The Heterogeneous Block Architecture

A Flexible Substrate for Building Energy-Efficient High-Performance Cores

Chris Fallin[1] Chris Wilkerson[2] Onur Mutlu[1]

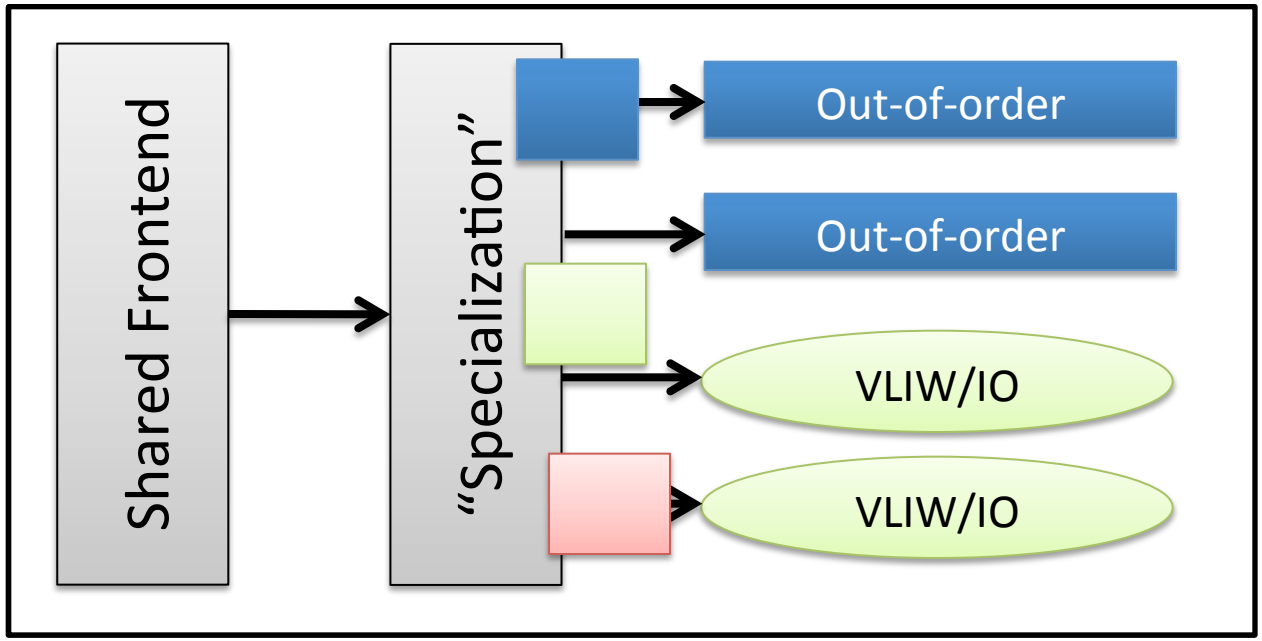
[1] Carnegie Mellon University
[2] Intel Corporation

SAFARI Carnegie Mellon intel

Executive Summary

- Problem: General purpose core design is a compromise between performance and energy-efficiency
- Our Goal: Design a core that achieves high performance and high energy efficiency at the same time
- Two New Observations: 1) Applications exhibit fine-grained heterogeneity in regions of code, 2) A core can exploit this heterogeneity by grouping tens of instructions into atomic blocks and executing each block in the best-fit backend
- Heterogeneous Block Architecture (HBA): 1) Forms atomic blocks of code, 2) Dynamically determines the best-fit (most efficient) backend for each block, 3) Specializes the block for that backend
- Initial HBA Implementation: Chooses from Out-of-order, VLIW or In-order backends, with simple schedule stability and stall heuristics
- Results: HBA provides higher efficiency than four previous designs at negligible performance loss; HBA enables new tradeoff points

Picture of HBA



Talk Agenda

- Background and Motivation
- Two New Observations
- The Heterogeneous Block Architecture (HBA)
- An Initial HBA Design
- Experimental Evaluation
- Conclusions

Competing Goals in Core Design

- High performance and high energy-efficiency
  - Difficult to achieve both at the same time

- High performance: Sophisticated features to extract it
  - Out-of-order execution (OoO), complex branch prediction, wide instruction issue, …
- High energy efficiency: Use of only features that provide the highest efficiency for each workload
  - Adaptation of resources to different workload requirements

- Today's high-performance designs: Features may **not** yield high performance, but every piece of code pays for their energy penalty

Principle: Exploiting Heterogeneity

- Past observation 1: Workloads have different characteristics at coarse granularity (thousands to millions of instructions)
  - Each workload or phase may require different features

- **Can We Design a More Energy-Efficient Core?**

- Past coarse-grained heterogeneous designs
  - provide higher energy efficiency
  - at slightly lower performance vs. a homogeneous OoO core

Talk Agenda

- Background and Motivation
- Two New Observations
- The Heterogeneous Block Architecture (HBA)
- An Initial HBA Design
- Experimental Evaluation
- Conclusions

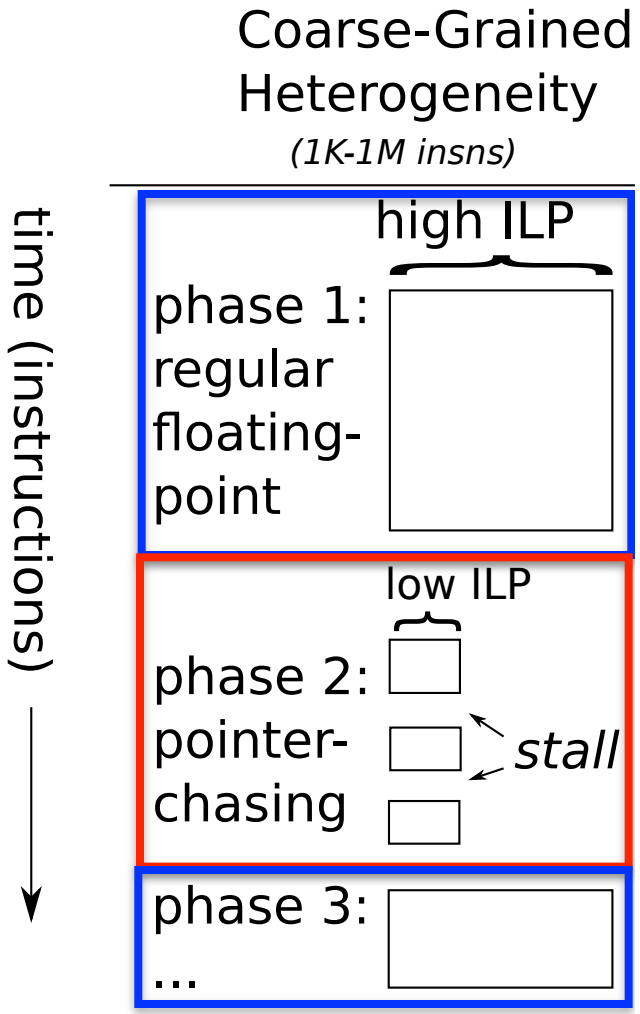
Two Key Observations

- Fine-Grained Heterogeneity of Application Behavior
  - Small chunks (10s or 100s of instructions) of code have different execution characteristics
    - Some instruction chunks always have the same instruction schedule in an out-of-order processor
    - Nearby chunks can have different schedules

- Atomic Block Execution with Multiple Execution Backends
  - A core can exploit fine-grained heterogeneity with
    - Having multiple execution backends
    - Dividing the program into atomic blocks
    - Executing each block in the best-fit backend

Fine-Grained Heterogeneity

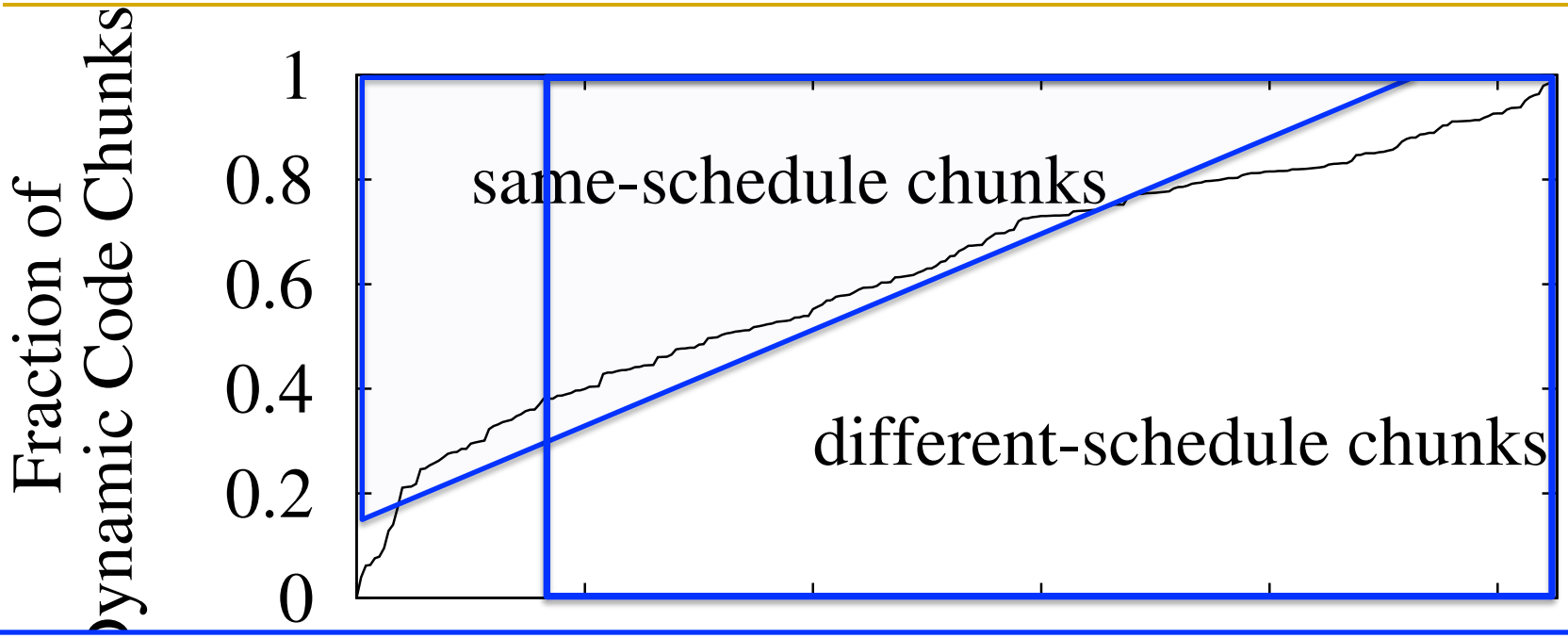
- Small chunks of code have different execution characteristics



Instruction Schedule Stability Varies at Fine Grain

- Observation 1: Some regions of code are scheduled in the same instruction scheduling order in an OoO engine across different dynamic instances
- Observation 2: Stability of instruction schedules of (nearby) code regions can be very different
- An experiment:
  - Same-schedule chunk: A chunk of code that has the same schedule it had in its previous dynamic instance
  - Examined all chunks of <=16 instructions in 248 workloads
  - Computed the fraction of "same schedule chunks" in each workload

Instruction Schedule Stability Varies at Fine Grain



3. Temporally-adjacent chunks can have different schedule stability

→ Need a core that can switch quickly between multiple schedulers

1. Many chunks have the same schedule in their previous instances

→ Can reuse the previous instruction scheduling order the next time

2. Stability of instruction schedule of chunks can be very different

→ Need a core that can dynamically schedule insts. in some chunks

Two Key Observations

- Fine-Grained Heterogeneity of Application Behavior
  - Small chunks (10s or 100s of instructions) of code have different execution characteristics
    - Some instruction chunks always have the same schedule
    - Nearby chunks can have different schedules

- Atomic Block Execution with Multiple Execution Backends
  - A core can exploit fine-grained heterogeneity with
    - Having multiple execution backends
    - Dividing the program into atomic blocks
    - Executing each block in the best-fit backend

Atomic Block Execution w/ Multiple Backends

- Fine grained heterogeneity can be exploited by a core that:
  - Has **multiple (specialized) execution backends**
  - Divides the program into **atomic (all-or-none) blocks**
  - Executes each atomic block in the **best-fit backend**

Atomicity enables specialization of a block for a backend (can freely reorder/rewrite/eliminate instructions in an atomic block)

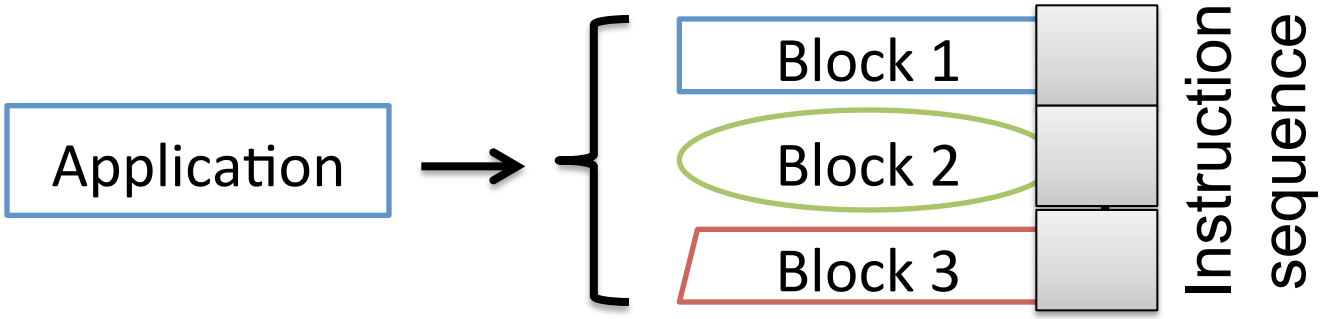
Talk Agenda

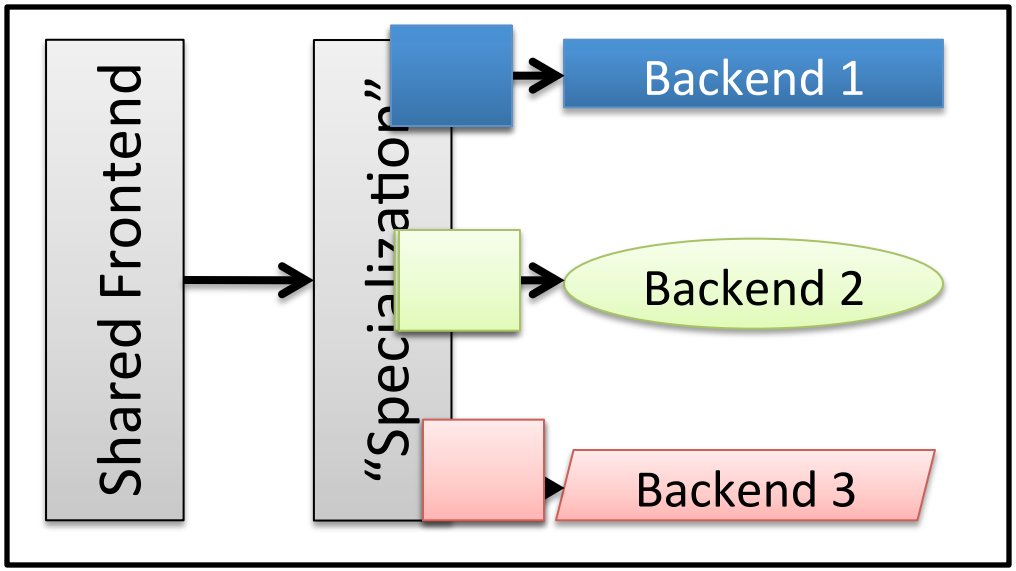
- Background and Motivation
- Two New Observations
- The Heterogeneous Block Architecture (HBA)
- An Initial HBA Design
- Experimental Evaluation
- Conclusions

HBA: Principles

- Multiple different execution backends
  - Customized for different block execution requirements: OoO, VLIW, in-order, SIMD, …
  - Can be simultaneously active, executing different blocks
  - Enables efficient exploitation of fine-grained heterogeneity
- Block atomicity
  - Either the entire block executes or none of it
  - Enables specialization (reordering and rewriting) of instructions freely within the block to fit a particular backend
- Exploit stable block characteristics (instruction schedules)
  - E.g., reuse schedule learned by the OoO backend in VLIW/IO
  - Enables high efficiency by adapting to stable behavior

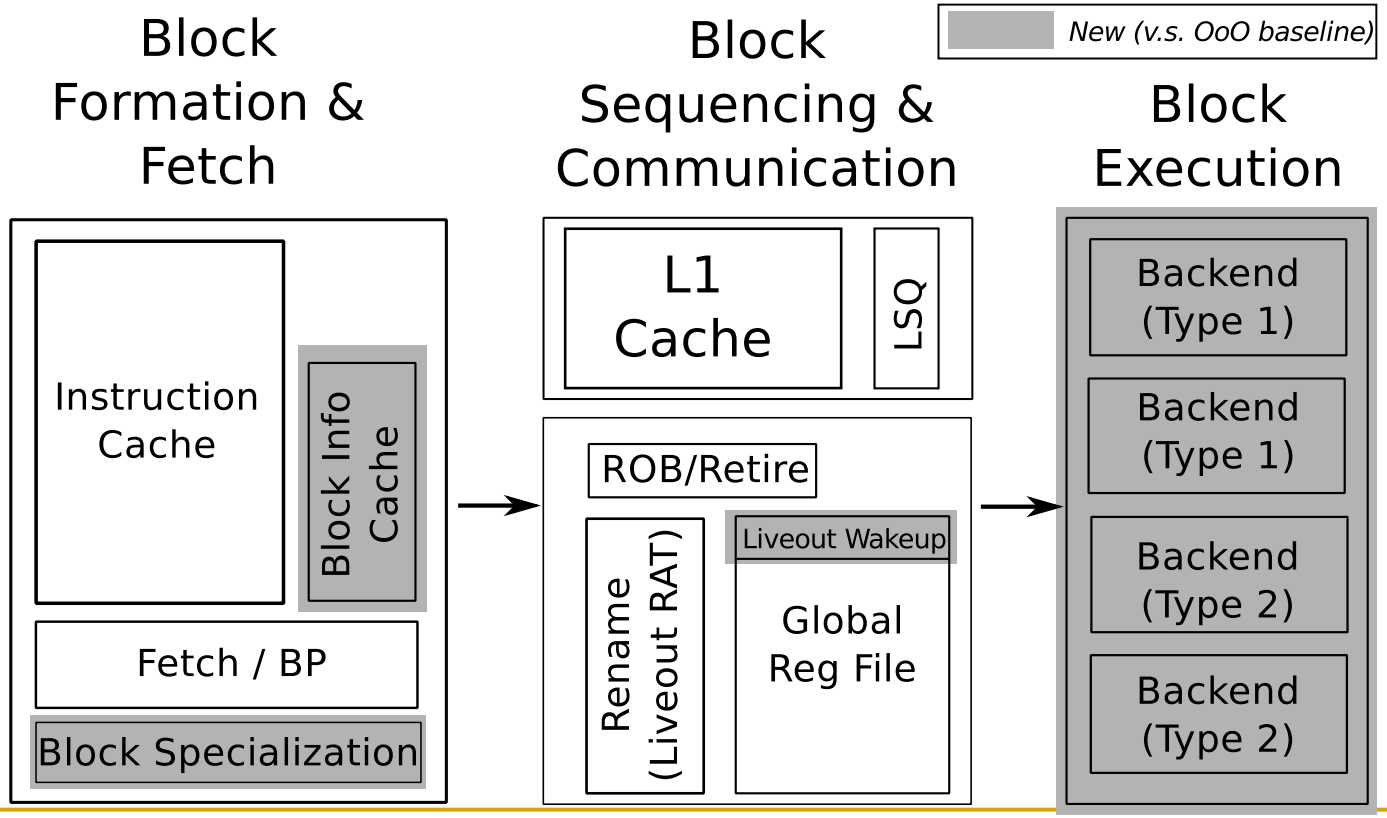
HBA Operation at High Level



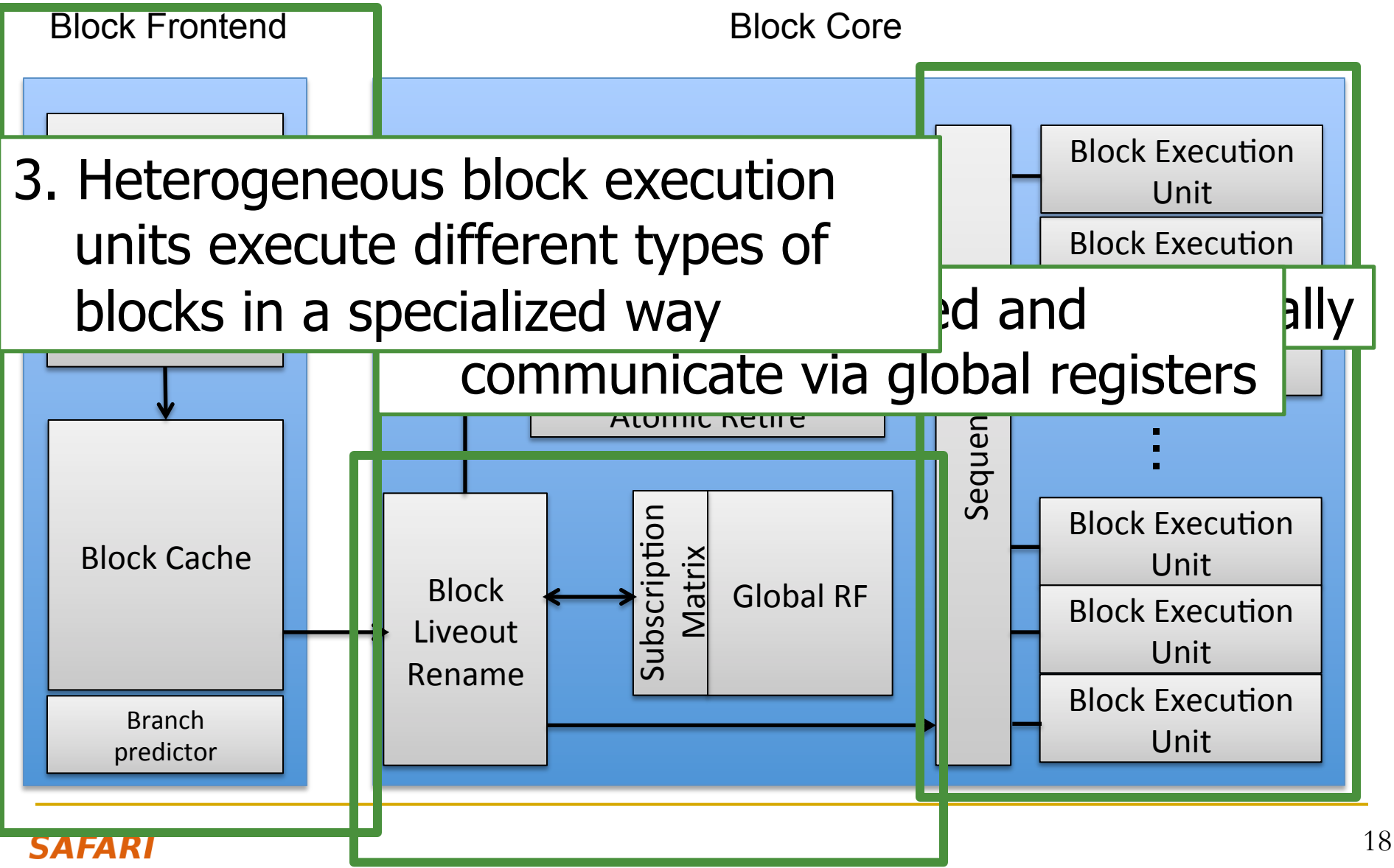


HBA Design: Three Components (I)

- Block formation and fetch
- Block sequencing and value communication
- Block execution



HBA Design: Three Components (II)



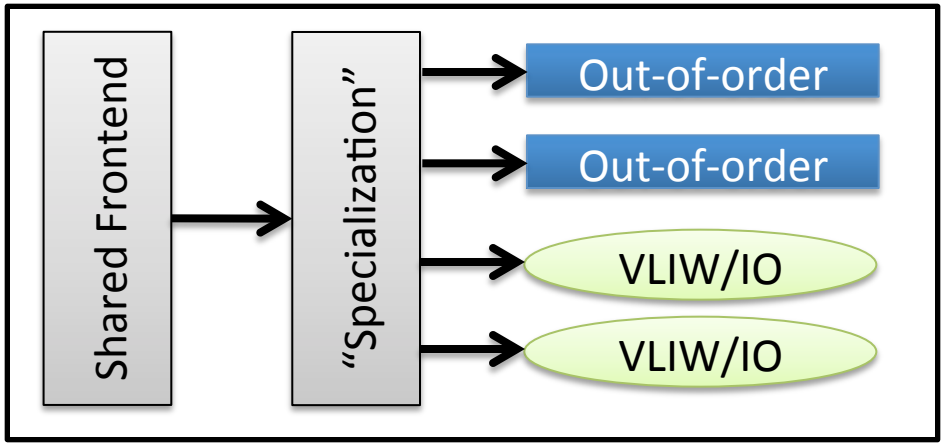
Talk Agenda

- Background and Motivation
- Two New Observations
- The Heterogeneous Block Architecture (HBA)
- An Initial HBA Design
- Experimental Evaluation
- Conclusions

An Example HBA Design: OoO/VLIW

- Block formation and fetch
- Block sequencing and value communication
- Block execution

- Two types of backends
  - OoO scheduling
  - VLIW/in-order scheduling



Block Formation and Fetch

- Atomic blocks are microarchitectural and dynamically formed
- Block info cache (BIC) stores metadata about a formed block
  - Indexed with PC and branch path of a block (ala Trace Caches)
  - Metadata info: backend type, instruction schedule, …

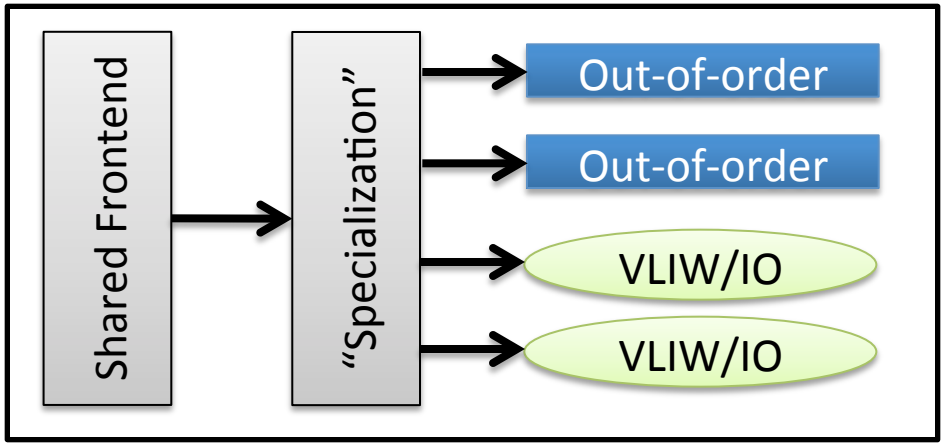
- Frontend fetches instructions/metadata from I-cache/BIC
- If miss in BIC, instructions are sent to OoO backend
- Otherwise, buffer instructions until the entire block is fetched

- The first time a block is executed on the OoO backend, its' OoO version is formed (& OoO schedule/names stored in BIC)
  - Max 16 instructions, ends in a hard-to-predict or indirect branch

An Example HBA Design: OoO/VLIW

- Block formation and fetch
- Block sequencing and value communication
- Block execution

- Two types of backends
  - OoO scheduling
  - VLIW/in-order scheduling



Block Sequencing and Communication

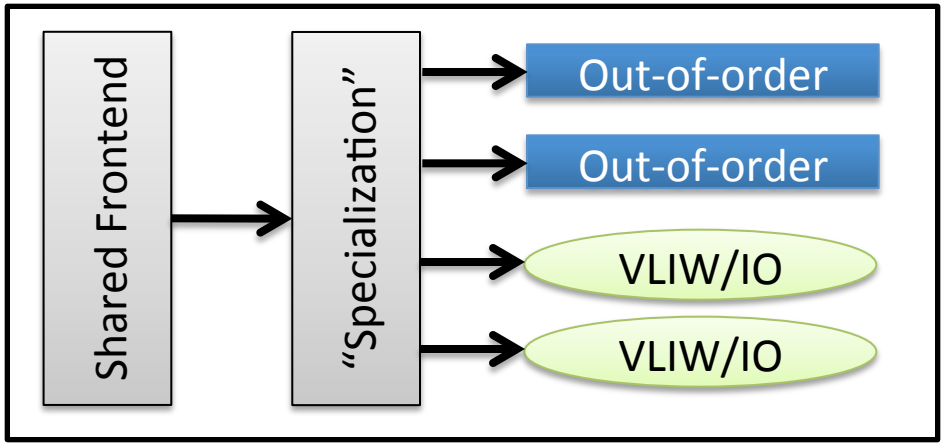
- Block Sequencing
  - Block based reorder buffer for in-order block sequencing
  - All subsequent blocks squashed on a branch misprediction
  - Current block squashed on an exception or intra-block misprediction
  - Single-instruction sequencing from non-optimized code upon an exception in a block to reach the exception point

- Value Communication
  - Across blocks: via the Global Register File
  - Within block: via the Local Register File
  - Only liveins and liveouts to a block need to be renamed and allocated global registers [Sprangle & Patt, MICRO 1994]
    - Reduces energy consumption of register file and bypass units

An Example HBA Design: OoO/VLIW

- Block formation and fetch
- Block sequencing and value communication
- Block execution

- Two types of backends
  - OoO scheduling
  - VLIW/in-order scheduling



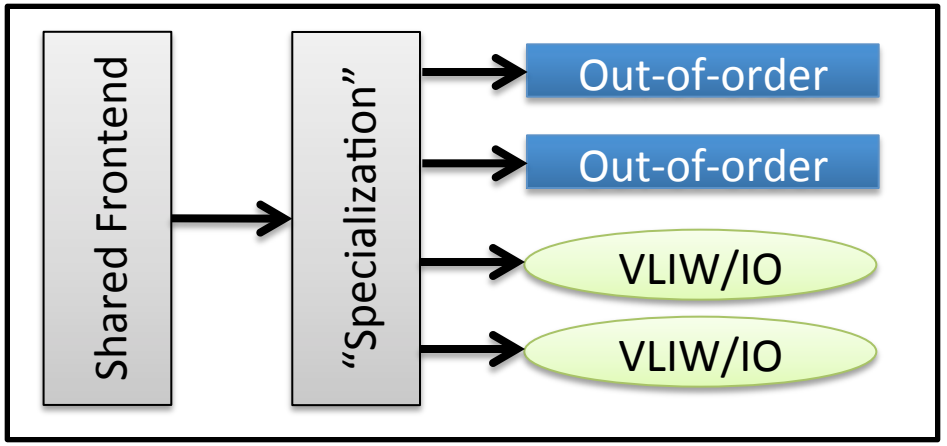
Block Execution

- Each backend contains
  - Execution units, scheduler, local register file
- Each backend receives
  - A block specialized for it (at block dispatch time)
  - Liveins for the executing block (as they become available)
- A backend executes the block
  - As specified by its logic and any information from the BIC
- Each backend produces
  - Liveouts to be written to the Global RegFile (as available)
  - Branch misprediction, exception results, block completion signal to be sent to the Block Sequencer
- Memory operations are handled via a traditional LSQ

An Example HBA Design: OoO/VLIW

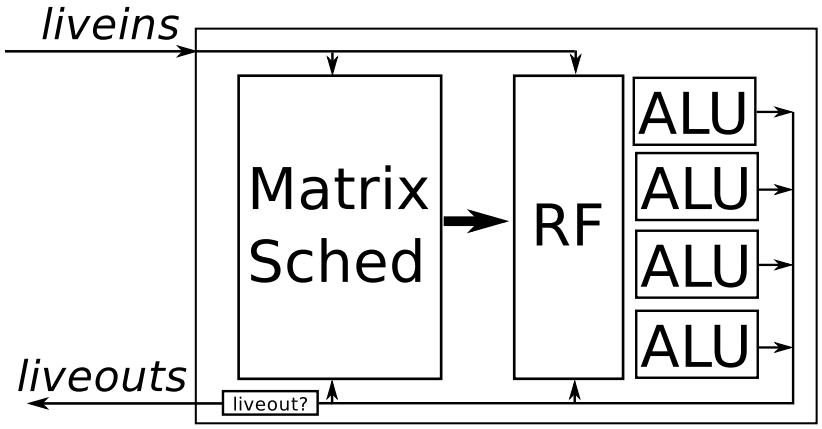
- Block formation and fetch
- Block sequencing and value communication
- Block execution

- Two types of backends
  - OoO scheduling
  - VLIW/in-order scheduling

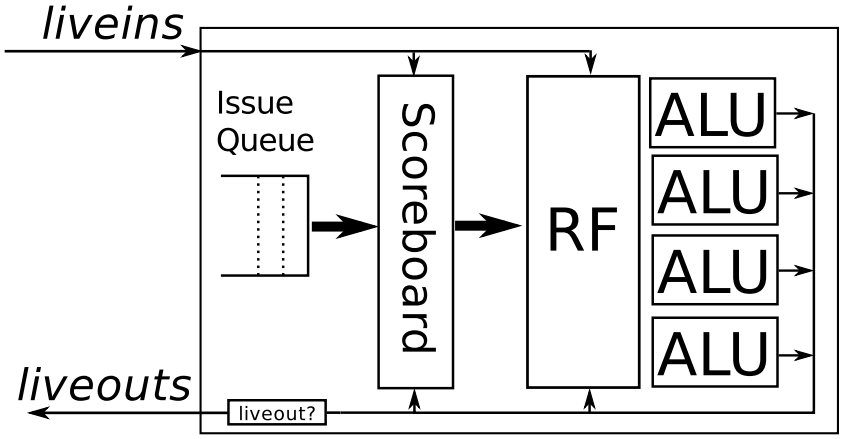


OoO and VLIW Backends

- Two customized backend types (physically, they are unified)



Out-of-order

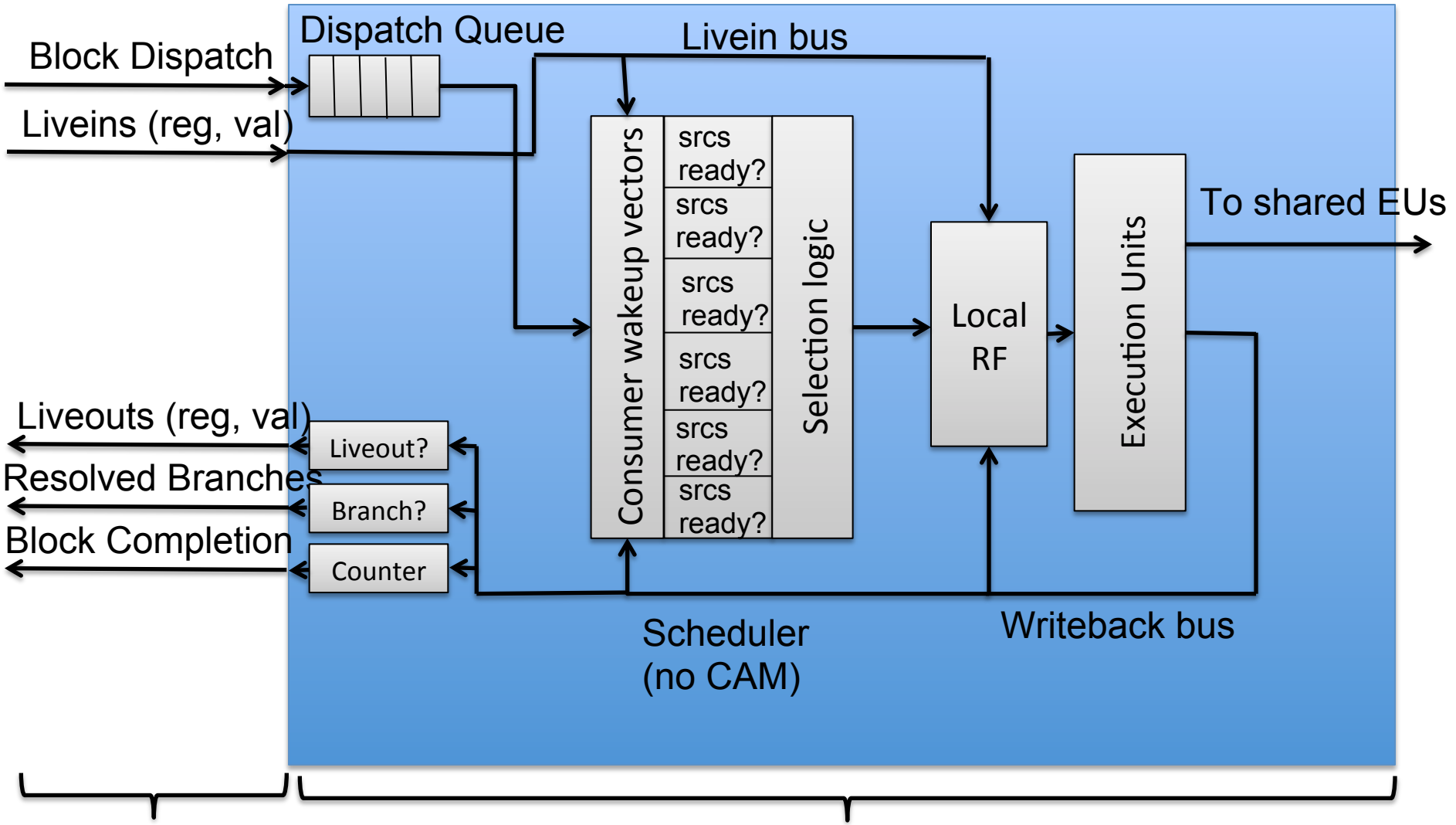


VLIW/In-order

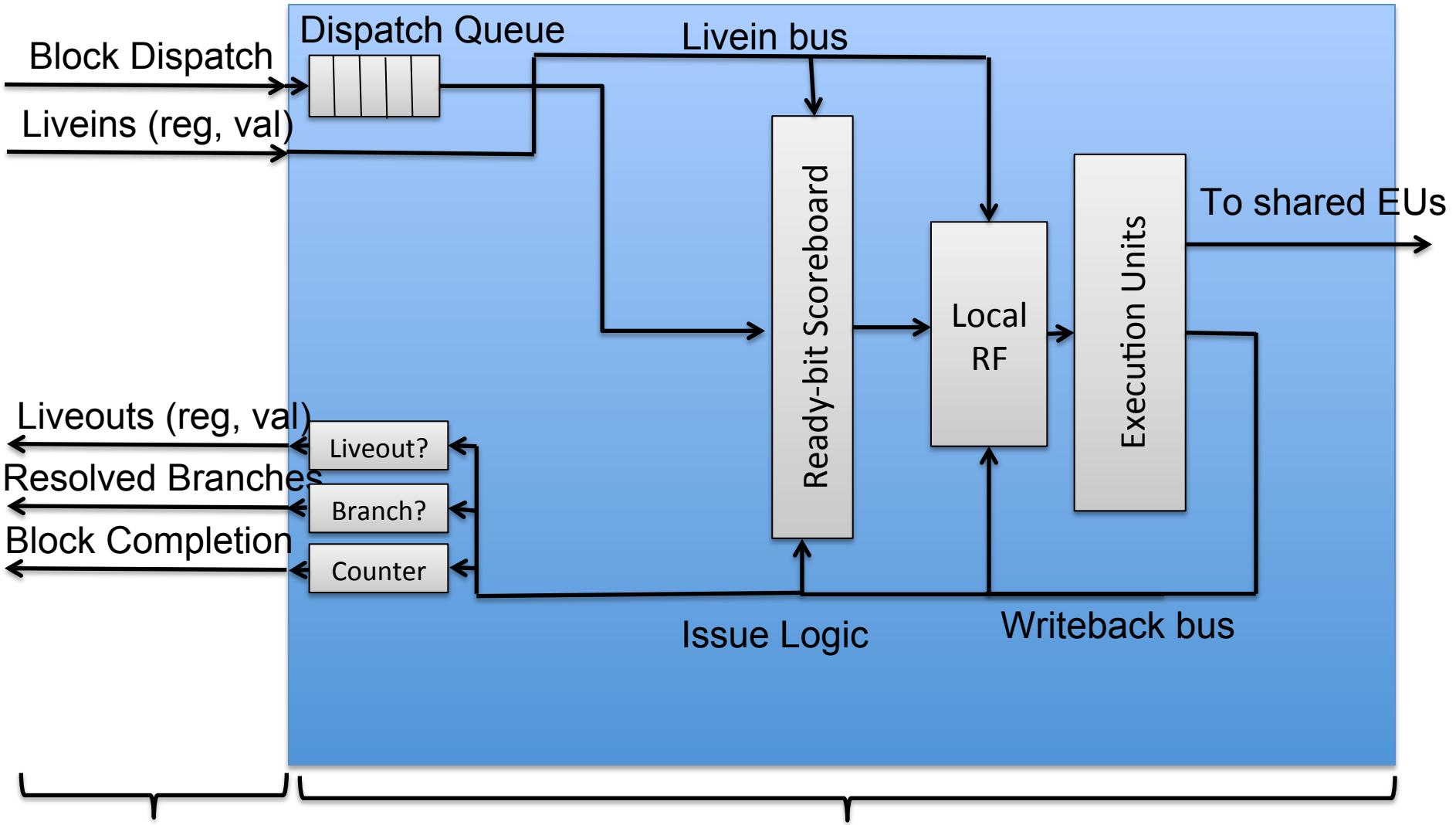
- Dynamic scheduling with matrix scheduler
- Renaming performed before the backend (block specialization logic)
- No renaming or matrix computation logic (done for previous instance of block)
- No need to maintain per-instruction order

- Static scheduling
- Stall-on-use policy
- VLIW blocks specialized by OoO backends
- No per-instruction order

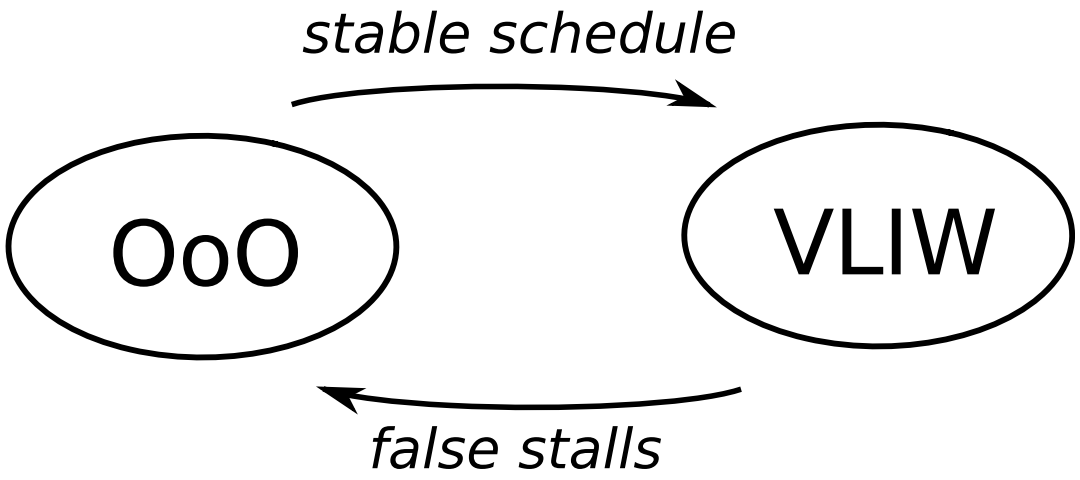
OoO Backend


Block Dispatch
Liveins (reg, val)
Dispatch Queue
Livein bus
Consumer wakeup vectors
srcs ready?
srcs ready?
srcs ready?
srcs ready?
srcs ready?
srcs ready?
Selection logic
Local RF
Execution Units
To shared EUs
Liveouts (reg, val)
Resolved Branches
Block Completion
Liveout?
Branch?
Counter
Scheduler (no CAM)
Writeback bus
Generic Interface
Dynamically-scheduled implementation

VLIW/In-Order Backend



Backend Switching and Specialization



- OoO to VLIW: If dynamic schedule is the same for N previous executions of the block on the OoO backend:
  - Record the schedule in the Block Info Cache
  - Mark the block to be executed on the VLIW backend
- VLIW to OoO: If there are too many *false stalls* in the block on the VLIW backend
  - Mark the block to be executed on the OoO backend

Talk Agenda

- Background and Motivation
- Two New Observations
- The Heterogeneous Block Architecture (HBA)
- An Initial HBA Design
- Experimental Evaluation
- Conclusions

Experimental Methodology

- Models
  - Cycle-accurate execution-driven x86-64 simulator
    - ISA remains unchanged
  - Energy model with modified McPAT [Li+ MICRO'09]
    - various leakage parameters investigated

- Workloads
  - 184 different representative execution checkpoints
  - SPEC CPU2006, Olden, MediaBench, Firefox, Ffmpeg, Adobe Flash player, MySQL, lighttpd web server, LaTeX, Octave, an x86 simulator

Simulation Setup

Parameter	Setting
	Baseline Core:
Fetch Unit	ISL-TAGE [55]; 64-entry return stack; 64K-entry BTB; 8-cycle restart latency; 4-wide fetch
Instruction Cache	32KB, 4-way, 64-byte blocks
Window Size	256-μop ROB, 320-entry physical register file (PRF), 96-entry matrix scheduler
Execution Units	4-wide; 4 ALUs, 1 MUL, 3 FPUs, 2 branch units, 2 load pipes, 1 store address and data pipe each
Memory Unit	96-entry load queue (LQ), 48-entry store queue (SQ)
L1/L2 Caches	64KB, 4-way, 5-cycle L1; 1MB, 16-way, 15-cycle L2; 64-byte blocks
DRAM	200-cycle latency; stream prefetcher, 16 streams
	Heterogeneous Block Architecture (HBA):
Block Size	16 μops, 16 liveins, 16 liveouts max
Fetch Unit	Baseline + 256-block info cache, 64 bytes/block
Global RF	256 entry; 16 rd/8 wr ports; 2-cyc. inter-backend comm.
Instruction Window	16-entry Block ROB
Backends	16 unified backends (OoO- or VLIW-mode)
OoO backend	4-wide, 4 ALUs, 16-entry local RF, 16-entry scheduler
VLIW backend	4-wide, 4 ALUs, 16-entry local RF, scoreboard sched.
Shared Execution Units	3 FPUs, 1 MUL, 2 load, 1 store address/1 store data; 2-cycle roundtrip penalty for use
LQ/SQ/L1D/DRAM	Same as baseline

Four Core Comparison Points

- Out-of-order core with large, monolithic backend
- Clustered out-of-order core with split backend [Farkas+ MICRO'97]
  - Clusters of <scheduler, register file, execution units>
- Coarse-grained heterogeneous design [Lukefahr+ MICRO'12]
  - Combines out-of-order and in-order cores
  - Switches mode for coarse-grained intervals based on a performance model (ideal in our results)
  - Only one mode can be active: in-order or out-of-order
- Coarse-grained heterogeneous design, with clustered OoO core

Key Results: Power, EPI, Performance

Averaged across 184 workloads

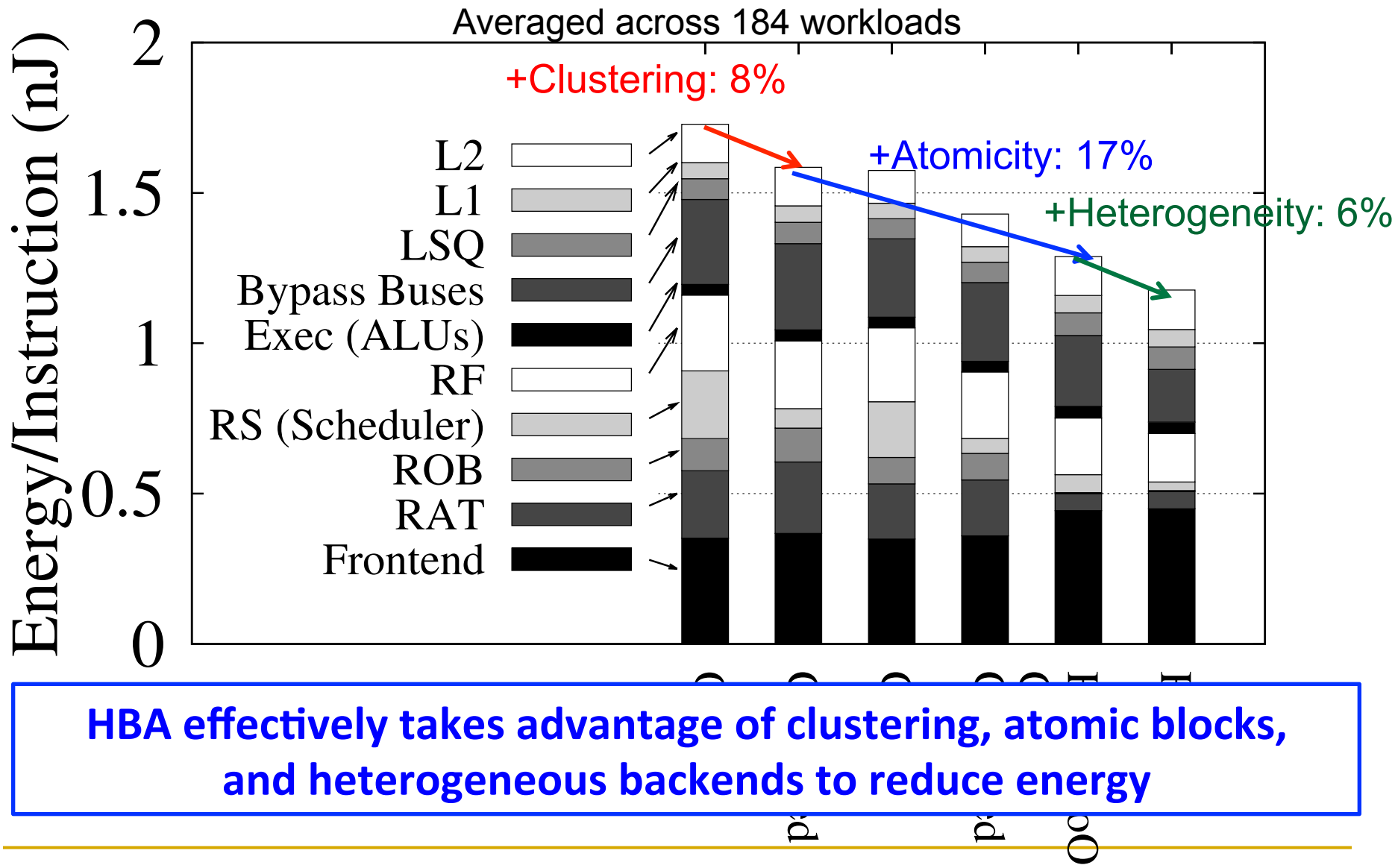
Row	Configuration	Δ Power	Δ EPI	Δ IPC
1	4-wide OoO (Baseline)	—	—	—
2	4-wide Clustered OoO [18]	-11.5%	-8.3%	-1.4%
3	Coarse-grained [37]	-5.4%	-8.9%	-1.2%
4	Coarse-grained, Clustered	-16.9%	-17.3%	-2.8%
5	HBA, OoO Backends Only	-28.7%	-25.5%	+0.4%
6	HBA, OoO/VLIW	-36.4%	-31.9%	-1.0%

- HBA greatly reduces power and energy-per-instruction (>30%)
- HBA performance is within 1% of monolithic out-of-order core
- HBA's performance higher than coarse-grained hetero. core
- HBA is the most power- and energy-efficient design among the five different core configurations

Energy Analysis (I)

- HBA reduces energy/power by concurrently
  - Decoupling Execution Backends to Simplify the Core (Clustering)
  - Exploiting Block Atomicity to Simplify the Core (Atomicity)
  - Exploiting Heterogeneous Backends to Specialize (Heterogeneity)
- What is the relative energy benefit of each when applied successively?

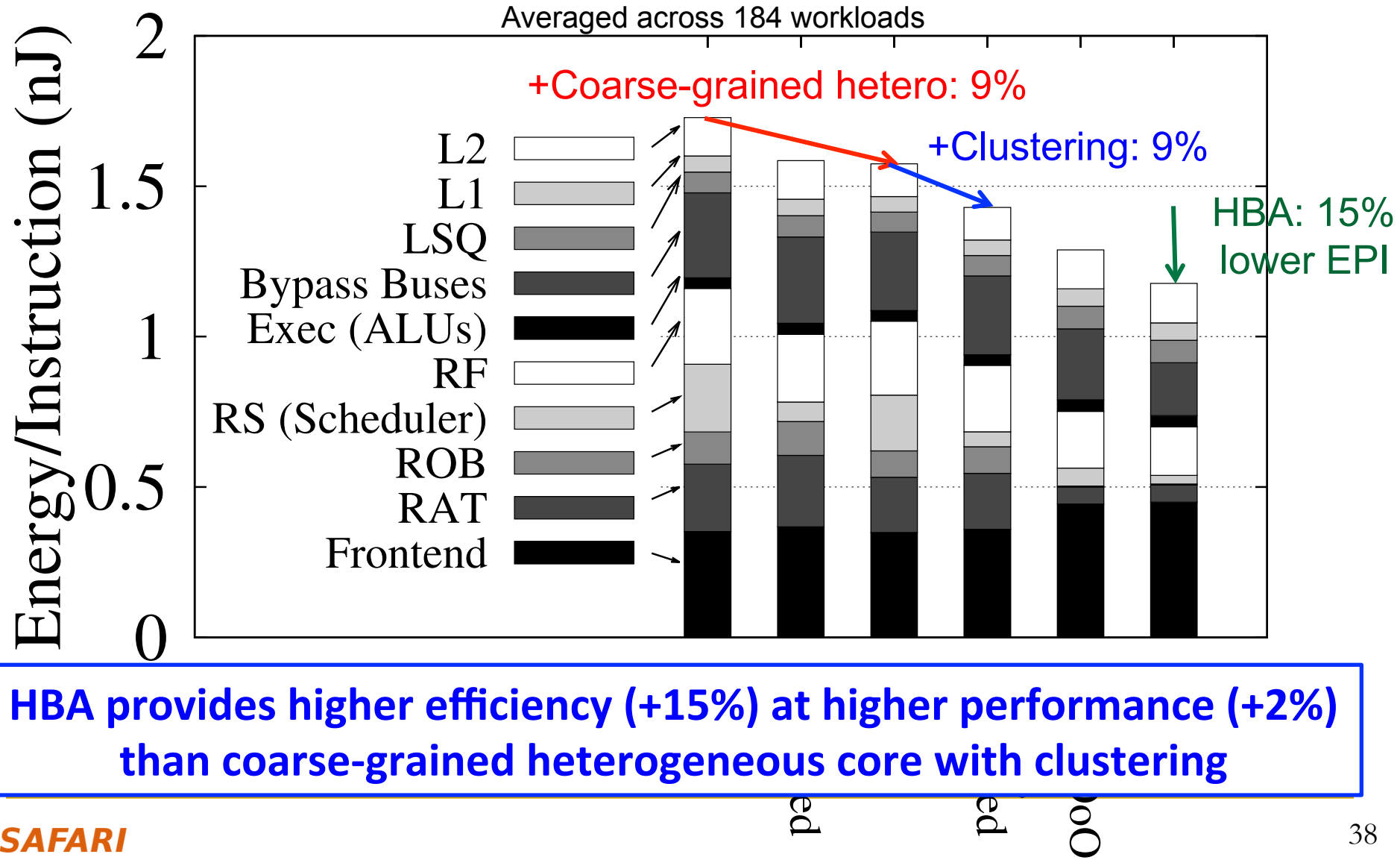
Energy Analysis (II)



HBA effectively takes advantage of clustering, atomic blocks, and heterogeneous backends to reduce energy

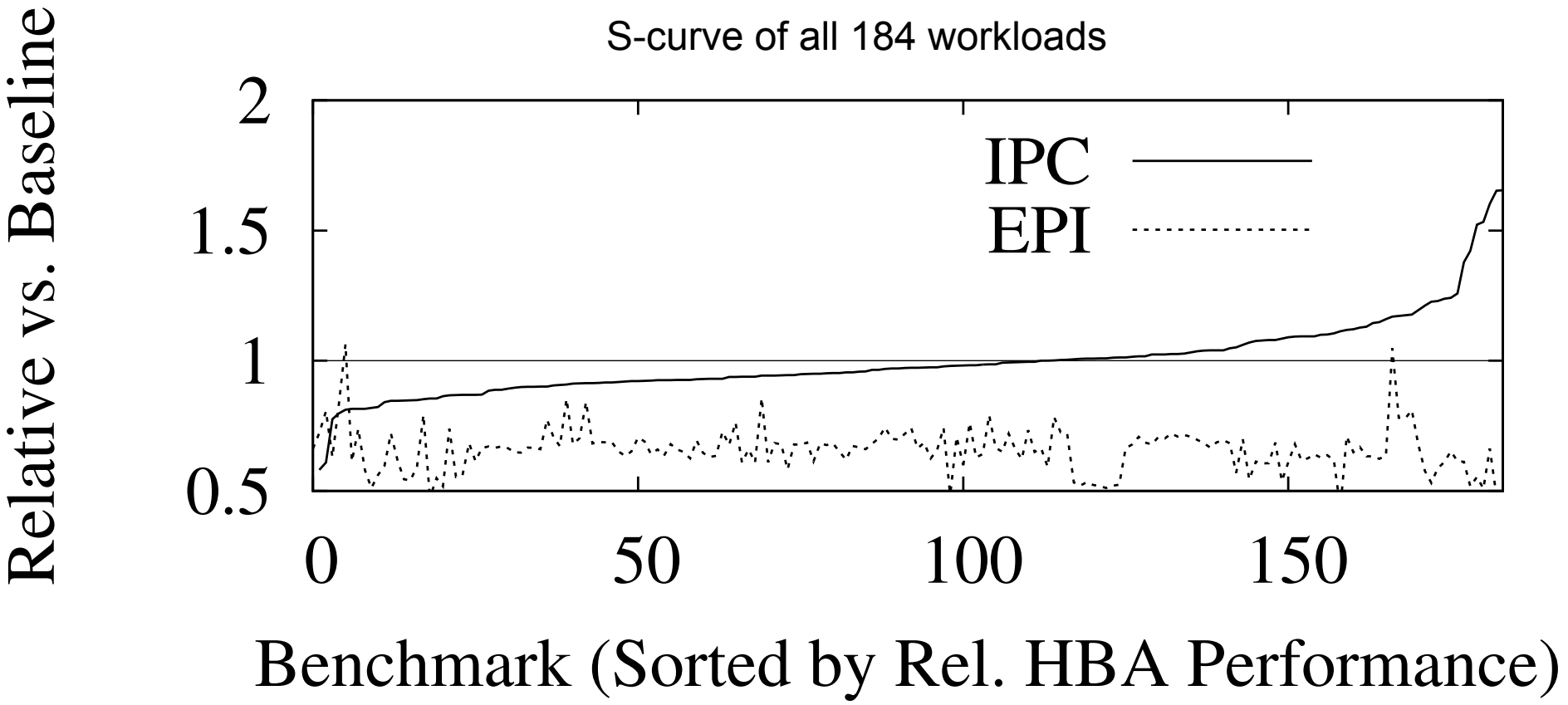
Comparison to Best Previous Design

- Coarse-grained heterogeneity + clustering the OoO core



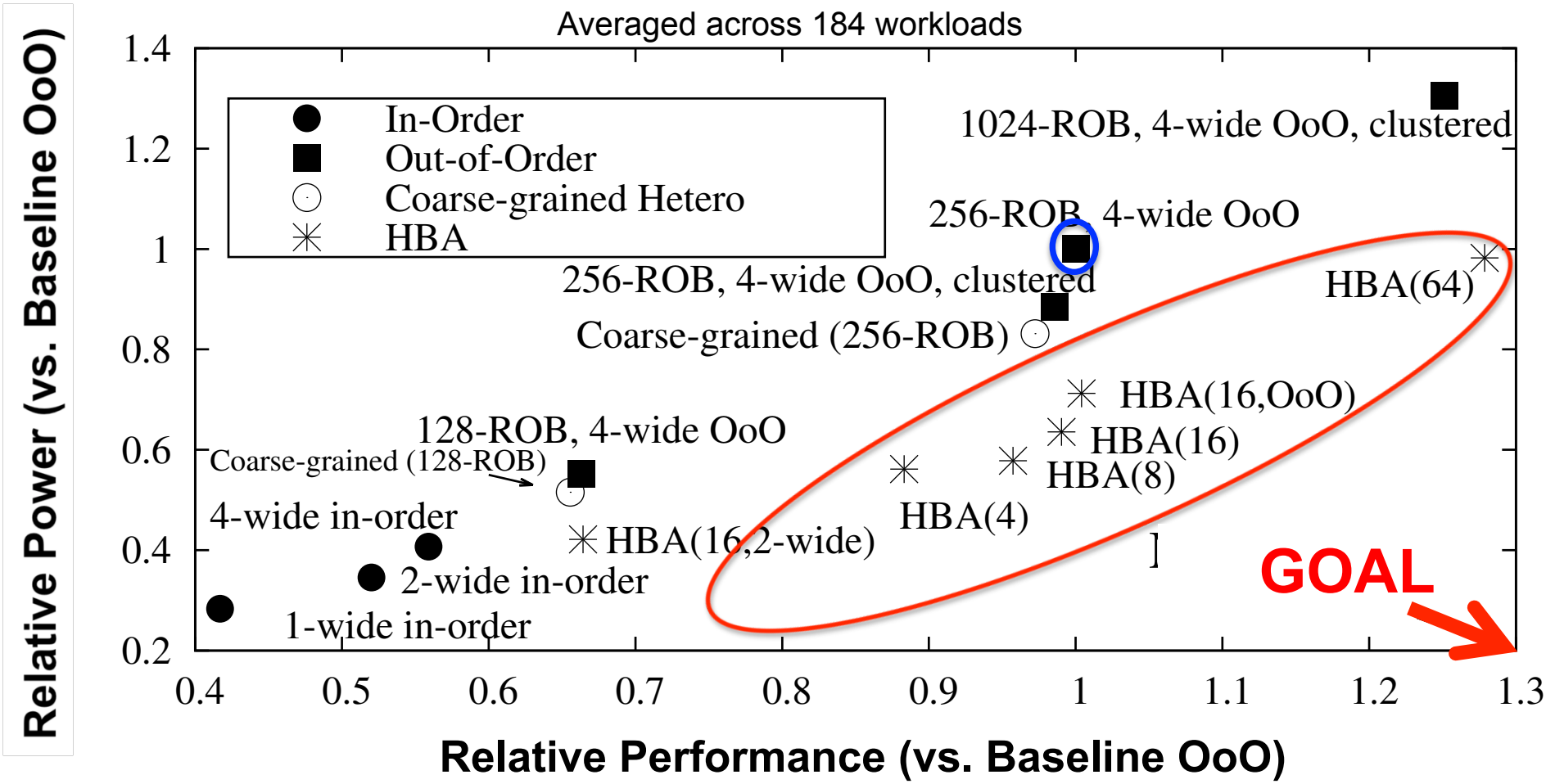
HBA provides higher efficiency (+15%) at higher performance (+2%) than coarse-grained heterogeneous core with clustering

Per-Workload Results



- HBA reduces energy on almost all workloads
- HBA can reduce or improve performance (analysis in paper)

Power-Performance Tradeoff Space



- HBA enables new design points in the tradeoff space

Cost of Heterogeneity

- Higher core area
  - Due to more backends
  - Can be optimized with good design (unified backends)
  - Core area cost increasingly small in an SoC
    - only 17% in Apple's A7
  - Analyzed in the paper but our design is not optimized for area

- Increased leakage power
  - HBA benefits reduce with higher transistor leakage
  - Analyzed in the paper

More Results and Analyses in the Paper

- Performance bottlenecks
- Fraction of OoO vs. VLIW blocks
- Energy optimizations in VLIW mode
- Energy optimizations in general
- Area and leakage

- Other and detailed results available in our technical report

Chris Fallin, Chris Wilkerson, and Onur Mutlu,
"The Heterogeneous Block Architecture"
SAFARI Technical Report, TR-SAFARI-2014-001,
Carnegie Mellon University, March 2014.

Talk Agenda

- Background and Motivation
- Two New Observations
- The Heterogeneous Block Architecture (HBA)
- An Initial HBA Design
- Experimental Evaluation
- Conclusions

Conclusions

- HBA is the first heterogeneous core substrate that enables
  - concurrent execution of fine-grained code blocks
  - on the most-efficient backend for each block

- Three characteristics of HBA
  - Atomic block execution to exploit fine-grained heterogeneity
  - Exploits stable instruction schedules to adapt code to backends
  - Uses clustering, atomicity and heterogeneity to reduce energy

- HBA greatly improves energy efficiency over four core designs

- A flexible execution substrate for exploiting fine-grained heterogeneity in core design

Building on This Work …

- HBA is a substrate for efficient core design

- One can design different cores using this substrate:
  - More backends of different types (CPU, SIMD, reconfigurable logic, …)
  - Better switching heuristics between backends
  - More optimization of each backend
  - Dynamic code optimizations specific to each backend
  - …

The Heterogeneous Block Architecture

A Flexible Substrate for Building Energy-Efficient High-Performance Cores

Chris Fallin[1] Chris Wilkerson[2] Onur Mutlu[1]

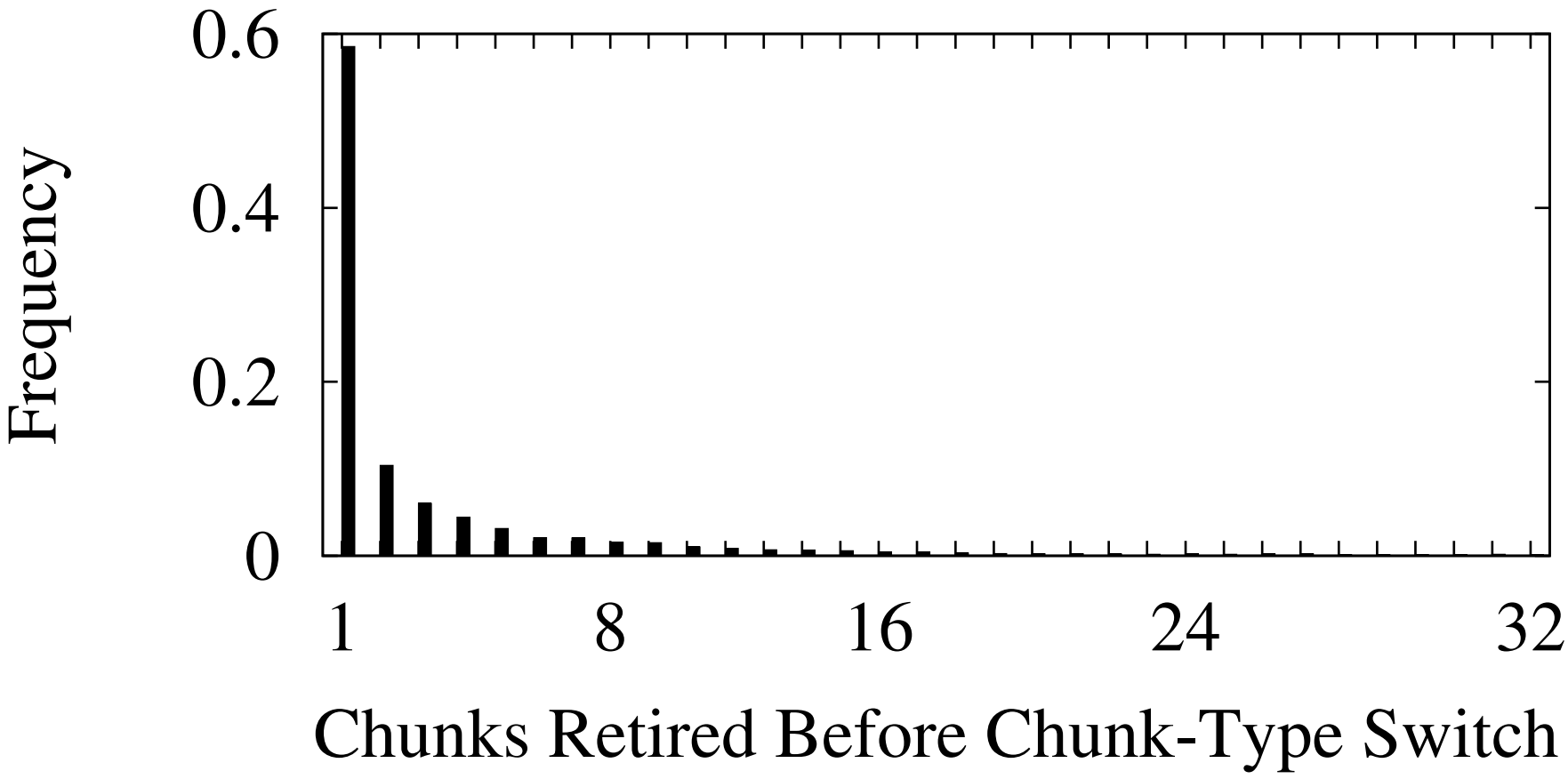
[1] Carnegie Mellon University
[2] Intel Corporation

SAFARI Carnegie Mellon intel

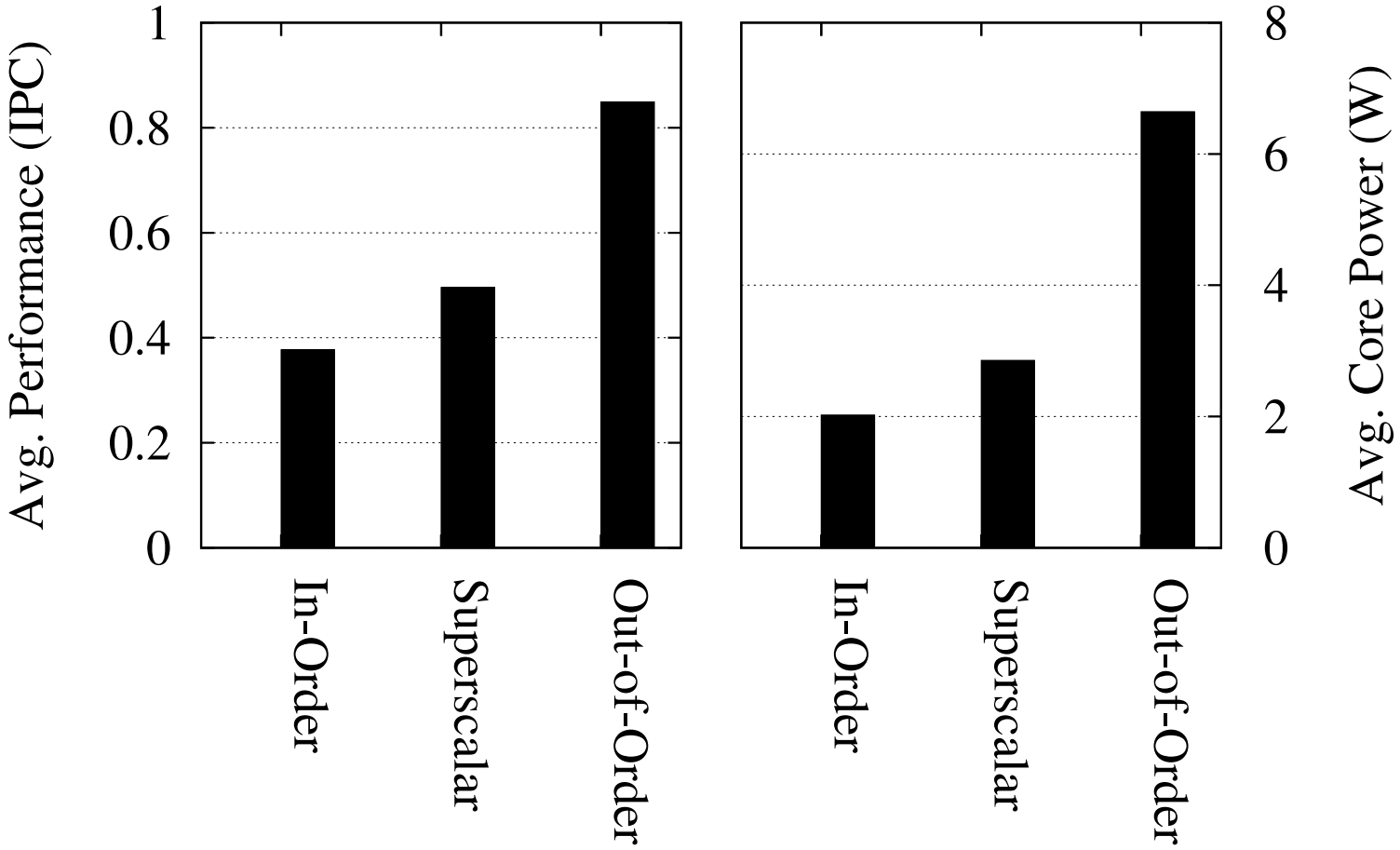
Why Atomic Blocks?

- Each block can be pre-specialized to its backend (pre-rename for OoO, pre-schedule VLIW in hardware, reorder instructions, eliminate instructions)
- Each backend can operate independently
- User-visible ISA can remain unchanged despite multiple execution backends

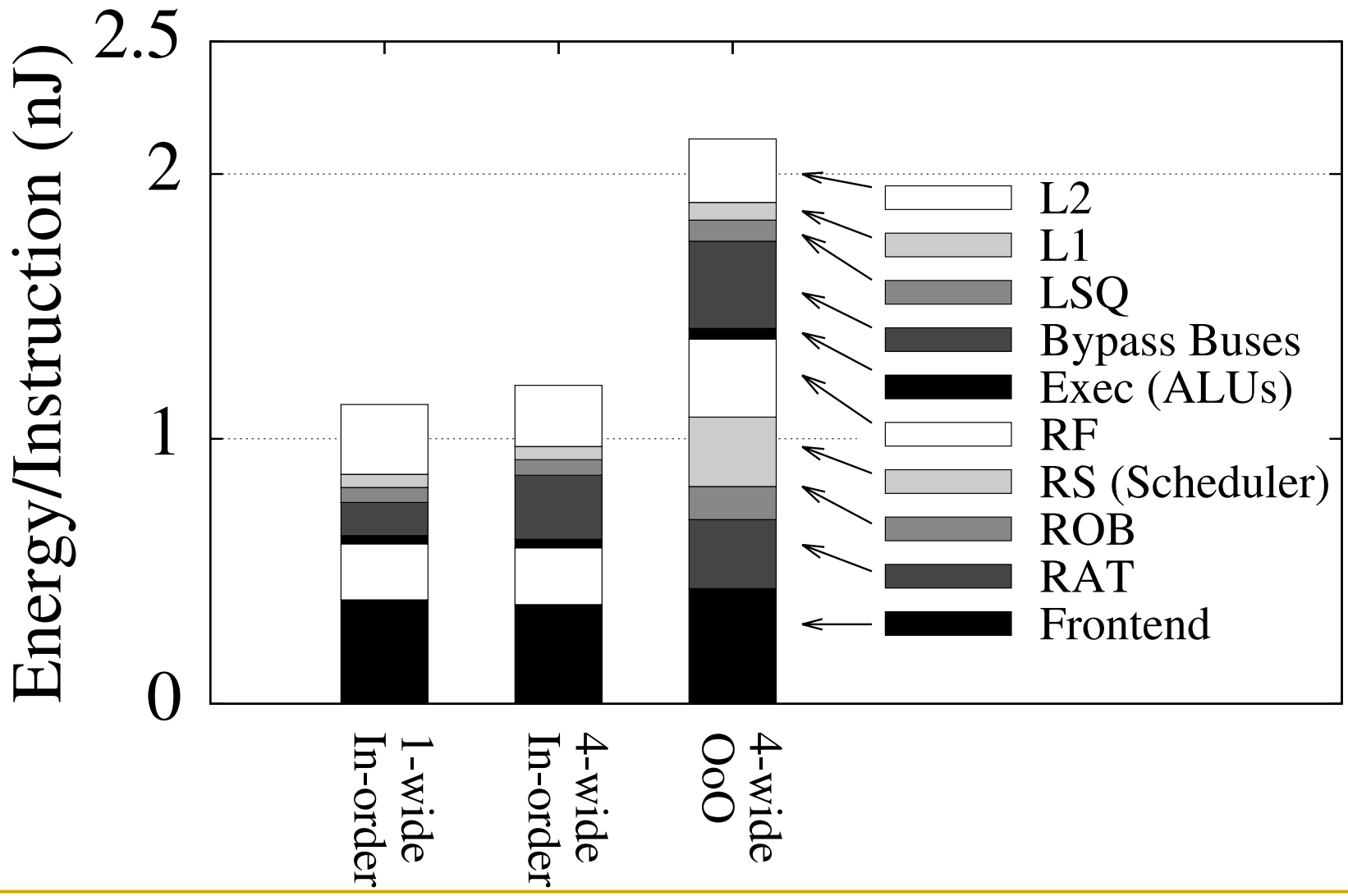
Nearby Chunks Have Different Behavior



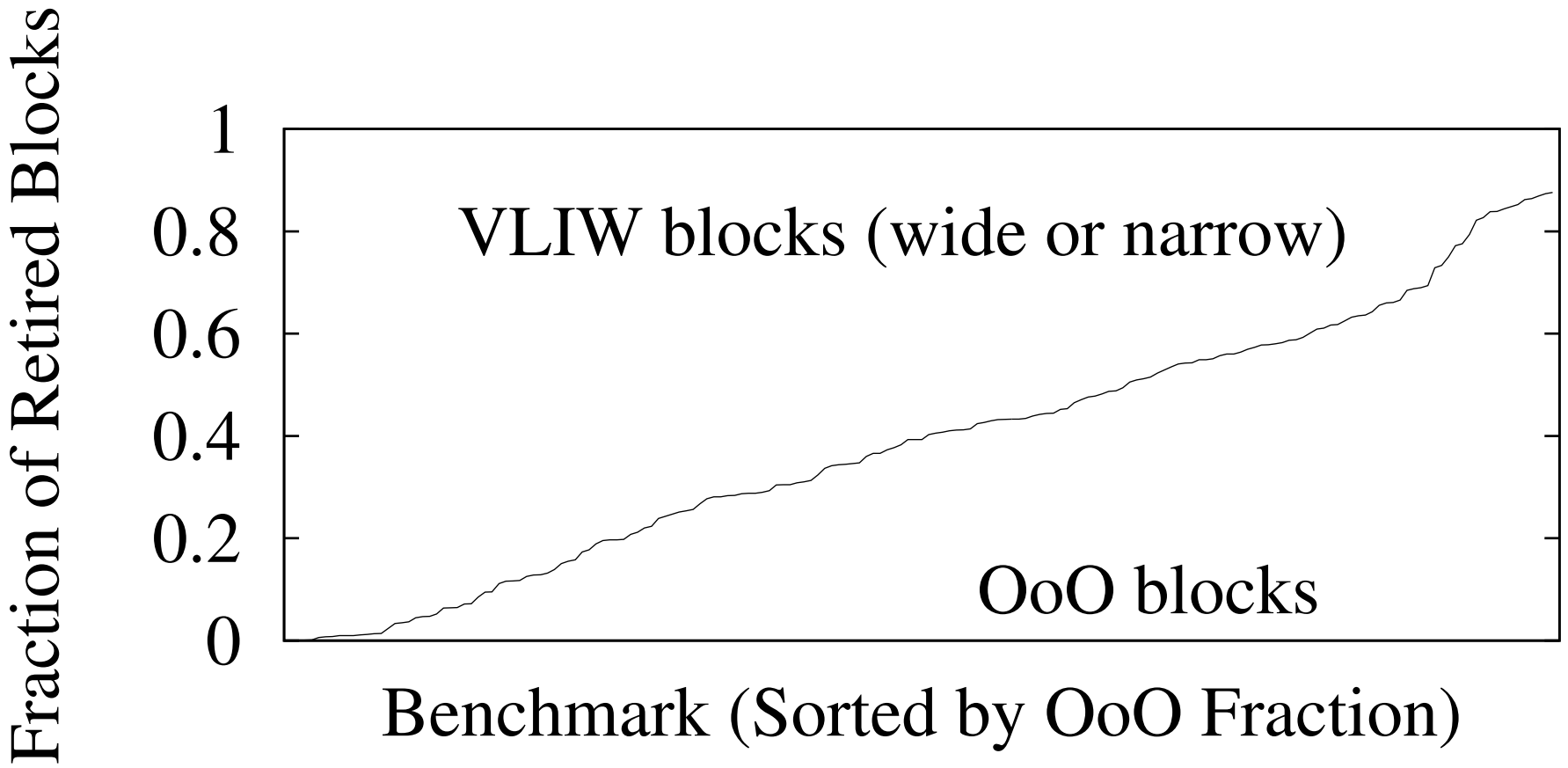
In-Order vs. Superscalar vs. OoO



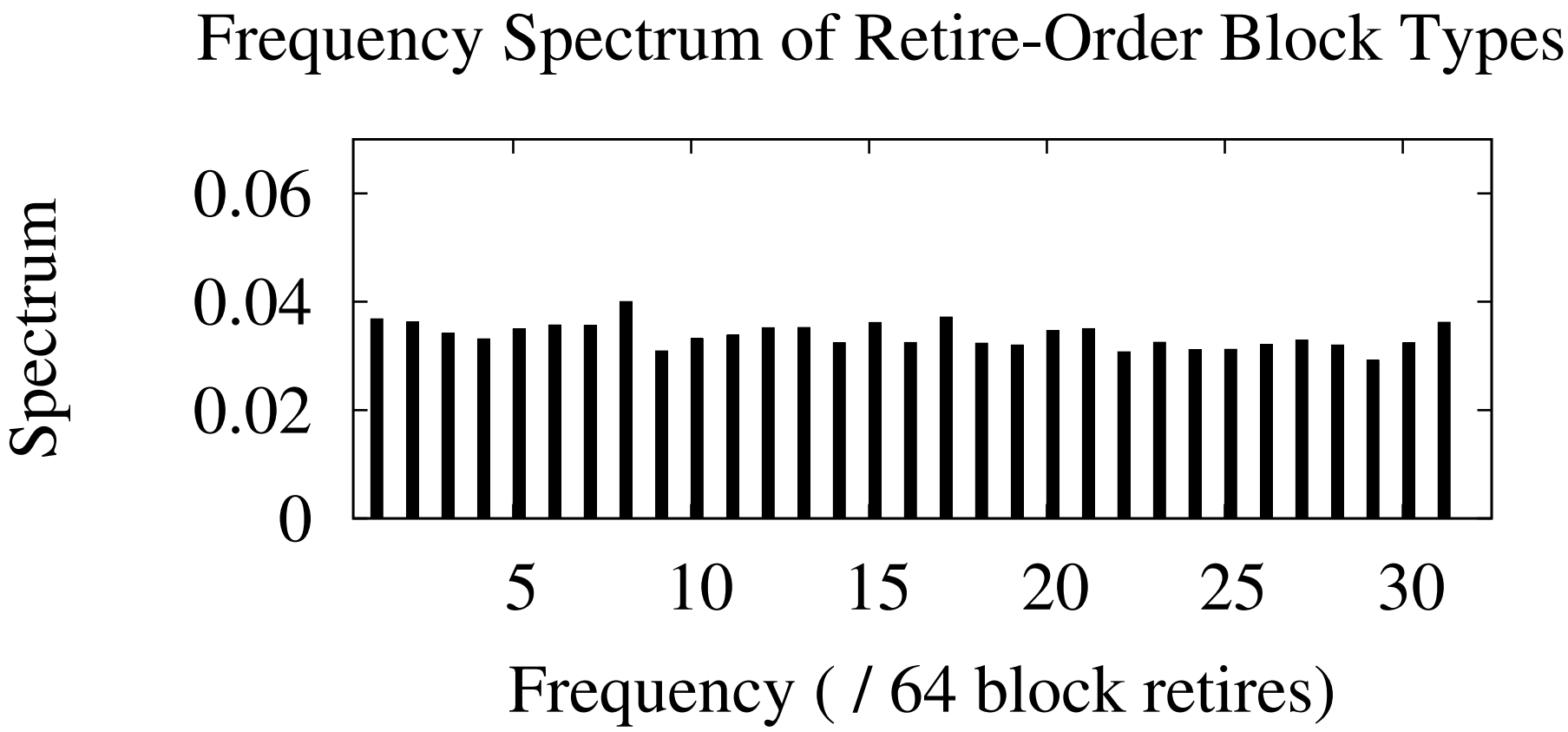
Baseline EPI Breakdown



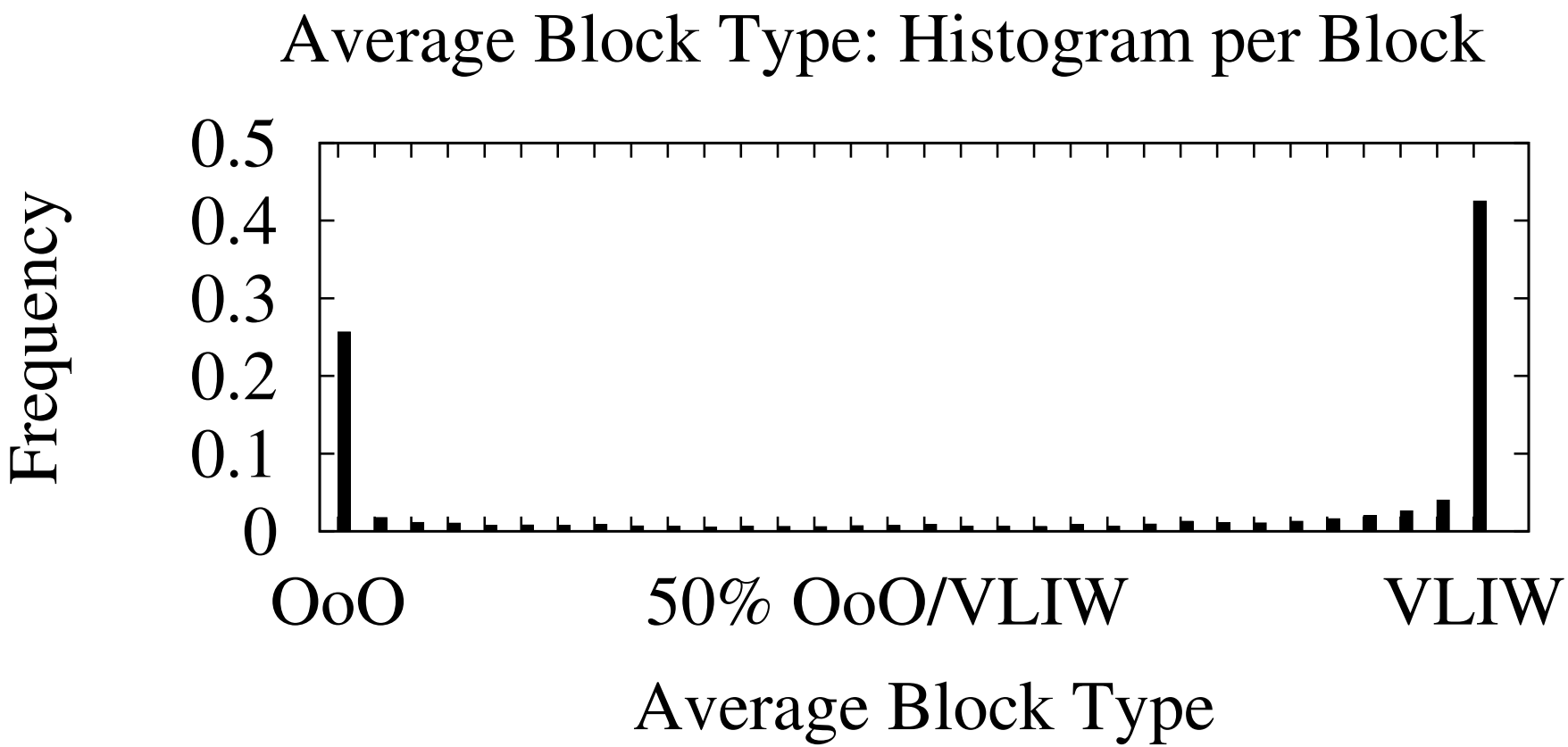
Retired Block Types (I)



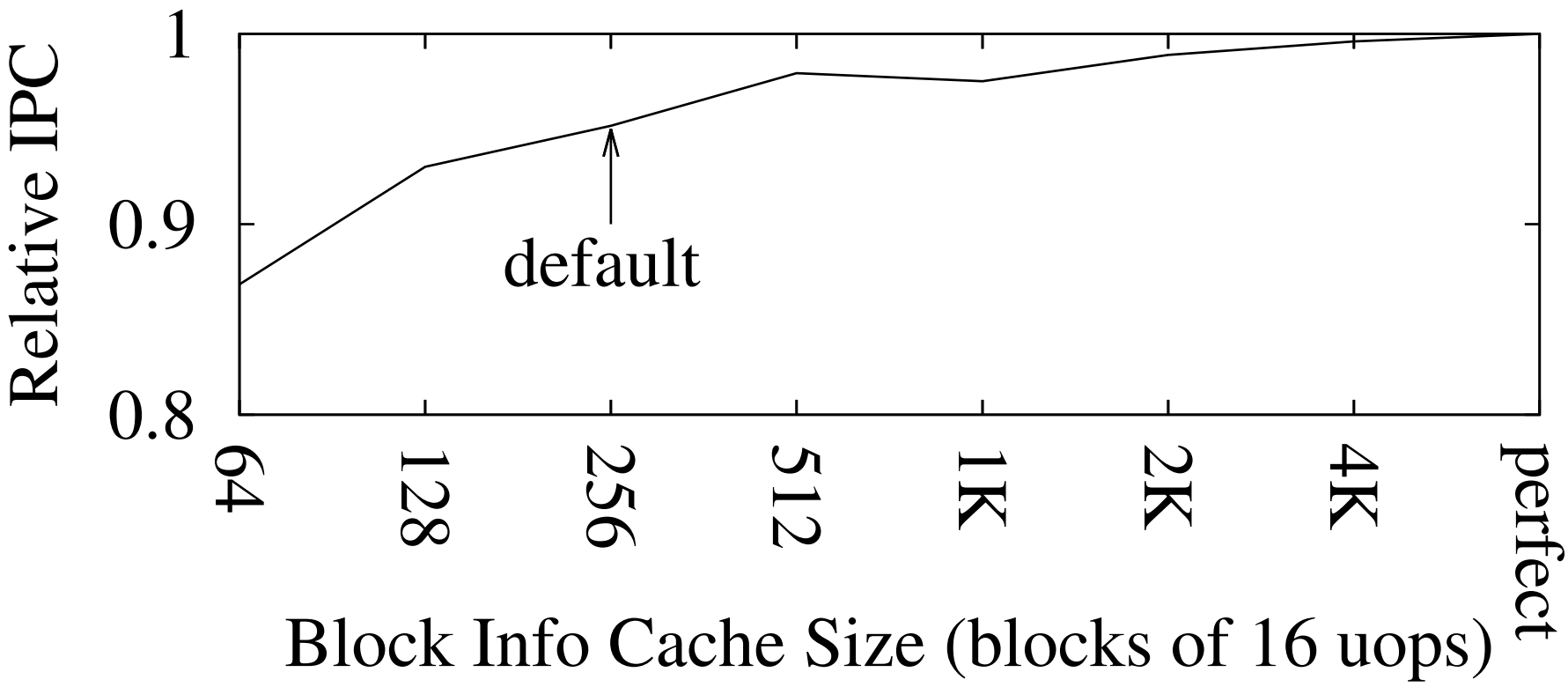
Retired Block Types (II)



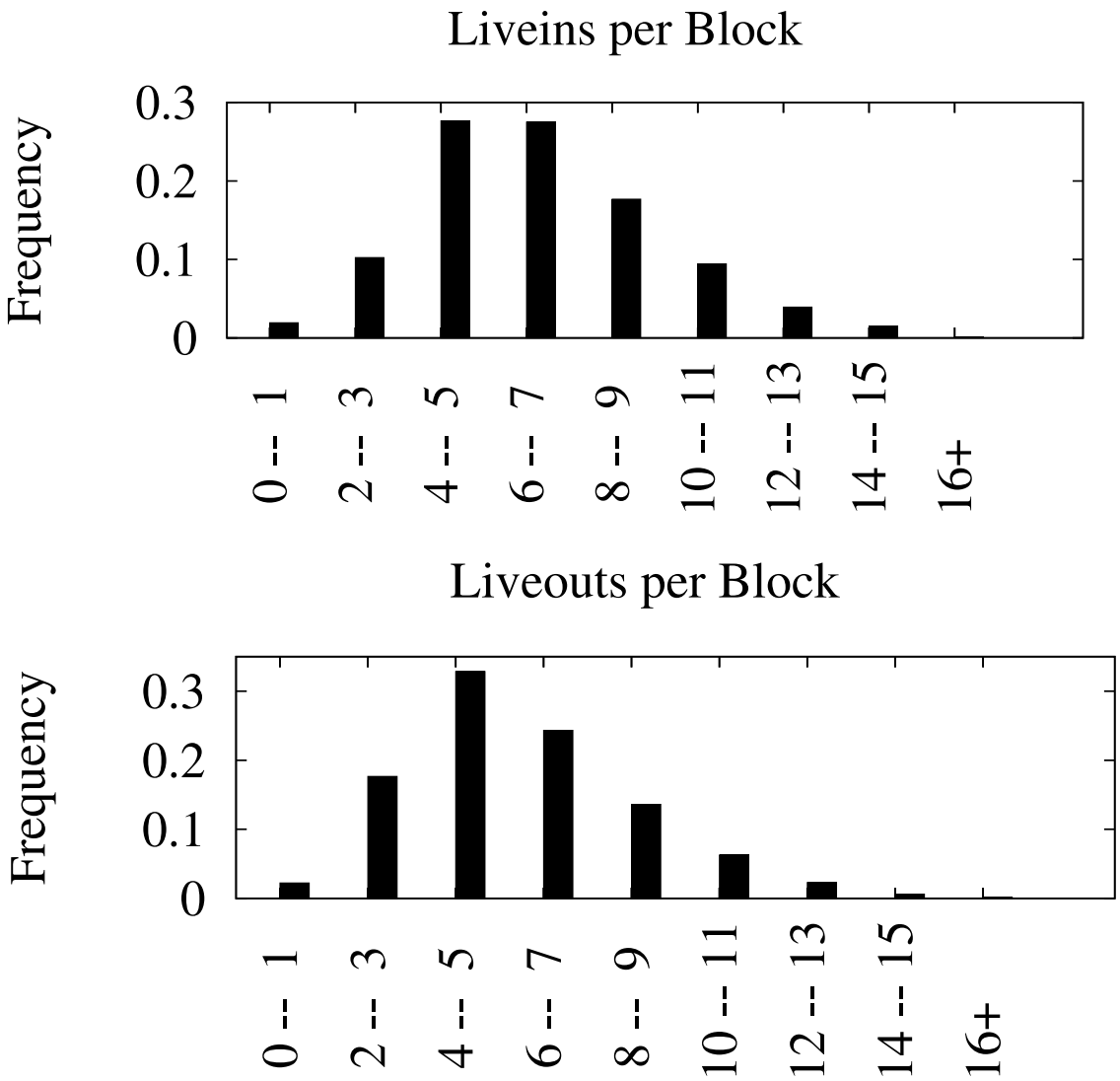
Retired Block Types (III)



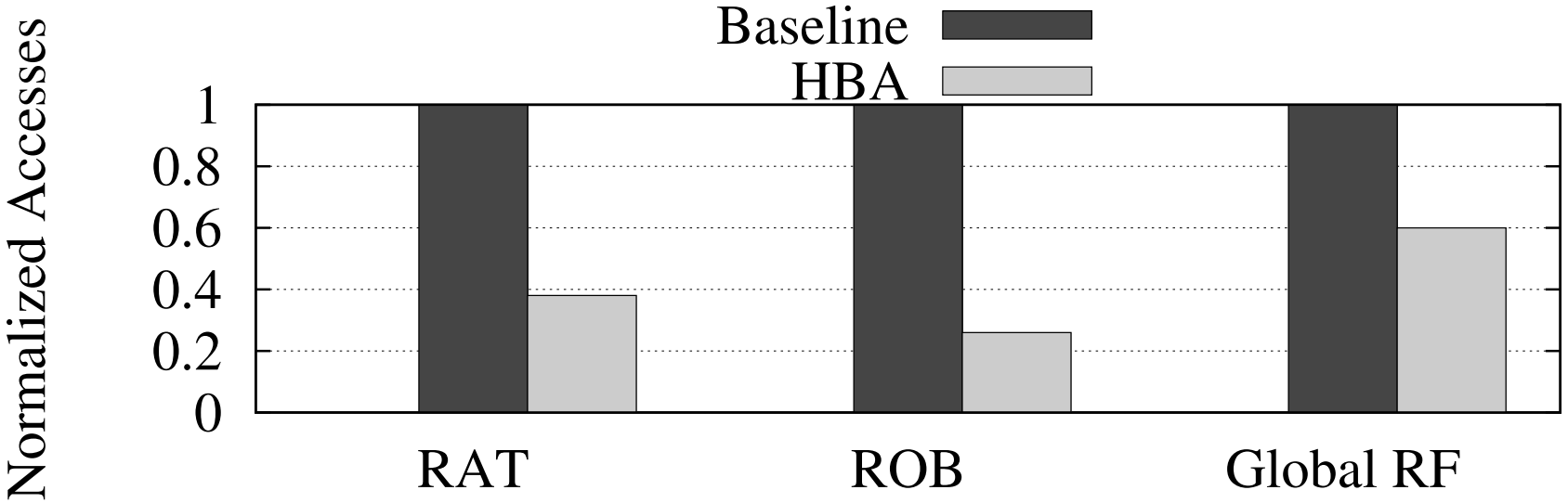
BIC Size Sensitivity



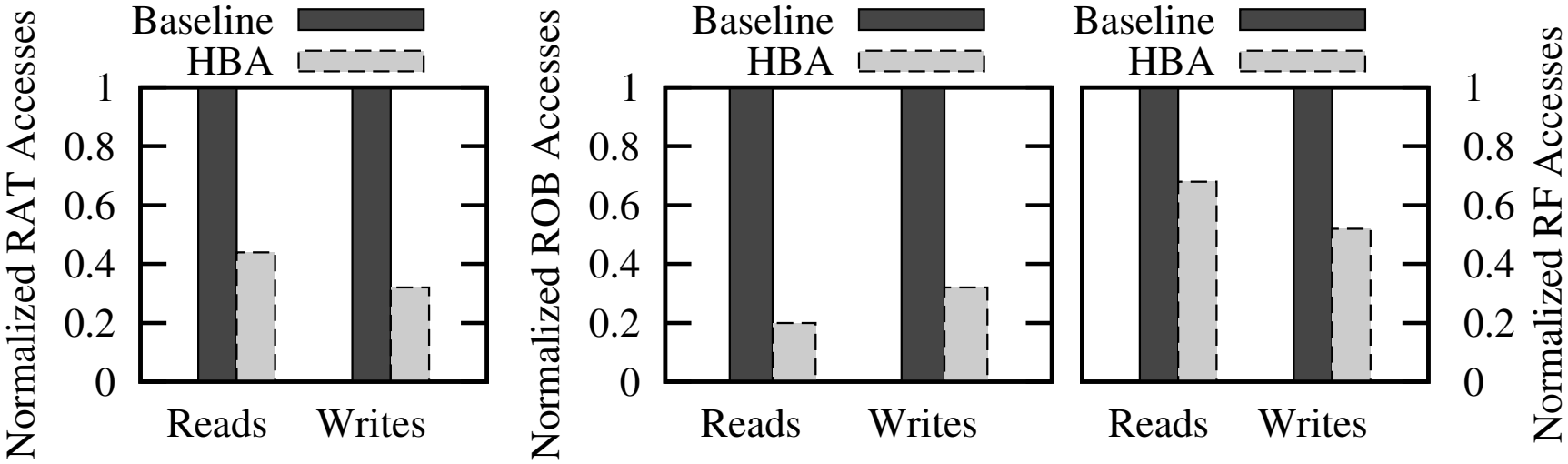
Livein-Liveout Statistics



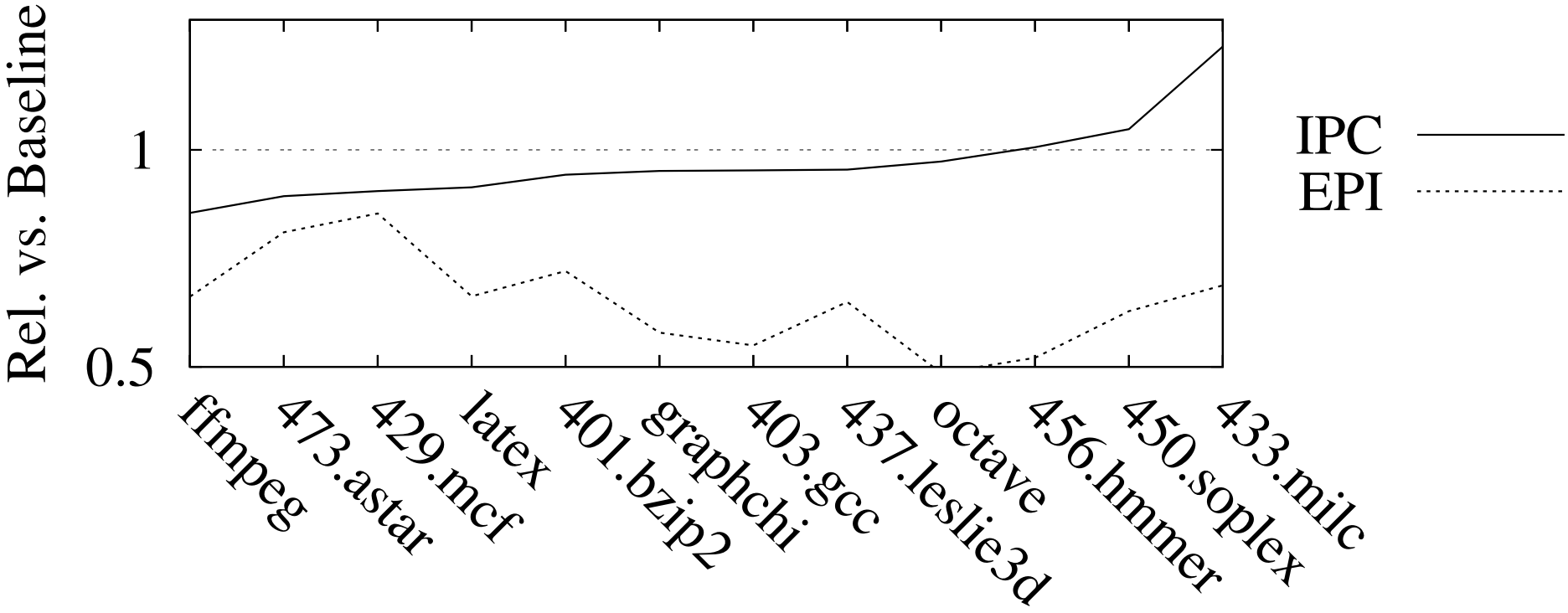
Auxiliary Data: Benefits of HBA



Auxiliary Data: Benefits of HBA



Some Specific Workloads



Related Works

- Forming and reusing instruction schedules [DIF, ISCA'97] [Transmeta'00][Banerjia, IEEE TOC'98][Palomar, ICCD'09]
- Coarse-grained heterogeneous cores [Lukefahr, MICRO'12]
- Atomic blocks and block-structured ISA [Melvin&Patt, MICRO'88, IJPP'95]
- Instruction prescheduling [Michaud&Seznec, HPCA'01]
- Yoga [Villavieja+, HPS Tech Report'14]