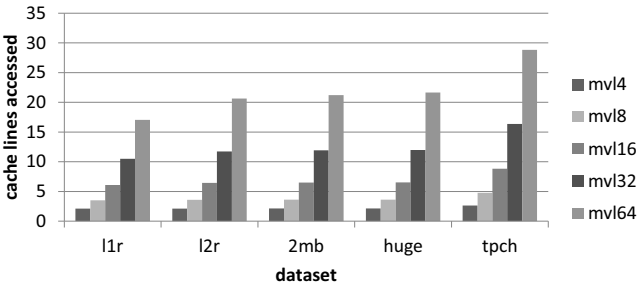


Figure 10: Average Number of Cache Lines Accessed

The hash join algorithm offers no such luxury to the hardware and the gather operations generally don't pull elements resident from the

same cache line. To help illustrate this, Figure 10 shows the average number of cache lines touched per indexed load. Each dataset is run with a different MVL. The average is quite high and furthermore many gathers from the larger software structures access as many cache lines as the MVL i.e. the worst case scenario. Consequently it appears that an architecture like Larrabee doesn't seem to be an optimal fit for an algorithm like this.

Intel has publicly released a list of new instructions that will part of the AVX2 standard [20]. AVX2 will include an indexed load instruction although doesn't include a complementary indexed store. The new indexed load could be useful when applied to the hash join algorithm however this also depends on how this instruction is implemented. The first microarchitecture set to use AVX2 is codenamed Haswell and expected to be released in 2013. So far, no details have been released on how AVX2 will be implemented on Haswell so a comparison with our work would be entirely speculative.

6. Related Work

This section details several works that have attempted to accelerate DBMS software through available DLP. We compare and contrast our own research with those mentioned.

[26] is the earliest work found looking at database operator implementations on vector processors. The report looks specifically at the hash join operator and implements a vectorised version for the Cray C90 [28]. The methodology of this work takes a different approach to ours. We profile an existing full-featured DBMS that has been optimised for modern out of order microarchitectures to find bottlenecks due to scalar inefficiencies. In contrast, this work proposes its own algorithm for hash joins with no reference to a real DBMS. Additionally, we are proposing vector extensions to an ISA that already dominates the server marker whereas this work is done exclusively on a supercomputer. In a similar vein to [26], [27] also looks at the vectorisation of database operators. This time the list is expanded to selection, projection and join operators however the methodology is still the same. Naïve scalar implementations are run against vectorised versions on a supercomputer and so the same arguments still apply.

[42] is a broad study accelerating various database operators using the existing SSE instruction extensions for x86. The work investigates the benefits of DLP and reduction of conditional branches in implementations of scans, aggregations, indexed operations and joins. Since our work is primarily focused on joins, a comparison of this feature is given.

The principal difference between our vectorised join and their SIMDised join is that their work looks at a simple nested loop implementation whereas our work looks at an optimised hash join implementation. A nested loop join compares every row from the left hand side table with every row from the right hand side table. This is not a problem when the tables are small, but if they are large this is a very inefficient join algorithm. In contrast, we look at a hash join implementation which is suitable for large tables typically found in decision support systems.

[16] is a study that ports a DBMS to the Cell Broadband Engine [14], an architecture abundant with DLP capabilities. What is interesting is that the query engine used in the study is MonetDB/X100 [7] which is an earlier version of the query engine used in Vectorwise. The work mostly discusses the challenges that arise from using this esoteric architecture. Furthermore, the work is evaluated using TPC-H query number one which lacks a join operation. Our work is primarily focused on joins so it is difficult to make a comparison.

[15] is a very comprehensive work investigating the performance benefits of running DBMS operations on graphics processing units (GPUs). The study includes a hash join implementation that runs on a GPGPU coprocessor. There are some performance benefits but the study concludes that the necessity to transfer data between the global memory and the GPU's local memory can be a large bottleneck. Our approach adds vector processing capabilities into the CPU's execution core so this penalty is never encountered. This is important when treating the DBMS software as a whole since it can have complex control flow mixed with segments more suitable for DLP-oriented hardware. Additionally, [24] discusses the merits of using vector-based architectures over GPGPU with respect to performance, area and energy efficiency.

7. Conclusions

In this work we have examined a leading decision support DBMS, Vectorwise, and found that hash joins can form a significant proportion of its execution- 61% of the total execution time. It was found that the probe phase of hash joins contains an abundance of DLP that isn't expressible using the multimedia SIMD extensions found in commodity processors. We have proposed instruction set extensions to the x86-64 ISA that are suitable for capturing the algorithm compactly and efficiently. These instructions have been introduced into a modern microarchitecture taking advantage of existing scalar structures where possible and without compromising their performance.

This work explored various trade-offs in the design space. The decision to use out of order logic coupled with the vector additions has been quantitatively evaluated and shown to give an extra 1.34x performance speedup on average. The benefit of using manually programmed memory fences was shown to give 1.39x extra performance although comparing our custom fence implementation against a naïve version yielded only a small speedup of 1.03x. This small gain makes it hard to justify the extra hardware used to implement fences in this manner. Finally the penalty incurred when bypassing the L1D cache was measured and found to be negligible.

Our results show that the new vectorised implementation of hash probe, accounting for 41% of total execution time, can get speedups between 1.94x and 4.56x over the scalar baseline. We have shown the benefits of using two memory controllers in conjunction with the vector hardware and also demonstrated that the scalar code cannot take advantage of the extra available bandwidth. Furthermore we show that increasing the out of order structures and superscalar widths gives disproportional returns whereas the vector approach achieves an order of magnitude greater speedup. Finally we confirm that software prefetching techniques described in [8] can accelerate the algorithm, albeit not as much as our solution using vectorisation. We also demonstrate that this strategy is complementary to our work and can be used in conjunction with vectorisation for an even greater speedup.

The performance gains observed don't come without a cost; an out of order vector unit is a major addition to a microprocessor. Future work will focus on measuring the area overhead and power consumption introduced with the new vector support. Additionally we will investigate *build*, the other phase of the hash join algorithm. Although less significant than the probe phase in overall execution time, it presents an interesting dilemma of updating a common structure and may violate the necessity of scatter stores being independent and conflict free at the element level.

8. Acknowledgements

The authors would like to thank Peter Boncz, Marcin Żukowski and Paul Rosenfeld for their helpful advice and feedback. This work was partially supported by the cooperation agreement between the Barcelona Supercomputing Center and Microsoft Research, by the Ministry of Science and Technology of Spain and the European Union (FEDER funds) under contracts TIN2007-60625 and TIN2008-02055-E, and by the European Network of Excellence on High-Performance Embedded Architecture and Compilation (HiPEAC).

References

[1] M. Abrash, "A First Look at the Larrabee New Instructions (LRBni)," http://drdobbs.com/high-performance-computing/216402188, 2009, accessed on 2011-09-08.

[2] D. Abts *et al.*, "The Cray BlackWidow: A Highly Scalable Vector Multiprocessor," in *Proceedings of the ACM/IEEE Conference on Supercomputing*, 2007, pp. 17:1–17:12.

[3] Actian, "Vectorwise. Record Breaking Action Engine for Big Data," http://www.actian.com/products/vectorwise.

[4] K. Asanović, "Vector Microprocessors," Ph.D. dissertation, EECS Department, University of California, Berkeley, 1998.

[5] L. A. Barroso, K. Gharachorloo, and E. Bugnion, "Memory System Characterization of Commercial Workloads," in *Proceedings of the 25th Annual International Symposium on Computer Architecture*, 1998, pp. 3–14.

[6] P. A. Boncz, S. Manegold, and M. L. Kersten, "Database Architecture Optimized for the New Bottleneck: Memory Access," in *Proceedings of the 25th International Conference on Very Large Data Bases*, 1999, pp. 54–65.

[7] P. A. Boncz, M. Zukowski, and N. Nes, "MonetDB/X100: Hyper-Pipelining Query Execution," in *Proceedings of the 2nd Biennial Conference on Innovative Data Systems Research*, 2005, pp. 225–237.

[8] S. Chen *et al.*, "Improving Hash Join Performance through Prefetching," in *Proceedings of the 20th International Conference on Data Engineering*, 2004, pp. 116–127.

[9] G. P. Copeland and S. N. Khoshafian, "A Decomposition Storage Model," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 1985, pp. 268–279.

[10] J. Corbal, M. Valero, and R. Espasa, "Exploiting a New Level of DLP in Multimedia Applications," in *Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture*, 1999, pp. 72–79.

[11] R. Espasa *et al.*, "Tarantula: A Vector Extension to the Alpha Architecture," in *Proceedings of the 29th Annual International Symposium on Computer Architecture*, 2002, pp. 281–292.

[12] R. Espasa, M. Valero, and J. E. Smith, "Out-of-Order Vector Architectures," in *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, 1997, pp. 160–170.

[13] R. Espasa, M. Valero, and J. E. Smith, "Vector Architectures: Past, Present and Future," in *Proceedings of the 12th International Conference on Supercomputing*, 1998, pp. 425–432.

[14] M. Gschwind *et al.*, "Synergistic Processing in Cell's Multicore Architecture," *IEEE Micro*, vol. 26, no. 2, pp. 10–24, 2006.

[15] B. He *et al.*, "Relational Query Coprocessing on Graphics Processors," *ACM Transactions on Database Systems*, vol. 34, no. 4, pp. 21:1–21:39, 2009.

[16] S. Héman *et al.*, "Vectorized Data Processing on the Cell Broadband Engine," in *Proceedings of the 3rd International Workshop on Data Management on New Hardware*, 2007, pp. 4:1–4:6.

[17] U. Hoelzle and L. A. Barroso, *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*, 1st ed. Morgan and Claypool Publishers, 2009.

[18] Ingres, "Ingres/VectorWise sneak Preview on the Intel Xeon Processor 5500 series-based platform," white paper, 2009.

[19] *Intel®64 and IA-32 Architectures Optimization Reference Manual*, Intel, June 2011.

[20] *Intel®Advanced Vector Extensions Programming Reference*, Intel, June 2011.

[21] B. Jacob, S. Ng, and D. Wang, *Memory Systems: Cache, DRAM, Disk*, 1st ed. Morgan Kaufmann Publishers Inc., 2007.

[22] C. Kim *et al.*, "Sort vs. Hash Revisited: Fast Join Implementation on Modern Multi-Core CPUs," *Proceedings of The VLDB Endowment*, vol. 2, no. 2, pp. 1378–1389, 2009.

[23] C. Kozyrakis and D. Patterson, "Vector Vs. Superscalar and VLIW Architectures for Embedded Multimedia Benchmarks," in *Proceedings of the 35th Annual ACM/IEEE International Symposium on Microarchitecture*, 2002, pp. 283–293.

[24] Y. Lee *et al.*, "Exploring the Tradeoffs between Programmability and Efficiency in Data-Parallel Accelerators," in *Proceedings of the 38th Annual International Symposium on Computer Architecture*, 2011, pp. 129–140.

[25] C. Lemuet *et al.*, "The Potential Energy Efficiency of Vector Acceleration," in *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, 2006.

[26] R. Martin, "A Vectorized Hash-Join," 1996, iRAM technical report, University of California at Berkeley.

[27] S. Meki and Y. Kambayashi, "Acceleration of Relational Database Operations on Vector Processors," *Systems and Computers in Japan*, vol. 31, no. 8, pp. 79–88, 2000.

[28] W. Oed and M. Walker, "An Overview of Cray Research Computers including the Y-MP/C90 and the new MPP T3D," in *Proceedings of the 5th Annual ACM Symposium on Parallel Algorithms and Architectures*, 1993, pp. 271–272.

[29] S. Palacharla, N. P. Jouppi, and J. E. Smith, "Complexity-Effective Superscalar Processors," in *Proceedings of the 24th Annual International Symposium on Computer Architecture*, 1997, pp. 206–218.

[30] F. Quintana *et al.*, "Adding a Vector Unit to a Superscalar Processor," in *Proceedings of the 13th International Conference on Supercomputing*, 1999, pp. 1–10.

[31] S. K. Raman, V. Pentkovski, and J. Keshava, "Implementing Streaming SIMD Extensions on the Pentium III Processor ," *IEEE Micro*, vol. 20, no. 4, pp. 47–57, 2000.

[32] P. Rosenfeld, E. Cooper-Balis, and B. Jacob, "DRAMSim2: A Cycle Accurate Memory System Simulator," *IEEE Computer Architecture Letters*, vol. 10, no. 1, pp. 16–19, 2011.

[33] W. Schönauer, *Scientific Computing on Vector Computers*. Elsevier Science Publisher B.V., 1987.

[34] L. Seiler *et al.*, "Larrabee: A Many-Core x86 Architecture for Visual Computing," *ACM Transactions on Graphics*, vol. 27, no. 3, pp. 18:1–18:15, 2008.

[35] N. T. Slingerland and A. J. Smith, "Multimedia Extensions for General Purpose Microprocessors: A Survey," *Microprocessors and Microsystems*, vol. 29, no. 5, pp. 225–246, 2005.

[36] J. E. Smith, G. Faanes, and R. Sugumar, "Vector Instruction Set Support for Conditional Operations," in *Proceedings of the 27th Annual International Symposium on Computer Architecture*, 2000, pp. 260–269.

[37] J. Sompolski, M. Zukowski, and P. Boncz, "Vectorization vs. Compilation in Query Execution," in *Proceedings of the 7th International Workshop on Data Management on New Hardware*, 2011, pp. 33–40.

[38] S. Srinivasan *et al.*, "CMP Memory Modeling: How Much Does Accuracy Matter?" in *Proceedings of the 5th Annual Workshop on Modeling, Benchmarking and Simulation*, 2009, pp. 24–33.

[39] Transaction Processing Performance Council, "TPC-H Standard Specification v2.14.2," http://www.tpc.org/tpch/, 2011.

[40] T. Willhalm *et al.*, "SIMD-Scan: Ultra Fast in-Memory Table Scan using on-Chip Vector Processing Units," *Proceedings of The VLDB Endowment*, vol. 2, no. 1, pp. 385–394, 2009.

[41] M. T. Yourst, "PTLsim: A Cycle Accurate Full System x86-64 Microarchitectural Simulator," in *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, 2007, pp. 23–34.

[42] J. Zhou and K. A. Ross, "Implementing Database Operations Using SIMD Instructions," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2002, pp. 145–156.

[43] M. Żukowski, "Balancing Vectorized Query Execution with Bandwidth-Optimized Storage," Ph.D. dissertation, Universiteit van Amsterdam, 2009.

2012 IEEE/ACM 45th Annual International Symposium on Microarchitecture

NOC-Out: Microarchitecting a Scale-Out Processor

Pejman Lotfi-Kamran Boris Grot Babak Falsafi
EcoCloud, EPFL
{pejman.lotfikamran,boris.grot,babak.falsafi}@epfl.ch

Abstract

Scale-out server workloads benefit from many-core processor organizations that enable high throughput thanks to abundant request-level parallelism. A key characteristic of these workloads is the large instruction footprint that exceeds the capacity of private caches. While a shared last-level cache (LLC) can capture the instruction working set, it necessitates a low-latency interconnect fabric to minimize the core stall time on instruction fetches serviced by the LLC. Many-core processors with a mesh interconnect sacrifice performance on scale-out workloads due to NOC-induced delays. Low-diameter topologies can overcome the performance limitations of meshes through rich inter-node connectivity, but at a high area expense.

To address the drawbacks of existing designs, this work introduces NOC-Out – a many-core processor organization that affords low LLC access delays at a small area cost. NOC-Out is tuned to accommodate the bilateral *core-to-cache access pattern, characterized by minimal coherence activity and lack of inter-core communication, that is dominant in scale-out workloads. Optimizing for the bilateral access pattern, NOC-Out segregates cores and LLC banks into distinct network regions and reduces costly network connectivity by eliminating the majority of inter-core links. NOC-Out further simplifies the interconnect through the use of low-complexity tree-based topologies. A detailed evaluation targeting a 64-core CMP and a set of scale-out workloads reveals that NOC-Out improves system performance by 17% and reduces network area by 28% over a tiled mesh-based design. Compared to a design with a richly-connected flattened butterfly topology, NOC-Out reduces network area by 9x while matching the performance.*

1. Introduction

Today's information-centric world is powered by servers. A recent report estimates the server hardware market to exceed $57 billion in 2014 [5], with various online services propelling the growth. The size of the market has motivated both established and start-up hardware makers to develop specialized processors for server workloads, as evidenced by designs such as Oracle's T-series and IBM's POWER.

Recent research examining scale-out workloads behind many of today's online services has shown that, as a class, these workloads have a set of common characteristics that differentiate them from desktop, media processing, and scientific domains [4]. A typical scale-out workload, be it a streaming service or web search, handles a stream of mostly independent client requests that require accessing pieces of data from a vast dataset. Processing a diversity of requests, scale-out workloads have large active instruction footprints, typically in the order of several megabytes.

The presence of common traits – namely, (a) request independence, (b) large instruction footprints, and (c) vast dataset sizes – indicates that processors can readily be specialized for this workload class. The abundant request-level parallelism argues for processor designs with a large number of cores to maximize throughput. The independent nature of requests virtually eliminates inter-thread communication activity; however, large instruction footprints require a fast communication path between the individual cores and the last-level cache (LLC) containing the applications' instructions. Finally, the vast dataset dwarfs on-die storage capacities and offers few opportunities for caching due to limited reuse [4].

Taking advantage of common workload features, and driven by the need to increase server efficiency, the industry has introduced processors, which we broadly refer to as *scale-out processors*, that are specialized to scale-out workloads. An example of an existing scale-out processor design is the Oracle T-series, which features up to 16 cores, 3-6 MB LLC capacities, and a low-latency crossbar interconnect. Extending and formalizing the space of scale-out processors, researchers introduced the Scale-Out Processor (SOP) design methodology [15]. The SOP methodology, which provides an optimization framework for deriving optimal core counts and LLC capacities based on microarchitectural and technology parameters, advocates many cores, modestly-sized LLCs, and low interconnect delays.

With both industry and researchers calling for many-core scale-out processor designs, an open question is how should the cores and LLC be arranged and interconnected for maximum efficiency. In light of known scalability limitations for crossbar-based designs, existing many-core chip multiprocessors (CMPs), such as Tilera's Tile series [19], have featured a mesh-based interconnect fabric and a tiled organization. Each tile integrates a core, a slice of the shared LLC with directory, and a router. The resulting organization enables cost-effective scalability to high core counts; however, the mesh-based design sacrifices performance on scale-out workloads due to its large average hop count [15]. Each hop involves a router traversal, which adds delay that prolongs the core stall time on instruction fetches serviced by the LLC.

To reduce NOC latency, researchers have proposed low-

1072-4451/12 $26.00 © 2012 IEEE
DOI 10.1109/MICRO.2012.25

IEEE computer society

diameter NOC topologies, such as the flattened butterfly [13], that leverage the abundant on-chip wire budget to achieve rich inter-node connectivity. By minimizing the number of router traversals, a low-diameter network improves performance over a mesh-based design by accelerating accesses to the LLC. However, the performance gain comes at considerable area overhead stemming from the use of many-ported routers and a multitude of repeater-intensive long-range links.

In this work, we address the scalability challenge for scale-out processors through NOC-Out – a core, cache, and interconnect organization specialized for the target workload domain. We identify the direct communication between cores and LLC banks, which we term *bilateral*, as the dominant permutation in scale-out workloads and show that other forms of communication, including coherence activity, is rare. Based on this insight, NOC-Out decouples LLC tiles from the cores and localizes them in a central portion of the die. The segregated organization naturally accommodates the bilateral core-to-cache access pattern. More importantly, with the traffic flowing between spatially distinct regions (cores to caches and back to the cores), NOC-Out virtually eliminates the need for direct inter-core connectivity, affording a significant reduction in network cost.

To further optimize interconnect cost and performance, NOC-Out deploys simple *reduction trees* to carry messages from the cores to the centrally-located LLC banks. Each reduction tree is shared by a small number of cores. A node in a tree is just a buffered 2-input mux that merges packets from a local port with those already in the network. This simple design reduces cost and delay by eliminating the need for routing, multi-port arbitration, complex switches, and deep buffers. Similarly, NOC-Out uses low-complexity *dispersion trees* to carry the data from the cache banks to the cores. A node in a dispersion tree is a logical opposite of that in a reduction tree, allowing packets to either exit the network or advancing them up the tree at minimal cost and delay.

We use a full-system simulation infrastructure, along with detailed area and energy models for a 32nm technology node, to evaluate NOC-Out in the context of a 64-core CMP on a set of scale-out workloads. Our results show that NOC-Out matches the performance of a conventional tiled organization with a flattened butterfly interconnect while reducing the network area by a factor of 9, from a prohibitive 23mm^2 to an affordable 2.5mm^2. Compared to a mesh-based design, NOC-Out improves system performance by 17% while requiring 28% less network area.

2. Background

In this section, we examine scale-out workloads and the demands they place on processor designs.

2.1. Scale-Out Workloads

Research analyzing the scale-out workload domain has shown that a key set of traits holds across a wide range of workloads, including web search, media streaming, and web serving. These traits can be summarized as (a) request independence, (b) large instruction footprint, and (c) vast dataset [4]. We next examine each of these traits to understand their effect on processor design.

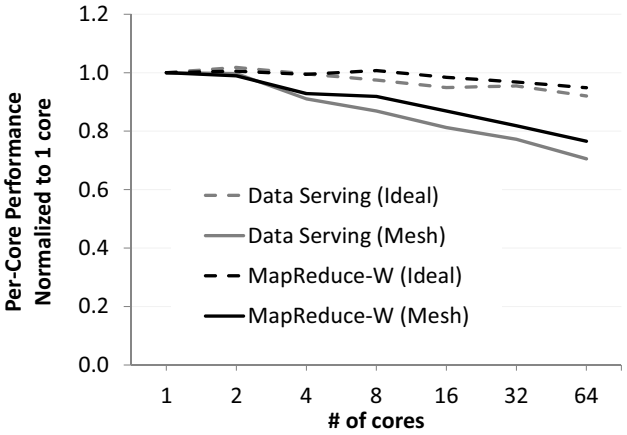


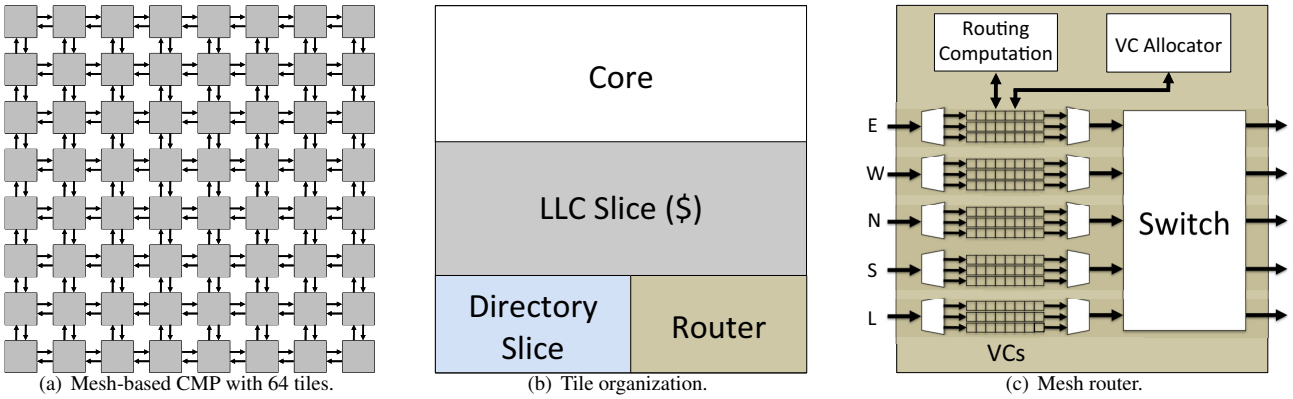
Figure 1: Effect of distance (which grows with core count) on per-core performance for ideal and mesh-based interconnects on two scale-out workloads.

Request Independence: Scale-out workloads handle a stream of requests that are, to an overwhelming extent, mutually independent. Fundamentally, request independence is the feature that makes scale-out workloads inherently parallel and attractive for execution on many-core chips. Another implication of request independence is the lack of inter-thread communication. Write sharing among cores working on separate requests is rare due to the vast data working set size; nonetheless, the shared memory programming model is valued in the scale-out domain as it simplifies software development and facilitates the use of existing software stacks.

Large instruction footprint: Active instruction working sets in scale-out workloads are typically measured in megabytes and are characterized by complex control flow. As a result, private last-level caches tend to lack the requisite capacity for capturing the instruction footprint. Shared last-level caches, on the other hand, have the capacity and reduce replication when compared to private caches as different cores are often executing the same workload and can share instructions [8].

One challenge with large, LLC-resident instruction working sets is that the on-die distance between the cores and the LLC adds delay to the cache access time. Because L1-I misses stall the processor, scale-out workloads are particularly sensitive to the on-die communication delays due to frequent instruction fetches from the LLC.

Figure 1 shows the effect of distance on per-core performance for two representative scale-out workloads. In this experiment, an 8MB LLC is shared by all cores on the die. The number of cores is indicated on the x-axis; more cores result in a larger die size and a longer average distance between each core and the target LLC bank. The figure compares the



(a) Mesh-based CMP with 64 tiles. (b) Tile organization. (c) Mesh router.

Figure 2: Elements of tiled CMPs.

performance of an idealized interconnect (labeled "Ideal") in which only the wire delay is exposed (i.e., routing, arbitration, switching, and buffering all take zero time) to a realistic mesh-based interconnect with a 3-cycle per-hop delay (router and wire delay). To focus the study, we do not model contention in either network. As the figure shows, interconnect delay has a significant effect on performance that increases with core count. At 64 cores, the average difference in performance between an ideal and mesh-based interconnect is 22%.

Vast dataset: Scale-out workloads operate on vast amounts of data that is frequently kept in DRAM to reduce the access latency. The data working set of these workloads dwarfs the capacity of on-die caches. Moreover, there is essentially no temporal reuse in the data stream. The combination of these features renders on-die caches ineffective for capturing the data working set, indicating that committing large swaths of the die real-estate to cache is not useful.

To recap, scale-out workloads are best served by many-core chips featuring a modestly-sized LLC for capturing the instruction working set and an on-die interconnect optimized for low cache access latency.

2.2. Scale-Out Processors

The observations captured in the previous section are reflected in several contemporary processors targeted at the scale-out market. A representative example is the Oracle T-series (formerly, Sun Niagara) family of processors. Depending on the model, the T-series features up to 16 cores, a banked LLC with 3-6 MB of storage capacity, and a delay-optimized crossbar switch connecting the cores to the cache banks.

Extending and formalizing the space of existing scale-out processor designs, researchers have proposed the SOP methodology – a framework for performing cost-benefit analysis at the chip level in the context of scale-out workloads [15]. Given a set of microarchitectural and technology parameters, the SOP methodology uses the metric of performance density to derive optimal resource configurations,

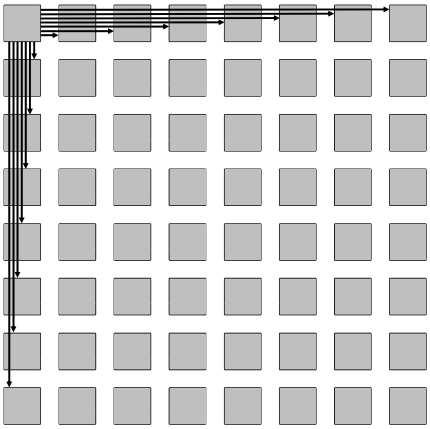


Figure 3: Flattened butterfly topology (links from only one node shown for clarity).

such as the number of cores and LLC capacity. An important conclusion of the work is that, indeed, scale-out processors benefit from many cores with a modestly-sized LLC and a fast interconnect. Subsequent work has demonstrated that many-core processor configurations derived using the SOP methodology improve performance and TCO at the datacenter level [6]. However, a key limitation of these earlier efforts has been their reliance on crossbar interconnects whose poor scalability forced suboptimal design choices.

2.3. Existing Many-Core Organizations

To overcome the scalability limitations of crossbar-based designs, emerging many-core processors, such as Tilera's Tile series, employ a tiled organization with a fully distributed last-level cache. Figure 2(a) shows an overview of a generic CMP based on a tiled design. Each tile, pictured in Figure 2(b), consists of a core, a slice of the distributed last-level cache, a directory slice, and a router. The tiles are linked via a routed, packet-based, multi-hop interconnect in a mesh topology.

The tiled organization and a structured interconnect fabric allow mesh-based designs to scale to large core counts. Unfortunately, the regularity of the mesh topology works to its disadvantage when it comes to performance scalability. Each hop in a mesh network involves the traversal of a multi-ported

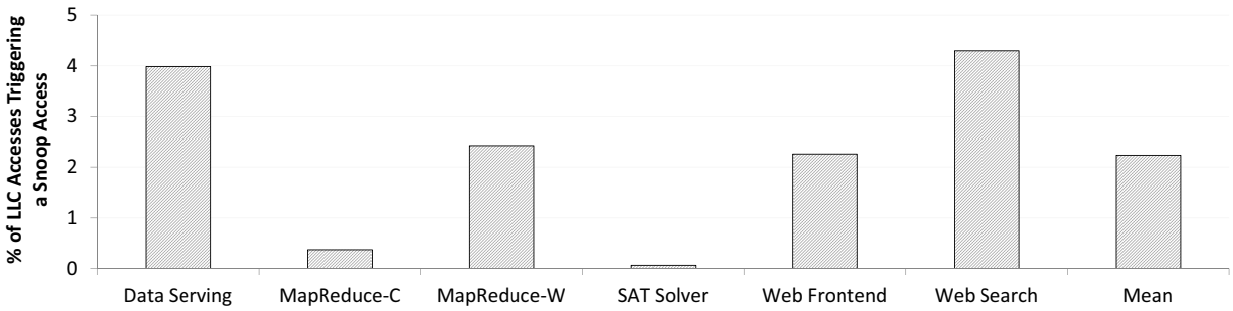


Figure 4: Percentage of LLC accesses causing a snoop message to be sent to a core.

router, shown in Figure 2(c), which adds delay due to the need to access the packet buffers, arbitrate for resources, and navigate the switch. As Figure 1 shows, in a 64-core CMP, these delays diminish the performance of a mesh-based tiled CMP by 22% compared to an ideal fabric in which only the wire delay is exposed.

To overcome the performance drawbacks of mesh-based interconnects, researchers developed low-diameter topologies suitable for on-die implementation. These topologies use rich inter-node connectivity to bypass intermediate routers between a packet's source and destination nodes. A state-of-the-art low-diameter topology is the flattened butterfly [13], shown in Figure 3. The flattened butterfly uses a set of dedicated channels to fully connect a given node to others along the row and column. The resulting network requires, at most, two hops (one in each of the X and Y dimensions) to deliver the packet to the destination. In doing so, the flattened butterfly greatly reduces the contribution of routers to the end-to-end delay, allowing performance to approach that of an ideal interconnect.

Problematically, the performance advantages of the flattened butterfly, or another richly-connected NOC, come at considerable area expense stemming from the use of many-ported routers and a multitude of links. For instance, in the flattened butterfly in Figure 3, each router necessitates 14 network ports (7 in each of the two dimensions) plus a local port. The network ports are costly due to the presence of deep packet buffers necessary to cover the flight time of the long-range links. Meanwhile, the routers' internal switch fabric is area-intensive due the need to interconnect a large number of ports. Finally, links consume valuable on-die real-estate due to the need for frequent repeater placement[1], even though wires themselves can be routed over tiles.

To summarize, existing NOC architectures require an uneasy choice between performance and area-efficiency. Meanwhile, scale-out processors demand both – good performance and good area-efficiency.

[1]Repeaters are necessary to overcome poor RC characteristics of wires in current and future technologies.

3. Memory Traffic in Scale-Out Workloads

In order to maximize the efficiency of scale-out processors, we examine the memory traffic in scale-out workloads to identify opportunities for specialization.

As noted earlier, scale-out workloads have large instruction footprints and vast datasets. Cores executing these workloads frequently access the LLC because neither the instructions nor the datasets fit in L1 caches. The multi-megabyte instruction footprints of scale-out workloads can be readily accommodated in the LLC while the vast datasets dwarf the LLC capacity and reside in memory. Consequently, the majority of accesses to the instruction blocks hit in the LLC while many dataset accesses miss and are filled from main memory.

On an L1 miss, the directory controller and the LLC check if the block is available on chip. If so, and if LLC's copy is the most recent, the LLC will service the miss and send the data to the requesting core. If the requesting core signals that it needs to modify the block, the directory will also send *snoop* messages to the set of sharers, instructing them to invalidate their copy. Conversely, if the directory indicates that another core has the block, it will send a snoop message to the appropriate core, instructing it to forward the block to the requester. Finally, in the case of a miss, the LLC fetches the block from main memory and returns it to the requesting core.

Importantly, coherence activity and core-to-core communication (i.e., L1-to-L1 forwarding) is triggered only as a result of data sharing at the L1 level. However, due to the high-level behavior of scale-out workloads, this type of data sharing is rare. Instructions are actively shared, but are read-only and served from the LLC; dataset is vast, and the likelihood of two independent requests sharing a piece of data is low.

Figure 4 shows the fraction of accesses to the LLC that cause a snoop message to be sent to an L1 cache across six scale-out workloads. As expected, coherence activity is negligible in these workloads, with an average of two out of 100 LLC accesses triggering a snoop. Earlier work made similar observations for both scale-out [4] and server [14] workloads.

The lack of coherence activity in scale-out workloads implies that the dominant traffic flow is from the cores to the LLC and back to the cores. We refer to this phenomenon as core-to-cache *bilateral* access pattern. In tiled processors, the coupled nature of core and LLC slices means that accesses

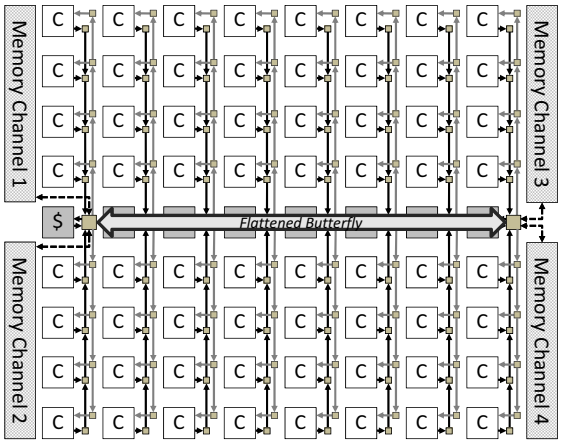


Figure 5: NOC-Out organization.

to the last-level cache from each individual core, over time, target all of the tiles, resulting in an all-to-all traffic pattern at the chip level. Achieving low latency under all-to-all traffic requires a richly connected topology, necessarily resulting in high area and wire cost.

4. NOC-Out

NOC-Out is a processor organization optimized for the bilateral access pattern dominant in scale-out workloads. NOC-Out leverages two insights to minimize interconnect delays at a small area footprint. First, NOC-Out segregates the LLC slices from the cores into separate cache-only tiles and concentrates the cache tiles in the center of the die. The segregation of cores and the LLC breaks the all-to-all traffic pattern characteristic of tiled CMPs and establishes a bilateral traffic flow between core and cache regions. Second, NOC-Out takes advantage of the bilateral traffic to limit network connectivity, enabling a reduction in network cost. Specifically, NOC-Out eliminates the bulk of the core-to-core links and the supporting router structures, preserving a minimum degree of connectivity to enable each core to reach the LLC region.

Figure 5 shows a high-level view of the proposed organization, featuring LLC slices in the center of the die and core tiles on both sides of the LLC. NOC-Out uses simple, routing-free *reduction trees* to guide packets toward the centralized cache banks, and *dispersion trees*, which are logical opposites of reduction trees, to propagate response data and snoop traffic out to the cores. Every reduction and dispersion tree connects a small number of cores to exactly one cache bank. The LLC banks are linked in a flattened butterfly topology forming a low-latency NUCA cache. Notably, NOC-Out does not support direct core-to-core connectivity, requiring all traffic to flow through the LLC region.

In the rest of the section, we detail the organization of the reduction, dispersion, and LLC networks.

4.1. Reduction Network

The reduction network is designed for a low-latency delivery of packets from the cores to the centralized cache banks. Figure 6(a) shows key features of a reduction tree, which spans

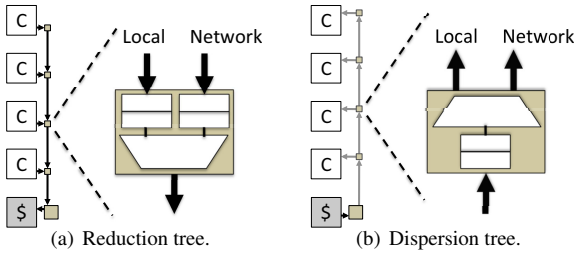


Figure 6: Details of NOC-Out networks.

a column of cores and terminates at the LLC bank at the end of the column. Effectively, a reduction tree is a many-to-one interconnect, with all packets that enter a reduction tree flowing to the same destination cache bank. A node in the tree is a buffered, flow-controlled, two-input multiplexer that merges packets from the local port with those already in the network.

Compared to a conventional packet-based NOC, the reduction network does not require routing, as all packets flow to a common destination. The switch, typically implemented as a crossbar or a mux tree in conventional NOCs, is reduced to a simple two-input mux in a reduction tree. The reduction network is similar to conventional NOCs in that it benefits from the use of virtual channels for protocol deadlock avoidance, and as such requires a virtual channel allocation mechanism. However, with just two ports (local and network), the VC allocator is trivially simple. In fact, given the low memory-level parallelism of scale-out workloads [4], static-priority arbitration policies that always prioritize the network over the local port (or vice-versa) tend to work well and afford further simplification of the arbitration logic.

NOC-Out distinguishes three message classes – data requests, snoop requests, and responses (both data and snoop) – to guarantee network-level deadlock freedom for its coherence protocol. Of these, only data requests and responses travel through the reduction trees, as snoop requests can only originate at the directory nodes at the LLC. As a result, each port in a reduction tree has two virtual channels, one per message class.

Upon arrival at a router in a reduction tree, a packet is buffered in the appropriate VC (determined by the packet's message class). With a total of four VCs in a router (two ports with two VCs per port), a 4:1 arbiter selects a winning VC based on priority and downstream buffer availability. In this work, we assume the following fixed priority ordering of VCs (highest to lowest): network responses, local responses, network requests, local requests. By prioritizing the network over the local port, we seek to mitigate the latency disadvantage of cores that are more distant from the LLC. Because a reduction tree router has exactly one output port, routing and output port selection logic is unnecessary, and just one arbiter is required per node.

4.2. Dispersion Network

The dispersion network carries packets (data responses and snoop requests) from the LLC to the cores. Figure 6(b) shows

a logical view of a dispersion tree. A dispersion tree is a logical opposite of the reduction tree, with a single source (a cache bank) and multiple destinations (cores). Each node in a tree is a buffered, flow-controlled demultiplexer that selects a local output port for packets that have reached their destination or propagates them farther up the tree toward the next node.

As is the case with the reduction network, virtual channels are necessary for deadlock avoidance to guarantee that snoop requests do not block data responses from reaching their destination. With two VCs per node (one per message class), on each clock cycle, simple control logic (1) uses message priority and buffer availability to select a winning VC, and (2) sets up demux control to forward a flit from the selected VC to the local or network output. Again, we use a static priority assignment to prioritize reply messages over snoop requests, subject to buffer availability.

4.3. LLC Network

As described above, NOC-Out segregates core and LLC slices[2] into separate tiles. Because each core connects to just one LLC tile through its reduction and dispersion trees, NOC-Out relies on a richly-connected flattened butterfly network to route traffic between LLC tiles. The choice of the network is motivated by the need to minimize delay and reduce contention in the LLC region.

In order to reduce the area and channel expense of the flattened butterfly, NOC-Out takes advantage of the fact that the number of LLC tiles need not match the number of core tiles. The number of LLC tiles can be reduced because low instruction- and memory-level parallelism in scale-out workloads naturally dampen the bandwidth pressure on the LLC. Our empirical data shows that a design with four cores per one LLC bank achieves a level of performance that is within 2% of a system with an equal number of cores and banks. Moreover, each LLC tile can house multiple banks that share the router. A reduction in the number of the LLC tiles diminishes the cost and extent of the richly-connected LLC network.

4.4. Additional Considerations

Before concluding the description of NOC-Out, we highlight several additional aspects of the proposed design; namely, its flow control architecture, connectivity to off-die interfaces, and support for shared memory.

Flow control: All three NOC-Out networks (reduction, dispersion, and LLC) rely on conventional virtual channel credit-based flow control. The amount of buffering per port in both reduction and dispersion trees is insignificant (a few flits per VC) thanks to a short round-trip credit time resulting from a trivial pipeline. The flattened butterfly LLC network requires more buffering per port to cover the multi-cycle delays of long-range links and multi-stage routers; however, this cost is restricted to just a fraction of the nodes.

[2] An LLC slice is composed of data, tags, and directory.

Off-die interfaces: Contemporary server chips integrate a number of off-die interfaces, such as memory controllers, to improve performance and reduce system cost. In the NOC-Out design, these are accessed through dedicated ports in the edge routers of the LLC network, as shown in Figure 5.

Shared memory: Shared memory is a prominent feature of today's software stacks. Despite being optimized for the bilateral core-to-cache communication, NOC-Out fully supports the shared memory paradigm through conventional hardware coherence mechanisms, preserving full compatibility with existing software. What NOC-Out sacrifices by eliminating direct core-to-core connectivity is the support for locality-optimized communication. Instead, NOC-Out specializes for cost and performance on scale-out server workloads that do not benefit from locality optimizations.

5. Methodology

Table 1 summarizes the key elements of our methodology, with the following sections detailing the specifics of the evaluated designs, technology parameters, workloads, and simulation infrastructure.

5.1. CMP Parameters

Our target is a many-core CMP implemented in 32nm technology. We use the Scale-Out Processor methodology [15] to derive the optimal core count, number of memory controllers, and LLC capacity for the assumed technology and microarchitectural parameters. The resulting processor features 64 cores, 8MB of last-level cache, and four DDR3-1667 memory channels. Core microarchitecture is modeled after an ARM Cortex-A15, a three-way out-of-order design with 32KB L1-I and L1-D caches. Cache line size is 64B.

We consider three system organizations, as follows:

Mesh: Our baseline for the evaluation is a mesh-based tiled CMP, as shown in Figure 2. The 64 tiles are organized as an 8-by-8 grid, with each tile containing a core, a slice of the LLC and a directory node.

At the network level, a mesh hop consists of a single-cycle link traversal followed by a two-stage router pipeline for a total of three cycles per hop at zero load. The router performs routing, VC allocation, and speculative crossbar (XB) allocation in the first cycle, followed by XB traversal in the next cycle. Each router port has 3 VCs to guarantee deadlock freedom across three message classes: data requests, snoop requests, and responses. Each VC is 5 flits deep, which is the minimum necessary to cover the round-trip credit time.

Flattened Butterfly (FBfly): The FBfly-based CMP has the same tiled organization as the mesh baseline, but enjoys rich connectivity afforded by the flattened butterfly organization as shown in Figure 3. Each FBfly router has 14 network ports (7 per dimension) plus a local port. Due to high arbitration complexity, the router does not employ speculation, resulting in a three-stage pipeline. Each router port has

Table 1: Evaluation parameters.

Parameter	Value
Technology	32nm, 0.9V, 2GHz
CMP features	64 cores, 8MB NUCA LLC, 4 DDR3-1667 memory channels
Core	ARM Cortex-A15-like: 3-way out-of-order, 64-entry ROB, 16-entry LSQ, 2.9mm^2, 1W
Cache	per MB: 3.2mm^2, 500mW
NOC Organizations:	
Mesh	Router: 5 ports, 3 VCs/port, 5 flits/VC, 2-stage speculative pipeline. Link: 1 cycle
Flattened Butterfly	Router: 15 ports, 3 VCs/port, variable flits/VC, 3 stage pipeline. Link: up to 2 tiles per cycle
NOC-Out	Reduction/Dispersion networks: 2 ports/router, 2 VCs/port, 1 cycle/hop (inc. link) LLC network: flattened butterfly

three VCs to guarantee deadlock freedom. The number of flit buffers per VC is optimized based on the location of the router in the network to minimize buffer requirements. Finally, the link delay is proportional to the distance spanned by the link. Given our technology parameters (detailed below) and tile dimensions, a flit in the channel can cover up to two tiles in a single clock cycle.

NOC-Out: Our proposed design, described in Section 4, segregates core and LLC tiles, and localizes the LLC in the center of the die. To connect cores to the LLC, NOC-Out uses specialized reduction and dispersion networks. Direct inter-core connectivity is not supported and all traffic must flow through the LLC region.

Both the reduction and dispersion networks require just two VCs per port. In the reduction network, only data requests and responses flow from the cores to the cache, as snoop requests cannot originate at the core tiles. Similarly, the response network only needs to segregate snoop requests and data responses, as data requests cannot originate at the LLC. In the absence of contention, both networks have a single-cycle per-hop delay, which includes traversal of both the link and the arbitrated mux (in the reduction tree) or demux (in the dispersion tree). This delay is derived based on the technology parameters and tile dimensions.

The LLC is organized as a single row of tiles, with each tile containing 1 MB of cache and a directory slice. The aspect ratio of the LLC tiles roughly matches that of the core tiles, allowing for a regular layout across the die, as shown in Figure 5. LLC tiles are internally banked to maximize throughput. For the evaluation, we model two banks per tile (16 LLC banks, in total), as our simulations show that this configuration achieves similar throughput at lower area cost as compared to designs with higher degrees of banking. The eight LLC tiles are fully connected via a one-dimensional flattened butterfly. LLC routers feature a 3-stage non-speculative pipeline, with three VCs per input port.

5.2. Technology Parameters

We use publicly available tools and data to estimate the area and energy of the various network organizations. Our study targets a 32nm technology node with an on-die voltage of 0.9V and a 2GHz operating frequency.

We use custom wire models, derived from a combination of sources [2, 10], to model links and router switch fabrics. For links, we model semi-global wires with a pitch of 200nm and power-delay-optimized repeaters that yield a link latency of 125ps/mm. On random data, links dissipate 50fJ/bit/mm, with repeaters responsible for 19% of link energy. For area estimates, we assume that link wires are routed over logic or SRAM and do not contribute to network area; however, repeater area is accounted for in the evaluation.

Our buffer models are taken from ORION 2.0 [11]. We model flip-flop based buffers for mesh and NOC-Out, as both have relatively few buffers per port. For the flattened butterfly, we assume SRAM buffers that are more area- and energy-efficient than flip-flops for large buffer configurations.

Cache area, energy, and delay parameters are derived via CACTI 6.5 [18]. A 1MB slice of the LLC has an area of 3.2mm^2 and dissipates on the order of 500mW of power, mostly due to leakage.

Finally, parameters for the ARM Cortex-A15 core are borrowed from Microprocessor Report and scaled down from the 40nm technology node to the 32nm target. Core area, including L1 caches, is estimated at 2.9mm^2. Core power is 1.05W at 2GHz. Core features include 3-way decode/issue/commit, 64-entry ROB, and 16-entry LSQ.

5.3. Workloads

We use scale-out workloads from CloudSuite [3]. The workloads include Data Serving, MapReduce, Web Frontend, SAT Solver, and Web Search. We consider two MapReduce workloads – text classification (MapReduce-C) and word count (MapReduce-W). For the Web Frontend workload, we use the e-banking option from SPECweb2009 in place of its open-source counterpart from CloudSuite, as SPECweb2009 exhibits better performance scalability at high core counts. Two of the workloads – SAT Solver and MapReduce – are batch, while the rest are latency-sensitive and are tuned to meet the response time objectives. Prior work [4] has shown that these workloads have characteristics representative of the broad class of server workloads as described in Section 2.1.

Four out of six workloads scale to 64 cores. The other two, namely Web Serving and Web Search, only scale to 16 cores

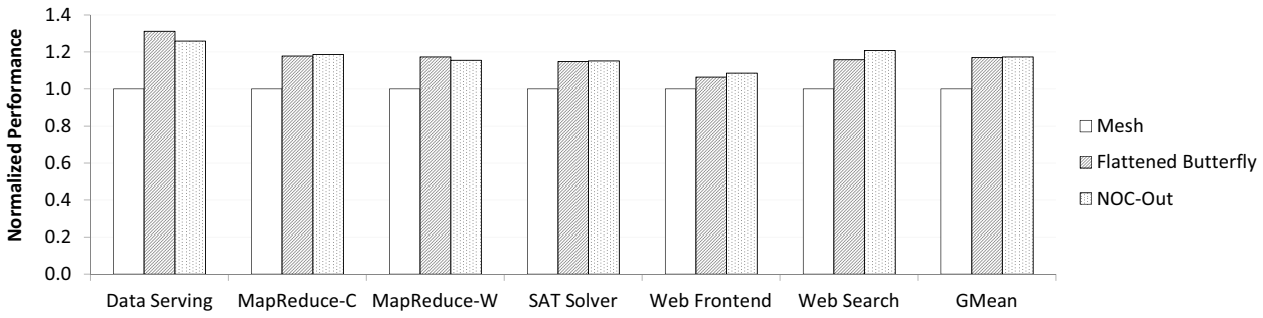


Figure 7: System performance, normalized to a mesh-based design.

due to various software bottlenecks. For these two workloads, we choose the 16 tiles in the center of the die for the mesh and flattened butterfly designs, and the 16 core tiles adjacent to the LLC in the NOC-Out design.

5.4. Simulation Infrastructure

We estimate the performance of the various processor designs using Flexus full-system simulation [22]. Flexus extends the Virtutech Simics functional simulator with timing models of cores, caches, on-chip protocol controllers, and interconnect. Flexus models the SPARC v9 ISA and is able to run unmodified operating systems and applications.

We use the SimFlex multiprocessor sampling methodology [22]. Our samples are drawn over an interval of 10 seconds of simulated time. For each measurement, we launch simulations from checkpoints with warmed caches and branch predictors, and run 100K cycles (2M cycles for Data Serving) to achieve a steady state of detailed cycle-accurate simulation before collecting measurements for the subsequent 50K cycles. We use the ratio of the number of application instructions to the total number of cycles (including the cycles spent executing operating system code) to measure performance; this metric has been shown to accurately reflect overall system throughput [22]. Performance measurements are computed with 95% confidence with an average error of less than 4%.

6. Evaluation

We first examine system performance and area efficiency of mesh, flattened butterfly, and NOC-Out designs given a fixed 128-bit link bandwidth. We then present an area-normalized performance comparison, followed by a discussion of power trends.

6.1. System Performance

Figure 7 shows full system performance, normalized to the mesh, under the various NOC organizations. Compared to the mesh, the richly-connected flattened butterfly topology improves performance by 7-31%, with a geomean of 17%. The highest performance gain is registered on the Data Serving workload, which is characterized by very low ILP and MLP, making it particularly sensitive to the LLC access latency.

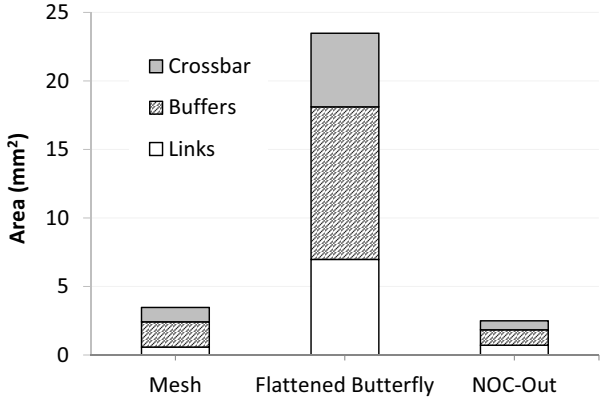


Figure 8: NOC area breakdown.

On average, the proposed NOC-Out design matches the performance of the flattened butterfly. On Data Serving, bank contention is responsible for a small performance degradation in NOC-Out, resulting in lower performance as compared to the flattened butterfly. On the other hand, on Web Search (a 16-core workload), NOC-Out enjoys a smaller average communication distance between the cores and the LLC, resulting in higher performance. The bottom line is that NOC-Out improves system performance by 17% over the mesh, and, on average, matches the performance of the flattened butterfly.

We conclude the performance assessment by noting that while the bisection bandwidths of the various topologies are different, the networks are not congested. Differences in latency, not bandwidth, across the topologies are responsible for the performance variations.

6.2. NOC Area

Figure 8 breaks down the NOC area of the three organizations by links, buffers, and crossbars. Only repeaters are accounted for in link area, as wires are assumed to be routed over tiles.

At over 23mm^2, the flattened butterfly has the highest NOC area, exceeding that of the mesh by nearly a factor of 7. The large footprint of the flattened butterfly is due to its large link budget and the use of buffer-intensive many-ported routers.

NOC-Out's interconnect footprint of 2.5mm^2 is the lowest among the evaluated designs, requiring 28% less area than a mesh and over 9 times less area than a flattened butterfly. NOC-Out's area advantage stems from minimal connectivity among the majority of the nodes (i.e., cores) and from the

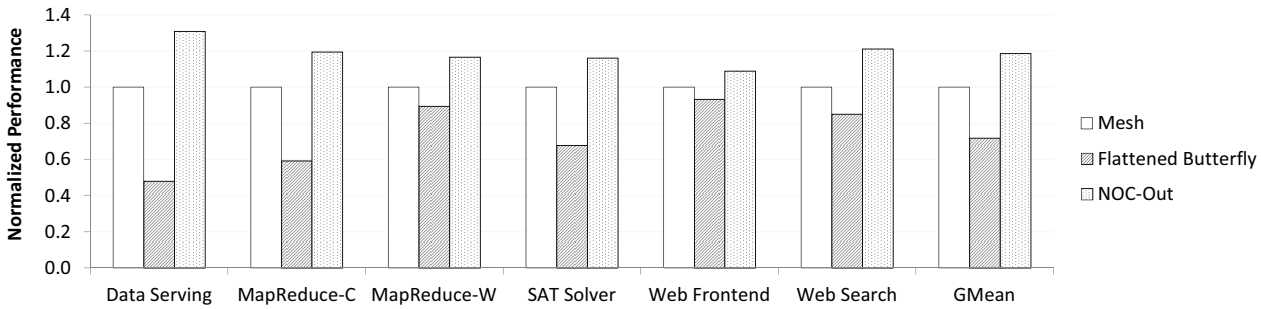


Figure 9: System performance, normalized to a mesh-based design, under a fixed NOC area budget.

use of low-complexity network trees (reduction and dispersion) that minimize router costs. Each of the two tree networks contributes just 18% to the total NOC footprint. In contrast, the flattened butterfly interconnecting NOC-Out's LLC region constitutes 64% of the total network area while linking just 11% of the tiles.

6.3. Area-Normalized Comparison

The performance and area analysis in the previous two sections assumed a fixed link width of 128 bits, resulting in vastly different NOC area costs and bisection bandwidths. To better understand how the various designs compare given a fixed NOC budget, we assess the performance of the mesh and flattened butterfly using NOC-Out's area of 2.5mm^2 as a limiting constraint. We reduce the width of both mesh and flattened butterfly NOCs until each of their respective areas (links + routers) equals that of NOC-Out and then measure the performance of the resulting designs.

Figure 9 summarizes the results of the study, with performance of the three organizations normalized to that of the mesh. Given a smaller area budget, the performance of both mesh and flattened butterfly degrades. The degradation is small in the mesh network, as the increase in the serialization latency continues to be dwarfed by the header delay. In contrast, the richly-connected flattened butterfly sees its link bandwidth shrink by a factor of 7, significantly impacting end-to-end latency through a spike in the serialization delay. Compared to the flattened butterfly at the same area budget, NOC-Out enjoys a 65% performance advantage. Compared to the mesh, NOC-Out's performance edge is 19%.

6.4. Power Analysis

Our analysis shows that the NOC is not a significant consumer of power at the chip level. For all three organizations, NOC power is below 2W. In contrast, cores alone consume in excess of 60W. Low ILP and MLP of scale-out workloads is the main reason for the low power consumption at the NOC level. Another factor is the near-absence of snoop traffic in these workloads.

NOC-Out results in the most energy-efficient NOC design, dissipating 1.3W of power, on average. Mesh and flattened butterfly average 1.8W and 1.6W, respectively. In all organizations, most of the energy is dissipated in the links. NOC-Out's higher efficiency stems from the lower average distance between the cores and the LLC, resulting in less energy spent in the wires. Meanwhile, the flattened butterfly's rich connectivity gives it an advantage over the mesh.

6.5. Summary

The evaluation results show that NOC-Out offers the performance of the richly-connected flattened butterfly topology at a fraction of the network area. Whereas the flattened butterfly requires a prohibitive 23mm^2 of die real-estate, NOC-Out necessitates just 2.5mm^2 for the interconnect. When constrained to NOC-Out's area budget, the performance of the flattened butterfly diminishes, giving NOC-Out a 65% performance advantage. In comparison to a mesh, NOC-Out improves performance by 17% and reduces the network area footprint by 28%.

7. Discussion

7.1. Scalability of NOC-Out

So far, our description and evaluation of NOC-Out has been in the context of a 64-core CMP. NOC-Out can be readily scaled to support larger numbers of cores through the use of concentration and, in configurations featuring hundreds of cores, through judicious use of express channels in reduction and dispersion networks. If necessary, the LLC network can be scaled up by extending its flattened butterfly interconnect from one to two dimensions. We now briefly discuss each of these options.

Concentration: Concentration can be used to reduce the network diameter by aggregating multiple terminals (e.g., cores) at each router node [2]. In the case of reduction and dispersion networks, a factor of two concentration at each node (i.e., two adjacent cores sharing a local port of the mux/demux) could be used to support twice the number of cores of the baseline design at nearly the same network area cost. With four times more nodes in the network and a concentration factor of four, we find that the 16B links in the tree networks are bottlenecked by insufficient bandwidth, necessitating either additional or wider links.

Express links: In future CMPs with hundreds of cores, the height of the reduction and dispersion trees may become a concern from a performance perspective. To mitigate the tree delay, express links can be judiciously inserted into the

tree to bypass some number of intermediate nodes, allowing performance to approach that of an "ideal" wire-only network. While express links increase the cost of the network due to greater channel expense, they are compatible with the simple node architectures described in Sections 4.1 and 4.2 and do not necessitate the use of complex routers.

Flattened butterfly in LLC: When executing scale-out workloads, much of the useful LLC content is the instruction footprint and OS data. Because this content is highly amenable to sharing by all the cores executing the same binary, adding cores to a scale-out processor does not mandate additional LLC capacity [15]. Should the need arise, however, to expand the LLC beyond a single row of tiles, the flattened butterfly network interconnecting the tiles can be readily scaled from one to two dimensions. While an expanded flattened butterfly increases the cost of NOC-Out, the expense is confined to the fraction of the die occupied by the LLC.

7.2. Comparison to Prior Work

NOC-Out is not the first attempt to specialize the on-chip interconnect to a specific application domain. Bakhoda et al. proposed a NOC design optimized for GPU-based throughput accelerators [1]. Significant similarities and differences exist between the two efforts. Both designs address the needs of thread-rich architectures characterized by a memory-resident data working set and a many-to-few-to-many traffic pattern. But whereas workloads running on throughput accelerators are shown to be insensitive to NOC latency, we show scale-out workloads to be highly sensitivity to interconnect delays due to frequent instruction fetches from the LLC. As a result, NOC-Out innovates in the space of delay-optimized on-chip topologies, whereas prior work has focused on throughput and cost in the context of meshes.

One effort aimed at boosting NOC efficiency specifically in the context of server processors was CCNoC, which proposed a dual-mesh interconnect with better cost-performance characteristics than existing multi-network alternatives [20]. Our work shows that mesh-based designs are sub-optimal from a performance perspective in many-core server processors.

A number of earlier studies sought to reduce NOC area cost and complexity through microarchitectural optimizations in crossbars [12, 21], buffers [17], and links [16]. A recent study examined challenges of NOC scalability in kilo-node chips and proposed an interconnect design that co-optimized buffering, topology, and flow control to reduce NOC area and energy [7]. All of these efforts assume a conventional tiled organization. In contrast, our NOC-Out design lowers NOC area overheads by limiting the extent of on-die connectivity. However, NOC-Out's efficiency can be further improved by leveraging many of the previously proposed optimizations.

Finally, Huh et al. preceded NOC-Out in proposing a segregated NUCA CMP architecture in which core and LLC tiles are disjoint [9]. Our design is different from Huh's in that it seeks to reduce the number of cache tiles to lower network cost, whereas Huh relied on a sea of cache tiles to optimize data placement and partitioning.

8. Conclusion

Server processors for scale-out workloads require many cores to maximize performance per die by exploiting request-level parallelism abundant in these workloads. Standing in the way of seamless performance scale-up resulting from additional cores is the on-die interconnect that adds delay on instruction fetches serviced by the last-level cache. The performance penalty is particularly acute in mesh-based networks that require a large number of router traversals on a typical LLC access. While a low-diameter flattened butterfly topology overcomes the performance bottleneck of meshes, it incurs a high area overhead through the use of many-ported routers and repeater-intensive long-range links.

This work introduced NOC-Out, a processor organization tuned to the demands of scale-out workloads. NOC-Out segregates LLC banks from core tiles and places the cache in the center of the die, naturally accommodating the bilateral core-to-cache data access pattern characteristic of scale-out workloads. With the bulk of the traffic flowing to the LLC and directly back to the cores, NOC-Out simplifies the interconnect by restricting direct connectivity among the cores. NOC-Out farther improves network cost and latency characteristics through the use of simple tree topologies that take advantage of the bilateral traffic pattern between the cores and the LLC. Finally, NOC-Out optimizes the intra-LLC interconnect by reducing the number of LLC tiles for a fixed cache capacity with respect to the conventional tiled design. The combination of these optimizations enable a low-cost low-latency interconnect fabric that matches the performances of a flattened butterfly at the cost of a mesh.

9. Acknowledgments

This work was partially supported by EuroCloud, Project No 247779 of the European Commission 7th RTD Framework Programme – Information and Communication Technologies: Computing Systems.

References

[1] A. Bakhoda, J. Kim, and T. M. Aamodt, "Throughput-Effective On-Chip Networks for Manycore Accelerators," in *International Symposium on Microarchitecture*, December 2010, pp. 421–432.

[2] J. D. Balfour and W. J. Dally, "Design Tradeoffs for Tiled CMP On-Chip Networks," in *International Conference on Supercomputing*, June 2006, pp. 187–198.

[3] "CloudSuite 1.0," 2012. [Online]. Available: http://parsa.epfl.ch/cloudsuite

[4] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi, "Clearing the Clouds: A Study of Emerging Scale-Out Workloads on Modern Hardware," in *International Conference on Architectural Support for Programming Languages and Operating Systems*, March 2012, pp. 37–48.

[5] "Global Server Hardware Market 2010-2014," March 2011. [Online]. Available: http://www.technavio.com/content/global-server-hardware-market-2010-2014

[6] B. Grot, D. Hardy, P. Lotfi-Kamran, B. Falsafi, C. Nicopoulos, and Y. Sazeides, "Optimizing Data-Center TCO with Scale-Out Processors," *IEEE Micro*, vol. 32, no. 5, pp. 52–63, September/October 2012.

[7] B. Grot, J. Hestness, S. W. Keckler, and O. Mutlu, "Kilo-NOC: A Heterogeneous Network-on-Chip Architecture for Scalability and Service Guarantees," in *International Symposium on Computer Architecture*, June 2011, pp. 268–279.

[8] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki, "Reactive NUCA: Near-Optimal Block Placement and Replication in Distributed Caches," in *International Symposium on Computer Architecture*, June 2009, pp. 184–195.

[9] J. Huh, C. Kim, H. Shafi, L. Zhang, D. Burger, and S. W. Keckler, "A NUCA Substrate for Flexible CMP Cache Sharing," in *International Conference on Supercomputing*, June 2005, pp. 31–40.

[10] "International Technology Roadmap for Semiconductors (ITRS), 2011 Edition." [Online]. Available: http://www.itrs.net/Links/2011ITRS/Home2011.htm

[11] A. Kahng, B. Li, L.-S. Peh, and K. Samadi, "ORION 2.0: A Fast and Accurate NoC Power and Area Model for Early-Stage Design Space Exploration," in *Design, Automation, and Test in Europe*, April 2009, pp. 423–428.

[12] J. Kim, "Low-Cost Router Microarchitecture for On-Chip Networks," in *International Symposium on Microarchitecture*, December 2009, pp. 255–266.

[13] J. Kim, J. Balfour, and W. Dally, "Flattened Butterfly Topology for On-Chip Networks," in *International Symposium on Microarchitecture*, December 2007, pp. 172–182.

[14] P. Lotfi-Kamran, M. Ferdman, D. Crisan, and B. Falsafi, "TurboTag: Lookup Filtering to Reduce Coherence Directory Power," in *International Symposium on Low Power Electronics and Design*, August 2010, pp. 377–382.

[15] P. Lotfi-Kamran, B. Grot, M. Ferdman, S. Volos, O. Kocberber, J. Picorel, A. Adileh, D. Jevdjic, S. Idgunji, E. Ozer, and B. Falsafi, "Scale-Out Processors," in *International Symposium on Computer Architecture*, June 2012, pp. 500–511.

[16] G. Michelogiannakis, J. Balfour, and W. Dally, "Elastic-Buffer Flow Control for On-Chip Networks," in *International Symposium on High-Performance Computer Architecture*, February 2009, pp. 151–162.

[17] T. Moscibroda and O. Mutlu, "A Case for Bufferless Routing in On-Chip Networks," in *International Symposium on Computer Architecture*, June 2009, pp. 196–207.

[18] N. Muralimanohar, R. Balasubramonian, and N. Jouppi, "Optimizing NUCA Organizations and Wiring Alternatives for Large Caches with CACTI 6.0," in *International Symposium on Microarchitecture*, December 2007, pp. 3–14.

[19] "Tilera TILE-Gx." [Online]. Available: http://www.tilera.com/products/TILE-Gx.php

[20] S. Volos, C. Seiculescu, B. Grot, N. Khosro Pour, B. Falsafi, and G. De Micheli, "CCNoC: Specializing On-Chip Interconnects for Energy Efficiency Cache-Coherent Servers," in *International Symposium on Networks-on-Chips*, May 2012, pp. 67–74.

[21] H. Wang, L.-S. Peh, and S. Malik, "Power-driven Design of Router Microarchitectures in On-chip Networks," in *International Symposium on Microarchitecture*, December 2003, pp. 105–116.

[22] T. Wenisch, R. Wunderlich, M. Ferdman, A. Ailamaki, B. Falsafi, and J. Hoe, "SimFlex: Statistical Sampling of Computer System Simulation," *IEEE Micro*, vol. 26, no. 4, pp. 18–31, July/August 2006.

2012 IEEE/ACM 45th Annual International Symposium on Microarchitecture

SLICC: Self-Assembly of Instruction Cache Collectives for OLTP Workloads

Islam Atta[†] Pınar Tözün[‡] Anastasia Ailamaki[‡] Andreas Moshovos[†]

[‡]École Polytechnique Fédérale de Lausanne
{pinar.tozun, anastasia.ailamaki}@epfl.ch

[†]University of Toronto
{iatta, moshovos}@eecg.toronto.edu

Abstract

Online transaction processing (OLTP) is at the core of many data center applications. OLTP workloads are known to have large instruction footprints that foil existing L1 instruction caches resulting in poor overall performance. Prefetching can reduce the impact of such instruction cache miss stalls; however, state-of-the-art solutions require large dedicated hardware tables on the order of 40KB in size.

SLICC is a programmer transparent, low cost technique to minimize instruction cache misses when executing OLTP workloads. SLICC migrates threads, spreading their instruction footprint over several L1 caches. It exploits repetition within and across transactions, where a transaction's first iteration prefetches the instructions for subsequent iterations or similar subsequent transactions. SLICC reduces instruction misses by 58% on average for TPC-C and TPC-E, thereby improving performance by 68%. When compared to a state-of-the-art prefetcher, and notwithstanding the increased storage overheads (42× as compared to SLICC), performance using SLICC is 21% higher for TPC-E and within 2% for TPC-C.

1. Introduction

Online transaction processing (OLTP) is a multi-billion dollar industry that increases 10% annually [7]. OLTP needs have been driving innovations by both database management system and hardware vendors, and OLTP performance has been a major metric of comparison across vendors [11, 31, 32]. Unfortunately, modern cloud and server infrastructures are not tailored well for the characteristics of OLTP applications [4]. Literature shows that OLTP workloads are memory bound; memory access stalls account for 80% of execution time, most of which are due to first-level instruction cache misses [15, 4, 28]. Software [9] and hardware [14, 26, 3, 5] efforts are trying to alleviate stall time related to instruction misses.

Transactions of canonical OLTP systems are randomly assigned to worker threads, each of which usually runs on one core of a modern multi-core system. The instruction footprint of a typical transaction does not fit into a single L1-I cache, thus thrashing the cache and incurring a high instruction miss rate. Although L2 and L3 caches are growing in size, today's technology and CPU clock cycle constraints prevent deploying L1-I caches larger than 32KB. As this work demonstrates, the instruction footprint of a typical OLTP transaction fits comfortably in the aggregate L1-I cache capacity of modern many-core chips. Provided that there is sufficient code reuse, spreading the footprint of transactions over multiple L1-I caches would reduce instruction cache misses. Fortunately, as corroborated by our experimental results, OLTP workloads exhibit a high-degree of instruction reuse both within a transaction and across concurrently running transactions [3, 9].

This paper proposes *SLICC* (Self-Assembly of Instruction Cache Collectives), a hardware technique that utilizes thread migration to minimize instruction misses for OLTP workloads. SLICC divides the instruction footprint of a transaction into smaller code segments and spreads them over multiple cores, so that each L1-I cache holds part of the instruction footprint. As part of this process the L1-I caches self-assemble to form a *collective* that reduces the instruction misses for this transaction and other similar ones. SLICC exploits intra- and inter-thread instruction locality in two orthogonal ways: (1) A thread looping over multiple code segments spread over multiple caches observes a lower miss rate (as opposed to a conventional system in which each segment would evict the others from the cache), thereby avoiding thrashing. (2) A preamble thread effectively prefetches and distributes common code segments for subsequent threads, thereby reducing the total miss rate. As execution progresses, old cache collectives are naturally disassembled and new ones are formed to hold the footprints of new transactions.

As opposed to previous OLTP instruction miss reduction techniques, SLICC is a hardware solution, that avoids undesirable instrumentation, utilizes available core and cache capacity resources, covers user as well as system-level code, and requires no changes to the existing user code or software system. SLICC incurs overheads due to thread migration; thus, context switching and increases in data misses must be amortized to improve performance. A hardware thread migration mechanism provides a programmer transparent solution that has low context switching overheads, and the positive impact of the reduction in instruction misses outpaces the extra data misses.

To evaluate SLICC, we execute two popular transactional benchmarks, TPC-C [29] and TPC-E [30], as well as a MapReduce [4] cloud workload. Our experiments show that, on average, thread migration eliminates 56% of the L1 instruction misses resulting in a 68% overall performance improvement over the baseline (described in Section 5.1). Compared to PIF [5], a state-of-the-art instruction prefetcher, SLICC improves performance by 21% for TPC-E and comes within 2% for TPC-C, with only 2.4% of relative storage area overhead. SLICC is also robust as it does not affect the performance of MapReduce [4], a cloud workload, which has a relatively small instruction footprint. In summary, this paper makes the following contributions:

- It characterizes the memory behavior of TPC-C [29] and TPC-E [30] showing that transactions suffer from instruction misses, and that their instruction streams exhibit intra- and inter-transaction recurring patterns leading to eviction of useful blocks that are re-accessed (96% of capacity misses are for instructions).
- It demonstrates that recently proposed cache replacement policies [24, 12] reduce instruction misses by 8% on average for the best policy, but leave ample room for improvement.
- It presents SLICC, a hardware thread migration algorithm, and shows that it reduces instruction misses by 56% on average with an overall 68% performance improvement for OLTP.

The remaining of this document is organized as follows. Section 2 analyzes the nature of the problem and Section 3 sets the requirements for an ideal solution. Section 4 describes the automated thread

1072-4451/12 $26.00 © 2012 IEEE
DOI 10.1109/MICRO.2012.26

IEEE computer society

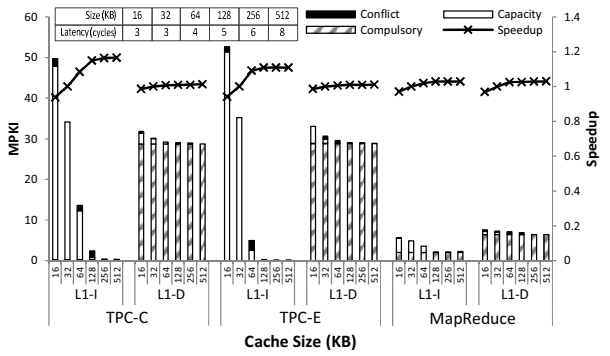


Figure 1: Instruction and data L1 misses and relative performance as a function of cache size.

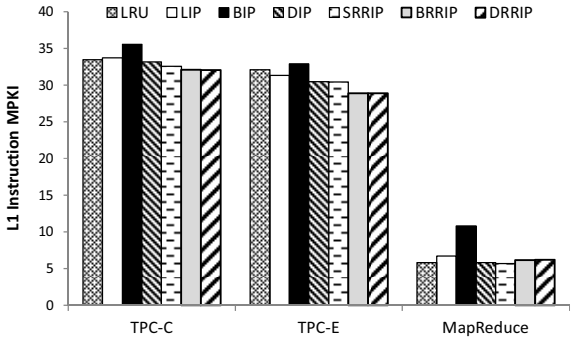


Figure 2: MPKI with different cache replacement policies.

migration algorithm SLICC. Section 5 demonstrates experimentally the performance benefits of SLICC. Sections 6 reviews related work, while Section 7 presents our conclusions.

2. Instruction MISS Analysis

This section examines the OLTP memory behavior that motivates thread migration for instruction cache miss reduction. The analysis targets TPC-C [29] and TPC-E [30], which resemble state-of-the-art commercial OLTP applications. We contrast their behavior with MapReduce [4], a data center workload which has a smaller instruction footprint. We find that for OLTP transactions:

1. Most instruction misses are due to limited cache capacity, whereas most data misses are compulsory.
2. The instruction footprint of most transactions would fit in the aggregate L1 instruction cache capacity of even small scale chip multiprocessors (eight cores). The same is not true for data footprints.
3. Existing non-LRU cache replacement policies reduce the instruction miss rate, but only by a fraction of what would be possible with larger caches.
4. There is intra-thread locality, but over code regions that are larger than a typical L1 cache size.
5. There is significant inter-thread locality, particularly across threads of the same transaction type.

2.1. OLTP Instructions and Data Misses

In typical multi-threaded OLTP systems, a transaction is assigned to a worker thread. Thus multiple similar transactions (threads) usually run concurrently. Individual threads, whose memory footprints do not fit in the L1 cache, suffer from high miss rates. Ferdman *et al.* show that in OLTP, memory stalls account for up to 80% of the execution time [4], while Tözün *et al.* show that instruction stalls account for 70–85% of the overall stall cycles [28].

2.1.1. L1 Miss Breakdown We further analyze instruction and data L1 misses. Figure 1 shows the number of misses per kilo-instructions (MPKI) for a range of L1 instruction (L1-I) and data (L1-D) cache sizes. Section 5 details the experimental methodology. We first vary the L1-I cache size (16KB–512KB) while keeping the L1-D cache size at 32KB (our baseline), and then we vary the L1-D cache size while keeping the L1-I at 32KB. CACTI 6 [20] is used to model the access latencies of different cache sizes. Figure 1 shows a breakdown of instruction and data L1 misses into three categories: capacity, conflict and compulsory [10]. By identifying where most misses come from, we highlight the reasons behind the memory stalls; is it cache size, associativity, or cold misses?

Figure 1 shows that for OLTP workloads, capacity misses dominate instruction misses. This implies that the instruction footprint does not fit in the cache and has lots of reuse; cache blocks are evicted from the cache before they are re-referenced. Hence larger L1-I caches, which can hold cache blocks for longer periods, can reduce instruction misses. To keep up with the CPU clock speeds, technology constraints have limited the sizes of L1 instruction caches to about 32KB today. With 32KB caches, instruction capacity misses are an order of magnitude more than data capacity misses. Compulsory misses, which occur for the first reference to each unique cache block, dominate data misses. Thus, larger data caches can do little to reduce data misses. For 32KB caches, compulsory data misses are an order of magnitude more than compulsory instruction misses. Unless data is prefetched, data misses cannot be reduced.

MapReduce is a cloud workload featuring a relatively smaller instruction footprint [4]. Since 71% of the total L1 misses are compulsory for 32KB caches, larger L1 instructions or data caches are not as beneficial.

Figure 1 also shows overall performance improvement normalized to the 32KB baseline. Performance improvements with larger L1-D caches are negligible at 1%, but can be as high as 16% with larger L1-I caches (MapReduce shows less than 3% improvement). If increasing cache size did not also increase latency, performance improvements would be higher; for example, a 512KB L1-I with the latency of a 32KB L1-I would result in a 61% performance improvement for TPC-C.

We conclude that (a) OLTP transactions have instruction footprints that, while larger than typical L1-I caches, could fit in the aggregate L1-I capacity of modern multi-cores. (b) OLTP data footprints are much larger and cannot fit onto the aggregate L1-D capacity of modern multi-cores. Additionally, (c) OLTP instruction streams exhibit a significant reuse over regions that exceed typical L1-I cache sizes leading to the eviction of useful blocks that are re-accessed.

2.1.2. Replacement Policies Conventional caches often use some approximation of LRU replacement. Qureshi *et al.* show that some workloads, including those that have long-term reuse, are not LRU-friendly [24]. For such workloads, LRU cycles through a large footprint while it would be best to keep at least some part of the footprint cache resident. They modify LRU by introducing new static (LIP, BIP) and dynamic (DIP) insertion policies where newly accessed blocks are not necessarily inserted at the most-recently-used position of the LRU stack. Jaleel *et al.* propose the SRRIP, BRRIP, and DRRIP re-reference interval based insertion policies [12]. Their Re-reference Insertion Prediction (RRIP) chain represents the order in which blocks are predicted to be re-referenced. The block at the

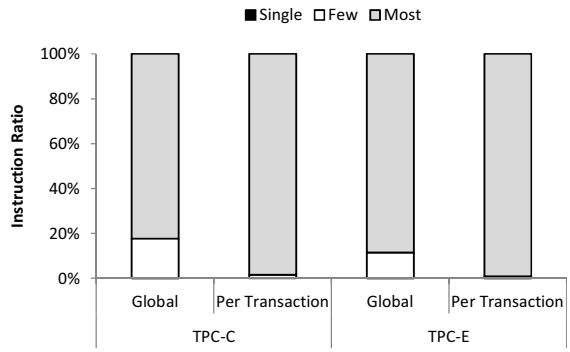


Figure 3: Breakdown of accesses accordingly to instruction block reuse.

head of the RRIP chain is predicted to have a *near-immediate* re-reference interval, while the block at the tail of the RRIP chain is predicted to have a *distant* re-reference interval. On a cache miss, a *distant* block is replaced. Re-references to a block promote its position towards the head of the chain.

Figure 2 reports the MPKI for these replacement policies for the baseline 32KB L1-I cache. BRRIP and DRRIP perform best reducing misses by an average of 8% over LRU. This reduction is only a fraction of what is possible with larger caches as Figure 1 showed.

Thrashing applications favor Bimodal RRIP (BRRIP), which predicts a *distant* re-reference interval for most blocks [12]. Dynamic RRIP (DRRIP) selects the best policy from Static RRIP (SRRIP) and BRRIP at runtime. Figure 2 shows that DRRIP chose BRRIP most of the time. Contrary to the Least Insertion Policy (LIP), which promotes a referenced block to the MRU position, BRRIP uses access frequency to promote cache blocks gradually towards the head of the chain.

Thus, we see that the recurring patterns exhibited by OLTP instruction streams have relatively long periods that cannot be fully captured by existing insertion/replacement policies. Nevertheless, replacement policies are orthogonal to thread migration.

2.1.3. Redundancy Across Threads Chakraborty *et al.* profile OLTP instruction accesses and report an 80% redundancy across multiple cores for user and OS code [3]. Figure 3 corroborates these results and further shows that 98% of the instruction cache blocks are common among threads executing the same transaction type; although similar transactions do not follow the exact same control flow path, they have common code segments at a coarser granularity. Figure 3 shows a breakdown of all instruction cache accesses classified according to the reuse experienced by the accessed block over the duration of the application. The figure presents three coarse reuse categories: *single*, *few* and *most* that correspond to blocks accessed by only one thread, at most, or more than 60% of all the threads, respectively. This behavior highlights an opportunity to reduce instruction misses by exploiting temporal locality across multiple threads, particularly threads of the same transaction type.

3. Thread Migration for OLTP

SLICC exploits the aggregate L1-I cache capacity of many cores and the availability of multiple concurrent threads with inherent code commonality. It is a hardware transaction scheduling algorithm that spreads the instruction code footprint across multiple L1-I caches. SLICC dynamically pipelines and migrates threads to cores that are predicted to hold the code blocks to be accessed next. By migrating

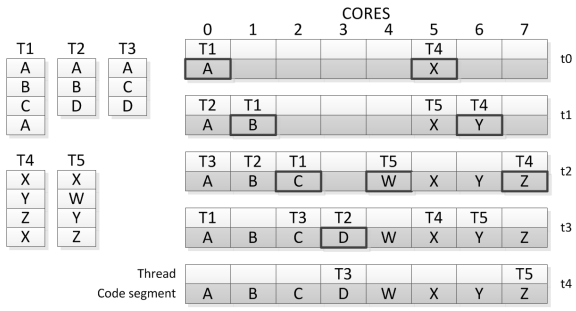


Figure 4: Thread migration common code segment reuse example. Left: T1-3 and T4-5 are threads running similar transactions. Right: 8-core system. The shaded area is the cache activity (thick border = warm-up phase).

threads, SLICC virtually increases the cache capacity observed by a thread, and thus, the reuse of instruction blocks brought in to the cache, avoiding thrashing.

The observations of Section 2.1 support SLICC's approach: (a) it can avoid many instruction misses by virtually increasing the L1-I cache size per thread; and (b) it can effectively increase locality by grouping similar transactions together. SLICC tends to increase intra- and inter-thread instruction locality.

3.1. Example Scenario

Figure 4 exemplifies how thread migration can reduce instruction misses. Threads T1-T5 are scheduled to run on an 8-core system, where T1-T3 and T4-T5 execute respectively transactions of the same type. The transactions' footprints are divided into code segments, where each *segment* fits in the L1-I cache of a single core, but two segments would not fit together. T1 executes the following code segments in order: A-B-C-A. Thus, its instruction footprint is 3× larger than the L1-I cache size. Since T2 and T3 are of the same type as T1, they share common segments with T1, but their execution paths are not identical. A conventional system would schedule T1, T2, and T3 on separate cores, and since their footprints are larger than the L1-I cache size, each thread would suffer all instruction misses.

Figure 4 (right) demonstrates an ideal scenario for thread migration. Initially (at time t0), T1 runs on core-0. When it is done with code segment A, and so all cache blocks for A have been brought in to the cache, T1 migrates to core-1 (at t1), where it continues execution fetching cache blocks of segment B. At the same time, T2 can be scheduled to start execution on core-0 (at t1), ideally reusing all blocks in A (miss rate close to zero). We refer to this as inter-thread reuse. The process continues and T1 warms-up caches 1 and 2 with B and C, respectively. At t3, when T1 goes back to A, it migrates to core-0 benefiting from intra-thread reuse.

Migration is beneficial even if T1-T3 do not follow identical paths, as segment D illustrates. Since T1 did not touch segment D, T2 will suffer due to the corresponding misses while executing on core-3. If T3 follows suit on core-3, it will not suffer any instruction misses for segment D.

T4-T5 that access different code segments benefit as well if they get assigned to a different set of cores, avoiding conflicts with T1-T3. This process applies to all subsequent threads: if they touch a code segment that exists in some L1-I cache, they migrate to the corresponding core and avoid missing for these segments.

Without thread migration multiple requests would be repeatedly sent out for the same cache block from multiple cores. With thread migration, an instruction cache block is, under the ideal scenario, requested only once, and reused multiple times.

3.2. SLICC Requirements

Based on our preceding discussion we identify three requirements for SLICC. SLICC should dynamically detect: (a) *When* a thread should migrate, i.e., when is the current cache *full*; and (b) *where* the thread should migrate to, i.e., which remote cache, if any, holds the code segment the thread will touch next. Since transactions vary in their control flow, SLICC should (c) not impose any specific pipelining, i.e., it should not restrict similar threads to follow the exact same path.

In order to meet these requirements, SLICC needs to maintain runtime information about caches and individual threads to be able to make judicious migration decisions. First, SLICC needs a mechanism to determine whether the cache is filled-up with useful cache blocks or not. In addition, with respect to a given core, SLICC should be able to predict which remote core holds the cache blocks that will be touched next.

The basic SLICC design is a type-oblivious algorithm, i.e., no information is provided about which threads resemble the same transaction type. Information about thread types could enhance SLICC's process. Thus, there are several alternatives. On one extreme, the hardware can dynamically migrate threads to cores irrespective of their types. On the other extreme, the software layer can transfer knowledge about thread types. In between, threads can be pre-processed to detect threads with similar starting address ranges. We detail the three alternatives in Section 4.

3.3. Effect on Data Misses

When a thread migrates, it leaves data that might be reused behind. This may increase the data miss rate. Section 5 shows that while SLICC does increase the data miss rate, the benefit from reducing instruction misses outpaces the performance loss due to data miss increase.

Intuitively, depending on the workload, instruction misses can impact performance more than data misses. For example, modern architectures use instruction level parallelism (ILP) to hide data miss latencies. Instruction misses restrict instruction supply, rendering such ILP techniques less effective. Existing core architectures make no effort to balance the relative cost of the two types of misses. SLICC provides a way of balancing the relative costs of data vs. instruction misses. In Section 5 we show that SLICC reduces the overall L1 miss rate.

4. SLICC Design

SLICC exploits intra- and inter-thread locality. (1) It virtually increases the L1-I cache capacity observed by a thread; thus, it improves locality within a thread. (2) It pipelines similar threads, such that one thread fetches instruction cache blocks that are reused by many threads.

This work presents three different SLICC designs. The first design (SLICC) is transaction-type-oblivious, while the other two exploit transaction type information. Given that threads of the same transaction type tend to have similar footprints, knowing each thread's transaction type can lead to better migration decisions. The transaction type information is either provided by the software (SLICC-SW), or detected at runtime by the hardware based on the initial instruction sequence each thread executes (SLICC-Pp). These implementations represent the two extremes and an in-between solution in terms of hardware/software co-operation. Future work may look at other alternatives.

4.1. Transaction-Type-Oblivious SLICC

SLICC is a dynamic hardware thread scheduling and migration algorithm that is programmer transparent. SLICC attempts to partition on-the-fly the instruction footprint of transactions into several segments where each segment fits in the L1-I cache, but two segments do not fit together. Ideally: (1) a thread will migrate to another core when it starts touching a different segment, and (2) the destination core will already have the segment cached.

Figure 5 shows the sequence of events that lead to thread migration. In the steady state, each core has a running thread and a hardware queue of waiting threads. Using a naïve load-balancing strategy, newly arrived threads are scheduled to the least congested core (i.e., the core with the least number of waiting threads). A SLICC agent at each core continuously monitors execution locally in order to determine whether (Q.1) the local cache is filled-up with useful instruction blocks, if so, (Q.2) whether these blocks are useful to the current thread and for how long, and (Q.3) where to migrate to if needed.

(Q.1) Is the cache full with useful blocks? As a thread starts executing on a core it may experience many misses. If the cache contains a segment that may be useful for other threads, it is best to migrate the current thread to another core. Otherwise, it is best to allow the current thread to load a new segment in the cache. SLICC uses a "cache full" detection heuristic to make this decision. Initially, all caches are "empty". To detect whether a cache has been filled up with a *segment*, SLICC counts the number of misses using a resettable, saturating miss counter (*MC*) local to each core. When the number of misses exceeds the threshold, *fill-up_t*, the cache is considered *full*. In the long run, all MCs will saturate, preventing new segments from being cached effectively due to premature thread migration. To create opportunities for loading new segments, SLICC resets the MC when the core's thread queue becomes empty. The currently cached blocks are not flushed, so if a subsequent thread requires the same segment it will still find it there. However, a thread touching a new segment will be given the opportunity to cache it.

(Q.2) Are the current cache contents useful to this thread and for how long? When running a thread on a *full* cache, SLICC tries to determine whether the thread is going over the cached segment, or whether it is about to move to a new segment. For this purpose SLICC measures *miss dilution*, that is, the recent frequency of misses (detailed in Section 4.2.2). If miss dilution is low, then SLICC predicts that thread is only temporarily diverting away from the cached segment. Since the thread will converge again soon, it is best to not migrate to benefit from the forthcoming instruction reuse. If miss dilution is high, then SLICC predicts that the thread is moving to a different segment. If it continues execution on this core it will evict useful cache blocks, which could be reused by other threads. SLICC predicts that it might be better to migrate the thread elsewhere. The question at this point becomes *where* to go?

(Q.3) Where to migrate to? Ideally, SLICC would migrate a thread to a cache that has the thread's next segment. SLICC attempts the following in order: (1) If the thread is going to touch a code segment that is available on another core, the thread migrates there.

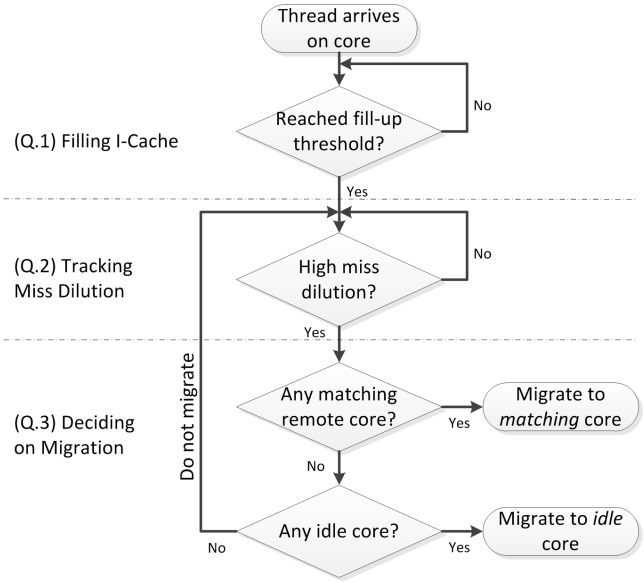


Figure 5: Thread Migration Algorithm.

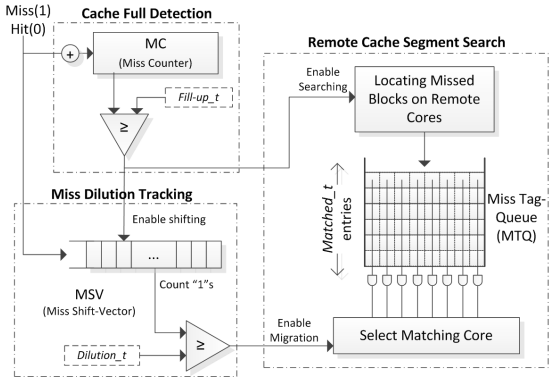


Figure 6: SLICC Architecture.

(2) Otherwise, the thread migrates to an idle core, if any. (3) The thread stays put. In the last case, migrating the thread would incur overheads and would evict remotely cached segments that may be useful for other threads. SLICC opts for incurring the instruction misses locally avoiding the migration overhead.

To detect which, if any, remote cache has the next segment, SLICC uses a short sequence of *matched_t* number of tags of recent misses, predicting that they form the preamble of the next segment. Conceptually, once SLICC decides to try to migrate a thread, it searches all remote L1-I caches for these recently missed tags. Section 4.2.3 explains how this search can be implemented including an incremental method that uses the existing coherence protocol responses.

Figure 5 summarizes the execution stages of a thread on a core until it migrates, or completes execution.

4.2. Implementation Requirements

Figure 6 shows that SLICC's implementation comprises: (a) a cache full detector, (b) a miss dilution tracker, and (c) a remote cache segment search unit. SLICC uses hardware thread migration, and thus, interacts with the OS as Section 4.4 explains in more detail. The three aforementioned units, described subsequently, track all cache accesses, including speculative ones.

4.2.1. Cache Full Detection A $log_2(L1I\ cache\ blocks)$ wide saturating miss counter (*MC*) continuously counts the number of misses. When MC saturates at a value of *fill-up_t* SLICC assumes that the cache has now captured a full segment and may trigger migrations accordingly. We experimentally found that using a value in the order of $\frac{cache\ size}{2}$ for the *fill-up_t* threshold works reasonably well, with little sensitivity to the exact value of this parameter. Other fill-up detection mechanisms may be possible but are beyond the scope of this paper.

4.2.2. Miss Dilution Tracking It is not always beneficial to migrate threads immediately after a cache becomes full or when a thread incurs a few misses. SLICC must predict whether the thread is only temporarily diverging due to conditional control flow or whether it is moving to a completely different segment. Furthermore, since threads have to miss for a few blocks before migrating (*matched_t* tags must be located on a remote cache), a few useful cache blocks may be evicted, creating gaps in the exiting segment and causing a corresponding number of misses for subsequent threads. Finally, a thread may immediately loop back to the same code segment or may temporarily follow a somewhat different path after being selected for migration.

SLICC handles these cases by considering the frequency of instruction misses; it restricts migration to the cases when a thread starts to miss more frequently. If the thread is moving to a new segment, it will incur more misses than hits. SLICC counts the number of misses in a window of recent accesses. When this count is above the dilution threshold, *dilution_t*, migration is enabled. The miss shift-vector (*MSV*) is a 100-bit FIFO shift vector recording the hit/miss history for the last 100 cache accesses (enabled when cache is filled-up). A logic-0 and logic-1 represent a cache hit and miss, respectively. When the number of logic-1 bits reaches a threshold (*dilution_t*), SLICC enables migration. SLICC resets the *MSV* with every migration.

4.2.3. Remote Cache Segment Search When SLICC decides to migrate a thread it has to determine which cache, if any, contains the segment the thread is executing. To do so, SLICC records recently missed tags in the Missed Tag Queue (*MTQ*), which is a *matched_t* entry FIFO of *n*-bit entries, where *n* is the number of cores. A logic-1 on bit index *C* for MTQ entry *i* indicates that the *i*th recently missed cache block was cached at core *C*. Thus, by ANDing all bits at index *C* we know whether core *C* holds all the recently missed cache blocks. This information does not have to be exact or accurate, since it is used by a prediction mechanism. SLICC gathers this information *incrementally* as misses occur and stores it in the MTQ. The remote cache segment search is distributed and the decision is made locally by the core we migrate from. A directory coherence protocol could report the complete or partial sharing vector for misses that are tracked by the MTQ.

Alternatively, or if the coherence protocol is snoop-based, SLICC could broadcast the missed tags as they occur and explicitly request that remote cores identify themselves. On snoop coherence systems, these requests can piggyback on the existing snoop requests. Searching remote L1-I caches requires extra bandwidth on the remote caches that is proportional to the number of missed tags and cores.

To avoid this bandwidth overhead, we use an approximate cache signature in the form of a partial-address bloom filter that supports evictions [23]. When the index size of the bloom filter is larger than the cache set index, collisions occur only within sets. Hence on evictions, only the set of the evicted block is checked for collisions. Every core maintains such a filter, representing a superset of the

currently cached blocks. In this design, once migration is triggered, remote-cache search requests are answered by the approximate signature, avoiding contention with the original cache references of the remote core. In Section 5.3, we evaluate the tradeoff of the bloom filter's accuracy versus its size. We find that for a 32KB cache, a 256B bloom filter is sufficient.

If no matching remote cache is found, SLICC will attempt to find an idle core. SLICC either broadcasts a request for idle cores to report, or piggy-backs this information on the responses received during the miss tag search phase. Thread migrations are relatively infrequent (every 3.2K instructions on average), reducing the relative overhead of remote cache segment and idle core searching.

4.3. Exploiting Transaction Type Information

Section 2.1.3 showed that the instruction footprint overlap is higher among threads of the same transaction type. The basic SLICC does not directly exploit this phenomenon. It tries to detect, on-the-fly, whether a thread matches the segment on the core it is currently executing. Thus, a thread of type *X* may partially kick-out cache blocks used by threads of type *Y*. If the transaction type for each thread were known, SLICC could schedule similar threads on the same set of cores to reduce conflicts. We propose two SLICC variants that exploit such thread transaction type information.

4.3.1. Assigning Transaction Types *SLICC-SW* relies on the OLTP software layer to annotate each thread upon launch with a transaction type. This guarantees correctness, but requires some modifications to the software/hardware interface.

Alternatively, *SLICC-Pp* uses a hardware preprocessing phase to assign types to threads as they launch. SLICC-Pp exploits the observation that in OLTP the first few instructions executed are the same for same-type threads, while they differ across different-type threads. SLICC-Pp only needs to know when a new thread is launched. A middle-ware layer assigns threads in groups to a core devoted for this purpose (*scout core*). There, each thread executes a few tens of instructions, while the instruction addresses are hashed. The resulting values are used as thread type identifiers. Experiments show that SLICC-Pp is 100% accurate when executing a small number of instructions. SLICC-Pp dedicates one core for pre-processing.

4.3.2. Type-Aware Migration Using thread type information, SLICC groups similar threads into teams. Creating teams is useful for two reasons: (1) it groups similar transactions to improve opportunities for co-scheduling and overlap, and (2) it helps scheduling reduce waiting times. For each thread SLICC records a unique numerical ID, a type ID, and an arrival timestamp. The timestamp of a team is that of its oldest thread. The oldest team is scheduled, without pre-emption if possible.

We intuitively design a scheduling algorithm that maximizes the core utilization and reduces the queuing delay of threads. Team sizes differ and for an *N*-core architecture we categorize them into *large* ($1.5\times$ to $2\times$ *N* threads), *medium* ($0.5\times$ to $1.5\times$ *N* threads), and *small* (less than $0.5 \times N$ threads) teams. Cores are time-multiplexed among teams. When large teams are scheduled, they are allowed to execute on all cores. Medium size teams are limited to half the resources ($0.5 \times N$ cores). Threads of a small team are treated as stray threads, and are not grouped. Rather, stray threads are scheduled, individually, to idle cores, or in parallel with a medium team. For SLICC-SW and SLICC-Pp, when a team of threads completes execution, SLICC resets all *MC*s, *MTQ*s and *MSV*s.

Table 1: Workload Parameters.

TPC-C-1	1 warehouse, 84 MB Wholesale supplier
TPC-C-10	10 warehouses, 1 GB Wholesale supplier
TPC-E	1000 customers, 20 GB Brokerage house
MapReduce	Hadoop 0.20.2, Mahout 0.4 library Wikipedia page articles (12 GB)

4.4. Support for Thread Migration

To allow for queuing threads, the thread migration performed in SLICC transfers architectural register files as in Thread Motion [25]. The thread's context is saved in the L2 cache closest to the target core and is then retrieved at the target core. This minimizes the set-up time for the thread. Since modern commercial processor technologies (e.g., Intel Virtualization (VT) [33] and AMD Secure Virtual Machine (SVM) [1]) provide hardware support for thread migration, minimal modifications are required to make the migration process transparent to higher software layers.

Canonical OS kernels are responsible for assigning threads to cores. Hardware support for thread migration that is transparent to higher layers avoids any software overhead. Otherwise, the OS scheduler must be informed about these migrations. An alternative is a hybrid system in which hardware mechanisms provide counters and migration acceleration, while leaving the policy choice to software. This enables easier integration between existing schedulers and platforms with virtualization support.

5. Evaluation

Our evaluation: (1) Studies the configuration thresholds for SLICC (Section 5.2). (2) Determines the trade-off between bloom filter size and remote cache segment search accuracy (Section 5.3). (3) Demonstrates SLICC's effect on instruction (Section 5.4) and data misses (Section 5.5), compared to the baseline. (4) Reports the performance improvement with the different flavors of SLICC compared to the baseline and to a state-of-the-art instruction prefetcher, PIF [5] (Section 5.6). (5) Estimates the HW cost for SLICC's components (Section 5.7). (6) Reports statistics about remote cache segment search activity (Section 5.8).

5.1. Methodology

Current operating systems do not support thread migration at the hardware level. The OS kernel assumes full control over thread assignment in multicore environments. To work around this limitation, we extract x86 execution traces using PIN [18], which are annotated to identify transactions. We then replay traces, modeling the timing of all events and maintaining the original thread sequence. We modify the Zesto x86 multicore architecture simulator [17]. Previous work shows that migrating threads to a set of dedicated cores to execute system level code improves performance [3]. While this work studies migration of user-level code, SLICC is generic and can apply to system level code as well. We model thread migrations by injecting writes and reads for all architectural state and thread context information.

We examine one scale-out workload and two server workloads as described in Table 1 [4]. TPC-C [29] and TPC-E [30] run on top

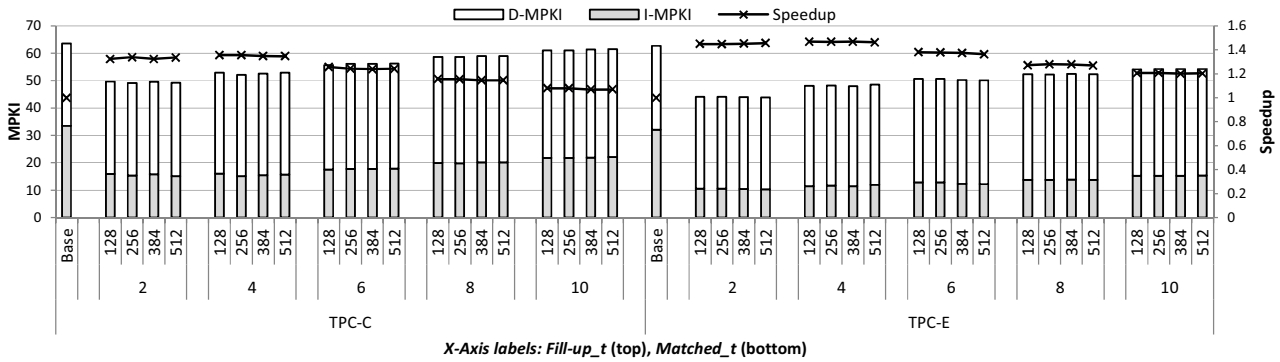


Figure 7: MPKI and relative performance as a function of *fill-up_t* and *matched_t* thresholds.

Table 2: System Parameters.

Processing Cores	16 OoO cores, 2.5GHz 6-wide Fetch/Decode/Issue 128-entry ROB, 80-entry LSQ BTAC (4-way, 512-entry) TAGE (5-tables, 512-entry, 2K-bimod)
Private L1 Caches	32KB, 64B blocks, 8-way 3-cycle load-to-use, 32 MSHRs MESI-coherence for L1-D
L2 NUCA Cache	Shared, 1MB per core, 16-way 64B blocks, 16 banks 16-cycle hit latency, 64 MSHRs
Interconnect	4×4 2D Torus, 1-cycle hop latency
Memory	DDR3 1.6GHz, 800MHz Bus, 42ns latency 2 Channels / 1 Rank / 8 Banks 8B Bus Width, Open Page Policy t_{CAS}-10, t_{RCD}-10, t_{RP}-10, t_{RAS}-35 t_{RC}47.5, t_{WR}-15, t_{WTR}-7.5 t_{RTRS}-1, t_{CCD}-4, t_{CWD}-9.5

of the scalable open-source storage manager Shore-MT [13]. The client-driver and the database are kept on the same machine, the buffer-pool is set big-enough to keep the whole database in memory, and due to the unavailability of a sufficiently fast I/O subsystem we flush the log to RAM. We simulate 1K tasks or approximately 1.1B instructions. We use two different databases in TPC-C-1 and TPC-C-10 to demonstrate that SLICC remains effective even with a larger database. TPC-C and TPC-E have larger instruction and data footprints compared to other scale-out workloads [4]. To demonstrate SLICC's robustness, we study the MapReduce CloudSuite workload [22], which does not have a large instruction footprint [4]. MapReduce divides the full input dataset across 300 threads, each performing a single map/reduce task. We focus most of our evaluation on TPC-C-1 (referred as TPC-C) and TPC-E.

Table 2 details the baseline architecture. We use misses per kilo instructions (MPKI) as our metric for instruction (I-MPKI) and data (D-MPKI) misses. We measure performance by counting the number of cycles it takes to execute all transactions. With *N*-core, our baseline architecture can run up to *N* concurrent threads with the OS making thread scheduling decisions. SLICC manages a thread pool of up to *2N* threads. Unless otherwise indicated, all SLICC results are for the SLICC-SW configuration – i.e., the SW layer transfers knowledge about thread types to the HW layer.

5.2. Exploring SLICC's Parameter Space

SLICC utilizes three thresholds to make thread migration decisions: *fill-up_t*, *matched_t* and *dilution_t*. This section explores their effect on L1 cache misses and overall performance. As defined in Section 4, *fill-up_t* sets the threshold for the initial fill-up period for an L1-I cache, during which instructions are brought in until the cache is almost *full*. When the miss counter (*MC*) is lower than *fill-up_t*, a thread is not allowed to migrate. *Matched_t* sets the minimum number of tags that should be found on a remote cache before a thread migrates to it. Larger *matched_t* limits migration, while smaller values trigger too frequent migrations. *Dilution_t* is the minimum number of misses in the last 100 accesses to allow migration. It tends to restrict migration to the cases when more frequent misses are observed by a thread. The parameter choices could be thought of as a 3D space. To simplify, we first keep *dilution_t* value at zero, and explore the parameter space of *fill-up_t* and *matched_t*. In addition, we assume zero-overhead to search for remote tags. We later model an actual search mechanism.

Figure 7 reports I-MPKI, D-MPKI and performance relative to the baseline as a function of *fill-up_t* and *matched_t*. The *fill-up_t* values shown correspond to fractions of the L1-I cache capacity (512 cache blocks): ¼, ½, ¾, and *one*. The *matched_t* range shown is $2-10$; larger *matched_t* values further degrade performance. SLICC reduces instruction misses and increases data misses. Since instruction stalls, for OLTP workloads, account for 70% of overall cycle stalls [28], reducing instruction misses has a major effect on performance.

The results show that SLICC is not sensitive to different values of *fill-up_t*. *Fill-up_t* is actually a proxy for warming-up the caches; it affects only the first migration from a core. Thus with more migrations, the effect of *fill-up_t* diminishes. TPC-C and TPC-E transactions have large instruction counts and migrations. Figure 7 demonstrates that for *matched_t* values larger than four, performance benefits drop. On the other hand, although the overall MPKI at two is lower, performance at four is higher due to fewer migrations, and thus, lower overhead.

Next, we explore the parameter space of *dilution_t*. Using a small value for *dilution_t* triggers more frequent migrations. Using too large a value for *dilution_t* reduces migration overhead, but with a possible I-MPKI increase since it results in partial cache thrashing. Figure 8 shows L1 MPKI and relative to the baseline performance for *dilution_t* values 1 through 30 when *fill-up_t = 256* and *matched_t = 4* (best configuration from Figure 7). As *dilution_t* increases, instruction misses are reduced improving performance up to a point. Afterwards, larger *dilution_t* leads to fewer migrations, lesser overhead, but higher I-MPKI. There is a tradeoff between reducing instruction

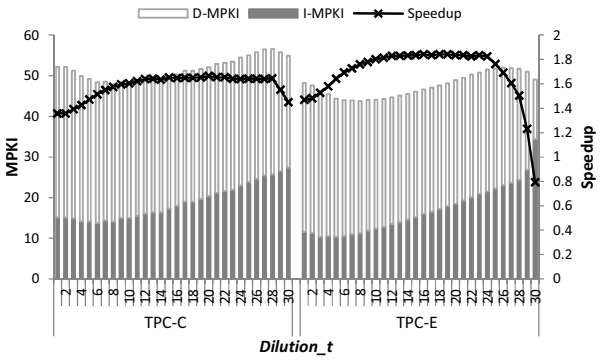


Figure 8: MPKI and relative performance as a function of *dilution_t*.

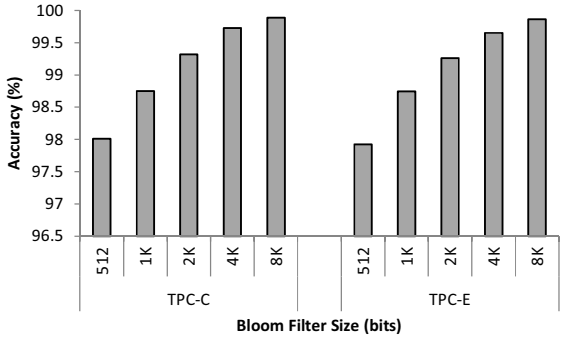


Figure 9: Partial-address bloom filter accuracy.

misses, and reducing migration overhead. Beyond *dilution_t* values of 28 (TPC-C) and 24 (TPC-E), although the overall MKPI is reduced, the performance degrades due to more limited migration. At even higher *dilution_t* values, migrations seize and performance drops below the baseline (for SLICC-SW, teams of transactions are injected to start on the same initial core, thus when migration stops some cores are underutilized).

In the remaining parts of this evaluation, we use *dilution_t = 10*, *fill-up_t = 256* and *matched_t = 4*. The last parameter means that the MTQ needs to keep track of only the four most recent misses, and that remote cache segment searching only requires finding where those four blocks are cached.

5.3. Cache Signature Accuracy

Section 4.2.3 explained that using a partial-address bloom filter reduces the overhead of remote cache segment searching. Figure 9 shows the accuracy of bloom filters of different sizes. The smallest bloom filter requires 512 bits to support evictions for a 32KB cache, with 64B blocks, and 512-sets. Accuracy is measured for all cache accesses and an access is accurate if the bloom filter and the cache agree on whether this is a hit or a miss. The trend is similar for TPC-C and TPC-E. In the rest of this paper, we experiment with 2K-bits filters as their effect on performance is less than 0.5% (99.3% accuracy).

5.4. Instruction Miss Change

Having determined a good SLICC configuration, this section shows that the three SLICC variants are able to reduce instruction misses much more than they increase data misses. Figure 10 shows the L1 I-MPKI for the baseline, SLICC, SLICC-SW, and SLICC-Pp. For MapReduce, since the instruction footprint fits in a 32KB cache, SLICC does not affect instruction or data misses.

Focusing on the other workloads, SLICC-SW reduces I-MPKI more than SLICC or SLICC-Pp. Compared to the baseline, SLICC-SW reduces I-MPKI by 56% and 61% for TPC-C and TPC-E, respectively. I-MPKI reductions are slightly lower with SLICC-Pp, more so for TPC-E than TPC-C, for the following reason: SLICC-Pp devotes one core to preprocessing. Given the transaction mix and transaction footprint sizes, having that extra core can be more important. All three variants of SLICC are sometimes forced to overcommit the caches by concurrently running transactions whose aggregate footprint does not fit on the total available L1 cache capacity. When overcommitting the caches, it is best to use stray threads since little opportunity for instruction reuse is lost when overcommitting with stray threads (a stray thread by definition is one that has few, if any, other ready threads sharing the same footprint). Overcommitting with non-stray threads happens more often for TPC-E than TPC-C partly because only 3% of TPC-E threads are stray compared to 12% of TPC-C threads. Furthermore, the need for stray threads is higher for TPC-E than TPC-C; SLICC spreads the transactions of TPC-E across $8-10$ cores, while TPC-C's transactions are spread across up to 14 cores.

SLICC improves I-MPKI less than the thread-type-aware alternatives, as it has to predict which is the next segment a thread will execute and where that segment currently is, using only a small preamble of the segment. The difference in reduction is more pronounced for TPC-C than TPC-E. TPC-C's overall instruction footprint is larger, resulting in higher variability in the instruction stream. Nevertheless, SLICC reduces instruction misses by 40.5%, on average.

As a comparison of the results for TPC-C-10 to those for TPC-C-1 shows, I-MPKI reductions persist mostly unaffected with the larger database.

5.5. Data Miss Change

During thread migration from core-A to core-B, three possible scenarios lead to extra data misses that would not have occurred otherwise: (1) a thread may read data on core-B that it fetched on core-A (extra misses on core-B for the same data blocks), (2) data writes on core-B to blocks fetched on core-A lead to invalidations that would not have occurred without migration (extra misses on core-B and invalidations on core-A), and (3) when a thread returns to core-A, it may find that data it originally fetched has since been evicted by another thread, or invalidated by itself (extra miss on core-A). Section 5.6 shows that instruction misses are more expensive than data misses performance-wise. Most data misses that result from migrations are served on-chip, allowing out-of-order execution to mostly absorb their latency.

Figure 10 reports the D-MPKI for all three SLICC variants and shows that SLICC-SW incurs an increase in D-MPKI of 11%, 1% and 4% over the baseline for TPC-C-1, TPC-C-10 and TPC-E, respectively. The other two variants exhibit a similar trend in D-MPKI increase. There is less locality and sharing in the larger data set of TPC-C-10, reducing the D-MPKI overhead when migrating.

Most of the increase in D-MPKI is for stores, which form 45% of total memory accesses, while loads are nearly unaffected. Due to slightly fewer migrations, SLICC-Pp increases D-MPKI less than SLICC-SW. As expected, SLICC is worse with an average D-MPKI increase of 9%.

We examined data prefetching to mitigate the increase in data misses. For each thread, we recorded the tags of the last n-referenced data blocks and then prefetched those blocks to the core the thread migrated to. This prefetcher did not improve performance, and past a

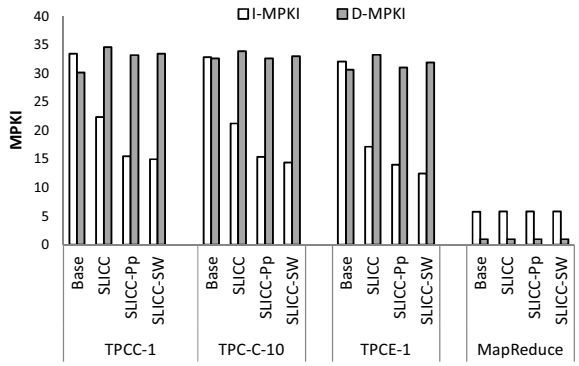


Figure 10: L1 I- and D-MPKI.

value of n, it hurts performance. There are several reasons why this prefetching proved ineffective. (1) The prefetched data increased the bandwidth on lower cache levels, which affects overall performance when n is high. (2) When n is low, there was not enough reuse. (3) Not all prefetched blocks are referenced again. (4) Finally, since 45% of the data accesses are stores, prefetching causes invalidations that would not have occurred otherwise.

This section showed that all SLICC variants improve I-MPKI significantly with a minor increase in D-MPKI, which is negligible with the larger database for TPC-C. These results suggest that SLICC can improve performance if, as expected, instruction cache misses degrade performance more than data cache misses; data misses can be partially overlapped with out-of-order execution. If this is the case, SLICC has the potential to offer a better balance of instruction *vs.* data cache misses over a conventional architecture.

Similar to D-MPKI, D-TLB misses increase on average by 11% and 8% with SLICC and SLICC-SW, respectively. I-TLB misses are within +/- 0.5% of the baseline.

5.6. Performance

This section reports the overall performance of SLICC relative to the baseline, a next-line instruction prefetcher, and a state-of-the-art prefetcher, PIF [5]. Figure 11 shows a 1.6× and 1.79× performance improvement over the baseline for SLICC-SW, on TPC-C-1 and TPC-E, respectively. On average, SLICC-SW and SLICC improve performance by 1.64× and 1.52× over the baseline, and 1.43× and 1.29× over a next-line instruction prefetcher.

Ferdman *et al.* report that PIF has nearly perfect coverage of L1-I misses [5]. Thus, we model an upper bound for PIF using a 512KB cache, with the delay of a 32KB cache. PIF's storage requirements are ∼40 KB per core. For TPC-C, SLICC-SW is within 2% of PIF's performance, with only 2.4% of PIF's storage requirements (see next section) per core. For TPC-E, SLICC-SW outperforms PIF by 21%. SLICC's speedup compared to PIF (and also the baseline) is a result of intelligent thread scheduling. Current schedulers assign threads to cores irrespective of their inherent-locality. Thus, even for larger caches, multiple similar threads run concurrently on different cores, each observing its own set of misses, for the same cache blocks. By pipelining similar threads, SLICC increases temporal inter-thread locality, hence it decreases overall miss rate observed by multiple threads.

MapReduce, which has an instruction footprint that fits in the L1-I cache, remains practically unaffected with SLICC.

5.7. Hardware Cost

Table 3 details the cost of all SLICC's hardware components. Section 4.2 described some of the components. In addition, SLICC

Table 3: Hardware Component Storage Costs.

Cache Monitor Unit	
Missed-Tag Queue (MTQ)	60-bits (16-core, *matched_t* = 4)
Miss Shift-Vector (MSV)	100-bits
Cache Signature (Bloom Filter)	2K-bits
Total	2208 bits (276 Bytes)
Thread Scheduler	
Thread Queue	30-entries (12-bits numerical ID, 48-bits pointer to thread context 4-bits core ID)
Total	1920 bits (240 Bytes)
Team Formation (SLICC-SW & SLICC-Pp)	
Team Management table	60-entries (12-bits numerical ID, 32-bits timestamp, 4-bits type ID, 4-bits team ID, 8-bits team index)
Total	3600 bits (450 Bytes)
Grand Total	7728 bits (966 Bytes)

requires a *thread queue* that holds threads waiting for cores. Each entry contains a unique numerical ID, a pointer to the threads' context, and a core ID. The thread queues can be local to each core, or centralized to one core. The table shows the cost for a centralized queue. Fewer entries are required when the queues are local to each core. The *team management table* is responsible for forming teams of similar threads (not required by SLICC). Each entry consists of: a unique numerical ID, a type ID, a team ID, index within a team, and a timestamp. The team management table is best thought of as being centralized, since every core needs to know which cores are assigned to which teams. We can either have one centralized copy, or per core copies that are kept coherent. For this work we simulated a centralized copy at one of the cores and modeled the necessary traffic.

On each core, a SLICC agent is responsible for managing the thread queue. The thread queue is a circular FIFO buffer and the first entry is executed until it migrates, completes, or gets blocked for I/O. On the latter case, the thread is moved to the end of the queue. With an over-provisioned thread queue of 30 threads, and a copy of the team management table, per core, SLICC requires a maximum of 966 bytes in addition to logic. All logic operations for SLICC are not on the critical path.

5.8. Remote Cache Segment Search Activity

As per the description of Section 4.2.3, a thread that wants to migrate has to find which cache, if any, holds the next code segment. Our results, thus, far modeled this searching by including separate messages for the corresponding miss messages using *separate* broadcasts. We do so to obtain an upper limit of the overhead these messages may induce. We report the frequency of these messages as Broadcasts per Kilo Instructions (BPKI) and find that it is very low. For TPC-C, BPKI is 2.204 for SLICC and 0.28 for SLICC-SW and SLICC-Pp. For TPC-E, BPKI is 1.328 for SLICC and 0.367 for SLICC-SW and SLICC-Pp. As Section 4.2.3 explained these requests are required

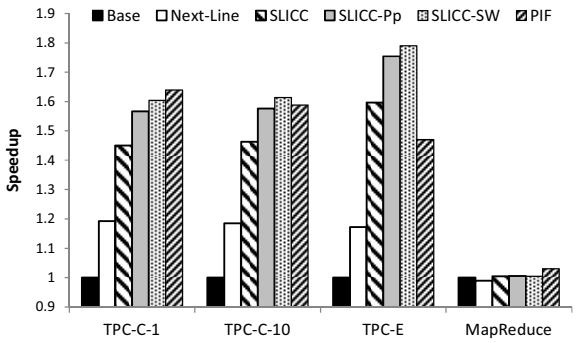


Figure 11: Performance.

anyhow for normal miss processing. The ownership information required by SLICC is either already available or should be possible to piggyback on existing responses.

6. Related Work

Instruction prefetching solutions have evolved from simple stream buffers [14, 26] to highly accurate, sophisticated stream predictors [6, 5]. Accurate prefetchers for OLTP are expensive, requiring ∼40KB of extra storage per L1 cache. Since these prefetchers track execution sequences, their storage requirements should increase with the instruction footprint. In addition, they neglect the possible presence of idle cores, and do not avoid code and prediction redundancy, under-utilizing on-chip resources. In this work we compared SLICC to PIF [5], a state-of-the art prefetcher that achieves near optimal instruction miss coverage. We showed that SLICC was able to achieve 98% of PIF's performance for TPC-C, using only 2.4% of the storage area overhead, while outperforming it by 21% for TPC-E.

Chakraborty *et al.* show a high-degree of redundancy in instruction fragments across threads concurrently running on multiple cores [3]. They propose CSP, which employs thread migration to distribute the dissimilar instruction code segments and group the similar ones together. For system code, which is commonly used by multiple threads, CSP fragments and distributes the code across a group of dedicated cores. CSP then migrates threads to these dedicated cores to execute system code. When threads are done, they return back to their original cores to resume execution for the user-level code. Thus CSP is limited to fragmenting OS code, losing opportunities of fragmentation within user code. SLICC generalizes thread migration to include interleaved user-OS code fragmentation points. In addition, thread migration in SLICC is managed by the hardware, while with CSP, the OS performs the migrations.

Atta *et al.* suggested using thread migration for reducing instruction misses in OLTP and demonstrated its potential to reduce I-MPKI without presenting a solution [2]. No performance analysis was conducted. This work presents a working solution and demonstrates its performance benefits.

STEPS [9] aims to minimize instruction misses from the software side. Like SLICC, it groups threads executing similar transactions into teams. It, either manually or by using a profiling tool, breaks each transaction's instruction footprint into smaller instruction chunks in a way that each chunk can fit in the L1-I cache. Then, all the threads in the same team execute the first chunk, rather than executing the whole transaction without any interruption, on the same core by context switching to the other thread when one completes the execution of the chunk. STEPS repeats this process for all the chunks, allowing instruction re-use across many threads for each chunk. SLICC exploits the same way of re-using the instructions already brought into the cache by previous threads. However, rather than context-switching on the same core, SLICC migrates threads to another core so that they can continue their execution. Moreover, SLICC dynamically detects the synchronization points in a transaction rather than using a priori manual or profiling based software instrumentation. Future work may look at combining the time-domain pipelining of STEPS with the space-domain pipelining of SLICC.

Data-oriented transaction execution (DORA) indirectly affects the instruction footprint of a transaction [21]. It divides a transaction into smaller actions based on the data being accessed at a particular transaction part. Then, each of those actions are sent to their corresponding worker threads, reducing the overall number instructions executed by a single thread per transaction. Such a design might not necessarily break a transaction into instruction parts that can fit in L1-I. However, if combined with SLICC, it can give better hints on where to migrate or reduce the total number of migrations needed to be done per worker thread.

Hardavellas *et al.* [8] observe that more than 60% of a distributed shared L2 accesses are for instructions. They adapt a NUCA block placement policy according to workload categorization, and allow replication of (read-only) instructions, which shortens the distance between L1-I caches and L2s. This reduces the L1-I miss penalty, but does not reduce the miss rate.

Other recent thread migration proposals target power management, data cache, or memory coherence [19, 25, 16, 27].

7. Conclusions

Literature showed that memory stalls for OLTP workloads account for 80% of their execution time, and L1 instruction misses account for 70-85% of overall stall cycles. We corroborate these results and show that 94% of L1 capacity misses are for instructions. Additionally, we show that recently proposed replacement policies, which reduce miss rates for some workloads, leave a lot of room for improvement compared to using larger L1-I caches. Previous works tackle this problem in software or hardware, but they are either impractical (require code instrumentation) or relatively expensive (large on-chip data structures).

This work presented a solution based on thread migration, SLICC. Similar to CSP [3] and STEPS [9], we exploit the code commonality observed across multiple concurrent threads. Unlike CSP, we do not limit code reuse to OS code segments. Unlike STEPS, instead of context switching on the same core, we distribute the instruction footprint across multiple cores and migrate execution. SLICC is a low-level hardware algorithm that requires no code instrumentation and efficiently utilizes available cache capacity, by improving intra- and inter-thread locality.

SLICC reduces the instruction misses for OLTP by 56% on average at the expense of an 5% average increase in data misses. SLICC improves the overall performance by 68% on average over the baseline and performs better when the input database is larger. Compared to a state-of-the-art instruction prefetcher (PIF), SLICC improves performance by 21% for TPC-E and comes within 2% for TPC-C, with only 2.4% of relative area overhead. When tested on MapReduce, a cloud workload that has a relatively small instruction footprint, SLICC was robust and did not affect the L1 miss rates or performance.

8. Acknowledgments

We thank the members of the AENAO and DIAS laboratories, Adrian Popescu, the reviewers, and Jared Smolens for their comments and

help. We thank Sudhakar Yalamanchili, Jun Wang, and the whole Georgia Tech development team for providing us with the Zesto simulator. This work was partially supported by an NSERC Discovery grant, an NSERC CRD with IBM, a Sloan research fellowship, NSF grants CCR-0205544, IIS-0133686, and IIS-0713409, an ESF EurYI award, and Swiss National Foundation funds.

References

[1] Advanced Micro Devices, "Secure virtual machine architecture reference manual," May 2005.

[2] I. Atta, P. Tözün, A. Ailamaki, and A. Moshovos, "Reducing OLTP instruction misses with thread migration," in *Proceedings of the Eighth International Workshop on Data Management on New Hardware*, 2012, pp. 9–15.

[3] K. Chakraborty, P. M. Wells, and G. S. Sohi, "Computation spreading: employing hardware migration to specialize CMP cores on-the-fly," in *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2006, pp. 283–292.

[4] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi, "Clearing the clouds: a study of emerging scale-out workloads on modern hardware," in *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2012, pp. 37–48.

[5] M. Ferdman, C. Kaynak, and B. Falsafi, "Proactive instruction fetch," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, 2011, pp. 152–162.

[6] M. Ferdman, T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos, "Temporal instruction fetch streaming," in *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture*, 2008, pp. 1–10.

[7] Gartner, "Market share: Database management system software, worldwide, 2008," 2009, available at http://www.gartner.com/DisplayDocument?id=1044912.

[8] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki, "Reactive NUCA: near-optimal block placement and replication in distributed caches," in *Proceedings of the 36th Annual International Symposium on Computer Architecture*, 2009, pp. 184–195.

[9] S. Harizopoulos and A. Ailamaki, "Improving instruction cache performance in OLTP," *ACM Transactions on Database Systems*, vol. 31, no. 3, pp. 887–920, Sep. 2006.

[10] M. D. Hill and A. J. Smith, "Evaluating associativity in CPU caches," *IEEE Transactions on Computers*, vol. 38, no. 12, pp. 1612–1630, Dec. 1989.

[11] IBM, "IBM breaks double digit performance barrier with 10 million transactions per minute," 2010, available at http://www-03.ibm.com/press/us/en/pressrelease/32328.wss.

[12] A. Jaleel, K. B. Theobald, S. C. Steely, Jr., and J. Emer, "High performance cache replacement using re-reference interval prediction (RRIP)," in *Proceedings of the 37th Annual International Symposium on Computer Architecture*, 2010, pp. 60–71.

[13] R. Johnson, I. Pandis, N. Hardavellas, A. Ailamaki, and B. Falsafi, "Shore-MT: a scalable storage manager for the multicore era," in *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*, 2009, pp. 24–35.

[14] N. P. Jouppi, "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers," in *Proceedings of the 17th Annual International Symposium on Computer Architecture*, 1990, pp. 364–373.

[15] K. Keeton, D. Patterson, Y. Q. He, R. Raphael, and W. Baker, "Performance characterization of a Quad Pentium Pro SMP using OLTP workloads," in *Proceedings of the 25th Annual International Symposium on Computer Architecture*, 1998, pp. 15–26.

[16] M. Lis, K. S. Shim, M. H. Cho, O. Khan, and S. Devadas, "Directoryless shared memory coherence using execution migration," in *Proceedings of the 24th IASTED International Conference on Parallel and Distributed Computing and Systems*, 2011.

[17] G. H. Loh, S. Subramaniam, and Y. Xie, "Zesto: A cycle-level simulator for highly detailed microarchitecture exploration," in *Proceedings of the International Symposium on Performance Analysis of Systems and Software*, 2009, pp. 53–64.

[18] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming language design and implementation*, 2005, pp. 190–200.

[19] P. Michaud, "Exploiting the cache capacity of a single-chip multi-core processor with execution migration," in *Proceedings of the 10th International Symposium on High Performance Computer Architecture*, 2004, pp. 186–.

[20] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, "CACTI 6.0: A tool to model large caches," HP, Tech. Rep., 2009.

[21] I. Pandis, R. Johnson, N. Hardavellas, and A. Ailamaki, "Data-oriented transaction execution," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 928–939, Sep. 2010.

[22] PARSA, "Data analytics benchmark with hadoop mapreduce framework," 2012, available at http://parsa.epfl.ch/cloudsuite/analytics.html.

[23] J.-K. Peir, S.-C. Lai, S.-L. Lu, J. Stark, and K. Lai, "Bloom filtering cache misses for accurate data speculation and prefetching," in *Proceedings of the 16th International Conference on Supercomputing*, 2002, pp. 189–198.

[24] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer, "Adaptive insertion policies for high performance caching," in *Proceedings of the 34th Annual International Symposium on Computer Architecture*, 2007, pp. 381–391.

[25] K. K. Rangan, G.-Y. Wei, and D. Brooks, "Thread motion: fine-grained power management for multi-core systems," in *Proceedings of the 36th Annual International Symposium on Computer Architecture*, 2009, pp. 302–313.

[26] P. Ranganathan, K. Gharachorloo, S. V. Adve, and L. A. Barroso, "Performance of database workloads on shared-memory systems with out-of-order processors," in *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, 1998, pp. 307–318.

[27] K. S. Shim, M. Lis, O. Khan, and S. Devadas, "Judicious thread migration when accessing distributed shared caches," in *Proccedings of the Third Computer Architecture and Operating System Co-design*, 2012.

[28] P. Tözün, I. Pandis, C. Kaynak, D. Jevdjic, and A. Ailamaki, "From A to E: Analyzing TPC's OLTP Benchmarks – The obsolete, the ubiquitous, the unexplored?" EPFL, Tech. Rep., 2012.

[29] TPC, "TPC benchmark C (OLTP) standard specification, revision 5.11," 2010, available at http://www.tpc.org/tpcc.

[30] TPC, "TPC benchmark E standard specification, revision 1.12.0," 2010, available at http://www.tpc.org/tpce.

[31] TPC, "TPC-C ten most recently published results," 2012, available at http://www.tpc.org/tpcc/results/tpcc_last_ten_results.asp.

[32] TPC, "TPC-E ten most recently published results," 2012, available at http://www.tpc.org/tpce/results/tpce_last_ten_results.asp.

[33] R. Uhlig, G. Neiger, D. Rodgers, A. L. Santoni, F. C. M. Martins, A. V. Anderson, S. M. Bennett, A. Kagi, F. H. Leung, and L. Smith, "Intel Virtualization Technology," *IEEE Computer*, pp. 48–56, 2005.

2012 IEEE/ACM 45th Annual International Symposium on Microarchitecture

Systematic Energy Characterization of CMP/SMT Processor Systems via Automated Micro-Benchmarks

Ramon Bertran[*†] Alper Buyuktosunoglu[†] Meeta S. Gupta[†] Marc Gonzàlez[*] Pradip Bose[†]

[*]Barcelona Supercomputing Center
C. Jordi Girona 29, Barcelona, Spain
{ramon.bertran,marc.gonzalez}@bsc.es

[†]IBM T.J. Watson Research Center
Yorktown Heights, NY, USA
{rbertra,alperb,mgupta,pbose}@us.ibm.com

Abstract

Microprocessor-based systems today are composed of multi-core, multi-threaded processors with complex cache hierarchies and gigabytes of main memory. Accurate characterization of such a system, through predictive pre-silicon modeling and/or diagnostic post-silicon measurement based analysis are increasingly cumbersome and error prone. This is especially true of energy-related characterization studies. In this paper, we take the position that automated micro-benchmarks generated with particular objectives in mind hold the key to obtaining accurate energy-related characterization. As such, we first present a flexible micro-benchmark generation framework (MicroProbe) that is used to probe complex multi-core/multi-threaded systems with a variety and range of energy-related queries in mind. We then present experimental results centered around an IBM POWER7 CMP/SMT system to demonstrate how the systematically generated micro-benchmarks can be used to answer three specific queries: (a) How to project application-specific (and if needed, phase-specific) power consumption with component-wise breakdowns? (b) How to measure energy-per-instruction (EPI) values for the target machine? (c) How to bound the worst-case (maximum) power consumption in order to determine safe, but practical (i.e. affordable) packaging or cooling solutions? The solution approaches to the above problems are all new. Hardware measurement based analysis shows superior power projection accuracy (with error margins of less than 2.3% across SPEC CPU2006) as well as max-power stressing capability (with 10.7% increase in processor power over the very worst-case power seen during the execution of SPEC CPU2006 applications).

1. Introduction

The power wall has proven to be a major obstacle in the quest to sustain the historical rates of performance growth in computing systems. The multi-core/multi-threaded design paradigm (CMP/SMT) has enabled the growth of throughput performance despite the dramatic slowdown in clock speed growth. However, power dissipation and current delivery limits make it hard to keep scaling indefinitely along the dimension of on-chip thread count. As such, it is important to understand the limits and sensitivities of energy-related metrics associated with current generation processors —so that future systems can invest into appropriate levels of power management in the right regions of the micro-architectural design space.

Microprocessor systems today are composed of multi-core, multi-threaded processors with complex cache hierarchies and gigabytes of memory. Predictive pre-silicon modeling and diagnostic post-silicon measurement studies are increasingly cumbersome and error prone. When it comes to power or energy-related metrics, the challenge is especially steep, since fine-grained power measurements or predictions across a complex, highly-threaded multi-core system are quite difficult. In this paper, we take the position that micro-benchmarks, generated with particular objectives in mind hold the key to obtaining accurate energy-related characterization. Specially crafted micro-benchmarks may be run on simulators (pre-silicon stage) or real machines (post-silicon stage) to help understand, diagnose and fix deficiencies systematically. However, manual generation of such 'stressmarks' is tedious, and requires intimate knowledge of the underlying micro-architecture pipeline semantics. Automated micro-benchmark generation is therefore crucial in this regard. Moreover, the automated generation facility must be flexible enough to generate different classes of micro-benchmarks that are useful in answering a range of different questions.

In this paper, we present a flexible micro-benchmark generation framework (MicroProbe) that is used to probe complex processor systems with a variety and range of energy-related queries in mind. In particular, three different characterization queries are illustrated in this paper. MicroProbe's automated generation facility is used to derive: (a) an accurate and decomposable power model that is used to project the power consumption for arbitrary CMP/SMT workloads; (b) energy-per-instruction (EPI) ratings for different instruction classes supported by the system; and (c) a systematically generated synthetic stress test that maximizes power consumption for the targeted system[1]. Experimental results are measured on a POWER7-based 8-core/32-thread system [42] in order to validate the efficacy of MicroProbe.

This is quite different from prior work [13,20,33,34,39,40], where 'black-box' automatic test case generators are focused on stressing or validating only a single metric: e.g. IPC, power or some utilization-based index of performance or power. MicroProbe presents the following unique features: *Detailed knowledge of low-level micro-architecture semantics* to assist the micro-benchmark generation process ('white-box' approach), a compiler-like *pass-based code generator* to provide flexibility and full control over the code being generated, and highly integrated *design space exploration support* to search for optimal micro-benchmarks. Overall, MicroProbe increases the productivity of the investigative micro-architect as he/she stresses the system to understand the fundamental trade-offs across power and performance metrics.

The main contributions of this paper are the following:

- We present the software architecture of *MicroProbe*: a framework for automated generation of micro-benchmarks that a user can adapt towards exercising a complex multi-core, multi-threaded computing system in a variety of redundant ways for answering a range of questions related to energy and performance. The illustrative use of MicroProbe in this paper is limited to three low-level energy-related case studies as stated below.
- We show how targeted micro-benchmarks generated by MicroProbe can be used to form a bottom-up power model that is able to predict general CMP/SMT workload power very accurately. To the best of our knowledge, this is the first bottom-up counter-based power model for a CMP/SMT processor. This type of models are

[1]From now on, we refer to it as *max-power stressmark*. This type of test cases are also known *synthetic TDP workloads*.

1072-4451/12 $26.00 © 2012 IEEE
DOI 10.1109/MICRO.2012.27

IEEE computer society

known to perform better [7–9] than those derived from common modeling approaches. However, they had limited applicability due to the lack of frameworks for automating the generation of the micro-architecture aware training data that they need. We show, through real measurements, that the model is able to predict POWER7 processor power consumption with average error of only 2.3% across the SPEC CPU2006 benchmarks. We compare the model against a set of models generated using existing approaches to show that the generated model outperforms existing approaches, even on extreme power situations. Finally, we use the extra information provided by the model to present the average SPEC CPU2006 processor power breakdown for different POWER7 SMT/CMP modes.

- We develop a taxonomy of the POWER7 instructions based on energy per instruction (EPI) and processor activity characteristics. As far as we know, this is the first EPI-based taxonomy at instruction level for a CMP/SMT processor such as the POWER7. We report up to 78% variations on EPI values across instructions, even when they stress the same functional unit at the same rate. These findings highlight the importance of such taxonomies in understanding the instruction-level power-performance trade-offs.
- We use EPI and IPC based formalisms to generate max-power synthetic stress test programs using the MicroProbe facility. Prior methods [20, 21, 33, 40] use abstract workload models to make the design space tractable, losing therefore opportunities during the instruction type selection pass. We exploit the rich information implemented in MicroProbe to use the instructions with higher EPI and IPC per functional unit as the building blocks of the max-power stressmark. Exhaustive exploration performed on that small subset of selected instructions was able to find a stress test that exceeds the maximum power seen during the full-suite SPEC 2006 benchmark execution by 10.7%. We also report that stressmarks with the same instruction type distribution and activity rate but different instruction order can show up to 17% difference in power consumption. The fact that the systematically generated stressmark slightly outperformed the hand-crafted stress tests generated by an expert confirms the utility of the proposed approach.

The rest of the paper is organized as follows. Section 2 explains the software architecture of the micro-benchmark generation framework. The hardware evaluation and experimental measurement platform is described in Section 3. Sections 4, 5 and 6 discuss the case studies. Section 7 summarizes the related research work and Section 8 provides concluding remarks.

2. MicroProbe framework

An overview of the design of MicroProbe and its usage flowchart is shown in Figure 1. MicroProbe provides a Python scripting interface to access to a rich set of mechanisms and features. The interface allows the users to identify the architecture components and their parameters in order to accommodate the micro-benchmark design to a target architecture. We show some examples highlighting the variety of possible user-defined micro-benchmark generation policies above the dotted horizontal line in Figure 1.

In MicroProbe, the micro-benchmarks are represented by a specific internal representation within the *Code generation module*. This representation can be transformed by a sequence of passes driven by the micro-benchmark synthesizer. The micro-benchmark synthesizer is in charge of generating the final code by applying the passes ordered in accordance to user-specified ordering rules. MicroProbe

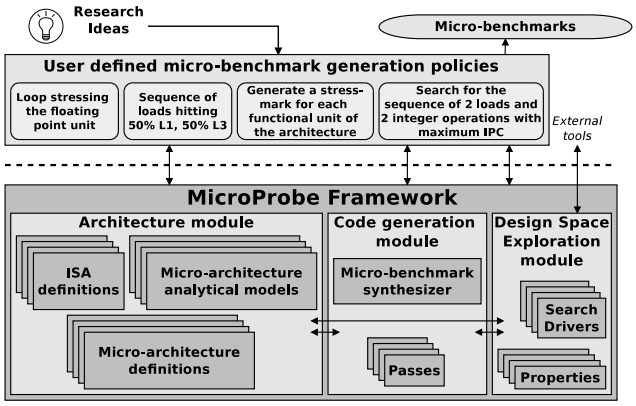


Figure 1: MicroProbe usage flowchart (top) and its design overview (bottom). The modular design provides flexibility in all the steps of the micro-benchmark generation process.

is therefore a framework that operates like a compiler infrastructure, achieving a high degree of flexibility and adaptability.

MicroProbe makes the whole micro-benchmark design process portable to different architectures. This feature is achieved by separating the architecture dependencies from the process itself (*Architecture module* in Figure 1). MicroProbe allows the user to describe the architecture through a set of readable text files where the architecture components and their parameters are set (the Instruction Set Architecture (ISA) and Micro-architecture definitions in Figure 1). All this information, in conjunction with micro-architecture analytical models, can be used (queried) to guide the micro-benchmark generation process. The generated micro-benchmarks are therefore bound to a specific architecture, but not the generation process.

Automated design space explorations (DSE) are required to assist the generation of micro-benchmarks with dynamic properties that cannot be ensured statically. MicroProbe integrates support for performing automatized DSEs within the *Design space exploration module* (See Figure 1). This module defines the mechanisms and features required to allow the user to define the design space and the search algorithm. Thus, MicroProbe is seen to provide full flexibility to perform any kind of DSE.

The modular design with standardized interfaces between the modules (as shown in Figure 1) makes the framework adaptable. In addition to this virtue, the following novel functionalities incorporated in MicroProbe advance the state of the art significantly:

MicroProbe is guided by low-level microarchitecture semantics. This is an important feature that was missing in previous work. This information is crucial to assist the generation of micro-architecture aware micro-benchmarks. It provides a 'white-box' solution to the users to define micro-benchmarks with very specific micro-architecture properties, avoiding the need to master every detail of the complex underlying architectures. We explain the Architecture module in detail in Section 2.1.

MicroProbe also presents novelty in the flexible code generation support (*Code generation module*) and in the *integrated* design space exploration (DSE) module. These functionalities, not available in previous work, improve the productivity and range of applicability of the micro-benchmark generation framework. In Section 2.2, we discuss the benefits of the compiler-like pass-based design of the *Micro-benchmark synthesizer* and in Section 2.3, we present the advantages of the integrated generic DSE support.

The rest of the section details the novel aspects within the three

```
1  import MicroProbe as MP
2  # Get the architecture object
3  arch = MP.arch.get_architecture("POWER7")
4  # Create the micro-benchmark synthesizer
5  synth = MP.code.Synthesizer(arch)
6  # Add the passes to be used
7  # to synthesize micro-benchmarks.
8  # Pass 1: Define the program skeleton
9  synth.add_pass("Single end-less loop
10                 of 4096 instructions")
11 # Pass 2: Define the instruction distribution
12 # Pass 2.1: Select the loads from the ISA
13 loads = [ Select ins in  arch.isa() if ins.load() ]
14 # Pass 2.2: Select the VSU Unit loads
15 loads_vsu = [ Select ins in loads
16               if  ins.stress(arch.comps["VSU"]) ]
17 synth.add_pass("Distribution using 'loads_vsu'")
18 # Pass 3: Model the memory behavior
19 # Pass 3.1 Define the memory model
20 model = "L1 = 33%", "L2 = 33%", "L3 = 34%"
21 synth.add_pass("Generate addresses
22                 according to 'model'")
23 # Pass 4: Init registers
24 synth.add_pass("Init registers to 0b01010101")
25 # Pass 5: Init immediate operands
26 synth.add_pass("Init immediates to 0b01010101")
27 # Pass 6: Model instruction level parallelism
28 synth.add_pass("Set instruction dependency
29                 distance randomly")
30 # Generate the 10 micro-benchmarks and save them
31 for idx in range from 1 to 10:
32   ubench = synth.synthesize() # Apply the passes
33   ubench.save("./example-%s.c"%(idx)) # Save
```

Figure 2: MicroProbe pseudo-code script that generates 10 micro-benchmarks consisting of an end-less loop with 4K load vector instructions that hit equally the three levels of the cache hierarchy. The highlighted parts in gray show how the micro-architecture information is queried to assist the micro-benchmark generation process.

improved features that we identified above. We focus the discussion on the features that are used in the case studies presented in Sections 4, 5 and 6. Other features implemented in MicroProbe are not included due to space limitations.

We guide the discussion using the MicroProbe script example shown in Figure 2. In this example, the user defines a policy to generate micro-benchmarks for the POWER7 micro-architecture (lines 2–3). The micro-benchmarks generated will be composed by an end-less loop of 4K instructions (lines 9–10). The instructions will be load vector instructions (lines 11–17) that hit equally to the three levels of the cache hierarchy (lines 18–22). The registers and immediate operands of the instructions will be initialized to a constant value (lines 23–26) and the dependency distance between the instructions will be assigned randomly (lines 27–29). Finally, the benchmarks synthesizer is invoked 10 times to generate 10 different micro-benchmarks (lines 31–33).

2.1. The Architecture module

The three main functionalities implemented in the *Architecture module* are the following: the PowerPC *ISA definition*, the POWER7 *Micro-architecture definition* and the set-associative cache model (a *Micro-architecture analytical model*).

The first functionality, the description of the PowerPC ISA, is used in the example to filter the load instructions of the ISA (lines 12–13 in Figure 2). The second, the POWER7 micro-architecture definition that provides the mapping between instructions and micro-architecture components stressed, is used in lines 14–16 of the example to select only the loads that stress the Vector Scalar Unit (VSU). The last functionality, the analytical set-associative cache model, is used to statically ensure a specific distribution among the memory hierarchy levels (lines 18–22). The following sections present the details of these three main modules of the *Architecture module*.

2.1.1. ISA definition module: This module implements the capability to generate assembly code for the target ISA. It leverages the format and the valid operands for each instruction of the ISA plus a rich set of semantic information for each of them. This includes the instruction type (e.g. load, store, vector, int, float or branch), the length of the operands of the instruction, if the instruction is executed conditionally, the privilege level required for the instruction, if the instruction is a data pre-fetch instruction, the registers used/defined by the instruction, the binary codification of the instruction, etc. This information is extensible and accessible by the user to perform any action based on it. For instance, one can select only the load instructions as shown in line 13 of Figure 2.

The ISA definitions are supplied to MicroProbe using readable text files. These definition text files are constructed using the information from ISA definition manuals. For this work, we implemented the definition text files for the Power ISA v2.06B [36]. This text-file based ISA definition approach provides an extra level of flexibility and adaptability. For instance, the user can add/remove instructions from the ISA and re-execute the very same MicroProbe script without requiring the modification of the MicroProbe internals.

2.1.2. Micro-architecture definition module: This module provides the information related to the specific micro-architecture implementation. From the architecture implementation point of view, this refers to the micro-architecture components and their hierarchy (functional units/sub-units), the cache hierarchy characteristics, the layout of the micro-architecture units (area, floor-plan information, etc.), the performance counters related to each micro-architecture component, etc. From the ISA point of view, this information includes the latency, throughput, power or EPI (energy-per-instruction) of the instructions. Moreover, the mapping between the instructions and the micro-architecture components they stress is also provided. For instance, the lines 14–16 of Figure 2 show how this information is used to select the instructions stressing the VSU unit. This rich set of low-level information, which simplifies the micro-benchmark generation task, is one of the new features that differentiates MicroProbe from all previous work.

Automatic bootstrap support. Similar to the ISA definition, the micro-architecture definition is supplied to MicroProbe using text files. This increases the portability of the framework. However, the process of setting up a complete micro-architecture definition is a time-consuming task that can still limit the portability of the framework. The reason is that all the details in the micro-architecture definition must be re-defined for each micro-architecture implementation. MicroProbe avoids this problem by implementing a bootstrap process that automatically completes a partial micro-architecture definition.

The following information is required to start the bootstrap process: (a) the micro-architecture functional units within the system. This includes their basic information (e.g. name) and their associated performance counters; (b) the definition of the 'IPC' property of the system (the performance counter-based formula); and (c) the ISA implemented in the micro-architecture.

The bootstrap process then generates two micro-benchmarks for each instruction of the ISA. The first micro-benchmark is an end-less

loop consisting of 4K instances of the instruction with a chain of dependencies across any two consecutive instructions. The second micro-benchmark is similar to the first one except that there are no dependencies. Both micro-benchmarks are executed and the performance counters related to the functional units and IPC are read. From these readings, MicroProbe derives the instruction latency and the units that are stressed. MicroProbe proceeds similarly with the second micro-benchmark to derive the throughput and confirm the functional units stressed.

In order to bootstrap the EPI or the average sustained power metrics, MicroProbe also reads the power sensors. MicroProbe uses the micro-benchmark version without dependencies to bootstrap these metrics. The micro-benchmarks generated use random values to initialize registers, immediate values and memory regions. This minimizes the possible data switching effects, allowing fair comparison between instructions [44]. The case study presented in Section 5 provides more insights about the automatically bootstrapped per-instruction EPI information.

2.1.3. Micro-architecture analytical models: Dynamic micro-benchmark properties are usually ensured by performing time-consuming design space explorations (DSEs). This process looks for the correct micro-benchmark generator input parameters to generate a micro-benchmark that satisfies the target dynamic properties. This process needs to evaluate each possible solution generated; therefore, it can be a practical limitation in real execution environments. However, it is known that under constrained conditions and detailed micro-architecture information, one can define analytical models to statically ensure dynamic properties of micro-benchmarks [14]. Therefore, the use of analytical models speeds up the micro-benchmark generation process, avoiding the time-consuming DSEs.

The level of detail provided in the *Micro-architecture definition* module enables the implementation of micro-architecture analytical models within MicroProbe. For instance, MicroProbe implements an analytical memory model for traditional set-associative cache hierarchies. We use it to generate in one step the micro-benchmarks with specific memory activities used in Section 4. The following section provides an overview of the rationale of this novel analytical memory model.

Set-Associative cache model: Previous work on micro-benchmark generation models the memory behavior by generating particular stride patterns that walk through pre-allocated memory [33]. It assumes that different stride values lead to different hit/miss ratios. Then, if a particular hit/miss ratio is required, a design space exploration (DSE) can be done to find the number of patterns, including their distribution and their strides. This would generate a targeted memory activity. Our modeling method avoids the need to perform a DSE and statically ensures the requested activity in each level of the cache hierarchy. The method is based on the following two observations:

First, with appropriate information —provided by the *Micro-architecture definition* (See Section 2.1.2)— it is possible to know and control the set used on each cache level when a memory operation is executed. For instance, Figure 3a shows how main memory blocks are mapped into a 4-way set associative cache. If we generate addresses within blocks 0, 128 or 256, we know that the data will be placed in Set 0.

Second, it is possible to ensure a hit or a miss in a particular cache level if enough accesses are generated. Taking into account the same example shown in Figure 3a, if we generate more than 4 consecutive

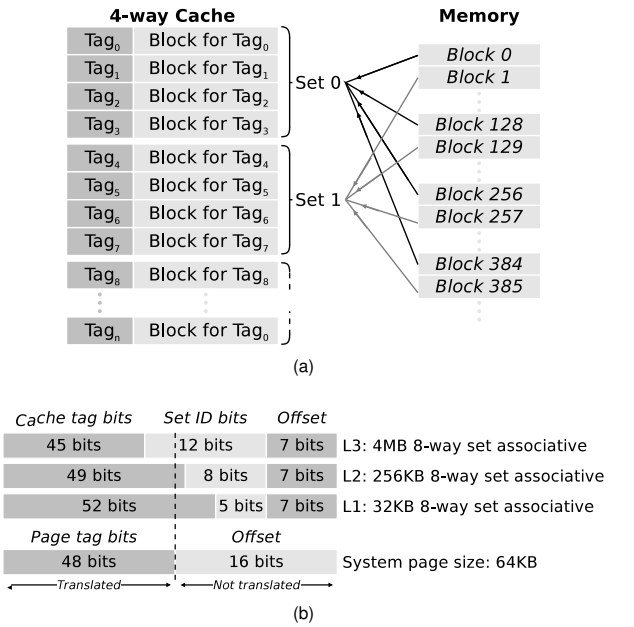


Figure 3: (a) Set-associative cache diagram. (b) Address fields at each level of the cache hierarchy and at the operating system level on a POWER7 platform.

memory requests hitting the set 0 within an end-less loop, we can ensure that the loop will enter a steady state where all accesses will miss. The memory requests should be randomized to minimize the interferences of the hardware pre-fetchers. On the contrary, if we generate 4 or less accesses hitting the set 0, the loop will always hit.

From these observations we can derive that it is possible to generate a sequence of memory accesses to ensure a particular distribution of the requests among the different levels of the cache hierarchy. For that purpose, we assign disjoint sets —sets that do not conflict— to each memory hierarchy level. We then generate the adequate number of accesses for each cache level. This is possible because: (a) MicroProbe provides to the user full control on the code being generated, and (b) the micro-architecture definition contains the required information to infer the set fields of each cache level. Figure 3b shows the set fields of each cache hierarchy level on our experimental platform.

This memory modeling method is used to apply the power modeling methodology presented in Section 4. This power modeling methodology requires several micro-benchmarks covering a wide range of memory activities. In this situation, being able to statically ensure memory activity rates reduces the time required to generate the micro-benchmarks.

2.2. The code generation module

The code generation module contains the micro-benchmark synthesizer. The micro-benchmark synthesizer is the core component of any micro-benchmark generation framework because it is in charge of driving the code generation process. Previous work [4, 33] identified that the code generation process requires a minimum number of steps to define the final behavior of the micro-benchmarks generated. These steps are the following: (1) define the program skeleton (e.g. the size of basic blocks; number of threads, etc.); (2) define the instruction distribution; (3) model the memory behavior (i.e. define how the memory is accessed); (4) model the branch behavior (i.e. control the level of speculation); (5) model the instruction level parallelism (ILP)

via register allocation (i.e. define the dependency distance between instructions). This step wise approach has been observed to be the common method to define the properties of the micro-benchmarks generated.

We designed the micro-benchmark synthesizer of MicroProbe to work in a compiler-like fashion. The rationale is that this design provides the flexibility and extensibility required to adapt the micro-benchmark generation process to the user's requirements. This differs from prior work, where the transformations and the sequence of steps are fixed and tailored to solve specific problems. The example script of Figure 2 shows how the user defines the sequence of transformations (i.e. their type and their order) required to generate the micro-benchmarks. We call these transformation steps *passes*.

Within MicroProbe, new passes can be added and sorted at user's will, making the framework extensible and adaptable. Many basic passes, like the ones in the example in Figure 2, are already available in our framework. This forms a general repository of passes for designing complex micro-benchmark generation policies. To name a few, we have implemented a pass to set up an end-less loop with *n* instructions (line 8 of Figure 2), a pass to generate a given instruction distribution (line 17 of Figure 2) and a memory pass that ensures a given memory activity (See Section 2.1.3). Several other passes to model branch behavior, initialize values, etc. are also implemented. We refer the reader to previous work on micro-benchmark synthesizers [18, 20, 21, 33, 40] to read about other possible transformation passes that can be implemented on top of MicroProbe.

We show the importance of having a compiler-like design explaining a possible real world example. Let's suppose that we have a computational kernel and we want to test the effect of certain transformations on it. We set up a MicroProbe script to generate the baseline code —i.e. the initial sequence of instructions comprising the kernel. We may then want to evaluate the effect on performance of unrolling the loop or the effect on power of using a *load immediate* and an *add* instruction instead of two *add immediate* instructions. For that purpose, we simply copy the original MicroProbe script that generates the computational kernel and then add the extra passes to apply the transformations. This level of adaptability is enabled by the pass-based design of the micro-benchmark synthesizer.

2.3. Design space exploration module

Design space explorations (DSE) have become mandatory to understand the performance of computer architectures due to their increase in complexity. In addition, DSE are required to find micro-benchmarks that fulfill a set of dynamic properties that cannot be ensured statically. DSE support is therefore a basic functionality that any productive micro-benchmark generation framework should have.

MicroProbe provides generic DSE support to be able to implement different customizable search strategies within the design space. For instance, MicroProbe currently supports exhaustive searches, genetic algorithm (GA) searches and user-defined searches. This is in contrast to previous work, which only provided GA search support [20, 21, 33, 40]. Thus, MicroProbe provides an adaptive framework for performing DSEs.

Moreover, the fact that DSE support is integrated within the same framework is also beneficial. Previous work decoupled the micro-benchmark synthesizer component from the search driver component, thus losing possible synergies between these components. MicroProbe integrates both functionalities into the same framework. This allows, for instance, the definition of user-guided drivers that query the micro-architecture information in order to guide the search. In Section 6 we use the integrated DSE support of MicroProbe to generate a max-power stressmark. The search driver we define uses the per-instruction EPI information and the mapping between instructions and the functional units to focus the search on certain parts of the design space.

Feature	Micro-Probe	Previous Work
ISA queries		
- *instruction type*	✓	✓
- *operand length*	✓	Manual[1]
Micro-architecture queries	✓	
- *functional unit*	✓	Manual[1]
- *latency*	✓	Manual[1]
- *throughput*	✓	Manual[1]
- *energy per instruction (EPI)*	✓	Manual[1]
- *average instruction power*	✓	Manual[1]
Micro-architecture models		
- *Set associative cache model*	✓	No
Code generation		
- *Skeleton definition pass*	✓	✓
- *Instruction definition pass*	✓	✓
- *Basic memory modeling pass*	✓	✓
- *Branch modeling pass*	✓	✓
- *ILP definition pass*	✓	✓
- *Configurable passes*	✓	No
Design space exploration		
- *Integrated*	✓	No
- *GA-based search*	✓	✓(External tool)
- *Exhaustive search*	✓	✓(Manually)

[1]The user manually or using an external tool has to obtain the information to pass the appropriate inputs to the code generator.

Table 1: Summary of the novel MicroProbe features and their implementation in previous work.

2.4. Summary of MicroProbe features

Table 1 summarizes the micro-benchmark generation features included in MicroProbe and their implementation in prior work. MicroProbe provides detailed architecture related information such as queries about ISA and micro-architecture information. This depth of architecture-related information is not offered in previous micro-benchmark generation frameworks. Some prior work includes limited instruction type semantics. However, simple queries like functional unit information (lines 15–16 in Figure 2) or instruction latency information require the user to obtain the information manually. In the end, the lack of this integrated low-level micro-architecture semantics diminishes the benefits of having an automatic micro-benchmark generator.

MicroProbe implements micro-architecture models such as the set associative cache model. As far as we know, this feature is not found in previous work. MicroProbe does provide the basic support for code generation, as in prior work. In other words, it supports at least the minimum set of transformation passes that define the behavior of the micro-benchmark generated. MicroProbe goes one step further by improving the flexibility of the code generation support by allowing the passes to be configured. Finally, regarding the DSE support, MicroProbe integrates such support within the same framework whereas previous work uses external tools or manual setups to perform DSEs.

3. Experimental Framework

The experimental platform is an IBM BladeCenter PS701 system. The system has one POWER7 processor running at 3.0 GHz and 32 GB of DDR3 SDRAM running at 800 MHz. The IBM POWER7 processor is an eight-core chip where each core can run up to four threads. Each core has 32KB first level, 256KB second level and 4MB third level data cache. A detailed specification of the architecture is available elsewhere [42]. The platform runs RHEL 5.7 OS with linux kernel version 3.0.1. This version provides the standard PCL API [17] to access hardware performance counters.

The platform implements the EnergyScale architecture [19] that allows the users to gather the power consumption of the processor via the Flexible Support Processor (FSP). The FSP accesses the micro-controller called Thermal and Power Management Device (TPMD) to perform the sensor readings. Both devices, the FSP and the TPMD, are managed by the BladeCenter chassis Management Module (MM).

We use an in-house software to monitor all the sensors required for the experiments. The software can sample sensors at 1-ms granularity. Power measurements are in the granularity of milliwatts, whereas the temperature measurements are in degrees celsius. We also gather performance monitoring counter (PMC) traces to account for different activity of the micro-benchmarks and the SPEC CPU2006 benchmarks that are executed. Power and performance counter traces are then analyzed and plotted using the POTRA framework [6].

Micro-benchmarks are deployed as one copy per hardware thread context that is available on the configuration. For example, in a 2-way SMT 6-core configuration, we deploy 12 copies of the micro-benchmark. We pin each copy to a logical CPU to avoid thread migrations. We run the micro-benchmarks for 10 seconds which helps us to shorten the data gathering process while still providing valid —and stable— power and performance counter values. Similarly, we also execute the SPEC CPU2006 [25] benchmark suite for model validation purposes. The SPEC CPU2006 benchmarks are run to completion. All power results presented in this work are in normalized form to avoid disclosure of absolute values.

4. Bottom-up CMP/SMT aware counter-based processor power model

One important area where MicroProbe provides special value is in the task of generating empirical counter-based power models. Such power models are of key interest because they provide a quick path to estimate run-time power consumption without the need to rely on direct measurement devices [5, 27]. Counter-based power models have not only been used to model the power consumption of the processor [5, 29, 35, 43], they also have been useful to predict the consumption of the rest of the components in the system [10, 11].

In particular, *bottom-up* counter-based power modeling methodologies have been shown to be a competitive approach [7]. Besides accuracy and generality [8, 9], these types of models provide a fine-grained granularity, sometimes allowing per functional unit breakdowns [27]. Although we do not focus this work to reach such low level of decomposability, we present a method to generate bottom-up counter-based power models for SMT/CMP processors such as the POWER7.

Bottom-up processor power models predict the overall power consumption of the processor as the sum of the power consumption of different power components. These power components are usually associated with micro-architecture components [8, 9, 27]. This allows the users to derive the power breakdown across these components. This adds insight on power behavior across workloads and individual components within the processor. In a CMP/SMT system, this capability is useful in discerning the power consumption of each core or hardware thread. In addition, the power-related effects of enabling the SMT logic or enabling/disabling cores can be easily quantified.

Previous methods of bottom-up processor power modeling were applied to processors that lack the level of parallelism and complexity of the POWER7. In [27], a bottom-up power model of a Pentium 4 is presented. In [7–9], the authors model a dual-core processor without SMT. In contrast, we model a highly parallel processor such as the POWER7, with 8 cores and up to 4-way SMT capabilities.

Bottom-up counter-based modeling methods require micro-benchmarks that stress different micro-architecture functional units at different levels. This is needed to estimate individual contributions to the overall power consumption [8, 9, 27]. In this context, a common rule of thumb is to use a very broad range of power contexts for training the model. This is known to result in a more general and accurate model. This implies a rather time-consuming task of generating a huge set of micro-architecture aware micro-benchmarks. This requirement delayed the application of bottom-up modeling methods on current architectures. The main reason was the lack of micro-benchmark generation frameworks like MicroProbe that have micro-architecture semantics.

We use MicroProbe to generate the rich set of micro-benchmarks shown in Table 2. We generate micro-benchmarks that stress different combinations of functional units at different levels (IPCs) by using the micro-architecture information and the DSE GA-based support implemented in MicroProbe. The functional units of the POWER7 processor that we stress are: the fixed point unit (FXU), the load store unit (LSU) and the vector scalar unit (VSU). We also generate micro-benchmarks stressing the memory hierarchy at different levels. We stress the four levels of the memory hierarchy: the first-level cache (L1), the second-level cache (L2), the third-level cache (L3) and the main memory (MEM). In this process, the analytical micro-architecture memory model of MicroProbe (See Section 2.1.3) removes the necessity to perform a DSE for each memory activity we target. Finally, micro-benchmarks with random activities are also generated in order to enrich the training set.

Notice that hand-crafting —and verifying— this micro-benchmark suite is normally a very time-consuming effort. With MicroProbe we are able to do it in a few hours without any human intervention. The next section explains the modeling methodology that uses these micro-benchmarks to produce a SMT/CMP aware bottom-up counter-baser processor power model.

4.1. SMT/CMP aware bottom-up modeling methodology

We apply the bottom-up methodology shown in Figure 4 to model the processor. This methodology ensures the decomposability of the model because it models the power consumption of the different processor power components defined separately. We define the following four power components: (a) the dynamic power consumption, i.e. the power related to the activity of the hardware contexts running on the system; (b) the SMT effect, i.e. the power contribution of enabling the SMT logic of the cores; (c) the CMP effect, i.e. the power contribution of enabling multiple cores on the system; and (d) the uncore power contribution, i.e. the constant power contribution of having activity on the processor. Moreover, there is the workload independent power consumption, which is the power consumption of the processor when there is no activity.

Name	Units stressed[1]	#	Description	MicroProbe features
Simple Integer	FXU or LSU	35	Mix of simple integer instructions (can be executed by the LSU or FXU units) with IPCs from 0.5 to 4 in steps of 0.1.	ISA & uarch queries & DSE GA support
Complex Integer	FXU	11	Mix of complex integer instructions (only can be executed by the FXU unit) with IPCs from 0.1 to 1.1 steps of 0.1.	"
Integer	FXU, LSU	12	Mix of integer instructions with IPCs from 0.10 to 1.20 in steps of 0.1.	"
Float/Vector	VSU	14	Mix of vector, float and decimal instructions with IPCs from 0.1 to 1.4 in steps of 0.1.	"
Unit Mix	VSU, FXU, LSU	20	Mix of all kind of instructions (non memory, no branch) with IPCs 0.1 to 2 in steps of 0.1.	" "
L1 ld	LSU, L1	10	Random mix of load instructions hitting the L1.	ISA queries & uarch model
L1 ld/st	LSU, L1, L2	10	Random mix of load/store instructions hitting the L1.	"
L1L2a	LSU, L1, L2	10	Random mix of load/store instructions 75% hitting the L1 and 25% hitting the L2.	"
L1L2b	LSU, L1, L2	10	Random mix of load/store instructions 50% hitting the L1 and 50% hitting the L2.	"
L1L2c	LSU, L1, L2	10	Random mix of load/store instructions 25% hitting the L1 and 75% hitting the L2.	"
L1L3a	LSU, L1, L2, L3	10	Random mix of load/store instructions 75% hitting the L1 and 25% hitting the L3.	"
L1L3b	LSU, L1, L2, L3	10	Random mix of load/store instructions 50% hitting the L1 and 50% hitting the L3.	"
L1L3c	LSU, L1, L2, L3	10	Random mix of load/store instructions 25% hitting the L1 and 75% hitting the L3.	"
L2	LSU, L1, L2	10	Random mix of load/store instructions hitting the L2.	"
L2L3a	LSU, L1, L2, L3	10	Random mix of load/store instructions 75% hitting the L2 and 25% hitting the L3.	"
L2L3b	LSU, L1, L2, L3	10	Random mix of load/store instructions 50% hitting the L2 and 50% hitting the L3.	"
L2L3c	LSU, L1, L2, L3	10	Random mix of load/store instructions 25% hitting the L2 and 75% hitting the L3.	"
L3	LSU, L1, L2, L3	10	Random mix of load/store instructions hitting the L3.	"
Caches	LSU, L1, L2, L3	10	Random mix of load/store instructions 33% hitting the L1, 33% hitting the L2 and 34% hitting the L3.	"
Memory	LSU, L1, L2, L3, MEM	20	Random mix of load/store instructions missing in all levels of the cache hierarchy.	"
Random	Unknown	331	Random micro-benchmarks.	ISA queries

[1]FXU: fixed point unit (integer), LSU: load store unit (memory operations) and VSU: vector scalar unit (vector, float and decimal operations).
L1: L1 cache, L2: L2 cache, L3: L3 cache, MEM: Main memory

Table 2: Micro-benchmarks automatically generated using MicroProbe. They cover a broader scope of possible processor activities in order to increase the accuracy of the models generated. They share a common skeleton: a 4K endless loop with the required instructions to stress particular functional units.

The bottom-up modeling methodology used, introduces two new components when compared to previous bottom-up modeling methods [8, 27]. The new components are the SMT effect and the CMP effect components.

The SMT effect component models the extra power required when SMT is enabled. We observed empirically that two workloads exhibiting the very same overall core activity consume a different amount of power depending on whether SMT is enabled or disabled. The reason is that the extra control logic that in operation when SMT is enabled consumes additional power. This effect is independent of whether 2-way SMT or 4-way SMT is enabled.

The second new component, the CMP effect, models the change of uncore power consumption depending on the number of cores enabled. This power consumption changes due to the different usage of the shared components when different number of cores are used. For instance, the 32MB last level cache of the POWER7 is partitioned into eight equally sized slices, one for each core. When a core is not used, the last level cache slice of that core is used only as a victim cache of the other slices, changing its usual power behavior. The CMP effect captures the specific conditions in power consumption that depend on the number of cores enabled.

The addition of these two variables is crucial to increasing the accuracy of the models. They are not directly related to the actual activity in the processor like performance counters. However, they affect how the activity is being performed as well as the power status of different micro-architecture components. Models without these two input variables —the SMT enabled and the number of cores enabled (*#cores*)— exhibit large errors in the predictions and show inconsistencies across the different SMT and CMP modes of operation. The rest of the section explains the details of each of the four modeling steps shown in in Figure 4.

Step 1: Model a Single Hardware Context: We model a single core in single-threaded (SMT-1) configuration using the bottom-up modeling method detailed in [8]. In brief, we define the FXU, VSU, LSU, L1, L2, L3 and MEM as the power components of the processor cores. We assign a performance monitoring counter (PMC) based formula for each of these components. A sequence of linear regressions is then performed to model separately the power contribution of each of these power components. This is possible because the specifically designed training set covers a wide set of scenarios that stress different units at different utilization rates [8]. The model intercept is then calibrated using the random micro-benchmarks to avoid underestimating the power when only particular units are stressed [8, 49]. The result of this process is a bottom-up power model for a single core in SMT-1 configuration. The dynamic component of the model, the part that it is dependent on the PMCs (Dynamic Power in Figure 4), is defined as:

$$\begin{aligned} P_{dyn} &= FXU_{pmcs} \times W_{fxu} + VSU_{pmcs} \times W_{vsu} \\ &+ LSU_{pmcs} \times W_{lsu} + L1_{pmcs} \times W_{l1} + L2_{pmcs} \times W_{l2} \\ &+ L3_{pmcs} \times W_{l3} + MEM_{pmcs} \times W_{mem} \end{aligned}$$

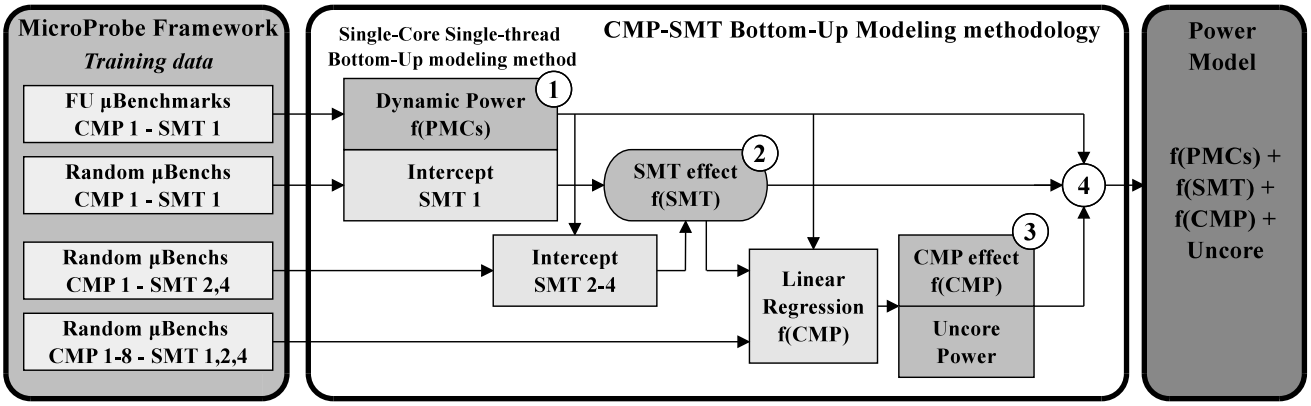


Figure 4: Proposed SMT/CMP aware bottom-up modeling methodology. (1) A single core —a single hardware context— is modeled; (2) the effect of enabling SMT is estimated; (3) the effect of enabling cores —the CMP effect— and the uncore power consumption are estimated; (4) the final model is defined as the sum of the power consumption of each hardware context, the SMT effect of each core with SMT enabled, the CMP effect and the uncore power consumption.

The non-dynamic component (intercept SMT-1 in Figure 4) is used in the next step to compute the SMT effect.

Step 2: Model the SMT Effect: As stated previously, we observed that the power consumption is higher when SMT is on. We simplify the modeling of this behavior by assuming that the power consumption increases by a fixed value when SMT is activated. We therefore model the SMT effect as a constant value, which is defined as:

$$SMT_{effect} = Intercept_{SMT2-4} - Intercept_{SMT1}$$

where the SMT effect value is the difference of the uncore power consumption between a model trained using SMT enabled data (intercept SMT-2–4) and the model trained using SMT disabled data (intercept SMT-1).

Step 3: Model the CMP Effect and the uncore power: To model the CMP effect and the uncore power, we apply the dynamic and the SMT effect models defined in steps 1 and 2 to the random micro-benchmarks executed in all SMT and CMP configurations (See step 3 of Figure 4). After applying the model, we obtain the residuals of the predictions. These residuals, which exhibit a positive correlation with the number of cores enabled, can be interpreted as the power consumption related to the change in the number of cores plus the uncore power. We therefore model the residuals as a function of the number of cores enabled (#*cores*) using a linear regression of the form $a \times x + b$. The intercept of the obtained regression (i.e. b) is assumed to be the uncore power consumption (P_{Uncore}) whereas the $a \times x$ component is assumed to be the CMP effect ($CMP_{effect} \times \#cores$).

Step 4: Combine the models: We combine all the modeled power components to obtain the final bottom-up power model. The model is therefore defined as:

$$P_{cpu} = \sum_{k=1}^{\#threads} P_{dyn_k} + \sum_{k=1}^{\#cores} SMT_{effect} \times SMT\_enabled_k + CMP_{effect} \times \#cores + P_{Uncore}$$

which is the addition of the power consumption of each hardware thread enabled on the platform (step 1), the SMT effect of the cores with SMT enabled (step 2), the CMP effect as a function of the number of cores enabled and the uncore power consumption (step 3).

4.1.1. Model Validation: Figure 5a shows how the model is able to track the power consumption of the SPEC CPU2006 on a 4 core, 4-way SMT configuration. The dynamic power consumption varies with the workload whereas the rest of components remain constant because they depend on the processor SMT/CMP configuration. This power consumption breakdown is only possible because of the bottom-up modeling methodology. Top-down modeling methods [5, 11, 23, 41] model the processor as a black box. They are able to perform per-core power estimations by gathering per core performance counters. However, they do not provide the same insights [7].

Figure 5b shows the percentage average absolute prediction error (PAAE) [10] of the proposed bottom-up (BU) model when compared to actual measured power of the SPEC CPU2006 workloads for all the configurations studied. The maximum PAAE is around 4% and most of the values are below 2.3%, which is the average PAAE. These results validate that the novel SMT/CMP aware bottom-up modeling method is able to model different SMT/CMP configurations accurately.

There is, however, a small trend that shows higher errors for higher number of cores. This might be related to the CMP and SMT attributes, which we modeled assuming a linear relation. This linear approximation is necessary to help us to create the bottom-up hierarchical CMP/SMT power model. The implicit assumption of linear dependence of these attributes on power is an approximation of what is most likely a non-linear model. For example, if the real values follow a monotonic convex/concave curve, a linear approximation will yield an error function that causes absolute error to first increase and then decrease, as seen in Figure 5b.

4.1.2. Comparison to other Models: We compare our bottom-up (BU) model against a set of top-down (TD) models [7] in order to bring out the benefits of the bottom-up modeling approach. TD modeling methodologies use parameter selection techniques to select the model inputs and then they apply a single multiple linear regression to model the entire processor. These models do not require specifically designed micro-benchmarks. They are therefore a popular solution due to their simple generation. However, they do not provide the same accuracy and generality as the bottom-up models.

We generate three TD models using the same inputs of our bottom-up (BU) model for fairness: namely, the functional unit performance counters, the numbers of cores enabled and the SMT mode. The models are named after the training set used to generate them: the

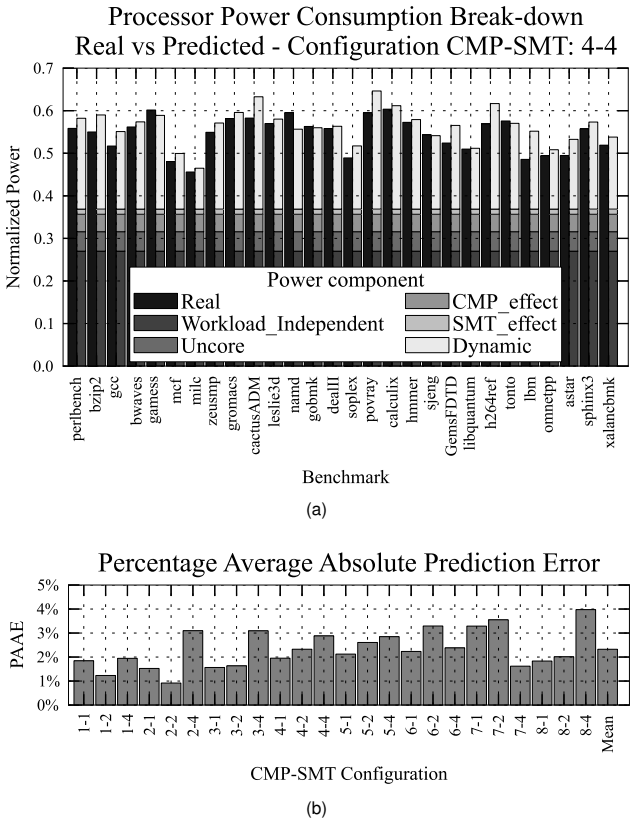


Figure 5: (a) Processor power consumption breakdown of the SPEC CPU2006 on a 4 core, 4-way SMT configuration. (b) Percentage average absolute prediction error (PAAE) of the model for all the configurations analyzed.

micro-architecture aware micro-benchmarks (TD_Micro), the random micro-benchmarks (TD_Random), and the SPEC benchmarks (TD_SPEC). TD_SPEC is therefore the optimistic model because it has been trained using the validation set. As explained previously, the BU model is trained using all the micro-benchmarks, namely the micro-architecture aware micro-benchmarks and the random micro-benchmarks.

Figure 6 shows the average PAAEs on the SPEC CPU2006 with respect to the models generated for each configuration studied. All the models show similar trends confirming that each training set covers enough power contexts to model the SPEC CPU2006 suite accurately. In general, the models are consistent across the different SMT/CMP configurations. Only the TD_Micro shows a consistently higher error for 2-way SMT configurations. In any case, all the models show acceptable results on average.

The last columns of Figure 6 show mean PAAEs around the 2–4% range. When compared to the optimistic model (TD_SPEC), the rest of the models show less than 2 percentage points of difference. These accurate predictions are enabled by the inclusion of the SMT and CMP variables to the models. The proposed BU model outperforms the rest, being the one closer to the optimistic TD_SPEC model.

4.1.3. Model Validation on Extreme Cases: Although there is not a clear difference in accuracy between the different modeling methods for general workloads, there is a significant difference when extreme cases are considered. We consider different extreme cases such as high and low integer (FXU) or vector activity (VSU), only L1 loads or only memory activity. Although we call these cases *extreme*, these types of activities are actually quite common in applications over short periods of time. For instance, consider the case of a highly optimized vector loop accessing only the first level cache. In such a case, the processor will show a period with only high IPC vector activity. Similarly, when the processor copies data from main memory to a local array, only main memory activity will be exhibited.

Figure 7 shows the PAAEs of the models for the extreme activity cases considered. The models trained using micro-architecture aware micro-benchmarks (i.e. the TD_Micro and the BU models) are capable of modeling these situations accurately, whereas the models trained using general workloads exhibit high errors. For instance, the TD_Random model shows a 62% PAAE for the FXU High case. This is because the models trained using general workloads are biased towards the *normal* activities they exhibit. In contrast, the models trained using micro-architecture aware training sets show similar accuracy levels across general and extreme workloads.

This observation highlights the benefits of generating micro-architecture-centric models like the bottom-up model instead of the workload-centric models like the top-down models. A framework like MicroProbe, capable of generating micro-architecture aware micro-benchmarks, is therefore essential for facilitating the generation of micro-architecture aware training sets.

4.1.4. SMT/CMP Effects on Power Consumption: In this section, we use the decomposability capability provided by the bottom-up model to analyze how the SMT/CMP configuration affects the distribution of power consumption. Notice that this level of insight is not possible using top-down models [7]. Figure 8 shows the average percentage power consumption breakdown for the SPEC CPU2006 for each configuration analyzed.

From the SMT point of view, changing the SMT configuration increases the percentage of dynamic power consumption of the processor by about 10 points. At the same time, it decreases the workload independent power component by an identical amount. The reason is twofold: (a) the more hardware contexts are enabled, the more dynamic power is consumed due to the increase of ILP within the cores; (b) this increase in dynamic activity exceeds the overhead of enabling the SMT feature (SMT_effect in Figure 8), which we found to be minimal (<3% in all the cases).

The components that do not depend on the CMP parameter —the workload independent and the uncore components— account for up to 85% of the overall power consumption in the lowest configuration (i.e. 1 core, 1-way SMT configuration). This percentage is reduced to 50% as we increase the number of hardware contexts (8 cores, 4-way SMT configuration). This is mainly due to the increase of the dynamic component. We also observe that the power breakdown remains comparable when a minimum of 4-cores are enabled. Beyond that point, adding extra cores results in a similar increase of dynamic and non-dynamic power consumption, suggesting that the shared resources are already fully utilized. For instance, in Figure 8, going from 1–1 to 2–1 CMP–SMT configuration reduces the workload independent and uncore power consumption from 85% to 77%. However, going from 7–1 to 8–1 only reduces these components by 1 percentage point, from 62% to 61%.

In summary, we present a novel bottom-up power modeling methodology capable of modeling the SMT/CMP features of current architectures. We show how the model generated using this methodology outperforms existing approaches for normal and extreme workloads. The basis of the model is a complete micro-architecture aware

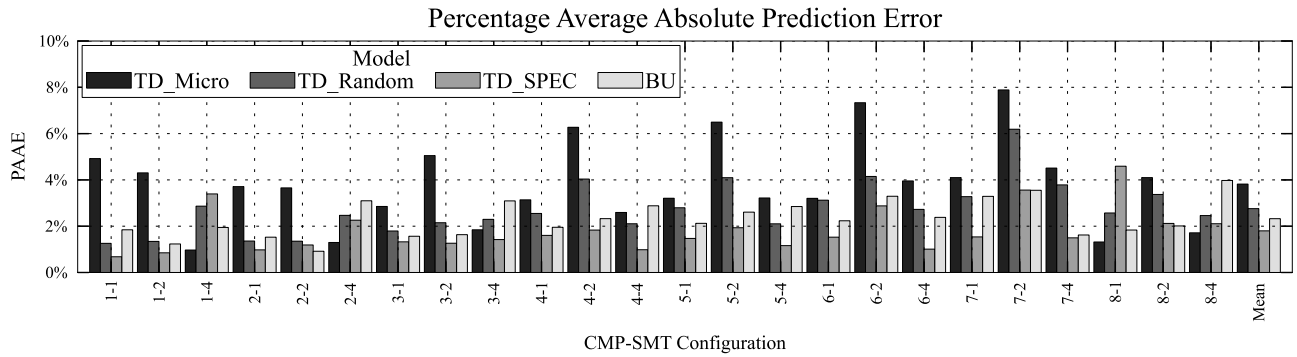


Figure 6: Percentage average absolute prediction errors of the models generated when compared to actual measured power of the SPEC CPU2006 workloads for all the configurations analyzed.

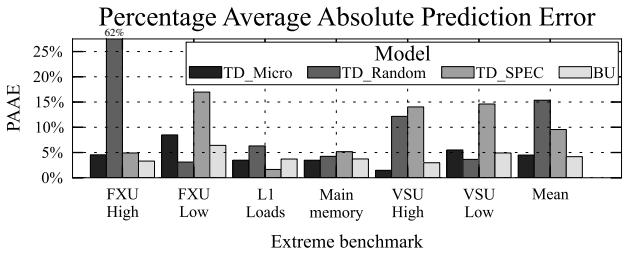


Figure 7: Percentage average absolute errors of the models generated for all configurations on the extreme situations analyzed.

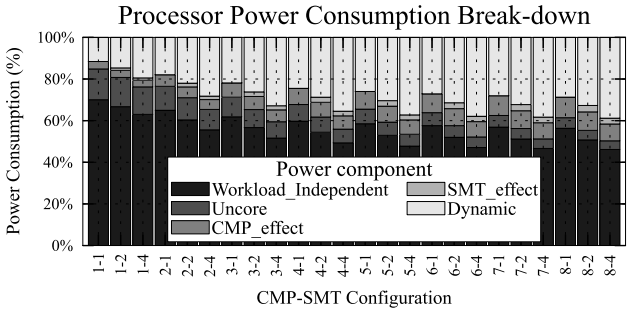


Figure 8: Average per component power consumption breakdown of the SPEC CPU2006 benchmarks for all the configurations of the processor.

training set systematically generated using MicroProbe. Finally, we use the extra information provided by the model to study the SMT/CMP effects on power consumption.

5. POWER7 energy-based instruction taxonomy

Another important area where MicroProbe is useful is in the low-level characterization of architectures. MicroProbe's bootstrap process explained in Section 2.1.2 automatically gathers per instruction micro-architecture information such as latency, throughput or energy per instruction (EPI). This information can be analyzed to generate, for instance, an instruction level EPI characterization.

An instruction-level EPI characterization is beneficial in a wide set of situations. For example, this is necessary for understanding tuning opportunities for hardware implementation of instructions or for improving compiler instruction selection algorithms. This characterization is also useful for guiding micro-benchmark generation policies when searching for max-power stressmarks as described in Section 6.

This section develops a taxonomy of the POWER7 instructions based on energy per instruction (EPI) and processor activity characteristics. We use the unit-stressing information that is implemented in MicroProbe to classify the instructions in categories based on the functional units that the instructions stress.

The results presented are for the 1-way SMT 8 core configuration. Notice that the EPI values are derived from the overall dynamic processor power consumption. Therefore, they depend on the processor configuration (i.e. number of cores and SMT mode) used. EPI values also depend on the input data used, which we randomized. We do not observe any significant variations in EPI when we randomly change the input values. This agrees with prior published results [44]. However, zero input data values sometimes result in a significant reduction in EPI, up to 40% in some cases.

Table 3 shows the core IPC and normalized EPIs of three instructions for each category defined. Categories are named after the functional units that they stress[2]. We group these categories to simplify the explanation. Category EPI column is normalized to the minimum EPI within the category, whereas global EPI column is normalized to the minimum EPI among all the categories. This simplifies the comparisons between instructions and categories of instructions. The top instruction in each category is the one with higher IPC*EPI product within the category. The other two instructions are selected examples with the same IPC but notable differences in EPI.

Analyzing by categories, we can see that the memory operations with *side-effects* (i.e. those that stress other units apart from the LSU) are the ones with higher EPI. The reason is twofold: (a) these types of instructions activate more functional units. For instance, the vector store operations use the LSU unit (address generation) and the VSU (data propagation of the stored value); and (b) these instructions exhibit a lower IPC —each instruction takes more time to be executed— and as a result, they are less efficient.

Overall, the simple integer operations are the most efficient. The reason is that this type of operations is the most common and therefore the execution is highly optimized. For instance, the load store unit (LSU) of the POWER7 is able to execute these simple integer operations. This allows the program to obtain a high IPC, thus lowering the EPI metric.

Analyzing each category, there are important EPI differences between instructions within the same category. This is observed even in the case where the instructions exhibit the same IPC. For instance, in the VSU category, the *xvmaddadp* instruction has a 75% higher

[2]FXU: fixed point unit (integer), LSU: load store unit (memory operations) and VSU: vector scalar unit (vector, float and decimal operations).

Category	Instr.	Core IPC	Normalized EPI	
			Global	Category
Functional units				
FXU	*mulldo*	1.40	2.60	2.60
	subf	2.00	1.69	1.69
	addic	2.00	1.00	1.00
LSU	*lxvw4x*	1.68	2.88	1.35
	lvewx	1.68	2.81	1.31
	lbz	1.68	2.14	1.00
VSU	*xvnmsubmdp*	2.00	2.35	1.78
	xvmaddadp	2.00	2.31	1.75
	xstsqrtdp	2.00	1.32	1.00
Simple integer operations				
FXU or LSU	*add*	3.50	1.73	1.49
	nor	3.50	1.58	1.36
	and	3.50	1.16	1.00
Integer memory operations				
LSU and FXU	*ldux*	1.00	5.12	1.21
	lwax	1.00	5.01	1.18
	lfsu	1.00	4.24	1.00
LSU and 2FXU	*lhaux*	1.00	5.51	1.15
	lwaux	1.00	5.29	1.10
	lhau	1.00	4.80	1.00
Vector/Float/Decimal memory operations				
LSU and VSU	*stxvw4x*	0.48	8.36	1.40
	stxsdx	0.48	7.16	1.20
	stfd	0.48	5.97	1.00
LSU and VSU and FXU	*stfsux*	0.48	10.00	1.19
	stfdux	0.48	9.49	1.13
	stfdu	0.48	8.40	1.00

Table 3: Taxonomy of POWER7 instructions based on energy per instruction (EPI) and functional unit usage. Core IPC, category EPI normalized to minimum EPI within the category and global EPI, normalized to *addic* EPI, the minimum shown in the table. The top instruction within each category is the one with higher IPC*EPI product. The other two instructions have the same core IPC but notably different EPI which demonstrates the high power consumption variability between instruction types, even in the same category.

EPI than the *xstsqrtdp* instruction. Similar observations can be seen in the rest of categories. These observations confirm the differences in energy consumption across various instruction types.

In summary, we use MicroProbe to generate an instruction-level EPI characterization of a POWER7 platform. The characterization helps us to understand better the energy trade-offs of the underlying architecture. In particular, the variability seen in the EPI results —even across instructions that use the same functional unit at the same utilization level— highlights the importance of taking into account such variations when generating power/energy aware code.

6. Max-power stressmark generation

Max-power stressmarks are very important for computer architects to make early-stage design decisions such as the design of the package and the power delivery network. Existing systematic max-power stressmarks generators rely on time-consuming genetic algorithm based design space explorations [20, 21, 33, 40]. These solutions use abstract workload models (e.g. %integer, %loads, %stores, etc.) and expert-defined design spaces to make the search of the solution tractable. They therefore provide a 'black-box' solution where intimate knowledge of the architecture is not required. This benefit comes at the expense of losing some discriminating opportunities.

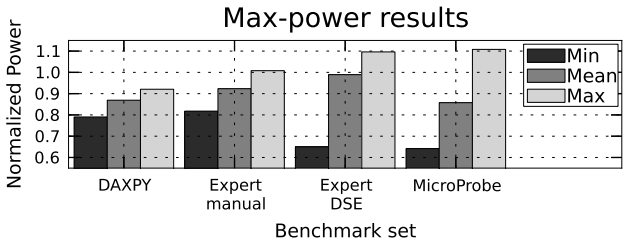


Figure 9: Max, Mean and Min power results for each stressmark set executed. Results are normalized to the maximum power exhibited by one of the SPEC CPU2006 benchmarks during its execution.

For instance, during the selection of instructions, they do not take into account the important differences in power consumption that we have shown in Section 5.

In this section, we show how MicroProbe is used as a 'white-box' framework to help an expert in the process of generating a max-power stressmark in a real measurement context, where the number of design points to explore is a practical limiting factor. In the end, we show that with proper heuristics, the entire process can be fully automated.

We focus this case study on finding the sequence of 6 instructions that when replicated within an endless loop of 4K instructions and executed concurrently on all the available hardware threads maximize the power consumption. The rationale is that basic knowledge in the field suggests that in order to generate a max-power stressmark one should maximize the activity (i.e. maximize the IPC) and maximize the number of functional units used, avoiding pipeline stalls and resource contention (i.e. no dependencies and no memory misses).

Previous work [21] suggests that it would be possible to achieve higher power consumption by executing heterogeneous workloads that stress the different parts of the processor (caches, interconnection network, etc.). We leave the exploration of these options to our future work. We focus this case study on the benefits of using the micro-architecture semantics when generating max-power stressmarks. The fact that we are consistently able to exceed expert level manually generated max-power stressmarks is reassuring.

First, we hand-craft some micro-benchmarks using the *mullw*, *xvmaddadp*, *lxvd2x* instructions. The rationale behind the selection of these instructions is to stress the FXU, the VSU and LSU units using the instructions with a wider data-path (or more complexity) and higher throughput (maximize IPC). This procedure is what a stressmark developer with some expertise in the target micro-architecture would do without support frameworks like MicroProbe. We call this micro-benchmark set as the *Expert Manual* set.

Second, since it is not practical to generate manually all the 540 possible combinations, we use the DSE support of MicroProbe to generate all the combinations of the expert selected instructions automatically. We call this micro-benchmark set as the *Expert DSE* set.

Lastly, instead of relying in our expert to select the instructions, we rely on MicroProbe to select the instruction candidates. We instruct MicroProbe to select the instructions with the highest IPC*EPI product within each functional unit category. This heuristic selects the instructions with a balanced trade-off between EPI and IPC, penalizing instructions with high IPC but low EPI and vice versa. The automatically selected instructions are the top ones shown in the FXU, LSU and VSU categories of Table 3. We call this micro-benchmark set as the *MicroProbe* set.

We execute the three micro-benchmark sets in the three available SMT modes. In addition, various DAXPY kernels with different L1 contained memory foot-prints are also executed. This computational kernel is commonly used as a stressmark. Figure 9 shows the maximum, minimum and average power consumption of each micro-benchmark set. Results are normalized to the maximum power exhibited by one of the SPEC CPU2006 benchmark during its execution.

We observe that with a bit of intuition the expert is able to conceive hand-crafted stressmarks (*Expert manual*) that are as good as the max-power of SPEC CPU2006. However, these stressmarks are still around 10% below the one achieved by the *Expert DSE* set —even though they use the same instruction types and exhibit the same IPC.

Examining closely, we find 181 different stressmarks within the *Expert DSE* set that achieve the maximum core IPC. The minimum and the maximum power exhibited by them is 7% below and 9.6% above the baseline, respectively. These results depict how difficult it is to search for the optimal power stressmark. Even while achieving the same maximum IPC with the very same instruction types, the actual instruction sequence can affect the power consumption quite considerably.

The *MicroProbe* stressmark set, automatically defined using the functional unit, IPC and EPI information as heuristics, achieves similar results as the *Expert DSE*. In fact, it improves the max-power stressmark *Expert DSE* by approximately 1 percentage point. Also, this exceeds the maximum power observed during the execution of the entire SPEC CPU2006 suite by a 10.7%. These results confirm that EPI, IPC plus functional unit information provide good heuristics to constrain the DSE and systematize the max-power stressmark generation process without requiring expert knowledge.

Finally, the fact that systematically generated stressmarks slightly outperform the hand-crafted stress tests generated by an expert, confirms the utility of the proposed approach. Moreover, in a real measurement context, being able to constrain the search space to the actual points of interest is crucial in avoiding practical limitations posed by design space explosion.

7. Related work

Benchmarks and Micro-benchmarks: From the pioneering Whetstone [16] and Dhrystone [46] to current benchmark suites such as the SPEC CPU2006 [25], benchmarks are used for both academic research and comparative evaluation of existing solutions. Moreover, specifically designed benchmarks, named micro-benchmarks, are needed in several situations. For instance, they have been used to reverse engineer structure latencies [24] or branch organization [37,45], to evaluate performance, power or thermal efficiency [15,22,28,38] or to generate and calibrate models [8,9,12].

Micro-benchmark Generation Frameworks: The need of a systematic method to generate micro-benchmarks was identified back in the 1980's [47,48]. The number of frameworks proposed since then has been growing continuously corroborating their importance for the community. In contrast to our adaptive framework, particular solutions —without the micro-architecture semantics of MicroProbe— were developed for different purposes: to generate synthetic micro-benchmarks [2–4, 26], to be able to reproduce proprietary application behavior [30,32], to perform architecture explorations [31], to generate power or reliability stress tests [20,21,33,39], to evaluate energy efficiency of systems [13], or to model cache behavior [1].

Counter-Based Processor Power Models: Most of the previous work on counter-based power modeling uses top-down approaches to model processor power consumption [5, 10, 11, 23, 41]. As a result, they lose the level of decomposability provided by bottom-up approaches. Moreover, we only found the work of Jimenez *et al.* [29] proposing a top-down model for a SMT/CMP processor, the POWER6.

Regarding bottom-up modeling methods, Isci *et al.* [27] was the first to propose a heuristic-based bottom-up modeling method using as heuristic the area size of the functional units. Bertran *et al.* [7–9] then proposed a bottom-up modeling method, entirely based on micro-benchmarks. Nevertheless, none of these bottom-up methods modeled a CMP/SMT system such as the POWER7. Finally, Bircher *et al.* [10, 11] present a system-level bottom-up method to derive the power breakdown of the entire system (cpu, memory, disks, etc.).

Max-Power Stressmark Generation: The systematization of the generation of max-power stressmarks has been investigated for different environments. In [33], the authors present a micro-benchmark generation framework and show its utility for generating processor max-power stressmarks. In that work, the design space is defined by an abstract workload model. Then, genetic algorithms are used to find an optimal solution. Ganesan *et al.* [20] present a similar approach but targeting overall system power consumption, including processor and memory. The same authors extended the work to multi-cores [21] showing that when taking into account processor and memory power consumption, simple parallel execution of single core max-power stressmarks, do not exhibit the maximum power consumption. Our work in max-power stressmark case study is orthogonal to these works, since we focus on the importance of using micro-architecture semantics to constrain the search within the design space. We believe that these prior 'black-box' proposals are significantly improved by taking into account the extra information provided by MicroProbe.

8. Conclusion

In this paper, we present an adaptive micro-benchmark generation framework (MicroProbe), with three salient features that distinguish it from prior work: detailed knowledge of low-level micro-architecture semantics, flexible code generation support and integrated design space exploration support. To highlight these features of MicroProbe, we present experimental results centered around an IBM POWER7 CMP/SMT system. First, we produce a MicroProbe-driven empirical power model that estimates the power consumption of the SPEC CPU2006 benchmarks with average errors that are below 2.3%. Then, we conclude that micro-benchmark trained power models are more reliable across a broader range of contexts (normal and extreme power activities). We also use the framework to derive a taxonomy of POWER7 instructions based on energy-per-instruction (EPI). The characterization highlights the differences in energy consumption between instructions. Finally, we propose a method —based on EPI, IPC and functional unit information— to systematize the generation of power stress tests. The method is used to derive a stress test that exhibits a 10.7% increase in processor power over the maximum power seen during the execution of the SPEC CPU2006 benchmarks.

Acknowledgements

From the Barcelona Supercomputing Center (BSC) and the Universitat Politècnica de Catalunya (UPC) side, this work was supported by the Spanish Ministry of Education [Contracts #TIN-2007-60625 and #CSD2007-00050]; the Generalitat de Catalunya [Contract #2009-SGR-980]; the European Commission in the context of the HiPEAC Network of Excellence [Contracts EU FP7/ICT #217068 and #287759]; and the BSC-IBM collaboration agreement. From the International Business Machines (IBM) Corporation side, this work was supported by DARPA under the Computational Reliability Project [Contract #N66001-11-C-4027] and by the Lawrence Livermore National Laboratory (LLNS) under the BlueGene/Q Project [Subcontract #B554331]. Finally, we would like to thank Srilatha Manne and the anonymous reviewers for their comments and feedback.

References

[1] G. Balakrishnan *et al.*, "WEST: Cloning data cache behavior using stochastic traces," in *Proc. of HPCA'12*, pp. 1–12, Feb 2012.

[2] R. H. Bell Jr. *et al.*, "Efficient power analysis using synthetic testcases," in *Proc. of IISWC'05*, pp. 110–118, Oct 2005.

[3] R. H. Bell Jr. *et al.*, "Automatic testcase synthesis and performance model validation for high performance PowerPC processors," in *Proc. of IISWC'06*, pp. 154–165, Mar 2006.

[4] R. H. Bell Jr. *et al.*, "Improved automatic testcase synthesis for performance model validation," in *Proc. of ICS'05*, pp. 111–120, Jun 2005.

[5] F. Bellosa, "The benefits of event-driven energy accounting in power-sensitive systems," in *Proc. of EW'00*, pp. 37–42, Sep 2000.

[6] R. Bertran *et al.*, "POTRA: A framework for building power models for next generation multicore architectures," in *Proc. of SIGMETRICS'12*, pp. 427–428, Jun 2012.

[7] R. Bertran *et al.*, "Counter-based power modeling methods: Top-down vs bottom-up," *The Computer Journal*, vol. 99, pp. 1–16, Aug 2012.

[8] R. Bertran *et al.*, "A systematic methodology to generate decomposable and responsive power models for CMPs," *IEEE Trans. on Comp.*, vol. 99, pp. 1–14, Apr 2012.

[9] R. Bertran *et al.*, "Decomposable and responsive power models for multicore processors using performance counters," in *Proc. of ICS'10*, pp. 147–158, Jun 2010.

[10] W. Bircher *et al.*, "Complete system power estimation: A trickle-down approach based on performance events," in *Proc. of ISPASS'07*, pp. 158–168, Apr 2007.

[11] W. Bircher *et al.*, "Complete system power estimation using processor performance events," *IEEE Trans. on Comp.*, vol. 61, no. 4, pp. 563–577, Apr 2011.

[12] B. Black *et al.*, "Calibration of microprocessor performance models," *Computer*, vol. 31, no. 5, pp. 59–65, May 1998.

[13] K. D. Bois *et al.*, "SWEEP: Evaluating computer system energy efficiency using synthetic workloads," in *Proc. of HIPEAC'11*, pp. 159–166, Jan 2011.

[14] P. Bose *et al.*, "Bounds modelling and compiler optimizations for superscalar performance tuning," *J. Syst. Archit.*, vol. 45, no. 12–13, pp. 1111–1137, Jun 1999.

[15] D. Bull *et al.*, "A power-efficient 32b ARM ISA processor using timing-error detection and correction for transient-error tolerance and adaptation to PVT variation," in *Proc. of ISSCC'10*, pp. 284–285, Feb 2010.

[16] H. J. Curnow *et al.*, "A synthetic benchmark," *The Computer Journal*, vol. 19, no. 1, pp. 43–49, Feb 1976.

[17] S. Eranian, "Linux has a generic performance monitoring API!" in *Proc. of CSCADS'09*, p. 1, Jul 2009.

[18] L. V. Ertvelde *et al.*, "Benchmark synthesis for architecture and compiler exploration," in *Proc. of IISWC'10*, pp. 1–11, Dec 2010.

[19] M. Floyd *et al.*, "Adaptive energy-management features of the IBM POWER7 chip," *IBM J. Res. & Dev.*, vol. 55, no. 3, pp. 276–293, May 2011.

[20] K. Ganesan *et al.*, "SYstem-level Max POwer (SYMPO): A systematic approach for escalating system-level power consumption using synthetic benchmarks," in *Proc. of PACT'10*, pp. 19–28, Sep 2010.

[21] K. Ganesan *et al.*, "MAximum Multicore POwer (MAMPO): an automatic multithreaded synthetic power virus generation framework for multicore systems," in *Proc. of SC'11*, pp. 1–12, Nov 2011.

[22] G. Gerosa *et al.*, "A sub-2W low power IA processor for mobile internet devices in 45nm high-k metal gate CMOS," *J. of Solid-State Circ.*, 2009.

[23] B. Goel *et al.*, "Portable, scalable, per-core power estimation for intelligent resource management," in *Proc. of GREEN'10*, pp. 135–146, Aug 2010.

[24] J. Gonzalez-Dominguez *et al.*, "Servet: A benchmark suite for autotuning on multicore clusters," in *Proc. of IPDPS'10*, pp. 1–9, Apr 2010.

[25] J. L. Henning, "SPEC CPU2006 benchmark descriptions," *ACM SIGARCH News*, vol. 34, no. Sep, pp. 1–17, 4 2006.

[26] C. Hsieh *et al.*, "Microprocessor power estimation using profile-driven program synthesis," *IEEE Trans. on Comp.-Aided Design. of Integ. Cir. & Sys.*, vol. 17, no. 11, pp. 1080–1089, Nov 1998.

[27] C. Isci *et al.*, "Runtime power monitoring in high-end processors: methodology and empirical data," in *Proc. of MICRO'03*, pp. 96–108, Dec 2003.

[28] R. Iyer *et al.*, "Comparing the memory system performance of the HP V-class and SGI Origin 2000 multiprocessors using microbenchmarks and scientific applications," in *Proc. of ICS'99*, pp. 339–347, Jun 1999.

[29] V. Jiménez *et al.*, "Power and thermal characterization of POWER6 system," in *Proc. of PACT'10*, pp. 7–18, Sep 2010.

[30] A. Joshi *et al.*, "Performance cloning: A technique for disseminating proprietary applications as benchmarks," in *Proc. of IISWC'06*, pp. 105–115, Oct 2006.

[31] A. Joshi *et al.*, "The return of synthetic benchmarks," in *Proc. of SPEC Benchmark Workshop*, pp. 1–11, Jan 2008.

[32] A. Joshi *et al.*, "Distilling the essence of proprietary workloads into miniature benchmarks," *ACM Trans. on Arch. & Code Opt.*, vol. 5, no. 2, pp. 1–33, Sep 2008.

[33] A. Joshi *et al.*, "Automated microprocessor stressmark generation," in *Proc. of HPCA'08*, pp. 229–239, Feb 2008.

[34] Y. Kim *et al.*, "Automated di/dt stressmark generation for microprocessor power delivery networks," in *Proc. of ISLPED'11*, pp. 253–258, Aug 2011.

[35] T. Li *et al.*, "Run-time modeling and estimation of operating system power consumption," pp. 160–171, Jun 2003.

[36] IBM staff, "Power ISA™. Version 2.06 Revision B," Jul 2010, [Online] http://www.power.org/resources/reading/.

[37] M. Milenkovic *et al.*, "Microbenchmarks for determining branch predictor organization," *Softw. Pract. Exper.*, vol. 34, no. 5, pp. 465–487, Apr 2004.

[38] S. Naffziger *et al.*, "The implementation of a 2-core, multi-threaded itanium family processor," *J. of Solid-State Circ.*, vol. 41, no. 1, pp. 197–209, Jan 2006.

[39] A. Nair *et al.*, "AVF stressmark: Towards an automated methodology for bounding the worst-case vulnerability to soft errors," in *Proc. of MICRO'10*, pp. 125–136, Dec 2010.

[40] S. Polfliet *et al.*, "Automated full-system power characterization," *IEEE Micro*, vol. 31, no. 3, pp. 46–59, May 2011.

[41] K. Singh *et al.*, "Real time power estimation and thread scheduling via performance counters," *ACM SIGARCH News*, vol. 37, no. 2, pp. 46–55, Jul 2008.

[42] B. Sinharoy *et al.*, "IBM POWER7 multicore server processor," *IBM J. Res. & Dev.*, vol. 55, no. 3, pp. 1–29, May 2011.

[43] D. C. Snowdon *et al.*, "Accurate on-line prediction of processor and memory energy usage under voltage scaling," in *Proc. of EMSOFT'07*, pp. 84–93, Oct 2007.

[44] V. Tiwari *et al.*, "Instruction level power analysis and optimization of software," in *Proc. of VLSI'96*, pp. 326–328, Jan 1996.

[45] V. Uzelac *et al.*, "Experiment flows and microbenchmarks for reverse engineering of branch predictor structures," in *Proc. of ISPASS'09*, pp. 207–217, Apr 2009.

[46] R. P. Weicker, "Dhrystone: a synthetic systems programming benchmark," *Comm. of ACM*, vol. 27, no. 10, pp. 1013–1030, Oct 1984.

[47] W. S. Wong *et al.*, "Synthesizing benchmarks with appropriate instruction mix and locality," in *Proc. of ICCA'87*, pp. 1–12, Jun 1987.

[48] W. S. Wong *et al.*, "Benchmark Synthesis Using the LRU Cache Hit Function," *IEEE Trans. on Comp.*, vol. 37, no. 6, pp. 637–645, Jun 1988.

[49] W. Wu *et al.*, "A systematic method for functional unit power estimation in microprocessors," in *Proc. of DAC'06*, pp. 554–557, Jul 2006.

2012 IEEE/ACM 45th Annual International Symposium on Microarchitecture

AUDIT: Stress Testing the Automatic Way

Youngtaek Kim Lizy Kurian John
Department of Electrical & Computer Engineering
The University of Texas at Austin
Austin, TX, USA
young.kim@utexas.edu ljohn@ece.utexas.edu

Sanjay Pant[1] Srilatha Manne[2] Michael Schulte[3]
W. Lloyd Bircher[3] Madhu S. Sibi Govindan[3]
Advanced Micro Devices, Inc.
[1]Fort Collins,CO, [2]Portland,OR, and [3]Austin,TX, USA
{sanjay.pant, srilatha.manne, michael.schulte,
lloyd.bircher, sibi.govindan}@amd.com

Abstract—**Sudden variations in current (large di/dt) can lead to significant power supply voltage droops and timing errors in modern microprocessors. Several papers discuss the complexity involved with developing test programs, also known as stressmarks, to stress the system. Authors of these papers produced tools and methodologies to generate stressmarks automatically using techniques such as integer linear programming or genetic algorithms. However, nearly all of the previous work took place in the context of single-core systems, and results were collected and analyzed using cycle-level simulators.**

In this paper, we measure and analyze di/dt issues on state-of-the-art multi-core x86 systems using real hardware rather than simulators. We build on an existing single-core stressmark generation tool to develop an AUtomated DI/dT stressmark generation framework, referred to as AUDIT, to generate di/dt stressmarks quickly and effectively for multi-core systems. We showcase AUDIT's capabilities to adjust to microarchitectural and architectural changes. We also present a dithering algorithm to address thread alignment issues on multi-core processors. We compare standard benchmarks, existing di/dt stressmarks, and AUDIT-generated stressmarks executing on multi-threaded, multi-core systems with complex out-of-order pipelines. Finally, we show how stress analysis using simulators may lead to flawed insights about di/dt issues.

Keywords- di/dt; inductive noise; stressmark generation; voltage droop; power distribution network; low power; genetic algorithm; hardware measurement

I. INTRODUCTION

Reliable operation is a fundamental requirement of processor design. The processor must work correctly across a range of applications regardless of process variations, voltage variations, environmental noise, and the aging of the system. Voltage margins are introduced to compensate for potential supply voltage fluctuations in the system. Fluctuations caused during program execution must stay within allowed margins, as shown in Fig. 1. These margins need to be designed carefully to be power-efficient and prevent malfunctions from program-induced voltage fluctuations.

Specialized benchmarks, referred to as stressmarks, are used to study the susceptibility of processors to voltage fluctuations. Stressmarks may or may not be used to set the voltage margins; however, they are necessary to develop an understanding of the susceptibilities of the system being analyzed. Stressmarks are either collected from benchmarks that have produced high di/dt stresses in the past, or they are specially designed to induce voltage fluctuations in microprocessors.

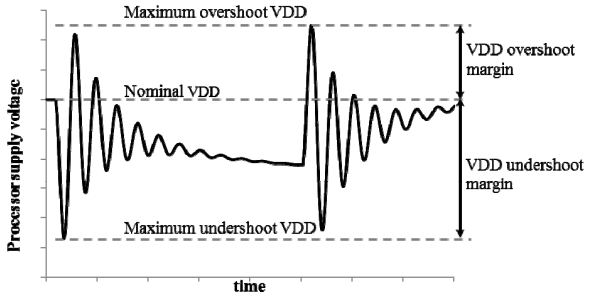


Figure 1. Voltage margins to manage fluctuations.

Stressmarks are difficult to generate manually; past work addressed the complexities involved with stressmark generation and developed tools and methodologies to generate stressmarks automatically [11][12][13]. Other work addressed mitigating droops that occur due to various architectural events [3][5][8][9][10][16][21] or leveraging the margin between typical benchmarks and stressmarks to operate with reduced voltage margins [7][15][22]. In these papers, many of the results were collected and/or analyzed using cycle-level simulators enhanced with power models. In addition, except for a few papers [6][16][23], the authors did not consider or investigate the complexities of multi-core processors with multi-threaded resources or operating system interactions.

In this paper, we address di/dt analysis and stressmark generation using hardware rather than simulators. We use state-of-the-art, multi-threaded, multi-core x86 hardware with complex out-of-order pipelines to produce a detailed analysis of voltage droops and failure points for standard benchmarks and stressmarks. We expand the single-core di/dt stressmark generation tool by Kim and John [13] to develop an AUtomated DI/dT stressmark generation framework, referred to as AUDIT, that efficiently generates stressmarks for multi-core x86 systems using simulators or hardware. We compare the effectiveness of AUDIT stressmarks against other manually generated stressmarks using state-of-the-art hardware. We also show the impact of mitigation mechanisms in hardware and AUDIT's flexibility

1072-4451/12 $26.00 © 2012 IEEE
DOI 10.1109/MICRO.2012.28

IEEE computer society

in handling microarchitectural and architectural changes. Finally, we show how simulators fail to capture the nuances of di/dt stresses and their behavior by demonstrating that (1) droop measurements do not always correlate to failure points in real hardware, (2) OS interference can influence how loops align in multi-core systems, and (3) alignment and synchronization that occur in simulators may not be repeatable in real hardware due to natural perturbations in how the system operates.

In the rest of this paper, Section 2 discusses voltage droops, droop management techniques, and manual di/dt stressmark generation. Section 3 presents the AUDIT stressmark generation framework and the dithering algorithm for multi-core stressmark alignment. Section 4 details our experimental set-up. Section 5 presents results and analysis of droop and failure characteristics of standard benchmarks, manual stressmarks, and AUDIT stressmarks. Section 6 discusses related work, and Section 7 concludes the paper.

II. Background

The power distribution network (PDN) of a typical microprocessor consists of inductive and resistive elements on the motherboard (MB), package, and die. The parasitic resistance of the network causes a droop (IR drop) in the power supply proportional to the current drawn from the network. In addition, the inductance in the network causes undershoots and overshoots in the power supply (referred to as the di/dt droop), which depend on the rate of change of the load-current.

To mitigate the inductive noise in the power supply, decoupling capacitance (decap) is added at different locations in the PDN (Fig. 2). The series combination of parasitic inductance (L) and decap (C) results in various resonance frequencies ($1/2\pi\sqrt{LC}$) in the network, as shown in the frequency and time domains in Fig. 3. While single voltage droops caused by non-repeating di/dt events may be harmful, droops that repeat at the resonance frequency of the system grow to high amplitudes and are much more likely to cause catastrophic failures.

The prominent resonance frequencies shown in Fig. 3 are the first droop resonance due to the interaction of package and on-die inductance ($L_{pkg2} + L_{die}$) with on-die decap (C_{die}), the second droop resonance due to the interaction of socket and package inductance (L_{pkg1}) with package decap (C_{pkg}), and the third droop resonance due to the interaction of board inductance (L_{MB}) with decap on the board (C_{MB}). A periodically varying load can induce one or more of these resonances and cause excessive undershoots and overshoots. Although second and third droop resonances can also affect the reliability of the system, they are typically smaller in magnitude than first droop resonance [14] and are not evaluated in this work.

The first droop resonance is a strong function of package inductance (L_{pkg2}) and C_{die}, and is typically in the range of 50–200MHz. First droops can be mitigated by explicitly

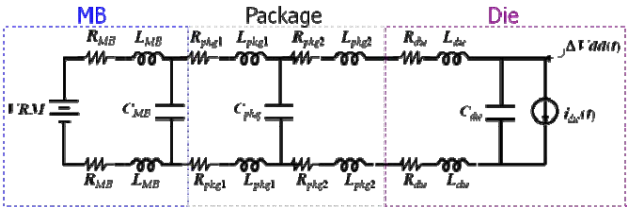


Figure 2. Simplified model of a power distribution network.

adding decap on the die [19]. However, there are limits to the feasibility of this approach due to area constraints and the leakage of the decap. Several techniques that limit the rate of change of activity in the processor are effective in suppressing first droops [8][9][10][21], but they may have a negative impact on performance.

If a single high-di/dt event occurs due to the machine executing a pattern of low-power instructions followed by a pattern of high-power instructions, there will be a droop in supply voltage, but the droop will taper off quickly, as shown on the left side of Fig. 4. However, a pattern that repeats periodically at the resonant frequency of the PDN (right side of Fig. 4) will build in amplitude and generate a larger droop than a single event, thereby increasing the risk of system failure. We cover first droop excitation and first droop resonance in our analysis.

III. Automatic Stressmark Generation

Stressmarks are common in both academia and industry. Several methods for automation have been proposed because generating stressmarks is a tedious and time-consuming process for designers [11][12][13]. Although such stressmarks may be representative of worst-case behavior, understanding this behavior is important when determining voltage and frequency margins and developing droop mitigation mechanisms to ensure reliable operation. They expose sensitivities and critical paths in the pipeline, as we will show in Section 5.

Automatic stressmark generation using a genetic algorithm (GA) was first proposed in [11] and expanded to cover di/dt resonance issues explicitly by Kim and John in [13]. This paper expands upon that work by:

- generating di/dt stressmarks for complex, multi-core x86 processors rather than single-core Alpha processors,
- utilizing real hardware rather than simulation to analyze full applications under realistic conditions,
- presenting a dithering algorithm to produce alignment across threads in a multi-core environment,
- extending AUDIT to automatically detect the resonant frequency of the system and generate both first droop excitation and first droop resonance stressmarks, and
- generating AUDIT stressmarks for varying processors, processor configurations, and operating conditions.

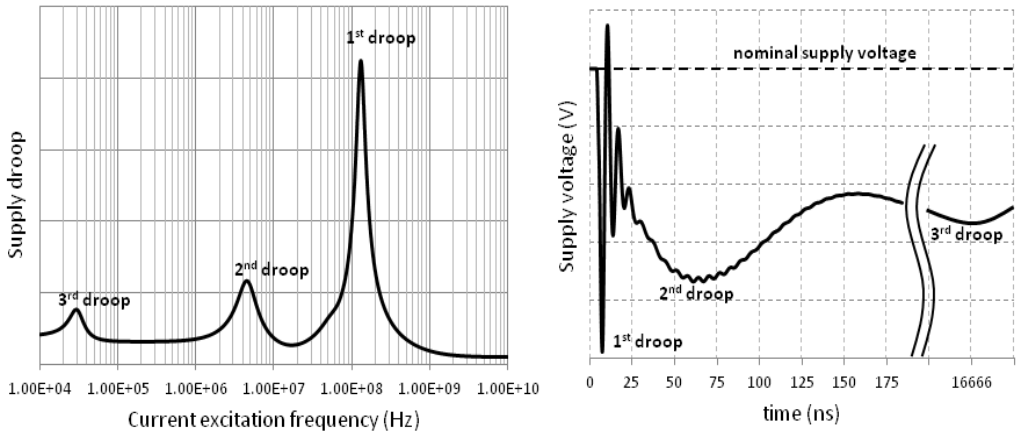


Figure 3. First, second, and third resonance droops in the frequency and time domains.

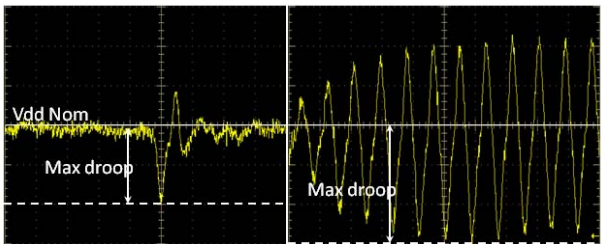


Figure 4. First droop excitation and first droop resonance generated using the AUDIT framework.

Fig. 5 shows the basic framework for AUDIT. AUDIT takes as input the instructions used to generate the stressmark and some control parameters such as the cost function and exit conditions. This information is fed to a code generator to produce a population of potential stressmarks. The initial population of stressmarks either can be generated randomly or seeded with existing benchmarks or stressmarks to improve the convergence rate.

Fig. 5 includes two possible paths for stressmark generation, simulation and hardware. With the simulation path, the voltage droops of generated instruction sequences are evaluated using a cycle-accurate simulator that produces current draw information followed by SPICE simulation. This path is most appropriate when hardware for performing di/dt stressmark generation is not available. With this approach, the assembly code instruction sequence is compiled into a simulator-friendly format (e.g., x86 binaries). The compiled code is executed on the cycle-accurate simulator and every cycle the simulator calculates the current draw of the processor based on the activity of internal modules of the processor. This methodology is similar to that used in [5][10][21]. AUDIT converts the per-cycle current profile into a current sink in HSPICE simulation using a lumped RLC model of the PDN. The HSPICE simulation produces a series of voltage droops over time from which the maximum voltage droop can be obtained.

With the hardware path, the stressmarks are run on a processor board and measurement tools capture voltage droops, power dissipation, and any other information necessary to evaluate the cost function of the stressmark. The stressmarks and their associated cost values are fed to the GA for further refinement until the exit conditions are met (e.g., the maximum voltage droop produced by AUDIT does not increase for several generations[1]). In prior research [13], we used the simulation-based path, whereas in this paper, the hardware-based path is used.

We use the same methodology as in [13] to generate the opcode sequence. However, there are some additional complexities because we are using x86 hardware and generating stressmarks for multi-core systems. First, we observe that data values used for the stressmark have a measureable impact on the final droop values, on the order of 10%. To take data values into account, we use an alternating set of values that guarantee maximum toggling between one instruction and the next executing on the same functional unit. Second, the resonance frequencies of the system can vary across different boards or even within the same board if the components of the board change (e.g., using a different processor on the same board, as is done later in the paper). Therefore, we extend AUDIT to do a sweep for the resonance frequency before attempting to generate a first-order resonant droop.

To determine the resonance frequency, AUDIT constructs a trivial stressmark consisting of a loop of high-power instructions and NOP instructions. It varies the number of cycles in the loop to determine the length that produces the worst-case droop. The number of cycles in the loop that produces the worst-case droop exercises the

[1] The cost function provided to AUDIT can vary. Although we focus on maximizing voltage droops in this paper, other, more complex cost functions such as maximizing the droop while minimizing the average power or maximizing the droop while exercising sensitive paths in the microarchitecture are also feasible and easy to implement.

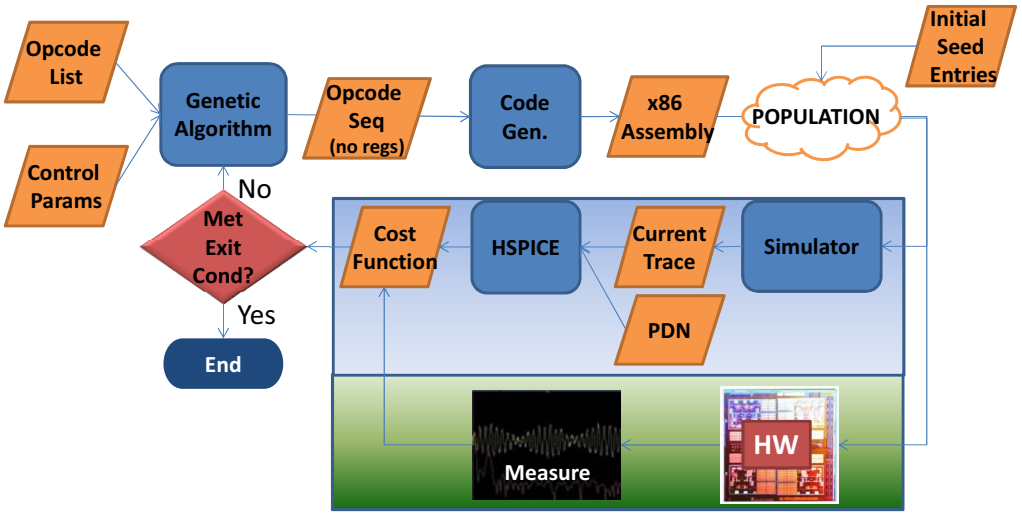


Figure 5. AUDIT framework for di/dt stressmark generation using simulators and hardware.

resonant frequency of the processor. Finally, we have to deal with the complexities of multi-core stressmark generation.

A. *Thread alignment in multi-core systems*

As noted in other papers [3][6][16][23], multiple threads running simultaneously can have a constructive or destructive impact on droops. If the threads align correctly, they produce significantly larger droops than without alignment. At first glance, thread alignment would seem to be a low-probability event in multi-core machines with complex, out-of-order cores and shared and non-shared resources. However, our analysis shows that alignment occurs relatively often when the stressmark consists of short loops due to natural perturbations in the threads caused by OS thread scheduling. We refer to this phenomenon as natural dithering.

Fig. 6 shows an example of natural dithering over the course of 100 ms when running a four-threaded resonant stressmark in which the threads are the same and consist of short loops. Each major grid point represents 10 ms and the y axis shows measured processor voltage (V_{dd}) values using a 100 megasamples/second (MS/s) sampling rate.

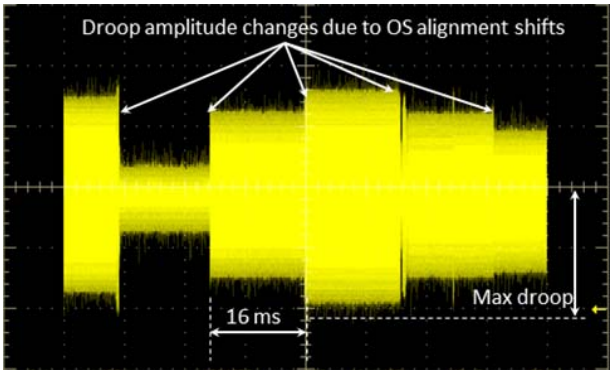


Figure 6. Scope shot of natural dithering due to OS interactions for resonant stressmark over a period of 100 ms.

Approximately every 16 ms, which corresponds to the OS timer tick on Windows systems, V_{dd} variability changes. When the threads align constructively, as is the case near the center point of the scope shot, the droop is maximized.

This data shows that small, repetitive loops occurring across multiple threads at the same time can result in significant di/dt stresses in the system due to natural dithering resulting from OS interaction. This phenomenon would have been difficult, if not impossible, to observe in a simulation environment. This type of behavior is more likely to occur in certain high-performance computing applications that consist of short, repeated loops.

Relying on OS behavior to align threads is not a reliable method to determine the worst-case droop. Hence, we propose a dithering algorithm that guarantees a worst-case droop within a fixed amount of time once OS interrupts are disabled.

B. *Dithering algorithm for guaranteed alignment*

Fig. 7 shows a periodic stress pattern with high- and low-power portions of duration H and L cycles, respectively. This waveform meets the requirements of an ideal di/dt-inducing resonant pattern described in Section 2. This periodic pattern is repeated for M cycles to produce a large resonant droop. The goal of the dithering algorithm is to guarantee that for C cores, the stressmarks running on each core align across all C cores for at least M cycles. Note that a first droop excitation is different in that it requires a low region followed by a high region where the sum of the regions is not necessarily periodic at the resonance frequency.

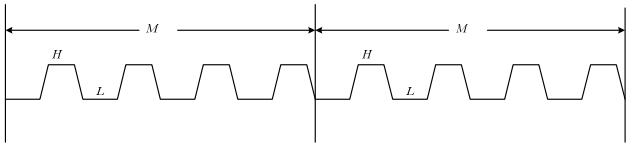


Figure 7. Periodic activity waveform for inducing power supply resonance and large voltage droops.

For a high-low sequence of length $H+L$ cycles running on C cores, the misalignment in cores 1 through $C-1$ can be represented as a $C-1$ dimensional variable $x = (x_1, x_2, ..., x_{C-1})$, where $x_i \in \{0,1,...,L+H-1\}$. Core 0 is considered the reference core. The search space for perfect alignment of all cores is therefore $(L+H)^{(C-1)}$ possible alignments. This search space can be fully traversed in $M \times (L+H)^{(C-1)}$ cycles, where M is the number of cycles required to cause and sustain supply droop resonance.

The dithering algorithm uses the following NOP padding procedure to align the threads and achieve resonance in a processor with C cores:

- Core 0: Apply no dithering and no extra padding of NOPs. Core 0 simply executes the periodic low-high activity sequence shown in Fig. 7 repetitively.
- Core c, where $1 \leq c \leq C-1$: Apply one cycle worth of NOP padding every $M \times (L+H)^{(c-1)}$ cycles.

The maximum number of cycles to guarantee alignment is $M \times (L+H)^{(C-1)}$.

As long as the number of processors is reasonably small, the alignment algorithm works well. However, the time required for alignment becomes prohibitively large for more than four cores. For example, on a 4-GHz system with $L+H=24$ and $M=24 \times 40=960$, the time required to align four cores is 3.3 ms, but eight cores require 18.35 minutes. The alignment must be done for each candidate stressmark in each generation of the GA.

To expand dithering to many-core systems, we use an approximate algorithm that sets a bound on the maximum misalignment between threads. Assume that the maximum mismatch allowed among the activities of different cores is δ cycles. Then, $L+H$ is chosen such that it is a multiple of $(\delta + 1)$ and $(L+H) \times f$ is close to the resonance frequency of the PDN, where f is the operating frequency of the system.

The search space for alignment of all cores within the maximum allowed mismatch of δ cycles then becomes $[(L+H)/(\delta + 1)]^{(C-1)}$, which can be fully traversed in $M \times [(L+H)/(\delta + 1)]^{(C-1)}$ cycles. The dithering algorithm proceeds as before; however, for core c, where $1 \leq c \leq C-1$, $(\delta + 1)$ cycles worth of NOP padding is applied every $M \times k^{(c-1)}$ cycles, where $k = (L+H)/(\delta + 1)$. If we use a δ of 3 in the previous example of eight cores, the maximum time required to reach alignment with the approximate algorithm shrinks from 18.35 minutes to 67 ms per candidate stressmark.

C. Expanding AUDIT for multi-core hardware

The stressmark solution space for AUDIT is a function of the number of cycles in the repeated loop (the loop length), the issue width of the processor, and the number of instructions being evaluated for code generation. The combination of loop length, issue width, and the number of instructions can result in a large solution space. The loop length for first droop resonance is determined by the resonance frequency, which can result in a large solution space. For example, a 3-GHz processor with a resonance frequency of 50 MHz has a loop length of 60 cycles. Assuming a four-wide processor, this results in 240 instruction slots for AUDIT to schedule. In addition, dithering increases AUDIT's run time by requiring a sweep through a large number of alignments for each stressmark, as discussed in Section 3.B.

To converge in a reasonable time (we define reasonable time as being a few hours), we modified AUDIT to use a hierarchical generation policy. First, AUDIT separates each member of the population into a high-power (HP) and low-power (LP) region. Initially, the LP region consists of NOPs. Second, AUDIT breaks the HP region into S replicated sub-blocks of length K. For example, a 24-cycle HP loop can be composed of four ($S = 4$) sub-blocks of length six cycles ($K = 6$). The GA algorithm in AUDIT generates the instructions for each subsection, and the full stressmark -- composed of an HP region of S sub-blocks of length K and an LP region of NOPs -- is evaluated in hardware using the dithering algorithm.

At the end of the AUDIT run for the HP region, we have a stressmark that has been synthesized to produce high power for the HP region of the stressmark. We also evaluated using AUDIT to generate the LP region of the stressmark using long-latency operations with dependencies as proposed in [10]. However, for the system we evaluated, a sequence of NOPs produced comparable power values to a sequence of long-latency, dependent operations. NOPs are designed to be very low-power instructions in our experimental processor, so the rest of the evaluation uses NOPs for the LP portion of the stressmark.

We compared the hierarchical AUDIT implementation to that proposed in [13] and found sub-blocking provided faster convergence as well as better results -- 19% higher droop in less than five hours compared to a 30-hour run without hierarchical generation.

IV. Experimental Set-up

The work in this paper targets the latest multi-core x86-64 processors due to their widespread use. The primary processor utilized for this study consists of four AMD Bulldozer modules with 2 MB of dedicated L2 cache per module and an 8 MB shared L3 cache [4][25].

Each Bulldozer module can execute two threads via a combination of shared and dedicated resources [2]. The front-end and floating-point logic is shared between two threads on the same module; however, the rest of the core

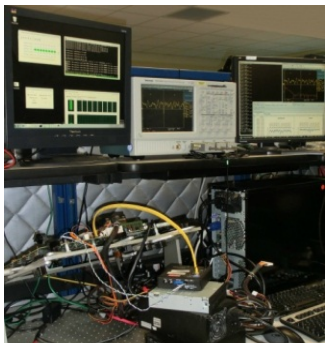


Figure 8. Experimental set-up.

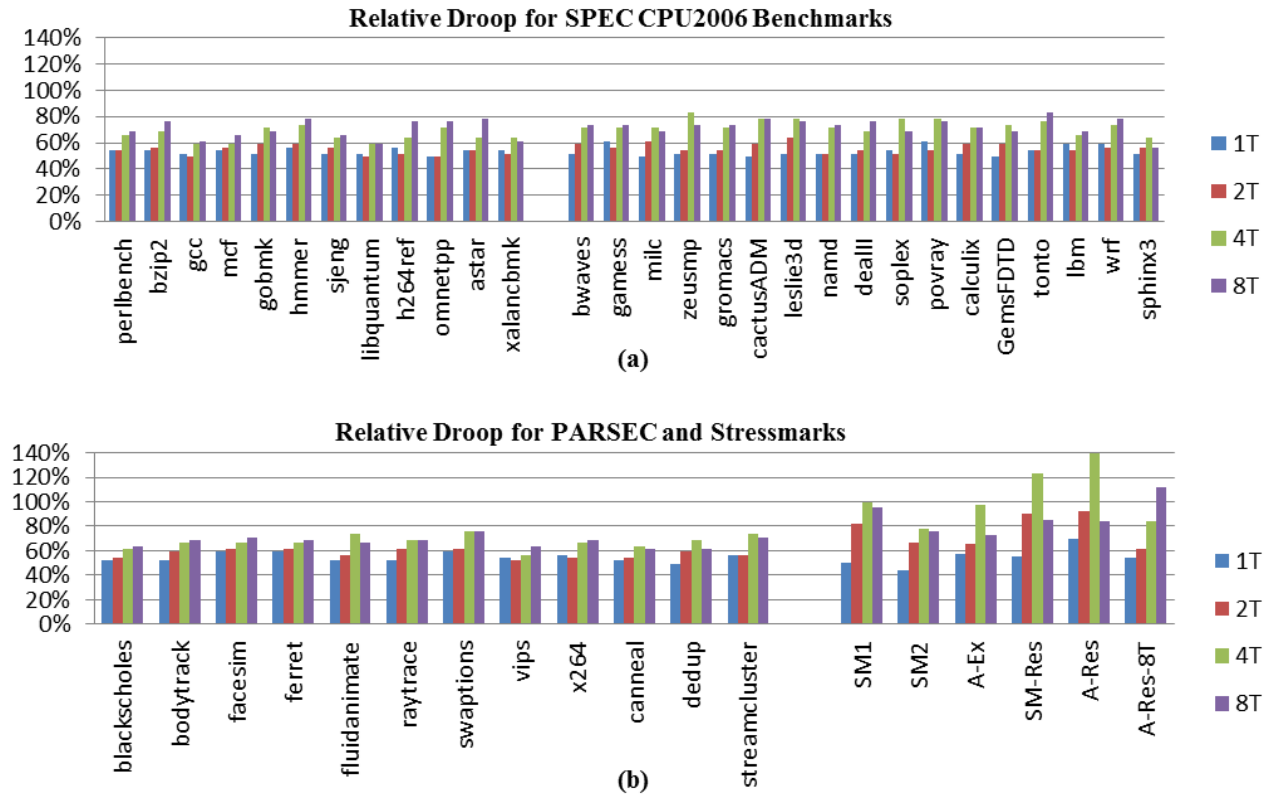


Figure 9. Hardware measurements of droop (relative to 4T *SM1*) for SPEC CPU2006, PARSEC, and stressmarks.

components (integer and retire logic, load/store unit, first-level TLB, and first-level cache) are separate. Each thread can issue four integer instructions per cycle, however, the two threads together can only issue four floating point instructions per cycle due to the sharing of the floating point units. A thread can have a maximum IPC of four. A more detailed description of the Bulldozer module and architectural features is given in [2]. In later experiments, we also replace the Bulldozer-based processor with an older-generation 45-nm AMD Phenom™ II X4 Model 925 processor to showcase AUDIT's ability to adapt to different systems and requirements.

AUDIT's code generation methodology is able to utilize all x86 instruction types, including integer, floating-point, and SIMD. General-purpose registers and 64-bit and 128-bit media registers are used for source and destination operands. Assembly code instructions are generated in NASM format and are compiled with NASM 2.09.08 [24].

Our experimental set-up is shown in Fig. 8. We measure voltage droops on hardware with a Tektronix TDS5104B oscilloscope and a 1.7-GHz Tektronix P6248 differential probe for triggering on large voltage droops. The probing points for the power supply voltage are attached to the package and on-die connection to enable accurate voltage droop measurements. The oscilloscope triggers and records the di/dt events at a sampling rate of 5 gigasamples/second (GS/s). We used Windows® 7 OS for SPEC CPU2006 benchmarks and stressmarks, and Red Hat Enterprise 6 for PARSEC [1] benchmarks.

V. Results and Analysis

This section covers the results obtained for multi-core stressmark generation. We compare and analyze standard benchmarks, existing stressmarks, and AUDIT-generated stressmarks. For each benchmark (stressmark), we present its maximum voltage droop and analyze the processor's ability to operate under degraded voltage conditions. We also present results showing AUDIT's ability to adapt to microarchitectural and architectural changes.

A. Droop and failure analysis

Fig. 9 shows the maximum droop measured from running SPEC CPU2006 benchmarks, PARSEC multi-threaded benchmarks, and a set of existing and AUDIT-generated stressmarks in configurations of one-, two-, four-, and eight-thread runs (1T, 2T, 4T, and 8T). For SPEC CPU2006 and stressmark runs with multiple threads, the program is replicated and executed on multiple cores, similar to SPECrate. Given the shared nature of the cores in the Bulldozer module, higher voltage droops occur for a given number of threads when threads are spatially distributed across modules. The evaluation processor has four Bulldozer modules, each with two cores. Hence, for the 1T, 2T, and 4T runs, each thread is assigned to a different module. For the

8T runs, there are two threads assigned to each module. All droop results are shown relative to the 4T *SM1* stressmark, and higher numbers indicate larger droops. The values are measured with the load line of the voltage regulator module (VRM) disabled to remove any load-line droop effects [15]. Hence, the results show the droop due to di/dt stresses only.

The multi-threaded AUDIT stressmarks (*A-Ex* and *A-Res*) and the hand-generated resonant stressmark *SM-Res* use the dithering methodology described in Section 3.B to align the threads for a worst-case voltage droop. The 2T and 4T configurations use the exact algorithm, and the 8T configuration uses the approximate algorithm with a δ of 3.

Unfortunately, the dithering methodology is not easily applicable to SPEC CPU2006 benchmarks or the PARSEC suite because they do not consist of a regular, repeatable loop that can be shifted to produce alignment between the threads. Although the lack of dithering for SPEC CPU2006 results in a smaller droop than is theoretically possible with ideal alignment [23], it also reflects the reality of multi-processor execution in which the natural misalignment between threads may counteract some worst-case stress generating behavior. Prior work shows that multithreaded programs such as those in the PARSEC suite have synchronization points that could align the threads and produce opportunities for high di/dt stresses [16].

1) Standard benchmarks

Fig. 9 shows that, in general, the magnitudes of the voltage droops increase with the number of threads for 1T, 2T, and 4T configurations. The 8T configurations do not always follow this trend due to multi-threading in the Bulldozer module (explained further in Section 5.A.2).

As noted in Section 2, one way to generate a significant droop is to have a large change in activity from idle to full execution. For high-performance pipelines, such a change in activity occurs naturally with certain pipeline events, such as pipeline recovery after a branch misprediction stall or high execution activity after a load miss resolves [22]. These events are commonplace in complicated pipelines, and how they interact with each other in a multi-threaded scenario dictates how large a droop they produce. Destructive interference may occur between threads in a multi-core system such that when one thread is in a high-power state others are in a low-power state. Reddi et al. describe the issue of thread misalignment for SPEC CPU2006 benchmarks, examine constructive and destructive interference in a dual-core system, and discuss co-scheduling threads to reduce voltage droops [23].

The PARSEC multi-threaded benchmark suite could have alignment between threads through its use of synchronization primitives. The expectation was that we would see higher droops due to the natural alignment resulting from barrier operations in benchmarks such as *fluidanimate* and *streamcluster* as discussed in [16]. However, our results show no significant difference in droops between PARSEC and the SPEC CPU2006 suite.

To evaluate further, we designed a barrier stressmark that repeatedly synchronizes on a barrier operation and then runs the high-power virus in a 4T configuration. We expected this to result in a large voltage droop due to all cores being aligned and idle at the barrier operation followed by high activity on the cores. The resulting droop, however, was not significant. On further examination, we noticed that a natural misalignment occurs between the cores when released from a barrier. On the Bulldozer module, there is no explicit mechanism to synchronize the barrier release signal, and the signal naturally reaches each core at different times based on from where in the memory hierarchy the core receives its data. This perturbs the start of activity across the cores by enough cycles to dampen the first droop excitation resulting from the synchronization operation.

The authors in [16] examined a different x86 processor that may have different characteristics. In addition, they use fluctuations in average power estimated at intervals of 1 ms on hardware as a proxy for expected di/dt variations. This may capture third droop excitations, but not first droop excitations that occur over the course of nanoseconds. Our measurement technique is capable of identifying the high-frequency first droop variations in voltage. Hence, the worst-case droops in PARSEC are most likely the result of the same microarchitectural events that align across multiple threads in the SPEC CPU2006 suite. The authors in [16] also note that barriers are not the only cause of high power swings and point to microarchitectural events such as long-latency cache misses followed by bursts of activity as other potential inducers of high droop. Furthermore, the effect in [16] is pronounced for cases with 32 threads, whereas our experiments did not include such configurations.

2) Stressmarks

Fig. 9(b) also shows the results for various stressmarks, either manually collected or hand-generated (*SM1*, *SM2*, and *SM-Res*) or automatically generated by AUDIT (*A-Ex* and *A-Res*). *A-Ex* is a first-droop excitation stressmark, and *A-Res* is a first-droop resonant stressmark. *SM1* and *SM2* contain both single-droop and resonant excitations, and *SM-Res* is a hand-generated resonant stressmark. The manual stressmarks are the result either of past di/dt issues or a non-trivial design effort (on the order of a week per stressmark) from a highly skilled engineer with detailed knowledge of the pipeline architecture. The goal of AUDIT is to generate similar or better stressmarks without detailed knowledge of the pipeline in question.

To produce the *A-Ex* and *A-Res* stressmarks in Fig. 9(b), we instructed AUDIT to generate a homogeneous stressmark with four identical threads, one assigned to each module. For the resonant stressmark, we use a high-power sub-block of length six cycles and repeat as many times as necessary to produce the high-power region. The low-power region of the stressmark, for the reasons noted in Section 3.C, consists of NOPs. The stressmark generation takes less than five hours to complete without human intervention.

With the exception of *SM2*, all stressmarks produce significantly greater droops than the standard benchmarks. As will be shown in Section 5.A.4, *SM2* is still a viable stressmark because it exercises the sensitive paths on the processor. The two resonant stressmarks (*SM-Res* and *A-Res*) produce significantly larger droops than all other stressmarks

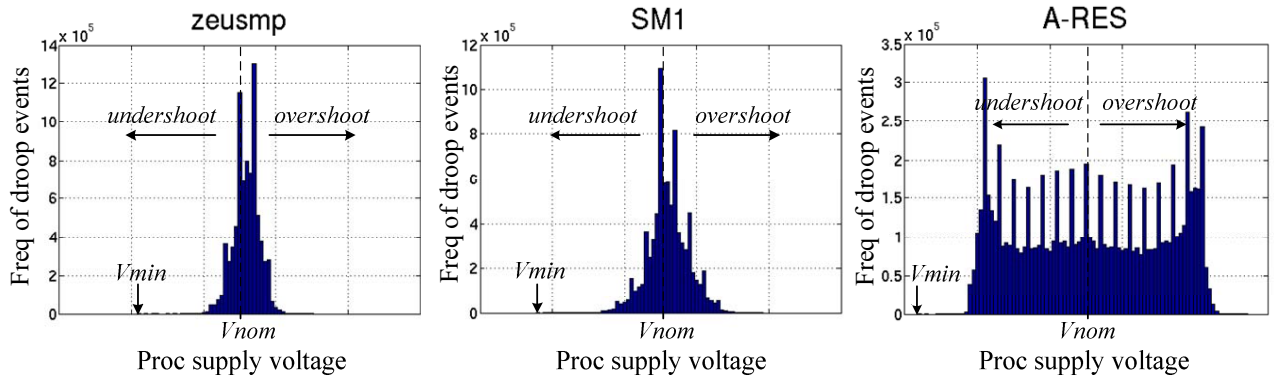


Figure 10. Frequency of droop events.

for the reasons described in Section 2. Both AUDIT-generated stressmarks (*A-Ex* and *A-Res*) produce droops that are either comparable or greater than that of the existing stressmarks. This highlights AUDIT's ability to produce results that are comparable to well-engineered stressmarks that require significantly more effort and knowledge to generate.

The stressmarks produce larger droops for the 4T case than for the 8T case. All stressmarks contain some amount of floating-point instructions, and the FPU is shared between the two threads in each Bulldozer module in the 8T runs. This results in interference between the threads; this shifts the loop lengths, making it difficult to align the first droop excitation across the threads or to oscillate at the resonant frequency. The same interference may not exist in the standard benchmarks depending on the density of floating-point operations and how the threads align.

The *A-Ex* and *A-Res* stressmarks are generated using four homogeneous threads assigned one to each module. Hence, the GA in AUDIT is not trained to deal with the shared FPU in the 8T run. To test our hypothesis, we ran AUDIT again to use eight homogeneous threads, with two threads per module, to generate a new stressmark (*A-Res-8T*). The resulting data is shown in Fig. 9(b). The 8T results for *A-Res-8T* are significantly better than the 8T results for *A-Res* or *SM-Res*. However, the 1T, 2T, and 4T results suffer for the same reason that the 8T run suffers for the other stressmarks -- because the characteristics assumed for the stressmark generation are not valid in some of the multi-threaded configurations. These results show that (1) system characteristics (such as shared resources) must be considered when generating stressmarks, (2) one type of stressmark may not apply to all configurations in a multi-core system, and (3) AUDIT is robust and flexible enough to find patterns that can exercise the characteristics of the system being evaluated with minimal manual intervention.

3) Droop probability

Fig. 9 shows the worst-case droop for the benchmarks and stressmarks. However, it does not show how often the droop occurs. The more frequently a large voltage droop occurs, the more likely it is to result in a catastrophic failure. Not only does first droop resonance produce larger droops than first droop excitation (see Fig. 4), it also produces more such events.

In Fig. 10, we use our hardware measurement tools to produce a histogram of voltage droops for *zeusmp*, *SM1*, and *A-Res*. Each plot contains 8 million samples. The x axis shows the measured V_{dd} and the x axis range is the same for all figures. The y axis shows the number of samples for the given V_{dd}. Values to the left (right) of center indicate voltage droops (overshoots).

The *zeusmp* benchmark has the least variation in voltage, as expected from the results in Fig. 9. *SM1* has a larger range of measured V_{dd}, yet the largest number of samples is centered at the nominal V_{dd} with a sharp reduction for lower voltages. There are spikes along the way, most likely due to code regions with resonant behavior, but the application has a long tail for both droops and overshoots. The resonance stressmark has the opposite characteristic with the highest number of events occurring near the worst-case droop values. Both stressmarks have a tail of low-probability droop events, but what dictates the failure point of these benchmarks is the higher-probability droop events near the tail. With hardware measurement, we can evaluate these characteristics across the entire run of the program, which is not possible in simulation. In the next section, we evaluate how these droop characteristics translate into failure points for each application.

4) Droop vs. voltage at failure

The size of the maximum voltage droop is an indirect indicator of the voltage operating margin of the program. The ultimate test is to determine the point at which failure occurs for each configuration. In the next experiment, running 4T configurations for two standard benchmarks with the largest droop (*swaptions* and *zeusmp*) and the stressmarks, we reduce the operating voltage in decrements of 12.5 mV until failure occurs. The higher the voltage at failure, the better the program is at stressing the system.

Table 1 shows the results relative to *A-Res*, which fails at the highest voltage (V_F). The other resonant stressmark, *SM-Res*, fails at a value 12 mV lower. The next to fail are the

other stressmarks, with *zeusmp* and *swaptions* failing last as V_{dd} is reduced.

As discussed earlier, the largest droops in the standard benchmarks are the result of a first droop excitation that tapers off quickly, as shown in the left side of Fig. 4. Hence, they may or may not cause system failure depending on whether the droop occurs when critical paths are being exercised. *A-Ex* also generates a first droop excitation, but it is large enough to cause a failure at higher voltages. *SM1* and *SM2* have both first droop excitation and first droop resonance, and they fail at a higher voltage than the standard benchmarks. This is expected for *SM1* due to its large droop. *SM2*, however, has a droop that is comparable to the standard benchmarks yet is more sensitive to the voltage levels. This is because *SM2*, unlike the benchmarks, is designed to exercise sensitive paths in the architecture.

What these results show is that the voltage droop is one indicator of potential failure, but not the only one. This insight would be difficult to gather from a cycle-accurate simulator that does not detect droop-induced system failures, and benchmarks such as *SM2* would be discarded as potential stressmarks.

As currently implemented, AUDIT's cost function for selecting successful populations is based on the measured droop in the system. However, it is trivial to adjust the cost function to reward the use of certain types of instructions that exercise critical paths if they are known. The key is that AUDIT is agile enough to manage these changes with little effort.

TABLE I. VOLTAGE AT FAILURE RELATIVE TO A-RES 4T FAILURE POINT.

A-Res	SM-Res	SM1	A-Ex	SM2	zeusmp	swaptions
$\mathbf{V_F}$	V_F – 12 mV	V_F – 62 mV	V_F – 75 mV	V_F – 87 mV	V_F – 125 mV	V_F – 125 mV

5) AUDIT loop analysis

To determine how AUDIT is able to produce large droops, we analyzed the main loop of the resonant stressmarks *SM-Res* and *A-Res*. *SM-Res* is hand-designed and regular in using floating-point and SIMD instructions during the high-power phase of the loop. *A-Res* uses a combination of integer and floating-point operations and high- and low-power instructions, including some NOPs in the high-power phase.

By mixing integer and floating-point operations, it is able to exercise multiple schedulers and execution clusters in the pipeline. What is more difficult to assess is why sprinkling NOPs in the code increases the droop.

To further understand the effect of the NOPs, we replaced the NOPs in the high-power region with independent, integer ADD operations and measured the resulting droop. If the pipeline flow remains the same, the ADDs should produce a higher droop than the NOPs since they are a higher power operation than NOPs. The modified *A-Res* stressmark generated a smaller droop (by 40 mV) than the original stressmark. In addition, the frequency of the di/dt pattern shifted lower than the ideal resonant frequency, indicating that the duration of the loop increased due to the inclusion of the ADD operations. Unlike ADDs, NOPs consume fetch and decode resources but do not affect other structures in the pipeline such as the schedulers, physical registers, or result busses. The use of the NOPs enabled the stressmark to attain resonance. Although the pipeline and the modified *A-Res* stressmark are constructed to attain a throughput of four instructions per cycle, resource hazards such as physical register availability, decode width capabilities, token-based scheduling restrictions, and result bus utilization impact the final outcome. AUDIT with its GA-based algorithm was able to construct a stressmark that worked around the pipeline hazards to produce a large droop. AUDIT's ability to accommodate pipeline restrictions is examined further in Section 5.B.

One valid concern is that AUDIT stressmarks are unrealistic because the droops generated by them are much worse than normal benchmarks or other stressmarks. As noted earlier, instead of using the stressmarks to set voltage margins, they can be used to understand the bounds of the problem and sensitivities of the pipeline. For example, the *A-Res* stressmark shows that it is possible to generate large droops by selecting both the floating-point and integer execution clusters in the pipeline rather than just focusing on the floating-point pipeline. Additionally, as will be shown in Section 5.B, when one di/dt stress path is blocked through droop mitigation mechanisms, AUDIT can find other high-stress paths in the pipeline. As noted by Patel [20], there are many sensitive paths on cores that can lead to catastrophic failures when the system is stressed by reduced noise margins, and it is imperative that we have the tools necessary to identify these paths.

6) Observations and summary

There are some key observations from the results and discussion so far:

- Benchmarks do not produce the same levels of first-order droop as the stressmarks. On our processor, this is true even for benchmarks that have global synchronization resulting from barriers.
- There are many different ways to construct a stressmark in a multi-core system depending on what structures and types of configurations one is trying to exercise. Therefore, a stressmark that works well for one configuration (such as *A-Res* for 4T runs) may not produce the best results for other configurations. AUDIT's flexibility and ease of use can be leveraged to develop a suite of stressmarks that can effectively exercise all significant usage scenarios in the system.
- The measured droop is not the only indicator of sensitivity to failure. The paths exercised by the stressmark and the number of times the droop event occurs also have an impact on overall program susceptibility.

- AUDIT is able to match or exceed the droops produced by benchmarks and other stressmarks by exercising a richer set of paths in the pipeline.

The rest of the results section presents ways in which AUDIT can accommodate microarchitectural and architectural changes.

B. Impact of FPU throttling

Floating-point and SIMD instructions are generally the highest-power instructions available in the execution pipeline and they are used extensively in the high-power portion of the stressmark. A number of papers have noted that hardware and software architectural throttling schemes reduce di/dt stresses by limiting the rate of change in the execution of high-power instructions [3][5][7][9][10][18][21][22][23]. We utilize a FPU throttling scheme that statically limits the maximum number of FPU instructions executed in a cycle.

We measured the droop on some of the stressmarks with FPU throttling enabled to determine the maximum droop and maximum voltage at failure. The results are shown in Table 2. As before, all droop data are relative to the 4T *SM1* stressmark with FPU throttling disabled. FPU throttling is highly effective for *A-Res* and *SM-Res*, but less so for *SM1*. *SM1* is composed of multiple high-stress code sequences, and FPU throttling does not affect all stress paths in *SM1*. Although the results vary, the droop and voltage at failure improve with FPU throttling. These results show that FPU throttling functions as expected by limiting di/dt stresses; however, the results so far do not show whether AUDIT can find another stressmark that can produce a significant droop with FPU throttling enabled.

TABLE II. IMPACT OF FPU THROTTLING ON RELATIVE DROOP (RELATIVE TO 4T SM1) AND FAILURE POINT (RELATIVE TO 4T A-RES).

	Stressmark	Rel. Droop	Failure Point
No Throttling	SM1	1	V_F – 62 mV
	A-Res	1.39	V_F
	SM-Res	1.25	V_F – 12 mV
FPU Throttling	SM1	0.93	V_F – 75 mV
	A-Res	0.86	V_F – 100 mV
	SM-Res	0.78	V_F – 113 mV
	A-Res-Th	0.98	V_F – 75 mV

We used AUDIT to generate a new stressmark (*A-Res-Th*) to determine if there are other opportunities to generate a large droop in conjunction with FPU throttling. We repeated the AUDIT stressmark generation using four threads, but with FPU throttling enabled. Table 2 shows the droop and failure levels for the new stressmark *A-Res-Th*. AUDIT was able to generate a stressmark that works around the FPU throttling restrictions to increase the size of the droop. However, it is not able to match the droops seen without FPU throttling because it is now limited to using fewer high-power floating-point and SIMD operations. With FPU throttling enabled, *A-Res-Th* exceeds the *SM1* stressmark for droop and matches it for sensitivity to voltage. It also highlights another stress path through the processor for engineers to evaluate.

The results show the experimental FPU throttling scheme works well for reducing voltage droops in the system, and AUDIT, in a relatively short time (~5 hours) has identified another path that can still produce significant voltage droops with FPU throttling enabled.

C. AUDIT on a different processor

To present AUDIT's ability to adjust to microarchitecture and system changes, we replaced the Bulldozer-based processor in the experimental system with an older-generation 45-nm AMD Phenom II X4 Model 925. The rest of the board remained unchanged. Each core in the AMD Phenom processor has local L1 and L2 caches, no multi-threading, and less variation between high- and low-power regions because it does not manage power as aggressively as the Bulldozer-based system. We generated new resonant stressmarks for the AMD Phenom processor using AUDIT and the results are shown in Table 3. We were unable to run *SM1* on the older processor due to incompatible instructions. As with the Bulldozer-based system, AUDIT was able to generate stressmarks that were comparable or better than hand-tuned stressmarks, highlighting the capabilities of automatic stressmark generation tools such as AUDIT.

TABLE III. DROOP AND FAILURE RESULTS FOR A 45-NM AMD PHENOM II PROCESSOR. DROOP AND FAILURE POINT ARE SHOWN RELATIVE TO SM2.

	zeusmp	SM2	A-Res
Rel. Droop	0.82	1	1.10
Failure Point	V_F – 50 mV	V_F	V_F

VI. PREVIOUS WORK

There has been some previous work on hardware analysis of production systems. In [23], Reddi et al. measured and analyzed droops on a two-core Intel system and discussed constructive and destructive interference between processors and the difference in droops between average and worst-case scenarios. This information was used to design a noise-aware thread scheduler to mitigate some of the di/dt stresses in the system. To date, the work by Reddi is the most detailed hardware analysis of droops.

We expand on that work by analyzing a more complex system with multi-threading and up to eight logical processors. In addition, we show that constructive interference occurs more often than expected due to OS effects, and use this knowledge to design effective stressmarks.

More recently, Miller et al. examined voltage emergencies in multi-core processors [16] with increasing numbers of cores, and showed how global synchronization points create large stresses in the system. This work used power variability as a proxy for di/dt stresses and examined the hardware at a coarse granularity of 1-ms intervals. In our work, we use true voltage droop measurements and fine-grained sampling to detect first-order droops and discuss droop values as well as voltage failure points in hardware.

The second major contribution of our work is automatic stressmark generation using real hardware. Joseph, Brooks,

and Martonosi presented a hand-coded di/dt stressmark [10]. Their basic idea was to create a sequence in which a high-current instruction follows a low-current instruction. The high-current component typically consisted of a memory load/store instruction and the low-current component consisted of a divide instruction followed by a dependent instruction, resulting in a long pipeline stall. However, their di/dt stressmark was manually crafted for a specific microarchitecture based on the knowledge of the current draw of various instructions. Furthermore, they focused only on memory-intensive behavior such as loads and stores and increased current draw by accessing L1 and L2 data caches. In contrast, our approach does not require microarchitectural knowledge and relies on measured voltage droops in a closed-loop measurement infrastructure.

Ketkar and Chiprout proposed a di/dt stressmark generation methodology using integer linear programming (ILP) [12]. They extracted current draw for certain instructions from a register transfer language (RTL) model for the hardware. Linear programming with constraints was used to maximize voltage droop. However, they focused only on the ALU. It is difficult to make ILP relationships of instructions for all the pipeline stages and the caches; hence, it is difficult to apply their technique to an entire processor, especially one with out-of-order processing, multiple cores, and complex shared resource structures.

Joshi et al. [11] presented a methodology for generating maximum-power viruses and mentioned in passing that high-power and low-power instruction sequences from two different power optimizations can be interleaved to generate a di/dt stressmark. This was only a suggestion, without implementation details or results. Also, they did not talk about the importance of repeating the sequence at the PDN's resonant frequency. Neither di/dt effects nor voltage droops were the focus of Joshi's work.

Kim and John [13] presented a methodology for automatic generation of a di/dt stressmark for a single-core processor. However, no methodology was presented for multi-core or multi-threaded processors. Furthermore, that was a simulation-based study using SimpleScalar with the Alpha instruction set; they did not use hardware to generate or validate the stressmarks. As discussed in Section 3, we have expanded on their work by introducing dithering and sub-blocking to produce better stressmarks, analyzing the results on hardware, comparing AUDIT's performance to existing stressmarks, and showing AUDIT's flexibility in handling microarchitectural and architectural constraints.

A significant number of other studies have focused on preventing, reducing, or recovering from di/dt effects or voltage droops [3][5][6][7][8][9][10][15][17][18][21][22][23]. However, none of these focus on automatically generating stressmarks for di/dt.

VII. Conclusion

This paper addressed the issue of di/dt stress generation and analysis in state-of-the-art multi-core x86 processors. We discussed the complexities involved with designing a stressmark, especially for multi-core systems. We presented the dithering algorithm to aid in multi-core stressmark generation and evaluated an automatic stressmark generation tool, AUDIT, that uses sub-blocking and dithering to produce stressmarks using hardware. We showed how AUDIT stressmarks compare to existing stressmarks and standard benchmarks, and presented results showing AUDIT's ability to adjust to different processor characteristics such as shared resources, FPU throttling, and different processors.

Acknowledgment

Lizy John and Youngtaek Kim are partially supported by NSF grant 1117895 and by AMD unrestricted research funding. Views and opinions expressed in this paper are that of the authors and not that of the National Science Foundation. This work was conducted when Youngtaek Kim was an intern at AMD Research. The authors thank Tom Snodgrass for helping set-up and run all the laboratory equipment, and thank other AMDers for giving great help and invaluable comments during this research.

References

[1] C. Bienia, Benchmarking Modern Multiprocessors, Ph.D. Thesis. Princeton University, 2011.

[2] M. Butler, L. Barnes, D. Sarma, and B. Gelinas, "Bulldozer: an approach to multithreaded compute performance," *IEEE Micro*, vol. 31, no. 2, pp. 6-15, 2011.

[3] W. El-Essawy and D. Albonesi, "Mitigating inductive noise in SMT processors," in *Proceedings of International Symposium on Low Power Electronics and Design (ISLPED)*, 2004.

[4] T. Fischer et al., "Design solutions for the Bulldozer 32nm SOI 2-core processor module in an 8-core CPU," in *Proceedings of IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, 2011.

[5] E. Grochowski, D. Ayers, and V. Tiwari, "Microarchitectural simulation and control of di/dt-induced power supply voltage variation," in *Proceedings of International Symposium on High-Performance Computer Architecture (HPCA-8)*, 2002.

[6] M. Gupta, J. Oatley, R. Joseph, G. Wei, and D. Brooks, "Understanding voltage variations in chip multiprocessors using a distributed power-delivery network," in *Proceedings of Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2007.

[7] M. Gupta, K. Rangan, M. Smith, G. Wei, and D. Brooks, "DeCoR: a delayed commit and rollback mechanism for handling inductive noise in processors," in *Proceedings of International Symposium on High-Performance Computer Architecture (HPCA-14)*, 2008.

[8] M. Gupta, K. Rangan, M. Smith, G. Wei, and D. Brooks, "Towards a software approach to mitigate voltage emergencies," in *Proceedings of International Symposium on Low Power Electronics and Design (ISLPED)*, 2007.

[9] K. Hazelwood and D. Brooks, "Eliminating voltage emergencies via microarchitectural voltage control feedback and dynamic optimization," in *Proceedings of International Symposium on Low Power Electronics and Design (ISLPED)*, 2004.

[10] R. Joseph, D. Brooks, and M. Martonosi, "Control techniques to eliminate voltage emergencies in high performance processors," in *Proceedings of International Symposium on High-Performance Computer Architecture (HPCA-9)*, 2003.

[11] A. Joshi, L. Eeckhout, L. John, and C. Isen, "Automated microprocessor stressmark generation," in *Proceedings of International Symposium on High-Performance Computer Architecture (HPCA-14)*, 2008.

[12] M. Ketkar and E. Chiprout, "A microarchitecture-based framework for pre- and post-silicon power delivery analysis," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-42)*, 2009.

[13] Y. Kim and L. John, "Automated di/dt stressmark generation for microprocessor power delivery networks," in *Proceedings of International Symposium on Low Power Electronics and Design (ISLPED)*, 2011.

[14] N. Kurd, J. Douglas, P. Mosalikanti, and R. Kumar, "Next generation Intel® Core micro-architecture (Nehalem) clocking architecture," *IEEE Journal of Solid-state Circuits*, vol. 44, iss. 4, pp. 1121-1129, 2009.

[15] C. Lefurgy, A. Drake, M. Floyd, M. Allen-Ware, B. Brock, J. Tiemo, and J. Carter, "Active management of timing guardband to save energy in POWER7," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-44)*, 2011.

[16] T. Miller, R. Thomas, X. Pan, and R. Teodorescu, "VRSync: characterizing and eliminating synchronization-induced voltage emergencies in many-core processors," in *Proceedings of the 39th International Symposium on Computer Architecture (ISCA-39)*, 2012.

[17] F. Mohamood, M. Healy, S. Lim and H. Lee, "A floorplan-aware dynamic inductive noise controller for reliable 2D and 3D microprocessors," in *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-39)*, 2006.

[18] M. Pant, P. Pant, D. Willis, and V. Tiwari, "Architectural solution for the inductive noise problem due to clock-gating," in *Proceedings of International Symposium on Low Power Electronics and Design (ISLPED)*, 1999.

[19] S. Pant and E. Chiprout, "Power grid physics and implications for CAD," in *Proceedings of the 43rd Annual Design Automation Conference (DAC-43)*, 2006.

[20] J. Patel, "CMOS process variations: a critical operation point hypothesis," *Online Presentation*, 2008.

[21] M. Powell and T. Vijaykumar, "Exploiting resonant behavior to reduce inductive noise," in *Proceedings of the 31st International Symposium on Computer Architecture (ISCA-31)*, 2004.

[22] V. Reddi, M. Gupta, G. Halloway, G. Wei, M. Smith, and D. Brooks, "Voltage emergency prediction: using signatures to reduce operating margins," in *Proceedings of International Symposium on High-Performance Computer Architecture (HPCA-15)*, 2009.

[23] V. Reddi, S. Kanev, W. Kim, S. Campanoni, M. Smith, G. Wei, and D. Brooks. "Voltage noise in production processors," *IEEE Micro,* vol. 31, no. 1, pp. 20-28, 2011.

[24] The NASM Development Team, "NASM - The Netwide Assembler," Available: http://www.nasm.us.

[25] D. Weiss et al., "An 8MB Level-3 cache in 32nm SOI with column-select aliasing," in *Proceedings of IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, 2011.

2012 IEEE/ACM 45th Annual International Symposium on Microarchitecture

Accurate Fine-Grained Processor Power Proxies

Wei Huang[†] Charles Lefurgy[*] William Kuk[**,‡] Alper Buyuktosunoglu[*]
Michael Floyd[**] Karthick Rajamani[*] Malcolm Allen-Ware[*] Bishop Brock[**]

[†]AMD, [*]IBM Research, [‡]Purdue University, [**]IBM System and Technology Group
WeiN.Huang@amd.com, {lefurgy,wwkuk,alperb,mfloyd,karthick,mware,bcbrock}@us.ibm.com [*]

Abstract

There are not yet practical and accurate ways to directly measure core power in a microprocessor. This limits the granularity of measurement and control for computer power management. We overcome this limitation by presenting an accurate runtime per-core power proxy which closely estimates true core power. This enables new fine-grained microprocessor power management techniques at the core level. For example, cloud environments could manage and bill virtual machines for energy consumption associated with the core. The power model underlying our power proxy also enables energy-efficiency controllers to perform what-if analysis, instead of merely reacting to current conditions.

We develop and validate a methodology for accurate power proxy training at both chip and core levels. Our implementation of power proxies uses on-chip logic in a high-performance multi-core processor and associated platform firmware. The power proxies account for full voltage and frequency ranges, as well as chip-to-chip process variations. For fixed clock frequency operation, a mean unsigned error of 1.8% for fine-grained 32ms samples across all workloads was achieved. For an interval of an entire workload, we achieve an average error of -0.2%. Similar results were achieved for voltage-scaling scenarios, too. We also present two sample applications of the power proxy: (1) per-core power billing for cloud computing services; and (2) simultaneous runtime energy saving comparisons among different power management policies without running each policy separately.

1. Introduction

In the last several years, direct measurement of power consumption has been widely deployed in servers [20]. At a system-level, bulk power supply measurement has enabled energy-efficiency optimizations, power capping and shifting, and cost-of-operation analysis. Power measurement of the microprocessor is becoming common as well and allows for more fine-grained power management. For example, Intel Node Manager 2.0, to be introduced this year, will use direct processor and memory power measurements to implement power capping by shifting power allocations between processor and memory subsystems [7]. Core-level management, on the other hand, has languished because there are still no accurate and practical ways to directly measure the power consumption associated with each core.

A viable alternative is to implement core *power proxies*, which are estimates of true core power consumption. They are constructed from real-time measurements of microarchitecture counters and physical sensors. Many previously published power proxy implementations made use of existing processor performance monitoring and analysis signals which were originally put into the hardware to assist in tuning compilers and operating mode settings. While these events are related to the activity of the processor, they typically track performance-sensitive events rather than events that contribute most to power consumption. This can lead to significant error across a wide variety of workloads. More recent work, including ours, leverage activity signals in the microarchitecture that correlate better with power. However, the prior studies have serious limitations. First, they typically model power at fixed voltage and frequencies, which ignores how the power proxy error tracks with dynamic frequency and voltage scaling processors. Second, they focus on active power and give only simplistic treatments of leakage power and do not consider the impact of process variations. This reduces the effectiveness of the proxy since conventional high-performance microprocessors have considerable leakage power consumption and the power consumption of processors of the same type and model can vary widely due to manufacturing-based variation. Without a strategy for covering the entire power of the core, the power proxy cannot be used for applications requiring high accuracy.

The value of accurate power proxies depends on how they are used. For billing applications, a 1% inaccuracy in energy consumption directly translates to an additional 1% cost to either the user or supplier. For power capping applications, reduced accuracy in power estimations means that additional margins added to the requested capping value must be taken to ensure the real power limit is maintained. In our system, we measure that every 1% of Vdd power accuracy translates into 1.2% throughput on the SPECpower workload. For example, the value of power capping with a power proxy that is 1% accurate compared to a power proxy that is 5% accurate is about 4.8% in performance. For energy-efficiency controllers that maximize operations per Watt, the inaccuracy may not matter when optimizing a single component if the power estimation is monotonic with true power. However, when optimizing across many processors that each provide a power estimate, suboptimal decisions could be made if the estimates reverse the true sense of which component draws the most power. Therefore, we believe that continuing to improve accuracy by even small amounts is meaningful. Additionally, shortening the time for estimation enables fine-grain power management. A review of prior work shows our power proxies achieve accuracy comparable to the best known solutions, but do so at a 30x smaller time resolution (32 ms estimations vs. 1 second estimations). Additionally, we validate the power proxies work across full frequency and voltage ranges, and account for process variation.

In this paper, we propose a methodology of constructing highly accurate power proxies, first at the chip level, then at the core level. Our architecture accounts for active power by utilizing specialized activity counters in the chip hardware. Firmware computes the final power proxy by using the measured activity values and incorporating real-time physical sensor measurements. In addition it calculates

[*]This work was done while Wei Huang was a researcher with IBM Research, and William Kuk was an intern with IBM System and Technology Group.

1072-4451/12 $26.00 © 2012 IEEE
DOI 10.1109/MICRO.2012.29

IEEE computer society

leakage power using manufacturing-time characterization data.

We implement our power proxy architecture on a IBM POWER7+® high-performance system and run many workloads to evaluate its accuracy over full voltage and frequency operating ranges. The power proxies also take into account chip-to-chip variations. We additionally show that the power proxy is flexible enough to account for power even when voltage and frequency pairings are not fixed, but can vary for undervolting and overclocking.

Similar power proxy on-chip circuits in AMD and Intel chips exist. However, implementation details and thorough accuracy evaluations of them have not been published. This paper also significantly extends previous IBM POWER7 power proxy publications by adding accurate models for both clock and leakage power, and for the first time presents a complete full-chip and per-core power proxy development methodology with better accuracy than published work.

After demonstrating an accurate power proxy, we illustrate two use scenarios: billing and predictive management.

First, power has become a precious resource in the data center as it has a growing impact in the cost of server ownership. In response, the idea of billing users for energy use in addition to time-based or MIPS-based accounting is gaining traction. Per-core power proxies open the opportunity for energy-based billing in cloud computing services.

Second, a plethora of runtime energy and power management techniques have been invented and implemented to achieve energy proportionality for servers and data centers to boost energy efficiency and reduce operational cost. Typically a trial-and-error method is used to determine which management policy is the most effective for a given workload. It is impractical to run the same workloads multiple times, each time with a different power management technique or policy enabled. Having power proxies that are accurate across voltage and frequency ranges solves this problem. The power consumption model that underlies the proxy can estimate the instantaneous power consumption for each power management technique simultaneously at run time according to its decision on operating voltage, frequency, temperature, as well as activity counts of the running workload, thus providing a direct comparison across all power management policies. The energy manager can then dynamically and intelligently select a policy in response to changing workload characteristics.

We summarize our contributions as follows:

- We develop a methodology to construct core-level and chip-level power proxies, which are implemented in hardware and system firmware.
- The power proxies take into account full voltage and frequency ranges. It is also adjustable to chip-to-chip process variations.
- We present the first power proxy that works accurately even when chip voltage and frequency settings do not have fixed pairings. This is useful for systems that dynamically undervolt (with fixed frequency) or overclock (with fixed voltage).
- The power proxy values are updated every 32 ms to enable fine-grain energy management which is 30x faster than prior work with comparable accuracy. The power proxies are based on activity counters and sensors for frequency, voltage and temperature that are gathered out-of-band so as not to disturb the running workload.
- We illustrate how to achieve per-core power accounting, which may be incorporated into existing proposals for per-VM power accounting for cloud computing services.
- We discuss the usefulness of power proxies for evaluating different energy management techniques simultaneously at run time.

Because of hardware differences (e.g. on-chip activity counter architecture, data collection rates, voltage rail power measurement availability, frequency range, process variation distribution, etc.), it is nearly impossible for us to conduct a direct comparison with existing methodologies using hardware measurements. However, we do cite and try our best to quantitatively compare with the claimed accuracy from existing work. In addition, we only solve one (probably the most complicated) part of the full-system power model, namely the power consumption of the processor. The methodology needs to be extended with additional techniques to account for power consumption of other system components.

The paper is organized as follows. An overview of power proxies in POWER7+ and discussion of design considerations is covered in Section 2. Next, Section 3 shows how we develop chip power models and incorporate core-based activity measurement with chip-based characterization. We report experimental results to determine their accuracy. Section 4 illustrates use cases for accurate core power estimates. In Section 5 we review related work. Finally, we conclude with Section 6.

2. Background

2.1. Power modeling

The power consumed by a microprocessor can be described by Eqn. (1).

$$\begin{aligned} P_{chip} &= P_{active} + P_{idle} \\ &= P_{active} + P_{clock} + P_{leak} \end{aligned} \quad (1)$$

P_{idle} represents the *idle power* consumed when the processor is on, but not executing instructions and P_{active} is the additional *active power* consumed due to instruction execution. The idle power can be further separated into clock grid power and temperature-dependent leakage power.

Prior work on power proxies has focused mainly on accurate active power estimations and given simplistic treatment of the idle power - often assuming a constant value or one that varied linearly with clock frequency. While this may be valid for steady-state workloads, it is not sufficient for dynamic workloads that induce changing voltages (voltage scaling) and temperatures in the chip since idle power depends strongly on voltage and temperature.

We describe our power model in more detail in Section 3.

2.2. POWER7 chip proxy logic

The POWER7 chip power proxy circuits have been disclosed before in prior publications [4][5][20]. The circuitry in POWER7+ is identical. These circuits are used to estimate core active power and alone cannot accurately account for idle power. Our work complements the prior publications by demonstrating how firmware can be used to accurately account for idle power and for voltage-scaling in the active power component.

The methodology to estimate on-chip active power is to accumulate a weighted sum of activity counters each measurement interval. We use the term *activity proxy* to denote this aggregated activity count. Each chiplet (combination of a single core with its L2 and L3

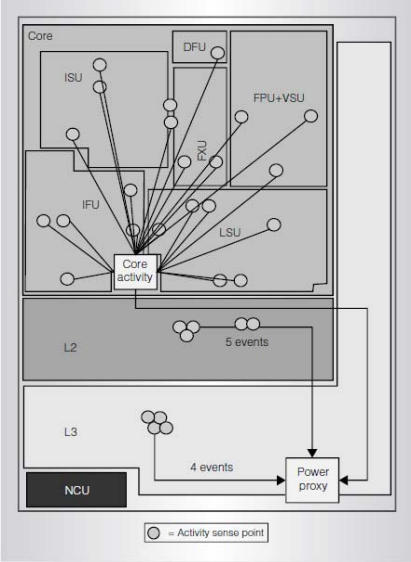


Figure 1: Activity counters in a POWER7 chiplet [4]. POWER7+ chiplet floorplan is slightly different.

caches) in POWER7+ contains activity proxy logic so that per-core active power can be separately accounted. Firmware then adjusts the core activity proxies for the effects of leakage, temperature, process variations and voltage to form the chip and core *power proxies*, which are the final estimations of true power consumption. A discussion of the firmware is in Section 3. The remainder of this section overviews the POWER7 chip power proxy circuit.

Figure 1, reproduced from Floyd et al. [4], shows a diagram of activity proxy event collection in the POWER7 processor chiplet. The activity signals were selected during the initial POWER7 microarchitecture design phase. The top events that caused the majority of the power consumption in each functional unit (e.g. Dispatch, Fixed point, Load-Store) were considered. Example events include: the deactivation of dynamic clock gating, data switching, and register file or array accesses. Relevant signals were added directly from those units to the on-chip proxy logic which adds a per-event programmable weight to an accumulation register whenever that event occurs. Care was taken to avoid redundant counting whenever possible. For example, counting Load-Store issue already covers data cache and D-ERAT (effective-to-real data address translation) reads and Load Miss Queues so both were not included. For some frequent events, such as General Purpose Register File access, a pre-count is performed on the activity before sending a summary signal out of the unit. Other events, such as instruction pipe issue or floating point operation type, are sent as encoded values and a multiplier is applied accordingly based on its anticipated power relative to the other types in that encode group. The L2 and L3 cache units also provide active power events based on cache lookup and types of access. All these activity count accumulators are then scaled and summed to form an aggregate activity proxy per core which can be converted into an estimate of active power consumed over the previous time period.

There are several challenges in such weighted counter-based proxy architecture, including:

- How to choose the minimal set of key activity counts to architect;
- Correctly sizing the counter and weight registers and the final scaling logic; and
- How to decompose the weighted aggregation step so as to minimize hardware complexity and calculation time.

To address these challenges, the methodology relies on systematic, linear-regression based formalism in conjunction with designer intuition and experience. A genetic algorithm (GA) optimization tool further refined the design under hardware constraints. Specifically, the reference (RTL-validated) performance simulator-driven power simulator projects the core power consumption across a carefully selected range of workloads. In each case, hundreds of activity count events were also collected over a pre-selected execution time window. The resulting matrix of several data elements (consisting of power values and activity counts) was fed into the GA-driven regression solver, to deduce the right set of activity counts as well as the size of weights to form the architecture. At the end, activity count events are carefully selected to capture those that correlate maximally with active power consumption as well as those that are the fundamental (pseudo-independent) positive correlators of power. The final design has an affordable number of hardware counters and weight bits which led to an accurate, yet flexible proxy architecture with reasonable cost. The final design measures close to 50 activity counts which include instructions dispatched, instructions completed, execution register file accesses, execution pipeline issue types, instruction fetch unit activity, load-store unit cache activity, load-store unit D-ERAT activity, load-store unit prefetch activity, L2 cache reads/writes and L3 cache reads/writes.

3. Power Proxies

In this section, we demonstrate our methodology and verify its accuracy by creating a proxy that replicates the physical power sensor for the chip Vdd power rail.

3.1. Experimental setup

We have implemented the power proxy in a prototype high-performance POWER7+ server. It has four microprocessors (P0-P3) with 6-cores each, 256 GB of main memory, and runs AIX 7.1. The maximum core clock frequency is 4228MHz. Voltage is controlled independently for each microprocessor with all cores on a chip sharing the same voltage level. A per-core digital phase-lock loop allows clock frequency to be set independently for each core. We purposefully selected the four microprocessors from different process corners and they are distinct in terms of leakage current, nominal supply voltage, and ring-oscillator delay measurements. This allows us to confirm that the power proxy accounts for manufacturing-based variation. We pick P1 as the reference chip because it has a more representative nominal voltage than other three chips. In the following text, for results that are related to a single chip, we show the results on P1.

The power proxies are implemented in two parts in a POWER7+ system. First, activity counters and the calculation of the activity proxies are implemented in hardware logic of the processor. The weights to different activity events are programmable by writing to special on-chip registers. Second, a service coprocessor receives measurements of activity proxies, chip supply voltage, core clock frequencies, and core temperatures from the POWER7+. The measurements are sent over a special out-of-band management interface that does not disrupt running workloads. The chip-level and core-level power proxies are calculated in firmware that runs on the service coprocessor. The firmware performs this computation every 32 milliseconds, which is constrained by the narrow bandwidth of the management interface.

3.1.1. Power delivery path The POWER7+ processor has four independent voltage rails, each fed by a voltage regulator module (VRM). Two of the voltage rails (Vio and Vmem) require fixed voltages, established at chip manufacturing, and are unique for each chip. The other two voltage rails (Vdd and Vcs) have voltages that can be dynamically changed to implement DVFS policies.

The power proxy hardware does not cover circuitry on the Vio and Vmem rails. This design choice was driven by the fact that the current drawn on these rails varies only a small amount and because the load is largely constant. During chip manufacturing, the current associated with Vio and Vmem is measured during a calibration workload and stored in the chip's Vital Product Data (VPD) non-volatile memory along with the associated voltage.

The Vdd and Vcs rails do have circuitry associated with the power proxy hardware. The Vdd rail feeds the cache and core logic and the Vcs rail feeds the L3 cache (embedded DRAM) on the chip. For both rails, chip manufacturing also measures current and voltages, but instead of the single point characterization done for Vio and Vmem, there are four unique points measured for each of the Vdd and Vcs rails. The four unique points cover four standard operating points across the full range of voltages and the associated frequencies that the chip can operate at safely, and these values are written into the chip's VPD. The VPD values are used by product firmware to program the Vdd and Vcs VRMs so that the output voltages from the VRM will be sufficient to establish the characterized chip voltage under worst-case conditions considering load line and other losses in the power delivery system.

The Vdd rail is the most interesting power rail in terms of dynamic power management, as it carries significantly more current than Vcs. The Vdd and Vcs voltages can be varied over a very wide range along with the frequency, with Vcs scaling proportionally to Vdd. At the low end of the voltage range, the minimum Vdd voltage or Vmin is typically only 70% of the maximum Vdd voltage or Vmax used. To go with this, the frequency range varies from a minimum frequency that is only 54.5% of the maximum frequency used.

Roughly 95% of the activity that the activity proxy measures is associated with the Vdd rail. For this reason the paper focuses exclusively around precise characterization of the Vdd rail. The methodology we present in this paper can be easily extended to the Vcs rail, and we leave this as future work.

3.1.2. Sensing In our test system, the power on the input side (12V rail) of the chip Vdd voltage regulator is directly measured with an accuracy of 2%. Therefore, our measurement includes loss due to voltage regulator conversion inefficiency. The firmware reads analog-to-digital converters to measure the average power during each 32 millisecond interval. We use this as the ground truth in all experiments and accuracy claims. The goal of our power proxy is to replicate as closely as possible the measured chip Vdd power for each 32 ms interval. We measure the per-core temperature by averaging the 5 digital thermal sensors located in each core. They provide temperature in units of 1 degree Celsius and are accurate to within 4 degrees of the true temperature.

3.2. Kernels and Benchmarks

We use two sets of workloads for building our runtime power proxies. First, kernels from a variety of available system characterization sources are used for training the activity proxy weights and power model coefficients. Second, testing and validation are done with a separate set of benchmarks.

The majority of our training workloads are kernels constructed from simple array-based loops such as in the popular `lmbench` [14] and STREAM [13] benchmarks. A desired size array is allocated and kernel-specific operations are performed in a sequential or random order over the array elements. By varying the nature of operations, number of distinct arrays and sizes of arrays a variety of workload instruction and storage access patterns are emulated. This gives us a reasonably rich set of power exercisers for the cores, on-chip caches and logic for accessing the different memory layers including off-chip DRAM. We additionally vary the workloads by selecting the multithreading mode of the chip. In all, we use 762 unique workloads for training. This simple kernel-based characterization also helps us set up very steady workloads in terms of activity and power to enable truly representative power measurements to be taken for the training phases. We complement these loops with a set of system stress-test workloads which are targeted at specific components of the system. This set also includes a maximum power workload developed for the POWER7 processor.

For testing and validation we use two sets of popular benchmarks from SPEC—SPECPower_ssj2008 [18] and SPEC CPU2006 [17]. The former provides workloads of different intensities helping us evaluate our models across the full range of system loads and the latter provides a rich variety of processor and memory hierarchy usage examples.

3.3. Activity proxy training

We use a procedure similar to [4] to train weights in the activity proxy calculation. All cores in all chips use the same learned weights. First, we measure the power consumption and temperature of the chip when it is idle. When running the training set of benchmarks, we set the processor (P1) to its nominal frequency and supply voltages. All cores run the same workload. We then sample power and temperature measurements as well as activity counts for each event. Per-core active power is calculated by subtracting the estimated idle power, which includes a temperature-based adjustment (Section 3.4), from total measured power and dividing by the number of cores. For each activity event, we also average across all the cores to reach a per-core count for that event. The IBM SNAP genetic algorithm optimization tool takes per-core active power and activity counts as inputs to derive a linear regression model for active power, in the form of

$$ActivityProxy = \Sigma(W_g \times \Sigma(W_{i_g} \times A_{i_g})) \quad (2)$$

where W_g is an activity group weight, W_{i_g} is the weight for activity event i in group g, and A_{i_g} is the count for event i in group g. The POWER7+ hardware overhead is minimized by splitting the activity weights into an event weight and a group weight. We use a genetic algorithm to optimize the weights rather than a simple linear least-square fit due to the limited scaling ranges and the fact that only W_g is signed. Once the set of weights is determined, the same weights are programmed into all the cores across all the processors. POWER7+ implements the activity proxy, Eqn. (2), in hardware.

Aside from the temperature effects previously mentioned, other sources of correctable errors and biases need to be considered in designing training experiments. One type of potentially correctable error is due to Simultaneous Multi-Threading (SMT) mode, which is effectively the number of active threads per core. We observe systematic variations of up to 5% in estimated power in the training sets based on whether the workload is running with 1, 2 or 4 threads per

core. Consequently, our training kernels are run in all SMT modes.

Even after correctable errors are considered, systematic errors exist that will always bound the maximum accuracy of an active power model based simply on event counting. One important example, to be addressed by future work, is the dependency of processor power on the actual data being processed, as the number of nodes switching at each cycle is data dependent. Cache and register file access power may also vary based on the data stored in the arrays. Using some simple cache-contained integer loops running on a POWER7 processor, we observed variations in total active power of up to 5% depending on the randomness of the data being processed, with power increasing with increasing randomness.

3.4. Chip-level power model

Activity proxy only accounts for active power at the nominal operating point, which is a specific frequency, and associated voltage values set defined in the chip VPD and identified as the default frequency for the processor. In order to achieve a true power proxy, we also must consider other factors, namely the whole supply voltage range and temperature-dependent leakage power. The impact of frequency on power consumption is largely captured by activity counters, since higher frequency results in proportionally more event counts for the same workload.

Chip power consumption can be further divided into idle power and active power, both of which are dependent on supply voltage. Idle power in turn can be divided into leakage power and clock grid power. Leakage power is also dependent on temperature. We model chip-level power consumption in the following format:

$$\begin{aligned} P_{chip} &= P_{active} + P_{clock} + P_{leak} \\ &= \frac{AP}{R}\left(\frac{V}{V_{nom}}\right)^{\alpha} + \frac{Freq}{S_0}\left(\frac{V}{V_{nom0}}\right)^{\beta} \\ &\quad + P_{leak\_nom}\left(\frac{V}{V_{nom}}\right)^{\gamma}(1+m_0(T-T_0)) \end{aligned} \tag{3}$$

where AP is the activity proxy; R is the ratio between activity proxy and active power for the reference processor at nominal frequency; V is the supply voltage at the VRM output; V_{nom} is the nominal voltage for each chip at the VRM output; $Freq$ is the chip frequency; S_0 is a constant scaling factor across all chips and is derived during chip characterization; V_{nom0} is the nominal voltage at VRM output for the characterized chip; P_{leak_nom} is the idle leakage power of the chip at nominal voltage; m_0 is a linear scaling factor for temperature-dependency of leakage power; T and T_0 are the actual temperature and characterization temperature of the chip, respectively; α, β and γ are constant exponents derived from characterizing the reference chip.

Among all these parameters in the model, S_0, T_0, m_0, V_{nom0}, α, β and γ are constants, regardless of chip-to-chip variations. R, V_{nom} and P_{leak_nom} are unique per chip, and R can be calculated based on VPD data (more details in Section 3.6). AP, $Freq$, V and T are runtime measurements at the chip level.

It is interesting to note that the active power term in Eqn. (3) does not include chip frequency. This is because changes to core frequency are reflected in the rate at which the activity proxy to accumulate events. Therefore, AP naturally incorporates core frequency.

Through a series of experiments we found the dependence of leakage power on temperature is approximately linear for a fixed voltage. This is the reason we include a linear dependence on temperature in the leakage power term in Eqn. (3). We ran different steady-state workloads such as the kernel loops and the maximum power workload with constant inputs at nominal voltage and frequency, each starting from a cool temperature and gradually reaching a warmer steady temperature. The only factor that causes power change for each workload is temperature-dependent leakage power. We found this power component is linearly proportional to temperature change.

We used the IBM SNAP genetic algorithm optimizer to determine the parameters for Eqn. (3). First, we measured the idle power and temperature of each chip across 22 voltage settings and 22 frequency settings, for a combination of 253 unique voltage-frequency points. Then we programmed SNAP to find the parameters that minimized the error for the idle power across all chips every voltage-frequency points. We obtained $S_0 = 159.634$, $m_0 \approx 0.031 W/^{\circ}C$, $\beta = 1.584$, $\gamma = 4.070$, and a unique P_{leak_nom} value for each chip.

We verified P_{leak_nom} for all the chips by measuring leakage power at nominal Vdd (when the chip is idle and before clock grid is enabled). After that, we repeat the same procedure for different voltages, to verify $\gamma = 4.070$ across all the chips. We verified S_0 by measuring chip idle power (including both leakage power and clock grid power) at nominal frequency, and subtracting leakage power calculated above from idle power.

With the total chip power measurement and the idle power model, we can calculate active power. From multiple training workloads and we find that $R = 2450$ for the reference chip, and $\alpha = 2.2$ for active power. The validation of total chip power is shown in Fig. 5 in Section 3.6, as we also show the total power for each of the four processors with manufacture-based variations.

3.5. Total chip power and idle power estimation results

We now present our main results for the total chip power model and idle power model. Additionally, we show that our models are accurate even when the chips are undervolted.

Fig. 2 shows results for total chip power for training kernels, testing kernels/SPECpower, and SPEC CPU2006. We report two metrics here. The first one is the absolute (i.e. unsigned) average percentage error among 32 ms samples of each workload, see Fig. 2(a)-(c). This metric helps evaluate instantaneous power proxy errors and is useful for what-if scenarios for runtime power management policies that need to make many decisions every second. The second metric is the average percentage error for each entire workload run, see Fig. 2(d)-(f). This is useful for evaluating energy consumption over a relatively longer time, such as a minute or longer.

We achieve 1.8% (std. dev. 2.0%) unsigned percentage errors for 32 ms samples of total chip power across all workloads (the last bar in Fig. 2(a)-(c)). The low standard deviation means the errors from a vast majority of activity proxy samples are close to each other. For mean percentage errors for each entire workload (Fig. 2(d)-(f)), on average, we achieve -0.2% (std. dev. 2.6%). This indicates the set of weights from the training is especially useful for long-term energy estimation. The worst case error across all testing workloads is under 9.5% (vector copy kernel). For SPEC CPU2006, the worst case workload error is 8.1% for *calculix*. This compares well to prior work [6] that achieved a median error of 1-5% (maximum error 7-10.7%) for SPEC CPU2006 workloads across multiple chip architectures, but used 1-second samples for validation.

Our idle power model ($P_{leak} + P_{clock}$) has a maximum error of 2 W across all chips and voltage-frequency points. The maximum

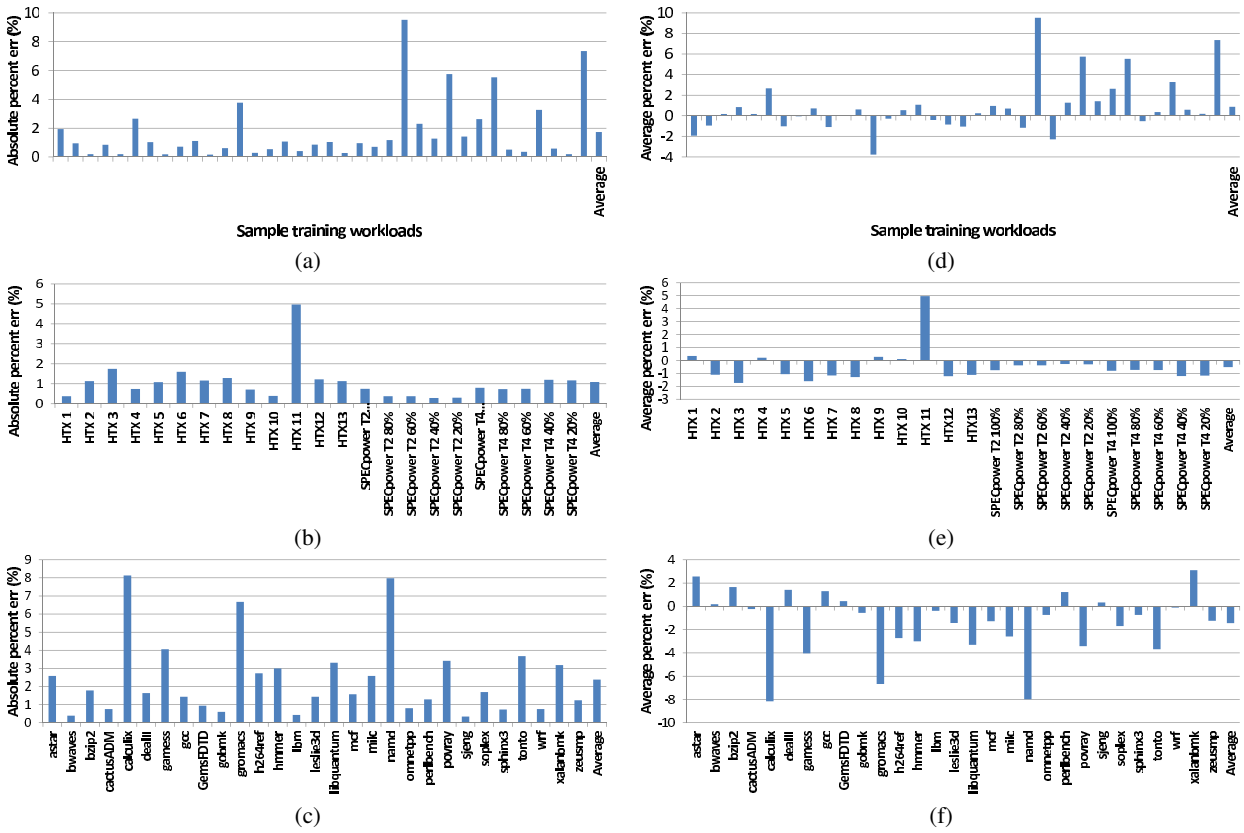


Figure 2: (a)-(c): Chip power (at nominal frequency) absolute (unsigned) percentage errors across all samples for training kernels, testing kernels/SPECpower, and SPEC CPU2006, respectively. (d)-(f): Chip power (at nominal frequency) average relative errors for each workload run.

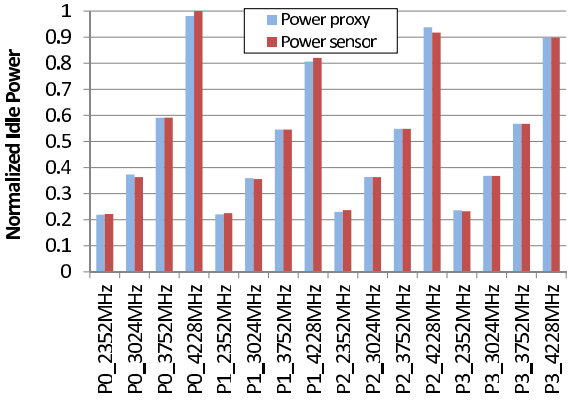


Figure 3: Idle power model validation at four different frequencies.

percent error was 3% which occurs near the lowest idle power measured. Fig. 3 shows the normalized idle power (i.e. leakage power and clock power) at different voltage-frequency pairs for all four chips.

So far, we have derived and verified the chip-level power proxy for a wide range of voltage-frequency pairs. An interesting experiment would be to test the power proxy's accuracy when voltage and frequency pairs are not fixed. Recently, Lefurgy et al. [12] proposed a method of safely undervolting a microprocessor while maintaining the same frequency. Voltage is dynamically selected to maintain a preset guardband level as chip activity changes, resulting in reduced chip power without performance loss. We apply this technique when running the maximum power workload, and allow undervolting up to 112.5 mV (about 9.5% of supply voltage) at the fixed frequency of 4228MHz. The first half of Fig. 4 is for the case where frequency is fixed at 4228 MHz and voltage is set to the traditional corresponding value, whereas the second half of the figure shows the case where dynamic undervolting is enabled such as frequency is fixed while voltage changes with available timing margin. This results in about 25% power reduction for *dealII* in the second half. Comparing the two halves, we see that for decoupled voltages and frequencies, our chip-level power proxy still achieves about the same level of accuracy when frequency and voltage are decoupled from each other. That is, 9.5% variations of supply voltage do not lead to worse power proxy estimation.

3.6. Chip-to-chip variations

The chip-level power proxy (or model) presented above is characterized from a single reference chip and does not consider chip-to-chip variations due to uncertainties in the manufacturing process. Therefore, Eqn. (3) must be adjusted to take variations into account. As mentioned before, to maximize the process variations among the tested chip, we intentionally evaluate four chips from distinct process corners. For example, when considering Performance Sort Ring Oscillator (PSRO) measurements, one chip is 2.6 standard deviation slow, one chip is 1.1 standard deviation fast and the other two chips

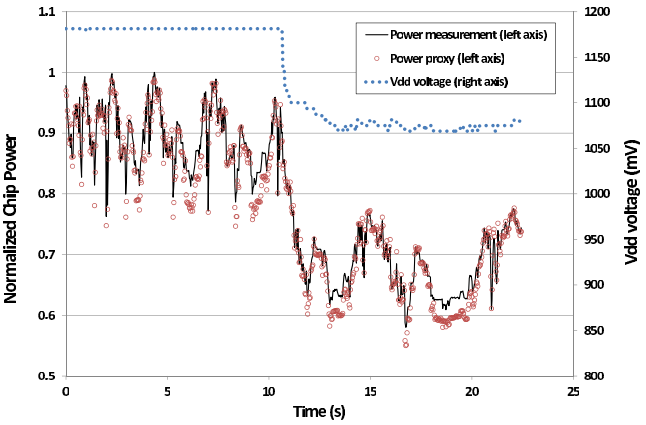


Figure 4: Chip power proxy validation for *dealII* with decoupled voltage and frequency.

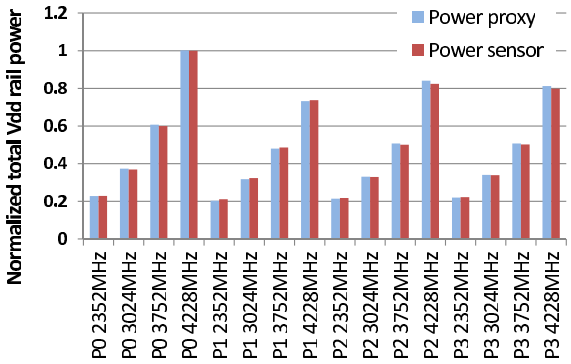


Figure 5: Chip power model validation at four different frequencies for the maximum power workload. Chip-to-chip variation is also accurately accounted for.

are within 0.5 standard deviation of the mean PSRO value. More simply, the PSRO spread of our 4 chips covers 88% of all chips from a sample size of thousands of chips passing module test. Data in the paper also show significant leakage power variations.

For clock power (P_{clock}), $Freq$ can be measured at run time, S_0 and β are constant across all chips. V_{nom0} is also a constant, which is the nominal Vdd for the reference chip. The only variance in clock power from one chip to another is from its supply voltage V that can be measured at run time. For the same operating frequency, different chips have their different characterized supply voltages. Therefore, variations among chips for the clock power component can be accounted for by the different settings of supply voltages.

For the leakage power component (P_{leak}), although each chip's leakage power at nominal voltage (the characterized voltage for this chip to run at nominal frequency) can be measured during system start up, it is also possible to calculate it by subtracting the clock power from the measured idle power for each chip. Both methods achieve almost the same accuracy. The variation among leakage power can be easily calculated from measurements. The assumptions of a constant γ and a constant m_0 for all chips are also valid according to Fig. 3.

For active power, there are two ways to adjust for chip-to-chip variations. The first approach is to utilize the characterized variation information in the VPD data. As mentioned in Section 3.1.1, each chip has VPD data that is established during chip manufacturing test that is unique to that chip. The uniqueness addresses manufacturing variability, and system specific information for the target system it is going into including load line effects between the VRM and the chip package and the losses associated with the unique package being used for that die in the system.

In Eqn. (3), for P_{active}, we can derive a chip-specific value for R based on VPD, load-line equation, and VRM efficiency. Specifically, as mentioned before, $R = (AP)_{nom}/P_{active\_nom}$. We denote R_0 for the reference chip, and R_1 for an un-characterized chip that we want to adjust the active power. If both chips run the same workload, at the same frequency, in the same environment, we know that $(AP)_{nom} = R_0 P_{active\_nom0} = R_1 P_{active\_nom1}$ because both chips have the same amount of activities. Therefore,

$$R_1 = R_0 \left(\frac{P_{active\_nom0}}{P_{active\_nom1}} \right) \tag{4}$$

where

$$P_{\text{active}\_i} = P_{\text{VRM}\_i} - P_{\text{idle}\_i} \quad \ldots(i = 0, 1) \tag{5}$$

We are able to calculate R_1 for any processors that are not characterized since 1) R_0 is known, 2) P_{idle_i} can be calculated for each chip, and 3) P_{VRM_i} can be calculated from the VPD data of chip i together with knowledge of the load-line equation and VRM efficiency.

However, VRM efficiency is quite sensitive to the load current and the number of phases, and can vary as much as 10%, causing a noticeable error in the resulting R_i calculation. Instead, we found that modifying the active power to the following format results in better accuracy for all the chips:

$$P_{active} = \frac{AP}{R_0} \left(\frac{V}{V_{nom0}} \right)^{\alpha} \tag{6}$$

Where R_0 is a constant value from the reference chip, and V_{nom0} is the nominal Vdd for the reference chip, too. The underlying reason is similar to the clock power component—chip-to-chip variation is largely captured by the different characterized supply voltage settings for different chips at the same operating frequency.

Therefore, the final format of the chip-level power proxy adjusted for chip-to-chip variations is:

$$\begin{aligned} P_{chip} &= P_{active} + P_{clock} + P_{leak} \\ &= \frac{AP}{R_0} \left(\frac{V}{V_{nom0}} \right)^{\alpha} + \frac{Freq}{S_0} \left(\frac{V}{V_{nom0}} \right)^{\beta} \\ &\quad + P_{leak\_nom} \left(\frac{V}{V_{nom}} \right)^{\gamma} (1 + m_0 (T - T_0)) \end{aligned} \tag{7}$$

Fig. 5 compares the modeled total power and measured total power for the maximum power workload for different voltage-frequency pairs, across all four chips. It shows that our proposed chip-level power proxy works accurately despite the large manufacture process variations among the chips. Similar results are achieved for other workloads, too.

3.7. Core-level power proxy

For POWER7+, in order to reach power consumption estimation associated with each core, we take the following steps.

- Active power: We begin with the per-core activity proxy value AP_i calculated by the core. R and V are the same for all the cores in one processor, since all cores share the same voltage rails in POWER7+. Specifically,

$$P_{\text{active\_core\_}i} = \frac{AP_i}{R_0}\left(\frac{V}{V_{nom0}}\right)^{\alpha} \tag{8}$$

- Clock grid power: first use average frequency across all the cores to calculate chip-level clock power. Then divide it to each core's contribution by proportionally scaling to the core's frequency. Specifically,

$$\begin{aligned} P_{\text{clock\_core\_}i} &= \frac{Freq_i}{N_{cores}Freq_{avg}}\frac{Freq_{avg}}{S_0}\left(\frac{V}{V_{nom0}}\right)^{\beta} \\ &= \frac{Freq_i}{S_0 \cdot N_{cores}}\left(\frac{V}{V_{nom0}}\right)^{\beta} \end{aligned} \tag{9}$$

- Leakage power: first use Eqn 3 to calculate chip-level leakage power. Then divide it to each core's contribution by proportionally scaling to the core's temperature change. Specifically,

$$P_{\text{leak\_core\_}i} = \frac{P_{leak\_nom}}{N_{cores}}\left(\frac{V}{V_{nom}}\right)^{\gamma}(1+m_0(T_i - T_{i0})) \tag{10}$$

  where T_i is the core temperature at run time, and T_{i0} is the core temperature during characterization.
- Final adjustment: each core's power proxy now becomes $Proxy_i = P_{\text{active\_core\_}i} + P_{\text{clock\_core\_}i} + P_{\text{leak\_core\_}i}$. To account for the difference between power proxies and power measurement, we can adjust the sum of all power proxies from all the cores to be equal to total measured chip power, by multiplying with scaling factor that equals to $P_{measured}/\Sigma Proxy_i$.

Although POWER7+ does not have individual voltage rails at the granularity of cores, the above approach can be easily extended to such situations by using V from each core. Use of core-level power proxies is shown in Section 4.1.

4. Use Cases of Power Proxies

The chip-level and core-level power proxies that incorporate voltage, frequency and process variations allow the implementation of many novel ideas that are otherwise impractical or impossible. One interesting usage scenario of per-core power proxies is in a power capping environment with a power constraint at the processor socket level. Our per-core power proxies enable a better judgment for balancing power among cores. Prior work [11] shows that every 1% improvement in power estimation accuracy can lead to roughly 1% performance improvement in a power-capped scenario, due to less guardbanding.

In general, this work provides the means of using core-level power proxies for both power-based accounting/billing of virtual machines and more fine-grained power management. This section provides two such example applications.

4.1. Fine-grained power accounting

Power proxies, especially per-core power proxies enable power-based billing for cloud computing services. Our power proxy will work with different assignments of cores to virtual machine and with the virtual machines operating the cores using different partition-level power management techniques. For virtual machines at the sub-core level (i.e. a subset of threads out of a SMT core), further extensions to per-thread power proxies are necessary, which is beyond the scope of this paper.

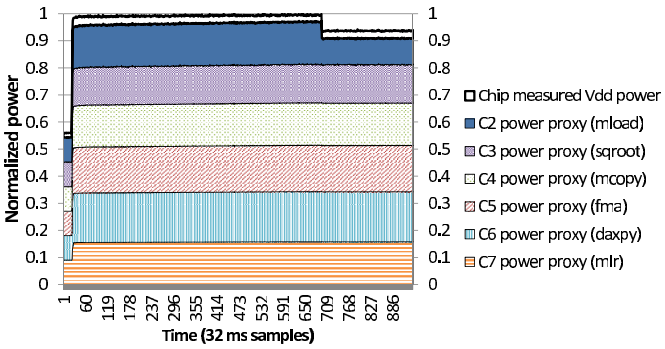


Figure 6: Stacked core power proxy for an estimate of total chip power for a multiprogram workload. Top solid line is the measured chip power.

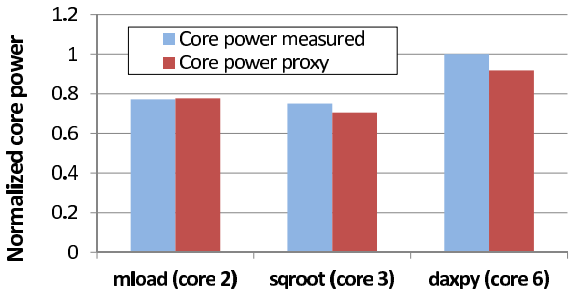


Figure 7: Validation of core power proxy for three kernel benchmarks running on individual cores.

We construct a case to show core-level power estimations on a 6-core processor. Each core runs an independent workload from our kernel benchmarks and each workload has been configured with different memory footprints and different number of threads (1, 2 or 4). The variations among workloads cause power consumption difference among cores. Fig. 6 shows the normalized power over time. We stack the six core power proxies together to get an estimated total chip power, each core power proxy is represented by one pattern in the plot. The top-most solid line is the measured chip Vdd power. As we can see, all the workloads start at the same time. But the kernel on core 2 ends earlier than the others. The sum of the core power proxies is about 3.0% less than the measured chip power.

In order to validate each core power proxy's accuracy, we pick three of the kernels that have memory footprint contained within per-core L2 cache from previous experiment, and run each alone on its associated core with all other cores idle. We calculate the "measured" core power by

$$P_{core\_measured} = \frac{P_{idle}}{N_{cores}} + (P_{chip} - P_{idle}) \tag{11}$$

and compare it with the temperature adjusted core power proxy for those cores in Fig. 6. The results in Fig. 7 show the core power proxies are quite accurate (0.6%, -6.2% and -8.2%, respectively).

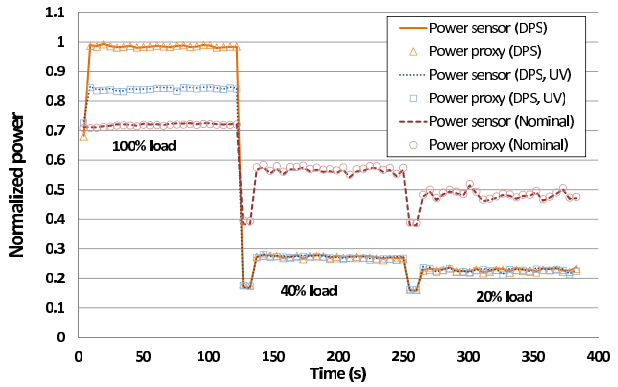


Figure 8: Chip power proxy comparison among three power management policies: Fixed nominal frequency, Dynamic Power Saving (DPS), Dynamic Power Saving with undervolting (DPS, UV).

4.2. Run-time power saving estimation

In this section, we evaluate different power management policies for a workload. Fig. 8 shows runtime power comparison of a SPECpower run under three power management policies. SPECpower requires each load level run for fixed amounts of time, regardless of operating mode. That is why the total run times in all cases are the same in the figure. Additionally, Fig. 8 shows DPS and DPS/UV have better "power proportionality" to load levels than Nominal. This results in significantly improved SPECpower scores for DPS and DPS/UV.

We show three calibration phases, one 100% load level, one 40% level and one 20% load level. The Nominal policy uses a fixed frequency and a fixed voltage throughout the run; The Dynamic Power Saving (DPS) policy dynamically adjusts voltage and frequency to keep processor at a relatively constant high utilization level. The DPS/UV (undervolting) policy allows dynamically lowering voltage for high load levels without changing frequencies, in order to reduce static margins and achieve higher power efficiency. Both chip power measurements and chip power proxy are shown. As can be seen, the chip power proxies match well with the power measurements in all cases.

It is also interesting to compare the three policies. The fixed frequency mode has relatively flat chip power consumption, despite the significant change in load levels and processor utilization levels. The DPS and DPS/UV modes achieve higher frequency, hence higher performance at high load levels, and significantly reduced power consumption at lower load levels and idle state. Thus these modes are more “power proportional” in that they respond to performance demand. In addition, the benefit of DPS/UV is evident that it consumes 15% less power at full load levels, while keeping the same peak performance.

We expect that a chip-level power proxy allows on-the-fly and accurate evaluation of such power management policies. Traditionally, each power management technique is implemented separately in hardware, firmware or software. The same set of workloads must be executed multiple times, once for each technique. Great care must be taken to ensure each run has the same architectural, environmental and initial conditions. This process is time consuming and not rigorous. With an accurate chip-level power proxy, processor power consumptions of different power management policies can be calculated simultaneously for different voltage, frequency and temperature scenarios.

5. Related Work

Bellosa [1] developed some of the first microprocessor power models based on performance counter measurement.

Contreras and Martonosi [2] augment the performance counter-based linear regression model of CPU power for a Intel PXA255 unicore processor to account for voltage scaling by using different weights for the performance counters for each voltage and frequency pairs. In modern server chips, voltages are tuned for each chip so that chips running at the same frequency in the same system likely use different voltages. As undervolting and overclocking become common place in commercial computers, the voltage and frequency pairs are not stable at run time and instead depend on the running workload and the electrical guardbands within the processor. Our methodology can account for both different chip voltages as well as dynamic undervolting and overclocking.

Intel’s Common Activity-based Model for Power (CAMP) [15] uses just 9 micro-architectural events in the processor to create models of activity factor for 180 physical structures in the processor. The activity factors are used to generate per-structure power models that can be combined to provide core-level dynamic power estimates at run time. They show an 8% average error for the core-level dynamic power and a maximum error of up to 12% for entire workloads. The comparison is against a detailed power simulator which itself is estimated to be 5% to 10% accurate. Additionally, the authors provide an excellent summary of prior work in the area. A limitation of the work is that it does not provide a methodology for dealing with run-time voltage scaling or manufacturing variation. Our work is distinguished from Intel’s CAMP work in that 1) we implement our solution in a real chip (not a simulator), 2) we account for manufacturing variation, and 3) variation in voltage and frequency. For example, two cores at different clock frequencies (due to utilization-based frequency selection), but operating on a shared voltage rail set for the higher frequency.

Jacobson et al. [8] improves on Powell et al. by providing a methodology for selecting the best architectural events. The authors use abstracted microarchitectural scaling models that are useful in early-stage power modeling of future generation designs.

Goel et al. [6] derives performance-counter based core-power models and tunes them using real system power measurement using linear regression. One difference from prior work is the inclusion of core-level temperature sensors. Another novel feature is using linear, inverse, exponential, logarithmic, or square-root transformations to scale the performance counters before the linear regression step to correlate better with power consumption. They achieve median errors per benchmark suite of 1-5% across six different CPU models, which demonstrates a portable methodology. Error in average power for individual workloads was measured up to 11%. This work has some of the best error reporting of prior work and an extensive review of prior work.

There are two recent papers that account system power to virtual machines (VMs). They both attempt to account CPU, memory, and device power to VMs running on the system. We limit discussion to the CPU power, which is the sole focus of our work.

Stoess et al. [19] account for processor power by recording CPU performance counters at VM context switches, weighting and accumulating the counters to form a power proxy, and assigning the power to the associated VM. Power during idle periods is equally di-

vided among the running VMs. A limitation of the work is its implementation on a uniprocessor system running a single core. The work does not consider how power could be allocated to virtual machines running on each core, or how voltage scaling or CPU temperature affect power consumption. We address these limitations in our work by architecting a power proxy for each processor core and accounting for voltage and temperature variation across cores and chips in the system.

Kansal et al. [10] take another approach to account for CPU power by tracking the logical CPU utilization of each VM. A simple, linear relationship is used to relate the utilization to a power consumption. The system-level power accuracy is measured to be within 5%. However, such models cannot accurately account power at a core-level due to manufacturing variation between cores, workload variation, or temperature variation. Our modeling uses fabrication-time testing to account for manufacturing variation and run-time sensing to account for workload variation and temperature.

Much of the prior work does not provide error estimates based on measured CPU power. Often system power measurement is used and discounted by power measurements for various system devices to arrive at the CPU power measurement. We base our error estimates on a highly accurate Vdd-rail current sensor for the CPU socket.

Do, Rawshdeh, and Shi [3] propose an application-level programming interface to allow processes to monitor their energy consumption. Their CPU power model relies on assigning a fixed power consumption to each processor frequency state and a fixed energy to frequency transitions. It does not deal with nuances of how instructions actually use the processor or manufacturing variation. Their reported results for an estimated system power of a laptop appear to have over 11% mean error. Our work could be used as a replacement (with higher accuracy) for their CPU power modeling.

AMD includes power-monitoring circuit in its processor cores [9]. 95 activity signals per core are monitored and weighted to form a dynamic power estimate that is considered to be 2% accurate. Since the purpose is for long-term thermal and power control, the circuitry is optimized to eliminate high-speed routing by not sampling every signal every cycle. It takes hundreds of time-based samples to achieve accurate dynamic power estimations.

Intel's Tukwila chip, an Itanium family processor, tracks approximately 120 architectural event per core to estimate switched capacitance every 8 microseconds and compare this to threshold values to select a maximum voltage-frequency pair to stay within a power envelope [16]. The application of power proxy sensors in Tukwila is guardbanding worst-case power, not attempting to replicate real power on a voltage rail. Since all processors must select the same frequency for identical instruction sequences despite manufacturing variation (leakage and circuit speed), it is not actual power that the sensors are responding to, but an estimation of power in a worst-case chip. The accuracy of these sensors compared to real power measurement is unpublished.

Our work complements the work of AMD and Intel in that we differentiate our power model across processors to calculate chip power as accurately as possible for charge-back purposes. Our power proxy has additional novel properties addressing real-world implementation challenges. First, it deals with significant chip-to-chip variations in an accurate yet concise way. Second, it is accurate for undervolting (UV) where voltage adjustment is independent of frequency. Prior work only evaluates with a fixed workset of voltage-frequency pairs. In addition, prior work on real systems [6] estimates power at 1-second intervals. Our estimates are for 32-millisecond intervals, which is more relevant for dynamic power capping and energy-efficiency controllers.

6. Conclusion

In this paper, we present accurate chip-level and core-level power proxies for the IBM POWER7+ processor. We validate the power proxies by accurately replicating an existing Vdd power rail sensor. For a fixed frequency run, we achieve a mean unsigned error of 1.8% for fine-grained 32 ms samples across all workloads. For an interval of an entire workload, we achieve a mean error of -0.2%. The worst-case workload error was under 9.5%. This accuracy is similar to the prior work with the highest accuracy, but is attained at a 30x smaller timescale which is more appropriate for fine-grain power management applications. We also show that the power proxies hold their accuracy across a range of frequency and voltage settings. Additionally, we demonstrate the first power proxies that work on a system that undervolts processors, whereas prior studies only show results for conventional voltage-and-frequency scaling with fixed voltage-frequency pairs.

Our demonstration in a real system shows the technique is sound for deployment in commercial multi-core servers. The power proxies account for full voltage and frequency ranges and also for chip-to-chip manufacturing variations. Such proxies are useful for a number of applications, such as power-based billing strategy for cloud-based services. They also enable powerful runtime what-if evaluations of different power management techniques.

Acknowledgement

We thank Jason F. Cantin for providing the IBM SNAP genetic algorithm optimizer used in this work.

References

[1] F. Bellosa, "The benefits of event-driven energy accounting in power-sensitive systems," in *Proceedings of the 9th Workshop on ACM SIGOPS European Workshop*, 2000.

[2] G. Contreras and M. Martonosi, "Power prediction for intel XScale processors using performance monitoring unit events," in *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED)*, 2005.

[3] T. Do, S. Rawshdeh, and W. Shi, "pTop: A process-level power profiling tool," in *Proceedings of the Workshop on Power Aware Computing and Systems, HotPower*, 2009.

[4] M. Floyd *et al.*, "Introducing the adaptive energy management features of the POWER7 chip," *Micro, IEEE*, vol. 31, no. 2, pp. 60–75, March/April 2011.

[5] M. Floyd *et al.*, "Adaptive energy-management features of the IBM POWER7 chip," *IBM Journal of Research and Development*, vol. 55, no. 3, pp. 8:1–8:18, May/June 2011.

[6] B. Goel *et al.*, "Portable, scalable, per-core power estimation for intelligent resource management," in *Proceedings of International Green Computing Conference (IGCC)*, 2010.

[7] Intel Corporation, *Intel Server Board S1200BT*, February 2012.

[8] H. Jacobson *et al.*, "Abstraction and microarchitecture scaling in early-stage power modeling," in *Proceedings of International Symposium on High Performance Computer Architecture (HPCA)*, 2011.

[9] R. Jotwani *et al.*, "An x86-64 core implemented in 32nm SOI CMOS," in *Proceedings of International Solid-State Circuits Conference(ISSCC)*, 2010.

[10] A. Kansal *et al.*, "Virtual machine power metering and provisioning," in *Proceedings of the ACM Symposium on Cloud Computing (SOCC)*, 2010.

[11] C. Lefurgy, X. Wang, and M. Ware, "Power capping: A prelude to power shifting," *Cluster Computing*, vol. 11, no. 2, pp. 183–195, June 2008.

[12] C. R. Lefurgy *et al.*, "Active management of timing guardband to save energy in POWER7," in *Proceedings of International Symposium on Microarchitecture (MICRO)*, 2011.
[13] J. McCalpin, "The STREAM2 Home Page," http://www.cs.virginia.edu/stream/stream2.
[14] L. W. McVoy and C. Staelin, "lmbench: Portable tools for performance analysis," in *Proceedings of the USENIX Annual Technical Conference (USENIX)*, 1996.
[15] M. Powell *et al.*, "CAMP: A technique to estimate per-structure power at run-time using a few simple parameters," in *Proceedings of International Symposium on High Performance Computer Architecture (HPCA)*, 2009.
[16] B. Stackhouse *et al.*, "A 65 nm 2-billion transistor quad-core Itanium processor," *IEEE Journal of Solid-State Circuits*, vol. 44, no. 1, pp. 18–31, January 2009.
[17] "SPEC CPU2006," http://www.spec.org/cpu2006.
[18] "SPECpower_ssj2008," http://www.spec.org/power_ssj2008.
[19] J. Stoess, C. Lang, and F. Bellosa, "Energy management for hypervisor-based virtual machines," in *Proceedings of the USENIX Annual Technical Conference (USENIX)*, 2007.
[20] M. Ware *et al.*, "Architecting for power management: The IBM POWER7 approach," in *Proceedings of International Symposium on High Performance Computer Architecture (HPCA)*, 2010.

2012 IEEE/ACM 45th Annual International Symposium on Microarchitecture

Fundamental Latency Trade-offs in Architecting DRAM Caches*

Outperforming Impractical SRAM-Tags with a Simple and Practical Design

Moinuddin K. Qureshi
Dept. of Electrical and Computer Engineering
Georgia Institute of Technology
moin@gatech.edu

Gabriel H. Loh
AMD Research
Advanced Micro Devices, Inc.
gabe.loh@amd.com

Abstract

This paper analyzes the design trade-offs in architecting large-scale DRAM caches. Prior research, including the recent work from Loh and Hill, have organized DRAM caches similar to conventional caches. In this paper, we contend that some of the basic design decisions typically made for conventional caches (such as serialization of tag and data access, large associativity, and update of replacement state) are detrimental to the performance of DRAM caches, as they exacerbate the already high hit latency. We show that higher performance can be obtained by optimizing the DRAM cache architecture first for latency, and then for hit rate.

We propose a latency-optimized cache architecture, called Alloy Cache*, that eliminates the delay due to tag serialization by streaming tag and data together in a single burst. We also propose a simple and highly effective* Memory Access Predictor *that incurs a storage overhead of 96 bytes per core and a latency of 1 cycle. It helps service cache misses faster without the need to wait for a cache miss detection in the common case. Our evaluations show that our latency-optimized cache design significantly outperforms both the recent proposal from Loh and Hill, as well as an impractical SRAM Tag-Store design that incurs an unacceptable overhead of several tens of megabytes. On average, the proposal from Loh and Hill provides 8.7% performance improvement, the "idealized" SRAM Tag design provides 24%, and our simple latency-optimized design provides 35%.*

1. Introduction

Emerging 3D-stacked memory technology has the potential to provide a step function in memory performance. It can provide caches of hundreds of megabytes (or a few gigabytes) at almost an order of magnitude higher bandwidth compared to traditional DRAM; as such, it has been a very active research area [2, 4, 7, 12, 13, 19]. However, to get performance benefit from such large caches, one must first handle several key challenges, such as architecting the tag store, optimizing hit latency, and handling misses efficiently. The prohibitive overhead of storing tags in SRAM can be avoided by placing the tags in DRAM, but naively doing so doubles the latency of DRAM cache (one access each for tag and data). A recent work from Loh and Hill [10, 11] makes the tags-in-DRAM approach efficient by co-locating the tags and data in the same row. However, similar to prior work on DRAM caches, the recent work also architects DRAM caches in largely the same way as traditional SRAM caches. For example by having a serialized tag-and-data access and employing typical optimizations such as high associativity and intelligent replacement.

We observe that the effectiveness of cache optimizations depends on technology constraints and parameters. What may be regarded as indispensable in one set of constraints, may be rendered ineffective when the parameters and constraints change. Given that the latency and size parameters of a DRAM cache are so widely different from traditional caches, and the technology constraints are disparate, we must be careful about the implicit optimizations that get incorporated in the architecture of the DRAM cache. In particular, we point out that DRAM caches are much slower than traditional caches, so optimizations that exacerbate the already high hit latency may degrade overall performance even if they provide a marginal improvement in hit rate. While this may seem to be a fairly simple and straight-forward concept, it has a deep impact (and often counter-intuitive implications) on the design of DRAM cache architectures. We explain the need for reexamining conventional cache optimizations for DRAM caches with a simple example.

Consider a system with a cache and a memory. Memory accesses incur a latency of 1 unit, and cache accesses incur 0.1 unit. Increasing the cache hit rate from 0% to 100% reduces the average latency linearly from 1 to 0.1, shown as "Base Cache" in Figure 1(a). Assuming the base cache has a hit rate of 50%, then the average memory access time for the base cache is 0.55. Now consider an optimization `A` that eliminates 40% of the misses (hit rate with `A`: 70%) but increases hit latency to 1.4x (hit latency with `A`: 0.14 unit). We want to implement `A` only if it reduces average latency. We may begin by examining the target hit-rate for `A` given the higher hit-latency, such that the average latency is equal to the base case, which we call the *Break-Even Hit Rate (BEHR)*. If the hit-rate with `A` is higher than the BEHR, then `A` will reduce average latency. For our example, the BEHR for `A` is 52%. So, we deem `A` to be a highly effective optimization, and indeed it reduces average latency from 0.55 to 0.40.

*The work on Memory Access Prediction (Section 5) was done in 2009 while the first author was a research scientist at IBM Research [15].

1072-4451/12 $26.00 © 2012 IEEE
DOI 10.1109/MICRO.2012.30

IEEE computer society

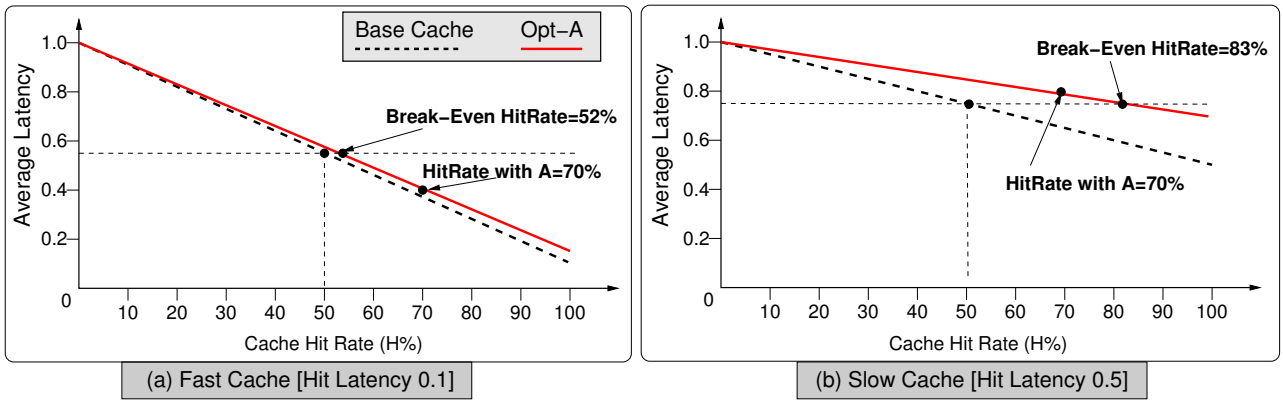


Figure 1: Effectiveness of cache optimizations depend on cache hit latency. Option A increases hit latency by 1.4x and hit-rate from 50% to 70%. (a) For a fast cache, A is highly effective at reducing average latency from 0.55 to 0.4 (b) For a slow cache, A increases average latency from 0.75 to 0.79.

Now, consider the same "highly effective" optimization A, but now the cache has a latency of 0.5 units, much like the relative latency of a DRAM cache. The revised hit latency with A will now be 1.4x0.5=0.7 units. Consider again that our base cache has a hit-rate of 50%. Then the average latency for the base cache would be 0.75 units, as shown in Figure 1(b). To achieve this average latency, A must have a hit rate of 83%. Thus optimization A, which was regarded as highly effective in the prior case, ends up increasing average latency (from 0.75 to 0.79). The Break Even Hit Rate depends also on the hit rate of the base cache. If the base cache had a hit rate of 60%, then A would need a 100% hit-rate simply to break even! Thus, seemingly indispensable and traditionally effective cache optimizations may be rendered ineffective if they have a significant impact on cache hit latency for DRAM caches. Note that typical cache optimizations, such as higher associativity and better replacement, do not usually provide a miss reduction as high as 40%, which we have considered for A. However, our detailed analysis (Section 2) shows that to support these optimizations, previously analyzed DRAM cache architectures do incur a hit latency overhead of more than 1.4x as considered for A.

It is our contention that DRAM caches should be designed from the ground-up keeping hit latency as a first priority for optimization. Design choices that increase hit latency by more than a negligible amount must be carefully analyzed to see if it indeed provides an overall improvement. We find that previously proposed designs for DRAM caches that try to maximize hit-rate are not well suited for optimizing overall performance. For example, they continue to serialize the tag and data access (similar to traditional caches), which increases hit latency significantly. They provide high associativity (several tens of ways) at the expense of hit latency. We can significantly improve the performance of DRAM caches by optimizing them for latency first, and then for hit rate. With this insight, this paper makes following contributions:

1. We analyze the latency of three designs: SRAM-Tags, the proposal from Loh and Hill, and an ideal latency-optimized DRAM cache. We find that the Loh-Hill proposal suffers from significant latency overheads due to tag serialization and due to the MissMap predictor. For SRAM-Tags, tag serialization latency limits performance. Both designs leave significant room for performance improvement compared to the latency-optimized design.
2. We show that *de-optimizing* the DRAM cache from a highly-associative structure to direct-mapped improves performance by reducing the hit latency, even if it degrades cache hit rate. For example, simply configuring the design of Loh and Hill from 29-way to direct-mapped enhances performance improvement from 8.7% to 15%. However, this design still suffers from tag serialization due to separate accesses to the "tag-store" and "data-store."
3. We propose the *Alloy Cache*, a highly-effective latency-optimized cache architecture. Rather than splitting cache space into "tag store" and "data store," it tightly integrates or *alloys* the tag and data into one unit (Tag and Data, TAD). Alloy Cache streams out a TAD unit on each cache access, thus avoiding the tag serialization penalty.
4. We present a simple and effective *Memory Access Predictor* [15] to avoid the cache access penalty in the path of servicing cache miss. Unlike MissMap, which incurs multi-megabyte storage and L3 access delay, our proposal requires a storage overhead of 96 bytes per core and incurs a latency of 1 cycle. Our predictor provides a performance improvement within 2% of a perfect predictor.

Our evaluations with a 256MB DRAM cache show that, on average, our latency-optimized design (35%) significantly outperforms both the proposal from Loh and Hill (8.7%) as well as the impractical SRAM-Tag design (24%). Thus, our simple design with less than 1KB overhead (due to predictor) provides 1.5x the performance benefits of the SRAM design that requires several tens of megabytes of overhead.

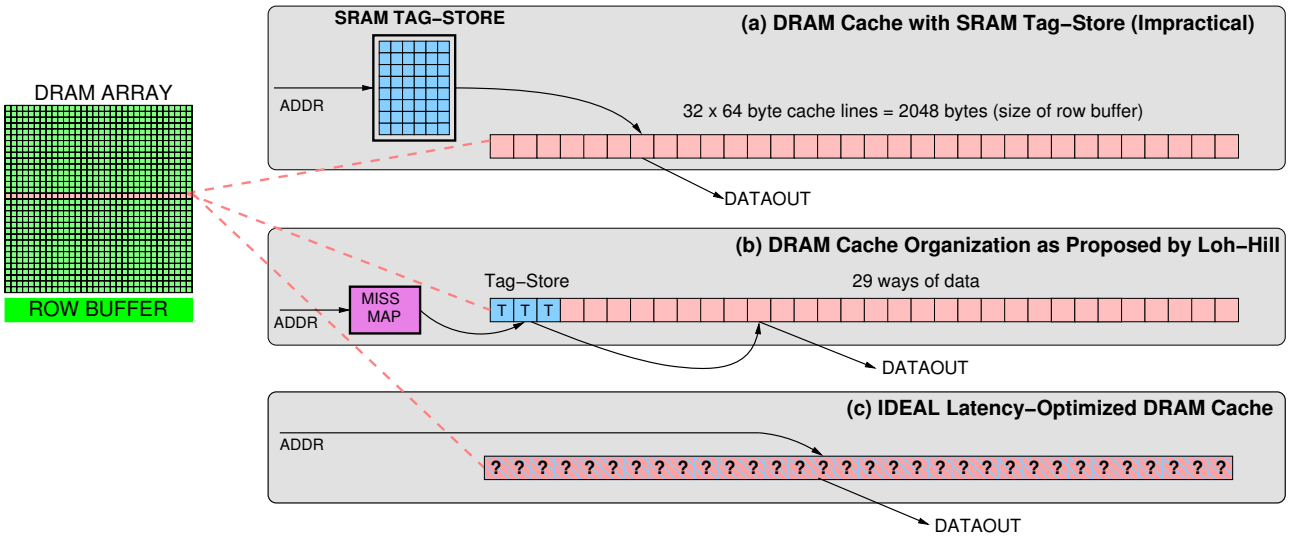


Figure 2: DRAM Cache organization and flow for a typical access for (a) SRAM Tag-store, (b) the organization proposed by Loh and Hill, and (c) an IDEAL latency-optimized cache.

2. Background and Motivation

While stacked memory can enable giga-scale DRAM caches, several challenges must be overcome before such caches can be deployed. An effective design of DRAM cache must balance (at-least) four goals. First, it should minimize the non-DRAM storage required for cache management (using a small fraction of DRAM space is acceptable). Second, it should minimize hit latency. Third, it should minimize miss latency, so that misses can be sent to memory quickly. Fourth, it should provide a good hit-rate. These requirements are often conflicting with each other, and a good design must balance these appropriately to maximize performance.

It is desirable to organize DRAM caches at the granularity of a cache line in order to efficiently use cache capacity, and to minimize the consumption of main memory bandwidth [10]. One of the main challenges in architecting a DRAM cache at a line granularity is the design of the tag store. A per-line tag overhead of 5-6 bytes quickly translates into a total tag-store overhead of a few tens of megabytes for a cache size in the regime of a few hundred megabytes. We discuss the options to architect the tag store, and how it impacts cache latency.

2.1. SRAM-Tag Design

This approach stores tags in a separate SRAM structure, as shown in Figure 2(a). For the cache sizes we consider, this design incurs an unacceptably high overhead (24MB for 256MB DRAM cache). We can configure the DRAM cache as a 32-way cache and store the entire set in one row of the cache [2, 10]. To obtain data, the access must first go through the tag-store. We call the latency due to serialization of tag access as *"Tag Serialization Latency" (TSL)*. TSL directly impacts the cache hit latency, and hence must be minimized.

2.2. Tags-in-DRAM: The LH-Cache

We can place the tags in DRAM to avoid the SRAM overhead. However, naively doing so would require that each DRAM cache access incurs a latency of two accesses, one for tag and the other for data, further exacerbating the already high hit latency. A recent work from Loh and Hill [10, 11] reduces the access penalty of DRAM tags by co-locating the tags and data for the entire set in the same row, as shown in Figure 2(b). It reserves three lines in a row for tag store, and makes the other 29 lines available as data lines, thus providing a 29-way cache. A cache access must first obtain the tags, and then the data line. The authors propose *Compound Access Scheduling* so that the second access (for data) is guaranteed to get a row buffer hit. However, the second access still incurs approximately half the latency of the first, so this design still incurs significant TSL overhead.

Given that the tag check incurs a full DRAM access, the latency for servicing a cache miss is increased significantly. To service cache misses quickly, the authors propose a *MissMap* structure that keeps track of the lines in the DRAM cache. If a miss is detected in the MissMap, then the access can go directly to memory without the need to wait for a tag check. Unfortunately, the MissMap structure requires multi-megabyte storage overhead. To implement this efficiently, the authors propose to embed the MissMap in the L3 cache. The MissMap is queried on each L3 miss, which means that the extra latency of the MissMap, which we call *Predictor Serialization Latency (PSL)*, is added to the latency of both cache hit and cache miss. Thus, the hit latency suffers from both TSL and PSL. Throughout this paper, we will assume that the design from Loh and Hill [10] is always implemented with the MissMap, and we will refer to it simply as the *LH-Cache*.

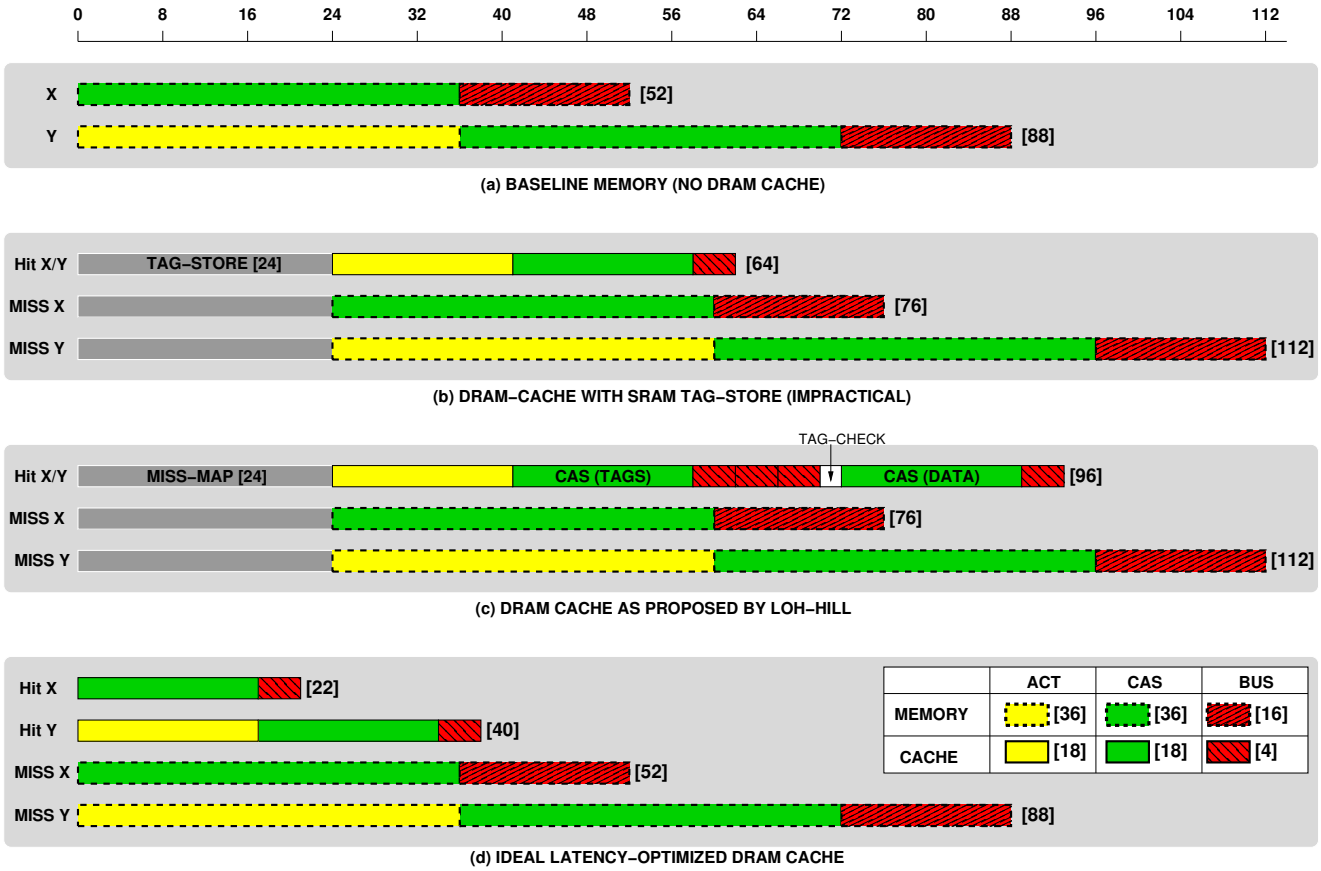


Figure 3: Latency breakdown for two classes of isolated accesses X and Y. X has good row buffer locality and Y needs to activate the memory row to get serviced. The latency incurred in an activity is marked as [N] processor cycles.

2.3. IDEAL Latency-Optimized Design

Both SRAM-Tags and LH-Cache have hit latency due to TSL. To reduce conflict misses, both designs are configured similar to conventional set-associative caches. They place the entire set in a row for conflict miss reduction, sacrificing the row-buffer hits for cache accesses (sequentially-addressed lines map to different sets, and the probability of temporally-close accesses going to same set is $\ll 1\%$). Furthermore, for LH-Cache, supporting high associativity incurs higher latency due to streaming a large number of tag lines, and the bandwidth consumed due to replacement update and victim selection further worsens the already high hit latency.

We argue that DRAM caches must be architected to minimize hit latency. This can be done by a suitable cache structure that avoids extraneous latency overheads and supports row buffer locality. Ideally, such a structure would have zero TSL and PSL, and would stream out exactly one cache line after a latency equal to the raw latency of the DRAM structure (ACT+CAS for accesses that open the row, and only CAS for row-buffer hits). Also, it would know a priori if the access would hit in cache or go to memory. We call such a design as *IDEAL-LO (Latency Optimized)*. As shown in Figure 2(c), it does not incur any latency overheads.

2.4. Raw Latency Breakdown

In this section, we quantitatively analyze the latency effectiveness of different designs. While there are several alternative implementations of both SRAM-Tags and LH-Cache, we will restrict the analysis in this section to the exact implementation of SRAM-Tags and LH-Cache as previously described, including identical latency numbers for all parameters [10], which are summarized in Table 2. We report latency in terms of processor cycles. Off-chip memory memory has tACT and tCAS of 36 cycles each, and needs 16 cycles to transfer one line on the bus. Stacked DRAM has tACT and tCAS of 18 cycles each, and needs 4 cycles to transfer one line on the bus. The latency for accessing the L3 cache as well as the SRAM-Tag store is assumed to be 24 cycles.

To keep the analysis tractable, we will initially consider only isolated accesses of two types, X and Y. Type X has a high row buffer hit-rate for off-chip memory and is serviced by memory with a latency equal to a row buffer hit. Type Y needs to open the row in order to get serviced. The baseline memory system would service X in 52 cycles (36 for CAS, and 16 for Bus), and Y in 88 cycles (36 for ACT, 36 for CAS, and 16 for Bus). Figure 3 shows the latency incurred by different designs to service X and Y.

As both SRAM-Tags and LH-Cache map the entire set to a single DRAM row, they get poor row buffer hit-rates in the DRAM cache. Therefore for both X and Y, neither cache design will give a row buffer hit. Therefore, a hit for both X and Y will incur a latency of ACT. However, with IDEAL-LO, X gets a row buffer hit and Y will need a latency of ACT.

The SRAM-Tag suffers a Tag Serialization Latency of 24 cycles for both cache hits and misses. A cache hit needs another 40 cycles (18 ACT + 18 CAS + 4 burst), for a total of 64 cycles. Thus SRAM-Tag increases latency for hits on X, decreases latency for hits on Y, and increases latency for misses on both X and Y due to the inherent latency of tag-lookup.

LH-Cache first probes the MissMap, which incurs a latency of 24 cycles.[1] For a hit, LH-Cache then issues a read for tag information (ACT+CAS, 36 cycles), then it streams out the three tag lines (12 cycles), followed by one DRAM cycle for tag check. This is followed by access to the data line (CAS+burst). Thus a hit in LH-Cache incurs a latency of 96 cycles, almost doubling the latency for X on hit, degrading the latency for Y on hit, and adding MissMap latency to miss.

An IDEAL-LO organization would service X with a row buffer hit, reducing the latency to 22 cycles. A hit for Y would incur 40 cycles. IDEAL-LO does not increase miss latency.

To summarize, we assumed that the raw latency of the stacked DRAM cache is half that of the off-chip memory. However, due to the inherent serialization latencies, LH-Cache (and in most cases SRAM-Tag) has a higher raw latency than off-chip memory. Whereas, IDEAL-LO continues to provide a reduction in hit latency on cache hits.

2.5. Bandwidth Benefits of DRAM Cache

Even with a higher raw hit latency than main memory, both LH-Cache and SRAM-Tag can still improve performance by providing two indirect benefits. First, stacked DRAM has $\sim$8x more bandwidth than off-chip DRAM, which means cache requests wait less. Second, contention for off-chip memory is reduced as DRAM cache hits are filtered. The performance benefit of LH-Cache and SRAM-Tags comes largely from these two indirect benefits and not due to raw latency. The first benefit relies on having a cache that has high-bandwidth. Although stacked DRAM has 8x raw bandwidth compared to off-chip, LH-Cache uses more than 4x line transfers on each cache access (3 for tag, 1 for data, and some for update), so the effective bandwidth becomes $<$ 2x. Both SRAM-Tag and IDEAL-LO maintains 8x bandwidth by efficiently using the bandwidth. Therefore, they are more effective than LH-Cache at reducing waiting time for cache requests. We found that the latency for servicing requests from off-chip memory is similar for all three designs.

[1]The MissMap serialization latency can be avoided by probing the MissMap in parallel with L3 access. However, this would double the L3 accesses, as MissMap would be probed on L3 hits as well, causing bank/port contention and increasing L3 latency and power consumption. Hence, prior work [10] used serial access for MissMap, and so did we.

2.6. Performance Potential

Figure 4 compares the performance of three designs: SRAM-Tag, LH-Cache, and IDEAL-LO. The numbers are speedups with respect to a baseline that does not have a DRAM cache, and are reported for a DRAM cache of size 256MB (methodology in Section 3).

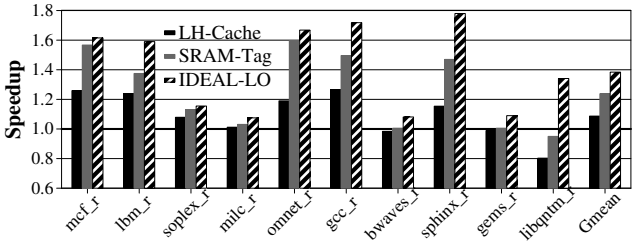


Figure 4: Performance potential of IDEAL-LO design.

Validation with Prior Work: On average, SRAM-Tags provide a performance improvement of 24% and LH-Cache 8.7%. Thus, LH-Cache obtains only one-third of the performance benefit of SRAM-Tags which is inconsistent with the original LH-Cache study [10], which reported that the LH-Cache obtains performance very close to SRAM-Tag. Given the difference in raw hit latencies between the two designs (see Figure 3) and 4x bandwidth consumption of LH-Cache compared to SRAM-Tags, it is highly unlikely that LH-Cache would perform close to SRAM-Tags. A significant part of this research study was to resolve this inconsistency with previously reported results. The authors of the LH-Cache study [10] have subsequently published an errata [9] that shows revised evaluations after correcting deficiencies in their evaluation infrastructure. The revised evaluations for 256MB show on average $\approx$10% improvement for LH-Cache and $\approx$ 25% for SRAM-Tag, consistent with our evaluations.

Note that IDEAL-LO outperforms both SRAM-Tags and LH-Cache, and provides an average of 38%. For libquantum, the memory access patterns has very high row-buffer hit rates in the off-chip DRAM resulting in mostly type X requests. Therefore, both SRAM-Tag and LH-Cache show performance degradations due to their inabilities to exploit the spatial locality of sequential access streams.

2.7. De-Optimizing for Performance

We now present simple *de-optimizations* that improve the overall performance of LH-Cache, at the expense of hit-rate. The first is using a replacement scheme that does not require update (random replacement, instead of LRU-based DIP): This avoids LRU-update and victim selection overheads, which improves hit latency due to the reduced bank contention. The second converts LH-Cache from 29-way to direct mapped. This has two advantages: direct, in that we do not need to stream out three tag lines on each access, and indirect, employing open page mode for lower latency. For

SRAM-Tag and LH-cache, sequentially addressed cachelines are mapped to different sets, and because each set is mapped to a unique row, the probability of a row-buffer hit is very low. With a direct-mapped organization, several consecutive sets map to the same physical DRAM row, and so accesses with spatial locality result in row buffer hits. The row-buffer hit rate for the direct-mapped configuration was measured to be 56% on average, compared to less than 0.1% when the entire set (29-way or 32-way) is mapped to the same row.

Table 1: Impact of De-Optimizing LH-Cache.

Configuration	Speedup	Hit-Rate	Hit Latency (cycles)
LH-Cache	8.7%	55.2%	107
LH-Cache + Rand Repl	10.2%	51.5%	98
LH-Cache (1-way)	**15.2%**	**49.0%**	**82**
SRAM-Tag (32-way)	23.8%	56.8%	67
SRAM-Tag (1-way)	24.3%	51.5%	59
IDEAL-LO (1-way)	38.4%	48.2%	35

Table 1 shows the speedup, hit-rate, and average hit latency for various flavors of LH-Cache. We also compare them with SRAM-Tag and IDEAL-LO. LH-Cache has hit latency of 107 cycles, almost 3x compared to IDEAL-LO. De-optimizing LH-Cache reduces the latency to 98 cycles (random replacement) and 82 cycles (direct mapped). These optimizations reduce hit-rate and increase misses significantly (a reduction in hit-rate from 55% to 49% represents almost 15% more misses). However, this still improves performance significantly. For SRAM-Tag, converting from 32-way to 1-way had little benefit ($\approx 0.5\%$), as the reduction in hit latency is offset by reduction in hit-rate.

While a direct-mapped implementation of LH-cache is more effective than the set-associative implementation, it still suffers from Tag Serialization Latency, as well as the Predictor Serialization Latency, resulting in a significant performance gap between LH-Cache and IDEAL-LO (15% vs. 38%). Our proposal removes these serialization latencies and obtains performance close to IDEAL-LO. We describe our experimental methodology before describing our solution.

3. Experimental Methodology

3.1. Configuration

We use a Pin-based x86 simulator with a detailed memory model. Table 2 shows the configuration used in our study. The parameters for the L3 cache and DRAM (off-chip and stacked) are identical to the original LH-Cache study [10], including a 24-cycle latency for the SRAM-Tag. For LH-Cache, we model an idealized unlimited-size Miss Map that resides in the L3 cache but does not consume any L3 cache capacity. For both LH-Cache and SRAM-Tag we use LRU-based DIP [16] replacement. We will perform detailed studies for a 256MB DRAM cache. In Section 6.1, we will analyze cache sizes ranging from 64MB to 1GB.

Table 2: Baseline Configuration

Processors	
Number of cores	8
Frequency	3.2GHz
Width	1 IPC
Last Level Cache	
L3 (shared)	8MB, 16-way 24 cycles
Off-Chip DRAM	
Bus frequency	800 MHz (DDR 1.6 GHz)
Channels	2
Ranks	1 Rank per channel
Banks	8 Banks per rank
Row buffer size	2048 bytes
Bus width	64 bits per channel
tCAS-tRCD-tRP-tRAS	9-9-9-36
Stacked DRAM	
Bus frequency	1.6GHz (DDR 3.2GHz)
Channels	4
Banks	16 Banks per rank
Bus width	128 bits per channel

3.2. Workloads

We use a single SimPoint [14] slice of 1 billion instructions for each benchmark from the SPEC2006 suite. We perform evaluations by executing 8 copies of each benchmark in rate mode. Given that our study is about large caches, we perform detailed studies only for the 10 workloads that have a speedup of more than 2 with a perfect L3 cache (100% hit-rate). Other workloads are analyzed in Section 6.4.

Table 3 shows the workloads sorted based on perfect L3 speedup, the Misses Per 1000 Instructions (MPKI), and footprint (the number of unique lines multiplied by linesize). We model a virtual-to-physical mapping to ensure two benchmarks do not map to the same physical address. We use a suffix *_r* with the name of the benchmark to indicate rate mode.

We perform timing simulation until all benchmarks in the workload finish execution and measure the execution time of the workload as the average execution time across all 8 cores.

Table 3: Benchmark Characteristics.

Workload Name	Perfect-L3 Speedup	MPKI	Footprint
mcf_r	4.9x	74.0	10.4 GB
lbm_r	3.8x	31.8	3.3 GB
soplex_r	3.5x	27.0	1.9 GB
milc_r	3.5x	25.7	4.1 GB
omnetpp_r	3.1x	20.9	259 MB
gcc_r	2.8x	16.5	458 MB
bwaves_r	2.8x	18.7	1.5 GB
sphinx_r	2.4x	12.3	80 MB
gems_r	2.2x	9.7	3.6 GB
libquantum_r	2.1x	25.4	262 MB

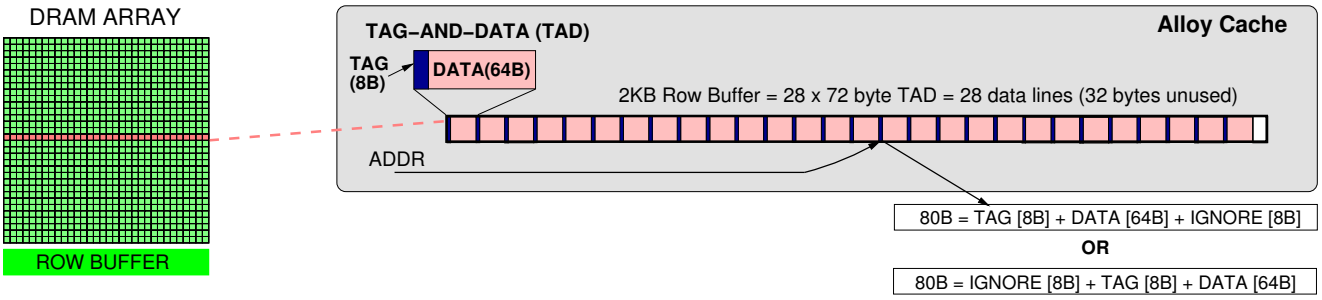


Figure 5: Architecture and Operation of Alloy Cache that integrates Tag and Data (TAD) into a single entity called TAD. The size of data transfers is determined by a 16-byte wide data-bus, hence minimum transfer of 80 bytes for obtaining one TAD.

4. Latency-Optimized Cache Architecture

While configuring the LH-Cache from a 29-way structure to a direct-mapped structure improved performance (from 8% to 15%), it still left significant room for improvement compared to a latency-optimized solution (38%). One of the main sources of this gap is the serialization latency due to tag lookup. We note that LH-Cache created a separate "tag-store" and "data-store" in the DRAM cache, similar to conventional caches. A separate tag-store and data-store makes sense for a conventional cache, because they are indeed physically separate structures. The tag-store is optimized for latency to support quick-lookups and can have multiple ports, whereas the data-store is optimized for density. We make an important observation that creating a separate contiguous tag-store (similar to conventional caches) is not necessary when tags and data co-exist in the same DRAM array.

4.1. Alloy Cache

Obviating the separation of tag-store and data-store can help us avoid the TSL overhead. This is the key insight in our proposed cache structure, which we call the *Alloy Cache*. The Alloy Cache tightly integrates or *alloys* tag and data into a single entity called *TAD (Tag and Data)*. On an access to the Alloy Cache, it provides one TAD. If the tag obtained from the TAD matches with the given line address, it indicates a cache hit and the data line in the TAD is supplied. A tag mismatch indicates cache miss. Thus, instead of having two separate accesses (one to the "tag-store" and the other to the "data-store"), Alloy Cache tightly integrates those two accesses into a single unified access, as shown in Figure 5. On a cache miss, there is a minor cost in that bandwidth is consumed transferring a data line that is not used. Note that this overhead is still substantially less than the *three* tag lines that must be transferred for both hits and misses in the LH-Cache.

Each TAD represents one set of the direct-mapped Alloy Cache. Given that the Alloy Cache has a non-power-of-two number of sets, we cannot simply use the address bits to identify the set. We assume that a modulo operation on the line address is used to determine the set index of the Alloy Cache.[2] A non-power-of-two number of sets also means that the tag entry needs to store full tags, which increases the size of the tag entry. We estimate that a tag entry of 8 bytes is more than sufficient for the Alloy Cache (for a physical address space of 48-bits, we need 42 tag bits, 1 valid bit, 1 dirty bit, and the remaining 20 bits for coherence support and other optimizations). The minimum size of a TAD is thus 72 bytes (64 bytes for data line and 8 bytes for tag). The Alloy Cache can store 28 lines in a row, reaching close to the 29-lines per row storage efficiency of the LH-Cache.

The size of data transfer from the Alloy Cache is also affected by the physical constraints of the DRAM cache. For example, the size of the databus assumed for our stacked DRAM configuration is 16 bytes, which means transfers to-and-from the cache occur at the granularity of 16 bytes. Thus, it will take a burst of five transfers to obtain one TAD of 72 bytes. To keep our design simple, we restrict the transfers to be aligned at the granularity of the data-bus size. This requirement means that for odd sets of the Alloy Cache, the first 8 bytes are ignored and for even sets the last 8 bytes are ignored. The tag-check logic checks either the first eight bytes or the next eight bytes depending on the low bit of the set index.

4.2. Impact on Effective Bandwidth

Table 4 compares the effective bandwidth of servicing one cache line from various structures. The raw bandwidths and effective bandwidths are normalized to off-chip memory. On a cache hit, LH-Cache transfers (3 lines of tag + 1 data + replacement update) reducing raw bandwidth of 8x into an effective bandwidth of less than 2x. Whereas, Alloy Cache can provide an effective bandwidth of up-to 6.4x.

[2]Designing a general purpose modulo-computing unit incurs high area and latency overheads. However, here we compute modulo with respect to a constant, so it is much simpler and faster compared to a general-purpose solution. In fact, modulo with respect to 28 (number of sets in one row of Alloy Cache) can be computed easily with eight 5-bit adders using residue arithmetic (28=32-4). This value can then be removed from the line address to get row-id of DRAM cache. We estimate the calculation to take two cycles and only a few hundred logic gates. We assume that the index calculation of the Alloy Cache happens in parallel with the L3 cache access (thus, we have up to 24 cycles to calculate the set index of the Alloy Cache).

Table 4: Bandwidth comparison (relative to off-chip memory).

Structure	Raw Bandwidth	Transfer per access (hit)	Effective Bandwidth
Off-chip Memory	1x	64 byte	1x
SRAM-Tag	8x	64 byte	8x
LH-Cache	8x	(256+16) byte	1.8x
IDEAL-LO	8x	64 byte	8x
Alloy Cache	8x	80 byte	6.4x

4.3. Latency and Performance Impact

The Alloy Cache avoids tag serialization. Instead of two serialized accesses, one each for tag and data, it provides tag and data in a single burst of five transfers on the data-bus. Comparatively, a transfer of only the data line would take four transfers, so the latency overhead of transferring TAD instead of only the data line is 1 bus cycle. However, this overhead is negligible compared to the TSL overhead incurred by SRAM-Tag (24 cycles) and LH-Cache (32-50 cycles). Because of the avoidance of TSL, the average hit latency for Alloy Cache is significantly better (42 cycles), compared to both SRAM-Tag (69 cycles) and LH-Cache (107 cycles).

The Alloy Cache reduces the TSL but not the PSL, so the overall performance depends on how misses are handled. We consider three scenarios: First, no prediction (wait for tag access until cache miss is detected). Second, use the MissMap (PSL of 24 cycles). Third, perfect predictor (100% accuracy, 0 latency). Figure 6 compares the speedup of these to the impractical SRAM-Tag design configured as 32-way.

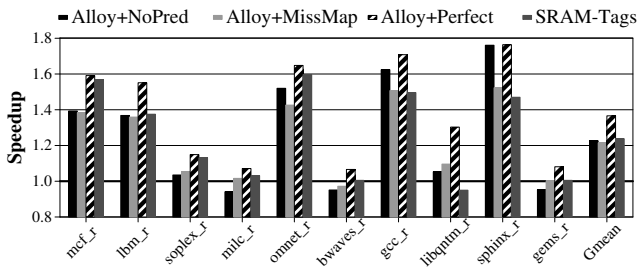


Figure 6: Speedup with Alloy Cache.

Even without any predictor, the Alloy Cache provides a 21% performance improvement, much closer to the impractical SRAM-Tag. This is primarily due to the lower hit latency. A MissMap provides better miss handling, but the 24-cycle PSL is incurred on both hits and misses, so the performance is actually worse than not using a predictor. With a perfect predictor (100% accuracy and zero-cycle latency), the Alloy Cache's performance increases to 37%. The next section describes effective single-cycle predictors that obtain performance close to that with a perfect predictor.

5. Low-Latency Memory Access Prediction

The MissMap approach focuses on getting perfect information about the presence of the line in the DRAM cache. Therefore, it needs to keep track of information on a per-line basis. Even if this incurred a storage of one-bit per line, given that a large cache can have many millions of lines, the size of the MissMap quickly gets into the megabyte regime. Given the large size of the MissMap, it is better to avoid dedicated storage and store it in an already existing on-chip structure such as the L3 cache. Hence, it incurs a significant latency of L3 cache access (24 cycles). In this section, we will describe accurate predictors that incur negligible storage and delay. We lay the background for operating such a predictor before describing the predictor. The ideas described in this Section are derived from the prior work from Qureshi [15].

5.1. Serial Access vs. Parallel Access

The implicit assumption made in the LH-Cache study was that the system needs to ensure that there is a DRAM cache miss before accessing memory. This assumption is similar to how conventional caches operate. We call this the *Serial Access Model (SAM)*, as the cache access and memory access get serialized. The SAM model is bandwidth-efficient as it sends only the cache misses to main memory, as shown in Figure 7.

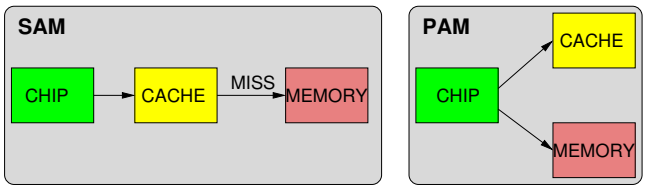


Figure 7: Cache Access Models: Serial vs Parallel

Alternatively, we may choose to use a less bandwidth efficient model, which probes both the cache and memory in parallel. We call this the *Parallel Access Model (PAM)*, as shown in Figure 7. The advantage of PAM is that it removes the serialization of the cache-miss detection latency from the memory access path. To implement PAM correctly though, we should give priority to cache content rather than the memory content, as cache content can be dirty. Also, if the memory system returns data before the cache returns the outcome of the tag check, then we must wait before using the data as the line could still be present in a dirty state in the cache.

At first blush, it may seem wasteful to access the DRAM cache in case of a DRAM cache miss. However, for both LH-Cache and Alloy Cache, the tags are located in DRAM. So, even on a DRAM cache miss, we still need to read the tags anyway to select a victim line and check if the victim is dirty (to schedule writeback). So, PAM does not have a significant impact on cache utilization compared to a perfect predictor.

5.2. To Wait or Not to Wait

We can get the best of both SAM and PAM by dynamically choosing between the two, based on an estimate of whether the line is likely to be present in the cache or not. We call this *Dynamic Access Model (DAM)*. If the line is likely to be present in the cache, DAM uses SAM to save on memory bandwidth. And if the line is unlikely to be present, DAM uses PAM to reduce latency. Note that DAM does not require *perfect information* for deciding between SAM and PAM, but simply a *good estimate*. To help with this estimate, we propose a hardware-based *Memory Access Predictor (MAP)*. To keep the latency of our predictor to a bare minimum, we consider only simple predictors.

5.3. Memory Access Predictor

The latency savings of PAM and the bandwidth savings of SAM depend on the cache hit rate. If the cache hit rate is very high, then SAM can reduce bandwidth. If the cache hit-rate is very low, then PAM can reduce latency. So, we can simply use cache hit rate for memory-access prediction. However, it is well known that both cache misses and hits show good correlation with previous outcomes [5] and exploiting such correlation results in more effective prediction than simply using the hit-rate. For example, if H is hit and M is miss, and the last eight outcomes are MMMMHHHH, then using the hit-rate would give an accuracy of 50%, but a simple last-time predictor would give an accuracy of 87.5% (assuming the first M was predicted correctly). Based on this insight, we propose to use *History-Based Memory-Access Predictors*.

5.3.1. Global-History Based MAP (MAP-G)

Our basic implementation, called *MAP Global* or *MAP-G*, uses a single saturating counter called the *Memory Access Counter (MAC)* that keeps track if the recent L3 misses resulted in a memory access or a hit in the DRAM cache. If the L3 miss results in a memory access, then the MAC is incremented, otherwise MAC is decremented (both operations are done using saturating arithmetic). For prediction, MAP-G simply uses the MSB of the MAC to decide if the L3 miss should employ SAM (MSB=0) or PAM (MSB=1). We employ MAP-G on a per-core basis and use a 3-bit counter for the MAC. Our results show that MAP-G bridges more than half the performance gap between SAM and perfect prediction. Note that because writes are not on the critical path (at this level, writes are mainly due to dirty evictions from on-chip caches), we do not make predictions for writes and simply employ SAM.

5.3.2. Instruction-Based MAP (MAP-I)

We can improve the effectiveness of MAP-G by exploiting the well-known observation that the cache hit/miss information is heavily correlated with the instruction address that caused the cache access [3, 8, 18]. We call this implementation *Instruction-Based MAP* or simply *MAP-I*. Instead of using a single MAC, MAP-I uses a table of MACs, called the *Memory Access Counter Table (MACT)*. The address of the L3 miss causing instruction is hashed (using folded-xor [17]) into the MACT to obtain the desired MAC. All predictions and updates happen based on this MAC. We found that simply using 256 entries (8-bit index) in the MACT is sufficient. The storage overhead for this implementation of MAP-I is 256*3-bit=96 bytes. We keep the MACT on a per-core basis to avoid interference between the cores (for eight cores, total overhead is only 96*8=768 bytes). Like MAP-G, MAP-I does not make predictions for write requests.

Note that our predictors do not require that the instruction address be stored in the cache. For read misses, the instruction address of miss causing load is forwarded with the miss request. As writeback misses are serviced with SAM, we do not need instruction addresses for writebacks.

5.4. Performance Results

Figure 8 shows the speedup from the Alloy Cache with different memory access predictors. If we use a prediction of always-cache-hit the system behaves like SAM, and if we use a prediction of never-cache-hit the system behaves like PAM. The perfect predictor assumes 100% accuracy at zero latency.

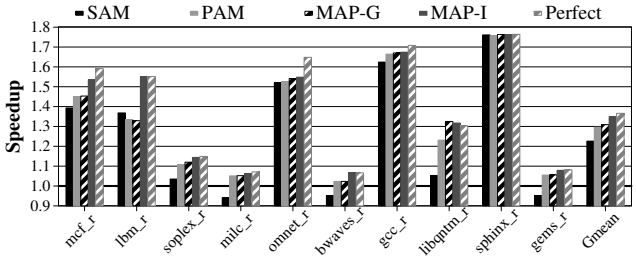


Figure 8: Performance improvement of Alloy Cache for different Memory Access Predictors

On average, there is a 14% gap between SAM (22.6%) and perfect prediction (36.6%). PAM provides 29.6% performance improvement but results in almost twice as many memory accesses as perfect prediction. MAP-G provides 30.9% performance, bridging half the performance difference between SAM and the perfect predictor. It thus performs similar to PAM but without doubling the memory traffic. MAP-I provides an average of 35%, coming within 1.6% of the performance of a perfect predictor. Thus, even though our predictors are simple ($<$ 100 bytes per core) and low latency (1 cycle), they get almost all of the potential performance.

For libquantum, MAP-G performs 3% better than the perfect predictor. This happens because some of the mispredictions avoid the row buffer penalty for later demand misses. For example, consider four lines A, B, C, D that map to the same DRAM row. Only A and B are present in the DRAM cache. A, B, C, D are accessed in a sequence. If A and B are predicted correctly, C would incur a row opening penalty

when it goes to memory. If, on the other hand, A is mispredicted it would avoid the row opening penalty for C.

5.5. Prediction Accuracy Analysis

To provide insights into the effectiveness of the predictors, we analyzed different outcome-prediction scenarios. There are four cases: 1) L3 miss is serviced by memory and our predictor predicts it as such, 2) L3 miss is serviced by memory and our predictor predicts that it will be serviced by the DRAM cache, 3) L3 miss is serviced by the DRAM cache and our predictor predicts memory access, and 4) L3 miss is serviced by the DRAM-cache and our predictor predicts it to be so. Scenarios 2 and 3 denote mispredictions. However, note that the cost of mispredictions are quite different in the two scenarios (scenario 2 incurs higher latency and scenario 3 extra bandwidth). Table 5 shows the scenario distribution for different predictors averaged across all workloads.

Table 5: Accuracy for Different Predictors

	Serviced by Memory		Serviced by Cache		Overall
Prediction	Memory	Cache	Memory	Cache	Accuracy
SAM	0	51.8%	0	48.1%	48.1%
PAM	51.8%	0	48.2%	0	51.8%
MAP-G	45.1%	6.7%	10.8%	37.4%	82.5%
MAP-I	**48.3%**	3.5%	1.9%	**46.2%**	**94.5%**
Perfect	51.8%	0%	0%	48.2%	100%

PAM almost doubles the memory traffic compared to other approaches (48% of L3 misses are wastefully deemed to access memory when they are in-fact serviced by the DRAM-cache). Compared to a perfect predictor, MAP-I has higher latency for 3.5% of the L3 misses, and extraneous bandwidth consumption for 1.9% of the L3 misses. For the remaining 94.5% of the L3 misses, MAP-I prediction is correct. Thus, even though our predictors are quite simple, low-cost, and low-latency, they are still highly-effective, provide high accuracy, and obtain almost all of the potential for performance improvement from memory access prediction. Unless stated otherwise, the Alloy Cache is always implemented with MAP-I in the remainder of this paper.

5.6. Implications on Memory Power and Energy

Accessing memory in parallel with the cache, as done in PAM and conditionally in DAM, increases power in memory system due to wasteful memory accesses. For PAM, all of the L3 misses would be sent to off-chip memory. Whereas with SAM, only the misses in the DRAM cache would get sent to memory. From Table 5, it can be concluded that PAM would almost double the memory activity compared to SAM. Hence, we do not recommend unregulated use of PAM (except as a reference point). For DAM, our MAP-I predictor is quite accurate which means wasteful parallel accesses account for only 1.9% of L3 misses, compared to 48% with PAM.

6. Analysis and Discussions

6.1. Sensitivity to Cache Size

The default DRAM cache size for all of our studies is 256MB. In this section, we study the impact of different schemes as the cache size is varied from 64MB to 1GB. Figure 9 shows the average speedup with LH-Cache (29-way), SRAM-Tag (32-way), Alloy Cache, and IDEAL-LO. IDEAL-LO is the latency optimized theoretical design that transfers only 64 byte on a cache hit and has perfect zero-latency predictor.

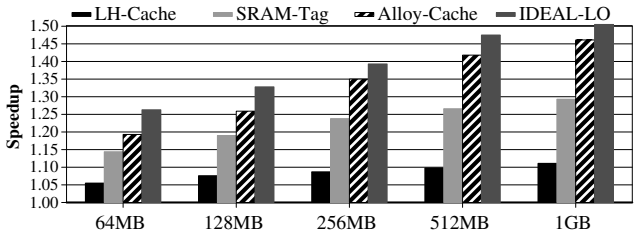


Figure 9: Performance impact across various cache size. Alloy-Cache continues to significantly outperform impractical SRAM-Tag and reaches close to the upperbound of IDEAL-LO.

The SRAM-Tag design suffers from Tag Serialization Latency (TSL). LH-Cache suffers from both TSL and PSL due to the MissMap. Alloy Cache avoids both TSL and PSL, hence it outperforms both the LH-Cache and SRAM-Tag across all studied cache sizes. For the 1GB cache size, LH-Cache provides an average improvement of 11.1%, SRAM-Tag provides 29.3%, and Alloy Cache provides 46.1%. Thus, Alloy Cache provides approximately 1.5 times the improvement of the SRAM-Tag design. Note that the SRAM-Tag implementation incurs an impractical storage overhead of 6MB, 12MB, 24MB, 48MB, and 96MB for DRAM cache sizes of 64MB, 128MB, 256MB, 512MB and 1GB, respectively. Our proposal, on the other hand, requires less than one kilobyte of storage, and still outperforms SRAM-Tag significantly, consistently reaching close to the performance of IDEAL-LO.

6.2. Impact on Hit Latency

The primary reason why the Alloy Cache performs so well is because it is designed from the ground-up to have lower latency. Figure 10 compares the average read latency of LH-Cache, SRAM-Tags, and Alloy Cache. Note that SRAM-Tags incur a tag serialization latency of 24 cycles, and LH-Cache incurs MissMap delay of 24 cycles in addition to the tag serialization latency (32-50 cycles). For the Alloy Cache, there is no tag serialization, except for the one additional bus cycle for obtaining the tag with dataline. The average hit latency for LH-Cache is 107 cycles. The Alloy Cache cuts this latency by 60%, bringing it to 43 cycles. This significant reduction causes Alloy Cache to outperform LH-Cache despite the lower hit rate. The SRAM-Tag incurs an average latency of 67 cycles, hence lower performance than the Alloy Cache.

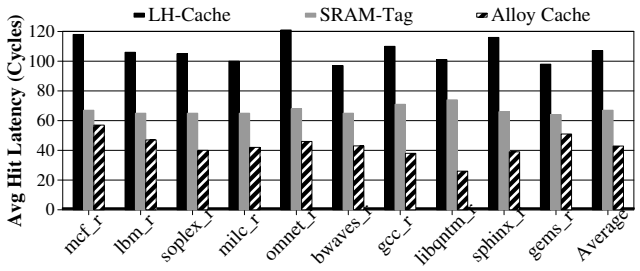


Figure 10: Average Hit-Latency: LH-Cache 107 cycles, SRAM-Tag 67 cycles, and Alloy Cache 43 cycles.

6.3. Impact on Hit-Rate

Our design de-optimizes the cache architecture from a highly-associative structure to a direct-mapped structure in order to reduce hit latency. We compare the hit rate of a highly-associative 29-way LH-cache with the direct-mapped Alloy Cache. Table 6 shows the average hit rate for different cache sizes. For a 256MB cache, the absolute difference in hit rates between the 29-way LH-Cache and direct-mapped Alloy Caches is 7%. Thus, the Alloy Cache increases misses by 15% compared to LH-Cache. However, we show that the 60% reduction in hit latency compared to LH-Cache provides much more performance benefit than a slight performance degradation from the reduced hit rate. Table 6 also shows that the hit-rate difference between a highly-associative cache and a direct-mapped cache reduces as the cache size is increased (at 1GB it is 2.5%, i.e., 5% more misses). The reducing gap between the hit rate of a highly-associative cache and direct-mapped cache as the cache size is increased is well known [6].

Table 6: Hit Rate: Highly associative vs. direct mapped

Cache Size	LH-Cache (29-way)	Alloy-Cache (1-way)	Delta Hit Rate
256 MB	55.2%	48.2%	7.0%
512 MB	59.6%	55.2%	4.4%
1 GB	62.6%	59.1%	2.5%

6.4. Other Workloads

In our detailed studies, we only considered memory-intensive workloads that have a speedup of at least 2x if L3 cache is made perfect (100% hit rate). Figure 11 shows the performance improvement from LH-Cache, SRAM-Tags, and Alloy-Cache for the remaining workloads that spend at least 1% of time in memory. These benchmarks were executed in rate mode as well. The bar labeled Gmean represents the geometric mean improvement over these fourteen workloads.

As the potential is low, the improvements from all designs are lowered compared to the detailed study. However, the broad trend remains the same. On average, LH-Cache improves performance by 3%, SRAM-Tag by 7.3%, and Alloy Cache by 11%. Thus, the Alloy Cache continues to outperform both LH-Cache and SRAM-Tag.

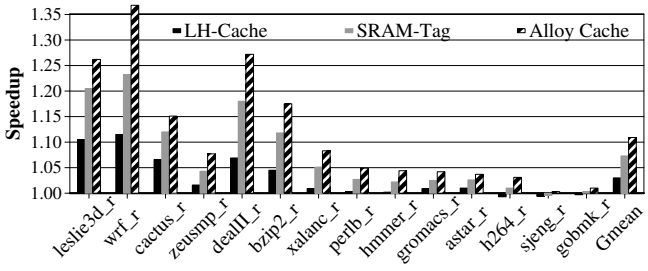


Figure 11: Performance impact for other SPEC workloads

6.5. Impact of Odd Size Burst Length

Our proposal assumes a burst length of five for the Alloy Cache, transferring 80 bytes on each DRAM cache access. However, conventional DDR specifications may restrict the burst length to a power-of-two even for stacked DRAM. If such a restriction exists, then the Alloy Cache can stream out burst of eight transfers (total 128 bytes per access). Our evaluation shows that a design with a burst of 8 provides 33% performance improvement on average, compared to 35% if the burst length can be set to five. Thus, our assumption of odd-size burst length has minimal impact on the performance benefit of Alloy Cache. Note that die-stacked DRAMs will likely use different interfaces than conventional DDR. The larger number of through-silicon vias could make it easier to provide additional control signals to, for example, dynamically specify the amount of data to be transferred.

6.6. Potential Room for Improvement

Our proposal is a simple and practical design that significantly outperforms the impractical SRAM-Tag design, but there is still room for improvement. Table 7 compares the average performance of Alloy Cache + MAP-I, with (a) perfect Memory Address Prediction (Perf-Pred) (b) IDEAL-LO, a configuration that incurs minimum latency and bandwidth and has Perf-Pred and (c) IDEAL-LO with no tag overhead, so all of the 256MB space is available to store data.

Table 7: Room for improvement

Design	Performance Improvement
Alloy Cache + MAP-I	35.0%
Alloy Cache + PerfPred	36.6%
IDEAL-LO	38.4%
IDEAL-LO + NoTagOverhead	41.0%

We observe that for our design we would get 1.6% additional performance improvement from a perfect predictor and another 1.8% from an IDEAL-LO cache. Thus, our practical solution is within 3% of the performance of an idealized design that places tags in DRAM. If we can come up with a way to avoid the storage overhead of tags in DRAM, then there is another 2.6% improvement possible. While all of the three optimizations show small opportunity for improvement,

we must be mindful that solutions to obtain these improvements must incur minimal latency overheads, otherwise the marginal improvements may be quickly negated.

6.7. How About Two-Way Alloy Caches?

We also evaluated two-way Alloy Caches that stream out two TAD entries on each access. While this improved the hit-rate from 48.2% to 49.7%, we found that the hit latency increased from 43 cycles to 48 cycles. This was due to increased burst length (≈2x), associated bandwidth consumption (≈2x), and the reduction in row buffer hit rate. Overall, the performance impact of degraded hit latency outweighs the marginal improvement from hit-rate. We envision that future researchers will look at reducing conflict misses in DRAM caches (and we encourage them to do so); however, we advise them to pay close attention to the impact on hit latency.

7. Conclusion

This paper analyzed the trade-offs in architecting DRAM caches. We compared the performance of a recently-proposed design (LH-Cache) and an impractical SRAM-based Tag-Store (SRAM-Tags) with a latency-optimized design, and show that optimizing for latency provides a much more effective DRAM cache than optimizing simply for hit-rate. To obtain a practical and effective latency-optimized design, this paper went through a three-step process:

1. We showed that simply converting the DRAM cache from high associativity to direct mapped can itself provide good performance improvement. For example, configuring LH-Cache from 29-way to 1-way enhances the performance improvement from 8.7% to 15%. This happens because of the lower latency of a direct-mapped cache as well as the ability to exploit row buffer hits.
2. Simply having a direct-mapped structure is not enough. A cache design that creates a separate "tag-store" and "data-store" still incurs the tag-serialization latency even for direct-mapped caches. To avoid this tag serialization latency, we propose a cache architecture called the *Alloy Cache* that fuses the data and tag together into one storage entity, thus converting two serialized accesses for tag and data into a single unified access. We show that a direct-mapped Alloy Cache improves performance by 21%.
3. The performance of the Alloy Cache can be improved by handling misses faster, i.e., sending them to memory before completing the tag check in the DRAM cache. However, doing so with a MissMap incurs megabytes of storage overhead and tens of cycles of latency, which negated much of the performance benefit of handling misses early. Instead, we present a low-latency (single cycle), low storage overhead (96 bytes per core), highly accurate (95% accuracy) hardware-based *Memory Access Predictor* that enhances the performance benefit of Alloy Cache to 35%.

Optimizing for latency enabled our proposed design to provide better performance than even an impractical option of having the tag store in an SRAM array (24% improvement), which would require tens of megabytes of storage. Thus, we showed that simple designs can be highly effective if they can exploit the constraints of the given technology.

While the technology and constraints of today are quite different from the 1980's, in spirit, the initial part of our work is similar to that of Mark Hill [6] from twenty-five years ago, making a case for direct-mapped caches and showing that they can outperform set-associative caches. Indeed, sometimes *"Big and Dumb is Better"* [1].

Acknowledgments

Thanks to André Seznec and Mark Hill for comments on earlier versions of this paper. Moinuddin Qureshi is supported by NetApp Faculty Fellowship and Intel Early CAREER award.

References

[1] *Quote from Mark Hill's Bio (short link http://tinyurl.com/hillbio):*. https://www.cs.wisc.edu/event/mark-hill-efficiently-enabling-conventional-block-sizes-very-large-die-stacked-dram-caches.
[2] X. Dong, Y. Xie, N. Muralimanohar, and N. P. Jouppi. Simple but Effective Heterogeneous Main Memory with On-Chip Memory Controller Support. In *Supercomputing*, 2010.
[3] M. Farrens, G. Tyson, J. Matthews, and A. R. Pleszkun. A modified approach to data cache management. In *MICRO-28*, 1995.
[4] M. Ghosh and H.-H. S. Lee. Smart Refresh: An Enhanced Memory Controller Design for Reducing Energy in Conventional and 3D Die-Stacked DRAMs. In *MICRO-40*, 2007.
[5] A. Hartstein, V. Srinivasan, T. R. Puzak, and P. G. Emma. Cache miss behavior: is it sqrt(2)? In *Computing Frontiers*, 2006.
[6] M. D. Hill. A case for direct-mapped caches. *IEEE Computer*, Dec 1988.
[7] X. Jiang, N. Madan, L. Zhao, M. Upton, R. Iyer, S. Makineni, D. Newell, Y. Solihin, and R. Balasubramonian. CHOP: Adaptive filter-based dram caching for CMP server platforms. In *HPCA-16*, 2010.
[8] S. M. Khan, D. A. Jiménez, D. Burger, and B. Falsafi. Using dead blocks as a virtual victim cache. In *PACT-19*, 2010.
[9] G. H. Loh and M. D. Hill. Addendum for "Efficiently enabling conventional block sizes for very large die-stacked DRAM caches". http://www.cs.wisc.edu/multifacet/papers/micro11_missmap_addendum.pdf.
[10] G. H. Loh and M. D. Hill. Efficiently enabling conventional block sizes for very large die-stacked DRAM caches. In *MICRO-44*, 2011.
[11] G. H. Loh and M. D. Hill. Supporting very large DRAM caches with compound access scheduling and missmaps. In *IEEE Micro TopPicks*, 2012.
[12] N. Madan, L. Zhao, N. Muralimanohar, A. Udipi, R. Balasubramonian, R. Iyer, S. Makineni, and D. Newell. Optimizing Communication and Capacity in a 3D Stacked Reconfigurable Cache Hierarchy. In *HPCA-15*, 2009.
[13] J. Meza, J. Chang, H. Yoon, O. Mutlu, and P. Ranganathan. Enabling efficient and scalable hybrid memories using fine-granularity dram cache management. *Computer Architecture Letters*, Feb 2012.
[14] E. Perelman et al. Using SimPoint for accurate and efficient simulation. *ACM SIGMETRICS Performance Evaluation Review*, 2003.
[15] M. K. Qureshi. Memory access prediction. U.S. Patent Application Number 12700043, Filed Feb 2010, Publication Aug 2011.
[16] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely Jr., and J. Emer. Adaptive insertion policies for high-performance caching. In *ISCA-34*, pages 167–178, 2007.
[17] A. Seznec and P. Michaud. A case for (partially) tagged geometric history length branch prediction. In *Journal of Instruction Level Parallelism*, 2006.
[18] C.-J. Wu, A. Jaleel, W. Hasenplaugh, M. Martonosi, S. C. Steely, Jr., and J. Emer. Ship: signature-based hit predictor for high performance caching. In *MICRO-44*, 2011.
[19] L. Zhao, R. Iyer, R. Illikkal, and D. Newell. Exploring DRAM cache architectures for CMP server platforms. In *ICCD*, 2007.

2012 IEEE/ACM 45th Annual International Symposium on Microarchitecture

A Mostly-Clean DRAM Cache for Effective Hit Speculation and Self-Balancing Dispatch

Jaewoong Sim* Gabriel H. Loh† Hyesoon Kim⋆ Mike O'Connor† Mithuna Thottethodi‡

*School of ECE ⋆College of Computing
Georgia Institute of Technology
{jaewoong.sim,hyesoon.kim}@gatech.edu

†AMD Research
Advanced Micro Devices, Inc.
{gabe.loh,mike.oconnor}@amd.com

‡School of ECE
Purdue University
mithuna@purdue.edu

Abstract

Die-stacking technology allows conventional DRAM to be integrated with processors. While numerous opportunities to make use of such stacked DRAM exist, one promising way is to use it as a large cache. Although previous studies show that DRAM caches can deliver performance benefits, there remain inefficiencies as well as significant hardware costs for auxiliary structures. This paper presents two innovations that exploit the bursty nature of memory requests to streamline the DRAM cache. The first is a low-cost Hit-Miss Predictor *(HMP) that virtually eliminates the hardware overhead of the previously proposed multi-megabyte MissMap structure. The second is a* Self-Balancing Dispatch *(SBD) mechanism that dynamically sends some requests to the off-chip memory even though the request may have hit in the die-stacked DRAM cache. This makes effective use of otherwise idle off-chip bandwidth when the DRAM cache is servicing a burst of cache hits. These techniques, however, are hampered by dirty (modified) data in the DRAM cache. To ensure correctness in the presence of dirty data in the cache, the HMP must verify that a block predicted as a miss is not actually present, otherwise the dirty block must be provided. This verification process can add latency, especially when DRAM cache banks are busy. In a similar vein, SBD cannot redirect requests to off-chip memory when a dirty copy of the block exists in the DRAM cache. To relax these constraints, we introduce a hybrid write policy for the cache that simultaneously supports write-through and write-back policies for different pages. Only a limited number of pages are permitted to operate in a write-back mode at one time, thereby bounding the amount of dirty data in the DRAM cache. By keeping the majority of the DRAM cache clean, most HMP predictions do not need to be verified, and the self-balancing dispatch has more opportunities to redistribute requests (i.e., only requests to the limited number of dirty pages must go to the DRAM cache to maintain correctness). Our proposed techniques improve performance compared to the MissMap-based DRAM cache approach while simultaneously eliminating the costly MissMap structure.*

1. Introduction

Advances in die-stacking technologies have made it possible to integrate hundreds of megabytes, or even several gigabytes, of DRAM within the same package as a multi-core processor [1, 8, 11, 20] or a GPU [19]. To avoid dependencies on operating system vendors, maintain software transparency, and provide benefit to legacy software, recent papers have suggested that using die-stacked DRAM as a large cache is a compelling approach [11, 12].

To reduce the overhead of supporting the tags in large, die-stacked DRAM caches, recent work has considered embedding the tags directly alongside the data within the DRAM array, which avoids the need for a dedicated external SRAM tag array (e.g., 96MB for a 1GB DRAM cache) [4, 11]. Placement of the tags within the die-stacked DRAM itself incurs a costly DRAM access even in the case where the request eventually misses in the cache. Loh and Hill proposed the MissMap, a multi-megabyte structure (much less overhead than a dedicated SRAM tag array) that allows the DRAM cache controller to skip the DRAM cache access when there is a cache miss [11].

While the MissMap provides a far more practical approach than using a massive SRAM tag array, its implementation cost is still likely to be prohibitively high to allow it to be deployed in commercial products (e.g., 4MB for a 1GB DRAM cache). Furthermore, the access latency of the MissMap is not trivial (the original paper used a latency of 24 cycles, which is added to *all* DRAM cache hits *and* misses). In this work, we point out that the MissMap approach is overly conservative (i.e., maintaining precise information about the DRAM cache's contents is not necessary) and that it is actually possible to *speculate* on whether a request can be served by the DRAM cache or main memory. We introduce a light-weight, low-latency *Hit-Miss Predictor* (HMP) that provides 97% accuracy on average, with a hardware cost of less than 1KB.

We also propose a self-balancing dispatch (SBD) mechanism that dynamically steers memory requests to either the die-stacked DRAM cache or to the off-chip main memory depending on the instantaneous queuing delays at the two memories. While the stacked DRAM can provide higher bandwidth than the off-chip memory, overall system bandwidth would be greater yet if *both* die-stacked and off-chip memories could be efficiently exploited at the same time.

While the HMP and SBD techniques can potentially streamline the design of a DRAM cache, these approaches are only useful if they can still ensure correct execution. The source of potential complication comes from dirty/modified data in the DRAM cache. Both the HMP and SBD can potentially send a request to main memory when the DRAM cache contains the most-recent, modified value. Returning the stale value from off-chip memory could then lead to incorrect program execution. Beyond a basic mechanism to validate predictions, we also introduce a hybrid write policy that forces the majority of the DRAM cache to operate in a write-through mode, and only enables write-back for a limited set of pages that have high write traffic. This results in a DRAM cache that is mostly clean, thereby allowing the DRAM cache to avoid waiting on HMP prediction verification and creates more opportunities for SBD to freely send requests off-chip.

2. Background

2.1. DRAM Architectures

Most conventional caches are implemented with SRAM technology, whereas this work considers die-stacked DRAM. DRAM consists of arrays of bit-cells, where each bit-cell is comprised of a capacitor to store charge and an access transistor to enable reading/writing of the cell. Accessing bit-cells in a DRAM requires storing the bit-cell values in a row buffer, and all read and write operations effectively

1072-4451/12 $26.00 © 2012 IEEE
DOI 10.1109/MICRO.2012.31

IEEE computer society

operate directly on the row buffer (rather than the actual bit-cells). When, the row is no longer needed (or often when a different row is requested), the contents of the row buffer are written back into the original row of bit-cells and then a new row may be accessed.

The DRAM access mechanism is quite different from SRAM arrays. In the case of DRAM, an entire bank is occupied while the row is open, and therefore any requests to other rows in this bank will be delayed until the current operations complete (although operations in independent banks may proceed concurrently subject to DRAM timing and bus constraints). In an SRAM, the access paths are more easily pipelined, and so even if a request has been sent to a particular bank, subsequent requests need only wait a few cycles before they can proceed.

2.2. Die-stacked DRAM Caches

Caches store two types of information: tags and data. In conventional SRAM-based caches, these are stored in two physically distinct structures (the tag and data arrays, respectively). For a DRAM cache, one could consider an SRAM-based tag array, as shown in Figure 1(a), but previous estimates have shown that such a structure would require tens of megabytes of SRAM, and therefore this approach is not practical considering that current L3 cache sizes are typically only around 8 MB [2, 3].

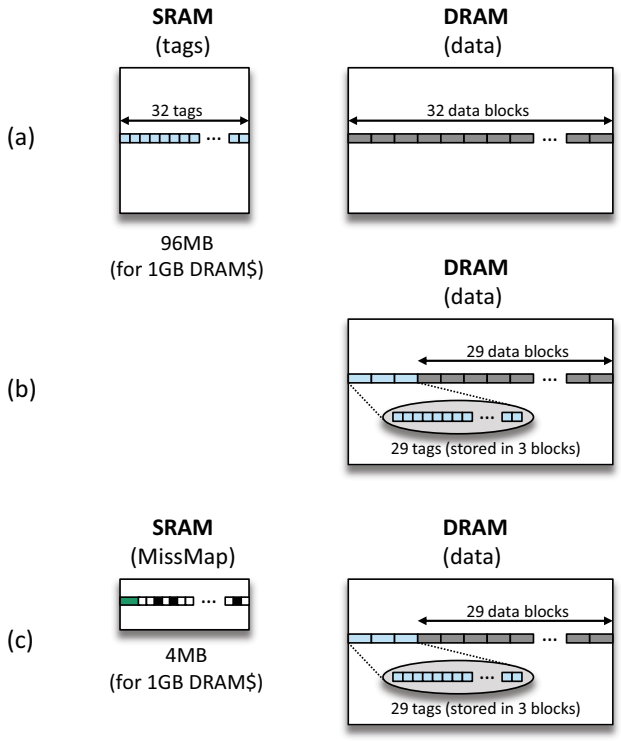


Figure 1: DRAM cache organizations using (a) an SRAM tag array, (b) tags embedded in the DRAM, and (c) tags in DRAM with a MissMap.

Instead, recent research has considered organizations where the tags and data are directly co-located within the die-stacked DRAM, as shown in Figure 1(b) [4, 11]. While this eliminates the unwieldy SRAM tag array, it introduces two more problems. First, naively accessing the cache could double the DRAM cache latency: one access to read the tags, followed by another access to read the data (on a hit). Second, for cache misses, the cost of the DRAM cache tag access is added to the overall load-to-use latency.

Loh and Hill observed that the tags and data reside in the same DRAM row, and so the actual latency of a cache hit can be less than two full accesses by exploiting row buffer locality [11]. That is, the DRAM row is opened or activated into the row buffer only once, and then tag and data requests can be served directly out of the row buffer at a lower latency compared to two back-to-back accesses to different rows. They also proposed a hardware data structure called a MissMap that precisely tracks the contents of the DRAM cache, as shown in Figure 1(c). Before accessing the DRAM cache, the MissMap is first consulted to determine whether the requested cache block is even resident in the cache. If the block is not in the cache (i.e., miss), the request can be sent directly to main memory without incurring the tag-check cost of the DRAM cache. For a 512MB DRAM cache, the MissMap needs to be about 2MB in size (which provides tracking of up to 640MB of data), and a 1GB cache would need a 4MB MissMap. While Loh and Hill argue that part of the L3 cache could be carved out to implement the MissMap, using the AMD Opteron^{TM} processor that consumes 1MB of its L3 to implement a "Probe Filter" as an example [2], it seems unlikely that designers would be willing to sacrifice *half* of their L3 to implement the MissMap.[1]

3. Motivation

In this section, we identify inefficiencies with the previously proposed DRAM cache organizations. First, we explain why the MissMap is overly conservative, which ultimately leads us to consider more speculative techniques with significantly lower overheads (both in terms of hardware cost and latency). Second, we describe scenarios where a conventional cache organization under-utilizes the available aggregate system bandwidth, which motivates our proposal for a *Self-Balancing Dispatch* mechanism. Third, we discuss how the presence of dirty/modified data in the DRAM cache can potentially limit how aggressively we can speculate on or rebalance DRAM cache requests.

3.1. The Overkill of the MissMap

The MissMap tracks memory at page (or other coarse-grain) granularity. Each MissMap entry consists of a tag that stores the physical page number, and a bit-vector that records which cache blocks from this page are currently resident in the DRAM cache. The bit vector is precisely maintained such that each time a new cache block is inserted, its corresponding bit in the vector will be set; conversely, when a cache block is evicted, its bit will be cleared. Furthermore, if a MissMap entry is evicted, then *all* dirty lines from the corresponding victim page must also be evicted and written back.

Loh and Hill mentioned that it is possible to allow the MissMap to have false positives [11]. That is, if the MissMap says that a block is present in the DRAM cache when in fact it is not, then there is only a performance impact as the system needlessly pays for the latency of the DRAM cache before going to main memory. However, if the MissMap reports that a line is *not* present when in fact it is (false negative), the request would be sent to main memory and returned to the processor. If the DRAM cache contains this block in a dirty state, then this can lead to incorrect program execution.

On a DRAM cache miss (whether the MissMap said so or not), the system sends the request to main memory. When the response

[1] Assuming a 4MB MissMap to support a 1GB DRAM cache and a baseline L3 cache size of 8MB. If such a system employed a Probe Filter as well, then only 3MB out of the original 8MB L3 would actually be available as a cache!

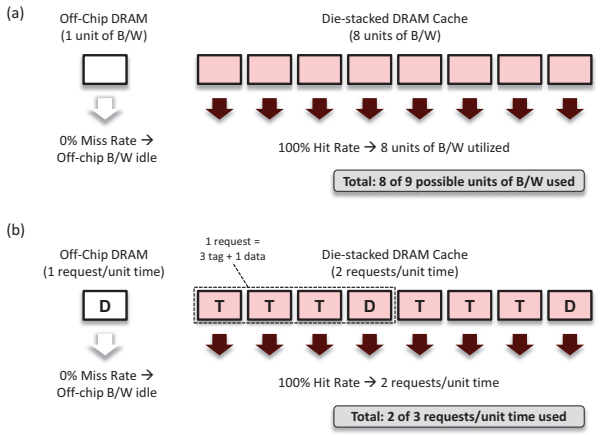


Figure 2: Example scenario illustrating under-utilized off-chip memory bandwidth in the presence of very high DRAM cache hit rates when considering (a) raw bandwidth in Gbps, and (b) in terms of request service bandwidth.

returns, the data are sent back to the L3 and the processor, and the data are also installed into the DRAM cache.[2] Prior to the installation of a new cache block, a victim must be selected. Furthermore, if the victim has been modified, then it must also be written back to main memory.

Note that when selecting a victim, the DRAM tags are checked. Therefore, if the system issued a request to main memory even though a modified copy of the block is in the DRAM cache, this can still be detected at the time of victim selection. Given this observation, the constraint that the MissMap *must* not allow false negatives is overly conservative. False negatives are tolerable so long as responses from memory are not sent back to the processor before having verified that a dirty copy does not exist in the DRAM cache.

Based on these observations, we propose a DRAM cache organization that can speculatively issue requests directly to main memory regardless of whether the decisions are "correct" or not. Section 4 describes a predictor design that exploits spatial correlation and the bursty nature of cache traffic to provide a light-weight yet highly accurate DRAM cache hit-miss predictor.

3.2. Under-utilization of Aggregate System Bandwidth

Die-stacked DRAM can potentially provide a substantial increase in memory bandwidth. Previous studies have assumed improvements in latency of 2×, 3× and as much as 4× compared to conventional off-chip DRAM [8, 11, 20]. At the same time, the clock speed can be faster, bus widths wider, and independent channels more numerous [9, 11]. Even with a rough estimate of half the latency, twice the channels, and double-width buses (compared to conventional off-chip memory interfaces), the stacked DRAM would provide an 8× improvement in bandwidth. In an "ideal" case of a DRAM cache with a 100% hit rate, the memory system could provide an eight-fold increase in delivered bandwidth, as shown in Figure 2(a). However, the off-chip memory is completely idle in this scenario, and that represents 11% ($\frac{1}{1+8}$) of the overall system bandwidth that is being wasted.

Figure 2(b) shows the same scenario again, but instead of raw bandwidth (in terms of Gbps), we show the *effective bandwidth* in terms of requests serviced per unit time. Note that a request to main memory only requires transferring a single 64B cache block, whereas a request to a tags-in-DRAM cache requires transferring three tag blocks (64B each) and finally the data block. Therefore, the sustainable effective bandwidth of the DRAM cache is only twice that of the off-chip memory (8× the raw bandwidth, but 4× the bandwidth-consumption per request). In this case, a 100%-hit rate DRAM cache would leave 33% of the overall effective bandwidth unused ($\frac{1}{1+2}$). While the DRAM cache typically does not provide a 100% hit rate, hits often come in bursts that can lead to substantial queuing delays from bank and bus contention.

Apart from the available bandwidth, bank and bus conflicts at the DRAM cache can lead to increased queuing delays, some of which could potentially be mitigated if some of these requests could be diverted to the off-chip memory. In practice, other timing constraints, resource conflicts, and specific access patterns and arrival rates would affect the exact amount of bandwidth available for both the DRAM cache and the off-chip memory. However, this simple example highlights that there will be times where the system will have some idle resources, and we propose a Self-Balancing Dispatch technique to capitalize on these resources.

3.3. Obstacles Imposed by Dirty Data

Dirty data in the DRAM cache can severely restrict the aggressiveness of speculatively sending requests to main memory, as the copy in main memory is stale and its usage can result in incorrect executions. Likewise, dirty data prevents the system from exploiting idle main-memory bandwidth because accesses to dirty data must be sent to the DRAM cache regardless of how busy the DRAM cache or how idle the off-chip memory is. This also raises the question as to how the system can know ahead of time that a request targets a dirty cache line without having first looked up in the cache to see if the line is present and dirty. A key contribution of this work is a new way to operate the DRAM cache (which could be applied to other types of caches) such that most of the cache will be clean, and for the majority of the cache, we can guarantee its cleanliness without having to check the cache's tags. This removes major limitations for both cache hit speculation and Self-Balancing Dispatch.

4. DRAM Cache Hit Speculation

The previously proposed MissMap provides precise tracking of DRAM cache contents, but as a result, the size (2-4MB) and latency (tens of cycles) of the structure introduce significant overheads. Section 3 explained how the DRAM cache can check for the existence of a dirty block at the time of a cache fill, and how this allows the DRAM cache to speculatively send requests to main memory so long as we ensure that the data are not returned to the processor until it has been verified that a modified copy is not also in the DRAM cache. In this section, we present the designs for lightweight and accurate region-based predictors that exploit the bursty nature of cache hits and misses [10].

4.1. Region-based Hit/Miss Prediction

Our region-based Hit/Miss Predictor (HMP_{region}) is structurally similar to a classic bimodal branch predictor [15]. The predictor itself consists of a table of two-bit saturating counters. For a DRAM cache with millions of cache blocks, it is not practical to directly index into the HMP_{region} table with a hash of the raw physical address; the aliasing and interference would render the predictor table nearly

[2] For this study, we assume that all misses are installed into the DRAM cache. Other policies are possible (e.g., write-no-allocate, victim-caching organizations), but these are not considered here.

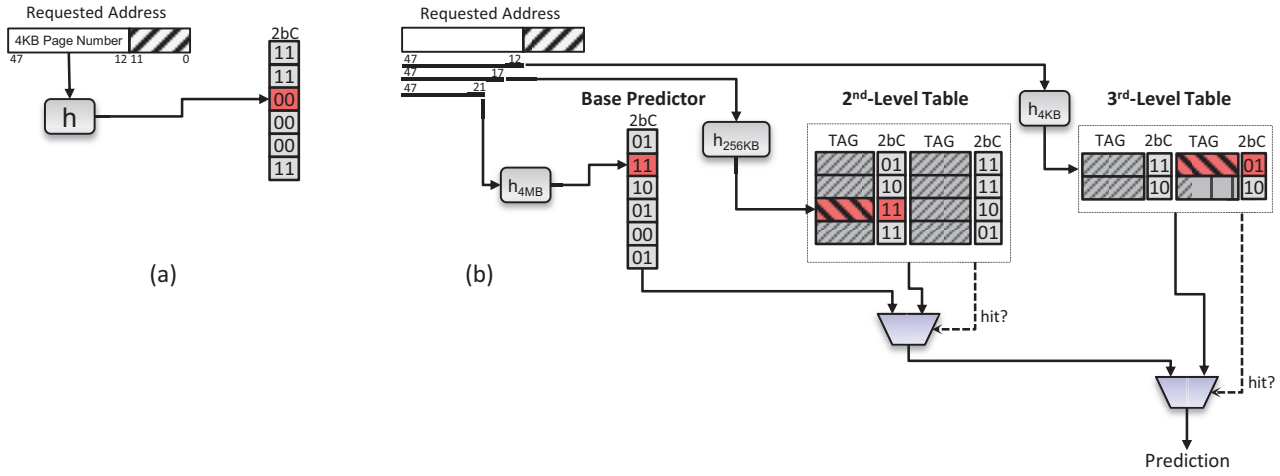


Figure 3: Hit-Miss Predictor designs: (a) one-level HMP_{region}, (b) multi-granular HMP_{MG}.

useless, or a gigantic table would be needed. Instead, we break up the memory space into coarser-grained regions (e.g., 4KB), and index into the HMP_{region} with a hash of the region's base address as shown in Figure 3(a). This allows the HMP_{region} table to be much smaller, but it also means that *all* accesses within a region will follow the same prediction. The operation of the HMP_{region} is otherwise analogous to the bimodal predictor: DRAM cache hits increment the predictor, and misses decrement the predictor (saturating at 3 or 0, respectively).

The coarse-grained predictor organization of HMP_{region} is actually a benefit rather than a shortcoming. Accesses tend to exhibit significant spatial locality, especially at lower-level caches such as a large DRAM cache. Figure 4(a) shows the number of cache blocks present in the DRAM cache for one particular 4KB page of leslie3d (from the WL-6 workload) with respect to the number of accesses to this page (our methodology is explained in Section 7). Initially, nothing from this page is in the DRAM cache, but as the page is used, more and more lines are installed. During this installation phase, most accesses result in cache misses, and a simple 2-bit counter corresponding to this region would mostly predict that these requests result in misses. After this "warm up" phase, the footprint from this region is stable, and all subsequent accesses to this region result in hits. Again, a simple 2-bit counter would quickly switch over to predicting "cache hit" for this region and achieve high accuracy. When the application is finished with using this region, the contents will gradually get evicted from the cache, as shown by the drop back down to zero. At some future point,[3] the page becomes hot again and the process repeats.

Figure 4(b) shows another 4KB region taken from leslie3d (from the same workload WL-6). This is just to illustrate that different regions and different applications may show different types of patterns, but so long as there exist sustained intervals where the curve is consistently increasing (mostly misses) or is consistently flat (mostly hits), then the simple 2-bit counter approach will be effective for making hit-miss predictions for the region.

The HMP_{region} approach is different from other previously proposed history-based hit-miss predictors. Past work has considered hit-miss predictors for L1 caches based on PC-indexed predictor organizations [18]; such an approach may not be as easy as to implement for a DRAM cache because PC information is not typically passed down the cache hierarchy, may not exist for some types of requests (e.g., those originating from a hardware prefetcher), or may not be well defined (e.g., a dirty cache block being written back to the DRAM cache that was modified by two or more store instructions may have multiple PCs associated with it).

[3]The figure's x-axis is based on accesses to the page. The time that elapses from the last access in the hit phase until the first access in the miss phase could easily span many thousands or even millions of cycles, but this all gets compressed here.

4.2. Multi-Granular Hit-Miss Predictor

The HMP_{region} predictor requires approximately one two-bit counter per region. For example, assuming a system with 8GB of physical memory and a region size of 4KB, the HMP_{region} would still need 2^{21} two-bit counters for a total cost of 512KB of storage. While this is already less than a 2-4MB MissMap, there is still room to further optimize.

We observed that even across large contiguous regions of memory spanning multiple physical pages, the hit-miss patterns generally remained fairly stable (that is, sub-regions often have the same hit-miss bias as other nearby sub-regions). While, in theory, different nearby physical pages may have nothing to do with each other (e.g., they may be allocated to completely independent applications), in practice memory allocation techniques such as page-coloring [17] tend to increase the spatial correlation across nearby physical pages. In our experiments, we found that memory would often contain large regions with *mostly* homogeneous hit/miss behavior, but smaller pockets within the larger regions would behave differently.

We propose a *Multi-Granular Hit/Miss Predictor* (HMP_{MG}) that is structurally inspired by the TAGE branch predictor [13], but operates on the base addresses of different memory regions (as opposed to branch addresses) and the different tables capture hit-miss patterns corresponding to different region sizes (as opposed to branch history lengths). Figure 3(b) shows the hardware organization of HMP_{MG}. The first-level predictor is similar to HMP_{region}, except that it makes predictions over very large 4MB regions. The second and third-level tables consist of set-associative tagged-structures that make predictions on finer-grained 256KB and 4KB region sizes, respectively. Each entry in the tagged tables consists of a (partial) tag and a two-bit counter for prediction. Tag hits in the tagged HMP_{MG} tables will override predictions from larger-granularity predictor tables.

The overall structure of the HMP_{MG} provides a more efficient

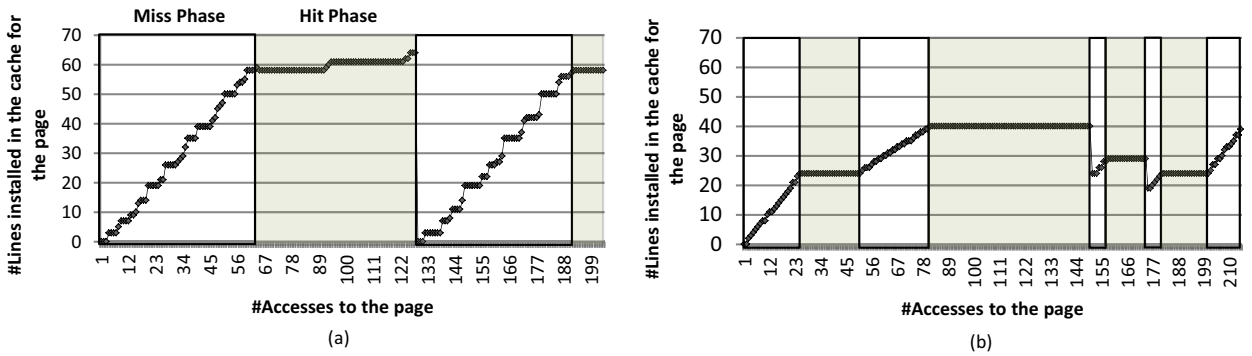


Figure 4: Hit and miss phases for two example pages from leslie3d (when run as part of the multi-programmed workload WL-6).

and compact predictor organization. A single two-bit counter in the first-level table covers a memory range of 4MB. In the single-level HMP_{region} predictor, this would require 1024 counters to cover the same amount of memory.

4.3. Predictor Operation

The entries in the HMP_{MG}'s base predictor are initially set to weakly miss or "1". To make a hit/miss prediction for a request, the base and tagged components are all looked up simultaneously. The base component makes a default prediction, while tagged components provide an overriding prediction on a tag hit.

The HMP_{MG} is updated when it has been determined whether there was a DRAM cache hit or not. The 2-bit counter of the "provider" component is always updated.[4] On a misprediction, an entry from the "next" table is allocated (a victim is chosen based on LRU). For example, if the prediction came from the first-level table, then a new entry will be allocated in the second-level table. Mispredictions provided by the third table simply result in the corresponding counter being updated without any other allocations. The newly allocated entry's 2-bit counter is initialized to the weak state corresponding to the actual outcome (e.g., if there was a DRAM cache hit, then the counter is set to "weakly hit" or 2).

4.4. Implementation Cost

Table 1 shows the storage overhead of the HMP_{MG} configuration used in this paper. Compared to a MissMap that requires 2-4MB of storage, the HMP_{MG} only requires 624 bytes of total storage. A single predictor is shared among all cores. At this size, the entire L3 cache can once again be used for caching (as opposed to implementing a MissMap). Also important, the small size of the HMP_{MG} allows it to be accessed in a single cycle as it is smaller than even many branch predictors. Compared to the 24-cycle latency assumed for the MissMap [11], this provides significant benefits both for performance and implementability.

Hardware	Size
Base Predictor (4MB region)	1024 entries * 2-bit counter = 256B
2nd-level Table (256KB region)	32 sets * 4-way * (2-bit LRU + 9-bit tag + 2-bit counter) = 208B
3rd-level Table (4KB region)	16 sets * 4-way * (2-bit LRU + 16-bit tag + 2-bit counter) = 160B
Total	**624B**

Table 1: Hardware cost of the Multi-Granular Hit-Miss Predictor.

[4]The "provider" is the terminology used for the TAGE predictor to indicate the table from which the final prediction came from.

5. Exploiting Unused Bandwidth

As described in Section 3, there are scenarios where a burst of DRAM cache hits (or predicted hits for that matter) can induce significant DRAM cache bank contention while the off-chip memory remains largely idle. In this section, we describe a *Self-Balancing Dispatch* (SBD) mechanism that allows the system to dynamically choose whether (some) requests should be serviced by the DRAM cache or by the off-chip memory.

In an ideal case, every request could be routed to either the DRAM cache or to off-chip memory. If both memories had the same latency per access, then the system could simply look at the number of requests already enqueued for each and send the request to the one with fewer requests. However, the different memories have different latencies, and so the request should be routed to the source that has the lowest *expected* latency or queuing time. The expected latency for each memory can be estimated by taking the number of requests already "in line" and then multiplying by the average or typical access latency for a single request at that memory. Overall, if one memory source is being under-utilized, then it will tend to have a lower expected latency and the SBD mechanism will start directing requests to this resource. In the steady-state, the bandwidth from both sources will be effectively put to use.

Complications arise due to the fact that not every request can or should be freely routed to whichever memory has the lowest expected latency. If a request is for a dirty block in the DRAM cache, then routing the request to the off-chip memory is of no use (in fact, it just wastes bandwidth) because the data must ultimately come from the DRAM cache. If the HMP predicts that a request will miss in the DRAM cache, then there is likely little benefit in routing it to the DRAM cache (even if it has a lower expected latency), because if the prediction is correct, there will be a cache miss which in the end simply adds more latency to the request's overall service time.

The above constraints mean that SBD can only be gainfully employed for requests that would have hit in the DRAM cache where the corresponding cache block is not dirty. To determine whether a request will (likely) hit in the DRAM cache, we simply rely on the HMP. While the HMP is not perfectly accurate, mispredictions simply result in lost opportunities for SBD to make better use of the available bandwidth. To deal with dirty data, we will first simply assume that the DRAM cache makes use of a write-through policy to ensure that all blocks are always clean. Algorithm 1 below describes the basic SBD algorithm assuming a write-through cache. In the next section, we will show how to remove the strict write-through requirement to avoid the unnecessary write traffic to main memory.

Note in Algorithm 1, we do not count *all* of the requests that are

Algorithm 1 Self-Balancing Dispatch

0) Self-balancing dispatch operates only on (predicted) hit requests.
1) $N_{\text{Off-Chip}}$:= Number of requests already waiting *for the same bank* in the off-chip memory.
2) $L_{\text{Off-Chip}}$:= Typical latency of one off-chip memory request, excluding queuing delays.
3) $E_{\text{Off-Chip}} := N_{\text{Off-Chip}} * L_{\text{Off-Chip}}$. (Total expected queuing delay if this request went off-chip.)
4-6) $N_{\text{DRAM\_Cache}}$, $L_{\text{DRAM\_Cache}}$, $E_{\text{DRAM\_Cache}}$ are similarly defined, but for the die-stacked DRAM cache.
7) If $E_{\text{Off-Chip}} < E_{\text{DRAM\_Cache}}$, **then** send the request to off-chip memory; **else** send to DRAM cache.

waiting to access off-chip memory, but we limit the count to those waiting on the same bank as the current request that is under SBD consideration (similar for the number of requests to the target off-chip DRAM cache bank). The above description uses the "typical" latency (e.g., for main memory we assume the latency for a row activation, a read delay (tCAS), the data transfer, and off-chip interconnect overheads; for the DRAM cache we assume a row activation, a read delay, three tag transfers, another read delay, and then the final data transfer). Other values could be used, such as dynamically monitoring the actual average latency of requests, but we found that simple constant weights worked well enough. Note also that these latency estimates only need to be close enough relative to each other; slight differences in the estimated expected latency and the actual observed latency do not matter if they do not lead to different SBD outcomes (i.e., an error of a few cycles will in most cases not cause the SBD mechanism to change its decision).

6. Maintaining a Mostly-Clean Cache

When a request is for a cached dirty block, the SBD mechanism has no choice but to send the request to the DRAM cache (it is possible that the HMP mispredicted it as a miss, but this would ultimately be detected and requires reading the data from the DRAM cache anyway). If the system could *guarantee* that a requested block is not cached *and* dirty, then SBD could more freely make bandwidth-balancing decisions with its effectiveness only constrained by the accuracy of the HMP.

6.1. Write-Through vs. Write-Back

We earlier discussed how employing a write-through policy for the DRAM cache can in fact ensure that all requests that hit in the cache are *not* for dirty blocks, but applying a write-through policy wholesale to the entire DRAM cache can result in significant increases in write-through traffic to main memory. Figure 5(a) shows the top most-written-to pages in the DRAM cache for the SPEC2006 benchmark soplex. The upper curve (dotted) shows the write traffic for a write-through policy, and the lower curve (solid) shows the write traffic for a write-back policy. The large differences between the curves indicate that the write-back policy achieves significant write-combining, and therefore employing a write-through policy could significantly increase write traffic to main memory. There are other scenarios, such as that shown in 5(b), where, even in a write-back cache, dirty lines are usually only written to once before they are subsequently evicted. However, on average across all of our workloads, we observed that a write-through DRAM cache results in ∼3.7× greater write traffic to main memory than a write-back policy (although the amount varies significantly based on the exact workloads).

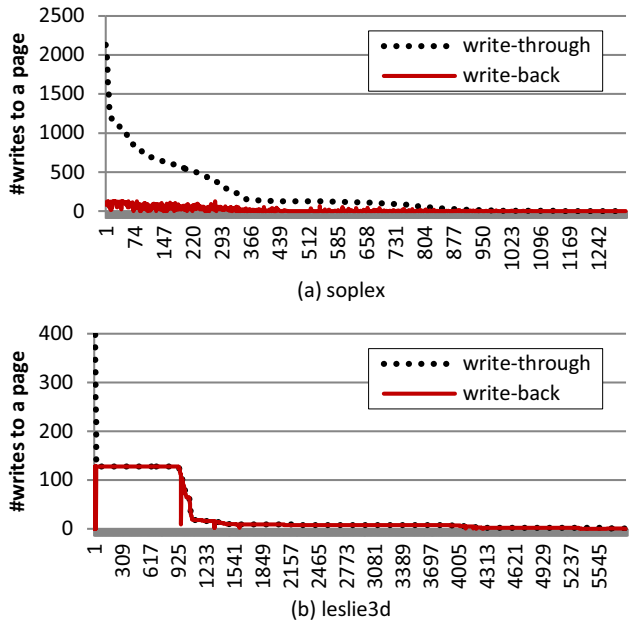


Figure 5: Number of writes for each page with write-through and write-back policy. The x-axis is sorted by top most-written-to pages.

Another important statistic is that, on average for our experiments, only about 5% of an application's pages ever get written to. This indicates that in typical scenarios, the vast majority of the DRAM cache's blocks are in fact clean. A write-through cache ensures cleanliness, but costs significantly more main-memory write traffic. A write-back cache minimizes off-chip write traffic, but then cannot provide any guarantees of cleanliness despite the fact that most blocks will in fact be clean.

6.2. The Dirty Region Tracker

We propose a hybrid write-policy for the DRAM cache where, by default, pages employ a write-through policy (to handle the common case of clean pages), but a limited subset of the most write-intensive pages are operated in write-back mode to keep the main-memory write traffic under control. To support this hybrid policy, we introduce the *Dirty Region Tracker* (DiRT). The DiRT consists of two primary components as shown in Figure 6. The first structure is a counting Bloom filter (CBF) that is used to approximately track the number of writes to different pages. On each write, the page address is hashed differently for each of the CBF tables, and the corresponding counters are incremented.[5] When a page's counters in *all three* CBFs exceed a threshold, then it is determined to be a write-intensive page (and each indexed CBF counter is reduced by half).

At this point, we introduce the second structure that is a *Dirty List* of all pages that are currently operated with a write-back policy. The Dirty List is a set-associative tagged structure where each entry consists of a tag to store a physical page number and 1 bit of storage to implement a not-recently-used (NRU) replacement policy. A page not currently in the Dirty List, but whose counters have exceeded the threshold, gets inserted into the Dirty List (and the NRU entry from the Dirty List is evicted). Note that when a page is evicted from the Dirty List, its write policy is switched back to write-through; at

[5] We use three CBFs with different hash functions, which increases the efficacy of identifying the most write-intensive pages due to the reduction in aliasing.

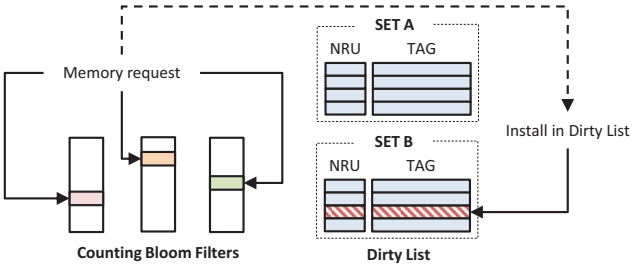


Figure 6: Dirty Region Tracker (DiRT).

this point, the system must ensure that any remaining dirty blocks from this page are written back to main memory. At first blush, this may seem like a high overhead, but a 4KB page only contains 64 cache blocks. Current die-stacked DRAMs already support 32 banks (e.g., 4 channels at 8 banks each [9]), and so the latency overhead is only two activations per bank (and the activations across banks can be parallelized) plus the time to stream out the data back to main memory. Note that all of these cache blocks will have very high spatial locality because they are all from the same page, so practically all of the writeback traffic will experience row buffer hits in main memory. Also, any clean blocks of course need not be written back. The detailed algorithm for DiRT management is listed in Algorithm 2.

Algorithm 2 DiRT Management

1) Check Dirty List for the written-to page; if it's there, update NRU replacement meta-data.
2) If not, increment each indexed counter in all three CBFs.
3) If all indexed counters are greater than threshold:
  a) Evict the NRU entry from the Dirty List;
    writeback any associated dirty blocks.
  b) Allocate the new page to the Dirty List.
  c) Reduce each indexed CBF counter by half.

6.3. Putting the DiRT to Work

6.3.1. Streamlining HMP: The DiRT works synergistically with Hit-Miss Prediction. In parallel with the HMP lookup, a request can also check the DiRT to see if it accesses a guaranteed clean page. If the page is clean (i.e., not currently in the Dirty List), then requests that are predicted misses can be issued directly to main memory. When the value is returned, this data can be forwarded directly back to the processor without having to verify whether there was actually a dirty copy of the block in the DRAM cache because the DiRT has already guaranteed the block to be clean. Without the DiRT, all returned predicted-miss requests must stall at the DRAM cache controller until the fill-time speculation has been verified. During times of high bank contention, this prediction-verification latency can be quite substantial.

6.3.2. Streamlining SBD: When combining the DiRT with the SBD mechanism, the DiRT can guarantee that accesses to certain (most) pages will be clean, and so SBD can freely choose the best memory source to route the request to. When the HMP predicts a hit, the system first consults the DiRT's Dirty List. If the requested page is found in the Dirty List, then we do not know if the requested *block* is dirty or not (e.g., it could be one of the few clean blocks in a mostly-dirty page). In this case, SBD always routes the request to the DRAM cache. However, if the requested page is not in the Dirty List, then the page (and therefore the requested block) is *guaranteed* to be clean, and therefore SBD can do as it wishes. Note that clean pages are the overwhelming common case (expect for a few benchmarks), and so using the DiRT provides SBD with many more opportunities to make use of otherwise under-utilized off-chip bandwidth.

6.4. Putting It All Together

Figure 7 shows the decision flow chart for memory requests with all of the proposed mechanisms. One should note that Hit-Miss Prediction, SBD, and the DiRT can all be accessed in parallel (SBD can speculatively make a decision assuming an access to a clean, predicted-hit block). Furthermore, HMP and DiRT lookups could even be initiated early before the L2 hit/miss status is known as these components only require the requested physical address.

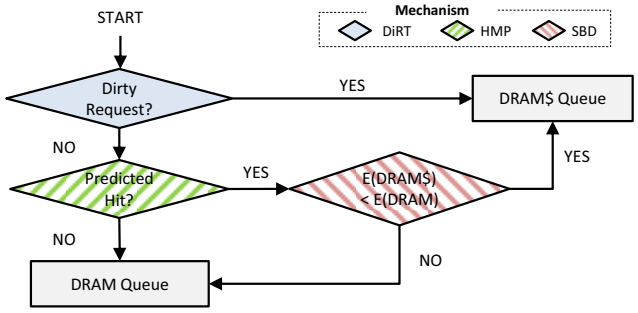


Figure 7: Decision flow chart for memory requests.

6.5. DiRT Implementation Cost

The DiRT is a slightly larger structure compared to the simple hit-miss predictors, but the overall hardware cost is still quite manageable (6.5KB, just 0.16% of our L2 data array size). Each of the three CBF tables has 1024 entries, and each entry consists of a five-bit saturating counter. We use a threshold of 16 writes to consider a page as write-intensive. For Dirty List, we use a 4-way set associative structure with 256 sets, so it supports up to 1024 pages operating in write-back mode at a time. Each entry of the Dirty List consists of 1-bit reference information for NRU replacement policy and a tag for the page. Other approximations (e.g., pseudo-LRU, SRRIP [7]) or even true LRU (this only requires 2-bit for a 4-way set associative structure) could also be used for the replacement policy, but a simple NRU policy worked well enough for our evaluations. In Section 8.7, we provide additional results while comparing our implementation with different DiRT organizations and management policies. For these estimates, we also conservatively assumed a 48-bit physical address (12 bits used for 4KB page offset), which increases our tag size. The total overheads are summarized in Table 2.

Hardware	Size
Counting Bloom Filters	3 * 1024 entries * 5-bit counter = 1920B
Dirty List	256 sets * 4-way * (1-bit NRU + 36-bit tag) = 4736B
Total	**6656B = 6.5KB**

Table 2: Hardware cost of the Dirty-Region Tracker.

7. Experimental Results

7.1. Methodology

Simulation Infrastructure: We use MacSim [6], a cycle-level x86 simulator, for performance evaluations. We model a quad-core processor with two-level SRAM caches (private L1 and shared L2) and an L3 DRAM cache. The stacked DRAM is expected to support more channels, banks, and wider buses per channel [9]. In this study, the DRAM cache has four channels with 128-bit buses, and each channel has eight banks, while the conventional off-chip DRAM has 2 channels, each with 8 banks and a 64-bit bus. Also, key DDR3 timing parameters with bank conflicts and data bus contention are modeled in our DRAM timing module. Table 3 shows the system configurations used in this study.

CPU	
Core	4 cores, 3.2GHz out-of-order, 4 issue width, 256 ROB
L1 cache	4-way, 32KB I-Cache + 32KB D-Cache (2-cycle)
L2 cache	16-way, shared 4MB (4 tiles, 24-cycle)
Stacked DRAM cache	
Cache size	128MB
Bus frequency	1.0GHz (DDR 2.0GHz), 128 bits per channel
Channels/Ranks/Banks	4/1/8, 2KB row buffer
tCAS-tRCD-tRP	8-8-15
tRAS-tRC	26-41
Off-chip DRAM	
Bus frequency	800MHz (DDR 1.6GHz), 64 bits per channel
Channels/Ranks/Banks	2/1/8, 16KB row buffer
tCAS-tRCD-tRP	11-11-11
tRAS-tRC	28-39

Table 3: System parameters used in this study.

Workloads: We use the SPEC CPU2006 benchmarks and sample 200M instructions using SimPoint [14]. Then we categorize the applications into two different groups based on the misses per kilo instructions (MPKI) in the L2 cache. We restrict the study to workloads with high memory traffic; applications with low memory demands have very little performance sensitivity to memory-system optimizations and therefore expose very little insight (we did verify that our techniques do not negatively impact these benchmarks). Out of the memory-intensive benchmarks, those with average MPKI rates greater than 25 are in Group H (for **H**igh intensity), and of the remaining, those with 15 MPKI or more are in Group M (for **M**edium). Table 4 shows MPKI values of the benchmarks and their group.

Group M	MPKI	Group H	MPKI
GemsFDTD	19.11	leslie3d	25.85
astar	19.85	libquantum	29.30
soplex	20.12	milc	33.17
wrf	20.29	lbm	36.22
bwaves	23.41	mcf	53.37

Table 4: L2 misses per kilo instructions (L2 MPKI).

We select benchmarks to form rate-mode (all cores running separate instances of the same application) and multi-programmed workloads. Table 5 shows the primary workloads evaluated for this study. Section 8 also includes additional results covering a much larger number of workloads.

For each workload, we simulate 500 million cycles of execution. We verified that the DRAM cache is sufficiently warmed up: the DRAM cache access statistics at the end of the simulation show that the number of valid cache lines is equal to the total capacity of the cache, and the total number of evictions is 5×-6× greater than the total cache capacity.

Mix	Workloads	Group
WL-1	4 × mcf	4×H
WL-2	4 × lbm	4×H
WL-3	4 × leslie3d	4×H
WL-4	mcf-lbm-milc-libquantum	4×H
WL-5	mcf-lbm-libquantum-leslie3d	4×H
WL-6	libquantum-mcf-milc-leslie3d	4×H
WL-7	mcf-milc-wrf-soplex	2×H + 2×M
WL-8	milc-leslie3d-GemsFDTD-astar	2×H + 2×M
WL-9	libquantum-bwaves-wrf-astar	1×H + 3×M
WL-10	bwaves-wrf-soplex-GemsFDTD	4×M

Table 5: Multi-programmed workloads.

Performance Metric: We report performance using weighted speedup [5, 16], which is computed as:

$$\text{Weighted Speedup} = \sum_i \frac{\text{IPC}_i^{\text{shared}}}{\text{IPC}_i^{\text{single}}}.$$

The geometric mean is also used to report average values.

7.2. Performance

Figure 8 shows the performance of the proposed hit-miss predictor (HMP), self-balancing dispatch (SBD), and dirty region tracker (DiRT) mechanisms for multi-programmed workloads. For comparison, we use a baseline where the DRAM cache is not employed. We also compare our mechanisms with the previously proposed MissMap structure (denoted as MM in the figure). We model a MissMap with zero storage overhead; i.e., no L2 cache capacity is sacrificed for the MissMap, but the L2 latency is still incurred for the lookup.

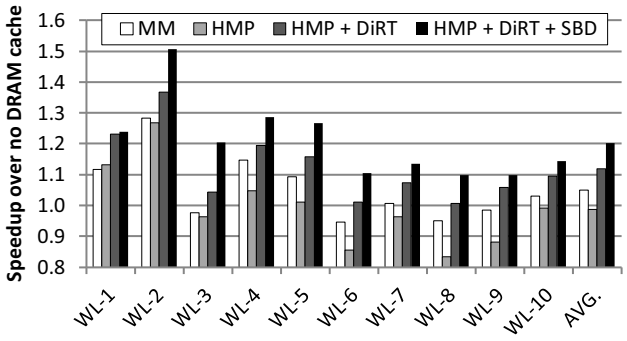


Figure 8: Performance normalized to no DRAM cache for MissMap (MM), and combinations of HMP, SBD, and DiRT.

We first evaluate the impact of hit-miss prediction (HMP) without DiRT. In this usage scenario, every predicted miss request serviced from off-chip memory must wait to be verified as *safe* (i.e., no dirty data in the DRAM cache). As a result, for most benchmarks, HMP without DiRT performs worse than MissMap. This is not necessarily a negative result when one considers that the HMP approach sacrifices the multi-megabyte MissMap for a much smaller sub-kilobyte predictor. Achieving even close to similar performance while removing such a large hardware overhead is still a desirable result. However, with DiRT support, HMP+DiRT performs even better than MissMap

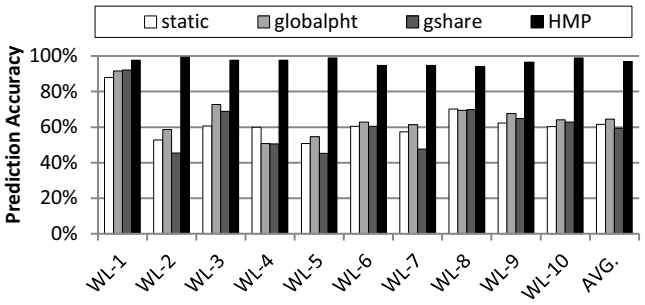


Figure 9: Prediction accuracy of HMP and its comparison with other types of predictors.

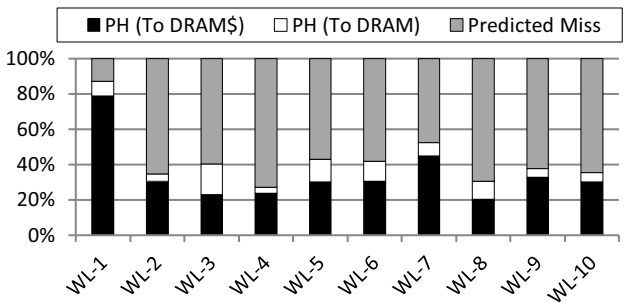


Figure 10: Issue direction breakdown. PH indicates predicted hit requests.

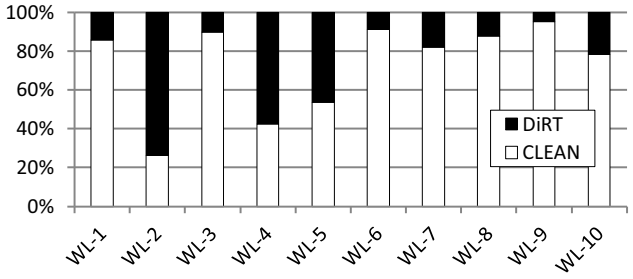


Figure 11: Percentage distribution of memory requests captured in DiRT.

due to the elimination of fill-time prediction verifications for clean blocks (which are the common case). At this point, the performance benefit *over* MissMap is primarily due to the replacement of the 24-cycle MissMap latency with a 1-cycle HMP lookup.

Next, we apply the SBD mechanism on top of HMP+DiRT. As shown in the results, SBD further improves performance (often significant, depending on the workload). Compared to HMP+DiRT, SBD provides an additional 8.3% performance benefit on average. In summary, the proposed mechanisms (HMP+DiRT+SBD) provide a 20.3% performance improvement over the baseline. Also, compared to MissMap, they deliver an additional 15.4% performance over the baseline. On a last remark, one should note that the evaluated MissMap does not sacrifice the L2 cache (i.e., ideal), so our mechanisms would perform even better when compared to a non-ideal MissMap that reduces the effective SRAM cache size.

8. Analysis

In this section, we provide additional analysis on the proposed hit-miss predictor, self-balancing dispatch, and the dirty region tracker.

8.1. HMP: Prediction Accuracy

Figure 9 shows the prediction accuracy of the proposed predictor with comparison to some other types of predictors. *static* indicates the best of either static-hit or static-miss predictors, so the value is always great than 0.5. A reasonable predictor at least should be better than *static*. *globalpht* is the implementation of only one 2-bit counter for all memory requests, where it is incremented/decremented on a hit/miss. *gshare* is a gshare-like cache predictor (i.e., using the XOR of a requested 64B block address with a global history of recent hit/miss outcomes to index into a pattern history table).

First, the results show that our predictor provides more than 95% prediction accuracy on the evaluated workloads (average of 97%), which implies that the spatial locality-based hit/miss prediction is highly effective. Next, compared to *static*, we can see that the other predictors actually do not improve prediction accuracy much. For WL-1, all of the predictors perform well because the workload has a high hit rate and is simply easy to predict. But, if the hit ratio is around 50% as in other workloads, the other predictors perform poorly. For *globalpht*, one core may be consistently hitting while the other is consistently missing, and as a result the simple counter could ping-pong back and forth generating mispredictions. For *gshare*, the hit/miss history register provides poor information, and its inclusion often introduces more noise than useful correlations, resulting in overall lower prediction rates. In summary, HMP outperforms other predictors that use the individual 64B request address and/or history information of the actual outcomes.

8.2. SBD: Percentage of Balanced Hit Requests

Figure 10 shows the distribution of SBD's issue decisions (i.e., DRAM or DRAM cache). The black bar (PH: To DRAM$) represents the percentage of requests that are predicted hits and are actually issued to the DRAM cache, while the white bar (PH: To DRAM) represents the predicted-hit requests that were diverted to off-chip memory. Note that SBD does not work on the predicted-miss requests; thus, the requests in the (Predicted Miss) portion are always issued to off-chip memory.

At first thought, one might think that SBD does not operate on the benchmarks whose hit ratios are low (e.g., below 50%) because the amount of traffic to off-chip DRAM would be greater than that to the DRAM cache. Due to the bursty nature of memory requests, however, the instantaneous hit ratio and/or bandwidth requirements vary from the average values; thus, the balancing mechanism provided by SBD can still be beneficial even for the low hit-ratio workloads. In fact, as shown in the results, SBD was able to redistribute some of the hit requests for all of the benchmarks.

8.3. DiRT: Benefit and Traffic

Figure 11 shows the percentage distribution of write-through mode (CLEAN) and write-back mode (DiRT) memory requests. The "CLEAN" portion indicates the number of requests that are not found in the DiRT; thus, they are free to be predicted or self-balanced. The results show that DiRT allows a significant amount of memory requests to be handled without fear of returning a stale value. Note that without DiRT, every request that is a predicted miss (or a predicted hit but diverted to DRAM) needs to wait until it has been verified that the DRAM cache does not contain a dirty copy.

Figure 12 illustrates the amount of write-back traffic to off-chip DRAM for write-through, write-back, and the DiRT-enabled hybrid

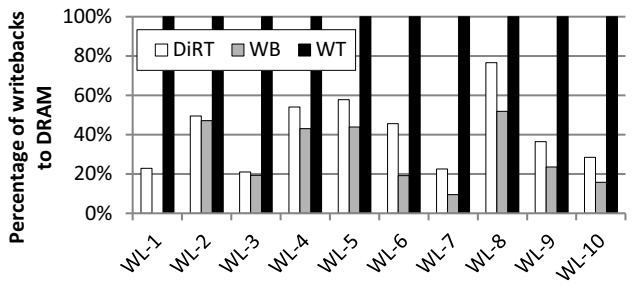


Figure 12: Write-back traffic to off-chip DRAM between write-through, write-back, and DiRT (WL-1 does not generate WB traffic), all normalized to the write-through case.

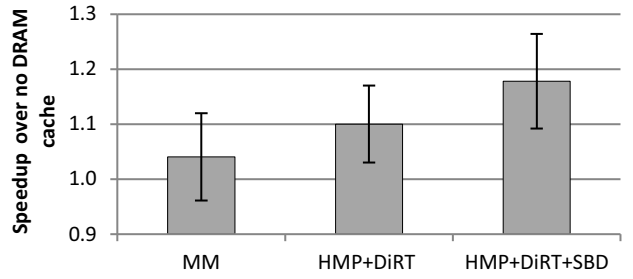


Figure 13: Average performance of MissMap and our proposed mechanisms over no DRAM cache baseline with +/-1 std. deviation for 210 workloads.

policy, all normalized to the write-through case. As shown in the results, a write-back policy performs a significant amount of write-combining and thereby greatly reduces the amount of write traffic to main memory as compared to a write-through policy. DiRT is not perfect and it does increase write traffic slightly compared to a true write-back policy, but the total write traffic from a DiRT-enabled DRAM cache is much closer to that of the write-back case than it is to write-through. The relatively small increase in write traffic due to the DiRT is more than compensated by the streamlined HMP speculation and increased opportunities for SBD.

8.4. Sensitivity to Different Workloads

To ensure that our mechanisms work for a broader set of scenarios beyond the ten primary workloads used thus far in this paper, we simulated all 210 combinations ($_{10}C_4$) of the ten Group H and Group M benchmarks. Figure 13 shows the performance results averaged over all of the 210 workloads, along with error-bars to mark one standard deviation. As shown in the figure, our mechanisms combine to deliver strong performance over the previously proposed MissMap-based DRAM cache approach.

8.5. Sensitivity to DRAM Cache Sizes

Figure 14 shows the performance of the proposed mechanisms with different sizes of DRAM caches. The results show that the benefit of MissMap, HMP+DiRT, and HMP+DiRT+SBD increases as the cache size grows. For all cache sizes, HMP+DiRT+SBD still performs best. In addition, the benefit of SBD increases as the DRAM cache size increases because the higher hit rate provides more opportunities for SBD to dispatch requests to main memory.

8.6. Sensitivity to DRAM Cache Bandwidth

In our evaluation, the ratio of peak DRAM cache bandwidth to main memory is 5:1 (2GHz vs. 1.6GHz, 4 channels vs. 2 channels, and

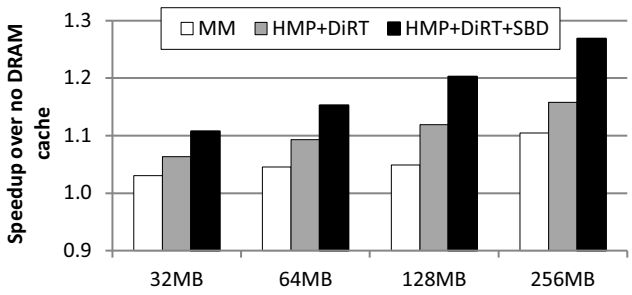


Figure 14: Performance sensitivity of the proposed mechanisms to different DRAM cache sizes.

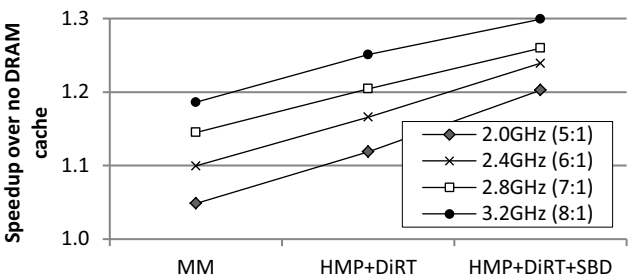


Figure 15: Performance sensitivity to different ratios of DRAM cache bandwidth to off-chip memory.

128-bit bus per channel vs. 64-bit bus per channel). While we believe that this is reasonable for plausible near-term systems,[6] it is also interesting to see how the effectiveness of HMP and SBD scales under different bandwidth assumptions. Figure 15 shows the performance sensitivity when we increase the DRAM cache frequency from 2.0GHz (what was used so far in this paper) up to 3.2GHz. First, as shown in the results, the benefit of HMP does not decrease if future die-stacked DRAMs provide more bandwidth (relative to off-chip). As the DRAM-cache frequency increases, the cost of the 24-cycle MissMap increases relative to the DRAM-cache latency, and therefore HMP provides a small but increasing relative performance benefit as well. On the other hand, increasing the DRAM cache frequency reduces the *relative* additional bandwidth provided by the off-chip DRAM, thereby potentially decreasing the effectiveness of SBD. In our experiments, we do observe that the relative benefit of SBD over HMP reduces as the DRAM cache bandwidth increases, but overall, SBD still provides non-negligible benefits even with higher-frequency DRAM caches. Note that the die-stacked DRAM bandwidth may not grow too rapidly (such as 32:1), as adding more TSVs requires die area on the memory chips (which directly impacts cost), and increasing bandwidth via higher-frequency interfaces has power implications.

8.7. Sensitivity to DiRT Structures

Figure 16 shows the performance results as we vary the number of Dirty List entries (first four bars), assuming a fully-associative structure with LRU replacement. Note that such a structure would be difficult to implement for these sizes (e.g., true LRU on 1K entries).

[6]For example, current x86 processors tend to have two DDR3 memory channels (some have three or four, which would provide even more opportunities for SBD). The JEDEC Wide-IO standard provides four channels at 128 bits each, which is the same as our stacked DRAM configuration.

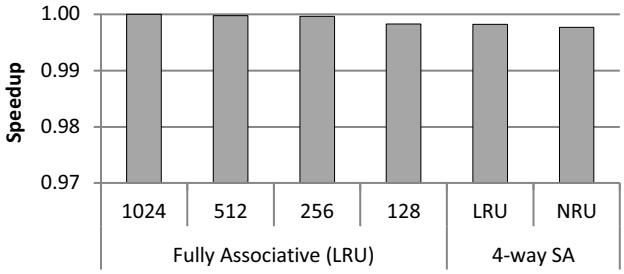


Figure 16: Performance sensitivity to different DiRT structures and management policies.

Overall, there is very little performance degradation even when reducing the size of the DiRT to only 128 entries, but we still chose to employ a 1K-entry table to reduce the performance variance across workloads.

The right side of Figure 16 also shows the results for 4-way set-associative implementations, each with 1K entries. The right-most bar (1K entry, 4-way, NRU) is the configuration used thus far in the paper and has the lowest implementation complexity and cost. Overall, the results show that even with our simplified DiRT organization, we lose very little performance compared to an impractical fully-associative, true-LRU solution.

9. Conclusion

The prior work by Loh and Hill provided an important step toward the realization of a more practical die-stacked DRAM cache solution. However in this paper, we have shown that there still exist inefficiencies in the prior solution. In particular, the assumption that precise cache-content tracking was needed led to a MissMap structure that was over-designed for the DRAM cache. By taking advantage of the simple observation that on a miss, tag reads for victim selection need to occur anyway, false-negative mispredictions can be verified to prevent returning stale data from main memory back to the processors. The ability to freely speculate enables our DRAM cache organization that avoids the hardware overheads of the MissMap.

We also observed that while the die-stacked DRAM may provide significant bandwidth, the off-chip memory bandwidth is still a valuable resource that should not be disregarded. Our Self-Balancing Dispatch approach allows our DRAM cache design to make better use of the system's *aggregate* bandwidth. For both the HMP and SBD approaches, we found that life is significantly easier when we do not need to worry about dirty data. Completely abolishing dirty data from the DRAM cache with a write-through policy causes write traffic to increase tremendously. However, by bounding (and tracking) a limited number of pages in write-back mode, we could greatly amplify the effectiveness of both HMP and SBD techniques. Overall, we have proposed a significantly streamlined DRAM cache organization that improves performance over the state of the art while eliminating the large MissMap structure.

Beyond the ideas presented in this paper, there likely remain other research opportunities to further improve the performance and/or practicality of die-stacked DRAM caches. For instance, the motivational example showing under-utilized off-chip bandwidth also illustrates the high cost of placing the tags in the DRAM cache in terms of bandwidth. In particular, the raw $8\times$ higher bandwidth of the die-stacked DRAM (compared to off-chip) is reduced to only a $2\times$ increase in the effective bandwidth in terms of serviceable requests per unit time. DRAM cache organizations that can make more efficient use of the DRAM cache's raw bandwidth would likely provide further performance benefits. Studies on further improving the practicality of die-stacked DRAM caches, such as the interaction with cache coherence, are also good directions for future research.

Acknowledgments

We would like to thank Andreas Moshovos, the Georgia Tech HPArch members, and the anonymous reviewers for their suggestions and feedback. Part of this work was conducted while Jaewoong Sim was on an internship and Mithuna Thottethodi was on sabbatical leave at AMD Research. We gratefully acknowledge the support of the NSF CCF-0644183 (Thottethodi); and AMD, Sandia National Laboratories, and NSF CAREER award 1139083 (Kim).

References

[1] B. Black, M. M. Annavaram, E. Brekelbaum, J. DeVale, L. Jiang, G. H. Loh, D. McCauley, P. Morrow, D. W. Nelson, D. Pantuso, P. Reed, J. Rupley, S. Shankar, J. P. Shen, and C. Webb, "Die Stacking (3D) Microarchitecture," in *MICRO-39*, 2006.

[2] P. Conway, N. Kalyanasundharam, G. Donley, K. Lepak, and B. Hughes, "Cache Hierarchy and Memory Subsystem of the AMD Opteron Processor," *IEEE Micro*, March–April 2010.

[3] S. Damaraju, V. George, S. Jahagirdar, T. Khondker, R. Milstrey, S. Sarkar, S. Siers, I. Stolero, and A. Subbiah1, "A 22nm IA Multi-CPU and GPU System-on-Chip," in *ISSCC*, 2012.

[4] X. Dong, Y. Xie, N. Muralimanohar, and N. P. Jouppi, "Simple but Effective Heterogeneous Main Memory with On-Chip Memory Controller Support," in *SC*, 2010.

[5] S. Eyerman and L. Eeckhout, "System-Level Performance Metrics for Multiprogram Workloads," *IEEE Micro*, May–June 2008.

[6] "Macsim simulator," http://code.google.com/p/macsim/, HPArch.

[7] A. Jaleel, K. Theobald, S. C. Steely, and J. Emer, "High Performance Cache Replacement Using Re-Reference Interval Prediction (RRIP)," in *ISCA-32*, 2010.

[8] X. Jiang, N. Madan, L. Zhao, M. Upton, R. Iyer, S. Makineni, D. Newell, Y. Solihin, and R. Balasubramonian, "CHOP: Adaptive Filter-Based DRAM Caching for CMP Server Platforms," in *HPCA-16*, 2010.

[9] J.-S. Kim, C. Oh, H. Lee, D. Lee, H.-R. Hwang, S. Hwang, B. Na, J. Moon, J.-G. Kim, H. Park, J.-W. Ryu, K. Park, S.-K. Kang, S.-Y. Kim, H. Kim, J.-M. Bang, H. Cho, M. Jang, C. Han, J.-B. Lee, K. Kyung, J.-S. Choi, and Y.-H. Jun, "A 1.2V 12.8GB/s 2Gb Mobile Wide-I/O DRAM with 4x128 I/Os Using TSV-Based Stacking," in *ISSCC*, 2011.

[10] H. Liu, M. Ferdman, J. Huh, and D. Burger, "Cache Bursts: A New Approach for Eliminating Dead Blocks and Increasing Cache Efficiency," in *MICRO-41*, 2008.

[11] G. H. Loh and M. D. Hill, "Efficiently Enabling Conventional Block Sizes for Very Large Die-Stacked DRAM Caches," in *MICRO-44*, 2011.

[12] G. H. Loh, N. Jayasena, K. McGrath, M. O'Connor, S. Reinhardt, and J. Chung, "Challenges in Heterogeneous Die-Stacked and Off-Chip Memory Systems," in *SHAW-3*, 2012.

[13] A. Seznec and P. Michaud, "A Case for (Partially) TAgged GEometric History Length Branch Prediction," *Journal of Instruction-Level Parallelism*, 2006.

[14] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically characterizing large scale program behavior," in *ASPLOS-X*, 2002.

[15] J. E. Smith, "A Study of Branch Prediction Strategies," in *ISCA-3*, 1981.

[16] A. Snavely and D. Tullsen, "Symbiotic Job Scheduling for a Simultaneous Multithreading Processor," in *ASPLOS-IX*, 2000.

[17] G. Taylor, P. Davies, and M. Farmwald, "The TLB Slice – A Low-Cost High-Speed Address Translation Mechanism," in *ISCA-12*, 1990.

[18] A. Yoaz, M. Erez, R. Ronen, and S. Jourdan, "Speculation Techniques for Improving Load Related Instruction Scheduling," in *ISCA-21*, 1999.

[19] J. Zhao, G. Sun, Y. Xie, and G. H. Loh, "Energy-Efficient GPU Design with Reconfigurable In-Package Graphics Memory," in *ISLPED*, 2012.

[20] L. Zhao, R. Iyer, R. Illikkal, and D. Newell, "Exploring DRAM Cache Architectures for CMP Server Platforms," in *ICCD-25*, 2007.

2012 IEEE/ACM 45th Annual International Symposium on Microarchitecture

CoLT: Coalesced Large-Reach TLBs

Binh Pham* Viswanathan Vaidyanathan* Aamer Jaleel† Abhishek Bhattacharjee*
* Dept. of Computer Science, Rutgers University † Intel Corporation, VSSAD
{binhpham, viswav, abhib}@cs.rutgers.edu aamer.jaleel@intel.com

Abstract

Translation Lookaside Buffers (TLBs) are critical to system performance, particularly as applications demand larger working sets and with the adoption of virtualization. Architectural support for superpages has previously been proposed to improve TLB performance. By allocating contiguous physical pages to contiguous virtual pages, the operating system (OS) constructs superpages which need just one TLB entry rather than the hundreds required for the constituent base pages. While this greatly reduces TLB misses, these gains are often offset by the implementation difficulties of generating and managing ample contiguity for superpages.

We show, however, that basic OS memory allocation mechanisms such as buddy allocators and memory compaction naturally assign contiguous physical pages to contiguous virtual pages. Our real-system experiments show that while usually insufficient for superpages, these intermediate levels of contiguity exist under various system conditions and even under high load. In response, we propose Coalesced Large-Reach TLBs (CoLT), which leverage this intermediate contiguity to coalesce multiple virtual-to-physical page translations into single TLB entries. We show that CoLT implementations eliminate 40% to 58% of TLB misses on average, improving performance by 14%.

Overall, we demonstrate that the OS naturally generates page allocation contiguity. CoLT exploits this contiguity to eliminate TLB misses for next-generation, big-data applications with low-overhead implementations.

1. Introduction

Translation Lookaside Buffers (TLBs) are crucial to system performance due to their long miss penalties [5, 11, 14, 19, 22, 26]. Past work has shown that TLB misses degrade performance by 5% to 14% for even nominally-sized applications. This number worsens to 50% in virtualized environments or when the application's memory footprint increases [8, 21].

Superpages have previously been proposed to increase TLB coverage [15, 16, 23, 25, 28, 29]. A superpage is a memory page that is sized as a multiple of a base page and is typically in the megabyte or gigabyte range. Each TLB superpage entry can thus replace hundreds of baseline TLB entries, boosting TLB coverage.

By construction, superpages target situations where the OS can seamlessly generate vast amounts of contiguity. Unfortunately, a number of issues may preclude this. For example, superpages may magnify an application's memory footprint, increasing paging traffic. Superpages also require specialized and high-overhead algorithms to assign aligned and contiguous physical page frames to contiguous virtual pages [25, 28, 29]. The OS therefore selectively uses superpages such that the benefits of reduced TLB misses are not outweighed by these overheads.

This work observes that there actually exists a second regime of page allocation contiguity, orthogonal to superpaging. Specifically, OS memory allocators use buddy allocators and memory compaction daemons which also, by construction, allocate contiguous physical page frames to contiguous virtual pages. These mechanisms generate *intermediate contiguity* (in the range of tens of pages), which falls short of superpage requirements (hundreds of contiguous pages). However, this contiguity is achieved without superpaging overheads like increased I/O traffic and sophisticated page construction algorithms. In response, we propose Coalesced Large-Reach TLBs (CoLT), a series of hardware mechanisms that allow TLBs to coalesce multiple contiguous virtual-to-physical address translations, increasing their memory reach. CoLT specifically targets large amounts of intermediate contiguity that superpaging cannot exploit. Our contributions are as follows:

- We study, on a real system, how often consecutive virtual pages are allocated consecutive physical pages. We find that even with high system fragmentation, tens of pages are usually contiguous. Furthermore, while superpaging does increase contiguity, much of it falls short of the level necessary to actually create large pages (a 2MB superpage needs 512 contiguous 4KB base pages, while we see tens of contiguous pages). Instead, we show that TLB coalescing effectively leverages this contiguity.
- We propose CoLT for set-associative, two-level TLB hierarchies commonly found in processors today [3, 17]. Our strategies eliminate roughly 40% of L1 and L2 TLB misses, resulting in average performance improvements of 12%.
- We develop CoLT support for small, fully-associative TLBs commonly found in processors to cache superpage entries [3, 17]. We show how to overcome the challenge of designing these small structures, achieving L1 and L2 TLB miss elimination rates of 58% on average. These translate to average performance improvements of 14%.
- Finally, we combine the benefits of coalescing on both set-associative and fully-associative TLBs, improving average performance by 14%.

Overall, we design low-overhead hardware to exploit intermediate levels of page allocation contiguity. Our studies evaluate this approach under a variety of system configurations with heavy system load and different superpaging configurations.

2. Background and Related Work

2.1. Prior TLB Enhancement Techniques

Address translation, especially with virtualization and larger application working sets, is a primary source of system performance degradation [8, 21]. In response, researchers have considered techniques to improve TLB structure, lookup, and placement [9, 13]. More sophisticated techniques such as TLB prefetching and mechanisms to accelerate page walks have also been considered [5, 11, 19, 27]. Our goal is to propose techniques, orthogonal to past work, to further boost TLB performance.

2.2. Superpaging Benefits and Problems

Superpages or hugepages have previously been proposed to lower TLB miss rates [15, 16, 23, 25, 28, 29]. Superpages are typically

1072-4451/12 $26.00 © 2012 IEEE
DOI 10.1109/MICRO.2012.32

IEEE computer society

sized as power-of-two multiples of baseline pages. For example, x86 systems use 4KB baseline pages and support 2MB and 1GB superpages. Furthermore, superpages must be aligned in both virtual and physical memory (superpages of size N must begin at virtual and physical addresses that are multiples of N).

While superpages lower TLB miss rates by replacing hundreds to thousands of base page translations with a single superpage translation entry, they have management overheads [23, 28]. For example, dedicated OS code is required to support multiple large page sizes [25, 28, 29]. The process of ensuring that sufficient contiguous physical pages are allocated to virtual pages can suffer high performance overheads, particularly when alignment restrictions are also imposed [15, 23]. Furthermore, if not fully-utilized, superpages can increase the amount of I/O traffic and increase page initialization/fault latency. A particular problem is the use of a single dirty bit for all the baseline pages of a superpage; if set, the entire superpage must be written back to disk even if only one base page has been modified, greatly increasing disk traffic. Therefore, the OS weighs these overheads against the benefits of superpaging. Typically, the OS uses superpages sparingly, only bothering to generate large amounts of contiguity when superpaging is deemed to be worthwhile [4].

2.3. TLB Subblocking and Speculation

Two hardware schemes have been proposed to mitigate superpaging overheads. Talluri and Hill [28] present complete-subblock and partial-subblock TLBs, which record ranges of physical pages per virtual page entry. Complete subblocking, while effective, requires non-trivial modifications to traditional TLB hardware. Partial-subblocking overcomes these overheads but with explicit OS support [28]. Furthermore, subblocking was originally proposed for fully-associative designs rather than the set-associative organizations most commonly used in products today. Finally, sub-blocking effectiveness depends heavily upon page alignment. Both complete and partial subblocking are most effective when the first virtual page of a contiguous group of virtual (and physical pages) is aligned to the subblock length. Partial subblocking goes beyond this, reducing hardware overheads by requiring that base physical pages also be placed in an aligned manner within subblock regions [28]. Overall, these requirements may constrain exploitable contiguity.

Alternately, past work [6] proposes TLB speculation for systems with reservation-based superpaging [23, 28]. Here, physical pages are allocated in aligned 2MB regions of physical memory corresponding to their alignment within a 2MB region of virtual memory. Barr, Cox, and Rixner exploit this property with a SpecTLB structure, which interpolates between existing TLB entries to predict physical translations for TLB misses. While effective, SpecTLB requires reservation-based superpaging, which is not universally used (eg. Linux superpaging [4] does not use reservation-based superpages). SpecTLB also requires an additional TLB-like structure, and can increase instruction replays on incorrect speculations.

2.4. Our Approach

In general, there are three regimes of page contiguity that the OS generates. In the first regime, the OS does not succeed in mapping contiguous physical pages to contiguous virtual pages. In these cases, traditional techniques such as changing TLB organizations or prefetching are likely to be successful in increasing hit rates [6, 11, 19]. At the other end of the spectrum, there exists a regime of extremely high contiguity, when the OS decides that the overheads of superpage construction are well worth the effort. Our goal, in this work, is to exploit a third regime with intermediate contiguity, where tens to hundreds of base pages are contiguous. Our characterization studies will show that the OS naturally produces this contiguity without the overheads of superpages and that this level of contiguity is more prevalent than the other regimes. Our studies will be on a real system under a comprehensive set of system environments which include heavily-fragmented systems and systems with and without superpaging.

We will then realize low-overhead TLB hardware to exploit intermediate contiguity. Unlike prior work on speculation or prefetching [6, 11, 19], CoLT does not augment the standard TLBs with separate structures. Unlike superpages or subblocking, we avoid OS intrusion and do not require prescribed amounts of contiguity. Instead, CoLT studies available contiguity and exploits as much of it as possible.

3. Understanding Page Allocation Contiguity

We now explore why operating systems often allocate contiguous physical page frames to contiguous virtual pages. Since CoLT relies on this behavior, we ascertain which memory allocation policies and mechanisms produce contiguity.

3.1. Defining Page Allocation Contiguity

We say that system contiguity exists when consecutive virtual pages are allocated consecutive physical page frames. For example, if virtual pages *1*, *2*, and *3* are allocated physical page frames *58*, *59*, and *60*, we say that these pages are contiguous. Moreover, since this example involves three pages, we say that this is an instance of *3-page* contiguity.

Our definition is distinct from superpages in two ways. First, superpages require a set amount of contiguity. For example, 2MB superpages on x86 systems require instances of *512-page* contiguity. Instead, we make no restrictions on the amount of contiguity that is useful. Second, unlike superpages, we make no assumption on alignment. Our relaxations on contiguity amounts and alignment restrictions reveal huge great intermediate contiguity.

Note that this definition requires simultaneous contiguity in both virtual *and* physical page numbers. As such, CoLT does not affect cases where only virtual (or physical) pages are contiguous; these will be cached by TLBs in the conventional manner.

3.2. Sources of Page Allocation Contiguity

Operating systems maintain a complex set of policies and mechanisms to determine how to allocate physical memory to applications. A number of these policies promote physical page contiguity. We elaborate on them here, focusing on Linux for our discussion. Note, however, that our observations extend more broadly to other operating systems which tend to utilize similar mechanisms to allocate physical pages.

3.2.1. Buddy allocation. Linux uses a buddy allocator to track physical pages and assign them to virtual pages on demand. Figure 1 illustrates the operation of a buddy allocator, assuming that pages 1, 2, and 3 are already allocated. All free physical pages or page frames (PFs) are grouped into ten lists of blocks, which we refer to as *free lists*. Entry x in the free list tracks groups of 2^x *contiguous* physical pages. For example, pages 4-7 have 4-page contiguity and are hence listed by entry two.

Physical page allocations proceed as follows. Suppose an application requires an N-page data structure. To accommodate this, the

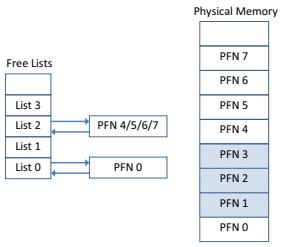


Figure 1: Buddy allocator used for physical page allocation. Already allocated pages are shaded, while free pages are tracked by the free lists.

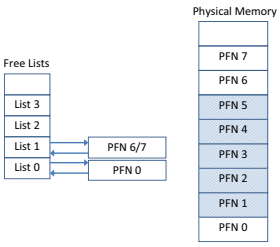


Figure 2: Buddy allocator state after an allocation for 2 pages is finished.

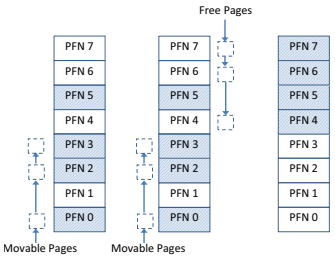


Figure 3: The memory compaction daemon tracks movable and free memory pages, exchanging them to eliminate fragmentation.

application makes a `malloc` call, simultaneously requesting N physical pages from the OS. The buddy allocator first searches the free list entry corresponding to the smallest contiguous page frames bigger than N (entry $\lceil log_2(N) \rceil$). If a block of free physical pages is found in that list, allocation successfully completes. Otherwise, the free list is progressively climbed until an entry with a block of free contiguous physical pages is found. Once a free block is found, the buddy allocator must minimize memory fragmentation. Therefore it iteratively halves the block, inserting these new blocks in their appropriate free list locations, until it extracts a block of N contiguous physical pages. As an example, Figure 2 shows the state of the free list after an application level request for two physical pages to be allocated. At first, entry 1 in the free list is checked; however, since this is empty, entry 2 is scanned. Here, a free block with contiguous physical pages 4, 5, 6, and 7 is found. Hence, the buddy allocator halves this block of four pages, returning pages 4 and 5 to the application and moving pages 6 and 7 to free list entry 1. Apart from allocation, the buddy allocator also updates its state when physical pages are released. At this point, the kernel attempts to merge pairs of free *buddy* blocks if both have the same size and are contiguous.

By construction therefore, the buddy allocator deliberately provides contiguous physical page frames to the application when it asks for multiple page frames together. Since applications usually make `malloc` calls that simultaneously request a number of physical pages together (rather than one page at a time), the buddy allocator is able to provide them a suitable contiguous range of pages. We will show that the buddy allocator successfully produces this contiguity even in the presence of significant system load. While insufficient for superpages, CoLT benefits from this substantially.

3.2.2. Memory compaction. In order to glean contiguous runs of physical pages, the buddy allocator relies on ensuring that memory fragmentation is tightly controlled. However, fragmentation is pronounced when multiple processes with large working sets simultaneously run on the system. Therefore, many operating systems boost the buddy allocator with a separate memory compaction daemon. Figure 3 details the Linux memory compaction daemon in three steps on a heavily-fragmented system.

First, as shown in the left-most diagram of Figure 3, memory compaction runs an algorithm that starts at the bottom of the physical memory and builds a list of allocated pages that are *movable*. While most user-level pages are movable, pinned and kernel pages usually are not. Nevertheless, user-level pages usually outnumber kernel pages, making most pages movable.

Second, the daemon starts at the top of physical memory and builds a list of free pages. Eventually, the two algorithms meet in the middle of the physical page list. At this point, Linux invokes migration code to shift the movable pages to the free page list, yielding the unfragmented diagram at the right of Figure 3.

Since there is a cost associated with moving pages, the compaction daemon is only triggered when there is heavy system fragmentation. As such, its operation naturally produces contiguity, especially in tandem with the buddy allocator. In fact, we will show that this daemon successfully generates contiguity even under heavy system fragmentation.

3.2.3. Transparent hugepage support. Aside from buddy allocation and memory compaction, support for superpages is a primary cause of page allocation contiguity. Unfortunately superpage management comes with high overheads. As a result, Linux's Transparent Hugepage Support (THS), supported since the 2.6.38 kernel [4], uses superpages sparingly. When THS is enabled, the memory allocator attempts to find a free 2MB block of memory. If this block is naturally aligned at a 2MB boundary, a superpage is constructed. In practice, the OS relies on the memory compaction daemon to construct these 2MB regions. Aligned 2MB regions are rare; when a superpage cannot be constructed, the system defaults to the buddy allocator. Even when the 2MB pages are allocated, increased load can eventually make them harmful. In these cases, system pressure triggers a daemon that breaks superpages into baseline 4KB pages.

In practice, THS struggles to maintain many superpages simultaneously. However, it does succeed in creating additional levels of contiguity for two reasons. First, while optimistically-allocated 2MB superpages are often eventually split due to system pressure, they retain contiguity among tens of baseline 4KB pages. Second, THS relies on the memory compaction daemon, triggering it more often and providing the buddy allocator even higher levels of contiguity suitable for CoLT.

3.2.4. System Load and Memory Fragmentation. Finally, page allocation contiguity is deeply affected by the system load. If many processes run simultaneously, main memory is likelier to be fragmented. Therefore, one may initially expect that higher load degrades contiguity. Surprisingly, we will show that contiguity can actually *increase* with greater system load. This occurs because system load has a complex relationship with the memory compaction daemon, triggering it more often when there is higher load. This can, in turn, free up more contiguous physical frames for the buddy allocator, eventually resulting in more contiguity.

4. CoLT Design and Implementation

Having detailed contiguity sources, we now propose three variants of CoLT. Overall, they share three design principles. First, they detect instances of consecutive virtual-to-physical address translations. These entries are coalesced into single TLB entries, so as to reduce miss rates. Second, CoLT coalesces only on *TLB misses*. While TLB hits could also prompt coalescing, this may increase

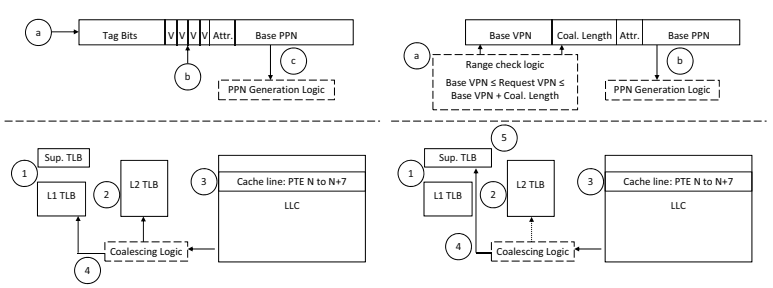


Figure 4: CoLT for set-associative L1 and L2 TLBs.

Figure 5: CoLT for fully-associative TLBs.

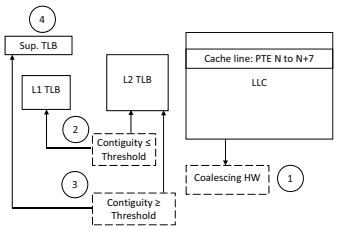


Figure 6: Combined CoLT for all TLBs.

lookup latencies. Third, coalescing is *unintrusive*, unlike speculation and prefetching [6, 11, 19, 27] which can degrade performance. For example, incorrect speculations suffer a high penalty. Incorrect prefetches lead to the eviction of useful entries and higher bandwidth usage. Prior work mitigates these problems by using separate structures to store prefetched translations or perform speculation [6, 11, 19]. In contrast, we coalesce entries directly into the TLBs but ensure that coalescing occurs only around *on-demand* translations. In the worst case, coalesced entries may be unused but are not harmful. This is crucial given that system contiguity does not necessarily imply that all contiguous translations are used in temporal proximity. We ensure coalesced entries are available if needed but do not harm TLB hit rates when they are unused.

We propose three variants of CoLT for two-level TLB hierarchies. This hierarchy contains set-associative L1 TLB and L2 TLBs, used to cache baseline 4KB pages [3, 17]. Superpages are cached in separate small, fully-associative TLBs that are accessed in parallel with the L1 TLB. Note that the L2 TLB is inclusive of just the set-associative L1 TLB and not the superpage TLB.

There are three natural coalescing mechanisms for this hierarchy. First, we coalesce in just the set-associative L1 and L2 TLBs. Second, we coalesce in the superpage TLB only. Third, we use a combined approach that routes some coalesced entries to the set-associative TLBs and others to the superpage TLBs. We now describe each of these schemes.

4.1. CoLT-SA Design and Implementation

CoLT-Set Associative (CoLT-SA) coalesces multiple virtual-to-physical page translations in the set-associative L1 and L2 TLBs. We first detail its high-level operation and then focus on specific design challenges.

4.1.1. Overall operation. The bottom half of Figure 4 shows a high-level view of CoLT-SA. In step 1, the set-associative L1 TLB and superpage TLB are looked up in parallel. Assuming L1 and L2 TLB misses (step 2), a page table walk brings in the desired translation entry into the LLC (step 3). We assume, like most x86 systems with dedicated MMU page table caches [5], that the LLC is the highest cache level for page table entries.

After the LLC fill, two parallel events occur. First, the requested translation is returned to the processor pipeline. In parallel, the *Coalescing Logic* studies the translations around the requested entry for contiguity. It coalesces as many of these translations as possible, as long as they map to the same set. This coalesced entry is inserted into the L1 and L2 TLBs (step 4). However, conventional set-associative TLBs map consecutive virtual addresses (and hence contiguous translations) to consecutive sets, precluding coalescing. We therefore modify the virtual page bits used for set-selection so that translations for groups of consecutive virtual page numbers do map to the same set. Furthermore, since we provide the requested translation to the pipeline in parallel with the *Coalescing Logic's* operation, the latter is off the critical path and does not affect TLB miss handling times.

4.1.2. TLB set selection. To understand how we modify TLB set selection to permit coalescing, consider the following example. An 8-set TLB would require three bits, bits 2 to 0 of the virtual page number for set selection (VPN[2-0]). Naturally, this would map consecutive translations to consecutive sets, preventing coalescing. However, if we left-shift the index bits by $log_2(N)$ bits, we may place N consecutive translations in the same set (permitting a maximum of N contiguous translations to be coalesced into a single entry). Therefore, to ensure that translations with four consecutive virtual pages map to the same set, we use VPN[4-2] as the new indexing bits.

To coalesce more entries, the indexing bits are further left-shifted (for example, to coalesce up to eight entries, VPN[5-3] must be used). However, using higher order bits for set indexing increases conflict misses since more consecutive entries are mapped to the same set. This is a fundamental tradeoff for CoLT-SA designs – in choosing the correct index bits, we must balance opportunities for coalescing with potentially higher conflict misses. We will show that allowing for coalescing of four contiguous translations generally performs best.

4.1.3. Lookup operation. The top half of Figure 4 illustrates CoLT-SA lookups. Each coalesced TLB entry maintains tag bits, the higher order bits left of the index bits used for set selection. For example, if up to four contiguous translations can be coalesced in a TLB with eight sets, VPN[4-2] is used for set selection and VPN[63-5] is the tag. In step (a), this tag is checked against the requested virtual page number. In step (b), the non-index lower-order virtual page bits (VPN[1-0] in our example) are used to select among multiple valid bits. There is one valid bit for every possible translation in a coalesced entry. These valid bits indicate the presence of a translation in the coalesced entry. If on step (b), a valid bit is set, there is a TLB hit. At this point, extra logic calculates the physical page number. CoLT entries store the base physical page number for each coalesced entry. This number corresponds to the virtual page represented by the first set valid bit. To reconstruct the physical page number, combinational logic (*PPN Generation Logic*) calculates the number of valid bits away this entry is from the first set valid bit. This number is added to the stored base physical page number to yield the desired physical page.

4.1.4. Practical coalescing restrictions. Ideally, after the page table is walked to handle a TLB miss, coalescing logic finds as many contiguous translations around the requested translation as possible. Practically, however, coalescing is restricted by two constraints. First, as we have already discussed, the choice of index bits for set selection places a limit on coalescing opportunity. A second limit arises from our desire to minimize the overhead associated with

searching for contiguous translations. On a TLB miss, a page table walk finds the desired translation. We aim to prevent any additional page walks when checking for contiguous entries adjacent to the requested translation. Since the page table walk accesses the last-level cache (LLC) and brings data in 64-byte cache lines, seven additional translations are fetched. These translations are brought without additional memory references; thus we check just them for contiguity. In practice, this approach restricts coalescing to a maximum of eight translations. Despite this restriction, CoLT eliminates a high number of TLB misses.

4.1.5. Replacement, invalidations, and attribute changes. CoLT-SA assumes standard LRU replacement policies. While there may be benefits in prioritizing entries with different coalescing amounts differently, we leave this for future work. We also assume a single set of attribute bits for all the coalesced entries, restricting coalescing opportunity. More sophisticated schemes supporting separate attribute bits per translation in a coalesced entry will improve our results. Furthermore on TLB invalidations, we flush out entire coalesced entries, losing information for pages that would be unaffected in standard TLBs. Gracefully uncoalescing TLB entries and only invalidating victim translations will perform even better. This too is the subject of future work.

4.1.6. Discussion of hardware overheads. To accommodate CoLT, the TLBs experience some key hardware changes. We argue, however, that these changes are modest. First, we believe that CoLT lookup remains low-overhead and does not impact TLB access cycle times. The initial tag match and check of valid bits is simple. The PPN generation logic addition is also low-overhead as the amount of coalescing is bounded (in our example, at best, an addition of four will be required). As such, readily-implementable combinational logic, similar to logic used to calculate prefetching strides and addresses or update branch predictor state, can calculate the physical page number. This is lower-overhead than prior prefetching schemes requiring dedicated adders [19].

Second, coalescing logic occurs on the TLB fill path rather than lookup, allowing subsequent TLB reads to proceed unimpaired. It is possible for subsequent reads to request translations that are part of the requested coalesced entries. These reads must wait until coalescing completes but we find these instances occur rarely. Furthermore, one might expect CoLT to require additional TLB ports to fill entries without conflicting with subsequent TLB reads. In our results, we assume no additional ports, finding that there is no significant performance degradation from this.

4.2. CoLT-FA Design and Implementation

Rather than supporting coalescing in set-associative TLBs and changing indexing schemes, we can instead coalesce into the fully-associative TLB (this structure is usually used exclusively for superpages). We refer to this as CoLT-Fully Associative (CoLT-FA).

4.2.1. Overall operation. The bottom half of Figure 5 delineates CoLT-FA operation. Assuming misses in all the TLBs (steps 1 and 2), a page walk is conducted in step 3. At this point, a cache line provides up to eight translations that can be checked for contiguity. Up to eight translations are now coalesced in step 4. If coalescable, the entry is loaded into the fully-associative TLB. If no coalescing is possible, it is loaded into the set-associative L1 and L2 TLBs.

On insertion into the fully-associative TLB, further coalescing is possible. Since contiguity may exist between the newly coalesced entry and a resident entry, the fully-associative TLB is scanned to seek further opportunities for coalescing. This scan is conducted while the requested TLB entry is returned to the processor. Further coalescing from the scan is done in step 5.

Empirically, we have found that due to the small size of the superpage-TLB, useful entries are frequently evicted. Therefore, for performance reasons, when bringing a coalesced entry into the fully-associative structure, we still bring just the requested entry (and not its coalesced neighbors) into the L2 TLB. While this does create some redundancy in terms of stored entries, we will show that performance is improved. Note that we leave the L1 TLB unaffected due to its much smaller capacity. Note also that CoLT-FA shares both superpage entries and coalesced entries in a single structure. One initial concern may be that if coalesced entries far outnumber superpage entries, the latter will be evicted from the fully-associative TLB. In practice, we find that this is not a problem for two reasons. First, superpages are used sparingly, requiring a very small number of entries in the buffer. Second, when used, these superpages are frequently accessed, meaning that they remain at the head of the LRU list, preventing their eviction.

4.2.2. Lookup operation. The top half of Figure 5 details CoLT-FA lookup. Each coalesced entry maintains a base virtual page number as the tag and a field that logs the number of entries coalesced. Unlike CoLT-SA, there are no coalescing restrictions due to indexing schemes. We find that using five bits for the coalescing length field suffices as this captures a contiguity of 32 pages. Each entry also stores the base physical page and attributes of all contiguous translations.

In step (a), *Range Checking Logic* compares the requested virtual page number against the range of translations stored by each entry of the fully-associative TLB. As shown, comparator and adder logic is required for the range check. If the virtual page is detected in the range, there is a TLB hit. At this point (step (b)), the *PPN Generation Logic* subtracts the tag base virtual page number from the requested virtual page number. This value is then added to the stored base physical page number to find the desired physical page.

4.2.3. Replacement, invalidations, and attribute changes. We assume standard LRU for the fully-associative structure. Due to its smaller size, we suspect though that smarter replacement policies will be even more effective. Furthermore, we share the same attribute bits for all coalesced entries and invalidate entire entries, but for larger amounts of coalescing. Despite this, we will show that CoLT-FA performs effectively.

4.2.4. Discussion of hardware overheads. Generally, CoLT-FA hardware for range checks and physical page number generation is more complex than CoLT-SA. To account for this, we reduce the size of the fully-associative TLB in CoLT-FA as compared to the baseline case without coalescing. Commercial systems tend to implement 16 to 24-entry fully-associative TLBs for superpages [17]. To ensure that the added lookup complexity does not bias our results, we assume only 8-entry fully-associative TLBs with coalescing. While a detailed circuit-level analysis of the lookup overhead is beyond the scope of this work, our decision to apply CoLT to a half-sized fully-associative TLB attempts to maintain the same access times.

An additional overhead arises from the secondary scan performed between existing fully-associative TLB entries and the coalesced entry being filled. While one might initially assume that we may need to increase port counts to ensure that subsequent lookups are not delayed, our implementation retains just the single port. Instead, we assume that the initial lookup of the fully-associative TLB iden-

tifies those resident entries likely to be coalescible with the filled entry. Once the coalescing logic merges translations from a single LLC cache line, it then checks whether those resident entries can be further coalesced. In this way, a second TLB scan can actually be avoided, minimizing coalescing overheads.

4.3. CoLT-All Design and Implementation

Finally, CoLT-All coalesces into both set-associative L1/L2 TLBs and the superpage TLB. Its primary benefit over CoLT-SA and CoLT-FA is that it provides potentially the largest reach, at the expense of modifying both the set-associative and superpage TLB.

4.3.1. Overall operation. Figure 6 illustrates CoLT-All's operation when all the TLBs experience a miss. In step 1, the page walk has occurred and the coalescing hardware has determined the amount of contiguity present in the cache line. In then checks this contiguity to see how it compares to a threshold. If it is lower than a threshold (step 2), this means that the contiguity can be accommodated by the indexing scheme of the set-associative TLBs. For example, suppose the contiguity is three pages and we use an 8-set TLB with VPN[4-2] for indexing (allowing coalescing of up to four translations). In this case, the coalesced entry is allocated into the set-associative L1 and L2 TLBs. However the contiguity may be higher than the threshold and the amount that the set-associative TLBs can accommodate. In our example, the contiguity may be five. In this case, the entry is coalesced and brought into the superpage-TLB. At the same time, because the superpage-TLB is small, useful coalesced entries may be frequently evicted. Therefore, like CoLT-FA, we allocate an entry at this point into the L2 TLB as well. Unlike CoLT-FA however, our set-associative L2 TLB can now also handle coalesced entries (albeit with smaller levels of coalescing permissible by its choice of index bits). Therefore, CoLT-All brings in as much of this coalesced entry as possible into the L2 TLB, unlike CoLT-FA which brings just the requested translation. Finally, in step 4, the new allocated superpage entry may be coalesced with already-resident entries.

4.3.2. Lookup, replacement, invalidation, and attributes. While lookups operate similarly to CoLT-SA and CoLT-FA, the only difference is that it is possible for an entry to be resident in both the set-associative and fully-associative TLBs. While this occurs only rarely in practice, there are no correctness issues associated with this. Furthermore, there are no changes in replacement, invalidation, and attribute policies.

5. Methodology

We now detail the infrastructure and workloads used to quantify real-system contiguity and CoLT's effectiveness at leveraging this contiguity to eliminate TLB misses. Our analysis focuses on data pages since data references cause far more misses than instruction references [10, 27].

5.1. Real-System Characterizations of Page Allocation

5.1.1. Experimental platform and methodology. We use a system with a 64-bit Intel i7 processor, 64-entry L1 TLBs, and a 512-entry L2 TLB, a 32KB L1 cache, a 256KB L2 cache, a 4MB last-level cache (LLC), and 3GB of main memory. Furthermore, we run Fedora 15 (Linux 2.6.38).

To measure contiguity, we modify the kernel to scan the page table looking for instances of contiguous address translations. We walk the page table every five seconds, capturing contiguity changes through the benchmark run. Our original definition of contiguity is based only on page numbers; however, we now additionally require that contiguous translations must share the same page attributes and flags. While this eases the hardware implementation of CoLT by allowing for the same set of attribute bits per coalesced entry, contiguity would be even higher if this constraint were relaxed.

Benchmark	Suite	THS on L1/L2 MPMI	THS off L1/L2 MPMI
Mcf	Spec	56550/28600	95600/49230
Tigr	BioB.	19000/18150	26950/18860
Mummer	BioB.	12910/11450	14760/12970
CactusADM	Spec	6610/8140	8420/6930
Astar	Spec	8480/4660	17390/11240
Omnetpp	Spec	8410/2730	34040/8080
Xalancbmk	Spec	2670/2150	14120/2100
Povray	Spec	7010/630	7310/630
GemsFDTD	Spec	1300/620	8030/3620
Gobmk	Spec	710/410	1550/510
FastaProt	BioB.	460/300	610/300
Sjeng	Spec	1840/200	3860/440
Bzip2	Spec	4070/150	7120/270
Milc	Spec	120/90	3780/1820

Table 1: Summary of benchmarks used in our studies.

To study the effect of memory compaction, we use the Linux `defrag` flag. Enabling this flag triggers the memory compaction daemon both on page faults and as system background activity. Disabling this flag greatly reduces the number of times the memory compaction daemon runs. In tandem, we enable and disable THS to study the impact of superpaging. We also ensure that our system is realistically fragmented by using a machine that has already run a number of applications (eg. web browsers, network clients, office utilities) for two months. To further load the system, we run `memhog`, a memory fragmentation utility [12], with our workloads. We study scenarios where `memhog` fragments 25% and 50% (a highly fragmented system when combined with the other background activities) of the memory. In all, we study twelve system configurations. Due to space constraints, this paper focuses on:

1. THS on, normal memory compaction, no memhog: this is the current default setting for Linux.

2. THS off, normal memory compaction, no memhog: this shows contiguity *without* superpaging.

3. THS off, low memory compaction, no memhog: conservative case for contiguity because neither THS no memory compaction occur. The buddy allocator struggles to find contiguous physical blocks.

4. THS on, normal memory compaction, memhog: we test the effect of system load on the default Linux setting by assigning 25% and 50% of system memory to memhog.

5. THS off, normal memory compaction, memhog: shows the impact of fragmentation without superpaging.

5.1.2. Evaluation workloads. We study system contiguity on the Spec 2006 benchmarks [1] and bioninformatics workloads from Biobench [2] in Table 1. We run each of the workloads with their maximum data sets (for Spec, this corresponds to *Ref*) to completion. From the real-system runs, we use on-chip performance counters to track L1 and L2 TLB misses per million instructions (MPMI) when THS support is enabled and disabled. The benchmarks are ordered from highest to lowest THS on L2 TLB MPMIs. `Mcf`, `Tigr`, `Mummer`, `CactusADM`, and `Astar` see particularly high TLB MPMIs. While

enabling superpaging does reduce TLB misses for some workloads, it alone is insufficient. For example, `Mcf` still has an L2 TLB MPMI of 57K with THS on, while `Mummer` is unchanged.

5.2. Simulation-Based CoLT Evaluations

5.2.1. Simulated system. Past work on TLBs [5, 6, 9, 27] focuses on miss rates rather than performance because it is infeasible to run memory-intensive applications for long enough durations to provide performance numbers. We also study miss rates, but consider performance too. We use a two-step evaluation to quantify changes in hit rate and to then offer performance numbers feasible for simulations.

Like the bulk of recent work on TLB analysis, we first use a trace-based approach to analyze miss rates [5, 6, 9, 10]. We extract detailed memory traces by simulating an x86 processor on Simics [30]. These highly detailed traces maintain logs of both data and instruction references at the micro-op level. Our traces also capture full-system effects by running benchmarks on a Linux 2.6.38 kernel. We hack the simulated kernel to provide full page table walk details for every single memory reference (this includes the virtual page, the physical page, and all attribute bits). We set the kernel to its default configuration of using THS and normal memory compaction. As we will show, since contiguity is present across all kernel configurations, CoLT will be effective across the range of superpaging and memory compaction settings.

We run the traces through a highly-detailed custom memory simulator. We need to stress our TLBs using simulated workloads in a manner that matches real-system stress; therefore, we use 32-entry and 128-entry L1 and L2 4-way set-associative TLBs. These sizes are chosen as they produce simulated load within 10% of the load experienced by a real system. Our baseline system also assumes a 16-entry fully-associative superpage TLB. As previously detailed, CoLT-FA and CoLT-All reduce this size to 8 entries in order to provide conservative performance improvement data and negate the impact of slightly more complex lookups. Furthermore, unlike past work [9, 11], we model a more realistic TLB hierarchy with 22-entry MMU caches, accessed on TLB misses to accelerate page table walks [5]. Finally, we assume a three-level cache hierarchy similar to the Intel Core i7 (32KB L1 cache, 256KB L2 cache, 4MB LLC).

Having assessed miss rates, we now go beyond prior work and the study the performance implications of our approach. We use the Pin-based [20] CMP$im [18] simulation framework to model a 4-way out-of-order processor with a 128-entry reorder buffer. The processor's TLB and cache parameters match those of our custom trace module. Unfortunately, the simulation speeds of this detailed microarchitectural framework are slow; hence we cannot use it to run full Linux distributions with the memory allocation behavior necessary to study CoLT on sufficiently long-running, large-data applications. However, while this simulator does not maintain virtual-to-physical address translations, it does observe the performance effects of TLB misses by tracking the allocated virtual pages. In tandem with the miss rate eliminations extracted from our trace-based approach, this allows us to interpolate CoLT's actual performance gains. This interpolation strategy is valid for two reasons. First, TLB miss penalties (page walks) are serialized as only one page walk can typically be handled at a time [9, 11]. Hence, TLB misses lie on the execution's critical path. Second, our interpolation approach is actually conservative as it does not account for the instruction replays that would likely occur on TLB misses. Therefore, our projected performance benefits would likely *increase* on a real system.

5.2.2. Evaluation workloads. We use the workloads from Table 1. However, due to slow simulation speeds, we use Simpoints [24] that total to one billion instructions per workload. These simpoints include operating system effects captured by Simics and assume realistic inputs (for Spec, this corresponds to *Ref*).

6. Characterizations of Page Allocation Contiguity

We now quantify how the buddy allocator, memory compaction, THS, and system load affect application contiguity on a real system. We show that page allocation contiguity *always* exists regardless of the kernel configuration.

We begin by discussing the cumulative density functions (CDFs) from Figures 7 to 15. These graph the distribution of contiguities experienced by non-superpage pages. Note that contiguity (the x-axis) is presented as a log scale.

6.1. Superpaging, Memory Compaction

Figures 7, 8, and 9, ordering the benchmarks from highest to lowest TLB MPMI, show contiguity assuming default Linux kernel settings (superpaging and normal memory compaction). The legend provides average contiguity numbers.

Figures 7, 8, and 9 show that there is heavy contiguity across the workloads that cannot be exploited by superpages. On average, pages are in 41-contiguity groupings. Furthermore, there can be large instances of contiguity above the average. For example, most CDFs see many 64 to 256-contiguity instances.

Interestingly, there exist many cases of 512 and 1024-page contiguity. Since THS is enabled, one might initially expect that these should be treated as superpages. However, this contiguity does not translate to superpages for two reasons. First, these memory chunks are not superpage-aligned. Second, THS currently supports superpaging for only *anonymous* pages created through `malloc` calls; as such, a number of file-backed pages created from are not superpage candidates. Overall, we find that 15% of non-superpage pages actually have over 512-page contiguity.

Fortunately, Figures 7, 8, and 9 enjoy particularly high contiguity for TLB-stressing benchmarks. `Mcf`, `Tigr`, and `CactusADM` see tens to hundreds of contiguous pages, indicating their amenability to TLB coalescing. For a number of these benchmarks, such as `Mcf`, high contiguity arises because `malloc` and `mmap` calls are made at the beginning of the execution to allocate large hash-based data structures. These structures span megabytes of space, which the buddy allocator ensures maps to contiguous physical pages.

6.2. No Superpaging, Memory Compaction

Figures 10 to 12 show how contiguity changes when superpaging support is disabled. Average contiguity drops compared to THS on from 41 to 18, for two reasons. First, THS optimistically creates as many 2MB page as possible. While these 2MB pages eventually get broken into 4KB pages due to system load, they do leave large amounts of smaller, residual contiguity. Without THS, contiguity is not generated this way. Second, disabling THS drastically reduces memory compaction daemon invocations. Nevertheless, sufficient exploitable intermediate contiguity remains (in the tens of pages, around 18). Furthermore, heavy TLB-pressure benchmarks like `Mcf` and `Mummer` see very high contiguity.

Surprisingly, some benchmarks like `Omnetpp` and `Sjeng` actually see *higher* contiguity without THS. This occurs because the lack of THS reduces superpages allocated to *other* running processes. As a result, the pages allocated to our workloads remain unfragmented and contiguous.

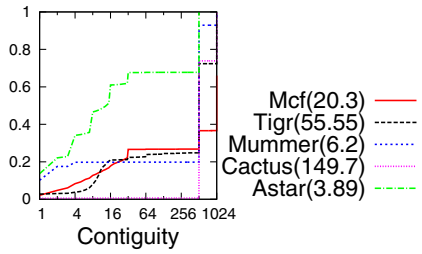


Figure 7: THS on, normal memory compaction contiguity CDF.

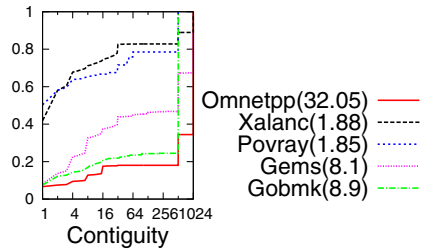


Figure 8: THS on, normal memory compaction contiguity CDF.

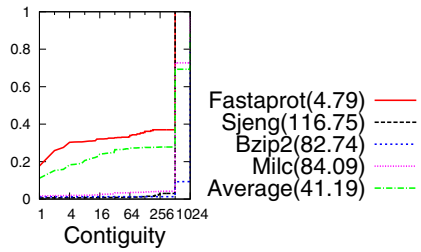


Figure 9: THS on, normal memory compaction contiguity CDF.

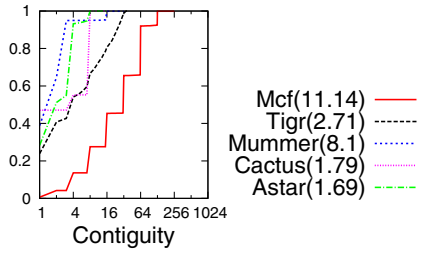


Figure 10: THS off, normal memory compaction contiguity CDF.

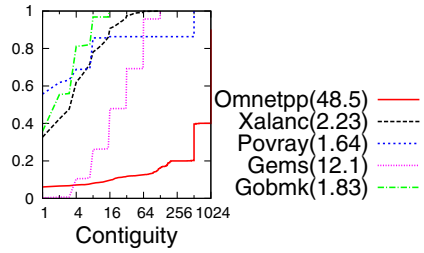


Figure 11: THS off, normal memory compaction contiguity CDF.

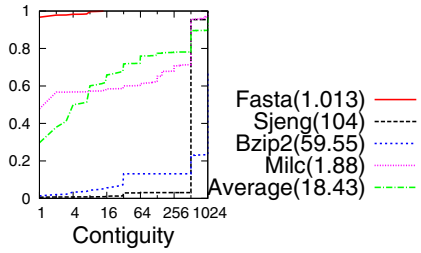


Figure 12: THS off, normal memory compaction contiguity CDF.

6.3. Superpaging, Low Memory Compaction

Figures 13, 14, and 15 present a worst-case scenario setting for Linux, where THS is turned off and memory compaction is greatly reduced via disabling the `defrag` kernel flag. While *no* kernel uses or recommends this setting, we study it to ensure that sufficient contiguity exists, even when there are almost no mechanisms to explicitly generate it. In fact, our results show that on average, contiguity drops only marginally compared to the THS off, normal memory compaction case to 15 pages on average. While important benchmarks like `Mcf`, `Mummer`, and `Omnetpp` do lose compared to the prior settings, they retain sufficiently high intermediate contiguity. For example, even though `Mummer's` average contiguity is now 1.3, roughly 50% of its 4KB pages enjoy 4-page contiguity. Correctly exploiting this gives our TLBs a 4× reach.

6.4. Superpaging, Memory Compaction, Memhog

We now focus on the impact of system load on fragmentation and contiguity. Figure 16 shows how contiguity is affected when `memhog` runs with each benchmark and fragments 25% and 50% of system memory. Our studies have shown that combined with the other running system processes, `memhog` with 50% heavily fragments almost all memory and causes page fault rates to greatly increase. We assume default Linux settings (THS enabled, normal memory compaction).

One might initially expect higher load to lower contiguity. Surprisingly, we find the *opposite* trend to hold when using `memhog(25%)`, with contiguity rising from 41 to roughly 43 pages on average. For some benchmarks, the gain is markedly high; for example, `Mcf` and `GemsFDTFD` contiguities are boosted by an order of magnitude. The primary reason for this is that higher load invokes the memory compaction daemon more often. This in turn provides the buddy allocator more contiguous physical blocks.

Greatly fragmenting the system with `memhog(50%)` however, does reduce contiguity. However, even this intermediate contiguity is relatively high, averaging close to 10 pages. For heavy TLB-pressure benchmarks like `Mcf` and `Mummer`, this configuration still achieves higher contiguity than without system load. As such, the buddy allocator, in tandem with memory compaction, manages to actually leverage the additional load to increase contiguity.

6.5. No Superpaging, Memory Compaction, Memhog

This represents the scenario where THS is turned off despite high system load. While kernel settings would not typically allow this, we use this setting to stress-test our measurements. We find that even under the pessimistic setting of no THS and `memhog(50%)`, the average contiguity is above 5. TLB coalescing can thus potentially provide a 5× reach.

6.6. Summary of Results

Three primary conclusions can be drawn from our real-system characterizations. First, under *every* single configuration, even those that are unrealistically severe, the buddy allocator, compaction daemon, and THS support succeed in inadvertently generating great intermediate contiguity. Second, system load can have surprising implications on contiguity, often increasing it. For some benchmarks that suffer from high TLB misses, such as `Mcf`, this is a promising observation. Third, superpages are ill-equipped to handle this contiguity. Therefore, coalescing techniques to harness this intermediate contiguity are warranted.

7. CoLT Evaluations

We now evaluate CoLT's benefits, focusing on per-application miss rate reductions and performance gains.

7.1. TLB Miss Rate Analysis

7.1.1. CoLT TLB miss rates. Figure 18 quantifies CoLT's TLB miss reductions. Benchmarks are ordered from highest to lowest TLB miss rates. We first capture the number of L1 and L2 TLB misses for a baseline configuration with 32-entry and 128-entry L1 and L2 TLBs (4-way) and a 16-entry superpage TLB. Note that we count misses for both the set-associative L1 TLB and the superpage TLB as L1 TLB misses since they are checked in parallel and have

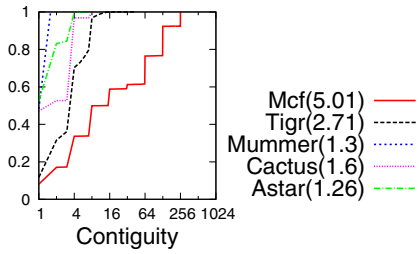


Figure 13: THS off, low memory compaction contiguity CDF.

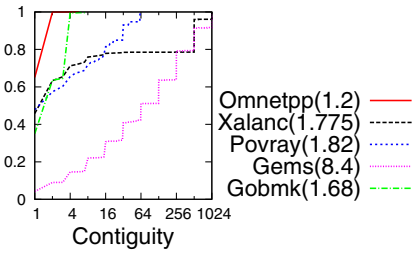


Figure 14: THS off, low memory compaction contiguity CDF.

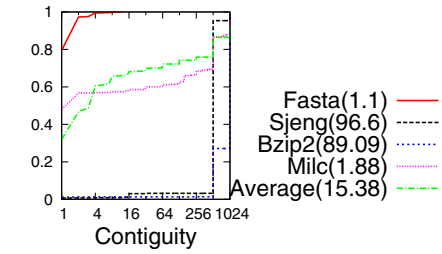


Figure 15: THS off, low memory compaction contiguity CDF.

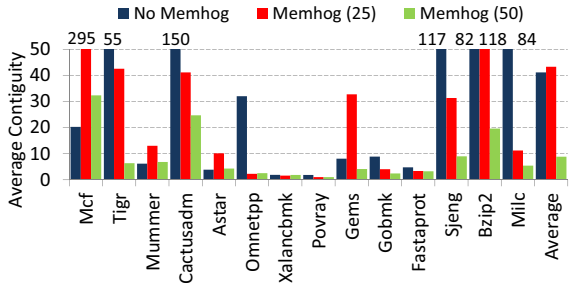


Figure 16: Average contiguity for THS on, normal memory compaction with varying `Memhog`.

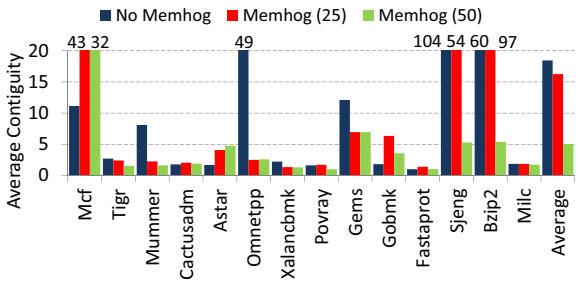


Figure 17: Average contiguity for THS off, normal memory compaction with varying `Memhog`.

the same hit time. After recording these misses, we then run the same benchmarks on configurations with CoLT-SA, CoLT-FA, and CoLT-All, tracking the new TLB miss rates. We assume that CoLT-SA uses VPN[4-2] and VPN[6-2] for L1 and L2 set selection, meaning that up to four translations can be coalesced per entry (we will later show the effect of using more aggressive indexing). We also conservatively assume 8-entry fully-associative TLBs when using CoLT-FA and CoLT-All.

First and foremost, Figure 18 shows that all three CoLT schemes improve *every* single benchmark by eliminating large chunks of the baseline misses. On average, CoLT-SA eliminates 40% of both L1 and L2 TLBs misses, while CoLT-FA and CoLT-All do even better, eliminating around 55% of both L1 and L2 misses. Second, Figure 18 shows that many of the benchmarks experiencing TLB pressure gain particularly from CoLT. For example, `Mcf`, `CactusADM`, and `Astar` all eliminate vast amounts of their TLB misses. In fact, `Astar` almost achieves perfect TLBs with no misses with CoLT-FA and CoLT-All.

Third, there is a correlation between system contiguity and effectiveness of CoLT. For example, `Mcf`, `Bzip2`, `Milc`, and `CactusADM`, which all see more instances of 20-page contiguity on average, can coalesce large amounts of translations, increasing TLB reach substantially. However, contiguity alone does not guarantee coalescing success; for coalescing to be effective, contiguous entries must actually be used close together in time. Without this temporal proximity, a coalesced entry will be evicted from the TLB before multiple member translations are used. This explains the lower benefits of `Tigr`, which sees 10% TLB miss elimination rates despite a contiguity of over 50 pages on average.

Fourth, Figure 18 shows that leveraging the superpage TLB in CoLT-FA and CoLT-All provides 10-15% gains over CoLT-SA on average. Benchmarks like `Mcf` and `Fastaprot` benefit particularly from this. These gains are achieved *despite* dropping from a 16-entry to an 8-entry structure. We find that the primary reason for this is that even with THS on, superpages are used sparingly. Therefore, a surprisingly high number of entries remain wasted in the fully-associative TLB in the baseline case. Instead, CoLT-FA and CoLT-All use these entries and can even perform unrestricted coalescing on them, unlike the set-associative TLBs.

The difference between CoLT-FA and CoLT-All remains more nuanced. We find generally that both schemes eliminate roughly 55% of TLB misses on average. Generally on the more TLB-intensive benchmarks (eg. `Mcf`, `Tigr`, `Mummer`, `CactusADM`), CoLT-All outperforms CoLT-FA slightly. However, in many benchmarks, CoLT-All falls surprisingly short of CoLT-FA. This occurs because CoLT-FA is better able to coalesce translations that reside across multiple LLC cache lines. Essentially, in these benchmarks only a few translations in a single cache line are coalescible with translations from another cache line. In CoLT-FA, since all the translations are brought into the fully-associative structure, these entries are coalesced. In CoLT-All however, if the cache line that maintains only a few coalescible translations falls below the pre-defined threshold, they are inserted into the set-associative TLB and can therefore never be merged with the second cache line's translations (which sits in the fully-associative TLB). This reduces CoLT-All's hit rates compared to CoLT-FA.

Overall, all CoLT designs eliminate a large fraction of TLB misses. We now focus on implementation details of the various CoLT designs to lend greater insight on our gains.

7.1.2. Impact of CoLT-SA's indexing scheme on TLB miss rates. Our initial CoLT-SA results assume that we use VPN[4-2] and VPN[6-2] for L1 and L2 set selection. This limits the amount of coalescing to four translations per entry. While additional contiguity could be coalesced by further left-shifting the index bits, this also increases conflict misses. Figure 19 studies these opposing forces on the 4-way associative TLBs by left-shifting the traditional index bits by one bit (VPN[3-1] and VPN[5-1] for L1 and L2 TLBs), two bits, and three bits (VPN[5-3] and VPN[7-3] for L1 and L2 TLBs). These correspond to maximum allowable coalescing of two, four and eight translations.

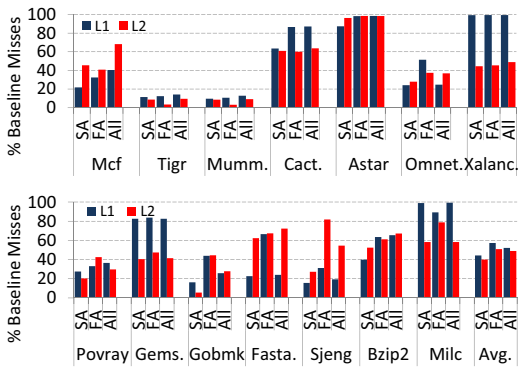


Figure 18: Percentage of baseline TLB misses eliminated using CoLT-SA, CoLT-FA, and CoLT-All.

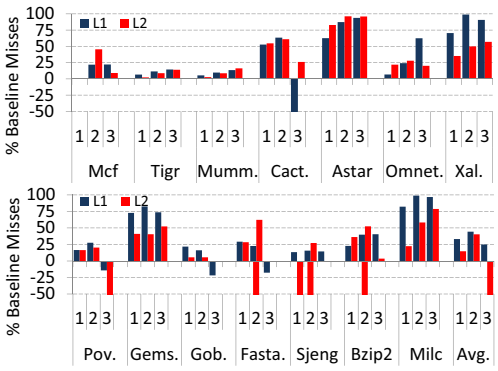


Figure 19: Percentage of baseline TLB misses eliminated by CoLT-SA when left-shifting the index by 1, 2, and 3 bits.

Figure 19 clearly shows that left-shifting the index bits by two provides the best balance between coalescing opportunity and conflict misses. Below this (left-shifting by one bit), we can only coalesce two entries, restricting our TLB miss elimination rates. However, left-shifting by three bits actually *increases* TLB misses in many cases due to the additional conflict misses. In general, unless there is very high contiguity, like for `Mummer`, `Tigr`, and `Milc`, left-shifting the index bits by three is overly-aggressive. Henceforth, we assume a left-shit of two bits for our indexing scheme.

7.1.3. Impact of bringing missing entries into L2 TLB for CoLT-FA and CoLT-All. As previously detailed, while CoLT-FA and CoLT-All bring coalesced entries into the fully-associative, superpage TLB, they also leverage the L2 TLB. For CoLT-FA, when a coalesced entry is brought into the superpage TLB, just the requested entry is also brought into the L2 TLB; for CoLT-All, a coalesced entry (where the coalescing amount is restricted by the index scheme) is brought into the L2 TLB. As we previously noted, this is useful since the superpage TLB is small (8-entry); as a result, only entries with high levels of coalescing are maintained there. As such, intermediate-level coalesced entries are often evicted. Bringing these entries into the L2 TLB as well increases the chance that these they remain available if necessary.

For CoLT-FA, we have run experiments to compare the case when (1) a coalesced entry is brought into the superpage TLB and just the translation triggering the coalescing is also brought into the L2 TLB, and (2) a coalesced entry is brought into the superpage TLB but the L2 TLB remains unaffected. We have found that on average, (1) outperforms (2) by an additional miss elimination of 10-15% for both L1 and L2 miss counts. We see particularly high gains with our approach on workloads with relatively lower contiguity such as `Povray`, since the small superpage TLB cannot coalesce a high enough number of entries to prevent eviction.

For CoLT-All, we have similarly run experiments to compare the case when (1) a coalesced entry is brought into the superpage TLB and its smaller coalesced version (a maximum of coalescing of four translations in our design) is brought into the L2 TLB, and (2) a coalesced entry is brought into the superpage TLB but the L2 TLB remains unaffected. We see again that our approach, (1) outperforms (2) by an average of 10-20% TLB miss eliminations.

7.1.4. Studying CoLT's effectiveness at higher associativities. We now consider CoLT effectiveness as TLB associativity is varied. A number of past studies have quantified how effectively increasing TLB associativity eliminates misses [13]. Generally, these studies have concluded that the slightly higher TLB hit rates are offset by huge power dissipation problems [7]. These observations are largely responsible for the relatively low associativity (typically 2-way or 4-way) supported on current TLBs.

CoLT, however, increases the benefits of higher set-associativity since the indexing scheme of the TLB can be more aggressively changed without as significant an increase in conflict misses. Overall, this allows higher levels of coalescing. Figure 20 compares how many L2 TLB misses in a 4-way 128-entry L2 TLB can be eliminated by CoLT-SA (4-way, CoLT-SA), by varying the associativity to 8-way but not allowing coalescing (8-way, No CoLT), and by allowing CoLT on the 8-way TLB (8-way, CoLT-SA). Note that all configurations use a fixed TLB size despite associativity changes.

First, Figure 20 shows that merely increasing the associativity to 8-way only eliminates 10% of the baseline L2 misses. In fact, even 4-way L2 TLBs with low-overhead CoLT-SA far exceed the benefits of higher associativity, eliminating 40% of baseline misses on average.

Figure 20 shows however, that the 8-way configuration augmented with CoLT-SA does provide significant benefits. CoLT-SA now eliminates 60% of the baseline misses on average, a substantial improvement over the other two scenarios. While a detailed power analysis is beyond the scope of this work, the performance, power ratio may therefore become more amenable with CoLT.

7.2. Performance Analysis

Up to this point, we have evaluated the benefits of CoLT in terms of miss rate eliminations. While this does indicate CoLT's effectiveness, we now focus on performance numbers which track how much faster each application runs with coalescing. Figure 21 details, for every benchmark, performance improvements from CoLT-SA, CoLT-FA, and CoLT-All. It also provides data on performance improvements that would occur with absolutely perfect, 100%-hit rate TLBs. The latter serves as a comparison point to determine how effectively CoLT performs. Once again, the baseline is a system with 4-way 32-entry and 128-entry L1 and L2 TLBs, and a 16-entry superpage TLB. CoLT-FA and CoLT-All conservatively reduce the superpage TLBs to 8 entries. Moreover, as previously detailed, we simulate a 4-way out-of-order processor.

Figure 21 shows that perfect TLBs would improve most benchmark runtimes by over 10% (eg. all except `Gobmk` and `Sjeng` in our benchmarks). In fact, `Xalancbmk` sees a huge 115% improvement in performance from TLBs that achieve 100% hit rate. These numbers indicate that TLB miss handling does significantly slow down bench-

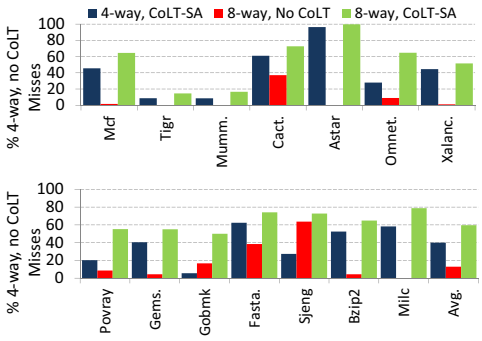


Figure 20: Percentage of baseline misses eliminated by CoLT-SA when increasing associativity.

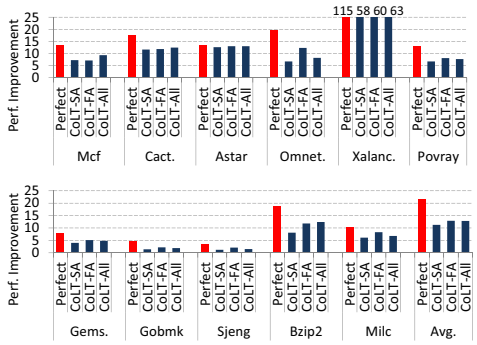


Figure 21: CoLT-SA, CoLT-FA, and CoLT-All performance improvements compared to perfect TLBs.

marks. This also implies that CoLT strategies have the potential to significantly improve performance.

Fortunately, Figure 21 shows that all of the CoLT approaches do indeed boost application performance significantly. On average, CoLT-SA achieves a 12% performance improvement, while CoLT-FA and CoLT-All achieve 14% improvements. On benchmarks like `Xalancbmk`, the performance improvements hover around 60% of runtime. Across other workloads like `Mcf`, `CactusADM`, `Astar`, `Omnetpp`, and `Bzip2`, at least one of the CoLT configurations improves performance over 10%. We anticipate that as applications with even larger working sets or virtualization are considered, these performance improvements will be even higher.

Figure 21 indicates that CoLT-SA, which has the simplest implementation, performs comparably to CoLT-FA and CoLT-All. Nevertheless, benchmarks like `Omnetpp` and `Bzip2` do see boosts from CoLT-FA/CoLT-All. Because we assume smaller 8-entry fully-associative TLBs, we expect CoLT-FA and CoLT-All results to be even higher with more realistically-sized superpage TLBs.

8. Conclusion

This paper proposes and designs Coalesced Large-Reach TLBs capable of exploiting address translation contiguity to achieve high reach. Due to a variety of OS memory management techniques involving buddy allocators, memory compaction, and superpaging, large amounts of translation contiguity are generated, even under heavy system load. While this contiguity typically cannot be exploited to generate superpages, CoLT provides lightweight hardware support to detect this behavior. As a result, large TLB miss eliminations are possible (on average, 40% to 58%), translating to performance improvements of 14% on average.

This work has a number of interesting implications for architects, system designers, and OS designers. We showcase TLB optimization techniques that architects can readily incorporate in existing processors. System designers and OS designers can tune their software systems to generate contiguity suitable for CoLT. For example, while superpages can overwhelm systems performance due to their overheads, CoLT provides alternate mechanisms to boost performance. We believe that CoLT will become even more critical as applications have increasingly large memory requirements and trends like virtualization become prevalent.

9. Acknowledgments

We thank the anonymous reviewers for their feedback. We also thank William Katsak for his help in modifying the Linux kernel for our studies. Finally, we thank Viji Srinivasan for her suggestions on improving the final version of the paper. The authors acknowledge the support of Rutgers University's Office of the Vice President for Research and Economic Development. This work was supported in part by their Faculty Research Grant award.

References

[1] "The Standard Performance Evaluation Corporation. SPEC CPU2006 Results," *http://www.spec.org/cpu2006*.
[2] K. Albayraktaroglu *et al.*, "BioBench: A Benchmark Suite of Bioinformatics Applications," *ISPASS*, 2005.
[3] AMD Corporation, "AMD Programmer's Manual," vol. 2, 2007.
[4] Andrea Arcangeli, "Transparent Hugepage Support," *KVM Forum*, 2010.
[5] T. Barr, A. Cox, and S. Rixner, "Translation Caching: Skip, Don't Walk (the Page Table)," *ISCA*, 2010.
[6] T. Barr, A. Cox, and S. Rixner, "SpecTLB: A Mechanism for Speculative Address Translation," *ISCA*, 2011.
[7] A. Basu, M. Hill, and M. Swift, "Reducing Memory Reference Energy with Opportunistic Virtual Caching," *ISCA*, 2012.
[8] R. Bhargava *et al.*, "Accelerating Two-Dimensional Page Walks for Virtualized Systems," *ASPLOS*, 2008.
[9] A. Bhattacharjee, D. Lustig, and M. Martonosi, "Shared Last-Level TLBs for Chip Multiprocessors," *HPCA*, 2010.
[10] A. Bhattacharjee and M. Martonosi, "Characterizing the TLB Behavior of Emerging Parallel Workloads on Chip Multiprocessors," *PACT*, 2009.
[11] A. Bhattacharjee and M. Martonosi, "Inter-Core Cooperative TLB Prefetchers for Chip Multiprocessors," *ASPLOS*, 2010.
[12] D. Bovet and M. Cesati, "Understanding the Linux Kernel," 2005.
[13] J. B. Chen, A. Borg, and N. Jouppi, "A Simulation Based Study of TLB Performance," *ISCA*, 1992.
[14] D. Clark and J. Emer, "Performance of the VAX-11/780 Translation Buffers: Simulation and Measurement," *ACM Transactions on Computer Systems*, vol. 3, no. 1, 1985.
[15] Z. Fang *et al.*, "Online Superpage Promotion with Hardware Support," *HPCA*, 2001.
[16] N. Ganapathy and C. Schimmel, "General-Purpose Operating System Support for Multiple Page Sizes," *USENIX*, 1998.
[17] Intel Corporation, "TLBs, Paging-Structure Caches and their Invalidation," *Intel Technical Report*, 2008.
[18] A. Jaleel *et al.*, "CMP$im: A Pin-based On-the-fly Multi-core Simulator," *4th Workshop on Modeling, Benchmarking, and Simulation*, 2008.
[19] G. Kandiraju and A. Sivasubramaniam, "Going the Distance for TLB Prefetching: An Application-Driven Study," *ISCA*, 2002.
[20] C.-K. Luk *et al.*, "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation," *PLDI*, 2005.
[21] C. McCurdy, A. Cox, and J. Vetter, "Investigating the TLB Behavior of High-End Scientific Appplications on Commodity Multiprocessors," *ISPASS*, 2008.
[22] D. Nagle *et al.*, "Design Tradeoffs for Software-Managed TLBs," *ISCA*, 1993.

[23] J. Navarro *et al.*, "Practical, Transparent Operating System Support for Superpages," *OSDI*, 2002.
[24] E. Perelman *et al.*, "Using SimPoint for Accurate and Efficient Simulation," *SIGMETRICS*, 2003.
[25] T. Romer *et al.*, "Reducing TLB and Memory Overhead Using Online Superpage Promotion," *ISCA*, 1995.
[26] M. Rosenblum *et al.*, "The Impact of Architectural Trends on Operating System Performance," *SOSP*, 1995.
[27] A. Saulsbury, F. Dahlgren, and P. Stenström, "Recency-Based TLB Preloading," *ISCA*, 2000.
[28] M. Talluri and M. Hill, "Surpassing the TLB Performance of Superpages with Less Operating System Support," *ASPLOS*, 1994.
[29] M. Talluri *et al.*, "Tradeoffs in Supporting Two Page Sizes," *ISCA*, 1992.
[30] Virtutech, "Simics for Multicore Software," 2007.

2012 IEEE/ACM 45th Annual International Symposium on Microarchitecture

NoRD: Node-Router Decoupling for Effective Power-gating of On-Chip Routers

Lizhong Chen Timothy M. Pinkston
Ming Hsieh Department of Electrical Engineering
University of Southern California
Los Angeles, CA
{lizhongc, tpink}@usc.edu

Abstract

While power-gating is a promising technique to mitigate the increasing static power of a chip, a fundamental requirement is for the idle periods to be sufficiently long to compensate for the power-gating and performance overhead. On-chip routers are potentially good targets for power optimizations, but few works have explored effective ways of power-gating them due to the intrinsic dependence between the node and router – any packet (sent, received or forwarded) must wakeup the router before being transferred, thus breaking the potentially long idle period into fragmented intervals. Simulation shows that directly applying conventional power-gating techniques would cause frequent state-transitions and significant energy and performance overhead. In this paper, we propose NoRD (Node-Router Decoupling), a novel power-aware on-chip network approach that provides for power-gating bypass to decouple the node's ability for transferring packets from the powered-on/off status of the associated router, thereby maximizing the length of router idle periods. Full system evaluation using PARSEC benchmarks shows that the proposed approach can substantially reduce the number of state-transitions, completely hide wakeup latency from the critical path of packet transport and eliminate node-network disconnection problems. Compared to an optimized conventional power-gating technique applied to on-chip routers, NoRD can further reduce the router static energy by 29.9% and improve the average packet latency by 26.3%, with only 3% additional area overhead.

1. Introduction

In recent years, power has become a critical design constraint, driving the microarchitecture design toward the paradigm of chip multiprocessors (CMPs). As a key component in CMPs, the network-on-chip (NoC) is the backbone for supporting communications among multiple cores. It is thus very important for NoCs to work efficiently and effectively to achieve both high performance and low power. However, recent studies show that NoCs can draw a substantial percentage of a chip's power, by up to 10%~36% [8, 9, 28]. In particular, the static power of routers has become a significant contributor of power consumption, consisting of more than 35% and 43% of the total NoC power at 45nm and 32nm processes, respectively (more details in Section 2). Unfortunately, the impact of static power will only get worse with continued scaling of transistor feature size and chip operating voltage.

Power-gating is a useful circuit-level technique applicable to power-aware architectures to mitigate the increasing static power, especially for circuit blocks that exhibit enough idleness [10, 19]. While power-gating in general is a promising technique, applying it directly to on-chip routers has been elusive as doing so requires several fundamental problems to be addressed in order to maximize energy-savings and minimize performance penalties. First, intermittent packet arrivals may cause a large number of idle periods to fall below the breakeven time needed to compensate for power-gating overhead, reducing the opportunity to apply power-gating techniques usefully. Second, packets encountering gated-off routers suffer additional transport latency to wait for routers to wake up and are likely to experience successive wakeup latencies on the critical path if routed over multiple hops. Third, as a gated-off router essentially disconnects the associated node from the rest of the network, the power-gating opportunity is upper bounded by the local node's traffic and none of the local resources (e.g., cache and directory) can be accessed by other nodes, unless connectivity is somehow supported another way. Without solving these fundamental problems, the effectiveness of applying power-gating to on-chip routers is severely limited.

The above problems are all caused by node-router dependence – whether a node can send, receive or forward a packet depends directly on the on/off status of the associated router of that node. In this paper, we propose *NoRD (Node-Router Decoupling)*, a novel approach that provides separate power-gating bypass to decouple the node's ability for transferring packets from the status of the router. This approach avoids unnecessary router wakeups and, more importantly, the associated performance penalty and energy overhead. NoRD effectively increases the length of idle periods, removes wakeup latency from the critical path, and eliminates power-gating disconnection problems. The main contributions of this paper are the following:

- Fundamental and critical problems of applying conventional power-gating techniques to on-chip network routers are identified;
- The concept of Node-Router Decoupling and a power-gating bypass technique to implement NoRD are proposed, which provides a unified and effective solution to the aforementioned problems;
- Full system simulations show a significant improvement in the use of power-gating with NoRD as compared to directly applying power-gating with conventional techniques.

The rest of the paper is organized as follows. Section 2 provides more background on the static power of routers and the power-gating technique. Section 3 highlights the problems of power-gating on-chip routers and motivates the need for a better approach. Section 4 explains the details of the proposed NoRD design. Section 5 discusses our evaluation methodology, and Section 6 presents simulation results. Finally, related work is summarized in Section 7, and Section 8 concludes the paper.

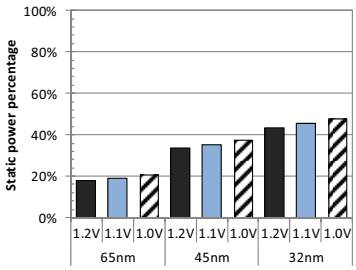
2. Background

2.1 Static Power of On-chip Routers

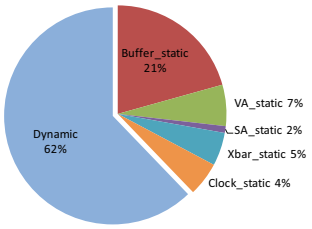
The static power of CMOS circuitry has been increasing substantially in recent years due to the continued scaling of transistor feature size and chip operating voltage. As a major

1072-4451/12 $26.00 © 2012 IEEE
DOI 10.1109/MICRO.2012.33

IEEE computer society



(a) Static power percentage



(b) Router power decomposition

Figure 1: Static power vs. dynamic power of on-chip routers.

component of multicore chips, on-chip networks consume around 10%~36% of a chip's power, as shown in recent industrial and research chips [8, 9, 28]. A considerable amount of NoC power comes from static consumption. To study the significance of NoC static power and the impact of technology scaling, Figure 1(a) plots the percentage of static power of on-chip routers at 3GHz for various manufacturing generations and operating voltages. Results are obtained from the Orion 2.0 [13] power model. To reflect realistic workloads, Orion is fed with statistics from full system simulation – Simics [21] plus GEMS [22] – running multi-threaded PARSEC 2.0 benchmarks [2] (more details of the simulation infrastructure are described in Section 5). As shown in the figure, the percentage of static power consumption increases as the feature size and operating voltage decrease, from 17.9% at 65nm and 1.2V, to 35.4% at 45nm and 1.1V, to 47.7% at 32nm and 1.0V. This trend clearly illustrates that the static power of on-chip routers has become a significant part of the overall router power consumption, and only worsens for process technologies beyond 45nm. Figure 1(b) further breaks down the total power consumption of on-chip routers at 45nm with 1.0V into dynamic and various static components. As can be seen, buffers consume 55% of the static power (21% of the total power) while other router components consume 45% of the static power (17% of the total power). This indicates that the static power consumption in router components other than buffers is significant and that appropriate techniques need to be adopted to reduce all contributors to static power.

2.2 Power-gating Techniques

One of the most effective techniques to mitigate the static power of a circuit block is power-gating as it cuts off the power supply of that block, which is the source of leakage currents in both subthreshold conduction and reverse-biased diodes. It is implemented by inserting appropriately sized header or footer transistor(s) with high threshold voltage (non-leaky "sleep switch") between Vdd and the block, or the block and GND, as illustrated in Figure 2(a). By asserting or de-asserting the sleep signal, the supply voltage to the power-gated block can be turned on and off.

Figure 2(b) depicts the key intervals of power-gating. At time t_0, the sleep signal is asserted and distributed to the sleep transistor with certain overhead energy. At t_1, this signal arrives at the sleep transistor and turns it off, so the virtual Vdd starts to drop. Correspondingly, the leakage current also decreases and the cumulative energy savings start to increase. From this moment, the block stays in the power-gated off state until t_2 when the sleep signal is de-asserted and distributed again, initiating the wakeup process. From t_2 to t_3, another energy overhead is incurred in distributing the sleep signal and waking up the gated-off block. The cumulative energy savings stop increasing at t_3 when the virtual Vdd restores to full Vdd and the wakeup process concludes. Consequently, an important parameter in power-gating is the "breakeven time" (BET), which is defined to be the minimum number of consecutive cycles that a gated block needs to remain in idle state before being awoken to offset power-gating energy overhead [19, 20]. Prior research using analytical modeling and simulation [10, 23] estimate the BET value to be around 10 cycles for functional units and on-chip routers under current technology parameters.

2.3 Use of Power-gating

Although power-gating can reduce power, it can also reduce system performance. This is because a powered-off block cannot perform the assumed functions temporarily, and waking up the block takes an additional wakeup delay, thus potentially stalling system progress. Therefore, effective use of power-gating should achieve two objectives in a balanced way:

(1) Maximize net energy savings, which means to maximize the idleness of unneeded functional blocks in order to increase the cumulative energy savings while reducing the associated energy overhead as much as possible;

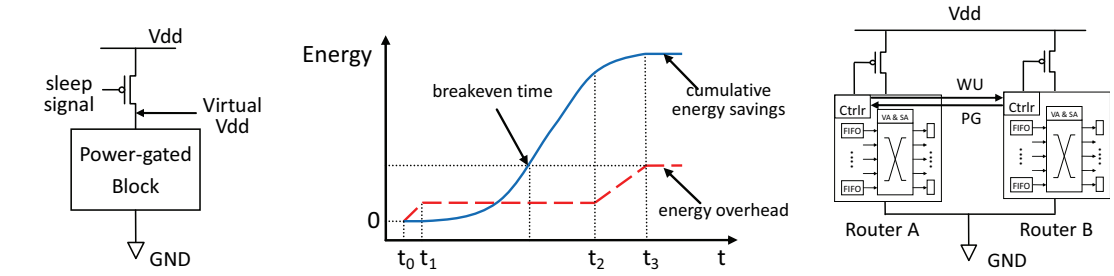
(2) Minimize performance penalty, which means to partially or completely reduce/hide the wakeup latency of needed functional blocks, so that execution can continue with minimal delay.

While power-gating has been used successfully in cores and execution units [10, 19, 20], only recently has research started to investigate its application to on-chip network routers [23, 24, 25]. However, as discussed shortly in the next section, due to the node-router dependence in on-chip networks, the conventional way of power-gating routers is ineffective in achieving the energy and performance objectives. Several fundamental and critical problems must be addressed to mitigate costly frequent state-transitions and performance overhead that comes with applying the conventional technique.

3. Motivation

3.1 Conventional Power-gating of On-chip Routers

The on-chip network is responsible for connecting the various components within a CMP, where each node may consists of a processor core, caches, and an associated router. *Node-router dependence* means that the ability for a node to send, receive or forward a packet depends directly on the on/off status of the associated router. For example, a node can inject a packet into the network only when the associated router is in the powered-on state. Conversely, routers become idle when the associated nodes have no packet to send, receive or forward. Our full system simulation results show that on-chip routers can be idle 30%~70% of the time



(a) Power-gating technique (b) Energy vs. time (c) Power-gating of on-chip routers

Figure 2: Power-gating technique and its application to on-chip routers.

(with *x264* having the lowest of 30.4% and *blackscholes* having the highest of 71.2%), depending on the physical location of the routers in the NoC and the load intensity of the applications. Therefore, power-gating techniques can be applied to on-chip routers to take advantage of their idleness.

When the internal datapaths of a router are empty (i.e., input ports, output latches, and the crossbar), the router microarchitecture can be power-gated off to save static power after notification of all its neighbors. Figure 2(c) shows an example of power-gating router *B* and handshaking with one of its upstream routers, router *A*. A canonical wormhole router [3] is assumed, which consists of routing computation (RC), VC allocation (VA), switch allocation (SA) and switch traversal (ST), with another stage of link traversal and buffered writing (LT). A small non-power-gated controller is added in the router to monitor the emptiness of the datapath and the wakeup signals from neighbors. When the datapath of router *B* is detected as empty and the *WU* (wakeup) signals are clear, the controller in router *B* asserts a sleep signal to put router *B* into gated-off state and asserts a *PG* (power-gate) signal to notify router *A*. Upon detecting the asserted *PG* signal, router *A* tags the output port that leads to router *B* as being power-gated and hence becomes unavailable in the SA stage[1]. Later, after router *B* is power-gated, some packet in router *A* or another neighbor of router *B* may request an output port to router *B* in the SA stage, triggering the *WU* signal to be asserted which causes the controller in router *B* to de-assert its sleep signal. The packet will then be stalled in the SA stage while waiting for router *B* to wake up and de-assert the *PG* signal. According to previous studies [23, 25], the wakeup latency for on-chip routers under typical technology parameters is a few nanoseconds, or around 10~20 cycles depending on the frequency. In what follows, we use the term *conventional power-gating* of routers to refer to the above mechanism of applying conventional power-gating to on-chip routers.

3.2 Intensified BET Limitation

A major obstacle to achieving effective power-gating of on-chip routers is the intensified limitation caused by breakeven time (BET). It has been observed that, when applying power-gating to functional units, the BET limitation may cause large energy penalty for some applications where functional units do not exhibit long enough idle periods [19]. Unfortunately, when applying conventional power-gating to on-chip routers, the BET limitation becomes much more prevalent due to intermittent packet arrivals seen by the routers. Figure 3 illustrates the problem even in the case where the NoC has substantial idleness, as given by a low average arrival rate of 0.1 flits/cycle (i.e., 10% traffic load). In (a), with two successive single-flit packets arriving in the first two cycles, the router has up to 18 idle cycles for useful power-gating; whereas in (b), discrete packet arrivals cut down idle periods to below the BET, leading to an energy *penalty* as opposed to savings if power-gated. Our evaluation on PARSEC benchmarks shows that the number of idle periods having a length less than or equal to the BET constitutes more than 61% of the total number of idle periods. Thus, on the one hand, routers on average exhibit very good idleness that could benefit from applying power-gating, but on the other hand, a large percentage of these idle periods are too short to meet the BET requirement as any sending, receiving or forwarding operation of a node would generate packets for the associated router to process, thus severely limiting the effectiveness of conventional power-gating of routers.

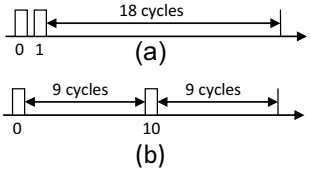


Figure 3: Intermittent packet arrival.

One direct way to address this problem is to reduce the BET through better circuit-level design or advanced manufacturing processes, which unavoidably have physical limitations (e.g., transistor sizing of the inverter-chain has limited ability in mitigating the energy overhead of sleep-signal distribution). Another possibility is to apply conventional power-gating to smaller individual components within each router, such as per input port or per virtual channel [24, 25]. This method, however, can only mitigate the impact of the BET problem as individual components have only slightly longer idle period, and even if the BET condition is satisfied, many power-gated cycles are wasted to offset the energy overhead. Moreover, this requires prohibitive hardware implementation overhead. For example, there are 35 power domains in a single router in [25] to implement this method of power-gating in addition to the complex coordination needed among different components, which incurs significant energy and area overhead with considerable design effort. Thus, a much more effective way of removing the dependence between the node and router is needed, so as to combat the BET limitation from the source by reducing the number of wakeups while maintaining the ability to transport packets in the NoC.

[1] To ensure the receiving of packets that are already in ST and LT stages, either router *B* needs to wait two more cycles before deciding to enter gated-off state, or *WU* should be generated early enough.

3.3 Cumulative Wakeup Latency in Multi-hop Networks

Just as the BET limitation of energy-savings is magnified in power-gated on-chip routers, the wakeup latency problem is also exacerbated in NoC environments, which affects performance negatively. Due to the node-router dependence, conventional power-gating of routers requires routers to be in on-state to forward packets, which makes the wakeup latency exposed directly to the critical path of packet transport to downstream routers. A packet routed in a multi-hop NoC can experience wakeup latency multiple times as routers at many hops along the path could be gated-off. To make things worse, power-gating works best when load rates are low, but in those situations more routers are in the gated-off state, making packets more likely to encounter multiple wakeups. One approach is to use early wakeup signal generation (e.g., generate the wakeup signal as soon as the output port is computed). However, this has limited ability to hide router wakeup latency, e.g., 3 cycles maximum out of the 10~20 cycles of wakeup latency for a 4-stage pipeline. Look-ahead wakeup is also possible [23, 25], in which the candidate router monitors all the wakeup signals two hops away so that it can hide at most 6 cycles of wakeup latency. This still limited technique requires monitoring hardware that is very complex and expensive to implement as every router essentially has to monitor every input port in up to 12 routers within a 2-hop distance, assuming a 2-D mesh topology. A much better approach would be to effectively *remove* the wakeup latency from the critical path by providing bypass of powered-off routers, as proposed in Section 4.

3.4 Disconnection Problem

The third major and most obvious problem in applying conventional power-gating to on-chip routers is the network disconnection problem. This problem is caused also by the node-router dependence, as whenever a router is power-gated off, the associated node is disconnected from the rest of the network. The disconnection problem impacts system in two ways. First, the local node cannot send/receive packets to/from the network if the associated router is powered-off, which limits the opportunity of power-gating to only those cases when the core and cache associated with the node are completely idle. Second, remote nodes cannot access any resource on the local node either, particularly the cache line and coherence directory. For a typical shared last level cache (LLC) configuration, this essentially decreases the effective cache size. For example, if half of the routers are power-gated off, the accessible LLC size available to the remaining nodes is reduced by 50%. Especially worth noting is that a private LLC does not help much due to the maintaining of cache coherence protocols. For instance, a dirty line in the private LLC of the local node is the unique last copy of the data in the entire system. Any other request to this line from remote nodes must wakeup the local router to access the data and resume correct execution, even if the local core is idle. Therefore, a more effective way to circumvent powered-off routers and maintain the connectivity of on-chip resources using some alternative path is needed.

4. Proposed Scheme: NoRD

In this section, we propose *NoRD (Node-Router Decoupling)*, a novel approach that removes the intrinsic dependence between nodes and routers, solving all the aforementioned problems unaddressed by conventional power-gating of on-chip routers.

4.1 The Basic Idea

The proposed approach is based on the simple idea of breaking node-router dependence via wakeup-avoidance decoupling bypass paths. Recall that in conventional power-gating of routers, due to the node-router dependence, any incoming packet from either a local node or other nodes would first have to wake up the gated-off router before further packet transport could occur. This wakeup incurs energy overhead and performance penalty on each occurrence. By providing decoupling bypass for each router, the ability to transport packets in the network is decoupled from the on/off status of the routers. This solves all three problems of conventional power-gating of routers. First, packets (sent, received or forwarded) have the option to go through bypass paths instead of powering-on the routers to continue progress, thus avoiding unnecessary wakeups and the associated energy overhead which causes BET in the first place. Second, bypass allows packets to be transferred *while* the router is being awoken, which removes the wakeup latency completely from the critical path of packet transport. Third, when the associated router is powered-off, the local node can still be connected with the rest of the network through the decoupling bypass paths, thus eliminating the disconnection problem.

While NoRD conceptually is a simple yet attractive solution, implementing decoupling bypass that provides chip-wide connectivity even when many or all routers are gated-off and transition between the gated-on/off state is not straightforward. In the proposed design, we add internal bypass paths in each router that can forward packets directly from a selected input port to the network interface (NI) and then forward the packets from the NI back to a selected output port. The input/output port pairs from all routers form – in the worst case – a unidirectional ring across the chip, so that all the NIs are always connected. The resulting bypass paths, together with all remaining paths provided by the normal deadlock-free routing algorithm, allow packets to be transported without deadlock in NoCs comprised of any combination of powered-on and powered-off routers. In the rest of this section, we present the detailed design of NoRD, addressing the construction of bypass paths, the implementation of NI forwarding, the transition and interface between routers in bypass mode and normal mode, the avoidance of deadlock and other network abnormalities under the presence of both on and off routers, and asymmetric wakeup threshold to further increase the efficiency of NoRD.

4.2 Decoupling Bypass

Without loss of generality, we start by describing the microarchitecture of bypass using a 4x4 2D mesh as an example. Decoupling bypass is achieved through two-level coordination. At the chip level, an input port (referred to as a Bypass Inport) and an output port (referred to as a Bypass Outport) from each router are chosen in a way such that, collectively across the network, they form a unidirectional ring (referred to as Bypass Ring) connecting all nodes, as shown in Figure 4(a). At individual router level, two datapaths are added as follows. In order to inject packets from the local node (e.g., processor core), a datapath is added from the NI input to the Bypass Outport (the bottom bold line in Figure 4(b)). In order to receive packets destined to the local node from the network, a second datapath is added from the Bypass Inport to the NI outport to eject packets from the router (the top bold line in Figure 4(b)). The bypass paths consisting of minimal hardware described here are not power-gated.

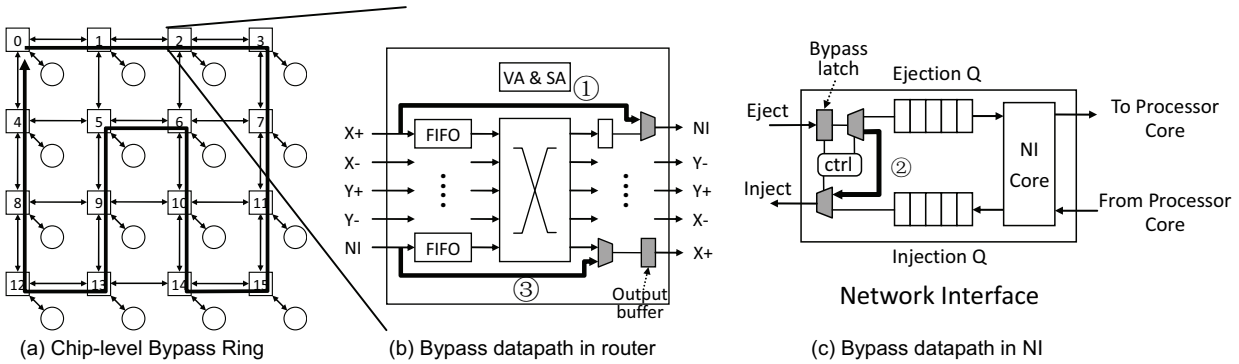


Figure 4: Decoupling bypass (shaded components in (b) and (c) are not power-gated).

To forward packets through a powered-off router, a bypass path from the router's Bypass Inport to its Bypass Outport is established through the node's NI. Flits are ejected from the powered-off router to the NI and injected back into the same router along the path of the Bypass Ring, as shown in Figure 4(c). In a typical NoC with wormhole switching, the NI is responsible for accepting data from the node and encapsulating it into packets and flits (NI core), allocating a virtual channel and checking flow control credits in the NI input port of the associated router, and injecting the formatted flits into the network. Receiving data from the network to the node has a similar but reversed process. Now, to implement router bypassing through the NI of the node, we add a latch and a demultiplexer ahead of the ejection queue, insert a multiplexer after the NI's injection queue, and create a path between the input and output ports of the NI according to Figure 4(c). With this forwarding path, a flit can now be forwarded from the gated-off router's Bypass Inport to its Bypass Outport in three stages, as annotated in Figure 4(b) and (c): ① at the end of link traversal, instead of being written into the router's input buffer as done when the router is powered-on, the flit is written directly into the NI bypass latch through the bypass datapath; ② based on the packet's destination header bits, the NI either sinks this flit in the local node or forwards the flit by allocating a VC (and checking its credits); ③ the flit is re-injected into the power-gated router's Bypass Output through the bypass datapath. The bypass datapath is enabled only when the router is in the power-gated off state.

The above two-level coordination essentially decouples nodes from the on/off status of routers, as now a node can send, receive and forward packets through the decoupling bypass even if the associated router is in the gated-off state. Moreover, it ensures the connectivity of all nodes. Packets can route through a combination of Bypass Ring paths to circumvent gated-off routers and normal paths of gated-on routers to minimize hop count. Even in the extreme case of all routers being gated-off, packets can still traverse along the Bypass Ring to reach any destination.

Owing to the decoupling bypass that provides network connectivity in all cases, deadlock-free adaptive routing based on Duato's Protocol [4] is easily supported. Escape resources are comprised of the unidirectional ring formed by the (Bypass Inport, Bypass Outport) pairs in both gated-on and gated-off router state, where two VCs can be used to break cyclic dependence. Additional VCs can be used as adaptive resources for adaptive routing over the NoC.

The deadlock- and livelock-free routing of NoRD is as follows. Every router has adaptive VCs and escape VCs (powered-off routers have no VCs but still have the corresponding adaptive/escape latches for bypassing). At normal routers, packets on adaptive VCs use minimal adaptive routing to choose the next hop, but packets on escape VCs are confined to choose the Bypass Outport (i.e., move along the bypass ring) and confined to escape VCs until destination. For packets on adaptive VCs, misrouting occurs only when all of the downstream routers on the minimal path are powered-off AND the Bypass Outport forces a detour (note that the Bypass Outport could, in fact, also be on the minimal path). In that case, packets must choose the Bypass Outport to traverse to next router (could be either normal or off) misrouted by one hop. However, packets are still allowed to remain on adaptive VCs for normal routers or the corresponding adaptive latches for bypassed routers (i.e., the entire set of adaptive resources) if the total misrouted hops are below a threshold; otherwise packets are forced to enter escape VCs (or the corresponding escape latches for bypassed router) and route along the unidirectional ring without returning to adaptive resources until the destination is reached. At the next router, if packets are still on adaptive VCs, they will repeat the above process (i.e., use minimal adaptive routing if available on the bypass ring or mesh, or enter escape resources on the Bypass Ring if needed) until reaching the destination. No U-turns are allowed at any hop. The above routing for NoRD follows Duato's Protocol for deadlock-free adaptive routing as the escape VCs on the Bypass Ring have no cycles in the extended channel dependence graph and the adaptive channels allow for fully adaptive routing. As detoured packets have a cap on the number of misroutes allowed before being forced to enter escape VCs with a bounded hop count, NoRD avoids both deadlock and livelock. Also, any additional hops from detours are partially offset by gains in completely hiding router wakeup latency as compared to conventional power-gating and reduced per hop latency of the bypass path. Finally, starvation for NI resources by the local node is easily avoided by granting priority over bypass traffic to the local node if not served for a predetermined number of consecutive cycles. However, this should happen rarely as the router is assumed to be power-gated off only when the load is low and contention is minimal.

4.3 Transition between Gated-on and Gated-off States

To transition between gated-on and gated-off states and to interface with neighboring routers for correct flow control, several handshaking signals are needed as illustrated in Figure 5. In this example, we focus on the state-transition of router *B*, and the bypass of router *B* is from router *A* through the NI of router *B* to router *D*.

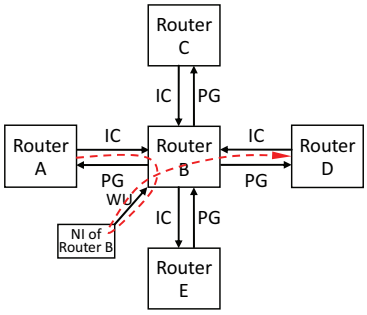


Figure 5: Handshaking in NoRD. PG: power-gate, WU: wakeup, IC: incoming

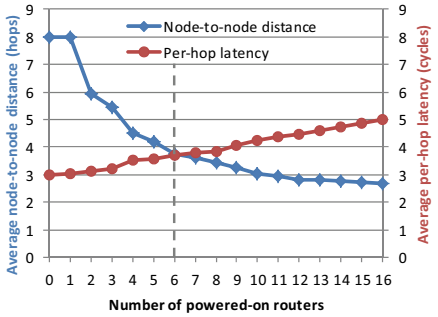


Figure 6: Impact of powering-on routers.

To transition from gated-on to gated-off state, similar to the conventional power-gating mechanism described in Section 3.1, if router *B* is empty and both *IC* and *WU* are clear (these two signals will be explained shortly), it asserts the *PG* signals, enables bypass and goes into gated-off state by asserting the sleep signal (not shown). Upon detecting the asserted *PG* signal, routers *C*, *D* and *E* tag the output port that leads to router *B* as power-gated (and becomes unavailable in the SA stage) and stop tracking credits, while router *A*, which is the Bypass Ring upstream router, sets the credit of each VC in that output port to 1 as router *B* now has only one output buffer available as shown in Figure 4(b). To ensure the receiving of packets that are already in the ST and LT stages of the neighboring routers, an *IC* (incoming) signal is generated at the beginning of SA if there is a flit in the SA stage and propagates to router *B*. In this way, the *IC* signal is always two cycles ahead of flits to notify router *B* that a flit is incoming and router *B* should not enter into gated-off state. Finally, for any flits that are in the VA and SA stages of routers *C*, *D* and *E*, they will restart the pipeline from RC using the new output port availability information as they are still in the input channel. Note that these flits must be head flits; otherwise if the head flits have left router *C*/*D*/*E* to *B* but body/tail flits have not yet arrived at router *B*, then the virtual channel is not de-allocated and router *B* is not considered as empty.

To transition router *B* from gated-off state to gated-on state, the *WU* signal first needs to be generated according to a wakeup metric. Ideally, the wakeup metric should de-assert *WU* when the load is low, and assert the signal when it is above a threshold when the load becomes high. A naïve way is to use the number of flits transmitted by the gated-off router in a fixed period of time, but this may not necessarily generate a wakeup signal when the load is high as flits could be stalled due to network congestion. Another traditional metric is to use router buffer utilization [27], which also is not suitable as input buffers are not used in the gated-off state. As all traffic to gated-off routers are forwarded through the NI and allocated a VC there to (re)inject into the network, we use as a threshold parameter the number of VC requests at the local NI over a period of time (10 cycles) for the wakeup metric. This metric works for both low and high load as the number of VC requests goes up even if the flits are stalled, and it remains valid in the extreme case when all the routers are gated-off, as the wakeup signal is generated locally.

With the number of VC requests used as threshold wakeup metric, the operation of turning on a gated-off router is straightforward. When the *WU* signal is asserted, router *B* starts to wake up while the bypass is still functioning. When wakeup finishes, router *B* de-asserts the *PG* signal. Upon detecting the de-asserted *PG* signal, routers *C*, *D* and *E* reset the credits to full while router *A* adds back (full-1) credits. Once the flit in the NI bypass datapath is written into the input buffer of router *B*, the bypass of router *B* is disabled to complete the state-transition.

4.4 Asymmetric Wakeup Thresholds

While previous subsections describe the necessary operations to keep NoRD functional, the efficiency of NoRD can be increased using *asymmetric wakeup thresholds*. For certain topologies and constructions of the Bypass Ring, some routers may have greater impact on performance than others based on their location in the NoC. For example, powering on Routers 4 and 5 in Figure 4(a) has larger performance benefits than powering on Routers 0 and 1, as the former provide a shortcut to route packets that would otherwise be detoured through 9->13->12->8. Therefore, taking the placement of bypass paths and routers into account, additional performance gains can be obtained.

To differentiate between routers in NoRD, asymmetric wakeup thresholds can be used. For example, NoC routers can fall broadly under two classes – *performance-centric* and *power-centric* – based on their importance, where a low wakeup threshold is assigned to the *performance-centric* class and a high wakeup threshold is assigned to the *power-centric* class. The intuition behind this is to wake up early a few performance-critical routers while waking up late the rest (majority) of the routers. In this way, not only performance improves due to the added shortcuts in routing paths, but also more static power can be saved by allowing non-performance-critical routers to stay in the gated-off state for a longer time. As a threshold metric is needed for wakeup anyway, no additional hardware is required.

To select the set of routers that are more critical to performance, we wrote a short off-line program based on the Floyd-Warshall all-pair shortest path algorithm [7]. Figure 6 plots the best node-to-node average distance and per-hop latency that can be achieved with a given number of powered-on routers for the 2-D mesh example in Figure 4(a). As expected, with more routers turned on, the average hop distance between nodes in NoRD decreases rapidly due to the added flexibility in routing paths. Meanwhile, more packets are routed through the normal pipeline of powered-on routers instead of the simpler and shorter bypass pipeline, thus gradually increasing the per-hop latency. Figure 6 also shows that, by turning on six routers, the average hop distance can be greatly reduced with moderate increase in the per-hop latency, indicating a viable trade-off point. The corresponding router set that achieves this data point consists of Routers 4, 5, 6, 7, 13 and 14 in Figure 4(a). In this example, these routers are designated as the *performance-centric* routers, and the remaining routers are classified as the *power-centric* routers. Other classifica-

tions are still possible and an optimal classification could be determined dynamically with comprehensive consideration of topology, traffic patterns, bypass placement, and routing algorithm. For instance, the routing algorithm may adaptively steer packets to a few *performance-centric* routers and the rest of the routers can be designated as *power-centric* routers. While further work can be conducted to investigate the design complexity of finding the optimal classification and the trade-off in doing so, this falls outside the scope of this paper. Here, we intend only to show that asymmetric wakeup threshold, even with a simple dual-mode classification, can provide additional benefits in both performance and energy to complement the proposed decoupling bypass mechanism.

4.5 Impact of NoRD on Energy and Performance

We mentioned before that there are two primary objectives when using power-gating techniques. Here, we analyze the impact of NoRD on achieving these two objectives to highlight the benefits of the proposed decoupling approach.

Impact on Net Energy Savings

NoRD maximizes the opportunity for saving energy by allowing fragmented idle periods that are even shorter than the BET to be exploited, which is not possible in conventional power-gating of routers. Moreover, by steering short packet spikes to bypass paths without waking up the routers, the energy overhead in distributing the sleep signal and powering-on the router is also largely avoided. Therefore, NoRD is able to increase the cumulative energy savings while reducing the power-gating energy overhead.

Impact on Performance

NoRD minimizes the performance penalty of power-gating techniques from the following aspects: (1) the use of decoupling bypass reduces the number of state-transitions and, hence, avoids the wakeup latency when routers do not need to be turned on; (2) when router wakeup is unavoidable, decoupling bypass provides temporary paths for packets while the router is being awoken, thus hiding wakeup latency; (3) a few performance-centric routers with low thresholds can be awoken earlier to guard performance. With these features, NoRD can greatly reduce the performance penalty of conventional power-gating of routers, as the following analysis shows.

5. Evaluation Methodology

5.1 Simulator Configuration

The proposed NoRD scheme is evaluated quantitatively under full-system simulation using Simics [21], with GEMS [22] and Garnet [1] for detailed timing of the memory system and on-chip network. Orion 2.0 [13] is integrated in Garnet for NoC power and area estimation using technology parameters from an industrial standard 45nm CMOS process and 1.1V operating voltage. The saved static power is modeled after [10] and the overhead is modeled after [10, 13]. A wakeup latency of 12 cycles is used assuming a 4ns wakeup delay and 3GHz frequency, and 3 cycles can be hidden when the early wakeup technique [23] is applied. We modify the simulators to model all the key additional hardware for power-gating and bypass, including the extra power consumption in the NI buffering and forwarding logic. The additional dynamic (static) power of the NI in NoRD is lumped into router dynamic (static) power to provide fair comparison across different schemes. Step 2 in Figure 4(c) that checks VC availability in the NI is assumed to take one cycle, as this step essentially reuses the original function in the NI which is modeled as one cycle in Garnet. Wormhole switching with credit-based flow control is assumed, although NoRD is agnostic to the switching and flow control mechanism used. Table 1 lists the key parameters used in the evaluations. Full system simulation uses a 16-node mesh, and synthetic traffic simulation uses both 16- and 64-node configurations to evaluate scalability.

Table 1: Key parameters used in simulation.

Core model	Sun UltraSPARC III+, 3GHz
Private I/D L1$	32KB, 2-way, LRU, 1-cycle latency
Shared L2 per bank	256KB, 16-way, LRU, 6-cycle latency
Cache block size	64Bytes
Coherence protocol	MOESI
Network topology	4x4 and 8x8 mesh
Router	4-stage, 3GHz
Virtual channel	4 per protocol class
Input buffer	5-flit depth
Link bandwidth	128 bits/cycle
Memory controllers	4, located one at each corner
Memory latency	128 cycles

We compare the following designs: (1) No_PG: baseline design with no power-gating; (2) Conv_PG: applying conventional power-gating to routers; (3) Conv_PG_OPT: conventional power-gating optimized with early wakeup (this optimized design not only improves performance by partially hiding wakeup latency, but also reduces power-gating overhead by avoiding powering-off all idle periods that are shorter than 4 cycles); (4) NoRD: our proposed approach based on node-router decoupling. In addition, all designs under evaluation are augmented with adaptive routing algorithms using Duato's Protocol [4]. The only difference is that (1)~(3) use adaptive routing in adaptive VCs and XY routing in escape VCs, whereas (4) uses adaptive routing and the ring-based escape mechanism described in Section 4.2.

5.2 Workloads

Multi-threaded PARSEC 2.0 benchmarks [2] are used for the majority of simulations, as the performance and power consumption of realistic workloads are of primary concern. Each core is warmed up for sufficiently long time (with a minimum of 10 million cycles) and then run until completion. We also perform simulations with synthetic traffic (uniform random and bit-complement [3]) to provide insight on the behavior of different designs across a wide range of load rates and parameter values. In those cases, packets are uniformly assigned two lengths. Short packets are single-flit while long packets have 5 flits. For synthetic traffic, the simulator is warmed up for 10,000 cycles and then the statistics are collected over another 100,000 cycles.

6. Results and Analysis

6.1 Wakeup Thresholds

To simulate NoRD, the appropriate wakeup thresholds must first be found. This is done empirically. All routers are forced into sleep mode without waking up – concentrating traffic on the Bypass Ring – and the number of VC requests (averaged over all routers) is recorded while varying the load rate. It can be seen from Figure 7 that the maximum achievable throughput of the Bypass

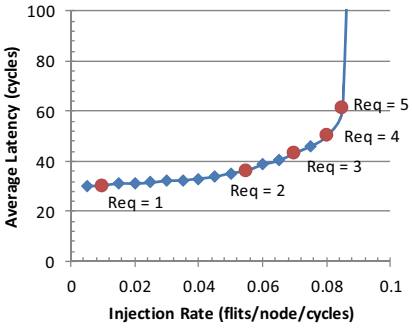


Figure 7: Determining wakeup threshold.

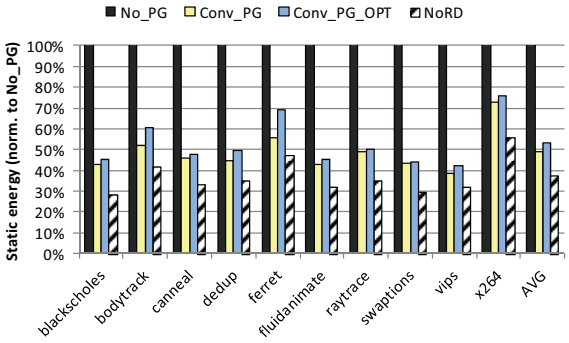


Figure 8: Static energy comparison (normalized to No_PG).

Ring is low (i.e., 14% of the throughput when all routers are turned on), indicating that some routers need to be awoken when network traffic increases, as measured by VC requests.

The objective of choosing the wakeup thresholds is to maximize the static power savings opportunity while not significantly increasing packet latency. In this sense, the dual-threshold technique in asymmetric wakeup thresholding provides more flexibility in achieving a good trade-off. In the current implementation of NoRD, the performance-centric routers are assigned a threshold of 1 as they are critical to performance and need to be awoken early. The remaining power-centric routers can use a higher threshold to enable more power-savings. Considering that a threshold value of 4 VC requests can lead to nearly 60% increase in packet latency, the power-centric routers are assigned a threshold of 3 to avoid large performance penalty. Although the thresholds here are determined empirically, they work very well across all benchmarks.

6.2 Impact on Static Energy

Figure 8 presents the results of static energy of different designs normalized to No_PG. It can be seen that, Conv_PG reduces the static energy slightly more than Conv_PG_OPT by 4.2% on average (51.2% vs. 47.0%). This is because Conv_PG does power-gating as long as the routers are empty whereas Conv_PG_OPT power-gates routers only if the idle periods are longer than 3 cycles as indicated by the early wakeup signal. As shown later, early wakeup pays off for Conv_PG_OPT in terms of performance. The lowest static power is achieved in the proposed NoRD approach for all benchmarks, with an average reduction of 62.9% compared with No_PG. When comparing relatively, NoRD provides savings relative to Conv_PG and Conv_PG_OPT of 23.9% and 29.9% on average, respectively. This improvement mainly comes from the increased opportunity in utilizing short idle periods and the reduced number of wakeups through decoupling bypass.

6.3 Reducing Power-gating Overhead

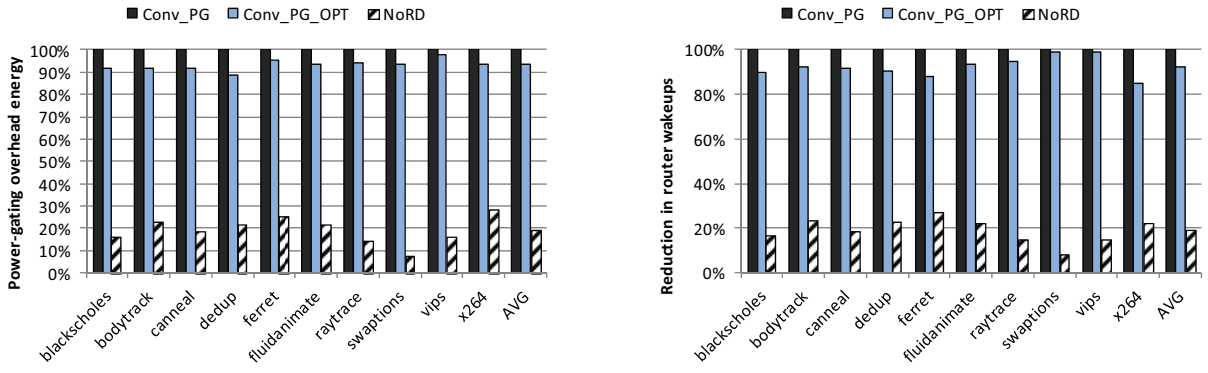
To provide more insight of the effectiveness of NoRD in reducing power-gating overhead, Figure 9(a) compares the energy overhead caused by router wakeup for conventional power-gating designs and the bypass design, normalized to Conv_PG (No_PG is not shown in the figure as it does not have any wakeups). As can be seen, the power-gating overhead in NoRD is considerably reduced by 80.7% and 74.0% compared with Conv_PG and Conv_PG_OPT, respectively. Figure 9(b) shows the reduction in the total number of wakeups in different designs normalized to Conv_PG. NoRD decreases the number of wakeups by 81.0% and 73.3% over Conv_PG and Conv_PT_OPT, respectively, which explains the above substantial reduction of power-gating overhead and demonstrates the usefulness of the decoupling approach.

6.4 Impact on Dynamic Energy

Due to the detour of some packets in bypassing powered-off routers, the dynamic energy of NoRD may increase. Figure 10 plots the breakdown of NoC energy across the benchmarks, so that the relative impact of each NoC energy component can be examined. For the NoC dynamic energy (routers plus links), NoRD incurs an overhead of 10.2% on average, which constitutes 4.0% of the total NoC energy consumption. However, the static energy and wakeup overhead savings offered by NoRD constitutes 24.7% of the total NoC energy. Compared to No_PG, Conv_PG and Conv_PG_OPT, this renders NoRD a net savings of NoC energy of 9.1% and 9.4% and 20.6%, respectively. As on-chip networks consume a varying percentage of chip's overall energy (e.g., around 10%~36% as mentioned in Section 2), the impact of NoRD on overall chip energy depends on particular chip microarchitectures.

6.5 Impact on Performance

After presenting the energy statistics, we now compare the performance impact of different designs, which is another importance objective of power-gating techniques. Figure 11 shows the average packet latency, and Figure 12 compares the execution time of the four designs. No_PG does not have any performance penalty as there is no power-gating, and hence provides a lower bound on average packet latency and execution time. As can be seen, the aggressive power-gating scheme, Conv_PG, significantly degrades the average packet latency by 63.8% on average; whereas Conv_PG_OPT with early wakeup mitigates this degradation to 41.5% on average. These large penalties in conventional power-gating designs mainly come from the fact that once a router is power-gated off, any packet from either local traffic or in-network traffic suffers additional wakeup latency before being processed by the node. The comparison between Conv_PG_OPT and Conv_PG indicates that early wakeup does help a lot in reducing the performance penalty, but still cannot mask entirely the negative effects of wakeup latency. In contrast, NoRD decouples nodes from routers, effectively removing the wakeup latency from the critical path. The latency overhead in NoRD is caused by packet detours, which is partially offset by reduced per hop latency and avoidance of long wakeup latency as discussed before. As a result,



(a) Power-gating energy overhead (b) Reduction in router of wakeups

Figure 9: Reduction of power-gating overhead.

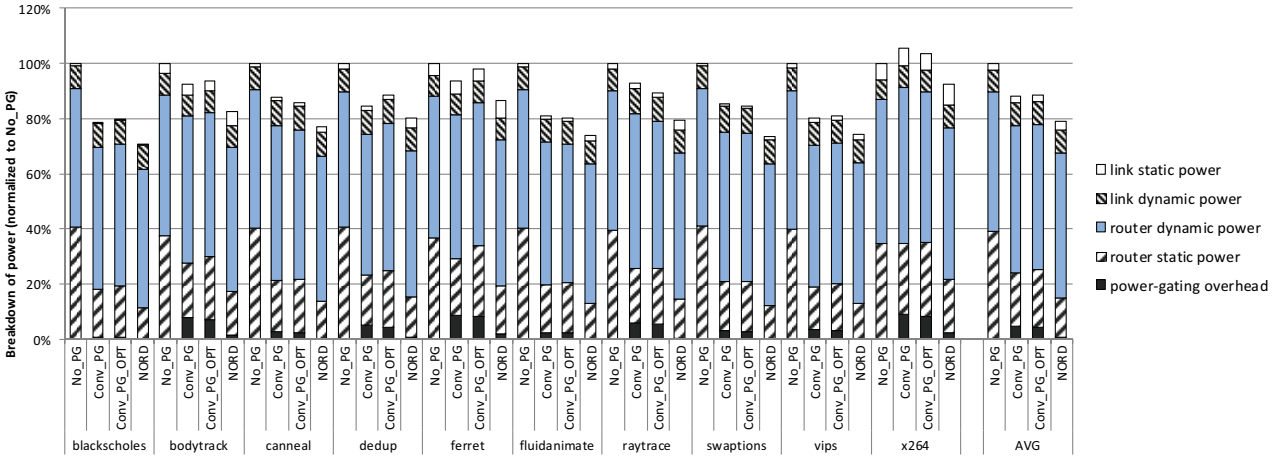


Figure 10: Overall NoC energy breakdown.

the overall degradation of average packet latency in NoRD is only 15.2%, on average. The disparities in average packet latency among these designs result in different execution time, as shown in Figure 12. Although different benchmarks exhibit variations in the specific percentage of degradation due to their difference in network sensitivity, the trend is similar that NoRD has the smallest performance penalty compared to Conv_PG and Conv_PG_OPT. Overall, the Conv_PG, Conv_PG_OPT and NoRD increase the execution time by 11.7%, 8.1% and 3.9%, respectively, in order to achieve the energy saving described previously.

6.6 Effects on Hiding Wakeup Latency

So far, the effectiveness of NoRD has been demonstrated in real applications using full system simulations. In addition to the above primary results, we also perform simulations with synthetic uniform random traffic to highlight key characteristics of NoRD. Recall that cumulative wakeup latency is one of the big obstacles to power-gating routers, particularly in multi-hop networks. To illustrate that NoRD fundamentally solves this problem, Figure 13 shows the average packet latency of Conv_PG, Conv_PG_OPT and NoRD while varying the wakeup latency across a wide range. The load rate is set to the average load rate of PARSEC benchmarks. As can be seen, the latency of Conv_PG and Conv_PG_OPT increases by nearly 1.5X and when the wakeup latency increases from 9 to 18 cycles; whereas the latency of NoRD remain similar for different wakeup latencies, which clearly demonstrates its ability to hide wakeup latency.

6.7 Behavior across Full Range of Network Loads

Next, we investigate the behavior of different designs across the entire network load range: from zero load to saturation loads. Figure 14 presents the performance and power results of a 16-node mesh under uniform random traffic, and Figure 15 presents for 64-node under uniform random and bit-complement traffic. Here, while the behavior of No_PG is very typical, interesting results are found for Conv_PG_OPT and NoRD. These are explained by separating the loads into three regions.

(1) Low to medium load region: When the load is very low, many routers are in the gated-off state for the majority of the time in both Conv_PG_OPT and NoRD. For Conv_PG_OPT, packets are likely to experience wakeup latency once or multiple times, so the average packet latency is high. For NoRD, packets use bypass more often, so average latency is increased due to detours. When load gradually increases, more routers are in the on-state, which tends to reduce the latency. This factor actually offsets the effect of increased load on average latency, leading to a net decrease in

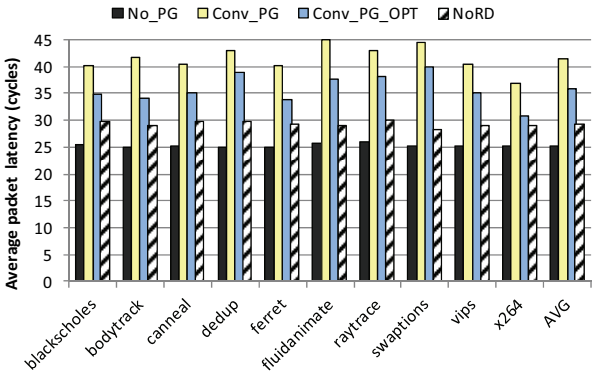


Figure 11: Average packet latency.

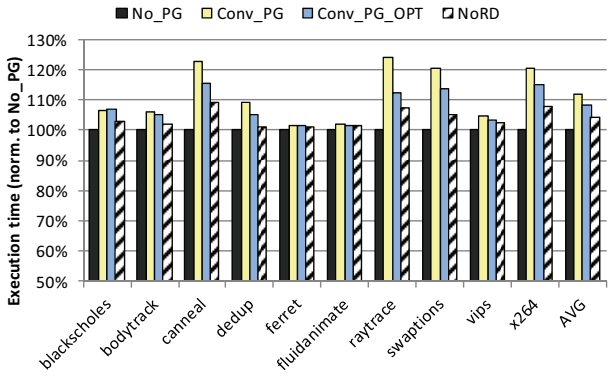


Figure 12: Execution time.

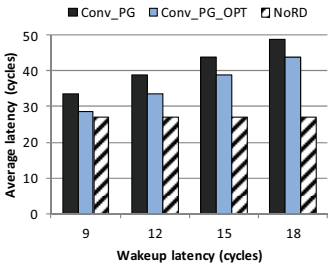


Figure 13: Impact of wakeup latency.

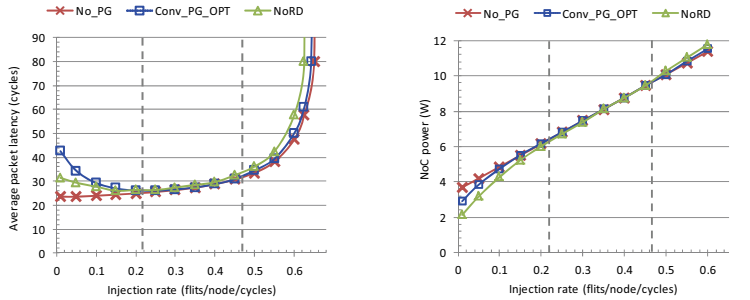


Figure 14: Packet latency and power of 16-node for different load ranges.

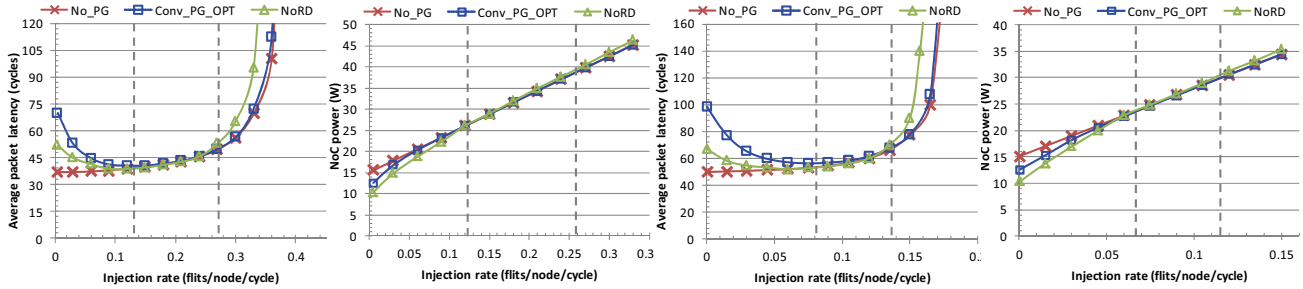


Figure 15: Packet latency and power for 64-node. Left two figures: uniform random; right two figures: bit-complement.

latency for Conv_PG_OPT and NoRD. As can be seen, NoRD achieves both lower average latency and lower power than Conv_PG_OPT. Note that, in this region, NoRD has increased benefits compared to Conv_PG_OPT for larger networks. This is because the cumulative wakeup latency problem in Conv_PG_OPT is more severe due to the increased NoC diameter in larger networks. A gated-off router at any hop of a packet's route adds extra wakeup latency, and every router has a high probability of being gated-off under low load. For instance, at 10% injection rate under uniform traffic, the latency for No_PG, Conv_PG_OPT and NoRD for a 4x4 mesh is 24, 34 and 29 cycles, respectively; whereas for an 8x8 mesh, it is 36, 52 and 44 cycles, respectively. This indicates that for a 64 node network, the latency of NoRD is lower than Conv_PG_OPT with an increased difference compared to the 16 node network. Curves for power for an 8x8 NoC are also similar in shape as for 4x4, indicating that the net energy-savings of NoRD that considers all energy contributors is still more favorable than conventional PG for larger networks.

(2) Medium to high load region: In this region, the three schemes have very similar latency and power characteristics. The relatively high load causes most of the routers to be turned on, making little difference between the designs with or without power-gating.

(3) Saturation region: In this region, as nearly all routers are in the on-state, both Conv_PG_OPT and NoRD are reduced to No_PG, except that they use different escape mechanisms. In this regard, as the escape ring in NoRD has less flexibility in routing packets as compared to escape XY routing of Conv_PG_OPT, NoRD saturates a little earlier. However, this is not an inherent limitation of node-router decoupling, as more efficient dead-lock-free routing algorithms such as in [5] can be used for the bypass ring to close the throughput difference.

Our full system simulations show that real application loads, in practice, typically stay within the low-to-medium region where NoRD has clear advantages over Conv_PG_OPT in both performance and power.

6.8 Discussion

Area Overhead

For any power-gating technique, there is hardware overhead for the sleep switch and the distribution of the sleep signal. While it greatly depends on the optimization level of circuit design, the area overhead of a well-designed power-gating block is usually between 4~10% [10, 12]. More of a concern for NoRD is the area overhead of the added bypass and related hardware. In evaluating this, the Orion 2.0 [13] on-chip network model is used with 45nm technology parameters. We modified the simulator to model all the additional key components of NoRD, including the added forwarding logic in the NI. Results show that NoRD has an area overhead of only 3.1% compared with Conv_PG_OPT.

Other Conventional Power-Gating Techniques

We have compared NoRD with conventional power-gating of routers optimized with early wakeup, which is one of the most effective optimizations so far. Another trade-off that can be done for conventional power-gating is to power-gate smaller individual components within a router, as mentioned in Section 3.2. As investigated in [25], this approach can reduce static energy by an additional 17.6% on top of conventional power-gating with early wakeup, but at the cost of 15.9% area overhead using a commercial standard cell library. In comparison, the proposed NoRD can reduce static energy by 29.9% with only 3.1% area overhead compared to Conv_PG_OPT, indicating NoRD is a much more cost-effective approach.

Bufferless Routing

Recently, bufferless routing has been proposed as a means of reducing router power consumption [6]. Although the bufferless approach may introduce livelock, deflection and packet reassembly issues, it can eliminate buffers and their associated power consumption. However, as shown in Figure 1(b), while buffers are the largest contributor of static power, other router components consume a considerable percentage (e.g., 45%) of total static power, which would remain even if a bufferless approach is used. In fact, bufferless routing is complementary to power-gating techniques in general, as both can be applied at the same time to reduce router power consumption. For example, flits in bufferless routing have the option to be deflected through the bypass paths in NoRD if needed.

Shorter router pipelines and aggressive NoRD design

In the baseline, a canonical router is used which takes 4 cycles for the pipeline plus 1 cycle for LT; whereas the bypass for gated-off routers in NoRD takes 2 cycles plus 1 cycle for LT. There are some techniques such as look-ahead routing [15] and speculative SA [26] that can potentially shorten the 4-cycle router pipeline to 2-cycle. However, NoRD is still competitive in that case for the following reasons. First, shortening the pipeline by two also reduces the number of cycles that can hide wakeup latency by two, making the total time (pipeline delay plus wakeup latency) to go through a gated-off router to remain the same. Second, these techniques come with overheads. Look-ahead routing requires contention information to be propagated one-hop ahead, while speculative SA may not always succeed, making 2 cycles a best-case scenario. Ironically, speculative SA is likely to succeed at low load, in which routers are also likely to be gated-off and the wakeup latency dominates the delay at those routers. Third, the bypass in NoRD can also be optimized to become more aggressive by directly connecting the Bypass Inport to the Bypass Outport. This has a similar rationale as for speculation in that the forwarding of flits optimistically assumes that there is no local flit to inject, thereby bypassing the router in just one cycle. In case of conflict, additional cycles are needed, just like that in speculative SA. Therefore, when optimizations are used for both the baseline and NoRD, there are no clear advantages for the baseline, and NoRD remains competitive.

7. Related Work

Power-gating as a circuit-level technique has been proposed for some time and has been applied to cores and execution units in CMPs [10, 19, 20]. Only recently has it been investigated for on-chip network routers [23, 24, 25]. These works apply power-gating to routers, but are severely limited by the BET requirement, wakeup delay and disconnection problem. In contrast, as our approach breaks node-router dependence, it provides a unified solution to these problems and enables effective use of power-gating to on-chip routers.

Bypass has been used for various purposes in on-chip networks. In [17], default backup paths are proposed to allow fault-tolerance with graceful performance degradation. This scheme assumes all routers are notified each time a router becomes faulty and requires re-computing the routing table for all routers for each fault occurrence. Therefore, it is not suitable for run-time power-gating in which the status of routers may change more frequently. In comparison, each router in the proposed NoRD approach can be powered-on/off independently without notifying all other routers or re-computing any routing tables. A modular router architecture is proposed in [16] that can bypass some internal faults within a router. However, this design does not provide chip-wide connectivity and does not explore the application of power-gating techniques as proposed in this paper. Express VC [18] also makes use of bypass in that it virtually bypasses routers to improve both performance and dynamic power. However, it does not reduce router static power. Another bypass design is proposed in [11] for adaptive flow control between bufferless and buffered router modes. It is based on bufferless design and is subject to the associated constraints, such as flit-by-flit routing, livelock and packet reassembly issues. Moreover, it only targets the buffers in a router and applies power-gating techniques conventionally, whereas our approach is able to bypass the entire router and implement node-router decoupling.

Many prior works have investigated techniques to save dynamic and static power of links [14, 27, 29]. These techniques can readily be used together with NoRD to provide more energy-efficient NoC designs. These works and other general-purpose dynamic power-saving techniques (such as clock-gating) have different targets other than router static power and, therefore, are orthogonal and complementary to this work.

8. Conclusion

While power-gating is a promising technique to reduce static power, node-router dependence severely limits its effective use in on-chip routers due to the BET limitation, wakeup delay and disconnection problem. In this paper, a novel approach that provides separate power-gating bypass to decouple the node's ability for sending, receiving and forwarding packets from the on/off status of the associated router is proposed. The resulting design can significantly reduce the number of state transitions, increase the length of idle periods, completely hide the wakeup latency from the critical path and eliminate node-network disconnection problems. Full system simulations show that,

compared to an optimized conventional power-gating technique applied to on-chip routers, NoRD can further reduce the router static energy by 29.9% and improve average packet latency by 26.3%, with only 3% additional area overhead.

Acknowledgements

We sincerely thank Ruisheng Wang, Siyu Yue, Di Zhu, and the anonymous reviewers for their helpful comments and suggestions. We especially acknowledge the efforts of Yuho Jin in creating Simics checkpoints prior to this research. We also thank Li-Shiuan Peh's research group for their assistance in Orion 2.0. This research was supported, in part, by the National Science Foundation (NSF), grant CCF-0946388.

References

[1] N. Agarwal, T. Krishna, L.-S. Peh, and N. K. Jha, "GARNET: A detailed on-chip network model inside a full-system simulator," in *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 33-42, 2009.

[2] C. Bienia and K. Li, "Parsec 2.0: A new benchmark suite for chip-multiprocessors," in *Proceedings of the 5th Annual Workshop on Modeling, Benchmarking and Simulation*, 2009.

[3] W. Dally and B. Towles, *Principles and Practices of Interconnection Networks*: Morgan Kaufmann Publishers Inc., 2003.

[4] J. Duato, "A new theory of deadlock-free adaptive routing in wormhole networks," *IEEE Transactions on Parallel and Distributed Systems (TPDS),* vol. 4, pp. 1320-31, 1993.

[5] J. Duato and T. M. Pinkston, "A general theory for deadlock-free adaptive routing using a mixed set of resources," *IEEE Transactions on Parallel and Distributed Systems (TPDS),* vol. 12, pp. 1219-1235, 2001.

[6] C. Fallin, C. Craik, and O. Mutlu, "CHIPPER: A low-complexity bufferless deflection router," in *17th International Symposium on High Performance Computer Architecture (HPCA)*, pp. 144-55, 2011.

[7] R. W. Floyd, "Algorithm 97: shortest path," *Communications of the ACM,* vol. 5, p. 345, 1962.

[8] Y. Hoskote, S. Vangal, A. Singh, N. Borkar, and S. Borkar, "A 5-GHz mesh interconnect for a Teraflops processor," *IEEE Micro,* vol. 27, pp. 51-61, 2007.

[9] J. Howard, S. Dighe, S. R. Vangal, G. Ruhl, N. Borkar, S. Jain, V. Erraguntla, M. Konow*, et al.*, "A 48-Core IA-32 Processor in 45 nm CMOS Using On-Die Message-Passing and DVFS for Performance and Power Scaling," *IEEE Journal of Solid-State Circuits,* vol. 46, pp. 173-83, 2011.

[10] Z. Hu, A. Buyuktosunoglu, V. Srinivasan, V. Zyuban, H. Jacobson, and P. Bose, "Microarchitectural techniques for power gating of execution units," in *International Symposium on Lower Power Electronics and Design (ISLPED)*, pp. 32-37, 2004.

[11] S. A. R. Jafri, Y.-J. Hong, M. Thottethodi, and T. N. Vijaykumar, "Adaptive flow control for robust performance and energy," in *43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 433-444, 2010.

[12] H. Jiang, M. Marek-Sadowska, and S. R. Nassif, "Benefits and costs of power-gating technique," in *International Conference on Computer Design (ICCD)*, pp. 559-566, 2005.

[13] A. Kahng, L. Bin, L.-S. Peh, and K. Samadi, "ORION 2.0: A fast and accurate NoC power and area model for early-stage design space exploration," in *Design, Automation and Test in Europe Conference and Exhibition (DATE)*, pp. 423-428, 2009.

[14] E. J. Kim, K. H. Yum, G. M. Link, N. Vijaykrishnan, M. Kandemir, M. J. Irwin, M. Yousif, and C. R. Das, "Energy Optimization Techniques in Cluster Interconnects," in *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED)*, pp. 459-464, 2003.

[15] J. Kim, D. Park, T. Theocharides, N. Vijaykrishnan, and C. R. Das, "A low latency router supporting adaptivity for on-chip interconnects," in *42nd Design Automation Conference (DAC)*, pp. 559-564, 2005.

[16] J. Kim, C. Nicopoulos, D. Park, V. Narayanan, M. S. Yousif, and C. R. Das, "A gracefully degrading and energy-efficient modular router architecture for on-chip networks," in *33rd International Symposium on Computer Architecture (ISCA)*, pp. 4-15, 2006.

[17] M. Koibuchi, H. Matsutani, H. Amano, and T. M. Pinkston, "A lightweight fault-tolerant mechanism for network-on-chip," in *2nd ACM/IEEE International Symposium on Networks-on-Chip (NOCS)*, pp. 13-22, 2008.

[18] A. Kumar, L.-S. Peh, P. Kundu, and N. K. Jha, "Express virtual channels: Towards the ideal interconnection fabric," in *34th Annual International Symposium on Computer Architecture (ISCA)*, pp. 150-161, 2007.

[19] A. Lungu, P. Bose, A. Buyuktosunoglu, and D. J. Sorin, "Dynamic power gating with quality guarantees," in *International Symposium on Low Power Electronics and Design (ISLPED)*, pp. 377-382, 2009.

[20] N. Madan, A. Buyuktosunoglu, P. Bose, and M. Annavaram, "A case for guarded power gating for multi-core processors," in *17th International Symposium on High-Performance Computer Architecture (HPCA)*, pp. 291-300, 2011.

[21] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt*, et al.*, "Simics: A full system simulation platform," *IEEE Computer,* vol. 35, pp. 50-58, 2002.

[22] M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill*, et al.*, "Multifacet's general execution-driven multiprocessor simulator toolset," *ACM SIGARCH Computer Architecture News,* vol. 33, pp. 92-99, 2005.

[23] H. Matsutani, M. Koibuchi, W. Daihan, and H. Amano, "Run-time power gating of on-chip routers using look-ahead routing," in *13th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pp. 55-60, 2008.

[24] H. Matsutani, M. Koibuchi, D. Wang, and H. Amano, "Adding slow-silent virtual channels for low-power on-chip networks," in *2nd ACM/IEEE International Symposium on Networks-on-Chip (NOCS)*, pp. 23-32, 2008.

[25] H. Matsutani, M. Koibuchi, D. Ikebuchi, K. Usami, H. Nakamura, and H. Amano, "Ultra fine-grained run-time power gating of on-chip routers for CMPs," in *4th ACM/IEEE International Symposium on Networks on Chip (NOCS)*, pp. 61-68, 2010.

[26] L. S. Peh and W. J. Dally, "A delay model and speculative architecture for pipelined routers," in *7th International Symposium on High Performance Computer Architecture (HPCA)*, pp. 255-66, 2001.

[27] V. Soteriou and P. Li-Shiuan, "Design-space exploration of power-aware on/off interconnection networks," in *2nd International Conference on Computer Design (ICCD)*, pp. 510-17, 2004.

[28] M. B. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffman, P. Johnson*, et al.*, "The Raw microprocessor: a computational fabric for software circuits and general-purpose programs," *IEEE Micro,* vol. 22, pp. 25-35, 2002.

[29] B. Zafar, J. Draper, and T. M. Pinkston, "Cubic Ring networks: A polymorphic topology for network-on-chip," in *39th International Conference on Parallel Processing (ICPP)*, pp. 443-452, 2010.

2012 IEEE/ACM 45th Annual International Symposium on Microarchitecture

Dynamic Reconfiguration of 3D Photonic Networks-on-Chip for Maximizing Performance and Improving Fault Tolerance

Randy Morris †, Avinash Karanth Kodi †, and Ahmed Louri ‡
†Electrical Engineering and Computer Science, Ohio University, Athens, OH 45701
‡Electrical and Computer Engineering, University of Arizona, Tucson, AZ 85721
rm700603@ohio.edu, kodi@ohio.edu, louri@email.arizona.edu

Abstract

As power dissipation in future Networks-on-Chips (NoCs) is projected to be a major bottleneck, researchers are actively engaged in developing alternate power-efficient technology solutions. Photonic interconnects is a disruptive technology solution that is capable of delivering the communication bandwidth at low power dissipation when the number of cores is scaled to large numbers. Similarly, 3D stacking is another interconnect technology solution that can lead to low energy/bit for communication. In this paper, we propose to combine photonic interconnects with 3D stacking to develop a scalable, reconfigurable, power-efficient and high-performance interconnect for future many-core systems, called R-3PO (Reconfigurable 3D-Photonic Networks-on-Chip). We propose to develop a multi-layer photonic interconnect that can dynamically reconfigure without system intervention and allocate channel bandwidth from less utilized links to more utilized communication links. In addition to improving performance, reconfiguration can re-allocate bandwidth around faulty channels, thereby increasing the resiliency of the architecture and gracefully degrading performance. For 64-core reconfigured network, our simulation results indicate that the performance can be further improved by 10%-25% for Splash-2, PARSEC and SPEC CPU2006 benchmarks, where as simulation results for 256-core chip indicate a performance improvement of more than 25% while saving 6%-36% energy when compared to state-of-the-art on-chip electrical and optical networks.

1. Introduction

Future projections based on ITRS roadmap indicates that complementary metal oxide semiconductor (CMOS) feature sizes will shrink to sub-nanometer within a few years, and we could possibly have as many as 256 cores on-chip by the next decade. While Networks-on-Chip (NoC) design paradigm offers modular and scalable performance, increasing core counts leads to increase in serialization latency and power dissipation as packets are processed at many intermdediate routers. Many electronic NoC designs such as Flattened butterfly [12], concentrated mesh and MECS topologies [10] provide express channels to avoid excess hops between distant nodes. While metallic interconnects can provide the required bandwidth due to abundance of wires in on-chip networks, ensuring high-speed inter-core communication within the allocated power budget in the face of technology scaling (and increased leakage currents) will become a major bottleneck for future multicore designs [4].

Emerging technologies such as photonic interconnects and 3D stacking are under serious consideration for meeting the communication challenges posed by the multicores. Photonic interconnects provides several advantages such as: (1) bit rates independent of distance, (2) higher bandwidth due to multiplexing of wavelengths, (3) larger bandwidth density by multiplexing wavelengths on the same waveguide/fiber, (4) lower power by dissipating only at the endpoints of the communication channel and many more [29, 3, 23]. Similarly, 3D stacking of multiple layers have shown to be advantageous due to (1) shorter inter-layer channel, (2) reduced number of hops and (3) increased bandwidth density. A prevalent way to connect 3D interconnects is to use TSVs (through-silicon vias), micro-bump or flip-chip bonding. The pitch of these vertical vias is very small (4μm$\sim10\mu$m), and delays on the order of 20 ps for a 20-layer stack. Most prior photonic interconnect are 2D designs that are plagued with high optical losses due to waveguide crossings or long snake-like waveguides that coil around the chip to prevent waveguide crossings altogether. For example, a photonic channel with 100 waveguide crossings will have a -5 dB loss if we assume a -0.05 dB loss per waveguide crossing [3]. [1] 3D stacking can avoid waveguide crossings and enable efficient stacking of multiple optical layers to design power-efficient topologies. Jalali's group at UCLA has fabricated a SIMOX (Separation by IMplantation of Oxygen) 3D sculpting to stack optical devices in multiple layers [16]. Lipson group at Cornell has successfully buried active optical ring modulator in polyscrystalline silicon [24]. Moreover, recent work on using silicon nitride has shown the possibility of designing multi-layer 3D integration of photonic layers with layer-to-layer optical losses as low as 0.1 dB [5].

With an emerging technology such as photonic interconnects, it is essential to realize that the hardware cost of designing large scale fully photonic networks requires a substantial investment.[2] Therefore, energy, hardware and architecture limitations could force future designs to limit the number of photonic components at the on-chip level. Moreover, the static channel allocation (wavelengths, waveguides) proposed for most photonic interconnects can provide good performance for uniform traffic, however, for non-uniform and temporal and spatial varying traffic as seen in real traffic, the static allocation could limit the network throughput. Moreover, in case of faults in the channel either due to photonic device or electronic backend circuitry failure, communication can breakdown isolating otherwise healthy cores. However, if the network itself could determine the current load on a channel and re-allocate bandwidth by reconfiguring the network at run-time, then we could improve the throughput, reduce the overall latency, provide alternate routes in case of channel failure and ensure that the network delivers the best performance-per-Watt per application.

To address the requirements of energy-efficient and high-throughput NoCs, we leverage the advantages of two emerging technologies, photonic interconnects and 3D stacking with architectural innovations to design high-bandwidth, low-latency, multi-layer, re-

[1] It should be noted that if the electro-optic integration is not monolithic, then the E/O layers are built separately and integrated in 3D via flip-chip bonding. However, here we refer to 2D designs only in the optic layer.

[2] For example, even after more than a decade of research in optics, Jagaur machine from CRAY which employs a dragonfly topology will account for 20-40% photonics and the rest being metal interconnects due to cost constraints.

1072-4451/12 $26.00 © 2012 IEEE
DOI 10.1109/MICRO.2012.34

IEEE computer society

configurable network, called **R-3PO (Reconfigurable 3D-Photonic On-chip Interconnect)**. R-3PO consists of 16 decomposed photonic interconnect based crossbars placed on four optical communication layers, thereby eliminating waveguide crossing and reducing the optical power losses. The proposed architecture divides a single large monolithic crossbar into several smaller and manageable crossbars which reduces the optical hardware complexity and provides additional disconnected waveguides which provide opportunities for reconfiguration. As the cost of integrating photonics with electronics will be high, statically designed network topologies will find it challenging to meet the dynamically varying communication demands of applications. Therefore, in order to improve network performance, we propose a reconfiguration algorithm whose purpose is to improve performance (throughput, latency) and bypass channel faults by adapting available network bandwidth to application demand by multiplexing signals on crossbar channels that are either idle or healthy. This is accomplished by monitoring the traffic load and applying a reconfiguration algorithm that works in the background without disrupting the on-going communication. Our simulation results on 64-cores and 256-cores using synthetic traffic, SPEC CPU2006, Splash-2 [30] and PARSEC [6] benchmarks provide an energy savings up to 6-36% and outperforms other leading photonic interconnects by more than 10%-25% for adversial traffic via reconfiguration. The significant contributions of this work are as follows:

- We maximize the available bandwidth by reconfiguring the network at run time by monitoring the bandwidth availability and applying the reconfiguration algorithm without disrupting the on-going communication.
- We explore the design space (power-area-performance) of reconfiguring across multiple layers on both synthetic traffic (uniform, permutation) as well as on real application traces (Splash-2, PARSEC, SPEC CPU2006).
- We apply our reconfiguration algorithm to overcome channel faults by effectively sharing the bandwidth of the remaining healthy channels, thereby allowing the performance to degrade gracefully.

2. Related Work

Photonic interconnects is a technology-based solution for designing next generation communication fabric for future multicores. Most photonic interconnects adopt an external laser and on-chip modulators, called micro-ring resonators (MRRs). On application of voltage V_{on}, the refractive index of the MRR is shifted to be in resonance with the incoming wavelength of light which causes a 0 to appear at the end of the waveguide. Similarly, when no voltage is applied, the MRR is not in resonance and a 1 appears at the output. MRRs are used both at the transmitter (modulators) and receiver (filters) sides and have become a favorable choice due to smaller footprint (10 μm), lower power dissipation (0.1 mW), high bandwidth ($>$ 10 Gbps) and low insertion loss (1 dB) [25]. Complementary-metal oxide semiconductor (CMOS) compatible silicon waveguides allow for signal propagation of on-chip light. Waveguides with micron-size cross-sections (5.5 μm) and low-loss (1.3 dB/cm) have been demonstrated [25]. Recent work has shown the possibility of multiplexing 64 wavelengths (wavelength-division multiplexing) within a single waveguide with 60 GHz spacing between wavelengths, although the demonstration was restricted to four wavelengths [3, 25]. An optical receiver performs the optical-to-electrical conversion of data, and consists of a photodetector, a transimpedance amplifier (TIA), and a voltage amplifier [32, 14]. A recent demonstration showed that Si-CMOS-Amplifier has energy dissipation of about 100 fJ/bit with a data rate of 10 Gbps [32]. Thermal stability of MRRs is one of the major challenges causing a mismatch between the incoming wavelength and MRR resonance. Techniques ranging from thermal tuning (more power), athermal tuning (applicable only at fabrication), tuning free-spectral range with backend circuitry (more power) and current injection (smaller tuning range) have been proposed which offer different power consumption levels [7, 9].

On the architecture side, there has been several photonic interconnects that tackle several important issues including arbitration, inter-core communication and core-memory communication[29, 3, 23, 15, 28]. Vantrease et.al. [29] proposed a 3D stacked 256-core photonic interconnect to completely remove all electrical interconnect by designing an optical crossbar and token control. Due to sharing of resources, contention can be high as well as the cost and complexity of designing an optical crossbar for very high core counts. Firefly is an optoelectronic interconnect [23] that reduces the crossbar complexity of [29] by designing smaller optical crossbars connecting select clusters and implementing electrical interconnect within the cluster. In the more recent “macrochip" from Oracle [15], multiple many-core chips are integrated in a single package and propose multi-phase arbitration protocols for communication. FlexiShare [22] is an optical crossbar that combines the advantages of both Corona (single-read, multiple-write) and Firefly (multiple-read, single-write). While Flexishare is concerned with improving bandwidth in the time domain (more slots on more channels), R-3PO improves performance on both space and time domain with a gradient of bandwidth (different percentages). Recently, a 3D photonic interconnect called MPNoCs was proposed that uses multiple layers to create a crossbar with no optical waveguide crossover points [31]. In this work, we extend the 3D photonic interconnect design space by implementing a reconfiguration algorithm that dynamically re-allocates bandwidth from under-utilized to over-utilized links. Prior work on dynamic reconfiguration has been restricted to time slot re-allocation, time and space re-allocation and both power and bandwidth regulation in multiprocessor systems [13]. To the best of our knowledge, this is the first work to propose bandwidth reconfigurability across multiple layers for both improving performance and reliability.

3. R-3PO: Reconfigurable 3D Photonic On-Chip Interconnect

The R-3PO architecture consists of 256 cores, running at 5 GHz, in 64 tile configuration on a 400 mm^2 3D IC. As shown in Figure 1, 256 cores are mapped on a 8×8 network with a concentration factor of four, called a *tile*. From Figure 1(a), the bottom layer, called the *electrical* die, adjacent to the heat sink, contains the cores, caches and memory controllers. To utilize the advantage of a vertical implementation of signal routing, we propose the use of separate optical and core/cache systems unified by a single set of connector vias. The upper die, called the *optical* die, consists of the electro-optic transceivers layer which is driven by the cores via TSVs and four decomposed photonic crossbar layers. The electro-optic layer consists of all the front-end system drivers and the back-end receiver circuitry for photonics. Using TSVs, each tile will modulate the optical signal from an external laser using MRRs and route the signal to the appropriate destination tile. Layers 0-3 contain optical signal routing elements composed of MRRs and bus waveguides and electrical contact for other layers, if necessary. From fabrication perspective, the TSV approach is more tedious due to the maintenance

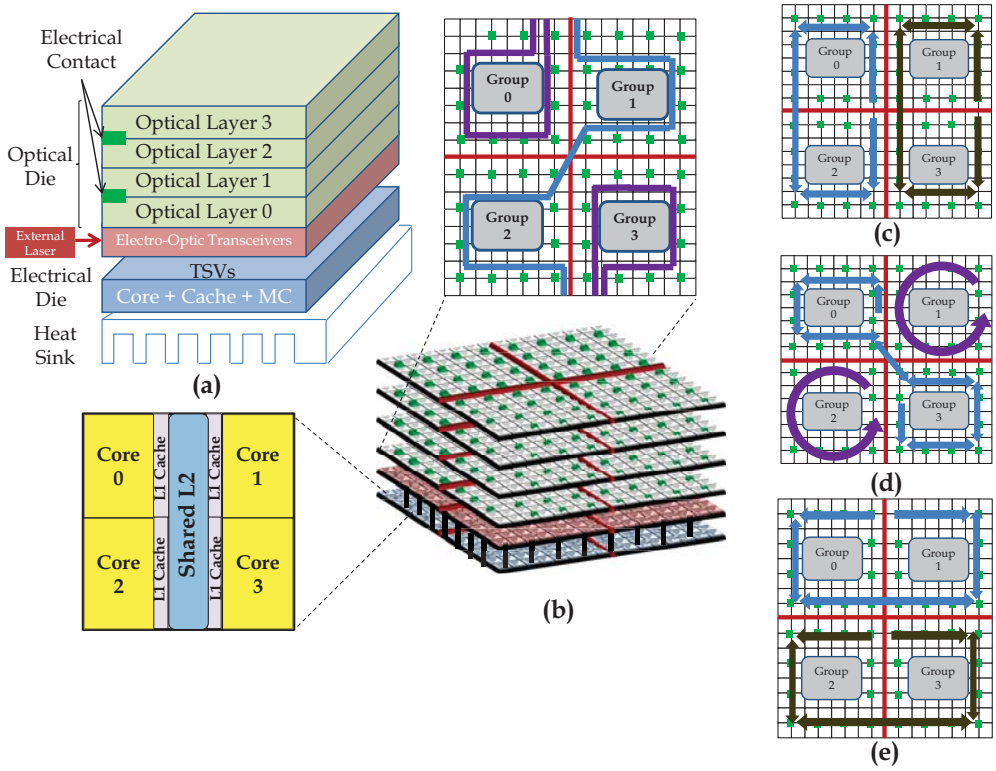


Figure 1: Proposed 256-core 3D chip layout. (a) Electrical die consists of the core, caches, the memory controllers and TSVs to transmit signals between the two dies. The optical die on the lower most layer contain the electro-optic transceivers and four optical layers. (b) 3D chip with four decomposed photonic crossbars with the top inset showing the communication among one group (layer 0) and the bottom inset showing the tile with a shared cache and 4 cores. The decomposition, slicing and mapping of the three additional optical layers: (c) optical layer 1, (d) optical layer 2 and (e) optical layer 3.

of precise alignment for electrical contacts (TSVs). An alternate technique of designing multiple layers is to incorporate racetrack configuration where electrical contact is restricted to the bottom layer (electro-optic layer), and the signal propagation to upper layers is via passive vertical coupling of MRRs. While the racetrack configuration eases fabrication, the technique could increase the laser power due to extra vertical coupling losses and complicate the thermal heating at the upper layers. Adding multiple ring resonators actually improves the filtering of the signal and reduces the crosstalk due to residual signals. Therefore, we propose to design multiple photonic ring resonators coupled in racetrack configuration to traverse multiple layers to prevent the over-reliance on TSVs.

3.1. Intra- and Inter-Group Communication

In the proposed 3D layout, we divide tiles into four groups based on their physical location. Each group contains 16 tiles. Unlike the global 64 $\times$ 64 photonic crossbar design in [29] and the hierarchical architecture in [23], R-3PO consists of 16 decomposed individual photonic crossbars mapped on four optical layers. Each photonic crossbar is a 16 $\times$ 16 crossbar connecting all tiles from one group to another (Inter-group). It is composed of Multiple-Write-Single-Read (MWSR) photonic channels, which requires lesser power than Single-Write-Multiple-Read (SWMR) channels described in [23]. A MWSR photonic channel allows multiple nodes the ability to write on the channel but only one node can read the channel. This channel design reduces power but requires arbitration as multiple nodes can write at the same time. On the other hand, a SWMR channel allows only one node the ability to write to the channel but multiple nodes can read the data. This channel design reduces latency as no arbitration is required but requires source destination handshaking protocol or else, the power to broadcast will be higher. We adopt MWSR and Token slot [29] in this architecture to improve the arbitration efficiency for the channel. Each waveguide used within a photonic crossbar has only one receiver which we define as the *home channel*. During communication, the source tile sends packets to their destination tile by modulating the light on the home channel of the destination tile. An off-chip laser generates the required 64 continuous wavelengths, $\bigwedge = \lambda_0, \lambda_1, \lambda_2 \ldots. \lambda_{63}$. Figure 1(b) shows the detailed floor plan for the first optical layer. For optical layer 0, a 32 waveguide bundle is used for communication between Groups 0 and 3 and two 16 waveguide bundles are used for communication within Groups 1 and 2. For inter-group communication between 0 and 3, the first 16 waveguide bundle is routed past Group 0 tiles so that any tile within Group 0 can transmit data to any destination tile in Group 3. Similarly, the next 16 waveguide bundle is routed past Group 3, so that any tile within Group 3 can communicate with a destination tiles located within Group 0. The bidirectional arrows illustrate that light travels in both directions and depends on which group is the source and destination. The remaining two independent waveguide bundles (16 waveguides) are used for intra-group communication for Groups 1 and 2 respectively. Therefore, we require a total of 64 waveguide bundle per layer. A detailed decomposition and slicing of the crossbar on the other three layers is shown in Figure 1(c-e).

3.2. Router Microarchitecture

Figure 2(a) shows the router microarchitecture in R-3PO for tile 0. Any packet generated from the L2 cache is routed to the input demux with the header directed towards RC (routing computation). The two MSBs are used to direct the packet to a one of the four sets of input buffers (IB_0 - IB_3) corresponding to each optical layer (0-3). For the second set of demuxes, the packet will utilize an unique identifier (that corresponds to the core number) to indicate the source of the packet to prevent any core from overwhelming the input buffers. Token (Request + Release) ensures that packets are transmitted from the IBs without collision and the MRRs are used to modulate the signal into the corresponding home channel. At the receiver, the reverse process takes place where the packet from the optical layer is converted into electronics and according to the unique identifier will find one set of buffers available. Token Control is used to prevent buffer overflow at the home node by checking the number of empty buffer slots. If the number of empty buffer slots falls below a certain threshold Buf_Th, then the destination tile will capture the circulating token and will not re-inject the token until the number of free slots increases to the threshold. Furthermore, the receiver of R-3PO does not require router computation for an incoming flit of a packet because, flit interleaving does not take place as an optical token is not re-injected until the whole packet is sent. The packets will then contend to obtain the switch (switch allocator (SA)) to reach the L2 cache. It should be noted that the proposed unique identifier is similar to virtual channel allocator, however we do not perform any allocation as the decision to enter any buffer is determined on the core number (source or destination). Figure 2(b) shows the proposed token control block. In the token control block, an optical token is only placed on the token inject waveguide when an optical token is present (high TR signal) and the buffer congestion (BC) signal is low. A low BC signal in this case represents a free buffer slot at the destination tile and a high BC signal represent that all the buffer slots are full at the destination tile. 2(c) shows the router pipeline. RC ensures that the packet is directed to the correct output port for both static and reconfigured communication. BWS writes the packet into the buffer slot. EO conversion takes place with appropriate buffer chain after the token is received. Optical transmission can take anywhere between 1-3 clock cycles running at 5 Ghz. OE conversion is repeated at the receiver, BWD writes the packet into the buffer slot and finally switch allocation (SA) ensures that the packet progresses into the L2 cache.

4. Reconfiguration

As future multicores will run diverse scientific and commercial applications, networks that can adapt to communication traffic at runtime will maximize the available resources while simultaneously improving the performance. Moreover, faults within the network or the channel can isolate healthy groups of tiles; with the natural redundancy available in the decomposed crossbar, we can take advantage of reconfiguration to overcome channel faults and maintain limited connectivity. To implement reconfiguration, we propose to include additional MRRs that can switch the wavelengths from different layers to create a reconfigurable network. Further, we also propose a reconfiguration algorithm to monitor traffic load and dynamically adjust the bandwidth by re-allocating excess bandwidth from under-utilized links to over-utilized links.

4.1. Bandwidth Re-Allocation

To illustrate with an example, consider a situation where tiles in Group 0 communicates only with tiles in Group 3. Figure 3 shows

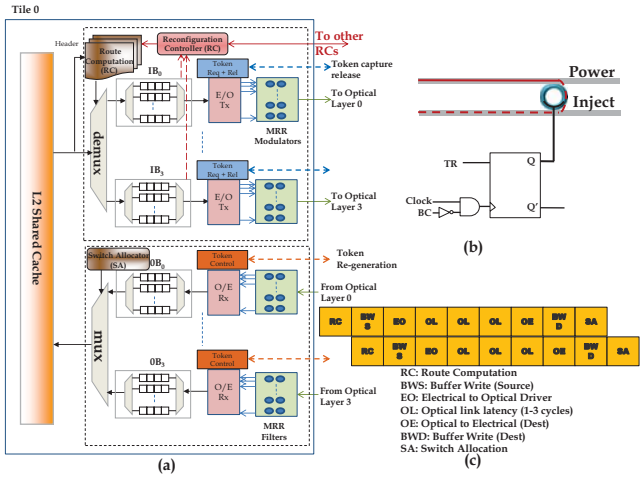


Figure 2: (a) Router microarchitecture, (b) token control and (c) router pipeline.

the reconfiguration mechanism. The *static* allocation of channel for communication are in layer 2 as shown in Figure 3(a). Suppose no tile within Group 1 (in layer 1) communicates with Group 3, then we can re-allocate the bandwidth from Group 1 to Group 0 to communicate with Group 3. To implement reconfiguration, however, we need to satisfy two important requirements: (1) There should be a source waveguide which should be freely available to start the communication on a source layer, and (2) there should be a destination waveguide which also should be freely available to receive the extra packets. As shown in Figure 3(b), as the two Groups 0 and 3 talk to each other, we have the first set of waveguides on layer 0 (generally used to communicate within the group) available, therefore this satisfies the first condition. As Group 1 does not communicate with Group 3, we can utilize the destination waveguide available in layer 1 and this satisfies the second condition. The signal originates on layer 0, switches to layer 1 to reach the destination. Note that this additional channel is available in addition to layer 2 static configuration, thereby doubling the bandwidth. Therefore, during reconfiguration Group 0 has doubled the bandwidth to communicate with Group 3 via layers 2 (static) and 1 (dynamic). Two different communication are disrupted when the reconfiguration occurs, namely, Group 0 in layer 0 can no longer communicate with itself and Group 1 in layer 1 can no longer communicate with Group 3.

4.2. Design-Space Exploration

The objective of reconfiguration is to improve performance by re-allocating bandwidth from under-utilized to over-utilized links. The design space of reconfiguration is large as there can be several combinations across multiple layers. Figure 4 shows four possible combinations that we will evaluate as they cover most of the design space. Row-column matrix indicates the statically allocated communication. For example layer 0 - layer 0 shows three combinations G0 $<->$ G0, G1 $<->$ G2 and G3 $<->$ G3 i.e. group 0 communicates with itself, groups 1 and 2 communicate with each other and group 3 communicates with itself. The square (red) boxes show which layers can be used for reconfiguration and the arrow indicates the layers that can be used for reconfiguration. Figure 4(a) shows layer 0 can reconfigure and take away bandwidth from layer 1; similarly layer 1 can reconfigure and can take away bandwidth from layer 0. Layer

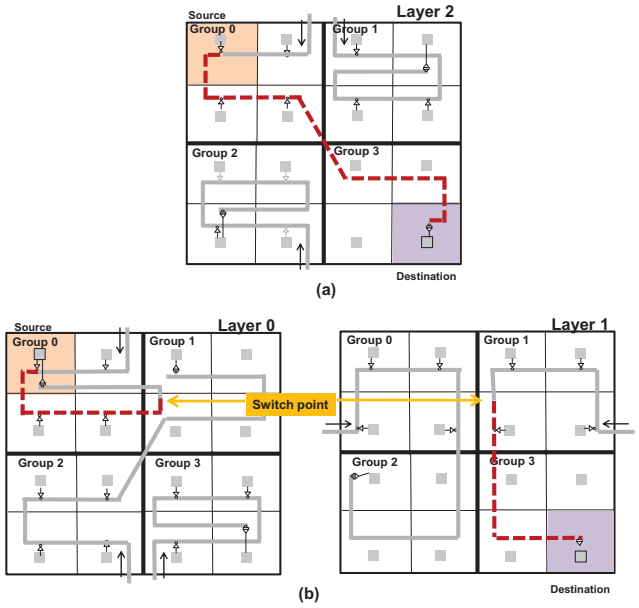


Figure 3: (a) Static communication between the source in Group 0 and destination in Group 3. (b) Illustration of reconfiguration between Groups 0 and 3 using partial waveguides from layers 0 and 1.

2 can take bandwidth from layer 3 and vice versa. This approach restricts to one additional layer that can be used for reconfiguration and we call this R-3PO-L1 (R-3PO-Limited to 1 Layer) and this restricted mechanism will reduce both the power consumption and area overhead. Figure 4(b) shows reconfiguration across one or two layers; however both layers have to be adjacent. Layer 0 can only reconfigure with layer 1, where as layer 1 can reconfigure with both layer 0 and layer 2 (adjacent). Adjacent layer reconfiguration is easier to implement as the next layer (above or below) will be used which improves on a single layer and we call this R-3PO-LA (R-3PO-Limited to adjacent layer). Figure 4(c) shows reconfiguration across two layers even if they are not adjacent and we call this configuration R-3PO-L2 (R-3PO-Limited to 2 Layers). This increases the power consumption as well as design fabrication as more TSVs will be needed. One side-effect of this reconfiguration is that as more layers are involved, there are more channels lost due to reconfiguration. This is primarily due to the fact that as additional waveguides are consumed, we are then restricting the number of layers that can be reconfigured. For adverse and embarrassingly parallel applications, this would be an interesting option as more layers can be used for reconfiguration. Figure 4(d) shows the complete reconfiguration, as any layer can go to any other layer, and we call this configuration R-3PO-L3 (R-3PO-All 3 Layers). This fully reconfigured design will need the most in terms of area overhead and also incur higher complexity in terms of fabrication as TSVs have to extend to all the layers.

4.3. Fault Tolerance

Fault tolerance occurs by allowing data from the faulty channel to be switched to an adjacent layer (channel) that communicates with the same destination. Figure 5 shows an example of how fault tolerance is implemented in R-3PO. In this example, the tiles in

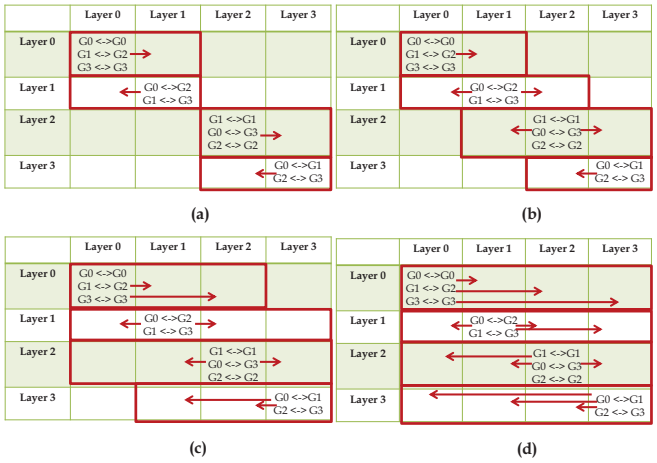


Figure 4: Various configurations evaluated: (a) R-3PO-L1 (R-3PO-Limited to 1 Layer), (b) R-3PO-LA (R-3PO-Limited to adjacent layer), (c) R-3PO-L2 (R-3PO-Limited to 2 Layers) and (d) R-3PO-L3 (R-3PO-All 3 Layers)

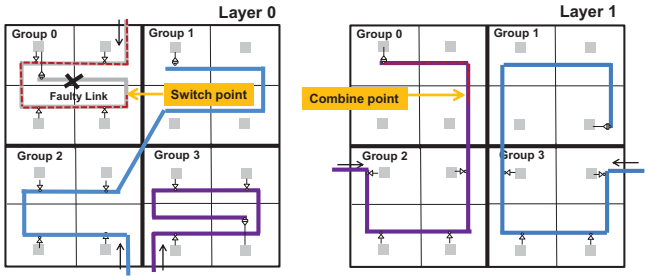


Figure 5: Fault tolerance in R-3PO.

Group 0 cannot communicate with Tile 0 because the optical receiver at Tile 0 is inoperable or faulty, thereby isolating Tile 0 from other tiles in Group 0. To detect a fault, we augment the reconfiguration algorithm and hardware counters to detect faulty links.[3] Once the fault is detected, data is re-routed to the adjacent layer waveguides that communicate with the same destination tile. After Group 0 detects that the communication to Tile 0 is faulty, any data originating from Group 0 will be switched to the waveguide in Layer 1 that communicates with Tile 0. In addition, we prevent a tile from Group 0 and a Tile from Group 2 from communicating to Tile 0 at the same time which requires the token sharing scheme to be updated. In this example, after the fault is detected, any tiles in Group 0 will need to capture the token that tiles in Group 2 use to communicate with Tile 0. This in essence increases the number of tiles that share a common link; therefore the bandwidth Group 2 utilizes to communicate with Tile 0 is now shared by all tiles in Group 0 to communicate with Tile 0. Reconfiguration allows bandwidth or channel sharing where the faulty channel can be bypassed by using bandwidth on adjacent layer.

4.4. Algorithm Implementation

We design our reconfiguration algorithm with the following objectives: (a) The algorithm should not be overly sensitive to traffic

[3]The design space of testing the functionality of the channel is vast as multiple sources or destination can be faulty; in this paper, we limit fault to the destination receiver which can be self tested by the home channel by transmitting a pinging packet.

fluctuations to prevent rapid changes in topology; (b) the algorithm should mostly work in the background, and (c) the algorithm should ensure that no tile is starved from bandwidth. To implement such a reconfiguration, we first take measurements that are available such as link utilization (Link_{util}) and buffer utilization (Buffer_{util}) using hardware counters [8]. This implies that each tile within a group will have four hardware counters (one for each of the three groups) that will monitor traffic utilization and provide the link and buffer information to Reconfiguration Controller (shown in 2). All these statistics are measured over a sampling time window called *Reconfiguration window* or phase, $R_W{}^t$, where t represents the reconfiguration time t. This sampling window impacts performance, as reconfiguring finely incurs latency penalty and reconfiguring coarsely may not adapt in time for traffic fluctuations. In our performance section, we show that we evaluated a number of PARSEC applications to determine the optimum size for R_W. For calculation of Link_{util} at configuration window t, we use the following equation:

$$Link_{util}^{t} = \frac{\sum_{cycle=1}^{R_W} Activity(cycle)}{R_W} \quad (1)$$

where $Activity(cycle)$ is 1 if a flit is transmitted on the link or 0 if no flit is transmitted on the link for a given cycle. For calculation of Buffer_{util} at configuration window t, we use the following equation:

$$Buffer_{util}^{t} = \frac{\sum_{cycle=1}^{R_W} Occupy(cycle)/Total_{buffers}}{R_W} \quad (2)$$

where $Occupy(cycle)$ is the number of buffers occupied at each cycle and $\text{Total}_{b}uffers$ is the total number of buffers available for the given link. When traffic fluctuates dynamically due to short term bursty behavior, the buffers could fill up instantly. This can adversely impact the reconfiguration algorithm as it tries to re-allocate the bandwidth faster leading to fluctuating bandwidth allocation. To prevent temporal and spatial traffic fluctuations affecting performance, we take a weighted average of current network statistics (Link_{util} and Buffer_{util}), so that the network will gradually re-allocate bandwidth. We calculate the Buffer_{util} as follows:

$$Buffer_{w}^{t} = \frac{\sum_{Buffer_{util}^{t} \times weight + Buffer_{util}^{t-1}}}{weight + 1} \quad (3)$$

where *weight* is a weighting factor and we set this to three in our simulations [26].

After each $R_W{}^t$, each tile will gather its link statistics (Link_{util} and Buffer_{util}) from the previous window $R_W{}^{t-1}$ and send to its local reconfiguration controller (RC) for analysis. We assume that Tile 0 of every group gathers the statistics from the remaining tiles and this can be few bytes of information that is periodically transmitted. Next, when each RC_i, ($\forall$ i = 0, 1, 2, 3), has finished gathering link and buffer statistics from all its hardware controllers, each RC_i will evaluate the available bandwidth for each link depending on the $\text{Link}_{util}{}^{t-1}$ and $\text{Buffer}_{util}{}^{t-1}$ and will classify its available bandwidth into a different thresholds β_{1-4} corresponding to 0%, 25%, 50% and 90%. We never allocate 100% of the bandwidth as the source group may have new packets to transmit to the destination tile before the next R_W. RC_i will send link information (availability) to its neighbor RC_j (j $\neq$ i). If RC_j needs the available bandwidth, RC_j will notify the source and the destination RCs so that they can switch the MRRs and inform the tiles locally of the availability. Once the source/desitnation RCs have switched their reconfiguration MRRs, RC_i will notify RC_j that the bandwidth is available for use. On the other hand, if a node within RC_i that throttled its bandwidth requires it back due to increase in network demand, RC_i will notify that it requires the bandwidth back and afterwards will deactivate the corresponding MRRs. The above reconfiguration completes a three-way handshake where RC_i first notifies RC_j, then RC_j notifies RC_i that RC_j will use the additional bandwidth, and finally RC_i notifies RC_j that the bandwidth can be used. Table 1 shows the reconfiguration algorithm in R-3PO.

Table 1: Reconfiguration Algorithm used in R-3PO.

Step 1:	Wait for Reconfiguration window, $R_W{}^t$
Step 2:	RC_i sends a request packet to all local tiles requesting $Link_{Util}$ and $Buffer_{Util}$ for previous $R_W{}^{t-1}$
Step 3:	Each hardware counter sends $Link_{Util}$ and $Buffer_{Util}$ statistics from the previous $R_W{}^{t-1}$ to RC_i
Step 4(a):	RC_i classifies the link statistic for each hardware counter as: If Link_{util} = 0.0 Not-Utilized: Use β_4 If $\text{Link}_{util} \leq \text{L}_{min}$ Under-Utilized: Use β_3 If $\text{Link}_{util} \geq \text{L}_{min}$ and $\text{Buffer}_{util} < \text{B}_{con}$ Normal-Utilized: Use β_2 If $\text{Buffer}_{util} > \text{B}_{con}$ Over-Utilized: Use β_1
Step 4(b):	Faulty links detected by RC_i are eliminated from reconfiguration; Token sharing updated to bypass the faulty link
Step 5:	Each RC_i sends bandwidth available information to RC_j, (i$\neq$j)
Step 6:	If RC_j can use any of the free links then notify RC_i of their use, else RC_j will forward to next RC_j
Step 7a:	RC_i receives response back from RC_j and activates corresponding microrings
Step 7b:	RC_j notifies the tiles of additional bandwidth and RC_i notifies RC_j that the additional bandwidth is now available
Step 8:	Goto Step 1

5. Performance Evaluation

In this section, we evaluate the performance, power-efficiency and impact of faulty channel in R-3PO when compared to competing electrical interconnects and photonic interconnects.

5.1. Simulation Setup

Our cycle-accurate simulator models in detail the router pipeline, arbitration, switching and flow control. An aggressive single cycle electrical router is applied in each tile and the flit transversal time is one cycle from the local core to electrical router [18]. As the delay of Optical/Electrical (O/E) and Electrical/Optical (E/O) conversion can be reduced to less than 100 ps [29], the total optical transmission latency is determined by physical location of source/destination pair (1 - 3 cycles) and two additional clock cycles for the conversion delay. We assume an input buffer of 16 flits with each flit consisting of 128

bits. The packet size is 4 flits which is sufficient to fit a complete cache line of 64 bytes. We assume a supply voltage V_{dd} of 1.0 V and a router clock frequency of 5 Ghz [29, 23]. We compare R-3PO architecture to three other crossbar-like photonic interconnects, Corona [29], Firefly [23], MPNoCs [31]; and two electrical interconnects (mesh and Flattened Butterfly) [12]. We implement all architectures such that four cores (one tile) are connected to a single router. We assume token slot for both R-3PO and Corona to pipeline the arbitration process to increase the efficiency. We use Fly_Src routing algorithm [23] for Firefly architecture, where intra-group communication via electrical mesh is implemented first and then inter-group via photonic interconnects. For a fair comparison, we ensure that each communication channel in either electrical or optical network is 640 Gbps with 64 wavelengths. We also evaluate the performance by restricting the channel bandwidth to 16/8 wavelengths and communication bandwidth limited to 160/80 Gbps. For each network, we ensure that identical bandwidth is maintained for each link in our network, thereby providing equal bandwidth between each source and destination pairs, whether it be electrical or optical networks.

For open-loop measurement, the packet injection rate is varied from 0.1 to 0.9 of the network capacity, and packets are injected according to the Bernoulli process based on the given network load. The simulator was warmed up under load without taking measurements until steady state was reached (up to 1000 cycles). Then a sample of injected packets were labeled during a measurement interval (1000 to 10,000). The simulation was allowed to run until all the labeled packets reached their destinations. We consider both uniform as well as permutation traffic such as bit-complement (bitcomp), bit-reversal (bitrev), transpose, butterfly, neighbor and prefect shuffle traffic patterns for 256-cores.

For closed-loop measurement, we collect traces from real applications using the full execution-driven simulator SIMICS from WindRiver with the memory package GEMS enabled [20]. We evaluate the performance of 64-core versions of each network on Splash-2 [30], PARSEC [6] and SPEC CPU2006 workloads. We assume a 2 cycle latency to access the L1 cache (64 KB, 4-way), a 4 cycle latency to access the L2 cache (4MB, 16-way), cache line size of 64 bytes and a 160 cycle latency to access the main memory. For Splash-2 traffic, we assume the following kernels and workloads: FFT (16K particles), LU (512 $\times$ 512 with a block size of 16 $\times$ 16), Radix (1 Million integers), Ocean (258 $\times$ 258), and Water (512 Molecules). We consider six PARSEC applications with medium inputs (blackscholes, facesim, fluidanimate, freqmin, and streamcluster) and two workloads from SPEC CPU2006 (bzip and hmmer). We ran several benchmarks of PARSEC and Splash-2 to determine the optimum size of R_W by varying the simulation cycles. While initially the performance improved with increasing window size as more statistics are available which enable better decision making; at very large window sizes, the performance diminishes as the algorithm cannot react fast enough to take advantage of the reconfiguration algorithm. Our simulation results show that 1300 cycles for R_W showed the best performance. We assume a 100 cycle latency for the reconfiguration to take place after each R_W (three-way handshake delay). It should be noted that the reconfiguration latency is only incurred by those links that already are lightly loaded and, therefore do not experience a significant delay.

5.2. Simulation Results

5.2.1. Splash-2, PARSEC and SPEC CPU2006 for 64 Cores: We analyze the speed-up for few selected Splash-2, PARSEC and SPEC CPU2006 applications [30] for 64/16/8 wavelengths, where the speed-up is normalized to mesh architecture. From Figure 6, all R-3PO configurations show a speedup of 2.5 - 3X over electrical mesh, 10-40% improvement over Flattened-Butterfly and Firefly architectures, 22-18% over MPNoC and Corona architectures for 64 wavelengths. The performance gains over electrical and electro-optic networks are derived primarily due to the decomposed crossbars which enable increased traffic outflows from the router into four difference optical crossbars. Further performance improvement over photonic crossbars such as Corona and MPNoC are due to the reconfiguration algorithm which takes advantage of the idle communication channels. Within the four different configurations of R-3POs, the best performing configuration is R-3PO-L3 which provides the maximum flexibility by reconfiguring all the optical layers. For 64 wavelengths, the performance improvements provided by R-3PO-L3 and R-3PO-LA is 6-8% for streamcluster and bzip over R-3PO-L1.

Figure 7 shows the performance of various networks on Splash-2, PARSEC and SPEC CPU2006 benchmarks for 16 wavelengths. From Figure 7, all R-3PO configurations show a speedup of 2.3 - 3.5X over electrical mesh, 17-40% improvement over Flattened-Butterfly and Firefly architectures, 32-18% over MPNoC and Corona architectures for 16 wavelengths. When the number of wavelengths is reduced, the performance improvements over 64 wavelengths are primarily due to the reconfiguration algorithm as the additional bandwidth has more of an impact on the speedup. Figure 8 shows the performance of various networks for 8 wavelengths. From Figure 8), all R-3PO configurations show a speedup of 2.1 - 3.6X over electrical mesh, 17-62% improvement over Flattened-Butterfly and Firefly architectures, 42-18% over MPNoC and Corona architectures for 8 wavelengths. When the resources are further constrained, the bandwidth is stressed where the re-allocated bandwidth via reconfiguration can alleviate performance. Clearly, the performance gains increases dramatically when we reduce the bandwidth and the reconfiguration algorithm can assist in improving the performance. For LU, water, streamcluster and facesim benchmarks, R-3PO-L2 and R-3PO-L3 show over a 10% increase in performance when compared to R-3PO-L1. From the figure, the average speed provided by R3P0-LA, R-3PO-L2, and R-3PO-L3 over R-3PO-L1 ranges from about 1% to as high as 10%. Multiple configurations of R-3PO provide different performance gains and the speedup increases with reduced bandwidth via reconfiguration.

5.2.2. Synthetic Traffic: 256 Cores The throughput for all synthetic traffic traces for 256-core implementations are shown in Figure 9 and is normalized to mesh network (for Uniform, the mesh has a throughput of 624 GBytes per sec). R-3PO-L1 has about a 2.5 $\times$ increase in throughput over Corona for uniform traffic due to the decomposition of the photonic crossbar. The decomposed crossbars allow for a reduction in contention for optical tokens as now a single token is shared between 16 tiles instead of 64 tiles as in Corona. Firefly slightly outperforms R-3PO-L1 for uniform traffic due to the contention found in the decomposed photonic crossbars. Moveover, Firefly uses a SWMR approach for communication which does not require optical arbitration. From the figure, R-3PO-L1 slightly outperforms Corona for bit-reversal and complement traffic traces. This is due to lower contention for optical tokens in the decomposed crossbars. R-3PO-L1 significantly outperforms mesh for the bit-reversal, matrix-transpose and complement traffic patterns. In these traffic patterns, packets need to traversal across multiple mesh routers which in turn increases the packet latency and thereby reduces the through-

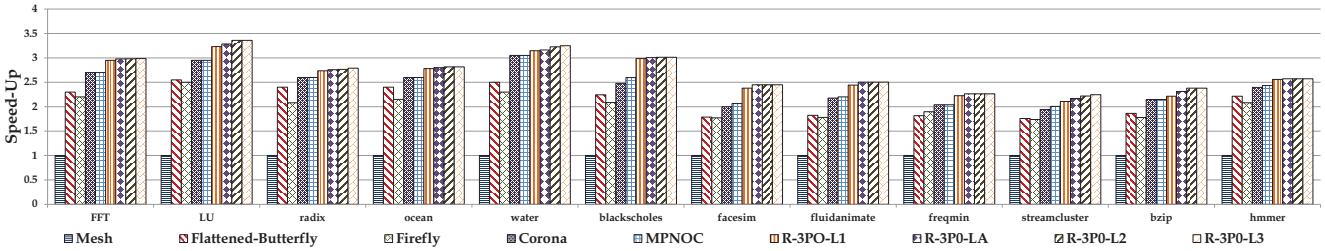


Figure 6: Speed-up for 64-core using SPLASH-2, PARSEC and SPEC CPU2006 traffic traces using 64 wavelengths.

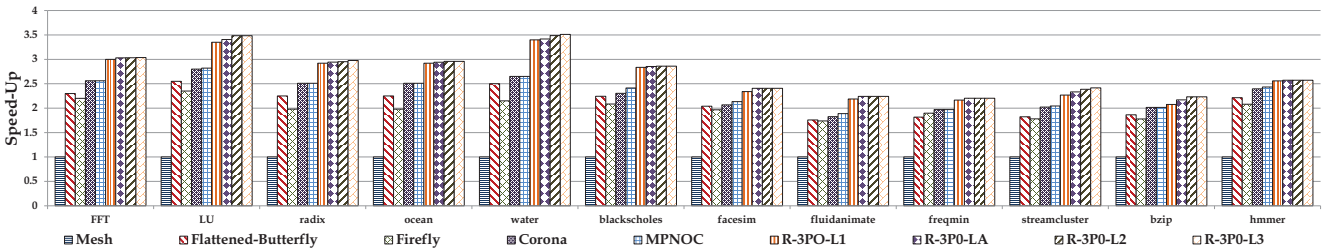


Figure 7: Speed-up for 64-core using SPLASH-2, PARSEC and SPEC CPU2006 traffic traces using 16 wavelengths.

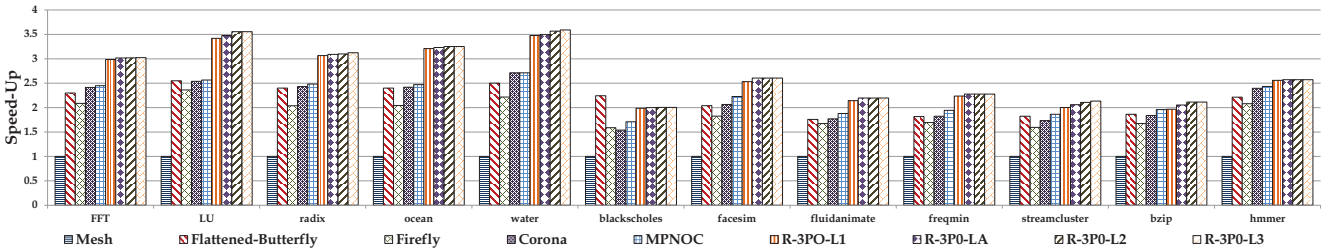


Figure 8: Speed-up for 64-core using SPLASH-2, PARSEC and SPEC CPU2006 traffic traces using 8 wavelengths.

put. When R-3PO-L1 is compared to Firefly, R-3PO-L1 outperforms Firefly by 2.5 $\times$. In Firefly, most traffic patterns will require packets to travel on several electrical routers and then an optical link to reach the destination. R-3PO-L3 is able to out perform R-3PO-L1 for complement, matrix-transpose and perfect shuffle traffic traces. These permutation traffic traces exhibit adversial patterns which will benefit R-3PO-L1. In complement traffic, R-3PO-L1 has about a 55% increase in performance when compared to R-3PO-L1.

5.2.3. Fault Tolerance We evaluated the performance degradation when 10%, 25% and 50% of the channels are faulty. These faults were randomly inserted such that they do not coincide with the reconfiguration window cycle. We assume that every tile checks the home channel working once at the beginning of the R_w and if there are any faults, bandwidth sharing is enabled where bandwidth from other healthy channels is re-allocated. Figure 10 shows the performance degradation for R-3PO-L1 for 64 wavelengths. The results show that with 10%, 25% and 50% link failures, performance degrades by 5%, 10-15% and 20-40% respectively. While the reconfiguration algorithm kicks in within a couple of iterations (in worst case scenario), the loss primarily arises from the sharing of channel which increases the latency for both faulty as well as non-faulty communication. The results show that reconfiguration algorithm can bypass the faults by efficiently sharing the link bandwidth with some performance degradation. While the fault model assumes a high fault rate (10% - 50%), with adequate process development and monolithic integration, variation-induced fault rates are actually much lower [2]. Our analysis assumes worst-case fault rate for the system evaluation with reconfiguration.

5.3. Energy Comparison

The energy consumption of a photonic interconnect can be divided into two parts, electrical energy and optical energy. Optical energy consists of the off-chip laser energy and on-chip MRRs heating energy. In what follows, we first discuss the electrical energy and then optical energy consumption.

5.3.1. Electrical Energy Model The electrical energy dissipated includes the energy of the link, router and back-end circuitry for optical transmitter and receiver. We use ORION 2.0 [11] to obtain the energy dissipation values for an electrical link and router and modified their parameters for 22nm technology according to ITRS. We assume all electrical links are optimized for delay and the injection rate to be 0.1. Moreover, we include the energy dissipated in both planar and vertical links (communicating with all layers). Furthermore, we incorporate the power dissipated within the router buffers, except for virtual channel allocation. The energy for planar link is conservatively obtained as 0.15 pJ/bit for Firefly, 0.075 pJ/bit for mesh, and

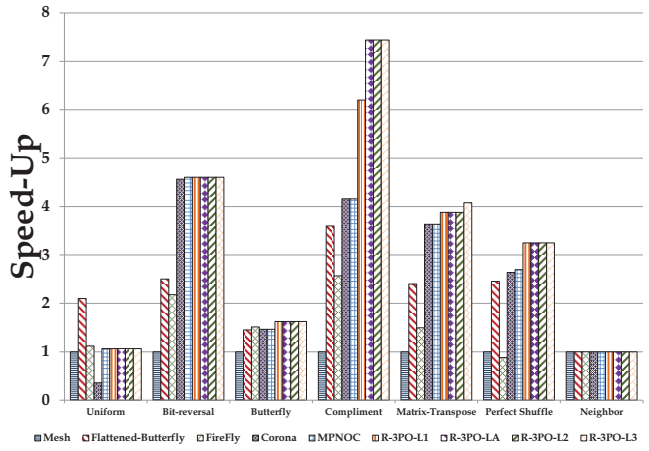


Figure 9: Simulation results showing normalized saturation throughput for seven traffic patterns for 256 cores.

0.15 pJ/bit per router bypass for Flattened-Butterfly under low swing voltage signalling [11]. The link energy dissipation depends on the location of the source and destination for Flattened-Butterfly. For a 10-layer chip, the vertical via is determined as ∼100- 200μm [15], which is significantly less than planar links. As a result, the energy consumption in vertical links are very small. We neglect it when we calculate our electrical link power model. We calculate the energy dissipated for a 10 × 10 router to be 0.42 pJ/bit, 8 × 8 router to be 0.30 pJ/bit [11], and 5 × 5 router will be 0.22 pJ/bit [11]. This is the energy dissipated per hop of communication. For each bit of optical transmission, we need to provide electrical back end circuit for transmitter and receiver. We assume the O/E and E/O converter energy is 100fJ/b, as predicted in [17]. For RC power dissipation, each RC optimizes performance by analyzing few bits (2-16) of information every 1300 cycles. While the data packets are as large as 512 bits of information, the static and dynamic power impact of the reconfiguration controller is negligible in comparison to the actual data movement.

5.3.2. Optical Energy and Loss Model The optical power budget is the sum of the laser power and the power dissipated in the MRRs. The laser power is determined by $P_{laser} = P_{rx} + C_{loss} + M_s$ where P_{laser} is the required laser power, P_{rx} is the receiver sensitivity, C_{loss} is the channel loss and M_s is the system margin. In order to perform an accurate comparison with the other two optical architectures, we use the same optical device parameters and loss values provided in [3, 1], as listed in Table 2. In this paper, we assume a flat thermal model that requires ring resonator heating power. However, this power can be lower as heating power can be shared by an array of rings [21], however this depends strongly on the actual layout of the ring resonators. Recent work has also demonstrated that flat thermal profile may not be practical and could increase off-resonance coupling losses [19, 15]. In this work, we show a preliminary analysis of the ring heating power which is a conservative model and more aggressive models can reduce this power [21, 19]. In addition, we assume a BER of 10^{-12} for each optical link and the Signal-Noise-Ratio (SNR) is given by [27]

$$BER = \frac{1}{2} - \frac{1}{2} erf(0.354\sqrt{SNR}) \quad (4)$$

Table 2: Electrical and optical power losses for select optical components.

Component	Value	Unit
Laser efficiency	5	dB
Coupler (Fiber to Waveguide)	1	dB
Waveguide	1	dB/cm
Splitter	0.2	dB
Non-Linearity	1	dB
Ring Insertion & scattering	1e-2 - 1e-4	dB
Ring Drop	1.0	dB
Waveguide Crossings	0.5	dB
Photo Detector	0.1	dB
Ring Heating	26	μW/ring
Ring Modulating	500	μW/ring
Receiver Sensitivity	-26	dBm

Table 3: Electrical power dissipation for various photonic interconnects.

	Corona	Firefly	R-3PO	Mesh
Link(electric)	-	0.15pJ/b	-	75fJ/b
Router	0.22pJ/b	0.30pJ/b	0.22pJ/b	0.22pJ/b
O/E, E/O	100fJ/b	100fJ/b	100fJ/b	-
Optical loss	-25.2dB	-17.6dB	-16dB	-
Power(λ)	0.81mW	0.14mW	0.10mW	-
Laser power	13.6W	2.4W	6.1W	-
Ring heating	26W	6.5W	27.5W	-

and the minimum power for a given SNR is

$$SNR = \frac{Po \cdot \eta}{NEP \cdot \sqrt{f}}, \quad (5)$$

where η is the quantum efficiency of the detector, NEP is the Noise-Equivalent-Power, and f is the transmission frequency [27]. Using the above equations, we determined the SNR in R-3PO to be 176.42.

Based on the energy model discussed in the previous section, we calculate the energy parameters of all four architectures as shown in Table 3. We test uniform traffic with 0.1 injection rate on the all architectures and obtain the energy per-bit as shown in Figure 11. Although Firefly has $\frac{1}{4}$ as many MRRs as Corona and R-3PO, which results in $\frac{1}{4}$ energy consumption per bit on ring heatings, it still consumes more energy than R-3PO and CORONA due to the overhead of electrical routers and links. In general, R-3PO saves 6.5%, 23.1%, 36.1% energy per bit compared to Corona, Firefly, and mesh respectively. R-3PO has a slight increase in power dissipation over MPNoCs due to the additional MRRs required for reconfiguration. The total network power for each application varied between 4 Watts to 6 Watts for 64-core simulation and 16 Watts to 24 Watts for 256-core simulation.

5.3.3. Laser Power Variations The optical losses shown in Table 3 are mostly conservative estimates that may not reflect the actual losses in future photonic devices. Figures 12(a) and 12(b) illustrate the impact on laser power when four optical parameters, namely receiver sensitivity, ring filter loss, wavelengths and waveguides are changed. We choose these four parameters from Table 3 as we believe they will have the greatest impact on the total laser power. We evaluate the variation in laser power with receiver sensitivity and the number of

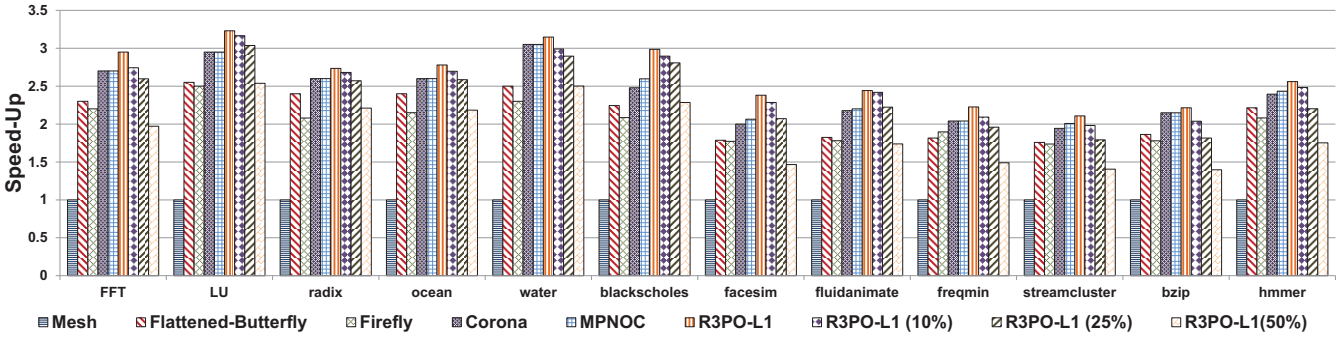


Figure 10: Speed-up for 64-core using SPLASH-2, PARSEC and SPEC CPU2006 traffic traces using 64 wavelengths for R-3PO-L1 with 10%, 25% and 50% faults in the channel.

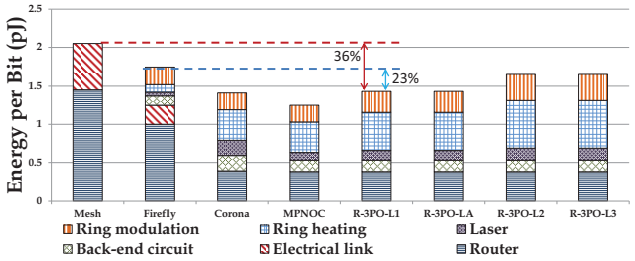


Figure 11: Average energy per-bit for electrical and photonic interconnects.

wavelengths in Figure 12(a). It should be noted that the bandwidth for each wavelength configuration is maintained at 640 Gbps in order to evaluate the laser power variation. Figure 12(a) shows that the laser power increases as the receiver sensitivity decreases because power per bit increases at the receiver. For example, a 6 dBm decrease in receiver sensitivity (-20 dBm) would result in a 4× increase in total laser power. Clearly, the receiver sensitivity has the greatest impact on the total network power for R-3PO, a low sensitivity receiver can increase the external laser power. Figure 12(b) shows the variations in laser power with the ring filter loss and the number of waveguides. From the figure, the increase in total laser power from waveguide losses has greater impact than the ring filter losses. This is due to the optical signal traversing several centimeters before arriving at the photodetector. For example, a 0.5 dB increase in waveguide loss (1.8 dB/cm) would more than double the total laser power.

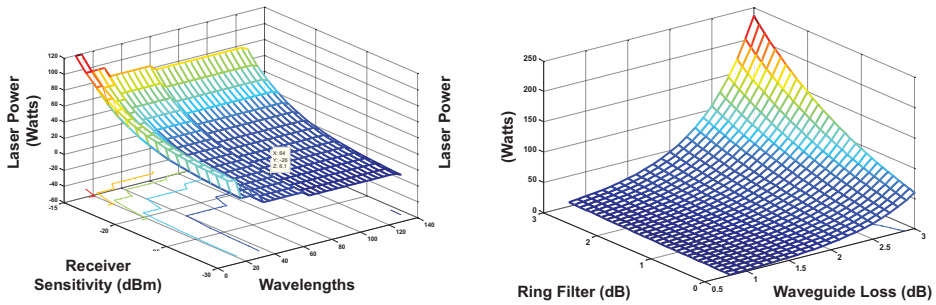
5.3.4. R-3PO Energy-Delay Product: In this paper, we propose different configurations of R-3PO that have different degrees of reconfiguration (increases bandwidth) and dissipate different energy. As such, the increase in performance due to more reconfiguration options may come at the price of higher energy dissipation. Figure 13 evaluates the energy-delay product (EDP) for all R-3PO configurations using the Splash-2, PARSEC, and SPEC CPU 2006 benchmarks. From the figure, it can be seen that R-3PO-L1 and R-3PO-LA have the least EDP. This is due to the fact that the slight increase in energy per bit over MPNoC and R-3PO-L1 is offset by the increase in performance over other networks. On the other hand, R-3PO-L2 and R-3PO-L3 have the highest EDP among all optical networks. This is obvious as these two networks have the highest energy per bit and there is only a slight increase in performance when compared to R-3PO-L1, R-3PO-LA and Corona. Mesh has the highest EDP as it is the worst network in terms of performance and has the highest energy per bit. When Flattened-Butterfly and Firefly are compared, their EDP have similar values. Firefly consumes lesser energy than Flattened-Butterfly although its performance is also proportionally lower than Flattened-Butterfly.

5.4. Area Analysis

In this subsection, we analytically compare the optical and electrical area overhead of R-3PO to Firefly [23] and Corona [29] photonic interconnects. For the optical area overhead, we consider the area required for all waveguides, MRRs and photodetectors. For the electrical layer, we consider the area required for all routers, electrical links and electrical receiver circuitry. Table 4 shows the area overhead of both optical and electrical components used in the area calculation. From Table 4, each router and electrical link values were obtained from Orion 2.0 by scaling 32 nm technology values to 22 nm technology. From our evaluation, we observe that both Corona and Firefly require 10% more optical area than R-3PO. This may be counter-intuitive, but R-3PO uses decomposed crossbars that permit waveguides in R-3PO to be shorter than the long serpentine waveguides used in both Corona and Firefly. In terms of electrical layer area overhead, R-3PO consumes 4X more electrical area than Corona. As each tile is connected to four optical layers to facilitate inter-group communication, each tile in turn should have the ability to receive four signals instead of one as in Corona. However, when R-3PO is compared to Firefly in terms of electrical area overhead, Firefly consumes about 75% more area. In Firefly, the electrical network can simultaneously receive from seven sets of optical receivers at once due to SWMR organization. R-3PO combines both MWSR (Corona) and SWMR (Firefly) communication channels, thereby increasing the communication channels to each tile while reducing the optical area overhead. For the different configurations of R-3PO, the additional increase in area overhead when compared to MPNOC is marginal as a single MRR can be used to switch all wavelengths from one layer to the other [5]. For example, the increase in area overhead for R-3PO-L1 is less than 1% and the increase in area overhead for R-3PO-L3 is about 1%. Table 5 shows the total area overhead for each network.

6. Conclusions

In this paper, we propose R-3PO that uses emerging photonic interconnects and 3D stacking to reduce the optical power losses found in 2D planar on-chip networks by decomposing a large 2D photonic



(a) (b)

Figure 12: (a)Laser power variations as a function of (a) receiver sensitivity and the number of wavelengths, and (b) ring filtering loss and the number of waveguides.

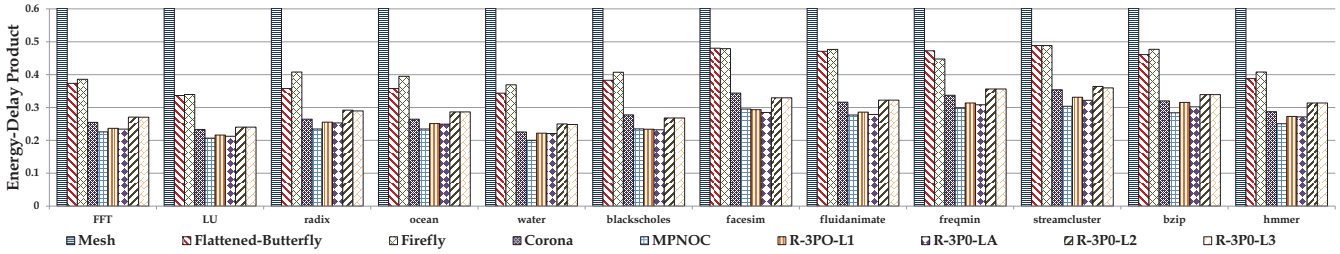


Figure 13: Simulation speed-up for 64-core using SPLASH-2, PARSEC and SPEC CPU2006 traffic traces using 8 wavelengths.

Table 4: Electrical and optical area overhead for select electrical and optical components

Component	Area
Electrical Link	0.0085 (mm^2)
Router (8×8)	0.128 (mm^2)
Photodetector receiver circuitry	0.02625 (mm^2)
Microring resonator	100(μm^2)
Photodetector	100(μm^2)
Waveguide	5.5 μm

Table 5: Electrical and optical area overhead for various networks (mm^2).

Network	Electrical	Optical
Firefly	712.25	78.5
Corona	107	78.5
R-3P0	407	70.9

crossbar into multiple smaller crossbars. In addition, we proposed a reconfiguration algorithm that maximizes the available bandwidth through run-time monitoring of network resources and dynamically re-allocating channel bandwidth. The reconfiguration algorithm improves performance by dynamically load balancing the network bandwidth and provides fault tolerance by bypassing faulty channels. For 64-core reconfigured network, our simulation results showed that the performance can be further improved by 10%-25% for Splash-2, PARSEC and SPEC CPU2006 benchmarks, where as simulation results for 256-core chip indicate a performance improvement of more than 25% while saving 6%-36% energy when compared to state-of-the-art on-chip electrical and optical networks.

Acknowledgment

This research was partially supported by NSF awards, ECCS-0725765, CCF-0915537, CCF-0915418, CCF-1054339 (CAREER) and ECCS-1129010 and by the IR/D program while Ahmed Louri was serving at the National Science Foundation.

References

[1] J. Ahn, M. Fiorentino, R. G. Beausoleil, N. Binkert1, A. Davis, D. Fattal, N. P. Jouppi, M. McLaren, C. M. Santori, R. S. Schreiber, S. M. Spillane, D. Vantrease, and Q. Xu, "Devices and architectures for photonic chip-scale integration," *Applied Physics A: Materials Science and Processing*, vol. 95, no. 4, pp. 989–997, June 2006.

[2] K. Aisopos, C.-H. O. Chen, and L.-S. Peh, "Enabling system-level modeling of variation-induced faults in networks-on-chip," in *48th Design Automation Conference (DAC)*, 2011.

[3] C. Batten, A. Joshi, J. Orcutt, A. Khilo, B. Moss, C. Holzwarth, M. Popovic, H. Li, H. Smith, J. Hoyt, F. Kartner, R. Ram, V. Stojanovi, and K. Asanovic, "Building manycore processor-to-dram networks with monolithic silicon photonics," in *Proceedings of the 16th Annual Symposium on High-Performance Interconnects*, August 27-28 2008.

[4] R. G. Beausoleil, P. J. Kuekes, G. S. Snider, S.-Y. Wang, and R. S. Williams, "Nanoelectronic and nanophotonic interconnect," *Proceedings of the IEEE*, vol. 96, no. 2, pp. 230–247, February 2008.

[5] A. Biberman, K. Preston, G. Hendry, N. Sherwood-Droz, J. Chan, J. S. Levy, M. Lipson, and K. Bergman, "Photonic network-on-chip architectures using multilayer deposited silicon materials for high-performance chip multiprocessors," *J. Emerg. Technol. Comput. Syst.*, vol. 7, pp. 1–25, July 2011.

[6] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite: Characterization and architectural implications," in *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, October 2008.

[7] N. L. Binkert, A. Davis, N. P. Jouppi, M. McLaren, N. Muralimanohar, R. Schreiber, and J. H. Ahn, "The role of optics in future high radix switch design," in *ISCA*, 2011, pp. 437–448.

[8] X. Chen, L.-S. Peh, G.-Y. Wei, Y.-K. Huang, and P. Pruncal, "Exploring the design space of power-aware opto-electronic networked systems," in

11th International Symposium on High-Performance Computer Architecture (HPCA-11), February 2005, pp. 120–131.

[9] M. Georgas, J. Leu, B. Moss, C. Sun, and V. Stojanovic, "Addressing link-level design tradeoffs for integrated photonic interconnects," in *CICC*, 2011, pp. 1–8.

[10] B. Grot, J. Hestness, S. W. Keckler, and O. Mutlu, "Express cube topologies for on-chip interconnects," in *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, 2009, pp. 163–174.

[11] A. B. Kahng, B. Li, L.-S. Peh, and K. Samadi, "Orion 2.0: A fast and accurate noc power and area model for early-stage design space exploration," in *in Proceedings of Design, Automation & Test in Europe Conference & Exhibition*, Nice, France, April 20-24 2009, pp. 423–428.

[12] J. Kim, W. J. Dally, and D. Abts, "Flattened butterfly: Cost-efficient topology for high-radix networks," in *Proceedings of 34th Annual International Symposium on Computer Architecture(ISCA)*, June 2007, pp. 126 – 137.

[13] A. K. Kodi and A. Louri, "Energy-efficient and bandwidth reconfigurable photonic networks for hpc systemss," *IEEE Journal of Selected Topics in Quantum Electronics*, vol. 17, pp. 384–395, April 2011.

[14] S. J. Koester, C. L. Schow, L. Schares, and G. Dehlinger, "Ge-on-soi-detector/si-cmos-amplifier receivers for high-performance optical-communication applications," *Journal of Lightwave Technology*, vol. 25, no. 1, pp. 46–57, January 2007.

[15] P. Koka, M. O. McCracken, H. Schwetman, X. Zheng, R. Ho, and A. V. Krishnamoorthy, "Silicon-photonic network architectures for scalable, power-efficient multi-chip systems," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, June 2010.

[16] P. Koonath and B. Jalali, "Multilayer 3-d photonics in silicon," *Opt. Express*, vol. 15, pp. 12 686–12 691, 2007.

[17] A. V. Krishnamoorthy, R. Ho, X. Zheng, H. Schwetman, J. Lexau, P. Koka, G. Li, I. Shubin, and J. E. Cunningham, "Computer systems based on silicon photonic interconnects," in *Proceedings of the IEEE*, vol. 97, no. 7, June 2009, pp. 1337–1361.

[18] A. Kumar, P. Kundu, A. P. Singh, L.-S. Peh, and N. K. Jha, "A 4.6tbits/s 3.6ghz single-cycle noc router with a novel switch allocator in 65nm cmos," in *ICCD 2007*, October 2007.

[19] Z. Li, M. Mohamed, X. Chen, E. Dudley, K. Meng, L. Shang, A. R. Mickelson, R. Joseph, M. Vachharajani, B. Schwartz, and Y. Sun, "Reliability modeling and management of nanophotonic on-chip networks," *IEEE Trans. VLSI Syst*, vol. 20, pp. 98–111, 2012.

[20] M. Martin, D. Sorin, B. Beckmann, M. Marty, M. Xu, A. Alameldeen, K. Moore, M. Hill, and D. Wood, "Multifacet's genreal execution-driven multiprocessor simulator (gems) toolset," *ACM SIGARCH Computer Architecture News*, no. 4, pp. 92–99, November 2005.

[21] C. Nitta, M. Farrens, and V. Akella, "Addressing system-level trimming issues in on-chip nanophotonic networks," in *Proceedings of the 17th International IEEE Symposium on High Performance Computer Architecture*, 2011, pp. 122–131.

[22] Y. Pan, J. Kim, and G. Memik, "Flexishare: Channel sharing for an energy-efficient nanophotonic crossbar," in *Proceedings of the 36th annual international symposium on High Performance Computer Architecutre (HPCA)*, 2010, pp. 1–12.

[23] Y. Pan, P. Kumar, J. Kim, G. Memik, Y. Zhang, and A. Choudhary, "Firefly: Illuminating future network-on-chip with nanophotonics," in *the Proceedings of the 36th annual International Symposium on Computer Architecture*, 2009.

[24] K. Preston, S. Manipatruni, A. Gondarenko, C. B. Poitras, and M. Lipson, "Deposited silicon high-speed integrated electro-optic modulator," *Opt. Express*, vol. 17, pp. 5118–5124, 2009.

[25] N. Sherwood-Droz, K. Preston, J. S. Levy, and M. Lipson, "Device guidelines for wdm interconnects using silicon microring resonators," in *Workshop on the Interaction between Nanophotonic Devices and Systems (WINDS), co located with Micro 43*, December 5th 2010, pp. 15–18.

[26] V. Soteriou, N. Eisley, and L.-S. Peh, "Software-directed power-aware interconnection networks," *ACM Trans. Archit. Code Optim.*, vol. 4, March 2007.

[27] T. H. Szymanski, "Optical link optimization using embedded forward error correcting codes," *Jornal of Selected Topics in Quantum Electronics*, vol. 9, no. 2, pp. 647–656, March/April 2003.

[28] D. Vantrease, N. Binkert, R. Schreiber, and M. H. Lipasti, "Light speed arbitration and flow control for nanophotonic interconnects," in *MICRO 42: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2009, pp. 304–315.

[29] D. Vantrease, R. Schreiber, M. Monchiero, M. McLaren, N. Jouppi, M. Fiorentino, A. Davis, N. Binker, R. Beausoleil, and J. H. Ahn, "Corona: System implications of emerging nanophotonic technology," in *Proceedings of the 35th International Symposium on Computer Architecture*, June 2008, pp. 153–164.

[30] S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta, "The splash-2 program: Characterization and methodological considerations," 1995, pp. 24–36.

[31] X. Zhang and A. Louri, "A multilayer nanophotonic interconnection network for on-chip many-core communications," in *Proceedings of the Design and Automation Conference (DAC)*, June 2010.

[32] X. Zheng, F. Liu, J. Lexau, D. Patil, G. Li, Y. Luo, H. Thacker, I. Shubin, J. Yao, K. Raj, R. Ho, J. Cunningham, and A. Krishnamoorthy, "Ultra-low power arrayed cmos silicon photonic transceivers for an 80 gbps wdm optical link," in *in Optical Fiber Communication Conference*, March 2011.

2012 IEEE/ACM 45th Annual International Symposium on Microarchitecture

Addressing End-to-End Memory Access Latency in NoC-Based Multicores*

Akbar Sharifi, Emre Kultursay, Mahmut Kandemir and Chita R. Das
The Pennsylvania State University
University Park, PA, 16802, USA
{akbar,euk139,kandemir,das}@cse.psu.edu

Abstract

To achieve high performance in emerging multicores, it is crucial to reduce the number of memory accesses that suffer from very high latencies. However, this should be done with care as improving latency of an access can worsen the latency of another as a result of resource sharing. Therefore, the goal should be to balance latencies of memory accesses issued by an application in an execution phase, while ensuring a low average latency value. Targeting Network-on-Chip (NoC) based multicores, we propose two network prioritization schemes that can cooperatively improve performance by reducing end-to-end memory access latencies. Our first scheme prioritizes memory response messages such that, in a given period of time, messages of an application that experience higher latencies than the average message latency for that application are expedited and a more uniform memory latency pattern is achieved. Our second scheme prioritizes the request messages that are destined for idle memory banks over others, with the goal of improving bank utilization and preventing long queues from being built in front of the memory banks. These two network prioritization-based optimizations together lead to uniform memory access latencies with a low average value. Our experiments with a 4×8 mesh network-based multicore show that, when applied together, our schemes can achieve 15%, 10% and 13% performance improvement on memory intensive, memory non-intensive, and mixed multiprogrammed workloads, respectively.

1. Introduction

From an architectural side, in parallel to increasing core counts, network-on-chip (NoC) is becoming one of the critical components which determine the overall performance, energy consumption and reliability of emerging multicore systems [14, 31, 35]. While NoC helps to improve communication scalability of data, it also contributes to memory access latencies. Consider as an example a data read access in an NoC-based multicore. Such an accesses first checks L1 cache, and if it misses there, accesses L2 cache by traversing the NoC (assuming an S-NUCA-based [15] cache space management). If it misses in L2, it takes another trip over the NoC to reach the target memory controller determined by the address of the requested data. Depending on the current status of the queue in the target memory bank, it waits in the queue for some amount of time before reaching the DRAM. After finally reading the data from the DRAM, the same set of components are visited in the reverse order until a copy of the data is brought into the core. Clearly, each of the components in this round-trip access path (L1, L2, network, bank queue, and memory access) contributes to the end-to-end latency of this data access.

From an application side, interferences across simultaneously executing applications on various shared resources (e.g, NoC, memory controllers) can lead to high variances across the latencies of the off-chip accesses made by an application [8, 18, 27]. Specifically, while one request can be lucky to retrieve its data very quickly, another request of the same application can get delayed in the network and/or the memory controller queue, suffering a much larger round-trip latency. This variance can in turn lead to degradation in overall system performance as requests that experience much higher latencies than the average can block the progress of the application. To achieve high performance, it is crucial to reduce the number of memory accesses that observe very high latencies, but this should be done with care as improving the latency of an access can worsen the latency of another as a result of resource sharing. Therefore, from an overall system performance point of view, instead of each application aggressively trying to minimize the latencies of all its memory accesses, it is better to have each application *balance* the latencies of its memory accesses.

Motivated by these observations, we propose two *network prioritization* schemes that can cooperatively reduce *end-to-end memory access latencies* in NoC-based multicores. One of these schemes targets memory response messages (i.e., messages from the memory controller), whereas the other targets memory request messages (i.e., messages from the core side). Our first scheme prioritizes memory response messages such that, in a given period of time, messages of an application that experience higher latencies than the average message latency of that application are expedited. This helps us reduce the number of off-chip requests with high latencies and achieve a more uniform memory latency pattern. While one might want to apply a similar optimization for memory request messages as well, it is hard to identify whether a data access will experience high latency at the time of the request generation as it has not even entered the network yet. Instead, our second scheme approaches problem from another angle, where it expedites request messages to improve memory bank utilization. More specifically, observing that at any given time frame some banks are heavily loaded whereas others are idle, our second scheme prioritizes (at network routers) the request messages that are destined for idle banks over the others. This helps to improve bank utilization and prevent long queues in front of the memory banks. These two network prioritization-based optimizations, when applied together, result in uniform memory access latencies with a low average value.

We implemented our schemes in a simulation environment and tested their effectiveness using a diverse set of multiprogrammed workloads. Our experiments with a 4×8 mesh network-based multicore show that our first scheme is very effective in reducing the number of off-chip requests with very high latencies. As a result, we are able to achieve 11%, 6%, and 10% performance improvement on memory intensive, memory non-intensive, and mixed multiprogrammed workloads, respectively. When both our schemes are used together, these improvements jump to 15%, 10% and 13%, in the same order. Our experimental evaluation also reveals that the proposed schemes consistently generate good results under different

*This research is supported in part by NSF grants #1213052, #1152479, #1147388, #1139023, #1017882, #0963839, #0811687 and a grant from Microsoft Corporation.

1072-4451/12 $26.00 © 2012 IEEE
DOI 10.1109/MICRO.2012.35

IEEE computer society

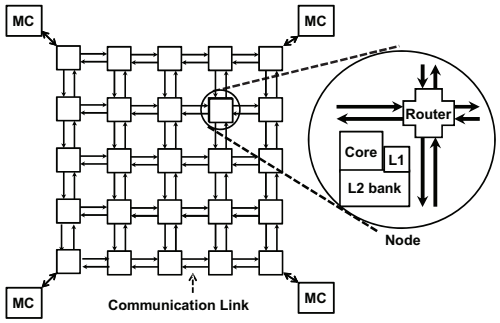


Figure 1: A 5×5 NoC based multicore.

values of the main simulation parameters.

The rest of this paper is structured as follows. We provide an overview of our target multicore architecture and the main motivations behind our optimizations in Section 2. In Section 3, we give the details of our proposed schemes and discuss our network prioritization implementation. Section 4 presents our experimental framework and discusses our results. We contrast our approach with related work in Section 5, and conclude the paper in Section 6.

2. Background and Motivation

2.1. Target Multicore System

Figure 1 shows the high level view of an NoC-based multicore system that we target in this paper. As can be observed, this multicore system has a number of nodes arranged in a 2D grid (an example present day product using a similar organization is TILEPro36 by Tilera [1]). In this architecture, each node contains a processing element and a hierarchy of caches; only two levels of caches are shown in Figure 1. The shared last level cache (L2) has a banked organization where the L2 space in each tile represents a separate bank. The mapping of data to L2 cache banks is very important as far as performance is concerned, and in one mapping scheme (SNUCA [15]) which is also used in this paper, each cache block-sized unit of memory is statically mapped to one of the cache banks based on its address [16]. This address-based mapping results in an interleaving of cache blocks across cache banks. The main memory is connected to this multicore via memory controllers (MCs) on the corners (or sides), each controlling a single memory channel with possibly many memory modules (DIMMs) connected. Each channel consists of one or more ranks, and each rank is a group of memory banks. Different memory banks can be accessed at the same time, but they share common address and data buses. More details on the internal structure of the main memory can be found in [3, 9, 29]. Note that the target memory bank for each memory request is fixed and depends on the OS address mapping. If more than one memory request are queued in a memory controller trying to access the same bank, the memory scheduler decides which one should be served first [33, 34].

2.2. Memory Accesses in an NoC-based Multicore

Once a core issues a memory request, first the local L1 is checked for the presence of the requested data and if this fails, the request is forwarded to the corresponding L2 bank via the on-chip network. Finally, if the data is not found in the L2 bank, an off-chip memory access is made, again using the on-chip network. Figure 2 depicts this flow in our target architecture. One can make several critical observations from this figure. First, the L2 access latency is a function

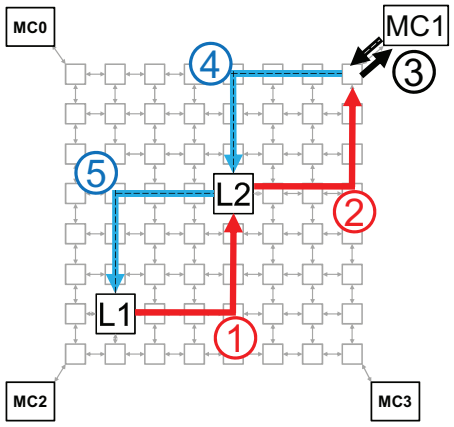


Figure 2: Detailed flow of a memory access. In order to look-up the corresponding L2 bank, a request message is sent from L1 to L2 (path1). If the data is a hit in L2 cache, then the response data is sent back to the L1 cache (path5). In case of an L2 miss, the request is forwarded to the memory controller (path 2). The memory controller schedules this request (path3) and the response is sent first to the L2 cache (path 4) and then to the L1 cache (path 5).

of the distance between the requesting core (L1) and the target L2. It is to be noted that, longer this distance gets, higher the chances for network contention. Second, the total latency to serve an L2 miss (i.e., the overall memory latency) includes memory controller queuing and memory service latencies (i.e., the memory access latency) as well as the latency of four trips on the network: L1 to L2, L2 to memory controller (MC), MC to L2, and L2 to L1. As a result, network latency can play a significant role in overall memory access latency (in fact, our experiments show that, even in a 4×8 multicore, cumulative network latency can be comparable to memory access latency). Third, the time spent in an MC depends on other requests that target the same MC as well as the (memory request) scheduling policy adopted by the architecture. Fourth, at any given time, there will be many memory requests traveling on the network. These requests contend for the shared network resources, resulting in additional delays. For example, a message from a core destined to an L2 may contend for the same link/router with another message going from an L2 to an MC. Lastly, the time it takes to service a memory request (after it gets its turn from the memory scheduler) depends on whether it results in a row buffer hit or not.

2.3. Out-of-Order Execution

In our target system, each processing element (core) is a high performance out-of-order processor. It can issue multiple instructions by looking at its instruction window. As a result, if these instructions need to access the main memory, the memory requests can be issued at the same time with the hope of improving the overall performance by not stalling the processor for a single memory request. This is referred to as Memory Level Parallelism (MLP) [11, 28]. Note that although memory requests may return out-of-order, the instructions are committed in-order and therefore, a higher memory access latency for an instruction may affect the performance of an application significantly [10]. As an example, suppose that an application issues four load instructions (load-A, load-B, load-C and load-D, as shown in Figure 3) that need to access memory. As discussed, these instructions may be ready to commit at different times due to the network and memory latencies and also whether they are hits or misses

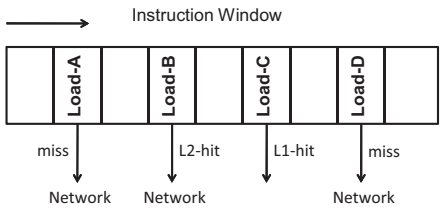


Figure 3: An instruction window with four outstanding memory requests.

in the on-chip caches. Note that in the instruction window shown in Figure 3, the load instruction (e.g, load-A) that needs to wait for a longer time to access the data (as compared to the other instructions) becomes a bottleneck for the application. The chances that a memory request will be a bottleneck depend on its delay and also other outstanding memory requests from the same application. For example, if an application is not memory intensive (an application with a low MPKI [misses per kilo instructions]), the likelihood that one of its off-chip memory requests would be a bottleneck is high. This is why previously-proposed memory scheduling schemes [17, 18] and network prioritization [7], higher priorities are given to the requests coming from this type of applications. However, these prior works do not consider the network and memory access latencies together. For instance, in the application-aware network prioritization scheme proposed in [7], the time it takes to access the memory is assumed to be constant, whereas in reality memory requests (when they reach the memory controller in an NoC based multicore) may face different latencies due to queuing.

2.4. Motivations

Our proposed approach consists of two network prioritization based schemes (Scheme-1 and Scheme-2). These schemes are proposed based on two main motivations presented in this section. Specifically, Motivation-1 and Motivation-2 below motivate for Scheme-1 and Scheme-2, respectively.

2.4.1. Motivation 1: Some Memory Accesses Experience Much Higher Delays than Others As illustrated in Figure 2, a round-trip memory access has five paths/stages. The total delay of each access is equal to the sum of the delays on these paths. To have a better understanding of how much time each memory request spends on different stages/paths on average, we simulated a 4×8 NoC-based multicore with 4 memory controllers attached to the corners, and each core executing an application from the SPEC2006 benchmark suite [12] (see workload-2 in Table 2; more details on the experimental configuration and workloads are given in Section 4.1). Figure 4 plots the average round-trip delays of the off-chip memory accesses issued by one of the cores (executing application *milc*), broken into individual components The values on the x-axis are the "delay ranges" and each bar gives the "average latency" of the memory accesses which experienced a delay value within the corresponding range on the x-axis. For example, the first bar from the left gives the average round-trip delay of the memory accesses with delay falling between 150 cycles and 200 cycles. Each bar is partitioned into five parts, which shows the breakdown of the average delays. These partitions from bottom to top represent: (1) the average network delay between L1 and L2 banks (path-1 in Figure 2), (2) the average delay from the L2 bank to the memory controller (path-2 in Figure 2), (3) the average delay added by the memory (queuing delay + memory access latency), (4) the average network delay from the

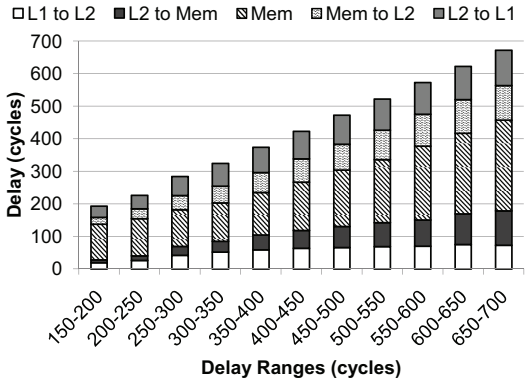


Figure 4: The average delays of the off-chip memory accesses issued by one of the cores (executing milc in workload-2).

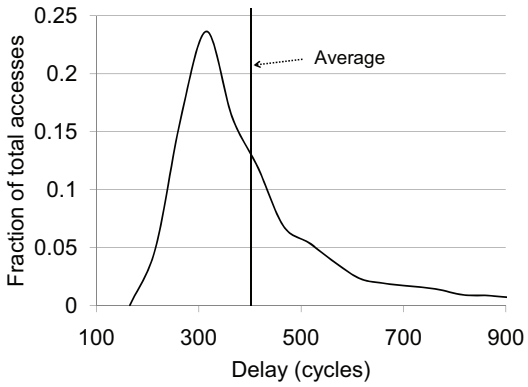


Figure 5: The latency distribution of the memory accesses issued by one of the cores (executing application milc in workload-2).

memory controller to the L2 bank (path-4 in Figure 2), and (5) the average network delay form the L2 bank to the source core. As can be seen from this figure, due to the large network and memory queuing delays, some messages experience significantly larger delays as compared to others.

However, the numbers of accesses in different delay ranges are not the same. Figure 5 plots the distribution of the memory accesses across different delay regions. The values on the x-axis correspond to different latencies observed during execution, and the y-axis shows the percentage of the total memory requests for different latencies. The area under the curve between $x = d_1$ and $x = d_2$ gives the fraction of the memory accesses that have a total latency between d_1 and d_2. As can be observed from this plot, the latencies of most of the accesses are around the average, but there are few accesses with very large delays (600 cycles or more). Unfortunately, these high latency accesses can be very problematic and degrade overall application performance significantly.

Our first scheme identifies these (high latency) accesses (after the DRAM access is completed) and attempts to reduce their latencies with the goal of improving the overall system performance. Specifically, our proposed scheme tries to expedite these slow messages on their return path by giving them a higher priority in the on-chip routers. In other words, we try to make the latencies of the different memory accesses *issued by the same application* as uniform as possible.

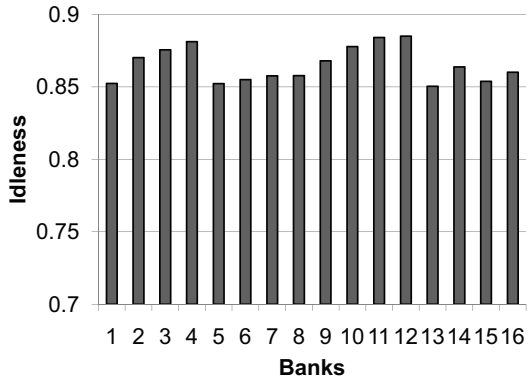


Figure 6: The idleness of different banks of a memory controller.

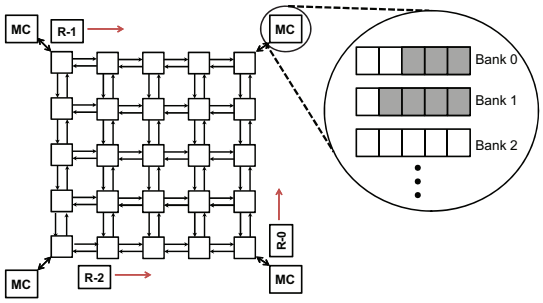


Figure 7: A snapshot that shows the states of the three banks controlled by one of the memory controllers.

2.4.2. Motivation 2: Bank Loads are Non-Uniform Once a memory request passes through the on-chip network and reaches the memory controller, it may face a queue in front of the target memory bank. We observed during our experiments that, at any given time, different banks can have very different queue lengths (i.e., the number of pending requests). For instance, some banks may be idle for a period of time, whereas other banks may be serving their requests. Figure 6 plots the *average idleness* of different banks of one of our memory controllers (again in our 4×8 multicore). To compute the average idleness, the queue of each bank is monitored at fixed intervals. For example, an average idleness of 0.8 in this plot implies that 80% of the times, the bank is monitored, there has been no request in its queue.

To explain how this observation can be exploited, consider Figure 7 which shows the states of three banks controlled by one of the memory controllers at time t_1. As can be seen, Banks 0 and 1 have requests to be serviced, whereas Bank-2 is idle. Assume further that there are three memory requests (from L2 misses) in the on-chip network destined for these banks as shown in Figure 7 (R-0, R-1 and R-2 which are destined for Bank 0, 1 and 2, respectively). In this example, the memory utilization could be improved if R-2 could reach its memory controller faster so it can be serviced by Bank-2 immediately. Our proposed scheme exploits this load variation across different banks by using the *local history* at each node to accelerate the memory requests on the network destined for the idle banks.

3. Our Approach

In this paper, we propose two network prioritization schemes that consider the states of the memory bank queues and the overall memory access delays in NoC-based multicores. In this section, we first explain how our schemes work and then discuss the implementation details of the specific network prioritization method we employ. In our proposed approach, Scheme-1 focuses on the response messages coming back from the memory and Scheme-2 considers request messages destined for the off-chip memory. We assume a one-to-one mapping between applications and cores.

3.1. Scheme-1: Reducing Variations across Memory Accesses

This scheme is based on the motivation discussed in Section 2.4.1. Recall from Figure 5 that some memory accesses issued by an application can take much longer times to complete compared to other accesses from the same application (we call them late accesses). As an example, in Figure 5, about 10% of the accesses have delays larger than 600 cycles (while the average latency is about 350 cycles). As mentioned earlier, these late accesses can severely degrade the performance of applications. Suppose for example that an application executes $a = b + c$. In this instruction, if one of the operands becomes ready only after a long time, this will delay the completion of this instruction as well as all instructions that depend on it. The main goal behind our Scheme-1 is to mitigate the impact of the late accesses by helping them reach their destinations faster.

As shown in Figure 4, three different factors can cause memory accesses to be late in the system: (1) the network latency from the source cache bank to the corresponding memory controller, (2) the memory access latency (which includes both queuing delay and the time it takes to access the DRAM), and (3) the network latency from the memory controller to the source cache bank. Therefore, in order to reduce the delay experienced by the late accesses, one can potentially target these three delays. The first and third delays can be reduced by giving higher priorities to the request and the response messages in the network. However, for the first part (network latency after an L1 miss, from the L1 bank to the memory controller), it is not known that a memory request is going to be a late memory access, since the total delay depends mainly on the memory response time (factor 2) and the network delay of the response messages (factor 3), as illustrated in Figure 4. To reduce the response time of the memory (factor 2), late memory requests could be scheduled faster at the bank queues. However, prioritizing these requests would harm other late requests in the same bank. In other words, if one request in a bank queue suffers long queuing delay, it is most likely that this bank is a "common bottleneck" for all co-running applications, and consequently prioritizing the requests from one application over others would not help in a global sense.

Motivated by these observations, Scheme-1 attempts to reduce the delays coming from the third factor, since right after the memory controller the delays accumulated so far can give a good *estimate* of whether a memory access is going to be "late" or not. In this scheme, each memory message (packet) has a field that maintains the age ("so-far delay") of the corresponding access (based on the delay at each router/stage, this field is updated over the round-trip of the memory access). Note that the variations in network latencies (the first factor) can come from two contributions: (1) the distances in the network from L1 to L2 and from L2 to MC, and (2) the network traffic. Both these sources are captured by our so-far delay field, and consequently handled by our scheme.

To illustrate how Scheme-1 works, let us explain it through a simple example. In Figure 8, suppose that MC1 has received a memory request from core-1, and R0 is the corresponding response message

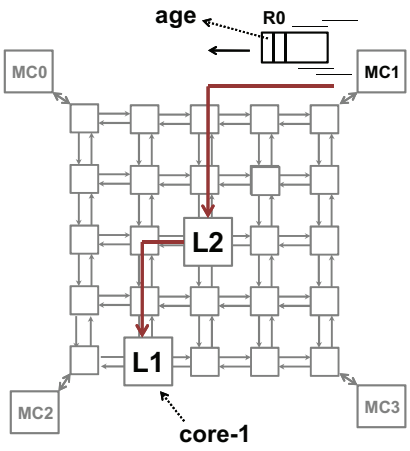


Figure 8: An example to illustrate how Scheme-1 works. MC1 has received a memory request from core-1, and R0 is the corresponding response message for that request after it is serviced by MC1.

for that request after it is serviced by MC1. Once R0 is ready, the age field (the so-far delay) in R0 gets updated based on the delay that has been imposed by the memory. After that, the updated so-far delay is compared with a threshold value Th. If it is larger than Th, this memory access is considered to be late and, on the return path, R0 will have a higher priority to reach its destinations (the paths from MC1 to L2 and L2 to L1 illustrated in Figure 8). Note that in the memory controllers, each core/application is associated with a threshold value. These threshold values are determined and sent to the MCs by the cores periodically (every 1ms) over the execution and they are also prioritized over other requests (each MC has storage to keep the threshold values). Since this happens every 1ms, the overhead on the other messages is small. In our default configuration, each core sets its threshold value to $1.2 \times Delay_{avg}$, where $Delay_{avg}$ is the *average delay* of the off-chip memory accesses that belong to that application/core. Once a response message for an off-chip memory access comes back, the round-trip latency of that off-chip memory access is read from the message (the age field) and $Delay_{avg}$ is updated accordingly. Consequently, during the course of execution, the threshold value used for each application changes dynamically.

Note also that we compute the priority of a response message when the message is about to be injected into the network by the memory controller. However, at that time instant, the memory controller only knows the so-far delay of the request, and therefore, any delay threshold that will be used in calculating the priority must be defined based on the average delay up to and including the memory access latency ($Delay_{so-far-avg}$). For this reason, we set our threshold to $1.2 \times Delay_{avg}$ (which is almost equivalent to $1.7 \times Delay_{so-far-avg}$; both averages are marked in Figure 9 using vertical solid lines). Figure 9 plots two delay distributions of the memory accesses (issued by the core executing application milc from workload-2). Specifically, the dashed curve shows the distribution of the round-trip delays of the messages, while the solid curve plots the distribution of the so-far delays at the point right after the memory controller (i.e., when the data is read from the memory, and is about to be injected into NoC).

Implementation Details. As discussed earlier, Scheme-1 needs each memory access message to have a field (called "age" field) cap-

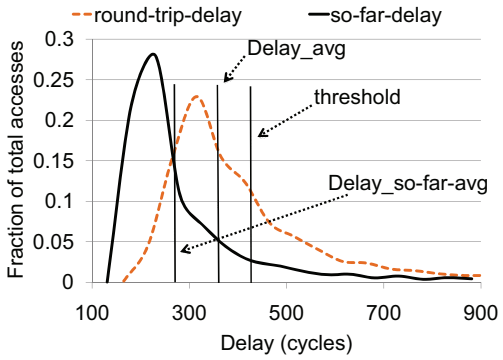


Figure 9: Latency distributions of the memory accesses (issued by the core executing application milc from workload-2). The solid curve shows the distribution of the so-far delays at the point right after the memory controller, while the dashed curve plots the distribution of the round-trip delays of the messages.

turing the so-far delay of the message in the NoC. We assume that this field is 12-bit long and its value is in cycles. We believe a 12-bit field is sufficient in our work, since it is very rare that the round-trip of an off-chip message takes more than $4096-1$ cycles in the system. We assume that the header flit has room for the so-far-delays (12-bit) in a 128-bit header. At each router/memory-controller, once the message is ready to be sent out, the age-field value is updated as follows:

$$\text{age} = age + \frac{(local\_time - message_{entry\_time}) \times FREQ\_MULT}{local\_frequency}, \quad (1)$$

where $local\_time$ is the local router/MC clock cycles, $message_{entry\_time}$ is the time when the message enters the routers/MCs, $FREQ\_MULT$ is a constant value (used to work in integer domain, instead of fractions of cycles) and $local\_frequency$ is the local operating frequency. As one can observe, in each hop, first, the local delay of each incoming message is computed ($local\_time - message_{entry\_time}$) and then this value is added to the age-field of the message. Note that dividing the local delay by the $local\_frequency$ allows our scheme to work when routers/MCs operate at different frequencies. Note also that, the entry time of each message is not transferred over the network, and each router keeps track of its own local delay and computes the so-far delay *locally*. Instead of updating the age field at each hop, one may choose to employ a global clock to compute the so-far-delays. In this method, each message is labeled with the starting time and, at the destination, the starting time is subtracted from the current time to obtain the so-far-delay. However, in this mechanism, all the nodes need to have clocks and these clocks should be synchronized.

3.2. Scheme-2: Balancing Memory Bank Loads

Based on Motivation-2 discussed in Section 2.4.2, the main goal behind this scheme is to increase the utilization of the memory banks in the system. To do this, the requests destined for the idle banks are given higher priorities in the network to reach their target memory controllers faster. However, the number of requests queued in each memory bank varies over the execution, and more importantly, a core in our 2D mesh does not know about the states of the memory banks. In other words, *no global* information is available to the

nodes to help them decide whether different memory requests they issue are destined for idle banks or not. To address this problem, in our proposed scheme, each node exploits "*local information*" to estimate the pressure imposed by the requests on the memory banks. Specifically, each router keeps and updates a table (called "Bank History Table") that records the number of off-chip memory requests sent from this router to each bank in the last T cycles. An off-chip request is given a higher priority in the on-chip network if no message has been sent to the same bank according to the value in the bank history table. In Section 4, we present results that show how bank idleness is reduced when Scheme-2 is employed.

Note that due to Scheme-2 and the variation in the network latencies, different requests that are destined for the same memory bank may not reach the bank queue in order (based on their ages). This may cause more delays for the late accesses. Message ordering can be handled by the memory scheduler, since the scheduler knows the timing information for all arriving messages.

3.3. Network Prioritization Implementation

Further, that would be very costly to use stronger routers and more network resources. Our schemes can be used along with other architectural solutions to provide additional benefits. In this section, we give the implementation details of our prioritization method. In our NoC-based multicore, we assume a typical architecture with 5-stage pipeline for the on-chip routers [21]. We further assume flit-buffering and VC (virtual channel) credit-based flow control at every router [4]. In this architecture, each network message is split into several flits with fixed-lengths and the flow of the flits follows the wormhole switching protocol [6]. For further details, we refer the reader to [21].

Once a header flit enters a router, in the first stage of the pipeline, which is called buffer write (BW), the flit is written into the allocated input buffer. In the next stage (routing computation (RC)), the routing algorithm is invoked and the right output port is determined for the packet based on the position of the current router and the destination address extracted from the header flit. The next stage, called virtual channel allocation (VA), is where a free output virtual channel is reserved for the packet. After this stage, the flit needs to get the permission to use the crossbar switch (switch allocation (SA)). In the last stage, the flit traverses the switch (ST) to be sent over the physical link (LT). Note that only the header flit passes through these stages, and the body and tail flits skip the RC and VA stages as they simply follow the header flit.

In our approach, the messages to be expedited have higher priorities in VC and SW arbitrations (for virtual channel and switch allocations). The VC arbitration is done when an output VC is preferred by more than one input VC and, one of them should be granted. The SA arbitration on the other hand has two phases. In the first phase, only one of the ready VCs is selected per input port and, in the second phase, one VC is selected for each output port. The reason for these two arbitrations is that, at each cycle, only one flit can be sent from an input and each output is able to receive a single flit.

Although prioritization in the arbitration stages expedites the messages and helps them reach their destinations faster, there is a limit in this message acceleration, since it takes at least 5 cycles for the data flits to pass through a router in the network. To tackle this limitation, we also employ a mechanism called "pipeline bypassing" [21], which gives the late messages the opportunity to go through fewer number of pipeline stages in the routers. In this mechanism, once

Figure 10: Pipeline stages in baseline and pipeline bypassing.

the header flit of a message (with high priority) enters a router, in the same cycle, after it is written to the input buffer, one free output VC is allocated to it (VC allocation) and the switch is reserved for this flit (SW allocation) (in other words, the BW, RC, VA and SA stages are combined and performed in the first stage which is called the "setup stage"). Figure 10 illustrates the pipeline stages for the baseline router as well as the one with pipeline bypassing). If in this cycle, there is a conflict in selecting the free output VC and the switch allocation between this flit and a flit (with normal priority) in another VC, the flit with the higher priority is prioritized over the others. Body flits also use this "bypassing" only when the input buffer is empty once they enter the router (Note that the input buffers are always idle when the header flits enter).

To avoid starvation in the system, we also use "age fields" in the messages (which are also used in Scheme-1). In our prioritization scheme, flit A is prioritized over flit B in the cases discussed above, if (1) flit A has a high priority but flit B has the normal one, and (2) the age of message B is not more than T cycles greater than that of message A (flits A and B belong to messages A and B). Note that the routers also consider the local delays in addition to the age fields in the messages. "Batching" [8] is another method that can be used to avoid starvation. In this method, time is divided into intervals of T cycles and there is a field in each message that indicates the batching interval in which the message was injected to the network. The packets are prioritized if they belong to the same batch. However, this method requires a synchronization among the cores since they need to access a common global time.

4. Experimental Evaluation

In this section, we first explain our simulation framework and the different workloads that we used in our evaluations and then, present and discuss our experimental results.

4.1. Setup, Metrics, and Workloads

Setup: We use GEMS [25] as our simulation framework to evaluate our proposed schemes. This framework allows us to simulate an NoC-based multicore system. GEMS consists of two main components called Opal and Ruby, and employs Simics [24] as the base simulator. Opal implements an accurate model for out-of-order cores, and Ruby implements the network modules (the routers and the links) and the memory controllers. The instructions are executed by Opal and, if they require a memory access, a request message is injected to Ruby. Opal receives the response message once the requested data comes back from an L2 bank or one of the memory controllers (simulated by Ruby). Table 1 gives our baseline configuration. As shown in this table, our baseline system has 32 cores connected by a 4×8 mesh-based NoC, and also 4 memory controllers are connected to the four corners of the network. In our evaluation, we employ cache line interleaving where the consecutive lines of an OS page are mapped to different memory controllers. Note that this strategy helps to avoid creating hot spots in the memory controllers. In the results presented below, the overheads incurred by our proposed schemes are included.

Processors	32 out-of-order cores with private L1 data and instruction caches. instruction window size: 128, LSQ size: 64
NoC Architecture	4×8
Private L1 D&I–Caches	Direct mapped, 32KB, 64 bytes block size, 3 cycle access latency
Number of L2 Cache Banks	32 (distributed over the network)
L2 Cache	64 bytes block size, 10 cycle access latency
L2 Cache Bank Size	512KB
Number of Banks Per Memory Controller	16
Memory Configuration	DDR-800, Memory Bus Multiplier: 5, Bank Busy Time: 22 cycles, Rank Delay: 2 cycles, Read-Write Delay: 3 cycles, Memory CTL latency: 20 cycles, Refresh Period: 3120
Cache Coherency Protocol	MOESI_CMP_Directory
NoC parameters	5-stage router, flit size: 128 bits, buffer size = 5 flits, Number of virtual channels per port = 4, Routing algorithm: X-Y

Table 1: Baseline configuration.

Evaluation Metric: We use normalized weighted speedup metric to quantify the benefits of our proposed scheme. Weighted speedup is defined as: $WS = \sum \frac{IPC_i(shared)}{IPC_i(alone)}$, where $IPC_i(shared)$ is the IPC of application i when it is executed with the other applications in the workload and $IPC_i(alone)$ is the IPC of the same application when it is executed alone without having any contention with the other applications. In this section, we present the weighted speedup values that are *normalized* to that achieved by the default case where *no* prioritization is employed.

Workloads: We formed our workloads using the applications from the SPEC2006 benchmark suite [12]. Table 2 gives the 18 workloads that we used in our experiments on our 32-core system (the numbers in the parenthesis represent the number of copies for each application in the workload). The workloads listed in Table 2 cover all applications and are categorized into three different groups based on the memory intensity of the applications: (1) Workloads 1 through 6 (mixed workloads): in these workloads, half of the applications are memory intensive (applications with high MPKI) and the remaining ones are memory non-intensive. (2) Workloads 7 through 12: all applications in this category are memory intensive. (3) Workloads 13 through 18: none of the applications in this group is memory intensive. In other words, to form a workload, we first categorize the applications into three groups based on the memory intensity (since our scheme is proposed for memory accesses) and then, for each workload, we randomly pick 32 applications from the specific category (note that this strategy may choose some applications multiple times, but all the applications in a workload belong to the same memory-intensity region). MPKI values representing the memory intensities of the applications in the SPEC2006 suite can be found in [17].

4.2. Results

We now present the experimental results collected using various configurations. In executing a workload, the simulation was fast-forwarded for 1 billion cycles, and then we collected our statistics in the next 100-million cycle run. As stated earlier, we employed a one-to-one mapping between applications and cores, that is, each core executes one application and the application-to-core mapping does not change during the execution.

Results for a 32-core system with the baseline configuration: As mentioned earlier, in this configuration, 32 cores are connected by a 4×8 2D mesh-based NoC and 4 memory controllers are placed in four corners of the mesh (other parameters are as given in Table 1).

Figure 11 plots the *normalized* weighted speedup values achieved for our workloads, which are categorized into three groups as shown in Table 2. As discussed in Section 3, our approach to reducing end-to-end latency employs two complementary schemes. In Scheme-1, if the so-far delays of the response messages are larger than a threshold value (the default value of this threshold is $1.2 \times Delay_{avg}$) after the memory controller stage, they are given a higher priority in the network on the return paths to reach their destinations faster, whereas Scheme-2 accelerates the request messages that are destined for the idle memory banks. As stated earlier, in this scheme, the decision of whether a request message is destined for a idle bank or not is made based on the local information. When an L2 miss occurs, if over the last T cycles, the number of requests sent to the same bank is less than a threshold (th), the request is given high priority (the default values for T and th are 200 cycles and 1, respectively).

In Figure 11, two bars are presented for each workload. The first bar shows the performance improvement achieved when Scheme-1 is employed in the system and the second one is for the case where Scheme-1 and 2 are employed together. Note that the weighted speedup values presented in Figure 11 are normalized to the base case in which the applications are running together but no prioritization scheme is employed.

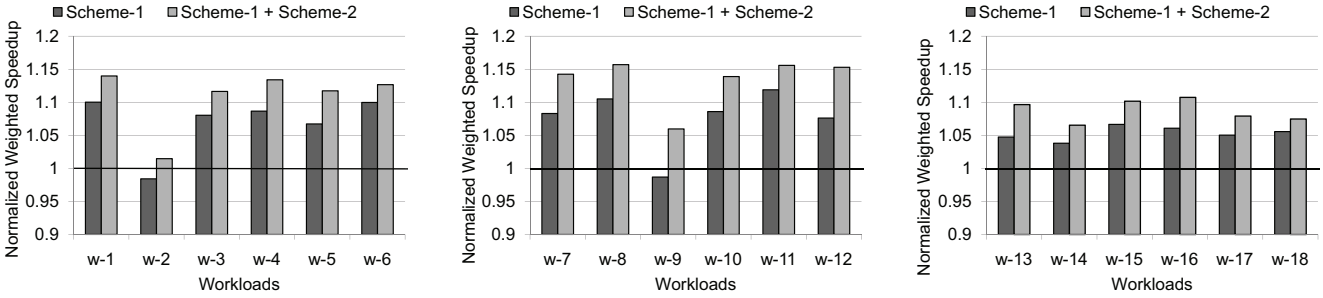
As can be observed from Figure 11, our approach (Scheme-1 + Scheme-2) improves performance by up to 13%, 15% and 10% for the mixed, memory intensive and memory non-intensive workloads, respectively. Further, in general, higher performance improvements are achieved when the applications are more memory intensive. This is because when the co-running applications are more memory intensive, the traffic on the network is higher and, as a result, the impact of our network prioritization based approach on accelerating the late messages becomes more pronounced. However, our proposed scheme does not improve the performance for all the workloads tested. For instance, the speedup values achieved by Scheme-1 for workloads 2 and 9 are slightly less than 1. The reason for this behavior is that giving higher priorities to some of the messages in the network hurts the other messages and, this can offset the benefits of our scheme in some cases.

The impact of employing Scheme-1 and Scheme-2 can be observed in the distribution of the end-to-end latencies and the average idleness of the memory banks, respectively. Figure 12a plots 8 "cumulative distribution functions" (CDFs) of the off-chip memory accesses generated by the first 8 applications in workload-1. In this figure, the values on the x-axis are the total latencies (in cycles) and the y-axis shows the fraction of the total number of off-chip memory accesses. Each curve corresponds to one of the applications and the $F(x)$ value gives the fraction of the total accesses with delays less than x. Figure 12b plots the *new* CDFs of the off-chip accesses for the same 8 applications when Scheme-1 is employed. It can be observed from Figure 12a that 90% of the messages have an average total latency about 700 cycles (shown with dashed lines), whereas Scheme-1 reduces it to about 600 cycles (see Figure 12b).

Figure 12c plots the "probability density function" (PDF) of the memory accesses issued by one of the applications (*lbm*) in workload-1 before and after Scheme-1 is employed (the area under the curve shows the fraction of total number of accesses). Employing Scheme-1 reduces the number of messages with large delays and move them from Region-1 to Region-2. This results in about 8%

MIXED	Workload-1	mcf(3), lbm(2), xalancbmk(1), milc(2), libquantum(1), leslie3d(5), GemsFDTD(1), soplex(1), omnetpp(2), perlbench(1), astar(1), wrf(1), tonto(1), sjeng(1), namd(1), hmmer(1), h264ref(1), gamess(1), calculix(1), bzip2(3), bwaves(1)
	Workload-2	mcf(4), lbm(2), xalancbmk(2), milc(3), libquantum(2), GemsFDTD(1), soplex(2), perlbench(2), astar(3), wrf(3), povray(1), namd(3), hmmer(1), h264ref(1), gcc(1), dealII(1)
	Workload-3	mcf(4), lbm(1), milc(2), libquantum(5), leslie3d(2), sphinx3(1), GemsFDTD(1), omnetpp(1), astar(2), zeusmp(2), wrf(2), tonto(1), sjeng(1), h264ref(1), gobmk(1), gcc(1), gamess(1), dealII(1), calculix(1), bwaves(1)
	Workload-4	mcf(1), lbm(2), xalancbmk(3), milc(2), leslie3d(1), sphinx3(3), GemsFDTD(1), soplex(3), omnetpp(1), astar(2), zeusmp(1), wrf(1), tonto(1), sjeng(1), h264ref(2), gcc(1), gamess(3), bzip2(2), bwaves(1)
	Workload-5	mcf(4), lbm(2), xalancbmk(3), milc(1), leslie3d(1), sphinx3(1), soplex(4), astar(2), zeusmp(2), wrf(1), sjeng(1), povray(2), namd(1), hmmer(1), h264ref(2), gromacs(1), gcc(1), calculix(1), bwaves(1)
	Workload-6	mcf(2), xalancbmk(2), milc(1), libquantum(1), leslie3d(2), sphinx3(3), GemsFDTD(3), soplex(2), omnetpp(1), perlbench(2), wrf(1), tonto(2), hmmer(1), gromacs(1), gobmk(1), gcc(1), gamess(1), dealII(2), bzip2(3)
MEM INTENSIVE	Workload-7	mcf(1), lbm(5), xalancbmk(5), milc(1), libquantum(5), leslie3d(4), sphinx3(3), GemsFDTD(6), soplex(2)
	Workload-8	mcf(3), lbm(2), xalancbmk(4), milc(3), libquantum(8), leslie3d(3), sphinx3(4), GemsFDTD(5)
	Workload-9	mcf(4), lbm(5), xalancbmk(4), milc(3), libquantum(4), leslie3d(2), sphinx3(6), GemsFDTD(2), soplex(2)
	Workload-10	mcf(4), lbm(3), xalancbmk(3), milc(2), libquantum(4), leslie3d(3), sphinx3(4), GemsFDTD(8), soplex(1)
	Workload-11	mcf(3), lbm(6), xalancbmk(2), milc(5), libquantum(1), leslie3d(2), sphinx3(4), GemsFDTD(4), soplex(5)
	Workload-12	mcf(2), lbm(3), xalancbmk(3), milc(6), libquantum(5), leslie3d(4), sphinx3(4), GemsFDTD(5)
MEM NON-INTENSIVE	Workload-13	perlbench(1), astar(3), zeusmp(2), wrf(2), sjeng(3), povray(2), hmmer(1), gromacs(2), gcc(1), gamess(2), dealII(2), calculix(5), bzip2(2), bwaves(4)
	Workload-14	omnetpp(3), perlbench(1), zeusmp(2), tonto(1), sjeng(1), povray(2), namd(2), hmmer(4), h264ref(3), gromacs(2), gobmk(3), gamess(3), bzip2(1), bwaves(4)
	Workload-15	omnetpp(2), perlbench(2), astar(1), zeusmp(3), sjeng(1), povray(1), namd(1), hmmer(2), h264ref(1), gromacs(2), gobmk(3), gcc(2), gamess(1), dealII(4), calculix(2), bzip2(2), bwaves(2)
	Workload-16	omnetpp(3), perlbench(3), astar(2), zeusmp(1), wrf(2), sjeng(3), povray(3), namd(1), hmmer(2), h264ref(1), gobmk(1), gcc(4), gamess(2), dealII(2), bzip2(1), bwaves(1)
	Workload-17	omnetpp(2), perlbench(2), astar(1), zeusmp(2), wrf(1), tonto(2), sjeng(1), povray(2), namd(1), hmmer(4), h264ref(1), gobmk(2), gcc(2), gamess(1), dealII(3), calculix(2), bzip2(3)
	Workload-18	omnetpp(2), perlbench(4), zeusmp(2), wrf(2), tonto(2), sjeng(2), namd(1), hmmer(2), h264ref(1), gromacs(2), gobmk(2), gcc(4), gamess(2), calculix(2), bzip2(1), bwaves(1)

Table 2: Workloads used in our 32-core experiments.



(a) Mixed workloads. (b) Memory intensive workloads. (c) Memory non-intensive workloads.

Figure 11: Speedup values achieved for different workloads executed on a 32-core system.

performance improvement for this application. As one can observe from this figure, not all the messages are transferred to Region-2 and there are still messages in Region-1. This is because Scheme-1 reduces a portion of the total memory access latency and, for instance, if a message spends a long time in the memory queue (e.g, 700 cycles), Scheme-1 will not be able to move this message to Region-2.

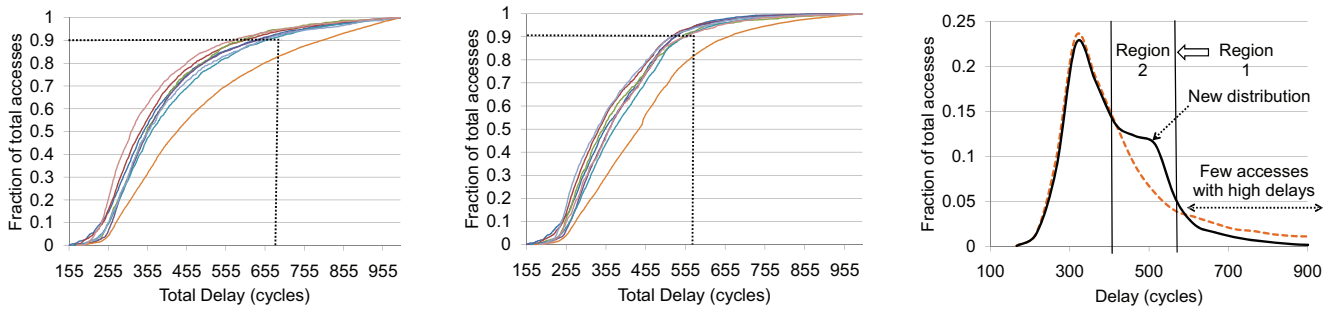
To illustrate the impact of Scheme-2, we plot in Figure 13 the idleness values for different banks of a memory controller when Scheme-2 is employed and when no scheme is used (workload-1). As can be seen, Scheme-2 reduces the idleness of our banks (resulting in an overall system performance improvement of more than 5%). Figure 14 plots dynamic reduction in bank idleness over the course of execution for one of our workloads (workload-1).

Results for a 16-core system: We also ran our experiments in a smaller system with 16 cores connected by a 4×4 mesh-based NoC. In this configuration, two memory controllers with the same parameters as in Table 1 are attached to two opposite corners of the mesh.

Figure 15 plots the speedup values achieved by our proposed scheme when our workloads are executed on this smaller system. We picked the first half of the applications in each workload shown in Table 2 for this experiment (for the mixed workload, the first half of the memory intensive and memory non-intensive applications are selected). Averaged over all the workloads, the speedups achieved in these experiments are about 8%, 10% and 5% for the mixed, memory intensive and memory non-intensive workloads, respectively.

A comparison between Figures 11 and 15 reveals that the speedup values for the 32-core system are generally higher than those for the 16-core system. This is because as the network size increases, the network delay contributes more to the round-trip latencies of the off-chip accesses and, as a result, the impact of the network prioritization is amplified.

Impact of the threshold value in Scheme-1: As discussed earlier, to determine whether a response message is late or not, after a request is serviced by the corresponding memory controller, the so-far delay of the request is compared against a threshold and if it is larger than this threshold value, the corresponding memory access is considered to be "late". Recall also that the default value for this threshold is $1.2 \times Avg_{delay}$. Note that Avg_{delay} is the average round-trip latency of the off-chip requests that belong to the same application (issued by the same core) and is computed dynamically by the source core. To study the impact of the threshold values on the achieved speedups, we performed experiments for two other threshold values: $1.1 \times Avg_{delay}$ and $1.4 \times Avg_{delay}$. Figure 16a plots the speedup values achieved for the three cases when workloads 1 through 6 are used. As can be observed, when the threshold is set to a larger value, the speedup values reduce since fewer messages are considered to be



(a) CDF (base). (b) CDF (Scheme-1). (c) PDFs.

Figure 12: (a) The CDFs of the off-chip accesses for the first 8 applications in workload-1. The values on the x-axis are the total (end-to-end) memory access latencies (in cycles), and the y-axis shows the fraction of the total number of off-chip memory accesses. (b) The CDFs of the off-chip accesses for the first 8 applications in workload-1, when Scheme-1 is employed. (c) The delay distributions of the memory accesses for one of the applications (lbm) in workload-1 before and after Scheme-1 is employed. This distribution change results in 8% performance improvement.

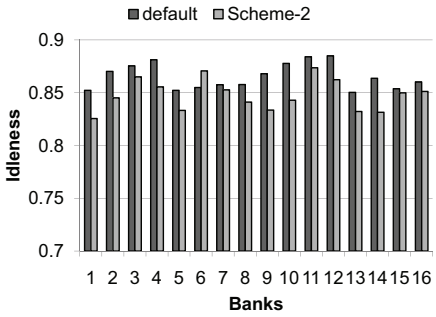


Figure 13: Idleness values of different banks of a memory controller when Scheme-2 is employed and when no scheme is used. As can be observed, Scheme-2 reduces idleness in most of the banks.

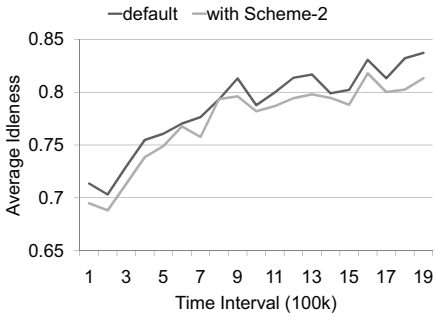


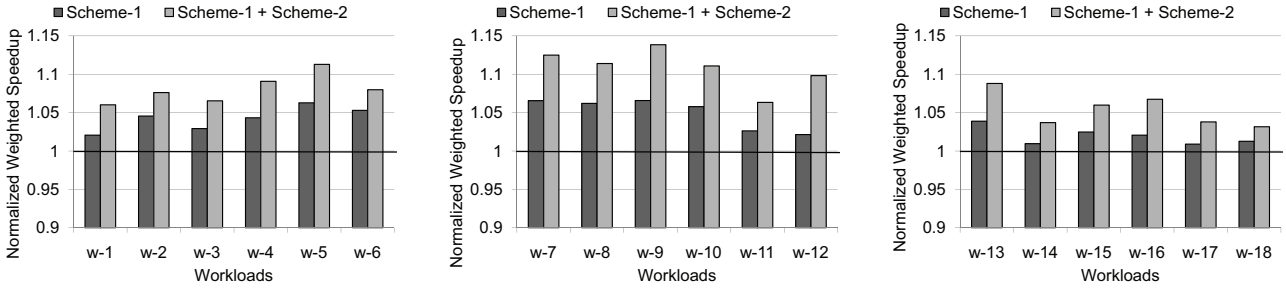
Figure 14: Average idleness of the memory banks (workload-1) over the execution.

late in this case, and prioritized in the network. However, reducing the threshold value may not help performance since when the number of late messages in the network increases, prioritizing too many messages in the network hurts other messages, leading eventually to network congestion and performance degradation.

Impact of the T value (history length): In Scheme-2, the decision of whether a request message is destined for a idle bank or not is made based on the local information. When an L2 miss occurs, if over the last T cycles, the number of requests sent to the same bank is less than a threshold (th), the request is given high priority (the default values for T and th are 200 cycles and 1, respectively). Figure 16b plots the speedup values for $T = 100$, 200, and 400. As can be observed, for $T = 400$ the speedup values are not as high as the case where T=200 since the number of late requests decreases. However, reducing the T length ($T = 100$) does not necessarily increase the speedups either (see workload-2 and workload-4) due to the imprecision in finding the idle banks and the slow down imposed on the other messages in the network.

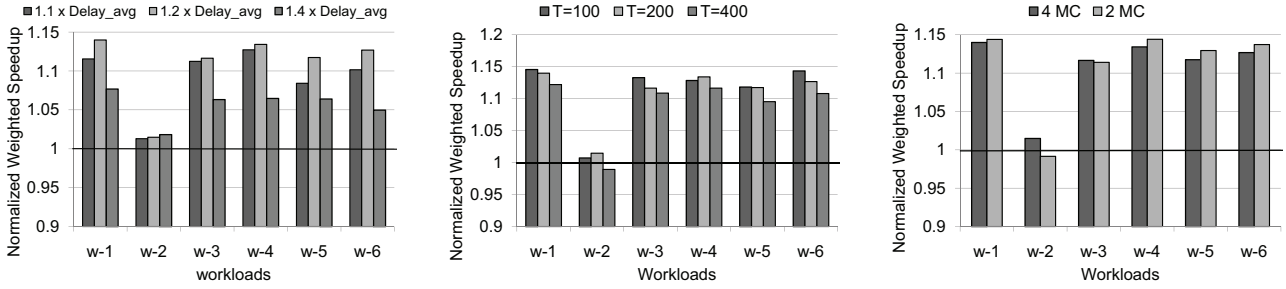
Impact of the number of memory controllers: Finally, Figure 16c plots the performance improvements for our mixed workloads when there are two and four memory controllers in the system. One can observe that, the performance improvement is slightly increased when there are fewer number of memory controllers in the system. The reason for this can be explained as follows. When there are fewer number of memory controllers in the system, due to the pressure increase on the bank queues, there will be more critical and late accesses in the system that can be enhanced by Scheme-1, and this results in higher performance. Note also that, although the benefits form Scheme-2 may reduce with fewer number of memory controllers (since there will be less idle banks), due to the improvement achieved by Scheme-1, the overall improvement (Scheme-1 + Scheme-2) is slightly better (see Figure 16c). However, the performance improvement is reduced for w-2 and w-3 due to the lower improvements achieved by Scheme-2.

Sensitivity to the structure of the routers: Although many schemes have been proposed by prior research to reduce message latency and contention in the on-chip networks, they do not completely eliminate the network contention and we are far from achieving the ideal performance. Therefore, our proposed schemes can be employed with the presence of the other proposed strategies to speed up the critical off-chip messages by prioritizing them in the network. One of the techniques to reduce the network latency is to employ routers with the fewer number of pipeline stages. As given in Table 1, in our default configuration, the routers are implemented as five-stage pipelines. However, the number of stages can be reduced to two (as discussed in Section 3.3, but here, all the flits have the opportunity to traverse each router in two cycles if there is no contention). Figure 17 compares the performance improvement achieved by our proposed schemes (for w-1 to w-6) when the routers have two and five pipeline stages. As can be observed from this plot, the performance is still improved up to 10% for the 2-stage pipeline. However, the improvement is 25-40% lower compared to the 5-stage pipeline case. This is because the percentage of the network latency that



(a) Mixed workloads. (b) Memory intensive workloads. (c) Memory non-intensive workloads.

Figure 15: Speedup values achieved for different workloads executed on a 16-core system.



(a) Impact of the threshold value in Scheme-1. (b) Impact of the history length in Scheme-2. (c) Impact of the number of memory controllers.

Figure 16: Results from our sensitivity experiments (32 cores).

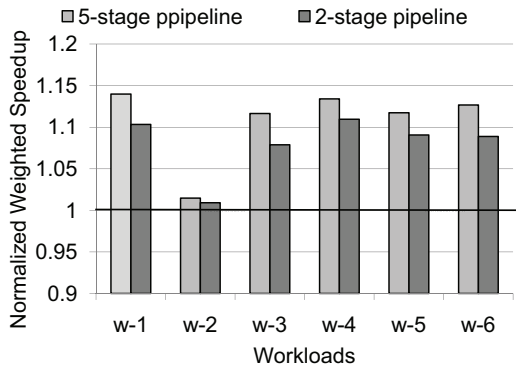


Figure 17: Sensitivity to the number of pipeline stages.

can be reduced (for the selected messages) by the network prioritization decreases for the 2-stage pipeline case (note that in this case no router bypassing is employed and the critical messages are prioritized in the message arbitration done in the routers). However, other techniques such as express channels proposed in [21] can be employed to increase the latency savings.

5. Related Work

Main memory accesses have been identified as a critical bottleneck in both multiprogrammed and multithreaded workloads [8, 18, 27]. Prior studies [2, 5, 22] have shown that packet latency variance can be reduced by using age-based prioritization at the network routers. These schemes added a new field to each packet to track its current network latency and assigned a higher priority in the router to packets that suffered higher network latencies during switch allocation. Lee et al. [23] proposed the use of a different metric with age-based prioritization, where the hop count of a packet is used to assign its network priority. However, livelocks can occur when such a static metric is used for prioritization, and hence, this approach required the implementation of complex probabilistic priority arbiters to prevent livelocks. In contrast to these age-based schemes, in our work, we compare the latency of a request coming from a core only against the average latency of the requests from the same core. Therefore, priorities for each core are calculated independently, which enables per-core optimization of the memory access latency variance.

Two recent approaches to network prioritization-based latency optimization are proposed by Das et al. [7, 8]. In [7], packets are prioritized based on the types of the co-running applications and in [8], the slack of a network packet is defined as the number of cycles the packet can be delayed without affecting the execution time of the application. Although these works also employ network prioritization, they do not have a detailed memory model (they assume fixed delay). In contrast, we have a detailed memory model (with queuing and row-buffer modeling) and use accurate so-far delay information dynamically updated during execution. Further, in [7, 8], latency/slack calculation is done at the core side based on indirect parameters such as hit/miss status of the packet and number of hops. However, it becomes extremely difficult to accurately predict latency/slack at the cores when multiple requests from different cores compete for the same MCs. Lastly, these prior works do not consider memory bank idleness. Consequently, our schemes are different from these prior works.

As mentioned earlier, in our work we use an S-NUCA [15] based cache space management. In [13], a novel scheme called Reactive NUCA is proposed for data placement in the distributed caches. In the proposed method, data is placed in the local L2 slices for the private data patterns seen in multi-programmed workloads. This scheme reduces the network latency of the memory accesses by elim-

inating paths #1 (L1 to L2) and #5 (L2 to L1) in Figure 2. However, it does not completely remove the network latency due to paths #2 (L2 to MC1) and #4 (MC1 to L2) in Figure 2, and our scheme will still provide performance benefits (at a lower level).

Another important component of off-chip memory access latency variance is the variations in memory queuing latency. In [29, 30], the authors targeted at reducing the variance of memory access latencies at the memory controller. These approaches coordinate multiple on-chip memory controllers to equalize the performance penalty suffered by each application due to interferences coming from other applications. The effectiveness of these schemes on a large multicore with high network latencies is unclear. On a large network, communicating information across memory controllers itself can easily have a very high latency, thus increasing the reaction time to dynamic changes in execution behavior.

Clearly, directly improving off-chip memory access latencies can also improve overall system performance. Memory queuing latencies can be improved by finding a better scheduling of memory requests in the memory bank queues [17, 18], designing lower latency routers [19, 26, 32], or employing better flow-control [20, 21]. We would like to note that the network prioritization schemes we propose in this work can be used with any memory scheduling algorithm, router micro architecture, or flow control scheme. For instance, while our Scheme-1 is orthogonal to the memory scheduling scheme employed (as it optimizes network latency component *after* the memory), the increase in the average number of entries in memory bank queues due to our Scheme-2 can enable these complex memory schedulers to find even better schedules as they can see a larger window of memory requests. We postpone the investigation of such interactions to a future study.

6. Conclusion

We proposed two network prioritization schemes that reduce the *end-to-end memory access latency* in multicores. Our first scheme addresses the latency variance in memory accesses that belong to the same application, and expedites memory response messages that experience high latencies by prioritizing them. This reduces the number of off-chip requests with high latencies and achieves a more uniform memory latency pattern. Our second scheme improves memory performance by prioritizing memory request messages that are destined for idle banks over other requests. This optimization increases bank level parallelism and improves memory utilization. Through an extensive evaluation of our schemes on a 4×8 mesh-based multicore, we show that our first scheme is very effective in reducing the number of off-chip requests that experience very high latencies and achieves 11% , 6%, and 10% performance improvement on memory intensive, memory non-intensive, and mixed multi-programmed workloads, respectively. When both our schemes used together, these improvements jump to 15%, 10% and 13%, in the same order.

References

[1] "Tilera tilepro36," http://www.tilera.com/products/processors/TILEPRO36.

[2] D. Abts and D. Weisser, "Age-based packet arbitration in large-radix k-ary n-cubes," in *SC*, 2007.

[3] V. Cuppu, B. Jacob, B. Davis, and T. Mudge, "A performance comparison of contemporary DRAM architectures," in *ISCA*, 1999.

[4] W. J. Dally, "Virtual-channel flow control," *IEEE Trans. Parallel Distrib. Syst.*, 1992.

[5] W. Dally and B. Towles, *Principles and Practices of Interconnection Networks*, 2003.

[6] W. J. Dally and B. Towles, "Route packets, not wires: on-chip inteconnection networks," in *DAC*, 2001.

[7] R. Das, O. Mutlu, T. Moscibroda, and C. R. Das, "Application-aware prioritization mechanisms for on-chip networks," in *MICRO*, 2009.

[8] R. Das, O. Mutlu, T. Moscibroda, and C. R. Das, "Argia: exploiting packet latency slack in on-chip networks," in *ISCA*, 2010.

[9] B. T. Davis, "Modern DRAM architectures," Ph.D. dissertation, 2001.

[10] B. Fields, S. Rubin, and R. Bodík, "Focusing processor policies via critical-path prediction," in *ISCA*, 2001.

[11] A. Glew, "MLP yes! ILP no! memory level parallelism, or, why I No Longer Worry About IPC." in *ASPLOS*, 1998.

[12] J. L. Henning, "SPEC CPU2006 benchmark descriptions," *SIGARCH Comput. Archit. News*, 2006.

[13] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki, "Reactive NUCA: near-optimal block placement and replication in distributed caches," in *ISCA*, 2009.

[14] Y. Hoskote, S. Vangal, A. Singh, N. Borkar, and S. Borkar, "A 5-ghz mesh interconnect for a teraflops processor," *IEEE Micro*, 2007.

[15] J. Huh, C. Kim, H. Shafi, L. Zhang, D. Burger, and S. W. Keckler, "A NUCA substrate for flexible CMP cache sharing," in *ICS*, 2005.

[16] C. Kim, D. Burger, and S. W. Keckler, "An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches," in *ASPLOS*, 2002.

[17] Y. Kim, D. Han, O. Mutlu, and M. Harchol-Balter, "ATLAS: A scalable and high-performance scheduling algorithm for multiple memory controllers," in *HPCA*, 2010.

[18] Y. Kim, M. Papamichael, O. Mutlu, and M. Harchol-Balter, "Thread cluster memory scheduling: Exploiting differences in memory access behavior," in *MICRO*, 2010.

[19] A. Kumar, P. Kundu, A. Singh, L.-S. Peh, and N. Jha, "A 4.6tbits/s 3.6ghz single-cycle noc router with a novel switch allocator in 65nm cmos," in *ICCD*, 2007.

[20] A. Kumar, L.-S. Peh, and N. K. Jha, "Token flow control," in *MICRO*, 2008.

[21] A. Kumar, L. shiuan Peh, P. Kundu, and N. K. Jha, "Express virtual channels: Towards the ideal interconnection fabric," in *ISCA*, 2007.

[22] J. W. Lee, M. C. Ng, and K. Asanovic, "Globally-synchronized frames for guaranteed quality-of-service in on-chip networks," in *ISCA*, 2008.

[23] M. M. Lee, J. Kim, D. Abts, M. Marty, and J. W. Lee, "Approximating age-based arbitration in on-chip networks," in *PACT*, 2010.

[24] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hållberg, J. Högberg, F. Larsson, A. Moestedt, and B. Werner, "Simics: A full system simulation platform," *Computer*, 2002.

[25] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood, "Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset," *SIGARCH Comput. Archit. News*, 2005.

[26] R. Mullins, A. West, and S. Moore, "Low-latency virtual-channel routers for on-chip networks," in *ISCA*, 2004.

[27] N. Muralimanohar and R. Balasubramonian, "Interconnect design considerations for large NUCA caches," in *ISCA*, 2007.

[28] O. Mutlu, H. Kim, and Y. N. Patt, "Efficient runahead execution: Power-efficient memory latency tolerance," *IEEE Micro*, 2006.

[29] O. Mutlu and T. Moscibroda, "Stall-time fair memory access scheduling for chip multiprocessors," in *MICRO*, 2007.

[30] O. Mutlu and T. Moscibroda, "Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared DRAM systems," in *ISCA*, 2008.

[31] D. Park, C. Nicopoulos, J. Kim, N. Vijaykrishnan, and C. R. Das, "Exploring fault-tolerant network-on-chip architectures," in *Proceedings of the International Conference on Dependable Systems and Networks*, 2006.

[32] L.-S. Peh and W. J. Dally, "A delay model and speculative architecture for pipelined routers," in *HPCA*, 2001.

[33] S. Rixner, "Memory controller optimizations for web servers," in *MICRO*, 2004.

[34] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens, "Memory access scheduling," in *ISCA*, 2000.

[35] M. B. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffman, P. Johnson, J.-W. Lee, W. Lee, A. Ma, A. Saraf, M. Seneski, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. Agarwal, "The raw microprocessor: A computational fabric for software circuits and general-purpose programs," *IEEE Micro*, 2002.

2012 IEEE/ACM 45th Annual International Symposium on Microarchitecture

MorphCore: An Energy-Efficient Microarchitecture for High Performance ILP and High Throughput TLP

Khubaib[†] M. Aater Suleman[‡] Milad Hashemi[†] Chris Wilkerson[§] Yale N. Patt[†]

[†]The University of Texas at Austin
{khubaib,miladh,patt}@hps.utexas.edu

[‡]Calxeda/HPS
suleman@hps.utexas.edu

[§]Intel Labs, Hillsboro, OR
chris.wilkerson@intel.com

Abstract

Several researchers have recognized in recent years that today's workloads require a microarchitecture that can handle single-threaded code at high performance, and multi-threaded code at high throughput, while consuming no more energy than is necessary. This paper proposes MorphCore, a unique approach to satisfying these competing requirements, by starting with a traditional high performance out-of-order core and making minimal changes that can transform it into a highly-threaded in-order SMT core when necessary. The result is a microarchitecture that outperforms an aggressive 4-way SMT out-of-order core, "medium" out-of-order cores, small in-order cores, and CoreFusion. Compared to a 2-way SMT out-of-order core, MorphCore increases performance by 10% and reduces energy-delay-squared product by 22%.

1. Introduction

Traditional core microarchitectures do not adapt to the thread level parallelism (TLP) available in programs. In general, industry builds two types of cores: large out-of-order cores (e.g., Intel's Sandybridge, IBM's Power 7), and small cores (e.g., Intel's MIC a.k.a Larrabee, Sun's Niagara, ARM's A15). Large out-of-order (OOO) cores provide high single thread performance by exploiting Instruction-Level Parallelism (ILP), but they are power-inefficient for multi-threaded programs because they unnecessarily waste energy on exploiting ILP instead of leveraging the available TLP. In contrast, small cores do not waste energy on wide superscalar OOO execution, but rather provide high parallel throughput at the cost of poor single thread performance.

Heterogeneous (or Asymmetric) Chip Multiprocessors (ACMPs) [11, 22, 28] have been proposed to handle this software diversity. ACMPs provide one or few large cores for speedy execution of single-threaded programs, and many small cores for high throughput in multi-threaded programs. Unfortunately, ACMPs require that the number of large and small cores be fixed at design time, which inhibits adaptability to varying degrees of software thread-level parallelism.

To overcome this limitation of ACMPs, researchers have proposed CoreFusion-like architectures [16, 5, 17, 25, 24, 32, 9, 10]. They propose a chip with small cores to provide high throughput performance in multi-threaded programs. These small cores can dynamically "fuse" into a large core when executing single-threaded programs. Unfortunately, the fused large core has low performance and high power/energy consumption compared to a traditional out-of-order core for two reasons: 1) there are additional latencies among the pipeline stages of the fused core, thus, increasing the latencies of the core's "critical loops", and 2) mode switching requires instruction cache flushes and incurs the cost of data migration among the data caches of small cores.

To overcome these limitations, we propose MorphCore, an adaptive core microarchitecture that takes the opposite approach of previously proposed reconfigurable cores. Rather than fusing small cores into a large core, MorphCore uses a large out-of-order core as the base substrate and adds the capability of in-order SMT to exploit highly parallel code. MorphCore provides two modes of execution: OutOfOrder and InOrder. In OutOfOrder mode, MorphCore provides the single-thread performance of a traditional out-of-order core with *minimal* performance degradation. However, when TLP is available, MorphCore "morphs" into a highly-threaded in-order SMT core. This allows MorphCore to hide long latency operations by executing operations from different threads concurrently. Consequently, it can achieve a higher throughput than the out-of-order core. Since no migration of instructions or data needs to happen on mode switches, MorphCore can switch between modes with minimal penalty.

MorphCore is built on two key insights. First, a highly-threaded (i.e., 6-8 way SMT) in-order core can achieve the same or better performance as an out-of-order core. Second, a highly-threaded in-order SMT core can be built using a subset of the hardware required to build an aggressive out-of-order core. For example, we use the Physical Register File (PRF) in the out-of-order core as the architectural register files for the many SMT threads in InOrder mode. Similarly, we use the Reservation Station entries as an in-order instruction buffer and the execution pipeline of the out-of-order core as-is.

MorphCore is more energy-efficient than a traditional out-of-order core when executing multi-threaded programs. It reduces execution time by exploiting TLP, and reduces energy consumption by turning off several power hungry structures (e.g., renaming logic, out-of-order scheduling, and the load queue) while in InOrder mode.

Our evaluation with 14 single-threaded and 14 multi-threaded workloads shows that MorphCore increases performance by 10% and reduces energy-delay-squared product by 22% over a typical 2-way SMT out-of-order core. We also compare MorphCore against three different core architectures optimized for different performance/energy design points, and against CoreFusion, a reconfigurable core architecture. We find that MorphCore performs best in terms of performance and energy-delay-squared product across a wide spectrum of single-threaded and multi-threaded workloads.

Contributions: This paper makes two contributions:

1. We present *MorphCore*, a new microarchitecture that combines out-of-order and highly-threaded in-order SMT execution within a single core. We comprehensively describe the microarchitecture needed to implement MorphCore, and the policy to switch between modes.
2. To the best of our knowledge, this is the first paper to quantitatively compare small, medium and large core architectures in

1072-4451/12 $26.00 © 2012 IEEE
DOI 10.1109/MICRO.2012.36

IEEE computer society

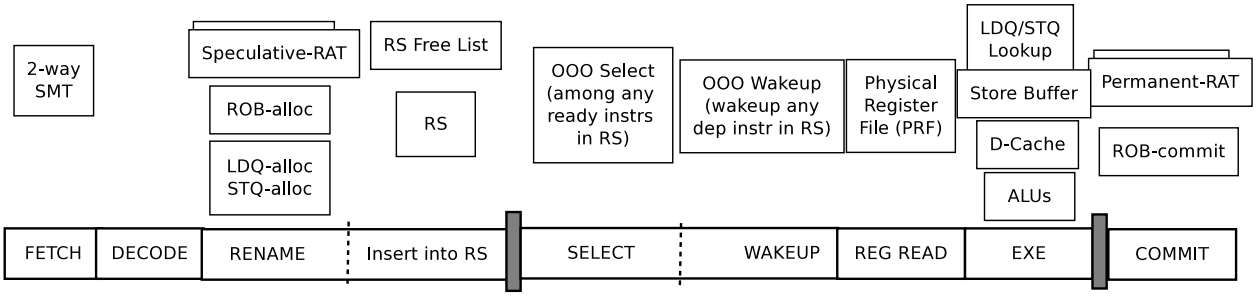


Figure 1: Out of Order core microarchitecture

terms of performance and energy-efficiency on single-threaded and multi-threaded workloads.

2. Background and Motivation

2.1. Out-of-order Execution

Out-of-order (OOO) cores provide better performance by executing instructions as soon as their operands become available, rather than executing them in program order. Figure 1 shows a high-level layout of a 2-way SMT OOO core pipeline. The top part shows major structures accessed and functionality performed in different stages of the pipeline. We describe a Pentium-4 like architecture [26], where the data, both speculative and architectural, is stored in the Physical Register File (PRF), and the per-thread Register Alias Table (RAT) entries point to PRF entries. The front-end Speculative-RAT points to the speculative state, and a back-end Permanent-RAT points to the architectural state. The front-end of the pipeline (from the Fetch stage until the Insert into Reservation Station (RS)) works in-order. Instructions are fetched, decoded, and then sent to the Rename Stage. The Rename stage renames (i.e. maps) the architectural source and destination register IDs into Physical Register File IDs by reading the Speculative-RAT of the thread for which instructions are being renamed, and inserts the instructions into the Reservation Station (also referred to as Issue Queue).

Instructions wait in the Reservation Station until they are selected for execution by the Select stage. The Select stage selects an instruction for execution once all of the source operands of the instruction are ready, and the instruction is the oldest among the ready instructions. When an instruction is selected for execution, it readies its dependent instructions via the Wakeup Logic block, reads its source operands from the PRF, and executes in a Functional Unit. After execution, an instruction's result is broadcast on the Bypass Network, so that any dependent instruction can use it immediately. The result is also written into the PRF, and the instruction updates its ROB status. The instruction retires once it reaches the head of the ROB, and updates the corresponding Permanent-RAT.

Problem With Wide Superscalar Out-of-order Execution. Unfortunately, the single-thread performance benefit of the large out-of-order (OOO) core comes with a power penalty. As we show in Section 6.2, a large OOO core consumes 92% more power than a medium OOO core, and 4.3x more than a small in-order core. This overhead exists to exploit instruction-level parallelism to increase core throughput, and is justified when the software has a single thread of execution. However, when multiple threads of execution exist, we propose that the core can be better utilized using *in-order* Simultaneous Multi-Threading (SMT).

2.2. Simultaneous Multi-Threading

Simultaneous Multi-Threading (SMT) [13, 35, 31] is a technique to improve the utilization of execution resources using multiple threads provided by the software. In SMT, a core executes instructions from multiple threads concurrently. Every cycle, the core picks a thread from potentially many ready threads, and fetches instructions from that thread. The instructions are then decoded and renamed in a regular pipelined fashion and inserted into a common (shared among all the threads) RS. Since instructions from multiple candidate threads are available in the RS, the possibility of finding ready instructions increases. Thus, SMT cores can achieve higher throughput provided that software exposes multiple threads to the hardware.

The Potential of In-Order SMT on a Wide Superscalar Core. The observation that a highly multi-threaded in-order core can achieve the instruction issue throughput similar to an OOO core was noted by Hily and Seznec [12]. We build on this insight to design a core that can achieve high-performance and low-energy consumption when software parallelism is available.

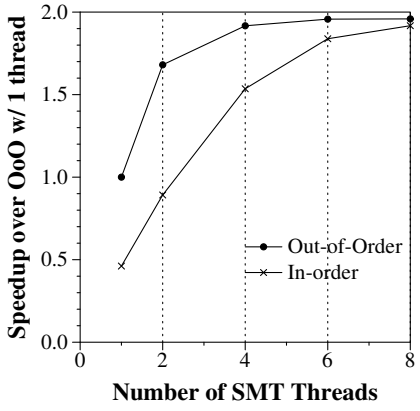


Figure 2: Performance of `black` with SMT

Figure 2 shows the performance of the workload `black` (Black-Scholes pricing [23]) on an out-of-order and an in-order core. For this experiment, both of the cores are similarly sized in terms of cache sizes, pipeline widths (4-wide superscalar) and depths (refer to Section 5 for experimental methodology). The performance of the in-order core is significantly less than the performance of the out-of-order core when both cores run only a single thread. As the

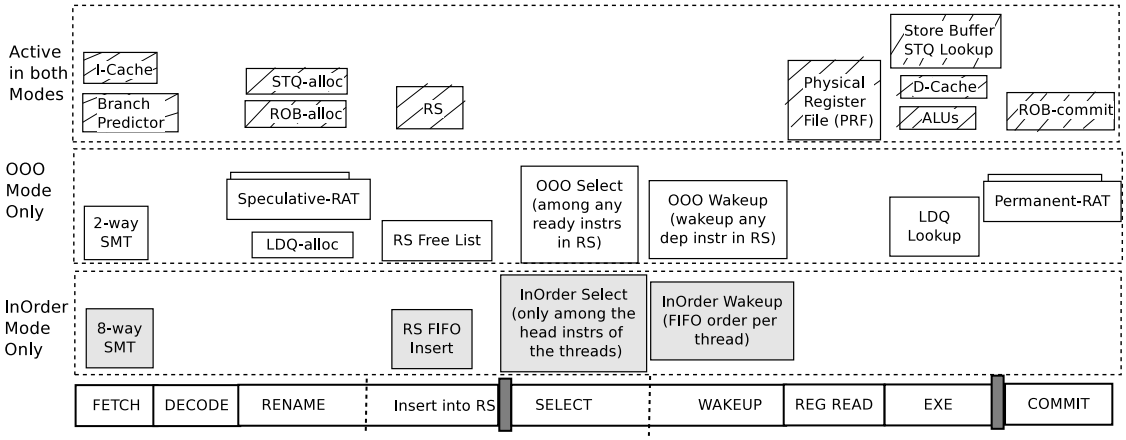


Figure 3: The MorphCore microarchitecture

number of SMT threads increases from 1 to 8, the performance of the out-of-order core increases significantly at 2 threads, but starts to saturate at 4 threads, because the performance is limited by the peak throughput of the core. In contrast, the performance of the in-order core continues to benefit from more threads (which allows it to better tolerate long latency operations and memory accesses). When the number of threads is equal to 8, the in-order core's performance begins to match the performance of the out-of-order core. This experiment shows that when high thread-level parallelism is available, high performance and low energy consumption can be achieved with *in-order SMT* execution, therefore the core need not be built with complex and power hungry structures needed for out-of-order SMT execution.

Summary. In spite of its high power cost, out-of-order execution is still desirable because it provides significant performance improvement over in-order execution. Thus, if we want high single-thread performance we need to keep support for out-of-order execution. However, when software parallelism is available, we can provide performance by using in-order SMT and not waste energy on out-of-order execution. To accomplish both, we propose the MorphCore architecture.

3. MorphCore Microarchitecture

3.1. Overview of the MorphCore Microarchitecture

The MorphCore microarchitecture is based on a traditional OOO core. Figure 3 shows the changes that are made to a baseline OOO core (shown in Figure 1) to build the MorphCore. It also shows the blocks that are active in both modes, and the blocks that are active only in one of the modes. In addition to out-of-order execution, MorphCore supports additional in-order SMT threads, and in-order scheduling, execution, and commit of simultaneously running threads. In OutofOrder mode, MorphCore works exactly like a traditional out-of-order core.

3.2. Fetch and Decode Stages.

The Fetch and Decode Stages of MorphCore work exactly like an SMT-enabled traditional OOO core. Figure 4 shows the Fetch Stage of the MorphCore. MorphCore adds 6 additional SMT contexts to the baseline core. Each context consists of a PC, a branch history register, and a Return Address Stack. In OutofOrder mode, only 2 of the SMT contexts are active. In InOrder mode, all 8 contexts are active. The branch predictor and the I-Cache are active in both modes.

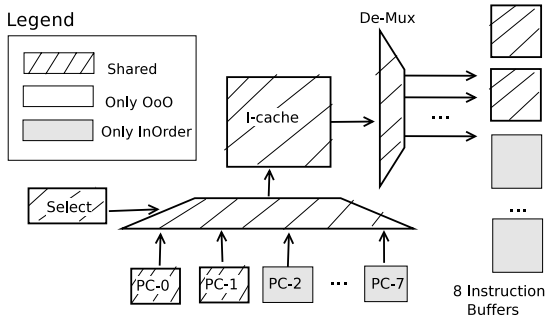


Figure 4: Microarchitecture of the Fetch stage

3.3. Rename Stage

Figure 5 shows the Rename Stage of the MorphCore. InOrder renaming is substantially simpler, and thus power-efficient, than OOO renaming. In InOrder mode, we use the Physical Register File (PRF) to store the architectural registers of the multiple in-order SMT threads: we logically divide the PRF into multiple fixed-size partitions where each partition stores the architectural register state of a thread (Figure 5(b)). Hence, the architectural register IDs can be mapped to the Physical Register IDs by simply concatenating the Thread ID with the architectural register ID. This approach limits the number of in-order SMT threads that the MorphCore can support to $num\_physical\_registers/num\_architectural\_registers$. However, the number of physical registers in today's cores is already large enough (and increasing) to support 8 in-order SMT threads which is sufficient to match the out-of-order core's performance. For the x86 ISA [15] that we model in our simulator, a FP-PRF partition of 24 entries and an INT-PRF partition of 16 entries per thread is enough to hold the architectural registers of a thread. The registers that are not renamed and are replicated 2-ways in the baseline OOO core need to be replicated 8-ways in MorphCore.

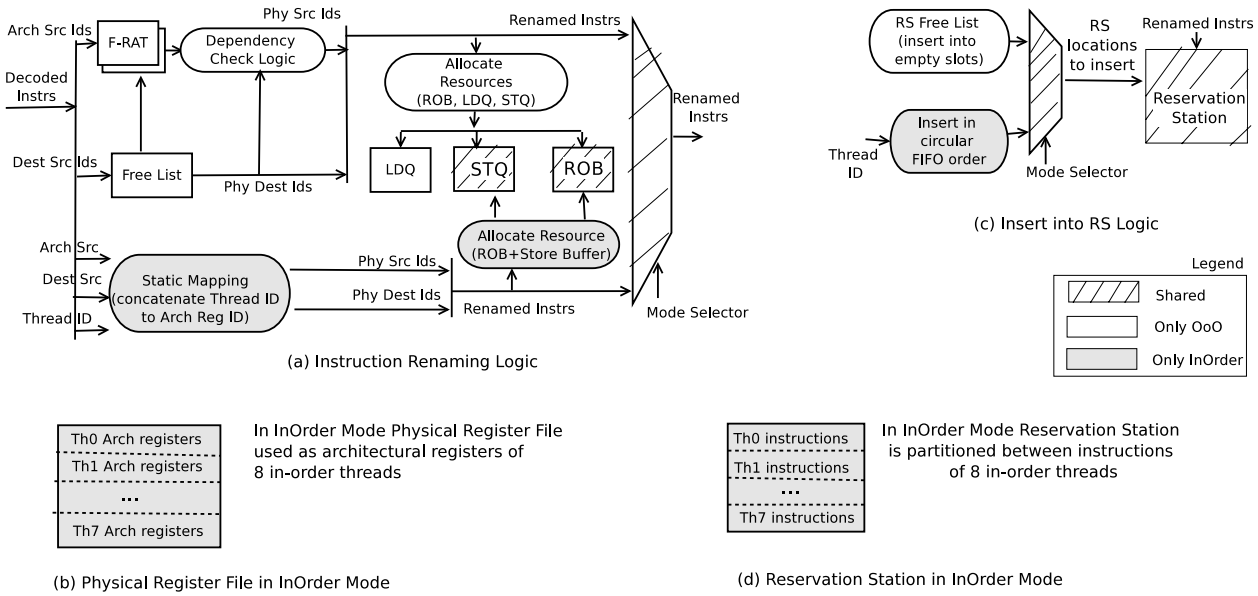


Figure 5: Microarchitecture of the Rename stage

Allocating/Updating the Resources. When the MorphCore is in OutofOrder mode, the instructions that are being renamed are allocated resources in the ROB and in the Load and Store Queues. In InOrder mode, MorphCore leverages the ROB to store the instruction information. We partition the ROB into multiple fixed-size chunks, one for each active thread. We do not allocate resources in the Load Queue in InOrder mode since memory instructions are not executed speculatively. Thus, the Load Queue is inactive. The Store Queue that holds the data from committed store instructions and the data that is not yet committed to the D-cache, is active in InOrder Mode as well, and is equally partitioned among the threads.

Insert into the Reservation Station (RS). Figure 5(c) shows the part of Rename stage that inserts renamed instructions into the RS. In OutofOrder mode, the RS is dynamically shared between multiple threads, and the RS entry that is allocated to an incoming renamed instruction is determined dynamically by consulting a Free List. In InOrder mode, the RS is divided among the multiple threads into fixed-size partitions (Figure 5(d)), and each partition operates as a circular FIFO. Instructions are inserted into consecutive RS entries pointed to by a per-thread RS-Insert-Ptr, and removed in-order after successful execution.

3.4. Select and Wakeup

MorphCore employs both OutofOrder and InOrder Wakeup and Select Logic. The Wakeup Logic makes instructions ready for execution, and the Select Logic selects the instructions to execute from the pool of ready instructions. Figure 6 shows these logic blocks.

OutofOrder Wakeup. OutofOrder Wakeup logic works exactly the same as a traditional out-of-order core. Figure 6 (unshaded) shows the structure of an RS entry [27]. An operand is marked ready (R-bit is set) when the corresponding MATCH bit has been set for the number of cycles specified in the DELAY field. When an instruction fires, it broadcasts its destination tag (power hungry), so that it can be compared against source tags of all instructions in the RS. If the destination tag matches the source tag of an operand, the MATCH bit is set and the DELAY field is set equal to the execution latency of the firing instruction (the latency of the instruction is stored in the RS entry allocated to the instruction). The DELAY field is also latched in the SHIFT field associated with the source tag. The SHIFT field is right shifted one-bit every cycle the MATCH bit is set. The R bit is set when the SHIFT field becomes zero. The RS-entry waits until both sources are ready, and then raises the Req OOO Exec line.

OutofOrder Select. The OutofOrder Select logic monitors *all* instructions in the RS (power hungry), and selects the oldest instruction(s) that have the Req OOO Exec lines set. The output of the Select Logic is a Grant bit vector, in which every bit corresponds to an RS entry indicating which instructions will fire next. When an instruction is fired, the SCHEDULED bit is set in the RS entry so that the RS entry stops requesting execution in subsequent cycles.

InOrder Wakeup. The InOrder mode executes/schedules instructions in-order, i.e., an instruction becomes ready after the previous instruction has either started execution or is ready and independent. We add 2 new bit-fields to each RS entry for in-order scheduling (Scheduled, and MATCH (M)). The new fields are shaded in Figure 6. The InOrder Wakeup Logic block also maintains the M/DELAY/SHIFT/R bit fields per architectural register, in order to track the availability of architectural registers. When an instruction fires, it sets the R, M, and DELAY bit fields corresponding to the destination register in the InOrder Wakeup Logic block as follows: resets the R bit, sets the MATCH (M) bit, and sets the DELAY field to the execution latency of the firing instruction (the DELAY/SHIFT mechanism works as explained above). Each cycle, for every thread, the InOrder Wakeup checks the availability of source registers of the two oldest instructions (R bit is set). If the sources are available, the Wakeup logic readies the instructions by setting the M bit in the RS entry to 1. The InOrder Wakeup is power-efficient since it avoids the broadcast and matching of the destination tag against the source

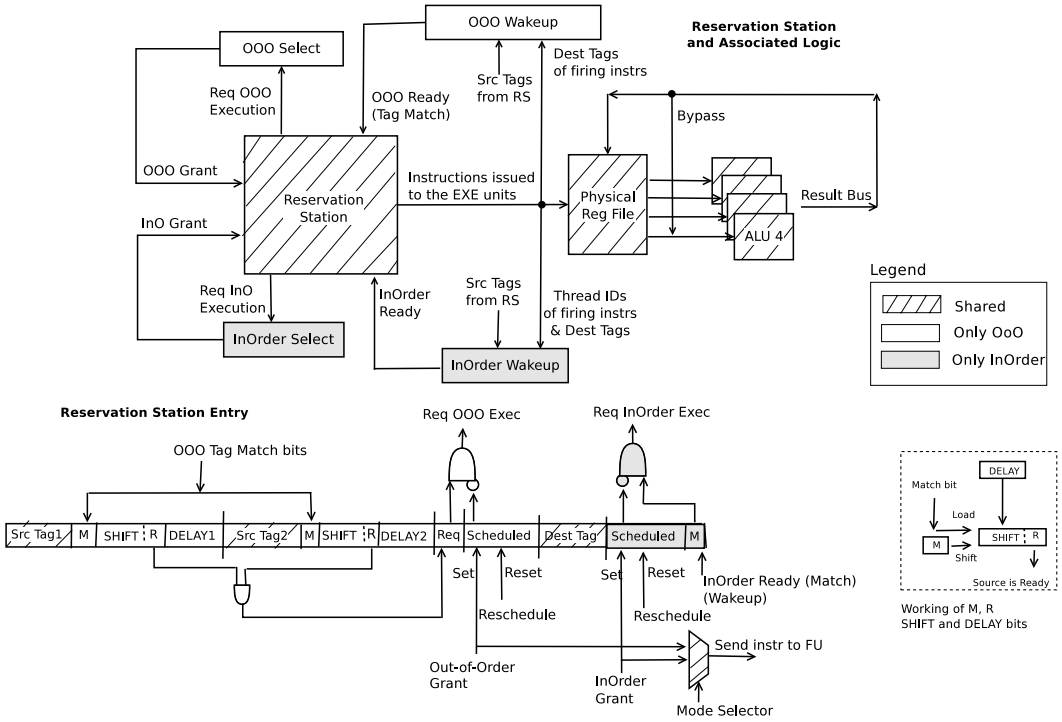


Figure 6: MorphCore Wakeup and Selection Logic

operands of all instructions in the RS.

InOrder Select. The InOrder Select Logic block works hierarchically in a complexity-effective (power-efficient) manner by maintaining eight InOrder select blocks (one per thread) and another block to select between the outcomes of these blocks. Furthermore, each in-order select logic only monitors the two oldest instructions in the thread's RS partition, rather than monitoring the entire RS as in OOO select. Note that only two instructions need monitoring in InOrder mode because instructions from each thread are inserted and scheduled/removed in a FIFO manner.

3.5. Execution and Commit

When an instruction is selected for execution, it reads its source operands from the PRF, executes in an ALU, and broadcasts its result on the bypass network as done in a traditional OOO core. In MorphCore, an additional PRF-bypass and data storage is active in InOrder mode. This bypass and buffering is provided in order to delay the write of younger instruction(s) in the PRF if an older longer latency instruction is in the execution pipeline. In such a case, younger instruction(s) write into a temporary small data buffer (4-entry per thread). The buffer adds an extra bypass in PRF-read stage. Instructions commit in traditional SMT fashion. For OutofOrder commit, the Permanent-RAT is updated as well. In InOrder mode, only the thread's ROB Head pointer needs to be updated.

3.6. Load/Store Unit

Figure 7 shows the Load/Store Unit. In OutofOrder mode, load/store instructions are executed speculatively and out of order (similar to a traditional OOO core). When a load fires, it updates its entry in the Load Queue and searches the Store Queue to get the latest data. When a store fires, it updates and stores data in the Store Queue, and searches the Load Queue to detect store-to-load program order violations. In InOrder mode, since load/store instructions are not executed speculatively, no Load Queue CAM searches are done. However, loads still search the Store Queue that holds committed data. Store instructions also update the Store Queue.

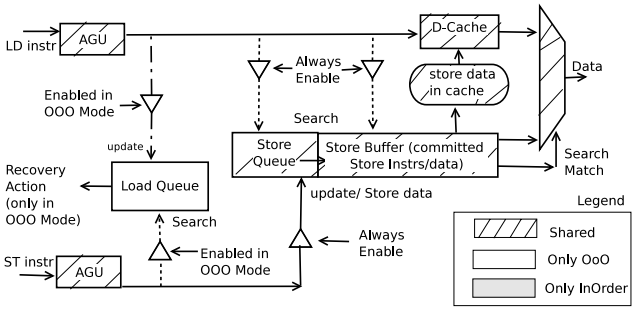


Figure 7: Load / Store unit

3.7. Recovering from Branch Mispredictions

In OutofOrder mode, a branch misprediction triggers a recovery mechanism that recovers the F-RAT to the state prior to the renaming of the mispredicted branch instruction. In InOrder mode, a branch misprediction squashes the instructions in the RS partition, the ROB partition and the front-end pipeline from the thread, followed by redirection of the PC to the correct target.

4. MorphCore Discussion

4.1. Area and Power Overhead of MorphCore

First, MorphCore increases the number of SMT ways from 2 to 8. This adds hardware to the Fetch stage and other parts of the core, which is less than 0.5% area overhead as reported by our modified McPAT [20] tool. Note that it does not incur the two biggest overheads of adding SMT contexts in an OOO core –additional Rename tables and physical registers– because the SMT contexts being added are in-order. Second, MorphCore adds InOrder Wakeup and Select logic, which we assume adds an area overhead of less than 0.5% of core area, half the area of the OOO Wakeup and Select logic blocks. Third, adding extra bypass/buffering adds an area overhead of 0.5% of core. Thus, MorphCore adds an area overhead of 1.5%, and a power overhead of 1.5% in InOrder mode.

4.2. Timing/Frequency Impact of MorphCore

MorphCore requires only two key changes to the baseline OOO core:

1) InOrder renaming/scheduling/execution logic. MorphCore adds a multiplexer in the critical path of three stages: a) in the Rename stage to select between OutofOrder mode and InOrder mode renamed instructions, b) in the Instruction Scheduling stage to select between the OutofOrder mode and InOrder mode ready instructions, and c) in PRF-read stage because of additional bypassing in InOrder mode. In order to estimate the frequency impact of this overhead, we assume that a multiplexer introduces a delay of one transmission gate, which we assume to be half of an FO4 gate delay. Assuming 20 FO4 gate delays per pipeline stage [33, 7], we estimate that MorphCore runs 2.5% slower than the baseline OOO core.

2) More SMT contexts. Addition of in-order SMT contexts can lengthen the thread selection logic in MorphCore's front-end. This overhead is changing the multiplexer that selects one out of many ready threads from 2-to-1 to 8-to-1. We assume that running MorphCore 2.5% slower than the baseline OOO core hides this delay.

In addition to the above mentioned timing-critical changes to the baseline OOO core, MorphCore adds InOrder Wakeup and Select logic blocks. Because InOrder instruction scheduling is simpler than OutofOrder instruction scheduling, we assume that newly added blocks can be placed and routed such that they do not affect the critical path of other components of the baseline OOO core. Thus, we conclude that the frequency impact of MorphCore is only 2.5%.

4.3. Turning Off Structures in InOrder Mode

The structures that are inactive in InOrder Mode (OOO renaming logic, OOO scheduling, and load queue) are unit-level clock-gated. Thus, no dynamic energy is consumed, but static energy is still consumed. Unit-level power-gating could be applied to further cut-down static energy as well, but we chose not to do so, since according to our McPAT estimates, static energy consumed by these structures is very small, whereas the overhead incurred by unit-level power-gating is significant.

4.4. Interaction with OS

MorphCore does not require any changes to the operating system, and acts like a core with the number of hardware threads equal to the maximum number of threads supported in the InOrder Mode (8 in our implementation). Switching between the two modes is handled in hardware.

4.5. When does MorphCore Switch Modes?

In our current implementation of MorphCore, we switch between modes based on the number of active threads (other policies are part of our future work). A thread is active when it is not waiting on any synchronization event. The MorphCore starts running in OutofOrder mode when the number of active threads is less than a threshold (2 in our initial implementation). If the OS schedules more threads on MorphCore, and the number of active threads becomes greater than the threshold, the core switches to InOrder mode. While running in InOrder mode, the number of active threads can drop for two reasons: the OS can de-schedule some threads or the threads can become inactive waiting for synchronization. We assume that the threading library uses MONITOR/MWAIT [15] instructions such that MorphCore hardware can detect a thread becoming inactive, e.g., inactive at a barrier waiting for other threads to reach the barrier, or inactive at a lock-acquire waiting for another thread to release the lock. If the number of active threads becomes smaller than or equal to the threshold, the core switches back to OutofOrder mode until more threads are scheduled or become active (the hardware makes the thread active when a write to the cacheline being monitored is detected).

4.6. Changing Mode from OutofOrder to InOrder

Mode switching is handled by a micro-code routine that performs the following tasks:

1) Drains the core pipeline.

2) Spills the architectural registers of all threads. We spill these registers to a reserved memory region. To avoid cache misses on these writes, we use Full Cache Line Write instructions that do not read the cache line before the write [15].

3) Turns off the Renaming unit, OutofOrder Wakeup and Select Logic blocks, and Load Queue. Note that these units do not necessarily need to be power-gated (we assume that these units are clock-gated).

4) Fills the register values back into each thread's PRF partitions. This is done using special load micro-ops that directly address the PRF entries without going through renaming.

4.7. Changing Mode from InOrder to OutofOrder

Since MorphCore supports more threads in InOrder Mode than in OutofOrder Mode, when switched into OutofOrder mode, MorphCore cannot run all the threads simultaneously and out-of-order. Thus some of the threads need to be marked inactive or "not running" (unless they are already inactive, which is the case in our current implementation). The state of the inactive threads is stored in memory until they become active. To load the state of the active threads, the MorphCore stores pointers to the architectural state of the inactive threads in a structure called the Active Threads Table. The Active Threads Table is indexed using the Thread ID, and stores an 8-byte pointer for each thread. Mode switching is handled by a micro-code routine that performs the following tasks:

1) Drains the core pipeline.

2) Spills the architectural registers of all threads. Store pointers to the architectural state of the inactive threads in the Active Thread Table.

3) Turns on the Renaming unit, OutofOrder Wakeup and Select Logic blocks, and Load Queue.

4) Fills the architectural registers of only the active threads into pre-determined locations in PRF, and updates the Speculative-RAT and Permanent-RAT.

Table 1: Configuration of the simulated machine

Core Configurations	
OOO-2	**Core:** 3.4GHz, 4-wide issue OOO, 2-way SMT, 14-stage pipeline, 64-entry unified Reservation Station (Issue Queue), 192 ROB, 50 LDQ, 40 STQ, 192 INT/FP Physical Reg File, 1-cycle wakeup/select **Functional Units:** 2 ALU/AGUs, 2 ALU/MULs, 2 FP units. ALU latencies (cycles): int arith 1, int mul 4-pipelined, fp arith 4-pipelined, fp divide 8, loads/stores 1+2-cycle D-cache **L1 Caches:** 32KB I-cache, D-cache 32KB, 2 ports, 8-way, 2-cycle pipelined **SMT:** Stages select round-robin among ready threads. ROB, RS, and instr buffers shared as in Pentium 4 [18]
OOO-4	3.23GHz (5% slower than OOO-2), 4-wide issue OOO, 4-way SMT, Other parameters are same as OOO-2.
MED	**Core:** 3.4GHz, 2-wide issue OOO, 1 Thread, 10-stage, 48-entry ROB/PRF. **Functional Units:** Half of OOO-2. Latencies same as OOO-2. **L1 Caches:** 1 port Dcache, other same as OOO-2. **SMT:** N/A
SMALL	**Core:** 3.4GHz, 2-wide issue In-Order, 2-way SMT, 8-stage pipeline. **Functional Units:** Same as MED. **L1 Caches:** Same as MED. **SMT:** Round-Robin Fetch
MorphCore	**Core:** 3.315GHz (2.5% slower than OOO-2), Other parameters are same as OOO-2. **Functional Units and L1 Caches:** Same as OOO-2. **SMT and Mode switching:** 2-way SMT similar to OOO-2, 8-way in-order SMT (Round-Robin Fetch) in InOrder mode. RS and PRF partitioned in equal sizes among the in-order threads. InOrder mode when active threads > 2, otherwise, OutofOrder mode
Memory System Configuration	
Caches	**L2 Cache:** private L2 256KB, 8-way, 5 cycles. **L3 Cache:** 2MB write-back, 64B lines, 16-way, 10-cycle access
Memory	8 banks/channel, 2 channels, DDR3 1333MHz, bank conflicts, queuing delays modeled. 16KB row-buffe, 15 ns row-buffer hit latency

Table 2: Characteristics of Evaluated Architectures

Core	Type	Freq (Ghz)	Issue-width	Num of cores	SMT threads per core	Total Threads	Total Norm. Area	Peak ST throughput	Peak MT throughput
OOO-2	out-of-order	3.4	4	1	2	2	1	4 ops/cycle	4 ops/cycle
OOO-4	out-of-order	3.23	4	1	4	4	1.05	4 ops/cycle	4 ops/cycle
MED	out-of-order	3.4	2	3	1	3	1.18	2 ops/cycle	6 ops/cycle
SMALL	in-order	3.4	2	3	2	6	0.97	2 ops/cycle	6 ops/cycle
MorphCore	out-of-order or in-order	3.315	4	1	OutOfOrder: 2, or InOrder: 8	2 or 8	1.015	4 ops/cycle	4 ops/cycle

4.8. Overheads of Changing the Mode

The overhead of changing the mode is pipeline drain, which varies with the workload, and the spill or fill of architectural register state of the threads. The x86 ISA [15] specifies an architectural state of $\sim$780 bytes per thread (including the latest AVX extensions). The micro-code routine takes $\sim$30 cycles to spill or fill the architectural register state of each thread after the pipeline drain (a total of $\sim$6KB and $\sim$250 cycles for 8 threads) into reserved ways of the private L2 cache (assuming a 256 bit wide read/write port to the cache, and a cache bandwidth of 1 read/write per cycle). We have empirically observed no loss in performance by taking away $\sim$6KB from the private 256KB cache. Note that the overhead of changing the mode can be reduced significantly by overlapping the spilling or filling of the architectural state with the pipeline drain. It is our future work to explore such mechanisms.

5. Experimental Methodology

Table 1 shows the configurations of the cores and the memory subsystem simulated using our in-house cycle-level x86 simulator. The simulator faithfully models microarchitectural details of the core, cache hierarchy and memory subsystem, e.g., contention for shared resources, DRAM bank conflicts, banked caches, etc. To estimate the area and power/energy of different core architectures, we use a modified version of McPAT [20]. We modified McPAT to: 1) report finer-grain area and power data, 2) increase SMT ways without increasing the Rename (RAT) tables, 3) use the area/energy impact of InOrder scheduling (1/2 of OOO), 4) model extra bypass/buffering, and 5) model the impact of SMT more accurately. Note that all core configurations have the same memory subsystem (L2, L3 and main memory).

Table 2 summarizes the key characteristics of the compared architectures. We run the baseline OOO-2 core at 3.4GHz and scale the frequencies of the other cores to incorporate the effects of both increase in area and critical-path-delay. For example, OOO-4's frequency is 5% lower than OOO-2 because adding the 2 extra SMT threads significantly increases the area/complexity of the core: it adds two extra Rename tables (RATs), at least a multiplexer at the end of Rename stage, and also adds extra buffering at the start of Rename stage (to select between 4, rather than 2 rename tables) which we estimate (using McPAT) to be an additional 5% area and thus lower frequency by 5%. MorphCore's frequency is reduced by 2.5% because its critical path increased by 2.5% (as explained in Section 4.2). Since the OOO-2 core has the highest frequency and supports 4-wide superscalar OOO execution, we can expect it to have the highest single thread (ST) performance. Since the SMALL and MED cores have the highest aggregate ops/cycle, we can expect them to have the highest multi-threaded (MT) performance. We expect the MorphCore to perform close to best in both ST and MT workloads. In Section 7.1, we also compare MorphCore against CoreFusion [16], a representative of reconfigurable core architectures proposed to date.

5.1. Workloads

Table 3 shows the description and input-set for each application. We simulate 14 single-threaded SPEC 2006 applications and 14 multi-threaded applications from different domains. We limit the number of single-thread workloads to 14 to ensure that the number of single-thread and multi-thread workloads is equal, so that the single-thread results do not dominate the overall average performance data. We randomly choose the 14 SPEC workloads. Each SPEC benchmark

Table 3: Details of the simulated workloads

Workload	Problem description	Input set
Multi-Threaded Workloads		
web	web cache [29]	500K queries
qsort	Quicksort [8]	20K elements
tsp	Traveling salesman [19]	11 cities
OLTP-1	MySQL server [2]	OLTP-simple [3]
OLTP-2	MySQL server [2]	OLTP-complex [3]
OLTP-3	MySQL server [2]	OLTP-nontrx [3]
black	Black-Scholes [23]	1M options
barnes	SPLASH-2 [34]	2K particles
fft	SPLASH-2 [34]	16K points
lu (contig)	SPLASH-2 [34]	512x512 matrix
ocean (contig)	SPLASH-2 [34]	130x130 grid
radix	SPLASH-2 [34]	300000 keys
ray	SPLASH-2 [34]	teapot.env
water (spatial)	SPLASH-2 [34]	512 molecules
Single-Threaded Workloads		
SPEC 2006	7 INT and 7 FP benchmarks	200M instrs

is run for 200M instructions with ref input set, where the representative slice is chosen using a Simpoint-like methodology. We do so since SPEC workloads are substantially longer (billions of instructions), and easier to sample using existing techniques like SimPoint. Single-threaded workloads run on a single core with other cores turned off. In contrast, multi-threaded workloads run with the number of threads set equal to the number of available contexts, i.e., $number of cores \times number of SMT contexts$. We run all multi-threaded workloads to completion and count only useful instructions, excluding synchronization instructions. Statistics are collected only in the parallel region, and initialization phases are ignored. For reference, Figure 8 shows the percentage of execution time in multi-threaded workloads when a certain number of threads are active. A thread is active when it is not waiting on any synchronization event. We will refer to this data when presenting our results next.

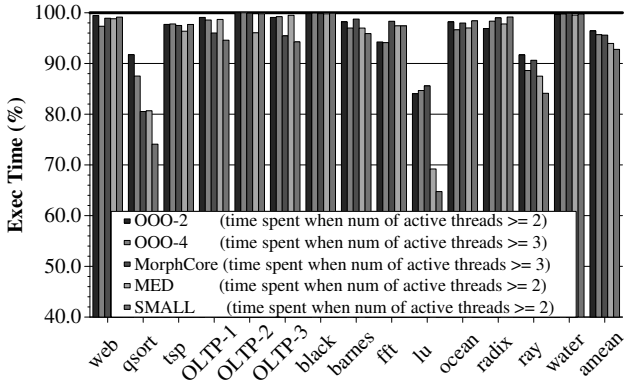


Figure 8: Percentage of execution time when a certain number of threads are active

6. Results

Since MorphCore attempts to improve performance and reduce energy, we compare our evaluated architectures on performance, energy-consumption, and a combined performance-energy metric, the energy-delay-squared product.

Table 4: Micro-op throughput (uops/cycle) on OOO-2

web	qsort	tsp	OLTP-1	OLTP-2	OLTP-3	black
3.47	1.95	2.78	2.29	2.23	2.35	3.39
barnes	fft	lu	ocean	radix	ray	water
3.31	2.68	3.11	2.17	1.94	3.06	3.51

6.1. Performance Results

Since design and performance trade-offs of ST and MT workloads are inherently different, we evaluate their performance separately. We will present a combined average across all ST and MT workloads in Section 6.1.3.

6.1.1. Single-Thread (ST) Performance Results. Figure 9a shows the speedup of each core normalized to OOO-2. As expected, OOO-2 achieves the highest performance on all workloads. The MorphCore is a close second. This is because MorphCore introduces minimal changes to a traditional out-of-order core. As a result of these changes, MorphCore runs at a 2.5% slower frequency than OOO-2, achieving 98.8% of the performance of OOO-2. The OOO-4 core provides slightly lower performance than MorphCore. This is because OOO-4 has a higher overhead when running in ST mode, a 5% frequency penalty, as it supports 4 OOO SMT threads. Note that the difference in performance among OOO-2, OOO-4, and MorphCore is the smallest for memory-bound workloads, e.g., `mcf`, `GemsFDTD`, and `lbm`. On the other hand, the cores optimized for multi-thread performance, MED and SMALL, have issue widths of 2 (as opposed to 4 for ST optimized cores) and either run in-order (SMALL) or out-of-order with a small window (MED). This results in significant performance loss in ST workloads: MED loses performance by 25% and SMALL by 59% as compared to OOO-2. The performance loss is more pronounced for FP workloads (right half of figure) as compared to INT workloads. In summary, MorphCore provides the second best performance (98.8% of OOO-2) on ST workloads.

6.1.2. Multi-Thread (MT) Performance Results. Multi-thread (MT) performance is affected by not only the performance potential of a single core, but the total number of cores and SMT threads on the cores. Figure 9b shows the speedup of each core normalized to OOO-2. As expected, the throughput optimized cores, MED and SMALL, provide the best MT performance (on average 30% and 33% performance improvement over OOO-2 respectively). This is because MED and SMALL cores have higher total peak throughput even though they take approximately the same area as OOO-2 (see Table 2).

More importantly, MorphCore provides a significant 22% performance improvement over OOO-2. MorphCore provides the highest performance improvement for workloads that have low micro-op execution throughput (uops/cycle) when run on the baseline OOO-2 core (Table 4). This is because MorphCore provides better latency tolerance and increases core throughput by executing up to 8 threads simultaneously. For example, `radix` gets the highest performance improvement of 84% over OOO-2 by increasing the uops/cycle from 1.94 to 3.58. In fact, MorphCore outperforms MED cores by 15% on `radix` because of its ability to run more SMT threads as compared

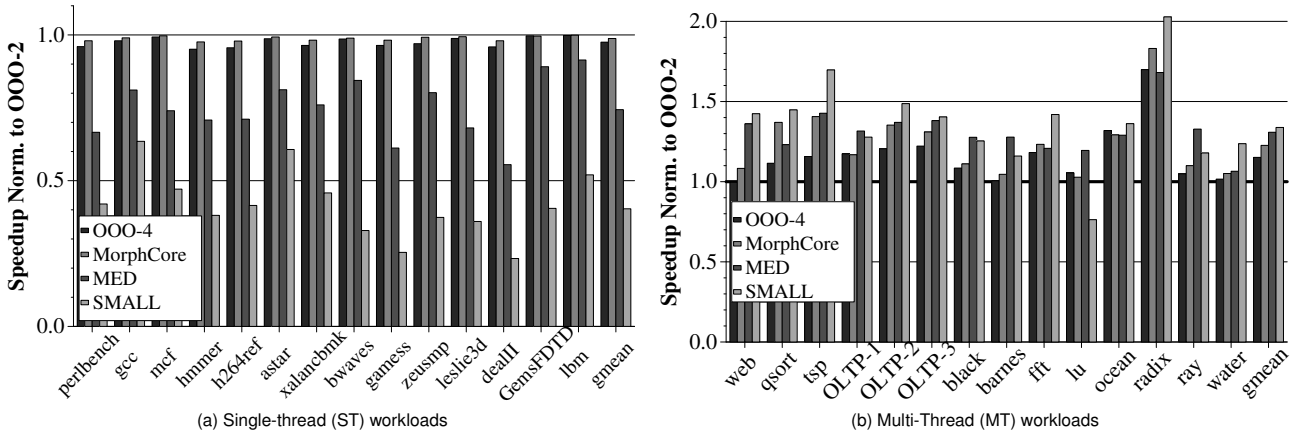


Figure 9: Performance results

to three MED cores. `qsort` is another workload with low uops/cycle (1.95), however MorphCore (similar to other throughput cores) does not provide as high a performance improvement as in the case of `radix`. This is because of two reasons: 1) when executing `qsort`, MorphCore does not spend a significant amount of time in InOrder mode (only 80% of execution time runs more than 2 threads active as shown in Figure 8), and 2) even when more than 2 threads are active, only 50% of the time are 6 or more threads active (data not shown in Figure 8). Thus, MorphCore does not get much opportunity to achieve higher throughput. Note that MorphCore still outperforms MED cores in `qsort` because of its ability to execute up to 8 threads.

Other workloads that have relatively high uops/cycle on OOO-2 (from 2.17 to 2.78) achieve relatively lower performance improvement with MorphCore over OOO-2 (from 23% for `fft` to 40% for `tsp`). The performance improvement of MorphCore is higher for `tsp` as compared to other workloads in this category even with a relatively high baseline uops/cycle of 2.78 (on OOO-2) because MorphCore ends up executing fewer number of total instructions (-10%) as compared to OOO-2 although doing the same algorithmic work. This is because `tsp` is a branch and bound algorithm, and the likelihood of quickly reaching the solution increases with more threads.

MorphCore provides the least performance improvement in workloads that can achieve a high uops/cycle (from 3.06 to 3.51) even when run with 2 threads on OOO-2 (`web`, `black`, `barnes`, `ray`, and `water`). These workloads have high per-thread ILP available, and thus do not benefit significantly from increasing the number of threads, because the performance improvement that can be achieved is limited by the peak throughput of MorphCore. However, as we later show, MorphCore is still beneficial because it is able to provide higher performance *at a lower energy consumption by executing SMT threads in-order*.

In general, MorphCore's performance improvement is lower than that of throughput optimized cores, MED and SMALL, over OOO-2 (on average 22% vs 30% and 33%) because of its lower peak MT throughput (Table 2). However, MorphCore outperforms MED cores in 3 workloads: `qsort`, `fft`, and `radix`. `qsort` and `radix` benefit from more threads as explained above. In `fft`, MED cores suffer from thread imbalance during the execution: 3 threads are active only for 72% of the execution time, and only 2 threads are active for 24% of execution time, and thus provide a slightly lower performance (-3%) than MorphCore. MorphCore also outperforms SMALL cores in `lu`. (In fact SMALL cores perform worse than OOO-2). This is because `lu`'s threads do not reach global barrier at the same time and have to wait for the lagging thread. Because SMALL cores have low single-thread performance, threads end up waiting for the lagging thread for a significant amount of time (only 1 thread is active for 35% of the execution time as shown in Figure 8), and thus execution time increases significantly. MorphCore does not suffer significantly from the problem of thread-imbalance-at-barrier because it switches into OutOfOrder mode when only 1 thread is active, therefore thread's waiting time is reduced.

MorphCore also outperforms OOO-4, a core architecture that has a higher area overhead and is significantly more complex than MorphCore (because OOO-4 supports 4 OOO SMT contexts), on average by 7% and up to 26% (for `qsort`). Although the peak throughput of both MorphCore and OOO-4 is the same (4, Table 2), MorphCore wins because it provides better latency tolerance by executing more threads than OOO-4. Thus, for workloads which have low uops/cycle and benefit from increasing the number of threads, MorphCore provides significantly higher MT performance compared to OOO-4.

6.1.3. Overall Performance Results. Figure 10a summarizes the average speedup of each architecture normalized to OOO-2. On single-thread (ST) workloads, MorphCore performs very close to OOO-2, the best ST-optimized architecture. On multi-thread (MT) workloads, MorphCore performs 22% higher than OOO-2, and achieves 2/3 of the performance potential of the best MT-optimized architectures (MED and SMALL). On average across all workloads (ST and MT), MorphCore outperforms all other architectures. We conclude that MorphCore is able to handle diverse ST and MT workloads efficiently.

6.1.4. Sensitivity of MorphCore's Results to Frequency Penalty. We find that for a MorphCore with an X% frequency penalty, performance of ST and MT workloads reduces by $\sim$ X/2% and X% respectively as compared to a MorphCore with no frequency penalty. This is because our ST workloads are core+memory bound while our MT workloads are primarily core bound.

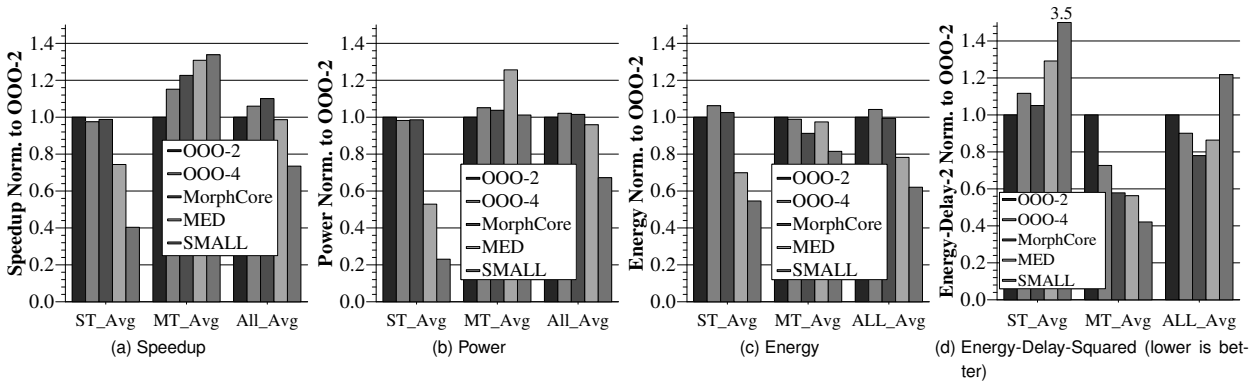


Figure 10: Speedup, Power, Energy, and Energy-Delay-Squared Results Summary

6.2. Energy-Efficiency Results

We first refer to Figure 10b that shows total power (static + dynamic) of each core configuration for ST and MT workloads. As expected, for ST workloads, the highest power configurations are large out-of-order cores (OOO-2, OOO-4, and MorphCore). A single MED or a single SMALL core takes area less than an out-of-order core, and thus toggle less capacitance, resulting in 47% and 77% lower power respectively. For MT workloads, all core configurations consume similar power (except MED cores), thus confirming that area-equivalent core comparisons result in power-equivalent core comparisons. The 3 MED cores take 18% more area (Table 2), and provide 30% higher performance (which translates into more dynamic power), resulting in 25% more power over OOO-2. Note that MorphCore consumes 2% less power than OOO-4 while providing 7% higher performance. This is because MorphCore does not waste energy on OOO renaming/scheduling, and instead, provides performance via highly-threaded in-order SMT execution.

Figure 10c shows the total (static + dynamic) energy consumed by each configuration (core includes L1 I and D caches but not L2 and L3 caches) normalized to OOO-2. As expected, SMALL cores are the most energy-efficient cores in both workload categories: for ST workloads, they have 59% lower performance for 77% lower power (an energy reduction of 46%), and for MT workloads they have 33% higher performance for 1% higher power (an energy reduction of 19%) than OOO-2. For MT workloads, MorphCore is the second best in energy-efficiency (after SMALL cores): MorphCore consumes 9%, 7%, and 6% less energy than OOO-2, OOO-4, and MED cores respectively.

MorphCore reduces energy consumption for two reasons: 1) MorphCore reduces execution time, thus keeping the core's structures active for shorter period of time, and 2) even when MorphCore is active, some of the structures that will be active in traditional out-of-order cores will be inactive in MorphCore's InOrder mode. These structures include the Rename logic, part of the instruction Scheduler, and the Load Queue. For reference, Table 5 shows the power for several key structures of OOO-2 core as a percentage of core power averaged across MT workloads. We find that 50% of the energy savings of MorphCore over OOO-2, and 75% of the energy savings of MorphCore over OOO-4 come from *reducing the activity of these structures*. MorphCore is also more energy-efficient than MED cores because even when it provides 8% lower performance, it does so at significantly (22%) lower power than MED cores, resulting an energy savings of 6% (70% of MorphCore's energy savings over MED cores happen because of reduced activity of MorphCore's structures in InOrder mode).

Table 5: Power of key structures of OOO-2

Structure	Power
Rename + RATs	4.9%
Scheduler	2.9%
Physical Register File	3.7%
Load + Store Queue	3.0%
ROB	2.1%

Figure 10d shows Energy-Delay-Squared (ED^2), a combined energy-performance efficiency metric [36, 21], of the five evaluated architectures. On average across all workloads, MorphCore provides the lowest ED^2: 22% lower than the baseline OOO-2, 13% lower than OOO-4, 9% lower than MED, and 44% lower than SMALL. We conclude that MorphCore provides a good balance between energy consumption and performance improvement in both ST and MT workloads.

7. Related Work

MorphCore is related to previous work in reconfigurable cores, heterogeneous chip multiprocessors, scalable cores, simultaneous multithreading, and power-efficient cores.

7.1. Reconfigurable Cores

Most closely related to our work are the numerous proposals that use reconfigurable cores to handle both latency- and throughput sensitive workloads [16, 5, 17, 25, 24, 32, 9, 10]. All these proposals share the *same fundamental idea*: build a chip with "simpler cores" and "combine" them using additional logic at runtime to form a high performance out-of-order core when high single thread performance is required. The cores operate independently in throughput mode.

TFlex [17], E2 dynamic multicore architecture [25], Bahurupi [24], and Core Genesis [10] require compiler analysis and/or ISA support for instruction steering to constituent cores to reduce the number of accesses to centralized structures. MorphCore does not require compiler/ISA support, and therefore can run legacy binaries without modification. Core Fusion [16], Federation Cores [5], Widget [32], and Forwardflow [9] provide scalability without any compiler/ISA support, similar to MorphCore.

Shortcomings. There are several shortcomings with the approach of combining simpler cores to form a large OOO core:
(1) Performance benefit of fusing the cores is limited because the constituent small cores operate in lock-step. Furthermore, fusing adds latencies among the pipeline stages of the fused core, and requires inter-core communication if dependent operations are steered to different cores.
(2) Switching modes incurs high overhead due to instruction cache flushes and data migration among the data caches of small cores.
(3) Core-Fusion-like proposals are not only in-efficient in "fused" mode, but also in their "non-fused" mode, because they use medium-size OOO cores as their base cores, which are power inefficient.

Comparison with CoreFusion. CoreFusion [16] fuses medium-sized OOO cores (2-wide, 48 entry OOO window) to form a large out-of-order core. Figure 11 shows the speedup of a single medium-sized OOO core (MED) and MorphCore normalized to CoreFusion for single-threaded workloads. In this experiment, CoreFusion combines three MED cores (see Table 2), and we use fusion latencies as described in [16] (7-cycle rename, 2-cycle extra branch misprediction, and 2-cycle inter-core communication penalties). We use the instruction steering heuristic described in the CoreFusion paper, and assume perfect LSQ bank prediction for steering loads/stores to cores. CoreFusion outperforms MED across all workloads except `mcf` (12% on average) because of its higher effective issue-width, window-size and L1 Dcache size. In `mcf`, these benefits are nullified by the overhead of inter-core communication introduced by CoreFusion. MorphCore outperforms both MED and CoreFusion across the board because unlike CoreFusion, it is a traditional aggressive out-of-order core without latency and communication overheads. On average MorphCore performs 17% better than CoreFusion.

Figure 12 shows the average speedup, power, energy, and ED^2 of MorphCore normalized to CoreFusion. On average, MorphCore provides 5% higher performance than CoreFusion. CoreFusion outperforms MorphCore in multi-threaded workloads (8% on average, per benchmark results are shown in Figure 9b) because it has a higher peak throughput as it consists of 3 medium-sized OOO cores. MorphCore reduces power (19%), energy (29%), and ED^2 (29%) when averaged across both single-threaded and multi-threaded workloads over CoreFusion because it uses less area (see Table 2), and thus, consumes less static power than CoreFusion's three MED cores while providing higher or comparable performance.

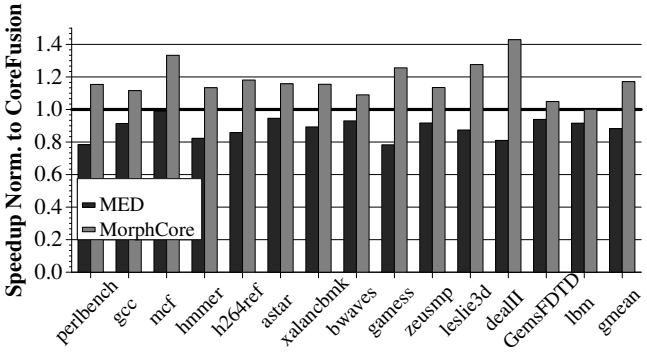


Figure 11: MorphCore's Single-Thread Performance versus CoreFusion

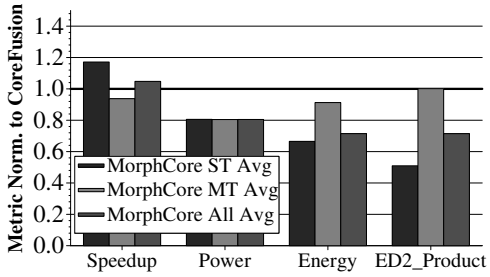


Figure 12: Average Speedup, Power, Energy, and ED^2 of MorphCore versus CoreFusion

7.2. Heterogeneous Chip-Multiprocessors

Heterogeneous (or Asymmetric) Chip Multiprocessors (ACMPs) [11, 22, 28] consist of one or a few large cores to accelerate single-threaded code, and many small cores to accelerate multi-threaded code. They have two limitations. First, the number of large and small cores is fixed at design time. In contrast, MorphCore can adapt the number of cores optimized for serial and parallel execution dynamically. Second, they incur a migration cost when execution is migrated between a small and a large core. Since MorphCore can accelerate threads "in-place," no migration overhead is incurred.

7.3. Scalable Cores

Scalable cores scale their performance and power consumption over a wide operating range. Dynamic Voltage and Frequency Scaling (DVFS) [14, 6] is a widely-used technique to scale a core's performance and power (e.g., Intel Turbo Boost [1]). However, increasing performance using DVFS costs significant increase in power consumption (power increases with the cube of frequency). Albonesi et al. [4] proposed dynamically tuning processor resources (e.g., cache size, register file size, issue queue entries etc.) in order to provide on-demand performance and energy savings. However, such techniques do not explore how these resources can be better used, and what other resources can be turned-off when TLP is available.

7.4. Simultaneous Multi-Threading

Simultaneous Multi-Threading (SMT) [13, 35, 31] was proposed to improve resource utilization by executing multiple threads on the same core. However, unlike MorphCore, previously proposed SMT techniques *do not reduce power consumption* when TLP is available. Furthermore, traditional SMT increases the area/complexity and power consumption of the core, whereas MorphCore leverages existing structures and does not increase area/complexity and power. Hily and Seznec observed in [12] that out-of-order execution becomes unnecessary when thread-level parallelism is available. In contrast, MorphCore saves energy and improves performance when executing multi-threaded workloads.

7.5. Power-Efficient Cores

Braids [30] provides OOO-like single-thread performance using in-order resources. However, Braids does not adapt to the software because it *requires* complicated compiler/software effort upfront. In contrast, MorphCore requires no software effort, and adapts to the software's needs.

8. Conclusion

We propose the MorphCore architecture which is designed from the ground-up to improve the performance and energy-efficiency of both single-threaded and multi-threaded programs. MorphCore operates as a high-performance out-of-order core when Thread-Level Parallelism is low, and as a high-performance low-energy, highly-threaded in-order SMT core when Thread-Level Parallelism is high. Our evaluation with 14 single-threaded and 14 multi-threaded workloads shows that MorphCore increases performance by 10% and reduces energy-delay-squared product (ED^2) by 22% over a typical 2-way SMT out-of-order core. We also show that MorphCore increases performance and reduces ED^2 when compared to an aggressive 4-way SMT out-of-order core, medium out-of-order cores, and small in-order cores. It also outperforms CoreFusion, a reconfigurable core architecture, in terms of performance (by 5%), and ED^2 (by 29%). We therefore suggest MorphCore as a promising direction for increasing performance, saving energy, and accommodating workload diversity while requiring minimal changes to a traditional out-of-order core. In the future we plan to further enhance MorphCore by exploring better policies for switching between in-order and out-of-order mode and by providing hardware mechanisms to support a low-power in-order single-thread mode.

Acknowledgments

We thank José Joao, other members of the HPS research group, Carlos Villavieja, our shepherd Scott Mahlke, and the anonymous reviewers for their comments and suggestions. Special thanks to Nikhil Patil, Doug Carmean, Rob Chappell, and Onur Mutlu for helpful technical discussions. We gratefully acknowledge the support of the Cockrell Foundation and Intel Corporation. Khubaib was supported by an Intel PhD Fellowship.

References

[1] "Intel Turbo Boost Technology," Intel Corporation, http://www.intel.com/technology/turboboost/index.htm.
[2] "MySQL database engine 5.0.1," http://www.mysql.com.
[3] "SysBench: a system performance benchmark v0.4.8," http://sysbench.sourceforge.net.
[4] D. H. Albonesi *et al.*, "Dynamically tuning processor resources with adaptive processing," *IEEE Computer*, 2003.
[5] M. Boyer, D. Tarjan, and K. Skadron, "Federation: Boosting per-thread performance of throughput-oriented manycore architectures," *ACM Trans. Archit. Code Optim. (TACO)*, 2010.
[6] T. Burd and R. Brodersen, "Energy efficient CMOS microprocessor design," in *Proceedings of the Twenty-Eighth Hawaii International Conference on System Sciences*, 1995.
[7] Z. Chishti and T. N. Vijaykumar, "Optimal power/performance pipeline depth for SMT in scaled technologies," *IEEE Trans. on Computers*, Jan. 2008.
[8] A. J. Dorta *et al.*, "The OpenMP source code repository," in *Euromicro*, 2005.
[9] D. Gibson and D. A. Wood, "Forwardflow: a scalable core for power-constrained CMPs," in *ISCA*, 2010.
[10] S. Gupta, S. Feng, A. Ansari, and S. Mahlke, "Erasing core boundaries for robust and configurable performance," in *MICRO*, 2010.
[11] M. D. Hill and M. R. Marty, "Amdahl's law in Multicore Era," Univ. of Wisconsin, Tech. Rep. CS-TR-2007-1593, 2007.
[12] S. Hily and A. Seznec, "Out-of-order execution may not be cost-effective on processors featuring simultaneous multithreading," in *HPCA*, 1999.
[13] H. Hirata *et al.*, "An elementary processor architecture with simultaneous instruction issuing from multiple threads," in *ISCA*, 1992.
[14] M. Horowitz *et al.*, "Low-power digital design," in *IEEE Symposium on Low Power Electronics*, 1994.
[15] Intel, "Intel 64 and IA-32 Architectures Software Dev. Manual, Vol-1," 2011.
[16] E. Ipek *et al.*, "Core fusion: accommodating software diversity in chip multiprocessors," in *ISCA-34*, 2007.
[17] C. Kim *et al.*, "Composable lightweight processors," in *MICRO-40*, 2007.
[18] D. Koufaty and D. Marr, "Hyperthreading technology in the Netburst microarchitecture," *IEEE Micro*, 2003.
[19] H. Kredel, "Source code for traveling salesman problem (tsp)," http://krum.rz.uni-mannheim.de/ba-pp-2007/java/index.html.
[20] S. Li *et al.*, "McPAT: an integrated power, area, and timing modeling framework for multicore and manycore architectures," in *MICRO 42*, 2009.
[21] A. J. Martin, *et al.*, *Power aware computing*. Kluwer Academic Publishers, 2002, ch. ET2: a metric for time and energy efficiency of computation.
[22] T. Y. Morad *et al.*, "Performance, power efficiency and scalability of asymmetric cluster chip multiprocessors," 2006.
[23] NVIDIA Corporation, "CUDA SDK code samples," 2009.
[24] M. Pricopi and T. Mitra, "Bahurupi: A polymorphic heterogeneous multi-core architecture," *ACM TACO*, January 2012.
[25] A. Putnam *et al.*, "Dynamic vectorization in the E2 dynamic multicore architecture," *SIGARCH Comp. Arch. News*, 2011.
[26] D. Sager, D. P. Group, and I. Corp, "The microarchitecture of the pentium 4 processor," *Intel Technology Journal*, vol. 1, p. 2001, 2001.
[27] J. Stark *et al.*, "On pipelining dynamic instruction scheduling logic," in *MICRO-33*, 2000.
[28] M. A. Suleman *et al.*, "Accelerating critical section execution with asymmetric multi-core architectures," in *ASPLOS*, 2009.
[29] Tornado Web Server, "Source code," http://tornado.sourceforge.net/, 2008.
[30] F. Tseng and Y. N. Patt, "Achieving out-of-order performance with almost in-order complexity," in *ISCA*, 2008.
[31] D. M. Tullsen *et al.*, "Simultaneous multithreading: Maximizing on-chip parallelism," in *ISCA-22*, 1995.
[32] Y. Watanabe *et al.*, "Widget: Wisconsin decoupled grid execution tiles," in *ISCA*, 2010.
[33] C. Wilkerson *et al.*, "Trading off cache capacity for reliability to enable low voltage operation," in *ISCA*, 2008.
[34] S. C. Woo *et al.*, "The SPLASH-2 programs: Characterization and methodological considerations," in *ISCA-22*, 1995.
[35] W. Yamamoto *et al.*, "Performance estimation of multistreamed, superscalar processors," in *Hawaii Intl. Conf. on System Sciences*, 1994.
[36] V. Zyuban *et al.*, "Integrated analysis of power and performance for pipelined microprocessors," *IEEE Transactions on Computers*, 2004.

2012 IEEE/ACM 45th Annual International Symposium on Microarchitecture

Composite Cores: Pushing Heterogeneity into a Core

Andrew Lukefahr, Shruti Padmanabha, Reetuparna Das, Faissal M. Sleiman, Ronald Dreslinski,
Thomas F. Wenisch, and Scott Mahlke

Advanced Computer Architecture Laboratory
University of Michigan
Ann Arbor, MI
{lukefahr, shrupad, reetudas, sleimanf, rdreslin, twenisch, mahlke}@umich.edu

Abstract

Heterogeneous multicore systems—comprised of multiple cores with varying capabilities, performance, and energy characteristics—have emerged as a promising approach to increasing energy efficiency. Such systems reduce energy consumption by identifying phase changes in an application and migrating execution to the most efficient core that meets its current performance requirements. However, due to the overhead of switching between cores, migration opportunities are limited to coarse-grained phases (hundreds of millions of instructions), reducing the potential to exploit energy efficient cores.

We propose Composite Cores*, an architecture that reduces switching overheads by bringing the notion of heterogeneity* ***within*** *a single core. The proposed architecture pairs big and little compute* μEngines *that together can achieve high performance and energy efficiency. By sharing much of the architectural state between the* μEngines*, the switching overhead can be reduced to near zero, enabling fine-grained switching and increasing the opportunities to utilize the little* μEngine *without sacrificing performance. An intelligent controller switches between the* μEngines *to maximize energy efficiency while constraining performance loss to a configurable bound. We evaluate* Composite Cores *using cycle accurate microarchitectural simulations and a detailed power model. Results show that, on average, the controller is able to map 25% of the execution to the little* μEngine*, achieving an 18% energy savings while limiting performance loss to 5%.*

1. Introduction

The microprocessor industry, fueled by Moore's law, has continued to provide an exponential rise in the number of transistors that can fit on a single chip. However, transistor threshold voltages have not kept pace with technology scaling, resulting in near constant per-transistor switching energy. These trends create a difficult design dilemma: more transistors can fit on a chip, but the energy budget will not allow them to be used simultaneously. This trend has made it possible for today's computer architects to trade increased area for improved energy efficiency of general purpose processors.

Heterogeneous multicore systems are an effective approach to trade area for improved energy efficiency. These systems comprise multiple cores with different capabilities, yielding varying performance and energy characteristics [20]. In these systems, an application is mapped to the most efficient core that can meet its performance needs. As its performance changes, the application is migrated among the heterogeneous cores. Traditional designs select the best core by briefly sampling performance on each. However, every time the application migrates between cores, its current state must be explicitly transferred or rebuilt on the new core. This state transfer incurs large overheads that limits migration between cores to a *coarse granularity* of tens to hundreds of millions of instructions. To mitigate these effects, the decision to migrate applications is done at the granularity of operating system time slices.

This work postulates that the coarse switching granularity in existing heterogeneous processor designs *limits* their effectiveness and energy savings. What is needed is a tightly coupled heterogeneous multicore system that can support fine-grained switching and is unencumbered by the large state migration latency of current designs.

To accomplish this goal, we propose *Composite Cores*, an architecture that brings the concept of heterogeneity to *within* a single core. A *Composite Core* contains several compute *μEngines* that together can achieve both high performance and energy efficiency. In this work, we consider a dual *μEngine Composite Core* consisting of: a high performance core (referred to as the big *μEngine*) and an energy efficient core (referred to as the little *μEngine*). As only one *μEngine* is active at a time, execution switches dynamically between *μEngines* to best match the current application's characteristics to the hardware resources. This switching occurs on a much finer granularity (on the order of a thousand instructions) compared to past heterogeneous multicore proposals, allowing the application to *spend more time on the energy efficient μEngine without sacrificing additional performance*.

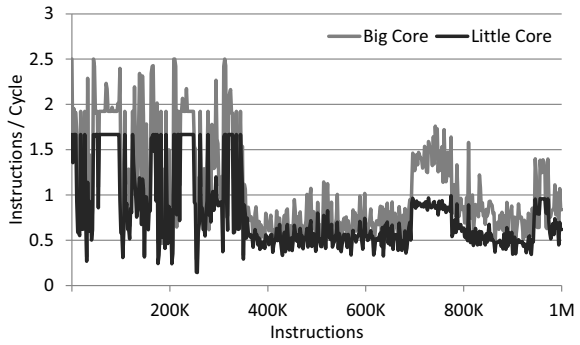
As a *Composite Core* switches frequently between *μEngines*, it relies on hardware resource sharing and low-overhead switching techniques to achieve near zero *μEngine* migration overhead. For example, the big and little *μEngines* share branch predictors, L1 caches, fetch units and TLBs. This sharing ensures that during a switch only the register state needs to be transfered between the cores. We propose a speculative register transfer mechanism to further reduce the migration overhead.

Because of the fine switching interval, conventional sampling-based techniques to select the appropriate core are not well-suited for a *Composite Core*. Instead, we propose an online performance estimation technique that predicts the throughput of the unused *μEngine*. If the predicted throughput of the unused *μEngine* is significantly higher or has better energy efficiency then the active *μEngine*, the application is migrated. Thus, the decision to switch *μEngines* maximizes execution on the more efficient little *μEngine* subject to a performance degradation constraint.

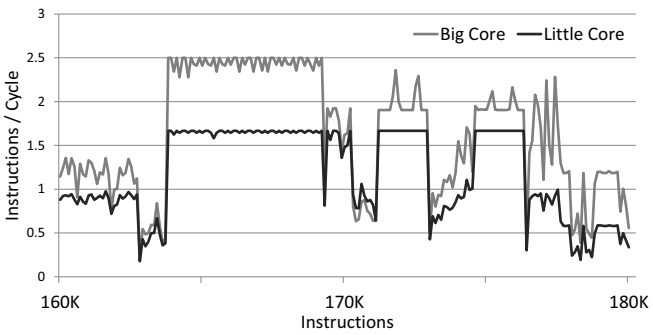
The switching decision logic tracks and predicts the accumulated performance loss and ensures that it remains within a user-selected bound. With *Composite Cores*, we allow the users or system architects to select this bound to trade off performance loss with energy savings. To accomplish this goal, we integrate a simple control loop in our switching decision logic, which tracks the current performance difference based on the performance loss bound, and a reactive model to detect the *instantaneous performance difference* via online perfor-

1072-4451/12 $26.00 © 2012 IEEE
DOI 10.1109/MICRO.2012.37

IEEE computer society



(a) Inst. window of length 2K over a 1M inst. interval



(b) Inst. window of length 100 over a 200K inst. interval

Figure 1: IPC Measured over a typical scheduling interval for 403.gcc

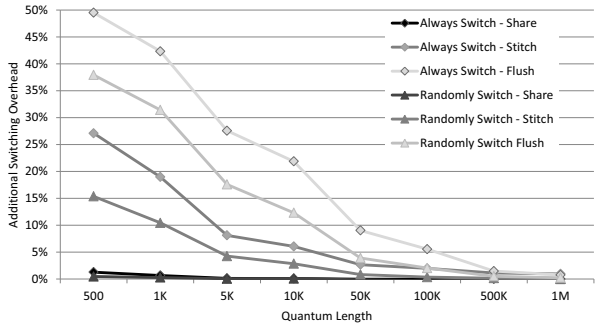


Figure 2: Migration overheads under different switching schemes and probabilities

mance estimation techniques.

In summary, this paper offers the following contributions:

- We propose *Composite Cores*, an architecture that brings the concept of heterogeneity *within* a single core. The *Composite Core* consists of two tightly coupled μ*Engines* that enable fine-grained matching of application characteristics to the underlying microarchitecture to achieve both high performance and energy efficiency.
- We study the benefits of fine-grained switching in the context of heterogeneous core architectures. To achieve near zero μ*Engine* transfer overhead, we propose low-overhead switching techniques and a core microarchitecture which shares necessary hardware resources.
- We design intelligent switching decision logic that facilitates fine-grain switching via predictive rather than sampling-based performance estimation. Our design tightly constrains performance loss within a user-selected bound through a simple feedback controller.
- We evaluate our proposed *Composite Core* architecture with cycle accurate full system simulations and integrated power models. Overall, a *Composite Core* can map an average of *25% of the dynamic execution* to the little μ*Engine* and *reduce energy by 18%* while bounding performance degradation to at most 5%.

2. Motivation

Industry interest in heterogeneous multicore designs has been gaining momentum. Recently ARM announced a heterogeneous multicore, known as big.LITTLE [9], which combines a set of Cortex-A15 (Big) cores with Cortex-A7 (Little) cores to create a heterogeneous processor. The Cortex-A15 is a 3-way out-of-order with deep pipelines (15-25 stages), which is currently the highest performance ARM core that is available. Conversely, the Cortex-A7 is a narrow in-order processor with a relatively short pipeline (8-10 stages). The Cortex-A15 has 2-3x higher performance, but the Cortex-A7 is 3-4x more energy efficient.

In big.LITTLE, all migrations must occur through the coherent interconnect between separate level-2 caches, resulting in a migration cost of about 20 μseconds. Thus, the cost of migration requires that the system migrate between cores only at coarse granularity, on the order of tens of milliseconds. The large switching interval forfeits potential gains afforded by a more aggressive *fine-grained* switching.

2.1. Switching Interval

Traditional heterogeneous multicore systems, such as big.LITTLE, rely on coarse-grained switching to exploit application phases that occur at a granularity of hundreds of millions to billions of instructions. These systems assume the performance within a phase is stable, and simple sampling-based monitoring systems can recognize low-performance phases and map them to a more energy efficient core. While these long term low-performance phases do exist, in many applications, they occur infrequently, limiting the potential to utilize a more efficient core. Several works [27, 32, 33] have shown that observing performance at much finer granularity reveals more low-performance periods, increasing opportunities to utilize a more energy efficient core.

Figure 1(a) shows a trace of the instructions per cycle (IPC) for 403.gcc over a typical operating system scheduling interval of one million instructions for both a three wide out-of-order (big) and a two wide in-order (little) core. Over the entire interval, the little core is an average of 25% slower than the big core, which may necessitate that the entire phase be run on the big core. However if we observe the performance with finer granularity, we observe that, despite some periods of relatively high performance difference, there are numerous periods *where the performance gap between the cores is negligible*.

If we zoom in to view performance at even finer granularity (100s to 1000s of instructions), we find that, even during intervals where the big core outperforms the little on average, there are brief periods where the cores experience similar stalls and the performance gap between them is negligible. Figure 1(b) illustrates a subset of the trace from Figure 1(a) where the big core has nearly forty percent better performance, yet we can see brief regions where there is no performance gap.

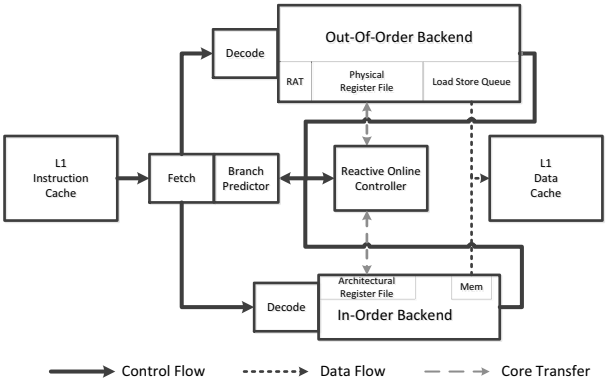


Figure 3: Microarchitectural overview of a *Composite Core*

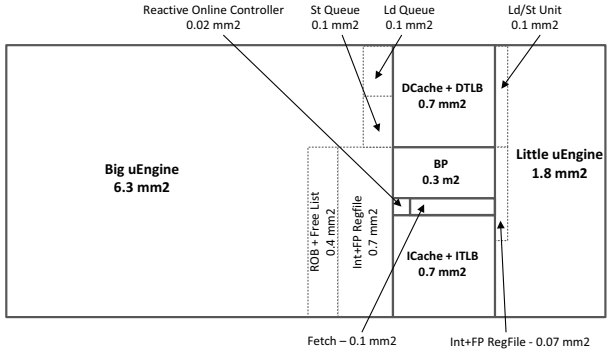


Figure 4: Estimated physical layout of a *Composite Core* in 32*nm* technology

2.2. Migration Overheads

The primary impediment to exploiting these brief low-performance periods is the cost (both explicit and implicit) of migrating between cores. Explicit migration costs include the time required to transport the core's architecturally visible state, including the register file, program counter, and privilege bits. This state must be explicitly stored into memory, migrated to the new core and restored. However, there are also a number of implicit state migration costs for additional state that is not transferred but must be rebuilt on the new core. Several major implicit costs include the extra time required to warm up the L1 caches, branch prediction, and dependence predictor history on the new core.

Figure 2 quantifies the effects of these migration overheads for different architectural implementations by measuring the additional migration overheads of switching at a fixed number of dynamic instructions, called a **quantum** or epoch. The figure shows the effects of switching at the end of every quantum with both a 100% probability (Always Switch) and with a $\frac{1}{3}$ probability (Randomly Switch). The $\frac{1}{3}$ probability is designed to weigh the instruction execution more heavily on the big core to better approximate a more typical execution mix between the big and little cores. The horizontal axis sweeps the quantum length while the vertical axis plots the added overhead due to increased switches.

The "Flush" lines correspond to a design where the core and cache state is invalidated when a core is deactivated (i.e., state is lost due to power gating), for example, ARM's big.LITTLE design. The "Stitch" lines indicate a design where core and cache state is maintained but not updated for inactive cores (i.e., clock gating of stateful structures). Finally, the "Shared" results indicate a design where both cores share all microarchitectural state (except the pipeline) and multiplex access to the same caches, corresponding to the *Composite Cores* approach. Observe that at large quanta, switching overheads are negligible under all three designs. However at small quantum lengths, the added overheads under both "Flush" and "Stitch" cause significant performance loss, while overhead under "Share" remains negligible.

These migration overheads preclude fine-grained switching in traditional heterogeneous core designs. In contrast, a *Composite Core* can leverage shared hardware structures to minimize migration overheads allowing it to target finer-grained switching, improving opportunities to save energy.

3. Architecture

A *Composite Core* consists of two tightly coupled compute μ*Engines* that together can achieve high performance and energy efficiency by rapidly switching between the μ*Engines* in response to changes in application performance. To reduce the overhead of switching, the μ*Engines* share as much state as possible. As Figure 3 illustrates, the μ*Engines* share a front-end, consisting of a fetch stage and branch predictor, and multiplex access to the same L1 instruction and data caches. The register files are kept separate to minimize the little μ*Engine*'s register access energy.

As both μ*Engines* require different control signals from decode, each μ*Engine* has its own decode stage. Each μ*Engine* has a separate back-end implementation, one striving for high performance and the other for increased energy efficiency. However, both μ*Engines* multiplex access to a single L1 data cache, again to maximize shared state and further reduce switching overheads. The register file is the only state that must be explicitly transferred to switch to the opposite μ*Engine*.

The big μ*Engine* is similar to a traditional high performance out-of-order backend. It is a superscalar highly pipelined design that includes complicated issue logic, a large reorder buffer, numerous functional units, a complex load/store queue (LSQ), and register renaming with a large physical register file. The big μ*Engine* relies on these complex structures to support both reordering and speculation in an attempt to maximize performance at the cost of increased energy consumption.

The little μ*Engine* is comparable to a more traditional in-order backend. It has a reduced issue width, simpler issue logic, reduced functional units, and lacks many of the associatively searched structures (such as the issue queue or LSQ). By only maintaining an architectural register file, the little μ*Engine* eliminates the need for renaming and improves the efficiency of register file accesses.

Figure 4 gives an approximate layout of a *Composite Core* system at 32nm. The big μ*Engine* consumes $6.3mm^2$ and the L1 caches consume an additional $1.4mm^2$. The little μ*Engine* adds an additional $1.8mm^2$, or about a 20% area overhead. However, this work assumes that future processors will be limited by power budget rather than transistor area. Finally, the *Composite Core* control logic adds an additional $0.02mm^2$ or an additional 0.2% overhead.

3.1. μ*Engine* Transfer

During execution, the reactive online controller collects a variety of performance metrics and uses these to determine which μ*Engine* should be active for the following quantum. If at the end of the quantum, the controller determines that the next quantum should be run on the inactive μ*Engine*, the *Composite Core* must perform a switch to transfer control to the new μ*Engine*. Figure 5 illustrates the sequence of events when the controller decides to switch μ*Engines*.

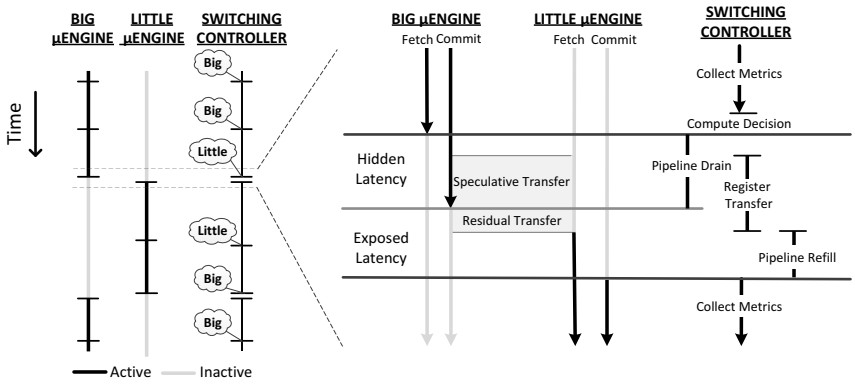


Figure 5: Mechanism of a *Composite Core* switch

As both μ*Engines* have different backend implementations, they have incompatible microarchitectural state. Therefore, when the *Composite Core* decides to switch, the current active μ*Engine* must first be brought to an architecturally precise point before control can be transferred. If the big μ*Engine* is active, it has potentially completed a large amount of work speculatively, making a pipeline flush potentially wasteful. Therefore, the *Composite Core* simply stops fetching instructions to the active μ*Engine*, and allows the pipeline to drain before switching.

As all other stateful structures have either been drained (e.g., pipeline stages) or are shared (e.g., branch predictor), the only state that must be explicitly transferred is the register file. While the active μ*Engine* is draining, the *Composite Core* attempts to speculatively transfer as much of the register state as possible to hide switching latency. Once the active μ*Engine* has completely drained, the remaining registers are transferred during the residual transfer. More details on the register transfer are given in Section 3.2.

Once the register transfer has been completed, fetch resumes with the instructions now redirected to the opposite μ*Engine*. The new μ*Engine* will incur an additional delay while its pipeline stages are refilled. Therefore the total switch latency is the sum of the pipeline drain, register transfer, and the pipeline refill delay. As the pipeline drain is totally hidden and a majority of the register file values can be speculatively transferred, the only exposed latency is the residual register transfer and the pipeline refill latency of the new μ*Engine*. As this latency is similar to that of a branch mispredict, the switching overheads behave very similarly to that of a branch misprediction recovery.

3.2. Register State Transfer

As the register file is the only architecturally visible stateful component that is not shared, its contents must be explicitly transferred during a μ*Engine* switch. This transfer is complicated by the fact that the little μ*Engine* only contains architectural registers with a small number of read and write ports while the big μ*Engine* uses register renaming and a larger multi-ported physical register file. To copy a register from the big μ*Engine* to the little, the architectural-to-physical register mapping must first be determined using the Register Allocation Table (RAT) before the value can be read from the physical register file. Typically this is a two cycle process.

When the *Composite Core* initiates a switch, the registers in the active μ*Engine* are marked as untransferred. The controller then utilizes a pipelined state machine, called the transfer agent, to transfer the registers. The first stage determines the next untransferred register, marks it as transferred, and uses the RAT to lookup the physical register file index corresponding to the architectural register. Recall that while the big μ*Engine* is draining, its RAT read ports are not needed by the pipeline (no new instructions are dispatched). The second stage reads the register's value from the physical register file. The final stage transfers the register to the inactive μ*Engine*.

To hide the latency of the register transfer, the *Composite Core* begins speculatively transferring registers before the active μ*Engine* is fully drained. Therefore, when a register value is overwritten by the draining pipeline it is again marked as untransferred. The transfer agent will then transfer the updated value during the residual transfer of Figure 5. *The transfer agent will continue to run until the pipeline is fully drained and all architectural registers have been transferred.* Once all registers have been transferred, the opposite μ*Engine* can begin execution. The process of transferring registers from the little μ*Engine* to the big μ*Engine* is similar, except there is now a single cycle register read on the little μ*Engine* and a two cycle register write on the big μ*Engine*.

4. Reactive Online Controller

The decision of when to switch is handled by the Reactive Online Controller. Our controller, following the precedent established by prior works [20, 30], attempts to maximize energy savings subject to a configurable maximum performance degradation, or slowdown. The converse, a controller that attempts to maximize performance subject to a maximum energy consumption, can also be constructed in a similar manner.

To determine the appropriate core to minimize performance loss, the controller needs to 1) estimate the dynamic performance loss, which is the difference between the observed performance of the *Composite Core* and the performance if the application were to run *entirely* on the big μ*Engine*; and 2) make switching decisions such that the estimated performance loss is within a parameterizable bound. The controller consists of three main components: a performance estimator, threshold controller, and switching controller illustrated in Figure 6.

The performance estimator tracks the performance on the active μ*Engine* and uses a model to provide an estimate for the performance of the inactive μ*Engine* as well as provide a cumulative performance estimate. This data is then fed into the switching controller, which estimates the performance difference for the following quantum. The threshold controller uses the cumulative performance difference to estimate the allowed performance drop in the next quantum for which running on the little μ*Engine* is profitable. The switching controller uses the output of the performance estimator and the threshold con-

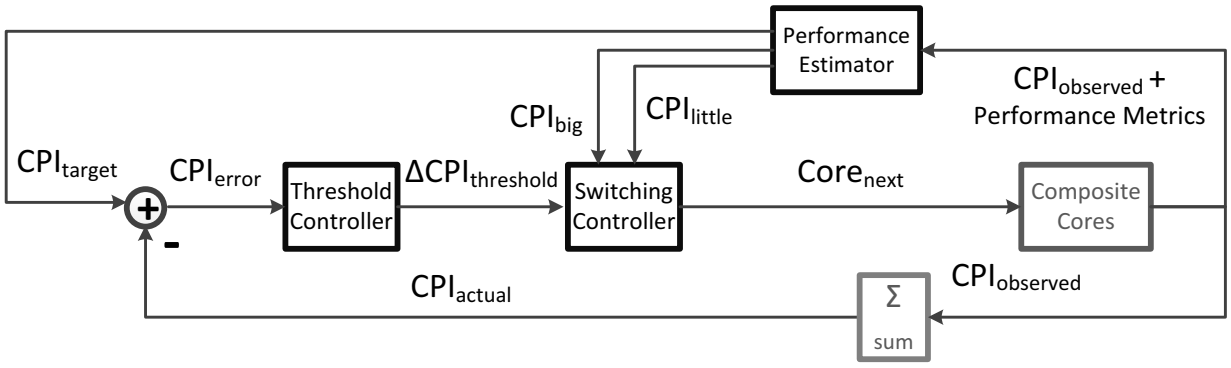


Figure 6: Reactive online controller overview

troller to determine which *µEngine* should be activated for the next quantum.

4.1. Performance Estimator

The goal of this module is to provide an estimate of the performance of both *µEngines* in the *previous* quantum as well as track the overall performance for *all* past quanta. While the performance of the active *µEngine* can be trivially determined by counting the cycles required to complete the current quantum, the performance of the inactive *µEngine* is not known and must be estimated. This estimation is challenging as the microarchitectural differences in the *µEngines* cause their behaviors to differ.

The traditional approach is to sample execution on both *µEngines* for a short duration at the beginning of each quantum and base the decision for the remainder of the quantum on the sample measurements. However, this approach is not feasible for fine-grained quanta for two reasons. First, the additional switching necessary for sampling would require much longer quanta to amortize the overheads, forfeiting potential energy gains. Second, the stability and accuracy of fine-grained performance sampling drops rapidly, since performance variability grows as the measurement length shrinks [32].

Simple rule based techniques, such as switching to the little *µEngine* on a cache miss, cannot provide an effective performance estimate needed to allow the user to configure the performance target. As this controller is run frequently, more complex approaches, such as non-linear or neural-network models, add too much energy overhead and hardware area to be practical.

Therefore the *Composite Core* instead monitors a selected number of performance metrics on the active *µEngine* that capture fundamental characteristics of the application and uses a simple performance model to estimate the performance of the inactive *µEngine*. A more detailed analysis of the performance metrics is given in Section 4.4.

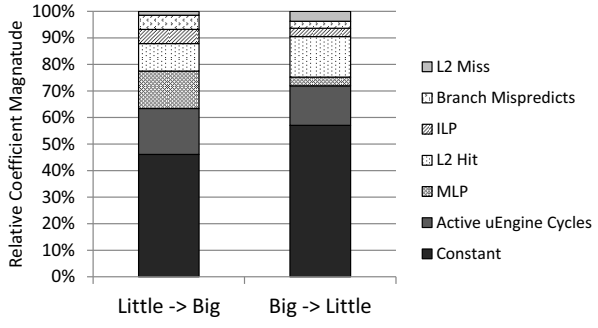


Figure 7: Magnitude of regression coefficients

4.1.1. Performance Model The performance model provides an estimate for the inactive *µEngine* by substituting the observed metrics into a model for the inactive *µEngine*'s performance. As this computation must be performed often, we chose a simple linear model to minimize computation overhead. Eq. 1 defines the model, which consists of the sum of a constant coefficient (a_0) and several input metrics (x_i) times a coefficient (a_i). As the coefficients are specific to the active *µEngine*, two sets of coefficients are required, one set is used to estimate performance of the big *µEngine* while the little *µEngine* is active, and vice versa.

$$y = a_0 + \sum a_i x_i \quad (1)$$

To determine the coefficients for the performance monitor, we profile each of the benchmarks on both the big and little *µEngine* for 100 million instructions (after a 2 Billion instruction fast-forward) using each benchmark's supplied training input set. We then utilize ridge regression analysis to determine the coefficients using the aggregated performance metrics from all benchmarks. The magnitude of each normalized coefficient for both models is shown in Figure 7, illustrating the relative importance of each metric to overall performance for each *µEngine*.

The *constant* term reflects the baseline weight assigned to the average performance of the active *µEngine* without considering the metrics. The *Active µEngine Cycles* metric scales the model's estimate based on the CPI of the active *µEngine*. *MLP* attempts to measure the levels of memory parallelism and account for the *µEngine*'s ability to overlap memory accesses. *L2 Hit* tracks the number of L2 cache hits and scales the estimate to match the *µEngine*'s ability to tolerate medium latency misses. *ILP* attempts to scale the performance estimate based on the inactive *µEngine*'s ability (or inability) to exploit independent instructions. *Branch Mispredicts* and *L2 Miss* scales the estimate based on the number of branch mispredictions and L2 cache misses respectively.

Little->Big Model: This model is used to estimate the performance of the big *µEngine* while the little *µEngine* is active. In general good performance on the little *µEngine* indicates good performance on the big *µEngine*. As the big *µEngine* is better able to exploit both MLP and ILP its performance can improve substantially over the little for applications that exhibit these characteristics. However, the increased pipeline length of the big *µEngine* makes it slower at recovering from a branch mispredict than the little *µEngine*, decreasing the performance estimate. Finally, as L2 misses occur infrequently and the big *µEngine* is designed to partially tolerate memory latency, the L2 Miss coefficient has minimal impact on the overall estimate.

Big->Little Model: While the big $\mu Engine$ is active, this model estimates the performance of the little $\mu Engine$. The little $\mu Engine$ has a higher constant due to its narrower issue width causing less performance variance. As the little $\mu Engine$ cannot exploit application characteristics like ILP and MLP as well as the big $\mu Engine$, the big $\mu Engine$'s performance has slightly less impact than in the Little->Big model. L2 Hits are now more important as, unlike the big $\mu Engine$, the little $\mu Engine$ is not designed to hide any of the latency. The inability of the little $\mu Engine$ to utilize the available ILP and MLP in the application causes these metrics to have almost no impact on the overall performance estimate. Additionally, as the little $\mu Engine$ can recover from branch mispredicts much faster, mispredicts have very little impact. Finally even though L2 misses occur infrequently, the little $\mu Engine$ suffers more performance loss than the big $\mu Engine$ again due to the inability to partially hide the latency.

Per-Application Model: While the above coefficients give a good approximation for the performance of the inactive $\mu Engine$, some applications will warrant a more exact model. For example, in the case of memory bound applications like `mcf`, the large number of L2 misses and their impact on performance necessitates a heavier weight for the L2 Miss metric in the overall model. Therefore the architecture supports the use of per-application coefficients for both the Big->Little and Little->Big models, allowing programmers to use offline profiling to custom tailor the model to the exact needs of their application if necessary. However, our evaluation makes use of generic models.

4.1.2. Overall Estimate The second task of the performance estimator is to track the actual performance of the *Composite Core* as well as provide an estimate of the target performance for the entire application. The actual performance is computed by summing the observed performance for all quanta (Eq. 2). The target performance is computed by summing all the observed and estimated performances of the big $\mu Engine$ and scaling it by the allowed performance slowdown. (Eq. 3). As the number of instructions is always fixed, rather than compute CPI the performance estimator hardware only sums the number of cycles accumulated, and scales the target cycles to compare against the observed cycles.

$$CPI_{actual} = \sum CPI_{observed} \tag{2}$$

$$CPI_{target} = \sum CPI_{Big} \times (1 - Slowdown_{allowed}) \tag{3}$$

4.2. Threshold Controller

The threshold controller is designed to provide a measure of the current maximum performance loss allowed when running on the little $\mu Engine$. This threshold is designed to provide an average per-quantum performance loss where using the little $\mu Engine$ is profitable given the performance target. As some applications experience frequent periods of similar performance between $\mu Engines$, the controller scales the threshold low to ensure the little $\mu Engine$ is only used when it is of maximum benefit. However for applications that experience almost no low performance periods, the controller scales the threshold higher allowing the little $\mu Engine$ to run with a larger performance difference but less frequently.

The controller is a standard PI controller shown in Eq. 5. The P (Proportional) term attempts to scale the threshold based on the current observed error, or difference from the expected performance (Eq. 4). The I (Integral) term scales the threshold based on the sum of all past errors. A Derivative term can be added to minimize overshoot.

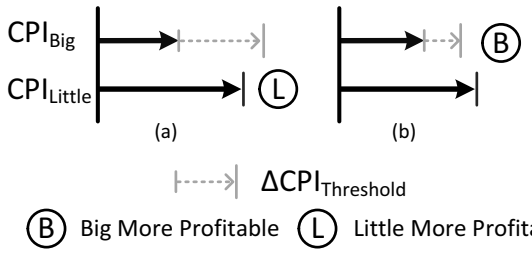


Figure 8: Switching controller behaviour: (a) If $CPI_{big} + \Delta CPI_{threshold} > CPI_{little}$ pick Little; (b) If $CPI_{big} + \Delta CPI_{threshold} < CPI_{little}$ pick Big.

However in our case, it was not included due to noisiness in the input signal. Similar controllers have been used in the past for controlling performance for DVFS [29].

The constant K_p and K_i terms were determined experimentally. The K_p term is large, reflecting the fact that a large error needs to be corrected immediately. However, this term suffers from systematically underestimating the overall performance target. Therefore the second term, K_i is introduced to correct for small but systematic under-performance. This term is about three orders of magnitude smaller than K_p, so that it only factors into the threshold when a long-term pattern is detected.

$$CPI_{error} = CPI_{target} - CPI_{actual} \tag{4}$$

$$\Delta CPI_{threshold} = K_p CPI_{error} + K_i \sum CPI_{error} \tag{5}$$

4.3. Switching Controller

The switching controller attempts to determine which $\mu Engine$ is most profitable for the next quantum. To estimate the next quantum's performance, the controller assumes the next quantum will have the same performance as the previous quantum. As show in Figure 8, the controller determines profitability by computing ΔCPI_{net} as shown in Eq. 6. If ΔCPI_{net} is positive, the little $\mu Engine$ is currently more profitable, and execution is mapped to the little $\mu Engine$ for the next quantum. However, if ΔCPI_{net} is negative, the performance difference between big and little is too large, making the little $\mu Engine$ less profitable. Therefore the execution is mapped to the big $\mu Engine$ for the next quantum.

$$\Delta CPI_{net} = (CPI_{Big} + \Delta CPI_{threshold}) - CPI_{little} \tag{6}$$

4.4. Implementation Details

We use several performance counters to generate the detailed metrics required by the performance estimator. Most of these performance counters are already included in many of today's current systems, including branch mispredicts, L2 cache hits and L2 cache misses. Section 4.4.1 details the additional performance counters needed in the big $\mu Engine$. Due to the microarchitectural simplicity of the little $\mu Engine$, tracking these additional metrics is more complicated. We add a small dependence table (described in Section 4.4.2) to the little $\mu Engine$ to capture these metrics.

4.4.1. Performance Counters The performance models rely heavily on measurements of both ILP and MLP, which are not trivially measurable in most modern systems. As the big $\mu Engine$ is already equipped with structures that exploit both ILP and MLP, we simply add a few low overhead counters to track these metrics. For ILP, a performance counter keeps a running sum of the number of instructions in the issue stage that are waiting on values from in-flight instructions.

This captures the number of instructions stalled due to serialization as an inverse measure of ILP. To measure MLP, an additional performance counter keeps a running sum of the number of MSHR entries that are in use at each cache miss. While not perfect measurements, these simple performance counters give a good approximation of the amount of ILP and MLP per quantum.

4.4.2. Dependence Table Measuring ILP and MLP on the little μ*Engine* is challenging as it lacks the microarchitectural ability to exploit these characteristics and therefore has no way of measuring them directly.

We augment the little μ*Engine* with a simple table that dynamically tracks data dependence chains of instructions to measure these metrics. The design is from Chen, Dropsho, and Albonesi [7]. This table is a bit matrix of registers and instructions, allowing the little μ*Engine* to simply look up the data dependence information for an instruction. A performance counter keeps a running sum per quantum to estimate the overall level of instruction dependencies as a measure of the ILP. To track MLP, we extended the dependence table to track register dependencies between cache misses over the same quantum. Together these metrics allow *Composite Cores* to estimate the levels of ILP and MLP available to the big μ*Engine*.

However, there is an area overhead associated with this table. The combined table contains two bits of information for each register over a fixed instruction window. As our architecture supports 32 registers and we have implemented our instruction window to match the length of the ROB in the big μ*Engine*, the total table size is $2 \times 32 \times 128$ bits, 1KB of overhead. As this table is specific to one μ*Engine*, the additional area is factored into the little μ*Engine*'s estimate rather than the controller.

4.4.3. Controller Power & Area To analyze the impact of the controller on the area and power overheads, we synthesized the controller design in an industrial 65nm process. The design was placed and routed for area estimates and accurate parasitic values. We used Synopsys PrimeTime to obtain power estimates which we then scaled to the 32nm target technology node. The synthesized design includes the required performance counters, multiplicand values (memory-mapped programmable registers), and a MAC unit. For the MAC unit, we use a fixed-point 16*16+36-bit Overlapped bit-pair Booth recoded, Wallace tree design based on the Static CMOS design in [18]. The design is capable of meeting a 1.0GHz clock frequency and completes 1 MAC operation per cycle, with a 2-stage pipeline.

Thus, the calculations in the performance model can be completed in 9 cycles as our model uses 7 input metrics. With the added computations for the threshold controller and switching controller, the final decision takes approximately 30 cycles. The controller covers $0.02mm^2$ of area, while consuming less than $5uW$ of power during computation. The MAC unit could be power gated during the remaining cycles to reduce the leakage power while not in use.

5. Results

To evaluate the *Composite Cores* architecture, we extended the Gem5 Simulator [6] to support fast switching. All benchmarks were compiled using gcc with -O2 optimizations for the Alpha ISA. We evaluated all benchmarks by fast forwarding for two billion instructions before beginning detailed simulations for an additional one billion instructions. The simulations included detailed modeling of the pipeline drain functionality for switching μ*Engines*.

We utilized McPAT to estimate the energy savings from a *Composite Core* [28]. We model the two main sources of energy loss in transistors, dynamic energy and static (or leakage) energy. We study only the effects of clock gating, due to the difficulties in estimating the performance and energy implications of power gating. Finally, as our design assumes tightly coupled L1 caches, our estimates include the energy consumption of the L1 instruction and data caches, but exclude all other system energy estimates.

Architectural Feature	Parameters
Big μ*Engine*	3 wide Out-Of-Order @ 1.0GHz 12 stage pipeline 128 ROB entries 160 entry register file Tournament branch predictor (Shared)
Little μ*Engine*	2 wide In-Order @ 1.0GHz 8 stage pipeline 32 entry register file Tournament branch predictor (Shared)
Memory System	32 KB L1 iCache, 1 cycle access (Shared) 32 KB L1 dCache, 1 cycle access (Shared) 1 MB L2 Cache, 15 cycle access 1024MB Main Mem, 80 cycle access

Table 1: Experimental *Composite Core* parameters

Table 1 gives more specific simulation configurations for each of the μ*Engines* as well as the memory system configuration. The big μ*Engine* is modeled as a 3-wide out-of-order processor with a 128-entry ROB and a 160-entry physical register file. It is also aggressively pipelined with 12 stages. The little μ*Engine* is modeled to simulate a 2-wide in-order processor. Due to its simplified hardware structures the pipeline length is also shorter, providing quicker branch misprediction recovery, and it only contains a 32-entry architectural register file. The branch predictor and fetch stage are shared between the two μ*Engines*.

5.1. Quantum Length

One of the primary goals of the *Composite Cores* architecture is to explore the benefits of fine-grained quanta to exploit short duration periods of low performance. To determine the optimum quantum length, we performed detailed simulations to sweep quantum lengths with several assumptions that will hold for the remainder of Section 5.1. To factor out controller inaccuracies, we assume the μ*Engine* selection is determined by an oracle, which knows the performance for both μ*Engines* for all quanta and switches to the little μ*Engine* only for the quanta with the smallest performance difference such that it can still achieve the performance target. We also assume that the user is willing to tolerate a 5% performance loss relative to running the entire application on the big μ*Engine*.

Given these assumptions, Figure 9 demonstrates the little μ*Engine*'s utilization measured in dynamic instructions as the quantum length varies. While the memory-bound `mcf` can almost fully utilize the little μ*Engine* at larger quanta, the remaining benchmarks show only a small increase in utilization until the quantum length decreases to less than ten thousand instructions. Once quantum sizes shrink below this level, the utilization begins to rise rapidly from approximately thirty percent to fifty percent at quantum lengths of one hundred instructions.

While a *Composite Core* is designed to minimize migration overheads, there is still a small register transfer and pipeline refill latency when switching μ*Engines*. Figure 10 illustrates the performance impacts of switching μ*Engines* at various quanta with the oracle targeting 95% performance relative to the all big μ*Engine* case. We observe that, with the exception of `mcf`, which actually achieves a small speedup, all the benchmarks achieve the target performance at

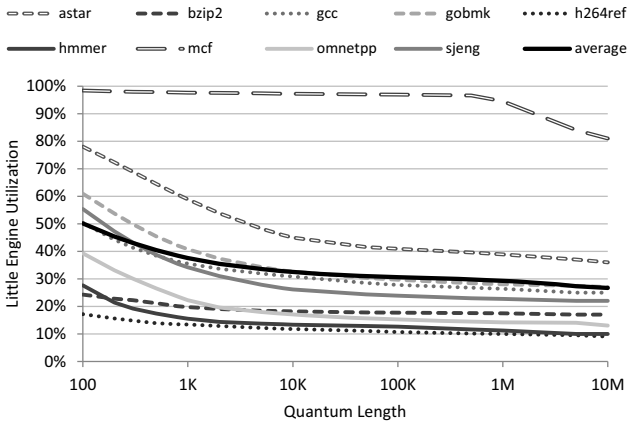


Figure 9: Impact of quantum length on little μ*Engine* utilization

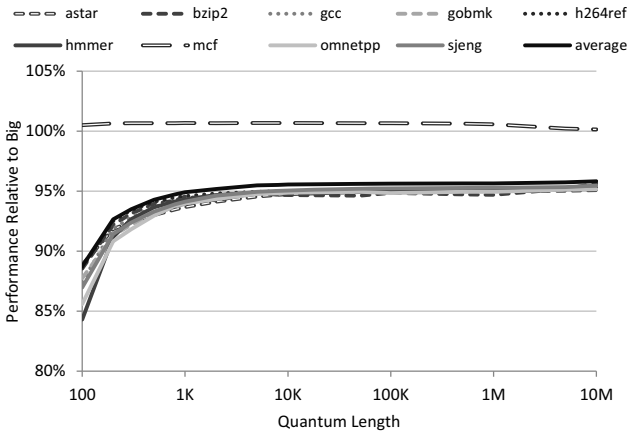


Figure 10: Impact of quantum length on overall performance with a 5% slowdown target

longer quanta. This result implies that the additional overheads of switching μ*Engines* are negligible at these quanta and can safely be ignored. However, for quantum lengths smaller than 1000 instructions we begin to see additional performance degradation, indicating that the overheads of switching are no longer negligible.

The main cause of this performance decrease is the additional switches allowed by the smaller quanta. Figure 11 illustrates the number of switches per million instructions the *Composite Core* performed to achieve its goal of maximizing the little μ*Engine* utilization. Observe that as the quantum length decreases, there is a rapid increase in the number of switches. In particular, for a quantum length of 1000 the oracle switches cores approximately 340 times every million instructions, or roughly every 3000 instructions.

As quantum length decreases the *Composite Core* has greater potential to utilize the little μ*Engine*, but must switch more frequently to achieve this goal. Due to increased hardware sharing, the *Composite Core* is able to switch at a much finer granularity than traditional heterogeneous multicore architectures. However below quantum lengths of approximately 1000 dynamic instructions, the overheads of switching begin to cause intolerable performance degradation. Therefore for the remainder of this study, we will assume quantum lengths of 1000 instructions.

5.2. μ*Engine* Power Consumption

A *Composite Core* relies on shared hardware structures to enable fine-grained switching. However these shared structures must be designed

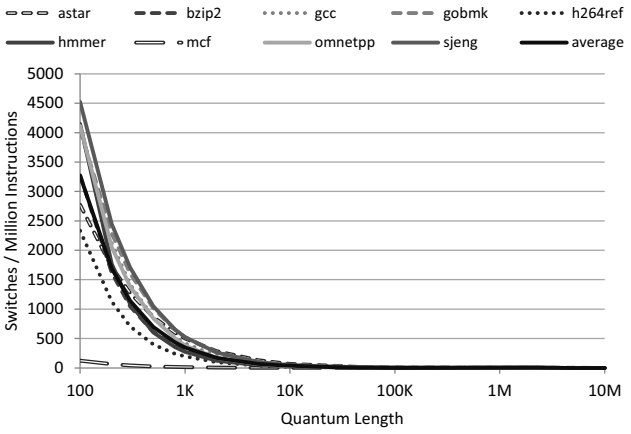


Figure 11: Impact of quantum length on μ*Engine* switches

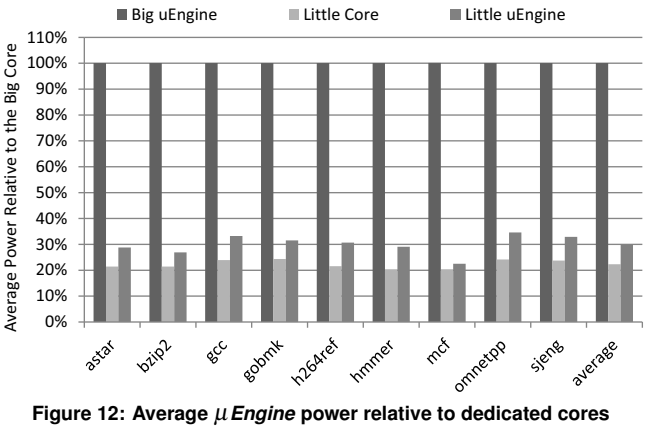


Figure 12: Average μ*Engine* power relative to dedicated cores

for the high performance big μ*Engine* and are over-provisioned when the little μ*Engine* is active. Therefore the little μ*Engine* has a higher average power than a completely separate little core. When the little μ*Engine* is active, its frontend now includes a fetch engine, branch predictor, and instruction cache designed for the big μ*Engine*. Also, the little μ*Engine* accesses a data cache that is designed to support multiple outstanding memory transactions. While this functionality is necessary for the big μ*Engine*, the little μ*Engine* cannot utilize it. Finally, the leakage power of *Composite Cores* will be higher as it is comprised of two μ*Engines*.

Figure 12 illustrates the average power difference between the μ*Engines* and separate big and little cores. Observe that while the big μ*Engine* includes the leakage of the little μ*Engine*, it does not use noticeably more power than a separate big core. As the little μ*Engine* is small, its contribution to leakage is minimal. However, while the little core requires only 22% of the big core's power, the shared hardware of the little μ*Engine* only allow it to reduce the power to 30% of the big core. This is caused by a combination of both the leakage energy of the big μ*Engine* and the inefficiencies inherent in using an over-provisioned frontend and data cache.

While the little μ*Engine* of *Composite Core* is not able to achieve the same power reductions as a separate little core, this limitation is offset by *Composite Core*'s ability to utilize the little μ*Engine* more frequently. As illustrated in Figure 9, a *Composite Core*, with a quantum length of 1000 instructions, executes more instructions on the little μ*Engine* than a traditional heterogeneous multicore system, which has a quantum length of 10 million instructions or more. Even

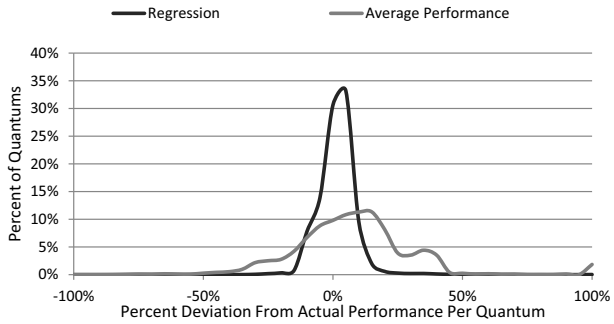


Figure 13: Distribution of Big->Little regression accuracy

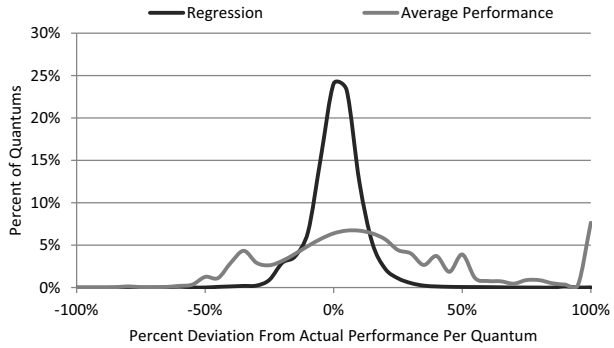


Figure 14: Distribution of Little->Big regression accuracy

after accounting for the inefficiencies of the little μ*Engine*, a *Composite Core* is still able to achieve a 27% decrease in average power compared to the traditional heterogeneous multicore approach.

5.3. Regression

While the oracle switching scheme was useful to determine the best quantum length, it is not implementable in a real system. Therefore in this section, we evaluate the accuracies of the performance model from Section 4.1. Figures 13 and 14 illustrate the accuracy for the Big->Little and Little->Big models respectively. The y-axis indicates the percent of the total quanta, or scheduling intervals. The x-axis indicates the difference between the estimated and actual performance for a single quantum. The accuracy of using a fixed estimate equal to the average performance of the inactive μ*Engine* is also given for comparison.

As the little μ*Engine* has less performance variance and fewer features, it is easier to model, and the Big->Little model is more accurate. However, the Little->Big model must predict the performance of the big μ*Engine*, which has hardware features that were designed to overlap latency, causing it to be less accurate. Also note that while the individual predictions have a larger tail, the tail is centered around zero error. Hence over a large number of quanta, positive errors are canceled by negative errors, allowing the overall performance estimate, CPI_{target} to be more accurate despite the variations in the models themselves.

5.4. Little Core Utilization

For Section 5.4-5.6 we evaluate three different switching schemes configured to allow a maximum of 5% performance degradation. The **Oracle** is the same as in Section 5.1 and picks only the best quanta to run on the little μ*Engine* so that it can still achieve its performance target. The **Perfect Past** has oracle knowledge of the past quanta only, and relies on the assumption that the next quantum has the same

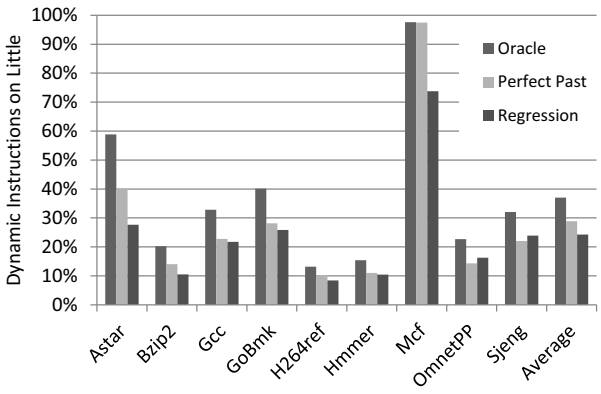


Figure 15: Little μ*Engine* utilization in dynamic instructions, for different switching schemes

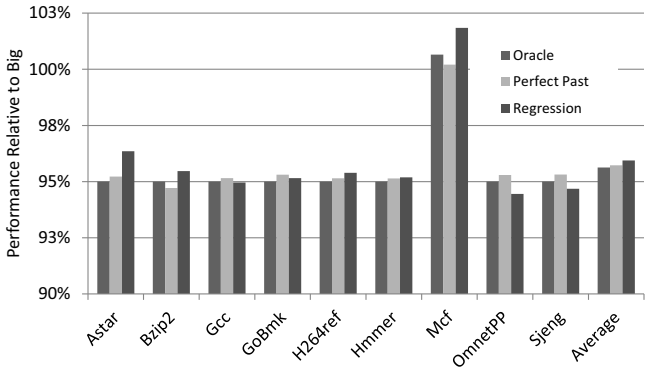


Figure 16: Performance impact for various switching schemes with a 5% slowdown target

performance as the most recent past quantum. The realistic **Regression Model** can measure the performance of the active μ*Engine*, but must rely on a performance model for the estimated performance of the inactive μ*Engine*. This model is the same for all benchmarks and was described in Section 4.1.

Figure 15 illustrates the little μ*Engine* utilization, measured in dynamic instructions, for various benchmarks using each switching scheme. For a memory bound application, like `mcf`, a *Composite Core* can map nearly 100% of the execution to the little μ*Engine*. For applications that are almost entirely computation bound with predictable memory access patterns, the narrower width of the little μ*Engine* limits its overall utilization. However, most applications lie somewhere between these extremes and the *Composite Core* is able to

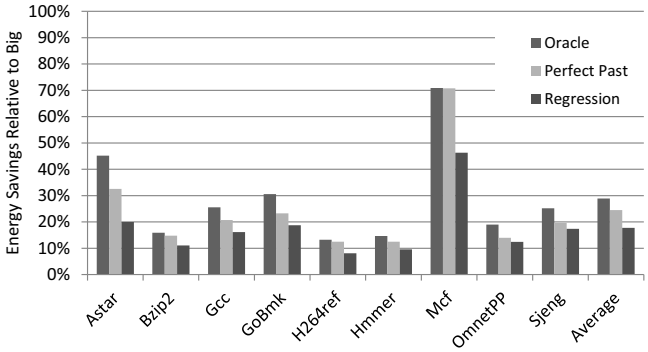


Figure 17: Energy savings for various switching schemes

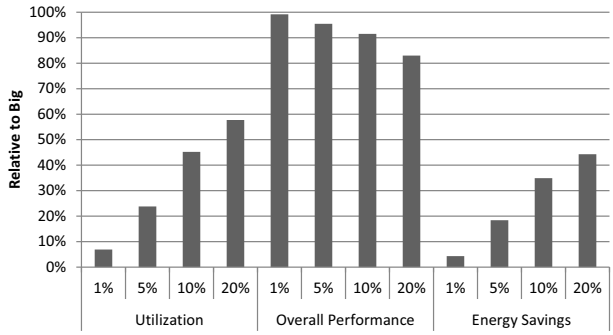


Figure 18: Slowdown sensitivity analysis

map between 20% to 60% of the instructions given oracle knowledge, with an average of 37% utilization. Given the imperfect regression model, these utilizations drop slightly, but still maintain an average utilization of 25% across all benchmarks. Finally on `omnetpp` and `sjeng`, the regression scheme actually achieves higher utilization than the perfect past, however this comes at the cost of a performance loss that is slightly below the target described in Section 5.5.

5.5. Performance Impact

Figure 16 illustrates the performance of the *Composite Core* relative to running the entire application on the big μ*Engine*. *Composite Core* is configured to allow a 5% slowdown, so the controller is targeting 95% relative performance. As `mcf` is almost entirely memory bound, the decrease in branch misprediction recovery latency actually causes a small performance speedup. All other benchmarks are at or near the target performance for all schemes. Note that the controller is designed to allow a small amount of oscillation around the exact performance target to reduce unnecessary switching, thus allowing `bzip2` to dip slightly below the target for the perfect past switching scheme. Both `omnetpp` and `sjeng` suffer from slight inaccuracies in the regression model which, when combined with the oscillation of the controller, causes their overall performance to be approximately an additional $\frac{1}{2}$% below the target performance.

5.6. Energy Reduction

Figure 17 illustrates the energy savings for different switching schemes across all benchmarks. Note that these results only assume clock-gating, meaning that both cores are always leaking static energy regardless of utilization. Again, as `mcf` is almost entirely memory bound, the *Composite Core* is able to map almost the entire execution to the little μ*Engine* and achieve significant energy savings. Overall, the oracle is able to save 29% the energy. Due to the lack of perfect knowledge, the perfect past scheme is not able to utilize the little μ*Engine* as effectively, reducing its overall energy savings to 24%. Finally, the implementable regression model achieves 18% energy savings as the additional inaccuracies in the regression model further reduce the effective utilization of the little μ*Engine*. When combined with the performance, the regression model is able to achieve a 21% reduction in EDP.

5.7. Allowed Performance Loss

As the *Composite Core* can be controlled to provide different levels of energy savings by specifying permissible performance slowdowns, the end user or OS can choose how much of a performance loss is tolerable in exchange for energy savings. Figure 18 illustrates the little μ*Engine* utilization, performance, and energy savings relative to the big μ*Engine* for various performance levels. As the system is tuned to permit a higher performance drop, utilization of the little μ*Engine* increases resulting in higher energy savings. Allowing only a 1% slowdown saves up to 4% of the energy whereas tuning to a 20% performance drop can save 44% of the energy consumed on the big μ*Engine*. This ability is particularly useful in situations where maintaining usability is essential, such as low-battery levels on laptops and cell phones.

6. Related Works

Numerous works motivate a heterogeneous multi-core design for the purposes of performance [22, 2, 4], power [20], and alleviating serial bottlenecks [10, 30, 13]. This paradigm has even begun to make its way into commercial products [9]. The heterogeneous design space can be broadly categorized into 1) designs which migrate thread context across heterogeneous processors, 2) designs which allow a thread to adapt (borrow, lend, or combine) hardware resources, and 3) designs which allow dynamic voltage/frequency scaling.

6.1. Heterogeneous Cores, Migratory Threads

Composite Cores falls within the category of designs which migrate thread context. Most similarly to our technique, Kumar et al. [20] consider migrating thread context between out-of-order and in-order cores for the purposes of reducing power. At coarse granularities of 100M instructions, one or more of the inactive cores are sampled by switching the thread to each core in turn. Switches comprise flushing dirty L1 data to a shared L2, which is slow and energy consuming. Rather than relying on sampling the performance on both cores, Van Craeynest et al. [31] propose a coarse-grained mechanism that relies on measures of CPI, MLP, and ILP to predict the performance on the inactive core.

On the other hand, Rangan et al. [27] examine a CMP with clusters of in-order cores sharing L1 caches. While the cores are identical architecturally, varied voltage and frequency settings create performance and power heterogeneity. A simple performance model is made possible by having exclusively in-order cores, and thread migration is triggered every 1000 cycles by a history-based (last value) predictor. Our solution combines the benefits of architectural heterogeneity [21], as well as those of fast migration of only register state, and contributes a sophisticated mechanism to estimate the inactive core's performance.

Another class of work targets the acceleration of bottlenecks to thread parallelism. Segments of code constituting bottlenecks are annotated by the compiler and scheduled at runtime to run on a big core. Suleman et al. [30] describe a detailed architecture and target critical sections, and Joao et al. [13] generalize this work to identify the most critical bottlenecks at runtime. Patsilaras, Choudhary, and Tuck [25] propose building separate cores, one that targets MLP and the other that targets ILP. They then use L2 cache miss rate to determine when an application has entered a memory intensive phase and map it to the MLP core. When the cache misses decrease, the system migrates the application back to the ILP core.

Other work studies the benefits of heterogeneity in real systems. Annavaram et al. [2] show the performance benefits of heterogeneous multi-cores for multithreaded applications on a prototype with different frequency settings per core. Kwon et al. [23] motivate asymmetry-aware hypervisor thread schedulers, studying cores with various voltage and frequency settings. Koufaty et al. [17] discover an application's big or little core bias by monitoring stall sources,

to give preference to OS-level thread migrations which migrate a thread to a core it prefers. A heterogeneous multi-core prototype is produced by throttling the instruction retirement rate of some cores down to one instruction per cycle.

6.2. Adaptive Cores, Stationary Threads

Alternatively, asymmetry can be introduced by dynamically adapting a core's resources to its workload. Prior work has suggested adapting out-of-order structures such as the issue queue [3], as well as other structures such as ROBs, LSQs, and caches [26, 5, 1]. Kumar et al. [19] explored how a pair of adjacent cores can share area-expensive structures, while keeping the floorplan in mind. Homayoun et al. [11] recently examined how microarchitectural structures can be shared across 3D stacked cores. These techniques are limited by the structures they adapt and cannot for instance switch from an out-of-order core to an in-order core during periods of low ILP.

Ipek et al. [12] and Kim et al. [14] describe techniques to compose or fuse several cores into a larger core. While these techniques provide a fair degree of flexibility, a core constructed in this way is generally expected to have a datapath that is less energy efficient than if it were originally designed as an indivisible core of the same size.

6.3. Dynamic Voltage and Frequency Scaling (DVFS)

DVFS approaches reduce the voltage and frequency of the core to improve the core's energy efficiency at the expense of performance. However, when targeted at memory-bound phases, this approach can be effective at reducing energy with minimal impact on performance. Similar to traditional heterogeneous multicore systems, the overall effectiveness of DVFS suffers from coarse-grained scheduling intervals in the millisecond range. In addition, providing independent DVFS settings for more than two cores is costly in terms of both area and energy [16]. Finally, traditional DVFS is only effective when targeting memory-bound phases, while the *Composite Core* architecture can also target phases of serial computation, low instruction level parallelism and high branch-misprediction rates.

Despite these limitations, DVFS is still widely used in production processors today, and has been incorporated into ARM's big.LITTLE heterogeneous multicore system [9]. Similar to big.LITTLE, DVFS could easily be incorporated into a *Composite Core* design. Here the operating system would attempt to maximize energy savings by reducing the voltage for the entire *Composite Core* at a coarse granularity of multiple operating system scheduling intervals. Within these intervals, the *Composite Core* would act as an additional layer of optimization by exploiting fine-grained phases to further reduce energy consumption. This approach can designed to achieve maximum energy savings by allowing DVFS and *Composite Core* to work together to save energy by targeting both coarse-grained and fine-grained phases.

In the future, a *Composite Core* may be able to utilize heterogenity in terms of both microarchitecture and voltage/frequency scaling to further improve energy efficiency. Two competing techniques to enable fine-grained DVFS, fast on-chip regulators [16, 15] and dual-voltage rails [24, 8], have recently been proposed that promise to deliver transition latencies similar to that of a *Composite Core*. These would allow the *Composite Core* to simultaneously switch μ*Engines* and scale the operating voltage/frequency to further maximize energy savings.

7. Conclusion

This paper explored the implications of migration between heterogeneous systems at a much finer granularity than previously proposed. We demonstrated the increased potential to utilize a more energy efficient core at finer intervals than traditional heterogeneous multicore systems. We proposed *Composite Cores*, an architecture that brings the concept of heterogeneity from between different cores to *within* a core by utilizing two tightly coupled μ*Engines*. A *Composite Core* takes advantages of increased hardware sharing to enable fine-grained switching while achieving near zero migration overheads. The *Composite Core* also includes an intelligent controller designed to maximize the utilization of the little μ*Engine* while constraining performance loss to a user-defined threshold. Overall, our system can map an average of *25% of the dynamic execution* to the little μ*Engine* and *reduce energy by 18%* while maintaining a *95% performance target*.

8. Acknowledgements

This work is supported in part by ARM Ltd and by the National Science Foundation under grant SHF-1227917. The authors would like to thank the fellow members of the CCCP research group, our shepherd (Krste Asanovic), and the anonymous reviewers for their time, suggestions, and valuable feedback.

References

[1] D. Albonesi, R. Balasubramonian, S. Dropsbo, S. Dwarkadas, E. Friedman, M. Huang, V. Kursun, G. Magklis, M. Scott, G. Semeraro, P. Bose, A. Buyuktosunoglu, P. Cook, and S. Schuster, "Dynamically tuning processor resources with adaptive processing," *IEEE Computer*, vol. 36, no. 12, pp. 49 –58, Dec. 2003.

[2] M. Annavaram, E. Grochowski, and J. Shen, "Mitigating amdahl's law through epi throttling," in *Proceedings of the 32nd annual international symposium on Computer Architecture*, 2005, pp. 298–309.

[3] R. Bahar and S. Manne, "Power and energy reduction via pipeline balancing," *Proc. of the 28th Annual International Symposium on Computer Architecture*, vol. 29, no. 2, pp. 218–229, 2001.

[4] S. Balakrishnan, R. Rajwar, M. Upton, and K. Lai, "The impact of performance asymmetry in emerging multicore architectures," in *Proc. of the 32nd Annual International Symposium on Computer Architecture*, Jun. 2005, pp. 506 – 517.

[5] R. Balasubramonian, D. Albonesi, A. Buyuktosunoglu, and S. Dwarkadas, "Memory hierarchy reconfiguration for energy and performance in general-purpose processor architectures," in *Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, 2000, pp. 245–257.

[6] N. Binkert *et al.*, "The gem5 simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, Aug. 2011.

[7] L. Chen, S. Dropsho, and D. Albonesi, "Dynamic data dependence tracking and its application to branch prediction," in *Proc. of the 9th International Symposium on High-Performance Computer Architecture*, 2003, pp. 65–.

[8] R. Dreslinski, "Near threshold computing: From single core to many-core energy efficient architectures," Ph.D. dissertation, University of Michigan, 2011.

[9] P. Greenhalgh, "Big.little processing with arm cortex-a15 & cortex-a7," Sep. 2011, http://www.arm.com/files/downloads/big_LITTLE_Final_Final.pdf.

[10] M. Hill and M. Marty, "Amdahl's law in the multicore era," *IEEE Computer*, no. 7, pp. 33 –38, 2008.

[11] H. Homayoun, V. Kontorinis, A. Shayan, T.-W. Lin, and D. M. Tullsen, "Dynamically heterogeneous cores through 3d resource pooling," in *Proc. of the 18th International Symposium on High-Performance Computer Architecture*, 2012, pp. 1–12.

[12] E. Ipek, M. Kirman, N. Kirman, and J. Martinez, "Core fusion: Accommodating software diversity in chip multiprocessors," in *Proc. of the 34th Annual International Symposium on Computer Architecture*, 2007, pp. 186–197.

[13] J. A. Joao, M. Suleman, O. Mutlu, and Y. N. Patt, "Bottleneck identification and scheduling in multithreaded applications," in *20th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2012, pp. 223–234.

[14] C. Kim, S. Sethumadhavan, M. S. Govindan, N. Ranganathan, D. Gulati, D. Burger, and S. W. Keckler, "Composable lightweight processors," in *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, 2007, pp. 381–394.

[15] W. Kim, D. Brooks, and G.-Y. Wei, "A fully-integrated 3-level dc-dc converter for nanosecond-scale dvfs," *IEEE Journal of Solid-State Circuits*, vol. 47, no. 1, pp. 206 –219, Jan. 2012.

[16] W. Kim, M. S. Gupta, G.-Y. Wei, and D. Brooks, "System level analysis of fast, per-core dvfs using on-chip switching regulators," in *Proc. of the 14th International Symposium on High-Performance Computer Architecture*, 2008, pp. 123–134.

[17] D. Koufaty, D. Reddy, and S. Hahn, "Bias scheduling in heterogeneous multi-core architectures," in *Proc. of the 5th European Conference on Computer Systems*, 2010, pp. 125–138.

[18] R. Krishnamurthy, H. Schmit, and L. Carley, "A low-power 16-bit multiplier-accumulator using series-regulated mixed swing techniques," in *Custom Integrated Circuits Conference, 1998. Proceedings of the IEEE 1998*, 1998, pp. 499 –502.

[19] R. Kumar, N. Jouppi, and D. Tullsen, "Conjoined-core chip multiprocessing," in *Proc. of the 37th Annual International Symposium on Microarchitecture*, 2004, pp. 195–206.

[20] R. Kumar, K. I. Farkas, N. P. Jouppi, P. Ranganathan, and D. M. Tullsen, "Single-ISA Heterogeneous Multi-Core Architectures: The Potential for Processor Power Reduction," in *Proc. of the 36th Annual International Symposium on Microarchitecture*, Dec. 2003, pp. 81–92.

[21] R. Kumar, D. M. Tullsen, and N. P. Jouppi, "Core architecture optimization for heterogeneous chip multiprocessors," in *Proc. of the 15th International Conference on Parallel Architectures and Compilation Techniques*, 2006, pp. 23–32.

[22] R. Kumar, D. M. Tullsen, P. Ranganathan, N. P. Jouppi, and K. I. Farkas, "Single-isa heterogeneous multi-core architectures for multithreaded workload performance," in *Proceedings of the 31st annual international symposium on Computer architecture*, 2004.

[23] Y. Kwon, C. Kim, S. Maeng, and J. Huh, "Virtualizing performance asymmetric multi-core systems," in *Proc. of the 38th Annual International Symposium on Computer Architecture*, 2011, pp. 45–56.

[24] T. N. Miller, X. Pan, R. Thomas, N. Sedaghati, and R. Teodorescu, "Booster: Reactive core acceleration for mitigating the effects of process variation and application imbalance in low-voltage chips," in *Proc. of the 18th International Symposium on High-Performance Computer Architecture*, vol. 0, 2012, pp. 1–12.

[25] G. Patsilaras, N. K. Choudhary, and J. Tuck, "Efficiently exploiting memory level parallelism on asymmetric coupled cores in the dark silicon era," *ACM Trans. Archit. Code Optim.*, vol. 8, no. 4, pp. 28:1–28:21, Jan. 2012.

[26] D. Ponomarev, G. Kucuk, and K. Ghose, "Reducing power requirements of instruction scheduling through dynamic allocation of multiple datapath resources," in *Proc. of the 34th Annual International Symposium on Microarchitecture*, Dec. 2001, pp. 90–101.

[27] K. K. Rangan, G.-Y. Wei, and D. Brooks, "Thread motion: fine-grained power management for multi-core systems," in *Proc. of the 36th Annual International Symposium on Computer Architecture*, 2009, pp. 302–313.

[28] L. Sheng, H. A. Jung, R. Strong, J.B.Brockman, D. Tullsen, and N. Jouppi, "Mcpat: An integrated power, area, and timing modeling framework for multicore and manycore architectures," in *Proc. of the 42nd Annual International Symposium on Microarchitecture*, 2009, pp. 469–480.

[29] J. Suh and M. Dubois, "Dynamic mips rate stabilization in out-of-order processors," in *Proc. of the 36th Annual International Symposium on Computer Architecture*, 2009, pp. 46–56.

[30] M. A. Suleman, O. Mutlu, M. K. Qureshi, and Y. N. Patt, "Accelerating critical section execution with asymmetric multi-core architectures," in *17th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2009, pp. 253–264.

[31] K. Van Craeynest, A. Jaleel, L. Eeckhout, P. Narvaez, and J. Emer, "Scheduling heterogeneous multi-cores through performance impact estimation (pie)," in *Proceedings of the 39th International Symposium on Computer Architecture*, ser. ISCA '12, 2012, pp. 213–224.

[32] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe, "Smarts: accelerating microarchitecture simulation via rigorous statistical sampling," in *Proceedings of the 30th annual international symposium on Computer architecture*, 2003, pp. 84–97.

[33] B. Xu and D. H. Albonesi, "Methodology for the analysis of dynamic application parallelism and its application to reconfigurable computing," vol. 3844, no. 1. SPIE, 1999, pp. 78–86.

2012 IEEE/ACM 45th Annual International Symposium on Microarchitecture

Control-Flow Decoupling

Rami Sheikh, James Tuck, Eric Rotenberg
Department of Electrical and Computer Engineering
North Carolina State University
{rmalshei, jtuck, ericro}@ncsu.edu

Abstract

Mobile and PC/server class processor companies continue to roll out flagship core microarchitectures that are faster than their predecessors. Meanwhile placing more cores on a chip coupled with constant supply voltage puts per-core energy consumption at a premium. Hence, the challenge is to find future microarchitecture optimizations that not only increase performance but also conserve energy. Eliminating branch mispredictions – which waste both time and energy – is valuable in this respect.

We first explore the control-flow landscape by characterizing mispredictions in four benchmark suites. We find that a third of mispredictions-per-1K-instructions (MPKI) come from what we call separable *branches: branches with large control-dependent regions (not suitable for if-conversion), whose backward slices do not depend on their control-dependent instructions or have only a short dependence. We propose control-flow decoupling (CFD) to eradicate mispredictions of separable branches. The idea is to separate the loop containing the branch into two loops: the first contains only the branch's predicate computation and the second contains the branch and its control-dependent instructions. The first loop communicates branch outcomes to the second loop through an architectural queue. Microarchitecturally, the queue resides in the fetch unit to drive timely, non-speculative fetching or skipping of successive dynamic instances of the control-dependent region.*

Either the programmer or compiler can transform a loop for CFD, and we evaluate both. On a microarchitecture configured similar to Intel's Sandy Bridge core, CFD increases performance by up to 43%, and reduces energy consumption by up to 41%. Moreover, for some applications, CFD is a necessary catalyst for future complexity-effective large-window architectures to tolerate memory latency.

1. Introduction

Good single-thread performance is important for both serial and parallel applications, and provides a degree of independence from fickle parallelism. This is why, even as the number of cores in a multi-core processor scales, processor companies continue to roll out flagship core microarchitectures that are faster than their predecessors. Meanwhile placing more cores on a chip coupled with stalled supply voltage scaling puts per-core energy consumption at a premium. Thus, the challenge is to find future microarchitecture optimizations that not only increase performance but also conserve energy.

Eliminating branch mispredictions is valuable in this respect. Mispredictions waste both time and energy, firstly, by fetching and executing wrong-path instructions and, secondly, by repairing state before resuming on the correct path. Figure 1a shows instructions-per-cycle (IPC) for several applications with hard-to-predict branches. The first bar is for our baseline core (refer to Table 3 in Section 5) with a state-of-art branch predictor (ISL-TAGE [28, 29]) and the second bar is for the same core with perfect branch prediction. Each application's branch misprediction rate is shown above its bars. Speedups with

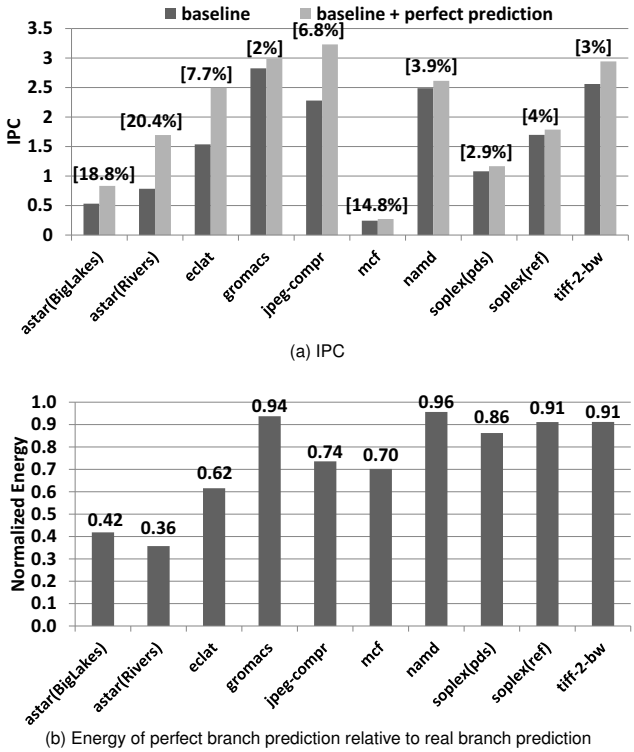


Figure 1: Impact of perfect branch prediction.

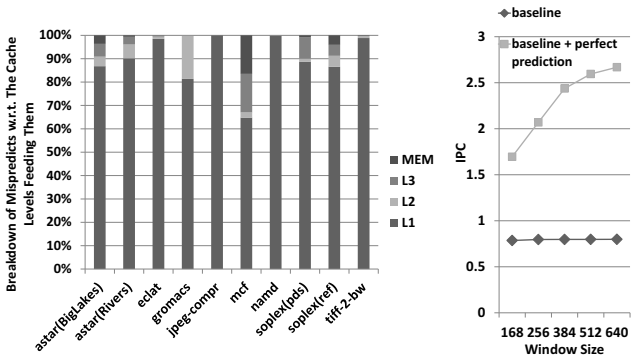
perfect branch prediction range from 1.05 to 2.16. Perfect branch prediction reduces energy consumption by 4% to 64% compared to real branch prediction (Figure 1b).

Some of these applications also suffer frequent last-level cache misses. Complexity-effective large-window processors can tolerate long-latency misses and exploit memory-level parallelism with small cycle-critical structures [31, 23]. Their ability to form an effective large window is degraded, however, when a mispredicted branch depends on one of the misses [31]. Figure 2a shows the breakdown of mispredicted branches that depend on data at various levels in the memory hierarchy: L1, L2, L3 and main memory. Figure 2b shows how the IPC of ASTAR (an application with high misprediction rate and significant fraction of mispredictions fed by L3 or main memory) scales with window size. Without perfect branch prediction, IPC does not scale with window size: miss-dependent branch mispredictions prevent a large window from performing its function of latency tolerance. Conversely, eradicating mispredictions acts as a catalyst for latency tolerance. IPC scales with window size in this case.

We first explore the current control-flow landscape by characterizing mispredictions in four benchmark suites using a state-of-art predictor. In particular, we classify the control-dependent regions guarded by hard-to-predict branches. About a third of mispredictions-per-1K-instructions (MPKI) come from branches with small control-

1072-4451/12 $26.00 © 2012 IEEE
DOI 10.1109/MICRO.2012.38

IEEE computer society



(a) Breakdown of mispredictions with respect to the furthest memory hierarchy level feeding them

(b) ASTAR(Rivers) IPC under different window sizes

Figure 2: Effect of branch mispredictions on memory latency tolerance.

dependent regions, e.g., hammocks. If-conversion using conditional moves, a commonly available predication primitive in commercial instruction-set architectures (ISA), is generally profitable for this class [2]. For completeness, we analyze why the gcc compiler did not if-convert such branches and manually do so at the source level in order to focus on other classes. We discover that another third of MPKI comes from what we call *separable* branches. A separable branch has two qualities:

1. The branch has a large control-dependent region, not suitable for if-conversion.
2. The branch does not depend on its own control-dependent instructions via a loop-carried data dependence (*totally separable*), or has only a short loop-carried dependence with its control-dependent instructions (*partially separable*).

For a totally separable branch, the branch's predicate computation is totally independent of the branch and its control-dependent region. This suggests "vectorizing" the control-flow: first generate a vector of predicates and then use this vector to drive fetching or skipping successive dynamic instances of the control-dependent region. This is the essence of our proposed technique, control-flow decoupling (CFD), for eradicating mispredictions of separable branches. The loop containing the branch is separated into two loops: a first loop contains only the instructions needed to compute the branch's predicate (generate branch outcomes) and a second loop contains the branch and its control-dependent instructions (consume branch outcomes). The first loop communicates branch outcomes to the second loop through an architectural queue, specified in the ISA and managed by push and pop instructions. At the microarchitecture level, the queue resides in the fetch unit to facilitate timely, non-speculative branching.

Partially separable branches can also be handled. In this case, the branch's predicate computation depends on some of its control-dependent instructions. This means a copy of the branch and the specific control-dependent instructions must be included in the first loop. Fortunately, this copy of the branch can be profitably removed by if-conversion due to few control-dependent instructions.

Either the programmer or compiler can transform a loop for CFD, and we evaluate both. On a microarchitecture configured similar to Intel's Sandy Bridge core [35], CFD increases performance by up to 43%, and reduces energy consumption by up to 41%. For hard-to-predict branches that traverse large data structures that suffer many cache misses, CFD acts as the necessary catalyst for future large-window architectures to tolerate these misses.

The paper is organized as follows. In Section 2, we discuss our methodology and classification of control-flow in a wide range of applications. In Section 3, we present the ISA, hardware and software aspects of CFD. In Section 4, we describe our implementation of CFD in the gcc compiler. In Section 5, we describe our evaluation framework and baseline selection process. In Section 6, we present an evaluation of the proposed techniques. In Section 7, we discuss prior related work. We conclude the paper in Section 8.

2. Methodology and Control-Flow Classification

The goal of the control-flow classification is first and foremost discovery: to gain insight into the nature of difficult branches' control-dependent regions, as this factor influences the solutions that will be needed, both old and new. Accordingly we cast a wide net to expose as many control-flow idioms as possible: (1) we use four benchmark suites comprised of over 80 applications, and (2) for the purposes of this comprehensive branch study, each application is run to completion leveraging a PIN-based branch profiling tool.

2.1. Methodology

We use four benchmark suites: *SPEC2006* [32] (engineering, scientific, and other workstation type benchmarks), *NU-MineBench-3.0* [24] (data mining), *BioBench* [1] (bioinformatics), and *cBench-1.1* [10] (embedded). All benchmarks[1] are compiled for x86 using gcc with optimization level *-O3* and run to completion using PIN [20]. We wrote a pintool that instantiates a state-of-art branch predictor (winner of CBP3, the third Championship Branch Prediction: 64KB ISL-TAGE [28]) that is used to collect detailed information for every static branch.

Different benchmarks have different dynamic instruction counts. In the misprediction contribution pie charts that follow, we weigh each benchmark equally by using its MPKI instead of its total number of mispredictions. Effectively we consider the average one-thousand-instruction interval of each benchmark.

Figure 3a shows the relative misprediction contributions of the four benchmark suites. Every benchmark of every suite is included[2], and, as just mentioned, each benchmark is allocated a slice proportional to its MPKI. We further refine the breakdown of each benchmark suite slice into *targeted* versus *excluded*, shown in Figure 3b. The excluded slice contains (1) benchmarks with misprediction rates less than 2%, and (2) benchmarks that we could not run in our detailed timing simulator introduced later (due to gcc Alpha cross-compiler problems). The targeted slice contains the remaining benchmarks. Table 1 lists the targeted benchmarks along with their MPKIs.

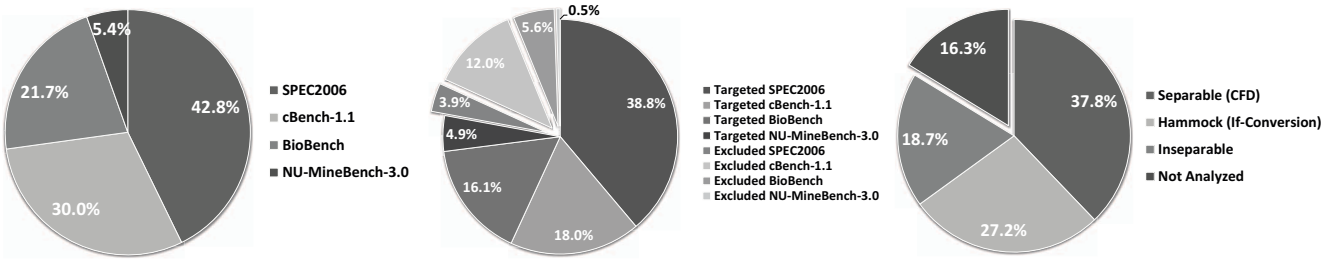
This paper focuses on the targeted slices which, according to Figure 3b, contribute almost 78% of cumulative MPKI in the four benchmark suites.

2.2. Control-Flow Classification

We inspected branches in the targeted benchmarks, and categorized them into the following four classes:

[1] For benchmarks with multiple ref inputs, we profiled then classified all inputs into groups based on the control-flow patterns exposed. One input is selected from each group in order to cover all observed patterns. For example, for *bzip* we select the ref inputs *input.source* and *chicken*.

[2] A benchmark that is present in multiple suites is included once. For example, *hmmer* appears in *BioBench* and *SPEC2006*. In both benchmark suites, the same hard-to-predict branches are exposed, thus, only one instance of *hmmer* is included.



(a) Mispredictions per benchmark suite (b) Targeted vs. excluded mispredictions (c) Breakdown of targeted mispredictions

Figure 3: Breakdown of branch mispredictions.

Benchmark Suite	Application	MPKI	Benchmark Suite	Application	MPKI
SPEC2006	astar (BigLakes)	10.11	cBench	gsm	2.10
	astar (Rivers)	25.98		jpeg-compr	8.17
	bzip2 (chicken)	4.08		jpeg-decompr	2.41
	bzip2 (input.source)	8.16		quick-sort	4.64
	gobmk	7.17		tiff-2-bw	5.42
	gromacs	1.13		tiff-median	3.60
	hmmer	11.72			
	mcf	9.06	BioBench	clustalw	4.25
	namd	1.17		fasta	16.64
	sjeng	5.15			
	soplex (pds)	6.14	MineBench	eclat	10.19
	soplex (ref)	2.25			

Table 1: Targeted applications.

```
1  if (sc > ic[k]) ic[k] = sc;
        ⇩
2  local = ic[k];
3  if(sc > local)  local = sc;
4  ic[k] = local;
        ⇩
5  local = ic[k];
6  CMOV(local, sc, sc > local);
7  ic[k] = local;
```

Figure 4: Hammock (from *HMMER*).

1. *Hammock*: Branches with small, simple control-dependent regions. Such branches will be if-converted. From what we can tell, the gcc compiler did not if-convert these branches because they guard stores. An example from *hmmer* is shown in Figure 4, line 1. To encourage if-conversion, the code can be adjusted (manually or using compiler) to unconditionally perform the store, if legal (i.e., if address is legal regardless of branch outcome). The control-dependent store to *ic[k]* (line 1) is moved outside the hammock (line 4) and the value being stored is a new local variable, *local*. Depending on the branch, *local* contains either the original value of *ic[k]* (line 2) or *sc* (line 3). Thus, the store to *ic[k]*, after the hammock, is effectively conditional – *ic[k]*'s value may or may not change – even though it is performed unconditionally. The new if-statement (line 3) is then if-converted by the compiler using a conditional move (line 6): conditionally move *sc* into *local* based on the condition *sc* > *local*. This transformation increases the number of retired stores, but the extra stores are silent. Obtaining the original value at the memory location requires a load, but we observed that most cases are like the *hmmer* example, in which the load already exists because the branch's test depends on a reference to *ic[k]* (line 1).
2. *Separable*: Branches with large, complex control-dependent regions, where the branch's backward slice (predicate computation) is either *totally separable* or *partially separable* from the branch and its control-dependent instructions. The backward slice is *totally separable* if it does not contain any of the branch's control-dependent instructions. Total separability allows all iterations of the backward slice to be hoisted outside the loop containing the branch, conceptually vectorizing the predicate computation, which is what CFD does via its first and second loops. The backward slice is *partially separable* if it contains very few of the branch's control-dependent instructions. In this case, the backward slice also contains the branch itself, since the branch guards the few control-dependent instructions in the slice. All iterations of the backward slice can still be hoisted but it contains a copy of the branch, therefore, the backward slice is if-converted. CFD will be applied to totally and partially separable branches.
3. *Inseparable*: Branches with large, complex control-dependent regions, where the branch's backward slice contains too many of the branch's control-dependent instructions. An *inseparable* branch differs from a *partially separable* branch, in that it is not profitable to if-convert its backward slice. This type of branch is very serial in nature: the branch is frequently mispredicted and it depends on many of the instructions that it guards. This class of branch cannot be handled by if-conversion or CFD, and will require a new solution which is outside the scope of this paper.
4. *Not Analyzed*: Branches we did not analyze, i.e., branches with small contributions to total mispredictions.

Figure 3c breaks down the *targeted* mispredictions of Figure 3b into these four classes. 37.8% of the targeted mispredictions can be handled using CFD. 27.2% of the targeted mispredictions can be handled using if-conversion. That CFD covers the largest percentage of MPKI after applying a sophisticated branch predictor, provides a compelling case for CFD software, architecture, and microarchitecture support. Its applicability is on par with if-conversion, a commercially mainstream technique that also combines software, architecture, and microarchitecture. In addition to comparable MPKI coverage, CFD and if-conversion apply to comparable numbers of benchmarks and static branches (see Table 5 in Section 6).

3. Control-Flow Decoupling

Figure 5a shows a high-level view of a totally separable branch within a loop. *Branch slice* computes the branch's predicate. Depending on the predicate, the branch is taken or not-taken, causing its control-dependent instructions to be skipped or executed, respectively. In this example, none of the branch's control-dependent instructions are in its backward slice, i.e., there isn't a loop-carried data dependency between any of the control-dependent instructions and the branch. A partially separable branch would look similar, except a small number of its control-dependent instructions would be in the branch slice; this would appear as a backward dataflow edge from these instructions to the branch slice.

Figure 5b shows the loop transformed for CFD. The loop is separated into two loops, each with the same trip-count as the original. The first loop has just the branch slice. It pushes predicates onto an

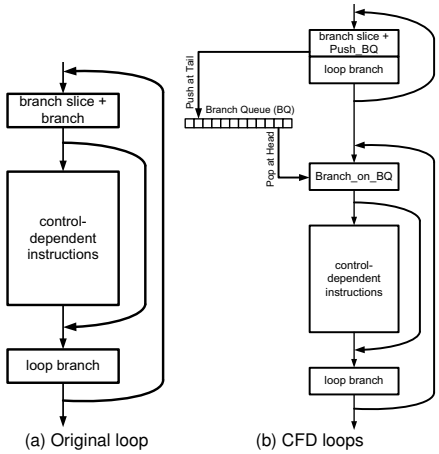


Figure 5: High-level view of the CFD transformation.

architectural *branch queue* (BQ) using a new instruction, Push_BQ. The second loop has the control-dependent instructions. They are guarded by a new instruction, Branch_on_BQ. This instruction pops predicates from BQ and the predicates control whether or not the branch is taken.

Hoisting all iterations of the branch slice creates sufficient *fetch separation* between a dynamic instance of the branch and its producer instruction, ensuring that the producer executes before the branch is fetched. If successive iterations are a, b, c, ..., instead of fetching *slice-a, branch-a, slice-b, branch-b, slice-c, branch-c, ...*, the processor fetches *slice-a, slice-b, slice-c, ... branch-a, branch-b, branch-c, ...* Additionally, to actually exploit the now timely predicates, they must be communicated to the branch in the fetch stage of the pipeline so that the branch can be resolved at that time. Communicating through the existing source registers would not resolve the branch in the fetch stage. This is why we architect the BQ predicate communication medium and why, microarchitecturally, it resides in the fetch unit.

While this paper assumes an OOO processor for evaluation purposes, please note that in-order and OOO processors both suffer branch penalties due to the fetch-to-execute delay of branches. We want to resolve branches in the fetch stage (so fetching is not disrupted) but they resolve in the execute stage, unless correctly predicted. Thus, the problem with branches stems from pipelining in general. OOO execution merely increases the pipeline's speculation depth (via buffering in the scheduler) so that, far from being a solution to the branch problem, OOO execution actually makes the branch problem more acute.

For a partially separable branch, the first loop would not only have (1) the branch slice and Push_BQ instruction, but also (2) the branch and just those control-dependent instructions that feed back to the branch slice. The branch is then removed by if-conversion, using conditional moves to predicate the control-dependent instructions. CFD is still profitable in this case because the subsetted control-dependent region is small and simple (otherwise the branch would be classed as inseparable).

CFD is a software-hardware collaboration. The following subsections discuss ISA, software, and hardware.

3.1. ISA Support and Benchmark Example

ISA support includes an architectural specification of the BQ and two instructions, Push_BQ and Branch_on_BQ. The architectural specification of the BQ is as follows:

1. The BQ has a specific size. BQ size has implications for software. These are discussed in the next subsection.
2. Each BQ entry contains a single flag indicating taken/not-taken (the predicate). Other microarchitectural state may be included in each entry of the BQ's physical counterpart, but this state is transparent to software and not specified in the ISA.
3. A length register indicates the BQ occupancy. Architecting only a length register has the advantage of leaving low-level management concerns to the microarchitect. For example, the BQ could be implemented as a circular or shifting buffer. Thus, at the ISA level, the BQ head and tail are conceptual and are not specified as architectural registers: their physical counterparts are implementation-dependent.
4. The ISA provides mechanisms to save and restore the BQ state (queue contents and length register) to memory. This is required for context-switches. We recommend the approach used in some commercial ISAs, which is to include the BQ among the special-purpose registers and leverage move-from and move-to special-purpose-register instructions to transfer the BQ state to and from general-purpose registers (which can be saved and restored via stores and loads, respectively). If this is not possible, then dedicated Save_BQ and Restore_BQ instructions could be used.

The Push_BQ instruction has a single source register specifier to reference a general-purpose register. If the register contains zero (non-zero), Push_BQ pushes a 0 (1). Branch_on_BQ is a new conditional branch instruction. Branch_on_BQ specifies its taken-target like other conditional branches, via a PC-relative offset. It does not have any explicit source register specifiers, however. Instead, it pops its predicate from the BQ and branches or doesn't branch, accordingly.

The ISA specifies key ordering rules for pushes and pops, that software must abide by. First, a push must precede its corresponding pop. Second, N consecutive pushes must be followed by exactly N consecutive pops in the same order as their corresponding pushes. Third, N cannot exceed the BQ size.

Figure 6 shows a real example from the benchmark *SOPLEX*. Referring to the original code: The loop compares each element of array *test[]* to variable *theeps*. The hard-to-predict branch is at line 3 and its control-dependent instructions are at lines 4-9. Neither the array nor the variable is updated inside the control-dependent region, thus, this is a totally separable branch. This branch contributes 31% of the benchmark's mispredictions (for *ref* input).

Decoupling the loop is fairly straightforward. The first loop computes predicates (lines 2-3) and pushes them onto the BQ (line 4). The second loop pops predicates from the BQ and conditionally executes the control-dependent instructions, accordingly (line 7).

An ISA enhancement must be carefully specified, so that its future obsolescence does not impede microarchitects of future generation processors. Accordingly, CFD is architected as an optional and scalable co-processor extension:

1. Optional: Inspired by configurability of co-processors in the MIPS ISA – which specifies optional co-processors 1 (floating-point unit) and higher (accelerators) – BQ state and instructions can be encapsulated as an optional co-processor ISA extension. Thus, future implementations are not bound by the new BQ co-processor ISA. Codes compiled for CFD must be recompiled for processors that do not implement the BQ co-processor ISA, but this is no different than the precedent set by MIPS' flexible co-processor specification.
2. Scalable: The BQ co-processor ISA can specify a BQ size of

```
    Original Loop
1   for ( … ) {
2     x = test[i];
3     if (x < -theeps) {                    // hard-to-predict branch
4       x *= x / penalty_ptr[i];
5       x *= p[i];
6       if (x > best) {                     // predictable branch
7         best = x;
8         selId = thesolver->id(i);
9       }
10    }
11  }
    Decoupled Loops
      First Loop
1   for ( … ) {
2     x = test[i];
3     pred = (x < -theeps);                 // the predicate is computed
4     Push_BQ(pred);                        // then pushed onto the BQ
5   }
      Second Loop
6   for ( … ) {
7     Branch_on_BQ{                         // pop the predicate
8       x = test[i];
9       x *= x / penalty_ptr[i];
10      x *= p[i];
11      if (x > best) {
12        best = x;
13        selId = thesolver->id(i);
14      }
15    }
16  }
```

Figure 6: SOPLEX' source code.

N: a machine-dependent parameter, thus allowing scalability to different processor window sizes.

3.2. Software Side

For efficiency, the trip-counts of the first and second loops should not exceed the BQ size. This is a matter of performance, not correctness, because software can choose to spill/fill the BQ to/from memory. In practice, this is an important issue because many of the CFD-class loops iterate thousands of times whereas we specify a BQ size of 128 in this paper.

We explored multiple solutions but the most straightforward one is loop strip mining. The original loop is converted to a doubly-nested loop. The inner loop is similar to the original loop but its trip-count is bounded by the BQ size. The outer loop iterates a sufficient number of times to emulate the original loop's trip-count. Then, CFD is applied to the inner loop.

Decoupling the loop can be done either manually by the programmer or automatically by the compiler. In this paper, CFD was initially applied manually which was a fairly easy task. In Section 4, we describe automating CFD in the gcc compiler and in Section 6 we evaluate how well it compares to the manual implementation.

3.3. Hardware Side

This subsection describes microarchitecture support for CFD. The BQ naturally resides in the instruction fetch unit. In our design, the BQ is implemented as a circular buffer. In addition to the software-visible predicate bit, each BQ entry has the following microarchitectural state: pushed bit, popped bit, and checkpoint id. For a correctly written program, a Push_BQ (push) instruction is guaranteed to be fetched before its corresponding Branch_on_BQ (pop) instruction. Because of pipelining, however, the push might not execute before the pop is fetched, referred to as a late push. The pushed bit and popped bit enable synchronizing the push and pop. We explain BQ operation separately for the two possible scenarios: early push and late push.

3.3.1. Early Push. The early push scenario is depicted in Figure 7, left-hand side.

When the push instruction is fetched, it is allocated the entry at the BQ tail. It initializes its entry by clearing the pushed and popped bits. The push instruction keeps its BQ index with it as it flows down the pipeline [3]. When the push finally executes, it checks the popped bit in its BQ entry. It sees that the popped bit is still unset. This means the scenario is early push, i.e., the push executed before its pop counterpart was fetched. Accordingly, the push writes the predicate into its BQ entry and sets the pushed bit to signal this fact.

Later, the pop instruction is fetched. It is allocated the entry at the BQ head, which by the ISA ordering rules must be the same entry as its push counterpart. It checks the pushed bit. It sees that the pushed bit is set, therefore, it knows to use the predicate that was pushed earlier. The pop executes right away, either branching or not branching according to the predicate.

3.3.2. Late Push. The late push scenario is depicted in Figure 7, right-hand side.

In this scenario, the pop is fetched before the push executes. As before, when the pop is fetched, it checks the pushed bit to see if the push executed. In this case the pushed bit is still unset so the pop knows that a predicate is not available. There are two options: (1) stall the fetch unit until the push executes, or (2) predict the predicate using the branch predictor. Our design implements option 2 which we call a *speculative pop*. When the speculative pop reaches the rename stage, a checkpoint is taken. (This is on top of the baseline core's branch checkpointing policy, which we thoroughly explore in Section 5.) Unlike conventional branches, the speculative pop cannot confirm its prediction – this task rests with the late push instruction. Therefore, the speculative pop writes its predicted predicate and checkpoint id into its BQ entry, and signals this fact by setting the popped bit. This information will be referenced by the late push to confirm/disconfirm the prediction and initiate recovery if needed.

When the push finally executes, it notices that the popped bit is set in its BQ entry, signifying a late push. The push compares its predicate with the predicted one in the BQ entry. If they don't match, the push initiates recovery actions using the checkpoint id that was placed there by the speculative pop. Finally, the push writes the predicate into its BQ entry and sets the pushed bit.

Empirically, late pushes are very rare in our CFD-modified benchmarks, less than 0.1% of pops (one per thousand). When fully utilized by software, a 128-entry BQ separates a push and its corresponding pop by 127 intervening pushes. This typically corresponds to a push/pop separation of several hundreds of instructions, providing ample time for a push to execute before its pop counterpart is fetched.

3.3.3. BQ Length. The BQ length (occupancy) is the sum of two components:

1. net_push_ctr: This is the net difference between the number of pushes and pops retired from the core up to this point in the program's execution. The ISA push/pop ordering rules guarantee this count will always be greater than or equal to zero and less than or equal to BQ size. This counter is incremented when a push retires and decremented when a pop retires.
2. pending_push_ctr: This is the number of pushes in-flight in the window, i.e., the number of fetched but not yet retired pushes. It is incremented when a push is fetched, decremented when a push is retired (because it now counts against net_push_ctr), and possibly adjusted when a mispredicted branch resolves (see next section).

BQ length must be tracked in order to detect the BQ stall condition. In particular, if BQ length is equal to BQ size and the fetch unit

[3] Having the BQ index in the push instruction's payload enables it to reference its BQ entry later, when it executes OOO. This is a standard technique for managing microarchitecture FIFOs such as the reorder buffer and load and store queues.

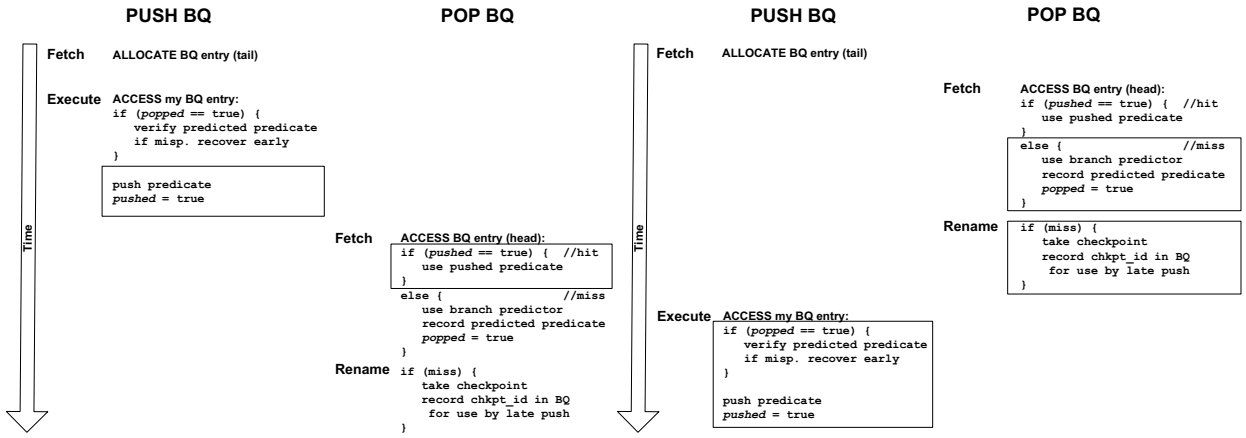


Figure 7: BQ operation. Two scenarios are shown: early push (left, common) and late push (right, uncommon).

fetches a push instruction, the fetch unit must stall. Note that the stall condition is guaranteed to pass for a bug-free program. The ISA push/pop ordering rules guarantee that there are *BQ size* in-flight pop instructions prior to the stalled push. The first one of these pops to retire will unstall the stalled push.

3.3.4. BQ Recovery. The core may need to roll back to a branch checkpoint, in the case of a mispredicted branch, or the committed state, in the case of an exception. In either case, the BQ itself needs to be repaired.

1. Preparing for misprediction recovery: Each branch checkpoint is augmented with state needed to restore the BQ to that point in the program execution. Namely, in addition to the usual checkpointed state (Rename Map Table, etc.), each checkpoint also takes a snapshot of the BQ head and tail pointers. This is a modest amount of state compared to other checkpointed state.
2. Preparing for exception recovery: Exception recovery requires maintaining committed versions of the BQ head and tail pointers, called arch_head and arch_tail. Arch_head and arch_tail are incremented when pops and pushes retire, respectively.

When there is a roll-back, the BQ head and tail pointers are restored from the referenced checkpoint (on a misprediction) or their committed versions (on an exception), and all popped bits between the restored head and tail are cleared. Moreover, pending_push_ctr (the second component of BQ length) is reduced by the number of entries between the tail pointers before and after recovery (this corresponds to the number of squashed push instructions).

3.3.5. Branch Target Buffer. Like all other branch types, Branch_on_BQ is cached in the fetch unit's Branch Target Buffer (BTB) so that there is no penalty for a taken Branch_on_BQ as long as the BTB hits. The BTB's role is to detect branches and provide their taken-targets, in the same cycle that they are being fetched from the instruction cache. This information is combined with the taken/not-taken prediction (normal conditional branch) or the popped predicate (Branch_on_BQ) to select either the sequential or taken target. As with other branches, a BTB miss for a taken Branch_on_BQ results in a 1-cycle misfetch penalty (detected in next cycle).

Predicates for potential Branch_on_BQ instructions in the current fetch bundle are obtained from the BQ in parallel with the BTB access, because these predicates are always at consecutive entries starting at the BQ head.

3.4. Optimization

This section describes an optimization on top of CFD, that can reduce CFD instruction overheads in some cases. We observed that values used to compute the predicate in the first loop are used again, thus recomputed, inside the control-dependent region in the second loop. A simple way to avoid duplication is to communicate values from the first loop to the second loop using an architectural value queue (VQ) and VQ push/pop instructions. We call this optimization CFD+.

An interesting trick to leverage existing instruction issue and register communication machinery in a superscalar core, is to map the architectural value queue onto the physical register file. This is facilitated by the *VQ renamer* in the rename stage. The VQ renamer is a circular buffer with head and tail pointers. Its entries contain physical register mappings instead of values. The mappings indicate where the values are in the physical register file. A VQ push is allocated a destination physical register from the freelist. Its mapping is pushed at the tail of the VQ renamer. A VQ pop references the head of the VQ renamer to obtain its source physical register mapping. The queue semantics ensure the pop links to its corresponding push through its mapping. In this way, after renaming, VQ pushes and pops synchronize in the issue queue and communicate values in the execution lanes the same way as other producer-consumer pairs. The physical registers allocated to push instructions are freed when the pops that reference them retire.

4. CFD Compiler Implementation

We have implemented a compiler pass to perform the CFD code transformation automatically. The pass needs a list of hard-to-predict branches derived from profiling or the programmer as input, and it transforms the inner-loop containing the branch into CFD form. We will refer to the decoupled first and second loops created by the compiler pass as the *Producer* and *Consumer* loops, respectively. Algorithm 1 shows the overall CFD compiler implementation. We start with the CFD function which takes as input a loop and the hard-to-predict predicates that are contained within the loop.

The first steps of the algorithm are inspired by the Decoupled Software Pipelining (DSWP) algorithm presented by Ottoni *et al.* [25]. In particular, we borrow their strategy of first constructing a full Program Dependence Graph (PDG) and then consolidating the strongly connected components (SCCs) into single nodes in the graph to create a directed acyclic graph (Lines 2-3). If the hard-to-predict branch forms the root of a control-dependent region which can be

Algorithm 1 Overall CFD algorithm.

```
1: function CFD(Loop l, Predicate p)
2:     pdg ← BuildPDG(l)
3:     dag ← ConsolidateSCCs(pdg)
4:     MarkPredicateSlices(dag,p)
5:     AssignStmtsToLoops(dag)
6:     if non-empty CFD region found in dag then
7:         producer ← l
8:         consumer ← CloneLoop(l)
9:         ConnectLoops(producer,consumer)
10:        for all control flow decoupled branches, b do
11:            Insert Push_BQ(Predicate(b)) in producer just before b
12:            Replace b in consumer with Branch_on_BQ
13:        end for
14:        for all r, def'ed in producer and used in consumer do
15:            Insert push in producer at definition of r
16:            Replace definition of r in consumer with "r=Pop_VQ()"
17:        end for
18:        Remove code from producer assigned only to consumer
19:        Remove code from consumer assigned only to producer
20:        Final Dead and Redundant Code Elimination
21:    end if
22: end function
```

isolated into one or more SCCs, then the branch is separable. We use the consolidated graph to assign nodes to the Producer and Consumer loops. Otherwise, if the control-dependent code is part of the same SCC as the loop's exit condition, then no decoupling may be possible (and our algorithm gives up).

In Line 4, we call the MarkPredicateSlices subroutine which carries out the following operations. For each predicate in the loop, it finds its corresponding node in the *dag*. All nodes in its forward slice (all immediate successors and those reached through a depth-first search (DFS)) are marked as belonging to the Consumer. All nodes in its backward slice and itself (all immediate predecessors and those reached through a reverse DFS) are marked as belonging to the Producer.

At this point, some nodes in *dag* have been scheduled among the Producer and Consumer loops, but many nodes may remain unscheduled. For example, any node that is both control independent and data independent from marked nodes will need to be assigned a loop. AssignStmtsToLoops, on line 5, completes the task of scheduling.

AssignStmtsToLoops. This function simultaneously solves several problems. First, it must create a correct schedule. A statement must be placed in the Producer if any dependent instruction has already been placed in the Producer. Similarly, a statement must be placed in the Consumer if any statement it depends upon has already been placed in the Consumer. These rules must always be enforced. Fortunately, some flexibility does exist that can be leveraged for optimization. For example, if a statement produces no side-effects, then we can optionally schedule it in both loops. This flexibility allows to choose between replicating work and communicating values depending on which is more efficient. We use a simple heuristic to solve both problems at once as shown in Algorithm 2.

In lines 2-7, each node is initially marked as NoReplicate to mean that it must be scheduled in only one loop. Next, we figure out if the node has any side-effects (stores or function calls) which would prevent replication, and if it does not, it is marked MaybeReplicate to mean that we can possibly schedule it in both loops.

The second for-all loop visits all nodes in topological order, which means we must visit a node's predecessors in an earlier iteration. This makes it easy to reason about predecessors since they have already been processed.

In lines 9-15, we assign a node to a loop if it was not assigned one in MarkPredicateSlices. Note, we prefer to place a node in the Producer unless forced to place it in the Consumer. Once we know where the node will be scheduled, we need to determine if it is better to communicate or replicate any values it produces for the Consumer loop. Lines 16-23 form this judgement. First, we check to make sure that the node is in the Producer and that it is marked MaybeReplicate. To determine if we should replicate, we compare the estimated runtime cost (EstCost) of the node against a minimal threshold that determines when communication will be cheaper. If the node is expensive to execute, we mark the node as NoReplicate which means that any register it defines must be communicated to the Consumer loop. Otherwise, we mark the node as Replicate and it will be computed in both loops. For all of our results, we use CommThreshold=2.

Algorithm 2 Assign all statements to the Producer and/or Consumer loops.

```
1: function AssignStmtsToLoops(PDG dag)
2:     for all n ∈ dag do
3:         Mark n as NoReplicate
4:         if n has no side-effects then
5:             Mark n as MaybeReplicate
6:         end if
7:     end for
8:     for all n ∈ dag, in topological order do
9:         if n has not been placed in a loop then
10:            if any predecessor of n is in the Consumer then
11:                place n in the Consumer
12:            else
13:                place n in the Producer
14:            end if
15:        end if
16:        if n placed in Producer and n marked MaybeReplicate then
17:            if n communicates to Consumer
18:                and EstCost(n) > CommThreshold then
19:                n marked NoReplicate (values will be communicated)
20:            else
21:                n marked Replicate
22:            end if
23:        end if
24:    end for
25: end function
```

Final Code Generation. If a non-empty CFD region is found (line 6 of Algorithm 1), we finalize the loops and generate the code. This process is shown in lines 7-20. First, we clone the loop (line 8) and use the original as the Producer and the clone as the Consumer. Next, we connect the loops so that the program will first execute the Producer and then the Consumer. This entails redirecting the exits of the Producer to the pre-header of the Consumer. The Consumer's exits, since they are a clone of the original loop, remain unchanged. Also, our implementation works on an SSA graph, so we also fix the phi-nodes at the pre-header of the Consumer loop and exits of the Consumer loop. Also while connecting the loops, we perform the loop strip mining transformation described in Section 3.2. This is easily accomplished by inserting a new outer loop to surround both the Producer and Consumer and by forcing a break from each loop every *BQ size* iterations. Early breaks/returns are handled by keeping a loop count in the Producer and passing that count to the Consumer for it to use as its trip count.

Next, we insert the necessary predicate and value communication. In lines 10-13, we visit all predicates that are computed in the Producer loop and communicated to the Consumer loop. We insert a Push_BQ in the Producer and place a Branch_on_BQ in the Consumer in place of the original branch. This loop will always handle the original hard-to-predict predicates but additional predicates may also be included if the partitioning algorithm places a predicate in the Producer that has control-dependent instructions in the Consumer. Ideally, the partitioning algorithm should limit the frequency of this case.

In lines 14-17, we insert value communication between the pro-

	AMD Bobcat	ARM Cortex A15	IBM Power6	INTEL Pentium 4
Fetch-to-Execute	13	15	13	20

Table 2: Minimum fetch-to-execute latency in cycles.

ducer and consumer loops. Any register that is defined in the Producer and used in the Consumer must be communicated. The push is placed at the definition in the Producer and the pop is placed at the same point (the cloned register definition) in the Consumer.

Finally, the Producer and Consumer code is cleaned up. All instructions assigned the Consumer partition are removed from the Producer loop and vice versa for the Consumer loop. Then, a final dead and redundant code elimination pass eliminates other inefficiencies, like empty basic blocks and useless control-flow paths.

5. Evaluation Environment

The microarchitecture presented in Section 3 is faithfully modeled in a detailed execution-driven, execute-at-execute, cycle-level simulator. The simulator runs Alpha ISA binaries. Recall, in Section 2, we used x86 binaries to locate hard-to-predict (easy-to-predict) branches, owing to our use of PIN. Our collected data confirms that hard-to-predict (easy-to-predict) branches in x86 binaries are hard-to-predict (easy-to-predict) in Alpha binaries. The predictability is influenced far more by program structure than the ISA that it gets mapped to.

Section 2 described the four benchmark suites used. All benchmarks are compiled to the Alpha ISA using gcc with -O3 level optimization. (We built gcc from scratch using the trunk SVN repository in the gcc-4 development line.) When applied, if-conversion and CFD modify the benchmark source. The modified benchmarks are verified by compiling natively to the x86 host, running them to completion, and verifying outputs (software queues are used to emulate the CFD queues).

Energy is measured using McPAT [19], which we augmented with energy accounting for the BQ (CFD, CFD+) and VQ (CFD+). Per-access energy for the BQ and VQ is obtained from CACTI tagless rams, and every read/write access is tracked during execution.

The parameters of our baseline core are configured as close as possible to those of Intel's Sandy Bridge core [35]. The baseline core uses the state-of-art ISL-TAGE predictor [28]. Additionally, in an effort to find the best-performing baseline, we explored the design space of misprediction recovery policies, including checkpoint policies (in-order vs. OoO reclamation, with confidence estimator [13] versus without) and number of checkpoints (from 0 to 64). We confirmed that: (1) An aggressive policy (OoO reclamation, confidence-guided checkpointing) performs best. (2) The harmonic mean IPC, across all applications of all workloads, levels off at 8 checkpoints.

The fetch-to-execute pipeline depth is a critical parameter as it factors into the branch misprediction penalty. Table 2 shows the minimum fetch-to-execute latency (number of cycles) for modern processors from different vendors. The latency ranges from 13 to 20 cycles [6, 17, 18, 7]. We conservatively use 10 cycles for this parameter. We also perform a sensitivity study with this parameter in Section 6.1.1.

Table 3 shows the baseline core configuration. The checkpoint management policy and number of checkpoints remain unchanged throughout our evaluation, even for studies that scale other window resources.

6. Results and Analysis

To evaluate the impact of our work on the top contributors of branch mispredictions in the targeted applications, we identify the regions to

Branch Prediction	**BP**: 64KB ISL-TAGE predictor - 16 tables: 1 bimodal, 15 partially-tagged. In addition to, IUM, SC, LP. - History lengths: {0, 3, 8, 12, 17, 33, 35, 67, 97, 138, 195, 330, 517, 1193, 1741, 1930} **BTB**: 4K entries, 4-way set-associative **RAS**: 64 entries
Memory Hierarchy	**Block size**: 64B **Victim caches**: each cache has a 16-entry FA victim cache **L1**: split, 64KB each, 4-way set-associative, 1-cycle access latency **L2**: unified, private for each core, 512KB, 8-way set-associative, 20-cycle access latency - **L2 stream prefetcher**: 4 streams, each of depth 16 **L3**: unified, shared among cores, 8MB, 16-way set-associative, 40-cycle access latency **Memory**: 200-cycle access latency
Fetch/Issue/Retire Width	4 instr./cycle
ROB/IQ/LDQ/STQ	168/54/64/36 (modeled after Sandy Bridge)
Fetch-to-Execute Latency	10-cycle
Physical RF	236
Checkpoints	8, OoO reclamation, confidence estimator (8K entries, 4-bit resetting counter, gshare index)
CFD	• **BQ**: 96B (128 6-bit entries) • **VQ renamer**: 128B (128 8-bit entries)

Table 3: Baseline core configuration.

Application	**Skip (B)**	**Overhead**		
		CFD	**CFD+**	**Compiler (CFD+)**
astar(BigLakes)	11.61	1.86	-	-
astar(Rivers)	0.53	1.81	-	-
eclat	7.10	1.28	1.12	1.14
gromacs	0.74	1.03	1.02	-
jpeg-compr	0.00	1.08	1.06	1.09
mcf	0.70	1.15	1.14	1.20
namd	2.17	1.01	-	-
soplex(pds)	9.94	1.02	1.02	1.04
soplex(ref)	49.25	0.90	-	1.41
tiff-2-bw	0.00	1.00	-	1.00
Application	**Skip (B)**	**If-Conversion**		
clustalw	0.04	1.0		
fasta	0.00	1.0		
gsm	0.00	1.03		
hmmer	0.02	1.0		
jpeg-decompr	0.00	1.0		
quick-sort	0.19	1.06		
sjeng	0.17	1.02		

Table 4: Application skip distances and overheads.

be simulated as follows. Given the set of top mispredicting branches and the functions in which they reside, we fast-forward to the first occurrence of the first encountered function of interest, warm up for 10M retired instructions, and then simulate for a certain number of retired instructions. When simulating the unmodified binary for the baseline, we simulate 100M retired instructions. When simulating binaries modified for CFD or if-conversion, we simulate as many retired instructions as needed in order to perform the same amount of work as 100M retired instructions of the unmodified binary. Table 4 shows the fast-forward (skip) distances of the applications and the overheads incurred by the modified binaries. Overhead is the factor by which retired instruction count increases (e.g., 1.5 means 1.5 times) for the same simulated region. In all cases except SOPLEX(ref), the modified binaries are simulated for more than 100M retired instructions[4]. Speedup is calculated as: $cycles_{baseline}/cycles_{CFD}$, where $cycles_{baseline}$ is the number of cycles to simulate 100M instructions of the unmodified binary and $cycles_{CFD}$ is the number of cycles to simulate overhead_factor x 100M instructions of the CFD-modified binary which corresponds to the same simulated region.

Table 5 shows detailed information about the modified source code, most importantly: (1) the affected branches and (2) the fraction of time spent in the functions containing these branches, as found by gprof-monitored native execution[5].

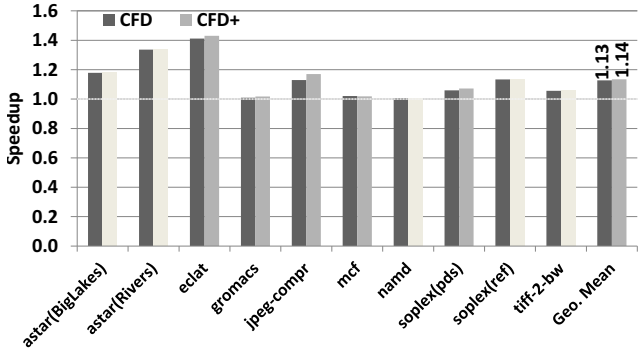
[4] SOPLEX(ref) is an exception. The original loop contains many variables whose live ranges overlap, increasing pressure on architectural registers and resulting in many stack spills/fills. CFD's two loops reduce register contention by virtue of some variables shifting exclusively to the first or second loop, eliminating most of the stack spills/fills, resulting in fewer retired instructions.

[5] The fraction of time spent in the function(s) of interest is found using *gprof* while running the x86 binaries (compiled using gcc with -O3) to completion 3 times on an idle, freshly rebooted Sandy Bridge Processor running in single-user mode.

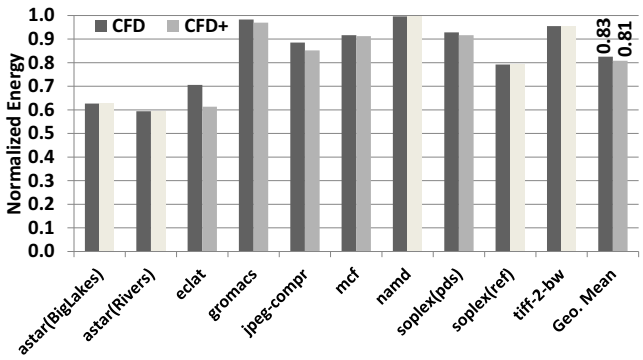
Application	File name	Function	Time spent	Loop line	Branch line	Loop strip mining	Communicate values
astar	Way_.cpp	makebound2	20% (BigLakes) 47% (Rivers)	57	62-63, 79-80 96-97, 113-114 130-131, 147-148 164-165, 181-182	Y	N
eclat	eclat.cc	get_intersect	46%	205	207, 211	Y	Y
gromacs	ns.c	ns5_core	11%	1503	1507, 1508, 1510	N	Y
jpeg-compr	jcdctmgr.c	forward_DCT	83%	322	251	N	Y
	jcphuf.c	encode_mcu_AC_first		488	489		
		encode_mcu_AC_refine		662	663, 686		N
mcf	pbeampp.c	primal_bea_mpp	39%	165	171	Y	Y
namd	ComputeNonbondedBase.h	ComputeNonbondedUtil	5%	397	410	Y	N
soplex	spxsteeppr.cc	selectLeaveX	5% (pds)	291	295	Y	Y
		selectEnterX	17% (ref)	449	452		N
tiff-2-bw	tif_lzw.c	LZWDecode	100%	377	411	N	N

Application	File name	Function	Time spent	Branch Line
clustalw	pairalign.c	forward_pass	98%	384, 388, 391-393
		reverse_pass		436, 440, 443-444
		diff		526-527, 529-530 555-556, 558-559
fasta	dropnfa.c	FLOCAL_ALIGN	47%	1085-1086, 1096-1097 1099-1101, 1104
gsm	add.c	gsm_div	49%	228
	long_term.c	Calculation_of_the_LTP_parameters		152
hmmer	fast_algorithms.c	P7Viterbi	100%	135-137, 142, 147
jpeg-decompr	jdhuff.c	HUFF_EXTEND	50%	372
	jdphuff.c			207
quick-sort	qsort_large.c	compare	43%	26
sjeng	moves.c	make	10%	1305
	search.c	remove_one		515

Table 5: Details of modified code: CFD (left), If-Conversion (right).



(a) Speedup (lightly shaded CFD+ bars mean: no values are communicated)



(b) Energy relative to baseline

Figure 8: Performance and energy impact of CFD.

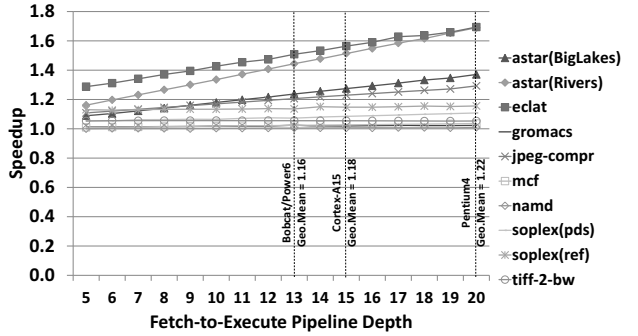


Figure 9: Varying the minimum fetch-to-execute latency.

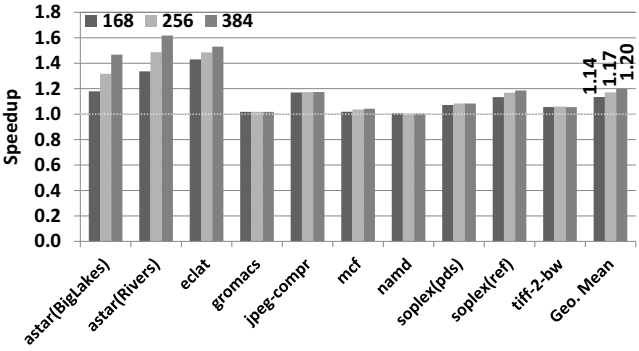


Figure 10: CFD speedups as we scale the processor structures.

6.1. CFD

6.1.1. Manual CFD. We manually apply then evaluate: CFD and CFD+. Figure 8a shows that CFD increases performance by up to 41% and 13% on average, while CFD+ increases performance by up to 43% and 14% on average [6].

Figure 8b shows that CFD reduces energy consumption by up to 41% and 17% on average, while CFD+ reduces energy consumption by up to 41% and 19% on average.

Figure 9 shows speedup with CFD as the minimum fetch-to-execute latency is varied from five to twenty cycles. As expected, CFD gains increase as the pipeline depth increases. The baseline IPC worsens with increasing depth, whereas CFD's eradication of mispredicted branches makes IPC insensitive to pipeline depth. Thus, as is true with better branch prediction, CFD has the added benefit of exacting performance gains from frequency scaling (i.e., deeper pipelining).

[6]The time spent in the functions of interest (shown in Table 5) along with the presented speedups, can be used in Amdahl's law to estimate the speedup of the whole benchmark. For example, ASTAR(Rivers) is sped up by 34% (s=1.34) in its CFD region which accounts for 47% of its original execution time (f=0.47); thus, we estimate 14% (1.14) speedup overall.

To project the gains of CFD+ on future processor generations, we evaluate it under larger instruction windows. Figure 10 shows the projection of CFD gains on two additional configurations labeled in the graph with ROB size[7]. The average performance improvement increases to 20%.

6.1.2. Automated CFD. We present results of our CFD compiler pass for six applications: ECLAT, JPEG, MCF, SOPLEX (pds and ref), and TIFF-2-BW. Figure 11 compares the performance improvements and energy savings of manual CFD+ vs. automated CFD+. The two approaches yield close results for five of the six applications. For SOPLEX(ref), the compiler was unable to register-promote a global variable accessed within the first loop, causing it to be repeatedly loaded within the loop, increasing the instruction overhead and decreasing speedup. The employed alias analysis cannot confirm that the global variable is not stored to by a store within the loop. Inspection of the whole benchmark gives us confidence that interprocedural alias analysis would be able to confirm safety of register-promoting the global variable, because its address is never taken.

As for the other four benchmarks:

1. NAMD, GROMACS: We simply have not yet attempted these

[7][ROB,IQ,LDQ,STQ,PRF] are as follows for the two additional configurations: [**256**, 82, 96, 54, 324] and [**384**, 122, 216, 82, 452]. Other parameters match those of the baseline, shown in Table 3 in Section 5.

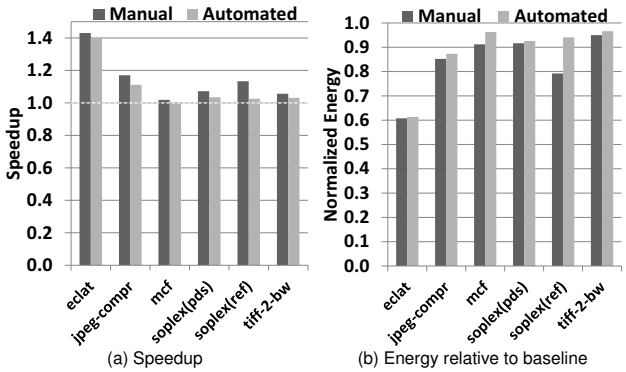


Figure 11: Comparison of manual and automated CFD.

benchmarks but do not anticipate difficulty. NAMD was deprioritized due to low MPKI and we recently added GROMACS to the mix.

2. ASTAR (Rivers and Biglakes): This benchmark has complexity that is not yet supported by our compiler pass: (1) It has a partially separable branch. Note that all other targeted benchmarks have only totally separable branches. (2) The control-dependent instruction in the branch's backward slice is a store, hence, if-converting the backward slice requires the transformation described in Section 2.2. (3) It has two nested, separable branches (one partially separable and one totally separable). We explore this complexity, in depth, in Section 6.1.3.

6.1.3. ASTAR Case Study. One of the most interesting cases we encountered in this work is ASTAR. Figure 12 shows a simplified version of ASTAR's original and decoupled loops.

ASTAR has a few challenging features that require special care when decoupling its loop. First, there are two nested hard-to-predict branches, with the inner predicate depending on a memory reference that is only safe if the outer predicate is true (lines 3 and 4 of original loop). Second, there is a short loop-carried dependency between the outer predicate and one of its control-dependent instructions (line 7 of original loop): this is a partially separable branch.

These challenges are naturally handled by CFD. The nested conditions are handled by decoupling the original loop into *three* loops. The first loop evaluates the outermost condition. The second loop, guarded by the outermost condition, evaluates the combined condition. The third loop guards the control-dependent instructions by the overall condition. The loop-carried dependency is handled by hoisting then if-converting the short loop-carried dependencies (shown in lines 12 and 13 of the second loop).

Due to the high percentage of branch mispredictions that are fed by the L3 cache and main memory, we expect a significant increase in performance gains when we apply CFD to ASTAR under large instruction windows. Figure 13 shows the effective IPC of the unmodified binaries (baseline) and the CFD binaries, as we scale the window size. Our expectations are confirmed for CFD.

6.2. If-Conversion

For completeness, we manually apply if-conversion (using conditional moves) to branches with small control-dependent regions (individual and nested hammocks). Figure 14 shows that if-conversion increases performance up to 76% and 23% on average, and reduces energy consumption by up to 35% and 16% on average.

Note that there is no overlap between the if-converted and control-flow decoupled applications.

	Original Loop	
1	for (…) {	
2	index1=index-yoffset-1;	// 8 instances of this body exist
3	if (**waymap[index1].fillnum != fillnum**)	// hard-to-predict branch (outer predicate)
4	if (**maparp[index1] == 0**) {	// hard-to-predict branch (inner predicate)
5	bound2p[bound2l]=index1;	
6	bound2l++;	
7	waymap[index1].fillnum=fillnum;	// loop-carried dependency
8	waymap[index1].num=step;	
9	}	
10	}	
	Decoupled Loops	
	First Loop	
1	for (…) {	
2	index1=index-yoffset-1;	
3	***pred* = (waymap[index1].fillnum != fillnum);**	// the outer predicate is computed
4	***Push_BQ(pred)*;**	// then pushed onto the BQ
5	}	
	Second Loop	
6	for (…) {	
7	***Branch_on_BQ*{**	// pop the outer predicate
8	index1=index-yoffset-1;	
9	output = waymap[index1].fillnum;	
10	***pred* = (output != fillnum) & (maparp[index1] == 0);**	// evaluate the overall predicate
11	***Push_BQ(pred)*;**	// push the overall predicate
12	***CMOV***(output, fillnum, ***pred***);	// conditional move
13	waymap[index1].fillnum = output;	// always store
14	}	
15	else ***Push_BQ(0)***;	// needed since we always pop in the 3rd loop
16	}	
	Third Loop	
17	for (…) {	
18	***Branch_on_BQ*{**	// pop the overall predicate
19	index1=index-yoffset-1;	
20	bound2p[bound2l]=index1;	
21	bound2l++;	
22	waymap[index1].num=step;	
23	}	
24	}	

Figure 12: ASTAR source code.

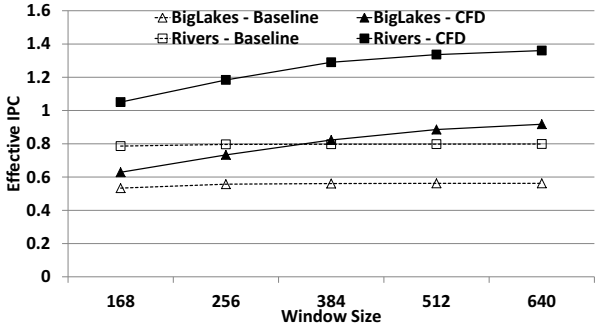


Figure 13: ASTAR: effective IPC of base and CFD binaries as we scale the window size.

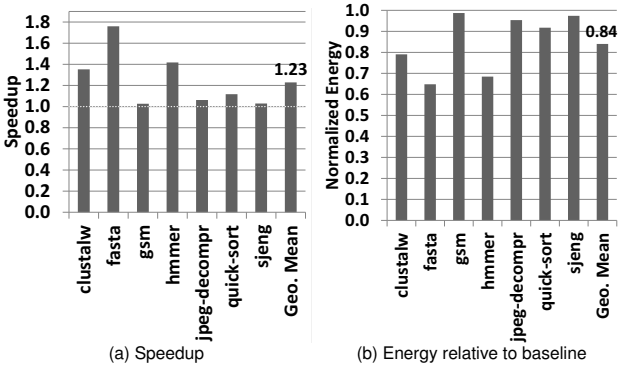


Figure 14: Impact of if-conversion.

7. Related Work

There has been a lot of work on predication and branch pre-execution. We focus on the most closely related work.

Various ingenious techniques for predication have been proposed, such as: software predication [2], predication using hyperblocks [22], dynamic hammock predication [16], wish branches [15], dynamic predication based on frequently executed paths [14], and predicate prediction [26], to name a few. In this paper, predication (i.e., if-conversion) is a key enabling mechanism for applying CFD to partially separable branches.

CFD resembles branch pre-execution [11, 36, 27, 8, 9]. The key difference is that CFD preserves the simple sequencing model of conventional superscalar processors: in-order instruction fetching of a single thread. This is in contrast with pre-execution which requires thread contexts or cores, and a suite of mechanisms for forking helper threads (careful timing, value prediction, etc.) and coordinating them in relation to the main thread. With CFD, a simplified microarchitecture stems from software/hardware collaboration, simple ISA push/pop rules, and recognition that multiple threads are not required for decoupling.

We now discuss several branch pre-execution solutions in more detail.

Farcy *et. al.* [11] identified backward slices of applicable branches, and used a stride value predictor to provide live-in values to the slices and in this way compute predictions several loop iterations in advance. The technique requires a value predictor and relies on live-in value predictability. CFD does not require either.

Zilles and Sohi [36] proposed pre-executing backward slices of hard-to-predict branches and frequently-missed loads using *speculative slices*. Fork point selection, construction and speculative optimization of slices were done manually. Complex mechanisms are needed to carefully align branch predictions generated by speculative slices with the correct dynamic branch instances. Meanwhile, CFD's Push_BQ/Branch_on_BQ alignment is far simpler, always delivers correct predicates, and has been automated in the compiler.

Roth and Sohi [27] developed a profile-driven compiler to extract data-driven threads (DDTs) to reduce branch and load penalties. The threads are non-speculative and their produced values can be integrated into the main thread via register integration. Branches execute more quickly as a result. Similarly, CFD is non-speculative and automation is demonstrated in this paper. CFD interacts directly with the fetch unit, eliminating the entire branch penalty. It also does not have the microarchitectural complexity of register integration. The closest aspect is the VQ renamer, but the queue-based linking of pushes and pops via physical register mappings is simpler, moreover, it is an optional enhancement for CFD.

Chappell *et al.* [8] proposed Simultaneous Subordinate Microthreading (SSMT) as a general approach for leveraging unused execution capacity to aid the main thread. Originally, programmer-crafted subordinate microthreads were used to implement a large, virtualized two-level branch predictor. Subsequently, an automatic run-time microthread construction mechanism was proposed for pre-executing branches [9].

In the Branch Decoupled Architecture (BDA), proposed by Tyagi *et al.* [33], the fetch unit steers copies of the branch slice to a dedicated core as the unmodified dynamic instruction stream is fetched. Creating the pre-execution slice as main thread instructions are being fetched provides no additional fetch separation between the branch's backward slice and the branch, conflicting with more recent evidence of the need to trigger helper threads further in advance, e.g., Zilles and Sohi [36]. Without fetch separation, the branch must still be predicted and its resolution may be marginally accelerated by a dedicated execution backend for the slice.

Mahlke *et al.* [21] implemented a predicate register file in the fetch stage, a critical advance in facilitating software management of the fetch unit of pipelined processors. The focus of the work, however, was compiler-synthesized branch prediction: synthesizing computation to generate predictions, writing these predictions into the fetch unit's predicate register file, and then having branches reference the predicate registers as *predictions*. The synthesized computation correlates on older register values because the branch's source values are not available by the time the branch is fetched, hence, this is a form of branch prediction. Mahlke *et al.* alluded to the theoretical possibility of truly resolving branches in the fetch unit, and August *et al.* [3] further explored opportunities for such *early-resolved branches*: cases where the existing predicate computation is hoisted early enough for the consuming branch to resolve in the fetch unit. These cases tend to exist in heavily if-converted code such as hyperblocks as these large scheduling regions yield more flexibility for code motion. Quinones *et al.* [26] adapted the predicate register file for an OOO processor, and in so doing resorted to moving it into the rename stage so that it can be renamed. Thus, the renamed predicate register file serves as an overriding branch predictor for the branch predictor in the fetch unit. CFD's branch queue (BQ) is innovative with respect to the above, in several ways: (1) The BQ provides renaming implicitly by allocating new entries at the tail. This allows for hoisting all iterations of a branch's backward slice into CFD's early loop, whereas it is unclear how this can be done with an indexed predicate register file as the index is static. (2) Another advantage is accessing the BTB (to detect Branch_on_BQ instructions) and BQ in parallel, because we always examine the BQ head (Section 3.3.5). In contrast, accessing a predicate register file requires accessing the BTB first, to get the branch's register index, and then accessing the predicate register file.

NSR [5] does not predict branches at all, rather, a branch waits in the fetch stage for an enqueued outcome from the execute stage. To avoid fetch stalls, a few instructions must be scheduled by the programmer or compiler in between the branch and its producer instruction. This is like branch delay slots except that, because the fetch unit can stall, no explicit NOPs need to be inserted when no useful instructions can be scheduled. NSR is a 5-stage in-order pipeline so its static scheduling requirement is of similar complexity to branch delay slot scheduling. CFD-class branches require our "deep" static scheduling technique (for in-order and out-of-order pipelines, alike) which in turn requires CFD's ISA, software, and hardware support.

Decoupled access/execute architectures [30, 4] are alternative implementations of OOO execution, and not a technique for hiding the fetch-to-execute penalty of mispredicted branches. DAE's access and execute streams, which execute on dual cores, each have a subset of the original program's branches. To keep them in sync on the same overall control-flow path, they communicate branch outcomes to each other through queues. However, each core still suffers branch penalties for its subset of branches. Bird *et al.* took DAE a step further and introduced a third core for executing all control-flow instructions, the control processor (CP). CP directs instruction fetching for the other two cores (AP and DP). CP depends on branch conditions calculated in the DP, however. These loss-of-decoupling (LOD) events

are equivalent to exposing the fetch-to-execute branch penalty in a modern superscalar processor.

The concept of loop decoupling has been applied in compilers for parallelization. For instance, decoupled software pipelining [25, 34, 12] parallelizes a loop by creating decoupled copies of the loop on two or more cores that cooperate to execute each iteration. All predicates in the backward slices of instructions in the decoupled loops that are not replicated must be communicated. However, predicates are not sent directly to the instruction fetch unit of the other core. Rather, the predicates are forwarded as values through memory or high speed hardware queues and evaluated in the execution stage by a branch instruction.

8. Conclusion

In this paper, we explored the control-flow landscape by characterizing branches with high misprediction contributions in four benchmark suites. We classified branches based on the sizes of their control-dependent regions and the nature of their backward slices (predicate computation), as these two factors give insight into possible solutions. This exercise uncovered an important class of high misprediction contributors, called separable branches. A separable branch has a large control-dependent region, too large for if-conversion to be profitable, and its backward slice does not contain any of the branch's control-dependent instructions or contains just a few. This makes it possible to separate all iterations of the backward slice from all iterations of the branch and its control-dependent region. CFD is a software/hardware collaboration for exploiting separability with low complexity and high efficacy. The loop containing the separable branch is split into two loops (*software*): the first contains only the branch's predicate computation and the second contains the branch and its control-dependent instructions. The first loop communicates branch outcomes to the second loop through an architectural queue (*ISA*). Microarchitecturally, the queue resides in the fetch unit to drive timely, non-speculative fetching or skipping of successive dynamic instances of the control-dependent region (*hardware*).

Measurements of native execution of the four benchmark suites show separable branches are an important class of branches, comparable to the class of branches for which if-conversion is profitable both in terms of number of static branches and MPKI contribution. CFD eradicates mispredictions of separable branches, yielding significant time and energy savings for regions containing them.

Acknowledgements

We thank the anonymous reviewers for their valuable feedback and André Seznec for shepherding the paper. We thank Mark Dechene for developing the simulator used in the study. This research was supported by Intel and NSF grant CCF-0916481. Any opinions, findings, and conclusions or recommendations expressed herein are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] K. Albayraktaroglu *et al.*, "Biobench: a benchmark suite of bioinformatics applications," in *Int'l Symp. on Performance Analysis of Systems and Software*, 2005, pp. 182 –188.

[2] J. R. Allen *et al.*, "Conversion of control dependence to data dependence," in *10th Symp. on Principles of Programming Languages*, 1983, pp. 177–189.

[3] D. August *et al.*, "Architectural support for compiler-synthesized dynamic branch prediction strategies: Rationale and initial results," in *3rd Int'l Symp. on High-Performance Computer Architecture*, 1997, pp. 84–93.

[4] P. L. Bird, A. Rawsthorne, and N. P. Topham, "The effectiveness of decoupling," in *7th Int'l Conf. on Supercomputing*, 1993, pp. 47–56.

[5] E. Brunvand, "The nsr processor," in *26th Hawaii Int'l Conf. on System Sciences*, vol. 1, 1993, pp. 428–435.

[6] B. Burgess *et al.*, "Bobcat: amd's low-power x86 processor," *IEEE Micro*, vol. 31, no. 2, pp. 16–25, 2011.

[7] D. Carmean, "Inside the pentium 4 processor micro-architecture." Presented at Intel Developer Forum, 2000.

[8] R. Chappell *et al.*, "Simultaneous subordinate microthreading (ssmt)," in *26th Int'l Symp. on Computer Architecture*, 1999, pp. 186–195.

[9] R. Chappell *et al.*, "Difficult-path branch prediction using subordinate microthreads," in *29th Int'l Symp. on Comp. Arch.*, 2002, pp. 307–317.

[10] cTuning, "Collective Benchmark," in *http://cTuning.org/cbench*.

[11] A. Farcy *et al.*, "Dataflow analysis of branch mispredictions and its application to early resolution of branch outcomes," in *31st Int'l Symp. on Microarchitecture*, 1998, pp. 59–68.

[12] J. Huang *et al.*, "Decoupled software pipelining creates parallelization opportunities," in *8th Int'l Symp. on Code Generation and Optimization*, 2010, pp. 121–130.

[13] E. Jacobsen, E. Rotenberg, and J. Smith, "Assigning confidence to conditional branch predictions," in *29th Int'l Symp. on Microarchitecture*, 1996, pp. 142–152.

[14] H. Kim *et al.*, "Diverge-merge processor (dmp): dynamic predicated execution of complex control-flow graphs based on frequently executed paths," in *39th Int'l Symp. on Microarchitecture*, 2006, pp. 53–64.

[15] H. Kim *et al.*, "Wish branches: combining conditional branching and predication for adaptive predicated execution," in *38th Int'l Symp. on Microarchitecture*, 2005, pp. 43–54.

[16] A. Klauser *et al.*, "Dynamic hammock predication for non-predicated instruction set architectures," in *7th Int'l Conf. on Parallel Architectures and Compilation Techniques*, 1998, pp. 278–285.

[17] T. Lanier, "Exploring the design of the cortex-a15 processor," 2011.

[18] H. Q. Le *et al.*, "Ibm power6 microarchitecture," *IBM Journal of Research and Development*, vol. 51, no. 6, pp. 639–662, 2007.

[19] S. Li *et al.*, "Mcpat: an integrated power, area, and timing modeling framework for multicore and manycore architectures," in *42nd Int'l Symp. on Microarchitecture*, 2009, pp. 469–480.

[20] C.-K. Luk *et al.*, "Pin: building customized program analysis tools with dynamic instrumentation," *SIGPLAN Not.*, vol. 40, no. 6, pp. 190–200, Jun. 2005.

[21] S. Mahlke and B. Natarajan, "Compiler synthesized dynamic branch prediction," in *29th Int'l Symp. on Microarch.*, 1996, pp. 153–164.

[22] S. Mahlke *et al.*, "Effective compiler support for predicated execution using the hyperblock," in *25th Int'l Symp. on Microarchitecture*, 1992, pp. 45–54.

[23] O. Mutlu *et al.*, "Runahead execution: an alternative to very large instruction windows for out-of-order processors," in *9th Int'l Symp. on High-Performance Computer Architecture*, 2003, pp. 129–140.

[24] R. Narayanan *et al.*, "Minebench: a benchmark suite for data mining workloads," in *Int'l Symp. on Workload Characterization*, 2006, pp. 182–188.

[25] G. Ottoni *et al.*, "Automatic thread extraction with decoupled software pipelining," in *38th Int'l Symp. on Microarchitecture*, 2005, pp. 105–118.

[26] E. Quinones, J.-M. Parcerisa, and A. Gonzalez, "Improving branch prediction and predicated execution in out-of-order processors," in *13th Int'l Symp. on High Perf. Computer Architecture*, 2007, pp. 75–84.

[27] A. Roth and G. Sohi, "Speculative data-driven multithreading," in *7th Int'l Symp. on High-Perf. Computer Architecture*, 2001, pp. 37–48.

[28] A. Seznec, "A 64 kbytes isl-tage branch predictor," in *3rd Championship Branch Prediction*, 2011.

[29] A. Seznec, "A new case for the tage branch predictor," in *44th Int'l Symp. on Microarchitecture*, 2011, pp. 117–127.

[30] J. E. Smith, "Decoupled access/execute computer architectures," in *9th Int'l Symp. on Computer Architecture*, 1982, pp. 112–119.

[31] S. T. Srinivasan *et al.*, "Continual flow pipelines," in *11th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, 2004, pp. 107–119.

[32] Standard Performance Evaluation Corporation, "The SPEC CPU 2006 Benchmark Suite," in *http://www.spec.org*.

[33] A. Tyagi, H.-C. Ng, and P. Mohapatra, "Dynamic branch decoupled architecture," in *17th Int'l Conf. on Comp. Design*, 1999, pp. 442–450.

[34] N. Vachharajani *et al.*, "Speculative decoupled software pipelining," in *16th Int'l Conf. on Parallel Architecture and Compilation Techniques*, 2007, pp. 49–59.

[35] B. Valentine, "Introducing sandy bridge." Presented at Intel Developer Forum, San Francisco, 2010.

[36] C. Zilles and G. Sohi, "Execution-based prediction using speculative slices," in *28th Int'l Symp. on Computer Architecture*, 2001, pp. 2–13.

2012 IEEE/ACM 45th Annual International Symposium on Microarchitecture

Spatiotemporal Coherence Tracking

Mohammad Alisafaee
École Polytechnique Fédérale de Lausanne, Lausanne, Switzerland
mohammad.alisafaee@epfl.ch

Abstract

Chip-multiprocessors require a coherence directory to track data sharing and order accesses to the shared data. Scaling coherence directories to support a large number of cores is challenging due to excessive area requirements of the directories. The state-of-the-art proposals reduce the directory size by not keeping coherence information for private data. These approaches are useful for workloads that have predominantly private data, but are not applicable to workloads with shared data.

We observe that data are not actively shared by multiple cores. In workloads with a shared dataset, although each core accesses the whole data, the chance that multiple cores access the same piece of data at the same time is low. Based on this observation we design a Spatiotemporal Coherence Tracking scheme that drastically reduces the directory size without sacrificing performance. The proposed directory scheme uses dual-grain tracking and switches between the granularities whenever possible to save the area. It dynamically detects spatial regions of data that are privately accessed by one core over a time period and for those regions, increases coherence tracking granularity from block-level to region-level. Our experimental results show that the proposed approach can reduce the baseline sparse directory size by at least 75% across a variety of commercial and scientific workloads, while sacrificing only 1% of performance. Using our approach, the directory can be under-provisioned to have fewer entries than the number of cache blocks that are being tracked.

1. Introduction

CMOS technology trends direct processor manufacturers towards integrating more cores in a single chip with every new technology generation. Scaling to larger shared-memory chip-multiprocessors (CMPs) requires scalable coherence mechanisms to keep the cores' private caches coherent. Directory-based coherence mechanisms have been used in chip-multiprocessors with a handful of cores. However, area overhead and energy consumption of directories increase with the core count and the size of private caches. Designing CMPs with a large number of cores calls for area- and energy-efficient coherence directories.

In directory designs, there exists a trade-off between the area and the associativity. Duplicate-tag directories are area-efficient but their associativity increases with the number of cores, making them practical only in small-scale designs [2, 9]. Sparse directories, on the other hand, use arrays with a low associativity. However, prior work shows that as the core count increases, the area requirements of sparse directories become comparable to the size of the caches they track [8, 11, 22]. The directory can consume a large fraction of the die area in systems with large private caches and a large number of cores. Part of the sparse directory area overhead is due to the over-provisioning in the number of entries that is needed to minimize conflicts in an array with a low associativity [8].

Prior research on sparse directory design tries to reduce the required number of entries notably by leveraging hashing techniques to reduce the conflict frequency and the required over-provisioning [8, 17, 19]. Recent proposals reduce the number of the directory entries by deactivating coherence for private data, which are accessed by one core all the time [6, 11]. Cuesta et al. propose a directory scheme that detects private pages and avoids tracking them in the directory [6]. Their proposal, similar to [11], relies on the operating system's TLB miss handler and the page table to detect private pages. This scheme can significantly reduce the directory size for workloads with predominantly private data. However, there are limited amount of private data in commercial server workloads [7, 11] and even in scientific workloads [12], making this scheme inapplicable.

We observe that across a variety of commercial server and scientific workloads, data are accessed and cached by one core most of the time. These workloads have large datasets and the likelihood that more than one core access the same data region at the same time, is low. The implication is that although the whole data is shared and accessed by all the cores, each data region is private to one core for a period of time. Using this property of data accesses offers a greater opportunity in reducing the directory size than using only private data. We refer to spatial regions of data that are accessed and cached by one core for a period of time as *temporarily-private* regions.

In this paper, we propose a coherence directory, Spatiotemporal Coherence Tracking (SCT), which aims at reducing the required number of directory entries. SCT is a dual-grain coherence tracking mechanism that tracks temporarily-private data at a region granularity and tracks shared data at a cache block granularity. SCT detects temporarily-private regions by observing cores' access requests and maintains only one directory entry for any number of blocks that are cached in such regions. Tracking coherence at a coarser granularity reduces the required directory capacity. Our proposal is applicable to a variety of workloads with both shared and private datasets.

We evaluate our proposal along with other coherence directory organizations using full-system simulation of a 16-core CMP running a variety of commercial and scientific workloads.

1072-4451/12 $26.00 © 2012 IEEE
DOI 10.1109/MICRO.2012.39

IEEE computer society

We analyze sharing patterns of the workloads to measure the available opportunity at reducing the directory size using temporarily-private regions. Our trace-based and cycle-accurate simulations show that:

- More than 91% of the occupied directory entries are temporarily private and 77% of the occupied directory entries map to temporarily-private pages. In comparison, only 32% of the entries map to private pages.
- SCT requires four times less area to store the coherence information compared to a sparse directory with a 2x over-provisioning while losing only 1% of performance. In other words, the SCT capacity is under-provisioned by 2x.
- In comparison with a recent proposal, which leverages private data to reduce the directory size, SCT requires 50% less area while delivering identical performance.
- SCT successfully classifies temporarily-private and shared regions. More than 95% of the directory accesses are made to regions that all of their cached blocks are either temporarily-private or shared.
- The percentage of temporarily-private data does not decrease in systems with large private caches.

The rest of this paper is organized as follows: Section 2 provides background on the directories and motivates the work. In Section 3, we propose our idea and evaluate the available opportunity at reducing the directory size. The design of the Spatiotemporal Coherence Tracking and the details of various coherence operations are described in Section 4. Section 5 contains our evaluation methodology and simulation results. Section 6 discusses the related work and finally Section 7 concludes the paper.

2. Motivation

In shared-memory multiprocessors with one or more levels of private caches per core, the coherence directory orders accesses to the shared memory space. The directory observes all the access requests to the data which come from cores and maintains a list of sharers for every privately-cached piece of data [8]. The granularity at which the directory keeps the track of the sharers is usually equal to the cache block size. Every block that is cached in the cores' private cache hierarchy has a corresponding entry in the directory. Therefore, the number of entries in the directory must be at least equal to the aggregate number of blocks of private caches.

Directory entries are kept inside a set-associative storage array. In designs where the associativity of the directory is less than the aggregate associativity of the private caches (i.e., number of cores times the associativity), conflicts can happen. A conflict happens when more addresses than the directory associativity map to the same set in the directory. Upon occurrence of a conflict, the directory has to invalidate an existing address from the conflicting set and all sharers' caches, to make room for the new entry. As a result, conflicts may increase the miss rate of private caches and can adversely affect the performance. To decrease the number of conflicts in a directory, one can either increase the associativity or over-provision the directory to have more entries than the aggregate number of cache blocks in private caches.

Prior work shows that highly associative directories consume a huge amount of power and cannot be deployed in designs with a high core count [8, 22]. Therefore, a feasible solution is a directory with a limited associativity, like a sparse directory. To minimize conflicts, a 2-4x over-provisioning is normally used in sparse directories implementations [1, 5]. However, [8] shows that even with an 8x over-provisioning, it is still possible to have conflicts in a sparse directory. The excessive overheads of the sparse directories make them area-inefficient for designs with a large number of cores.

In this work, we focus on the sparse directories and propose a technique to reduce their area overheads. Our proposal aims at reducing number of entries that required to be kept in the directory by increasing the coherence granularity.

3. Spatiotemporal Coherence Tracking

Our proposed directory scheme decreases the directory size by dynamically increasing the coherence tracking granularity for temporarily-private data regions. In this section, we describe the concept of temporarily-private data and the way we use it to reduce the directory size.

3.1. Temporarily-private Data

Data accessed by a program are conventionally categorized into *shared* and *private* based on their access pattern. Private data are always accessed by the same thread during the entire program execution while shared data are accessed by multiple threads. At the architectural level, data accesses come from cores and no notion of threads is available. From the hardware perspective, private data are those that are always accessed by the same core while shared data are accessed by multiple cores.

We observe that although shared data are accessed by more than one core, they are not accessed concurrently (i.e., they are not actively shared). Based on this observation we introduce a new definition of private data which includes a notion of time in it. Our notion of time here is the period in which the data is cached in the private cache hierarchies of the cores. We call a piece of data *temporarily private* if it is currently cached by only one core. Similarly, a piece of data is *temporarily shared* at a particular time, if it is cached by at least two cores. Our definition of temporarily-private only considers the number of cores that are accessing the data instead of looking at the number of cores that have accessed the data. Using these new definitions, the same piece of data can be temporarily private or temporarily shared at different times. This is in contrary to the core-centric classification of data in which data are forever considered to be shared as soon as they are accessed by more than one core.

3.2. Spatial Coherence Tracking

In order to reduce the directory size, our proposal tracks coherence at a coarser granularity than the cache block size. We

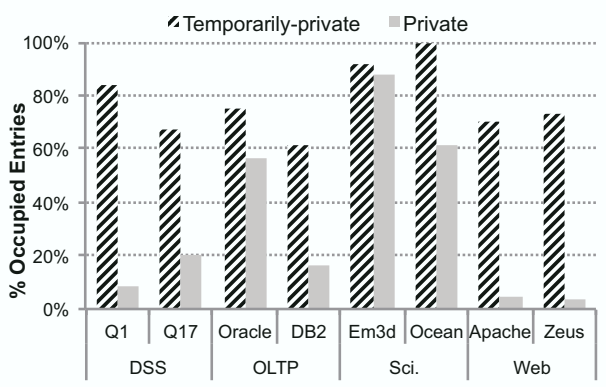


Figure 1. Directory size reduction opportunity when using temporarily-private data compared to using private data.

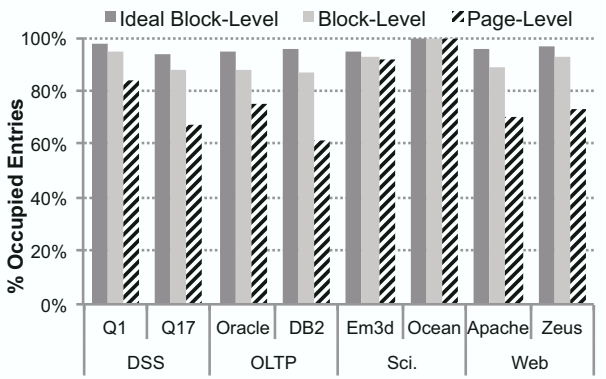


Figure 2. Directory size reduction opportunity when using temporarily-private data with ideal block-level, block-level, and page-level granularities.

virtually divide the memory address space into fixed-size spatial regions. Each region contains multiple consecutive cache blocks. Our proposal dynamically identifies spatial regions that are temporarily private and keeps coherence information at the region-level for those regions. The proposed directory stores only a single entry for an entire temporarily-private region. At best, if all the regions are temporarily-private, the directory size can be reduced by a factor equal to the average number of blocks per temporarily-private region.

3.3. Classifying Regions

To classify regions as temporarily-private or temporarily-shared, we rely on the access requests sent by the cores. Memory access requests (i.e., read, write, and evict) determine when a core caches and evicts data from data regions (we assume a coherence protocol with explicit clean eviction messages). The first read or write access to a region that has no cached blocks in cores' private caches, hints the start of a period in which the region is cached. The region is evicted when all of its cached blocks are evicted. The period between the start of an access to a region until the eviction of the last cached block can be identified by looking at access requests.

Relying on memory access requests to identify temporarily-private regions, however, can result in false-positives. A region might be misclassified as temporarily-shared even though it really is temporarily-private. A misclassification happens when the time periods in which a region is cached by different cores overlap while the accesses do not overlap. Cached blocks of a region are not immediately evicted after the last access and will remain in the cache as dead blocks for a while before they get replaced. If other cores access these dead cache blocks, a false-positive happens. The rate of false-positives depends on the residency period of regions in caches and is affected by the cache parameters (i.e., size, associativity, and replacement policy) and the region size.

3.4. Opportunity Analysis

We do an analysis to quantify the available opportunity at reducing the directory size. Details of the evaluation methodology and the workloads can be found in Section 5.

At first, we measure the available opportunity when using private data and compare it to using temporarily-private data to reduce the directory size (Figure 1). A mechanism that uses private data to reduce the directory size is similar to what proposed in [6], which relies on the OS to identify private pages. To make a fair comparison, we use a region size of 8 KB in our approach (8 KB is the OS page size).

Each bar in Figure 1 shows the percentage of the occupied directory entries that map to temporarily-private or private pages. These results show that only a small percentage of data are always accessed by the same core and thus are private, which corroborates prior work [7, 11]. In workloads with predominantly shared data (e.g., web workloads, and DSS) using private data to reduce the directory size shows limited opportunity while using temporarily-private provides considerably higher opportunity. Scientific workloads like Ocean also show higher opportunity when using temporarily-private instead of private. Averaged across all the workloads, using temporarily-private data provides 45% higher opportunity at reducing the directory size compared to using private data.

We, now, focus on the opportunity of using temporarily-private data at reducing the directory size. To measure the maximum available opportunity, we implement an ideal scheme that can exactly determine the residency period of regions in private caches. This ideal scheme uses a perfect dead-block predictor to predict when the last access to a cached region happens. It removes all the false positives when classifying regions by perfectly avoiding all unnecessary overlapped caching of regions. We also measure the opportunity loss when using a larger region than a cache block. Our approach needs to keep some information about the spatial regions that are not considered in these results.

Figure 2 shows the opportunity analysis results for three variants of our proposal: Ideal block-level, block-level, and page-level. The block-level schemes track coherence at the cache block granularity (i.e., 64 bytes) and the page-level tracks coherence at OS page granularity (the same as Figure 1). This figure shows that more than 97% of the entries are temporarily-private if an ideal scheme is used. In other words, the directory

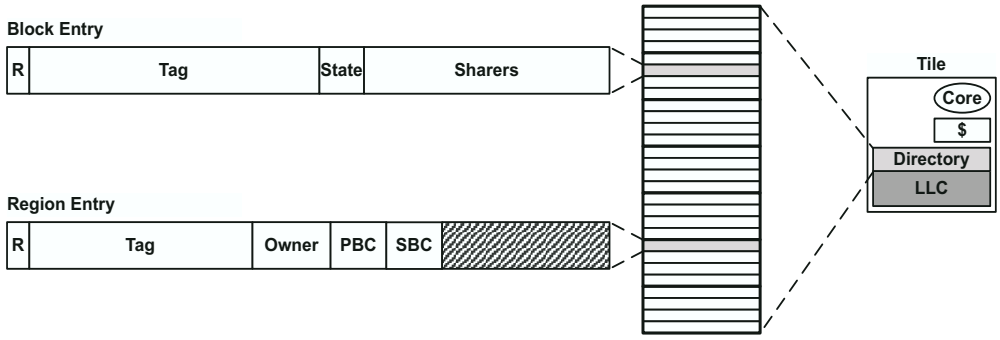


Figure 3. SCT uses a set-associative array to store block- and region-level coherence information.

size can be reduced to 4% without losing any information on data tracking. Tracking temporarily-private data at the block granularity yields 6% less opportunity compared to the block-level ideal scheme due to the false positives. When tracking coherence at the page granularity instead of the block granularity, the available opportunity is reduced by 14%. Larger spatial regions are more likely to be accessed by multiple cores and be temporarily-shared. In addition, the chance of a false-positive is higher with larger regions.

4. Design

Our baseline system is a CMP with a tiled architecture. Every tile contains a core with its private caches, a slice of the shared last-level cache (LLC), a slice of the directory, and routers and interconnects. In our design, we use a 4-way set associative sparse directory as the baseline directory to store coherence information. The directory maintains a bit-vector of sharers for every block that is cached in the cores' private caches.

SCT keeps coherence information at both region and block granularities. It uses the same storage array to store both region- and block-level coherence information. Each entry can hold coherence information for either a region or a block. Information stored for blocks are the same as the baseline directory. We add a single bit to each entry to determine if the entry contains coherence information for a block or a region. Information that SCT needs to maintain for regions are described in Section 4.2.

4.1. Directory Organization

Our SCT design uses a distributed storage for keeping coherence information. Directory and LLC slices are address interleaved at the region granularity. Therefore, directory entries for cache blocks in a region map to the same tile as the directory entry for the region. Figure 3 shows schematic of a directory slice and the storage formats for region- and block-level entries.

4.2. Region-level Information

SCT needs to keep coherence information for every region that has at least one cached block in the cores' private caches. These information are updated when the cores send requests for cache blocks in a region, when cache block get evicted, or when the directory invalidates a conflicting entry to make room for a new address.

A temporarily-private region is a region which its blocks are cached by only one core. When a region is first accessed by a core, SCT marks it as temporarily-private. If further accesses to the region come from the same core, the region remains temporarily private, but if other cores access the region, it becomes temporarily shared. For temporarily-shared regions, SCT allocates one block-level entry in the directory per shared block. The region remains in either states until all of its cached blocks are evicted.

We name the first core that accesses a region as the *owner* of the region. For every region, SCT needs to know which core is its owner and which of its blocks have been cached by the owner so far. SCT must also know the state of the region, which can be temporarily private or temporarily shared. For temporarily-shared regions, SCT needs to know which blocks are shared.

Figure 3 shows the format of the region-level information as is kept in the SCT directory. SCT stores the core ID of the owner for every cached region. We do not maintain an exact list of the cache blocks that are cached by the owner. Instead, we only keep a counter, *Private Blocks Counter (PBC)*, to count the number of blocks cached by the owner. Using a counter instead of tracking the exact list of the cached blocks in a bit-vector reduces the storage overhead. However, when the directory needs to know whether the owner has cached a block or not, a probe message must be sent to the owner. Similarly, we keep a counter, *Shared Blocks Counter (SBC)*, to count the number of shared blocks. SBC shows the number of block-level entries that are allocated for a temporarily-shared region. We need to maintain the number of shared blocks to determine when a temporarily-shared region has all of its shared blocks evicted.

SCT updates PBC and SBC according to memory access requests. PBC is incremented when the owner sends a read or

write request to a temporarily-private region and is decremented upon a clean or dirty eviction. PBC is also decremented when one of the owner's cached blocks is accessed by another core in a temporarily-shared region. SBC is incremented when a new cache block is accessed in a temporarily-shared region and is decremented when a shared block is evicted by all of its sharers or invalidated due to a conflict in the directory. Based on the values of these two counters, SCT knows if the region is temporarily-private (SBC==0), temporarily-shared (SBC>0), or has been evicted (PBC==0 && SBC==0). When a region is evicted, SCT de-allocates its directory entry. Later, when the region is accessed by a core, it becomes temporarily-private to that core. Section 4.3 describes various SCT operations in more details.

4.3. Coherence Operations

Directory operations differ for various region types. When a request arrives, SCT always looks for an associated region-level entry for the requested address. Depending on the region information, type of the access, and the requester's core ID, several scenarios are possible: Access to a non-existing region, access to a temporarily-private region either from its owner or from another core, access to a temporarily-shared region, and cache blocks evictions. The following list describes each of these scenarios in more details:

Access to a non-existing region: If SCT does not find an associated region-level entry, the region has no block in private caches. SCT allocates an entry for the region in the directory and fills the owner field of the newly allocated entry with the core ID of the requester. PBC is set to one, indicating that the region is temporarily private. Allocating a new entry in the directory might require invalidating another block- or region-level entry (see Section 4.4).

Accesses to a temporarily-private region from its owner: When SCT finds out that the region is temporarily private and is being accessed by its owner, the only operation it needs to do is to update PBC according to the request type. A read or write request means that the owner does not have the cached block and PBC must be incremented.

Access to a temporarily-private region from a core other than its owner: When a temporarily-private region is accessed by a core other than its owner, the region becomes temporarily shared. SCT does not invalidate the blocks that are privately cached by the owner and allows them to coexist with the shared blocks. It allocates a new entry for the requested cache block and sets the requesting core as a sharer. The SBC counter is set to one, indicating that the region has one shared block. A copy of the block must be sent to the requesting core. SCT must also know if the owner has a copy of the block and update the sharing information accordingly.

SCT looks up LLC for a copy of the block and checks the state of the block (on a hit) which might reveal if the owner has a copy or not. If the state of the block tells that the owner has a copy, SCT decrements PBC and adds the owner to the sharers list of the block if the access is a read. On a write access, SCT invalidates the owner's copy. If LLC has a valid copy then it replies to the requester and otherwise SCT asks the owner to forward the data to the requester.

When the block is not found in LLC, SCT needs to send a probe message to the owner. Upon receiving a probe message, the owner core looks up its cache for a copy of the block and directly replies to the requesting core with the data if a copy exists in its cache (we implement a 3-hop protocol). The owner then sends back a reply to SCT to either acknowledge the existence of the data or notify the miss. Based on the owner's reply, SCT either updates PBC and the sharers list or brings data from the off-chip memory.

If LLC has a valid copy of the data but its state does not reveal that the owner has a copy or not, SCT replies to the requester with the data and tells the requester to wait for an acknowledge or a negative acknowledge from the owner if the access is a write (a read accesses does not need to wait as the LLC has a valid copy). SCT then sends a probe message to the owner to inspect the existence of the block in its cache. Similar to the previous scenario, the owner looks up its cache and acknowledges the requesting core (upon a write) and replies back to SCT to update the sharing information.

The probe message sent to the owner might introduce an extra delay for some memory accesses. In our SCT protocol, the access latency increases only when the owner does not have a copy and either LLC does not have a copy or LLC has a copy and the access is a write. In all the other cases, the probe message would also be sent in the baseline directory protocol or is not on the critical path of the access.

Accesses to a temporarily-shared region: When SCT finds out that the region is temporarily-shared, it looks up the directory for the requested block address. If an entry is found then it contains exact sharer information for the block and SCT acts as a normal block-level directory. If the block does not have an entry and the requester is not the owner and the owner has some cached blocks (PBC>0), then the same scenario that a temporarily-private region was being shared, happens. In all the other cases, it is guaranteed that the owner does not have a copy of the block and SCT allocates a new entry for the block and increments SBC. The data is then filled either from LLC or from the memory.

Evicting cache blocks from a temporarily-private region: Upon receiving an eviction message from the owner, SCT decrements PBC. If the counter reaches zero, then the region is not cached anymore and SCT de-allocates the region directory entry.

Evicting cache blocks from a temporarily-shared region: SCT looks up the directory to find the corresponding block-level entry. If an entry is found, then the sender is removed from the sharers list. Then, if the block has no more sharers, its directory entry is de-allocated and the SBC is decremented. If an entry for the block is not found, then the sender of the eviction message must be the region owner. In this case, PBC is decremented. If both PBC and SBC are zero, then the region directory entry is de-allocated.

4.4. Invalidating Directory Entries

Because our implementation of SCT uses a sparse directory as the baseline, it is possible to have conflicting entries when inserting a new entry to the directory. On a conflict, SCT must select a victim from the existing entries in the directory. The victim can be a block- or a region-level entry. Invalidating a block-level entry involves in sending invalidation message to its sharers.

Invalidating a region-level entry requires more work: SCT sends an invalidation message to the owner if PBC value is not zero. The owner's private cache must be looked up for all blocks in the region because we do not exactly know which blocks are cached. Similarly, if the victim region is temporarily-shared, the directory must be looked up for all blocks in the region. During a region invalidation operation, access requests to that region are suspended. When the region invalidation is done, suspended accesses (if any) are resumed. In our implementation, we select a block-level victim whenever possible.

5. Evaluation

5.1. Methodology

We evaluate our proposal using trace-based and cycle-accurate full system simulation of a chip-multiprocessor running commercial server and scientific workloads. We use FLEXUS [21] to model the system. FLEXUS can run unmodified binaries of commercial applications and operating systems. FLEXUS extends Virtutech Simics functional simulator with cycle-accurate models of out-of-order cores, cache hierarchy, and interconnection network.

We model two 16-core tiled chip-multiprocessor systems: one with a 2-level cache hierarchy (private L1s and a shared L2) and the other with a 3-level cache hierarchy (private L1s and L2s and a shared L3). The system with a 3-level cache hierarchy is only used in Section 5.7 to evaluate scalability of SCT to larger caches. All the other results, including the performance comparisons, are reported only for the system with a 2-level cache hierarchy. Table 1 shows various parameters of the modeled systems.

Simulated systems run the *Solaris 8* operating system and execute a variety of scientific and commercial server workloads. The commercial workloads include online transaction processing (OLTP), decision support systems (DSS), and web servers. Table 2 lists the workloads and their associated parameters. Our set of workloads covers a wide range of applications with various sharing patterns [11].

We use LLC access traces for opportunity analysis and miss-rate comparison. These traces are generated by running FLEXUS in an in-order execution mode with an IPC of 1. For OLTP, web, and DSS Q1 workloads, we warm up the simulator state for one billion cycles and then run for another one billion cycles and report the results. For DSS Q17 we run the simulations for 300 million cycles, which covers the entire query execution time. For scientific workloads, Em3d and Ocean, we run the simulations for five iterations.

To compare performance, we use cycle-accurate simulations based on the SimFlex sampling methodology [21]. Our samples cover an interval of 10 to 30 seconds of simulation time for OLTP and Web, over the complete query execution for DSS, and over one iteration for scientific workloads. Each sample has warmed caches and branch predictor states. We run each sample for 100,000 cycles in a cycle-accurate simulator to warm up micro-architectural states (e.g., ROB, LSQ) and then measure and report performance for 50,000 cycles. We show the results along with the 95% confidence interval.

In addition to SCT, we implement several other directories including duplicate-tag, sparse, and a scheme similar to [6], which we refer to it as PRV. PRV does not allocate cache blocks that belong to private pages in the directory.

Table 1: System parameters.

CMP	16-core, tiled architecture
Cores	UltraSPARC III ISA, 2 GHz, OoO cores, 3-wide dispatch/commit, 60-entry ROB, 16-entry LSQ and store buffer
L1 cache	32 KB, 4-way split I/D caches, 64-byte lines, 3-cycle load-to-use, 2 ports
L2 cache	2-level: shared, 512 KB per tile, 16-way, 64-byte lines 3-level: private, 128, 256, and 512 KB per tile, 8-way (128 and 256 KB) and 16-way (512 KB), inclusive, 64-byte lines
L3 cache	Shared, 1 MB per tile, 16-way, 64-byte lines
Coherence protocol	3-hop, MESI
Main memory	3 GB, 8 KB pages, 45ns access latency
Interconnect	4x4 mesh, 16-byte links, 1 cycle link latency, 2 cycles router latency

Table 2: Workloads.

Online Transaction Processing (TPC-C v3.0)	
Oracle	100 warehouses (10 GB), 16 clients, 1.4 GB SGA Oracle 10g Enterprise Database Server
DB2	100 warehouses (10 GB), 64 clients, 450 MB buffer pool IBM DB2 v8 ESE
Decision Support (TPC-H on DB2 v8)	
Q1	Scan-dominated, 450 MB buffer pool
Q17	Balanced scan-join, 450 MB buffer pool
Web Server (SPECweb99)	
Apache	16K connections, FastCGI, worker threading model Apache HTTP Server v2.0
Zeus	16K connections, FastCGI Zeus Web Server v4.3
Scientific	
Em3d	3M nodes, degree 2, span 5, 15% remote
Ocean	1026x1026 grid, 9600s relaxations, 20K res., err tol 1e-07

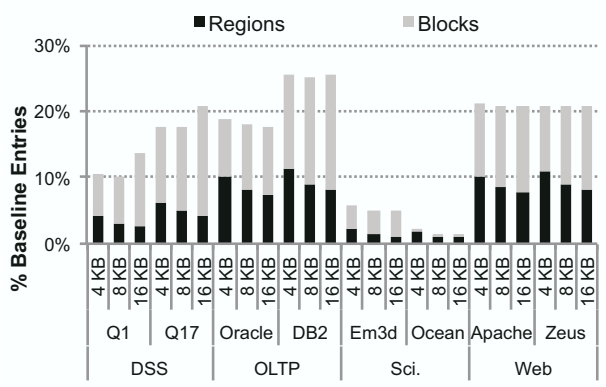


Figure 4. Sensitivity of the SCT size to the region size.

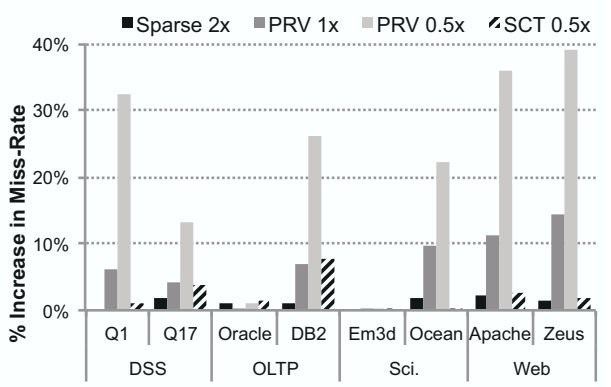


Figure 5. Increase in the L1D miss-rate over duplicate-tag.

In all of the experiments and for all of the implemented directory schemes, we separate instruction blocks from data blocks and focus only on the data directory. Instruction blocks are rarely updated and are shared by all the cores [11], meaning that they either do not need a coherence directory or can be tracked with a small directory. We use a dedicated directory for instructions and make sure that they receive all the updates.

5.2. Impact of the Region Size

At first, we need to select the size of the spatial regions as it affects the performance of our proposal. When selecting the region size, there exists a trade-off between the storage requirements of block-level and region-level entries. Smaller regions reduce the likelihood of misclassifying a temporarily-private region as temporarily-shared and thus need less block-level storage. However, when the region size is small, more regions need to be tracked and more storage is required to store region-level information. A larger region size increases the chance of misclassifying a temporarily-private region as shared, which limits the benefits of spatial coherence tracking.

To quantify the effect of the region size on the storage requirements, we measure the opportunity in using spatiotemporal coherence tracking for various region sizes. Figure 4 shows the total number of required entries in a SCT directory as a fraction of the sparse 2x baseline for various region sizes. These results include the area overhead used to store region information. Each bar is divided into two parts, showing the percentage of the entries that are taken up by regions and cache blocks.

In workloads that have predominantly private data (scientific workloads and OLTP on Oracle), the total required storage decreases as the region size increases. This behavior is expected because in these workloads, increasing the region size does not increase the region misclassification probability as the data are private anyhow. In workloads with predominantly shared data, increasing the region size up to 8 KB has minor effect on the storage requirements. In these workloads, the storage saving achieved by using a larger region is offset by an increase in the number of block-level entries. When the region size is increased to 16 KB, the total required storage increases because of the larger number of block-level entries. We choose a region size of 8 KB for the rest of our simulations because it is the optimal size across all the workloads.

One important conclusion from this experiment is that the average number of directory entries required to store region information is 8%, 6%, and 5% of the baseline directory for region sizes of 4 KB, 8 KB, and 16 KB respectively. SCT incurs this area overhead to enable tracking of temporarily-private regions which provides a greater area saving in return.

5.3. Miss-rate Comparison

Directories with a low associativity storage array have forced cache block invalidations and therefore may increase the cache miss rate. We measure L1D miss rates for various directories and report the increase in the miss rate compared to a duplicate-tag directory. A duplicate-tag directory does not have any conflicts and therefore does not generate directory-induced invalidations.

Figure 5 shows the percentage of increase in the L1D miss rate over a duplicate-tag directory for four different directory organizations: The sparse 2x baseline, PRV 1x, PRV 0.5x, and SCT 0.5x. The number after each directory name denotes the over-provisioning factor. The sparse 2x baseline has twice the number of entries as the total number of cache blocks in L1Ds. A 1x directory has the same number of entries as the total number of L1Ds blocks while a 0.5x directory has half the number of entries.

On average, SCT incurs a 2% higher L1D miss rate over the duplicate-tag design. The maximum increase in miss rate is 8% for OLTP on DB2. L1D miss rates in SCT and the Sparse 2x baseline are almost identical. In comparison to SCT, PRV 0.5x, which occupies identical silicon area, has significantly higher miss rates (22%). This is specially true in workloads in which temporarily-private provides a higher opportunity compared to private (see Figure 1). PRV 1x also has higher miss rates compared to SCT 0.5x although it has twice the number of directory entries.

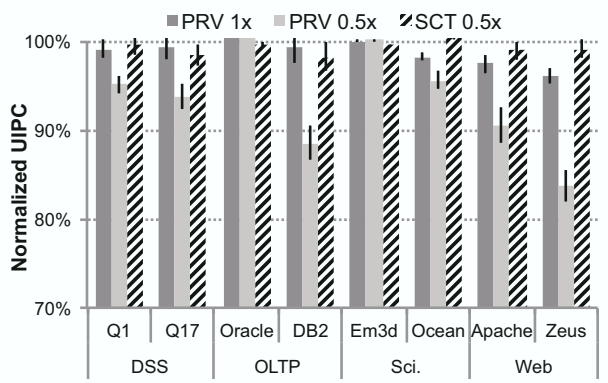


Figure 6. Comparison of performance normalized to sparse 2x.

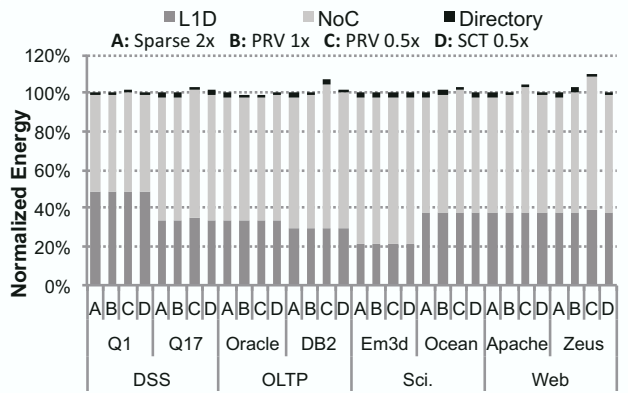


Figure 7. Energy consumption of L1D, the on-chip network, and the directory normalized to sparse 2x.

5.4. Performance Comparison

We compare SCT's performance against the prior work in the literature. The performance metric that we use is committed user instructions per cycles (UIPC) that has been shown to be proportional to the execution time for our set of workloads [21]. Figure 6 shows UIPC of PRV 1x, PRV 0.5x, and SCT 0.5x normalized to the sparse 2x baseline.

Performance results match the miss rate comparison results of Figure 5. Different workloads show different performance sensitivity to the miss rate. For example, DSS Q1 and Ocean are less sensitive to the miss rate (due to the abundance of instruction and data level parallelism in these workloads [15]) while the web server workloads show more sensitivity to the miss rate.

PRV and SCT designs show identical performance as the baseline for workloads with a private dataset (i.e., OLTP on Oracle and Em3d). However, for workloads with extensive data sharing (i.e., DSS, OLTP on DB2, Web, and Ocean) the PRV design shows significant performance loss when the directory size is decreased. SCT, on the other hand, is able to sustain almost the same performance as the baseline (1% less on average) while reducing the directory size by a factor of four. Comparing SCT with PRV, SCT 0.5x provides the same performance as PRV 1x using half the area and it has 6% higher performance over PRV 0.5x.

5.5. Comparison of Energy Consumption

Organization of coherence directories affects the processor energy consumption. Directory-induced cache invalidations cause extra traffic passing through the on-chip network and increase the cache lookups. Directories with a lower conflict rate and smaller size consume less energy.

We calculate the total energy consumption of the system for various directory organizations using McPAT [13]. To compare the results across various designs, we divide the consumed energy by the number of committed instructions to calculate energy consumption per instruction. Figure 7 shows the consumed energy per committed instruction for various directories normalized to the sparse 2x baseline. We only include L1D, the on-chip network (NoC), and the directory components in the results because the energy consumption of the rest of the system remains constant.

The directory energy consumption is negligible compared to two other components. A system with a PRV 0.5x directory consumes up to 7% more energy. The average increase in the energy consumption for PRV 0.5x is 3% in comparison with the sparse 2x baseline. Most of this increase is due to the on-chip network activity. Systems with SCT 0.5x and PRV 1x directories consume the same amount of energy as the baseline. SCT occupies four times less area compared to the baseline while does not increase the energy consumption.

5.6. Classification of Directory Accesses in SCT

SCT does not keep exact coherence information for blocks that are cached by region owners. When SCT needs to know if the owner has a copy of a block or not, it must send a probe message to the owner (see Section 4.3). These probe messages increase the latency of memory accesses and might degrade the performance if they constitute a large fraction of accesses. In practice, majority of the accesses are made to temporarily-private regions or can find coherence information for individual cache blocks in the directory. These accesses do not need a probe message because either the owner itself is accessing the region or the exact sharing information is available. In addition, SCT can avoid a probe message by checking the state of the block in LLC which can tell if the owner has a copy of the block or not.

To quantify the number of requests that experience an extra latency due to SCT, we measure the number of the SCT accesses that are made to various region types. Figure 8 shows a classification of the SCT accesses in four categories: accesses to temporarily-private regions (Temporarily-private), accesses to temporarily-shared regions that find exact block-level sharing information (Block), accesses that require sending a probe message on the critical path (Probe), and accesses that find sharing information in LLC, require a forward message even in the baseline protocol, or do not send the probe on the critical path (Other).

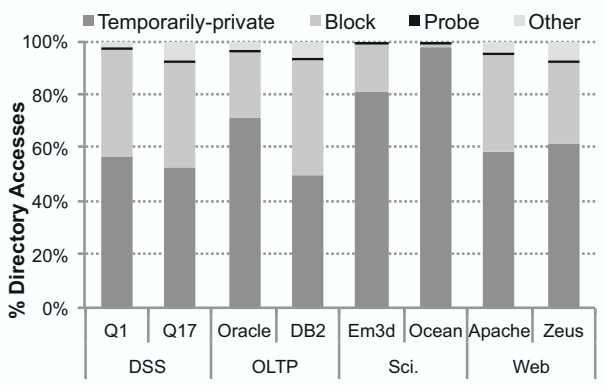


Figure 8. Breakdown of the SCT accesses based on the location where the coherence information is found.

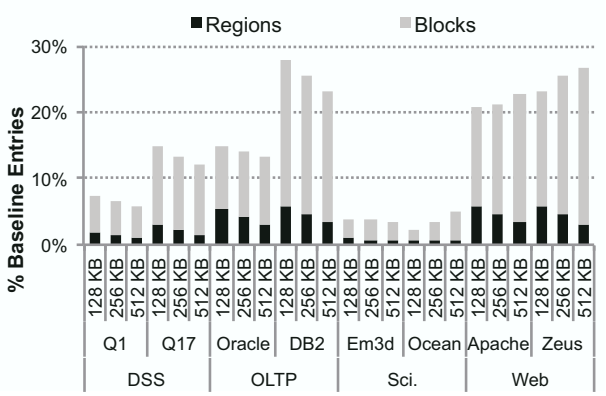


Figure 9. Directory size reduction opportunity for various cache sizes.

Majority of the accesses are made to the temporarily-private regions (66%) or to the temporarily-shared regions that have exact block-level sharing information (29%). This means that, SCT is successful as a dual-grain coherence directory, which allocates most of the directory entries in their correct positions. Moreover, the number of accesses that experience an extra delay is less than 0.5% of total accesses. Because such accesses are mainly off-chip misses, the extra latency of the probe message (i.e., a round-trip interconnect traversal) does not increase their latency by much. Therefore, the effect of these accesses on performance is negligible.

5.7. Scalability to Larger Caches

To detect temporarily-private data, SCT relies on access and eviction requests that are sent by the cores. As discussed in Section 3.3, any system parameter that increases the residency time of blocks in the cache, can potentially reduce the likelihood of data to be temporarily private. For example, larger caches retain the data for a longer period of time, which might increase the overlap of accesses from different caches to the same data and might reduce the percentage of temporarily-private data.

We measure the opportunity of using temporarily-private data to reduce the directory size for large private caches. Figure 9 shows the opportunity of reducing the directory size across various cache sizes for a region size of 8 KB. This experiment is done using a CMP with a 3-level cache hierarchy, where each core has a private inclusive L2 cache. Each bar in the figure shows the required number of entries in a SCT directory as a percentage of the number of baseline entries. The baseline is a sparse 2x directory for the corresponding cache size. We show the percentage that are taken up by region-level and black-level entries.

SCT can reduce the size of the baseline sparse 2x directory by 86% on average across all the workloads and at minimum by 75%. The opportunity increases with the cache size in all workloads except Web and Ocean. Systems with larger caches require a larger coherence directory for coherence tracking. SCT shows more opportunity in reducing the directory size as the baseline directory gets larger. In Web and Ocean, the opportunity decreases slightly because of an increase in the number of block-level entries.

6. Related Work

Techniques that have been proposed to reduce the area requirements of the sparse directories can be categorized into two broad classes: those that try to reduce the size of each directory entry and those that try to reduce the number of directory entries. Gupta et al. propose a coarse-vector representation of sharers where each bit in the sharing vector corresponds to more than one core [10]. Using a limited number of pointers to store sharers has been evaluated in [1]. When number of sharers is more than the number of pointers, one of the sharers is invalidated, sharers are kept in a coarse-vector [10], or all future accesses are broadcasted [1]; LimitLESS directory [4] traps to the software to enforce coherence on an overflow. Hierarchical directories store sharers bit-vector in a hierarchy that allocates only the necessary parts of the bit-vector [20]. Sanchez and Kozyrakis propose a design that uses a similar representation of sharers as the hierarchical directory, but in a flat organization [17]. In SCT, we store both block-level and region-level entries in the same structure. The format of block-level entries in SCT is the same as the chosen baseline. SCT does not impose any limitation on the representation of the sharers and can use any of these approaches as the baseline directory. Moreover, region-level entries have only one field that its width increases logarithmically with the core count (i.e., the owner field). In a system with a high core count, the SCT's region-level entries will not determine the width of the storage array because they are normally narrower than the baseline block-level entries.

Several techniques try to minimize the number of conflicts in sparse directories to reduce the amount of required over-provisioning. Hashing has been used to uniformly distribute addresses across all sets, thus, reducing conflicts [19]. Ferdman et al. propose using Cuckoo hashing to minimize conflicts in the directory [8]. Their proposal, Cuckoo directory, uses different hashes for different cache ways and tries to insert a

conflicting address into another cache way. Skew-associate caches [18] and ZCache [16] use a similar concept to minimize invalidations due to set conflicts. Although these techniques reduce the need for over-provisioning, none of them reduces the required number of addresses that the directory needs to track. SCT, on the other hand, reduces the number of addresses that need to be tracked in the directory by using a coarser tracking granularity. All of these techniques are orthogonal to SCT as it also can use hashing to minimize conflicts in the baseline storage array.

The most similar techniques to ours are [11] and [6], which use the OS TLB miss handler and the page table to classify shared and private pages. These proposals add a 1-bit flag to each TLB entry to indicate if the corresponding page is private or not. Cuesta et al. use this mechanism to disable coherence tracking for cache blocks that map to private pages [6]. Such mechanisms require modifications to be made in the page table organization and the OS. SCT does not need any modifications in the software. More importantly, private pages have limited opportunity at reducing the directory size [7, 11, 12]. We leverage temporarily-private data to decrease the directory size which provides significantly higher opportunity.

The concept of coarse-grain coherence tracking has been used in snoopy coherence protocols in order to avoid unnecessary broadcasts and cache lookups for non-shared data [3, 14]. RegionScout [14] and Region Coherence Array maintain a list of private regions next to each processor that is looked up to check if a broadcast is needed. The list is updated based on the messages seen on the snoop bus. We use a similar concepts in SCT to reduce the directory size in a directory-based system.

7. Conclusion

Scaling to large shared-memory chip-multiprocessors calls for scalable coherence tracking mechanisms. Sparse directories provide a scalable solution but their excessive area requirements is prohibitive in large-scale systems. Prior work reduces the required number of entries in sparse directories by avoiding coherence tracking for private data. This approach is not applicable to workloads with predominantly shared data like commercial server applications.

In this paper, we observe that data are accessed by one core most of the time, making them temporarily private. We advocate the use of temporarily-private regions to reduce the directory size. Our directory design, Spatiotemporal Coherence Tracking, uses dual-grain tracking to reduce the directory size while delivering identical performance as the baseline. It dynamically detects temporarily-private regions as the program executes and stores one entry in the directory for any number of cached blocks in those regions. For shared regions, it stores one entry per shared cache block. We show that across a variety of commercial server and scientific workloads, 77% of cache blocks map to temporarily-private pages that translates into a 75% reduction in baseline directory size. This saving is twice more than a technique that deactivates coherence for private pages.

Acknowledgements

The author would like to thank Michael Ferdman for his technical advice and feedback, Babak Falsafi for his contributions in early stages of this work, and Pejman Lotfi-Kamran and the anonymous reviewers for their feedback on earlier drafts of this paper.

References

[1] A. Agarwal, R. Simoni, J. Hennessy, and M. Horowitz. An evaluation of directory schemes for cache coherence. In *Proceedings of International Symposium on Computer Architecture*, 1988.

[2] L.A. Barroso, K. Gharachorloo, R. McNamara, A. Nowatzyk, S. Qadeer, B. Sano, S. Smith, R. Stets, and B. Verghese. Piranha: A scalable architecture based on single-chip multiprocessing. In *Proceedings of the International Symposium on Computer Architecture*, New York, NY, 2000.

[3] J. F. Cantin, M. H. Lipasti, and J. E. Smith. Improving multiprocessor performance with coarse-grain coherence tracking. In *Proceedings of the International Symposium on Computer Architecture*, June 2005.

[4] D. Chaiken, J. Kubiatowicz, and A. Agarwal. LimitLESS directories: A scalable cache coherence scheme. In Proceedings of the international conference on Architectural Support for Programming Languages and Operating Systems, 1991.

[5] P. Conway, N. Kalyanasundharam, G. Donley, K. Lepak, and B. Hughes. Cache hierarchy and memory subsystem of the AMD Opteron processor. *IEEE Micro*, 30(2):16-29, 2010.

[6] B. A. Cuesta, A. Ros, M. E. Gómez, A. Robles, and J. F. Duato. Increasing the effectiveness of directory caches by deactivating coherence for private memory blocks. In *Proceedings of the International Symposium on Computer Architecture*, 2011.

[7] A. Das, M. Schuchhardt, N. Hardavellas, G. Memik, and A. Choudhary. Dynamic directories: A mechanism for reducing on-chip interconnect power in multicores. In Proceedings of the Design, Automation, and Test in Europe (DATE), Germany, 2012.

[8] M. Ferdman, P. Lotfi-Kamran, K. Balet, and B. Falsafi. Cuckoo directory: A scalable directory for many-core systems. In *Proceeding of International Conference on High-Performance Computer Architecture*, 2011.

[9] R. Golla. Niagara2: A highly threaded server-on-a-chip. *Fall Microprocessor Forum 2006*, San Jose, CA, 2006.

[10] A. Gupta, W. Weber, and T. Mowry. Reducing memory and traffic requirements for scalable directory-based cache coherence schemes. In *Proceeding of the International Conference on Parallel Processing*, 1990.

[11] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki. Reactive NUCA: near-optimal block placement and replication in distributed caches. In *Proceedings of the International Symposium on Computer Architecture*, June 2009.

[12] D. Kim, J. Ahn, J. Kim, and J. Huh. Subspace snooping: filtering snoops with operating system support. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, New York, NY, 2010.

[13] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi. McPAT: an integrated power, area, and timing modeling framework for multicore and manycore architectures. In *Proceedings of the International Symposium on Microarchitecture*, December 2009.

[14] A. Moshovos. RegionScout: Exploiting coarse grain sharing in snoop-based coherence. *ACM SIGARCH Computer Architecture News*, 33(2): 234-245, May 2005.

[15] P. Ranganathan, K. Gharachorloo, S. V. Adve, and L. A. Barroso. Performance of database workloads on shared-memory systems with out-of-order processors. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 1998.

[16] D. Sanchez and C. Kozyrakis. The ZCache: Decoupling ways and associativity. In *Proceedings of the International Symposium on Microarchitecture*, 2010.

[17] D. Sanchez and C. Kozyrakis. SCD: A scalable coherence directory with flexible sharer set encoding. In *Proceedings of the International Symposium on High Performance Computer Architecture*, February 2012.

[18] A. Seznec. A case for two-way skewed-associative caches. In *Proceedings of the International Symposium on Computer Architecture*. NY, 1993.

[19] N. Topham and A. González. Randomized cache placement for eliminating conflicts. *IEEE Transactions on Computers,* 48(2): 185-192, 1999.

[20] D. A. Wallach. PHD: A Hierarchical Cache Coherent Protocol. *Massachusetts Institute of Technology Master's Thesis*, Cambridge, MA, 1992.

[21] T. F. Wenisch, R. E. Wunderlich, M. Ferdman, A. Ailamaki, B. Falsafi, and J. C. Hoe. SimFlex: statistical sampling of computer system simulation. *IEEE Micro*, 26(4):18–31, 2006.

[22] J. Zebchuk, V. Srinivasan, M. K. Qureshi, and A. Moshovos. A tagless coherence directory. In *Proceedings of the International Symposium on Microarchitecture*, New York, NY, 2009.

2012 IEEE/ACM 45th Annual International Symposium on Microarchitecture

Predicting Coherence Communication by Tracking Synchronization Points at Run Time

Socrates Demetriades[†] and Sangyeun Cho[‡†]
[†]Computer Science Department, University of Pittsburgh
[‡]Memory Division, Samsung Electronics Co.
{socrates,cho}@cs.pitt.edu

Abstract

Predicting target processors that a coherence request must be delivered to can improve the miss handling latency in shared memory systems. In directory coherence protocols, directly communicating with the predicted processors avoids costly indirection to the directory. In snooping protocols, prediction relaxes the high bandwidth requirements by replacing broadcast with multicast. In this work, we propose a new run-time coherence target prediction scheme that exploits the inherent correlation between synchronization points in a program and coherence communication. Our workload-driven analysis shows that by exposing synchronization points to hardware and tracking them at run time, we can simply and effectively track stable and repetitive communication patterns. Based on this observation, we build a predictor that can improve the miss latency of a directory protocol by 13%. Compared with existing address- and instruction-based prediction techniques, our predictor achieves comparable performance using substantially smaller power and storage overheads.

1. Introduction

Inter-thread communication in shared memory systems is realized by allowing different threads to access a common memory space. This model simplifies the concept of communication; however, it creates important scaling challenges mainly due to the cache coherence problem [32]. Traditionally, shared memory architectures employ either a directory- or a snooping-based protocol to keep the per-processor caches coherent. Directories maintain a full sharing state for each cache line and therefore can precisely direct each miss to its destinations. The indirection to the directory adds, however, considerable extra latency to cache misses that are serviced by other caches. Snooping protocols avoid the latency and storage overheads of a directory by resorting to broadcasting messages on each miss; however, they place significant bandwidth demands on the interconnect even for a moderate number of processors.

A common approach to improving coherence communication is to predict the processors that a coherence request must be delivered to. Accurate prediction would reduce the latency of a cache miss by avoiding indirection to the directory, or reduce the high bandwidth demands of broadcasting by using multicasting in snooping protocols. Such predictions can be made by programmers (e.g., [1]), compilers [29, 47], or transparently by the hardware [2, 3, 8, 11, 28, 30, 31, 33, 36, 39]. Given that compiler techniques are limited to static optimization [29] and that the shared memory model should be kept transparent while offering high performance [39], a preferred communication predictor would dynamically learn and adapt to an application's sharing behavior and communication patterns.

Much prior work explored coherence target prediction using address- and instruction-based approaches [2, 3, 8, 27, 28, 30, 36, 39]. Address-based coherence prediction was first proposed by Mukherjee and Hill [39], who showed that coherence events are correlated with the referenced address of a request. To exploit the correlation, they associate pattern history tables with memory addresses, train them by monitoring coherence activity, and probe them on each request to obtain prediction. Alternatively, instruction-based prediction, as proposed by Kaxiras and Goodman [28], correlates coherence events with the history of load and store instructions. This allows a more concise representation of history information since the number of static loads and stores is significantly smaller than that of accessed memory blocks.

The basic design of address- and instruction-based predictor has been extended further to mainly relax the large space requirements of those approaches [8, 30, 36, 40]. However, the extensions still require relatively large and frequently accessed prediction tables. Furthermore, to attain high accuracy, they often keep long sharing pattern history per entry or rely on multi-level prediction mechanisms. Designs that exploit the spatial locality of coherence requests, such as the ones based on macroblock indexing [36], have shown improvements for both space efficiency and prediction accuracy, indicating that predicting sharing patterns at very fine granularities is not necessarily optimal. Nevertheless, the window for capturing such opportunities is still tight to hardware-level observation, limiting the scope in which communication localities can be expressed and exploited.

In this work, we propose *Synchronization Point based Prediction* (SP-prediction), a novel run-time technique to predict coherence request targets. SP-prediction builds on the intuition that *inter-thread communication caused by coherence transactions is tightly related with the synchronization points* in parallel execution. The main idea of SP-prediction is to dynamically track communication behavior across synchronization points and uncover important communication patterns. Discovered communication patterns are then associated with each synchronization point in the instruction stream and used to predict the communication of requests that follow each synchronization point.

SP-prediction is different than existing hardware techniques because it exploits inherent application characteristics to predict communication patterns. In contrast to address- and instruction-based approaches, it associates communication patterns with *variable-length, application-defined execution intervals*. It also employs a simple history structure to recall past communication patterns when the program execution repeats previously seen synchronization points. These two properties allow a very low implementation cost and hardware resource usage, yet delivering relatively high performance. In summary, this work makes the following contributions:

• We examine the communication behavior as observed between synchronization points for various multithreaded applications (Section 3). Our characterization reveals prominent prediction opportunities by identifying (1) strong communication locality during periods between consecutive synchronization points and (2) predictable communication patterns across repeating instances of such periods.

1072-4451/12 $26.00 © 2012 IEEE
DOI 10.1109/MICRO.2012.40

IEEE computer society

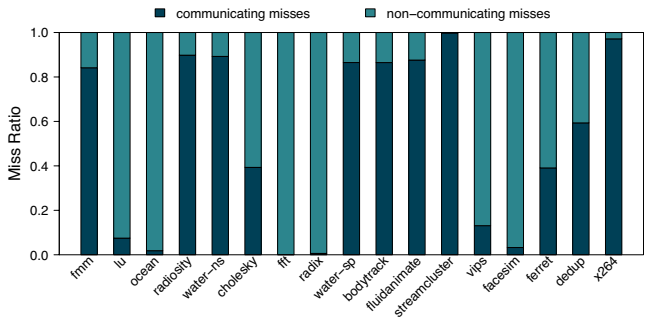


Figure 1: ***Ratio of communicating misses.*** **(Note: Details on the evaluation environment are given in later sections.)**

• We propose SP-prediction, a run-time technique to accurately predict the destination of each coherence request using a small amount of hardware resources. SP-prediction captures synchronization points at run time and monitors the communication activity between them. By doing so, it extracts a simple communication signature and uses it to predict the set of processors that are likely to satisfy coherence requests of the program interval, as well as requests that will occur in future dynamic instances of the same interval (Section 4).

• We fully evaluate SP-prediction over a directory-based coherence protocol on an elaborate chip multiprocessor (CMP) model (Section 5). Our results show that SP-prediction can accurately predict up to 75% of the misses that must communicate with other caches, without adding excessive bandwidth demands to the baseline directory protocol (below 10% of what broadcasting would add). Correct predictions translate into sizable reduction in miss latency (13% on average) and execution time (7% on average) compared to the baseline directory protocol. Compared to existing address- and instruction-based predictors, our approach achieves comparable performance, albeit at significantly lower cost.

2. Background and Motivation

Communicating misses. Coherence communication occurs on every memory request that must contact at least one other processor in order to be satisfied. Those requests, also called *communicating misses*[1], are read/write misses or write upgrades (upgrade misses) on cache blocks that have valid copies residing in non-local caches. Prior studies have shown that many applications incur a large fraction of such communicating misses [5, 36]. This fraction depends primarily on application characteristics like working set size, data sharing, and data reuse distance, as well as on cache parameters. Figure 1 shows results for the workloads studied in this work. On average, communicating misses account for 62%, with considerable variation among different applications. In general, applications with a high rate of communicating misses benefit from coherence target prediction.

Coherence communication prediction. Predicting the communication requirements of a coherence request involves guessing a set of processors *sufficient* to satisfy a given miss. A prediction scheme may exploit the communication behavior of recent misses to predict the next one, assuming that misses exhibit temporal communication locality. For instance, a prior study has shown that the two most recent destinations grab a cumulative 65% chance of sourcing the data of the next miss [25]. Communication locality is better captured, however, if misses are tracked based on the address they refer to, or the corresponding static instructions, thus motivating the address- and instruction-based prediction approaches.

Address-based prediction builds on the expectation that misses to the same address (cache block) will have to communicate with the processor that wrote on the same address previously, or the set of processors that read from the same address recently. Tracking misses in such fine granularity, however, adds significant area requirements. To reduce the overhead, a practical address-based predictor is implemented with limited capacity (i.e., as a cache), or/and indexed by blocks of larger granularity, e.g., a macroblock or page. As for the case of macroblock indexing, it has been shown to in fact improve both accuracy and space efficiency, since misses on adjacent addresses are likely to have identical communication behavior [36]. Similar in concept and motivation, instruction-based prediction resorts to the expectation that misses generated by the same static instructions will have related coherence activity. This compacts further the tracked information since the number of static load and store instructions is much smaller than the number of data addresses accessed.

The above prediction approaches are typically implemented as hardware mechanisms that consume a considerable amount of resources and are unaware of any application-level characteristics. However, the way parallel applications are coded and structured embodies intuition to create high-level understanding of how communication activity occurs and changes through time. This work examines the idea of exploiting such opportunity through the synchronization points that exist in applications.

Synchronization points. The shared memory model eliminates the explicit software management of data exchange between processors. Nevertheless, race conditions between concurrent threads require the explicit enforcement of synchronization points, through software mechanisms, to ensure that operations on shared memory locations are consistent. As a result, they naturally indicate points when certain data private to a processor will become visible—and possibly be communicated—to other processors. In what follows, we give a motivating example that shows how synchronization points partition the execution of an application into intervals, capture the existing communication locality in the application and, expose the repeatability of those partitioned intervals throughout the execution.

Figure 2 plots how a processing core communicates with other cores on a simulated 16-core CMP over (a) the whole execution (b) different execution intervals, and (c) dynamic instances of a single interval. By zooming into a granularity defined by synchronization points (plot (b)), it becomes clearer that the spatial behavior of the communication is strongly related to the specific intervals chosen. The sharp changes in communication behavior at the interval boundaries suggests that synchronization points are likely to indicate directly when behavior changes, and potentially hint a predictor to adapt faster to such changing behavior. In addition, the small set of processors that are contacted during each interval suggest that tracking the behavior on individual addresses or instructions within the interval may not necessarily result in more accurate prediction. Lastly, predictable communication patterns that may appear across the dynamic instances of the same interval (plot (c)) create a new scope of temporal predictability and a key opportunity to exploit the repeatability of the communication behavior.

To illustrate how such variations in communication behavior are manifested through shared memory programming practices, we list a simple example code in the following. Shared data (`ME` and `LE`)

[1] "Coherence request", "coherence miss", and "cache-to-cache miss" are also commonly used names.

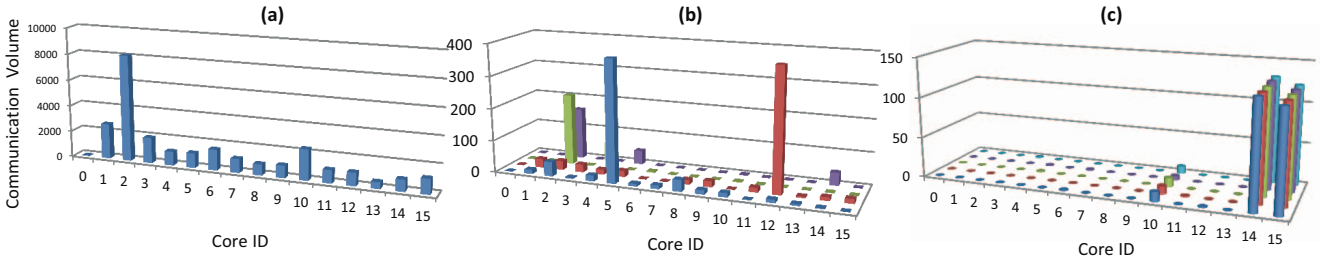


Figure 2: ***Communication Distribution of Core 0 in*** *bodytrack:* **(a) As seen during the whole execution. (b) As seen during the execution of four consecutive synchronization-defined sub-intervals. (c) As seen across five different dynamic instances of the same sync-defined interval.**

are exchanged between parents, children and siblings in a tree-like structure, which has its nodes arranged across multiple processors in a balanced way. During interval A, processors act as leafs and communicate data from processors where their parents and parents' sibling nodes reside. However, during interval B, processors act as inner nodes, hence the communication direction switches towards the set of processors that hold their children. This shift can be successfully detected and exposed by the synchronization point separating the two intervals.

Example Program Code

```
for nodes in this processor:
...
barrier();                          // interval A begins
node is a leaf:
  p = node.parent.LE[];
  for some node.parent.sibling:
    ps = node.parent.sibling.LE[];
...
barrier();                          // interval A ends
...                                 // interval B begins
node is a parent:
  for each node.child:
    node.LE[] = translate(node.child.ME[]);
...
barrier();                          // interval B ends
```

3. Communication Characterization

The communication behavior of a core over a certain interval can be characterized by the target cores with which it communicates (called *communication set*) and the *distribution of the communication volume* across that set. We have already shown examples of such distributions in Figure 2. In this section, we first introduce simple notions about synchronization point based intervals, and then we characterize the communication behavior of those intervals for various workloads.

3.1. Synchronization based Epochs

Synchronization primitives are implemented by various software libraries, often with different terminology and semantics, e.g., POSIX threads, OpenMP. Nonetheless, their range and use are similar in concept in most programming environments. We assume a POSIX thread library in this work; however, our methodology is applicable to other implementations.

A synchronization point (*sync-point*) is an execution point in which a software synchronization routine is invoked. Each sync-point has a type, e.g, `barrier`, `join`, `wakeup`, `broadcast`, `lock`, and `unlock`, and a static and dynamic ID. The static ID identifies each sync-point statically in the program code and corresponds to its calling location (e.g., program counter) or the lock address in the case of a lock sync-point. At run time, the dynamic ID uniquely identifies the multiple dynamic appearances of sync-points that have the same static ID. The

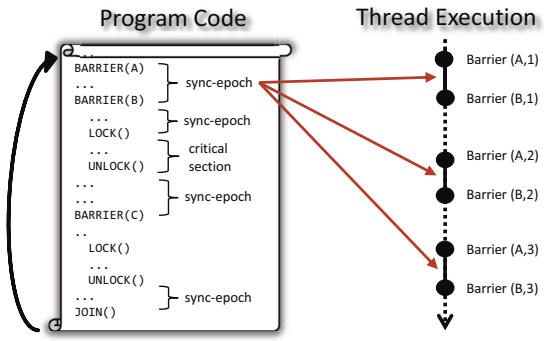


Figure 3: Static and dynamic sync-points and sync-epochs.

dynamic ID of a sync-point can be expressed with the corresponding static sync-point ID and how many times it has been executed so far.

Next, we define *synchronization epoch (sync-epoch)* as the execution interval enclosed by two consecutive sync-points. Based on this simple definition, on each sync-point, a new sync-epoch starts and the previous sync-epoch ends. A sync-epoch is described by the type, static ID, and dynamic ID of the beginning sync-point. Using our terminology, a critical section could be simply a sync-epoch that begins with a lock and ends with an unlock. A static sync-epoch that is exercised multiple times during execution creates *dynamic instances* of itself. Figure 3 depicts different sync-epochs and the notion of static and dynamic ID.

3.2. Simulation Environment

For the characterization study in this section, we employ a 16-core CMP model based on Simics full-system simulator [35]. The target system incorporates 2-issue in-order SPARC cores with 1MB private L2 cache, and a MESIF coherence protocol [23]. To track inter-core communication, we collected L2 miss traces that contain the miss data address, type, PC, and the target set of cores that must communicates with. The traces also contain all sync-points along with their type and static/dynamic IDs. Traces do not capture the effects of timing and are used only for characterization purposes. A full evaluation of our prediction scheme uses a detailed execution-driven performance model and is described in Section 5.

We study benchmarks from the splash2 and parsec suites [7, 48]. Table 1 lists key statistics related to sync-epochs for each studied benchmark. We use all available processor cores by spawning 16 concurrent threads in all experiments. For stable and repeatable measurements, we prevent thread migration by binding each thread to the first touched core. This was done except for dedup, ferret, and x264, because they create more threads than the available CPUs and rely on the OS for scheduling. Section 5.5 describes how our scheme can handle thread migration.

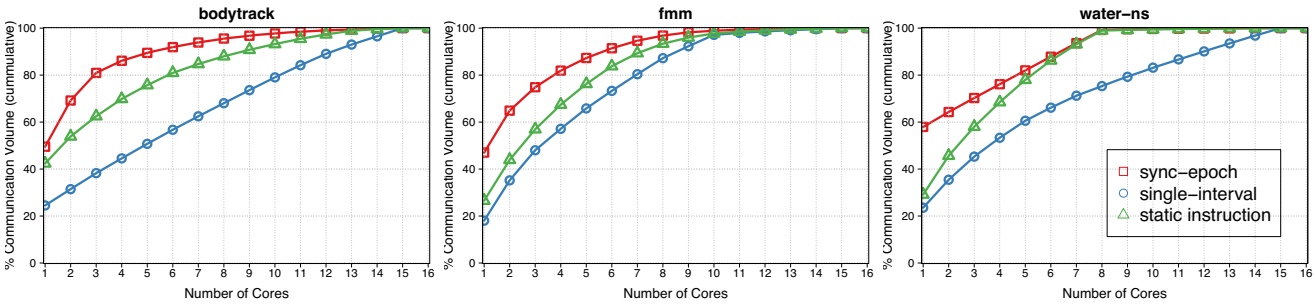


Figure 4: ***Average communication locality of*** *bodytrack, water-ns and fmm:* **Each curve shows the average cumulative communication distribution as seen in different granularities. Higher communication coverage for a given number of cores translates to better communication locality.**

BENCHMARK	# STATIC CRIT. SECT.	# STATIC SYNC-EPOCHS	PROGRAM INPUT	# TOTAL DYN. SYNC-EPOCHS
fmm	30	20	16K (particles)	2,789
lu	7	5	521 (matrix)	185
ocean	28	20	258 (grid)	2,685
radiosity	34	12	room	17,637
water-ns	20	8	512 (mol.)	1,224
cholesky	28	27	tk15.O	1,998
fft	8	8	256K (points)	22
radix	8	4	4M (keys)	35
water-sp	17	1	512 (mol.)	83
bodytrack	16	20	simsmall	456
fluidanimate	11	20	simsmall	8,991
streamcluster	1	24	simsmall	11,454
vips	14	8	simsmall	419
facesim	2	3	simsmall	3,826
ferret	4	6	simsmall	25
dedup	3	4	simsmall	508
x264	2	3	simsmall	56

Table 1: Sync-epoch statistics of benchmarks (per core average).

3.3. Communication Locality

The distribution of the communication volume characterizes the spatial behavior of the communication during an interval and illustrates whether it is "localized" to a certain set of targets. Examples of such localization are clearly observable in the communication distributions of Figure 2. For instance, core 0 during the first sync-epoch in example (b) communicates mostly with a single "hot" target, core 5, while nine other targets are contacted sporadically.

The communication locality is expressed by measuring the amount of communication volume that is *covered* by a certain number of cores. Using the previous example, core 5 covers more than 90% of the communication volume. Generally, if each individual miss communicates with C targets on average and the overall volume of the interval appears to be fully covered by C cores, then the interval has a perfect locality. When comparing different intervals with a similar C value, we can simply say that better locality exists as the communication is concentrated to fewer destinations.

A question that arises is *how good* is this locality relative to various granularities. For example, based on Figure 2(a), one could say that a certain level of locality also exists at the whole execution granularity since core 2 is "hotter" than the rest. To answer this question, Figure 4 shows the communication locality in applications, as captured by three different granularities: The sync-epoch granularity, the whole interval (as in Figure 2(a)), and the one that is based on static instruction indexing. Curves display average cumulative distributions over the whole execution and each point in the curve directly measures the average volume covered by a certain number of cores.

As the comparison shows, sync-epochs can capture the communication locality considerably better than a direct observation over the whole execution, suggesting that localities in communication's spatial behavior are closely related to sync-epochs. Moreover, sync-epochs often show better locality even to instruction-based granularity. This implies that communication activity could possibly be tracked as effectively as in traditional methods using sync-epochs—which is a much coarser-grain granularity. The results indicate that, overall, sync-epochs are attractive for extracting and exploiting repeatable communication behavior.

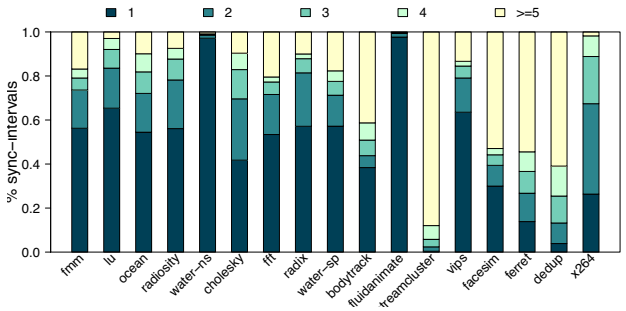


Figure 5: ***Distribution of intervals based on their hot communication set size:*** **More than 78% of intervals have a hot communication set size of smaller than or equal to 4.**

To create a representative signature of the communication behavior over each execution interval, we derive a *hot communication set* for each sync-epoch. A core is considered hot if it draws more than a certain amount of the total communication activity in the interval. Hence, the hot set could be formed based on a threshold over the communication distribution of the interval. The *size* of the set represents the amount of the interval's hot targets. Figure 5 shows, for each application, the distribution of sync-epochs based on the size of their hot communication set. The results consider a threshold of 10%, meaning that a core is considered hot if it is contacted by at least 10% of the total communication activity of the interval. In contrast to Figure 4 where only the average number of the hot communication set size is clear, the latter figure shows how this size varies among the sync-epochs of the applications. Note that to further measure how close the hot set size is to the optimal locality, one should also consider the average communication set size per miss.

3.4. Dynamic Instances of Sync-Epochs

Sync-points are executed repeatedly and create a sequence of dynamic instances for each sync-epoch. As these instances exercise the same or similar code and operate on the same (or related) data structures, it is likely that they present behavioral similarities between them [14]. Such similarities or variations may also be reflected on the communication's behavior, depending on how the shared data are accessed in each instance, their sharing patterns, the level of deter-

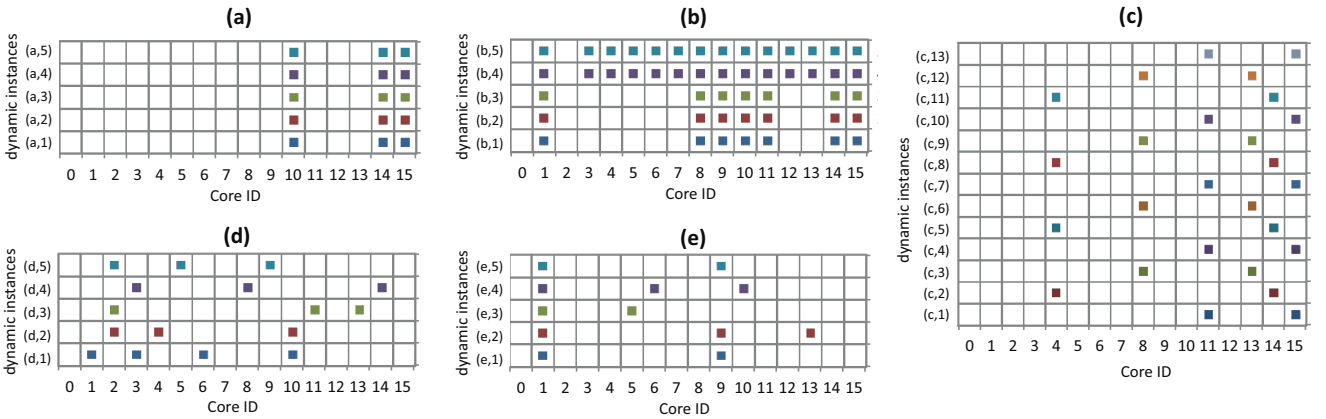


Figure 6: ***Example hot communication set patterns across dynamic instances of a sync-epoch:*** **(a) Stable pattern. (b) Change from one stable pattern to another. (c) Repetitive pattern with stride 3. (d) Random pattern (critical section). (e) Combination of stable and random hot destinations.**

minism, and possible machine artifacts, e.g., local cache capacity, false sharing effects.

Here we present our general observations on how communication activities appear in the dynamic sync-epoch instances in the examined applications. Our findings are derived from extracting the hot communication set of every dynamic instance of a sync-epoch, and characterizing how it changes from instance to instance.

Hot communication set patterns. Hot communication sets change across the dynamic instances of a sync-epoch following predictable or random pattern. We categorize the patterns into: *Stable*, *repetitive*, *random*, or some *combination* of these. Figure 6 illustrates example patterns by representing each hot communication set as a bit vector.

Stable hot communication sets occur when the majority of the data consumed each time are provided by a single core. This case is common in applications with stable producer-consumer sharing aligned to sync-epoch granularity. Hot communication sets that follow repetitive patterns are commonly found in fairly structured parallel algorithms that exercise a different but finite number of data paths on different sync-epoch iterations. For similar reasons, communication sets may also demonstrate spatial-stride or next neighbor patterns. In contrast, random patterns are usually caused by accesses on migratory and widely shared data that are produced/consumed in a non-deterministic order. Those occur when threads repeatedly compete before they are granted the privilege to produce data that will be shared (e.g., accesses within critical sections), or when the data sharing sequences are dynamically determined by the parallel algorithm (e.g., decisions made within critical sections). Patterns that appear to combine various patterns are usually an artifact caused by the granularity in which we track the communication (e.g., a long sync-epoch may span across multiple functions and data structures, each having different sharing patterns).

"Noisy" sync-epoch instances. Oftentimes, some dynamic instances of a sync-epoch appear to have very low communication activity relative to other instances. This is usually caused by a control statement, which forces specific instances to flow through different execution paths that exercise code with relatively few accesses to shared data. Such instances may not give a representative sample when forming a hot communication set due to statistical bias; therefore, we treat them as noise and exclude them from the dynamic pattern.

4. Sync-Epoch based Target Prediction

The existence of communication locality at the sync-epoch granularity implies that misses within the sync-epoch are likely to communicate with processors in the hot communication set. Thus, the hot set, if known, could be a *relatively small* and *sufficient* target predictor for the majority of misses within the interval. Based on this observation and, on evidences that many hot communication sets are predictable, we propose **SP-prediction**, a run-time scheme that exploits the temporal predictability within and across sync-epochs to predict the communication destination of misses.

SP-prediction is different from other prior approaches that exploit the temporal sharing patterns of misses in two fundamental ways. First, it makes use of the communication locality over application-specific execution intervals to predict for each miss in the interval, with no reliance on the temporal communication locality between consecutive misses. This is a significant advantage when communication locality is only seen among a broader temporal and spatial set of misses. Second, it can recall communication patterns from the past at a sync-epoch granularity and not for specific address or instruction. This may allow the predictor to adapt quickly to old and forgotten patterns without complex mechanisms and long history information.

4.1. Basic Idea of Run-Time Prediction

SP-predictor exploits sync-epochs' communication locality to predict the destinations of a miss. Each program thread is seen as a sequence of sync-epochs, many of which are exercised multiple times during program execution. Obtaining a predictor of the communication behavior in a sync-epoch involves retrieving history information from previously executed instances of the same sync-epoch, as well as tracking the coherence communication of the currently executed interval. Each private L2 cache controller would hold the obtained predictor and accelerate miss-incurred communication by invoking a prediction action in the standard coherence protocol on each miss.

Synchronization primitives are exposed to the hardware so that it can identify the sync-epochs and sense their beginning and end. This requires simple annotations in the related software library (or program code) and corresponding support in the hardware. The hardware design cost entails the addition of a new instruction that retrieves the PC or lock address of the sync-point and forwards it to the coherence controller. The insertion of the instruction in the code is trivial and could be done by the library developer or automatically by a compiler. We consider that such support is feasible in today's hardware and software, and similar implementations exist (e.g., [10, 45]).

EVENT	ACTION
Sync-point captured (*sync-epoch begins*)	- Store sync-epoch's tag and type into SP-table. - Reset all communication counters.
Data response on RD/WR-miss	If the response comes from a remote node's cache: - Increment communication-counters[responder].
Invalidation Ack responses	- Increment communication-counters[responders]
Sync-point captured (*sync-epoch ends*)	- Extract hot communication set from counters - Store the hot set as a signature to the SP-table

Table 2: Building communication signatures.

EVENT	ACTION
Sync-point captured	Retrieve d signature(s) from SP-table Obtain predictor: - If d=0 => extract current hot set (after warmup) - If d=1 => last hot set - If d=2 => last stable hot set - If d>=2 => test for pattern (if supported) - If sync-point is a lock => last d processors holding the lock Forward predictor to the L2 controller
RD/WR-miss	- Invoke a prediction action using the obtained predictor.
Confidence alert	- Extract new hot communication set - Replace predictor with new hot set

Table 3: Obtaining prediction.

4.2. Building Communication Signatures

Each processor monitors its communication activities by tracking responses to misses that have invoked the coherence protocol. A set of *communication counters* record the overall communication towards each destination. Responses for read misses include the data provider's ID and increment the communication counter that corresponds to the source processor. Responses for write and upgrade misses include a bit vector capturing the invalidated processors and increment the communication counters that correspond to the invalidated set. The communication counters are reset at the beginning of each sync-epoch. Effectively, as the execution progresses within the sync-epoch, the counters would reflect the processor's communication spatial behavior up to the current execution point within the sync-epoch. At the end of the sync-epoch, the hot communication set is extracted from the counters and stored as a communication signature (bit vector) in a history table called *SP-table*.

When the sync-epoch is a critical section, the communication signature encodes only the ID of the processor that releases the lock. This allows other critical sections that are protected by the same lock to retrieve and use this information as their possible communication target. Note that for noisy instances (Section 3.4), no communication signature is stored. Table 2 summarizes how the communication signatures are constructed during the execution.

4.3. SP-Table

SP-table is an associative table where each entry records a single, per processor, static sync-epoch. Entries are indexed/tagged with the static ID of the sync-epoch and the processor ID. For locks, entries are tagged with the lock variable and are shared by all processors. This allows all critical sections protected by the same lock (in the same or different threads) to share the same communication history.

Each SP-table entry keeps a sequence of communication signatures. This sequence has a bounded size d, the *history depth*. Whenever a sync-point is encountered, SP-table is probed to store the signature of the ending sync-epoch and retrieve the signature(s) of the next sync-epoch. Updates involve shifting out the oldest signature and shifting in the newest. For critical sections, updates occur just after the lock is acquired. This ensures atomic updates in the shared entries and avoids lookups of the table when a processor spins on a lock.

4.4. Obtaining Predictions

When a new instance of a previously seen epoch is detected, the associated communication signature(s) are retrieved from SP-table to generate a destination predictor for the misses that will occur in the new instance. The obtained predictor for the sync-epoch will be forwarded to the processor's L2 cache controller and will trigger an action to the coherence protocol on each miss. The state of the predictor would be simply the previous communication signature or some combination of previous signatures. A summary of how the predictor is formed is given in Table 3. More specifically:

No history available ($d = 0$)**.** If the sync-epoch is met for the first time (or if no history table exists), then history information is not available. In this case, the predictor uses a hot communication set that is extracted from the communication counters while the sync-epoch runs, after allowing some warm-up time, e.g., 30 misses. This would essentially form a predictor that predicts requests based on the activity recorded in the early stages of the interval.

Last hot communication set ($d = 1$)**.** If only one history signature is available so far (or if the table has history depth of one), then the predictor uses the last—and only available—communication signature stored in the corresponding predictor entry.

Last stable hot communication set ($d = 2$)**.** The intersection between communication bit vectors (bit-wise AND) returns the set of destinations that remain stable across the instances. Our predictor combines only the two most recent bit vectors, since this successfully catches stable destinations across consecutive instances, as well as adapts faster to changing stable patterns such as the one shown in Figure 6(b).

Pattern-based hot communication set ($d \geq 2$)**.** A longer history of signatures available to a sync-epoch could capture further hot communication set patterns such as the repetitive pattern shown in Figure 6(c). Specifically, to capture such repeatable patterns, history depth should be at least as large as the repetition distance (or stride) of the pattern, e.g., $d \geq 3$ for the same example. Hardware could detect a repetitive pattern by comparing a new bit vector with all the stored bit vectors, saving the depth s of the one that matches, and correctly predicting the next bit vectors using the one at depth $s-1$. Our current predictor is tuned to detect only repetitive patterns of stride-2, as it uses a history depth of no more than two.

Lock sync-point. If the captured sync-point is a `lock`, then the retrieved signatures will indicate the sequence of processors holding the lock last. A union of the available d signatures will therefore form a prediction set that includes the last d processors that have held the lock. The predictor may be further extended to return a union that also includes the bit vector of the preceding sync-epoch, as coarse critical sections are likely to benefit from it.

In order to detect and recover from pathological cases where the predicted communication set does not provide correct prediction, we employ a mechanism that sense low prediction confidence and adapts to a new hot communication set. A recovery step is usually needed in coarse sync-epochs, where communication's spatial behavior could oscillate within a sync-epoch instance. In our current design, the confidence mechanism is a simple 4-bit saturating counter that increments on correct predictions and decrements otherwise. On each new interval, the counter starts with a high confidence towards the predicted communication signature (counter is fully set) and triggers a recovery step if the confidence level drops below a threshold (counter is zero). To recover, we reconstruct the predictor by extracting the hot communication set of the currently running interval, as it appears up to the current point. The hot set is extracted based on the information recorded in the communication counters that dynamically track the

communication activity of the interval.

4.5. Integration to the Coherence Protocol

SP-prediction requires additional functionality in the coherence protocol. However, it does not interfere with the base protocol and operates on top of it. We briefly describe how our protocol arbitrates prediction actions, verifies results, and recovers from mispredictions.[2] As a baseline protocol, we use a directory-based MESIF coherence protocol, an extended version of MESI that effectively supports cache-to-cache transfers of clean data [23]. Note that the prediction engine can be integrated into any directory-based protocol, or any snoop-based protocols that can recover from mispredictions [8, 36].

• Requesting node: When an L2 miss for a memory line occurs, a prediction request is generated. The request is sent to the node(s) predicted to have the valid copy of the line and includes a bit identifying it as predicted. The request is also sent to the directory along with a bit vector identifying the predicted nodes.

• Directory: The directory node will receive the bit vector of predicted nodes for every miss and detect whether the targeted set was sufficient or not. Upon detecting a misprediction, it will satisfy the request as it would normally do, resulting in a miss latency similar to the baseline protocol. If the request was for upgrade or write miss with multiple sharers, the directory will invalidate the sharers that were not predicted (if any), and reply to the requesting node, indicating whether the predicted set was sufficient or not and which sharers were correctly predicted.

• Predicted node: When a predicted request for a memory line arrives at the cache controller, the line is searched in the L2 cache. If the line is in Exclusive, Modified, or Forwarding state [23], then a copy of the line is immediately forwarded to the requesting processor. Also, an update message is sent to the directory indicating the new sharing state of the cache line. If the line must be invalidated (i.e., due to request for exclusive ownership), an Ack message is sent back to the requesting processor after invalidation. Otherwise, the cache replies with a Nack message.

• The requesting node will receive responses from the predicted nodes, and also from the directory in case the request was for exclusive ownership (write or upgrade miss). Upon receiving data, the controller will perform line replacement as usual and, if the request was a read, the miss will be completed. If the request was for exclusive ownership, then it will be completed only after the response from the directory and the necessary invalidation Acks from the correctly predicted sharers have arrived (if any). Given that the directory is always aware of the prediction result and can proceed as normal on mispredictions, it is unnecessary for the requesting node to reissue requests.

4.6. Discussion on SP-Table Implementation

SP-table can be implemented either in system software or hardware. In the former case, the table is statically allocated at boot time by the OS and kept at a certain memory location. Every sync-point will invoke a trap to the OS, which will handle all necessary operations on SP-table and return a predictor for the upcoming sync-epoch. In a hardware embodiment, a slice of SP-table can be integrated with the L2 cache controller on each processor and hold the information specific to that processor. Table entries that are shared by all processors (for lock sync-points) could be either located at a centralized location on chip, or distributed across the slices in an address-interleaved fashion. All implementations assume that the sync-point's PC, lock address and the processor ID can be extracted at the processor, and the necessary information can be piggybacked and transferred between the hardware and software components involved.

[2]More details on how the protocol handles race conditions and conflicts can be found in similar extensions [2, 3, 8, 43].

Parameter	*Value*	*Parameter*	*Value*
Proc. model	in-order	L1 I/D Cache	
Issue width	2	Line size	64 B
L2 Cache (private)		Size/Assoc.	16 KB, 1-way
Line size	64 B	Load-to-Use lat.	2 cycles
Size/Assoc.	1 MB, 8-way	Network-on-Chip	
Tag latency	2 cycles	Topology	4×4 2D mesh
Data latency	6 cycles	Router	2-stage pipeline
Repl. policy	LRU	Main mem. lat.	150 cycles

Table 4: Simulated machine architecture configuration.

SP-table has fairly low space requirements. Each slice requires as many entries as the number of static sync-points in an application, which is generally small ($\leq 30 + 2\sim3$ entries as shared portion). Each entry may hold more than one signatures, depending on the history depth (we allow no more than two in our evaluation). The length of the signature (in bits) is equal to the number of processors (e.g., 16-bits for a 16-core CMP). Each entry also has a 32- or 64-bit tag (PC) depending on the machine's architecture and an additional bit indicating whether the entry is shared, i.e., a lock. Although each SP-table slice is considered to work as fully-associative, a smaller set-associativity array is also possible without much cost from set conflicts. A 2 KB aggregate SP-table is adequate to hold all necessary information for even the most demanding applications (including 32-bit tags). As we will discuss later in Section 5, this size is significantly smaller compared to address- or instruction-based tables.

The location and management of SP-table is an implementation choice that has no significant performance implications, since it is small and accessed relatively infrequently (only on sync-points). A hardware implementation would generally be more appropriate if sync-epochs are short, e.g., the application has very fine-grain locking. In general, the SP-table design should be dictated by both the design goals and the target application domain.

5. Evaluation

5.1. Methodology

To evaluate the performance of the proposed predictor, we extend the system described in Section 3.2 with detailed timing models for cache hierarchy and interconnect. The target system is a 16-core tiled CMP with a 4×4 2D mesh network-on-chip (NoC), similar to models used in recent studies and commercial developments [12, 46]. Each tile incorporates a processor core that has two levels of private caches, coherence logic, and a NoC router. Coherence is maintained through a distributed directory-based MESIF coherence protocol with some extensions as described in Section 4.5. The NoC operates at the processor core frequency and is a wormhole-switched network with deterministic X-Y routing and Ack/Nack flow control. Table 4 summarizes our architecture configuration parameters.

For the SP-table, we consider a distributed hardware implementation and each entry can hold no more than two signatures ($d = 2$). The SP-table is accessed only on sync-points and the access latency is rarely in the critical path. Updates on communication counters complete in a single cycle, and we account four cycles for extracting a hot communication set. We present the performance of the SP-predictor with respect to the baseline directory protocol and a

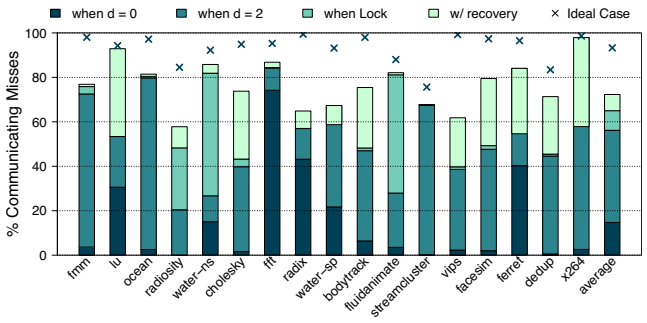


Figure 7: ***SP-prediction accuracy:*** **Percentage of communicating misses that avoid indirection to the directory.**

broadcast protocol. Results consider both serial and parallel sections, although the predictor is effective only during parallel sections. To fairly evaluate a broadcast snoop-based protocol, we assume a totally ordered interconnect with the same configuration as the one with directory. At the end, we compare our prediction approach against a simple locality-based predictor and state-of-the-art address- and instruction-based destination set predictors [36].

5.2. Prediction Effectiveness

Prediction is correct when the predicted set is sufficient to satisfy a communicating miss, i.e., a superset of the sharing information in the directory. The *size* of the predicted set—which is the size of the hot communication set in our case—creates a trade-off between prediction accuracy and bandwidth waste. The fewer the cores included in the predicted set, the less the probability to communicate with the correct cores(s) for each request. On the other hand, the more cores in the predicted set, the more redundant messages will be sent, and hence the more bandwidth will be added on the interconnect. In our evaluated scheme, the size of the hot communication set depends on the communication locality of each sync-epoch as explained in Section 3.3, and adapts to the changing communication patterns as described in Section 4.4.

Figure 7 shows the percentage of communicating requests predicted correctly. On average, the SP-predictor correctly predicts and eliminates indirection to the directory for 77% of all communicating requests, with 98% (x264) and 59% (radiosity) as the best and the worst case, respectively. The crosses indicate the accuracy that the SP-predictor could obtain ideally, if the hot communication set for each sync-epoch was known a priori. The gap between the actual and the ideal accuracy comes from the lack of predictability in some sync-epoch instances and the sensitivity level of the recovery mechanism. This gap may be bridged somewhat if off-line profiling offers initial prediction information and the sensitivity level is adjusted dynamically.

The percentage breakdown indicates the prediction accuracy when different information was available to the SP-predictor. The bottom stack accounts for correct predictions made when no information from past sync-epoch instances was available. Such situations appear in applications where major sync-epochs are not replayed (fft, radix and ferret). In those cases, the predictor relies mostly on most recent within-interval communication activity to predict miss targets. The next two stacks correspond to misses correctly predicted based on signatures from past sync-epochs, indicating separately those occurring within critical sections. Applications with highly repeatable sync-epochs such as ocean and streamcluster can take advantage of the pattern-based prediction policy. Similarly, applications with fine-locking such as water-ns and fluidanimate gain with highly accurate predictions due to the ability of our predictor to retrieve the random sequence in which threads execute the critical sections. On average, those sync-epoch history-based predictions account for up to 40% in prediction accuracy. Sync-epochs with unpredictable intervals will eventually adapt their predictors based on the recovery mechanism and correctly predict an additional 9% of requests on average.

BENCHMARK	AVG. ACTUAL TARGETS PER REQ.	AVG. PREDICTED TARGETS PER REQ.	RATIO OF PREDICTED TO ACTUAL
fmm	1.19	3.11	2.61
lu	1.01	2.46	2.46
ocean	1.08	3.15	2.94
radiosity	1.11	4.12	3.71
water-ns	1.41	2.53	1.80
cholesky	1.04	1.89	1.83
fft	1.01	2.37	2.36
radix	1.00	2.75	2.75
water-sp	1.58	2.75	1.75
bodytrack	1.13	2.8	2.49
fluidanimate	1.14	2.05	1.79
streamcluster	1.14	1.95	1.72
vips	1.01	2.06	2.05
facesim	1.04	2.56	2.47
ferret	1.01	1.14	1.13
dedup	1.10	2.34	2.15
x264	1.01	1.93	1.93

Table 5: Average actual and predicted set size.

Messages will be wasted if the predicted target set for a miss is incorrect, or larger than the minimum sufficient target set. Table 5 summarizes the differences between the minimum and the predicted average target set size. The minimum sufficient set size is generally close to 1 since read requests—which are the majority—must always contact only a single destination.[3] By comparing separately the reads and writes, we found that, on average, the predicted set includes 1.4 and 0.5 more targets per request respectively. More insight on how the prediction affects the bandwidth demands is given by more detail results presented later in this section.

The way the hot communication set is extracted (Section 3.3) strongly affects the trade-off between latency and bandwidth. The current policy leads to some bias towards higher bandwidth when the locality is poor, since there are no strict bounds on the maximum size of the set. In general, the policy can be tuned depending on the design goals and requirements. For example, in a case where bandwidth demands must be bounded to avoid exceeding a power envelope, one could tune the policy to extract a hot set that does not exceed a certain size.

5.3. Performance Results

Impact on miss latency. Correct predictions will satisfy misses without paying the cost of indirection to the directory, thereby reducing the average cache miss latency. Incorrect predictions are detected by the directory, which will then satisfy the miss without noticeably degrading the latency of the indirected miss. Figure 8 shows the average miss latency achieved by the SP-predictor and the baseline protocols. Average latency is calculated by treating each miss individually, and results are normalized to the directory protocol. The results show that on average, SP-prediction reduces miss latency by 13% relative to the directory protocol and attains up to 75% of what the broadcast snooping protocol can achieve. Under the (true) assumption that the NoC does not get severely congested, the broadcast scheme approximates the ideal case in terms of miss latency.

The predictor predicts correctly and reduces the latency for both read and write requests. A correctly predicted "read" has slightly

[3]The reported numbers assume a cache-to-cache transfer request for clean data to have a sufficient set size of 1, which is not necessarily true in a MESIF protocol [23].

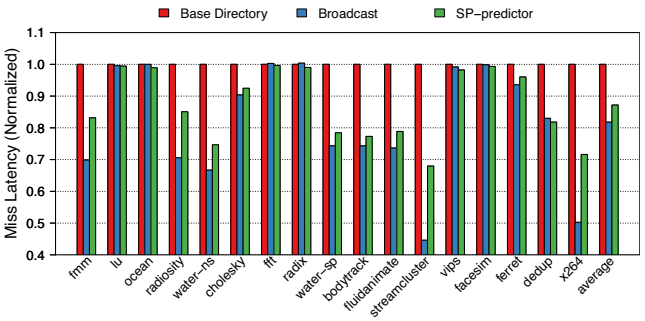


Figure 8: Average miss latency. (Note: Y-axis starts at 0.4.)

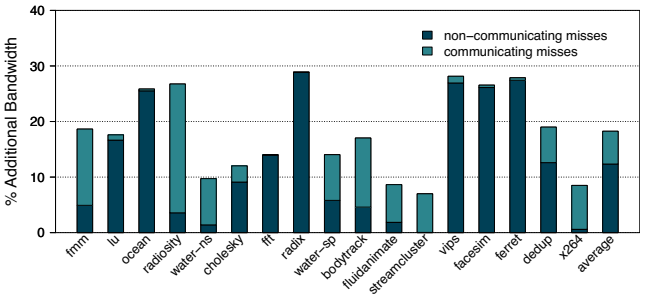


Figure 9: Additional bandwidth demands of SP-prediction relative to the base directory protocol.

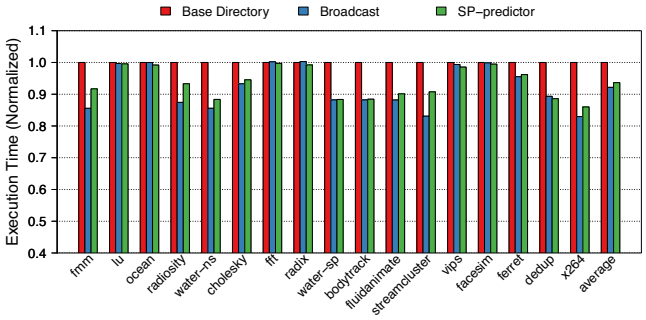


Figure 10: Execution time. (Note: Y-axis starts at 0.4.)

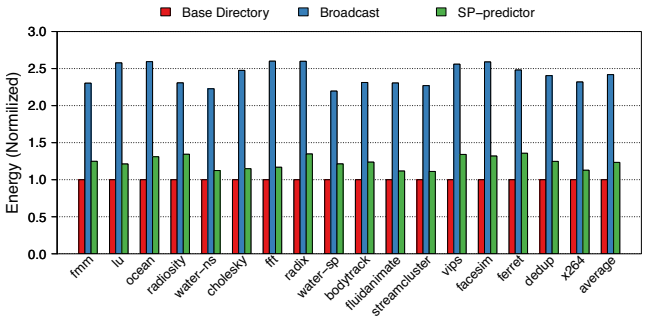


Figure 11: Energy consumed on NoC and cache lookups.

higher impact compared to a correctly predicted "write", as writes may have multiple targets to reach and wait for acknowledgments. Also, the prediction accuracy slightly declines as the number of the targets increases. Nevertheless, write requests with multiple targets are generally a small fraction of the overall misses, and their impact on the overall reductions in latency is limited.

Marginal improvements in some applications (e.g., lu, radix) are due to the limited fraction of communicating misses (recall Figure 1). The smaller this fraction is, the fewer the opportunities for latency reduction. Moreover, the high miss latency of non-communicating misses (i.e., off-chip misses) will, in the end, overshadow the improvements coming from accelerating on-chip, communicating misses. A quick look at how this fraction varies across the applications directly explains why the miss latency reduction is limited for each application. Note that this also limits the effectiveness of the broadcasting scheme. It is generally possible for a larger cache size to elevate the fraction of communicating misses for memory bound applications, and hence increase the impact of the predictor to the miss latency reduction. Sensitivity analysis of cache parameters and workload input sizes (not reported in this work) have shown expected observations and trends.

Impact on bandwidth requirements. To measure the impact of target prediction on bandwidth, we track the number of bytes transmitted on the NoC due to L2 cache misses. These include request messages to predicted cores, request and update messages to the directory, and control and data responses. Figure 9 shows the additional average bandwidth requirements of a coherence request, relative to those of the baseline directory protocol. The results show that SP-prediction increases the bandwidth requirements by 18% compared to the baseline. The snooping protocol would have the highest bandwidth demands since messages are broadcast to all targets on each miss, whereas the directory protocol essentially approximates the ideal case possible. Overall, SP-prediction keeps its additional bandwidth requirements below 10% of what the broadcasting protocol would additionally demand to the baseline directory protocol (the actual bars for broadcasting are not shown due to the very large difference).

Much of the additional bandwidth comes from the (always unfortunate) attempts to predict non-communicating misses. This portion is shown by the bottom stack and accounts for 70% of the overhead. Applications with a large fraction of non-communicating misses will therefore increase the bandwidth demands with no positive return in latency. Prior work has shown that most of such attempts can be detected and avoided by simple snoop filtering [38]. For example, a simple low cost TLB-based snoop filter can detect ∼75% of them [17]. Thus, the use of orthogonal techniques can substantially reduce the associated bandwidth overheads without compromising the latency improvements.

Impact on execution time. Figure 10 depicts the overall improvements in execution time as a result of reducing miss latency. SP-prediction improves the execution time by 7% on average, with x264 seeing the best improvement (14%). Depending on the interconnect design and control parameters, an excessive traffic could congest the network and affect the performance negatively. In our simulated system, congestion levels remain low for both, the prediction-augmented directory protocol and base broadcast protocol. Marginal negative impact was observed for broadcasting only in applications with very small fraction of communicating misses.

Impact on energy. We estimate the energy impact of SP-prediction using an intuitive analytical model that considers the dynamic energy consumed on the interconnect and L2 cache snoops. For the network, we assume that the energy consumed is proportional to the amount of data transferred [4]. We also assume that the energy consumed in a router is four times that consumed in the link. For cache snoops, a single cache tag lookup energy is estimated using CACTI [21], assuming a 32nm technology. Figure 11 presents the normalized

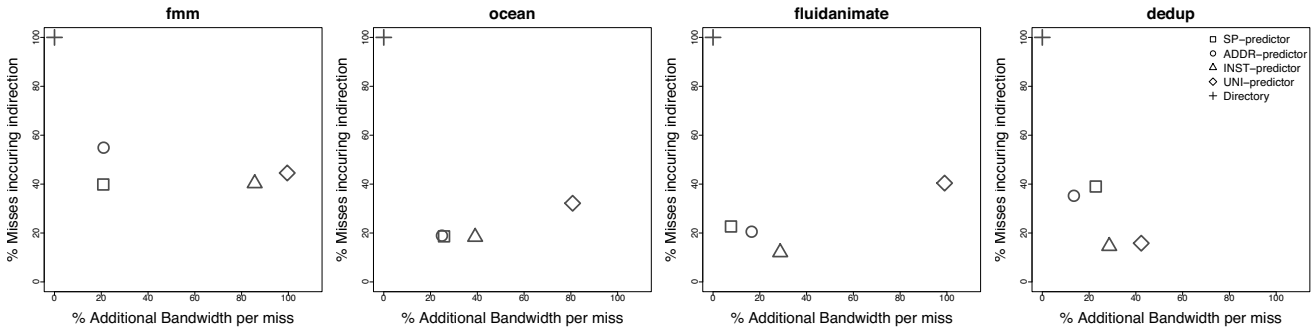


Figure 12: ***Performance/bandwidth trade-off comparison:*** **The lower-left corner represents the best point on the trade-off space. The results are expressed relative to the directory-based protocol, which is indicated with a "cross" symbol at the upper-left corner.**

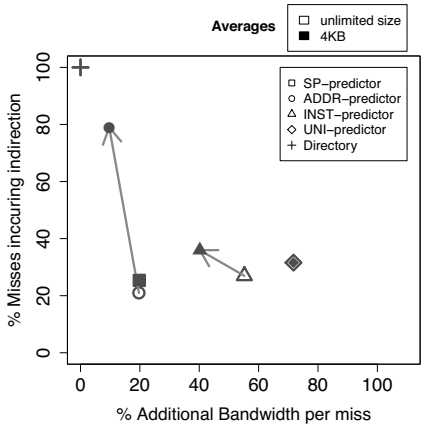


Figure 13: ***The effect of space requirements to prediction performance:*** **SP-prediction and UNI-prediction are not affected since they have significantly lower space requirements.**

results. Enabling SP-prediction over a directory protocol increases the energy requirements on network and cache lookups by 25% in total. Yet, this is substantially less compared with the energy requirements of snoop broadcasting (2.4×). Considering that a large fraction of traffic and snoop overhead could be filtered, as discussed previously, the new energy demands could be brought down to below 8%.

5.4. Comparison with other Predictors

We compare SP-prediction with address- and instruction-based prediction, implemented according to the "group" destination set prediction model proposed by Martin et al. [36]. In addition, we compare with a simple locality-based predictor that uses no index, i.e., predicts simply based on the coherence activity of previous misses, independent of their address or instruction. For abbreviation we will refer to them as ADDR-, INST-, and UNI-prediction, respectively. The ADDR and INST prediction models use both external coherence requests and coherence responses to train a predictor for each data block or instruction. The UNI predictor uses only the coherence responses, i.e., it is trained based on the targets of previous misses by the same core.

All the predictors return a group of possible sharers, aiming high prediction accuracy while making best efforts to keep the bandwidth requirements small[4]. Each predictor entry incorporates a two-bit counter per core that accumulates the recent activity towards each destination, and a train-down mechanism which ensures that the predictor eventually removes inactive destinations [36]. For a 16-core machine, each group predictor entry requires a total of 37 bits (tag not included): 32 bits for the train-up counters and a 5-bit roll-over counter for the train-down purposes. For SP-prediction, we consider an SP-table with two signatures per entry (total of 33 bits) as a fair setting for comparison. Note that SP-prediction requires also a set of communication counters (1-byte each) and a predictor register, which account for a fixed cost of 17 bytes per core.

Each predictor represents a point in the trade-off between latency and bandwidth. To effectively visualize this trade-off, we plot results on a two dimensional plane (Figures 12, 13). The horizontal dimension represents *request bandwidth per miss* (as additional to that of the based directory). The vertical dimension represents *latency*, measured as the *percent of misses that require indirection*. The chosen metrics provide a desirable level of detail for deriving insightful results for the performance of the predictors under consideration.

Figure 12 displays the results for the four predictors in four different applications for illustration. The results assume predictors with an infinite number of table entries for their indexed tables, i.e., they do not consider space efficiency. Overall, SP-prediction lays in the trade-off plane comparably to address- and instruction-based prediction. Among the examples, fmm presents a case in which SP-prediction outperforms all other predictors, achieving both higher accuracy and lower bandwidth. In contrast, dedup presents a counter case, where SP-prediction is weaker on the accuracy dimension. Accuracy levels between ADDR and INST appear to be similar, with the ADDR-predictor having more tendency towards lower bandwidth requirements. UNI-prediction is shown to have lower accuracy, which also negatively affects the bandwidth demands since incorrect predictions place unnecessary messages on the interconnect.

Each scheme has, however, a very different space demands to meet the illustrated maximum performance. A perfect ADDR-prediction scheme suggests storage requirements in proportion to the size of the memory blocks, which is prohibitively large. Common practice is for ADDR to consider, instead, predictors per macro block (e.g., 256-bytes in our implementation). This reduces the maximum space requirements, and also improves further the predictor by capturing spatial locality. However, even with macro-blocks, the number of entries required to achieve the maximum performance is in the order of Kilo. INST has been promoted for its low storage needs; however, it requires significantly more table entries than the SP-table (equal to static load/stores). UNI-prediction requires only a single prediction entry and represents the cheapest possible solution. The SP-prediction's storage requirements are inherently bounded by the

[4]Other prediction policies such as "owner" or "group/owner" can also be used and fairly compared as far as all predictors are tuned to the same base policy.

number of static sync-points of the application as shown in Table 1. This corresponds to substantially lower space demands compared to ADDR and INST. Assuming that the SP-table is easily implementable in the software layer, its hardware space requirements can be largely eliminated, reaching those of UNI-predictor.

To evaluate the sensitivity of the predictors to space requirements, we implement them with limited number of table entries. Figure 13 compares the performance of different predictors when table entries go from unlimited to a finite number of 512 (∼4KB of storage space). To simplify the illustration, we show only the average results for each predictor, over all the studied applications. The results indicate that limited space yields lower accuracy for ADDR and INSTR compared to SP-prediction. Nonetheless, they present a corresponding decrease in bandwidth, since prediction is attempted on fewer misses.

The prediction performance per space requirements is in a sense the measure of how well the prediction information is encoded, or in other words, the measure of a predictor's space efficiency and cost. Considering that SP-prediction requires significantly smaller storage than ADDR and INST, we argue that the reported small performance differences are insignificant when space and power requirements are a primary design constraint, as is the clear case in modern and emerging CMP implementations [19]. In conclusion, from the space requirements perspective, an SP-predictor with ∼256 entries can achieve performance equivalent to INST with ∼1K entries, or macro-block ADDR with ∼8K entries, on average.

5.5. Discussion

Predictor's power consumption comparison. Prediction tables consume static and dynamic power. Static power is proportional to the table size, which is substantially smaller with SP-prediction. Dynamic power is primarily affected by the associativity, and the access frequency of the predictor tables. While the ADDR and INST access their tables on every miss, SP-predictor keeps the prediction set in a single register, and accesses the SP-table for updates only on sync-points. This directly translates into power savings. Based on an overall observation, the SP-table would be accessed once for every ∼300 accesses of an ADDR- or INST-based table.

Thread migration. So far we have assumed that communication signatures and predictors consist of bit vectors representing target physical cores. If thread movements are allowed between cores, then those representations should track a "logical core-ID" (e.g., thread-id) rather than physical ID. The logical-to-physical destination mapping must be known at the core side, and could be applied before or after the formation of the predictor, depending on the coherence controller implementation.

Projections for commercial workloads. Database, server, and OS workloads are mostly based on lock synchronization and as a result have less regular and predictable communication patterns [42]. The proposed SP-predictor can effectively predict the communication activity within critical sections since it can retrieve communication signatures on lock points that include the cores (or the sequence of cores) holding the lock previously in time. Results from applications with a high count of critical sections (e.g., fluidanimate and water-ns) show high prediction accuracy for the misses occurring within critical sections (Figure 7). Therefore, although we have not performed experiments on such workloads, we expect our predictor to work reasonably well.

6. Related Work

Address and instruction-based indexing have been the basis of hardware coherence predictors [27,28,31,39]. In the context of destination set prediction, Acacio et al. [2] studied a two-level owner predictor where the first level decides whether to predict an owner and the second level decides which node might be the owner. In a similar work, they study a single-level design to predict sharers for an upgrade request [3]. Bilir et al. [8] studied multicast snooping using a "Sticky Spatial" predictor. Martin et al. [36] explored different policies for destination set predictors to improve the latency/bandwidth trade-off under ordered interconnects. Other studies have further explored the impact of predictor caches [40] and perceptron-based predictors [34].

There have been numerous other efforts to improve coherence performance. Many protocols were developed or extended to optimize for specific sharing patters, such as pairwise sharing [22], migratory sharing [13,44], producer-consumer sharing [11] and some mix of those [20]. Dynamic self-invalidation was proposed to eliminate the invalidation overhead [31,33]. Alternatively, software-driven approaches have proposed programming models or utilized compilers to effectively *prefetch* or *forward* shared data to reduce miss latencies [1,29,47]. A thorough characterization of data sharing patterns and inter-processor communication behavior in emerging workloads is presented in a work by Barrow et al. [5].

More recent work has exploited properties relevant to CMP architectures to accelerate coherence, such as core proximity and fast and flexible on-chip interconnect. Brown et al. [9] describe an extension to the directory-based coherence protocol where requests are first sent to neighboring cores. Barrow et al. [6] propose adding new dedicated links for forwarding the requests to the nearby caches, delegating directory functions in case of proximity hits. Various other proposals, such as Token Coherence [37], examine novel approaches on maintaining coherence in unordered interconnects without requiring directory indirection. Eisley et al. [16] propose to embed directories within the network routers that manage and steer requests towards nearby sharers. Jerger et al. [18] propose a virtual tree structure to maintain coherence in an unordered interconnect, with the root of the tree acting as an ordering point for requests. In Circuit-Switch Coherence [25], the same authors show how coherence predictors can leverage existing circuits to optimize pairwise sharing between cores. Similar to virtual tree coherence, DiCo-CMP [41] delegates directory responsibilities to the owner caches.

Synchronization points have also been utilized by other recently proposed techniques to direct hardware-level optimization. In BarrierWatch [14], the authors identify the relation between barriers and time-varying program behavior and propose the use of this relation to guide run-time optimizations in CMP architectures. Under the MPI model, Ioannou et al. [24] propose tracking MPI calls to guide phase-based power management in Intel's MPI Cloud processor research prototype. In heterogeneous architectures, locks and other synchronization points may trigger scheduling/migration actions to accelerate critical sections [45] and other critical bottlenecks [26]. Work on memory scheduling for parallel applications has also made use of loop-based synchronization to effectively manage inter-thread DRAM interference [15]. Lastly, exposing shared-memory synchronization primitives to the hardware has been the underlying support for software based coherence enforcement, e.g., [10].

7. Conclusions

This paper proposed and studied *Synchronization Point based Coherence Prediction* (SP-Prediction), a novel run-time technique for predicting communication destinations of misses in cache-coherent shared-memory systems. SP-prediction employs mechanisms that capture synchronization points at run time, track the communication

activity between them, and extract simple communication signatures that guide target prediction for future misses. SP-prediction is substantially simpler than existing techniques because it exploits the inherent characteristics of an application to predict communication patterns. Compared with address- and instruction-based predictors, SP prediction requires smaller area and consumes less energy while achieving comparative high accuracy. We anticipate that the synchronization point driven prediction approach could be applicable to further communication optimization cases, and this work will be basis for future investigation towards this direction.

Acknowledgments

We thank our shepherd Prof. Milos Prvulovic, members of Pitt's XCG (formerly CAST) group, and the anonymous reviewers for their constructive comments and suggestions. This work was supported in part by the US NSF grants: CCF-1064976, CCF-1059283 and CNS-1012070.

References

[1] H. Abdel-Shafi *et al.*, "An evaluation of fine-grain producer-initiated communication in cache-coherent multiprocessors," in *Proc. of the 3rd IEEE Symp. on High-Performance Computer Architecture*, 1997.

[2] M. E. Acacio *et al.*, "Owner prediction for accelerating cache-to-cache transfer misses in a CC-NUMA architecture," in *Proc. of Conf. on Supercomputing*, 2002.

[3] ——, "The use of prediction for accelerating upgrade misses in CC-NUMA multiprocessors," in *Proc. Int'l Conf. on Parallel Architectures and Compilation Techniques*, 2002.

[4] A. Banerjee *et al.*, "An energy and performance exploration of network-on-chip architectures," *IEEE Trans. Very Large Scale Integr. Syst.*, 2009.

[5] N. Barrow-Williams *et al.*, "A communication characterisation of SPLASH-2 and PARSEC," in *Proc. Int'l Symp. on Workload Characterization*, 2009.

[6] ——, "Proximity coherence for chip multiprocessors," in *Proc. Int'l Conf. on Parallel Aarchitectures and Compilation Techniques*, 2010.

[7] C. Bienia *et al.*, "The PARSEC benchmark suite: characterization and architectural implications," in *Proc. Int'l Conf. on Parallel Architectures and Compilation Techniques*, 2008.

[8] E. E. Bilir *et al.*, "Multicast snooping: a new coherence method using a multicast address network," in *Proc. Int'l Symp. on Computer Architecture*, 1999.

[9] J. A. Brown *et al.*, "Proximity-aware directory-based coherence for multi-core processor architectures," in *Proc. Int'l Symp. on Parallel Algorithms and Architectures*, 2007.

[10] J. B. Carter *et al.*, "Implementation and performance of munin," in *Proc. Int'l Symp. on Operating Systems Principles*, 1991.

[11] L. Cheng *et al.*, "An adaptive cache coherence protocol optimized for producer-consumer sharing," in *Proc. of the Int'l Symp. on High Performance Computer Architecture*, 2007.

[12] S. Cho and L. Jin, "Managing distributed, shared l2 caches through os-level page allocation," in *Proc. Int'l Symp. on Microarchitecture*, 2006.

[13] A. L. Cox and R. J. Fowler, "Adaptive cache coherency for detecting migratory shared data," in *Proc. of the 20th Int'l Symp. on Computer Architecture*, 1993.

[14] S. Demetriades and S. Cho, "Barrierwatch: characterizing multithreaded workloads across and within program-defined epochs," in *Proc. of the 8th ACM Int'l Conf. on Computing Frontiers*, 2011.

[15] E. Ebrahimi *et al.*, "Parallel application memory scheduling," in *Proc. of the 44th Annual IEEE/ACM Int'l Symp. on Microarchitecture*, 2011.

[16] N. Eisley *et al.*, "In-network cache coherence," in *Proc. Int'l Symp. on Microarchitecture*, 2006.

[17] M. Ekman *et al.*, "TLB and snoop energy-reduction using virtual caches in low-power chip-multiprocessors," in *Proc. of the 2002 Int'l Symp. on Low power electronics and design*, 2002.

[18] N. D. Enright Jerger *et al.*, "Virtual tree coherence: Leveraging regions and in-network multicast trees for scalable cache coherence," in *Proc. Int'l Symp. on Microarchitecture*, 2008.

[19] H. Esmaeilzadeh *et al.*, "Dark silicon and the end of multicore scaling," in *Proc. of the 38th annual Int'l Symp. on Computer architecture*, 2011.

[20] H. Hossain *et al.*, "Improving support for locality and fine-grain sharing in chip multiprocessors," in *Proc. Int'l Conf. on Parallel Architectures and Compilation Techniques*, 2008.

[21] http://quid.hpl.hp.com:9081/cacti/, "CACTI 5.3."

[22] IEEE Computer Society, "IEEE standard for scalable coherent interface (SCI)." 1992.

[23] Intel Co., "MESIF protocol," uS Patent 6922756.

[24] Ioannou and et al, "Phase-based application-driven hierarchical power management on the single-chip cloud computer," in *Proc. of the Int'l Conf. on Parallel Architectures and Compilation Techniques*, 2011.

[25] N. D. E. Jerger *et al.*, "Circuit-switched coherence," in *IEEE 2nd Network on Chip Symp.*, 2008.

[26] J. A. Joao *et al.*, "Bottleneck identification and scheduling in multithreaded applications," in *Proc. of the Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, 2012.

[27] S. Kaxiras and C. Young, "Coherence communication prediction in shared-memory multiprocessors," in *Proc. Int'l Symp. on High-Performance Computer Architecture*, 2000.

[28] S. Kaxiras and J. Goodman, "Improving CC-NUMA performance using instruction-based prediction," in *Proc. Int'l Symp. on High-Performance Computer Architecture*, 1999.

[29] D. A. Koufaty *et al.*, "Data forwarding in scalable shared-memory multiprocessors," in *Proc. Int'l Conf. on Supercomputing*, 1995.

[30] A. Lai and B. Falsafi, "Memory sharing predictor: The key to a speculative coherent DSM," in *Proc. Int'l Symp. on Computer Architecture*, 1999.

[31] ——, "Selective, accurate, and timely self-invalidation using last-touch prediction," in *Proc. Int'l Symp. on Computer Architecture*, 2000.

[32] J. Laudon and D. Lenoski, "The SGI Origin: A CC-NUMA highly scalable server," in *Proc. Int'l Symp. on Computer Architecture*, 1997.

[33] A. R. Lebeck and D. A. Wood, "Dynamic self-invalidation: Reducing coherence overhead in shared-memory multiprocessors," in *Proc. Int'l Symp. on Computer Architecture*, 1995.

[34] S. Leventhal and M. Franklin, "Perceptron based consumer prediction in shared-memory multiprocessors," in *Int'l Conf. on Computer Design*, 2006.

[35] P. S. Magnusson *et al.*, "Simics: A full system simulation platform," *IEEE Computer*, 2002.

[36] M. M. K. Martin *et al.*, "Using destination-set prediction to improve the latency/bandwidth tradeoff in shared-memory multiprocessors," in *Proc. Int'l Symp. on Computer Architecture*, 2003.

[37] ——, "Token coherence. decoupling performance and correctness," in *Proc. of the 30th Annual Int'l Symp. on Computer Architecture*, 2003.

[38] A. Moshovos, "Regionscout: Exploiting coarse grain sharing in snoop-based coherence," in *Proc. Int'l Symp. on Computer Architecture*, 2005.

[39] S. Mukherjee and M. Hill, "Using prediction to accelerate coherence protocols," in *Proc. Int'l Symp. on Computer Architecture*, 1998.

[40] J. Nilsson *et al.*, "The coherence predictor cache: a resource-efficient and accurate coherence prediction infrastructure," in *Proc. of the Int'l Parallel and Distributed Processing Symp.*, 2003.

[41] A. Ros *et al.*, "A direct coherence protocol for many-core chip multiprocessors," *IEEE Trans. Parallel Distrib. Syst.*, 2010.

[42] S. Somogyi *et al.*, "Memory coherence activity prediction in commercial workloads," in *Workshop on Memory Performance Issues*, 2004.

[43] D. J. Sorin *et al.*, "Specifying and verifying a broadcast and a multicast snooping cache coherence protocol," *IEEE Transactons on Parallel and Distributed Systems*, 2002.

[44] P. Stenström *et al.*, "An adaptive cache coherence protocol optimized for migratory sharing," in *Proc. of the Int'l Symp. on Computer Architecture*, 1993.

[45] M. A. Suleman *et al.*, "Accelerating critical section execution with asymmetric multi-core architectures," in *Proc. of the Int'l Conf. on Architectural Support for Programming Languages and Op. Syst.*, 2009.

[46] Tilera Co. and http://www.tilera.com, "Tilera TILE64 processor."

[47] P. Trancoso and J. Torrellas, "The impact of speeding up critical sections with data prefetching and forwarding," in *Proc. Int'l Conf. on Parallel Processing*, 1996.

[48] S. C. Woo *et al.*, "The SPLASH-2 programs: characterization and methodological considerations," in *Proc. Int'l Symp. on Computer Architecture*, 1995.

Vulcan: Hardware Support for Detecting Sequential Consistency Violations Dynamically *

Abdullah Muzahid†, Shanxiang Qi, and Josep Torrellas
University of Illinois at Urbana-Champaign
http://iacoma.cs.uiuc.edu

Abstract

Past work has focused on detecting data races as proxies for Sequential Consistency (SC) violations. However, most data races do not violate SC. In addition, lock-free data structures and synchronization libraries sometimes explicitly employ data races but rely on SC semantics for correctness. Consequently, to uncover SC violations, we need to develop a more precise technique.

This paper presents *Vulcan*, the first hardware scheme to precisely detect SC violations at runtime, in programs running on a relaxed-consistency machine. The scheme leverages cache coherence protocol transactions to dynamically detect cycles in memory-access orders across threads. When one such cycle is about to occur, an exception is triggered. For the conditions considered in this paper and with enough hardware, Vulcan suffers neither false positives nor false negatives. In addition, Vulcan induces negligible execution overhead, requires no help from the software, and only takes as input the program executable. Experimental results show that Vulcan detects *three new SC violation bugs* in the Pthread and Crypt libraries, and in the fmm code from SPLASH-2. Moreover, Vulcan's negligible execution overhead makes it suitable for on-the-fly use.

1. Introduction

The model that programmers have in mind when they program and debug shared-memory threads is Sequential Consistency (SC). SC requires the memory operations of a program to appear to execute in some global sequence as if the threads where multiplexed on a uniprocessor [18]. In practice, however, current hardware overlaps, pipelines, and reorders the memory accesses of threads. As a result, a program's execution can be unintuitive.

As an example, consider Figure 1(a). Processor P_A allocates a variable and then sets a flag; later, P_B tests the flag and, if set, it uses the variable. While the particular interleaving in Figure 1(a) produces expected results, the interleaving in Figure 1(b) does not. In here, the hardware reorders the *completion* of the stores in the two statements in P_A. In this unlucky interleaving, P_B ends up using an unallocated variable. This order is an SC Violation (SCV).

From the hardware point of view, several conditions must be met for an SCV to occur. First, we need to have at least two data races — i.e., races on variables *buff* and *init* in the example. Secondly, these races must be of a very special type: they must be *overlapping* in time and *intertwined* in a manner that can form a cycle [30]. For two threads, it requires a pattern like that in Figure 2(a) where, if we follow program order, the two threads reference the same two

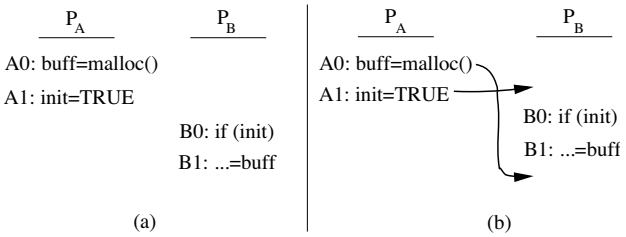


Figure 1. Example of an SC violation.

variables in opposite orders, and each variable is written at least once. Finally, the order of the references in these two racing pairs *has to form a cycle* at runtime. This is shown in Figure 2(b), where we have arbitrarily picked reads and writes: *A1* must occur before *B0* and *B1* must occur before *A0*. This is exactly what happened in Figure 1(b), where *y* was *init* and *x* was *buff*.

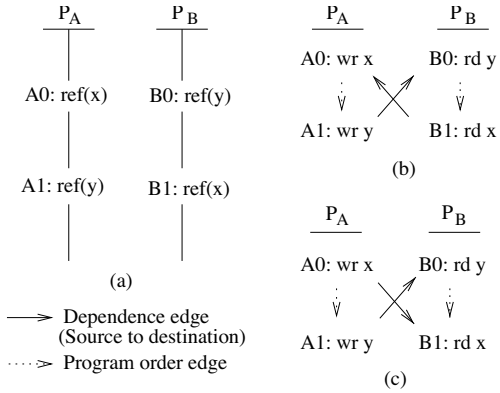


Figure 2. SC violations are possible.

Note, however, that if the timing at runtime is such that at least one of the two dependence arrows occurs in the opposite direction, there is no SCV. For example, Figure 2(c) shows the case when *A1* executed before *B0* but *A0* executed before *B1*. Since there is no cycle, SC is not violated. This case corresponds to the timing in Figure 1(a).

Data race patterns that cause SCVs are sometimes found in double-checked locking constructs [29], some synchronization libraries, and code for lock-free data structures.

Detecting SCVs is important because, in practically all cases, they are harmful, clear-cut bugs. The reason is that, as the example in Figure 1(b) shows, they require memory access orders that contradict a programmer's intuition. In addition, the programmer cannot reproduce them using a single-stepping debugger.

Past work has attempted to find SCVs by focusing on detecting data races (e.g., [4, 9, 15, 16, 23, 24, 32]). However, as we just saw, using data races as proxies for SCVs is very imprecise. The specific race pattern and interleaving required for an SCV is not necessarily

∗ This work was supported in part by the National Science Foundation under grants CNS 0720593, CCF 1012759 and CNS 1116237, and by Intel under the Illinois-Intel Parallelism Center (I2PC).
† Abdullah Muzahid is now with the University of Texas at San Antonio. His e-mail address is muzahid@cs.utsa.edu.

1072-4451/12 $26.00 © 2012 IEEE
DOI 10.1109/MICRO.2012.41

IEEE computer society

common. In large commercial codes, conventional race-detection tools typically flag many data races, often causing the programmer to spend time examining races that are much less likely to cause code malfunctioning than SCVs [13, 26].

A second reason for not using data races as proxies is that we may want to find SCVs in codes that have intentional data races, such as in lock-free data structures. We may want to debug the code for SCVs, while being less concerned about non-SC-violating races. Here, a race-detection tool would not be a good instrument to use. If we want to detect SCVs, we need to precisely zero-in on the types of data races and interleavings that cause them.

Given the importance of these bugs and the difficulty in isolating them, this paper contributes with *Vulcan*, the first hardware scheme to precisely detect SCVs at runtime, in programs running on a relaxed-consistency machine. Vulcan leverages cache coherence protocol transactions to dynamically detect cycles in memory access orders across threads. When a cycle is about to occur, an exception is triggered, providing information to debug the SCV.

The Vulcan design in this paper focuses on finding cycles of overlapping races between only two processors — since cycles involving three and more processors are much rarer. In addition, it does not consider speculative loads from mispredicted branch paths. Moreover, it is not concerned with SCVs due to compiler transformations — Vulcan only reports SCVs due to hardware-initiated reference reordering. Within these constraints, and with large-enough hardware structures, Vulcan suffers neither false positives nor false negatives.

Vulcan's approach has several advantages: it induces negligible execution overhead, requires no help from the software, and only takes as input the program executable. Experimental results show that Vulcan detects *three new bugs* in popular codes. Specifically, it finds SCVs in the Pthread and Crypt libraries, and in the fmm program from SPLASH-2. We have reported the bugs to the developers. In addition, Vulcan's negligible execution overhead makes it suitable for on-the-fly use.

We also contribute with a new taxonomy of data races.

This paper is organized as follows: Section 2 gives a background; Section 3 introduces a taxonomy of data races; Sections 4 and 5 present Vulcan; Section 6 outlines its limitations; Section 7 evaluates Vulcan; and Section 8 discusses related work.

2. Background

A Sequential Consistency Violation (SCV) occurs when the memory operations of a program have executed in an order that does not conform to any SC interleaving. It is virtually always a harmful bug, since it is the outcome of an unintuitive execution. Moreover, it is difficult to debug because single-stepping debuggers cannot reproduce it.

Shasha and Snir [30] show what causes an SCV: overlapping data races where the dependences end up ordered in a cycle. Recall that a data race occurs when two threads access the same memory location without an intervening synchronization and at least one is writing. Figure 2(a) showed the required program pattern for two threads (where each variable is written at least once) and Figure 2(b) showed the required order of the dependences at runtime (where we assigned reads and writes to the references arbitrarily).

If at least one of the dependences occurs in the opposite direction (e.g., Figure 2(c)), no SCV occurs. In addition, if the code of the two threads references the two variables in the same order (Figure 3(a)), no SCV is possible — no matter how the hardware reorders these references at runtime. For example, in Figure 3(b), no SCV can occur, no matter the direction of the inter-thread dependences.

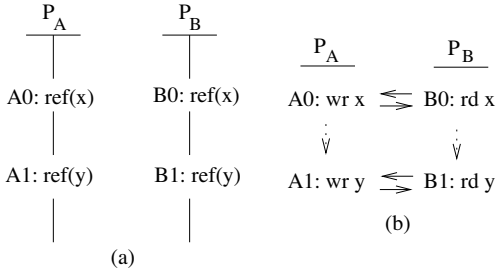


Figure 3. SC violations are not possible.

Given the pattern in Figure 2(a), Shasha and Snir [30] prevent the SCV by placing one fence between references *A0* and *A1*, and another between *B0* and *B1*. Their algorithm to find where to put the fences is called the Delay Set.

The commonly used Double-Checked Locking (DCL) [29] is a major source of SCVs. This is a programming technique to reduce the overhead of acquiring a lock by first testing the locking criterion without actually acquiring the lock. Only if the test indicates that locking is required does the actual locking logic proceed. The code takes a structure like in Figure 1(a). Because the code is typically involved, programmers often miss putting the two fences needed.

Data races and SCVs are very different, and programs have more data races than SCVs. However, past work has focused on detecting data races as proxies for SCVs. Specifically, one line of work detects incoming coherence messages on data that has local outstanding loads or stores. This work started with Gharachorloo and Gibbons [15] and now includes many aggressive speculative designs (e.g., [4, 9, 16, 32]). Another line of work detects a conflict between two concurrent synchronization-free regions. This includes DRFx [24] and Conflict Exceptions [23]. In general, all of these works look for a data race with two accesses that occur within a short time — but still, only a *single race*. Overall, while focusing on these races may be a good way to discard many irrelevant ones, it is still a very different problem than focusing on uncovering SCVs.

Other researchers have used the compiler to identify race pairs that could cause SCVs, typically using the Delay Set algorithm, and then insert fences to prevent cycles [12, 14, 17, 19, 31]. Since the compiler has limited information, these approaches tend to be very conservative and result in substantial slowdowns. Lin *et al.* [20] hide much of the resulting fence delay with architectural support.

Lin *et al.* [21] have recently proposed a design to support SC in a relaxed-consistency machine. While its goal is different than Vulcan's, it also involves the analysis of race cycles. We discuss it in Section 8. Finally, in the program testing and verification domains, there are proposals to detect SCVs by checking the semantic correctness of programs, or by collecting traces and then, off-line, applying reordering rules [6, 7, 8]. While such techniques are promising, they are typically limited to small-sized codes and are performed statically or as an off-line pass. Vulcan's goal is to detect SCVs in large codes on-the-fly and with negligible overhead. More details on related work are presented in Section 8.

3. A Taxonomy of Data Races

To assess the relationship between data races and SCVs, we develop a taxonomy of data races. We examined the bug databases of

popular programs such as Apache, MySQL, and Mozilla, and collected all the data-race bugs we could find. Since these are races reported by users, we know that they caused the program to malfunction. Table 1 lists the applications and the number of reported data races.

Application	# Reported Data Races	# Multi-Races	# SCV Races	# DCL SCVs
Apache	24	5	5	5
MySQL	13	1	1	1
Mozilla	11	2	1	1
Redhat (glibc)	2	2	2	1
Java SDK	2	1	1	1
PostgreSQL	1	0	0	0
Pbzip2	1 from [33]	0	0	0
Windows kernel	1 from [13]	0	0	0
Isolator bench.	1 from Isolator [27]	0	0	0
Total	56	11	10	9

Table 1. Reported data races that we studied.

Overall, we found 56 reported race-based bugs. For each of these bugs, if they contain more than one race, we call them *Multi-races*; otherwise, we call them *Single-races*. In addition, if a multi-race bug can create an SCV, we call it an *SCV Race*; otherwise it is a *Non-SCV Race*. Finally, SCV races are classified into those that are DCLs [29] and those that are not.

Table 1 shows the breakdown of the bugs per application. We see that, of the total 56 reported race bugs, 11 are multi-races (20%). Of these, 10 can cause SCVs (91%). The only one that, due to its reference pattern cannot ever create an SCV is in Mozilla [2]. Of the 10 SCV races, 9 are DCLs (90%).

It is well known from practical experience and from the literature [13, 26] that real programs contain many data races that users and developers do not consider important enough to report or to fix. Consequently, to put the previous numbers in context, we have to assume that there is a potentially sizable number of additional, unreported data races. Therefore, we can build the tree of Figure 4(a), which shows the frequency of each type of data race relative to its parent's. To visualize the frequency relative to all the race instances, Figure 4(b) shows a diagram where the area is proportional to the frequency of occurrence. Even if we do not know the actual number of unreported data races, the figure suggests that previous approaches that focus on data races as proxies for SCVs are insufficient.

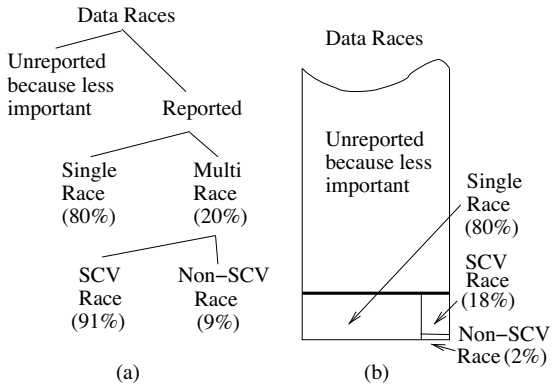


Figure 4. Relative frequency of data-race types.

The figure also shows why a special technique for SCV races is warranted: they comprise a substantial fraction of the reported data races, namely 18%. Importantly, they are very hard to debug, since current debuggers cannot reproduce them.

4. Vulcan: Detecting SC Violations

Our goal is to develop an approach to detect SCVs in relaxed-consistency machines that is highly precise. In addition, we want a solution that can deliver information to debug the SCV, uses no other input than the executable, and has a negligible execution overhead. Hence, we focus on a hardware-only solution to detect cycles of inter-thread data dependences at runtime.

The idea behind our approach, called *Vulcan*, is to rely on the cache coherence protocol to dynamically record the observed inter-thread data dependences, while checking whether they form cycles. These dependences are kept around only for as long as they can participate in a cycle, and are discarded soon after. Both the recording and the checking of these dependences is done in hardware to minimize execution overhead.

4.1. Basic Algorithm to Detect Cycles

Figure 5(a) repeats the pattern that can lead to an SCV with two threads. An SCV occurs when, due to the out-of-order execution of *ref(x)* and *ref(y)* in one thread or in both threads, *A1* executes before *B0*, and *B1* executes before *A0* — creating a dependence cycle.

To understand how Vulcan works, consider the dependence arrow of Figure 5(b), which represents that reference *A1* executed before reference *B0*. This arrow creates two regions, *R1* and *R2*, such that any future dependence whose source is in *R1* and destination is in *R2* will cause an SCV. Consequently, after Vulcan records *A1*→*B0*, it monitors that no new dependence is created from an access in P_B at or after *B0* to an access in P_A at or before *A1*. We put this requirement as the two restrictions of Figure 5(c):

• For any dependence whose source reference is in P_B at or after *B0*, the Allowed Destination (AD) in P_A is after *A1*.

• For any dependence whose destination reference is in P_A at or before *A1*, the Allowed Source (AS) in P_B is before *B0*.

If there are multiple dependences between two threads, then the *AD* of a dependence *from* a reference is the latest (i.e., maximum) of the contributing *AD*s, while the *AS* of a dependence *to* a reference is the earliest (i.e., minimum) of the contributing *AS*s. This is shown in Figure 5(d). In the figure, for each of the two dependences, we use the algorithm of Figure 5(c) to set the *AD*s of their R1 Regions and the *AS*s of their R2 regions. In the areas where the two R1 regions overlap (*B0* and later in P_B), Vulcan sets the *AD* to the maximum of the two values; in the areas where the two R2 regions overlap (*A1* and earlier in P_A), Vulcan sets the *AS* to the minimum of the two values.

Based on this discussion, Vulcan tags each monitored reference with three labels. They are shown in Figure 5(e). The first one is the Sequence Number (SN), which is the local dynamic reference count, assigned when the load or store enters the pipeline (e.g., at issue). The second one is the Allowed Destination (AD), which is the *SN* of the reference in the other processor after which the local reference can send data to. The last one is the Allowed Source (AS), which is the *SN* of the reference in the other processor before which the local reference can receive data from. Since a processor can have dependences with every other processor, *AD* and *AS* are arrays of *N*-1 entries, where *N* is the processor count. In each processor, *SN* starts up as 0 and increases monotonically. *AD* starts up as 0 and AS as ∞.

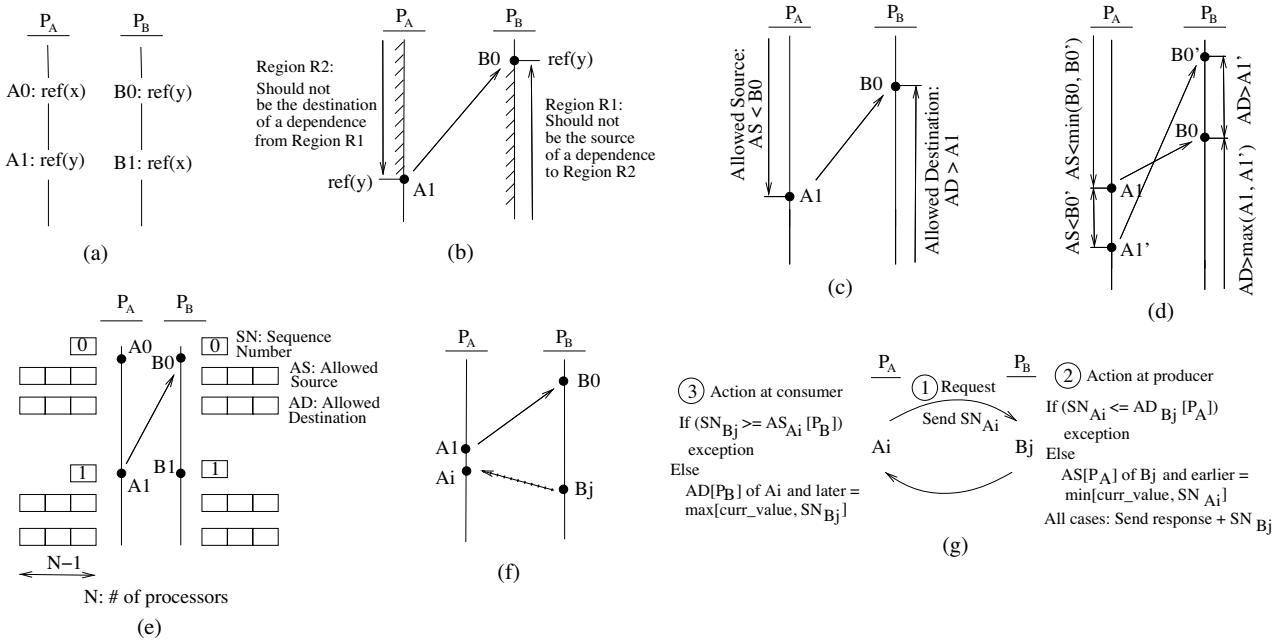


Figure 5. Basic algorithm to detect cycles.

These structures are updated in hardware when a new cross-processor dependence is created. The algorithm is shown in Figure 5(g), which refers to the example in Figure 5(f). Assume that we already have the solid arrow *A1→B0*; now P_A issues a request from reference *Ai* that prompts reference *Bj* in P_B to respond, creating the dotted arrow *Bj→Ai*. Figure 5(g) shows that there are three steps in the creation of the *Bj→Ai* arrow. Step 1 is the request from P_A, which carries the SN of the requesting access (SN_{Ai}). In Step 2, P_B operates on its Vulcan metadata, sends the response, and possibly raises an exception. Specifically, P_B checks that a cycle is not about to form by confirming that *Ai* is an allowed destination of *Bj*. If it is not ($SN_{Ai} \leq AD_{Bj}[P_A]$), a cycle is about to form and, hence an SCV is detected. In this case, P_B sends the response with the SN of the producer access (SN_{Bj}) and raises an exception. Otherwise, as in the example, the metadata is updated: the $AS[P_A]$ of *Bj* and earlier accesses in P_B are set to the minimum of their current values and SN_{Ai}. Also, P_B sends the response with SN_{Bj}.

Finally, in Step 3, when the data reaches P_A, P_A operates on its metadata and possibly raises an exception. Specifically, P_A checks that a cycle is not formed by confirming that *Bj* is an allowed source of *Ai*. If it is not ($SN_{Bj} \geq AS_{Ai}[P_B]$), a cycle is formed and an SCV has occurred. Consequently, an exception is raised. Otherwise, as in the example, the $AD[P_B]$ of *Ai* and later accesses in P_A are set to the maximum of their current values and SN_{Bj}.

With this algorithm, Vulcan raises exceptions immediately when a dependence closes a cycle and causes an SCV. This provides valuable information for debugging the SCV. The exception at the processor that receives the response always occurs. The exception at the producer processor may not occur since, at send time, there may not be enough dependences for a cycle yet. In Section 5.5, we consider all the information that is available to debug the SCV.

4.2. Safe Accesses

As a processor issues references, the Vulcan hardware monitors them. To understand for how long they need to be monitored, we define the concept of a *Safe* (and *Unsafe*) access:

• An access is *Safe* when no data dependence involving this access can cause an SCV any more. Otherwise, it is *Unsafe*. Vulcan can stop monitoring an access when it becomes Safe.

To find out when an access becomes Safe, let us define the *Performed Point* (PP) of a thread in an out-of-order processor. The PP is the latest memory access (in program order) such that it and all the accesses preceding it in the thread in program order have been performed. A load is performed when it has retired; a store is performed when it has retired and the cache has received the line and all the invalidation acknowledgments.

As a thread executes, its PP keeps advancing. When the PP reaches an access, it is clear that the access is completed. However, the access *may still* participate in an SCV and, therefore, be Unsafe. To see why, consider Figure 6(a). The creation of the *A1→B1* dependence makes the *B1* and subsequent accesses in P_B vulnerable. Indeed, even if they complete and P_B's PP goes past them, they can still participate in cycles. Specifically, if any access in P_A prior to *A1* requests data from them (or generally becomes dependent on them), a cycle is created. In precise terms: *B1* and subsequent accesses in P_B remain Unsafe for as long as P_A's PP has not reached the reference in their AD (*A1* in the example).

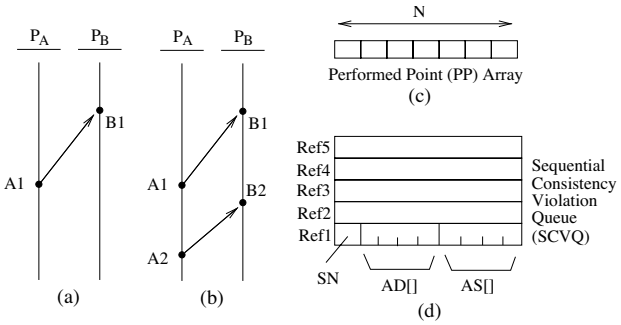


Figure 6. Understanding when an access is Safe.

The condition for an access *Ci* in processor P_C to be Safe is:

• Suppose that we have an array *PP[]* with the current value of the PPs for each processor (given as *SN* numbers). *Ci* is Safe when ($SN_{Ci} \leq PP[P_C]$) and ($AD_{Ci}[P_K] \leq PP[P_K]$), for all processors $K \neq C$. [Proof in *Theorem 1* of Appendix 1].

As an example, consider Figure 6(b). The accesses in P_A become Safe as soon as $PP[P_A]$ reaches them (since their AD has not been changed from 0). The accesses in P_B remain Unsafe even as $PP[P_B]$ reaches them. After that, as soon as *A1* becomes Safe, all the accesses in P_B up to (but not including) *B2* become Safe.

We also say that an access *Ci* in processor P_C is Safe *with respect to* another processor P_M:

• *Ci* is Safe with respect to P_M when ($SN_{Ci} \leq PP[P_C]$) and ($AD_{Ci}[P_M] \leq PP[P_M]$).

Vulcan uses these insights as follows. First, each processor has a PP[] array (Figure 6(c)). In this array, the entries corresponding to the other processors are kept largely up-to-date thanks to the fact that each processor includes its PP in every response message.

Second, a processor only keeps the *SN*, *AD*, and *AS* information for its references that are Unsafe. Such information is kept in a per-processor FIFO hardware queue associated with the cache controller called *SC Violation Queue* (SCVQ) (Figure 6(d)). When the processor issues a load or store, Vulcan allocates an SCVQ entry and sets its *SN* field. Later, as the access executes and coherence actions are received, the *AD* and *AS* fields are updated. Finally, when the access becomes Safe, Vulcan deallocates the entry.

An SCVQ entry does not contain the data loaded or stored. Moreover, the entry can remain allocated long after the access has completed — for as long as it remains Unsafe.

4.3. Detecting Dependences

When an SCV occurs, the following must be true:

• The two inter-processor dependence arrows that form the cycle must share a property: their source reference is Unsafe with respect to the destination processor. If one of the arrows fails this condition, there is no SCV. [Proof in *Theorem 2* of Appendix 1].

For example, in Figure 7(a), arrow *1* could participate in an SCV, while arrow *2* cannot. Consequently, we conclude:

• Vulcan only needs to watch for inter-processor data dependences where the source reference is Unsafe with respect to the destination processor. We call such dependences *Unsafe* dependences.

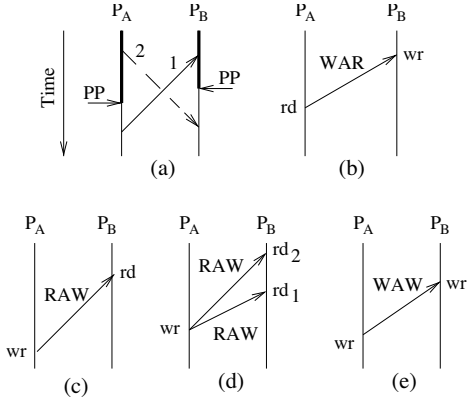


Figure 7. Inter-processor data dependences.

To find the Unsafe dependences, we will see that Vulcan uses the cache coherence protocol transactions (to a large extent). When one is found, the hardware performs the basic algorithm described in Section 4.1: the source and destination references exchange *SN*s, the source checks its *AD* and potentially updates its *AS* (and those of earlier accesses), and the destination checks its *AS* and potentially updates its *AD* (and those of later accesses).

Figures 7(b)-(e) show the three types of dependences possible: WAR, RAW, and WAW. Figure 7(b) shows a WAR. The write triggers Vulcan to search the other processors' SCVQs for accesses to the address. Multiple reader processors may be identified. Each reader processor has to take-in the write's *SN*, provide its read's *SN* and run the Vulcan algorithm; the writer has to take-in all the reads' *SN*s and run the Vulcan algorithm using the correct entries in its *AD* and *AS* arrays. In addition, since the write will be the source of *all* the future dependence(s) on this address, the write also triggers the removal (i.e., invalidation) of the SCVQ entries for this address in all the other processors.

Figure 7(c) shows a RAW. The read triggers Vulcan to search the other processors' SCVQs for a write to the address, ignoring SCVQ entries for reads. The usual algorithm is then run. Figure 7(d) shows a special case of a RAW, where the reader thread performs two reads to the same address *out of order*: first a later read (rd_1) and then a read that is earlier in program order (rd_2). In this case, both reads must communicate with the writer's SCVQ entry. In the process, rd_1 will first set the *AS* of the write (and of P_A's prior accesses) to rd_1's *SN*; later, rd_2 will set them to rd_2's *SN*, which is lower.

Figure 7(e) shows a WAW. As usual, the consumer write invalidates the SCVQ entry of the producer write. Note that other processors may have read the address in between the two writes. In this case, the consumer writer forms WAR dependences with the readers and a WAW dependence with the producer writer, and invalidates all the SCVQ entries for this address but its own.

We next show how we detect all the Unsafe dependences. The Appendix shows that:

• If Vulcan records all the Unsafe dependences, then it detects all the SCVs between processors. [Proof in *Theorem 3* of Appendix 1].

4.4. Leveraging the Coherence Protocol

To detect all the Unsafe dependences, Vulcan partially relies on piggybacking on the cache coherence protocol transactions. In this paper, we describe the operation assuming a snoopy-based MSI protocol; other protocols may require slightly different arrangements. Moreover, we assume a single-level private cache hierarchy per processor, where the SCVQ is associated with the cache controller, and multi-word cache lines. Without loss of generality, we describe our system using words (i.e., 32 bits) as the grain of processor accesses. We later consider finer-grained accesses such as bytes.

To understand how Vulcan uses the coherence protocol, this section starts by assuming single-word cache lines; Section 5 shows the final Vulcan design, which uses multi-word lines. With single-word lines, the destination access of the WAR, RAW, and WAW dependences of Figure 7 induces a coherence transaction in the network. Vulcan leverages such a transaction. The only exception is the second read (rd_2) in the RAW with out-of-order reads to the same address (Figure 7(d)). We describe this special case later.

As part of the coherence transaction, if the source reference is Unsafe (i.e., it is in an SCVQ), the Vulcan metadata is exchanged and operated upon. Specifically, on a processor read transaction in the network, the hardware searches the SCVQs that may contain

the referenced address (we will see how we know this). In a given SCVQ, it tries to find the latest write to the address in program order. From the above discussion, at most one SCVQ can have writes. If a write is found, we have detected a RAW. The Vulcan metadata is exchanged (as part of the transaction) and operated upon.

On a processor write transaction in the network, the hardware searches the SCVQs that may contain the referenced address. In each SCVQ, the search tries to find the latest access to the address in program order and, if that is a read, also any preceding write. Vulcan looks for the latest accesses because they form the most conservative dependences. If we find any, we have detected a WAR or a WAW. The metadata is exchanged and operated upon. As part of the transaction, all the entries for the address in all SCVQs (except in the requesting processor) are invalidated.

The second read (rd_2) in the RAW with out-of-order reads of Figure 7(d) presents a difficulty. On the one hand, the read hits in the cache and would not cause a coherence transaction. On the other hand, it needs to exchange *SN*s with the write and update the metadata (importantly, the *AS* of the write and prior accesses in P_A must become smaller). Vulcan solves the problem by forcing a *Metadata Network Access*, namely one exactly like a regular one (the SCVQs are searched and, if there is a hit, the Vulcan metadata is exchanged and operated on) except that no data is returned. Hence, when a load executes and finds that a later load to the same address has accessed the network, the hardware forces a metadata network access.

Vulcan's operation requires that, on a network transaction, the hardware looks-up the SCVQs that may have the referenced address. Vulcan cannot rely on the cache snoopers to flag which SCVQs may have the address — since the corresponding cache line may have been evicted from the cache. Consequently, Vulcan adds a per-processor bloom filter that encodes the addresses currently in the local SCVQ. If the address on the network hits in the filter, the SCVQ is searched. Section 5.3 presents a detailed design.

5. Vulcan Hardware Design

We present Vulcan's hardware structures: the coherence protocol and the SCVQs. We use a bus for the network. Section 7.2 summarizes the hardware needs for the configurations evaluated.

5.1. Supporting Multiple Words per Line

Detecting all the Unsafe dependences was easy with single-word cache lines because, conveniently, in all cross-thread dependences (Unsafe or otherwise, and except for RAWs with out-of-order read-read) the destination reference induces a coherence action in an MSI protocol (Figure 7). During the resulting bus access, if the dependence is Unsafe, processors exchange Vulcan metadata. Unfortunately, this is not the case with multi-word cache lines. As a processor misses on a word, other words are also brought into the cache. Consequently, some Unsafe dependences do not trigger coherence actions. Further, some coherence actions are caused by false sharing rather than by data dependences.

To solve this problem, Vulcan decouples, to some extent, the coherence actions from the Vulcan metadata operations. It ensures that every time that an Unsafe dependence occurs, either (1) the coherence protocol triggers a coherence action, or (2) Vulcan forces a Metadata bus access.

Let us use a plain line-based MSI coherence protocol using word accesses (for now). We assume that a bus transaction includes the address of the word accessed within the line. Vulcan adds two State bits per word in each line currently in the cache. These bits represent the word's *Vulcan-State* (or V-State). A word in the cache can be in one of three V-states: *CanWrite*, *CanRead*, and *NeedCheck*. Irrespective of the cache line state, a processor can write and read a *CanWrite* word in its cache without trying to exchange Vulcan metadata; it can only read a *CanRead* word without trying to exchange metadata; and it must try to exchange metadata at every access to a *NeedCheck* word. When needed, Vulcan metadata is piggybacked on the coherence bus transaction if the access induces one; otherwise, a Metadata bus accesses is initiated. These V-states are largely independent of the cache coherence state of the line. They follow rules when multiple caches have coherent copies of the word. Specifically, if one cache keeps the word in *CanWrite* state, then any other cache with the word must keep it in *NeedCheck* state. Also, if one cache keeps it in *CanRead* state, then any other cache can keep it in *CanRead* or *NeedCheck* state. Finally, the word may be in *NeedCheck* state in all of the cached copies.

Before describing how a word reaches each state, consider the (word) addresses of the accesses in an SCVQ. Typically, their corresponding line addresses are present in the local cache. However, there is one exception: when, after the access, the line was invalidated or displaced from the cache. In this case, the corresponding entries in the SCVQ have no V-state. In addition, when an invalidation is received, the SCVQ entry for the written word is cleared.

A word *w* in a line cached by a processor reaches the three V-states as follows:

• *CanWrite*: Either (i) the local processor was the last writer of *w* or, (ii) when the processor loaded *w* into its cache on a write miss to another word of the line, *w* was not in any other SCVQ (if the line was in another cache, it got invalidated). In addition, since any of these two events occurred, no other processor has (i) accessed *w*, or (ii) read-missed on another word in *w*'s line and loaded *w* as *CanRead*, or (iii) written *w*'s line. A *CanWrite* word may be in the local processor's SCVQ but not in other processors' SCVQs.

• *CanRead*: Either (i) the local processor has been involved in a dependence where the destination was a read of *w* (i.e., either the local processor wrote and then a remote one read, or a remote one wrote and then the local one read), or (ii) when the processor loaded *w* into its cache on a read miss to another word of the line, *w* was not in any other SCVQ. In addition, since any of these two events occurred, the local processor has not written *w* and no other processor has written to *w*'s line. A *CanRead* word may be in the SCVQs of the local and other processors.

• *NeedCheck*: When the local processor loaded *w* on a miss to another word of the line, *w* was in another processor's SCVQ. Since then, the local processor has not accessed *w* and no other processor has written to *w*'s line. A *NeedCheck* word may be in the SCVQs of the local and other processors.

We handle out-of-order read-read accesses to the same word like in Section 4.4: when a read executes and finds that a later read to the same address has already been sent to the bus, the hardware will eventually force a second bus access.

5.2. V-State Transitions for a Word

Figure 8 shows how the V-state of a word changes. For simplicity, we break the transitions into two figures. Figure 8(a) shows the transitions of the word as its line moves in and out of the cache, possibly due to accesses to other words in the same line; Figure 8(b) shows the transitions as the word is accessed inside the cache.

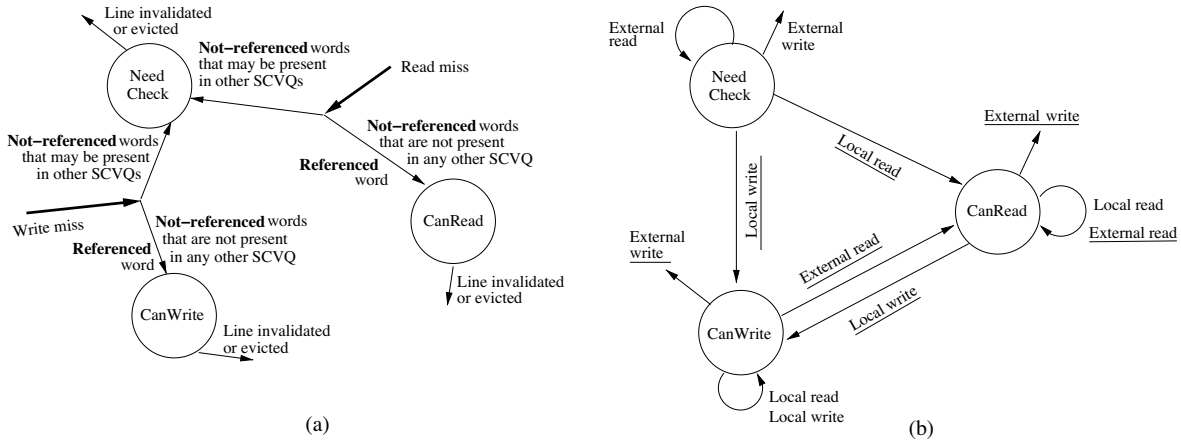


Figure 8. V-state transitions for a word. In (b), the underlined transitions may require metadata exchange (only needed if the source of the dependence is Unsafe) and, therefore, need a bus access. Such access can reuse a coherence bus transaction.

Starting with Figure 8(a), as a processor brings-in a line on a read miss, the hardware operates on the Vulcan metadata of the *referenced* word as indicated before, recording any Unsafe dependence. Hence, the word is loaded into the cache as *CanRead*. The other words in the line (i.e., *not-referenced* words) are loaded as either *CanRead* — if their address is not in any of the other processor's SCVQs — or as *NeedCheck* otherwise. This functionality is supported by adding one control line in the bus for each word in a line. During the bus transaction, all the other processors also check the addresses of the not-referenced words in the line against their bloom filter. If any processor finds a match for a given word, it sets the control line for that word. If the control line for a particular word is not set by the end of the bus transaction, it means that no processor has the word in its SCVQ, and the word is loaded as *CanRead*. As a word is loaded as *CanRead*, any other cached copies of the word that were *CanWrite* transition to *CanRead*.

Hardware-prefetched lines work seamlessly. We apply the algorithm for not-referenced words to all the words in the line.

If the line is brought-in on a write miss, the state becomes *CanWrite* for the referenced word. For the other words, the bloom filters are checked as above and the state is set as *CanWrite* if no SCVQ has the address or *NeedCheck* otherwise. Other cached copies of the line are invalidated.

When a line is evicted from the cache or invalidated by an external write to any of its words, the V-states of all its words are lost.

In Figure 8(b), the word is being accessed. The transitions correspond to accesses to the word. The transitions underlined may require metadata exchange (only needed if the source of the dependence is Unsafe) and, therefore, need a bus access — which can reuse a coherence transaction. Consider a *CanWrite* word. The local processor can read and write it silently. An external read requires a transition to *CanRead* and attempts metadata exchange. Consider a *CanRead* word. A local read is silent. A local write brings the local state to *CanWrite* and induces a bus access to try to exchange metadata. All other copies of the line are invalidated. An external read keeps the local state as *CanRead* and may involve metadata exchange. Finally, in an *NeedCheck* word, a local read and write bring the word to *CanRead* and *CanWrite*, respectively, and induce a bus access to try to exchange metadata. An external read keeps the word in *NeedCheck*. In all states, an external write invalidates the line (and the corresponding SCVQ entry). It may involve metadata exchange if the state was *CanRead* or *CanWrite*.

Figure 9 shows two examples of processors *P1* and *P2* accessing a line with words *A* and *B*. The figures show the transitions in V-states and line states. For each access (e.g., *rd A* by *P2*), the figure shows the resulting local V-state of each of the two words (*CW*, *CR*, and *NC* mean *CanWrite*, *CanRead*, and *NeedCheck*, respectively), the resulting local line state (*D* and *S* mean Dirty and Shared Clean, respectively), and the type of bus request (*CO* and *ME* mean coherence and metadata request, respectively). For example, the first read in Figure 9(a) brings the line to *P2* in state *S* with both words as *CanRead*. This is a coherence request without metadata exchange. As we go down the access stream, some accesses cause bus requests with only metadata exchange. We assume all SCVQ entries stay unless they are invalidated. Figure 9(b) shows an access stream with false sharing. All accesses cause coherence-only bus requests.

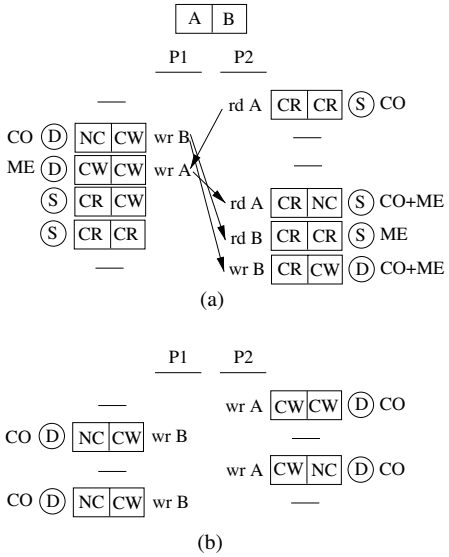


Figure 9. V-state transitions for two access streams.

Using V-state bits is an effective way to minimize metadata bus accesses when a processor references variables with temporal and spatial locality. Indeed, without V-state bits, all the words in the cache would effectively be in *NeedCheck* state, and every single access would require a metadata bus access. Unfortunately, V-state bits take space. Hence, a compromise that we employ is to keep

V-state bits *only* for lines that currently have at least one word in the local SCVQ. Since processor references have spatial and temporal locality, we are still likely to avoid many metadata bus accesses. When all the addresses of the words in the line leave the SCVQ (in addition to when the line is invalidated or evicted from the cache), the line's V-state information is discarded. A subsequent cache hit on the line initializes the V-state bits as *NeedCheck* for all the words (before the access). With this optimization, the V-state bits are stored in a hardware structure whose size is proportional to the maximum number of SCVQ entries rather than the number of lines in the L1 cache.

5.3. SCVQ Implementation

The SC Violation Queue (SCVQ) is a FIFO queue that contains the Vulcan metadata for Unsafe local loads and stores. Each entry contains the address loaded or stored, and the access' SN, AS, and AD. As a load or store enters the pipeline, an SCVQ entry is allocated, setting SN to the current value plus one, AS to ∞, and AD to the preceding access' AD. The AS and AD are updated later, when (1) the load or store executes, or (2) external accesses create dependences with the load or store, or other entries in the SCVQ.

Figure 10 shows the SCVQ. It stores the information in a FIFO circular queue. On a bus transaction, we need to look-up the SCVQ for an address match. Hence, we route the word addresses from the bus through a hash table and into the queue. With this design, it is easy to allocate and deallocate entries, and to find the entries that match bus transaction addresses. Finally, a write bus transaction that invalidates an SCVQ entry simply marks it as "empty".

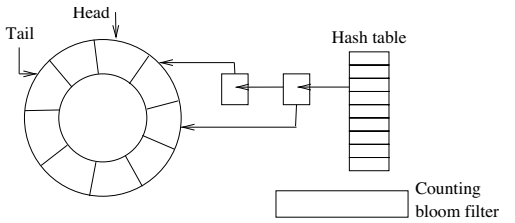


Figure 10. Implementation of the SC Violation Queue (SCVQ).

We want to minimize the number of useless SCVQ look-ups. However, we cannot rely on the cache snooper to filter them. This is because an SCVQ match may occur even if the corresponding line has been evicted from the cache. Hence, Vulcan uses a counting bloom filter [5] that hashes all the word addresses currently in the SCVQ. This structure uses counters to allow the removal of an individual hashed address. As entries are inserted and removed from the SCVQ, the addresses are added and removed from the filter. Bus transactions check the filter for a match before initiating a hash-table access. Any resulting false positives do not affect correctness; false negatives do not occur.

Inserting or removing addresses from the filter is not in a critical path. Insertion can occur any time from when the address of the reference is known until when the load or store completes and can be the source of an inter-thread dependence — in the meantime, the SCVQ entry is effectively not full. Removal can be done lazily, since at most it can induce false positive filter matches, which cause unnecessary SCVQ searches.

5.4. Granularity of V-State Bits

For most accurate SCV detection, the finest granularity of a program's accesses and the granularity of Vulcan's V-state bits have to be the same. Specifically, if a program loads or stores bytes, then Vulcan needs per-byte V-state bits — with per-word V-state bits, Vulcan may incur SCV false positives and false negatives.

To support byte accesses, Vulcan adds per-byte V-state bits to each line with at least one entry in the SCVQ. Individual entries in the SCVQ may refer to a byte or to a coarser access. Since the SCVQ bloom filter is looked-up by bus transactions accessing bytes or coarser data elements, Vulcan conservatively hashes word (rather than byte) addresses in the filter — at worst, it results in unnecessary SCVQ lookups. The transitions of Figure 8 operate on bytes or words depending on the granularity of the access. Specifically, on a byte access, when a line is brought into the cache (Figure 8(a)), the requested byte is searched in all of the SCVQs and loaded in the correct state; the other bytes of the line are loaded as *NeedCheck* or *CanRead/CanWrite* depending on whether their *word* address hits in the bloom filters. There are no additional filter lookups over a word transaction. Note that the design is such that, if the program only has word accesses, the per-byte V-state bits create no additional traffic over having only per-word V-state bits.

5.5. Information Available to Debug an SCV

Consider the cycle shown in Figure 2(b). There are two possible cases for when the SCV is detected. The first case is when one dependence arrow (e.g., *A1*→*B0*) is fully recorded when the source of the second dependence (*B1*) sends the response; the second case is when it is not, because both responders (*A1* and *B1*) respond concurrently. In the first case, the SCV is detected at both the source (*B1*) and destination (*A0*) of the second dependence; in the second case, it is detected at the destinations of the two dependences (*A0* and *B0*). In either case, when each processor detects the SCV, it raises an exception.

The information that is available to the debugger in the interrupted processor at the destination of the dependence is the address being accessed, the instruction's PC and, depending on the protocol implementation, the ID of the sender processor. If the destination reference is a read, the exception gets the precise processor state; if it is write, it is not generally possible to get the precise state at the reference — the reason is that the write is in the write buffer and later operations may have already retired and completed. The information available to the debugger in the interrupted processor at the source of the dependence is the address accessed and the ID of the requesting processor. The instruction's PC is unavailable — unless we augment the SCVQ with PCs. The exception in the source processor is not precise because newer instructions may have finished. Finally, the debugger can also inspect the Vulcan metadata of all the Unsafe requests in the two processors, to provide more information.

6. Limitations of the Current Vulcan Design

The current Vulcan design has some limitations. The first one is that it focuses on cycles involving only two processors. In practice, this is not a major limitation because cycles involving more processors are much rarer — they need the overlapping of three or more data races. Much of the related work also focuses on two-processor interactions only (e.g., [6, 8, 10]). We could extend Vulcan to handle several-processor cycles by propagating the *AS/AD* information along the dependence arrows, instead of just sending *SN*.

A second limitation is that the current design does not consider speculative loads from mispredicted branch paths. In a real system, these loads cannot generate SCVs. However, to be able to take them

into account, we would need to change Vulcan. For example, the hardware may have to delay performing metadata updates until the load becomes non-speculative. However, Vulcan supports hardware prefetches and within-processor load forwarding.

Vulcan is not concerned with the impact of compiler transformations on SCVs. It simply takes the executable that the compiler provides to the hardware and reports SCVs due to hardware-initiated reference reordering. Similarly, since Vulcan is a dynamic scheme, it only provides information for the actual performed runs.

We discussed in Section 5.4 that the finest granularity of program accesses and of Vulcan's V-state bits have to be the same — otherwise, both SCV false positives and false negatives may occur. Finally, the SCVQs need to be large enough to hold all the Unsafe accesses. If they are not and they have to drop some of these accesses, then SCV false negatives may occur.

Overall, within these constraints (two-processor cycles only, no misspeculated loads, and no compiler effects) and with appropriate hardware structures (correct grain of V-state bits and large-enough SCVQs), Vulcan has neither false positives nor false negatives.

Finally, our Vulcan design in a snoopy protocol with all-to-all hardware structures may not scale well to large numbers of processors. However, this is not a major limitation. First, our evaluation shows that Vulcan scales well until at least 8 processors (and we did not explore beyond). Also, it is well known that runs with few processors are typically enough to find concurrency bugs [22].

7. Evaluation

Our goal is to (1) validate Vulcan's effectiveness in detecting SCVs, (2) determine the size of its hardware structures, and (3) assess its overhead in terms of network traffic and execution time.

7.1. Experimental Setup

We model Vulcan's architecture using cycle-level execution-driven simulations. We use the SESC simulator [28] to model a multicore with a variety of configurations: four or eight out-of-order cores with 2- or 4-issue wide pipelines and supporting the Release Consistency (RC) or Processor Consistency (PC) memory models. They have a simple cache hierarchy composed of a private L1 cache and a shared L2 cache. Table 2 shows the architecture parameters. When there is a choice, the values in bold are the default ones. In most of the evaluation, we use per-word V-state bits; in the last part, we use per-byte V-state bits.

Architecture	Chip multiprocessor with **4** or 8 cores.
Core pipeline	Out-of-order; 2.0GHz; **2-issue** or 4-issue.
ROB size	**32**, 64, 128, or 256 entries.
Consistency	**Release (RC)** or Processor (PC) consistency.
Private L1 cache	32KB WB, 4-way asso., 2-cycle round trip.
Shared L2 cache	1MB WB, 8-way asso., 20-cycle round trip.
Cache line size	**32B** or 4B.
Coherence	Snoopy MSI protocol; 1.0GHz 16B-wide bus.
Round-trip lat.	L1-L1: 38 cyc; processor-memory: 500 cyc.
Vulcan parameters	SCVQ: 256 entries; SN, AD[i], AS[i]: 4B each. Bloom filter: 128B with 2-bit counts, H3 hash. **Word** or byte V-state bits for lines in SCVQ.

Table 2. Multicore architectures evaluated.

We use three sets of applications for the evaluation (Table 3). The first set has implementations of concurrent data structures and mutual exclusion algorithms that have SCVs. They are taken from [6, 8]. The second set has some reported SCV bugs from open source libraries. The last set has 8 codes from SPLASH-2. The first two sets have known SCVs and are used to evaluate Vulcan's effectiveness. The last set has lengthy applications, supposedly free of SCVs, and is used to estimate Vulcan's overheads.

Set	Program	Description
Conc. Algo.	Dekker	Algorithm for mutual exclusion.
	Lazylist	List-based concurrent set.
	Snark	Nonblocking double-ended queue.
	Harris	Nonblocking set.
Bug Kernels	Pthread_cancel from glibc	Unwind code after canceling thread needs a fence [3].
	Crypt_util from glibc	Small table initialization code needs a fence [1].
	DCL bug	Kernel using double checked locking without fences.
Full Apps	SPLASH-2	8 programs form SPLASH-2.

Table 3. Applications analyzed.

7.2. Hardware Requirements

Vulcan adds to each core the following hardware: (1) SCVQ circular queue with its hash table, (2) SCVQ bloom filter, (3) V-state bits in the lines with at least one entry in the SCVQ, and (4) Performed Point array. For the default parameters in Table 2, in a 4-core chip, the storage requirements are about 8448B, 128B, 512B, and 16B, respectively, or a total of 9KB per core. For a machine with *N* cores, the overhead per core can be shown to be (2052*N + 896) bytes. This means that, in an 8-core chip, the overhead is 17KB per core.

If we want to support byte-level accesses, we need per-byte V-state bits for each line with at least one entry in the SCVQ. Moreover, each SCVQ entry needs a 2-bit longer address and 2 bits to denote whether the reference was to a byte, half-word, or word. The SCVQ bloom filter still conservatively hashes word addresses. All this support only adds 1.7KB more Vulcan storage overhead per core, irrespective of the number of cores in the machine.

7.3. SC Violation Detection Ability

To test Vulcan's SCV detection ability, we run each application multiple times — 100 times for the concurrent algorithms and bug kernels, and 5 times for the SPLASH-2 codes. In each run, we generate different interleavings by forcing the processors to miss some random number of fetch cycles. For each application, we report, over all the runs, the number of *unique* and *total* SCVs observed. This information is shown in Table 4, for cache lines of 4 and 32 bytes, and for RC and PC memory models. For SPLASH-2, the table only shows fmm because Vulcan finds no SCV in the other SPLASH-2 codes.

Under RC (Columns 4 and 5), Vulcan detects SCVs in all of these codes (except in two codes with 4B lines). Under PC (Columns 6 and 7), Vulcan finds slightly fewer unique SCVs, and none in Lazylist or Snark. This is because PC is stricter than RC, and some SCVs may be impossible or less likely to occur. Also, the number of SCVs found changes with the line size. This shows that this bug is highly dependent on the timing of events.

Overall, we find that Vulcan is very effective at finding SCVs in these two different memory models. With more runs, new interleavings may occur and Vulcan may find more SCVs.

Finally and most importantly, Vulcan *finds three new, previously unreported SC violation bugs* in the codes in bold in Table 4: one in Pthread_cancel, one in Crypt_util, and one in fmm (which appears as three unique SCVs). We discuss them next.

Appl.	Line Size (B)	# of Runs	# of SC Violations Found			
			Under RC		Under PC	
			Uniq.	Total	Uniq.	Total
Dekker	4	100	1	1982	1	1784
	32	100	1	224	1	518
Lazylist	4	100	0	0	0	0
	32	100	1	150	0	0
Snark	4	100	1	745	0	0
	32	100	1	1467	0	0
Harris	4	100	0	0	1	2
	32	100	1	18	1	2
Pthread _cancel	4	100	2	298	1	104
	32	100	2	142	1	400
Crypt _util	4	100	2	564	1	228
	32	100	2	130	1	800
DCL	4	100	2	648	1	600
	32	100	1	2	1	491
fmm	4	5	1	2	3	14
	32	5	3	18	0	0

Table 4. SC violations found in various applications. Vulcan found three new SC violations in the codes in bold.

7.4. Three New SC Violation Bugs Found

• New SC Violation in the Pthread Library

One of the SCVs in the *Pthread_cancel* kernel of Table 4 is Bug ID 2644 in the Redhat bug database [3], which has been fixed by the developers. After running Vulcan, we found a *new SC violation even in the bug fix*. We reported the new bug and its fix to the developers, who have recently implemented the fix.

Figure 11 shows the bug in the original bug fix. Figure 11(a) shows the *pthread_cancel_init* and *_Unwind_Resume* functions, together with the fence (write barrier) that the developers inserted in an attempt to fix the bug. Assume that thread T1 is in *pthread_cancel_init*, and about to initialize function pointers *libgcc_s_resume* (in A0) and *libgcc_s_gtecfa* (in A1). Before it does so, thread T2 is in *_Unwind_Resume* and calls *pthread_cancel_init*. There, it finds *libgcc_s_gtecfa* already non-null (in B0), returns from *pthread_cancel_init* and uses *libgcc_s_resume* (in B1). However, due to an SCV, *libgcc_s_resume* is still uninitialized and the program crashes.

The references involved and the fence are shown in Figure 11(b). This code is the same as Figure 1(a) except for the fence. Unfortunately, the fence only prevents the A0-A1 reorder. In an RC (or PowerPC) memory model, B0 and B1 can *effectively* get reordered as in Figure 11(c), causing a cycle. Specifically, the condition in B0 is predicted true by the branch predictor (although it is currently false) and B1 is executed before A0. After A0 and A1 execute, the B0 branch resolves, confirming that B1 is in the correct path. However, B1 used the old value and the code crashes. To fix this, we also add a fence between B0 and B1.

• New SC Violation in the Crypt Library

A similar situation occurs for *Crypt_util*. One of its SCVs in Table 4 is Bug ID 11449 in the database [1], which had also been incorrectly fixed by the developers. After running Vulcan, we found a new SCV in the bug fix. We reported the new bug and its fix to the developers. They declined to fix it because the bug also only happens in memory models more relaxed than Intel's x86 and the cryptography library is used little.

Figure 12(a) shows the buggy code of function *__init_des_r*, which uses DCL to initialize shared tables, and the fence that the developers added to fix the bug. Assume that thread T1 enters the

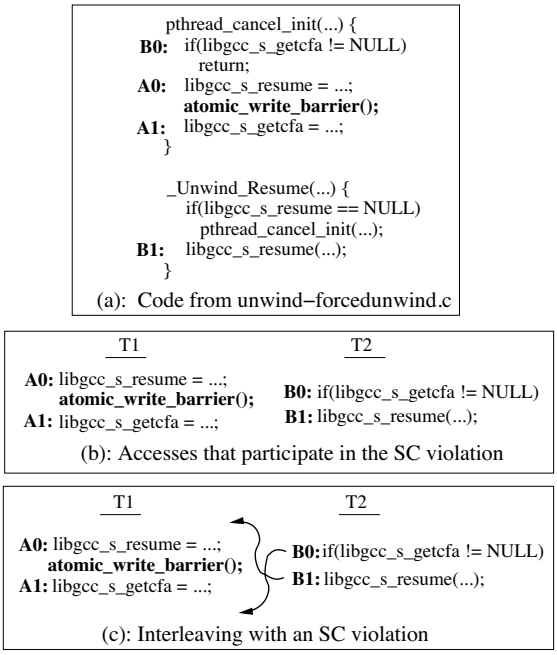


Figure 11. New SC violation found in the glibc pthread library.

function, grabs the lock and is about to initialize table *eperm32tab* (in A0) and then set *small_tables- _initialized* (in A1). Thread T2 enters the function, finds *small_tables_initialized* set (in B0) and uses *eperm32tab* (in B1). Unfortunately, *eperm32tab* is still uninitialized due to the SCV.

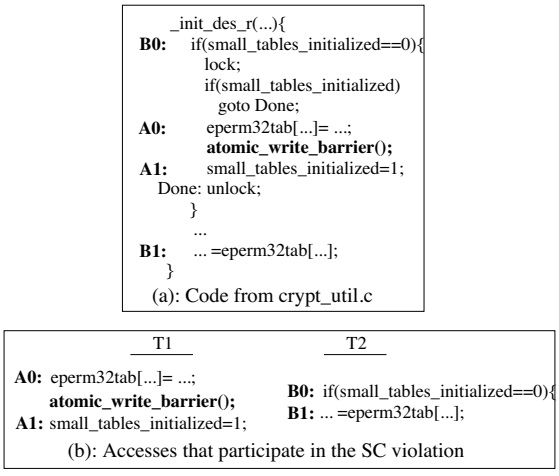


Figure 12. New SC violation found in the glibc crypt library.

The references involved and the fence are shown in Figure 12(b). The code is similar to the one in Figure 11(b). We need another fence between B0 and B1.

• New SC Violation in fmm from SPLASH-2

Vulcan finds three new SCVs in fmm, caused by a single flag dependence racing against three pairs of references. The code for one of the racing pairs is shown in Figure 13. Inside the *SetColleagues* function, a thread (T2) sets structure *colleagues* (in A0) and then flag *construct_synch* (in A1); another one spins on the flag (in B0) and then uses the structure (in B1). This is the pattern of Figure 1, and an SCV occurs. In the fmm code, the flag was de-

clared as volatile. However, in C, while volatile prevents compiler optimizations, it does not prevent reordering by the hardware.

```
        T1                                   T2
   SetColleagues(...) {                 SetColleagues(...) {
B0: while(b->construct_synch==0);    A0: b->colleagues[...] =...;
B1: ... = parent_b->colleagues[...]; A1: child_b->construct_synch=1;
   }                                    }
                Code from construct_grid.c
```

Figure 13. New SC violation found in fmm from SPLASH-2.

This SCV affects the precision of the program's output because thread T1 uses "old data". However, since fmm is an N-body problem, the output might still be acceptable. Still, this is a serious bug because the programmer can hardly reason about the bug's impact on the code. This bug can be fixed by either placing a fence between the two references in each thread, or by using a synchronization instruction to access the flag.

7.5. SCVQ Size and Sensitivity

To size the SCVQ, we need to know the number of Unsafe accesses that individual processors maintain. Consequently, we count the average and maximum number of Unsafe accesses per processor over time. We use only SPLASH-2 applications because the others are too small to provide useful information. For our measurements, we take a sample every time a memory operation is issued. We additionally count the average and maximum number of pending accesses. These are loads and stores that have been issued but not yet completed, and are a strict subset of Unsafe accesses — an access remains Unsafe at least while pending and often beyond that. Figure 14 shows the results for each application.

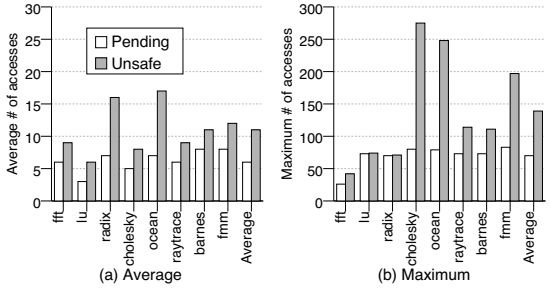


Figure 14. Number of pending and Unsafe accesses.

The average number of Unsafe accesses ranges from 6 to 17 (Figure 14(a)). This is a small number, and is about double of the average number of pending accesses. However, accesses are typically bursty and the maximum number of Unsafe accesses is higher. Across applications, it ranges from 40 to 270 (Figure 14(b)). If we average out all the codes, the number is about 140, which is also about double of the maximum number of pending accesses.

Overall, to be conservative, we size the SCVQ with 256 entries. Most of the time, only about 10 or so entries are in use. In one application, namely cholesky, there are 170 times in the execution of the 147-million memory-access program when we need more than 256 entries. Hence, we have rerun the program with a 270-entry SCVQ, which is large enough, and found no SCVs either.

We now measure how the number of Unsafe and pending accesses changes with the ROB size and processor issue width. This is shown in Figure 15, which plots the average across all SPLASH-2 codes. For each ROB size and issue width, we show the average and maximum number of pending and Unsafe accesses. The number on top of the maximum Unsafe bars is the number of SCVQ overflows, as a percentage of total instructions. We see that, for our default issue width (Figure 15(a)), changes in the ROB size have negligible impact. For 4-issue cores (Figure 15(b)), if the ROB reaches 128 entries or more, the SCVQ starts to overflow.

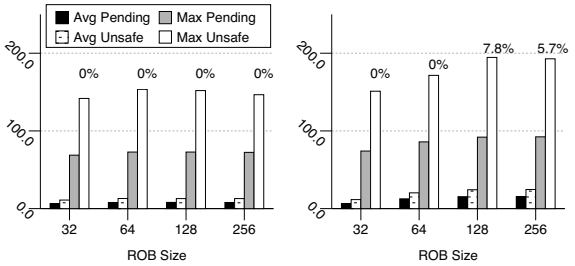


Figure 15. Pending and Unsafe accesses for different ROB sizes and issue widths.

7.6. Network Traffic & Execution Overhead

Vulcan's execution overhead comes from the additional bus traffic that it induces. This traffic has two sources: (i) the information that Vulcan piggybacks on some of the ordinary coherence transactions on the bus and (ii) the Metadata bus accesses that it induces (Section 4.4). In both cases, Vulcan sends a Sequence Number in the request (4 bytes), and both a Sequence Number and a Performed Point in the response (8 bytes).

To see the magnitude of this traffic, Figure 16 breaks down the total bytes of traffic in the bus for each application. We run the experiments for both 4-core and 8-core systems. The categories are: traffic in a Vulcan-free execution (*No Vulcan*), traffic piggybacked by Vulcan on the normal coherence (*Piggybacked*), and traffic in Metadata bus accesses (*Extra*). We see that Vulcan's effect is modest: on average for 4 cores, *Piggybacked* accounts for 9% of the traffic and *Extra* for 12%. For 8 cores, the result is similar.

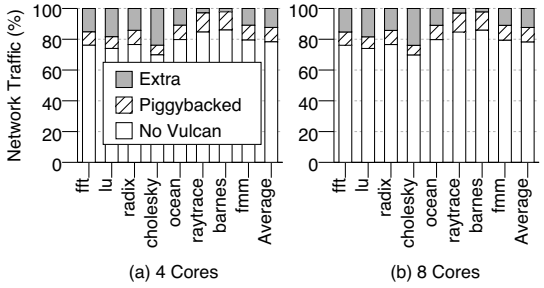


Figure 16. Breakdown of total bus traffic in bytes.

Given the bus parameters of Table 2, we assume that the additional bytes piggybacked by Vulcan on a coherence transaction do not increase the bus occupancy cycles of the transaction. However, for Metadata bus accesses, we assume bus occupancies of 2 bus cycles for request and 2 for reply. The contention induced by these accesses causes Vulcan's execution overhead.

Tables 5 and 6 show Vulcan's execution overhead for 4 and 8 core systems, respectively. Each table shows the execution overhead with word and byte granularity for V-state bits. For each core count and V-state granularity, the tables show the number of bus

accesses, the fraction of those that are Metadata bus accesses, and the increase in the program's execution time due to Vulcan.

Appl.	Word Granularity			Byte Granularity		
	Tot. Bus Acc. (Mil.)	Meta. Bus Acc. (%)	Exec. Time Over. (%)	Tot. Bus Acc. (Mil.)	Meta. Bus Acc. (%)	Exec. Time Over. (%)
fft	0.4	32.4	4.9	0.4	32.6	4.9
lu	1.2	34.4	3.8	1.2	34.4	3.8
radix	2.0	32.5	0.7	2.0	32.6	0.7
chole.	34.4	38.9	8.1	34.4	38.9	8.1
ocean	21.5	26.7	5.5	21.5	26.7	5.5
raytr.	3.1	8.8	4.2	3.6	19.8	6.7
barnes	30.7	6.9	2.7	30.9	6.9	2.7
fmm	19.2	25.8	2.6	19.2	25.8	2.6
Avg.	14.1	25.8	4.1	14.2	27.2	4.4

Table 5. Vulcan's execution overhead for 4 cores.

Appl.	Word Granularity			Byte Granularity		
	Tot. Bus Acc. (Mil.)	Meta. Bus Acc. (%)	Exec. Time Over. (%)	Tot. Bus Acc. (Mil.)	Meta. Bus Acc. (%)	Exec. Time Over. (%)
fft	0.4	31.8	9.5	0.4	31.9	9.5
lu	1.2	35.3	3.6	1.2	35.3	3.6
radix	2.1	33.0	1.4	2.1	33.0	1.4
chole.	35.6	38.4	9.0	35.7	38.5	9.0
ocean	21.6	27.7	12.3	21.7	27.6	12.3
raytr.	3.6	9.6	4.4	4.0	19.0	6.9
barnes	35.0	6.5	2.8	35.1	6.3	2.7
fmm	19.4	26.0	2.8	19.4	26.0	2.8
Avg.	14.9	26.0	5.7	14.9	27.2	6.0

Table 6. Vulcan's execution overhead for 8 cores.

The tables show that Metadata bus accesses account for an average of 26-27% of the bus accesses, and that such fraction does not change much with the core count. Importantly, Vulcan's execution time overhead is small. On average for word granularity, it is 4.1% for 4-core systems and 5.7% for 8-core systems.

When Vulcan supports V-state byte granularity, the overhead increases in the applications with a non-negligible fraction of byte accesses. For the applications considered, only Raytrace is in this class. As a result, in Raytrace, the number of Metadata bus accesses increases and the execution time overhead increases a modest 2.5 percentage points, as we go from word to byte granularity for both processor counts. For the other codes, since they reference mostly words rather than bytes, Vulcan's execution behaves as if it had word- rather than byte-granularity V-state bits. On average for all the applications, the execution overhead with byte-granularity V-state bits is 4.4% for 4-core systems and 6.0% for 8-core systems.

Overall, we conclude that Vulcan's execution overhead is small enough to allow on-the-fly use — both with word- and byte-granularity V-state bits. In addition, the overhead scales nicely from 4- to 8-core systems.

8. Other Related Work

There is related work in architecture, compilation, testing, and hardware verification. In architecture, the most related work is Conflict Ordering (CO) by Lin *et al.* [21], which is a technique to support SC in a relaxed-consistency machine. CO is also based on identifying Shasha's delay sets [30] in hardware. At a high level, CO and Vulcan differ in that their goals are different: Vulcan focuses on identifying SCVs, while CO focuses on supporting SC. However, Vulcan could be extended to support SC when an upcoming SCV is suspected, and CO could stop execution when an SCV is possible. Hence, at a deeper level, CO and Vulcan are similar in that they both attempt to identify race cycles.

CO's key contribution is to use information about pending accesses in the directory module to avert cycles. Unfortunately, CO requires introducing stalls in processor requests. Specifically, there are two stall types: write- and read-induced. Write-induced stalls occur when the write *W* that is about to retire misses in the cache. At that point, the next read or write cannot retire until *W* goes to the directory, leaves its address there, and brings back the list of pending writes (write-list). This stall cannot be eliminated with exclusive prefetching. Read-induced stalls occur when a speculative read *R* misses in the cache. When *R* reaches the ROB head, *R* has to perform a directory access again, to obtain the write-list. Only when the write-list returns can *R* retire and allow subsequent reads and writes to retire. Again, this cannot be fixed by prefetching.

CO also differs from Vulcan in that, to detect cycles, it compares line addresses rather than word or byte addresses. This causes false positives. Luckily, false positives simply cause stalls — although this approach would not work to debug SCVs like Vulcan. If, instead, CO compared word addresses, then a processor accessing multiple words of the same line in sequence would have to make multiple directory accesses to deposit the addresses of all the words.

There are compiler techniques to identify race cycles and put fences (e.g., [14, 17, 19, 31]). They are conservative because they only use static information, and typically cause large slowdowns. Lin *et al.* [20] can hide some of the resulting fence delay with architectural support. Duan *et al.* [12] use a race detector to construct a graph of races dynamically. Then, off-line, they traverse the graph to find potential SCVs. Vulcan differs in that: (1) it is an on-the-fly scheme, while Duan's SCV detection is off-line; (2) it needs no software support; and (3) it has no false positives, while Duan's scheme may point to SCVs that never occur.

The software testing community has proposed static and off-line techniques to check for SCVs (e.g., [6, 7, 8]). While promising, these techniques are not designed for on-the-fly SCV detection in large codes with negligible overhead. The hardware verification community has designed techniques to verify if a memory system hardware is correctly implemented (e.g., [10, 11, 25]). While related, these works have a different goal: we focus on debugging software as it runs on a relaxed-consistent machine; they focus on verifying that the hardware correctly implements a memory model.

9. Conclusion

This paper proposed Vulcan, the first hardware scheme to precisely detect SCVs at runtime, in programs running on a relaxed-consistency machine. Vulcan uses cache coherence protocol transactions to dynamically detect cycles in memory access orders across threads. When a cycle is about to occur, an exception is triggered, providing information to debug the SCV. For the conditions considered in this paper and with enough hardware, Vulcan suffers neither false positives nor false negatives. It induces negligible execution overhead, requires no software help, and only takes as input the program executable. Our results showed that Vulcan detected *three new SCV bugs* in popular codes: Pthread and Crypt libraries, and fmm from SPLASH-2. Vulcan's negligible execution overhead makes it suitable for on-the-fly use.

References

[1] Sources bugzilla. Bug 11449. http://sources.redhat.com/bugzilla/show_bug.cgi?id=11449.
[2] Sources bugzilla. Bug 133773. https://bugzilla.mozilla.org/show_bug.cgi?id=133773.
[3] Sources bugzilla. Bug 2644. http://sources.redhat.com/bugzilla/show_bug.cgi?id=2644.
[4] C. Blundell, M. M. Martin, and T. F. Wenisch. InvisiFence: Performance-transparent memory ordering in conventional multiprocessors. In *ISCA*, June 2009.
[5] F. Bonomi et al. An improved construction for counting Bloom filters. In *Ann. Euro. Symp. on Algo.*, September 2006.
[6] S. Burckhardt, R. Alur, and M. M. K. Martin. CheckFence: Checking consistency of concurrent data types on relaxed memory models. In *PLDI*, June 2007.
[7] S. Burckhardt and M. Musuvathi. Effective program verification for relaxed memory models. In *CAV*, July 2008.
[8] J. Burnim, K. Sen, and C. Stergiou. Sound and complete monitoring of sequential consistency for relaxed memory models. In *Tools and Algo. for the Const. and Ana. of Sys.*, July 2011.
[9] L. Ceze, J. Tuck, P. Montesinos, and J. Torrellas. BulkSC: Bulk enforcement of sequential consistency. In *ISCA*, June 2007.
[10] K. Chen, S. Malik, and P. Patra. Runtime validation of memory ordering using constraint graph checking. In *HPCA*, February 2008.
[11] A. Deorio et al. DACOTA: Post-silicon validation of the memory subsystem in multi-core designs. In *HPCA*, February 2009.
[12] Y. Duan, X. Feng, L. Wang, C. Zhang, and P.-C. Yew. Detecting and eliminating potential violations of sequential consistency for concurrent C/C++ programs. In *CGO*, March 2009.
[13] J. Erickson, M. Musuvathi, S. Burckhardt, and K. Olynyk. Effective data-race detection for the kernel. In *OSDI*, February 2010.
[14] X. Fang, J. Lee, and S. P. Midkiff. Automatic fence insertion for shared memory multiprocessing. In *ICS*, June 2003.
[15] K. Gharachorloo and P. B. Gibbons. Detecting violations of sequential consistency. In *SPAA*, July 1991.
[16] C. Gniady, B. Falsafi, and T. N. Vijaykumar. Is SC + ILP = RC? In *ISCA*, May 1999.
[17] A. Krishnamurthy and K. Yelick. Analyses and optimizations for shared address space programs. *Jour. Paral. Dist. Comp.*, Nov 1996.
[18] L. Lamport. How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs. *IEEE Trans. Comp.*, July 1979.
[19] J. Lee and D. A. Padua. Hiding relaxed memory consistency with a compiler. *IEEE Trans. Comp.*, August 2001.
[20] C. Lin, V. Nagarajan, and R. Gupta. Efficient sequential consistency using conditional fences. In *PACT*, September 2010.
[21] C. Lin, V. Nagarajan, R. Gupta, and B. Rajaram. Efficient sequential consistency via conflict ordering. In *ASPLOS*, March 2012.
[22] S. Lu et al. Learning from mistakes: A comprehensive study on real world concurrency bug characteristics. In *ASPLOS*, March 2008.
[23] B. Lucia, L. Ceze, K. Strauss, S. Qadeer, and H.-J. Boehm. Conflict Exceptions: Simplifying concurrent language semantics with precise hardware exceptions for data-races. In *ISCA*, June 2010.
[24] D. Marino, A. Singh, T. Millstein, M. Musuvathi, and S. Narayanasamy. DRFx: A simple and efficient memory model for concurrent programming languages. In *PLDI*, June 2010.
[25] A. Meixner and D. J. Sorin. Dynamic verification of sequential consistency. In *ISCA*, June 2005.
[26] S. Narayanasamy, Z. Wang, J. Tigani, A. Edwards, and B. Calder. Automatically classifying benign and harmful data races using replay analysis. In *PLDI*, June 2007.
[27] S. Rajamani et al. ISOLATOR: Dynamically ensuring isolation in concurrent programs. In *ASPLOS*, March 2009.
[28] J. Renau, B. Fraguela, J. Tuck, W. Liu, M. Prvulovic, L. Ceze, S. Sarangi, P. Sack, K. Strauss, and P. Montesinos. SESC Simulator, January 2005. http://sesc.sourceforge.net.
[29] D. C. Schmidt and T. Harrison. Double-checked locking: An optimization pattern for efficiently initializing and accessing thread-safe objects. In *Patt. Lang. of Prog. Design*, 1996.
[30] D. Shasha and M. Snir. Efficient and Correct Execution of Parallel Programs that Share Memory. *ACM TOPLAS*, April 1988.
[31] Z. Sura, X. Fang, C.-L. Wong, S. P. Midkiff, J. Lee, and D. Padua. Compiler techniques for high performance sequentially consistent Java programs. In *PPoPP*, June 2005.
[32] T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos. Mechanisms for store-wait-free multiprocessors. In *ISCA*, June 2007.
[33] J. Yu and S. Narayanasamy. A case for an interleaving constrained shared-memory multi-processor. In *ISCA*, June 2009.

Appendix 1: Correctness Proofs

Theorem 1: An access C_i of processor P_C is Safe when ($SN_{C_i} \leq PP[P_C]$) and ($AD_{C_i}[P_K] \leq PP[P_K]$), for all processors $K \neq C$.
Proof: Recall that C_i becomes Safe as soon as it cannot participate in an SCV anymore. Assume that C_i can participate in an SCV with another access of P_C: either an earlier one C_{i-m} (Case 1) or a later one C_{i+m} (Case 2) in program order (Figure 17).

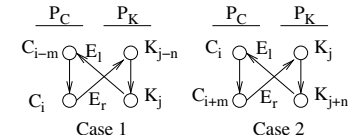


Figure 17. Possible cases for SCV.

Case 1: Consider two situations. In the first one, edge E_r occurs first. Although C_i has executed, it can still participate in an SCV for as long as P_C's previous accesses (C_{i-m} where $1 \leq m \leq i$) are not performed — since such accesses can still be the destination of an E_l edge. Hence, when $PP[P_C] \geq SN_{C_i}$, then C_i is Safe. The second situation is when edge E_l occurs first. C_i is not Safe until all of the P_K accesses up to K_j (K_{j-n} where $0 \leq n \leq j$) are performed, without consuming an edge from C_i. Note that the allowed destinations of C_i are the accesses after the E_l source K_j ($AD_{C_i}[P_K] = K_j$). The E_l edge can point to any P_C access preceding C_i. Hence, C_i is only Safe when it and all the previous accesses in P_C are performed, and all the accesses in P_K up to and including $AD_{C_i}[P_K]$ are performed. Hence, the Safe condition in Case 1 is ($SN_{C_i} \leq PP[P_C]$) and ($AD_{C_i}[P_K] \leq PP[P_K]$).

Case 2: There are two situations. In the first one, edge E_l occurs first. Although C_i has executed, it can still participate in an SCV for as long as its disallowed destinations in P_K (K_{j+n} and earlier) have not performed — since such accesses can be the destination of an E_r edge. Hence, when $AD_{C_i}[P_K] \leq PP[P_K]$, then C_i is Safe. The second situation is when E_r occurs first. In this case, when C_i performs, we know whether it creates a cycle with E_r. Hence, C_i is Safe when $SN_{C_i} \leq PP[P_C]$. Overall, the Safe condition in Case 2 is the same as Case 1, namely ($SN_{C_i} \leq PP[P_C]$) and ($AD_{C_i}[P_K] \leq PP[P_K]$).

Generalizing to all the processors, C_i is safe when ($SN_{C_i} \leq PP[P_C]$) and ($AD_{C_i}[P_K] \leq PP[P_K]$), for all processors $K \neq C$.

Theorem 2: In order to form an SCV cycle with two dependences, their source references have to be Unsafe with respect to their destination processors.
Proof: This is proved by contradiction. Assume that one of the dependences has a source that is Safe (with respect to the destination processor) and it forms an SCV cycle with another dependence whose source is Unsafe (with respect to the destination processor). According to the definition of a Safe access, once an access becomes Safe (with respect to a processor), no dependence from this access to an access of that other processor can cause an SCV. This contradicts our previous assumption and proves the theorem.

Theorem 3: If Vulcan records all the Unsafe dependences, then it detects all the SCVs between processors.
Proof: Referring to Figure 17, an access C_i can participate in an SCV with one of P_C's earlier accesses (Case 1) or one of the later accesses (Case 2). Without loss of generality, assume that, of the two dependence edges in the cycle, the edge at C_i is formed the latest. We now show that, when that edge is formed, C_i has all the information that it needs to detect the SCV.

In Case 1, the information that C_i needs to keep is the sources of the dependences pointing *to* any of the P_C accesses before C_i. More specifically, it needs to keep the maximum *SN* of such sources. With this information, it cannot miss a cycle when the edge at C_i occurs. But this is precisely the information in AD_{C_i}.

In Case 2, the information that C_i needs to keep is the destinations of the dependences pointing *from* any of the P_C accesses after C_i. More specifically, it needs to keep the minimum *SN* of such destinations. With it, C_i will not miss a cycle when the edge at C_i occurs. But this is precisely the information in AS_{C_i}.

Overall, if Vulcan records all the sources and destinations of the dependences (with AD and AS), it can find all the SCVs. Moreover, Theorem 2 proves that SCVs occur only among Unsafe dependences. Hence, if Vulcan records all the Unsafe dependences, then it detects all the SCVs.

2012 IEEE/ACM 45th Annual International Symposium on Microarchitecture

Amoeba-Cache: Adaptive Blocks for Eliminating Waste in the Memory Hierarchy*

Snehasish Kumar, Hongzhou Zhao†, Arrvindh Shriraman
Eric Matthews*, Sandhya Dwarkadas†, Lesley Shannon*

School of Computing Sciences, Simon Fraser University
†Department of Computer Science, University of Rochester
*School of Engineering Science, Simon Fraser University

Abstract

The fixed geometries of current cache designs do not adapt to the working set requirements of modern applications, causing significant inefficiency. The short block lifetimes and moderate spatial locality exhibited by many applications result in only a few words in the block being touched prior to eviction. Unused words occupy between 17—80% of a 64K L1 cache and between 1%—79% of a 1MB private LLC. This effectively shrinks the cache size, increases miss rate, and wastes on-chip bandwidth. Scaling limitations of wires mean that unused-word transfers comprise a large fraction (11%) of on-chip cache hierarchy energy consumption.

We propose Amoeba-Cache, a design that supports a variable number of cache blocks, each of a different granularity. Amoeba-Cache employs a novel organization that completely eliminates the tag array, treating the storage array as uniform and morphable between tags and data. This enables the cache to harvest space from unused words in blocks for additional tag storage, thereby supporting a variable number of tags (and correspondingly, blocks). Amoeba-Cache adjusts individual cache line granularities according to the spatial locality in the application. It adapts to the appropriate granularity both for different data objects in an application as well as for different phases of access to the same data. Overall, compared to a fixed granularity cache, the Amoeba-Cache reduces miss rate on average (geometric mean) by 18% at the L1 level and by 18% at the L2 level and reduces L1—L2 miss bandwidth by ≃46%. Correspondingly, Amoeba-Cache reduces on-chip memory hierarchy energy by as much as 36% (mcf) and improves performance by as much as 50% (art).

1 Introduction

A cache block is the fundamental unit of space allocation and data transfer in the memory hierarchy. Typically, a block is an aligned fixed granularity of contiguous words (1 word = 8bytes). Current processors fix the block granularity largely based on the average spatial locality across workloads, while taking tag overhead into consideration. Unfortunately, many applications (see Section 2 for details) exhibit low— moderate spatial locality and most of the words in a cache block are left untouched during the block's lifespan. Even for applications with good spatial behavior, the short lifespan of a block caused by cache geometry limitations can cause low cache utilization. Technology trends make it imperative that caching efficiency improves to reduce wastage of interconnect bandwidth. Recent reports from industry [2] show that on-chip networks can contribute up to 28% of total chip power. In the future an L2 — L1 transfer can cost up to 2.8× more energy than the L2 data access [14, 20]. Unused words waste ≃ 11% (4%—21% in commercial workloads) of the cache hierarchy energy.

*This material is based upon work supported in part by grants from the National Science and Engineering Research Council, MARCO Gigascale Research Center, Canadian Microelectronics Corporation, and National Science Foundation (NSF) grants CNS-0834451, CCF-1016902, and CCF-1217920.

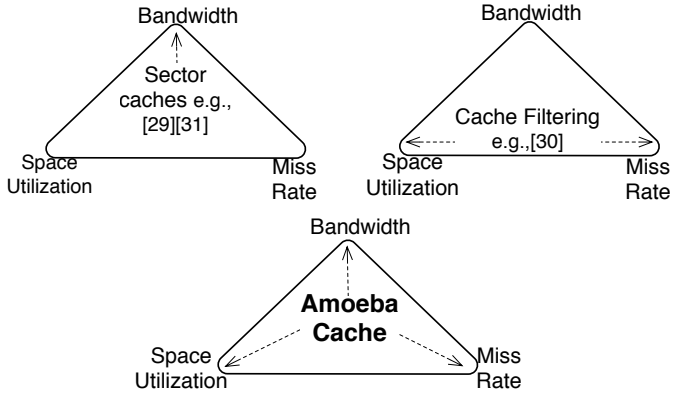


Figure 1: Cache designs optimizing different memory hierarchy parameters. Arrows indicate the parameters that are targeted and improved compared to a conventional cache.

Figure 1 organizes past research on cache block granularity along the three main parameters influenced by cache block granularity: miss rate, bandwidth usage, and cache space utilization. Sector caches have been used to [32, 15] minimize bandwidth by fetching only sub-blocks but miss opportunities for spatial prefetching. Prefetching [17, 29] may help reduce the miss rate for utilized sectors, but on applications with low—moderate or variable spatial locality, unused sectors due to misprediction, or unused regions within sectors, still pollute the cache and consume bandwidth. Line distillation [30] filters out unused words from the cache at evictions using a separate word-granularity cache. Other approaches identify dead cache blocks and replace or eliminate them eagerly [19, 18, 13, 21]. While these approaches improve utilization and potentially miss rate, they continue to consume bandwidth and interconnect energy for the unutilized words. Word-organized cache blocks also dramatically increase cache associativity and lookup overheads, which impacts their scalability.

Determining a fixed optimal point for the cache line granularity at hardware design time is a challenge. Small cache lines tend to fetch fewer unused words, but impose significant performance penalties by missing opportunities for spatial prefetching in applications with high spatial locality. Small line sizes also introduce high tag overhead, increase lookup energy, and increase miss processing overhead (e.g., control messages). Larger cache line sizes minimize tag overhead and effectively prefetch neighboring words but introduce the negative effect of unused words that increase network bandwidth. Prior approaches have proposed the use of multiple caches with different block sizes [33, 12]. These approaches require word granularity caches that increase lookup energy, impose high tag overhead (e.g., 50% in [33]), and reduce cache efficiency when there is good spatial locality.

1072-4451/12 $26.00 © 2012 IEEE
DOI 10.1109/MICRO.2012.42

IEEE computer society

In this paper, we propose a novel cache architecture, *Amoeba-Cache*, to improve memory hierarchy efficiency by supporting fine-grain (per-miss) dynamic adjustment of cache block size and the # of blocks per set. To enable variable granularity blocks within the same cache, the tags maintained per set need to grow and shrink as the # of blocks/set vary. *Amoeba-Cache* eliminates the conventional tag array and collocates the tags with the cache blocks in the data array. This enables us to segment and partition a cache set in different ways: For example, in a configuration comparable to a traditional 4-way 64K cache with 256 sets (256 bytes per set), we can hold eight 32-byte cache blocks, thirty-two 8-byte blocks, or any other collection of cache blocks of varying granularity. Different sets may hold blocks of different granularity, providing maximum flexibility across address regions of varying spatial locality. The *Amoeba-Cache* effectively filters out unused words in a conventional block and prevents them from being inserted into the cache, allowing the resulting free space to be used to hold tags or data of other useful blocks. The *Amoeba-Cache* can adapt to the available spatial locality; when there is low spatial locality, it will hold many blocks of small granularity and when there is good spatial locality, it can adapt and segment the cache into a few big blocks.

Compared to a fixed granularity cache, *Amoeba-Cache* improves cache utilization by 90% - 99% for most applications, saves miss rate by up to 73% (omnetpp) at the L1 level and up to 88% (twolf) at the LLC level, and reduces miss bandwidth by up to 84% (omnetpp) at the L1 and 92% (twolf) at the LLC. We compare against other approaches such as Sector Cache and Line distillation and show that *Amoeba-Cache* can optimize miss rate and bandwidth better across many applications, with lower hardware overhead. Our synthesis of the cache controller hit path shows that *Amoeba-Cache* can be implemented with low energy impact and 0.7% area overhead for a latency- critical 64K L1.

The overall paper is organized as follows: § 2 provides quantitative evidence for the acuteness of the spatial locality problem. § 3 details the internals of the *Amoeba-Cache* organization and § 4 analyzes the physical implementation overhead. § 5 deals with wider chip-level issues (i.e., inclusion and coherence). § 6 — § 10 evaluate the *Amoeba-Cache*, commenting on the optimal block granularity, impact on overall on-chip energy, and performance improvement. § 11 outlines related work.

2 Motivation for Adaptive Blocks

In traditional caches, the cache block defines the fundamental unit of data movement and space allocation in caches. The blocks in the data array are uniformly sized to simplify the insertion/removal of blocks, simplify cache refill requests, and support low complexity tag organization. Unfortunately, conventional caches are inflexible (fixed block granularity and fixed # of blocks) and caching efficiency is poor for applications that lack high spatial locality. Cache blocks influence multiple system metrics including bandwidth, miss rate, and cache utilization. The block granularity plays a key role in exploiting spatial locality by effectively prefetching neighboring words all at once. However, the neighboring words could go unused due to the low lifespan of a cache block. The unused words occupy interconnect bandwidth and pollute the cache, which increases the # of misses. We evaluate the influence of a fixed granularity block below.

2.1 Cache Utilization

In the absence of spatial locality, multi-word cache blocks (typically 64 bytes on existing processors) tend to increase cache pollution and fill the cache with words unlikely to be used. To quantify this pollution, we segment the cache line into words (8 bytes) and track the words touched before the block is evicted. We define utilization as the average # of words touched in a cache block before it is evicted. We study a comprehensive collection of workloads from a variety of domains: 6 from PARSEC [3], 7 from SPEC2006, 2 from SPEC2000, 3 Java workloads from DaCapo [4], 3 commercial workloads (Apache, SpecJBB2005, and TPC-C [22]), and the Firefox web browser. Subsets within benchmark suites were chosen based on demonstrated miss rates on the fixed granularity cache (i.e., whose working sets did not fit in the cache size evaluated) and with a spread and diversity in cache utilization. We classify the benchmarks into 3 groups based on the utilization they exhibit: Low (<33%), Moderate (33%—66%), and High (66%+) utilization (see Table 1).

Table 1: Benchmark Groups

Group	Utilization %	Benchmarks
Low	0 — 33%	art, soplex, twolf, mcf, canneal, lbm, omnetpp
Moderate	34 — 66%	astar, h2, jbb, apache, x264, firefox, tpc-c, freqmine, fluidanimate
High	67 — 100%	tradesoap, facesim, eclipse, cactus, milc, ferret

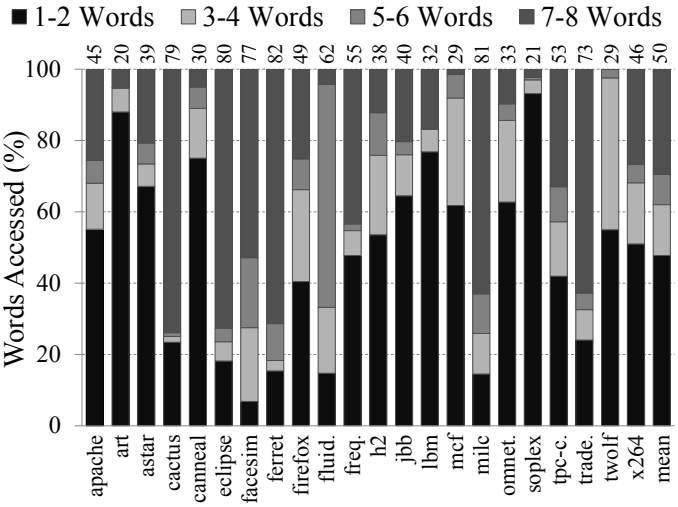


Figure 2: Distribution of words touched in a cache block. Avg. utilization is on top. (Config: 64K, 4 way, 64-byte block.)

Figure 2 shows the histogram of words touched at the time of eviction in a cache line of a 64K, 4-way cache (64-byte block, 8 words per block) across the different benchmarks. Seven applications have less than 33% utilization and 12 of them are dominated (>50%) by 1-2 word accesses. In applications with good spatial locality (cactus, ferret, tradesoap, milc, eclipse) more than 50% of the evicted blocks have 7-8 words touched. Despite similar average utilization for applications such as astar and h2 (39%), their distributions are dissimilar; ≃70% of the blocks in astar have 1-2 words accessed at the time of eviction, whereas ≃50% of the blocks in h2 have 1-2 words accessed per block. Utilization for a single application also changes over time; for example, ferret's average utilization, measured as the average fraction of words used in evicted cache lines over 50 million instruction windows, varies from 50% to 95% with a periodicity of roughly 400 million instructions.

2.2 Effect of Block Granularity on Miss Rate and Bandwidth

Cache miss rate directly correlates with performance, while under current and future wire-limited technologies [2], bandwidth directly correlates with dynamic energy. Figure 3 shows the influence of block granularity on miss rate and bandwidth for a 64K L1 cache and a 1M L2 cache keeping the number of ways constant. For the 64K L1, the plots highlight the pitfalls of simply decreasing the block size to accommodate the Low group of applications; miss rate increases by 2× for the High group when the block size is changed from 64B to 32B; it increases by 30% for the Moderate group. A smaller block size decreases bandwidth proportionately but increases miss rate. With a 1M L2 cache, the lifetime of the cache lines increases significantly, improving overall utilization. Increasing the block size from 64→256 halves the miss rate for all application groups. The bandwidth is increased by 2× for the Low and Moderate.

Since miss rate and bandwidth have different optimal block granularities, we use the following metric: $\frac{1}{MissRate \times Bandwidth}$ to determine a fixed block granularity suited to an application that takes both criteria into account. Table 2 shows the block size that maximizes the metric for each application. It can be seen that different applications have different block granularity requirements. For example, the metric is maximized for apache at 128 bytes and for firefox (similar utilization) at 32 bytes. Furthermore, the optimal block sizes vary with the cache size as the cache lifespan changes. This highlights the challenge of picking a single block size at design time especially when the working set does not fit in the cache.

2.3 Need for adaptive cache blocks

Our observations motivate the need for adaptive cache line granularities that match the spatial locality of the data access patterns in an application. In summary:

- Smaller cache lines improve utilization but tend to increase miss rate and potentially traffic for applications with good spatial locality, affecting the overall performance.
- Large cache lines pollute the cache space and interconnect with unused words for applications with poor spatial locality, significantly decreasing the caching efficiency.
- Many applications waste a significant fraction of the cache space. Spatial locality varies not only across applications but also within each application, for different data structures as well as different phases of access over time.

Table 2: Optimal block size. Metric: $\frac{1}{\text{Miss}-\text{rate} \times \text{Bandwidth}}$

64K, 4-way	
Block	Benchmarks
32B	cactus, eclipse, facesim, ferret, firefox, fluidanimate,freqmine, milc, tpc-c, tradesoap
64B	art
128B	apache, astar, canneal, h2, jbb, lbm, mcf, omnetpp, soplex, twolf, x264
1M, 8-way	
Block	Benchmarks
64B	apache, astar, cactus, eclipse, facesim, ferret, firefox, freqmine, h2, lbm, milc, omnetpp, tradesoap, x264
128B	art
256B	canneal, fluidanimate, jbb, mcf, soplex, tpc-c, twolf

3 Amoeba-Cache : Architecture

The *Amoeba-Cache* architecture enables the memory hierarchy to fetch and allocate space for a range of words (a variable granularity

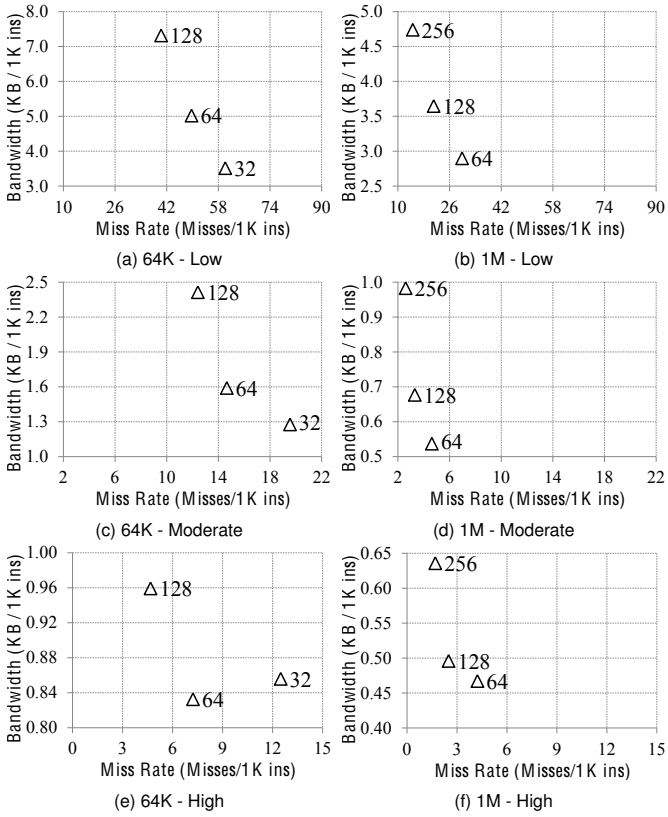


Figure 3: Bandwidth vs. Miss Rate. (a),(c),(e): 64K, 4-way L1. (b),(d),(f): 1M, 8-way LLC. Markers on the plot indicate cache block size. Note the different scales for different groups.

cache block) based on the spatial locality of the application. For example, consider a 64K cache (256 sets) that allocates 256 bytes per set. These 256 bytes can adapt to support, for example, eight 32-bytes blocks, thirty-two 8-byte blocks, or four 32-byte blocks and sixteen 8-byte blocks, based on the set of contiguous words likely to be accessed. The key challenge to supporting variable granularity blocks is how to grow and shrink the # of tags as the # of blocks per set vary with block granularity? *Amoeba-Cache* adopts a solution inspired by software data structures, where programs hold metadata and actual data entries in the same address space. To achieve maximum flexibility, *Amoeba-Cache* completely eliminates the tag array and collocates the tags with the actual data blocks (see Figure 4). We use a bitmap (`T? Bitmap`) to indicate which words in the data array represent tags. We also decouple the conventional valid/invalid bits (typically associated with the tags) and organize them into a separate array (V? : Valid bitmap) to simplify block replacement and insertion. V? and T? bitmaps both require 1 bit for very word (64bits) in the data array (total overhead of 3%). *Amoeba-Cache* tags are represented as a range (`Start` and `End` address) to support variable granularity blocks. We next discuss the overall architecture.

3.1 Amoeba Blocks and Set-Indexing

The *Amoeba-Cache* data array holds a collection of varied granularity *Amoeba-Block*s that do not overlap. Each *Amoeba-Block* is a 4 tuple consisting of `<Region Tag, Start, End, Data-Block>` (Figure 4). A `Region` is an aligned block of memory of size `RMAX` bytes. The boundaries of any *Amoeba-Block* block (`Start` and `End`) always will lie within the regions' boundaries. The minimum granularity of the data in an *Amoeba-Block* is 1 word

and the maximum is RMAX words. We can encode Start and End in $log_2(RMAX)$ bits. The set indexing function masks the lower $log_2(RMAX)$ bits to ensure that all *Amoeba-Block*s (every memory word) from a region index to the same set. The Region Tag and Set-Index are identical for every word in the *Amoeba-Block*. Retaining the notion of *sets* enables fast lookups and helps elude challenges such as synonyms (same memory word mapping to different sets). When comparing against a conventional cache, we set RMAX to 8 words (64 bytes), ensuring that the set indexing function is identical to that in the conventional cache to allow for a fair evaluation.

3.2 Data Lookup

Figure 5 describes the steps of the lookup. ❶ The lower $log_2(RMAX)$ bits are masked from the address and the set index is derived from the remaining bits. ❷ In parallel with the data read out, the T? bitmap activates the words in the data array corresponding to the tags for comparison. Note that given the minimum size of a *Amoeba-Block* is two words (1 word for the tag metadata, 1 word for the data), adjacent words cannot be tags. We need $\frac{N_{words}}{2}$ 2-1 multiplexers that route one of the adjacent words to the comparator ($\in$ operator). The comparator generates the hit signal for the word selector. The $\in$ operator consists of two comparators: a) an aligned Region tag comparator, a conventional == (64 - log_2N_{sets} - log_2RMAX bits wide, e.g., 50 bits) that checks if the *Amoeba-Block* belongs to the same region and b) a $Start <= W < END$ range comparator (log_2RMAX bits wide; e.g., 3 bits) that checks if the *Amoeba-Block* holds the required word. Finally, in ❸, the tag match activates and selects the appropriate word.

3.3 Amoeba Block Insertion

On a miss for the desired word, a spatial granularity predictor is invoked (see Section 5.1), which specifies the range of the *Amoeba-Block* to refill. To determine a position in the set to slot the incoming block we can leverage the V? (Valid bits) bitmap. The V? bitmap has 1 bit/word in the cache; a "1" bit indicates the word has been allocated (valid data or a tag). To find space for an incoming data block we perform a substring search on the V? bitmap of the cache set for contiguous 0s (empty words). For example, to insert an *Amoeba-Block* of five words (four words for data and one word for tag), we perform a substring search for *00000* in the V? bitmap / set (e.g., 32 bits wide for a 64K cache). If we cannot find the space, we keep triggering the replacement algorithm until we create the requisite contiguous space. Following this, the *Amoeba-Block* tuple (Tag and Data block) is inserted, and the corresponding bits in the T? and V? array are set. The 0s substring search can be accomplished with a lookup table; many current processors already include a substring instruction [11, PCMISTRI].

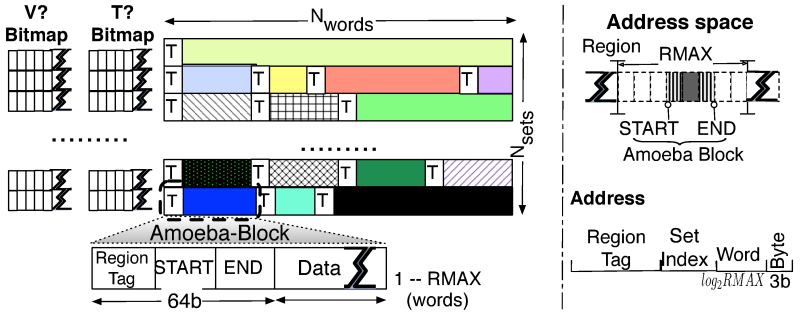
3.4 Replacement : Pseudo LRU

To reclaim the space from an *Amoeba-Block* the tag bits T? (tag) and V? (valid) bits corresponding to the block are unset. The key issue is identifying the *Amoeba-Block* to replace. Classical pseudo-LRU algorithms [16, 26, 16] keep the metadata for the replacement algorithm separate from the tags to reduce port contention. To be compatible with pseudo-LRU and other algorithms such as DIP [27] that work with a fixed # of ways, we can logically partition a set in *Amoeba-Cache* into N_{ways}. For instance, if we partition a 32 word cache set into 4 logical ways, any access to an *Amoeba-Block* tag found in words 0 —7 of the set is treated as an access to logical way 0. Finding a replacement candidate involves identifying the selected replacement way and then picking (possibly randomly) a candidate *Amoeba-Block* . More refined replacement algorithms that require per-tag metadata can harvest the space in the tag-word of the *Amoeba-Block* which is 64 bits wide (for alignment purposes) while physical addresses rarely extend beyond 48 bits.

3.5 Partial Misses

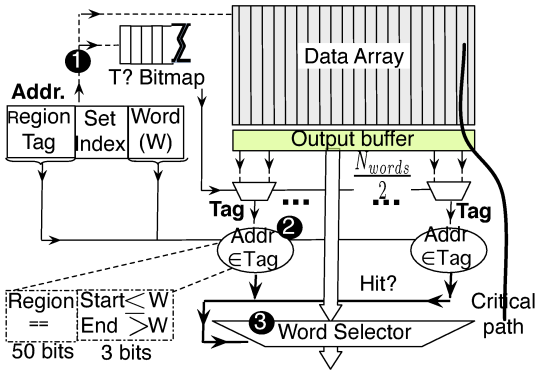
With variable granularity data blocks, a challenging although rare case (5 in every 1K accesses) that occurs is a *partial miss*. It is observed primarily when the spatial locality changes. Figure 6 shows an example. Initially, the set contains two blocks from a region R, one *Amoeba-Block* caches words 1 –3 (Region:R, START:1 END:3) and the other holds words 5 –6 (Region:R START:5 END:6). The CPU reads word 4, which misses, and the spatial predictor requests an *Amoeba-Block* with range START:0 and END:7. The cache has *Amoeba-Block*s that hold subparts of the incoming *Amoeba-Block*, and some words (0, 4, and 7) need to be fetched.

Amoeba-Cache removes the overlapping sub-blocks and allocates a new *Amoeba-Block*. This is a multiple step process: ❶ On a miss, the cache identifies the overlapping sub-blocks in the cache using the tags read out during lookup. $\cap \neq NULL$ is true if $START_{new} < END_{Tag}$ and $END_{new} > START_{Tag}$ (New = incoming block and



Left: Cache Layout. Size: $N_sets*N_words*8 bytes$. T? (Tag map) 1 bit/word. Is word a a tag? Yes(1). V? (Valid map): 1 bit/word. Is word allocated? Yes(1). Total overhead: 3%. **Right:** Address space layout: Region: Aligned regions of size RMAX words (×8 bytes). *Amoeba-Block* range cannot exceed Region boundaries. Cache Indexing: Lower log_2RMAX bits masked and set index is derived from remaining bits; all words (and all *Amoeba-Block*s) within a region index to the same set.

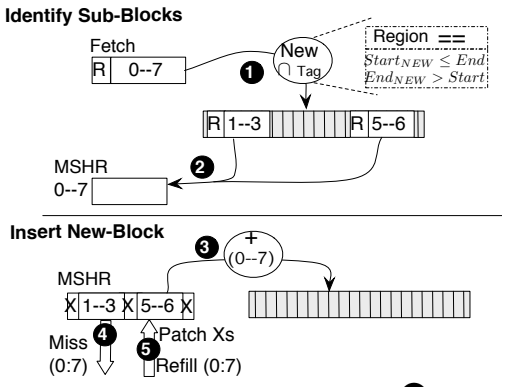
Figure 4: Amoeba-Cache Architecture.



Lookup steps. ❶ Mask lower log_2RMAX bits and index into data array. ❷ Identify tags using T? bitmap and initiate tag checks ($\in$) operator (Region tag comparator ==? and $\leq$ range comparator). ❸ Word selection within *Amoeba-Block*.

Figure 5: Lookup Logic

Tag = *Amoeba-Block* in set). ❷ The data blocks that overlap with the miss range are evicted and moved one-at-a-time to the MSHR entry. ❸ Space is then allocated for the new block, i.e., it is treated like a new insertion. ❹ A miss request is issued for the entire block (START:0 — END:7) even if only some words (e.g., 0, 4, and 7) may be needed. This ensures request processing is simple and only a single refill request is sent. ❺ Finally, the incoming data block is patched into the MSHR; only the words not obtained from the L1 are copied (since the lower level could be stale).



❶ Identify blocks overlapping with `New` block. ❷ Evict overlapping blocks to MSHR. ❸ Allocate space for new block (treat it like a new insertion). ❹ Issue refill request to lower level for entire block. ❺ Patch only newer words as lower-level data could be stale.

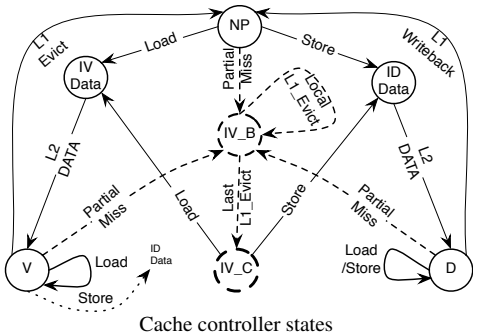
Figure 6: Partial Miss Handling. Upper: Identify relevant sub-blocks. Useful for other cache controller events as well, e.g., recalls. Lower: Refill of words and insertion.

4 Hardware Complexity

We analyze the complexity of *Amoeba-Cache* along the following directions: we quantify the additions needed to the cache controller, we analyze the latency, area, and energy penalty, and finally, we study the challenges specifically introduced by large caches.

4.1 Cache Controller

The variable granularity *Amoeba-Block* blocks need specific consideration in the cache controller. We focus on the L1 controller here, and in particular, partial misses. The cache controller manages operations at the aligned RMAX granularity. The controller permits only one in-flight cache operation per RMAX region. In-flight cache operations ensure no address overlap with stable *Amoeba-Block*s in order to eliminate complex race conditions. Figure 7 shows the L1 cache controller state machine. We add two states to the default protocol, `IV_B` and `IV_C`, to handle partial misses. `IV_B` is a blocking state that blocks other cache operations to RMAX region until all relevant *Amoeba-Block*s to a partial miss are evicted (e.g., 0–3 and 5–7 blocks in Figure 6). `IV_C` indicates partial miss completion. This enables the controller to treat the access as a full miss and issue the refill request. The other stable states (I, V, D) and transient states (`IV_Data` and `ID_Data`) are present in a conventional protocol as well. `Partial-miss` triggers the clean-up operations (`1` and `2` in Figure 6). `Local_L1_Evict` is a looping event that keeps retriggering for each *Amoeba-Block* involved in the partial miss; `Last_L1_Evict` is triggered when the last *Amoeba-Block* involved in the partial miss is evicted to the MSHR. A key difference



Cache controller states

State	Description
NP	*Amoeba-Block* not present in the cache.
V	All words corresponding to *Amoeba-Block* present and valid (read-only)
D	Valid and atleast one word in *Amoeba-Block* is dirty (read-write)
IV_B	Partial miss being processed (blocking state)
IV_Data	Load miss; waiting for data from L2
ID_Data	Store miss; waiting for data. Set dirty bit.
IV_C	Partial miss cleanup from cache completed (treat as full miss)
Amoeba-specific Cache Events	
Partial miss: Process partial miss.	
Local_L1_Evict: Remove overlapping *Amoeba-Block* to MSHR.	
Last_L1_Evict: Last *Amoeba-Block* moved to MSHR. Convert to full miss and process load or store.	

Bold and Broken-lines: *Amoeba-Cache* additions.

Figure 7: Amoeba Cache Controller (L1 level).

between the L1 and lower-level protocols is that the Load/Store event in the lower-level protocol may need to access data from multiple *Amoeba-Block*s. In such cases, similar to the partial-miss event, we read out each block independently before supplying the data (more details in § 5.2).

4.2 Area, Latency, and Energy Overhead

The extra metadata required by *Amoeba-Cache* are the T? (1 tag bit per word) and V? (1 valid bit per word) bitmaps. Table 3 shows the quantitative overhead compared to the data storage. Both the T? and V? bitmap arrays are directly proportional to the size of the cache and require a constant storage overhead (3% in total). The T? bitmap is read in parallel with the data array and does not affect the critical path; T? adds 2%—3.5% (depending on cache size) to the overall cache access energy. V? is referred only on misses when inserting a new block.

Table 3: *Amoeba-Cache* Hardware Complexity.

Cache configuration			
	64K (256by/set)	1MB (512by/set)	4MB (1024by/set)
Data RAM parameters			
Delay	0.36ns	2ns	2.5 ns
Energy	100pJ	230pJ	280pJ
Amoeba-Cache components (CACTI model)			
T?/V? map	1KB	16KB	64KB
Latency	0.019ns (5%)	0.12ns (6%)	0.2ns (6%)
Energy	2pJ (2%)	8pJ (3.4%)	10pJ (3.5%)
LRU	$\frac{1}{8}$KB	2KB	8KB
Lookup Overhead (VHDL model)			
Area	0.7%	0.1%	
Latency	0.02ns	0.035ns	0.04ns

% indicates overhead compared to data array of cache. 64K cache operates in *Fast mode*; 1MB and 4MB operate in *Normal mode*. We use 32nm ITRS HP transistors for 64K and 32nm ITRS LOP transistors for 1MB and 4MB.

We synthesized[1] the cache lookup logic using Synopsys and quantify the area, latency, and energy penalty. *Amoeba-Cache* is compatible with *Fast* and *Normal* cache access modes [28, -access-mode config], both of which read the entire set from the data array in parallel with the way selection to achieve lower latency. *Fast* mode transfers the entire set to the edge of the H-tree, while *Normal* mode, only transmits the selected way over the H-tree. For synthesis, we used the Synopsys design compiler (Vision Z-2007.03-SP5).

Figure 5 shows *Amoeba-Cache*ś lookup hardware on the critical path; we compare it against a fixed-granularity cache's lookup logic (mainly the comparators). The area overhead of the *Amoeba-Cache* includes registering an entire line that has been read out, the tag operation logic, and the word selector. The components on the critical path once the data is read out are the 2-way multiplexers, the $\in$ comparators, and priority encoder that selects the word; the T? bitmap is accessed in parallel and off the critical path. *Amoeba-Cache* is made feasible under today's wire-limited technology where the cache latency and energy is dominated by the bit/word lines, decoder, and H-tree [28]. *Amoeba-Cache*'s comparators, which operate on the entire cache set, are 6$\times$ the area of a fixed cache's comparators. Note that the data array occupies 99% of the overall cache area. The critical path is dominated by the wide word selector since the comparators all operate in parallel. The lookup logic adds 60% to the conventional cache's comparator time. The overall critical path is dominated by the data array access and *Amoeba-Cache*'s lookup circuit adds 0.02ns to the access latency and $\simeq$ 1pJ to the energy of a 64K cache, and 0.035ns to the latency and $\simeq$2pJ to the energy of a 1MB cache. Finally, *Amoeba-Cache* amortizes the energy penalty of the peripheral components (H-tree, Wordline, and decoder) over a single RAM.

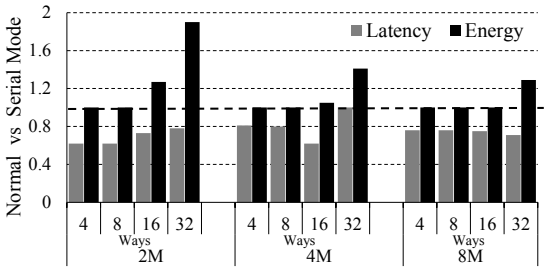
Amoeba-Cache's overhead needs careful consideration when implemented at the L1 cache level. We have two options for handling the latency overhead a) if the L1 cache is the critical stage in the pipeline, we can throttle the CPU clock by the latency overhead to ensure that the additional logic fits within the pipeline stage. This ensures that the number of pipeline stages for a memory access does not change with respect to a conventional cache, although all instructions bear the overhead of the reduced CPU clock. b) we can add an extra pipeline stage to the L1 hit path, adding a 1 cycle overhead to all memory accesses but ensuring no change in CPU frequency. We quantify the performance impact of both approaches in Section 6.

4.3 Tag-only Operations

Conventional caches support tag-only operations to reduce data port contention. While the *Amoeba-Cache* merges tags and data, like many commercial processors it decouples the replacement metadata and valid bits from the tags, accessing the tags only on cache lookup. Lookups can be either CPU side or network side (coherence invalidation and Wback/Forwarding). CPU-side lookups and writebacks ($\simeq$ 95% of cache operations) both need data and hence *Amoeba-Cache* in the common case does not introduce extra overhead. *Amoeba-Cache* does read out the entire data array unlike serial-mode caches (we discuss this issue in the next section). Invalidation checks and snoops can be more energy expensive with *Amoeba-Cache* compared to a conventional cache. Fortunately, coherence snoops are not common in many applications (e.g., 1/100 cache operations in SpecJBB) as a coherence directory and an inclusive LLC filter them out.

[1]We do not have access to an industry-grade 32nm library, so we synthesized at a higher 180nm node size and scaled the results to 32 nm (latency and energy scaled proportional to Vdd (taken from [36]) and Vdd^2 respectively).

4.4 Tradeoff with Large Caches



Baseline: Serial. $\leq$ 1 Normal is better. 32nm, ITRS LOP.

Figure 8: Serial vs Normal mode cache.

Large caches with many words per set ($\equiv$ highly associative conventional cache) need careful consideration. Typically, highly associative caches tend to serialize tag and data access with only the relevant cache block read out on a hit and no data access on a miss. We first analyze the the tradeoff between reading the entire set (normal mode), which is compatible with *Amoeba-Cache* and only the relevant block (serial mode). We vary the cache size from 2M—8M and associativity from 4(256B/set) — 32 (2048B/set). Under current technology constraints (Figure 8), only at very high associativity does serial mode demonstrate a notable energy benefit. Large caches are dominated by H-tree energy consumption and reading out the entire set at each sub-bank imposes an energy penalty when bitlines and wordlines dominate (2KB+ # of words/set).

Table 4: % of direct accesses with fast tags

	64K(256by/set)		1MB(512by/set)		2MB(1024 by/set)	
# Tags/set	2	4	4	8	8	16
Overhead	1KB	2KB	2KB	16KB	16KB	32KB
Benchmarks						
Low	30%	45%	42%	64%	55%	74%
Moderate	24%	62%	46%	70%	63%	85%
High	35%	79%	67%	95%	75%	96%

Amoeba-Cache can be tuned to minimize the hardware overhead for large caches. With many words/set the cache utilization improves due to longer block lifetimes making it feasible to support *Amoeba-Block*s with a larger minimum granularity ($>$ 1 word). If we increase minimum granularity to two or four words, only every third or fifth word could be a tag, meaning the # of comparators and multiplexers reduce to $\frac{N_{words/set}}{3}$ or $\frac{N_{words/set}}{5}$. When the minimum granularity is equal to max granularity (RMAX), we obtain a fixed granularity cache with $N_{words/set}/RMAX$ ways. Cache organizations that collocate all the tags together at the head of the data array enable tag-only operations and serial *Amoeba-Block* accesses that need to activate only a portion of the data array. However, the set may need to be compacted at each insertion. Recently, Loh and Hill [23] explored such an organization for supporting tags in multi-gigabyte caches.

Finally, the use of *Fast Tags* help reduce the tag lookups in the data array. Fast tags use a separate traditional tag array-like structure to cache the tags of the recently-used blocks and provide a pointer directly to the *Amoeba-Block*. The # of *Fast Tags* needed per set is proportional to the # of blocks in each set, which varies with the spatial locality in the application and the # of bytes per set (more details in Section 6.1). We studied 3 different cache configurations (64K 256B/set, 1M 512B/set, and 2M 1024B/set) while varying the number of fast tags per set (see Table 4). With 8 tags/set (16KB

overhead), we can filter 64—95% of the accesses in a 1MB cache and 55— 75% of the accesses in a 2MB cache.

5 Chip-Level Issues

5.1 Spatial Patterns Prediction

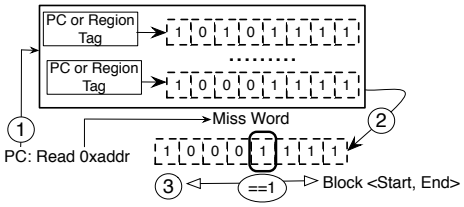


Figure 9: Spatial Predictor invoked on a *Amoeba-Cache* miss

Amoeba-Cache needs a spatial block predictor, which informs refill requests about the range of the block to fetch. *Amoeba-Cache* can exploit any spatial locality predictor and there have been many efforts in the compiler and architecture community [7, 17, 29, 6]. We adopt a table-driven approach consisting of a set of access bitmaps; each entry is `RMAX` (maximum granularity of an *Amoeba-Block*) bits wide and represents whether the word was touched during the lifetime of the recently evicted cache block. On a miss, the predictor will search for an entry (indexed by either the miss PC or region address) and choose the range of words to be fetched on a miss on either side (left and right) of the critical word. The PC-based indexing also uses the critical word index for improved accuracy. The predictor optimizes for spatial prefetching and will overfetch (bring in potentially untouched words), if they are interspersed amongst contiguous chunks of touched words. We can also bypass the prediction when there is low confidence in the prediction accuracy. For example, for streaming applications without repeated misses to a region, we can bring in a fixed granularity block based on the overall global behavior of the application. We evaluate tradeoffs in the design of the spatial predictor in Section 6.2.

5.2 Multi-level Caches

We discuss the design of inclusive cache hierarchies including multiple *Amoeba-Cache*s; we illustrate using a 2-level hierarchy. Inclusion means that the L2 cache contains a superset of the data words in the L1 cache; however, the two levels may include different granularity blocks. For example, the Sun Niagara T2 uses 16 byte L1 blocks and 64 byte L2 blocks. *Amoeba-Cache* permits non-aligned blocks of variable granularity at the L1 and the L2, and needs to deal with two issues: a) L2 recalls that may invalidate multiple L1 blocks and b) L1 refills that may need data from multiple blocks at the L2. For both cases, we need to identify all the relevant *Amoeba-Block*s that overlap with either the recall or the refill request. This situation is similar to a Nigara's L2 eviction which may need to recall 4 L1 blocks. *Amoeba-Cache*'s logic ensures that all *Amoeba-Block*s from a region map to a single set at any level (using the same RMAX for both L1 and L2). This ensures that L2 recalls or L1 refills index into only a single set. To process multiple blocks for a single cache operation, we use the step-by-step process outlined in Section 3.5 (❶ and ❷ in Figure 6). Finally, the L1-L2 interconnect needs 3 virtual networks, two of which, the L2→L1 data virtual network and the L1→L2 writeback virtual network, can have packets of variable granularity; each packet is broken down into a variable number of smaller physical flits.

5.3 Cache Coherence

There are three main challenges that variable cache line granularity introduces when interacting with the coherence protocol: 1) How is the coherence directory maintained? 2) How to support variable granularity read sharing? and 3) What is the granularity of write invalidations? The key insight that ensures compatibility with a conventional fixed-granularity coherence protocol is that a *Amoeba-Block* always lies within an aligned RMAX byte region (see § 3). To ensure correctness, it is sufficient to maintain the coherence granularity and directory information at a fixed granularity $\leq$ RMAX granularity. Multiple cores can simultaneously cache any variable granularity *Amoeba-Block* from the same region in *S*hared state; all such cores are marked as sharers in the directory entry. A core that desires exclusive ownership of an *Amoeba-Block* in the region uses the directory entry to invalidate every *Amoeba-Block* corresponding to the fixed coherence granularity. All *Amoeba-Block*s relevant to an invalidation will be found in the same set in the private cache (see set indexing in § 3). The coherence granularity could potentially be $<$ RMAX so that false sharing is not introduced in the quest for higher cache utilization (larger RMAX). The core claiming the ownership on a write will itself fetch only the desired granularity *Amoeba-Block*, saving bandwidth. A detailed evaluation of the coherence protocol is beyond the scope of the current paper.

6 Evaluation

Framework We evaluate the *Amoeba-Cache* architecture with the Wisconsin GEMS simulation infrastructure [25]; we use the in-order processor timing model. We have replaced the SIMICS functional simulator with the faster Pin [24] instrumentation framework to enable longer simulation runs. We perform timing simulation for 1 billion instructions. We warm up the cache using 20 million accesses from the trace. We model the cache controller in detail including the transient states needed for the multi-step cache operations and all the associated port and queue contention. We use a Full— LRU replacement policy, evicting *Amoeba-Block*s in LRU order until sufficient space is freed up for the block to be brought in. This helps decouple our observations from the replacement policy, enabling a fairer comparison with other approaches (Section 9). Our workloads are a mix of applications whose working sets stress our caches and includes SPEC- CPU benchmarks, Dacapo Java benchmarks [4], commercial workloads (SpecJBB2005, TPC-C, and Apache), and the Firefox web browser. Table 1 classifies the application categories: Low, Moderate, and High, based on the spatial locality. When presenting averages of ratios or improvements, we use the geometric mean.

6.1 Improved Memory Hierarchy Efficiency

Result 1:*Amoeba-Cache increases cache capacity by harvesting space from unused words and can achieve an 18% reduction in both L1 and L2 miss rate.*

Result 2:*Amoeba-Cache adaptively sizes the cache block granularity and reduces L1↔L2 bandwidth by 46% and L2↔Memory bandwidth by 38%.*

In this section, we compare the bandwidth and miss rate properties of *Amoeba-Cache* against a conventional cache. We evaluate two types of caches: a **Fixed** cache, which represents a conventional set-associative cache, and the *Amoeba-Cache*. In order to isolate the benefits of *Amoeba-Cache* from the potentially changing accuracy of the spatial predictor across different cache geometries, we use utilization at the next eviction as the spatial prediction, determined from a prior run on a fixed granularity cache. This also ensures that the spatial granularity predictions can be replayed across multiple simulation runs. To ensure equivalent data storage space, we set the *Amoeba-Cache* size to the sum of the tag array and the data array in a conventional cache. At the L1 level (64K), the net capacity of

the *Amoeba-Cache* is 64K + 8*4*256 bytes and at the L2 level (1M) configuration, it is 1M + 8*8*2048 bytes. The L1 cache has 256 sets and the L2 cache has 2048 sets.

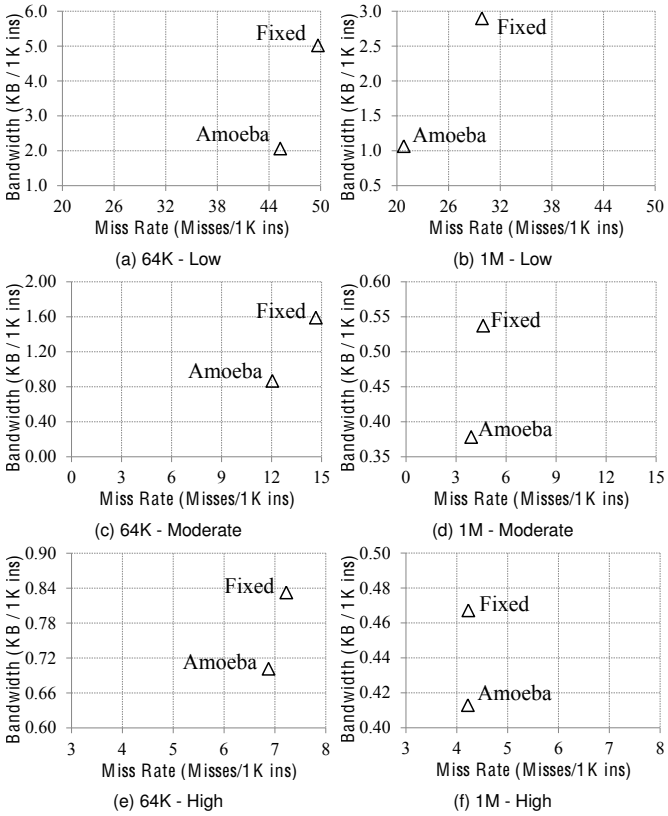


Figure 10: Fixed vs. Amoeba (Bandwidth and Miss Rate). Note the different scale for different application groups.

Figure 10 plots the miss rate and the traffic characteristics of the *Amoeba-Cache*. Since *Amoeba-Cache* can hold blocks varying from 8B to 64B, each set can hold more blocks by utilizing the space from untouched words. *Amoeba-Cache* reduces the 64K L1 miss rate by 23%(stdev:24) for the Low group, and by 21%(stdev:16) for the moderate group; even applications with high spatial locality experience a 7%(stdev:8) improvement in miss rate. There is a 46%(stdev:20) reduction on average in L1↔L2 bandwidth. At the 1M L2 level, *Amoeba-Cache* improves the moderate group's miss rate by 8%(stdev:10) and bandwidth by 23%(stdev:12). Applications with moderate utilization make better use of the space harvested from unused words by *Amoeba-Cache*. Many low utilization applications tend to be streaming and providing extra cache space does not help lower miss rate. However, by not fetching unused words, *Amoeba-Cache* achieves a significant reduction (38%(stdev:24) on average) in off-chip L2↔Memory bandwidth; even High utilization applications see a 17%(stdev:15) reduction in bandwidth. Utilization and miss rate are not, however, always directly correlated (more details in § 8).

With *Amoeba-Cache* the # of blocks/set varies based on the granularity of the blocks being fetched, which in turn depends on the spatial locality in the application. Table 5 shows the avg.# of blocks/set. In applications with low spatial locality, *Amoeba-Cache* adjusts the block size and adapts to store many smaller blocks. The 64K L1

Table 5: Avg. # of *Amoeba-Block* / Set

# Blocks/Set	64K Cache, 288 B/set
4—5	ferret, cactus, firefox, eclipse, facesim, freqmine, milc, astar
6—7	tpc-c, tradesoap, soplex, apache, fluidanimate
8—9	h2, canneal, omnetpp, twolf, x264, lbm, jbb
10—12	mcf, art
	1M Cache, 576 B/set
3—5	eclipse, omnetpp
8—9	cactus, firefox, tradesoap, freqmine, h2, x264, tpc-c
10—11	facesim, soplex, astar, milc, apache, ferret
12—13	twolf, art, jbb, lbm , fluidanimate
15—18	canneal, mcf

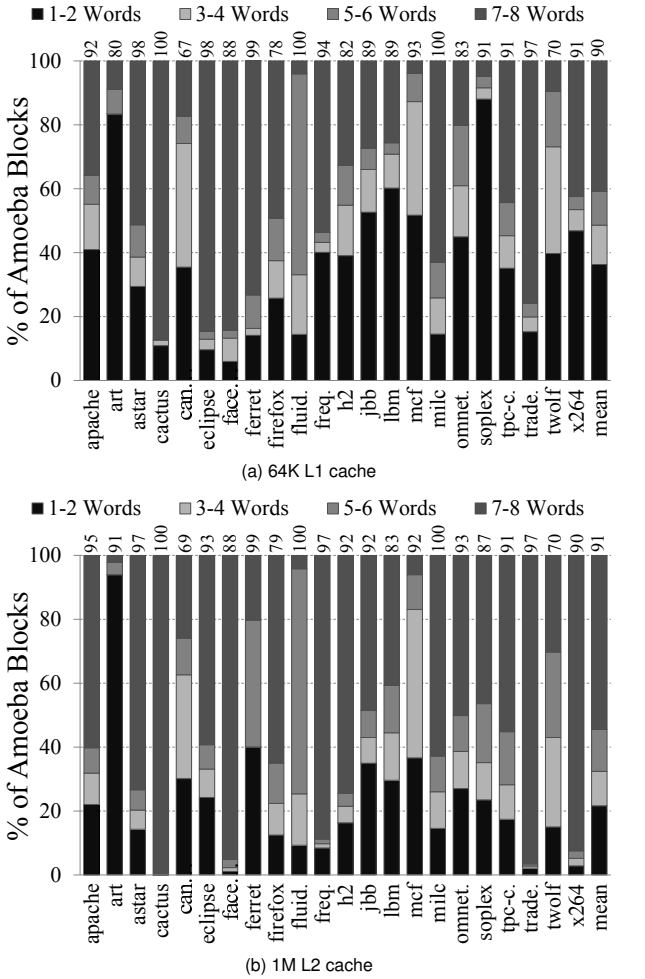


Figure 11: Distribution of cache line granularities in the 64K L1 and 1M L2 Amoeba-Cache. Avg. utilization is on top.

Amoeba-Cache stores 10 blocks per set for mcf and 12 blocks per set for art, effectively increasing associativity without introducing fixed hardware overheads. At the L2, when the working set starts to fit in the L2 cache, the set is partitioned into fewer blocks. Note that applications like eclipse and omnetpp hold only 3—5 blocks per set on average (lower than conventional associativity) due to their low miss rates (see Table 8). With streaming applications (e.g., canneal), *Amoeba-Cache* increases the # of blocks/set to >15 on average. Finally, some applications like apache store between 6—7 blocks/set with a 64K cache with varied block sizes (see Figure 11): approximately 50% of the blocks store 1-2 words and 30% of the blocks store 8 words at the L1. As the size of the cache increases and thereby the lifetime of the blocks, the *Amoeba-Cache* adapts to store larger size blocks as can be seen in Figure 11.

Utilization is improved greatly across all applications (90%+ in many cases). Figure 11 shows the distribution of cache block granularities in *Amoeba-Cache*. The *Amoeba-Block* distribution matches the word access distribution presented in § 2). With the 1M cache, the larger cache size improves block lifespan and thereby utilization, with a significant drop in the % of 1—2 word blocks. However, in many applications (tpc-c, apache, firefox, twolf, lbm, mcf), up to 20% of the blocks are 3–6 words wide, indicating the benefits of adaptivity and the challenges faced by Fixed.

6.2 Overall Performance and Energy

Result 3:*Amoeba-Cache improves overall cache efficiency and boosts performance by 10% on commercial applications*[2]*, saving up to 11% of the energy of the on-chip memory hierarchy. Off-chip L2↔memory energy sees a mean reduction of 41% across all workloads (86% for art and 93% for twolf)* .

We model a two-level cache hierarchy in which the L1 is a 64K cache with 256 sets (3 cycles load-to-use) and the L2 is 1M, 8192 sets (20 cycles). We assume a fixed memory latency of 300 cycles. We conservatively assume that the L1 access is the critical pipeline stage and throttle CPU clock by 4% (we evaluate an alternative approach in the next section). We calculate the total dynamic energy of the *Amoeba-Cache* using the energy #s determined in Section 4 through a combination of synthesis and CACTI [28]. We use 4 fast tags per set at the L1 and 8 fast tags per set at the L2. We include the penalty for all the extra metadata in *Amoeba-Cache*. We derive the energy for a single L1—L2 transfer ((6.8pJ per byte) from [2,28].The interconnect uses full-swing wires at 32nm, 0.6V.

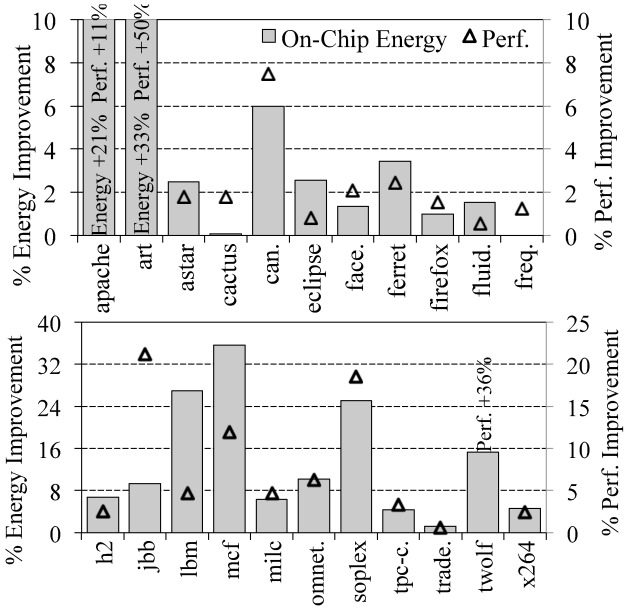


Figure 12: % improvement in performance and % reduction in on-chip memory hierarchy energy. Higher is better. Y-axis terminated to illustrate bars clearly. Baseline: Fixed, 64K L1, 1M L2.

Figure 12 plots the overall improvement in performance and reduction in on-chip memory hierarchy energy (L1 and L2 caches, and L1↔L2 interconnect). On applications that have good spatial locality (e.g., tradesoap, milc, facesim, eclipse, and cactus), *Amoeba-Cache* has minimal impact on miss rate, but provides significant bandwidth benefit. This results in on-chip energy reduction: milc's L1↔L2 bandwidth reduces by 15% and its on-chip energy reduces by 5%. Applications that suffer from cache pollution under Fixed (apache, jbb, twolf, soplex, and art) see gains in performance and energy. Apache's performance improves by 11% and on-chip energy reduces by 21%, while SpecJBB's performance improves by 21% and energy reduces by 9%. Art gains approximately 50% in performance. Streaming applications like mcf access blocks with both low and high utilization. Keeping out the unused words in the under-utilized blocks prevents the well-utilized cache blocks from being evicted; mcf's performance improves by 12% and on-chip energy by 36%.

Extra cache pipeline stage. An alternative strategy to accommodate *Amoeba-Cache*'s overheads is to add an extra pipeline stage to the cache access which increases hit latency by 1 cycle. The cpu clock frequency entails no extra penalty compared to a conventional cache. We find that for applications in the moderate and low spatial locality group (for 8 applications) *Amoeba-Cache* continues to provide a performance benefit between 6—50%. milc and canneal suffer minimal impact, with a 0.4% improvement and 3% slowdown respectively. Applications in the high spatial locality group (12 applications) suffer an average 15% slowdown (maximum 22%) due to the increase in L1 access latency. In these applications, 43% of the instructions (on average) are memory accesses and a 33% increase in L1 hit latency imposes a high penalty. Note that all applications continue to retain the energy benefit. The cache hierarchy energy is dominated by the interconnects and *Amoeba-Cache* provides notable bandwidth reduction. While these results may change for an out-of-order, multiple-issue processor, the evaluation suggests that *Amoeba-Cache* if implemented with the extra pipeline stage is more suited for lower levels in the memory hierarchy other than the L1.

Off-chip L2↔Memory energy The L2's higher cache capacity makes it less susceptible to pollution and provides less opportunity for improving miss rate. In such cases, *Amoeba-Cache* keeps out unused words and reduces off-chip bandwidth and thereby off-chip energy. We assume that the off-chip DRAM can provide adaptive granularity transfers for *Amoeba-Cache*'s L2 refills as in [35]. We use a DRAM model presented in a recent study [9] and model 0.5nJ per word transferred off-chip. The low spatial locality applications see a dramatic reduction in off-chip energy. For example, twolf sees a 93% reduction. On commercial workloads the off-chip energy decreases by 31% and 24% respectively. Even for applications with high cache utilization, off-chip energy decreases by 15%.

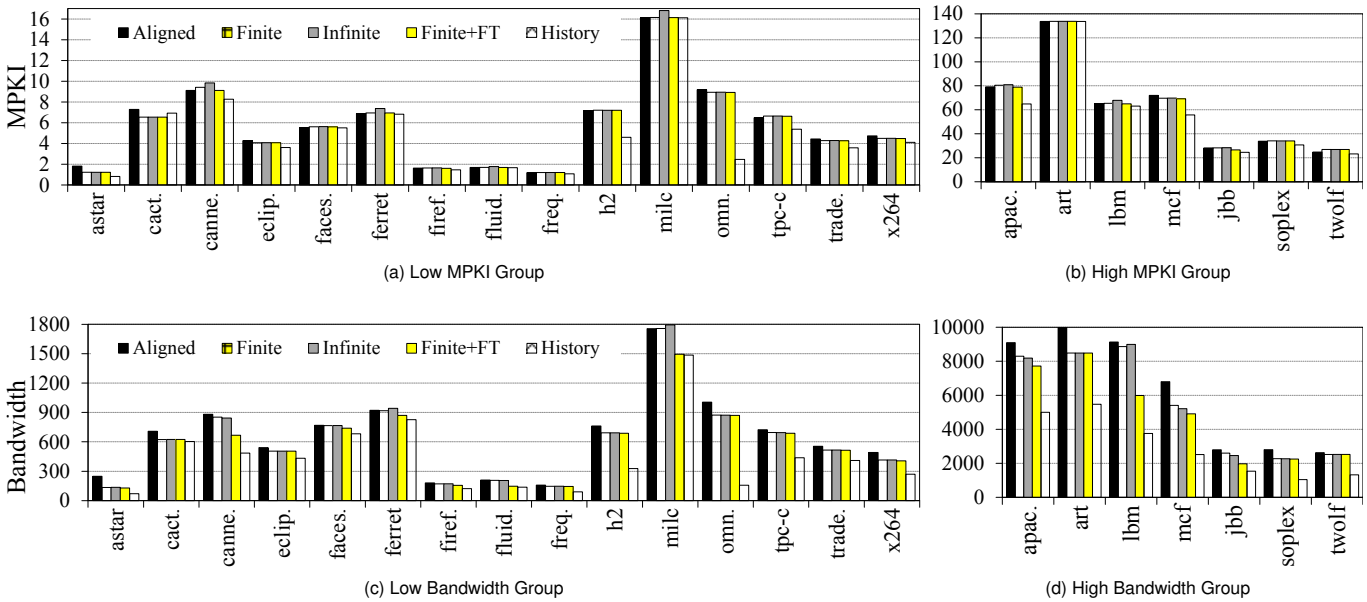
7 Spatial Predictor Tradeoffs

We evaluate the effectiveness of spatial pattern prediction in this section. In our table-based approach, a pattern history table records spatial patterns from evicted blocks and is accessed using a prediction index. The table-driven approach requires careful consideration of the following: prediction index, predictor table size, and training period. We quantify the effects by comparing the predictor against a baseline fixed-granularity cache. We use a 64K cache since it induces enough misses and evictions to highlight the predictor tradeoffs clearly.

7.1 Predictor Indexing

A critical choice with the history-based prediction is the selection of the predictor index. We explored two types of predictor indexing a) a PC-based approach [6] based on the intuition that fields in a data structure are accessed by specific PCs and tend to exhibit similar spatial behavior. The tag includes the the PC and the critical word index:

[2]"Commercial" applications includes Apache, SpecJBB and TPC-C.



ALIGNED: fixed-granularity cache (64B blocks). FINITE: *Amoeba-Cache* with a REGION predictor (1024 entry predictor table and 4K region size). INFINITE: *Amoeba-Cache* with an unbounded predictor table (REGION predictor).FINITE+FT is FINITE augmented with hints for predicting a default granularity on compulsory misses (first touches). HISTORY: *Amoeba-Cache* uses spatial pattern hints based on utilization at the next eviction, collected from a prior run.

$((PC >> 3) << 3)+\frac{(addr\%64)}{8}$. and b) a Region-based (REGION) approach that is based on the intuition that similar data objects tend to be allocated in contiguous regions of the address space and tend to exhibit similar spatial behavior. We compared the miss rate and bandwidth properties of both the PC (256 entries, fully associative) and REGION (1024 entries, 4KB region size) predictors. The size of the predictors was selected as the sweet spot in behavior for each predictor type. For all applications apart from cactus (a high spatial locality application), REGION-based prediction tends to overfetch and waste bandwidth as compared to PC-based prediction, which has 27% less bandwidth consumption on average across all applications. For 17 out of 22 applications, REGION-based prediction shows 17% better MPKI on average (max: 49% for cactus). For 5 applications (apache, art, mcf, lbm, and omnetpp), we find that PC demonstrates better accuracy when predicting the spatial behavior of cache blocks than REGION and demonstates a 24% improvement in MPKI (max: 68% for omnetpp).

7.2 Predictor Table

We studied the organization and size of the pattern table using the REGION predictor. We evaluated the following parameters a) region size, which directly correlates with the coverage of a fixed-size table, and b) the size of the predictor table, which influences how many unique region patterns can be tracked, and c) the # of bits required to represent the spatial pattern.

Large region sizes effectively reduce the # of regions in the working set and require a smaller predictor table. However, a larger region is likely to have more blocks that exhibit varied spatial behavior and may pollute the pattern entry. We find that going from 1KB (4096 entries) to 4KB (1024 entries) regions, the 4KB region granularity decreased miss rate by 0.3% and increased bandwidth by 0.4% even though both tables provide the same working set coverage (4MB). Fixing the region size at 4KB, we studied the benefits of an unbounded table. Compared to a 1024 entry table (FINITE in Figure 6.2), the unbounded table increases miss rate by 1% and decreases bandwidth by 0.3% . A 1024 entry predictor table (4KB region granularity per-entry) suffices for most applications. Organizing the 1024 entries as a 128-set×8-way table suffices for eliminating associativity related conflicts (<0.8% evictions due to lack of ways).

Focusing on the # of bits required to represent the pattern table, we evaluated the use of 4-bit saturation counters (instead of 1-bit bitmaps). The saturation counters seek to avoid pattern pollution when blocks with varied spatial behavior reside in the same region. Interestingly, we find that it is more beneficial to use 1-bit bitmaps for the majority of the applications (12 out of 22); the hysteresis introduced by the counters increases training period. To summarize, we find that a REGION predictor with region size 4KB and 1024 entries can predict the spatial pattern in a majority of the applications. CACTI indicates that the predictor table can be indexed in 0.025ns and requires 2.3pJ per miss indexing.

7.3 Spatial Pattern Training

A widely-used approach to training the predictor is to harvest the word usage information on an eviction. Unfortunately, evictions may not be frequent, which means the predictor's training period tends to be long, during which the cache performs less efficiently and/or that the application's phase has changed in the meantime. Particularly at the time of first touch (compulsory miss to a location), we need to infer the global spatial access patterns. We compare the finite region predictor (FINITE in Figure 6.2) that only predicts using eviction history, against a FINITE+FT: this adds the optimization of inferring the default pattern (in this paper, from a prior run) when there is no predictor information. FINITE+FT demonstrates an avg. 1% (max: 6% for jbb) reduction in miss rate compared to FINITE and comes within 16% the miss rate of HISTORY. In terms of bandwidth FINITE+FT can save 8% of the bandwidth (up

to 32% for lbm) compared to FINITE. The percentage of first-touch accesses is shown in Table 8.

7.4 Predictor Summary

- For the majority of the applications (17/22) the address-region predictor with region size 4KB works well. However, five applications (apache, lbm, mcf, art, omnetpp) show better performance with PC-based indexing. For best efficiency, the predictor should adapt indexing to each application.
- Updating the predictor only on evictions leads to long training periods, which causes loss in caching efficiency. We need to develop mechanisms to infer the default pattern based on global behavior demonstrated by resident cache lines.
- The online predictor reduces MPKI by 7% and bandwidth by 26% on average relative to the conventional approach. However, it still has a 14% higher MPKI and 38% higher bandwidth relative to the HISTORY-based predictor, indicating room for improvement in prediction.
- The 1024-entry (4K region size) predictor table imposes ≃0.12% energy overhead on the overall cache hierarchy energy since it is referenced only on misses.

8 *Amoeba-Cache* Adaptivity

We demonstrate that *Amoeba-Cache* can adapt better than a conventional cache to the variations in spatial locality.

Tuning RMAX for High Spatial Locality A challenge often faced by conventional caches is the desire to widen the cache block (to achieve spatial prefetching) without wasting space and bandwidth in low spatial locality applications. We study 3 specific applications: milc and tradesoap have good spatial locality and soplex has poor spatial locality. With a conventional 1M cache, when we widen the block size from 64 to 128 bytes, milc and tradesoap experience a 37% and 39% reduction in miss rate. However, soplex's miss rate increases by 2× and bandwidth by 3.1×.

With *Amoeba-Cache* we do not have to make this uneasy choice as it permits *Amoeba-Block*s with granularity 1—RMAX words (RMAX: maximum block size). When we increase RMAX from 64 bytes to 128 bytes, miss rate reduces by 37% for milc and 24% for tradesoap, while simultaneously lowering bandwidth by 7%. Unlike the conventional cache, *Amoeba-Cache* is able to adapt to poor spatial locality: soplex experiences only a 2% increase in bandwidth and 40% increase in miss rate.

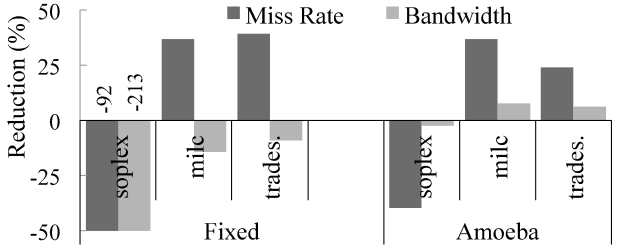


Figure 13: Effect of increase in block size from 64 to 128 bytes in a 1 MB cache

Predicting Strided Accesses Many applications (e.g., firefox and canneal) exhibit strided access patterns, which touch a few words in a block before accessing another block. Strided accesses patterns introduce intra-block holes (untouched words). For instance, canneal accesses ≃10K distinct fixed granularity cache blocks with a specific access pattern, **[--x--x--]** (**x** indicates i^{th} word has been touched). Any predictor that uses the access pattern history has two choices when an access misses on word 3 or 6 a) A miss oriented policy (Policy-Miss) may refill the entire range of words 3–6 and eliminate the secondary miss but bring in untouched words 4–5, increasing bandwidth, and b) a bandwidth focused choice (Policy- BW) that refills only the requested word but will incur more misses. Table 6 compares the two contrasting policies for *Amoeba-Cache* (relative to a Fixed granularity baseline cache). Policy-BW saves 9% bandwidth compared to Policy-Miss but suffer 25-30% higher miss rate.

Table 6: Predictor Policy Comparison

	canneal		firefox	
	Miss Rate	BW	Miss Rate	BW
Policy-Miss	10.31%	81.2%	11.18%	47.1%
Policy-BW	–20.08%	88.09%	–13.44%	56.82%
Spatial Patterns	**[--x--x--]**	**[x--x----]**	**[-x--x---]**	**[x---x---]**

–: indicates Miss or BW higher than Fixed.

9 *Amoeba-Cache* vs other approaches

We compare *Amoeba-Cache* against four approaches.

- **Fixed-2x**: Our baseline is a fixed granularity cache 2× the capacity of the other designs (64B block).
- **Sector** is the conventional sector cache design (as in IBM Power7 [15]): 64B block and a small sector size (16bytes or 2 words). This design targets optimal bandwidth. On any access, the cache fetches only the requisite sector, even though it allocates space for the entire line.
- **Sector-Pre** adds prefetching to Sector. This optimized design prefetches multiple sectors based on a spatial granularity predictor to improve the miss rate [17, 29].
- **Multi$** combines features from line distillation [30] and spatial-temporal caches [12]. It is an aggressive design that partitions a cache into two: a line organized cache (LOC) and a word-organized cache (WOC). At insertion time, Multi$ uses the spatial granularity hints to direct the cache to either refill words (into the WOC) or the entire line. We investigate two design points: 50% of the cache as a WOC (Multi$-50) and 25% of cache as a WOC (Multi$-25).

Sector, Sector-Pre, Multi$, and *Amoeba-Cache* all use the same spatial predictor hints. On a demand miss, they prefetch all the sub-blocks needed together. Prefetching only changes the timing of an access; it does not change the miss bandwidth and cannot remove the misses caused by cache pollution.

Energy and Storage The sector approaches impose low hardware complexity and energy penalty. To filter out unused words, the sector granularity has to be close to word granularity; we explored 2words/sector which leads to a storage penalty of ≃64KB for a 1MB cache. In Multi$, the WOC increases associativity to the number of words/block. Multi$-25 partitions a 64K 4-way cache into a 48K 3-way LOC and a 16K 8-way WOC, increasing associativity to 11. For a 1M cache, Multi$-50 increases associativity to 36. Compared to Fixed, Multi$-50 imposes over 3× increase in lookup latency, 5× increase in lookup energy, and ≃4× increase in tag storage. *Amoeba-Cache* provides a more scalable approach to using adaptive cache lines since it varies the storage dedicated to the tags based on the spatial locality in the application.

Miss Rate and Bandwidth Figure 14 summarizes the comparison; we focus on the moderate and low utilization groups of applica-

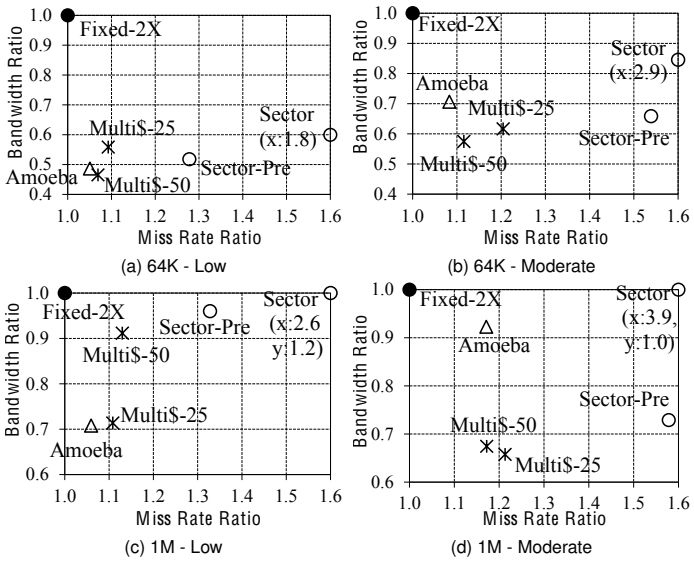


Figure 14: Relative miss rate and bandwidth for different caches. Baseline (1,1) is the Fixed-2x design. Labels: • Fixed-2x, ○ Sector approaches. ∗ : Multi$, △ Amoeba. (a),(b) 64K cache (c),(d) 1M cache. Note the different Y-axis scale for each group.

tions. On the high utilization group, all designs other than Sector have comparable miss rates. *Amoeba-Cache* improves miss rate to within 5%—6% of the Fixed-2× for the low group and within 8%—17% for the moderate group. Compared to the Fixed-2×, *Amoeba-Cache* also lowers bandwidth by 40% (64K cache) and 20% (1M cache). Compared to Sector-Pre (with prefetching), *Amoeba-Cache* is able to adapt better with flexible granularity and achieves lower miss rate (up to 30% @ 64K and 35% @ 1M). Multi$'s benefits are proportional to the fraction of the cache organized as a WOC; Multi$-50 (18-way@64K and 36-way@1M) is needed to match the miss rate of *Amoeba-Cache*. Finally, in the moderate group, many applications exhibit strided access. Compared to Multi-$'s WOC, which fetches individual words, *Amoeba-Cache* increases bandwidth since it chooses to fetch the contiguous chunk in order to lower miss rate.

10 Multicore Shared Cache

We evaluate a shared cache implemented with the *Amoeba-Cache* design. By dynamically varying the cache block size and keeping out unused words, the *Amoeba-Cache* effectively minimizes the footprint of an application. Minimizing the footprint helps multiple applications effectively share the cache space. We experimented with a 1M shared *Amoeba-Cache* in a 4 core system. Table 7 shows the application mixes; we chose a mix of applications across all groups. We tabulate the change in miss rate per thread and the overall change in bandwidth for *Amoeba-Cache* with respect to a fixed granularity cache running the same mix. Minimizing the overall footprint enables a reduction in the miss rate of each application in the mix. The commercial workloads (SpecJBB and TPC-C) are able to make use of the space available and achieve a significant reduction in miss rate (avg: 18%). Only two applications suffered a small increase in miss rate (x264 Mix#2: 2% and ferret Mix#3: 4%) due to contention. The overall L2 miss bandwidth significantly improves, showing 16%—39% reduction across all workload mixes. We believe that the Amoeba-based shared cache can effectively enable the shared cache to support more cores and increase overall throughput. We leave the design space exploration of an Amoeba-based coherent cache for future work.

Table 7: Multiprogrammed Workloads on 1M Shared *Amoeba-Cache*% reduction in miss rate and bandwidth. Baseline: Fixed 1M.

	Miss	Miss	Miss	Miss	BW
Mix	T1	T2	T3	T4	(All)
jbb×2, tpc-c×2	12.38%	12.38%	22.29%	22.37%	39.07%
firefox×2, x264×2	3.82%	3.61%	–2.44%	0.43%	15.71%
cactus, fluid., omnet., sopl.	1.01%	1.86%	22.38%	0.59%	18.62%
canneal, astar, ferret, milc	4.85%	2.75%	19.39%	–4.07%	17.77%

–: indicates Miss or BW higher than Fixed. T1—T4, threads in the mix; in the order of applications in the mix

11 Related Work

Burger et al. [5] defined cache efficiency as the fraction of blocks that store data that is likely to be used. We use the term cache utilization to identify touched versus untouched words residing in the cache. Past works [6, 29, 30] have also observed low cache utilization at specific levels of the cache. Some works [18, 19, 13, 21] have sought to improve cache utilization by eliminating cache blocks that are no longer likely to be used (referred to as dead blocks). These techniques do not address the problem of intra-block waste (i.e., untouched words).

Sector caches [31, 32] associate a single tag with a group of contiguous cache lines, allowing cache sizes to grow without paying the penalty of additional tag overhead. Sector caches use bandwidth efficiently by transferring only the needed cache lines within a sector. Conventional sector caches [31] may result in worse utilization due to the space occupied by invalid cache lines within a sector. Decoupled sector caches [32] help reduce the number of invalid cache lines per sector by increasing the number of tags per sector. Compared to the Amoeba cache, the tag space is a constant overhead, and limits the # of invalid sectors that can be eliminated. Pujara et al. [29] consider a word granularity sector cache, and use a predictor to try and bring in only the used words. Our results (see Figure 14) show that smaller granularity sectors significantly increase misses, and optimizations that prefetch [29] can pollute the cache and interconnect with unused words.

Line distillation [30] applies filtering at the word granularity to eliminate unused words in a cache block at eviction. This approach requires part of the cache to be organized as a word-organized cache, which increases tag overhead, complicates lookup, and bounds performance improvements. Most importantly, line distillation does not address the bandwidth penalty of unused words. This inefficiency is increasingly important to address under current and future technology dominated by interconnects [14, 28]. Veidenbaum et al. [33] propose that the entire cache be word organized and propose an online algorithm to prefetch words. Unfortunately, a static word-organized cache has a built-in tag overhead of 50% and requires energy-intensive associative searches.

Amoeba-Cache adopts a more proactive approach that enables continuous dynamic block granularity adaptation to the available spatial locality. When there is high spatial locality, the *Amoeba-Cache* will automatically store a few big cache blocks (most space dedicated for data); with low spatial locality, it will adapt to storing many small cache blocks (extra space allocated for tags). Recently, Yoon et

al. have proposed an adaptive granularity DRAM architecture [35]. This provides the support necessary for supporting variable granularity off-chip requests from an *Amoeba-Cache*-based LLC. Some research [10,8] has also focused on reducing false sharing in coherent caches by splitting/merging cache blocks to avoid invalidations. They would benefit from the *Amoeba-Cache* design, which manages block granularity in hardware.

There has a been a significant amount of work at the compiler and software runtime level (e.g. [7]) to restructure data for improved spatial efficiency. There have also been efforts from the architecture community to predict spatial locality [29, 34, 17, 36], which we can leverage to predict *Amoeba-Block* ranges. Finally, cache compression is an orthogonal body of work that does not eliminate unused words but seeks to minimize the overall memory footprint [1].

12 Summary

In this paper, we propose a cache design, Amoeba-Cache, that can dynamically hold a variable number of cache blocks of different granularities. The Amoeba-Cache employs a novel organization that completely eliminates the tag array and collocates the tags with the cache block in the data array. This permits the Amoeba-Cache to trade the space budgeted for the cache blocks for tags and support a variable number of tags (and blocks). For applications that have low spatial locality, *Amoeba-Cache* can reduce cache pollution, improve the overall miss rate, and reduce bandwidth wasted in the interconnects. When applications have moderate to high spatial locality, *Amoeba-Cache* coarsens the block size and ensures good performance. Finally, for applications that are streaming (e.g., lbm), *Amoeba-Cache* can save significant energy by eliminating unused words from being transmitted over the interconnects.

Acknowledgments

We would like to thank our shepherd, André Seznec, and the anonymous reviewers for their comments and feedback.

Appendix

Table 8: *Amoeba-Cache* Performance. Absolute #s.

	MPKI		BW bytes/1K		CPI	Predictor Stats	
	L1 MPKI	L2 MPKI	L1⟷L2 #Bytes/1K	L2⟷Mem #Bytes/1K	Cycles/Ins.	First Touch % Misses	Evict Win. # ins./Evict
apache	64.9	19.6	5,000	2,067	8.3	0.4	17
art	133.7	53.0	5,475	1,425	16.0	0.0	9
astar	0.9	0.3	70	35	1.9	18.0	1,600
cactus	6.9	4.4	604	456	3.5	7.5	162
canne.	8.3	5.0	486	357	3.2	5.8	128
eclip.	3.6	<0.1	433	<1	1.8	0.1	198
faces.	5.5	4.7	683	632	3.0	41.2	190
ferre.	6.8	1.4	827	83	2.1	1.3	156
firef.	1.5	1.0	123	95	2.1	11.1	727
fluid.	1.7	1.4	138	127	1.9	39.2	629
freqm.	1.1	0.6	89	65	2.3	17.7	994
h2	4.6	0.4	328	46	1.8	1.7	154
jbb	24.6	9.6	1,542	830	5.0	10.2	42
lbm	63.1	42.2	3,755	3,438	13.6	6.7	18
mcf	55.8	40.7	2,519	2,073	13.2	0.0	19
milc	16.1	16.0	1,486	1,476	6.0	2.4	66
omnet.	2.5	<0.1	158	<1	1.9	0.0	458
sople.	30.7	4.0	1,045	292	3.1	0.9	35
tpcc	5.4	0.5	438	36	2.0	0.4	200
trade.	3.6	<0.1	410	6	1.8	0.6	194
twolf	23.3	0.6	1,326	45	2.2	0.0	49
x264	4.1	1.8	270	190	2.2	12.4	274

MPKI : Misses / 1K instructions. BW: # words / 1K instructions
CPI: Clock cycles per instruction.
Predictor First touch: Compulsory misses. % of accesses that use default granularity.
Evict window: # of instructions between evictions. Higher value indicates predictor training takes longer.

References

[1] A. R. Alameldeen. Using Compression to Improve Chip Multiprocessor Performance. In *Ph.D. dissertation, Univ. of Wisconsin-Madison*, 2006.

[2] D. Albonesi, A. Kodi, and V. Stojanovic. NSF Workshop on Emerging Technologies for Interconnects (WETI). 2012.

[3] C. Bienia. Benchmarking Modern Multiprocessors. In *Ph.D. Thesis. Princeton University*, 2011.

[4] S. M. Blackburn et al. The DaCapo benchmarks: java benchmarking development and analysis. In *Proc. of the 21st OOPSLA*, 2006.

[5] D. Burger, J. Goodman, and A. Kaigi. The Declining Effectiveness of Dynamic Caching for General-Purpose Workloads. In *University of Wisconsin Technical Report*, 1995.

[6] C. F. Chen et al. Accurate and Complexity-Effective Spatial Pattern Prediction. In *Proc. of the 10th HPCA*. 2004.

[7] T. M. Chilimbi, M. D. Hill, and J. R. Larus. Cache-Conscious Structure Layout. In *Proc. of the PLDI*, 1999.

[8] B. Choi et al. DeNovo: Rethinking the Memory Hierarchy for Disciplined Parallelism. In *Proc. of the PACT*, 2011.

[9] B. Dally. Power, Programmability, and Granularity: The Challenges of ExaScale Computing. In *Proc. of the IPDPS*, 2011.

[10] C. Dubnicki and T. J. Leblanc. Adjustable Block Size Coherent Caches. In *Proc. of the 19th ISCA*, 1992.

[11] A. Fog. Instruction tables. 2012. `http://www.agner.org/optimize/instruction_tables.pdf`.

[12] A. González, C. Aliagas, and M. Valero. A data cache with multiple caching strategies tuned to different types of locality. In *Proc. of the ACM Intl. Conf. on Supercomputing*, 1995.

[13] Z. Hu, S. Kaxiras, and M. Martonosi. Timekeeping in the memory system: predicting and optimizing memory behavior. In *Proc. of the 29th ISCA*, 2002.

[14] C. Hughes, C. Kim, and Y.-K. Chen. Performance and Energy Implications of Many-Core Caches for Throughput Computing. In *IEEE Micro Journal*, 2010.

[15] R. Kalla, B. Sinharoy, W. J. Starke, and M. Floyd. Power7: IBM's Next-Generation Server Processor. In *IEEE Micro Journal*, 2010.

[16] K. Kedzierski, M. Moreto, F. J. Cazorla, and M. Valero. Adapting Cache Partitioning Algorithms to Pseudo-LRU Replacement Policies. In *Proc. of the IPDPS*, 2010.

[17] S. Kumar and C. Wilkerson. Exploiting spatial locality in data caches using spatial footprints. In *Proc. of the 25th ISCA*, 1998.

[18] A.-C. Lai and B. Falsafi. Selective, accurate, and timely self-invalidation using last-touch prediction. In *Proc. of the 27th ISCA*, 2000.

[19] A. R. Lebeck and D. A. Wood. Dynamic self-invalidation: reducing coherence overhead in shared-memory multiprocessors. In *Proc. of the 22nd ISCA*. 1995.

[20] S. Li et al. McPAT: an integrated power, area, and timing modeling framework for multicore and manycore architectures. In *Proc. of the 42nd MICRO*, 2009.

[21] H. Liu et al. Cache bursts: A new approach for eliminating dead blocks and increasing cache efficiency. In *Proc. of the 41st MICRO*, 2008.

[22] D. R. Llanos and B. Palop. TPCC-UVA: An Open-source TPC-C implementation for parallel and distributed systems. In *Proc. of the 2006 Intl. Parallel and Distributed Processing Symp. PMEO Workshop*, 2006. `http://www.infor.uva.es/~diego/tpcc-uva.html`.

[23] G. H. Loh and M. D. Hill. Efficiently enabling conventional block sizes for very large die-stacked DRAM caches. In *Proc. of the 44th Intl. Symp. on Microarchitecture*, 2011.

[24] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proc. of the PLDI*, 2005.

[25] Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset. In *ACM SIGARCH Computer Architecture News*, Sept. 2005.

[26] S. Microsystems. OpenSPARC T1 Processor Megacell Specification. 2007.

[27] Moinuddin K. Qureshi et al. Adaptive insertion policies for high performance caching In *Prof. of the 34th ISCA*, 2007.

[28] N. Muralimanohar, R. Balasubramonian, and N. Jouppi. Optimizing NUCA Organizations and Wiring Alternatives for Large Caches with CACTI 6.0. In *Proc. of the 40th MICRO*, 2007.

[29] P. Pujara and A. Aggarwal. Increasing the Cache Efficiency by Eliminating Noise. In *Proc. of the 12th HPCA*, 2006.

[30] M. K. Qureshi, M. A. Suleman, and Y. N. Patt. Line Distillation: Increasing Cache Capacity by Filtering Unused Words in Cache Lines. In *Proc. of the 13th HPCA*, 2007.

[31] J. B. Rothman and A. J. Smith. The pool of subsectors cache design. In *Proc. of the 13th ACM ICS*, 1999.

[32] A. Seznec. Decoupled sectored caches: conciliating low tag implementation cost. In *Proc. of the 21st ISCA*, 1994.

[33] A. V. Veidenbaum, W. Tang, R. Gupta, A. Nicolau, and X. Ji. Adapting cache line size to application behavior. In *Proc. of the 13th ACM ICS*. 1999.

[34] M. A. Watkins, S. A. Mckee, and L. Schaelicke. Revisiting Cache Block Superloading. In *Proc. of the 4th HIPEAC*. 2009.

[35] D. H. Yoon, M. K. Jeong, and M. Erez. Adaptive granularity memory systems: a tradeoff between storage efficiency and throughput. In *Proc. of the 38th ISCA*. 2011.

[36] D. H. Yoon, M. K. Jeong, M. B. Sullivan, and M. Erez. The Dynamic Granularity Memory System. In *Proc. of the 39th ISCA*, 2012.

[36] `http://cpudb.stanford.edu/`

2012 IEEE/ACM 45th Annual International Symposium on Microarchitecture

Improving Cache Management Policies Using Dynamic Reuse Distances

Nam Duong†, Dali Zhao†, Taesu Kim†, Rosario Cammarota†,
Mateo Valero§ and Alexander V. Veidenbaum†
†*University of California, Irvine* §*Universitat Politecnica de Catalunya, Barcelona Supercomputing Center*
{nlduong, daliz, tkim15, rcammaro, alexv}@ics.uci.edu, mateo.valero@bsc.es

Abstract

Cache management policies such as replacement, bypass, or shared cache partitioning have been relying on data reuse behavior to predict the future. This paper proposes a new way to use dynamic reuse distances to further improve such policies. A new replacement policy is proposed which prevents replacing a cache line until a certain number of accesses to its cache set, called a Protecting Distance (PD). The policy protects a cache line long enough for it to be reused, but not beyond that to avoid cache pollution. This can be combined with a bypass mechanism that also relies on dynamic reuse analysis to bypass lines with less expected reuse. A miss fetch is bypassed if there are no unprotected lines. A hit rate model based on dynamic reuse history is proposed and the PD that maximizes the hit rate is dynamically computed. The PD is recomputed periodically to track a program's memory access behavior and phases.

Next, a new multi-core cache partitioning policy is proposed using the concept of protection. It manages lifetimes of lines from different cores (threads) in such a way that the overall hit rate is maximized. The average per-thread lifetime is reduced by decreasing the thread's PD.

The single-core PD-based replacement policy with bypass achieves an average speedup of 4.2% over the DIP policy, while the average speedups over DIP are 1.5% for dynamic RRIP (DRRIP) and 1.6% for sampling dead-block prediction (SDP). The 16-core PD-based partitioning policy improves the average weighted IPC by 5.2%, throughput by 6.4% and fairness by 9.9% over thread-aware DRRIP (TA-DRRIP). The required hardware is evaluated and the overhead is shown to be manageable.

1 INTRODUCTION

Reduction in cache miss rates continues to be an important issue in processor design, especially at the last level cache (LLC). Cache management policies such as replacement, bypass or shared cache partitioning, have been relying – directly or indirectly – on the data reuse behavior to improve cache performance. The future reuse is predicted based on past behavior and thus may not necessarily be accurate. In addition, such policies do not define an accurate cache performance model based on the reuse information and thus cannot achieve their full performance potential. This paper proposes a way to address these issues for the above-mentioned cache management policies and focuses on the LLC.

Let us start with the replacement policy. Many such policies have been proposed for the LLC [1, 6, 14, 18, 19, 20, 24, 31] aiming to improve over the LRU replacement policy, the most widely used replacement policy, which has been shown [33, 29, 24] to have anomalous behavior for applications whose working set is larger than the LLC. The newer policies are often adaptive and use heuristics based on predicted future memory reference behavior for cache line replacement. A widely used estimate of the future behavior is the observed reuse distance.

Many ways of reuse distance prediction and its use by replacement heuristics have been proposed. For instance, EELRU [29] accurately measures the reuse (stack) distance but uses a probabilistic model to predict which cache line should be evicted on a cache miss. RRIP [14] approximates the reuse distances as near, long or distant future and tries not to replace lines which are predicted to be reused sooner. IGDR [31] builds an accurate reuse distance distribution but replacement is based on a "weight" function, with a line of smallest weight replaced. However, the weight function does not necessarily reflect future behavior of a line. The counter-based algorithm [19] uses a counter matrix to predict when lines are reused. The approach in [17] predicts reuse distances using program counters and evicts lines with the longest remaining distance (more in Sec. 7).

A more precise knowledge of reuse would allow a replacement policy (1) not to evict lines too early, before their next reuse point, and at the same time (2) not to keep the lines in the cache for too long to avoid cache pollution. A better "balance" between (1) and (2) can improve the policy performance. Sec. 2.1 presents a case study of this phenomenon, and the replacement policy proposed in this paper indeed achieves such balance.

The *reuse distance* (RD) used in this paper is defined as the number of accesses to a cache set between two accesses to the same cache line (Sec. 7 compares this definition to others). A *reuse distance distribution* (RDD) is a distribution of RDs observed in a program at a given time. It is a unique signature of a program or a phase for a given cache configuration. Fig. 1 shows the RDDs of several SPEC CPU2006 benchmarks (see Sec. 5 for our measurement methodology). As RDs can be very large, this paper limits the *maximum measured distance* d_{max} to 256. The fraction of RDs below the d_{max} is shown as a bar on the right of each figure.

Using the RDD to direct prediction, lines can be kept only until a desired level of reuse is achieved *and* cache pollution is minimized. For example, in Fig. 1, in 436.cactusADM enough lines are reused at or below the RD of 64, increasing the RD beyond 64 leads to cache pollution. The problem, however, is how to find an RD balancing reuse vs. pollution and how to use it to manage replacement. This paper proposes a way to solve this problem.

Similar to prior work [19, 31], this paper measures the RDs in execution and builds the RDD dynamically. The RDD is used to

1072-4451/12 $26.00 © 2012 IEEE
DOI 10.1109/MICRO.2012.43

IEEE computer society

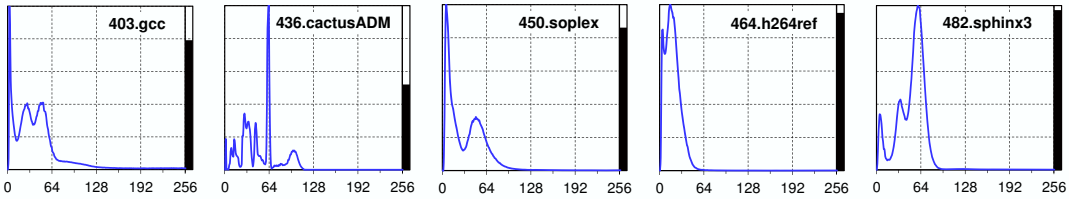


Figure 1. Distribution of reuse distances for selected benchmarks.

compute an RD that maximizes the cache hit rate, i.e., the RD that balances the opportunity for reuse with timely eviction. This RD is called a *Protecting Distance* (PD) and our replacement algorithm "protects" a line for this many accesses to its set before it can be evicted. This policy is thus called the *Protecting Distance based Policy* (PDP).

It can be said that the LRU policy protects a line for W unique accesses before eviction, where W is the cache associativity. However, W may not be the RD that maximizes the hit rate. PDP achieves the same effect for protecting distances PD $\leq W$, i.e. in LRU-friendly benchmarks. But the PDP also works for PDs $> W$ using the same algorithm. Thus PDP does not need to switch between LRU and another algorithm as has been done in prior work [29, 24, 14, 17]. Note also that EELRU [29] can be said to "protect" the late eviction point by evicting any "older" lines first, if present (see Sec. 7).

The RDD can also be used to manage a cache bypass policy in non-inclusive or exclusive caches. Typically, the bypass is used to place a line in an upper level cache and avoid duplicated use of storage in the lower level cache. However, it may be better to store a line with a lot of reuses in the lower level. A bypass policy that makes such a choice intelligently can, in fact, have a better performance than a strictly inclusive cache. The PDP can be combined with the bypass policy to enhance the replacement policy by protecting cache lines from early eviction. The non-inclusive and exclusive caches are already used in commercial products, such as the AMD processors [8], and several policies to support bypass have been proposed [6, 13, 18, 19, 5]. The work in [10] proposed a policy to achieve performance similar to bypass in an inclusive cache, and the algorithms in [6] are developed specifically for exclusive caches.

Finally, this paper describes a PD-based shared cache partitioning policy that is based on the measured RDDs of different threads and a selection of PDs for the threads such that the shared cache hit rate is maximized.

The paper discusses the hardware support required to measure the reuse distance, compute the RDD dynamically, determine the hit rate and find the PD. The hardware was synthesized to evaluate the overheads and is shown to be feasible.

In summary, the main contributions of this paper are:

1. The concept of protecting distance PD and a policy for protecting cache lines, which is adaptive to changes in program phases and memory access behavior.
2. A hit rate model for a single-core processor as a function of reuse distances, which is shown to accurately predict the cache behavior of an application.
3. A method and a hardware design to compute the PD that maximizes the hit rate.
4. A bypass policy that guarantees a desired level of reuse in the LLC and is integrated with the replacement policy.
5. A multi-core shared LLC hit rate model that is a function of PDs of individual threads.
6. A shared cache partitioning policy based on reuse distances, which computes a set of PDs, one per thread, to maximizes the overall hit rate.

The proposed policies were evaluated and compared to a number of prior policies. For single-core replacement PDP was compared to DIP, DRRIP, EELRU and sampling dead-block prediction (SDP). Shared cache partitioning for multi-core processors was compared to TA-DRRIP, UCP and PIPP.

The rest of the paper is organized as follows. The PD concept is defined in Sec. 2 together with a hit rate model to quantitatively relate the PD and the RDD. The hardware design is in Sec. 3. The hit rate model for multi-core cache partitioning is defined in Sec. 4. The experimental methodology is in Sec. 5. The experimental results are in Sec. 6. Prior work is discussed in Sec. 7. Appendix A reports results for SPEC CPU2006 benchmarks.

2 SINGLE-CORE PROTECTING DISTANCE

This section defines the protecting distance PD for a single-core processor. A replacement policy using the PD and a combined replacement and bypass policy are presented. A hit rate model for the policy is also described.

2.1 Protection – A Case Study

Let us start with a case study using RRIP policy [14], which was recently proposed to improve the performance of applications with a working set which is larger than the LLC. RRIP is able to preserve a portion of the working set in the cache. It tackles the thrashing and scanning issues by predicting that most of the inserted lines are re-referenced (reused) in a *distant* future, the rest inserted lines in a *long* future, and the reused lines in a *near* future.

Now consider RRIP from the perspective of *protection*. The re-reference future of lines is predicted using a parameter, ϵ. This is the probability that inserted lines are predicted to be reused in a long future. In RRIP, ϵ is experimentally set to 1/32. This means that an inserted line which is classified as being reused in the long future can be protected for 32 misses. In RRIP with 2-bit storage per line, on average a reused line can be protected for as long as $32 \times (2^2 - 1) = 96$ misses (and an even larger number of accesses)[1].

Fig. 2 shows the behavior of dynamic RRIP (DRRIP) as a function of ϵ for four benchmarks (483.xalancbmk.3 is a certain window of execution, see Sec. 5 for details). ϵ is varied from 1/128 to 1/4 and two trends can be observed. For 436.cactusADM and 483.xalancbmk.3 decreasing ϵ increases the number of misses. However, for 403.gcc and 464.h264ref a higher ϵ can reduce misses. This means that, in the latter two benchmarks, lines should not be

[1] These numbers vary during runtime depending on access behavior.

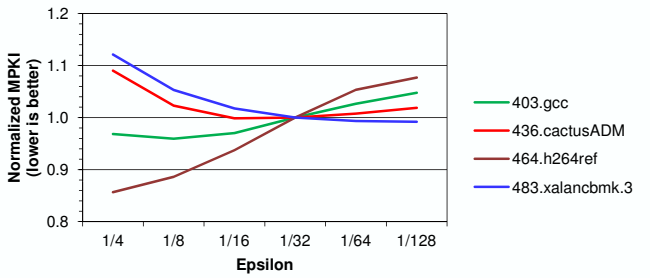


Figure 2. DRRIP misses as a function of Epsilon ϵ.

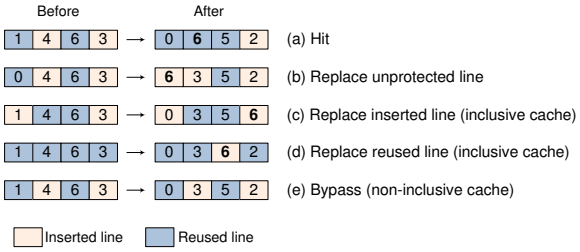


Figure 3. Example of the PDP cache operation.

protected too long if they are not reused. Yielding cache space to new lines improves performance. Sec. 2.3 will analyze these benchmarks in more detail.

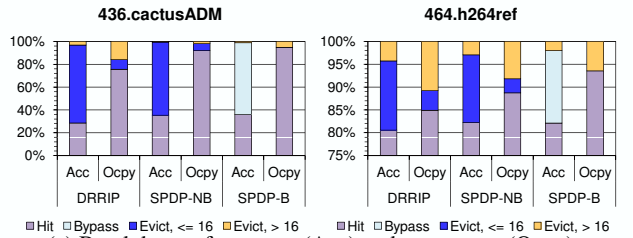
2.2 The Protecting Distance

The *protecting distance* PD is a reuse distance that "covers" a majority of lines in the cache, that is they are reused at this or smaller distance. It is a single value used for all lines inserted in the cache. The PD is the number of accesses to a cache set that a line is protected for, i.e. the line cannot be evicted until after PD accesses. The PD is used as follows. Each cache line is assigned a value to represent its *remaining PD* (RPD), which is the number of accesses it remains protected for. This distance is set to the PD when a line is inserted or promoted. After each access to a set, the RPD of each line in the set is decremented (saturating at 0). A line is protected only if its RPD is larger than 0. An unprotected line is chosen as the victim. Victim selection when there are no unprotected lines is different for inclusive and non-inclusive caches.

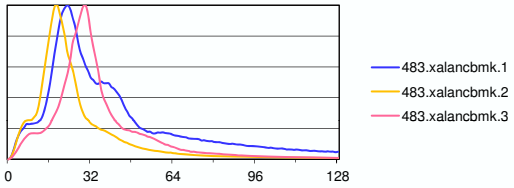
In an inclusive cache a line with the highest RPD, i.e. the youngest inserted or promoted line, is chosen in order to protect older lines. Similar to DIP and RRIP, the fact that inserted lines are less likely to be hit than reused lines is utilized. An inserted line with the highest RPD is replaced or, if there are no such lines, a reused line with the highest RPD is chosen. This requires a "reuse bit" to distinguish between inserted and reused lines.

For a non-inclusive cache, the cache is bypassed if no unprotected lines are found. This is based on the prediction that the protected lines are more likely to be reused in a the nearer future than the missed lines. Note that the bypassed lines are inserted in a higher-level cache. Here the bypass policy and the replacement policy work together to further protect lines as opposed to only the replacement policy in the inclusive cache. The reuse bit is not required here.

The example in Fig. 3 illustrates the PDP policy in a cache set with four lines. Let us assume that the current predicted PD is 7. In the figure, the number inside each box is the RPD of the corresponding cache line. There are 5 possible scenarios in this example.



(a) Breakdown of accesses (Acc) and occupancy (Ocpy).



(b) RDDs in 3 windows of 483.xalancbmk.

Figure 5. Case studies.

For each scenario, RPDs before and after the access are shown. In Fig. 3a, the access results in a hit to the second line. In the other 4 cases, the access results in a miss leading to victim selection. In Fig. 3b, the victim is the unprotected line. Fig. 3c and 3d are for a cache without bypass. In Fig. 3c, the inserted line with the highest RPD is evicted, while in Fig. 3d there are no inserted lines and the reused line with highest RPD is evicted. Fig. 3e is for the cache with bypass, where the cache is bypassed as there are no unprotected lines. In all cases, the RPD of the promoted or inserted line is set to 7, and the RPDs of all lines, including the new line, are decremented.

2.3 Evaluation of the Static PDP

The static PDP (SPDP) uses a constant PD throughout program execution. The SPEC CPU2006 benchmarks were simulated with static PDs between 16 (the associativity) and $d_{max} = 256$. The PD which minimizes misses per 1K instructions (MPKI) varies from benchmark to benchmark. Even in the three simulation windows of 483.xalancbmk, the best PDs are different (see Appendix A for details). Fig. 4 compares the static PDP with the best PD for non-bypass (SPDP-NB) and bypass (SPDP-B) policies to DRRIP with the best ϵ (as described in Sec. 2.1).

First, better DRRIP performance can be achieved with a "dynamic" ϵ. It is significant for 403.gcc, 450.soplex and 464.h264ref. Second, both SPDP-NB and SPDP-B can further reduce misses over DRRIP, by as much as 20% for SPDP-NB and by 30% in SPDP-B in 464.h264ref. Third, SPDP-NB and SPDP-B have different behaviors in different benchmarks. For example, in 436.cactusADM the miss reduction is similar, whereas for 483.xalancbmk.3 SPDP-B has a significantly higher reduction than SPDP-NB. In general, SPDP-B achieves a higher miss reduction than SPDP-NB and both perform better than DRRIP.

The PDP aims to protect cache lines long enough to be reused, but not for too long to avoid pollution. Let us define the *occupancy* of a line as the number of accesses to its cache set between an insertion or a promotion and the eviction or the next promotion and analyze accesses and occupancy for two benchmarks, 436.cactusADM and 464.h264ref. The accesses and occupancy are shown in Fig. 5a, each broken into promoted lines and evicted lines. The latter is further divided into lines which are evicted before 16 accesses and all the rest The fraction of bypassing for SPDP-B is also shown. The

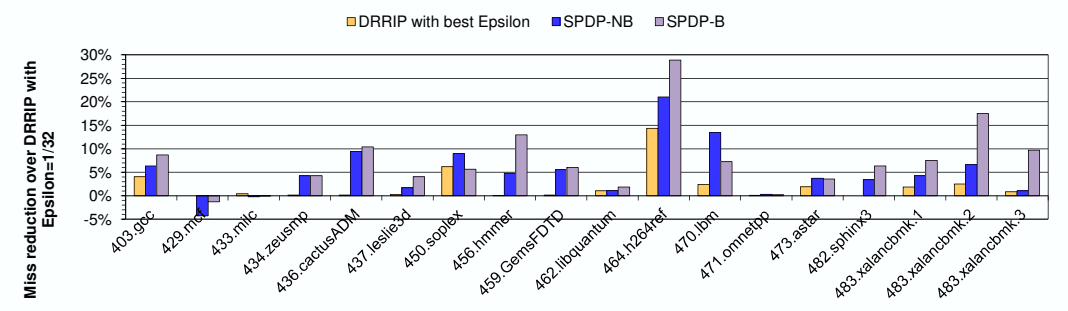


Figure 4. Comparing RRIP and static PDP.

RDDs of three execution windows for 483.xalancbmk are shown in (Fig. 5b).

436.cactusADM. 28% of the accesses in DRRIP are hits and the rest are misses, leading to replacement. Most of the missed lines are evicted early, before 16 accesses, with only 3% evicted after that. However, the total occupancy of the latter lines is as high as 16%, whereas for early evicted lines, it is only 8%. In fact, our results show that a number of lines occupy the cache for more than 256 accesses without reuse.

The occupancy of evicted lines is smaller for both PDP policies, 8% for SPDP-NB and 5% for SPDP-B, and much smaller for long eviction distance lines. Neither has lines with occupancy above 90 accesses. Also, the small difference between the occupancy of evicted lines in SPDP-NB and SPDP-B leads to the small difference between their performance – SPDP-B reduces 1% more misses than SPDP-NB compared to DRRIP (Fig. 4). The values of PD are 76 (SPDP-NB) and 72 (SPDP-B), respectively, which cover the highest peak of the RDD.

464.h264ref. Both static PDP policies result in a smaller occupancy fraction for lines evicted after more than 16 accesses. SPDP-B is better than SPDP-NB due to a high bypass rate (89% of the misses), hence protecting more lines and leading to higher miss reduction. This shows that the bypass plays an important role in this and several other benchmarks.

483.xalancbmk. Fig. 4 shows that the three windows of execution within the same benchmark have different performance for both static PDP policies. The best PD for each window are also different in SPDP-B – 100, 88 and 124, respectively (see Appendix A). This difference is due to different RDDs of these windows, as seen in Fig. 5b. The peaks are at different reuse distances and even the shapes of RDDs are not quite the same. This implies that a periodic dynamic computation of the PD would perform even better.

In summary, replacement based on protecting distance reduces cache pollution. A combination of replacement and bypass further improves reuse and reduces pollution. The rest of the paper thus targets non-inclusive caches with bypass.

2.4 The Hit Rate Model for a Non-Inclusive Cache

The definition of the protecting distance PD is not very helpful in finding the optimal PD, static or dynamic. This section develops a quantitative model relating the RDD, the PD and the cache hit rate for a non-inclusive cache. The RDD or its approximation can be built dynamically and the model then used to find the PD maximizing the hit rate. The model takes into account both replacement and bypass policies.

The following notation is used in this section: d_p is a value of the PD, N_i is the hit count for reuse distance i, and N_t is the total number of cache accesses. W is the set associativity. A line with an RD larger than d_p is called a *long line* and N_L is the number of *long lines*.

A function $E(d_p)$ approximating the hit rate is derived as following. Given a set of counters $\{N_i\}$ representing the RDD and N_t, $N_L = N_t - \sum_{i=1}^{d_p} N_i$.

Let us use the concept of occupancy defined in Sec. 2.3. A line with an RD of i, $i \leq d_p$, has an occupancy of i and the line is hit after i accesses. The total occupancy L_i of all lines with RD of i in the cache is $L_i = N_i * i$.

The total occupancy of the long lines is $L_L = N_L * (d_p + d_e)$. The additional term d_e accounts for the fact that a line may not be immediately evicted after becoming unprotected. This may happen due to the presence of an invalid or another unprotected line during victim selection or when an access to a set is a hit on another line. The total occupancy of all lines is therefore $L_T = \sum_{i=1}^{d_p} L_i + L_L$.

The number of hits contributed by all lines is $Hits(d_p) = \sum_{i=1}^{d_p} N_i + H_L$, where H_L is the number of hits contributed by long lines. For $d_p \geq W$ the number of hits from long lines is quite small compared to that from protected lines. And the computed PD is greater than W (see below). Therefore, the following approximation is used $Hits(d_p) \approx \sum_{i=1}^{d_p} N_i$.

One access to a set with W lines increases the occupancy of each line in the set by 1 unit, thus W units for the whole set. The total number of accesses, therefore, is $Accesses(d_p) = L_T/W$ and the hit rate is $HR(d_p) = Hits(d_p)/Accesses(d_p)$.

To eliminate the dependence on cache organization (W) let us define $E(d_p) = HR(d_p)/W$. Here $E(d_p)$ is proportional to the hit rate and will be used to compute the protecting distance PD that maximizes the hit rate.

Substituting the expressions for L_i and L_L in the equation for $E(d_p)$ results in

$$E(d_p) \approx \frac{\sum_{i=1}^{d_p} N_i}{\sum_{i=1}^{d_p} (N_i * i) + \left(N_t - \sum_{i=1}^{d_p} N_i\right) * (d_p + d_e)} \quad (1)$$

Finally, the d_e is a function of cache associativity and program behavior. It has an impact on $E(d_p)$ only when the d_p is small. It

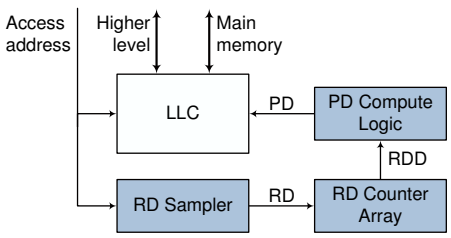


Figure 7. The PDP cache organization.

has been experimentally determined that it can be set to a constant equal to W. This is another reason why $E(d_p)$ is an approximation of the hit rate.

Fig. 6 shows the $E(d_p)$ and the actual hit rate as a function of static $d_p < 256$ for four SPEC CPU2006 benchmarks (Appendix A shows the complete suite). The RDD is also shown. The model can be seen to approximate the actual hit rate well, especially around the PD which maximizes hit rate.

Eq. 1 (with $d_e = W$) is used to find the optimal PD, i.e. the PD achieving the highest hit rate. The search algorithm computes $E(d_p)$ for all $d_p < 256$ and finds the maximum $E()$. Not that this search is only performed periodically and that $E(d_p + 1)$ can be computed from $E(d_p)$ to significantly reduce the search time.

3 PDP CACHE ORGANIZATION

This section describes the cache operation and the additional hardware required for dynamic computation of the optimal PD. Fig. 7 shows a block diagram of the PDP cache. The additional blocks include an RD sampler to measure the reuse distances (RDs), an array of RD counters which track the number of hits at each reuse distance (i.e. the RDD), and logic to find the PD using Eq. 1.

The RD sampler. The RD sampler monitors access to a small number of cache sets to collect the reuse distances observed. It uses a per-set FIFO of addresses which accessed the set. A new access to a sampled set has its address compared the set's FIFO entries. The FIFO position of the most recent match, if any, is the reuse distance RD. Its corresponding reuse counter in the counter array is incremented. An RD as large as d_{max} needs to be detected.

It has been shown in [24, 14] that sampling just 1/64 of the total sets is sufficient to capture the behavior of an application. A reduced FIFO insertion rate may be used to allow smaller FIFO size (note that cache tag check is still performed for each access). In this case a new entry is only inserted on every M^{th} access to the set, with a *sampling counter* counting up to M. The reuse distance RD is now computed as $RD = n \times M + t$, where n is the FIFO position of the hit, t is the value of of the sampling counter on the sampler hit. An entry is marked invalid on such a hit to reduce error in RD measurement. Similar to the work in [18], a FIFO entry uses 16 bits to store a tag. The total number of bits per sampled set is $\frac{d_{max}}{M} * 16 + \log_2 M$.

The array of RD counters. The RD counter array stores the RDD $\{N_i\}$. The i^{th} counter is the number of accesses with the RD of i. An RD arrives from the RD sampler and the corresponding counter is incremented. An additional counter is used for the total number of accesses N_t. These are saturating counters and, when a counter saturates, all other counters are frozen to maintain the RDD shape.

A space optimization for the array is to store hit counts for a consecutive range of RDs in one counter. The range extent is called a *step counter*, (S_c). For instance, if $S_c = 4$ then the first counter is incremented for RDs from 1 to 4, the next one for the RDs from 5 to 8, and so on. The number of counters can thus be reduced to $\frac{d_{max}}{S_c}$. This also reduces the computation time to select the protecting distance PD. However, this also means that the PD belongs to a range rather than being a single value. The proposed PDP implementation uses 16-bit N_i counters and a 32-bit N_t counter. Total number of bits required for the counter array is $\frac{d_{max}}{S_c} \times 16 + 32$.

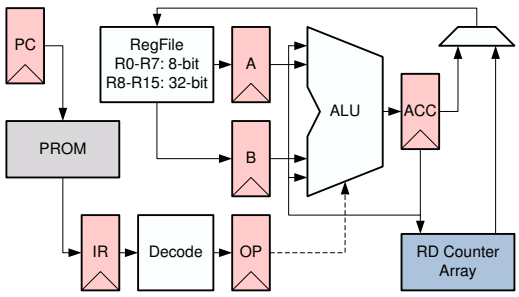


Figure 8. A "PD compute logic" special-purpose processor.

Logic to compute the PD. The PD is recomputed infrequently and thus its computation is not time critical. The logic to find the optimal PD can thus be implement it as a special-purpose "processor". A possible 4-stage pipelined architecture is shown in Fig. 8. It uses a 32-bit ALU, eight 32-bit registers and eight 8-bit wide. The processor uses the RD counter array as input and outputs the optimal PD. It executes one algorithm using sixteen integer instructions: add/sub, logical, move, branch, mult8 and div32. The mult8 multiplies a 32-bit register by an 8-bit register and is performed using shift-add. The div32 is a division of two 32-bit numbers and is performed as shift subtract/add non-restoring division (33 cycles). The processor takes 64 cycles to compute $E(d_p)$ for one d_p in Eq. 1. Thus the total time to compute the optimal PD, 64×256 cycles, is negligible compared to the number of cycles between two PD computations (512K accesses to the LLC in this paper). The processor can be put in low-power sleep mode when not in use. The processor was synthesized using a 65nm technology library and required 10K NAND gates operating at 500MHz clock frequency.

Cache tag overhead. The computed PD is used by the cache to set the initial value of remaining PD, RPD (see Sec. 2.2 for the definition), for inserted or promoted lines. Storing the RPD that can range up to $d_{max} = 256$ requires $n_c = 8$ bits per line. This overhead can be reduced by using an increment, the *Distance Step* S_d. An update of RPDs is done once every S_d accesses to the set using a per-set counter (and counting bypasses). An access causing the counter to reach $S_d - 1$ triggers the decrement of all RPDs in the set. This does not have to be done in parallel but can be done sequentially to reduce complexity. The maximum value of S_d is $\frac{d_{max}}{2^{n_c}}$.

The PDP parameters. A number of hardware parameters are used in the PDP: (1) the RD sampler size, (2) the S_c counter used in the counter array, (3) the number of bits to store the PD n_c, and (4) the maximum distance d_{max} (Sec. 1). The choice of these parameter values impacts both the hardware overhead and the cache performance. Sec. 6.1 presents a design space exploration of these parameters.

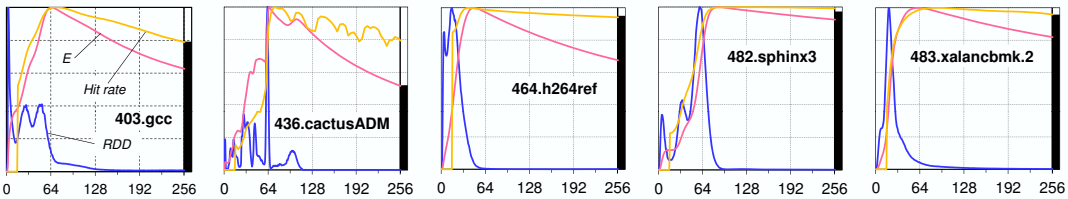


Figure 6. $E(d_p)$ vs the actual hit rate.

4 A PD-BASED CACHE PARTITIONING POLICY

All high-performance multi-core processors today use a shared LLC. The shared LLC can benefit significantly from (cooperative) cache partitioning. This section defines a new multi-core partitioning policy based on the concept of protecting distance. The key insight is that decreasing a thread's PD leads to a faster replacement of its lines and thus shrinks its partition. Increasing the PD has the opposite effect. The proposed policy defines a per-thread protecting distance and computes a set of PDs that partitions the cache to maximize hit rate. The computation is based on a new shared LLC hit rate model that is a function of a set of PDs.

The numerator and denominator in Eq. (1) are $H(d_p)$, the number of hits, and $A(d_p)/W$, the number of accesses. They need to be computed per thread when multiple threads access the shared cache. A thread t has $H_t(d_p^{(t)})$ hits and $A_t(d_p^{(t)})/W$ accesses for a protecting distance of $d_p^{(t)}$. The total number of hits and accesses for T threads are $Hits(d_p^{(t)}) = \sum_{t=0}^{T-1} H_t\left(d_p^{(t)}\right)$ and $Accesses(d_p^{(t)}) = \sum_{t=0}^{T-1} A_t\left(d_p^{(t)}\right)/W$, respectively. The multi-core function E_m is $E_m = Hits/(Accesses * W)$. Using a vector $\overrightarrow{d_p} = \left[d_p^{(0)}, ..., d_p^{(T-1)}\right]$ to denote an ordered set of $d_p^{(t)}$, the multi-core hit rate approximation as a function of $\overrightarrow{d_p}$ is:

$$E_m\left(\overrightarrow{d_p}\right) = \frac{\sum_{t=0}^{T-1} H_t\left(d_p^{(t)}\right)}{\sum_{t=0}^{T-1} A_t\left(d_p^{(t)}\right)} \quad (2)$$

A vector $\overrightarrow{PD} = \left[PD^{(0)}, ..., PD^{(T-1)}\right]$ that maximizes $E_m(\overrightarrow{PD})$ defines the protecting distances $PD^{(t)}$ of each thread.

A heuristic is used to find the vector $\overrightarrow{PD}$ instead of an an exhaustive search. It is based on three observations. First, a thread with a high single-core E will also make a high contribution to the multi-core E_m. Second, a computed multi-core PD for a given thread is likely to be near one of the "peaks" in its single-core E. And third, the number of "important" peaks in an application is quite small. The heuristic thus builds $\overrightarrow{PD}$ by adding a thread at a time and searches for the added thread's PD near one of its peaks only.

The multi-core partitioning policy requires a counter array per thread but still uses the single-core PD computing logic to generate the "high peaks" for each thread . The heuristic first sorts the threads by their E's, then adds the thread with the "highest E" to the vector. The next highest thread is processed next, each of its peaks (computed using the single-core E) is considered in combination with peaks of the thread already in the vector. A search algorithm is used to find the peak of the combination that maximizes E_m. The process is repeated for each remaining thread.

It has been experimentally determined that considering just three peaks per thread is sufficient. For each thread combination, the search algorithm has the complexity of $O\left(T \times S\right)$, where S is the number of single-thread E re-computations. Given that the number of combinations is a linear function of T, the complexity of E_m is $O\left(T^2 \times S\right)$. The complexity is similar to that of UCP [25]. The processor described in the previous section can be easily extended to execute this heuristic. The latency of the PD vector search is still negligible compared to the PD recomputation interval.

Pipeline Depth	8
Processor Width	4
Instruction Window	128
DCache	32KB, 8-way, 64B, 2 cycles
ICache	32KB, 4-way, 64B, 2 cycles
L2Cache	256KB, 8-way, 64B, 10 cycles
L3Cache (LLC)	2MB, 16-way, 64B, 30 cycles
Memory latency	200 cycles

Table 1. The single-core processor.

5 EXPERIMENTAL METHODOLOGY

The proposed single- and multi-core management policies at the LLC were evaluated. A memory hierarchy with three levels of cache is modeled which is similar to the Intel Nehalem processor. An out-of-order core is modeled with the CMP$im [11] simulator. The core and the memory hierarchy parameters are described in Table 1. The L1 and L2 caches use the LRU policy. A multi-core processor with a shared LLC is also modeled with CMP$im. The shared LLC size is the single-core LLC size times p, where p is the number of cores.

The following SPEC CPU2006 benchmarks were used for the single-core evaluation: 403.gcc, 429.mcf, 433.milc, 434.zeusmp, 436.cactusADM, 437.leslie3d, 450.soplex, 456.hmmer, 459.GemsFDTD, 462.libquantum, 464.h264ref, 470.lbm, 471.omnetpp, 473.astar, 482.sphinx3 and 483.xalancbmk. Other benchmarks in the suite were not considered because they do not stress the LLC, e.g. their MPKI is less than 1 for the baseline DIP policy. A window of 1 billion consecutive instructions per individual benchmark was simulated. To see how each policy reacts to phase change within an application, three different 1B instruction windows were studied for 483.xalancbmk to observe the phase changes, with results denoted as 483.xalancbmk.X, where X is the window number. A subset of benchmarks which demonstrate significant phase changes were simulated with a window of 10 billion instructions.

Multi-core, shared LLC simulations used workloads generated

by combining the individual benchmarks described above. Benchmark duplication was allowed in a workload. 80 workloads were generated using random selection for the 4-core and 16-core configurations. A workload completes execution when each of its threads completes the execution of its 1B instructions. A thread completing before other threads "rewinds" and continues its execution from the first instruction of its window. Per-thread statistics are collected when a thread finishes its first one billion instructions.

Single-core performance metrics are misses per thousand instructions (MPKI) and the IPC. Multi-core performance metrics are the *weighted IPC* ($W = \sum \frac{IPC_i}{IPC_{Singlei}}$), the *throughput* ($T = \sum IPC_i$), and the *harmonic mean of normalized IPC* ($H = N/\sum \frac{IPC_{Singlei}}{IPC_i}$). IPC_i above is the IPC of a thread i in multi-core, multi-programmed execution for a given replacement policy. $IPC_{Singlei}$ is the IPC of the thread i executed stand-alone on the multi-core and using the LRU policy. LRU is used as the baseline here for consistentency with prior work [25]. The results for each metric for a given replacement policy are are shown normalized to the shared cache DIP.

The single-core PDP is compared with DIP [24], DRRIP [14], a variant of EELRU [29] and the sampling dead block predictor (SDP) [18]. DIP and DRRIP used the dynamic policy. The $\epsilon = 1/32$ was used for BIP and BRRIP except when evaluating the impact of ϵ. An SDM with 32 sets and a 10-bit PSEL counter was used for DIP and DRRIP. Writebacks were excluded in updating PSEL counters in these policies. The hardware overhead for SDP was made 3 times as large as that reported in the original work in order to maximize the performance.

Each cache set for EELRU evaluation was augmented with a recency queue to measure the number of hits at a stack position. Two global counter arrays were used to count the number early hits and total hits for each pair of early eviction point *e* and late eviction point *l* over all sets. The parameters *e* and *l* were chosen aggressively to make sure that EELRU achieves its best possible performance. The maximum value of *l* is set to $d_{max} = 256$ to be compatible with PDP.

Three thread-aware shared cache partitioning policies were compared with PDP: the UCP [25], the PIPP [36] and the TA-DRRIP [14]. The lookahead algorithm was used to compute the ways for UCP and PIPP. Thirty two sampling sets were used for UCP and PIPP. $p_{prom} = \frac{3}{4}$, $p_{stream} = \frac{1}{128}$, $\theta_m \geq 4095$, and $\theta_{mr} = \frac{1}{8}$ were used for PIPP, per original work. The implementations of DIP, RRIP and TA-DRRIP were verified against the source code from the author's website.

6 EVALUATION AND ANALYSIS

The performance of PDP depends on a number of parameters. Thus we start with a parameter space design exploration for the single-core PDP. The parameters can be chosen to balance overhead and performance. Once most of the parameters are selected, this section presents and compares results for the single-core replacement and bypass policies, as well as the PDP shared cache partitioning policy.

6.1 A PDP Parameter Space Exploration

The performance and hardware overhead of the PDP are a function of the maximum PD allowed, sampler and FIFO sizes, and the number of bits to store the remaining PD per line. Fig. 9 shows the effect of two PDP parameters: the RD sampler size and the counter step S_c. The *Full* configuration uses a FIFO per LLC line. The *Real* configuration uses a 32-entry RD sampler (32 FIFOs, each with 32 entries). The impact of the counter step S_c is shown for the *Real* configuration. The specific PDs of each configuration can be found in Appendix A.

Range of PD	16-32	33-64	65-128	129-256
# of benchmarks	4	5	4	3

Table 2. The PD distribution of SPEC CPU2006 benchmarks.

The results show that the RDDs obtained using the 32-entry RD sampler are basically identical to those obtained without sampling. An even smaller sampler size can be used, but it will take longer to warm up and to build a good-quality RDD. Therefore, the rest of the paper uses the *Real* configuration of the RD sampler.

Varying the counter step S_c from 1 to 8, the $S_c = 2$ has mostly no difference with the $S_c = 1$. Two benchmarks, 456.hmmer and 470.lbm, show a noticeable change for higher values of S_c. This is due to the rounding errors in the PD computation. The $S_c = 4$ is selected for the rest of the paper in a trade-off between performance and hardware overhead,

A third parameter, the maximum protecting distance d_{max}, is chosen based on results in Table 2. The table shows the distribution of optimal PD for the sixteen SPEC CPU2006 benchmarks used and the *Full* sampler configuration. None of the benchmarks has a PD larger than 256, even if d_{max} is set to be larger. Therefore the $d_{max} = 256$ is used in the rest of the paper. The table also shows that a smaller d_{max} can also be used, but with some impact on performance. For example, the $d_{max} = 128$ results in lower performance for three benchmarks.

A fourth parameter, the number of bits per cache tag, is evaluated in the next section.

6.2 The Single-core PDP

Fig. 10 shows the performance of PDP and three prior policies: DRRIP [14], EELRU and SDP [18]. The static SPDP-B is also shown. All the results are normalized to the DIP [24] pollcy. The evaluated metrics are the miss reduction, IPC improvement, and the fraction of accesses bypassed. Three different PDP configurations are shown varying the number of extra bits per cache tag n_c (e.g, PDP-3 has $n_c = 3$).

First, let us compare prior replacement policies. DRRIP has several benchmarks which improve over DIP, significantly in the 450.soplex, 456.hmmer, and 483.xalancbmk.3. The benchmark 464.h264ref shows degradation over DIP. This benchmark was analyzed in Sec. 2.1. In fact, DRRIP achieves similar performance to DIP with $\epsilon = 1/4$ for 464.h264ref. On average, DRRIP reduces the misses by 1.8%, leading to a 1.5% improvement in IPC over DIP. This is consistent with results for the SPEC CPU2006 suite reported in the DRRIP work [14]. The EELRU is shown to have a significant degradation compared to DIP in several benchmarks. This is due to the fact that a majority of cache lines are evicted before reaching their reuse point, hence they pollute the cache without contributing to hits. In fact, it was previously reported that DIP outperforms EELRU [24].

Second, let us compare the miss reduction for the static PDP (SPDP-B) and the dynamic PDP (PDP-8). Recall that the dynamic PDP uses the hit rate model to find the PD which maximizes the

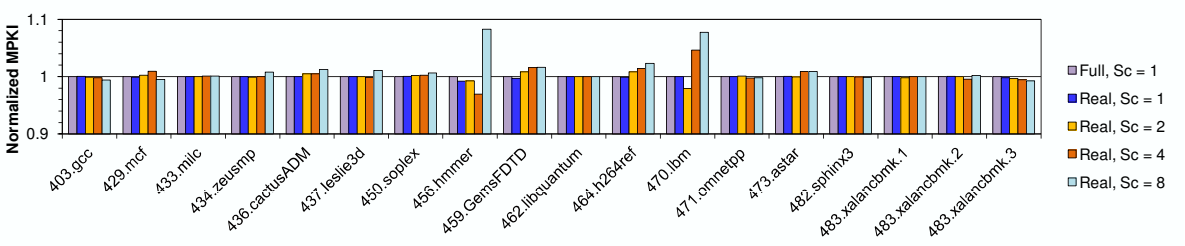


Figure 9. Comparing PDP parameters.

hit rate. The upper bound for short traces (1B instructions) is the static PDP and the results show that in most cases the dynamic PDP is close to this bound. However, there is a significant difference in the case of 429.mcf and 456.hmmer. Appendix A also shows that these two benchmarks have the best static PD which is quite different from the computed PD. On average, SPDP-B eliminates 1.4% more misses than PDP-8.

Third, consider the impact of n_c. PDP-8 has a 1.5% and 0.5% higher miss reduction, respectively, compared to PDP-2 and PDP-3. This results in a 2.0% and 0.5% higher improvement in IPC, respectively. But PDP-2 and PDP-3 have significantly less overhead. Thus the n_c parameter can be used to reduce the overhead with a small performance impact. But for 436.cactusADM PDP-2 leads to a significant performance reduction due to the approximation errors eviction of many lines before reuse. Sometimes though, benchmarks get a improvement with smaller n_c, such as 429.mcf and 473.astar, although the impact is less significant for the latter. 462.libquantum has a big performance loss for $n_c < 8$, because its PD is 256, equal to the chosen d_{max}, and all lines are evicted before this PD. A larger d_{max} will be able to avoid this loss.

Note that the three different execution windows of 483.xalancbmk have different performance. The second window sees a miss reduction of up to 21%, with a 21% improvement in IPC (over DIP) which is higher than in the other two windows. The results from only one window, with medium improvement, are used in computing all the averages, for a fair comparison among policies.

Overall, PDP significantly improves performance over DIP and DRRIP in several benchmarks. 436.cactusADM, 450.soplex, 482.sphinx3 and 483.xalancbmk have more than a 10% improvement over DIP. Other benchmarks do not have a significant improvement. In some the LRU replacement works fine (LRU-friendly), such as 473.astar. Others are streaming benchmarks with very large data sets, such as 433.milc, 459.GemsFDTD, 470.lbm. The average IPC improvement over DIP for PDP-2 and PDP-3 is 2.9% and 4.2%, respectively, while the DRRIP improvement over DIP is 1.5%.

Next, let us compare PDP with SDP, a bypass policy using the program counter based dead block prediction. SDP is able to identify and bypass many "dead-on-arrival" lines, but it does not always have a good prediction. This happens for 464.h264ref and 483.xalancbmk, where SDP loses significantly to DIP. Note that for 483.xalancbmk SDP still improves over LRU, the baseline policy it builds upon. A look at the RDDs shows that these benchmarks do not have many lines whose RDs are large, the target of the SDP predictor. This explains the difference in performance for SDP and PDP. Benchmarks where SDP is significantly better than all other policies are 437.leslie3d and 459.GemsFDTD where the use of the PC-based prediction is shown to be advantageous. On average, SDP improves the IPC of 1.6% over DIP.

Bypassing a cache reduces its active power dissipation, an important issue for large LLCs, by not writing the data into the LLC. Fig. 10c shows the fraction of bypassed lines normalized to the number of LLC accesses. Benchmarks which have high miss rates also have high bypass rates. For nearly 40% of accesses the LLC does not have to be written.

The results in this section show that PDP-2 or PDP-3 proved a good balance of overhead and performance. The hardware overhead of the RD sampler, the array of RD counters, the processor to compute the PD and per-line bits was estimated for a 2MB LLC using the PDP-2 and PDP-3 policies. Expressed in terms of SRAM bits, the overhead is 0.6% for PDP-2 and 0.8% for PDP-3 of the total LLC size. The overheads for DRRIP and DIP are 0.4% and 0.8%, respectively.

6.3 A Direction to Improve PDP

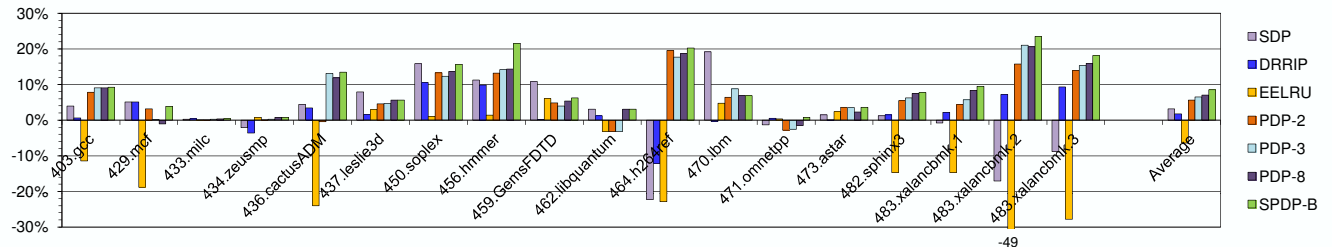
The analysis above gives some hints to further ways to improve PDP. For example, 429.mcf has a higher performance with smaller n_c while SDP is better than other policies in 437.leslie3d and 459.GemsFDTD. The common cause in all cases is the inability to determine how long a line should be protected. For 437.leslie3d and 459.GemsFDTD this can be solved by using a PC-based predictor. A variant of PDP was used to better understand the case of 429.mcf. The variant inserts missed lines with the PD = 1 (mostly unprotected) instead of the computed PD and this result in an 8% miss reduction over DIP. The reduction for SPDP-B is 3.9% and 5.1% for DRRIP. This means that when the inserted lines are removed faster a higher miss reduction is achieved for this benchmark.

The above suggests that the PDP can be improved by grouping lines into different classes, each with its own PD, and where most of the lines are reused. The lines in a class are protected until its PD only, thus they are not overprotected if they are not reused. In fact, prior approaches have classified and treated cache lines differently. A popular way is using the program counters [18, 22, 9]. Another way is to use a counter matrix to group lines [19]. SHiP [34] proposes to improve RRIP using different ways to group cache lines. However, the hardware overhead needs to be carefully considered.

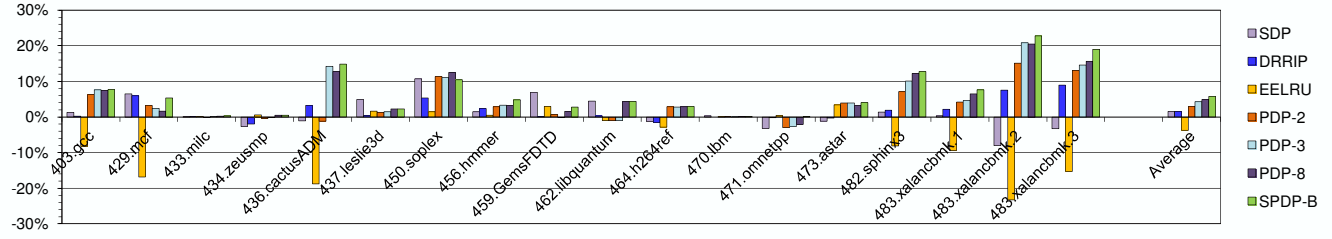
6.4 Adaptation to Program Phases

An application phase change may require the PD to be recomputed. For instance, the three execution windows of 483.xalancbmk have different PDs and different behaviors. to detect phase change The PD needs to be recomputed and the RD counter array reset frequently enough to detect the change. The computed PD is used until the next recompilation while the counters collect the new RDD.

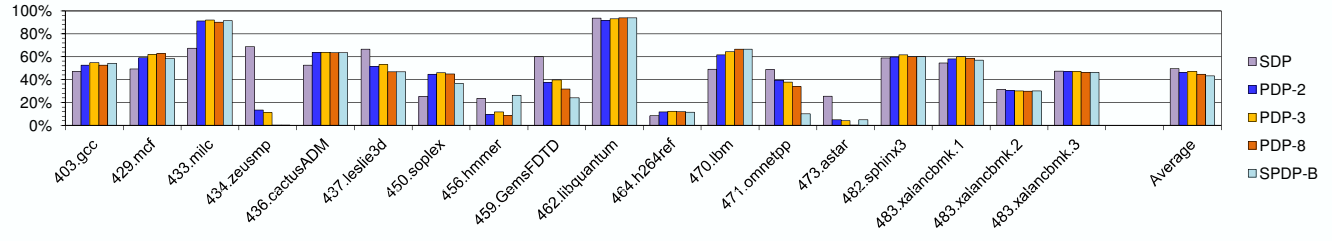
Five SPEC CPU2006 benchmarks have phase changes in an ex-



(a) Miss reduction vs. DIP



(b) IPC improvement vs. DIP



(c) Cache bypass as a fraction of accesses

Figure 10. Performance of replacement and bypass policies (Results for 483.xalancbmk.1 and 483.xalancbmk.2 are excluded from averages)**.**

ecution window of 10B instructions. Fig. 11c shows the PD change over time. Fig. 11a shows the effect of the PD recomputing interval between 1M and 8M accesses, which can be significant. Fig. 11b compares different replacement policies for these benchmarks. PDP is able to adapt to phase changes for these benchmarks.

6.5 The Case for Prefetch-Aware PDP

The PDP policy can be applied in the presence of prefetching. The RDDs in this case are dependent of a specific prefetching mechanism. Note that prefetching techniques often target very long distance access streams, i.e. lines which have very large RDs. Two prefetch-aware variants of the PDP were investigated: (1) prefetched lines are inserted with the PD of 1, and (2) prefetched lines bypass the LLC.

The initial evaluation using a simple stream prefetcher and the modified PDP-8 showed the following. First, the prefetch-unaware PDP had a 3.1% improvement over the prefetch-unaware DRRIP, which is similar to the results without prefetching. The two PDP variants further improve the IPC to 4.1% and 5.6% over prefetch-unaware PDP. Many benchmarks show an improvement of over 20%, including 403.gcc, 450.soplex, 482.sphinx3 and 483.xalancbmk. The improvement is due to the fact that PDP removes prefetched lines fast and even bypasses them, hence they do not pollute the cache. This shows that the PDP can be easily modified to work with prefetching.

6.6 The Cache Partitioning Policy

Fig. 12 shows the performance of the multi-core PD-based partitioning together with other thread-aware policies for 4- and 16-core workloads. Three metrics are reported: the weighted IPC (W), the throughput (T), and the harmonic mean of normalized IPC (H). The PD-based partitioning parameters are the same as for the single-core PDP, except that $S_c = 16$.

The average W, T, and H on four cores are slightly higher for both PDP-2 and PDP-3 compared to the TA-DRRIP and are higher than the UCP and PIPP. The PD-based partitioning is significantly better than other policies for approximately 30 individual workloads, for another 20 it has a negligible improvement, and is not as good as the TA-DRRIP for the rest.

The PD-based partitioning is shown to be more scalable with a larger improvement on the 16-core workloads. Its performance is better than that of all other partitioning policies for more than 70 of the 80 workloads. UCP and PIPP do not scale as well compared to the other policies. On average, PDP-3 has a 5.2%, 6.4% and 9.9% improvement in W, T, H, respectively, over the TA-DRRIP.

7 RELATED WORK

Cache line protection. Let us discuss prior policies from the perspective of line protection, even if they do not use this concept explicitly. LRU can only guarantee that a line is protected for the number of unique accesses equal or smaller than associativity. A line

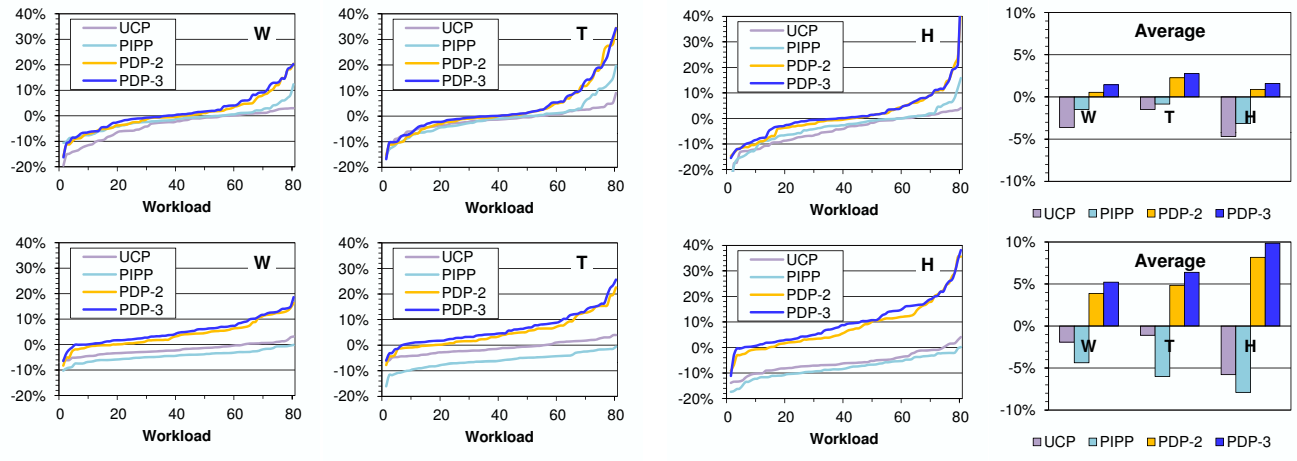
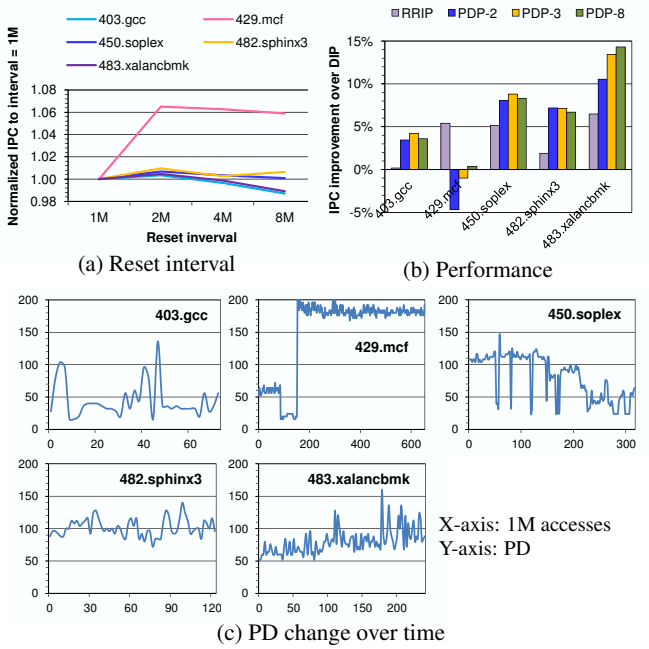


Figure 12. Cache partitioning policies for 4-core (top) and 16-core (bottom) workloads (normalized to TA-DRRIP)**.**



(a) Reset interval

(b) Performance

(c) PD change over time

Figure 11. Adaptation to program phases.

with a reuse distance RD larger than associativity can be protected either by replacing other lines or by explicitly protecting the line. DIP [24] and RRIP [14] predict the re-reference future of lines and protect lines which are predicted to be reused sooner. The counter-based replacement policy [19], using a matrix of counters, protects lines by not evicting them until they expire. This approach does not use an explicit protecting distance, but rather predicts how long a line should be protected by using the past behavior of lines in the same class. The work in [15] measures Inter Reference Recency (IRR) which is similar to reuse distance. Based on this IRR information, cache lines are classified into 2 categories: high IRR and low IRR, and low IRR lines are protected over high IRR lines. The work in [21] uses PC information to identify which lines to protect and the cache is also divided into two regions: MainWays and DeliWays. The use of DeliWays is to protect cache lines longer. Dead block prediction [18] uses program counters (PC) to predict which lines have a very long RD and to bypass them, thus protecting existing lines in the cache.

The approach in [17] used a PC-based reuse distance predictor and monitored the distance remaining till reuse. An LRU line or a line with the largest "distance till reuse" is evicted, whichever is larger. This can be viewed as "protecting" lines with lower distance till reuse. The two issues with this policy are: 1) the PC-based reuse distance prediction is not very accurate and 2) the decision to switch eviction to an LRU line is an approximation without a model to support it.

Shepherd cache [26] emulates the Belady [2] policy to estimate RD and evict lines with a long RD and protect lines with a short RD. The insertion algorithms in [6], which target an exclusive LLC, insert a line with a certain age to determine how long the line should be protected. Similarly, SHiP [34] determines the insertion age of a line by using information about either the memory region or program counters or instruction sequence.

The EELRU [29], or early eviction LRU, was defined for page replacement using the model in [33] and accurately measured the reuse distance. It used LRU for part of the set (physical memory) and switched to early eviction when LRU failed. It can be said to divide the pages into protected and unprotected regions and to compute the region boundaries dynamically. The computation used hit counters and the search process for the best boundary is similar to the search for the best PD. However, unlike the PDP, the EELRU "protection" is managed via eviction only (by evicting all older lines first), i.e. all missed blocks are inserted into the set.

The PDP is different from prior work in that it explicitly estimates how long a line should be protected using the concept of protecting distance. After this explicit protection, if it is not reused, the line becomes unprotected and can be replaced.

Distance measurements. Three ways to measure the reuse distance were defined in prior work. The first one is the stack distance, used by LRU and its variants, with the time unit defined as the number of unique accesses [24, 29, 28]. Stack distance can be used to tell which lines are older, but not its real age. The other two are non-stack: one uses a cache miss as the time unit [21], and the other one uses an access as a unit [22, 17, 4]. The former is dependent

on the replacement policy. Computing stack distance is more complex. Thus we used access-based distance measurement in this paper. Note that the approach in [17] uses global RDs, while PDP uses set-based RDs, which significantly reduces overhead.

Cache bypass. There has been a number of studies using bypass [6, 18, 20, 16, 32, 19, 5]. The dead block prediction work [18] bypasses blocks predicted dead. The work in [32] explores bypass by characterizing data cache behavior for individual load instructions. The approach in [20] uses a hash table to identify never-accessed lines. The approach in [16] uses memory addresses to identify which lines to bypass. Recently, the work in [6] proposed bypass algorithms that use sampling sets to identify dead and live blocks. Our approach bypasses a block when existing blocks are still protected, but without explicitly identifying which lines are likely to be dead. The bypass mechanism in [19] also bypasses lines if there are no expired lines in the set.

Cache sampling. Prior work used sampling on a subset of cache sets [24, 14, 18] and on a fraction of the accesses [18, 22]. Other approaches use feedback to adjust the cache replacement [3] or partitioning [30]. PDP's only contribution here is a low hardware overhead.

RDDs and hit rate model. The IGDR policy [31] used the Inter-Reference Gap Distribution which is similar to the RDD. It used the concept of Inter-Reference Gap from [23]. IGDR used the distribution to compute a "weight" of cache lines and evict the line with a smallest weight. The work in [19] used the distribution implicitly. The approach in [22] used PCs to compute the RD of a line. PDP uses the RDD to explicitly compute a global protecting distance. The PD computation searches for a maximum of E(PD, RDD) and uses the PD achieving the maximum. EELRU [29] used a similar search approach to choose the best set {eviction point, probability}.

Prefetch-aware caching policies. Prior work investigated replacement policy in the presence of prefetching [7, 35, 6]. PACMan [35] is one such approach showing that RRIP can be modified to adapt to prefetching. Our preliminary results also show opportunities to improve PDP in the presence of prefetching.

Shared-cache policies. A number of multi-core, shared cache partitioning policies [25, 36] have been proposed as well as thread aware insertion or eviction [12, 14]. Recently, Vantage mechanism [27] used an analytical model to enhance partitioning in CMP and this model can be used with existing policies. Similar to the work in [12, 14], multi-core PDP does not perform partitioning among cores explicitly, rather it estimates the needs of each thread from its PD and chooses a set of PDs, one per thread, that maximize the hit rate.

8 CONCLUSIONS

Cache management policies such as replacement, bypass, and shared cache partitioning, have been exploiting – directly or indirectly – the data reuse behavior to improve cache performance. This paper proposed the novel PDP cache management policy which explicitly protects a cache line for a predicted reuse distance, the protecting distance PD. This guarantees a hit if the reuse occurs while the line is protected. Unprotecting a line speeds up its eviction and reduces cache pollution, thus freeing the cache space for new lines. The PDP achieves a good balance between reusing lines and reducing pollution. A bypass mechanism can be used when all the lines in a set are protected. A hit rate model as a function of program behavior and PD is proposed and shown to be accurate. The model is used to periodically recompute the PDP to adapt to phase changes and memory access behavior. A shared-cache hit rate model is also developed and used by the new PD-based partitioning policy. The additional hardware is shown to have the overhead similar to existing replacement policies. The performance evaluation of PDP shows that it outperforms existing management policies for both single-core and multi-core configurations.

Benchmark	SPDP		Full	Real, $S_c =$			
	NB	B		1	2	4	8
403.gcc	68	72	63	64	64	64	64
429.mcf	136	152	182	182	180	184	176
433.milc	248	248	216	216	216	216	208
434.zeusmp	16	16	16	16	16	16	16
436.cactusADM	76	72	65	66	66	66	64
437.leslie3d	36	40	40	40	40	40	48
450.soplex	60	52	73	74	72	72	80
456.hmmer	44	176	22	22	24	24	32
459.GemsFDTD	24	24	26	26	28	32	32
462.libquantum	256	256	256	256	256	256	256
464.h264ref	40	40	48	48	48	48	48
470.lbm	52	52	50	50	52	56	64
471.omnetpp	20	20	33	32	32	32	32
473.astar	24	20	16	16	16	16	16
482.sphinx3	84	120	84	84	84	88	80
483.xalancbmk.1	88	100	113	114	116	112	112
483.xalancbmk.2	72	88	56	56	56	56	64
483.xalancbmk.3	96	124	74	74	72	72	80

Table 3. The PDs of SPEC CPU2006 benchmarks.

A RESULTS FOR SPEC CPU2006 BENCHMARKS

Table 3 shows the PDs of the benchmarks used in this paper. The first two columns are for the static PDP (see Sec. 2) with and without bypass (B and NB). The third column shows the PDs of the dynamic PDP with a full RD sampler (see Sec. 6.1). The last four columns are the PDs with a real RD sampler and different values of S_c (see Sec. 6.1). These results were collected at the end of the 1B instruction execution window.

Fig. 13 shows the RD distribution for all benchmarks and windows, the modeled hit rates (E) and the static hit rates as described in Sec. 2. Each is normalized to its highest peak.

ACKNOWLEDGMENTS

This work is supported in part by NSF grant CISE-SHF 1118047 and by the Ministry of Science and Technology of Spain and the European Union (FEDER funds) under contract TIN2007-60625. Nam Duong is also supported by the Dean's Fellowship, Donald Bren School of Information and Computer Sciences, UC Irvine. The authors would like to thank the anonymous reviewers for their useful feedback and Drs. G. Loh and M. Qureshi for their shepherding help.

REFERENCES

[1] A. Basu, N. Kirman, M. Kirman, M. Chaudhuri, and J. Martinez. Scavenger: A new last level cache architecture with global block priority. In *MICRO'07*.

[2] L. A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal, 1966*.

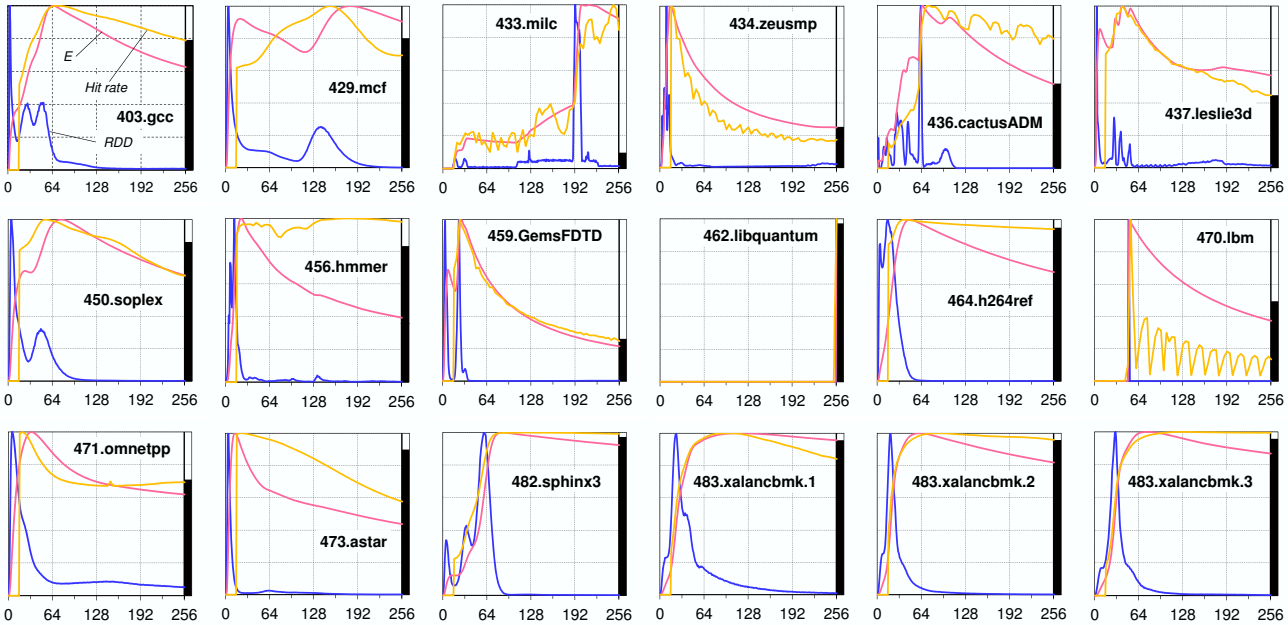


Figure 13. The RDDs, modeled and actual hit rates of SPEC CPU2006 benchmarks.

[3] N. Duong, R. Cammarota, D. Zhao, T. Kim, and A. Vendenbaum. SCORE: A score-based memory cache replacement policy. In *JWAC'10*.

[4] M. Feng, C. Tian, C. Lin, and R. Gupta. Dynamic access distance driven cache replacement. *ACM Transactions on Architecture and Code Optimization, 2011*.

[5] H. Gao and C. Wilkerson. A dueling segmented LRU replacement algorithm with adaptive bypassing. In *JWAC'10*.

[6] J. Gaur, M. Chaudhuri, and S. Subramoney. Bypass and insertion algorithms for exclusive last-level caches. In *ISCA'11*.

[7] B. S. Gill and D. S. Modha. SARC: sequential prefetching in adaptive replacement cache. In *ATEC'05*.

[8] D. Hackenberg, D. Molka, and W. E. Nagel. Comparing cache architectures and coherency protocols on x86-64 multicore SMP systems. In *MICRO'09*.

[9] M. Hayenga, A. Nere, and M. Lipasti. MadCache: A PC-aware cache insertion policy. In *JWAC'10*.

[10] A. Jaleel, E. Borch, M. Bhandaru, S. C. Steely Jr., and J. Emer. Achieving non-inclusive cache performance with inclusive caches: Temporal locality aware (TLA) cache management policies. In *MICRO'10*.

[11] A. Jaleel, R. S. Cohn, C. keung Luk, and B. Jacob. CMP$im: A pin-based on-the-fly multi-core cache simulator. In *MoBS'08*.

[12] A. Jaleel, W. Hasenplaugh, M. Qureshi, J. Sebot, S. Steely, Jr., and J. Emer. Adaptive insertion policies for managing shared caches. In *PACT'08*.

[13] A. Jaleel, H. H. Najaf-abadi, S. Subramaniam, S. C. Steely, and J. Emer. CRUISE: cache replacement and utility-aware scheduling. In *ASPLOS'12*.

[14] A. Jaleel, K. B. Theobald, S. C. Steely, Jr., and J. Emer. High performance cache replacement using re-reference interval prediction (RRIP). In *ISCA'10*.

[15] S. Jiang and X. Zhang. LIRS: an efficient low inter-reference recency set replacement policy to improve buffer cache performance. In *SIGMETRICS'02*.

[16] T. Johnson, D. Connors, M. Merten, and W.-M. Hwu. Run-time cache bypassing. *IEEE Transactions on Computers, 1999*.

[17] G. Keramidas, P. Petoumenos, and S. Kaxiras. Cache replacement based on reuse-distance prediction. In *ICCD'07*.

[18] S. M. Khan, Y. Tian, and D. A. Jimenez. Sampling dead block prediction for last-level caches. In *MICRO'10*.

[19] M. Kharbutli and Y. Solihin. Counter-based cache replacement and bypassing algorithms. *IEEE Transactions on Computers, 2008*.

[20] H. Liu, M. Ferdman, J. Huh, and D. Burger. Cache bursts: A new approach for eliminating dead blocks and increasing cache efficiency. In *MICRO'08*.

[21] R. Manikantan, K. Rajan, and R. Govindarajan. NUCache: An efficient multicore cache organization based on next-use distance. In *HPCA'11*.

[22] P. Petoumenos, G. Keramidas, and S. Kaxiras. Instruction-based reuse-distance prediction for effective cache management. In *SAMOS'09*.

[23] V. Phalke and B. Gopinath. An inter-reference gap model for temporal locality in program behavior. In *SIGMETRICS'95*.

[24] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer. Adaptive insertion policies for high performance caching. In *ISCA'07*.

[25] M. K. Qureshi and Y. N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *MICRO'06*.

[26] K. Rajan and G. Ramaswamy. Emulating optimal replacement with a shepherd cache. In *MICRO'07*.

[27] D. Sanchez and C. Kozyrakis. Vantage: scalable and efficient fine-grain cache partitioning. In *ISCA'11*.

[28] D. L. Schuff, M. Kulkarni, and V. S. Pai. Accelerating multicore reuse distance analysis with sampling and parallelization. In *PACT'10*.

[29] Y. Smaragdakis, S. Kaplan, and P. Wilson. EELRU: simple and effective adaptive page replacement. In *SIGMETRICS'99*.

[30] S. Srikantaiah, M. Kandemir, and Q. Wang. SHARP control: controlled shared cache management in chip multiprocessors. In *MICRO'09*.

[31] M. Takagi and K. Hiraki. Inter-reference gap distribution replacement: an improved replacement algorithm for set-associative caches. In *ICS'04*.

[32] G. Tyson, M. Farrens, J. Matthews, and A. R. Pleszkun. A modified approach to data cache management. In *MICRO'95*.

[33] C. Wood, E. B. Fernandez, and T. Lang. Minimization of demand paging for the LRU stack model of program behavior. *Information Processing Letters, 1983*.

[34] C.-J. Wu, A. Jaleel, W. Hasenplaugh, M. Martonosi, S. Steely, and J. Emer. SHiP: Signature-based hit predictor for high performance caching. In *MICRO'11*.

[35] C.-J. Wu, A. Jaleel, M. Martonosi, S. C. Steely, Jr., and J. Emer. PACMan: prefetch-aware cache management for high performance caching. In *MICRO'11*.

[36] Y. Xie and G. H. Loh. PIPP: promotion/insertion pseudo-partitioning of multi-core shared caches. In *ISCA'09*.

2012 IEEE/ACM 45th Annual International Symposium on Microarchitecture

Kernel Partitioning of Streaming Applications: A Statistical Approach to an NP-complete Problem

Petar Radojković[1,2] Paul M. Carpenter[1,2] Miquel Moretó[1,2,4] Alex Ramirez[1,2] Francisco J. Cazorla[1,3]
[1]Barcelona Supercomputing Center [2]Universitat Politècnica de Catalunya - Barcelona TECH
[3]Spanish National Research Council (IIIA-CSIC) [4]International Computer Science Institute, Berkeley
{petar.radojkovic, paul.carpenter, miquel.moreto, alex.ramirez, francisco.cazorla}@bsc.es

Abstract

One of the greatest challenges in computer architecture is how to write efficient, portable, and correct software for multi-core processors. A promising approach is to expose more parallelism to the compiler, through the use of domain-specific languages. The compiler can then perform complex transformations that the programmer would otherwise have had to do. Many important applications related to audio and video encoding, software radio and signal processing have regular behavior that can be represented using a stream programming language. When written in such a language, a portable stream program can be automatically mapped by the stream compiler onto multicore hardware. One of the most difficult tasks of the stream compiler is partitioning the stream program into software threads. The choice of partition significantly affects performance, but finding the optimal partition is an NP-complete problem.

This paper presents a method, based on Extreme Value Theory (EVT), that statistically estimates the performance of the optimal partition. Knowing the optimal performance improves the evaluation of any partitioning algorithm, and it is the most important piece of information when deciding whether an existing algorithm should be enhanced. We use the method to evaluate a recently-published partitioning algorithm based on a heuristic. We further analyze how the statistical method is affected by the choice of sampling method, and we recommend how sampling should be done. Finally, since a heuristic-based algorithm may not always be available, the user may try to find a good partition by picking the best from a random sample. We analyze whether this approach is likely to find a good partition. To the best of our knowledge, this study is the first application of EVT to a graph partitioning problem.

1. Introduction

Stream programming is suitable for applications that process long sequences of data, such as voice, image, multimedia content, Internet and communication traffic. Stream programming languages such as StreamIt [8, 39], Brook [7] and SPM [2, 10] represent the program as concurrent kernels, which communicate only via point-to-point streams. A kernel is a basic computation block with a user-defined function that processes one or more input data streams into one or more output data streams. Dependencies between different kernels are described explicitly through the communication data channels. The whole application can be represented as a *stream graph*. The nodes of the stream graph correspond to the kernels, while the directed edges represent the communication data channels.

There are three main advantages of stream programming languages, compared with traditional languages such as *C*. First, a stream programming language is a domain-specific language, which provides a natural way to describe streaming applications. Second, when the program is described using a stream language, the compiler may perform complex optimizations over the stream graph, producing an efficient multithreaded program. Some optimizations involve major changes to the program's structure and data layout. Third, unlike some other parallel programming models, including multithreading, a stream program is deterministic, and therefore easier to debug.

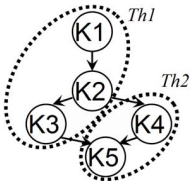


Figure 1: A kernel partition example

In order to take advantage of multiple processor cores the stream program is automatically compiled into a multithreaded executable by the stream compiler. One of the most important tasks of the stream compiler is to partition the kernels in the stream graph into software threads. In Figure 1, we illustrate a stream graph of a simple program, which is comprised of five kernels (K1 to K5). The figure also shows one possible kernel partition of the graph: kernels K1, K2, and K3 are to be compiled to software thread *Th1*, while kernels K4 and K5 are to be compiled into *Th2*.

Kernel partitioning can significantly affect the overall system performance. For example, for the benchmarks included in the StreamIt 2.1.1 suite, the relative performance difference between good and bad kernel partitions of the same benchmark mapped into four software threads ranges from 2.4× to 3.9×, and on average it is 3.5×.

The optimal kernel partition cannot be determined because the essence of the analysis is graph partitioning, which is an NP-complete problem [18, 20]. Due to the large exploration space, brute force exploration is impractical: as streaming applications comprise tens or hundreds of interconnected kernels (54 kernels on average in the StreamIt 2.1.1 suite), the number of possible kernel partitions is vast. For example, the *channelvocoder* benchmark has 55 kernels, and it can be distributed into 10^{22} partitions of exactly four software threads. The number of possible kernel partitions increases rapidly with the number of output software threads, so the number of partitions using eight threads is 10^{34}.

Several studies (see Section 6) propose heuristic-based algorithms to address the kernel partitioning problem. As kernel partitioning is an intractable problem, it is impossible in general to know the performance of the optimal partition, so the room for improvement is also unknown. It is hard to decide whether to invest additional effort to try to improve a given algorithm, since it may already be close to optimal.

In this paper, we present a statistical approach to the kernel partitioning problem. We present a method that predicts the performance of the optimal partition based on the observed performance of each kernel partition in a random sample. The method is based on Extreme Value Theory (EVT), a branch of statistics that analyzes extreme de-

1072-4451/12 $26.00 © 2012 IEEE
DOI 10.1109/MICRO.2012.44

IEEE computer society

viation from the population median. We also present several different approaches to generate the random samples. Finally, we show that the performance of the best observed kernel partition in a random sample is likely to be close to the optimal one. If a heuristic is not available for the user's problem, a good kernel partition can be found using random sampling on its own.

The main contributions of our study are:

- We present a method based on EVT that statistically estimates the performance of the optimal kernel partition. To the best of our knowledge, this is the first study that applies EVT to a graph partitioning problem.
- We show that the sampling method, used to generate random partitions, has a significant effect on the applicability of the statistical method. We analyze different sampling methods, and our results strongly recommend that the samples should be uniformly distributed.
- We use the estimates of the optimal performance to evaluate a state-of-the-art heuristic-based kernel partitioning algorithm.
- Finally, since a complex heuristic-based algorithm may not always be available, the user may pick the best from a random sample, and measure its quality using the estimates of the optimal performance. We analyze whether random sampling is likely to find a good kernel partition.

The presented analysis is evaluated for the benchmarks included in the StreamIt 2.1.1 suite. The method based on EVT successfully estimates the performance of the optimal kernel partitions for all the benchmarks under study. In all experiments, the two different kernel partitioning methods, the heuristics-based algorithm and the method based on random sampling, detected kernel partitions with practically the same performance.

The rest of the paper is organized as follows. Section 2 describes metrics that can be used to measure the performance of streaming applications. Section 3 discusses methods to generate random samples of kernel partitions. Section 4 presents the statistical analysis that we use to estimate the performance of the optimal kernel partition. In Section 5, we apply the presented analysis to the StreamIt 2.1.1 benchmark suite, and evaluate the results. Section 6 describes related work, and Section 7 presents the conclusions of the study.

2. Background

In this section, we describe different metrics that could be of interest when doing kernel partitioning. We also briefly describe related work with a focus on the conclusions that directly affect our study.

2.1. Target metric

There are several metrics that can describe the performance of streaming applications. In our study, we analyze the *cost* of streaming applications. The *cost* is proportional to the time needed to process a fixed amount of input data. This metric corresponds to execution time for non-streaming applications. Other metrics include energy or power, the hardware utilization of the target architecture, or some weighted sum of them.

The input to the statistical analysis is the cost of each kernel partition in a random sample. The costs are generated using metrics from the StreamIt 2.1.1 compiler. Instead of using the streaming compiler, the target metric could alternatively have been measured using real execution or simulation.

The proposed statistical approach and the general conclusions of this study are independent of both the target metric and the way in which the metric is evaluated: program compilation, execution or simulation. It is important to note, however, that the results from the statistical analysis are clearly dependent on the quality of the samples provided to it.

If the application behavior is sensitive to its input data, which is generally not the case for streaming applications, the user should consider the analysis for different input datasets that are representative for different application behavior.

If the user wants to use the statistical method for multiple objective functions separately, then it is only necessary to do a full set of compilations, executions or simulations once. After obtaining a complete set of metrics, the statistical analysis can be done multiple times using different metrics.

2.2. Convexity constraint

Carpenter et al. [9] present a partitioning and allocation algorithm for an iterative stream compiler. The algorithm produces kernel partitions that are easier to compile and that require short pipelines of software threads. The authors evaluate their proposal on the benchmarks included in the StreamIt 2.1.1 suite.

One of the conclusions in that paper is that the kernel partition should be convex. A kernel partition is convex if the dependencies between different software threads form an *acyclic* graph. This means that every directed path between two kernels in the same software thread is internal to that thread. The reason for the convexity constraint is that the choice of partition affects the length of the pipeline generated by the streaming compiler. The convexity constraint controls the length of that pipeline. The authors demonstrate that without the convexity constraint, the compiler may generate long pipelines of software threads, which increases memory use and latency of the inter-thread communication, significantly affecting the overall performance.

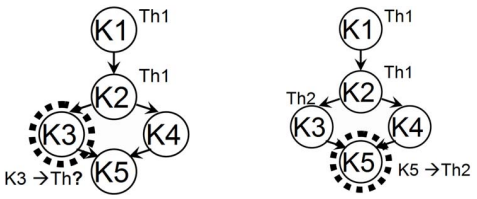
We follow these instructions and focus our study on the analysis of *convex* kernel partitions. We pay special attention to generating random samples comprised only of kernel partitions that satisfy the convexity constraint. It is important to notice, however, that convexity is not a requirement of the proposed statistical approach. The approach can be also used for the analysis of non-convex kernel partitions.

Although the convexity constraint significantly reduces the number of kernel partitions, their number is still vast, and brute force exploration is impractical. For example, the fm benchmark can be distributed into 10^{12} convex kernel partitions of exactly four software threads, radar into 10^{14} partitions, filterbank into 10^{20}, and vocoder into 10^{23}.

3. Sampling methods

In order to apply Extreme Value Theory (EVT) to the kernel partitioning problem for streaming applications, we need to generate random convex partitions of the stream graph that are independent and identically distributed (*i.i.d.*). Intuitively, random variables are independent if knowing the value of one of them gives no new information about the values of the others; they are identically distributed if they all have the same probability distribution, which does not have to be uniform. Uniform distribution would mean that each kernel partition would be selected with the same probability.

The different methods we used to select the random *i.i.d.* kernel partitions are described next. For each method, we describe how to select a single random kernel partition. To generate a sample of N *i.i.d.* kernel partitions, repeat the sampling method N times.



(a) Assigning kernel K3 (b) Assigning kernel K5

Figure 2: DFS sampling method

3.1. Depth-First Search (DFS)

The first sampling method generates a random kernel partition using a depth-first search (DFS) of the stream graph. The kernels are visited in sequence, each kernel being assigned with equal probability to any software thread that would not violate the convexity constraint. Thus, this method generates random kernel partitions in a single traversal of the stream graph.

We illustrate the sampling method with the example shown in Figure 2. The example stream graph contains five kernels (K1 to K5) that are to be compiled into two software threads (Th1 and Th2). For example, assume that kernels K1 and K2 are already assigned to Th1, and that the next kernel to be assigned is K3 (Figure 2(a)). K3 can be assigned with equal probability to Th1 or Th2. If K3 is assigned to Th2, kernel K5 has to be assigned to Th2 in order to generate a convex partition (see Figure 2(b)). Since the number of candidate threads that do not break the convexity constraint decreases rapidly, the DFS sampling method often generates kernel partitions with an unbalanced number of kernels per threads.

3.2. Edge Contraction (EC)

The second sampling method generates a random partition using edge contraction of the stream graph. Initially, each kernel is placed in its own cluster. Then, the edges of the stream graph are visited in random order. In each step the selected edge of the graph is contracted by fusing the clusters connected through this edge. If the resulting graph violates the convexity constraint, the contraction is undone. The process is continued, by moving to the next edge in random sequence, until the number of clusters equals the number of software threads. Finally, the clusters are randomly assigned to the threads.

Figure 3 illustrates the EC kernel partitioning of a simple stream graph into two software threads. For example, assume that K1→K2 edge is selected as the first edge to be contracted. In this case, kernels K1 and K2 are fused into a single cluster while the rest of the graph is not modified. Afterwards, we illustrate the contraction of edges K3→K5 and (K1&K2)→K4. Finally, the clusters are randomly distributed among software threads. In comparison with DFS, the EC sampling method generates random kernel partitions with a balanced number of kernels.

3.3. Edge Contraction with Filter (EC-F)

The third sampling method is an enhancement of the EC method, designed to bias the sampling towards kernel partitions with a low cost, which will lead to good application performance. The EC-F method selects the edges of the stream graph in random order, fuses the corresponding clusters, and checks whether the convexity constraint is violated, as for the EC sampling method. The only difference is that EC-F performs an additional check: if contracting the current edge generates a cluster with a high cost (i.e. that exceeds a given

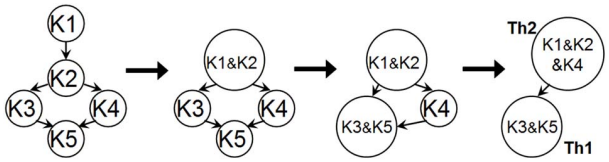


Figure 3: EC sampling method: Contracting K1→K2, K3→K5, and (K1&K2)→K4 edges, respectively.

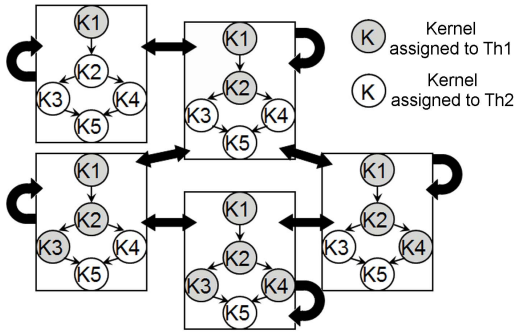


Figure 4: UD sampling method: Example partition graph for a small stream program

threshold), the contraction is undone and the process is repeated for a different edge. In the experiments presented in the paper, the threshold was the lowest cost detected in ten random kernel partitions generated using the EC sampling method.

It may happen that, although the number of clusters is still greater than the number of threads, none of the edges can be contracted without creating a cluster of cost exceeding the threshold. In this case, the remaining edges are visited in a new random order, and edges are contracted without checking whether the cost of the final clusters is over the threshold. Finally, the clusters are then randomly assigned to threads. In contrast with other presented sampling methods, this method does take into account the cost of each particular kernel when generating partitions. The main target of this algorithm is to generate random partitions with a balanced cost among clusters.

3.4. Uniformly Distributed (UD) sampling

The final sampling method generates a uniform sampling distribution of the kernel partitions. This means that each convex kernel partition is selected with the same probability. It is important to notice that the statistical method used in the study does not require the kernel partitions to be uniformly distributed, since it only requires their costs to be independent and identically distributed (*i.i.d.*). In general, previous sampling methods do not provide uniformly distributed kernel partitions.

This sampling method comprises three steps.

Step 1: We analyze different kernel partitions using the *partition graph*, the graph of all possible convex partitions of the stream graph under study. Each node of the partition graph is a different kernel partition, so the number of nodes is equal to the number of partitions. There is an undirected edge between two nodes of the partition graph if they differ in the assignment of exactly one kernel partition. Also, each node contains a self-loop edge, an edge that connects the node to itself. Due to its large size (this is an NP-complete problem), the partition graph is never actually constructed in its entirety. An example partition graph, for a small stream program that is to be assigned to two software threads, is shown in Figure 4.

Step 2: We perform a random walk on the partition graph. First, we have to choose an initial node of the partition graph to start the

random walk. This node can be selected by any method that generates random kernel partitions. In the experiments presented in our study, the initial kernel partition (initial node of the partition graph) is selected using the EC sampling method. The random walk starts from the initial node in the partition graph and calculates all its neighbors. Then it randomly chooses one of the neighbors (using a uniform distribution) to be the next node that is to be visited. The neighbor selection is repeated N times, with N large enough to potentially visit all the nodes of the partition graph of the benchmarks used in the study[1]. The last node of the partition graph that is visited is the outcome of the random walk. The probability that a given node is selected using the random walk is directly proportional to its degree (the number of its neighbors in the graph) [32]. As we know the probability of each visited node of the partition graph to be selected, the random walk generates samples with a known distribution.

Step 3: In the final step of this sampling method, we convert the output of the random walk from a known distribution to a uniform distribution. In order to do so, each kernel partition selected using the random walk is included in the outcome of this sampling method with a probability that is inversely proportional to its degree in the partition graph. This way, every convex partition has the same probability of being selected, i.e. the method provides uniformly distributed samples.

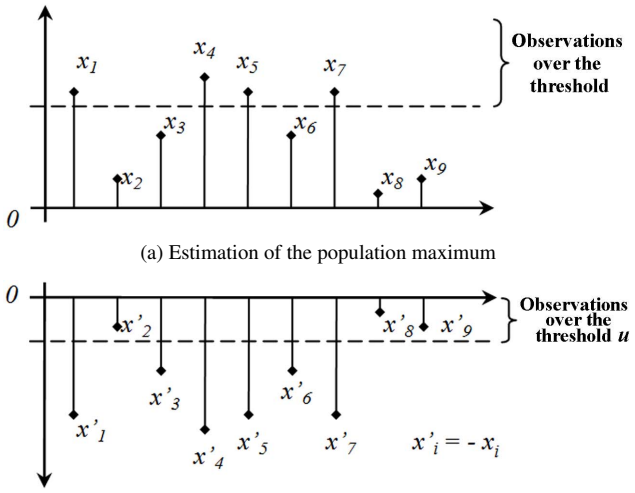
3.5. Statistical *i.i.d.* tests

The sampling methods described in the previous section are designed to generate random *i.i.d.* samples. After generating the samples, we perform statistical tests to confirm that they are indeed independent and identically distributed.

Wald–Wolfowitz test: The *Wald–Wolfowitz* test or *runs* test examines whether the observations in the sample are mutually independent [6, 16]. The test comprises two main steps. First, the costs of kernel partitions (non-negative real numbers) have to be converted into binary values. We converted the cost of a given kernel partition to *'0'* if its value was below the median cost in the sample, and converted it to *'1'* otherwise. This way, the sequence of non-negative real numbers was converted into a sequence of *0s* and *1s*, e.g. *000110000*. In the second step, the test analyzes the sub-sequences of consecutive identical values (*0s* or *1s*), which are referred to as *runs*. For example, the sequence *000110000* is composed of three runs: *000*, *11*, and *0000*. The Wald–Wolfowitz test validates that the observations in the sample under study are mutually independent if the lengths of the runs follow a Gaussian distribution [6]. The mutual independence hypothesis was tested at the 0.05 significance level. All the samples used in the study passed the test.

Kolmogorov–Smirnov test: In order to validate that selected kernel partitions in a given sample are identically distributed, we used a two-sample *Kolmogorov–Smirnov* test [16, 19]. The test compares the empirical cumulative distribution functions (ECDF) of two data sets and, based on the maximum distance between the two ECDFs, it confirms or rejects the hypothesis that the data sets correspond to the same distribution. The identically distributed test that we performed contains three steps. First, we generated a random sample of 20,000 kernel partitions and observed the cost of each partition. The costs of the kernel partitions in the sample followed the order in which the partitions were generated. Second, in each experiment, we observed two randomly-selected segments of m consecutive values from the original sample. Finally, we used a two-sample *Kolmogorov–Smirnov* test to check whether the randomly selected segments of the sample have the same probability distribution. If the kernel partition costs are indeed identically distributed, then all segments of consecutive values in the sample have the same distribution. For each sample used the study, we performed the test for segments of m =100, 500, 1,000 and 5,000 observations. All the samples used in the study passed the test at the 0.05 significance level.

[1] In our experiments, $N = 100$.



(a) Estimation of the population maximum

(b) Estimation of the population minimum

Figure 5: Exceedances over the threshold

4. A statistical approach to kernel partitioning of streaming applications

We estimate the minimal cost of kernel partitions (that lead to optimal performance) using Extreme Value Theory (EVT). EVT is a branch of statistics that studies extreme deviations from the median [5, 12]. One of the approaches in EVT is the *Peak Over Threshold* (POT) method. In its original form, the POT method takes into account only the distribution of the observations that exceed a given (high) threshold to estimate the population maximum [4, 35]. For example, in Figure 5(a), the observations x_1, x_4, x_5, and x_7 exceed the threshold and constitute extreme values, which can be used by POT analysis.

The POT method can also be used to estimate the population minimum [21, 28]. Estimation of the minimum requires the following five steps explained in detail in the next section:

- Obtain *i.i.d.* observations x_i of the cost of kernel partitions.
- Invert the sign of the observations: $x'_i = -x_i$.
- Determine the threshold u, shown in Figure 5(b).
- Use the values x'_i over the threshold u to estimate the maximum cost of the inverse population ($Max(Cost_{Inv})$).
- The minimum cost of the original population ($Min(Cost)$) corresponds to the negative value of the maximum of the inverse population: $Min(Cost) = -Max(Cost_{Inv})$;

The POT method can also be explained using cumulative distribution functions (CDF). For example, assume that F is the CDF of a random variable X. The POT method can be used to estimate the cumulative distribution function F_u of values of x above a certain threshold u. The function F_u is called the *conditional excess distribution function* and it is defined as

$$F_u(y) = P(X - u \leq y \mid X > u), \quad 0 \leq y \leq x_F - u,$$

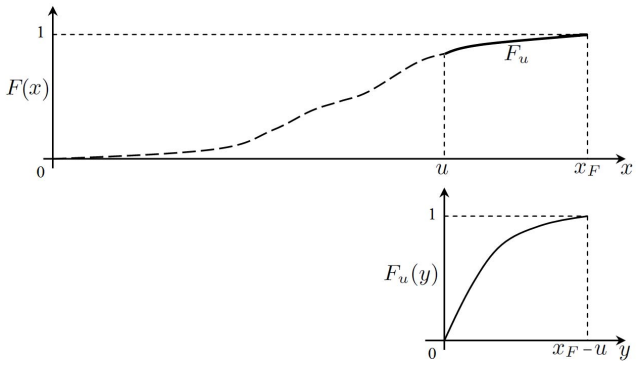


Figure 6: Cumulative distribution function $F(x)$ and matching conditional excess distribution function $F_u(y)$

where X is the observed random variable, u is the given threshold, $y = x - u$ are the exceedances over the threshold, and $x_F \leq \infty$ is the right endpoint of the cumulative distribution function F. Figure 6 shows a CDF of a random variable X (upper chart) and the corresponding conditional excess distribution function $F_u(y)$ (bottom chart).

The POT method is based on the following theorem [4, 35]:

Theorem 1 *For a large class of underlying distribution functions F, the conditional excess distribution function $F_u(y)$, for large threshold u, is well approximated by $F_u(y) \approx G_{\xi,\sigma}(y)$ where*

$$G_{\xi,\sigma}(y) = \begin{cases} 1 - (1 + \frac{\xi}{\sigma} y)^{-1/\xi} & for \quad \xi \neq 0 \\ 1 - e^{-y/\sigma} & for \quad \xi = 0 \end{cases}$$

for $y \in [0, (x_F - u)]$ if $\xi \geq 0$ and $y \in [0, -\frac{\sigma}{\xi}]$ if $\xi < 0$, where $G_{\xi,\sigma}$ is called Generalized Pareto Distribution (GPD).

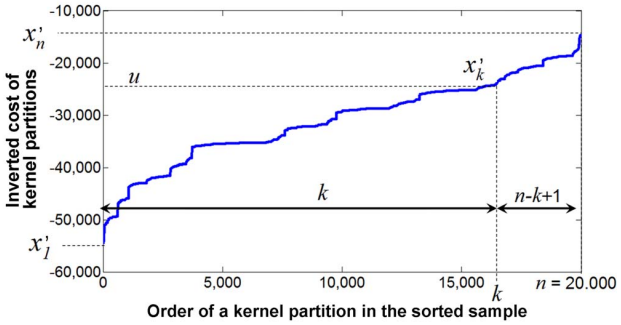
This means that for numerous distributions that present real-life problems, F_u can be approximated with a GPD. For each particular problem, the decision of whether GPD can be used to model the problem or not, is based on how well the sample of observations can be fitted to GPD. We describe the goodness of fit of observations to GPD in Steps 3 and 4 of Section 4.1. GPD is defined with two parameters: shape parameter ξ and scaling parameter σ. One of the characteristics of GPD is that for $\xi < 0$ the upper bound of the observed value equals $u - \frac{\sigma}{\xi}$, where σ and ξ are the GPD parameters and u is the selected threshold [21, 28].

In Theorem 1, the definition of $G_{\xi,\sigma}(y)$ for $\xi = 0$ can only be used to model problems with an infinite upper bound [21, 28]. In this study we use the GPD to estimate the minimal cost of kernel partitions for streaming applications. The value of this cost is always finite, and the estimated values of the parameter ξ are always $\hat{\xi} < 0$. Therefore, for the sake of simplicity of the presented mathematical formulas, in the rest of the paper we do not present $G_{\xi,\sigma}(y)$ formulas for $\xi = 0$.

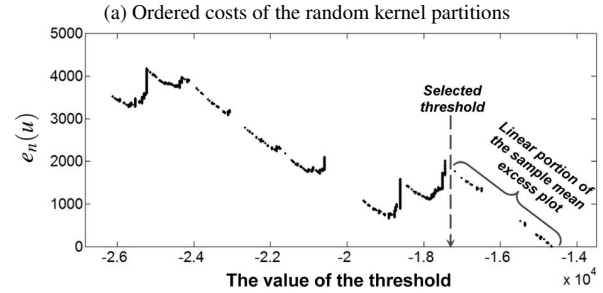
4.1. Application of Peak Over Threshold method

We use the POT method to estimate the minimal cost of kernel partitions for streaming benchmarks based on the cost of a sample of random partitions. Application of the POT method involves the following six steps:

Step 1: Generate the sample of random kernel partitions, and determine the cost of each partition in the sample (x_i). A requisite of the presented statistical analysis is that the selected kernel partitions must be independent and identically distributed (*i.i.d.*). The proposed methods to generate *i.i.d.* kernel partitions are described in Section 3. All of them passed the described *i.i.d.* tests.



(a) Ordered costs of the random kernel partitions



(b) Sample mean excess plot

Figure 7: Selection of the threshold for *mpeg2-subset*

Step 2: Invert the sign of the values of the observed costs. Originally, the POT method was used to estimate the maximum of a population based on a set of random *i.i.d.* observations. In order to estimate the *minimum* cost of kernel partitions, we invert the sign of the observed values ($x'_i = -x_i$) and estimate the *maximum* of the *inverse* population.

Step 3: Select the threshold u. The selection of the threshold u is an important step in POT analysis. Gilli and Këllezi [21, 28] propose using the *sample mean excess plot*, a graphical tool for threshold selection. This method first sorts all task assignments in the sample in non-decreasing cost order: $x'_1 \leq x'_2 \leq \cdots \leq x'_n$. Figure 7(a) shows the sorted cost of 20,000 uniformly distributed random kernel partitions of *mpeg2-subset* benchmark (see Section 3.4).

Then, the possible threshold u takes the values from x'_1 to x'_n ($x'_1 \leq u \leq x'_n$), and for each value we compute the sample mean excess function $e_n(u)$:

$e_n(u) = \frac{\sum_{i=k}^{n}(x'_i - u)}{n-k+1}$, where $k = min\{ \ i \ | \ x'_i > u\}$.

In this formula, the factor $n-k+1$ is the number of observations that exceed the threshold. Finally, the sample mean excess plot is defined by the points $(u, e_n(u))$ for $x'_1 \leq u \leq x'_n$. Figure 7(b) shows the example of the sample mean excess plot for the *mpeg2-subset* benchmark.

As commented before, the estimated parameter ξ of GPD must be negative ($\hat{\xi} < 0$) to obtain the upper bound of the $Max(Cost_{Inv})$. A characteristic of the GPD with parameter $\xi < 0$ is that it has a linear mean excess function plot. In order to have a good fit of the conditional distribution function F_u to GPD, the threshold should be selected so that the observations that exceed the threshold have a roughly linear sample mean excess plot. As an example, for the data presented in Figure 7, the threshold should be selected

to be $u = -17,500$. The sample mean excess plot is also a good tool to test whether GPD can be used to model a particular set of observations. If the right portion of the mean excess plot for the sample of measured task assignments performance is not roughly linear, that particular problem cannot be modeled using GPD.

Another important tool that can be used to understand if a given sample of observations can be modeled with a GPD is a *quantile plot* [5, 28]. In a quantile plot, the sample quantiles x'_i are plotted against the quantiles of a target distribution $F^{-1}(q_i)$ for $i = 1,\ldots,n$. If the sample data originates from the family of distributions F, the plot is close to a straight line.

The linear sample mean excess plot and the quantile plot are not the only constraints that should be considered when selecting the threshold. If the threshold is too low, the estimated parameters of GPD may be biased to the median values of the cumulative distribution function instead of to the maximum values. In order to avoid this bias, when selecting a threshold we have to ensure that the number of observations that exceed the selected threshold is not higher than 5% of the task assignments in the whole sample. This is a commonly used limit in studies that use POT analysis [21, 28, 36].

Step 4: Fit the GPD function to the observations that exceed the threshold and estimate parameters ξ and σ. Once the threshold u is selected, the observations over the threshold can be fitted to GPD, and the parameters of the distribution can be estimated. For the sake of simplicity, we assume that observations from x'_k to x'_n in the sorted sample presented in Figure 7(a) exceed the threshold. We rename the exceedances $y_{i-k+1} = x'_i - u$ for $k \le i \le n$ and use the set of elements $\{y_1, y_2, ..., y_m\}$ to estimate the parameters of GPD. The number of elements in the set, $m = n-k+1$, is the number of exceedances over the threshold.

Different methods can be used to estimate the parameters of GPD from a sample of observations [11, 24, 26, 38]. In our study, we used estimation based on the *likelihood* function [3]. The GPD has parameters ξ and σ. The likelihood that a set of observations $Y = \{y_1, y_2, ..., y_m\}$ is the outcome of a GPD with parameters $\xi = \xi_0$ and $\sigma = \sigma_0$ is defined to be the probability that GPD with parameters ξ_0 and σ_0 has outcome Y.

We make use of the likelihood function to compute the probability of different values of GPD parameters for a given set of observations $\{y_1, y_2, ..., y_m\}$. As the logarithm is a monotonically increasing function, the logarithm of a positive function achieves the maximum value at the same point as the function itself. This means that instead of finding the maximum of a likelihood function, we can determine the maximum of the logarithm of the likelihood function, the *log-likelihood* function. In statistics, log-likelihood is frequently used instead of the likelihood function because it simplifies computations. The estimation of parameters ξ and σ of $G_{\xi,\sigma}(y)$ involves the following steps:

(i) Determine the corresponding probability density function as a partial derivative of $G_{\xi,\sigma}(y)$ with respect to y:

$$g_{\xi,\sigma}(y) = \frac{\partial G_{\xi,\sigma}(y)}{\partial y} = \frac{1}{\sigma}(1+\frac{\xi}{\sigma}y)^{-\frac{1}{\xi}-1}$$

(ii) Find the logarithm of $g_{\xi,\sigma}(y)$:

$$log\ (g_{\xi,\sigma}(y)) = -log\ \sigma - (\frac{1}{\xi}+1)\ log(1+\frac{\xi}{\sigma}y)$$

(iii) Compute the log-likelihood function $L(\xi,\sigma|y)$ for the GPD as the logarithm of the joint density of the observations $\{y_1, y_2, ..., y_m\}$:

$$L(\xi,\sigma|y) = \sum_{i=1}^{m} log\ g_{\xi,\sigma}(y_i)$$

$$L(\xi,\sigma|y) = -m\ log\ \sigma - (\tfrac{1}{\xi}+1)\sum_{i=1}^{m} log(1+\frac{\xi}{\sigma}y_i)$$

We compute estimated values of parameters $\hat{\xi}$ and $\hat{\sigma}$, to *maximize* the value of the log-likelihood function $L(\xi,\sigma|y)$ for observations $\{y_1, y_2, ..., y_m\}$:

$$L(\hat{\xi},\hat{\sigma}|y) = \max_{\xi,\sigma}(L(\xi,\sigma|y))$$

$$L(\hat{\xi},\hat{\sigma}|y) = \max(-m\ log\ \sigma - (\frac{1}{\xi}+1)\sum_{i=1}^{m} log(1+\frac{\xi}{\sigma}y_i))$$

In order to determine the parameters $\hat{\xi}$ and $\hat{\sigma}$, we find the minimum of the negative log-likelihood function, $\min_{\xi,\sigma}(-L(\xi,\sigma|y))$, using the procedure *fminsearch()* included in Matlab® R2007a [13]. The values $\hat{\xi}$ and $\hat{\sigma}$ are called the point estimates of the parameters ξ and σ, respectively.

Step 5: Estimate the maximum of the inversed costs. The maximum of the inverse cost can be determined only for $\hat{\xi} < 0$ which is satisfied for all data sets that are presented in this paper. The point estimate of $Max(Cost_{Inv})$ is computed as $\widehat{Max(Cost_{Inv})} = u - \hat{\sigma}/\hat{\xi}$.

In order to indicate the confidence of the estimate, we compute the confidence intervals of the estimated $\widehat{Max(Cost_{Inv})}$. The confidence intervals is computed using the *likelihood ratio test* [3], which consists of the following steps:

(i) Define GPD as a function of ξ and $Max(Cost_{Inv})$:

$$G_{\xi,Max(Cost_{Inv})}(y) = 1 - (1 - \frac{1}{Max(Cost_{Inv})-u}y)^{-1/\xi}$$

(ii) Determine the corresponding probability density function:

$$g_{\xi,Max(Cost_{Inv})}(y) = \frac{\partial G_{\xi,Max(Cost_{Inv})}(y)}{\partial y} = \\ = -\frac{1}{\xi(Max(Cost_{Inv})-u)}(1-\frac{1}{Max(Cost_{Inv})-u}y)^{-\frac{1}{\xi}-1}$$

(iii) Compute the joint log-likelihood function for observations $\{y_1, ..., y_m\}$:

$$L(\xi,Max(Cost_{Inv})|y) = \sum_{i=1}^{m} log\ g_{\xi,Max(Cost_{Inv})}(y_i)$$

$$L(\xi,Max(Cost_{Inv})|y) = -n\ log(-\xi(Max(Cost_{Inv})-u)) - \\ -(1+\tfrac{1}{\xi})\sum_{i=1}^{n} log(1-\frac{1}{Max(Cost_{Inv})-u}y_i)$$

(iv) Find the $Max(Cost_{Inv})$ confidence interval. We determine the confidence interval for $Max(Cost_{Inv})$ using the likelihood ratio test [3] and Wilks's theorem [14, 41, 42]. The maximum log-likelihood function is determined as:

$$L(\hat{\xi},\widehat{Max(Cost_{Inv})}|y) = \max_{\xi,Max(Cost_{Inv})}(L(\xi,Max(Cost_{Inv}))).$$

The function $L(\hat{\xi},\widehat{Max(Cost_{Inv})}|y)$ has two parameters that are free to vary (ξ and $Max(Cost_{Inv})$), hence it has two degrees of freedom ($df_1 = 2$). As $Max(Cost_{Inv})$ is our parameter of interest, the profile log-likelihood function is defined as:

$$L^*(Max(Cost_{Inv})) = \max_{\xi} L(\xi,Max(Cost_{Inv})).$$

The function $L^*(Max(Cost_{Inv}))$ has one parameter that is free to vary, *i.e.* one degree of freedom ($df_2 = 1$). Wilks's theorem applied to the problem that we are addressing claims that, for a large number of exceedances over the threshold, the distribution of $2(L(\hat{\xi},\widehat{Max(Cost_{Inv})}) - L^*(Max(Cost_{Inv})))$ converges to a χ^2 distribution with $df_1 - df_2$ degrees of freedom. Therefore, the confidence interval of $Max(Cost_{Inv})$ includes all values of $Max(Cost_{Inv})$ that satisfy the following condition:

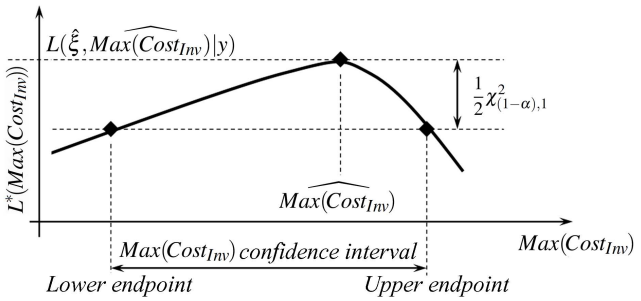


Figure 8: $Max(Cost_{Inv})$ confidence interval

$$L(\hat{\xi}, \widehat{Max(Cost_{Inv})}) - L^*(Max(Cost_{Inv})) < \frac{1}{2}\chi^2_{(1-\alpha),1} \quad (1)$$

$\chi^2_{(1-\alpha),1}$ is the $(1-\alpha)$-level quantile of the χ^2 distribution with one degree of freedom ($df_1 - df_2 = 1$). α is the confidence level for which we compute $Max(Cost_{Inv})$ confidence intervals. We illustrate the computation of the $Max(Cost_{Inv})$ confidence interval in Figure 8. The figure plots $L^*(Max(Cost_{Inv}))$ for different values of $Max(Cost_{Inv})$. For $Max(Cost_{Inv}) = \widehat{Max(Cost_{Inv})}$, L^* reaches its maximum. The confidence interval of $Max(Cost_{Inv})$ includes all values of $Max(Cost_{Inv})$ that satisfy the condition $L^*(Max(Cost_{Inv})) > L(\hat{\xi}, \widehat{Max(Cost_{Inv})}) - \frac{1}{2}\chi^2_{(1-\alpha),1}$, which corresponds to Equation 1. We computed the $Max(Cost_{Inv})$ confidence interval using an iterative method based on the *fminsearch()* function included in Matlab® R2007a.

Step 6: Estimate the minimum cost of the kernel partitions. The minimum cost of the kernel partitions corresponds to the estimated maximum of the inverse cost: $Min(Cost) = -Max(Cost_{Inv})$. Also, the lower and upper endpoints of the $Min(Cost)$ confidence interval correspond to the inverse upper and lower endpoints of the $Max(Cost_{Inv})$ confidence interval, respectively.

The code that performs the statistical *i.i.d.* test, generates the sample mean excess plots, infers the parameters of the GPD distribution, and estimates the minimum cost of kernel partitions was developed in Matlab® R2007a.

5. Results

In this section, we use the POT method to estimate, for each of the StreamIt 2.1.1 benchmarks, the cost of the optimal kernel partition. We compare the four sampling methods described in Section 3. We also evaluate the number of random kernel partitions that are required by the presented statistical approach. Finally, we analyze whether a good kernel partition would be found using random sampling on its own.

Before using any heuristics-based algorithm for the concrete application under study, the user should check whether exhaustive search would be impractical. In general, the number of valid kernel partitions is vast (e.g. 10^{20}). It is possible, however, that a particular benchmark has a small stream graph, so that exhaustive search would work. For example, the *dct* benchmark included in the StreamIt 2.1.1 suite contains only eight kernels, linked in a simple stream graph. The number of partitions of this benchmark, onto four threads, is just 32. In this case, exhaustive search is the simplest and fastest way to find the optimal kernel partition.

Table 1: Applicability of the POT method

Benchmark	Sampling method			
	DFS	EC	EC-F	UD
bitonic-sort	*NA*	✔	*NA*	✔
channelvocoder	✔	*NA*	*NA*	✔
des	*NA*	✔	✔	✔
fft	*NA*	✔	*NA*	✔
filterbank	*NA*	*NA*	*NA*	✔
fm	✔	*NA*	*NA*	✔
mpeg2-subset	*NA*	✔	*NA*	✔
radar	✔	*NA*	*NA*	✔
serpent_full	*NA*	✔	*NA*	✔
tde_pp	*NA*	✔	*NA*	✔
vocoder	*NA*	✔	*NA*	✔

5.1. Estimation of the minimal cost using the POT method

We apply our technique to estimate the performance of the optimal kernel partition on four threads, for each of the benchmarks in the StreamIt 2.1.1 suite. As discussed in the previous section, the *dct* benchmark can be solved using exhaustive search, leaving eleven benchmarks to be analyzed using the POT statistical method.

For each benchmark, we use the four sampling methods described in Section 3 to generate random samples, and use these samples as the input to the statistical analysis. Each sample contains 20,000 random kernel partitions. Table 1 shows whether or not the statistical method could produce an estimate of the optimal performance. Each row in the table corresponds to one of the benchmarks, and each column corresponds to a different sampling method. A tick sign (✔) means that the POT method did generate an estimate. An *NA* (Not Applicable) entry means that the statistical method failed to produce any estimate.

There are two reasons why the POT method is sometimes unable to produce an estimate. First, the lower bound of the estimated minimal cost may diverge to minus infinity. Second, the iterative method that determines the confidence bounds of the estimated minimal cost (see Step 5 in Section 4.1) may not converge to a solution. In all experiments in which the POT method was not applicable, the sample mean excess plot and the quantile plot strongly suggested that the POT method could not be applied to that dataset (see Step 3 in Section 4.1).

From the results in the table, we see that the POT method using the Depth First Search (DFS) sampling method was successfully applied for only three out of eleven benchmarks. The results for the Edge Contraction (EC) method are better, but still moderate: the POT method estimated the minimum cost for seven out of eleven benchmarks. The Edge Contraction with Filter (EC-F) method was an attempt to improve load-balance over EC. However, the POT analysis could now only be applied to one of the benchmarks. We compared the costs of the random kernel partitions sampled by the EC and EC-F methods, and confirmed that the EC-F method did indeed select kernel partitions with lower cost. This was, however, not sufficient to make the samples appropriate for POT analysis. In future work, we plan to analyze this phenomenon in detail. Finally, when the POT method was applied to the uniformly distributed random samples (UD column of the table), a minimum cost was generated for all eleven benchmarks under study.

From the results presented in Table 1, we conclude that the sampling method is an important step in the analysis. All presented sampling methods select *i.i.d.* samples and fulfill the requirements of the POT statistical analysis. However, only the uniformly distributed

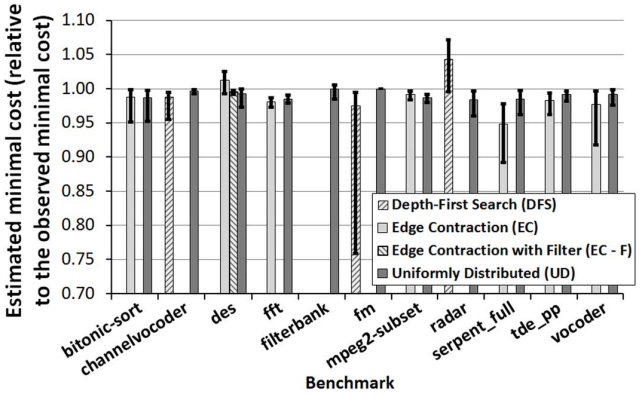


Figure 9: Estimated minimal cost

samples always led to an estimate of the cost of the optimal kernel partition. Other sampling method used to address *the same problem, for the same benchmarks, using the same statistical analysis* provided moderate (EC method) or low performance (DFS and EC-F methods).

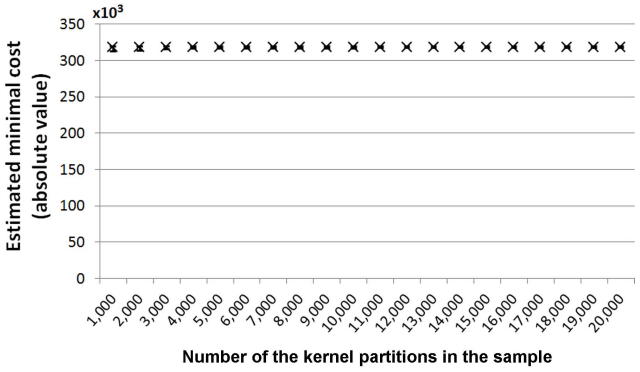
5.2. Precision of the estimation

The results that we use to analyze the precision of the estimated values are presented in Figure 9. The X-axis of the figure lists the benchmarks, while the Y-axis shows the estimated minimal cost, i.e. the estimated cost of the optimal kernel partition. The results are presented relative to the cost of the best kernel partition captured in 80,000 random kernel partitions from all four sampling methods. Kernel partitions were generated using four different sampling methods (DFS, EC, EC-F, and UD), and each method generated 20,000 random partitions.

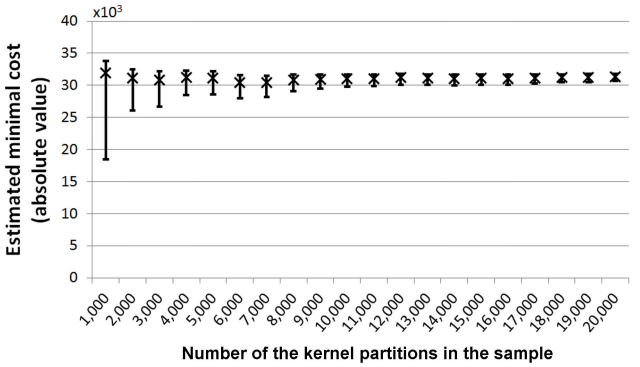
Different bars of the chart correspond to the different sampling methods used: DFS, EC, EC-F, and UD. If the POT method could not be used to estimate the minimal cost for a given benchmark and sampling method, the corresponding bar is not plotted. The height of the solid bars correspond to the point estimation of the minimal cost, while the error bars correspond to the confidence bounds for 0.95 confidence level.

High precision of the estimated minimal cost is indicated by tight confidence bounds. The width of the confidence bounds is below 10% for all bars except one (DFS sample for the *fm* benchmark). For 18 out of 22 cases, the width of the confidence bounds is below 5%.

Required number of random kernel partitions: UD method is the only sampling method that provided samples appropriate for the POT statistical analysis for all the benchmarks under study. Therefore, from this point on, we analyze only the samples that are generated with this method. In order to understand the impact of the sample size on the estimated minimal cost, we generated samples that contain between 1,000 and 20,000 random kernel partitions. For each sample, we used the POT method to estimate the minimal kernel cost. Intuitively, we expect that the POT method provides more precise estimation as the number of kernel partitions in the sample increases. In general, larger samples contain more kernel partitions in the tail that are fitted to the Generalized Pareto Distribution (GPD), and therefore the estimated GPD parameters and the minimal cost are more precise. Figure 10 shows the results for the *channelvocoder* and *serpent_full* benchmarks. In each figure, X-axis lists the number of random kernel partitions in the sample, while the Y-axis shows the



(a) channelvocoder benchmark (fastest convergence)



(b) serpent_full benchmark (slowest convergence)

Figure 10: The impact of the sample size on the estimation of the minimal cost (UD sampling method)

estimated minimal cost. The cross markers show the point estimation of the minimal cost, and the error bars correspond to the confidence bounds for the 0.95 confidence level.

For the *channelvocoder* benchmark, 1,000 random kernel partitions are sufficient to estimate the minimal cost with a high precision (see Figure 10(a)). We detect similar results for *fft, filterbank, fm, mpeg2-subset, tde-pp,* and *vocoder*. On the other hand, for the *serpent_full* benchmark, estimation based on 1,000 random kernel partitions has wide confidence bounds (see Figure 10(b)). Precise estimation of the minimal cost requires more than 8,000 random kernel partitions. The width of the confidence bounds reduces significantly as the sample increases from 1,000 to 8,000 kernel partitions. Further increment in the sample size only slightly improves the precision of the estimation. From the results shown in Figure 10(b), we also see that, as the sample size increases, the point estimation remains roughly the same and the confidence bounds converge to this value. Results for the benchmarks *bitonic-sort, des,* and *radar* follow the same trend.

Based on the presented analysis, we see that the sample size required for the precise estimation of the minimal cost significantly depends on the benchmark under study. If a user requires a minimal cost to be estimated with a given precision, we propose the following iterative method. The user can generate a small sample of random kernel partitions and estimate the minimal cost using the POT method. As long as the estimated cost does not fulfill the user's requirements, the user can increase the sample size and repeat the analysis.

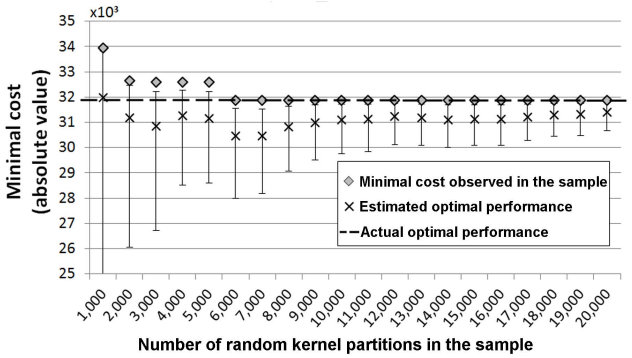


Figure 11: Comparison between the actual and the estimated kernel partition costs (serpent_full benchmark, UD sampling method)

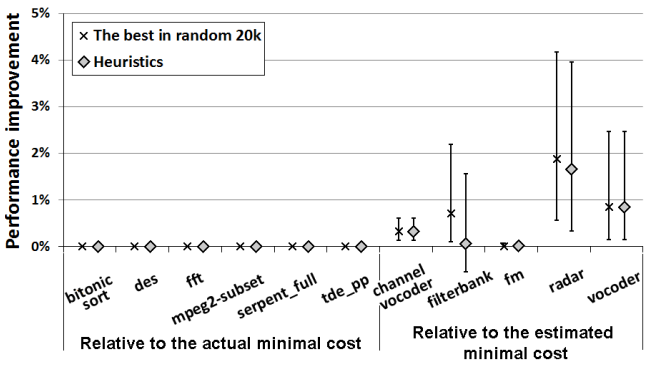


Figure 12: Comparison of random sampling (UD method) and heuristics-based algorithm

5.3. Accuracy of the estimation

In general, the kernel partitioning problem is an intractable problem. However, for *bitonic-sort, des, fft, mpeg2-subset, serpent_full,* and *tde_pp* benchmarks partitioned into exactly four software threads, brute force exploration is time consuming, but feasible. Therefore, we were able to determine the cost of all the kernel partitions, and to compare the actual and the estimated best kernel partition costs for these benchmarks. The results for the *serpent_full* benchmark are shown in Figure 11. We also analyze the estimation accuracy for different numbers of uniformly distributed random kernel partitions in the sample. The X-axis of the figure lists the size of the sample, while the Y-axis shows the absolute value of the kernel partition cost. The cross data markers show the point estimation of the minimal cost, and the error bars correspond to the confidence bounds for the 0.95 confidence level. The actual best kernel partition cost is marked with the horizontal dashed line. Finally, we also plot the minimal kernel cost observed in each random sample (diamond data markers).

First, we observe that the estimated best kernel partition cost (with confidence bounds included) is always lower than the minimal kernel cost detected in the corresponding random sample. Intuitively, this is because the statistical method estimates that the best kernel partition cost in the population (out of all possible partitions) is not higher than the minimal cost observed in the sample. We also detected that the upper confidence bound of the estimated best kernel partition cost asymptotically approaches the minimal kernel partition cost observed in the sample, as the confidence level of the estimation increases.

The estimation of the best kernel partition cost is accurate if its confidence bounds include the actual best (minimal) cost, which is satisfied for the samples that contain from 1,000 to 5,000 random kernel partitions in the Figure 11. For the samples that contain more than 6,000 kernel partitions, the presented statistical method slightly underestimates the minimal cost. This is because these samples capture a kernel partition with the best actual cost, as we explain in the previous paragraph. The underestimation is very low and it decreases with the number of kernel partitions in the sample, from 0.9% (6,000 kernel partitions) to 0.3% (20,000 partitions). The results for *bitonic-sort, des, fft, mpeg2-subset,* and *tde_pp* benchmarks follow the same trend.

Brute force exploration of the kernel partitioning problem for *channelvocoder, filterbank, fm, radar,* and *vocoder* benchmarks is infeasible. Therefore, for these benchmarks, we cannot determine the optimal kernel partition and its cost, and we cannot validate that the estimated values of the POT method were correct. However, from the results presented in Figure 9, we can detect that the estimation is incorrect if:

- The confidence bounds of different bars that correspond to the same benchmark do not overlap. This means that the POT method applied to different samples of the same benchmark estimated different minimal cost.
- The ratio between the estimated minimal cost (with confidence bounds included) and the minimal cost detected in random samples is higher than 1. This means that we detected a kernel partition with the cost that is lower than the estimated minimal cost.

From the results presented in Figure 9, we did not detect a single mispredicted cost of the optimal kernel partition.

5.4. Random sampling approach to a kernel partitioning

Our previous study [36] addresses the problem of process scheduling for modern multicore/multithreaded processors. The results presented in the study demonstrate that a random sample of several thousand random process schedules likely captures a schedule with a good performance. The study analyzes the probability that a uniformly distributed random sample of N observations contains at least one observation from the best-performing $P\%$ of the population (e.g. the best 1% of the population). This probability can be computed using the following formula: $Prob = 1 - \left(\frac{100\text{-}P}{100}\right)^N$. As P is a small positive number, the value of the fraction $\frac{100\text{-}P}{100}$ is always between 0 and 1. Therefore, for large N, the factor $\left(\frac{100\text{-}P}{100}\right)^N$ converges to 0, and the observed probability converges to 1. For example, the probability that a uniformly distributed random sample of 1,000 observations contains at least one element from the best 1% of the population exceeds 99.99%.

In order to analyze whether random sampling can be used to select a good kernel partition, we compare the cost provided by the fairly-complex heuristics-based kernel partitioning algorithm proposed by Carpenter et al. [9] with the minimal cost observed in the random sample. The sample was comprised of 20,000 kernel partitions generated using the UD sampling method. The results are shown in Figure 12. The X-axis of the figure lists different benchmarks, while the Y-axis shows the possible performance improvement of the kernel partitioning approaches. For *bitonic-sort, des, fft, mpeg2-subset, serpent_full,* and *tde_pp* benchmarks, brute force exploration is feasible, so we compare the kernel partition costs provided by the random sampling and heuristics-based algorithm with the actual minimal costs. For the remaining five benchmarks, *channelvocoder, filterbank, fm, radar,*

and *vocoder*, the results are plotted relative to the estimated minimal costs. The minimal cost is estimated using the POT method on 20,000 uniformly distributed random kernel partitions. The markers correspond to the point estimation of the minimal cost, while the error bars correspond to the estimated confidence bounds for a 0.95 confidence level.

For *bitonic-sort, des, fft, mpeg2-subset, serpent_full,* and *tde_pp* benchmarks, the best costs observed by the random sampling and the heuristics-based algorithm match the actual best costs of kernel partitions. For the *channelvocoder, filterbank, fm, radar,* and *vocoder* benchmarks, random sampling and heuristics-based algorithm, detect kernel partitions with a cost that is close to the estimated optimal one. For four out of five benchmarks (all except *radar*), the possible improvement of both approaches is below 3% (confidence bounds included). For *radar* benchmark, the estimated improvement ranges up to 4% and 4.2% for random sampling and heuristics-based algorithm, respectively. If we consider the point estimation, the estimated performance improvement is below 2% for all the benchmarks, and below 1% for four out of five benchmarks. For *channelvocoder, fm,* and *vocoder* benchmarks, the performance of the best kernel partitions in the random sample match the performance of the heuristics-based algorithm. For *filterbank* and *radar* benchmarks, the heuristics-based algorithm selected kernel partitions with 0.6% and 0.2% lower cost, respectively, which is a negligible difference. If a good heuristics-based approach is available for the applications, hardware, and metric under study, the user can choose whether to use the heuristics or the random sampling. However, it is common that heuristics-based approaches are not directly applicable to the exact situation under study. It is often difficult and time-consuming to adapt a heuristic to a particular target scenario. On the other hand, the random sampling approach is simple and easy to apply.

Required number of random kernel partitions: The formula $Prob = 1 - \left(\frac{100\text{-}P}{100}\right)^N$ can be used to compute the probability that an uniformly distributed random sample of N observations captures at least one out of P% of the kernel partitions with the lowest cost. However, as we do not know the difference in the cost in the best P% of all kernel partitions, the formula cannot be used to compute the difference between the minimal cost captured in a random sample and the actual optimal cost.

In order to analyze whether a sample of N randomly selected kernel partitions captures a good partition, we observe the minimal cost detected in the random sample and compare it with the minimal cost determined by the statistical estimation or brute force exploration, when feasible. The random samples are generated with the UD sampling method. Figure 13 shows the results of the experiments for *serpent_full* benchmark. We repeat the experiment for different sample sizes that are listed along the X-axis of the figure. Dashed vertical lines separate the results for tens (from 10 to 90), hundreds (from 100 to 900), and thousands (from 1,000 to 20,000) of random kernel partitions in the sample. The Y-axis shows the relative difference between the minimal cost captured in the random sample and the actual minimal cost determined by brute force exploration. In order to present statistically significant results, for each sample size, we randomly generate the sample 100 times and report the mean (cross marker) and the standard deviation (error bars) of the minimal cost detected in different runs.

We see that tens of random kernel partitions in the sample are unlikely to capture a good partition. We also detect a high standard deviation, which means that the cost of the best-captured kernel

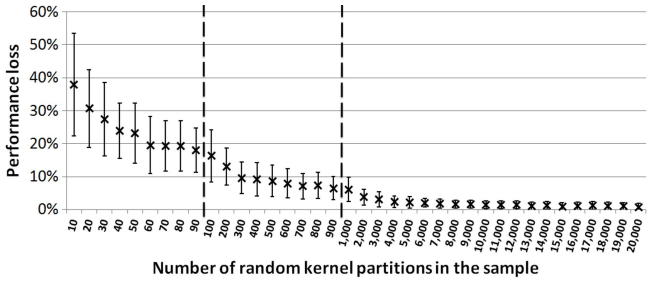


Figure 13: The impact of the sample size on the performance of the random sampling (serpent_full benchmark, UD sampling method)

partition is significantly different for different samples of the same size. For hundreds of kernel partitions in the sample, the best captured cost slowly converges to the estimated optimal one. The standard deviation decreases, which means that different samples of the same size provide similar performance. Finally, several thousand random kernel partitions capture a cost that is very close to the optimal one. Also, the detected standard deviation is low (1-2%).

We detect the same trend for all eleven benchmarks under study. Therefore, we conclude that uniformly distributed random sampling can be used to find a good kernel partition. However, we recommend this method only when the random sample contains at least several thousand kernel partitions.

In order to determine the sample size that captures a good kernel partition, a user could also observe the convergence rate of the best observed kernel partition performance in the sample as the sample increases. For example, from the Figure 13, we could observe that increasing the random sample over several thousand kernel partitions insignificantly improves the observed performance, and stop the random sampling at that point without estimation of the optimal performance. Although this approach may provide good results, it is not clear how it would avoid convergence to a local minimum of the population (e.g. see the results for 60, 70, and 80 kernel partitions in Figure 13). Also, without estimated optimal performance, we cannot determine the quality of the delivered kernel partition, i.e. we cannot provide the confidence bounds of the estimated performance improvement.

5.5. Other considerations

There are several additional aspects to consider regarding the presented statistical approach.

Experimentation time: The presented approach requires thousands of random kernel partitions to be generated and evaluated. The time to generate and evaluate 1,000 uniformly distributed random kernel partitions, for one of the StreamIt 2.1.1 benchmarks, was on average 28 minutes, using a single core of an Intel Xeon E5649 processor at 2.5GHz with 4GB memory. Running the POT method takes less than a minute. The program that generates the uniformly distributed samples is not optimized and is implemented in the Python programming language. An implementation in *C* would be much faster. Also, since different kernel partitions can be generated independently, the time to generate the sample would decrease linearly with the number of cores. The experimentation time is acceptable considering that the selected kernel partition will be compiled to the executable that can be used on numerous systems based on the same hardware during the lifetime of the system.

Scalability: The number of cores and the number of hardware threads increase in each processor generation [34]. In order to optimally use future multicore processors, kernel partitioning algorithms will have to generate a significantly larger number of threads. It is important, therefore, to analyze how these algorithms scale with the number of software threads. On the other hand, at the application level, it is reasonable to expect that the complexity of streaming applications increases leading to more complex stream graphs that comprise a larger number of kernels.

The statistical analysis that estimates the optimal kernel cost is based on the values of the performance metric, so its cost is independent of the number of threads and the complexity of the stream graph. The cost of the sampling method that generates the uniformly distributed random kernel partitions scales linearly with the number of kernels in the stream graph and with the number of output software threads. It also scales linearly with the mixing time of the partition graph [30].

Compiler optimizations and system constraints: When the program is described using a stream language, the compiler may perform complex optimizations over the stream graph; it can combine adjacent filters, split computationally intensive filters into multiple parts, or duplicate filters to have more parallel computation [8]. In order to find the set of optimizations that provides the best performance, it is important to determine good kernel partitions for different optimization sets. In this case, the proposed kernel partitioning approach does not change, but the random sampling and the presented statistical analysis are simply repeated for different optimization sets, i.e. for different stream graphs of the same program. Kernel partitioning that satisfies different system constraints, such as optimizing performance subject to memory limits, is an interesting avenue for future work.

Evaluation on real hardware: In this paper, the presented statistical approach was evaluated based on the estimates of the kernel partitions' costs provided by the StreamIt compiler. The approach was not evaluated on real hardware because of limitations in the experimental environment. The back-end of the StreamIt compiler, that we used in the study, was not capable of generating working code for different user-defined kernel partitions. As a part of future work, we plan to evaluate the presented statistical approach on real hardware. In order to do so, we intend to modify the StreamIt compiler, so it can generate the executables that correspond to any given kernel partition.

6. Related work

Several projects and studies propose different tools for compiling of streaming-like applications and their mapping onto multicore architectures.

StreamIt is a project with publicly available compiler and benchmark suite [1] . The StreamIt source language imposes a structure on the stream program graph to the compiler. The StreamIt compiler performs fully automated load balancing, communication scheduling, routing, and a set of cache optimizations [22, 23, 37]. The StreamIt compiler targets the Raw Microprocessor [40], symmetric multicore architectures, and clusters of workstations.

The Stream Graph Modulo Scheduling (SGMS) algorithm is part of StreamRoller [29], a StreamIt compiler for the Cell Architecture. This algorithm splits stateless kernels, partitions the graph, and statically schedules the software threads onto the Cell architecture. The splitting and partitioning problem is translated into an integer linear programming problem, which is solved using CPLEX Optimization Studio [27], an software package for mathematical programming.

Gedae Graph Language [33] is a proprietary GUI tool that supports the hierarchical development of data flow graphs. Gedae allows the user to specify different graph partitions and automatically maintains the data flow and connectivity of the graph. However, all the graph partition is done under user control, not by the compiler.

The Ptolemy II software environment [17] is designed to model heterogeneous embedded computing systems. Ptolemy views computing systems as a set of basic processing blocks (*actors*) that are connected using explicitly-defined communication channels. This view is very similar to the state-of-the-art interpretation of streaming-like applications. Related work from the Ptolemy project explores the more theoretical aspects of partitioning and scheduling data flow graphs for multiprocessors [25].

Liao et al. [31] present a parallel compiler for the Brook streaming language [7] with aggressive data and computation transformations. The compiler models each streaming kernel as an implicit loop nest over stream elements and uses affine partitioning to map regular programs onto multicore processors.

Farhad et al. [18] show that state-of-the-art linear programming approaches are impractical for transformations of large stream graphs to be executed on a large number of processor cores. The authors also propose an approximation algorithm for deploying stream graphs on multicore processors.

Our study shows a different approach to the kernel partitioning problem. Instead of using complex heuristics-based algorithms, we address the problem using random sampling and statistical inference. We present a statistical method that estimates performance of the optimal kernel partition based on measured performance of a sample of random partitions. We also demonstrate that random sampling can be used to find a kernel partition with performance close to the optimal one.

In our previous study [36], we use random sampling and statistical inference to analyze the optimal assignment of existing software threads onto different processor cores. There are two main contributions of this article, beyond our previous work: (1) In this article, we apply EVT to a different domain. Kernel partitioning and thread assignment are fundamentally different problems. In its essence, kernel partitioning is a graph partitioning problem, and the thread assignment problem addressed in our previous study is a multiprocessor scheduling problem [20]. (2) In this article, we also show that the sampling method has a significant effect on the applicability of the statistical method. We analyze different sampling methods, and our results strongly recommend that the samples should be uniformly distributed.

7. Conclusions

One of the greatest difficulties in using modern computing systems is how to write efficient, portable, correct software for multicore processors. A promising approach is to expose more parallelism to the compiler through domain-specific languages, enabling the compiler to perform complex high-level transformations. An important application domain comprises stream programs. A prominent step in compiling a stream program to multiple processors is kernel partitioning, which significantly affects application performance. Finding an optimal kernel partition is, however, an intractable problem.

In this paper, we proposed a statistical approach to the kernel partitioning problem. We described a method that statistically estimates, with a given confidence level, the performance of the optimal kernel partition. Knowing the optimal performance improves the evaluation of any kernel partitioning algorithm, and it is the most important piece of information for the system designer when

deciding whether an existing algorithm should be enhanced. We demonstrated that the sampling method is an important part of the analysis, and that not all methods that generate $i.i.d.$ samples provide good results. We also showed that random sampling on its own can be used to find a good kernel partition, and that it could be an alternative to heuristics-based approaches.

The presented statistical method does not depend on the application. It does not require any application profiling nor does it require the understanding of the application stream graph. The method can be applied to streaming applications with any number of kernels, and it can target any number of software threads. The presented method can analyze different metrics such as throughput, maximum hardware utilization, and minimum energy or power consumption.

We successfully applied the presented statistical analysis to the benchmarks included in the StreamIt 2.1.1 suite. The method precisely estimated the optimal kernel partition performance for all the benchmarks under study. Also, in our experiments, several hundred or several thousand random kernel partitions were enough to find a partition with close to optimal performance. The performance of the kernel partitions that were selected using random sampling matched the performance provided by the complex heuristic-based approach.

Acknowledgments

This work has been supported by the Ministry of Science and Technology of Spain under the contract TIN-2007-60625, and by the European HiPEAC-3 Network of Excellence. Also, this work has been partially supported by the Department of Universities, Research and Information Society (DURSI) of the Catalan Government (grant 2010-BE-00352). Petar Radojković holds the FPU grant (Programa Nacional de Formación de Profesorado Universitario) under contract AP2008-02370, of the Ministry of Education of Spain. Miquel Moretó is supported by an MEC/Fulbright Fellowship. The authors wish to thank to Liliana Cucu-Grosjean and Luca Santinelli from INRIA, and Jaume Abella from Barcelona Supercomputing Center for their technical support.

References

[1] "StreamIt project," http://groups.csail.mit.edu/cag/streamit/.

[2] ACOTES, "IST ACOTES Project Deliverable D2.2 Report on Streaming Programming Model and Abstract Streaming Machine Description Final Version," 2008.

[3] A. Azzalini, *Statistical Inference Based on the Likelihood*. Chapman and Hall, 1996.

[4] A. A. Balkema and L. de Haan, "Residual life time at great age," *Annals of Probability*, vol. 2, 1974.

[5] J. Beirlant *et al.*, *Statistics of Extremes: Theory and Applications*. John Wiley and Sons, Ltd, 2004.

[6] J. V. Bradley, *Distribution-Free Statistical Tests*. Prentice-Hall, 1968.

[7] I. Buck, "Brook Spec v0.2," 2003.

[8] CAG MIT, *StreamIt Language Specification, Version 2.1*, 2006.

[9] P. M. Carpenter, A. Ramirez, and E. Ayguade, "Mapping stream programs onto heterogeneous multiprocessor systems," in *Proceedings of the international conference on Compilers, Architecture, and Synthesis for Embedded Systems*, 2009.

[10] P. M. Carpenter *et al.*, "A streaming machine description and programming model," *Proceedings of the International Symposium on Systems, Architectures, Modeling and Simulation*, 2007.

[11] E. Castillo and A. Hadi, "Fitting the Generalized Pareto Distribution to data," *Journal of the American Statistical Association*, vol. 92, 1997.

[12] E. Castillo, *Extreme value theory in engineering*. Academic Press, Inc., 1988.

[13] S. J. Chapman, *Essentials of MATLAB Programming*. Cengage Learning, 2009.

[14] H. Chernoff, "On the distribution of the likelihood ratio," *Annals of Mathematical Statistics*, vol. 25, 1954.

[15] W. G. Cochran, *Sampling Techniques, 3rd edition*. Wiley-India, 2007.

[16] L. Cucu-Grosjean *et al.*, "Measurement-based probabilistic timing analysis for multi-path programs," in *Proceedings of the 2012 24th Euromicro Conference on Real-Time Systems*, 2012.

[17] J. Eker *et al.*, "Taming heterogeneity–the Ptolemy approach," *Proceedings of the IEEE*, vol. 91, no. 1, 2003.

[18] S. M. Farhad *et al.*, "Orchestration by approximation: mapping stream programs onto multicore architectures," in *Proceedings of the sixteenth international conference on Architectural Support for Programming Languages and Operating Systems*, 2011.

[19] W. Feller, *An introduction to Probability Theory and Its Applications*. John Wiley & Sons, Inc., 1971.

[20] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman and Co., 1979.

[21] M. Gilli and E. Këllezi, "An application of extreme value theory for measuring financial risk," *Computational Economics*, vol. 27, 2006.

[22] M. Gordon, W. Thies, and S. Amarasinghe, "Exploiting coarse-grained task, data, and pipeline parallelism in stream programs," in *Twelfth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2006.

[23] M. Gordon *et al.*, "A Stream Compiler for Communication-Exposed Architectures," in *Tenth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2002.

[24] S. Grimshaw, "Computing the maximum likelihood estimates for the Generalized Pareto Distribution to data," *Technometrics*, vol. 35, 1993.

[25] S. Ha and E. Lee, "Compile-time scheduling and assignment of data-flow program graphs with data-dependent iteration," *IEEE Transactions on Computers*, vol. 40, no. 11, 1991.

[26] J. R. M. Hosking and J. R. Wallis, "Parameter and quantile estimation for the generalised pareto distribution," *Technometrics*, vol. 29, 1987.

[27] ILOG, "CPLEX Math Programming Engine," http://www.ilog.com/products/cplex/.

[28] E. Këllezi and M. Gilli, "Extreme value theory for tail-related risk measures," International Center for Financial Asset Management and Engineering, FAME Research Paper Series, 2000.

[29] M. Kudlur and S. Mahlke, "Orchestrating the execution of stream programs on multicore platforms," in *Proceedings of ACM SIGPLAN Conference on Programming Language Design & Impl.*, 2008.

[30] D. Levin, Y. Peres, and E. Wilmer, *Markov chains and mixing times*. American Mathematical Society, 2009.

[31] S. Liao *et al.*, "Data and Computation Transformations for Brook Streaming Applications on Multiprocessors," in *Proceedings of the International Symposium on Code Generation and Optimization*, 2006.

[32] L. Lovász, "Random walks on graphs: A survey," *Combinatorics, Paul Erdos is Eighty*, vol. 2, no. 1, 1993.

[33] W. Lundgren, K. Barnes, and J. Steed, "Gedae: Auto Coding to a Virtual Machine," in *8th High Performance Embedded Computing Workshop*, 2004.

[34] K. Olukotun and L. Hammond, "The future of microprocessors," *Queue*, vol. 3, no. 7, Sep. 2005.

[35] J. I. Pickands, "Statistical inference using extreme value order statistics," *Annals of Statististics*, vol. 3, 1975.

[36] P. Radojković *et al.*, "Optimal task assignment in multithreaded processors: A statistical approach," in *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*, 2012.

[37] J. Sermulins *et al.*, "Cache aware optimization of stream programs," in *Proceedings of the 2005 ACM SIGPLAN/SIGBED conference on Languages, Compilers, and Tools for Embedded Systems*, 2005.

[38] N. Tajvidi, "Design and implementation of statistical computations for Generalized Pareto Distributions," *Technical Report, Chalmers University of Technology*, 1996.

[39] W. Thies, M. Karczmarek, and S. Amarasinghe, "StreamIt: A Language for Streaming Applications," *International Conference on Compiler Construction*, vol. 4, 2002.

[40] E. Waingold *et al.*, "Baring It All to Software: Raw Machines," *Computer*, 1997.

[41] S. S. Wilks, "The large-sample distribution of the likelihood ratio for testing composite hypotheses," *Annals of Mathematical Statistics*, vol. 9, 1938.

[42] S. S. Wilks, *Mathematical Statistics*. Princeton University, 1943.

2012 IEEE/ACM 45th Annual International Symposium on Microarchitecture

Inferred Models for Dynamic and Sparse Hardware-Software Spaces

Weidan Wu Benjamin C. Lee
Duke University
{weidan.wu, benjamin.c.lee}@duke.edu

Abstract

Diverse software and heterogeneous hardware pose new challenges in systems and architecture management. Managers benefit from improving introspective capabilities, which provide data across a spectrum of platforms, from software and datacenter profilers to performance counters and canary circuits. Despite this wealth of data, management has become more difficult as sophisticated decisions are demanded.

To address these challenges, we present modeling strategies for integrated hardware-software analysis. These strategies include (i) identifying shared software behavior; (ii) quantifying that behavior in a portable, microarchitecture-independent manner; (iii) inferring generalized trends with statistical regression models; and (iv) automatically constructing/updating these models as new software profiles are obtained.

Models produced by these strategies are accurate for general SPEC2006 applications with median errors of 8-10%. Predicted and actual performance are strongly correlated with coefficients of $\rho > 0.9$. Moreover, when we exploit application semantics and domain-specific software parameters, model accuracy improves and model complexity falls. In a case study for sparse linear algebra, we present models with 5% median error and new capabilities in coordinated hardware-software tuning.

1. Introduction

Hardware management is the art of linking data to decisions. But forging this link is increasingly difficult. Across a spectrum of computing platforms, decisions must be made with increasingly sparse data. Not every node can be profiled yet datacenter managers must navigate diverse hardware-software interactions. Not every configuration can be profiled yet adaptive chips must navigate performance and power trade-offs.

Our ability to collect data has advanced significantly at all scales, from software and datacenter profilers to performance counters and canary circuits. But even as introspection has supplied more data, the decisions demanded of hardware managers have become more complicated. Datacenters must allocate and schedule software while navigating heterogeneity and contention. Architectures must adapt operating parameters and structural resources to dynamic application behavior.

All of these decisions require linking data to expectations of performance and closing the data-to-decision gap. But because many of these decisions involve diverse software on heterogeneous hardware, prior efforts that separate application analysis and architectural optimization are insufficient. Instead, we must reason about software, hardware, and their interactions in a coordinated fashion.

To integrate hardware-software analysis yet keep costs tractable, we present strategies to share profiled behavior (§2). First, we break an application into shards and profile microarchitecture-independent measures of behavior. Given profiles, we construct models that predict performance as a function of software behavior and hardware parameters. Finally, because the number of parameters explodes in an integrated hardware-software space, an automated heuristic constructs the model.

We demonstrate this strategy in two settings. In the general and more difficult setting, we measure detailed software behavior for arbitrary applications and infer a model. Models in the second, domain-specific setting exploits programmer-level knobs in tunable codes. In both settings, we link hardware-software interactions to performance. Thus, we make the following contributions:

- Laying a foundation for system management, we construct predictive models for hardware-software interactions. But software behavior is dynamic, exhibits high variance, and introduces an unwieldy number of parameters. We present a heuristic that automatically builds and updates regression models as software behavior is profiled. (§3)
- Inferred models interpolate and extrapolate performance for diverse hardware-software interactions. Median errors are 8-10%. Predicted and actual performance are strongly correlated with coefficients of $\rho > 0.9$. (§4)
- Given domain knowledge, software behavior is captured more concisely. Domain-specific software parameters produce smaller, more accurate models. In a case study for sparse linear algebra, we show accurate models for highly irregular performance topologies and demonstrate their application to coordinated hardware-software tuning. (§5)

Collectively, these results lay the foundation for understanding diverse software on heterogeneous hardware. By linking sparse data to performance predictions, we enable future work in control mechanisms for reconfigurable architectures and allocation mechanisms for heterogeneous datacenters.

2. Sharing – Principles and Strategies

Capturing hardware-software performance is made difficult by highly variable software behavior. To address this challenge, we infer shared behavior and construct integrated models with four strategies.

2.1. Shard-level Profiles

Models are most effective when inferred from diverse data. To increase diversity, we break an application into shards, each with an equal number of instructions. When collected for short shards, profiles detect fine-grained phase behavior. In contrast, monolithic application profiles (e.g., average instruction mix) obscure intra-application diversity.

This diversity is needed when sharing profiles between applications. Suppose we have profiled several applications and are given a new one. Profiles of monolithic application behavior are useful only if the new application resembles a previously observed one. This constraint is too restrictive to meaningfully share profiled insights.

Relaxing this similarity constraint, fine-grained shards in the new application may resemble disparate shards from others. As illustrated in Figure 1, profiles from relevant shards can be drawn from several applications to capture partial similarities. Sharding increases the

1072-4451/12 $26.00 © 2012 IEEE
DOI 10.1109/MICRO.2012.45

IEEE computer society

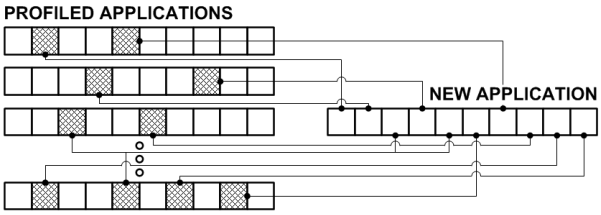


Figure 1: Reflecting partial similarity, a new application might be understood in terms of shards drawn from several previously profiled applications.

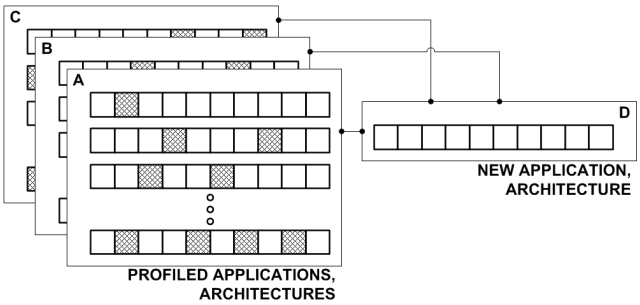


Figure 2: Portable, microarchitecture-independent characteristics that are profiled for architectures A, B, and C are applicable to new architecture D.

value of an application's profile since part of it is likely relevant for other applications of interest.

In contrast to related work [35], our approach to sharding is agnostic to underlying phase behavior. We simply ensure that shards are shorter than phases so that intra-application diversity is preserved. A short, pre-determined shard length is sufficient. Forgoing sophisticated phase analysis simplifies the mechanics of profiling software characteristics.

2.2. Portable Characteristics

We measure application characteristics that are portable and independent of the microarchitecture [14, 36]. For example, data re-use distance is a portable measure while cache miss rate is not. Such portability is important when profiling tunable codes on reconfigurable cache architectures. We cannot profile every code on every cache configuration [43].

At the system-level, portability is needed to navigate increasing heterogeneity. The Google-wide Profiler samples application behavior across many datacenter nodes [39] . And future node managers must anticipate heterogeneous hardware demand in the form of diverse resource containers [2, 17, 19], contention [30], or big/small cores [38]. Thus, software profiles will be collected from increasingly diverse platforms.

Portable measures of software are needed when sharing profiles between architectures. Figure 2 shows how profiles collected for different shards and architectures might provide insight for a new application on a new architecture. While these links might be found explicitly, perhaps with distance calculations and clustering, the costs of doing so are prohibitive given the number of dimensions in an integrated hardware-software space. Implicitly inferring these links is more tractable.

2.3. Statistical Inference

To infer these links, we extend related work in predictive modeling. Previous models predict performance as a function of parameters from the processor design space. Sampled measurements from the space are used to train neural networks [11, 21, 22] or fit regression models [26, 27]. These efforts infer hardware performance for design space exploration.

In this paper, inferred performance also accounts for software behavior and lays a foundation for run-time decisions. Let z be performance. And let $x = (x_1, \ldots, x_p)$ and $y = (y_1, \ldots, y_q)$ be sets of p hardware parameters and q software characteristics. By sparsely profiling hardware-software interactions, a model can be inferred to predict $z = F(x,y) + \varepsilon$ with some approximation error ε. With statistical regression, we construct an integrated hardware-software model.

2.4. Automated Modeling

Unfortunately, an integrated hardware-software space is unwieldy. To infer a model, we must navigate a space of hardware and software parameters, several non-linear transformations on them, and a combinatorially increasing number of interactions between them. In prior work, users manually specified hardware-only models. But specifying hardware-software models is complicated by the sheer number of variables and requires additional guidance.

We present a heuristic to search for effective model specifications, which are defined by variables, transformations, and interactions. We encode model specifications as genetic sequences, which evolve toward better fits. Unlike stepwise regression, which considers only one term at a time, crossovers and mutation in genetic algorithms support a rapid search of possible models. Moreover, the heuristic accommodates new data by updating the model specification and fitting new regression coefficients. This capability supports dynamic run-time environments with evolving software profiles.

3. Generalized Hardware-Software Models

Accommodating dynamic software behavior is difficult for two reasons. First, software characteristics have high variance and long tails (i.e., infrequent instances of large values), which models have difficulty capturing. Second, profilers have little control over the software behavior used for training.

Nonetheless, modeling software behavior is critically important. Behavior differs dramatically between application phases and between different applications. These differences are exploited by many of the most innovative heterogeneous systems and adaptive architectures. The viability of these innovations depends on linking software dynamics to system and architecture preferences.

Consider a system with diverse software that requests computation from heterogeneous hardware. The space of software behavior is large. Moreover, it is sparsely and non-uniformly populated by real applications. As applications run, models have the difficult task of generalizing trends by profiling software behavior that it cannot precisely sample and manipulate. In comparison, hardware modeling is easier since simulators allow architects to sample uniformly from a cleanly defined design space.

3.1. Inference and Software Behavior

Regression is the starting point for our models. Suppose we have independent variables $x = (x_1, \ldots, x_p)$ for software and $y = (y_1, \ldots, y_q)$

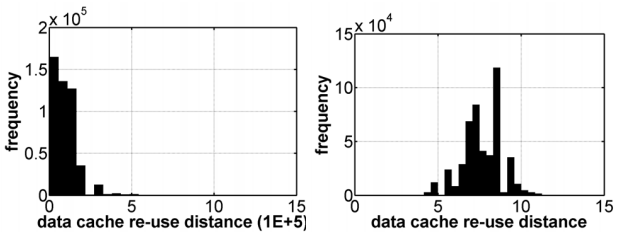


Figure 3: Each shard reports the sum of its re-use distances for 256B data cache blocks. (a) Histogram for this sum-of-distances (x) shows long tail of outliers across SPEC2006 shards. (b) Transforming $x \rightarrow x^{1/5}$ stabilizes variance.

for hardware. And we have a dependent variable z for performance. A basic regression fits $z = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \ldots + \beta_{p+q} y_q + \varepsilon$ by finding β's to minimize error given x, y, and z from training data.

In practice, more sophisticated models are needed. Users must determine which independent variables to include (x_i or y_j), how these variables are transformed to accommodate non-linear trends ($S(x_i)$), and which variables interact to affect performance ($x_i y_j$). In each of these decisions, software characteristics introduce new challenges that we address with an automated heuristic for constructing and updating models.

Choosing Variables. In some cases, the impact of software behavior is clear from domain knowledge. For example, rare floating-point divides are not strong predictors of performance. But more generally, we cannot always anticipate the precise mix of applications in a system and determining the best software predictors of performance is complicated.

Further affecting the choice of variables are strong relationships between software characteristics. For example, consider measures of locality. Temporal locality measures time between two consecutive accesses to a cache block. And spatial locality is the quotient of two measures for temporal locality at different block sizes. From an architect's perspective, this link is clear and both locality measures should be modeled.

However, from a statistician's perspective, these locality measures are linearly dependent and highly correlated. Such subtle collinearity, which prevents solvers from fitting a model, is common amongst software variables. Although we use domain expertise to eliminate obvious cases of collinearity, many are not easily discovered until model construction. For this reason, the modeling heuristic must also check for and eliminate collinear variables as it dynamically and automatically seeks a mix of variables with high predictive ability.

Transforming Variables. Once chosen for the model, variables benefit from non-linear transformations that provide flexibility and mitigate the high variance in software behavior, which is particularly important since we cannot explicitly control training samples from the space of software behavior.

To illustrate challenges posed by behavioral asymmetry, Figure 3(a) plots a histogram of temporal locality for SPEC2006 shards. We measure a shard's locality as the sum of all re-use distances within it.[1] While most profiles report small sum-of-distances, many profiles report much larger ones. Outliers are an order of magnitude larger than the common case; sum-of-distances at 5E+4 are most common but sum-of-distances at 5E+5 are observed. Such tails are typical in software.

Because this heteroscedasticity (i.e., non-constant variance) breaks underlying regression assumptions, we apply variance stabilizing transformations. Rather than use x, we use $x^{1/n}$ in the model.[2] With such a transformation, our measure of locality exhibits more symmetry and less variance around the mean as shown in Figure 3(b).

Transformations also provide flexibility to capture non-linear or non-monotonic trends. We can use splines that divide a variable into pieces and fit different cubic polynomials to each piece [18]. For example, $S(x) = \beta_0 + \beta_1 x + \beta_2 x^2 + \beta_3 x^3 + \beta_4 (x-a)_+^3 + \beta_5 (x-b)_+^3 + \beta_6 (x-c)_+^3$ splits x into pieces delimited by three inflections at a, b, and c. Since $(x-a)_+ = max(x-a,\ 0)$, β_4 has an effect only if x is greater than a. In this way, different coefficients are fit to different parts of the space.

These transformations require decisions, such as the exponent in variance stabilization or the number of inflections in splines. The right choice depends on the training data. But since this data evolves in dynamic systems, our modeling heuristic searches the space of transformations and applies the best one automatically.

Specifying Interactions. Lastly, variables interact to affect the predicted value. For example, the performance impact of branch prediction is larger in deeper pipelines because the price of wrong-path execution is higher. Pairwise hardware-software interactions are particularly important since they are a fundamental determinant of performance.

Regression captures such interactions with a product term. In $z = \beta_0 + \beta_1 x + \beta_2 y + \beta_3 xy + \varepsilon$, the interaction between x and y is captured by coefficient β_3. Note the partial derivative $\delta z / \delta x_1 = \beta_1 + \beta_3 x_2$. Given p variables and t possible transformations on those variables, there are $\binom{p \times t}{2}$ possible pairwise interactions. These possibilities are explored and evaluated by the modeling heuristic.

3.2. Accommodating System Dynamics

Training models in dynamic systems is difficult. Because we add software behavior to the model and rely on real applications for sparse profiles, we have little control over training data. Some software behavior may be well represented while others are not. And, in systems with run-time profiling, new software behavior requires model updates.

We describe the modeling process for a large system with diverse architectures and applications. As software runs, sparse run-time profiling collects hardware-software interactions and their effect on performance (e.g., Google-wide Profiler [39]). And in the future, these profiles may be collected on heterogeneous platforms with big/small cores [38], differentiated virtual machines [2], or diverse servers in federated clouds. As data accrues, the model is updated.

An Inductive Analysis. To describe system and model dynamics, we take an inductive approach. In the inductive hypothesis, the system is in steady state. Architectures and applications from spaces H and S have been sparsely profiled. And this data has trained an accurate model M for the integrated H/S space. In practice, this hypothesis holds because models can be boot-strapped with data from benchmark suites.

In the inductive step, the system is perturbed by a new architecture or application.[3] Suppose software space S is perturbed by a new application $+s$. Since the application runs on at least one architecture, there exists a profile with microarchitecture-independent software

[1] Re-use distance is the number of instructions separating two consecutive accesses to the same data block.

[2] $n \geq 1$ and statistics packages, like `ladder` in Stata, help identify the best power transformation.

[3] We use the words "architecture" to represent a hardware environment. A new architecture could arise from new hardware, virtual machines, or contention conditions. Similarly, a new application could arise from new jobs, input data, or code optimizations.

behavior x_{+s}, hardware parameters y, and performance z. With this data, we check the existing model's accuracy, comparing measured performance z against a prediction $M(x_{+s}, y)$.

If predicted performance is accurate, the new application likely shares behavior with already observed software. A sufficiently accurate model will have errors for $+s$ that are competitive with those for applications in S. And, in practice, the desired accuracy depends on how predictions are used. For example, median errors less than 10-15% may be sufficient to make coarse-grained resource allocations.

Input: Profiles P_S
Output: Regression Model M
foreach *Generation* $g \leq G$ **do**
  foreach *Model* $m \in g$ **do**
    foreach *Software* $s \in S$ **do**
      Split P_s data into training T_s, validation V_s
      Fit m using $\{P_{-s}, T_s\} \times w$
      Set software fitness f_s as m's accuracy on V_s
    end
    Set model fitness f_m as $(\sum_{s \in S} f_s)/|S|$
  end
  Populate N% of $g+1$ with g's N% best models
  Populate (100-N)% of $g+1$ with crossovers, mutations
end

3.3. Updating System Models

An inaccurate prediction may suggest that the new application is poorly served by existing regression coefficients and/or model specifications. However, the error could also be an outlier. To determine whether to trigger a model update, more data is needed. Profiling $+s$, or variants of it, on a few more architectures would provide additional insight. In practice, we find 10-20 additional data points are sufficient.

Exactly when additional profiles are obtained determines model responsiveness. A model might be updated immediately by invoking profilers for $+s$ on various hardware. Alternatively, because large systems invoke profilers periodically and selectively, a model might be updated only after a sufficient number of additional profiles has accrued. This latter scenario would introduce hysteresis into system models.

To update the model, we insert the new application and its profiles into S. With profiles P_s for each application $s \in S$, we invoke a heuristic to re-specify and perform a weighted fit of the model. This heuristic chooses new variables, transformations, and interactions. To ensure model updates accommodate all profiled applications, the inner loop evaluates the fit of a candidate model for every application ($f_s, s \in S$), which then determines average model fitness (f_m).

3.4. Genetic Search

The heuristic's outer loops implement a genetic search, in which the best models propagate into the next generation while the others are subject to crossovers and mutations. Each model is described by a chromosome that encodes variables, transformations, and interactions.

Each gene encodes a variable. If the genetic value for x_i is 0, the variable is excluded. If the value is 1, 2, or 3, we add x_i with a linear, quadratic, or cubic transformation. And if the genetic value is 4, we apply a piecewise-cubic transformation with three inflection points.

Instruction Mix	
x_1	# Control
x_2	# Taken Branches
x_3	# Float ALU
x_4	# Float Mul/Div
x_5	# Integer Mul/Div
x_6	# Integer ALU
x_7	# Memory
Memory – Temporal Locality	
x_8	average re-use distance for 64B d-cache blocks
x_9	average re-use distance for 64B i-cache blocks
Instruction-Level Parallelism	
x_{10}	# of instructions between floating-point ALU and its consumer
x_{11}	# of instructions between floating-point multiply and its consumer
x_{12}	# of instructions between integer multiply and its consumer
x_{13}	Average basic block size # instructions / # branches

Table 1: Software characteristics that are microarchitecture-independent and profiled for 10M-instruction shards.

Pipeline Parameter		
y_1	Width	1 :: 2x :: 8
y_2	Load/Store queue Physical registers Instruction queue Reorder buffer	11 :: 5+ :: 38 86 :: 42+ :: 300 22 :: 10+ :: 72 64 :: 32+ :: 224
Cache		
y_3	L1 Associativity L2 Associativity	1 :: 2x :: 8 2 :: 2x :: 8
y_4	MSHR	1 2 4 6 8
y_5	Data cache size (KB)	16 :: 2x :: 128
y_6	Instruction cache size (KB)	16 :: 2x :: 128
y_7	L2 cache size (KB)	256 :: 2x :: 4096
y_8	L2 latency (cy)	6 :: 2+ :: 14
Functional Unit Number		
y_9	Integer ALU	1 :: 1+ :: 4
y_{10}	Integer Mult/Div	1, 2
y_{11}	Float ALU	1 :: 1+ :: 3
y_{12}	Float Mult	1, 2
y_{13}	Cache Read/Write Port	1 :: 1+ :: 4

Table 2: Hardware parameters that include extreme designs so that models infer interior points more accurately.

The chromosome also encodes interactions, specifying a pair of numbers $i-j$ for interaction $x_i x_j$. Given p variables and t transformations on them, $\binom{p \times t}{2}$ interactions are possible interactions. Because we cannot statically specify a chromosome long enough to accommodate so many interactions, we dynamically expand/shrink its length as the search runs.

The genetic search starts with a random population of models, which evolves with crossovers and mutations. With evolving chromosomes, the heuristic quickly covers a large space of model specifications. Models may be affected by three crossover operators: (C1) single variable randomly exchanged between two chromosomes, (C2) interaction randomly exchanged between two chromosomes, (C3) interaction randomly created using single variables from two chromosomes.

We further consider two mutation operators: (M1) interaction randomly changed for a chromosome, (M2) single variable randomly

	Software Parameters	Hardware Parameters
un-used		y_{12}
linear	x_6, x_8, x_9	y_3, y_4, y_8, y_{10}
poly, degree 2	$x_1, x_4, x_5, x_7, x_{10}$ x_{11}, x_{12}, x_{13}	y_1, y_6
spline, 3 knots	x_2, x_3, x_6	$y_2, y_7, y_9, y_{11}, y_{13}$

Table 3: Transformations after 20 genetic search generations.

changed for a chromosome. Each crossover occurs with 12.5% probability and each mutation occurs with 5% probability, which we find experimentally effective.

4. Evaluating Generalized Models

We evaluate the effectiveness of the modeling heuristic, illustrating its convergence and describing the nature of the produced model. We further demonstrate accuracy, both in steady state and after updates, for integrated hardware-software performance prediction.

4.1. Experimental Methodology

To define an integrated hardware-software space, we consider a spectrum of microarchitectures and key measures of software behavior. We sparsely sample application-architecture profiles to train models.

Software Parameters. We break SPEC2006 applications into shards of 10M dynamic instructions. For each shard, we profile portable measures of software behavior as listed in Table 1. In the datapath, these characteristics capture instruction mix. They also capture instruction-level parallelism via the number of instructions that separate producer and consumer instructions. In the cache hierarchy, the profile captures locality by measuring the number of instructions separating two consecutive accesses to the same data block [40].

These characteristics primarily capture processor-bound workload behavior. Other workloads may require memory or I/O characteristics. For memory-bound workloads, such parameters might include memory hierarchy latencies, memory channel bandwidth, application concurrency, and memory request burstiness. Similar strategies apply for I/O-bound workloads.

Hardware Parameters. Applications are profiled on the diverse microarchitectures listed in Table 2. Such hardware diversity may manifest physically in an implemented design, or manifest logically during run-time as partitioning schemes or contention for shared resources. To collect profiles, these microarchitectures must support introspective performance counters.

We embed such counters into Gem5 [1], extending the simulator to profile software behavior during the commit pipeline stage, which ensures that software behavior is independent of the out-of-order microarchitecture. Gem5 simulates the Alpha instruction set and we cross-compiled the following SPEC2006 applications: *astar, bwaves, bzip2, gemsFDTD, hmmer, omnetpp, sjeng*.

Just as large system profilers selectively profile hardware-software pairs [39], we sample the integrated hardware-software space to produce profiles. With profiled data and R statistics libraries, the modeling heuristic fits a regression model [18]. For performance, we parallelize the genetic search with R libraries `doMC` and `Multicore`, which automatically fork and join R threads.

4.2. Automated Modeling

An initial collection of random models may produce a few with reasonable accuracy. And as these models evolve toward better specifications, errors fall. In practice, useful models begin appearing

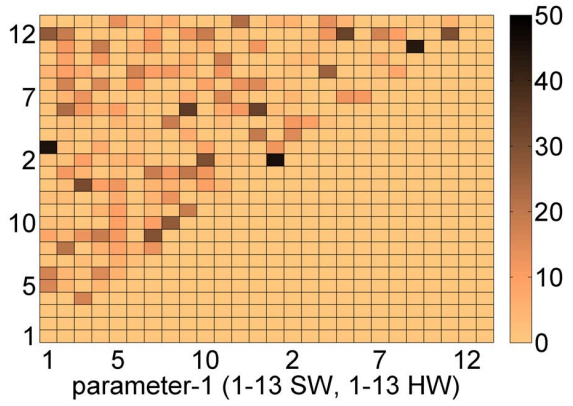


Figure 4: Frequency of interactions in the 50 best models after 20 generations of a genetic search. Interactions are shown between software parameters (lower-left), software-hardware parameters (upper-left), and hardware parameters (upper-right). Matrix is symmetric and we show upper triangle without loss of generality.

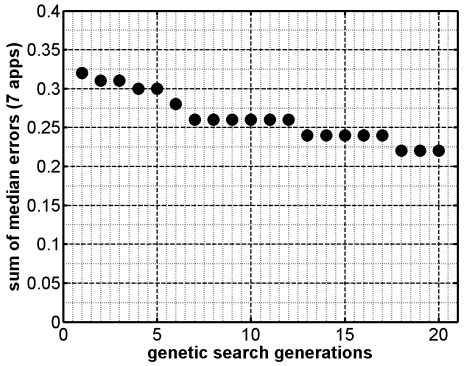


Figure 5: Accuracy improves genetic algorithm evolves for 20 generations. Median errors summed for 7 applications used by a genetic algorithm.

after only a few generations. We see diminishing marginal benefits to accuracy as the search approaches 20 generations. Figure 5 illustrates the benefit to our heuristic's measure of accuracy: sum of median errors for applications of interest.

Comparison with Manual Modeling. A research assistant with no prior experience in regression requires nearly ten months to produce an integrated hardware-software model by hand. Much of this time is spent identifying variables, transformations and interactions.

Moreover, a manually specified model is susceptible to human biases, which limit the number of candidate models he considers. Automatically derived models benefit from a comprehensive search. We find that model errors from genetic search are 10% lower than those from hand-tuning.

Modeling Time. The speed of parallelized and automated genetic search significantly reduces the time to find and fit an accurate model. As long as the heuristic's three nested loops are ordered to minimize data movement, we find that twelve processor cores provide a 9× speedup.

Moreover, the genetic algorithm's inner loop is embarrassingly parallel. Statistical computation for each candidate model in a gener-

ation is independent; a generation with n models could benefit from n-way parallelism.

With such speed, the search can evaluate one generation every 20 minutes. In practice, we expect even faster training times. As the search begins with more effective models in the starting population, fewer generations are required. And each generation is evaluated more quickly as more cores provide greater parallelism.

Parameter Significance. System analysts benefit, not only from speed and accuracy, but also from an additional source of insight as the genetic search identifies determinants of performance. The search begins with a population of random models. But as models evolve, the population increasingly prefers certain variables, transformations, and interactions.

Some parameters, such as those that support out-of-order execution (y_2), have complex relationships with performance that require sophisticated spline transformations. Other parameters, such as the number of floating-point multiplies (y_{12}) is less significant and dropped from the model. Consider a genetic algorithm evolved for more than 20 generations using SPEC 2006 shard profiles. After the search converges, we examine the best models and Table 3 presents common transformations.

In practice, hardware-software interactions are sophisticated and span many different parameters. However, the model accommodates only pairwise interactions. The genetic search must use many such pairs to capture the desired effect. Specifically, the two-dimensional histogram of Figure 4 indicates how often a particular pairwise interaction appears in the 50 best models. After 20 generations, the best models still exhibit considerable diversity in its choice of pairwise interactions.

4.3. Accuracy in Steady State – Interpolation

We evaluate model accuracy in two scenarios. In the first, the system is in steady state. An integrated hardware-software space has been sparsely profiled to construct a model that interpolates performance. As illustrated in Figure 6(a), interpolation lends itself to accurate models. For every prediction made, we have likely profiled similar hardware for another application or profiled similar software for another architecture.

Interpolation Accuracy. To evaluate interpolation accuracy, we train and validate a model. First, we randomly sample architectures. For each architecture, we randomly sample applications. The number of samples is many orders of magnitude smaller than the cross-product of applications and architectures. With these sparse samples, the automated heuristic produces a model. On average, each of 7 applications is profiled on 360 architectures.

We assess accuracy in two ways. First, we examine the distribution of prediction errors with boxplots, which show the median and quartiles computed over the validation data. Second, we consider the correlation between predicted and true performance, which is a better measure of accuracy in the context of optimization. For example, correlation is important in hill climbing heuristics that use models to find higher performance.

Training data is randomly selected from application-architecture pairs. Validation data is select randomly and independently of the training data. Validation against 140 separately profiled application-architecture pairs illustrates the effectiveness of our automated modeling heuristic. The resulting model has low median errors of 5% in Figure 7(a) and high correlation coefficients of $\rho > 0.9$ in Figure 8(a).

Reduced Profiling Costs. Not only is the integrated hardware-software model accurate, it requires less data to train when compared against prior approaches in regression and neural networks [21, 26]. Previously, each application would require its own architectural model and 400-800 architectural profiles to train it.

With our integrated approach, we require fewer architectural samples per application to construct a single model shared by all applications. Shared software behavior reduces the number of required profiles. If applications s_1 and s_2 exhibit similar software behavior, each benefits from the other's architectural profiles. By exploiting such shared behavior, profiling costs per application falls by $2-4\times$. Cost reductions are even greater (20-40$\times$) when existing profiles are used to extrapolate new application or architecture performance.

4.4. Accuracy after Updates – Extrapolation

Shared shard behavior is the basis for extrapolation and the second modeling scenario. In this scenario, the system is perturbed by a new architecture, new application, or both. By exploiting similar behavior in existing profiles, models can be updated inexpensively to extrapolate performance as illustrated in Figure 6(b-d).

Extrapolation for Shards. We first evaluate the notion of shard similarity by extrapolating individual shard performance. Profiles of shards from $n-1$ applications train a model, which is used to predict the performance of shards from application n. Each SPEC2006 application takes a turn as application n; the other $n-1$ applications train.

Accurate shard-level predictions indicate exploitable relationships across application shards. For example, *astar* shard performance is predicted accurately by sparse shard profiles from $\{bwaves, \ldots, sjeng\}$. We validate against 300 separately profiled shards for each application. Figure 10 shows low median errors of 8%. Moreover, predictions correlate with true performance values; $\rho \geq 0.9$.

Extrapolation for Applications/Architectures. Automatic model updates further enhance accuracy when the system is perturbed by a new application and/or architecture. As illustrated in Figure 6(d), a system in steady state has sampled shard profiles from $n-1$ applications on diverse architectures. These profiles produce an accurate hardware-software model for interpolation. When perturbed by application n, the model is updated (§3.2–§3.4).

To predict application performance, we predict the performance of its constituent shards and aggregate their contributions to the application. The performance for most shards can be extrapolated accurately. A few inaccurate shard predictions have a small effect on the end-to-end prediction since an application contains many 10M instruction shards.

We first consider systems perturbed by variants of existing applications, which perform the same fundamental computation but differ in code structure or input data. These differences alter the dynamic instruction stream, significantly affecting both performance and the underlying microarchitecture-independent characteristics we profile. For example, we find the choice of back-end compiler optimizations affect performance by up to 60%; mean effect is 26%.

Consider a system perturbed by applications with code optimizations (`-O1`,`-O3`) or input data (`-v1`,`-v2`,`-v3`) that differ from those in existing profiles. For these software variants, updated models accurately predict performance for 150 application-architecture pairs. Median errors are 8% in Figure 7(b). Correlation coefficients $\rho \geq 0.9$ in Figure 8(b).

Beyond the common perturbations of varying software, systems may also encounter fundamentally new software. In such scenarios,

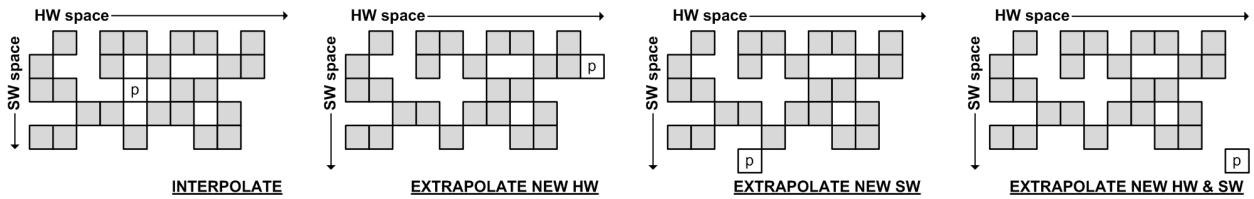


Figure 6: Shaded HW-SW pairs are profiled. Pair p denotes a prediction. (a) Interpolation for previously observed hardware and software. (b)-(d) Extrapolation after updates for new hardware, new software, or both.

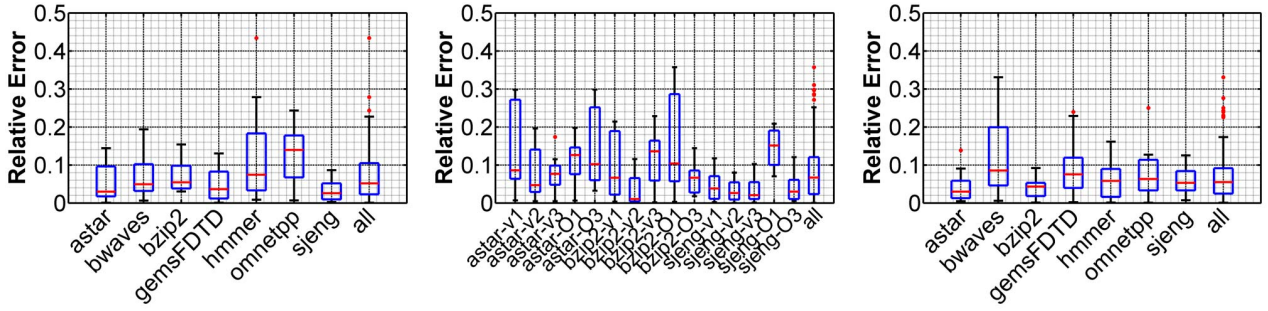


Figure 7: Distributions of performance prediction error when (a) interpolating in steady state, (b) extrapolating for new software variant, (c) extrapolating for new hardware/software.

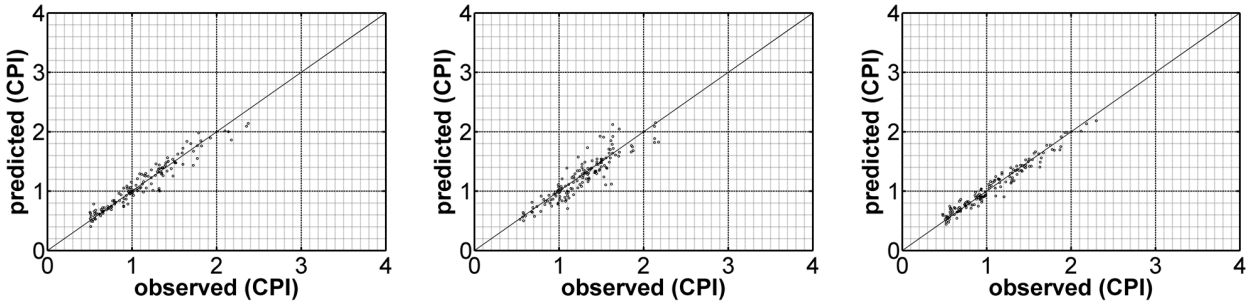


Figure 8: Correlation between predicted and true performance when (a) interpolating in steady state, (b) extrapolating for new software variant, (c) extrapolating for new hardware/software.

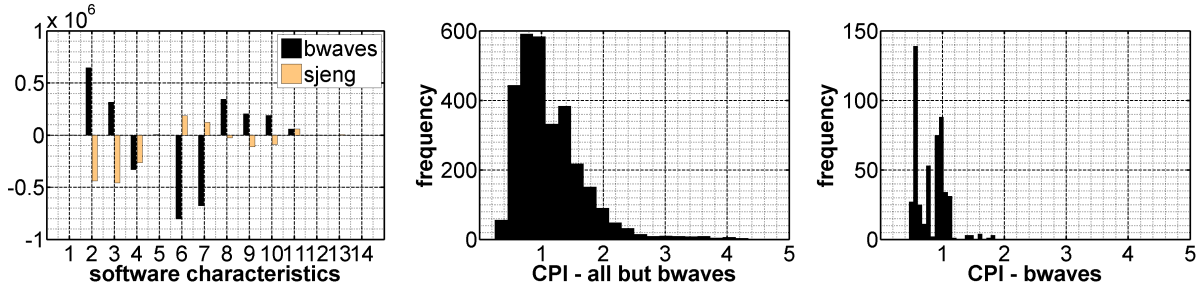


Figure 9: Extrapolation for bwaves suffers from significant differences in software behavior and performance. (a) illustrates the difference between training data mean and bwaves/sjeng mean for various software characteristics; x-axis refers to Table 2. (b) plots CPI distribution for all applications excluding bwaves and (c) plots a very different CPI distribution for bwaves.

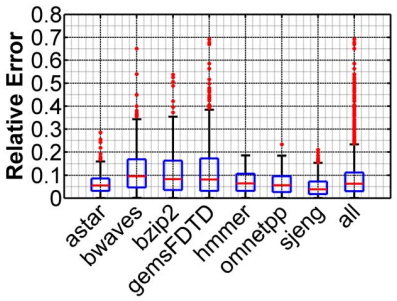


Figure 10: Error distribution when predicting shard performance. Model is trained and validated with separate, randomly sampled shards.

extrapolation reaches further beyond the already profiled space. To assess accuracy, each SPEC2006 application takes a turn as application n; the other $n-1$ train.

We predict the performance of application n for 140 new application-architecture pairs. While inherently more difficult than interpolation, extrapolation with updated models captures performance trends with low median errors of 6% in Figure 7(b) and strong correlations between predicted and true values; $\rho \geq 0.9$ in Figure 8(b).

4.5. Outliers

Extrapolation is more difficult when already profiled software behavior does not cover those in the target application. For example, compare performance extrapolation for *sjeng* and *bwaves*. Software behavior in *sjeng* is very similar to that in the $n-1$ other applications. In contrast, *bwaves* exhibits very different behavior.

Figure 9 quantifies this difference. For each software characteristic of Table 1, we take the mean value observed for a given application and subtract the mean value observed for its training applications. If this difference is zero, application n behaves like the other $n-1$ applications. However, if this difference is large, the training data is not representative of the target application.

Figure 9(a) indicates that, while *sjeng* differences are modest, *bwaves* is not well represented by its training data. Compared to training applications, *bwaves* has far more taken branches and floating-point operations. And it has far fewer integer and memory operations.

Such large behavioral differences translate into performance differences. Figure 9(b-c) show CPI histograms for *bwaves*' shards versus those in all the other applications. While performance of other applications' shards are clustered around CPI=1, *bwaves* CPI exhibits much greater variance and bimodal behavior around CPI = 0.5 and 1.0. Even model updates cannot accommodate this difference in performance distribution.

However, these challenges are not fundamental and, in an avenue for future work, training data can be augmented to better cover the space of software behavior. Synthetic benchmarks provide explicit control on software behavior and enable uniform profiling across the software space [23]. If synthetic benchmarks were used, they would need to be coordinated with real application profiles.

5. Domain-Specific Models

To model performance from instruction-level software behavior, §3–§4 develop an extensive methodology to accommodate generality. This generality is expensive. Capturing performance requires many measures of software behavior that then require automated modeling heuristics. Identifying determinants of performance is difficult without application semantics.

$$\mathbf{A} = \begin{pmatrix} a_{00} & a_{01} & 0 & 0 & 0 & 0 \\ a_{10} & a_{11} & 0 & 0 & a_{14} & a_{15} \\ 0 & 0 & a_{22} & 0 & a_{24} & a_{25} \\ 0 & 0 & 0 & a_{33} & a_{34} & a_{35} \end{pmatrix}$$

$$\texttt{b\_row\_start} = (0\ 2\ 4); \qquad \texttt{b\_col\_idx} = (0\ 4\ 2\ 4)$$

$$\texttt{b\_value} = (a_{00}\ a_{01}\ a_{10}\ a_{11}\ 0\ 0\ a_{14}\ a_{15}\ a_{22}\ 0\ 0\ a_{33}\ a_{24}\ a_{25}\ a_{34}\ a_{35})$$

Figure 11: BCSR with 2×2 blocks. 2×2 blocks are stored contiguously in `b_value`. The first column index of entry (1,1) in each 2×2 block is stored in `b_col_idx`. Pointers to block row starting positions in `b_col_idx` are stored in `b_row_start`.

Given domain knowledge, however, we can express software behavior more concisely. Rather than analyze individual instructions, we can analyze the behavior of libraries and algorithms. This approach is gaining traction in a number of domains, such as signal processing [37], linear algebra [3, 44], and program sketching [42]. From these diverse domain-specific program generators, we choose sparse matrix-vector multiply (SpMV [44]) for a case study.

5.1. Sparse Matrix-Vector Multiply (SpMV)

SPMV poses interesting challenges for integrated hardware-software models. Its performance trends are non-monotonic and its hardware-software interactions are sophisticated. We capture these subtleties while exploiting program-level software characteristics that keep model complexity modest.

Sparse matrix-vector multiply (SpMV) computes $v = v + Au$ when most elements in A are zero. We refer to u and v as the source and destination vectors, respectively. SpMV performance tuning is complicated by irregularity and variation in the best choice of sparse matrix data structure and code transformation across matrices and machines.

A particularly effective transformation improves locality by organizing a sparse matrix into $r\times c$ sub-blocks. Blocks with at least one non-zero are stored. The multiply proceeds block by block, re-using c source elements and streaming through r destination elements in a row-major matrix.

5.2. SpMV Hardware-Software Interactions

Choosing a matrix block size requires navigating sophisticated trade-offs between hardware and software. Sparse matrices store non-zero values with row and column pointers that map each value to a location in the matrix. Index overheads are reduced in a blocked compressed format since indices point to matrix blocks instead of individual matrix values (Figure 11).

However, this reduction is offset by explicit zeros which must be stored to create dense blocks of the same size; Figure 11 stores four such zeros. The fill ratio is the number of stored values (original non-zeros plus filled zeros) in a blocked matrix divided by the original number of non-zeros. Fill increases the number of unnecessary floating-point operations.

The precise balance of blocking's costs and benefits depends on input data, block size, and cache architecture. Sparser matrices and larger block sizes require more filled zeros to produce dense structure. And as density improves spatial locality, SpMV benefits from larger cache lines.

There exists an optimal balance between fill and locality. But we must strike this balance in an irregular performance topology. Rather than profile the cross-product of sparse matrix patterns, block sizes,

	Matrix	Dimension	Non-Zeros	Sparsity
1	3dtube	45330	1629474	7.93E-04
2	bayer02	13935	63679	3.28E-04
3	bcsstk35	30237	740200	8.10E-04
4	bmw7st	141347	3740507	1.87E-04
5	crystk02	13965	491274	2.52E-03
6	memplus	17758	126150	4.00E-04
7	nasasrb	54870	1366097	4.54E-04
8	olafu	16146	515651	1.98E-03
9	pwtk	217918	5926171	1.25E-04
10	raefsky3	21200	1488768	3.31E-03
11	venkat01	62424	1717792	4.41E-04

Table 4: Sparse matrices. N is dimension in a square matrix. Sparsity is number of non-zero elements divided by N^2.

SpMV		
x_1	brow, block row	1 :: 1+ :: 8
x_2	bcol, block column	1 :: 1+ :: 8
x_3	fR, fill ratio	function of brow,bcol,matrix
Cache Architecture		
y_1	lsize, line size	16B :: 2x :: 128B
y_2	dsize, data size	4KB :: 2x :: 256KB
y_3	dways, data ways	1 :: 2x :: ::8
y_4	drepl, data repl	LRU, NMRU, RND
y_5	isize, inst size	2KB :: 2x :: 128KB
y_6	iways, inst ways	1 :: 2x :: ::8
y_7	irepl, inst repl	LRU, NMRU, RND

Table 5: Hardware-software space, which includes software block sizes and hardware cache parameters.

and cache architectures, we infer models that accurately capture these complex relationships for coordinated optimization.

5.3. Accuracy for Coordinated Optimization

Software and Hardware Space. The integrated hardware-software space spans matrices, block sizes, and cache architectures. Table 4 lists matrices drawn from various application domains [34]. SpMV for each matrix has 64 variants with block sizes from 1×1 to 8×8, each with a corresponding fill ratio. These codes are generated automatically by OSKI [44].

Given the diversity in matrices and data structures, we evaluate a reconfigurable architecture that can accommodate them. To accurately assess core and cache reconfigurability, we use a simulator for a 400MHz Tensilica Xtensa processor, supplemented with CACTI and Micron to estimate cache and memory power [31, 33]. Because SpMV is memory-intensive, we focus on the cache (Table 5).

Non-monotonic Performance. Performance in this integrated hardware-software space is highly irregular.[4] Matrix blocking has a direct but discontinuous performance impact. Locality and performance increase with block size. But the largest blocks require the greatest number of filled zeros and produce diminishing marginal returns.

Figure 12 illustrates this irregularity in an example matrix, *raefsky3*. 8 block rows maximize performance but 6 or 7 block rows are only as effective as 2. For block columns, 1, 4, and 8 are equally effective, which reflects inherent dense sub-structure that arises in multiples of 4. A poorly chosen block size increases the fill ratio, which can harm performance (e.g., fR $>$ 1.25).

[4]Performance is true floating-point operations per second. The numerator excludes operations on filled zeros. The denominator includes reduced execution time from blocking.

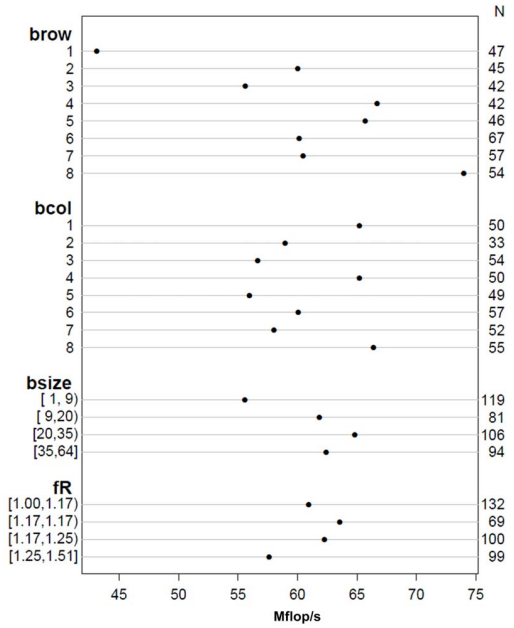


Figure 12: SpMV blocking parameters and performance. Data includes 400 samples drawn from integrated SpMV-cache space for *raefsky3*. Average Mflop/s is reported for all samples at each parameter value.

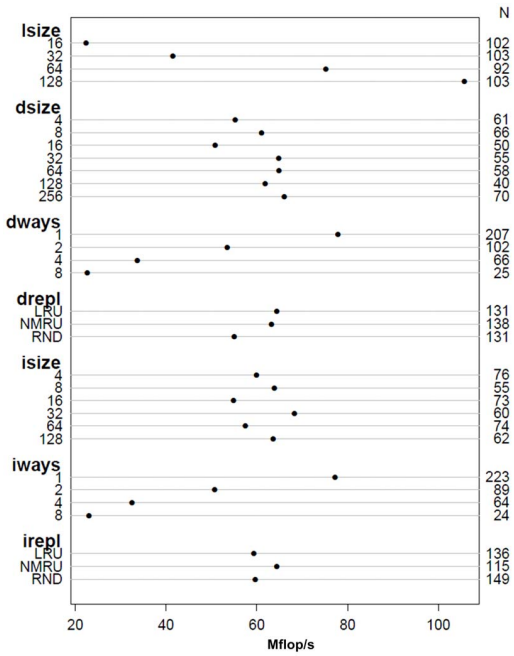


Figure 13: Cache architecture and performance trends. Data includes 400 samples drawn from integrated SpMV-cache space for *raefsky3*. Average Mflop/s is reported for all samples at each parameter value.

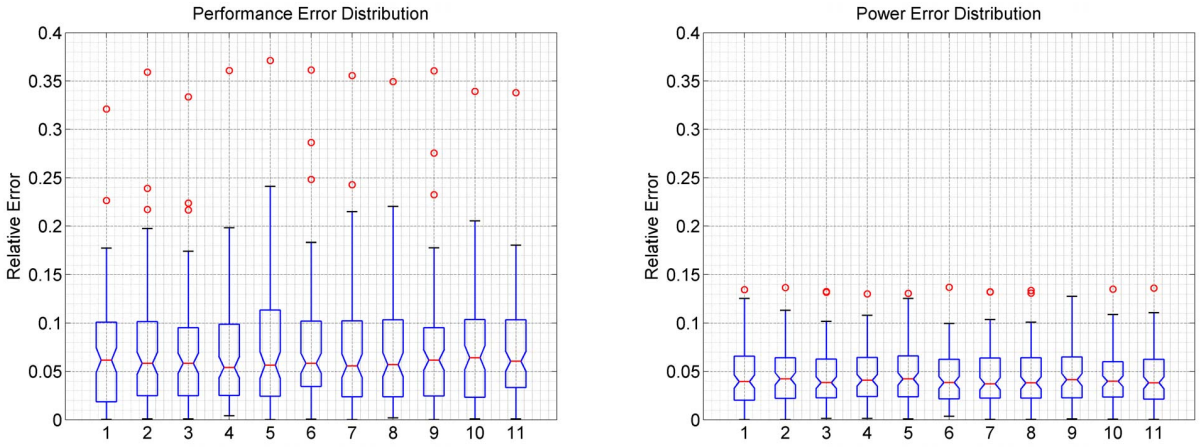


Figure 14: Distributions of (a) performance and (b) power prediction error. Matrix numbers refer to Table 4.

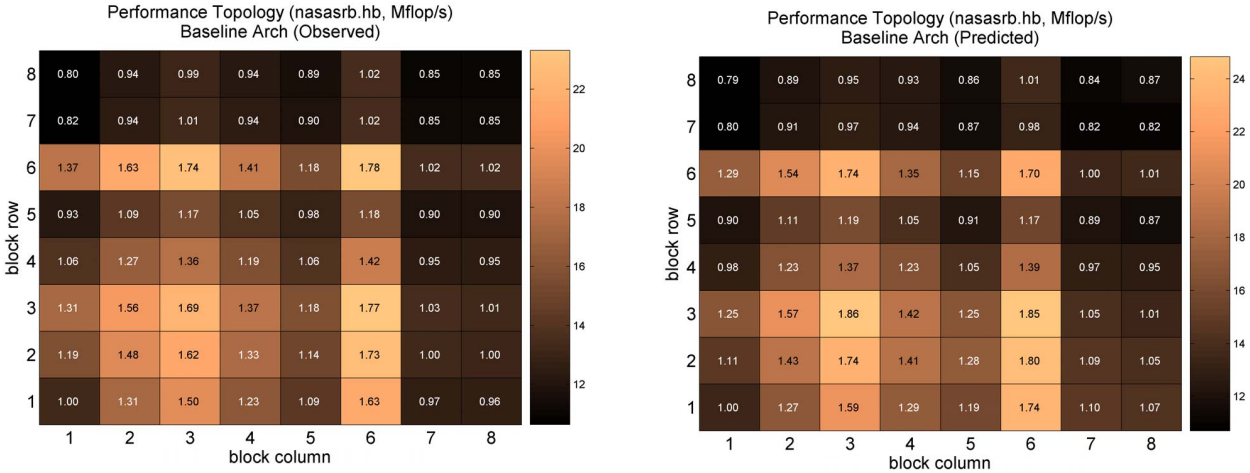


Figure 15: (a) Profiled and (b) predicted performance topologies. Colormap illustrates Mflop/s and numbers in each cell indicate speedup over 1×1 code. Data shown for a representative matrix *nasasrb*.

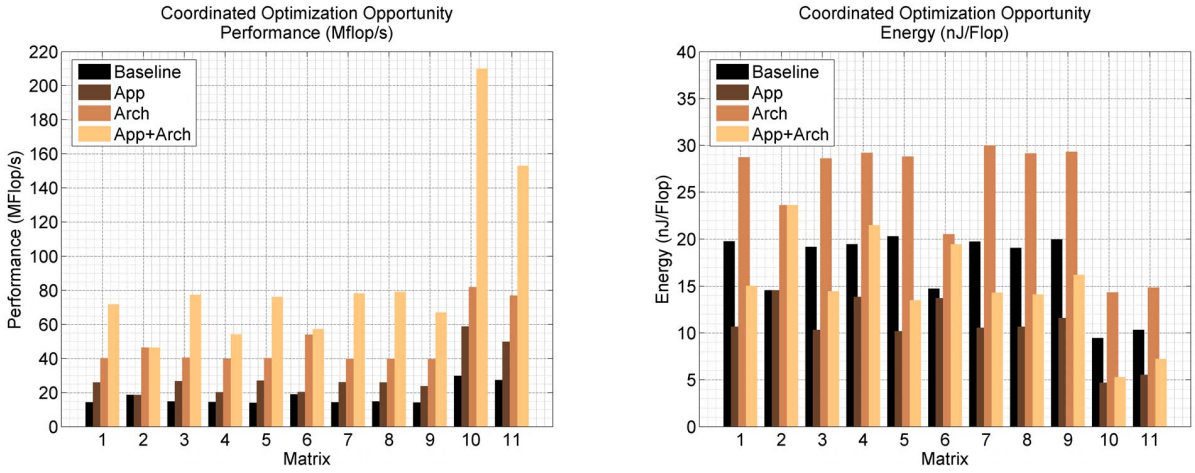


Figure 16: (a) Performance and (b) energy efficiency optimization. Matrix numbers refer to Table 4.

These blocking effects interact with cache structure, as shown in Figure 13. A larger cache line amortizes off-chip latency over a larger number of bytes to increase streaming bandwidth. Ideally, matrix values would not be cached since they are never re-used. But in a highly associative cache, matrix blocks occupy cache lines longer as they must travel down the LRU stack.

SpMV-Cache Modeling Accuracy. We train and validate a model for these complex performance effects. First, we randomly sample combinations of matrix block sizes and cache architectures. With these sparse samples, we fit regression models that predict performance and power using SpMV specific software parameters.

Such domain-specific parameters encapsulate much more information than generic instruction-level profiles. Rather than measure locality with re-use distances, SpMV block sizes directly quantify the amount of exploitable locality. Models use fewer, semantic-rich parameters to greater effect. With 400 sparsely sampled profiles for training and another 100 for validation, performance and power are accurately predicted with median errors between 4-6% across 11 matrices. Accuracy is shown in Figure 14.

Moreover, for a representative matrix *nasasrb*, Figure 15 illustrates the accuracy of inferred models on an irregular performance topology. The model accurately captures high performance at the same block sizes (3×3, 3×6, 6×3, 6×6). The model also captures discontinuities; many block sizes adjacent to 6×6 are worse than not blocking at all.

Coordinated Optimization. Advances in parameterized code generators and reconfigurable architectures make navigating an integrated hardware-software space imperative. In many domains, we can choose to tune the application, the architecture, or both. We compare these strategies for SpMV, exploiting the tractability of inferred models.

For SpMV, application tuning identifies the best matrix block size. Sparse matrices that have dense sub-structure (e.g., matrices 10 and 11) benefit from larger block sizes and modest fill ratios. Because SpMV is memory-bound, architecture tuning tailors the cache. Larger cache lines amortize off-chip memory latency. In Figure 16(a), application and architecture tuning improve performance by 1.6× and 2.7×, respectively. When tuned together, performance improves by 5.0×.

However, these tuning strategies incur different costs. Application tuning improves performance while simultaneously reducing energy as blocking improves locality and reduces the number of expensive memory accesses. With less data movement, both latency and energy fall. The optimal block size reduces energy from 17 to 11 nJ/Flop in Figure 16(b).

In contrast, energy increases to 25 nJ/Flop with architecture tuning. Larger cache lines increase the number of memory transfers, which cost 6nJ per 64b double-precision word [31]. This cost is difficult to justify unless the matrix is blocked to increase spatial locality. With coordinated tuning, energy per floating-point operation falls by 0.9× (i.e., 10% reduction) even as performance increases by 5.0×.

Collectively, these results motivate new thinking on efficiency. Architects cannot afford to ignore application tuning, which increases performance and reduces energy. For such tuning, inferred models provide tractability and insight.

6. Related Work

Hardware performance is modeled with statistical regression [26, 28] or neural nets [21]. Dubach et al. and Khan et al. separately construct models that predict new applications as a weighted combination of others by defining canonical machines and software profiles on them [11, 10, 25]. Instead of profiling each application on a few pieces of canonical hardware, we embed fundamental measures of software behavior into an integrated hardware-software model. Alternatively, analytical models link instruction behavior with pipeline structure [15, 24] and can help coordinate multiprocessor management [5].

We leverage prior work in microarchitecture-independent software characteristics to parameterize our model [6, 14, 20, 36]. We encounter challenges with variance in real application behavior, which might be mitigated with synthetic benchmarks controlled to produce uniform training data [13, 23]. Sampling identifies short, representative instruction segments within an application to predict its overall performance [41, 45]. Alternatively, a robust approach to experimental design might identify the subset of architectural simulations required to understand performance trends [46]. Like these prior works, we sample to reduce measurement costs but we do so uniformly at random. Moreover, our software samples profile behavior of fine-grained shards, which are shorter and more diverse than coarse-grained application phases.

Hardware-software co-tuning has been applied to dense matrix-matrix multiply, sparse matrix-vector multiply, and stencil computation [32]. Instead of exploratory profiling, we construct a model and reduce co-tuning costs. Similar models would benefit code generators and optimizers in a variety of application domains, including signal processing [37], linear algebra[3, 44], sorting [29], and back-end compilation [8].

Beyond software tuning for specific application domains, prior work has studied other predictors of software performance, such as how often particular methods are called and which compiler optimizations are applied. Chun et al. extract application-specific features (e.g., method invocation counts) from program analysis to predict performance but these features do not generalize across different applications [7]. Dubach et al. identifies more general predictors of software performance, some of which overlap with ours [9]. Navigating complex, back-end compiler optimizations is difficult and prior work has analyzed their impact on software performance [4, 16].

Many of these prior efforts focus entirely on performance variability in the software space for a given hardware platform. A notable exception, Dubach et al. coordinate the choice of compiler optimizations to the architectural design [12]. Rather than use compiler flags as predictors of software performance, we rely on sharded application behavior or domain-specific tuning parameters, allowing us to apply models beyond optimizing compilers.

7. Conclusions

We demonstrate a model for general applications by relying on shared behavior across application shards. By inferring performance models from software characteristics and hardware parameters, we better understand software preferences for hardware. Moreover, with modeling heuristics, such models can be updated automatically to reflect system dynamics.

Further accuracy is possible with domain-specific code generators. As application and architecture designers pursue performance and efficiency together, we need frameworks that maintain the integrity of abstraction layers while building new bridges across them. We preserve abstractions by synthesizing descriptors of software structure (e.g., matrix block size, fill ratio) and understanding their interactions with underlying hardware.

There are a number of directions for future work. In future, inferred models will need to accommodate virtual machines, which are prevalent in large, datacenter systems. We must determine whether modeling strategies change when hardware resources are virtualized and shared amongst co-located applications. These strategies may further extend to memory and I/O systems. Models for such systems require new parameters to characterize software behavior. By broadening the system scope, these models can be made even more relevant for datacenter management and big data computation.

Acknowledgements

This work is supported by NSF grant CCF-1149252 (CAREER). Any opinions, findings, conclusions, or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

References

[1] —. gem5. http://www.m5sim.org/.
[2] Amazon. Elastic cloud computing. http://aws.amazon.com/ec2/.
[3] J. Ansel et al. PetaBricks: a language for compiler and algorithmic choice. In *PLDI*, 2009.
[4] J. Cavazos et al. Automatic performance model construction for the fast software exploration of new hardware designs. In *CASES*, 2006.
[5] J. Chen and L. John. Predictive coordination of multiple on-chip resources for chip multiprocessors. In *ICS*, 2011.
[6] J. Chen, L. John, and D. Kaseridis. Modeling program resource demand using inherent program characteristics. In *SIGMETRICS*, 2011.
[7] B. Chun, L. Huang, S. Lee, P. Maniatis, and M. Naik. Mantis: Predicting system performance through program analysis and modeling. *Computing Research Repository (CoRR)*, 2010.
[8] K. Cooper, D. Subramanian, and L. Torczon. Exploring the structure of the space of compilation sequences using randomized search algorithms. *The Journal of Supercomputing*, 36(2), 2006.
[9] C. Dubach, J. Cavazos, B. Franke, G. Fursin, M. O'Boyle, and O. Temam. Fast compiler optimisation evaluation using code-feature based performance prediction. In *CF*, 2007.
[10] C. Dubach, T. Jones, E. Bonilla, G. Fursin, and M. O'Boyle. Portable compiler optimisation across embedded programs and microarchitectures using machine learning. In *MICRO*, 2009.
[11] C. Dubach, T. Jones, and M. O'Boyle. Microarchitectural design space exploration using an architecture-centric approach. In *MICRO*, 2007.
[12] C. Dubach, T. Jones, and M. O'Boyle. Exploring and predicting the architecture/optimising compiler co-design space. In *CASES*, 2008.
[13] L. Eeckhout, K. DeBosschere, and H. Neefs. Performance analysis through synthetic trace generation. In *ISPASS*, 2000.
[14] L. Eeckhout, J. Sampson, and B. Calder. Exploiting program microarchitecture independent characteristics and phase behavior for reduced benchmark suite simulation. In *IISWC*, 2005.
[15] S. Eyerman, L. Eeckhout, T. Karkhanis, and J. Smith. A performance counter architecture for computing accurate CPI components. In *ASPLOS*, 2006.
[16] G. Fursin and O. Temam. Collective optimization: A practical collaborative approach. *TACO*, 2010.
[17] A. Ghodsi et al. Dominant resource fairness: Fair allocation of multiple resource types. In *Proc. NSDI*, 2011.
[18] F. Harrell. *Regression Modeling Strategies*. Springer, 2001.
[19] B. Hindman et al. Mesos: A platform for fine-grained resource sharing in the data center. In *Proc. NSDI*, 2011.
[20] K. Hoste et al. Performance prediction based on inherent program similarity. In *PACT*, 2006.
[21] E. Ipek, S. McKee, B. de Supinski, M. Schulz, and R. Caruana. Efficiently exploring architectural design spaces via predictive modeling. In *ASPLOS*, 2006.
[22] P. Joseph, K. Vaswani, and M. J. Thazhuthaveetil. A predictive performance model for superscalar processors. In *MICRO*, 2006.
[23] A. Joshi, L. Eeckhout, L. John, and C. Isen. Automated microprocessor stressmark generation. In *HPCA*, 2008.
[24] T. Karkhanis and J. Smith. Automated design of application specific superscalar processors: An analytical approach. In *ISCA*, 2007.
[25] S. Khan, P. Xekalakis, J. Cavazos, and M. Cintra. Using predictive modeling for cross-program design space exploration in multicore systems. In *PACT*, 2007.
[26] B. Lee and D. Brooks. Accurate and efficient regression modeling for microarchitectural performance and power prediction. In *ASPLOS*, 2006.
[27] B. Lee and D. Brooks. Illustrative design space studies with microarchitectural regression models. In *HPCA*, 2007.
[28] B. Lee, J. Collins, H. Wang, and D. Brooks. CPR: Composable performance regression for scalable multiprocessor models. In *MICRO*, 2008.
[29] X. Li et al. A dynamically tuned sorting library. In *CGO*, 2004.
[30] J. Mars et al. Bubble-Up: Increasing utilization in modern warehouse scale computers via sensible co-locations. In *MICRO*, 2011.
[31] Micron. Technical note TN-47-04: Calculating memory system power for DDR2. In *www.micron.com*, 2006.
[32] M. Mohiyuddin et al. A design methodology for domain-optimized power-efficient supercomputing. In *SC*, 2009.
[33] N. Muralimanohar, R. Balasubramonian, and N. Jouppi. Optimizing NUCA organizations and wiring alternatives for large caches with CACTI 6.0. In *MICRO*, 2006.
[34] NIST. Matrix market. http://math.nist.gov/MatrixMarket/.
[35] E. Perelman et al. Using SimPoint for accurate and efficient simulation. In *SIGMETRICS*, 2003.
[36] A. Phansalkar et al. Analysis of redundancy and application balance in SPEC CPU 2006 benchmark suite. In *ISCA*, 2007.
[37] M. Pueschel et al. SPIRAL: Code generation for DSP transforms. *Proc. IEEE*, 2005.
[38] V. Reddi et al. Web search using mobile cores: Quantifying and mitigating the price of efficiency. In *ISCA*, 2010.
[39] G. Ren et al. Google-wide profiling: A continuous profiling infrastructure for data centers. *IEEE Micro*, 2010.
[40] X. Shen, Y. Zhang, and C. Ding. Locality phase prediction. In *ASPLOS*, 2004.
[41] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *ASPLOS*, 2002.
[42] A. Solar-Lezama, R. Rabbah, R. Bodk, and K. Ebcioglu. Programming by sketching for bitstreaming programs. In *PLDI*, 2005.
[43] A. Solomatnikov et al. Using a configurable processor generator for computer architecture prototyping. In *MICRO*, 2009.
[44] R. Vuduc, J. Demmel, and K. Yelick. OSKI: A library of automatically tuned sparse matrix kernels. In *SciDAC, Journal of Physics*, 2005.
[45] R. Wunderlich et al. Smarts: Accelerating microarchitecture simulation via rigorous statistical sampling. In *ISCA*, 2003.
[46] J. Yi, D. Lilja, and D. Hawkins. A statistically rigorous approach for improving simulation methodology. In *HPCA*, 2003.

2012 IEEE/ACM 45th Annual International Symposium on Microarchitecture

SMARQ: Software-Managed Alias Register Queue for Dynamic Optimizations

Cheng Wang Youfeng Wu Hongbo Rong Hyunchul Park

Programming Systems Lab
Microprocessor and Programming Research
Intel Labs
{cheng.c.wang, youfeng.wu, hongbo.rong, hyunchul.park}@intel.com

Abstract

Traditional alias analysis is expensive and ineffective for dynamic optimizations. In practice, dynamic optimization systems perform memory optimizations speculatively, and rely on hardware, such as alias registers, to detect memory aliases at runtime. Existing hardware alias detection schemes either cannot scale up to a large number of alias registers or may introduce false positives. Order-based alias detection overcomes the limitations. However, it brings considerable challenges as how software can efficiently manage the alias register queue and impose restrictions on optimizations. In this paper, we present SMARQ, a Software-Managed Alias Register Queue, which manages the alias register queue efficiently and supports more aggressive speculative optimizations. We conducted experiments with a dynamic optimization system on a VLIW processor that has 64 alias registers. The experiments on a suite of SPECFP2000 benchmarks show that SMARQ improves the overall performance by 39% as compared to the case without hardware alias detection. By scaling up to a large number (from 16 to 64) of alias registers, SMARQ improves performance by 10%. Compared to a technique with false positives (similar to Itanium), SMARQ improves performance by 13%. To reduce the chance of alias register overflow, the novel alias register allocation algorithm in SMARQ reduces the alias register working set by 74% as compared to a straightforward alias register allocation based on program order.

1. Introduction

Dynamic optimization systems perform optimizations at runtime. Thus it is crucial to keep under control the time spent on the optimizations. It is difficult and expensive to perform traditional memory alias analysis at runtime [13]. First, the dynamic optimizer may not have source level information, such as data type, array and subscript information. Second, for dynamic optimizations the alias analysis time is part of the execution time, and the expensive alias analysis can seriously impact the overall performance [13]. On the other hand, memory alias information is critical to the effectiveness of optimizations, especially for in-order processors [3]. Due to this dilemma, dynamic optimizations on in-order processors usually use simple or speculative alias analysis algorithms [1, 14] and rely on hardware (HW) to detect aliases at runtime. In this way, they can optimize memory accesses speculatively [6, 10].

Figure 1 shows the overall framework for a dynamic optimization system with HW alias detection. The application code runs on top of the system, which dynamically optimizes the application code to run on the CPU through an *optimizer*. The optimizer may assume no alias for memory operations that are unlikely to alias to each other. When the optimized code runs, the *alias HW* in CPU may detect the aliases for these operations and raise an alias exception. The optimized code is put into atomic regions for speculative execution [10] such that at the entry of each atomic region execution, the *atomicity HW* in CPU will create a checkpoint for rolling back the region in case of alias exception. The *runtime* module in the dynamic optimization system will catch the alias exceptions and trigger the optimizer to re-optimize the region conservatively: this time it assumes the two memory operations that just triggered the exception are always aliased. All the HW interrupts, exceptions and memory consistency violations are also caught by the runtime module to trigger region rollbacks [11].

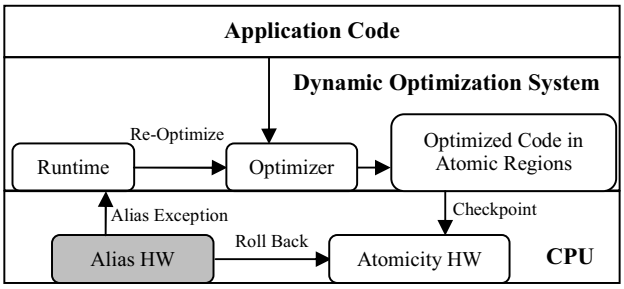


Figure 1: A Dynamic Optimization System with HW Alias Detection

Existing HW alias detection schemes suffer from several problems. HW alias detection on Transmeta Efficeon [6] is not scalable and cannot support more than 15 alias registers. HW alias detection on Itanium [2] (which is called Advanced Load Address Table or ALAT) may introduce false positives and cannot detect aliases between stores. Order-based alias detection overcomes the limitations in Efficeon and Itanium. However, it brings considerable challenges to the efficient software management of the alias register queue and imposes restrictions on optimizations. (We will describe all of them in details in Section 2). In this paper, we present SMARQ, a Software-Managed Alias Register Queue for speculative dynamic optimization, which overcomes all the problems in the existing HW alias detection schemes. We conducted experiments with a dynamic binary translation/optimization system on a VLIW processor that has 64 alias registers. The experiments on a suite of SPECFP2000 benchmarks show that optimizations with SMARQ improve the overall performance by 39% as compared to those without hardware alias detection support. By scaling up the alias register number from 16 to 64, SMARQ improves performance by 10%. Compared to a technique with false positives (similar to Itanium), SMARQ improves performance by 13%. To reduce the chance of alias register overflow (i.e. running out of alias registers), the novel alias register allocation algorithm in SMARQ reduces the alias register working set by 74% as compared to a straightforward alias register allocation based on program order.

The rest of paper is organized as follows. In Section 2, we present the motivation of the work. In Section 3, we use several examples to show the architectural features used in SMARQ. In Section 4, we develop the compiler analysis framework for identi-

1072-4451/12 $26.00 © 2012 IEEE
DOI 10.1109/MICRO.2012.46

IEEE computer society

fying the alias detection constraints. In Section 5, we describe our fast alias register allocation algorithm to satisfy all alias detection constraints. We present our experiments in Section 6 and discuss the related works in Section 7. At last, we conclude the paper and point out the future direction in Section 8.

2. Motivation

In this section, we illustrate the concept of HW alias detection. Then we show the HW alias detection in Efficeon and Itanium, and point out their weaknesses. We motivate our solution that addresses the weaknesses.

2.1 HW Alias Detection

Figure 2 shows an example for HW alias detection. The code in Figure 2 (a) is speculatively optimized into the code in Figure 2 (b). For simplicity, the only optimization applied here is reordering the memory accesses such that loads are performed as early as possible. Since it is unlikely that load M_3 and store M_2 alias to each other, M_3 is speculatively reordered above M_2 after the optimization. Also, M_0 is reordered below M_2. For HW alias detection between M_3 and M_2, the optimizer needs to annotate M_3 to set an *alias register AR0* and annotate M_2 to check *AR0* for aliasing. Assume each memory access is 4 bytes. Then during the execution, M_3 will set *AR0* with its memory access range *[r2, r2+3]*. Later M_2 will check this range in *AR0* against its own memory access range *[r0, r0+3]*. If the two ranges overlap, an alias exception is raised.

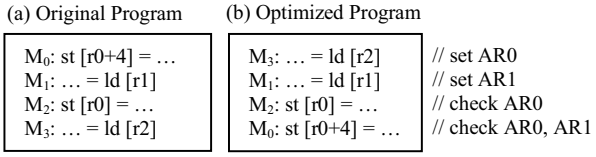


Figure 2: A HW Alias Detection Example

2.2 Efficeon HW Alias Detection

A memory operation may need to check multiple alias registers. For example in Figure 2, M_0 needs to check 2 alias registers, *AR0* and *AR1*. Transmeta Efficeon [6] uses a bit-mask in the instruction to specify the individual alias registers that need to be checked. There is limited encoding space for the bit-mask in an instruction. For this reason, the alias register file cannot scale up to a large number of alias registers. In the current encoding scheme, Efficeon cannot support more than 15 alias registers. This is tolerable in Efficeon where an atomic region is typically small, with only about 30 x86 instructions on average. However, large scheduling/optimization regions are critical for achieving good performance on in-order processors [15, 19]. Since speculative optimization with large regions will inevitably use more alias registers, the scalability of alias register support will be a serious issue. In our experiment with a dynamic binary translation/optimization system on a VLIW processor, we see performance improvement for *ammp* benchmark in SPEC 2000 by 30% by using 64 alias registers instead of 16 alias registers.

2.3 Itanium HW Alias Detection

Itanium memory alias detection [2] (which is called Advanced Load Address Table or ALAT) requires stores to automatically check all the alias registers set by previous advanced loads (i.e. ld.a instruction) without the need to specify the individual alias registers that need to be checked. Unfortunately, that may lead to false positives in the alias detection, which is a serious issue in dynamic optimization (Note that Itanium was not designed for dynamic optimization). Figure 3 shows the Itanium alias detection for the example shown in Figure 2. M_2 does not need to check M_1 (i.e. check AR1) for aliasing, instead, Itanium ALAT hardware checks M_2 against all alias registers. This would result in a false positive if M_1 aliases with M_2 (Remember that M_1 and M_2 have not been reordered). In our experiments on *ammp* benchmark, the false positives can lead to performance degradation of 47%. Moreover, Itanium memory alias detection cannot detect aliases between stores, and therefore, it does not allow speculative reordering between stores. However, our experiments show that speculative reordering between stores is a desirable optimization: it can improve performance by 13% for *mesa* benchmark in SPEC2000.

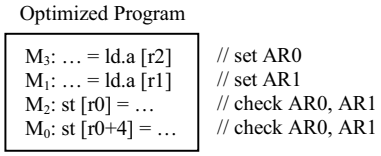


Figure 3: Itanium Alias Detection for Optimization in Figure 2

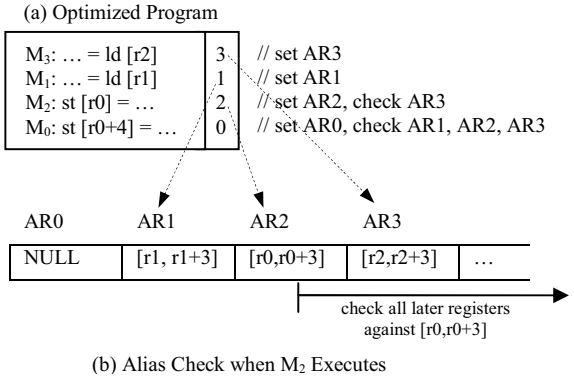


Figure 4: Order-Based Alias Detection for Optimization in Figure 2

2.4 Order-Based HW Alias Detection

To overcome the scaling bottleneck, prevent the false positives and allow aggressive optimizations (like reordering stores), we can adopt the idea of order-based memory alias detection [4], which was used in Memory Order Buffer (MOB) in out-of-order processor to detect aliases between memory operations reordered by SW. In order-based memory alias detection, the alias registers are organized into an ordered circular queue (called alias register queue in this paper). Each memory operation is allocated an alias register, whose order matches the original program execution order. Figure 4 shows how the order-based alias detection works for the example shown in Figure 2. The alias register numbers 0~3 (see the second column of the code in Figure 4 (a)) corresponding to alias registers AR0~AR3. They are allocated to memory operations M_0~M_3 respectively in their original program execution order. When a memory operation executes, HW will set its memory access range into the allocated alias register and check all *later* alias registers in the alias register queue set by previously executed instructions for aliasing. Here by "alias register ARx is later than alias register ARy", we mean $x>y$. So when M_2 executes, the alias registers will have contents as shown in Figure 4 (b), and HW will only check *AR3* but not *AR1* because in the alias register queue, *AR1* is earlier than the alias register allocated to M_2 (i.e. *AR2*). HW also auto-

matically marks the alias registers that are set by loads so that later loads do not check against them. So in this example, M_1 will not check M_3 for aliasing (i.e. not check *AR3*). In general, order-based alias detection can guarantee to *detect all the aliases between reordered memory operations without any false positive*. Table 1 shows the comparison between order-based alias detection and alias detection on Efficeon and Itanium.

Table 1: Comparison between different HW Alias Detections

	Efficeon	Itanium	Order-Based
Features	bit-mask	ALAT	ordered queue
Scalability	Poor	Good	Good
False positive	No	Yes	No
Detect alias between stores	Yes	No	Yes

Even though the order-based alias detection can overcome all the problems with Efficeon and Itanium, it has three serious drawbacks. First, it is very inefficient in alias register usage by allocating each memory operation an alias register in program order. Second, it may perform many unnecessary alias detections, which costs energy. For example in Figure 4, the compiler analysis may find that M_0 and M_2 never alias to each other. So *M_0 does not need to check M_2 (i.e. check AR2) for aliasing, even though they are reordered in optimization*. Moreover, allocating alias registers in program order can only detect aliases if speculative memory reordering is the only optimization applied, but not if there are other optimizations such as speculative load/store elimination, which may need alias detection between memory operations that are not re-ordered. For example in Figure 5 (a), M_1 and M_4 access the same memory location so we speculatively eliminate the load at M_4 by forwarding data from M_1 to M_4 (see Figure 5 (b)), assuming there is no alias between M_1 and M_3. To ensure correctness, *M_3 needs to check M_1 for aliasing, even though they are not reordered in the optimization.*

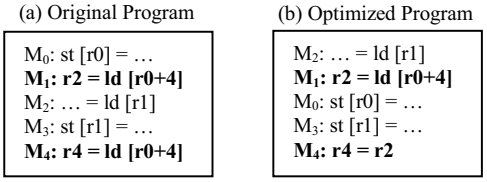


Figure 5: A Speculative Load Elimination Example

2.5 SMARQ: Software-Managed Alias Register Queue

To reduce unnecessary alias register usage in order-based alias detection and keep the alias register allocation algorithm to be fast and suitable for dynamic optimizations, SMARQ use a *constraint order* among the memory operations derived from compiler analysis to enforce the order among the registers assigned to the memory operations. Our constraints not only make sure that all required alias detections are performed, but also avoid any unnecessary alias check that may lead to false positive alias exception. Based on the constraint graph, SMARQ integrates alias register allocation and instruction scheduling in a single pass, and leverage a few architecture features to achieve the above objectives. SMARQ improve over the order-based alias detection approaches in following way:

- reducing unnecessary alias detection between reordered memory operations (between M_0 and M_2 in Figure 4)
- performing the necessary alias detection between even non-reordered memory operations for speculative load/store eliminations (between M_1 and M_3 in Figure 5)

Below we will first describe the architectural features used in SMARQ. Then we show the two components in our solution in details: constraint analysis and alias register allocation algorithm.

3. SMARQ Architectural Features

In this section, we introduce the key architectural features in SMARQ, which the compiler analysis can leverage to allocate alias registers efficiently. These features include 1) protection (*P*) and check (*C*) bits; 2) alias register rotation; 3) alias moving. These features are useful for avoiding unnecessary alias detection, reducing alias register overflow and preventing false positive in the alias detection.

3.1 Avoid Unnecessary Alias Detection with P/C Bits

To avoid unnecessary alias detection, in addition to the alias register number, we allow the compiler to set a *P* bit to a memory operation that needs to set an alias register, and a *C* bit to a memory operation that needs to check alias registers. So we get:

[ORDERED-ALIAS-DETECTION-RULE] Memory operation *X* checks memory operation *Y* for memory aliasing if and only if:

- *Y* precedes *X* in the optimized program execution and,
- *X* has *C* bit and,
- *Y* has *P* bit and,
- The alias register allocated to *X* is not later than the alias register allocated to *Y* in the alias register queue.

For a memory operation with both *P* bit and *C* bit, the memory operation will check alias registers before setting the alias register to prevent the alias detection on itself. As an example, Figure 6 shows the order-based alias detection with *P*/*C* bit for the optimization shown in Figure 2. M_3 and M_1 do not need to check any earlier executed memory operation for memory aliasing. So M_3 and M_1 only set alias register numbers 1 and 0 respectively with *P* bits without *C* bits. M_0 and M_2 do not need to set an alias register because no later executed memory operation needs to check them for memory aliasing. So memory operations M_2 and M_0 only have *C* bits but no *P* bits. In this way, we achieve the same efficient alias detection as Efficeon (Figure 2) without using a bit-mask. Compared to the order-based alias detection of Memory Ordered Buffer (Figure 4), we not only avoid the unnecessary alias detection between M_0 and M_2, but also reduce the alias registers used in the optimization from 4 to 2.

instruction	alias annotation			
M3: … = ld [r2]	1	P	-	// set AR1
M1: … = ld [r1]	0	P	-	// set AR0
M2: st [r0] = …	1	-	C	// check AR1
M0: st [r0+4] = …	0	-	C	// check AR0, AR1

Figure 6: Alias Detection with P/C bits for Optimization in Figure 2

3.2 Reduce Alias Register Overflow with Rotation

An Alias register rotation instruction increments a base alias register pointer, *BASE,* by a specified value (with the wrap-around of the circular queue), and clean up the registers between the previous BASE and the current BASE. With register rotation, alias registers are referenced in the program only by non-negative offsets relative to the current *BASE*. As an example, Figure 7 (a) shows an alias

register allocation for a reordering optimization, where M_0, M_1, …, M_5 represent their original program execution order. Since no memory operation after M_0 will set or check *AR0*, we can rotate the alias register by *1* after M_0 as shown in Figure 7 (b). After the rotation, the base alias register pointer *BASE* will point to *AR1*, and alias register *AR0* is cleaned up and logically becomes a free alias register at the end of the alias register queue. Then M_4 needs to use offset *1* to reference alias register *AR2* instead of offset *2*. Similarly, we can rotate the alias register by another *1* after M_1.

Rotation can reduce the register usage, and thus the chance of alias register overflow. For example, if HW supports only 2 alias registers, we can run the code in Figure 7 (b), as after the first rotation, AR0 is reused as alias register *AR2* at offset 1. However, we cannot run the code in Figure 7 (a) as there is no alias register at offset 2. In general, the maximum offset + 1 gives the minimum number of HW alias registers required to run the code without alias register overflow.

It seems that in Figure 7 (a), we may directly reuse and set/check the alias register *AR0* in M_4 and M_3 without the rotation. However, the check of *AR0* in M_3 in this example may lead to a false positive alias detection between M_2 and M_3 (*AR1* allocated to M_2 is later than *AR0*). In general, reusing alias register without rotation makes it hard for the alias register allocation algorithm to manage the ordered alias registers to prevent false positive (see more details in section 5). Therefore, SMARQ is designed to reuse alias registers only through rotation.

(a) Without Rotation

M_5: … = ld [r1+8]	0	P	-	// set AR0
M_2: … = ld [r2]	1	P	-	// set AR1
M_0: st [r0] = …	0		C	// check AR0, AR1
M_4: … = ld [r4]	**2**	P	-	// **set AR2**
M_1: st [r1] = …	1	-	C	// check AR1, AR2
M_3: st [r1+4] =	2	-	C	// check AR2

(b) With Rotation

M_5: … = ld [r1+8]	0	P	-	// set AR0
M_2: … = ld [r2]	1	P	-	// set AR1
M_0: st [r0] = …	0	-	C	// check AR0, AR1
rotate 1	-	-	-	**// rotate BASE to AR1**
M_4: … = ld [r4]	**1**	P	-	// **set AR2**
M_1: st [r1] = …	0	-	C	// check AR1, AR2
rotate 1	-	-	-	// rotate BASE to AR2
M_3: st [r1+4] =	0	-	C	// check AR2

(c) Dependences (d) Constraints

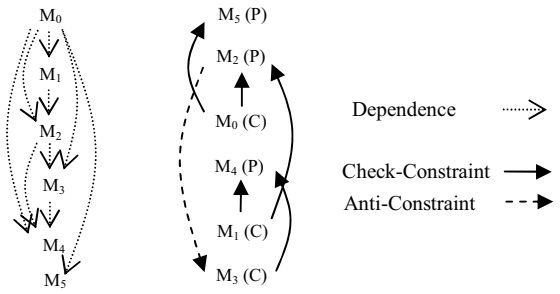


Figure 7: An Alias Register Rotation Example

Due to the rotation of *BASE*, the "offset" relative to *BASE* does not reflect the actual order of the alias registers in the alias register queue. To avoid confusion, in the rest of the paper, we will use "*order*" to refer to the alias register number relative to *BASE 0*, and "*offset*" to refer to the alias register number relative to the *BASE* at the memory operation execution. So in Figure 7 (b), the offset of the register allocated to M_4 is *1* (denoted $offset(M_4) = 1$), but the order of the register allocated to M_4 is *2* (denoted $order(M_4) = 2$) because the BASE at M_4 execution is *1* (denoted $base(M_4) = 1$). In general, *base(X)* keeps increasing in the optimized program execution with the rotation instructions and *order(X)* reflects the actual order of alias registers in the circular queue with the invariance: *order(X) = base(X) + offset(X)*. Note that *base(X)* and *order(X)* are independent of HW register count and can be from 0 to infinity, but *offset(X)* must be smaller than the HW alias register count. Since alias register allocation in SMARQ assigns an *order(X)* to each memory operation *X* that needs an alias register, we may refer to the allocation of alias registers as the *allocation of alias register orders*.

3.3 Prevent False Positive with Alias Moving

Although order-based alias detection can guarantee to detect all the aliases between reordered memory operations without any false positive, the alias detection between non-reordered memory operations for speculative load/store eliminations (see an example in Figure 5) may introduce false positives (see detailed discussions in Section 5.3). To prevent false positives, SMARQ introduces an alias moving instruction denoted "*AMOV offset1, offset2*", which moves the memory access range stored in the alias register at *offset1* to the alias register at *offset2*. After the moving, the alias register at *offset1* is cleaned up. We also allows *offset2* to be same as *offset1*, which only cleans up memory access range in the alias register at *offset1* without moving it to another alias register. The alias moving instructions can prevent all false positives in the alias detection.

4. Constraint Analysis

In this section, we develop the compiler analysis to identifying the necessary alias detection constraints in three general speculative optimizations: memory operation reordering, load elimination and store elimination. These general optimizations subsume other specific memory optimizations, such as speculative register promotion [7], etc. As the first study on this topic, we focus on optimization in superblock regions [15], and we believe the solution can be generalized to arbitrary code regions.

4.1 Check-constraints

In our compiler analysis, we first derive a set of *check-constraint*s, denoted *X* →*check Y*, such that *X* needs to check *Y* for aliasing to ensure optimization correctness. We first consider only instruction scheduling without speculative load/store elimination. In this case, check-constraints can be computed in two steps. Before the instruction scheduling, we compute a set of dependences, denoted *X* →*dep Y*, such that memory operation *Y* depends on memory operation *X* with the following conditions:

[DEPENDENCE] *X* →*dep Y* if:

- *X* precedes *Y* in the original program execution order and,
- *X* and *Y* may (including must) access the same memory location and,
- At least one of *X* and *Y* is a store.

Then after the instruction scheduling, we derive the check-constraints *X* →*check Y* from dependences *X* →*dep Y* with the following condition:

[CHECK-CONSTRAINT] $X \rightarrow check\ Y$ if:

- $X \rightarrow dep\ Y$ and,
- Y precedes X after scheduling.

As an example, for the memory reordering in Figure 7, Figure 7 (c) shows all the dependences before scheduling. The solid edges in Figure 7 (d), on the other hand, show all the check-constraints derived from dependences after scheduling. There is no dependence $M_1 \rightarrow dep\ M_5$ or $M_3 \rightarrow dep\ M_5$ since the compiler can easily disambiguate them. Due to that, there is no check-constraint $M_1 \rightarrow check\ M_5$ or $M_3 \rightarrow check\ M_5$, even though they are reordered in the scheduling. For each check-constraint $X \rightarrow check\ Y$, we set the C bit to X and the P bit to Y, as shown aside with the memory operations in Figure 7 (d).

To handle speculative load elimination, we only need to extend dependences with the following condition:

[EXTENDED-DEPENDENCE 1] If a load Z is eliminated speculatively by forwarding data from an earlier memory operation X (a store or a load) to Z, we add extended dependences $Y \rightarrow dep\ X$ for all stores Y satisfying:

- Y is between X and Z in the original program execution order and,
- Y and X may access the same memory.

As an example, in the optimization with speculative load elimination in Figure 5, M_1 and M_4 access the same memory location so we speculatively eliminate the load at M_4 by forwarding data from M_1 to M_4, and according to EXTENDED-DEPENDENCE 1, derive the extended dependence $M_3 \rightarrow dep\ M_1$, as shown in Figure 8 (a). With the extended dependence, we derive a check-constraint $M_3 \rightarrow check\ M_1$ after the scheduling according to CHECK-CONSTRAINT, as shown by the bold edge in Figure 8 (b). This is necessary because we need to enforce the alias detection between M_1 and M_3 in order for the load elimination to be correct, even though M_1 and M_3 are not reordered. Note that there are both dependence $M_1 \rightarrow dep\ M_3$ and extended dependence $M_3 \rightarrow dep\ M_1$ in Figure 8 (a). So depending on the orders between M_1 and M_3 after the scheduling, we will have either check-constraint $M_1 \rightarrow check\ M_3$ or $M_3 \rightarrow check\ M_1$ to enforce the alias detection between M_1 and M_3.

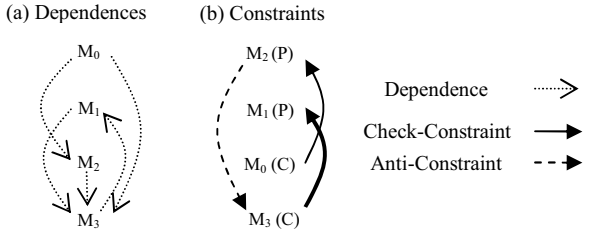


Figure 8: Dependences and Constraints for Optimization in Figure 5

Similarly, to handle speculative store elimination, we only need to extend dependences with the following condition:

[EXTENDED-DEPENDENCE 2] If a store X is eliminated due to the overwriting of the same memory location by a later store Z, we add extended dependence $Z \rightarrow dep\ Y$ for all loads Y satisfying:

- Y is between X and Z in the original program execution order and,
- Z and Y may access the same memory.

Figure 9 shows an example for speculative store elimination. M_5 overwrites the same memory location as accessed by M_0. So we speculatively eliminate M_0, and according to EXTENDED-DEPENDENCE 2, derive the extended dependence $M_5 \rightarrow dep\ M_1$, as shown in Figure 9 (c). With the extended dependence, we derive the check-constraint $M_5 \rightarrow check\ M_1$ as shown by the bold edge in Figure 9 (d). This is necessary because we need to enforce the alias detection between M_1 and M_5 in order for the store elimination to be correct, even though M_1 and M_5 are not reordered after optimization. It is interesting that in EXTENDED-DEPENDENCE 2, we add extended dependence $Z \rightarrow dep\ Y$ only for *load* Y between X and Z, but not for *store* Y between X and Z. Due to that, we do not enforce the alias detection between M_2 and M_5, or between M_4 and M_5, as the aliases between them do not affect the correctness of the store elimination (i.e. there is no need for check-constraint $M_5 \rightarrow check\ M_2$ or $M_5 \rightarrow check\ M_4$).

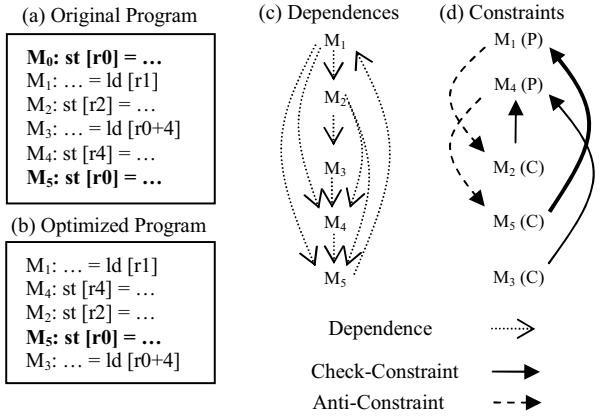


Figure 9: A Speculative Store Elimination Example

4.2 Anti-Constraints

In the optimization example shown in Figure 5, we get the check-constraints $M_0 \rightarrow check\ M_2$ and $M_3 \rightarrow check\ M_1$, as shown in Figure 8 (b). Due to ORDERED-ALIAS-DETECTION-RULE (described in Section 3), the alias detection between M_0 and M_2 requires $order(M_0) \leq order(M_2)$, and the alias detection between M_3 and M_1 requires $order(M_3) \leq order(M_1)$. If we allocate $order(M_2) < order(M_1)$, transitively, we will have $order(M_0) \leq order(M_2) < order(M_1)$, which means that M_0 will additionally check M_1 for aliasing. The resulting alias detection is shown in Figure 10 (a) (note the additional check of *AR1* by M_0). Similarly, if we allocate $order(M_1) < order(M_2)$, transitively, we will have $order(M_3) \leq order(M_1) < order(M_2)$, which means that M_3 will additionally check M_2 for aliasing. The resulting alias detection is shown in Figure 10 (b) (note the additional check of *AR1* by M_3). In either case, we have to perform an additional alias detection not specified in the check-constraints in Figure 8 (b).

Although the additional detection between M_0 and M_1 in Figure 10 (a) is benign (i.e. will never generate alias exception), the additional alias detection between M_2 and M_3 in Figure 10 (b) will generate an alias exception, even though the alias does not affect the optimization correctness. This false positive will result in expensive rollback of execution.

To prevent false positive, our compiler analysis also derives *anti-constraint*, denoted $X \rightarrow anti\ Y$, such that X should not be checked by Y. So for the optimization in Figure 5, we need to derive the anti-constraint $M_2 \rightarrow anti\ M_3$ to prevent M_2 from being checked by M_3. Figure 11 shows how to prevent false positive

with anti-constraints. There are multiple alias register allocations (i.e. allocation 1 and allocation 2) that can perform all the alias detections enforced by check-constraints. However, allocation 2 also performs certain alias detections prohibited by anti-constraints which may lead to false positive. Our alias register allocation algorithm aims to find allocation 1, which performs all the alias detections enforced by the check-constraints without any alias detections prohibited by the anti-constraints.

(a) Alias Register Allocation 1

M_2: … = ld [r1]	0	P	-	// set AR0
M_1: r2 = ld [r0+4]	1	P	-	// set AR1
M_0: st [r0] = …	0	-	C	// check AR0, **AR1**
M_3: st [r1] = …	1	-	C	// check AR1
M_4: r4 = r2	-	-	-	

(b) Alias Register Allocation 2

M_2: … = ld [r1]	1	P	-	// set AR1
M_1: r2 = ld [r0+4]	0	P	-	// set AR0
M_0: st [r0] = …	1	-	C	// check AR1
M_3: st [r1] = …	0	-	C	// check AR0, **AR1**
M_4: r4 = r2	-	-	-	

Figure 10: False Positive for the Example in Figure 5

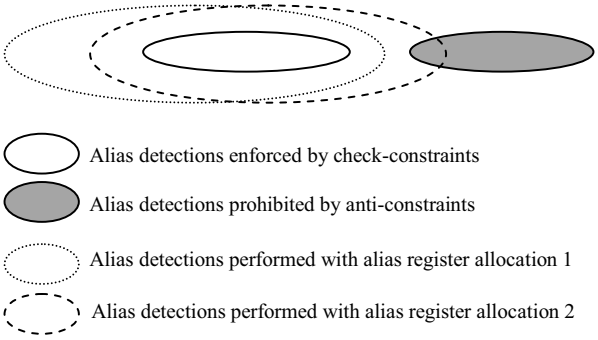


Figure 11: Preventing False Positive

An anti-constraint $X \rightarrow anti\ Y$ is needed only if X and Y may alias to each other. So we compute the anti-constraints $X \rightarrow anti\ Y$ from the subset of dependences $X \rightarrow dep\ Y$ such that X precedes Y after scheduling. Moreover, if $Y \rightarrow check\ X$, then Y does need to check X for aliasing to ensure optimization correctness. We cannot prohibit Y from checking X in that case (Remember that $X \rightarrow anti\ Y$ means Y cannot check X, which is opposite to what $Y \rightarrow check\ X$ *means).* So an anti-constraint $X \rightarrow anti\ Y$ can be enforced only when there is no $Y \rightarrow check\ X$. At last, since only a memory operation X with P bit may be checked by a memory operation Y with C bit, an anti-constraint $X \rightarrow anti\ Y$ is needed only if X has P bit and Y has C bit. In summary, we compute the anti-constraints $X \rightarrow anti\ Y$ with the following condition:

[ANTI-CONSTRAINT] $X \rightarrow anti\ Y$ if:

- $X \rightarrow dep\ Y$ and,
- X precedes Y after scheduling and,
- there is no $Y \rightarrow check\ X$ and,
- X has P bit and,
- Y has C bit.

As an example in Figure 8 (b), we will derive the anti-constraint $M_2 \rightarrow anti\ M_3$, with ANTI-CONSTRAINT. There is no anti-constraint $M_1 \rightarrow anti\ M_3$ due to $M_3 \rightarrow check\ M_1$. There is also no anti-constraint $M_0 \rightarrow anti\ M_3$ because M_0 does not have P bit.

5. Alias Register Allocation Algorithm

In this section, we show how to allocate alias registers to memory operations to satisfy all the check-constraints and anti-constraints. These constraints together form a *constraint graph*. First, we show how to perform a fast allocation if there is no cycle in the constraint graph. Then based on that, we show how to handle cycles. We then discuss how to avoid register overflow. Finally, we put together all of these parts into a complete algorithm.

5.1 Fast Alias Register Allocation in Constraint Order

According to ORDERED-ALIAS-DETECTION-RULE, we have:

[REGISTER-ALLOCATION-RULE]

- Given a check-constraint $X \rightarrow check\ Y$, the alias register allocated to X must be no later than the alias register allocated to Y, i.e. $order(X) \leq order(Y)$.
- Given an anti-constraint $X \rightarrow anti\ Y$, the alias register allocated to X must be earlier than the alias register allocated to Y, i.e. $order(X) < order(Y)$.

So the constraints naturally provide the orders for alias registers. If there is no cycle in the constraint graph, we can simply allocate alias registers to memory operations with a topological traversal of the constraint graph:

[FAST ALGORITHM] We use *next_order* to keep track of the next available alias register (i.e. its order) to be allocated, which is initialized to *0*. We traverse memory operations in a topological order of the constraint graph to allocate alias registers. For a memory operation X, if $P(X)$ is set, we allocate a new alias register order to X with $order(X) = next\_order$ and increase *next_order* by *1*. If only $C(X)$ is set, we just set $order(X) = next\_order$ without increasing *next_order*.

Without alias register rotation, we can directly use $order(X)$ computed in FAST ALGORITHM as the alias register offset to satisfy all the constraints. As an example, for the optimization shown in Figure 7, we can allocate the alias registers in the topological order M_0, M_5, M_1, M_2, M_3, M_4 of the constraint order in Figure 7 (d) and get the alias register allocation as shown in Figure 7 (a).

Alias register overflow happens when an alias register offset is equal to or larger than the physical alias register count. Given $order(X)$, due to the invariance $order(X) = base(X) + offset(X)$, we can minimize $offset(X)$ by maximizing $base(X)$ through alias register rotation. Assume $base(X) = i$. Then all alias registers with orders in the range $[0, i-1]$ are released at the execution of X and can no longer be set/checked by X or any memory operations executed after X. *So* mathematically, we can compute maximum $base(X)$ with the formula:

[MAX-BASE] $base(X) = MIN\ \{order(Y) |\ \forall Y$ *that is X or succeeds X after scheduling}*.

Then we needs to insert rotating instruction "rotate $base(X2) - base(X1)$" between two consecutive memory operations $X1$, $X2$ if $base(X1) < base(X2)$. For the alias register allocation shown in Figure 7 (a), by maximizing $base(X)$, we will get $base(M_5) = base(M_2) = base(M_0) = 0$, $base(M_4) = base(M_1) = 1$, $base(M_3) = 2$, and the register allocation shown in Figure 7 (b).

5.2 Handle Cycles in Constraint Graph

For an optimization that only speculatively reorders memory operations in superblocks, the constraint order cannot contain cycles because all the dependences and hence all the constraints (which are subset of dependences) follow the original program execution order. This is the reason that the alias register allocation in program order can always detect all the aliases without any false positive. However, with the speculative load/store elimination, we introduce extended dependences in the backward execution order of the original program. Hence we may encounter cycles in the constraint graph as shown in Figure 9 (d).

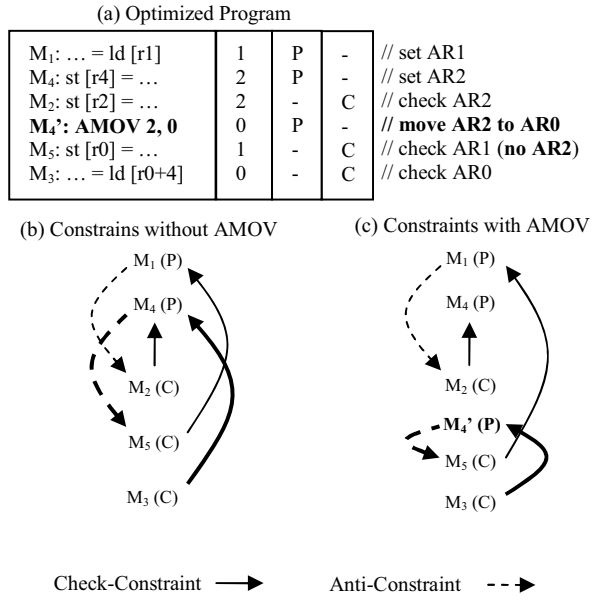


Figure 12: AMOV for the Example in Figure 9

According to REGISTER-ALLOCATION-RULE, no alias register order can satisfy all the constraints in Figure 9 (d). So we need to insert an alias moving instruction *AMOV* to break the cycles. Figure 12 (b) shows the same constraints as Figure 9 (d). Suppose an *AMOV* instruction M_4' is inserted between M_2 and M_5 to move the memory access range from the alias register allocated to M_4 to the new alias register allocated to M_4' (how to choose an alias register to move is explained later in Section 5.4). Then constraints M_3 →*check* M_4 and M_4 →*anti* M_5 need to be changed to M_3 →*check* M_4' and M_4' →*anti* M_5 respectively as shown in Figure 12 (c), as at the execution of M_3 and M_5, the alias access range of M_4 will stay in the alias register allocated to M_4' instead of the alias register allocated to M_4. So the alias moving instruction M_4' breaks the cycles in Figure 12 (b) to get the constraints without cycle as shown in Figure 12 (c). Allocating the alias registers in constraint order in Figure 12 (c) will get the alias register allocation as shown in Figure 12 (a). Instruction M_4' moves the memory access range from *AR2* (i.e. the alias register allocated to M_4) to *AR0* (i.e. the alias register allocated to M_4'). Then M_5 will not check M_4 for aliasing. However, M_3 still can check the memory access range of M_4 in *AR0* for aliasing.

In many cases, the *AMOV* instruction does not really need an additional alias register to save the memory access range, and the *AMOV* often needs merely to clean up the memory access range in the source alias register to avoid false positive. We will quantify this effect in the experiment section.

5.3 Prevent Alias Register Overflow

Since there is no hardware support to spill alias register, we embed our alias register allocation within a list scheduling framework so that we can allocate alias register during the instruction scheduling. In this way, we can dynamically adjust the schedule to prevent alias register overflow. Our list scheduler schedules the instructions in two different modes. When there are enough alias registers and thus no danger of overflow, we schedule instructions in the speculation mode, which speculatively reorders memory operations with alias registers. When the alias register may overflow, we switch to the non-speculation mode, which only rotates alias registers to release unused alias registers without allocating new alias registers. After releasing enough alias registers, we will switch back to the speculation mode.

5.4 Overall Alias Register Allocation Algorithm

The overall algorithm for the alias register allocation is shown as pseudo code in Figure 13, with details discussed in following sections. Note that alias register allocation is integrated with list scheduling, and we omit any necessary steps required by scheduler here.

5.4.1 Algorithm Overview

Initialization (lines 1-3): The algorithm begins with initializing required information. All the dependences are calculated for memory operations. For detecting cycles in the constraint graph, a partial order *T(X)* for each memory operation *X* is initialized to the original program execution order (line 2). The details on cycle detection with *T(X)* are explained later in this section.

Incrementally Building Constraints (lines 8-17): After scheduling a memory operation *Y* in the list scheduler, we incrementally build the constraints by adding all the constraints *X* →*check Y* and *X* →*anti Y* (line 11 and 14).

Once a memory operation *Y* is scheduled, all the dependences *X* →*dep Y* are examined for adding corresponding constraints (line 8). In our list scheduler, the schedule is constructed by filling time slots in an increasing manner. That is, instructions scheduled later than *X* are always placed in the same or later time slots. Since check-constraints are always in a backward direction in the schedule and anti-constraints are always in a forward direction in the schedule, we only need to consider not-yet-scheduled instructions for check-constraints, while only considering already-scheduled instructions for anti-constraints. Lines 9 and 13 show the conditions for adding check-constraints and anti-constraints, respectively.

Cycle Detection (lines 33-54): An incremental cycle detection algorithm [12] is employed for cycle detection in the constraint graph. The algorithm maintains the following partial order *T* for all the instructions to ensure that there is no cycle in the constraint graph.

Invariance: if there exists a constraint X →check Y or X →anti Y, then T(X) < T(Y)

We first initialize *T(X)* for all the instructions to the original program execution order (line 2). Since there is no constraint added yet, the invariance holds at this point. As we add each constraint *X* →*check Y* or *X* →*anti Y*, we check if the invariance *T(X) < T(Y)* is met. If not (i.e. *T(X) ≥ T(Y)*), we take necessary actions so that the invariance is met throughout all the constraints.

```
1   compute all dependences (including extended dependences);
2   initialize all T(X) to the original program execution order;
3   next_order = 0;
4
5   while (there is not-scheduled instruction) {
6       Y = next_instruction(); // next instruction for scheduling
7       if (Y is not memory instruction) continue;
8       for (all dependences X →dep Y) {
9           if (X is not scheduled) {
10              set C(X), P(Y);
11              add X →check Y;
12              if (T(X) ≥ T(Y)) T(X) = T(Y) – 1;
13          } else if (offset(Y) not set, P(X), C(Y), no Y →check X) {
14              add X →anti Y;
15              if (T(X) ≥ T(Y)) detect_cycle(X, Y);
16          }
17      }
18      if (P(Y) or C(Y)) allocate_reg(Y);
19  }
20
21  next_instruction() {
22      min_base = base(head of delay_queue);
23      max_order = next_order + # of Z's with P(Z) in delay_queue;
24      reserved = # of non-scheduled Z's
                        with extended dependences * →dep Z;
25      max_offset = max_order – min_base + reserved;
26      if (max_offset ≥ # physical alias registers)
27          schedule Y without speculation;
28      else
29          schedule Y with speculation;
30      return Y;
31  }
32
33  detect_cycle(X, Y) { // for X →anti Y
34      delta = T(X) – (T(Y) – 1);
35      set H = instructions reachable from Y in constraint order;
36      if (X not in set H) { // no cycle detected
37          for (all instruction Z in H)
38              T(Z) = T(Z) + delta;
39      } else { // cycle detected
40          insert AMOV(X') before Y;
41          for (all Z →check X, Z not scheduled) {
42              replace Z →check X with Z →check X';
43              set P(X');
44              T(Z) = T(Z) – delta;
45          }
46          remove X →anti Y;
47          if (P(X')) {
48              add X' →anti Y;
49              T(X') = T(Y) – 1;
50              add X' to delay_queue;
51              base(X') = next_order;
52          }
53      }
54  }
55
56  allocate_reg(Y) {
57      base(Y) = next_order;
58      if there is no *→check Y or *→anti Y
59          add Y into ready_queue;
60      else
61          add Y into delay_queue;
62      while (ready_queue not empty) {
63          X = ready_queue.dequeue();
64          offset(X) = next_order – base(X);
65          if (P(X)) next_order++;
66          for (X →check Z or X →anti Z) {
67              delete X →check Z or X →anti Z;
68              if (there is no *→check Z or *→anti Z)
69                  add Z into ready_queue;
70          }
71      }
72      if (next_order > base(Y))
73          insert rotate instruction after Y by next_order – base(Y);
74      dequeue all X at the head of delay_queue if offset(X) is set
75  }
```

Figure 13: Alias Register Allocation Algorithm

For a check-constraint $X \rightarrow check\ Y$, we simply decrease $T(X)$ to $T(Y) - 1$ (line 12) to ensure the invariance. Since X is not scheduled yet, there is no constraint $* \rightarrow check\ X$ or $* \rightarrow anti\ X$ yet. Therefore, this action will neither break any existing invariance conditions nor create any cycle in the constraints.

However, adding an anti-constraint $X \rightarrow anti\ Y$ requires more complicated step (*detect_cycle()* in line 33), since it can create a cycle. We first build a set H of instructions that are reachable from Y in the constraint order (including Y itself). If X is not in H, there is no cycle created and we simply increase the order numbers of Y and its connected component (set H) as in line 37-38. When a cycle is detected (X is in set H), we break the cycle by inserting an *AMOV* instruction (X') just before Y (line 40). After the execution of X', the memory access range for X is transferred to the alias register allocated to X'. So, all the instructions not yet scheduled (Z) should check X' instead of X (line 42). We also need to remove the original anti-constraint $X \rightarrow anti\ Y$ (line 46) and add a new one $X' \rightarrow anti\ Y$ if $P(X')$ is set (line 48). Finally, X' is put into the *delay_queue* since it is not ready for allocation yet.

Allocation (lines 56-75): First, we remember the base pointer at Y execution in *base(Y)*. This is used to calculate the offset of the allocated register (*offset(Y)*). We maintain two FIFO queues for allocation: *ready_queue* and *delay_queue*. If Y has no constraints $* \rightarrow check\ Y$ or $* \rightarrow anti\ Y$, Y is ready for allocation and put into *ready_queue*. Otherwise, allocation is delayed by putting it into *delay_queue*. For instructions X in *ready_queue*, a simple allocation is performed as in line 64-65. When the alias register for an instruction X is allocated, we delete all the constraints $X \rightarrow check\ Z$ or $X \rightarrow anti\ Z$. This could make other instructions (Z) in *delay_queue* ready for allocation and we can continue with them. Due to the delayed alias register allocation, an alias register is allocated only when the last instruction that uses it is scheduled (i.e. the alias register can be released after the scheduled instruction). So we rotate the base pointer after the scheduled instruction with the amount of newly allocated registers (line 72-73) and dequeue allocated memory operations from the head of delay_queue (line 74).

Preventing Overflow (lines 21-31): To avoid overflow, we need to ensure that *offset(X)* should not equal to or exceed the number of physical alias registers. We bound the maximum possible offset with the following three numbers.

First, we estimate the minimum possible base pointer (line 22). Then the maximum register order (*max_order*) is estimated. In addition to the allocated registers (*next_order*), each instruction Z with *P(Z)* set in *delay_queue* requires a new alias register so we estimate that they each will use a separate alias register (line 23). Thirdly, we need to consider the future register usage due to not-yet-scheduled instructions. Even if we follow the data dependences without speculation, we might still need more alias registers for extended dependences introduced by load/store eliminations (line 24). The maximum possible offset is conservatively calculated (line 25) and it is compared against the number of physical alias registers to determine the memory reordering policy in the list scheduler.

5.4.2 Allocation Example

As an example of alias register allocation, let's take a look at the optimization performed in Figure 7. After scheduling M_5, we insert a check-constraint $M_0 \rightarrow check\ M_5$ and delay the allocation for M_5. Similarly, after scheduling M_2, we insert check-constraints $M_0 \rightarrow check\ M_2$, $M_1 \rightarrow check\ M_2$ and delay the allocation for M_2 as well. Then when scheduling M_0, we allocate *next_order 0* (i.e.

offset(M_0) = 0) to M_0 with C bit. After the register allocation for M_0, we delete the check-constraints M_0 →*check* M_5 and M_0 →*check* M_2. Since there is no constraint * →*check* M_5 or * →*anti* M_5, M_5 becomes ready for allocation. Then we allocate *next_order 0* (i.e. *offset(M_5) = 0*) to M_5 with *P* bit and increase *next_order* by *1*. We also insert rotation by *1* after M_0 to rotate the allocated alias register at order *0*. After scheduling M_4, the alias register allocation for M_4 is delayed due to the check-constraint M_1 →*check* M_4 and M_3 →*check* M_4. Since now *next_order* is *1*, *base(M_4)* will get *1*, which indicates that *BASE* is *1* at the execution of M_4. When scheduling M_1, we will allocate *next_order 1* to M_1 with *C* bit and delete the check-constraint M_1 →*check* M_2 and M_1 →*check* M_4. Since *base(M_1)* is *1*, we will *get offset(M_1) = next_order – base(M_1) = 0*. Now M_2 becomes ready for register allocation and we allocate *next_order 1* to M_2 with *P* bit and increase *next_order* to *2*. Since *base(M_2)* is *0*, we will get *offset(M_2) = 1*. After M_1, we insert rotation by *1* to rotate the allocated alias register at order *1*. At last after scheduling M_3, we allocate *next_order 2* to M_3 and M_4. Since *base(M_3)* is *2* and *base(M_4)* is *1*, we get *offset(M_3) = 0* and *offset(M_4) = 1*. The final register allocation will be as shown in Figure 7 (b).

5.4.3 Cycle Detection Example

We take the constraints in Figure 12 (b) (based on the optimization shown in Figure 9) and illustrate how the cycle detection algorithm [12] works in our allocation scheme. We initialize $T(M_1) = 1$, $T(M_2) = 2$, $T(M_3) = 3$, $T(M_4) = 4$ and $T(M_5) = 5$. After scheduling of M_1, we add check-constraint M_5 →*check* M_1 and delay the alias register allocation for M_1. Since $T(M_5) \geq T(M_1)$, we update $T(M_5) = T(M_1) - 1 = 0$. After the scheduling of M_4, we add check-constraint M_2 →*check* M_4 and M_3 →*check* M_4, and delay the alias register allocation for M_4. After the scheduling of M_2, we add anti-constraint M_1 →*anti* M_2 and delay the alias register allocation for M_2. When scheduling M_5, we need to add anti-constraint M_4 →*anti* M_5. However, $T(M_4) \geq T(M_5)$. Then we find the set $H = \{M_5, M_1, M_2, M_4\}$ of instructions that are reachable from M_5. Since M_4 is in set H, a cycle will be created with the adding of anti-constraint M_4 →*anti* M_5. So we insert an alias moving instruction M_4' before M_5 to move the memory access range of M_4. We then replace the check-constraint M_3 →*check* M_4 with M_3 →*check* M_4', add anti-constraint M_4' →*anti* M_5 and delay the alias register allocation for M_4' and M_5. We also set $T(M_4') = -1$ and update $T(M_3)$ to *-2*. The resulting constraints are as shown in Figure 12 (c). At last we schedule M_3 and allocate alias registers to all the instructions in *delay_queue*. The resulting code and alias register allocation is shown in Figure 12 (a).

Table 2: Architecture parameters

Architecture Features	Parameter
8-wide VLIW	2 INT units, 2 FP units, 2 MEM unit, 1 BRANCH unit, 1 ALIAS unit
L1 I-Cache	4-way 256KB
L1 D-Cache	4-way 64KB, HW prefetch
L2 Cache	8-way 2MB, 8 cycle latency, HW perfetch
L3 Cache	8-way 8MB, 25 cycle latency
Memory	1GB, 104 cycle latency

6. Experiments

We evaluate SMARQ in a dynamic optimization framework as shown in Figure 1, which translates and optimizes x86 binary codes into code running on an internal VLIW CPU modeled by a cycle-accurate simulator. The VLIW simulator supports atomic region execution [10, 11] with 64 alias registers. Table 3 lists the important architecture parameters for the VLIW simulator.

Our dynamic optimizer forms superblock regions for optimization similar to [6]. X86 binary code is first executed through interpretation. The system profiles the execution for hot basic blocks. When a hot block is identified (i.e. the execution count of a basic block reaches a threshold for hotness), the dynamic optimizer forms a region along the hot execution paths starting from the basic block until it reaches a cold block (i.e. execution count lower than a threshold for coldness). After region formation, the x86 binary code is translated into an Internal Representation (IR) for optimizations. Table 2 lists the important optimizations. After all the optimizations, the optimizer performs register allocation and then the instruction scheduling algorithm as shown in Figure 13.

Table 3: Important Optimizations

If-conversion	Copy propagation	Loop invariant hoisting
Load elimination	Store elimination	Dead code elimination
Constant propagation	Alias analysis	Common sub-expression elimination

SMARQ focus on a suite of SPECFP 2000 benchmarks, which can form large superblocks for optimization, to show the benefit of SMARQ. Figure 14 shows the average number of memory operations in superblocks.

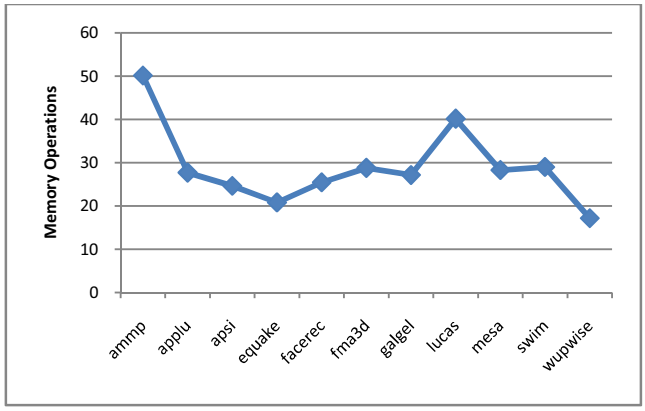


Figure 14: Average Number of Memory Operations in Superblocks

For each benchmark we select 10 code segments for our experiment. The simulated code segments are selected to be representative of the applications. We simulate each segment for 1 billion x86 instructions functionally to warm up dynamic optimizations, 30 million instructions cycle-accurately to warm up micro-architectural state, and 100 million instructions cycle-accurately to collect performance data.

6.1 Performance Improvement

Figure 15 shows the speedups of SMARQ and two other alias detection schemes. The first one (SMARQ16) models an alias detection approach that uses only 16 ordered alias registers (we may refer to it as Efficeon-like since Efficeon scheme can only supports

15 alias registers). The second scheme (Itanium-like) models an alias detection approach that does not consider alias register orders.

The left bar shows the speedup with SMARQ. Overall, optimizations with SMARQ improve the performance by 39% as compared to that without hardware alias detection support. The middle bar in Figure 15 shows the speedup of SMARQ16 with only 16 alias registers. On average, we get 29% speedup, a 10% smaller speedup than SMARQ. For *ammp*, the impact is as high as 30% due to the large number of memory operations in the superblocks, as shown in Figure 14. So the scalable alias register scheme in SMARQ is important for speculative optimization on programs with large superblocks. We believe SMARQ is even more promising for larger region and loop level optimizations. The right bar in Figure 15 shows the speedup with the Itanium-like model. With the non-ordered alias detection, we can only get 26% speedup, a 13% smaller speedup than SMARQ. For *ammp*, the impact is as high as 47%.

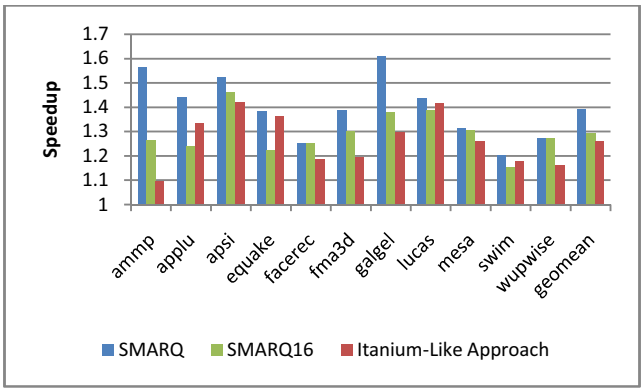


Figure 15: Speedup with Different Alias Detection

Note that, Itanium alias detection may not even achieve the performance in the right bar of Figure 15 because Itanium cannot detect alias between reordered stores. Figure 16 shows the performance impact of disabling store reordering. Without the store reorder in the optimization, on average, we observe 2.6% performance impact. For *mesa*, the impact is as high as 13%. Note that the *ammp* benchmark shows slight performance loss with store reorder. This can happen when the reordered stores alias to each other at runtime and cause execution rollback.

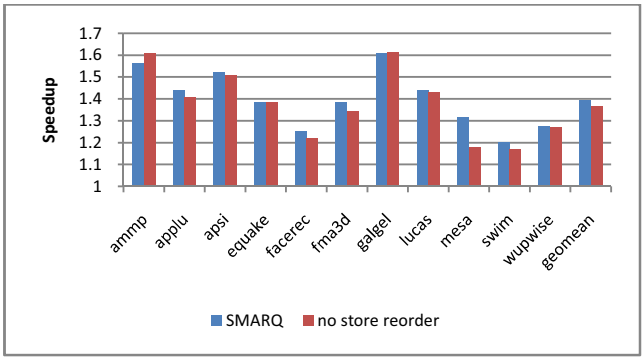


Figure 16: Impact of Store Reordering

6.2 Working Set Reduction

In SMARQ, all the live alias registers stay in a slide window referenced through alias register offsets. The window is from the current BASE to the maximum offset. So the maximum offset + 1 determines the size of the alias register working set. To prevent alias register overflow, the size of the alias register working set cannot exceed the physical alias register count.

The straightforward alias register allocation in program order cannot handle speculative load/store elimination, so we cannot directly compare performance with it. Since supporting aggressive reordering without alias register overflow is the key to performance, we collect statistics for the working set in the alias register allocation, as shown in Figure 17. All the data are normalize to the number of memory operations averaged over all the superblocks, which represents the size of the alias register working set by allocating each memory operation an alias register in program order. The first bar shows the number of memory operations that have P bits, which represents the size of the working set by allocating alias registers in program order to the memory operations that set alias registers. The second bar shows the size of the alias register working set in SMARQ. We can see that SMARQ can reduce the alias register working set by 74% as compared to the straightforward alias register allocation for each memory operation in program order. Even compare to the alias register allocation in program order for only the memory operations that set alias registers (i.e. the first bar), SMARQ can reduce the working set by 25% through rotation.

Given a check-constraint *X* →*check Y*, the alias register set by *Y* needs to be kept alive between the execution of instruction *Y* and instruction *X*. So like traditional register allocation, the maximum number of live-ranges that cross any program point provides a *lower-bound* on the size of the alias register working set in all possible alias register allocations. The last bar of Figure 17 shows that lower-bound. We can see that the size of the alias register working set in our fast alias register allocation with constraint order is close to the lower bound.

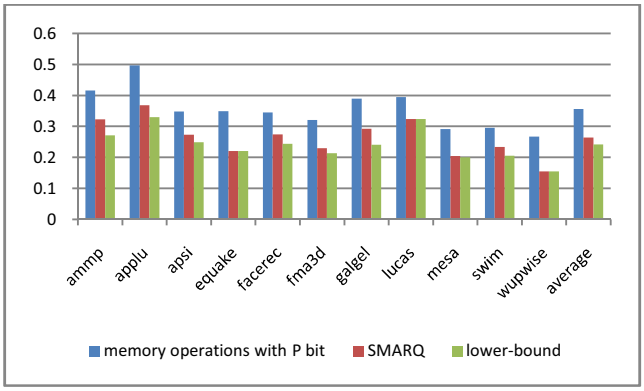


Figure 17: Alias Register Working Set

6.3 Optimization Overhead

To measure the optimization overhead, we put special markers (symbols) around our algorithm implementation. During simulation, the simulator will detect the markers and record the execution time to measure the optimization overheads. Figure 18 shows the translation overheads. The left bar shows the percentage of execution time spent in the overall optimization and the right bar shows the percentage of execution time spent in the scheduling. Overall, only 0.05% of execution time is spent in the optimization. Within it, around half of time is spent in the scheduling, which includes our alias register allocation.

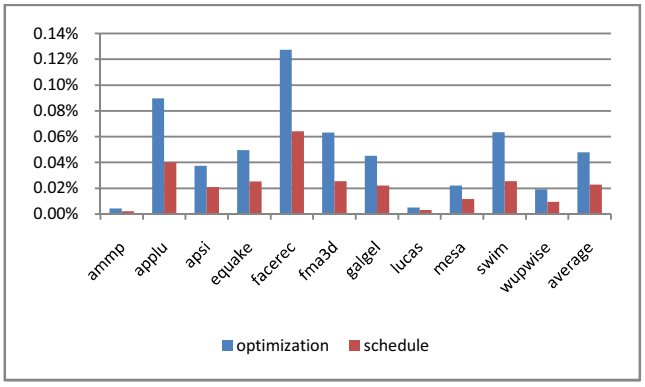


Figure 18: Optimization Overhead

6.4 Alias Register Allocation Statistics

To further show the efficiency of SMARQ, Figure 19 shows the number of constraints normalized to the number of memory operations. On average, for each scheduled memory operation, we insert 1.3 check-constraints and 0.1 anti-constraints. So the constraint graph is very sparse and the number of edges in the constraint graph is close to the number of memory operations.

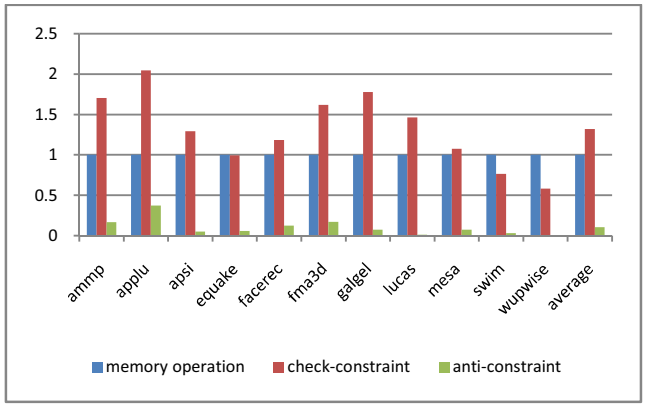


Figure 19: Constraint Count

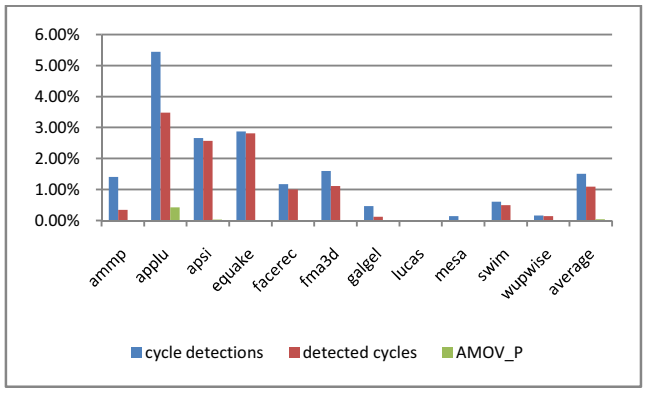


Figure 20: Statistic Data in Cycle Detection

Figure 20 shows the statistics on the cycle detections and AMOV insertions. All the data are normalized to the number of scheduled memory operations. The left bar show the number of cycle detections (i.e. *detect_cycle()* in Figure 13). The middle bar shows the number of cycles detected. On average, only 1.5% scheduled memory operations need to detect cycles and about 1% are inside cycles that need to insert the *AMOV* instructions. The right bar shows the number of inserted *AMOV* instructions that need to set an alias register (i.e. with *P* bit). In majority of the cases, the *AMOV* instruction does not need to set an alias register (i.e. just clean up the memory access range in an alias register without copying it to a new alias register).

7. Related Works

Dynamic binary optimization morphs existing binary programs to improve performance [6,17], save power [16], enhance security, reliability, and parallelism [20]. Architectural supports, such as atomic region [10,11] and alias registers [3,4] are crucially important for aggressive instruction scheduling and speculative optimizations, due to many constraints at binary level [18].

Compiler managed data speculation has been considered an important technique for instruction scheduling [1,3,5,7,8]. DAISY [17] used a load-verify instruction to reload the value of a previously speculated load in program order. Itanium [2] uses ALAT (Advanced Load Address Table) to enable scheduling of load above stores. The load moved above a store is encoded as an advanced load, and in the original load location there is a check-load to verify that no stores have invalidated the advanced load. This scheme requires each store to check all the advanced loads executed before the store and may result in false positives. Also it cannot be used for general instruction scheduling (e.g. move store above aliased store) or optimizations (e.g. store load forwarding, etc).

The Transmeta Effecion [6] uses a mask to selectively check multiple registers without false positive. It can also support scheduling of stores. However, the scheme is not scalable and limits the number of alias registers to the width of the bit-mask.

There is a proposal to use order-based alias detection for instruction scheduling [4]. However, it targets primarily at reordering memory operations. It has serious limitation when apply to other speculative optimizations, such as load/store eliminations. We identified the potential false positive issue and extended hardware support with the novel register move technique to handle potential false positive issues. We developed a novel fast algorithm to allocate the alias registers in constraint order and demonstrated its effectiveness via extensive experiment.

The study in [9] shows that compiler managed data speculation is less useful in a source compiler due to recent advance in static alias analysis techniques, and is significantly more useful in a dynamic binary optimization environment where sophisticated alias analysis is impractical. A simple and fast binary level alias analysis technique was proposed in [13]. However, it can only discover a small subset of aliases, mostly those between direct memory accesses and those indexed by stack-frame registers [14]. The technique proposed in [14] can discover reasonable amount of alias between memory operations indexed by non-stack-frame registers. Although it is useful for static analysis of executable files, it is impractical for dynamic optimizations, as it may take hours to analyze one binary. Speculative alias analysis [1] uses memory region analyses and profile information to increase alias analysis coverage, and use checking and recovery code for correctness. The increased alias analysis coverage can help our speculative optimizations to filter out the likely aliased cases and reduce recovery overhead.

8. Conclusion and Future Work

In this paper, we first studied in details the order-based alias detection mechanism and identified two serious limitations. First, it can potentially generate false positives when it is used to support general speculative optimizations. Second, it imposes serious challenge to the allocation algorithm and a straightforward allocation in program order unnecessarily uses too many alias registers. We extended the mechanism with a simple alias-move instruction to avoid potential false positive, and develop a novel fast allocation algorithm to use the alias registers not only in instruction scheduling, but also in speculative optimizations. The new algorithm can avoid all the potential false positives so all the aliases detected will affect the optimization correctness. Our experiments show that dynamic binary optimization on a VLIW processor with 64 alias register can improve the overall performance by 39% as compared to the optimization without alias register. Our software-managed order-based alias detection can reduce the alias register working set by 74% over the straightforward allocation in original program execution order.

This work can be extended in several directions. First, we have informally argued that register "rotation" is the preferred method for reusing order-based alias registers. It would be interesting to prove that the traditional lifetime analysis may not be able to achieve better register usage efficiency than our SMARQ algorithm for order-based alias registers. Second, our alias register allocation algorithm is integrated with a list scheduler. We may try to integrate the allocation algorithm with other instruction scheduling techniques, such as software pipelining, etc.

References

1. Manel Fernández , Roger Espasa, Speculative Alias Analysis for Executable Code, Proceedings of the 2002 International Conference on Parallel Architectures and Compilation Techniques, p.222-231, September 22-25, 2002
2. Crawford, J., "Introducing the Itanium Processors." IEEE Micro, Volume: 20 Issue: 5, Sept.-Oct. 2000, Page(s): 9 –11
3. Gallagher, David M., Chen, William Y., Hwu, Wen-mei W., "Dynamic Memory Disambiguation Using the Memory Conflict Buffer," ACM SIGPLAN notices, NOV 01 1994 v 29 n 11, Page: 183
4. Holscher, Brian; Rozas, Guillermo; Van Zoeren, James; Dunn, David, "Systems and methods for reordering processor instructions," United States Patent 7634635
5. Huang, A. S., G. Slavenburg, and J. P. Shen, "Speculative Disambiguation: A Compilation Technique for Dynamic Memory Disambiguation, "In *Proc.of the 21st Annual Int'l Symp. on Computer Architecture*, pp. 200-210, April 1994.
6. K. Krewell, "Transmeta Gets More Efficeon", *Microprocessor report*. v.17, October, 2003.
7. Jin Lin , Tong Chen , Wei-Chung Hsu , Pen-Chung Yew, Speculative register promotion using Advanced Load Address Table (ALAT), Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization, March 23-26, 2003
8. Nicolau, A., "Run-time Disambiguation: Coping with Statically Unpredictable Dependencies," *IEEE Trans. on Computers*, Vol. 38, No. 5, pp. 663-678, May 1989.
9. Y. Wu, Li-Ling Chen, R. Ju, J. Fang, "Performance potentials of compiler-directed data speculation," ISPASS-2003, IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'03)
10. N. Neelakantam, R. Rajwar, S. Srinivas, U. Srinivasan and C. Zilles, "Hardware atomicity for reliable software speculation", ISCA 2007.
11. M. Herlihy and J. E. B. Moss, "Transactional memory: Architectural support for lock-free data structures", In Proceedings of *the 20th annual International Symposium on Computer Architecture* (ISCA) 1993.
12. Bernhard Haeupler, Telikepalli Kavitha, Rogers Mathew, Siddhartha Sen, Robert Endre Tarjan, "Incremental Cycle Detection, Topological Ordering, and Strong Component Maintenance", ICALP-2008
13. Saumya Debray, Robert Muth, Matthew Weippert, "Alias analysis of executable code", POPL 1998.
14. Thomas Reps , Gogul Balakrishnan, "Improved memory-access analysis for x86 executables", *Compiler Construction*, 2008
15. W. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery, "The Superblock: An effective technique for VLIW and superscalar compilation," *The Journal of Supercomputing*, vol. 7, 1993
16. R. Rosner, Y. Almog, Y, M. Moffie, N. Schwartz and A. Mendelson, "Power Awareness through Selective Dynamically Optimized Frames", ISCA 2004.
17. K. Ebcioglu , E. R. Altman, "DAISY: dynamic compilation for 100% architectural compatibility", ISCA 1997.
18. C. Wang, Y. Wu, "Modeling and Performance Evaluation of TSO-Preserving Binary Optimization", PACT-2011
19. J. Bharadway, K. Menezes, C. McKinsey, "Wavefront scheduling: path based data representation and scheduling of subgraphs", MICRO 1999.
20. C. Wang, Y. Wu, E. Borin, S. Hu, W. Liu, D. Sager, T. Ngai, J. Fang, "Dynamic parallelization of single-threaded binary programs using speculative slicing", ICS 2009

2012 IEEE/ACM 45th Annual International Symposium on Microarchitecture

Profiling Data-Dependence to Assist Parallelization: Framework, Scope, and Optimization

Alain Ketterlin Philippe Clauss
INRIA & Université de Strasbourg (France)
{Alain.Ketterlin,Philippe.Clauss}@inria.fr

Abstract

This paper describes a tool using one or more executions of a sequential program to detect parallel portions of the program. The tool, called Parwiz, uses dynamic binary instrumentation, targets various forms of parallelism, and suggests distinct parallelization actions, ranging from simple directive tagging to elaborate loop transformations.

The first part of the paper details the link between the program's static structures (like routines and loops), the memory accesses performed by the program, and the dependencies that are used to highlight potential parallelism. This part also describes the instrumentation involved, and the general architecture of the system.

The second part of the paper puts the framework into action. The first study focuses on loop parallelism, targeting OpenMP parallel-for directives, including privatization when necessary. The second study is an adaptation of a well-known vectorization technique based on a slightly richer dependence description, where the tool suggests an elaborate loop transformation. The third study views loops as a graph of (hopefully lightly) dependent iterations.

The third part of the paper explains how the overall cost of data-dependence profiling can be reduced. This cost has two major causes: first, instrumenting memory accesses slows down the program, and second, turning memory accesses into dependence graphs consumes processing time. Parwiz uses static analysis of the original (binary) program to provide data at a coarser level, moving from individual accesses to complete loops whenever possible, thereby reducing the impact of both sources of inefficiency.

1. Introduction

The last decade has seen a radical evolution of programming practice towards parallelism. Processor clock frequencies have practically reached their physical limits, newer processors include more and more cores, and parallelism is now vital for old and new applications. However, parallel programming is much more difficult than its sequential counterpart, and productivity and correctness are much more difficult to achieve in a parallel world. Moreover, parallelism can take highly diverse forms, using constructs like parallel loops, threads, futures, and so on. The choice of a specific implementation "language" has a tremendous impact on the quality of the application.

There have been many attempts at automating the detection and use of parallelism in programs. Two distinct approaches have dominated the research in this domain. The first approach consists in building compilers that are able to parallelize programs. Sophisticated and powerful techniques have been developed, but their scope is severely restricted by the requirements they put on the programs they are able to handle. The second approach consists in leaving the work to the execution environment, with perhaps some help from the compiler. In this case, efficiency is seldom guaranteed, and specific hardware, e.g., a transactional memory, is usually required. For all these reasons, it is not unreasonable to consider that parallel programs will continue to be written by hand. Therefore, programmers will need tools to assist them in their task.

Commercial tools are already available to help a programmer parallelizing his program. A good example of such a tool is Intel Advisor[1]: it lets the user insert annotations to "sketch out potential parallelism", generates the corresponding code, and monitors program execution to detect data-races and to evaluate the resulting gain in performance. A paper by Kim and colleagues [11] has a very good explanation of why this is insufficient in practice: the difficulty is much more in locating the parallelism than in implementing it, and Intel Advisor focuses on the latter, but is of no help in the former. Several research works have aimed at filling the gap: Prospector [11], SD3 [12], and Kremlin [6] are some of the most recent tools designed to assist in the discovery of parallelism in sequential programs.

This paper describes a new tool called Parwiz that provides hints to the programmer, pointing out parallel sections in a sequential program and suggesting how parallelism should be implemented. The basic idea is that the programmer provides a representative set of input data. The tool then observes one or more executions of the program on these data and collects information on the data dependencies exhibited by the program. Using these dynamic, empirical dependencies, various parallelization strategies are tentatively applied to suggest adequate transformations of the program. Since conclusions are drawn from data samples, only the programmer can validate the resulting suggestions, and choose to implement them, or not. Tools that facilitate the implementation phase already exist, where the developer can select abstract strategies and let the tools generate the corresponding code. Our goal is to describe a tool that would suggest a set of valid strategies to help the programmer.

The main characteristic of Parwiz is that it relies exclusively on data dependencies. These dependencies are obtained by instrumenting the original program, and are described with positions in the program execution. They are carried by static program structures like loops and functions, at various levels of resolution. Section 2 details the dynamic dependence analysis framework, regarding dependence capture and collection, as well as all data structures kept by the analyzer. The system is designed to work directly on executable code, it is therefore independent of any compiler or library. When collecting dependencies, it associates them to program structures (functions and loops), and accumulates them across a single execution, across a complete program execution, or even across several executions of the same program on different input data sets. Such an accumulated set of dependencies can then be submitted to different parallelism analysis algorithms to obtain a description of parallel sections, or even transformed code.

Building on this general framework, this paper makes two main contributions:

1. it shows that Parwiz can be used for various kinds of parallelization tasks, covering the discovery of parallel loops but also paral-

[1] See `http://software.intel.com/en-us/intel-advisor-xe`.

1072-4451/12 $26.00 © 2012 IEEE
DOI 10.1109/MICRO.2012.47

IEEE computer society

lelization strategies leading to complex program transformations;
2. it describes two different optimizations strategies, memory access coalescing and parametrized loop nests extraction, that both reduce the cost of dynamic data-dependence profiling without affecting its accuracy.

Section 3 gives several example analysis techniques, including detecting parallel loops (where OpenMP can be used to implement a parallel version), loop nest transformations (applying a well-known vectorization algorithm), and task-parallelism.

Section 4 then focuses on the cost of profiling a program and extracting its dependencies. Two distinct strategies are suggested to reduce the cost of memory tracing and dependency modeling. Both rely on the regularity and locality one may expect in programs. This work is not complete yet, but we think it can be useful in other forms of memory profiling. We illustrate its use in Parwiz.

2. A Data-Dependence Profiling Framework

The goal of Parwiz is to capture all data dependencies happening at run time, and to build from that a dependence graph covering all sequentiality constraints between program structures, at various levels of resolution.

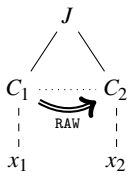
The relevant program structures are single instructions, loops and their iterations, and routine calls. Individual instructions in the binary code are used to to hold the location of individual memory accesses. These structures are arranged into a static hierarchy in the program code (an extended form of the call-graph). During execution, this static hierarchy unfolds into an *execution tree*, where loops "fan out" into the iterations they perform.

After an initial warm-up, every memory access may lead to a data-dependence. The relative execution points of the current access and the previous one(s) at the same section of memory completely describe the dynamic data-dependence. This pair of positions can then be "projected" back onto the static call-graph, where it is merged with other (static) dependencies to lead to the final dependence graph.

2.1. Execution Points and Data-Dependence

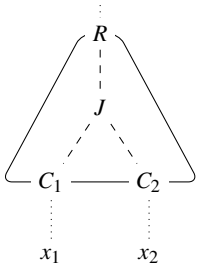
To detect data dependencies, Parwiz maintains an extensive execution tree. This tree holds all information about the current execution points, as well as past execution points as long as they may be required to correctly describe dependencies. The tree holds several kinds of nodes: CALL and LOOP nodes represent their respective type of program structures, ITER nodes (to be found only immediately under LOOP nodes) represent individual loops iterations, and ACCESS nodes represent individual memory accesses and are leaves of the execution tree. Every node is identified by its program address, and has a "position" relative to its parent node. On ITER nodes the position is an iteration number, and on other nodes it is an abstract position consistent with the textual position of the corresponding program structure.

Every memory access is uniquely identified by the path starting at the root of the tree (which by convention is a CALL node) and going down to the corresponding ACCESS node. This path alternates CALL nodes and LOOP-ITER pairs of nodes. The sequence of positions on the path is the global iteration vector of that access. Given two memory accesses x_1 and x_2 hitting the same address (or overlapping ranges of addresses), one can compute the least common ancestor of both accesses, i.e., the node where both execution points join, labeled J on the picture. Nodes C_1 and C_2 are immediate children of node J.



We say J "carries" the dependence, and nodes C_1 and C_2 are source and sink of the dependence. For instance, if J denotes a loop, C_1 and C_2 are both iterations (not necessarily consecutive). Section 3.1 describes an experiment collecting dependencies between loop iterations. In the general case, J, C_1, and C_2 could be nodes of any type: for instance, J and C_1 could be calls and C_2 a loop. A dependence is labeled by its type, indicating what kind of accesses its source and sink perform. We use the usual notations (e.g., RAW for *read-after-write* on the picture).

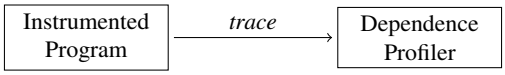
In many cases, one may want to capture data-dependence across several levels of the execution tree, for instance inside a deep loop nest, or across a certain set of routine calls. To account for this, we adopt a slightly more general set of definitions. In the general case, data-dependence profiling in Parwiz considers *tiles* covering parts of the execution tree. A tile has a single root node R, and covers a subset of the tree with the property that any node in the tile has either all or none of its children in the tile. Tiles are either defined implicitly (e.g., rooted on all LOOP nodes that are immediate children of a CALL node, down to the first CALL or ACCESS nodes), or explicitly, as a user-selected portion of the static call-graph. Whenever a dependence appears between accesses x_1 and x_2, their join-point J is computed, and the dependence is added to all tiles containing J. The description of the dependence includes the iteration vectors of both accesses, restricted to their sections that belong to the tile.



In Parwiz, tiles are allowed to overlap. The tile is the major unit of dependence analysis: tiles collect dependencies, and when the execution of a tile is finished its dependence graph is computed. The kind of information retained in the graph depends on the type of the tile, and can be set on a per-tile basis. We will see several examples of tile types below, in section 3.

2.2. Instrumentation and Profiling Architecture

To put the framework just described into practice, Parwiz needs to instrument the sequential program under study to obtain a trace of all the relevant program actions. This trace of events is then passed to a profiling component that maintains data structures representing the relevant part of the execution tree, a shadow memory, and the various dependence graphs. Both the instrumented program and the profiling component could be integrated into a single executable. For practical reasons, we have preferred to keep them separate and let them communicate over a standard channel, e.g., a pipe:



Some processing time is saved by letting both components run in parallel and synchronize on their communication channel. Traces of events can also be saved, and different types of dependence analysis can be run without relaunching the program.

The instrumented program provides a trace of events signaling the execution of basic program structures (routines, loops and iterations,

Main algorithm:

On control-flow events (CALL, ...):
update the current execution path

On memory access events (ACCESS, ...):
(0) create a node x_n for the access at address a
(1) lookup the previous conflicting access x_o to a
(2) locate the latest common execution point p
(3) for each tile containing p:
(4) record a data dependence in the table associated with the root of the tile

Core data structures:

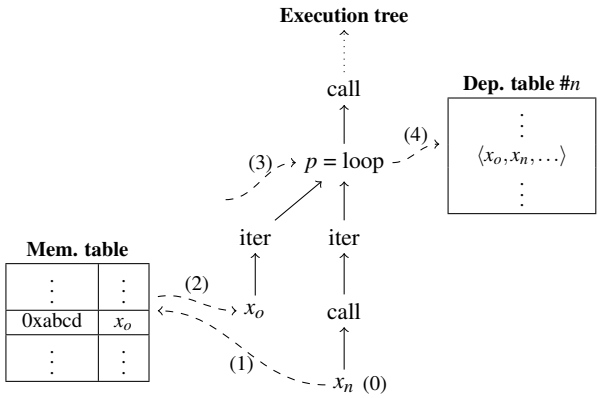


Figure 1: The process of data-dependence profiling.

and memory accesses). Parwiz performs instrumentation on the binary code, without using source code (debugging information is used to provide source code locations to the user). This makes Parwiz independent of any compiler infrastructure, and lets it instrument libraries for which no source code is available. The parsing component analyzes routines individually, extracts a control-flow graph, a dominator tree, and a loop hierarchy [21]. We use the Xed library to parse the binary, and Pin to do the instrumentation [17].

Once the loop hierarchy is built, the code is instrumented to output events on routine entry and return, on loop start and end, on iterating (i.e., following a back-edge), and on accessing memory. Some systems calls, like `malloc` and `free` and similar utilities, are also wrapped to emit necessary information about address space management. The various types of events are the following:

CALL id Routine *id* has just been called, starts executing;
ENDCALL The current call ends;
LOOP id Loop *id* starts executing: this event is emitted once before the first iteration of the loop;
ENDLOOP The current loop terminates;
ITER A new iteration of the current loop starts;
BLOCK id Execution of basic block *id* starts;
ACCESS id mode size addr Instruction *id* loads/stores (indicated by *mode*) *size* bytes starting at address *addr*;
MALLOC addr size Address range [*addr*,*addr*+*size*) has become available to the program;
FREE addr Range of addresses starting at *addr* becomes unavailable.

New types of events will be added later, in Section 4. The BLOCK event type is generated optionally. It is mostly useful to wrap static or dynamic quantitative data, e.g, an instruction mix or number of cache misses. Since this paper doesn't cover quantitative analysis, we do not give more details on this.

2.3. Algorithm and Core Data Structures

The main profiling algorithm processes a trace of events and detects data dependencies as they appear. Events describing control-flow are used to maintain an execution tree, the right edge of which is the currently active stack of function calls and loops. Events representing memory accesses require looking up the previous access to the same range of addresses, from which a description of a new data-dependence is derived, and then updating internal data-structures. The algorithm is sketched on Figure 1.

The profiling component maintains three major data structures (see Figure 1). The first one, the *execution tree*, keeps enough of the dynamic execution tree to be able to detect and describe dependencies. It is actually maintained through a stack containing the current execution path. Getting events for routines entries, loop starts, and iterations all push a new node, or "frame", onto the stack; events for routine returns, end of loops and iterations all make a node be popped from that stack. The stack is actually "cactus-shaped": popped nodes continue to exist as long as they are referenced from inside the memory table. Even though, theoretically, the execution tree may grow very large, there are several ways to keep its size within limits. First, unused tree nodes are immediately collected (via reference counting). Second, at any time, the tiles that lie on the currently executing path precisely delimit the set of nodes that may me used in a future dependence; the others can be removed, making the tree look like a comb more often that not. In practice, this simple garbage-collection strategy has been sufficient to avoid any kind of excessive memory overhead.

The second data structure is the *memory table*, which is indexed by memory addresses. It is in fact a shadow memory, holding a pointer to the latest execution point for each interval of addresses, along with a single bit to indicate whether this access was a read or a write. Conceptually, the memory table is organized similarly to page directories/tables used in MMUs, as a sparse hierarchy of tables. In practice, it is implemented as a combination of a lexicographic tree and an interval tree. Consecutive entries with the same contents are coalesced into a single entry.

The third data structure maintained by the profiler is the *dependence table*. This name actually covers many different types of data structures. A dependence table is attached to each node that is the root of a tile in the execution tree. Its exact nature depends on the desired analysis: in simple studies on loop parallelism as in Section 3.1, a couple of bits may be enough. In more elaborate studies, e.g., the vectorization algorithm used in Section 3.2, a dependence table contains a set of tuples containing references to instructions and iteration vector abstractions. In this section we will consider dependence tables as abstract objects supporting two operations:

1. adding a (fully described) dependence, and
2. drawing a conclusion on the parallelism present in the program structure to which it is attached.

The next section presents several examples of dependence table types. New types can easily be added to Parwiz.

Program	Executed		OpenMP-annotated loops				Dynamic		Slowdown/overhead		
	#Loops	#Par.	#Loops	#Par.	Main cause of failure	#Priv.	#Acc. (10^9)	Dur. (sec)	Trace (×)	Prof. (×)	Mem. (Mb)
312.swim_m	26	25	8	7	reduction	7	92	149.39	33	118	2527
314.mgrid_m	58	52	12	11	reduction	11	163	216.32	39	147	1376
316.applu_m	168	135	30	17	priv. + reduction	25	115	128.12	48	148	1082
318.galgel_m	541	455	37	30	priv. required	30	107	176.05	42	121	1394
320.equake_m	73	67	11	3	priv. required	10	58	73.88	43	150	723
324.apsi_m	191	147	28	13	priv. + reduction	27	132	196.12	44	134	4798
326.gafort_m	58	43	9	7	priv. + reduction	7	29	74.18	35	93	679
328.fma3d_m	233	192	29	22	reduction	22	129	203.59	42	99	2223
330.art_m	79	65	5	4	(non-openmp code)	4	1	5.77	34	92	200
332.ammp_m	76	48	7	5	priv. required	7	87	178.46	37	97	504

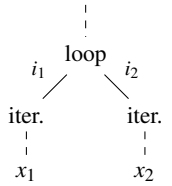
Table 1: Results of boolean dependence analysis on SPEC OMP-2001 programs

3. Scope and Applications

3.1. Loop Parallelism and OpenMP

Our first experiment aims at detecting trivially parallel loops, i.e., loops whose iterations can be executed in parallel without any need for privatization, scalar expansion, or any kind of transformation. The idea is to detect when two distinct iterations of the same loop access the same address in a conflicting way, and to flag the loop when this happens.

Parwiz implements this task by placing a tile rooted at each loop and extending one level down to cover the loop's iterations. Each loop node has an attached dependence table, in Parwiz parlance, which is restricted to a single boolean flag, initially set to true. Any detected dependence carried by a loop makes the associated boolean false. A loop is parallel if all its executions have been found parallel. At the end of execution, the boolean indicates whether the loop is trivially parallel.



We have applied this strategy to the SPEC OMP-2001 benchmark suite. The source programs contain OpenMP pragmas, which we ask the compiler to ignore, but which will provide "ground truth", i.e., a reference parallelization to which we are going to compare the result of a dynamic dependence analysis. All programs have been compiled with `gcc` version 4.4.3 with option `-O2` (compiling with `-O3` provides similar results, but makes it harder to interpret results). The programs were statically analyzed offline to locate functions, loops and memory accesses, and then instrumented and run sequentially to completion on their train input data set. Parwiz worked directly from the binary program, and debug information was used to display analysis results in terms of program source code files and lines.

Table 1 shows the results of the experiment. For each program we indicate the total number of loops executed at least once (#Loops), and the number of loops that have been discovered trivially parallel by the analyzer (#Par.). However, since source code is annotated with OpenMP pragmas, we have also performed a detailed analysis of the loops that were marked parallel by the programmer. The second set of columns shows first the number of annotated loops in source code (#Loops), then how many loops have been found parallel by Parwiz among these (#Par.). A short remark gives the major causes of failure: some loops carry a reduction operation or other dependent operations, and some have specific implementations used when OpenMP is disabled. However, the vast majority of failures comes from the need to privatize data before parallelization.

Since privatization is such a major cause for failure, we have slightly adapted our setting to account for this. The idea is the following: loops that only carry dependencies of types WAR or WAW can be parallelized by privatizing the data involved. Therefore, dependence tables attached to loop nodes keep a set of observed dependence types (i.e., RAW, WAR, or WAW), instead of a single boolean, and can also store a set of program locations where these dependencies occur. This strategy works because Parwiz detects a dependence between an access and the *last* conflicting access during program execution: if an iteration has a dependence of type WAR or WAW with a previous iteration, it simply means that it reuses some memory location for its own usage, without really restraining parallelization. Results are shown on Table 1 in Column #Priv. Basically, reductions remain the last cause of failure, all other loops have been recognized. If the dependence table stores code locations of the dependence-triggering accesses, debug information can be used to give hints on which variable is responsible for the false dependence.

Table 1 also provides quantitative data about the actual executions, like the number of memory accesses performed by the program, which is also the number of addresses processed by Parwiz (#Acc, in billions), and the duration of the execution of the uninstrumented program (Dur, in seconds). The column labeled Trace gives the slowdown of the instrumented program, whereas Prof. gives the overhead incurred by Parwiz. The latter is also the overall slowdown, since the program and Parwiz run in parallel. All slowdowns are given as multiples of the bare program execution time (Dur), and include time spent in I/O. Dynamic data-dependence profiling typically slows the program down by a factor ranging from 100 to 150: reducing this overhead is the topic of Section 4. Finally, the last column (Mem) in Table 1 shows the overall memory usage of Parwiz alone: the requirements are reasonable (except maybe in one case), mainly because Parwiz uses a single shadow memory, and also because of the careful management of the execution tree.

Note that the results given above have been obtained directly from the binary code of the various programs, without any need to manipulate source code. Also, some of the loops that have been discovered parallel contained functions calls, used pointer-based data structures or indirect addressing inside arrays. Dynamic dependence analysis with Parwiz is insensitive to programing language idiosyncrasies.

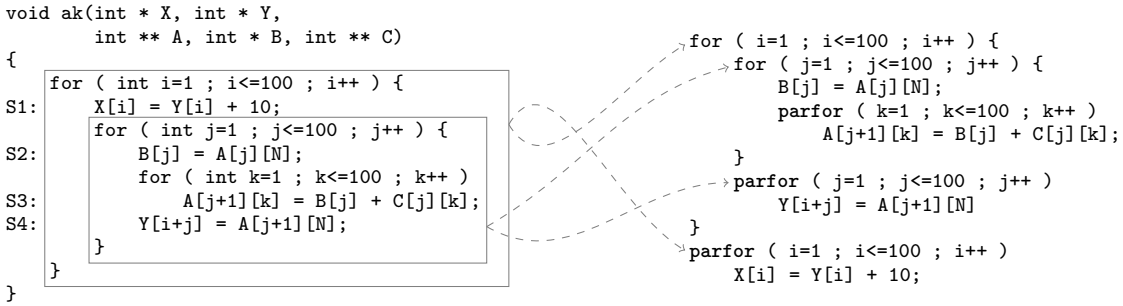
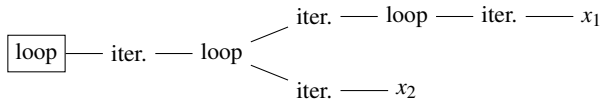


Figure 2: Code example for Section 3.2: the original code is on the left, the result of Allen & Kennedy's *codegen* algorithm is on the right. Arrows highlight loop transformations, `parfor` indicates vectorized loops.

3.2. Program Transformations

The goal of our second experiment is to illustrate the use of a more elaborate dependence abstraction, by considering dependencies across levels of the execution tree. This experiment also illustrates how the result of dynamic dependence analysis can be submitted to standard parallelization algorithms to suggest program transformations.

In this experiment, a tile can be placed on any loop, either implicitly or based on user selection. The tile extends down to all memory accesses and calls, i.e., we do not consider loops that extend across function boundaries. This means that a path between the common ancestor of two execution points and the nearest tile root may contain several loop nodes, as on the following figure, where x_1 and x_2 are memory accesses, and the boxed loop is the root of the tile:



If x_1 and x_2 access the same address, the dependence will be carried by the boxed loop, with a dependence level of 2 (the depth of the common ancestor of both execution points, the root being at depth 1). Hence, the dependence table attached to a tile's root will collect dependencies described by their end points, a type, and a dependence level. At the end of the run, dependencies of a given tile are collected into a dependence graph.

The left side of Figure 2 shows the loop nest used to illustrate dependence-level based dynamic dependence analysis (we will come back to the right part in a moment). This loop nest manipulates arrays that come in as parameters of the function named `ak`. Note that a source code compiler cannot apply any transformation to this loop nest, unless it is able to prove that the various arrays and the various rows of the two 2-dimensional arrays `A` and `C` do not alias.

Figure 3 shows the corresponding x86-64 binary code before instrumentation (in Intel syntax). Individual instructions are labeled with the last three hexadecimal digits of their address. The extents of the various loops are shown on the left side of the code. Statements `S1` to `S4` are also shown: they have been located by first selecting the memory stores, and then slicing back along register use-def links until reaching either constants or memory loads (the extraction of use-def links is explained below). It is important to remember that vertices of the dependence graph are individual binary instructions, i.e., *not* source statements.

Figure 4 shows the dependence graph emitted by Parwiz: vertices are the instructions, edges are dependencies, and gray rectangles group instructions into statements. An important remark about this

```
4004fc: mov r12d,0x0
   502: mov r13d,0x2
   508: mov r14d,0x0
   50e: mov ebp,0x4
i  513: mov eax,DWORD [rsi+r12*1+0x4]
   518: add eax,0xa                      S1
   51b: mov DWORD [rdi+r12*1+0x4],eax
   520: mov ebx,r13d
   523: mov r9,r14
   526: mov rax,QWORD [rdx+r9*2+0x8]
j  52b: mov eax,DWORD [rax+0x4]          S2
   52e: mov DWORD [rcx+r9*1+0x4],eax
   533: mov rax,rbp
   536: mov r10,QWORD [rdx+r9*2+0x10]
k  53b: mov r11,QWORD [r8+r9*2+0x8]
   540: mov r11d,DWORD [r11+rax*1]
   544: add r11d,DWORD [rcx+r9*1+0x4]    S3
   549: mov DWORD [r10+rax*1],r11d
   54d: add rax,0x4
   551: cmp rax,0x194
   557: jne 400536
   559: movsxd rax,ebx
   55c: mov r10,QWORD [rdx+r9*2+0x10]
   561: mov r10d,DWORD [r10+0x4]
   565: mov DWORD [rsi+rax*4],r10d       S4
   569: add r9,0x4
   56d: add ebx,0x1
   570: cmp r9,0x190
   577: jne 400526
   579: add r13d,0x1
   57d: add r12,0x4
   581: cmp r12,0x190
   588: jne 400513
```

Figure 3: Binary code (before instrumentation) corresponding to Figure 2

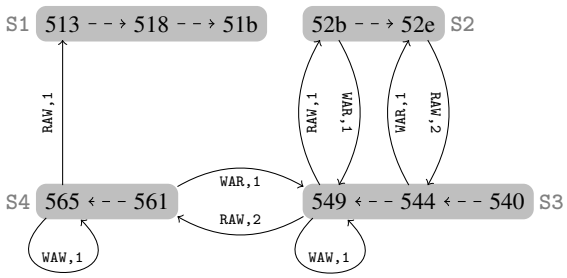


Figure 4: Empirical dependence graph for Figure 2

graph is that is includes "static" dependencies (displayed with dashed lines) along with dynamic dependencies. Static dependencies are use-def links, and represent RAW dependencies between registers (e.g., on register `eax` between instructions `52b` and `52e`).

A dependence graph like the one on Figure 4 can be used to derive an optimal program transformation. The algorithm, named *codegen*, has been devised by Kennedy & Allen for vectorization [8], but can be applied to any kind of dependence graph since it uses only the topology of the graph and the dependence levels labeling its edges. The transformation is optimal in the sense that no algorithm can vectorize (parallelize) more loops given only dependence level information [4]. The result of applying *codegen* to the dependence graph on Figure 4 is shown on the right part of Figure 2, where parallel loops are denoted with `parfor`, and arrows indicate how several loops have been split and rearranged. The reader interested in the details of the algorithm execution can consult the book by Allen & Kennedy [8, Section 2.4.3]: we have kept exactly the same example.

There are several important remarks on the code produced. First, the final loop nest is the result of several complex transformations: loops have been split and rearranged according to the dependence graph. Second, the dependence graph produced by dynamic dependence analysis has fewer edges than "classical" dependence graphs produced by considering all possible pairs of memory accesses. This is, once again, due to the fact that dynamic dependencies automatically highlight dependencies with the *last* conflicting access. However, the complete graph (like the one displayed in Allen & Kennedy's book) can be produced by computing the transitive closure of the dynamic dependence graph. The *codegen* algorithm is insensitive to this aspect of the graph, because it computes strongly connected components.

Extracting dependencies across loop-levels and suggesting complex transformations is, as far as we know, a distinguishing feature of Parwiz.

3.3. Bags of Tasks

Our last experiment illustrates the use of an explicit dependence table. The goal is to use Parwiz to evaluate the potential parallelization of a loop over a linked list. We will not describe a full-scale experiment, but rather use a small example giving rise to a dependence graph between iterations of a list. The code of interest is a loop iterating over a list, merging the two elements at the front of the list and appending the result at the end of the list (in the actual program the list stores graphs, but this doesn't matter here). The target source code loop operates a pairwise merging over the elements of the list:

```
list<Graph*> graphs;
...
while ( graphs.size() > 1 ) {
    Graph * g1 = graphs.front();
    graphs.pop_front();
    Graph * g2 = graphs.front();
    graphs.pop_front();
    Graph * m = merge(g1,g2);
    graphs.push_back(m);
}
```

We would like to know whether there are hidden data dependencies in this loop, and would like to obtain the dependence graph between iterations.

Parwiz can be directed to profile every execution of a given loop. In this case, the tile is rooted at the loop, and extends downwards until reaching either calls or memory accesses. When run with a single title, Parwiz maintains a restricted execution tree: all accesses inside a call (e.g., to `merge`) are directly linked to the call node. In this experiment, we have attached to the tile a dependence table collecting three attributes per dependence: its type (RAW,...) and the two executions points on the lower edge of the tile. Here is an example of the contents of the dependence table:

```
RAW 0x4008d4:CALL(0x400a80) (8,0)
--> 0x40087e:CALL(0x400b18) (3,1)
```

This means that there is a RAW dependence between the call at address 0x4008d4 (calling the routine at 0x400a80—actually `push_back()`) to the call at 0x40087e (calling 0x400b18, i.e., `pop_front()`). A vector like (8,0) indicates that the given program structure is the 9th child of the iteration node, which itself is the 1st child of the loop.

To graphically depict the dependence table, one can create a graph with an (unordered) vertical axis for the various program structures, and another (ordered) horizontal axis for iterations, and then draw an arc for each dependence. When the loop is run with a list containing 16 elements, the result appears in Figure 5 (showing RAW dependencies only). On this graph, solid lines represent dependencies between memory accesses, whereas dashed lines are dependencies between registers. The lower part of the graph represents dependencies between calls to `merge()`. The upper part groups dependencies between list operations (`front()`, `pop_front()`, and `push_back()`)

This graph describes two major aspects of the loop. First, the list handling actions and the processing done on elements are clearly separated, all traffic between the two going through registers and remaining local to an iteration. Hence, any kind of iterator could be used instead of the (inherently sequential) linked list. Second, the processing of the various elements leaves some room for parallelism. The dependence graph between calls to `merge` (at the bottom of the graph) has a critical path length of four, which means that with enough processors four steps are enough to complete the computation. Various other metrics could be used to estimate the load balance between iterations, or the gain given a parallelization scenario. This is out of the scope of this paper.

Before moving to the optimization of Parwiz, we would like to briefly mention two characteristics of the real experiment that inspired this section. First, each iteration actually executes tens of million instructions (in the call to `merge`). This makes almost *any* parallelization opportunity worthwhile, including those that require explicit scheduling control. Second, any single call to `merge` performs several library calls, most of them to functions provided by the GNU Multi-Precision library (GMP). The fact that Parwiz directly handles binary code has saved a lot of work in setting up the experiment.

4. Optimization

Parwiz is made of two subsystems, the instrumented program and the profiling component, that run in parallel and synchronize on a unidirectional communication channel. The time taken to analyze a realistic execution of a program is caused by two distinct, but correlated, factors:

1. the large amount of instrumentation code dilates the original program, making it slower, and leading to a large volume of data output to the communication channel;

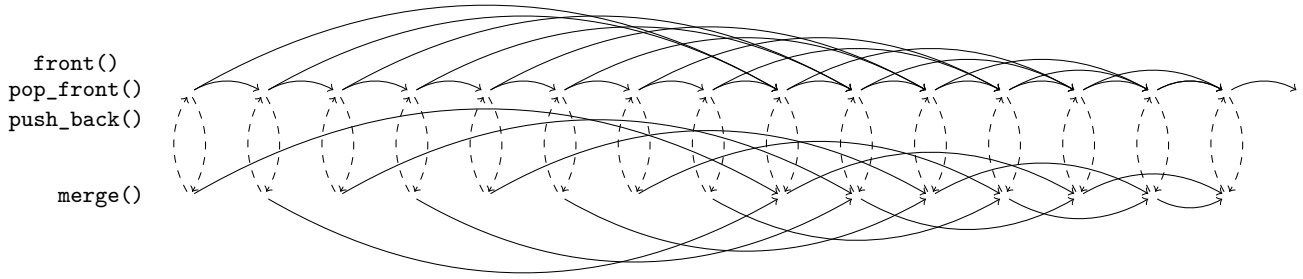


Figure 5: An explicit dependence graph: list handling on top, application-specific processing at the bottom). Solid lines are data-dependencies, dashed lines represent traffic between registers.

2. the processing of events by the profiler takes a significant number of operations, and requires updating the memory table and one or more dependence tables for each memory access.

A first approach to reduce processing time is to sample memory accesses (and control-flow events). Several sampling techniques have been developed, targeting executions paths and/or values (see, e.g., [18]). These techniques reduce the overhead without sacrificing too much accuracy. Unfortunately, sampling does not apply well to data-dependence profiling, for two main reasons. First, a data-dependence is made of two distinct memory accesses. Omitting only one of these is enough to miss a dependence, and therefore introduce spurious dependencies. In consequence, sampling may decrease accuracy exponentially for data-dependence profiling. Second, the omission of a single dependence may completely change the structure of the resulting dependence graph, and then the nature of the hints provided to the user. Since dynamic dependence profiling is already sensitive to the accuracy of input data selection, sampling memory accesses would add another two layers of approximation. We consider that sampling introduce an excessive risk. We are looking for tracing optimizations that fully preserve the accuracy of the traces.

The idea we develop in this section is the following: if several contiguous memory accesses can be considered "atomically", instead of individually, then the time taken for these accesses will be divided by a factor proportional to the number of accesses that have been coalesced. Moreover, if these intervals can be detected statically in the program, leading to the production and processing of a single event, then the benefit will apply to both the instrumented program and the dependence profiler. We focus on two cases where memory accesses can be coalesced. The first works at a small scale, trying to combine nearby individual accesses, typically appearing when the program updates the various fields of a structure. The second works at a larger scale, combining all accesses performed by a loop when the loop appears to be traversing one or more arrays.

4.1. Static Analysis of Binary Code

We have already mentioned in Section 2.2 that Parwiz discovers loops in the control-flow graph. But it actually goes further, performing data-flow analysis on instructions and using the semantics of specific classes of instructions to derive a symbolic model of parts of the program. This section details the various steps of the analysis.

Static Single Assignment Each routine is put into single assignment form (SSA) [3], where each instruction is represented as a transformation between *uses* (input operands) and *definitions* (output operands). All definitions get unique names, and all uses point directly to the corresponding definition. This applies to all machine registers, but also to a special variable `M` representing memory as a whole (every memory store defines a new "version" of memory). In the following, the word "register" designates a definition, i.e., a specific version of a machine register whose value is set at a well-defined location in the code. Figure 6 shows several fragments of code where register names are decorated with their version number.

Forming Symbolic Expressions The second phase follows use-def links and considers instruction semantics to form symbolic expressions for all memory operands, and for as many branch conditions as possible. Starting with every memory operand, expressed in x86-64 as a linear combination of at most two registers, a symbolic expression is formed for each involved register, and the results are combined into the memory operand expression. This recursive process uses opcode semantics to combine expressions. With an instruction set as exuberant as x86-64, this may seem an extremely complex process. It is however reasonably simple for two reasons. First, it considers memory addresses, the computation of which is usually restricted to adding offsets and indexing. Second, it restricts the acceptable expressions to contain only additions, subtractions, and multiplications of registers (remember that "register" means "definition"), and stops the substitution process when reaching an opcode whose semantics goes beyond these operations. Also, the substitution process stops when reaching either the entry point of the routine (which by convention contains a implicit definition of all registers, with unknown value), or an instruction using a memory operand (whose value is, in the general case, hard to track back), or a ϕ-function (which is an SSA abstraction representing the uncertainty on the provenance of the value). An example appears in Figure 6(a), where a memory address gets a symbolic expression formed by combining several basic definitions. Branch conditions are handled in the same way, the branching instruction providing the comparison used, and the instruction setting the `rflags` register providing a symbolic expression. In some cases however, a branch condition cannot be expressed by the kind of expressions we target (e.g., branch on parity), and remains unknown.

Scalar Evolution The third step analyzes ϕ-functions used in symbolic expressions. The goal is to derive a closed form for the ϕ-function, using loop counters. Suppose a ϕ-function $r = \phi(r_1, r_2)$ appears on the head block of a loop, and suppose r_1 is defined outside, before the loop, whereas r_2 is defined inside the loop (thereby providing the updated version of r). If r_2 has an associated expression that is of the form $r + \alpha$, where α involves only loop invariant registers, then r can be expressed as $r_1 + i \cdot \alpha$, where all registers involved are loop-invariant, and i is a normalized loop counter (starting at zero with step one). Then, every occurrence of r inside the loop can be

replaced with $r_1 + i \cdot \alpha$, making the resulting expression dependent only on the loop counter. This step is crucial, because in favorable cases it makes the description of a loop depend only on loop-invariant registers, and introduces the loop counter. Two examples of inductive variable evolution appear in Figure 6(b), illustrating the introduction of loop counters in symbolic expressions.

Loop Trip-Counts The fourth step combines branching conditions inside loop bodies to derive iteration numbers. The idea is to compute the condition under which the control will "come back" to the loop head, i.e., follow one of the back-edges. Using control-dependence, this condition can often be determined even when some internal branch conditions are unknown. If the iteration condition is known, and involves only the loop counter and loop-invariant registers, then it can be turned into a simple upper bound on the loop counter. This is true because we put an additional restriction on scalar evolution: symbolic expressions have to be linear in loop counters, i.e., they are affine combinations of loop counters where coefficients are products of regular registers. Figure 6(c) shows two symbolic branch conditions, one of which "jumps over" a loop, the other being used to determine whether to start a new iteration of a loop. The latter can be used to extract a trip count for the corresponding loop.

4.2. Coalescing Consecutive Accesses

Our first optimization technique consists in detecting a set of accesses to consecutive or overlapping memory places in straight-line code, such that it is legal to consider this set of accesses as a single, large access. To reach this goal, there are two problems to solve: First, detecting that two accesses reach consecutive areas of memory , and second checking for accesses that *may* conflict, to avoid incorrect coalescing. We will restrict the process to sets of accesses appearing in the same basic-block.

Parwiz' static analysis of the code provides symbolic expressions for all memory addresses, and individual instructions provide the mode of access (load or store) as well as the size of the accesses. The sequence of instructions in a basic-block can then be turned into a list of accesses $[(m_1, s_1, a_1), \ldots, (m_n, s_n, a_n)]$, where m_i is either load or store, s_i is the size, and a_i is a symbolic expression of the address. We can make no hypothesis on the values of the registers appearing in the expressions. However, we can hope that two accesses near to each other have a good chance of using the same registers, and that comparing the expressions can in a reasonable number of cases let us conclude on whether they reach consecutive areas.

Comparing two address expressions works as follows. First, if both expressions point to distinct areas, one to the current stack frame and the other to outer memory, the addresses are necessarily distinct. Otherwise, the difference between the expressions is formed and simplified (this is noted $(a_i \ominus a_p)$ in Figure 7). If the result involves registers, then nothing can be said about the value of the difference. Otherwise, the difference is a constant, and if it is zero both addresses are equal.

The next step is to examine each access in turn, and compare it to the accesses that precede it. The goal is to go back as far as possible, as long as the accesses met on the way do not conflict. The backward search stops on a potential conflict (addresses are not distinct and accesses are of different modes), or when a coalescing opportunity has been found (same mode, and equal or consecutive memory ranges). Of course, the search also stops when reaching the start of the basic block, or when crossing a definition of a register

```
0x406ad2 mov r13.8, qword ptr [...]  ; value unknown
0x406af5 mov r15.58, qword ptr [...] ; value unknwon
0x406afd r11.93 = phi(...)           ; value unknown
         ...
0x406b10 rdi.98  = phi(rdi.97,rdi.99)
               ; = r11.93 + i_0x406b10*r13.8

0x406b28 rcx.304 = phi(rcx.303,rcx.302)
               ; = r15.58 + 8*rdi.98 + 8*i_0x406b28

0x406b2e mov rbx.44, qword ptr [rcx.304]
         ; @ r15.58 + 8*r11.93
         ; + 8*i_0x406b28 + 8*i_0x406b10*r13.8
```

(a) Combining symbolic expressions to describe memory addresses

```
0x406ad2 mov r13.8, qword ptr[...] ; value unknown
         ...
0x406afd r11.93 = phi(...)         ; value unknown
         ...
0x406b05 mov rdi.97, r11.93        ; = r11.93
0x406b08 mov esi.244, 0x1          ; = 0x1
```

```
            ...
┌─→0x406b10 rsi.245 = phi(rsi.244,rsi.246)
│             ; = 0x1 + i_0x406b10*0x1
│           rdi.98  = phi(rdi.97,rdi.99)
│             ; = r11.93 + i_0x406b10*r13.8
│           ...
│  0x406b3e inc rsi.246/.245          ; = 0x1 + rsi.245
│  0x406b41 add rdi.99/.98, r13.8     ; = rdi.98 + r13.8
│           ...
└──0x406b4a j... 0x406b10
```

(b) Scalar evolution and artificial loop counters

```
   0x402493 movsxd ebp.2, dword ptr[...] ; unknown
            ...
   0x406b10 test rbp.2, rbp.2
┌──0x406b13 jle 0x406b3e
│           ; if rbp.2 <= 0
│           ...
│ ┌→0x406b28 rax.175 = phi(rax.174,rax.176)
│ │           ; = 0x1 + i_0x406b28*0x1
│ │           ...
│ │  0x406b2b inc rax.176/.175
│ │           ...
│ │  0x406b39 cmp rbp.2, rax.176
│ └──0x406b3c jnl 0x406b28
│             ; if -0x2 + rbp.2 - i_0x406b28 >= 0
└──→0x406b3e ...
```

(c) Branch conditions and control-flow

Figure 6: Static analysis of binary code. Registers carry suffixes representing their SSA names, and `phi`-pseudo instructions are inserted. Symbolic expressions are placed in comments (after `;`). Loop counters are named after the address of their loop, e.g., `i_0x406b10`.

```
COALESCE([(m_1,s_1,a_1),...,(m_n,s_n,a_n)])
— Input: n accesses, with type m_i, size s_i, and address a_i
    let C_i = {(m_i,s_i,a_i)} for all i
    for i = 1 to n
        SEARCH(i)
    for each C_i such that |C_i| > 1
        coalesce all accesses in C_i
SEARCH(i)
    let p = i − 1
    while p ≥ 0 and a_i is defined at p
        if a_i and a_p both point to the same area
            if (a_i ⊖ a_p) is not a constant
                if m_i ≠ m_p then return
            else if (a_i ⊖ a_p) is 0
                if m_i = m_p then merge(C_i,C_p)
                return
            else if m_i = m_p and a_i,a_p consecutive then
                merge(C_i,C_p)
                return
        decrement p
```

Figure 7: Coalescing accesses inside a basic-block

involved in the expression of the address. This is function SEARCH in Figure 7.

Coalescing is implemented by computing classes of accesses. For a given basic-block, each access starts in its own class. Finding a coalescing opportunity during SEARCH simply merges the classes of both accesses. When all accesses have been considered, classes with more than one access are coalesced: it is easy to see that a class contains accesses of the same type, corresponding to consecutive ranges of addresses. The resulting "super-access" event is emitted by instrumenting any access belonging to the class, e.g., the one having the lowest address, and giving it a synthetic size corresponding to the merging of the coalesced address ranges.

To evaluate the effect of coalescing, we use the same set of programs as in Section 3.1 and measure three quantities: the number of events, the time taken by the tracing component, and the time taken by Parwiz. Table 2 shows how much is gained in these quantities, compared to full instrumentation. The first column (Events) shows the proportion of events that can be avoided: the numbers vary substantially between programs, ranging from a negligible amount up to 30%. The second column (Tracing) shows the reduction in tracing time: several factors condition this number but intuitively, we should see here a lower reduction percentage, because avoiding memory access events doesn't change the amount of other types of events. The third column (Profil.) shows the reduction in data-dependence profiling time: this number is roughly equivalent to tracing time, since all events emitted by the tracing component are processed by the profiler. The general conclusion of this experiment is that any saving on instrumentation (around 20% on average) translates directly into the corresponding saving on profiling time, albeit with a difference of 5% on average. Larger variations are due to other software artifacts, such as the sensitivity of the tracing infrastructure to instrumentation position, or the handling of the shadow memory in Parwiz.

4.3. Parametric Loop Nests

After static analysis, symbolic expressions are available for all memory addresses and for some branch conditions. For some loops, an iteration condition has also been found, which can be turned into a number of iterations. An example loop, extracted from the `436.cactusADM` SPEC program, appears in Figure 8 (part of the original code was shown in Figure 6). Back-edge conditions have been turned into loop bounds, branch conditions have been converted into regular conditionals, and memory accesses have an explicit expression. The loop model is turned into C code, and parameters used in either expressions or conditions come in as parameters to the embedding function. Such code can be compiled and, when provided with parameter values, can simulate the execution of the original code.

Program	Events	Tracing	Parwiz
`312.swim_m`	32.61	27.27	24.87
`314.mgrid_m`	5.52	2.56	1.97
`316.applu_m`	26.96	16.67	17.57
`318.galgel_m`	17.76	19.05	13.32
`320.equake_m`	29.31	21.15	19.33
`324.apsi_m`	16.67	9.09	19.70
`326.gafort_m`	0.80	0.21	0.10
`328.fma3d_m`	31.01	19.05	16.08
`330.art_m`	2.56	8.82	6.13
`332.ammp_m`	51.72	44.73	41.62

Table 2: Coalescing memory accesses: gain (in percentage) in number of events, in tracing time, and in profiling time.

To implement our second optimization, Parwiz will select loops whose description is precise enough to not require a complete instrumentation of its code. This description will be used to avoid instrumenting the loop, transmitting just enough parameters for the profiler to be able to infer the overall impact of the loop on memory.

A loop nest N (containing any number of sub-loops, down to any depth) is said to be *parametric affine* if it meets the following conditions:

- all address expressions inside N involve only counters of N's loops and registers that are invariant in N, i.e., registers defined *before* N;
- all branch conditions are known, and their expressions respect the same restrictions;
- all expressions are linear in the loop counters, but not necessarily in parameters;
- all loops have a known iteration condition, i.e., an iteration has a uniform effect on the loop counter.

Such loops are also known as *static control loops*, because their behavior does not depend on any data manipulated during the iterations. The memory accesses of a parametric loop can be reproduced given only the values of the parameters it uses, irrespective of the contents of the memory areas it uses. Intuitively, loops iterating over multi-dimensional arrays are parametric, whereas loops over linked structures are not.

Let us now explain how parametric affine loop nests will be used. First, since the loop is parametrized, the original program will be instrumented to provide the values of the parameters required by the loop, instead of the memory accesses of the loop. This requires two new event types:

PARAM id value this is emitted when a parameter is defined (parameter ids are simple sequential numbers);

EXEC id this is emitted by instrumenting the control-flow edge(s) leading to the loop header, once and for all before the execution of the loop starts.

```
void 0x406b10_1(reg_t r15_58, reg_t r9_81, reg_t r11_93, reg_t rbp_2,
                reg_t r14_7, reg_t r13_8, reg_t rsi_214, reg_t r10_94)
{
    for ( reg_t i_0x406b10=0 ; (-0x1 + r9_81 + -i_0x406b10 >= 0) ; i_0x406b10++ ) {
        if ( (rbp_2 > 0) ) {
            for ( reg_t i_0x406b28=0 ; (-0x1 + rbp_2 + -i_0x406b28 >= 0) ; i_0x406b28++ ) {
                ACCESS('R', 8, r15_58 + 8*r11_93 + 8*i_0x406b28 + 8*r13_8*i_0x406b10);
                ACCESS('W', 8, r14_7 + 8*r10_94 + 8*i_0x406b28 + 8*rsi_214*i_0x406b10);
    }}}
}
```

Figure 8: A symbolic model for a loop extracted from binary code, translated to C code.

Instrumenting loops this way lowers the weight of instrumentation. More importantly, it replaces instrumentation of repeatedly executed instructions (the memory accesses inside the loop) by instrumentation of register definitions that are executed only once, before the loop starts. So we expect an important gain in instrumentation and communication costs. But this is only half the story.

The profiling component, on the other side of the pipe, receives parameter values, and an indication that the loop is executed. It is therefore responsible for transforming this information into the set of memory accesses. The first solution is for the profiler to simply run the loop (or a recompiled, minimal form of it—static analysis has provided enough information to create a "trace generator", see Figure 8), but this leads to no gain at all in profiling costs, not counting the overhead of running the loop. However, it saves a lot in instrumentation and communication time.

A better solution would to turn the loop into a set of memory ranges. It seems that this problem is far from easy for parametric loops, and the ones that Parwiz finds are heavily parametrized, with little or no available hypothesis on the values of the parameters: the simple loop in Figure 8 requires eight parameter values. We have not yet solved this problem. Many clever techniques (and tricks) can be used for restricted families of loops, but a generally applicable solution requires a lot more theoretical work.

4.4. Evaluation

To evaluate Parwiz' optimization techniques, we have run a sample of SPEC2006 benchmarks (compiled with `gcc` at `-O2` optimization level) on their train input data set, and measured how many memory accesses were "saved". Since coalescing memory accesses and loop modeling are completely unrelated, we evaluate them separately, and then combined. Our evaluation is entirely based on the number of memory accesses that do not need to be transmitted from the instrumented program to the dynamic dependence analyzer.

Reducing the number of memory accesses that have to be processed has a triple effect. First, it reduces the static amount of instrumentation that has to be inserted into the original program, thereby avoiding deleterious effects on instruction caches, instruction scheduling, branch predictors, and so on. Second, it reduces the quantity of instrumentation code that needs to be executed. The magnitude of this reduction depends on the frequency with which the various parts of the code are exercised. Third, it reduces the actual running time of the instrumented program, but in a way that depends on the instrumentation infrastructure: although the amount of dynamic instrumentation and the actual run time are highly correlated, they are not absolutely equivalent.

Table 3 displays the effect of both optimization techniques used alone, and also of their combined use, on the three quantities of interest. The reference values are those provided by an execution of the instrumented program without any optimization. Each number quantifies the gain along a given quantity observed when one or both optimizations have been applied. For instance, the first three columns show the effect of memory access coalescing alone on static instrumentation, dynamic instrumentation, and run time respectively. Looking at the line in this table (for `401.bzip`) in this graph, one can observe that memory coalescing removes around 20% of the static instrumentation, but that removal led to a gain of only around 10% in dynamic instrumentation (more instrumentation has been removed in less often executed code). This may seem disappointing, by eventually more than 25% of the run time was saved. This shows that the three factor have complex interactions. Therefore, we present all three quantities in each optimization setting.

There are several lessons to learn from these graphs. First, coalescing consecutive memory accesses almost always leads to a gain. In some cases, e.g. `416.gamess`, more accesses are coalesced in less frequently executed portions of code, and vice-versa. In other cases, e.g. `435.gromacs`, coalesced memory accesses are obviously placed in hot spots, and the gain in dynamic instrumentation and run time is far superior to what the static gain would have led to expect.

Second, finding parametric loop nests can have a dramatic effect on run time. Programs like `403.gcc`, which have almost no loops, see no change in their behavior, whereas programs like `433.milc` see their run time divided by 4 thanks to one or more loops occupying less than 4% of the program space.

Finally, the last group of columns shows that both optimization techniques are complementary in terms of dynamic instrumentation. Since coalescing seems to apply across the whole program code, whereas loops nests focus on small pieces of the code, their effects usually accumulate. For instance, `454.calculix` gains 20% dynamic instrumentation thanks to memory coalescing alone, and 27% thanks to parametric loop nests alone: when both are combined, the gain is over 46%. However, this doesn't always translate into an equivalent gain in run-time. We have no plausible explanation for this phenomenon, which is certainly highly dependent on the instrumentation infrastructure.

5. Related Work

One of the first paper on empirical dependence analysis is that of Larus [13], which defines the basic process and data structures (including simple execution points). A similar mechanism has been used to provide feedback to the compiler, even when dependence analysis is not exact [2]. Nowadays, a significant number of works on dynamic dependence profiling is more or less related to thread-level speculative parallelization (TLS), where the goal is to find portions of programs that can be executed in parallel with a small probability of

Program	Coalescing			Loop nests			Combined		
	Stat.	Dyn.	Time	Stat.	Dyn.	Time	Stat.	Dyn.	Time
401.bzip2	79.20	89.73	73.30	99.53	99.95	90.84	78.96	91.05	80.46
403.gcc	84.59	70.02	82.27	99.90	98.54	98.58	84.51	68.59	76.77
410.bwaves	51.37	71.53	76.15	96.86	19.53	42.15	51.52	9.50	36.54
416.gamess	68.79	89.38	93.16	99.06	95.88	82.59	68.53	85.40	74.06
429.mcf	76.68	87.10	77.25	98.80	99.54	99.47	76.46	86.65	95.10
433.milc	73.00	57.17	49.32	96.37	23.29	25.96	71.16	19.65	22.15
434.zeusmp	61.82	86.71	80.15	94.14	40.34	51.19	57.66	33.89	47.61
435.gromacs	61.96	20.90	24.43	98.99	98.27	93.41	61.39	19.56	23.84
436.cactusADM	68.89	21.28	24.79	93.95	0.12	4.14	67.79	0.09	4.24
437.leslie3d	54.19	79.62	70.76	70.04	4.45	14.51	42.72	4.40	15.31
444.namd	72.46	58.73	52.33	99.68	99.82	76.76	72.41	58.63	49.17
445.gobmk	74.60	77.84	81.77	99.87	98.55	95.71	74.52	76.48	79.08
447.dealII	70.17	80.88	76.53	99.12	87.53	85.15	69.86	70.94	72.03
450.soplex	80.90	92.21	95.04	99.21	94.43	79.18	81.13	86.65	85.83
454.calculix	68.58	80.15	83.24	98.71	72.99	64.48	68.00	53.81	57.01
456.hmmer	78.26	97.00	91.62	99.14	94.96	96.90	78.45	91.96	89.74
458.sjeng	85.83	85.44	80.16	99.33	99.23	92.84	85.48	84.67	84.57
462.libquantum	63.23	99.17	99.42	99.67	65.54	71.65	63.05	64.71	71.06
464.h264ref	81.48	64.57	62.91	98.09	91.98	89.63	79.99	56.78	61.75
465.tonto	59.47	50.15	50.17	99.87	43.24	40.53	64.41	45.77	50.83
470.lbm	49.39	59.99	60.60	80.81	98.22	73.31	46.08	59.69	56.15
473.astar	68.80	73.88	63.82	99.71	99.99	77.48	68.71	73.88	82.37
482.sphinx3	79.51	92.86	99.99	99.97	36.58	43.46	79.62	29.51	37.66
483.xalancbmk	73.86	73.38	75.61	99.94	99.93	90.95	73.82	73.31	73.99
average	70.29	73.32	71.87	96.70	73.45	70.04	69.43	56.07	59.47

Table 3: Evaluation of the optimization techniques according to the amount of static instrumentation (Stat.), the number of executions of instrumentation points (Dyn.), and raw run-time (Time). All numbers are percentages relative to the unoptimized program.

violating dependencies (in which case, all or a part of execution must be rolled back) [25], or when it may prove profitable [5]. The POSH system [16] is a representative example of a mixed system, combining static analysis to select tasks and dynamic data to refine this selection. Speculation usually either uses specific hardware structures like caches [22], or requires support for transactions [1].

The goal of our work is more related to systems that help the developer in using parallel programing structures, provided the program is actually parallel but the compiler cannot infer that fact from the source code. Examples of this use case include interactive compiler tools, like the SUIF Explorer system which let the user explore dependencies that prevent parallelization [15]. Using dynamic information, some authors suggest visualizing "ready plots", which highlight parallel sections in raw memory access traces [19]. Other works focus on providing loop information back to the developer for final approval [24]. Experiments have also been conducted on managed runtime environments [7].

The major goal of the framework presented in Section 2 is to handle fine-grain and coarse-grain program structures in an homogeneous way. Coarse-grain parallelism is a common target for dynamic systems, sometimes targeting specific parallel construction like pipelining [23]. Going beyond parallel constructions, some authors target program partitioning, including the distribution of program parts onto accelerators [20]. The ALCHEMIST system [27] includes variations on several features of the framework we describe in this paper: it uses execution points and a simpler version of our execution tree. As far as we know, none of these systems include a generic dependence graph builder, and/or target loop nest transformations.

Recent work on dynamic dependence analysis includes the SD3 system [12], which mainly targets loop parallelism (what we call *Boolean dependencies* in Section 3.1). Kim's thesis [10] also covers privatization, pipelining, and other enabling transformations. An interesting aspect of SD3 is its focus on optimizing the profiling phase, performing linear interpolation on memory accesses to reduce memory overhead, and parallelizing the profiler itself to speed-up the whole process. Our static analysis technique could certainly help SD3 by avoiding completely the run time interpolation phase, since many (but probably not all) linear access functions are discovered off-line. Another major difference between SD3 and our system is the handling of the memory "shadows", i.e., our memory table. SD3 uses an address table per loop execution, and a secondary table to detect inter-iteration dependencies. Parwiz uses a single table, a strategy that seems to reduce memory consumption significantly, since we have been able to run the unoptimized version of our system on SPEC benchmarks. Also, the use of complete execution points in Parwiz makes it easier to collect dependencies across loop-levels (Section 3.2) and process them in order of occurrence. SD3's approach to parallelize dependence analysis looks promising, and we plan to include some variation of it in later versions of our system.

A recent development based on SD3 called "Multi-slicing" [26] uses compiler-provided information on memory aliasing to cluster memory accesses into slices that can be analyzed independently. This partitioning is then used to parallelize the analysis, by running several distinctly instrumented versions of the program, one for each slice. The complete chain of tools relies on an existing compiler (GCC) for part of the static aliasing analysis, with additional steps on the

complete, whole-program call graph. This approach is essentially orthogonal to many current approaches (including the one presented in this paper). Its efficiency relies heavily on the accuracy of the aliasing information used.

The Kremlin system [6] is another notable recent work, with objectives very similar to ours. It uses the notion of a "parallelization plan", covering OMP-like loop parallelization and Cilk++-like task parallelism, even though the latter is not evaluated. The major focus of the system is on evaluating the expected speedup of the parallel program, a goal it shares with [24]. Kremlin uses the notion of self-parallelism, which in turn is based on evaluating an amount of work (apparently instruction counts) and critical path analysis. We have not covered this quantitative and predictive aspect of dynamic dependence analysis at all. Kremlin also uses a shadow memory, which seems to keep several versions ("availability times") of memory accesses, something Parwiz solves by keeping an execution tree.

Both Multi-slicing and Kremlin use an existing compiler architecture to analyze and instrument their target program. Compiler-based approaches are essentially unable to cover library code when no source is available. Parwiz uses static analysis of binary code instead, which is more flexible but probably less accurate in some cases.

Reducing the overhead of memory tracing has been an explicit or implicit concern in almost all trace-based studies. Most often the impact of tracing is reduced by using an acceptable approximation of the full trace: [18] is a recent representative example of this approach. Few studies have used static analysis of the executable code to reduce overhead while preserving perfect accuracy. Larus' qpt [14] is, as far as we know, the first system to trace abstract events that are then submitted to a "trace generator" (as our PARAM and EXEC events). Our own contribution to this approach can be found in [9].

6. Conclusion

This paper has presented a framework for dynamic dependence analysis. The system, called Parwiz, instruments the binary program under study to obtain a stream of events describing the execution. A profiler consumes this stream and keeps a model of the execution, which it uses to detect all data dependencies at runtime. We have shown three applications of the framework, covering three different kinds of parallelization tasks: searching for parallel loops, transforming a loop nest for vectorization, and studying the behavior of a given loop.

Finally, we have suggested some ways to optimize the whole process, by limiting the number of events generated by the program, and processed by the profiler. Memory access coalescing searches for accesses to regions of data that are larger than the word size. Loop replacement searches for complete loops whose effect on memory can be simulated. Both techniques have shown promising results. Loop replacement needs more work, especially on the theoretical aspect of the process, to be a convincing optimization in this context.

Probably the main aspect of Parwiz is its constant combination of static and dynamic techniques. Even though it acts directly on x86-64 binary programs, it shares many techniques with traditional compilers. We think this combination is an important strategy to lower the cost of empirical analysis, for dynamic data-dependence analysis but probably also for other types of studies.

Future research will tell where dynamic dependence analysis is most useful: as a profiling tool for developers, as a mechanism to collect feedback to compilers, or even as a component for dynamic optimization systems.

References

[1] M. Chen and K. Olukotun, "Test: a tracer for extracting speculative threads," in *CGO'03.*, march 2003.

[2] T. Chen, J. Lin, X. Dai, W.-C. Hsu, and P.-C. Yew, "Data dependence profiling for speculative optimizations," in *Compiler Construction*, E. Duesterwald, Ed. Springer Berlin / Heidelberg, 2004.

[3] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM TOPLAS*, vol. 13, no. 4, 1991.

[4] A. Darte and F. Vivien, "On the optimality of allen and kennedy's algorithm for parallelism extraction in nested loops," in *Euro-Par'96 Parallel Processing*, L. Bougé, P. Fraigniaud, A. Mignotte, and Y. Robert, Eds., 1996.

[5] Z.-H. Du, C.-C. Lim, X.-F. Li, C. Yang, Q. Zhao, and T.-F. Ngai, "A cost-driven compilation framework for speculative parallelization of sequential programs," in *PLDI'04*, 2004.

[6] S. Garcia, D. Jeon, C. M. Louie, and M. B. Taylor, "Kremlin: rethinking and rebooting gprof for the multicore age," in *PLDI'11*, 2011.

[7] C. Hammacher, K. Streit, S. Hack, and A. Zeller, "Profiling java programs for parallelism," in *IWMSE'09*, 2009.

[8] K. Kennedy and J. R. Allen, *Optimizing compilers for modern architectures: a dependence-based approach*. Morgan Kaufmann, 2002.

[9] A. Ketterlin and P. Clauss, "Efficient memory tracing by program skeletonization," in *ISPASS'11*, 2011, pp. 97–106.

[10] M. Kim, "Dynamic program analysis algorithms to assist parallelization," Ph.D. dissertation, Georgia Institute of Technology, 2012, available at http://www.cc.gatech.edu/~minjang/.

[11] M. Kim, H. Kim, and C.-K. Luk, "Prospector: A dynamic data-dependence profiler to help parallel programming," in *2nd USENIX Workshop on Hot Topics in Parallelism (HotPar'10)*, 2010.

[12] ——, "Sd3: A scalable approach to dynamic data-dependence profiling," in *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, 2010.

[13] J. R. Larus, "Loop-level parallelism in numeric and symbolic programs," *IEEE Trans. Parallel Distrib. Syst.*, vol. 4, July 1993.

[14] J. Larus, "Efficient program tracing," *Computer*, vol. 26, pp. 52–61, May 1993.

[15] S.-W. Liao, A. Diwan, R. P. Bosch, Jr., A. Ghuloum, and M. S. Lam, "Suif explorer: an interactive and interprocedural parallelizer," in *PPoPP'99*, 1999.

[16] W. Liu, J. Tuck, L. Ceze, W. Ahn, K. Strauss, J. Renau, and J. Torrellas, "Posh: a tls compiler that exploits program structure," in *PPoPP'06*, 2006.

[17] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," in *PLDI'05*, 2005.

[18] T. Moseley, A. Shye, V. J. Reddi, D. Grunwald, and R. Peri, "Shadow profiling: Hiding instrumentation costs with parallelism," in *CGO'07*, 2007.

[19] G. D. Price, J. Giacomoni, and M. Vachharajani, "Visualizing potential parallelism in sequential programs," in *PACT'08*, 2008.

[20] S. Rul, H. Vandierendonck, and K. De Bosschere, "Towards automatic program partitioning," in *CF'09*, 2009.

[21] V. C. Sreedhar, G. R. Gao, and Y.-F. Lee, "Identifying loops using dj graphs," *ACM Trans. Program. Lang. Syst.*, vol. 18, pp. 649–658, 1996.

[22] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry, "A scalable approach to thread-level speculation," *SIGARCH Comput. Archit. News*, vol. 28, May 2000.

[23] W. Thies, V. Chandrasekhar, and S. Amarasinghe, "A practical approach to exploiting coarse-grained pipeline parallelism in C programs," in *MICRO 40*, 2007.

[24] G. Tournavitis, Z. Wang, B. Franke, and M. F. O'Boyle, "Towards a holistic approach to auto-parallelization: integrating profile-driven parallelism detection and machine-learning based mapping," in *PLDI'09*, 2009.

[25] C. von Praun, R. Bordawekar, and C. Cascaval, "Modeling optimistic concurrency using quantitative dependence analysis," in *PPoPP'08*, 2008.

[26] H. Yu and Z. Li, "Multi-slicing: a compiler-supported parallel approach to data dependence profiling," in *ISSTA'12*, 2012.

[27] X. Zhang, A. Navabi, and S. Jagannathan, "Alchemist: A transparent dependence distance profiling infrastructure," in *CGO'09*, 2009.

2012 IEEE/ACM 45th Annual International Symposium on Microarchitecture

Neural Acceleration for General-Purpose Approximate Programs

Hadi Esmaeilzadeh Adrian Sampson Luis Ceze Doug Burger*
University of Washington *Microsoft Research
{hadianeh,asampson,luisceze}@cs.washington.edu dburger@microsoft.com

Abstract

This paper describes a learning-based approach to the acceleration of approximate programs. We describe the Parrot transformation*, a program transformation that selects and trains a neural network to mimic a region of imperative code. After the learning phase, the compiler replaces the original code with an invocation of a low-power accelerator called a* neural processing unit *(NPU). The NPU is tightly coupled to the processor pipeline to accelerate small code regions. Since neural networks produce inherently approximate results, we define a programming model that allows programmers to identify approximable code regions—code that can produce imprecise but acceptable results. Offloading approximable code regions to NPUs is faster and more energy efficient than executing the original code. For a set of diverse applications, NPU acceleration provides whole-application speedup of 2.3× and energy savings of 3.0× on average with quality loss of at most 9.6%.*

1. Introduction

Energy efficiency is a primary concern in computer systems. The cessation of Dennard scaling has limited recent improvements in transistor speed and energy efficiency, resulting in slowed general-purpose processor improvements. Consequently, architectural innovation has become crucial to achieve performance and efficiency gains [10].

However, there is a well-known tension between efficiency and programmability. Recent work has quantified three orders of magnitude of difference in efficiency between general-purpose processors and ASICs [21, 36]. Since designing ASICs for the massive base of quickly changing, general-purpose applications is currently infeasible, practitioners are increasingly turning to programmable accelerators such as GPUs and FPGAs. Programmable accelerators provide an intermediate point between the efficiency of ASICs and the generality of conventional processors, gaining significant efficiency for restricted domains of applications.

Programmable accelerators exploit some characteristic of an application domain to achieve efficiency gains at the cost of generality. For instance, FPGAs exploit copious, fine-grained, and irregular parallelism but perform poorly when complex and frequent accesses to memory are required. GPUs exploit many threads and SIMD-style parallelism but lose efficiency when threads diverge. Emerging accelerators, such as BERET [19], Conservation Cores and QsCores [47, 48], or DySER [18], map regions of general-purpose code to specialized hardware units by leveraging either small, frequently-reused code idioms (BERET and DySER) or larger code regions amenable to hardware synthesis (Conservation Cores). Whether an application can use an accelerator effectively depends on the degree to which it exhibits the accelerator's required characteristics.

Tolerance to approximation is one such program characteristic that is growing increasingly important. Many modern applications—such as image rendering, signal processing, augmented reality, data mining, robotics, and speech recognition—can tolerate inexact computation in substantial portions of their execution [7, 14, 28, 41]. This tolerance can be leveraged for substantial performance and energy gains.

This paper introduces a new class of programmable accelerators that exploit approximation for better performance and energy efficiency. The key idea is to *learn* how an original region of approximable code behaves and replace the original code with an efficient computation of the learned model. This approach contrasts with previous work on approximate computation that extends conventional microarchitectures to support selective approximate execution, incurring instruction bookkeeping overheads [1, 8, 11, 29], or requires vastly different programming paradigms [4, 24, 26, 32]. Like emerging flexible accelerators [18, 19, 47, 48], our technique automatically offloads code segments from programs written in mainstream languages; but unlike prior work, it leverages changes in the semantics of the offloaded code.

We have identified three challenges that must be solved to realize effective trainable accelerators:

1. A **learning algorithm** is required that can accurately and efficiently mimic imperative code. We find that neural networks can approximate various regions of imperative code and propose the Parrot transformation, which exploits this finding (Section 2).
2. A **language and compilation framework** should be developed to transform regions of imperative code to neural network evaluations. To this end, we define a programming model and implement a compilation workflow to realize the Parrot transformation (Sections 3 and 4). The Parrot transformation starts from regions of approximable imperative code identified by the programmer, collects training data, explores the topology space of neural networks, trains them to mimic the regions, and finally replaces the original regions of code with trained neural networks.
3. An **architectural interface** is necessary to call a neural processing unit (NPU) in place of the original code regions. The NPU we designed is tightly integrated with a speculative out-of-order core. The low-overhead interface enables acceleration even when fine-grained regions of code are transformed. The core communicates both the neural configurations and run-time invocations to the NPU through extensions to the ISA (Sections 5 and 6).

Rather than contributing a new design for neural network implementation, this paper presents a new technique for harnessing hardware neural networks in general-purpose computations. We show that using neural networks to replace regions of imperative code is feasible and profitable by experimenting with a variety of applications, including FFT, gaming, clustering, and vision algorithms (Section 7). These applications do not belong to the class of modeling and prediction that typically use neural networks. For each application, we identify a single approximable function that dominates the program's execution time. NPU acceleration provides 2.3× average whole-application speedup and 3.0× average energy savings for these benchmarks with average accuracy greater than 90% in all cases. Continuing work on NPU acceleration will provide a new class of accelerators—with implementation potential in both analog and digital domains—for emerging approximate applications.

1072-4451/12 $26.00 © 2012 IEEE
DOI 10.1109/MICRO.2012.48

IEEE computer society

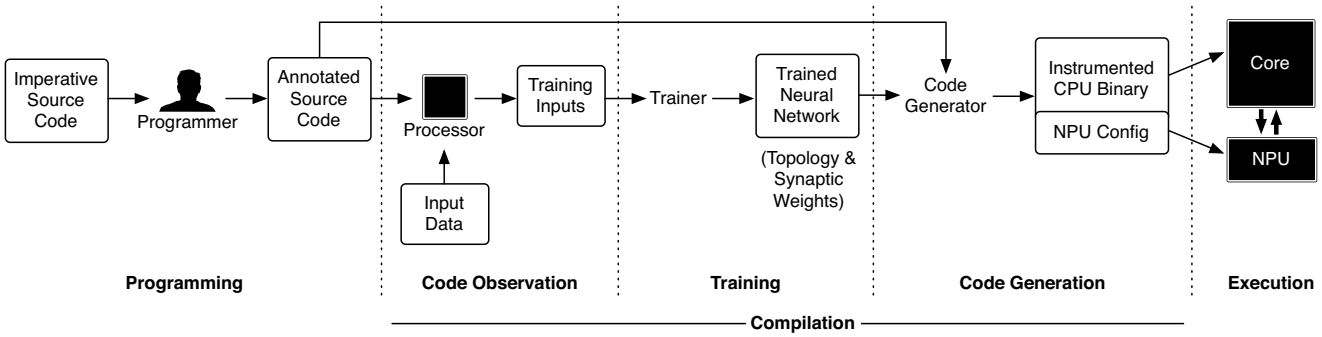


Figure 1: The Parrot transformation at a glance: from annotated code to accelerated execution on an NPU-augmented core.

2. Overview

The *Parrot transformation* is an algorithmic transformation that converts regions of imperative code to neural networks. Because neural networks expose considerable parallelism and can be efficiently accelerated using dedicated hardware, the Parrot transformation can yield significant performance and energy improvements. The transformation uses a training-based approach to produce a neural network that approximates the behavior of candidate code. A transformed program runs primarily on the main core and invokes an auxiliary hardware structure, the neural processing unit (NPU), to perform neural evaluation instead of executing the replaced code. Figure 1 shows an overview of our proposed approach, which has three key phases: programming, in which the programmer marks code regions to be transformed; compilation, in which the compiler selects and trains a suitable neural network and replaces the original code with a neural network invocation; and execution.

Programming. During development, the programmer explicitly annotates functions that are amenable to approximate execution and therefore candidates for the Parrot transformation. Because tolerance of approximation is a semantic property, it is the programmer's responsibility to select code whose approximate execution would not compromise the overall reliability of the application. This is common practice in the approximate computing literature [8, 11, 41]. We discuss our programming model in detail in Section 3.

Compilation. Once the source code is annotated, as shown in Figure 1, the compiler applies the Parrot transformation in three steps: (1) code observation; (2) neural network selection and training; and (3) binary generation. Section 4 details these steps.

In the code observation step, the compiler observes the behavior of the candidate code region by logging its inputs and outputs. This step is similar to profiling. The compiler instruments the program with probes on the inputs and outputs of the candidate functions. Then, the instrumented program is run using representative input sets such as those from a test suite. The probes log the inputs and outputs of the candidate functions. The logged input–output pairs constitute the training and validation data for the next step.

The compiler uses the collected input–output data to configure and train a neural network that mimics the candidate region. The compiler must discover the topology of the neural network as well as its synaptic weights. It uses the backpropagation algorithm [40] coupled with a topology search (see Section 4.2) to configure and train the neural network.

The final step of the Parrot transformation is code generation. The compiler first generates a configuration for the NPU that implements the trained neural network. Then, the compiler replaces each call to the original function with a series of special instructions that invoke the NPU, sending the inputs and receiving the computed outputs. The NPU configuration and invocation is performed through ISA extensions that are added to the core.

Execution. During deployment, the transformed program begins execution on the main core and configures the NPU. Throughout execution, the NPU is invoked to perform a neural network evaluation in lieu of executing the original code region. The NPU is integrated as a tightly-coupled accelerator in the processor pipeline. Invoking the NPU is faster and more energy-efficient than executing the original code region, so the program as a whole is accelerated.

Many NPU implementations are feasible, from all-software to specialized analog circuits. Because the Parrot transformation's effectiveness rests on the efficiency of neural network evaluation, it is essential that invoking the NPU be fast and low-power. Therefore, we describe a high-performance hardware NPU design based on a digital neural network ASIC (Section 6) and architecture support to facilitate low-latency NPU invocations (Section 5).

A key insight in this paper is that it is possible to automatically discover and train neural networks that effectively approximate imperative code from diverse application domains. These diverse applications do not belong to the class of modeling and prediction applications that typically use neural networks. The Parrot transformation enables a novel use of hardware neural networks to accelerate many approximate applications.

3. Programming Model

The Parrot transformation starts with the programmer identifying candidate code regions. These candidate regions need to comply with certain criteria to be suitable for the transformation. This section discusses these criteria as well as the concrete language interface exposed to the programmer. After the candidate regions are identified, the Parrot transformation is fully automated.

3.1. Code Region Criteria

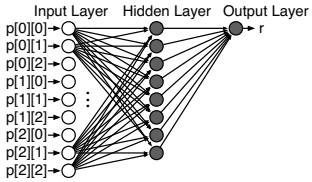
Candidate code for the Parrot transformation must satisfy three criteria: it must be frequently executed (i.e., a "hot" function); it must tolerate imprecision in its computation; and it must have well-defined inputs and outputs.

Hot code. Like any acceleration technique, the Parrot transformation should replace hot code. The Parrot transformation can be applied to a wide range of code from small functions to entire algorithms. The code region can contain function calls, loops, and complex control flow whose cost can be elided by the Parrot transformation. When applied to smaller regions of code, the overhead of

```
1  float sobel [[PARROT]] (float[3][3] p){
     float x, y, r;
3    x = (p[0][0] + 2 * p[0][1] + p[0][2]);
     x = (p[2][0] + 2 * p[2][1] + p[2][2]);
5    y = (p[0][2] + 2 * p[1][2] + p[2][2]);
     y = (p[0][0] + 2 * p[1][1] + p[2][0]);
7    r = sqrt(x * x + y * y);
     if (r >= 0.7071) r = 0.7070;
9    return r;
   }
```

```
   void edgeDetection(Image& srcImg, Image& dstImg){
2    float[3][3] p; float pixel;
     for(int y = 0; y < srcImg.height; ++y)
4      for(int x = 0; x < srcImg.width; ++x)
         srcImg.toGrayeScale(x,y);
6    for(int y = 0; y < srcImg.height; ++y)
       for(int x = 0; x < srcImg.width; ++x){
8        p = srcImg.build3x3Window(x, y);
         pixel = sobel(p);
10       dstImg.setPixel(x, y, pixel);
       }
12 }
```

(a) Original implementation of the Sobel filter



(b) The sobel function transformed to a $9 \rightarrow 8 \rightarrow 1$ NN

```
   void edgeDetection(Image& srcImg, Image& dstImg){
2    float[3][3] p; float pixel;
     for(int y = 0; y < srcImg.height; ++y)
4      for(int x = 0; x < srcImg.width; ++x)
         srcImg.toGrayeScale(x,y);
6    for(int y = 0; y < srcImg.height; ++y)
       for(int x = 0; x < srcImg.width; ++x){
8        p = srcImg.build3x3Window(x, y);
         NPU_SEND(p[0][0]); NPU_SEND(p[0][1]); NPU_SEND(p[0][2]);
10       NPU_SEND(p[1][0]); NPU_SEND(p[1][1]); NPU_SEND(p[1][2]);
         NPU_SEND(p[2][0]); NPU_SEND(p[2][1]); NPU_SEND(p[2][2]);
12       NPU_RECEIVE(pixel);
         dstImg.setPixel(x, y, pixel);
14     }
   }
```

(c) parrot transformed code; an NPU invocation replaces the function call

Figure 2: Three stages in the transformation of an edge detection algorithm using the Sobel filter.

NPU invocation needs to be low to make the transformation profitable. A traditional performance profiler can reveal hot code.

For example, edge detection is a widely applicable image processing computation. Many implementations of edge detection use the Sobel filter, a 3×3 matrix convolution that approximates the image's intensity gradient. As the bottom box in Figure 2a shows, the local Sobel filter computation (the sobel function) is executed many times during edge detection, so the convolution is a hot function in the overall algorithm and a good candidate for the Parrot transformation.

Approximability. Code regions identified for the Parrot transformation will behave approximately during execution. Therefore, programs must incorporate application-level tolerance of imprecision. This requires the programmer to ensure that imprecise results from candidate regions will not cause catastrophic failures. As prior work on approximate programming [2, 8, 29, 41, 43] has shown, it is not difficult to deem regions approximable.

Beyond determining that a code region may safely produce imprecise results, the programmer need not reason about the mapping between the code and a neural network. While neural networks are more precise for some functions than they are for others, we find that they can accurately mimic many functions from real programs (see Section 7). Intuitively, however, they are less likely to effectively approximate chaotic functions, in which even large training sets can fail to capture enough of the function's behavior to generalize to new inputs. However, the efficacy of neural network approximation can be assessed empirically. The programmer should annotate all approximate code; the compiler can then assess the accuracy of a trained neural network in replacing each function and select only those functions for which neural networks are a good match.

In the Sobel filter example, parts of the code that process the pixels can be approximated. The code region that computes pixel addresses and builds the window for the sobel function (line 8 in the bottom box of Figure 2a) needs to be precise to avoid memory access violations. However, the sobel function, which estimates the intensity gradient of a pixel, is fundamentally approximate. Thus, approximate execution of this function will not result in catastrophic failure and, moreover, is unlikely to cause major degradation of the overall edge detection quality. These properties make the sobel function a suitable candidate region for approximate execution.

Well-defined inputs and outputs. The Parrot transformation replaces a region of code with a neural network that has a fixed number of inputs and outputs. Therefore, it imposes two restrictions on the code regions that can feasibly be replaced. First, the inputs to and outputs from the candidate region must be of a fixed size known at compile time. For example, the code may not dynamically write an unbounded amount of data to a variable-length array. Second, the code must be *pure:* it must not read any data other than its inputs nor affect any state other than its outputs (e.g., via a system call). These two criteria can be checked statically.

The sobel function in Figure 2a complies with these requirements. It takes nine statically identifiable floating-point numbers as input, produces a single output, and has no side effects.

3.2. Annotation

In this work, we apply the Parrot transformation to entire functions. To identify candidate functions, the programmer marks them with an annotation (e.g., using C++11 [[annotation]] syntax as shown in Figure 2a). The programmer is responsible for ensuring that the function has no side effects, reads only its arguments, and writes only its return value. Each argument type and the return type must have a fixed size. If any of these types is a pointer type, it must point to a fixed-size value; this referenced value is then considered the neural network input or output rather than the pointer itself. If the function needs to return multiple values, it can return a fixed-size array or a C struct. After the programmer annotates the candidate functions, the Parrot transformation is completely automatic and transparent: no further programmer intervention is necessary.

Other annotation approaches. Our current system depends on explicit programmer annotations at the granularity of functions. While we find that explicit function annotations are straightforward to apply (see Section 7), static analysis techniques could be used to further simplify the annotation process. For example, in an approximation-aware programming language such as EnerJ [41], the programmer

uses type qualifiers to specify which data is non-critical and may be approximated. In such a system, the Parrot transformation can be automatically applied to any block of code that only affects approximate data. That is, the candidate regions for the Parrot transformation would be implicitly defined.

Like prior work on approximate computing, we acknowledge that some programmer guidance is essential when identifying error-tolerant code [2, 8, 11, 29, 41]. Tolerance to approximation is an inherently application-specific property. Fortunately, language-level techniques like EnerJ demonstrate that the necessary code annotations can be intuitive and straightforward for programmers to apply.

4. Compilation Workflow

Once the program has been annotated, the compilation workflow implements the Parrot transformation in three steps: observation, training, and instrumented binary generation.

4.1. Code Observation

In the first phase, the compiler collects input–output pairs for the target code that reflect real program executions. This in-context observation allows the compiler to train the neural network on a realistic data set. The compiler produces an instrumented binary for the source program that includes probes on the input and output of the annotated function. Each time the candidate function executes, the probes record its inputs and outputs. The program is run repeatedly using test inputs. The output of this phase is a training data set: each input–output pair represents a sample for the training algorithm. The system also measures the minimum and maximum value for each input and output; the NPU normalizes values using these ranges during execution.

The observation phase resembles the profiling runs used in profile-guided compilation. Specifically, it requires representative test inputs for the application. The inputs may be part of an existing test suite or randomly generated. In many cases, a small number of application test inputs are sufficient to train a neural network because the candidate function is executed many times in a single application run. In our edge detection example, the `sobel` function runs for every pixel in the input image. So, as Section 7 details, training `sobel` on a single 512×512 test image provides 262144 training data points and results in acceptable accuracy when computing on unseen images.

Although we do not explore it in this paper, automatic input generation could help cover the space of possible inputs and thereby achieve a more accurate trained neural network. In particular, the compiler could synthesize new inputs by interpolating values between existing test cases.

4.2. Training

The compiler uses the training data to produce a neural network that replaces the original function. There are a variety of types of artificial neural networks in the literature, but we narrow the search space to multilayer perceptrons (MLPs) due to their broad applicability.

The compiler uses the backpropagation algorithm [40] to train the neural network. Backpropagation is a gradient descent algorithm that iteratively adjusts the weights of the neural network according to each input–output pair. The learning rate, a value between 0 and 1, is the step size of the gradient descent and identifies how much a single example affects the weights. Since backpropagation on MLPs is not convex and the compilation procedure is automatic, we choose a small learning rate of 0.01. Larger steps can cause oscillation in the training and prevent convergence. One complete pass over the training data is called an epoch. Since the learning rate is small, the epoch count should be large enough to ensure convergence. Empirically, we find that 5000 epochs achieve a good balance of generalization and accuracy. Larger epoch counts can cause overtraining and adversely affect the generalization ability of the network while smaller epoch counts may result in poor accuracy.

Neural network topology selection. In addition to running backpropagation, this phase selects a network topology that balances between accuracy and efficiency. An MLP consists of a fully-connected set of neurons organized into layers: the input layer, any number of "hidden" layers, and the output layer (see Figure 2b). A larger, more complex network offers better accuracy potential but is likely to be slower and less power-efficient than a small, simple neural network.

To choose the topology, we use a simple search algorithm guided by the mean squared error of the neural network when tested on an unseen subset of the observed data. The error evaluation uses a typical cross-validation approach: the compiler partitions the data collected during observation into a *training set*, 70% of the observed data, and a *test set*, the remaining 30%. The topology search algorithm trains many different neural network topologies using the training set and chooses the one with the highest accuracy on the test set and the lowest latency on the NPU (prioritizing accuracy).

The space of possible topologies is large, so we restrict the search to neural networks with at most two hidden layers. We also limit the number of neurons per hidden layer to powers of two up to 32. (The numbers of neurons in the input and output layers are predetermined based on the number of inputs and outputs in the candidate function.) These choices limit the search space to 30 possible topologies. The maximum number of hidden layers and maximum neurons per hidden layer are compilation options and can be specified by the user. Although the candidate topologies can be trained in parallel, enlarging the search space increases the compilation time.

The output from this phase consists of a neural network topology—specifying the number of layers and the number of neurons in each layer—along with the weight for each neuron and the normalization range for each input and output. Figure 2b shows the three-layer MLP that replaces the `sobel` function. Each neuron in the network performs a weighted sum on its inputs and then applies a sigmoid function to the result of weighted sum.

On-line training. Our present system performs observation and training prior to deployment; an alternative design could train the neural network concurrently with in-vivo operation. On-line training could improve accuracy but would result in runtime overheads. To address these overheads, an on-line training system could offload neural network training and configuration to a remote server. With off-site training, multiple deployed application instances could centralize their training to increase input space coverage.

4.3. Code Generation

After the training phase, the compiler generates an instrumented binary that runs on the core and invokes the NPU instead of calling the original function. The program configures the NPU when it is first loaded by sending the topology parameters and synaptic weights to the NPU via its configuration interface (Section 6.2). The compiler replaces the calls to the original function with special instructions that send the inputs to the NPU and collect the outputs from it. The

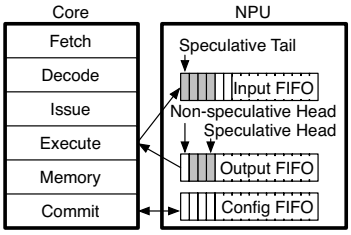


Figure 3: The NPU exposes three FIFO queues to the core. The speculative state of the FIFOs is shaded.

configuration and input–output communication occurs through ISA extensions discussed in Section 5.1.

5. Architecture Design for NPU Acceleration

Since candidate regions for the Parrot transformation can be fine-grained, NPU invocation must be low-overhead to be beneficial. Ideally, the NPU should integrate tightly with the processor pipeline. The processor ISA also needs to be extended to allow programs to configure and invoke the NPU during execution. Moreover, NPU invocation should not prevent speculative execution. This section discusses the ISA extensions and microarchitectural mechanism for tightly integrating the NPU with an out-of-order processor pipeline.

5.1. ISA Support for NPU Acceleration

The NPU is a variable-delay, tightly-coupled accelerator that communicates with the rest of the core via FIFO queues. As shown in Figure 3, the CPU–NPU interface consists of three queues: one for sending and retrieving the configuration, one for sending the inputs, and one for retrieving the neural network's outputs. The ISA is extended with four instructions to access the queues. These instructions assume that the processor is equipped with a single NPU; if the architecture supports multiple NPUs or multiple stored configurations per NPU, the instructions may be parameterized with an operand that identifies the target NPU.

- **enq.c %r**: enqueues the value of the register r into the config FIFO.
- **deq.c %r**: dequeues a configuration value from the config FIFO to the register r.
- **enq.d %r**: enqueues the value of the register r into the input FIFO.
- **deq.d %r**: dequeues the head of the output FIFO to the register r.

To set up the NPU, the program executes a series of enq.c instructions to send configuration parameters—number of inputs and outputs, network topology, and synaptic weights—to the NPU. The operating system uses deq.c instructions to save the NPU configuration during context switches. To invoke the NPU, the program executes enq.d repeatedly to send inputs to the configured neural network. As soon as all of the inputs of the neural network are enqueued, the NPU starts computation and puts the results in its output FIFO. The program executes deq.d repeatedly to retrieve the output values.

Instead of special instructions, an alternative design could use memory-mapped IO to communicate with the NPU. This design would require special fence instructions to prevent interference between two consecutive invocations and could impose a large overhead per NPU invocation.

5.2. Speculative NPU-Augmented Architecture

Scheduling and issue. To ensure correct communication with the NPU, the processor must issue NPU instructions in order. To accomplish this, the renaming logic implicitly considers every NPU instruction to read and write a designated "dummy" architectural register. The scheduler will therefore treat all NPU instructions as dependent. Furthermore, the scheduler only issues an enqueue instruction if the corresponding FIFO is not full. Similarly, a dequeue instruction is only issued if the corresponding FIFO is not empty.

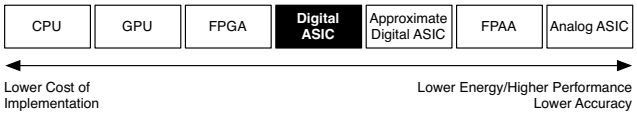


Figure 4: Design space of NPU implementations. This paper focuses on a precise digital ASIC design.

Speculative execution. The processor can execute enq.d and deq.d instructions speculatively. Therefore, the head pointer of the input FIFO can only be updated—and consequently the entries recycled—when: (1) the enqueue instruction commits; and (2) the NPU finishes processing that input. When an enq.d instruction reaches the commit stage, a signal is sent to the NPU to notify it that the input FIFO head pointer can be updated.

To ensure correct speculative execution, the output FIFO maintains two head pointers: a speculative head and a non-speculative head. When a dequeue instruction is issued, it reads a value from the output FIFO and the speculative head is updated to point to the next output. However, the non-speculative head is not updated to ensure that the read value is preserved in case the issue of the instruction was a result of misspeculation. The non-speculative head pointer is only updated when the instruction commits, freeing the slot in the output FIFO.

In case of a flush due to branch or dependence misspeculation, the processor sends the number of squashed enq.d and deq.d instructions to the NPU. The NPU adjusts its input FIFO tail pointer and output FIFO speculative head pointer accordingly. The NPU also resets its internal control state if it was processing any of the invalidated inputs and adjusts the output FIFO tail pointer to invalidate any outputs that are based on the invalidated inputs. The rollback operations are performed concurrently for the input and output FIFOs.

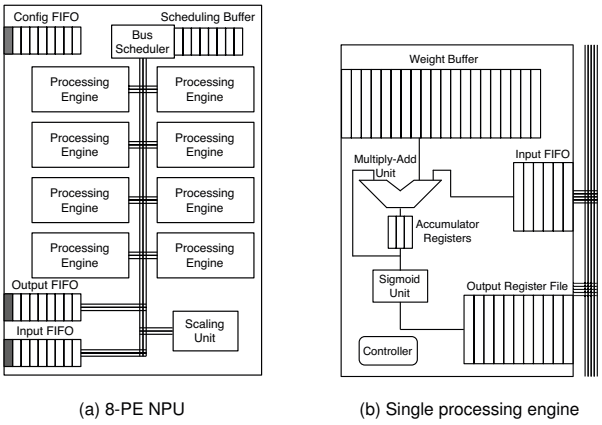
The enq.c and deq.c instructions, which are only used to read and write the NPU configuration, are not executed speculatively.

Interrupts. If an interrupt were to occur during an NPU invocation, the speculative state of the NPU would need to be flushed. The remaining non-speculative data in the input and output FIFOs would need to be saved and then restored when the process resumes. One way to avoid this complexity is to disable interrupts during NPU invocations; however, this approach requires that the invocation time is finite and ideally short as to not delay interrupts for too long.

Context switches. The NPU's configuration is architectural state, so the operating system must save and restore the configuration data on a context switch. The OS reads out the current NPU configuration using the deq.c instruction and stores it for later reconfiguration when the process is switched back in. To reduce context switch overheads, the OS can use the same lazy context switch techniques that are typically used with floating point units [33].

6. Neural Processing Unit

There are many implementation options for NPUs with varying trade-offs in performance, power, area, and complexity, as illustrated by Figure 4. At one extreme are software implementations running on a CPU or GPU [20, 34]. Since these implementations have higher computation and communication overheads, they are likely more suitable for very large candidate regions, when the invocation cost



(a) 8-PE NPU (b) Single processing engine

Figure 5: Reconfigurable 8-PE NPU.

can be better amortized. Next on the scale are FPGA-based implementations [50]. Digital ASIC designs are likely to be lower-latency and more power-efficient than FPGA-based implementations [9, 37]. Since neural networks are themselves approximable, their implementation can also be approximate. Therefore, we can improve efficiency further and use approximate digital circuits (e.g., sub-critical voltage supply). In the extreme, one can even use custom analog circuitry or FPAAs [3, 42, 44]. In fact, we believe that analog NPUs have significant potential and we plan to explore them in future work. We focus on an ASIC design operating at the same critical voltage as the main core. This implementation represents a reasonable trade-off between efficiency and complexity; it is able to accelerate a wide variety of applications without the complexity of integrating analog or sub-critical components with the processor.

6.1. Reconfigurable Digital NPU

The Parrot transformation produces different neural network topologies for different code regions. Thus, we propose a reconfigurable NPU design that accelerates the evaluation of a range of neural topologies. As shown in Figure 5a, the NPU contains eight identical processing engines (PEs) and one scaling unit. Although the design can scale to larger numbers of PEs, we find that the speedup gain beyond 8 PEs is small (see Section 7). The scaling unit scales the neural network's inputs and outputs if necessary using scaling factors defined in the NPU configuration process.

The PEs in the NPU are statically scheduled. The scheduling information is part of the configuration information for the NPU, which is based on the neural network topology derived during the training process. In the NPU's schedule, each neuron in the neural network is assigned to one of the eight PEs. The neural network's topology determines a static schedule for the timing of the PE computations, bus accesses, and queue accesses.

The NPU stores the bus scheduling information in its circular scheduling buffer (shown in Figure 5a). Each entry in this buffer schedules the bus to send a value from a PE or the input FIFO to a set of destination PEs or the output FIFO. Every scheduling buffer entry consists of a source and a destination. The source is either the input FIFO or the identifier of a PE along with an index into its output register file (shown in Figure 5b). The destination is either the output FIFO or a bit field indicating the destination PEs.

Figure 5b shows the internal structure of a single PE. Each PE performs the computation for all of its assigned neurons. Namely, because the NPU implements a sigmoid-activation multilayer perceptron, each neuron computes its output as $y = \text{sigmoid}(\sum_i (x_i \times w_i))$ where x_i is an input to the neuron and w_i is its corresponding weight. The weight buffer, a circular buffer, stores the weights. When a PE receives an input from the bus, it stores the value in its input FIFO. When the neuron weights for each PE are configured, they are placed into the weight buffer; the compiler-directed schedule ensures that the inputs arrive in the same order that their corresponding weights appear in the buffer. This way, the PE can perform multiply-and-add operations in the order the inputs enter the PE's input FIFO.

Each entry in the weight buffer is augmented by one bit indicating whether a neuron's multiply-add operation has finished. When it finishes, the PE applies the sigmoid function, which is implemented as a lookup table, and write the result to its output register file. The per-neuron information stored in the weight buffer also indicates which output register should be used.

6.2. NPU Configuration

During code generation (Section 4.3), the compiler produces an NPU configuration that implements the trained neural network for each candidate function. The static NPU scheduling algorithm first assigns an order to the inputs of the neural network. This order determines both the sequence of `enq.d` instructions that the CPU will send to the NPU during each invocation and the order of multiply-add operations among the NPU's PEs. Then, the scheduler takes the following steps for each layer of the neural network:

1. Assign each neuron to one of the processing engines.
2. Assign an order to the multiply-add operations considering the order assigned to the inputs of the layer.
3. Assign an order to the outputs of the layer.
4. Produce a bus schedule reflecting the order of operations.

The ordering assigned for the final layer of the neural network dictates the order in which the program will retrieve the NPU's output using `deq.d` instructions.

7. Evaluation

To evaluate the effectiveness of the Parrot transformation, we apply it to several benchmarks from diverse application domains. For each benchmark, we identify a region of code that is amenable to the Parrot transformation. We evaluate whole-application speedup and energy savings using cycle-accurate simulation and a power model. We also examine the resulting trade-off in computation accuracy. We perform a sensitivity analysis to examine the effect of NPU PE count and communication latency on the performance benefits.

7.1. Benchmarks and the Parrot Transformation

Table 1 lists the benchmarks used in this evaluation. These benchmarks are all written in C. The application domains—signal processing, robotics, gaming, compression, machine learning, and image processing—are selected for their usefulness to general applications and tolerance to imprecision. The domains are commensurate with evaluations of previous work on approximate computing [1, 11, 28, 29, 41, 43].

Table 1 also lists the input sets used for performance, energy, and accuracy assessment. These input sets are different from the ones used during the training phase of the Parrot transformation. For applications with random inputs we use a different random input set. For applications with image input, we use a different image.

Table 1: The benchmarks evaluated, characterization of each transformed function, 0 data, and the result of the Parrot transformation.

	Description	Type	Evaluation Input Set	# of Function Calls	# of Loops	# of ifs/ elses	# of x86-64 Instructions	Training Input Set	Neural Network Topology	NN MSE	Error Metric	Error
fft	Radix-2 Cooley-Tukey fast Fourier	Signal Processing	2048 Random Floating Point Numbers	2	0	0	34	32768 Random Floating Point Numbers	1 -> 4 -> 4 -> 2	0.00002	Average Relative Error	7.22%
inversek2j	Inverse kinematics for 2-joint arm	Robotics	10000 (x,y) Random Coordinates	4	0	0	100	10000 (x,y) Random Coordinates	2 -> 8 -> 2	0.00563	Average Relative Error	7.50%
jmeint	Triangle intersection detection	3D Gaming	10000 Random Pairs of 3D Triangle Coordinates	32	0	23	1,079	100000 Random Pairs of 3D Triangle Coordinates	18 -> 32 -> 8 -> 2	0.00530	Miss Rate	7.32%
jpeg	JPEG encoding	Compression	220x200-Pixel Color Image	3	4	0	1,257	Three 512x512-Pixel Color Images	64 -> 16 -> 64	0.00890	Image Diff	9.56%
kmeans	K-means clustering	Machine Learning	220x200-Pixel Color Image	1	0	0	26	50000 Pairs of Random (r, g, b) Values	6 -> 8 -> 4 -> 1	0.00169	Image Diff	6.18%
sobel	Sobel edge detector	Image Processing	220x200-Pixel Color Image	3	2	1	88	One 512x512-Pixel Color Image	9 -> 8 -> 1	0.00234	Image Diff	3.44%

Code annotation. The C source code for each benchmark was annotated as described in Section 3: we identified a single pure function with fixed-size inputs and outputs. No algorithmic changes were made to the benchmarks to accommodate the Parrot transformation. There are many choices for the selection of target code and, for some programs, multiple NPUs may even have been beneficial. For the purposes of this evaluation, however, we selected a single target region per benchmark that was easy to identify, frequently executed as to allow for efficiency gains, and amenable to learning by a neural network. Qualitatively, we found it straightforward to identify a reasonable candidate function in each benchmark.

Table 1 shows the number of function calls, conditionals, and loops in each transformed function. The table also shows the number of x86-64 instructions for the target function when compiled by GCC 4.4.6 at the -O3 optimization level. We do not include the statistics of the standard library functions in these numbers. In most of these benchmarks, the target code contains complex control flow including conditionals, loops, and method calls. In jmeint, the target code contains the bulk of the algorithm, including many nested method calls and numerous conditionals. In jpeg, the transformation subsumes the discrete cosine transform and quantization phases, which contain function calls and loops. In fft, inversek2j, and sobel, the target code consists mainly of arithmetic operations and simpler control flow. In kmeans, the target code is the 0 distance calculation, which is simple and fine-grained yet frequently executed. In each case, the target code is side-effect-free and the number of inputs/outputs are statically identifiable.

Training data. To train the NPU for each application, we have used either (1) typical program inputs (e.g., sample images) or (2) a limited number of random inputs. For the benchmarks that use random inputs, we determined the permissible range of parameters in the code and generated uniform random inputs in that range. For the image-based benchmarks, we used three standard images that are used to evaluate image processing algorithms (lena, mandrill, and peppers). For kmeans, we supplied random inputs to the code region to avoid overtraining on a particular test image. Table 1 shows the specific image or application input used in the training phase for each benchmark. We used different random inputs and different images for the final accuracy evaluation.

Neural networks. The "Neural Network Topology" column in Table 1 shows the topology of the trained neural network discovered by the training stage described in Section 4.2. The "NN MSE" column shows the mean squared error for each neural network on the test subset of the training data. For example, the topology for jmeint is $18 \rightarrow 32 \rightarrow 8 \rightarrow 2$, meaning that the neural network takes in 18 inputs, produces 2 outputs, and has two hidden layers with 32 and 8 neurons respectively. As the results show, the compilation workflow was able to find a neural network that accurately mimics each original function. However, different topologies are required to approximate different functions.

> Different applications require different neural network topologies, so the NPU structure must be reconfigurable.

Output quality. We use an application-specific error metric, shown in Table 1, to assess the quality of each benchmark's output. In all cases, we compare the output of the original untransformed application to the output of the transformed application. For fft and inversek2j, which generate numeric outputs, we measure the average relative error. jmeint calculates whether two three-dimensional triangles intersect; we report the misclassification rate. For jpeg, kmeans, and sobel, which produce image outputs, we use the average root-mean-square image difference. The column labeled "Error" in Table 1 shows the whole-application error of each benchmark according to its error metric. Unlike the "NN MSE" error values, this application-level error assessment accounts for accumulated errors due to repeated execution of the transformed function.

Application average error rates range from 3% to 10%. This quality-of-service loss is commensurate with other work on quality trade-offs. Among hardware approximation techniques, Truffle [11] shows similar error (3–10%) for some applications and much greater error (above 80%) for others in a moderate configuration. The evaluation of EnerJ [41] also has similar error rates; two thirds of the applications exhibit error greater than 10% in the most energy-efficient configuration. Green [2], a software technique, has error rates below 1% for some applications but greater than 20% for others. A case study by Misailovic et al. [30] explores manual optimizations of a video encoder, x264, that trade off 0.5–10% quality loss.

Table 2: Microarchitectural parameters for the core, caches, memory, NPU, and each PE in the NPU.

Core		Caches and Memory		NPU	
Architecture	x86-64	L1 Cache Size	32 KB instruction, 32 KB data	Number of PEs	8
Fetch/Issue Width	4/6	L1 Line Width	64 bytes	Bus Schedule FIFO	512×20-bit
INT ALUs/FPUs	3/2	L1 Associativity	8	Input FIFO	128×32-bit
Load/Store FUs	2/2	L1 Hit Latency	3 cycles	Output FIFO	128×32-bit
ROB Entries	96	ITLB/DTLB Entries	128/256	Config FIFO	8×32-bit
Issue Queue Entries	32	L2 Cache Size	2 MB	**NPU PE**	
INT/FP Physical Registers	256/256	L2 Line Width	64 bytes		
Branch Predictor	Tournament, 48 KB	L2 Associativity	8	Weight Cache	512×33-bit
BTB Sets/Ways	1024/4	L2 Hit Latency	12	Input FIFO	8×32-bit
RAS Entries	64	Memory Latency	50 ns (104 cycles)	Output Register File	8×32-bit
Load/Store Queue Entries	48/48			Sigmoid Unit LUT	2048×32-bit
Dependence Predictor	4096-entry Bloom Filter			Multiply-Add Unit	32-bit Single-Precision FP

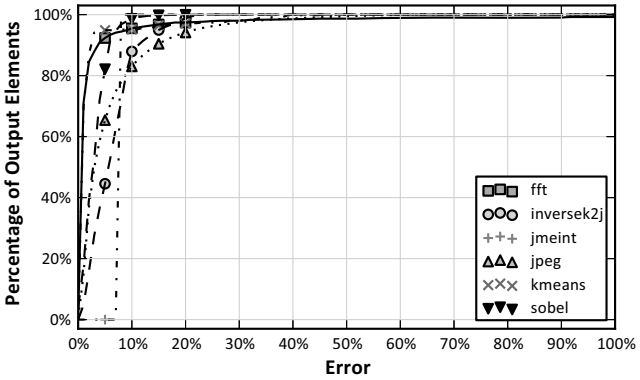


Figure 6: Cumulative distribution function (CDF) plot of the applications' output error. A point (x,y) indicates that y fraction of the output elements see error less than or equal to x.

The Parrot transformation degrades each application's average output quality by less than 10%, a rate commensurate with other approximate computing techniques.

To study the application level quality loss in more detail, Figure 6 depicts the CDF (cumulative distribution function) plot of final error for each element of application's output. The output of each benchmark consists of a collection of elements—an image consists of pixels; a vector consists of scalars; etc. The error CDF reveals the distribution of output errors among an application's output elements and shows that very few output elements see large quality loss.

The majority (80% to 100%) of each transformed application's output elements have error less than 10%.

7.2. Experimental Setup

Cycle-accurate simulation. We use the MARSSx86 cycle-accurate x86-64 simulator [35] to evaluate the performance effect of the Parrot transformation and NPU acceleration. Table 2 summarizes the microarchitectural parameters for the core, memory subsystem, and NPU. We configure the simulator to resemble Intel's Penryn microarchitecture, which is an aggressive out-of-order design. We augment MARSSx86 with a cycle-accurate NPU simulator and add support for NPU queue instructions through unused x86 opcodes. We use C assembly inlining to add the NPU invocation code. We compile the benchmarks using GCC version 4.4.6 with the -O3 flag to enable aggressive compiler optimizations. The baseline in all of the reported results is the execution of the entire benchmark on the core without the Parrot transformation.

Energy modeling. MARSSx86 generates an event log during the cycle-accurate simulation of the program. The resulting statistics are sent to a modified version of McPAT [27] to estimate the energy consumption of each execution. We model the energy consumption of an 8-PE NPU using the results from McPAT and CACTI 6.5 [31] for memory arrays, buses, and steering logic. We use the results from Galal et al. [17] to estimate the energy of multiply-and-add operations. We model the NPU and the core at the 45 nm technology node. The NPU operates at the same frequency and voltage as the main core. We use the 2080 MHz frequency and $V_{dd} = 0.9\ V$ settings because the energy results in Galal et al. [17] are for this frequency and voltage setting.

7.3. Experimental Results

Dynamic instruction subsumption. Figure 7 depicts dynamic instruction count of each transformed benchmark normalized to the instruction count for CPU-only execution. The figure divides each application into NPU communication instructions and application instructions. While the potential benefit of NPU acceleration is directly related to the amount of CPU work that can be elided, the queuing instructions and the cost of neural network evaluation limit the actual benefit. For example, inversek2j exhibits the greatest potential for benefit: even accounting for the communication instructions, the transformed program executes 94% fewer instructions on the core. Most of the benchmark's dynamic instructions are in the target region for the Parrot transformation and it only communicates four values with the NPU per invocation. This is because inversek2j is an ideal case: the entire algorithm has a fixed-size input ((x,y) coordinates of the robot arm), fixed-size output ((θ_1,θ_2) angles for the arm joints), and tolerance for imprecision. In contrast, kmeans is representative of applications where the Parrot transformation applies more locally: the target code is "hot" but only consists of a few arithmetic operations and the communication overhead is relatively high.

Performance and energy benefits. Figure 8a shows the application speedup when an 8-PE NPU is used to replace each benchmark's target function. The rest of the code runs on the core. The baseline is executing the entire, untransformed benchmark on the CPU. The plots also show the potential available speedup: the hypothetical speedup if the NPU takes zero cycles for computation. Among the benchmarks inversek2j sees the highest speedup (11.1×) since the Parrot transformation substitutes the bulk of the application with a relatively small NN ($2 \rightarrow 8 \rightarrow 2$). On the other hand, kmeans sees a 24% slowdown even though it shows a potential speedup of 20% in the limit. The transformed region of code in kmeans consists of 26 mostly arithmetic instructions that can efficiently run on the core

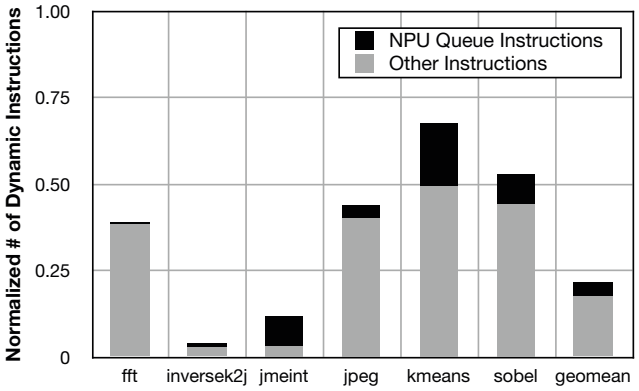
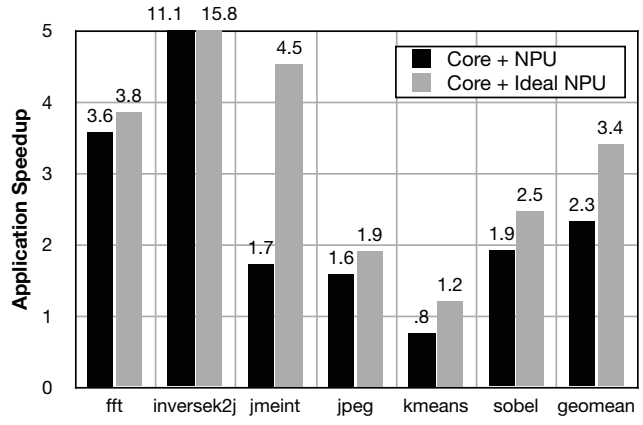
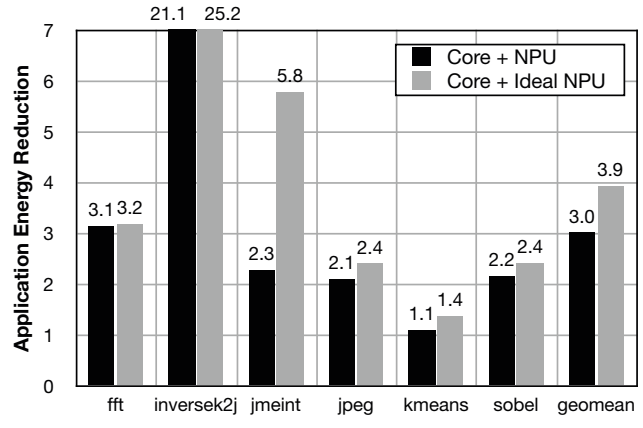


Figure 7: Number of dynamic instructions after Parrot transformation normalized to the original program.



(a) Total application speedup with 8-PE NPU



(b) Total application energy saving with 8-PE NPU

Figure 8: Performance and energy improvements.

while the NN ($6 \rightarrow 8 \rightarrow 4 \rightarrow 1$) for this benchmark is comparatively complex and involves more computation (84 multiply-adds and 12 sigmoids) than the original code. On average, the benchmarks see a speedup of 2.3× through NPU acceleration.

Figure 8b shows the energy reduction for each benchmark. The baseline is the energy consumed by running the entire benchmark on the unmodified CPU and the ideal energy savings for a hypothetical zero-energy NPU. The Parrot transformation elides the execution of significant portion of dynamic instructions that otherwise would go through power-hungry stages of the OoO pipeline. The reduction in the number of dynamic instructions and the energy-efficient design of the NPU yield a 3.0× average application energy reduction.

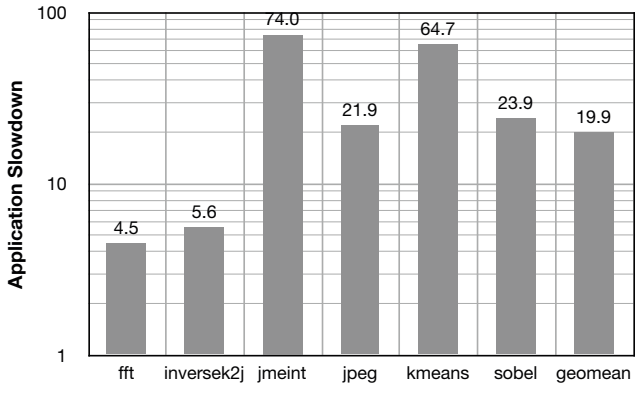


Figure 9: Slowdown with software neural network execution.

For the applications we studied, the Parrot transformation and NPU acceleration provided an average 2.3× speedup and 3.0× energy reduction.

Results for a hypothetical zero-cost NPU suggest that, in the limit, more efficient implementation techniques such as analog NPUs could result in up to 3.4× performance and 3.7× energy improvements on average.

Software neural network execution. While our design evaluates neural networks on a dedicated hardware unit, it is also possible to run transformed programs entirely on the CPU using a software library for neural network evaluation. To evaluate the performance of this all-software configuration, we executed each transformed benchmark using calls to the widely-used Fast Artificial Neural Network (FANN) library [15] in place of NPU invocations. Figure 9 shows the slowdown compared to the baseline (untransformed) execution of each benchmark. Every benchmark exhibits a significant slowdown when the Parrot transformation is used without NPU acceleration. jmeint shows the highest slowdown because 1079 x86 instructions—which take an average of 326 cycles on the core—are replaced by 928 multiplies, 928 adds, and 42 sigmoids. FANN's software multiply-and-add operations involve calculating the address of the neuron weights and loading them. The overhead of function calls in the FANN library also contributes to the slowdown.

The Parrot transformation requires efficient neural network execution, such as hardware acceleration, to be beneficial.

Sensitivity to communication latency. The benefit of NPU-based execution depends on the cost of each NPU invocation. Specifically, the latency of the interconnect between the core and the NPU can affect the potential energy savings and speedup. Figure 10 shows the speedup for each benchmark under five different communication latencies. In each configuration, it takes n cycles to send data to the NPU and n cycles to receive data back from the NPU. In effect, $2n$ cycles are added to the NPU invocation latency. We imagine a design with pipelined communication, so individual enqueue and dequeue instructions take one extra cycle each in every configuration.

The effect of communication latency varies depending on the application. In cases like jpeg, where the NPU computation latency is significantly larger than the communication latency, the speedup is mostly unaffected by increased latency. In contrast, inversek2j sees

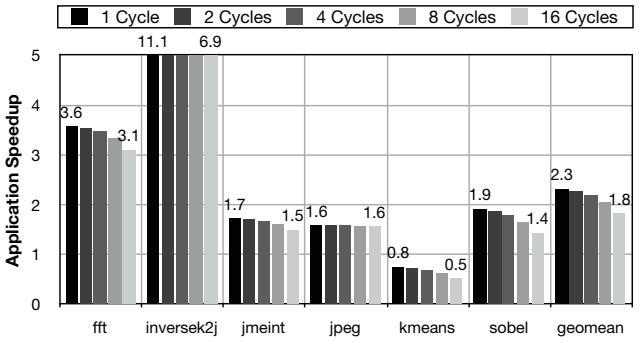


Figure 10: Sensitivity of the application's speedup to NPU communication latency. Each bar shows the speedup if communicating with the NPU takes n cycles.

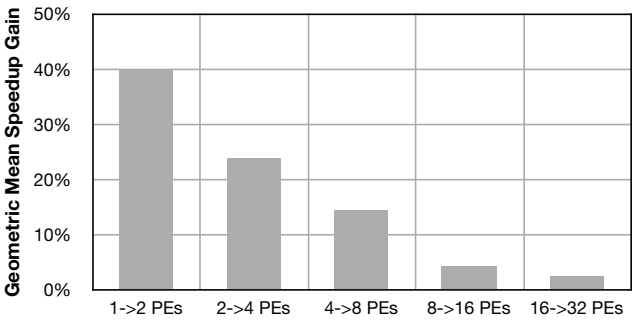


Figure 11: Performance gain per doubling the number of PEs.

a significant reduction in speedup from 11.1× to 6.9× when the communication latency increases from one cycle to 16 and becomes comparable to the computation latency. For kmeans, the slowdown becomes 48% for a latency of 16 cycles compared to 24% when the communication latency is one cycle.

> For some applications with simple neural network topologies, a tightly-coupled, low-latency NPU–CPU integration design is highly beneficial. Other applications we studied can tolerate a higher-latency interconnect.

Number of PEs. Figure 11 shows the geometric mean speedup gain from doubling the number of PEs in the NPU. Doubling the number of PEs beyond eight yields less than 5% geometric mean speedup gain, which does not justify the complexity of adding more than eight PEs for our benchmarks.

8. Limitations and Future Directions

Our results suggest that the Parrot transformation and NPU acceleration can provide significant performance and energy benefits. However, further research must address three limitations to the Parrot transformation as described in this work: (1) applicability; (2) programmer effort; and (3) quality and error control.

Applicability. Since neural networks inherently produce approximate results, not all code regions can undergo the Parrot transformation. As enumerated in Section 3.1, a target code region must satisfy the following conditions:

- The region must be hot in order to benefit from acceleration.
- The region must be approximable. That is, the program must incorporate application-level tolerance of imprecision in the results of the candidate region.
- The region must have a bounded number of statically identifiable inputs and outputs.

Although these criteria form a basis for programmers or compilers to identify nominees for the Parrot transformation, they do not guarantee that the resulting neural network will accurately approximate the code region. There is no simple criterion that makes a certain task (here a candidate region) suited for learning by a neural network. However, our experience and results suggest that empirical assessment is effective to classify a wide variety of approximate functions as NPU-suitable. Follow-on work can improve on empirical assessment by identifying static code features that tend to indicate suitability for learning-based acceleration.

Programmer effort. In this paper, the Parrot transformation requires programmers to (1) identify approximable code regions and (2) provide application inputs to be used for training data collection.

As with the other approaches that ensure the safety of approximate computation and avoid catastrophic failures [41], the programmer must explicitly provide information for the compiler to determine which code regions are safe to approximate. As Section 3.2 outlines, future work should explore allowing the compiler to automatically infer which blocks are amenable to approximation.

Because NPU acceleration depends on representative test cases, it resembles a large body of other techniques that use programmer-provided test inputs, including quality assurance (e.g., unit and integration testing) and profile-driven compilers. Future work should apply traditional coverage measurement and improvement techniques, such as test generation, to the Parrot transformation. In general, however, we found that it was straightforward to provide sufficient inputs for the programs we examined. This is in part because the candidate function is executed many times in a single application run, so a small number of inputs can suffice. Furthermore, as Section 4.2 mentions, an on-line version of the Parrot transformation workflow could use samples of post-deployment inputs if representative tests are not available pre-deployment.

Quality and error control. The results in this paper suggest that NPU acceleration can effectively approximate code with accuracy that is commensurate with state-of-the art approximate computing techniques. However, there is always a possibility that, for some inputs, the NPU computes a significantly lower-quality result than the average case. In other words, without exhaustively exploring the NPU's input space, it is impossible to give guarantees about its worst-case accuracy.

This unpredictability is common to other approximation techniques [11, 41]. As long as the frequency of low-quality results is low and the application can tolerate these infrequent large errors, approximation techniques like NPUs can be effective. For this reason, future research should explore mechanisms to mitigate the frequency of such low-quality results. One such mechanism is to predict whether the NPU execution of the candidate region will be acceptable. For example, one embodiment would check whether an input falls in the range of inputs seen previously during training. If the prediction is negative, the original code can be invoked instead of the NPU. Alternatively, the runtime system could occasionally measure the error by comparing the NPU output to the original function's output. In case the sampled error is greater than a threshold, the neural network can be retrained. These techniques are similar in spirit to related research on estimating error bounds for neural networks [46].

9. Related Work

This work represents a convergence of three main bodies of research: approximate computing, general-purpose configurable acceleration,

and hardware neural networks. Fundamentally, the Parrot transformation leverages hardware neural networks to create a new class of configurable accelerators for approximate programs.

Approximate computing. Many categories of "soft" applications have been shown to be tolerant to imprecision during execution [7, 14, 28, 49]. Prior work has explored relaxed hardware semantics and their impact on these applications, both as (1) extensions to traditional architectures [1, 8, 11, 29] and (2) in the form of fully approximate processing units [4, 24, 26, 32].

In the former category (1), a conventional processor architecture is extended to enable selective approximate execution. Since all the instructions, both approximate and precise, still run on the core, the benefits of approximation are limited. In addition, these techniques' fine granularity precludes higher-level, algorithmic transformations that take advantage of approximation. The Parrot transformation operates at coarser granularities—from small functions to entire algorithms—and potentially increases the benefits of approximation. Furthermore, NPU acceleration reduces the number of instructions that go through the power-hungry frontend stages of the processor pipeline. In the latter category (2), entire processing units carry relaxed semantics and thus require vastly different programming models. In contrast, NPUs can be used with conventional imperative 0 languages and existing code. No special code must be written to take advantage of the approximate unit; only lightweight annotation is required.

Some work has also exposed relaxed semantics in the programming language to give programmers control over the precision of software [2, 8, 41]. As an implementation of approximate semantics, the Parrot transformation dovetails with these programming models.

General-purpose configurable acceleration. The Parrot transformation extends prior work on configurable computing, synthesis, specialization, and acceleration that focuses on compiling traditional, imperative code for efficient hardware structures. One research direction seeks to synthesize efficient circuits or configure FPGAs to accelerate general-purpose code [6, 13, 38, 39]. Similarly, static specialization has shown significant efficiency gains for irregular and legacy code [47, 48]. More recently, configurable accelerators have been proposed that allow the main CPU to offload certain code to a small, efficient structure [18, 19]. These techniques, like NPU acceleration, typically rely on profiling to identify frequently executed code sections and include compilation workflows that offload this "hot" code to the accelerator. This work differs in its focus on accelerating *approximate* code. NPUs represent an opportunity to go beyond the efficiency gains that are possible when strict correctness is not required. While some code is not amenable to approximation and should be accelerated only with correctness-preserving techniques, NPUs can provide greater performance and energy improvements in many situations where relaxed semantics are appropriate.

Neural networks. There is an extensive body of work on hardware implementation of neural networks (neural hardware) both digital [9, 37, 50] and analog [3, 25, 42, 44]. Recent work has proposed higher-level abstractions for implementation of neural networks [23]. Other work has examined fault-tolerant hardware neural networks [22, 45]. In particular, Temam [45] uses datasets from the UCI machine learning repository [16] to explore fault tolerance of a hardware neural network design. That work suggests that even faulty hardware can be used for efficient simulation of neural networks. The Parrot algorithmic transformation provides a compiler workflow that allows general-purpose approximate applications to take advantage of this and other hardware neural networks.

An early version of this work [12] proposed the core idea of automatically mapping approximable regions of imperative code to neural networks. A more recent study [5] showed that 5 of 13 applications from the PARSEC suite can be manually reimplemented to make use of various kinds of neural networks, demonstrating that some applications allow higher-level algorithmic modifications to make use of hardware neural networks (and potentially an architecture like NPUs). However, that work did not prescribe a programming model nor a preferred hardware architecture.

10. Conclusion

This paper demonstrates that neural accelerators can successfully mimic diverse regions of approximable imperative code. Using this neural transformation and the appropriate hardware accelerator, significant application-level energy and performance savings are achievable. The levels of error introduced are comparable to those seen in previous approximate computing techniques. For the technique to be effective, two challenges must be met. First, the program transformation must consider a range of neural network topologies; a single topology is ineffective across diverse applications. Second, the accelerator must be tightly coupled with a processor's pipeline to accelerate fine-grained regions of code. With these requirements met, our application suite ran 2.3× faster on average while using 3.0× less energy and maintaining accuracy greater than 90% in all cases.

Traditionally, hardware implementations of neural networks have been confined to specific classes of learning applications. In this paper, we show that the potential exists to use them to accelerate general-purpose code that can tolerate small errors. In fact, the transformation was successful for every approximable code region that we tested. This acceleration capability aligns with both transistor and application trends, as transistors become less reliable and as imprecise applications grow in importance. NPUs may thus form a new class of trainable accelerators with potential implementations in the digital and analog domains.

Acknowledgments

We would like to thank the anonymous reviewers for their valuable comments. We thank our shepherd, Mike Schlansker, for his feedback and encouragement. We also thank Brandon Lucia, Jacob Nelson, Ardavan Pedram, Renée St. Amant, Karin Strauss, Xi Yang, and the members of the Sampa group for their feedback on the manuscript. This work was supported in part by NSF grant CCF-1016495 and gifts from Microsoft.

References

[1] C. Alvarez, J. Corbal, and M. Valero, "Fuzzy memoization for floating-point multimedia applications," *IEEE Trans. Comput.*, vol. 54, no. 7, 2005.

[2] W. Baek and T. M. Chilimbi, "Green: A framework for supporting energy-conscious programming using controlled approximation," in *PLDI*, 2010.

[3] B. E. Boser, E. Säckinger, J. Bromley, Y. Lecun, L. D. Jackel, and S. Member, "An analog neural network processor with programmable topology," *J. Solid-State Circuits*, vol. 26, pp. 2017–2025, 1991.

[4] L. N. Chakrapani, B. E. S. Akgul, S. Cheemalavagu, P. Korkmaz, K. V. Palem, and B. Seshasayee, "Ultra-efficient (embedded) SOC architectures based on probabilistic CMOS (PCMOS) technology," in *DATE*, 2006.

[5] T. Chen, Y. Chen, M. Duranton, Q. Guo, A. Hashmi, M. Lipasti, A. Nere, S. Qiu, M. Sebag, and O. Temam, "Benchnn: On the broad potential application scope of hardware neural network accelerators?" in *IISWC*, Nov. 2012.

[6] N. Clark, M. Kudlur, H. Park, S. Mahlke, and K. Flautner, "Application-specific processing on a general-purpose core via transparent instruction set customization," in *MICRO*, 2004.

[7] M. de Kruijf and K. Sankaralingam, "Exploring the synergy of emerging workloads and silicon reliability trends," in *SELSE*, 2009.

[8] M. de Kruijf, S. Nomura, and K. Sankaralingam, "Relax: An architectural framework for software recovery of hardware faults," in *ISCA*, 2010.

[9] H. Esmaeilzadeh, P. Saeedi, B. Araabi, C. Lucas, and S. Fakhraie, "Neural network stream processing core (NnSP) for embedded systems," in *ISCAS*, 2006.

[10] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," in *ISCA*, 2011.

[11] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, "Architecture support for disciplined approximate programming," in *ASPLOS*, 2012.

[12] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, "Towards neural acceleration for general-purpose approximate computing," in *WEED*, Jun. 2012.

[13] K. Fan, M. Kudlur, G. Dasika, and S. Mahlke, "Bridging the computation gap between programmable processors and hardwired accelerators," in *HPCA*, 2009.

[14] Y. Fang, H. Li, and X. Li, "A fault criticality evaluation framework of digital systems for error tolerant video applications," in *ATS*, 2011.

[15] FANN, "Fast artificial neural network library," 2012. Available: http://leenissen.dk/fann/wp/

[16] A. Frank and A. Asuncion, "UCI machine learning repository," 2010. Available: http://archive.ics.uci.edu/ml

[17] S. Galal and M. Horowitz, "Energy-efficient floating-point unit design," *IEEE Trans. Comput.*, vol. 60, no. 7, pp. 913–922, 2011.

[18] V. Govindaraju, C.-H. Ho, and K. Sankaralingam, "Dynamically specialized datapaths for energy efficient computing," in *HPCA*, 2011.

[19] S. Gupta, S. Feng, A. Ansari, S. Mahlke, and D. August, "Bundled execution of recurring traces for energy-efficient general purpose processing," in *MICRO*, 2011.

[20] A. Guzhva, S. Dolenko, and I. Persiantsev, "Multifold acceleration of neural network computations using GPU," in *ICANN*, 2009.

[21] R. Hameed, W. Qadeer, M. Wachs, O. Azizi, A. Solomatnikov, B. C. Lee, S. Richardson, C. Kozyrakis, and M. Horowitz, "Understanding sources of inefficiency in general-purpose chips," in *ISCA*, 2010.

[22] A. Hashmi, H. Berry, O. Temam, and M. H. Lipasti, "Automatic abstraction and fault tolerance in cortical microarchitectures," in *ISCA*, 2011.

[23] A. Hashmi, A. Nere, J. J. Thomas, and M. Lipasti, "A case for neuromorphic ISAs," in *ASPLOS*, 2011.

[24] R. Hegde and N. R. Shanbhag, "Energy-efficient signal processing via algorithmic noise-tolerance," in *ISLPED*, 1999.

[25] A. Joubert, B. Belhadj, O. Temam, and R. Heliot, "Hardware spiking neurons design: Analog or digital?" in *IJCNN*, 2012.

[26] L. Leem, H. Cho, J. Bau, Q. A. Jacobson, and S. Mitra, "ERSA: Error resilient system architecture for probabilistic applications," in *DATE*, 2010.

[27] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures," in *MICRO*, 2009.

[28] X. Li and D. Yeung, "Exploiting soft computing for increased fault tolerance," in *ASGI*, 2006.

[29] S. Liu, K. Pattabiraman, T. Moscibroda, and B. G. Zorn, "Flikker: Saving refresh-power in mobile devices through critical data partitioning," in *ASPLOS*, 2011.

[30] S. Misailovic, S. Sidiroglou, H. Hoffman, and M. Rinard, "Quality of service profiling," in *ICSE*, 2010.

[31] N. Muralimanohar, R. Balasubramonian, and N. Jouppi, "Optimizing NUCA organizations and wiring alternatives for large caches with CACTI 6.0," in *MICRO*, 2007.

[32] S. Narayanan, J. Sartori, R. Kumar, and D. L. Jones, "Scalable stochastic processors," in *DATE*, 2010.

[33] NetBSD Documentation, "How lazy FPU context switch works," 2011. Available: http://www.netbsd.org/docs/kernel/lazyfpu.html

[34] K.-S. Oh and K. Jung, "GPU implementation of neural networks," *Pattern Recognition*, vol. 37, no. 6, pp. 1311–1314, 2004.

[35] A. Patel, F. Afram, S. Chen, and K. Ghose, "MARSSx86: A full system simulator for x86 CPUs," in *DAC*, 2011.

[36] A. Pedram, R. A. van de Geijn, and A. Gerstlauer, "Codesign tradeoffs for high-performance, low-power linear algebra architectures," *Computers, IEEE Transactions on*, vol. 61, no. 12, Dec. 2012.

[37] K. Przytula and V. P. Kumar, Eds., *Parallel Digital Implementations of Neural Networks*. Prentice Hall, 1993.

[38] A. R. Putnam, D. Bennett, E. Dellinger, J. Mason, and P. Sundararajan, "CHiMPS: A high-level compilation flow for hybrid CPU-FPGA architectures," in *FPGA*, 2008.

[39] R. Razdan and M. D. Smith, "A high-performance microarchitecture with hardware-programmable functional units," in *MICRO*, 1994.

[40] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning internal representations by error propagation," in *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*. MIT Press, 1986, vol. 1, pp. 318–362.

[41] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman, "EnerJ: Approximate data types for safe and general low-power computation," in *PLDI*, 2011.

[42] J. Schemmel, J. Fieres, and K. Meier, "Wafer-scale integration of analog neural networks," in *IJCNN*, 2008.

[43] S. Sidiroglou-Douskos, S. Misailovic, H. Hoffmann, and M. Rinard, "Managing performance vs. accuracy trade-offs with loop perforation," in *FSE*, 2011.

[44] S. Tam, B. Gupta, H. Castro, and M. Holler, "Learning on an analog VLSI neural network chip," in *SMC*, 1990.

[45] O. Temam, "A defect-tolerant accelerator for emerging high-performance applications," in *ISCA*, 2012.

[46] N. Townsend and L. Tarassenko, "Estimations of error bounds for neural-network function approximators," *IEEE Transactions on Neural Networks*, vol. 10, no. 2, Mar. 1999.

[47] G. Venkatesh, J. Sampson, N. Goulding, S. Garcia, V. Bryksin, J. Lugo-Martinez, S. Swanson, and M. B. Taylor, "Conservation cores: Reducing the energy of mature computations," in *ASPLOS*, 2010.

[48] G. Venkatesh, J. Sampson, N. Goulding, S. K. Venkata, S. Swanson, and M. Taylor, "QsCores: Trading dark silicon for scalable energy efficiency with quasi-specific cores," in *MICRO*, 2011.

[49] V. Wong and M. Horowitz, "Soft error resilience of probabilistic inference applications," in *SELSE*, 2006.

[50] J. Zhu and P. Sutton, "FPGA implementations of neural networks: A survey of a decade of progress," in *FPL*, 2003.

2012 IEEE/ACM 45th Annual International Symposium on Microarchitecture

Designing a Programmable Wire-Speed Regular-Expression Matching Accelerator

Jan van Lunteren[1] Christoph Hagleitner[1] Timothy Heil[2] Giora Biran[3] Uzi Shvadron[3] Kubilay Atasu[1]

[1]IBM Research - Zurich, Switzerland
[2]IBM Systems and Technology Group, Rochester, MN, USA*
[3]IBM Research and Development Labs, Haifa, Israel
{jvl,hle,kat}@zurich.ibm.com timheil@microsoft.com {gbiran,shvadron}@il.ibm.com

Abstract

A growing number of applications rely on fast pattern matching to scan data in real-time for security and analytics purposes. The RegX accelerator in the IBM Power Edge of Network™ (PowerEN) processor supports these applications using a combination of fast programmable state machines and simple processing units to scan data streams against thousands of regular-expression patterns at state-of-the-art Ethernet link speeds. RegX employs a special rule cache and includes several new micro-architectural features that enable various instruction dispatch and execution options for the processing units. The architecture applies RISC philosophy to special-purpose computing: hardware provides fast, simple primitives, typically performed in a single cycle, which are exploited by an intelligent compiler and system software for high performance. This approach provides the flexibility required to achieve good performance across a wide range of workloads. As implemented in the PowerEN™ processor, the accelerator achieves a theoretical peak scan rate of 73.6 Gbit/s, and a measured scan rate of about 15 to 40 Gbit/s for typical intrusion detection workloads.

1. Introduction

Increasing network speeds drive network intrusion detection systems to demand ever greater throughputs. Spiraling sophistication and numbers of attacks have led to a steady rise in the number and complexity of intrusion detection signature sets. At the same time, new applications, such as text analytics for business intelligence, are emerging that also rely increasingly on deep inspection of packet contents.

As a result, regular-expression matching algorithms have re-gained the attention of the research community in recent years. Although the basic research in the field has existed for many decades, the new requirements imposed by fast growing network link speeds and for supporting larger and more complex pattern sets have fostered research into innovative approaches. These are primarily focused on hardware accelerators [11], FPGAs [19, 8] and GPUs [25], because conventional software implementations running on general-purpose CPUs are typically not capable of realizing the target scan rates that can be on the order of several tens of Gbit/s. Most of this work is based on either non-deterministic finite automata (NFAs) or deterministic finite automata (DFAs) [14]. The main advantage of DFAs over NFAs is the lower processing complexity, which makes them more suitable for hardware implementations. The main disadvantage of DFAs is the storage requirements, which can be substantially larger than those of NFAs. Various methods have been developed for remedying the latter issue, several of which will be discussed in Section 10.

*currently with Microsoft IEB

Our contribution presented in this paper consists of the complete design of a novel high-performance regular-expression accelerator called RegX, which includes a hardware implementation as part of the IBM PowerEN™ processor in 45 nm SOI technology, a full software stack, and results measured on the actual hardware. RegX uses programmable state machines, called B-FSMs [30], to perform the basic input scanning using a DFA approach to achieve low processing complexity and high scan rates. The use of multiple parallel B-FSM engines allows to exploit NFA-like properties – multiple parallel states and transitions – to optimize the storage efficiency. RegX combines the B-FSMs with small dedicated processing elements, called local result processors (LRPs). The LRPs provide more advanced processing capabilities for regular-expression matching based on various bit manipulation, count, shift and test operations. The LRP can be viewed as an additional NFA-like mechanism to improve the storage efficiency, whereby problematic NFA states can be offloaded from the B-FSMs. The RegX accelerator is operated under tight control of a sophisticated compiler in a RISC fashion, allowing a range of algorithms, including those developed as part of related work (see Section 10), to be programmed efficiently on the accelerator to achieve good performance for a wide range of workloads.

In this paper, we present the following novel architectural and micro-architectural features of the RegX accelerator:

- The LRP design for regular-expression scanning, in particular its ability to handle eight instructions fully in parallel in every cycle. The LRP supports self-running instructions, which are a new concept to dynamically instantiate autonomously running counters and shift registers within the LRP register file.
- The transition-rule caches, including the global/local address translation and the cache line placement by a software application denoted as upload manager. The upload manager exploits hardware-based profiling to optimize the selection and placement of the most frequently used cache lines.
- The transition-rule cache organization into two banks, used for performing regular and default transition-rule lookups in parallel.
- The physical/logical lane concept involving a time-interleaved processing of multiple streams using multiple B-FSMs and LRPs that can sustain the maximum scan rate of one input character per cycle for each stream when the B-FSMs process out of the rule-caches, without requiring a back-pressure mechanism and independently of the input characteristics. This was key to maximize the single-channel and aggregate scan throughput, while enabling flexible allocation of the scanner resources to the input streams.

The remainder of this paper is organized as follows. Section 2 outlines the design objectives. Section 3 describes the architectural features of RegX and explains the motivations behind its design.

1072-4451/12 $26.00 © 2012 IEEE
DOI 10.1109/MICRO.2012.49

IEEE computer society

Sections 4 and 5 discuss the key elements: the B-FSM engines and the LRP. Section 6 describes how the RegX compiler takes advantage of the architectural features, and Section 7 reveals how these features are implemented for high throughput in the PowerEN™ processor. Section 8 describes how the upload manager software manages the RegX cache hierarchy. Section 9 presents performance figures for this implementation. Section 10 discusses related work, and Section 11 concludes the paper.

2. Design Objectives

The RegX accelerator was designed to meet the demands of modern high-performance intrusion detection systems and analytics applications. These systems require:

1. *Large, complex pattern sets*: The RegX accelerator supports typical pattern sets used in commercial and non-commercial state-of-the-art network intrusion detection systems. These can include several thousands of patterns and involve a mixture of simple string patterns and more complex regular expression patterns.
2. *Multiple contexts*: The pattern set may be divided into hundreds of *contexts*. A context represents a set of patterns which are scanned at one time, and typically targets a specific portion of the input data stream, for example, selected packet protocols. All contexts should be simultaneously active, available for scanning with low latency, and it should be possible to use different contexts in parallel.
3. *High scan rates*: The RegX accelerator detects all matches based on a single-pass on-the-fly processing of the input stream. Aggregate scan rates of about 20 Gbit/s are targeted for typical network intrusion detection workloads. The actual scan rate should be as independent of the input stream characteristics as possible in order to minimize the vulnerability to denial-of-service (DoS) attacks.
4. *Parallel and interleaved scan operations*: The RegX accelerator processes multiple input streams fully in parallel. Parallel scans can be submitted from one or more threads, from the same or different processes, on multiple virtual partitions. Furthermore, multi-session support enables the processing of millions of input streams in an interleaved fashion by storing the complete scanner state when switching from one input stream to another, and restoring that state when switching back to the original stream.
5. *Incremental and dynamic pattern updates:* The architecture supports incremental updates of the pattern set, involving the addition and removal of one or several patterns. The internal scanner data structures can be updated dynamically without interrupting the ongoing scan operations.

3. RegX Architecture Overview

3.1. Processor Bus Interface

The RegX accelerator was designed as a coprocessor directly attached to the processor bus (PBus) as shown in Fig. 1. It provides high-performance regular-expression matching support to software threads running on general-purpose cores. To initiate a scan, a thread sends a command to the RegX accelerator over the PBus in the form of a *co-processor request block* (CRB). The CRB includes pointers to (1) input data, (2) an indication of which context should be used for scanning, and (3) an output buffer. The RegX accelerator will read the input data, scan it against the compiled pattern data, write the scan results into the output buffer, and then notify the software thread.

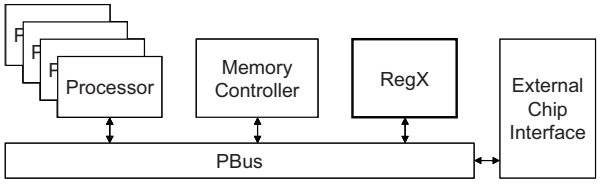


Figure 1: PBus attached RegX accelerator.

Software communicates with the accelerator using normal virtual addresses, without the need for any special handling (e.g., DMA data to specific locations, pin pages, flush caches). The accelerator performs coherent memory accesses, ensuring that input data, pattern data and output buffers are correctly read and written, regardless of where they are cached in the system.

The algorithmic core of RegX is the B-FSM, which will be discussed in Section 4. The B-FSM is a programmable state machine that executes very compact representations of DFAs which define the regular-expression matching functions. These DFAs are created using conventional algorithms that map multiple regular expressions on a single DFA.

Four B-FSMs are combined into a *lane*, the minimum set of resources that will be allocated to execute a single scan command. The four B-FSMs dispatch instructions to a local result processor (LRP), which is discussed in Section 5 and is also contained in the lane. The instructions are embedded within the compiled DFA structures, and operate on a small register file in the LRP. The compiler uses the LRP to split problematic patterns across multiple B-FSMs. The LRP determines when the complete pattern has been matched by identifying when the pieces have been found in the correct order by the B-FSMs. Information flow is one way only; no results flow back from the LRP to the B-FSMs, which greatly eases implementation. The contents of the LRP register file combined with the state of the four B-FSMs form the overall session state of a given lane. This session state is stored to and retrieved from main memory to support multiple sessions.

3.2. Handling the State Explosion Problem

A major challenge faced by practically all DFA-based pattern scanner designs is the so-called state explosion problem that occurs when certain combinations of regular-expression patterns are mapped onto the same DFA, which can result in extremely large DFAs owing to the properties of these patterns [26, 17]. Although no de facto solution exists that completely eliminates this problem, several approaches have been proposed in the literature to address this problem, as will be discussed in Section 10. Two of these approaches are quite effective: 1) the distribution of patterns over multiple parallel DFAs [29, 33, 22] and 2) the extension of DFAs with memory and instructions to operate on it [17, 26, 24]. The RegX architecture was designed specifically to support these two approaches using its lane concept and LRPs. In addition, these mechanisms are flexible enough to extend readily to other algorithms presented in related work. This subsection will illustrate how the lane and LRP concepts can be used to deal with the state explosion problem, whereas Section 10 will compare these concepts in more detail with related work.

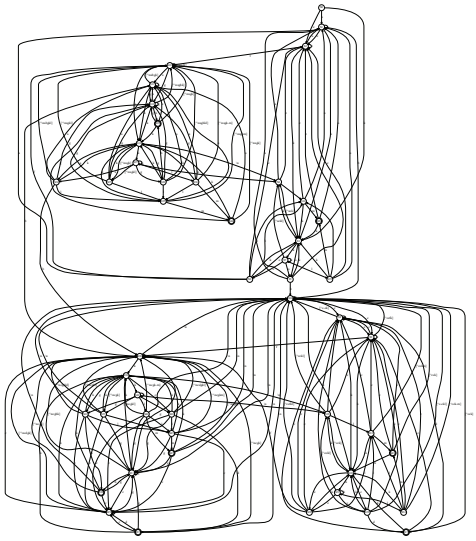
The effectiveness of multiple B-FSMs and the LRP for addressing the state explosion problem can be illustrated by an example. Consider the following three regular-expression patterns: `ab.*cd`, `ef[^\n]*gh` and `k.lm`. The pattern `ab.*cd` will match if the input

stream contains the string `ab`, followed by the string `cd`. There can be zero or more characters of any type in between, as defined by the combination of the dot metasymbol '`.`' and the Kleene star '`*`'. The second pattern is similar, except that there are no newline characters (`\n`) allowed between the strings `ef` and `gh`. The third pattern is also similar, except that there should be exactly one character of any type between the strings `k` and `lm`. For these patterns, standard algorithms can be used to generate the (obviously unreadable) DFA shown in Fig. 2(a). This DFA contains about 50 states and almost 250 transitions — a large expansion over the original three small patterns. This is caused in this case by the '`.*`', '`[^\n]*`' and '`.`' parts of the patterns, which allow a multitude of different overlaps to occur between potentially matching strings, which all have to be covered in the DFA for correct detection. The problem can easily grow worse. For example, increasing the number of dot metasymbols in the third pattern results in even larger DFA sizes. Two dot symbols (`k..lm`) result in a DFA with about 500 transitions, whereas three, four, five, six, and seven dot symbols require DFAs with about 1000, 2000, 4000, 8500, 17000 transitions, respectively, nicely illustrating the exponential nature of the state explosion problem.

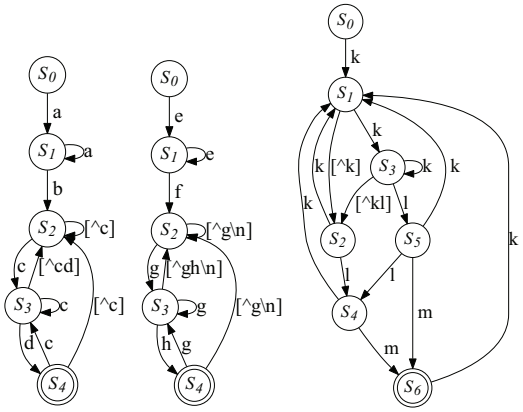
By splitting the patterns among multiple B-FSMs, the compiler can separate problematic patterns. The benefit of this is that the accumulated DFA size can be reduced because the number of possible overlaps that have to be covered by each individual DFA is decreased. This is illustrated in Fig. 2(b), which shows separate DFAs for the three patterns involved in this example, that are executed in parallel by different B-FSMs to process a given input stream and involve a total of 34 transitions. This reduction comes, however, at the cost of using multiple B-FSMs in parallel, which consume additional computation per character, more memory bandwidth and a larger session state. Note that the transitions to the initial state S_0 have been omitted because these are covered by default transitions according to the B-FSM concept that will be discussed in Section 4.

The LRP is a second weapon against the state explosion problem. A part of the scan state can be transferred from the DFAs to registers inside the LRP, which can be manipulated alongside the conventional DFA processing inside the B-FSMs. Using the LRP, problematic regular expressions that cause state explosion can be split into smaller non-problematic subpatterns that can be scanned more efficiently by the B-FSM engines. Rather than generating a match report directly, each of the subpatterns will typically dispatch an LRP instruction. The LRP then checks if these subpatterns are found in the right order and at the right distance to determine whether the original pattern was matched. The LRP instructions are stored in the DFA alongside the state transition rules and are conceptually attached to transitions in the DFA. When an associated transition rule matches, the attached LRP instruction is dispatched. The LRP instructions perform simple operations on general-purpose registers (GPRs) inside the LRP, and can generate a match report when the desired GPR state is reached. This is explained in detail in Section 5.

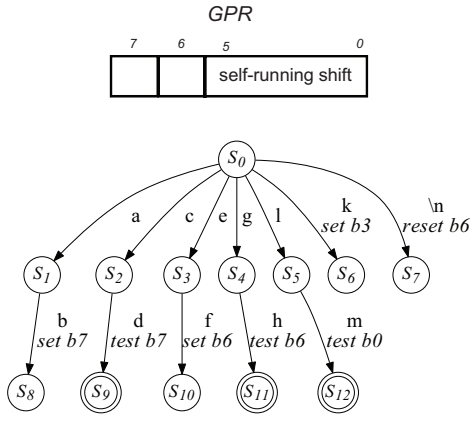
Fig. 2(c) shows a DFA that scans the input stream against the subpatterns `ab`, `cd`, `ef`, `gh`, `k`, `lm` and `\n` (newline) that were split from the original three patterns. The set, test and reset instructions that operate on the GPR shown in Fig. 2(c) allow the LRP to determine the right order of occurrence of these subpatterns, as will be explained in Section 5. By comparing Figs. 2(a) and (c), which both implement the same match function, it is clear that the LRP drastically reduces the size of the DFA. LRP instructions consume storage in the B-FSM rule tables, and the LRP register bits will increase the session state.



(a) Original DFA



(b) Distribution over multiple B-FSMs



(c) Exploitation of LRP

Figure 2: Handling the state-explosion problem.

The trade-offs involved in exploiting the two mechanisms described above are made by the compiler, which is discussed in Section 6.

4. B-FSM Engine

This section briefly reviews the B-FSM technology published in [29, 30]. It also presents a new way of default rule handling that enables more efficient sharing of the rule-caches between multiple pattern contexts and denser rule encoding.

4.1. B-FSM Concept

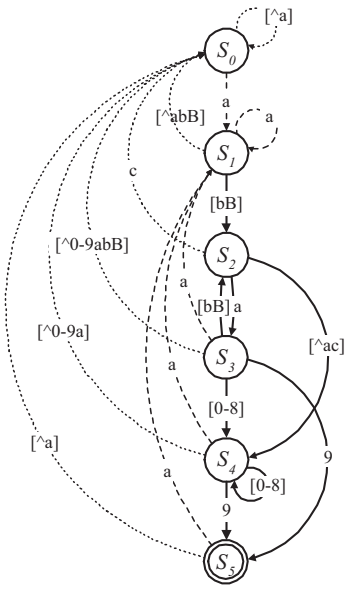
The B-FSM is a programmable state machine that is based on an optimized version of the Balanced Routing Table (BaRT) search algorithm [28], hence the name BART-based FSM (B-FSM). It is programmed using a compiled DFA specification, comprised of *state transition rules* which define flexible conditions for the current state and input and have priorities. This is illustrated in Fig. 3 for an artificial regular expression `a[bB][^c][0-8]*9`, which detects all strings in an input stream that start with an `a`, followed by a `b` (case-insensitive), followed by a character that is not a `c`, followed by zero or more digits in the range from `0` to `8`, and ending with a `9`. Fig. 3(a) shows the DFA that was created for this pattern and Fig. 3(b) a corresponding set of transition rules.

In each cycle, the B-FSM determines the highest-priority transition rule that matches the current state and input value, and branches to the next state indicated by that rule. For example, if the B-FSM is in state S_3 and the current input character is a 9, then rules R_0, R_6 and R_7 would all match. Based on its priority, rule R_7 is selected, resulting in a transition to state S_5 which is consistent with the DFA shown in Fig. 3(a). The correctness of the transition-rule-based specification of the given DFA can be verified in a similar fashion for any state and input value combination. As this example shows, transition rules can involve flexible input conditions including exact match, case-insensitive match, character classes, and negated versions of those conditions.

Fig. 4(a) shows a block diagram of the B-FSM engine. Testing all transition rules for every character is impractical, so the B-FSM organizes the transition rules into multiple hash tables. The B-FSM accesses the hash tables using three registers: (1) the table register indicates which hash table holds the rules of the current state; (2) the state register indicates where in the hash table the rules are stored, and (3) the mask register defines the size and shape of the rules in the hash table.

The B-FSM follows the four-step process for each character illustrated in Fig. 4(a): (1) given the three registers and the current character, use a hash function to generate an index into the hash table, (2) read one *line* containing a fixed number of rules from the hash table, (3) test the current state and character against those rules to determine the matching rule, and (4) update the registers to the next state as indicated by the rules. If no matching rule is found in step (3), then a *default rule* will be selected, as discussed in the next section. Each hash table contains a fixed number of lines (128 in RegX), and each line contains a fixed number of rules (3 in RegX).

The hash function is simple. Each bit in the mask corresponds to one bit of the table index. The corresponding mask bit determines if the index bit is taken from the state vector, or the character. For instance, if the low-order bit of the mask is 1, the low-order bit of the index comes from the character, otherwise it comes from the state vector. By setting many mask bits to one, states with many transition rules can be spread over many lines. By setting few mask bits to one,



(a) DFA

rule	current state	input	→	next state	priority
R_0	*	*	→	S_0	0
R_1	*	a	→	S_1	1
R_2	S_1	[bB]	→	S_2	2
R_3	S_2	[^a]	→	S_4	2
R_4	S_2	a	→	S_3	2
R_5	S_2	c	→	S_0	3
R_6	S_3	[0-9]	→	S_4	2
R_7	S_3	9	→	S_5	3
R_8	S_3	[bB]	→	S_2	3
R_9	S_4	[0-9]	→	S_4	2
R_{10}	S_4	9	→	S_5	3

(b) Transition rules

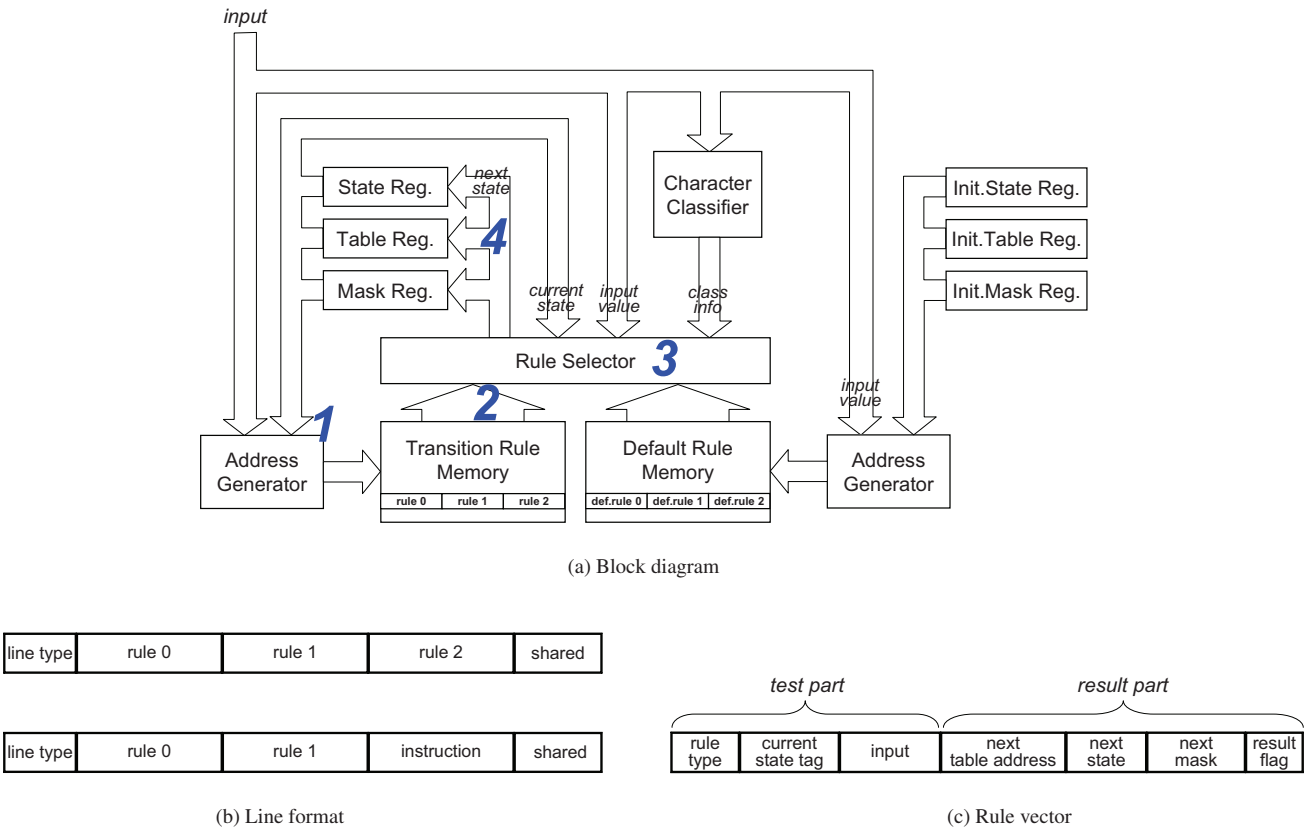
Figure 3: Example DFA and transition rules.

small states can be held compactly in one or a few lines. Note also that rules from multiple states can share a single line.

Fig. 4(b) illustrates a rule line format used by the RegX accelerator. The line may contain three rules or it may contain two rules and an LRP instruction attached to one or more of the rules, which is defined by the line type. In this way, LRP instructions only consume storage when actually used. An additional shared field provides another option for attaching data to the rule vectors.

The rules themselves are shown in Fig. 4(c). The *test part* checks the current state and character against the rule to determine if the rule matches. If the rule matches, the *result part* defines the next state. If the result flag is set, a rule match represents a pattern match and a match report will be generated. The transition rule priorities in Fig. 3(b) are encoded in the order of rules within a line. When multiple rules match, the first matching rule is selected.

The compiler converts the transition rules for each state of Fig. 3 into B-FSM states by selecting a compact mask based on the number and type of the transition rules in each state. It then assigns table IDs, state vectors and masks to states such that states pack efficiently into the tables, without placing more than three rules (or two rules and an LRP instruction) on any one line.



(a) Block diagram

(b) Line format

(c) Rule vector

Figure 4: B-FSM engine.

For more details on the B-FSM, including the integration of instructions inside the data structure, the reader is referred to [30].

4.2. Enhanced Default Rule Handling

Default rules are transition rules that involve a "don't care" state condition and consequently apply to all states. Examples of such rules are rules R_0 and R_1 in Fig. 3(b), which cover multiple transitions in the DFA in Fig. 3(a) that are represented using dotted and dashed lines, respectively. Because the default rules only depend on the input, a simple lookup on the input value can be used to determine the highest-priority matching default rule, which will then be used by the rule selector if no regular matching transition rule is found.

In Fig. 4(a), the original default lookup table described in [29] has been replaced by a default-rule memory in which the default rules are stored using the same hash concept as the regular rules, and which is accessed in parallel. The rule selector will now select the matching rule from the regular transition rules and default rules retrieved from the two rule memories, which are tested in parallel. Because the default rules have the lowest priority, these are only selected if no matching regular rule is found. Although this results in exactly the same default rule being selected as with the original scheme based on a table lookup, the new approach enables default rules from different pattern contexts to be resident at the same time in the default-rule memory. Another advantage is that LRP instructions can now be attached to default rules in the same storage-efficient way as to regular transition rules (see [30]). This type of instruction is called a default instruction and is handled slightly different from regular instructions: if the highest-priority matching default rule in a given cycle has a default instruction attached, then that instruction will always be dispatched to the LRP regardless of whether a higher-priority matching regular rule exists. This allows the LRP to execute certain instructions when particular input values occur, without the need to attach these instructions to all transitions in the DFA that involve those input values.

Analysis of DFAs compiled for actual intrusion detection pattern sets revealed that many states exist for which the transition rules defined cover the entire input value space. An example of such a state is state S_2 in Fig. 3(b). As a result, the B-FSM will always find a matching regular transition rule for those states, rendering the default-rule lookup useless. A storage optimization technique called dual-hash was developed to exploit this. It is based on mapping the lowest-priority rules for such a state in a separate hash table stored in the default-rule memory. A special instruction will indicate how this table is accessed in parallel to the regular transition-rule memory lookup by providing additional table and mask vectors. The rule selection is done as before. The main advantage is that the mapping and lookup of the transition rules on two parallel hash tables allow a further compression of the data structure.

5. Local Result Processor (LRP)

The LRP constitutes an important feature of the RegX accelerator targeted at fighting the state-explosion problem as described in Section 3.2. In addition, it allows scanner features to be supported that cannot be implemented using standard DFAs.

Fig. 5 shows a high-level block diagram of the LRP. It contains a register file comprised of general-purpose registers (GPRs) that

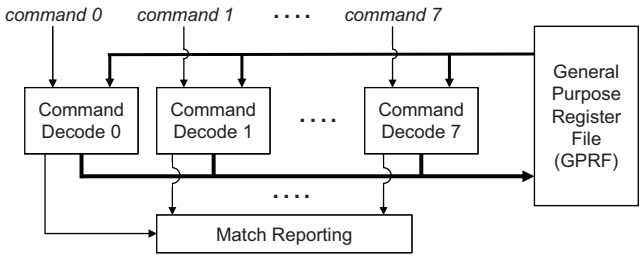


Figure 5: Local Result Processor.

can be manipulated using the LRP instruction set, which includes operations such as set, reset, load immediate, count and shift. These instructions also have conditional versions that are only executed if the conditions specified are tested positively against selected bits or bytes in the GPRs. In addition, the register file also includes multiple offset registers (OFRs) for storing copies of the input stream read pointer (offset) that can be processed and tested at a later stage.

The LRP contains eight identical command units that simultaneously handle the maximum of four default instructions and four regular LRP instructions that the four B-FSM engines in a single lane can dispatch in each cycle to the LRP as described in Section 4. The compiler ensures that these maximum numbers will never be exceeded for each lane, for example, by adapting the distribution of the patterns over the B-FSMs in a lane or by allocating additional lanes to scan a given pattern set. As a result, the LRP can sustain the full peak scan rate of the RegX accelerator without requiring a back-pressure algorithm. Moreover, at the same time the input stream characteristics cannot impact the overall scan throughput by influencing the LRP operation. The actual load of the LRP depends on the type of patterns, in particular if there are many complex patterns or pattern combinations that have to be split and processed by the LRP.

The basic LRP operation will now be illustrated using the example introduced in Section 3.2 involving the three regular-expression patterns `ab.*cd`, `ef[^\n]*gh` and `k.lm`. Fig. 2(c) shows a DFA for detecting matches against the subpatterns `ab`, `cd`, `ef`, `gh`, `k`, `lm` and `\n` (newline) that are split from these three patterns. This DFA is extended with several LRP instructions operating on a GPR (only 8 bits shown) to allow the detection of matches on the original patterns. If, for example, a match on `ab` is detected, then the corresponding transition from state S_1 to S_8 will result in a set instruction on bit 7. If a string `cd` is detected at a later stage (transition from state S_2 to S_9), a test instruction is performed on the same bit. A positive test result would then imply that the string `ab` had already been found before and that a match is detected on the pattern `ab.*cd`. The LRP is used in a similar way to check for matches against the pattern `ef[^\n]*gh`, now involving a set and test operation on GPR bit 6. The only difference is that newline characters are not allowed to occur between the two subpatterns, as represented by `[^\n]*`. This is enforced by a default instruction that resets bit 6 each time a newline character occurs (see the transition from S_0 to S_7). As a result, bit 6 will equal one only if the first subpattern `ef` was detected in the input stream not followed by a newline character. If the second subpattern `gh` is found, then a positive test on bit 6 (transition from state S_4 to S_{11}) indicates a match on the original pattern `ef[^\n]*gh`.

As this example shows, often only one or a few bits in a given GPR will be allocated to maintain state for a particular regular-expression pattern. To make efficient use of the available GPR bits, multiple

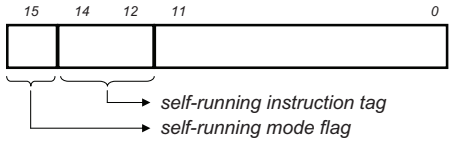


Figure 6: General-purpose register with self-running tag.

Table 1: Self-running instruction vectors.

bits 15-12	instruction	bits 15-12	instruction
`0xxxb`	nop		
`1000b`	6-bit shift reg.	`1100b`	30-bit shift reg.
`1001b`	12-bit shift reg.	`1101b`	36-bit shift reg.
`1010b`	18-bit shift reg.	`1110b`	8-bit counter
`1011b`	24-bit shift reg.	`1111b`	12-bit counter

patterns can share a single GPR. This is realized by allowing instructions to select one or several GPR bits in a flexible way with selection masks, and by allowing multiple instructions to operate in parallel on the same GPR. Multiple instructions of specific types may operate on the same GPR bit position. In that case, a priority order is defined (e.g., set takes precedence over reset, which has priority over shift) that clearly specifies how each bit position will be updated. For other instruction types, the compiler stack (see Section 6), which performs the actual bit allocation and instruction generation, guarantees that no conflicts can occur on the same GPR bits.

Note that the three artificial regular expressions used in the above example, although representative for actual cases, are only intended to illustrate the basic operation of the LRP. Actual pattern sets can include more complex patterns that require multiple splits and involve different types of overlaps of the split portions requiring special handling using instructions beyond simple set and test operations (see for example [2]). This topic, however, is outside the scope of this paper.

Since only the LRP has enough information to know when a complex pattern has been matched, the LRP generates the match report in these cases. The B-FSM dispatches a match report instruction to the LRP, which may conditionally check GPR state prior to generating the match report. Additional forms of the match instruction can also copy instruction data and/or GPR contents to the match report for further processing by a so-called software result processor (SRP).

5.1. Self-Running Register Operation

Certain pattern forms benefit from operations, that are performed for every input character in certain states. For example, incrementing a GPR for every character can be used to measure the distance between two (sub)patterns, or the length of a matching string. For these cases, the LRP has been extended with autonomously self-running instructions.

Every GPR can be placed into a self-running mode by storing a special code in the high-order bits of the GPR, as shown in Fig. 6. Bit 15 enables self-running operation. When enabled, bits 12 to 14 indicate the operation performed for each input character. Otherwise bits 0 to 14 may be used as normal GPR bits. The LRP supports autonomous counters and shift registers of various sizes, as listed in Table 1. As shown in Fig. 7, the LRP supports shift registers of up to 36 bits by daisy-chaining the shift registers across adjacent GPRs.

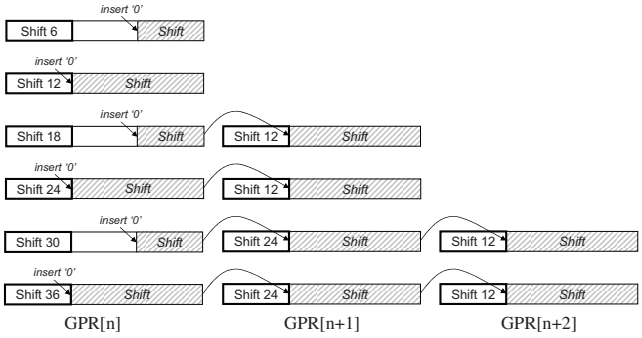


Figure 7: Daisy-chained self-running shift registers.

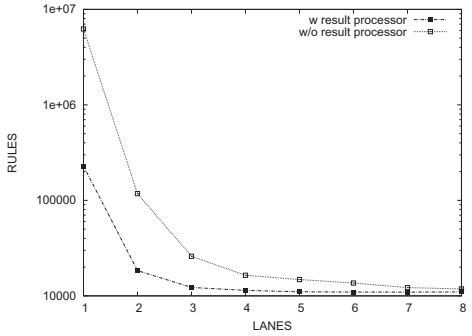


Figure 8: Lane count and LRP impact on storage efficiency.

Configuration and activation of a self-running instruction on a given GPR are done by writing the self-running mode flag bit, the self-running instruction vector and an initial value of the GPR bits covered by the self-running instruction using a regular GPR instruction. The self-running mode can be deactivated by resetting the flag bit. During the self-running operation, regular LRP instructions can be used to write and test GPR bits in the self-running region of the GPR, for example, to write one or several bits into a shift register or to test whether a counter has reached a certain value.

A self-running shift operation will now be illustrated using a variation of the third pattern in the example introduced in Section 3.2 involving two dot symbols, (`k..lm`) as this allows interleaved matches. This pattern is matched if the input stream contains a `k` character at exactly four character positions before the `lm` subpattern is detected (counting back from the position of the last character `m`). The interesting property of this pattern is that it can match several interleaved strings in the input stream: For example, the input string `abckkklmlmdef` will result in two matches, one caused by the underlined characters and one caused by the characters in bold font. A self-running shift operation involving 6 bits (as shown in Fig. 2(c)) can be used to efficiently handle this pattern, while supporting interleaved matching. Each time a `k` is detected in the input stream, a default instruction (attached to the transition from S_0 to S_6 in Fig. 2(c)), will set bit 3 in the self-running shift register portion of the GPR. It will also set bits 12 to 15 to enable 6-bit shift-register operation. After four cycles, this set bit will have been shifted towards bit position 0. If a subpattern `lm` is detected (transition from S_5 to S_{12}), then this bit 0 is tested. If it equals one, it implies that a `k` character occurred exactly four characters earlier and a match will be reported on the pattern `k..lm`. Note that variations of this pattern involving one to seven dot symbols as mentioned in Section 3.2 are all matched using the same DFA shown in Fig. 2(c), only differing in the bit position involved in the set instruction attached to the transition from state S_0 to state S_6.

This concept of self-running register operation enables a flexible and dynamic allocation of the available register resources for use as variable-width counters and shift registers, or as general-purpose registers, while these can all be operated using standard instructions such as set, reset and load.

6. Pattern Compilation

The compilation of the patterns into a data structure that can be executed by the B-FSMs and LRPs in the RegX accelerator is performed in a two-step process by the so-called *pattern compiler* and *B-FSM compiler*.

The pattern compiler converts the pattern sets into an intermediate structure comprised of multiple DFAs that can include LRP instructions. This conversion includes a preprocessing step in which complex regular expressions are split into simpler subpatterns while accompanying instructions are generated that allow the LRP to determine whether the original pattern was matched based on the detection of these subpatterns, as was illustrated in Section 5. This step also allocates the LRP register bits used by these instructions. By considering constraints arising from the instruction generation (e.g., maximum number of parallel instructions, available register bits) in combination with other pattern-set-related properties, the pattern compiler will then select the number of lanes and B-FSM engines that will be used to scan the pattern set. If the compiler decides to use more than one lane, it will break the context into multiple so-called *sub-contexts*, each of which is compiled on a separate lane. This is followed by an intelligent distribution of the patterns in the (sub)contexts over the B-FSM engines in those lanes, aiming to optimize storage efficiency and performance [22]. Finally, the pattern compiler will use standard techniques [14] to create DFAs from the regular-expression patterns that are allocated to each B-FSM.

In the next step, the B-FSM compiler will convert each DFA into a B-FSM data structure comprised of linked hash tables as described in Section 4, by applying state encoding and hash function selection for each individual state. A range of optimizations ensures a high compression of the data structure while supporting flexible transition conditions and efficient integration of LRP instructions. For more details, refer to [29, 30].

To illustrate the compiler's capability to improve the storage-efficiency by exploiting multiple lanes and the LRP, several compilation experiments were performed using the publicly available Linux layer-7 application protocol classifier [1]. Fig. 8 shows the accumulated size of the DFAs generated by the pattern compiler, expressed in transition rule count, for compiles involving one to eight lanes, with and without LRP support. These results show that already for this relatively small pattern set very large improvements in storage efficiency can be obtained. For example, compiling at two lanes with LRP support results in a rule reduction by over a factor 34 compared with a single-lane compile without LRP. The resulting B-FSM structure consumes about 128 KB.

7. PowerEn^{TM} RegX Implementation

The RegX accelerator has been implemented in the IBM PowerEN^{TM} processor, which closely couples general-purpose processor cores, hardware accelerators and I/O in a system on a chip (SoC) [10, 13]. The chip has been realized in 45-nm SOI technology and measures

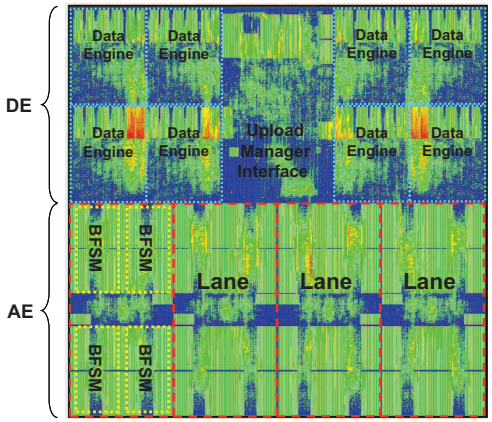


Figure 9: PowerEN™ RegX floorplan.

a total of 410 mm^2, out of which 15.4 mm^2 is used to implement the RegX accelerator (Fig. 9). RegX runs at a nominal frequency of 2.3 GHz, although some portions are clocked at half speed.

7.1. Data Engine and Algorithmic Engine

The RegX engine comprises two main parts: a data engine (DE) and an algorithmic engine (AE). These are illustrated in Fig. 10. The data engine enqueues and schedules the scan commands it receives from the PBus, and fetches and transmits the input data involved in those scans to the algorithmic engine. The data engine also controls the storage and retrieval of the scan state when switching between different sessions and the writing of the scan results into the output buffers. The algorithmic engine contains the lanes with the B-FSMs and the LRPs and performs the scanning function. The area is almost equally divided between the algorithmic engine and the data engine.

7.2. Multiple Lanes

RegX contains four independent physical lanes, each with four B-FSMs and two LRPs. To process a character as discussed in Section 4, the B-FSM reads the DFA rules in one cycle and computes the next state in the next cycle in a pipelined fashion. This makes it easy to run two independent scans on the same physical lane in a time-multiplexed fashion. Hence the AE has two *logical lanes* overlaid on each of the four *physical lanes*, and can process 8 scans simultaneously.

To ease timing and render wiring less complicated, the LRPs run at half the main clock, and are physically duplicated on each physical lane to support the two logical lanes. This makes it possible to implement the LRP functionality at the target frequency without stalls or hazards. The LRPs can execute all the instructions that the B-FSMs can dispatch every cycle. The LRP register file consists of eight 16-bit GPRs comprising a total of 128 bits.

In total there are eight logical lanes with 32 logical B-FSMs and eight LRPs, which can scan eight independent input streams in parallel. Each logical lane can process one byte every other cycle, each physical lane can process a byte every cycle, and the peak throughput is four bytes per cycle, yielding a theoretical peak throughput of 73.6 Gbit/s at 2.3 GHz. The peak scan rate for a single stream is one byte every other cycle or 9.2 Gbit/s.

7.3. B-FSM Cache Hierarchy

As shown in Fig. 4(a), the B-FSM must access both transition rules and default rules for each character scanned. The two rule memories are implemented as a two-level memory hierarchy. The L1 rule cache consists of the two SRAM banks for each B-FSM shown in Fig. 10, with one bank caching the default rules and the other bank caching the regular transition rules. Each bank contains 16 KB of storage for a total of 512 KB of L1 rule cache over the 16 physical B-FSMs. If the required rules are not in the L1 rule cache, then they are fetched from main memory over the PBus – resulting in a penalty of as much as 400 cycles. To reduce this penalty, scans are removed from the logical lane during the miss, and the logical lane is reallocated to another scan if one is available. Because the set of transition rules is typically much larger than the set of default rules, the banks used for each rule type can be selected on a per-context basis: By placing the default rules on one bank for some contexts and on the other bank for others, the space pressure on the two banks can be balanced.

The L1 rule cache must be accessed in a single 2.3 GHz cycle, including both tag access and compare, which is faster than the L1 data cache in the processor core. This severely limits the number of tags and the associativity that could be implemented for a hardware-managed cache. For this reason, each of the two rule cache banks is divided into two regions. A 14 KB *locked area* is managed by the upload manager software (see Section 8). The remaining 2 KB comprise the *temporary area*, which is implemented as a two-way set-associative cache with 64 B lines.

The locked area is kept tagless by exposing this region to the B-FSM architecture as an addressable memory area. When the upload manager places a state in the locked area, it modifies all rules that transition to that state so that they point directly at the physical location in the locked area. A special flag in the next state information of the rule indicates that the next state is in the locked area, rather than in the global rule memory. When the B-FSM detects it is in a locked state, it can deterministically compute the SRAM index directly from the state information. Because rules can be updated to point to almost any location in the locked area, the locked area is nearly fully-associative.

There are a number of cases where the B-FSM state is exposed to software. For instance, state is saved and restored from main memory to support multiple sessions. Also, matches are reported using the B-FSM state that generated the match. Because the locked area is exposed as a new architected location, the upload manager effectively changes the address or name of a state when it moves a state into the locked area. To make this transparent to software, the RegX hardware will translate locked addresses to and from global addresses as needed. Hence, match reports and session state are always stored to memory in global form. When B-FSM state is read back in for the next packet in a session, it is converted from global form to locked form if the state is in the locked area. This translation process is managed by the *global/local address translation unit* (GLAT) in the RegX accelerator. The upload manager configures this table to indicate the global address corresponding to locked states in the locked area.

8. Upload Manager

The main task of the upload manager is to place the most frequently accessed transition rules into the locked area. This approach was chosen after initial experiments with the RegX architecture revealed that pure hardware managed rule caches did not result in good performance for large target workloads. The limitation of two-way set-associativity was simply insufficient to handle workloads involving frequent switches between large numbers of competing pattern contexts.

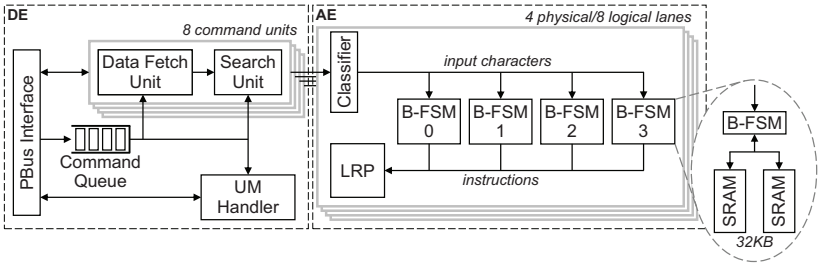


Figure 10: PowerEN™ RegX block diagram.

The placement algorithm is driven by a statistical profile of rule access patterns, produced by dedicated performance counters embedded within the RegX unit. A profile indicates the fraction of rule accesses performed to every state in all active pattern contexts. Significant changes in the profile over time will trigger a re-placement of the locked states.

An optimal placement of the B-FSM states is similar to the bin packing problem, known to be NP-hard. However, state placement has both additional degrees of freedom, and additional constraints. While general solutions such as linear programming can be effective, the run time of such solutions would be impractical. The heuristic approach described here was found to give results similar to highly-associative hardware caches, and still have reasonable runtime.

The algorithm uses the following steps to select a placement from a profile.

1. Context replication: If a pattern context is used frequently, it may benefit from being placed on multiple physical lanes. A single physical lane allows two scans of the same context to proceed in parallel (18.4 Gb/s peak throughput). By replicating the context on multiple lanes, the peak throughput for a single pattern context is increased, at the cost of using more rule cache. Replicated contexts must be cached on each physical lane they are placed on, and they must be laid out in exactly the same way on each physical lane. A very simple algorithm is used based on the fraction of characters scanned by each context.
2. Placed context size estimation: This step estimates the amount of space used by each context in the set. For this step, we treat the entire locked cache area as if it were one large cache, and simply fill it up, ignoring most constraints. DFA states are given priority based on the *access density* of the state:

$$AccessDensity = FractionOfAccesses / SizeOfState \quad (1)$$

   The basic concept is that we will maximize the number of accesses that are locked, if we maximize the number of accesses per byte of storage. Hence, small states are locked before large states, and frequently accessed states are locked before rarely accessed states. The result of this step is an estimate of the number of bytes that will be locked for each (sub)context.
3. Physical lane and B-FSM mapping: This step assigns physical lanes to the contexts, and selects how the contexts will be laid out on the lane. The RegX unit has the ability to alter which physical B-FSM is used for each logical B-FSM in each (sub)context.
   This step attempts to balance the cache space pressure of the contexts, based on the size estimates from step 2, with load balancing across the physical lanes, known from profile. A greedy algorithm processes contexts one at a time, starting with the most frequently used. The number of possible layouts of a context is relatively small – for an unreplicated context, four physical lanes times 24 B-FSM mappings. Evaluation is fast so we essentially evaluate them all and pick the best.
4. Physical state placement: After step 3, the B-FSM for each DFA state is known. We process all states greedily, following the access density function of Step 2. The algorithm attempts to find a good fit for each state avoiding fragmentation. States which do not fit at all are not locked.

Once the placement has been determined, the respective rules are loaded into the locked area. The replacement of the existing content is entirely transparent to the application, and can occur while searches are running.

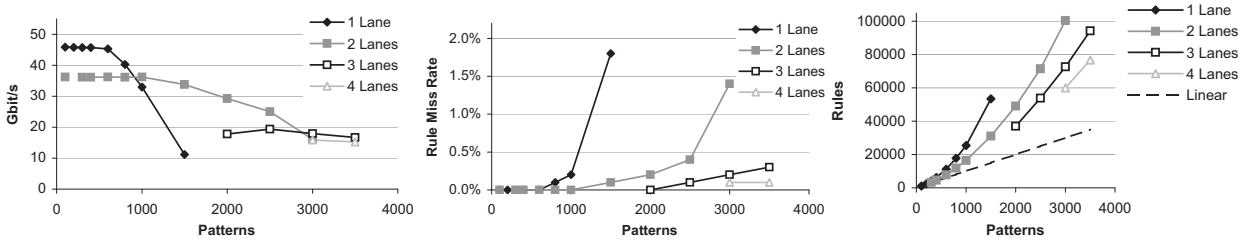
To load a state into the local memory, the upload manager first copies the rules from global memory into the B-FSM cache. Then it programs the global-local address translation unit so that the RegX knows how to perform global/local translation for the new state (as per Section 7.3). Finally, it updates the rules in global memory to point to the copy of the state in the B-FSM cache. During some parts of this process, scans may be using both the cached copy and the global copy, which is fine since both copies are semantically identical. Unloading a cache state follows a similar process in reverse.

9. Performance Evaluation

9.1. Context Size

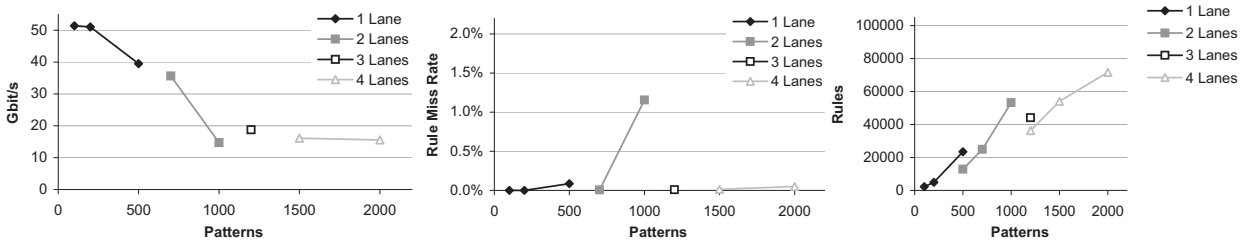
The pattern matching throughput of a single context is largely driven by a trade-off between the rule miss rate and the number of sub-contexts. Fig. 11 shows results for a synthetic context made up entirely of simple fixed-string patterns. These string patterns consist of patterns made up of uniform distributions of upper- and lower-case letters and digits. Each pattern is a case-sensitive unanchored string that is 10 characters long. Example patterns are: `GGdTOObQHh`, `tw3XAPWgNi`, `VgfkkGAkZ6`. Using synthetic workloads allows us to uniformly vary key parameters (e.g., pattern size, number of patterns), and make the set publicly available [15]. In addition, we also show results for large complex pattern sets, which are not widely available at this time.

The input data stream can strongly affect the rule miss rate. Most of the input characters are chosen from a random uniform distribution over letters and digits. However, random data does not stress the B-FSMs realistically because they will not tend to traverse a wide variety of DFA states. By interspersing byte sequences that match the initial pieces of the patterns, the B-FSMs are driven to explore a much wider variety of states, stressing the rule cache hierarchy. The sequence intervals are chosen from an exponential distribution with a mean of 100 characters, and the sequence length is chosen from an exponential distribution with a mean of 2. The target pattern for the pattern-based sequence is random uniform over all patterns in the pattern context. All scans are 1000 B long.



(a) (b) (c)

Figure 11: Results for simple string patterns.



(a) (b) (c)

Figure 12: Results for complex patterns.

Returning to Fig. 11, the number of patterns in the context is plotted on the X-axis. The Y-axis of graph (a) contains the measured throughput. Throughout this section, performance is the average of three identical measurements. Individual measurements typically vary by less than 1%. Graph (b) plots the miss rate, which is measured as DFA rule misses per logical character scanned. With a single sub-context, a single logical character could theoretically generate up to 8 misses (2 banks times 4 B-FSMs), and 16 misses with two sub-contexts. Hence miss rates above 100% are possible in extreme situations. Graph (c) plots the size of the context in terms of DFA rules. As described previously, three rules fit in a 16 B line, yielding 5.3 B/rule. However, the size of the contexts in bytes is larger than this because of LRP commands and inefficiencies in packing. In general, the contexts use six to seven bytes per rule.

We plot four curves for each of 1 to 4 sub-contexts. Performance for a single sub-context is a constant 45 Gbit/s up to 600 patterns. In this range, there are no rule misses. Beyond this, performance starts to fall because of increasing miss rates as the context grows. At 1000 patterns, performance is better with two sub-contexts. The total context size drops by 35% and the miss rate drops back to zero, more than offsetting the cost of scanning every character twice. This trend continues with 3 and 4 sub-contexts. The context grows to the point where rule misses become a significant hinderance to performance, at which point another sub-context is added. The result is a gentle performance degradation from 45 Gbit/s with small contexts (<600 patterns) on one sub-context to 17 Gbit/s with 3500 patterns on 4 sub-contexts.

Results are similar for the complex pattern set, see Fig. 12. The complex pattern sets consist of a mix of simple strings, extended string-like elements and *separator* elements. The extended string-like elements contain broad character classes (`[a-z]`, `[A-Z]`, `[0-9]`), alternatives (`(abc|xyz|123)`) and optional components (`abc(123)?`). They represent 35% of all string elements. In addition, 20% of the patterns contain multiple such string-like elements separated by wildcards and repeat constructs (`.`, `.*`, `[^"]*"`, `[^\n\r]*`, ...). Of the patterns, 50% are case insensitive, and 20% are front-anchored (`^`). Table 2 shows several example patterns from the complex pattern sets.

Table 2: Example patterns from the complex pattern sets.

Example Pattern	Description
`6zfe4tOikumP`	Simple string pattern
`cbPvd(WT)?`	Extended string pattern with optional element
`^jVC6(Wj|zq|A7)vX[^\n\r]*3FFh3GAOz`	Two string elements with separator, front-anchored
`E[a-z]kRqDMf.*xbgyHH.*9Ia3UbnPdhW4tq[A-Z]`	Three string elements with separators

About 500 complex patterns fit into a single sub-context, as compared to about 1000 for the string patterns. One significant difference with the complex set is that sub-contexts are borderline LRP-resource bounded. The string patterns did not use the LRP at all, and the number of sub-contexts was completely determined by the performance trade-off. The complex patterns make heavy use of the LRP, and as the LRP has limited state, the number of sub-contexts is dictated by the amount of LRP state needed to efficiently represent the patterns. As a result, sub-contexts cannot grow overly large, and miss rates are typically low (Fig. 12(c)).

9.2. Multiple Contexts

RegX can support even larger numbers of patterns using multiple contexts. Fig. 13 shows the throughput (a) and miss rate (b) when running multiple contexts. Each context contains 1000 string or complex patterns. To facilitate the comparison with the preceding graphs, the total pattern count is plotted on the X-axis. For instance, 4000 patterns represent four contexts, each with 1000 patterns.

The string contexts use a single sub-context each, whereas the complex contexts use two sub-contexts each. The input data is as before, but distributed among the contexts in a random uniform fashion.

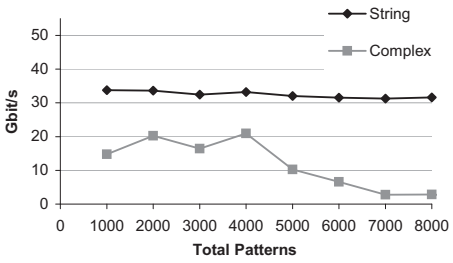


Figure 13: Results for multiple contexts.

This is a worst-case scenario. Real-world workloads typically use a few contexts much more frequently than others.

First of all, performance for the string patterns is completely flat around 32 Gbit/s. Why? The answer has to do with how the upload manager assigns physical lanes to sub-contexts. With a single active sub-context, the context is placed on all four physical lanes to maximize throughput. When the workload changes to two contexts/sub-contexts, the upload manager places one context on two physical lanes, and the second on the other two physical lanes. As a result, the rule pressure on the B-FSM caches does not increase, and performance is unchanged. Similar reasoning applies to 4 contexts/sub-contexts.

Typically two large sub-contexts fit on one physical lane. This is due to the flexible assignment of default rules and transition rules to the two rule caches in each B-FSM: The small default rule set packs efficiently with the larger transition rule set from another sub-context. As a result, up to 8 contexts, 8000 patterns, fit on the engine with minimal performance loss. The complex pattern results follow a similar trend, but the complex contexts are about twice the size. Performance remains relatively flat up to 8 sub-contexts, four contexts, and then tapers off as the miss rates increase.

9.3. Worst-Case Performance

For small and medium-sized pattern sets that fit entirely into the rule caches, RegX delivers a constant, maximal scan rate that is essentially independent of the input stream and patterns, and is only affected by the number of sub-contexts used, as the latter determines the memory bandwidth consumed per input character. For large pattern sets containing several thousand patterns that exceed the size of the rule caches, the scan rate depends primarily on the rule miss rate. The latter is influenced by the input stream characteristics, as these affect which portions of the data structure, cached or non-cached, are accessed in each step. The worst-case scenario would be almost no hits in the transition-rule caches, in which case the scan rate would drop below 1 Gbit/s. However, this is an inherent problem that is common to all scanner designs based on a memory hierarchy.

The use of a hardware- and software-managed cache structure in RegX controlled by the upload manager makes it unlikely that the worst-case situation will ever occur for typical intrusion detection pattern sets which have a relatively narrow working set (e.g., usually very few matches are found). Furthermore, it will be very hard to create a DoS attack to exploit this, because one would have to create special input streams targeted at the pattern sets used, that adapt at the unknown rate of the upload manager to create a worst-case situation. At the same time, the upload manager can actively monitor for the occurrence of DoS attacks, and react accordingly.

10. Related Work

To the best of our knowledge, this RegX paper is the first publication that combines a hardware implementation, an upload manager, a software-based extension (SRP), and a compiler to consistently achieve scan rates of 20 Gbps for large pattern sets and a wide range of workloads. This section will compare the new architectural and micro-architectural features of the RegX accelerator, as listed in Section 1, with their counterparts in related work.

As outlined in Section 1, most work on regular-expression scanners is based on either NFAs or DFAs. Although NFA-based hardware acceleration has been investigated, in particular using FPGAs [27, 20, 23, 32], these approaches typically do not scale efficiently to larger pattern sets, involve large session states, which renders multi-session support more difficult, and cannot support fast dynamic pattern updates because of high reconfiguration times. Therefore, this section will focus primarily on DFA-based pattern scanning, which is also the underlying processing model of the B-FSM engines in the RegX accelerator.

Various methods have been proposed for DFA compression. These include techniques for regular expression rewriting [33], merging of non-equivalent states [6], and character class support and alphabet reduction [9, 3, 30, 5], which are compared with the B-FSM concept in more detail in [30]. Storage optimizations based on the removal of redundant transitions by replacing these with default transitions or related concepts [12, 16, 18, 4] try to exploit the property that DFAs for regular-expression matching typically involve many states that transit to identical next states based on the same input characters. The B-FSM engines use this default-transition concept in two ways: based on the enhanced default-rule handling presented in Section 4.2 and using so-called common rules [30]. Both concepts are implemented in the B-FSM such that exactly two independent memory accesses are required to process each input character, which are performed fully in parallel in a single cycle on the two memory banks that make up the rule cache. This results in a constant scan rate when the B-FSMs operate directly out of the rule caches. In contrast, the default-transition mechanisms applied by the schemes presented in [4, 18] involve a variable number of transitions for each input character, with the actual number being influenced by the input characteristics, resulting in a non-deterministic scan rate.

Besides the above methods that target compact *representations* of given DFAs, other approaches have been proposed that try to optimize the DFAs *themselves*. These include the distribution of patterns over multiple parallel DFAs such that the aggregate DFA size is minimized [29, 33, 22] and the extension of DFAs with memory and instruction execution logic [17, 26, 24]. These approaches combine the DFA concept with NFA-like features to achieve good compression [7, 31, 21]. The approaches involving parallel DFAs allow the separation of problematic pattern combinations that cause a state explosion when mapped together on the same DFA, whereas the approaches involving memory and instruction execution logic allow individual problematic patterns to be split into simpler subpatterns (see Section 3.2). Both types of approaches are supported by the RegX accelerator through its lane concept in which four B-FSM engines connected to an LRP are used to scan a given input stream, whereas multiple lanes can be allocated to a single scan operation. The RegX lane concept differs from the related work that is referred to above in two essential aspects: (1) a single LRP can execute instructions originating from multiple DFAs, and (2) each LRP can process eight instructions in parallel in a single cycle (the compiler guarantees that this will never be

exceeded), allowing it to sustain the maximum scan rate of a single lane without requiring any back-pressure mechanism. In contrast, the related work that we are aware of only involves configurations with a single processing unit per DFA, and these processing units execute instructions in a serial fashion, which can cause variations in the scan rate and makes it dependent on the input characteristics.

A key feature of the entire RegX design that distinguishes it from other pattern scanners is that the applied algorithms and mechanisms (including the B-FSMs, the LRPs, and the rule caches) were designed to process each input character in a constant number of cycles that is independent of the input characteristics when operating out of the rule caches. Moreover, they were designed such that a maximum aggregate scan rate of one character per cycle can be sustained by applying pipelining techniques. This enabled a simpler and more efficient scheduling of the processing of multiple streams in parallel, and rendered RegX less vulnerable to DoS attacks than other schemes that involve scan rates that depend on input characteristics, for example, because of the type of DFA compression or instruction execution logic they used.

11. Conclusion

This paper has presented a new regular-expression accelerator, called RegX, which targets state-of-the-art network intrusion detection systems. The RegX accelerator is a complete design that has been implemented as part of the IBM PowerEN^{TM} processor in 45 nm SOI technology, and for which a corresponding software stack has been created. RegX combines several novel (micro)-architectural features that are exploited by the software stack to obtain a theoretical scan performance of 73.6 Gbit/s. Hardware measurements were presented showing scan rates on the order of 15 to 40 Gbit/sec for typical intrusion detection workloads.

Acknowledgment

The authors would like to thank Charlotte Bolliger and Anne-Marie Cromack for their help with the preparation of this manuscript.

References

[1] "Application layer packet classifier for linux." [Online]. Available: http://l7-filter.sourceforge.net/

[2] K. Atasu *et al.*, "Hardware-accelerated regular expression matching with overlap handling on IBM PowerEN processor," Research Report RZ3833, 2012. Available: http://domino.research.ibm.com/library/cyberdig.nsf/papers/ACA8409641D63C4A85257A8C004E0F9F

[3] M. Bando *et al.*, "Range hash for regular expression pre-filtering," in *Proc. Architectures for Networking and Communications Systems, ANCS*, 2010, pp. 1–12.

[4] M. Becchi and P. Crowley, "An improved algorithm to accelerate regular expression evaluation," in *Proc. Architectures for Networking and Communications Systems, ANCS*, 2007, pp. 145–154.

[5] M. Becchi and P. Crowley, "Efficient regular expression evaluation - theory to practice," in *Proc. Architectures for Networking and Communications Systems, ANCS*, 2008, pp. 50–59.

[6] M. Becchi and S. Cadambi, "Memory-efficient regular expression search using state merging," in *Proc. IEEE INFOCOM*, 2007, pp. 1064–1072.

[7] M. Becchi and P. Crowley, "A hybrid finite automaton for practical deep packet inspection," in *Proc. of the ACM CoNEXT conference*, 2007, pp. 1–12.

[8] J. Bispo *et al.*, "Regular expression matching for reconfigurable packet inspection," in *Proc. of IEEE Intl. Conf. on Field Programmable Technology (FPT)*, 2006, pp. 119 –126.

[9] B. Brodie, R. Cytron, and D. Taylor, "A scalable architecture for high-throughput regular-expression pattern matching," in *Proc. 33rd Int. Symp. on Computer Architecture ISCA*, 2006, pp. 191–202.

[10] J. Brown *et al.*, "IBM Power Edge of Network processor: A wire-speed system on a chip," *IEEE Micro*, vol. 31, no. 2, pp. 76 –85, 2011.

[11] Cavium Networks, "Nitrox DPI CN17XX L7 Content Processor Family," Product Brief, 2009.

[12] D. Ficara *et al.*, "Differential encoding of DFAs for fast regular expression matching," *IEEE/ACM Transactions on Networking*, vol. 19, no. 3, pp. 683–694, 2011.

[13] H. Franke *et al.*, "Introduction to the wire-speed processor and architecture," *IBM J. of Res. Develop.*, vol. 54, no. 1, pp. 3:1 –3:11, 2010.

[14] J. E. Hopcroft, R. Motwani, and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, 3rd ed. Addison-Wesley Longman Publishing Co., Inc., 2006.

[15] IBM, "PowerEN PME Public Pattern Sets Wiki," 2012. Available: https://www.ibm.com/developerworks/mydeveloperworks/wikis/home?lang=en#/wiki/PowerEN%20PME%20Public%20Pattern%20Sets/page/Welcome

[16] S. Kumar, J. Turner, and J. Williams, "Advanced algorithms for fast and scalable deep packet inspection," in *Proc. Architectures for Networking and Communications Systems, ANCS*, 2006, pp. 81–92.

[17] S. Kumar *et al.*, "Curing regular expressions matching algorithms from insomnia, amnesia, and acalculia," in *Proc. Architectures for Networking and Communications Systems, ANCS*, 2007, pp. 155–164.

[18] S. Kumar *et al.*, "Algorithms to accelerate multiple regular expressions matching for deep packet inspection," in *Proc. ACM SIGCOMM*, 2006, pp. 339–350.

[19] C.-H. Lin *et al.*, "Optimization of pattern matching circuits for regular expression on FPGA," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 15, no. 12, pp. 1303 –1310, 2007.

[20] A. Mitra, W. Najjar, and L. Bhuyan, "Compiling PCRE to FPGA for accelerating SNORT IDS," in *Proc. Architectures for networking and Communications Systems, ANCS*, 2007, pp. 127–136.

[21] H. Nakahara, T. Sasao, and M. Matsuura, "A regular expression matching circuit based on a decomposed automaton," in *ARC'11, Lecture Notes in Computer Science*, 2011, vol. 6578, pp. 16–28.

[22] J. Rohrer *et al.*, "Memory-efficient distribution of regular expressions for fast deep packet inspection," in *Proc. Int. Conf. on Hardware Software Codesign, CODES+ISSS*, 2009, pp. 147–154.

[23] R. Sidhu and V. Prasanna, "Fast regular expression matching using FPGAs," in *Proc. 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, FCCM*, 2001, pp. 227–238.

[24] R. Smith, C. Estan, and S. Jha, "XFA: Faster signature matching with extended automata," in *Proc. IEEE Symp. on Security and Privacy, SP*, 2008, pp. 187–201.

[25] R. Smith *et al.*, "Evaluating GPUs for network packet signature matching," in *Proc. IEEE Int. Symp. Performance Analysis of Systems and Software, ISPASS*, 2009, pp. 175–184.

[26] R. Smith *et al.*, "Deflating the big bang: fast and scalable deep packet inspection with extended finite automata," *Comput. Commun. Rev.*, vol. 38, pp. 207–218, 2008.

[27] I. Sourdis *et al.*, "Regular expression matching in reconfigurable hardware," *Journal of Signal Processing Systems*, vol. 51, pp. 99–121, 2008.

[28] J. van Lunteren, "Searching very large routing tables in wide embedded memory," in *Proc. IEEE Globecom*, 2001, pp. 3–1615.

[29] J. van Lunteren, "High-performance pattern-matching for intrusion detection," in *Proc. IEEE INFOCOM*, 2006, pp. 1–13.

[30] J. van Lunteren and A. Guanella, "Hardware-accelerated regular expression matching at multiple tens of Gb/s," in *Proc. IEEE INFOCOM*, 2012, pp. 1737–1745.

[31] Y.-H. Yang and V. Prasanna, "Space-time tradeoff in regular expression matching with semi-deterministic finite automata," in *INFOCOM, 2011 Proceedings IEEE*, april 2011, pp. 1853 –1861.

[32] Y.-H. Yang, W. Jiang, and V. Prasanna, "Compact architecture for high-throughput regular expression matching on FPGA," in *Proc. Architectures for Networking and Communications Systems, ANCS*, 2008, pp. 30–39.

[33] F. Yu *et al.*, "Fast and memory-efficient regular expression matching for deep packet inspection," in *Proc. Architectures for Networking and Communications Systems, ANCS*, 2006, pp. 93–102.

Author Index

Author Index

IEEE Computer Society Technical & Conference Activities Board

T&C Board Vice President
Paul R. Croll
Computer Sciences Corporation

IEEE Computer Society Staff
Evan Butterfield, *Director of Products and Services*
Lynne Harris, *CMP, Senior Manager, Conference Support Services*
Alicia Stickley, *Senior Manager, Publishing Operations*
Silvia Ceballos, *Manager, Conference Publishing Services*
Patrick Kellenberger, *Supervisor, Conference Publishing Services*

IEEE Computer Society Publications

The world-renowned IEEE Computer Society publishes, promotes, and distributes a wide variety of authoritative computer science and engineering texts. These books are available from most retail outlets. Visit the CS Store at *http://www.computer.org/portal/site/store/index.jsp* for a list of products.

IEEE Computer Society *Conference Publishing Services* (CPS)

The IEEE Computer Society produces conference publications for more than 300 acclaimed international conferences each year in a variety of formats, including books, CD-ROMs, USB Drives, and on-line publications. For information about the IEEE Computer Society's *Conference Publishing Services* (CPS), please e-mail: cps@computer.org or telephone +1-714-821-8380. Fax +1-714-761-1784. Additional information about *Conference Publishing Services* (CPS) can be accessed from our web site at: *http://www.computer.org/cps*

Revised: 18 January 2012

CPS Online is our innovative online collaborative conference publishing system designed to speed the delivery of price quotations and provide conferences with real-time access to all of a project's publication materials during production, including the final papers. The ***CPS Online*** workspace gives a conference the opportunity to upload files through any Web browser, check status and scheduling on their project, make changes to the Table of Contents and Front Matter, approve editorial changes and proofs, and communicate with their CPS editor through discussion forums, chat tools, commenting tools and e-mail.

The following is the URL link to the ***CPS Online*** Publishing Inquiry Form:
http://www.computer.org/portal/web/cscps/quote