

Multi-Core Architectures and Shared Resource Management

Lecture 1.2: Cache Management

Prof. Onur Mutlu

<http://www.ece.cmu.edu/~omutlu>

onur@cmu.edu

Bogazici University

June 7, 2013

SAFARI Carnegie Mellon

Last Lecture

- Parallel Computer Architecture Basics
 - Amdahl's Law
 - Bottlenecks in the Parallel Portion (3 fundamental ones)
- Why Multi-Core? Alternatives, shortcomings, tradeoffs.
- Evolution of Symmetric Multi-Core Systems
 - Sun NiagaraX, IBM PowerX, Runahead Execution
- Asymmetric Multi-Core Systems
 - Accelerated Critical Sections
 - Bottleneck Identification and Scheduling

What Will You Learn?

- Tentative, Aggressive Schedule
 - Lecture 1: Why multi-core? Basics, alternatives, tradeoffs
Symmetric versus asymmetric multi-core systems
 - Lecture 2: Shared cache design for multi-cores
(if time permits) Interconnect design for multi-cores
 - Lecture 3: Data parallelism and GPUs (if time permits)
(if time permits) Prefetcher design and management

Agenda for Today and Beyond

- Wrap up Asymmetric Multi-Core Systems
 - Handling Private Data Locality
 - Asymmetry Everywhere
- Cache design for multi-core systems
- Interconnect design for multi-core systems
- Prefetcher design for multi-core systems
- Data Parallelism and GPUs

Readings for Lecture June 6 (Lecture 1.1)

- Required – Symmetric and Asymmetric Multi-Core Systems
 - Mutlu et al., “Runahead Execution: An Alternative to Very Large Instruction Windows for Out-of-order Processors,” HPCA 2003, IEEE Micro 2003.
 - Suleman et al., “Accelerating Critical Section Execution with Asymmetric Multi-Core Architectures,” ASPLOS 2009, IEEE Micro 2010.
 - Suleman et al., “Data Marshaling for Multi-Core Architectures,” ISCA 2010, IEEE Micro 2011.
 - Joao et al., “Bottleneck Identification and Scheduling for Multithreaded Applications,” ASPLOS 2012.
 - Joao et al., “Utility-Based Acceleration of Multithreaded Applications on Asymmetric CMPs,” ISCA 2013.
- Recommended
 - Amdahl, “Validity of the single processor approach to achieving large scale computing capabilities,” AFIPS 1967.
 - Olukotun et al., “The Case for a Single-Chip Multiprocessor,” ASPLOS 1996.
 - Mutlu et al., “Techniques for Efficient Processing in Runahead Execution Engines,” ISCA 2005, IEEE Micro 2006.

Videos for Lecture June 6 (Lecture 1.1)

■ Runahead Execution

- <http://www.youtube.com/watch?v=z8YpjqXQJIA&list=PL5PHm2jkkXmidJOd59REog9jDnPDTG6IJ&index=28>

■ Multiprocessors

- Basics:
http://www.youtube.com/watch?v=7ozCK_Mgxfk&list=PL5PHm2jkkXmidJOd59REog9jDnPDTG6IJ&index=31
- Correctness and Coherence:
<http://www.youtube.com/watch?v=U-VZKMgItDM&list=PL5PHm2jkkXmidJOd59REog9jDnPDTG6IJ&index=32>
- Heterogeneous Multi-Core:
<http://www.youtube.com/watch?v=r6r2NJxj3kI&list=PL5PHm2jkkXmidJOd59REog9jDnPDTG6IJ&index=34>

Readings for Lecture June 7 (Lecture 1.2)

■ Required – Caches in Multi-Core

- Qureshi et al., “A Case for MLP-Aware Cache Replacement,” ISCA 2005.
- Seshadri et al., “The Evicted-Address Filter: A Unified Mechanism to Address both Cache Pollution and Thrashing,” PACT 2012.
- Pekhimenko et al., “Base-Delta-Immediate Compression: Practical Data Compression for On-Chip Caches,” PACT 2012.
- Pekhimenko et al., “Linearly Compressed Pages: A Main Memory Compression Framework with Low Complexity and Low Latency,” SAFARI Technical Report 2013.

■ Recommended

- Qureshi et al., “Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches,” MICRO 2006.

Videos for Lecture June7 (Lecture 1.2)

- Cache basics:

- <http://www.youtube.com/watch?v=TpMdBrM1hVc&list=PL5PHm2jkkXmidJOd59REog9jDnPDTG6IJ&index=23>

- Advanced caches:

- <http://www.youtube.com/watch?v=TboaFbjTd-E&list=PL5PHm2jkkXmidJOd59REog9jDnPDTG6IJ&index=24>

Readings for Lecture June 10 (Lecture 1.3)

■ Required – Interconnects in Multi-Core

- ❑ Moscibroda and Mutlu, “A Case for Bufferless Routing in On-Chip Networks,” ISCA 2009.
- ❑ Fallin et al., “CHIPPER: A Low-Complexity Bufferless Deflection Router,” HPCA 2011.
- ❑ Fallin et al., “MinBD: Minimally-Buffered Deflection Routing for Energy-Efficient Interconnect,” NOCS 2012.
- ❑ Das et al., “Application-Aware Prioritization Mechanisms for On-Chip Networks,” MICRO 2009.
- ❑ Das et al., “Aergia: Exploiting Packet Latency Slack in On-Chip Networks,” ISCA 2010, IEEE Micro 2011.

■ Recommended

- ❑ Grot et al. “Preemptive Virtual Clock: A Flexible, Efficient, and Cost-effective QOS Scheme for Networks-on-Chip,” MICRO 2009.
- ❑ Grot et al., “Kilo-NOC: A Heterogeneous Network-on-Chip Architecture for Scalability and Service Guarantees,” ISCA 2011, IEEE Micro 2012.

Videos for Lecture June 10 (Lecture 1.3)

■ Interconnects

- <http://www.youtube.com/watch?v=6xEpbFVgnf8&list=PL5PHm2jkkXmidJOd59REog9jDnPDTG6IJ&index=33>

■ GPUs and SIMD processing

- Vector/array processing basics:
<http://www.youtube.com/watch?v=f-XL4BNRoBA&list=PL5PHm2jkkXmidJOd59REog9jDnPDTG6IJ&index=15>
- GPUs versus other execution models:
<http://www.youtube.com/watch?v=dl5TZ4-oao0&list=PL5PHm2jkkXmidJOd59REog9jDnPDTG6IJ&index=19>
- GPUs in more detail:
<http://www.youtube.com/watch?v=vr5hbSkb1Eg&list=PL5PHm2jkkXmidJOd59REog9jDnPDTG6IJ&index=20>

Readings for Prefetching

■ Prefetching

- ❑ Srinath et al., "Feedback Directed Prefetching: Improving the Performance and Bandwidth-Efficiency of Hardware Prefetchers," HPCA 2007.
- ❑ Ebrahimi et al., "Coordinated Control of Multiple Prefetchers in Multi-Core Systems," MICRO 2009.
- ❑ Ebrahimi et al., "Techniques for Bandwidth-Efficient Prefetching of Linked Data Structures in Hybrid Prefetching Systems," HPCA 2009.
- ❑ Ebrahimi et al., "Prefetch-Aware Shared Resource Management for Multi-Core Systems," ISCA 2011.
- ❑ Lee et al., "Prefetch-Aware DRAM Controllers," MICRO 2008.

■ Recommended

- ❑ Lee et al., "Improving Memory Bank-Level Parallelism in the Presence of Prefetching," MICRO 2009.

Videos for Prefetching

■ Prefetching

- ❑ <http://www.youtube.com/watch?v=IIkIwiNNl0c&list=PL5PHm2jkkXmidJOd59REog9jDnPDTG6IJ&index=29>
- ❑ <http://www.youtube.com/watch?v=yapQavK6LUk&list=PL5PHm2jkkXmidJOd59REog9jDnPDTG6IJ&index=30>

Readings for GPUs and SIMD

- GPUs and SIMD processing
 - Narasiman et al., “Improving GPU Performance via Large Warps and Two-Level Warp Scheduling,” MICRO 2011.
 - Jog et al., “OWL: Cooperative Thread Array Aware Scheduling Techniques for Improving GPGPU Performance,” ASPLOS 2013.
 - Jog et al., “Orchestrated Scheduling and Prefetching for GPGPUs,” ISCA 2013.
 - Lindholm et al., “NVIDIA Tesla: A Unified Graphics and Computing Architecture,” IEEE Micro 2008.
 - Fung et al., “Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow,” MICRO 2007.

Videos for GPUs and SIMD

■ GPUs and SIMD processing

- Vector/array processing basics:

<http://www.youtube.com/watch?v=f-XL4BNRoBA&list=PL5PHm2jkkXmidJOd59REog9jDnPDTG6IJ&index=15>

- GPUs versus other execution models:

<http://www.youtube.com/watch?v=dl5TZ4-oao0&list=PL5PHm2jkkXmidJOd59REog9jDnPDTG6IJ&index=19>

- GPUs in more detail:

<http://www.youtube.com/watch?v=vr5hbSkb1Eg&list=PL5PHm2jkkXmidJOd59REog9jDnPDTG6IJ&index=20>

Online Lectures and More Information

■ Online Computer Architecture Lectures

- <http://www.youtube.com/playlist?list=PL5PHm2jkkXmidJOd59REog9jDnPDTG6IJ>

■ Online Computer Architecture Courses

- Intro: <http://www.ece.cmu.edu/~ece447/s13/doku.php>
- Advanced: <http://www.ece.cmu.edu/~ece740/f11/doku.php>
- Advanced: <http://www.ece.cmu.edu/~ece742/doku.php>

■ Recent Research Papers

- <http://users.ece.cmu.edu/~omutlu/projects.htm>
- <http://scholar.google.com/citations?user=7XyGUGkAAAAJ&hl=en>

Asymmetric Multi-Core Systems Continued

Handling Private Data Locality: Data Marshaling

M. Aater Suleman, Onur Mutlu, Jose A. Joao, Khubaib, and Yale N. Patt,
"Data Marshaling for Multi-core Architectures"
*Proceedings of the 37th International Symposium on Computer Architecture (ISCA),
pages 441-450, Saint-Malo, France, June 2010.*

Staged Execution Model (I)

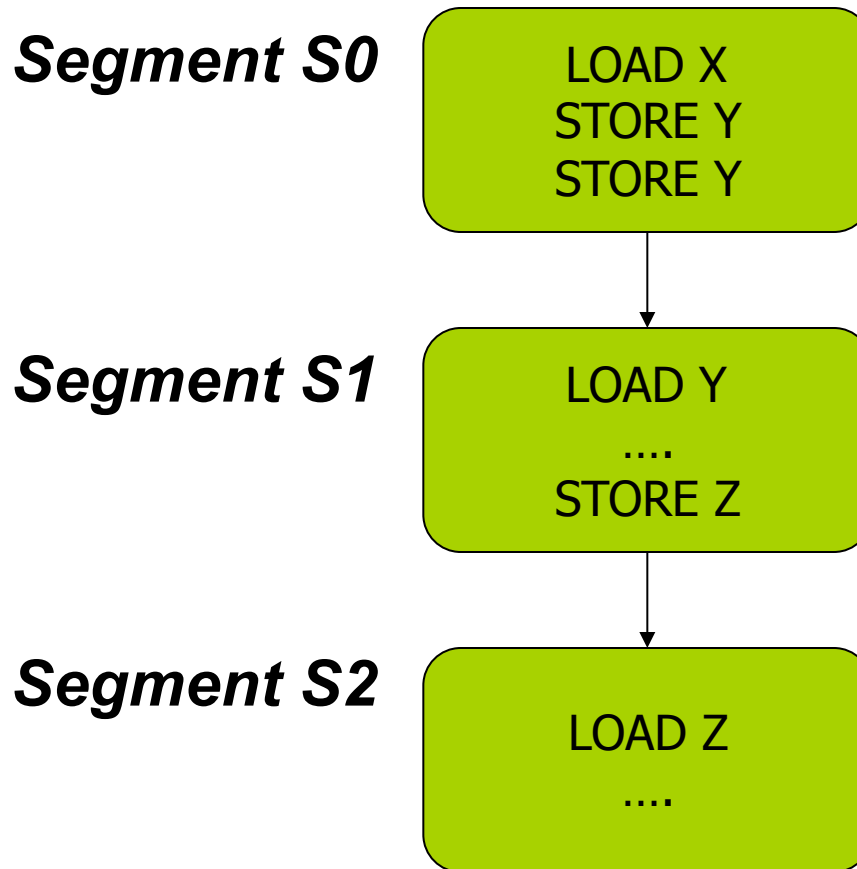
- Goal: speed up a program by dividing it up into pieces
- Idea
 - Split program code into *segments*
 - Run each segment on the core best-suited to run it
 - Each core assigned a work-queue, storing segments to be run
- Benefits
 - Accelerates segments/critical-paths using specialized/heterogeneous cores
 - Exploits inter-segment parallelism
 - Improves locality of within-segment data
- Examples
 - Accelerated critical sections, Bottleneck identification and scheduling
 - Producer-consumer pipeline parallelism
 - Task parallelism (Cilk, Intel TBB, Apple Grand Central Dispatch)
 - Special-purpose cores and functional units

Staged Execution Model (II)

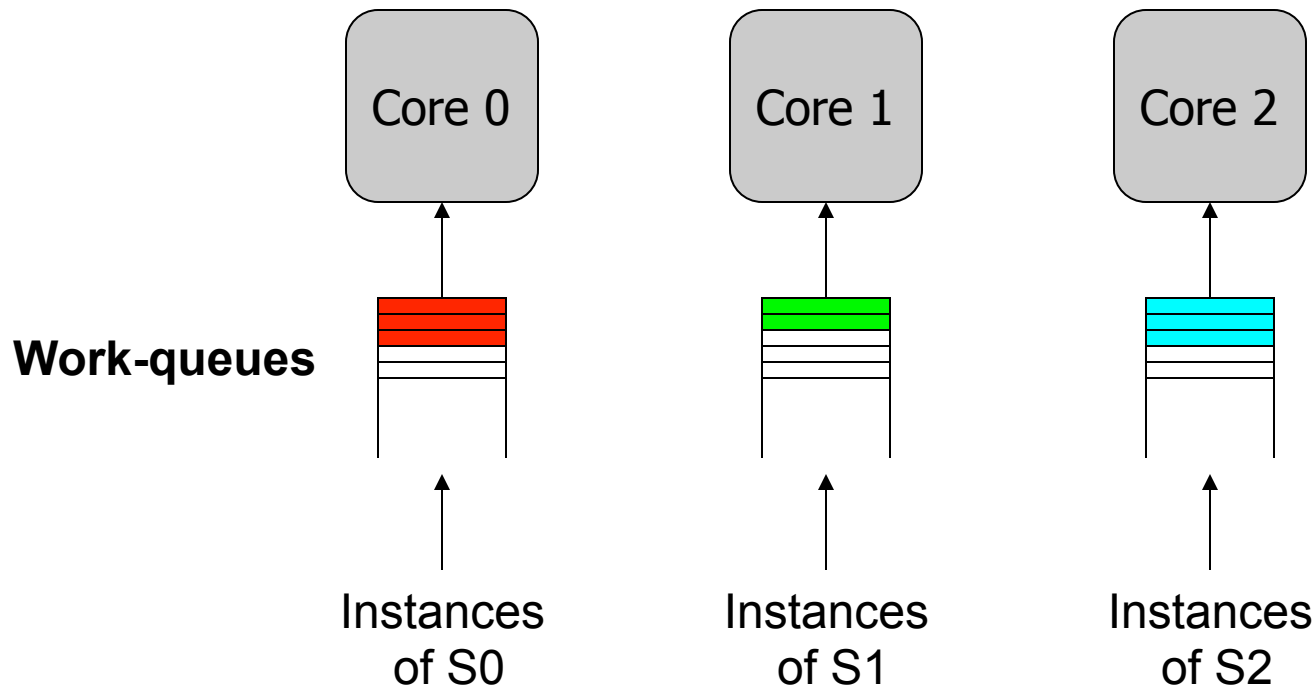


Staged Execution Model (III)

Split code into segments



Staged Execution Model (IV)

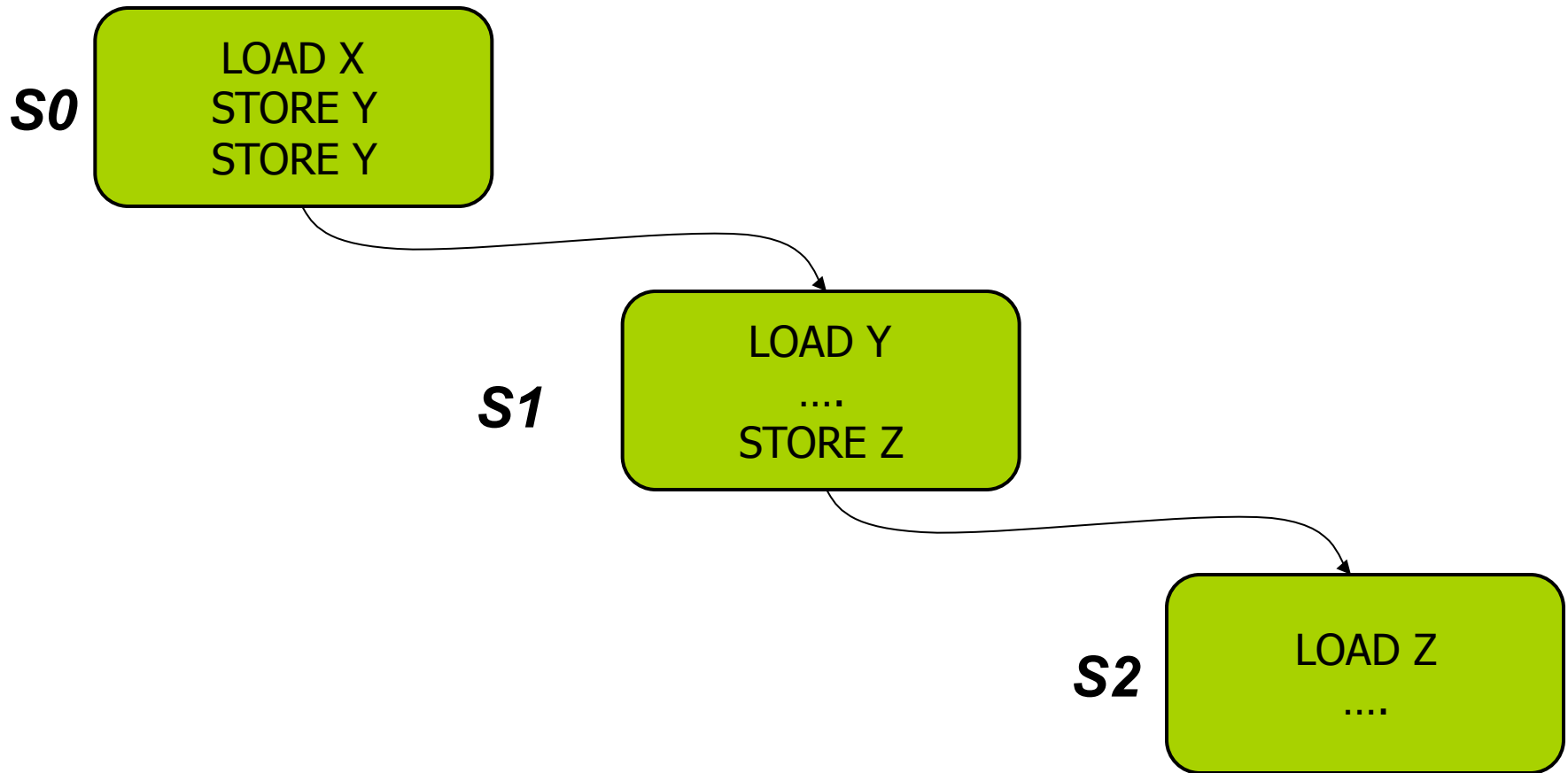


Staged Execution Model: Segment Spawning

Core 0

Core 1

Core 2



Staged Execution Model: Two Examples

- **Accelerated Critical Sections** [Suleman et al., ASPLOS 2009]
 - Idea: Ship critical sections to a large core in an asymmetric CMP
 - Segment 0: Non-critical section
 - Segment 1: Critical section
 - Benefit: Faster execution of critical section, reduced serialization, improved lock and shared data locality

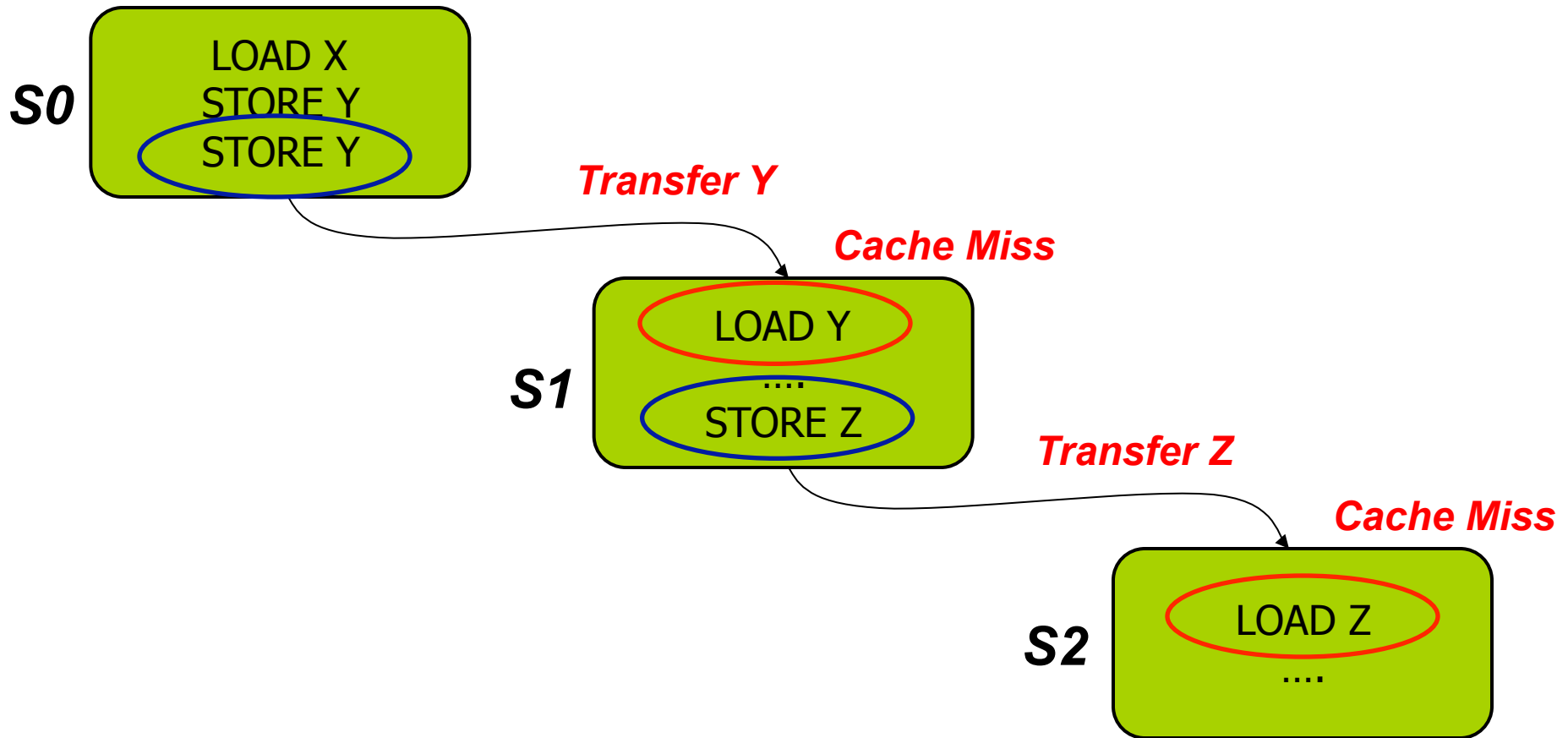
- **Producer-Consumer Pipeline Parallelism**
 - Idea: Split a loop iteration into multiple “pipeline stages” where one stage consumes data produced by the next stage → each stage runs on a different core
 - Segment N: Stage N
 - Benefit: Stage-level parallelism, better locality → faster execution

Problem: Locality of Inter-segment Data

Core 0

Core 1

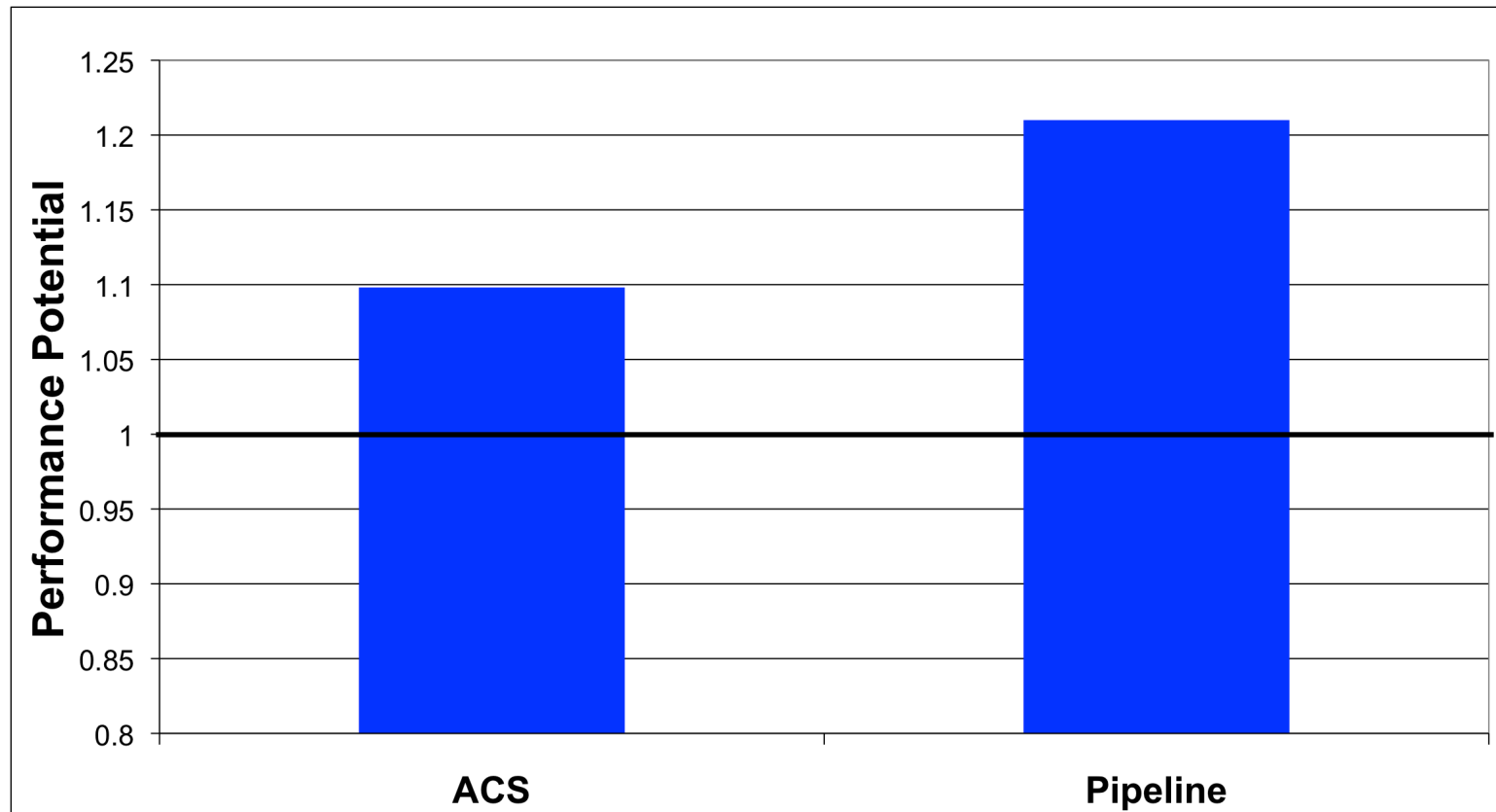
Core 2



Problem: Locality of Inter-segment Data

- Accelerated Critical Sections [Suleman et al., ASPLOS 2010]
 - Idea: Ship critical sections to a large core in an ACMP
 - Problem: Critical section incurs a cache miss when it touches data produced in the non-critical section (i.e., thread private data)
- Producer-Consumer Pipeline Parallelism
 - Idea: Split a loop iteration into multiple “pipeline stages” → each stage runs on a different core
 - Problem: A stage incurs a cache miss when it touches data produced by the previous stage
- Performance of Staged Execution limited by inter-segment cache misses

What if We Eliminated All Inter-segment Misses?

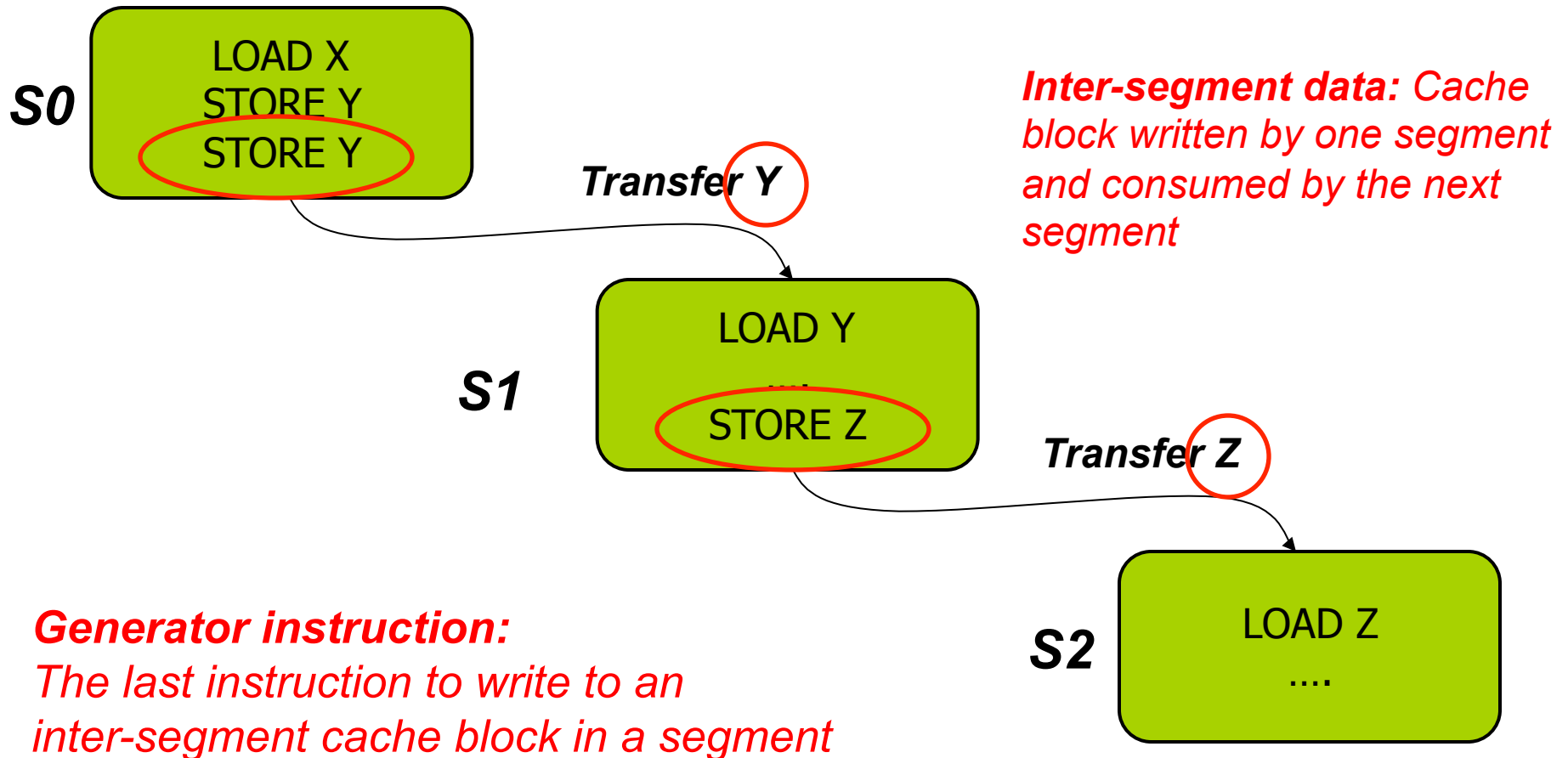


Terminology

Core 0

Core 1

Core 2



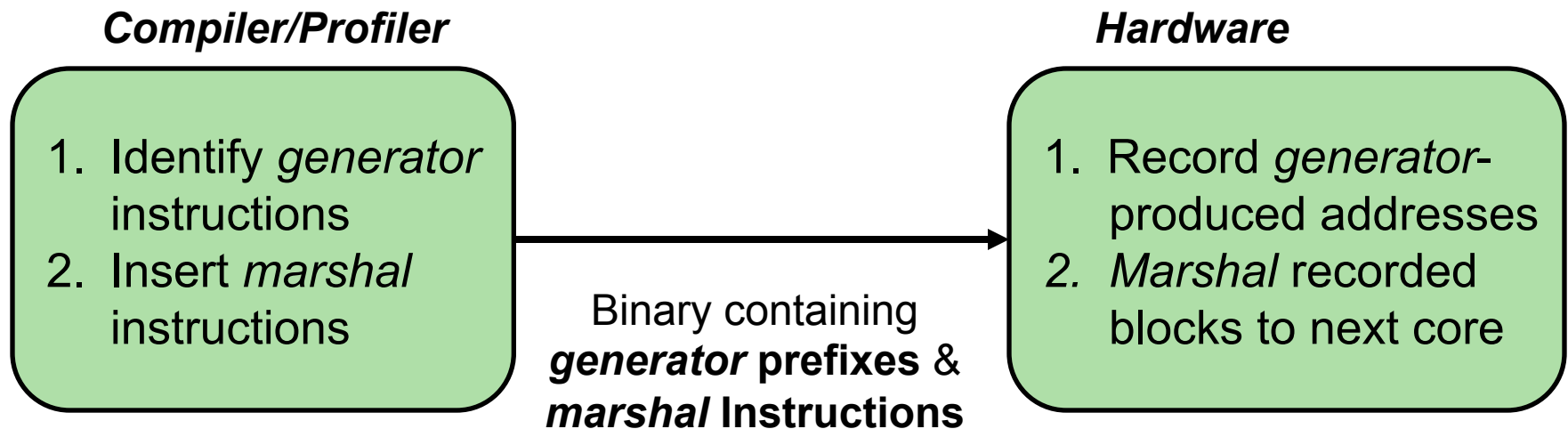
Generator instruction:

The last instruction to write to an inter-segment cache block in a segment

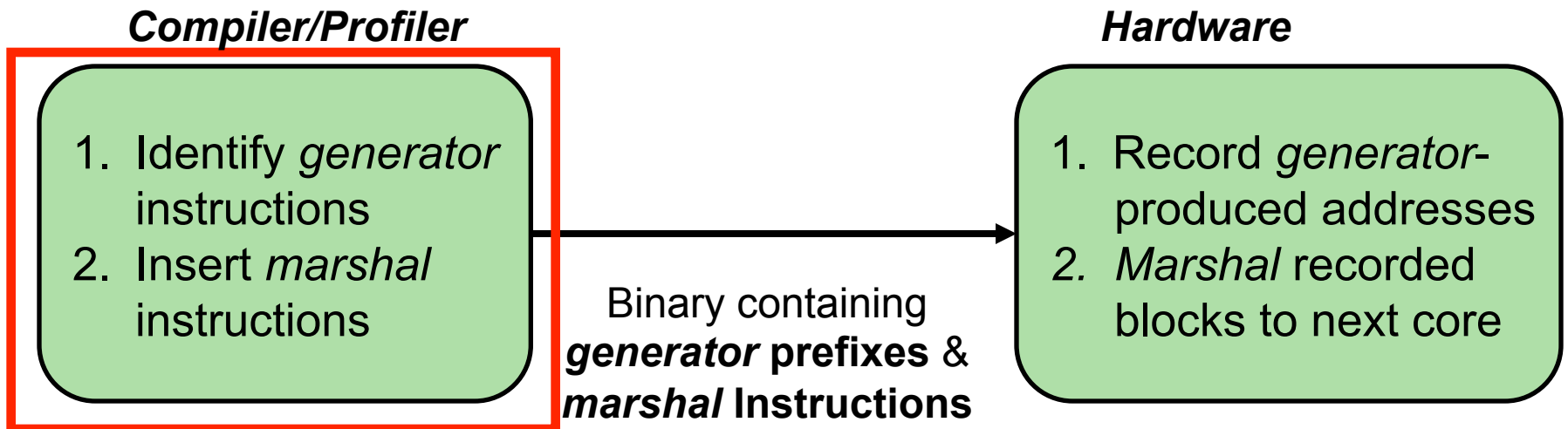
Key Observation and Idea

- Observation: Set of generator instructions is stable over execution time and across input sets
- Idea:
 - Identify the generator instructions
 - Record cache blocks produced by generator instructions
 - Proactively send such cache blocks to the next segment's core before initiating the next segment
- Suleman et al., “Data Marshaling for Multi-Core Architectures,” ISCA 2010, IEEE Micro Top Picks 2011.

Data Marshaling



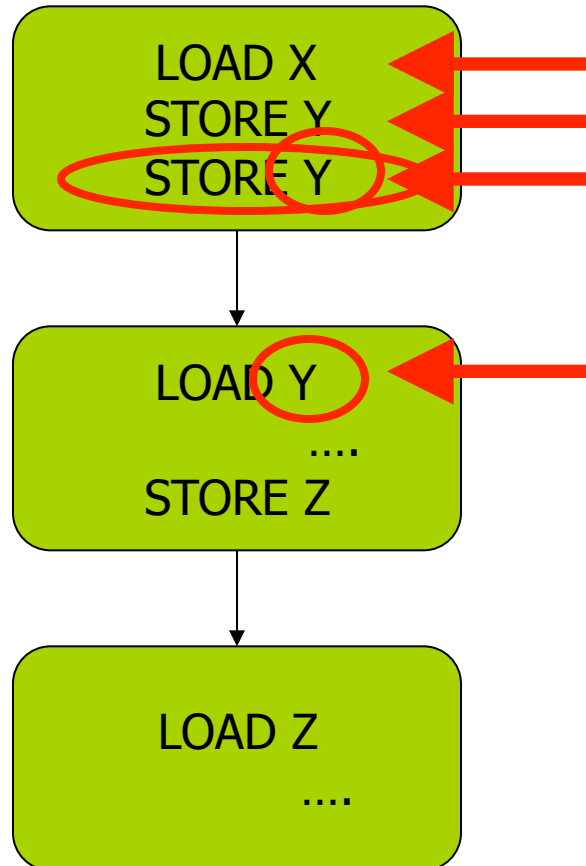
Data Marshaling



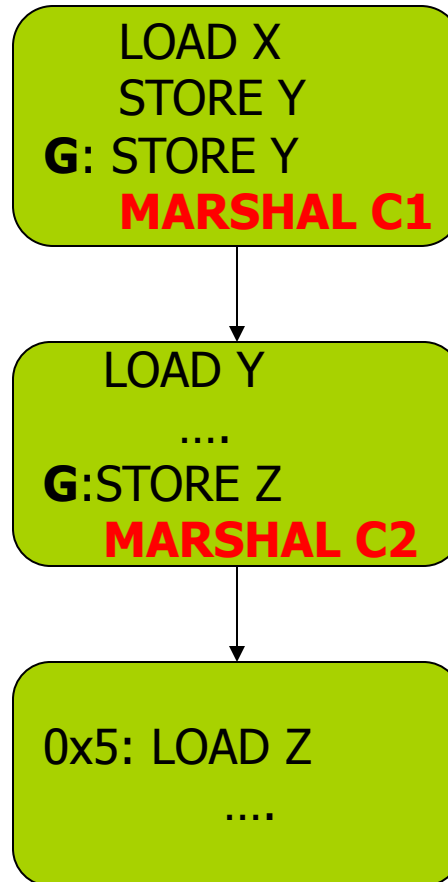
Profiling Algorithm

Inter-segment data

*Mark as Generator
Instruction*



Marshal Instructions



When to send (Marshal)

Where to send (C1)

DM Support/Cost

- Profiler/Compiler: Generators, marshal instructions
- ISA: Generator prefix, marshal instructions
- Library/Hardware: Bind next segment ID to a physical core

- Hardware
 - Marshal Buffer
 - Stores physical addresses of cache blocks to be marshaled
 - 16 entries enough for almost all workloads → 96 bytes per core
 - Ability to execute generator prefixes and marshal instructions
 - Ability to push data to another cache

DM: Advantages, Disadvantages

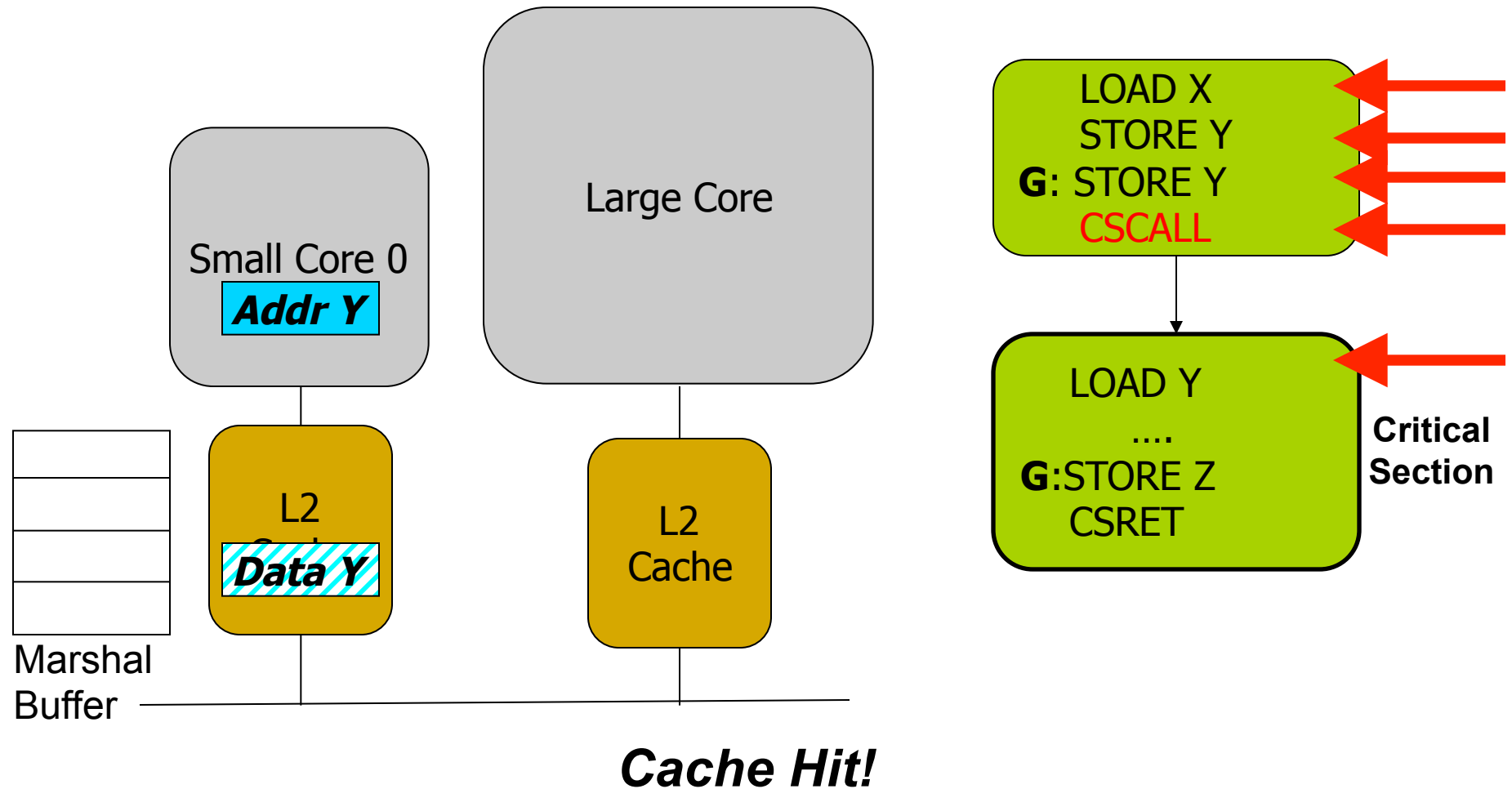
■ Advantages

- ❑ **Timely data transfer**: Push data to core before needed
- ❑ **Can marshal any arbitrary sequence of lines**: Identifies generators, not patterns
- ❑ **Low hardware cost**: Profiler marks generators, no need for hardware to find them

■ Disadvantages

- ❑ **Requires profiler and ISA support**
- ❑ **Not always accurate (generator set is conservative)**: Pollution at remote core, wasted bandwidth on interconnect
 - Not a large problem as number of inter-segment blocks is small

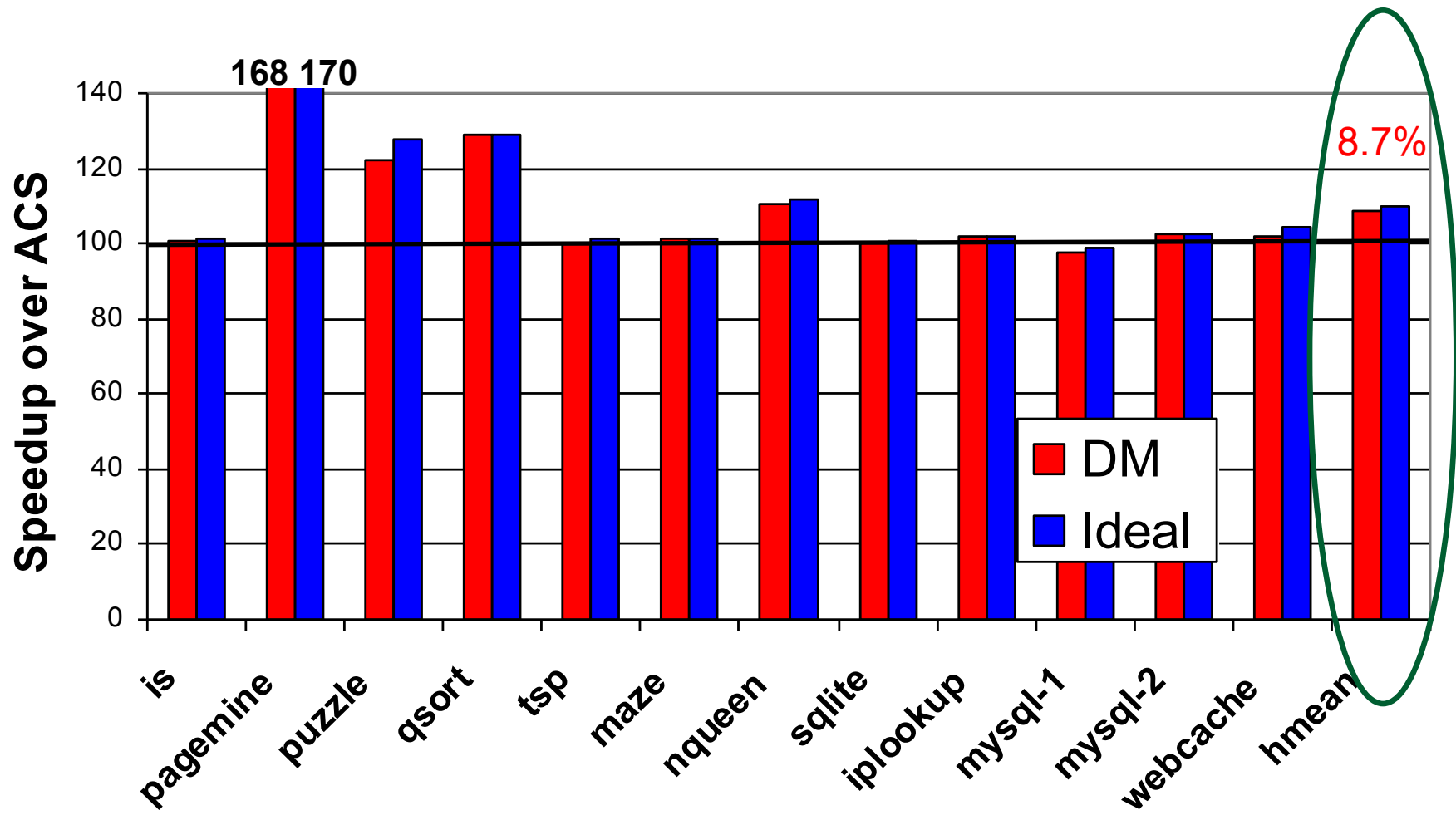
Accelerated Critical Sections with DM



Accelerated Critical Sections: Methodology

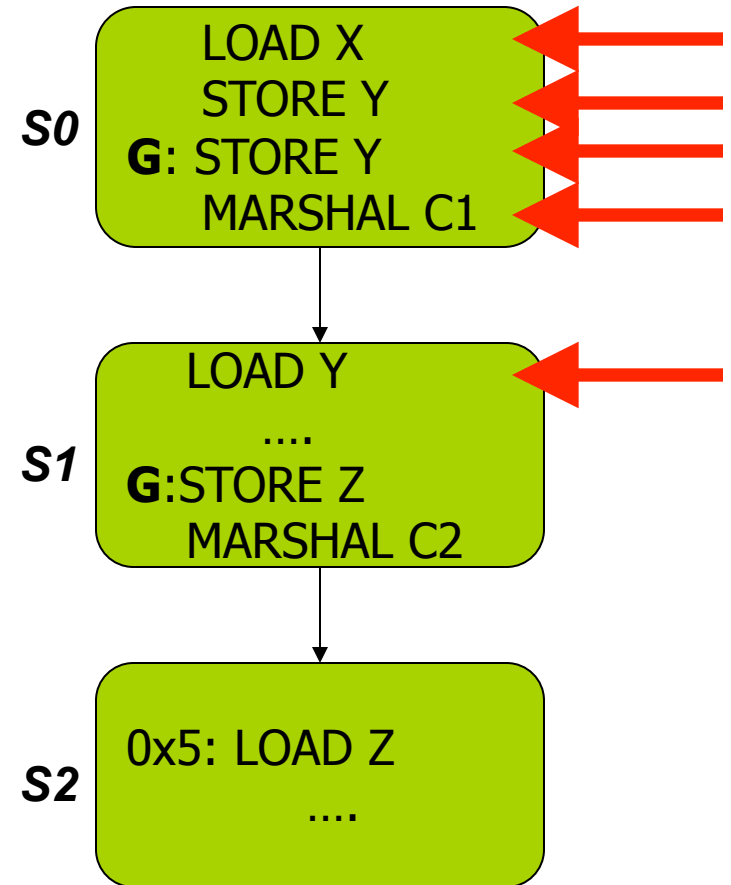
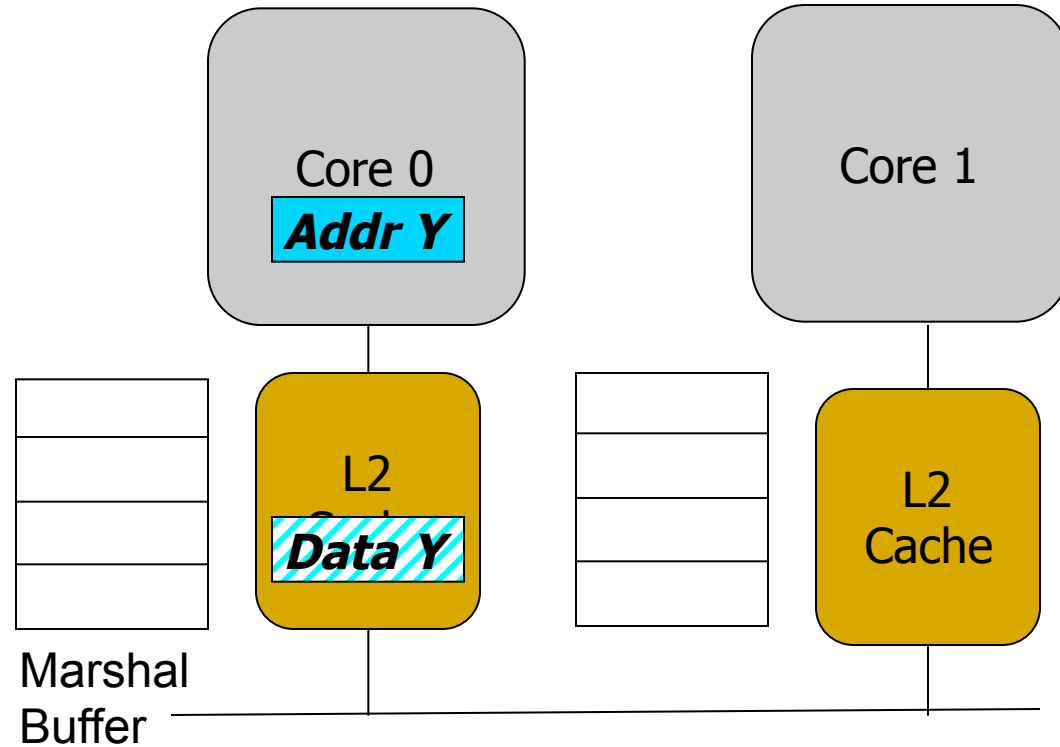
- Workloads: 12 critical section intensive applications
 - Data mining kernels, sorting, database, web, networking
 - Different training and simulation input sets
- Multi-core x86 simulator
 - 1 large and 28 small cores
 - Aggressive stream prefetcher employed at each core
- Details:
 - Large core: 2GHz, out-of-order, 128-entry ROB, 4-wide, 12-stage
 - Small core: 2GHz, in-order, 2-wide, 5-stage
 - Private 32 KB L1, private 256KB L2, 8MB shared L3
 - On-chip interconnect: Bi-directional ring, 5-cycle hop latency

DM on Accelerated Critical Sections: Results



Pipeline Parallelism

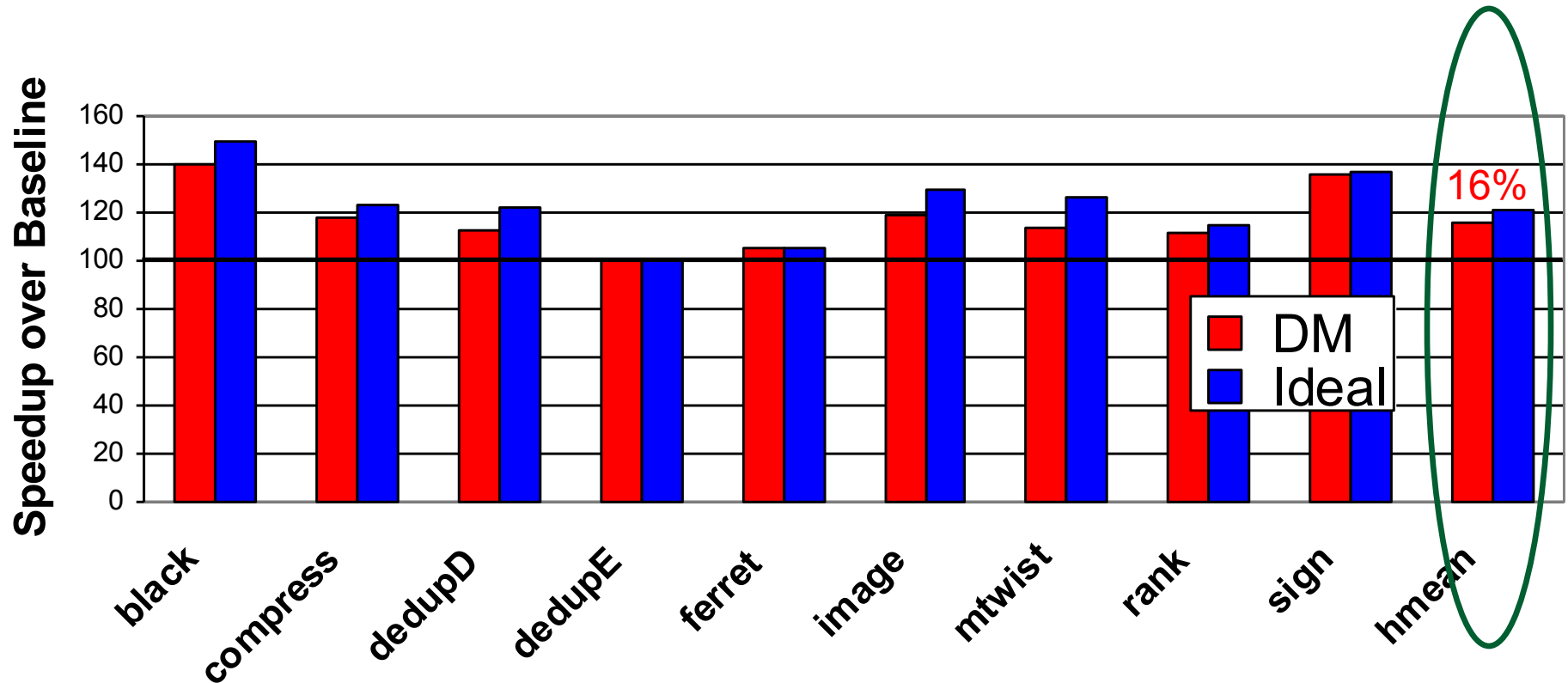
Cache Hit!



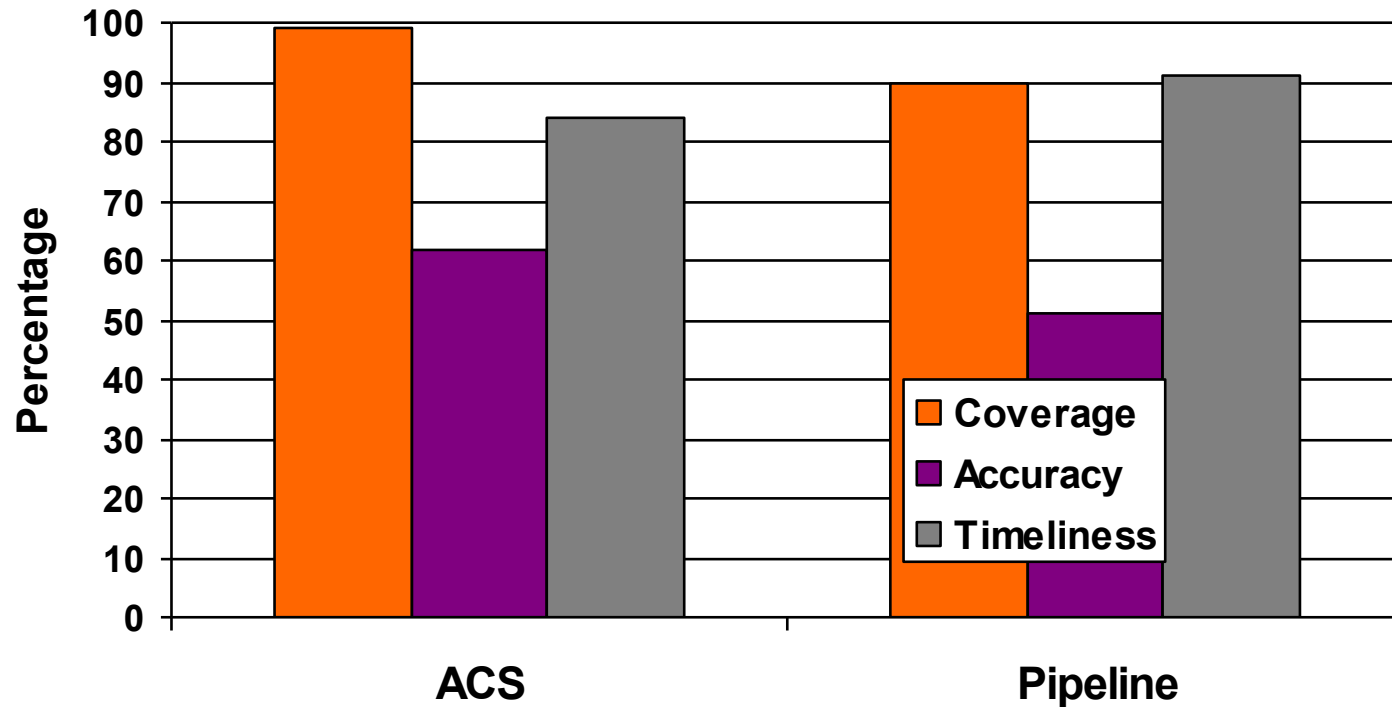
Pipeline Parallelism: Methodology

- Workloads: 9 applications with pipeline parallelism
 - Financial, compression, multimedia, encoding/decoding
 - Different training and simulation input sets
- Multi-core x86 simulator
 - 32-core CMP: 2GHz, in-order, 2-wide, 5-stage
 - Aggressive stream prefetcher employed at each core
 - Private 32 KB L1, private 256KB L2, 8MB shared L3
 - On-chip interconnect: Bi-directional ring, 5-cycle hop latency

DM on Pipeline Parallelism: Results



DM Coverage, Accuracy, Timeliness



- High coverage of inter-segment misses in a timely manner
- Medium accuracy does not impact performance
 - Only 5.0 and 6.8 cache blocks marshaled for average segment

Scaling Results

- DM performance improvement increases with
 - More cores
 - Higher interconnect latency
 - Larger private L2 caches
- Why?
 - Inter-segment data misses become a larger bottleneck
 - More cores → More communication
 - Higher latency → Longer stalls due to communication
 - Larger L2 cache → Communication misses remain

Other Applications of Data Marshaling

- Can be applied to other Staged Execution models
 - Task parallelism models
 - Cilk, Intel TBB, Apple Grand Central Dispatch
 - Special-purpose remote functional units
 - Computation spreading [Chakraborty et al., ASPLOS' 06]
 - Thread motion/migration [e.g., Rangan et al., ISCA' 09]
- Can be an enabler for more aggressive SE models
 - Lowers the cost of data migration
 - an important overhead in remote execution of code segments
 - Remote execution of finer-grained tasks can become more feasible → finer-grained parallelization in multi-cores

Data Marshaling Summary

- **Inter-segment data transfers between cores** limit the benefit of promising Staged Execution (SE) models
- Data Marshaling is a hardware/software cooperative solution: **detect inter-segment data generator instructions and push their data to next segment's core**
 - ❑ Significantly reduces cache misses for inter-segment data
 - ❑ Low cost, high-coverage, timely for arbitrary address sequences
 - ❑ Achieves most of the potential of eliminating such misses
- Applicable to several existing Staged Execution models
 - ❑ Accelerated Critical Sections: 9% performance benefit
 - ❑ Pipeline Parallelism: 16% performance benefit
- Can enable new models → **very fine-grained remote execution**

A Case for Asymmetry Everywhere

Onur Mutlu,

"Asymmetry Everywhere (with Automatic Resource Management)"

CRA Workshop on Advancing Computer Architecture Research: Popular
Parallel Programming, San Diego, CA, February 2010.

Position paper

The Setting

- Hardware resources are shared among many threads/apps in a many-core based system
 - Cores, caches, interconnects, memory, disks, power, lifetime, ...
- Management of these resources is a very difficult task
 - When optimizing parallel/multiprogrammed workloads
 - Threads interact unpredictably/unfairly in shared resources
- Power/energy is arguably the most valuable shared resource
 - Main limiter to efficiency and performance

Shield the Programmer from Shared Resources

- Writing even sequential software is hard enough
 - Optimizing code for a complex shared-resource parallel system will be a nightmare for most programmers
- Programmer should not worry about (hardware) resource management
 - What should be executed where with what resources
- Future cloud computer architectures should be designed to
 - Minimize programmer effort to optimize (parallel) programs
 - Maximize runtime system's effectiveness in automatic shared resource management

Shared Resource Management: Goals

- Future many-core systems should manage power and performance automatically across threads/applications
- Minimize energy/power consumption
- While satisfying performance/SLA requirements
 - Provide predictability and Quality of Service
- Minimize programmer effort
 - In creating optimized parallel programs
- Asymmetry and configurability in system resources essential to achieve these goals

Asymmetry Enables Customization

c	c	c	c
c	c	c	c
c	c	c	c
c	c	c	c

Symmetric

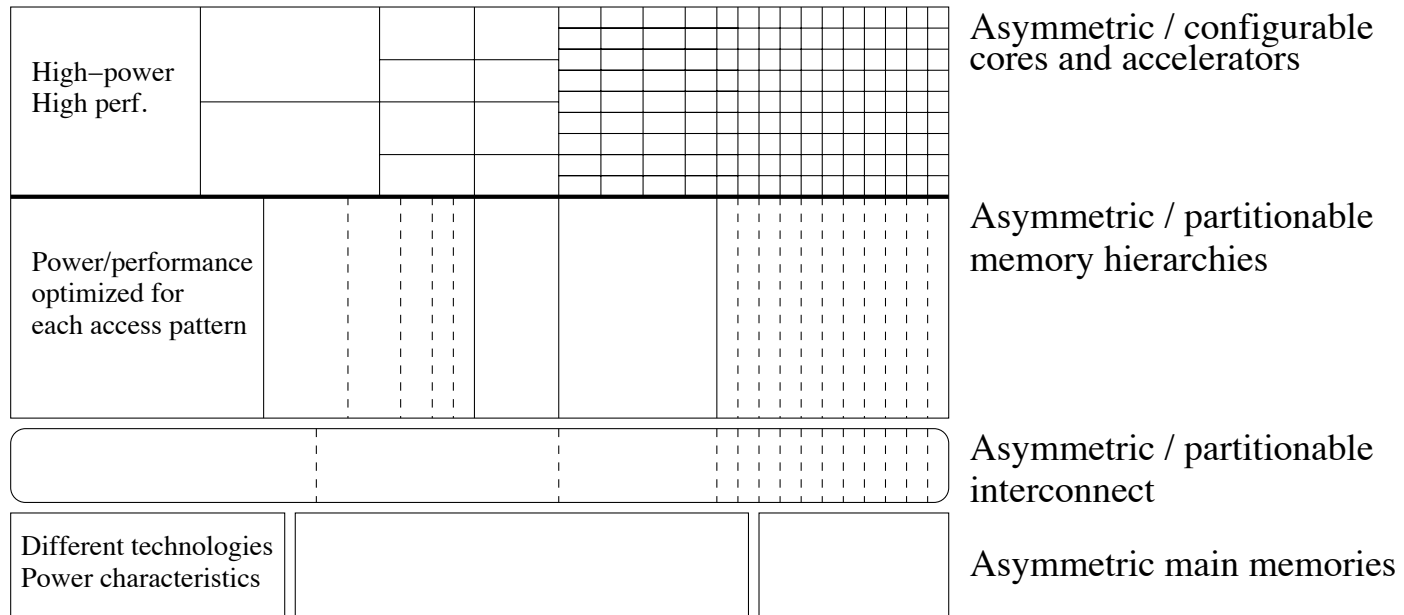
C1		C2	
		C3	
C4	C4	C4	C4
C5	C5	C5	C5

Asymmetric

- Symmetric: One size fits all
 - Energy and performance suboptimal for different phase behaviors
- Asymmetric: Enables tradeoffs and customization
 - Processing requirements vary across applications and phases
 - Execute code on best-fit resources (minimal energy, adequate perf.)

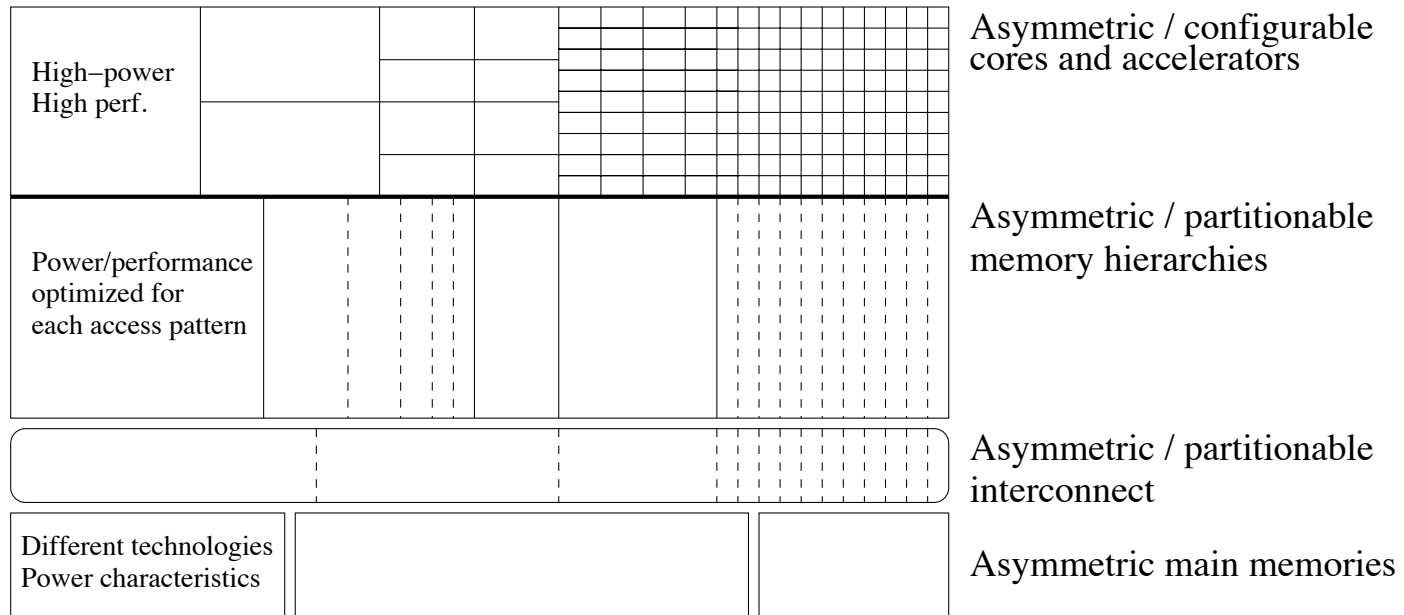
Thought Experiment: Asymmetry Everywhere

- Design each hardware resource with **asymmetric, (re-)configurable, partitionable components**
 - ❑ Different power/performance/reliability characteristics
 - ❑ To fit different computation/access/communication patterns



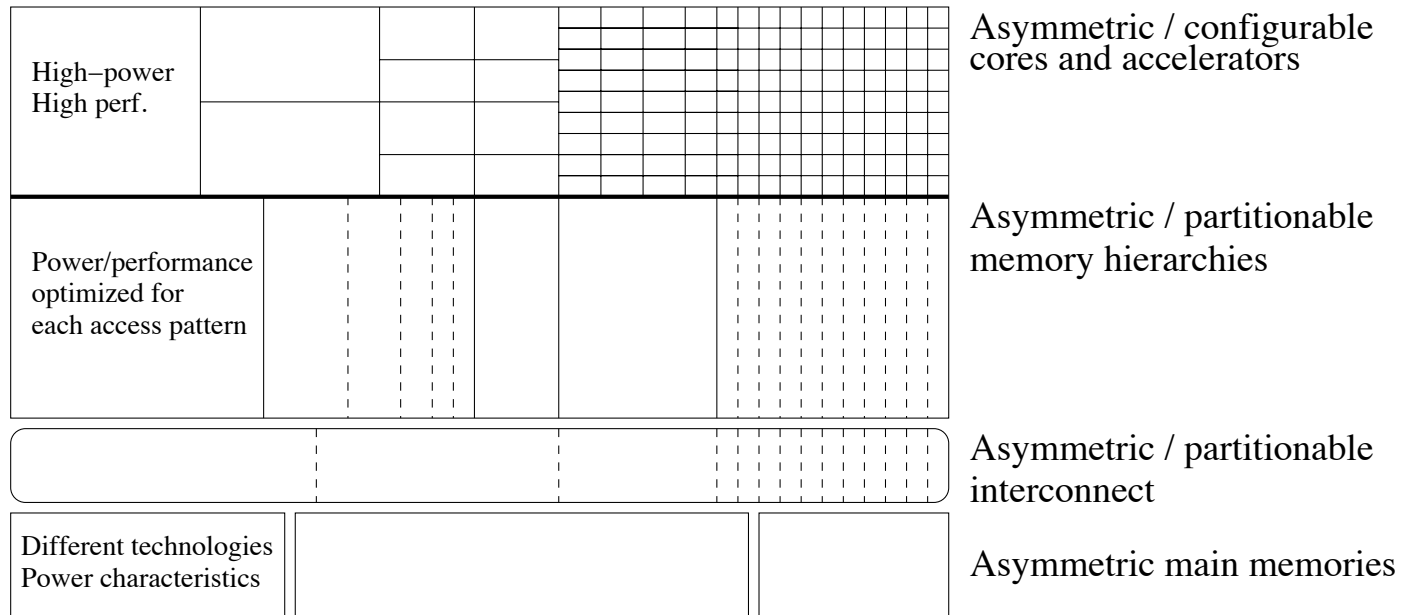
Thought Experiment: Asymmetry Everywhere

- Design the **runtime system (HW & SW)** to **automatically choose** the best-fit components for each workload/phase
 - Satisfy performance/SLA with minimal energy
 - **Dynamically stitch together the “best-fit” chip for each phase**



Thought Experiment: Asymmetry Everywhere

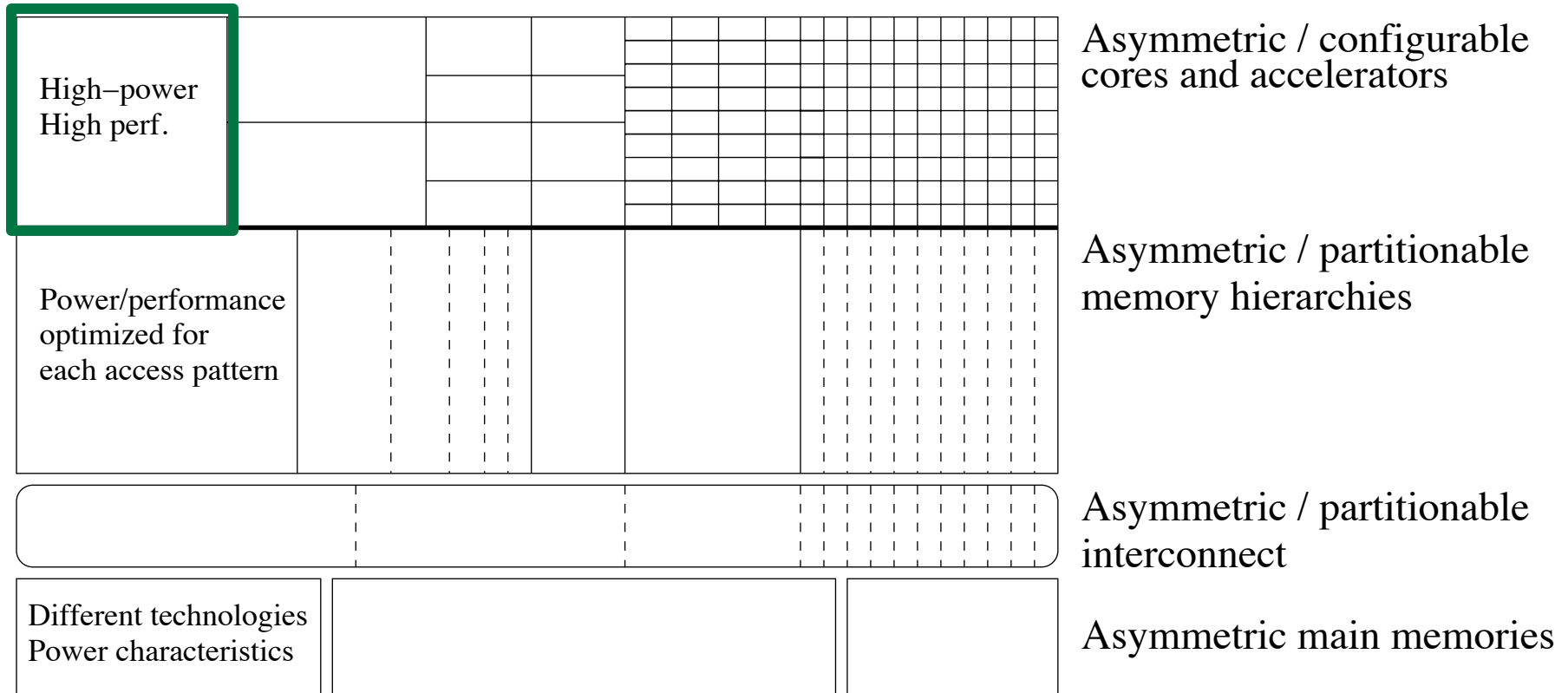
- **Morph software components** to match asymmetric HW components
 - Multiple versions for different resource characteristics



Many Research and Design Questions

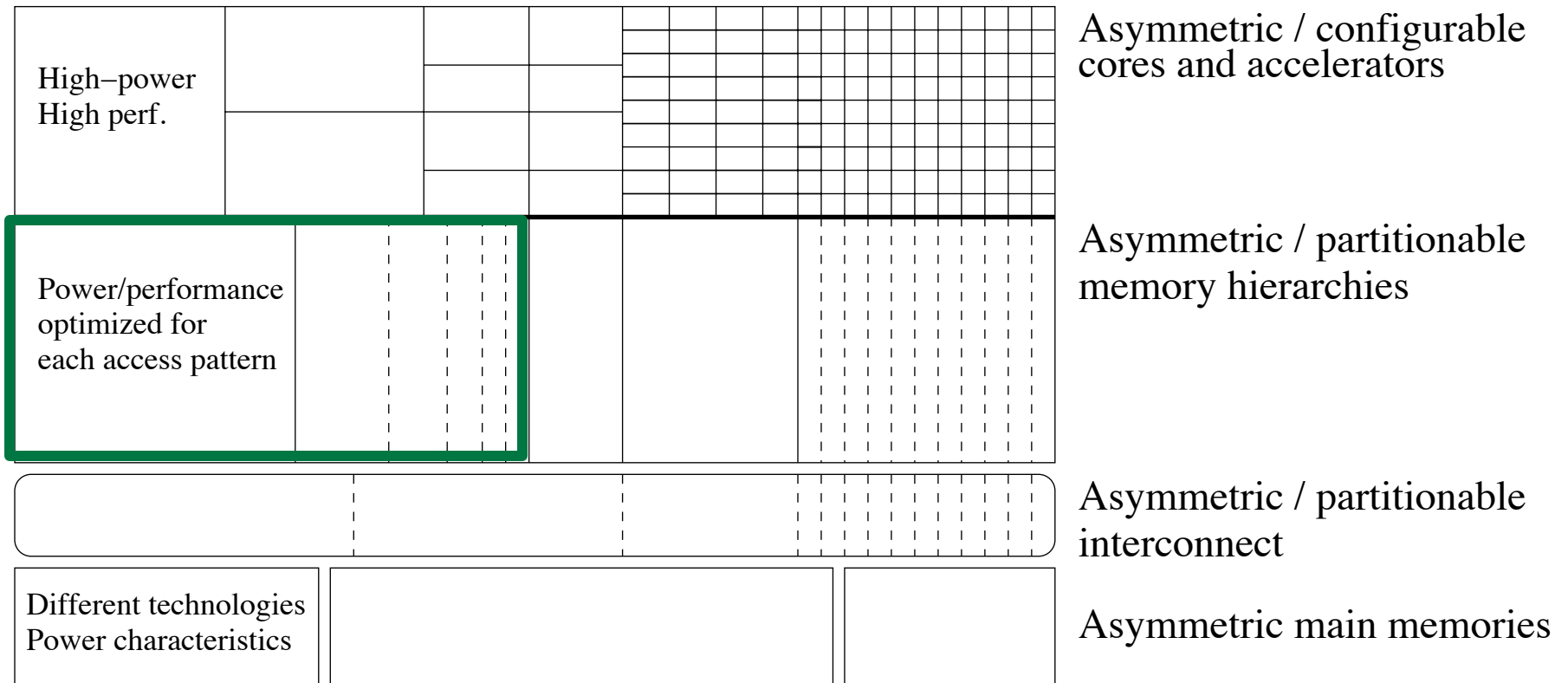
- How to design asymmetric components?
 - Fixed, partitionable, reconfigurable components?
 - What types of asymmetry? Access patterns, technologies?
- What monitoring to perform cooperatively in HW/SW?
 - Automatically discover phase/task requirements
- How to design feedback/control loop between components and runtime system software?
- How to design the runtime to automatically manage resources?
 - Track task behavior, pick “best-fit” components for the entire workload

Exploiting Asymmetry: Simple Examples



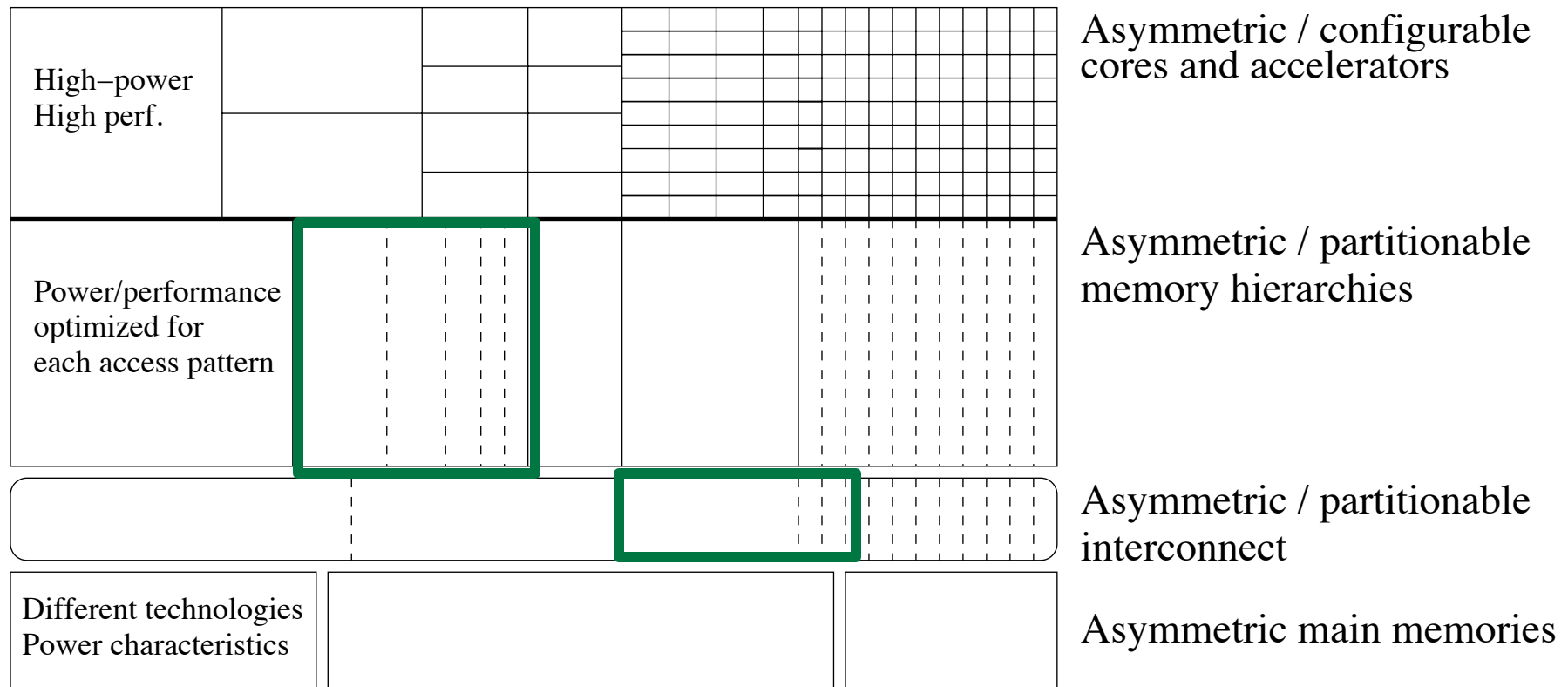
- Execute critical/serial sections on high-power, high-performance cores/resources [Suleman+ ASPLOS'09, ISCA'10, Top Picks'10'11, Joao+ ASPLOS'12]
 - Programmer can write less optimized, but more likely correct programs

Exploiting Asymmetry: Simple Examples



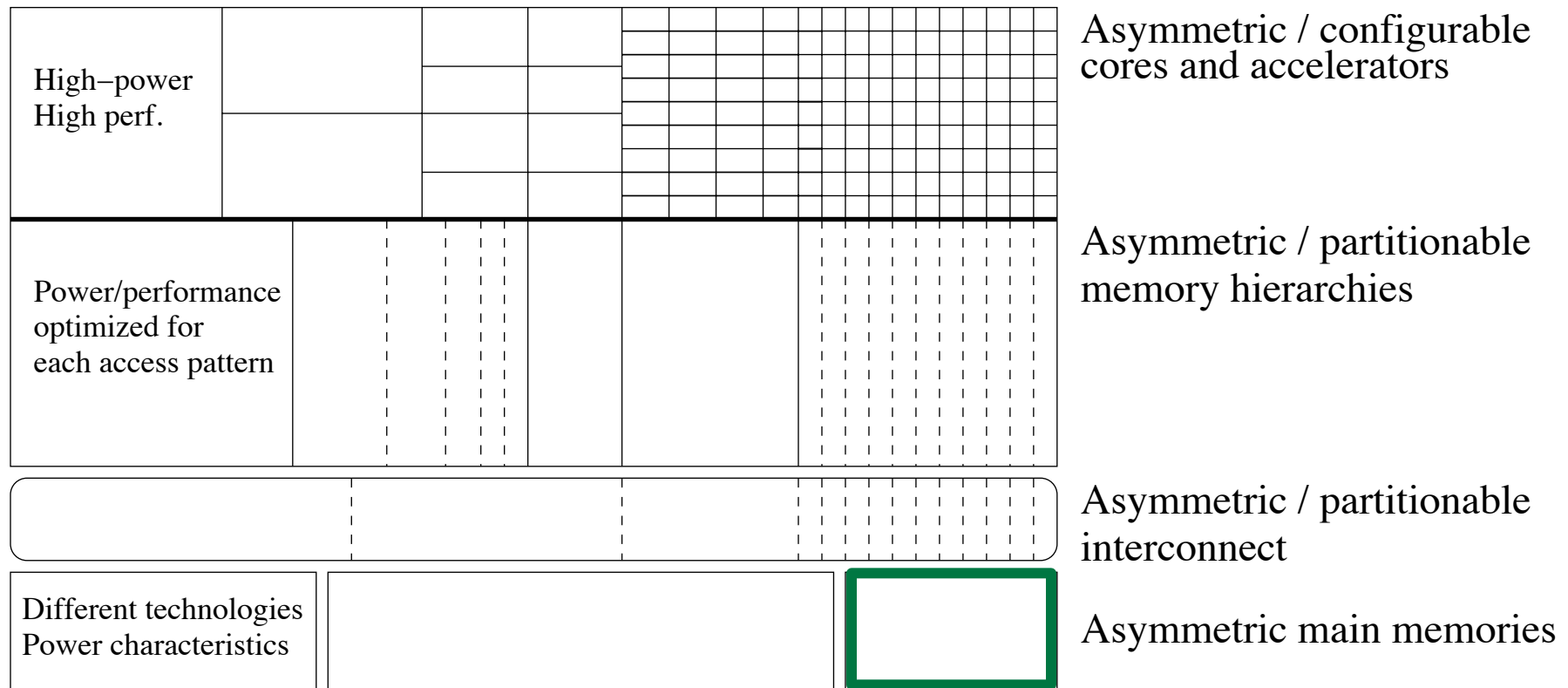
- Execute streaming “memory phases” on streaming-optimized cores and memory hierarchies
 - More efficient and higher performance than general purpose hierarchy

Exploiting Asymmetry: Simple Examples



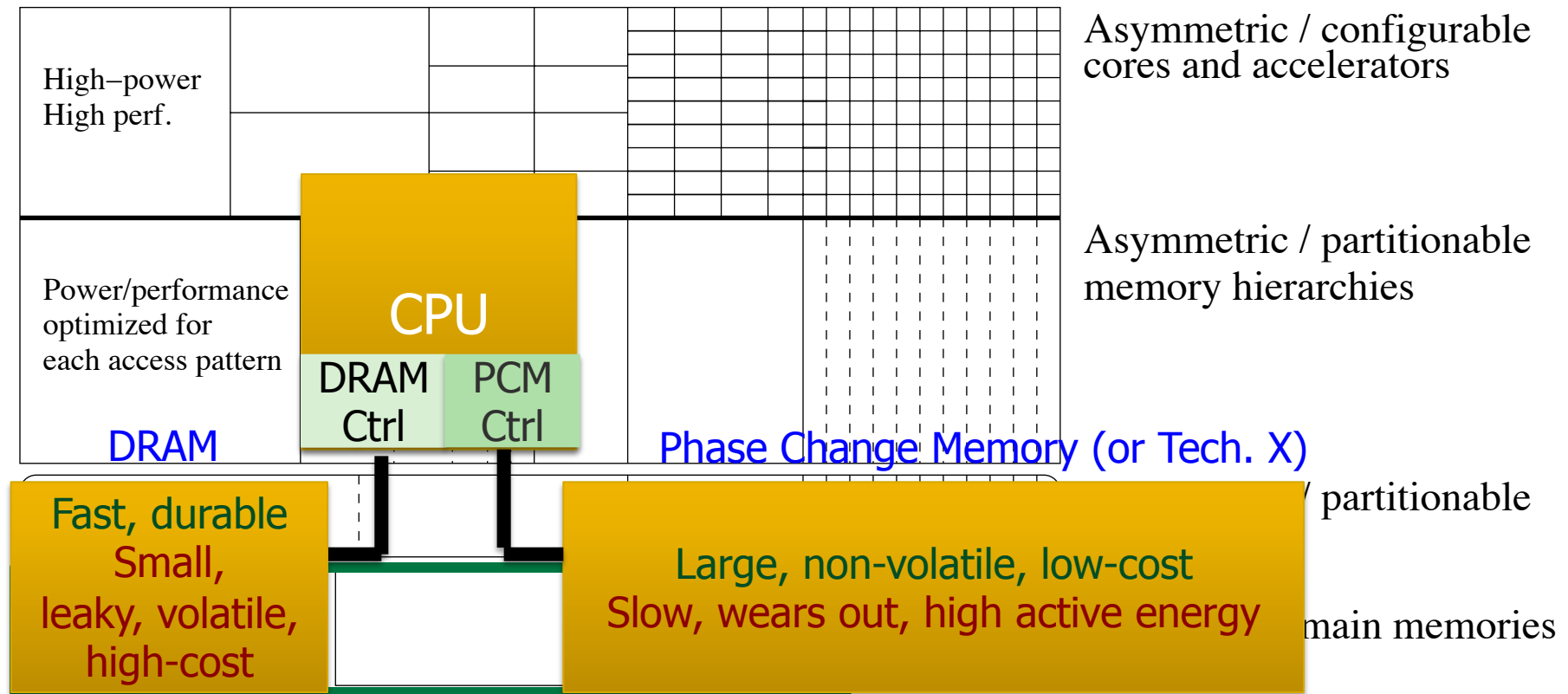
- Partition memory controller and on-chip network bandwidth asymmetrically among threads [Kim+ HPCA 2010, MICRO 2010, Top Picks 2011] [Nychis+ HotNets 2010] [Das+ MICRO 2009, ISCA 2010, Top Picks 2011]
 - Higher performance and energy-efficiency than symmetric/free-for-all

Exploiting Asymmetry: Simple Examples



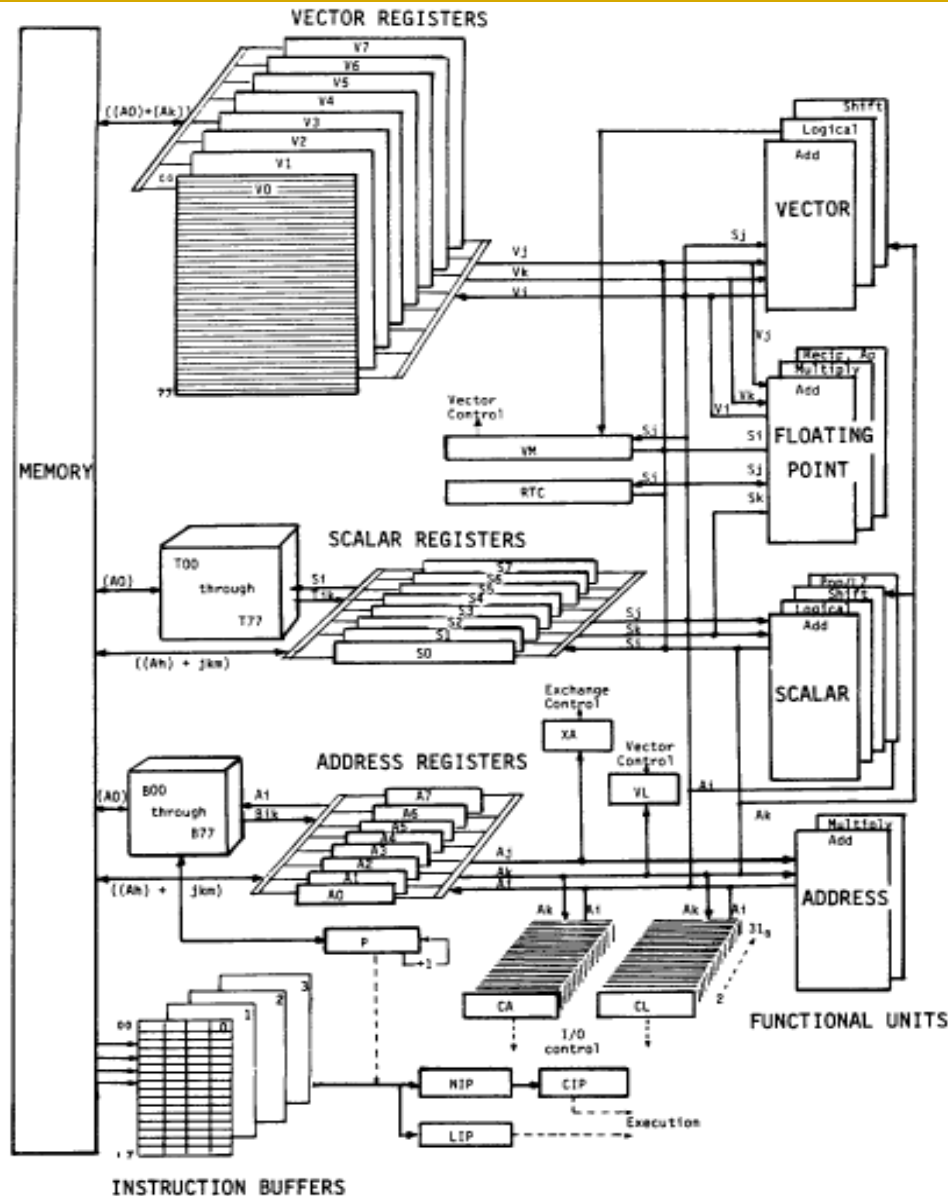
- Have multiple different memory scheduling policies; apply them to different sets of threads based on thread behavior [Kim+ MICRO 2010, Top Picks 2011] [Ausavarungnirun, ISCA 2012]
 - Higher performance and fairness than a homogeneous policy

Exploiting Asymmetry: Simple Examples



- Build main memory with different technologies with different characteristics (energy, latency, wear, bandwidth) [Meza+ IEEE CAL'12]
 - Map pages/applications to the best-fit memory resource
 - Higher performance and energy-efficiency than single-level memory

Vector Machine Organization (CRAY-1)



- CRAY-1
- Russell, “The CRAY-1 computer system,” CACM 1978.
- Scalar and vector modes
- 8 64-element vector registers
- 64 bits per element
- 16 memory banks
- 8 64-bit scalar registers
- 8 24-bit address registers

Research in Asymmetric Multi-Core

- How to Design Asymmetric Cores
 - Static
 - Dynamic
 - Can you fuse in-order cores easily to build an OoO core?
 - How to create asymmetry
- How to divide the program to best take advantage of asymmetry?
 - Explicit vs. transparent
- How to match arbitrary program phases to the best-fitting core?
- How to minimize code/data migration overhead?
- How to satisfy shared resource requirements of different cores?

Related Ongoing/Future Work

- Dynamically asymmetric cores
- Memory system design for asymmetric cores
- Asymmetric memory systems
 - Phase Change Memory (or Technology X) + DRAM
 - Hierarchies optimized for different access patterns
- Asymmetric on-chip interconnects
 - Interconnects optimized for different application requirements
- Asymmetric resource management algorithms
 - E.g., network congestion control
- Interaction of multiprogrammed multithreaded workloads

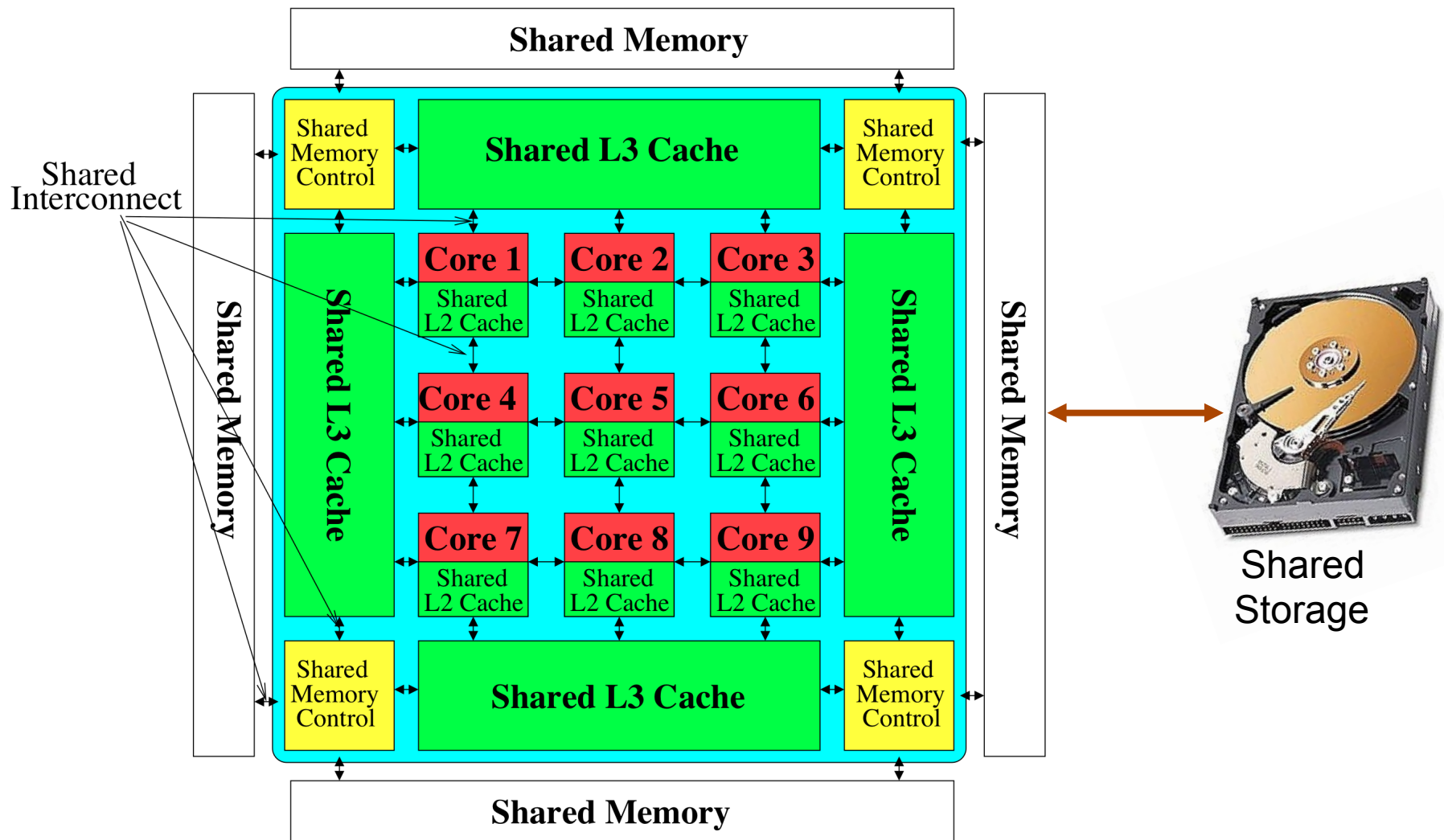
Summary: Asymmetric Design

- Applications and phases have varying performance requirements
 - Designs evaluated on multiple metrics/constraints: energy, performance, reliability, fairness, ...
 - **One-size-fits-all** design cannot satisfy all requirements and metrics: **cannot get the best of all worlds**
 - **Asymmetry** in design enables tradeoffs: **can get the best of all worlds**
 - Asymmetry in core microarch. → **Accelerated Critical Sections, BIS, DM**
→ Good parallel performance + Good serialized performance
 - Asymmetry in memory scheduling → **Thread Cluster Memory Scheduling**
→ Good throughput + good fairness
 - Asymmetry in main memory → Best of multiple technologies
 - Simple asymmetric designs can be effective and low-cost
-

Further Reading

Shared Resource Design for Multi-Core Systems

The Multi-Core System: *A Shared Resource View*



Resource Sharing Concept

- Idea: Instead of dedicating a hardware resource to a hardware context, allow multiple contexts to use it
 - Example resources: functional units, pipeline, caches, buses, memory
 - Why?
-
- + Resource sharing improves utilization/efficiency → throughput
 - When a resource is left idle by one thread, another thread can use it; no need to replicate shared data
 - + Reduces communication latency
 - For example, shared data kept in the same cache in SMT processors
 - + Compatible with the shared memory model

Resource Sharing Disadvantages

- Resource sharing results in **contention for resources**
 - When the resource is not idle, another thread cannot use it
 - If space is occupied by one thread, another thread needs to re-occupy it
- **Sometimes reduces each or some thread's performance**
 - Thread performance can be worse than when it is run alone
- **Eliminates performance isolation** → inconsistent performance across runs
 - Thread performance depends on co-executing threads
- Uncontrolled (free-for-all) sharing **degrades QoS**
 - Causes unfairness, starvation

Need to efficiently and fairly utilize shared resources

Need for QoS and Shared Resource Mgmt.

- Why is unpredictable performance (or lack of QoS) bad?
- Makes programmer's life difficult
 - An optimized program can get low performance (and performance varies widely depending on co-runners)
- Causes discomfort to user
 - An important program can starve
 - Examples from shared software resources
- Makes system management difficult
 - How do we enforce a Service Level Agreement when hardware resources are sharing is uncontrollable?

Resource Sharing vs. Partitioning

- Sharing improves throughput
 - Better utilization of space
- Partitioning provides performance isolation (predictable performance)
 - Dedicated space
- Can we get the benefits of both?
- Idea: Design shared resources such that they are efficiently utilized, controllable and partitionable
 - No wasted resource + QoS mechanisms for threads

Shared Hardware Resources

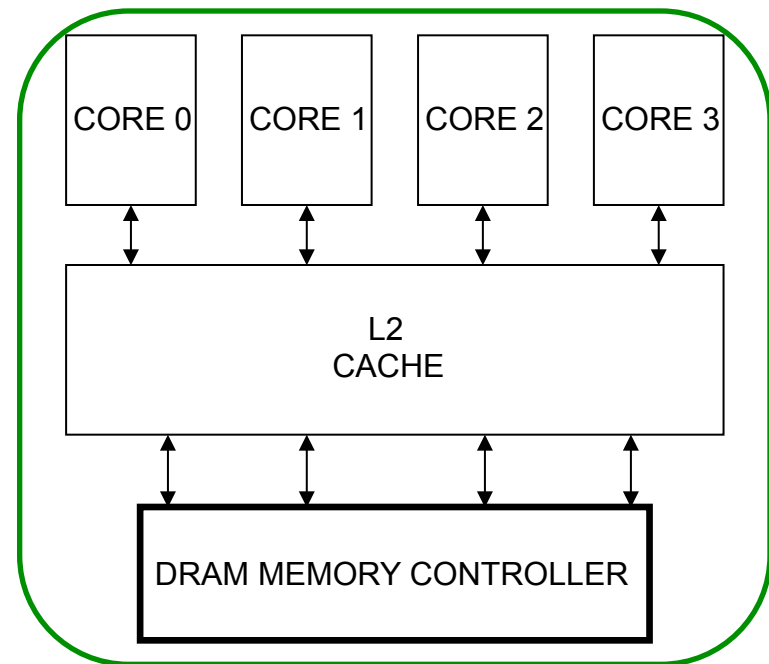
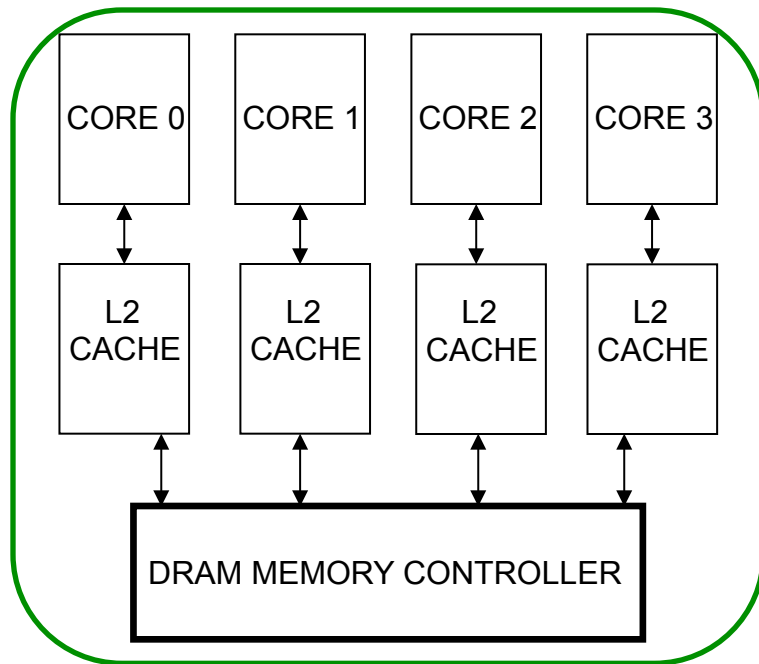
- Memory subsystem (in both MT and CMP)
 - Non-private caches
 - Interconnects
 - Memory controllers, buses, banks

- I/O subsystem (in both MT and CMP)
 - I/O, DMA controllers
 - Ethernet controllers

- Processor (in MT)
 - Pipeline resources
 - L1 caches

Multi-core Issues in Caching

- How does the cache hierarchy change in a multi-core system?
- **Private** cache: Cache belongs to one core (a shared block can be in multiple caches)
- **Shared** cache: Cache is shared by multiple cores



Shared Caches Between Cores

■ Advantages:

- ❑ High effective capacity
- ❑ **Dynamic partitioning** of available cache space
 - No fragmentation due to static partitioning
- ❑ **Easier to maintain coherence** (a cache block is in a single location)
- ❑ **Shared data and locks do not ping pong between caches**

■ Disadvantages

- ❑ Slower access
- ❑ Cores incur **conflict misses due to other cores' accesses**
 - Misses due to inter-core interference
 - Some cores can destroy the hit rate of other cores
- ❑ Guaranteeing a minimum level of service (or fairness) to each core is harder (how much space, how much bandwidth?)

Shared Caches: How to Share?

■ Free-for-all sharing

- ❑ Placement/replacement policies are the same as a single core system (usually LRU or pseudo-LRU)
- ❑ Not thread/application aware
- ❑ An incoming block evicts a block regardless of which threads the blocks belong to

■ Problems

- ❑ Inefficient utilization of cache: LRU is not the best policy
- ❑ A cache-unfriendly application can destroy the performance of a cache friendly application
- ❑ Not all applications benefit equally from the same amount of cache: free-for-all might prioritize those that do not benefit
- ❑ Reduced performance, reduced fairness

Controlled Cache Sharing

■ Utility based cache partitioning

- Qureshi and Patt, “Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches,” MICRO 2006.
- Suh et al., “A New Memory Monitoring Scheme for Memory-Aware Scheduling and Partitioning,” HPCA 2002.

■ Fair cache partitioning

- Kim et al., “Fair Cache Sharing and Partitioning in a Chip Multiprocessor Architecture,” PACT 2004.

■ Shared/private mixed cache mechanisms

- Qureshi, “Adaptive Spill-Receive for Robust High-Performance Caching in CMPs,” HPCA 2009.
- Hardavellas et al., “Reactive NUCA: Near-Optimal Block Placement and Replication in Distributed Caches,” ISCA 2009.

Efficient Cache Utilization

- Qureshi et al., “A Case for MLP-Aware Cache Replacement,” ISCA 2005.
- Seshadri et al., “The Evicted-Address Filter: A Unified Mechanism to Address both Cache Pollution and Thrashing,” PACT 2012.
- Pekhimenko et al., “Base-Delta-Immediate Compression: Practical Data Compression for On-Chip Caches,” PACT 2012.
- Pekhimenko et al., “Linearly Compressed Pages: A Main Memory Compression Framework with Low Complexity and Low Latency,” SAFARI Technical Report 2013.

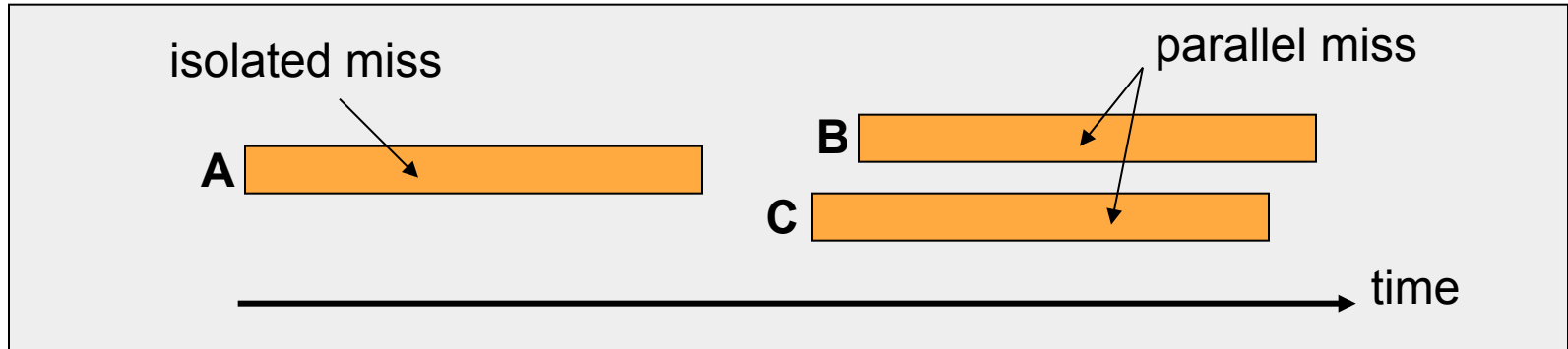
MLP-Aware Cache Replacement

Moinuddin K. Qureshi, Daniel N. Lynch, Onur Mutlu, and Yale N. Patt,

"A Case for MLP-Aware Cache Replacement"

*Proceedings of the 33rd International Symposium on Computer Architecture
(**ISCA**), pages 167-177, Boston, MA, June 2006. Slides (ppt)*

Memory Level Parallelism (MLP)



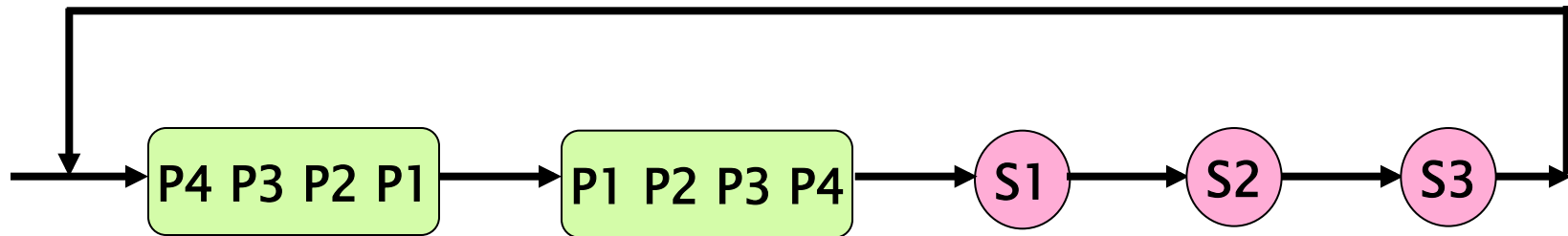
- ❑ Memory Level Parallelism (MLP) means generating and servicing multiple memory accesses in parallel [Glew' 98]
- ❑ Several techniques to improve MLP (e.g., out-of-order execution, runahead execution)
- ❑ MLP varies. Some misses are isolated and some parallel

How does this affect cache replacement?

Traditional Cache Replacement Policies

- ❑ Traditional cache replacement policies try to reduce miss count
- ❑ **Implicit assumption**: Reducing miss count reduces memory-related stall time
- ❑ Misses with varying cost/MLP **breaks** this assumption!
- ❑ Eliminating an isolated miss helps performance more than eliminating a parallel miss
- ❑ Eliminating a higher-latency miss could help performance more than eliminating a lower-latency miss

An Example



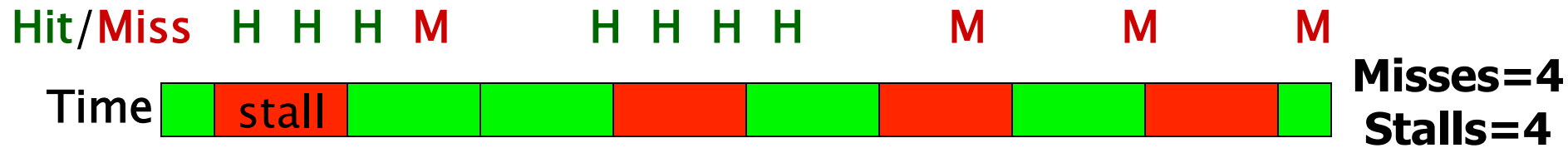
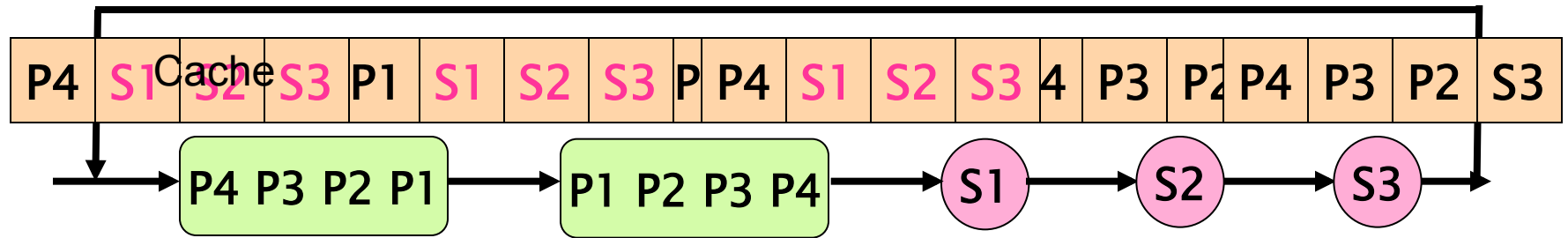
Misses to blocks P1, P2, P3, P4 can be parallel
Misses to blocks S1, S2, and S3 are isolated

Two replacement algorithms:

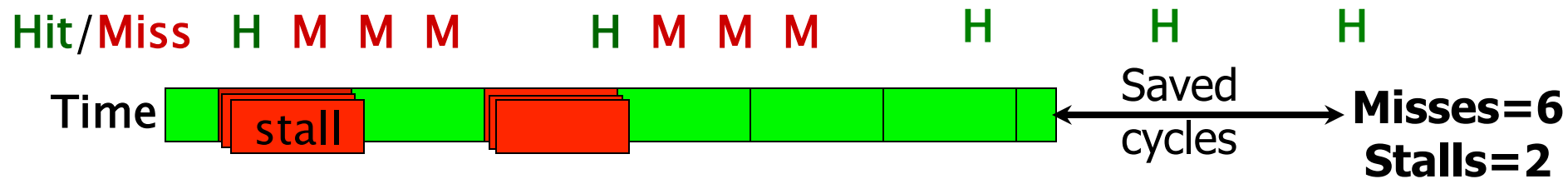
1. Minimizes miss count (Belady's OPT)
2. Reduces isolated misses (MLP-Aware)

For a fully associative cache containing 4 blocks

Fewest Misses \neq Best Performance



Belady's OPT replacement



MLP-Aware replacement

Motivation

- ❑ MLP varies. Some misses more costly than others
- ❑ MLP-aware replacement can improve performance by reducing costly misses

Outline

❑ Introduction

❑ MLP-Aware Cache Replacement

- Model for Computing Cost
- Repeatability of Cost
- A Cost-Sensitive Replacement Policy

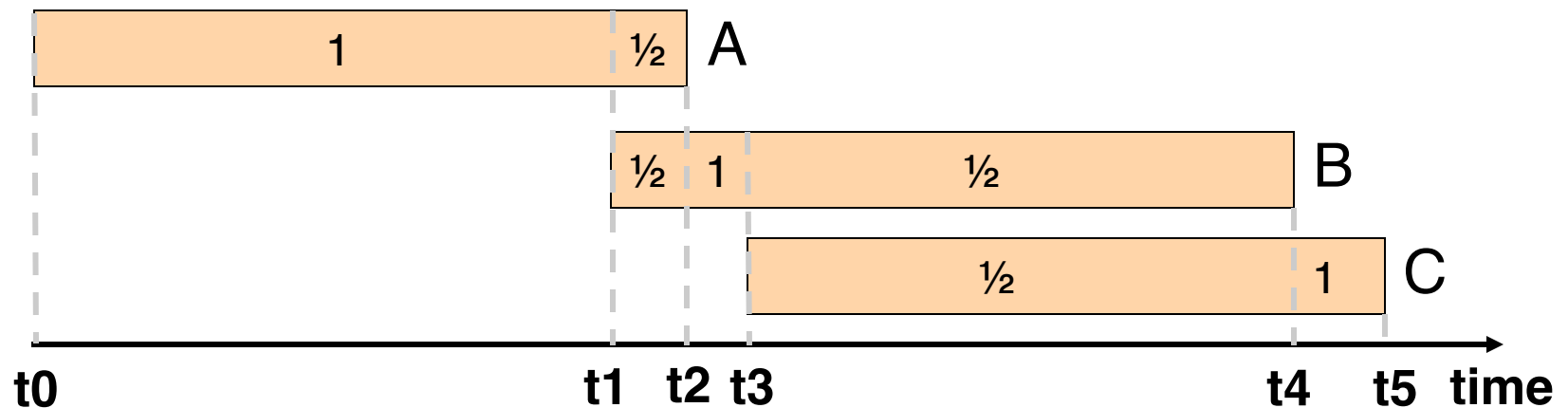
❑ Practical Hybrid Replacement

- Tournament Selection
- Dynamic Set Sampling
- Sampling Based Adaptive Replacement

❑ Summary

Computing MLP-Based Cost

- ❑ Cost of miss is number of cycles the miss stalls the processor
- ❑ Easy to compute for isolated miss
- ❑ Divide each stall cycle equally among all parallel misses



A First-Order Model

- ❑ Miss Status Holding Register (MSHR) tracks all in flight misses
- ❑ Add a field **mlp-cost** to each MSHR entry
- ❑ Every cycle for each demand entry in MSHR

$$\text{mlp-cost} += (1/N)$$

N = Number of demand misses in MSHR

Machine Configuration

❑ Processor

- aggressive, out-of-order, 128-entry instruction window

❑ L2 Cache

- 1MB, 16-way, LRU replacement, 32 entry MSHR

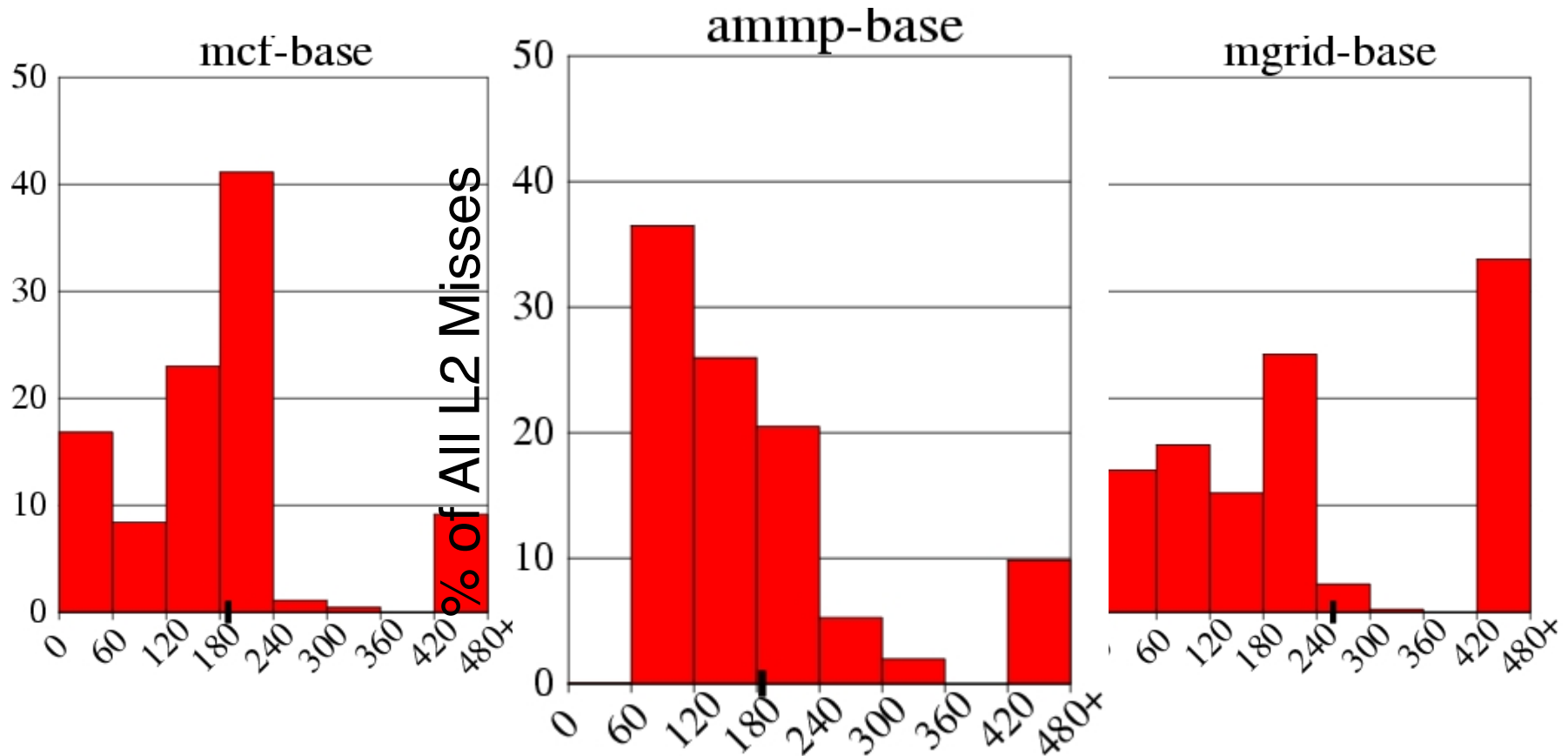
❑ Memory

- 400 cycle bank access, 32 banks

❑ Bus

- Roundtrip delay of 11 bus cycles (44 processor cycles)
-

Distribution of MLP-Based Cost

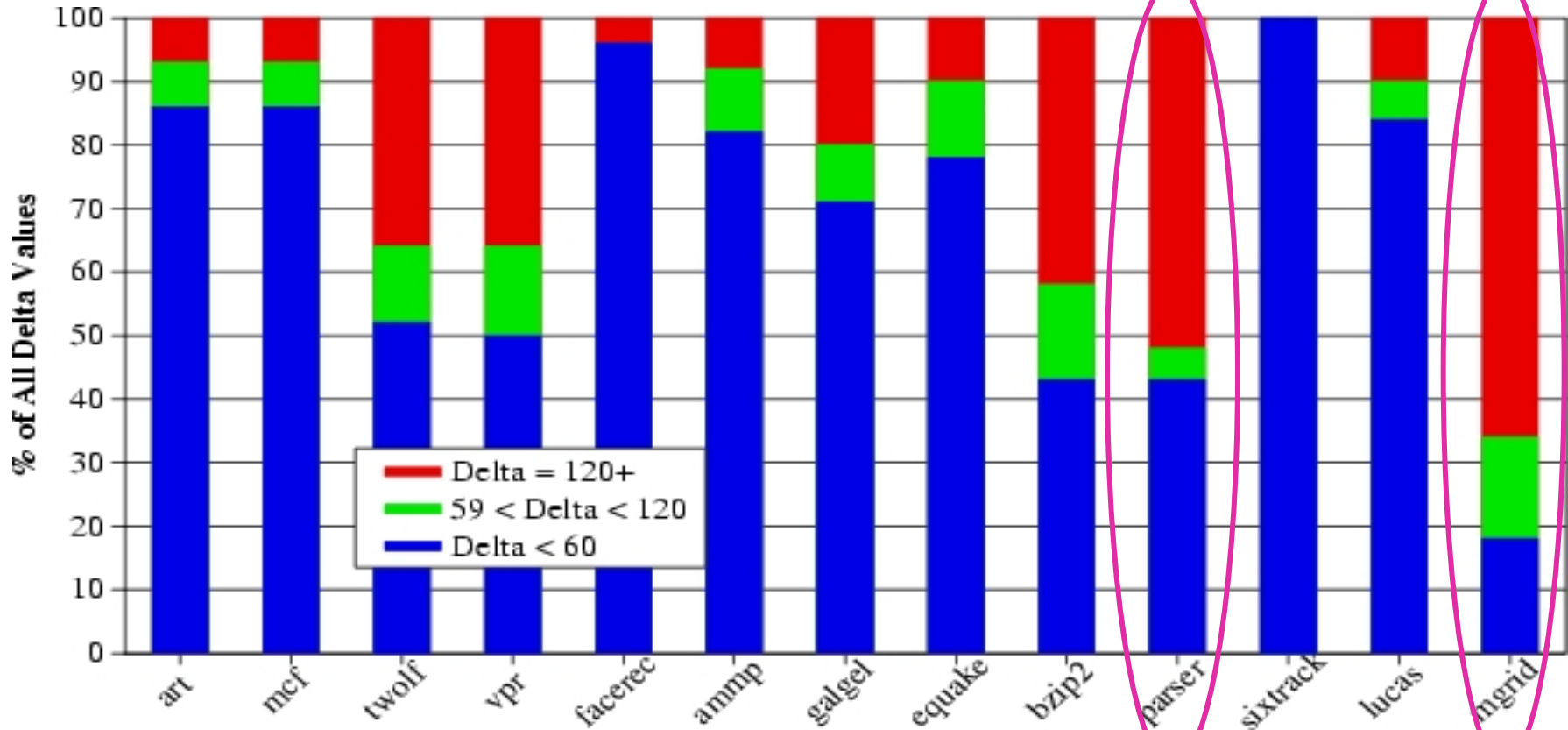


Cost varies. Does it repeat for a given cache block?

Repeatability of Cost

- ❑ An isolated miss can be parallel miss next time
- ❑ Can current cost be used to estimate future cost ?
- ❑ Let δ = difference in cost for successive miss to a block
 - Small $\delta \rightarrow$ cost repeats
 - Large $\delta \rightarrow$ cost varies significantly

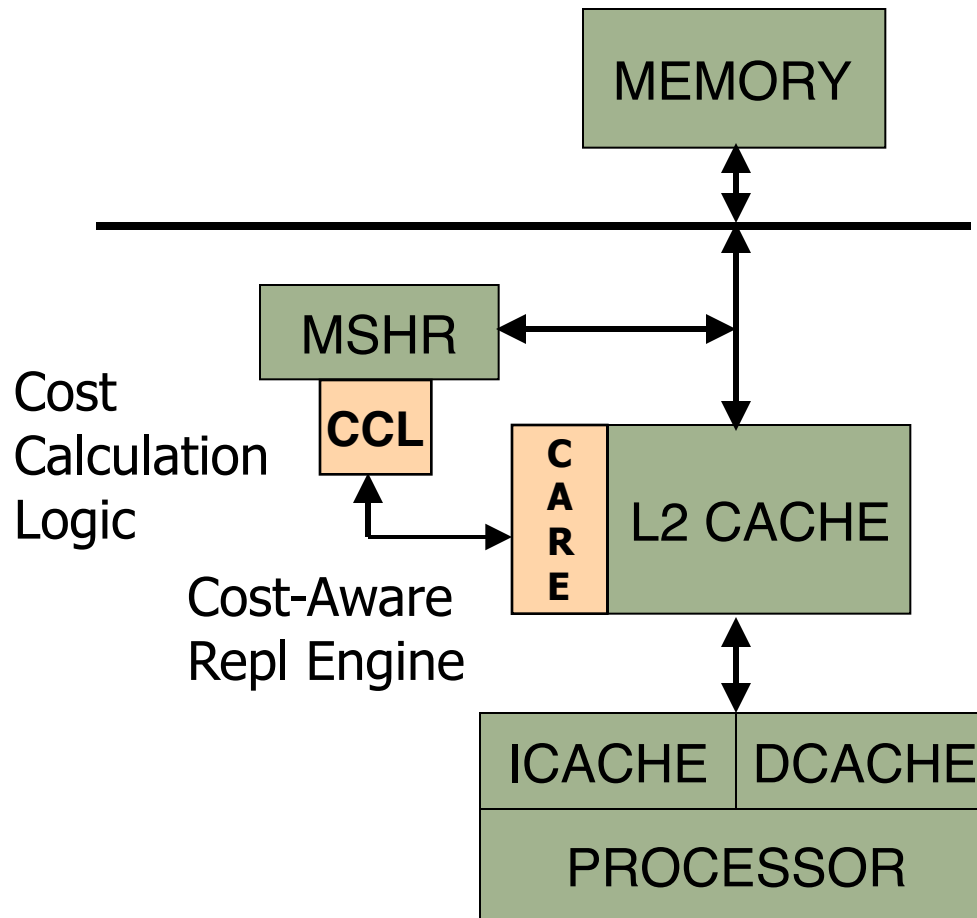
Repeatability of Cost

 $\delta < 60$ $\delta > 120$ $59 < \delta < 120$ 

❑ In general δ is small \rightarrow repeatable cost

❑ When δ is large (e.g. parser, mgrid) \rightarrow performance loss

The Framework



Quantization of Cost

Computed mlp-based cost is quantized to a 3-bit value

Design of MLP-Aware Replacement policy

- ❑ LRU considers only recency and no cost

$$Victim-LRU = \min \{ Recency(i) \}$$

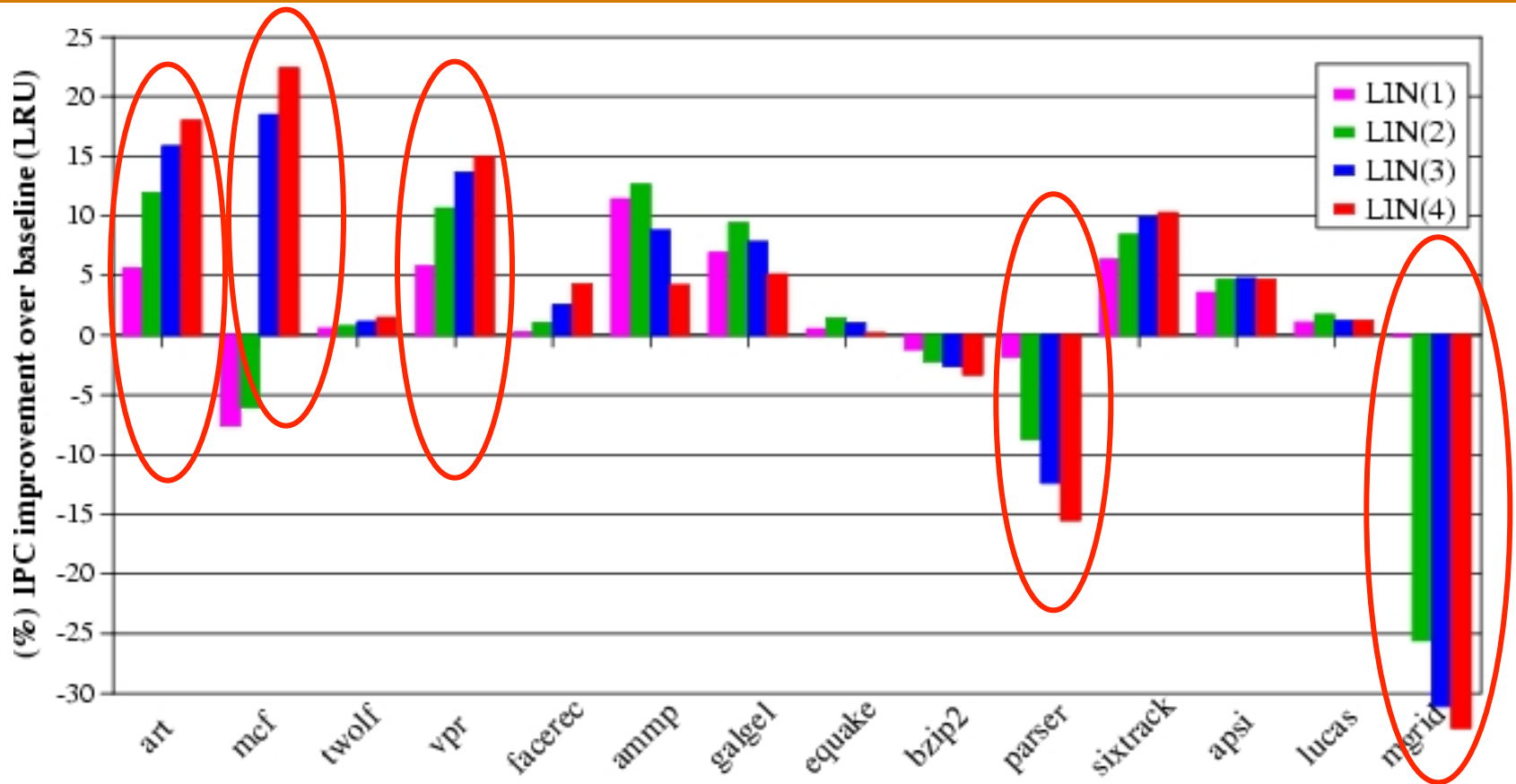
- ❑ Decisions based only on cost and no recency hurt performance. Cache stores useless high cost blocks

- ❑ A Linear (LIN) function that considers recency and cost

$$Victim-LIN = \min \{ Recency(i) + S * cost(i) \}$$

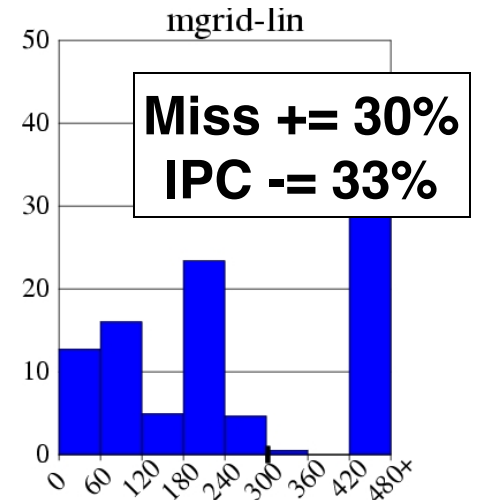
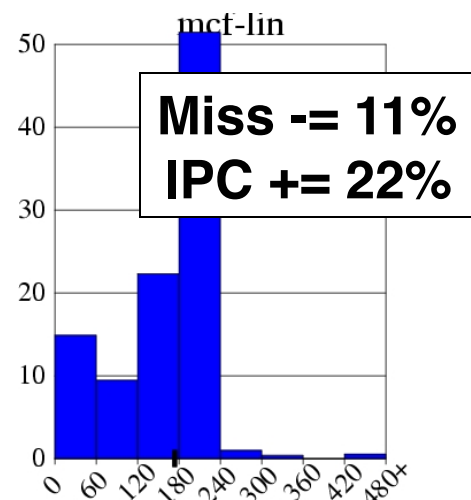
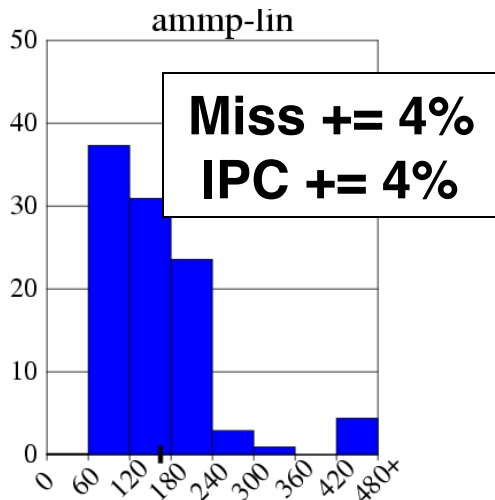
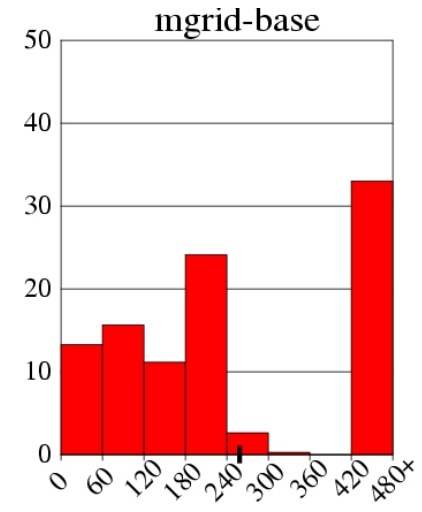
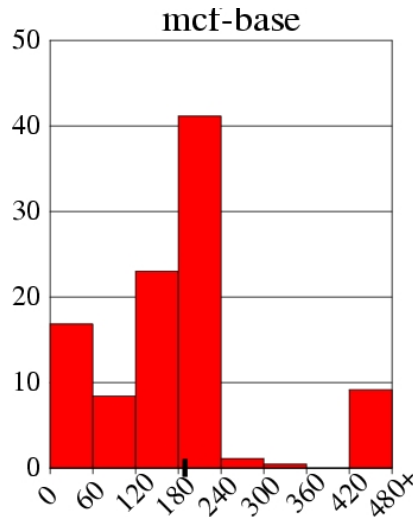
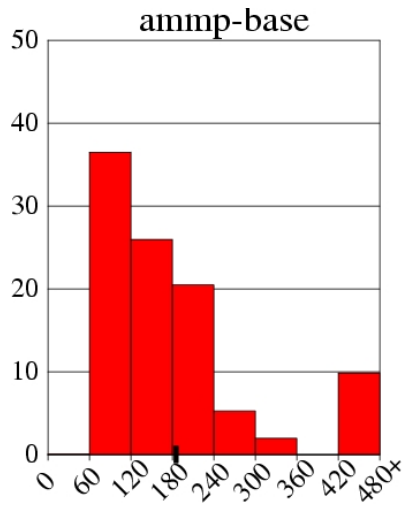
S = significance of cost. $Recency(i)$ = position in LRU stack
 $cost(i)$ = quantized cost

Results for the LIN policy



Performance loss for parser and mgrid due to large δ

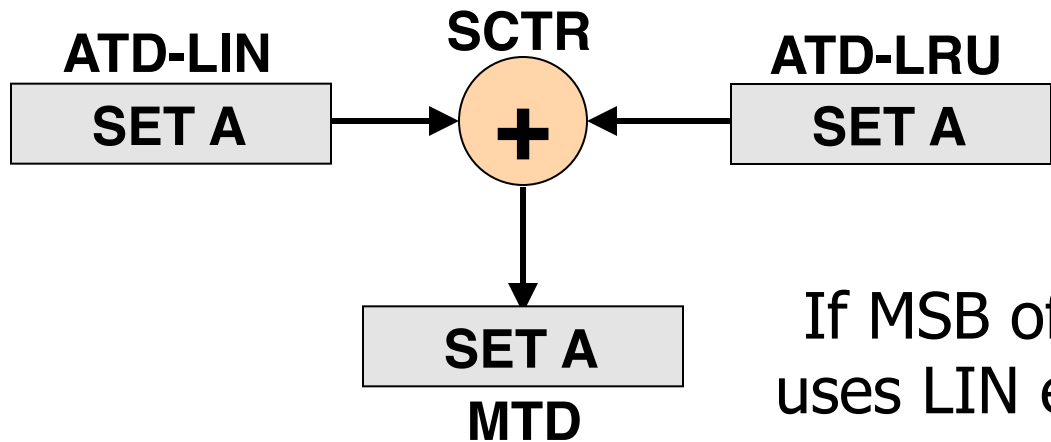
Effect of LIN policy on Cost



Outline

- ❑ Introduction
 - ❑ MLP-Aware Cache Replacement
 - Model for Computing Cost
 - Repeatability of Cost
 - A Cost-Sensitive Replacement Policy
 - ❑ Practical Hybrid Replacement
 - Tournament Selection
 - Dynamic Set Sampling
 - Sampling Based Adaptive Replacement
 - ❑ Summary
-

Tournament Selection (TSEL) of Replacement Policies for a Single Set

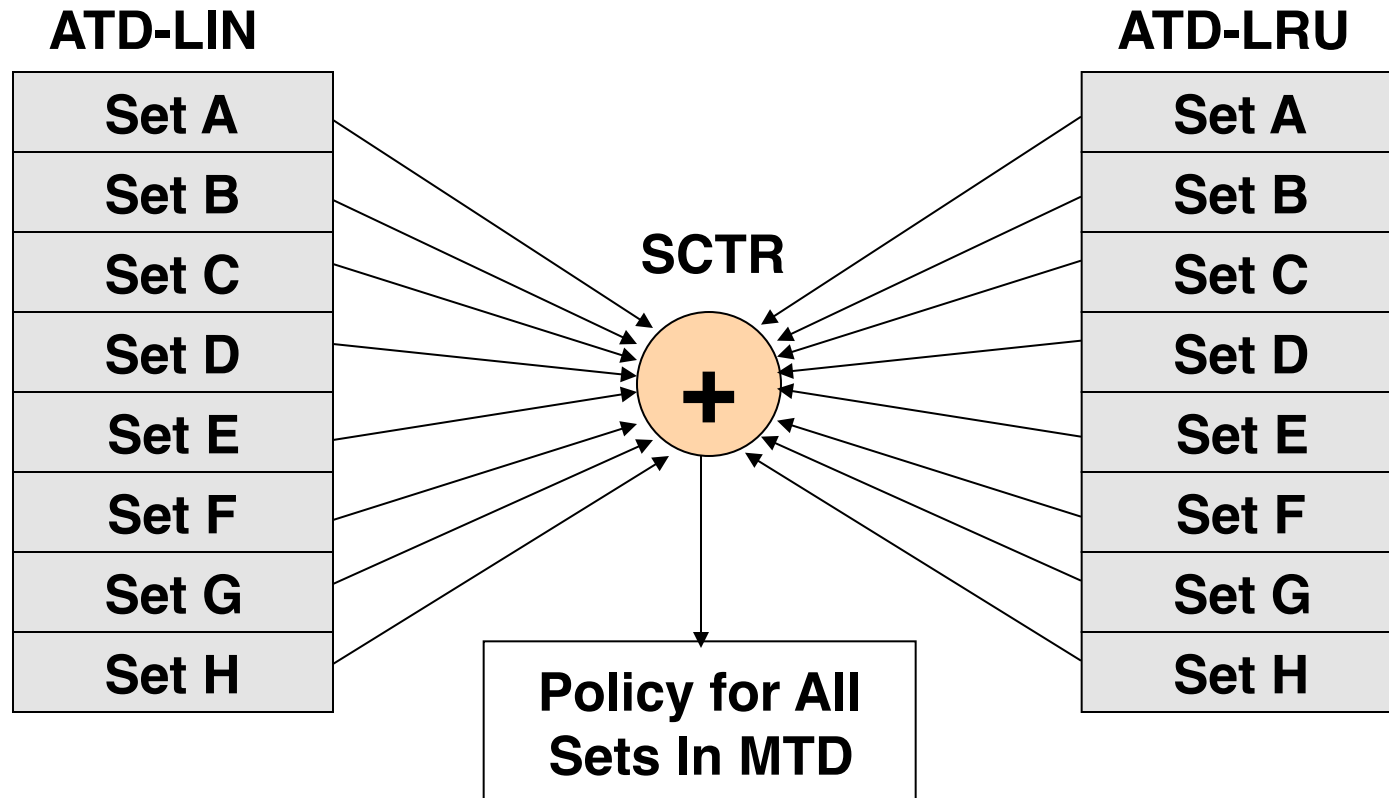


If MSB of SCTR is 1, MTD uses LIN else MTD use LRU

ATD-LIN	ATD-LRU	Saturating Counter (SCTR)
HIT	HIT	Unchanged
MISS	MISS	Unchanged
HIT	MISS	$+=$ Cost of Miss in ATD-LRU
MISS	HIT	$-=$ Cost of Miss in ATD-LIN

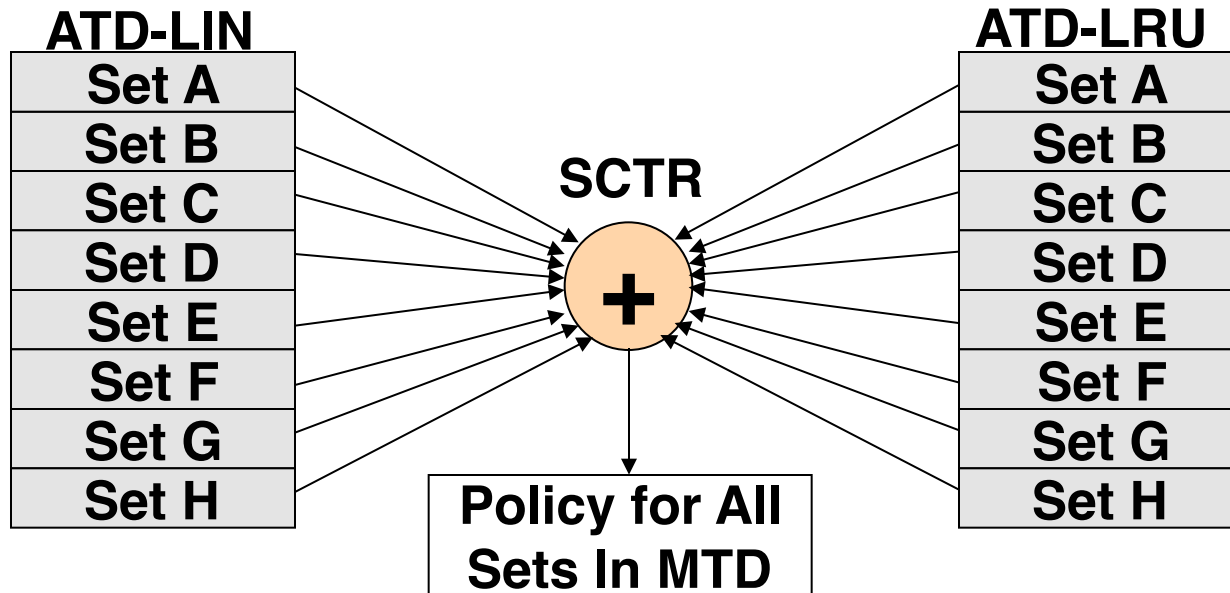
Extending TSEL to All Sets

Implementing TSEL on a per-set basis is expensive
Counter overhead can be reduced by using a global counter



Dynamic Set Sampling

Not all sets are required to decide the best policy
Have the ATD entries only for few sets.



Sets that have ATD entries (B, E, G) are called **leader sets**

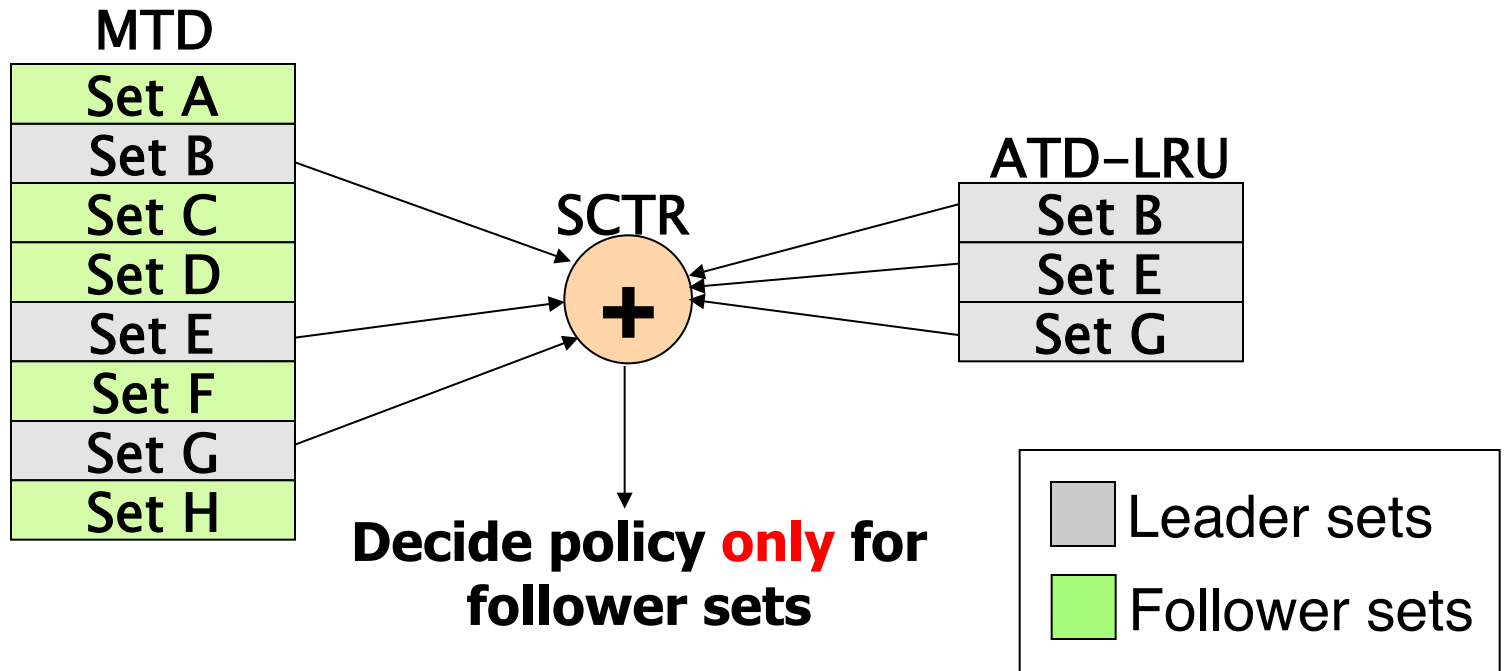
Dynamic Set Sampling

How many sets are required to choose best performing policy?

- ❑ Bounds using analytical model and simulation (in paper)
- ❑ DSS with **32 leader sets** performs similar to having all sets
- ❑ Last-level cache typically contains 1000s of sets, thus ATD entries are required for **only 2%-3%** of the sets

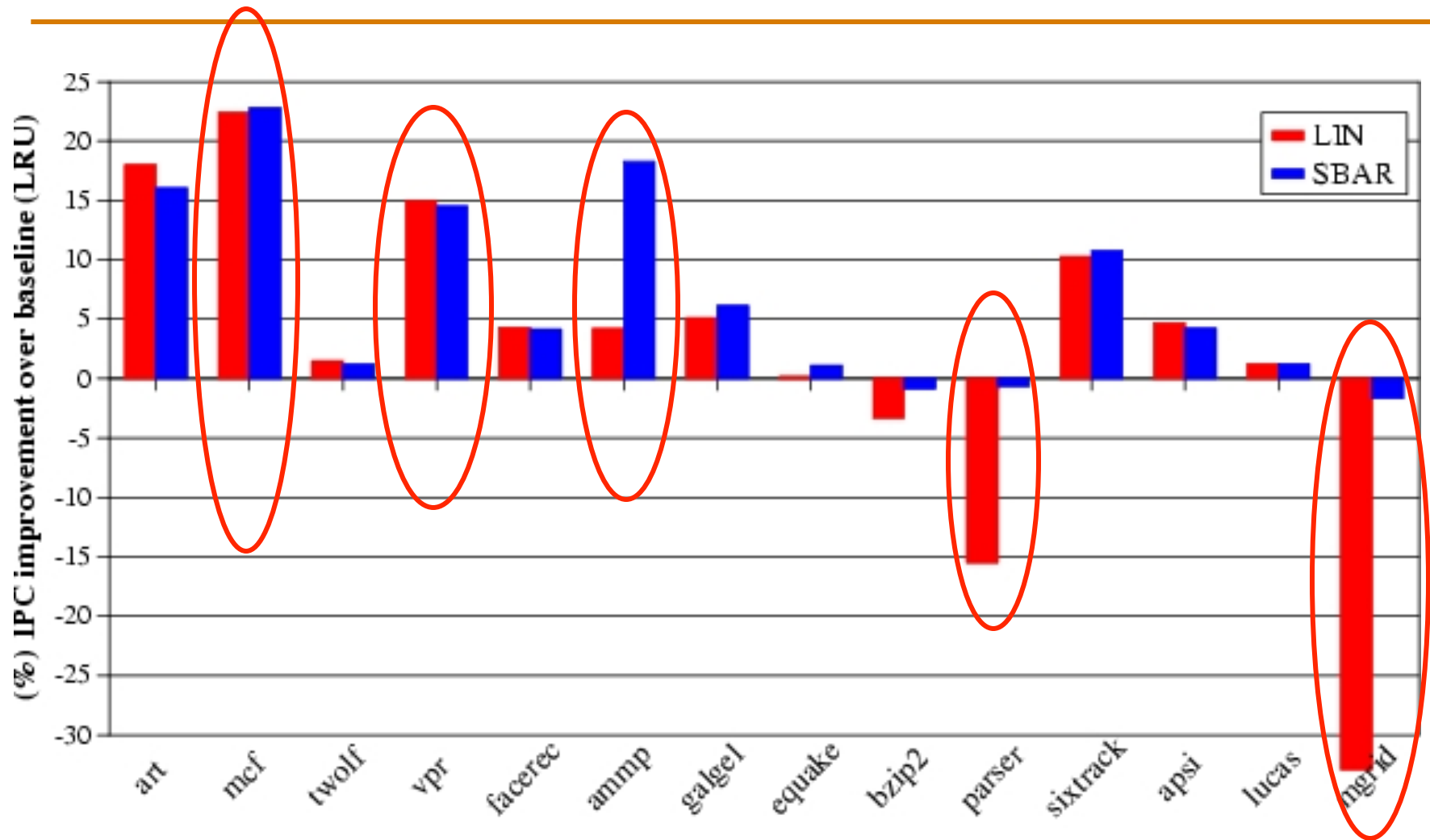
ATD overhead can further be reduced by using MTD to always simulate one of the policies (say LIN)

Sampling Based Adaptive Replacement (SBAR)

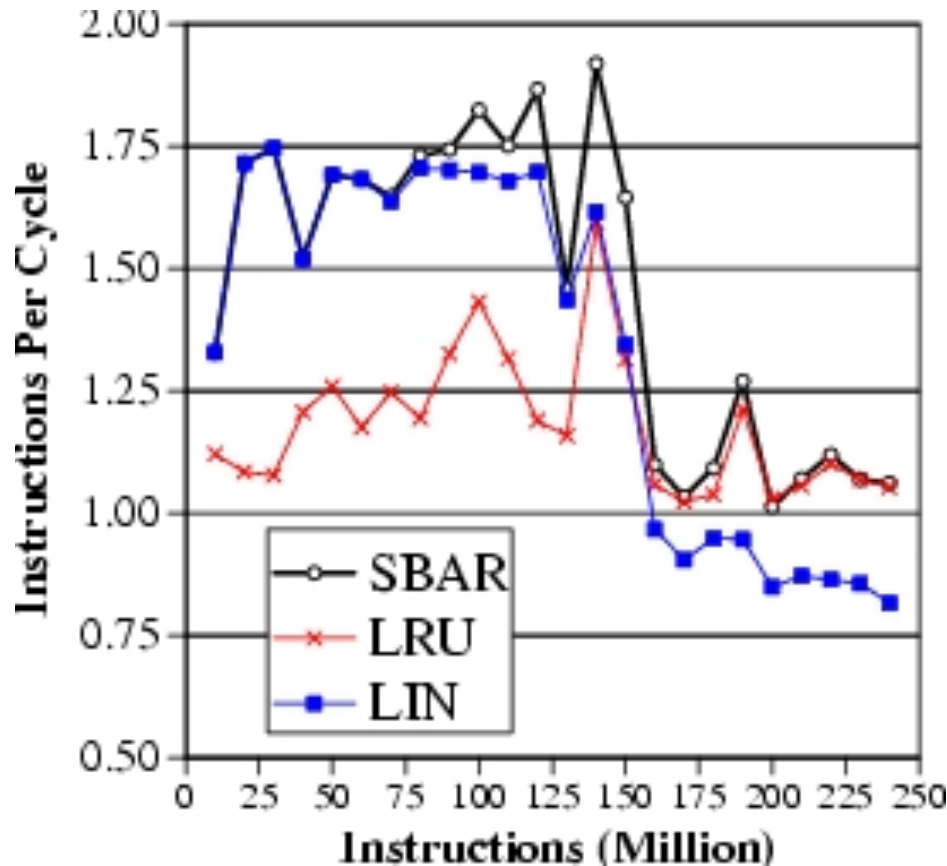


The storage overhead of SBAR is less than 2KB
(0.2% of the baseline 1MB cache)

Results for SBAR



SBAR adaptation to phases



SBAR selects the best policy for each phase of ammp

Outline

❑ Introduction

❑ MLP-Aware Cache Replacement

- Model for Computing Cost
- Repeatability of Cost
- A Cost-Sensitive Replacement Policy

❑ Practical Hybrid Replacement

- Tournament Selection
- Dynamic Set Sampling
- Sampling Based Adaptive Replacement

❑ Summary

Summary

- ❑ MLP varies. Some misses are more costly than others
- ❑ MLP-aware cache replacement can reduce costly misses
- ❑ Proposed a runtime mechanism to compute MLP-Based cost and the LIN policy for MLP-aware cache replacement
- ❑ SBAR allows dynamic selection between LIN and LRU with low hardware overhead
- ❑ Dynamic set sampling used in SBAR also enables other cache related optimizations

The Evicted-Address Filter

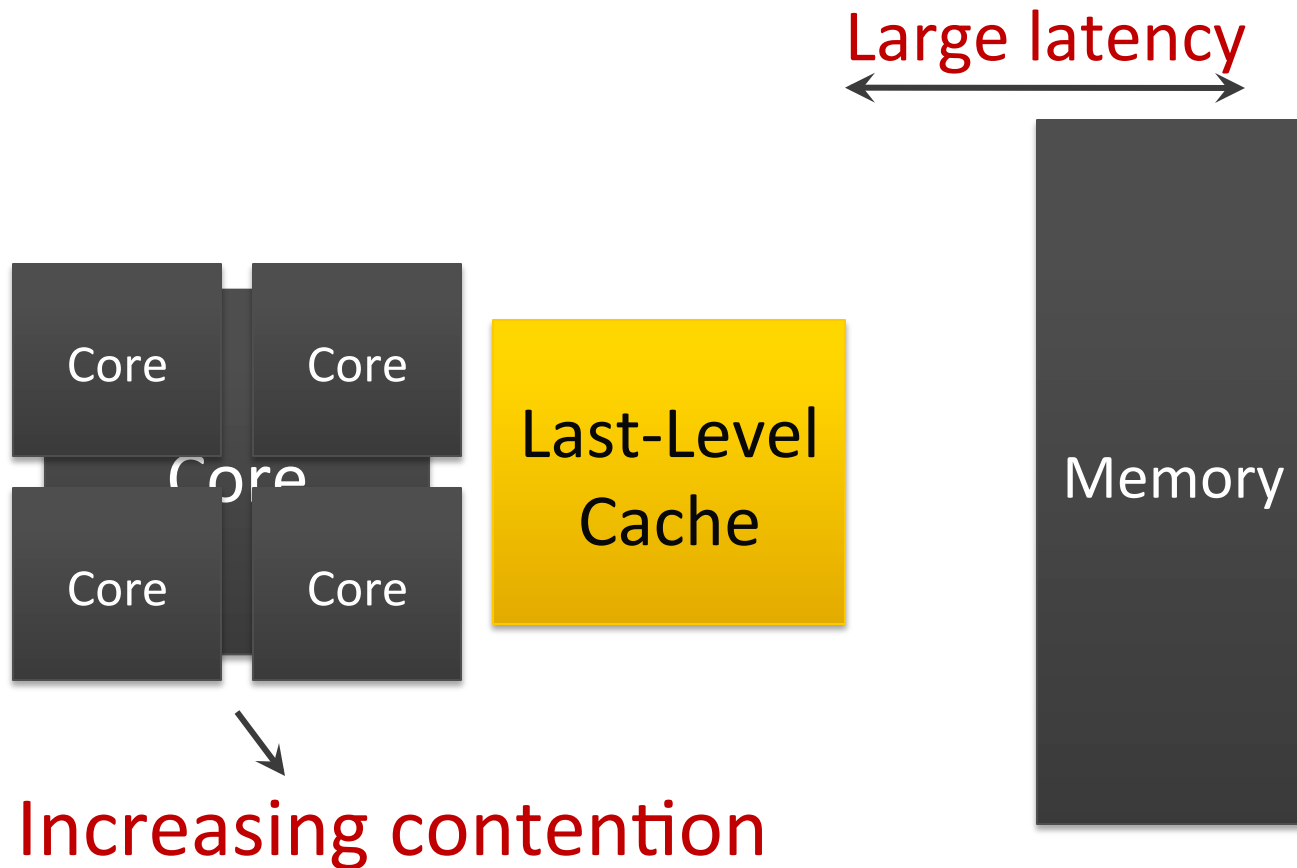
Vivek Seshadri, Onur Mutlu, Michael A. Kozuch, and Todd C. Mowry,
**"The Evicted-Address Filter: A Unified Mechanism to Address Both
Cache Pollution and Thrashing"**

*Proceedings of the
21st ACM International Conference on Parallel Architectures and Compilation
Techniques (PACT), Minneapolis, MN, September 2012. Slides (pptx)*

Executive Summary

- Two problems degrade cache performance
 - Pollution and thrashing
 - Prior works don't address both problems concurrently
- Goal: A mechanism to address both problems
- EAF-Cache
 - Keep track of recently evicted block addresses in EAF
 - Insert low reuse with low priority to mitigate pollution
 - Clear EAF periodically to mitigate thrashing
 - Low complexity implementation using Bloom filter
- EAF-Cache outperforms five prior approaches that address pollution or thrashing

Cache Utilization is Important

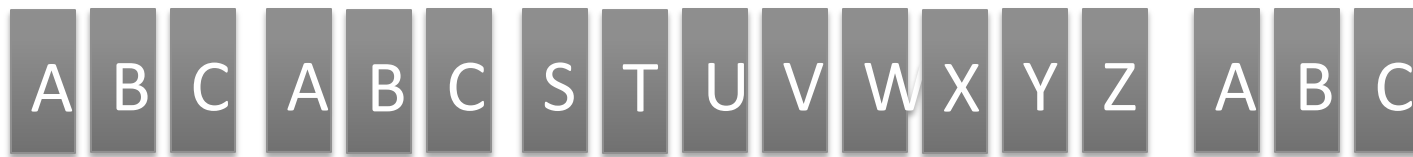


Effective cache utilization is important

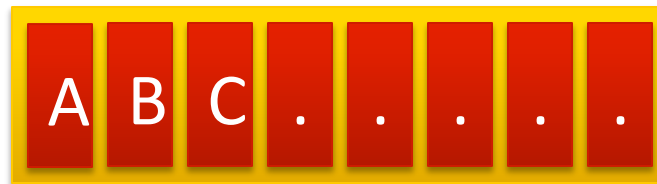
Reuse Behavior of Cache Blocks

Different blocks have different reuse behavior

Access Sequence:



Ideal Cache



Cache Pollution

Problem: Low-reuse blocks evict high-reuse blocks

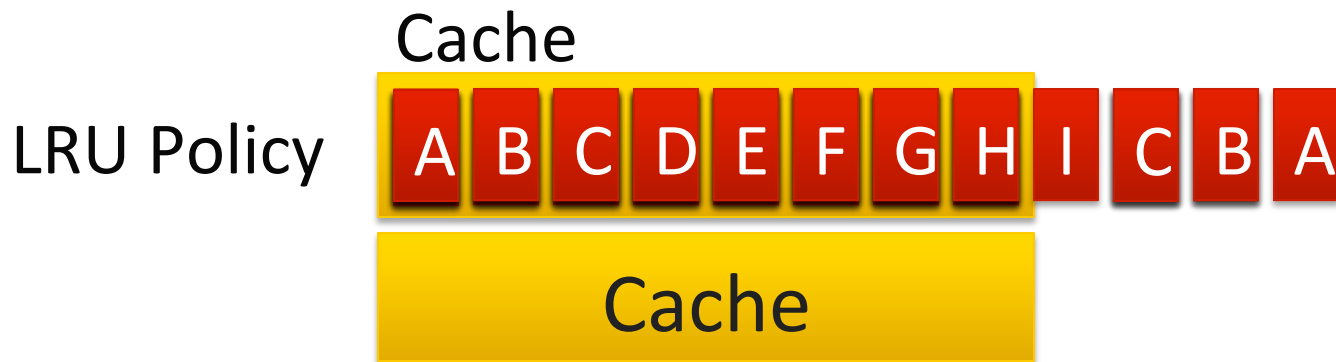


Prior work: Predict reuse behavior of missed blocks. Insert low-reuse blocks at LRU position.



Cache Thrashing

Problem: High-reuse blocks evict each other



Prior work: Insert at MRU position with a very low probability (**Bimodal insertion policy**)

A fraction of
working set
stays in cache



Shortcomings of Prior Works

Prior works do not address both pollution and thrashing concurrently

Prior Work on Cache Pollution

No control on the number of blocks inserted with high priority into the cache

Prior Work on Cache Thrashing

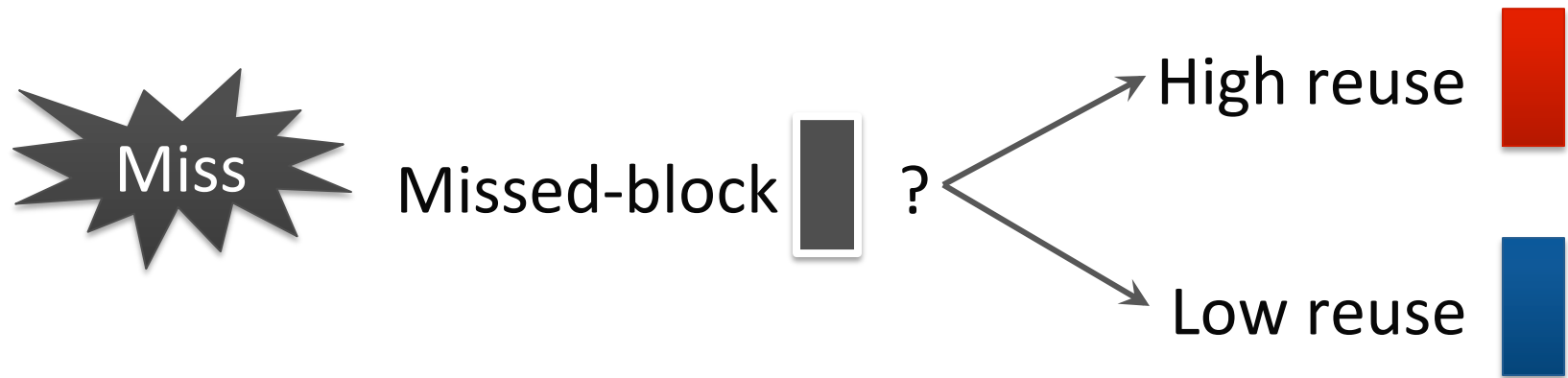
No mechanism to distinguish high-reuse blocks from low-reuse blocks

Our goal: Design a mechanism to address both pollution and thrashing concurrently

Outline

- Background and Motivation
- Evicted-Address Filter
 - Reuse Prediction
 - Thrash Resistance
- Final Design
- Advantages and Disadvantages
- Evaluation
- Conclusion

Reuse Prediction



Keep track of the reuse behavior of every cache block in the system

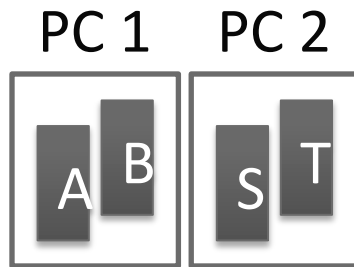
Impractical

1. High storage overhead
2. Look-up latency

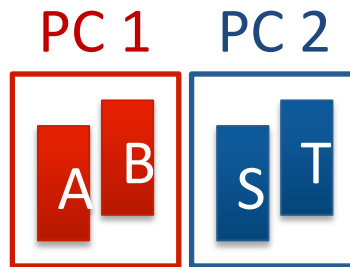
Prior Work on Reuse Prediction

Use program counter or memory region information.

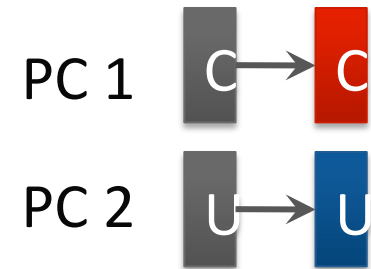
1. Group Blocks



2. Learn group behavior



3. Predict reuse

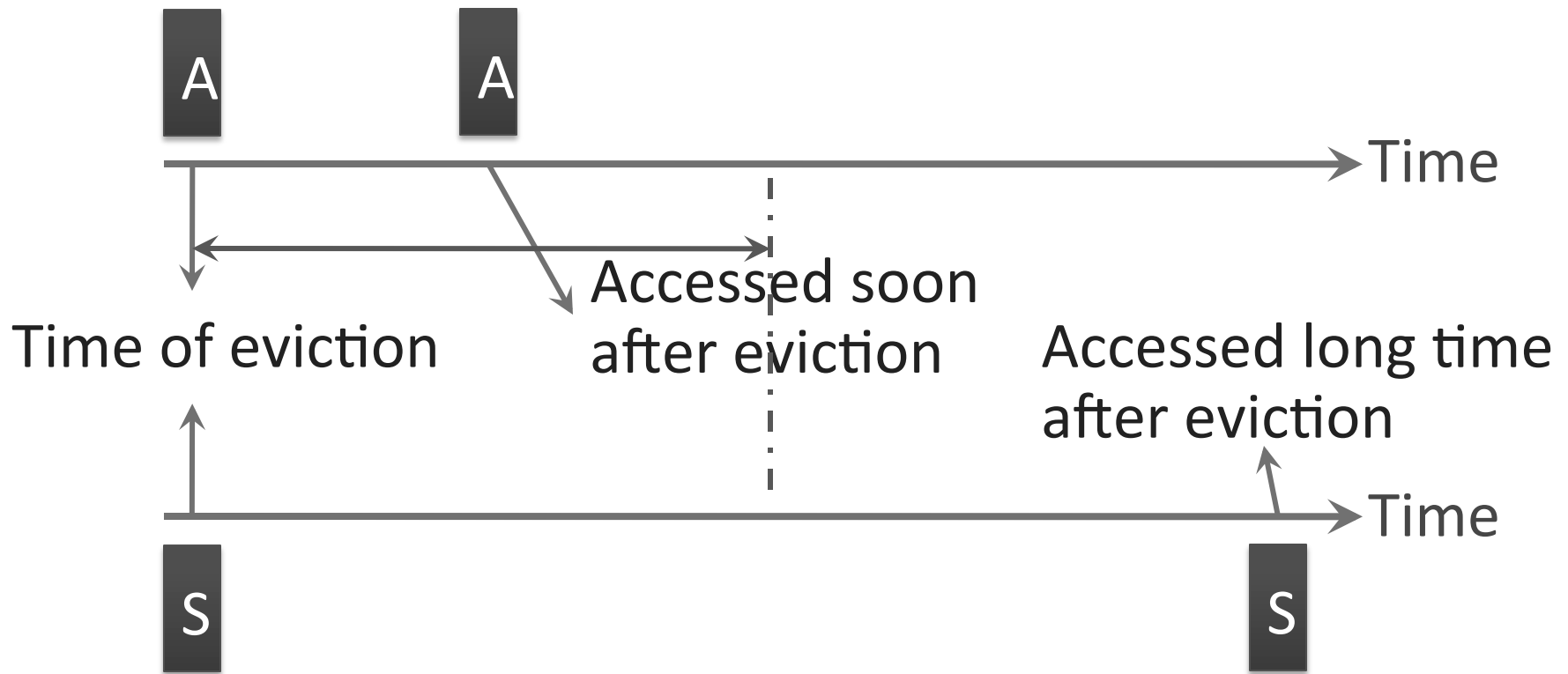


1. Same group \nrightarrow same reuse behavior
2. No control over number of high-reuse blocks

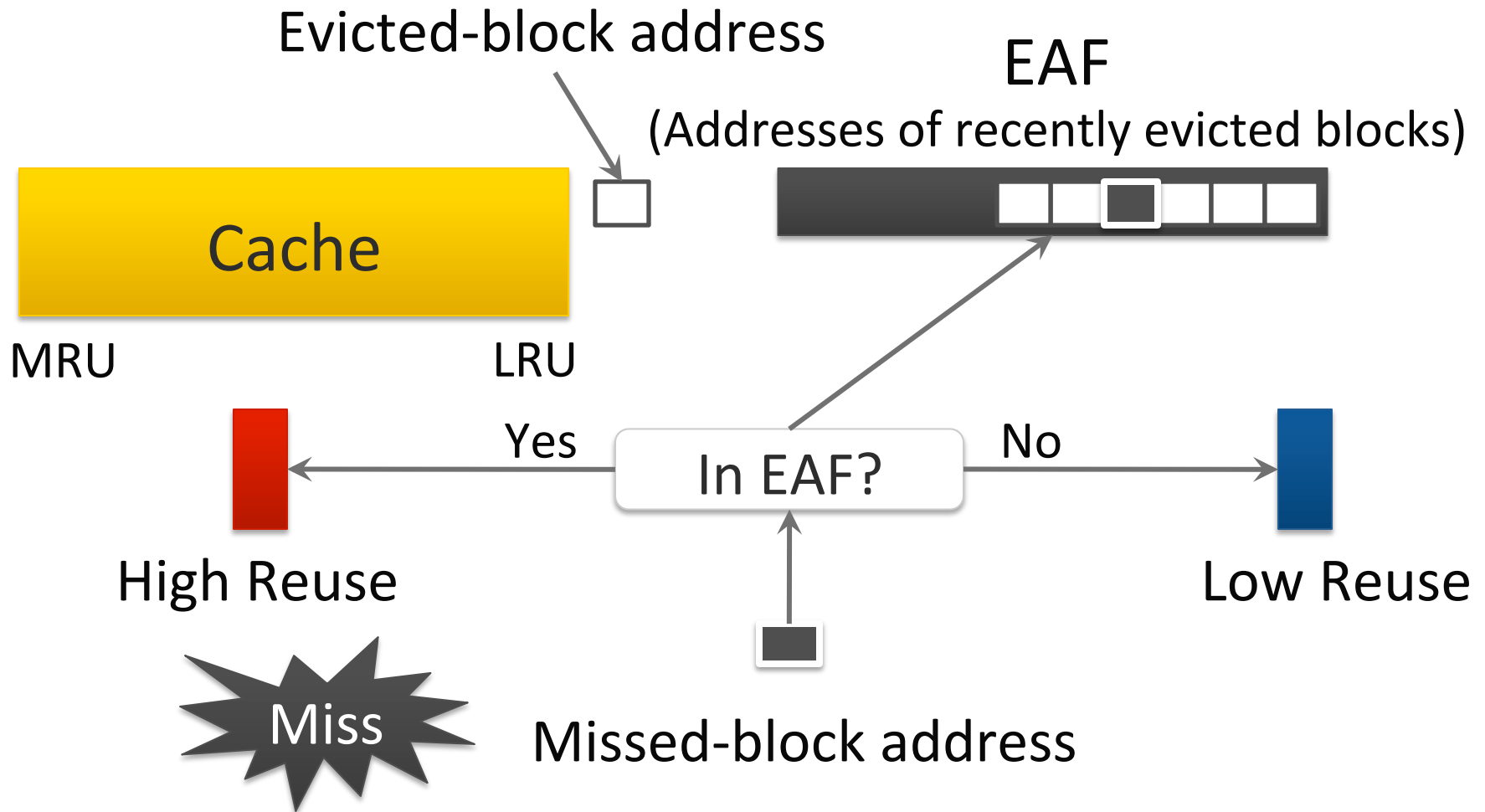
Our Approach: Per-block Prediction



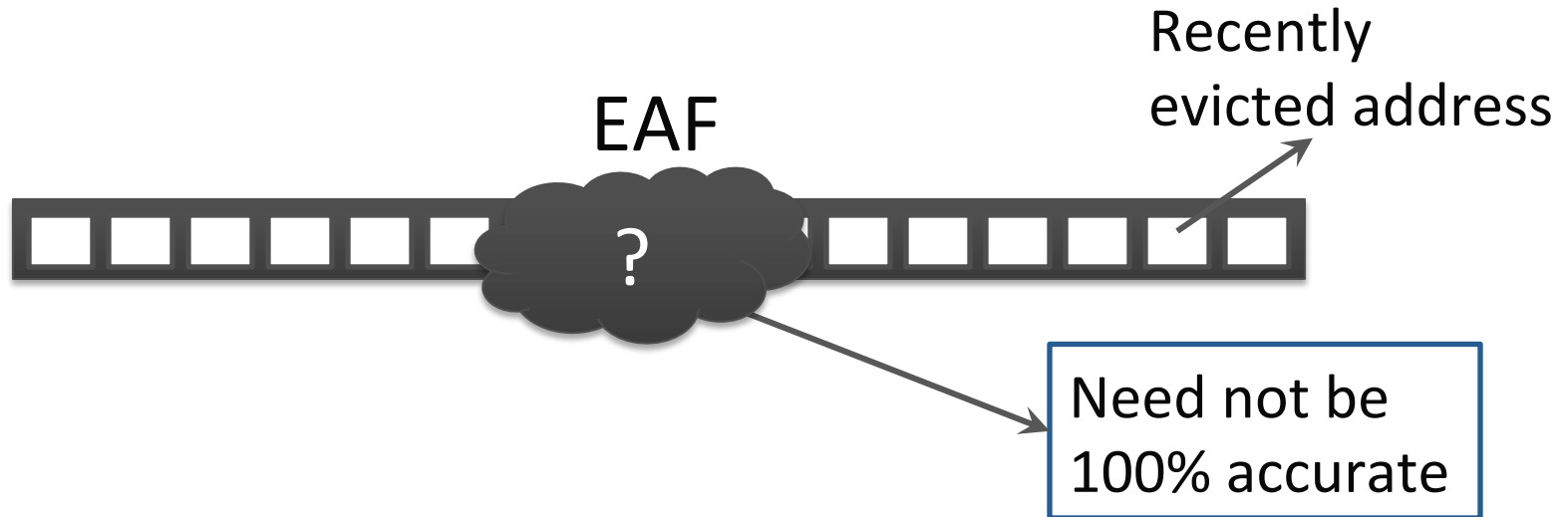
Use recency of eviction to predict reuse



Evicted-Address Filter (EAF)

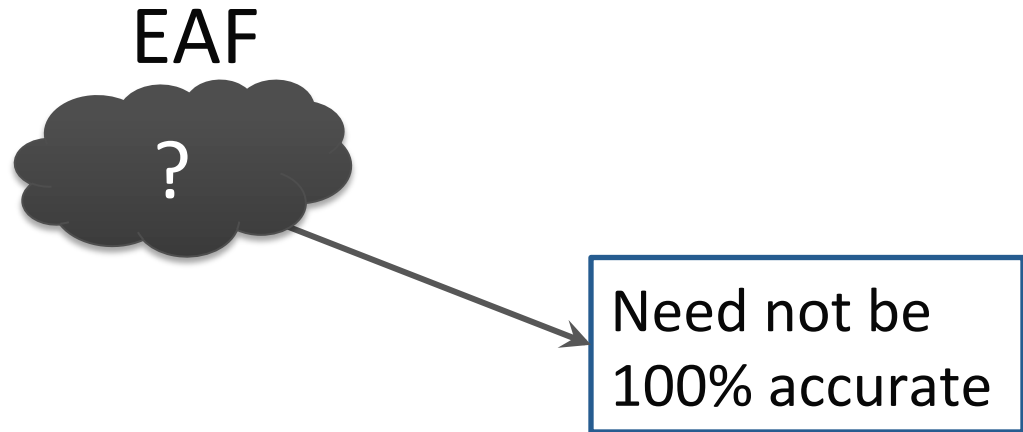


Naïve Implementation: Full Address Tags



1. Large storage overhead
2. Associative lookups – High energy

Low-Cost Implementation: Bloom Filter



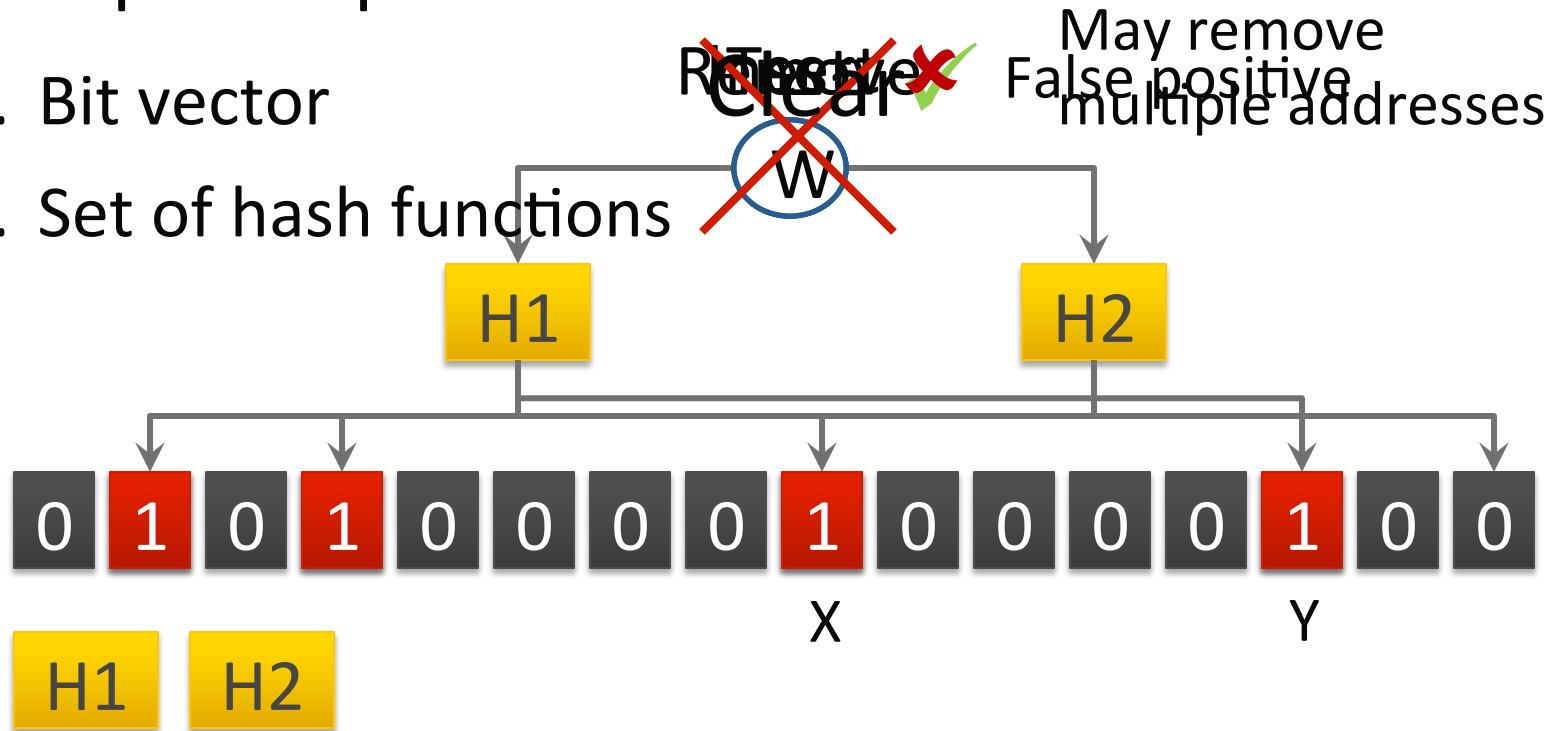
Implement EAF using a **Bloom Filter**
Low storage overhead + energy

Bloom Filter

Compact representation of a set

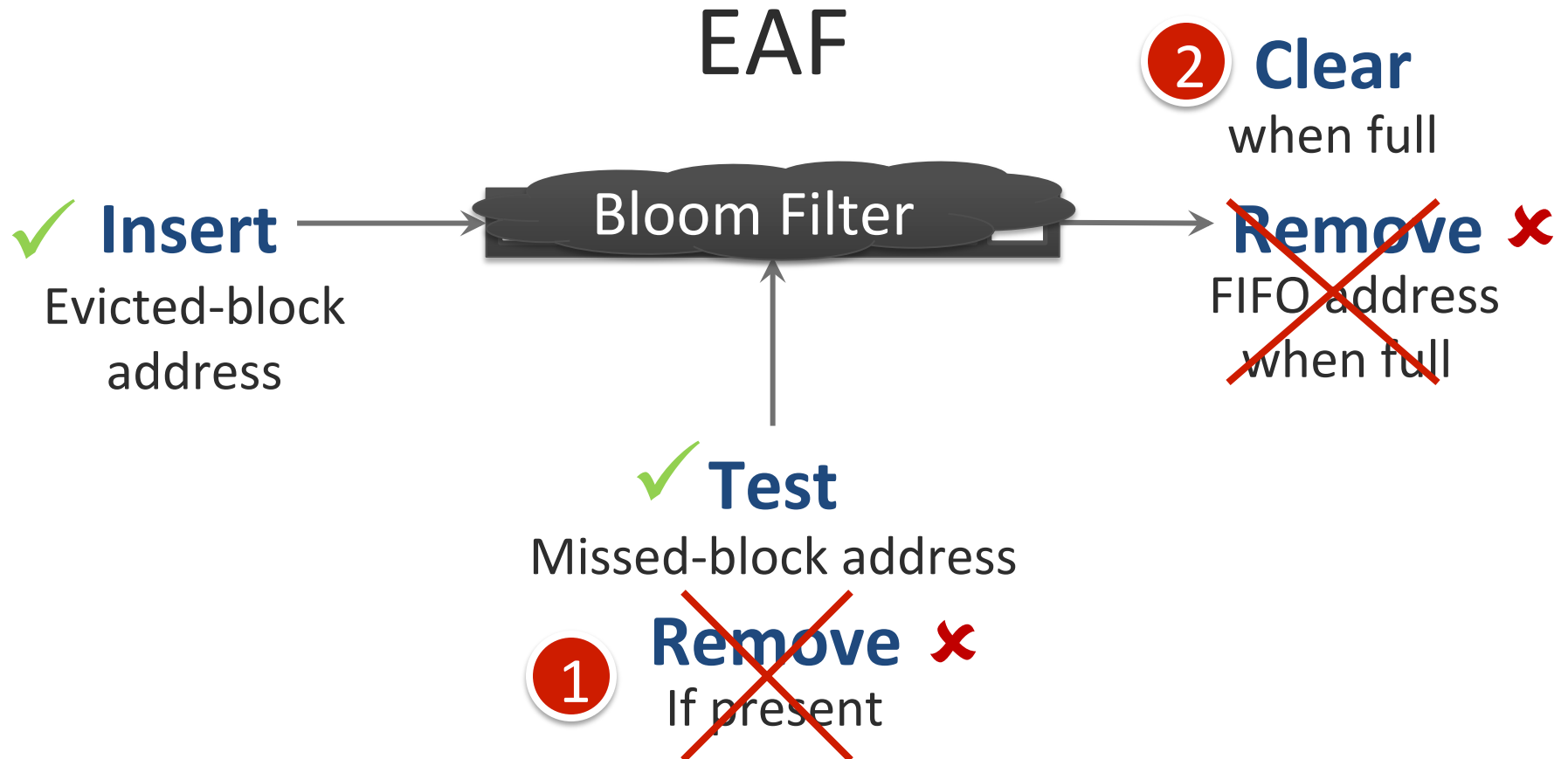
1. Bit vector

2. Set of hash functions



Inserted Elements: X Y

EAF using a Bloom Filter



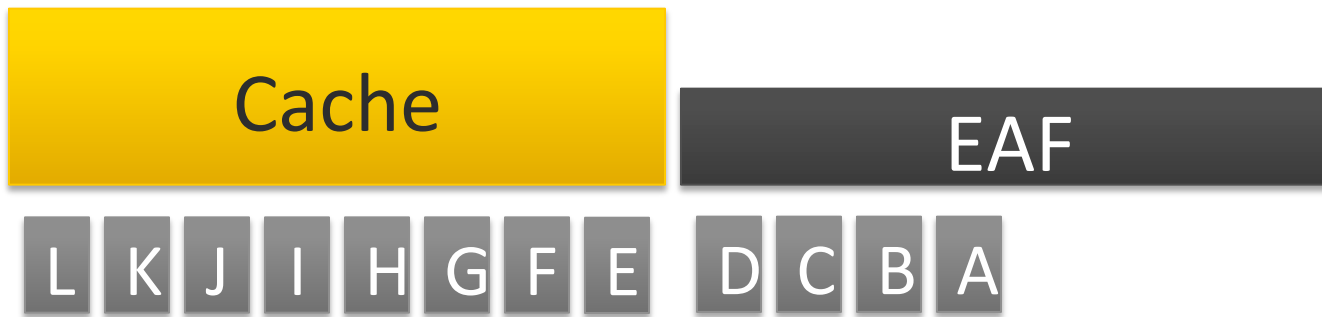
Bloom-filter EAF: 4x reduction in storage overhead,
1.47% compared to cache size

Outline

- Background and Motivation
- Evicted-Address Filter
 - Reuse Prediction
 - Thrash Resistance
- Final Design
- Advantages and Disadvantages
- Evaluation
- Conclusion

Large Working Set: 2 Cases

1 Cache < Working set < Cache + EAF



2 Cache + EAF < Working Set



Large Working Set: Case 1

Cache < Working set < Cache + EAF

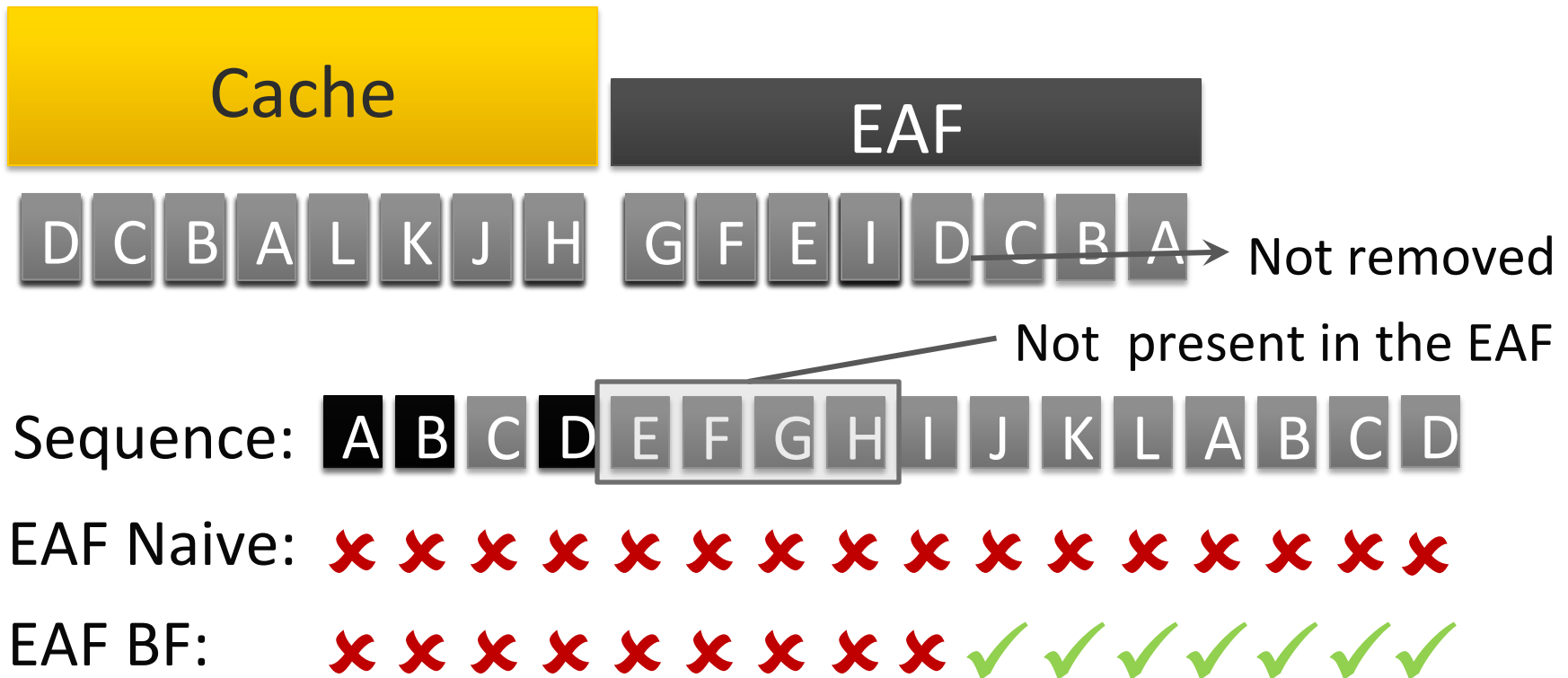


Sequence: **A** **B** **C** D E F G H I J K L A B C D

EAF Naive: **x** **x** **x** **x** **x** **x** **x** **x** **x** **x** **x** **x** **x** **x** **x** **x**

Large Working Set: Case 1

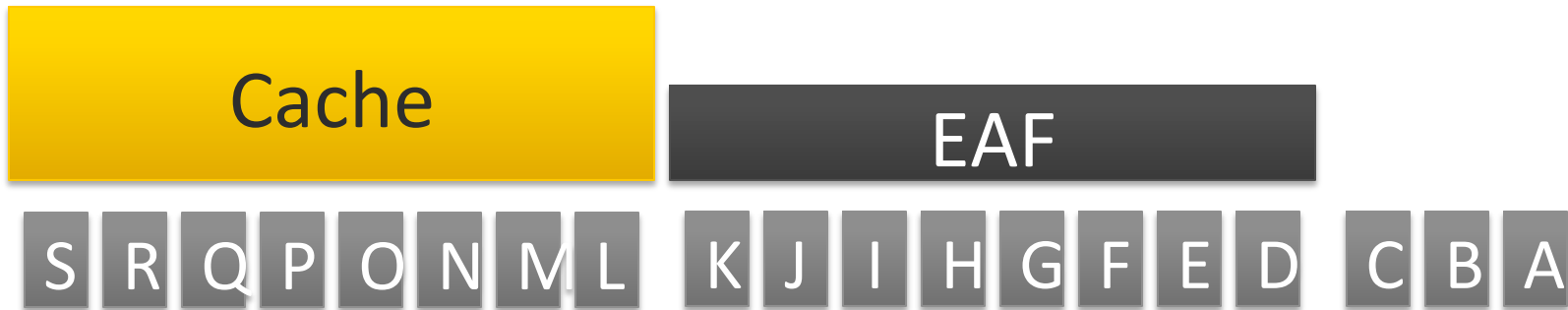
Cache < Working set < Cache + EAF



Bloom-filter based EAF mitigates thrashing

Large Working Set: Case 2

Cache + EAF < Working Set



Problem: All blocks are predicted to have low reuse

Allow a fraction of the working set to stay in the cache



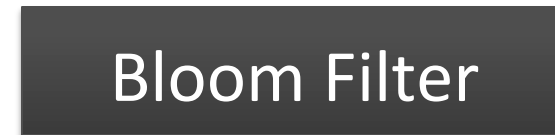
Use **Bimodal Insertion Policy** for low reuse blocks. Insert few of them at the MRU position

Outline

- Background and Motivation
- Evicted-Address Filter
 - Reuse Prediction
 - Thrash Resistance
- Final Design
- Advantages and Disadvantages
- Evaluation
- Conclusion

EAF-Cache: Final Design

- 1 Cache eviction**
Insert address into filter
Increment counter

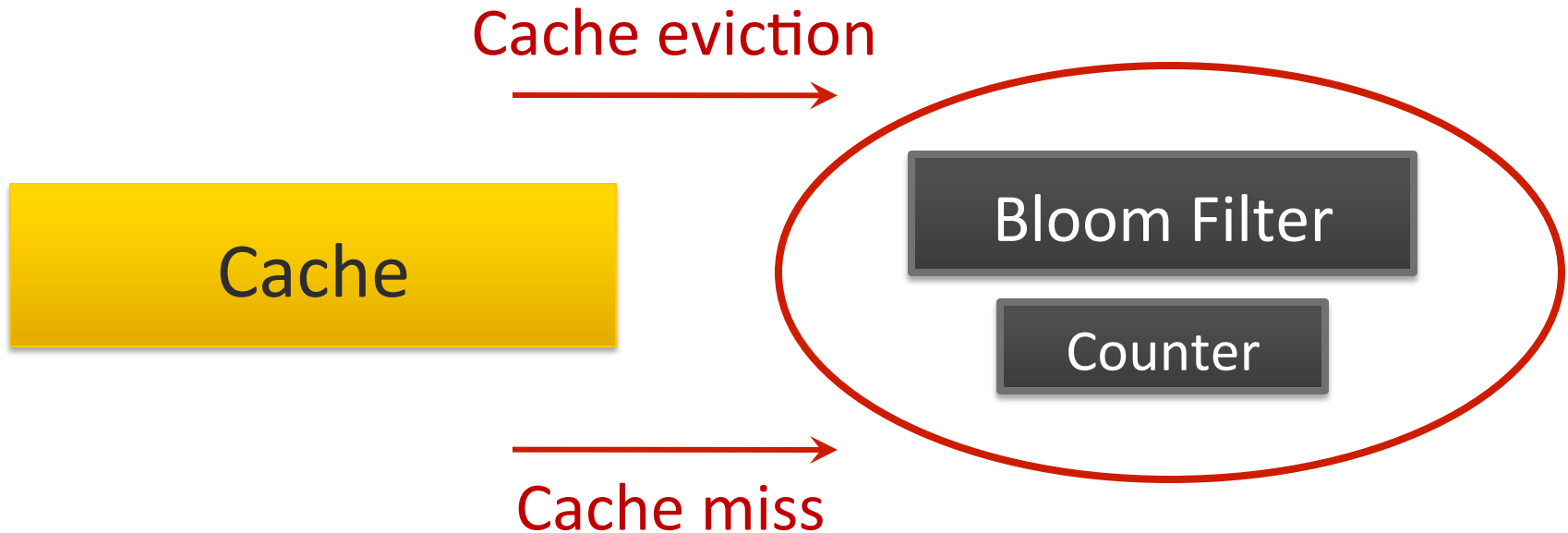


- 3 Counter reaches max**
Clear filter and counter
- 2 Cache miss**
Test if address is present in filter
Yes, insert at MRU. No, insert with BIP

Outline

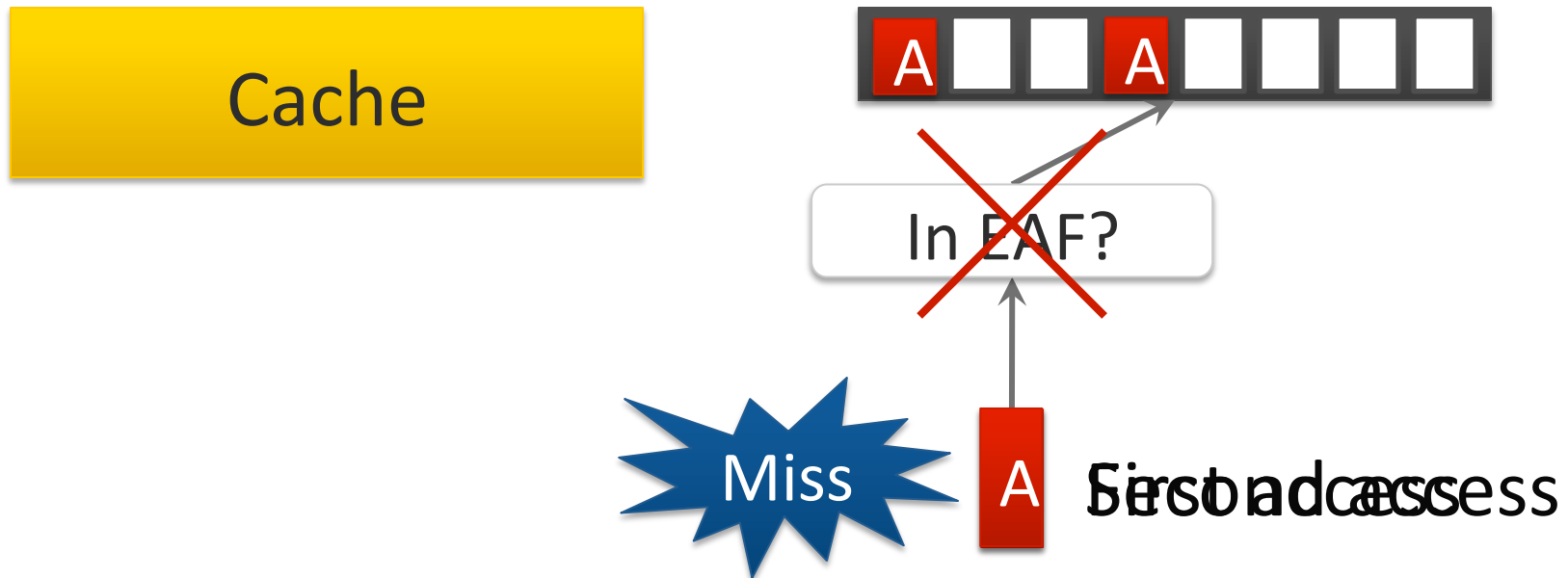
- Background and Motivation
- Evicted-Address Filter
 - Reuse Prediction
 - Thrash Resistance
- Final Design
- Advantages and Disadvantages
- Evaluation
- Conclusion

EAF: Advantages



1. Simple to implement
2. Easy to design and verify
3. Works with other techniques (replacement policy)

EAF: Disadvantage



Problem: For an **LRU-friendly application**, EAF incurs one **additional** miss for most blocks



Dueling-EAF: set dueling between EAF and LRU

Outline

- Background and Motivation
- Evicted-Address Filter
 - Reuse Prediction
 - Thrash Resistance
- Final Design
- Advantages and Disadvantages
- Evaluation
- Conclusion

Methodology

- **Simulated System**
 - In-order cores, single issue, 4 GHz
 - 32 KB L1 cache, 256 KB L2 cache (private)
 - Shared L3 cache (1MB to 16MB)
 - Memory: 150 cycle row hit, 400 cycle row conflict
- **Benchmarks**
 - SPEC 2000, SPEC 2006, TPC-C, 3 TPC-H, Apache
- **Multi-programmed workloads**
 - Varying memory intensity and cache sensitivity
- **Metrics**
 - 4 different metrics for performance and fairness
 - Present weighted speedup

Comparison with Prior Works

Addressing Cache Pollution

Run-time Bypassing (RTB) – Johnson+ ISCA'97

- Memory region based reuse prediction

Single-usage Block Prediction (SU) – Piquet+ ACSAC'07

Signature-based Hit Prediction (SHIP) – Wu+ MICRO'11

- Program counter based reuse prediction

Miss Classification Table (MCT) – Collins+ MICRO'99

- One most recently evicted block
- No control on number of blocks inserted with high priority \Rightarrow Thrashing

Comparison with Prior Works

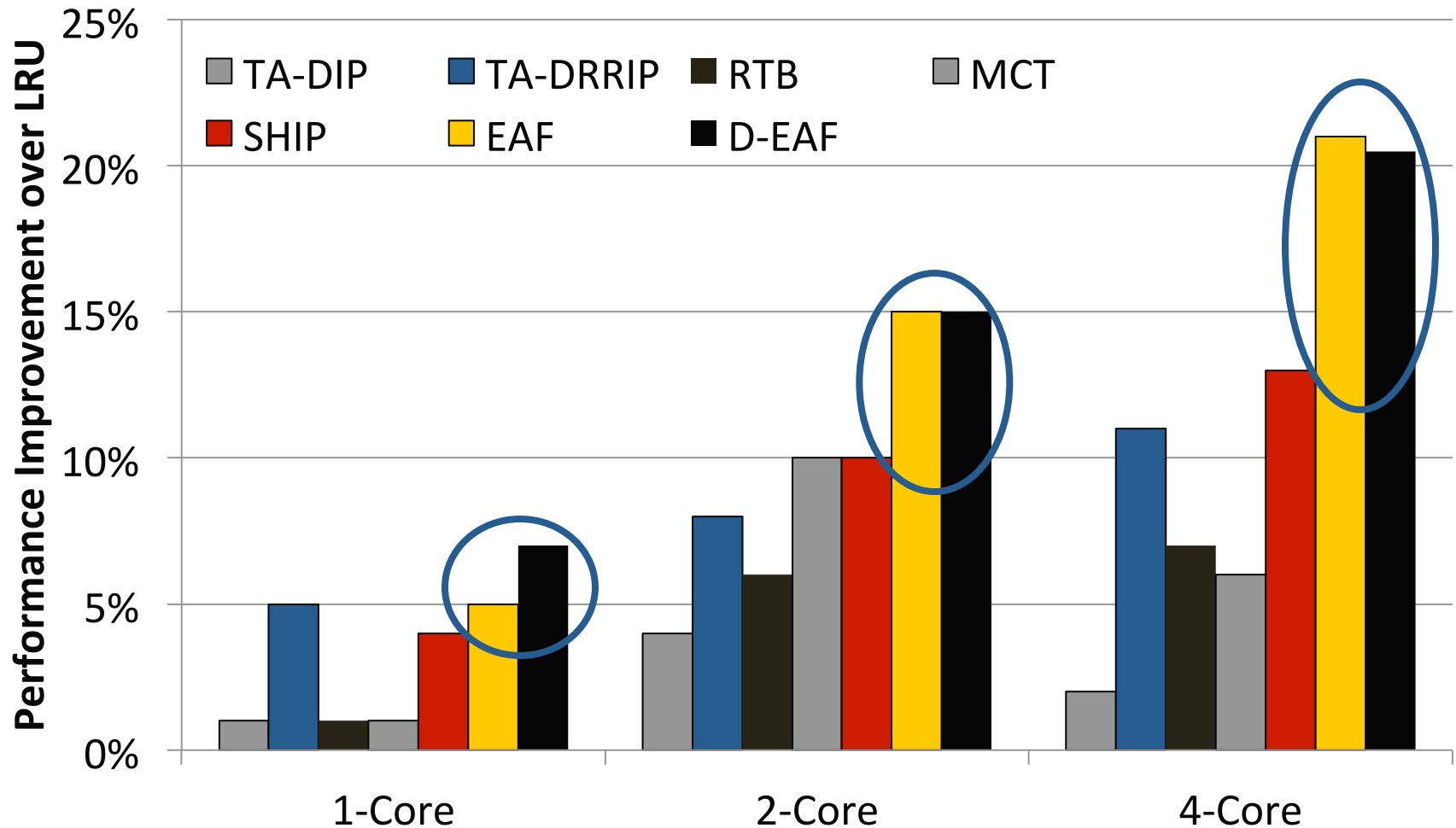
Addressing Cache Thrashing

TA-DIP – Qureshi+ ISCA'07, Jaleel+ PACT'08

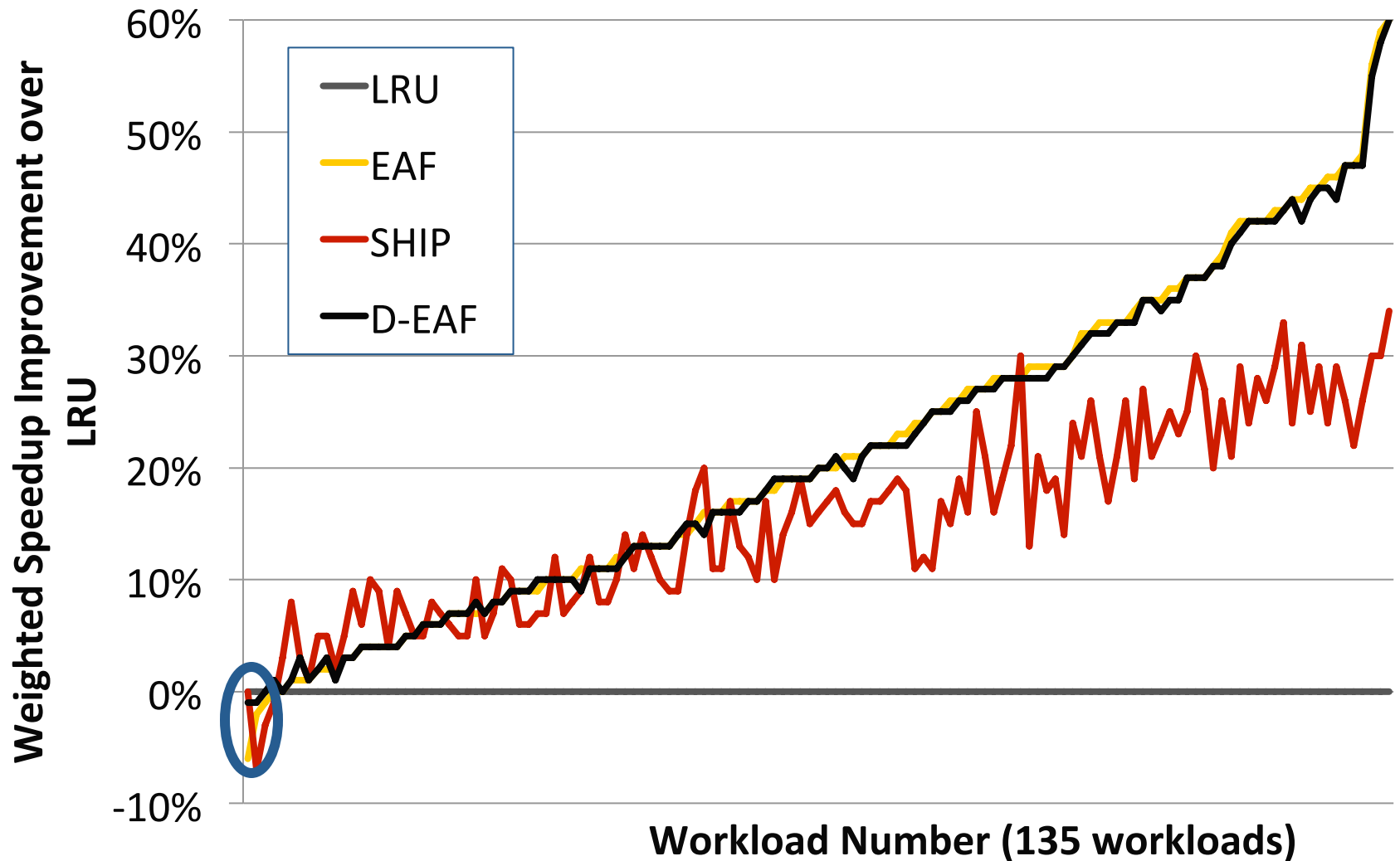
TA-DRRIP – Jaleel+ ISCA'10

- Use set dueling to determine thrashing applications
- No mechanism to filter low-reuse blocks \Rightarrow Pollution

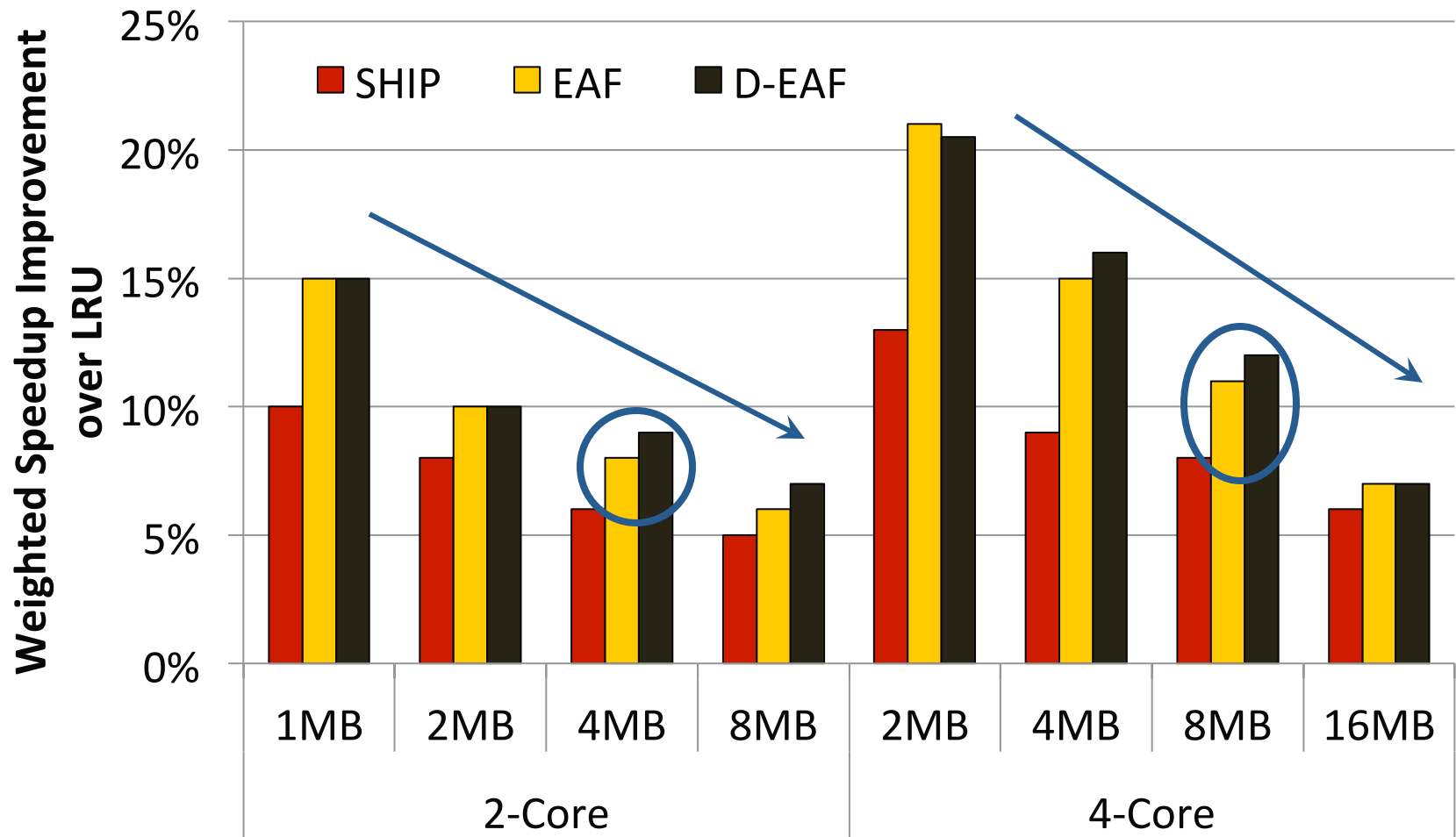
Results – Summary



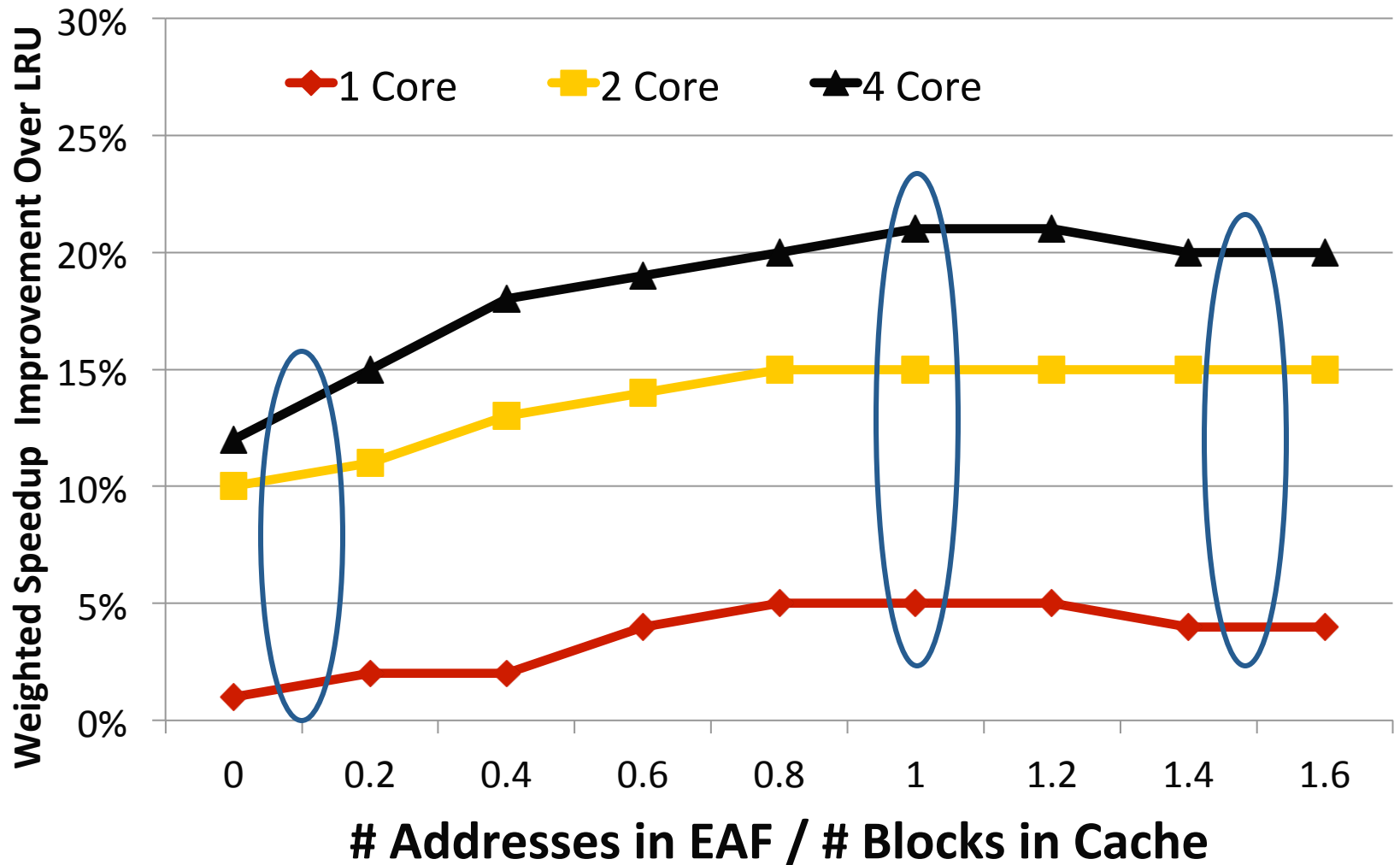
4-Core: Performance



Effect of Cache Size



Effect of EAF Size



Other Results in Paper

- EAF orthogonal to replacement policies
 - LRU, RRIP – Jaleel+ ISCA'10
- Performance improvement of EAF increases with increasing memory latency
- EAF performs well on four different metrics
 - Performance and fairness
- Alternative EAF-based designs perform comparably
 - Segmented EAF
 - Decoupled-clear EAF

Conclusion

- Cache utilization is critical for system performance
 - Pollution and thrashing degrade cache performance
 - Prior works don't address both problems concurrently
- EAF-Cache
 - Keep track of recently evicted block addresses in EAF
 - Insert low reuse with low priority to mitigate pollution
 - Clear EAF periodically and use BIP to mitigate thrashing
 - Low complexity implementation using Bloom filter
- EAF-Cache outperforms five prior approaches that address pollution or thrashing

Base-Delta-Immediate Cache Compression

Gennady Pekhimenko, Vivek Seshadri, Onur Mutlu, Philip B. Gibbons, Michael A. Kozuch, and Todd C. Mowry,

**"Base-Delta-Immediate Compression: Practical Data Compression
for On-Chip Caches"**

*Proceedings of the
21st ACM International Conference on Parallel Architectures and Compilation
Techniques (PACT), Minneapolis, MN, September 2012. Slides (pptx)*

Executive Summary

- Off-chip memory latency is high
 - Large caches can help, **but** at significant cost
- Compressing data in cache enables larger cache at low cost
- **Problem**: Decompression is on the execution critical path
- **Goal**: Design a new compression scheme that has
 1. low decompression latency, 2. low cost, 3. high compression ratio
- **Observation**: Many cache lines have low dynamic range data
- **Key Idea**: Encode cachelines as a base + multiple differences
- **Solution**: Base-Delta-Immediate compression with low decompression latency and high compression ratio
 - Outperforms three state-of-the-art compression mechanisms

Motivation for Cache Compression

Significant redundancy in data:

0x00000000	0x0000000B	0x00000003	0x00000004	...
------------	------------	------------	------------	-----

How can we exploit this redundancy?

- **Cache compression** helps
- Provides effect of a larger cache without making it physically larger

Background on Cache Compression



- Key requirements:
 - **Fast** (low decompression latency)
 - **Simple** (avoid complex hardware changes)
 - **Effective** (good compression ratio)

Shortcomings of Prior Work

Compression Mechanisms	Decompression Latency	Complexity	Compression Ratio
Zero	✓	✓	✗

Shortcomings of Prior Work

Compression Mechanisms	Decompression Latency	Complexity	Compression Ratio
Zero	✓	✓	✗
Frequent Value	✗	✗	✓

Shortcomings of Prior Work

Compression Mechanisms	Decompression Latency	Complexity	Compression Ratio
Zero	✓	✓	✗
Frequent Value	✗	✗	✓
Frequent Pattern	✗	✗ / ✓	✓

Shortcomings of Prior Work

Compression Mechanisms	Decompression Latency	Complexity	Compression Ratio
Zero	✓	✓	✗
Frequent Value	✗	✗	✓
Frequent Pattern	✗	✗ / ✓	✓
Our proposal: BΔI	✓	✓	✓

Outline

- Motivation & Background
- Key Idea & Our Mechanism
- Evaluation
- Conclusion

Key Data Patterns in Real Applications

Zero Values: initialization, sparse matrices, NULL pointers

0x00000000	0x00000000	0x00000000	0x00000000	...
------------	------------	------------	------------	-----

Repeated Values: common initial values, adjacent pixels

0x000000FF	0x000000FF	0x000000FF	0x000000FF	...
------------	------------	------------	------------	-----

Narrow Values: small values stored in a big data type

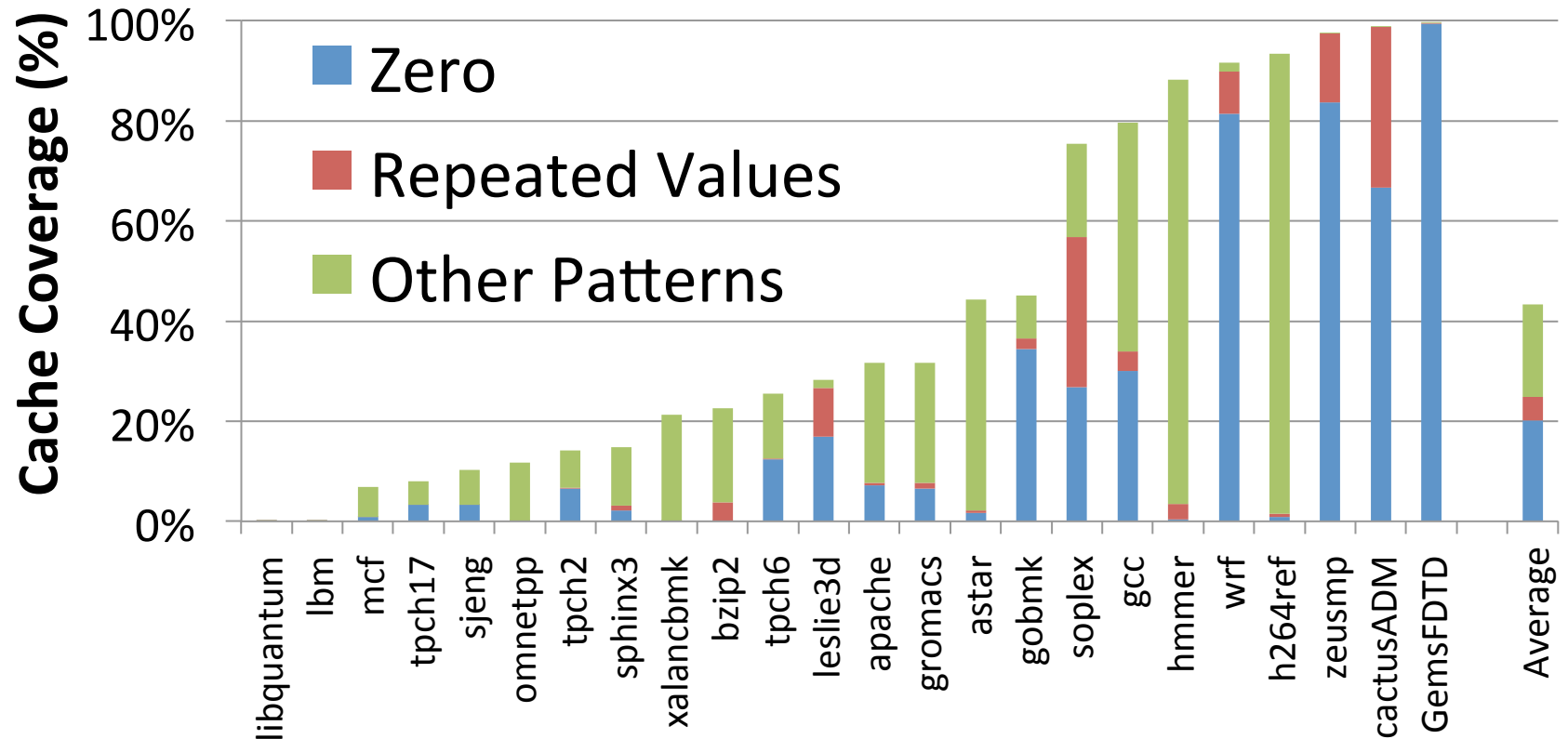
0x00000000	0x0000000B	0x00000003	0x00000004	...
------------	------------	------------	------------	-----

Other Patterns: pointers to the same memory region

0xC04039C0	0xC04039C8	0xC04039D0	0xC04039D8	...
------------	------------	------------	------------	-----

How Common Are These Patterns?

SPEC2006, databases, web workloads, 2MB L2 cache
“Other Patterns” include Narrow Values



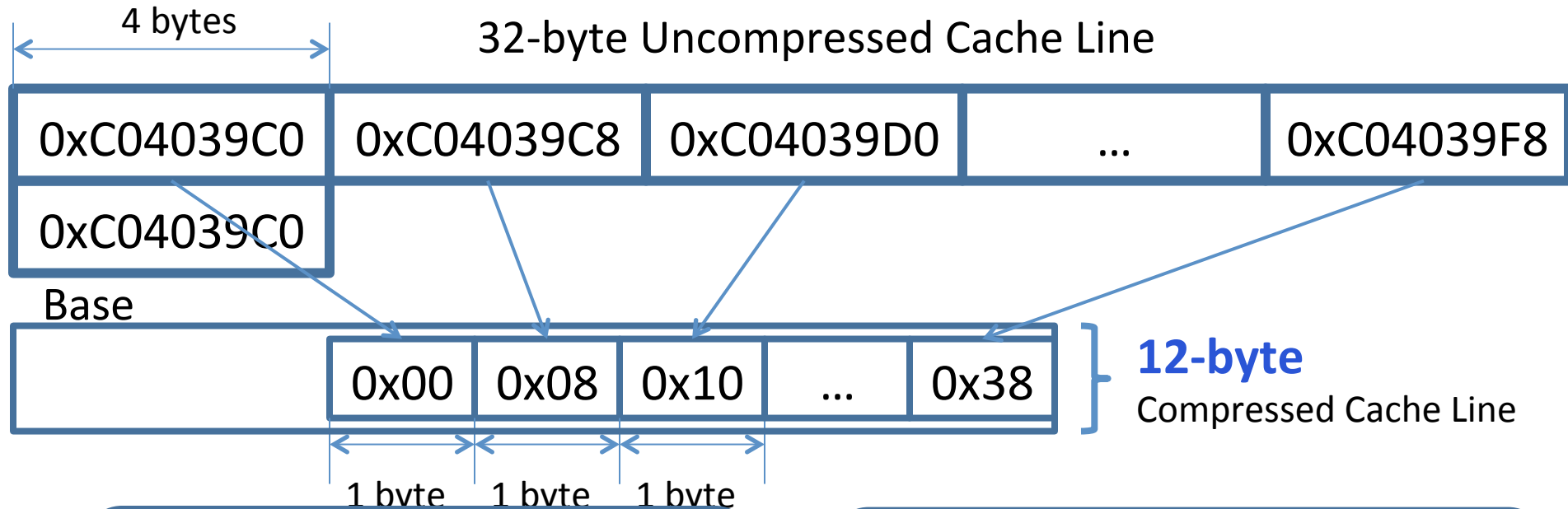
43% of the cache lines belong to key patterns

Key Data Patterns in Real Applications

Low Dynamic Range:

Differences between values are significantly smaller than the values themselves

Key Idea: Base+Delta (B+ Δ) Encoding



✓ **Fast Decompression:**
vector addition

✓ **Simple Hardware:**
arithmetic and comparison

✓ **Effective:** good compression ratio

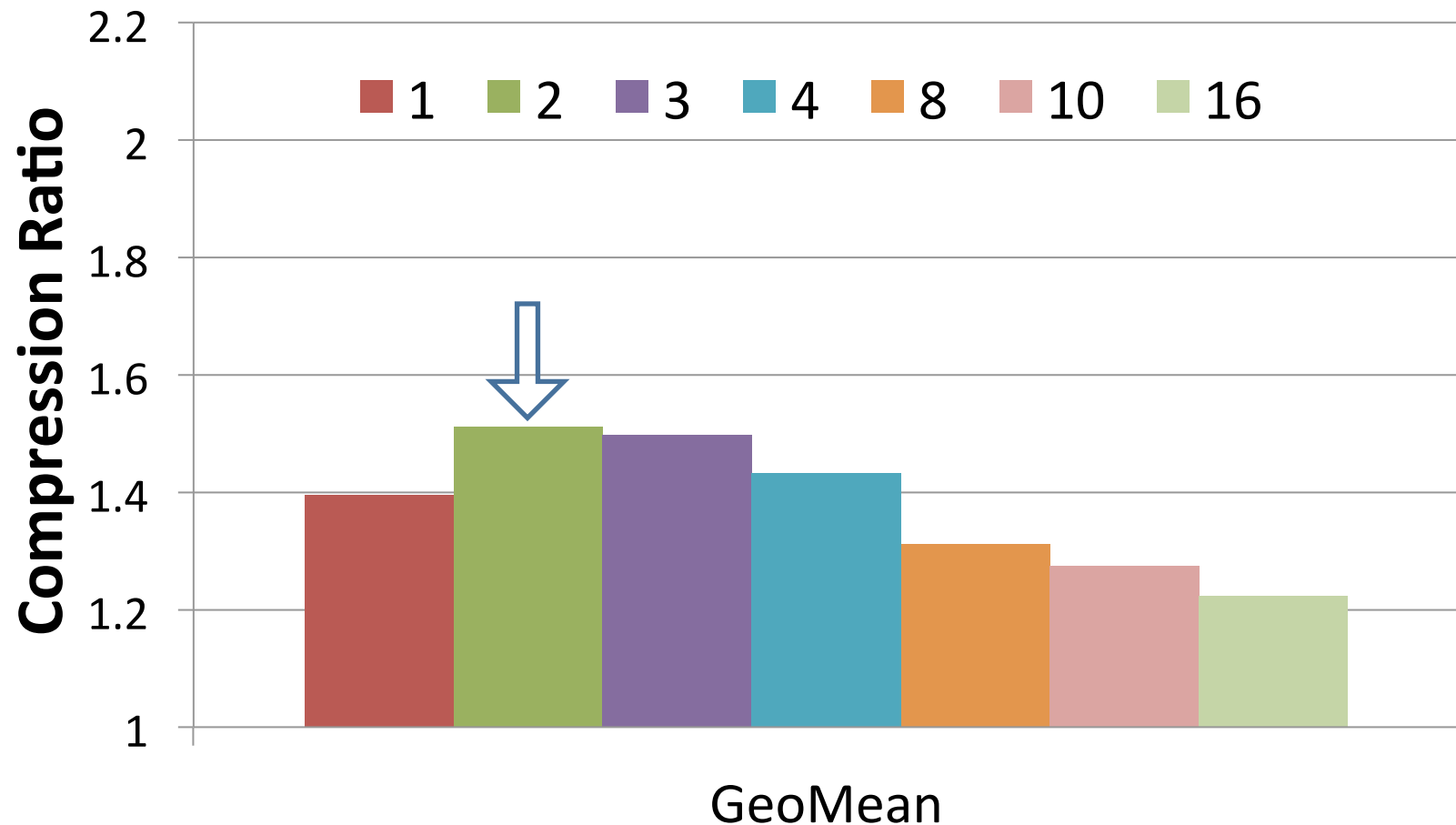
Can We Do Better?

- Uncompressible cache line (with a single base):

0x00000000	0x09A40178	0x0000000B	0x09A4A838	...
------------	------------	------------	------------	-----

- **Key idea:**
Use more bases, e.g., two instead of one
- **Pro:**
 - More cache lines can be compressed
- **Cons:**
 - Unclear how to find these bases efficiently
 - Higher overhead (due to additional bases)

B+ Δ with Multiple Arbitrary Bases



✓ **2 bases** – the best option based on evaluations

How to Find Two Bases Efficiently?

1. First base - first element in the cache line

✓ Base+Delta part

2. Second base - implicit base of 0

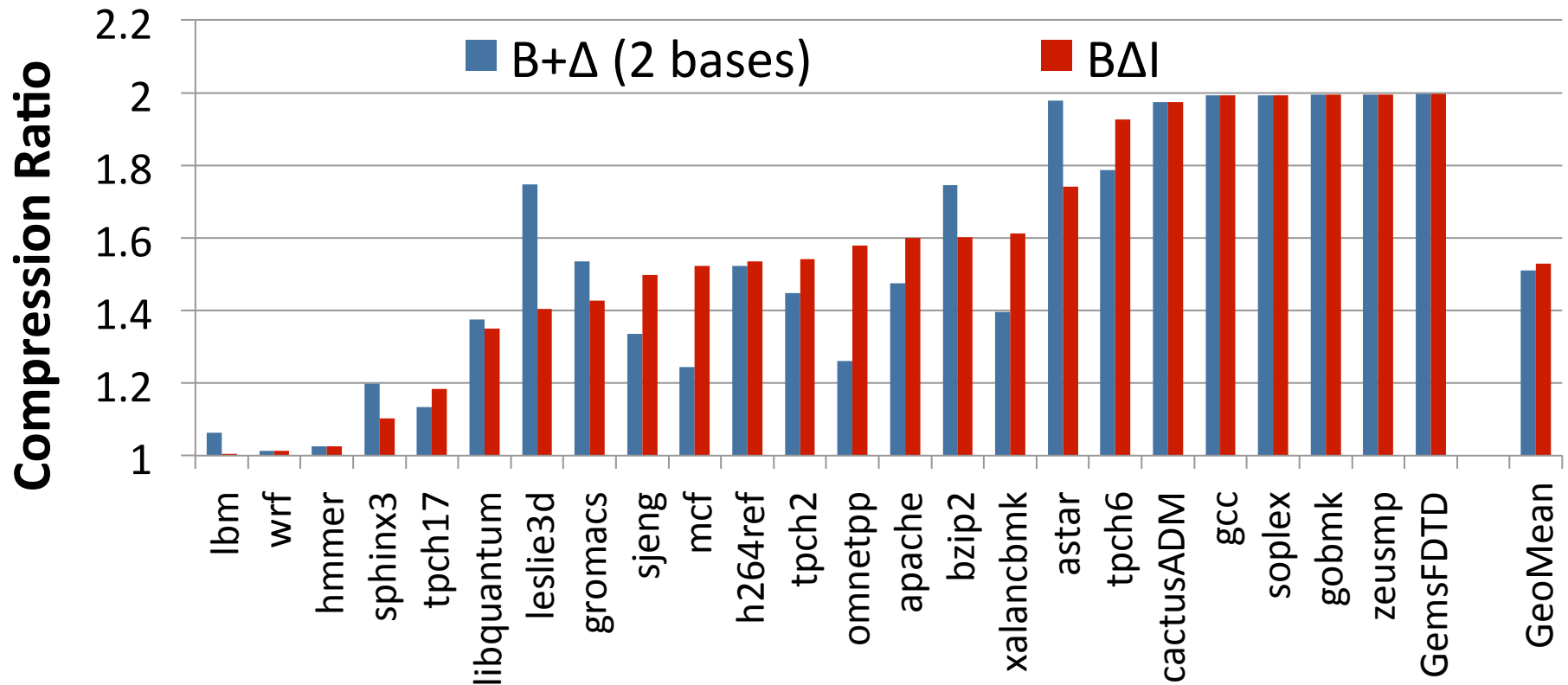
✓ Immediate part

Advantages over 2 arbitrary bases:

- Better compression ratio
- Simpler compression logic

Base-Delta-Immediate (BΔI) Compression

B+ Δ (with two arbitrary bases) vs. B Δ I



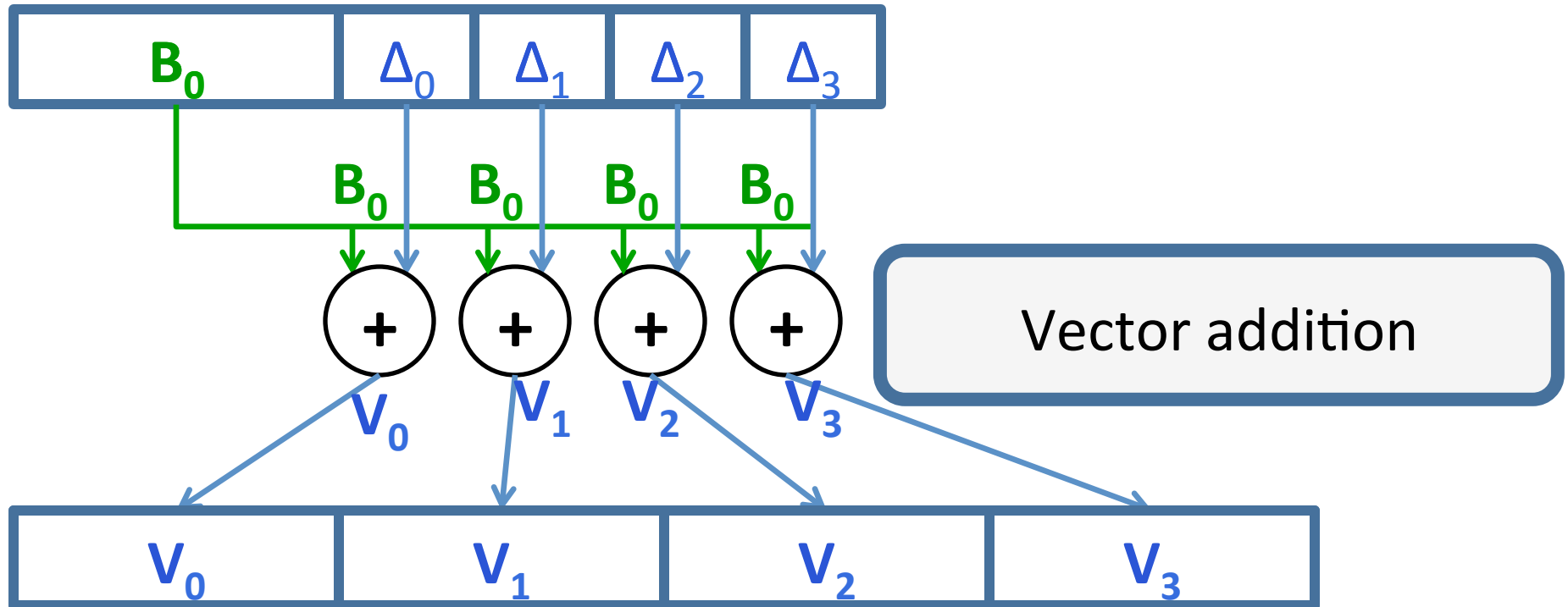
Average compression ratio is close, but **B Δ I** is **simpler**

B Δ I Implementation

- **Decompressor Design**
 - Low latency
- **Compressor Design**
 - Low cost and complexity
- **B Δ I Cache Organization**
 - Modest complexity

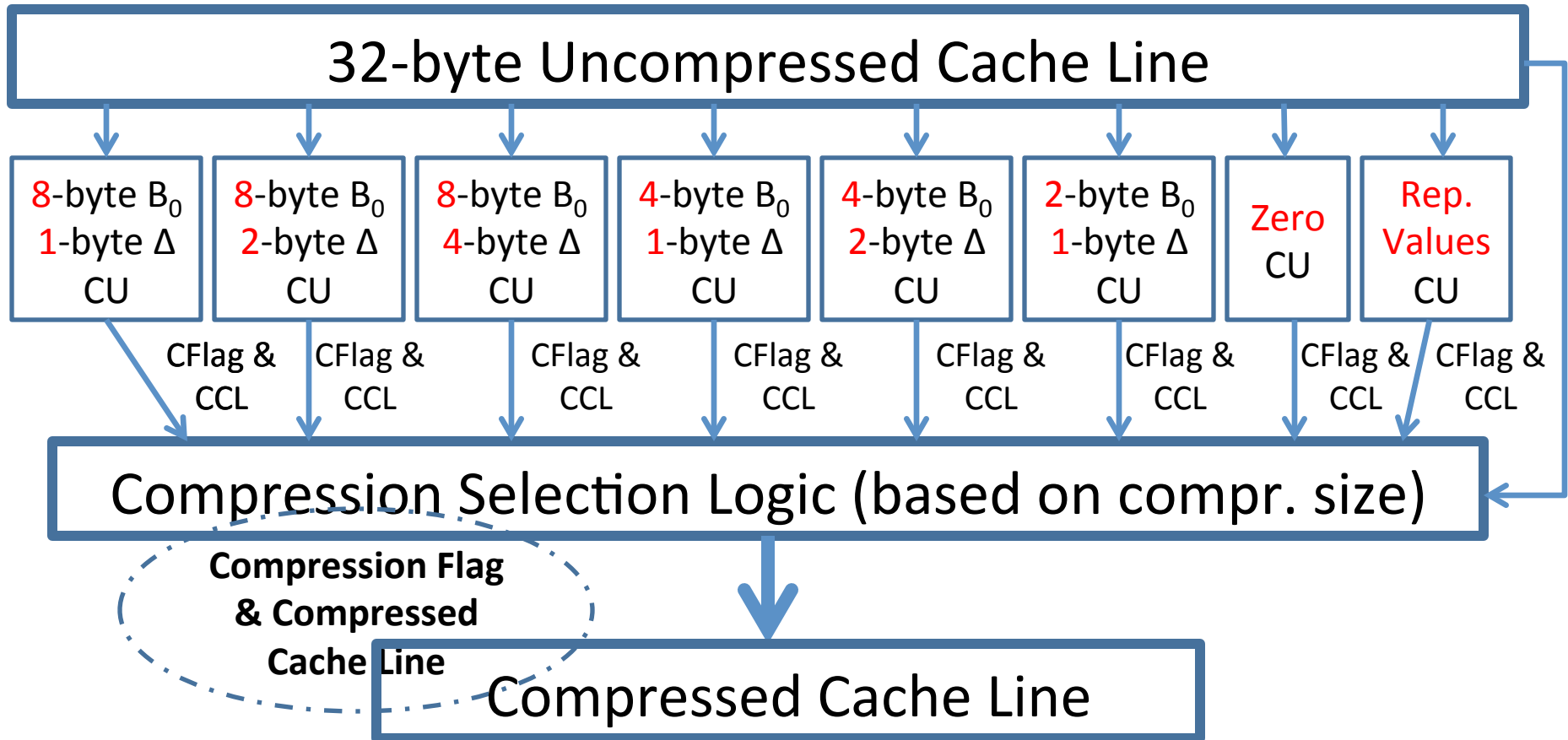
B Δ I Decompressor Design

Compressed Cache Line



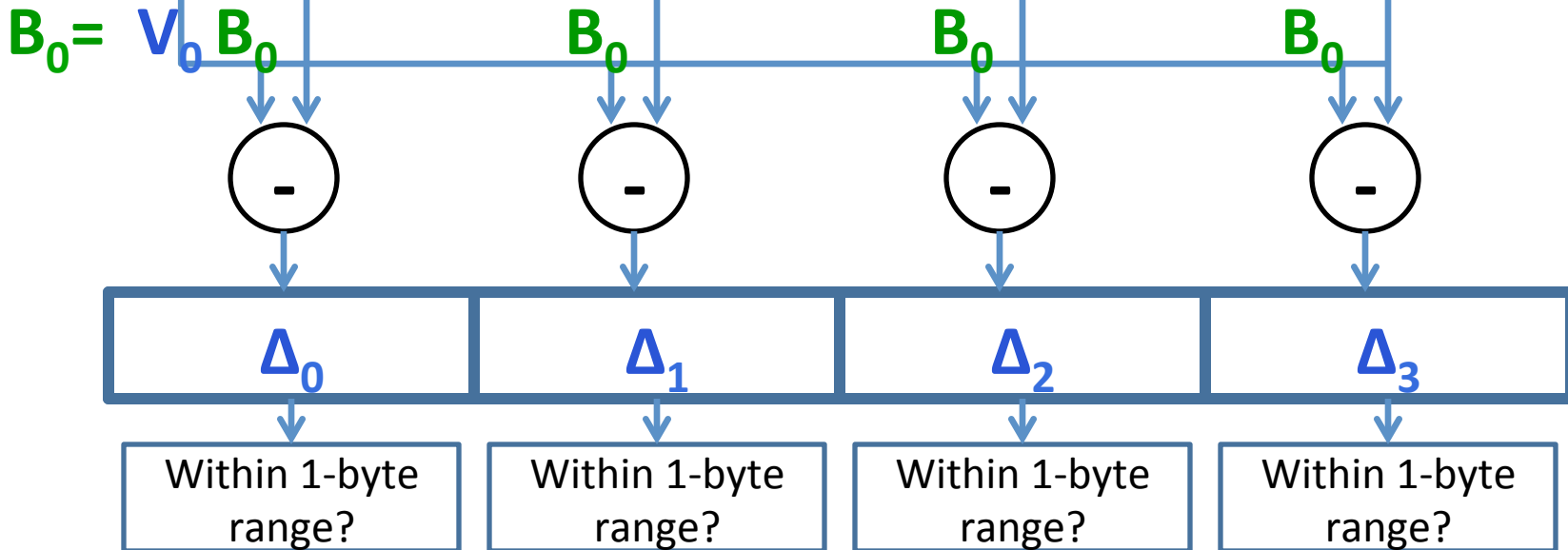
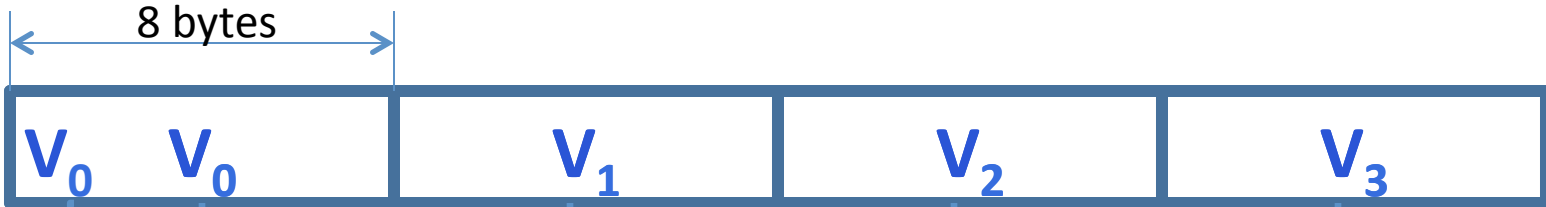
Uncompressed Cache Line

B Δ I Compressor Design

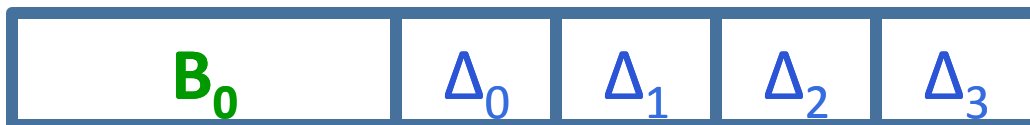


B Δ I Compression Unit: 8-byte B₀ 1-byte Δ

32-byte Uncompressed Cache Line

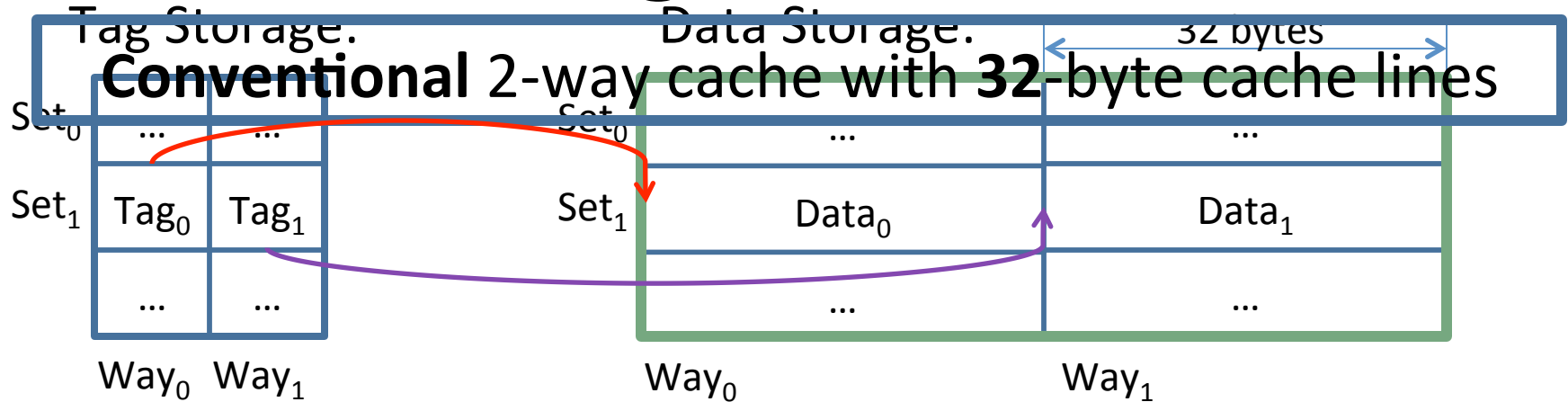


Is every element within 1-byte range?

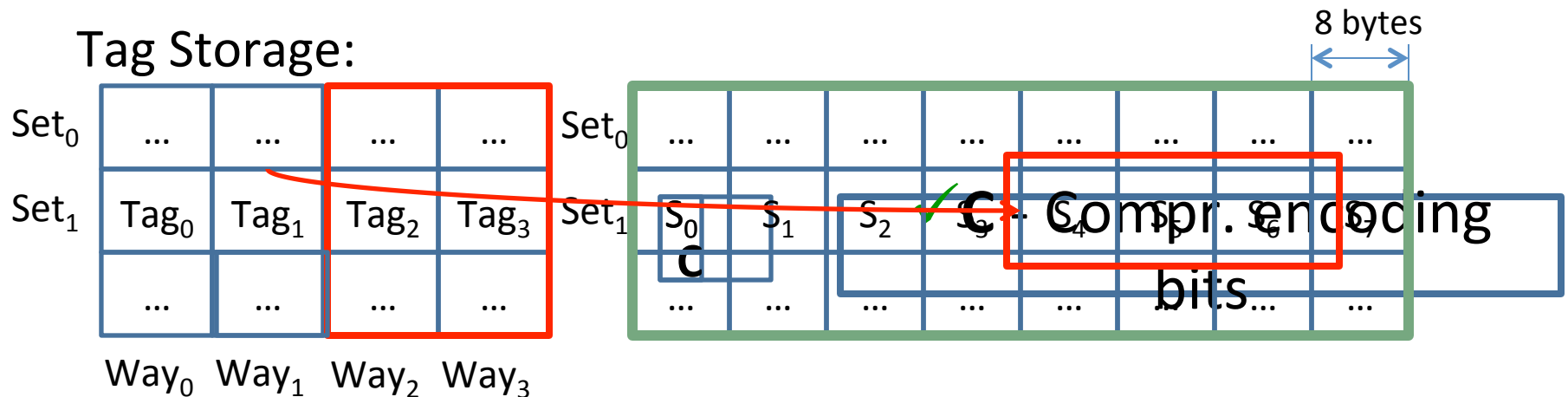


Yes No

BΔI Cache Organization



BΔI: 4-way cache with 8-byte segmented data



✓ Twice as many tags ✓ 20-30% overhead for 2 MB cache

Qualitative Comparison with Prior Work

- **Zero-based designs**
 - ZCA [Dusser+, ICS'09]: zero-content augmented cache
 - ZVC [Islam+, PACT'09]: zero-value cancelling
 - Limited applicability (only zero values)
- **FVC** [Yang+, MICRO'00]: frequent value compression
 - High decompression latency and complexity
- **Pattern-based compression designs**
 - FPC [Alameldeen+, ISCA'04]: frequent pattern compression
 - High decompression latency (5 cycles) and complexity
 - C-pack [Chen+, T-VLSI Systems'10]: practical implementation of FPC-like algorithm
 - High decompression latency (8 cycles)

Outline

- Motivation & Background
- Key Idea & Our Mechanism
- **Evaluation**
- Conclusion

Methodology

- **Simulator**

- x86 event-driven simulator based on Simics *[Magnusson+, Computer'02]*

- **Workloads**

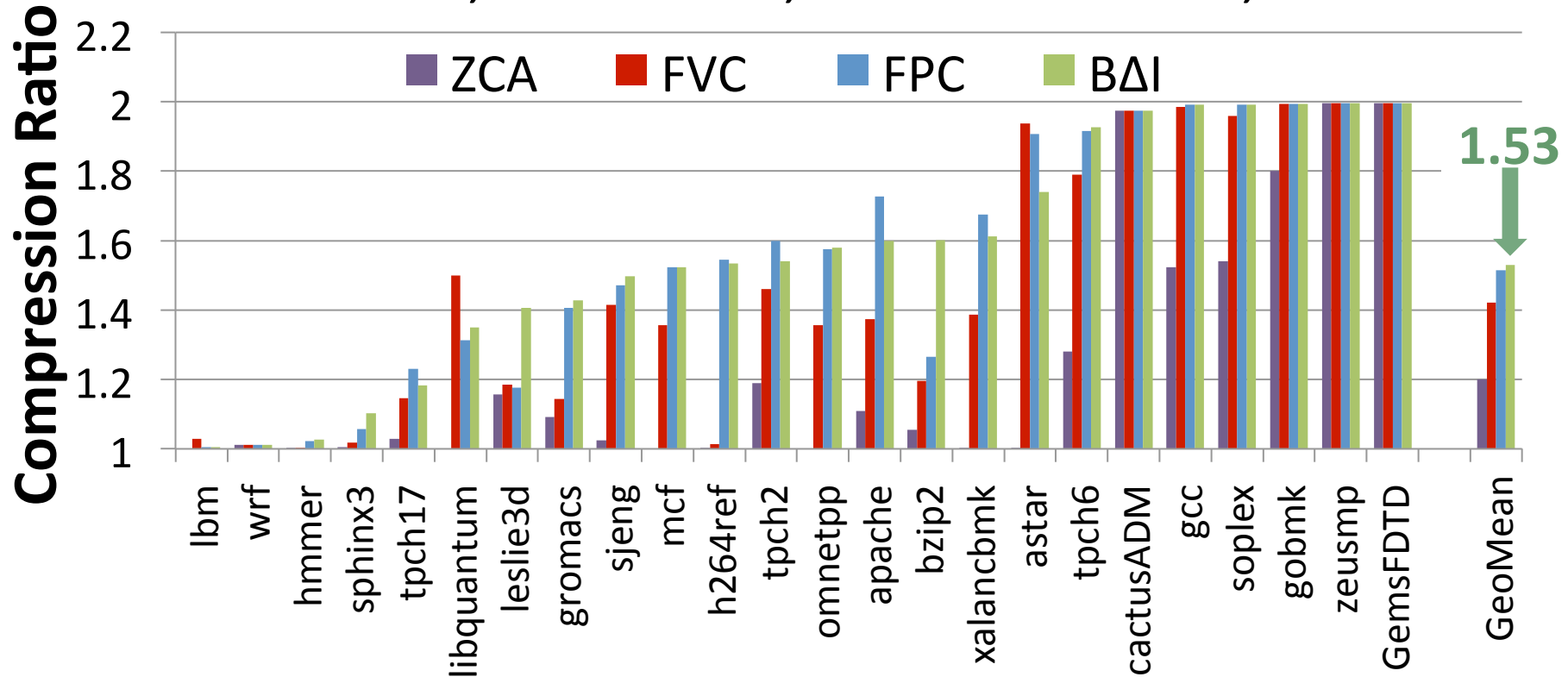
- SPEC2006 benchmarks, TPC, Apache web server
- 1 – 4 core simulations for 1 billion representative instructions

- **System Parameters**

- L1/L2/L3 cache latencies from CACTI *[Thoziyoor+, ISCA'08]*
- 4GHz, x86 in-order core, **512kB - 16MB** L2, simple memory model (**300**-cycle latency for row-misses)

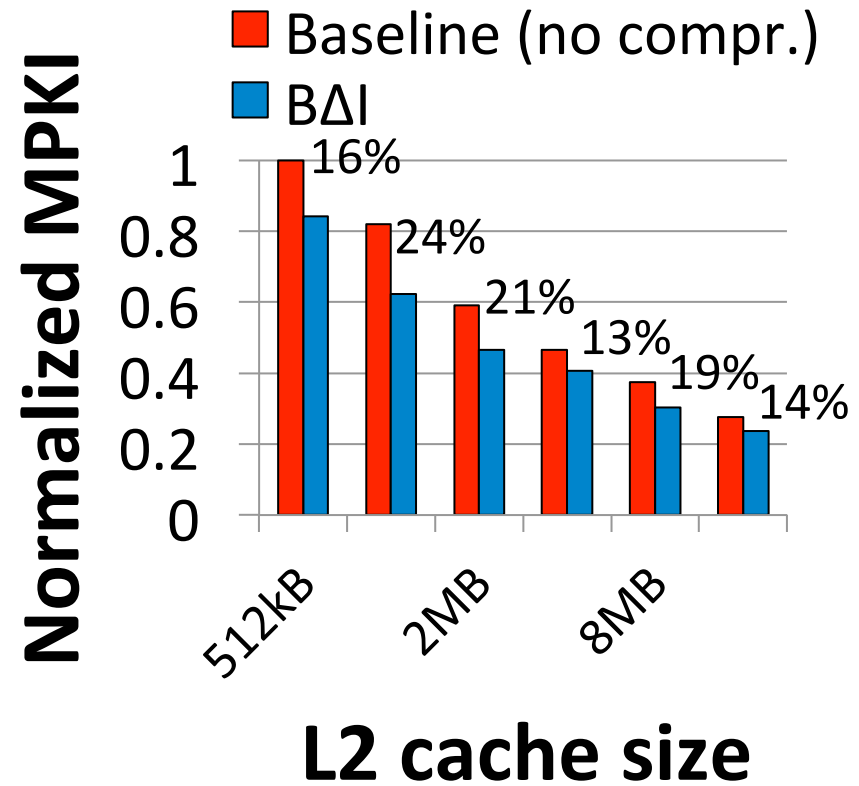
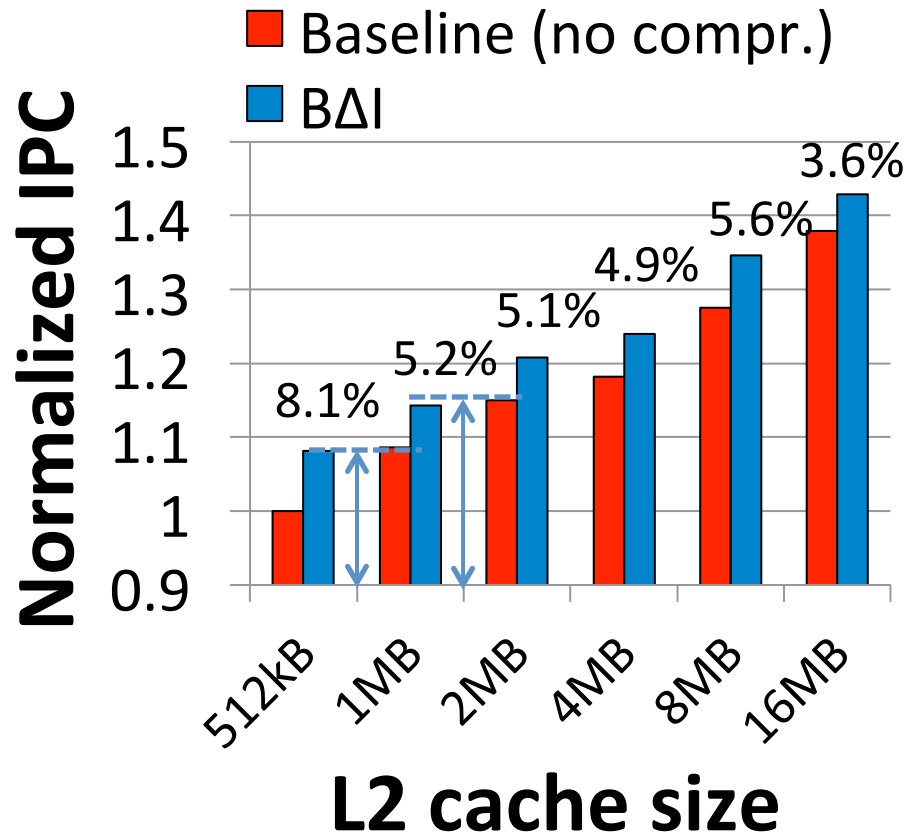
Compression Ratio: B Δ I vs. Prior Work

SPEC2006, databases, web workloads, 2MB L2



B Δ I achieves the highest compression ratio

Single-Core: IPC and MPKI



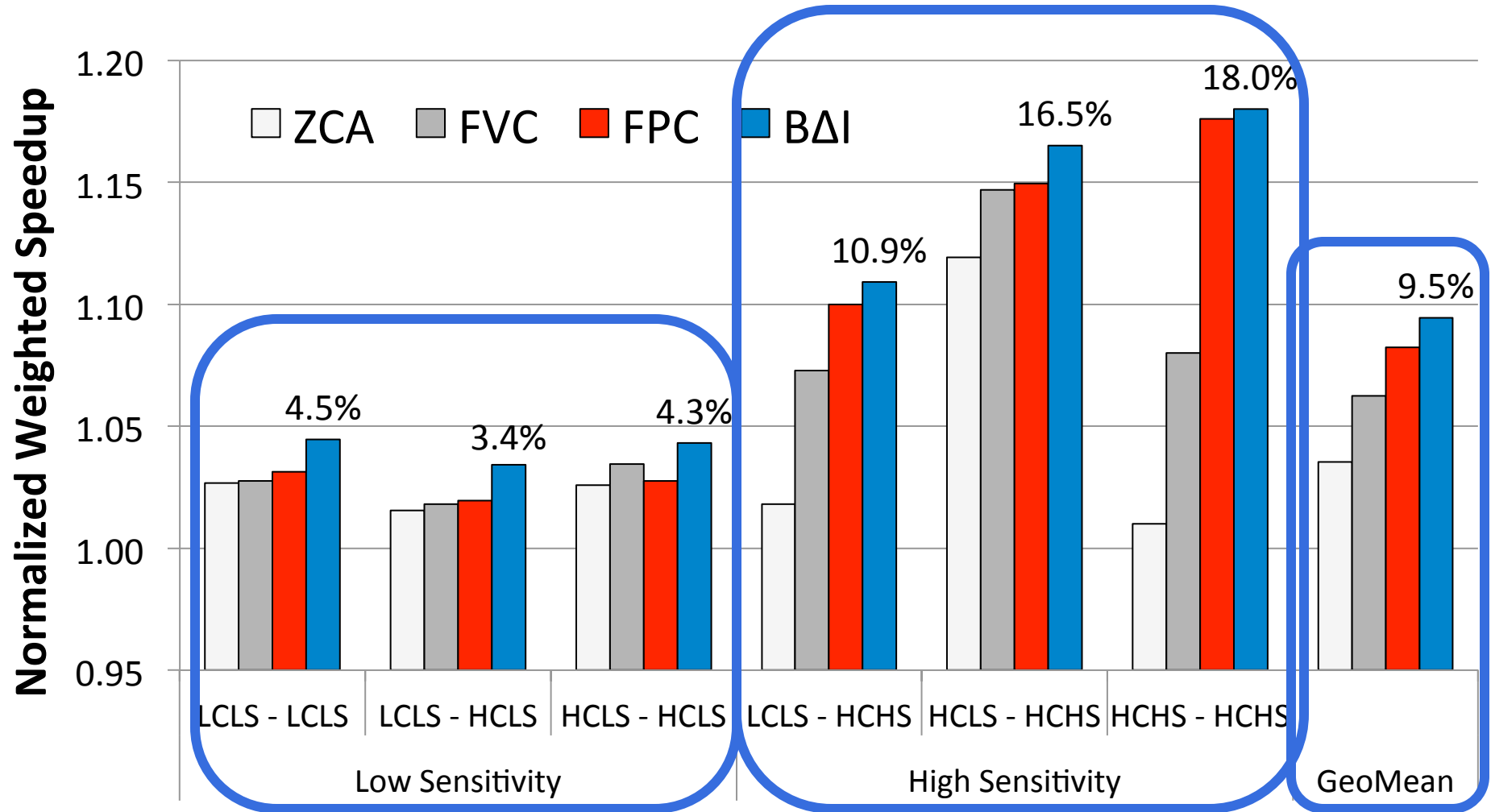
BΔI achieves the performance of a 2X-size cache

Performance improves due to the decrease in MPKI

Multi-Core Workloads

- Application classification based on
 - Compressibility:** effective cache size increase
(Low Compr. (**LC**) < 1.40, High Compr. (**HC**) >= 1.40)
 - Sensitivity:** performance gain with more cache
(Low Sens. (**LS**) < 1.10, High Sens. (**HS**) >= 1.10; 512kB -> 2MB)
- Three classes of applications:
 - LCLS, HCLS, HCHS, **no LCHS** applications
- For 2-core - **random** mixes of each possible class pairs
(20 each, 120 total workloads)

Multi-Core: Weighted Speedup



If at least one application is sensitive, then the BΔI performance improvement is the highest (9.5%) performance improves

Other Results in Paper

- IPC comparison against **upper** bounds
 - BΔI almost achieves performance of the 2X-size cache
- Sensitivity study of having **more** than 2X tags
 - Up to 1.98 average compression ratio
- Effect on **bandwidth** consumption
 - 2.31X decrease on average
- Detailed quantitative comparison with prior work
- **Cost analysis** of the proposed changes
 - 2.3% L2 cache area increase

Conclusion

- A new **Base-Delta-Immediate** compression mechanism
- Key insight: many cache lines can be efficiently represented using **base + delta encoding**
- Key properties:
 - **Low** latency decompression
 - **Simple** hardware implementation
 - **High compression ratio** with high coverage
- **Improves** *cache hit ratio* and *performance* of both single-core and multi-core workloads
 - Outperforms state-of-the-art cache compression techniques: FVC and FPC

Linearly Compressed Pages

Gennady Pekhimenko, Vivek Seshadri, Yoongu Kim, Hongyi Xin, Onur Mutlu,
Michael A. Kozuch, Phillip B. Gibbons, and Todd C. Mowry,
**"Linearly Compressed Pages: A Main Memory Compression
Framework with Low Complexity and Low Latency"**
SAFARI Technical Report, TR-SAFARI-2012-005, Carnegie Mellon University,
September 2012.

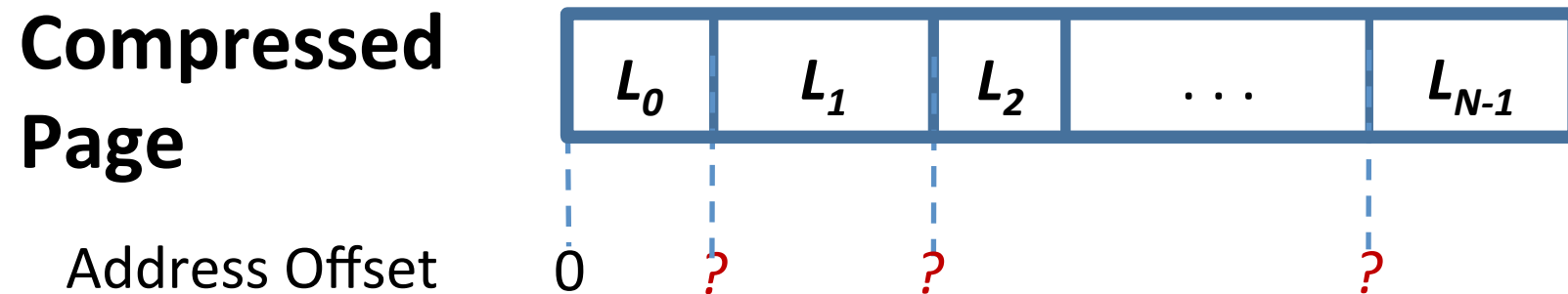
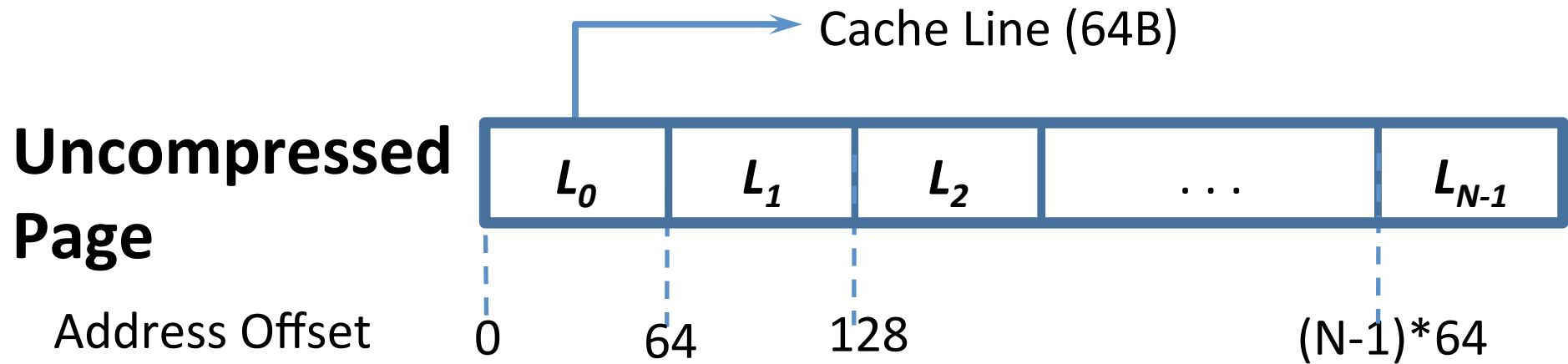
Executive Summary

- Main memory is a limited shared resource
- **Observation**: Significant data redundancy
- **Idea**: Compress data in main memory
- **Problem**: How to avoid latency increase?
- **Solution**: Linearly Compressed Pages (LCP):
fixed-size cache line granularity compression
 1. Increases capacity (**69%** on average)
 2. Decreases bandwidth consumption (**46%**)
 3. Improves overall performance (**9.5%**)

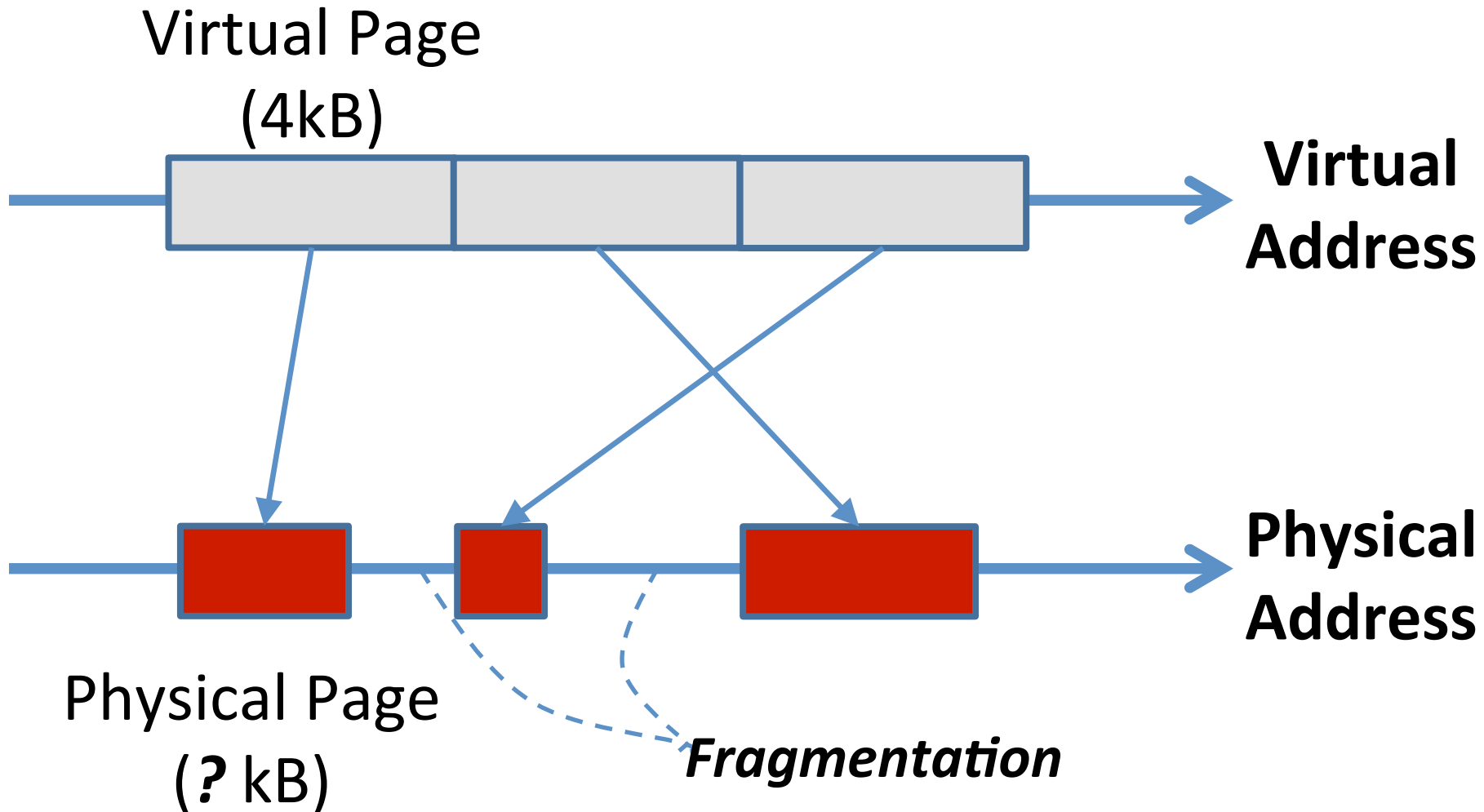
Challenges in Main Memory Compression

1. Address Computation
2. Mapping and Fragmentation
3. Physically Tagged Caches

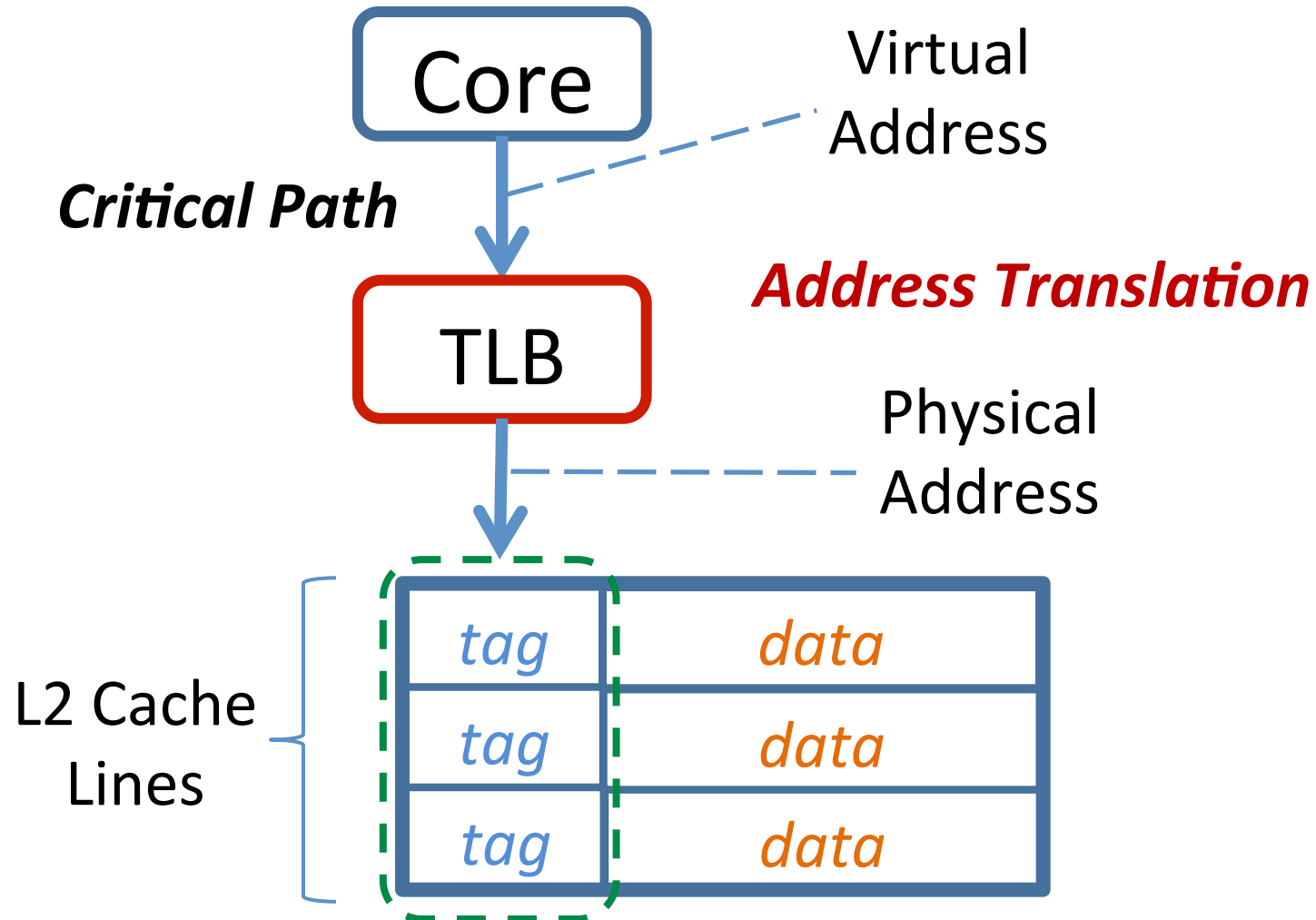
Address Computation



Mapping and Fragmentation



Physically Tagged Caches



Shortcomings of Prior Work

Compression Mechanisms	Access Latency	Decompression Latency	Complexity	Compression Ratio
IBM MXT <i>[IBM J.R.D. '01]</i>	✗	✗	✗	✓

Shortcomings of Prior Work

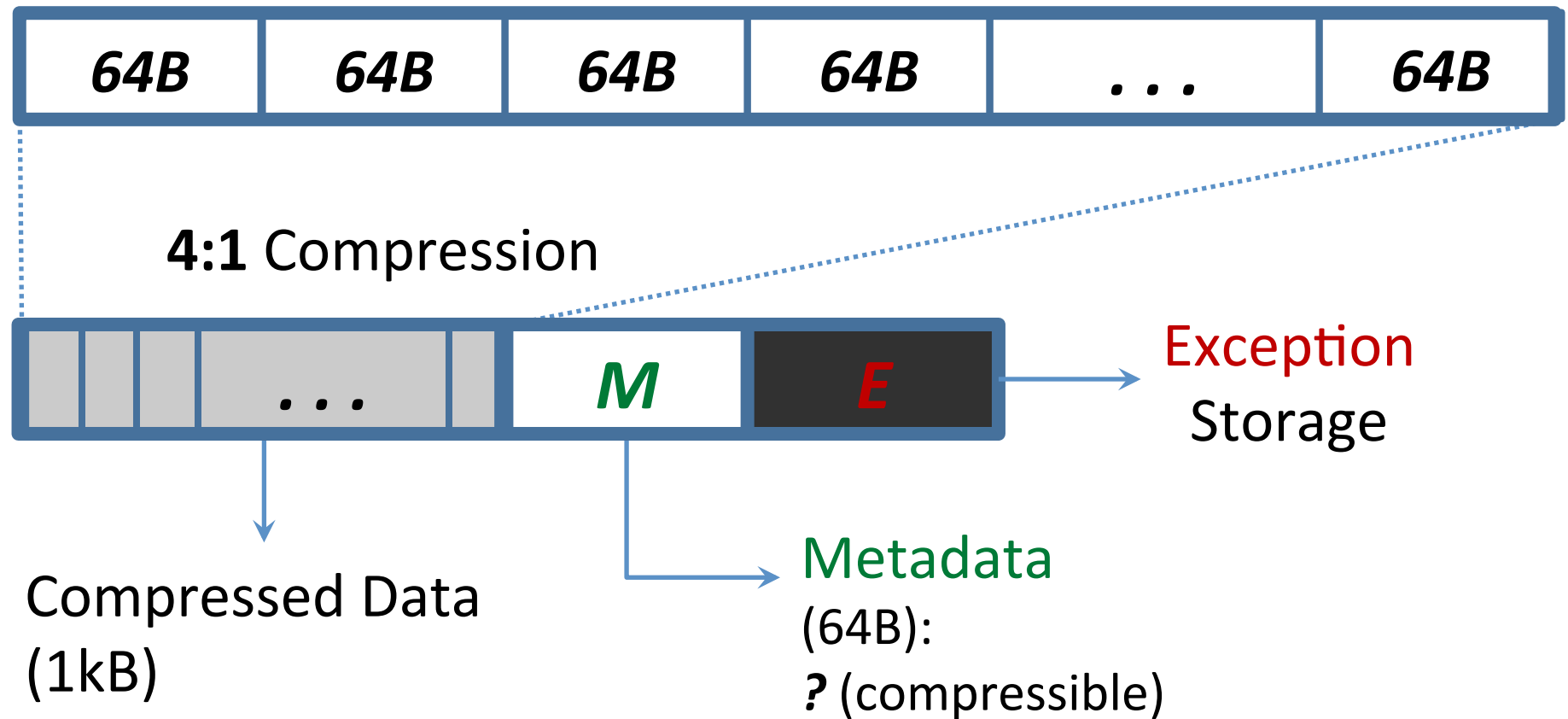
Compression Mechanisms	Access Latency	Decompression Latency	Complexity	Compression Ratio
IBM MXT <i>[IBM J.R.D. '01]</i>	✗	✗	✗	✓
Robust Main Memory Compression <i>[ISCA'05]</i>	✗	✓	✗	✓

Shortcomings of Prior Work

Compression Mechanisms	Access Latency	Decompression Latency	Complexity	Compression Ratio
IBM MXT <i>[IBM J.R.D. '01]</i>	✗	✗	✗	✓
Robust Main Memory Compression <i>[ISCA'05]</i>	✗	✓	✗	✓
LCP: Our Proposal	✓	✓	✓	✓

Linearly Compressed Pages (LCP): Key Idea

Uncompressed Page (4kB: 64*64B)



LCP Overview

- Page Table entry extension
 - compression type and size
 - extended physical base address
- Operating System management support
 - 4 memory pools (512B, 1kB, 2kB, 4kB)
- Changes to cache tagging logic
 - physical page base address + **cache line index**
(within a page)
- Handling page overflows
- Compression algorithms: **BDI** [PACT'12] , **FPC** [ISCA'04]

LCP Optimizations

- **Metadata** cache
 - Avoids additional requests to metadata
- Memory bandwidth reduction:



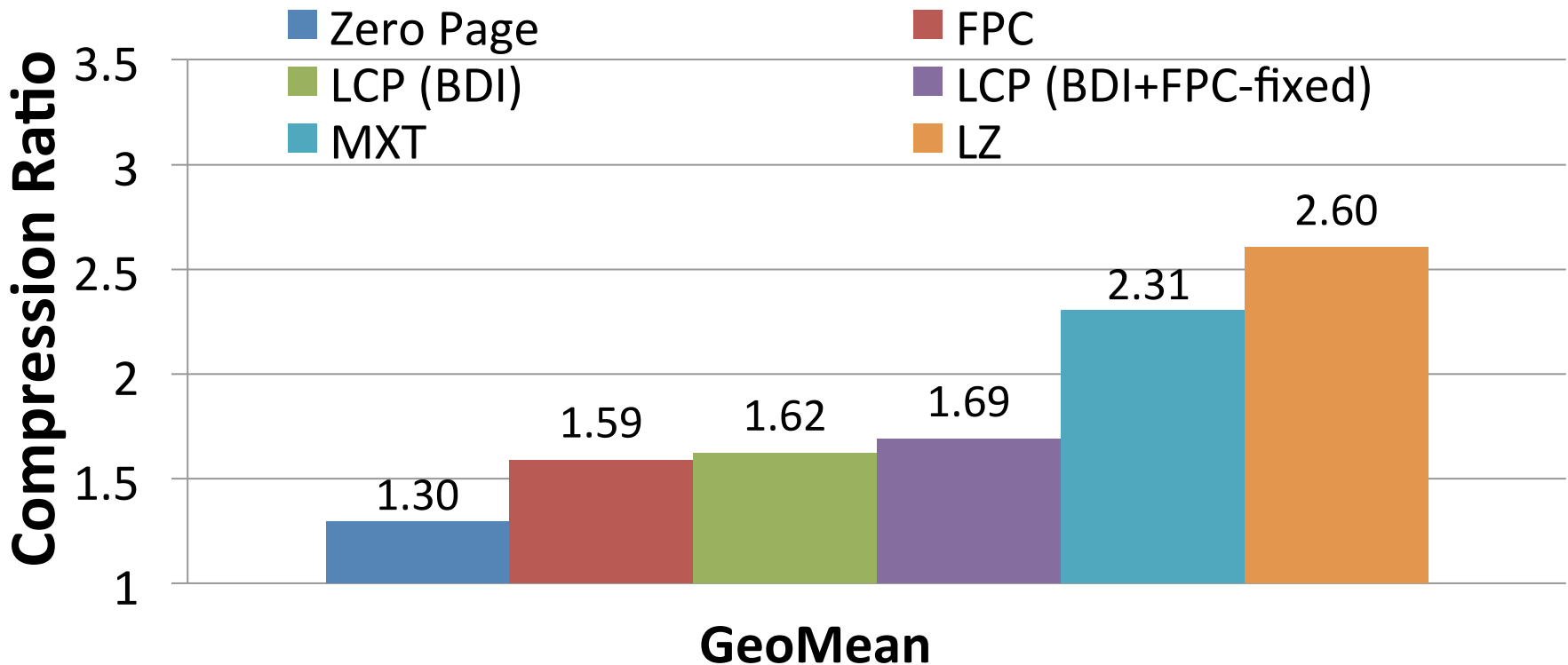
- Zero pages and zero cache lines
 - Handled separately in TLB (1-bit) and in metadata (1-bit per cache line)
- Integration with cache compression
 - BDI and FPC

Methodology

- **Simulator**
 - x86 event-driven simulators
 - Simics-based [Magnusson+, Computer'02] for CPU
 - Multi2Sim [Ubal+, PACT'12] for GPU
- **Workloads**
 - SPEC2006 benchmarks, TPC, Apache web server, GPGPU applications
- **System Parameters**
 - L1/L2/L3 cache latencies from CACTI [Thoziyoor+, ISCA'08]
 - 512kB - 16MB L2, simple memory model

Compression Ratio Comparison

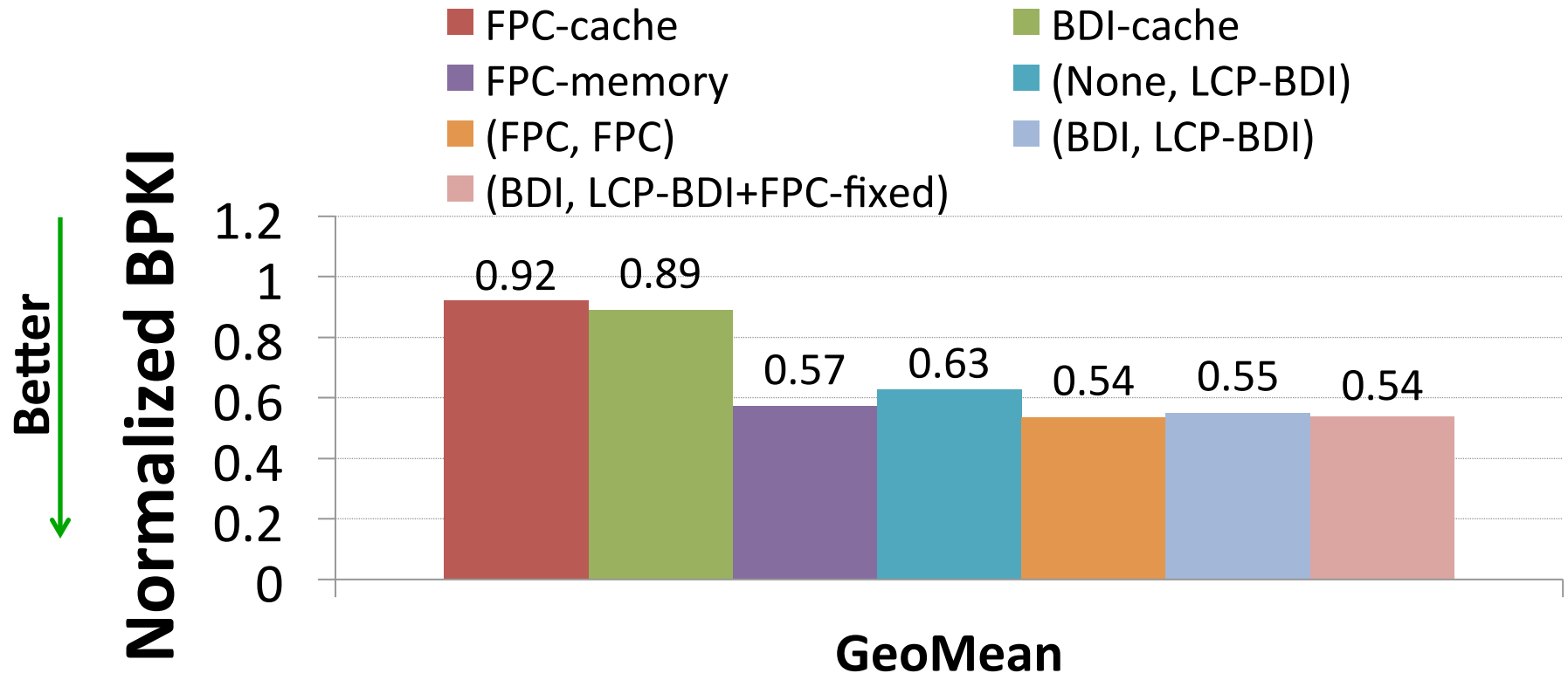
SPEC2006, databases, web workloads, 2MB L2 cache



LCP-based frameworks achieve competitive average compression ratios with prior work

Bandwidth Consumption Decrease

SPEC2006, databases, web workloads, 2MB L2 cache



LCP frameworks significantly reduce bandwidth (**46%**)

Performance Improvement

Cores	LCP-BDI	(BDI, LCP-BDI)	(BDI, LCP-BDI+FPC-fixed)
1	6.1%	9.5%	9.3%
2	13.9%	23.7%	23.6%
4	10.7%	22.6%	22.5%

LCP frameworks significantly improve performance

Conclusion

- A new main memory compression framework called **LCP (Linearly Compressed Pages)**
 - **Key idea: fixed size** for compressed cache lines within a page and **fixed compression algorithm** per page
- LCP evaluation:
 - Increases capacity (**69%** on average)
 - Decreases bandwidth consumption (**46%**)
 - Improves overall performance (**9.5%**)
 - Decreases energy of the off-chip bus (**37%**)

Controlled Shared Caching

Controlled Cache Sharing

■ Utility based cache partitioning

- Qureshi and Patt, “Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches,” MICRO 2006.
- Suh et al., “A New Memory Monitoring Scheme for Memory-Aware Scheduling and Partitioning,” HPCA 2002.

■ Fair cache partitioning

- Kim et al., “Fair Cache Sharing and Partitioning in a Chip Multiprocessor Architecture,” PACT 2004.

■ Shared/private mixed cache mechanisms

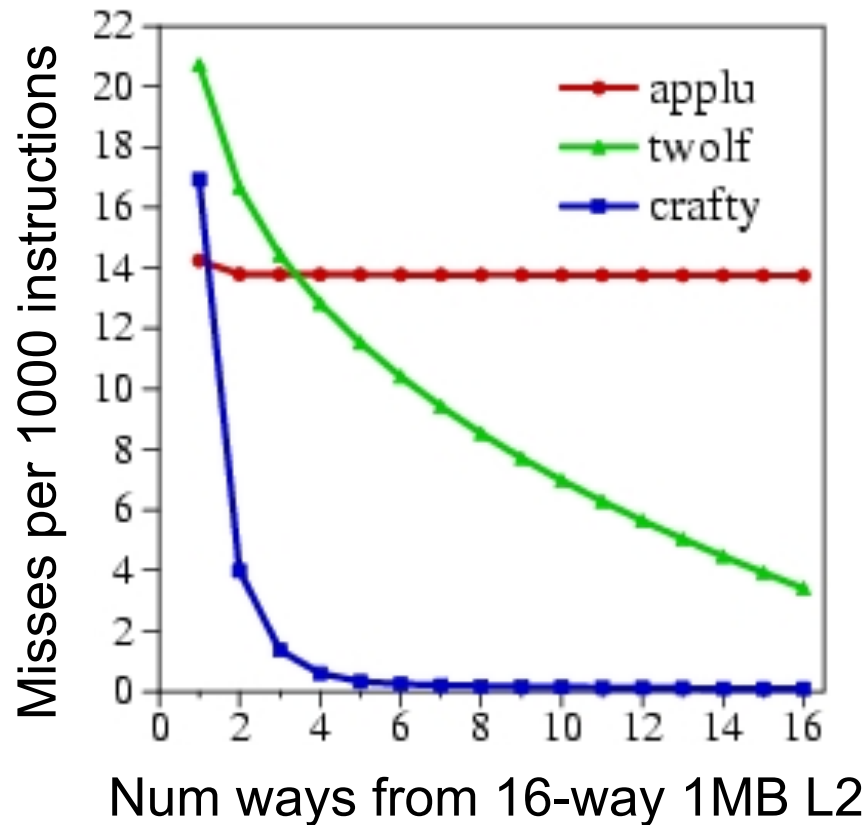
- Qureshi, “Adaptive Spill-Receive for Robust High-Performance Caching in CMPs,” HPCA 2009.
- Hardavellas et al., “Reactive NUCA: Near-Optimal Block Placement and Replication in Distributed Caches,” ISCA 2009.

Utility Based Shared Cache Partitioning

- Goal: Maximize system throughput
- Observation: Not all threads/applications benefit equally from caching → simple LRU replacement not good for system throughput
- Idea: Allocate more cache space to applications that obtain the most benefit from more space
- The high-level idea can be applied to other shared resources as well.
- Qureshi and Patt, “Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches,” MICRO 2006.
- Suh et al., “A New Memory Monitoring Scheme for Memory-Aware Scheduling and Partitioning,” HPCA 2002.

Marginal Utility of a Cache Way

Utility $U_a^b = \text{Misses with } a \text{ ways} - \text{Misses with } b \text{ ways}$

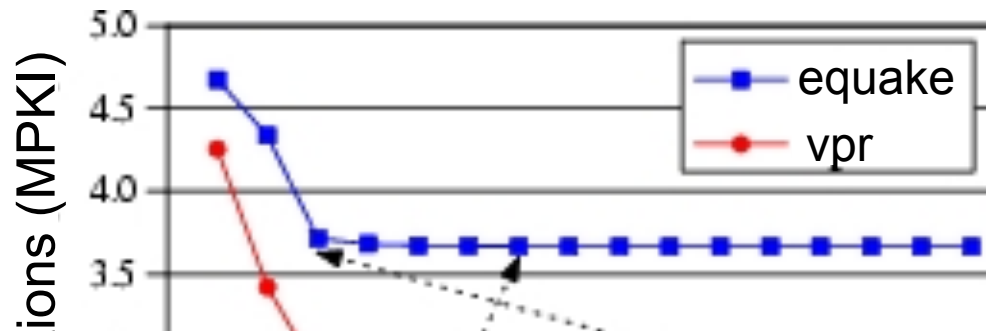


Low Utility

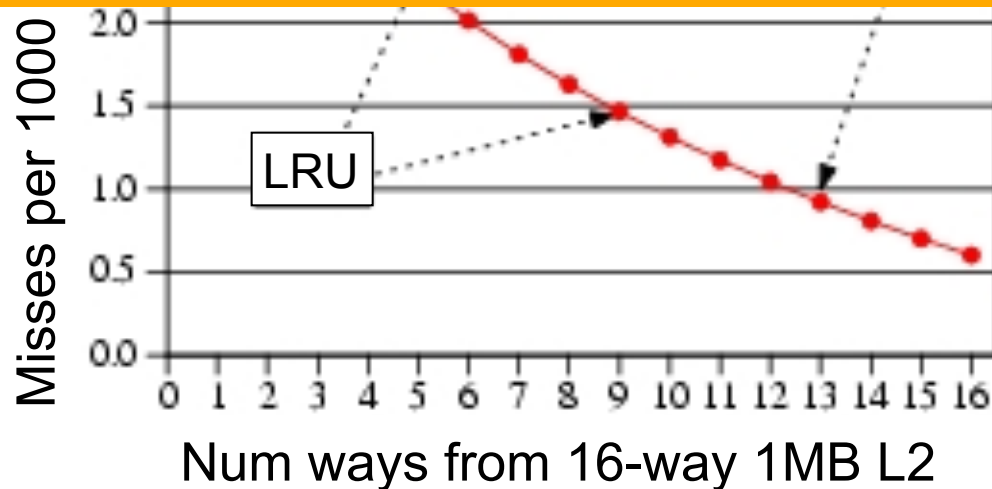
High Utility

Saturating Utility

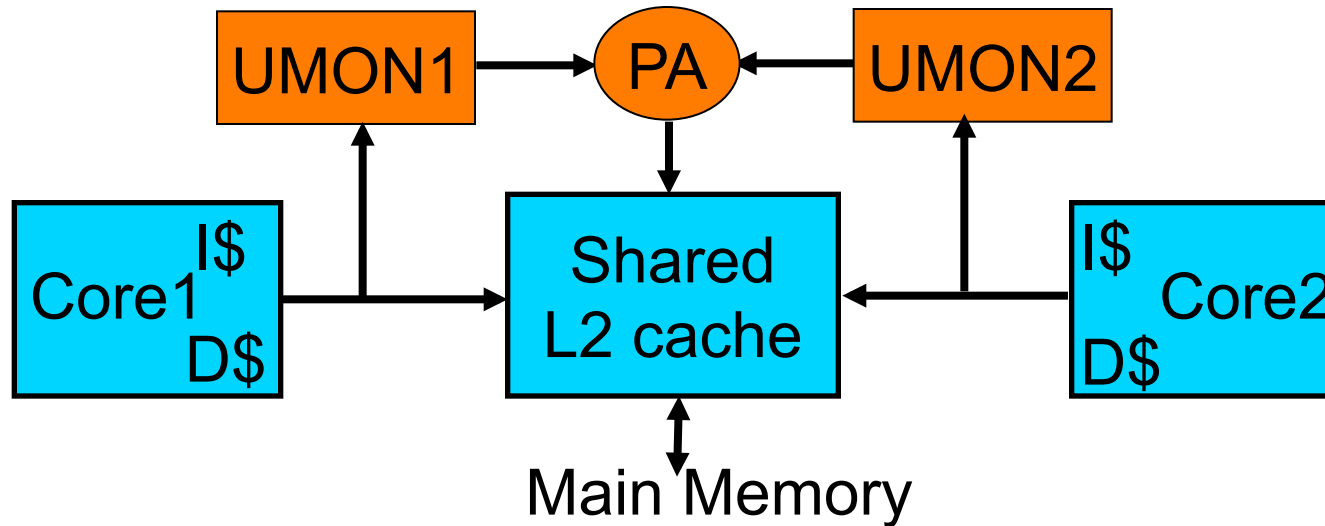
Utility Based Shared Cache Partitioning Motivation



Improve performance by giving more cache to the application that benefits more from cache



Utility Based Cache Partitioning (III)



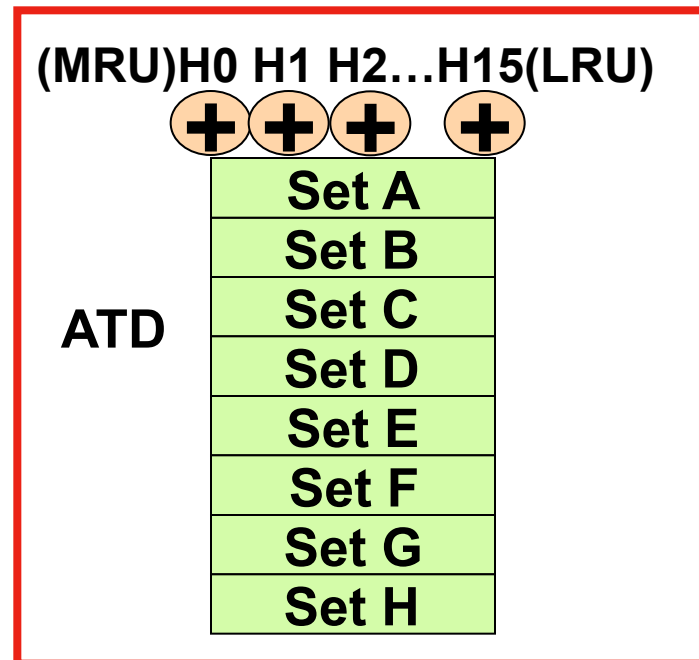
Three components:

- ❑ Utility Monitors (UMON) per core
- ❑ Partitioning Algorithm (PA)
- ❑ Replacement support to enforce partitions

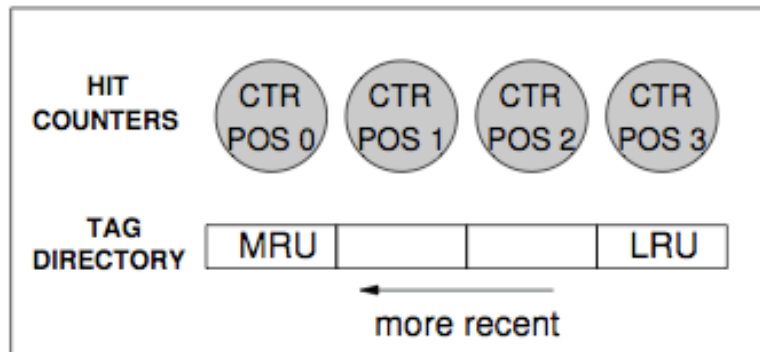
Utility Monitors

- ❑ For each core, simulate LRU policy using ATD
- ❑ Hit counters in ATD to count hits per recency position
- ❑ LRU is a stack algorithm: hit counts \rightarrow utility
E.g. hits(2 ways) = $H_0 + H_1$

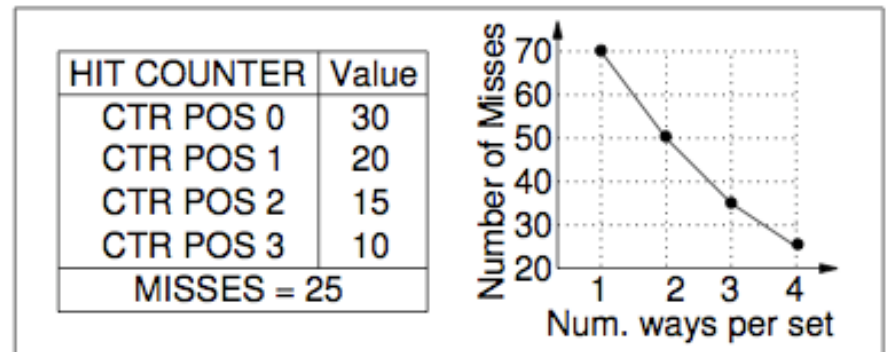
MTD
Set A
Set B
Set C
Set D
Set E
Set F
Set G
Set H



Utility Monitors



(a)

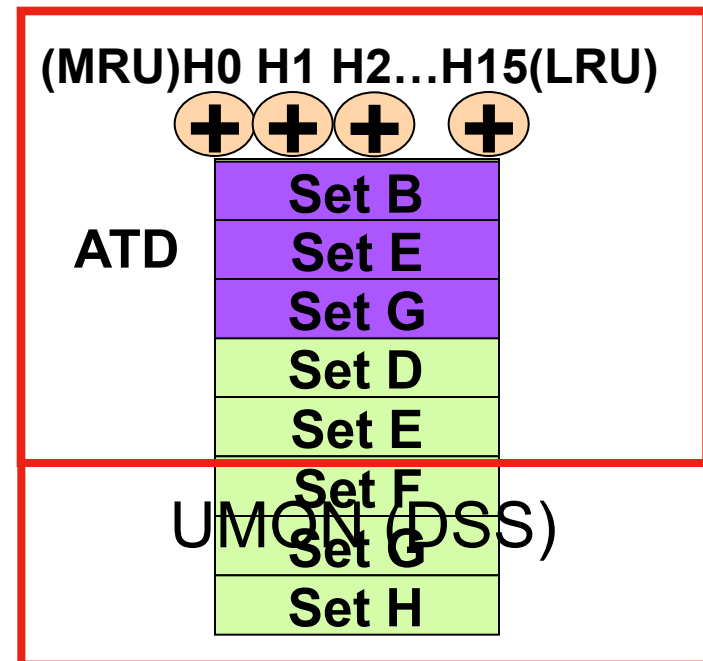
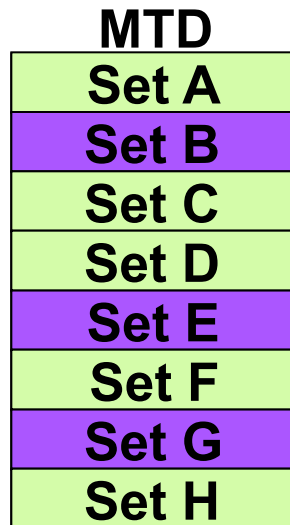


(b)

Figure 4. (a) Hit counters for each recency position. (b) Example of how utility information can be tracked with stack property.

Dynamic Set Sampling

- ❑ Extra tags incur hardware and power overhead
- ❑ Dynamic Set Sampling reduces overhead [Qureshi, ISCA'06]
- ❑ 32 sets sufficient (analytical bounds)
- ❑ Storage < 2kB/UMON



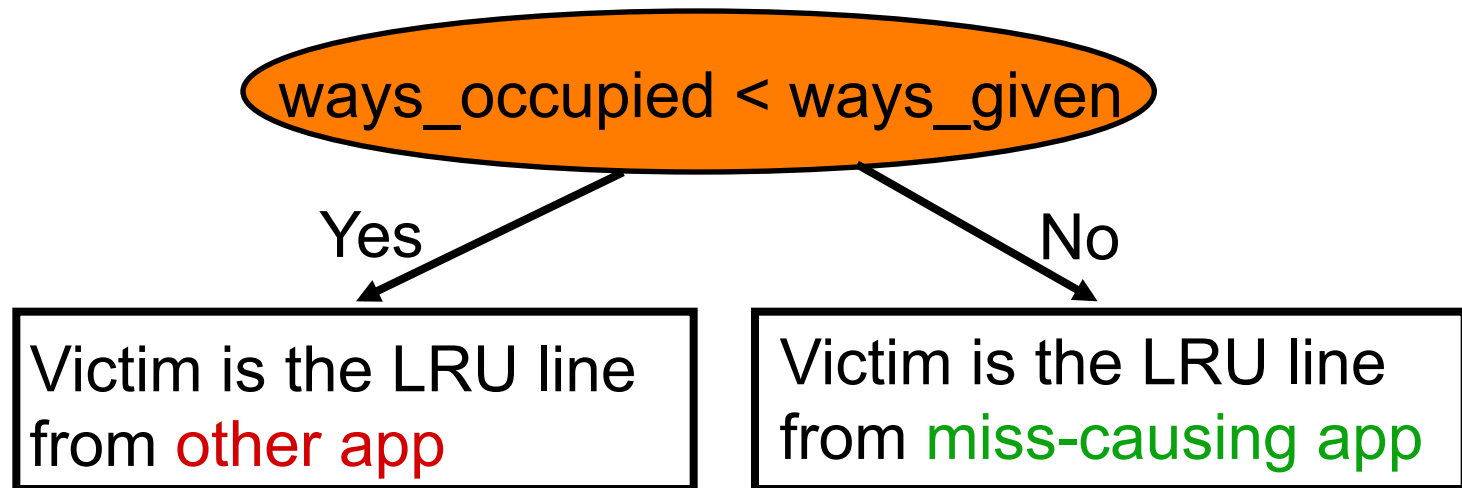
Partitioning Algorithm

- ❑ Evaluate all possible partitions and select the best
- ❑ With a ways to core1 and $(16-a)$ ways to core2:
$$\text{Hits}_{\text{core1}} = (H_0 + H_1 + \dots + H_{a-1}) \quad \text{---- from UMON1}$$
$$\text{Hits}_{\text{core2}} = (H_0 + H_1 + \dots + H_{16-a-1}) \quad \text{---- from UMON2}$$
- ❑ Select a that maximizes $(\text{Hits}_{\text{core1}} + \text{Hits}_{\text{core2}})$
- ❑ Partitioning done once every 5 million cycles

Way Partitioning

Way partitioning support: [Suh+ HPCA' 02, Iyer ICS' 04]

1. Each line has core-id bits
2. On a miss, count **ways_occupied** in set by miss-causing app



Performance Metrics

- Three metrics for performance:

1. Weighted Speedup (default metric)

- $\text{perf} = \text{IPC}_1 / \text{SingleIPC}_1 + \text{IPC}_2 / \text{SingleIPC}_2$
- correlates with reduction in execution time

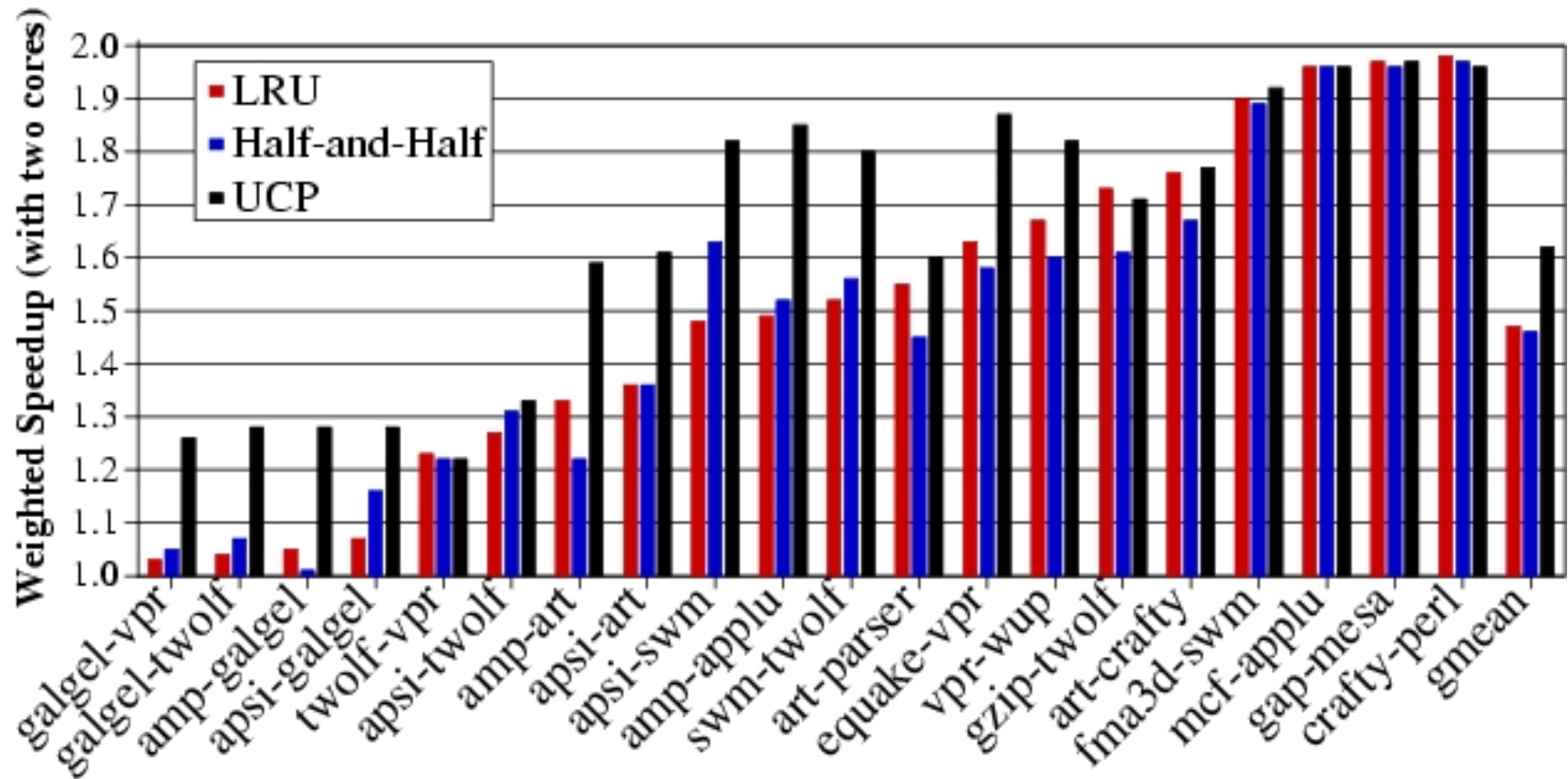
2. Throughput

- $\text{perf} = \text{IPC}_1 + \text{IPC}_2$
- can be unfair to low-IPC application

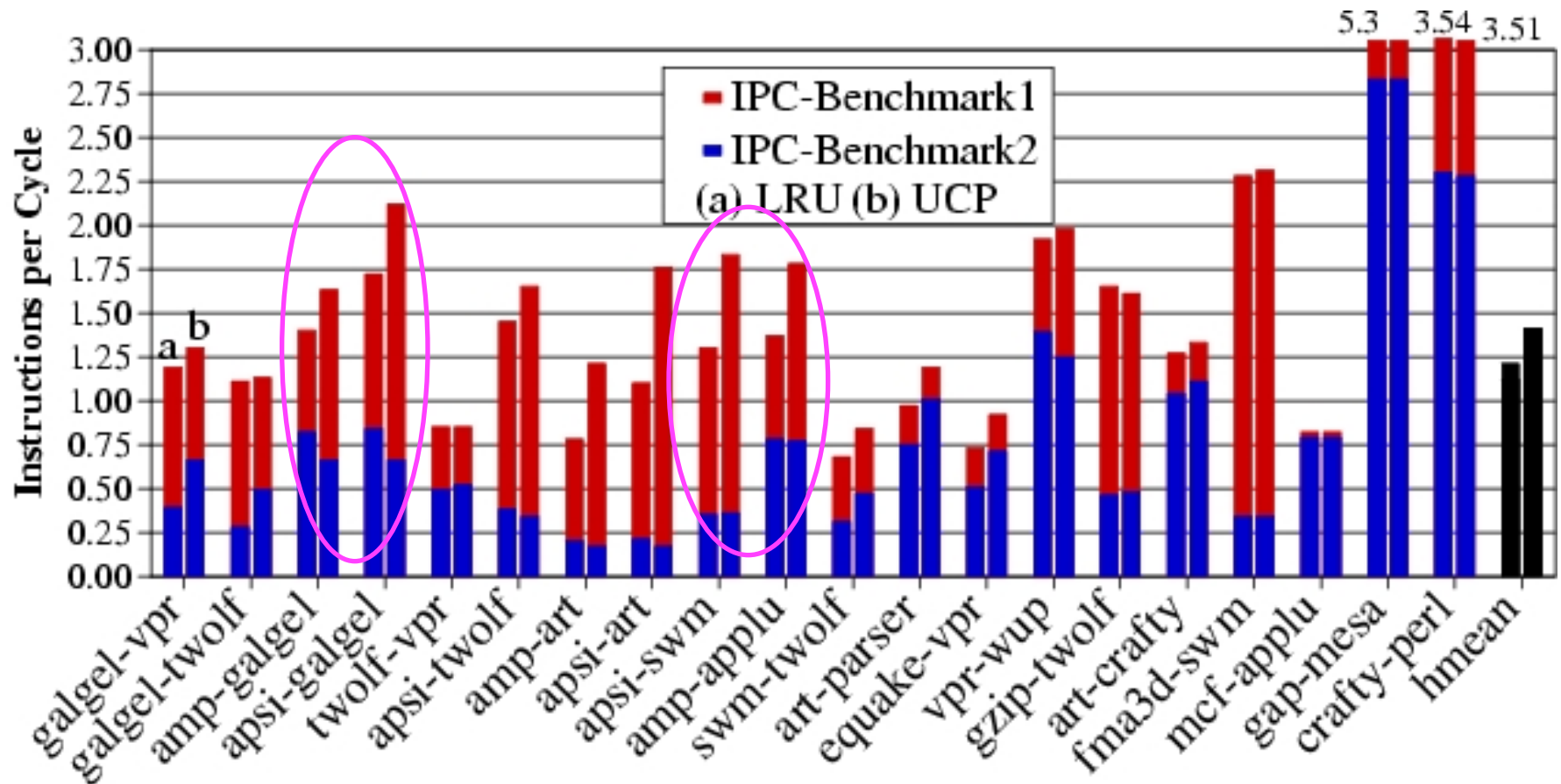
3. Hmean-fairness

- $\text{perf} = \text{hmean}(\text{IPC}_1 / \text{SingleIPC}_1, \text{IPC}_2 / \text{SingleIPC}_2)$
- balances fairness and performance

Weighted Speedup Results for UCP



IPC Results for UCP



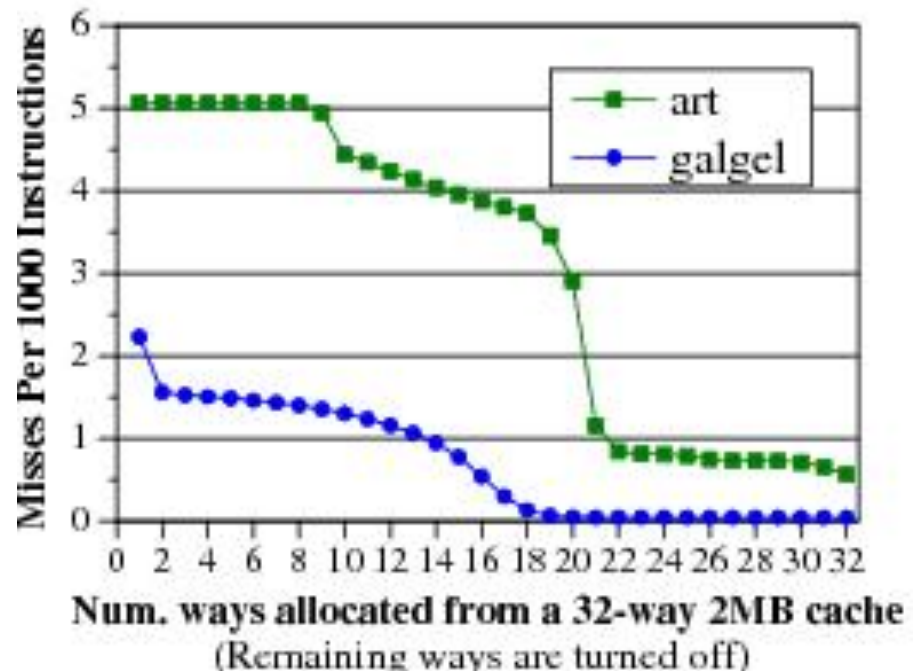
UCP improves average throughput by 17%

Any Problems with UCP So Far?

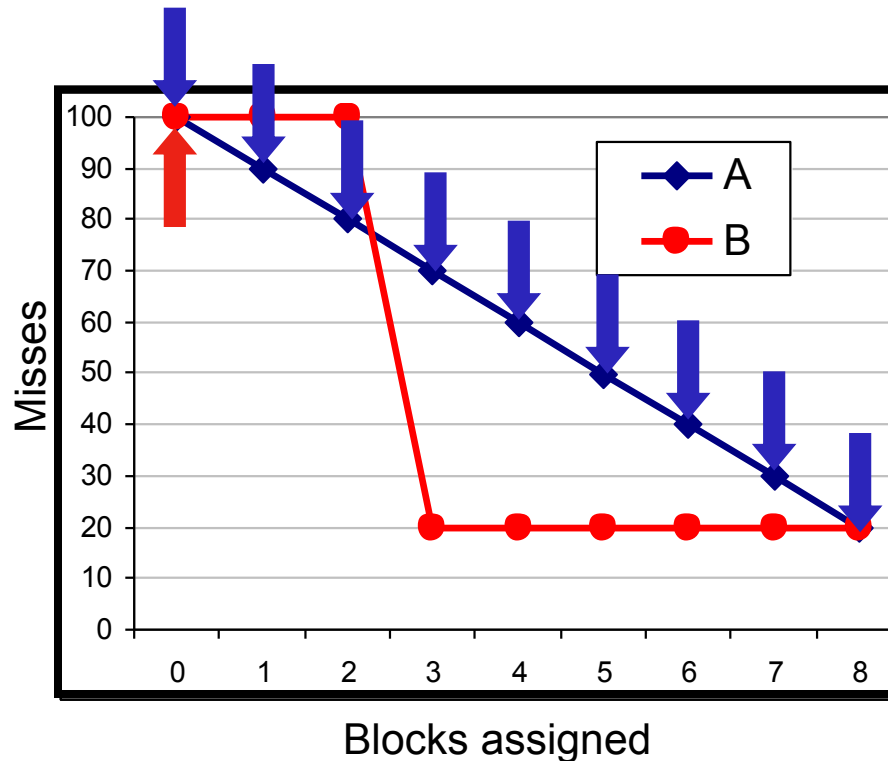
- Scalability
- Non-convex curves?
- Time complexity of partitioning low for two cores (number of possible partitions \approx number of ways)
- Possible partitions increase exponentially with cores
- For a 32-way cache, possible partitions:
 - 4 cores \rightarrow 6545
 - 8 cores \rightarrow 15.4 million
- Problem NP hard \rightarrow need scalable partitioning algorithm

Greedy Algorithm [Stone+ ToC '92]

- GA allocates 1 block to the app that has the max utility for one block. Repeat till all blocks allocated
- Optimal partitioning when utility curves are convex
- Pathological behavior for non-convex curves



Problem with Greedy Algorithm



In each iteration, the utility for 1 block:

$$U(A) = 10 \text{ misses}$$

$$U(B) = 0 \text{ misses}$$

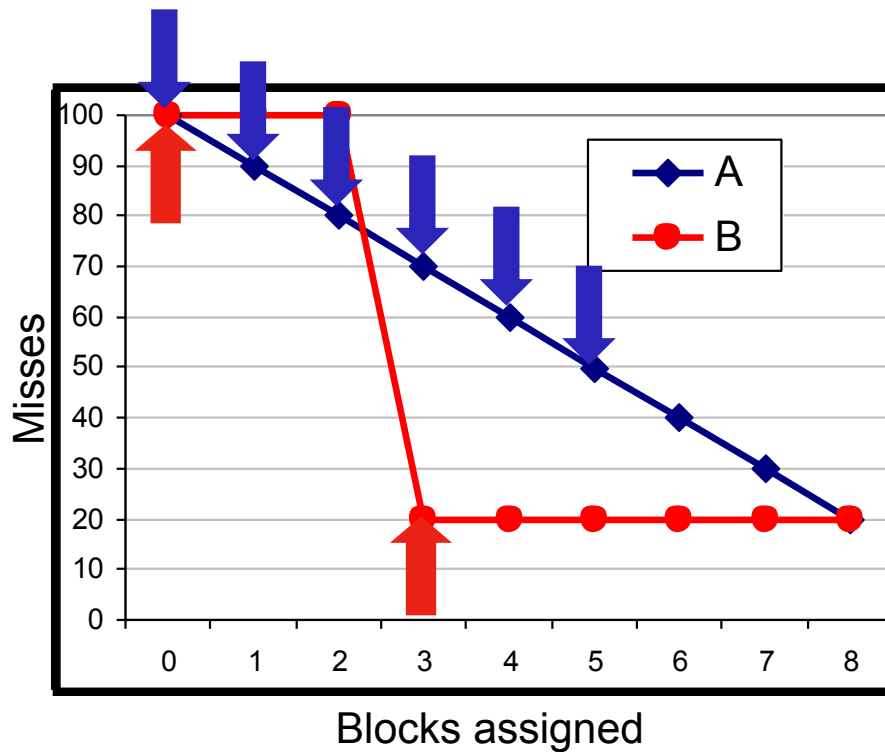
All blocks assigned to A, even if B has same miss reduction with fewer blocks

- Problem: GA considers benefit only from the immediate block. Hence, it fails to exploit large gains from looking ahead

Lookahead Algorithm

- Marginal Utility (MU) = Utility per cache resource
 - $MU_a^b = U_a^b / (b - a)$
- GA considers MU for 1 block. LA considers MU for all possible allocations
- Select the app that has the max value for MU.
Allocate it as many blocks required to get max MU
- Repeat till all blocks assigned

Lookahead Algorithm Example



Iteration 1:

$MU(A) = 10/1$ block

$MU(B) = 80/3$ blocks

B gets 3 blocks

Next five iterations:

$MU(A) = 10/1$ block

$MU(B) = 0$

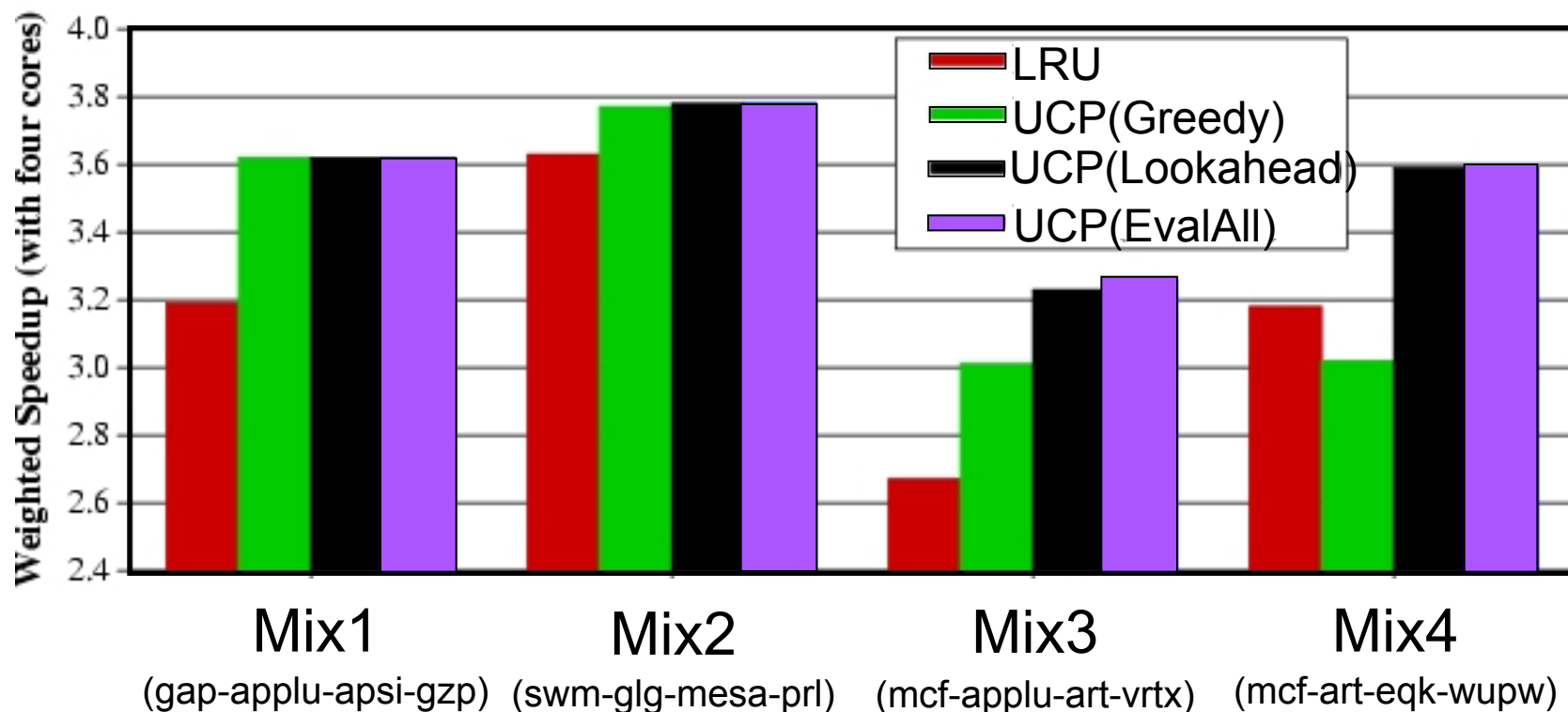
A gets 1 block

Result: A gets 5 blocks and B gets 3 blocks (Optimal)

Time complexity $\approx \text{ways}^2/2$ (512 ops for 32-ways)

UCP Results

Four cores sharing a 2MB 32-way L2



LA performs similar to EvalAll, with low time-complexity

Utility Based Cache Partitioning

- Advantages over LRU
 - + Improves system throughput
 - + Better utilizes the shared cache

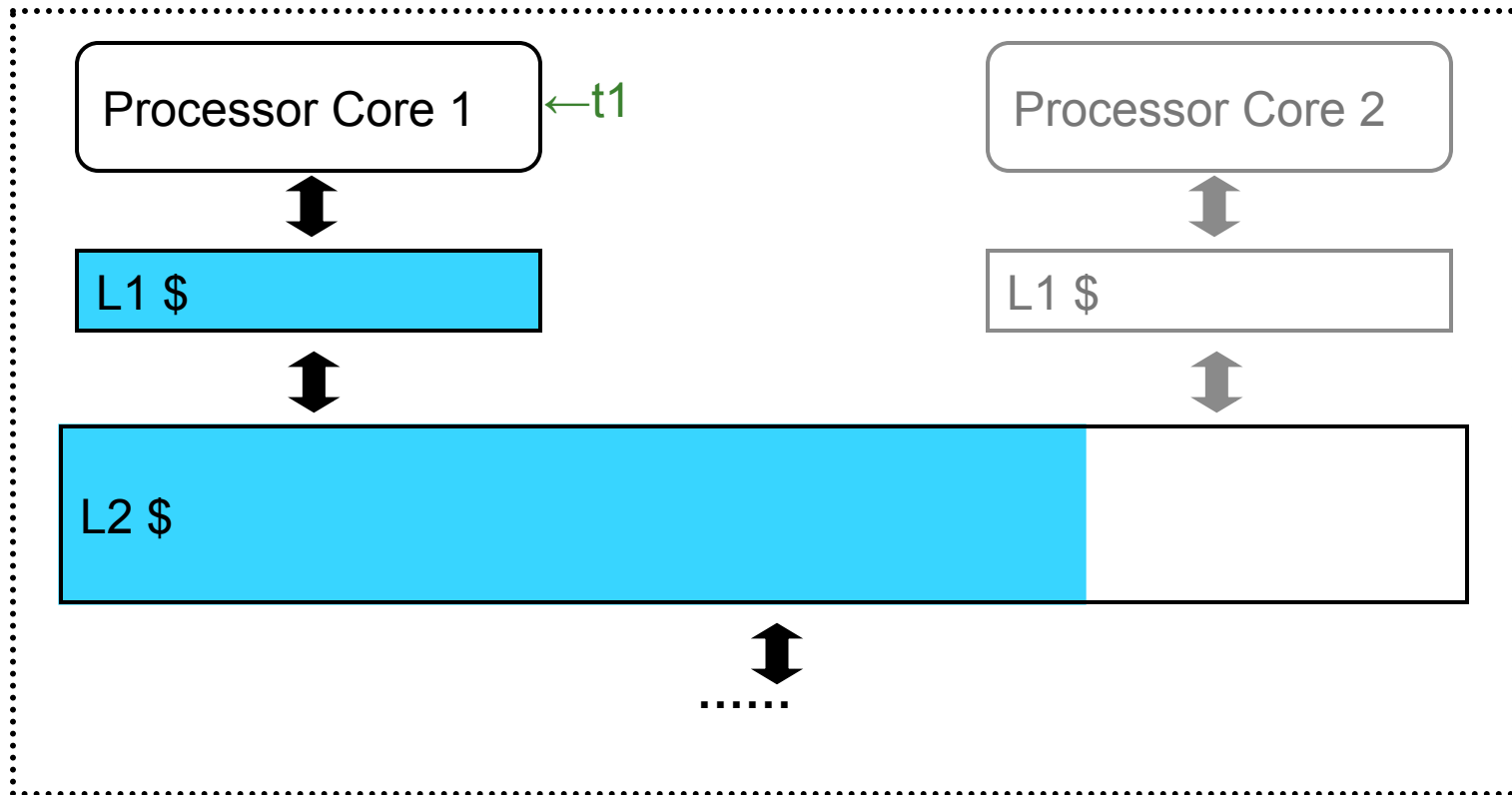
- Disadvantages
 - Fairness, QoS?

- Limitations
 - Scalability: Partitioning limited to ways. What if you have $\text{numWays} < \text{numApps}$?
 - Scalability: How is utility computed in a distributed cache?
 - What if past behavior is not a good predictor of utility?

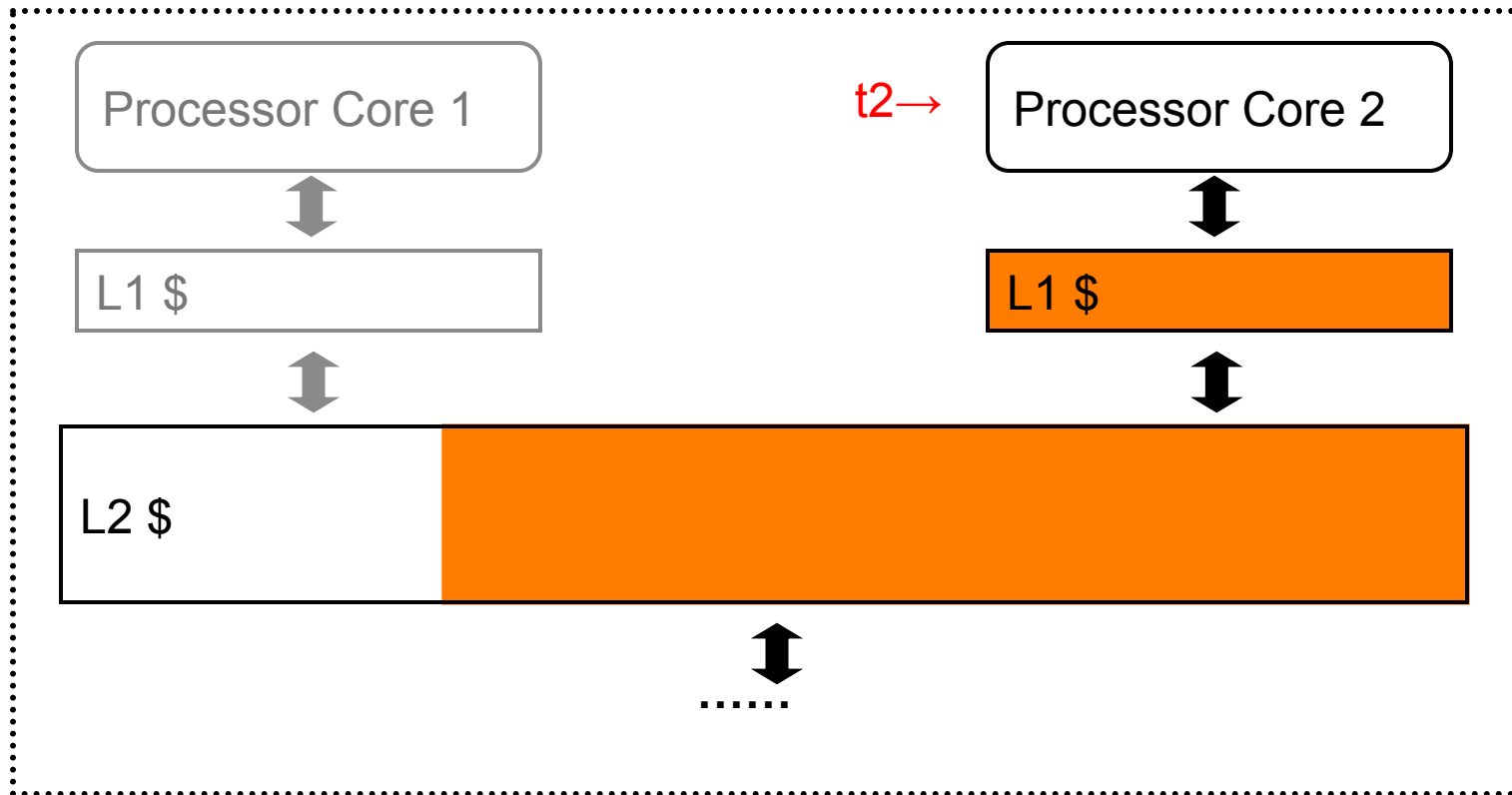
Fair Shared Cache Partitioning

- Goal: Equalize the slowdowns of multiple threads sharing the cache
- Idea: Dynamically estimate slowdowns due to sharing and assign cache blocks to balance slowdowns
- Approximate slowdown with change in miss rate
 - + Simple
 - Not accurate. Why?
- Kim et al., “Fair Cache Sharing and Partitioning in a Chip Multiprocessor Architecture,” PACT 2004.

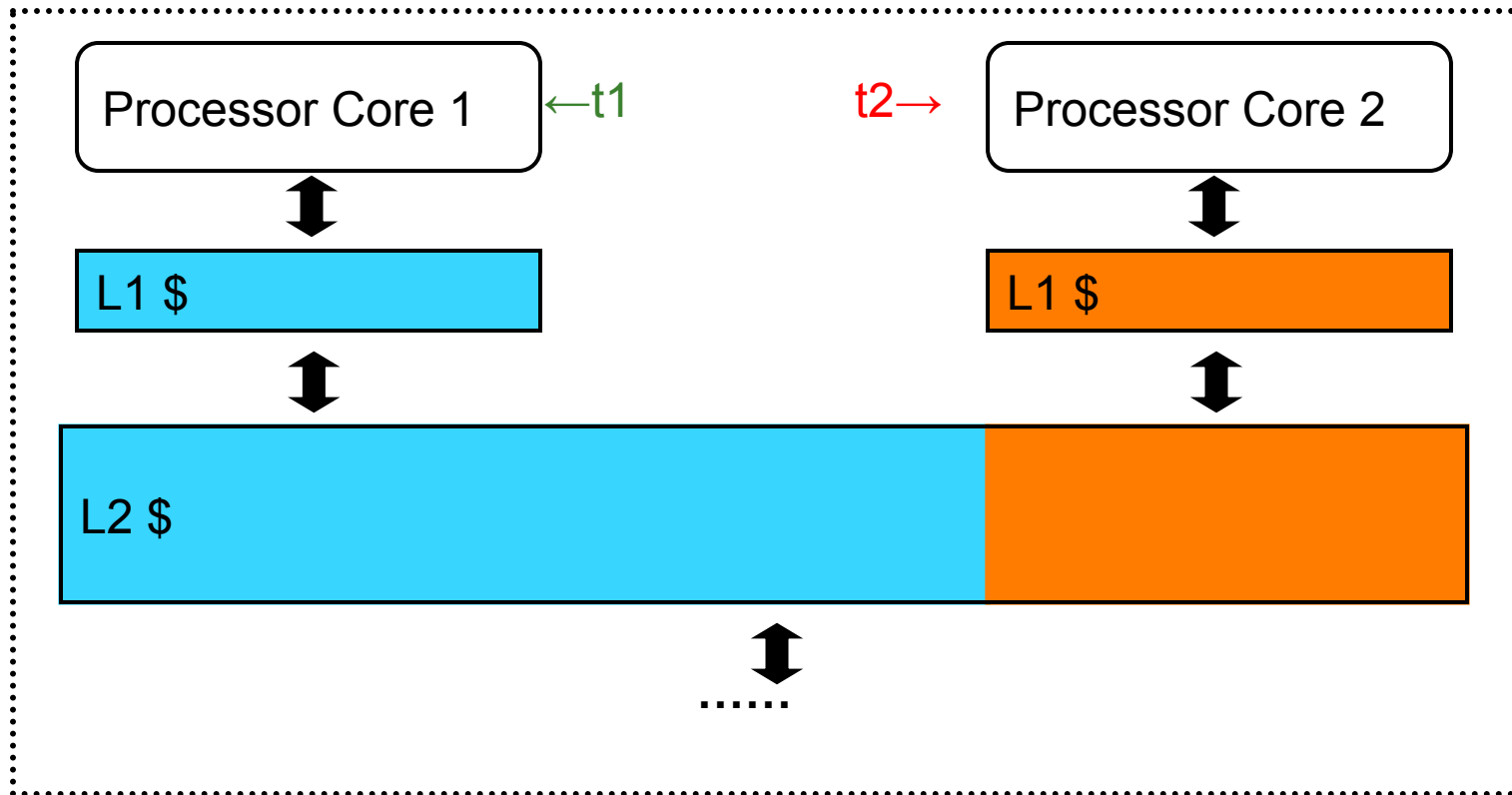
Problem with Shared Caches



Problem with Shared Caches

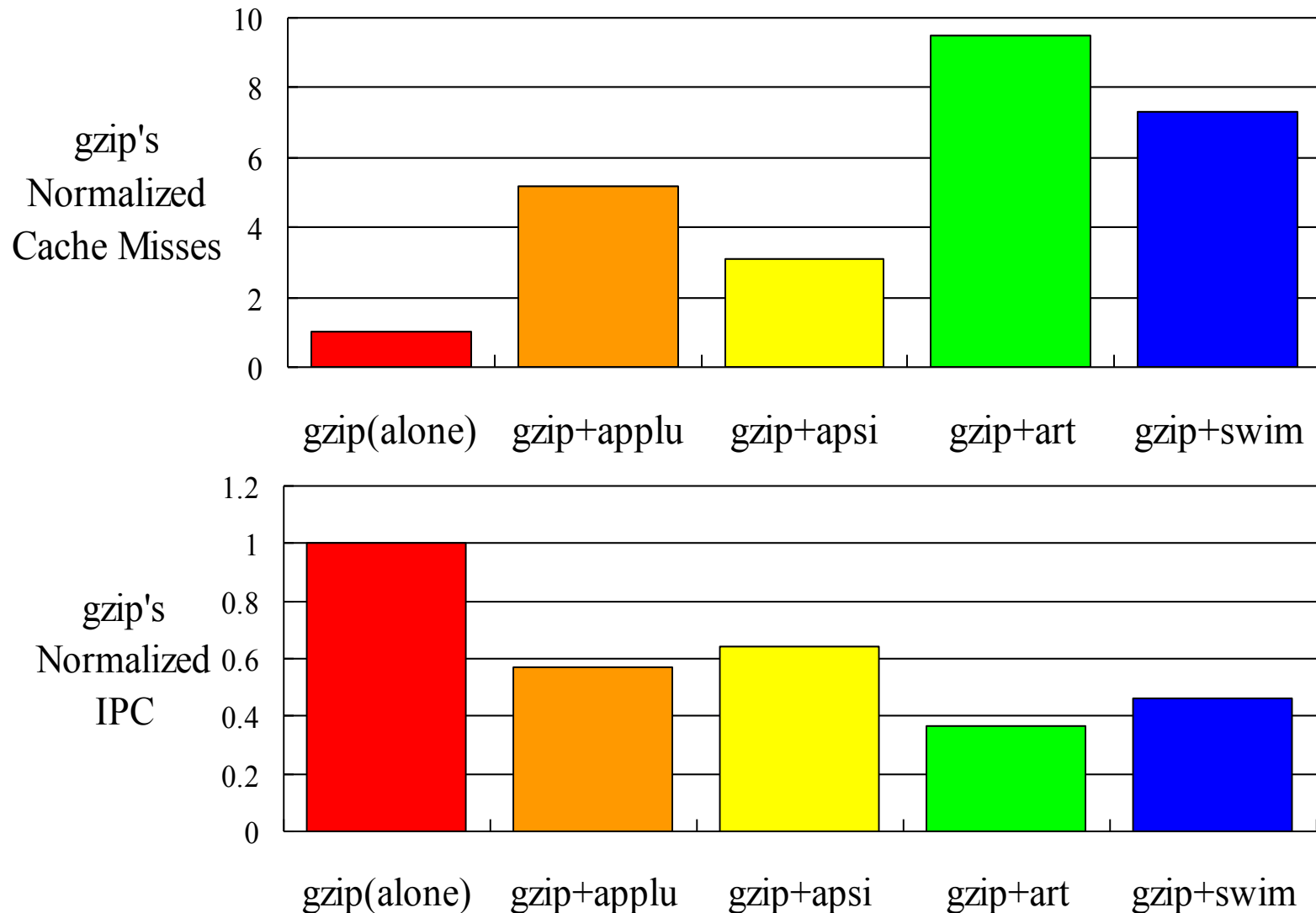


Problem with Shared Caches



$t2$'s throughput is significantly reduced due to unfair cache sharing.

Problem with Shared Caches



Fairness Metrics

- Uniform slowdown

$$\frac{T_shared_i}{T_alone_i} = \frac{T_shared_j}{T_alone_j}$$

- Minimize:

- Ideally:

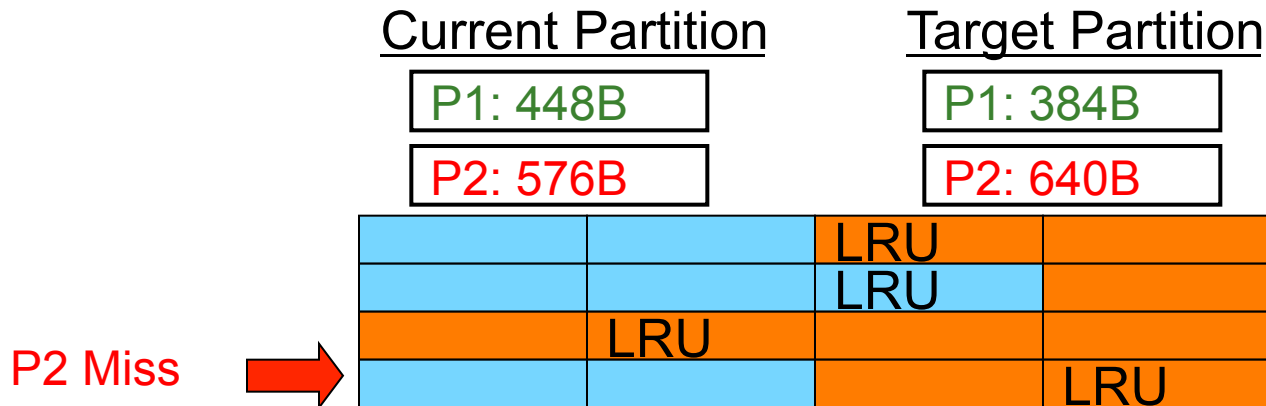
$$M_0^{ij} = |X_i - X_j|, \text{ where } X_i = \frac{T_shared_i}{T_alone_i}$$

$$M_1^{ij} = |X_i - X_j|, \text{ where } X_i = \frac{Miss_shared_i}{Miss_alone_i}$$

$$M_3^{ij} = |X_i - X_j|, \text{ where } X_i = \frac{MissRate_shared_i}{MissRate_alone_i}$$

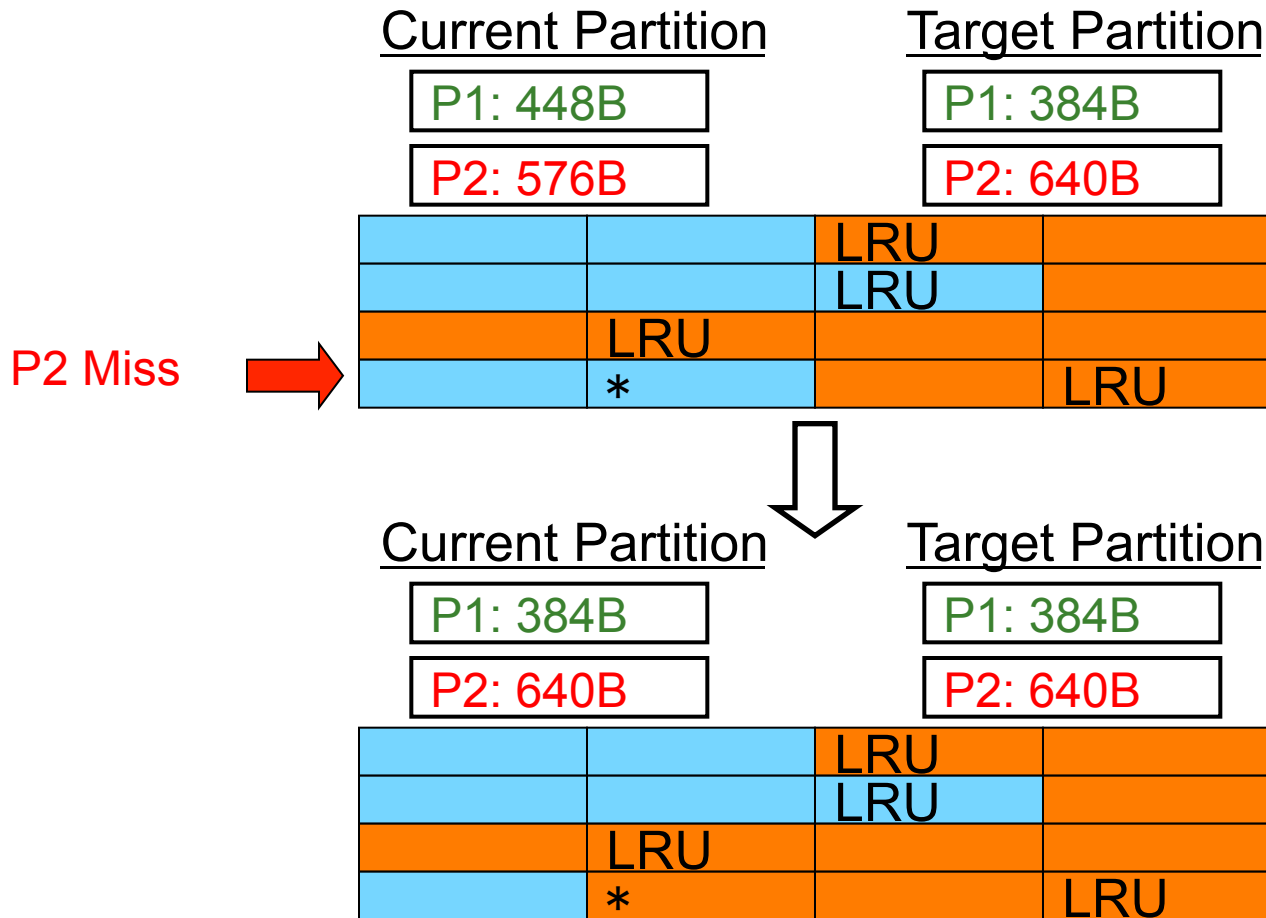
Block-Granularity Partitioning

- Modified LRU cache replacement policy
 - G. Suh, et. al., HPCA 2002



Block-Granularity Partitioning

- Modified LRU cache replacement policy
 - G. Suh, et. al., HPCA 2002



Dynamic Fair Caching Algorithm

Ex) Optimizing
M3 metric

MissRate alone

P1:

P2:

MissRate shared

P1:

P2:

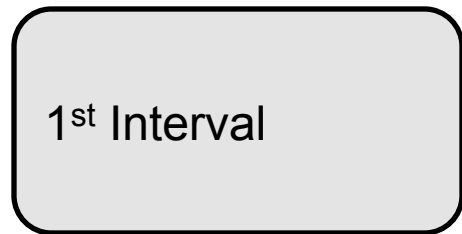


Target Partition

P1:

P2:

Dynamic Fair Caching Algorithm



MissRate alone

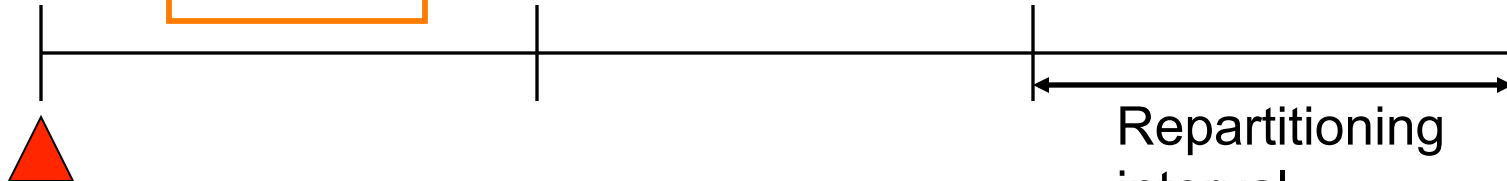
P1:20%

P2: 5%

MissRate shared

P1:20%

P2:15%

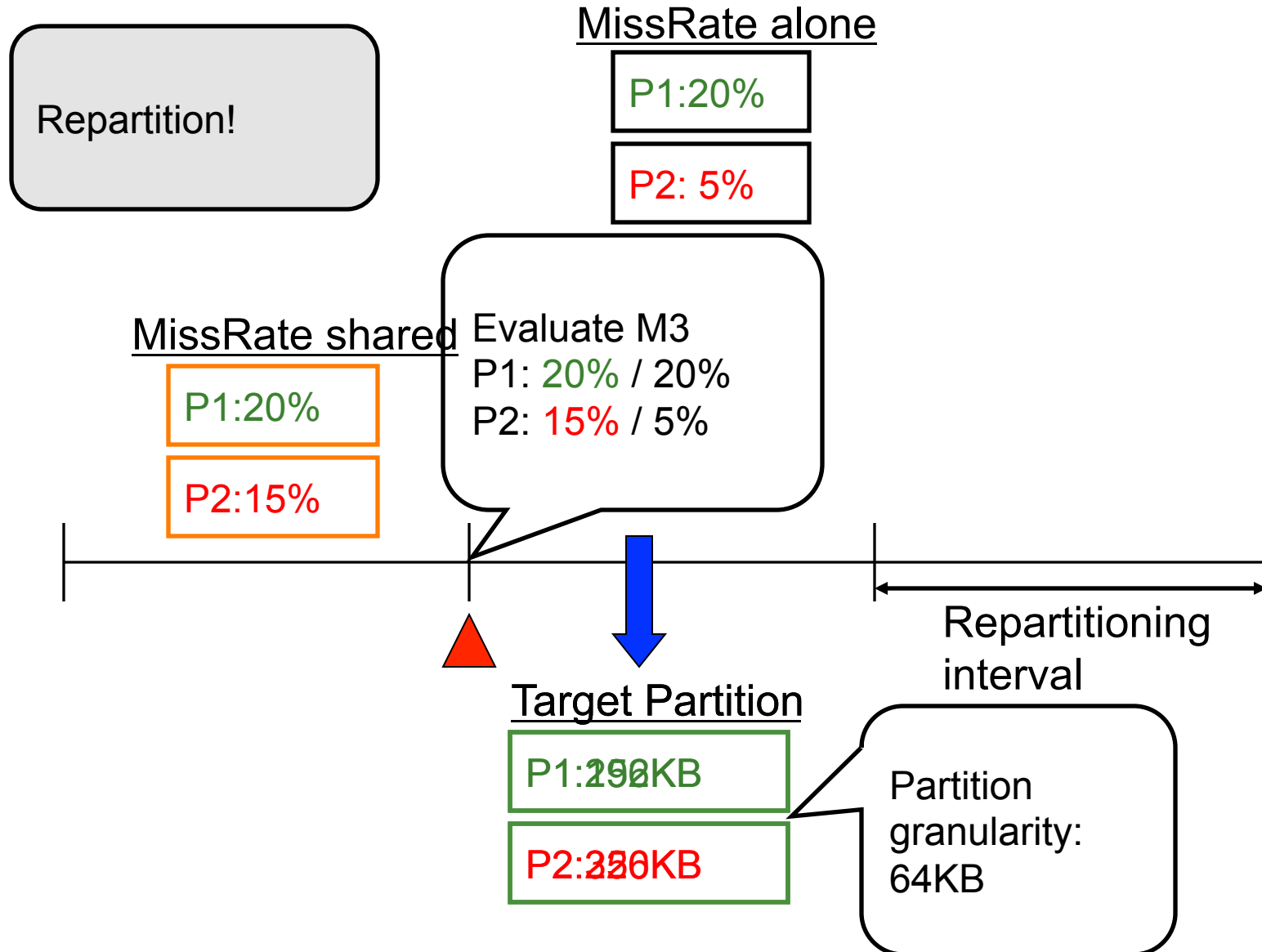


Target Partition

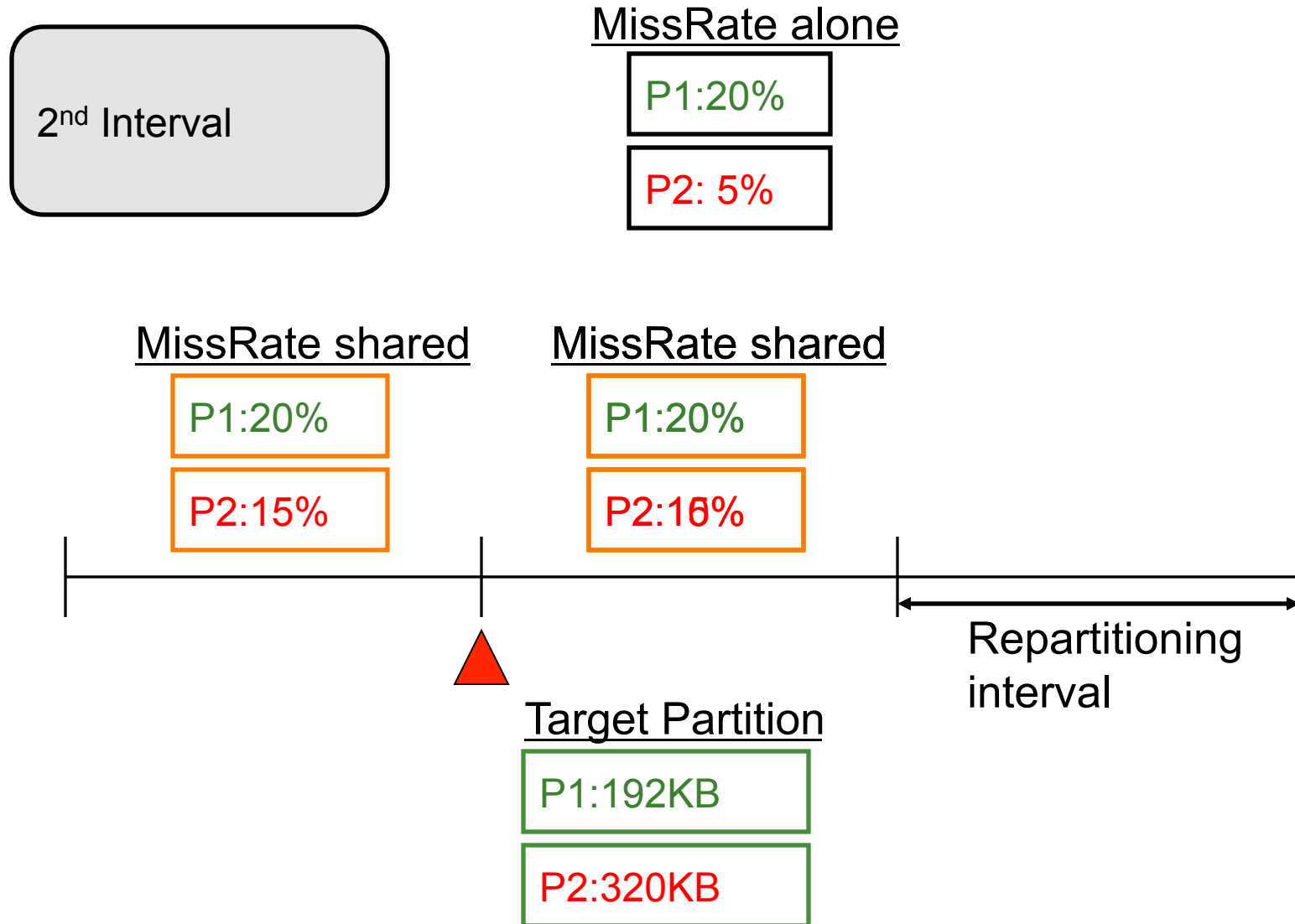
P1:256KB

P2:256KB

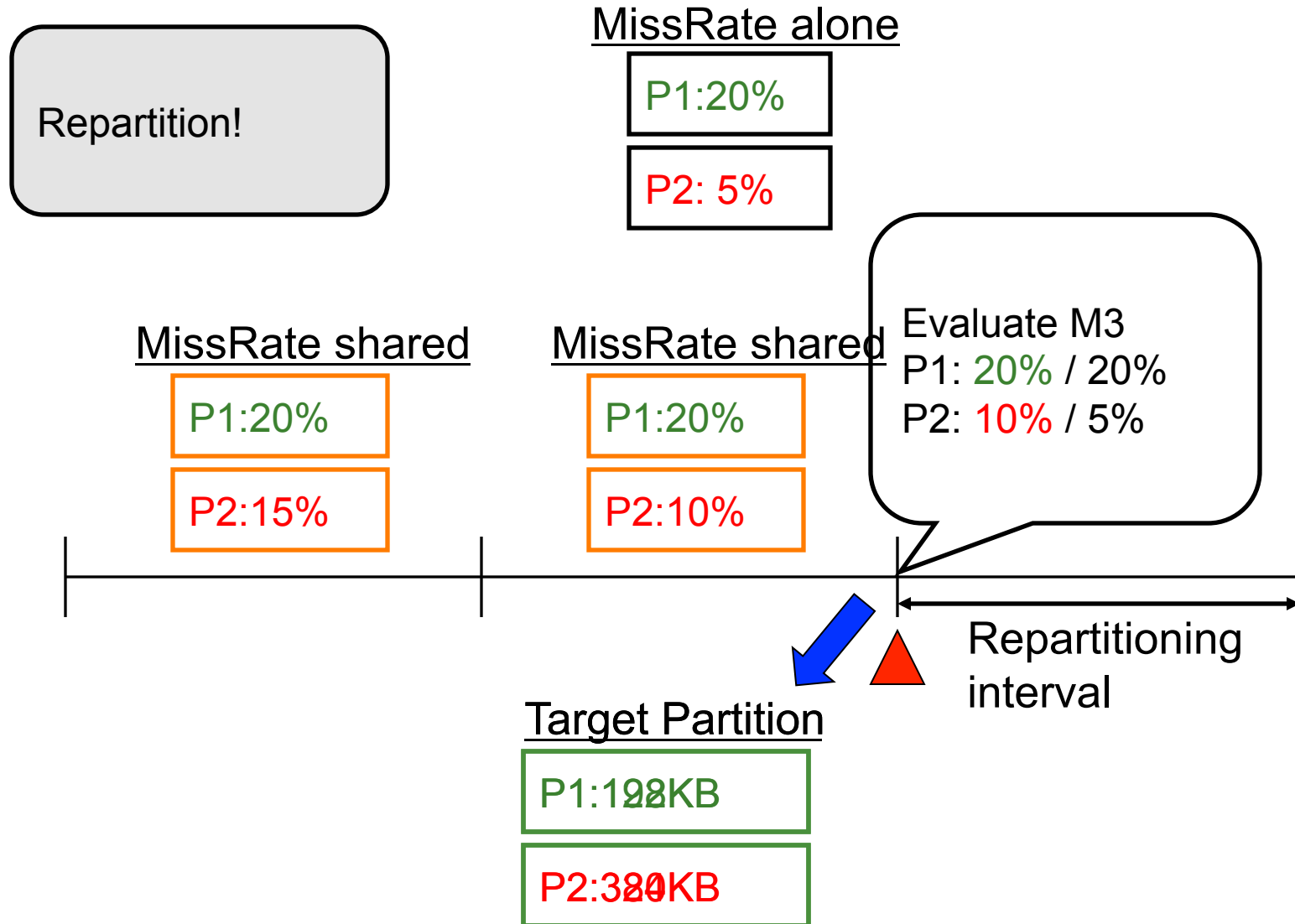
Dynamic Fair Caching Algorithm



Dynamic Fair Caching Algorithm



Dynamic Fair Caching Algorithm



Dynamic Fair Caching Algorithm

3rd Interval

MissRate alone

P1:20%

P2: 5%

MissRate shared

P1:20%

P2:10%

MissRate shared

P1:26%

P2:10%

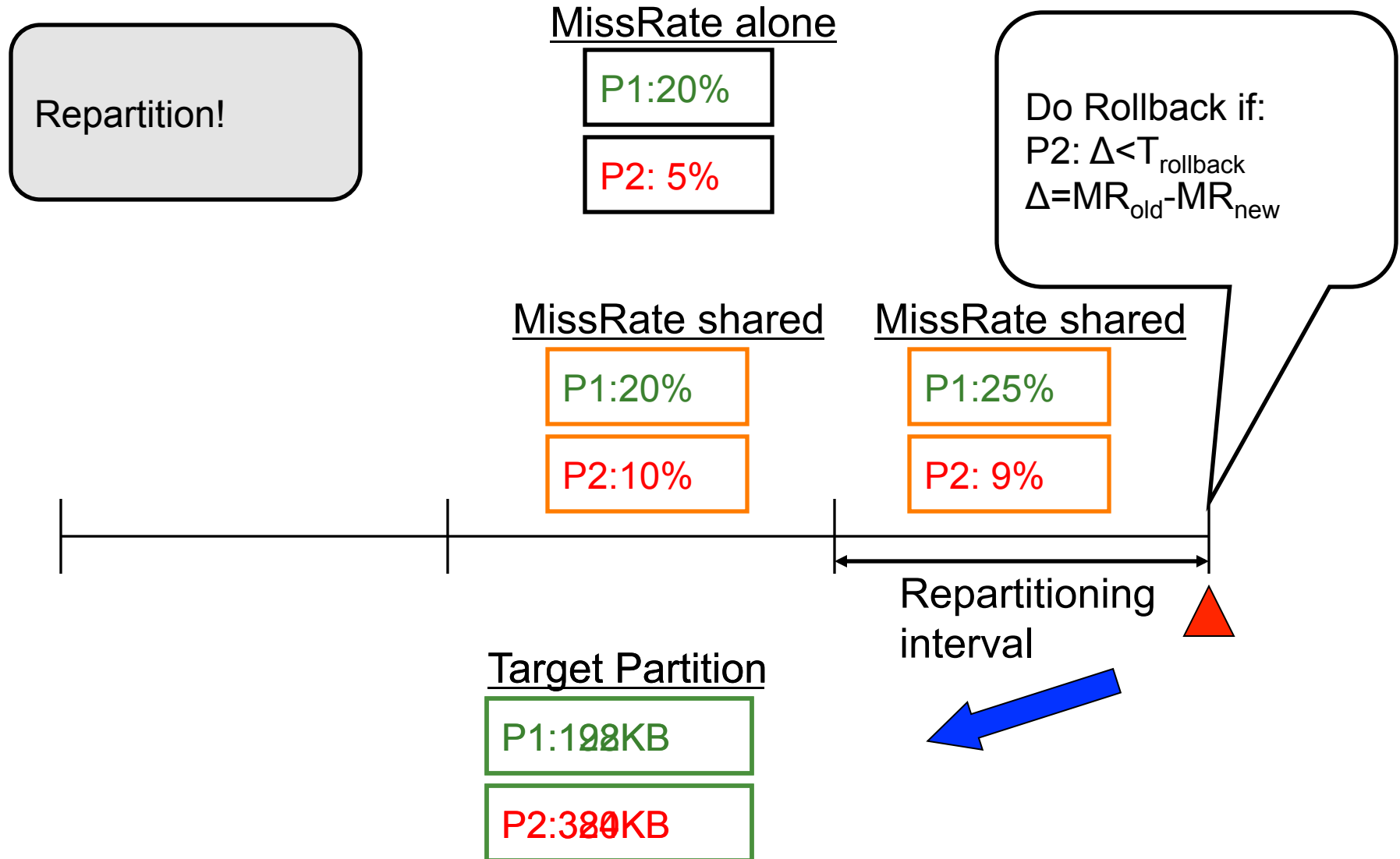
Repartitioning
interval

Target Partition

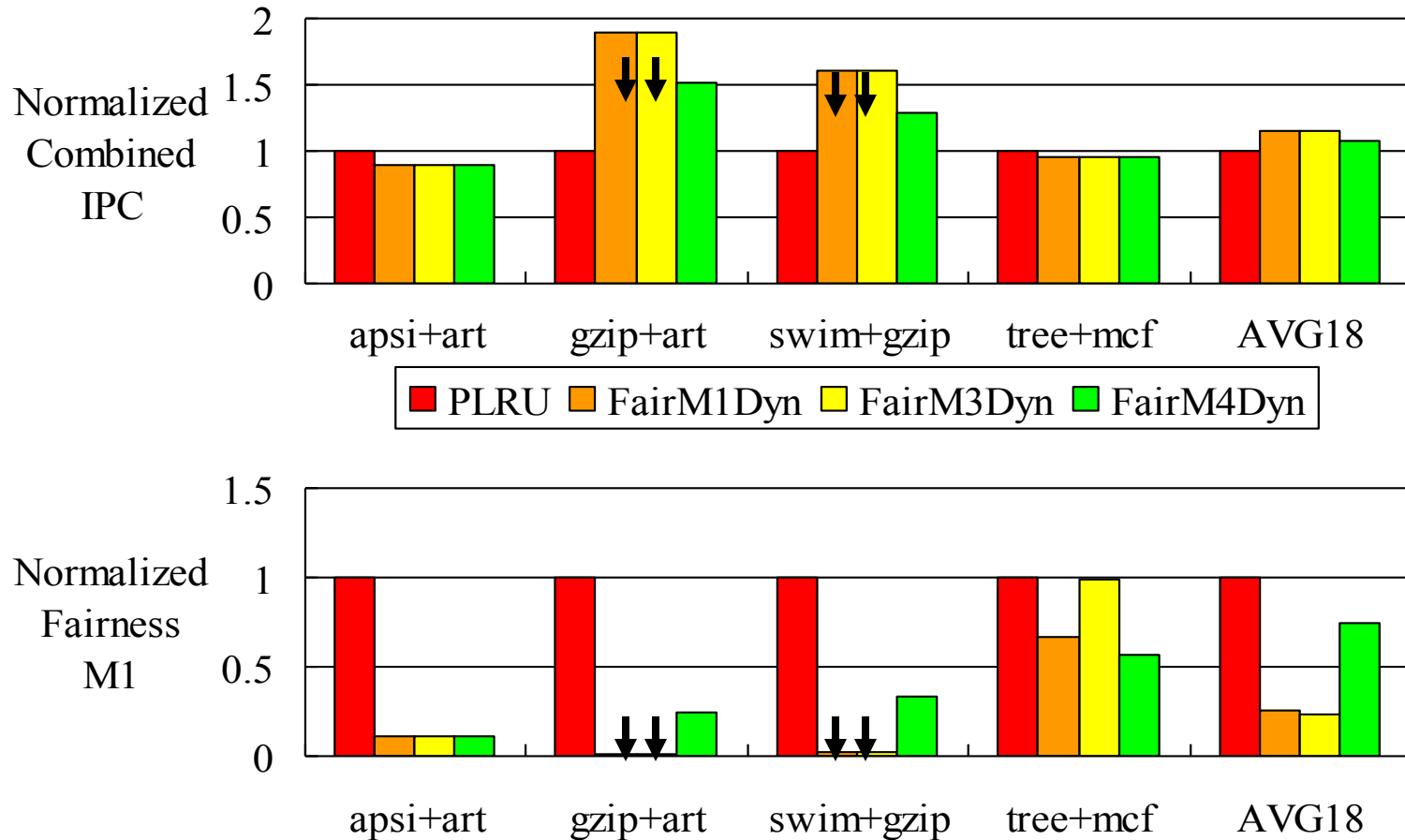
P1:128KB

P2:384KB

Dynamic Fair Caching Algorithm

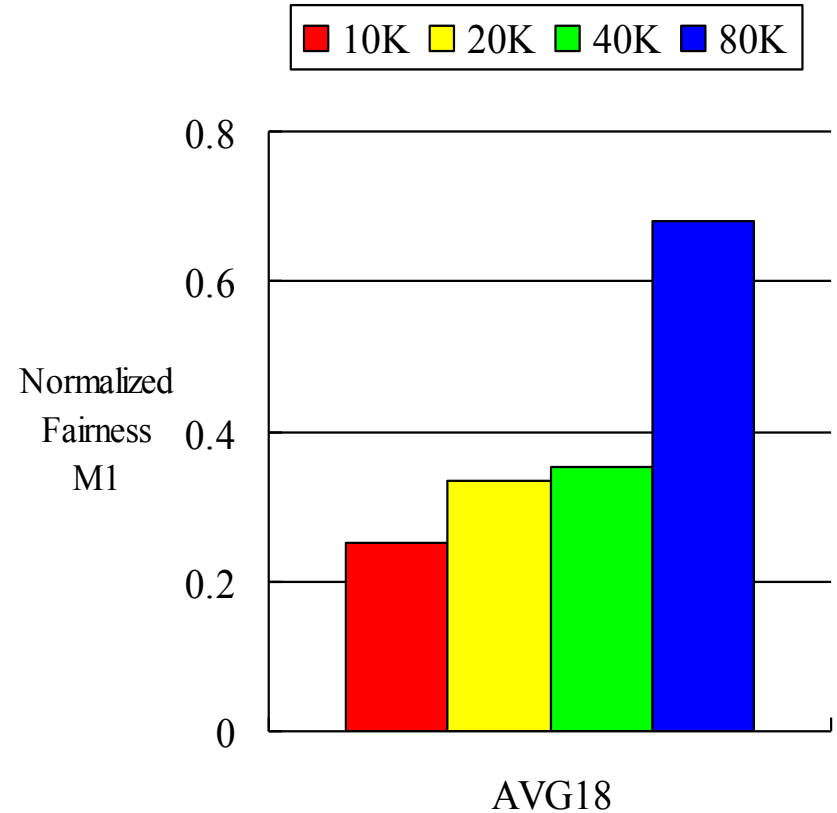
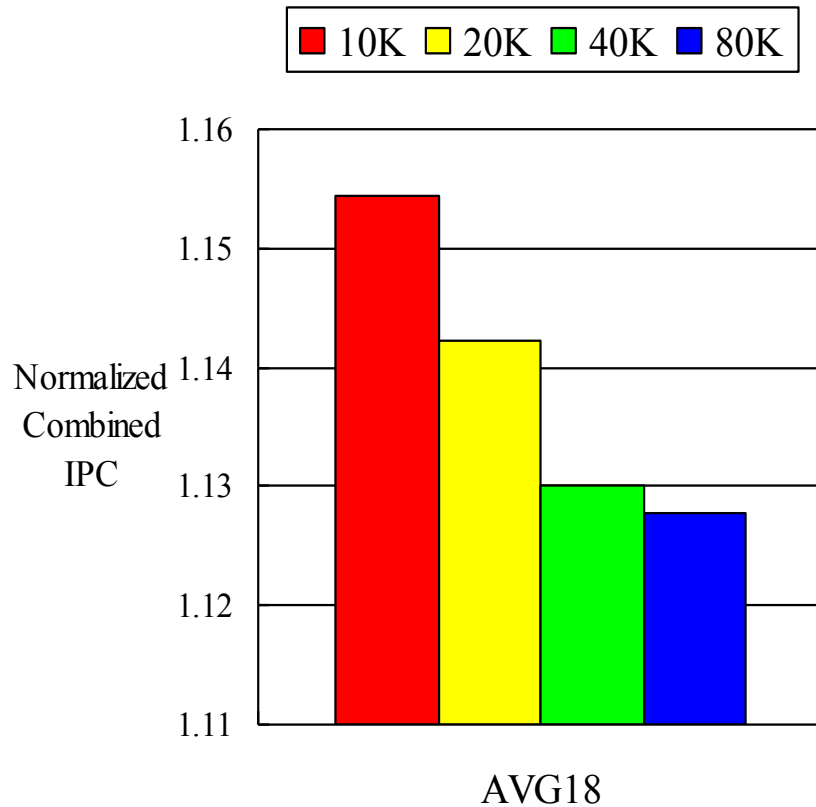


Dynamic Fair Caching Results



- Improves both fairness and throughput

Effect of Partitioning Interval



- Fine-grained partitioning is important for both fairness and throughput

Benefits of Fair Caching

- Problems of unfair cache sharing
 - ❑ Sub-optimal throughput
 - ❑ Thread starvation
 - ❑ Priority inversion
 - ❑ Thread-mix dependent performance

- Benefits of fair caching
 - ❑ Better fairness
 - ❑ Better throughput
 - ❑ Fair caching likely simplifies OS scheduler design

Advantages/Disadvantages of the Approach

■ Advantages

- + No (reduced) starvation
- + Better average throughput

■ Disadvantages

- Scalable to many cores?
- Is this the best (or a good) fairness metric?
- Does this provide performance isolation in cache?
- Alone miss rate estimation can be incorrect (estimation interval different from enforcement interval)

Software-Based Shared Cache Management

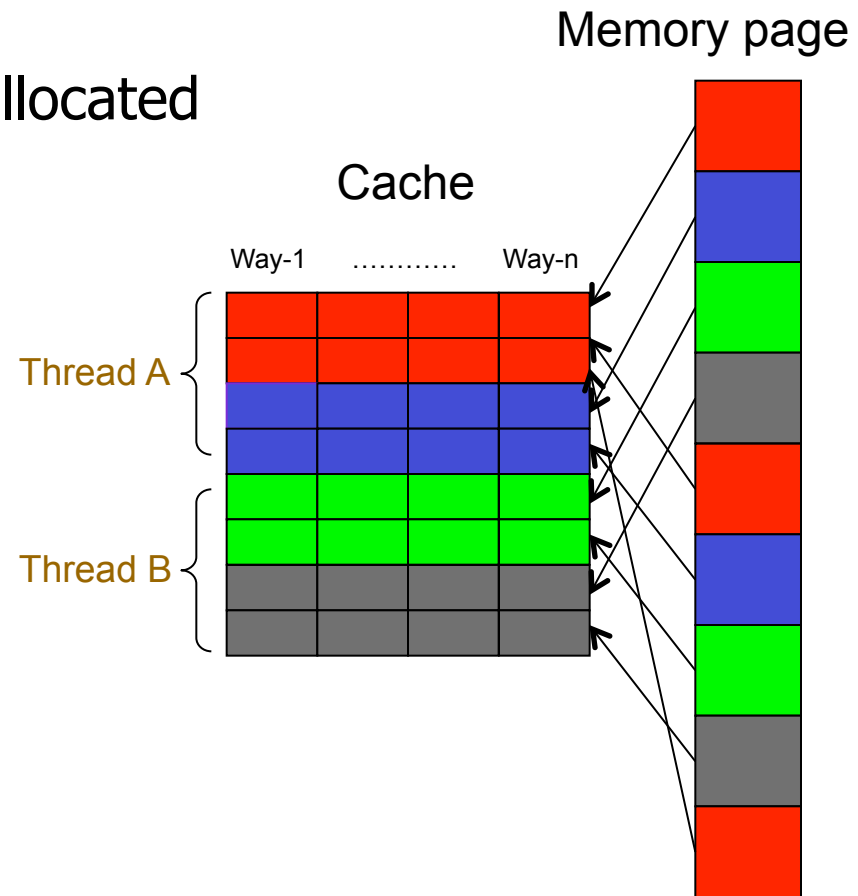
- Assume no hardware support (demand based cache sharing, i.e. LRU replacement)
- How can the OS best utilize the cache?
- Cache sharing aware [thread scheduling](#)
 - Schedule workloads that “play nicely” together in the cache
 - E.g., working sets together fit in the cache
 - Requires static/dynamic profiling of application behavior
 - Fedorova et al., “[Improving Performance Isolation on Chip Multiprocessors via an Operating System Scheduler](#),” PACT 2007.
- Cache sharing aware [page coloring](#)
 - Dynamically monitor miss rate over an interval and change virtual to physical mapping to minimize miss rate
 - Try out different partitions

OS Based Cache Partitioning

- Lin et al., “Gaining Insights into Multi-Core Cache Partitioning: Bridging the Gap between Simulation and Real Systems,” HPCA 2008.
- Cho and Jin, “Managing Distributed, Shared L2 Caches through OS-Level Page Allocation,” MICRO 2006.
- **Static cache partitioning**
 - ❑ Predetermines the amount of cache blocks allocated to each program at the beginning of its execution
 - ❑ Divides shared cache to multiple regions and partitions cache regions through OS page address mapping
- **Dynamic cache partitioning**
 - ❑ Adjusts cache quota among processes dynamically
 - ❑ Page re-coloring
 - ❑ Dynamically changes processes' cache usage through OS page address re-mapping

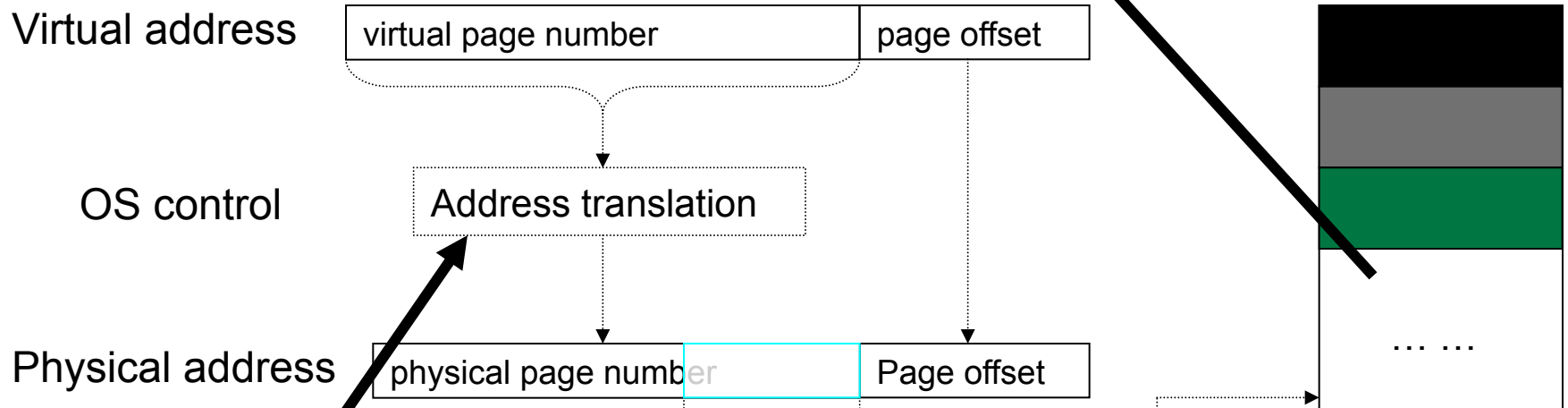
Page Coloring

- Physical memory divided into colors
- Colors map to different cache sets
- Cache partitioning
 - Ensure two threads are allocated pages of different colors

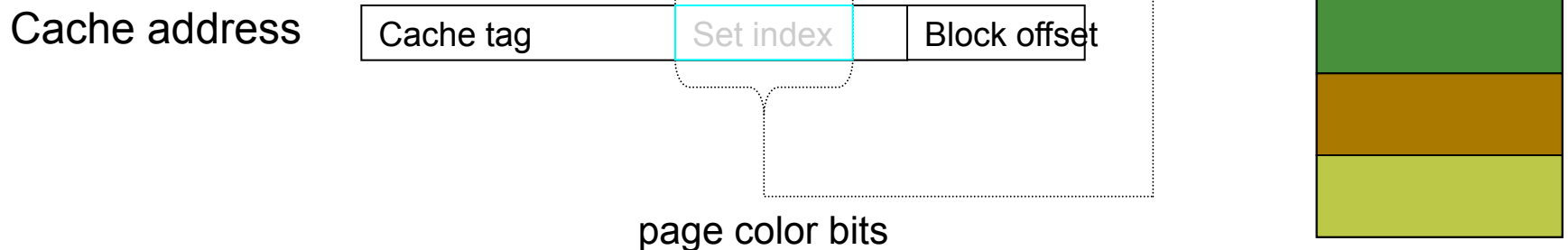


Page Coloring

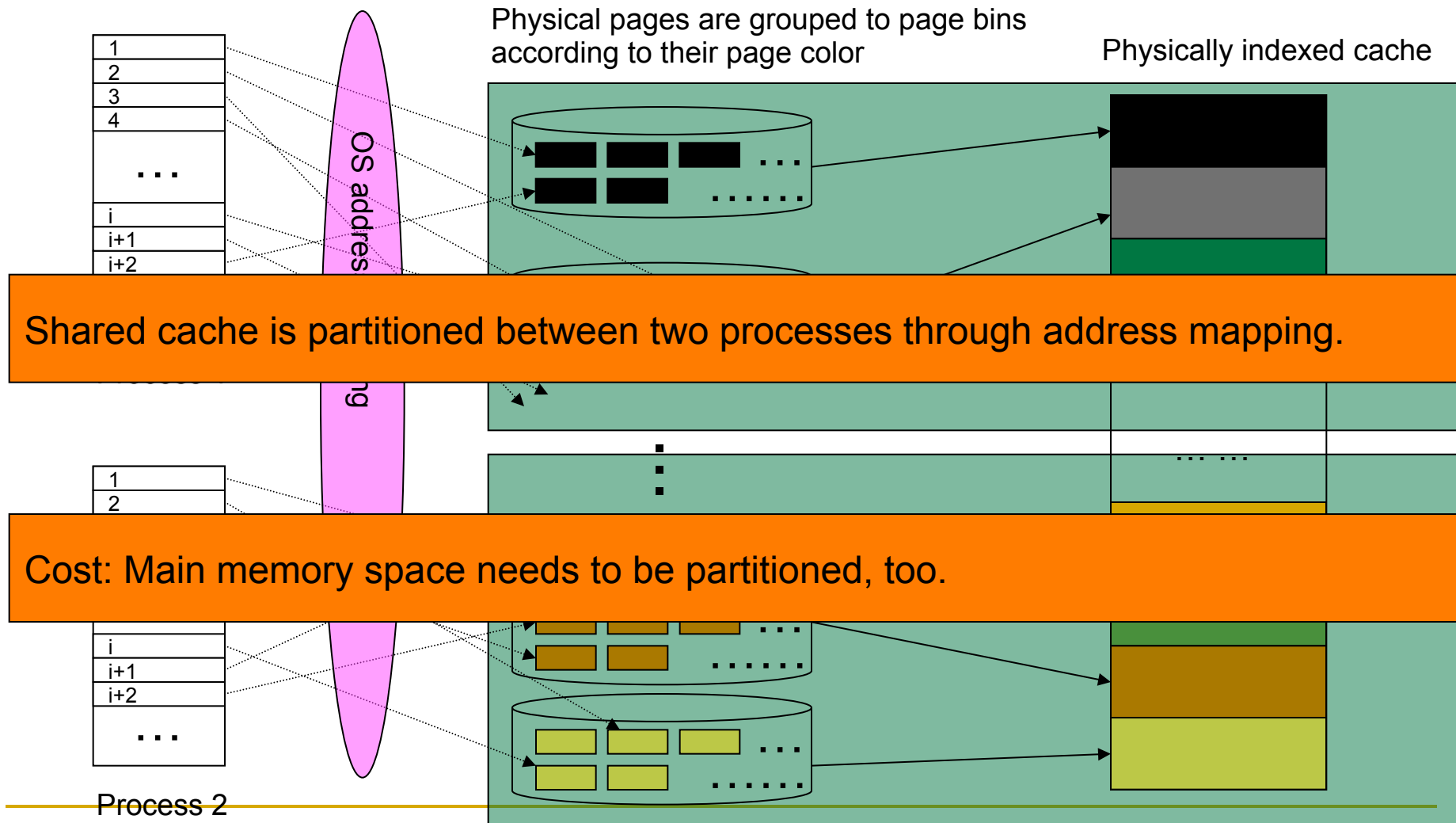
- Physically indexed caches are divided into multiple regions (colors).
- All cache lines in a physical page are cached in one of those regions (colors).



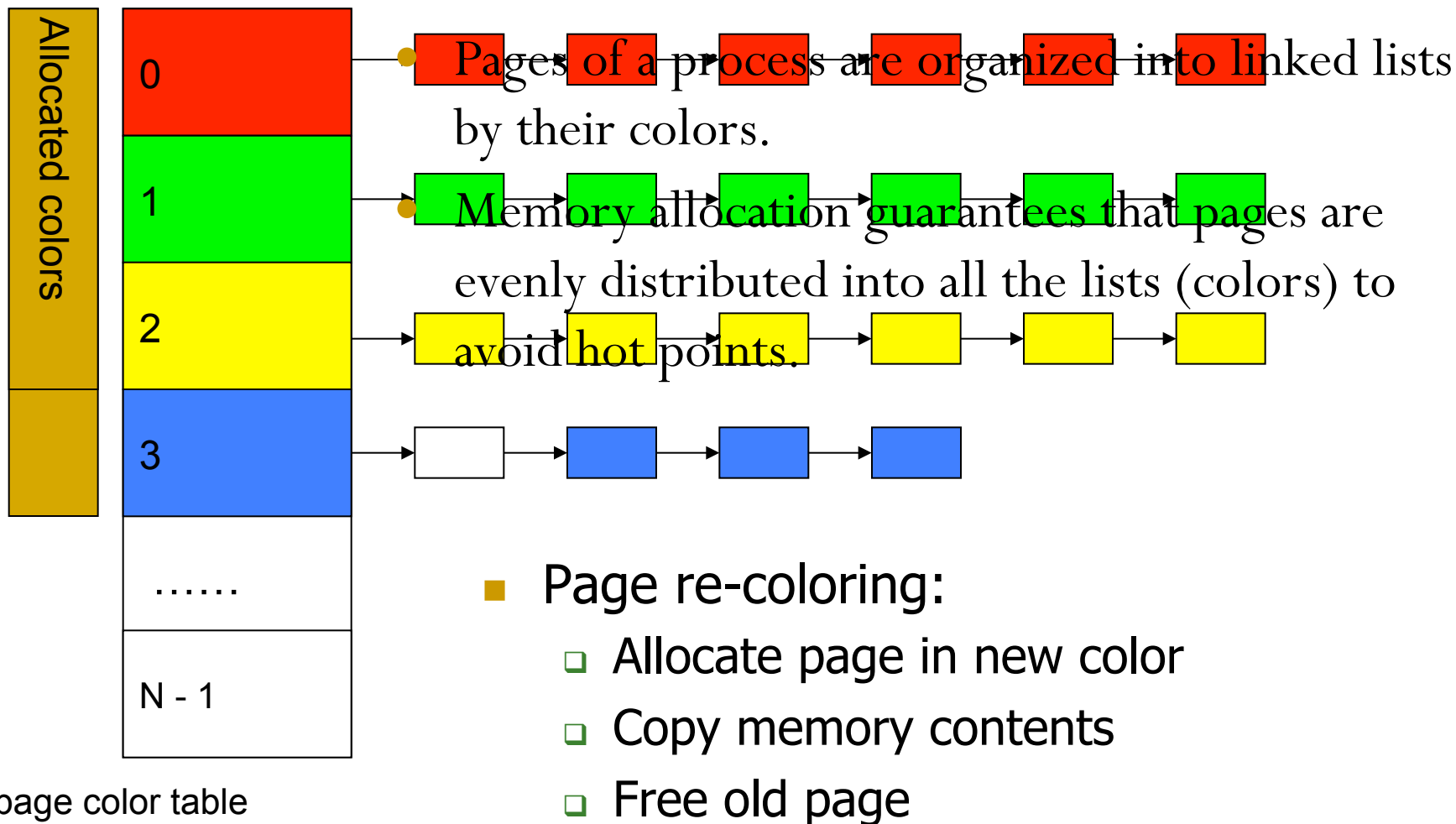
OS can control the page color of a virtual page through address mapping (by selecting a physical page with a specific value in its page color bits).



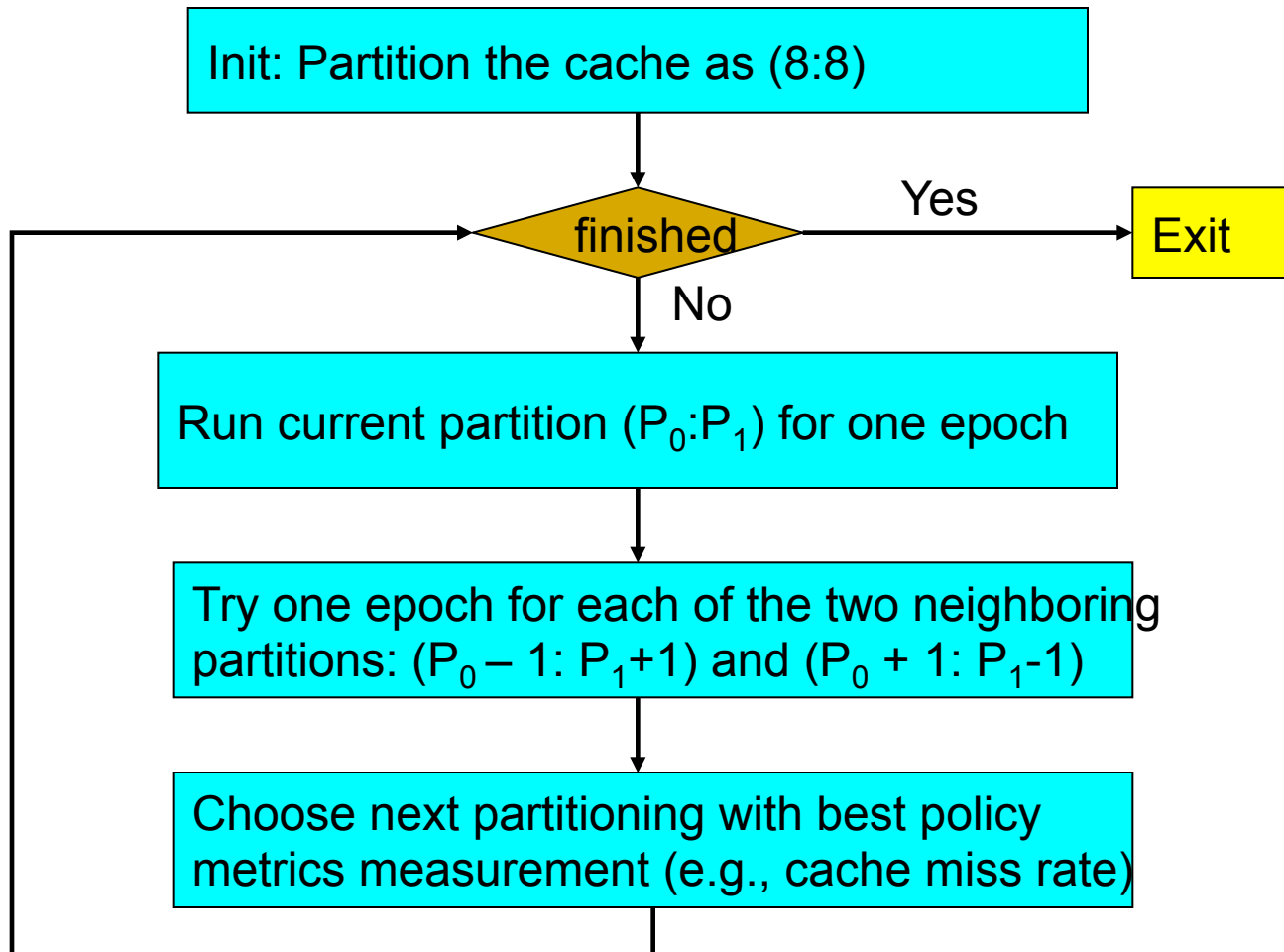
Static Cache Partitioning using Page Coloring



Dynamic Cache Partitioning via Page Re-Coloring



Dynamic Partitioning in Dual Core

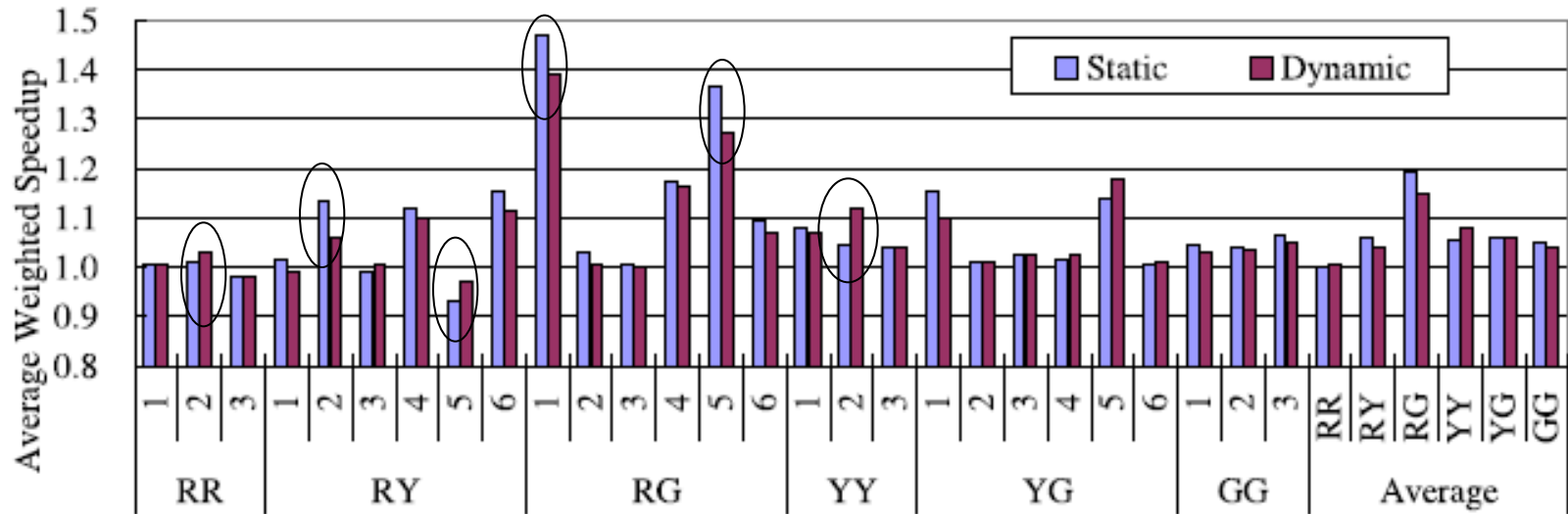


Experimental Environment

- Dell PowerEdge1950
 - ❑ Two-way SMP, Intel dual-core Xeon 5160
 - ❑ Shared 4MB L2 cache, 16-way
 - ❑ 8GB Fully Buffered DIMM

 - Red Hat Enterprise Linux 4.0
 - ❑ 2.6.20.3 kernel
 - ❑ Performance counter tools from HP (Pfmon)
 - ❑ Divide L2 cache into 16 colors
-

Performance – Static & Dynamic



- Aim to minimize combined miss rate
- For RG-type, and some RY-type:
 - Static partitioning outperforms dynamic partitioning
- For RR- and RY-type, and some RY-type
 - Dynamic partitioning outperforms static partitioning

Software vs. Hardware Cache Management

- Software advantages
 - + No need to change hardware
 - + Easier to upgrade/change algorithm (not burned into hardware)

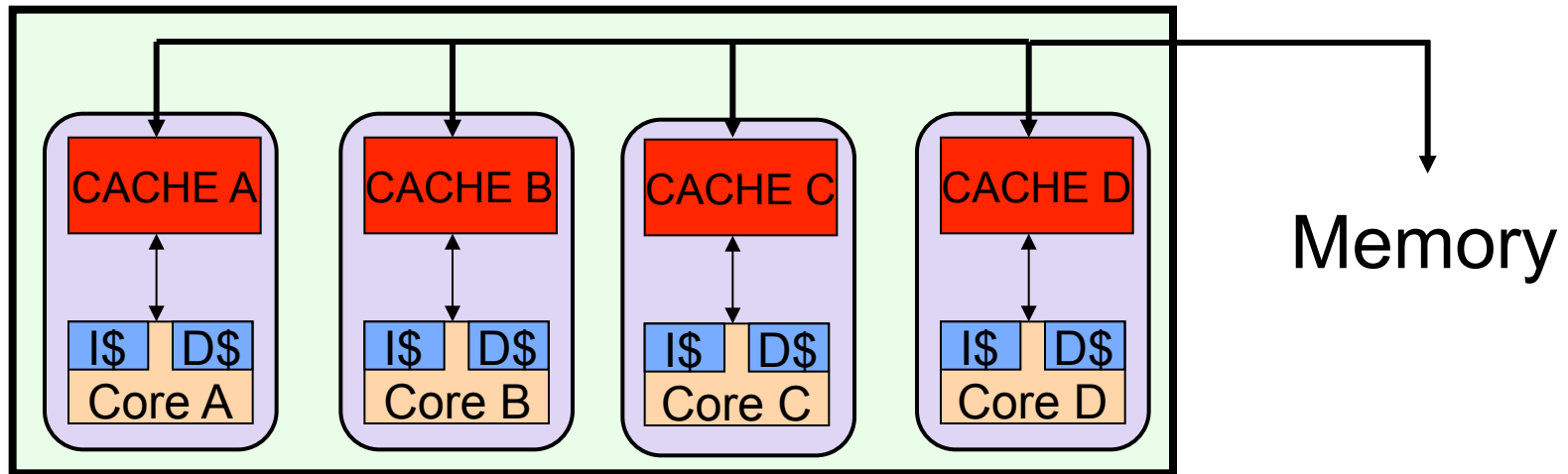
- Disadvantages
 - Less flexible: large granularity (page-based instead of way/block)
 - Limited page colors → reduced performance per application (limited physical memory space!), reduced flexibility
 - Changing partition size has high overhead → page mapping changes
 - Adaptivity is slow: hardware can adapt every cycle (possibly)
 - Not enough information exposed to software (e.g., number of misses due to inter-thread conflict)

Private/Shared Caching

- Example: Adaptive spill/receive caching
- Goal: Achieve the benefits of private caches (low latency, performance isolation) while sharing cache capacity across cores
- Idea: Start with a private cache design (for performance isolation), but dynamically steal space from other cores that do not need all their private caches
 - Some caches can spill their data to other cores' caches dynamically
- Qureshi, “Adaptive Spill-Receive for Robust High-Performance Caching in CMPs,” HPCA 2009.

Revisiting Private Caches on CMP

Private caches avoid the need for shared interconnect
++ fast latency, tiled design, performance isolation

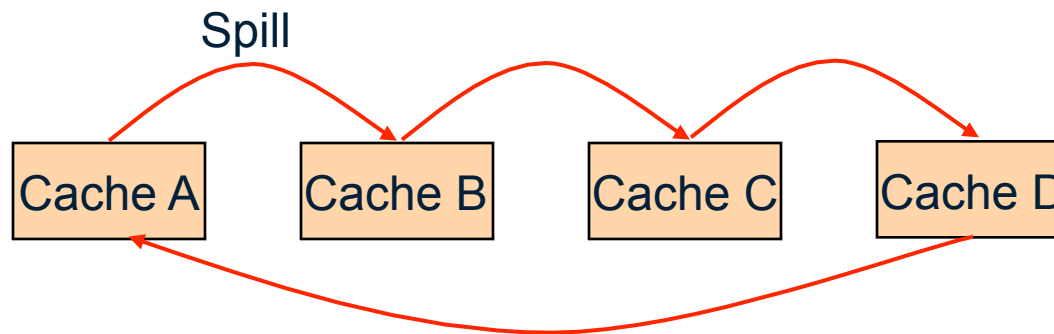


Problem: When one core needs more cache and other core has spare cache, private-cache CMPs cannot share capacity

Cache Line Spilling

Spill evicted line from one cache to neighbor cache

- Co-operative caching (CC) [Chang+ ISCA' 06]



Problem with CC:

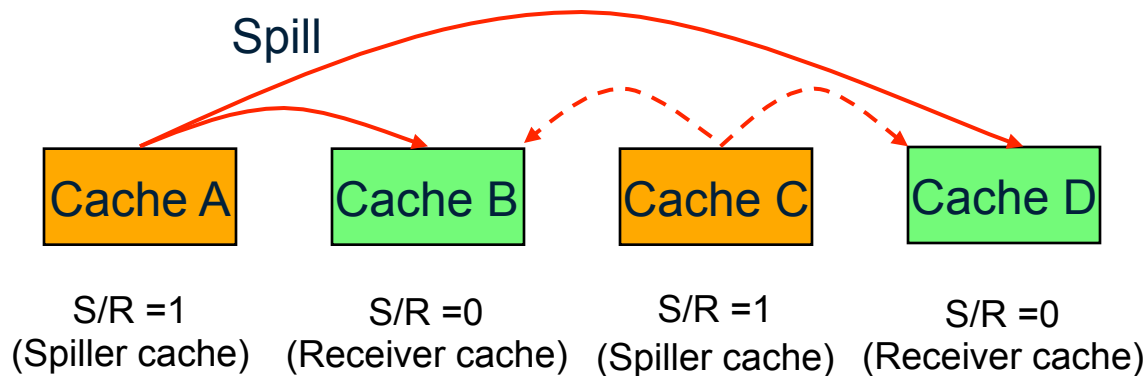
1. Performance depends on the parameter (spill probability)
2. All caches spill as well as receive → Limited improvement

Goal: Robust High-Performance Capacity Sharing with Negligible Overhead

Spill-Receive Architecture

Each Cache is either a Spiller or Receiver but not both

- Lines from spiller cache are spilled to one of the receivers
- Evicted lines from receiver cache are discarded



What is the best N-bit binary string that maximizes the performance of Spill Receive Architecture → Dynamic Spill Receive (DSR)

Dynamic Spill-Receive via “Set Dueling”

Divide the cache in three:

- Spiller sets
- Receiver sets
- Follower sets (winner of spiller, receiver)

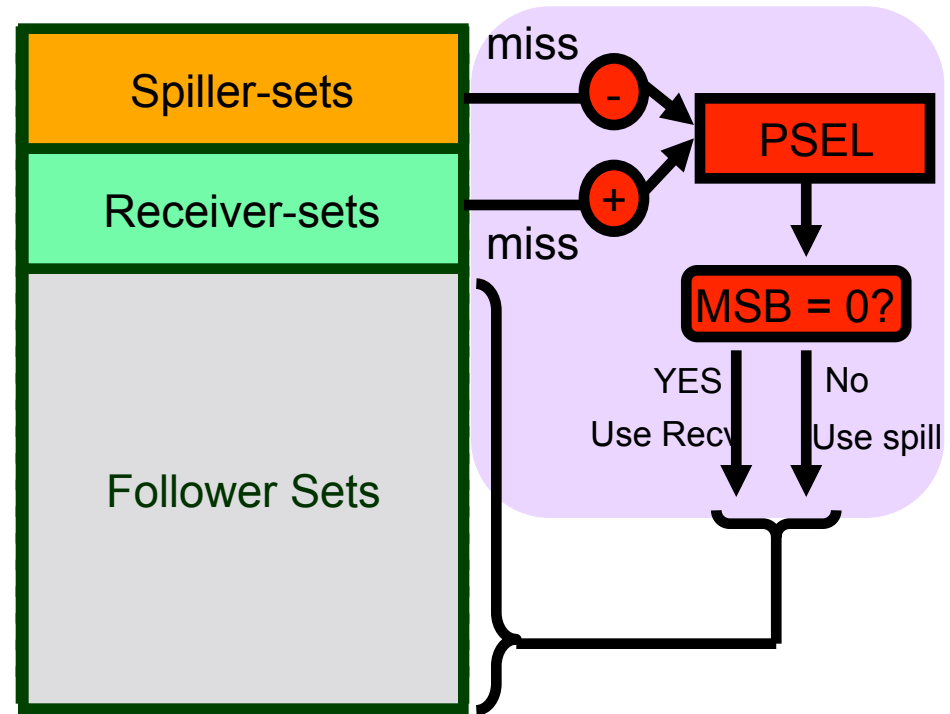
n-bit PSEL counter

misses to spiller-sets: PSEL--

misses to receiver-set: PSEL++

MSB of PSEL decides policy for Follower sets:

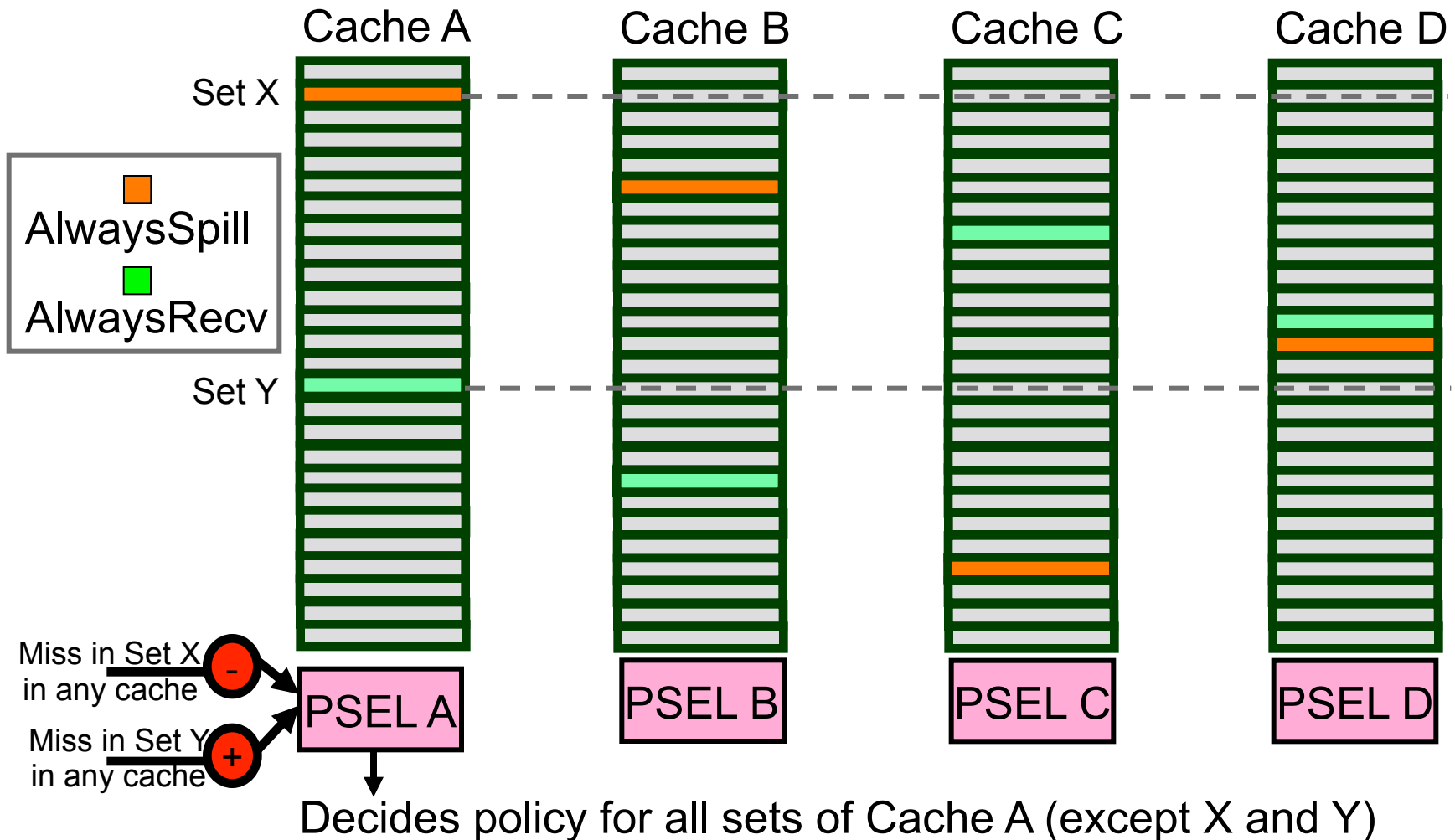
- MSB = 0, Use spill
- MSB = 1, Use receive



monitor → choose → apply
(using a single counter)

Dynamic Spill-Receive Architecture

Each cache learns whether it should act as a spiller or receiver



Experimental Setup

□ Baseline Study:

- 4-core CMP with in-order cores
- Private Cache Hierarchy: 16KB L1, 1MB L2
- 10 cycle latency for local hits, 40 cycles for remote hits

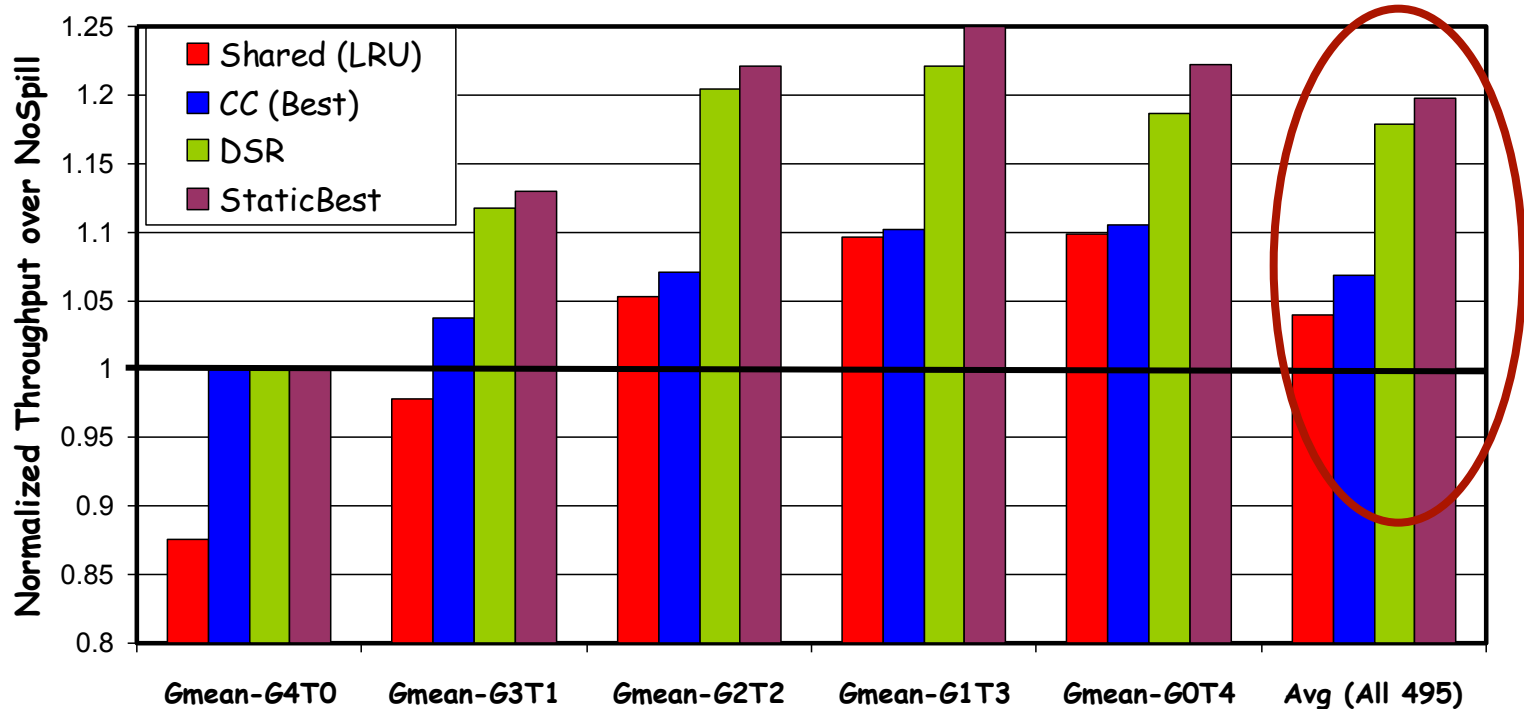
□ Benchmarks:

- 6 benchmarks that have extra cache: “Givers” (G)
- 6 benchmarks that benefit from more cache: “Takers” (T)
- All 4-thread combinations of 12 benchmarks: 495 total

Five types of workloads:



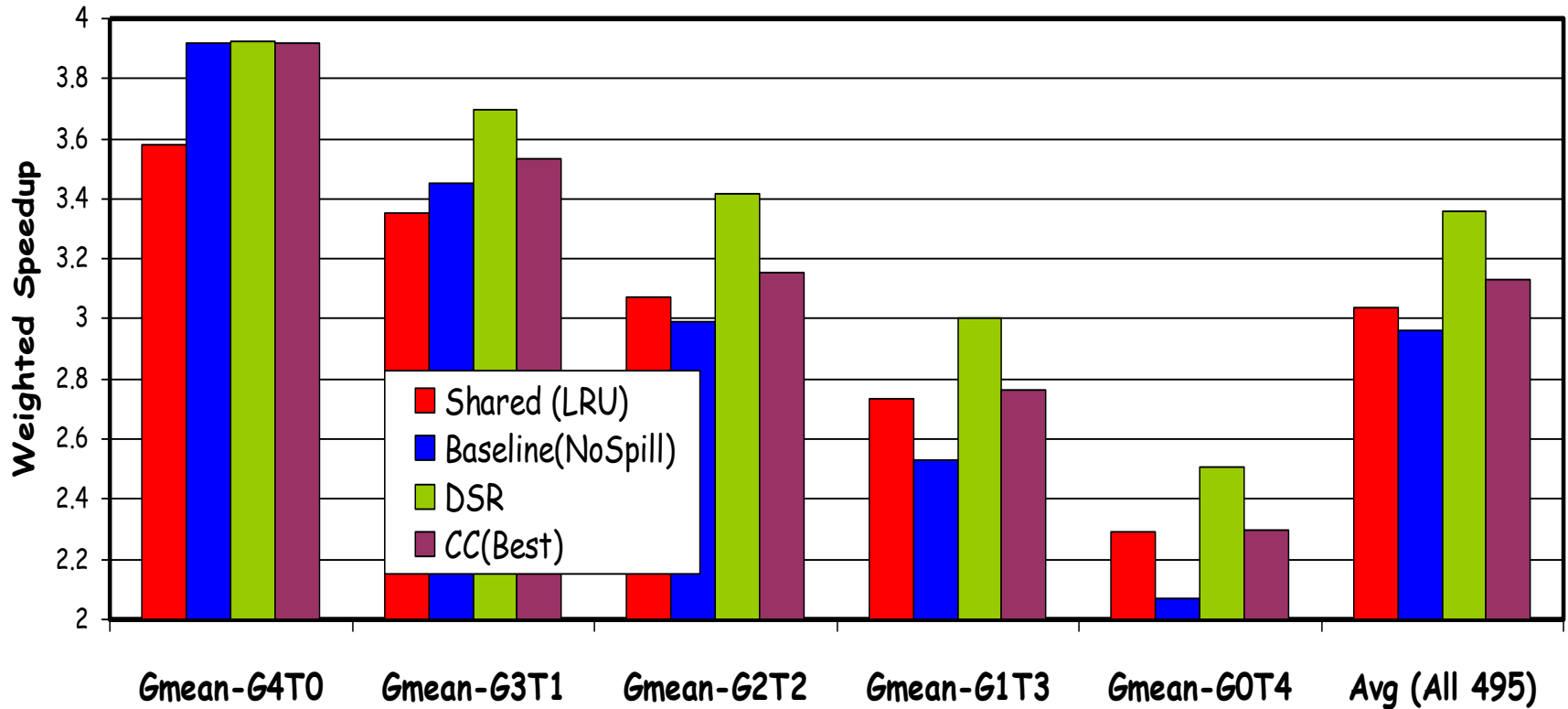
Results for Throughput



On average, DSR improves throughput by 18%, co-operative caching by 7%
DSR provides 90% of the benefit of knowing the best decisions a priori

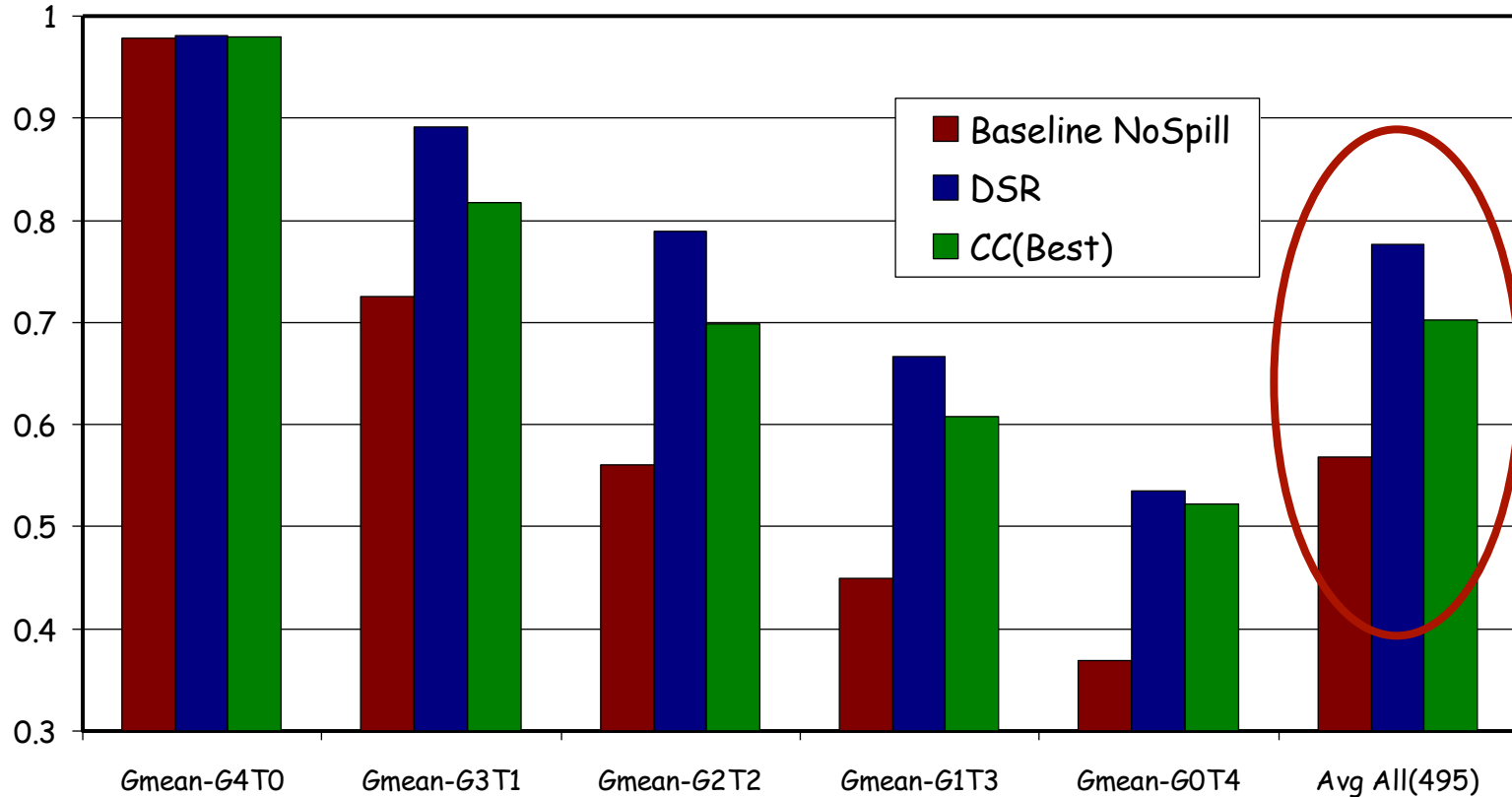
* DSR implemented with 32 dedicated sets and 10 bit PSEL counters

Results for Weighted Speedup



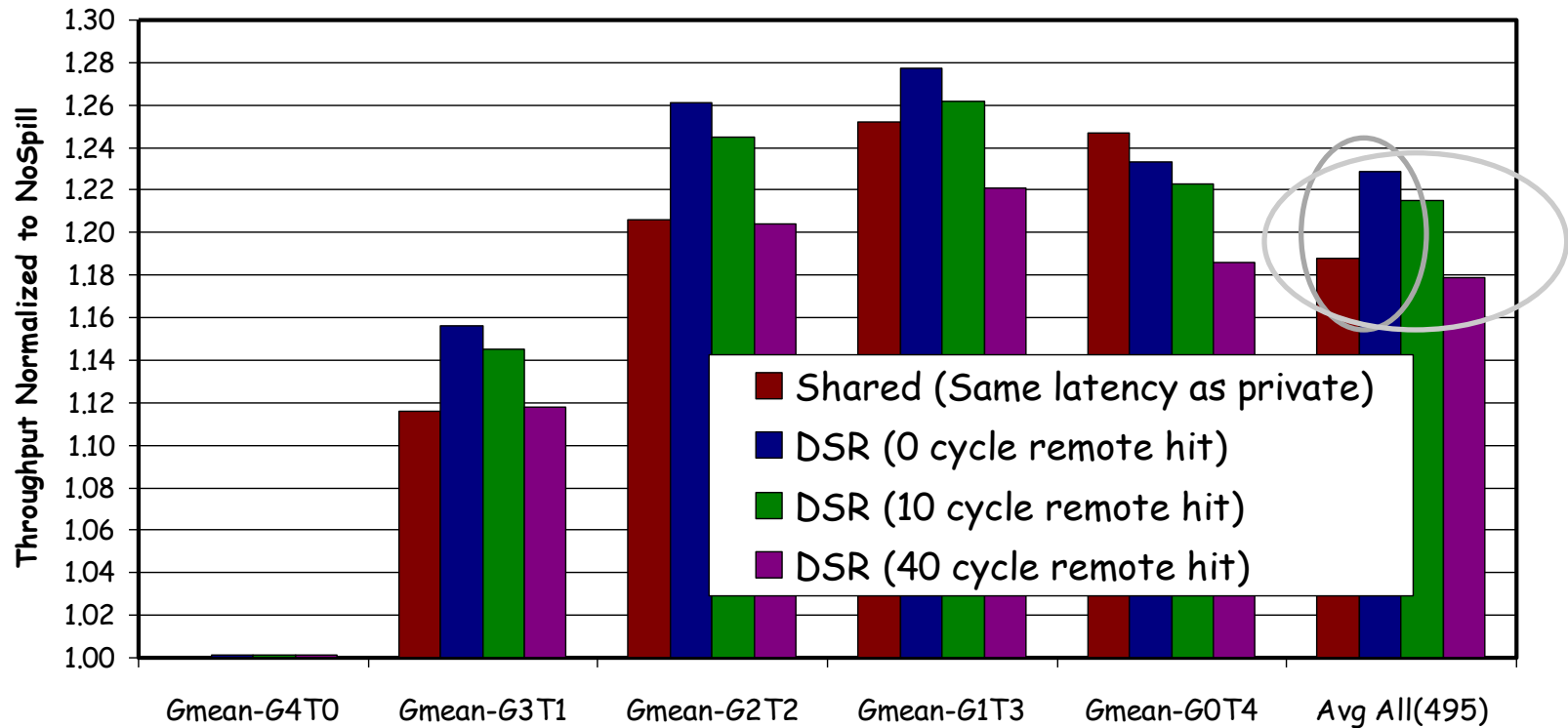
On average, DSR improves weighted speedup by 13%

Results for Hmean Speedup



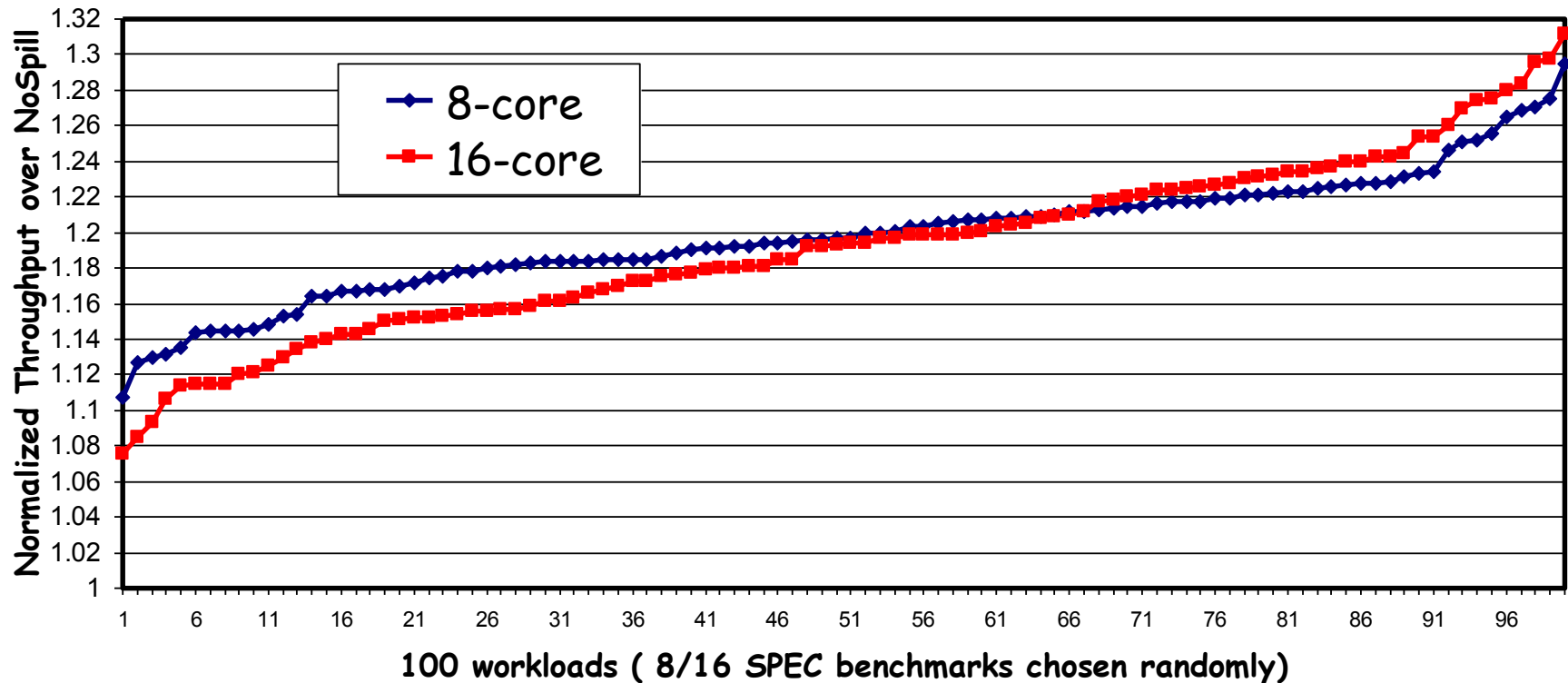
On average, DSR improves Hmean Fairness from 0.58 to 0.78

DSR vs. Faster Shared Cache



DSR (with 40 cycle extra for remote hits) performs similar to shared cache with zero latency overhead and crossbar interconnect

Scalability of DSR



DSR improves average throughput by 19% for both systems
(No performance degradation for any of the workloads)

Quality of Service with DSR

For 1 % of the $495 \times 4 = 1980$ apps, DSR causes IPC loss of $> 5\%$

In some cases, important to ensure that performance does not degrade compared to dedicated private cache → QoS

DSR can ensure QoS: change PSEL counters by weight of miss:

$$\Delta\text{Miss} = \text{MissesWithDSR} - \text{MissesWithNoSpill}$$

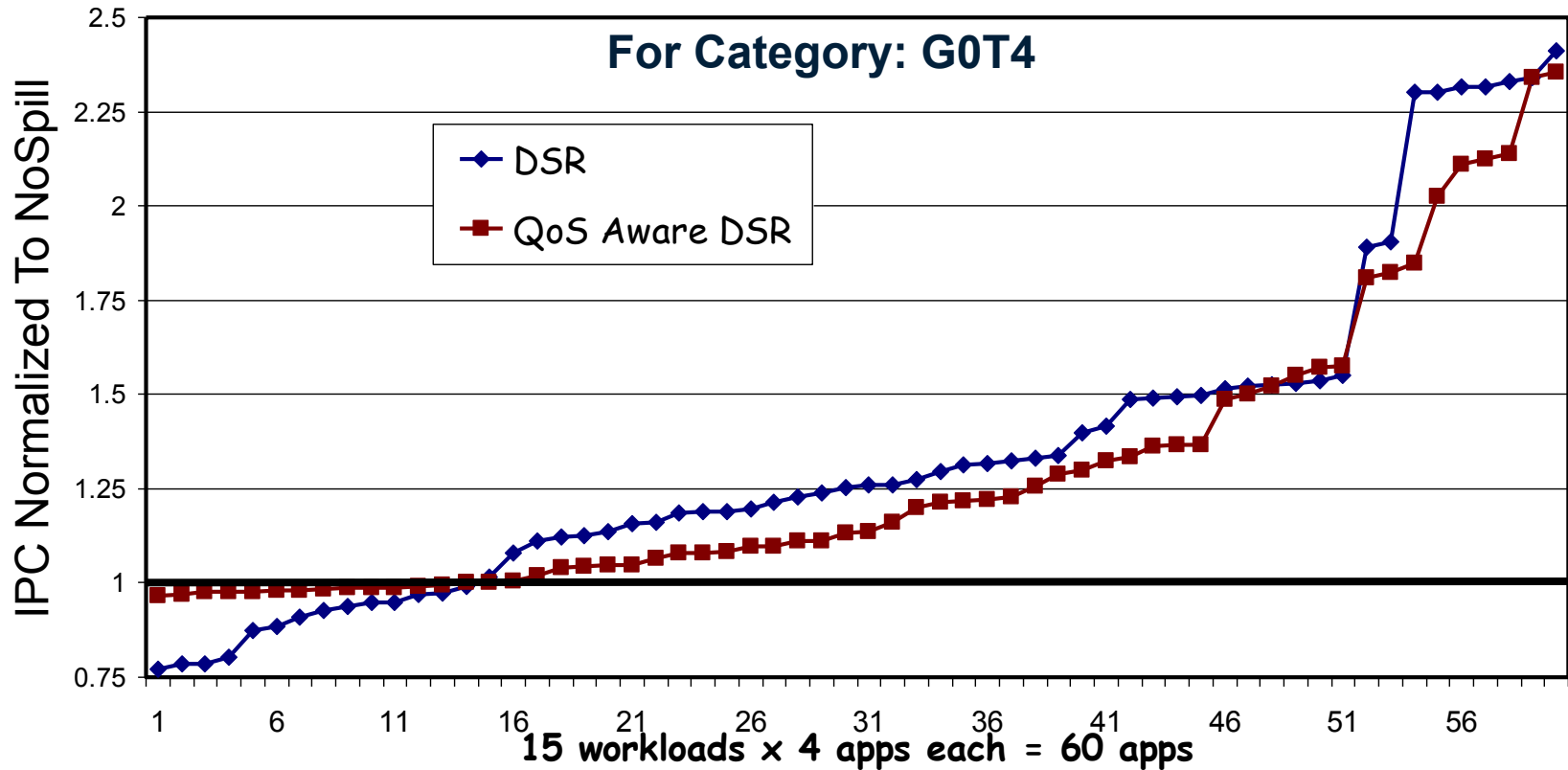
Estimated by Spiller Sets

$$\text{Weight of Miss} = 1 + \text{Max}(0, f(\Delta\text{Miss}))$$

Calculate weight every 4M cycles. Needs 3 counters per core

Over time, $\Delta\text{Miss} \rightarrow 0$, if DSR is causing more misses.

IPC of QoS-Aware DSR



IPC curves for other categories almost overlap for the two schemes.
Avg. throughput improvement across all 495 workloads similar (17.5% vs. 18%)

Distributed Caches

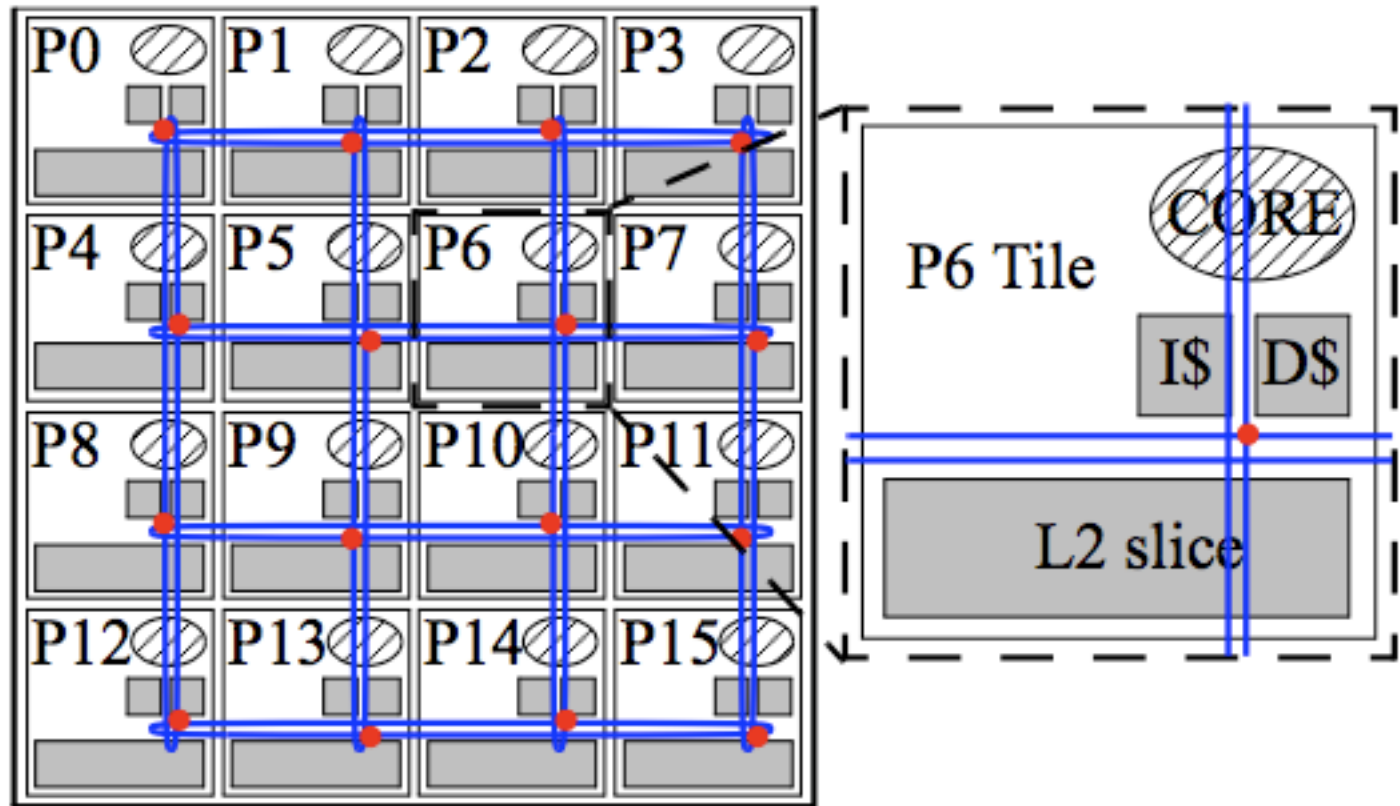
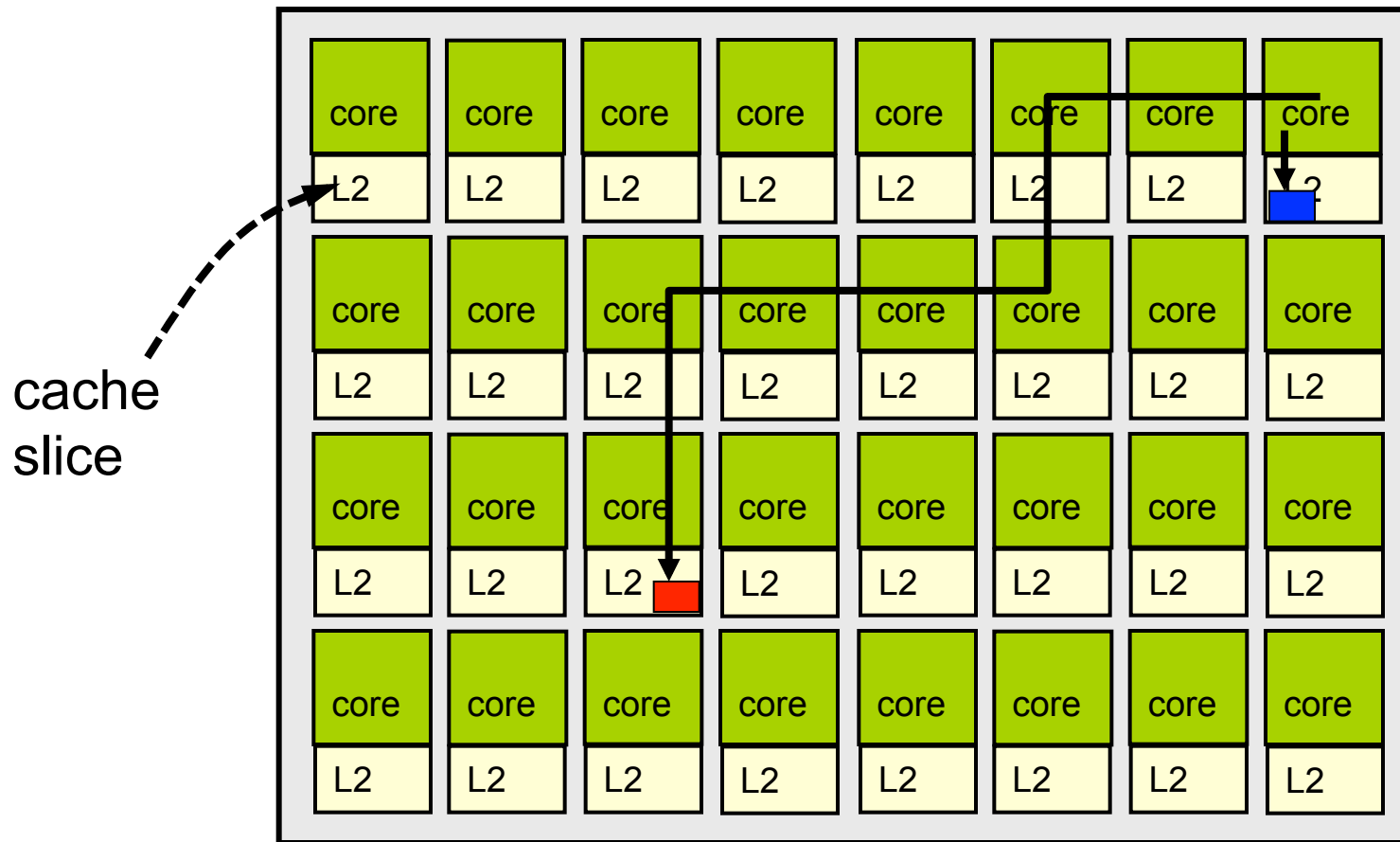


FIGURE 1. Typical tiled architecture. Tiles are interconnected into a 2-D folded torus. Each tile contains a core, L1 instruction and data caches, a shared-L2 cache slice, and a router/switch.

Caching for Parallel Applications



- ➡ Data placement determines performance
- ➡ Goal: place data on chip close to where they are used

Shared Cache Management: Research Topics

- Scalable partitioning algorithms
 - Distributed caches have different tradeoffs
- Configurable partitioning algorithms
 - Many metrics may need to be optimized at different times or at the same time
 - It is not only about overall performance
- Ability to have high capacity AND high locality (fast access)
- Within vs. across-application prioritization
- Holistic design
 - How to manage caches, NoC, and memory controllers together?
- Cache coherence in shared/private distributed caches