CoRAM: An In-Fabric Memory Architecture for FPGA-Based Computing

Submitted in partial fulfillment of the requirements for

the degree of

Doctor of Philosophy

in

Electrical and Computer Engineering

Eric S. Chung

B.S., Electrical Engineering and Computer Science
University of California, Berkeley

Carnegie Mellon University
Pittsburgh, PA

August, 2011

Abstract

Field Programmable Gate Arrays (FPGA) have been used in many applications to achieve orders-of-magnitude improvement in absolute performance and energy efficiency relative to conventional microprocessors. Despite their newfound potency in both processing performance and energy efficiency, FPGAs have not gained widespread acceptance as mainstream computing devices. A fundamental obstacle to FPGA-based computing can be traced to the FPGA's lack of a common, scalable memory abstraction. When developing for FPGAs, application writers are often responsible for crafting the application-specific infrastructure logic that transports the data to and from the processing kernels, which are the ultimate producers and consumers within the fabric. Very often, this infrastructure logic not only increases design time and effort but will inflexibly lock a design to a particular FPGA product line, hindering scalability and portability. To create a common, scalable memory abstraction, this thesis proposes a new FPGA memory architecture called Connected RAM (CoRAM) to serve as a portable bridge between the distributed computation kernels and the edge memory interfaces. In addition to improving performance and efficiency, the CoRAM architecture provides a virtualized memory environment as seen by the hardware kernels to simplify application development and to improve an application's scalability and portability.

Acknowledgements

I owe all of my success to my academic advisor, Professor James Hoe, who taught me how to think, speak, and write. It has been a tremendous privilege to work with James, who generously provided the resources and the comfortable environment that enabled me to pursue my ideas to fruition. I am sincerely grateful for his consistent encouragement, insight, and support throughout all these years.

I also thank Professor Ken Mai for his constant source of feedback throughout my time at CMU and for inviting me into his office when I had yet-another-circuit-question. I thank Professor Babak Falsafi for his intellectual contributions to my research projects, his genuine support for those all around him, and his encouragement and advice even all the way from Switzerland. I also thank my external committee members Professor Guy Blelloch, Professor Joel Emer, and Professor John Wawrzynek for serving on my defense. Their helpful comments and constructive criticisms significantly enhanced the quality of this thesis. I also wish to thank various members of the RAMP community for the friendships and the interactions that I have benefited from over so many years: Professor Derek Chiou, Hari Angepat, Nikhil Patil, Michael Pellauer, Professor Mark Oskin, and Andrew Putnam, and Professor Martha Mercaldi-Kim. I would also like to thank my mentors at Microsoft Research, Chuck Thacker and John Davis.

I would like to acknowledge the National Science Foundation (CCF-1012851), C2S2, CSSI, Sun Microsystems, Microsoft Research, and John and Claire Bertucci for providing the generous funding that supported my work at CMU. I would like to thank Bluespec and Xilinx for their tool donations.

There are many fellow graduate students and friends whom I wish to acknowledge. Jared Smolens, Brian Gold, and Jangwoo Kim were valuable mentors to me when I joined the TRUSS project. Thanks to Jared for being a great friend, for his feedback, for maintaining our cluster, and for his ability to proofread anything I could throw at him. Brian always gave valuable opinions and was never too busy to answer my questions; thanks also to Brian and his wife Robin for making me feel welcome when I first arrived to CMU. Jangwoo was always supportive and providing helpful late-night discussions and advice on demand. Nikos Hardavellas was a sounding board for many ideas in the wee hours of the night. I hope Nikos will forgive me for the events of ISCA'05. Michael Papamichael provided the network-on-chip generator that helped to make this thesis possible. Thanks to Michael for being a constant source of feedback, for introducing me to the best Pad-See-Ew in all of existence, and for accompanying me on our adventure to New York's finest Brazilian BBQ. Peter Milder is a great friend, collaborator, and always found time to stop by and chat. Peter Klemperer was an entertaining officemate and friend who kept me up-to-date with his endless stories and projects. Yoongu Kim was my mutual commiserator and kept me company during the late work hours. I am grateful to Tom Wenisch, who frequently brought me out to half-priced margaritas at Mad Mex and gave me good advice on many subjects. I am also grateful to Mike Ferdman for his feedback on many of my research projects; thanks for Mike and Alisa Ferdman for also inviting me to their parties. Thanks to Gabriel Weisz, who endlessly tolerated my design bugs and also shared his homebrew with us. Thanks goes out to Eriko Nurvitadhi, who tolerated me as his roommate and officemate for nearly 7 years. In addition, I also greatly enjoyed the company and interactions with many other members of CMU and CALCM: Roland Wunderlich, Stephen Somogyi, Evaggelis Vlachos, Ippokratis Pandis, Vasileios Liaskovitis, Kai Yu, Wei Yu, Jennifer Black, Qian Yu, Professor Onur Mutlu, Chris Craik, Chris Fallin, Theodoros Strigkos, Marek Telgarsky, Bob Koutsoyannis, Kristen Dorsey, Ryan Johnson, and Brett Meyer.

I'd like to thank Elaine Lawrence, Reenie Kirby, Matt Koeske, and all the other assistants in our department for handling all the administrative duties that made things run smoothly for all the students. Susan Farrington always made me feel welcome and appreciated at CMU. The Isomac and Expobar machines deserve a special thanks for distilling the endless amounts of caffeine that transformed into this thesis.

I would like to thank Edward Lin and Fabian Manan who were great friends who kept my life interesting beyond the lab. Thank you to old friends who kept close from afar: Ben Liu, Patrick Charoenpong, Brandon Ooi, Mike Liao, Chethana Kulkarni, Kazim Zaidi, and Menzies Chen.

I am grateful to our cats, Cho Cho and Mi Mi, who kept us company all these years with an unlimited supply of entertainment, exasperation, and cat hair.

I cannot thank my parents enough for the sacrifices that they have made so that I could be here today. My family members, Alan, Lily, Melody, and Katie have been a pillar of support throughout my life and time at CMU.

Finally, I express my gratitude to Kari Cheng. As the significant other of an ‘eternal’ graduate student, Kari tolerated my habits, my unusual hours, and my times of absence with love, grace, and understanding. This journey would not have been complete without her.

Contents

List of Tables

List of Figures

To my family and Kari

Chapter 1

Introduction

And now for something completely different.

Monty Python

Over the past several decades, the defining characteristic of general-purpose processors has been their programmability and flexibility. With steady improvements in VLSI technology, general-purpose processors have attained tremendous levels of success and adoption. In recent years, the deviation from classical scaling law [37] has begun to threaten the sustained scalability of future multicores built solely out of general-purpose cores [23]. Although transistor dimensions continue to shrink with each major technology node, transistor threshold and supply voltages have not been scaling as commensurately due to leakage effects. The inability to scale the supply voltage has now constrained us to an era where power—not area—is becoming the ultimate limiting resource. With the recent emphasis on power efficiency coupled with the ever-increasing appetite for high computational throughput, designers must think beyond classical von Neumann architectures that fundamentally sacrifice efficiency for their generality.

In the quest for energy-efficient computing, Field Programmable Gate Arrays (FPGAs) have emerged as a class of general-purpose accelerators with high potential to address the demand for high performance and efficiency. FPGAs comprise a sea of programmable logic gates that can be reconfigured on-demand to accelerate a particular problem or task. Despite their proven advantages, today's commercial FPGAs are not built in mind for computing and have mostly eluded mainstream adoption in general-purpose computing. A fundamental limitation of the FPGA is its lack of a standard memory architecture. Beyond having to developing the core logic, FPGA application writers are burdened by a low-level fabric abstraction that exposes complex details specific to particular devices and platforms. Applications often lack portability, and substantial effort must be invested to develop and/or optimize the cloud of infrastructure that surrounds the core logic of the application.

To address these challenges, the contribution of this thesis is a major reconception of the architectural paradigm of FPGA-based computing. The thesis presents a top-to-bottom exploration of a new, all-purpose memory architecture called Connected RAM (CoRAM), which simplifies difficult aspects of FPGA programming and memory management while enhancing the programmability and portability of applications.

1.1 FPGAs for Computing

The merits of FPGA-based reconfigurable computing have been known since the early 1990s [35]. FPGAs consist of up to millions of tiny, programmable lookup tables that can be configured "in the field" to implement arbitrary logic functions. Unlike the sequential-style programming languages of general-purpose processors, FPGA "programs" are typically captured with hardware description languages that expose a highly concurrent abstraction to the user. FPGAs have been demonstrated in many cases to accelerate a wide variety of applications ranging from financial analytics, bioinformatics, physics, and databases [98, 118, 87, 53]. FPGAs typically achieve their feats through massive, fine-grained parallelism and the pipelining of arbitrary dataflows.

1.2 Limitations of Conventional FPGAs

While accumulated VLSI advances have steadily improved the FPGA fabric's processing capability, FPGAs have yet to gain widespread acceptance as mainstream computing devices. A commonly cited obstacle is the difficulty in programming FPGAs using low-level hardware description languages and development flows. A further problem lies in the fact that FPGAs today are not architected for computing purposes but rather to emulate or replace ASICs.

While the former concern is being gradually addressed by advances in high-level synthesis [45, 17], a more fundamental problem lies in the FPGA's lack of a stable and standard "architecture" for

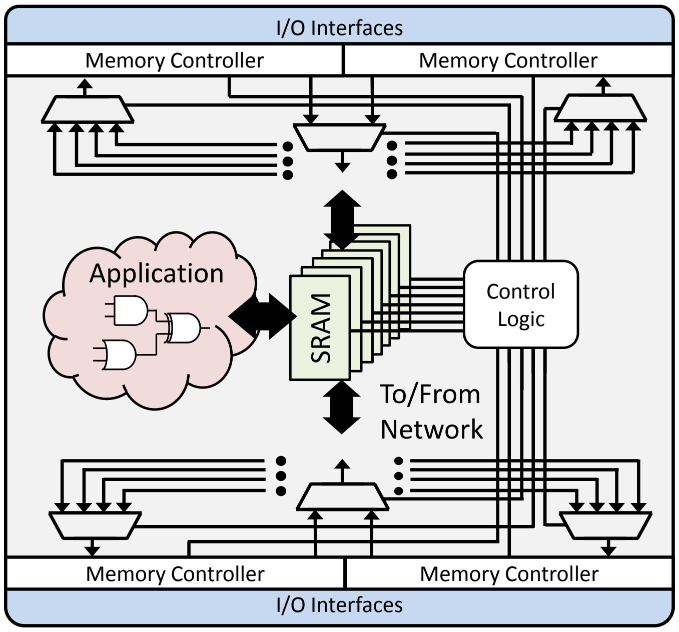


Figure 1: Application Writer's View of a Conventional FPGA.

application writers. Unlike their general-purpose counterparts, conventional FPGAs expose nothing to the user but a "sea" of logic and a collection of external I/O pins. A general-purpose processor by contrast exports its computational and memory resources through a generic and structured interface (i.e., Instruction Set Architecture) that abstracts away the machine's underlying details.

When developing for the FPGA, a designer often has to create from bare fabric not only the target application kernel itself but also the application-specific infrastructure logic, which requires detailed knowledge of the specific device and platform being utilized. This infrastructure often includes user-developed mechanisms to support and optimize the transfer of data to and from external DRAM interfaces and to I/O devices (see Figure 1). Very often, creating or using this infrastructure logic not only increases design time and effort but will frequently lock a design to a particular FPGA platform or device, hindering scalability and portability. Further, the support mechanisms which users are directly responsible for will be increasingly difficult to manage in the future as: (1) the number of on-die SRAMs and off-chip DRAM interfaces increase in number and become more distributed across the fabric, and (2) long-distance interconnect delays become more difficult to tolerate in larger fabric designs, requiring more spatial awareness by the user to distribute memory data effectively.

1.3 FPGAs Lack A Standard Memory Architecture

The root of the aforementioned challenges can be traced to the fact that current FPGAs lack essential abstractions and built-in mechanisms that one comes to expect in a general purpose computer—i.e., an Instruction Set Architecture (ISA) that defines a standard agreement between hardware and software. From a computing perspective, a standard and stable architectural definition is a critical ingredient for programmability and for application portability.

A crucial starting point for addressing this challenge is to rethink how the notion of "memory" should be presented and architected within an FPGA. In many cases, the lack of application portability stems from unenforced separation between the application kernel itself and the mechanisms needed to obtain its data from the environment (i.e., main memory and I/O). To specifically address the challenges related to memory on FPGAs, the central goal of this thesis is to create a shared, scalable, and portable memory architecture suitable for future FPGA-based computing devices. Such a memory architecture would be used in a way that is analogous to how general purpose programs universally access main memory through standard "loads" and "stores" as defined by an ISA—without any knowledge of hierarchy details such as caches, memory controllers, etc. At the same time, the FPGA memory architecture definition cannot simply adopt what exists for general purpose processors and instead should reflect the spatially distributed nature of today's FPGAs—consisting of up to millions of interconnected LUTs and thousands of embedded SRAMs [116].

Working under the above premises, the guiding principles for the desired FPGA memory architecture are:

- The architecture should present to the user a common, virtualized appearance of the FPGA fabric, which encompasses reconfigurable logic, its external memory interfaces, and the multitude of SRAMs—while freeing designers from details irrelevant to the application itself.
- The architecture should provide a standard, easy-to-use mechanism for controlling the transport of data between memory interfaces and the SRAMs used by the application throughout the course of computation.
- Applications for the architecture should port effortlessly to newer devices and platforms without modification.

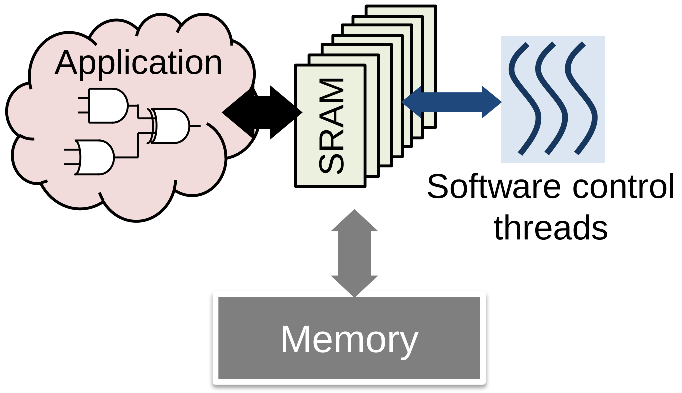


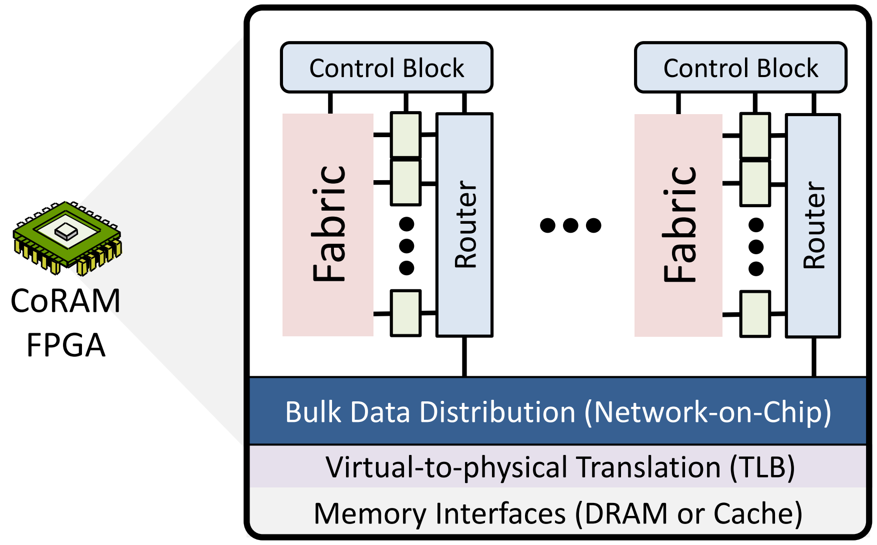
Figure 2: User's View of an FPGA with CoRAM Support.

- The architecture should be amenable to scalable implementations on the FPGA without affecting the architectural view presented to existing applications.

1.4 The Connected RAM Memory Architecture

To satisfy the above goals, this thesis proposes the Connected RAM (CoRAM) memory architecture for future FPGAs designed for general-purpose computing. The CoRAM architecture allows the application writer to focus on creating efficient, high-performance kernels within a virtualized FPGA environment while relying on a standard set of abstracted and distributed memory mechanisms to sustain the kernel's data consumption and production. As shown in Figure 2, an FPGA with CoRAM support presents a highly simplified view of the fabric to the user, consisting of programmable logic, on-die SRAMs, and a standard software abstraction for mediating accesses to main memory.

To facilitate portability, application logic is never permitted to directly access the edge memory interfaces nor the I/O pins that are typical in modern FPGAs. A unique characteristic of the CoRAM architecture is that the SRAMs themselves are used as portals into global memory and are programmed using a stylized, high-level language over the course of a computation. The use of a portable language to specify the memory requirements of a given application effectively "virtualizes" a kernel and would make it possible to easily relocate a kernel within a fabric or to port an application between different FPGA families with a common CoRAM architecture. The CoRAM abstraction itself is also naturally distributed to suit the nature of FPGA-like fabrics and can be used



CoRAM FPGA
Control Block
Control Block
Fabric
Router
Fabric
Router
Bulk Data Distribution (Network-on-Chip)
Virtual-to-physical Translation (TLB)
Memory Interfaces (DRAM or Cache)

Figure 3: CoRAM: Physical Implementation.

to form convenient memory structures with higher-level semantics without the loss of portability or efficiency.

Beneath the CoRAM abstraction lies a high-performance microarchitecture designed to scale up to hundreds to thousands of "connected" CoRAMs. Figure 3 illustrates an island-style microarchitecture devised in this thesis that distributes control threads and core logic components across replicated clusters of aggregated CoRAMs and control blocks. In between the clusters, a high-performance network-on-chip provides global connectivity between the end-points and the off-chip memory interfaces. An important property of the microarchitecture shown in Figure 3 is the speedup provided by hardening of the network-on-chip and the clustering logic. The over-provisioning of bandwidth and latency of general-purpose mechanisms has the notable benefit of allowing applications to achieve near-ideal performance potential and efficiency with modest overheads in die area and power, which will be demonstrated in Chapter 8.

1.5 Thesis Contributions

The scope of this thesis encompasses two major research thrusts: (1) defining a proper memory abstraction both useful to a wide range of FPGA applications, and (2) the investigation of the underlying hard and soft mechanisms needed to support the abstraction effectively. The contributions of this thesis are:

- Identification and rationale for essential abstractions in the CoRAM architecture.
- The demonstrated effectiveness of CoRAM in comparison to conventional approaches.
- An exploration of the microarchitectural design space for CoRAM on conventional and future FPGAs.
- Prototype RTL designs that demonstrate the feasibility of CoRAM.
- A study of the performance and efficiency gap between CoRAM and conventional development approaches.
- An investigation on determining which mechanisms are needed in future FPGAs to support the CoRAM architecture effectively.

1.6 Thesis Outline

The remainder of this thesis is organized as follows. Chapter 2 gives background on FPGAs and their applications in computing. Chapter 3 introduces the CoRAM architectural paradigm. Chapter 4 describes the Cor-C Architecture Specification, which is a devised instance of the CoRAM paradigm. Chapter 5 describes several case studies using Cor-C. Chapter 6 gives a detailed treatment of the CoRAM microarchitecture. Chapter 7 describes the prototype implementation of CoRAM. Chapter 8 gives an evaluation of the CoRAM microarchitecture. Chapter 9 discusses related work. Chapter 10 offers conclusions and directions for future work.

Chapter 2

Background

Traditionally, FPGAs have been the bastard step-brother of ASICs.

André Dehon, FPGA 2004

This chapter presents background material on FPGAs and their technological trends over the past decade. Section 2.1 covers the basic anatomy of the FPGA, beginning from fabric architecture down to the embedded memories in today's commercial devices. Section 2.2 explains the merits of FPGAs for computing. Section 2.3 discusses state-of-the-art FPGA-based computing systems. Section 2.4 concludes with implications and guiding design principles for the CoRAM memory architecture.

2.1 FPGA Anatomy

Field Programmable Gate Arrays (FPGA) are silicon devices that allow "in-the-field" reconfiguration of programmable logic after manufacture. Modern FPGAs consist of up to millions of small, programmable lookup tables (LUTs) and interconnect wires that can be configured at the bit- and wire-level to implement arbitrary logic functions [74]. Figure 4 illustrates the basic anatomy of FPGA "soft fabric", which comprises a sea of regularly tiled configurable logic blocks (CLBs) surrounded by programmable routing [112]. Typically, each CLB contains one or more lookup tables (LUTs), which are small n-deep SRAMs that can be "configured" with specific values to implement a desired 2^n-to-1 truth table. CLBs additionally include hardwired elements such as flip-flops

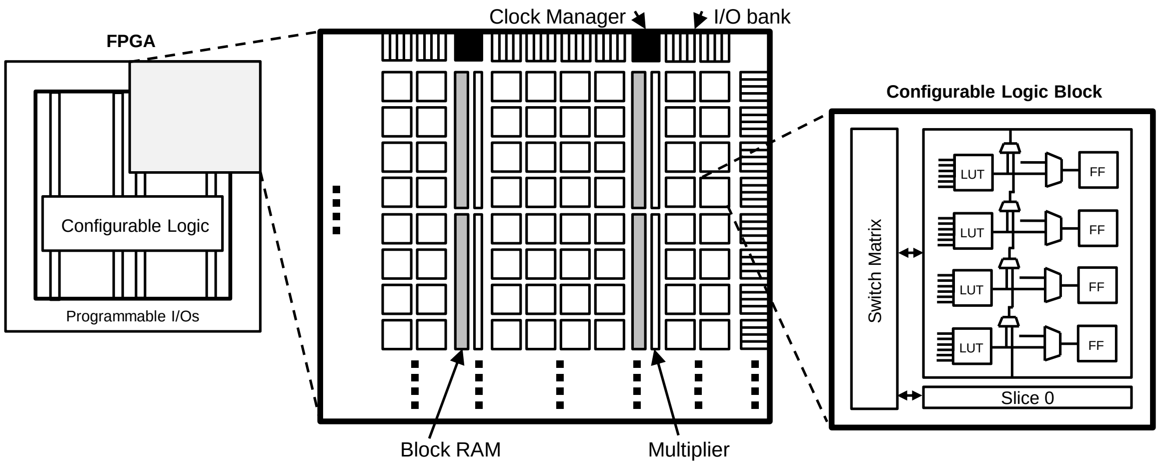


Figure 4: Conventional FPGA Fabric with Configurable Logic Blocks [114].

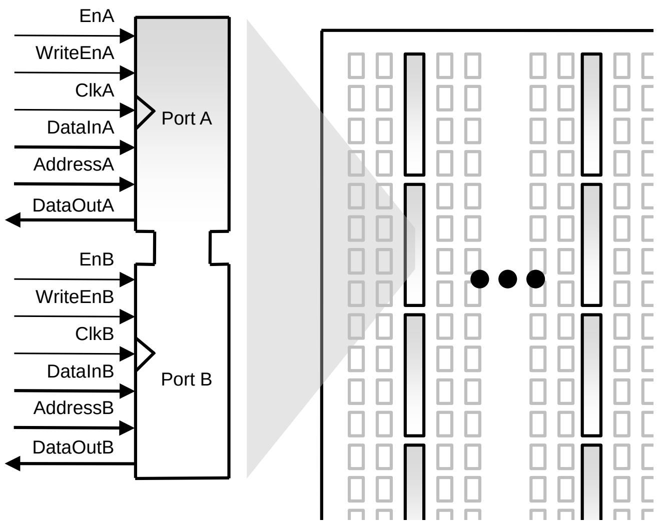


Figure 5: Conventional Block RAM Interface.

to form sequential logic and buffering. In between the CLBs shown in Figure 4, a sea of programmable interconnect allows neighboring logic to communicate and thus form the composition of multiple gates. The architecture of soft logic is typically characterized by a bevy of parameters, including the granularity of the CLB, the LUT size, the input and output connectivity to the CLB, the switch block flexibility, the number of routing tracks per channel, etc. [26][1].

Reconfigurable logic alone typically does not suffice for many FPGA-based applications. Modern commercial FPGAs usually embed up to several megabytes of on-chip memory in the form of

[1]Soft logic architecture will not be a major focus of this work. The Xilinx Virtex-6 FPGA architecture will largely serve as a baseline fabric for this thesis.

many small embedded SRAMs distributed throughout the reconfigurable fabric (e.g., Xilinx uses 18Kbit SRAMs called BlockRAMs or BRAMs [115]). These embedded memories provide a basic wire-level SRAM interface for reading and writing by gate-level reconfigurable logic (see Figure 5) and can typically operate up to 500MHz [115]. Unlike conventional SRAMs, the memories can be efficiently configured with varying aspect ratios (e.g., 16384x1 or 4096x32) or combined to emulate larger blocks of memory. The architecture of the FPGA itself usually includes hard mechanisms that avoid the use of LUT resources up to a certain composition size of multiple SRAMs [122].

The large number of distributed SRAMs plays an essential role in providing FPGA-based applications with tremendous on-chip memory bandwidth (up to terabytes per second) in order to sustain the high rate of consumption that fabric-based applications require. SRAMs are typically populated prior to configuration of the FPGA by embedding the required data within the FPGA bitstream itself. Another alternative, although costly in terms of SRAM port and LUT usage, is to dynamically populate the SRAMs with data from off-chip interfaces during the runtime of an application.

FPGA Technological Trends. The technological trends of FPGAs have closely mirrored the trajectory of Moore's Law since their inception. Figure 6 plots the characteristics of all Xilinx FPGA devices since 2002. As shown, FPGAs have continued to double in LUT area density every 18 to 24 months—progressing from tens of thousands of LUTs up to millions on a single device[2]. To visualize this level of scale, one million LUTs would be sufficient to synthesize over 800 minimally-configured soft microblaze processors in a single device [3]. The scaling of fabric is also accompanied by a commensurate increase in the number of SRAMs and multipliers as can be seen in Figure 6. The largest FPGAs today provide enough on-chip memory (tens of megabytes) to rival the capacity that of the last-level caches in today's state-of-the-art multicores. The scaling of FPGA fabric has also been accompanied by an unprecedented increase in external I/O bandwidth. Xilinx, for example, now manufactures FPGAs with high-speed transceivers up to 20Gbps per pin [116]. From a memory bandwidth perspective, high-end FPGAs would be able to provide up to 175GB/sec of off-chip memory bandwidth by the year 2012.

[2]Note that the slight anomaly in the LUT area increase from 2004 to 2007 occurred when Xilinx transitioned from a 4-input LUT architecture to a larger 6-input LUT.

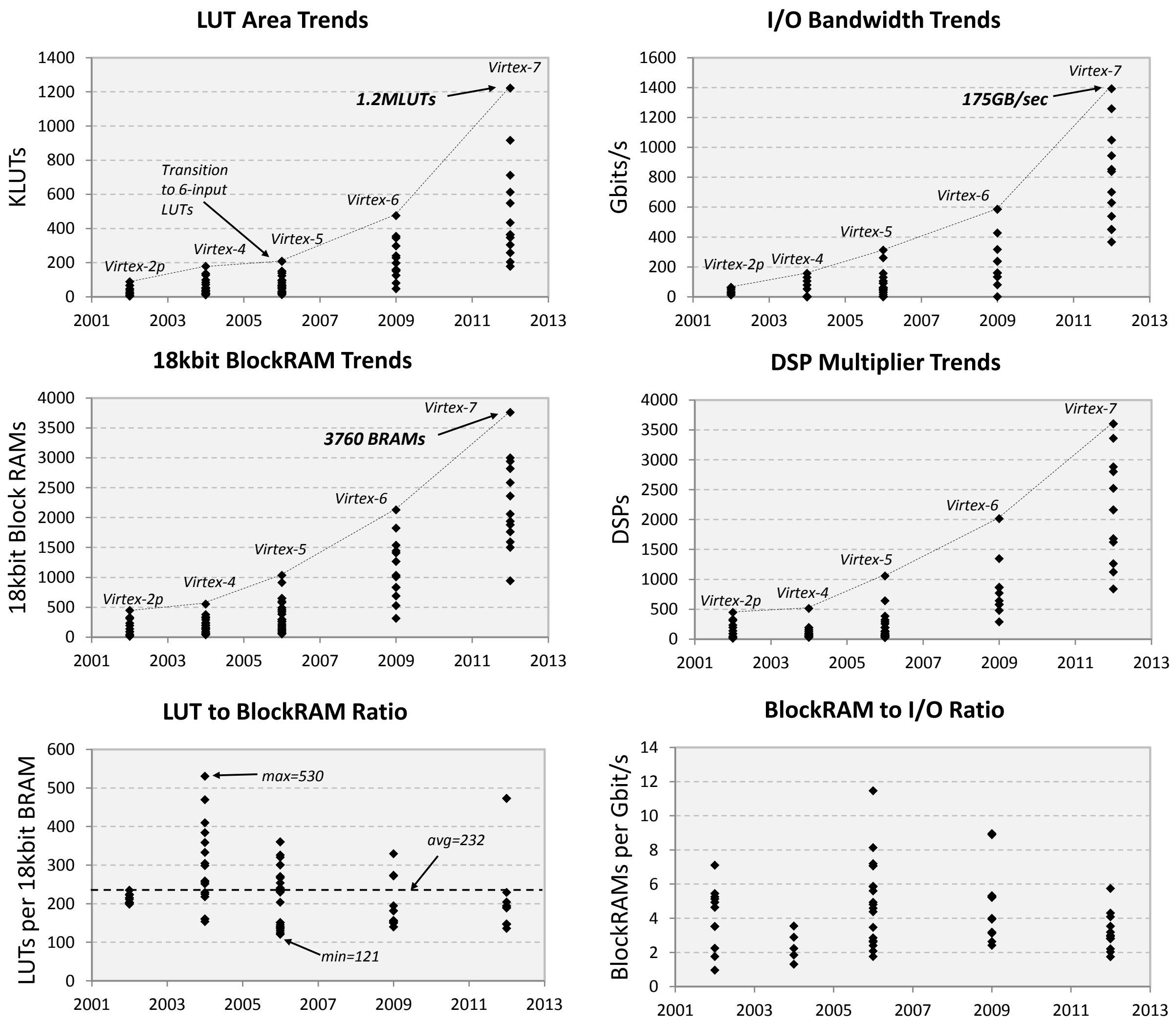


Figure 6: Xilinx FPGA Technology Trends.

2.2 Why Compute With FPGAs?

Since 2005, processor designers have shifted their focus towards increasing core counts to achieve performance commensurate with Moore's Law. Moore's Law, which has been a fundamental driver for technological innovations in the industry, projects that the number of components in a single device will double every 18 to 24 months. The recent departure from classical scaling laws [37] has placed Moore's Law in jeopardy, and thus the expected scalability of future multicore systems. Figure 7 shows the long-term expected trends in pin count, Vdd, and gate capacitance according to the ITRS 2009 roadmap [57]. Although transistor densities are projected to double with each major technology node, supply voltages are only expected to decrease by a very small amount

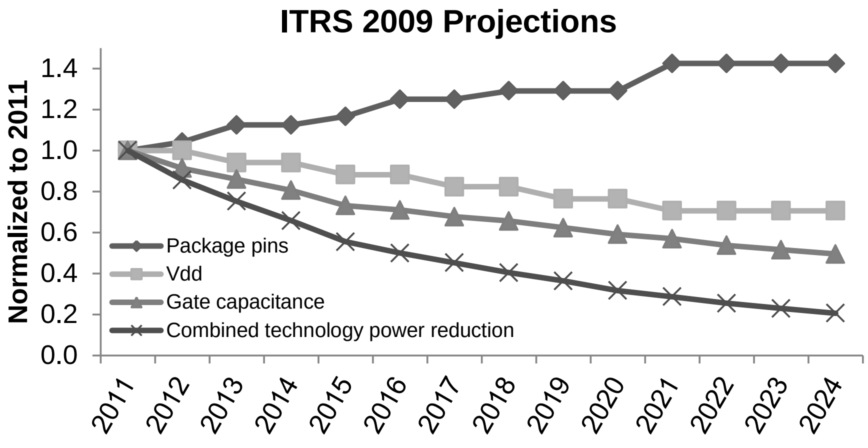


Figure 7: ITRS 2009 Scaling Projections
(High-performance MPUs and ASICs [57]).

due to leakage concerns. With the inability to reduce the threshold voltage, supply voltages must also be held high enough to maintain sufficient overdrive. Assuming that clock frequencies do not increase substantially, the reduction in power per transistor is expected to drop only by a factor of 5X over the next fifteen years (in contrast to the doubling of transistor density with each additional technology node).

Unconventional Architectures. The use of "unconventional" computing architectures such as FPGAs, GPGPUs, or even ASICs today offers a promising path in the quest for energy-efficient computing. For computing applications, the customizable and fine-grained nature of FPGAs enables them to achieve significant improvements in absolute performance and energy efficiency relative to conventional microprocessors (e.g., [43, 23, 21, 96, 63]). For example, FPGAs have been recently used to accelerate a wide range of non-traditional applications, ranging from quantitative finance [98], databases [53], bioinformatics [118], speech recognition [71], etc.

In a recent study from [23], Table 1 gives a comparison of physically-measured performance and energy efficiency between state-of-the-art multicores, GPUs, FPGAs, and Custom Logic for single-precision Black-Scholes, Matrix Matrix Multiplication and Fast Fourier Transform. As shown, the FPGA architecture offers competitive performance and energy efficiency gains over other computing devices, even relative to GPGPUs with dedicated floating point units. The recent spate of evidence supporting FPGAs, GPGPUs, and ASICs suggests that future multicore devices will transition into becoming heterogeneous, where general-purpose cores will be combined with specialized

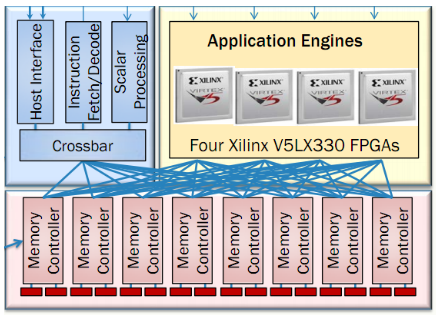
Table 1: Comparison Between FPGAs, GPGPUs, CPUs, and Custom Logic.

		GFLOP/s	*(GFLOP/s)/mm^2 (40nm)*	*GFLOP/J (40nm)*
Matrix-matrix multiplication	Intel Core i7-960	96	0.50	1.14
	Nvidia GTX285	425	2.40	6.78
	Nvidia GTX480	541	1.28	3.52
	ATI R5870	1491	5.95	9.87
	Virtex-6 LX760	204	0.53	3.62
	65nm standard cell	694	19.28	50.73
Fast Fourier Transform (N=1024)	Intel Core i7-960	67	0.35	0.71
	Nvidia GTX285	250	1.41	4.2
	Nvidia GTX480	453	1.08	4.3
	ATI R5870	-	-	-
	Virtex-6 LX760	380	0.99	6.5
	65nm standard cell	952	239	90
		MOptions/s	*(MOptions/s)/mm^2*	*MOptions/J*
Black-Scholes	Intel Core i7-960	487	2.52	4.88
	Nvidia GTX285	10756	60.72	189
	Nvidia GTX480	-	-	-
	ATI R5870	-	-	-
	Virtex-6 LX760	7800	20.26	138
	65nm standard cell	25532	1719	642.5

hardware suited for specific tasks [23, 101]. Recent commercial products that tightly couple GPUs and FPGAs with high-performance processors appear to support this trend [13, 55].

2.3 FPGA-Based Computing

The impressive raw capabilities of FPGAs have led to numerous projects and commercial efforts to develop computing platforms that incorporate FPGAs. Historically, FPGA-based reconfigurable computing systems existed either as a stand-alone board (e.g., [20]) or as a peripheral card on a computer system's low-performance I/O bus (e.g, [110]). To address bandwidth and latency concerns, newer commercial systems have begun to place FPGAs on the primary memory bus of multiprocessor systems. In particular, Cray [29] and SGI [90] first marketed computing systems that allowed a mixed population of FPGA-based processing modules and standard microprocessors on the shared-memory interconnect. Today, FPGA-based processing modules that plug into standard sockets of multiprocessor PCs or servers are widely available from SRC [85], DRC [40], XtremeData [117], and Nallatech [80, 72]. Many of the recent systems that integrate FPGAs onto the memory bus employ soft memory controllers that are implemented directly within the reconfigurable soft logic (e.g., BEE3 [32], Xilinx MIG [111]). Many of these soft implementations operate much slower



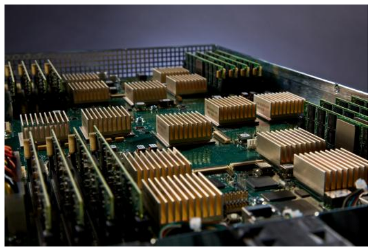


Figure 8: The Convey HC-1 Architecture (from `www.convey.com`).

than comparable hardwired implementations [28] (for example, the Virtex-6 LX240T FPGA on an ML605 platform [110] can only operate the lowest-rated DDR3 memory DIMM at 400MHz compared to the nominal clock speeds of 800MHz and above achieved by standard processors). For this reason, FPGA companies have begun integrating hardwired memory controllers that can operate at high speeds in future announced products [9, 116].

In the next subsections, we describe several existing systems that represent the state-of-the-art in FPGA-based computing. The CoRAM architecture proposed in this thesis assumes and builds upon external memory subsystems that closely resembles the commercial designs presented below.

2.3.1 The Convey HC-1

The state-of-the-art in FPGA-based computing is best exemplified by the recent Convey HC-1 Computer [27], which marries several large FPGAs to a high-bandwidth memory subsystem. Figure 8 illustrates the system-level architecture of the HC-1, which couples four large Virtex-5 LX330 FPGAs to the front-side bus of an Intel-based processor system. A unique feature of the HC-1 is that each memory controller guarantees full coherency with the caches of the host processor[3]. This feature simplifies many aspects of application development, including eliminating the need to manually transfer data between the host processor and the FPGA. The Convey memory controllers further simplify this task by implementing translation lookaside tables (TLBs) that allow the FPGAs to access the same virtual memory address space shared by the host processor.

From the perspective of fabric, each individual FPGA sustains a peak 20GB/sec of memory bandwidth as long as the traffic is distributed evenly across eight independent off-chip memory

[3]The memory controllers of the Convey are also implemented using smaller nearby FPGAs, although they cannot be re-configured by the user.

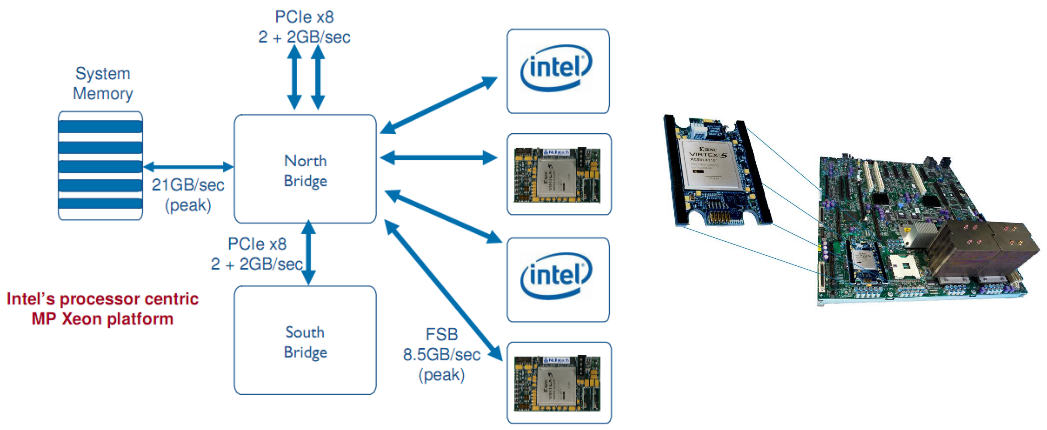


Figure 9: The Nallatech Xeon Accelerator Module (from `www.nallatech.com`).

controllers (for a combined total of 80GB/sec across four FPGAs). From the fabric's point of view, the HC-1 exposes 16 64b-wide load-store interfaces to partitioned addresses spaces within each FPGA. Each interface operates at a clock speed of 156 MHz, enabling 1.25GB/s of bandwidth per port. To achieve the maximum throughput of 20GB/sec per FPGA, applications must distribute their accesses sequentially and uniformly across the 16 load-store interfaces.

2.3.2 Nallatech Accelerator Module

The Nallatech Xeon Accelerator Module developed by Intel and Xilinx is another example of a hybrid CPU-FPGA platform that allows mixing and matching of processors and FPGAs in a multi-socket backplane (see Figure 9). The Nallatech in-socket accelerator module allows adding a Virtex-5 LX330 FPGA onto the front-side-bus (FSB) of a conventional Intel-based multiprocessor system. A unique feature of the Nallatech is the ability to stack multiple FPGAs in a single processor socket with a lower footprint. The Nallatech also allows attachment of external SRAMs to individual FPGAs to extend the on-die memory capacity. In a Nallatech system, FPGAs are treated as slave devices that receive their tasks and computation state through the host Intel processor. From the perspective of fabric, each stand-alone adapter hosting one or more FPGAs can sustain a peak 8GB/sec of bandwidth to memory, which is limited by the 64-bit 1066MHz FSB interface. A round-trip read access to memory is approximately 700ns [72].

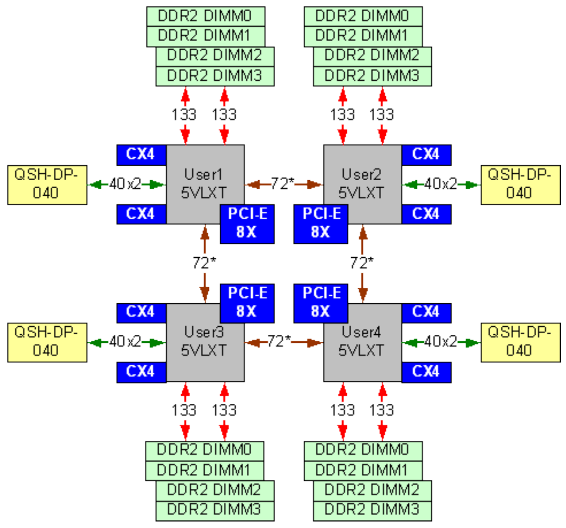


Figure 10: The BEE3 Platform (from [32]).

2.3.3 Berkeley Emulation Engine 3 (BEE3)

The Berkeley Emulation Engine (BEE3) [32] exemplifies another class of FPGA-based computing platforms, which comprise multiple Virtex-5 LX155T FPGAs (or footprint-compatible parts) connected using high-speed links in a stand-alone board. In the BEE3, four FPGAs are arranged in a ring topology with high-speed links (8.0GB/s) between them. Host-to-FPGA or FPGA-to-FPGA communication is facilitated through multiple PCI-E 8X links.

A feature unique to the BEE3 is the distributed DRAMs local to each FPGA. From the perspective of fabric, each individual FPGA holds up to four private DDR2 DIMMs managed by two dual-channel memory controllers. Each DDR2 memory controller operates at 500MT/s, amounting to a total of 16GB/sec of off-chip memory bandwidth per FPGA [32].

2.4 The Need For A Standard Memory Architecture

Despite the impressive raw capabilities of general-purpose systems, the bifurcation of multiple platforms and devices creates a serious challenge for application writers. From the application writer's perspective, an application written in mind for one platform easily becomes obsolete once the system becomes out of fashion or if a platform change is desired. Applications are frequently either discarded or re-ported to newer systems but at the cost of substantial development effort and time. If we closely examine the systems above, a root problem can be traced to the fact that applications perceive and access memory very differently across multiple platforms. The Convey HC-1 exposes a coherent global memory system through 16 local interfaces per FPGA; the BEE3 distributes its memory across multiple devices, with each device hosting two memory controllers; the Nallatech relies on the host processor to deliver the application state and memory.

In devising a standard memory architecture for FPGA-based computing, the objective of CoRAM is to abstract away the details of lower-level memory management and to enable portability and scalability of applications. To achieve portability and scalability, we devise several guiding design principle for CoRAM:

- An application should not be aware or coded to a specific protocol of a given low-level memory interface specific to a platform.
- An application should not be aware of the number of memory controllers or memory ports specific to a platform.
- Applications using a general-purpose abstraction should scale with ease and without modification to the core logic.

As will be introduced in Chapter 3, the CoRAM paradigm is designed to hide the underlying interface details by utilizing the embedded memories common in any FPGA as a logical portal into external memory (called the embedded CoRAM). Chapter 6 will later describe a style of microarchitecture for CoRAM devised in mind to support scaling up to thousands of CoRAMs per FPGA.

Chapter 3

CoRAM Architecture

Why would you want more than one machine language?

John von Neumann

This chapter motivates and introduces the Connected RAM (CoRAM) architecture, which embodies a paradigm for how users perceive and manage the computational and memory resources of an FPGA-based computing device. Much like how conventional general-purpose ISAs abstract away the lower-level details of memory hierarchy through standard loads and stores, the CoRAM architecture provides a highly distributed programming interface for managing the on- and off-chip memory resources of an FPGA. This chapter begins first by describing how CoRAM-based FPGAs "fit" into the computational stack of conventional systems. Following a discussion of key assumptions, the architectural ideas behind CoRAM are explained and justified.

3.1 CoRAM System Organization and Assumptions

The CoRAM architecture assumes the co-existence of FPGA-based computing devices alongside general-purpose processors in the context of a shared memory multiprocessor system (see Figure 11). The CoRAM architecture assumes that reconfigurable logic resources will exist either as discrete FPGA devices on a multiprocessor memory interconnect or integrated as fabric into a single-chip heterogeneous multicore. In this context, the FPGA-based components operate as peer computing devices that can independently fetch from and store to main memory.

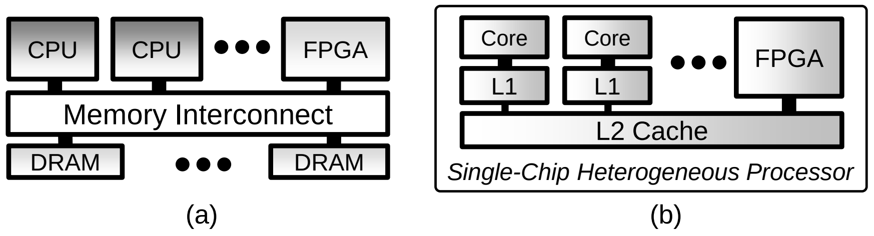


Figure 11: Assumed System Context.

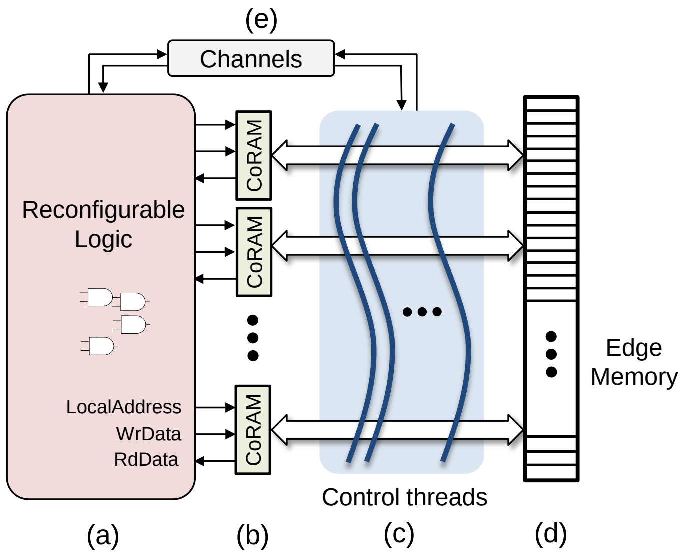


Figure 12: CoRAM Memory Architecture.

Regardless of the configuration, it is assumed that memory interfaces for loading from and storing to a global linear address space will exist at the boundaries of the reconfigurable logic (referred as edge memory in this thesis). These implementation-specific edge memory interfaces could be realized as dedicated memory/bus controllers or even coherent cache interfaces to a shared on-die memory hierarchy. Like commercial systems available today (e.g., Convey Computer [27]), reconfigurable logic devices can directly access the same virtual address space of general purpose processors (e.g., by introducing MMUs at the boundaries of fabric). The combined integration of virtual memory and direct access to the memory bus allows applications to be efficiently and easily partitioned across general-purpose processors and FPGAs, while leveraging the unique strengths of each respective device. A nearby processor is useful for handling tasks not well-suited to FPGAs—e.g., providing the OS environment, executing irregular sequential tasks (e.g., system calls), and initializing the memory contents of an application prior to its execution on the FPGA.

3.2 The CoRAM Program Model

The CoRAM programming model fundamentally embodies three independent ideas: (1) enforced separation of concerns between computation and memory management, (2) the use of embedded SRAMs as standard interfaces for accessing on- and off-chip memory, and (3) the use of software control threads as a portable memory management interface. Figure 12 offers a conceptual view of how applications are decomposed when mapped into reconfigurable logic with CoRAM support. The core logic component shown in Figure 12a is a collection of LUT and sequential resources (e.g., flip-flops) used to host the state and logic of the algorithmic kernels of a user application. It is important to note that the CoRAM programming model preserves the hardware-centric view for developing core logic and places no fundamental restriction on the description language used. For portability reasons, the only requirement is that core logic is never permitted to directly interface with off-chip I/O pins, access memory interfaces, or be aware of platform-specific details.

An application hosted in this environment is only allowed to interact with external memory and I/O devices through a collection of specialized, distributed embedded SRAMs called CoRAMs, as diagrammed in Figure 12b. Much like current FPGA memory architectures, the embedded CoRAMs serve the same role that of conventional FPGA SRAMs [81]—(1) they provide the application with many independent banks of on-chip storage, (2) they present a simple wire-level SRAM interface to the core logic with deterministic access times, (3) they are spatially distributed, and (4) they provide high aggregate on-chip bandwidth on the order of terabytes per second. Like traditional FPGA-based embedded SRAMs, CoRAMs can be further composed together and configured with flexible aspect ratios (e.g., 16384x1, 4096x32). CoRAMs, however, deviate drastically from conventional SRAMs in the sense that the data contents of individual CoRAMs are actively managed by finite state machines called "control threads" as shown in Figure 12c.

3.2.1 Software Control Threads

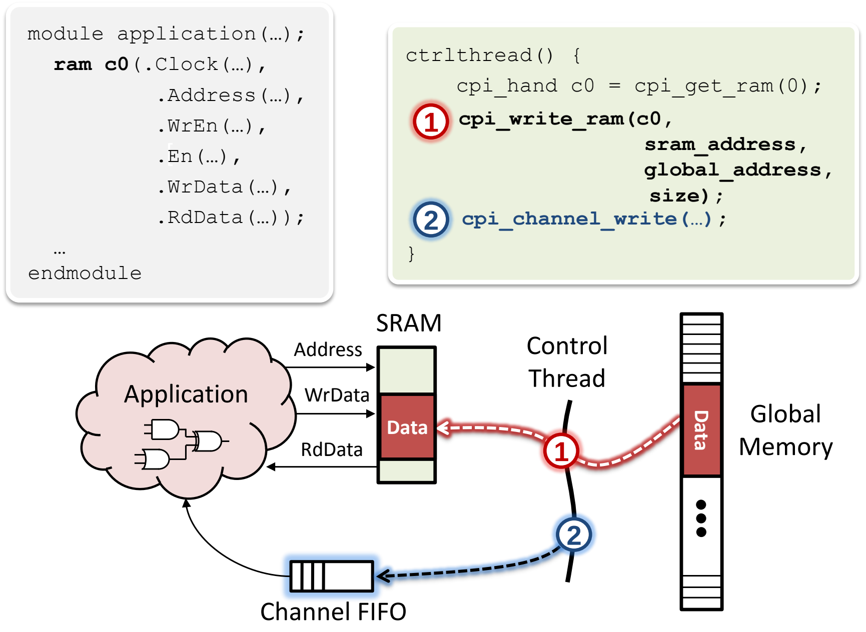
The heart of the CoRAM memory architecture is the **software control thread**. Software control threads form a fabric-distributed collection of logical, asynchronous finite state machines for managing and mediating the data transfers between embedded CoRAMs instantiated by an application and the edge memory interfaces. At a high level, control threads can be viewed as an abstract, general-purpose mechanism for prefetching an application's required data from the edge memory interface to the fabric-distributed CoRAMs. At the lowest level, control thread programs describe an ordered sequence of memory commands directed by control flow. Under the CoRAM architecture, the application writer relies solely on software control threads to access external main memory and I/O over the course of computation. Each CoRAM is managed by at most a single control thread, although a single control thread could manage multiple CoRAMs.

Control threads and the core logic of an application are peer entities that interact over low-latency, bidirectional channels (see Figure 12e). In the CoRAM architecture, channels are built out of FIFOs and registers that allow control threads and core logic to exchange information when necessary. A control thread maintains local state to facilitate its sequencing activities and issues transfer commands to the edge memory interface on behalf of the application; upon completion, the control thread informs the core logic by channels when data within the CoRAMs are ready to be accessed through their locally-addressed SRAM interfaces.

Control Actions. To utilize memory, an FPGA application instantiates one or more embedded CoRAMs to be used as a logical "portal" into external memory. To "program" the embedded CoRAMs, an associated software control thread invokes a predefined collection of memory and communication primitives called **control actions**. Control actions constitute a memory management interface that allows control threads to mediate accesses between main memory and the CoRAMs embedded throughout the fabric. At the most basic level, a control thread describes a sequence of control actions executed over the course of a program. In general, a control thread issues control actions along a dynamic sequence that can include cycles and conditional paths.

Example. To illustrate how control threads and control actions operate, Figure 13 shows how a user would (1) instantiate an embedded CoRAM as a Verilog black-box module within their application, and (2) program a corresponding control thread to read a single data word from edge memory into the CoRAM. The control thread program shown in Figure 13 (right) first acquires a special object called the co-handle and passes it into a *cpi_write_ram*[1] control action, which performs a 4-byte memory transfer from the edge memory address space to the CoRAM embedded blocks referred to by the co-handle. To inform the application when the data is ready to be accessed for

[1]cpi = CoRAM Programming Interface.



```
module application(…);
  ram c0(.Clock(…),
         .Address(…),
         .WrEn(…),
         .En(…),
         .WrData(…),
         .RdData(…));
  …
endmodule
```

```
ctrlthread() {
    cpi_hand c0 = cpi_get_ram(0);
 1  cpi_write_ram(c0,
                  sram_address,
                  global_address,
                  size);
 2  cpi_channel_write(…);
}
```

Figure 13: Example Usage of CoRAM.

computation, the control thread issues a token over a bidirectional channel to the core logic using the *cpi_channel_write* control action.

Discussion. As illustrated in the above example, a control thread is simply a high-level description of an application's memory access pattern. The control thread description deliberately presents a simple abstraction to the user but is flexible enough to be re-targeted to different hardware implementations. The example also shows that a complete "application" written for the CoRAM program model is defined as both the core logic component written in an HDL of choice along with the requisite control thread programs written in software. Fundamentally, CoRAM requires inter-operation with existing hardware description languages such as Verilog, which serve the complementary role of describing the computational or processing logic components of an application (for example, the design of a highly tuned floating point functional unit). The decision to allow arbitrary hardware description languages is motivated by the need to preserve a hardware-centric view of fabric to the application writer. In today's FPGAs, a diversity of approaches exist to describe FPGA-based applications in an efficient manner—ranging from high-level synthesis languages (e.g., Bluespec, C) down to hand-tuned IP libraries (e.g., Xilinx Coregen). A fundamental goal of CoRAM is to not restrict the user to a specific style of implementation when developing core logic.

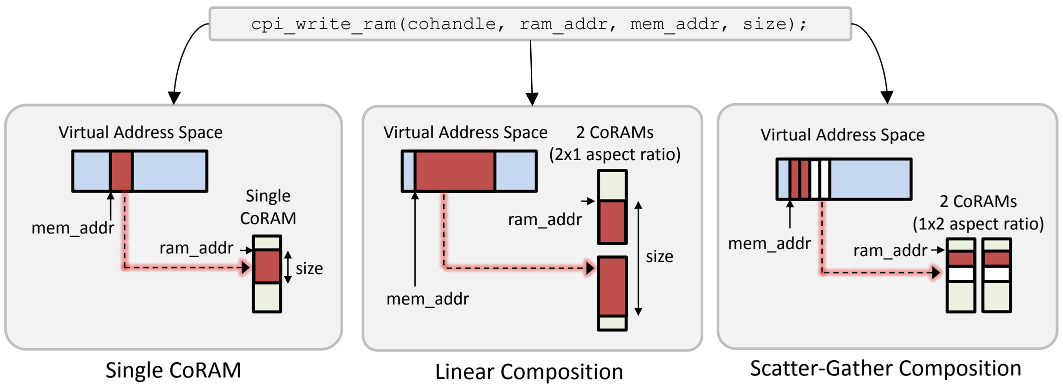


Figure 14: Memory Control Action.

Embedded CoRAM Composition. The CoRAM programming model fundamentally exposes a hardware-centric view of the FPGA that allows the user to customize data partitioning and manage the port and bandwidth usage of the embedded CoRAMs. A unique feature of CoRAM is the ability to compose multiple embedded CoRAMs to form logical portals into memory with arbitrary aspect ratios. For instance, the control action shown below is contextually interpreted based on the aspect ratio of the memories referred to by the `cohandle` object:

```
cpi_write_ram(cpi_hand     cohandle,
              cpi_ram_addr ram_addr,
              cpi_addr     mem_addr,
              int          bytes);
```

Figure 14 gives illustrated examples of how the above control action would behave differently depending on the composition style and aspect ratios of the memories. A linear composition, for example, would concatenate the local addresses of multiple embedded CoRAMs to form a deeper memory, while a scatter-gather composition would create wider memories. When executing the control action, the sequential data streaming from memory would be divided into words that match the desired composition. This feature of CoRAM gives FPGA application writers the ability to customize the interfaces of memories to their application's needs; meanwhile, the CoRAM memory management interface automatically handles the data transfer and layout of data that matches the customization.

3.2.2 How Does CoRAM Solve The Portability Challenge?

The basic challenge of portability arises from the fact that today's FPGAs require an all-to-all mapping from high-level applications down to specific devices and platforms. In the worst case, a full cross-product of implementations are needed. The root of this problem stems from the fact that conventional memory subsystems of FPGAs expose very low-level interfaces connected to external memory devices such as DRAM or a processor memory bus. Users are often required to familiarize with or even devise the protocols needed to interface with a myriad of vendor-specific specifications and devices.

Rather than forcing designers or compute vendors to target their applications and libraries to specific protocols, the software control thread paradigm of CoRAM fundamentally replaces the low-level interfaces with an **intermediate machine abstraction** that both application designers and system vendors can agree upon. The intermediate machine abstraction must be flexible and easy to use while exposing sufficient intent by the user such that efficient hardware implementations can be facilitated.

The software-based control threads serve this goal by raising the level of abstraction of an application's memory access pattern in a stylized and portable manner. Yet, at the same time, the software control threads along with the embedded CoRAM instantiations are high-level, parallelized descriptions of an application that can be efficiently re-targeted to different FPGAs and platforms. As will be shown quantitatively in Chapter 8—from a performance perspective, expressing control threads in a high level language does not become a limiting factor to performance because most time is either spent waiting for memory responses or for computation to progress. Chapter 8 will also demonstrate why CoRAM is portable by automatically retargeting applications to different hardware configurations scaled across multiple technology nodes.

3.2.3 CoRAM Microarchitecture

The dual software and HDL descriptions that constitute a CoRAM-based application must ultimately be synthesized and mapped into physical hardware. Like any "standard" architecture, a deliberate separation exists between what the application writer perceives and that of the physical mechanisms that lay beneath the abstraction. To fill in the gulf that separates a high-level con-

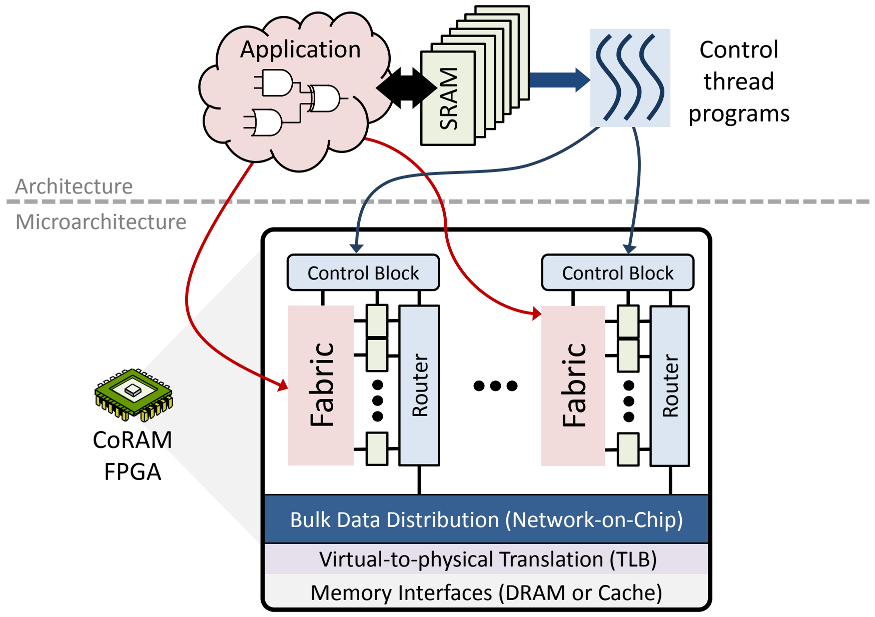


Figure 15: Beneath the CoRAM Programming Abstraction.

trol thread program and actual hardware, synthesis tools must translate the memory access patterns described by a control thread into hardware mechanisms of an independent device or platform.

Figure 31 illustrates the physical archetype of a CoRAM-based FPGA proposed in this thesis consisting of distributed islands of reconfigurable logic and embedded CoRAMs. The control threads written by an application writer are mapped down into physical control blocks co-located to various embedded CoRAMs throughout the fabric. Control blocks are responsible for interpreting and executing the control thread programs as well as generating memory requests and handling responses through a general-purpose network-on-chip (NoC). The NoC fundamentally provides connectivity and bandwidth between the CoRAM endpoints and the edge memory interfaces embedded throughout the FPGA. At the very edge of the fabric lies the memory subsystem component. Multiple memory controllers and/or bus interfaces are present to manage the data transfers between the edge of the FPGA and external storage devices such as DRAM or SRAM. Above the memory interfaces, optional last-level caches provide a bridge into the FPGA, which can be used to filter unnecessary accesses to memory. Finally, translation lookaside tables above the caches may exist in order to facilitate accesses to a virtual memory address space compliant with other peer computing devices situated on the shared memory bus.

A recurring theme echoed throughout the remainder of this thesis will be the question of **hard**

versus soft—that is, which sub-components required in the CoRAM microarchitecture merit implementation in existing FPGAs today, and what features merit "hardening" into future FPGAs designed for computing. The microarchitecture design space of Figure 31 will be explored extensively in Chapter 6.

3.3 CoRAM versus Alternative Approaches

In this section, we compare and contrast the CoRAM program model to alternative memory architecture choices. The objective of this discussion is to elaborate upon the design choices of the CoRAM architecture and to evaluate it against alternative styles that could also potentially satisfy the desired goals of portability and simplified FPGA memory management.

Standardized Memory Interfaces. A hypothetical way to standardize the memory architecture of the FPGA is to restrict all interactions to off-chip memory and I/O through uniform load-store interfaces between the edges of the fabric and the off-chip interfaces. In this case, FPGA design vendors who adopt the same standard must agree upon a particular convention and protocol for accessing the memory or bus controllers of off-chip interfaces. In this setting, the application writer is still responsible for sharing and multiplexing the off-chip memory resources among the multiple clients that require access to memory. The application writer must also implement manually within soft logic the mechanisms for transferring data between the now-standardized edge memory interfaces and the on-die SRAMs.

In-Fabric Memory Hierarchies. A natural extension of the previous approach is to distribute the standardized load-store interfaces to within the fabric itself and to provide a dedicated memory port to any client that requires memory. A distributed load-store interface requires the underlying hardware to facilitate the movement and buffering of data between external memory and the client that issued the request. The LEAP scratchpad concept [10] is a soft-logic demonstration of this approach where load-store interfaces can be instantiated on-demand by multiple clients in an application.

Behind the distributed load-store interfaces, the LEAP scratchpad concept further employs a processor-like multi-level cache hierarchy built out of embedded SRAMs within the fabric of the FPGA (with a parameterizable backing store made up of DRAM and/or on-board SRAMs). LEAP supports the data distribution automatically by instantiating an on-chip network along with parameterized levels of caching to reduce latency and bandwidth between the distributed clients and memory. The use of caching further virtualizes the capacity of on-die FPGA memory from the perspective of the application. To preserve portability of applications across different LEAP implementations, the control logic that manages the accesses of on- or off-chip memory must conform to a timing-insensitive request-response protocol [10]. The abstraction presented by LEAP could hypothetically be implemented in a future FPGA with dedicated cache controllers and arrays embedded within the fabric.

CoRAM versus In-Fabric Memory Hierarchy. The CoRAM architecture differs from an in-fabric memory hierarchy by deliberately exposing control of the on-die SRAMs in a way that preserves the hardware-centric view of FPGA memory. As in a conventional FPGA, the application writer still perceives many independent banks of memory, where each bank exposes a private address space and a single-cycle access latency. This hardware-centric view gives the application writer the liberty to perform custom data partitioning across multiple banks, implement composition of multiple SRAMs with flexible aspect ratios that match the application's requirements, and maintain guaranteed control over usage of SRAM port bandwidth and latency. An in-fabric memory hierarchy could hypothetically support similar optimizations through application-level hints to the memory subsystem; however, a demand-based memory hierarchy with automatic management of data between cache levels has a more restricted timing-insensitive interface and semantic that limits the ability of the user to precisely control data placement and port usage of the underlying on-chip SRAMs.

What CoRAM adds to the conventional SRAM interface is a standard way of simplifying and automating a frequent pattern found in FPGA-based computing, which is the movement of data between off-chip interfaces and their ultimate destinations, the on-die SRAMs. Unlike an in-fabric memory hierarchy, the CoRAM architecture resembles a "close-to-metal" ISA for the FPGA that encompasses low-level memory management primitives that can in turn be used to form higher-level services and semantics. For example, embedded CoRAMs and control threads can be used in conjunction with reconfigurable logic to provide the illusion of a timing-insensitive, cache-like interface. This style of layering deliberately decouples the FPGA architect or platform builder from having to implement higher levels of abstraction such as caches or memory streams. Instead, implementations are only required to conform to a simple set of requirements laid out by the CoRAM architecture specification, as will be discussed in detail in Chapter 4. As shown later in Chapter 8, an FPGA that directly implements the minimum set of features required by CoRAM allows applications to achieve near-ideal performance potential while incurring modest overheads in implementation cost.

CoRAM Control Threads versus Wire-Level Interfaces. A unique feature to CoRAM is the software control thread, which is an orthogonal architectural feature with respect to the previously discussed memory systems. In principle, a variant of CoRAM could allow the user to combine the control thread and core logic in a monotholic RTL application without enforcing separation of compute and memory. In this case, the user would describe control logic that issues control actions over a wire-level interface in RTL. As mentioned earlier, the rationale for using software is to avoid having the designer implement address generation using low-level RTL languages. The software abstraction allows the designer to naturally describe memory accesses as an untimed sequence of commands.

In some cases, applications may exhibit a tight coupling between the application data itself and the logic used to generate requests to memory. Algorithms such as graph traversal and sparse multiplication exhibit memory accesses that are memory data dependent. As will be shown in our examples in Chapter 5, the lightweight control threads of CoRAM communicate through high-bandwidth, low-latency channels to the core logic. This capability allows data-dependent memory accesses to be supported efficiently, as will be demonstrated in the Sparse Matrix-Vector Multiplication case study of Chapter 5.

CoRAM versus Scratchpad Memories. Beyond the systems described thus far, the style of memory management in CoRAM closely resembles the concept of software-based Scratchpad Memories (SPM) [16]. SPMs are typically implemented in specialized or power-constrained settings such as embedded systems, DSPs, and GPGPUs, which expose software-level control over the on-die memories. The control threads of CoRAM fulfill a similar role by allowing explicit and controlled data movements between global memory and the remote scratchpads (i.e., CoRAMs) distributed throughout the fabric (see Figure 16).

There are several differences between SPM and CoRAM. In CoRAM, the consumer and producer of the scratchpad is not a fixed-width processor but is instead a reconfigurable fabric that requires a different style of interface and semantics. CoRAM further separates the computation into asynchronous memory threads and reconfigurable core logic—whereas in a processor-based system, a single thread of control is often the case. In systems with SPM, memory must usually be managed carefully to avoid introducing bottlenecks in the computation. As will be shown in Chapter 5, the FPGA-based applications tend to avoid this bottleneck through the asynchronous execution of core logic and software control threads.

CoRAM versus DMA Engines. The mechanisms for CoRAM share similarities to Direct Memory Access (DMA) engines, which are typically used to manage bulk transfers between memory and devices on a shared memory or I/O bus. From a certain perspective, CoRAM could be viewed as a programmable software abstraction for a multitude of distributed DMA engines throughout the FPGA fabric. The control actions of a thread, in fact, closely resemble high-level DMA commands that ultimately become translated into low-level control signals in the memory subsystem. One difference between a raw DMA engine and CoRAM, however, is the explicit decoupling of the mechanisms from the interface. A control thread program can be interpreted in any number of ways in the implementation, making no explicit requirement on what hardware must exist beneath it.

Other Memory Architectures. Aside from CoRAM and the memory systems described in this section, other memory architectures and organizations are also valid candidates for standardization. For example, a variant on CoRAM could incorporate only a standardized, general-purpose data distribution network rather than coupling the network end-points to the on-die embedded SRAMs and the off-chip memory. In this case, the user would be given a flexible, general-purpose mechanism for data distribution throughout the fabric but would still be responsible for manually layering the memory mechanisms over the network. Both CoRAM and the in-fabric hierarchies, in particular, make the deliberate decision to couple data distribution and on-die memories due to its frequent occurrence as an application pattern. The investigation and comparison between all potential styles of memory architectures certainly merits further investigation and research. It is a contention of this thesis, however, that the CoRAM architecture is a strong candidate for investigation due to an offering of features that simplify application development, retain a hardware-centric view of on-die memory, and can be implemented efficiently in future FPGAs for computing.

3.4 Communication and Synchronization

In this section, we discuss several topics related to communication and synchronization in the CoRAM architecture.

FPGA-to-FPGA Communication. An important question that should be raised is how CoRAM as an abstraction handles multiple FPGAs or multiple disjoint fabrics on a single die. In some platforms, direct on-board links are provided between multiple FPGA nodes [32], enabling low latency and high bandwidth. The direct link approach, although beneficial in providing excellent communication bandwidth, can be detrimental to application portability because applications are explicitly partitioned and tuned to a specific platform's topology. The CoRAM abstraction deliberately makes a compromise by not allowing direct links to be exposed between multiple FPGAs; instead, CoRAM requires all communication and data transfers to propagate through global shared memory in a similar fashion between multiple processors on a shared memory bus[2]. The Convey HC-1 [27], for example, adopts a similar convention by requiring all of its four FPGAs to communicate through global coherent shared memory.

Host-to-FPGA Communication. As discussed earlier, the CoRAM abstraction assumes that all communication between general-purpose processors and FPGAs are carried out over the shared memory subsystem. Another scenario that must be considered is when the FPGA incorporates embedded hard processors in the fabric. The recently announced Xilinx Virtex-7, for example, marries the reconfigurable logic with a dual-core ARM processor [116]. The introduction of embedded hard cores with private cache hierarchies can have the unfortunate side effect of deterring portability if a given FPGA-based application assumes the features of a certain core and interacts with it through direct links. Even in this scenario, the CoRAM abstraction deliberately restricts all processor-to-logic interactions through shared memory in order to standardize the communication between any FPGA application to a processor, whether on- or off-chip. To interact with processors, an FPGA-based

[2]Note that CoRAM does not preclude FPGA-to-FPGA links as a hardware optimization; it only requires that the links cannot be exposed directly to the application.

application would instantiate one or CoRAMs and would employ control threads to communicate with general-purpose programs.

Synchronization Primitives. An important question is whether CoRAM should support native synchronization primitives in the architecture specification (e.g., read-modify-write, test-and-set). In a conventional multicore or multiprocessor system, the coherent cache subsystem is a shared resource that serves the dual role of data distribution and communication between multiple processors. When multiple readers and writers are sharing the same data, semaphores must typically be employed to guarantee atomicity and proper ordering of data accesses.

In the case of a single-chip FPGA system, the soft logic fabric presents a unique environment where point-to-point communication and data distribution can be channeled and steered through the fabric without the use of memory. In the CoRAM abstraction, control threads can further communicate directly to core logic through the Channel FIFOs, which can be layered atop fabric to form efficient message-passing primitives between multiple threads. In the case of a multi-FPGA system, where distinct FPGAs share data on the memory bus (and have no direct channels with each other), the existing blocking control actions described in Chapter 4 can be used to implement serialization or atomicity if required by an application that shares data between multiple FPGAs. The same primitives can also be used to coordinate accesses between the FPGAs and other hosts such as general-purpose processors.

3.5 Other Issues

Virtual Memory and I/O. In our discussion of CoRAM, we have yet to cover the issue of virtual memory and how FPGAs with CoRAM handle I/O devices and external interaction. It is assumed that FPGAs co-existing with conventional general-purpose processors will be able to directly access the virtual address space shared by all existing processing cores on the memory bus. As noted earlier in Section 3.1, the components at the edge of the fabric will host translation lookaside tables (TLBs) that can map global addresses to physical addresses beyond the boundaries of the FPGA fabric. The TLBs would be accompanied by small controllers that provide services such as TLB miss handling and replacement. The existence of TLBs would allow access to the I/O space of physical devices through conventional memory mapping. The design and implementation of virtual memory for FPGAs has already been demonstrated in commercial designs [27] such as the Convey HC-1 (as covered in Chapter 2) and will not be a major focus of this thesis.

Supporting Alternative Styles of Memories. It has been assumed up to this point that FPGA-based applications are exposed only to a homogeneous collection of embedded memories on the FPGA. Altera MRAMs [12], for example, are divided into block types of varying aspect ratios and capacities (20Kbit M20K versus the 640-bit MLAB). Other styles of memories include LUTRAM-based memories [7] and externally-attached SRAMs [80]. Supporting heterogeneous memories in CoRAM does not introduce new difficulties because applications are already allowed to instantiate CoRAMs of arbitrary aspect ratios. Supporting LUTRAM would require emulation of the CoRAM mechanisms by means of soft logic or introducing dedicated mechanisms into the CLB. Finally, external SRAMs can be accommodated in CoRAM in one of several ways: (1) by treating and wrapping the external SRAM as another embedded CoRAM, (2) extending the edge memory interfaces and caches with an extra level of hierarchy (which makes the SRAM accesses implicit), or (3) exposing the SRAM devices explicitly as memory-mapped I/O devices.

Operating Systems and Application Environments. In a general-purpose processor, application portability is not provided alone by simply having an ISA. The operating system, libraries, and established conventions play a crucial role in allowing forward compatibility of applications. The CoRAM architecture most similarly resembles an ISA for an FPGA and would also require a software stack of libraries and components that are built upon the virtual machine abstraction (e.g., `stdlib`). It is not difficult to imagine that libraries of control thread programs could be written to form standard environments and services that support FPGA-based applications. Chapter 5 will give concrete examples of re-usable "memory personality" libraries built out of control threads and core logic.

What CoRAM Does Not Virtualize. Although CoRAM provides a "virtual" interface to memory, it does not completely solve all of the portability challenges associated with FPGAs. For instance, a fixed application running on a particular FPGA may not necessarily scale down to another FPGA half its size—that is, CoRAM does not virtualize the LUT and on-die memory resources of a given

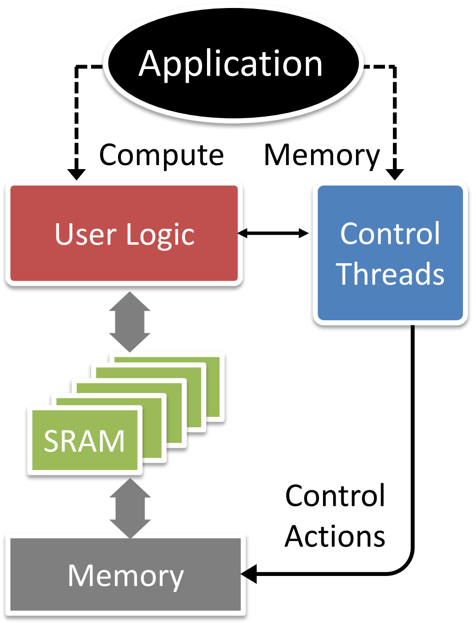


Figure 16: CoRAM as a Software-Managed Memory Hierarchy.

fabric. However, CoRAM as an abstraction does provide forward compatibility. Assuming that future FPGAs increase in density, existing applications should run without modification on subsequent devices that support the same CoRAM architectural specification.

3.6 Summary

Processing and memory are inseparable aspects of any real-world computing problems. A proper memory architecture is a critical requirement for FPGAs to succeed as a general-purpose computing technology. In this chapter, we presented a new, portable memory architecture called CoRAM to provide deliberate support for memory accesses from within the fabric of a future FPGA engineered to be a computing device. CoRAM is designed to match the memory requirements of highly concurrent and spatially distributed processing kernels that consume and produce memory data from within the fabric. The subsequent chapters of this thesis will explain the full features of CoRAM and will also demonstrate the plausibility of software-based memory management through concrete examples.

Chapter 4

Cor-C Architecture and Compiler

I have stopped reading Stephen King novels. Now I just read C code instead.

Richard O'Keefe

The Cor-C architecture specification is an *instance* of the CoRAM concept, which establishes all of the requisite details, data types, constraints, and semantics that are necessary for a real portable hardware/software interface[1]. The Cor-C architecture specification defines a dialect of the C language that can be used to express the desired behavior of control thread programs. The use of a standard, high-level language such as C affords an application developer not only simpler but also more natural expressions of control flow and memory pointer manipulations. It is important to note that Cor-C is not intended to be used as a medium for expressing the computational components of an application but rather, to be used as a lightweight memory management interface that "wraps" a given application to facilitate portability and to reduce design effort.

This chapter begins by introducing the salient features of the Cor-C language, including data types, thread invocation and management, control actions, and the semantics of memory. Section 4 will describe a prototype compiler for the Cor-C specification, which compiles control thread programs into finite state machines. Chapter 5 will later present actual uses of the prototype compiler for developing real applications using the Cor-C language.

[1]An appropriate analogy would be the MIPS ISA being an instance of the RISC concept.

Data type	Description
`bool`	1-bit boolean
`char, uchar`	8-bit signed and unsigned integers
`sint, suint`	16-bit signed and unsigned integers
`int, uint`	32-bit signed and unsigned integers
`int64, uint64`	64-bit signed and unsigned integers
`cpi_channel_ty`	An enumeration of channel object types reg, fifo
`cpi_addr`	64-bit virtual address
`cpi_ram_addr`	16-bit local ram address
`cpi_hand`	Static handle for CoRAMs and channel objects
`cpi_tag`	Transaction tag for logical memory transactions

Table 2: Cor-C Data Types

4.1 CoR-C Overview

The standard collection of primitives in Cor-C are divided into *static* versus *dynamic* control actions. Tables 3 illustrates accessor control actions that are statically processed at compile-time, while Table 4 illustrates control actions that are executed dynamically throughout the course of an application. The control actions have the appearance of a memory management API, and abstract away the details of the underlying hardware support—similar to the role served by the Instruction Set Architecture (ISA) between software and evolving hardware implementations. As will be shown later in Chapter 5, the basic set of control actions defined are powerful building blocks that can be used to compose more sophisticated memory abstractions such as scratchpads, caches, and FIFOs—each which are tailored to the memory patterns and desired interfaces of specific applications.

4.1.1 Control Threads

Every application in CoRAM begins with a source-level description of control threads to act as a "wrapper" around the core processing logic. Control threads are written in the Cor-C language, which is syntactically identical to C [64]. Table 2 summarizes the types in the language, which include several types specific to the Cor-C language. To begin an application, threads are declared using the `cpi_register` function from Table 3, which takes as argument a unique thread name and a scale factor that replicates the body of the containing function `N` times. The code below illustrates how two separate Cor-C functions would be instantiated in a single program. In this example, a total of three threads would be executed during runtime (one of threadA, two of threadB).

Control Action	Description
`cpi_register`	Registers a control thread with name `thread_name` and replicates it `N` times. `void cpi_register_thread(cpi_str thread_name, cpi_int N);`
`cpi_instance`	Returns the thread ID as an `int`. `int cpi_instance();`

`cpi_get_ram`	Returns a ram co-handle uniquely identified by `obj_id` and an optional list of sub-ids.
	`cpi_hand cpi_get_ram(int obj_id);`
	`cpi_hand cpi_get_ram(int obj_id, int sub_id);`
	`cpi_hand cpi_get_ram(int obj_id, ...);`

`cpi_get_rams`	Returns a co-handle that combines `N` rams together as a single logical memory. When `scatter` is enabled, the rams are combined in a word-interleaved fashion. If `scatter` is disabled, the rams are composed linearly. The rams selected are based on `N` consecutively numbered ids from `id...id+N-1`, where `id` is the last argument used in the control action.
	`cpi_hand cpi_get_rams(int N, bool scatter, int obj_id);`
	`cpi_hand cpi_get_rams(int N, bool scatter, int obj_id, int sub_id);`
	`cpi_hand cpi_get_rams(int N, bool scatter, int obj_id, ...);`

`cpi_get_channel`	Returns a channel co-handle based on the enumeration `ty`. The channel is uniquely identified by `obj_id` and an optional list of sub-ids.
	`cpi_hand cpi_get_channel(cpi_channel_ty ty, int obj_id);`
	`cpi_hand cpi_get_channel(cpi_channel_ty ty, ...);`

Table 3: Cor-C Accessor Control Actions (Static).

```
// single thread
void threadA() {
    cpi_register("thread-A", 1);
    ...
}

// two threads
void threadB() {
    cpi_register("thread-B", 2);
    ...
}
```

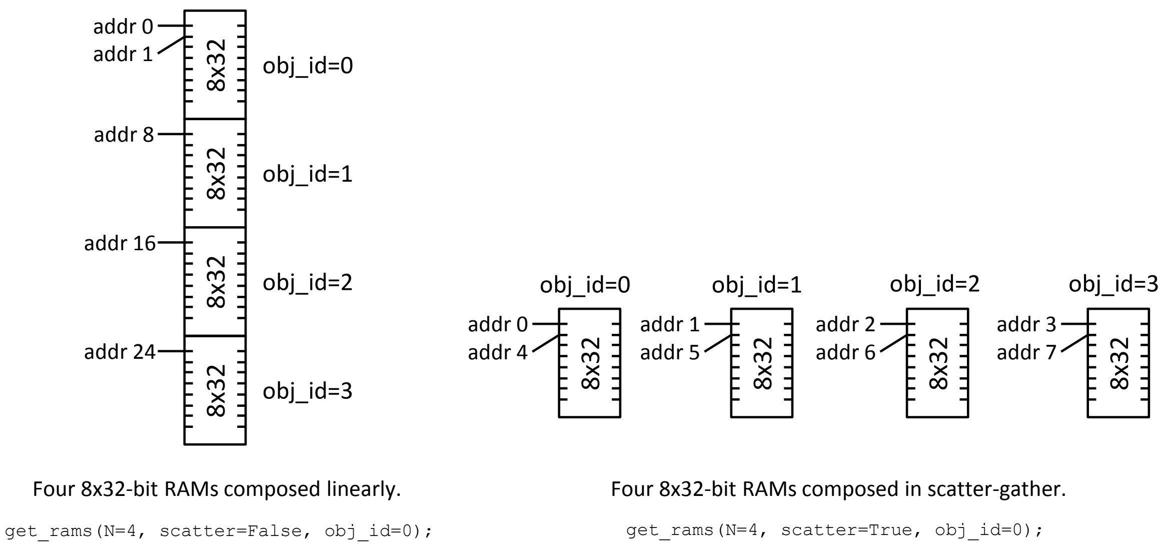
4.1.2 Object Instantiation and Identification

To utilize embedded CoRAMs, a designer begins with a pre-defined library of black-box module wrappers written in a specific hardware description language. Listing 4.1 (top) shows the Verilog port list of an embedded CoRAM with a single read-write SRAM port. Unlike a typical SRAM, the embedded CoRAM includes extra parameters specific to the Cor-C architecture specification. The `THREAD` is a string that names a particular control thread associated with the CoRAM. The additional field, `THREAD_ID` is necessary for scale factors greater than 1 (i.e., when a thread is replicated with `cpi_register`). Finally, the `OBJECT_ID` and an optional list of `SUB_ID` parameters distinguish between multiple CoRAMs managed by a single control thread instance. In addition to the CoRAMs, users may also instantiate channel FIFOs (shown in Listing 4.1, bottom) that enable the core logic to communicate with specific control threads in the application. The convention to identifying and acquiring channel objects are the same as that of acquiring CoRAMs.

When performing accesses to memory, a control thread typically gathers one or more instantiated CoRAMs into a single, program-level identifier called the **co-handle**— or `cpi_hand` for short. The co-handle establishes a compile-time binding between an individual control thread and a collection of one or more CoRAMs that are functioning as a single logical unit. Like conventional FPGAs, CoRAMs can be combined to form flexible aspect ratios and capacities. Figure 17 illustrates how multiple CoRAMs are composed to form a single RAM with deeper entries (called linear) or a single RAM with wider data words (called scatter/gather). The composition of multiple RAMs

Listing 4.1: Verilog black-box definition for single-ported embedded CoRAM and Channel FIFO.

```
module CORAM1(CLK, RST_N, en, rnw, addr, din, dout);

    parameter THREAD     = "thread_name";
    parameter THREAD_ID  = 0;   // corresponds to cpi_instance()
    parameter OBJECT_ID  = 0;   // object id
    parameter WIDTH      = 32;  // RAM data width
    parameter DEPTH      = 512; // RAM depth
    parameter INDEXWIDTH = 9;   // log of RAM depth

    // Additional optional parameters
    parameter SUB_ID     = -1;  // valid if not -1
    parameter SUBSUB_ID  = -1;
    ...

    input   CLK,RST_N, en, rnw;
    input   [INDEXWIDTH-1:0]  addr;
    input   [WIDTH-1:0] din;
    output  [WIDTH-1:0] dout;

endmodule



module ChannelFIFO(CLK, RST_N, din, din_rdy, din_en, dout, dout_rdy, dout_en);

    parameter THREAD      = "thread-name";
    parameter THREAD_ID   = 0;   // corresponds to cpi_instance()
    parameter OBJECT_ID   = 0;   // object id
    parameter WIDTH       = 64;  // channel data width
    parameter DEPTH       = 16;  // channel depth
    parameter LOGDEPTH    = 4;   // log of channel depth

    // Additional optional parameters
    parameter SUB_ID     = -1;  // valid if not -1
    ...

    input                   CLK, RST_N;
    // From user logic to control thread
    input                   dout_en;
    input [WIDTH-1:0]       dout;
    output                  dout_rdy;
    // From control thread to user logic
    input                   din_en;
    output [WIDTH-1:0]      din;
    output                  din_rdy;

endmodule
```



addr 0
addr 1
addr 8
addr 16
addr 24
8x32
obj_id=0
obj_id=1
obj_id=2
obj_id=3
Four 8x32-bit RAMs composed linearly.
get_rams(N=4, scatter=False, obj_id=0);
addr 0
addr 4
addr 1
addr 5
addr 2
addr 6
addr 3
addr 7
Four 8x32-bit RAMs composed in scatter-gather.
get_rams(N=4, scatter=True, obj_id=0);

Figure 17: Linear and Scatter-Gather RAM Compositions.

can be declared in a control thread using the `get_rams` accessor function, which returns a co-handle that represents one or more CoRAMs functioning as a single logical unit. The `get_rams` accessor takes as argument `N` number of CoRAMs, an option to compose the CoRAMs linearly or in scatter-gather mode, and the base `object_id` (plus an optional list of sub-ids) to uniquely identify a range of CoRAMs.

4.1.3 Memory Control Actions

The basic role of the control thread is to perform memory operations upon co-handles and to inform the core processing logic through channels when particular operations have completed. The most basic way to operate upon a co-handle is to pass it into a `cpi_ram_write` memory control action, which performs a logical memory transfer of `size` bytes from the global memory address `mem_addr` to the local address `ram_addr` of the CoRAMs named by co-handle. When completed, a sequential block of data from memory will be split into RAM-sized words that are written in sequence according to the arranged memory-mapping of addresses of each individual CoRAM (see Figure 17).

Blocking vs. Non-Blocking. Memory control actions are subdivided into blocking versus non-blocking behaviors (see Table 4). The CoRAM architecture presents a behavior where sequences of "blocking" control actions (`cpi_write_ram`, `cpi_read_ram`) will appear to execute atomically "one-

Control Action	Description
cpi_nb_write_ram	Performs a non-blocking transfer of N bytes from memory at address addr to the rams co-handle beginning at local address ram_addr. Returns a transaction tag cpi_tag which can be valid or invalid (CPI_INVALID_TAG). If the last argument tag_append is set to a value equal to the tag of a previous non-blocking transfer, the current transfer will be appended to the previous transaction and will share the same tag. `tag = cpi_nb_write_ram(cpi_hand rams, cpi_ram_addr ram_addr, cpi_addr addr, cpi_int N, cpi_tag tag_append);`
cpi_nb_read_ram	Same as cpi_nb_write_ram except that transfers move from rams to memory. `tag = cpi_nb_read_read(cpi_hand rams, cpi_ram_addr ram_addr, cpi_addr addr, cpi_int N, cpi_tag tag_append);`
cpi_write_ram	Same as cpi_nb_write_ram except that control threads suspend until the transaction completes. `cpi_write_ram(cpi_hand rams, cpi_ram_addr ram_addr, cpi_addr addr, cpi_int N);`
cpi_read_ram	Same as cpi_nb_read_ram except that control threads suspend until the transaction completes. `cpi_read_ram(cpi_hand rams, cpi_ram_addr ram_addr, cpi_addr addr, cpi_int N);`
cpi_test	Takes as argument a rams co-handle and tag and returns a bool indicating whether previous transactions associated with cpi_nb_read_ram or cpi_nb_write_ram have completed. `bool cpi_test(cpi_hand rams, cpi_hand tag);`
cpi_wait	Takes as argument a rams co-handle and tag and blocks the control thread until the previous transactions associated with tag have completed. `void cpi_wait(cpi_hand rams, cpi_hand tag);`
cpi_bind	Establishes a static binding between a rams co-handle and a channel co-handle, which will automatically deliver notifications on transactions applied to rams to channel. Once a cpi_bind is established, the control thread is no longer permitted to use cpi_test or cpi_wait on rams. The control thread also can no longer perform write_channel to channel.

Table 4: Memory Control Actions (Dynamic).

Control Action	Description
cpi_read_channel	Reads from channel and returns data of type cpi_int64. The control thread will block if channel is empty. `cpi_hand channel, cpi_int64 cpi_read_channel(cpi_hand);`
cpi_write_channel	Writes data to channel. The control thread will block if the channel is full. `void cpi_write_channel(cpi_hand channel, cpi_int64 data);`
cpi_test_channel	Takes as argument a channel co-handle and returns a bool indicating whether the channel is either empty or full (depends on the test input boolean option check_empty. `bool cpi_test_channel(cpi_hand channel, bool check_empty);`

Table 5: Channel Control Actions (Dynamic).

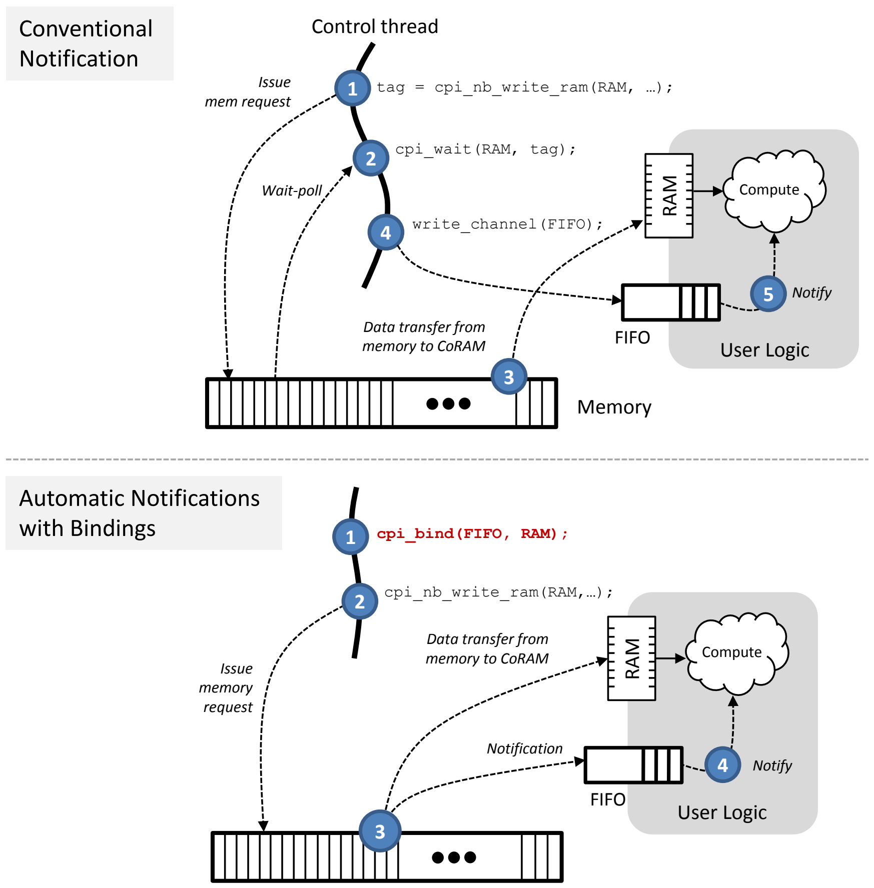


Figure 18: Supporting Automatic Notification with Channel-to-CoRAM Bindings.

at-a-time” from the perspective of a single control thread. In some circumstances, it is desirable from a performance perspective to explicitly allow multiple outstanding control actions to proceed in parallel (i.e., to pipeline multiple address requests). Non-blocking control actions support this by immediately returning control to the thread and providing a tag that must be tested later to determine when a transaction has completed (see `cpi_test` and `cpi_wait` in Table 4). Note that in some cases, the underlying hardware may return an invalid tag, which requires the control thread to retry the transaction at a later time. A tag is held indefinitely until a `cpi_test` or `cpi_wait` is called, which has the side-effect of releasing the tag when the operation returns successfully.

A common task of the control thread is to periodically inform the core logic when specific memory transactions have completed. Table 5 summarize the channel control actions that enable bidirectional communication through FIFOs and registers. A very typical synchronization pattern is shown in Figure 18(top), where (1) a control thread issues a memory control action and receives a transaction tag, (2+3) tests the transaction tag for completion, (4) writes a token to the core logic through a channel FIFO, and (5) the core logic consumes the token and processes data from the CoRAM.

Transaction Coalescing. The use of non-blocking transactions requires a control thread to track multiple outstanding tags, which can lead to overheads in tag state management and cycles consumed by periodic testing. The Cor-C architecture provides an optimization to reduce this overhead by allowing a memory control action to coalesce multiple transactions to an existing tag held by the thread. For example:

```
cpi_tag reused_tag = CPI_INVALID_TAG;
for(int i=0; i < 10; i++) {
    tag = cpi_nb_write_ram(ramA, i, i*4, 4, reused_tag);
}
cpi_wait(reused_tag);
```

In the example shown above, 10 non-blocking memory transactions are executed by the control thread and coalesced into a single tag. At the end of the loop, only a single `cpi_wait` operation is required. When passing in a re-used tag, the memory control action will merge the new transaction with the prior ones.

Automatic Notifications. Another feature of the Cor-C specification is the ability to completely eliminate the need for control threads to synchronize directly with core logic. The `cpi_bind` control action shown in Table 4 allows a control thread to establish a static binding between a CoRAM co-handle and a channel FIFO. Figure 18 illustrates the method of operation—when a control action is performed upon a specific co-handle, the associated channel FIFO will automatically enqueue a token that presents to the core logic the completion of a memory transaction. Completions are placed into the channel FIFO in the same order that transactions are issued. The `cpi_bind` operation reduces the overall latency of a round-trip memory access and also allows a control thread to pipeline non-blocking multiple memory requests without having to periodically test for completions.

Thread-to-Thread Communication. Thread-to-thread synchronization can be provided natively in the Cor-C specification for message-passing between multiple threads. In some applications, the need for synchronization arises when dependencies must be enforced between phases of computation and when there are multiple concurrent control threads. Custom forms of synchronization can also be facilitated through the use of channels. For example, to implement a fast barrier, users can instantiate channels as needed into the soft logic fabric to implement their own desired synchronization methods.

4.2 Disallowed Behaviors

Although control threads have the appearance of general-purpose software threads, there are a number of restrictions in the Cor-C specification:

- The static control actions listed in Table 3 can only be executed unconditionally (e.g., cannot be conditioned by a loop variable).
- Control threads are limited to 64 CoRAMs per co-handle[2].
- Control threads may not test invalid tags or perform control actions with invalid arguments.
- Threads may not dynamically allocate memory or instantiate global variables.
- Control threads may not dereference memory pointers directly[3].
- Control threads may not execute floating point operations.
- Function stacks are allowed but must be statically bounded.
- No recursion allowed.

Many of the language restrictions above are intended to reduce the likelihood of "abusing" control threads for computation purposes. The various restrictions also ensure that control threads are highly amenable to lightweight implementations in hardware (i.e., synthesized threads or executed on lightweight microprocessors).

[2]The architectural limit is placed here due to physical constraints imposed by the cluster-style microarchitecture presented in Chapter 6.

[3]When a control threads needs to directly access memory, a single CoRAM along with a channel FIFO can be allocated and "wrapped" together to form a simple load-store interface (see Chapter 5).

4.3 Simple Example: Vector Addition

To concretely illustrate the features of the Cor-C language, the code below gives a complete top-to-bottom example of the vector increment kernel, where a sequential array of data is read in from memory, incremented, and written back to main memory. The particular kernel in this example performs two concurrent increments per clock cycle.

```
void vector_increment_thread()
{
    cpi_register_thread("vector_add", 1/*number of threads*/);
    cpi_hand data_store = cpi_get_rams(2/*numRams*/, true/*scatter*/, 0, 0);
    cpi_hand bind_channel = cpi_get_channel(cpi_fifo, 0);
    cpi_hand done_channel = cpi_get_channel(cpi_fifo, 1);
    cpi_bind(bind_channel, data_store);
    cpi_tag tag = CPI_INVALID_TAG;

    /* Read memory */
    for(int i=0; i < 128; i+=8)
        tag = cpi_nb_write_ram(data_store, i, i*8, 8, tag);

    /* Wait for computation to finish */
    while(!cpi_read_channel(done_channel)) {}
    cpi_tag tag = CPI_INVALID_TAG;

    /* Writeback to memory */
    for(int i=0; i < 128; i+=8)
        tag = cpi_nb_read_ram(data_store, i, i*8, 8, tag);
    cpi_wait(tag);
}

module vector_kernel(CLK, RST_N);
    input CLK, RST_N;
    reg busy, writeback, done, dout_en;
    reg [5:0] addr, waddr;
    reg [31:0] din0, din1;
    wire [31:0] dout0, dout1;
```

```
CORAM2#("vector_add", 0/*thread-id*/, 0 /*obj-id*/,
        0/*sub-id*/, 32/*data width*/, 32 /*depth*/, 5/*addr-width*/)
        arr0 (.CLK(CLK), .RST_N(RST_N), .en(1'b1), .wen(wen),
              .waddr(waddr), .addr(addr), .din(din0), .dout(dout0));

CORAM2#("vector_add", 0/*thread-id*/, 0 /*obj-id*/,
        1/*sub-id*/, 32/*data width*/, 32 /*depth*/, 5/*addr-width*/)
        arr1 (.CLK(CLK), .RST_N(RST_N), .en(1'b1), .wen(wen),
              .waddr(waddr), .addr(addr), .din(din1), .dout(dout1));

ChannelFIFO cfifo(.CLK(CLK), .RST_N(RST_N),
                  .dout_en(dout_en), .dout(0), .../*unused signals*/);

always@(posedge CLK) begin
    if(RST_N) begin
        if(dout_rdy && !busy) begin
            writeback <= 0;
            busy <= 1;
            addr <= 0;
            waddr <= 0;
        end
        else if(busy) begin
            addr <= addr + 1;
            writeback <= (addr < 32);
            busy <= (addr < 32);
        end
        else writeback <= 0;

        if(writeback) begin
            wen <= 1'b1;
            din0 <= dout0+1;
            din1 <= dout1+1;
            waddr <= waddr+1;
            if(waddr == 31) dout_en <= 1;
        end
        else begin
```

```
                wen <= 1'b0;
                dout_en <= 1'b0;
            end
        end
        else begin
            writeback <= 0;
            dout_en <= 0;
            busy <= 0;
            wen <= 0;
        end
    end

endmodule
```

In the first phase of the control thread program above, the thread sets up a programmed transfer that reads in 128B of data from memory into 2 separate embedded CoRAMs represented by a single co-handle. To present a wide 64-bit word interface to the fabric, the co-handle is composed with the scatter-gather argument set to true. The `cpi_bind` operation thereafter establishes an implicit channel between the memory system and the core logic, which is the ultimate consumer of the memory data. Within the core logic, four embedded CoRAMs and a single channel FIFO are instantiated as black-box modules. As memory transactions stream through one-at-a-time, the core logic will receive tokens through the channel FIFO, indicating that data is ready for access within the CoRAMs. In the simple example above, the core logic waits until all the tokens are received before performing the accumulation steps. During the compute phase, the core logic reads and writes 32 clock cycles worth of data from the embedded CoRAMs. Upon completion, a token is written back from the core logic to the control thread indicating that a writeback to memory is pending. The control thread in the background wait-polls on the channel until receiving the token and then performs the final set of memory control actions to write data from the CoRAMs to memory.

Summary. It is not difficult to imagine that many variants of control actions could be added to the Cor-C architecture specification to support more sophisticated patterns or optimizations (e.g., broadcast from one CoRAM to many, prefetch, strided access, programmable patterns, etc.). In

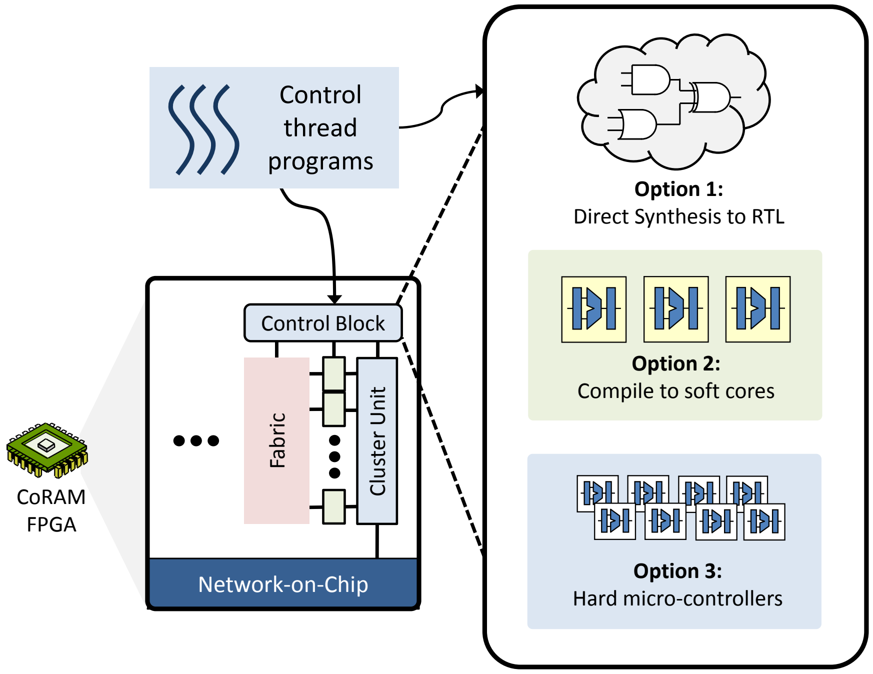
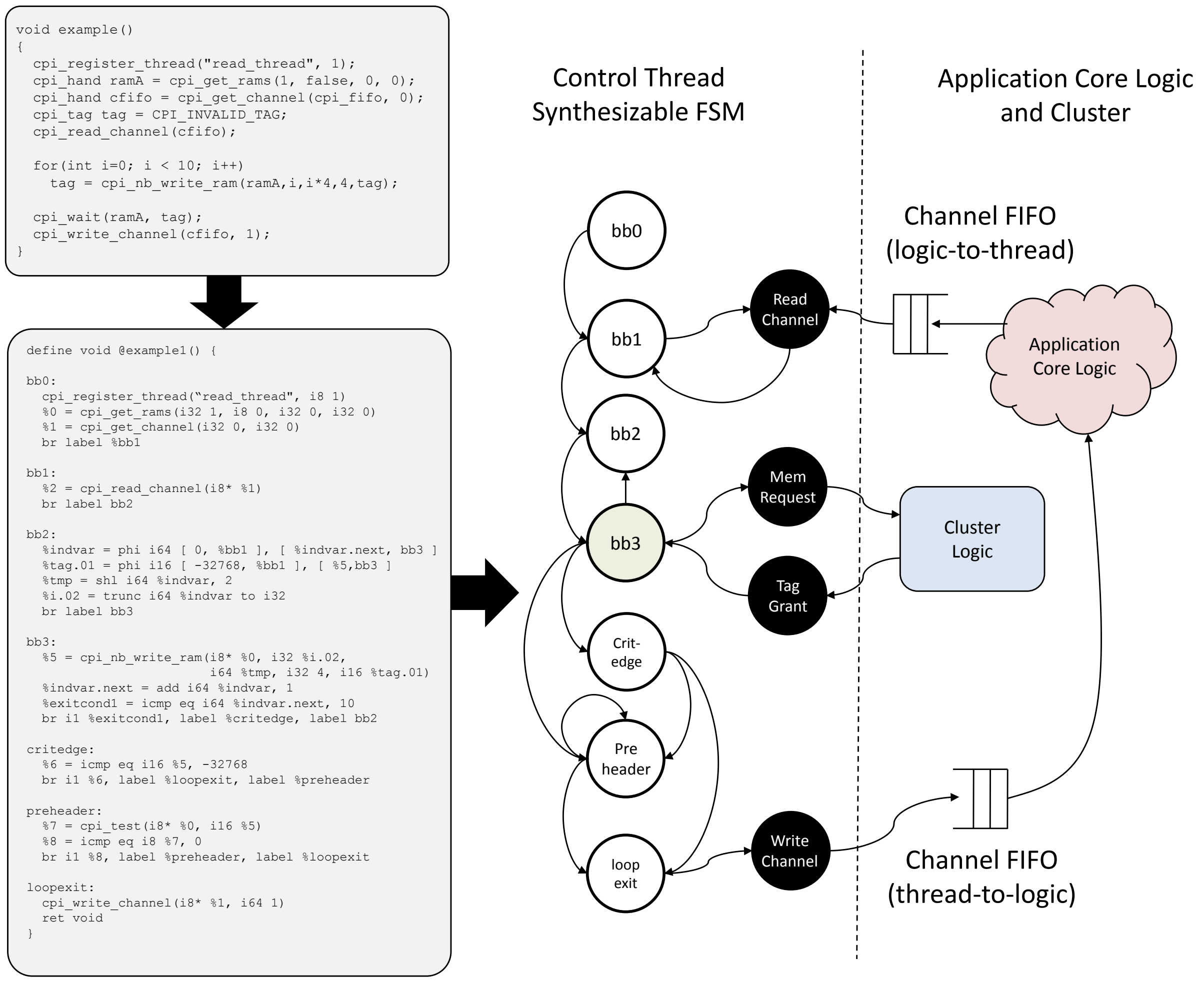


Figure 19: Options for Synthesizing Control Threads.

a commercial production setting, control actions—like instructions in an ISA—must be carefully defined and preserved to achieve the value of portability and compatibility. Optimizing compilers could also could play a significant role in static optimization of control thread programs. Analysis could be used, for example, to identify non-conflicting control actions that are logically executed in sequence but can actually be executed concurrently without affecting correctness. The next section describes a compiler and proof-of-concept of the Cor-C architecture specification. Chapter 5 will later present concrete demonstrations of Cor-C-based applications.

4.4 The CoRAM Control Compiler (CORCC)

The CoRAM Control Compiler (CORCC) was developed in this thesis to explore the various implementation options for control threads. Figure 19 shows the several ways in which a control thread program can be mapped down into control logic in an FPGA with CoRAM support: (1) directly compiling control thread programs into soft logic state machines via high-level synthesis, (2) compiling control threads to pre-implemented soft microprocessor cores (e.g., Xilinx Microblaze [112] or Altera Nios [11]) or (3) compiling to a hard microprocessor serving as a dedicated microcontroller.



```
void example()
{
  cpi_register_thread("read_thread", 1);
  cpi_hand ramA = cpi_get_rams(1, false, 0, 0);
  cpi_hand cfifo = cpi_get_channel(cpi_fifo, 0);
  cpi_tag tag = CPI_INVALID_TAG;
  cpi_read_channel(cfifo);

  for(int i=0; i < 10; i++)
    tag = cpi_nb_write_ram(ramA,i,i*4,4,tag);

  cpi_wait(ramA, tag);
  cpi_write_channel(cfifo, 1);
}
```

```
define void @example1() {

bb0:
  cpi_register_thread("read_thread", i8 1)
  %0 = cpi_get_rams(i32 1, i8 0, i32 0, i32 0)
  %1 = cpi_get_channel(i32 0, i32 0)
  br label %bb1

bb1:
  %2 = cpi_read_channel(i8* %1)
  br label bb2

bb2:
  %indvar = phi i64 [ 0, %bb1 ], [ %indvar.next, bb3 ]
  %tag.01 = phi i16 [ -32768, %bb1 ], [ %5,bb3 ]
  %tmp = shl i64 %indvar, 2
  %i.02 = trunc i64 %indvar to i32
  br label bb3

bb3:
  %5 = cpi_nb_write_ram(i8* %0, i32 %i.02,
                        i64 %tmp, i32 4, i16 %tag.01)
  %indvar.next = add i64 %indvar, 1
  %exitcond1 = icmp eq i64 %indvar.next, 10
  br i1 %exitcond1, label %critedge, label bb2

critedge:
  %6 = icmp eq i16 %5, -32768
  br i1 %6, label %loopexit, label %preheader

preheader:
  %7 = cpi_test(i8* %0, i16 %5)
  %8 = icmp eq i8 %7, 0
  br i1 %8, label %preheader, label %loopexit

loopexit:
  cpi_write_channel(i8* %1, i64 1)
  ret void
}
```

Figure 20: CORCC Example.

CORCC supports direct synthesis of control threads into synthesizable RTL from standard C code and can also be configured to model the cycle-time performance of a simple microprocessor. The implementation of CORCC leverages the Low Level Virtual Machine (LLVM) framework [69], which is an open-source, end-to-end compiler with pluggable extensions for custom passes and backends. CORCC leverages the modularity of LLVM and its language-independent intermediate representation (IR) to implement a simple form of high-level synthesis with extensions for microprocessor performance modeling.

Implementation. The CORCC LLVM extension is implemented in 6000L of C++ as a series of LLVM passes. CORCC extends LLVM with special objects and data types that are specific to the Cor-C architecture specification. These include CoRAM and channel accessors, co-handles, and the memory/channel control actions. LLVM includes front-ends for several popular languages such as C and C++ and can automatically generate an intermediate representation (IR) in Single Static Assignment (SSA) form. In SSA form, each variable in a routine is assigned exactly once, which is useful for various optimizations and simplifying the properties of variables. An important feature of the IR is the LLVM type system, which provides high-level program-level information accessible at the assembly level. The first stage of CORCC is automatically handled by LLVM, which translates the high-level control thread program into IR organized into basic blocks. The assembly employed by LLVM constitutes about 70 instructions [69], of which a subset of about 30 are supported in CORCC[4].

Thread-to-Hardware Interface. The control threads of an application, which exist either in the form of soft finite state machines or as microcontrollers can be viewed as clients that issue memory requests to the underlying memory subsystem comprising embedded CoRAMs, the network-on-chip, and the edge memory interfaces (as illustrated earlier in Figure 19). CORCC assumes that in a soft implementation of control threads, a well-defined request-response interface exists between the underlying subsystem and the control threads implemented in the fabric. The details of such interfaces are described further in Chapter 6.

Co-handle Pass. The first stage of CORCC performs a sweep through the LLVM-generated IR and identifies `call` instructions that match the function signatures of static control actions such as co-handle and channel accessors (see Table 3) that are used to establish bindings to various CoRAM-related object. In LLVM, any function with a return value is assigned to a register identifier with a unique integer. Within CORCC, the static pass creates an internal map between an identified co-handle and its corresponding destination register. When processing a co-handle, the function arguments are checked to be constant and valid values. During this pass, any dynamic control actions are annotated and linked against the detected co-handles. The link step performs a backtracing through registers in the IR to identify the specific co-handle associated with a dynamic control action.

Thread Synthesis. Once all co-handles have been identified, CORCC performs a synthesis step that translates the LLVM instructions and the Cor-C dynamic control actions into synthesizable

[4]Use of unsupported instructions result in compile-time errors in CORCC.

Verilog. The basic approach taken by CORCC is to perform a direct mapping of basic blocks into single-cycle states in a finite state machine. To handle register state, CORCC instantiates a physical register for each assigned variable in a program. To implement logic, all of the instructions in a basic block are converted into combinational statements, where the inputs to the logic are read from registers in a single clock cycle (and in the same clock cycle, the output is written to the destination registers). The SSA form of LLVM guarantees that no registers are read and written at the same time within a single basic block.

To handle dynamic control actions, special states are introduced at locations in the LLVM IR where `call` instructions are detected. For example, when a `cpi_write_ram` function call is detected, the parent basic block will be split into two states, one containing the original and the other for invocation of the control thread. The predecessor basic block will always jump into the special state first, which handles the actual issue of the control action to the memory subsystem through a request-response interface; thereafter, the FSM jumps into the original basic block while returning the value of the control action. Figure 20 gives a complete example of compiling the simple example from Chapter 3 into synthesizable hardware.

Microprocessor Performance Modeling. To explore the design space for microcontroller-based control threads in our evaluation in Chapter 8, CORCC includes an additional feature that approximates the performance characteristics of a simple in-order microprocessor core. The core is modeled with a constant CPI value (cycles per LLVM instruction) and is assumed to have specialized logic that interfaces directly to the underlying memory subsystem described in Chapter 6. The programmed CPI value sets the rate at which control threads advance through the LLVM basic blocks in order to mimic the performance characteristics of an idealized microprocessor. Chapter 8 will later present simulation-driven results that compare direct synthesis by CORCC to soft and hard microprocessor cores.

CORCC Limitations. The CORCC compiler employs a relatively simple approach to high level synthesis, which completely expands the basic blocks of an application into synthesizable hardware. The simple approach taken here can have a detrimental effect on performance and area, especially if LLVM produces large basic blocks or allocates a large number of registers. A potential way to mitigate large critical paths within a basic block are to split basic blocks where necessary, which can either be supported automatically or guided by the user. More advanced high-level synthesis techniques can also be applied—e.g., constraining and scheduling the usage of resources. As will be shown later in Chapter 8, without any optimizations, the FSMs generated by the CORCC compiler consume relatively modest area while operating at nominal FPGA clock frequencies.

Cor-C vs. Parallel Languages. Our selection of the C language is not a fundamental requirement of the CoRAM paradigm. An area that merits further research is the use of functional or parallel languages to express higher levels of parallelism within control threads. A particular consequence of using a sequential-like language of C is the serialization of requests during dynamic execution. Consider the for loop below, which generates a stream of requests to the memory subsystem:

```
for(int i=0; i < 8192; i+= BLOCK_BYTES) {
    tag = cpi_nb_write_ram(ramA, 0, 0, BLOCK_BYTES, tag);
}
```

In the example above, CORCC would not allow the multiple control actions to execute in parallel due to serialization on the coalesced tag variable. In such case, parallel constructs such as `forall` can explicitly declare that the loop body operations are independent.

4.5 Summary

This chapter presented the Cor-C architecture specification and compiler. Cor-C is a devised instance of the CoRAM concept, and provides an example of how the CoRAM concept is applied in a real-world environment. The CoRAM Control Compiler (CORCC) is a proof-of-concept that implements the Cor-C specification and is evaluated further in Chapter 8.

Chapter 5

Cor-C Examples

Programming is usually taught by examples.

Niklaus Emil Wirth, author of *Pascal*

Three applications are presented in this chapter to provide a concrete demonstration of the Cor-C language. The first example, Matrix-Matrix Multiplication, illustrates how blocking algorithms can be expressed using a centralized control thread in Cor-C. The next example, Black-Scholes, demonstrates the description of wide, concurrent memory streams using the Cor-C language. The last example, Sparse Matrix-Vector Multiplication, demonstrates simultaneous thread execution and the support for irregular, indirect references in memory. The control thread programs presented in this section are excerpts from applications compiled using the CoRAM Control Compiler (CORCC) from Chapter 4. Chapter 8 will quantify the performance and efficiency of these applications and compare them against manual approaches for the FPGA.

5.1 Matrix-Matrix Multiplication

Matrix-matrix multiplication (MMM) is of critical importance for a broad range of scientific and engineering applications and is often a starting point for demonstrating new architectures. In this section, we demonstrate how the Cor-C architecture language from Chapter 4 can be used to succinctly express the control and memory access requirements of an FPGA-based implementation of MMM.

5.1.1 Background

The standard matrix-matrix multiplication procedure is defined as:

$$C_{i,j} = \sum_{k=0}^{N-1} A_{i,k} \times B_{k,j}, 0 \leq i < M, 0 \leq j < R$$

where A, B, and C are matrices of dimensions $M \times N$, $N \times R$, and $M \times R$, respectively. In typical usage, the matrices of MMM are encoded in row-major format—i.e., the data words of each row are consecutively ordered in main memory beginning from the first row of the matrix to the last. Assuming row-major encoding, the standard C code that implements MMM is:

```
void mmm(Data *A, Data *B, Data *C)
{
    int i, j, k;
    for(i=0; i < M; i++) {
        for(j=0; j < R; j++) {
            for(k=0; k < N; k++)
                C[i][j] = C[i][j] + A[i][k] * B[k][j];
    }
}
```

The basic challenge in optimizing MMM is to avoid excessive demands in off-chip memory bandwidth when computing very large matrices that do not fit in aggregate on-chip memory. The MMM kernel exhibits high data re-use with $O(N^3)$ computations carried out over $O(N^2)$ data accesses. MMM is also easily parallelizable given that all the dot product calculations are independent. The basic approach to reducing bandwidth in MMM is to exploit data re-use through **blocking**. Blocking works by sub-dividing the large matrix calculation into smaller MMM multiplications that have a working set size that can fit within a given device's on-chip memory constraint. The following C code gives an example of blocking in square matrix-matrix multiplication, where N is the square dimension and NB is the blocking factor.

```
void mmm_kernel(Data* A, Data* B, Data* C, int N, int NB) {
    int i, j, k;
    for (j = 0; j < NB; j++)
        for (i = 0; i < NB; i++)
            for (k = 0; k < NB; k++)
        C[i * N + j] += A[i * N + k] * B[k * N + j];
}

void blocked_mmm(Data* A, Data* B, Data *C, int N, int NB)
{
    int j, i, k;
    for (j = 0; j < N; j += NB)
        for (i = 0; i < N; i += NB)
           for (k = 0; k < N; k += NB)
               mmm_kernel(&(A[i*N+k]),
                          &(B[k*N+j]),
                          &(C[i*N+j]), N, NB);
}
```

Blocking improves the arithmetic intensity of MMM by increasing the average number of floating-point operations performed for each external memory byte transferred. As shown in the code above, the blocked MMM kernel works by computing smaller $NB \times NB$ matrices within C. Assuming that NB is sized to fit within aggregate on-chip memories, the bandwidth required to perform the computation is:

$$\begin{aligned} readbytes &= 3 \times D \times NB^2 && (5.1) \\ writebytes &= D \times NB^2 && (5.2) \\ flops &= 2 \times NB^3 && (5.3) \\ bytes/flop &= 2 \times D/NB && (5.4) \end{aligned}$$

where D is the data type used in the matrix multiplication. To maintain a balanced computer system [66], the total memory bandwidth B must scale with the computational throughput C. That is:

$$\begin{aligned} gflops/sec &= C && (5.5) \\ GB/sec &= B && (5.6) \end{aligned}$$

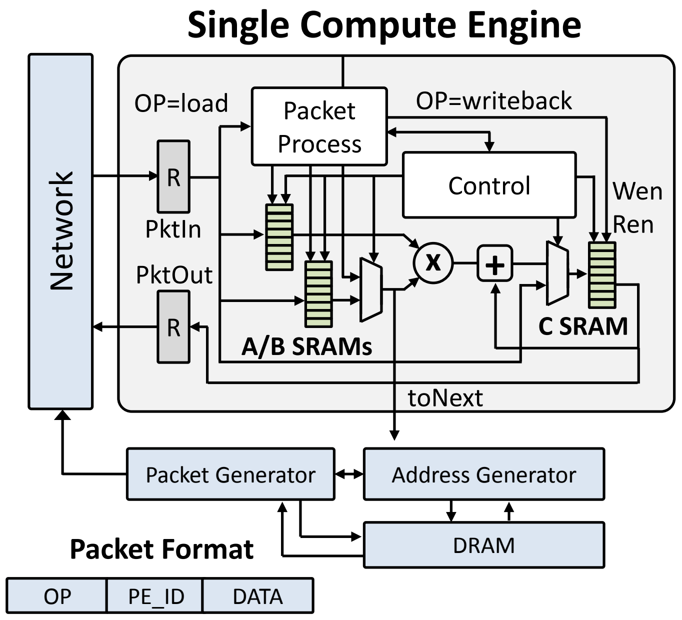


Figure 21: Matrix-Matrix Multiplication Processing Element.

$$B \;=\; C \times 2 \times D/NB \qquad (5.7)$$

In general, a baseline MMM implementation that is sped up (e.g., through parallelization) by a factor of p will increase the memory bandwidth to pB. Increasing the blocking factor NB can similarly reduce the on-chip memory bandwidth. For a given blocking factor NB, $3 \times NB \times D$ bytes of on-chip memory is needed.

5.1.2 Related Work

A significant body of work has examined blocking and parallelization of the MMM kernel for a variety of microarchitectures, ranging from multiprocessors [22, 56] to GPGPUs [82, 78] to FPGAs [39, 31, 65, 60]. The Intel Math Kernel Library, for example, provides parallelized BLAS routines optimized for microarchitectural features in Intel-based multicores [56]. The CUBLAS library from Nvidia supplies multithreaded GPGPU-optimized MMM routines [82]. A variety of floating- and fixed-point FPGA accelerators have also been demonstrated in literature [39, 31, 65, 60].

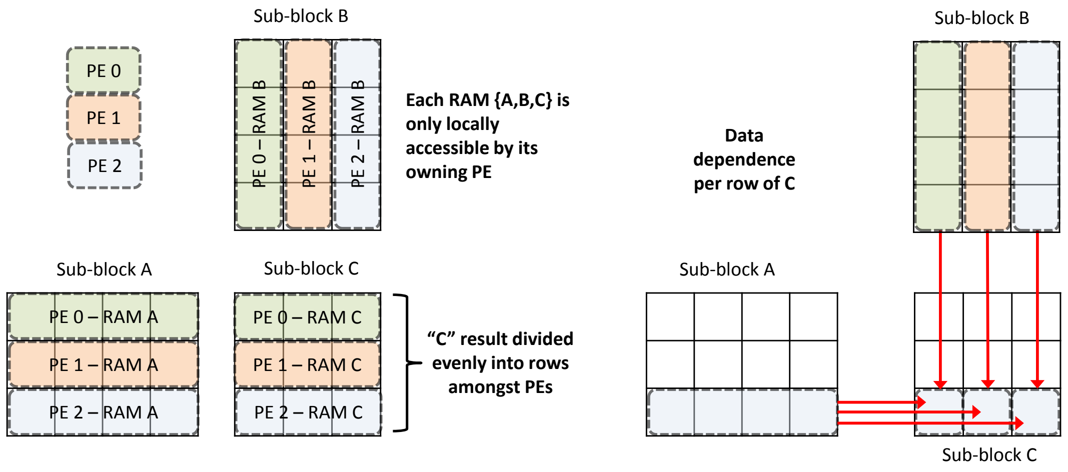


Figure 22: Work Distribution in Matrix-Matrix Multiplication.

5.1.3 Parallelization on the FPGA

A basic approach to parallelizing MMM on the FPGA is to create p identical processing elements (PE) that each perform the dot-product accumulation for a single row of matrix `C`. Each PE contains either a single- or double-precision accumulator that performs up to one multiply-addition per FPGA clock cycle. The parallelization strategy simplifies dependences by allowing each PE to compute on independent dot product accumulations. Figure 21 illustrates a parameterized hardware kernel developed for single-precision blocked MMM. The design assumes that the large input matrices `A`, `B`, and the output matrix `C` are stored in external memory in row-major. In each iteration: (1) different sub-matrices `subA`, `subB`, and `subC` are read in from external memory, and (2) `subA` and `subB` are multiplied to produce intermediate sums accumulated to sub-matrix `subC`. The sub-matrices are sized to utilize available SRAM storage on the FPGA.

Figure 22 shows how the work is distributed evenly between three PEs that compute a 3x3 `subC` matrix from 4x3 and 3x4 `subA` and `subB` inputs. The row slices of `subB` and `subC` are divided evenly among the `p` PEs and held in per-PE local buffers. The column slices of `subA` are also divided evenly and stored similarly. Figure 23 gives an example of how each phase of computation works. A complete iteration repeats the following steps `p` times: (1) each PE performs dot-products of its local slices of `subA` and `subB` to calculate intermediates sum to be accumulated into `subC`, and (2) each PE passes its local column slice of `subA` to its right neighbor cyclically. Note that as an optimization, step 2 is overlapped with step 1 in the background as illustrated in the neighbor exchanges shown

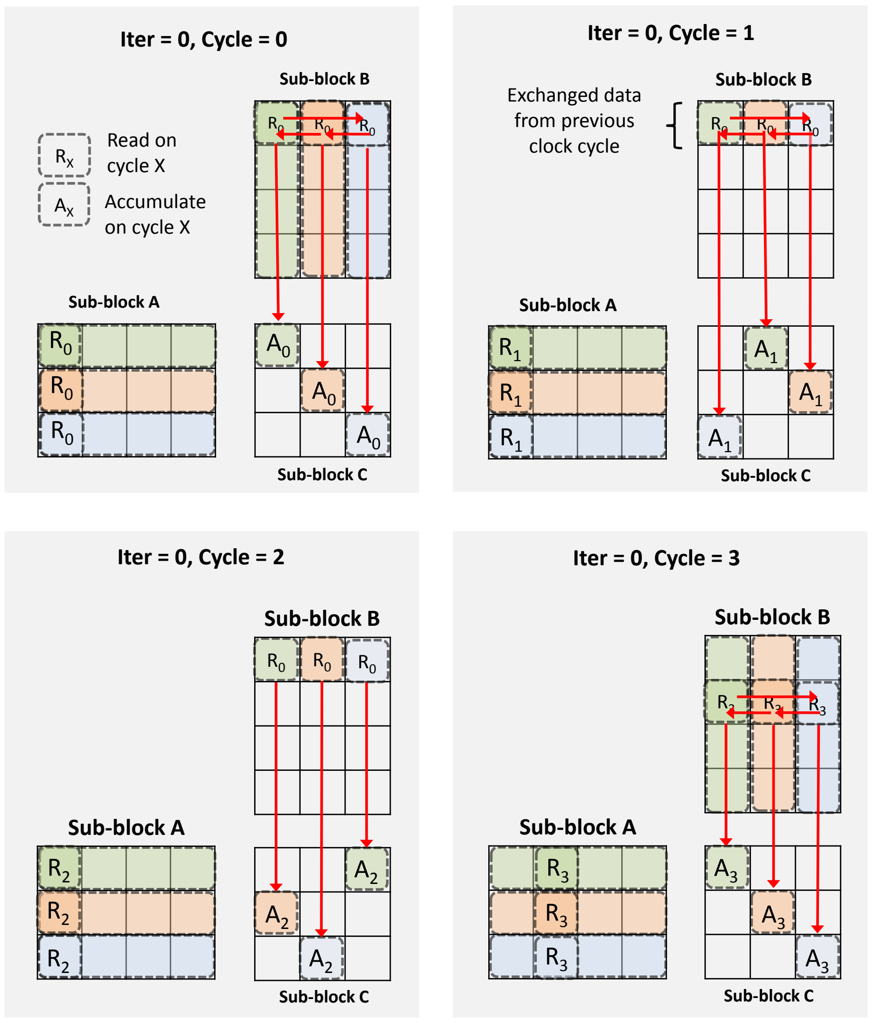


Figure 23: Computation Phase of Matrix-Matrix Multiplication.

in Figure 23.

Memory Accesses and Double-buffering. Prior to the computation phase, `subA`, `subB`, and `subC` must be read in sequentially and buffered into the local PEs' SRAM buffers. The `subC` matrix is revisited and accumulated across multiple iterations. Once the computation phase finishes, the updated `subC` must be written back to external memory, while the next set of values for `subA` and `subB` are streamed in. An important optimization for the FPGA-based implementation of MMM is to simultaneously overlap computation and memory transfers through **double-buffering**. Double-buffering effectively splits the on-chip memory into halves, where one half of the storage is used for reading in data for the next iteration, while the other half is being used for computation in the current iteration. Double-buffering reduces the blocking factor ($NB/2$) because only one half

Single PE resources	˜1.2KLUTs (area can be reduced using DSP48Es)
Clock frequency	300 MHz
Peak PE throughput	600 MFLOP/s

Table 6: Single-Precision PE Characteristics.

of the available SRAM storage is used for computation at any given moment. Figure 24 shows a timeline of double-buffered execution, with each colored box representing a particular iteration of the computation. Note how in steady-state, up to three separate iterations are simultaneously overlapped.

5.1.4 Manual Implementation

Our implementation of MMM for the FPGA was developed in two phases: (1) a stand-alone kernel that targets conventional FPGAs, and (2) a revised implementation that employs the Cor-C architecture language. Figure 21 shows a single compute engine for a conventional FPGA. The PE internally maintains a single- or double-precision accumulator surrounded by local buffers. External to the PE, a custom ring network provides connectivity between one or more PEs to an external DMA controller responsible for issuing memory accesses to the native DRAM interface. The ring network employs a custom packet format as shown in Figure 21(bottom), which allows the DMA controller to issue per-PE read and write commands to any of the local buffers. The distributed control logic of each PE coordinates the shared SRAM reads and writes between the ring network and the functional units.

The manual approach was developed in about 2000L of Bluespec System Verilog [17] over a period of 12 man-weeks. Xilinx Coregen 13.1 was used to generate the optimized floating point cores used in the accumulator [6]. Table 6 shows synthesis characteristics of a single optimized PE, which was fully placed-and-routed at 300MHz for a Virtex-6 LX760 FPGA [23]. The area consumed when the PE is configured in single-precision floating point mode is about 1.2KLUTs, although the design does include parameterized options to utilize DSP48Es slices if available on the FPGA [115].

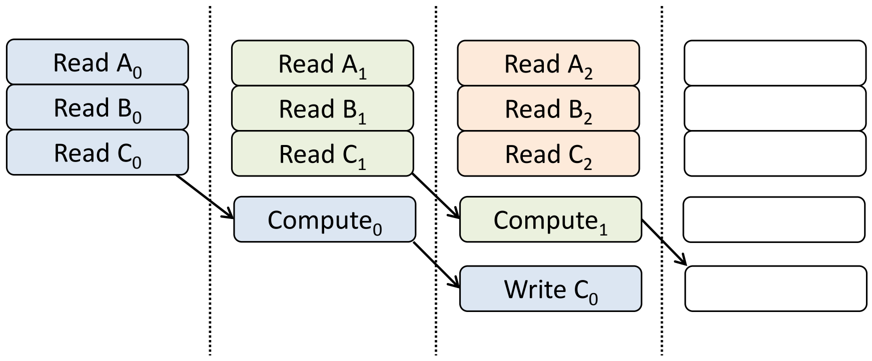
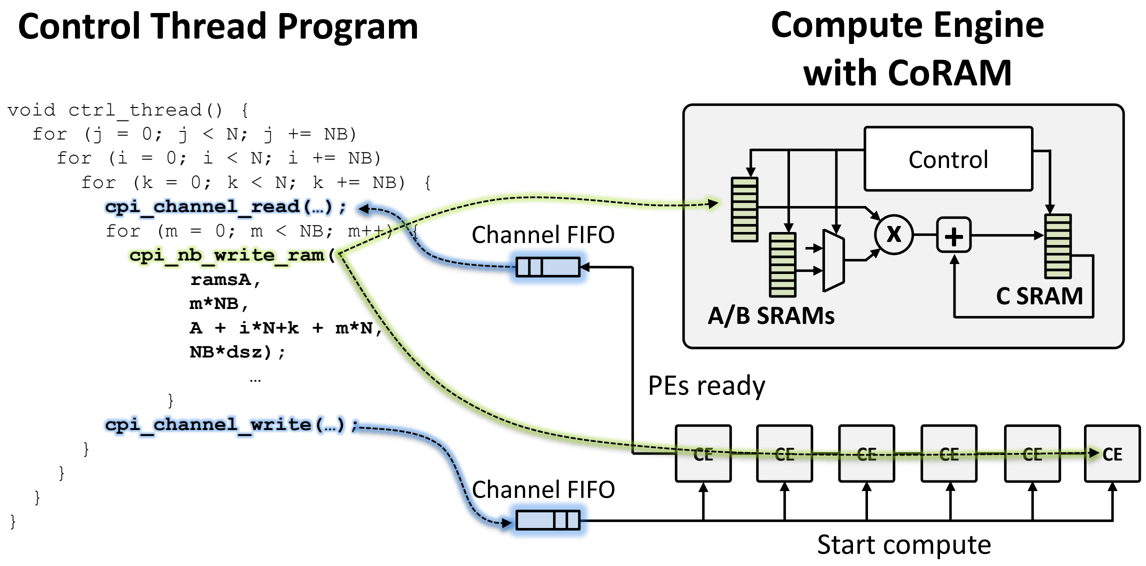


Figure 24: Matrix-Matrix Multiplication with Double Buffering.



```
void ctrl_thread() {
  for (j = 0; j < N; j += NB)
    for (i = 0; i < N; i += NB)
      for (k = 0; k < N; k += NB) {
        cpi_channel_read(…);
        for (m = 0; m < NB; m++) {
          cpi_nb_write_ram(
               ramsA,
               m*NB,
               A + i*N+k + m*N,
               NB*dsz);
                    …
             }
        cpi_channel_write(…);
      }
    }
  }
}
```

Figure 25: Matrix-Matrix Multiplication Control Thread (Non-double-buffered).

5.1.5 Cor-C Implementation

In this section, we describe how the Cor-C language was used to simplify and express the control and memory access requirements of our FPGA-based implementation of MMM. In the Cor-C version of MMM, the local buffers of each PE are replaced with black-box embedded CoRAMs as was described in Chapter 4. The Cor-C version replaces the entire data distribution network shown in Figure 21 with a centralized control thread as shown in Figure 25. As can be seen, the custom ring network, the DMA engine, as well as the native interface to DRAM are eliminated in the new design. The Cor-C version introduces a single channel FIFO that enables all the PEs to communicate with the control thread. To populate the CoRAM buffers of each PE, the centralized control thread blocks on the channel FIFO until a token is received from the core logic. Upon receiving the token, the control thread performs memory accesses to all of the necessary per-row and per-column slices of `subA`, `subB`, and `subC`.

Centralized Control Thread. The pseudo-code for the centralized control thread is shown in Figure 25. For brevity and simplicity, Figure 25 only shows the non-double-buffered implementation and omits the reading and writing of the `subC` matrix. It is worth noting how the code in Figure 25 appears similar to the C reference code for blocked MMM, with the exception that the inner-most loop now consists of memory control actions rather than computation. In the inner-most loop of Figure 25, the first action is a `cpi_channel_read`, which waits on a single token that indicates when all of the PEs are ready to load the next round of data.

In the code, the `ramsA` co-handle represents an aggregation of all the labeled ‘a’ embedded CoRAMs belonging to the PEs. The ‘a’ CoRAMs are combined as a single wide memory such that sequential data arriving from memory will be scattered and written across multiple CoRAMs in a word-by-word interleaved fashion. This is the due to the fact that data residing in external memory is encoded in row-major format, while individual CoRAMs of ‘a’ expects the sequential data from memory in column-major format. The co-handle `ramsB` expects data in a row-major format and is written to as a linear concatenation of all of the local addresses of the ‘b’ CoRAMs. Within the body of the inner loop, the control thread executes a series of `cpi_nb_write_ram` control actions to populate the embedded CoRAMs with the requisite data. Upon completion, the control thread informs the user logic when the data is ready to be accessed by writing to the bidirectional channel fifo using `cpi_channel_write`. The control thread terminates after iterating over all the blocks of matrix C.

Double-buffered Cor-C. A complete example of the double-buffered version of the control thread for MMM is shown in Listing 5.1. A single boolean variable `which` determines which phase of the buffering the control thread is operating in. Within the innermost body of the loop, the same `cpi_nb_write_ram` control actions appear as before—however, the local ram addresses are switched between lower and upper halves in each phase. The double-buffered version also splits the `C` CoRAM into two separate CoRAMs `C0` and `C1` in order to allow simultaneous reads and writes to the logical C buffer in a single phase. In early development phases, we found that performing the read and writes to a single logical memory for `C` could cause increased wait times in memory,

Listing 5.1: MMM control thread program.

```
#define N 2*NumPEs
#define N_PE NumPEs
#define DTYPE sizeof(float)

int gemm_thread() {
  int NB = N_PE, ram_depth = N_PE * 2;
  cpi_addr dataA = 0, dataB = B_OFF, dataC = C_OFF;
  cpi_int64 token = 0;
  cpi_tag tagA = CPI_INVALID_TAG, tagB = CPI_INVALID_TAG, tagC0 =
      CPI_INVALID_TAG, tagC1 = CPI_INVALID_TAG;

  cpi_register_thread("gemm_thread", N_THREADS);
  cpi_hand cfifo = cpi_get_channel(cpi_fifo, 0);
  cpi_hand ramsA = cpi_get_rams(N_PE, false, 0, 0);
  cpi_hand ramsB = cpi_get_rams(N_PE, true, 0, N_PE);
  cpi_hand ramsC0 = cpi_get_rams(N_PE, false, 0, 2*N_PE);
  cpi_hand ramsC1 = cpi_get_rams(N_PE, false, 0, 3*N_PE);

  bool which = false; // for double buffer
  int prev_i = 0, prev_j = 0;

  for (int k = 0; k < N; k += NB) {
    for (int j = 0; j < N; j += NB) {
      for (int i = 0; i < N; i += NB) {
        int offset = which ? ram_depth/2:0;
        int c_offset = !which ? ram_depth/2:0;
        for(int r=0; r < NB; r++)
        {
          cpi_poll(tagA, cpi_nb_write_ram(ramsA, r*ram_depth+offset, dataA+
              DTYPE*(i*N+k+r*N), NB*DTYPE, tagA));
          cpi_poll(tagB, cpi_nb_write_ram(ramsB, r+offset, dataB+DTYPE*(k*N+j+
              r*N), NB*DTYPE, tagB));

          if(!which) {
            cpi_poll(tagC0, cpi_nb_write_ram(ramsC0, r*ram_depth+offset, dataC
                +DTYPE*(i*N+j+r*N), NB*DTYPE, tagC0));
            cpi_poll(tagC1, cpi_nb_read_ram(ramsC1, r*ram_depth+c_offset,
                dataC+DTYPE*(prev_i*N+prev_j+r*N), NB*DTYPE, tagC1));
          }
          else {
            cpi_poll(tagC1, cpi_nb_write_ram(ramsC1, r*ram_depth+offset, dataC
                +DTYPE*(i*N+j+r*N), NB*DTYPE, tagC1));
            cpi_poll(tagC0, cpi_nb_read_ram(ramsC0, r*ram_depth+c_offset,
                dataC+DTYPE*(prev_i*N+prev_j+r*N), NB*DTYPE, tagC0));
          }
        }
        cpi_wait(ramsA, tagA);
        cpi_wait(ramsB, tagB);
        cpi_wait(ramsC0, tagC0);
        cpi_wait(ramsC1, tagC1);
        cpi_write_channel(cfifo, token);
        token = cpi_read_channel(cfifo);
        prev_i = i;
        prev_j = j;
        which = !which;
      }
    }
  }
}
```

especially in the soft logic implementations of CoRAM (see Chapter 8).

Distributed MMM. The Cor-C architecture specification places a logical limit on the number of CoRAMs that can be combined into a single Co-handle (up to 64, see Chapter 4). For MMM, this limits the number of PEs that can be managed by a single control thread. To scale beyond the limit of 64, multiple PEs can be grouped into **cores** that are replicated to operate on disjoint sub-blocks of the large matrices. This is a parallelization typically employed by multicores and GPGPUs. Assuming that double-buffering completely overlaps the memory transfer time with the computation, the expected performance of the distributed MMM kernel is:

$$GFLOPs/sec = 2 \times N_{pe} \times ghz_{fpga}$$

5.1.6 Discussion

Our example of MMM highlights the simplicity and convenience of using a high-level language such as C to succinctly express the memory access requirements of a highly distributed FPGA-based kernel. The notion of CoRAM allowed us to completely replace portions of the original MMM datapath including the custom network and DMA engine, both of which contributed considerably to the complexity of the design. CoRAM also allowed us to easily express the re-assignment of FPGA kernels to different regions of the external memory over the course of a large computation. This feature of the CoRAM architecture could potentially be used to simplify the task of building out-of-core FPGA-based applications that support inputs much larger than the total on-chip memory capacity. Thus far, our discussion of MMM has not considered the performance of our Cor-C implementation and how it would compare against the manual approach. Chapter 6 will later discuss how the Cor-C architecture specification is mapped into physical implementations; Chapter 8 will perform a detailed quantitative evaluation that illustrates how our implementation of MMM in Cor-C can achieve comparable if not better performance than the manual approach.

5.2 Black-Scholes

Our next example focuses on stream-based computation, which is a common pattern found in many FPGA-based applications. In this section, we introduce a fundamental concept in the CoRAM paradigm called the **memory personality**. A memory personality is a re-usable library component built out of native RTL and from the primitives available in the Cor-C architecture language. Memory personalities are designed to provide an extra layer of abstraction above Cor-C to facilitate interfaces that are better suited for a particular applications' need (in our case, streaming). In the subsections below, we give background of Black-Scholes and describe a detailed example of the Stream FIFO memory personality used to support the application.

5.2.1 Background

The Black-Scholes formula is a popular instrument used in the trading of European-style options [103]. The option is an agreement between a buyer and seller where the buyer is granted the right to exercise the option at a certain time in the future. A *call option* allows the buyer to purchase an underlying asset at a *strike price* at some moment in the future; a *put option* grants the right to sell the underlying asset. A profit is made when there is a difference between the strike price and the actual price of the underlying asset minus the price of the option. The Black-Scholes model shown below gives a partial differential equation for the evolution of an option price under a given set of assumptions [103].

$$\frac{\partial V}{\partial t} + \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 V}{\partial S^2} + rS\frac{\partial V}{\partial S} - rV = 0$$

Where:

- V = the price of the option as a function of time and stock price
- r = the risk-free interest rate
- t = time
- σ = volatility of the stock

For European Options which can only be exercised on the expiration date, a closed-form solution exists for the PDE above (as reprinted from [103]):

$$V_{call} = S \times CND(d_1) - X * e^{-rT} \times CND(d_2)$$

$$V_{put} = X \times e^{-rT} \times CND(-d_2) - S \times CND(-d_1)$$

$$d_1 = \frac{log(S/X) + (r + v^2/2)T}{v\sqrt{T}}$$

$$d_2 = \frac{log(S/X) + (r - v^2/2)T}{v\sqrt{T}}$$

$$CND(-d) = 1 - CND(d)$$

Where:

- V_{call} = price for option call
- V_{put} = price for option put
- $CND(d)$ = cumulative normal distribution function
- S = current option price
- X = strike price of option
- T = time until option expires
- r = risk free interest rate
- v = implied volatility for underlying asset

5.2.2 Parallelization on the FPGA

The Black-Scholes formula is typically used to compute many independent solutions, making it an easy task to parallelize on the FPGA. The Black-Scholes formula employs a rich mixture of arithmetic floating-point operators but exhibits a very simple sequential memory access pattern. Assuming that the input vectors are laid out sequentially in memory, a fully pipelined processing

PE resources (single-precision)	~30KLUTs, 57 DSPs
Clock frequency	300 MHz
Peak PE throughput	600 Moptions/s

Table 7: Black-Scholes Processing Characteristics.

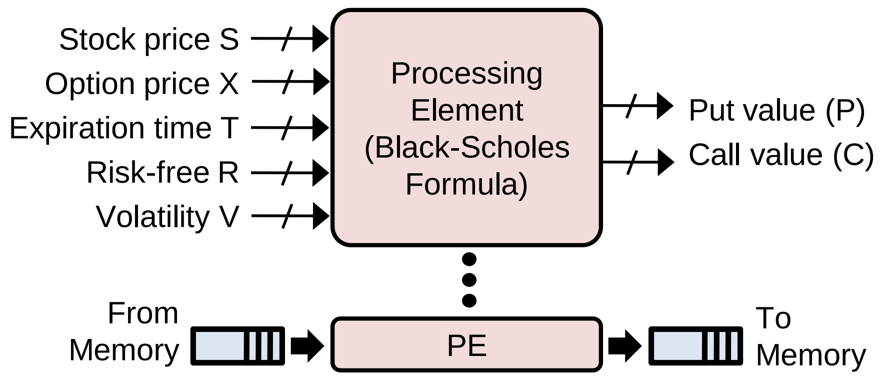


Figure 26: Black-Scholes Processing Element.

element (PE) can be built that streams the data in, computes V_{call} and V_{put}, and writes them sequentially out to memory. From a memory bandwidth perspective, each computed option reads 5 data values and writes 2. Assuming single precision floating point, this amounts to an arithmetic intensity of 28 bytes per option.

Figure 26 illustrates a fully pipelined, single-precision Black-Scholes PE implemented in this thesis. The various floating point operators needed in the Black-Scholes equation are built from a combination of optimized IP cores generated from the FloPoCo framework [34] and Xilinx Coregen [6]. The application's performance is highly scalable; one could increase performance by instantiating multiple PEs that consume and produce independent input and output data streams. Performance continues to scale until either the reconfigurable logic capacity is exhausted or the available external memory bandwidth is saturated. The characteristics of the Black-Scholes PE are shown in Table 7.

5.2.3 Memory Streaming

Streaming from and to external memory is a typical memory access pattern in many FPGA-based applications. In a conventional FPGA, a stream is supported by inserting FIFO buffers between the native memory interfaces and the core logic of the application. The buffers are often im-

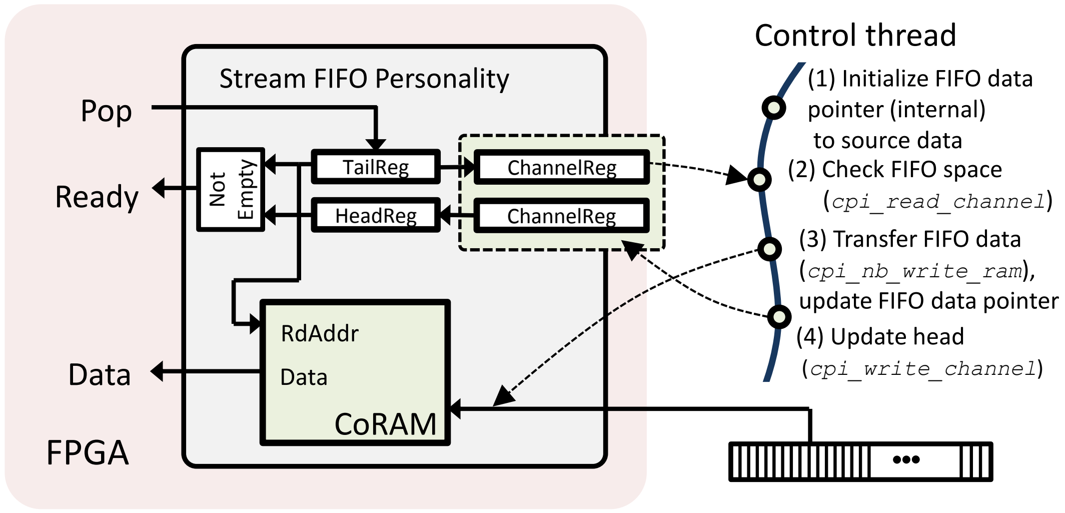

Control thread
(1) Initialize FIFO data pointer (internal) to source data
(2) Check FIFO space (cpi_read_channel)
(3) Transfer FIFO data (cpi_nb_write_ram), update FIFO data pointer
(4) Update head (cpi_write_channel)
Stream FIFO Personality
Pop
Ready
Not Empty
TailReg
HeadReg
ChannelReg
ChannelReg
RdAddr
Data
CoRAM
Data
FPGA

Figure 27: Stream FIFO Memory Personality.

plemented using the available embedded SRAMs in the FPGA. The surrounding FIFO control logic is tasked with issuing memory addresses to the native memory interfaces and filling or draining the buffers as data arrives or leaves. Despite the simplicity of streaming, there is no standard convention or agreement on how FPGA-based applications access streams. Application writers are frequently required to either develop their own stream modules or tie themselves in to vendor-specific offerings (e.g., ACP [72], Convey [27], etc.).

The Stream FIFO Memory Personality. The Cor-C architecture language is intended to raise the level of abstraction by replacing low-level control logic with a portable, software-based specification. In our example, the Black-Scholes processing element can be viewed as a *stream client*, which performs sequential reads and writes to specific starting locations in memory. A stream client simply expects to see a standard FIFO interface without any knowledge of the underlying memory subsystem.

To support stream clients using the Cor-C language, we develop the notion of a *Stream FIFO memory personality*. A memory personality is a library component re-usable across many devices and platforms. From the perspective of the application's core logic, a personality appears to be a black-box module in native RTL with a particular client interface such as a FIFO. The black-box module contains one or more embedded CoRAMs and an associated control thread. Figure 27 illustrates the Stream FIFO memory personality module, which exports to the core logic in native

Listing 5.2: Control program for Memory-to-Logic Stream FIFO.

```
void write_fifo(cpi_hand rams, cpi_hand creg, cpi_addr src,
                int bytes, int word_size, int depth) {

    int words_left = bytes / word_size, src_word = 0, tail = 0, head = 0;

    while(words_left > 0)
    {
        tail = cpi_read_channel(creg);
        int free_words = head >= tail ? depth-1-(head-tail) : (tail-head-1);
        int bsize_words = MIN(free_words, words_left);

        if(bsize_words != 0)
        {
            cpi_write_ram(rams, head, src + src_word * word_size,
                                      bsize_words * word_size);
            src_word   += bsize_words;
            words_left -= bsize_words;
            head = (head + bsize_words) & (depth - 1);
            cpi_channel_write(creg, head);
        }
    }
}
```

Listing 5.3: Control program for Logic-to-Memory Stream FIFO.

```
void read_fifo(cpi_hand rams, cpi_hand creg, cpi_addr dst,
               int bytes, int word_size, int depth) {

    int words_left = bytes / word_size, dst_word = 0, tail = 0, head = 0;

    while(words_left > 0)
    {
        head = cpi_read_channel(creg);
        int used_words = (head >= tail) ? head - tail : depth - (tail-head);
        int bsize_words = MIN(words_left, used_words);

        if(head != tail)
        {
            cpi_nb_read_ram(rams, tail, dst + dst_word * word_size,
                                   bsize_words * word_size);
            dst_word += bsize_words;
            words_left -= bsize_words;
            tail = (bsize_words+tail) & (depth - 1);
            cpi_write_channel(creg, tail);
        }
    }

    return ;
}
```

RTL a standard FIFO interface: `data, ready, pop`. The stream FIFO can be replicated in a single application to support multiple clients if needed.

Anatomy of the Stream FIFO. Figure 27 illustrates the anatomy of the Stream FIFO module, which utilizes a single embedded CoRAM as a circular buffer and as a logical portal to external memory. The Stream FIFO shown performs transfers from memory to reconfigurable logic and is implemented using a combination of RTL and Cor-C. Within the boundary shown in Figure 27, the Stream FIFO contains head/tail pointers and comparator logic used to determine when the FIFO is empty or full. The FIFO consumer (i.e., the Black-Scholes PE) only sees a simple read data interface with a valid bit.

Unlike a typical FIFO, the writer of the FIFO is not an entity hosted in reconfigurable logic but is managed by a single control thread. Figure 27 (right) illustrates a software control thread used to fulfill the FIFO producer role (the corresponding pseudo-code is shown in Listing 5.2). The event highlighted in Step 1 of Listing 5.2 first initializes a source pointer to the location in memory where the starting Black-Scholes data resides. In Step 2, the control thread samples the head and tail pointers to compute how much available space is left within the FIFO (L8-L10 in Figure 5.2). If sufficient space exists, the event in step 3 performs a multi-word byte transfer from the edge memory interface into the CoRAM using the `cpi_nb_write_write` control action (L14-L15 in Listing 5.2). The event in Step 4 completes the FIFO production by having the control thread update the head pointer using the `cpi_write_channel` control action to inform the reconfigurable logic within the stream FIFO module when new data has arrived (L19 in Listing 5.2). Finally, L16-L18 show updates to the internal state maintained by the control thread. For completeness, Listing 5.3 shows the corresponding logic-to-memory Stream FIFO, which performs memory transfers from logic to external memory.

Thread Invocation. Appendix C.1 shows the actual RTL for the Stream FIFO library code implemented in Bluespec System Verilog. In the modules, the embedded CoRAMs and channels are instantiated to work in tandem with the associated control threads; the `thread` names and `object_id` must be matched by convention with the names and values used in the associated control threads. The parameterized RTL allows the application writer to form Stream FIFOs of arbitrary dimensions. In the case of the reader thread, the Stream FIFO is configured with a 20B width (corresponding to

5 floating point inputs) and 1024 entries (the FIFO depth), while the writer thread is configured with a 8B width (2 floating point outputs) and 1024 entries. The code below illustrates how the reader and writer Stream FIFOs are invoked by the control thread of the Black-Scholes kernel.

```
void bscholes_write_thread()
{
    cpi_register_thread("bscholes_write_thread", 1);
    cpi_hand creg = cpi_get_channel(cpi_reg, 0/*obj_id*/);
    cpi_hand rams = cpi_get_rams(5/*width*/, true, 0/*obj_id*/, 0/*sub_id*/);
    write_fifo(rams, creg, READ_VA,
              NUM_OPTIONS*sizeof(float)*5, 5*sizeof(float), 1024);
}

void bscholes_read_thread()
{
    cpi_register_thread("bscholes_read_thread", 1);
    cpi_hand creg = cpi_get_channel(cpi_reg, 0/*obj_id*/);
    cpi_hand rams = cpi_get_rams(2/*width*/, true, 0/*obj_id*/, 0/*sub_id*/);
    read_fifo(rams, creg, WRITE_VA,
             NUM_OPTIONS*sizeof(float)*2, 2*sizeof(float), 1024);
}
```

As shown in the code example above, the Black-Scholes kernel consists of two thread descriptions, one used to stream data from external memory to the kernel and the other used to write results back to memory. The functions described in Listings 5.2 and 5.3 present the appearance of simple library functions invoked by the application. The application writer simply has to specify the starting read and write virtual addresses of the application read and write streams, respectively. In the case where multiple stream clients are desired, the `cpi_register_thread` static control action can be configured with an argument value greater than 1 to instantiate multiple concurrent threads.

5.2.4 Discussion

Our example of Black-Scholes in this section illustrates how the CoRAM paradigm virtualizes the memory subsystem through memory personalities. From our example, a few salient observations can be made. First, the memory personality concept is a hybrid hardware-software entity that targets an abstract intermediate representation presented by the Cor-C architecture specification. The personality presents a higher level abstraction above the Cor-C interface and is a portable, re-usable component that can be invoked by any application that requires its interface. The accessing of memory is achieved easily without requiring any underlying knowledge of the memory subsystem. Memory throughput is scaled by instantiating and replicating memory personalities as many times as needed. Personalities are also highly parameterizable and decoupled from the underlying characteristics of the FPGA memory system. For example, the width of the Stream FIFO can be configured arbitrarily without being coupled to the underlying widths of native memory interfaces on an FPGA.

Although the memory personalities present a simpler, higher level abstraction than a low-level native interface to memory, an open question up to this point is whether these abstractions can be effectively translated into good efficiency and performance, which are crucial requirements in highly tuned FPGA-based applications. We postpone a detailed analysis of performance for now, but show later in Chapter 8 that personalities are scalable and can be supported without major losses in efficiency or performance.

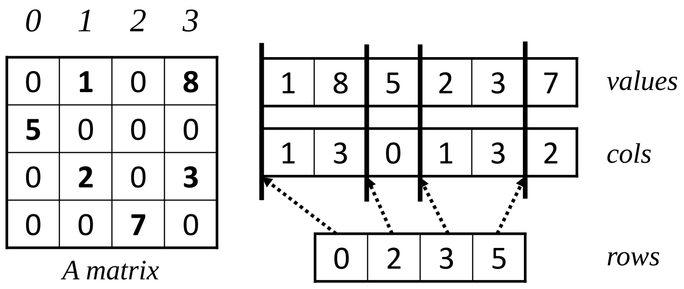


Figure 28: Example of Compressed Sparse Row Format in SpMV.

5.3 Sparse Matrix-Vector Multiplication

The last case study we present in this chapter is the most sophisticated demonstration of CoRAM presented in this thesis. In this section, we show how multiple memory personalities are composed to support an irregular memory access pattern with indirect references. Sparse Matrix Vector Multiplication (SpMV) is a ubiquitious kernel used in many scientific and engineering applications and continues to be actively researched on many different architectures.

5.3.1 Background

The SpMV kernel solves the statement $y = Ax$, where y and x are 1-dimensional dense vectors and A is a matrix populated with mostly zeros. In SpMV, the large A matrix is encoded in a compressed sparse format that only stores the non-zero values. A commonly used encoding is the Compressed Sparse Row (CSR) [2] format, shown as an example in Figure 28. The non-zero values of matrix A are stored in row-order as a linear array of `vals` in external memory. The column number of each entry in `vals` is stored in a corresponding entry in a separate column array called `cols`. The i'th entry of another array (`rows`) holds the index to the first entry in `vals` (and `cols`) belonging to row `i` of `A`. The reference C code for computing $y = Ax$ is given as follows:

```
void spmv_csr (int n_rows, int *cols, Data *vals, Data *rows, Data *x, Data *y)
{
    for(int r = 0; r < n_rows; r++) {
        int sum = 0;
        for(i = rows[r]; i <= rows[r+1]; i++)
            sum += vals[i] * x[cols[i]];
        y[r] = sum;
    }
}
```

As shown, each dot product is accumulated by accessing the linear array `vals` using the row index and by indirect reference to the dense `x` vector through values stored in the `cols` vector.

5.3.2 Related Work

The optimization of the SpMV kernel continues to be an actively studied research problem. SpMV is particularly challenging due to the indirect references of the `x` vector, which can be unpredictable or irregular depending on the sparse patterns of the input matrix. When parallelizing SpMV across multiple compute cores, load balancing is also another issue in performance tuning given that the dot product sizes being accumulated can be of arbitrary length up to the dimension of the matrix. SpMV has been studied and optimized extensively for general-purpose architecture [108, 54, 48, 105, 104, 70, 95, 100, 84]. In the domain of FPGAs, a significant body of work exists in developing hardware SpMV kernels [36, 124, 93, 123, 47]. Many prior works focus exclusively on the development of efficient floating-point accumulators, which forms the heart of the SpMV computation.

The main difficulty in achieving effective throughput on FPGAs is the pipelining of single- or double-precision floating point accumulators. This particular problem stems from the fact that a single floating point add cannot be achieved in a single FPGA cycle with a reasonable clock period. A variety of circuits have been proposed in literature to address this. The most simplest approach is to statically schedule inputs from disjoint rows at an interleaving interval that corresponds to the latency of the floating point accumulator. This approach was employed by deLorimier and Dehon [36] and shown to achieve about 66% peak floating point throughput. More sophisticated approaches have been proposed based on dynamic scheduling, which maintains partially accumulated sums across multiple rows. The state-of-the-art for this approach was demonstrated by Nagar and Bakos, which only required a single double-precision adder and could handle an arbitrary number of accumulation sets with no knowledge of the dot product sizes. A third approach that does not require scheduling, is based on the modification of the floating point adder itself to reduce the latency during the accumulation step [34].

Single PE resources	˜5KLUTs
Clock frequency	300 MHz
Peak PE throughput	600 MFLOP/s

Table 8: Single-Precision Sparse Matrix-Vector Multiplication Kernel Characteristics.

Listing 5.4: Value Stream FIFO Thread

```
void
val_thread()
{
    cpi_register_thread("val_thread", N_PE);

    DECLARE_LDST(ldst_fifo, ldst_ram, 1/*obj_id*/);
    DECLARE_CFIFO_BIND(rams, 1, creg, cfifo, hfifo, bfifo, head, 0);

    cpi_addr val_ptr = mp_thread_read(ldst_ram, ldst_fifo, BASE_ADDR +
        VAL_OFFSET);

    int first, second;

    while(1) {
        cpi_int64 val = cpi_read_channel(cfifo);
        unpack_data(val, &first, &second);
        write_fifo_bind(rams, creg, hfifo, val_ptr+first*sizeof(int), second*
            sizeof(int), 4, FIFO_DEPTH, &head);
    }
}
```

5.3.3 Parallelization on the FPGA

The basic strategy for parallelizing SpMV on the FPGA is to distribute independent dot products across multiple processing elements, each with a single floating point accumulator. Figure 30 illustrates an FPGA design for SpMV, where multiple processing elements (PEs) operate concurrently on distinct rows of matrix A. The contents of the `rows` array are streamed in from external memory to a centralized work scheduler that assigns rows to different PEs. The role of the work scheduler is to track the amount of work pending for each PE and to dynamically load balance the resources to maintain high utilization. The centralized work scheduler is simple to implement in the FPGA due to the fine-grained communication possible within reconfigurable fabric.

For each assigned row, a PE employs two memory-to-logic Stream FIFO memory personalities from Section 5.2 to sequentially read in data blocks from specific location offsets from the `vals` and `cols` arrays, respectively. A single logic-to-memory Stream FIFO writes the accumulated dot-products out to memory. To configure the offsets for each of the two stream FIFOs, the PE logic

Listing 5.5: Cache Thread

```
void
xcache_thread()
{
    cpi_register_thread("xcache_thread", N_PE);
    DECLARE_LDST(ldst_fifo, ldst_ram, 1);

    cpi_hand fifo = cpi_get_channel(cpi_fifo, 0, 0);
    cpi_hand bfifo = cpi_get_channel(cpi_fifo, 0, 1);
    cpi_hand data_ram = cpi_get_rams(XCACHE_WIDE, true, 0, 0);
    cpi_hand tag_ram = cpi_get_rams(1, false, 0, XCACHE_WIDE);
    cpi_addr x_ptr = mp_thread_read(ldst_ram, ldst_fifo, BASE_ADDR + X_OFFSET)
        ;
    cpi_bind(bfifo, data_ram);

    int log_word_bytes = log2(XCACHE_WIDE * (WORD_WIDTH/8));

    while(1) {
        cpi_int64 message = cpi_read_channel(fifo);
        cpi_addr miss_addr = message & 0xffffffffu & ~(XCACHE_BLK_BYTES-1);
        cpi_int64 data_index = miss_addr & (XCACHE_SIZE_BYTES-1) & ~(
            XCACHE_BLK_BYTES-1);
        cpi_tag tag = CPI_INVALID_TAG;
        while(1) {
            tag = cpi_nb_write_ram(data_ram, data_index >> log_word_bytes,
                x_ptr + miss_addr, XCACHE_BLK_BYTES, tag);
            if(tag != CPI_INVALID_TAG) break;
        }
    }
}
```

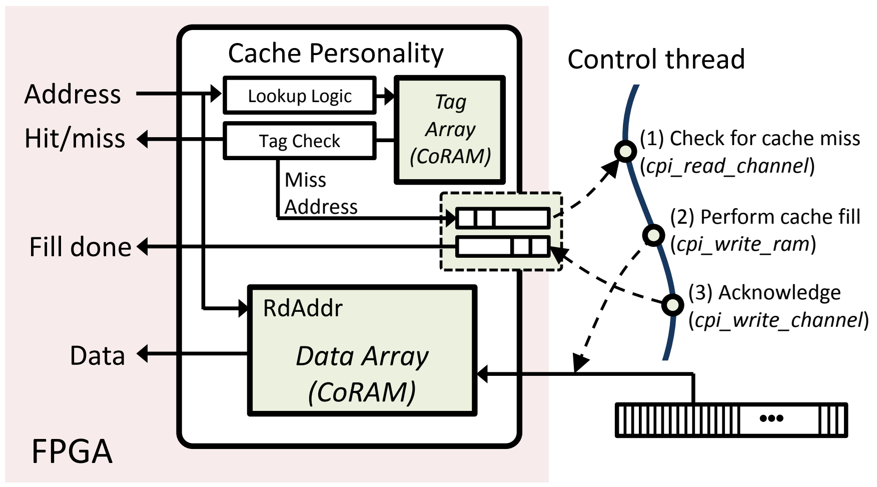


Figure 29: Cache Memory Personality.

must pass row assignment information (via channels) to the control threads belonging to the stream FIFOs for each new dot product that is being accumulated by the PE. The row information describes the logical offset in memory as well as the size of the dot product being accumulated. Listing 5.4 shows the code for the `value` thread that performs this operation.

To compute the dot products, a PE must make indirect references to vector `x` based on the index values received from the `cols` Stream FIFO. A simple approach to handling this case is to have an `x` thread that takes the column value from the `cols` FIFO and directly issues an access to memory. Unfortunately, this approach leads to sub-optimal performance because the access patterns are not necessarily sequential and could be highly irregular or random. Furthermore, `x` can be a very large data structure that cannot fit into aggregate on-chip memory.

Cache Memory Personality. An effective optimization is to exploit any reuse or locality of the elements of `x` across multiple rows through caching. To implement caching within each PE, Figure 29 illustrates a simple **cache memory personality** built using the Cor-C architecture language. Within the cache, embedded CoRAMs are composed to form data and tag arrays while conventional reconfigurable logic implements the bulk of the cache controller logic. A single control thread is used to implement cache fills to the CoRAM data arrays. When a miss is detected, the address is enqueued to the control thread through an asynchronous FIFO (step 1 in Figure 29). Upon a pending request, step 2 of the control thread transfers a cache block's worth of data to the data array using the `cpi_nb_write_ram` control action. In step 3, the control thread acknowledges the cache controller

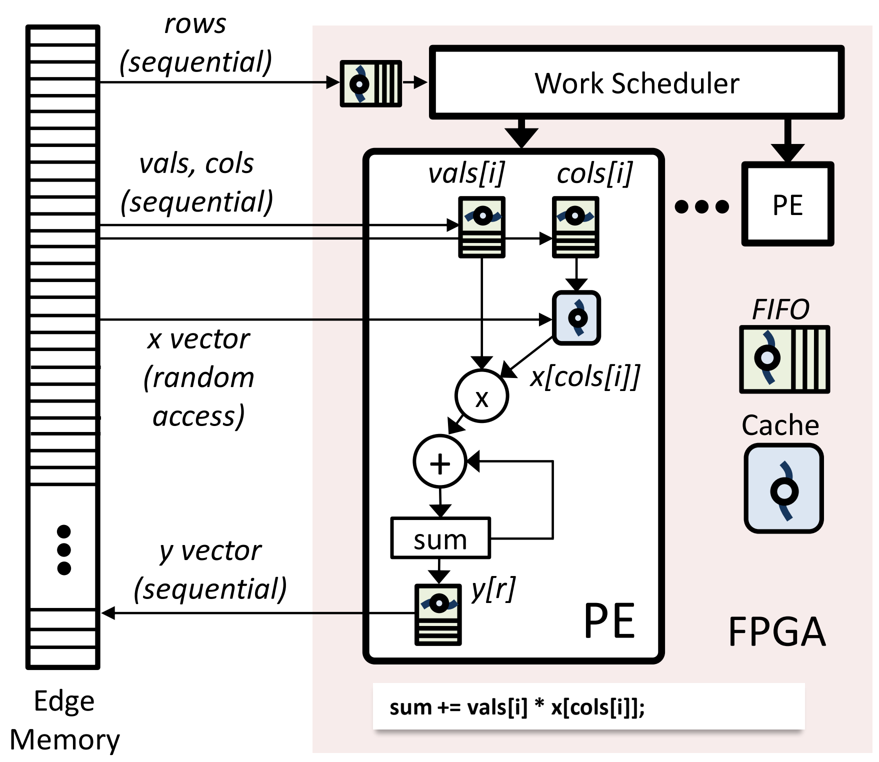


Figure 30: Implementation of SpMV with CoRAM.

using the `cpi_write_channel` control action. The cache controller shown in Figure 29 is somewhat simplified compared to our actual implementation. The cache design is actually further extended to support multiple outstanding misses through the use of MSHRs [88]. Appendix **??** gives further details on this implementation.

Summary. As shown in Figure 30, each individual PE of SpMV employs a total of four memory personalities (3 Stream FIFOs, 1 Cache) attached to a single accumulator. The single SpMV PE forms a highly optimized pipeline where multiple outstanding dot products are being streamed in, computed, and written out. Ideally, the SpMV kernels developed in this section should scale linearly with the number of PEs until either the reconfigurable logic resources are exhausted or off-chip memory bandwidth is fully utilized. Chapter 8 will later perform detailed quantitative evaluations of our SpMV implementation.

5.3.4 Discussion

The SpMV example demonstrates how different memory personalities built out of CoRAM can be composed to support different memory access patterns. Caches, in particular, were used to support the random access pattern of the `x` vector, whereas the stream FIFOs were re-used from Black-Scholes for the remaining sequential accesses. In our development efforts of SpMV, the instantiation of CoRAM-based memory personalities along with spatially-distributed PEs allowed us to quickly instantiate "virtual taps" to external memory wherever needed. This level of abstraction was convenient as it allowed us to concentrate our efforts on optimizing only the processing components of SpMV.

5.4 Summary

What should be apparent from the examples presented in this chapter is that control thread programs are relatively easy to develop compared to conventional RTL and are succinct in expressing the memory access pattern requirements of a given application. While the CoRAM architecture does not eliminate the effort needed to develop optimized stand-alone processing kernels, it does free application writers from having to explicitly manage memory and data distribution in a low-level RTL abstraction. Specifically, control threads did not require us to specify the sequencing details at the RTL level. Further, Cor-C did not limit our ability to support fine-grained interactions between the core logic and the control threads. Fundamentally, the high-level abstraction provided by CoRAM and Cor-C is what enables portability across multiple hardware implementations, as will be demonstrated in Chapter 8. Lastly, the memory personalities demonstrated in our examples (Stream FIFO, Cache) are by no means sufficient for all possible applications and only offered a flavor of what is possible with CoRAM. It is conceivable in the future that soft libraries consisting of many types of memory personalities could be built and shared across many applications.

Chapter 6

CoRAM Memory System

Software comes from heaven when you have good hardware.

Ken Olsen, Founder of Digital Equipment Corporation

A deliberate separation exists between what the CoRAM application writer perceives and that of the memory mechanisms that lay beneath the abstraction. Like any general-purpose memory subsystem, CoRAM can only be practical if the area, performance, and power overheads do not overwhelm the benefits gained through high-level abstraction and portability. A recurring problem visited throughout this chapter is the question of "hard versus soft"—that is, what components of the CoRAM architecture merit implementation in pre-fabricated silicon, and what components can be supported practically in soft logic.

Throughout the process of devising a microarchitecture for CoRAM, this chapter contributes: (1) identification and development of core mechanisms needed in both hard and soft implementations to support CoRAM effectively, (2) a lower-bound cost analysis for data distribution in soft logic designs, and (3) the design and implementation of a distributed, cluster-style microarchitecture that can be adapted to either soft or hard implementations of CoRAM.

6.1 Devising a Memory System for CoRAM

Efficient data distribution forms the heart of any practical CoRAM memory microarchitecture. From the application writer's perspective, the delivery of data from memory into a specific CoRAM

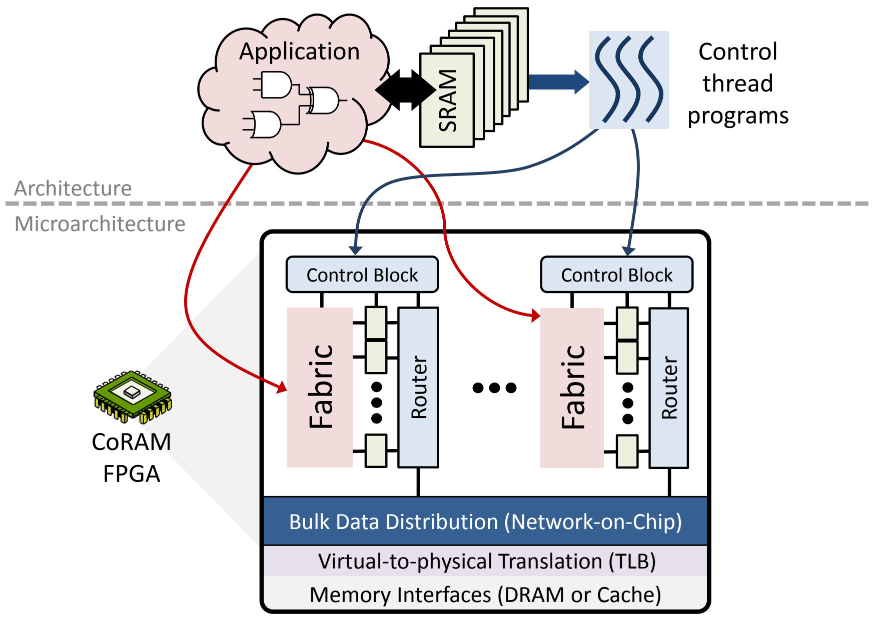


Figure 31: Beneath the CoRAM Architecture.

appears implicit and automatic. Beneath the abstraction, CoRAM requires a high-speed datapath that connects all of the edge memory interfaces to any embedded CoRAMs that are employed by the application.

Figure 31 illustrates the physical archetype of an FPGA with CoRAM memory system support. From top to bottom, the FPGA comprises: (1) the embedded CoRAMs used to store application data, (2) a collection of control blocks used to translate high level control thread programs into low-level memory transactions and control signals, (3) a distribution mechanism for marshalling and steering data between endpoints, and (4) the edge memory components at the boundaries of the fabric, which comprise translation-lookaside tables (TLBs), caches, and memory controllers.

Data Distribution. CoRAM requires a high performance data distribution network for transporting data between the edge memory interfaces and the embedded CoRAMs. Ideally, the network should be provisioned with sufficient bandwidth and latency to sustain the core logic. Further, the mechanisms should perform robustly across many applications' memory access patterns without requiring substantial tuning by the application writer.

The requirements for data distribution can be subdivided into: (1) bulk data transfers and (2) fine-grained steering and alignment to individual CoRAMs. Bulk data movements occur when high-level memory control actions are translated into memory requests that must be routed to remote memory interfaces. Once memory responds with a block of data, it must be subdivided into smaller-sized words that are aligned and steered to specific destination CoRAMs.

Edge Memory Interfaces. This thesis assumes that future FPGAs devised for computing will incorporate hard memory controllers and caches at the edges of the fabric (see Figure 31). Current and future memory technologies such as DDR3 can already operate at I/O speeds that vastly exceed the clock frequencies of conventional fabric [28]. Commodity DDR3-1600, for example, operates at 800MHz on the I/O bus and transfers data on both edges. Today's fastest FPGAs can barely operate practically in the 300MHz range, which severely limits the amount of bandwidth sustainable by fabric. This thesis assumes that future FPGAs will incorporate hardened memory interfaces to keep pace with expected memory technology trends.

The increasing gap between memory clock speeds and the FPGA's clock has the unfortunate side-effect of making fabric memory interfaces appear wider and wider. For example, an FPGA operating at 200MHz would "see" a standard DDR3-1600 interface as a 512b datapath. To compound this problem further, commodity DRAMs are typically optimized for bulk transfers (i.e., cache block sizes) and become inefficient when multiple column accesses are not made to contiguous addresses [58] (e.g., 8n-prefetch architecture of DDR3). To bridge the interface gap between fabric and edge controllers, high-speed stream buffers or caches can be inserted between the fabric and the memory controllers as shown in Figure 31. A highly-banked cache, for example, would allow the backend ports to scale as needed with memory speeds while providing multiple, slower fabric-speed interfaces without reducing the overall throughput to the external interfaces. Commercial vendors such as Convey [27] have also observed this problem and addressed it by creating custom "scatter-gather" memory DIMMs that allow full bandwidth at narrow data widths.

Virtual Memory. The memory accesses that are issued by control threads are assumed to be virtual addresses in CoRAM. Supporting virtual memory within the FPGA requires translation look-aside tables (TLBs) either at the boundaries of the fabric or near the control threads. It is assumed in this thesis that a nearby general-purpose processor (either on-chip with the FPGA or off-chip) is responsible for running the operating system that populates the requisite page tables within global virtual memory. TLB miss handling on the FPGA would be handled through either an on-die dedicated FSM direct access to the memory controllers (e.g., x86-style hardware-managed

TLBs) or through special-purpose programmable microcontrollers responsible for TLB replacements. Companies such as Convey [27] have already demonstrated the plausibility of virtual memory in commercially-available FPGA-based systems. Although virtual memory forms an important component in CoRAM, it will not be a major focus of this thesis.

6.2 CoRAM Memory System: Hard or Soft?

Today's FPGAs contain an abundance of reconfigurable logic, flip-flops, and programmable interconnect, any of which can be used to implement the mechanisms required for CoRAM. It is instructive to first provide a clear definition on what "soft logic fabric" means as distinguished from "hard logic". Soft logic fabric generically refers to any logic function implemented as a series of programmable gates connected through a programmable routing fabric [26]. Hard logic, on the other hand is typically a dedicated structure embedded within the FPGA that could otherwise also be implemented in soft logic fabric. From this definition, it is clear that modern FPGAs are already heterogeneous and contain a varied mixture of hard blocks such as flip-flops, multi-bit block RAMs, I/O transceivers, DSPs, and clock generators [115]. Rose et al. distinguishes heterogeneity based on granularity [8]. Soft logic heterogeneity, such as the dedicated carry chains per logic block, is replicated homogeneously across the chip. This type of heterogeneity is distinguished from another style where nearby tiles are completely different from that of logic blocks—e.g., BlockRAMs or DSPs in Xilinx FPGAs [115].

The decision whether to implement application-specific circuitry within the FPGA as hard or soft logic fabric is challenging, especially when economics, markets, and programmability are factored into the decision making. On one hand, although hard circuits provide dramatic benefits in improving the area, speed, and power efficiency of a given functionality [67], the consumed area becomes wasted if the circuitry is not utilized by a particular benchmark or application. The issue of applicable workloads presents an unusual challenge for CoRAM because it does not have an established market or user base. Further, given that today's FPGA are not architected for computing purposes, it may be difficult to suggest architectural changes to the status quo, especially when the vast majority of the FPGAs in the market today are targeted towards non-compute applications such as high-speed communications or computer systems emulation.

It can be argued, nevertheless, that unlike application-specific circuits, CoRAM addresses a general-purpose need that spans all applications that utilize memory on the FPGA. The embedding of hard memory controllers in commercial FPGAs already suggests that dedicated memory systems will become the norm in future FPGAs, whether for computing purposes or otherwise. The central question of this chapter is thus: *if general-purpose memory mechanisms are beneficial to a wide range of applications, what subset of the CoRAM mechanisms merit implementation in the FPGA itself?*

To answer these questions, the remainder of this chapter will focus on implementing some of the required mechanisms of CoRAM in soft logic and determining what components and structures present the largest sources of overhead. As noted by Rose et al. [8], an alternative to hardening is to enhance the soft fabric itself in ways that can perhaps benefit all applications without over-specializing. In the remainder of this chapter, we will identify opportunities where soft fabric can be enhanced to support CoRAM efficiently.

6.3 Understanding the Cost of Soft CoRAM

To answer the soft versus hard question, we begin by first determining the cost of CoRAM implemented in soft logic fabric. This section will primarily focus our attention and analysis on data distribution, which constitutes the largest source of overhead. We begin with a simple cost analysis followed by a more detailed microarchitectural design that will also address the overheads of control and state management. Our analysis and optimizations in the sections below will primarily target the Virtex-6-style architecture from Xilinx.

Simple Shift Register Network. Figure 34 illustrates a simple network that represents a "minimalist" method for providing full read-write connectivity between a single b-wide memory interface and r CoRAMs. At each FPGA clock cycle, b/s words of data can be injected into the network as long as each s data word is destined for a CoRAM in an independent lane. At each stage of the shift register network, a mux selects between the data of the previous stage and the output of the current CoRAM. It should not be difficult to see that such a network minimizes the mux depth at each stage at the cost of a worst-case latency of rs/b, which can be significant for large FPGA configurations (e.g., $r > 1000$).

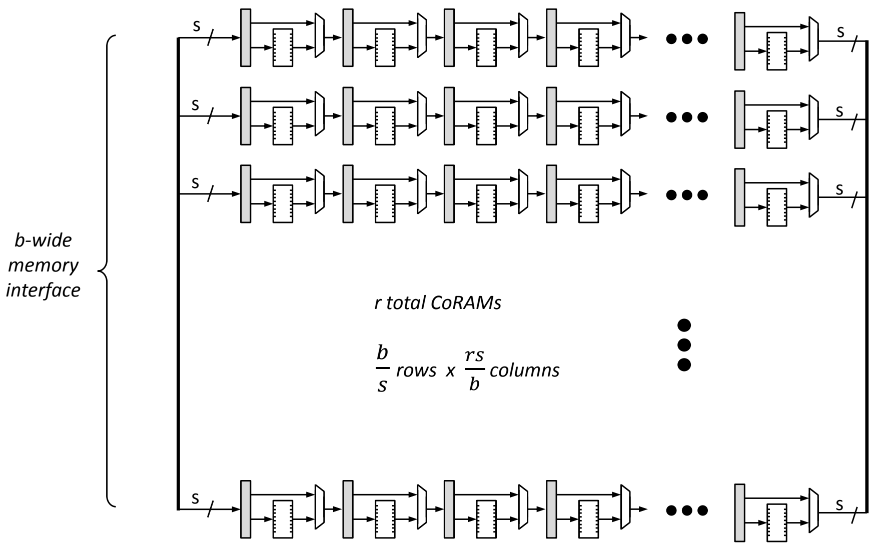


Figure 32: Shift Register Network for CoRAM.

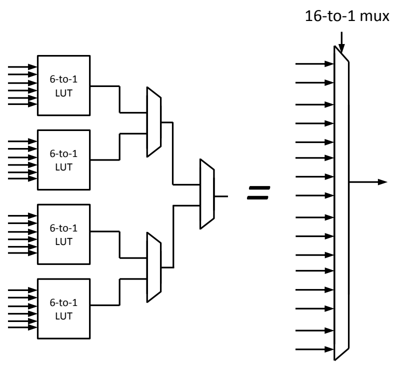


Figure 33: Efficient 16-to-1 multiplexer using 6-input LUTs [115].

In practice, deeper multiplexers up to a certain depth can be implemented in modern FPGAs without the cost of intermediate gates. The Xilinx Virtex-6 architecture, for example, allows implementing a 1-bit, $16 : 1$ mux using only four 6-input LUTs, as shown in Figure 33. Figure 34 shows a refined architecture that aggregates multiple CoRAMs into a single stage, which reduces the delay to rs/bk, where k is the height of each mux. The parameter k can be viewed as a configurable parameter that tunes the delay and mux area of the shift register network.

Multiple Memory Interfaces. Figure 35 extends the previous design to multiple memory interfaces, where the shift register network is now multiplexed across $m \times b$-wide interfaces. A new parameter g, splits the original chain into multiple clusters, each containing a private group of r/g CoRAMs. Separating the chain into multiple clusters allows concurrent accesses between the m in-

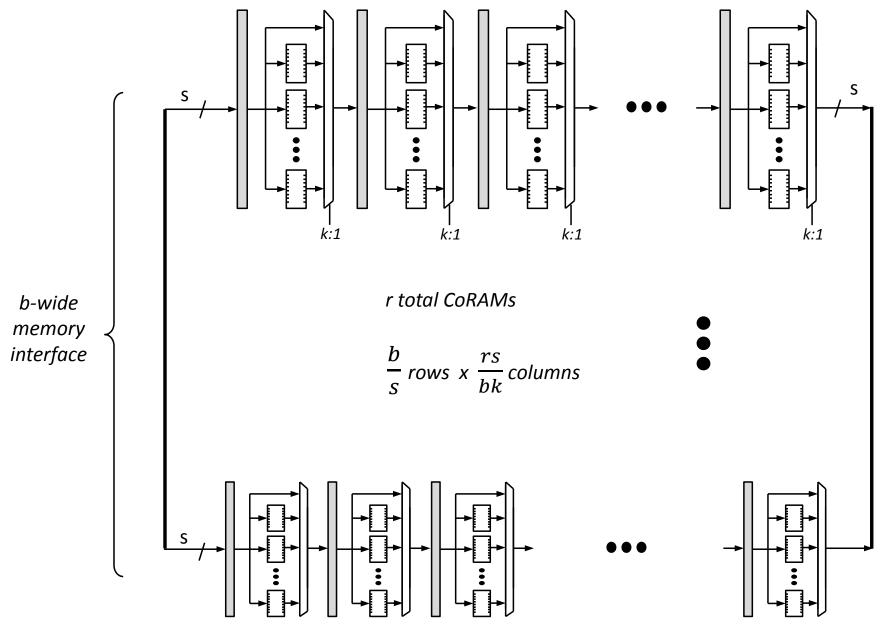


Figure 34: Shift Register Network (Combined) for CoRAM.

terfaces and the g clusters, which is supported by a b-wide switch that connects m memory ports to g clusters. A generic high-level view of the "cluster-style" microarchitecture is shown in a stylized fashion in Figure 36, where distinct clusters of CoRAMs are connected globally using a network that provides bulk data distribution.

Key Relations. Table 9 establishes the key relations in this style of microarchitecture. The free variables set by the designer include f, the nominal clock frequency of the FPGA fabric, s, the port access width of each individual CoRAM, r, the total number of CoRAMs per FPGA, B, the aggregate off-chip bandwidth in GB/s, b, the bit-width of each independent memory access port, g, the total number of clusters, and k, the mux height used to tune the delay and area of each cluster.

The cost of data steering can vary dramatically based on the result of three key variables: b, g and m. When g or m assume large values, the cost of the network dominates due to the increased number of paths that exist between m memory interfaces to multiple clusters g. To maintain a minimally-balanced system where all clusters can concurrently transfer B's worth of bandwidth, the relation $g \geq m$ must be kept true to avoid under-utilizing the memory bandwidth. In many cases, a value of g greater than m is desired to provide increased localized bandwidth to individual, small clusters. This can become necessary when multiple applications share a single FPGA or when memory traffic to a specific cluster is bursty.

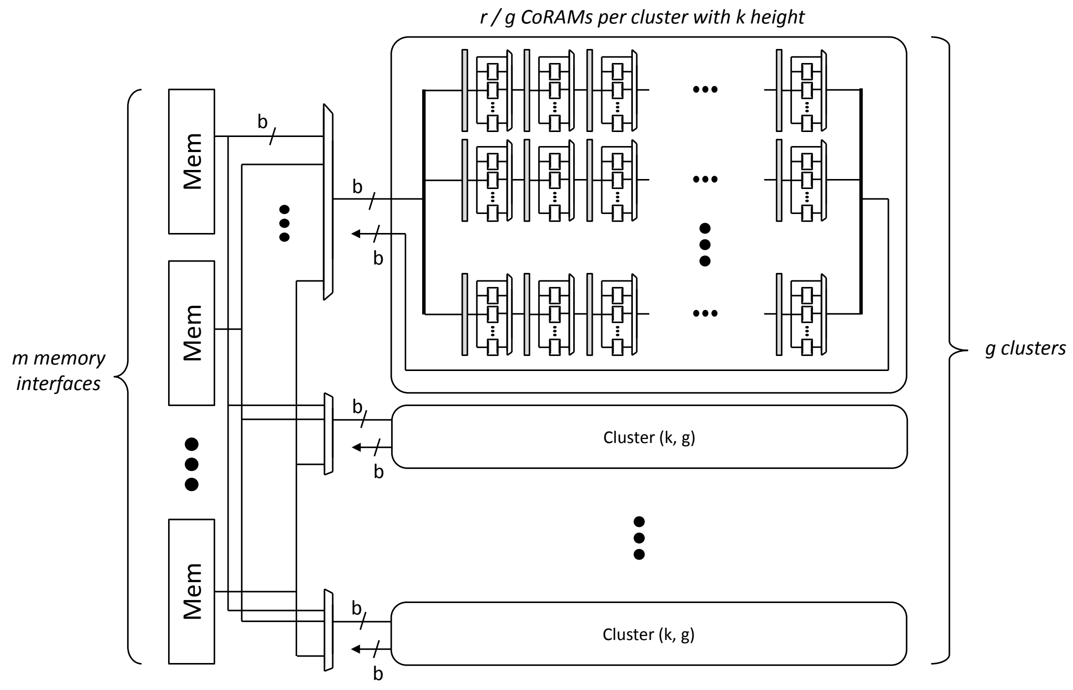


Figure 35: Multiple Memory Interfaces.

The b parameter corresponds to how wide of a datapath individual memory requests and responses correspond to. Increasing b lowers the overall number of clusters required to balance the memory traffic (and hence, the overhead of the network); however, b cannot assume unrealistically high values due to the potential waste of bandwidth. Generally, it is not the case that all applications can be constructed in such a way that all memory accesses occur in large bulk transfers (see Sparse Matrix-Vector Multiplication example in Section 5).

Parameter Selection and Estimates. To populate our free variables with meaningful parameters, Table 10 provides mux estimates across a variety of nominal FPGA configurations based on the technological trends discussed in Chapter 2. The four configurations shown represent a spectrum of baseline FPGAs ranging from designs inspired by real devices today to more speculative FPGAs in the future that can sustain high levels of bandwidth to the fabric.

The bottom of Table 10 shows a variety of costs by setting g equal to multiples of m, where is m is determined automatically by the FPGA configuration and the selection of $b = 128$. As can be seen, the overheads are generally acceptable (under 10%) when the number of clusters exactly equals the total number of memory ports (m). It can be seen that when provisioning for increased localized bandwidth per cluster (e.g., $g = 2 \times m$), the overhead increases non-negligibly.

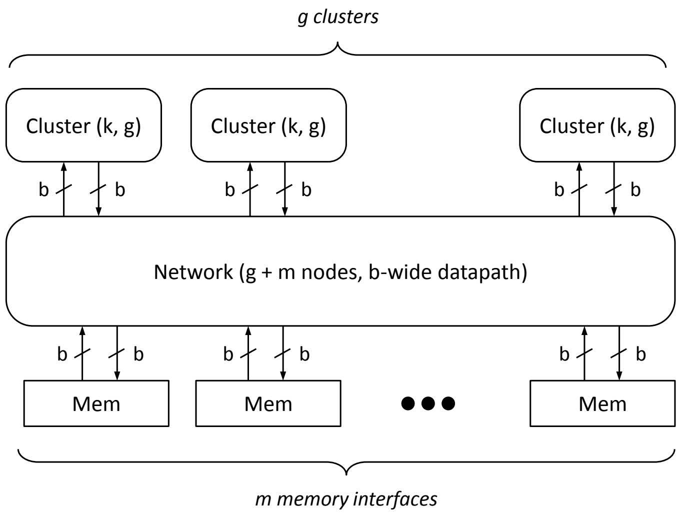


Figure 36: Cluster-Style Microarchitecture.

Summary. Overall, the relative overheads increase as we look more towards the future. Very large FPGAs with up to thousands of SRAMs and hundreds of GB/sec of memory bandwidth can incur substantial overheads due to the need to connect so many CoRAMs to distinct interfaces (15%). The initial analytical results here suggest that a potential avenue for reducing the cost of data distribution would be to introduce dedicated distribution mechanisms to mitigate the MUX overheads. Our analysis thus far has only evaluated the mux cost of data steering and has ignored other important factors such as overheads and programmability. Buffers, control logic, flow control, address generation, state machines, and other "overheads" are additional area costs that cannot be ignored when implementing data distribution in soft logic.

6.4 Implementing A Cluster-Style Microarchitecture for CoRAM

In this section, we devise a microarchitecture that satisfies the requirements outlined in Section 6.3. The design is based on the cluster-style microarchitecture and is both adaptable to soft or hard logic implementations. The soft version is architected in mind to be highly parameterizable and only expends the minimal soft logic area needed to support a particular application. The soft design further employs FPGA-specific optimizations to minimize area overheads. As will be discussed towards the end of this section—with little effort, the same cluster-style microarchitecture with fixed parameters can also be hardened into dedicated silicon for improving efficiency and performance.

Parameters	
FPGA nominal fabric frequency (GHz)	f
Data port-width of single embedded CoRAM	s
CoRAMs per FPGA	r
Aggregate memory bandwidth (GB/s)	B
Fabric-level memory interface bit-width	b
Total clusters	g
Cluster mux height	k
Relations	
CoRAMs per Cluster	r/g
Total number of memory interfaces	$m = (8 \times B)/(f \times b)$
Rows per cluster	$y = b/s$
Columns per cluster (=latency)	$x = (r \times s)/(b \times k)$
Per-CoRAM Bandwidth (GB/s)	$(f \times b)/(8r/g)$
s-wide $(k:1)$ muxes (across all clusters)	$x \times y \times g$
b-wide $(m:1)$ muxes (across all clusters)	g
b-wide $(g:1)$ muxes (across all memory)	m

Table 9: Summary of Key Relations in Cluster-Style Microarchitecture

	Small	**Medium**	**Large**	**Future**
Technology	45nm	32nm	22nm	16nm
6-input LUTs (K)	500	1000	2000	4000
FFs (K)	1000	2000	4000	8000
Fabric Frequency (MHz)	200	200	225	250
DRAM Interfaces	2 x DDR3-1600	4 x DDR3-1600	4 x DDR4-3200	8 x DDR4-3200
DRAM Bandwidth (GB/s)	25.6	51.2	102.4	204.8
4kB CoRAMs	1024	2048	4096	8192
Model parameters	$s = 32, b = 128, k = 16$			
	$B = 25.6$ $m = 8$	$B = 51.2$ $f = 0.2, m = 16$	$B = 102.4$ $f = 0.25, m = 26$	$B = 204.8$ $f = 0.3, m = 43$
LUTs ($g = m$)	13980	36152	104828	305799
CoRAMs per cluster	128	128	160	192
BW per CoRAM (MB/s)	25	25	25	25
Area overhead (%)	**3%**	**3.5%**	**5%**	**7.5%**
LUTs ($g = 2 \times m$)	23864	80497	250103	777231
CoRAMs per cluster	64	64	80	96
BW per CoRAM (MB/s)	50	50	50	50
Area overhead (%)	**5%**	**8%**	**12.5%**	**19.5%**
LUTs ($g = 4 \times m$)	64113	234721	736002	2045870
CoRAMs per cluster	32	32	40	48
BW per CoRAM (MB/s)	100	100	100	100
Area overhead (%)	**13%**	**23.5%**	**37%**	**51%**

Table 10: Estimated MUX Costs Across Projected FPGA Configurations

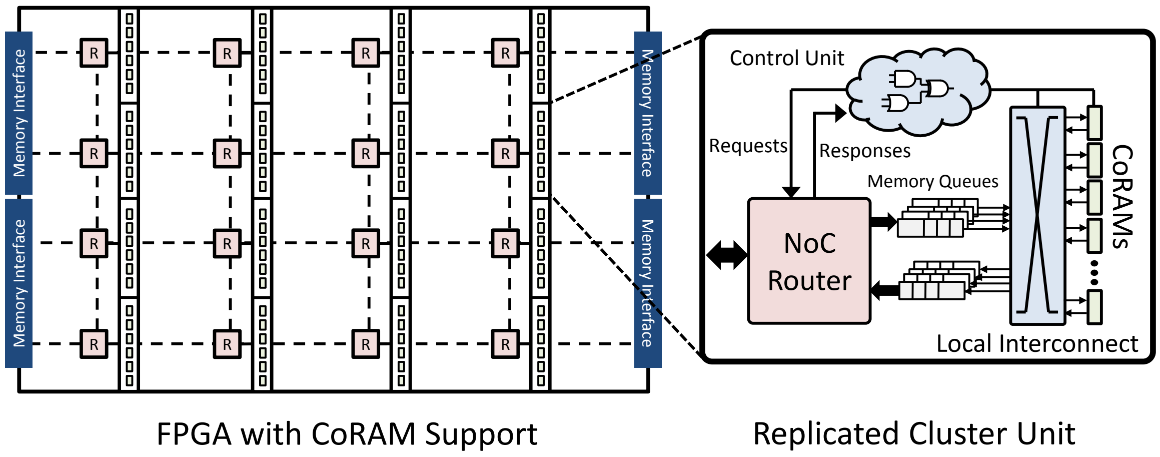


Figure 37: Cluster-Style Microarchitecture for CoRAM.

6.4.1 Design Overview

Figure 36 shows a complete overview of the cluster microarchitecture, consisting of address generation, the memory-to-CoRAM datapath, as well as the network-on-chip and memory. In a soft fabric design, the logical embedded CoRAMs used by an application must be mapped into physical existing SRAMs on an FPGA. The SRAMs, in turn, are organized into **clusters** that provide connectivity to memory interfaces (e.g., memory controllers) situated at the edges of the fabric. The **cluster** is a fundamental building block of the CoRAM memory subsystem, which simultaneously handles requests, control logic, and local data steering for a finite collection of physical CoRAMs. Figure 37 illustrates how physically-mapped CoRAMs are organized into soft clusters connected over a network-on-chip. Adjacent to the clusters are the control threads, which act as clients that submit memory address requests and status queries to the clusters. In a soft implementation of CoRAM, one or more clusters are instantiated as-needed to support the total number of CoRAMs used by the application. The number of clusters to instantiate depends on the maximum number of CoRAMs allowed per cluster (typically 32 to 64) and the desired bandwidth to the memory subsystem.

Parameters. We introduce several parameters to guide our discussion of the cluster design. Each instantiated cluster is provisioned N CoRAM slots where each slot provides a physical 4kB CoRAM corresponding to a discrete physical SRAM on a Virtex-6 FPGA. At compile-time, the multiple CoRAMs employed by any given control thread are typically mapped into the N CoRAM slots

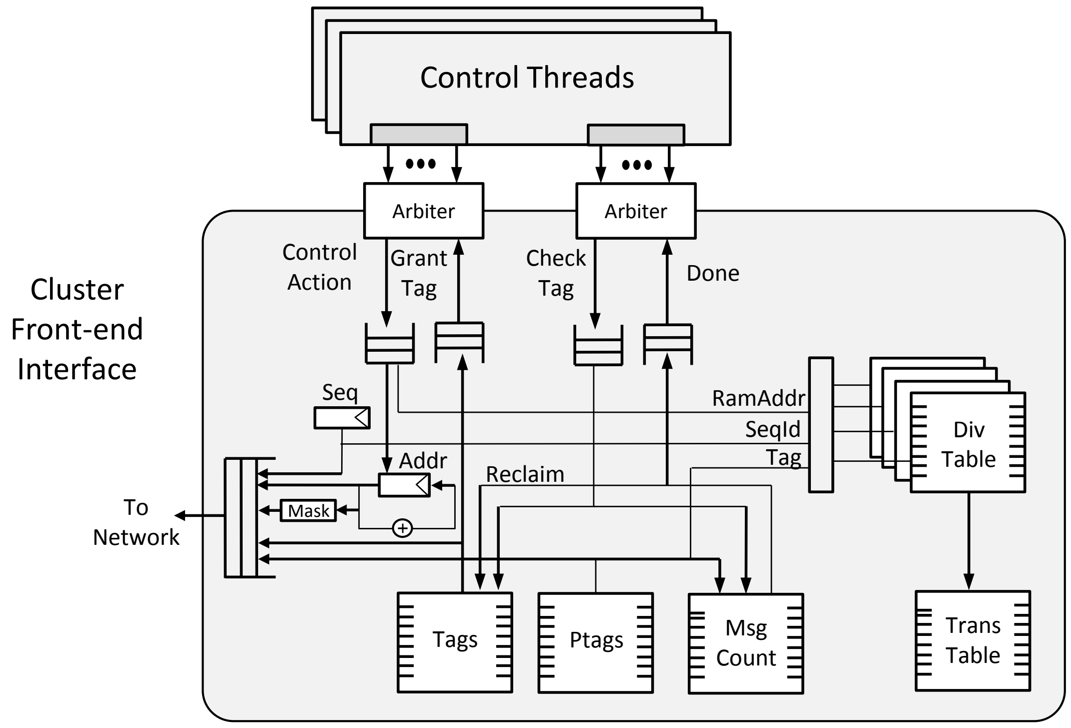


Figure 38: Cluster Front-End Interface.

within a cluster. Each cluster will typically host 8-64 CoRAMs, depending on the configuration set by user-guided policies.

From the perspective of the user and the cluster, each individual CoRAM provides a dual-ported read-write memory with s-wide data ports, where s is assumed to be 32b when targeting a Virtex-6 FPGA. Assuming 4kB words and a 32b datapath, each CoRAM stores 1024 words and is accessed using a 10-bit local RAM address. In the soft logic implementation, one SRAM port is dedicated to the cluster logic, while the other is exported to the application.

A key parameter b defines the link width between the cluster and the on-chip network. The link width places an upper limit on the bandwidth and throughput between a single cluster (and hence its private set of CoRAMs) and the edge memory interfaces. Typical values of b range from 64- to 256-bits, depending on the implementation and the desired throughput to memory.

6.4.2 Clients and Interfaces

The control threads of an application, which exist either in the form of soft finite state machines or as microcontrollers (as described in Chapter 4), can be viewed as clients that issue memory requests to the clusters. In the soft implementation of CoRAM, multiple control threads may share a single cluster, and a single control thread may also access multiple clusters. A memory access begins with a control thread that issues a memory control action over the wire-level interface shown in Figure 38. Recall from Chapter 3 that memory requests are generated using software control actions such as:

```
tag = cpi_nb_write_ram(
                    cpi_hand rams,          /* Cohandle */
                    cpi_ram_addr ram_addr, /* RAM local address */
                    cpi_addr addr,          /* virtual memory address */
                    cpi_int N,              /* N bytes */
                    cpi_tag tag_append      /* append to existing tag */
             );
```

The fields of the wire-level transactions shown in Table 11 correspond to the arguments specified by control actions at the software-level. The `rams` argument defines the co-handle that points to a collection of one or more CoRAMs hosted by the cluster. In hardware, this argument is mapped into several sub-fields (`map`, `width` and `map`) that will be explained in further subsections below.

The `ram_addr` argument determines the local address of the logical memory that which the transfer operations pertain to. The `ram_addr` argument is interpreted contextually depending on the configuration of the co-handle. For example, four 1024x32-bit CoRAMs that are composed linearly in a co-handle would have a valid 12-bit address space for accessing its individual 32-bit words. Conversely, four 1024x32-bit CoRAMs that are composed in scatter-gather would have a valid 10-bit address space, where each word of the logical co-handle is viewed as 128-bits wide.

Address Requests and Tag-based Allocation. The cluster microarchitecture is designed to support concurrent memory transactions to increase throughput to the memory system. A tag-based allocation scheme helps achieve this objective by allowing out-of-order delivery of messages and multiple transactions between simultaneous threads. Out-of-order gives flexibility to other components such as the network-on-chip and the edge memory cache subsystem.

Within each cluster, an issued control action by a control thread results in the transmission of one or more tagged memory requests to the network and edge memory interfaces. A N-byte control action, for instance, would result in $N/8b$ separate requests to memory. When a control action is

Signal	Width	Description
prio	1-bit	High-priority request
rnw	1-bit	Read-not-write request
ram_addr	16-bit	CoRAM Word Address
mem_addr	32- or 64-bit	Virtual Memory Address
bytes	12-bit	Size of memory transaction
width	$log_2(N_{corams\_per\_cluster})$	Number of CoRAMs to access
width_idx	5-bit	Division lookup table index
map	5-bit	Cohandle-to-CoRAM map table index
append	1-bit	Append to existing transaction
append_tag	$log_2(N_{tags\_per\_cluster})$	Tag to use during append

Table 11: Transaction Interface

requested, a thread receives a logical tag from a FIFO-based freelist that corresponds to a single control action as well as the allocated resources within the cluster. A thread is responsible at a later time to query the status of all the tags it receives. Due to re-ordering, tags are not guaranteed to complete in order of allocation.

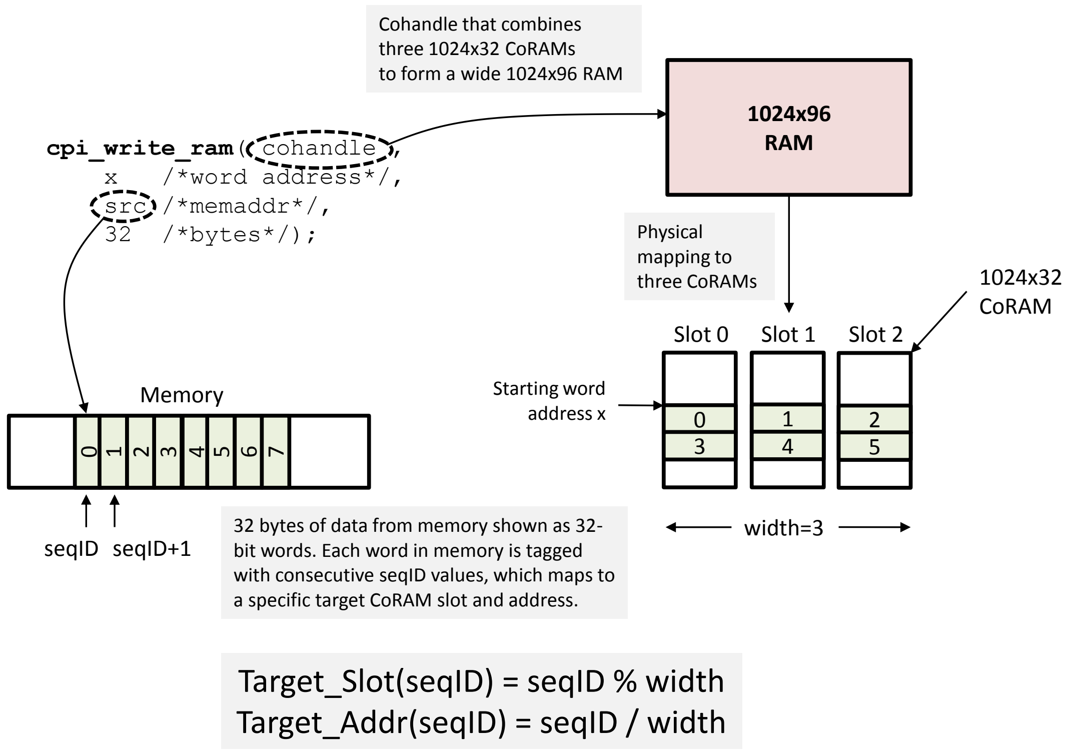
Figure 38 shows how the logical tag is used to index and allocate various hardware structures active throughout the duration of a given transaction. The *TransTable*, for instance, records information necessary for lookups once memory responses arrive. Another structure, *MsgCount*, tracks the total number of outstanding messages corresponding to a single logical control action. The *Ptags* structure is a separate freelist that contains physical tags associated with each individual memory message—individual physical tags are used to index structures that track per-word steering information used when memory responses arrive (as well be discussed further below).

6.4.3 Message Sequencing and Out-of-Order Delivery

When a data response arrives for a memory control action, the cluster is responsible for steering the data appropriately to the destination CoRAMs named by the co-handle used in the control action. Recall from Chapter 4 that software co-handles are used to represent one or more physical CoRAMs functioning as a single logical unit. The ability of co-handles to encapsulate arbitrary aspect ratios of CoRAMs requires the cluster to handle various steering and alignment scenarios. For example, a "scatter-gather" co-handle combining 4 separate CoRAMs would expect a $N = 16B$ stream of data to be split into 4 individual words, each written to separate CoRAMs. Conversely, writing a large stream of data to a "linear" co-handle would write as much as data to the first CoRAM until the end of the 10-bit local address is reached, followed by the second CoRAM, and so forth. For each s-wide word of data, the steering logic must uniquely determine: (1) the CoRAM slot target from 0 to $N-1$, and (2) the 10-bit local address to use when writing the s-wide word to the CoRAM. Another requirement is the need to support memory responses that may arrive out-of-order of issue.

Figure 39 illustrates a sequenced mapping approach that simultaneously addresses the steering problem and the handling of out-of-order message delivery. When a control action is executed, every s-wide word of data of the sequential memory stream is tagged with a small, consecutively increasing sequence ID. The sequence ID allows simultaneous computation of both the CoRAM slot (i.e., a value between 0 and $N-1$) and the local address of where the destination word belongs. For each word with a given sequence ID i, the CoRAM slot target is computed by taking $i\%width$, where $width$ is the logical width of the co-handle divided by s. For example, a co-handle configured as 1024x128b would have $width = 128/32 = 4$. The local address of the target CoRAM is computed by taking `ram_addr` $+i/width$, where `ram_addr` is the local address specified as an argument in the original control action. In our implementation, the sequence ID is a simple 16-bit field attached to every memory request and facilitates word-level steering, address calculation, and in-place delivery of memory messages into the CoRAMs.

Implementation. The mapping functions for steering requires division in hardware if the $width$ field is not assumed to be a power-of-two. To support this operation efficiently on the FPGA, we exploit the fact that local RAM address spaces are small (between 10b to 16b) and can be efficiently implemented using lookup tables hosted in just 1 or 2 BRAMs. A further optimization that can be exploited is the fact that not all dividend-divisor combinations are required given that only a finite number of CoRAMs and co-handles are mapped into a given cluster. Prior to runtime, the division lookup tables are populated only with certain dividend-divisor combinations that are offset by `width_idx`, corresponding to the wire-level transaction field as shown in Table 11. Figure 38 (right) shows a division lookup stage that takes as argument the `width_idx` field along with an assigned sequence ID i to compute the CoRAM slot target and the target RAM address. Both results are stored into the "TransTable" for lookup later once a memory response arrives.



Cohandle that combines three 1024x32 CoRAMs to form a wide 1024x96 RAM
1024x96 RAM
cpi_write_ram(cohandle,
x /*word address*/,
src /*memaddr*/,
32 /*bytes*/);
Physical mapping to three CoRAMs
1024x32 CoRAM
Slot 0
Slot 1
Slot 2
Memory
Starting word address x
0 1 2 3 4 5 6 7
0 1 2
3 4 5
seqID
seqID+1
32 bytes of data from memory shown as 32-bit words. Each word in memory is tagged with consecutive seqID values, which maps to a specific target CoRAM slot and address.
width=3
Target_Slot(seqID) = seqID % width
Target_Addr(seqID) = seqID / width

Figure 39: Mapping Functions to Support Arbitrary CoRAM Aspect Ratios.

6.4.4 Memory-to-CoRAM Datapath

The memory-to-CoRAM datapath is shown in Figure 40, which is responsible for receiving a b-wide block of data from the network and steering the data to the target CoRAMs. At this point in the datapath, it is already assumed that the CoRAM target slot (from 0 to $N-1$) and the 10-bit RAM address is known per s-wide word (see previous section). The front of the datapath first splits the b-wide block of data into b/s lanes, where each lane provides a dedicated datapath to $N \div b/s$ CoRAMs. The b-wide blocks of data that arrive from memory are first queued into a special input buffer called the Timeshift FIFO (to be explained further below). At the head of the FIFO, the b-sized blocks are separated into b/s lanes of data.

The target CoRAM slots at the head of each FIFO lane are fed into a conflict detector that determines which entries at the head of the FIFO are allowed to proceed down the pipeline. Conflicts occur when multiple words at the head of the FIFO are destined to the same CoRAM or to a conflicting lane. After the conflict detector evaluates, a *used bitmask* register is updated for each lane that is ready to proceed. When the entire bitmask is set, the FIFO is dequeued. At each clock

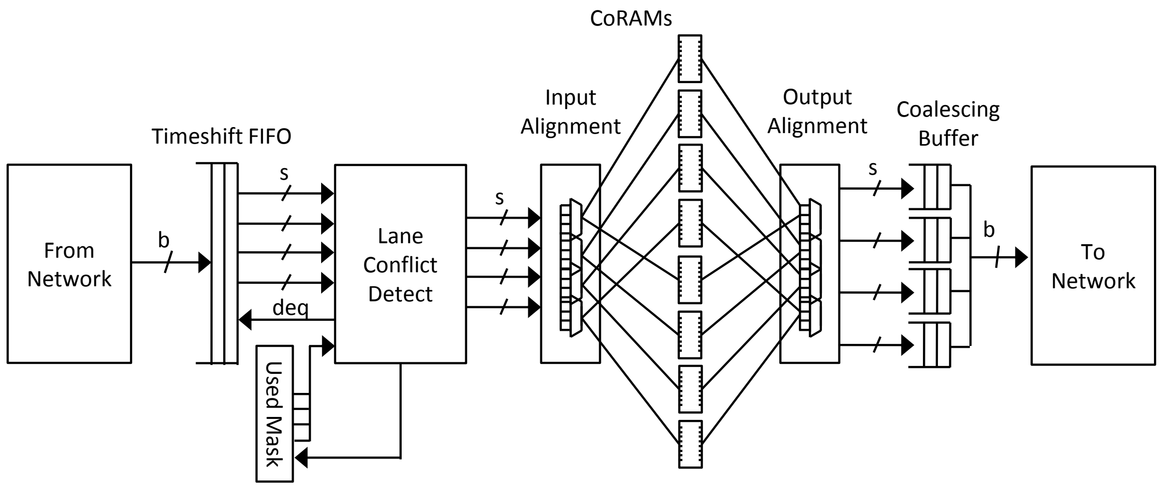


Figure 40: Memory-to-CoRAM Datapath.

cycle, an input alignment step takes place on the conflict-free words, which re-shuffles them into their respective lanes. After the alignment stage, the CoRAM address and data is presented to the respective CoRAMs. The reverse operation, i.e., reading from CoRAMs and writing to memory, is similar to writes, except that the input data is ignored, while the output data is read out, re-aligned, and collected into an output coalescing buffer to the network.

Eliminating Head-of-Line Blocking. The Timeshift FIFO (TFIFO) mentioned earlier is a core component of the cluster logic that addresses a specific problem related to head-of-line blocking (HOL). Figure 41 (left) illustrates a frequently-occurring case where two b-wide blocks of memory (A and B) are unable to read or write CoRAMs concurrently because all of the words of A are serialized and waiting at the head of the FIFO. The HOL blocking occurs when multiple control threads operate concurrently upon co-handles configured with narrow aspect ratios. A straightforward but costly approach to address this problem is to add extra FIFOs per lane as shown in Figure 41. The extra FIFOs would allow independent destination words to be exposed at the head of the FIFOs but at the cost of increased area and clock delay. Figure 42 illustrates how the TFIFO addresses this problem by re-ordering input data into diagonally-shifted departures. The TFIFO has the property of consuming only as much storage as that of a standard FIFO but can completely eliminate HOL blocking within the cluster.

TFIFO Implementation. A TFIFO is characterized by D, the depth, s, the width of individual

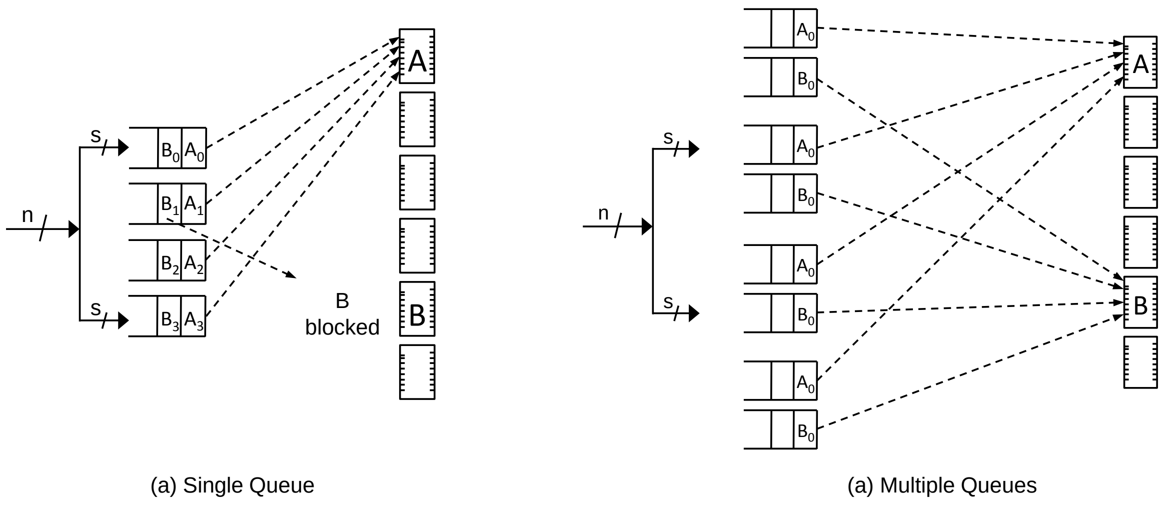


Figure 41: Single and Multi Memory-to-CoRAM Queues.

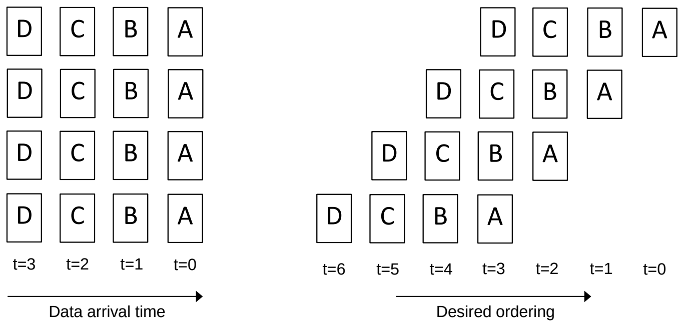


Figure 42: Time-shifted Schedule.

words, N, the total number of write ports, and C, a special parameter that determines the number of so-called **colors**. The idea of colors is to sort traffic into different allocated slots in the FIFO such that independent traffic can proceed without experiencing HOL blocking. For instance, if a control thread X issues a large stream of accesses to a single narrow CoRAM, any accesses by thread X should be assigned to only a single color, which will force the received packets into time-shifted delayed slots that minimize the opportunity to blocks others at the head of the TFIFO. If another thread Y is writing simultaneously, a separate color can be assigned to it, which will place it into a slot that allows transactions from both threads to proceed at full throughput.

A TFIFO maintains a throughput of is $s \times N$, while keeping a total storage of $D \times s \times N/8$ bytes. The TFIFO contains N independent memories of dimension $D \times s$. When employed in a single cluster, the parameter N is assigned to b/s, corresponding to the number of lanes in the memory-to-CoRAM datapath in Figure 40. Figure 43 illustrates the front- and back-end ports of

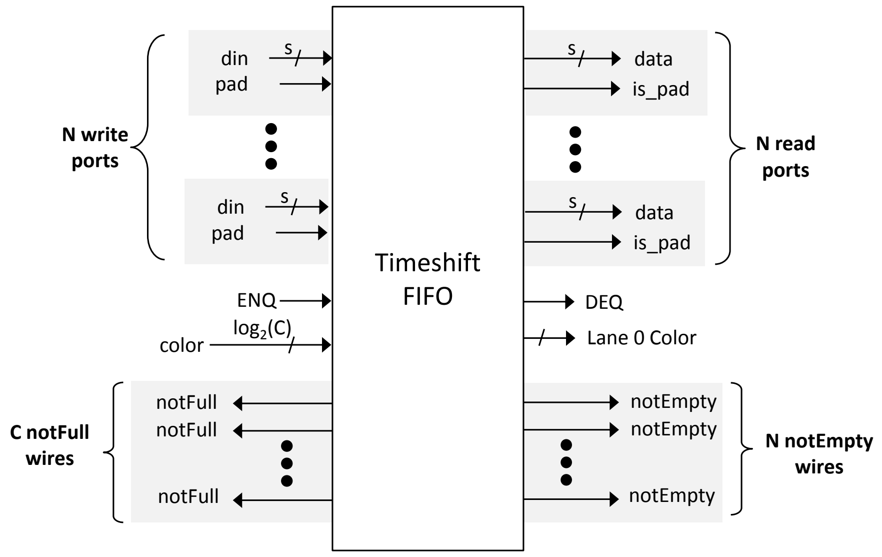


Figure 43: Front and Back Interfaces of Timeshift FIFO.

the TFIFO. A single ENQ signal writes an entire b-wide memory word on each clock cycle across b/s write ports. The TFIFO is configured with C colors, where D/C entries of the TFIFO are allocated to each color. When ENQ is enabled, an associated `color` input determines where the data becomes written. Internally, the TFIFO maintains $C \times N$ write pointers, where N is the total number of write ports and memories. Each write pointer is initialized to consecutive values starting from 0 to $C - 1$ in the first lane, from 1 to C in the second lane, 2 to $C + 1$ in the third lane, and so forth. The off-by-one shifting ensures that the s-sized words of the input data will be shifted diagonally across multiple internal memories. Any write pointers can only increment by C on each ENQ, which guarantees that no two pointers can write to the same location within a single lane. To read values from the TFIFO, a single read pointer advances one-by-one and must check against all head pointers of each color on each clock cycle to determine the emptiness of the TFIFO.

TFIFO Example. Figure 44 gives a concrete example of the TFIFO in action. In this example, two streams of data originating from separate control threads are written into two separate, 32-bit-wide CoRAMs (labeled by A and B, respectively). The TFIFO shown has $D = 8$ entries, $N = 2$ lanes, and $C = 2$ colors (gray and white). At time=0, the gray pointer is initialized to 1 while white is initialized to 0. When the first block of data A arrives from memory, its upper half is written into slot 0/lane 0, while the lower half is written into slot 1/lane 1 in a diagonal fashion. Note how slot 0/lane 1 is preinitialized with a padded value called 'X'. After the first write, both pointers increment by

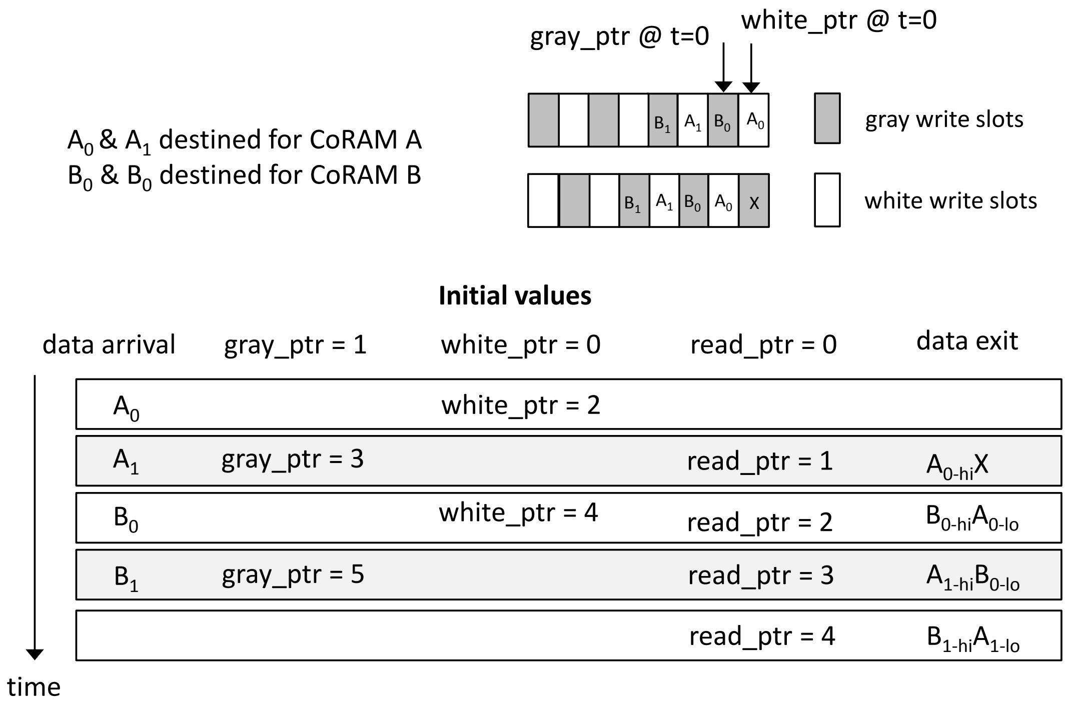


Figure 44: Example Timeline of Timeshift FIFO.

$C = 2$. The second word that arrives is also destined for A and will have its upper and lower halves written into slot 2/lane 0 and slot 3/lane 1, respectively. Note how this arrangement preserves slot 1/lane 0 and slot 2/lane 1 for B at time=2. As the read pointer advances, the upper and lower halves of A and B will appear mixed together, which allows concurrent writes to their respective CoRAMs.

Color Assignment. Within the TFIFO, each word of memory must be colored appropriately to minimize HOL blocking. To color, the cluster must know ahead of time which memory accesses are destined to the same CoRAM or to independent CoRAMs. In the case of wide co-handles, HOL blocking is a non-issue because all words can be written independently, and thus any color can be selected without problem. In the case of narrow co-handles, a first-order approximation that works effectively is to sort the traffic by taking the lower-order bits of a number associated with each unique control thread attached to the cluster. Note that this is the reason why the Thread-to-Cluster interface shown earlier in Table 11 includes a `width` field associated with each control action.

Avoiding Deadlock With Dummy Injections. Another problem that occurs within the TFIFO is that deadlock could occur if only a single stream actively written to a single CoRAM—in this case, all but one color is used within the TFIFO. To handle the deadlock problem, the cluster must ensure

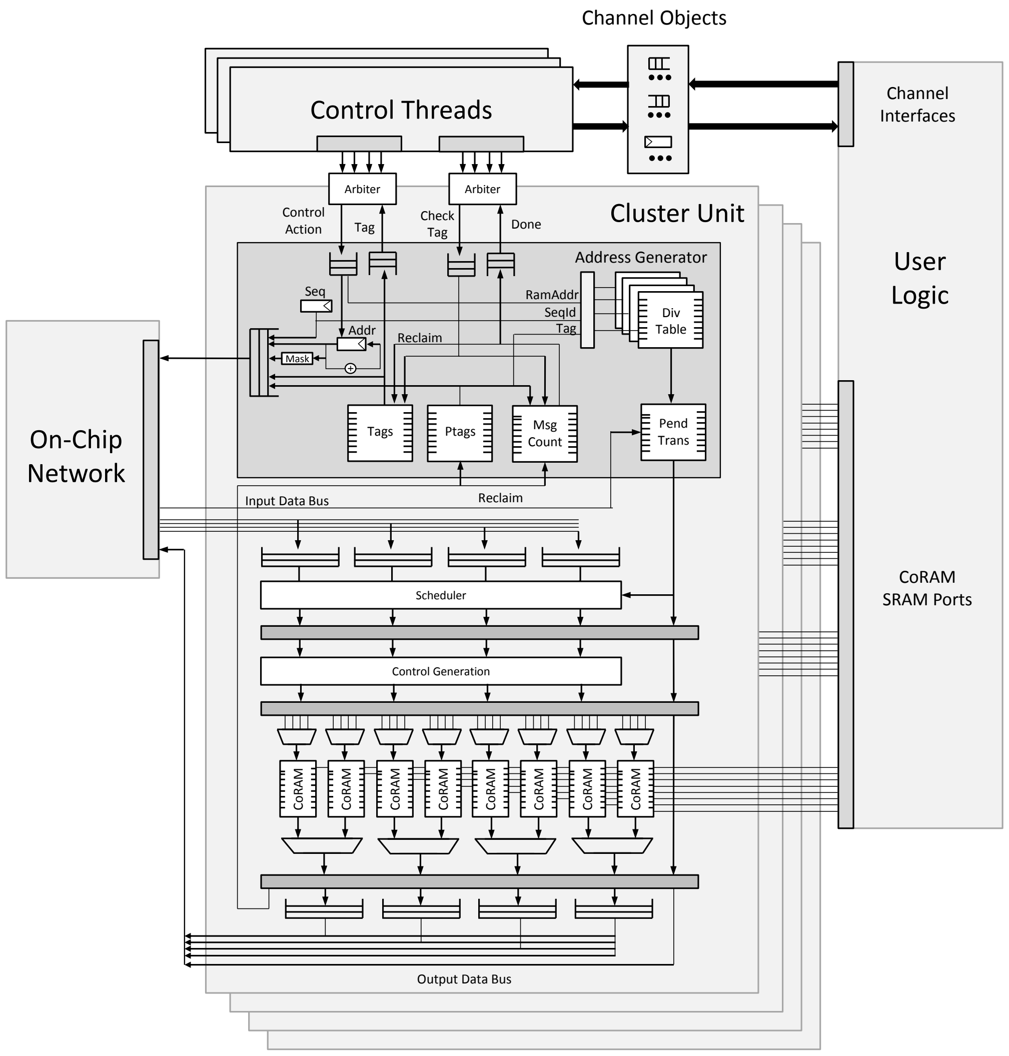


Figure 45: Complete View of CoRAM Microarchitecture.

that the word count for all colors is kept high enough within the TFIFO such that at any given clock cycle, an entry at the head of the FIFO is allowed to dequeue. To achieve this, Figure 43 shows an additional "pad" signal that allows the cluster logic to inject dummy words into the TFIFO to maintain a high enough word count per color. A simple strategy is to simply enqueue into the TFIFO with a random color whenever a valid data word is not present to be written. However, this approach can dramatically increase the latency of data responses because the TFIFO is kept full at almost all times. A much better strategy is to maintain level counters per color to track which color requires injection. We have found that this approach offers the best level of throughput and latency for the cluster.

Component	Description
Per-thread Request FIFOs	Small per-thread queues that accept control action requests
Central Request FIFOs	Central queue with arbiter that selects from a request queue
Ptags Freelist	Circular buffer with free ptags
Tag Queues	Circular buffer with free tags
Status Request FIFOs	Small per-thread queues that accept tag check requests
Memory Request FIFOs	Queue for all outgoing memory requests
Transaction Table	SRAM that stores logical transaction state
Input Timeshift FIFO	Front-end re-ordering queue for the memory-to-CoRAM datapath
Output Coalescing FIFO	Outgoing queue for CoRAM-to-memory transfers
Message Counters	Multi-ported memory that maintains a message count per tag
Transaction Locks	1-bit-wide table used to avoid race conditions
Map/slot tables	ROM tables used to map co-handles to CoRAMs
Division lookup table	ROM tables used to perform mapping functions
Reverse ptag-to-tag table	Small table that provides reverse mapping of tags to ptags

Table 12: Summary of Key Structures in a Single Cluster.

6.4.5 Cluster Summary

In this section, we have covered the salient features of the cluster-style microarchitecture. Figure 45 shows all of the major sub-components of the cluster in a single diagram, comprising the thread interface, the address generation unit, the memory-to-coram datapath, and the interfaces to surrounding external components. In summary, the cluster microarchitecture supports a number of key features:

- Highly concurrent throughput with multiple outstanding memory transactions and out-of-order message delivery.
- Arbitrary aspect ratios and compositions of multiple CoRAMs.
- Reduced head-of-line blocking using the specialized Timeshift FIFO.

Table 12 summarizes the key structures, components, and parameters of a single cluster. In the soft fabric implementation presented in this thesis, each of the components are configurable at compile-time and only consume the minimum resources needed to support a particular configuration. Recalling our soft CoRAM analysis from Section 6.3, it can be seen that even within just a single cluster, there can be significant overheads relative to just the baseline steering requirements. The next section describes the communication and memory components external to the clusters.

6.5 Network-on-Chip

In many digital systems, performance is often limited by communication or interconnection, and not necessarily from logic or memory. The CoRAM paradigm fundamentally requires a high-performance communication and data distribution mechanism that connects all of the clusters and memory ports. Based on current scaling trends, a future FPGA with CoRAM support would expect to have upwards of up to 128 clusters and up to 32 memory nodes by the 16nm technology node (see Table 10). A Network-on-Chip (NoC) is a promising approach that handles the critical role of communication and data distribution between clusters and the memory ports at the edges of the fabric. Daly et al. [30] states various requirement that dictates a network's parameters:

- Number of terminals
- Peak bandwidth of each terminal
- Average bandwidth of each terminal
- Required latency
- Message size
- Traffic pattern(s)
- Quality of service
- Reliability

The parameters of a desired CoRAM-to-memory interconnect for a hardened "large" FPGA configuration (see Table 10) are summarized in Table 13. The performance of an NoC must be provisioned to at least sustain the peak bandwidth of off-chip memory and to minimize contention and latency perceived by the application. It is important to note how the peak bandwidth of an NoC is provisioned significantly higher than the aggregate amount of memory bandwidth available in the system. Although one could theoretically divide the $102.4GB/s$ of bandwidth between all 64 clusters, the effect of *serialization* at each cluster port can have a detrimental effect on latency [30].

For a given application, a designer must work within technology constraints to implement the *topology*, *routing*, and *flow control* of the network. In CoRAM, the technology constraints can vary

Parameter	Value
Cluster Ports	64
Memory Ports	16
Peak bandwidth	922 GB/s
Average bandwidth	102.4 GB/s
Message latency	100ns
Message size	16-32B
Traffic patterns	arbitrary
Quality of service	none
Reliability	no message loss

Table 13: Parameters of Cluster-Memory Interconnection Network (Large)

depending on the implementation style, i.e., hard vs. soft. In a hard implementation of an NoC, each of the above parameters must be selected prior to fabrication and hardening within the FPGA. In an soft implementation of CoRAM, the NoC can be a flexible design choice at compile-time—an NoC can be configured with precisely the right topology, routing, and flow control needed to support the requirements of an application. Soft NoCs, however, are limited to slow fabric clock frequencies and must also share resources with the application. Even though a hardened NoC is "inflexible", it is likely to provide substantially improved throughput and latency relative to soft logic.

Topology Selection. The archetype of a network-on-chip comprises a collection of shared router nodes and wire-level links (or channels). The topology of a network-on-chip generally refers to the arrangement of such nodes and links. As noted by [30], a good topology exploits characteristics of the underlying medium to meet the bandwidth and latency requirements of an application while minimizing costs. A topology is often characterized by its *bisection bandwidth*, which is the bandwidth across equal parts when the network is segmented in half. Topology also determines the average distance between nodes (or the hop count), which directly relates to how much perceived latency nodes observe.

When developing either a hard or soft NoC design for CoRAM, the topology and routing is restricted to a few practical choices due to the placement of SRAMs in modern FPGAs today. As shown in Figure 46, the Block RAMs in the Xilinx Virtex-6 architecture are typically arranged in adjacent parallel columns separated by soft logic fabric. The arrangement of SRAMs in parallel columns is due to both design and manufacturing constraints—design-wise because the column arrangement simplifies multiple-SRAM composition, and manufacturing-wise due to the ease of

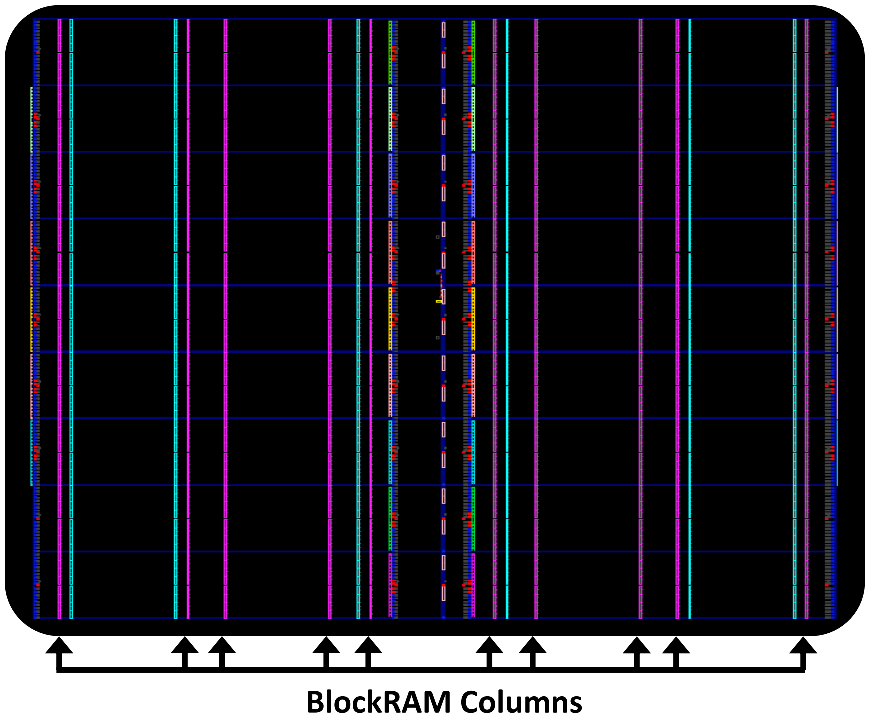


Figure 46: Virtex-6 LX760 Layout Schematic (Block RAM columns are highlighted in purple).

replicating identical fabric tiles to create different FPGA sub-families [4].

Assuming that the arrangement of SRAMs do not change in style in the future, this thesis only considers two well-studied topologies: the mesh and ring. A mesh topology is a natural fit for several reasons: (1) the spatial layout of physical RAMs and memory interfaces at the edges match well with the mesh topology, which would minimize wire distances between nodes, and (2) the mesh router only requires a 4x4 crossbar, which can be implemented very efficiently in a soft logic fabric design. The ring topology represents the lowest-cost design point for soft logic designs, which is important to determine a lower bound for implementing CoRAM using soft logic.

NoC: Hard vs. Soft. From the perspective of a fabric designed to support computing, a hardwired NoC offers significant advantages, especially if it reduces or eliminates the need for long-distance routing tracks. Under the CoRAM architectural paradigm, global bulk communications are restricted to between CoRAM-to-CoRAM or CoRAM-to-edge. Such a usage model would be better served by the high performance (bandwidth and latency) and the reduced power and energy from a dedicated hardwired NoC that connects the CoRAMs and the edge memory interfaces. With a hardwired NoC, it is also more cost-effective in area and energy to over-provision network bandwidth and latency to deliver robust performance across different applications. Chapters 7 and 8 will later demonstrate the benefits of a hard NoC context of real applications.

6.6 Memory Controllers and Caches

The edge memory subsystem comprises the hard memory controllers and caches that bridge the external main memory to the cluster subsystem. With memory I/O speeds rapidly outpacing the clock frequencies of fabric [28], conventional FPGAs have begun incorporating dedicated on-die memory controllers [113]. A memory controller is typically responsible for translating high-level application requests into row and column commands to the raw DRAM interface. Controllers are also responsible for issuing refresh commands to memory DIMMs.

Typical commodity DRAMs are optimized for large bulk transfers (e.g., $8 \times 8B$ bursts=64B) that correspond to common cache block sizes in modern general purpose processors [58]. A particular challenge with burst-oriented DRAM interfaces is the waste of bandwidth if the memory request sizes originating from the clusters are not exact multiples of the burst length (e.g., 64B). The introduction of caches at the boundaries of the fabric is a simple but effective solution to address bandwidth waste. A cache would retain the entire burst of DRAM data and allow subsequent requests to the same burst to access the data without further communication with the DRAM controller.

Figure 48 shows the internals of a multi-banked cache controller used in the implementation of the CoRAM memory subsystem implemented in this thesis. Each bank exposes an independent lookup stage connected to a network port and allows multiple in-flight transactions to the memory controller through tracking of Miss Status Holding Registers (MSHRs) [88]. On a cache miss, all of the MSHRs are looked up concurrently in a single clock cycle. If a matching tag is found, the new miss is merged with an existing outstanding miss—otherwise, a new MSHR is allocated and a request is issued to the DRAM controller.

6.7 Network-to-DRAM-Cache Interface

Figure 47 illustrates the connection between the edge cache and multiple ports of the network-on-chip. Typically, the cache-to-NoC link width provides a lower throughput than the actual raw DRAM interface (especially in a soft NoC design). To balance the system, the edge caches can be banked into multiple memory ports, with each port now exposed as an additional memory node on the NoC. Banking helps to increase parallelism and utilization at the memory controllers. The degree of banking must be kept high enough to rate-match the NoC link and the cache port as described by the following relation:

$$ports = \frac{8 \times f_{i/o}}{f_{noc} \times b \times 1/8} \times BankFactor$$

The relation assumes a JEDEC-standard DRAM interface with a 64-bit datapath between the FPGA and each DIMM managed by the memory controller [59]. The peak throughput of each DIMM is $f_{i/o} \times 8$ bytes per second. To balance the traffic, the throughput must at least be equal to $f_{noc} \times b \times 1/8$, where b is the link width of the network-on-chip. Given that the cache controller can also act as a filter, a $BankFactor$ variable adds additional ports and cache banks to ensure that the memory controller will be maximally utilized. In our experiments of Chapter 8, a bank factor of 2 was sufficient to achieve bandwidth-limited performance in our applications.

Address mapping. With multiple memory ports distributed along the network-on-chip, a mapping must exist between the request's address to the memory controller where the data resides. The most simplest approach to mapping is to block-interleave at a fixed granularity by masking out the lower-order bits of an address. Block-interleaving will uniformly distribute the traffic from clusters to the edge memory ports[1]. Other mapping schemes are also possible, which involve XOR hashing of higher- and lower-order bits to uniformly distribute the traffic [99]. In our experiments of Chapter 8, a block-interleaving scheme was found to be sufficient for our applications.

[1]The Convey HC-1 computer, for instance, blocks-interleave at a granularity of 32B across 16 independent DDR2 DIMMs [27]

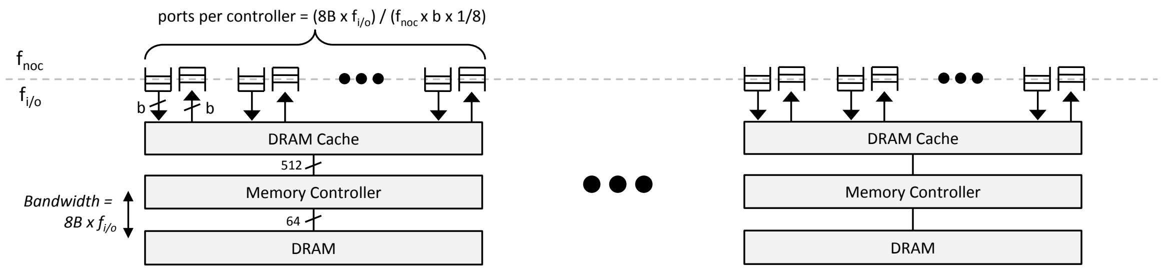


Figure 47: NoC-to-DRAM Interface.

6.8 Summary

In this chapter, we developed the archetype of the cluster-style memory subsystem from first principles. The central building block of the CoRAM memory subsystem is the cluster, which aggregates multiple embedded CoRAMs across a single network endpoint. The cluster serves as a central point for issuing requests, collecting responses, and fine-grained data steering and alignment. At the macro-scale level, a network-on-chip provides scalable bulk data distribution between the cluster endpoints and the multitude of distributed memory ports throughout the fabric. At the memory nodes, non-blocking caches bridge the memory controllers and DRAM to the network-on-chip. In Chapter 8, we will evaluate the merits of this style of microarchitecture for both hard and soft implementation targets.

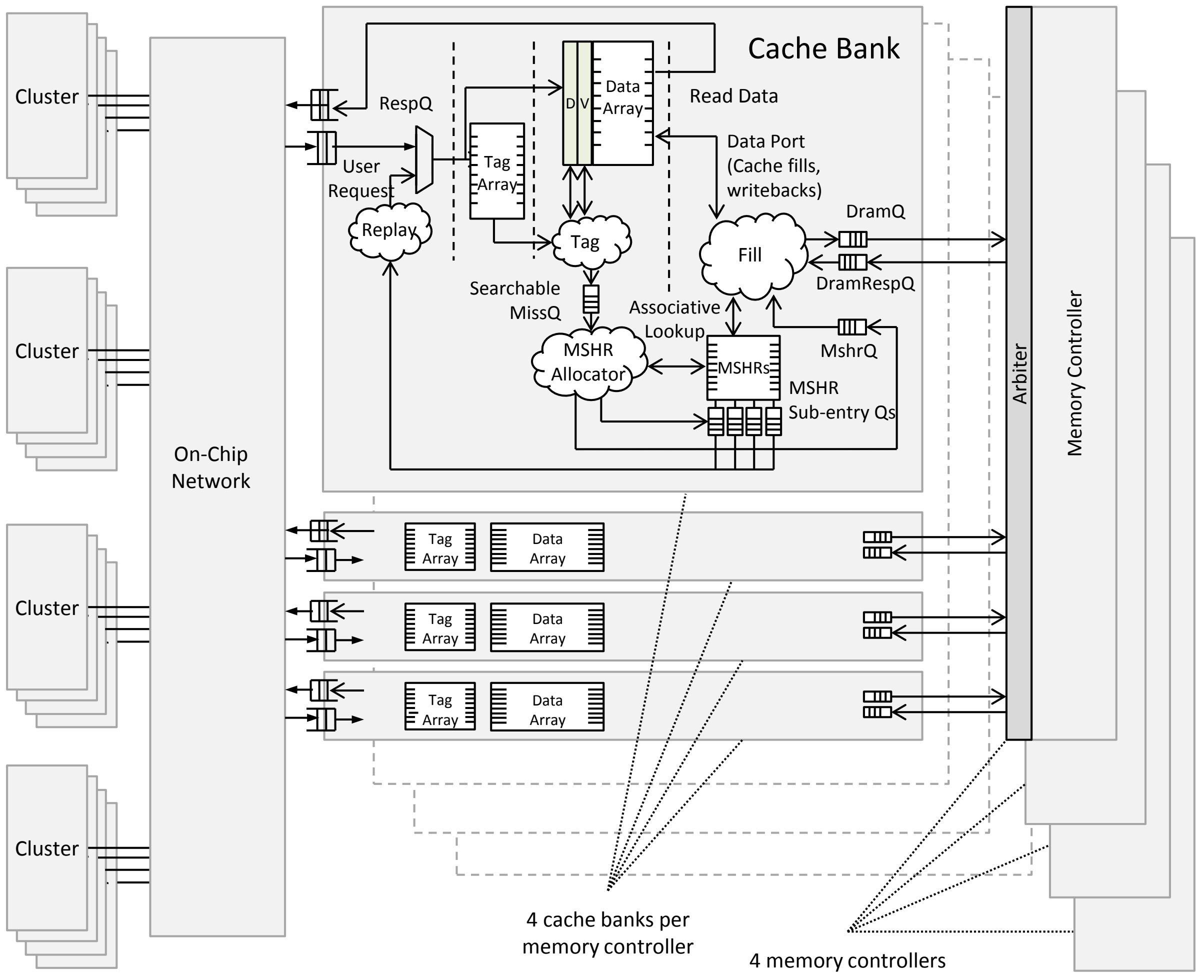


Figure 48: Memory Subsystem.

Chapter 7

Prototype

No way of thinking or doing, however ancient, can be trusted without proof.

Henry David Thoreau

The CoRAM memory subsystem from Chapter 6 has been fully implemented in synthesizable RTL and validated in Verilog simulations. The implementation along with the CoRAM Control Compiler (CORCC) from Chapter 4 is integrated into a unified tool called **Corflow**, which supports a stand-alone flow for compiling Cor-C applications into synthesizable RTL. The prototype serves several key objectives:

- To obtain accurate FPGA and ASIC synthesis results for evaluating and comparing implementation alternatives in Chapter 8.
- To carry out performance simulations of real applications with bit-level accuracy and for validation.
- To provide a testbed for application development and tuning.
- To demonstrate the plausibility of CoRAM as a practical abstraction for on-die FPGA-based memory management.

Section 7.1 describes the RTL implementation of the CoRAM memory subsystem. Section 7.2 describes a synthesizable network-on-chip generator called CONECT that enables exploration of

Parameter	Options	Description
Cluster Base RAM Width (s)	Any power-of-two width	Base CoRAM width (typically maps to BRAM width)
Memory Data Width (b)	Any multiple of $b > 0$, $b > s$	Cluster-to-NoC link width
Memory Address Width	32, 64	Virtual address space width (host-dependent)
Max Cluster Size	≥ 4	Number of CoRAMS in cluster
Max Request Size	Any multiple of $b > 0$	Maximum memory request size per control action
Max Threads	≥ 1	Number of thread-to-cluster interfaces
Num Tags	≥ 1	Total number of logical tags
Num Ptags	≥ 1	Total number of physical tags

Table 14: Customizable Parameters in a Single Cluster Implemented in Bluespec.

multiple network designs. Section 7.3 describes the edge memory cache controller designs. Section 7.4 gives an overview of the Corflow tool that integrates CORCC and the physical hardware and provides an automatic Cor-C-to-RTL development flow. Section 7.5 describes a Corflow backend that targets the ML605 FPGA prototype board.

7.1 Cluster Design

The cluster design of Chapter 6 was developed in 8300L of Bluespec System Verilog (BSV) [17] over a period of nine man-months. The Bluespec RTL is highly parameterized and targetable to both hard and soft logic implementations. The Bluespec language was selected due its features that greatly accelerated hardware development and validation. The strong type checking and rule-based semantics of BSV was instrumental in reducing the likelihood of design bugs, while the built-in static elaboration engine enabled highly parameterized and modular designs, allowing for rapid exploration of different designs. The salient parameters of the cluster RTL are listed in Table 14, which allow the user to configure desired parameters for a given application.

FPGA Design. A significant effort (3 man-months) was invested in optimizing the area and clock frequency of the soft cluster design on the Virtex-6 FPGA architecture [112]. The optimizations focused on reducing the area costs of various FIFOs, memories, and logic used for buffering and tracking the state of transactions. Particular attention was given towards the efficient use of LUTRAM-based FIFOs and BRAMs to implement the various required structures. BRAMs can be configured into multiple memory aspect ratios (e.g., 1024x36-bit to 16384x1-bit) and are typically useful for constructing deep- and wide-aspect ratio memories. A single Xilinx Virtex-6 LUT, on the other hand, can be used to construct a single, dual-ported 32x1-bit memory (LUTRAM). The LUTRAM is typically most efficient when implementing shallow memories. The division lookup tables described in Section 6.4.3, for example, requires thousands of entries, and were mapped into dual-ported BRAMs. Other objects such as buffers and state tracking structures were implemented judiciously in LUTRAMs. Some structures, such as the transaction message counters, required multiple concurrent write ports, which if implemented naively, could consume a significant amount of area (1-2KLUTs). Rather than implementing multiple write ports in logic, a banked multi-ported approach based on work by et al. was employed [68], which reduced the multi-ported register file's area by an order of magnitude.

ASIC Design. The same RTL developed for the soft cluster is adaptable to a hardened ASIC implementation. A hard implementation of the cluster must permanently fix the values of various parameters shown in Table 14 prior to fabrication. Further, the LUT- and BRAM-based memories used to implement buffering and state tracking must be replaced with standard SRAMs generated using a memory compiler or through custom design.

7.2 CONECT Network-on-Chip Generator

The network-on-chip for CoRAM employs a highly optimized packet-switched network-on-chip generator called the COnfigurable NEtwork Creation Tool (CONECT) [83]. The CONECT framework used in CoRAM is adapted from the original winning submission to the MEMOCODE 2011 design contest [83] (courtesy of Michael K. Papamichael). The goal was to devise a fast, parameterized, network-on-chip simulator, of which CONECT achieved by leveraging the rich, static elaboration features of Bluespec System Verilog and through extensive optimizations specific to the Virtex-6 FPGA architecture.

The heart of CONECT is a programmable, packet-switched router, which is configurable in VCs, ports, allocators, buffers, and routing tables, all which enable the generation of highly parameterized networks. CONECT implements an input-queued router that switches packets in a single clock cycle. Various networks of CONECT have been validated on the Xilinx ML605 board and through extensive Verilog simulations [83]. The set of parameters accepted by CONECT include:

- Pre-defined network topologies (mesh, ring, crossbar, torus, star, fully-connected, custom)
- Up to 1024 routers
- Up to 16 inputs and outputs per router
- Up to 8 virtual channels per router
- Configurable buffer depths > 2
- Configurable allocators (separable, monolithic)

CONECT and CoRAM. CONECT is used to generate NoCs for both soft and hard implementations of CoRAM. When synthesizing an implementation, CONECT is configured with a total number of nodes equal to the number of clusters and memory ports in the fabric. The CONECT router is configured with a flit width that matches the Cluster-to-NoC data link (typically 16B or 32B) plus the additional meta-fields associated with each memory request or acknowledgement (i.e., address, mask, is_write, tag, etc.). In the version of CONECT used in this thesis, all packets are switched in a single clock cycle (i.e., packet = flit).

7.3 Edge Memory Cache Controllers

In addition to the cluster design and CONECT, several cache controller designs were implemented in RTL to functionally simulate the edge memory caches that bridge the external memory ports (i.e., memory controllers) to the network-on-chip, as described in Chapter 6. The caches are intended to eliminate bandwidth waste by supporting large sequential transfers efficiently that may take place over multiple memory requests issued by the clusters.

7.4 Corflow

The Corflow tool combines the cluster RTL design, the CONECT network-on-chip generator, and the CoRAM Control Compiler (CORCC) from Chapter 4 into a single flow that converts Cor-C applications into functioning CoRAM designs. Corflow serves several objectives: (1) to generate soft RTL designs for programming Cor-C applications, (2) to generate non-existent, simulatable

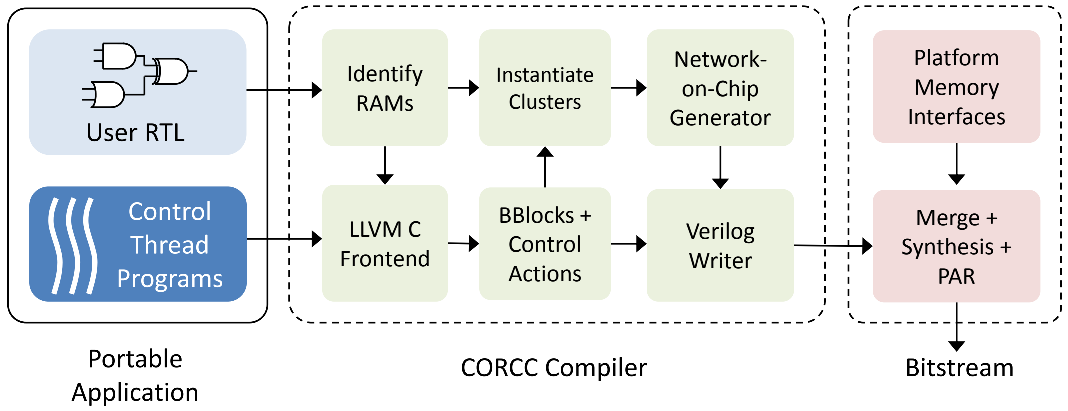


Figure 49: Corflow Method of Compilation.

FPGA designs for evaluation and comparison, and (3) to offer a proof-of-concept of the feasibility of having a software-managed memory abstraction for FPGA-based computing. The major components of Corflow tool have all been tested extensively with over 10 microbenchmarks and real applications. Over the course of the CoRAM project, two students new to the project were able to successfully develop new applications and microbenchmarks using Corflow.

Figure 49 illustrates the entire flow beginning from an application's source-level description down to the final configuration bits on the FPGA. The initial steps begin with translation of the Cor-C program into synthesizable RTL using CORCC. In conjunction with this step, the application core logic component (e.g., Verilog) is parsed separately to collect all elaborated instances of Cor-C objects such as embedded CoRAMs, channel FIFOs, and channel registers. Each object's Cor-C parameters (e.g., `THREAD`, `OBJECT_ID`) are linked with the co-handles extracted by the control threads. Following the identification of co-handles, Corflow instantiates clusters, an NoC, and a cache memory subsystem based on user-guide specifications. A post-processing step for the Verilog creates shadow ports that route and connect CoRAMs and channel objects nested within the core logic to the generated clusters and synthesized control threads. Figure 50 shows how embedded CoRAM and channel objects can exist at any level of a nested hierarchy and are wired automatically to the top-level design.

7.4.1 Command Line Parameters and Options

Invoking the Corflow tool requires passing several parameters as follows:

```
corflow -spec=<SPECFILE>
```

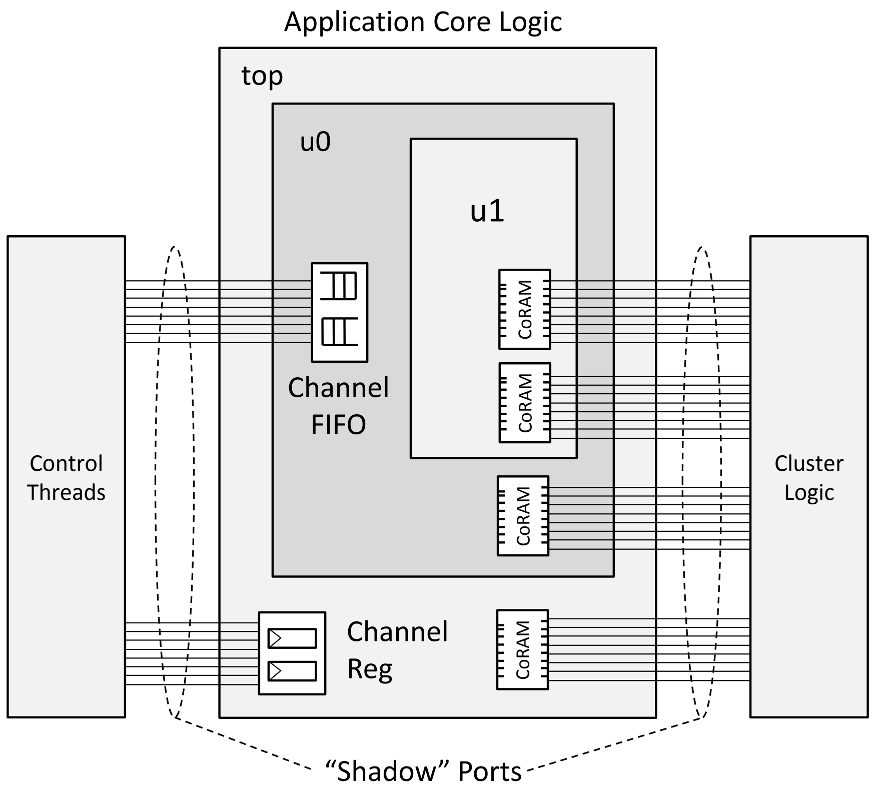


Figure 50: Creating Shadow Ports.

```
-vdir=<RTL_SOURCE_FILES>
-top=<TOP_MODULE>
-cfile=<CTRL_THREAD_FILE>
-mfile=<MACROS>
-bdir=<BUILD_DIR>
```

The `-vdir` flag specifies the directory containing files that implement the application's core logic (e.g., Verilog, VHDL), while `-cfile` points to the top-level control thread program that “wraps” the core logic. The specification parameter (`-spec`) takes as an input a plaintext file with key parameters listed in Table 15. When executed, the Corflow tool generates a workspace directory and invokes the Bluespec compiler to compile various backend components such as the cluster logic and NoC into synthesizable Verilog. In the generated design, each of the major sub-components of CoRAM reside in separate clock domains buffered by FIFO-based synchronizers: (1) core logic, (2) cluster/NoC, (3) control thread, and (4) I/O, DRAM, and Edge Caches. For soft CoRAM designs generated by Corflow, components 1, 2, and 3 share a common clock.

Parameter	**Description**
base_ram_width	Baseline CoRAM Data Width
memory_data_width	Cluster-to-NoC Link Width
memory_addr_width	Virtual address width
max_cluster_size	Maximum CoRAMs per Cluster
max_reqsize_bytes	Maximum Request Size in Memory Control Actions
num_tags	Number of Logical Transaction Tags per Cluster
num_ptags	Number of Physical Tags Per Cluster
num_mc	Number of Memory Controllers
ports_per_mc	Number of Ports and Cache Banks Per Memory Controller
flit_buffer_depth	Flit Buffer Depth in Network
mem_ileave_bytes	Memory port interleaving granularity
stack_depth	Maximum stack depth in control threads
debug_id_width	Width of debugging identifier
platform	Backend target for Corflow
dram_delay_cycles	Native DRAM delay in I/O clock cycles
dram_hex_file	Simulated DRAM file
simulate_cpi	Simulate Control Threads as Core (when set to nonzero)
sysclk_multiplier	NoC/Cluster clock multiplier relative to FPGA clock
tclk_multiplier	Control thread clock multiplier relative to FPGA clock
ioclk_multiplier	I/O clock multiplier relative to FPGA clock
fpga_clk_ghz	FPGA clock frequency
topology	Network-on-chip topology
group_by_instance	Cluster mapping policy
misses_per_bank	Maximum number of outstanding misses per cache bank
hard_cluster_cap	Simulate a fixed number of clusters in a hard CoRAM design

Table 15: Specification File in Corflow.

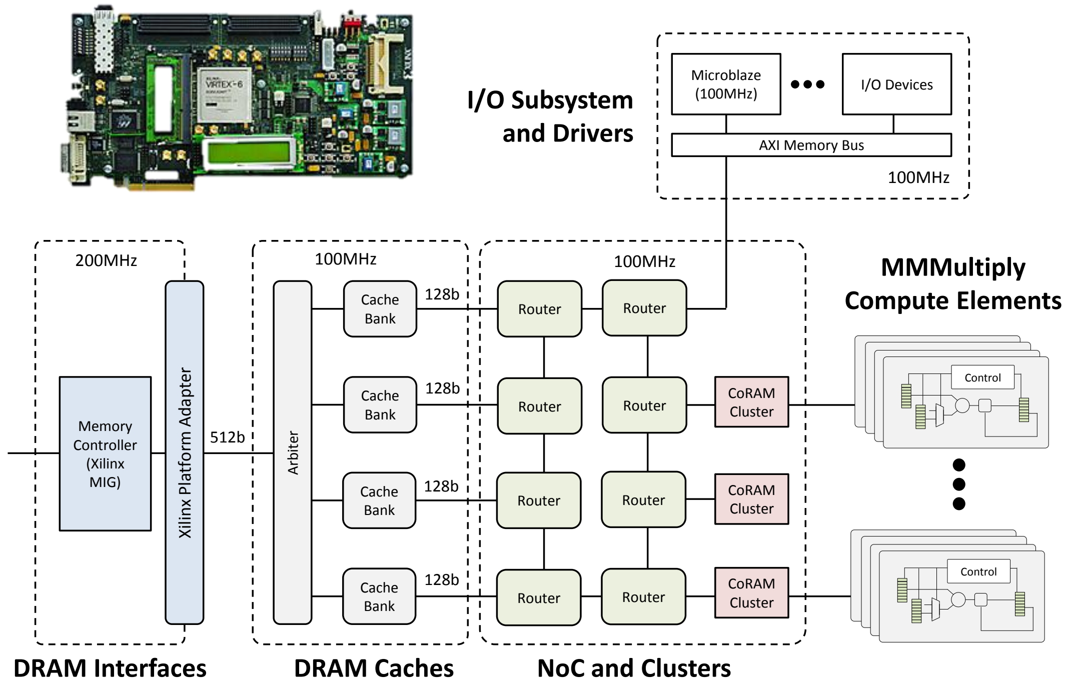


Figure 51: ML605 Prototype.

7.5 Xilinx ML605 Board

Corflow currently includes an experimental backend that targets any Cor-C applications to the Xilinx ML605 platform [110] (see Figure 51). The ML605 is a stand-alone FPGA board that hosts a single Virtex-6 LX240T FPGA. The FPGA is connected to a single DDR3 DIMM that is managed by a soft DDR3-400 memory controller generated by the Xilinx MIG software tool [111]. When generating a backend for Corflow, the specification file along with the Cor-C control thread program is used to automatically create a memory hierarchy comprising the network-on-chip, the clusters, and the core logic of application as shown in an example of Figure 51(bottom). In the ML605 backend, the edge memory cache controllers which are normally assumed to be hardened (as discussed in Chapter 6) are instantiated in soft logic. To interface the NoC generated by CONECT to the memory controller, a platform adapter was developed that translates memory commands received by the clusters into memory messages according to the memory request-response protocol employed by the memory controllers generated by MIG.

To interconnect all the components, Corflow generates two `pcores` (in Xilinx parlance), which are IP cores in the Xilinx EDK tool flow that allow easy insertion of modules into an existing

	Language	**Lines**	**Design Time**
Simulator	Bluespec and C	10000L	6 months
Cluster Design	Bluespec	8300L	8 months
Cache Subsystem	Bluespec	2200L	2 months
CORCC Compiler	C++	6000L	4 months
CONECT Tool ([83])	Bluespec	5300L	2 months
Microbenchmarks	Bluespec and Cor-C	1000L	6 months
Applications	Bluespec and Cor-C	5000L	12 months

Table 16: Corflow Design Statistics.

system-on-chip. The system-on-chip comprises a single Microblaze core connected to an ARM AXI bus with peripheral devices such as RS232. In a normal stand-alone system generated by Xilinx EDK, the memory controllers are directly attached to the AXI bus. Corflow instead introduces a bridge component that allows memory transactions on the AXI bus to traverse the generated NoC by CONECT in order to access the memory controller. This arrangement ensures that memory accesses between the Microblaze and the control threads are to the same shared address space.

7.5.1 Tool Release and Future Plans

The Corflow tool is being released to the public for evaluation and experimentation. By disseminating the infrastructure and source code, the Cor-C standard can serve as a starting point for more refined tools and an agreed-upon memory standard by the community. Table 16 presents a few statistics of the general infrastructure. There are plans in the future to create additional backends for Corflow, including support for the Convey HC-1 [27] and for Altera devices [11].

Chapter 8

Evaluation

A fool is a man who never tried an experiment in his life.

Erasmus Darwin

Our evaluation of the CoRAM architecture considers several key questions posed in this thesis: (1) What is the performance and efficiency of applications developed using control threads? (2) What are the relative merits of hard versus soft implementations of CoRAM in the cluster-style microarchitecture? and (3) How scalable and portable is CoRAM across current and future generations of FPGAs? The experiments and studies conducted in this chapter examine these questions across multiple dimensions:

- **Performance Characterization.** We characterize system throughput and latency as a function of varying CoRAM microarchitecture designs, ranging from soft to hard CoRAM, and from synthesized control threads to hardened microcontrollers.
- **Synthesis Characterization.** We characterize the area, frequency, and power of different CoRAM designs and quantify the gaps in efficiency between hard and soft CoRAM.
- **Application Performance, Efficiency, and Scalability.** We evaluate application performance and area efficiency across various hardware designs. We also explore application scalability across multiple generations of FPGAs.
- **CoRAM Versus Conventional.** We compare our results against idealized application implementations on conventional FPGAs.

The results from our studies support several key conclusions:

- For the applications we studied, CoRAM as an abstraction does not limit the performance potential of FPGA-based applications and can achieve performance and efficiency comparable, if not better, than manual-based approaches to memory management.
- Soft logic implementations of CoRAM incur high performance and area penalties stemming from low clock frequencies and resource-sharing between the user application and the infrastructure.
- Control threads, when synthesized into soft logic or implemented as hardened microcontrollers, do not limit the performance potential of the applications studied.
- The devised cluster-style microarchitecture from Chapter 6 for CoRAM scales well to the projected FPGA capacity and memory bandwidth in the 16nm technology node.

8.1 Experimental Setup

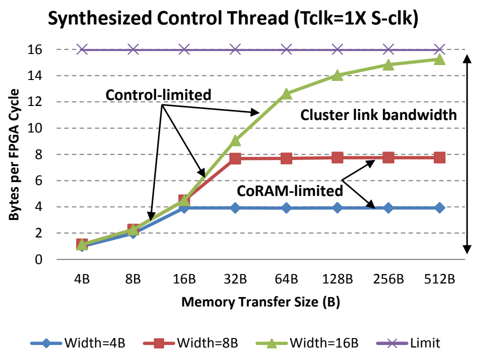
Performance Modeling. All performance results are collected from simulations of RTL generated from the Corflow tool as described in Chapter 7. Given an application or microbenchmark description in Cor-C along with a specification file, Corflow automatically generates a hardware implementation in synthesizable register-transfer level Verilog (see Chapters 4 and 7). Depending on the specification, the emitted RTL models two system styles: (1) the core logic of the application combined with soft logic clusters and a CONECT-generated NoC that targets conventional FPGAs (designated throughout this section as S-CoRAM), and (2) the core logic of the application hosted within soft logic fabric that instantiates hardened clusters and an NoC built into the fabric of the FPGA itself (designated as H-CoRAM). In all experiments, the core logic of all applications and microbenchmarks are pre-compiled into synthesizable Verilog using Bluespec v2011-06D [17]. The Cor-C specification is compiled by CORCC, which is automatically invoked by the Corflow tool.

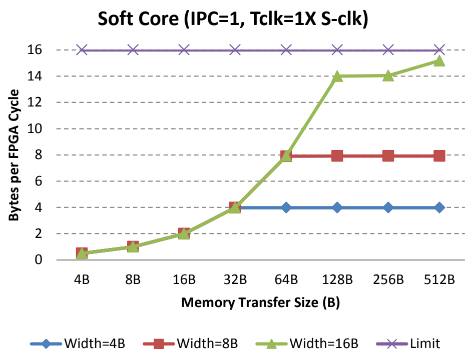
Terminology. In many experiments, we are interested in the performance and efficiency of S-CoRAM versus H-CoRAM. To model S-CoRAM in our Verilog simulations, a single FPGA clock is used for the core logic, the clusters, and the NoC, which is designated as **S-clock** throughout this section. In the H-CoRAM simulations, the clusters and NoC are simulated in a separate clock domain scaled to multiples of the S-Clock (designated as the **H-clock**). The implementation of control threads is independent of H- or S-CoRAM. In our experiments, control threads that are synthesized directly into soft finite state machines are designated as S-FSM. Control threads that execute on soft microprocessors cores are designated as S-Core, while hard cores are designated as H-Core. In all experiments, soft and hard cores are modeled as idealized processors that can execute LLVM instructions once per clock cycle (i.e., IPC=1). More details on microprocessor modeling can be found in Chapter 4. To collect performance statistics, we generate formatted traces using Verilog `$display` statements and perform trace-based post-processing with Python. Our tracing infrastructure allows us to collect detailed statistics on user-defined events such as counts, delays, and inter-arrival distributions.

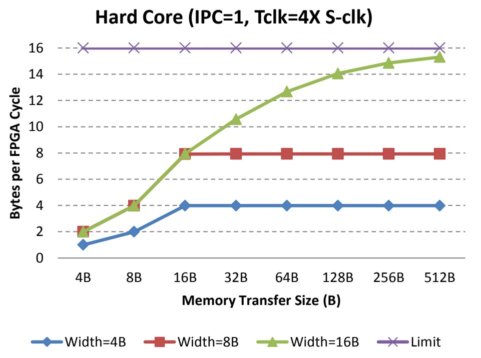
Area, Frequency, and Power Estimation. All FPGA area and performance results are collected using Xilinx XST 13.1 [7]. Unless otherwise noted, all FPGA synthesis experiments are configured to target the Xilinx Virtex-6 LX760 (speed grade 2). The Xilinx Power Estimator tool is used to calculate power consumption of the fabric. To attain ASIC area and power estimates, we use Synopsys Design Compiler vC-2009.06-SP5 configured with a commercial 65nm standard cell library; power estimates are based on DC's `report_power` option. Due to the lack of a memory compiler for the 65nm library, CACTI 6.5 [97] is used to estimate the area, power, and latency for all memories in ASIC designs.

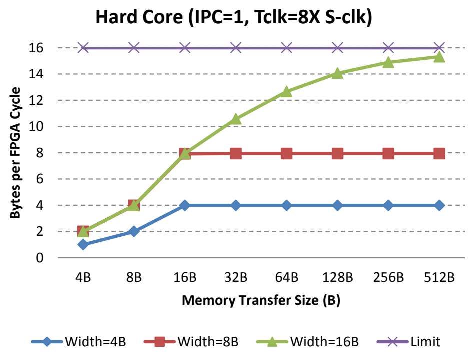
8.2 Microbenchmark Characterization

The characterizations of this section aim to answer the question: relative to a conventional memory subsystem on the FPGA, what is the penalty in raw throughput and latency (if any) when utilizing an implementation of the CoRAM high-level abstraction? When speaking of penalties, we refer to the latency of memory accesses relative to the raw DRAM access delay and the effective throughput between the fabric and the edge memory interfaces (i.e., bandwidth). Table 17 reports the multiple parameters of simple hardware systems generated by Corflow used in a simple set of experiments. The system is configured with only a single cluster and control thread. The network-on-chip is configured as a simple crossbar, while a single memory controller is configured with four









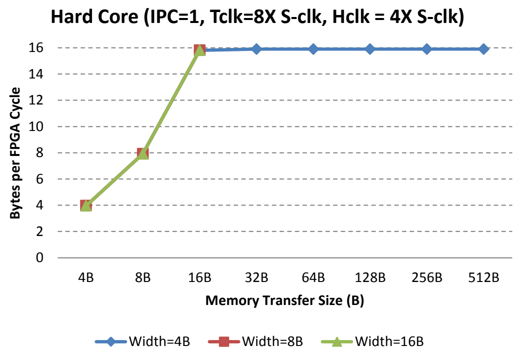


Figure 52: Single Cluster Throughput Characteristics.

cache ports. Both the NoC and the memory controllers are provisioned with sufficient bisection bandwidth and memory bandwidth, respectively, to avoid becoming the performance bottleneck.

8.2.1 Latency Characterization

In the first experiment, we measure the latency of a single memory access generated by the following control thread program:

```
cpi_tag tag = cpi_nb_write_ram(ramA, 0, 0, 4, CPI_INVALID_TAG);
cpi_wait(ramA, tag);
```

Component	Parameter
Base CoRAM width	32b
Clusters	1
Control Threads	1
Cluster link width	128
Network	Crossbar
DRAM delay	60ns
Memory controllers	1
Memory ports	4
Memory I/O speed	1.6GHz
Cluster + NoC	Soft-200MHz, Hard-1.6GHz
Control	Soft synthesized FSM-200MHz
	Hard core-800MHz (CPI=1)

Table 17: Configuration Used in Single Cluster Characterization.

Table 18 reports the cycle-by-cycle timeline of a memory-to-cluster memory access generated by the code above. Each column reflects different configurations, beginning with S-CoRAM/S-FSM to H-CoRAM/H-Core. Events are reported in FPGA cycles (assuming 200MHz) and absolute time (ns). As Table 18 reports, the delay of a single control action can vary considerably depending on the implementation. In S-CoRAM/S-FSM, the total round-trip delay between the moment a thread issues a non-blocking control action to when the thread unblocks is 52 FPGA cycles (170ns), compared to the raw DRAM delay of about 12 FPGA clock cycles.

As shown, S-CoRAM introduces a significant overhead relative to the raw DRAM latency due to the various added stages of the cluster logic, the network-on-chip, the buffers, and various arbitration stages. As will be discussed in Section 8.4, the increased latency can have a detrimental effect on latency-sensitive applications. Column 2 shows how the delay can be mitigated by hardening the cluster and NoC logic and operating it at multiples of the FPGA clock rate. H-CoRAM operating at 1.6GHz, for example, can reduce the latency to 26 FPGA cycles with the remaining overhead due to the control thread and DRAM latency. Column 3 shows that the latency can be reduced even further to 19 cycles (95ns) with a hardened microcontroller (H-Core) that also operates at 1.6GHz. As shown in Section 8.4, H-CoRAM is sufficient to allow memory-intensive applications such as SpMV to operate at peak performance potential.

Event	Soft-200MHz		Hard-1.6GHz		H-1.6GHz, T-1.6GHz	
	Cycle	Time(ns)	Cycle	Time(ns)	Cycle	Time(ns)
Thread issues memory control action	0	0	0	0	0	0
Cluster receives request	3	15	2	10	0	0
Cluster issues memory request to network	4	20	2	10	1	5
Memory message exits network	6	30	2	10	1	5
Edge cache lookup (miss)	7	35	2	10	1	5
Request issued to DRAM controller	8	40	2	10	1	5
DRAM controller responds	20	100	16	80	14	70
Cache bank begins replay	21	105	16	80	15	75
Response arrives from network	25	125	16	80	15	75
First word written to CoRAM	34	170	16	80	17	85
Thread detects completion	52	260	26	130	19	95

Table 18: Timeline of Single Unloaded Access to Memory.

8.2.2 Bandwidth Characterization

In the next experiment, we measure the data throughput between the CoRAM clusters and the edge memory interfaces. The objective is to characterize bottlenecks in the memory subsystem introduced by the cluster-style microarchitecture and control threads. The simple microbenchmark below performs a series of bulk memory transfers through repeated non-blocking memory control actions in a loop. A single tag is used to coalesce all the control actions for a single test at the end of a loop as described in Chapter 4.

```
cpi_hand ramA = cpi_get_rams(W, true /*scatter-gather*/, ...);
cpi_tag tag = CPI_INVALID_TAG;
for(int i=0; i < 8192; i+= BLOCK_BYTES) {
    tag = cpi_nb_write_ram(ramA, 0, 0, BLOCK_BYTES, tag);
}
cpi_wait(ramA, tag);
```

Figure 52 shows the memory-to-cluster throughput as a function of different implementations and with varying `W` and `BLOCK_BYTES` values. The two parameters, `W` and `BLOCK_BYTES`, define the width of a co-handle and the byte size of the memory transfer, respectively. The `W` parameter is defined as a multiple of the base CoRAM width, which is 32b in our example. $W = 2$, for instance, would represent a co-handle with two CoRAMs combined with an aspect ratio of 1024x64b. For all of the graphs shown in Figure 52, the x-axis plots the memory transfer size in bytes, while the y-axis plots the data throughput of the cluster generated by the control thread. Note that the cluster is limited to a peak throughput of 16B/cycle due to the link width between the cluster and the associated network port.

Discussion. Each graph in Figure 52 shows several curves with different values of `W`, each representing different logical aspect ratios of co-handles. There are several notable operating regimes in Figure 52. When memory transfer sizes are small, the throughput of the cluster becomes **control-limited** due to the overhead of executing each control action within the loop by the control thread. In the case of Figure 52a, regardless of the co-handle width, a control thread is only able to sustain a throughput of 1B/cycle when issuing a continuous stream of 4 bytes per control action, which translates to the initiation of a new control action once every four FPGA clock cycles. This limitation is due multiple states needed to serially setup and perform a single memory transaction (see Chapter 4). As the transfer sizes increase, the throughput transitions from becoming control-limited to **port-limited**. In the curve where W=1, the throughput approaches the maximum rate of 4B per clock cycle because only a single 32b-wide embedded CoRAM is being accessed in the co-handle. As W increases to 4, the throughput becomes **link-limited** at 16B/cycle.

Figure 52b shows the effect of replacing the synthesized control thread with an S-Core of IPC=1. The control-limited throughput now decreases to one control action per 8 clock cycles—requiring even larger memory transfer sizes to amortize the cost of a single control action. Figures 52c and 52d shows that the loss in throughput can be reclaimed with an H-core that operates at multiples of the FPGA clock rate. Note that at H-clock = 4X S-clock, the maximum throughput is attained and any increases in the H-clock are no longer beneficial with the cluster logic now becoming the bottleneck. Figure 52e shows the result if both the cluster and the control threads were operating at a clock rate 4-8X faster than the FPGA clock rate itself. In this case, even for small co-handle widths and small access sizes, the throughput immediately approaches saturation.

8.2.3 Summary

Our simple characterization experiments show that a soft implementation of CoRAM can introduce significant overheads in latency (2-3X the raw DRAM delay) and throughput. Our experiments show that applications writers cannot be oblivious to the microarchitectural details of CoRAM in order to extract performance from the memory system. In a cluster-style microarchitecture, application writers must be aware of the three operating regimes of a cluster: (1) control-limited, (2) port-limited, and (3) link saturation. Operating in a control- or port-limited regime is suboptimal and would typically require tuning on the part of the application developer to reach link saturation. Achieving link saturation can also be achieved by attaching multiple threads to a cluster to perform concurrent memory transfers that increase overall utilization.

In the soft implementation of control threads using CORCC, small memory accesses can also introduce high overheads due to the multiple clock cycles it takes to issue a single control action. However, as shown in our experiments, hardened implementations of CoRAM can easily close the gap between CoRAM and a raw interface to DRAM. Hardening the cluster and NoC logic introduces reduces latency and increases throughput that eliminates the cluster bottleneck from the perspective of the FPGA fabric. Hardening the thread logic as a microcontroller also eliminates the serialization of small accesses and co-handle sizes. The next section will evaluate the effectiveness of CoRAM in real applications.

8.3 Synthesis Characterization

The next set of experiments we perform characterize the area, power, and frequency of the various building blocks used to implement CoRAM.

Cluster Characterization. Columns 2 and 3 of Table 19 show the FPGA area costs for a single cluster assuming both *soft* and *hard* configurations. Note that the estimates only reflect the cost of logic and buffers and do not include the embedded CoRAMs themselves. On a Virtex-6 LX760 FPGA, a *soft* cluster occupies about 1.04% of the total area (5KLUTs), comparable to 4X the area of a minimally-configured microblaze processor [3] or about one-half the area of a soft logic DDR3 memory controller [112]. The design achieves a clock frequency of about 150MHz. The bottom rows of Table 19 further shows the FPGA power consumption of a single cluster if utilized at 50% activity factor.

Column 3 of Table 19 illustrates the significant reductions in area and improvement in performance when synthesizing the same cluster configuration to 65nm standard cells. In absolute die area, the hard cluster consumes $0.738mm^2$, which would displace about 244 LUTs if the soft logic

	Cluster		Mesh router	
	Soft (V6,40nm)	Hard (65nm)	Soft	Hard
LUTs	4962	-	6002	-
FFs	2326	-	1144	-
40nm Virtex-6 FPGA Frequency	150MHz	-	160MHz	-
65nm Hard Frequency	-	840 MHz	-	610 MHz
65nm Hard Logic Die Area	-	0.17 mm^2	-	0.068 mm^2
65nm Hard SRAM Die Area	-	0.568 mm^2	-	0.232 mm^2
65nm Hard Die Area	15 mm^2	0.738 mm^2	18.1 mm^2	0.3 mm^2
Hard Die Area (λ^2)	14.2B	0.69B	17.1B	0.284B
65nm Hard Logic Power	-	81 mW/GHz	-	48.4 mW/GHz
65nm Hard Dynamic SRAM Power	-	64.5 mW/GHz	-	28.24 mW/GHz
65nm Hard Static SRAM Power	-	22.6 mW	-	7.8 mW
40nm Soft Dynamic Power (LX760)	950 mW/GHz	-	970 mW/GHz	-
40nm Soft Static Power (LX760)	4.44 W	-	4.44 W	-

Table 19: Area and Power Characteristics of Single Cluster and Mesh Router.

area were traded for a hard CoRAM cluster[1]. Further, the hardened cluster would achieve a clock rate of 840MHz without any ASIC-specific tuning, about 5X faster than the soft cluster design. Note also how the power consumption is reduced by nearly an order of magnitude between hard versus soft.

Mesh Router Characterization. Columns 4 and 5 of Table 19 show the synthesis results for a 2-VC mesh router generated by the CONECT framework (described in Chapter 7). The mesh router is configured with 5 input queues with a depth of 32 entries per buffer. In soft logic, the router consumes an area of 6KLUTs, about 5X the cost of a minimally configured microblaze core [3]. The bulk of the costs are due to the buffers, allocators, and internal crossbar. Column 5 of Table 19 shows the synthesis results for the equivalent mesh router synthesized for the 65nm standard cell. As expected, the results show that the single mesh router achieves over an order-of-magnitude improvement in both area and power consumption relative to soft logic.

Control Threads. Table 20 shows the FPGA synthesis estimates for several control threads used in our applications. For reference, the synthesized control threads are compared against a soft Microblaze processor core configured with 4kB caches and with area-optimized parameters [3]. As can be seen, the majority of the control threads consume less area than a soft Microblaze core, with

[1]The estimated LUT area was attained through physical die area measurements of Virtex devices. The estimate for equivalent die area for a single LUT includes the programmable I/O, interconnect, on-die DSPs, and BlockRAMs.

	Area (LUTs)	**Flip-flops**	**BRAMs**	**Fmax (MHz)**
Microblaze (4kB caches, minimum-area-cfg)	1210	973	4	161
Latency Microbenchmark	158	90	0	344
Throughput Microbenchmark	155	118	0	345
Matrix Matrix Multiplication	2581	2802	0	192
SpMV 'row' FIFO thread	544	523	0	204
SpMV 'val' FIFO thread	729	556	0	201
SpMV 'col' FIFO thread	729	556	0	201
SpMV 'y' FIFO thread	699	685	0	269
SpMV 'x' cache thread	242	316	0	354

Table 20: Control Thread Synthesis Results.

the exception of the MMM control thread. Note how the control threads also achieve comparable frequencies to the Microblaze. The synthesis results for the control threads should further be treated conservatively considering the few optimizations that have been applied when developing the C-to-RTL generator in CORCC from Chapter 4. In CORCC, all basic blocks of the LLVM intermediate code are automatically unrolled into physical hardware blocks, resulting in maximum area consumption. In practice, state-of-the-art HLS tools can reduce the amount of resources needed by adding constraints and scheduling.

8.3.1 Comparing Area and Power Between Hard vs. Soft CoRAM

In this section, we project total area and power costs between hard versus soft CoRAM implementations based on the results listed in Table 19. All area and power estimates are standardized across a variety of FPGA configurations shown in Table 21. Each of the configurations progressively increase in LUT density and memory bandwidth, in accordance with ITRS technology predictions [57] and calibrated according to commercial FPGA technology trends discussed in Chapter 2. The first configuration, *small*, is based on a state-of-the-art FPGA with similar characteristics to a Xilinx Virtex-6 LX760 [112]. Each of the subsequent configurations double in LUT density relative to the previous generations in accordance to scaling predictions. For each of the design points listed in Table 21, we project both the area and power of adding hardened clusters and a network-on-chip into the FPGA fabric. Table 21 shows the parameters of the cluster memory subsystem and the network-on-chip for each of the configurations. In the case of hard logic, a maximum cluster size of 64 was selected, while the cluster-to-NoC link was configured to 16B.

		Small	**Medium**	**Large**	**Future**
	Technology	45nm	32nm	22nm	16nm
	Supply Voltage	1.0V	0.94V	0.88V	0.76V
	6-input LUTs (K)	500	1000	2000	4000
	FFs (K)	1000	2000	4000	8000
	Fabric Frequency (MHz)	200	200	225	250
	DRAM Interfaces	2 x DDR3-1600	4 x DDR3-1600	4 x DDR4-3200	8 x DDR4-3200
	DRAM Bandwidth (GB/s)	25.6	51.2	102.4	204.8
	4kB CoRAMs	1024	2048	4096	8192
Soft Configurations	Max Cluster/NoC clock	0.2GHz	0.2GHz	0.225GHz	0.25GHz
	Cluster/NoC link width	128	128	128	128
	# Clusters	Variable	Variable	Variable	Variable
	# Cache Banks	4	8	16	32
	MSHRs per bank	16	16	16	16
	# Nodes	Variable	Variable	Variable	Variable
	Topologies	Ring, Mesh, Xbar	Ring, Mesh, Xbar	Ring, Mesh, Xbar	Ring, Mesh, Xbar
	CoRAMs/Cluster	Variable	Variable	Variable	Variable
Hard Configurations	Max Cluster/NoC Clock	0.8GHz	0.8GHz	0.9GHz	1GHz
	Cluster/NoC link width	128	128	128	128
	# Clusters	16	32	64	128
	# Cache Banks	4	8	16	32
	MSHRs per bank	16	16	16	16
	# Nodes	20	40	80	160
	Topologies	Ring, Mesh	Ring, Mesh	Ring, Mesh	Ring, Mesh
	CoRAMs/Cluster	64	64	64	64
Area	Fabric Die Area (λ^2)	1430B	2860B	5720B	11440B
	Hard Cluster Area (λ^2)	11	22	45	89
	Hard NoC Area (λ^2)	14	27	36	73
	Soft CoRAM Area (%)	73.5%	73.5%	54.3%	54.3%
	Hard CoRAM Die Ovhd.	1.7%	1.7%	1.4%	1.4%
Power	65nm Hard Cluster Pwr @ 0.8GHz	2.2	4.4	8.9	17.8
	65nm Hard NoC Pwr @ 0.8GHz (W)	3.3	6.6	8.8	17.7
	Tech-normalized CoRAM Power (W)	3.2	4.2	4.4	5.5
	Peak Fabric Dyn. Power (W)	50	63	76	82
	Worst-Case CoRAM Power Ovhd.	6%	7%	6%	7%

Table 21: FPGA System Parameters with CoRAM Support.

	Small	**Medium**	**Large**	**Future**
Technology	45nm	32nm	22nm	16nm
6-input LUTs (K)	500	1000	2000	4000
FFs (K)	1000	2000	4000	8000
Fabric Frequency (MHz)	200	200	225	250
DRAM Interfaces	2 x DDR3-1600	4 x DDR3-1600	4 x DDR4-3200	8 x DDR4-3200
DRAM Bandwidth (GB/s)	25.6	51.2	102.4	204.8
4kB CoRAMs	1024	2048	4096	8192
Bscholes Kernel Parameters	1,4,8 PEs	1,8,16 PEs	1,16,32 PEs	1,32,64 PEs
	1,4,8 clusters	1,8,16 clusters	1,16,32 clusters	1,32,64 clusters
	2,8,16 threads	2,16,32 threads	2,32,64 threads	2,64,128 threads
MMM Kernel Parameters	1,2 cores	2,4 cores	4,8 cores	8,16 cores
	64,128 PEs	128,256 PEs	256,512 PEs	512,1024 PEs
	6,12 clusters	12,24 clusters	24,48 clusters	48,96 clusters
	1,2 thread	2,4 threads	4,8 threads	8,16 threads
SpMV Kernel Parameters	16 PEs	32 PEs	64 PEs	128 PEs
	16 clusters	32 clusters	64 clusters	128 clusters
	48 threads	96 threads	192 threads	384 threads

Table 22: Mapping MMM and SpMV to Various FPGA Configurations.

Table 21 shows the relative power and area overheads of introducing hard CoRAM into a conventional fabric. For reference, the table also shows the same hard configuration implemented as soft logic (note that this should not be interpreted as a fair comparison because in a soft version of CoRAM, the application only instantiates what is needed, as discussed later in Section 8.4). As can be seen, both the hard clusters and the network-on-chip altogether introduce a modest increase in die area ($< 2\%$) and worst-case overhead in peak power relative to the baseline FPGA fabric ($< 7\%$). Although the results are generated approximately based on Design Compiler, there is sufficient evidence that a feasible, practical microarchitectural design space exists, with additional headroom for optimization. The synthesized RTL reported in Table 21 is highly tuned for FPGA-based fabrics and not tuned for standard cells. For instance, our FPGA-optimized mesh router consumes relatively high area and power compared to a state-of-the-art design in the same technology node [61]. Nevertheless, our results show that even with less optimized designs, our implementations can achieve modest overheads in power and area relative to the baseline fabric.

8.4 Application Evaluation

This section presents a detailed architectural evaluation of the three FPGA-based applications developed in this thesis (see Chapter 5): Matrix Matrix Multiplication (MMM), Black-Scholes (Bsc-

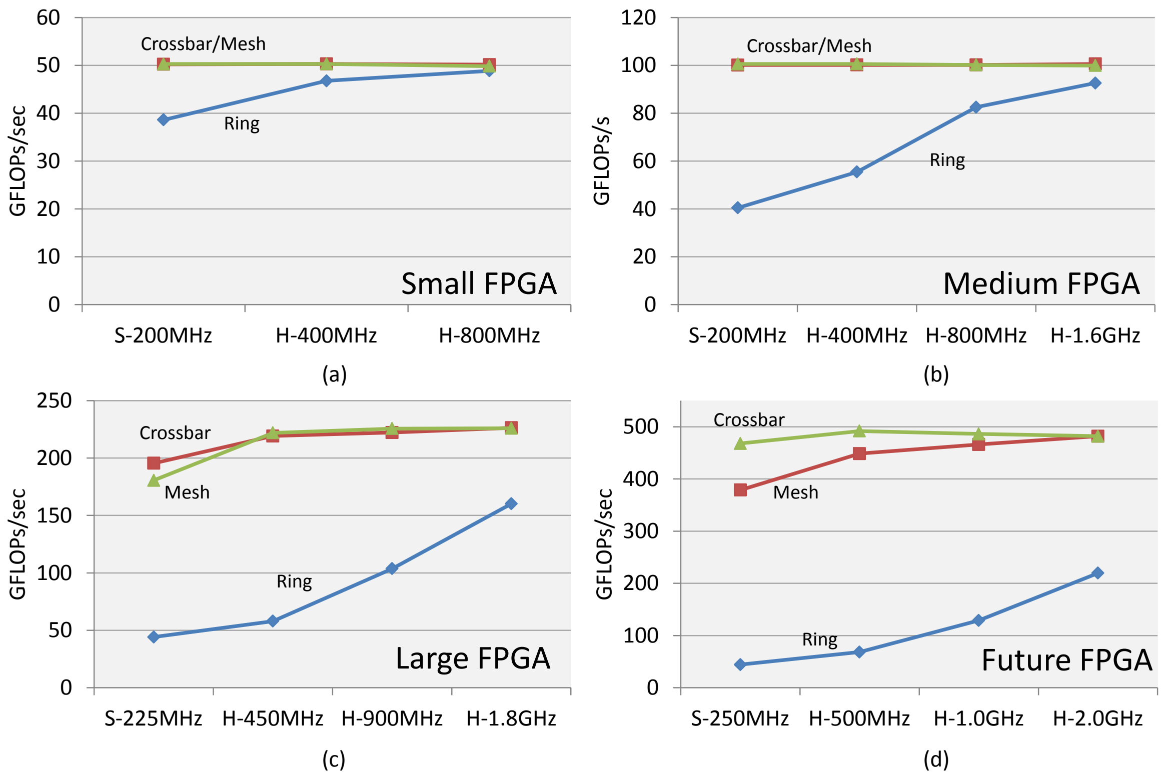


Figure 53: MMM Performance Trends.

holes), and Sparse Matrix-Vector Multiplication (SpMV). In our performance studies, we compare the applications across multiple dimensions: (1) the multiple FPGA configurations shown in Table 22, (2) network topology, and (3) soft versus hard logic implementations of the cluster microarchitecture. The network topologies we consider in the studies are: (1) the bidirectional ring, (2) the 2D mesh, and (3) the crossbar. The control threads used in our experiments are synthesized directly to FSMs using CORCC (see Chapter 4). The architectural simulations for S-CoRAM are configured with an optimistic clock frequency of 200MHz for the soft cluster and NoC designs, assuming that further increases in the soft clock rate can be achieved with additional tuning and floorplanning relative to the synthesis results presented in Section 8.3.

8.4.1 Matrix-Matrix Multiplication

The MMM kernel we measure is scaled to each FPGA reported in Table 22. The MMM kernel is subdivided into multiple cores, where each core is a single double-buffered kernel with 64 PEs as discussed in Chapter 5. Each core employs a total of four clusters managed by a single control

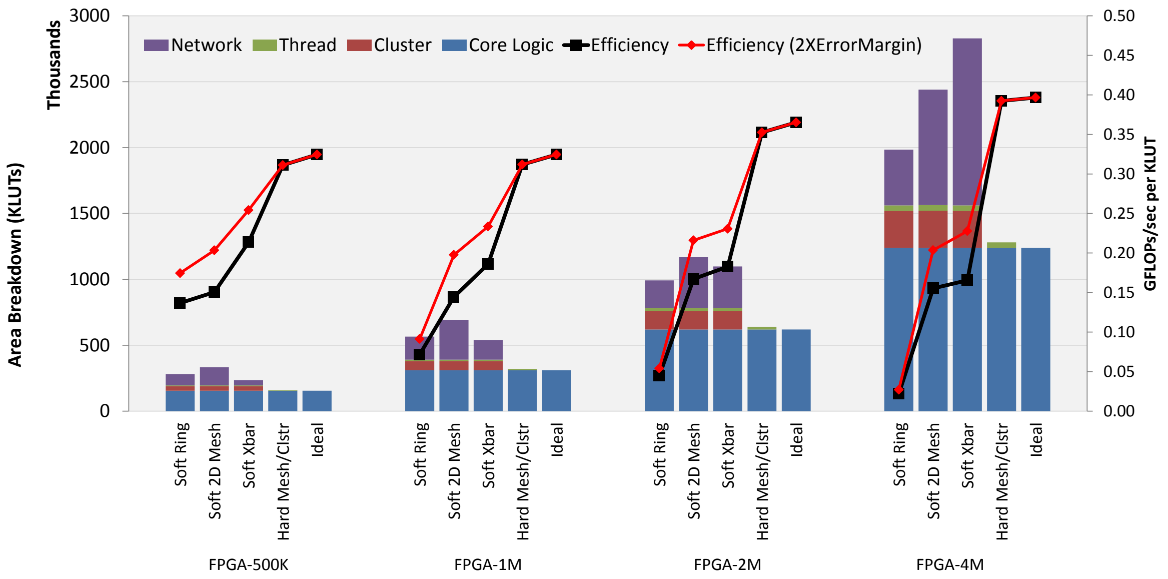


Figure 54: GEMM Area and Efficiency Trends.

thread program. The four clusters each correspond to the embedded CoRAMs of the `A`, `B`, `C0` and `C1` sub-blocks as described in Chapter 5.1. The cores cooperate together to solve a single large matrix problem and execute asynchronously with respect to other cores. Each of the cores' control threads are pre-programmed at compile-time to operate on disjoint sub-blocks of C (where each sub-block is a 64x64 square). The blocked MMM kernel we measure is double-buffered and concurrently performs a phase of computation while reading the data for the next phase and writing back data from the previous phase. The performance measurements we take are based on steady-state throughput (GFLOPs/sec) of the MMM kernel, including the non-overlapped time spent waiting on memory. We measure a single iteration of computation.

Performance. Figure 53 shows the performance trends in MMM across multiple FPGA design generations. In all the graphs shown in Figure 53, points along the x-axis are labeled by the CoRAM implementation style, where $S-Freq$ refers to S-CoRAM operating at frequency $Freq$, and $H-Freq$ refers to H-CoRAM. Beginning with the *small FPGA* in Figure 53a, designs based on the crossbar and the mesh achieve peak compute-bound performance across S- and H-CoRAM. The only design that does not achieve peak performance is S-CoRAM ring, which has the lowest bisection bandwidth of 12.8GB/sec (which is exceeded by the available memory bandwidth). The S-CoRAM ring suffers from increased contention and latency, which translates to memory stall times that cannot be completely overlapped by computation in the MMM kernel. This leads to a 26% degradation in performance in S-CoRAM. The hardened ring network at H-400MHz, however, is doubled in bisection bandwidth and able to achieve within 4% of peak performance.

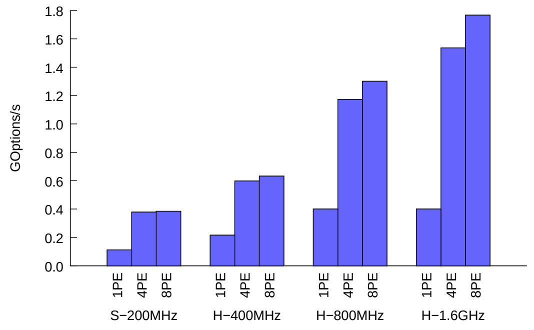
In subsequent FPGA designs shown in Figures 53b, 53c, and 53d, the gap between the ring and the crossbar/mesh increases due to the doubling of nodes each successive generation. Further, even aggressively scaled implementations of the ring (e.g., H-2.0GHz in Figure 53d) have difficulty achieving peak performance potential. A notable trend across all the design points is that the mesh closely tracks the performance of a full crossbar, suggesting that a scalable mesh design can replace a centralized crossbar in physical implementations. Both the crossbar and the mesh with lower clock rates (200-250MHz) in Figure 53c and 53d begin to show slight degradations in performance. However, across all design points employing either the mesh or the ring, H-CoRAM operating at 2X or higher the clock rate of S-CoRAM is sufficient to achieve peak performance potential.

Area Efficiency. At first glance, it would appear that for the mesh and crossbar, S-CoRAM is comparable in performance to the H-CoRAM across the design points listed in Figure 53. Peak performance alone, however, cannot be used to make a comparison. A more important figure of merit for FPGAs is the performance normalized to area. Figure 54 shows the area breakdown of various sub-components for all the design points, comprising the core logic, the cluster, the NoC, and the control threads. Within each FPGA category, the soft designs are placed side-by-side to an implementation of H-CoRAM where the clock rate is 4X of S-CoRAM. The right axis of Figure 54 takes the measured performance of each design point and normalizes it to KLUTs (reported in log scale). Note that the estimates do not factor in the increased die area as a result of hardening the clusters and the NoC—however, as shown earlier in Chapter 7, the area and power overhead of adding dedicated CoRAM support is modest, requiring less than a 2% increase in the die area. Once factoring in area, it is apparent from Figure 54 that a significant efficiency gap of at least 2X exists between the best possible S-CoRAM implementation versus H-CoRAM. The soft logic designs incur a high area penalty from the soft NoC and the cluster logic. In all of the designs, the NoC and clusters contribute the largest sources of overhead in the soft designs relative to the core logic. It should be noted that the control threads contribute a relatively small overhead relative to the core logic (less than 4% area relative to the core logic). The *2XErrorMargin* bar further shows that even if the soft logic area overhead were **halved**, a substantial gap in efficiency would still exist between

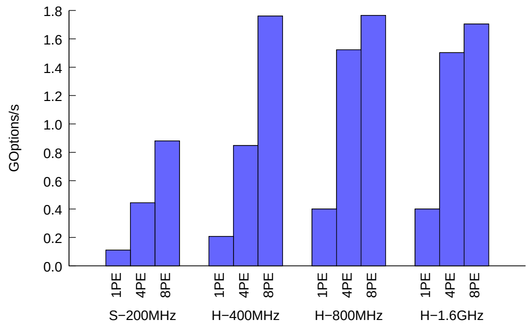
H- and S-CoRAM.

An important question that merits discussion is how the efficiency results would compare against a manual approach to MMM on a conventional FPGA. The right-most design point of each category in Figure 54 shows the hypothetical efficiency of an **idealized** MMM core that can operate at peak performance potential and incurs no overhead relative to the core logic. In this case, the core logic would only constitute the control logic, the floating point datapath, and the neighbor exchange datapath. As can be seen, H-CoRAM achieves comparable if not equal efficiency to the idealized version of MMM. The results from MMM suggests that H-CoRAM is effective at matching the performance and efficiency of manual approaches to memory management on the FPGA.

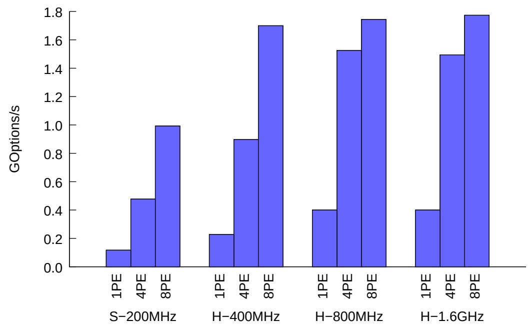
8.4.2 Black-Scholes



(a) Ring Topology



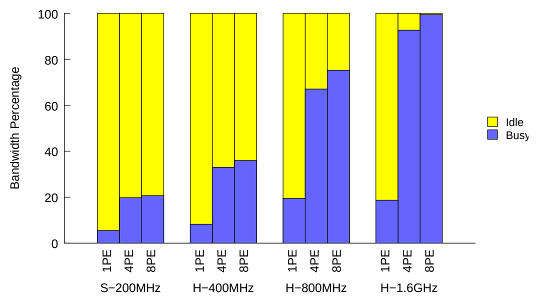
(b) Mesh Topology



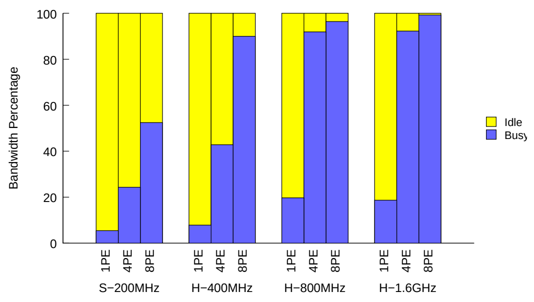
(c) Crossbar

Figure 55: Black-Scholes Performance Across Network Topologies (small FPGA).

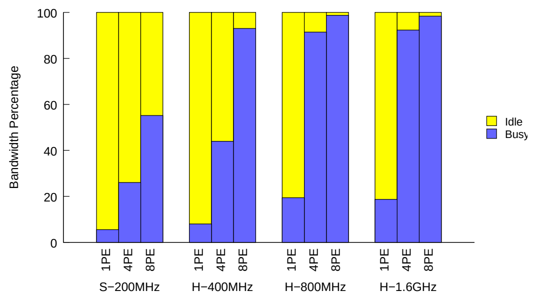
The Black-Scholes kernel differs from MMM in that it is more likely to be a bandwidth-bound application rather than compute-bound. Figure 57 shows the performance scaling trends for the *small FPGA* across multiple networks and the number of instantiated PEs. Like we saw in MMM,



(a) Ring Topology



(b) Mesh Topology



(c) Crossbar

Figure 56: Black-Scholes Bandwidth Usage Across Network Topologies (small FPGA).

the ring network experiences difficulty scaling unless the clock rate of H-CoRAM is about 8X of S-CoRAM. Across most design points, the performance saturates when transitioning from 4 instantiated PEs to 8. For the same set of design points, Figure 56 shows the off-chip bandwidth consumption corresponding to each performance point. As can be seen, the saturation at 8 PEs is due to the Bscholes kernel reaching bandwidth-limited performance.

Figure 57 illustrate trends in scalability across all FPGA design points. As we observed before in MMM, both the mesh and the crossbar closely track in performance across design generations. A surprising result we found in Bscholes is that performance did not improve monotonically when increasing the clock frequency of H-CoRAM. For example, in Figure 57b, the performance of the crossbar design drops slightly by 2% when doubling the clock frequency from H-800MHz to H-1.6GHz. These performance drops appeared to be an artifact of slight fluctuations in the miss rates of the edge memory caches situated between the network-on-chip and the memory controllers. The increased clock in some cases caused re-ordering of miss references to the cache controllers, resulting in different effective performances. However, in most cases, the fluctuations are relatively

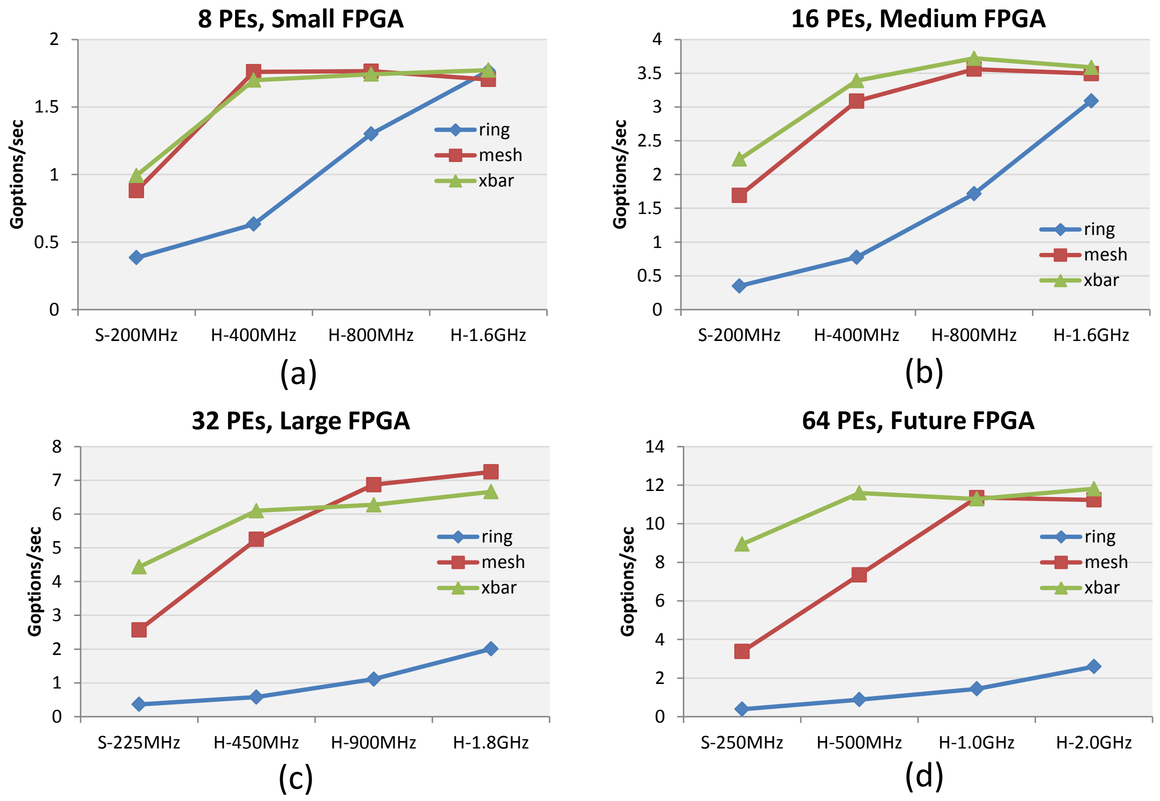


Figure 57: Black-Scholes Performance Trends.

negligible compared to the general performance trends when transitioning from S-CoRAM to H-CoRAM and between network topologies.

An unusual property of the Bscholes kernel is that the data structures are laid out in non-aligned boundaries of 20B per option. Non-aligned accesses have a tendency to be more inefficient because the cluster logic may have a link width that does not exactly align with the data size. This often requires multiple issued requests with certain bytes masked off to perform the whole transfer. Further, the cluster logic may experience inefficiencies in the memory-to-CoRAM datapath due to the none-powers-of-two writes of bulk data to individual target CoRAMs (see Chapter 6 for more details). The inefficiencies caused by non-aligned accesses are highlighted in Figure 57c. In S-CoRAM for instance, even a single PE at H-400MHz is unable to achieve full throughput due to inefficient usage of the cluster logic. Note, however, that these inefficiencies are offsetted by increasing the clock rate sufficiently such that application performance does not become impacted (i.e., H-800MHz or above).

Area Efficiency. Figure 58 shows Bscholes performance normalized to area. Like we saw in

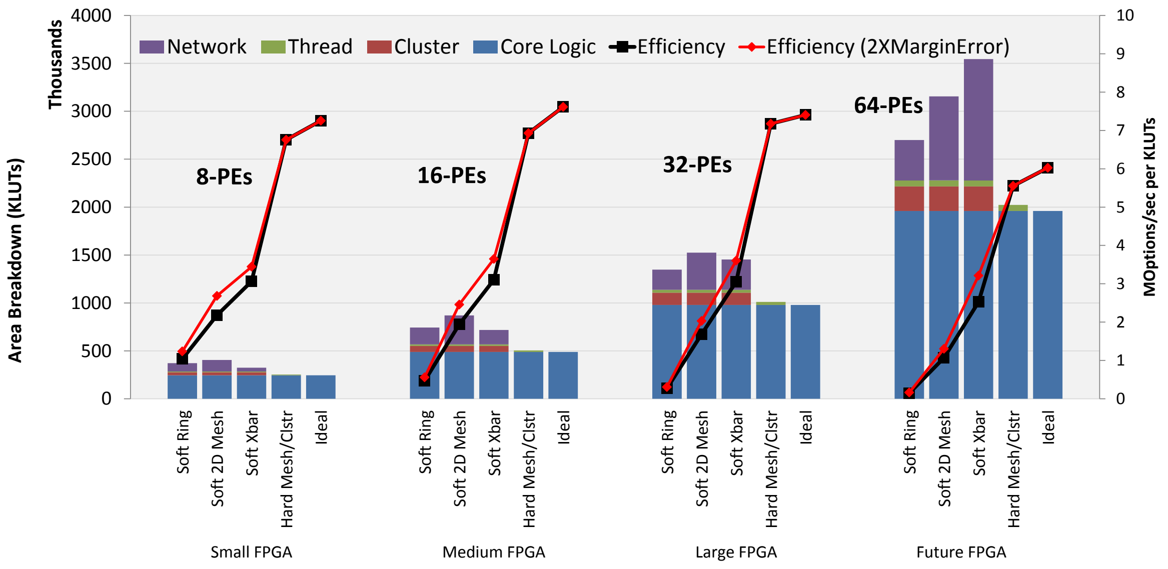


Figure 58: Black-Scholes Area and Efficiency Trends.

MMM, a substantial gap in efficiency exists between S- and H-CoRAM implementations. In a small FPGA, the S-CoRAM with the soft crossbar achieves an efficiency of about 3.1MOptions per KLUTs, while H-CoRAM with a mesh achieves about 6.8MOptions per KLUTs. For comparison, the **ideal** efficiency only considers the cost of the compute logic alone and achieves an efficiency of 7.0MOptions per KLUTs. Like we saw before, the H-CoRAM designs achieve comparable performance efficiency relative to an ideal design, and are about 2X or better relative to S-CoRAM. In a bandwidth-limited application like Bscholes, increased area efficiency is still beneficial because reconfigurable logic can be freed up to perform other tasks— the required die size of the device can be reduced.

8.4.3 Sparse Matrix-Vector Multiplication

The SpMV results we present in this section are measured across a collection of Compressed Sparse Row (CSR)-formatted inputs. Table 23 shows the inputs and their descriptions, which are drawn from the University of Florida Sparse Matrix Collection [33]. Our implementation of SpMV comprises a parameterized linear array of PEs, with each PE capable of fetching, computing, and writing the outcomes of independent rows as discussed in Chapter 5. Table 22 shows the number of PEs scaled across multiple FPGA configurations. Each PE is mapped to a single cluster configured with a 16B link to the NoC. Four control threads are attached to each cluster, corresponding to the

Input	Rows	Non-zeros	Description
cant	62451	2034917	FEM cantilever
consph	83334	3046907	FEM concentric spheres
cop20k	99843	1362087	Accelerator cavity design
mac_econ_fwd500	206500	1273389	Macroeconomic model
mc2depi	525825	2100225	2D Markov model of epidemic
pdb1HYS	36417	2190591	Protein data bank 1HYS
qcd5_4	49152	1916928	quark propagators (QCD/LGT)
scircuit	170998	958936	Motorola Circuit Simulation
shipsec1	140874	3977139	FEM Ship section / detail
webbase1M	1000005	3105536	Web connectivity matrix

Table 23: Matrices Used in Sparse Matrix-Vector Multiplication Experiments.

memory personalities associated with each PE: the x-cache, the address FIFO, the value FIFO, and the output FIFO. Our measurements are collected from 10,000 cycles of steady-state throughput after a warmup period of 10,000 cycles.

Figure 59 shows the performance of SpMV reported in GOPS/s averaged across all inputs in Table 23. Similar to MMM and Bscholes, both the mesh and crossbar designs achieve comparable performance, while the ring network suffers from poor scalability in the larger FPGAs.

H-CoRAM versus S-CoRAM. Figures 60 show the per-input performances for the *small FPGA* across various S- and H-CoRAM configurations. The x-axis is sorted from the left beginning with a soft logic implementation with an S-clock of 200MHz followed by hard implementations with gradually increasing H-clocks. The right-most design point represents **idealized** performance, which is modeled by an idealistic hard cluster/crossbar design with control threads operating with an H-clock of 6.4GHz (i.e., no further performance improvements were observed by increasing H-clock further).

The results show that the performance of SpMV is highly input-dependent. For well-behaved inputs such as *cant* and *consph*, the ideal performance is quickly achieved when H-clock $\approx 2\times$ S-clock. At this operating point, the effective x-cache miss latency approaches the raw DRAM latency and any further improvements in the clock amount to diminishing returns in performance. For other more memory-intensive inputs such as *web* and *cop20k*, the H-clock must reach about $4\times$ the S-clock to reach near-ideal performance. This is due to the fact that such inputs exhibit much more irregular memory access patterns, which results in increased miss rates in the x-cache.

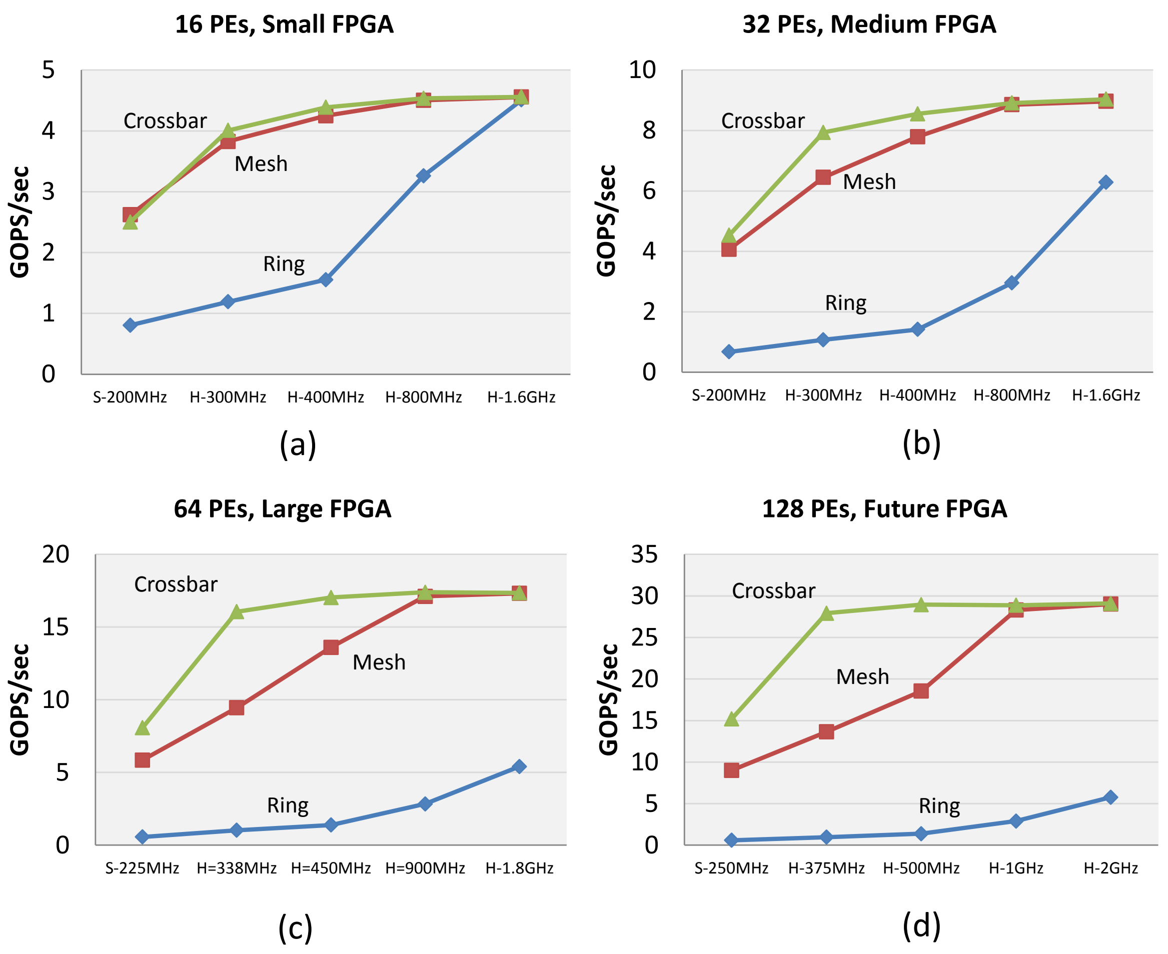
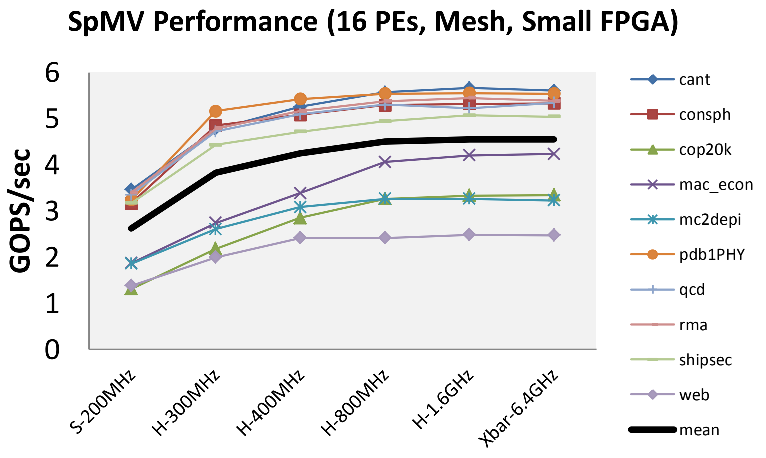
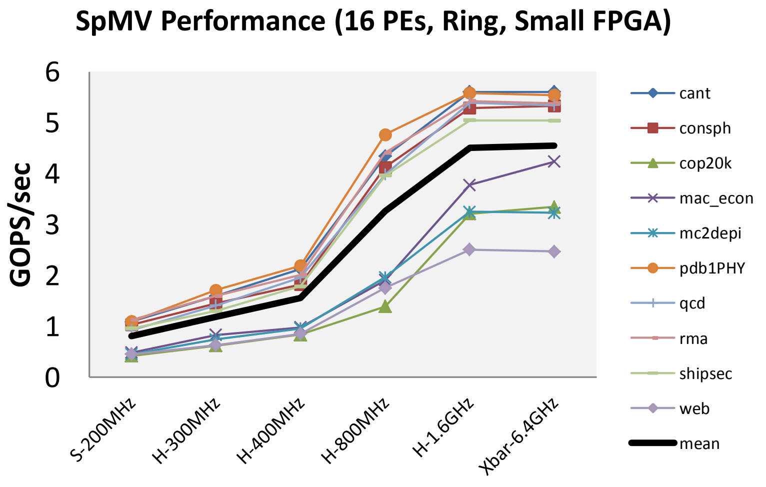


Figure 59: SpMV Performance Trends.

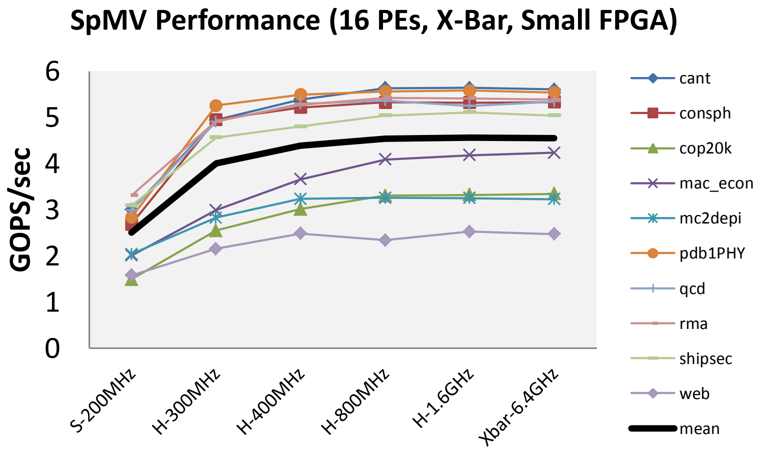
Performance Breakdown. Figure 62 shows detailed graphs comparing the timing and bandwidth characteristics of a well-behaved SpMV input (*cant*) versus a poorly-behaved input (*cop20k*) running on a small FPGA configured with a mesh network. Data points along the x-axis show near-linear scalability in performance with the number of instantiated SpMV PEs. All of the hard implementations show linear scalability with the number of PEs until the off-chip memory bandwidth is saturated (see Figure 62c). The timing breakdown of Figure 62b explains the performance gap between *cant* and *cop20k*. The categories show the percentage of time spent by each PE either waiting on memory stalls (indicated in blue or red) or performing computation (indicated in black). *Cop20k* experiences increased stall time due to a higher rate of x-cache misses. Each cache miss incurs the full latency of having a control thread detect the cache miss, issue a control action, having the transaction traverse through the cluster logic and the network-on-chip, and the DRAM latency as described earlier in Section 8.2.1. As the hard clock increases to 6.4GHz, the exposed x-cache



(a) Mesh Topology



(b) Ring Topology



(c) Crossbar

Figure 60: SpMV Per-Input Performance Trends.

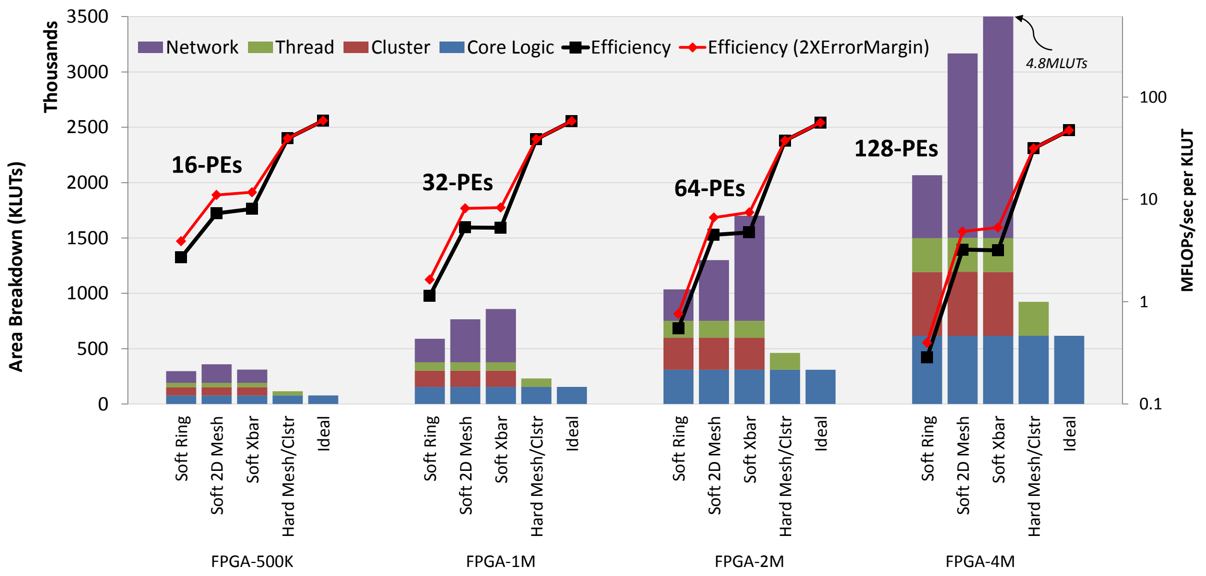


Figure 61: SpMV Area and Efficiency Trends.

stall becomes dominated by DRAM latency alone.

Area Efficiency. Figure 61 shows the area consumption of SpMV for different design points. Compared to the previous results of MMM and Bscholes, the overhead of CoRAM is substantial compared to the core logic, even in H-CoRAM. This result is unsurprising given that SpMV is a highly memory-intensive application and that very few compute resources are needed to saturate the available memory bandwidth. The gap between S-CoRAM and H-CoRAM is even wider in SpMV relative to Bscholes and MMM. In a *small FPGA*, the best of S-CoRAM achieves 8.1 MFLOPs/sec per KLUTs, while H-CoRAM achieves about 39.4 MFLOPs/sec. The nearly 5X gap in efficiency is attributed to lower clock-by-clock performance of S-CoRAM as well as the added overhead of the clusters and NoC in soft logic.

On average, H-CoRAM achieves about 65% performance efficiency relative to an idealized implementation of SpMV that operates at bandwidth-limited performance and without the cost of **any** infrastructure needed to route data operands from and to the memory interfaces. The bulk of the inefficiency of H-CoRAM is attributed to the overhead of the soft synthesizable control threads generated by CORCC. Recall from Chapter 5 that each PE of SpMV requires four threads to manage independent Stream FIFOs. Although we do not present concrete results, the main sources of overhead in the control threads are due to the use of 32b wide data types, which incur a high cost in muxing and arithmetic units. In a more optimized experimental version, the use of 16b short

	Bscholes (MOptions/sec per KLUTs)			MMM (GFLOPs/sec per KLUTs)			SpMV (MFLOPs/sec per KLUTs)		
	S-CoRAM	H-CoRAM	Ideal	S-CoRAM	H-CoRAM	Ideal	S-CoRAM	H-CoRAM	Ideal
Small	3.1	6.8	7.0	0.2	0.3	0.31	8.1	39.4	59
Medium	3.1	6.9	7.2	0.19	0.32	0.31	4.7	38.8	58.2
Large	3.1	7.2	7.4	0.16	0.35	0.35	4.8	37.6	56
Future	2.5	5.6	5.7	0.18	0.39	0.39	3.2	31.5	47.4

Table 24: Summary of Area-Normalized Performances Across Workloads.

integers can reduce the area of the control threads by nearly half, which would bring the relative efficiency to within 80% of an idealized implementation.

8.5 Summary

Table 24 summarizes the key results of our study, which compare the best efficiencies of S-CoRAM versus H-CoRAM. For Bscholes and MMM, the relative efficiency gap between S- and H-CoRAM is about 2-3X. In SpMV, the gap widens to about 5-10X, depending on the FPGA configuration.

Effectiveness of Software Abstraction. A major premise of this thesis is that a software-based abstraction for FPGA-based computing should not “prevent ” optimized applications from achieving peak performance potential. We have demonstrated in our results that a portable, scalable software-based abstraction for managing the memory resources of the FPGA is indeed plausible. We have shown through our RTL simulations of prototypes that hardened implementations of CoRAM can achieve performance and efficiency comparable to idealized implementations of applications. Another key result we show is that the separation of computation and memory is feasible without penalizing the efficiency or performance of applications.

Hard versus Soft CoRAM. Despite our best effort to develop an efficient soft logic implementation of CoRAM, a significant gap in performance and efficiency between 2-10X exists still between soft and hard implementations of CoRAM. The soft CoRAM designs incur a high area penalty as a result of resource sharing between the core logic and the memory subsystem within the fabric. Further, the high latency introduced by general-purpose subcomponents can have a negative impact on latency-sensitive applications. Most importantly, we have shown that hard CoRAM comprising the cluster logic and the network-on-chip can be implemented with modest impact on area (less than 2%). For

FPGAs in the future that employ CoRAM as an abstraction, hard logic is a highly cost-effective and efficient microarchitectural feature that can benefit a wide array of applications.

Network Topology. In our studies, we found that a 2D-mesh provided performance that was comparable equivalent to a full crossbar across all design points. A 2D-mesh is a particularly good match for the spatially distributed nature of FPGAs and was shown to be scalable for the futuristic FPGA designs presented in Table 10.

Application Portability and Scalability. In the examples from our studies, we have shown that an application written with CoRAM in mind is scalable to a variety of different FPGA designs without requiring modifications to the source code. The only change required between designs was increasing the number of instantiated elements along with the embedded CoRAMs employed by the application. We have shown that hard CoRAM is effective and scalable to futuristic FPGA designs with up to 200GB/sec of off-chip memory bandwidth and up to thousands of embedded CoRAMs in a single chip.

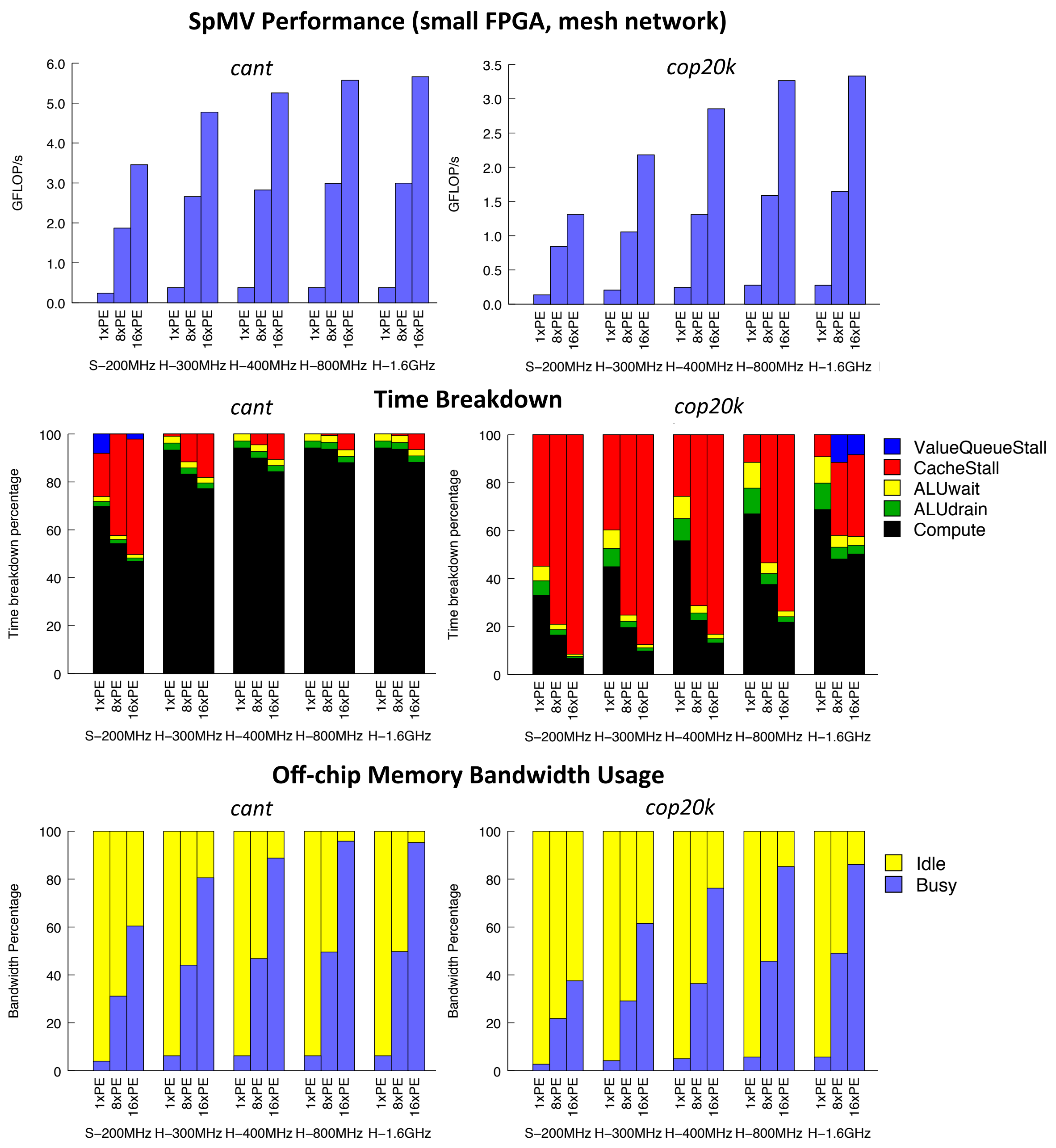


Figure 62: Detailed Comparison Between Good and Bad Inputs on SpMV.

Chapter 9

Related Work

If I have seen a little further—it is by standing on the shoulders of giants.

Isaac Newton

9.1 Specialized Reconfigurable Devices

The concept of reconfigurable computing has existed since the 1960s (e.g., Estrin [44]) and was the subject of a diverse range of research efforts in the 1990s (e.g., Programmable Active Memories [106], Virtual Computer [19], Splash and Splash 2 [49], and Teramac [14]). In recent years, specialized reconfigurable computing devices such as PipeRench [50, 89], RaPiD [42], and Amalgam [51] proposed reconfigurable fabrics for specialized streaming pipelined computations, while Garp [52], Chameleon [41], DISC [109], OneChip [18], PRISC [86], and Chimaera [119] combined fabrics with conventional processor cores. Designs such as ADRES [75, 102], Tilera [107], RAW [94], MATRIX [77], and MorphoSys [91] fall into the category of coarse-grain reconfigurable architectures, which incorporate coarse-grain functional units (or entire processing cores) rather than fine-grain lookup tables.

9.2 Reconfigurable Memory Architectures

Configurable memory systems have been explored in various settings. GARP [52] is an example that fabricates a MIPS core and cache hierarchy along with a collection of reconfigurable processing elements (PE). The PEs share access to the processor cache but only through a centralized access queue at the boundary of the reconfigurable logic. Tiled architectures (e.g., Tilera [107]) consist of a large array of simple von Neumann processors instead of fine-grained lookup tables. The memory accesses by the cores are supported through per-core private caches interconnected by an on-chip network. Smart Memories [73] on the other hand employs reconfigurable memory tiles that selectively act as caches, scratchpad memory, or FIFOs.

A body of work has also examined soft memory hierarchies for FPGAs (e.g., [121, 38, 62, 79]). The most closely related work to CoRAM is LEAP (Logic-based Environment for Application Programming) [10], which exports a standard, software-like programming environment to the user. The core functionalities of LEAP comprise a library of services and device abstractions that enable FPGA-based core logic to communicate with other software processes and platform-specific devices through standardized interfaces. A notable feature of LEAP is the Scratchpad service, which allows the user to dynamically allocate the on-die Block RAMs to form large storage elements on the FPGA. The scratchpads are accessed by an application using timing-insensitive request-response interfaces to local client address spaces. Beneath the interface, LEAP automatically performs multiple levels of caching through on-die BRAMs, off-chip SRAMs, and DRAM to provide the appearance of a large backing store. Chapter 3 compared the key differences between LEAP and CoRAM.

9.3 High Level Synthesis

The software-based control threads of CoRAM share similarities with traditional high-level synthesis (HLS), which attempts to replace low-level hardware description languages with familiar sequential languages such as C or C++ [45]. A significant body of academic and commercial work exists in the domain of high-level synthesis (HLS) with numerous commercial tools available such as SystemC [5], Catapult-C [76], SpecC [46], and AutoESL [1]. These languages retain the familiarity of sequential programming languages while generating hardware through automatic refinement and synthesis flows. Unlike HLS, a fundamental feature of CoRAM is the decoupling of computation and memory management into core logic and control threads. Unlike traditional HLS, which requires that computation and memory are expressed in a single language, CoRAM deliberately retains the ability to devise core logic in any RTL language. This deliberate decision is based on the observation that not all applications can be expressed efficiently in sequential languages.

9.4 Architectures and Program Models

The architecture of CoRAM shares similarities with other architectures and software-based programming models. For instance, the idea of decoupling memory management from computation in CoRAM has been explored previously in decoupled, general-purpose processors [92, 25].

The programming model of CoRAM also shares significant similarities to the Partitioned Global Address Space (PGAS) [120] and the Cray SHard MEMory Library (SHMEM). The PGAS model establishes a partitioned global address space accessible by all threads in the system. However, PGAS associates with each thread a partition of the global address space with a local affinity. The local affinity encourages programmers to allocate portions of distributed data structures close to the thread that uses it, thereby reducing communication costs. CoRAM closely resembles PGAS in that control threads have a global address view and are responsible for pinning frequently accessed private data to the embedded CoRAMs.

The Cray SHMEM model provides a programming model in a hybrid form of message passing and shared memory. Each processor in SHMEM observes a logically global shared memory view but can only access remote data through explicit *put* and *get* operators to a global address space. The explicit get/put operators are similar to the control actions used by control threads to access bulk data.

Chapter 10

Conclusions

A conclusion is the place where you got tired of thinking.

Arthur McBride Bloch, author of *Murphy's Laws*

As we stand on the brink of multi-billion transistor integration on a single die, FPGAs have emerged as a viable contender in the quest for power-efficient computing. Inspired by recent progress in FPGA-based computing, this thesis investigated a new memory architecture to provide deliberate support for memory accesses from within the fabric of a future FPGA engineered to be a computing device. The CoRAM memory architecture is a new paradigm designed to match the requirements of highly concurrent, spatially distributed processing kernels that consume and produce memory data from within the fabric. To reduce design effort and increase portability, CoRAM allows the application writer to express the memory access patterns of an application using a high-level specification language. This thesis showcased the usage of the CoRAM architecture through the design and implementation of three non-trivial application kernels. The thesis explored a cluster-style microarchitecture design for supporting the CoRAM architecture on a reconfigurable fabric and presented evaluations of the design space, paying attention to the tradeoffs between performance, power, and area. Based on prototyping and experimental results, this thesis concludes that CoRAM as an architecture is a highly compelling and feasible idea that merits further investigation and research.

10.1 Future Work

The investigation presented in this thesis is only preliminary and merits further investigation along the following areas:

Case Studies. The Cor-C specification in Chapter 4 only described a baseline collection of primitives sufficient to support the three applications examined in this thesis. The study of applications with new memory access patterns will help to refine the specification and to identify areas for improvement. The UC Berkeley Dwarves [15], for instance, describes various communication and memory patterns in parallel algorithms, three of which are covered in this thesis—exploring the remaining patterns in CoRAM is a good starting point for future investigations.

Automatic Extraction of Control Threads. Beginning with a single high-level description of an application (e.g., written in C), it should be possible to extract the control flow and memory accesses from the application description and generate the control thread automatically. Performing extraction would require developing compiler techniques that can separate the computation from memory within a single program description. Preliminary efforts undertaken by Gabriel Weisz using the LLVM framework have shown that this technique can work for certain styles of C code without loop-carried dependences.

CoRAM for ASICs. The CoRAM architecture has the potential simplify and standardize ASIC designs that require access to off-chip memory. The microarchitecture described in Chapter 6 could be viewed as a stylized, general-purpose template for implementing distributed memory accesses. The stylized microarchitecture comprising clusters and the network-on-chip could be generated automatically and synthesized efficiently using standard cells. An potential direction to pursue would be to compare ASIC applications that manually implement support for memory against designs that employ CoRAM.

VLSI Integration. The majority of the studies presented in this thesis were at the microarchitectural level and did not examine low-level VLSI issues such as layout, wiring, and circuit design. Also, the area, power, and frequency estimates reported in this thesis are only approximate since they are collected from early stages of the design flow. Further investigation is needed to pinpoint the costs and overheads of introducing hard CoRAM into traditional fabrics. Preliminary efforts are underway to develop a test chip that will help to answer these questions definitively.

Circuit-Switched Networks. The majority of the network-on-chips examined in this thesis were based on dynamic, packet-switched architectures such as the mesh and ring. Circuit-switched, multi-staged networks such as Clos networks [24] may offer more cost-efficient solutions, especially in FPGA soft logic. Compared to a crossbar switch, a Clos network requires fewer crosspoints to provide a full connection between input and output ports.

System-on-Chips. Although the CoRAM architecture focuses exclusively on compute applications, it can be just as applicable for embedded development and System-on-Chips (SoCs). Many FPGA-based SoCs employ processor bus architectures (e.g., IBM's Processor Local Bus, ARM's AXI) to provide a standard glue between multiple IP components. For design compatibility, designers must frequently redesign their IP cores to match a particular bus protocol's specification. Further, many of the "soft" busses on the FPGA are simply cloned adaptions of their ASIC counterparts, which makes them sub-optimal in performance and ill-suited to the low operating clock rates of fabric. The CoRAM architecture could be easily to adapted to replace the archaic bus architectures with a scalable, portable abstraction for IP-to-IP and IP-to-memory communication.

Appendix A

Interfaces and Libraries

A.1 Bluespec

```
interface ChannelFIFO;
    method Bool   put_valid();
    method Action put(Bit#(64) din);
    method Bool   get_valid();
    method ActionValue#(Bit#(64)) get();
endinterface

interface ChannelReg;
    method Action put(Bit#(64) in);
    method Bit#(64) get();
endinterface

interface RamIfc1#(type idx_t, type data_t);
  (* always_ready *) method Action req(Rnw access, idx_t idx, data_t data);
  (* always_ready *) method ActionValue#(data_t) val();
endinterface

interface Coram#(numeric type ports, type addr, type data);
    interface Vector#(ports, RamIfc1#(addr, data)) p;
endinterface

/* Black-box modules */

module mkChannelReg#(String thread,
                     Uint32 thread_id,
                     Uint32 object_id)(ChannelReg);

module mkChannelFIFOSub#(String thread,
                         Uint32 thread_id,
                         Uint32 object_id,
                         Uint32 sub_id)(ChannelFIFO);

module mkCoram#(String thread,
                Uint32 thread_id,
                Uint32 object_id,
                Uint32 sub_id)(Coram#(ports, addr, data))
                     provisos(Bits#(addr, addr_nt), Bits#(data, data_nt));
```

A.2 Cor-C Include Header

```
/******* Data types *******/

typedef void *                  cpi_hand;
typedef const char *            cpi_str;
typedef int                     cpi_int;
typedef long long               cpi_int64;
typedef unsigned long long      cpi_addr;

typedef unsigned int            cpi_ram_addr;
typedef unsigned short int      cpi_tag;

typedef enum {
    cpi_fifo = 0,
    cpi_reg = 1
} cpi_channel_ty;

#define CPI_INVALID_TAG 0x8000
#define CPI_ATTR_SINGLE_CLUSTER 0

#define cpi_poll(tag, x) \
        while(1) { \
            cpi_tag __temp__ = x;\
            if(__temp__ != CPI_INVALID_TAG) { tag = __temp__; break; }\
        }

#define cpi_wait(hand, tag) if(tag != CPI_INVALID_TAG)\
               { while(!cpi_test(hand, tag)) {}; tag = CPI_INVALID_TAG; }

#define cpi_write_ram(han, raddr, maddr, words) { \
    cpi_tag _tag_tmp = CPI_INVALID_TAG; \
    _tag_tmp = cpi_nb_write_ram(han, raddr, maddr, words, _tag_tmp); \
    cpi_wait(hand, _tag_tmp); \
}

#define cpi_read_ram(han, raddr, maddr, wrods) { \
    cpi_tag _tag_tmp = CPI_INVALID_TAG; \
    _tag_tmp = cpi_nb_read_ram(han, raddr, maddr, words, _tag_tmp); \
    cpi_wait(hand, _tag_tmp); \
}

/******* Thread control actions *******/
void        cpi_register_thread(cpi_str thread_name, cpi_int instances);
cpi_int     cpi_instance();
cpi_int     cpi_time();

/******* Accessor control actions *******/
cpi_hand    cpi_get_ram(cpi_int/*object_id*/, .../*sub_ids*/);
cpi_hand    cpi_get_rams(cpi_int/*num_rams*/,
                         bool/*scatter/gather*/,
                         cpi_int/*object_id*/, .../*subids*/);
cpi_hand    cpi_get_channel(cpi_channel_ty/*channel type*/,
                            cpi_int/*object_id*/, .../*subids*/);

/******* Channel control actions *******/
cpi_int64   cpi_read_channel(cpi_hand/*channel*/);
void        cpi_write_channel(cpi_hand/*channel*/, cpi_int64/*write data*/);
```

```
/******* Nonblocking memory control actions *******/
cpi_tag     cpi_nb_write_ram(cpi_hand/*dest ram*/,
                              cpi_ram_addr/*ram_addr*/,
                              cpi_addr/*mem_addr*/,
                              cpi_int/*words*/,
                              cpi_tag/*tag_append*/,
                              .../*optional priority argument*/);
cpi_tag     cpi_nb_read_ram(cpi_hand/*source ram*/,
                             cpi_ram_addr/*ram addr*/,
                             cpi_addr/*mem_addr*/,
                             cpi_int/*words*/,
                             cpi_tag/*tag_append*/,
                             .../*optional priority argument*/);
void        cpi_bind(cpi_hand/*channel*/, cpi_hand rams);
void        cpi_set_attr(cpi_hand, cpi_int);

/******* Test control actions *******/
bool        cpi_test_channel(cpi_hand/*channel*/, bool/*test if writable*/);
bool        cpi_test(cpi_hand/*cohandle*/, cpi_tag/*ram tag*/);

/******* Others *******/
void        cpi_printf(const char *fmt, ...);
void        cpi_split(); // force LLVM to split basic block here
```

Appendix B

Cache Memory Personality

B.1 Bluespec

```
module mkCocache#(String thread,
                  Uint32 thr_id,
                  Uint32 obj_id,
                  String name
                 ) (Cocache#(`CACHE))
             provisos(`CACHE_PROVISOS,
              Div#(data_nt,32,wide), Mul#(wide, 32, data_nt));

    Vector#(wide, Coram#(1, Bit#(idx_width), Bit#(32))) data_arr;

    for(Integer i=0; i < valueOf(wide); i=i+1)
        data_arr[i] <- mkCoram(thread, thr_id, obj_id, fromInteger(i));

    Coram#(2, Bit#(tag_idx_width), Bit#(32))
                      tag_arr <- mkCoram(thread,
                                         thr_id,
                                         obj_id,
                                         fromInteger(valueOf(wide)));

    ChannelFIFO  cfifo <- mkChannelFIFOSub(thread, thr_id, obj_id, 0);
    ChannelFIFO  bfifo <- mkChannelFIFOSub(thread, thr_id, obj_id, 1);

    ///////////// Miss handling //////////////

    Reg#(Bool) miss_pend     <- mkConfigReg(False);
    PulseWire  miss_pend_en  <- mkPulseWire();
    PulseWire  miss_pend_dis <- mkPulseWire();

    FIFOCountIfc#(CacheReq_t#(addr_t,data_t),16)    replayQ       <- mkLUTFIFO(False);
    Count#(Bit#(4),1)                               replayQ_cnt   <- mkCount(0);
    FIFOF#(addr_t)                                  missQ         <- mkUGSizedFIFOF(2);
    FIFOCountIfc#(CacheReq_t#(addr_t, data_t),4)    inQ           <- mkLUTFIFO(False);
    FIFOCountIfc#(data_t, 8)                        ackQ          <- mkLUTFIFO(False);
    Count#(Bit#(3),1)                               ackQ_cnt      <- mkCount(0);

    ///////////// Cache Pipeline State //////////////

    Reg#(CacheReq_t#(addr_t, data_t))               s0_req      <- mkRegU;
    Reg#(Bool)                                      s0_valid    <- mkReg(False);
```

```
Reg#(CacheReq_t#(addr_t, data_t))               s1_req      <- mkRegU;
Reg#(Bool)                                      s1_valid    <- mkReg(False);

Reg#(CacheReq_t#(addr_t, data_t))               s2_req      <- mkRegU;
Reg#(Bool)                                      s2_valid    <- mkReg(False);
Reg#(data_t)                                    s2_data     <- mkRegU;
Reg#(Tuple2#(CacheBits_t, Bit#(tag_width)))     s2_tag      <- mkRegU;

Reg#(CacheReq_t#(addr_t, data_t))               s3_req      <- mkRegU;
Reg#(Bool)                                      s3_valid    <- mkReg(False);
Reg#(data_t)                                    s3_data     <- mkRegU;
Reg#(Bool)                                      s3_hit      <- mkRegU;

PulseWire                                       stat_miss_w   <- mkPulseWire;
PulseWire                                       stat_lookup_w <- mkPulseWire;

function Bit#(idx_width) get_data_index(addr_t a);
    return truncate(pack(a) >> (valueOf(wd_width)));
endfunction

function Bit#(tag_idx_width) get_tag_index(addr_t a);
    return truncate(pack(a) >> (valueOf(blk_width)));
endfunction

function Bit#(tag_width) get_tag(addr_t a);
    return truncate(pack(a) >> (valueOf(idx_width)+valueOf(wd_width)));
endfunction

Reg#(Command) save <- mkRegU;

rule updateMissPend(True);
    if(miss_pend_en) miss_pend<=True;
    else if(miss_pend_dis) begin
        miss_pend<=False;
    end
endrule

rule connect_mem_ack(bfifo.get_valid && miss_pend);
    let ack <- bfifo.get();
    save <= save & ack;
    miss_pend_dis.send();
    Bit#(32) tag_bits = zeroExtend(pack(tuple2(CacheBits_t{
                  valid:True, dirty:False}, get_tag(missQ.first))));
    tag_arr.p[0].req(Write, get_tag_index(missQ.first), tag_bits);
    missQ.deq();
endrule

(* fire_when_enabled *)
rule work(True);

    /////////////////////////////////
    // stage 4 (start cache
    // miss, issue response)
    /////////////////////////////////

    if(!miss_pend && s3_valid) begin
        if(!s3_hit) begin
            miss_pend_en.send();
            Bit#(32) miss_addr  = zeroExtend(pack(s3_req.addr));
            Bit#(32) data_index = zeroExtend(pack(get_data_index(s3_req.addr)));

            missQ.enq(s3_req.addr);
            replayQ.enq(s3_req); // replay missed request
            replayQ_cnt.add(1);
            cfifo.put({data_index, miss_addr});
            stat_miss_w.send();
        end
        else begin
            ackQ_cnt.add(1);
            ackQ.enq(s3_data); // return data
        end
    end
    else begin
        if(s3_valid) begin
            replayQ.enq(s3_req);
            replayQ_cnt.add(1);
        end
    end

   /////////////////////////////////
    // stage 3 (tag-check)
    /////////////////////////////////

    let status  = tpl_1(s2_tag);
    let tag     = tpl_2(s2_tag);
    let realtag = get_tag(s2_req.addr);
    Bool valid  = status.valid;

    s3_hit    <= valid && (tag==realtag);
    s3_req    <= s2_req;
    s3_valid  <= s2_valid;
    s3_data   <= s2_data;

    /////////////////////////////////
    // stage 2 (latch RAM responses)
    /////////////////////////////////

    Vector#(wide, Bit#(32)) data_vec;
    for(Integer i=0; i < valueOf(wide); i=i+1) begin
        let d <- data_arr[i].p[0].val();
        data_vec[i] = d;
    end
    data_t data = unpack(pack(data_vec));
    let tag_bits <- tag_arr.p[1].val();
    Tuple2#(CacheBits_t, Bit#(tag_width)) taginfo = unpack(truncate(tag_bits));

    s2_req    <= s1_req;
    s2_valid  <= s1_valid;
    s2_data   <= data;
    s2_tag    <= taginfo;

    /////////////////////////////////
    // stage 1 (issue RAM request)
    /////////////////////////////////

    s1_req   <= s0_req;
```

```
        s1_valid <= s0_valid;

        for(Integer i=0; i < valueOf(wide); i=i+1)
            data_arr[i].p[0].req(Read, get_data_index(s0_req.addr), ?);

        tag_arr.p[1].req(Read, get_tag_index(s0_req.addr), ?);

        /////////////////////////////////
        // stage 0 (latch inputs)
        /////////////////////////////////

        if(replayQ.notEmpty && !miss_pend) begin
            s0_req   <= replayQ.first;
            s0_valid <= True;
            replayQ.deq();
            replayQ_cnt.sub(1);
        end
        else if(!miss_pend) begin
            s0_req <= inQ.first;
            s0_valid <= inQ.notEmpty;
            if(inQ.notEmpty) inQ.deq();
        end
        else s0_valid <= False;

    endrule

    interface Server user_ifc;

        interface Put request;
            method Action put(CacheReq_t#(addr_t,data_t) req)
                      if(!miss_pend && (inQ.count <= 1)
                         && (ackQ_cnt.value <= 1) && (replayQ_cnt.value <= 1));
                stat_lookup_w.send();
                inQ.enq(req);
            endmethod
        endinterface

        interface Get response;
            method ActionValue#(data_t) get() if(ackQ.notEmpty && !miss_pend);
                ackQ_cnt.sub(1);
                ackQ.deq();
                return ackQ.first;
            endmethod
        endinterface

    endinterface

endmodule
```

B.2 Control Thread Program

```
void
cache_thread()
{
    cpi_register_thread("cache_thread", N_PE);
    DECLARE_LDST(ldst_fifo, ldst_ram, 1/*obj_id*/);

    cpi_hand fifo = cpi_get_channel(cpi_fifo, 0, 0);
    cpi_hand bfifo = cpi_get_channel(cpi_fifo, 0, 1);
    cpi_hand data_ram = cpi_get_rams(XCACHE_WIDE, true, 0, 0);
    cpi_hand tag_ram = cpi_get_rams(1, false, 0, XCACHE_WIDE);
    cpi_addr x_ptr = mp_thread_read(ldst_ram, ldst_fifo, BASE_ADDR + X_OFFSET);
    cpi_bind(bfifo, data_ram);

#define log2(x) \
    ((x) == 1) ? 0 : \
    ((x) == 2) ? 1 : \
    ((x) == 4) ? 2 : \
    ((x) == 8) ? 3 : \
    ((x) ==16) ? 4 : \
    ((x) ==32) ? 5 : \
    ((x) ==64) ? 6 : 0

    int log_word_bytes = log2(XCACHE_WIDE * (WORD_WIDTH/8));

    while(1) {
        cpi_int64 message = cpi_read_channel(fifo);
        cpi_addr miss_addr = message & 0xffffffffu & ~(XCACHE_BLK_BYTES-1);
        cpi_int64 data_index =
         miss_addr & (XCACHE_SIZE_BYTES-1) & ~(XCACHE_BLK_BYTES-1);
        cpi_tag tag = CPI_INVALID_TAG;
        while(1) {
            tag = cpi_nb_write_ram(data_ram,
                                   data_index >> log_word_bytes,
                                   x_ptr + miss_addr,
                                   XCACHE_BLK_BYTES,
                                   tag, 1/*hiprio*/);
            if(tag != CPI_INVALID_TAG) break;
        }
    }
}
```

Appendix C

FIFO Memory Personality

C.1 Bluespec

```
interface StreamFIFO#(type addr, type data);
    method Action deq();
    method data first();
    method Bool notEmpty();
    method Uint32 get_id();
endinterface

module mkStreamFIFO#(String thread, Uint32 thread_id, Uint32 object_id)
                                           (StreamFIFO#(addr_t, data_t))
               provisos(Bits#(addr_t, addr_nt),
               Bits#(data_t, data_nt),
               Add#(addr_nt, v, 32),
               Add#(addr_nt, t, 64),
               Div#(data_nt, 32, nr_nt),
               Bits#(Vector#(nr_nt, Bit#(32)), data_nt));

    function makeCoram(Integer i) =
                   mkCoram(thread, thread_id, object_id, fromInteger(i));

    Vector#(nr_nt, Coram#(1, addr_t, Bit#(32))) corams <- genWithM(makeCoram);
    FIFOCountIfc#(Bit#(data_nt),32) dfifo <- mkLUTFIFO(False);

    Reg#(Bit#(addr_nt)) tail   <- mkConfigReg(0);
    Reg#(Bit#(addr_nt)) head   <- mkConfigReg(0);
    FIFO#(void)         vfifo  <- mkSizedFIFO(16); // valid tokens

    ChannelReg          creg   <- mkChannelReg(thread, thread_id, object_id);
    ChannelFIFO         headQ  <- mkChannelFIFOSub(thread, thread_id, object_id, 1);
    ChannelFIFO         bindQ  <- mkChannelFIFOSub(thread, thread_id, object_id, 2);

    rule updateHead(bindQ.get_valid && headQ.get_valid);
        let x <- bindQ.get();
        let nhead <- headQ.get();
        head <= truncate(nhead);
    endrule

    rule fill_fifo((tail != head) && dfifo.count <= 16);
        for(Integer i=0; i < valueOf(nr_nt); i=i+1) begin
            corams[i].p[0].req(Read, unpack(truncate(tail)), ?);
        end
        tail<=tail+1;
        vfifo.enq(?);
    endrule

    rule fillFifo(True);
        vfifo.deq();
        Vector#(nr_nt, Bit#(32)) vv;
        for(Integer i=0; i < valueOf(nr_nt); i=i+1) begin
            let val <- corams[i].p[0].val;
            vv[i] = val;
        end
        dfifo.enq(pack(vv));
    endrule

    rule update_tail(True);
        creg.put(zeroExtend(tail));
    endrule

    method Action deq();
        dfifo.deq();
    endmethod

    method first() = unpack(pack(dfifo.first));
    method notEmpty() = dfifo.notEmpty;
    method get_id() = corams[0].get_id;

endmodule
```

C.2 Control Thread Program

```
void write_stream_fifo
           (cpi_hand rams,
           cpi_hand creg,
           cpi_hand hfifo,
           cpi_addr src,
           int bytes,
           int word_size,
           int depth,
           int *head)
{
    int words_left = divide(bytes, word_size);
    int mask = depth - 1;
    int src_word = 0;

    while(words_left > 0)
    {
        int tail = (int)cpi_read_channel(creg);
        int free_words = (*head >= tail) ?
                depth - 1 - (*head - tail) : (tail - *head - 1);
        int bsize_words = MIN(((MAX_REQ_BYTES/2)/word_size),
                              MIN(free_words, words_left));

        if((bsize_words != 0) && (free_words >= 64)) {
            cpi_int tag = cpi_nb_write_ram(rams, *head,
                                             src + src_word * word_size,
                                             bsize_words * word_size,
                                             CPI_INVALID_TAG);
            if(tag != CPI_INVALID_TAG) {
                src_word += bsize_words;
                words_left -= bsize_words;
                *head = (*head + bsize_words) & mask;
                cpi_write_channel(hfifo, *head);
            }
        }
    }
    return ;
}
```

Appendix D

Matrix-Matrix Multiplication

D.1 Bluespec

```
interface CogemmPE;
    method Action set_thread_id(Uint32 id);
    method Action set_id(Uint32 id);
    method Action putOp(Opand b_in);
    method Opand  getOp();
    method Action start();
    method ActionValue#(Bit#(0)) done();
    interface AntiCoram#(1, RamAddr, Opand) antiRamA;
    interface AntiCoram#(1, RamAddr, Opand) antiRamB;
    interface AntiCoram#(2, RamAddr, Opand) antiRamC0;
    interface AntiCoram#(2, RamAddr, Opand) antiRamC1;
endinterface

module mkCogemmPE(CogemmPE);

  RWire#(Uint32) id_w <- mkRWire();
  RWire#(Uint32) tid_w <- mkRWire();
  function Uint32 gid() = validValue(id_w.wget);
  function Uint32 tid() = validValue(tid_w.wget);

  ////////////////////////////////////////////////////////////////////
  // Datapath (BRAMs + FU)
  ////////////////////////////////////////////////////////////////////

  Reg#(Opand)                 cLatch            <- mkRegU;
  Madd                        madder            <- mkMadd(0);
  Vector#(3, Reg#(Pstage))    stages            <- replicateM(mkConfigReg(unpack(0)));

  ////////////////////////////////////////////////////////////////////
  // Control
  ////////////////////////////////////////////////////////////////////

  Reg#(PeState)               state             <- mkReg(PeWait);
  Reg#(Bit#(1))               compPhase         <- mkReg(0);
  FIFOF#(Bit#(0))             readyQ            <- mkSizedFIFOF(2);
  FIFOF#(Bit#(0))             doneQ             <- mkSizedFIFOF(2);
  PulseWire                   drainDone         <- mkPulseWire();

  ////////////////////////////////////////////////////////////////////
  // Operand network
```

```
/////////////////////////////////////////////////////////////////

RWire#(Opand)              opIn             <- mkRWire();
Reg#(Opand)                bOutQ            <- mkRegU;

////////////////////////////////////////////////////////////////
// Memory
/////////////////////////////////////////////////////////////////

AntiRamToRam#(1, RamAddr, Opand) aRam <- mkAntiRamToRam();
AntiRamToRam#(1, RamAddr, Opand) bRam <- mkAntiRamToRam();
AntiRamToRam#(2, RamAddr, Opand) cRam0 <- mkAntiRamToRam();
AntiRamToRam#(2, RamAddr, Opand) cRam1 <- mkAntiRamToRam();

Reg#(BigRamAddr)  loop_index         <- mkReg(0);
Reg#(RamCount)    outer_index        <- mkReg(0); // outer loop
Reg#(RamCount)    inner_index        <- mkReg(0); // inner loop
Reg#(Bool)        inner_loop_done    <- mkReg(False);
Reg#(Bool)        last_loop_done     <- mkReg(False);

rule startCompute(readyQ.notEmpty);
  readyQ.deq();
  inner_index     <= 0;
  outer_index     <= 0;
  inner_loop_done <= False;
  last_loop_done  <= False;
  state <= PeLoop;
endrule

(* fire_when_enabled *)
rule compute_stage_0(state == PeLoop);
  Pstage p = ?;

  p.valid = True;
  p.addrA = {compPhase, truncate(outer_index)};
  p.addrB = {compPhase, truncate(outer_index)};

  RamIndex c_ram_index = truncate(inner_index);
  c_ram_index = c_ram_index + truncate(gid);

  p.addrC = {compPhase, truncate(c_ram_index)};
  p.useNeighbor = (inner_index != 0);
  p.comp_phase = compPhase;

  stages[0] <= p;

  inner_loop_done <= (inner_index == fromInteger(valueOf(BlockDim)-2));
  last_loop_done  <= (loop_index == fromInteger(valueOf(BlockDim)*valueOf(A_width)-2));
  outer_index     <= inner_loop_done ? outer_index+1 : outer_index;
  inner_index     <= inner_loop_done ? 0 : inner_index+1;
  loop_index      <= last_loop_done  ? 0 : loop_index+1;
  state           <= last_loop_done  ? PeDrain : state;
endrule

(* fire_when_enabled *)
rule not_stage_0(state != PeLoop);
  Pstage p = ?;
  p.valid  = False;
  stages[0] <= p;
```

```
endrule

(* fire_when_enabled *)
rule compute_stage_1(True);
  let p = stages[0];
  let cAddr = p.addrC;

  if(p.valid) begin
    aRam.ram.p[0].req(Read, p.addrA, ?);
    bRam.ram.p[0].req(Read, p.addrB, ?);
    cRam0.ram.p[0].req(Read, cAddr, ?);
    cRam1.ram.p[0].req(Read, cAddr, ?);
  end

  stages[1] <= p;
endrule

(* fire_when_enabled *)
rule compute_stage_2(True);
  let p  = stages[1];
  let av <- aRam.ram.p[0].val();
  let bv <- bRam.ram.p[0].val();
  let cv0 <- cRam0.ram.p[0].val();
  let cv1 <- cRam1.ram.p[0].val();

  p.valA = av;
  p.valB = bv;
  p.valC = p.comp_phase == 1 ? cv1 : cv0;

  stages[2] <= p;
endrule

(* fire_when_enabled *)
rule compute_stage_3(True);
  let p = stages[2];
  let bVal = p.useNeighbor ? validValue(opIn.wget) : p.valB;
  if(p.valid)
    madder.issue(p.valA, bVal, p.valC);
  bOutQ <= bVal;
endrule

/////////////////////////////////////////////////////////////////
// Local writes to C ram
/////////////////////////////////////////////////////////////////

Reg#(BigRamAddr) cWrCnt         <- mkReg(0); // outer_c
Reg#(RamCount)   cWrIndex       <- mkReg(0); // inner_c
Reg#(Bool)       cLoop          <- mkReg(False); // inner_c
Reg#(Bool)       cDone          <- mkReg(False); // inner_c

(* fire_when_enabled *)
rule write_dotprod(madder.done);
  RamIndex c_ram_index = truncate(cWrIndex) + truncate(gid);
  RamAddr cAddr = {compPhase,truncate(c_ram_index)};
  if(compPhase == 1) cRam1.ram.p[1].req(Write, cAddr, madder.result);
  else cRam0.ram.p[1].req(Write, cAddr, madder.result);
  cWrIndex <= cLoop ? 0 : cWrIndex+1;

  if(cDone) begin
```

```
      cWrCnt <= 0;
      cLoop  <= False;
      cDone  <= False;
      doneQ.enq(?);
      drainDone.send();
    end else
    begin
      cWrCnt <= cWrCnt+1;
      cLoop  <= (cWrIndex == fromInteger(valueOf(BlockDim)-2));
      cDone  <= (cWrCnt == fromInteger(valueOf(BlockDim)*valueOf(A_width)-2));
    end
  endrule

  rule restartCompute(drainDone && (state == PeDrain));
    compPhase <= ~compPhase;
    state <= PeWait;
  endrule

  ////////////////////////////////////////////////////////////////////
  // Interfaces
  ////////////////////////////////////////////////////////////////////

  method Action start();
      readyQ.enq(?);
  endmethod

  method ActionValue#(Bit#(0)) done();
      actionvalue
          doneQ.deq();
          return doneQ.first();
      endactionvalue
  endmethod

  method Action putOp(Opand b_in);
    opIn.wset(b_in);
  endmethod

  method Opand getOp();
    return bOutQ;
  endmethod

  method Action set_thread_id(Uint32 _id);
      tid_w.wset(_id);
  endmethod

  method Action set_id(Uint32 _id);
      id_w.wset(_id);
  endmethod

  interface antiRamA = aRam.anti;
  interface antiRamB = bRam.anti;
  interface antiRamC0 = cRam0.anti;
  interface antiRamC1 = cRam1.anti;

endmodule
```

D.2 Control thread program

```
#define N 2*NumPEs
#define N_PE NumPEs
#define DTYPE sizeof(float)
#define A_OFF (cpi_instance()*(3*N*N) + 0)
#define B_OFF (cpi_instance()*(3*N*N) + (N*N)*sizeof(float))
#define C_OFF (cpi_instance()*(3*N*N) + (2*N*N)*sizeof(float))
#define N_THREADS ReplicationFactor

int
gemm_thread()
{
    int NB = N_PE, ram_depth = N_PE * 2;
    cpi_addr dataA = 0, dataB = B_OFF, dataC = C_OFF;
    cpi_int64 token = 0;
    int prev_j = 0, prev_i = 0;
    cpi_tag tagA = CPI_INVALID_TAG,
            tagB = CPI_INVALID_TAG,
            tagC0 = CPI_INVALID_TAG,
            tagC1 = CPI_INVALID_TAG;

    cpi_register_thread("gemm_thread", N_THREADS);
    cpi_hand cfifo = cpi_get_channel(cpi_fifo, 0);
    cpi_hand ramsA = cpi_get_rams(N_PE, false, 0, 0);
    cpi_hand ramsB = cpi_get_rams(N_PE, true, 0, N_PE);
    cpi_hand ramsC0 = cpi_get_rams(N_PE, false, 0, 2*N_PE);
    cpi_hand ramsC1 = cpi_get_rams(N_PE, false, 0, 3*N_PE);

    bool which = false; // for double buffer

    for (int k = 0; k < N; k += NB) {
        for (int j = 0; j < N; j += NB) {
            for (int i = 0; i < N; i += NB) {
                int offset = which ? ram_depth/2:0;
                int c_offset = !which ? ram_depth/2:0;
                for(int r=0; r < NB; r++)
                {
                    cpi_poll(tagA, cpi_nb_write_ram(ramsA,
                                   r*ram_depth+offset,
                                   dataA+DTYPE*(i*N+k+r*N),
                                   NB*DTYPE,
                                   tagA));

                    cpi_poll(tagB, cpi_nb_write_ram(ramsB,
                                   r+offset,
                                   dataB+DTYPE*(k*N+j+r*N),
                                   NB*DTYPE,
                                   tagB));

                    if(!which) {
                        cpi_poll(tagC0, cpi_nb_write_ram(ramsC0,
                                   r*ram_depth+offset,
                                   dataC+DTYPE*(i*N+j+r*N),
                                   NB*DTYPE,
                                   tagC0));

                        cpi_poll(tagC1, cpi_nb_read_ram(ramsC1,
                                   r*ram_depth+c_offset,
                                   dataC+DTYPE*(prev_i*N+prev_j+r*N),
                                   NB*DTYPE,
                                   tagC1));
                    }
                    else {
                        cpi_poll(tagC0, cpi_nb_write_ram(ramsC0,
                                   r*ram_depth+offset,
                                   dataC+DTYPE*(i*N+j+r*N),
                                   NB*DTYPE,
                                   tagC0));
                        cpi_poll(tagC1, cpi_nb_read_ram(ramsC1,
                                   r*ram_depth+c_offset,
                                   dataC+DTYPE*(prev_i*N+prev_j+r*N),
                                   NB*DTYPE,
                                   tagC1));
                    }
                }
                cpi_wait(ramsA, tagA);
                cpi_wait(ramsB, tagB);
                cpi_wait(ramsC0, tagC0);
                cpi_wait(ramsC1, tagC1);
                cpi_write_channel(cfifo, token);
                token = cpi_read_channel(cfifo);
                prev_i = i;
                prev_j = j;
                which = !which;
            }
        }
    }

    return 0;
}
```

Appendix E

Sparse Matrix-Vector Multiplication

E.1 Bluespec

```
module mkPE#(Uint32 uid)(PE);

    FIFOCountIfc#(Pe_addr, 32)   inAddrQ  <- mkLUTFIFO(True);
    FIFOCountIfc#(Pe_input, 32)  inQ      <- mkLUTFIFO(True);
    FIFOCountIfc#(Pe_input, 32)  pendQ    <- mkLUTFIFO(True);
    FIFOCountIfc#(Rword, 32)     dotSizeQ <- mkLUTFIFO(True);

    let row_ls                   <- mkLoadStore("row_thread", uid, 1);
    let val_ls                   <- mkLoadStore("val_thread", uid, 1);
    let addr_ls                  <- mkLoadStore("addr_thread", uid, 1);
    let x_ls                     <- mkLoadStore("xcache_thread", uid, 1);
    let y_ls                     <- mkLoadStore("y_thread", uid, 1);
    let ref_ls                   <- mkLoadStore("ref_thread", uid, 1);

    Valfifo          valQ       <- mkCofifoBind("val_thread", uid, 0);
    Addrfifo         addrQ      <- mkCofifoBind("addr_thread", uid, 0);
    Xcache           xcache     <- mkXcache("xcache_thread", "[ x-cache ]", uid);
    FIFOCountIfc#(
        Dword, 32)   xcacheQ    <- mkLUTFIFO(True);

    Yfifo            yQ         <- mkToCofifoBind("y_thread", uid, 0);
    Rfifo            refQ       <- mkCofifoBind("ref_thread", uid, 0);

    Reg#(Rword)      dotCount   <- mkReg(0);
    IntMacc#(Dword,1) macc      <- mkIntMacc();
    Reg#(Bool)       waitSum    <- mkReg(False);
    Reg#(Bit#(64))   numMadds   <- mkReg(0); // Stats

    ////////////////////////////////////////////////////////////////////////////////

    rule requestsToMemory(valQ.cfifo.put_valid && addrQ.cfifo.put_valid);
        let in = inAddrQ.first;
        inAddrQ.deq();
        valQ.cfifo.put({pack(in.dotsize), pack(in.val_base)});
        addrQ.cfifo.put({pack(in.dotsize), pack(in.val_base)});
        yQ.cfifo.put({fromInteger(valueOf(RowSize)), pack(in.row)});
    endrule

    rule processWork(True);
        let in = inQ.first;
        inQ.deq();
        pendQ.enq(in);
        dotSizeQ.enq(in.dotsize-1);
    endrule

    rule processAddresses(addrQ.notEmpty);
        let addr = addrQ.first;
        addrQ.deq();
        let corr = addr;
        addr = addr << valueOf(TLog#(TDiv#(WordWidth,8)));
       xcache.user_ifc.request.put(
           CacheReq{addr:addr, rnw:True, rtoken:?, size:?, data:?, id: ?} );
    endrule

    rule drainSum(waitSum);
        let dot_info = pendQ.first;

        if(macc.rdy) begin
            yQ.enq(macc.get);
            macc.clear();
            pendQ.deq();
            dotSizeQ.deq();
            waitSum <= False;
            Int#(WordWidth) sumRes = unpack(macc.get);
        end
    endrule

    rule drainXcache(True);
        Dword xval;
        let _xval <- xcache.user_ifc.response.get();
        xval = _xval;
        xcacheQ.enq(xval);
    endrule

    rule processValues(!waitSum && xcacheQ.notEmpty && valQ.notEmpty);
        let dot_info = pendQ.first;
        let xval = xcacheQ.first;
        let val = valQ.first;
        xcacheQ.deq();
        valQ.deq();
        Dword v = unpack(val);
        Dword x = unpack(xval);
        macc.put(v, x);

        if(dotCount == dotSizeQ.first) begin
            waitSum  <= True;
            dotCount <= 0;
        end
        else dotCount <= dotCount + 1;
    endrule

    method Action put(Pe_input in);
        inQ.enq(in);
    endmethod

    method Action put_addr(Pe_addr in);
        inAddrQ.enq(in);
    endmethod
```

```
endmodule
```

E.2 Control thread program

```
#include "cpi.h"
#include "Cofifo.h"
#include "CofifoBind.h"
#include "LoadStore.h"

#define BASE_ADDR    0x0
#define N_ROW_OFFSET 0x0
#define VAL_OFFSET   0x4
#define COL_OFFSET   0x8
#define ROW_OFFSET   0xc
#define X_OFFSET     0x10
#define Y_OFFSET     0x14

static void
unpack_data(cpi_int64 bits, int *first, int *second)
{
    *first  = bits & 0xffffffff;
    *second = (bits >> 32ull) & 0xffffffff;
}

void
row_thread()
{
    cpi_register_thread("row_thread", 1);

    DECLARE_LDST(ldst_fifo, ldst_ram, 1/*obj_id*/);

    /* Determine start address for rows */
    int n_rows, row_offset;
    n_rows = mp_thread_read(ldst_ram, ldst_fifo, BASE_ADDR + N_ROW_OFFSET);
    row_offset = mp_thread_read(ldst_ram, ldst_fifo, BASE_ADDR + ROW_OFFSET);
    cpi_addr row_ptr = BASE_ADDR + row_offset;

    /* Create stream Co-fifo object*/
    DECLARE_CFIFO_BIND(rams, ROW_BUNDLE_WIDTH/32, creg, cfifo, hfifo, bfifo, head, 0);
    cpi_set_attr(rams, CPI_ATTR_SINGLE_CLUSTER);

    /* Use built-in channel to communicate row information */
    cpi_write_channel(cfifo, n_rows);

    /* Start streaming in schedule data */
    write_fifo_bind(rams, creg, hfifo, row_ptr, sizeof(int)*(n_rows+2),
         ROW_BUNDLE_WIDTH/8 /*word_size*/, FIFO_DEPTH, &head, "\"stream-thread\"");
}

void
val_thread()
{
    cpi_register_thread("val_thread", N_PE);

    DECLARE_LDST(ldst_fifo, ldst_ram, 1/*obj_id*/);
    DECLARE_CFIFO_BIND(rams, 1, creg, cfifo, hfifo, bfifo, head, 0);
    cpi_addr val_ptr = mp_thread_read(ldst_ram, ldst_fifo, BASE_ADDR + VAL_OFFSET);
    int first, second;

    while(1)
    {
        cpi_int64 val = cpi_read_channel(cfifo);
        unpack_data(val, &first, &second);
        write_fifo_bind(rams, creg, hfifo, val_ptr+first*sizeof(int),
                  second*sizeof(int), 4, FIFO_DEPTH, &head);
    }
}


void
addr_thread()
{
    cpi_register_thread("addr_thread", N_PE);

    DECLARE_LDST(ldst_fifo, ldst_ram, 1/*obj_id*/);
    DECLARE_CFIFO_BIND(rams, 1, creg, cfifo, hfifo, bfifo, head, 0);
    cpi_addr col_ptr = mp_thread_read(ldst_ram, ldst_fifo, BASE_ADDR + COL_OFFSET);
    int first, second;
    while(1)
    {
        cpi_int64 val = cpi_read_channel(cfifo);
        unpack_data(val, &first, &second);
        write_fifo_bind(rams, creg, hfifo, col_ptr + first * sizeof(int),
             second * sizeof(int), 4, FIFO_DEPTH, &head);
    }
}

void
xcache_thread()
{
    cpi_register_thread("xcache_thread", N_PE);
    DECLARE_LDST(ldst_fifo, ldst_ram, 1/*obj_id*/);

    cpi_hand fifo = cpi_get_channel(cpi_fifo, 0, 0);
    cpi_hand bfifo = cpi_get_channel(cpi_fifo, 0, 1);
    cpi_hand data_ram = cpi_get_rams(XCACHE_WIDE, true, 0, 0);
    cpi_hand tag_ram = cpi_get_rams(1, false, 0, XCACHE_WIDE);
    cpi_addr x_ptr = mp_thread_read(ldst_ram, ldst_fifo, BASE_ADDR + X_OFFSET);
    cpi_bind(bfifo, data_ram);

#define log2(x) \
    ((x) == 1) ? 0 : \
    ((x) == 2) ? 1 : \
    ((x) == 4) ? 2 : \
    ((x) == 8) ? 3 : \
    ((x) ==16) ? 4 : \
    ((x) ==32) ? 5 : \
    ((x) ==64) ? 6 : 0

    int log_word_bytes = log2(XCACHE_WIDE * (WORD_WIDTH/8));

    while(1) {
        cpi_int64 message = cpi_read_channel(fifo);
        cpi_addr miss_addr = message & 0xffffffffu & ~(XCACHE_BLK_BYTES-1);
        cpi_int64 data_index = miss_addr
        & (XCACHE_SIZE_BYTES-1) & ~(XCACHE_BLK_BYTES-1);
        cpi_tag tag = CPI_INVALID_TAG;
        while(1) {
            tag = cpi_nb_write_ram(data_ram, data_index >> log_word_bytes,
    x_ptr + miss_addr, XCACHE_BLK_BYTES, tag, 1/*hiprio*/);
            if(tag != CPI_INVALID_TAG) break;
        }
    }
}

void
y_thread()
{
    cpi_register_thread("y_thread", N_PE);
    DECLARE_LDST(ldst_fifo, ldst_ram, 1/*obj_id*/);
    DECLARE_CFIFO_BIND(rams, 1, creg, cfifo, tfifo, bfifo, tail, 0);

    cpi_addr y_ptr = mp_thread_read(ldst_ram, ldst_fifo, BASE_ADDR + Y_OFFSET);
    int n_rows = mp_thread_read(ldst_ram, ldst_fifo, BASE_ADDR + N_ROW_OFFSET);

    while(1) {
        int first, second;
        cpi_int64 val = cpi_read_channel(cfifo);
        unpack_data(val, &first, &second);
        read_fifo_bind(rams, creg, tfifo, y_ptr + (first + n_rows) * sizeof(int),
                   second * sizeof(int), 4, FIFO_DEPTH, &tail);
    }
}
```

References

[1] AutoESL High-Level Synthesis Tool. `http://www.xilinx.com/tools/autoesl.htm`.

[2] Compressed Sparse Row Format. `http://netlib.org/linalg`.

[3] MicroBlaze Soft Processor Core. `http://www.xilinx.com/tools/microblaze.htm`.

[4] Personal communication.

[5] SystemC: The Open SystemC Initiative. `http://www.systemc.org`.

[6] Xilinx CORE Generator System. `http://www.xilinx.com/tools/coregen.htm`.

[7] XST User Guide for Virtex-6, Spartan-6, and 7 Series Devices. `http://www.xilinx.com/support/documentation/sw_manuals/xilinx13_1/xst_v6s6.pdf`.

[8] Hard vs. Soft: The Central Question of Pre-Fabricated Silicon. In *Proceedings of the 34th International Symposium on Multiple-Valued Logic*, pages 2–5, Washington, DC, USA, 2004. IEEE Computer Society.

[9] Achronix, Inc. Speedster Product Brief. `http://www.achronix.com`.

[10] Michael Adler, Kermin E. Fleming, Angshuman Parashar, Michael Pellauer, and Joel Emer. LEAP Scratchpads: Automatic Memory and Cache Management for Reconfigurable Logic. In *FPGA'11: Proceedings of the 2011 ACM/SIGDA 19th International Symposium on Field Programmable Gate Arrays*, 2011.

[11] Altera, Inc. `http://www.altera.com`.

[12] Altera, Inc. Internal memory (ram and rom) user guide. `http://www.altera.com/literature/ug/ug_ram_rom.pdf`.

[13] AMD, Inc. `http://www.fusion.amd.com`.

[14] R. Amerson, R. J. Carter, W. B. Culbertson, P. Kuekes, and G. Snider. Teramac-configurable custom computing. In *FCCM'95: Proceedings of the IEEE Symposium on FPGA's for Custom Computing Machines*, page 32, Washington, DC, USA, 1995. IEEE Computer Society.

[15] Krste Asanovic, Rastislav Bodik, James Demmel, Tony Keaveny, Kurt Keutzer, John Kubiatowicz, Nelson Morgan, David Patterson, Koushik Sen, John Wawrzynek, David Wessel, and Katherine Yelick. A view of the parallel computing landscape. *Commun. ACM*, 52:56–67, October 2009.

[16] Luca Benini, Alberto Macii, Enrico Macii, and Massimo Poncino. Synthesis of Application-Specific Memories For Power Optimization in Embedded Systems. In *Proceedings of the 37th Annual Design Automation Conference*, DAC'00, pages 300–303, New York, NY, USA, 2000. ACM.

[17] Bluespec, Inc. `http://www.bluespec.com/products/bsc.htm`.

[18] Jorge E. Carrillo and Paul Chow. The Effect of Reconfigurable Units in Superscalar Processors. In *FPGA'01: Proceedings of the 2001 ACM/SIGDA Ninth International Symposium on Field Grogrammable Gate Arrays*, pages 141–150, New York, NY, USA, 2001. ACM.

[19] S. Casselman. Virtual computing and the Virtual Computer. pages 43 –48, apr. 1993.

[20] C. Chang, J. Wawrzynek, and R.W. Brodersen. BEE2: A High-End Reconfigurable Computing System. *Design Test of Computers, IEEE*, 22(2):114 – 125, mar. 2005.

[21] Shuai Che, Jie Li, Jeremy W. Sheaffer, Kevin Skadron, and John Lach. Accelerating Compute-Intensive Applications with GPUs and FPGAs. In *SASP'08: Proceedings of the 2008 Symposium on Application Specific Processors*, pages 101–107, Washington, DC, USA, 2008. IEEE Computer Society.

[22] J. Choi. A Fast Scalable Universal Matrix Multiplication Algorithm on Distributed-Memory Concurrent Computers. In *Proceedings of the 11th International Symposium on Parallel Processing*, IPPS'97, pages 310–314, Washington, DC, USA, 1997. IEEE Computer Society.

[23] Eric S. Chung, Peter A. Milder, James C. Hoe, and Ken Mai. Single-Chip Heterogeneous Computing: Does the Future Include Custom Logic, FPGAs, and GPGPUs? In *MICRO-43: Proceedings of the 43th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2010.

[24] Charles Clos. A Study of Non-Blocking Switching Networks. In *Bell Sys. Tech. J., 32 (1953) 406-424.*, 1953.

[25] E.U. Cohler and J.E. Storer. Functionally Parallel Architecture for Array Processors. *Computer*, 14:28–36, 1981.

[26] Katherine Compton and Scott Hauck. Reconfigurable Computing: A Survey of Systems and Software. *ACM Comput. Surv.*, 34(2):171–210, 2002.

[27] Convey, Inc. `http://www.convey.com`.

[28] Adrian Cosoroaba. Designing High Efficiency DDR3 Memory Controllers with today's FPGAs. In *Proceedings of the MEMCON'11*, 2010.

[29] Cray, Inc. `http://www.cray.com`.

[30] William Dally and Brian Towles. *Principles and Practices of Interconnection Networks*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003.

[31] Nirav Dave, Kermin Fleming, Myron King, Michael Pellauer, and Muralidaran Vijayaraghavan. Hardware Acceleration of Matrix Multiplication on a Xilinx FPGA. In *Proceedings of the 5th IEEE/ACM International Conference on Formal Methods and Models for Codesign*, MEMOCODE'07, pages 97–100, Washington, DC, USA, 2007. IEEE Computer Society.

[32] John D. Davis, Charles P. Thacker, and Chen Chang. BEE3: Revitalizing Computer Architecture Research. Technical Report MSR-TR-2009-45, Microsoft Research, April 2009.

[33] Timothy A. Davis. University of Florida Sparse Matrix Collection. *NA Digest*, 92, 1994.

[34] Florent de Dinechin and Bogdan Pasca. Designing Custom Arithmetic Data Paths with FloPoCo. *IEEE Des. Test*, 28:18–27, July 2011.

[35] Andre Maurice Dehon. *Reconfigurable architectures for general-purpose computing*. PhD thesis, 1996. Supervisor-Knight,Jr., Thomas.

[36] Michael deLorimier and André DeHon. Floating-Point Sparse Matrix-Vector Multiply for FPGAs. In *Proceedings of the 2005 ACM/SIGDA 13th International Symposium on Field-Programmable Gate Arrays*, FPGA'05, pages 75–85, New York, NY, USA, 2005. ACM.

[37] Robert H. Dennard, Fritz H. Gaensslen, Hwa-nien Yu, V. Leo Rideovt, Ernest Bassous, and Andre R. Leblanc. Design of Ion-Implanted MOSFET's with Very Small Physical Dimensions. *Solid-State Circuits Newsletter, IEEE*, 12(1):38 –50, winter 2007.

[38] Harald Devos, Jan Van Campenhout, and Dirk Stroobandt. Building an Application-specific Memory Hierarchy on FPGA. *2nd HiPEAC Workshop on Reconfigurable Computing*, 2008.

[39] Yong Dou, S. Vassiliadis, G. K. Kuzmanov, and G. N. Gaydadjiev. 64-bit Floating-Point FPGA Matrix Multiplication. In *Proceedings of the 2005 ACM/SIGDA 13th International Symposium on Field-Programmable Gate Arrays*, FPGA'05, pages 86–95, New York, NY, USA, 2005. ACM.

[40] DRC Computer, Inc. `http://www.drccomputer.com`.

[41] Drew Wilson. Chameleon takes on FPGAs, ASICs. `http://www.edn.com/article/CA50551.html?partner=enews`, Oct 2000.

[42] Carl Ebeling, Darren C. Cronquist, and Paul Franklin. RaPiD - Reconfigurable Pipelined Datapath. In *FPL'96: Proceedings of the 6th International Workshop on Field-Programmable Logic, Smart Applications, New Paradigms and Compilers*, pages 126–135, London, UK, 1996. Springer-Verlag.

[43] T. El-Ghazawi, E. El-Araby, Miaoqing Huang, K. Gaj, V. Kindratenko, and D. Buell. The Promise of High-Performance Reconfigurable Computing. *Computer*, 41(2):69 –76, feb. 2008.

[44] Gerald Estrin. Reconfigurable computer origins: the UCLA fixed-plus-variable (F+V) structure computer. *IEEE Ann. Hist. Comput.*, 24(4):3–9, 2002.

[45] Daniel D. Gajski, Nikil D. Dutt, Allen C.-H. Wu, and Steve Y.-L. Lin. High Level Synthesis: Introduction to Chip and System Design. 1992.

[46] Daniel D. Gajski, Jianwen Zhu, Rainer Dmer, Andreas Gerstlauer, and Shuqing Zhao. SpecC: Specification Language and Methodology. 2000.

[47] Marco Gerards. Streaming Reduction Circuit for Sparse Matrix Vector Multiplication in FPGAs. Master's thesis, 2008.

[48] Roman Geus and Stefan Röllin. Towards a fast parallel sparse symmetric matrix-vector multiplication. *Parallel Comput.*, 27:883–896, May 2001.

[49] M. Gokhale, W. Holmes, A. Kosper, D. Kunze, D. Lopresti, S. Lucas, R. Minnich, and P. Olsen. SPLASH: A reconfigurable linear logic array. In *International Conference on Parallel Processing*, 1990.

[50] Seth Copen Goldstein, Herman Schmit, Matthew Moe, Mihai Budiu, Srihari Cadambi, R. Reed Taylor, and Ronald Laufer. PipeRench: A Coprocessor for Streaming Multimedia Acceleration. In *ISCA'99: Proceedings of the 26th Annual International Symposium on Computer Architecture*, pages 28–39, Washington, DC, USA, 1999. IEEE Computer Society.

[51] Derek B. Gottlieb, Jeffrey J. Cook, Joshua D. Walstrom, Steven Ferrera, Chi wei Wang, and Nicholas P. Carter. Clustered Programmable-Reconfigurable Processors. In *Proc. of the 1st IEEE International Conference on Field Programmable Technology (FPT), Hong Kong*, pages 134–141. IEEE, 2002.

[52] J. R. Hauser and J. Wawrzynek. Garp: a MIPS processor with a reconfigurable coprocessor. In *FCCM'97: Proceedings of the 5th IEEE Symposium on FPGA-based Custom Computing Machines*, page 12, Washington, DC, USA, 1997. IEEE Computer Society.

[53] IBM, Inc. The Netezza Data Appliance Architecture: A Platform for High Performance Data Warehousing and Analytics. `http://www.netezza.com/documents/whitepapers/Netezza_Appliance_Architecture_WP.pdf`.

[54] Eun-Jin Im, Katherine Yelick, and Richard Vuduc. Sparsity: Optimization framework for sparse matrix kernels. *Int. J. High Perform. Comput. Appl.*, 18:135–158, February 2004.

[55] Intel, Inc. `http://ark.intel.com/products/codename/42360/Stellarton`.

[56] Intel, Inc. Intel Math Kernel Library web site. `http://www.intel.com/software/products/mkl`.

[57] International Technology Roadmap for Semiconductors. `http://www.itrs.net`.

[58] Bruce Jacob and David Wang. DRAM: Architectures, Interfaces, and Systems: A Tutorial. In *Proceedings of the 2002 International Symposium on Computer Architecture*, 2002.

[59] JEDEC. `http://www.jedec.org`.

[60] Jiang Jiang, Vincent Mirian, Kam Pui Tang, Paul Chow, and Zuocheng Xing. Matrix Multiplication Based on Scalable Macro-Pipelined FPGA Accelerator Architecture. In *Proceedings of the 2009 International Conference on Reconfigurable Computing and FPGAs*, RECONFIG'09, pages 48–53, Washington, DC, USA, 2009. IEEE Computer Society.

[61] Andrew B. Kahng, Bin Li, Li-Shiuan Peh, and Kambiz Samadi. ORION 2.0: a fast and accurate NoC power and area model for early-stage design space exploration. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE'09, pages 423–428, 3001 Leuven, Belgium, Belgium, 2009. European Design and Automation Association.

[62] George Kalokerinos, Vassilis Papaefstathiou, George Nikiforos, Stamatis Kavadias, Manolis Katevenis, Dionisios Pnevmatikatos, and Xiaojun Yang. FPGA implementation of a configurable cache/scratchpad memory with virtualized user-level RDMA capability. In *Proceedings of the 9th International Conference on Systems, Architectures, Modeling and Simulation*, SAMOS'09, pages 149–156, Piscataway, NJ, USA, 2009. IEEE Press.

[63] Nachiket Kapre and Andre DeHon. Accelerating SPICE Model-Evaluation using FPGAs. volume 0, pages 37–44, Los Alamitos, CA, USA, 2009. IEEE Computer Society.

[64] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall Press, Upper Saddle River, NJ, USA, 1988.

[65] Vinay B. Y. Kumar, Siddharth Joshi, Sachin B. Patkar, and H. Narayanan. FPGA Based High Performance Double-Precision Matrix Multiplication. In *Proceedings of the 2009 22nd International Conference on VLSI Design*, pages 341–346, Washington, DC, USA, 2009. IEEE Computer Society.

[66] H. T. Kung. Memory requirements for balanced computer architectures. In *ISCA'86: Proceedings of the 13th Annual International Symposium on Computer Architecture*, pages 49–54, Los Alamitos, CA, USA, 1986. IEEE Computer Society Press.

[67] Ian Kuon and Jonathan Rose. Measuring the Gap Between FPGAs and ASICs. In *FPGA'06: Proceedings of the 2006 ACM/SIGDA 14th International Symposium on Field Programmable Gate Arrays*, pages 21–30, New York, NY, USA, 2006. ACM.

[68] Charles Eric LaForest and J. Gregory Steffan. Efficient Multi-Ported Memories for FPGAs. In *Proceedings of the 18th Annual ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA'10, pages 41–50, New York, NY, USA, 2010. ACM.

[69] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, CGO'04, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society.

[70] Benjamin C. Lee, Richard W. Vuduc, James W. Demmel, and Katherine A. Yelick. Performance Models for Evaluation and Automatic Tuning of Symmetric Sparse Matrix-Vector Multiply. In *Proceedings of the 2004 International Conference on Parallel Processing*, ICPP'04, pages 169–176, Washington, DC, USA, 2004. IEEE Computer Society.

[71] Edward C. Lin and Rob A. Rutenbar. A Multi-FPGA 10x-Real-Time High-Speed Search Engine for a 5000-word Vocabulary Speech Recognizer. In *Proceeding of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA'09, pages 83–92, New York, NY, USA, 2009. ACM.

[72] Liu Ling, Neal Oliver, Chitlur Bhushan, Wang Qigang, Alvin Chen, Shen Wenbo, Yu Zhihong, Arthur Sheiman, Ian McCallum, Joseph Grecco, Henry Mitchel, Liu Dong, and Prabhat Gupta. High-performance, Energy-efficient Platforms Using In-Socket FPGA Accelerators. In *FPGA'09: Proceeding of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 261–264, New York, NY, USA, 2009. ACM.

[73] Ken Mai, Tim Paaske, Nuwan Jayasena, Ron Ho, William J. Dally, and Mark Horowitz. Smart Memories: a modular reconfigurable architecture. In *ISCA'00: Proceedings of the*

27th Annual International Symposium on Computer Architecture, pages 161–171, New York, NY, USA, 2000. ACM.

[74] A. Marquardt, V. Betz, and J. Rose. Speed and Area Tradeoffs in Cluster-Based FPGA Architectures. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 8(1):84 –93, February 2000.

[75] Bingfeng Mei, Serge Vernalde, Diederik Verkest, Hugo De Man, and Rudy Lauwereins. ADRES: An Architecture with Tightly Coupled VLIW Processor and Coarse-Grained Reconfigurable Matrix. In *Field-Programmable Logic and Applications*, volume 2778 of *Lecture Notes in Computer Science*, pages 61–70. Springer Berlin / Heidelberg, 2003.

[76] Mentor Graphics. Catapult C. `http://www.mentor.com/esl`, 2009.

[77] E. Mirsky and A. DeHon. MATRIX: A Reconfigurable Computing Architecture with Configurable Instruction Distribution and Deployable Resources. In *FPGAs for Custom Computing Machines, 1996. Proceedings. IEEE Symposium on*, pages 157 –166, April 1996.

[78] Naohito Nakasato. A fast GEMM implementation on the cypress GPU. *SIGMETRICS Perform. Eval. Rev.*, 38:50–55, March 2011.

[79] Pradeep Nalabalapu and Ron Sass. Bandwidth Management with a Reconfigurable Data Cache. In *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Workshop 3 - Volume 04*, IPDPS'05, pages 159.1–, Washington, DC, USA, 2005. IEEE Computer Society.

[80] Nallatech. `http://www.nallatech.com`.

[81] T. Ngai, J. Rose, and S.J.E. Wilton. An SRAM-programmable field-configurable memory. In *Custom Integrated Circuits Conference, 1995., Proceedings of the IEEE 1995*, May 1995.

[82] Nvidia, Inc. CUDA CUBLAS Library 2.0. `http://developer.download.nvidia.com/compute/cuda/2_0/docs/CUBLAS_Library_2.0.pdf`.

[83] Michael K. Papamichael. Fast Scalable FPGA-based Network-on-Chip Simulation Models. In *Proceedings of the 9th IEEE/ACM International Conference on Formal Methods and Models for Codesign*, MEMOCODE'11, 2011.

[84] Ali Pinar and Michael T. Heath. Improving performance of sparse matrix-vector multiplication. In *Proceedings of the 1999 ACM/IEEE conference on Supercomputing (CDROM)*, Supercomputing'99, New York, NY, USA, 1999. ACM.

[85] Daniel S. Poznanovic. Application Development on the SRC Computers, Inc. Systems. In *IPDPS'05: Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Papers*, page 78.1, Washington, DC, USA, 2005. IEEE Computer Society.

[86] Rahul Razdan and Michael D. Smith. A High-Performance Microarchitecture with Hardware-Programmable Functional Units. In *MICRO'94: Proceedings of the 27th Annual International Symposium on Microarchitecture*, pages 172–180, New York, NY, USA, 1994. ACM.

[87] Kentaro Sano, Wang Luzhou, Yoshiaki Hatsuda, Takanori Iizuka, and Satoru Yamamoto. FPGA-Array with Bandwidth-Reduction Mechanism for Scalable and Power-Efficient Numerical Simulations Based on Finite Difference Methods. *ACM Trans. Reconfigurable Technol. Syst.*, 3:21:1–21:35, November 2010.

[88] C. Scheurich and M. Dubois. The design of a lockup-free cache for high-performance multiprocessors. In *Proceedings of the 1988 ACM/IEEE conference on Supercomputing*, Supercomputing'88, pages 352–359, Los Alamitos, CA, USA, 1988. IEEE Computer Society Press.

[89] H. Schmit, D. Whelihan, A. Tsai, M. Moe, B. Levine, and R. Reed Taylor. PipeRench: A Virtualized Programmable Datapath in 0.18 Micron Technology. In *Custom Integrated Circuits Conference, 2002. Proceedings of the IEEE 2002*, pages 63 – 66, 2002.

[90] Silicon Graphics, Inc. Extraordinary acceleration of workflows with reconfigurable application-specific computing from SGI. `http://www.sgi.com/pdfs/3721.pdf`, 2004.

[91] H. Singh, Ming-Hau Lee, Guangming Lu, F.J. Kurdahi, N. Bagherzadeh, and E.M. Chaves Filho. MorphoSys: An Integrated Reconfigurable System for Data-parallel and Computation-intensive Applications. *Computers, IEEE Transactions on*, 49(5):465 –481, May 2000.

[92] James E. Smith. Decoupled access/execute computer architectures. *SIGARCH Comput. Archit. News*, 10:112–119, April 1982.

[93] Junqing Sun, Gregory Peterson, and Olaf Storaasli. Sparse Matrix-Vector Multiplication Design on FPGAs. In *Proceedings of the 15th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 349–352, Washington, DC, USA, 2007. IEEE Computer Society.

[94] Michael Bedford Taylor, Walter Lee, Jason Miller, David Wentzlaff, Ian Bratt, Ben Greenwald, Henry Hoffmann, Paul Johnson, Jason Kim, James Psota, Arvind Saraf, Nathan Shnidman, Volker Strumpen, Matt Frank, Saman Amarasinghe, and Anant Agarwal. Evaluation of the Raw Microprocessor: An Exposed-Wire-Delay Architecture for ILP and Streams. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, ISCA'04, pages 2–, Washington, DC, USA, 2004. IEEE Computer Society.

[95] O. Temam and W. Jalby. Characterizing the behavior of sparse algorithms on caches. In *Proceedings of the 1992 ACM/IEEE conference on Supercomputing*, Supercomputing'92, pages 578–587, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.

[96] David Barrie Thomas, Lee Howes, and Wayne Luk. A Comparison of CPUs, GPUs, FPGAs, and Massively Parallel Processor Arrays for Random Rumber Generation. In *FPGA'09: Proceeding of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 63–72, New York, NY, USA, 2009. ACM.

[97] David Tarjan Shyamkumar Thoziyoor, David Tarjan, and Shyamkumar Thoziyoor. Cacti 4.0. Technical Report HPL-2006-86, HP Labs, 2006.

[98] Xiang Tian and Khaled Benkrid. High-Performance Quasi-Monte Carlo Financial Simulation: FPGA vs. GPP vs. GPU. *ACM Trans. Reconfigurable Technol. Syst.*, 3:26:1–26:22, November 2010.

[99] Hans Vandierendonck and Koen De Bosschere. Xor-based hash functions. *IEEE Trans. Comput.*, 54:800–812, July 2005.

[100] Brendan Vastenhouw and Rob H. Bisseling. A two-dimensional data distribution method for parallel sparse matrix-vector multiplication. *SIAM Rev.*, 47:67–95, January 2005.

[101] Ganesh Venkatesh, Jack Sampson, Nathan Goulding, Saturnino Garcia, Vladyslav Bryksin, Jose Lugo-Martinez, Steven Swanson, and Michael Bedford Taylor. Conservation Cores: Reducing the Energy of Mature Computations. In *ASPLOS'10: Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 205–218, New York, NY, USA, 2010. ACM.

[102] Francisco-Javier Veredas, M. Scheppler, W. Moffat, and Bingfeng Mei. Custom Implementation of the Coarse-grained Reconfigurable ADRES Architecture for Multimedia Purposes. In *Field Programmable Logic and Applications, 2005. International Conference on*, pages 106 – 111, 2005.

[103] Victor Podlozhnyuk, Nvidia Inc. Black-Scholes Option Pricing, 2007.

[104] Richard Vuduc, Shoaib Kamil, Jen Hsu, Rajesh Nishtala, James W. Demmel, and Katherine A. Yelick. Automatic performance tuning and analysis of sparse triangular solve. In *In ICS 2002: Workshop on Performance Optimization via High-Level Languages and Libraries*, 2002.

[105] Richard Wilson Vuduc. *Automatic performance tuning of sparse matrix kernels*. PhD thesis, 2003. AAI3121741.

[106] Jean E. Vuillemin, Patrice Bertin, Didier Roncin, Mark Shand, Hervé H. Touati, and Philippe Boucard. Programmable active memories: reconfigurable systems come of age. *IEEE Trans. Very Large Scale Integr. Syst.*, 4:56–69, March 1996.

[107] David Wentzlaff, Patrick Griffin, Henry Hoffmann, Liewei Bao, Bruce Edwards, Carl Ramey, Matthew Mattina, Chyi-Chang Miao, John F. Brown III, and Anant Agarwal. On-Chip Interconnection Architecture of the Tile Processor. *IEEE Micro*, 27(5):15–31, 2007.

[108] Samuel Williams, Leonid Oliker, Richard Vuduc, John Shalf, Katherine Yelick, and James Demmel. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. *Parallel Comput.*, 35:178–194, March 2009.

[109] M. J. Wirthlin. A Dynamic Instruction Set Computer. In *FCCM'95: Proceedings of the IEEE Symposium on FPGA's for Custom Computing Machines*, page 99, Washington, DC, USA, 1995. IEEE Computer Society.

[110] Inc. Xilinx. `http://www.xilinx.com/products/devkits/EK-V6-ML605-G.htm`.

[111] Inc. Xilinx. Memory Interface Solutions. `http://www.xilinx.com/support/documentation/ip_documentation/ug086.pdf`.

[112] Xilinx, Inc. `http://www.xilinx.com`.

[113] Xilinx, Inc. Spartan-6 FPGA Memory Controller. `http://www.xilinx.com/support/documentation/user_guides/ug388.pdf`.

[114] Xilinx, Inc. Virtex-II Platform FPGAs: Complete Data Sheet, 2005.

[115] Xilinx, Inc. Virtex-6 Family Overview, 2009.

[116] Xilinx, Inc. Virtex-7 Series Overview, 2010.

[117] Inc. XtremeData. `http://www.xtremedata.com`.

[118] Yoshiki Yamaguchi, Kuen Hung Tsoi, and Wayne Luk. A Comparison of FPGAs, GPUS and CPUS for Smith-Waterman Algorithm. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA'11, pages 281–281, New York, NY, USA, 2011. ACM.

[119] Zhi Alex Ye, Andreas Moshovos, Scott Hauck, and Prithviraj Banerjee. CHIMAERA: A High-Performance Architecture with a Tightly-Coupled Reconfigurable Functional Unit. In *ISCA'00: Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 225–235, New York, NY, USA, 2000. ACM.

[120] Katherine Yelick, Dan Bonachea, Wei-Yu Chen, Phillip Colella, Kaushik Datta, Jason Duell, Susan L. Graham, Paul Hargrove, Paul Hilfinger, Parry Husbands, Costin Iancu, Amir Kamil, Rajesh Nishtala, Jimmy Su, Welco Michael, and Tong Wen. Productivity and Performance Using Partitioned Global Address Space Languages. In *Proceedings of the 2007 International Workshop on Parallel Symbolic Computation*, PASCO'07, pages 24–32, New York, NY, USA, 2007. ACM.

[121] Peter Yiannacouras and Jonathan Rose. A parameterized automatic cache generator for FPGAs. In *Proc. Field-Programmable Technology (FPT*, pages 324–327, 2003.

[122] Steven P. Young. FPGA architecture having RAM blocks with programmable word length and width and dedicated address and data lines, United States Patent No. 5,933,023. 1996.

[123] Ling Zhuo, G.R. Morris, and V.K. Prasanna. High-Performance Reduction Circuits Using Deeply Pipelined Operators on FPGAs. *Parallel and Distributed Systems, IEEE Transactions on*, 18(10):1377 –1392, oct. 2007.

[124] Ling Zhuo and Viktor K. Prasanna. Sparse matrix-vector multiplication on FPGAs. In *In Proceedings of the ACM International Symposium on Field Programmable Gate Arrays*, pages 63–74. ACM Press, 2005.