Computer Architecture

Lecture 18b:

Multi-Core Cache Management

Prof. Onur Mutlu

ETH Zürich

Fall 2018

22 November 2018

Summary of Last Few Lectures

- Approaches to mitigate and control memory interference, provide QoS
  - Request Scheduling
  - Source Throttling
  - Data Mapping
  - Thread Scheduling

Today

- Shared Cache Management
- Making Caching More Effective

Before That…

- Let's take a broader view of what we have done so far
  - https://safari.ethz.ch/architecture/fall2018/doku.php?id=schedule
- 17.5 lectures
  - All cutting edge yet fundamental topics
  - All research areas, ongoing
- 3 labs
- 3 homeworks
- Many readings (hopefully)

Any Feedback or Thoughts/Ideas

- Please email me directly
  - omutlu@gmail.com

- I am always interested in:
  - Any type of feedback about the course
  - Suggestions for better learning on your part
  - Any ideas you might have on any related topic

- If you want to do research in any of the covered topics or any topic in Comp Arch, HW/SW Interaction & related areas
  - We have many projects and a great environment to perform top-notch research, bachelor's/master's/semester projects
  - So, talk with me (email, in-person, WhatsApp, etc.)

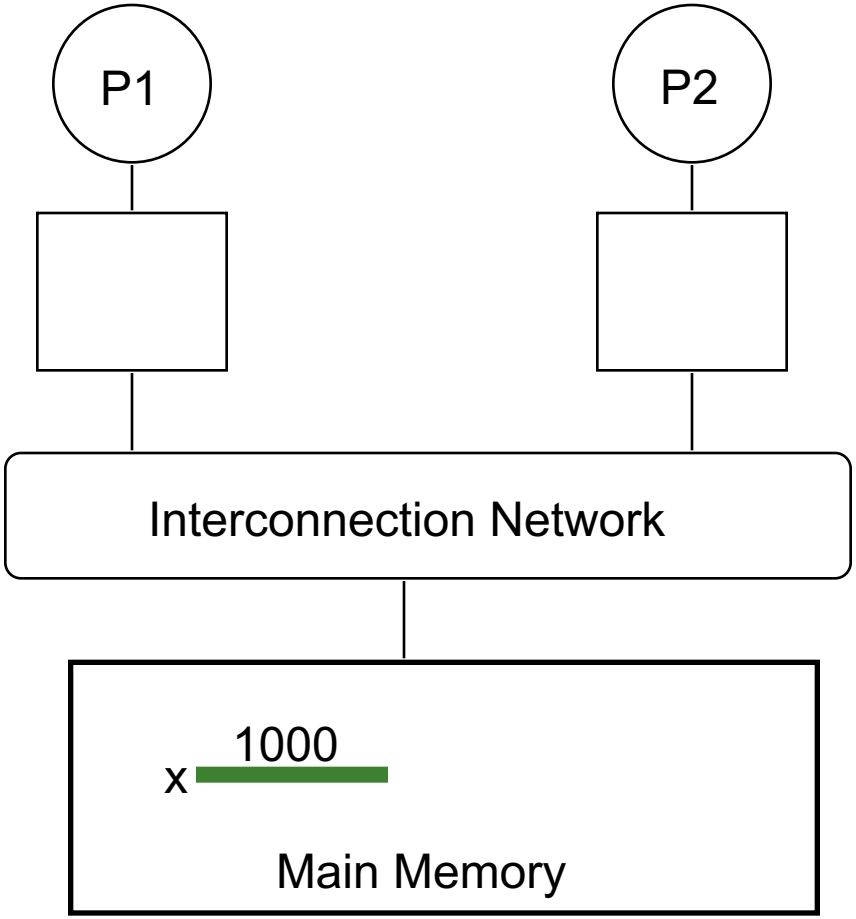
Multi-Core Caching Issues

Multi-Core Issues in Caching

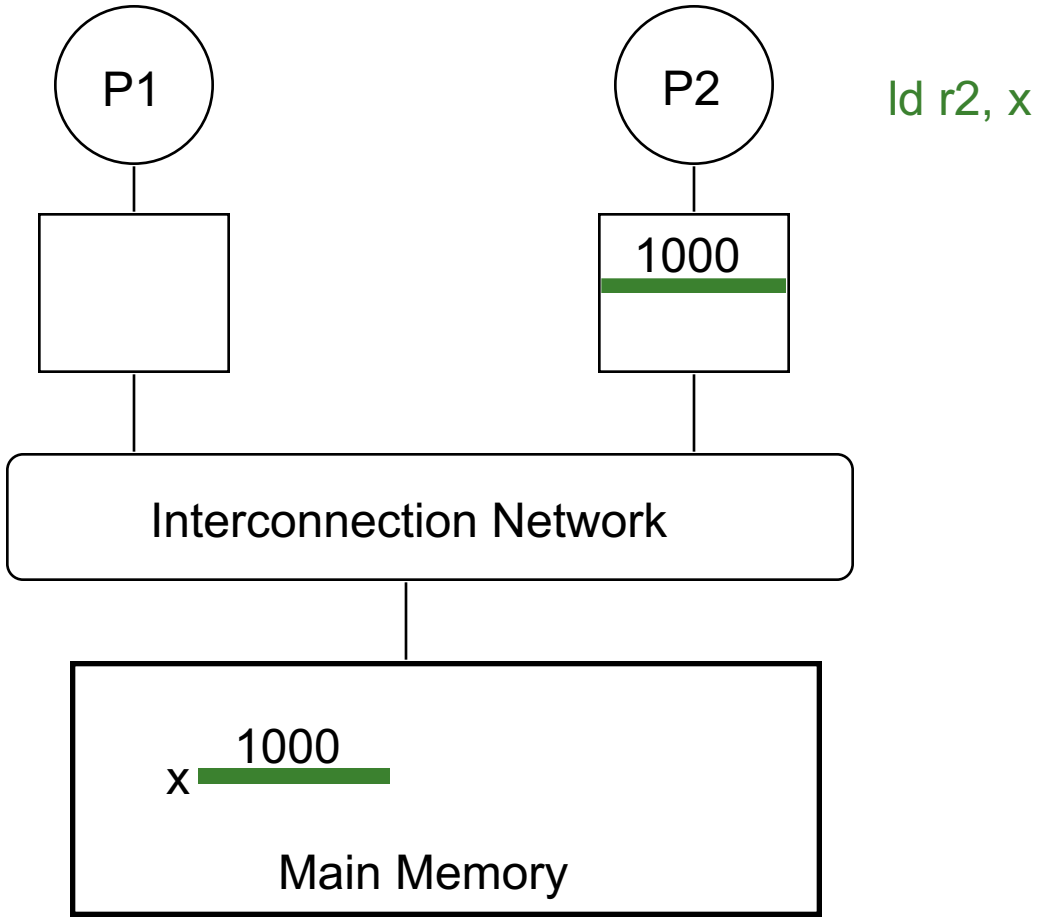
- Multi-core
  - More pressure on the memory/cache hierarchy → cache efficiency a lot more important
  - Private versus shared caching
  - Providing fairness/QoS in shared multi-core caches
  - How to handle shared data between cores
  - How to organize/connect caches:
    - Non-uniform cache access and cache interconnect design
- Placement/insertion
  - Identifying what is most profitable to insert into cache
  - Minimizing dead/useless blocks
- Replacement
  - Cost-aware: which block is most profitable to keep?

Cache Coherence

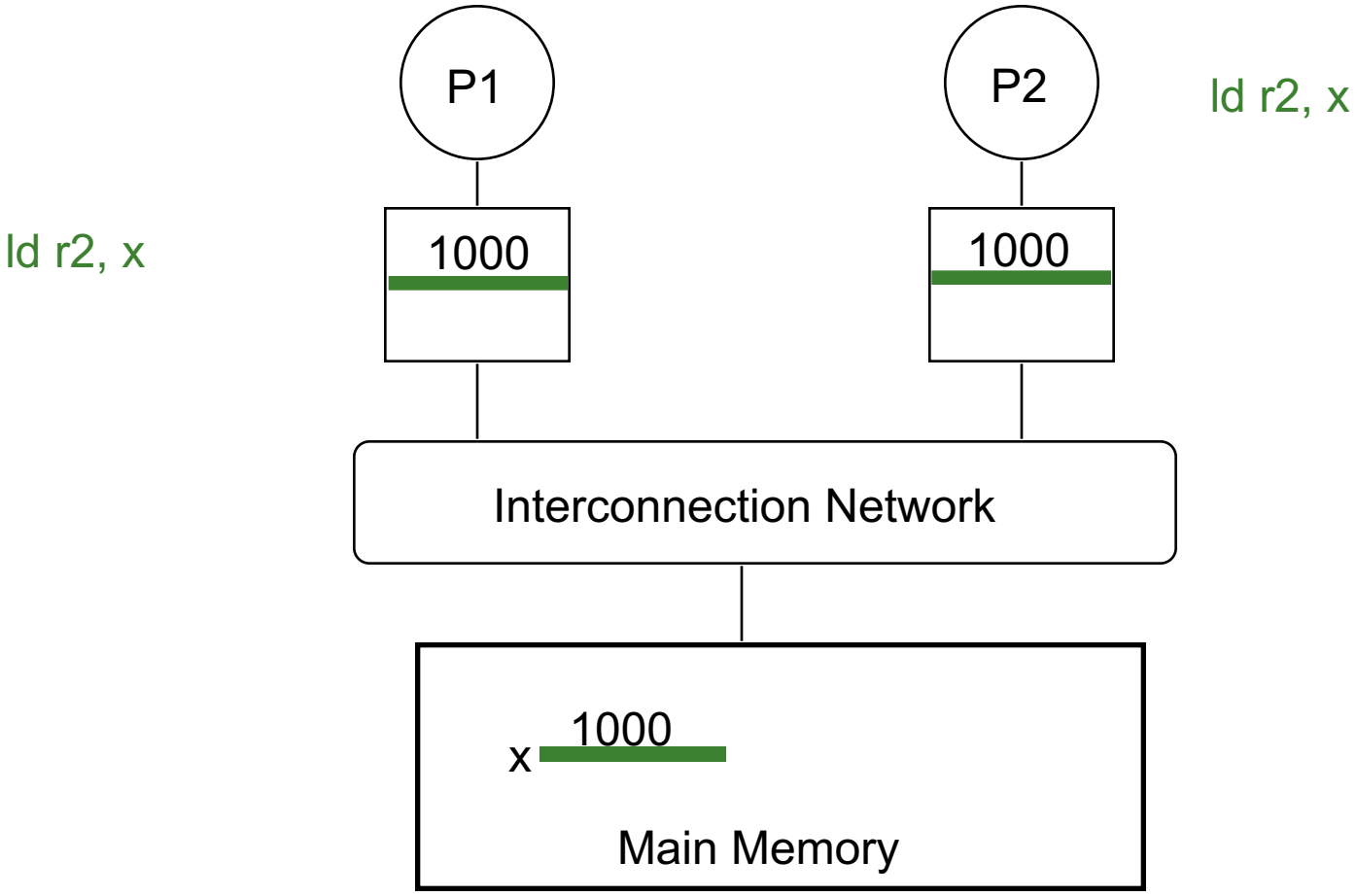
- Basic question: If multiple processors cache the same block, how do they ensure they all see a consistent state?



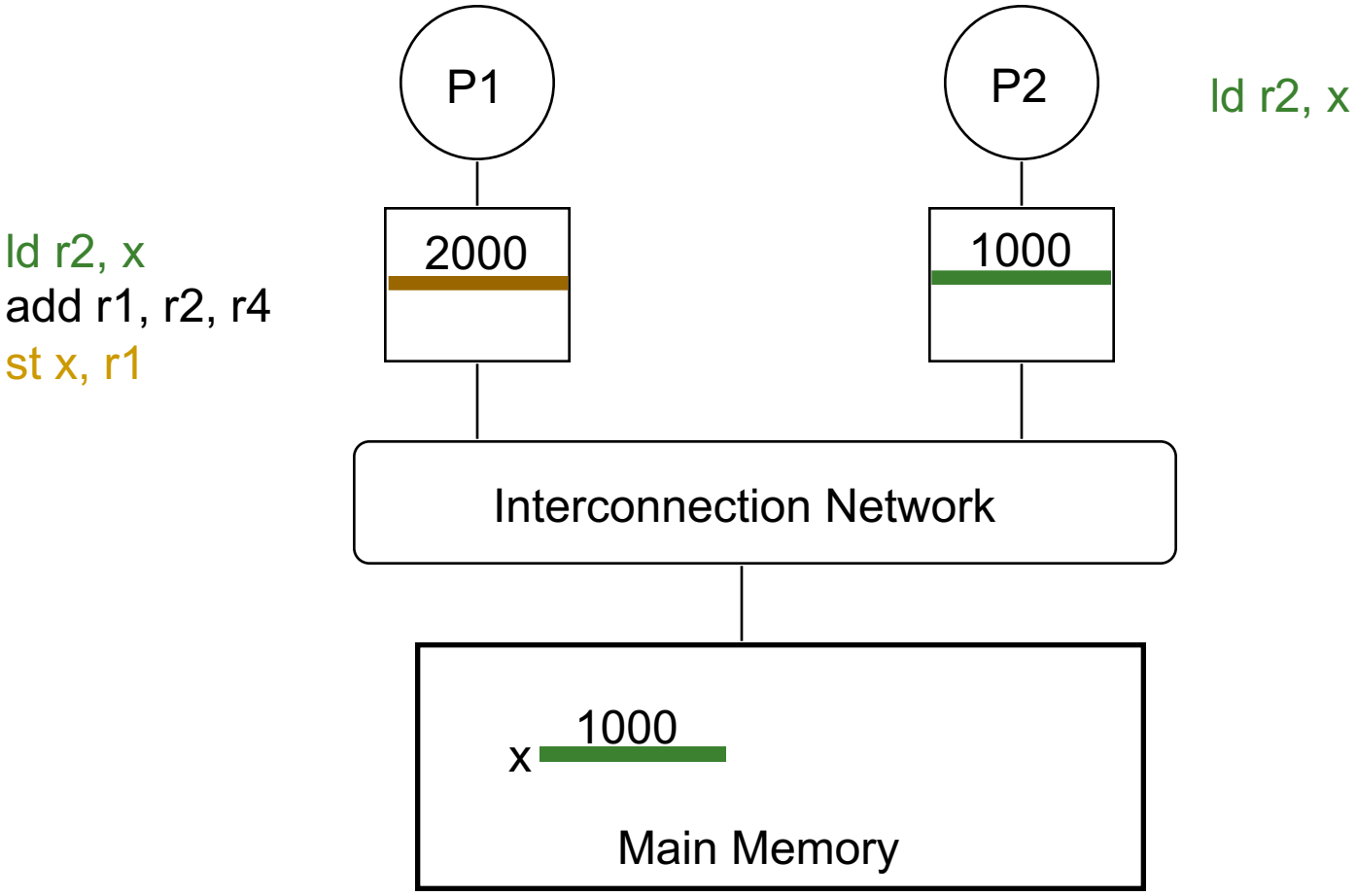
The Cache Coherence Problem



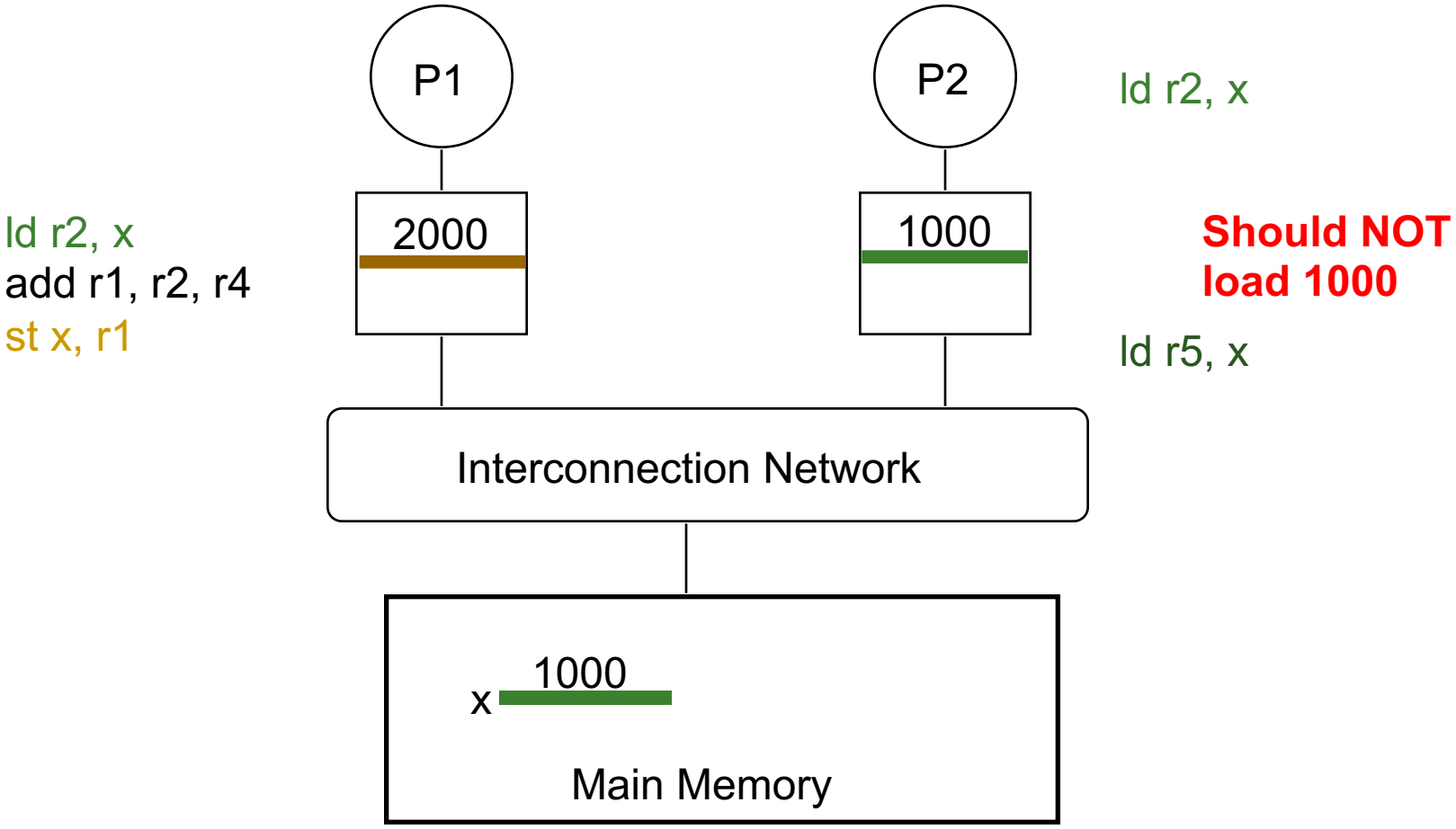
The Cache Coherence Problem



The Cache Coherence Problem



The Cache Coherence Problem



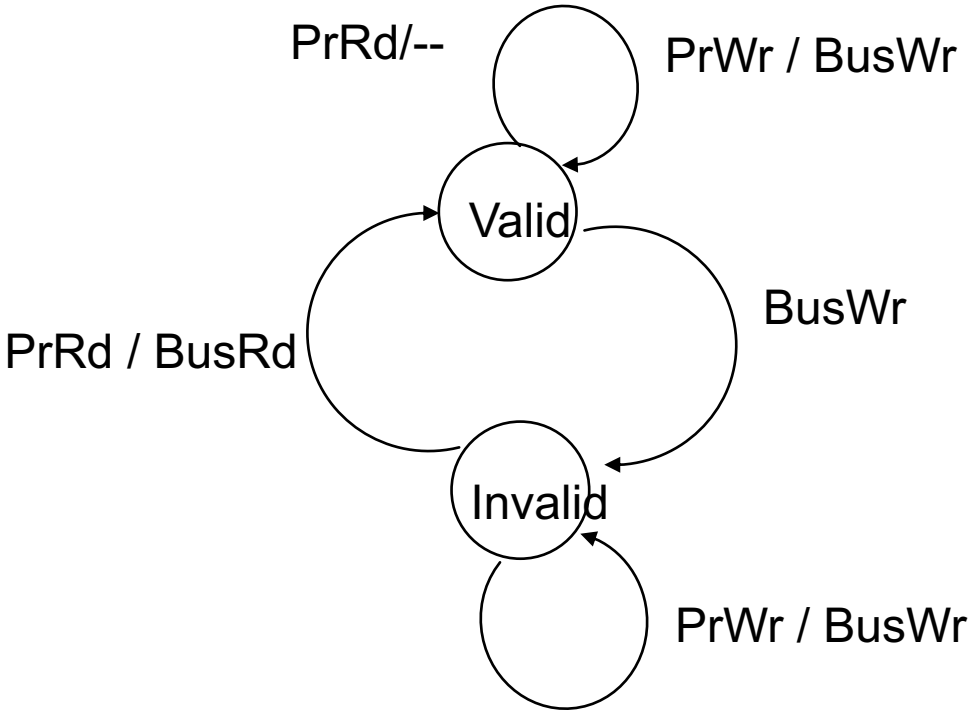
Cache Coherence: Whose Responsibility?

- Software
  - Can the programmer ensure coherence if caches are invisible to software?
  - What if the ISA provided the following instruction?
    - FLUSH-LOCAL A: Flushes/invalidates the cache block containing address A from a processor's local cache
    - When does the programmer need to FLUSH-LOCAL an address?
  - What if the ISA provided the following instruction?
    - FLUSH-GLOBAL A: Flushes/invalidates the cache block containing address A from all other processors' caches
    - When does the programmer need to FLUSH-GLOBAL an address?

- Hardware
  - Simplifies software's job
  - One idea: Invalidate all other copies of block A when a processor writes to it

Snoopy Cache Coherence

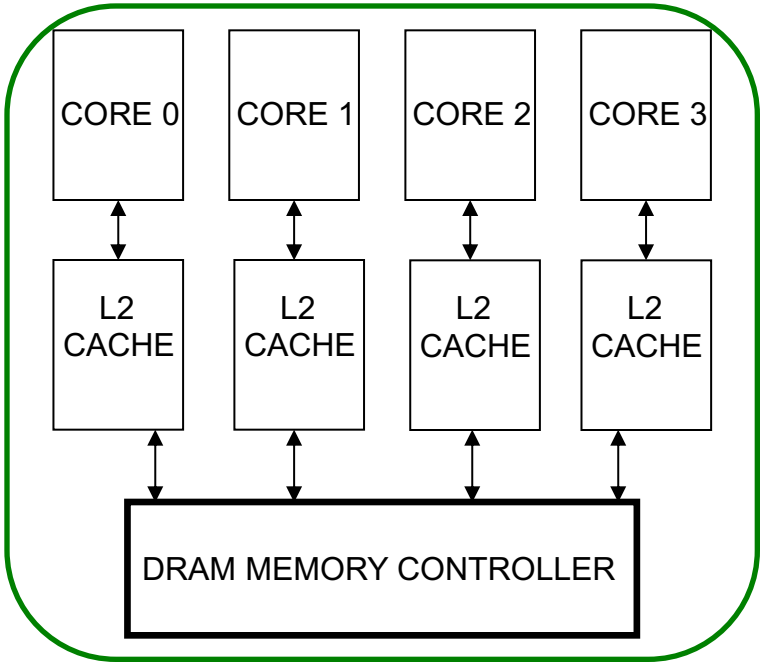
- Caches "snoop" (observe) each other's write/read operations
- A simple protocol:

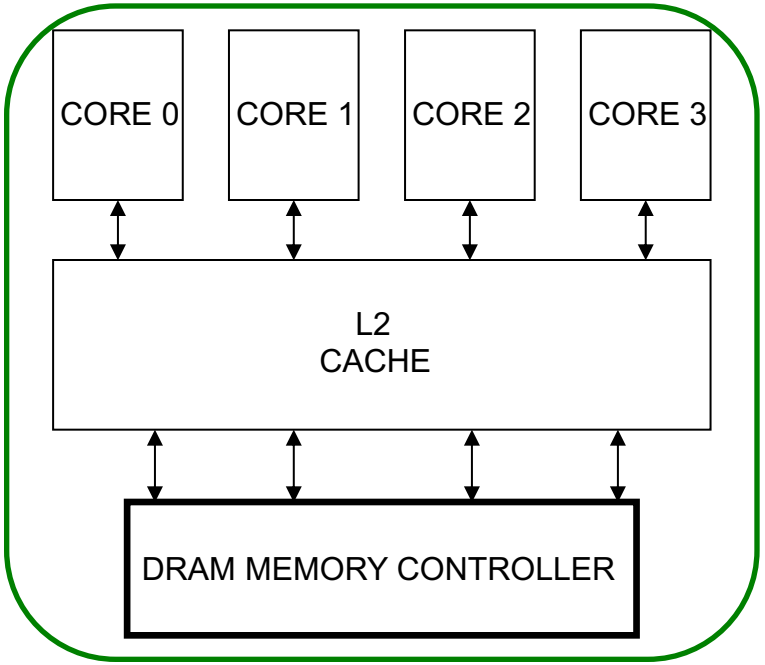


- Write-through, no-write-allocate cache
- Actions: PrRd, PrWr, BusRd, BusWr

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

Handling Shared Caches

- Controlled cache sharing
  - Approach 1: Design shared caches but control the amount of cache allocated to different cores
  - Approach 2: Design "private" caches but spill/receive data from one cache to another

- More efficient cache utilization
  - Minimize the wasted cache space
    - by keeping out useless blocks
    - by keeping in cache blocks that have maximum benefit
    - by minimizing redundant data

Controlled Cache Sharing: Examples

- 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: Examples

- Qureshi et al., "A Case for MLP-Aware Cache Replacement," ISCA 2005.
- Qureshi et al., "Adaptive Insertion Policies for High Performance Caching," ISCA 2007.
- 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.

Controlled Shared Caching

Hardware-Based Cache Partitioning

Utility Based Shared Cache Partitioning

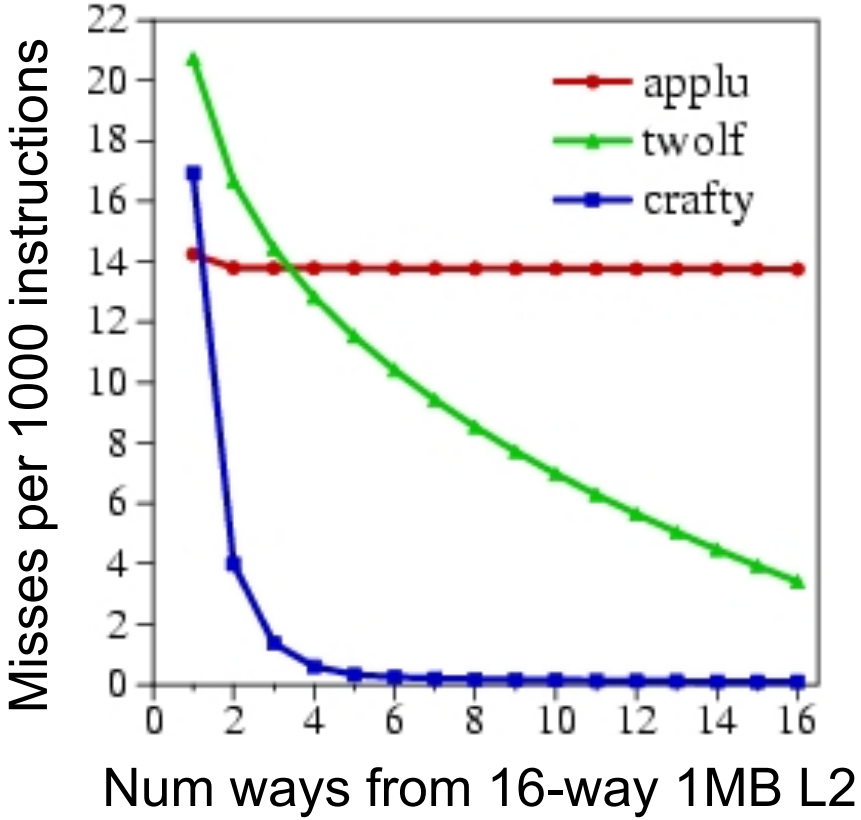
- Goal: Maximize system throughput
- Observation: Not all threads/applications benefit equally from caching → simple LRU replacement not good for system throughput
- Idea: Allocate more cache space to applications that obtain the most benefit from more space

- 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 = Misses with a ways – Misses with b ways

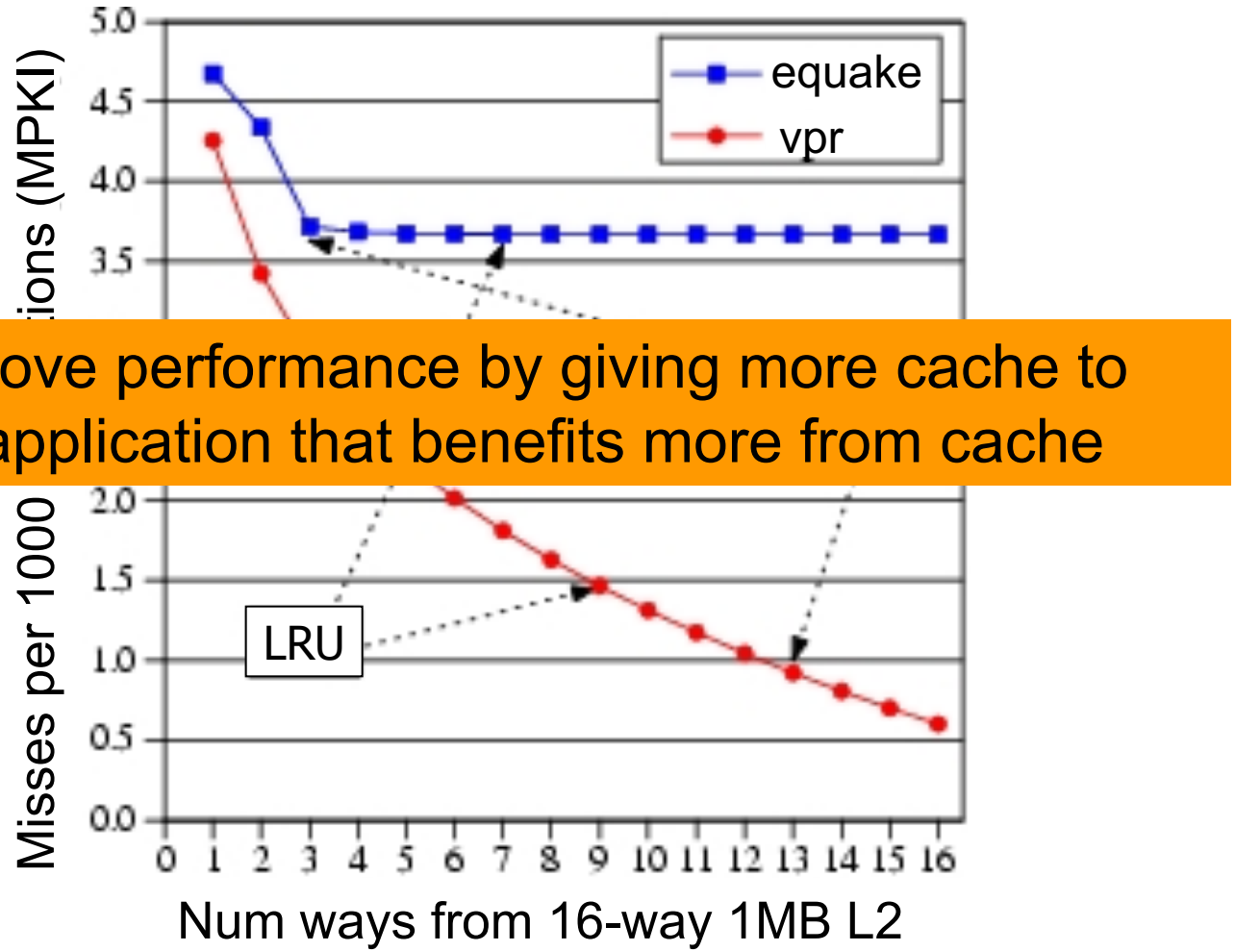


Low Utility

High Utility

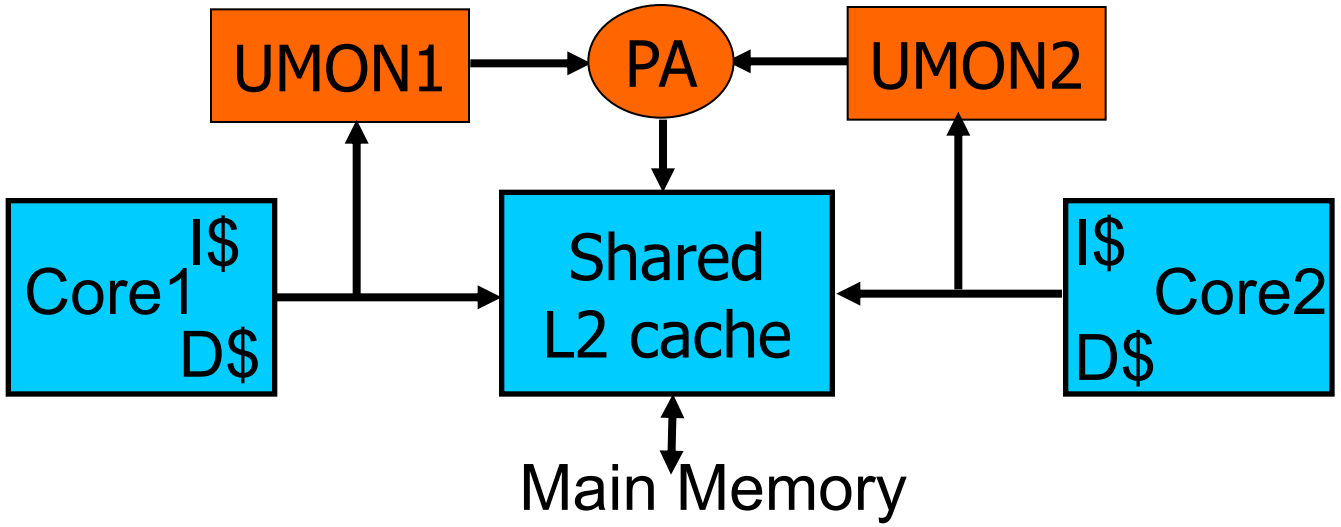
Saturating Utility

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)

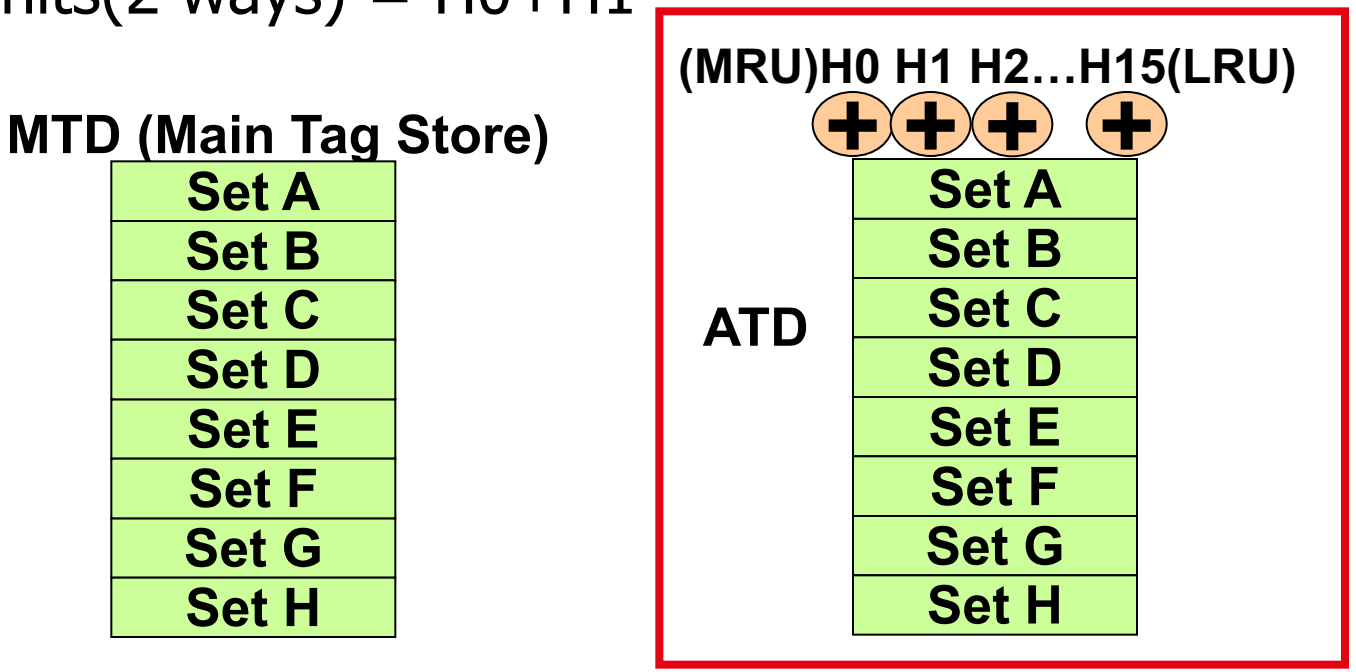


Three components:

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

1. Utility Monitors

- For each core, simulate LRU policy using a separate tag store called ATD (auxiliary tag directory/store)
- Hit counters in ATD to count hits per recency position
- LRU is a stack algorithm: hit counts ➔ utility
  E.g. hits(2 ways) = H0+H1



Utility Monitors

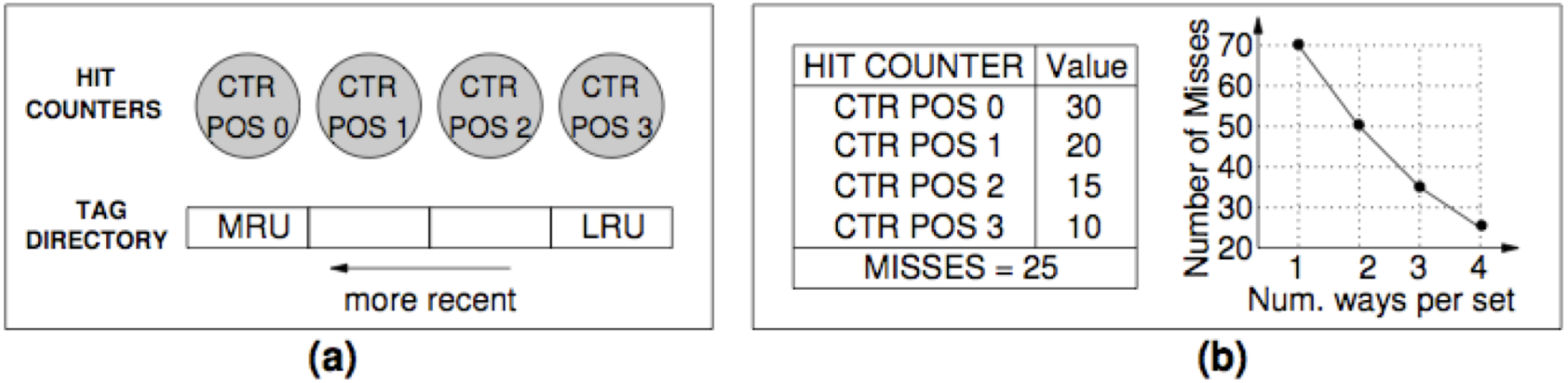
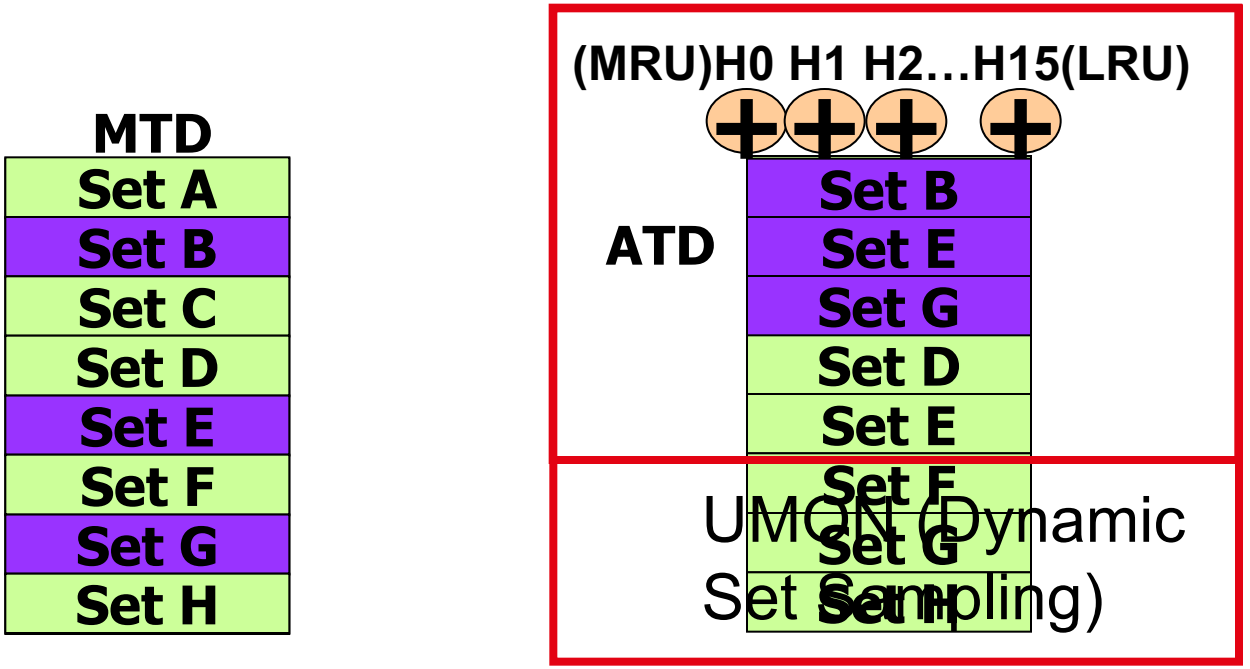


Figure 4. (a) Hit counters for each recency position. (b) Example of how utility information can be tracked with stack property.

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



2. Partitioning Algorithm

- Evaluate all possible partitions and select the best

- With a ways to core1 and (16-a) ways to core2:

  $Hits_{core1} = (H_0 + H_1 + \ldots + H_{a-1})$ ---- from UMON1
  $Hits_{core2} = (H_0 + H_1 + \ldots + H_{16-a-1})$ ---- from UMON2

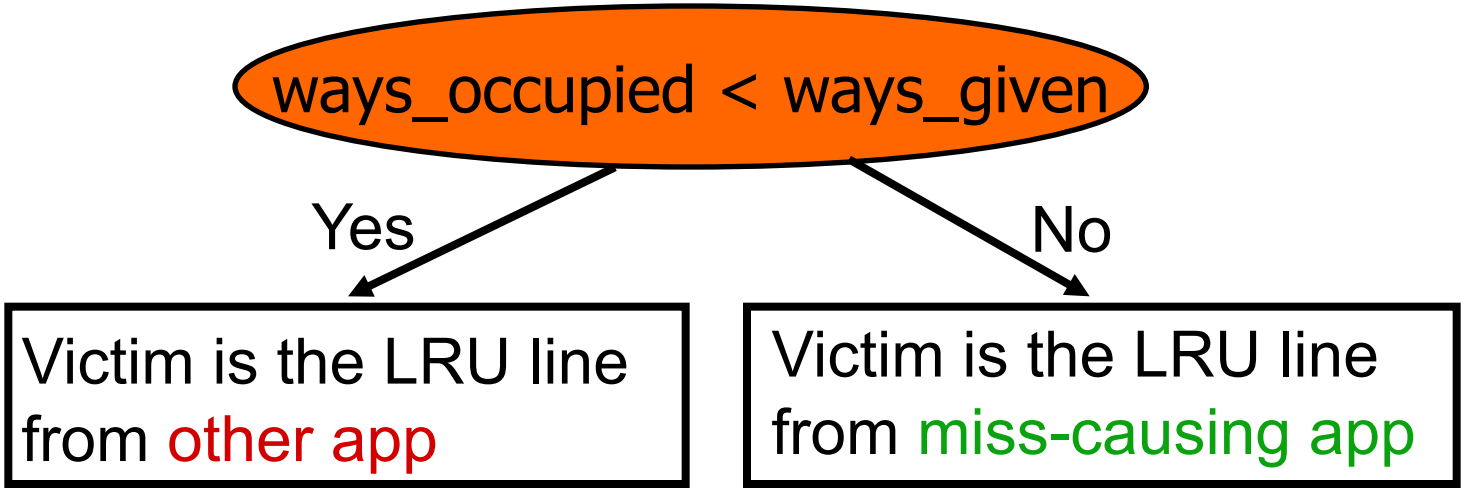
- Select a that maximizes ($Hits_{core1} + Hits_{core2}$)

- Partitioning done once every N million cycles

3. Enforcing Partitions: 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

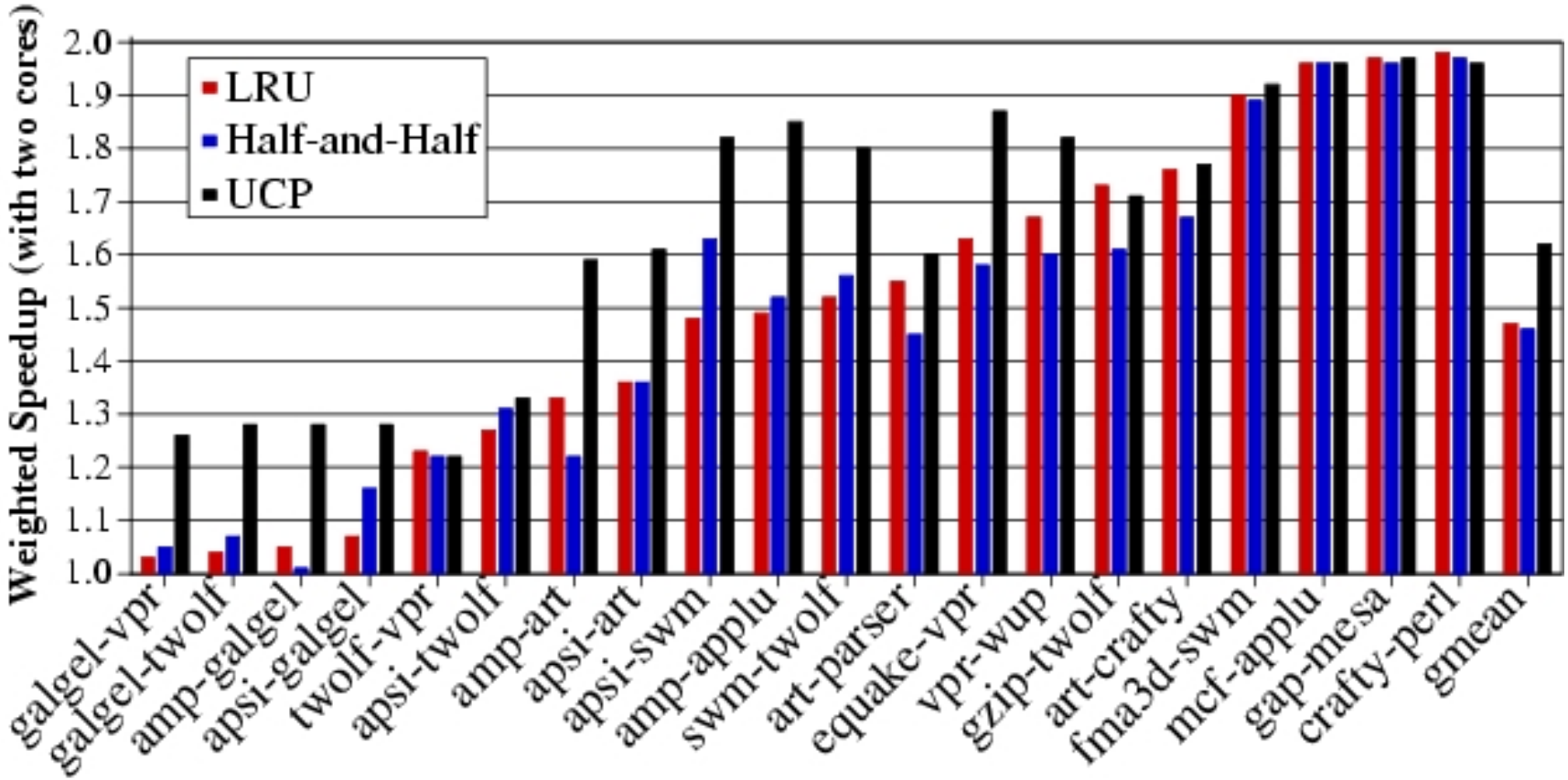
- Three metrics for performance:

1. Weighted Speedup (default metric)
   - ➔ perf = $IPC_1/SingleIPC_1$ + $IPC_2/SingleIPC_2$
   - ➔ correlates with system throughput [Eyerman+, IEEE Micro'08]

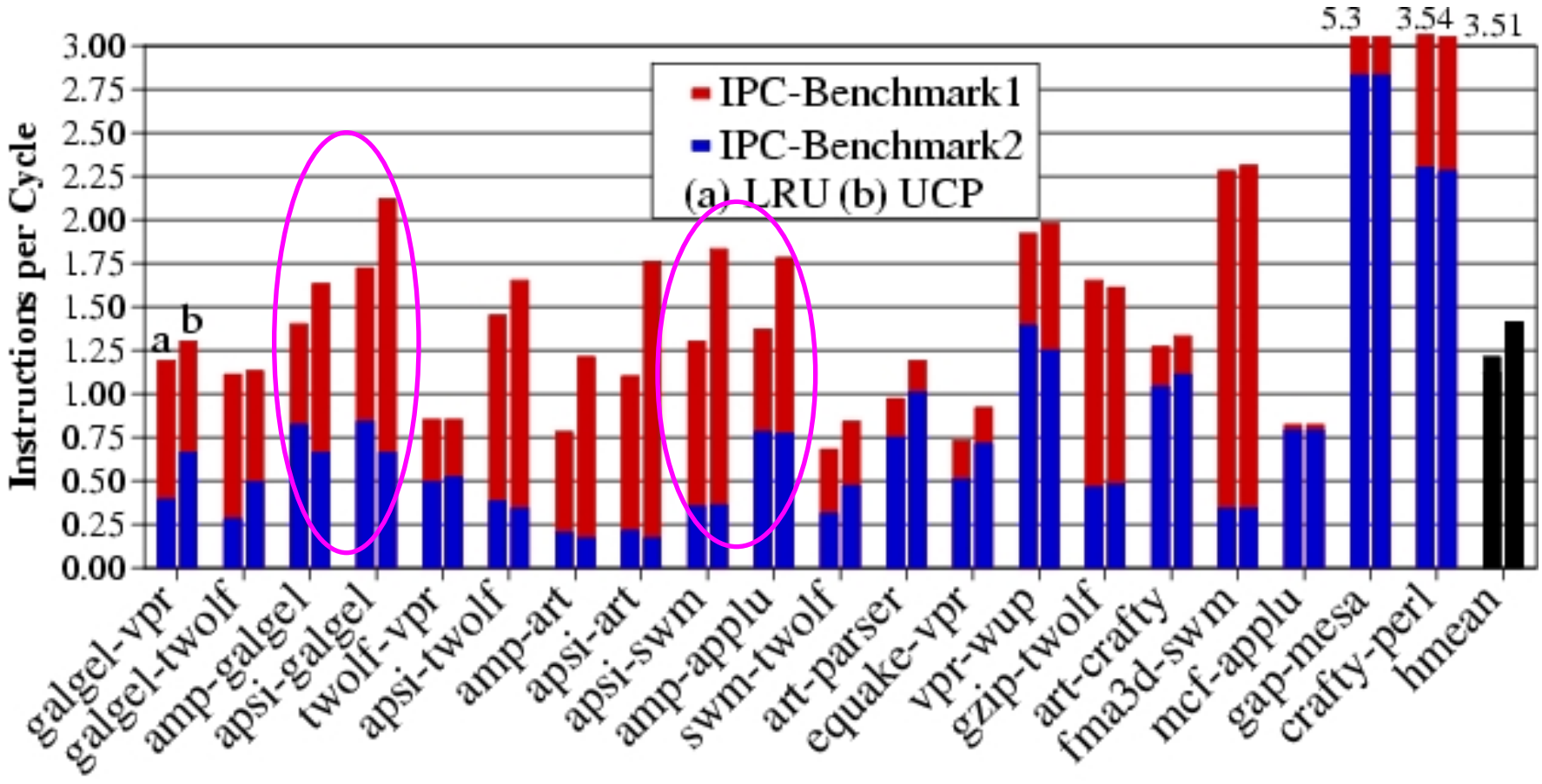
2. Throughput
   - ➔ perf = IPC_1 + IPC_2
   - ➔ can be unfair to low-IPC application

3. Hmean-fairness
   - ➔ perf = hmean($IPC_1/SingleIPC_1$, $IPC_2/SingleIPC_2$)
   - ➔ 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 to many cores

- Time complexity of partitioning low for two cores (number of possible partitions ≈ number of ways)

- Possible partitions increase exponentially with cores

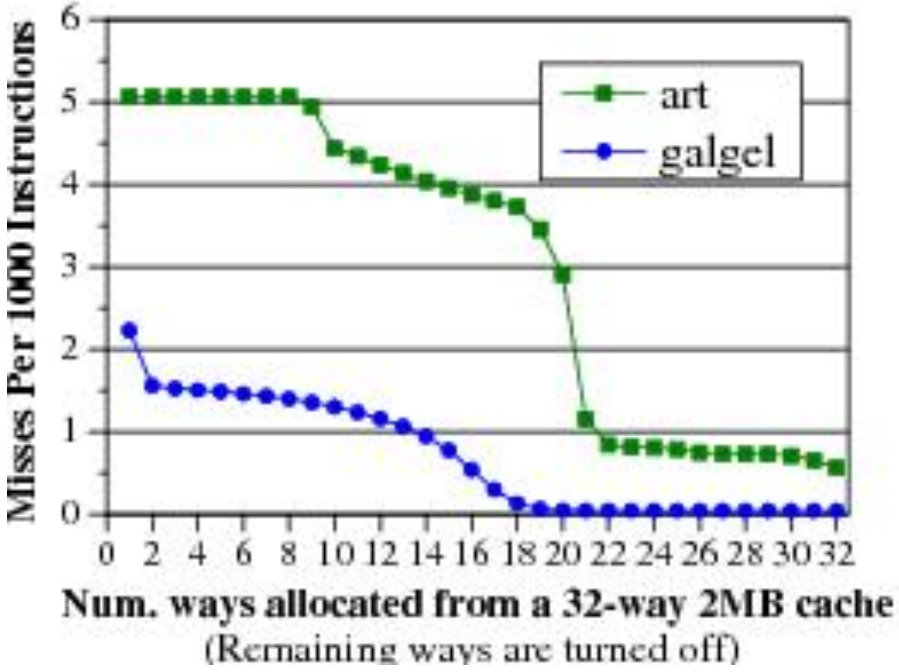
- For a 32-way cache, possible partitions:
  - 4 cores → 6545
  - 8 cores → 15.4 million

- Problem NP hard → need scalable partitioning algorithm

Greedy Algorithm [Stone+ ToC'92]

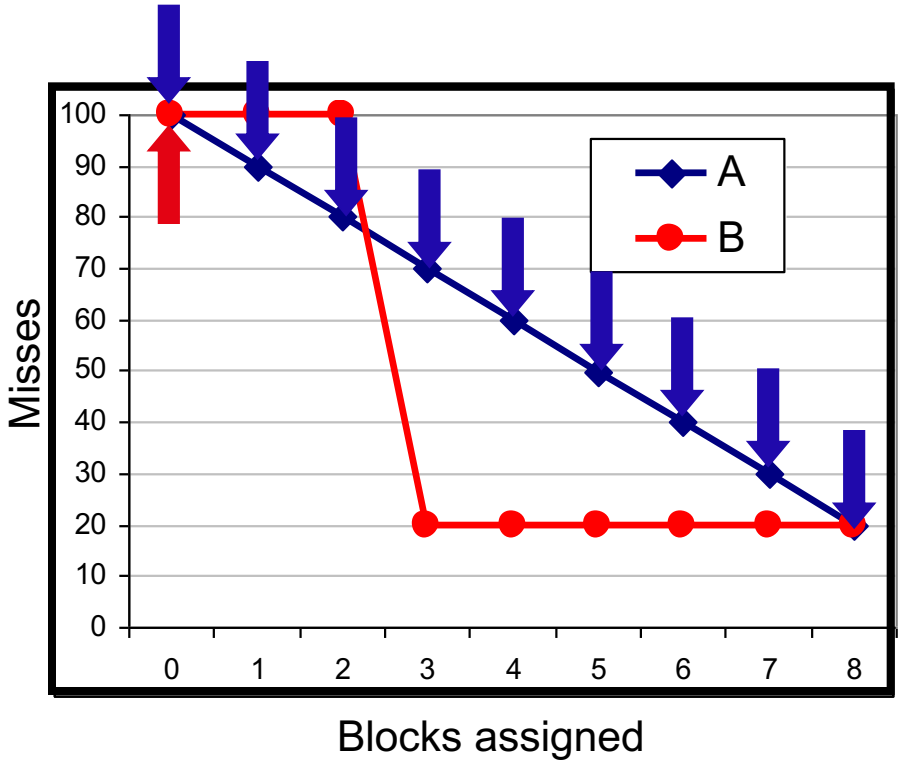
- Goal: Minimize overall number of misses
- Greedy Algorithm (GA) allocates 1 block to the app that has the max utility for one block. Repeat till all blocks allocated
- Provides optimal partitioning when utility curves are convex

- Pathological behavior for non-convex curves
  - Lookahead of only 1 block



Stone et al., "Optimal Partitioning of Cache Memory," IEEE ToC 1992.

Problem with Greedy Algorithm



In each iteration, the utility for 1 block:

U(A) = 10 misses
U(B) = 0 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

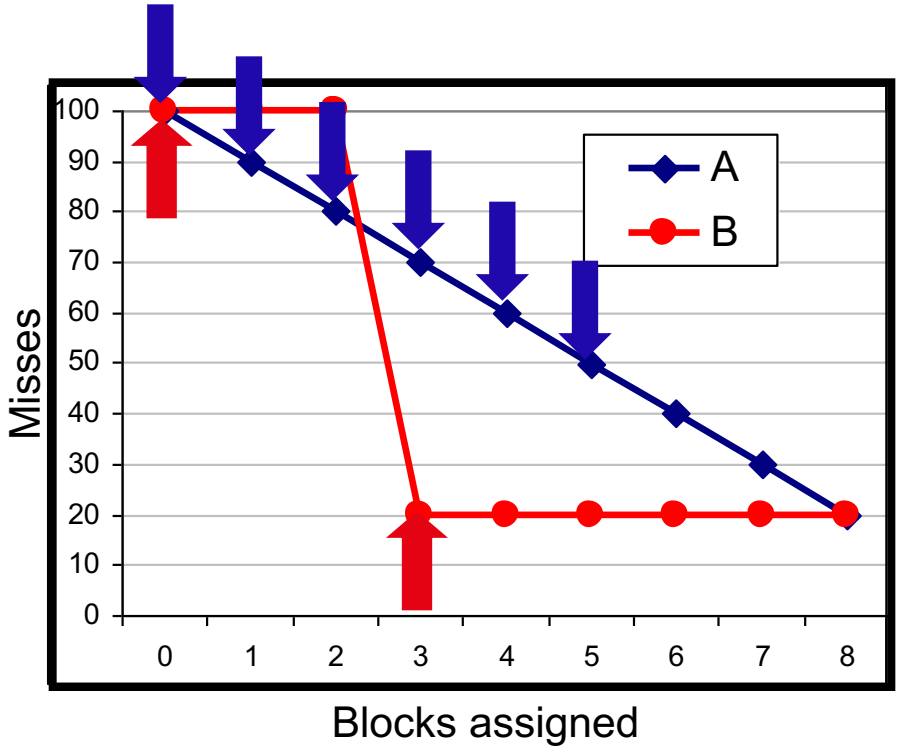
- Marginal Utility (MU) = Utility per cache resource
  - $MU_a^b = U_a^b/(b-a)$

- GA considers MU for 1 block.
- LA (Lookahead Algorithm) 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 until all blocks are 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

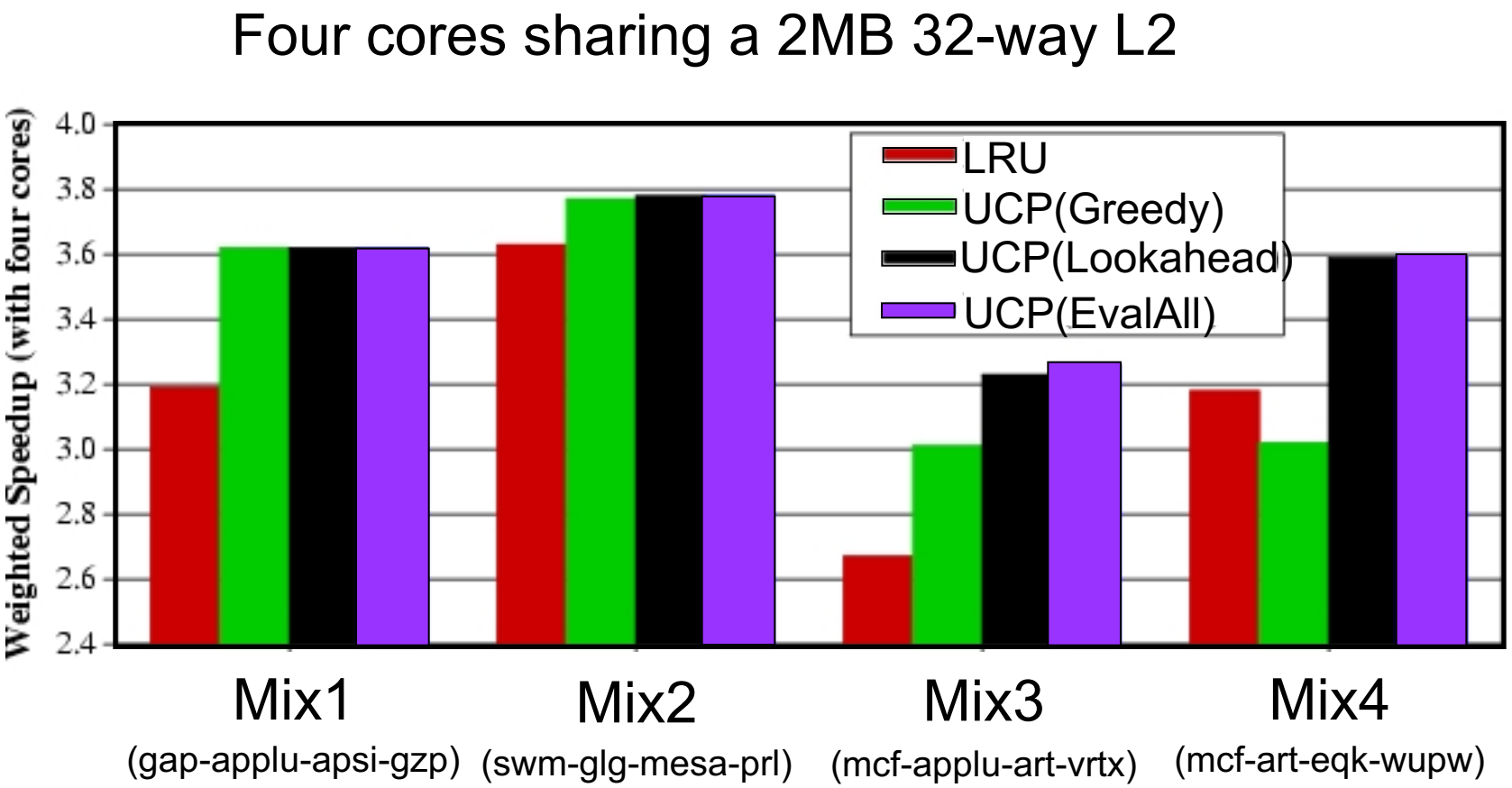
MU(B) = 0

A gets 1 block

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

Time complexity ≈ ways^2/2 (512 ops for 32-ways)

UCP Results



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?

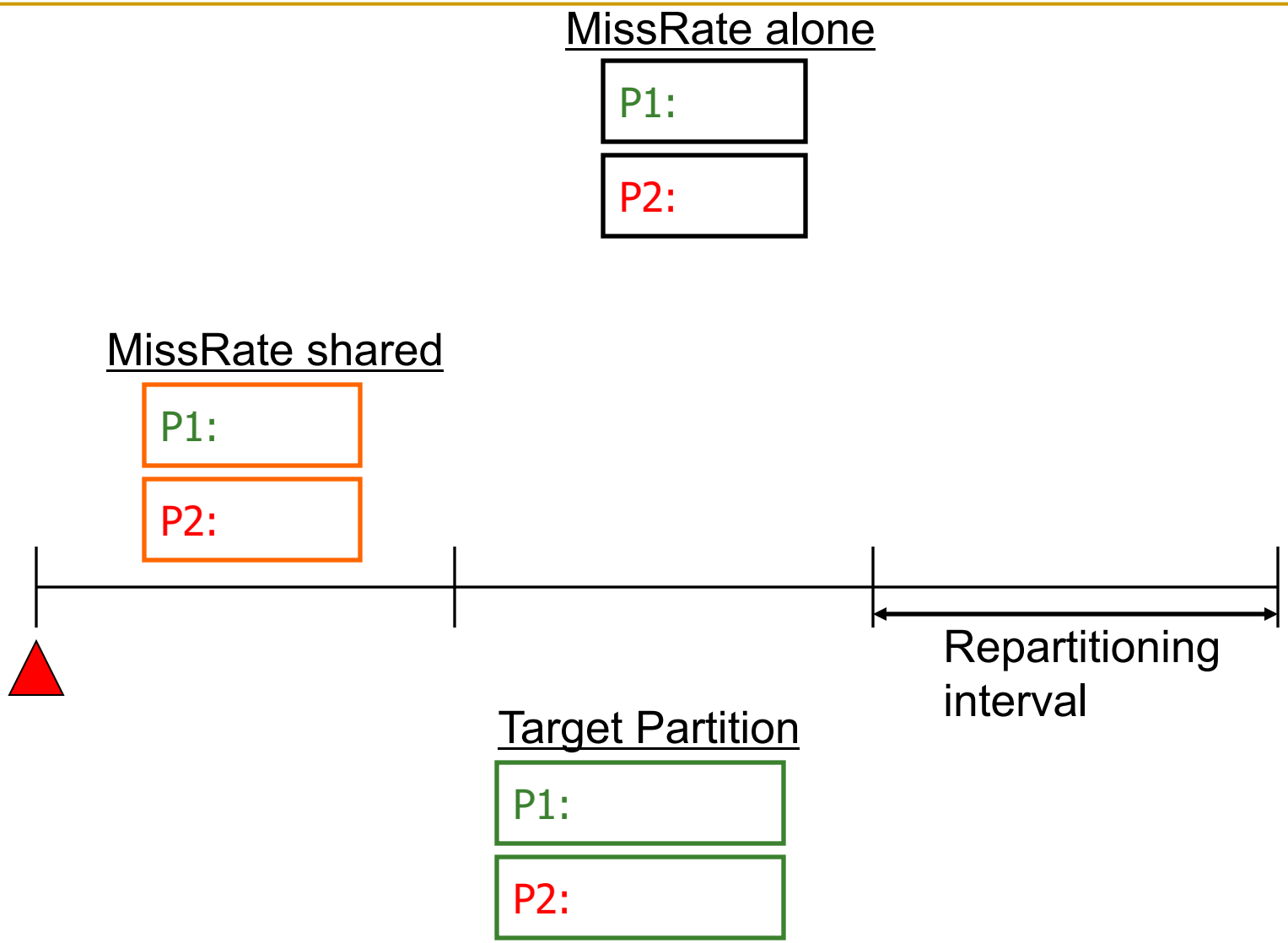
- Limitations
  - Scalability: Partitioning limited to ways. What if you have numWays < numApps?
  - Scalability: How is utility computed in a distributed cache?
  - What if past behavior is not a good predictor of utility?

Fair Shared Cache Partitioning

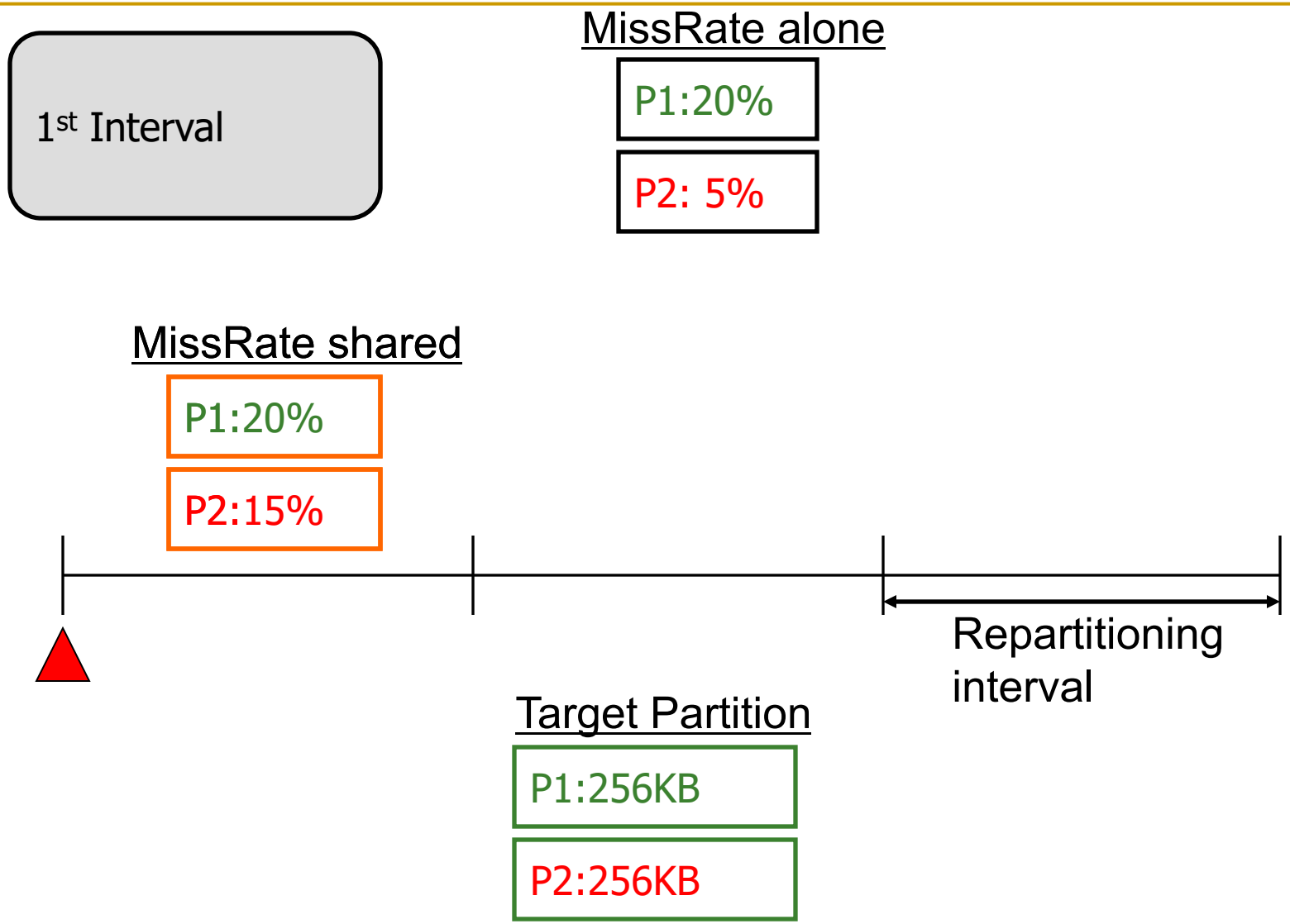
- Goal: Equalize the slowdowns of multiple threads sharing the cache
- Idea: Dynamically estimate slowdowns due to sharing and assign cache blocks to balance slowdowns
  - Approximate slowdown with change in miss rate

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

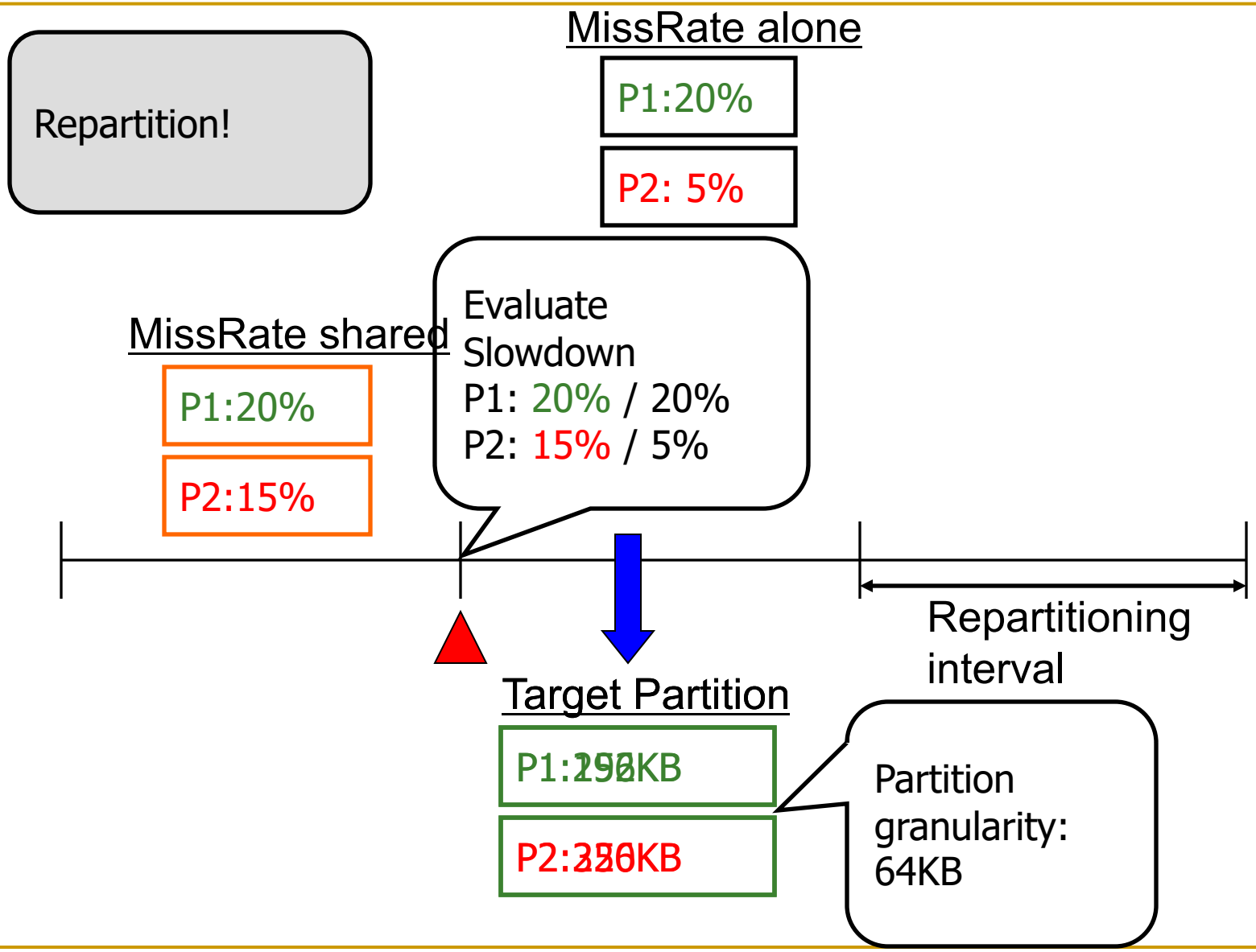
Dynamic Fair Caching Algorithm



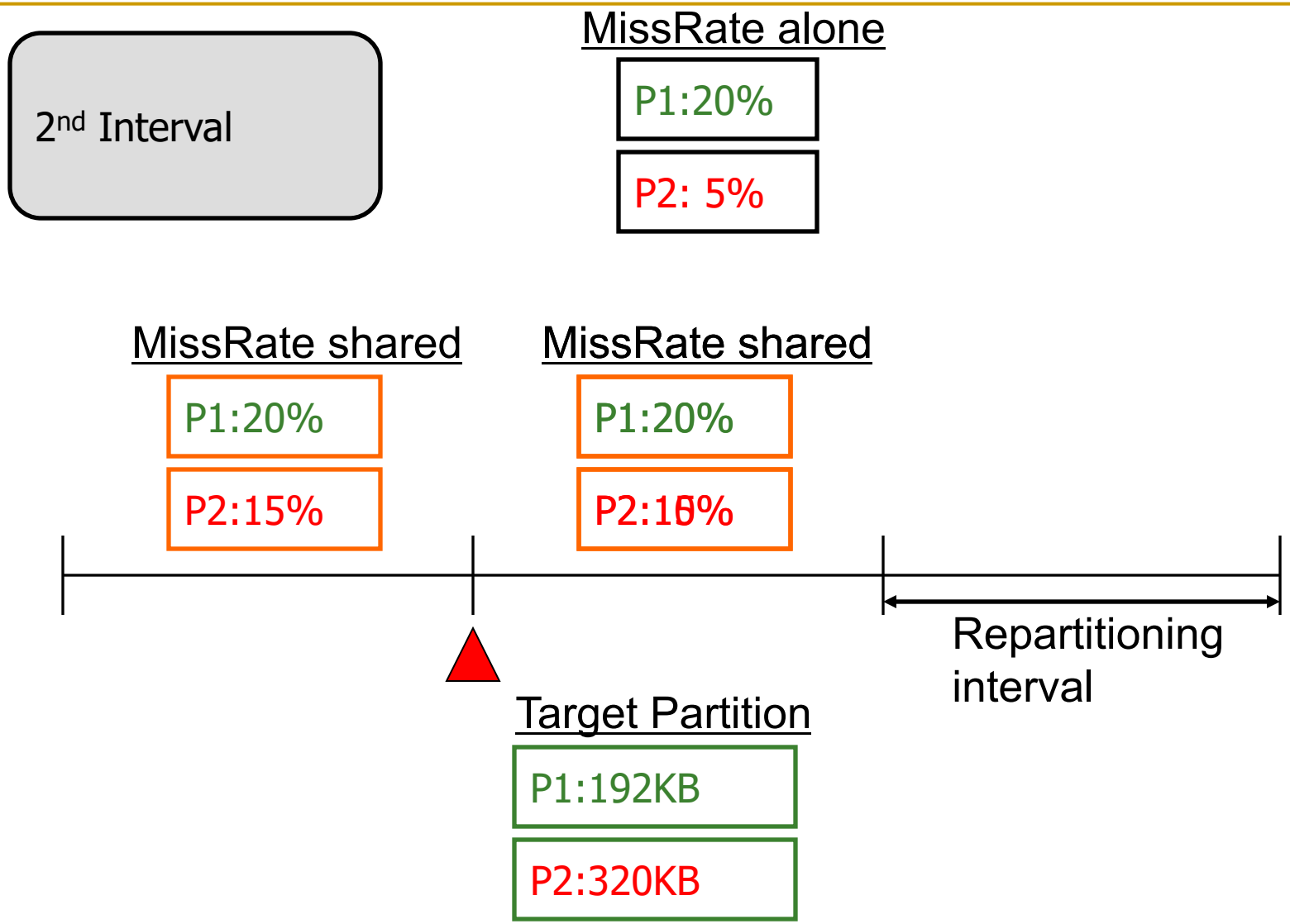
Dynamic Fair Caching Algorithm



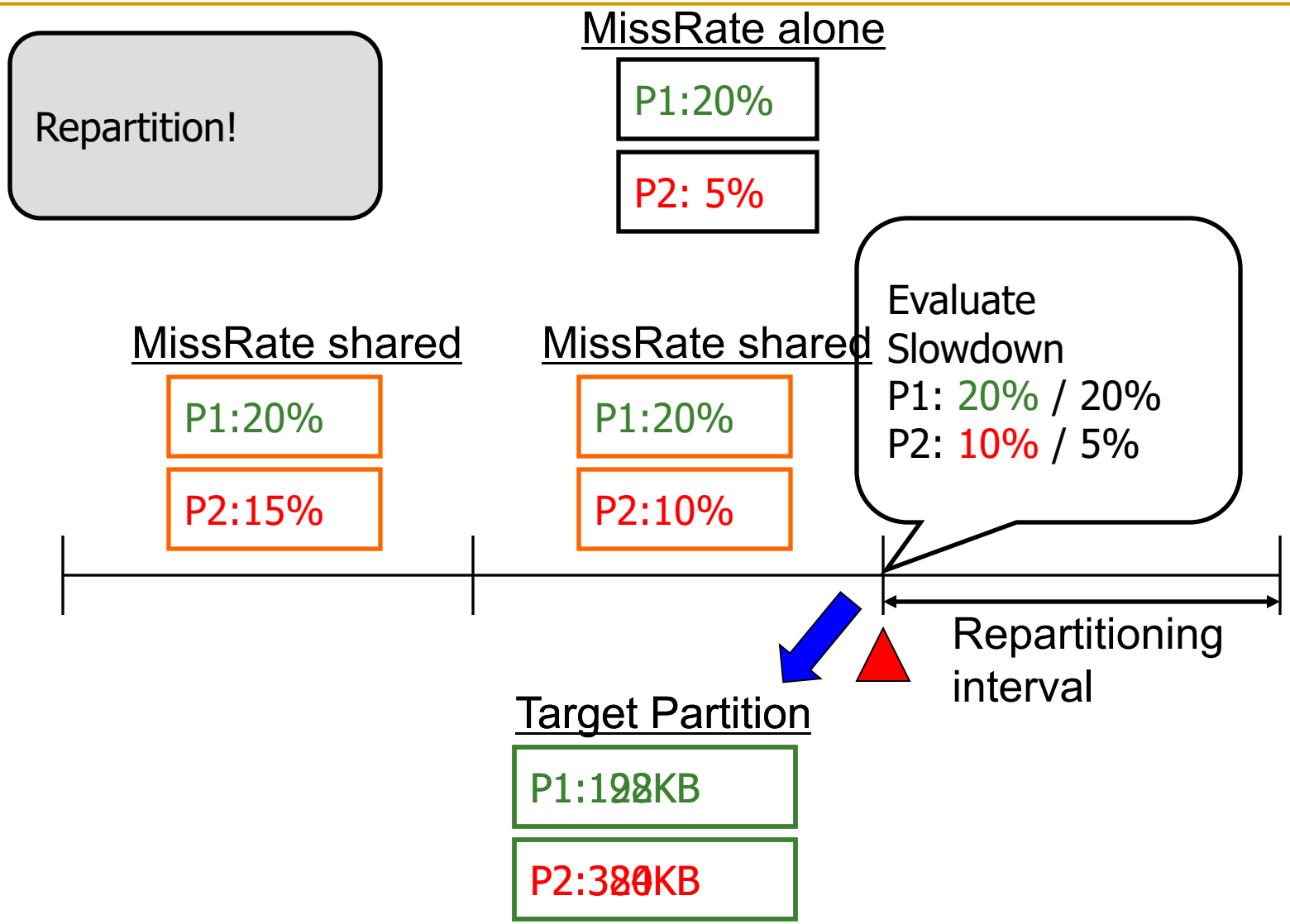
Dynamic Fair Caching Algorithm



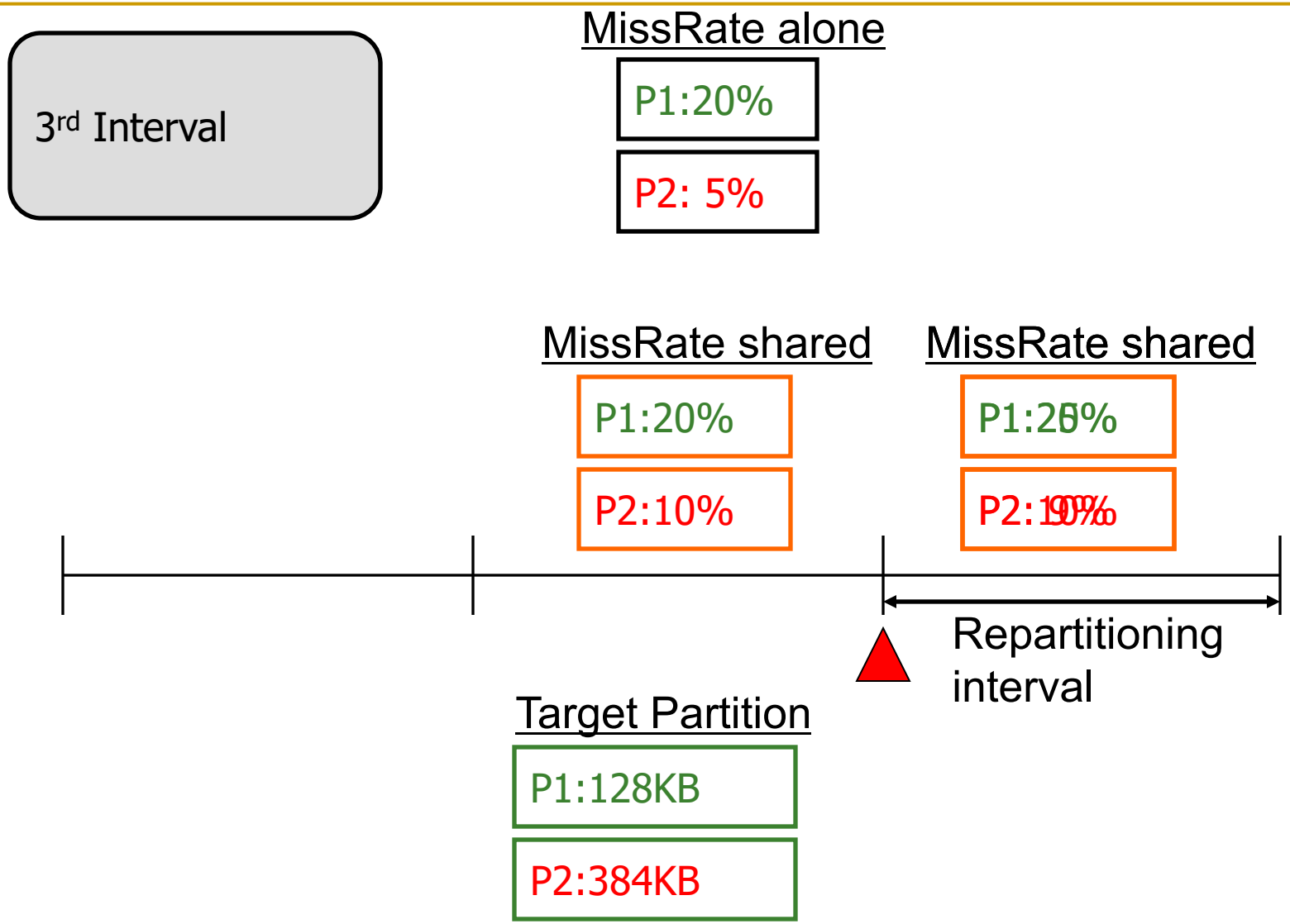
Dynamic Fair Caching Algorithm



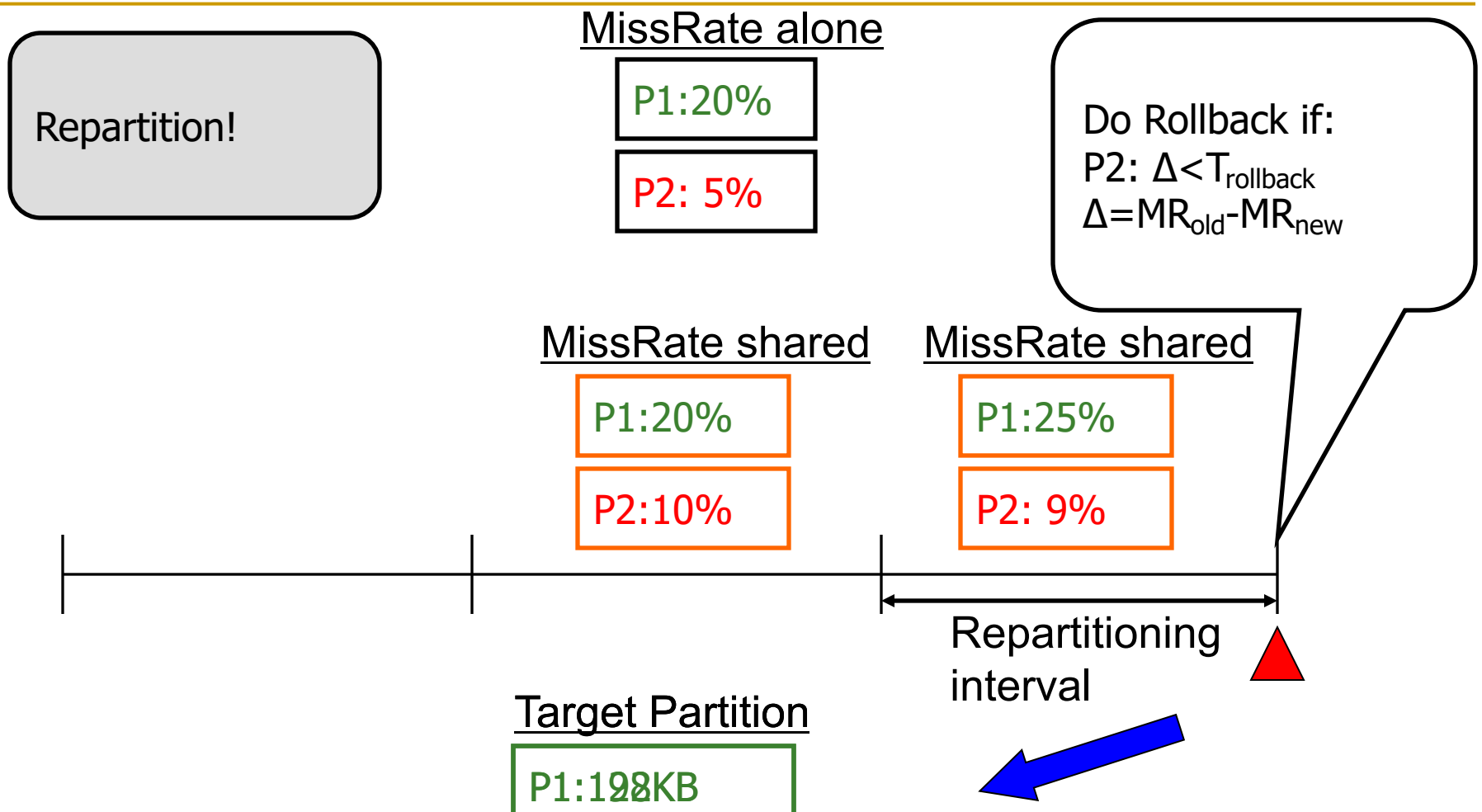
Dynamic Fair Caching Algorithm



Dynamic Fair Caching Algorithm



Dynamic Fair Caching Algorithm



Advantages/Disadvantages of the Approach

- Advantages
  + Reduced starvation
  + Better average throughput
  + Block granularity partitioning

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

Software-Based Shared Cache Partitioning

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

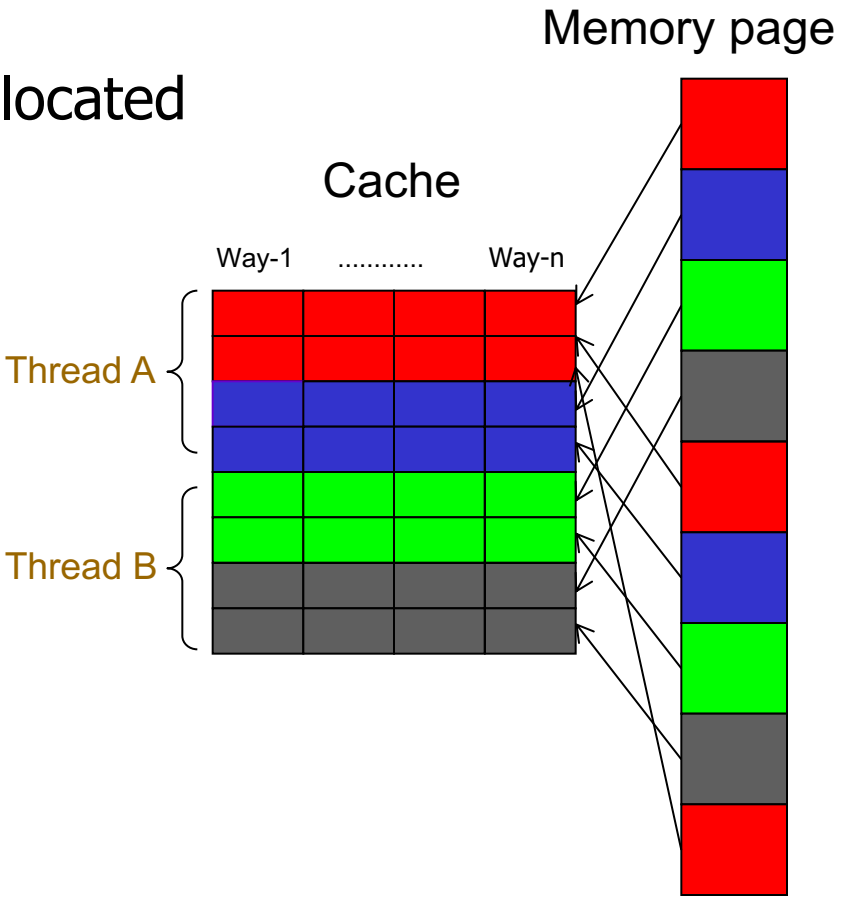
OS Based Cache Partitioning

- 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 number 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

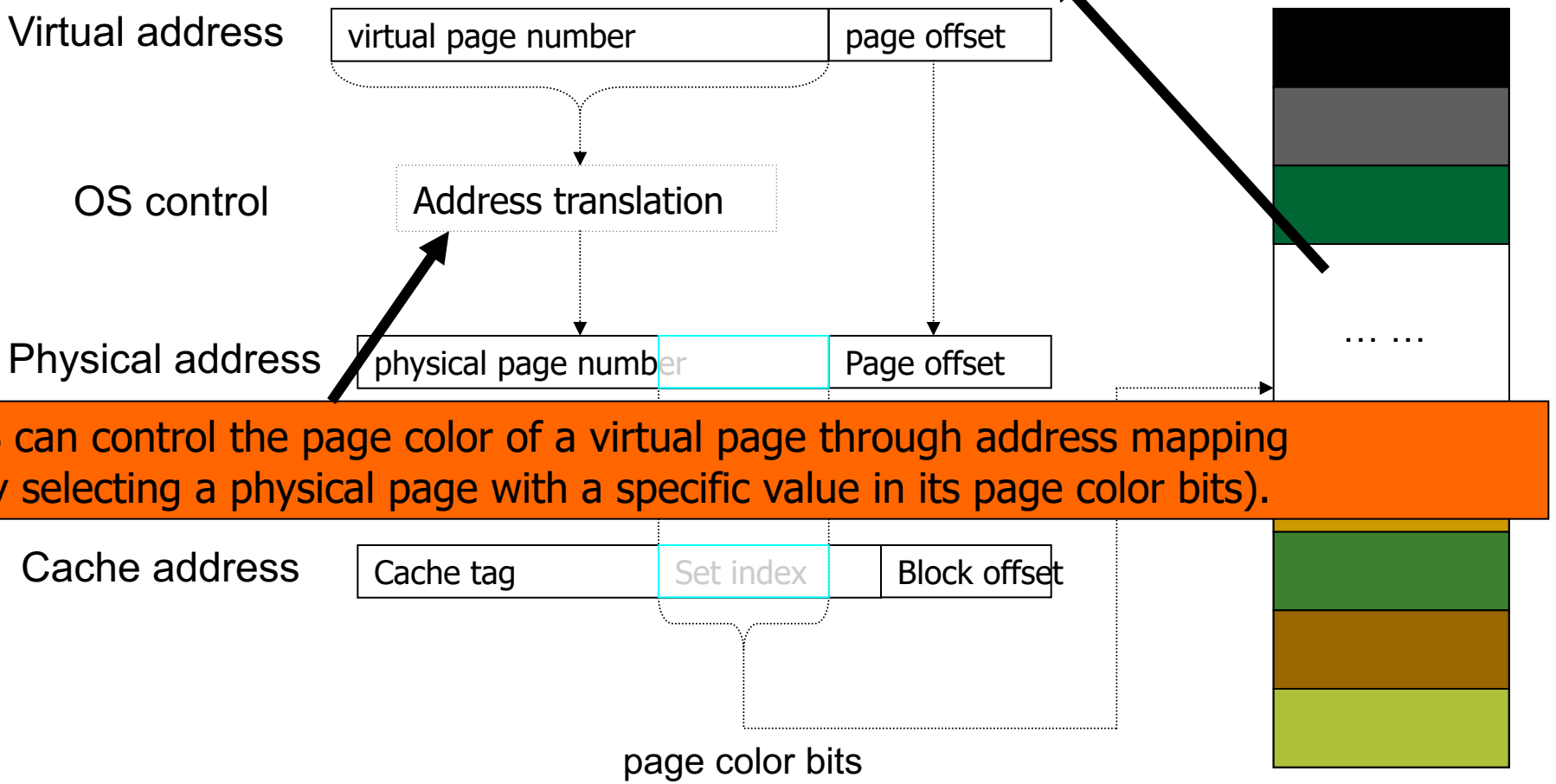
Page Coloring

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



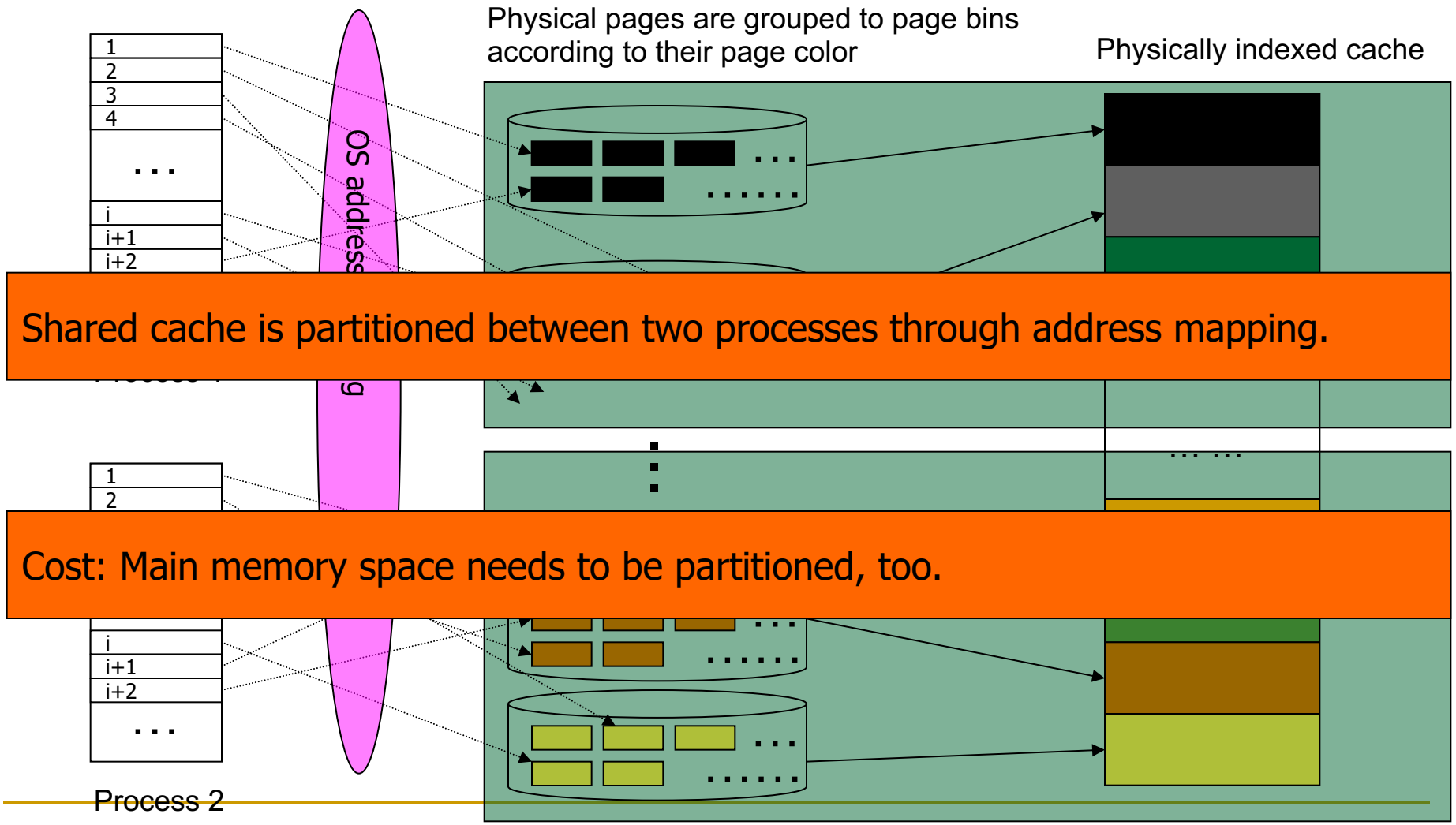
Page Coloring

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

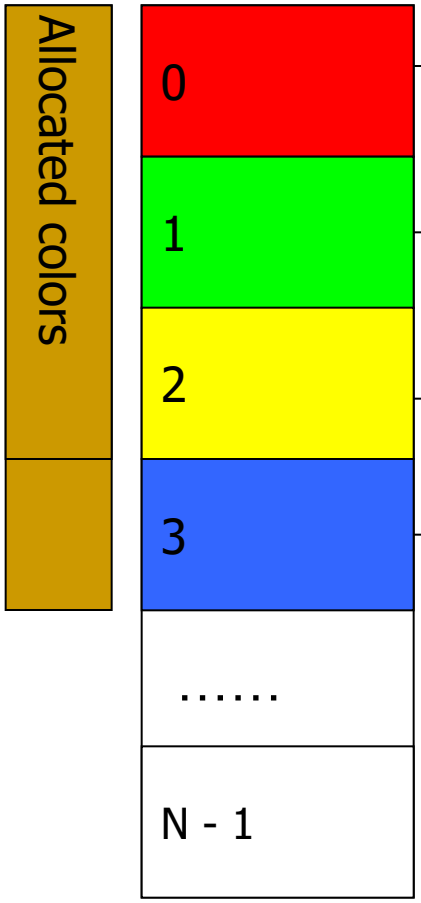


OS can control the page color of a virtual page through address mapping (by selecting a physical page with a specific value in its page color bits).

Static Cache Partitioning using Page Coloring



Dynamic Cache Partitioning via Page Re-Coloring

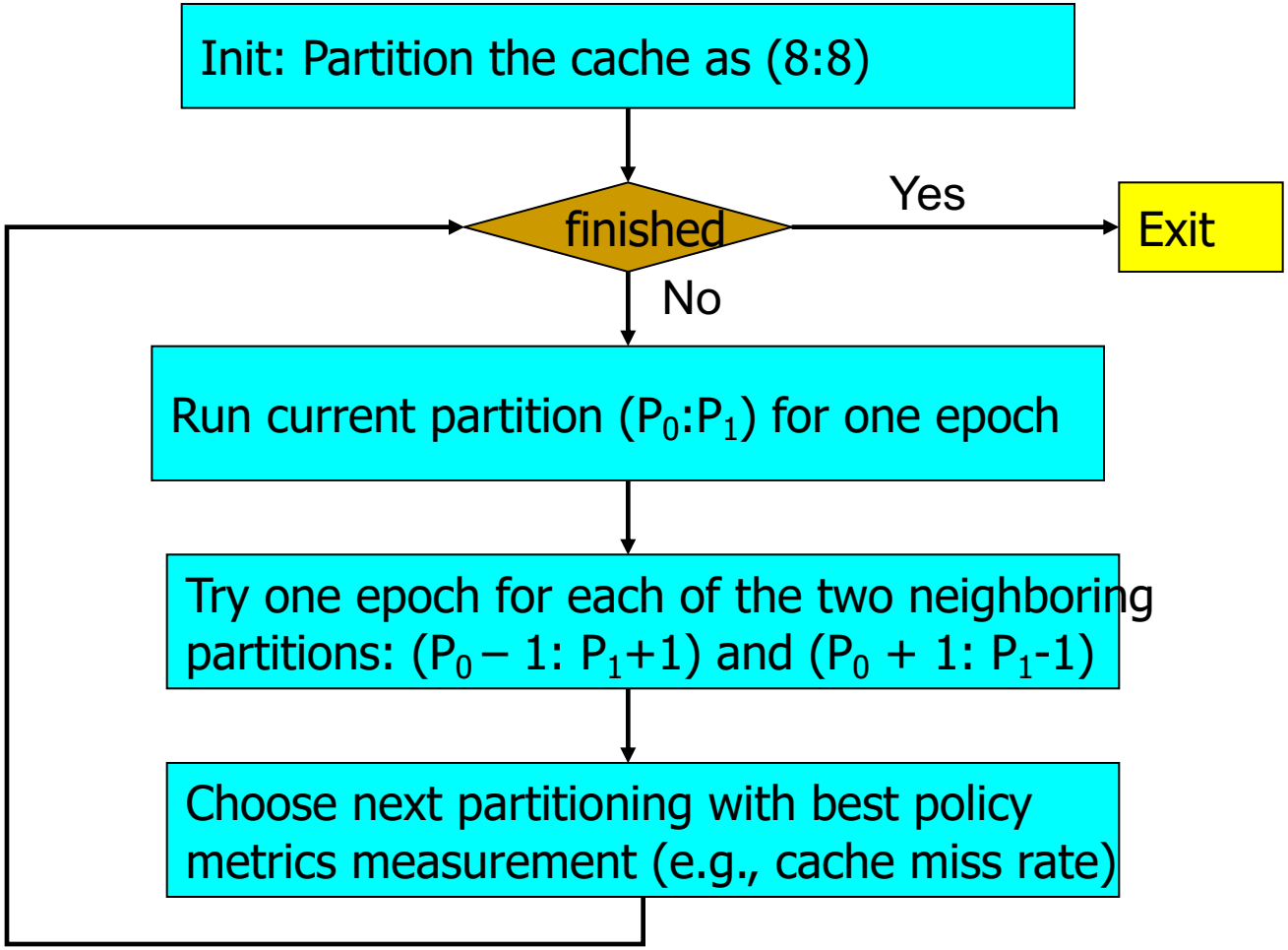


Pages of a process are organized into linked lists by their colors.

Memory allocation guarantees that pages are evenly distributed into all the lists (colors) to avoid hot points.

- Page re-coloring:
  - Allocate page in new color
  - Copy memory contents
  - Free old page

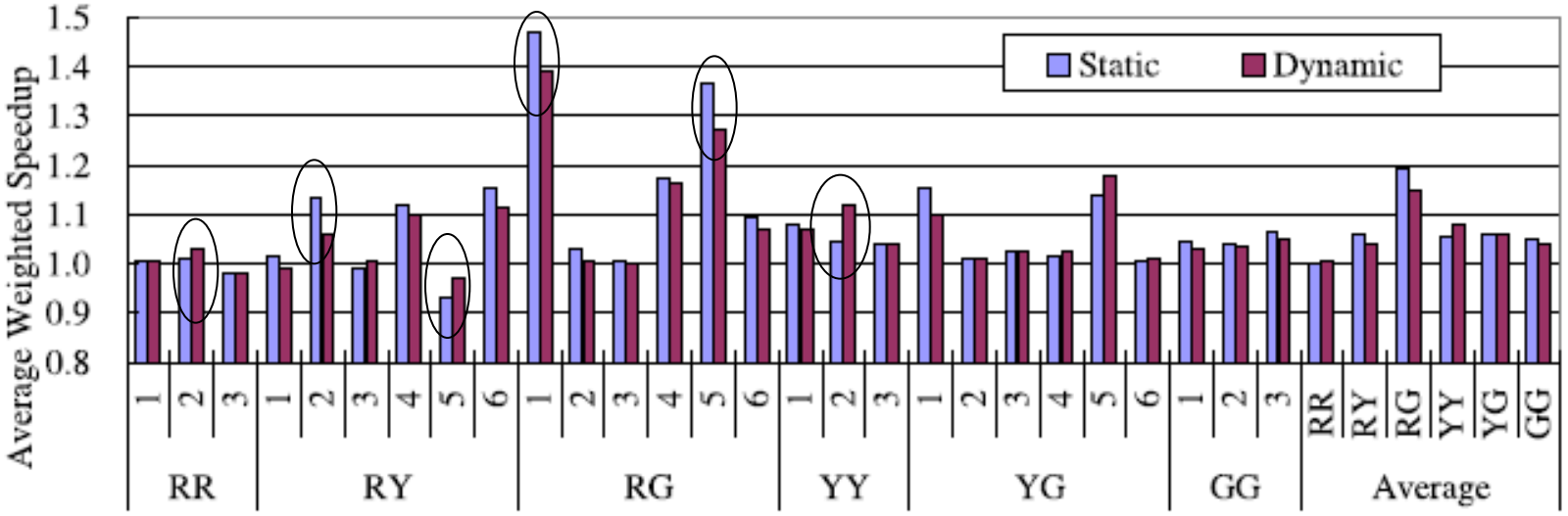
Dynamic Partitioning in a Dual-Core System



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

Lin et al., "Gaining Insights into Multi-Core Cache Partitioning: Bridging the Gap between Simulation and Real Systems," HPCA 2008.

Software vs. Hardware Cache Management

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

- Disadvantages
  - Large granularity of partitioning (page-based versus 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 may be exposed to software (e.g., number of misses due to inter-thread conflict)

Computer Architecture

Lecture 18b:

Multi-Core Cache Management

Prof. Onur Mutlu

ETH Zürich

Fall 2018

22 November 2018

We did not cover the following slides in lecture. These are for your preparation for the next lecture.

Private/Shared Caching

Private/Shared Caching

- Goal: Achieve the benefits of private caches (low latency, performance isolation) while sharing cache capacity across cores

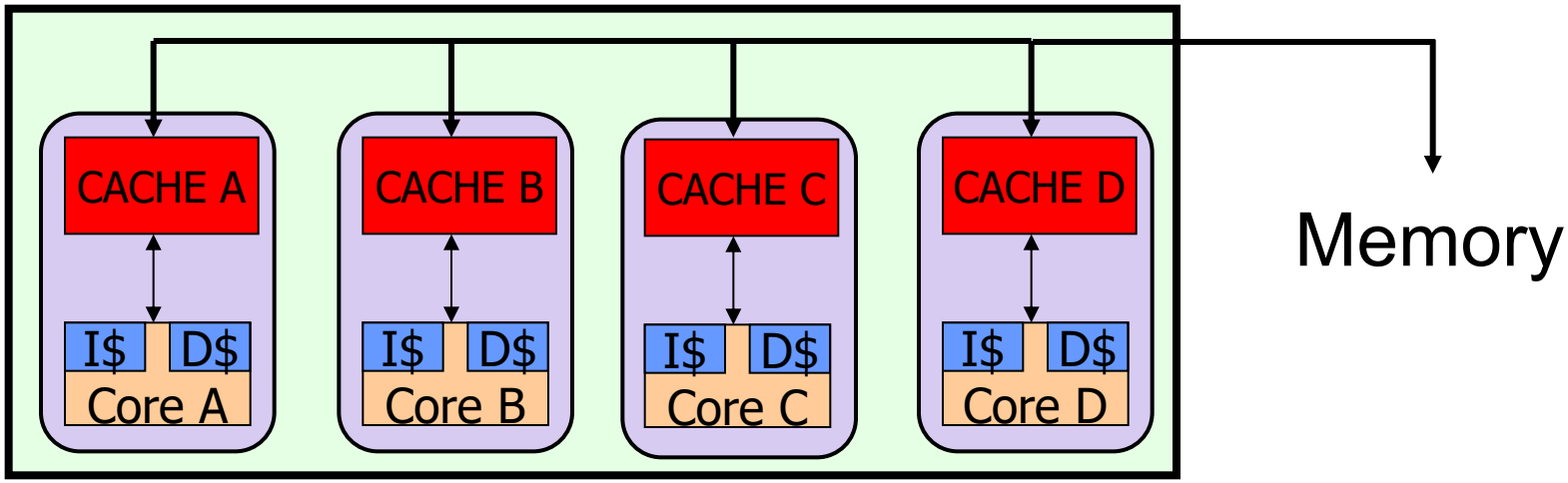
- Example: Adaptive spill/receive caching

- 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 Multi-Core

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

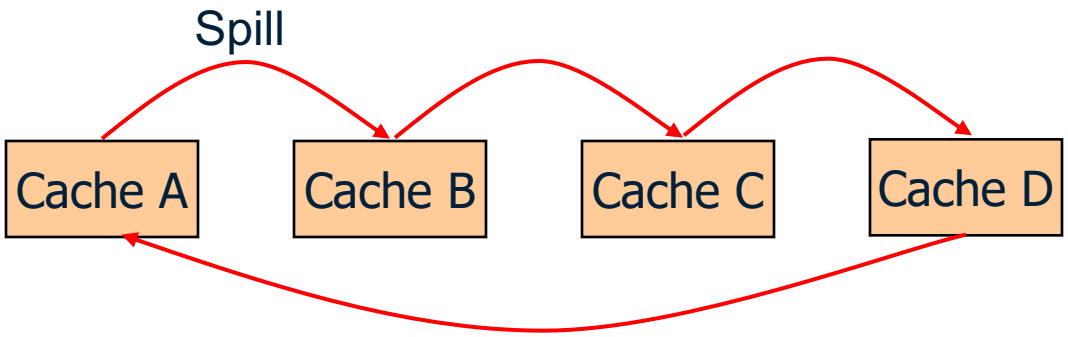


Problem: When one core needs more cache and other core has spare cache, private-cache based systems cannot share capacity

Cache Line Spilling – Cooperative Caching

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

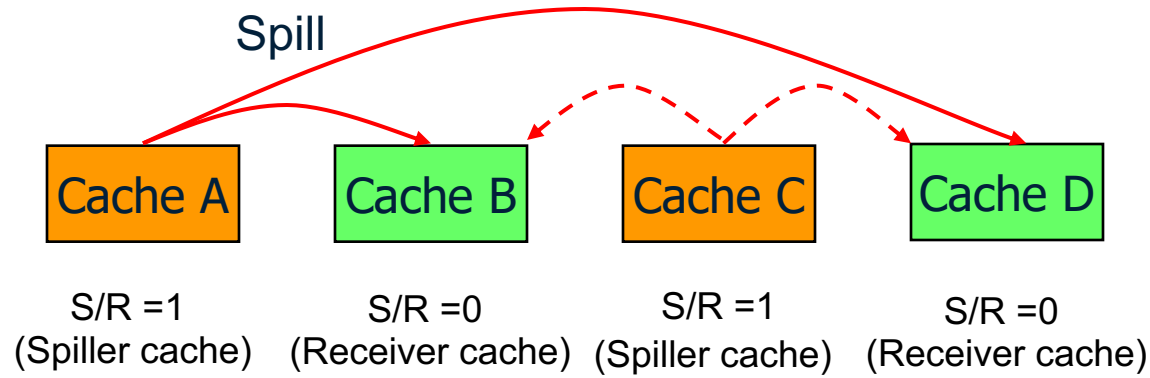
Goal: Robust High-Performance Capacity Sharing with Negligible Overhead

Chang and Sohi, "Cooperative Caching for Chip Multiprocessors," ISCA 2006.

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)

Qureshi, "Adaptive Spill-Receive for Robust High-Performance Caching in CMPs," HPCA 2009.

Dynamic Spill-Receive via "Set Dueling"

Divide the cache in three:

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

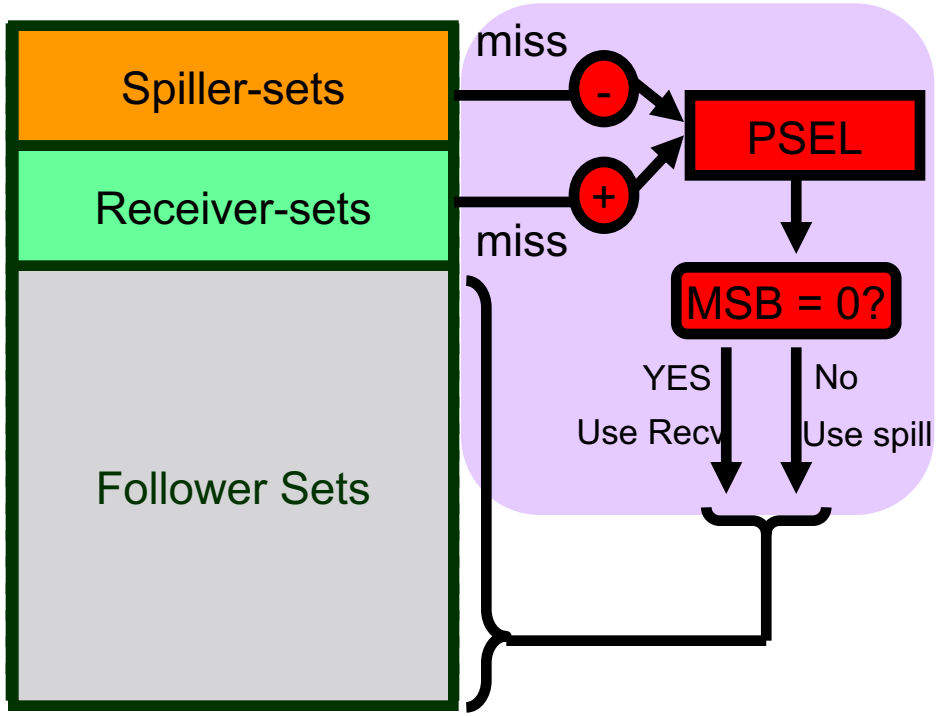
n-bit PSEL counter

misses to spiller-sets: PSEL--

misses to receiver-set: PSEL++

MSB of PSEL decides policy for Follower sets:

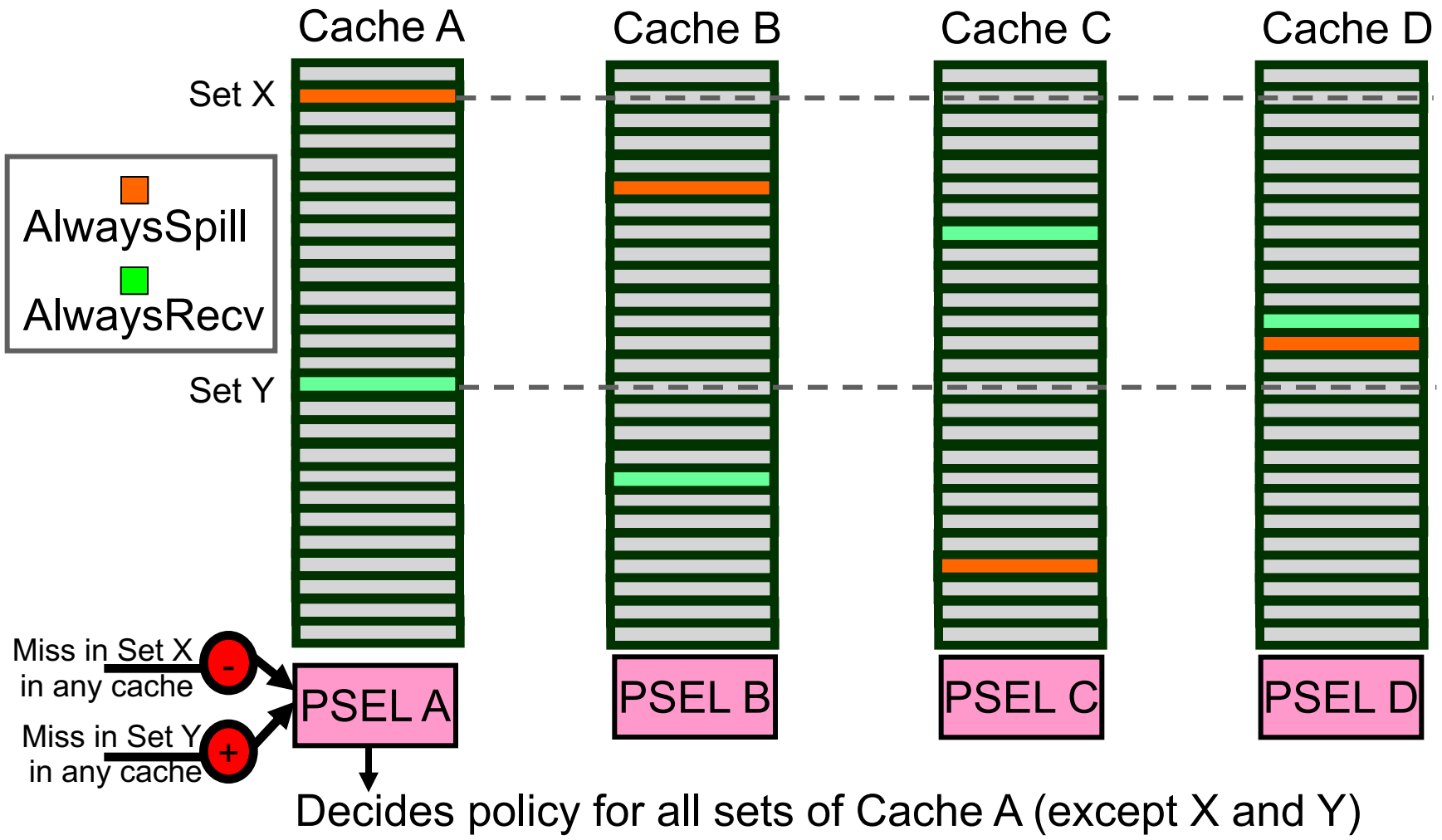
- MSB = 0, Use spill
- MSB = 1, Use receive



monitor ➔ choose ➔ apply
(using a single counter)

Dynamic Spill-Receive Architecture

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



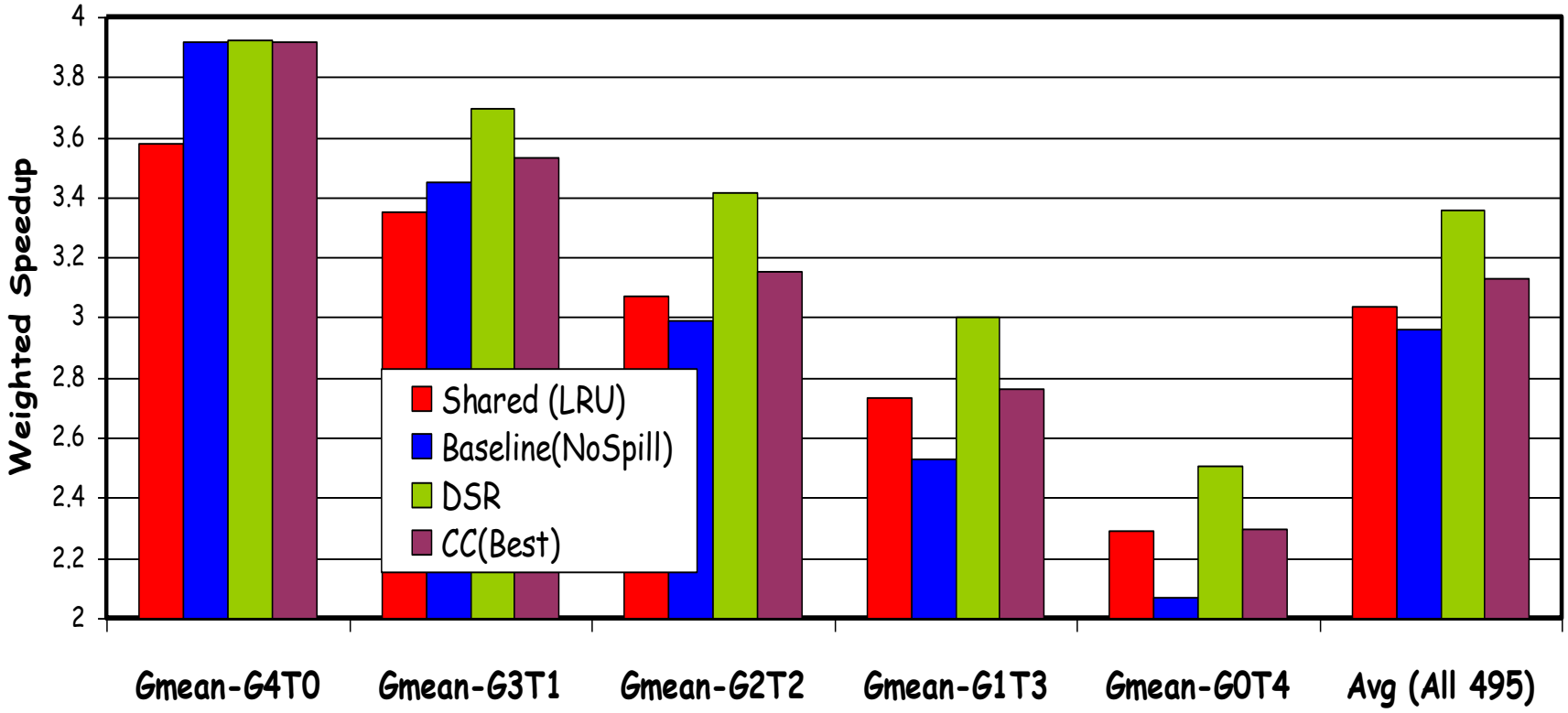
Experimental Setup

- 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 Weighted Speedup



On average, DSR improves weighted speedup by 13%

Distributed Caches

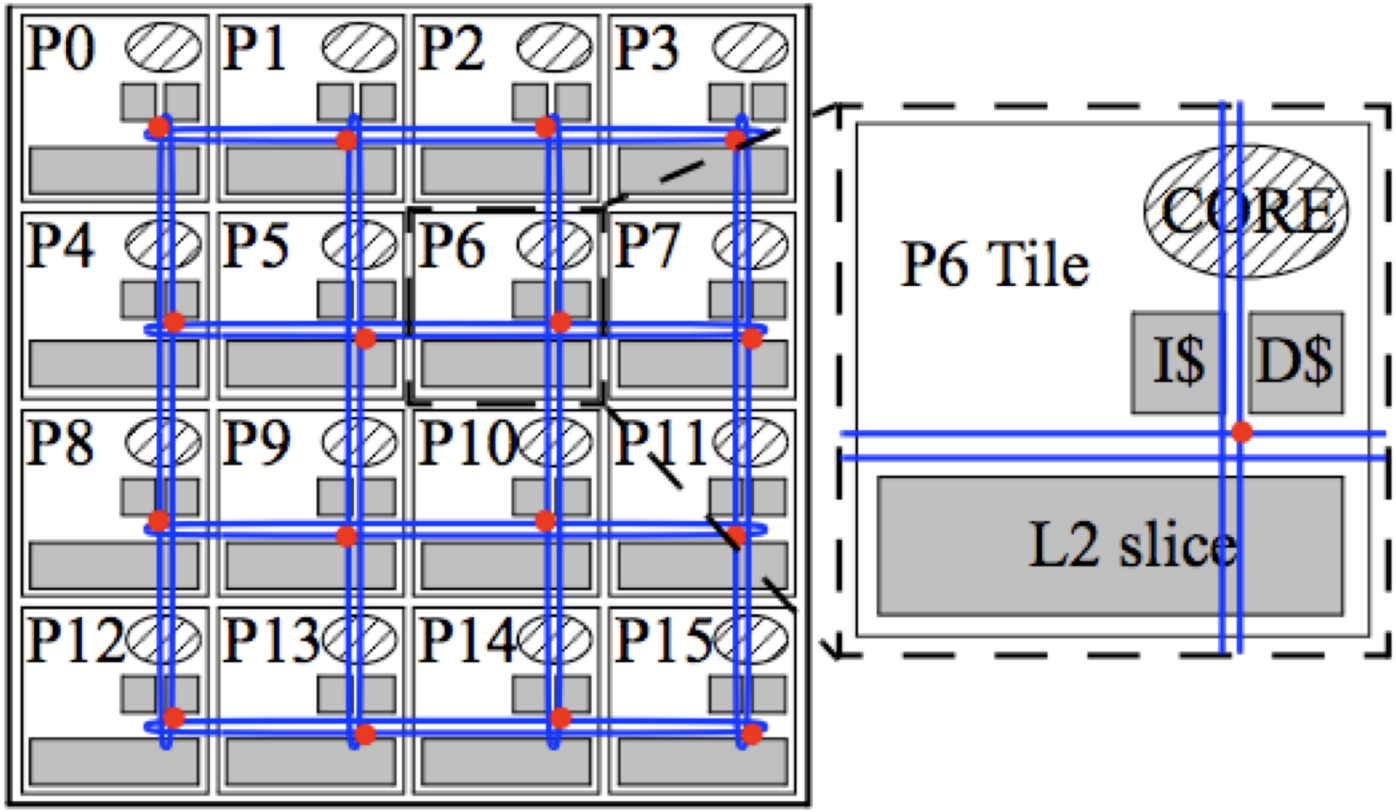
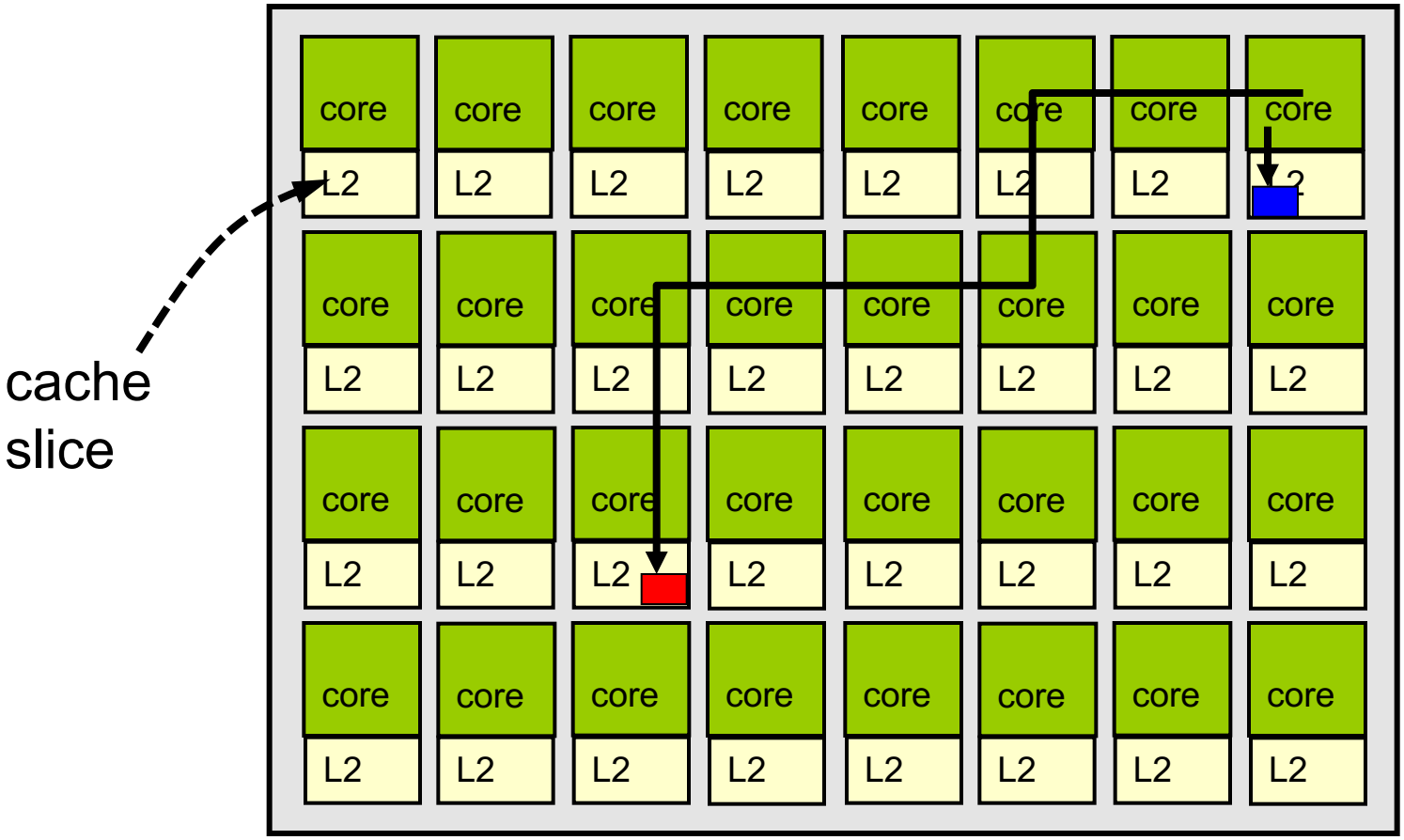


FIGURE 1. Typical tiled architecture. Tiles are interconnected into a 2-D folded torus. Each tile contains a core, L1 instruction and data caches, a shared-L2 cache slice, and a router/switch.

Caching for Parallel Applications



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

Handling Shared Data in Private Caches

- Shared data and locks ping-pong between processors if caches are private
  - -- Increases latency to fetch shared data/locks
  - -- Reduces cache efficiency (many invalid blocks)
  - -- Scalability problem: maintaining coherence across a large number of private caches is costly

- How to do better?
  - Idea: Store shared data and locks only in one special core's cache. Divert all critical section execution to that core/cache.
    - Essentially, a specialized core for processing critical sections
    - Suleman et al., "Accelerating Critical Section Execution with Asymmetric Multi-Core Architectures," ASPLOS 2009.

Non-Uniform Cache Access

- Problem: Large caches take a long time to access
- Wire delay
  - Closeby blocks can be accessed faster, but furthest blocks determine the worst-case access time

- Idea: Variable latency access time in a single cache
- Partition cache into pieces
  - Each piece has different latency
  - Which piece does an address map to?
    - Static: based on bits in address
    - Dynamic: any address can map to any piece
      - How to locate an address?
      - Replacement and placement policies?
- Kim et al., "An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches," ASPLOS 2002.

Multi-Core Cache Efficiency: Bandwidth Filters

- Caches act as a filter that reduce memory bandwidth requirement
  - Cache hit: No need to access memory
  - This is in addition to the latency reduction benefit of caching
  - GPUs use caches to reduce memory BW requirements

- Efficient utilization of cache space becomes more important with multi-core
  - Memory bandwidth is more valuable
    - Pin count not increasing as fast as # of transistors
      - 10%/year vs. 2x every 2 years
  - More cores put more pressure on the memory bandwidth

- How to make the bandwidth filtering effect of caches better?

Efficient Cache Utilization

Efficient Cache Utilization: Examples

- Qureshi et al., "A Case for MLP-Aware Cache Replacement," ISCA 2005.
- Qureshi et al., "Adaptive Insertion Policies for High Performance Caching," ISCA 2007.
- 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.

Cache Compression

Motivation for Cache Compression

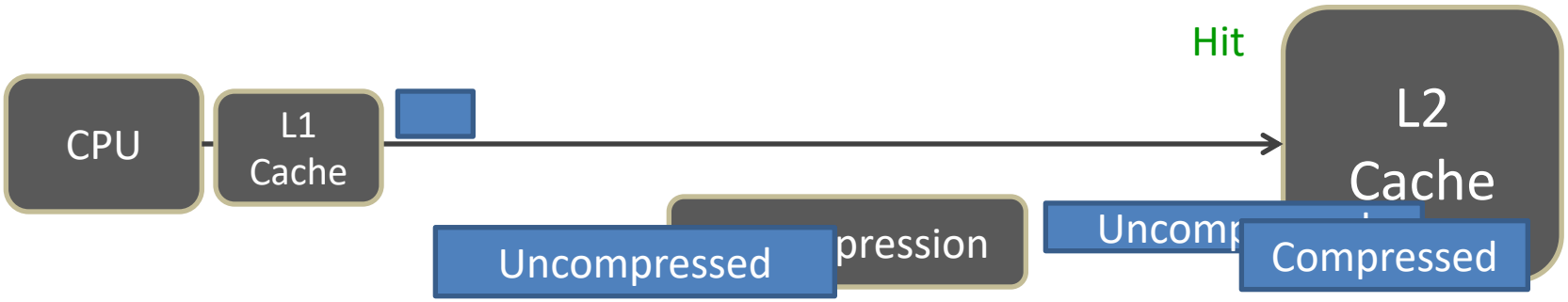
Significant redundancy in data:

0x**00000000**	0x**0000000**B	0x**00000003**	0x**00000004**	…

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)

Summary of Major Works

Compression Mechanisms	Decompression Latency	Complexity	Compression Ratio
Zero	✓	✓	✗

Summary of Major Works

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

Summary of Major Works

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

Summary of Major Works

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

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

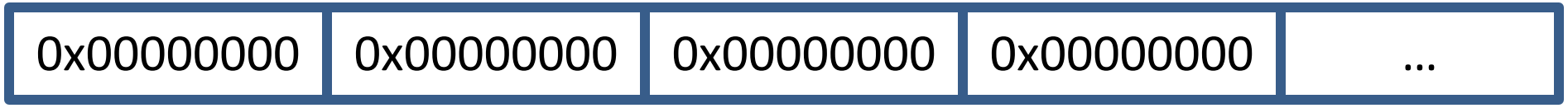
- Off-chip memory latency is high
  - Large caches can help, **but** at significant cost
- Compressing data in cache enables larger cache at low cost
- **Problem**: Decompression is on the execution critical path
- **Goal**: Design a new compression scheme that has

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

Key Data Patterns in Real Applications

Zero Values: initialization, sparse matrices, NULL pointers



Repeated Values: common initial values, adjacent pixels

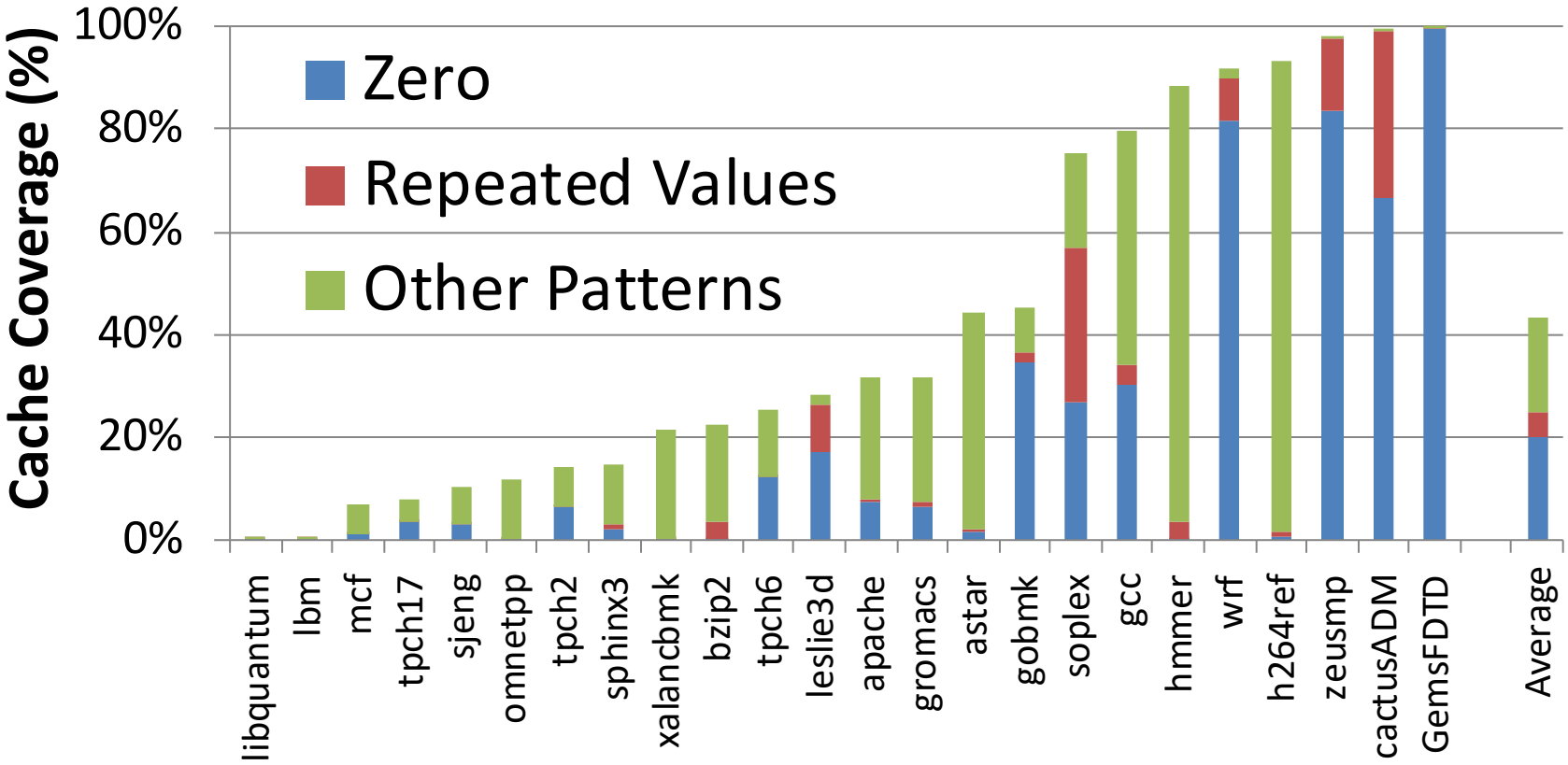
Narrow Values: small values stored in a big data type

Other Patterns: pointers to the same memory region

How Common Are These Patterns?

SPEC2006, databases, web workloads, 2MB L2 cache

"Other Patterns" include Narrow Values



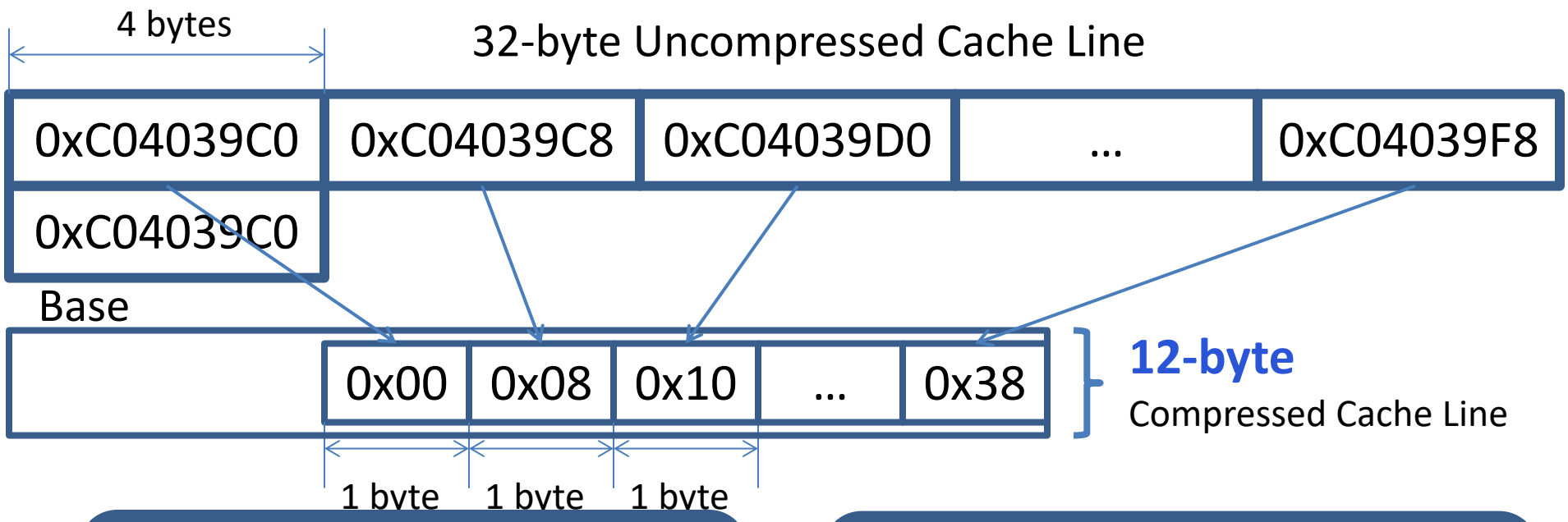
43% of the cache lines belong to key patterns

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

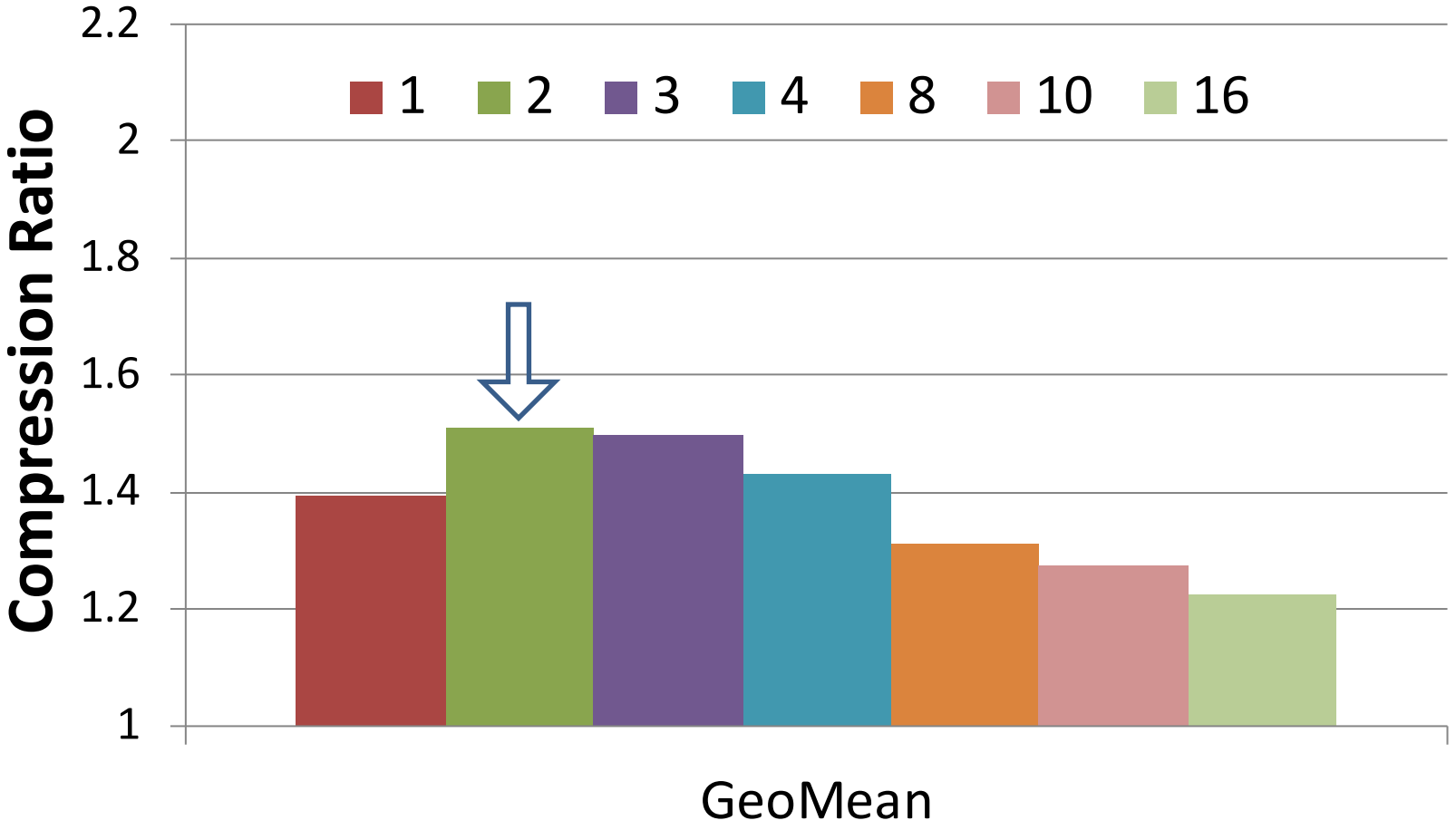
Can We Do Better?

- Uncompressible cache line (with a single base):

0x00000000	**0x09A4**0178	**0x000000**0B	**0x09A4**A838	…

- **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**

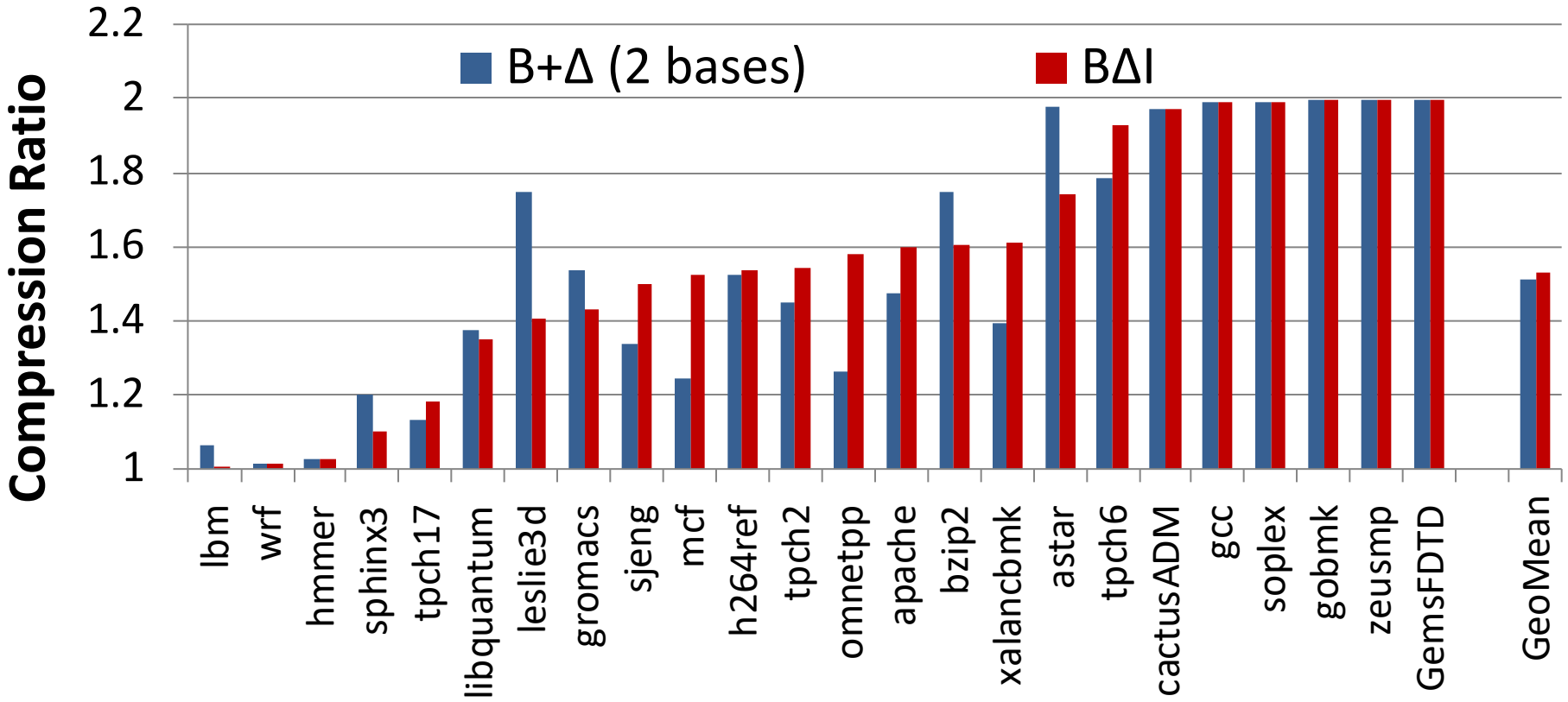
Advantages over 2 arbitrary bases:

– Better compression ratio

– Simpler compression logic

Base-Delta-Immediate (BΔI) Compression

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

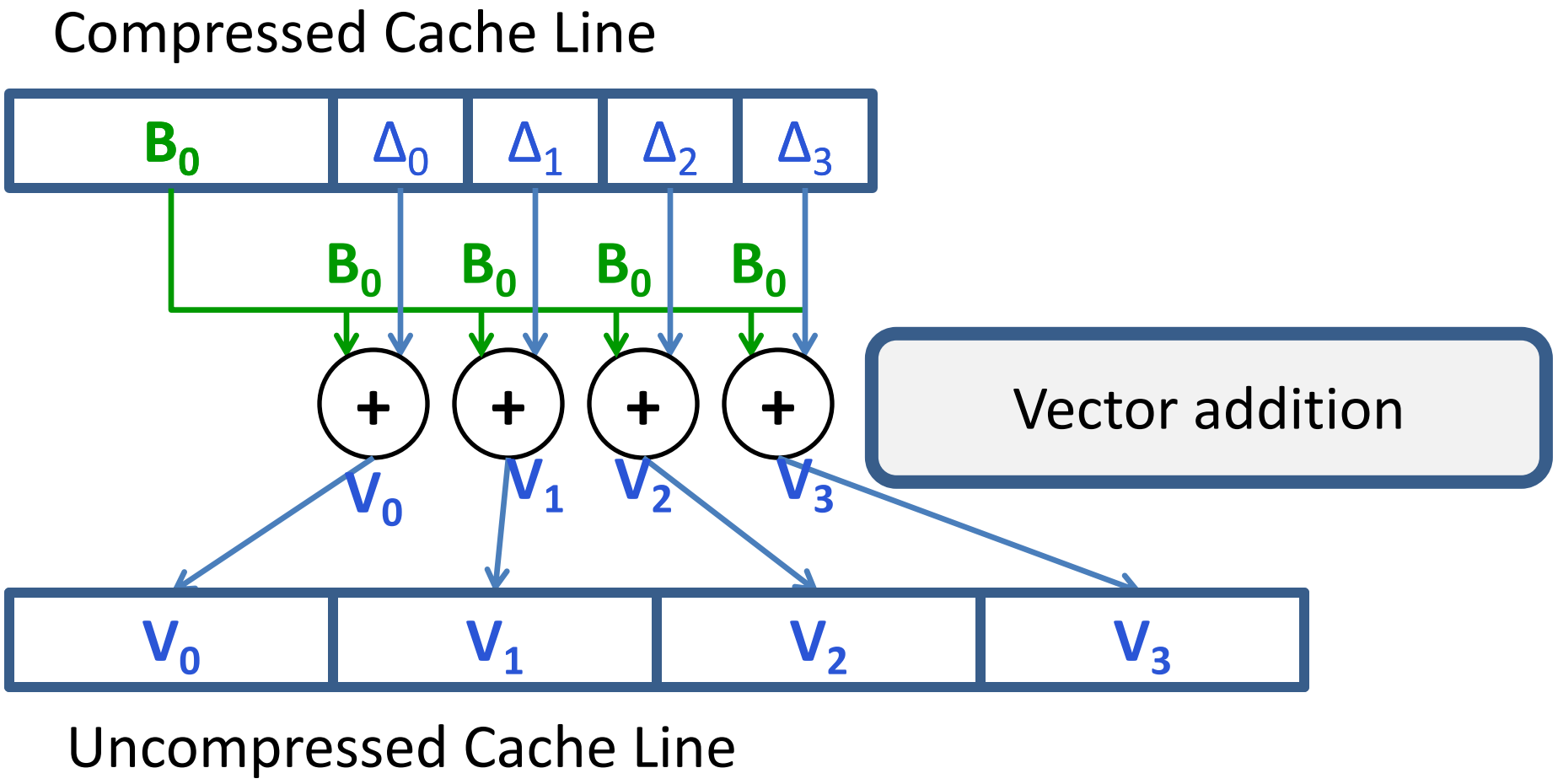


Average compression ratio is close, but **BΔI** is **simpler**

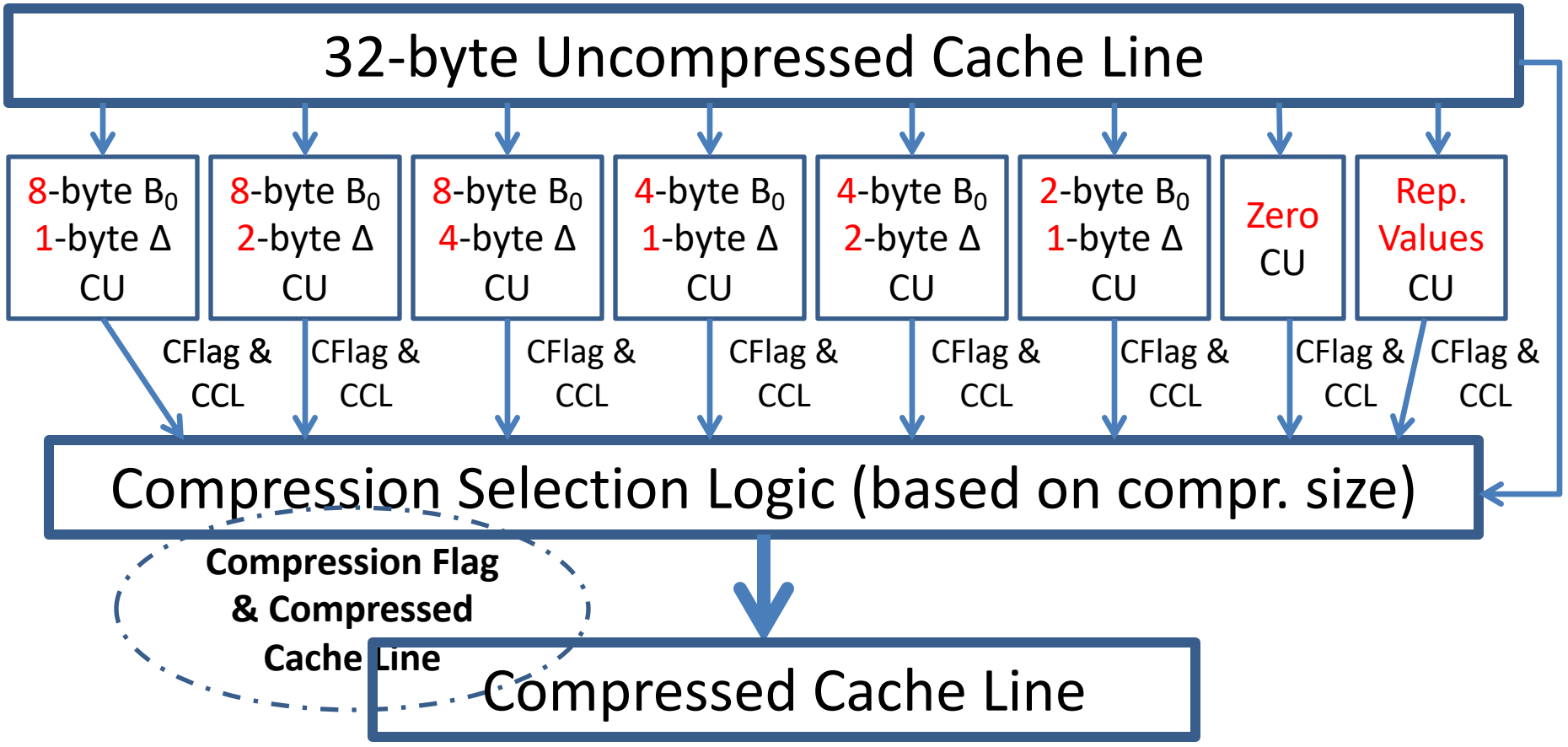
BΔI Cache Compression Implementation

- **Decompressor Design**
  - Low latency
- **Compressor Design**
  - Low cost and complexity
- **BΔI Cache Organization**
  - Modest complexity

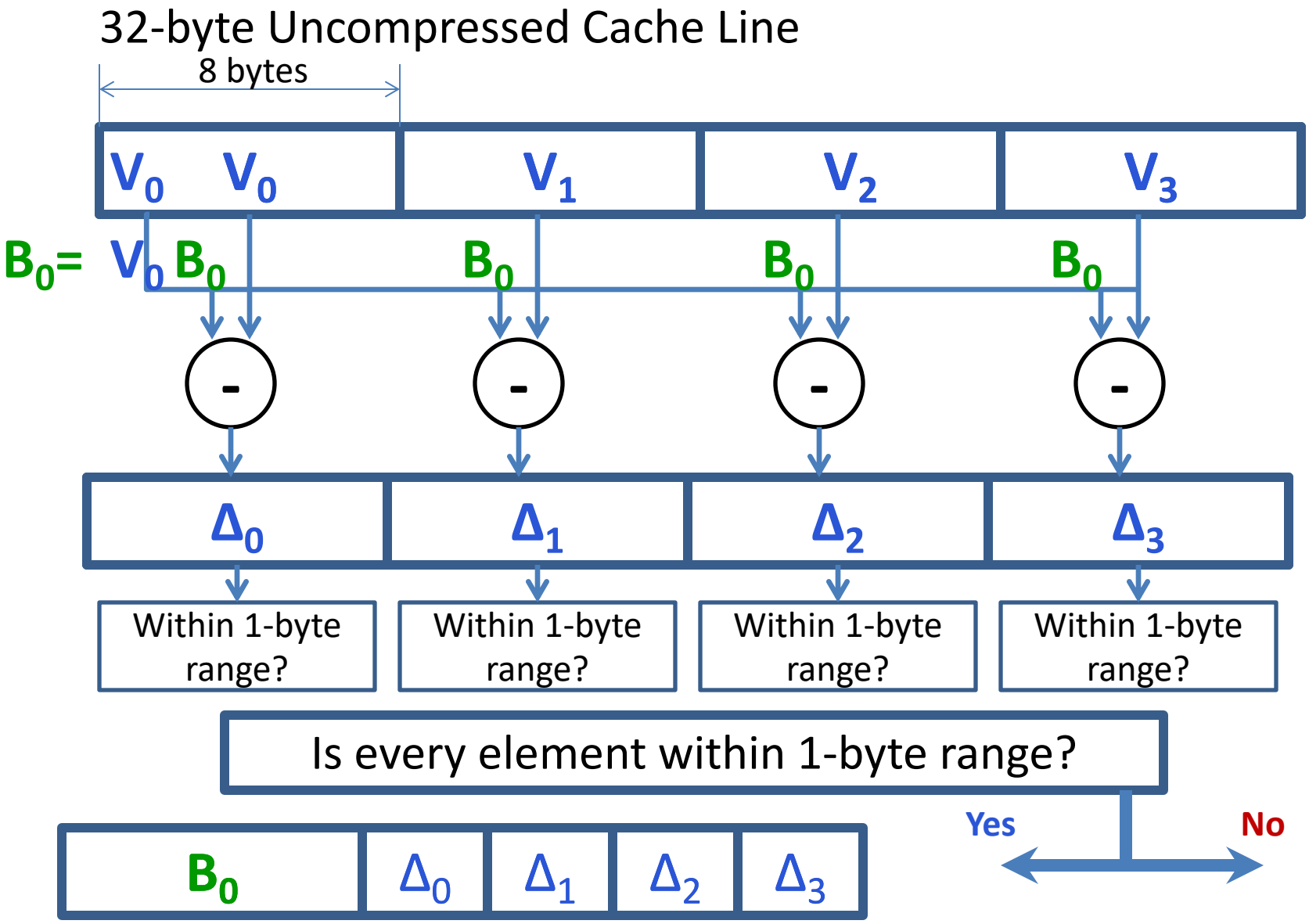
BΔI Decompressor Design



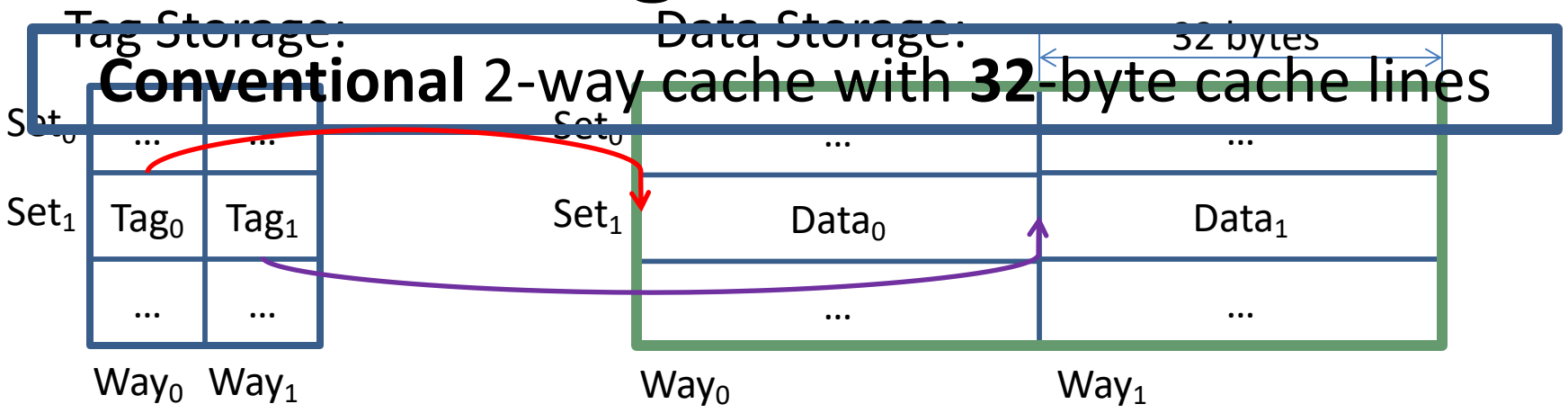
BΔI Compressor Design

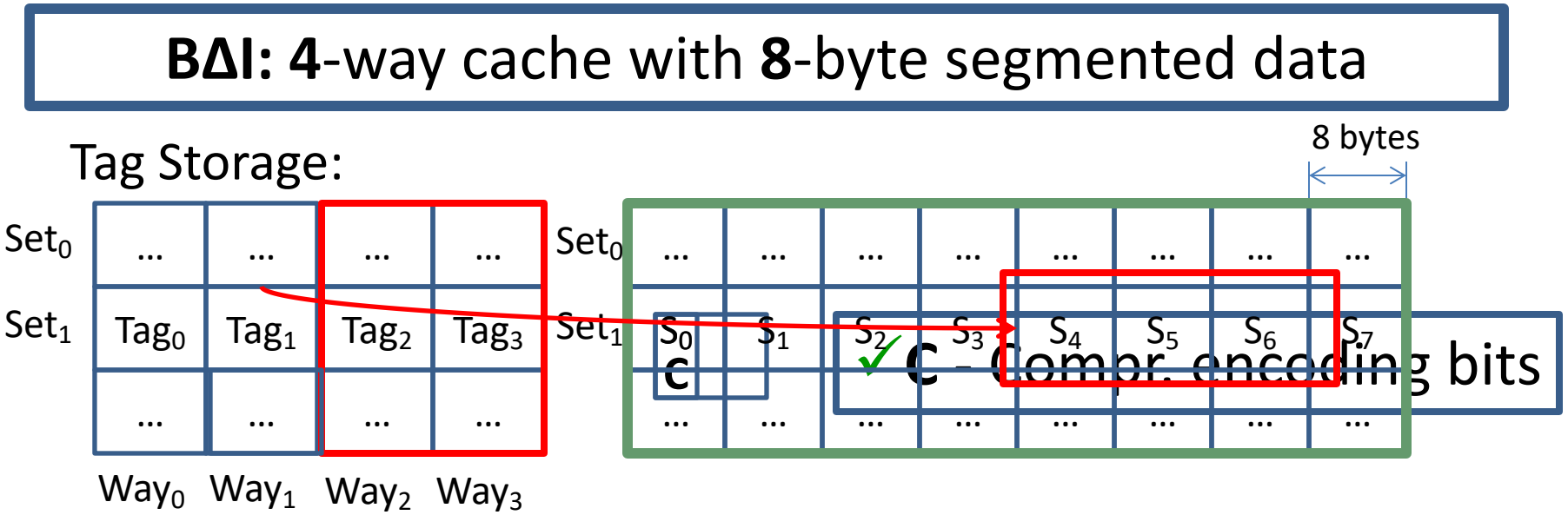


BΔI Compression Unit: 8-byte B_0 1-byte Δ



BΔI Cache Organization

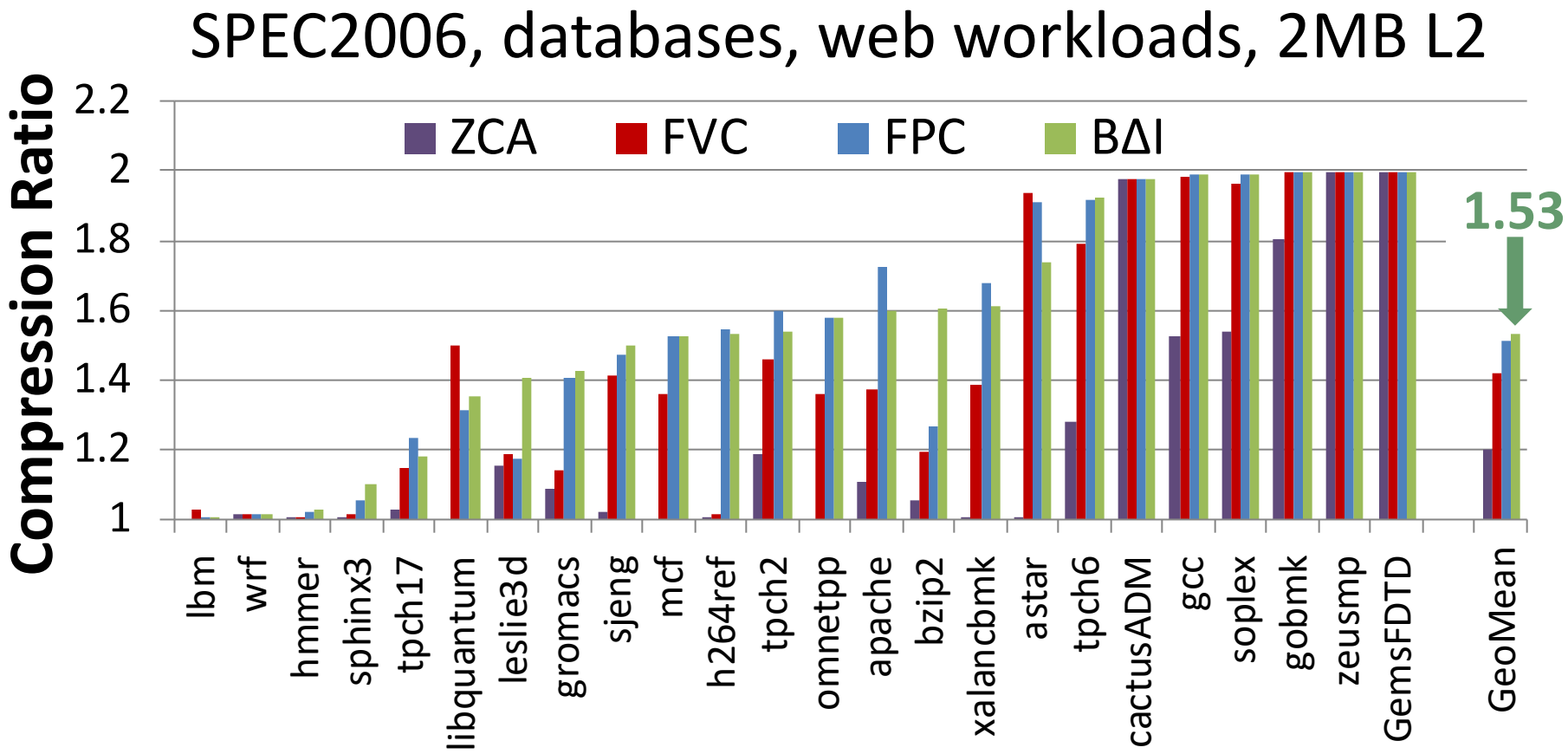




Qualitative Comparison with Prior Work

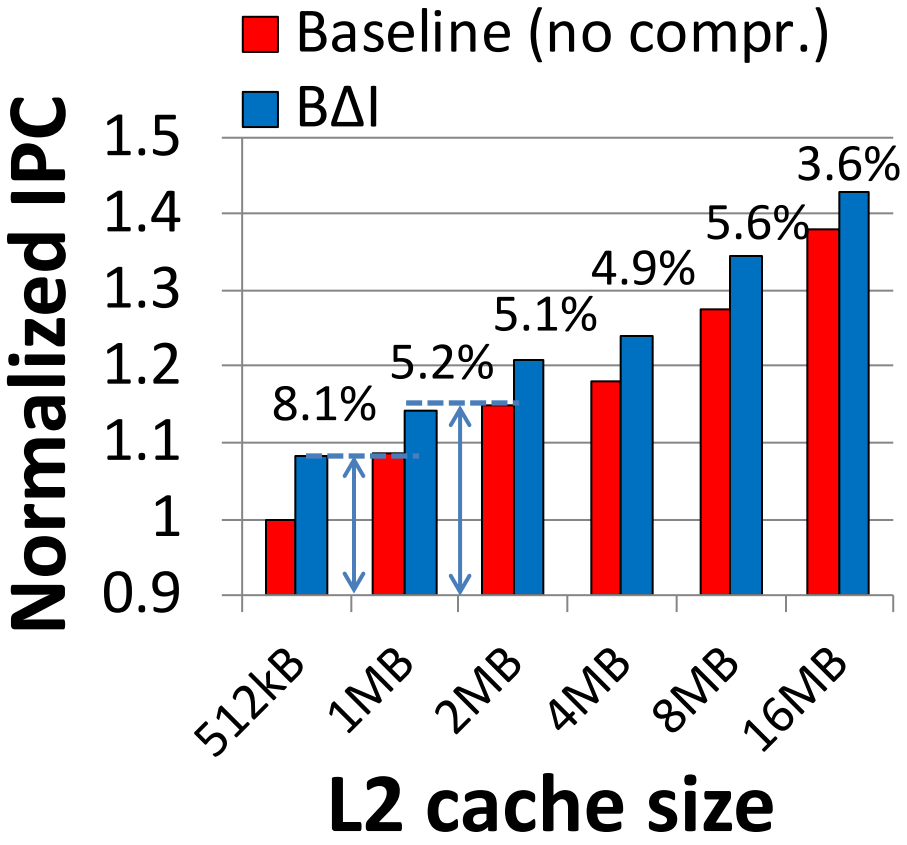
- **Zero-based designs**
  - ZCA *[Dusser+, ICS'09]*: zero-content augmented cache
  - ZVC *[Islam+, PACT'09]*: zero-value cancelling
  - Limited applicability (only zero values)
- **FVC** *[Yang+, MICRO'00]*: frequent value compression
  - High decompression latency and complexity
- **Pattern-based 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)

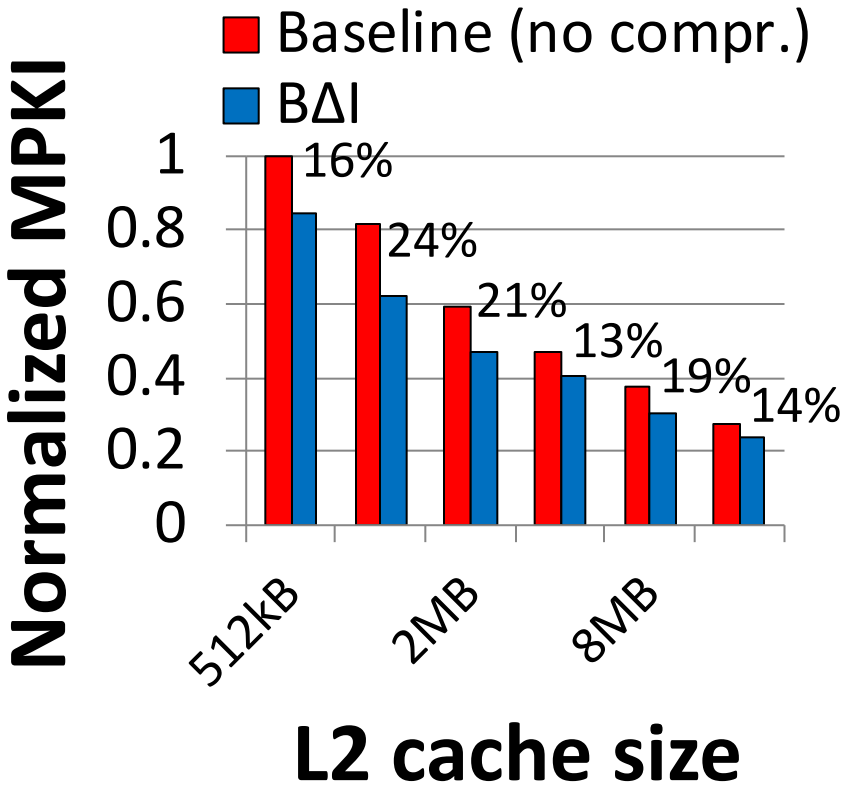
Cache Compression Ratios



BΔI 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)

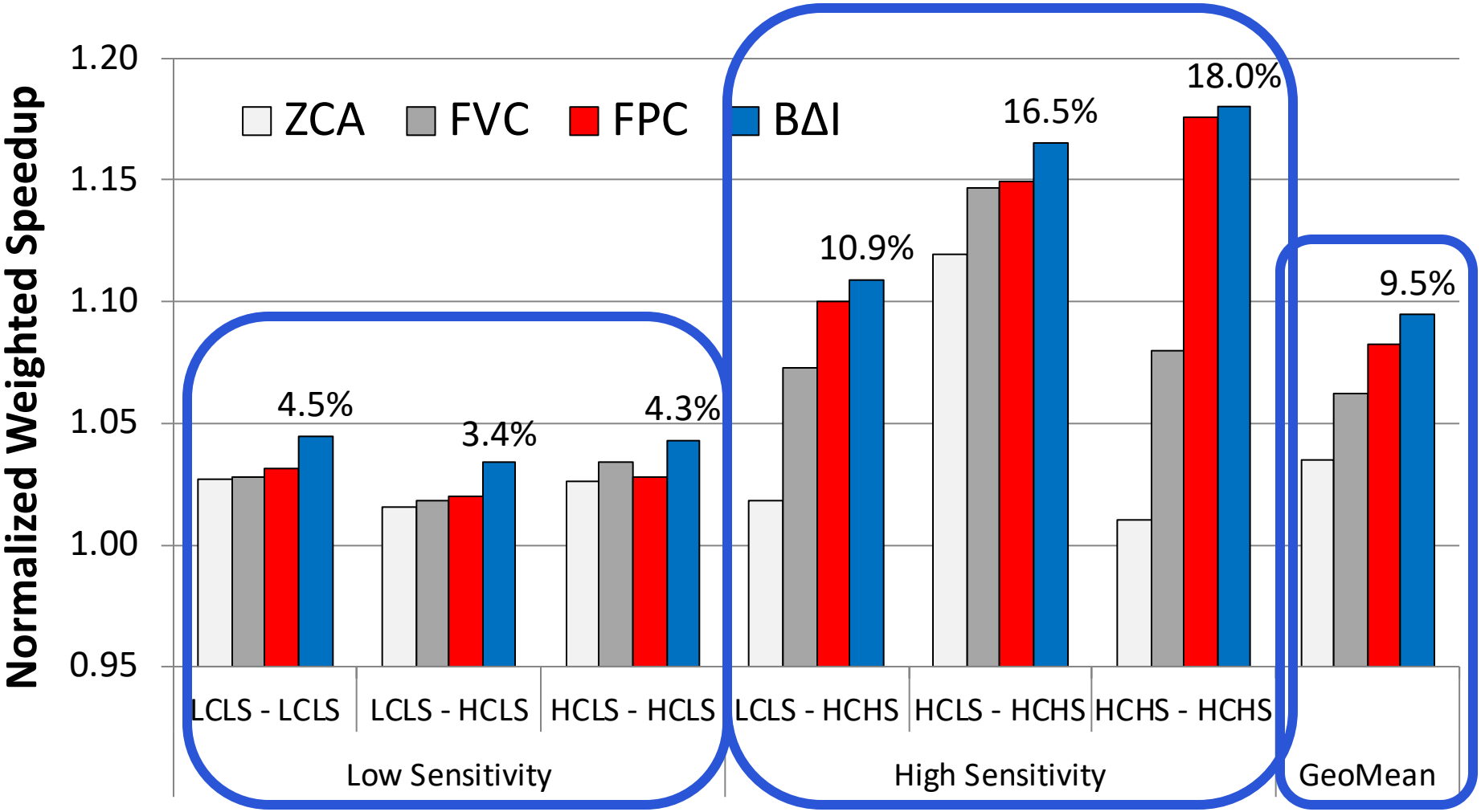
  Sensitivity: performance gain with more cache

  (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 performance improves

BΔI performance improvement is the highest (9.5%)

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

Readings on Memory Compression (I)

- 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 International Conference on Parallel Architectures and Compilation Techniques (***PACT***), Minneapolis, MN, September 2012. Slides (pptx) Source Code

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

Gennady Pekhimenko† gpekhime@cs.cmu.edu

Vivek Seshadri† vseshadr@cs.cmu.edu

Onur Mutlu† onur@cmu.edu

Michael A. Kozuch* michael.a.kozuch@intel.com

Phillip B. Gibbons* phillip.b.gibbons@intel.com

Todd C. Mowry† tcm@cs.cmu.edu

†Carnegie Mellon University *Intel Labs Pittsburgh

Readings on Memory Compression (II)

- Gennady Pekhimenko, Vivek Seshadri, Yoongu Kim, Hongyi Xin, Onur Mutlu, Michael A. Kozuch, Phillip B. Gibbons, and Todd C. Mowry, **"Linearly Compressed Pages: A Low-Complexity, Low-Latency Main Memory Compression Framework"**
  *Proceedings of the 46th International Symposium on Microarchitecture (**MICRO**)*, Davis, CA, December 2013. [Slides (pptx) (pdf)] [Lightning Session Slides (pptx) (pdf)] Poster (pptx) (pdf)]

Linearly Compressed Pages: A Low-Complexity, Low-Latency Main Memory Compression Framework

Gennady Pekhimenko† gpekhime@cs.cmu.edu
Vivek Seshadri† vseshadr@cs.cmu.edu
Yoongu Kim† yoongukim@cmu.edu
Hongyi Xin† hxin@cs.cmu.edu
Onur Mutlu† onur@cmu.edu
Phillip B. Gibbons⋆ phillip.b.gibbons@intel.com
Michael A. Kozuch⋆ michael.a.kozuch@intel.com
Todd C. Mowry† tcm@cs.cmu.edu
†Carnegie Mellon University ⋆Intel Labs Pittsburgh

Readings on Memory Compression (III)

- Gennady Pekhimenko, Tyler Huberty, Rui Cai, Onur Mutlu, Phillip P. Gibbons, Michael A. Kozuch, and Todd C. Mowry,
  "Exploiting Compressed Block Size as an Indicator of Future Reuse"
  *Proceedings of the 21st International Symposium on High-Performance Computer Architecture (**HPCA**)*, Bay Area, CA, February 2015.
  [Slides (pptx) (pdf)]

Exploiting Compressed Block Size as an Indicator of Future Reuse

Gennady Pekhimenko† *gpekhime@cs.cmu.edu*
Tyler Huberty† *thuberty@alumni.cmu.edu*
Rui Cai† *rcai@alumni.cmu.edu*
Onur Mutlu† *onur@cmu.edu*
Phillip B. Gibbons* *phillip.b.gibbons@intel.com*
Michael A. Kozuch* *michael.a.kozuch@intel.com*
Todd C. Mowry† *tcm@cs.cmu.edu*
†Carnegie Mellon University *Intel Labs Pittsburgh

Readings on Memory Compression (IV)

- Gennady Pekhimenko, Evgeny Bolotin, Nandita Vijaykumar, Onur Mutlu, Todd C. Mowry, and Stephen W. Keckler,
  "A Case for Toggle-Aware Compression for GPU Systems"
  *Proceedings of the 22nd International Symposium on High-Performance Computer Architecture (**HPCA**)*, Barcelona, Spain, March 2016.
  [Slides (pptx) (pdf)]

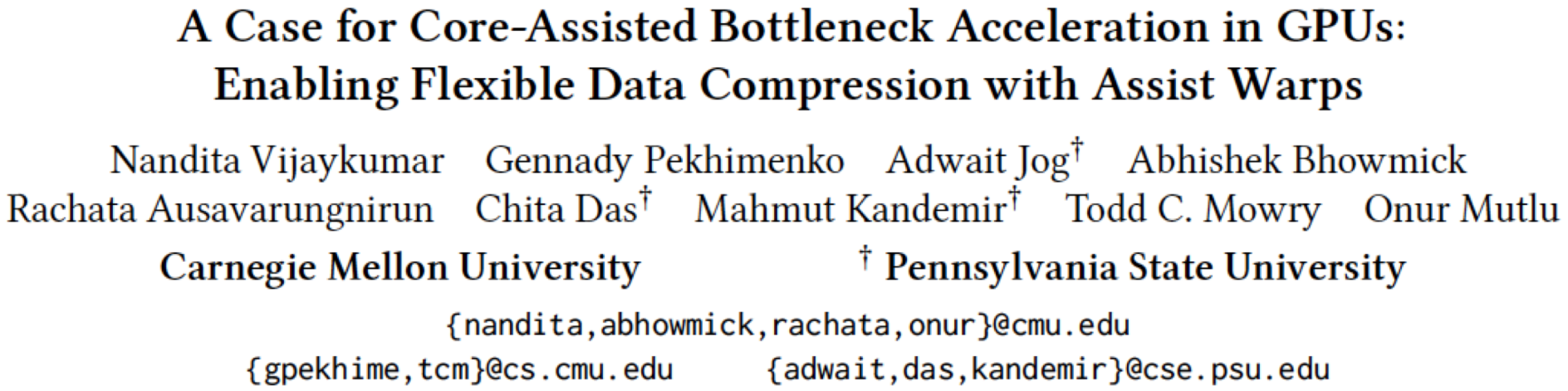
A Case for Toggle-Aware Compression for GPU Systems

Gennady Pekhimenko[†], Evgeny Bolotin[⋆], Nandita Vijaykumar[†],
Onur Mutlu[†], Todd C. Mowry[†], Stephen W. Keckler[⋆#]

[†]Carnegie Mellon University [⋆]NVIDIA [#]University of Texas at Austin

Readings on Memory Compression (VI)

- Nandita Vijaykumar, Gennady Pekhimenko, Adwait Jog, Abhishek Bhowmick, Rachata Ausavarungnirun, Chita Das, Mahmut Kandemir, Todd C. Mowry, and Onur Mutlu,
  "A Case for Core-Assisted Bottleneck Acceleration in GPUs: Enabling Flexible Data Compression with Assist Warps"
  *Proceedings of the 42nd International Symposium on Computer Architecture (**ISCA**)*, Portland, OR, June 2015.
  [Slides (pptx) (pdf)] [Lightning Session Slides (pptx) (pdf)]



A Case for Core-Assisted Bottleneck Acceleration in GPUs: Enabling Flexible Data Compression with Assist Warps

Nandita Vijaykumar Gennady Pekhimenko Adwait Jog† Abhishek Bhowmick
Rachata Ausavarungnirun Chita Das† Mahmut Kandemir† Todd C. Mowry Onur Mutlu
Carnegie Mellon University † Pennsylvania State University
{nandita,abhowmick,rachata,onur}@cmu.edu
{gpekhime,tcm}@cs.cmu.edu {adwait,das,kandemir}@cse.psu.edu

Efficient Cache Utilization: Examples

- Qureshi et al., "A Case for MLP-Aware Cache Replacement," ISCA 2005.
- Qureshi et al., "Adaptive Insertion Policies for High Performance Caching," ISCA 2007.
- 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.

Revisiting Cache Placement (Insertion)

- Is inserting a fetched/prefetched block into the cache (hierarchy) always a good idea?
  - No allocate on write: does not allocate a block on write miss
  - How about reads?

- Allocating on a read miss

  -- Evicts another potentially useful cache block

  + Incoming block potentially more useful

- Ideally:
  - we would like to place those blocks whose caching would be most useful in the future
  - we certainly do not want to cache never-to-be-used blocks

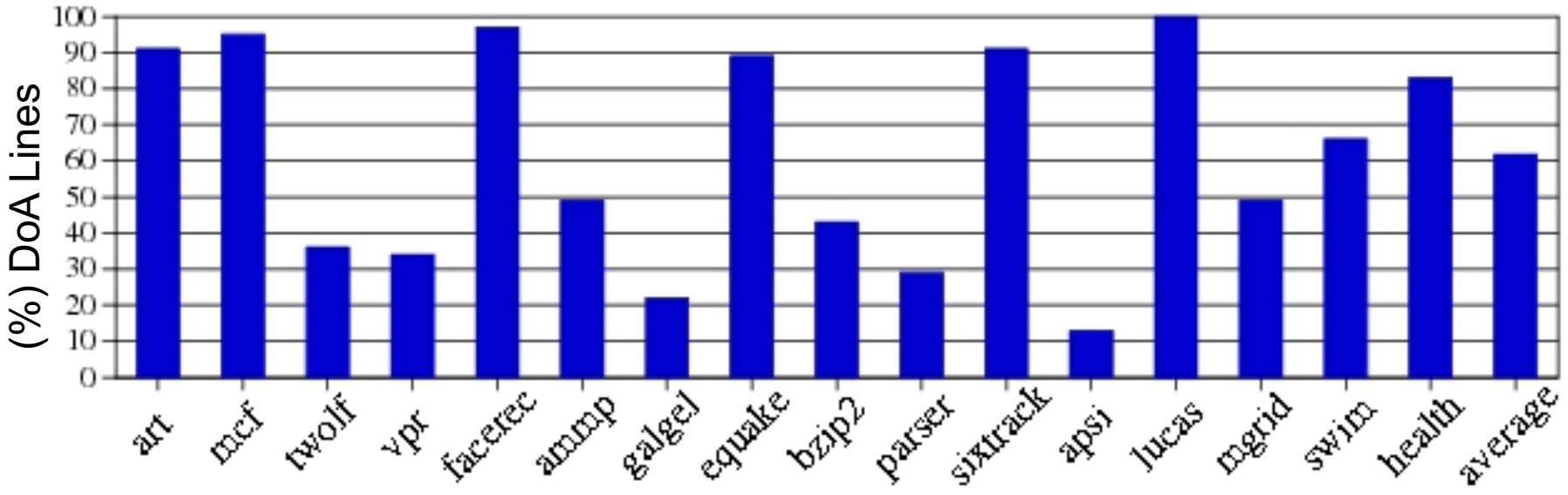
Revisiting Cache Placement (Insertion)

- Ideas:
  - Hardware predicts blocks that are not going to be used
    - Tyson et al., "A Modified Approach to Data Cache Management," MICRO 1995.
    - Lai et al., "Dead Block Prediction," ISCA 2001.
  - Software (programmer/compiler) marks instructions that touch data that is not going to be reused
    - How does software determine this?

- Streaming versus non-streaming accesses
  - If a program is streaming through data, reuse likely occurs only for a limited period of time
  - If such instructions are marked by the software, the hardware can store them temporarily in a smaller buffer (L0 cache) instead of the cache

Reuse at L2 Cache Level

DoA Blocks: Blocks unused between insertion and eviction

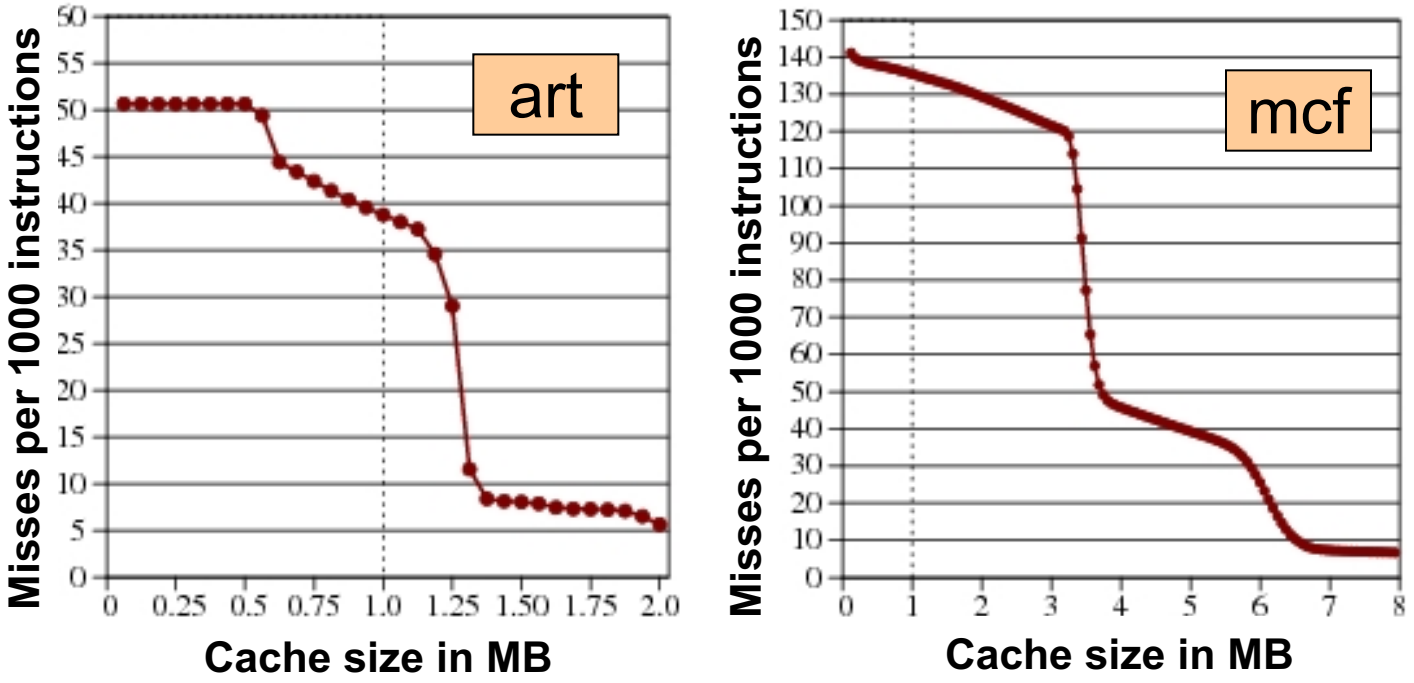


For the 1MB 16-way L2, 60% of lines are DoA

➔ Ineffective use of cache space

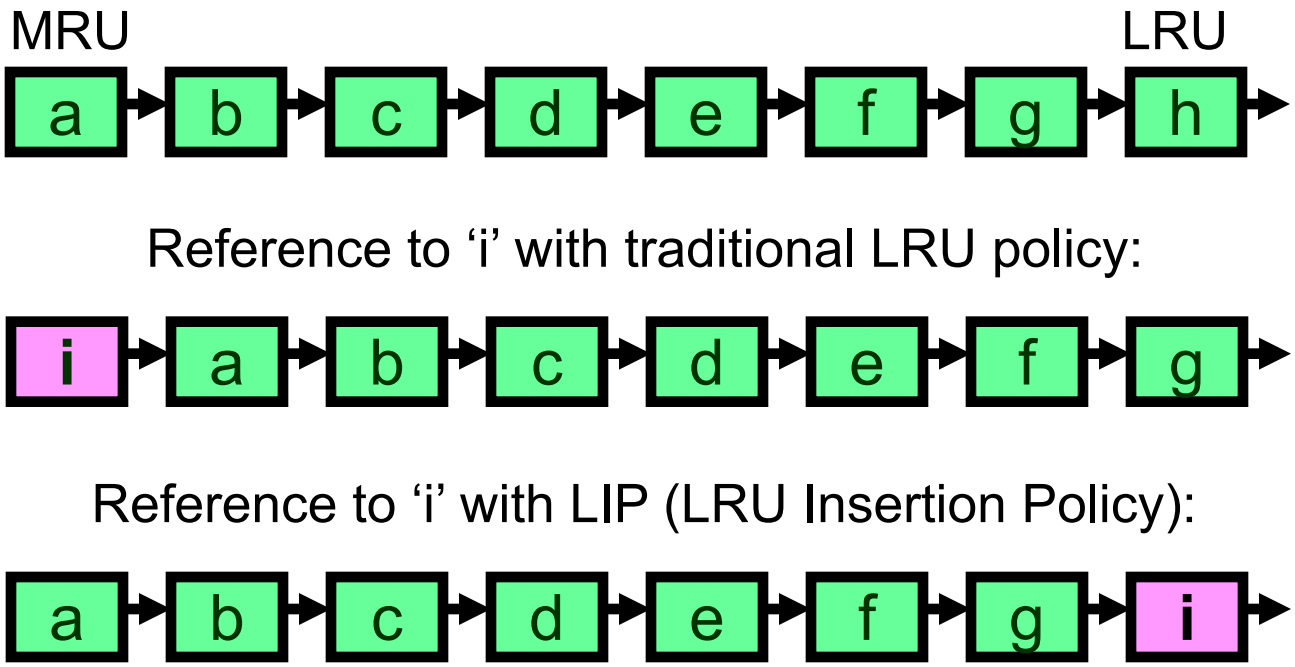
Why Dead on Arrival Blocks?

- Streaming data ➔ Never reused. L2 caches don't help.
- Working set of application greater than cache size



Solution: if working set > cache size, retain some working set

Cache Insertion Policies: MRU vs. LRU



Choose victim. Do NOT promote to MRU

Lines do not enter non-LRU positions unless reused

Other Insertion Policies: Bimodal Insertion

LIP does not age older lines

Infrequently insert lines in MRU position

Let ε = Bimodal throttle parameter

```
if  ( rand() < ε )
        Insert at MRU position;
else
        Insert at LRU position;
```

For small ε , BIP retains thrashing protection of LIP while responding to changes in working set

Analysis with Circular Reference Model

Reference stream has T blocks and repeats N times.
Cache has K blocks (K<T and N>>T)

Cache hit rates of two consecutive reference streams:

Policy	$(a_1\ a_2\ a_3\ \ldots\ a_T)^N$	$(b_1\ b_2\ b_3\ \ldots\ b_T)^N$
LRU	0	0
OPT	(K-1)/T	(K-1)/T
LIP	(K-1)/T	0
BIP (small ε)	$\approx$ (K-1)/T	$\approx$ (K-1)/T

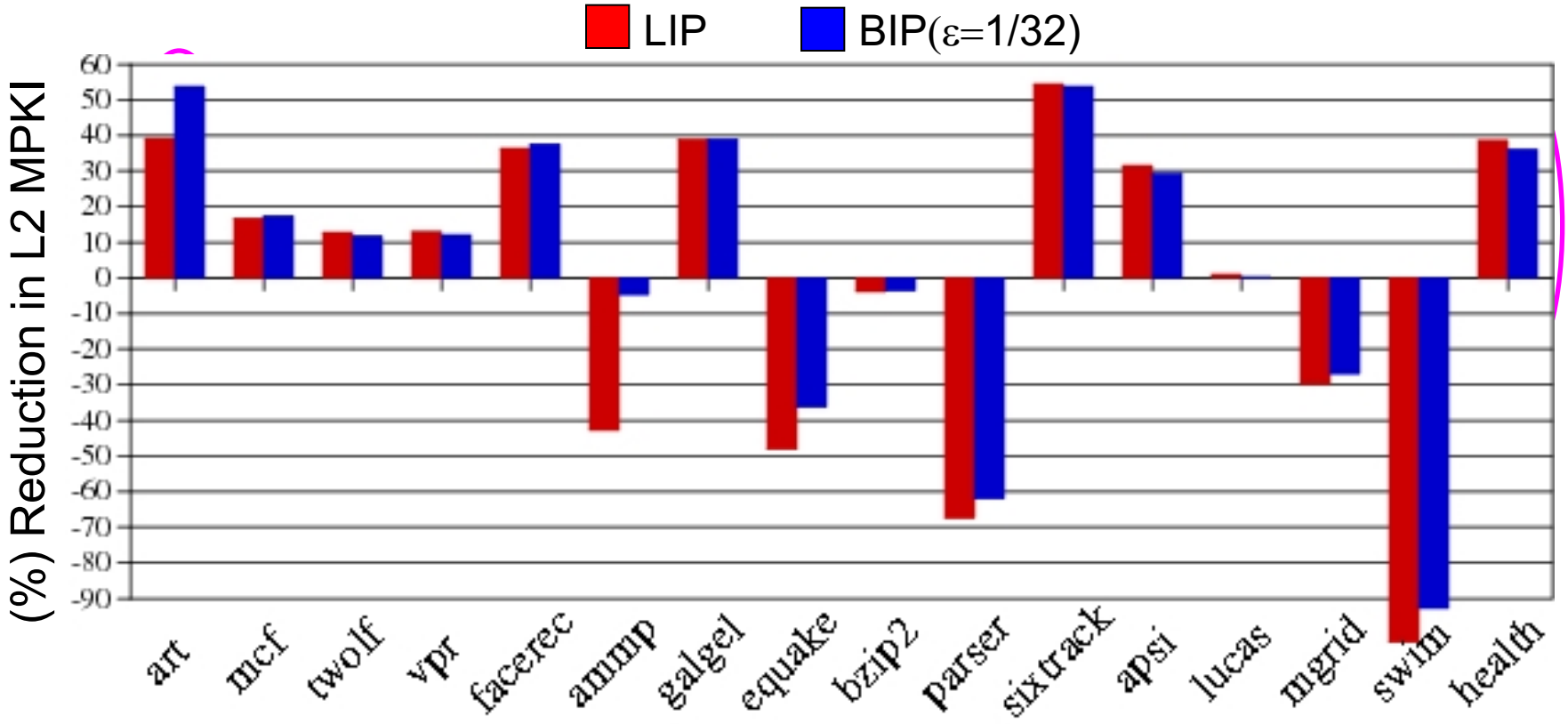
For small ε , BIP retains thrashing protection of LIP while adapting to changes in working set

Analysis with Circular Reference Model

Table 3: Hit Rate for LRU, OPT, LIP, and BIP

	$(a_1 \cdots a_T)^N$	$(b_1 \cdots b_T)^N$
LRU	0	0
OPT	$(K-1)/T$	$(K-1)/T$
LIP	$(K-1)/T$	0
BIP	$(K-1-\epsilon \cdot [T-K])/T$ $\approx (K-1)/T$	$\approx (K-1-\epsilon \cdot [T-K])/T$ $\approx (K-1)/T$

LIP and BIP Performance vs. LRU



Changes to insertion policy increases misses for LRU-friendly workloads

Dynamic Insertion Policy (DIP)

- Qureshi et al., "Adaptive Insertion Policies for High-Performance Caching," ISCA 2007.

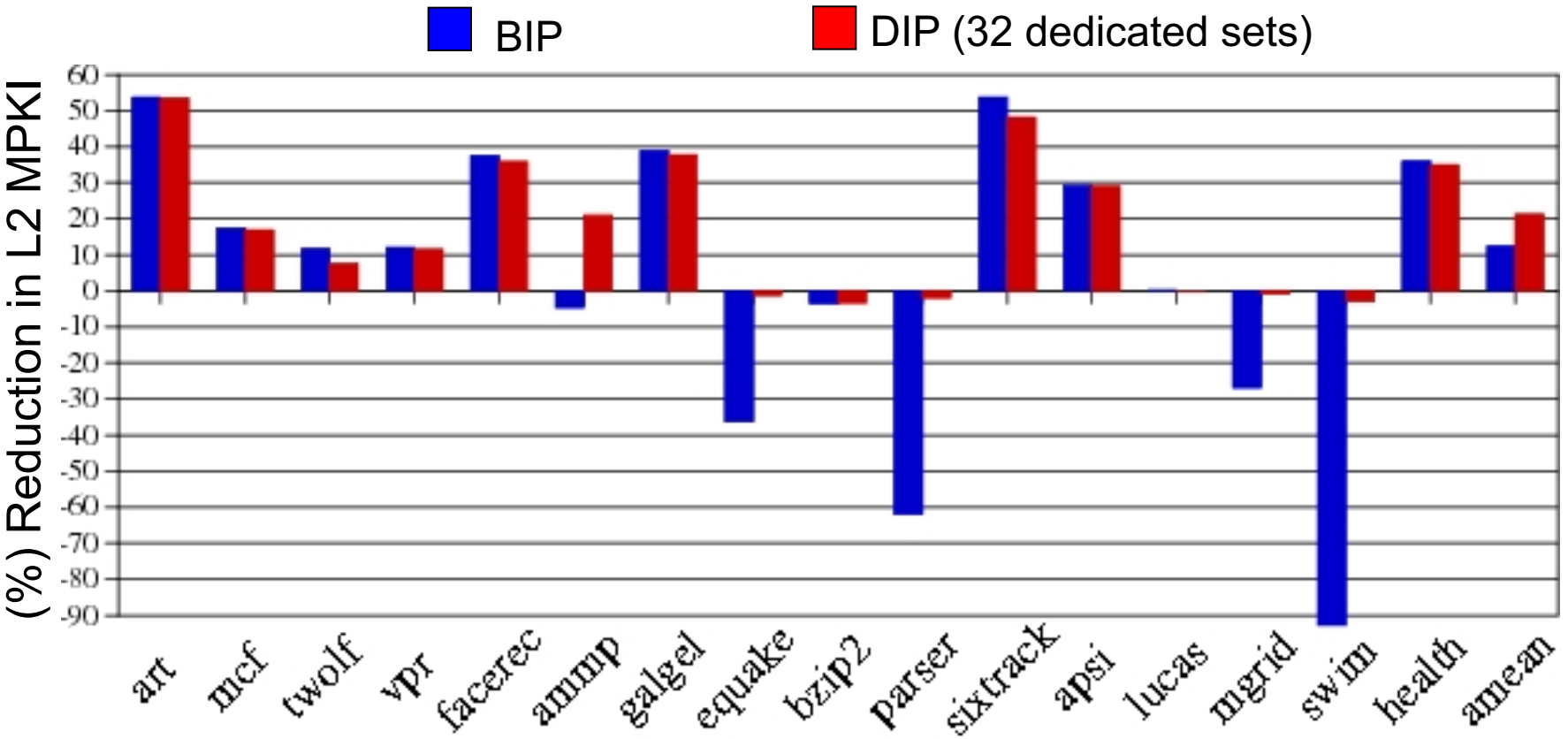
Two types of workloads: LRU-friendly or BIP-friendly

DIP can be implemented by:

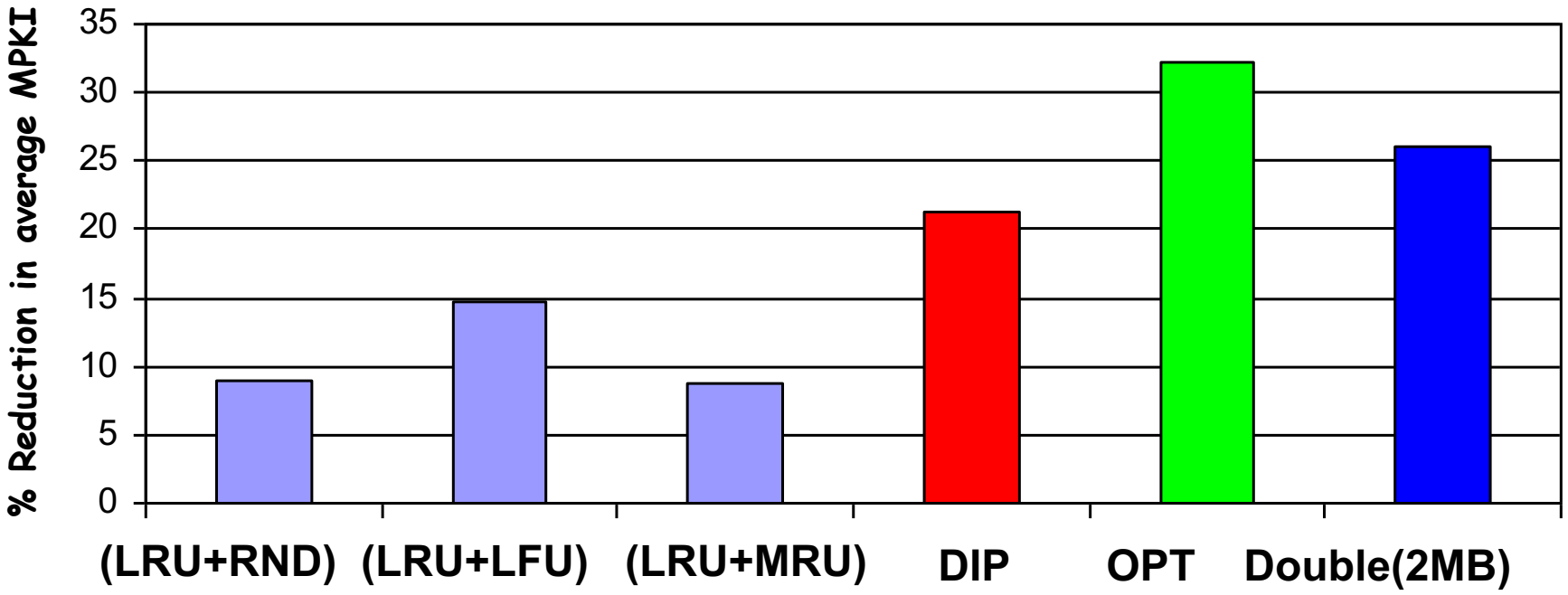
1. Monitor both policies (LRU and BIP)
2. Choose the best-performing policy
3. Apply the best policy to the cache

Need a cost-effective implementation ➔ Set Sampling

Dynamic Insertion Policy Miss Rate



DIP vs. Other Policies



- Qureshi et al., "Adaptive Insertion Policies for High-Performance Caching," ISCA 2007.

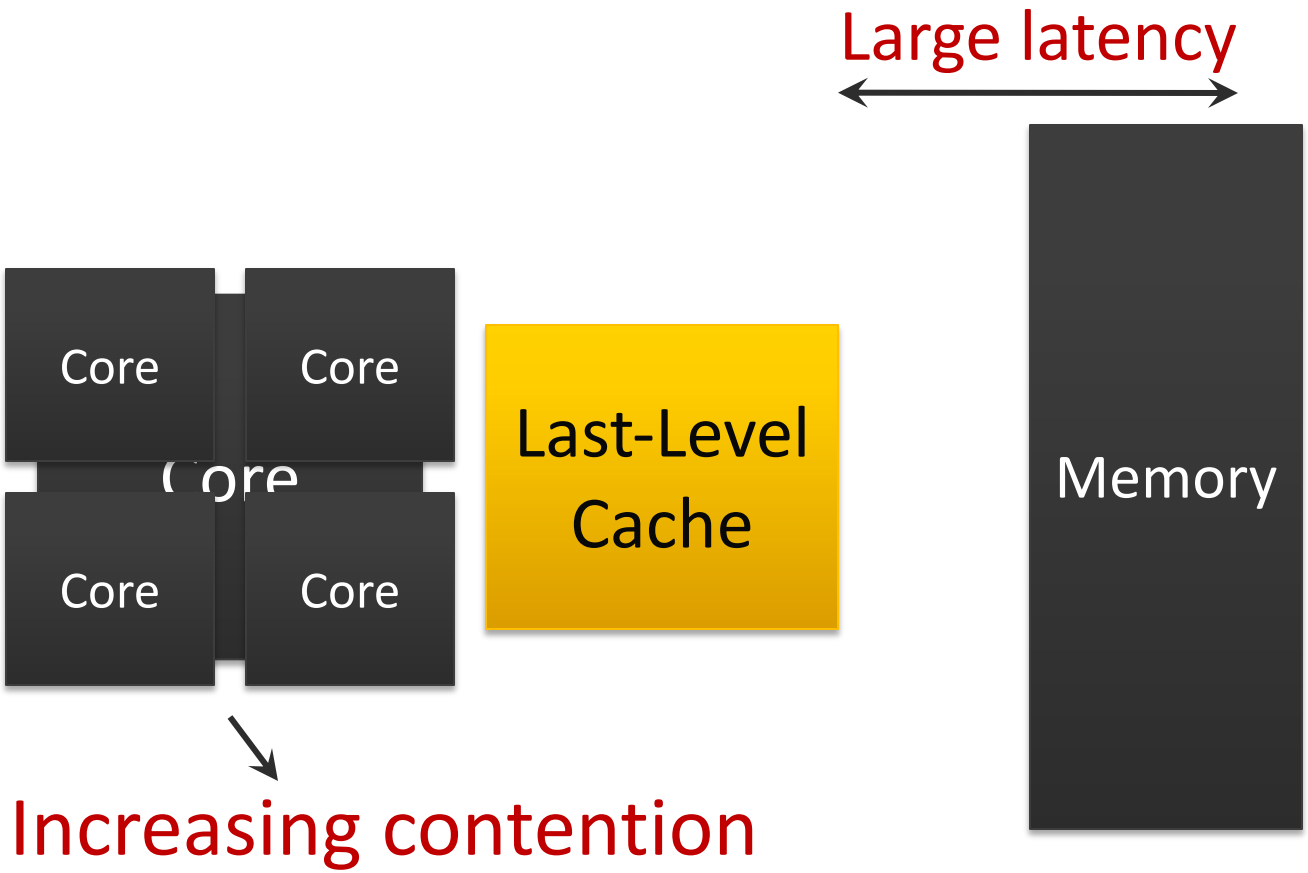
Efficient Cache Utilization: Examples

- Qureshi et al., "A Case for MLP-Aware Cache Replacement," ISCA 2005.
- Qureshi et al., "Adaptive Insertion Policies for High Performance Caching," ISCA 2007.
- 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.

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)

Cache Utilization is Important



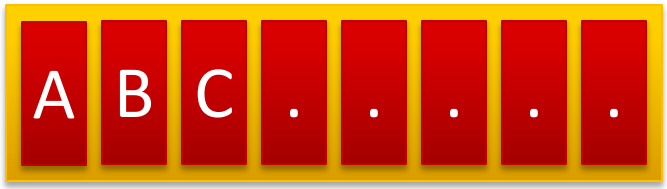
Effective cache utilization is important

Reuse Behavior of Cache Blocks

Different blocks have different reuse behavior

Access Sequence:

Ideal Cache



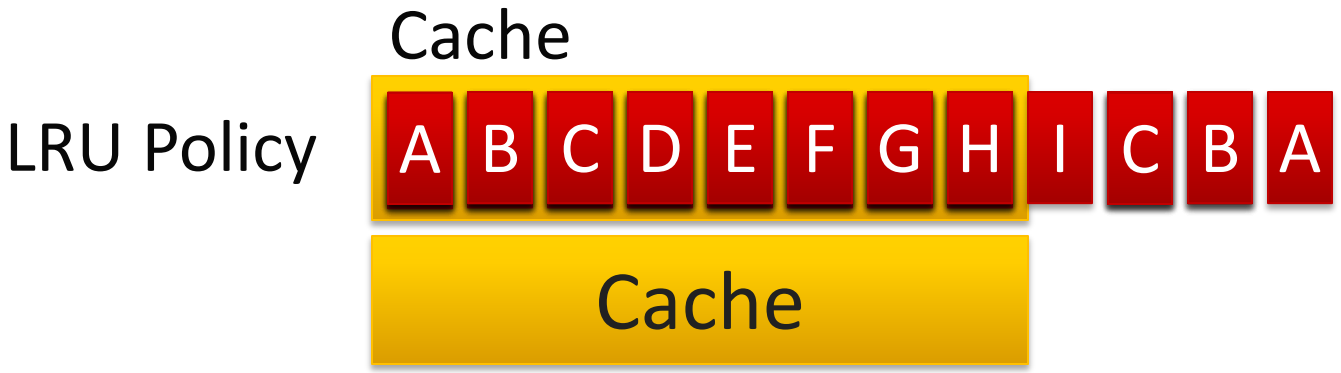
Cache Pollution

Problem: Low-reuse blocks evict high-reuse blocks

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

Cache Thrashing

Problem: High-reuse blocks evict each other



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

Qureshi+, "Adaptive insertion policies for high performance caching," ISCA 2007.

Handling Pollution and Thrashing

Need to address both pollution and thrashing concurrently

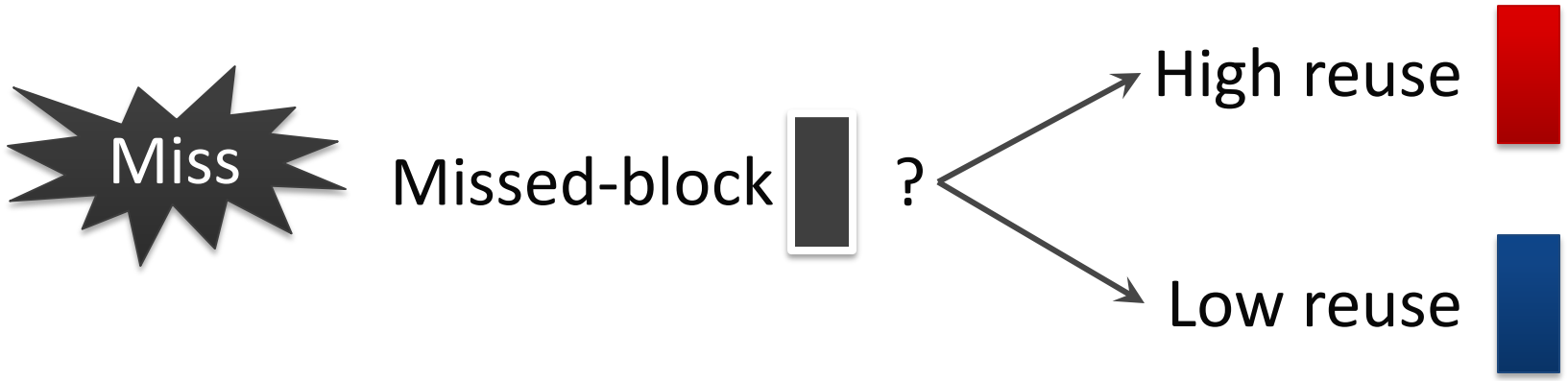
Cache Pollution

Need to distinguish high-reuse blocks from low-reuse blocks

Cache Thrashing

Need to control the number of blocks inserted with high priority into the cache

Reuse Prediction



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

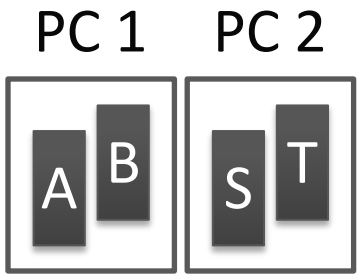
Impractical

1. High storage overhead
2. Look-up latency

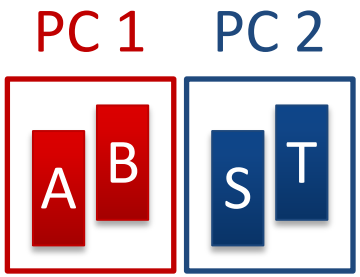
Approaches to Reuse Prediction

Use program counter or memory region information.

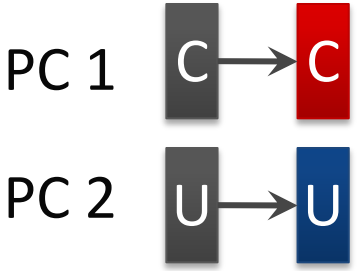
1. Group Blocks



2. Learn group behavior



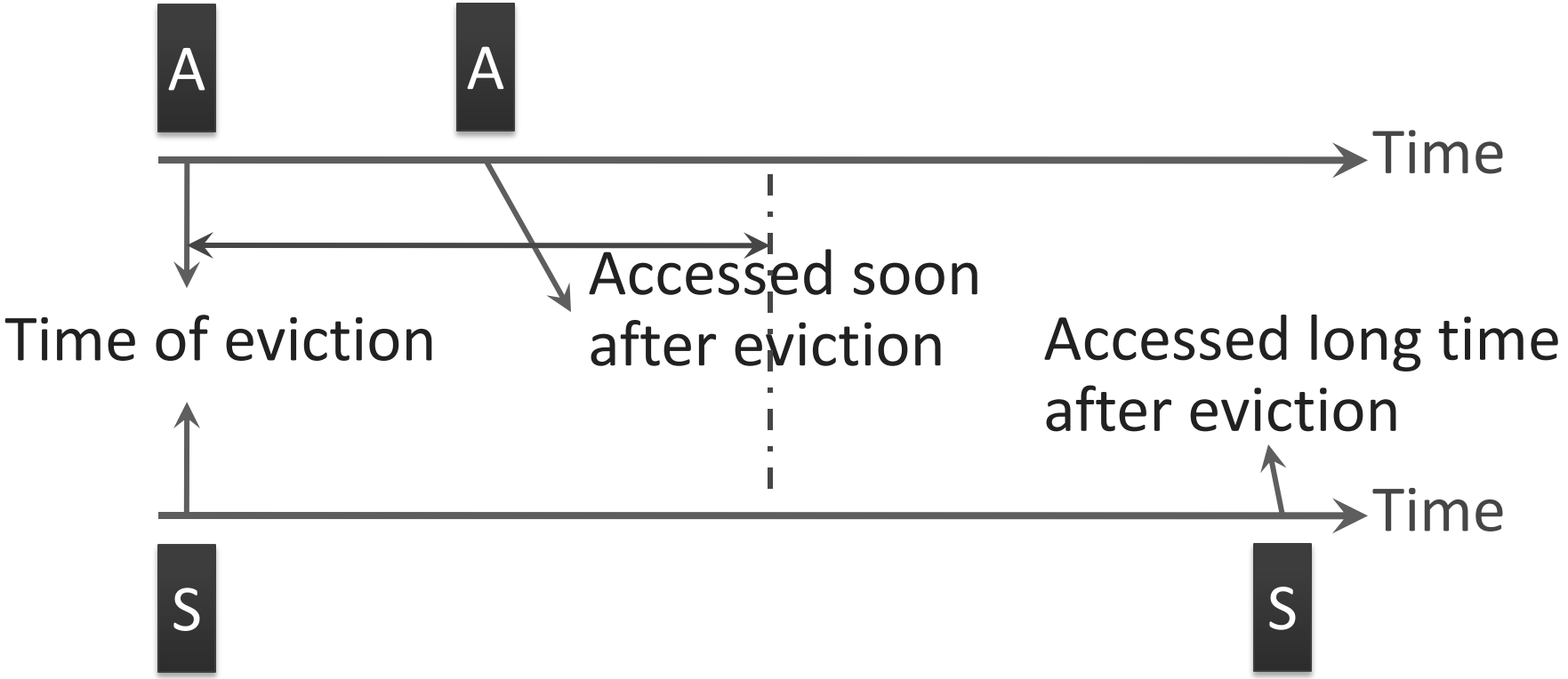
3. Predict reuse



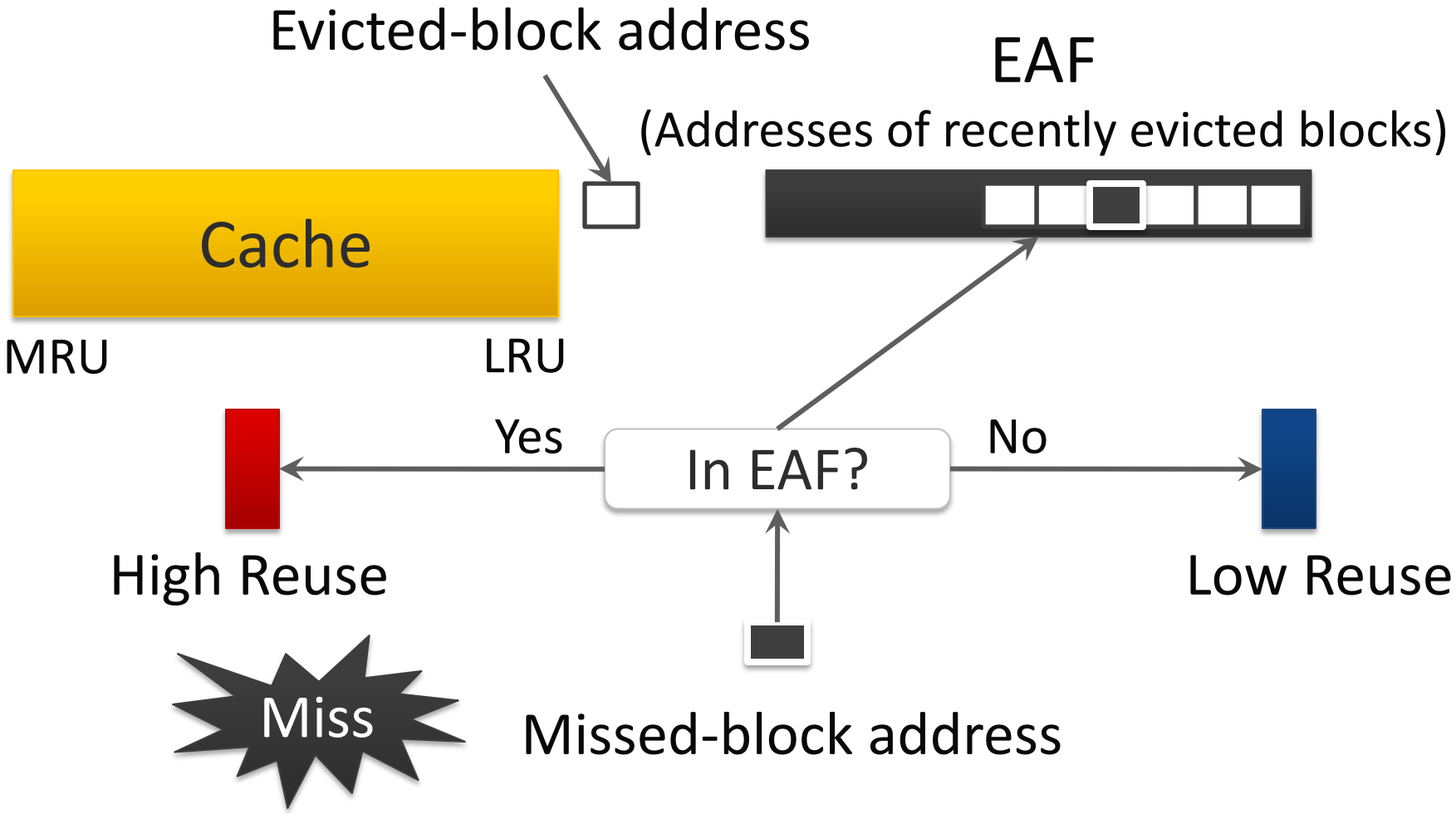
1. Same group $\not\rightarrow$ same reuse behavior
2. No control over number of high-reuse blocks

Per-block Reuse Prediction

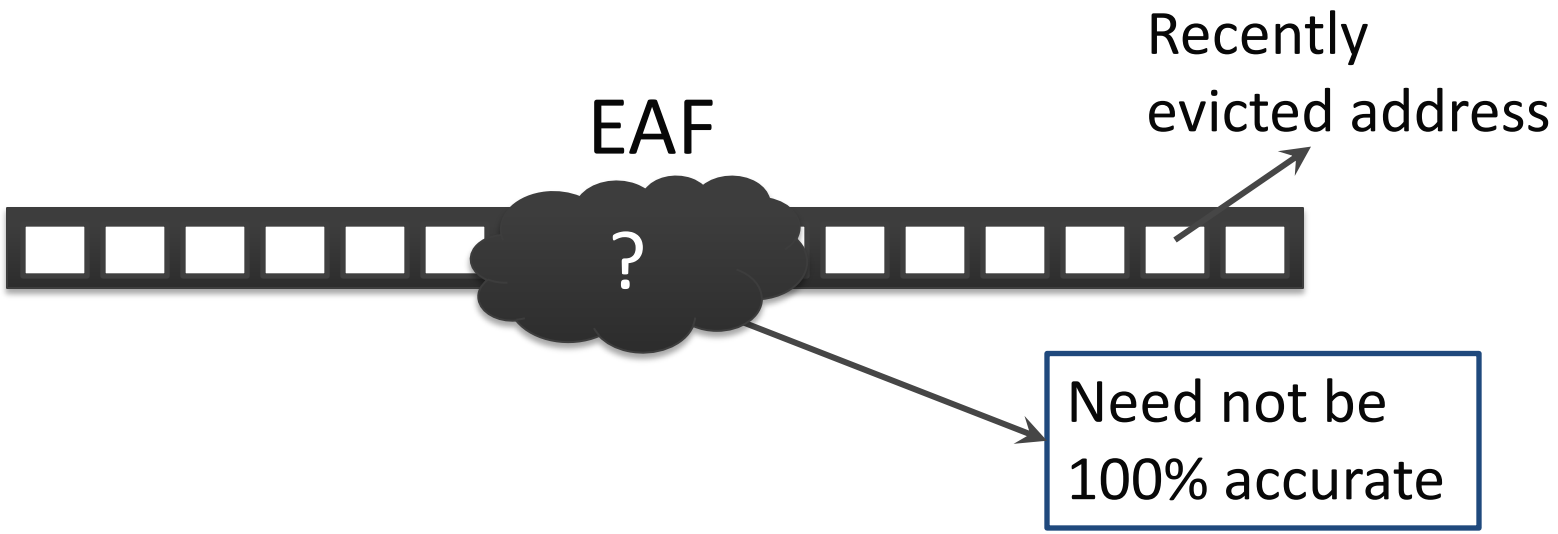
Use recency of eviction to predict reuse



Evicted-Address Filter (EAF)

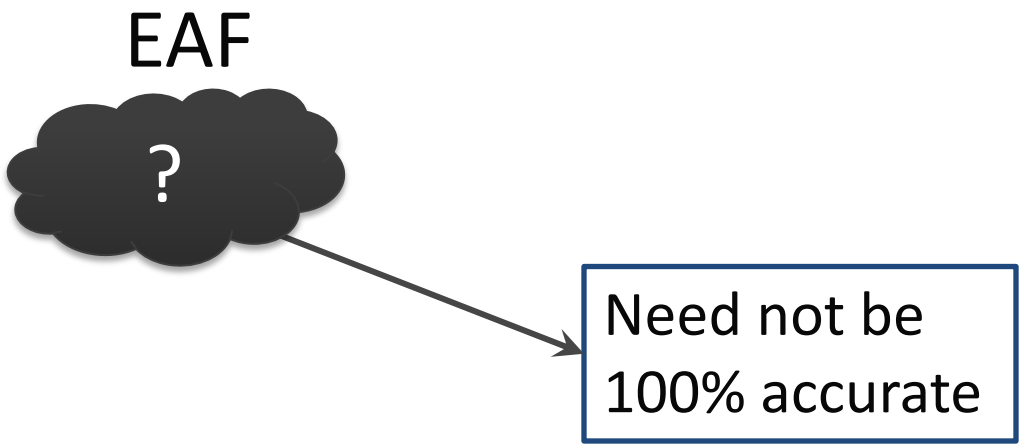


Naïve Implementation: Full Address Tags



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 Filters (From Lecture 1)

Space/Time Trade-offs in Hash Coding with Allowable Errors

BURTON H. BLOOM
Computer Usage Company, Newton Upper Falls, Mass.

In such applications, it is envisaged that overall performance could be improved by using a smaller core resident hash area in conjunction with the new methods and, when necessary, by using some secondary and perhaps time-consuming test to "catch" the small fraction of errors associated with the new methods. An example is discussed which illustrates possible areas of application for the new methods.

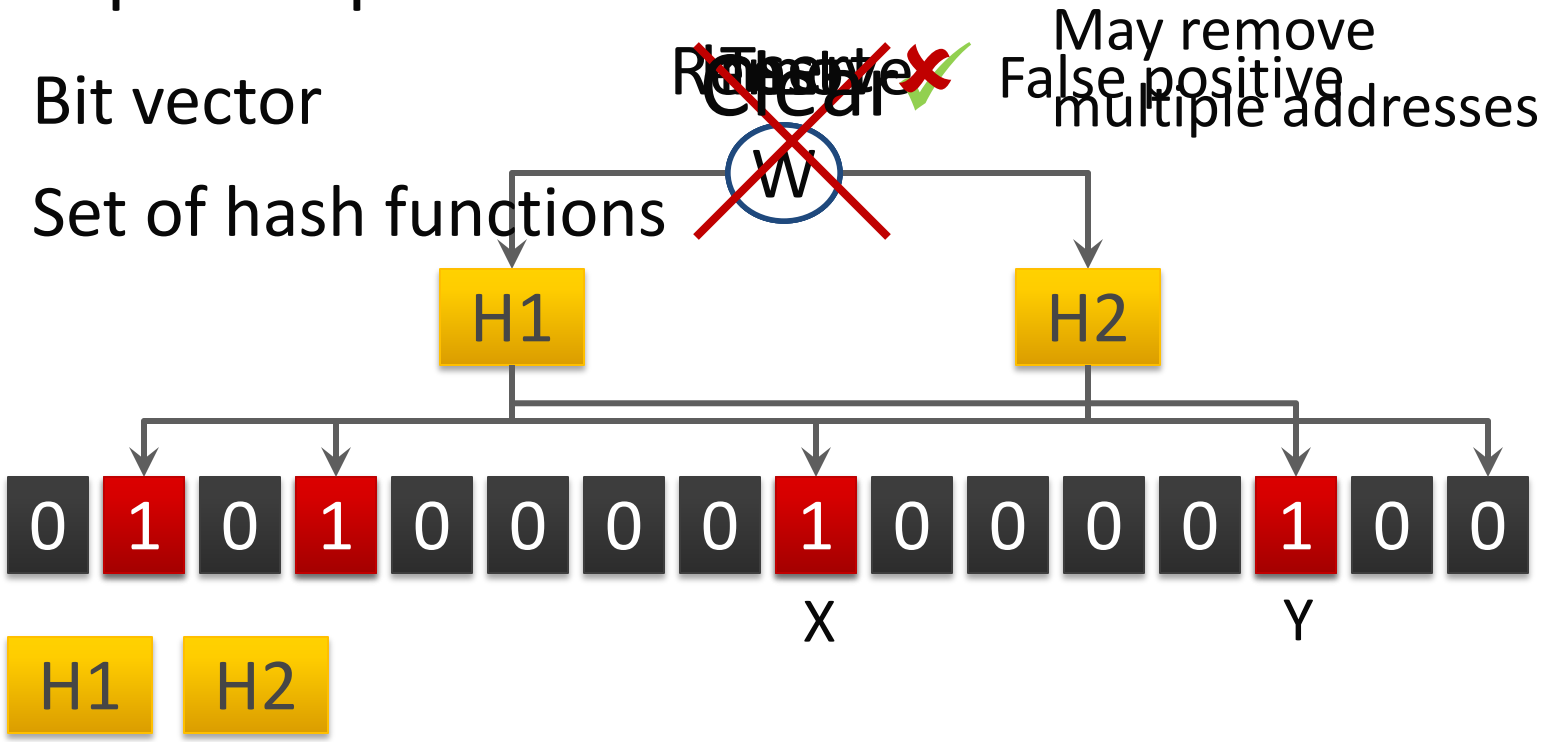
In this paper trade-offs among certain computational factors in hash coding are analyzed. The paradigm problem considered is that of testing a series of messages one-by-one for membership in a given set of messages. Two new hash-coding methods are examined and compared with a particular conventional hash-coding method. The computational factors considered are the size of the hash area (space), the time required to identify a message as a nonmember of the given set (reject time), and an allowable error frequency.

Bloom, "Space/Time Trade-offs in Hash Coding with Allowable Errors", CACM 1970.

Bloom Filter

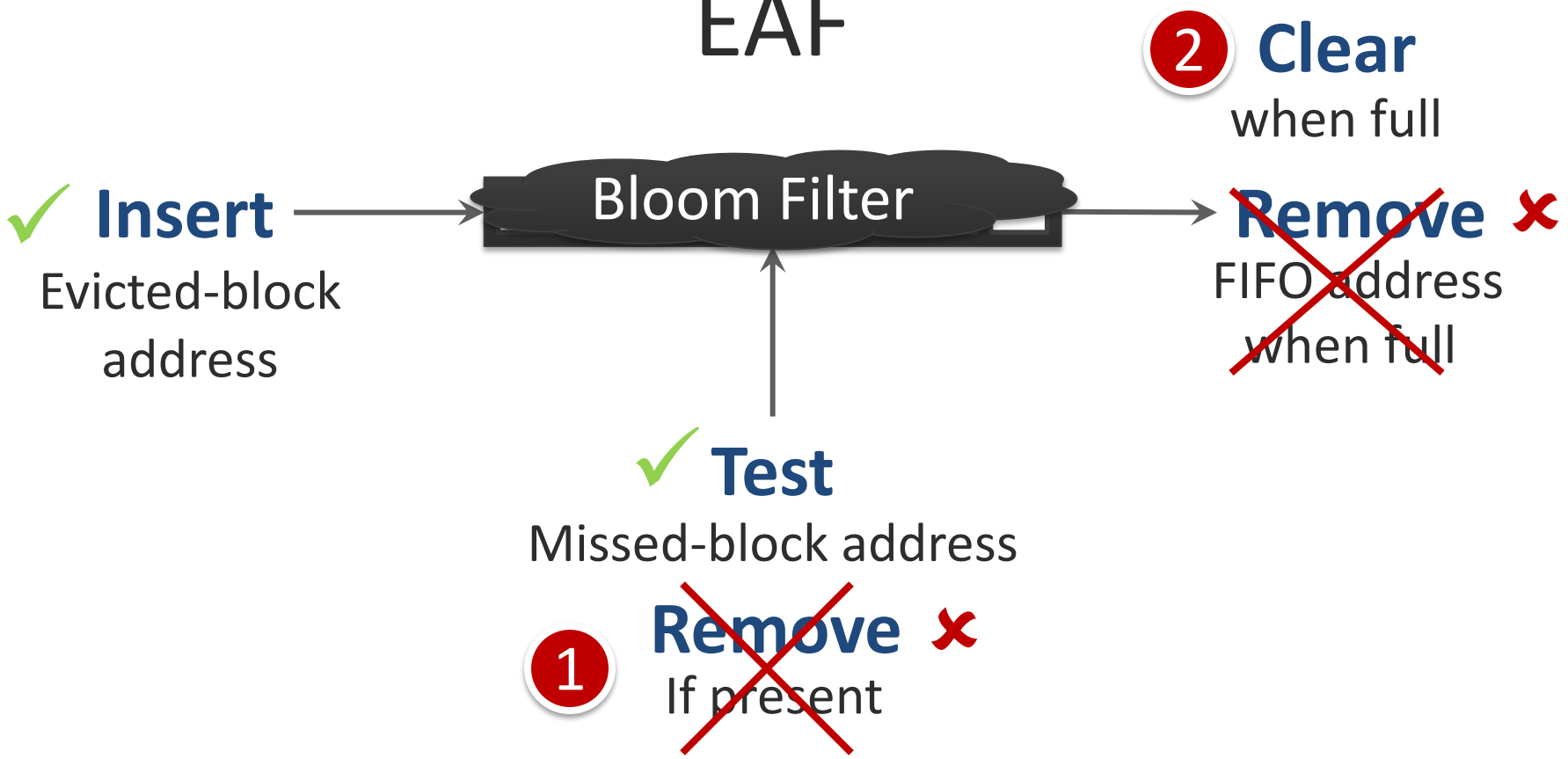
Compact representation of a set

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

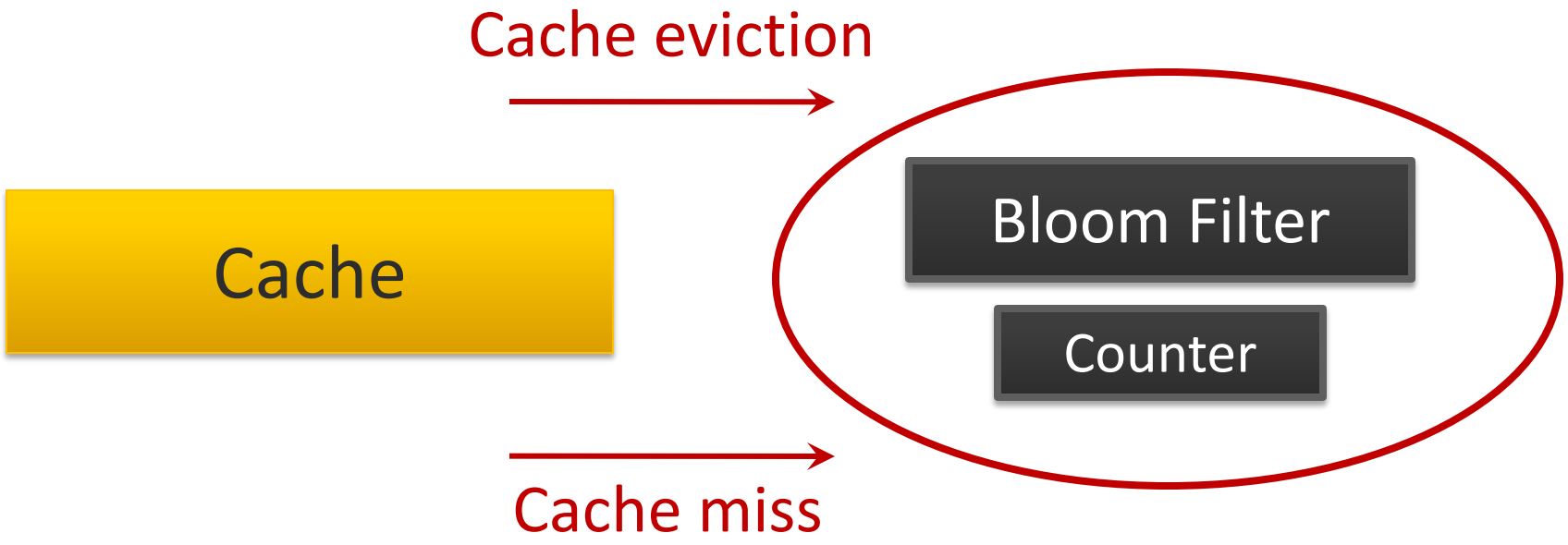
EAF-Cache: Final Design

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

3 **Counter reaches max**
Clear filter and counter

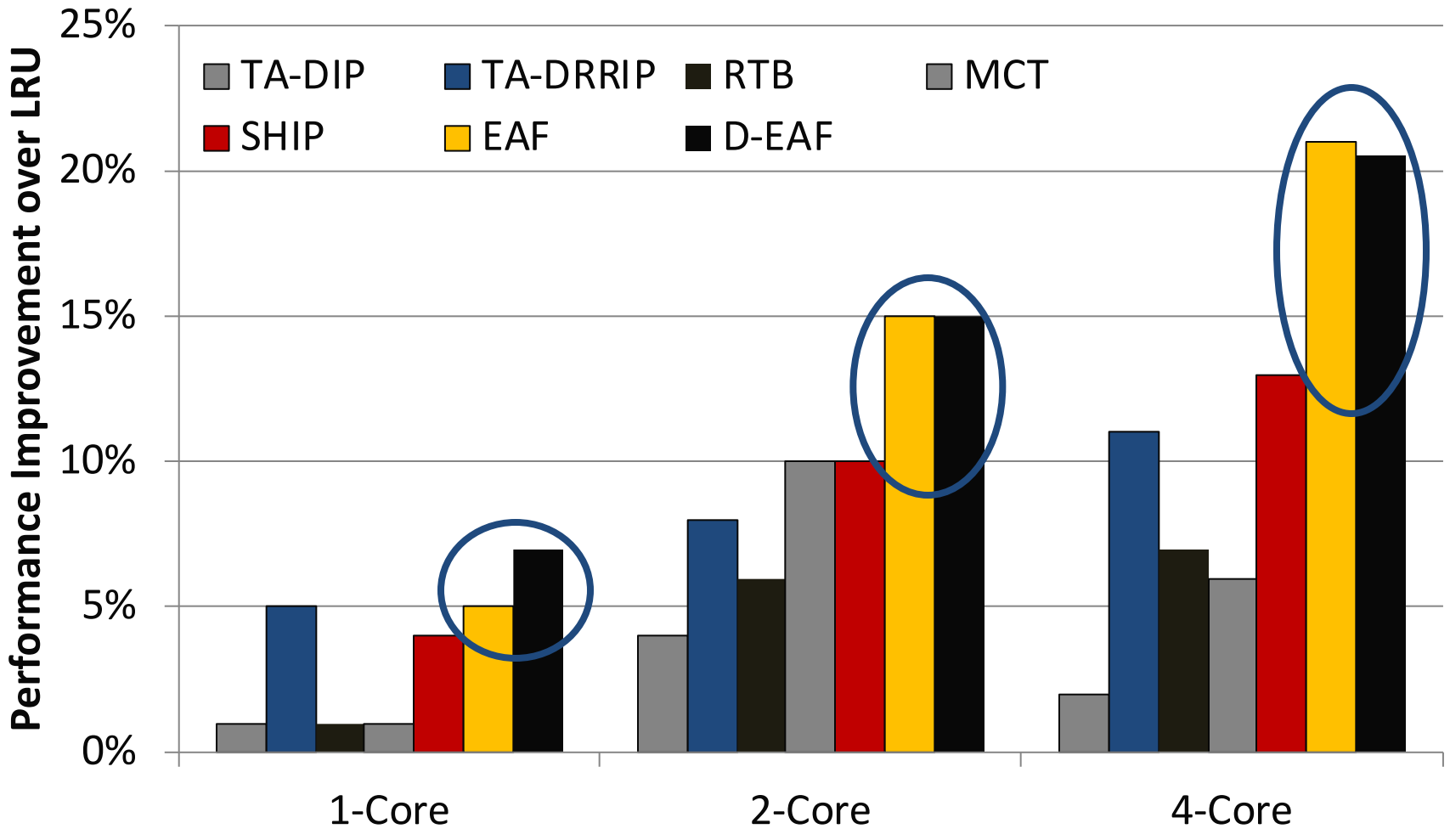
2 **Cache miss**
Test if address is present in filter
Yes, insert at MRU. No, insert with BIP

EAF: Advantages



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

EAF Performance – Summary



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 $\Longrightarrow$ Thrashing

Comparison with Prior Works

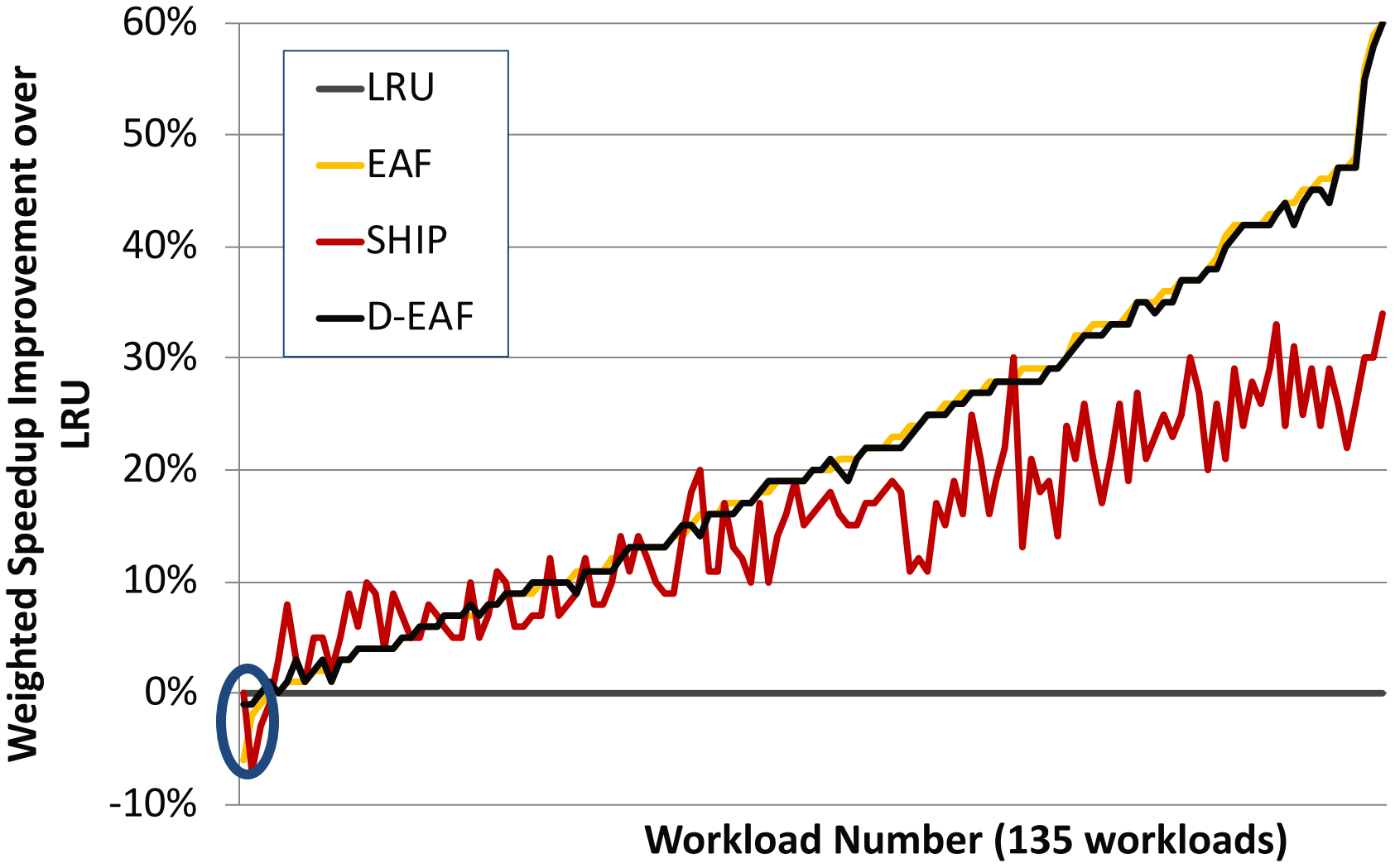
Addressing Cache Thrashing

TA-DIP – Qureshi+ ISCA'07, Jaleel+ PACT'08
TA-DRRIP – Jaleel+ ISCA'10

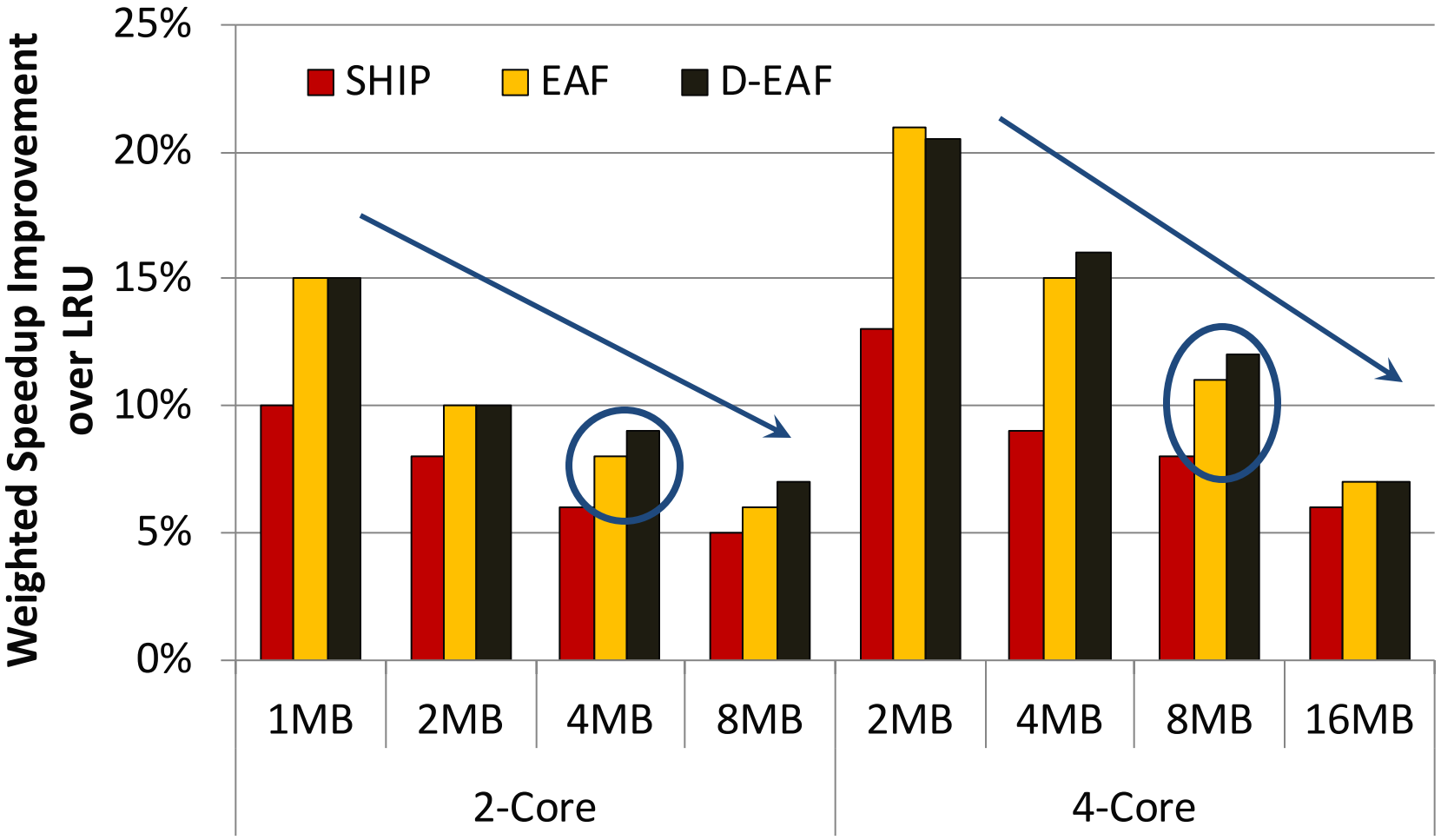
- Use set dueling to determine thrashing applications

- No mechanism to filter low-reuse blocks $\Longrightarrow$ Pollution

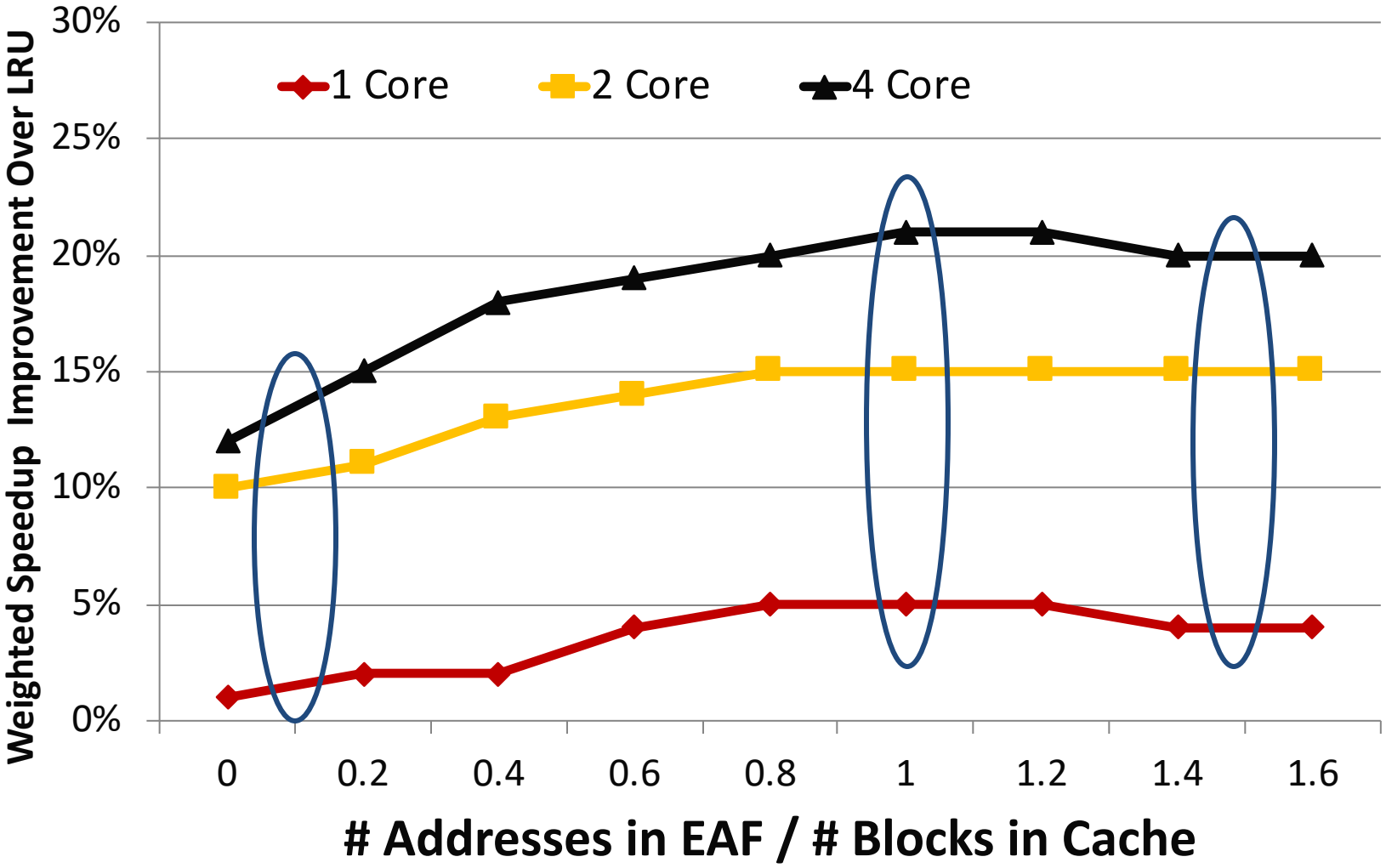
4-Core: Performance



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

More on Evicted Address Filter Cache

- 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 International Conference on Parallel Architectures and Compilation Techniques (**PACT**)*, Minneapolis, MN, September 2012. Slides (pptx) Source Code

The Evicted-Address Filter: A Unified Mechanism to Address Both Cache Pollution and Thrashing

Vivek Seshadri† vseshadr@cs.cmu.edu
Onur Mutlu† onur@cmu.edu
Michael A Kozuch⋆ michael.a.kozuch@intel.com
Todd C Mowry† tcm@cs.cmu.edu

†Carnegie Mellon University ⋆Intel Labs Pittsburgh

Computer Architecture

Lecture 18b:

Multi-Core Cache Management

Prof. Onur Mutlu

ETH Zürich

Fall 2018

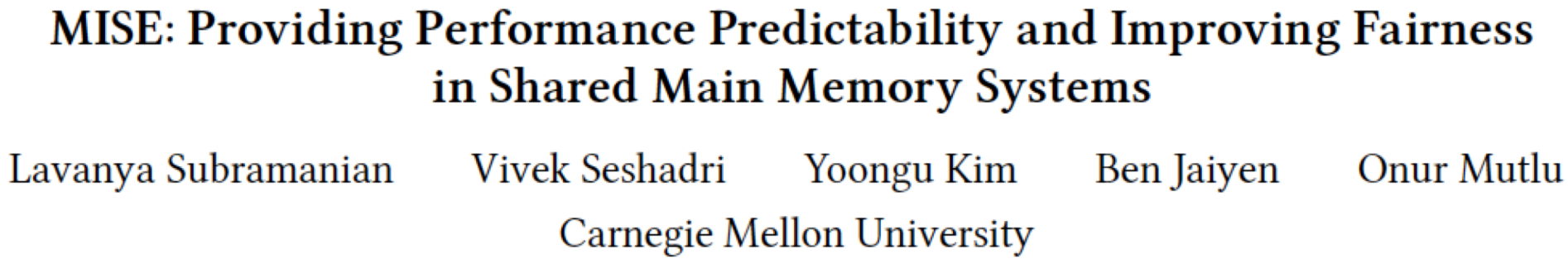
22 November 2018

We did not cover the following slides in lecture. These are for your preparation for the next lecture.

Predictable Performance Again: Strong Memory Service Guarantees

Remember MISE?

- Lavanya Subramanian, Vivek Seshadri, Yoongu Kim, Ben Jaiyen, and Onur Mutlu,
  "MISE: Providing Performance Predictability and Improving Fairness in Shared Main Memory Systems"
  *Proceedings of the 19th International Symposium on High-Performance Computer Architecture (**HPCA**)*, Shenzhen, China, February 2013. Slides (pptx)



MISE: Providing Performance Predictability and Improving Fairness in Shared Main Memory Systems

Lavanya Subramanian Vivek Seshadri Yoongu Kim Ben Jaiyen Onur Mutlu

Carnegie Mellon University

Extending Slowdown Estimation to Caches

- How do we extend the MISE model to include shared cache interference?
- Answer: Application Slowdown Model
- Lavanya Subramanian, Vivek Seshadri, Arnab Ghosh, Samira Khan, and Onur Mutlu,
  "The Application Slowdown Model: Quantifying and Controlling the Impact of Inter-Application Interference at Shared Caches and Main Memory"
  *Proceedings of the 48th International Symposium on Microarchitecture (**MICRO**)*, Waikiki, Hawaii, USA, December 2015.
  [Slides (pptx) (pdf)] [Lightning Session Slides (pptx) (pdf)] [Poster (pptx) (pdf)]
  [Source Code]

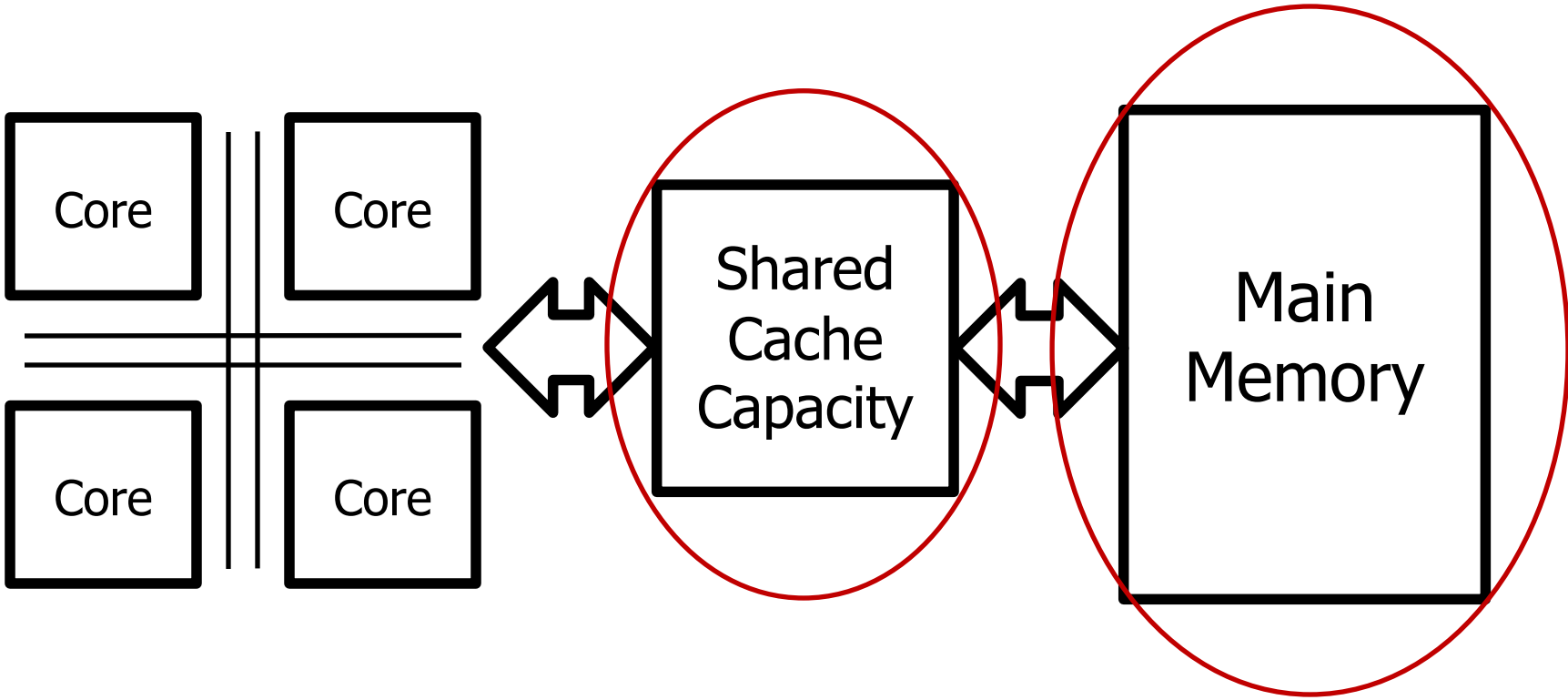
Application Slowdown Model

Quantifying and Controlling Impact of Interference at Shared Caches and Main Memory

Lavanya Subramanian, Vivek Seshadri,
Arnab Ghosh, Samira Khan, Onur Mutlu

SAFARI Carnegie Mellon intel UNIVERSITY OF VIRGINIA 1819

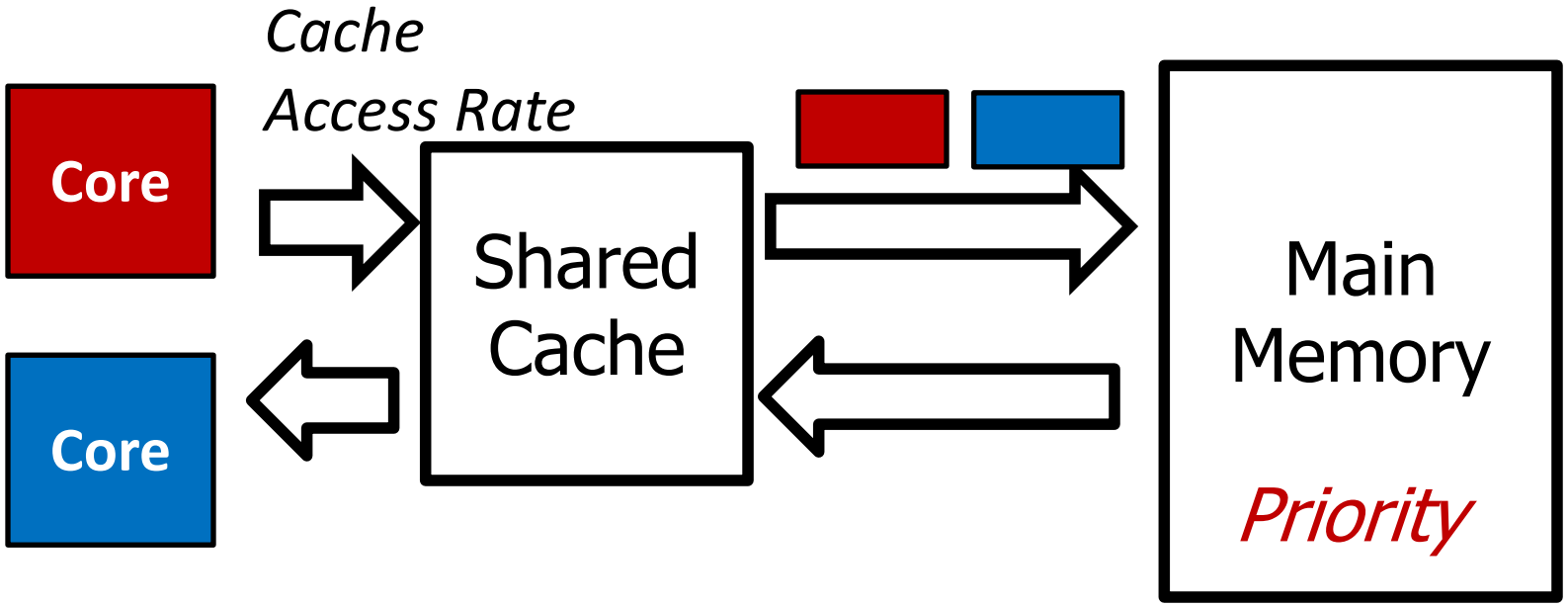
Shared Cache and Memory Contention



$$\text{Slowdown} = \frac{\text{Request Service Rate}_{\text{Alone}}}{\text{Request Service Rate}_{\text{Shared}}}$$

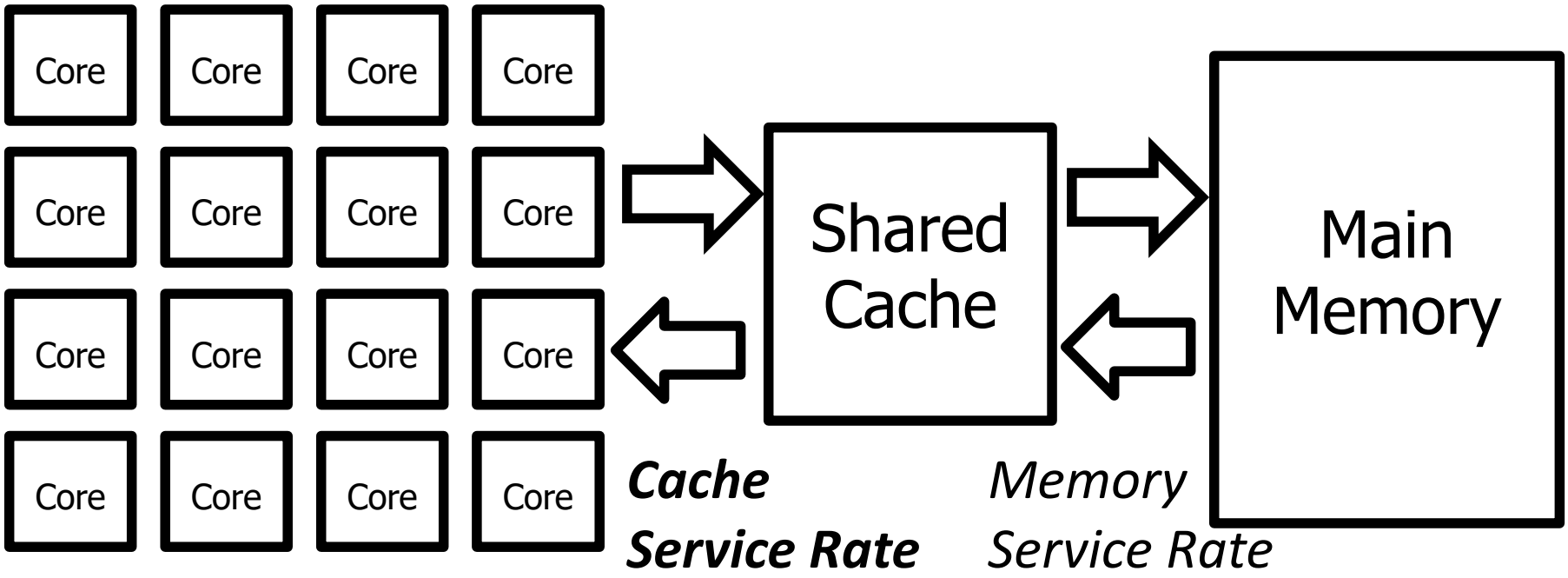
MISE [HPCA'13]

Cache Capacity Contention

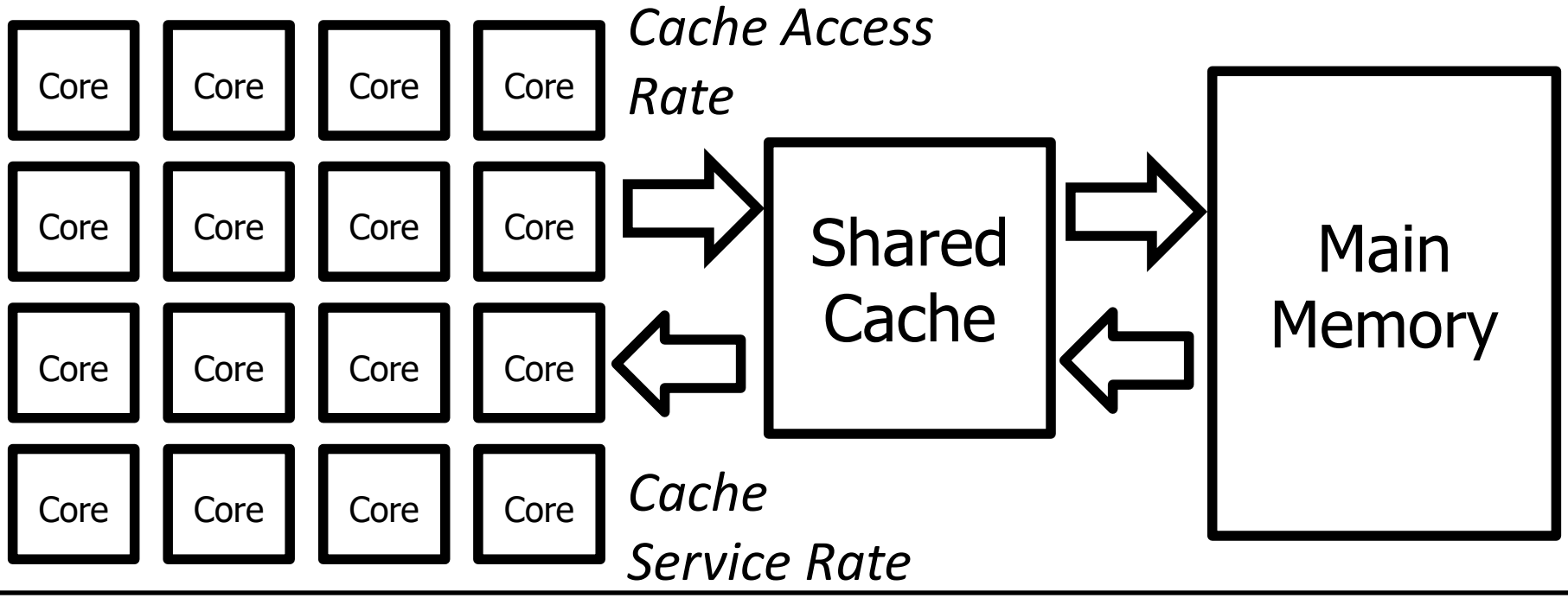


Applications evict each others' blocks from the shared cache

Estimating Cache and Memory Slowdowns

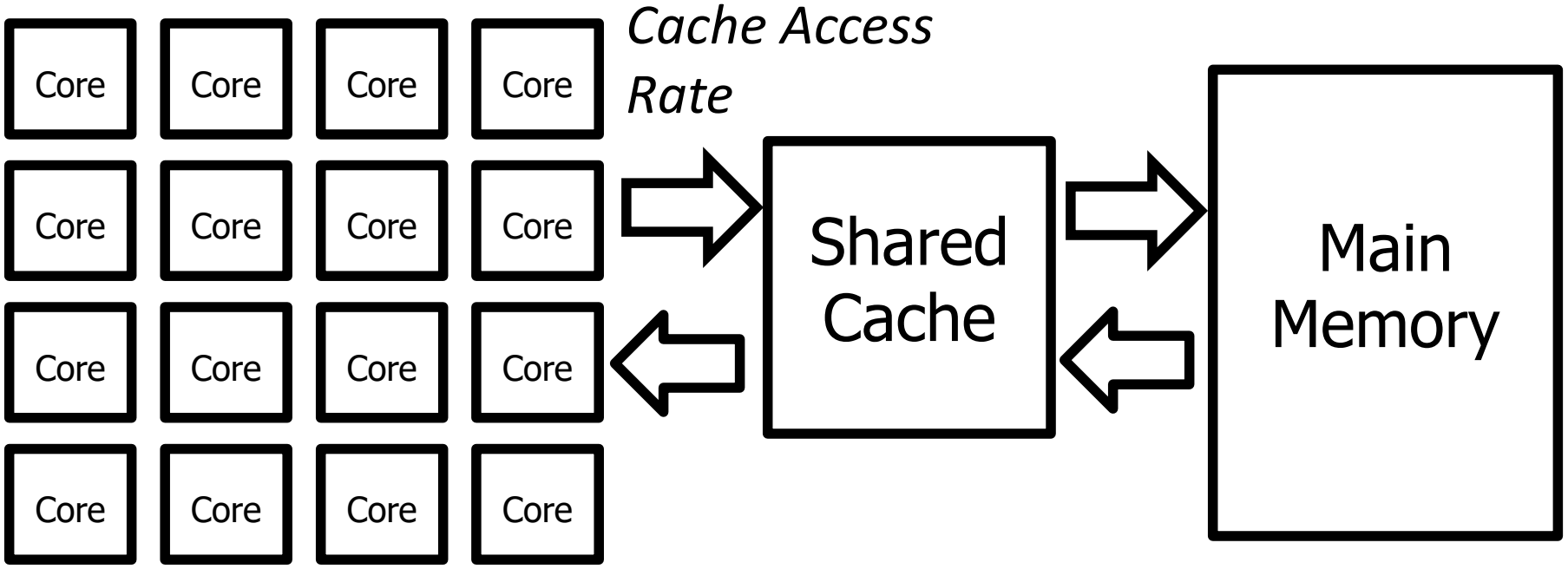


Service Rates vs. Access Rates



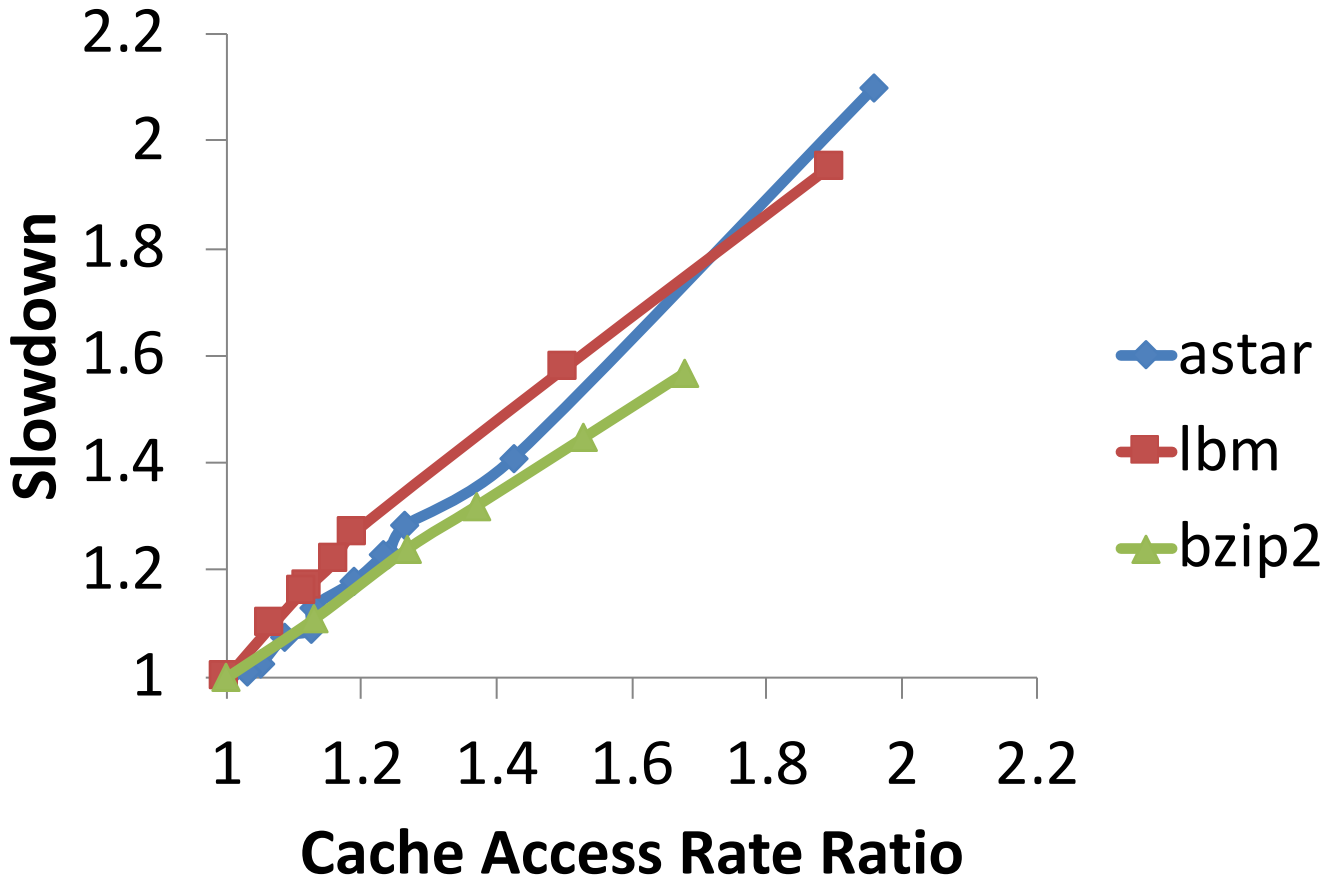
Request service and access rates are tightly coupled

The Application Slowdown Model



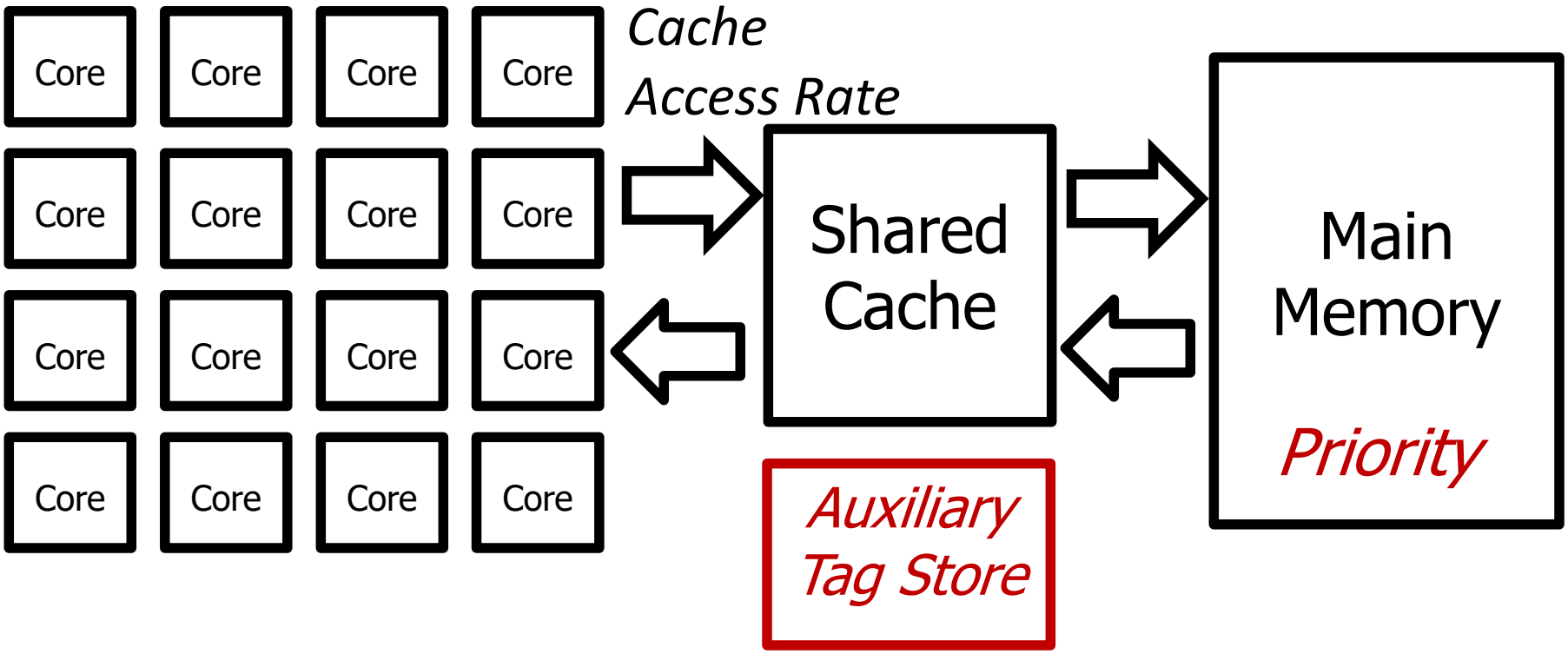
$$\text{Slowdown} = \frac{\text{Cache Access Rate}_{\text{Alone}}}{\text{Cache Access Rate}_{\text{Shared}}}$$

Real System Studies: Cache Access Rate vs. Slowdown

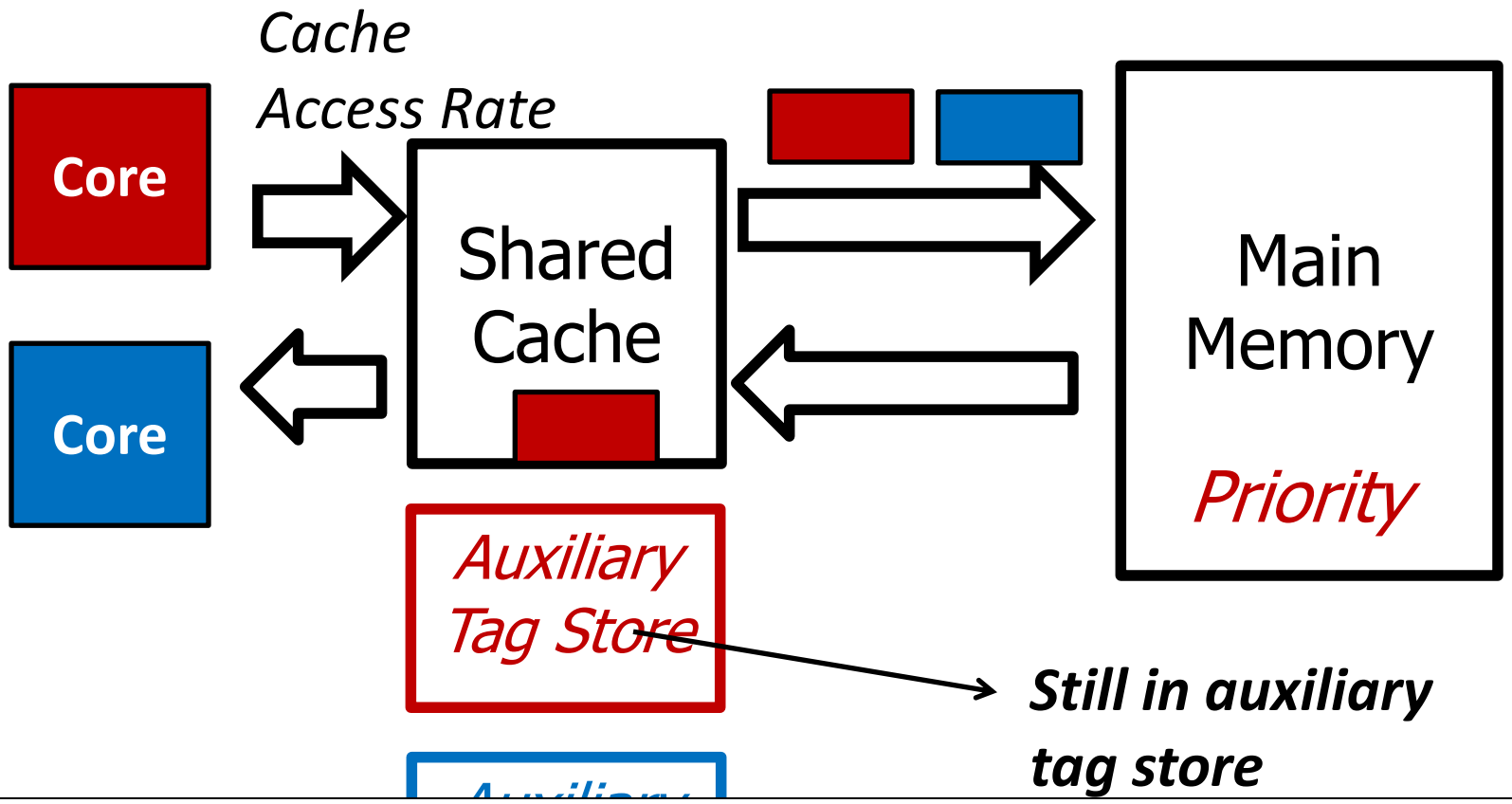


Challenge

How to estimate alone cache access rate?



Auxiliary Tag Store



Auxiliary tag store tracks such ***contention misses***

Accounting for Contention Misses

- Revisiting alone memory request service rate

$$\text{Alone Request Service Rate of an Application} = \frac{\text{\# Requests During High Priority Epochs}}{\text{\# High Priority Cycles}}$$

Cycles serving contention misses should not count as high priority cycles

Alone Cache Access Rate Estimation

$$\text{Cache Access Rate}_{\text{Alone}} \text{ of an Application} = \frac{\#\text{ Requests During High Priority Epochs}}{\#\text{ High Priority Cycles} - \#\text{Cache Contention Cycles}}$$

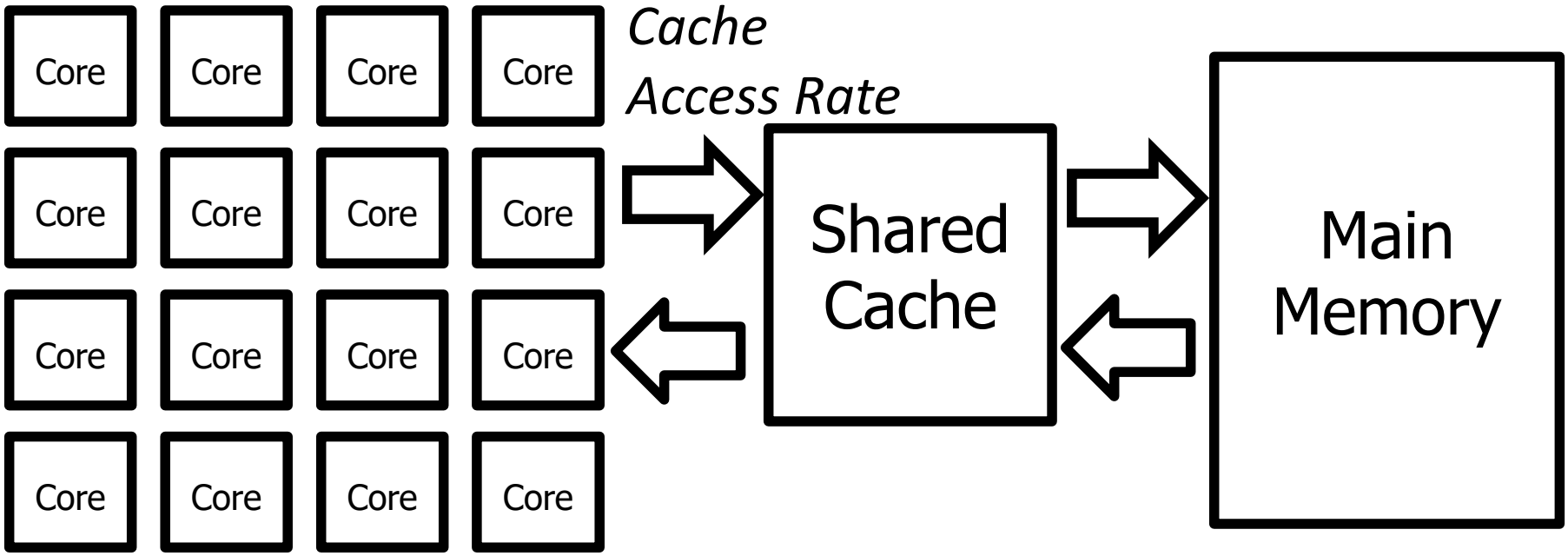
Cache Contention Cycles: Cycles spent serving contention misses

$$\text{Cache Contention Cycles} = \#\text{ Contention Misses} \times \text{Average Memory Service Time}$$

From auxiliary tag store when given high priority

Measured when given high priority

Application Slowdown Model (ASM)



$$\text{Slowdown} = \frac{\text{Cache Access Rate}_{\text{Alone}}}{\text{Cache Access Rate}_{\text{Shared}}}$$

Previous Work on Slowdown Estimation

- Previous work on slowdown estimation
  - **STFM** (Stall Time Fair Memory) Scheduling [Mutlu et al., MICRO '07]
  - **FST** (Fairness via Source Throttling) [Ebrahimi et al., ASPLOS '10]
  - **Per-thread Cycle Accounting** [Du Bois et al., HiPEAC '13]

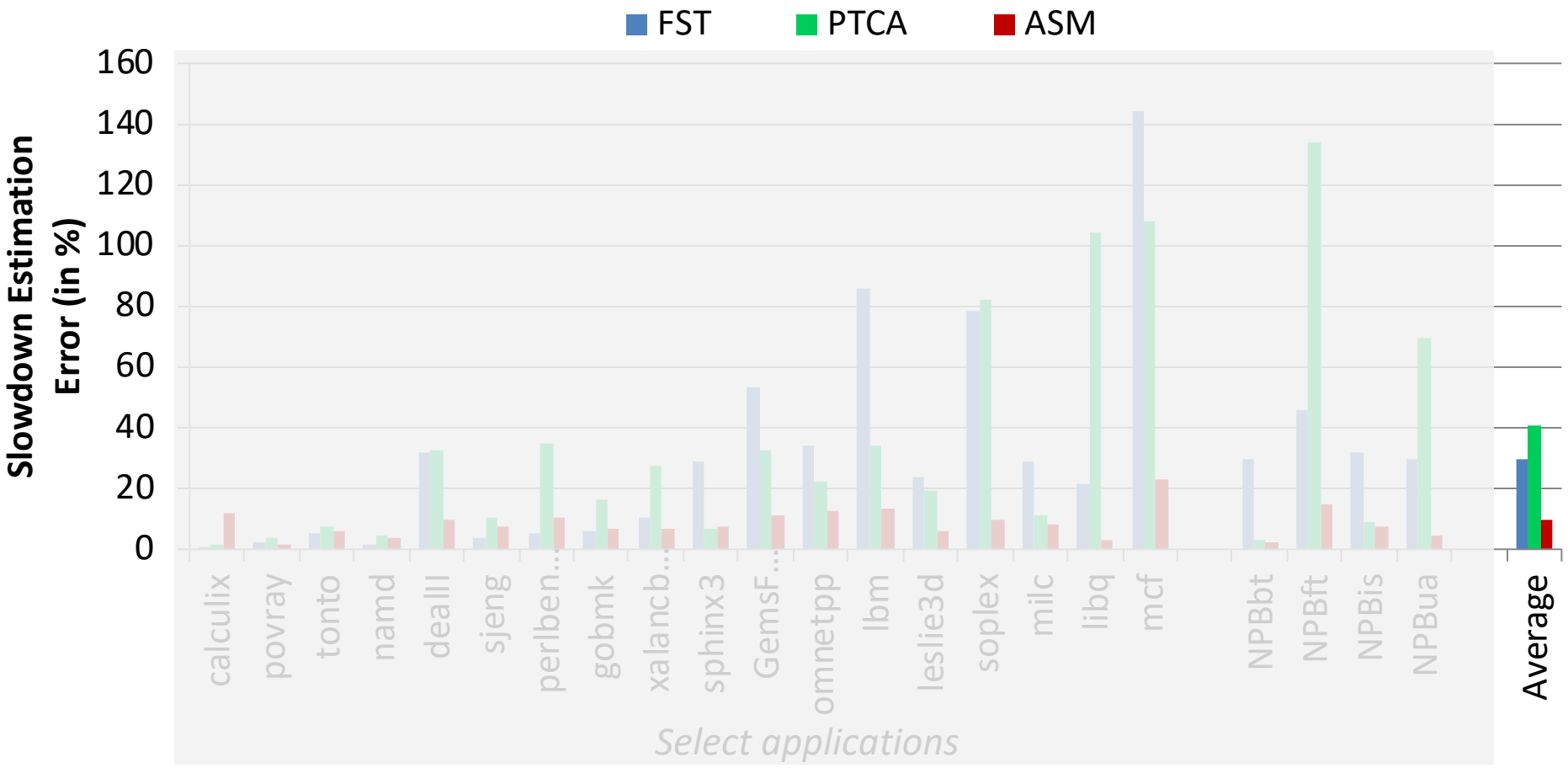
- Basic Idea:

$$\text{Slowdown} = \frac{\text{Execution Time}_{\text{Alone}}}{\text{Execution Time}_{\text{Shared}}}$$

Count interference experienced by each request → Difficult

ASM's estimates are much more coarse grained → Easier

Model Accuracy Results

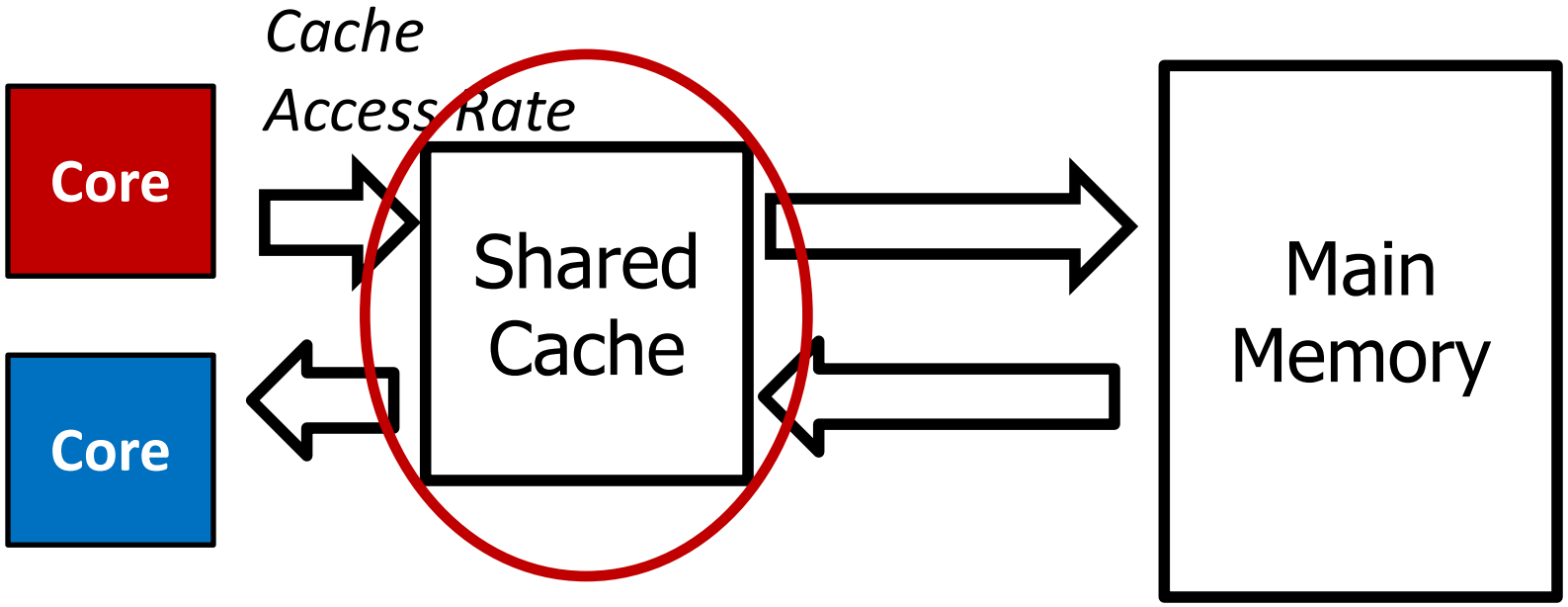


Average error of ASM's slowdown estimates: 10%

Leveraging ASM's Slowdown Estimates

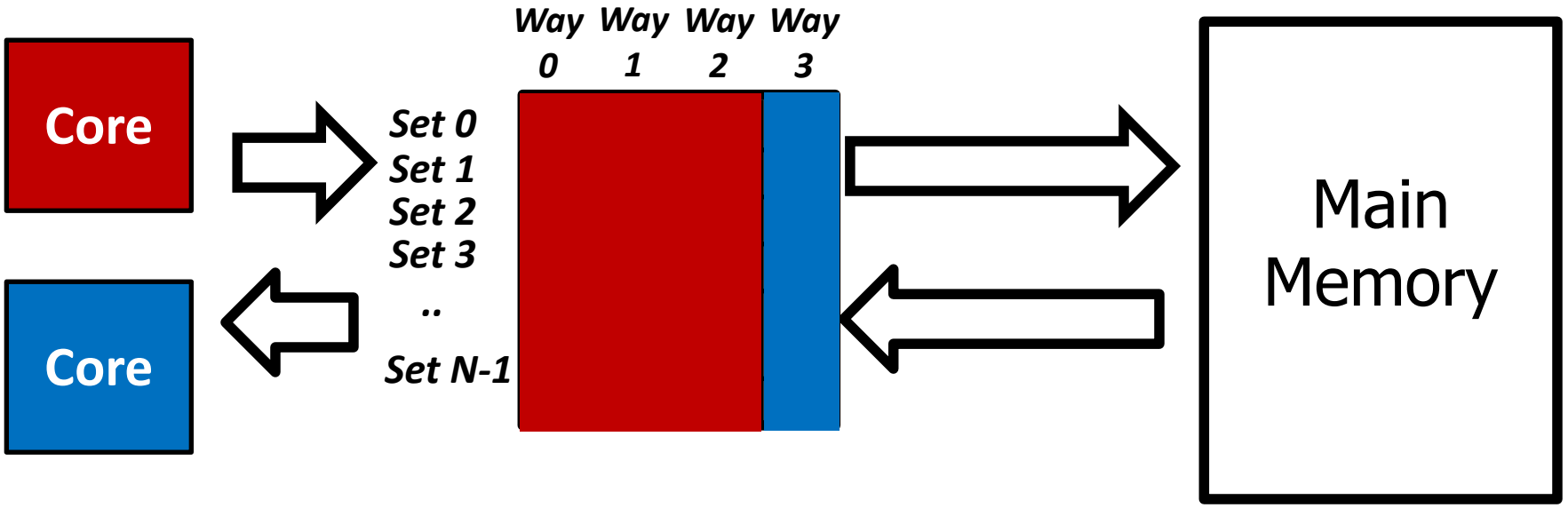
- *Slowdown-aware resource allocation for high performance and fairness*
- *Slowdown-aware resource allocation to bound application slowdowns*
- *VM migration and admission control schemes* ***[VEE '15]***
- *Fair billing schemes in a commodity cloud*

Cache Capacity Partitioning



Goal: Partition the shared cache among applications to mitigate contention

Cache Capacity Partitioning



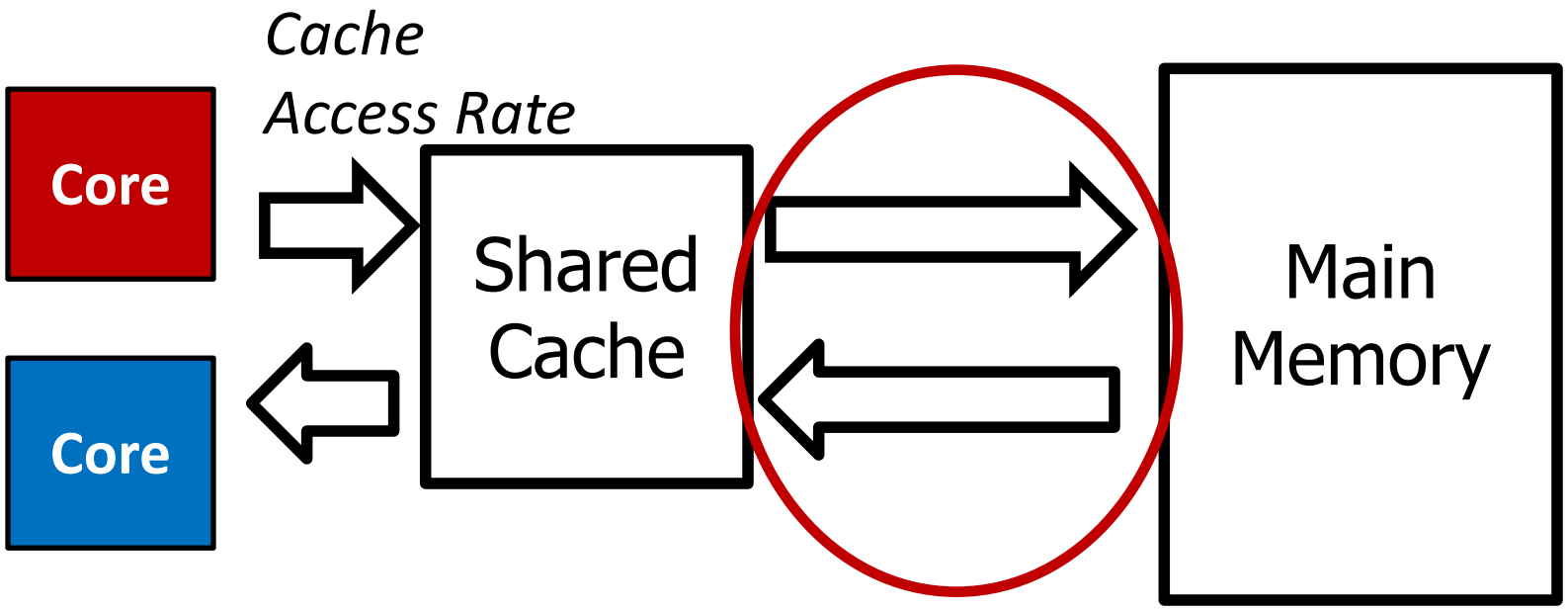
Previous partitioning schemes optimize for miss count

Problem: Not aware of performance and slowdowns

ASM-Cache: Slowdown-aware Cache Way Partitioning

- *Key Requirement: Slowdown estimates for all possible way partitions*
- *Extend ASM to estimate slowdown for all possible cache way allocations*
- *Key Idea: Allocate each way to the application whose slowdown reduces the most*

Memory Bandwidth Partitioning



Goal: Partition the main memory bandwidth among applications to mitigate contention

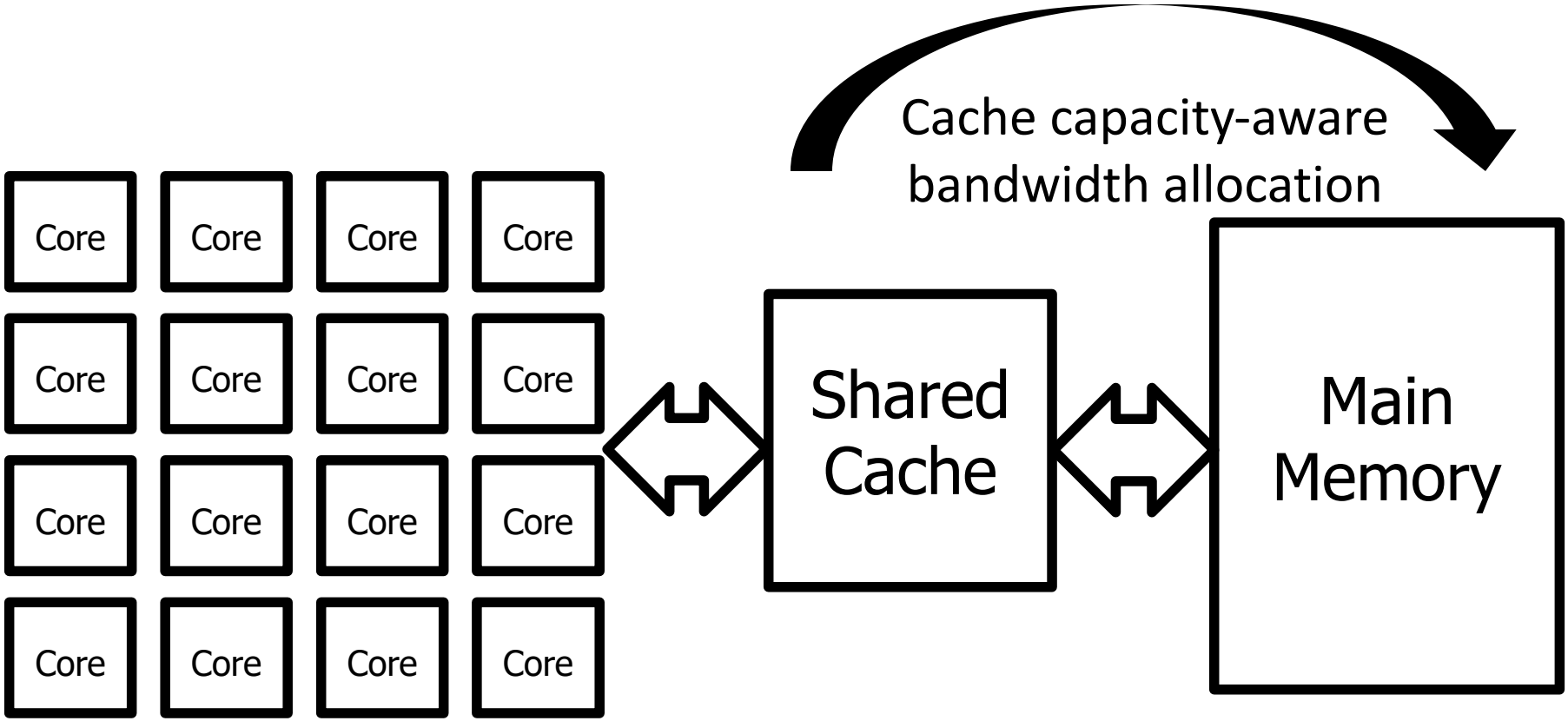
ASM-Mem: Slowdown-aware Memory Bandwidth Partitioning

- *Key Idea: Allocate high priority proportional to an application's slowdown*

$$\text{High Priority Fraction}_i = \frac{\text{Slowdown}_i}{\sum_j \text{Slowdown}_j}$$

- *Application i's requests given highest priority at the memory controller for its fraction*

Coordinated Resource Allocation Schemes

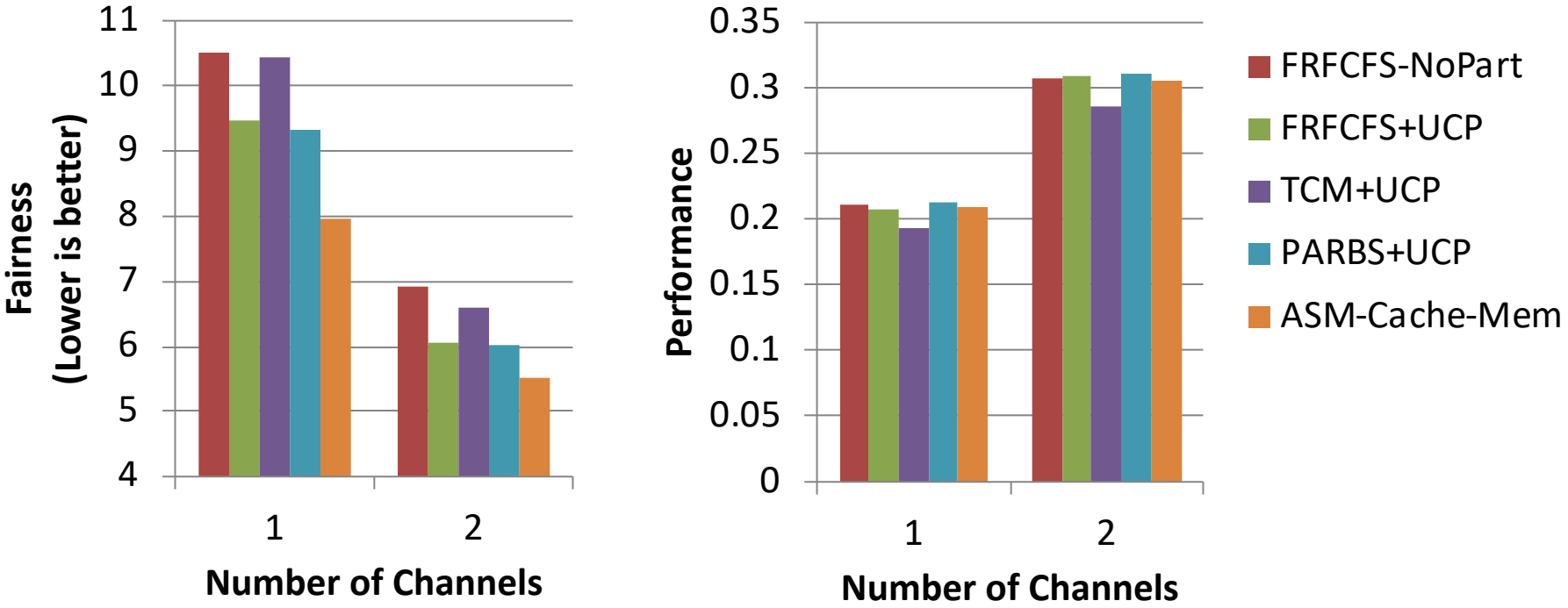


1. Employ ASM-Cache to partition cache capacity

2. Drive ASM-Mem with slowdowns from ASM-Cache

Fairness and Performance Results

16-core system
100 workloads



Significant fairness benefits across different channel counts

Summary

- Problem: Uncontrolled memory interference cause high and unpredictable application slowdowns
- Goal: Quantify and control slowdowns
- Key Contribution:
  - ASM: An accurate slowdown estimation model
  - Average error of ASM: 10%
- Key Ideas:
  - Shared cache access rate is a proxy for performance
  - Cache Access Rate $_{Alone}$ can be estimated by minimizing memory interference and quantifying cache interference
- Applications of Our Model
  - Slowdown-aware cache and memory management to achieve high performance, fairness and performance guarantees
- *Source Code Released in January 2016*

More on Application Slowdown Model

- Lavanya Subramanian, Vivek Seshadri, Arnab Ghosh, Samira Khan, and Onur Mutlu,
  "The Application Slowdown Model: Quantifying and Controlling the Impact of Inter-Application Interference at Shared Caches and Main Memory"
  *Proceedings of the 48th International Symposium on Microarchitecture (**MICRO**)*, Waikiki, Hawaii, USA, December 2015.
  [Slides (pptx) (pdf)] [Lightning Session Slides (pptx) (pdf)] [Poster (pptx) (pdf)]
  [Source Code]

The Application Slowdown Model: Quantifying and Controlling the Impact of Inter-Application Interference at Shared Caches and Main Memory

Lavanya Subramanian*§ Vivek Seshadri* Arnab Ghosh*†
Samira Khan*‡ Onur Mutlu*

*Carnegie Mellon University §Intel Labs †IIT Kanpur ‡University of Virginia