QoS-Aware Memory Systems (Wrap Up)

Onur Mutlu
onur@cmu.edu
July 9, 2013
INRIA

Carnegie Mellon

SAFARI

Slides for These Lectures

- Architecting and Exploiting Asymmetry in Multi-Core
  - http://users.ece.cmu.edu/~omutlu/pub/onur-INRIA-lecture1-asymmetry-jul-2-2013.pptx
- A Fresh Look At DRAM Architecture
  - http://www.ece.cmu.edu/~omutlu/pub/onur-INRIA-lecture2-DRAM-jul-4-2013.pptx
- QoS-Aware Memory Systems
  - http://users.ece.cmu.edu/~omutlu/pub/onur-INRIA-lecture3-memory-qos-jul-8-2013.pptx
- QoS-Aware Memory Systems and Waste Management
  - http://users.ece.cmu.edu/~omutlu/pub/onur-INRIA-lecture4-memory-qos-and-waste-management-jul-9-2013.pptx

Videos for Similar Lectures

- Basics (of Computer Architecture)
  - http://www.youtube.com/playlist?list=PL5PHm2jkkXmidJOd59REog9jDnPDTG6IJ
- Advanced (Longer versions of these lectures)
  - http://www.youtube.com/playlist?list=PLVngZ7BemHHV6N0ejHhwOfLwTr8Q-UKXj

Designing QoS-Aware Memory Systems: Approaches

- Smart resources: Design each shared resource to have a configurable interference control/reduction mechanism
  - QoS-aware memory controllers [Mutlu+ MICRO'07] [Moscibroda+, Usenix Security'07] [Mutlu+ ISCA'08, Top Picks'09] [Kim+ HPCA'10] [Kim+ MICRO'10, Top Picks'11] [Ebrahimi+ ISCA'11, MICRO'11] [Ausavarungnirun+, ISCA'12][Subramanian+, HPCA'13]
  - QoS-aware interconnects [Das+ MICRO'09, ISCA'10, Top Picks '11] [Grot+ MICRO'09, ISCA'11, Top Picks '12]
  - QoS-aware caches
- Dumb resources: Keep each resource free-for-all, but reduce/control interference by injection control or data mapping
  - Source throttling to control access to memory system [Ebrahimi+ ASPLOS'10, ISCA'11, TOCS'12] [Ebrahimi+ MICRO'09] [Nychis+ HotNets'10] [Nychis+ SIGCOMM'12]
  - QoS-aware data mapping to memory controllers [Muralidhara+ MICRO'11]
  - QoS-aware thread scheduling to cores [Das+ HPCA'13]

ATLAS Pros and Cons

- Upsides:
  - Good at improving overall throughput (compute-intensive threads are prioritized)
  - Low complexity
  - Coordination among controllers happens infrequently

- Downsides:
  - Lowest/medium ranked threads get delayed significantly → high unfairness

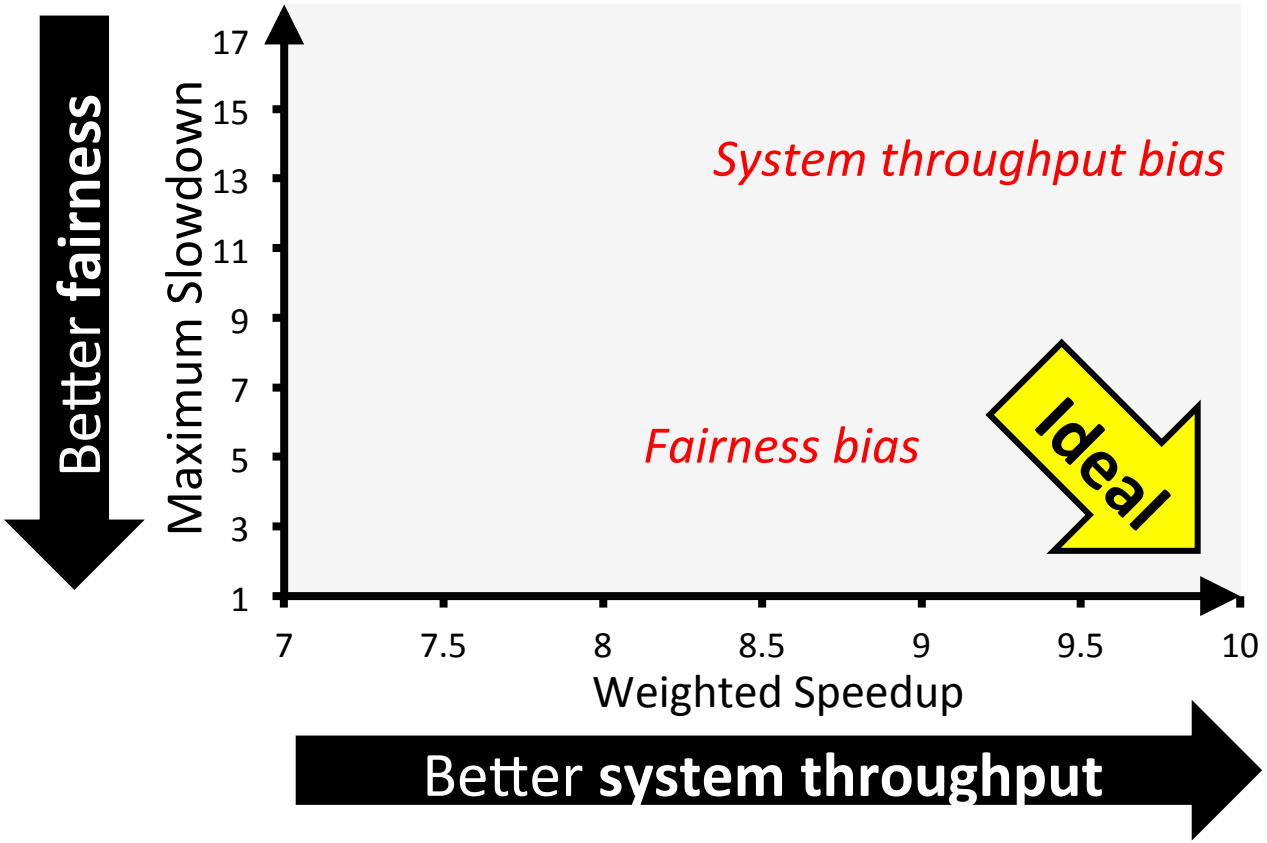
TCM:
Thread Cluster Memory Scheduling

Yoongu Kim, Michael Papamichael, Onur Mutlu, and Mor Harchol-Balter,
**"Thread Cluster Memory Scheduling:
Exploiting Differences in Memory Access Behavior"**
43rd International Symposium on Microarchitecture (***MICRO***),
pages 65-76, Atlanta, GA, December 2010. Slides (pptx) (pdf)

TCM Micro 2010 Talk

Previous Scheduling Algorithms are Biased

24 cores, 4 memory controllers, 96 workloads



No previous memory scheduling algorithm provides both the best fairness and system throughput

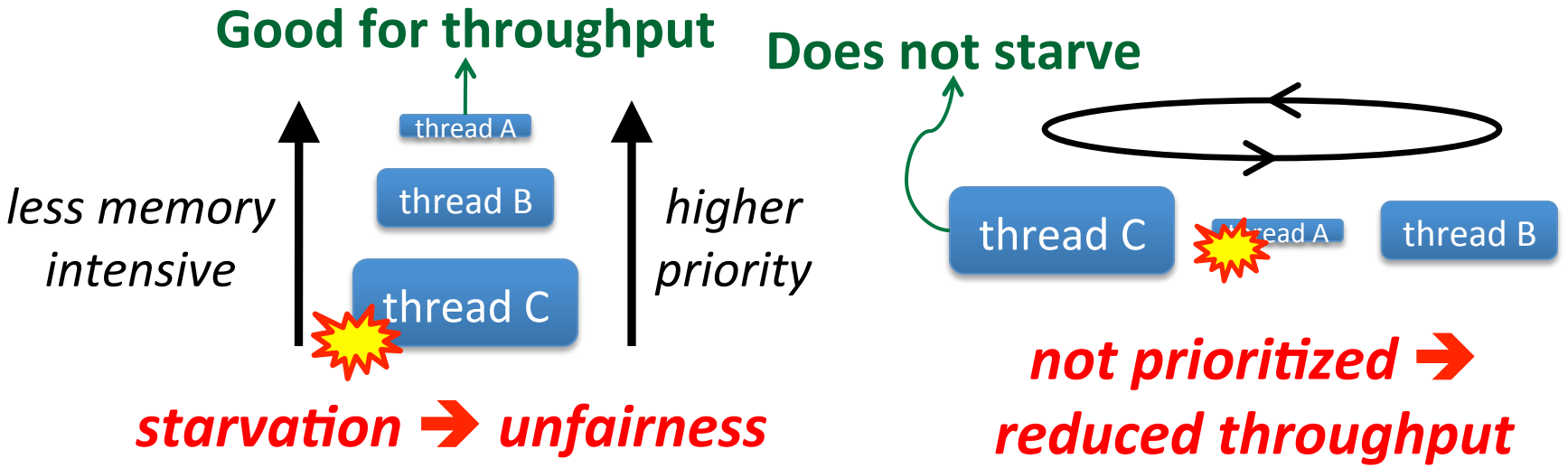
Throughput vs. Fairness

***Throughput biased** approach*

Prioritize less memory-intensive threads

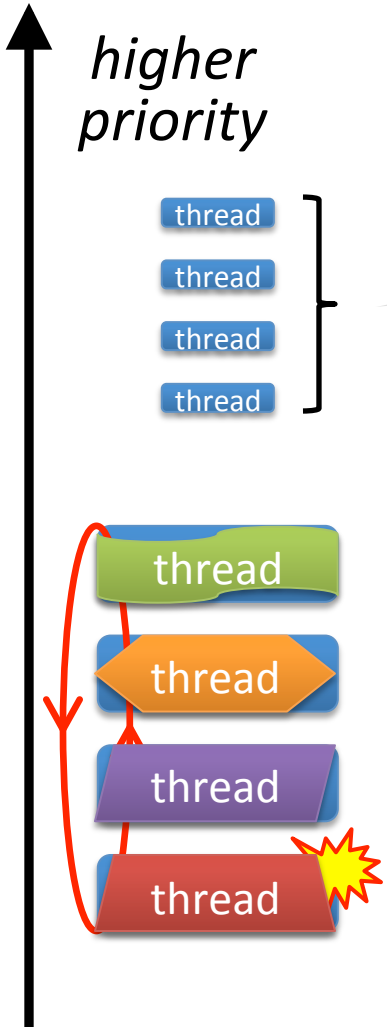
***Fairness biased** approach*

Take turns accessing memory



Single policy for all threads is insufficient

Achieving the Best of Both Worlds



For Throughput

Prioritize memory-non-intensive threads

For Fairness

Unfairness caused by memory-intensive being prioritized over each other

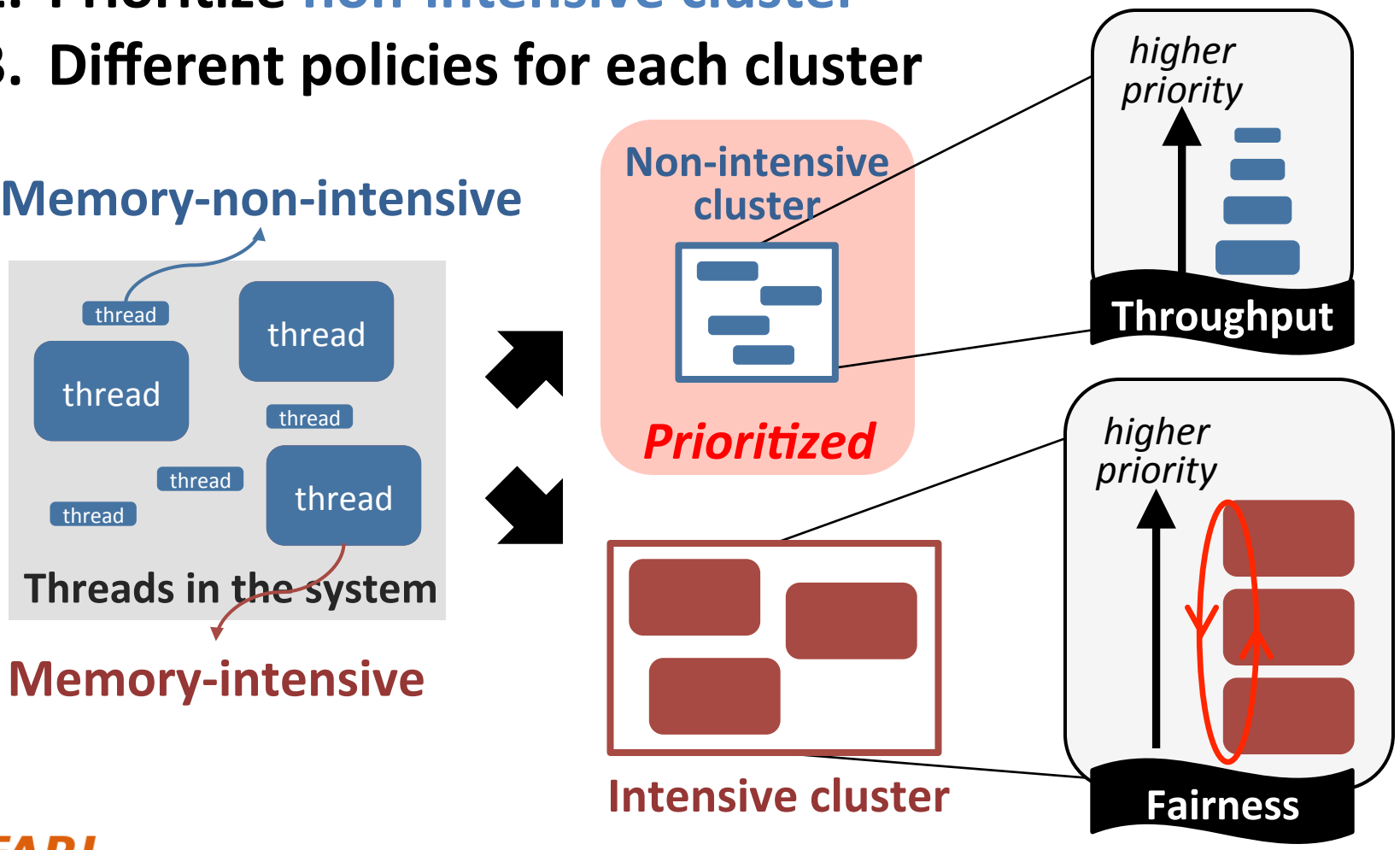
- Shuffle thread ranking

Memory-intensive threads have different vulnerability to interference

- Shuffle asymmetrically

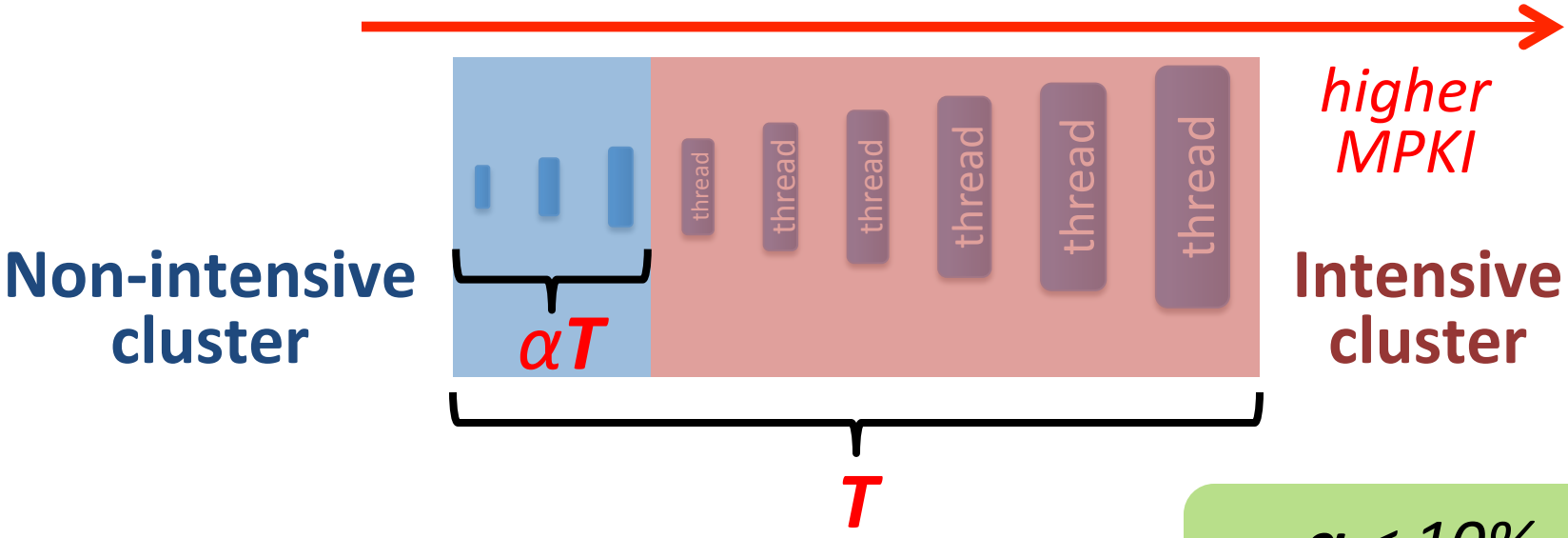
Thread Cluster Memory Scheduling [Kim+ MICRO'10]

1. Group threads into two *clusters*
2. Prioritize non-intensive cluster
3. Different policies for each cluster



Clustering Threads

Step1 Sort threads by **MPKI** (misses per kiloinstruction)



T = Total ***memory bandwidth usage***

α < 10%
ClusterThreshold

Step2 Memory bandwidth usage ***αT*** divides clusters

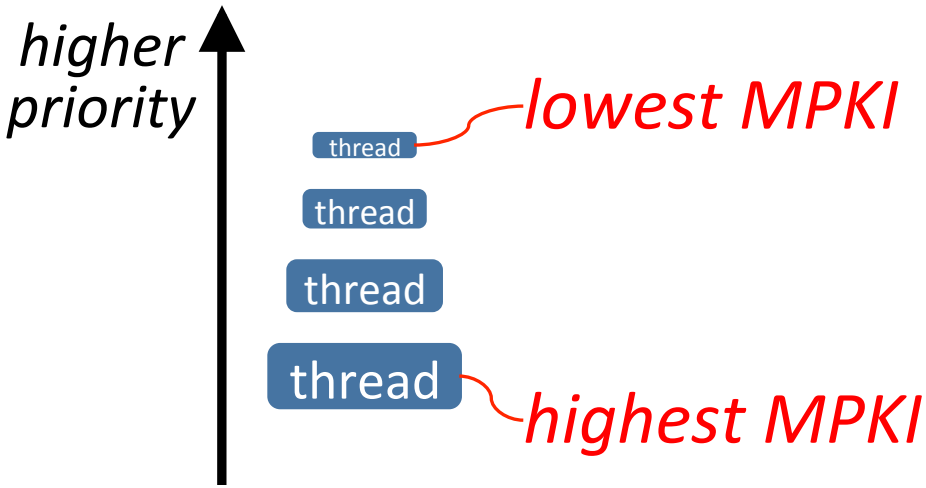
Prioritization Between Clusters

Prioritize non-intensive cluster

- **Increases system throughput**
  - Non-intensive threads have greater potential for making progress
- **Does not degrade fairness**
  - Non-intensive threads are "light"
  - Rarely interfere with intensive threads

Non-Intensive Cluster

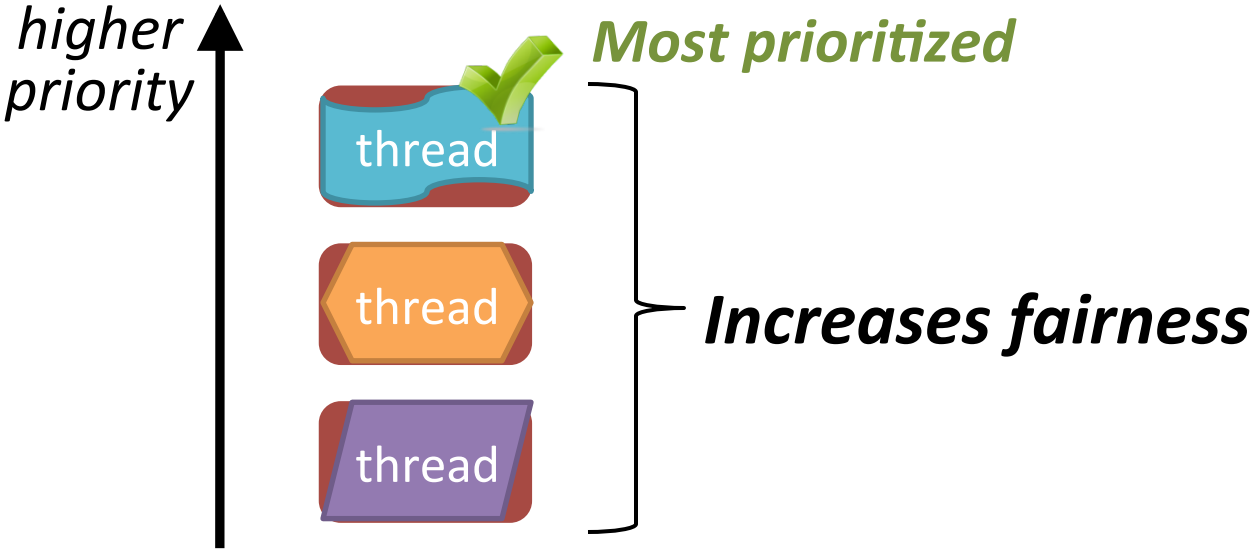
Prioritize threads according to MPKI



- **Increases system throughput**
  - Least intensive thread has the greatest potential for making progress in the processor

Intensive Cluster

Periodically shuffle the priority of threads



- Is treating all threads equally good enough?
- ***BUT:*** *Equal turns ≠ Same slowdown*

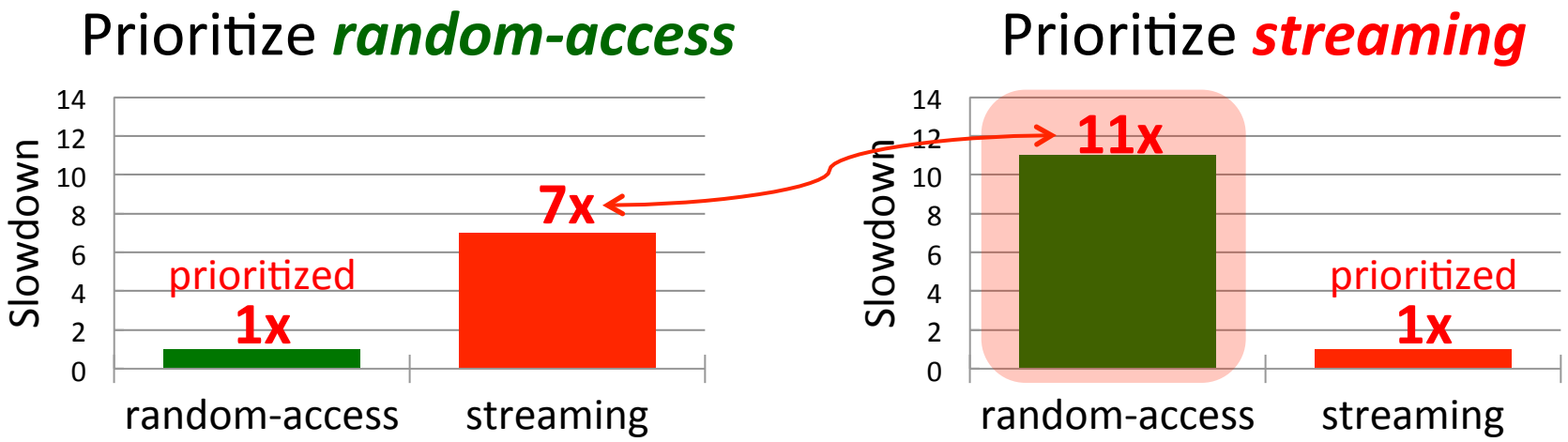
Case Study: A Tale of Two Threads

Case Study: Two intensive threads contending

1.random-access

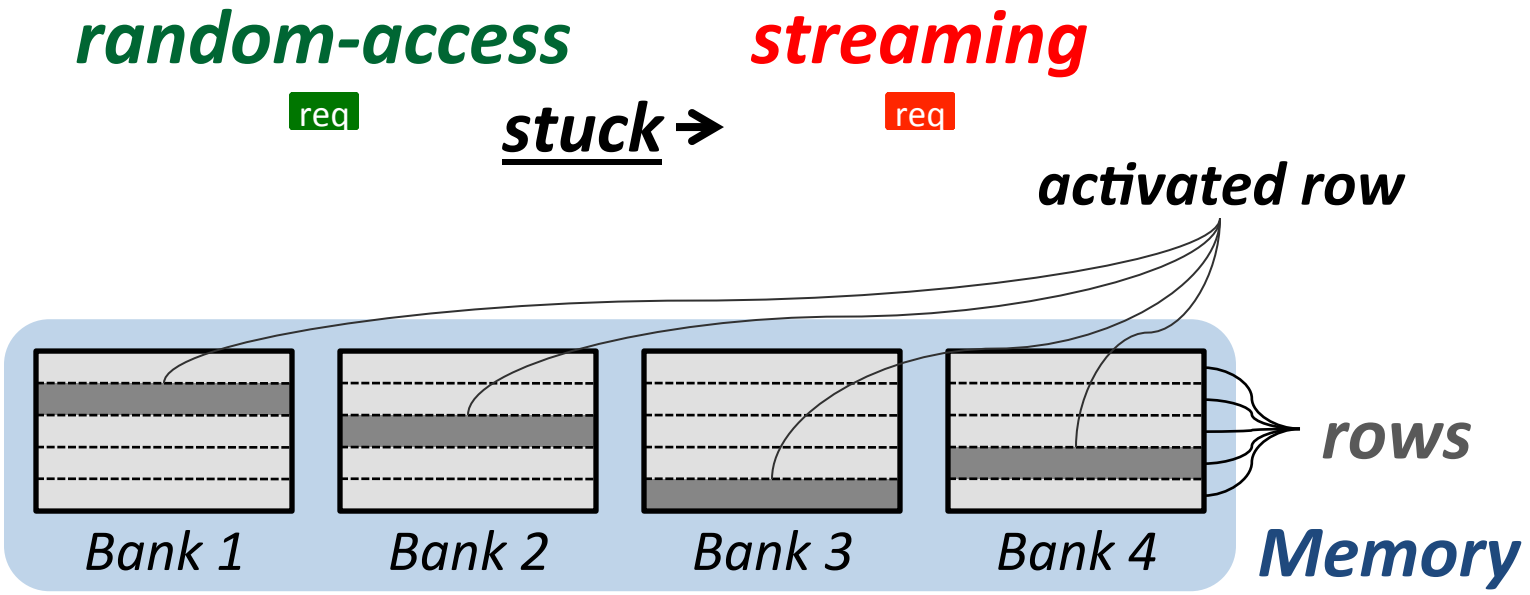
2.streaming

Which is slowed down more easily?



random-access thread is more easily slowed down

Why are Threads Different?



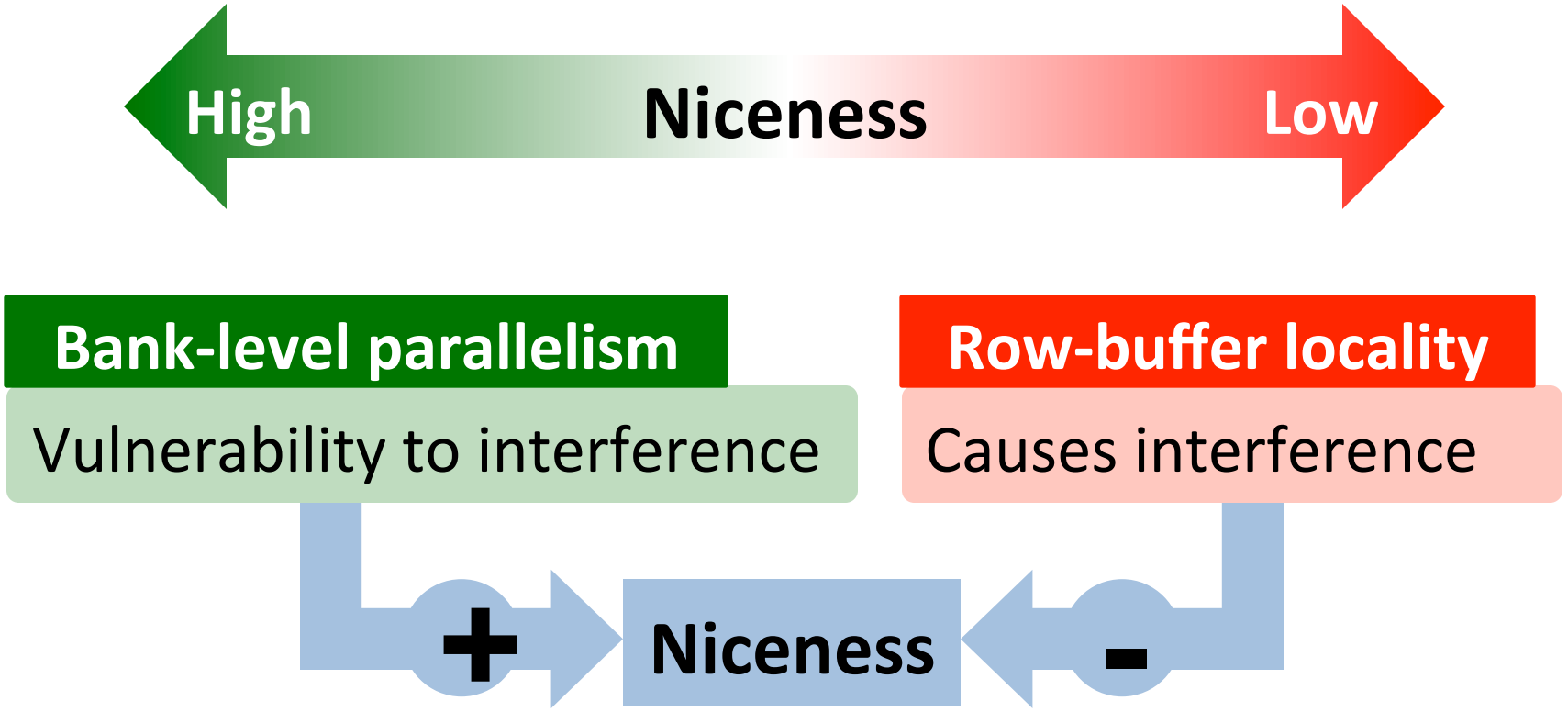
- All requests parallel
- High **bank-level parallelism**

- All requests ➔ Same row
- High **row-buffer locality**

Vulnerable to interference

Niceness

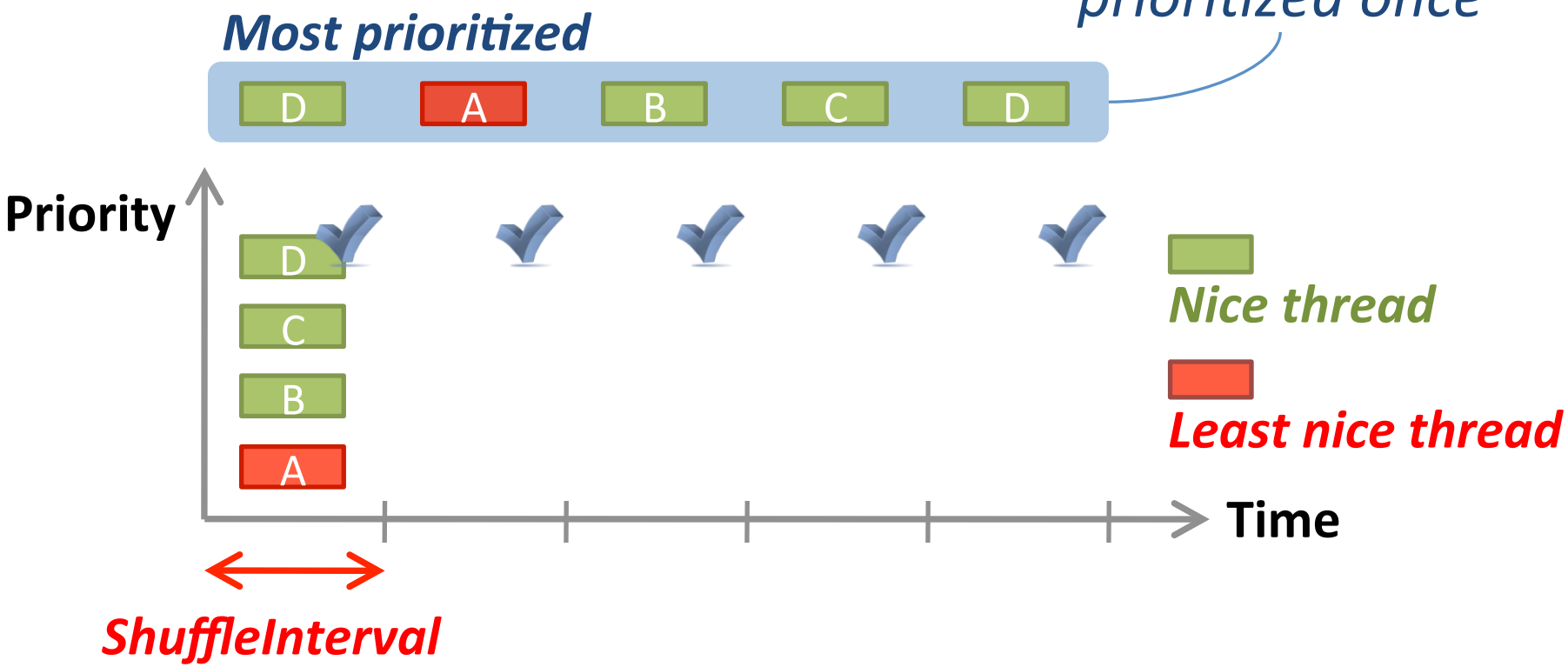
How to quantify difference between threads?



Shuffling: Round-Robin vs. Niceness-Aware

1. ***Round-Robin*** *shuffling* ← *What can go wrong?*
2. ***Niceness-Aware*** *shuffling*

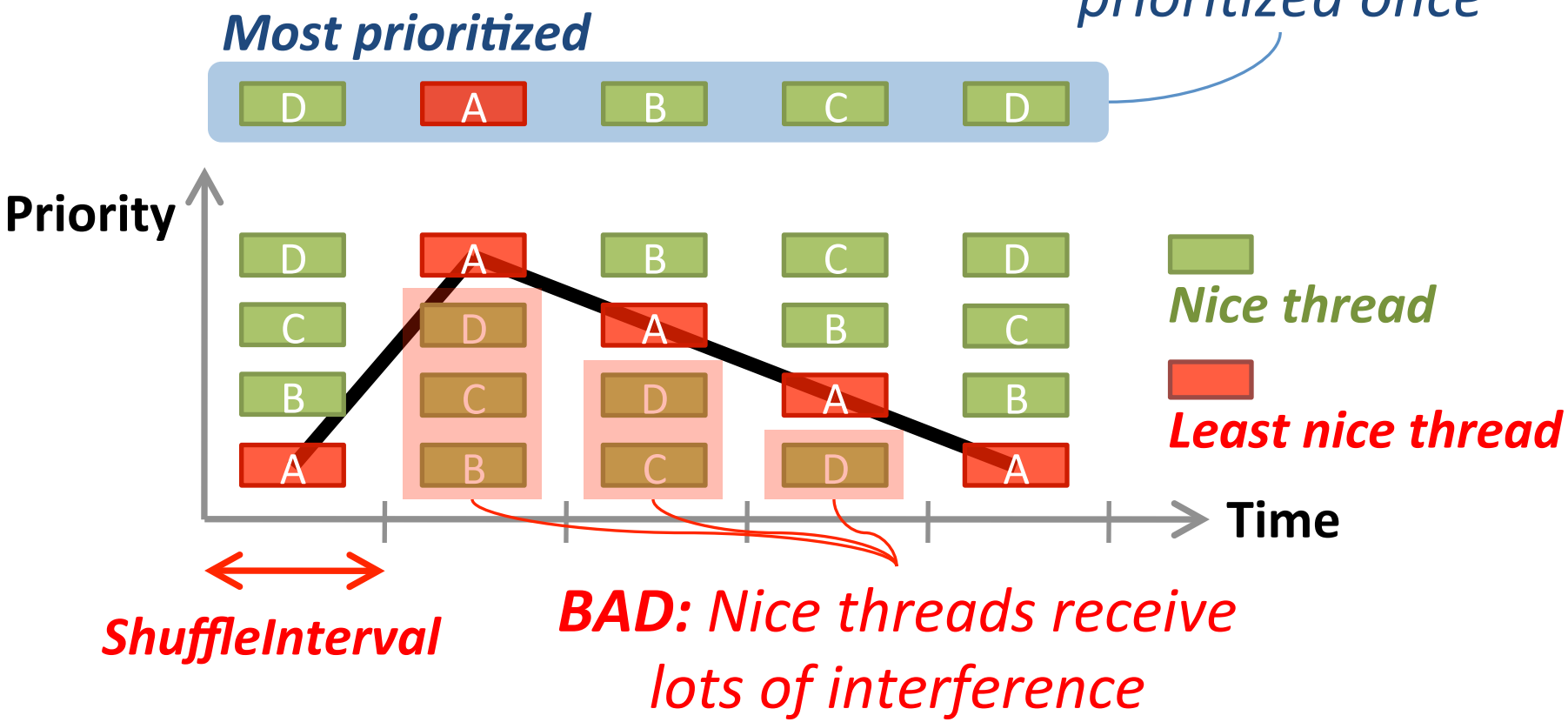
GOOD: *Each thread prioritized once*



Shuffling: Round-Robin vs. Niceness-Aware

1. **Round-Robin** *shuffling* ← *What can go wrong?*
2. **Niceness-Aware** *shuffling*

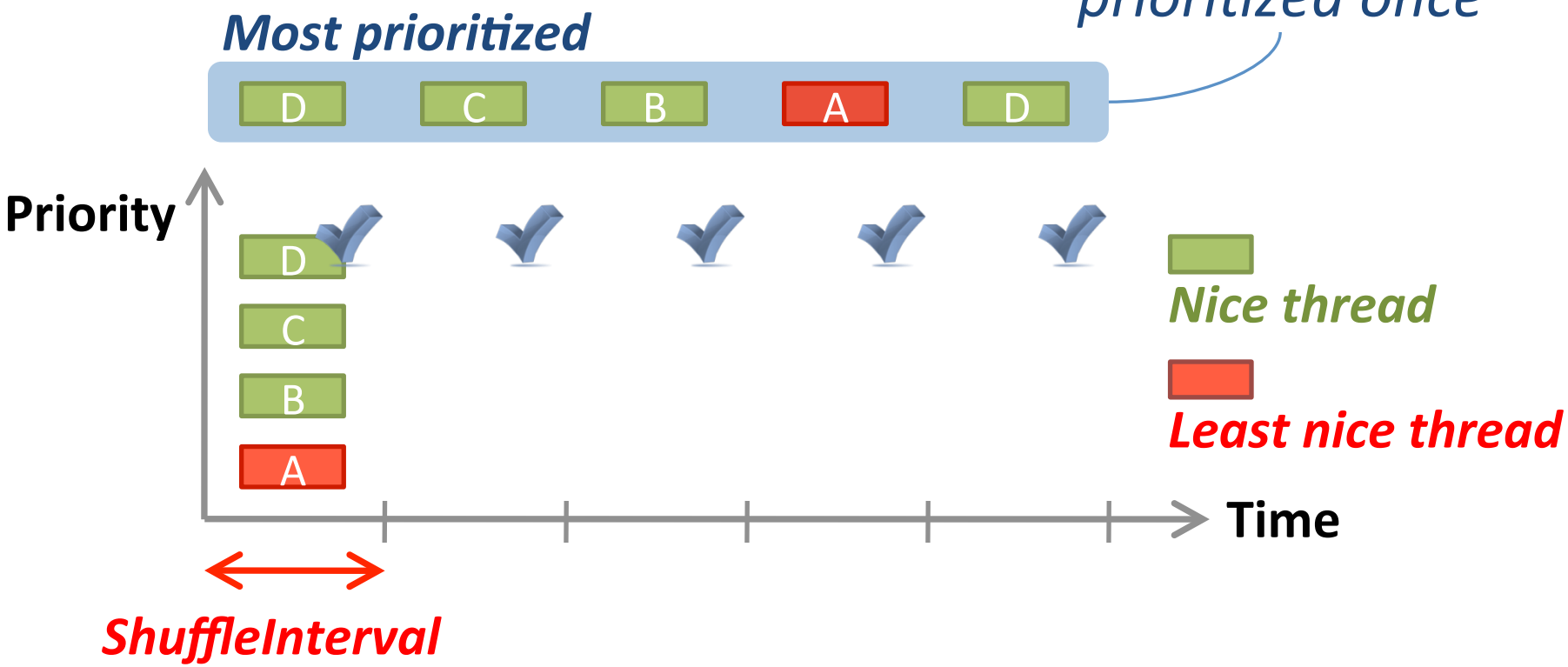
GOOD: *Each thread prioritized once*



Shuffling: Round-Robin vs. Niceness-Aware

1. ***Round-Robin*** *shuffling*
2. ***Niceness-Aware*** *shuffling*

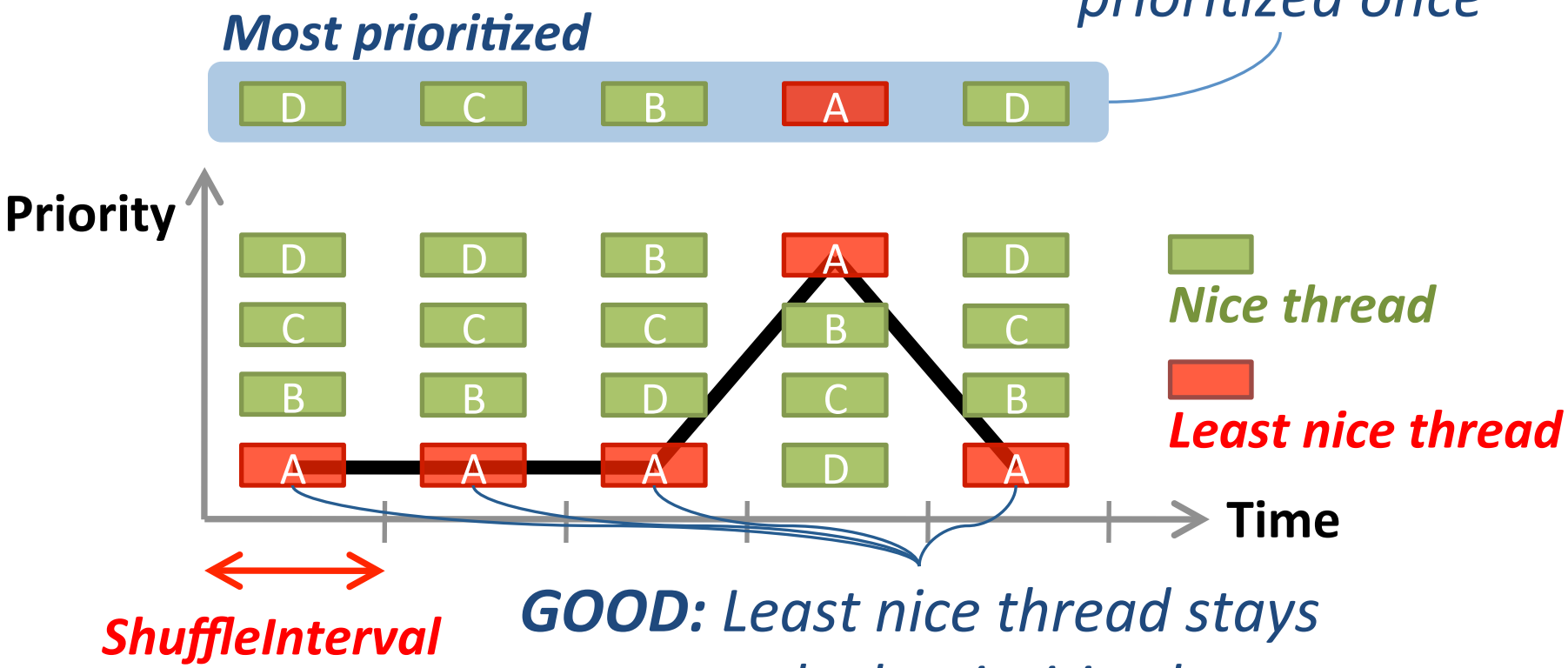
GOOD: *Each thread prioritized once*



Shuffling: Round-Robin vs. Niceness-Aware

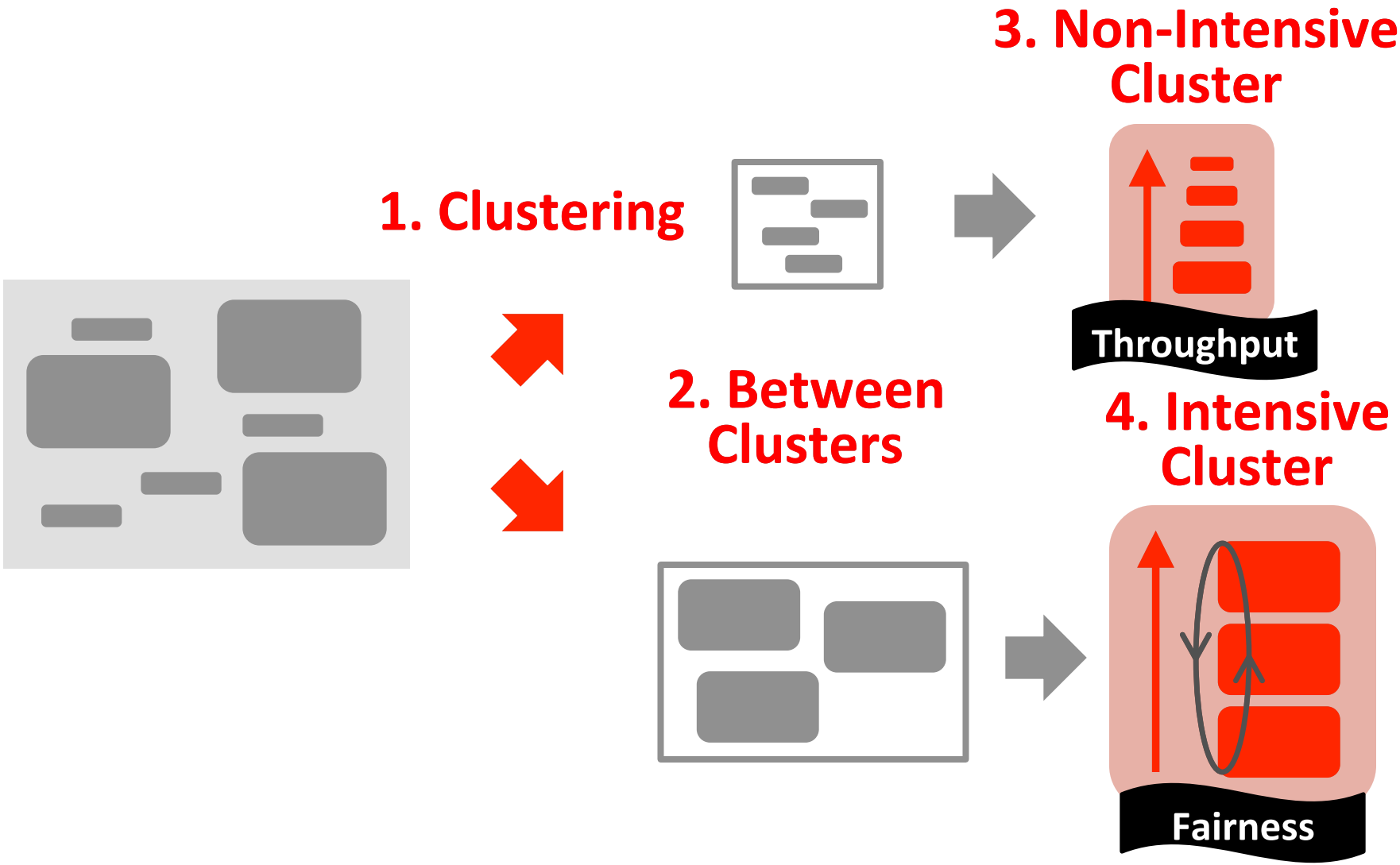
1. ***Round-Robin** shuffling*
2. ***Niceness-Aware** shuffling*

***GOOD:** Each thread prioritized once*

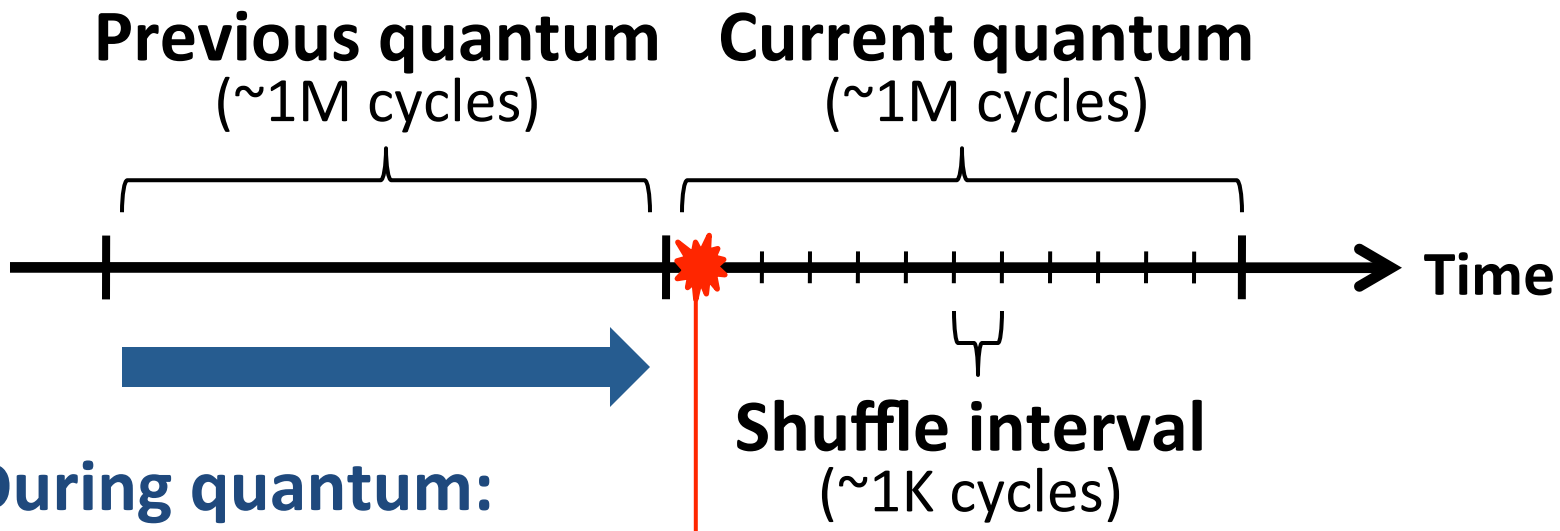


***GOOD:** Least nice thread stays mostly deprioritized*

TCM Outline


3. Non-Intensive Cluster
1. Clustering
Throughput
2. Between Clusters
4. Intensive Cluster
Fairness

TCM: Quantum-Based Operation



During quantum:

- Monitor thread behavior
  1. Memory intensity
  2. Bank-level parallelism
  3. Row-buffer locality

Beginning of quantum:

- Perform clustering
- Compute niceness of intensive threads

TCM: Scheduling Algorithm

1. ***Highest-rank***: Requests from higher ranked threads prioritized
   - **Non-Intensive** cluster **>** **Intensive** cluster
   - **Non-Intensive** cluster: lower intensity ➔ higher rank
   - **Intensive** cluster: rank shuffling

2. ***Row-hit***: Row-buffer hit requests are prioritized

3. ***Oldest***: Older requests are prioritized

TCM: Implementation Cost

Required storage at memory controller *(24 cores)*

Thread memory behavior	Storage
MPKI	~0.2kb
Bank-level parallelism	~0.6kb
Row-buffer locality	~2.9kb
Total	**< 4kbits**

- No computation is on the critical path

Previous Work

FRFCFS [Rixner et al., ISCA00]: Prioritizes row-buffer hits

- Thread-oblivious → Low throughput & Low fairness

STFM [Mutlu et al., MICRO07]: Equalizes thread slowdowns

- Non-intensive threads not prioritized → Low throughput

PAR-BS [Mutlu et al., ISCA08]: Prioritizes oldest batch of requests while preserving bank-level parallelism

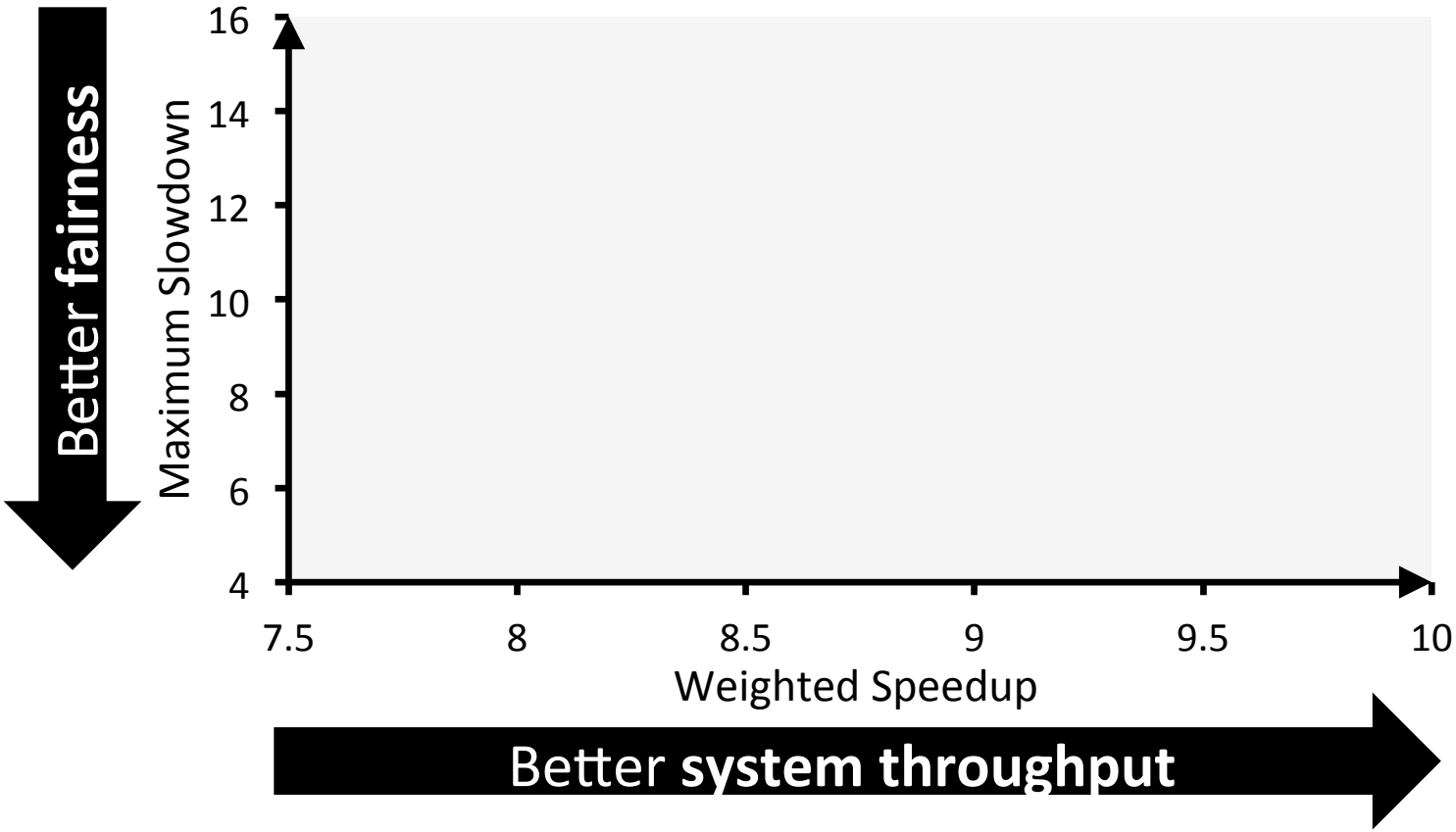
- Non-intensive threads not always prioritized → Low throughput

ATLAS [Kim et al., HPCA10]: Prioritizes threads with less memory service

- Most intensive thread starves → Low fairness

TCM: Throughput and Fairness

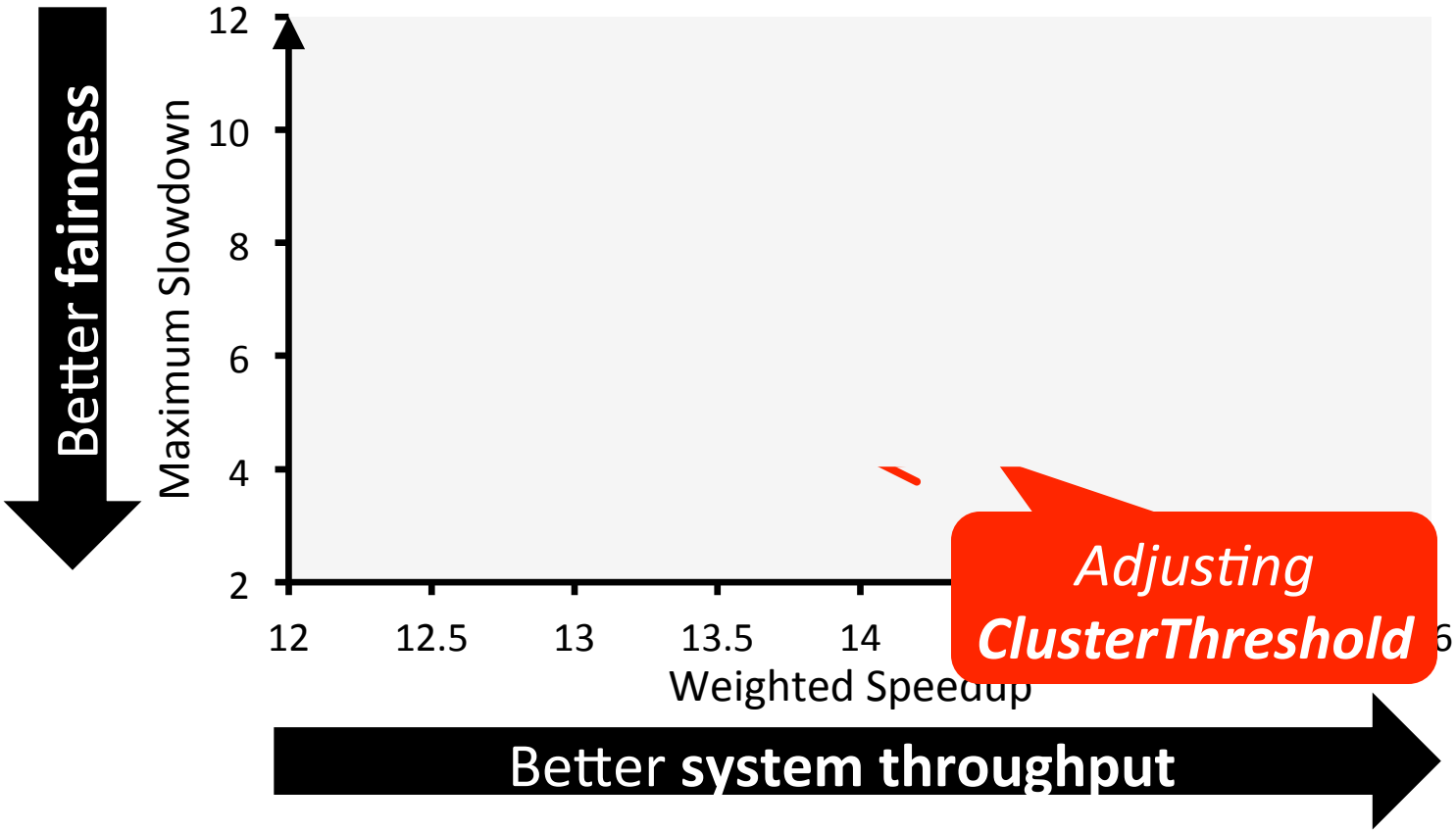
24 cores, 4 memory controllers, 96 workloads



TCM, a heterogeneous scheduling policy, provides best fairness and system throughput

TCM: Fairness-Throughput Tradeoff

When configuration parameter is varied…



TCM allows robust fairness-throughput tradeoff

Operating System Support

- ***ClusterThreshold*** is a tunable knob
  - OS can trade off between fairness and throughput

- Enforcing thread weights
  - OS assigns weights to threads
  - TCM enforces thread weights within each cluster

Conclusion

- No previous memory scheduling algorithm provides both high ***system throughput*** and ***fairness***
  - **Problem:** They use a single policy for all threads

- TCM groups threads into two ***clusters***
  1. Prioritize ***non-intensive*** cluster → throughput
  2. Shuffle priorities in ***intensive*** cluster → fairness
  3. Shuffling should favor ***nice*** threads → fairness

- *TCM provides the best system throughput and fairness*

TCM Pros and Cons

- Upsides:
  - Provides both high fairness and high performance
- Downsides:
  - Scalability to large buffer sizes?
  - Effectiveness in a heterogeneous system?

Staged Memory Scheduling

Rachata Ausavarungnirun, Kevin Chang, Lavanya Subramanian, Gabriel Loh, and Onur Mutlu,

"Staged Memory Scheduling: Achieving High Performance and Scalability in Heterogeneous Systems"

39th International Symposium on Computer Architecture (***ISCA***),
Portland, OR, June 2012.

SMS ISCA 2012 Talk

SMS: Executive Summary

- **Observation:** Heterogeneous CPU-GPU systems require memory schedulers with large request buffers
- **Problem:** Existing monolithic application-aware memory scheduler designs are hard to scale to large request buffer sizes
- **Solution:** Staged Memory Scheduling (SMS)

  decomposes the memory controller into three simple stages:

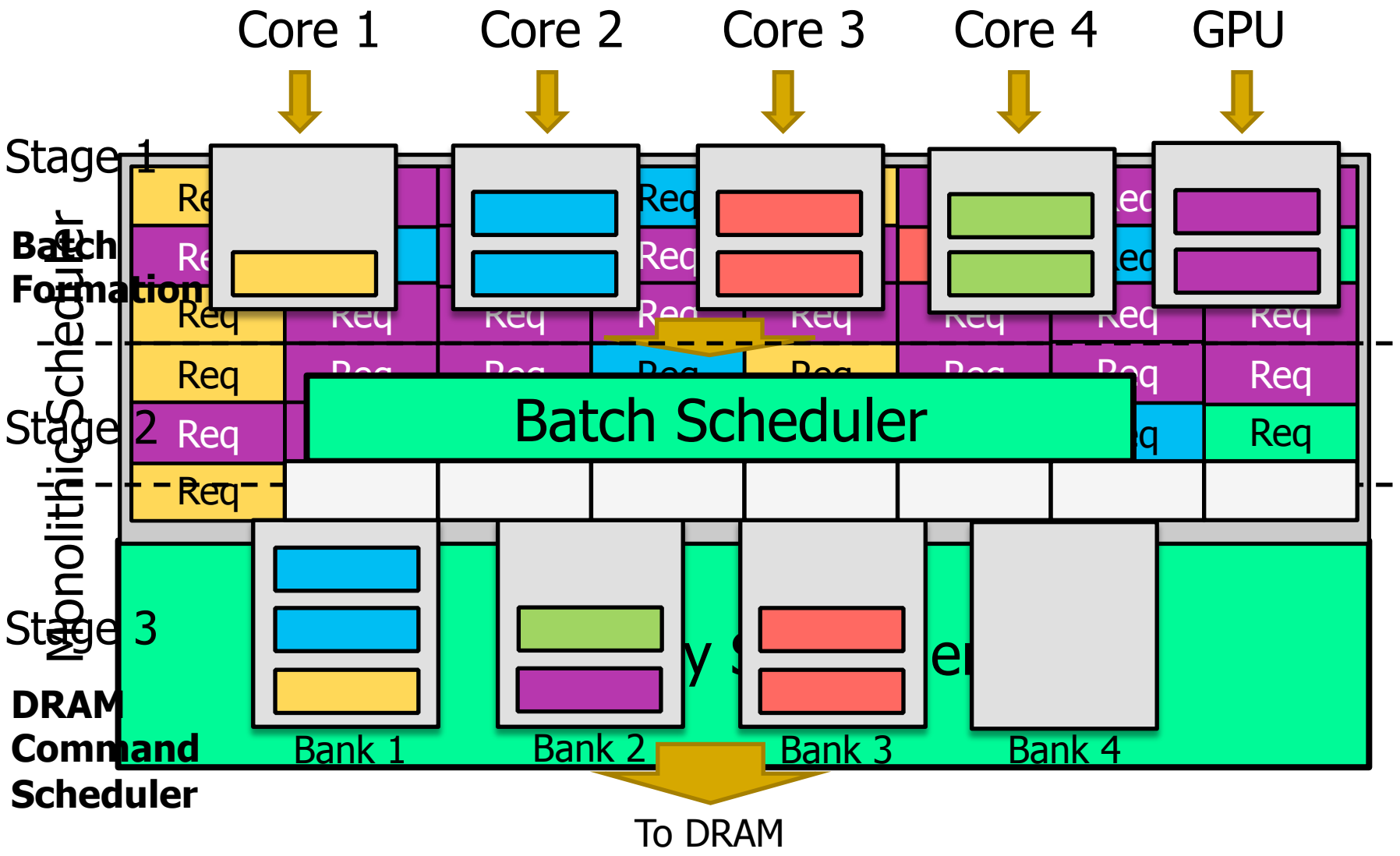
  1) Batch formation: maintains row buffer locality

  2) Batch scheduler: reduces interference between applications

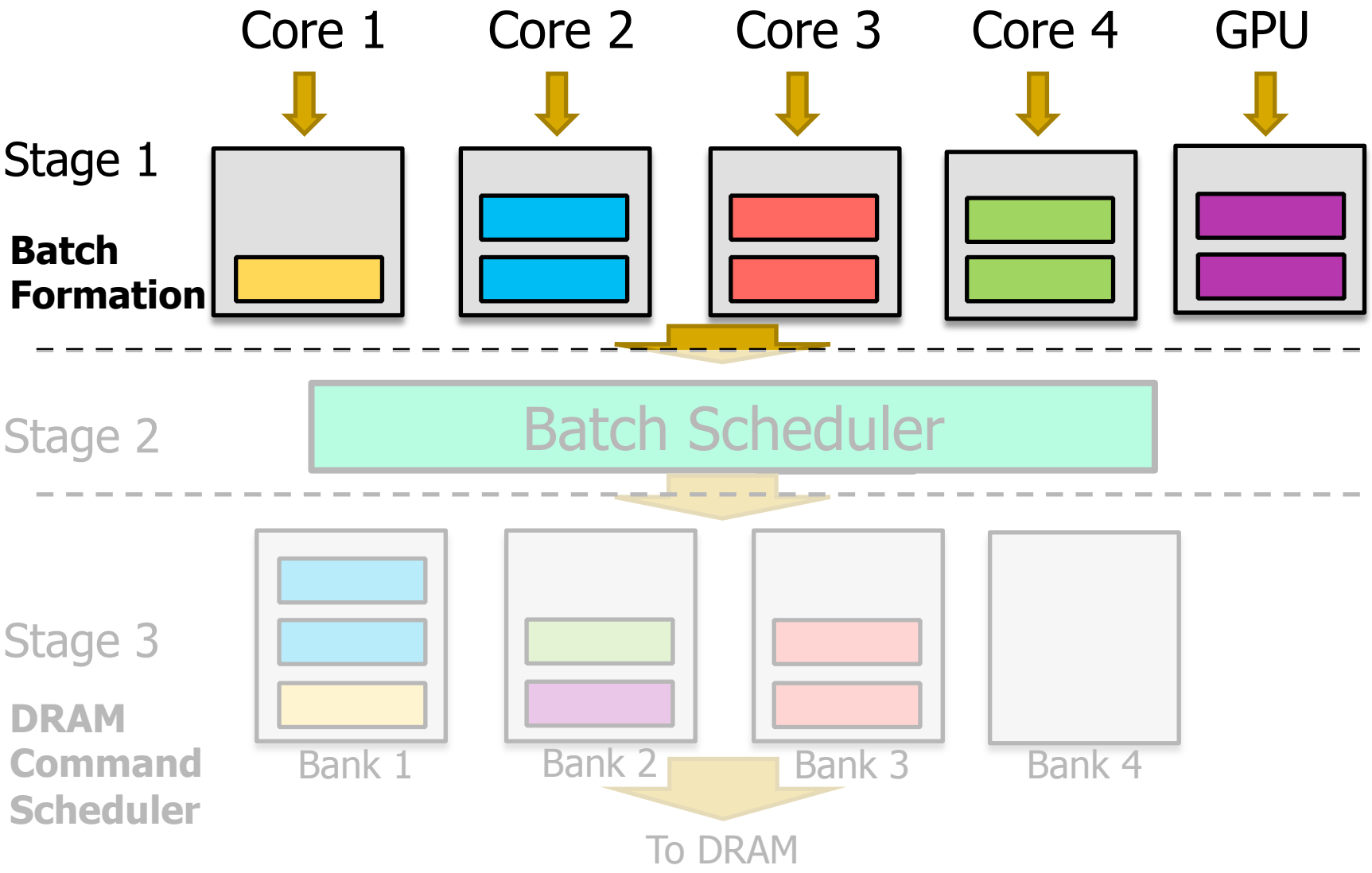
  3) DRAM command scheduler: issues requests to DRAM

- Compared to state-of-the-art memory schedulers:
  - SMS is significantly simpler and more scalable
  - SMS provides higher performance and fairness

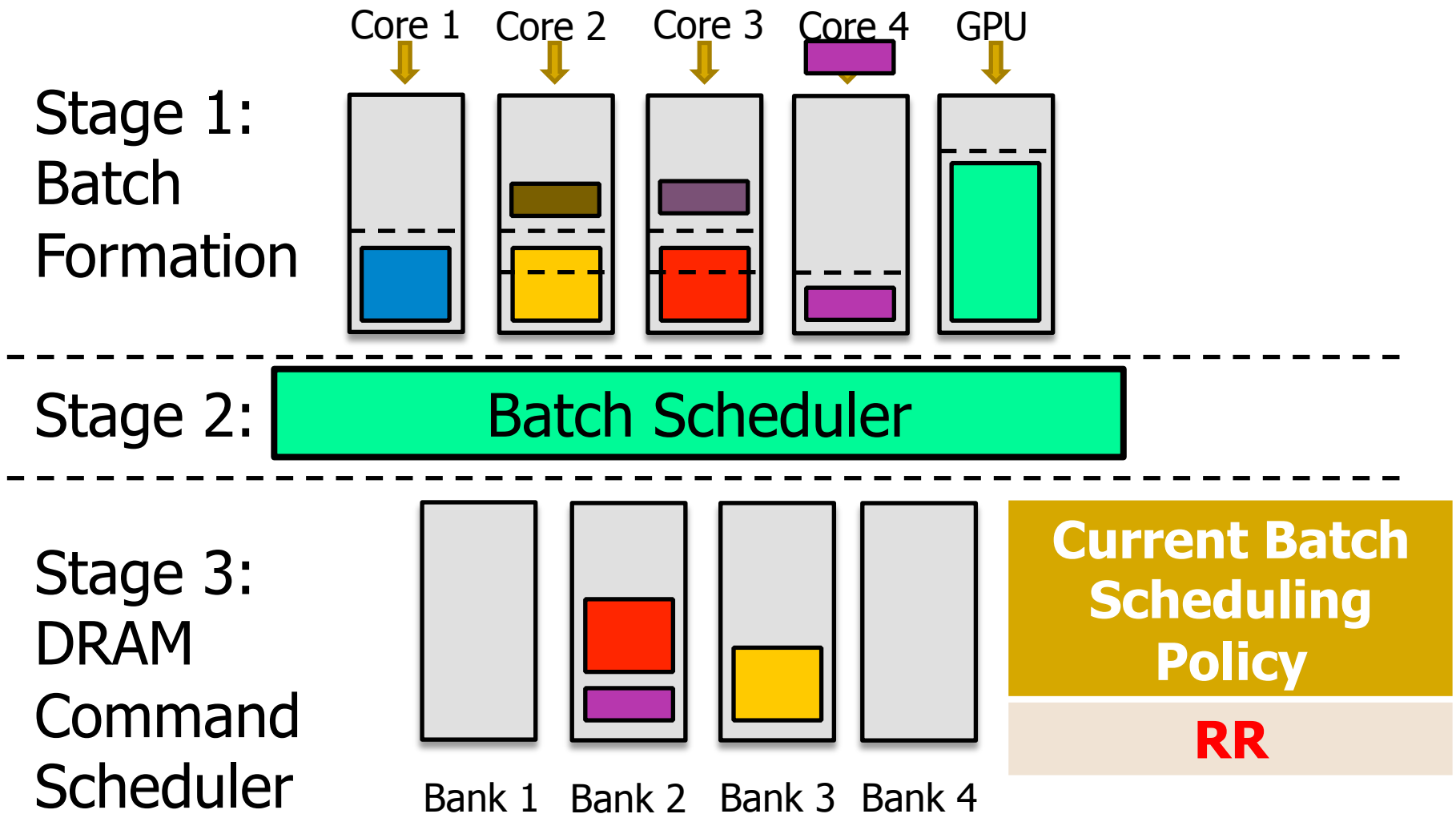
SMS: Staged Memory Scheduling



SMS: Staged Memory Scheduling



Putting Everything Together



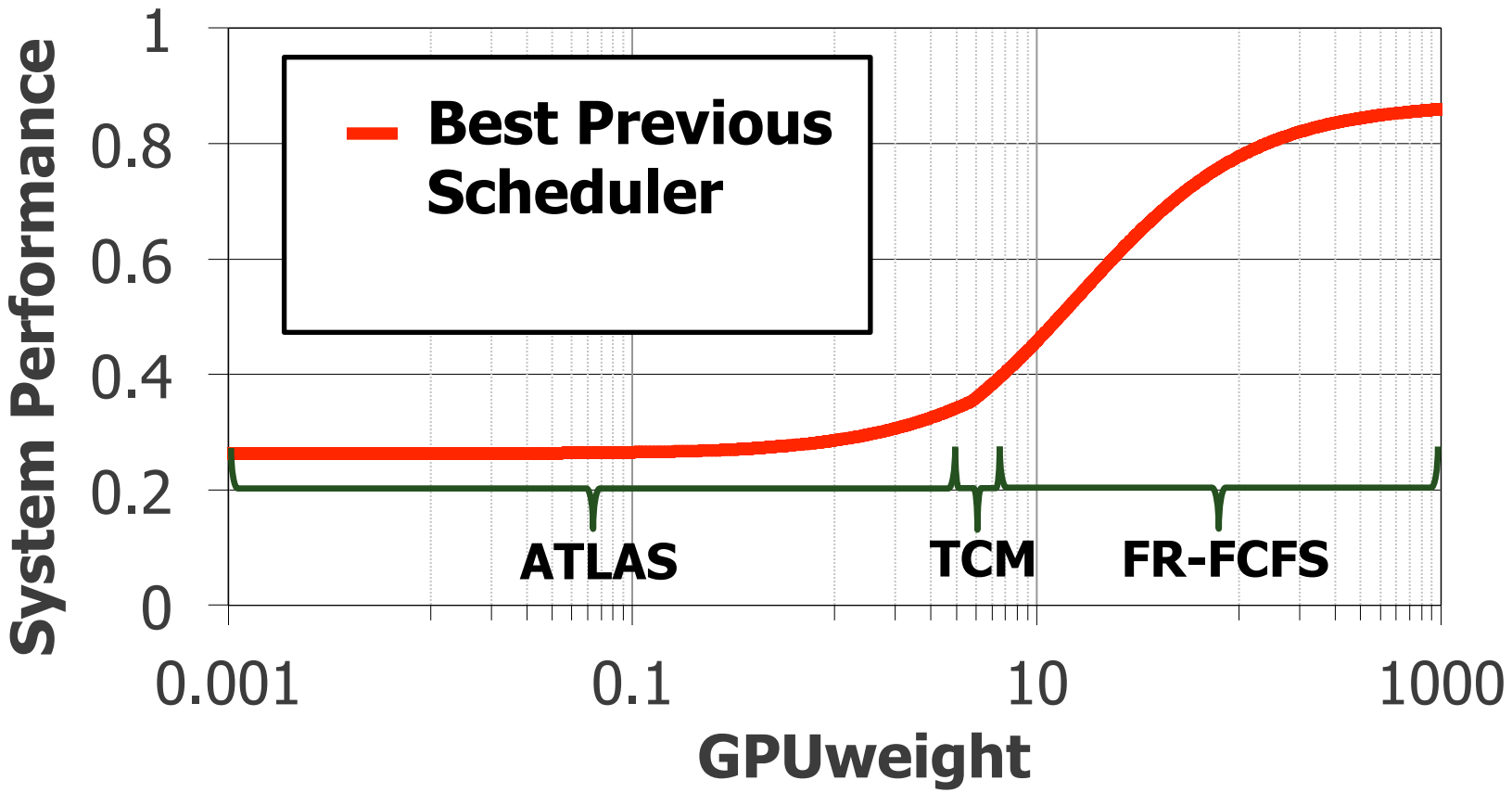
Complexity

- Compared to a row hit first scheduler, SMS consumes*
  - 66% less area
  - 46% less static power

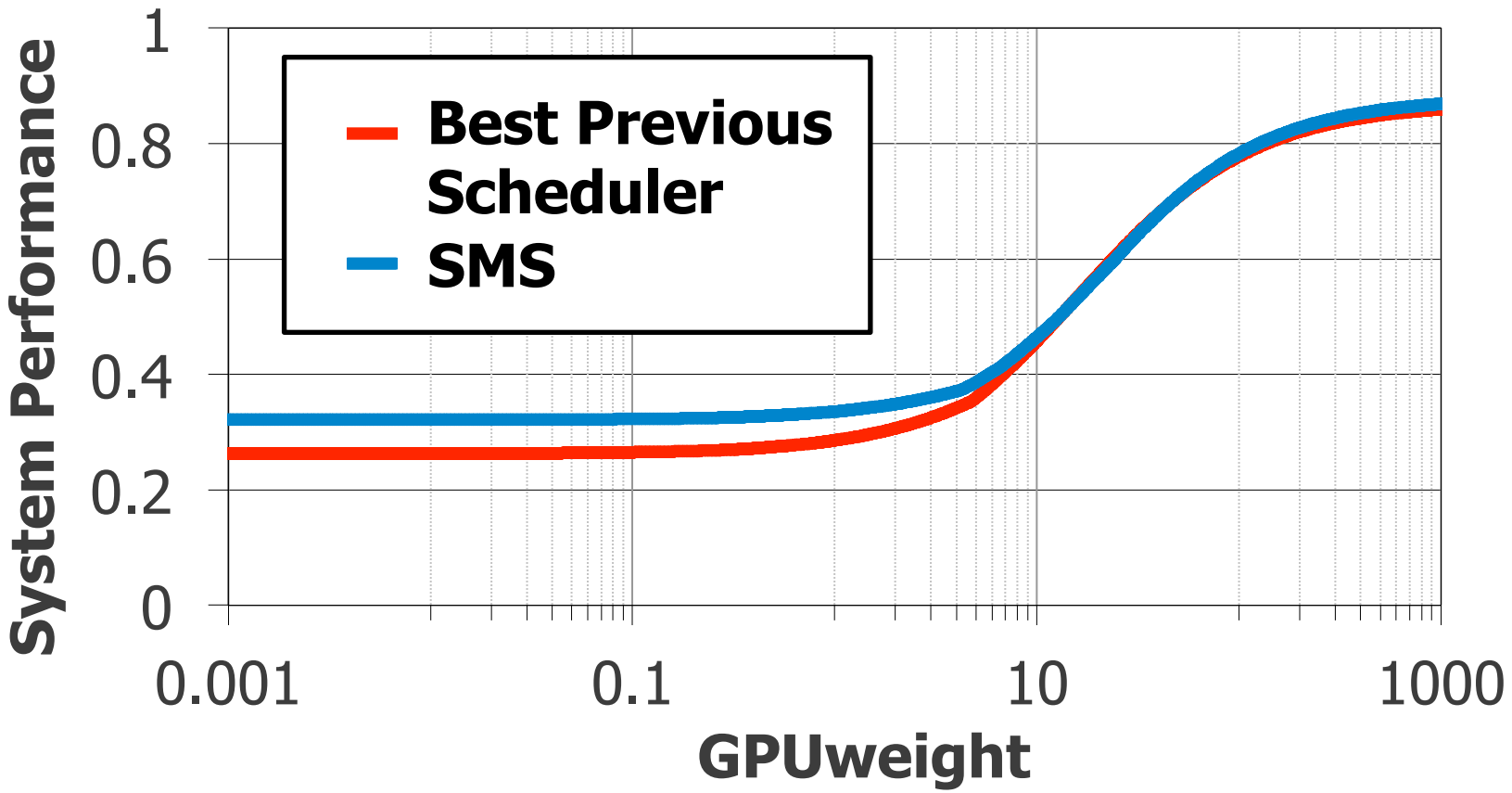
- Reduction comes from:
  - Monolithic scheduler → stages of simpler schedulers
  - Each stage has a simpler scheduler (considers fewer properties at a time to make the scheduling decision)
  - Each stage has simpler buffers (FIFO instead of out-of-order)
  - Each stage has a portion of the total buffer size (buffering is distributed across stages)

*** Based on a Verilog model using 180nm library**

Performance at Different GPU Weights



Performance at Different GPU Weights



- At every GPU weight, SMS outperforms the best previous scheduling algorithm for that weight

Stronger Memory Service Guarantees

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)

Strong Memory Service Guarantees

- Goal: Satisfy performance bounds/requirements in the presence of shared main memory, prefetchers, heterogeneous agents, and hybrid memory

- Approach:
  - Develop techniques/models to accurately estimate the performance of an application/agent in the presence of resource sharing
  - Develop mechanisms (hardware and software) to enable the resource partitioning/prioritization needed to achieve the required performance levels for all applications
  - All the while providing high system performance

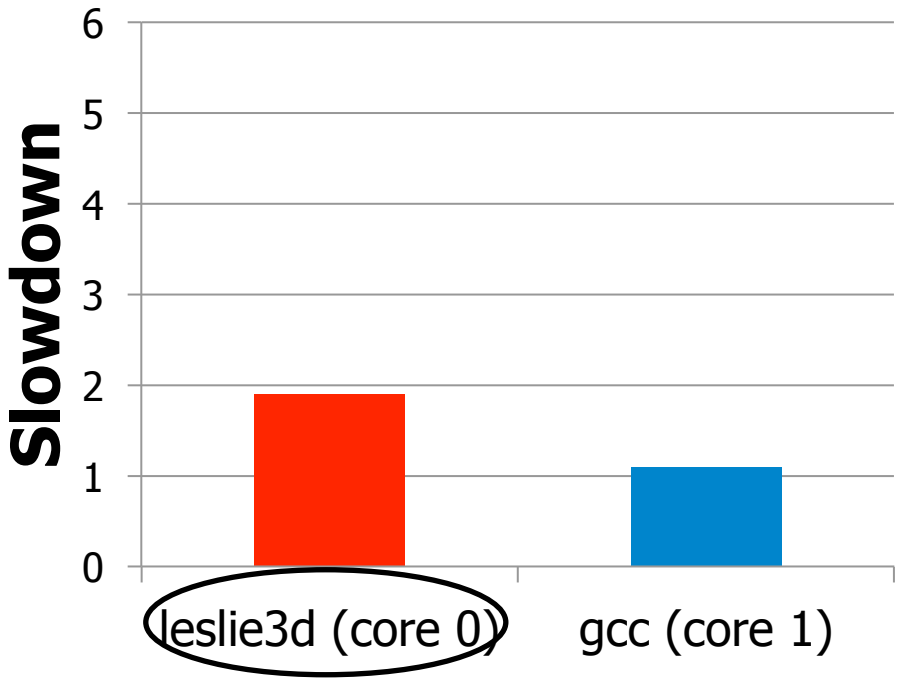
MISE: Providing Performance Predictability in Shared Main Memory Systems

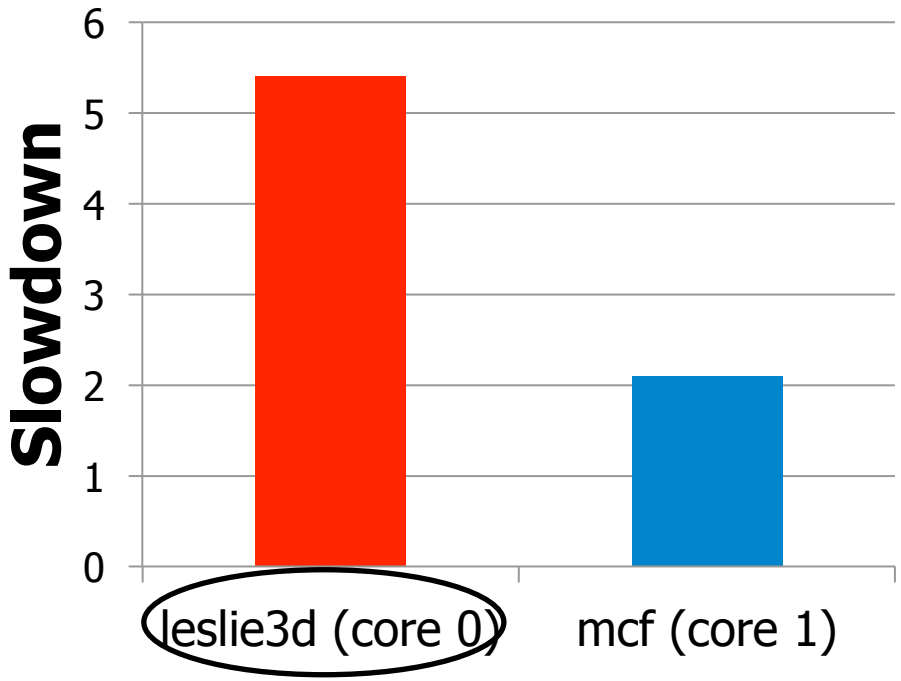
Lavanya Subramanian, Vivek Seshadri, Yoongu Kim, Ben Jaiyen, Onur Mutlu

SAFARI

Carnegie Mellon

Unpredictable Application Slowdowns





An application's performance depends on which application it is running with

Need for Predictable Performance

- There is a need for predictable performance
  - When multiple applications share resources
  - Especially if some applications require performance guarantees

Our Goal: Predictable performance in the presence of memory interference

- Example 2: In server systems
  - Different users' jobs consolidated onto the same server
  - Need to provide bounded slowdowns to critical jobs

Outline

1. Estimate Slowdown

- Key Observations
- Implementation
- MISE Model: Putting it All Together
- Evaluating the Model

2. Control Slowdown

- Providing Soft Slowdown Guarantees
- Minimizing Maximum Slowdown

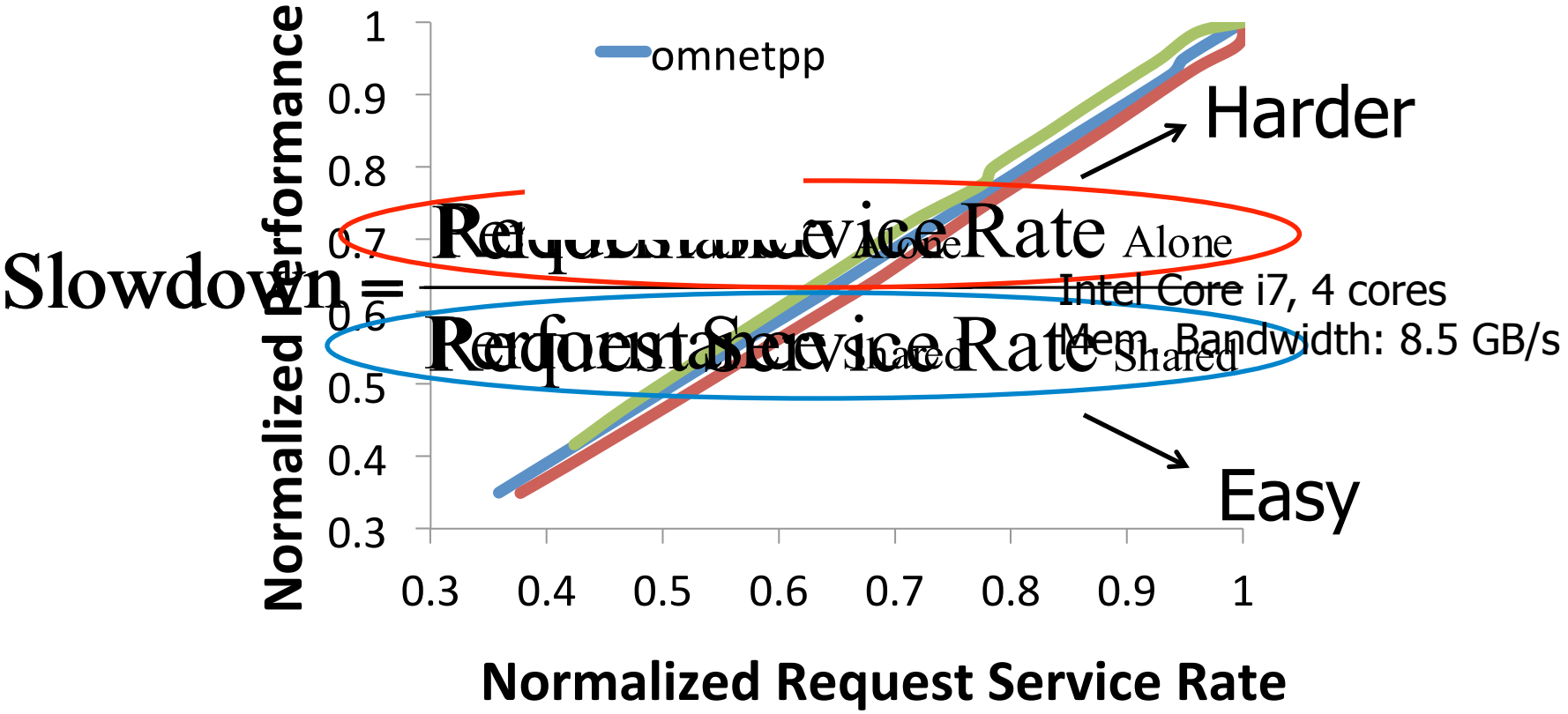
Slowdown: Definition

$$\text{Slowdown} = \frac{\text{Performance}_{\text{Alone}}}{\text{Performance}_{\text{Shared}}}$$

Key Observation 1

For a memory bound application,

Performance ∝ Memory request service rate



Key Observation 2

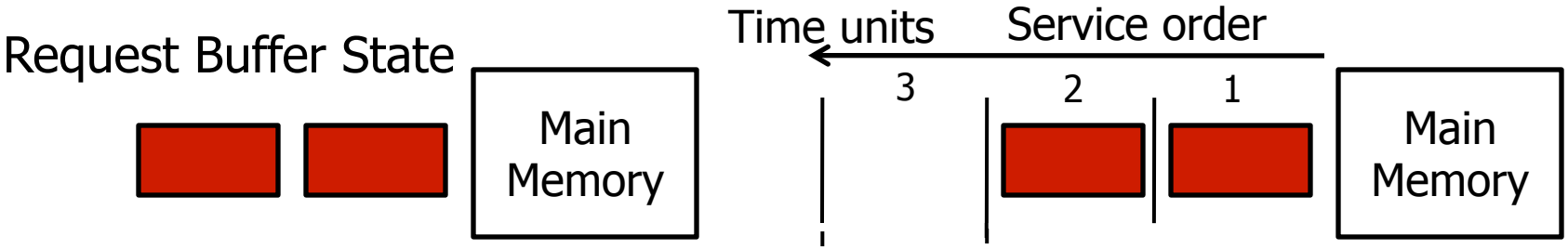
Request Service Rate $_{Alone}$ (RSR_{Alone}) of an application can be estimated by giving the application highest priority in accessing memory

Highest priority → Little interference

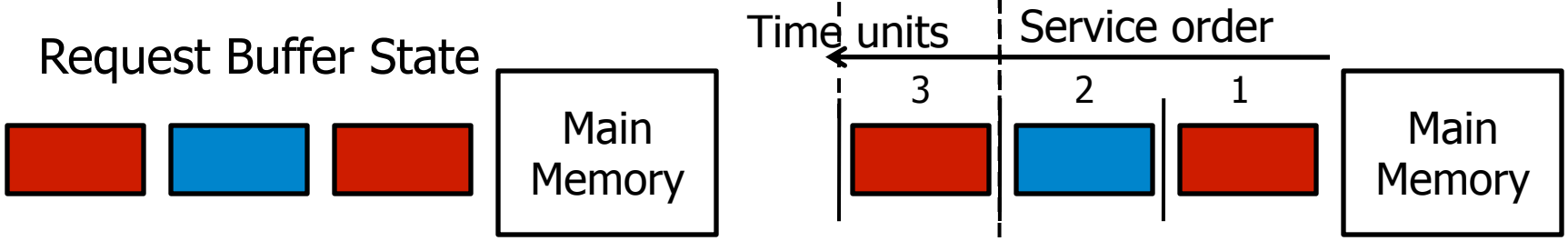
(almost as if the application were run alone)

Key Observation 2

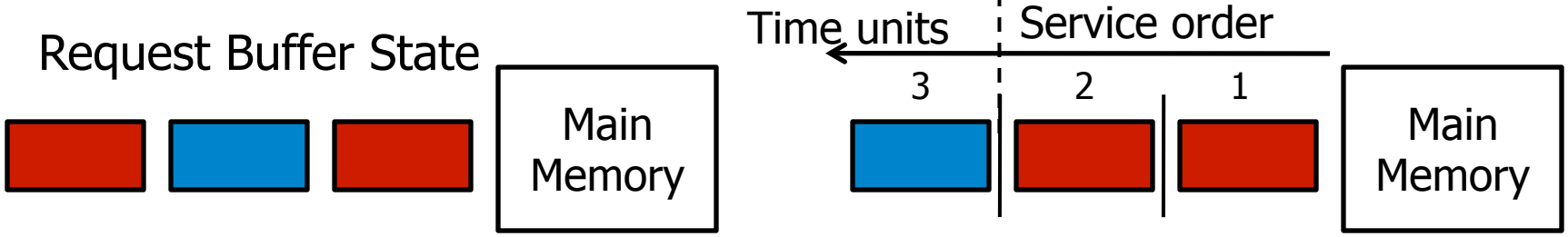
1. Run alone



2. Run with another application



3. Run with another application: highest priority

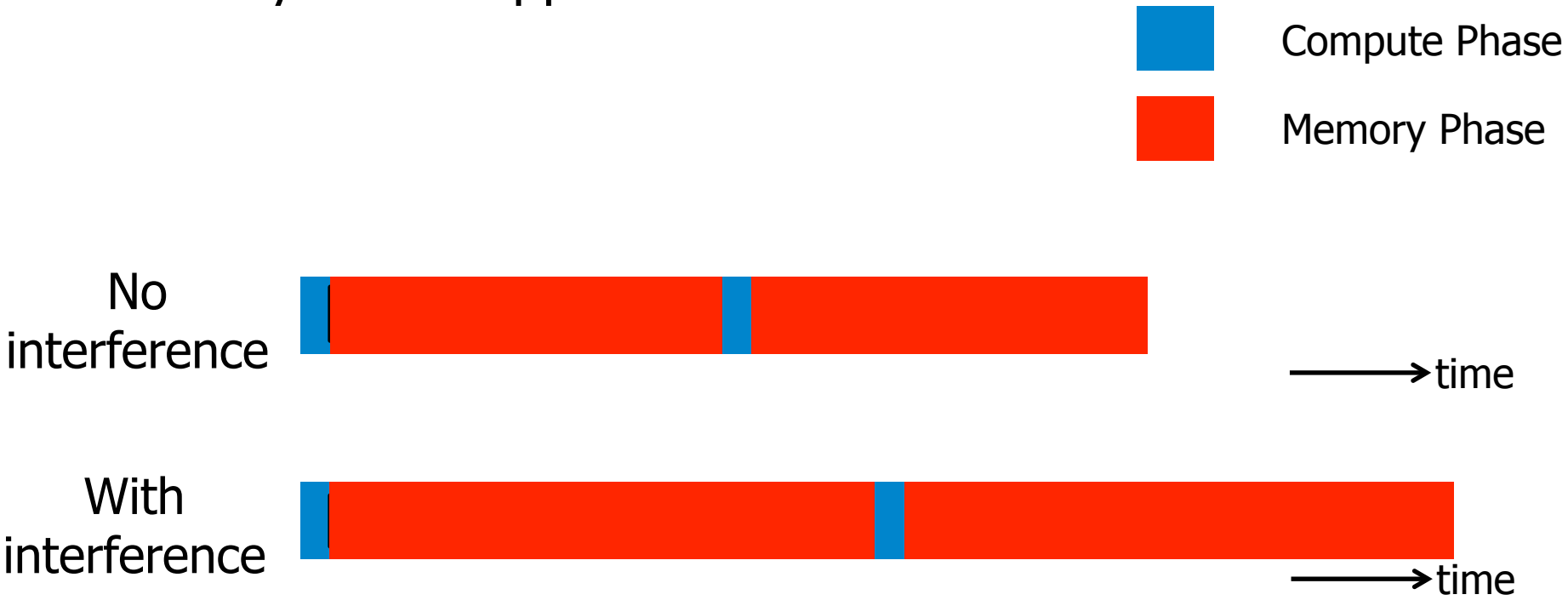


Memory Interference-induced Slowdown Estimation (MISE) model for memory bound applications

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

Key Observation 3

- Memory-bound application



Memory phase slowdown dominates overall slowdown

Key Observation 3

Memory Interference-induced Slowdown Estimation (MISE) model for non-memory bound applications

$$\text{Slowdown} = (1 - \alpha) + \alpha \frac{\text{RSR}_{\text{Alone}}}{\text{RSR}_{\text{Shared}}}$$

Measuring RSR_{Shared} and α

- Request Service Rate $_{Shared}$ (RSR_{Shared})
  - Per-core counter to track number of requests serviced
  - At the end of each interval, measure

$$RSR_{Shared} = \frac{\text{Number of Requests Serviced}}{\text{Interval Length}}$$

- Memory Phase Fraction (α)
  - Count number of stall cycles at the core
  - Compute fraction of cycles stalled for memory

Estimating Request Service Rate $_{Alone}$ (RSR_{Alone})

- Divide each interval into shorter epochs
- At the beginning of each epoch
  - Memory controller randomly picks an application as the highest priority application

Goal: Estimate RSR_{Alone}

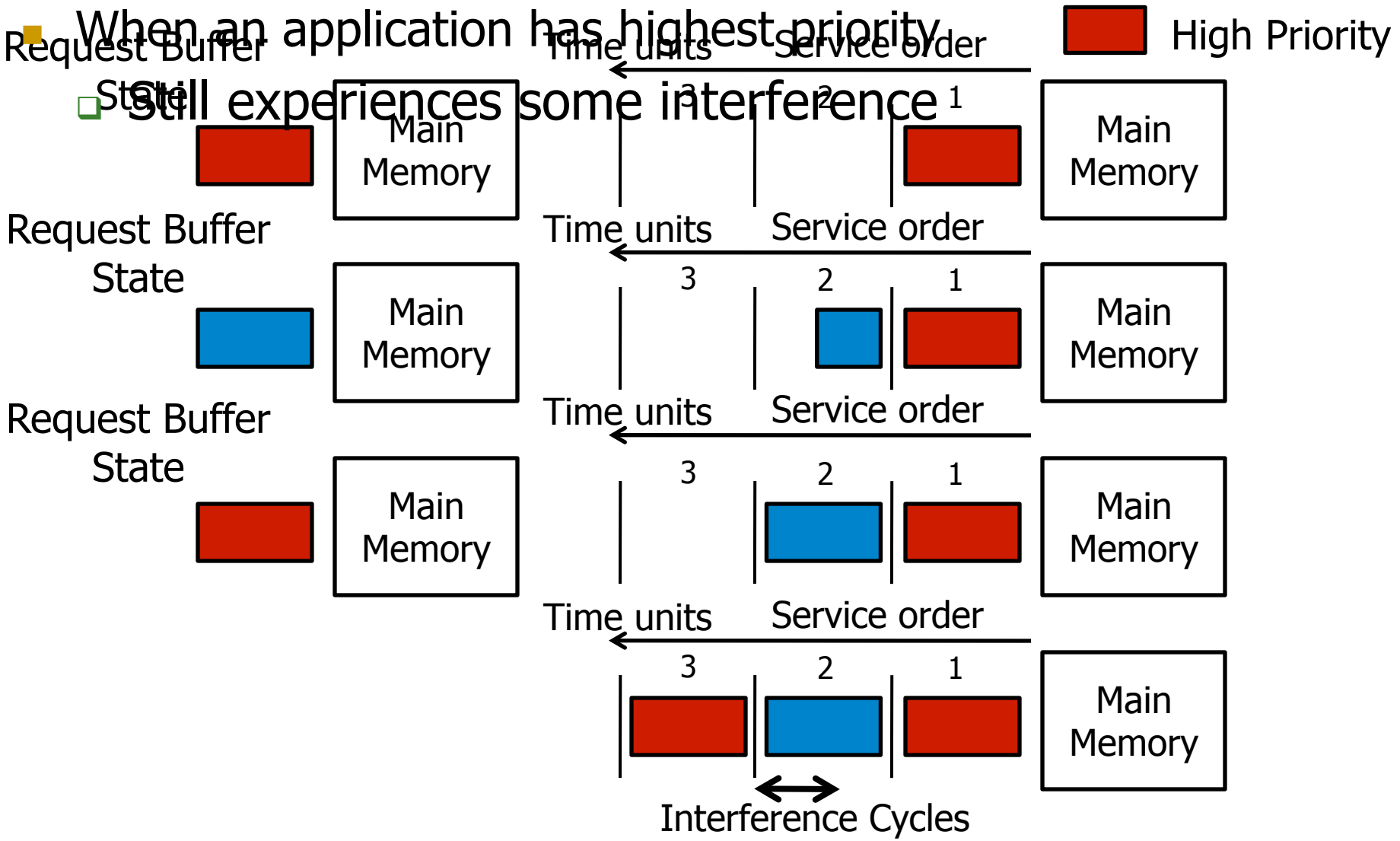
How: Periodically give each application highest priority in accessing memory

- At the end of an interval, for each application, estimate

$$RSR_{Alone} = \frac{\text{Number of Requests During High Priority Epochs}}{\text{Number of Cycles Application Given High Priority}}$$

Inaccuracy in Estimating RSR_{Alone}

- When an application has highest priority
  - Still experiences some interference



Accounting for Interference in RSR_{Alone} Estimation

- Solution: Determine and remove interference cycles from RSR_{Alone} calculation

$$RSR_{Alone} = \frac{\text{Number of Requests During High Priority Epochs}}{\text{Number of Cycles Application Given High Priority} - \text{Interference Cycles}}$$

- A cycle is an interference cycle if
  - a request from the highest priority application is waiting in the request buffer *and*
  - another application's request was issued previously

Outline

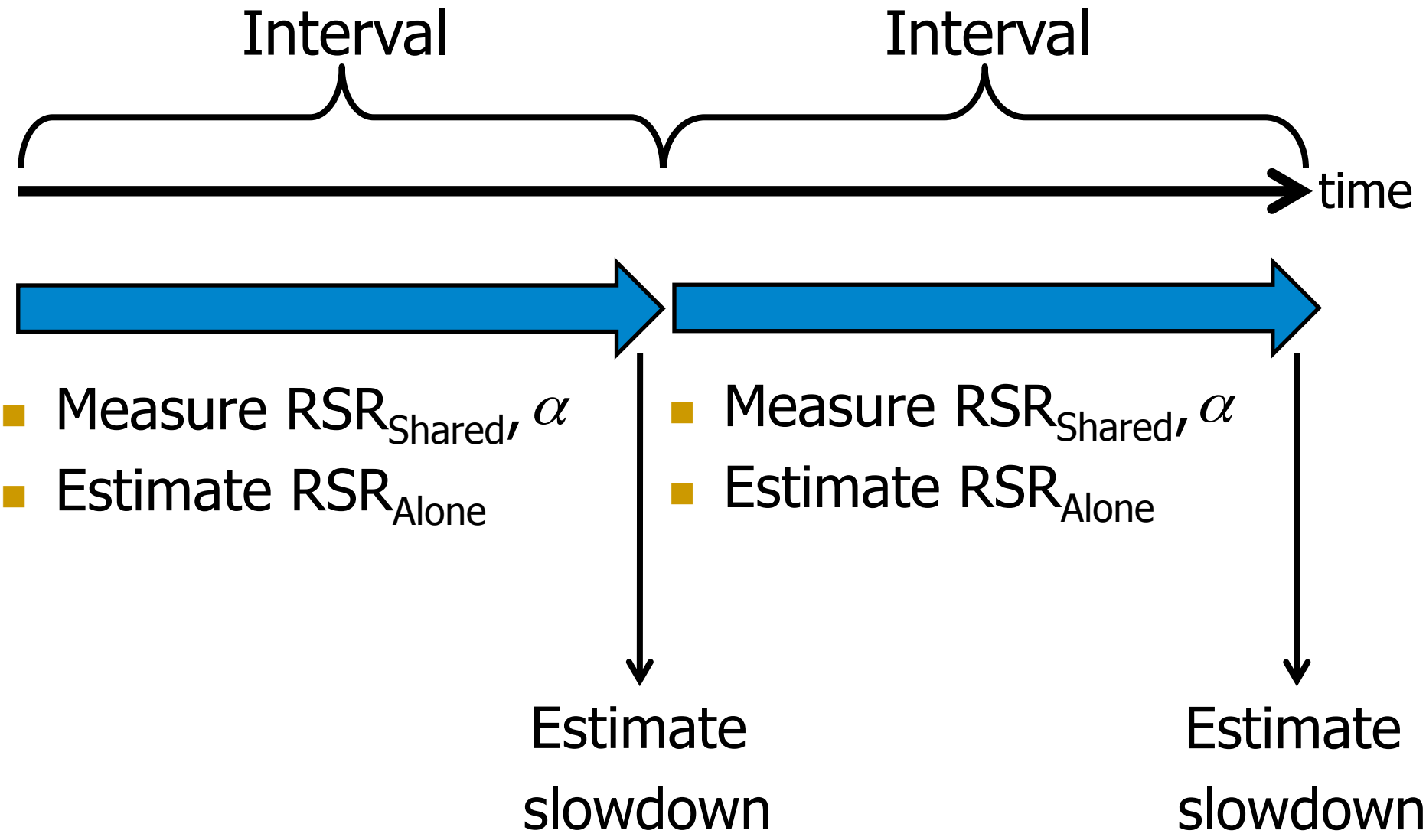
1. Estimate Slowdown

- Key Observations
- Implementation
- MISE Model: Putting it All Together
- Evaluating the Model

2. Control Slowdown

- Providing Soft Slowdown Guarantees
- Minimizing Maximum Slowdown

MISE Model: Putting it All Together



Previous Work on Slowdown Estimation

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

- Basic Idea:

$$\text{Slowdown} = \frac{\text{Stall Time Alone}}{\text{Stall Time Shared}}$$

Hard

Easy

Count number of cycles application receives interference

Two Major Advantages of MISE Over STFM

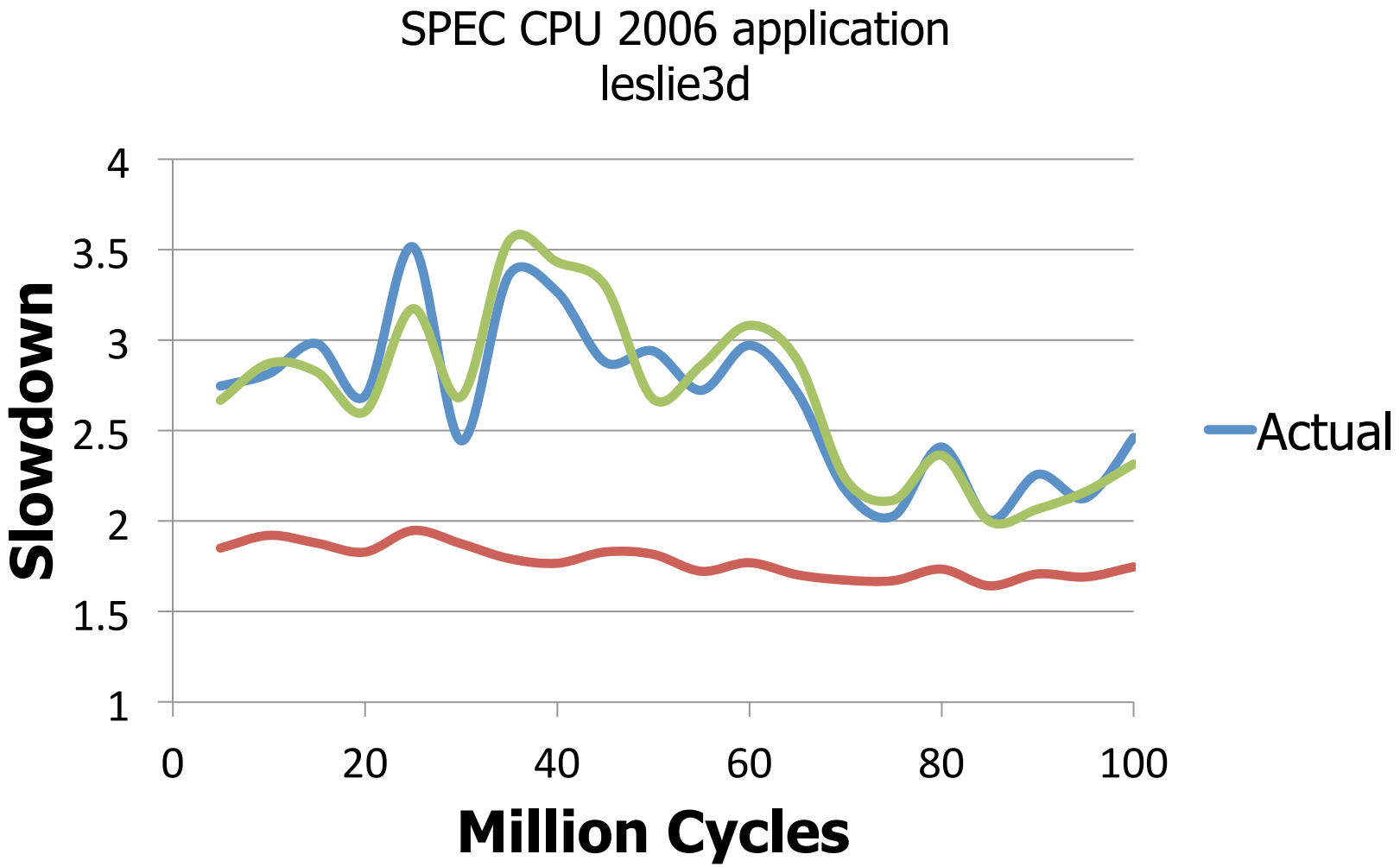
- Advantage 1:
  - STFM estimates alone performance while an application is receiving interference → Hard
  - MISE estimates alone performance while giving an application the highest priority → Easier
- Advantage 2:
  - STFM does not take into account compute phase for non-memory-bound applications
  - MISE accounts for compute phase → Better accuracy

Methodology

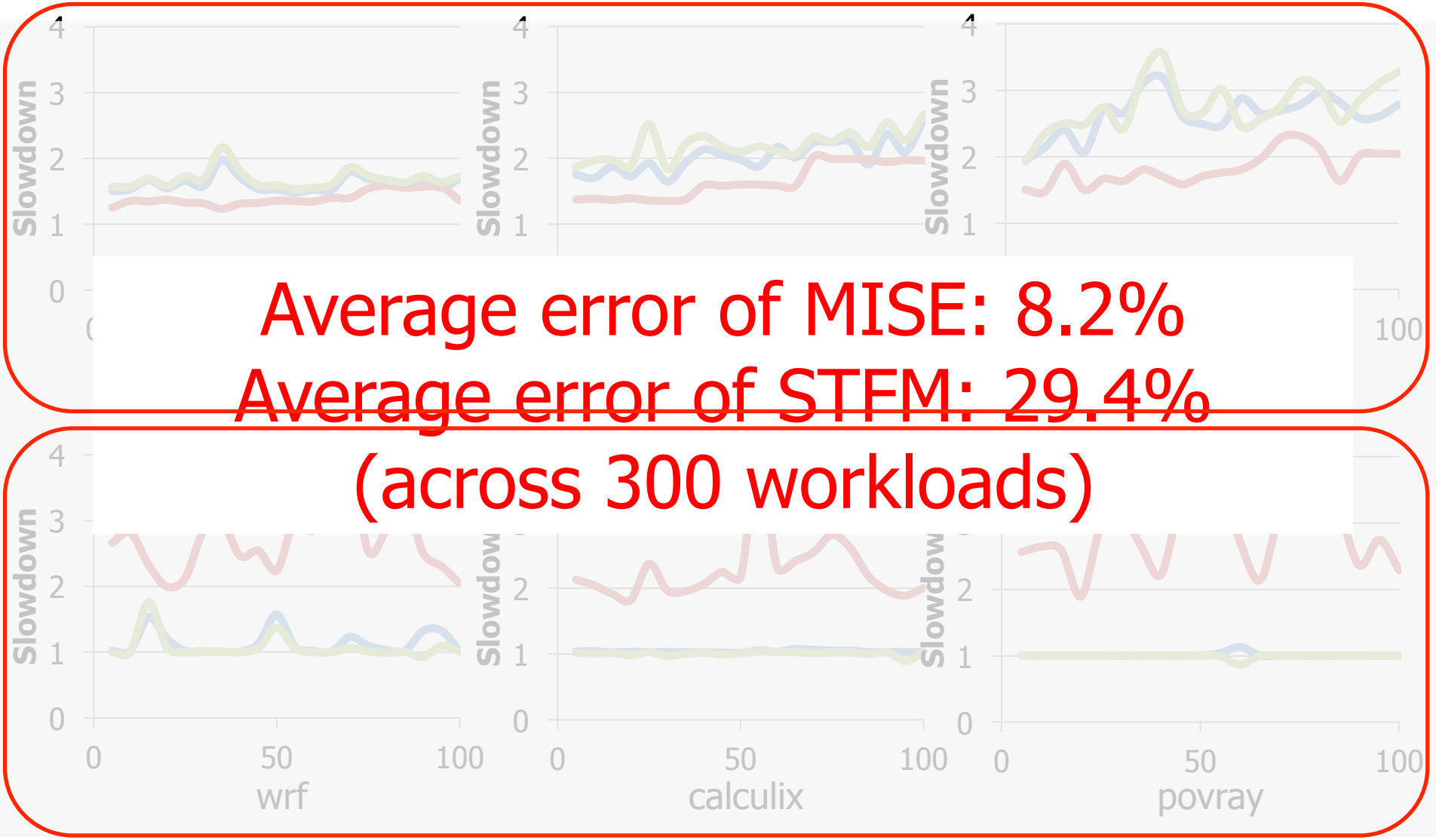
- Configuration of our simulated system
  - 4 cores
  - 1 channel, 8 banks/channel
  - DDR3 1066 DRAM
  - 512 KB private cache/core

- Workloads
  - SPEC CPU2006
  - 300 multi programmed workloads

Quantitative Comparison



Comparison to STFM



Average error of MISE: 8.2%
Average error of STFM: 29.4%
(across 300 workloads)

Providing "Soft" Slowdown Guarantees

- Goal
  1. Ensure QoS-critical applications meet a prescribed slowdown bound
  2. Maximize system performance for other applications

- Basic Idea
  - Allocate just enough bandwidth to QoS-critical application
  - Assign remaining bandwidth to other applications

MISE-QoS: Mechanism to Provide Soft QoS

- Assign an initial bandwidth allocation to QoS-critical application
- Estimate slowdown of QoS-critical application using the MISE model
- After every N intervals
  - If slowdown > bound B +/- ε, increase bandwidth allocation
  - If slowdown < bound B +/- ε, decrease bandwidth allocation
- When slowdown bound not met for N intervals
  - Notify the OS so it can migrate/de-schedule jobs

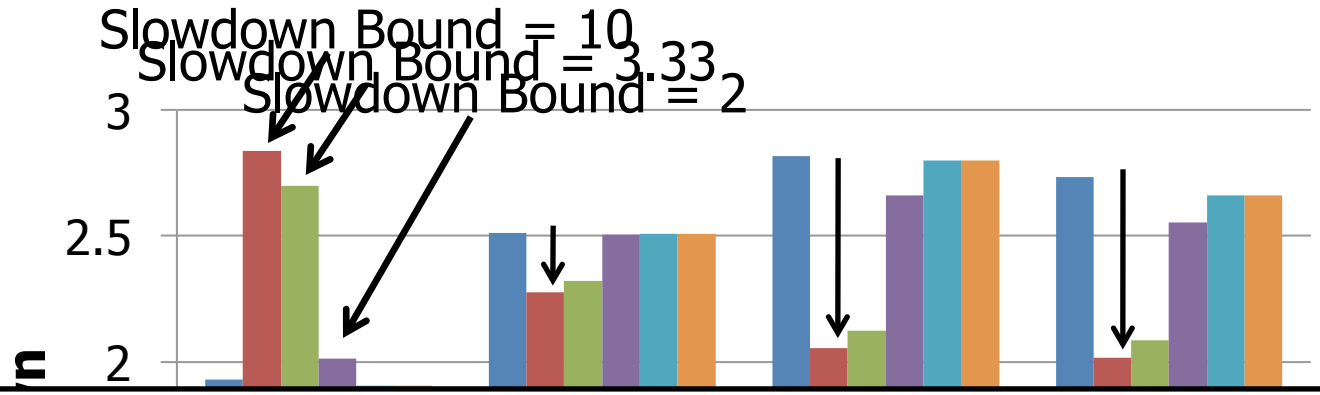
Methodology

- Each application (25 applications in total) considered the QoS-critical application
- Run with 12 sets of co-runners of different memory intensities
- Total of 300 multiprogrammed workloads
- Each workload run with 10 slowdown bound values
- Baseline memory scheduling mechanism
  - Always prioritize QoS-critical application

    [Iyer+, SIGMETRICS 2007]
  - Other applications' requests scheduled in FRFCFS order

    [Zuravleff +, US Patent 1997, Rixner+, ISCA 2000]

A Look at One Workload



MISE is effective in

1. meeting the slowdown bound for the QoS-critical application
2. improving performance of non-QoS-critical applications

leslie3d hmmer lbm omnetpp

QoS-critical **non-QoS-critical**

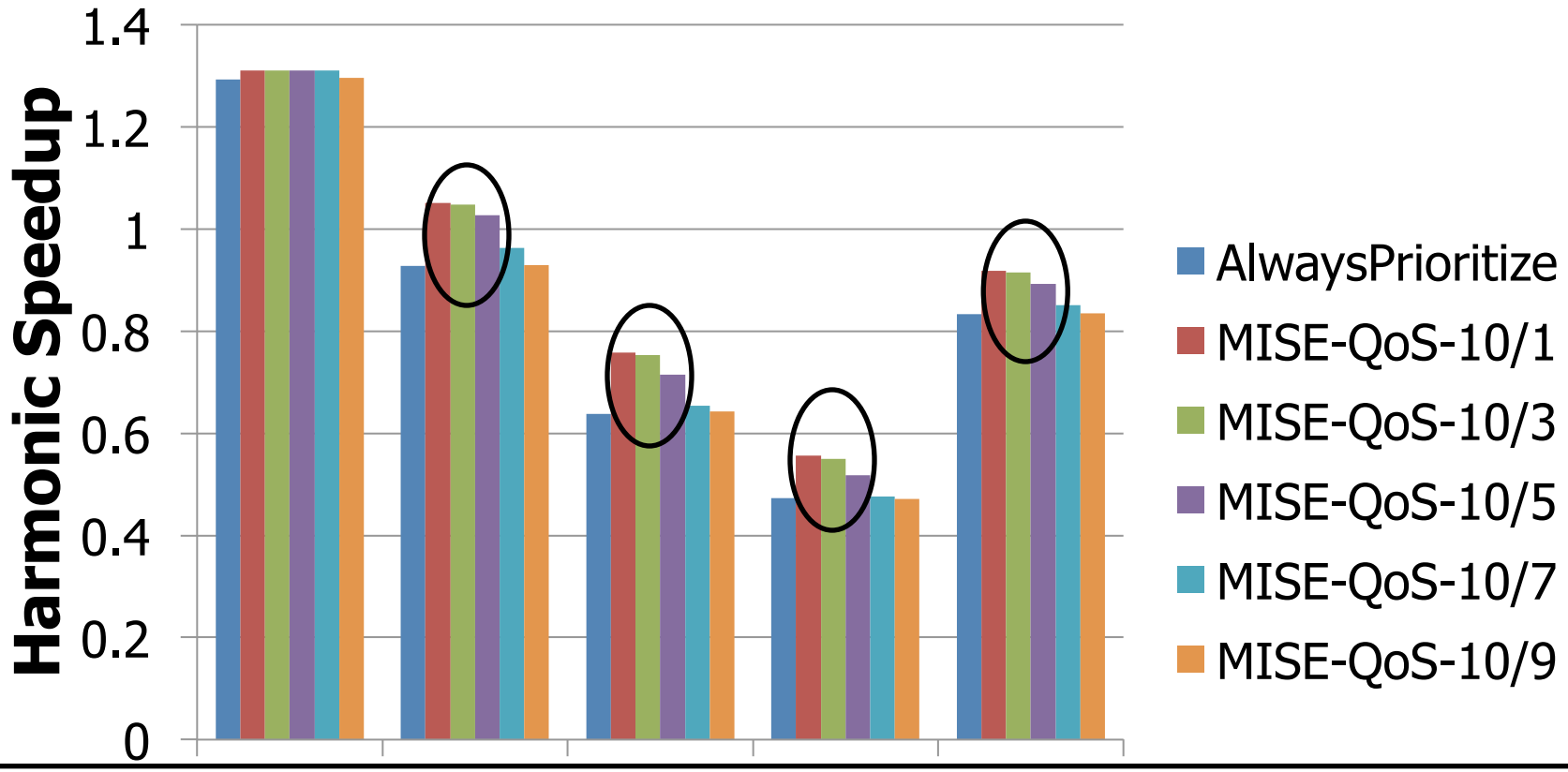
Effectiveness of MISE in Enforcing QoS

Across 3000 data points

	Predicted Met	Predicted Not Met
QoS Bound Met	78.8%	2.1%
QoS Bound Not Met	2.2%	16.9%

MISE-QoS correctly predicts whether or not the bound is met for 95.7% of workloads

Performance of Non-QoS-Critical Applications



When slowdown bound is 10/3
MISE-QoS improves system performance by 10%

Other Results in the Paper

- Sensitivity to model parameters
  - Robust across different values of model parameters
- Comparison of STFM and MISE models in enforcing soft slowdown guarantees
  - MISE significantly more effective in enforcing guarantees
- Minimizing maximum slowdown
  - MISE improves fairness across several system configurations

Summary

- Uncontrolled memory interference slows down applications unpredictably
- Goal: Estimate and control slowdowns
- Key contribution
  - MISE: An accurate slowdown estimation model
  - Average error of MISE: 8.2%
- Key Idea
  - Request Service Rate is a proxy for performance
  - Request Service Rate $_{Alone}$ estimated by giving an application highest priority in accessing memory
- Leverage slowdown estimates to control slowdowns
  - Providing soft slowdown guarantees
  - Minimizing maximum slowdown

MISE: Providing Performance Predictability in Shared Main Memory Systems

Lavanya Subramanian, Vivek Seshadri, Yoongu Kim, Ben Jaiyen, Onur Mutlu

SAFARI

Carnegie Mellon

Memory Scheduling for Parallel Applications

Eiman Ebrahimi, Rustam Miftakhutdinov, Chris Fallin,
Chang Joo Lee, Onur Mutlu, and Yale N. Patt,
"Parallel Application Memory Scheduling"
*Proceedings of the 44th International Symposium on Microarchitecture (**MICRO**)*,
Porto Alegre, Brazil, December 2011. Slides (pptx)

Handling Interference in Parallel Applications

- Threads in a multithreaded application are inter-dependent
- Some threads can be on the critical path of execution due to synchronization; some threads are not
- How do we schedule requests of inter-dependent threads to maximize multithreaded application performance?

- Idea: Estimate limiter threads likely to be on the critical path and prioritize their requests; shuffle priorities of non-limiter threads to reduce memory interference among them [Ebrahimi+, MICRO'11]

- Hardware/software cooperative limiter thread estimation:
  - Thread executing the most contended critical section
  - Thread that is falling behind the most in a *parallel for* loop

Aside: Self-Optimizing Memory Controllers

Engin Ipek, Onur Mutlu, José F. Martínez, and Rich Caruana,
"Self Optimizing Memory Controllers: A Reinforcement Learning Approach"
Proceedings of the 35th International Symposium on Computer Architecture (***ISCA***),
pages 39-50, Beijing, China, June 2008. Slides (pptx)

Why are DRAM Controllers Difficult to Design?

- Need to obey DRAM timing constraints for correctness
  - There are many (50+) timing constraints in DRAM
  - tWTR: Minimum number of cycles to wait before issuing a read command after a write command is issued
  - tRC: Minimum number of cycles between the issuing of two consecutive activate commands to the same bank
  - …
- Need to keep track of many resources to prevent conflicts
  - Channels, banks, ranks, data bus, address bus, row buffers
- Need to handle DRAM refresh
- Need to optimize for performance (in the presence of constraints)
  - Reordering is not simple
  - Predicting the future?

Many DRAM Timing Constraints

Latency	Symbol	DRAM cycles	Latency	Symbol	DRAM cycles
Precharge	${}^tRP$	11	Activate to read/write	${}^tRCD$	11
Read column address strobe	CL	11	Write column address strobe	CWL	8
Additive	AL	0	Activate to activate	${}^tRC$	39
Activate to precharge	${}^tRAS$	28	Read to precharge	${}^tRTP$	6
Burst length	${}^tBL$	4	Column address strobe to column address strobe	${}^tCCD$	4
Activate to activate (different bank)	${}^tRRD$	6	Four activate windows	${}^tFAW$	24
Write to read	${}^tWTR$	6	Write recovery	${}^tWR$	12

Table 4. DDR3 1600 DRAM timing specifications

- From Lee et al., "DRAM-Aware Last-Level Cache Writeback: Reducing Write-Caused Interference in Memory Systems," HPS Technical Report, April 2010.

More on DRAM Operation and Constraints

- Kim et al., "A Case for Exploiting Subarray-Level Parallelism (SALP) in DRAM," ISCA 2012.
- Lee et al., "Tiered-Latency DRAM: A Low Latency and Low Cost DRAM Architecture," HPCA 2013.

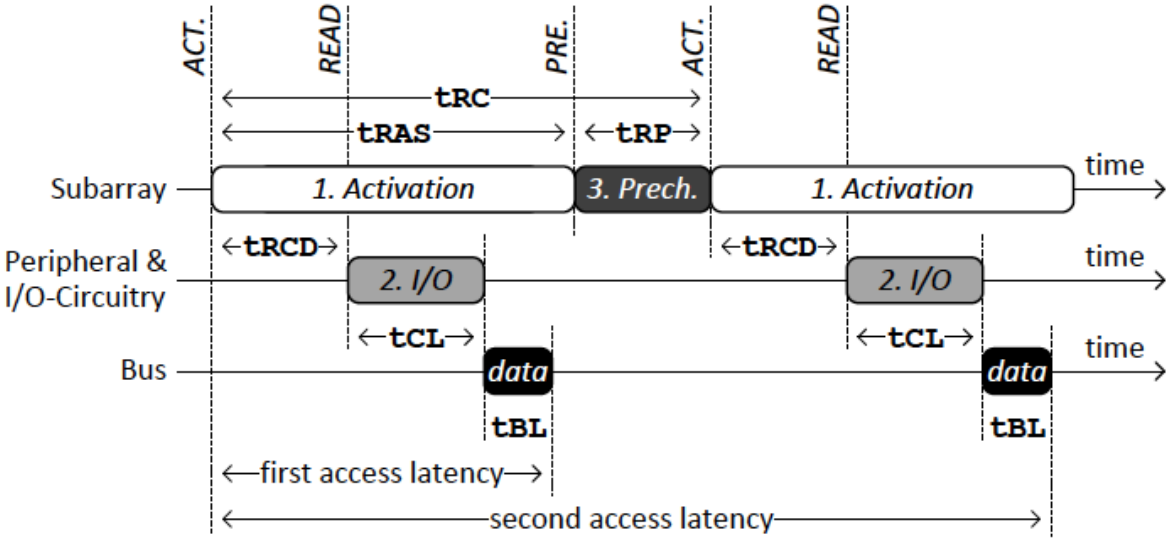


Figure 5. Three Phases of DRAM Access

Table 2. Timing Constraints (DDR3-1066) [43]

Phase	Commands	Name	Value
1	ACT → READ ACT → WRITE	tRCD	15ns
	ACT → PRE	tRAS	37.5ns
2	READ → *data* WRITE → *data*	tCL tCWL	15ns 11.25ns
	data burst	tBL	7.5ns
3	PRE → ACT	tRP	15ns
1 & 3	ACT → ACT	tRC (tRAS+tRP)	52.5ns

Self-Optimizing DRAM Controllers

- Problem: DRAM controllers difficult to design → It is difficult for human designers to design a policy that can adapt itself very well to different workloads and different system conditions

- Idea: Design a memory controller that adapts its scheduling policy decisions to workload behavior and system conditions using machine learning.

- Observation: Reinforcement learning maps nicely to memory control.

- Design: Memory controller is a reinforcement learning agent that dynamically and continuously learns and employs the best scheduling policy.

Self-Optimizing DRAM Controllers

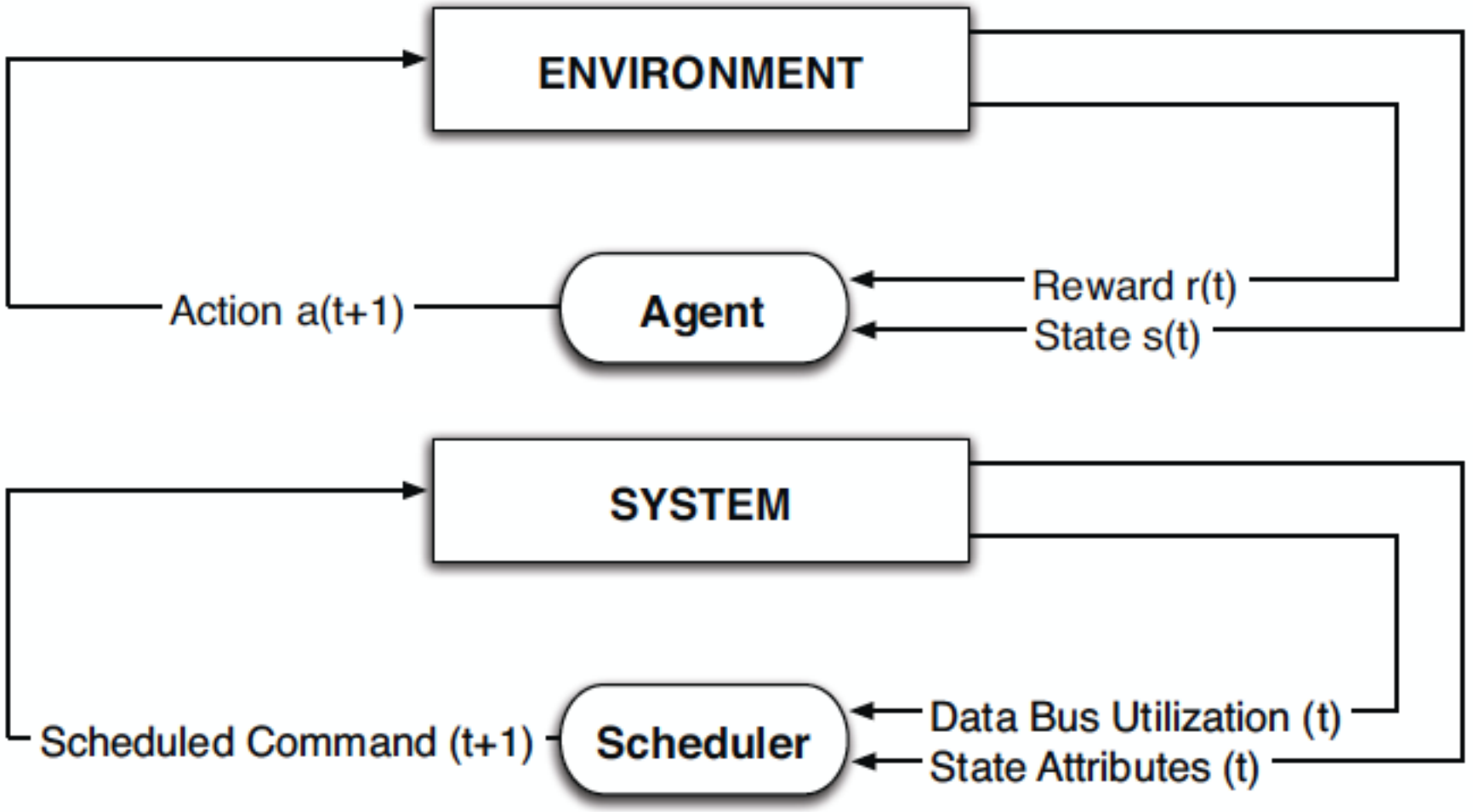


Figure 2: (a) Intelligent agent based on reinforcement learning principles; (b) DRAM scheduler as an RL-agent

Self-Optimizing DRAM Controllers

- Engin Ipek, Onur Mutlu, José F. Martínez, and Rich Caruana, **"Self Optimizing Memory Controllers: A Reinforcement Learning Approach"**
  *Proceedings of the 35th International Symposium on Computer Architecture (**ISCA**)*, pages 39-50, Beijing, China, June 2008.

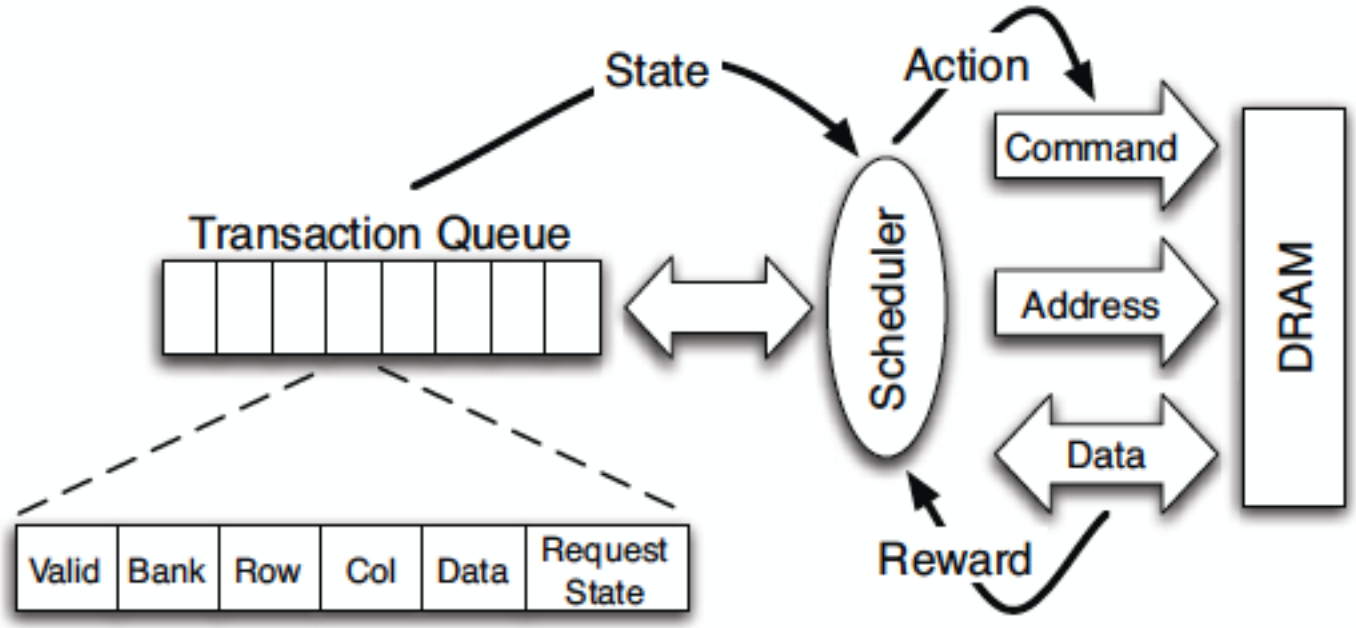


Figure 4: High-level overview of an RL-based scheduler.

Performance Results

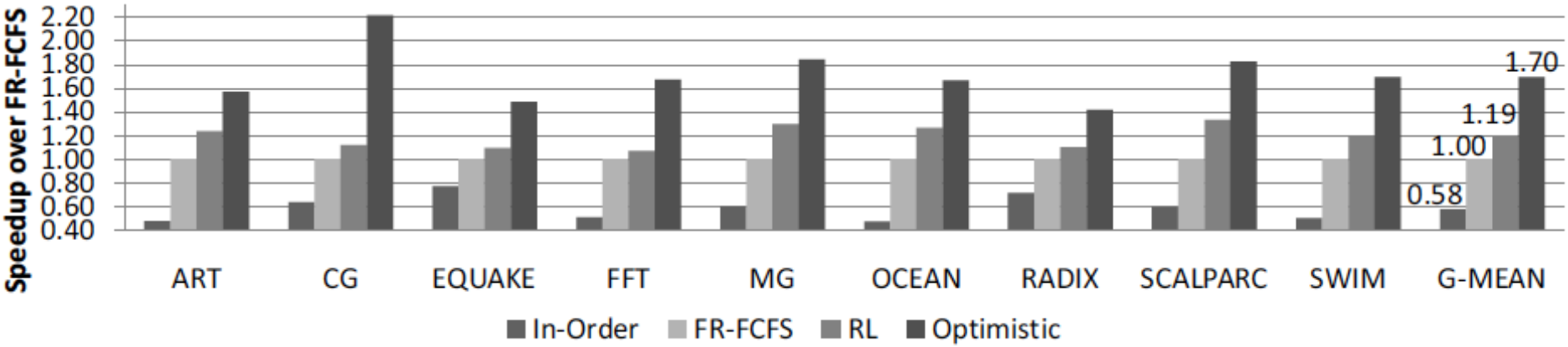


Figure 7: Performance comparison of in-order, FR-FCFS, RL-based, and optimistic memory controllers

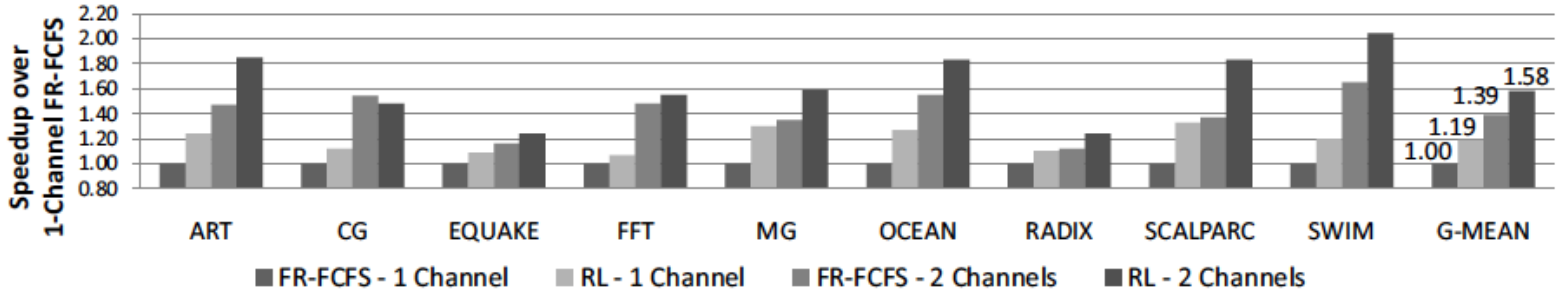


Figure 15: Performance comparison of FR-FCFS and RL-based memory controllers on systems with 6.4GB/s and 12.8GB/s peak DRAM bandwidth

QoS-Aware Memory Systems: The Dumb Resources Approach

Designing QoS-Aware Memory Systems: Approaches

- Smart resources: Design each shared resource to have a configurable interference control/reduction mechanism
  - QoS-aware memory controllers [Mutlu+ MICRO'07] [Moscibroda+, Usenix Security'07] [Mutlu+ ISCA'08, Top Picks'09] [Kim+ HPCA'10] [Kim+ MICRO'10, Top Picks'11] [Ebrahimi+ ISCA'11, MICRO'11] [Ausavarungnirun+, ISCA'12] [Subramanian+, HPCA'13]
  - QoS-aware interconnects [Das+ MICRO'09, ISCA'10, Top Picks '11] [Grot+ MICRO'09, ISCA'11, Top Picks '12]
  - QoS-aware caches
- Dumb resources: Keep each resource free-for-all, but reduce/control interference by injection control or data mapping
  - Source throttling to control access to memory system [Ebrahimi+ ASPLOS'10, ISCA'11, TOCS'12] [Ebrahimi+ MICRO'09] [Nychis+ HotNets'10]
  - QoS-aware data mapping to memory controllers [Muralidhara+ MICRO'11]
  - QoS-aware thread scheduling to cores [Das+ HPCA'13]

Fairness via Source Throttling

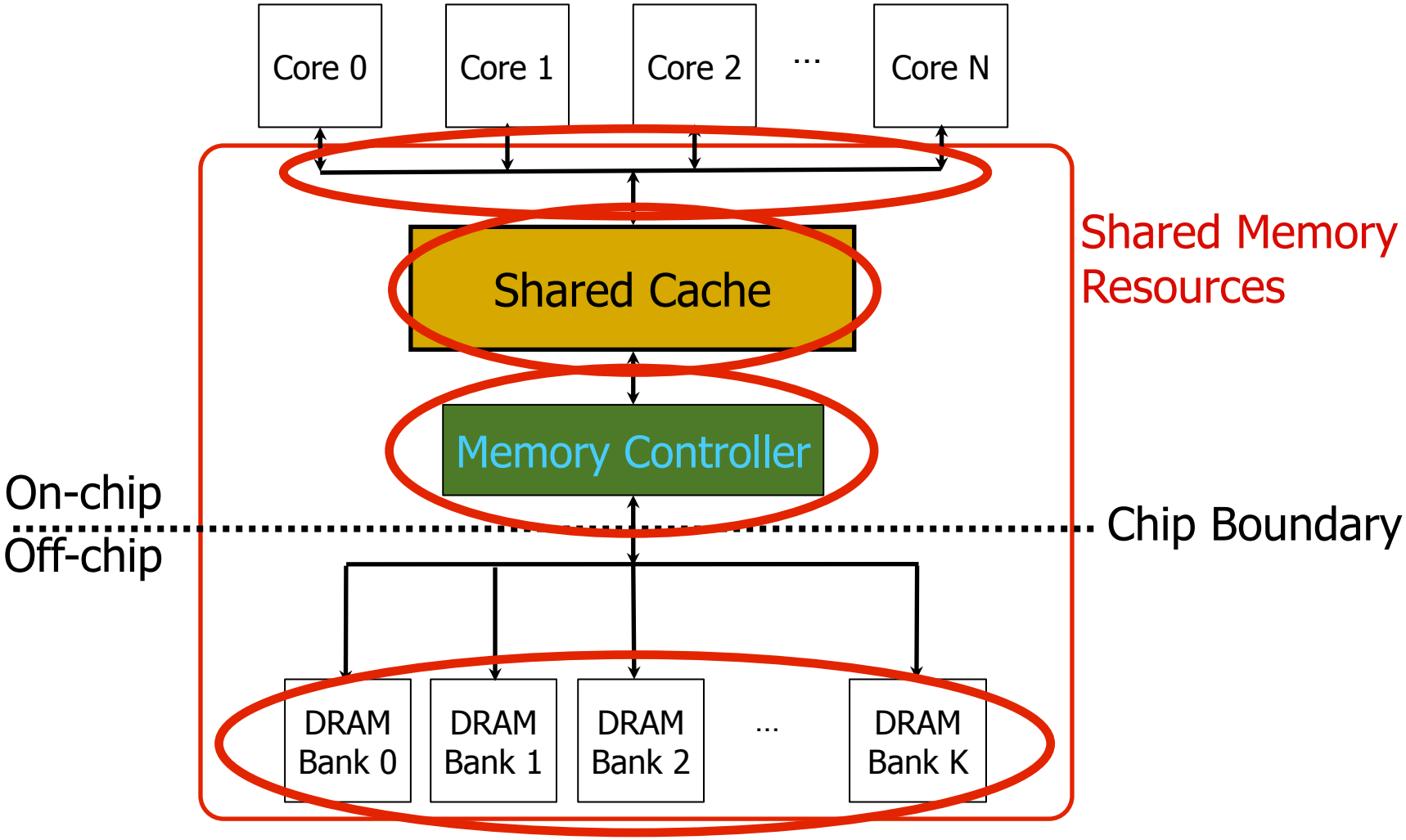
Eiman Ebrahimi, Chang Joo Lee, Onur Mutlu, and Yale N. Patt,

"Fairness via Source Throttling: A Configurable and High-Performance Fairness Substrate for Multi-Core Memory Systems"

15th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (***ASPLOS***), pages 335-346, Pittsburgh, PA, March 2010. Slides (pdf)

FST ASPLOS 2010 Talk

Many Shared Resources



The Problem with "Smart Resources"

- Independent interference control mechanisms in caches, interconnect, and memory can contradict each other

- Explicitly coordinating mechanisms for different resources requires complex implementation

- How do we enable fair sharing of the entire memory system by controlling interference in a coordinated manner?

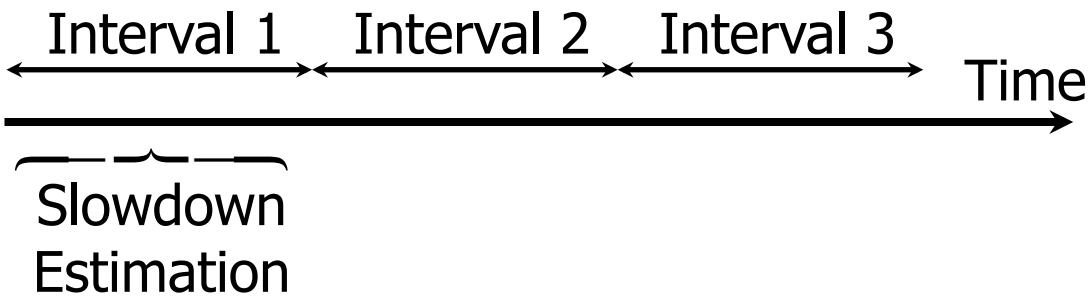
An Alternative Approach: Source Throttling

- Manage inter-thread interference at the cores, not at the shared resources

- Dynamically estimate unfairness in the memory system
- Feed back this information into a controller
- Throttle cores' memory access rates accordingly
  - Whom to throttle and by how much depends on performance target (throughput, fairness, per-thread QoS, etc)
  - E.g., if unfairness > system-software-specified target then throttle down core causing unfairness & throttle up core that was unfairly treated

- Ebrahimi et al., "Fairness via Source Throttling," ASPLOS'10, TOCS'12.

Fairness via Source Throttling (FST) [ASPLOS'10]



FST

1- Estimating system unfairness
2- Find app. with the highest slowdown (App-slowest)
3- Find app. causing most interference for App-slowest (App-interfering)

```
if (Unfairness Estimate >Target)
{
 1-Throttle down App-interfering
    (limit injection rate and parallelism)
 2-Throttle up App-slowest
}
```

System Software Support

- Different fairness objectives can be configured by system software
  - Keep maximum slowdown in check
    - Estimated Max Slowdown < Target Max Slowdown
  - Keep slowdown of particular applications in check to achieve a particular performance target
    - Estimated Slowdown(i) < Target Slowdown(i)

- Support for thread priorities
  - Weighted Slowdown(i) =
    Estimated Slowdown(i) x Weight(i)

Source Throttling Results: Takeaways

- Source throttling alone provides better performance than a combination of "smart" memory scheduling and fair caching
  - Decisions made at the memory scheduler and the cache sometimes contradict each other
- Neither source throttling alone nor "smart resources" alone provides the best performance
- Combined approaches are even more powerful
  - Source throttling and resource-based interference control

Designing QoS-Aware Memory Systems: Approaches

- Smart resources: Design each shared resource to have a configurable interference control/reduction mechanism
  - QoS-aware memory controllers [Mutlu+ MICRO'07] [Moscibroda+, Usenix Security'07] [Mutlu+ ISCA'08, Top Picks'09] [Kim+ HPCA'10] [Kim+ MICRO'10, Top Picks'11] [Ebrahimi+ ISCA'11, MICRO'11] [Ausavarungnirun+, ISCA'12] [Subramanian+, HPCA'13]
  - QoS-aware interconnects [Das+ MICRO'09, ISCA'10, Top Picks '11] [Grot+ MICRO'09, ISCA'11, Top Picks '12]
  - QoS-aware caches
- Dumb resources: Keep each resource free-for-all, but reduce/control interference by injection control or data mapping
  - Source throttling to control access to memory system [Ebrahimi+ ASPLOS'10, ISCA'11, TOCS'12] [Ebrahimi+ MICRO'09] [Nychis+ HotNets'10] [Nychis+ SIGCOMM'12]
  - QoS-aware data mapping to memory controllers [Muralidhara+ MICRO'11]
  - QoS-aware thread scheduling to cores [Das+ HPCA'13]

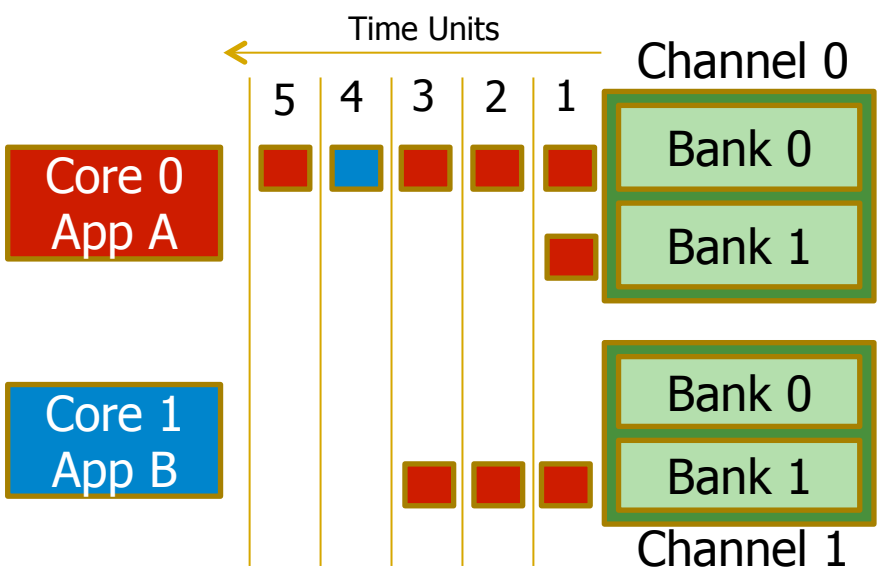
Memory Channel Partitioning

Sai Prashanth Muralidhara, Lavanya Subramanian, Onur Mutlu, Mahmut Kandemir, and Thomas Moscibroda,
"Reducing Memory Interference in Multicore Systems via Application-Aware Memory Channel Partitioning"
44th International Symposium on Microarchitecture (***MICRO***),
Porto Alegre, Brazil, December 2011. Slides (pptx)

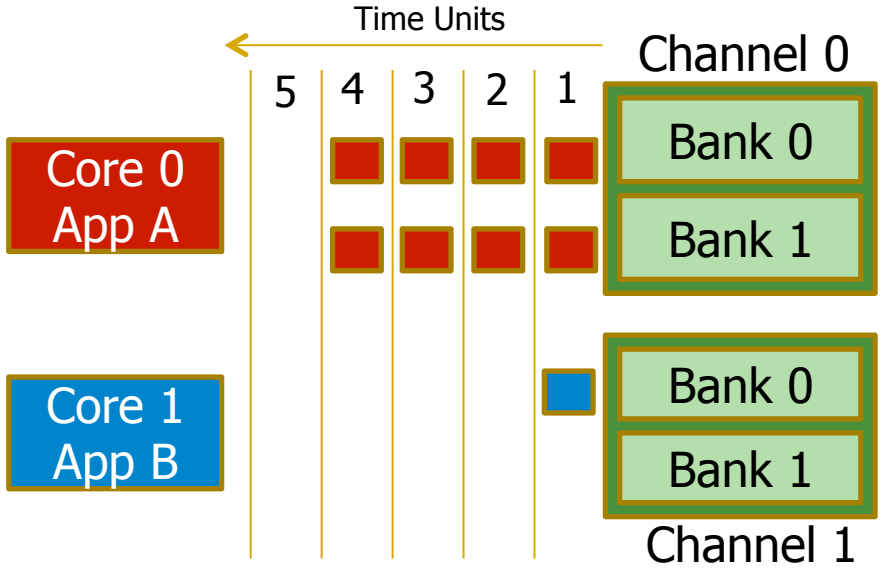
MCP Micro 2011 Talk

Another Way to Reduce Memory Interference

- Memory Channel Partitioning
  - Idea: System software maps badly-interfering applications' pages to different channels [Muralidhara+, MICRO'11]



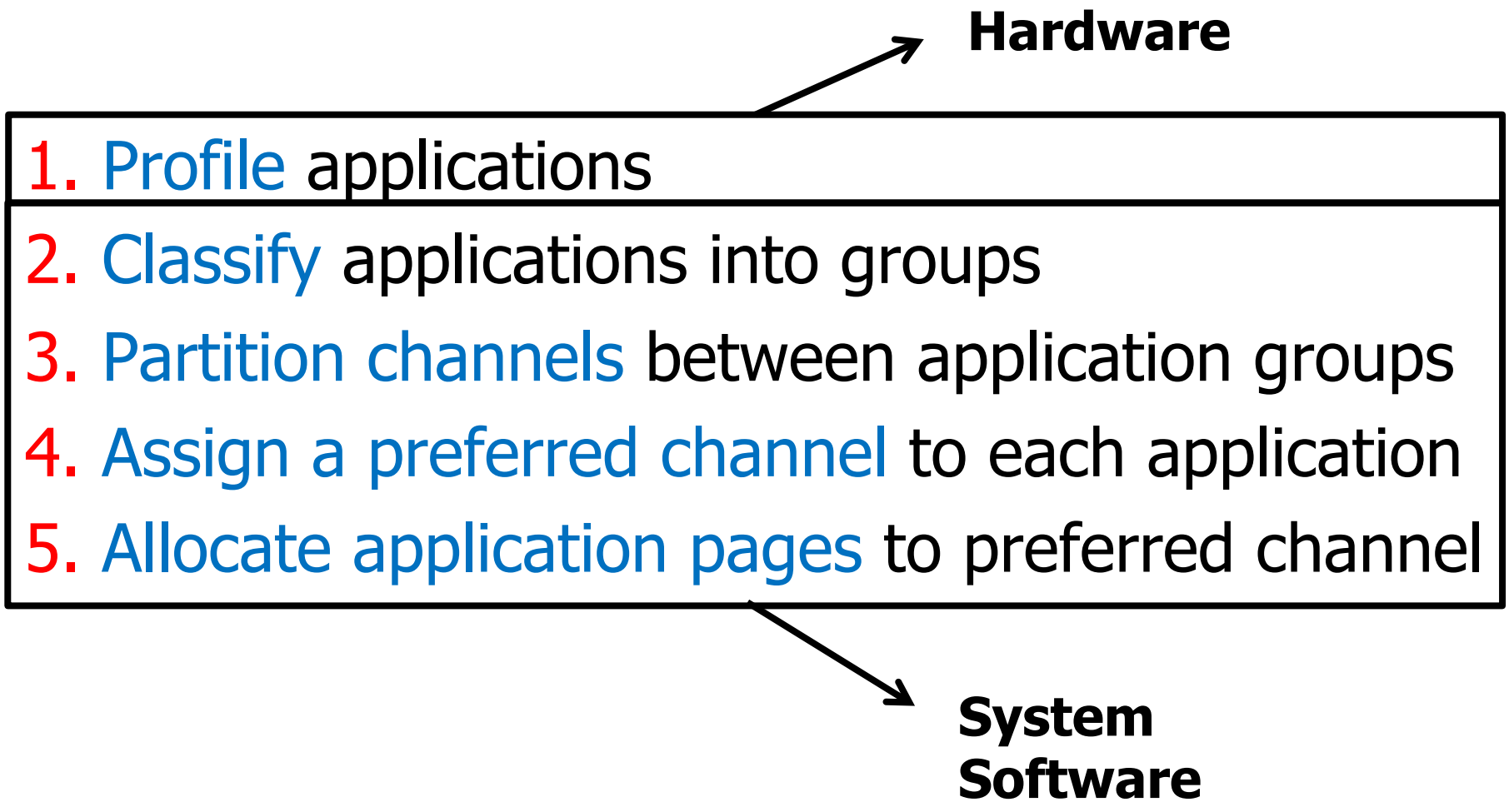
Conventional Page Mapping



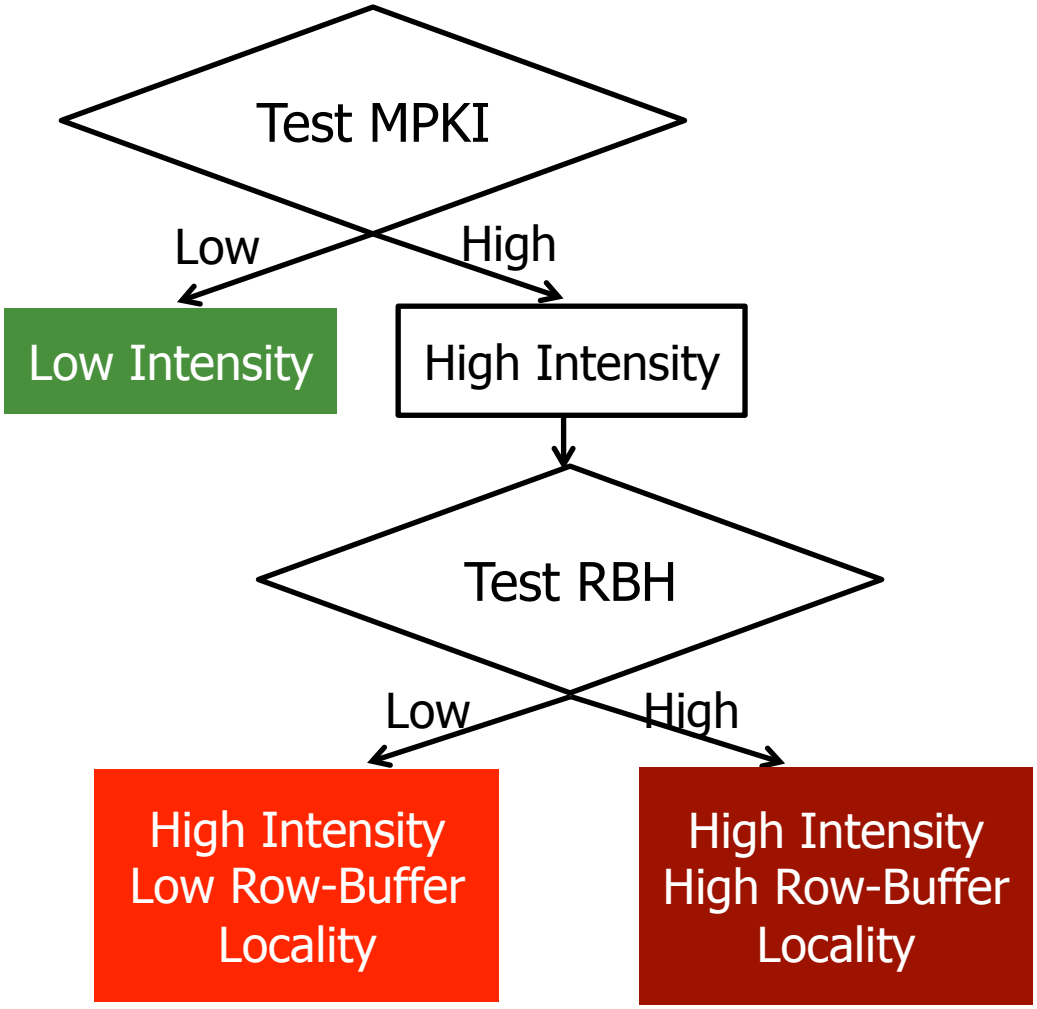
Channel Partitioning

- Separate data of low/high intensity and low/high row-locality applications
- Especially effective in reducing interference of threads with "medium" and "heavy" memory intensity
  - 11% higher performance over existing systems (200 workloads)

Memory Channel Partitioning (MCP) Mechanism



2. Classify Applications



Summary: Memory QoS

- Technology, application, architecture trends dictate new needs from memory system

- A fresh look at (re-designing) the memory hierarchy
  - Scalability: DRAM-System Codesign and New Technologies
  - QoS: Reducing and controlling main memory interference: QoS-aware memory system design
  - Efficiency: Customizability, minimal waste, new technologies

- QoS-unaware memory: uncontrollable and unpredictable
- Providing QoS awareness improves performance, predictability, fairness, and utilization of the memory system

Summary: Memory QoS Approaches and Techniques

- Approaches: Smart vs. dumb resources
  - Smart resources: QoS-aware memory scheduling
  - Dumb resources: Source throttling; channel partitioning
  - Both approaches are effective in reducing interference
  - No single best approach for all workloads
- Techniques: Request/thread scheduling, source throttling, memory partitioning
  - All approaches are effective in reducing interference
  - Can be applied at different levels: hardware vs. software
  - No single best technique for all workloads
- Combined approaches and techniques are the most powerful
  - Integrated Memory Channel Partitioning and Scheduling [MICRO'11]

Cache Potpourri: Managing Waste

Onur Mutlu
onur@cmu.edu
July 9, 2013
INRIA

Carnegie Mellon

SAFARI

More Efficient Cache Utilization

- Compressing redundant data
- Reducing pollution and 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

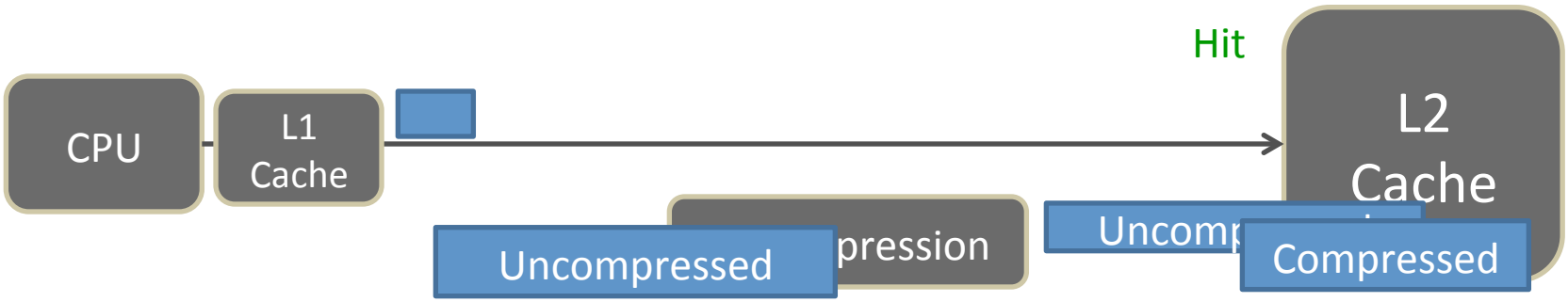
Significant redundancy in data:

0x**00000000**	0x**0000000**B	0x**00000000**3	0x**00000000**4	…

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

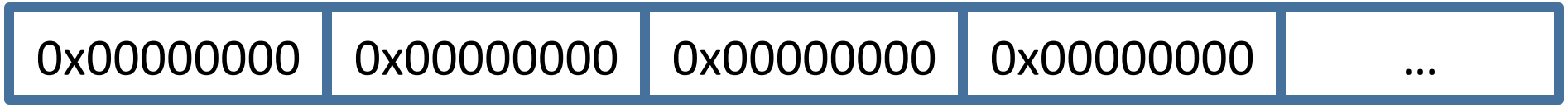
Compression Mechanisms	Decompression Latency	Complexity	Compression Ratio
Zero	✓	✓	✗
Frequent Value	✗	✗	✓
Frequent Pattern	✗	✗/✓	✓
Our proposal: BΔI	✓	✓	✓

Outline

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

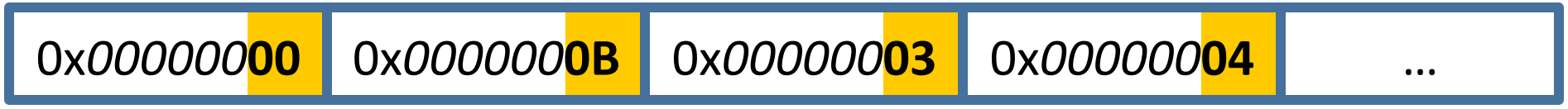
Key Data Patterns in Real Applications

Zero Values: initialization, sparse matrices, NULL pointers

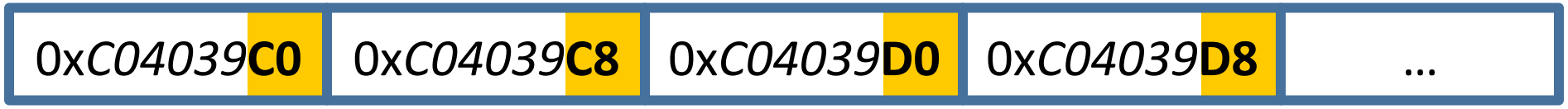


Repeated Values: common initial values, adjacent pixels

Narrow Values: small values stored in a big data type

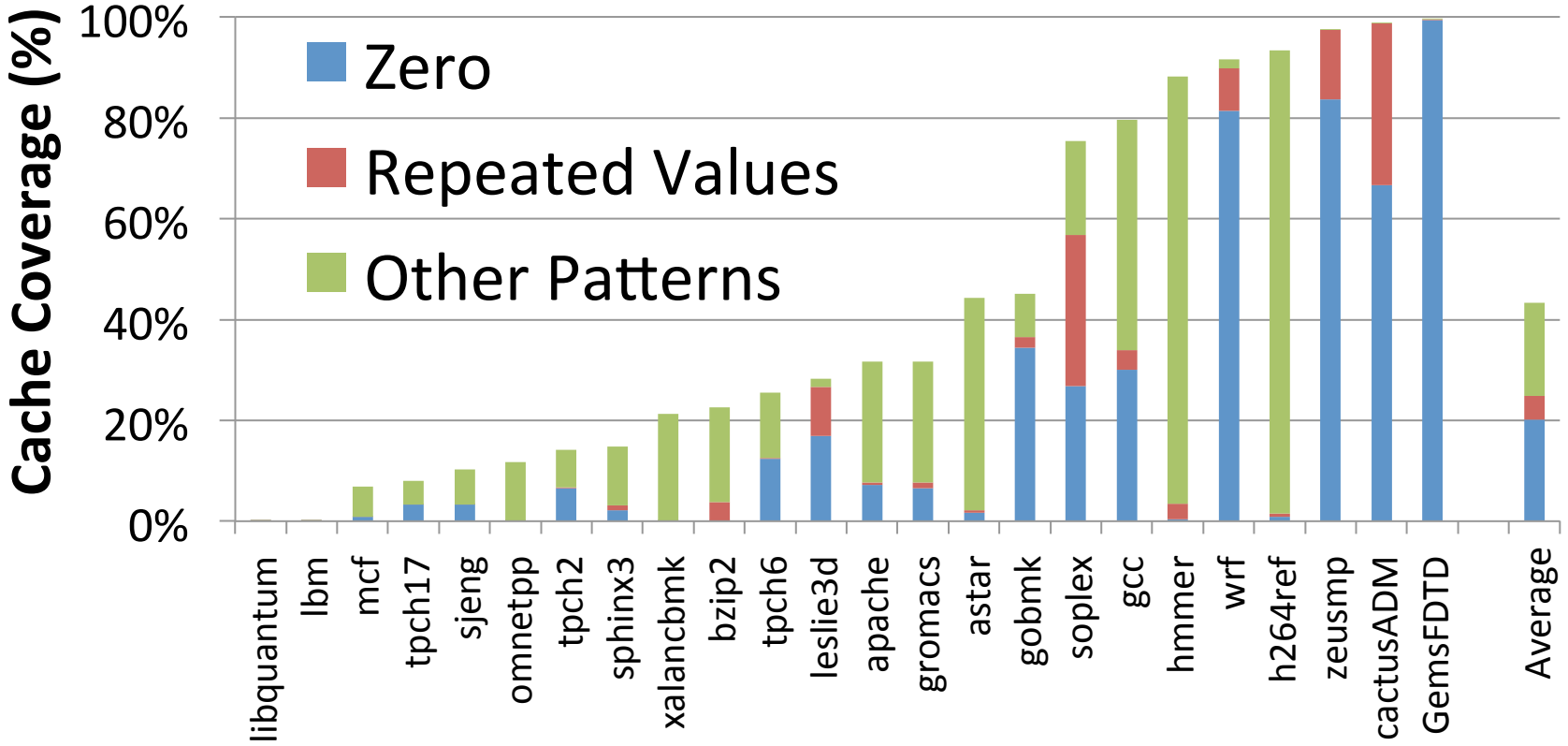


Other Patterns: pointers to the same memory region



How Common Are These Patterns?

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



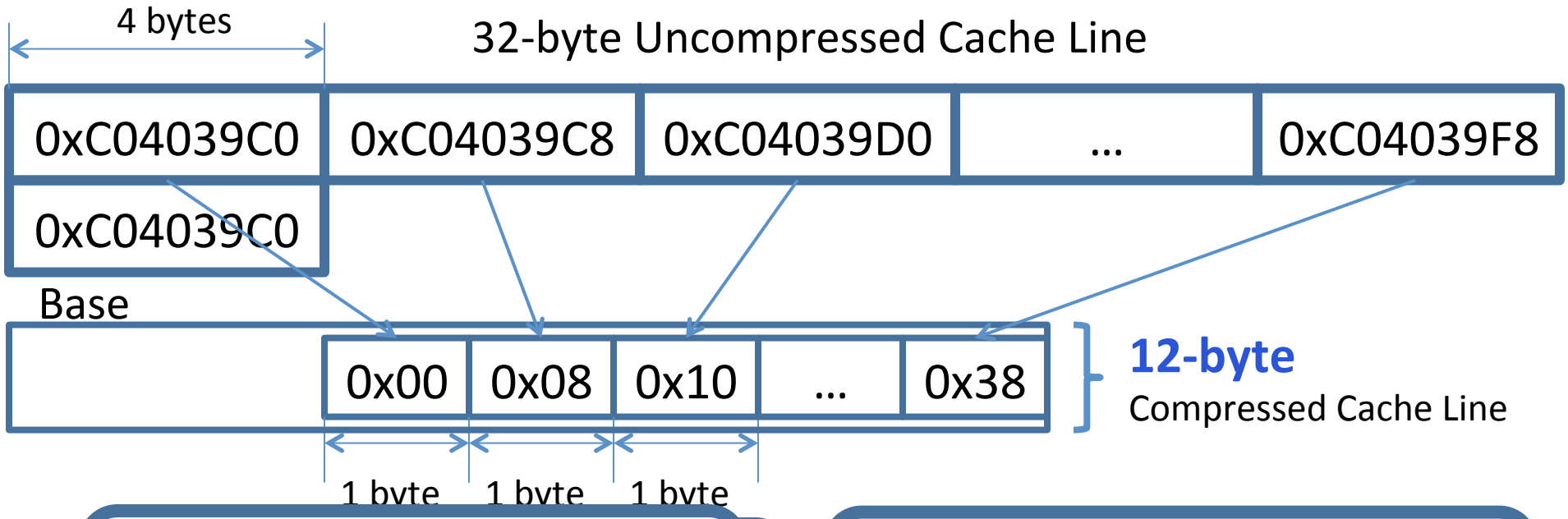
43% of the cache lines belong to key patterns

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

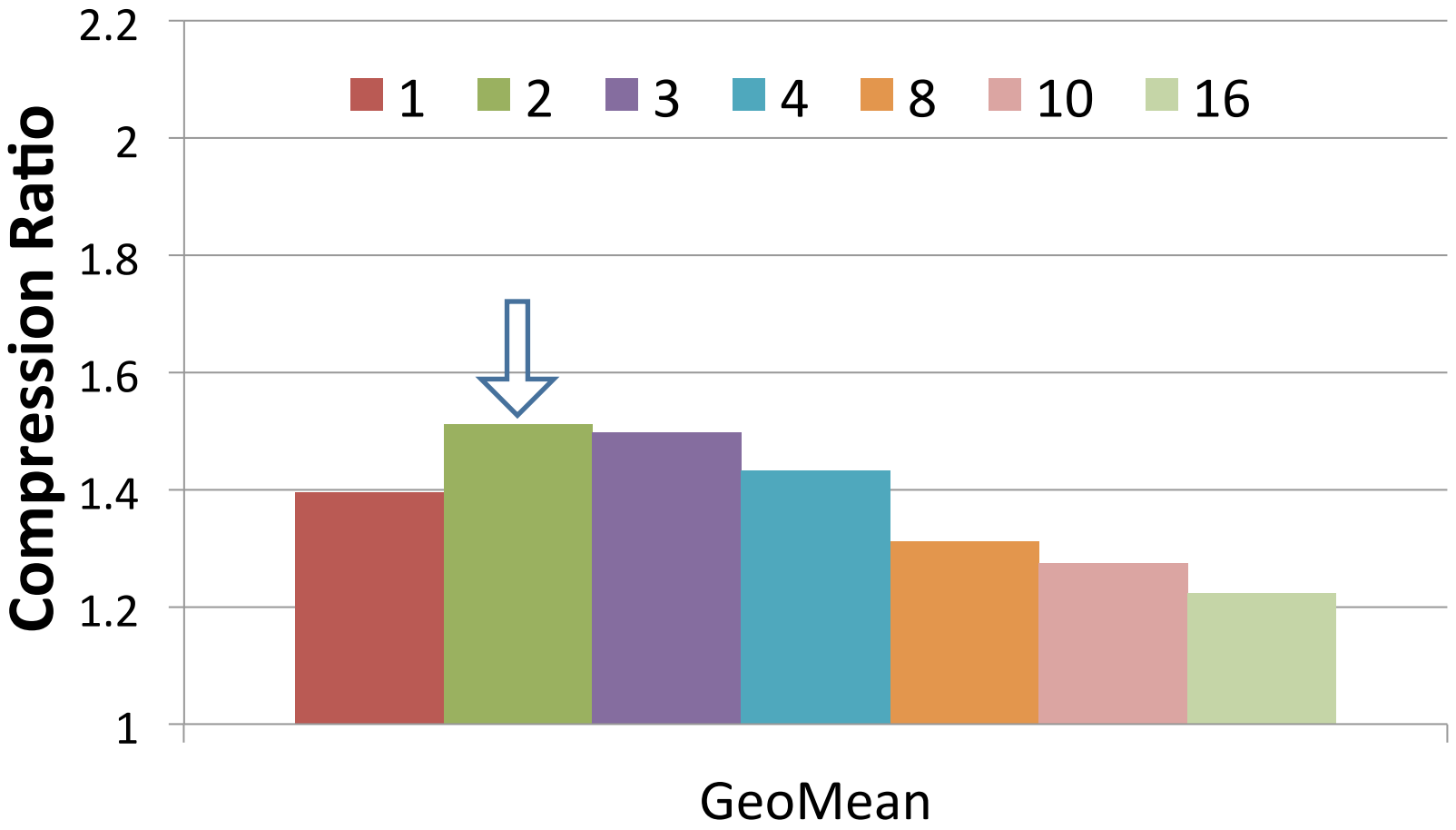
Can We Do Better?

- 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**

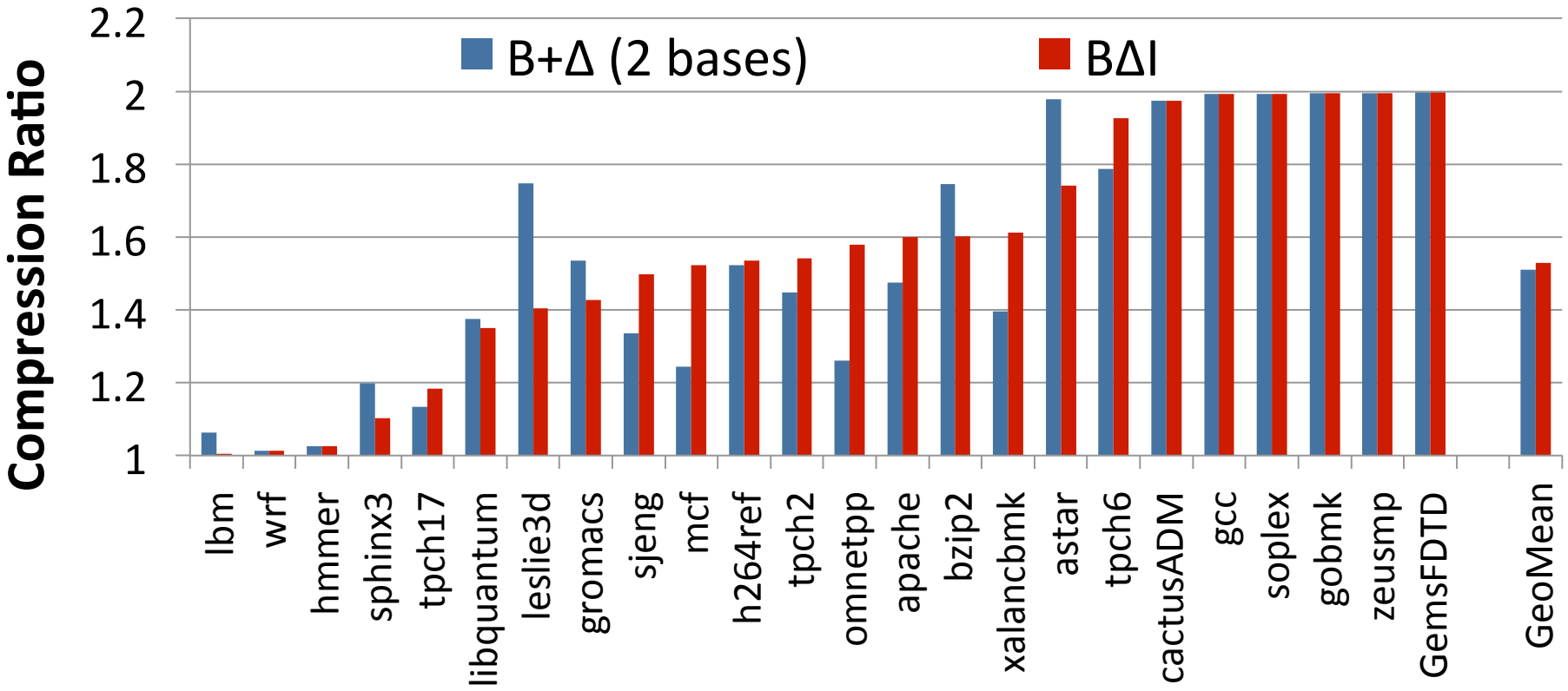
   ✓ **Immediate part**

Advantages over 2 arbitrary bases:

- Better compression ratio
- Simpler compression logic

Base-Delta-Immediate (BΔI) Compression

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

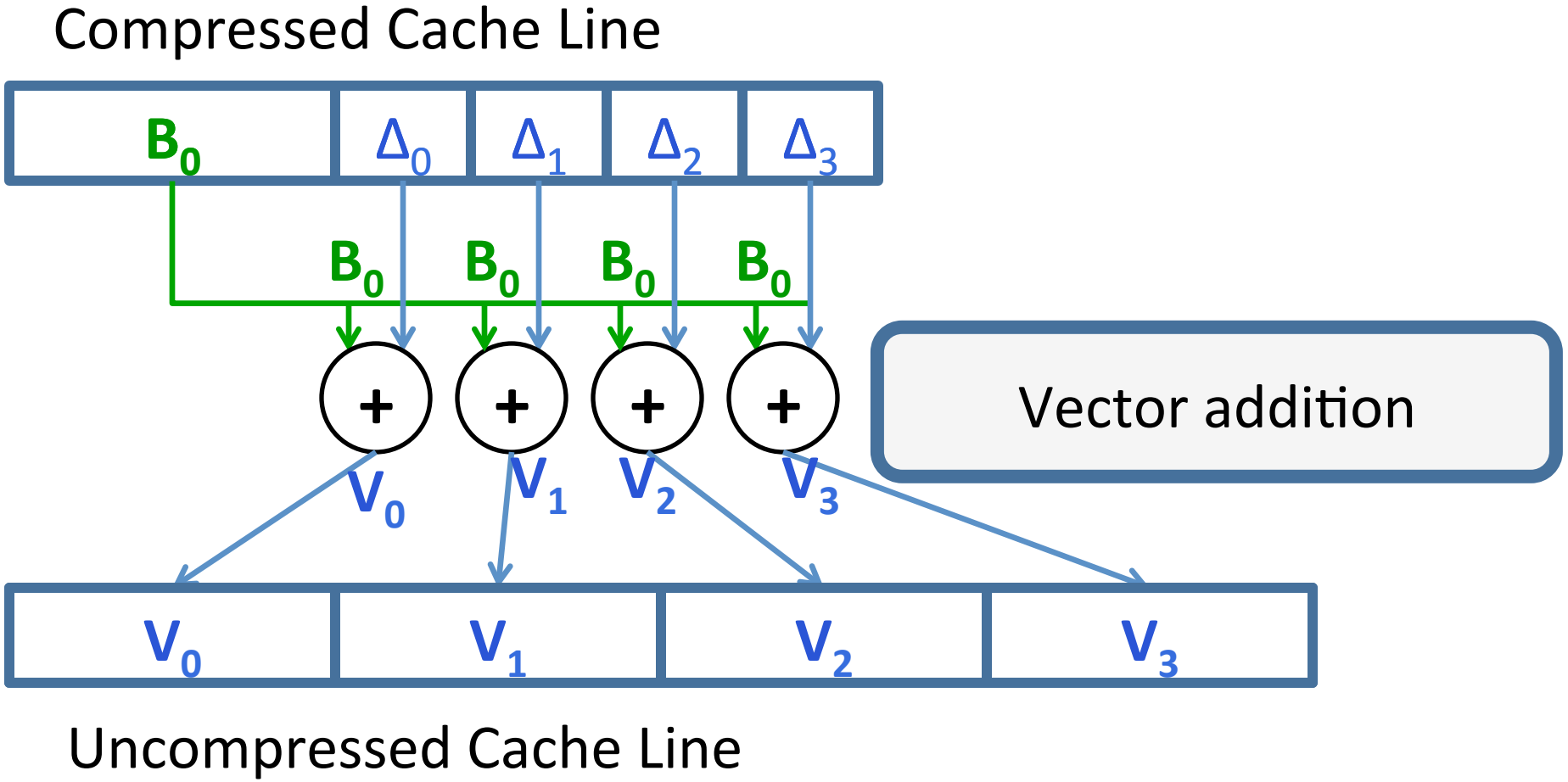


Average compression ratio is close, but **BΔI** is **simpler**

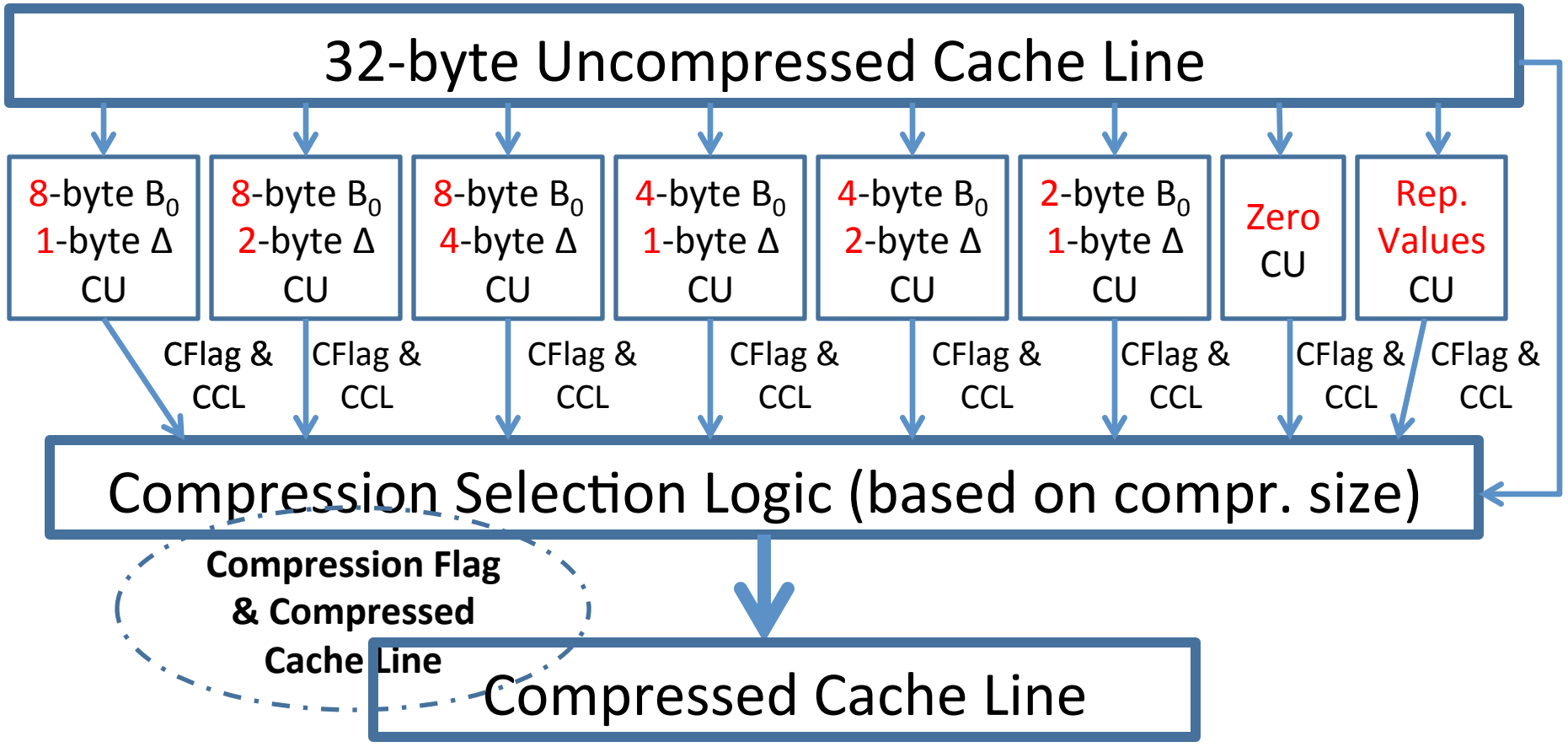
BΔI Implementation

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

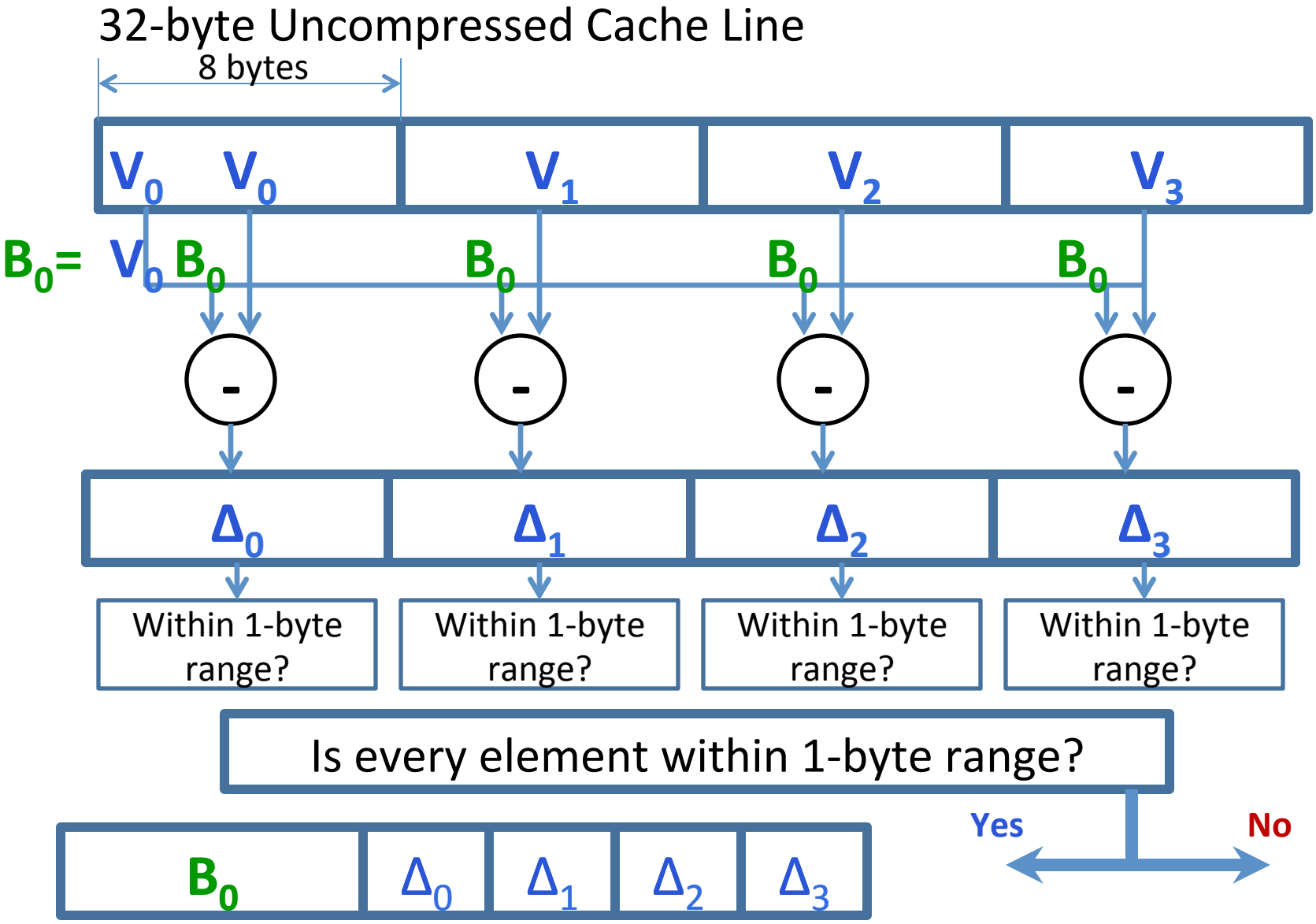
BΔI Decompressor Design



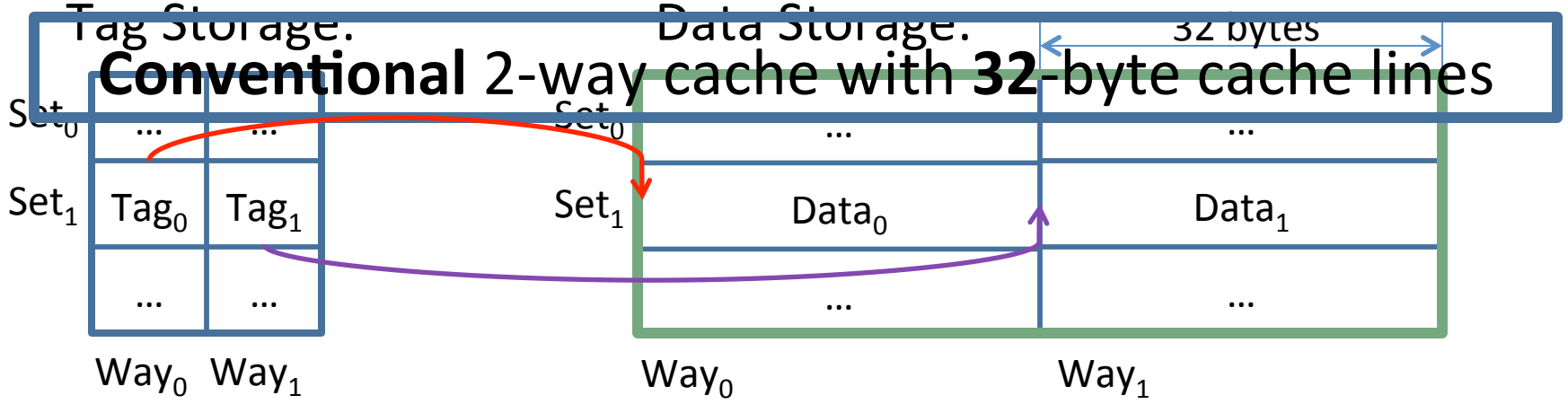
BΔI Compressor Design

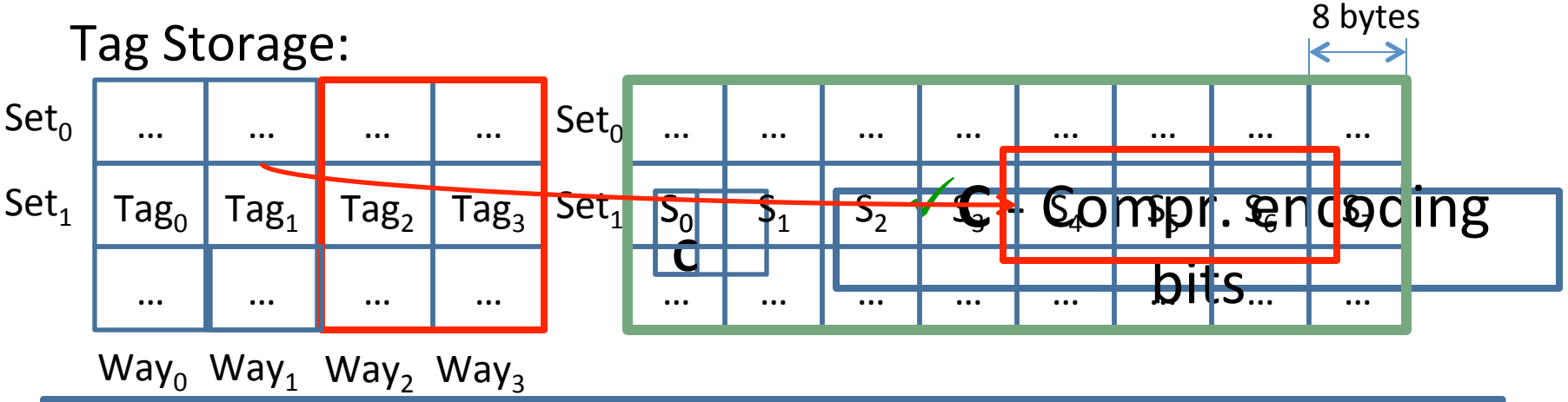


BΔI Compression Unit: 8-byte B_0 1-byte Δ



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)

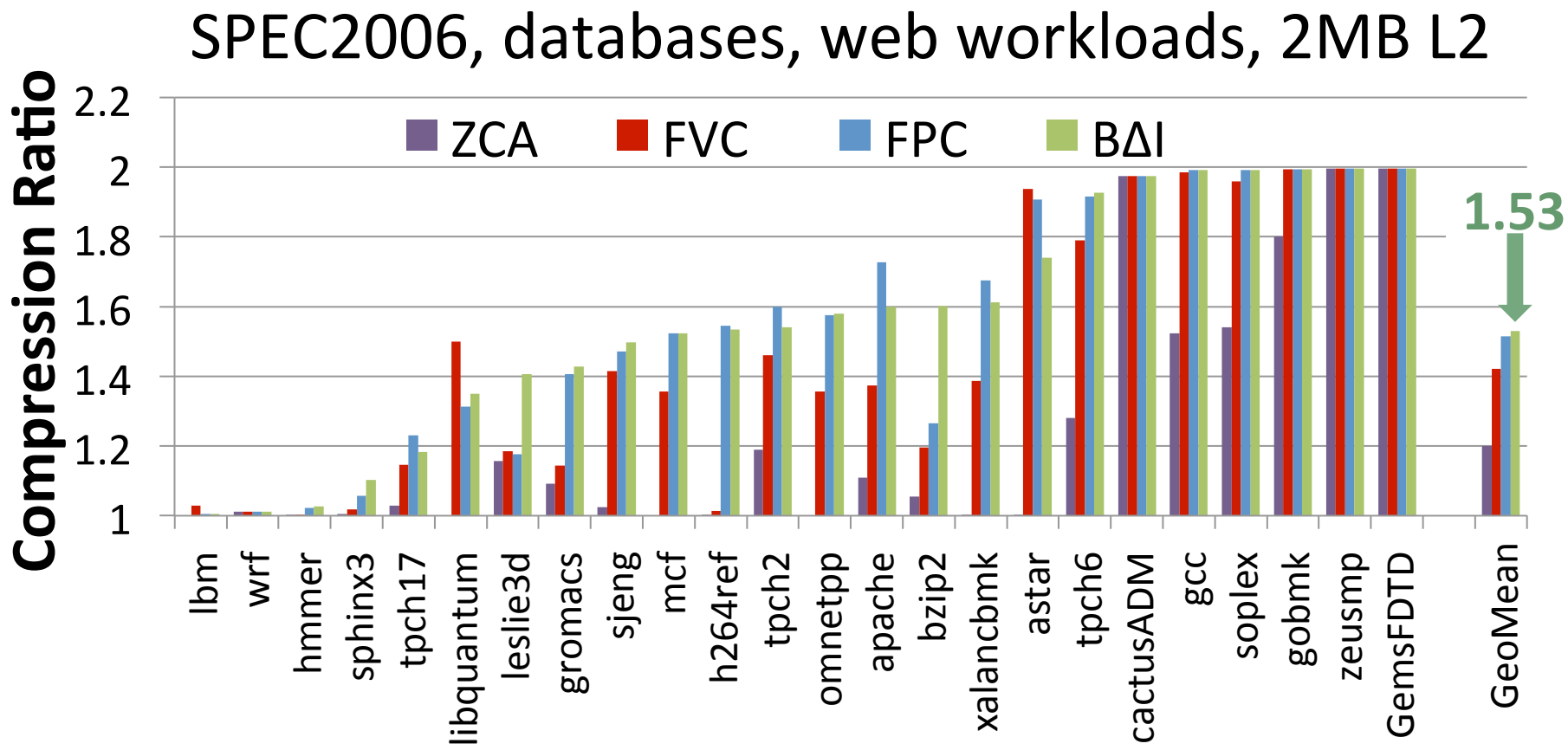
Outline

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

Methodology

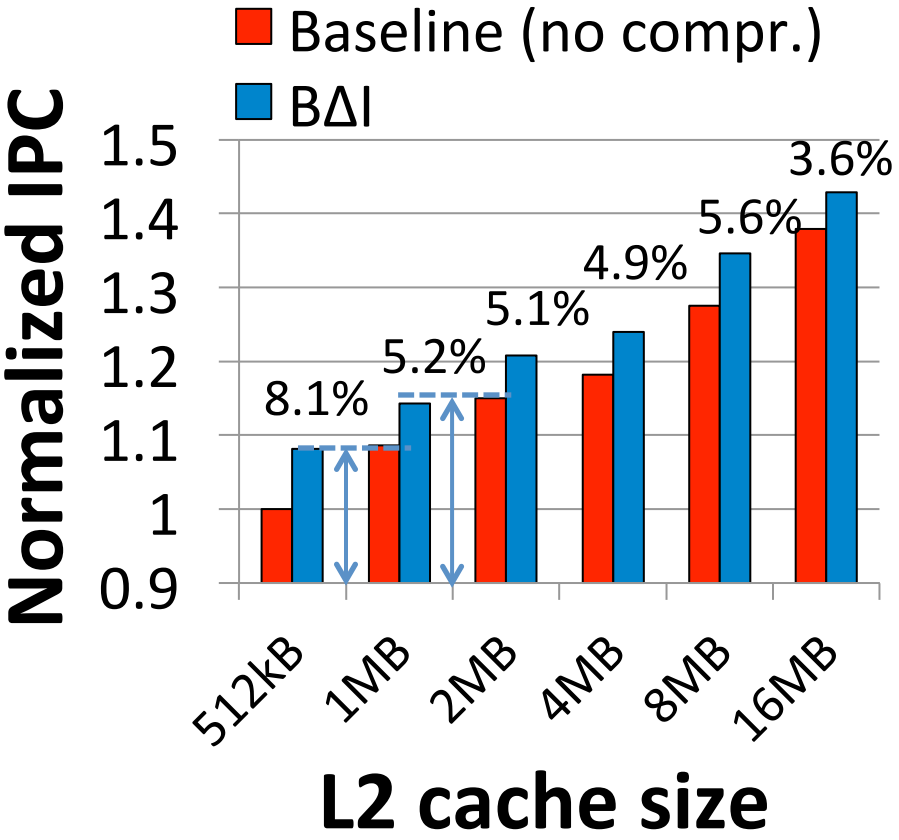
- **Simulator**
  - x86 event-driven simulator based on Simics *[Magnusson+, Computer'02]*
- **Workloads**
  - SPEC2006 benchmarks, TPC, Apache web server
  - 1 – 4 core simulations for 1 billion representative instructions
- **System Parameters**
  - L1/L2/L3 cache latencies from CACTI *[Thoziyoor+, ISCA'08]*
  - 4GHz, x86 in-order core, **512kB - 16MB** L2, simple memory model (**300**-cycle latency for row-misses)

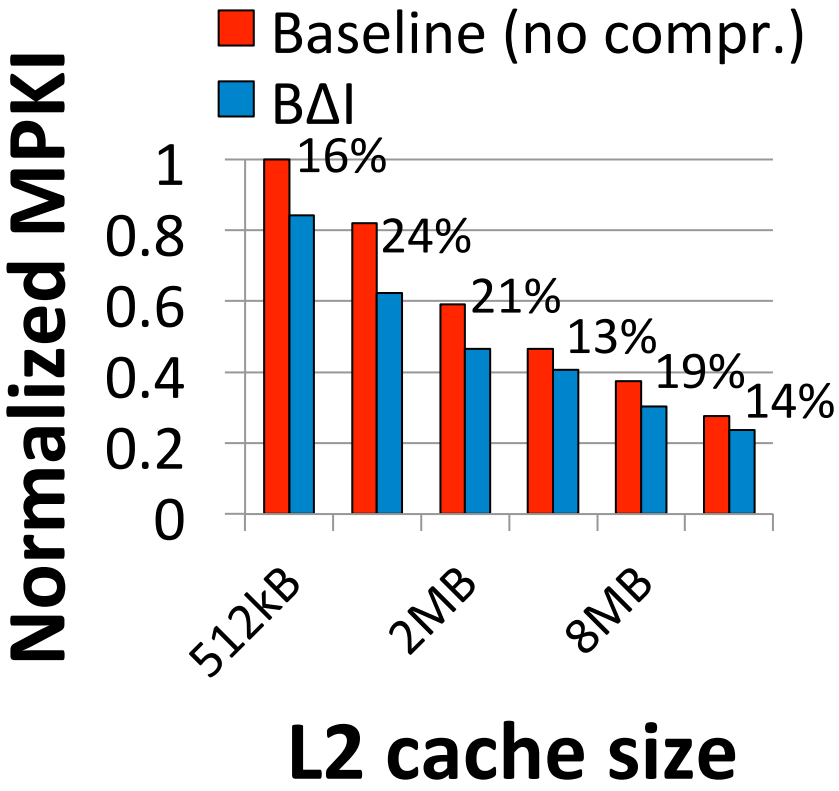
Compression Ratio: BΔI vs. Prior Work



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)

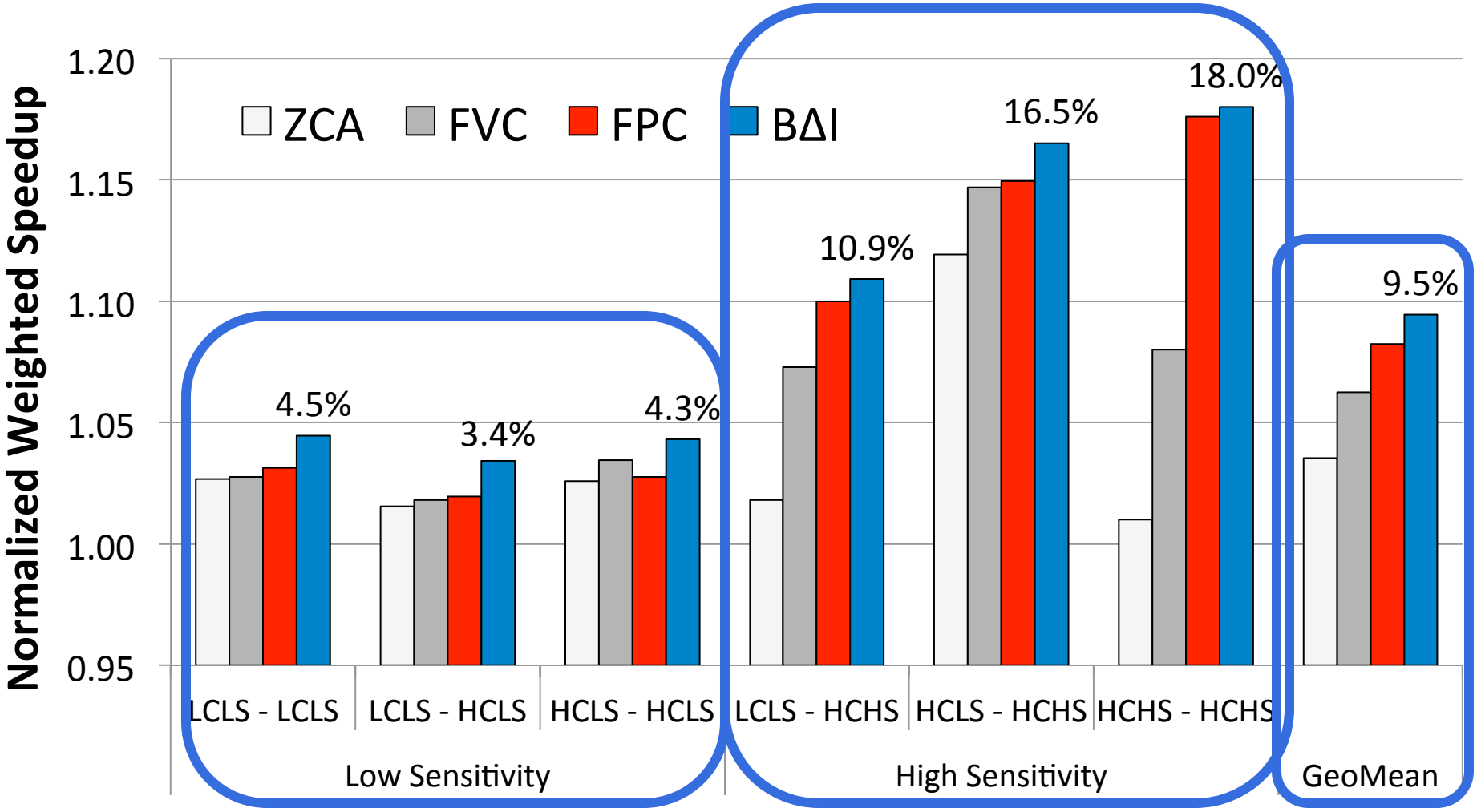
  Sensitivity: performance gain with more cache

  (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

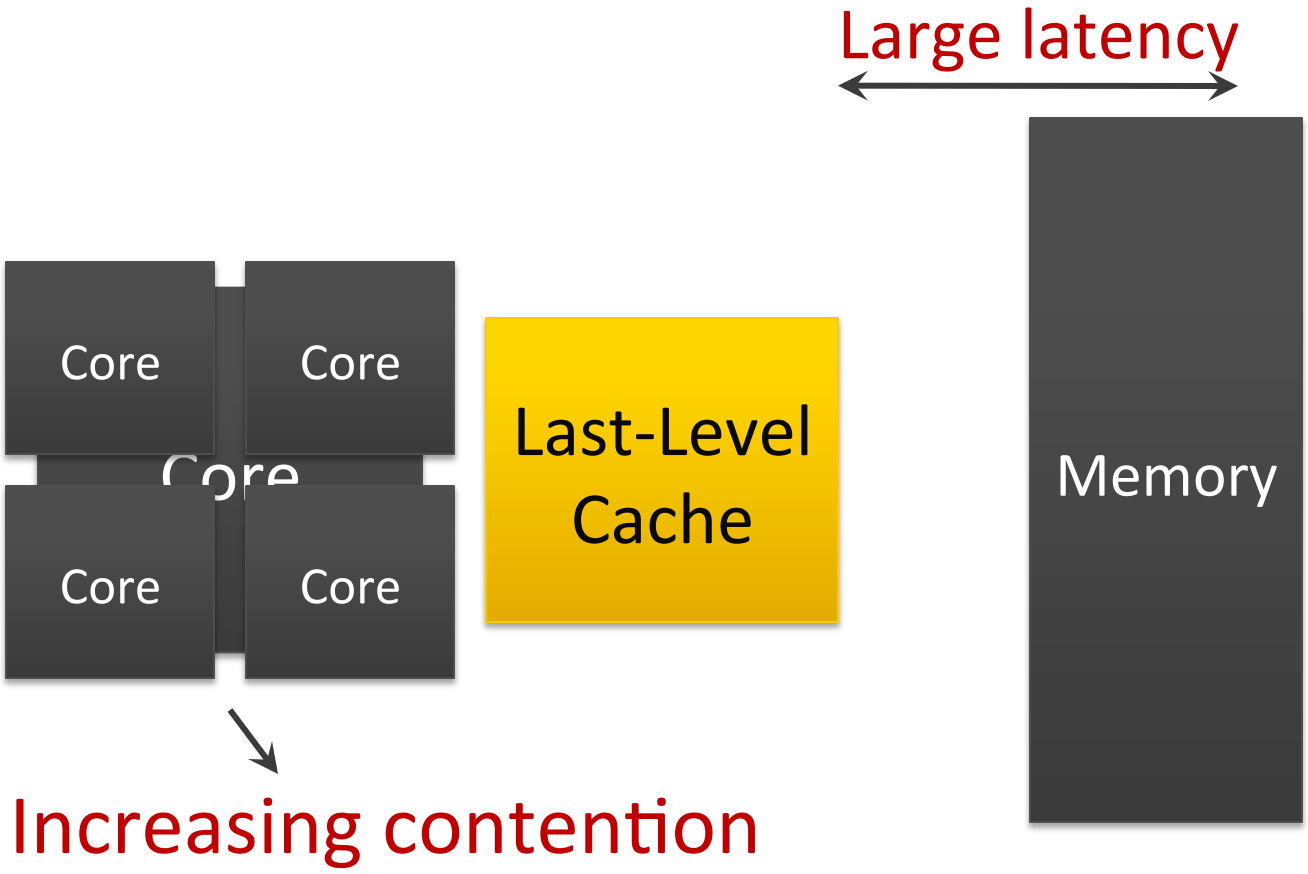
The Evicted-Address Filter

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

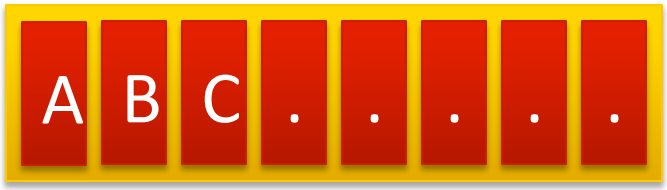
Different blocks have different reuse behavior

Access Sequence:


High-reuse block


Low-reuse block

Ideal Cache



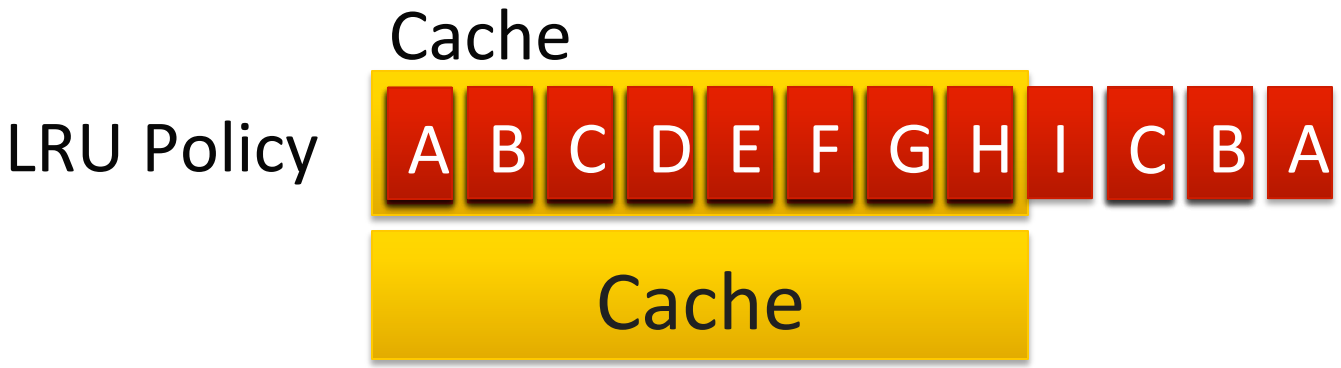
Cache Pollution

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**)

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

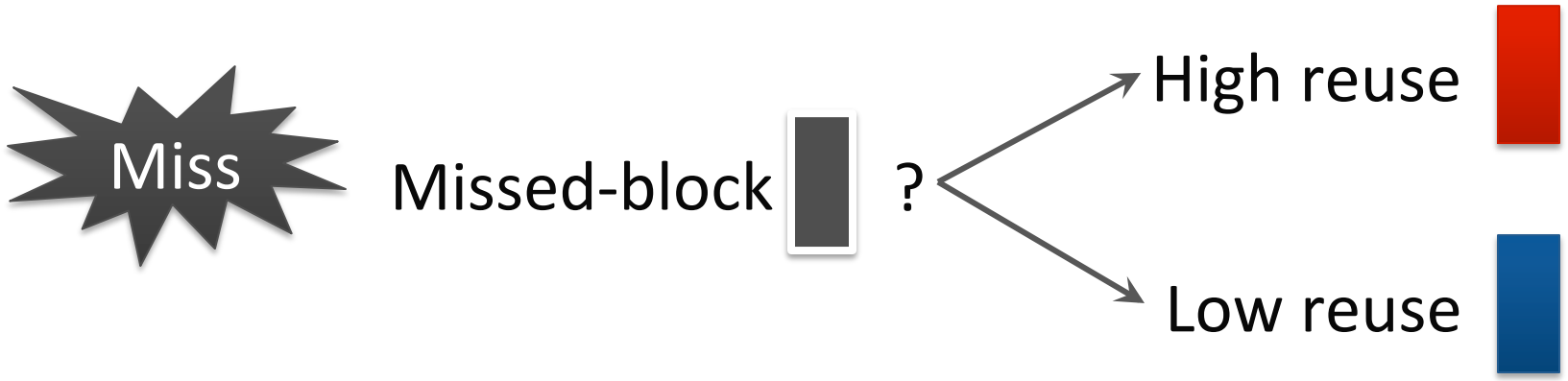
No mechanism to distinguish high-reuse blocks from low-reuse blocks

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

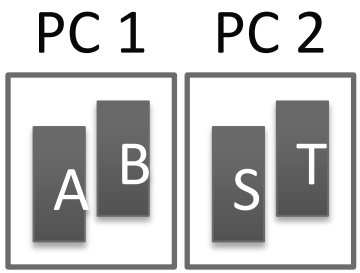
Impractical

1. High storage overhead
2. Look-up latency

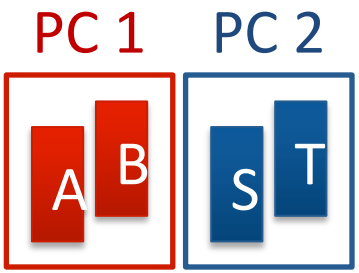
Prior Work on Reuse Prediction

Use program counter or memory region information.

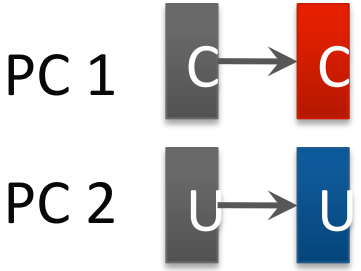
1. Group Blocks



2. Learn group behavior



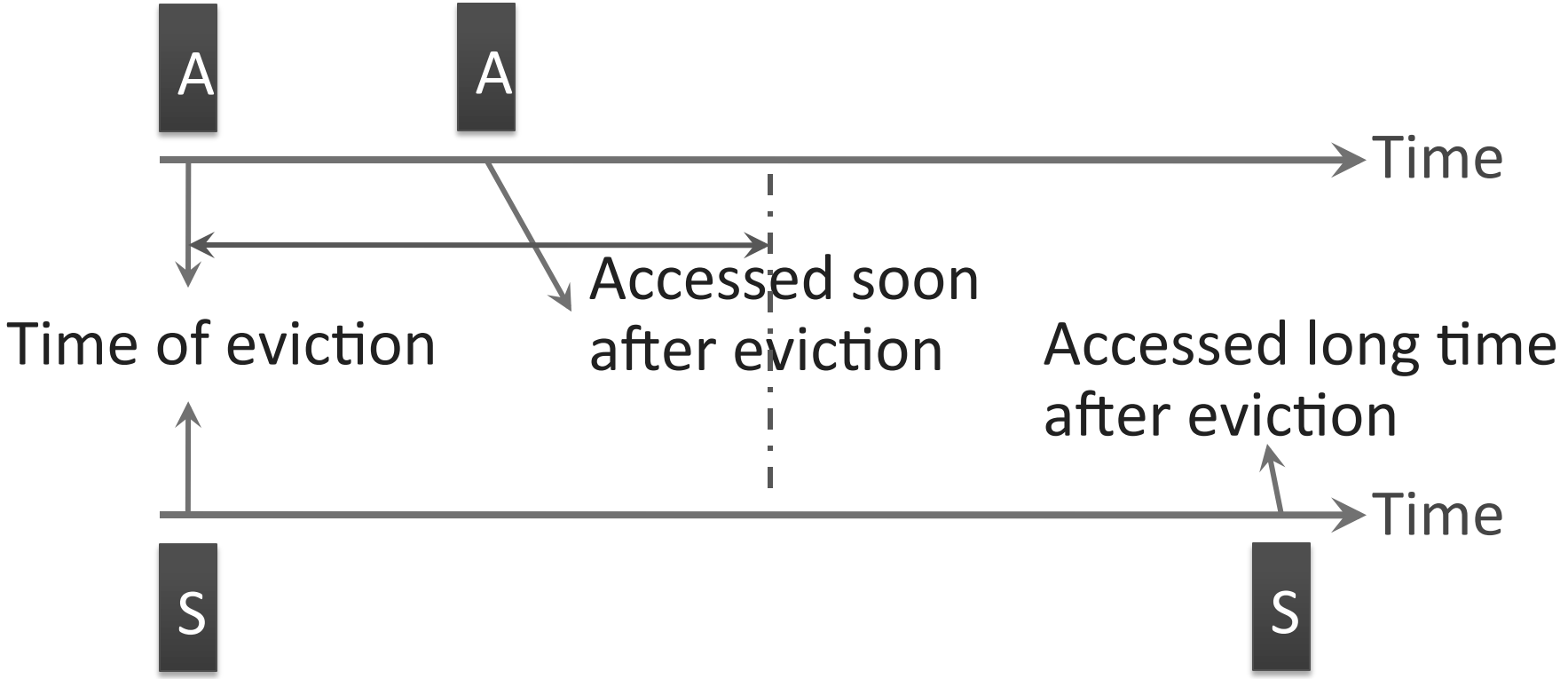
3. Predict reuse



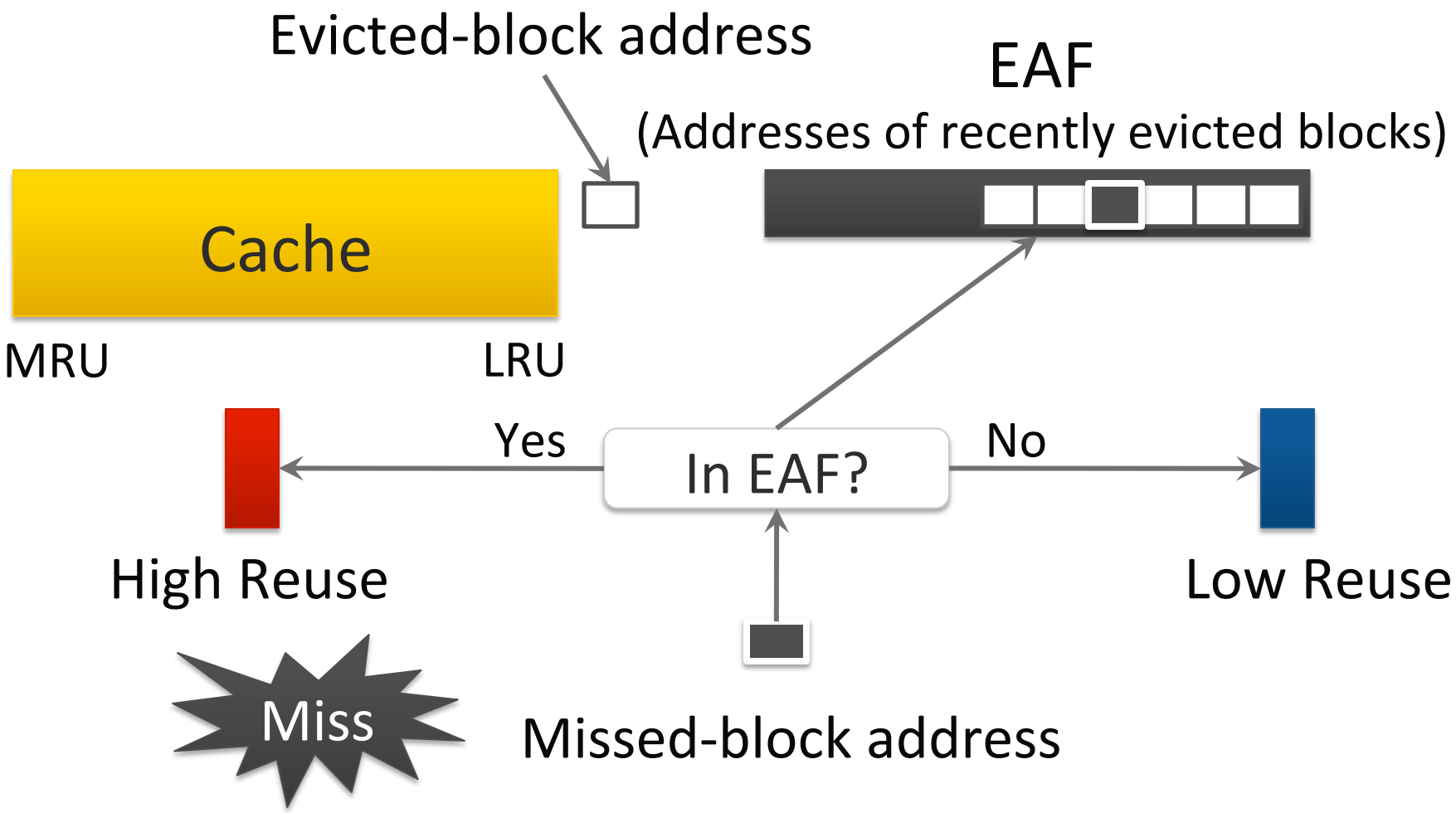
1. Same group $\not\rightarrow$ same reuse behavior
2. No control over number of high-reuse blocks

Our Approach: Per-block Prediction

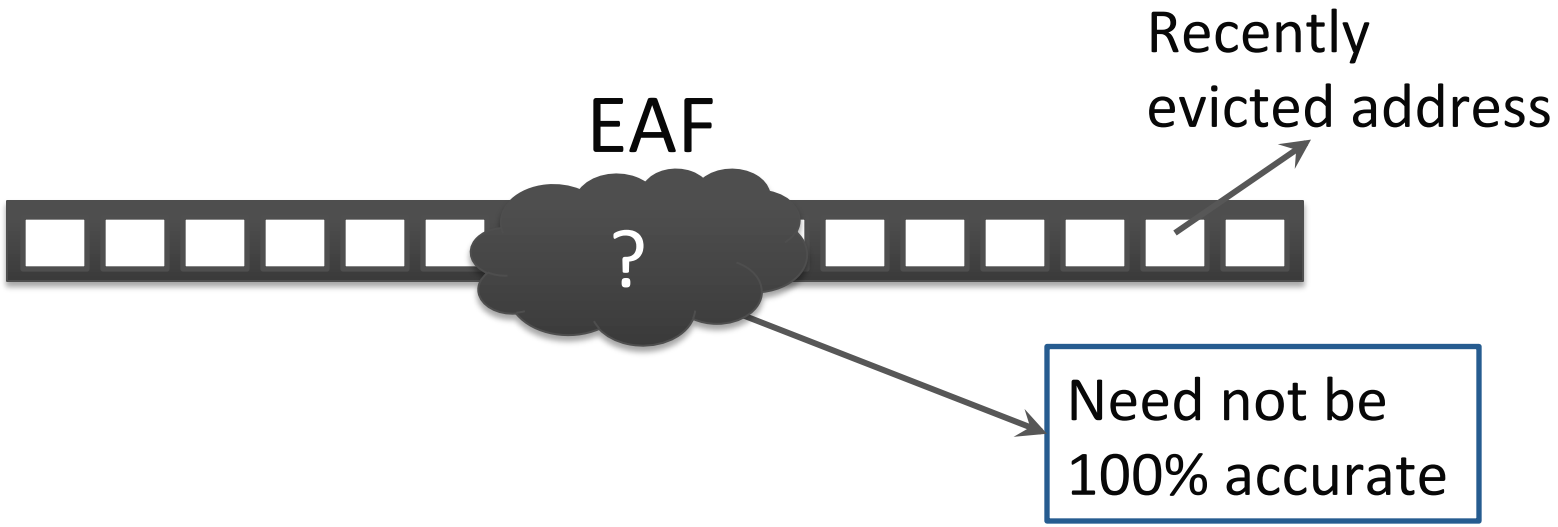
Use recency of eviction to predict reuse



Evicted-Address Filter (EAF)

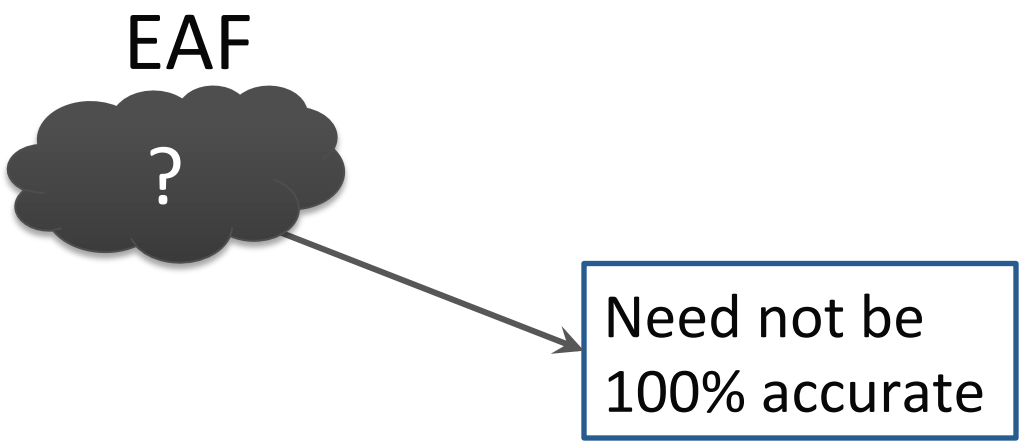


Naïve Implementation: Full Address Tags



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

Low-Cost Implementation: Bloom Filter

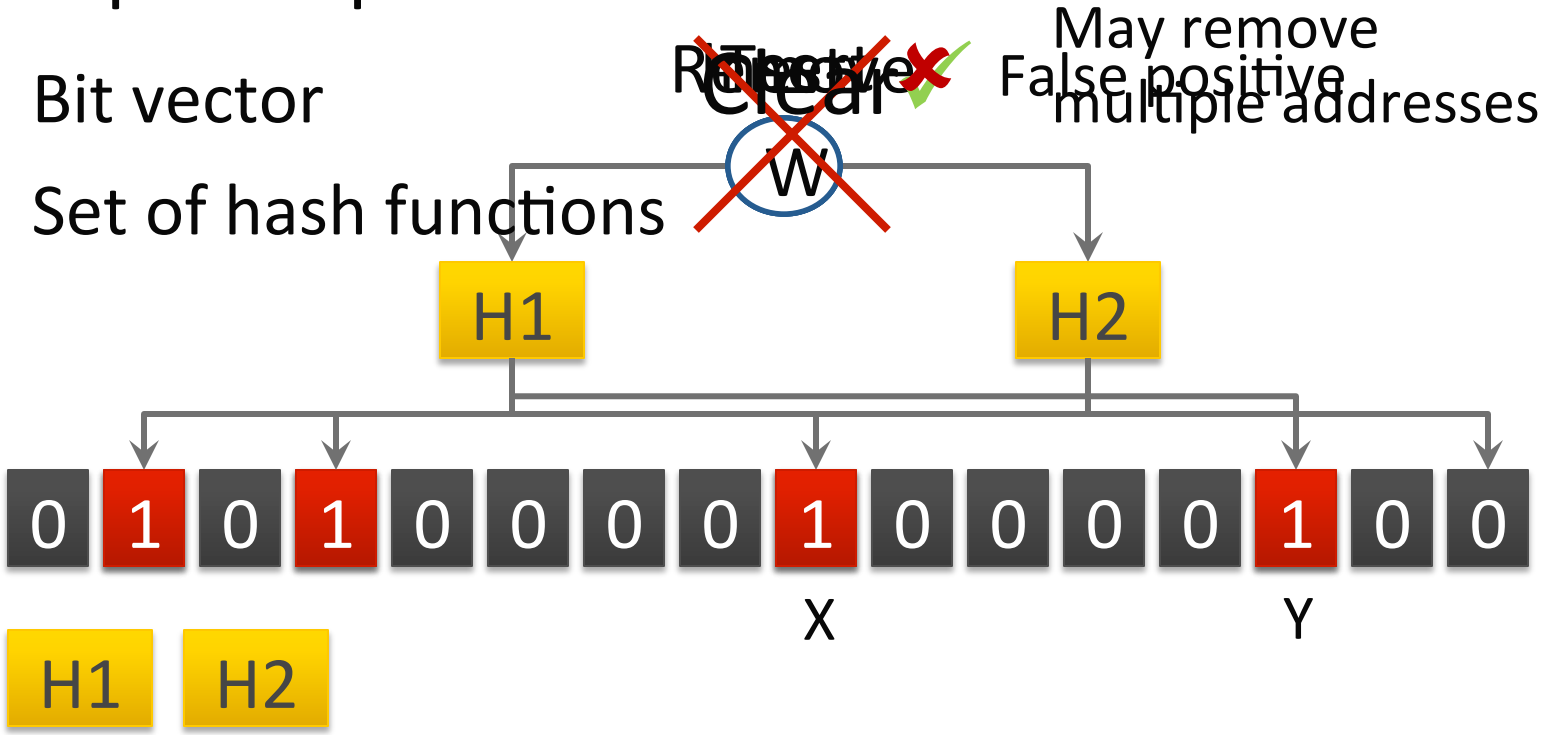


Implement EAF using a **Bloom Filter**
Low storage overhead + energy

Bloom Filter

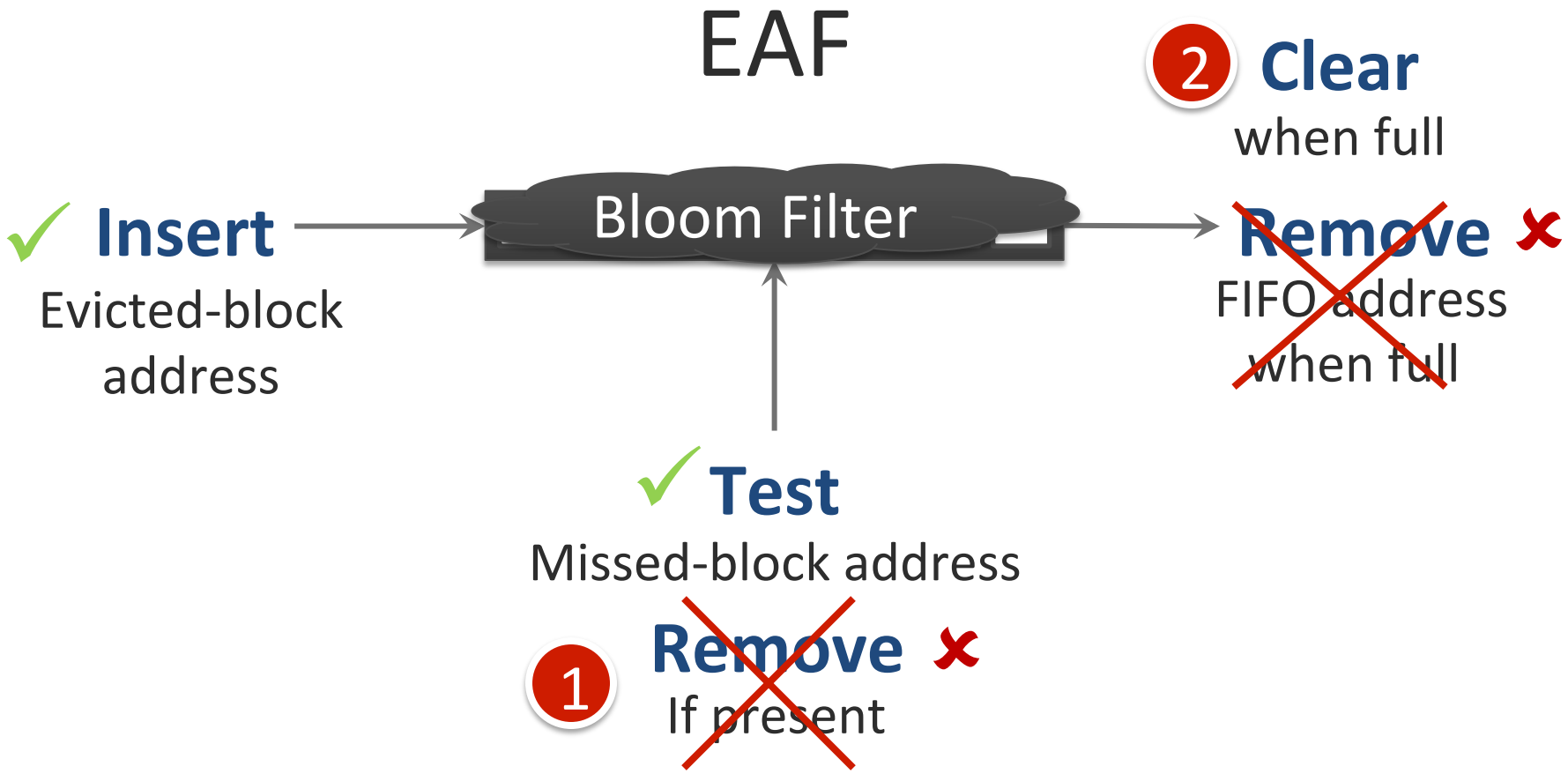
Compact representation of a set

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

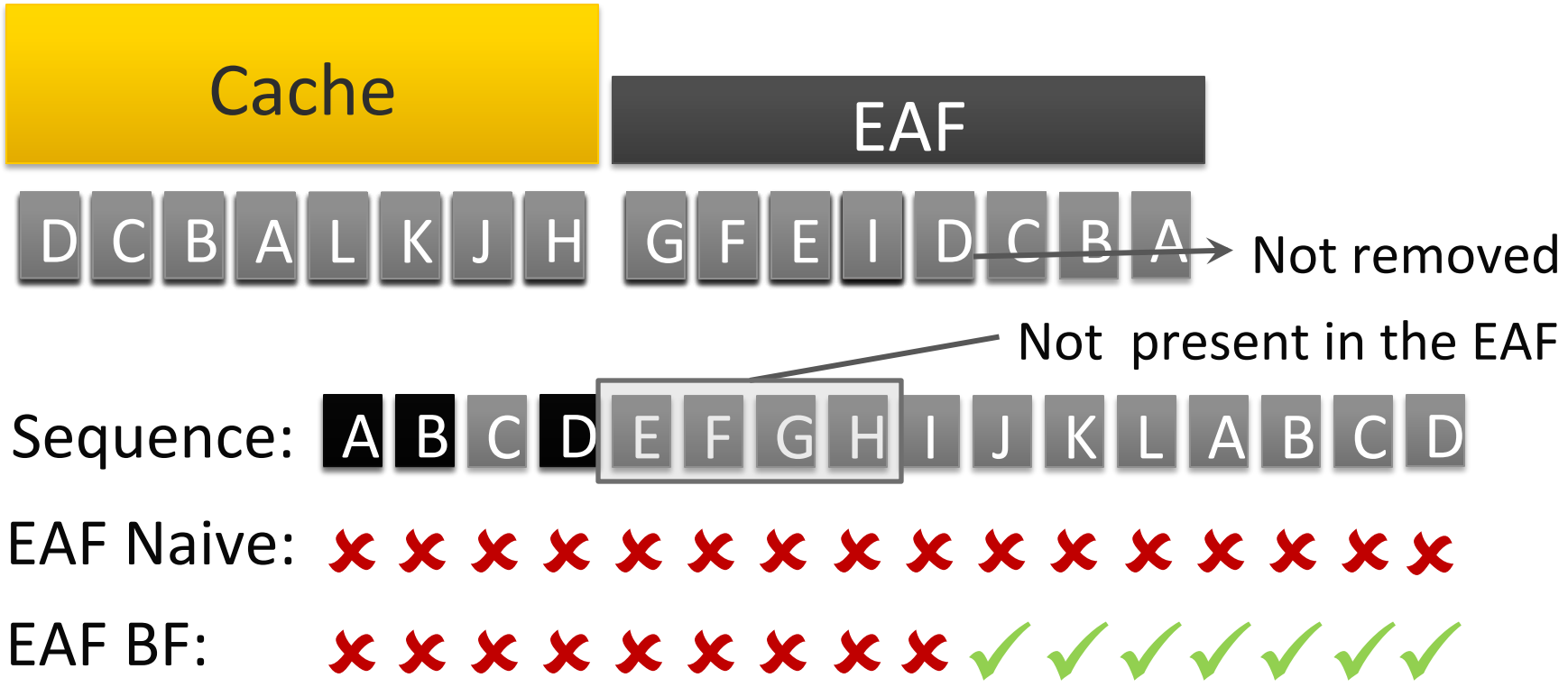
Cache < Working set < Cache + EAF

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

EAF Naive: ✗ ✗ ✗ ✗ ✗ ✗ ✗ ✗ ✗ ✗ ✗ ✗ ✗ ✗ ✗ ✗

Large Working Set: Case 1

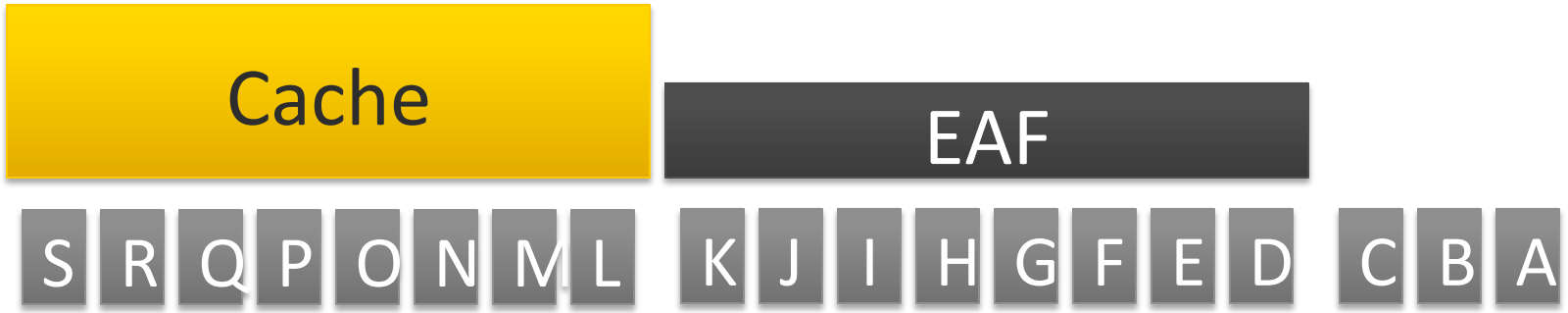
Cache < Working set < Cache + EAF



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

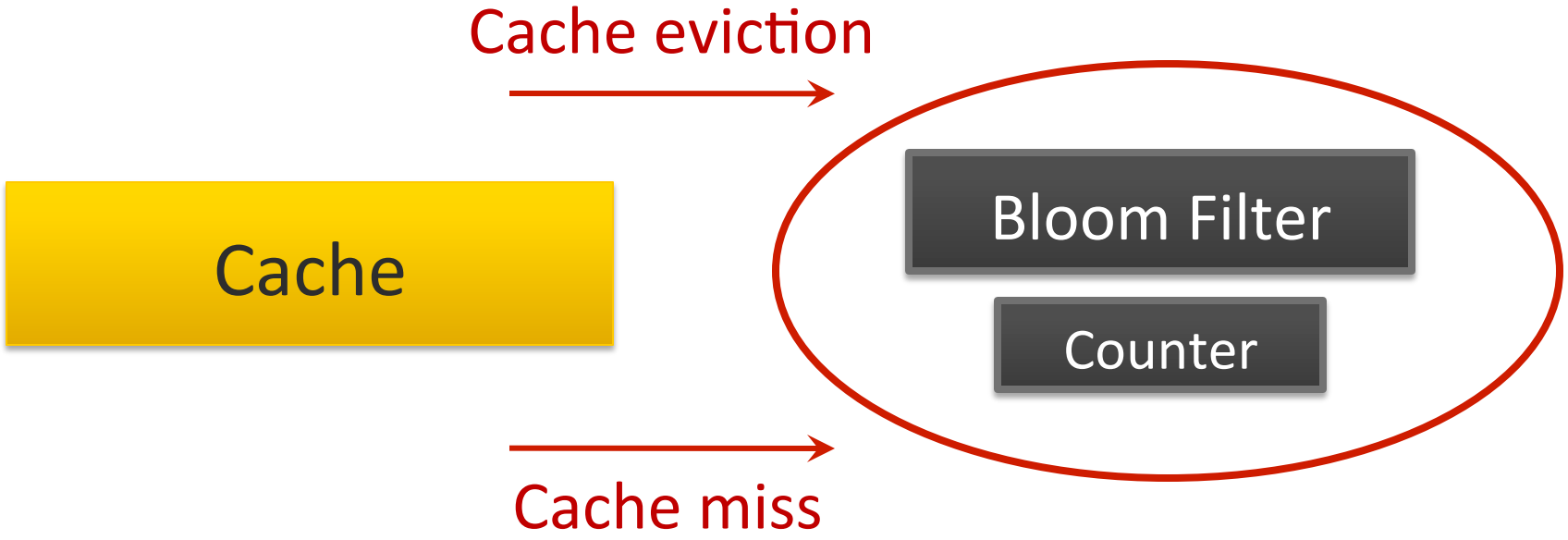
3 **Counter reaches max**
Clear filter and counter

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

Outline

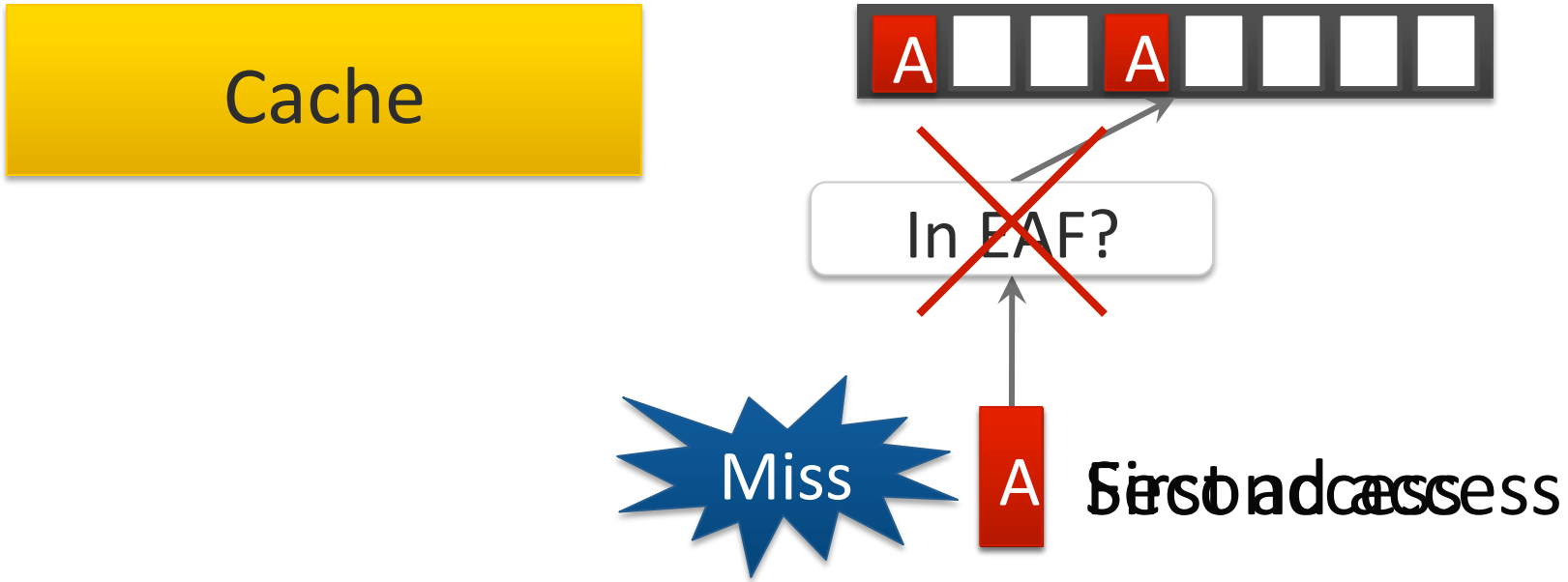
- Background and Motivation
- Evicted-Address Filter
  - Reuse Prediction
  - Thrash Resistance
- Final Design
- Advantages and Disadvantages
- Evaluation
- Conclusion

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 $\Longrightarrow$ Thrashing

Comparison with Prior Works

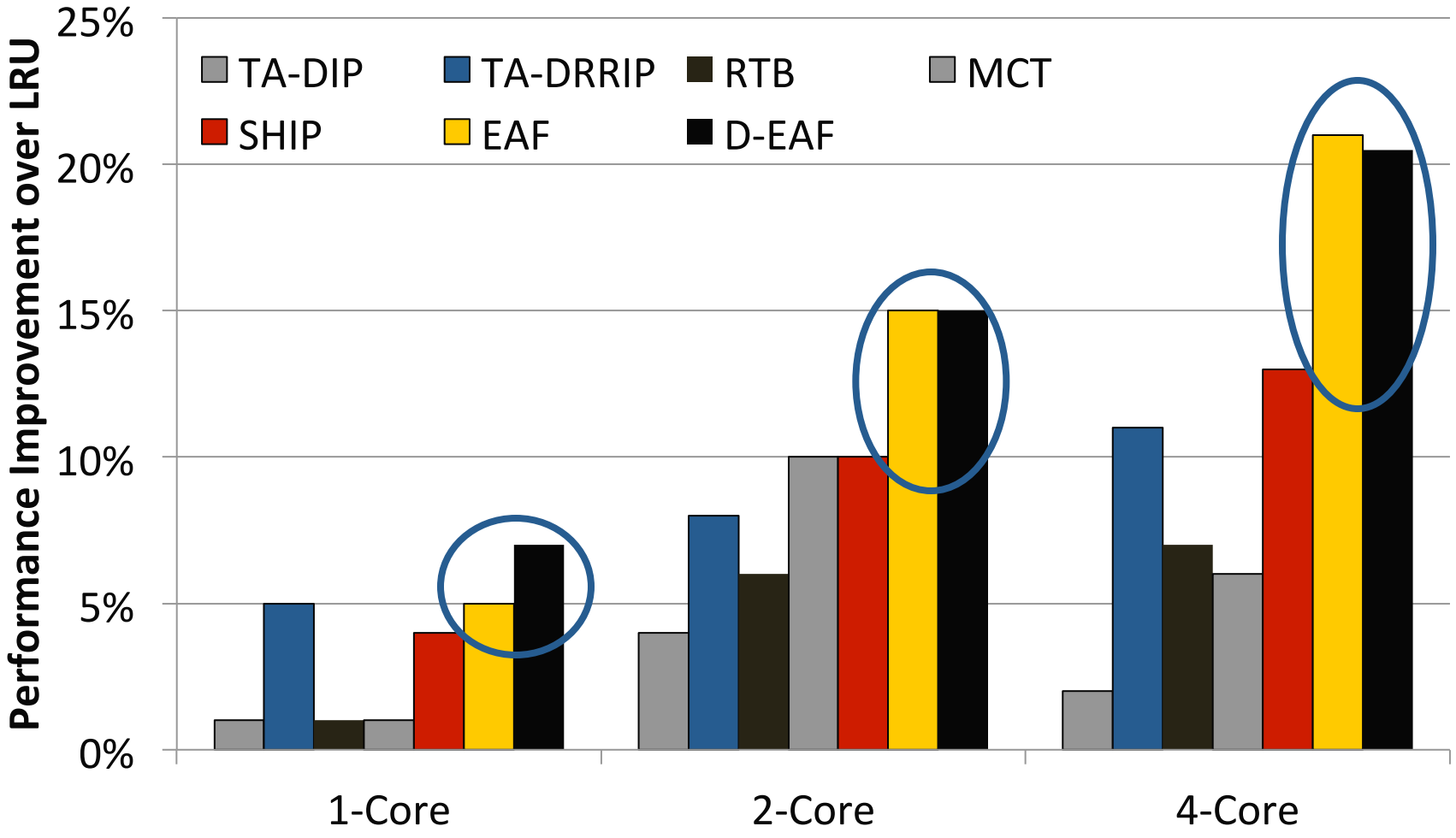
Addressing Cache Thrashing

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

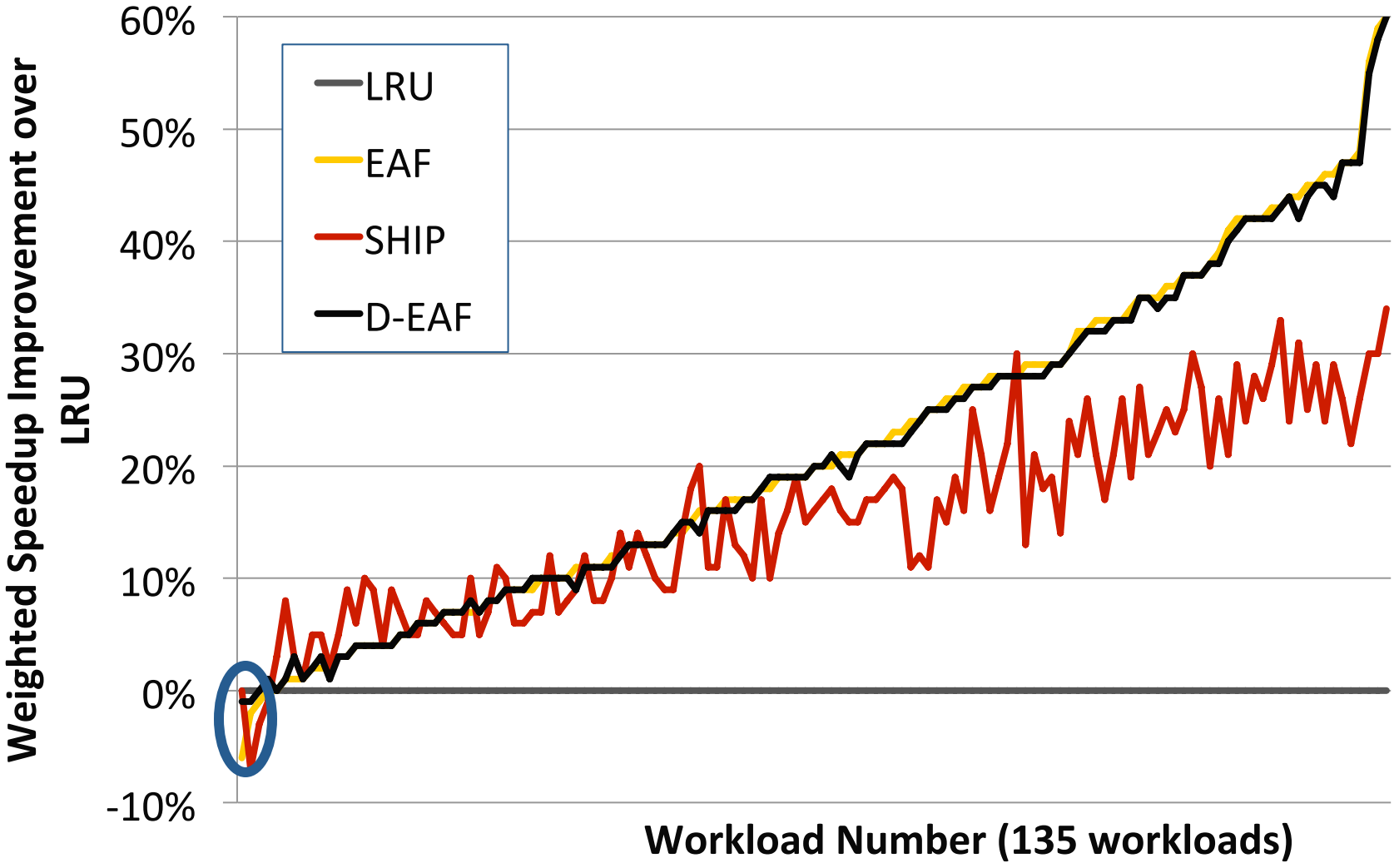
- Use set dueling to determine thrashing applications

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

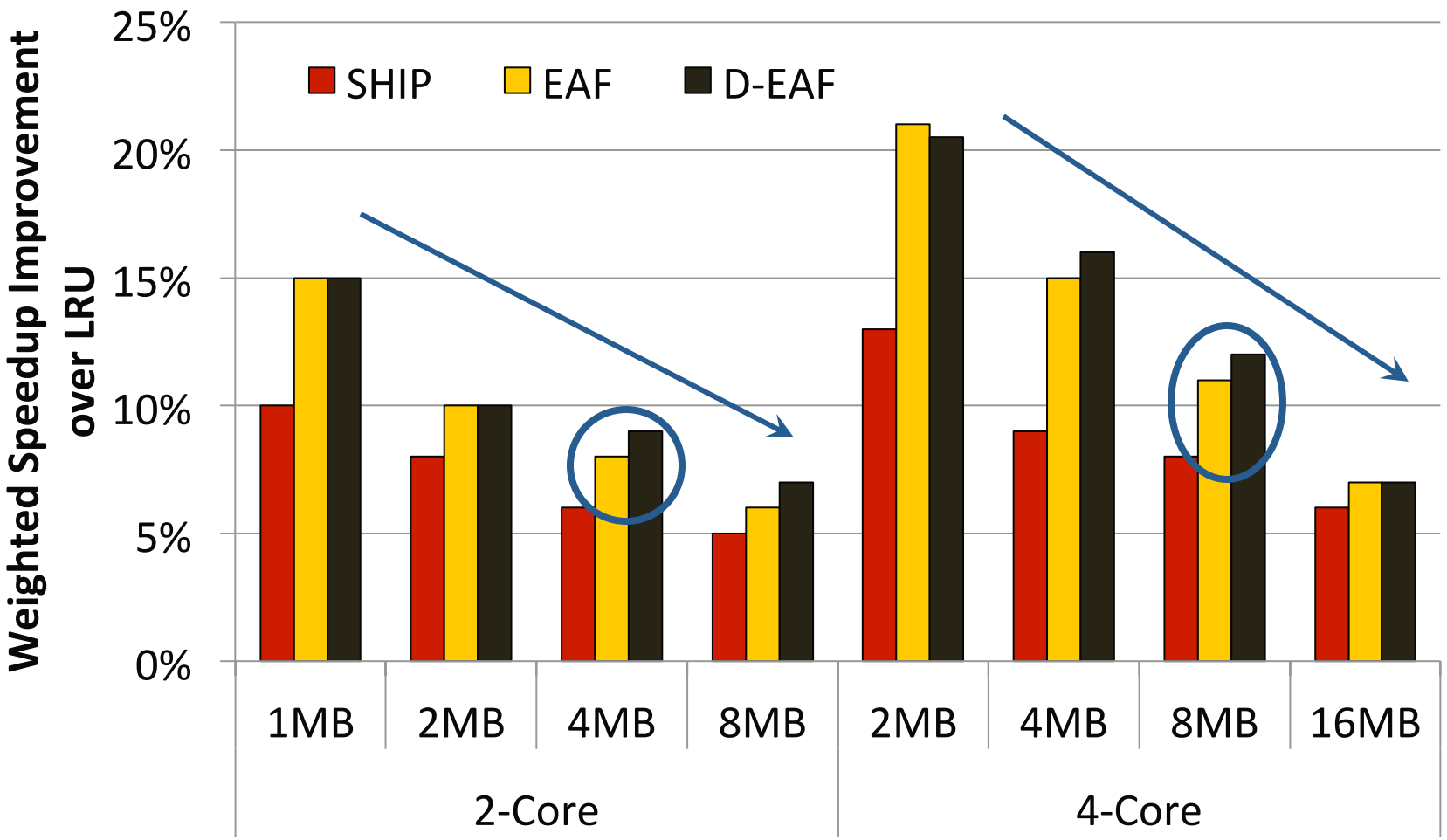
Results – Summary



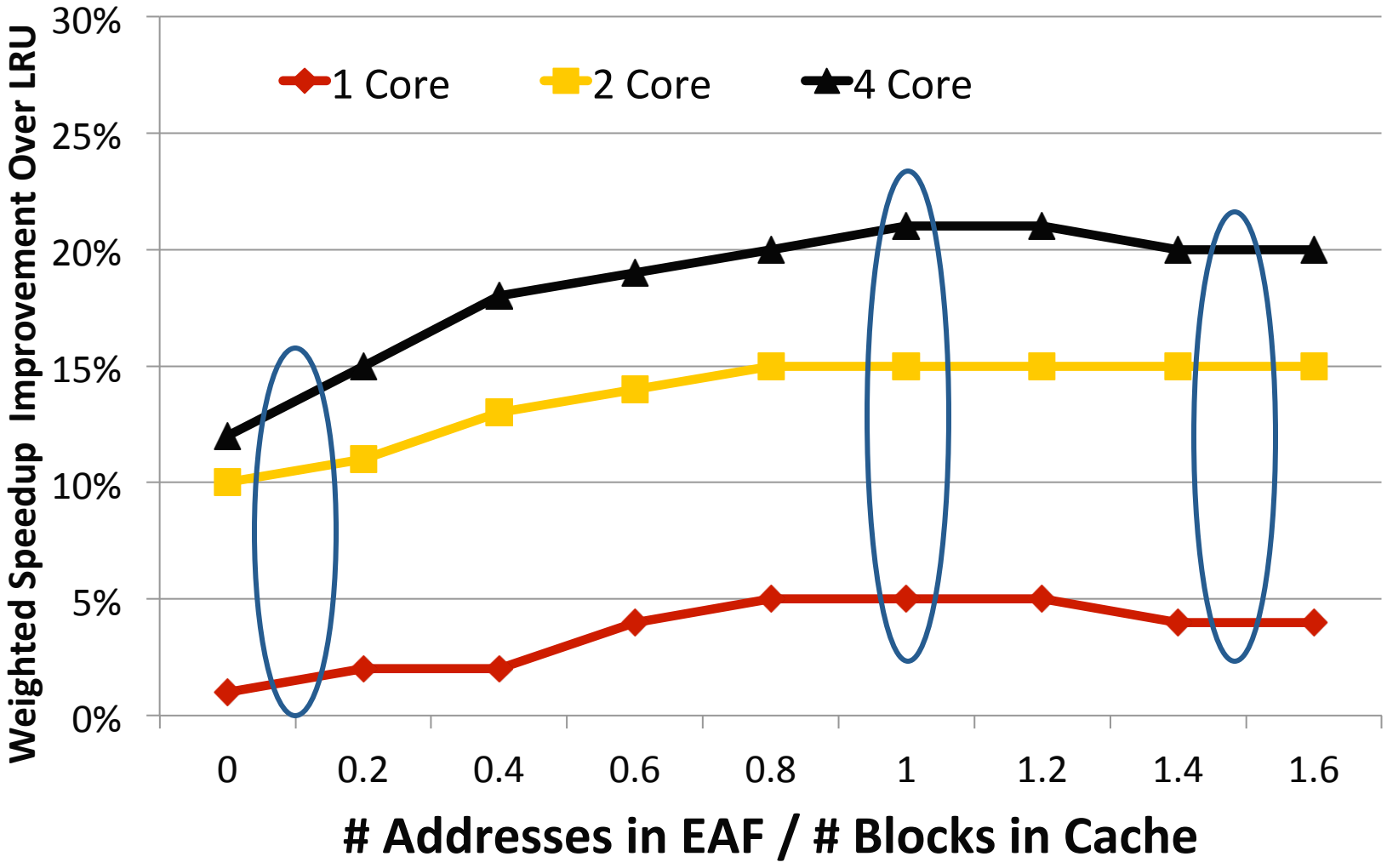
4-Core: Performance



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

Cache Potpourri: Managing Waste

Onur Mutlu
onur@cmu.edu
July 9, 2013
INRIA

Carnegie Mellon

SAFARI

Additional Material

Main Memory Compression

- 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.

Caching for Hybrid Memories

- Justin Meza, Jichuan Chang, HanBin Yoon, Onur Mutlu, and Parthasarathy Ranganathan,
  "Enabling Efficient and Scalable Hybrid Memories Using Fine-Granularity DRAM Cache Management"
  IEEE Computer Architecture Letters (***CAL***), February 2012.

- HanBin Yoon, Justin Meza, Rachata Ausavarungnirun, Rachael Harding, and Onur Mutlu,
  "Row Buffer Locality Aware Caching Policies for Hybrid Memories"
  Proceedings of the
  30th IEEE International Conference on Computer Design (***ICCD***),
  Montreal, Quebec, Canada, September 2012. Slides (pptx) (pdf)
  Best paper award (in Computer Systems and Applications track).

Four Works on Memory Interference (I)

- Eiman Ebrahimi, Chang Joo Lee, Onur Mutlu, and Yale N. Patt,
  "Fairness via Source Throttling: A Configurable and High-Performance Fairness Substrate for Multi-Core Memory Systems"
  *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems (**ASPLOS**)*, pages 335-346, Pittsburgh, PA, March 2010. Slides (pdf)

- Sai Prashanth Muralidhara, Lavanya Subramanian, Onur Mutlu, Mahmut Kandemir, and Thomas Moscibroda,
  "Reducing Memory Interference in Multicore Systems via Application-Aware Memory Channel Partitioning"
  *Proceedings of the 44th International Symposium on Microarchitecture (**MICRO**)*, Porto Alegre, Brazil, December 2011. Slides (pptx)

Four Works on Memory Interference (II)

- Reetuparna Das, Rachata Ausavarungnirun, Onur Mutlu, Akhilesh Kumar, and Mani Azimi,
  "Application-to-Core Mapping Policies to Reduce Memory System Interference in Multi-Core Systems"
  *Proceedings of the 19th International Symposium on High-Performance Computer Architecture (**HPCA**)*, Shenzhen, China, February 2013. Slides (pptx)

- Eiman Ebrahimi, Rustam Miftakhutdinov, Chris Fallin, Chang Joo Lee, Onur Mutlu, and Yale N. Patt,
  "Parallel Application Memory Scheduling"
  *Proceedings of the 44th International Symposium on Microarchitecture (**MICRO**)*, Porto Alegre, Brazil, December 2011. Slides (pptx)

Enabling Emerging Memory Technologies

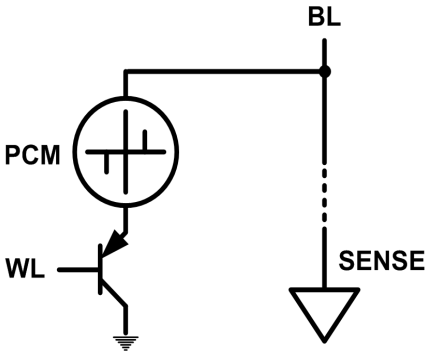
Aside: Scaling Flash Memory [Cai+, ICCD'12]

- NAND flash memory has low endurance: a flash cell dies after 3k P/E cycles vs. 50k desired → Major scaling challenge for flash memory
- Flash error rate increases exponentially over flash lifetime
- Problem: Stronger error correction codes (ECC) are ineffective and undesirable for improving flash lifetime due to
  - diminishing returns on lifetime with increased correction strength
  - prohibitively high power, area, latency overheads
- Our Goal: Develop techniques to tolerate high error rates w/o strong ECC
- Observation: Retention errors are the dominant errors in MLC NAND flash
  - flash cell loses charge over time; retention errors increase as cell gets worn out
- Solution: Flash Correct-and-Refresh (FCR)
  - Periodically read, correct, and reprogram (in place) or remap each flash page before it accumulates more errors than can be corrected by simple ECC
  - Adapt "refresh" rate to the severity of retention errors (i.e., # of P/E cycles)
- Results: FCR improves flash memory lifetime by 46X with no hardware changes and low energy overhead; outperforms strong ECCs

Solution 2: Emerging Memory Technologies

- Some emerging resistive memory technologies seem more scalable than DRAM (and they are non-volatile)

- Example: Phase Change Memory
  - Data stored by changing phase of material
  - Data read by detecting material's resistance
  - Expected to scale to 9nm (2022 [ITRS])
  - Prototyped at 20nm (Raoux+, IBM JRD 2008)
  - Expected to be denser than DRAM: can store multiple bits/cell



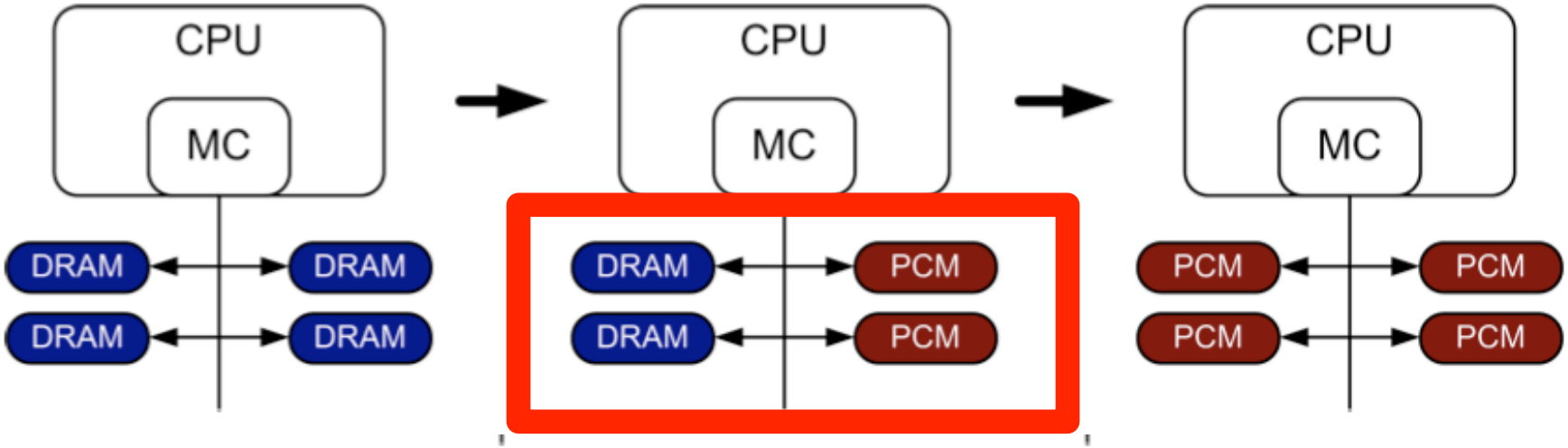
- But, emerging technologies have (many) shortcomings
  - Can they be enabled to replace/augment/surpass DRAM?

Phase Change Memory: Pros and Cons

- Pros over DRAM
  - Better technology scaling (capacity and cost)
  - Non volatility
  - Low idle power (no refresh)
- Cons
  - Higher latencies: ~4-15x DRAM (especially write)
  - Higher active energy: ~2-50x DRAM (especially write)
  - Lower endurance (a cell dies after ~10^8 writes)
- Challenges in enabling PCM as DRAM replacement/helper:
  - Mitigate PCM shortcomings
  - Find the right way to place PCM in the system

PCM-based Main Memory (I)

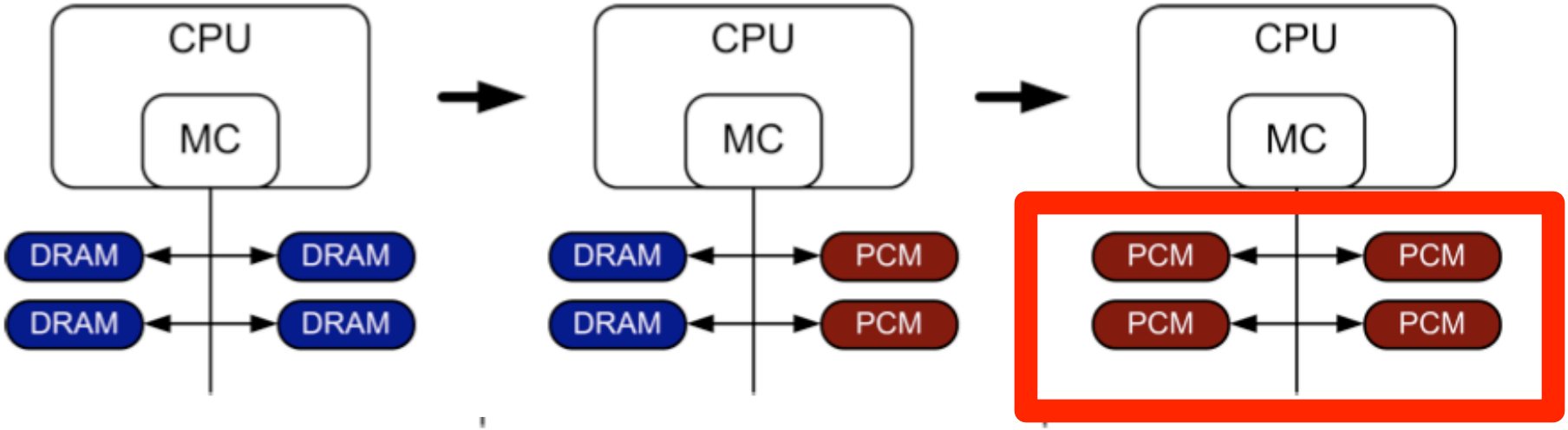
- How should PCM-based (main) memory be organized?



- Hybrid PCM+DRAM [Qureshi+ ISCA'09, Dhiman+ DAC'09]:
  - How to partition/migrate data between PCM and DRAM

PCM-based Main Memory (II)

- How should PCM-based (main) memory be organized?



- Pure PCM main memory [Lee et al., ISCA'09, Top Picks'10]:
  - How to redesign entire hierarchy (and cores) to overcome PCM shortcomings

PCM-Based Memory Systems: Research Challenges

- Partitioning
  - Should DRAM be a cache or main memory, or configurable?
  - What fraction? How many controllers?
- Data allocation/movement (energy, performance, lifetime)
  - Who manages allocation/movement?
  - What are good control algorithms?
  - How do we prevent degradation of service due to wearout?
- Design of cache hierarchy, memory controllers, OS
  - Mitigate PCM shortcomings, exploit PCM advantages
- Design of PCM/DRAM chips and modules
  - Rethink the design of PCM/DRAM with new requirements

An Initial Study: Replace DRAM with PCM

- Lee, Ipek, Mutlu, Burger, "Architecting Phase Change Memory as a Scalable DRAM Alternative," ISCA 2009.
  - Surveyed prototypes from 2003-2008 (e.g. IEDM, VLSI, ISSCC)
  - Derived "average" PCM parameters for F=90nm

Density

- 9 - 12F^2 using BJT
- 1.5× DRAM

Latency

- 50ns Rd, 150ns Wr
- 4×, 12× DRAM

Endurance

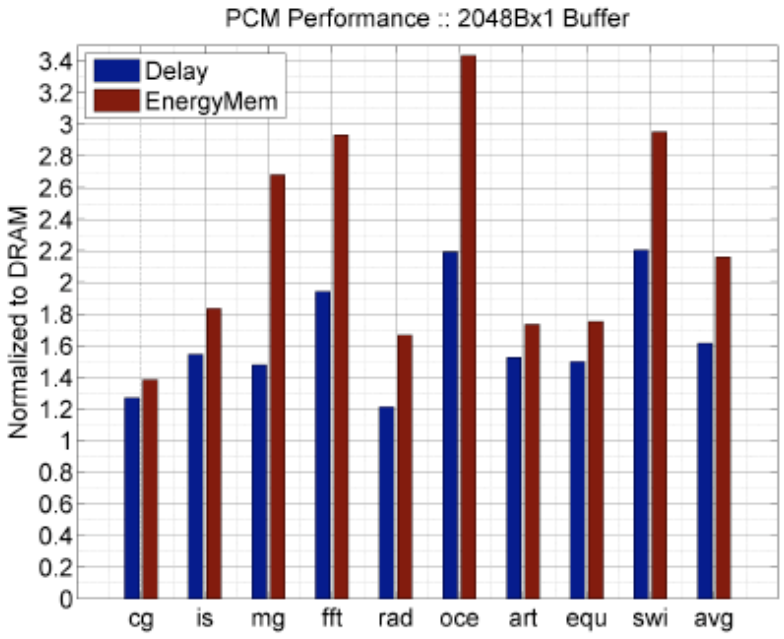
- 1E+08 writes
- 1E-08× DRAM

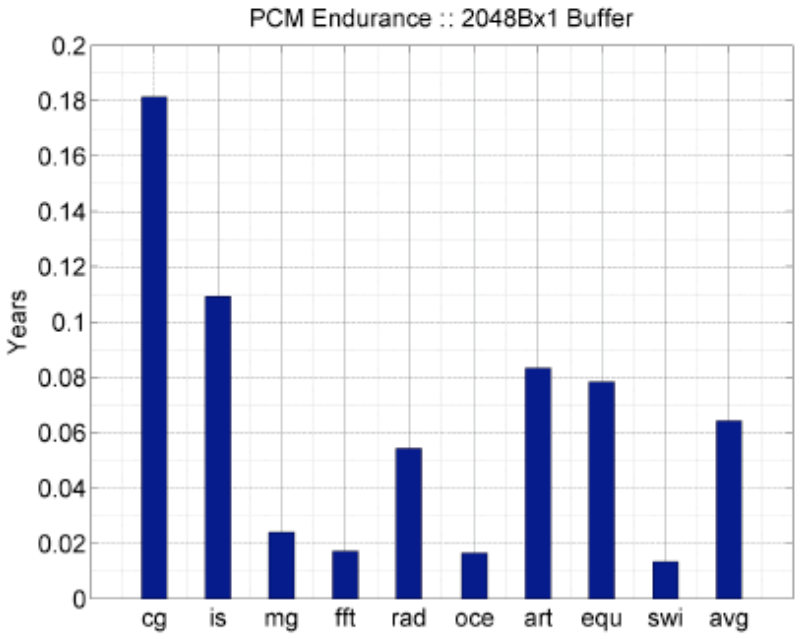
Energy

- 40μA Rd, 150μA Wr
- 2×, 43× DRAM

Results: Naïve Replacement of DRAM with PCM

- Replace DRAM with PCM in a 4-core, 4MB L2 system
- PCM organized the same as DRAM: row buffers, banks, peripherals
- 1.6x delay, 2.2x energy, 500-hour average lifetime



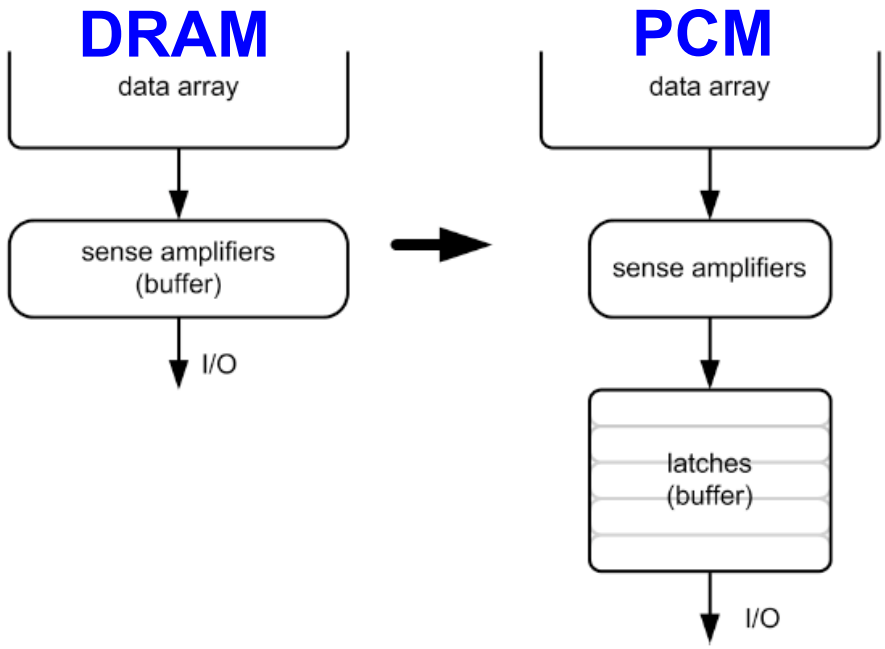


- Lee, Ipek, Mutlu, Burger, "Architecting Phase Change Memory as a Scalable DRAM Alternative," ISCA 2009.

Architecting PCM to Mitigate Shortcomings

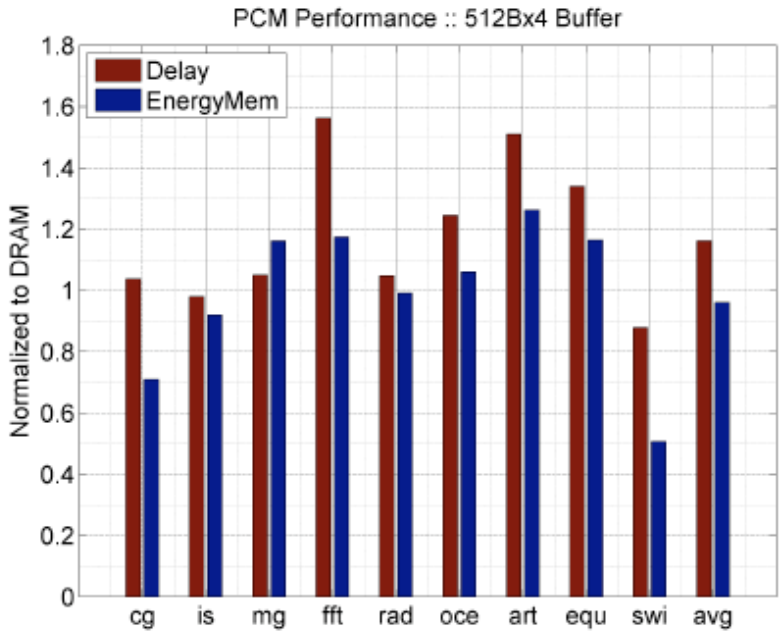
- Idea 1: Use multiple narrow row buffers in each PCM chip
  → Reduces array reads/writes → better endurance, latency, energy

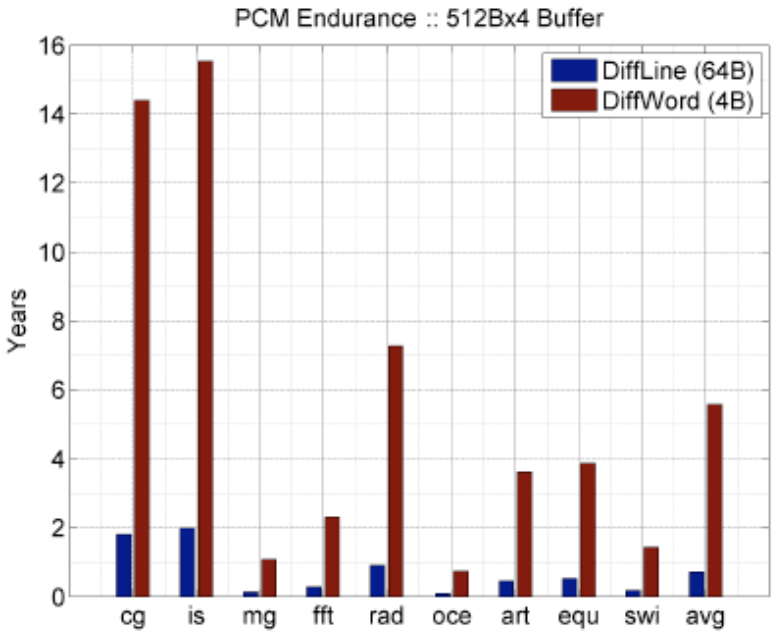
- Idea 2: Write into array at cache block or word granularity
  → Reduces unnecessary wear



Results: Architected PCM as Main Memory

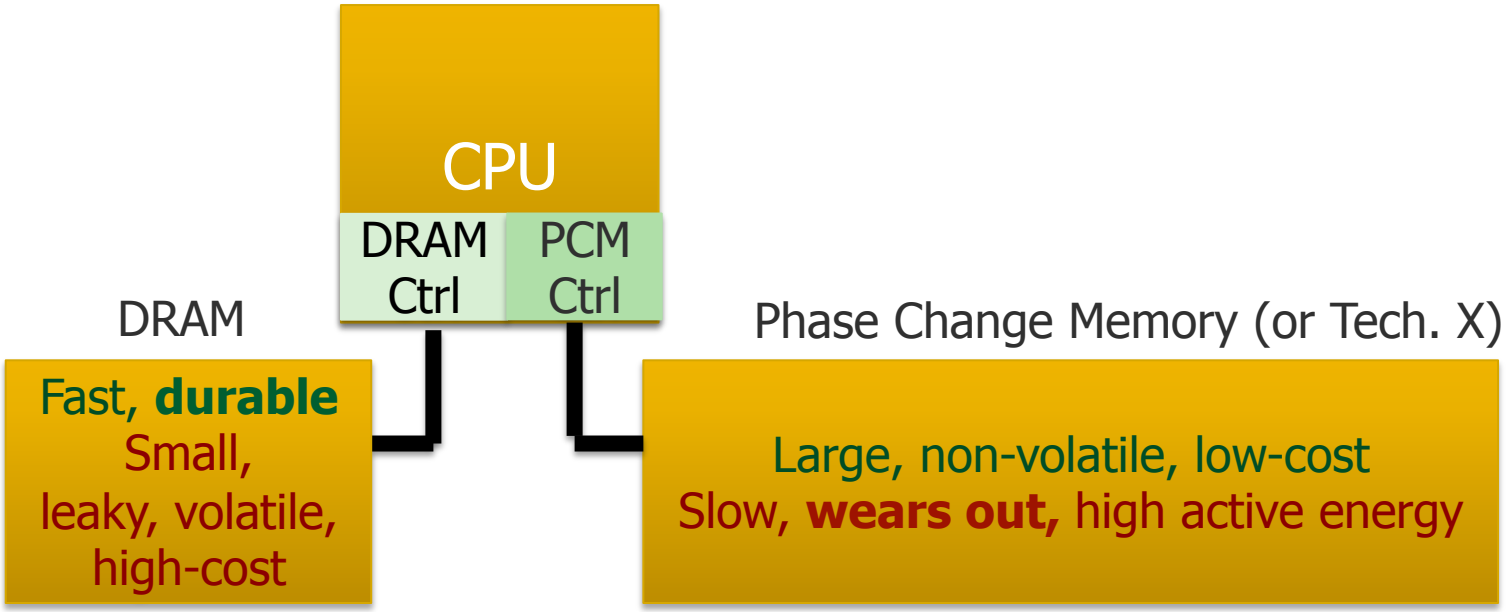
- 1.2x delay, 1.0x energy, 5.6-year average lifetime
- Scaling improves energy, endurance, density





- Caveat 1: Worst-case lifetime is much shorter (no guarantees)
- Caveat 2: Intensive applications see large performance and energy hits
- Caveat 3: Optimistic PCM parameters?

Hybrid Memory Systems



Hardware/software manage data allocation and movement
to achieve the best of multiple technologies
(5-9 years of average lifetime)

Meza, Chang, Yoon, Mutlu, Ranganathan, "Enabling Efficient and Scalable Hybrid Memories," IEEE Comp. Arch. Letters, 2012.

One Option: DRAM as a Cache for PCM

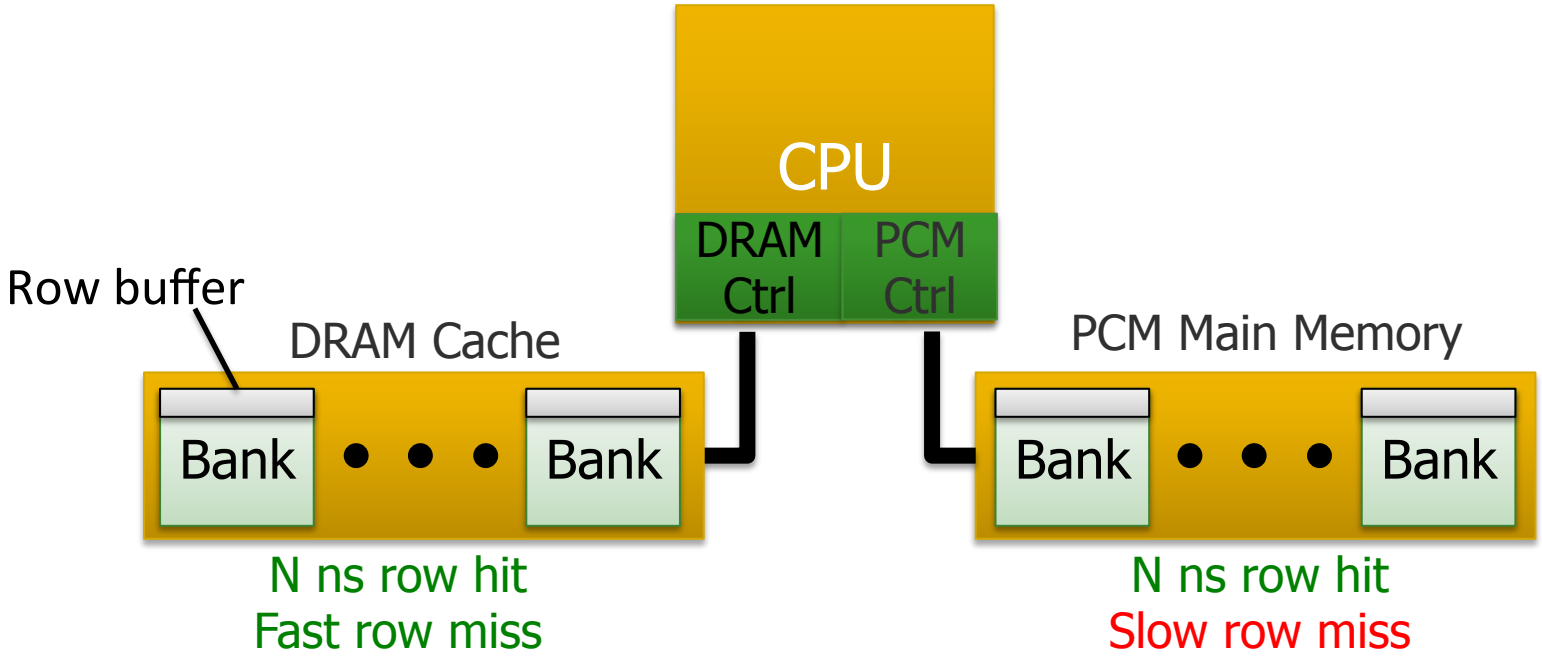
- PCM is main memory; DRAM caches memory rows/blocks
  - Benefits: Reduced latency on DRAM cache hit; write filtering
- Memory controller hardware manages the DRAM cache
  - Benefit: Eliminates system software overhead

- Three issues:
  - What data should be placed in DRAM versus kept in PCM?
  - What is the granularity of data movement?
  - How to design a low-cost hardware-managed DRAM cache?

- Two idea directions:
  - Locality-aware data placement **[Yoon+ , ICCD 2012]**
  - Cheap tag stores and dynamic granularity **[Meza+, IEEE CAL 2012]**

DRAM vs. PCM: An Observation

- Row buffers are the same in DRAM and PCM
- Row buffer hit latency **same** in DRAM and PCM
- Row buffer miss latency **small** in DRAM, **large** in PCM



- Accessing the row buffer in PCM is fast
- What incurs high latency is the PCM array access → avoid this

Row-Locality-Aware Data Placement

- Idea: Cache in DRAM only those rows that
  - Frequently cause row buffer conflicts → because row-conflict latency is smaller in DRAM
  - Are reused many times → to reduce cache pollution and bandwidth waste

- Simplified rule of thumb:
  - Streaming accesses: Better to place in PCM
  - Other accesses (with some reuse): Better to place in DRAM

- Bridges half of the performance gap between all-DRAM and all-PCM memory on memory-intensive workloads

- Yoon et al., "Row Buffer Locality-Aware Caching Policies for Hybrid Memories," ICCD 2012.

Row-Locality-Aware Data Placement: Mechanism

- For a subset of rows in PCM, memory controller:
  - Tracks **row conflicts** as a predictor of future locality
  - Tracks **accesses** as a predictor of future reuse

- Cache a row in DRAM if its row conflict and access counts are greater than certain thresholds

- Determine thresholds dynamically to adjust to application/ workload characteristics
  - Simple cost/benefit analysis every fixed interval

Implementation: "Statistics Store"

- Goal: To keep count of row buffer misses to recently used rows in PCM

- Hardware structure in memory controller
  - Operation is similar to a cache
    - Input: row address
    - Output: row buffer miss count
  - 128-set 16-way statistics store (9.25KB) achieves system performance within 0.3% of an unlimited-sized statistics store

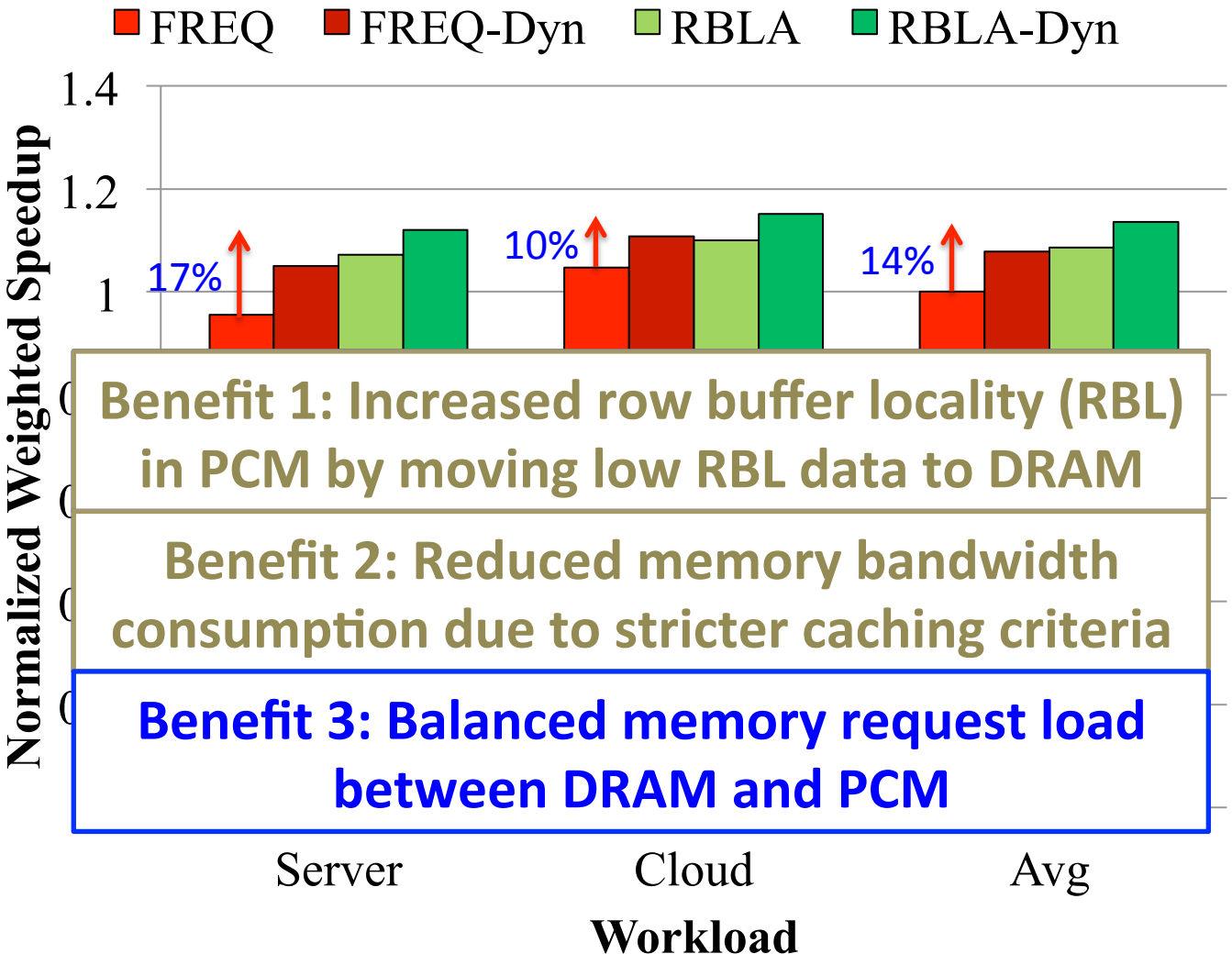
Evaluation Methodology

- Cycle-level x86 CPU-memory simulator
  - **CPU**: 16 out-of-order cores, 32KB private L1 per core, 512KB shared L2 per core
  - **Memory**: 1GB DRAM (8 banks), 16GB PCM (8 banks), 4KB migration granularity
- 36 multi-programmed server, cloud workloads
  - Server: TPC-C (OLTP), TPC-H (Decision Support)
  - Cloud: Apache (Webserv.), H.264 (Video), TPC-C/H
- Metrics: Weighted speedup (perf.), perf./Watt (energy eff.), Maximum slowdown (fairness)

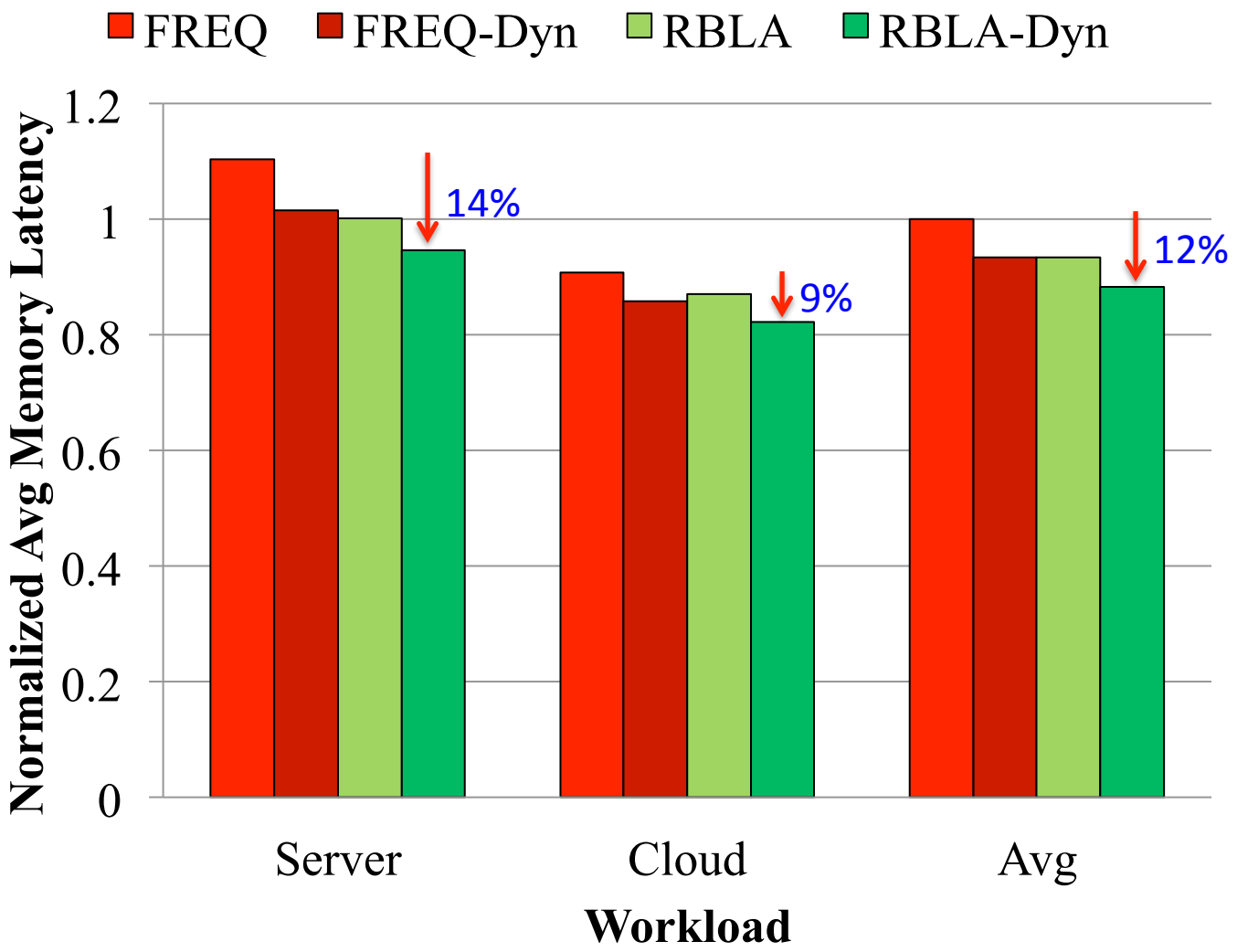
Comparison Points

- **Conventional LRU Caching**
- **FREQ**: Access-frequency-based caching
  - Places "hot data" in cache [Jiang+ HPCA'10]
  - Cache to DRAM rows with accesses ≥ threshold
  - Row buffer locality-*unaware*
- **FREQ-Dyn**: Adaptive Freq.-based caching
  - **FREQ** + our dynamic threshold adjustment
  - Row buffer locality-*unaware*
- **RBLA**: Row buffer locality-aware caching
- **RBLA-Dyn**: Adaptive RBL-aware caching

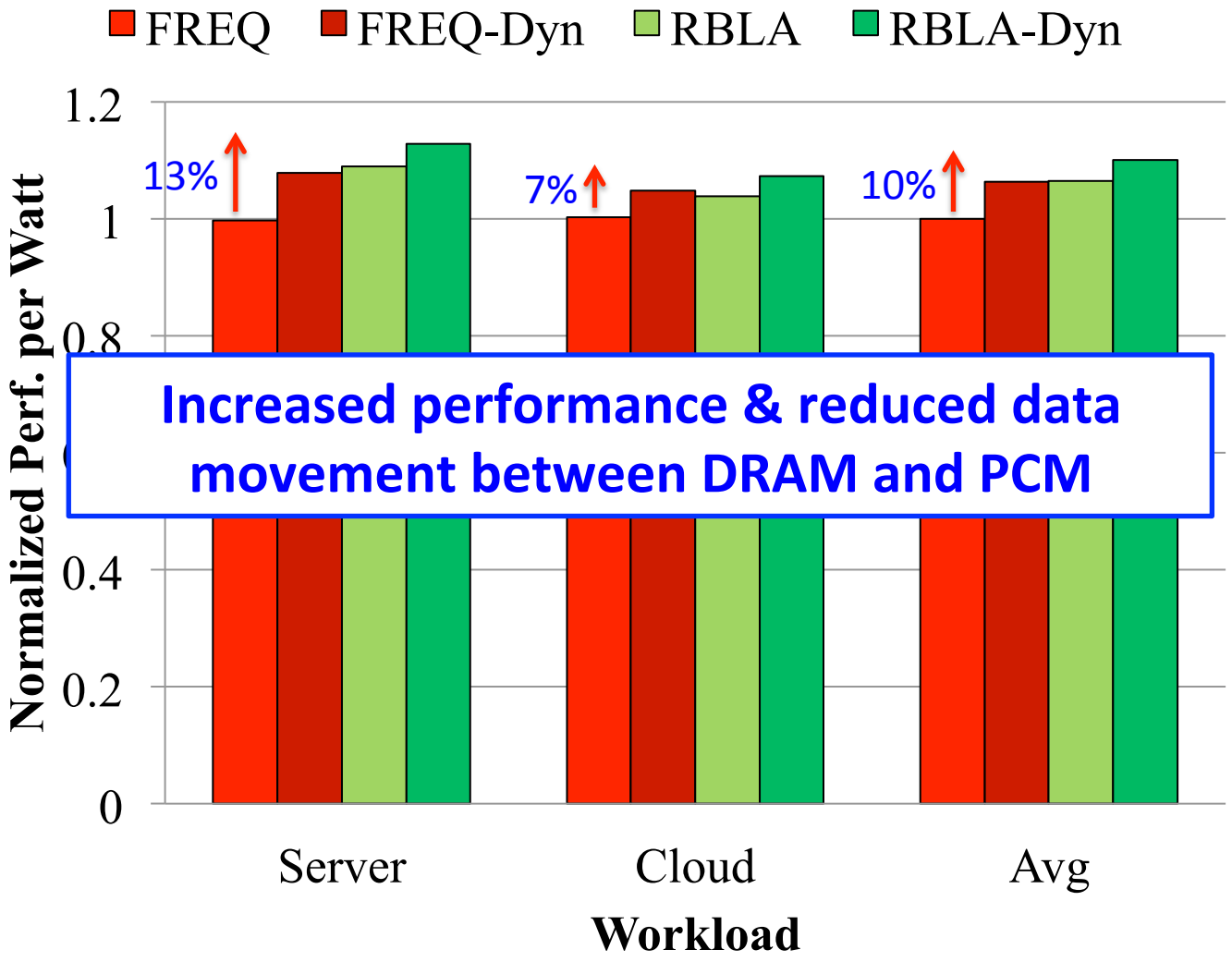
System Performance



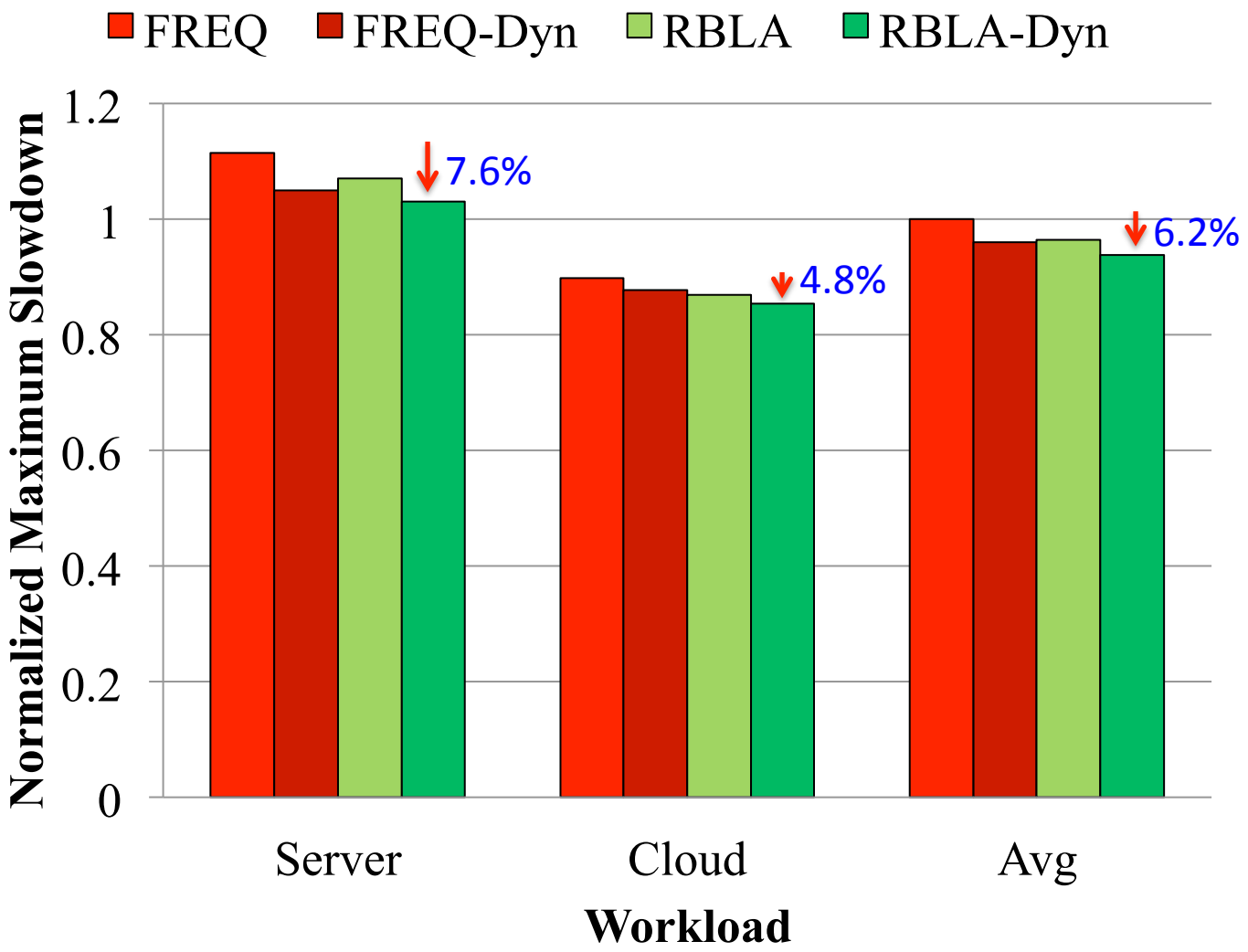
Average Memory Latency



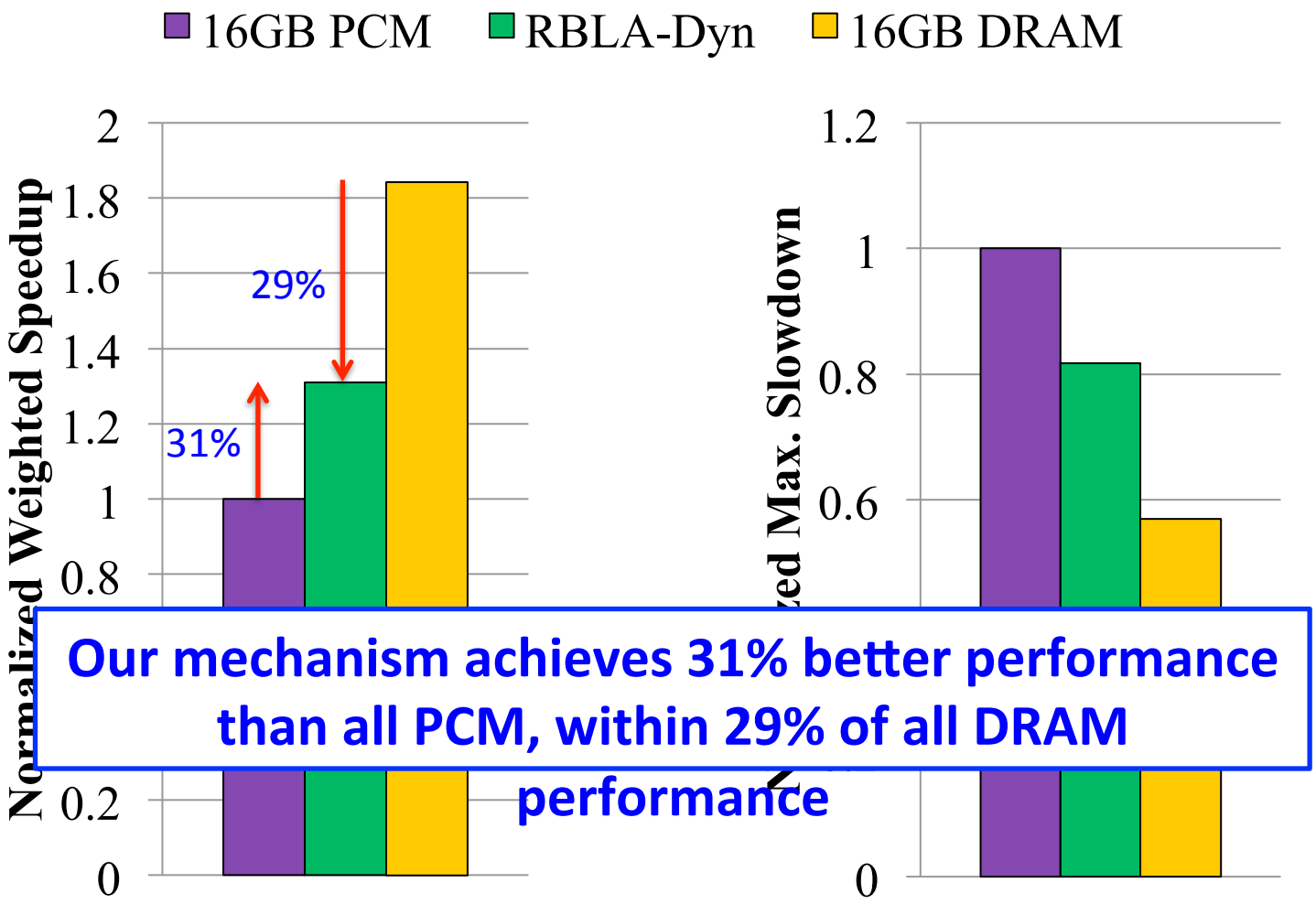
Memory Energy Efficiency



Thread Fairness

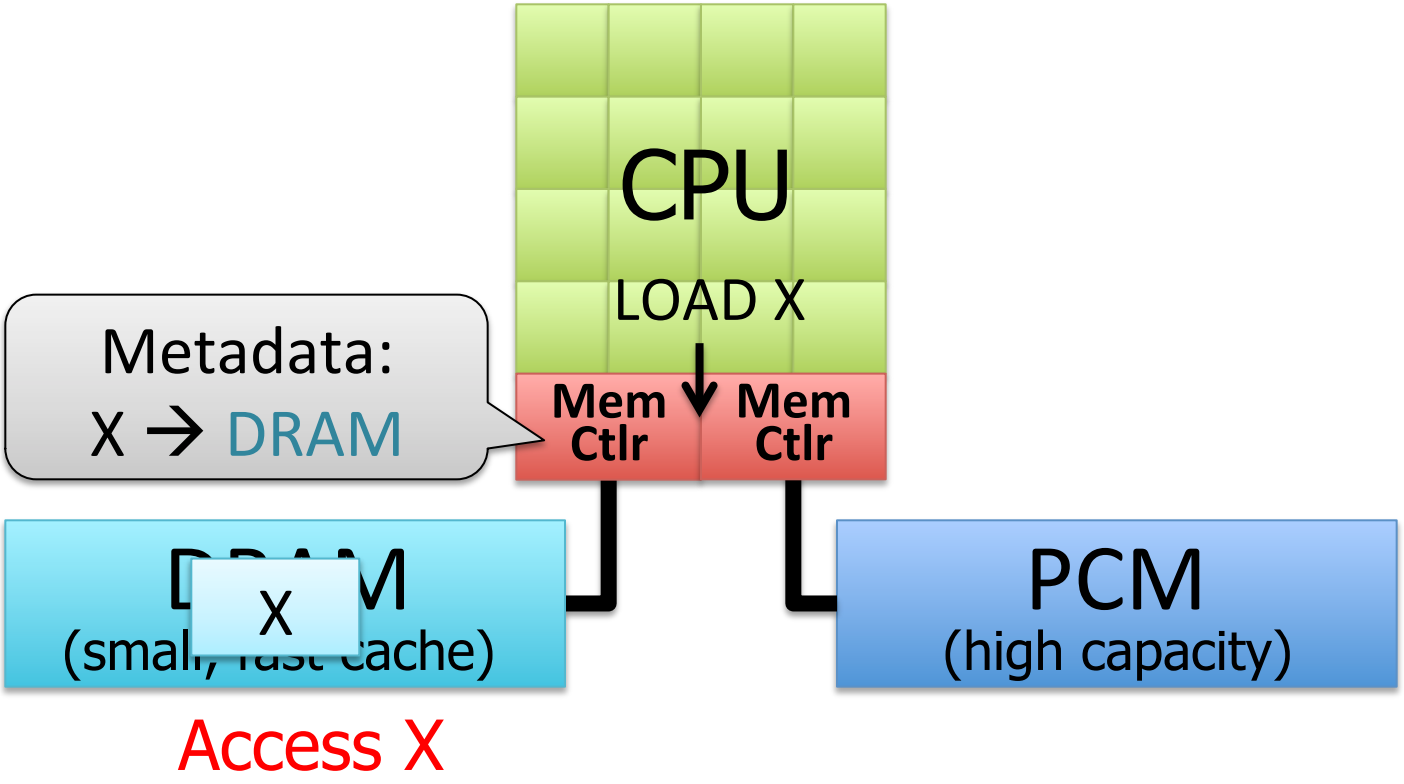


Compared to All-PCM/DRAM



The Problem with Large DRAM Caches

- A large DRAM cache requires a large metadata (tag + block-based information) store
- How do we design an efficient DRAM cache?



Idea 1: Tags in Memory

- Store tags in the same row as data in DRAM
  - Store metadata in same row as their data
  - Data and metadata can be accessed together

- Benefit: No on-chip tag storage overhead
- Downsides:
  - Cache hit determined only after a DRAM access
  - Cache hit requires two DRAM accesses

Idea 2: Cache Tags in SRAM

- Recall Idea 1: Store all metadata in DRAM
  - To reduce metadata storage overhead

- Idea 2: Cache in on-chip SRAM frequently-accessed metadata
  - Cache only a small amount to keep SRAM size small

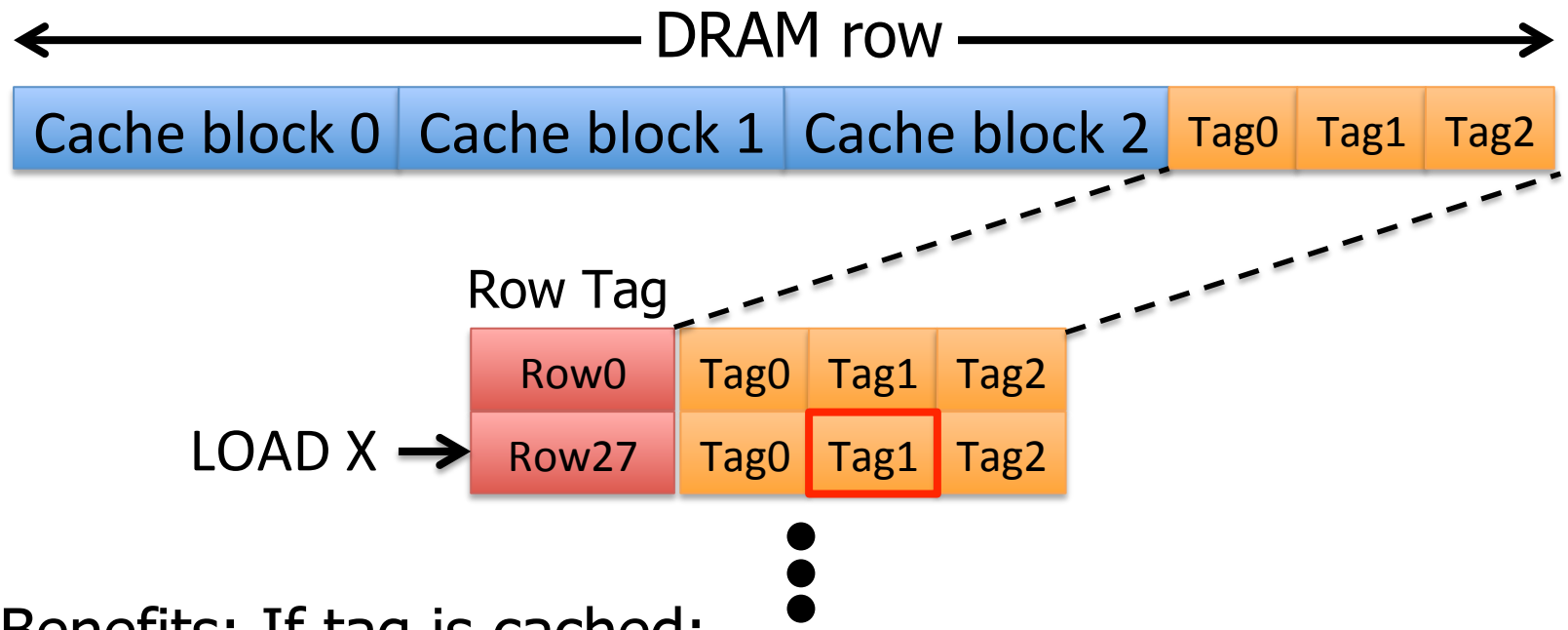
Idea 3: Dynamic Data Transfer Granularity

- Some applications benefit from caching more data
  - They have good spatial locality
- Others do not
  - Large granularity wastes bandwidth and reduces cache utilization

- Idea 3: Simple dynamic caching granularity policy
  - Cost-benefit analysis to determine best DRAM cache block size
  - Group main memory into sets of rows
  - Some row sets follow a fixed caching granularity
  - The rest of main memory follows the best granularity
    - Cost–benefit analysis: access latency versus number of cachings
    - Performed every quantum

TIMBER Tag Management

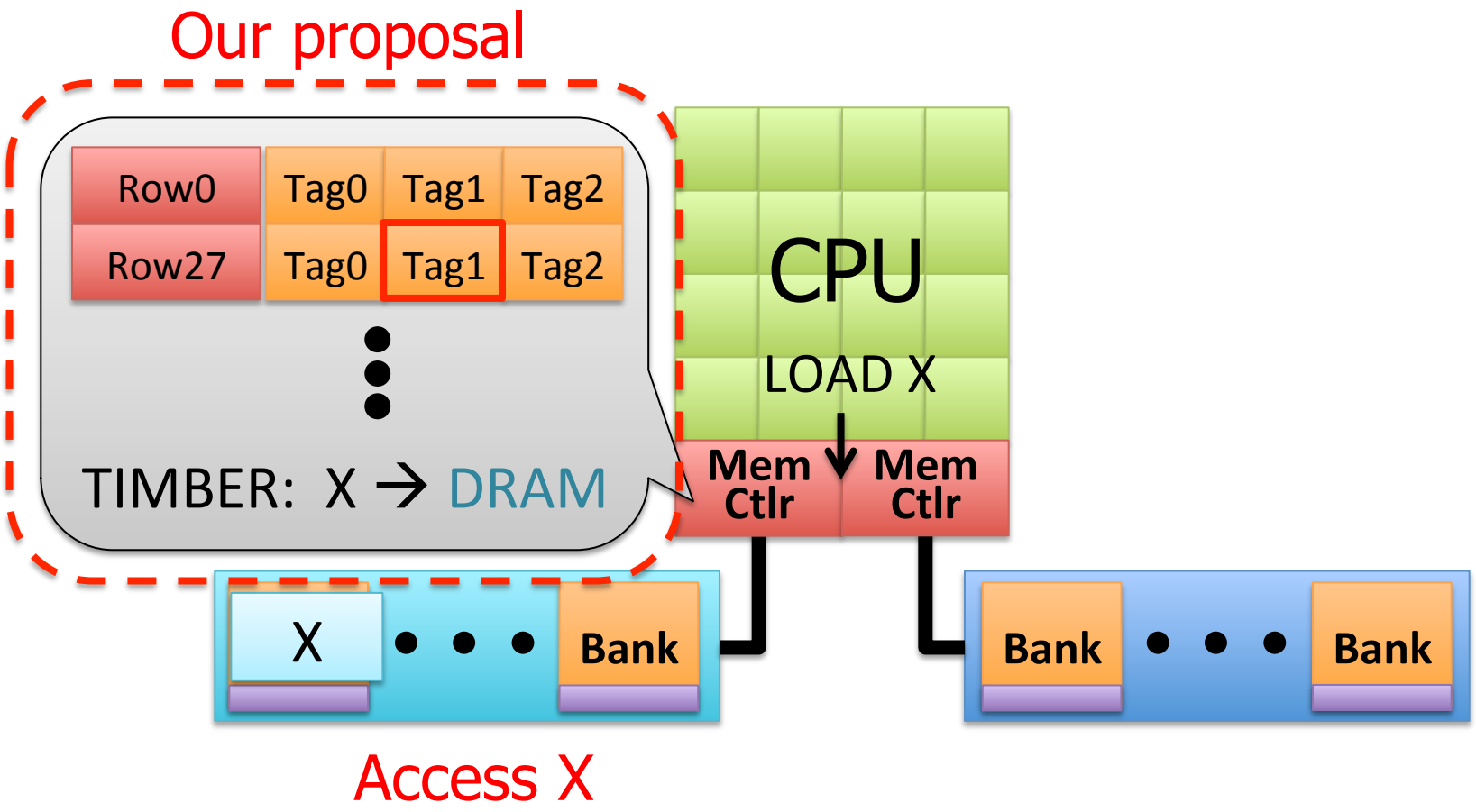
- A Tag-In-Memory BuffER (TIMBER)
  - Stores recently-used tags in a small amount of SRAM



- Benefits: If tag is cached:
  - no need to access DRAM twice
  - cache hit determined quickly

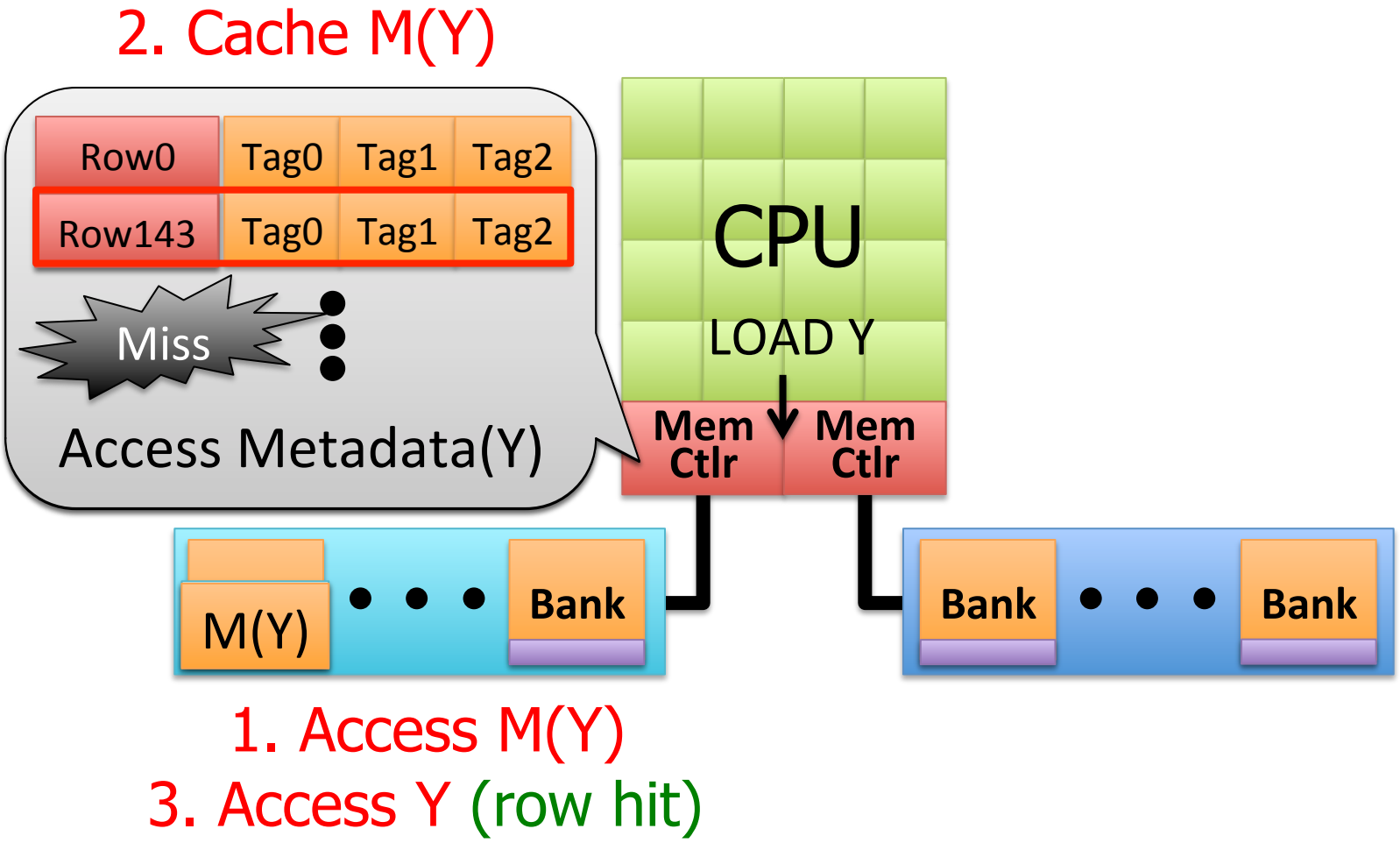
TIMBER Tag Management Example (I)

- Case 1: TIMBER hit



TIMBER Tag Management Example (II)

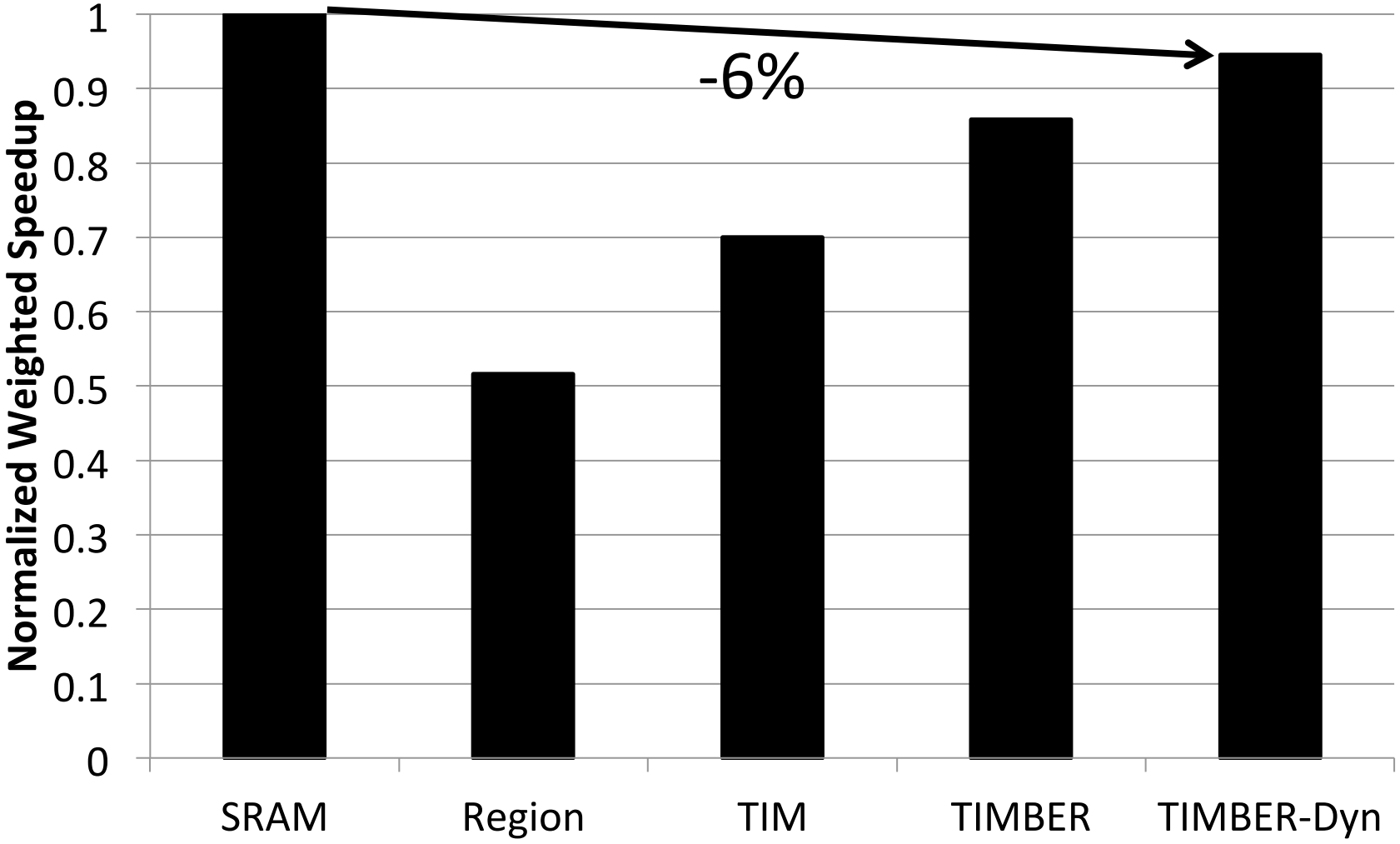
- Case 2: TIMBER miss



Methodology

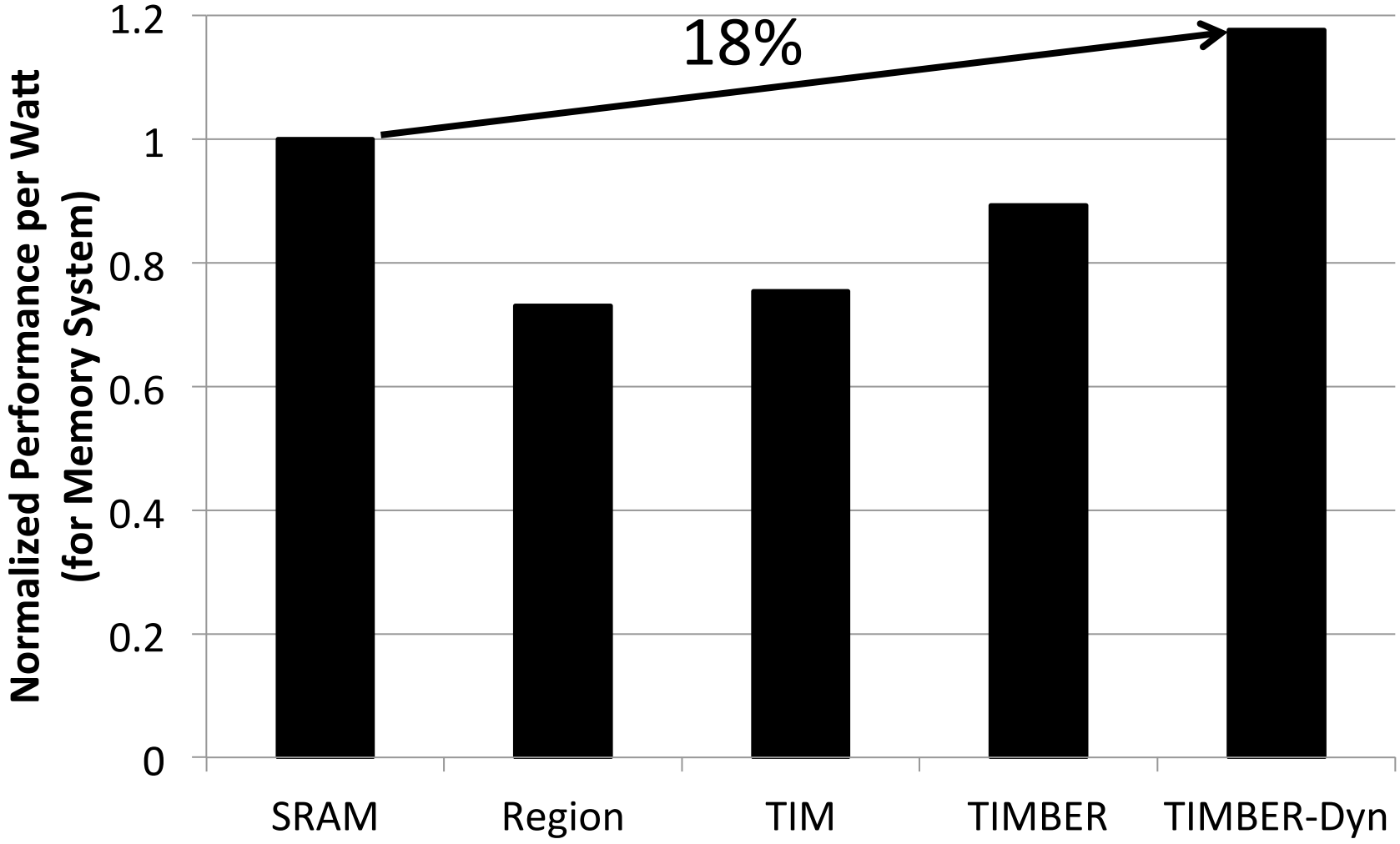
- System: 8 out-of-order cores at 4 GHz
- Memory: 512 MB direct-mapped DRAM, 8 GB PCM
  - 128B caching granularity
  - DRAM row hit (miss): 200 cycles (400 cycles)
  - PCM row hit (clean / dirty miss): 200 cycles (640 / 1840 cycles)
- Evaluated metadata storage techniques
  - All SRAM system (8MB of SRAM)
  - Region metadata storage
  - TIM metadata storage (same row as data)
  - TIMBER, 64-entry direct-mapped (8KB of SRAM)

TIMBER Performance



Meza, Chang, Yoon, Mutlu, Ranganathan, "Enabling Efficient and Scalable Hybrid Memories," IEEE Comp. Arch. Letters, 2012.

TIMBER Energy Efficiency



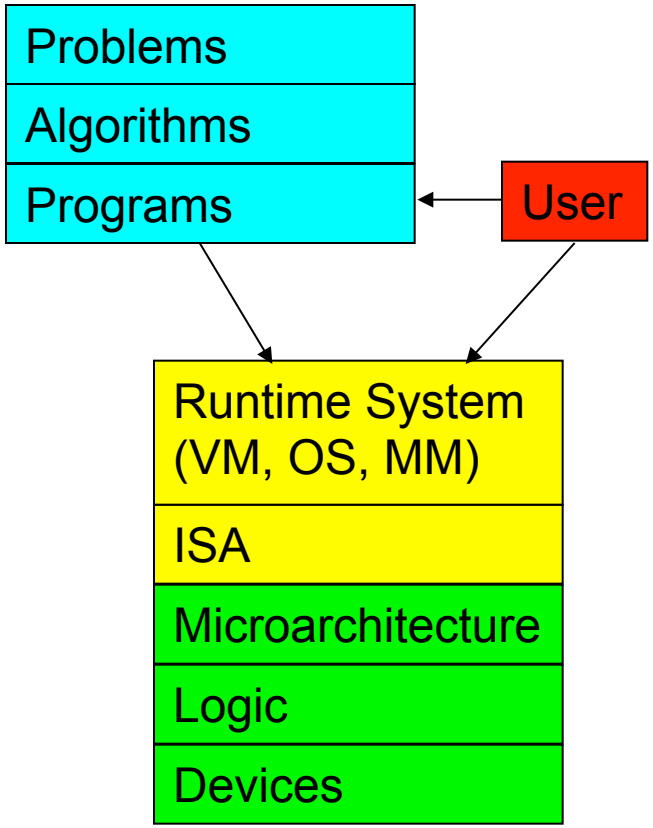
Meza, Chang, Yoon, Mutlu, Ranganathan, "Enabling Efficient and Scalable Hybrid Memories," IEEE Comp. Arch. Letters, 2012.

Hybrid Main Memory: Research Topics

- Many research ideas from technology layer to algorithms layer

- Enabling NVM and hybrid memory
  - How to maximize performance?
  - How to maximize lifetime?
  - How to prevent denial of service?

- Exploiting emerging tecnologies
  - How to exploit non-volatility?
  - How to minimize energy consumption?
  - How to minimize cost?
  - How to exploit NVM on chip?



Security Challenges of Emerging Technologies

1. Limited endurance → Wearout attacks

2. Non-volatility → Data persists in memory after powerdown
   → Easy retrieval of privileged or private information

3. Multiple bits per cell → Information leakage (via side channel)

Securing Emerging Memory Technologies

1. Limited endurance → Wearout attacks
   - Better architecting of memory chips to absorb writes
   - Hybrid memory system management
   - Online wearout attack detection

2. Non-volatility → Data persists in memory after powerdown
   → Easy retrieval of privileged or private information
   - Efficient encryption/decryption of whole main memory
   - Hybrid memory system management

3. Multiple bits per cell → Information leakage (via side channel)
   - System design to hide side channel information

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%**)

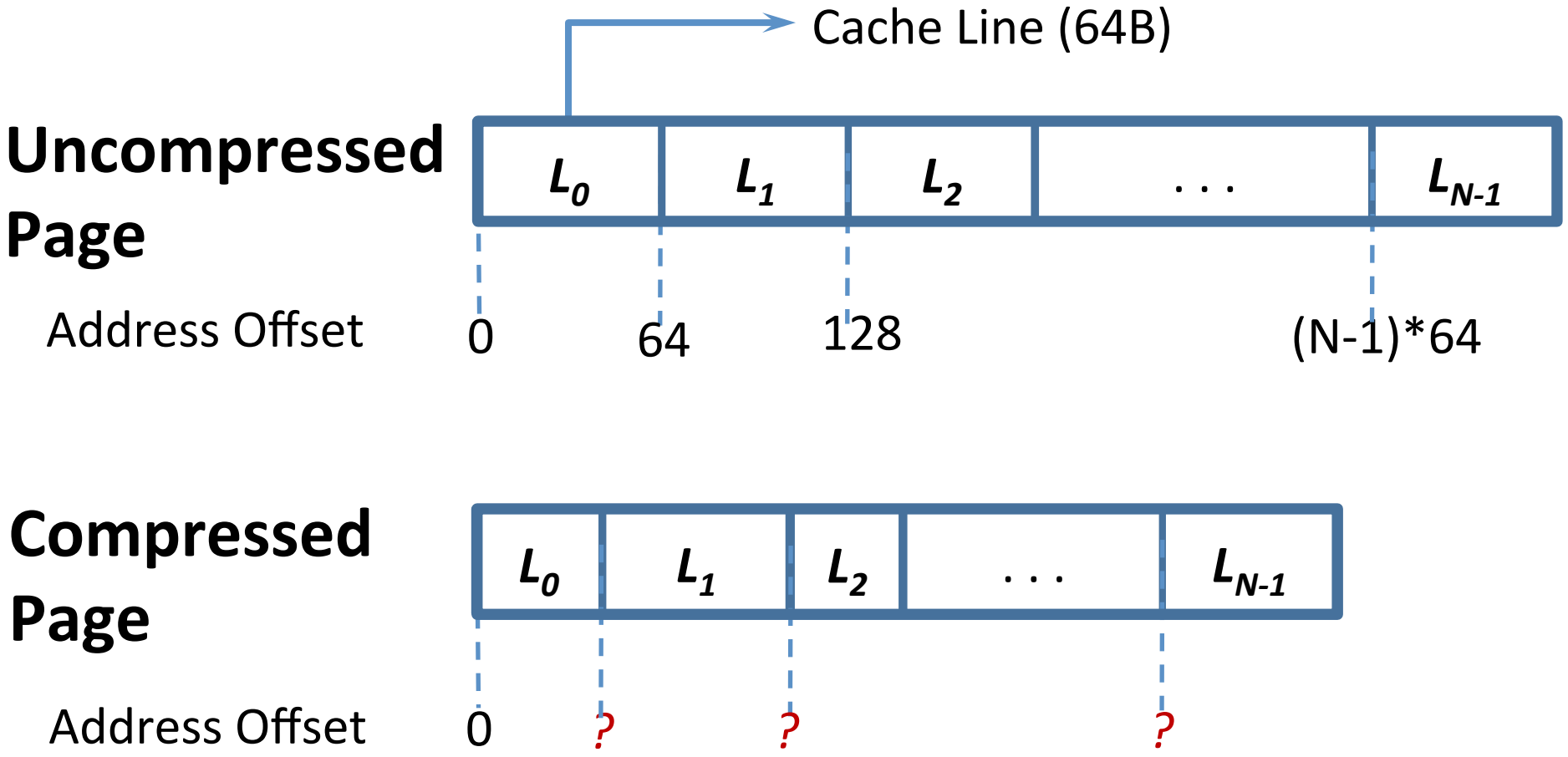
Challenges in Main Memory Compression

1. Address Computation

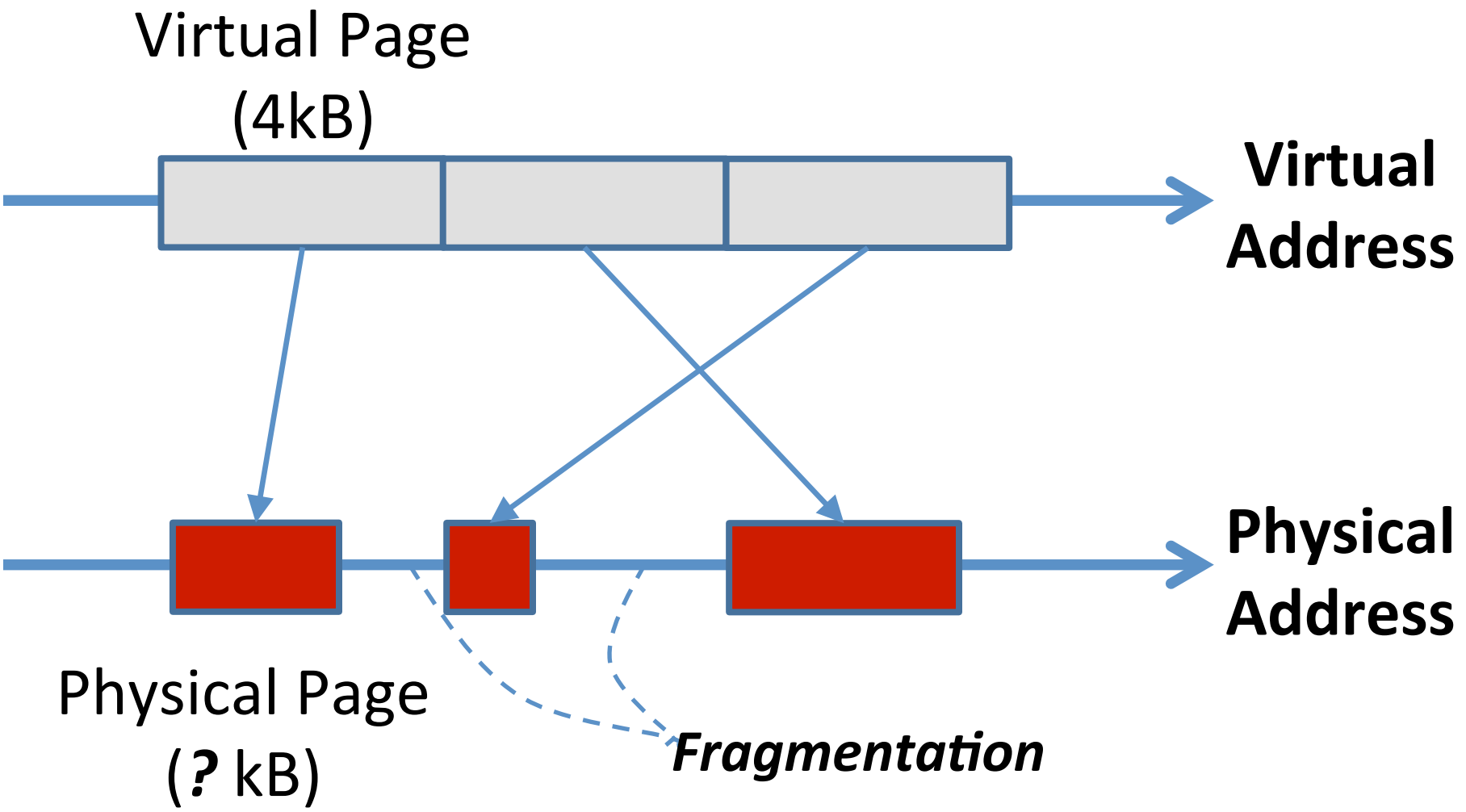
2. Mapping and Fragmentation

3. Physically Tagged Caches

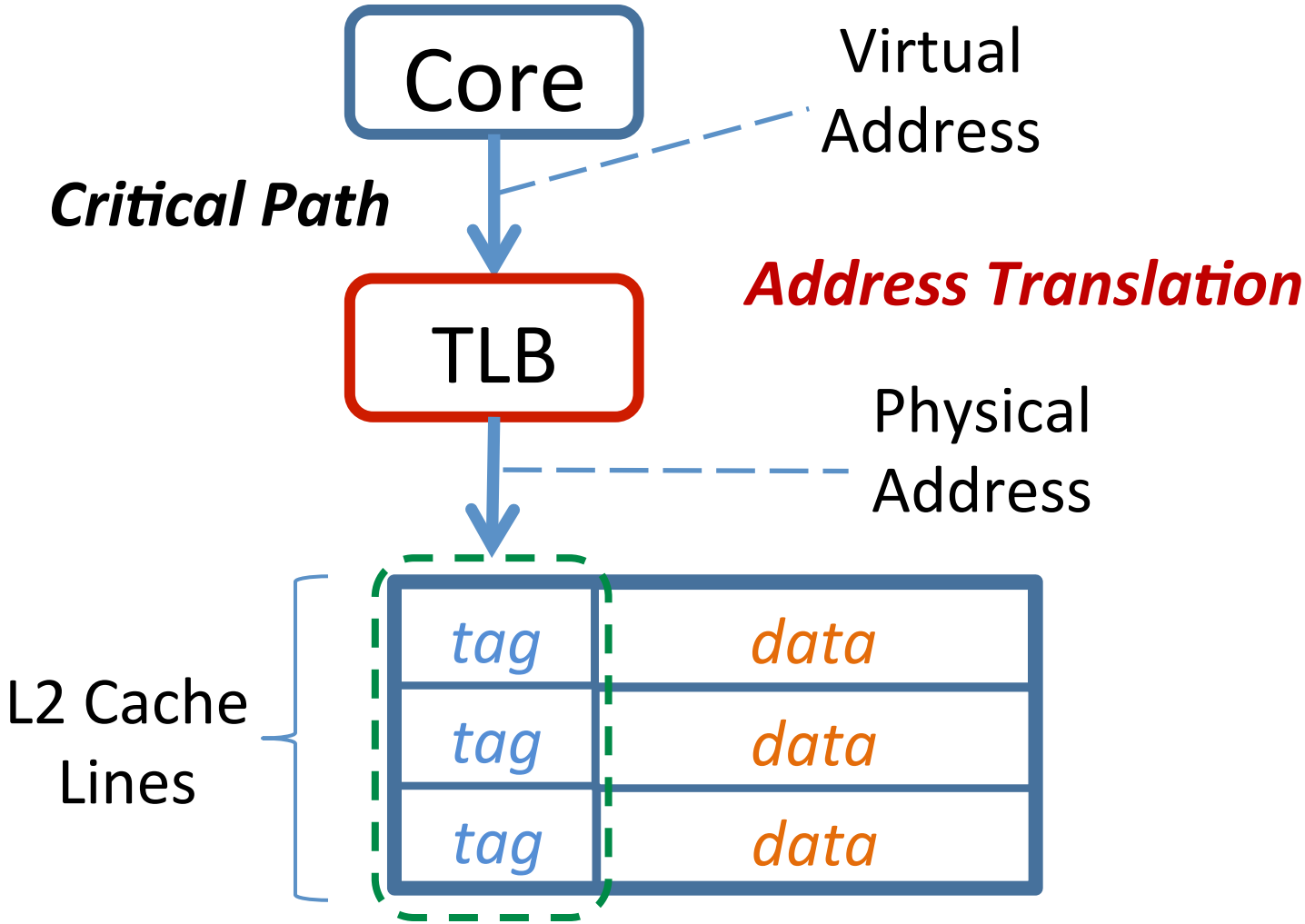
Address Computation



Mapping and Fragmentation



Physically Tagged Caches



Shortcomings of Prior Work

Compression Mechanisms	Access Latency	Decompression Latency	Complexity	Compression Ratio
IBM MXT *[IBM J.R.D. '01]*	✗	✗	✗	✓

Shortcomings of Prior Work

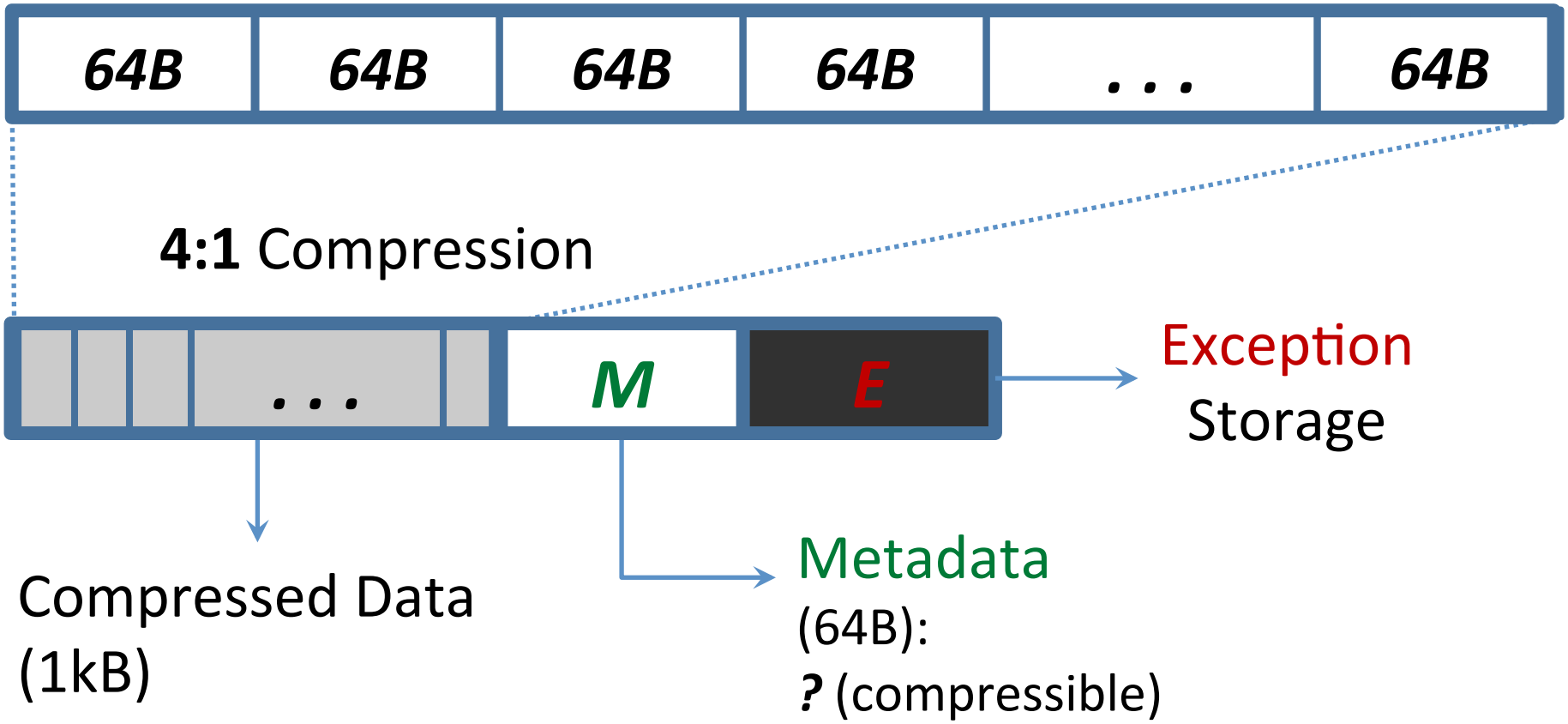
Compression Mechanisms	Access Latency	Decompression Latency	Complexity	Compression Ratio
IBM MXT *[IBM J.R.D. '01]*	✗	✗	✗	✓
Robust Main Memory Compression *[ISCA'05]*	✗	✓	✗	✓

Shortcomings of Prior Work

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

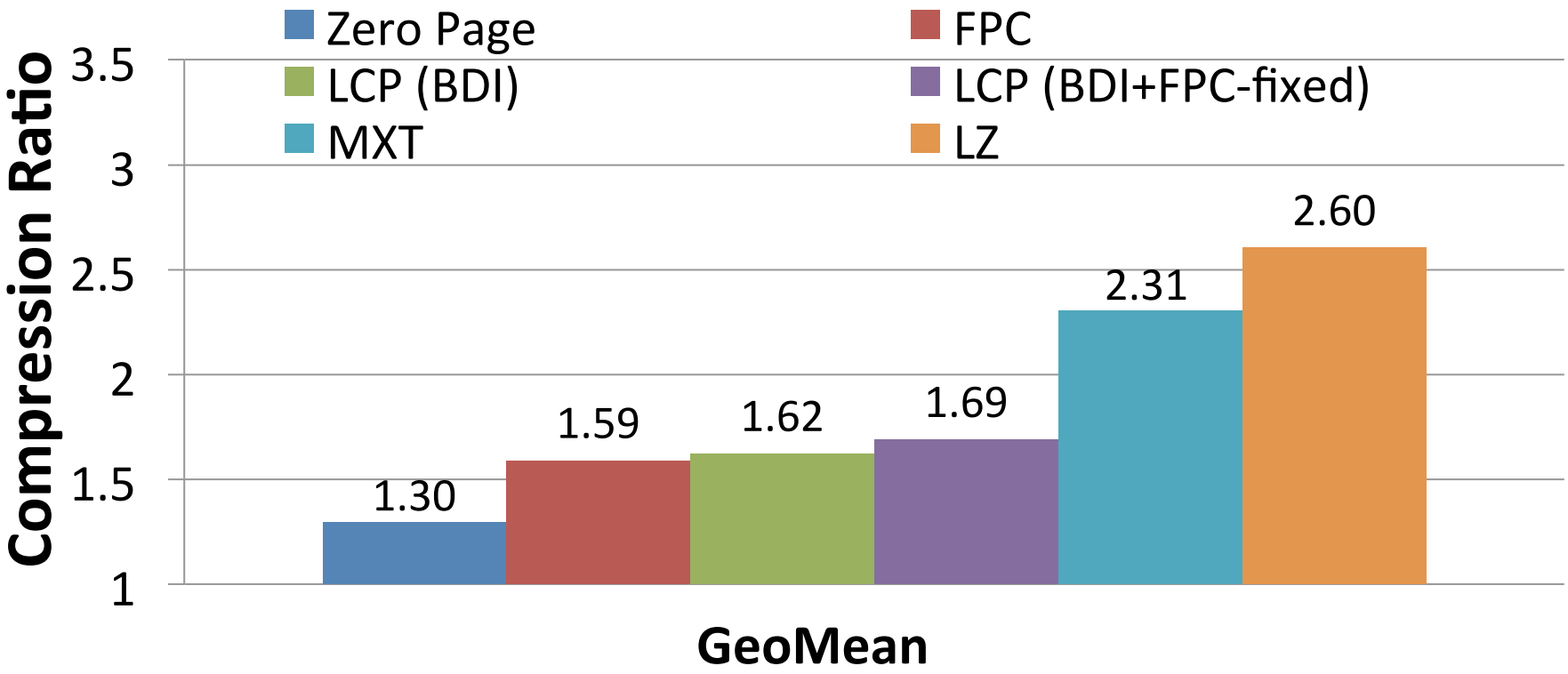
- Zero pages and zero cache lines
  - Handled separately in TLB (1-bit) and in metadata (1-bit per cache line)
- Integration with cache compression
  - BDI and FPC

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

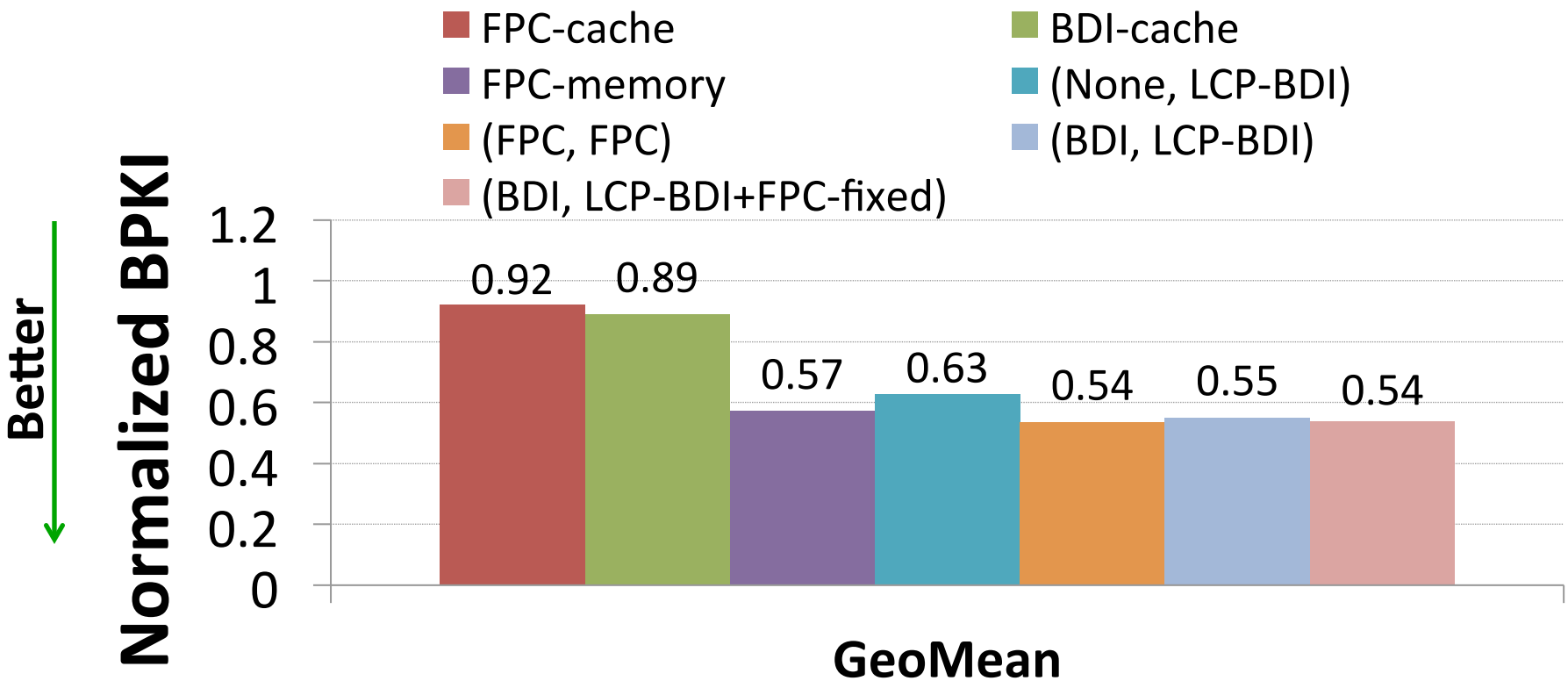
SPEC2006, databases, web workloads, 2MB L2 cache



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**%)

Fairness via Source Throttling

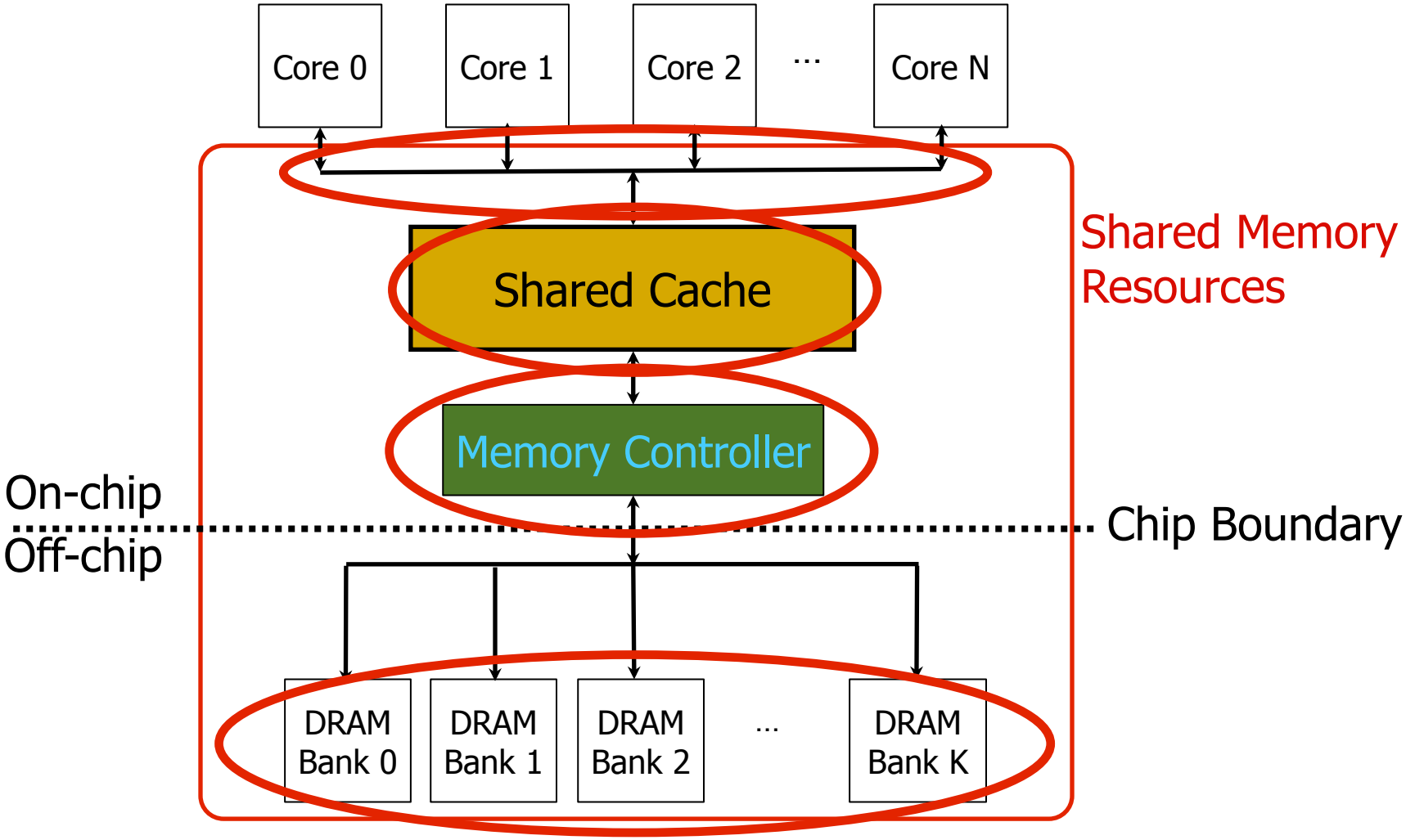
Eiman Ebrahimi, Chang Joo Lee, Onur Mutlu, and Yale N. Patt,

"Fairness via Source Throttling: A Configurable and High-Performance Fairness Substrate for Multi-Core Memory Systems"

15th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (***ASPLOS***), pages 335-346, Pittsburgh, PA, March 2010. Slides (pdf)

FST ASPLOS 2010 Talk

Many Shared Resources



The Problem with "Smart Resources"

- Independent interference control mechanisms in caches, interconnect, and memory can contradict each other

- Explicitly coordinating mechanisms for different resources requires complex implementation

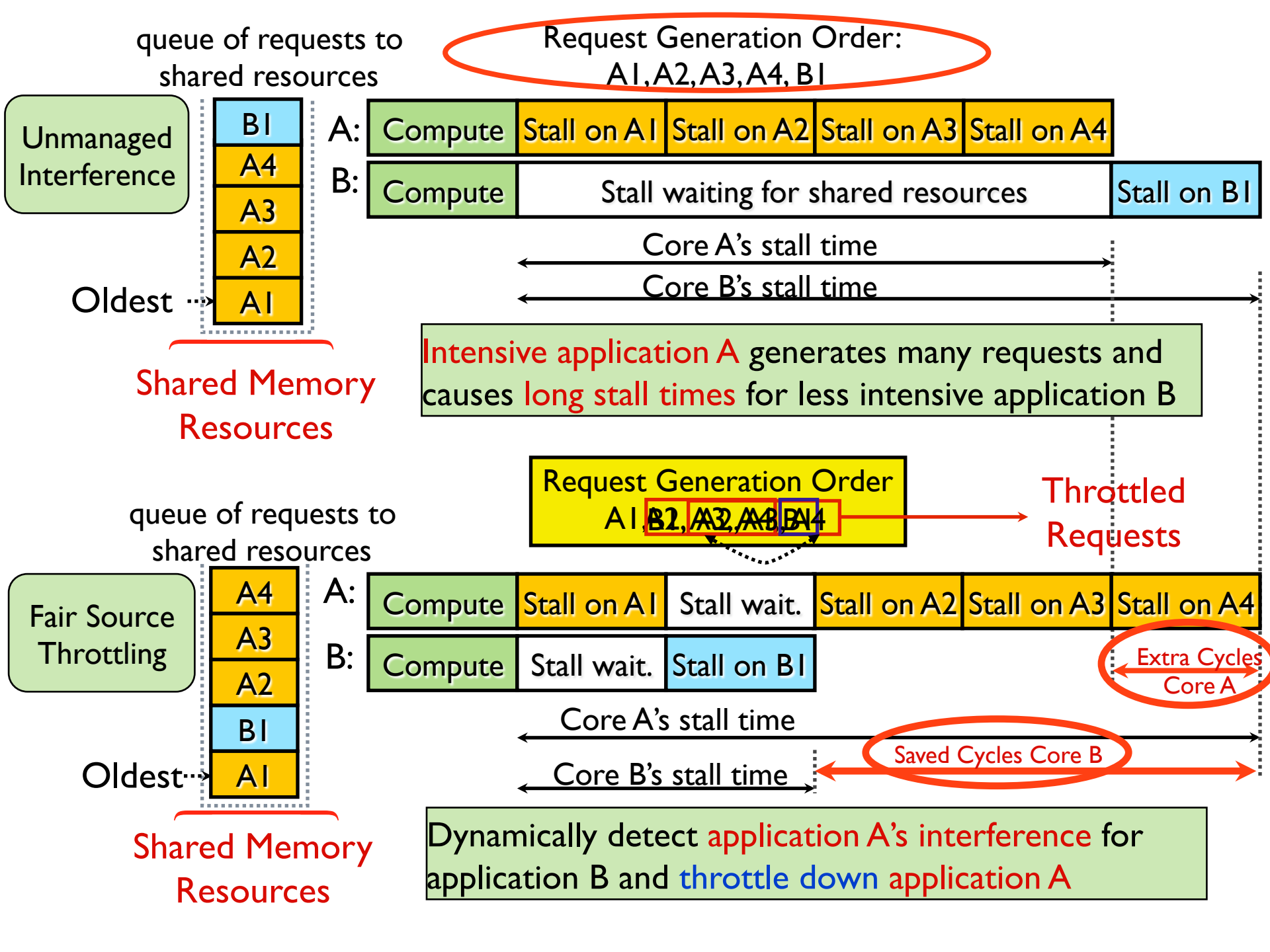
- How do we enable fair sharing of the entire memory system by controlling interference in a coordinated manner?

An Alternative Approach: Source Throttling

- Manage inter-thread interference at the cores, not at the shared resources

- Dynamically estimate unfairness in the memory system
- Feed back this information into a controller
- Throttle cores' memory access rates accordingly
  - Whom to throttle and by how much depends on performance target (throughput, fairness, per-thread QoS, etc)
  - E.g., if unfairness > system-software-specified target then throttle down core causing unfairness & throttle up core that was unfairly treated

- Ebrahimi et al., "Fairness via Source Throttling," ASPLOS'10, TOCS'12.


queue of requests to shared resources
Request Generation Order: A1, A2, A3, A4, B1
Unmanaged Interference
B1
A4
A3
A2
A1
Oldest
A:
Compute
Stall on A1
Stall on A2
Stall on A3
Stall on A4
B:
Compute
Stall waiting for shared resources
Stall on B1
Core A's stall time
Core B's stall time
Shared Memory Resources
Intensive application A generates many requests and causes long stall times for less intensive application B
Request Generation Order
A1, B1, A2, A3, A4, B1
Throttled Requests
queue of requests to shared resources
Fair Source Throttling
A4
A3
A2
B1
A1
Oldest
A:
Compute
Stall on A1
Stall wait.
Stall on A2
Stall on A3
Stall on A4
B:
Compute
Stall wait.
Stall on B1
Extra Cycles Core A
Core A's stall time
Saved Cycles Core B
Core B's stall time
Shared Memory Resources
Dynamically detect application A's interference for application B and throttle down application A

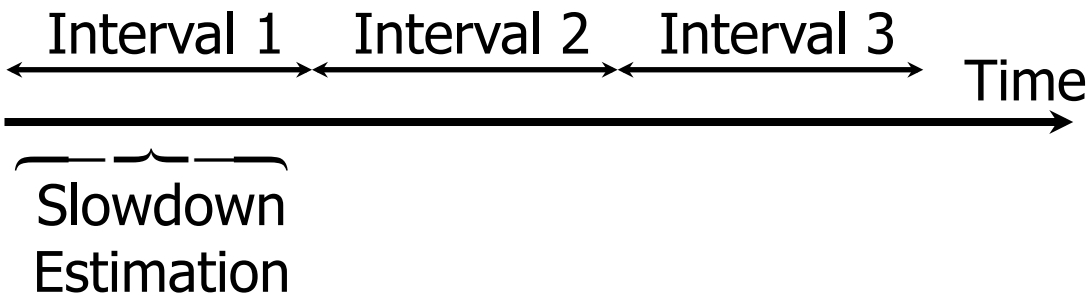
Fairness via Source Throttling (FST)

- Two components (interval-based)

- Run-time unfairness evaluation (in hardware)
  - Dynamically estimates the unfairness in the memory system
  - Estimates which application is slowing down which other

- Dynamic request throttling (hardware or software)
  - Adjusts how aggressively each core makes requests to the shared resources
  - Throttles down request rates of cores causing unfairness
    - Limit miss buffers, limit injection rate

Fairness via Source Throttling (FST)

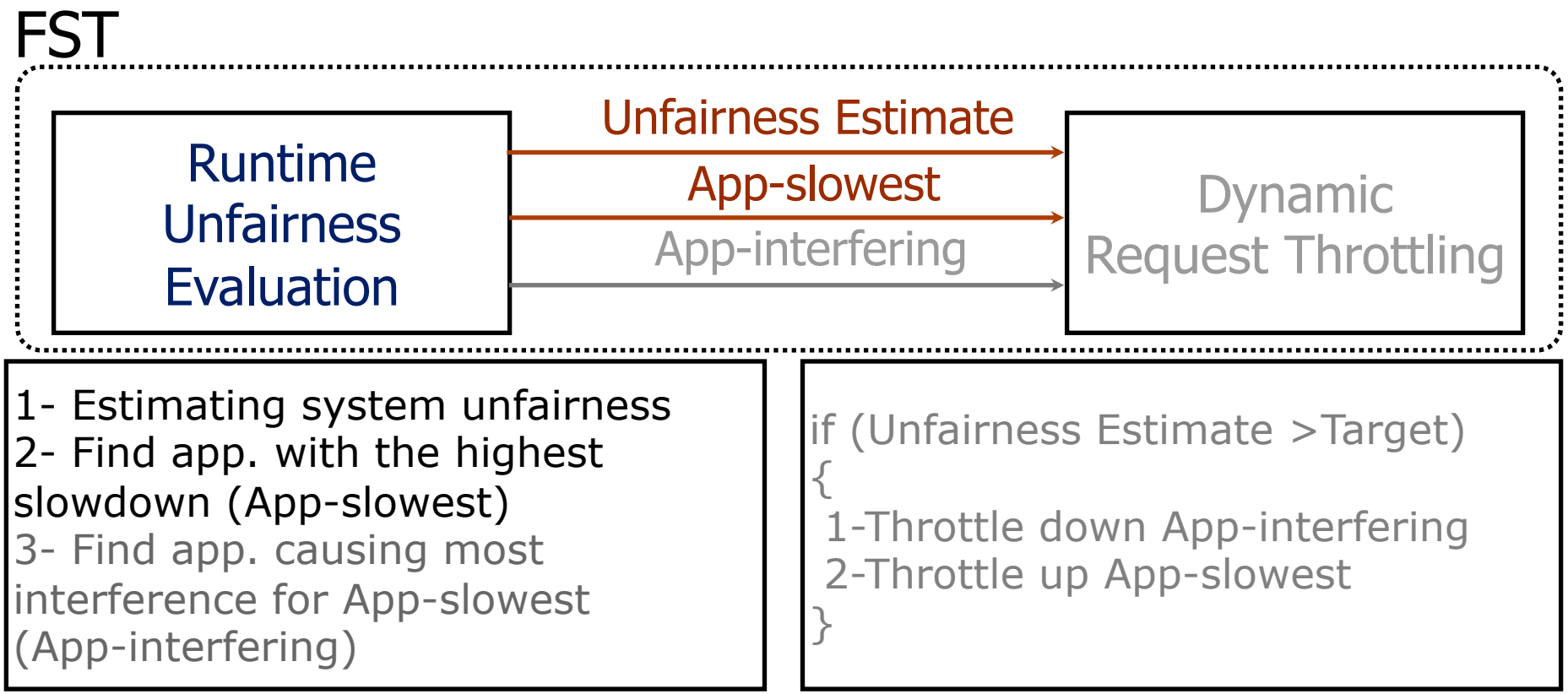


FST

1- Estimating system unfairness
2- Find app. with the highest slowdown (App-slowest)
3- Find app. causing most interference for App-slowest (App-interfering)

```
if (Unfairness Estimate >Target)
{
 1-Throttle down App-interfering
 2-Throttle up App-slowest
}
```

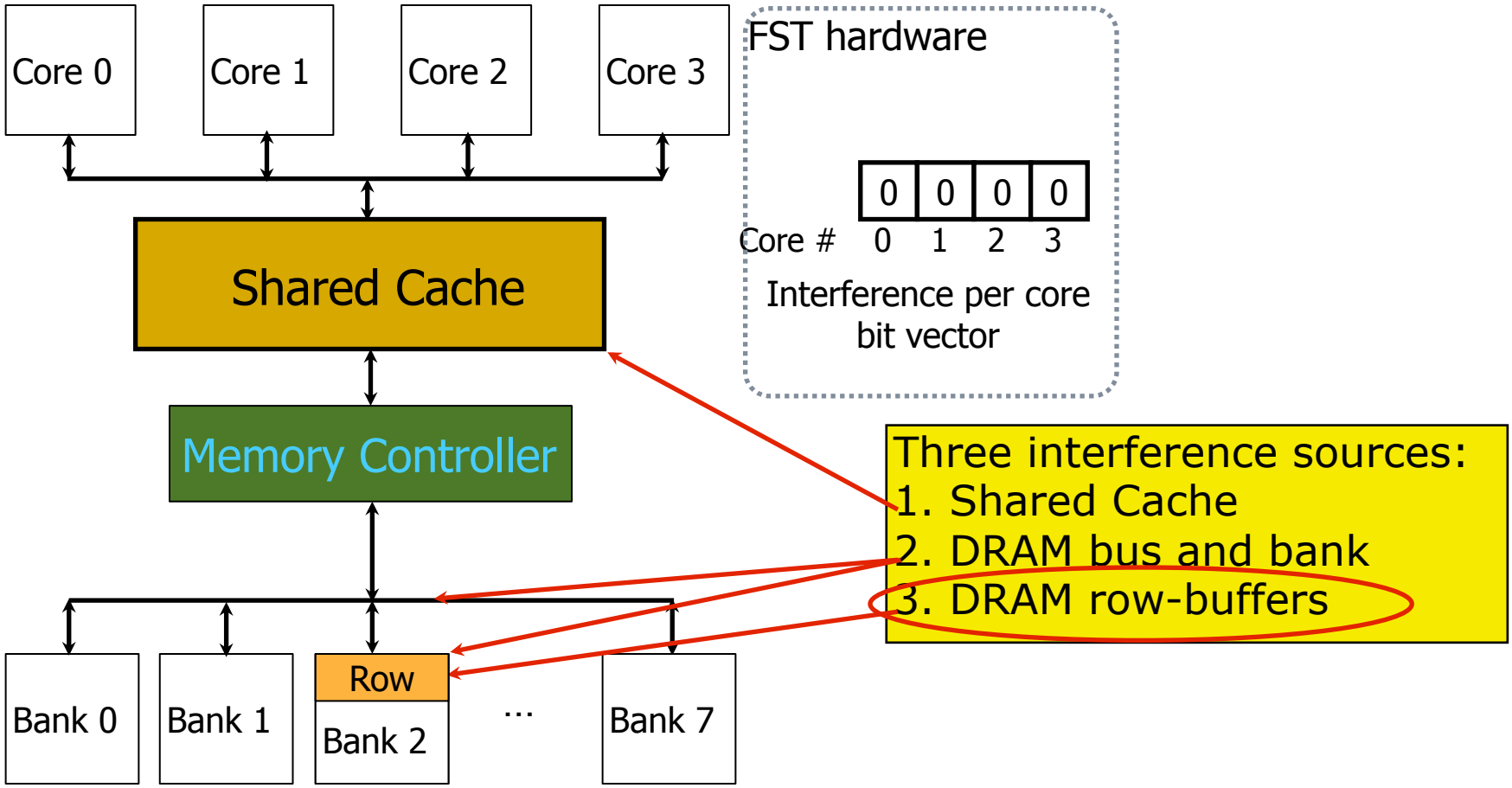
Fairness via Source Throttling (FST)



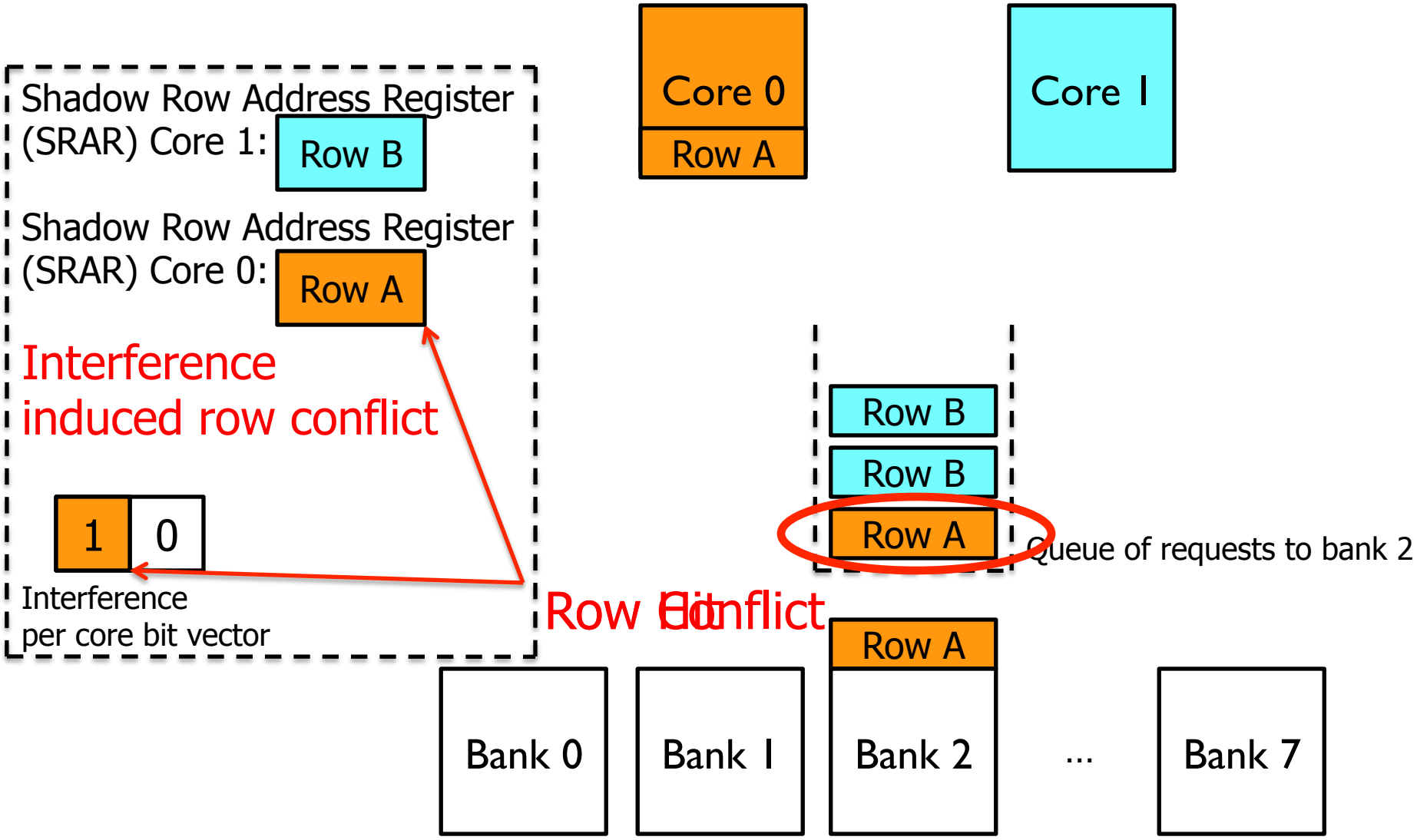
Estimating System Unfairness

- $$\text{Unfairness} = \frac{\text{Max}\{\text{Slowdown i}\} \text{ over all applications i}}{\text{Min}\{\text{Slowdown i}\} \text{ over all applications i}}$$
- $$\text{Slowdown of application i} = \frac{T_i^{Shared}}{T_i^{Alone}}$$
- How can T_i^{Alone} be estimated in shared mode?
- T_i^{Excess} is the number of extra cycles it takes application i to execute due to interference
- $T_i^{Alone} = T_i^{Shared} - \boxed{T_i^{Excess}}$

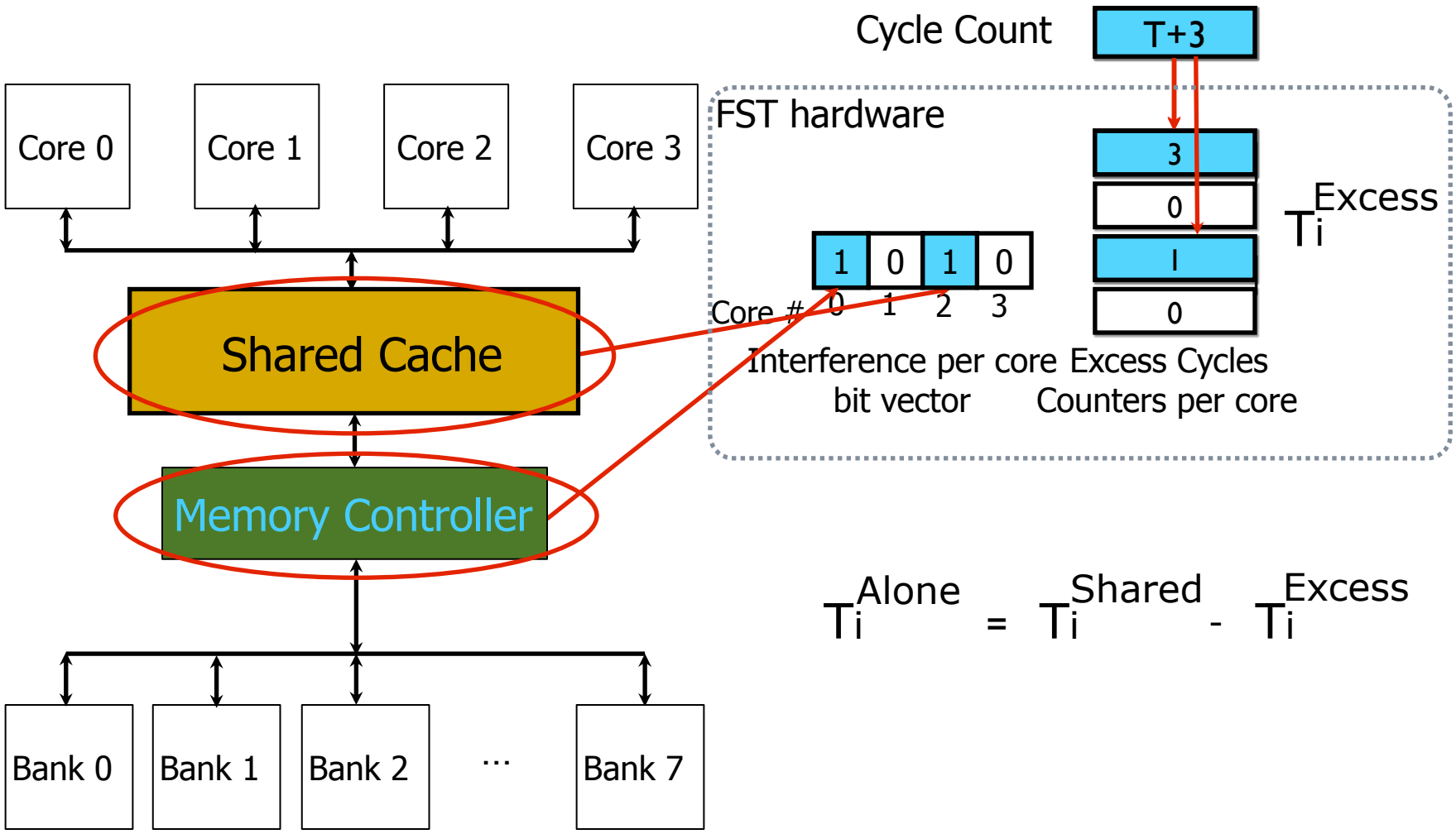
Tracking Inter-Core Interference



Tracking DRAM Row-Buffer Interference

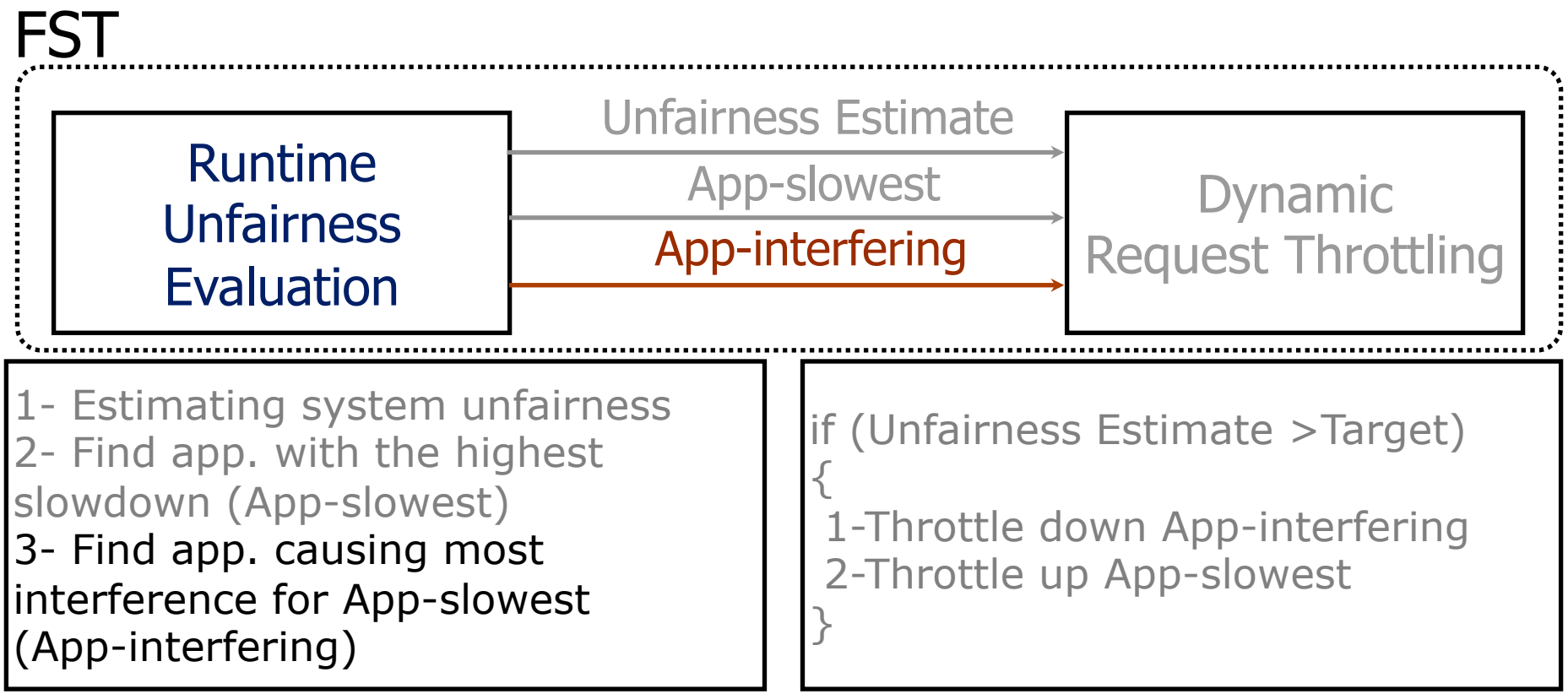


Tracking Inter-Core Interference



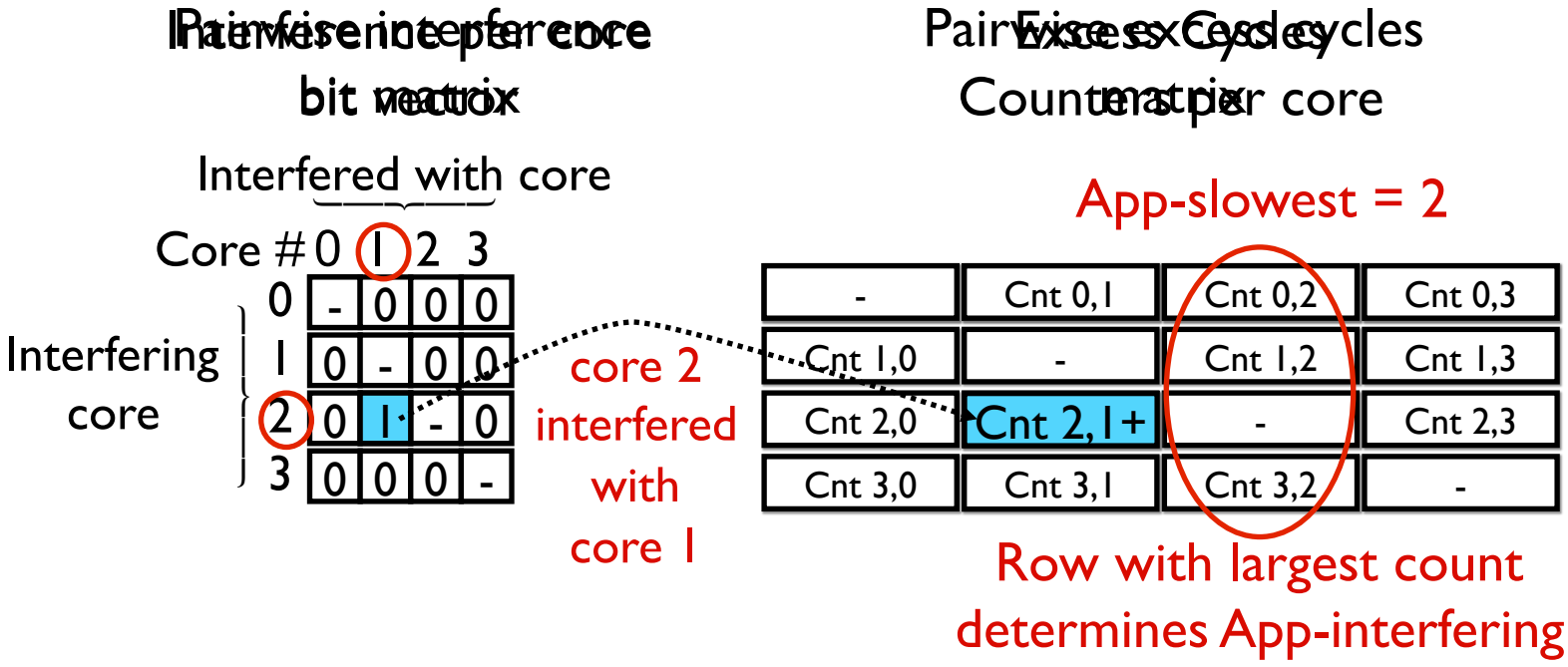
$$T_i^{Alone} = T_i^{Shared} - T_i^{Excess}$$

Fairness via Source Throttling (FST)

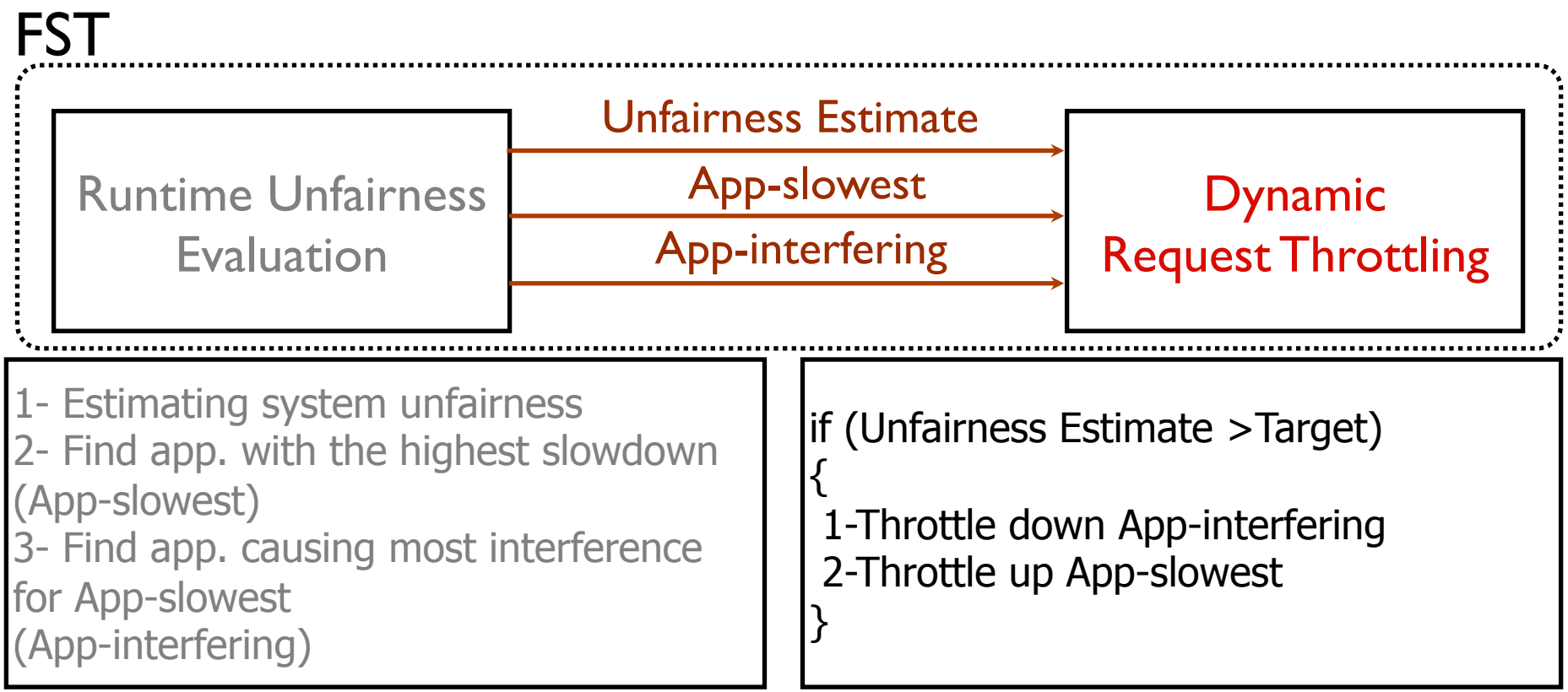


Tracking Inter-Core Interference

- To identify App-interfering, for each core i
  - FST separately tracks interference caused by each core j (j ≠ i)



Fairness via Source Throttling (FST)



Dynamic Request Throttling

- Goal: Adjust how aggressively each core makes requests to the shared memory system

- Mechanisms:
  - Miss Status Holding Register (MSHR) quota
    - Controls the number of concurrent requests accessing shared resources from each application
  - Request injection frequency
    - Controls how often memory requests are issued to the last level cache from the MSHRs

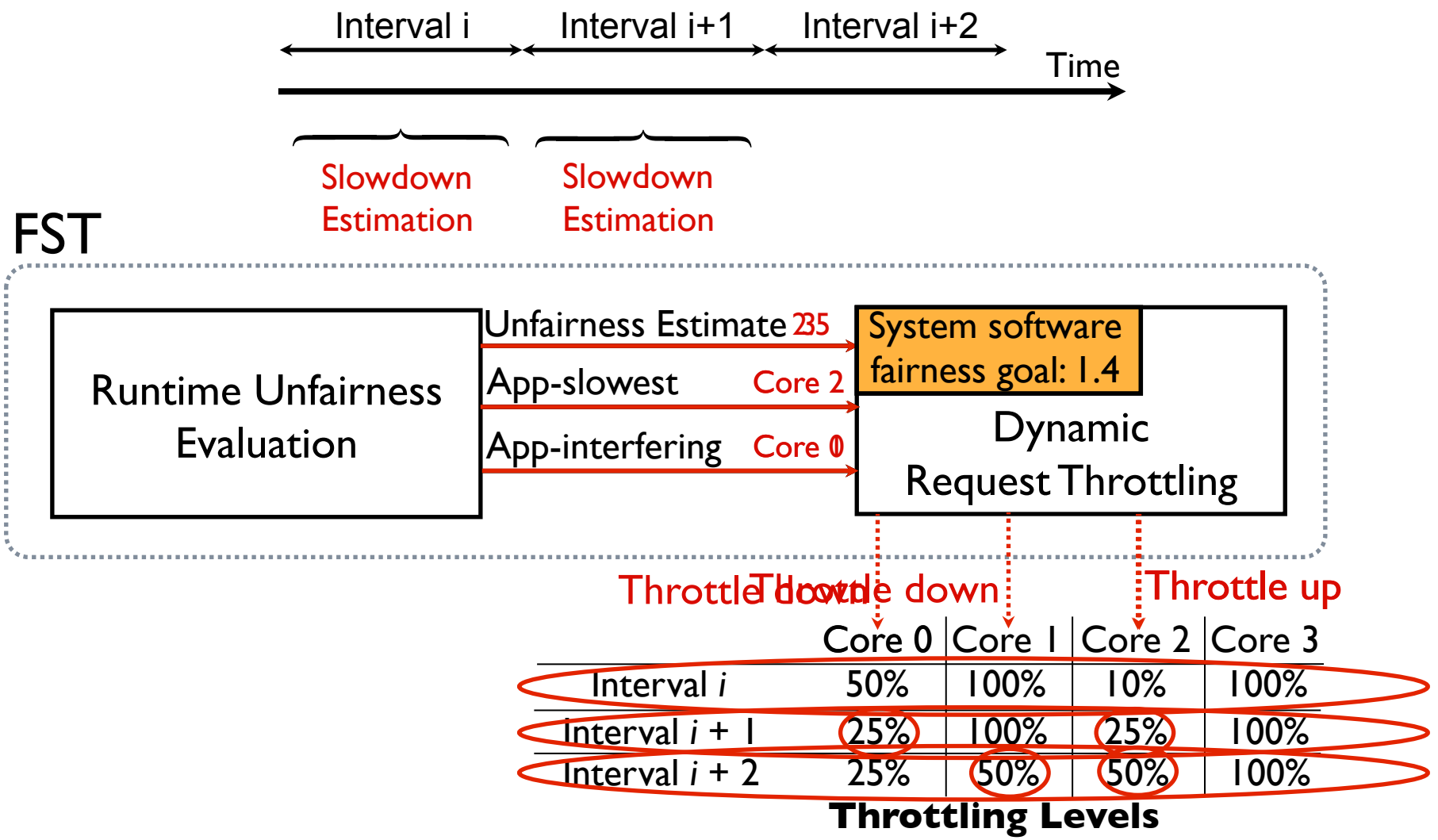
Dynamic Request Throttling

- Throttling level assigned to each core determines both MSHR quota and request injection rate

Throttling level	MSHR quota	Request Injection Rate
100%	128	Every cycle
50%	64	Every other cycle
25%	32	Once every 4 cycles
10%	12	Once every 10 cycles
5%	6	Once every 20 cycles
4%	5	Once every 25 cycles
3%	3	Once every 30 cycles
2%	2	Once every 50 cycles

Total # of MSHRs: 128

FST at Work


Interval i
Interval i+1
Interval i+2
Time
Slowdown Estimation
Slowdown Estimation
FST
Runtime Unfairness Evaluation
Unfairness Estimate 2.35
App-slowest Core 2
App-interfering Core 0
System software fairness goal: 1.4
Dynamic Request Throttling
Throttle down
Throttle down
Throttle up
Core 0 Core 1 Core 2 Core 3
Interval i 50% 100% 10% 100%
Interval i + 1 25% 100% 25% 100%
Interval i + 2 25% 50% 50% 100%
Throttling Levels

System Software Support

- Different fairness objectives can be configured by system software
  - Keep maximum slowdown in check
    - Estimated Max Slowdown < Target Max Slowdown
  - Keep slowdown of particular applications in check to achieve a particular performance target
    - Estimated Slowdown(i) < Target Slowdown(i)

- Support for thread priorities
  - Weighted Slowdown(i) =
    Estimated Slowdown(i) x Weight(i)

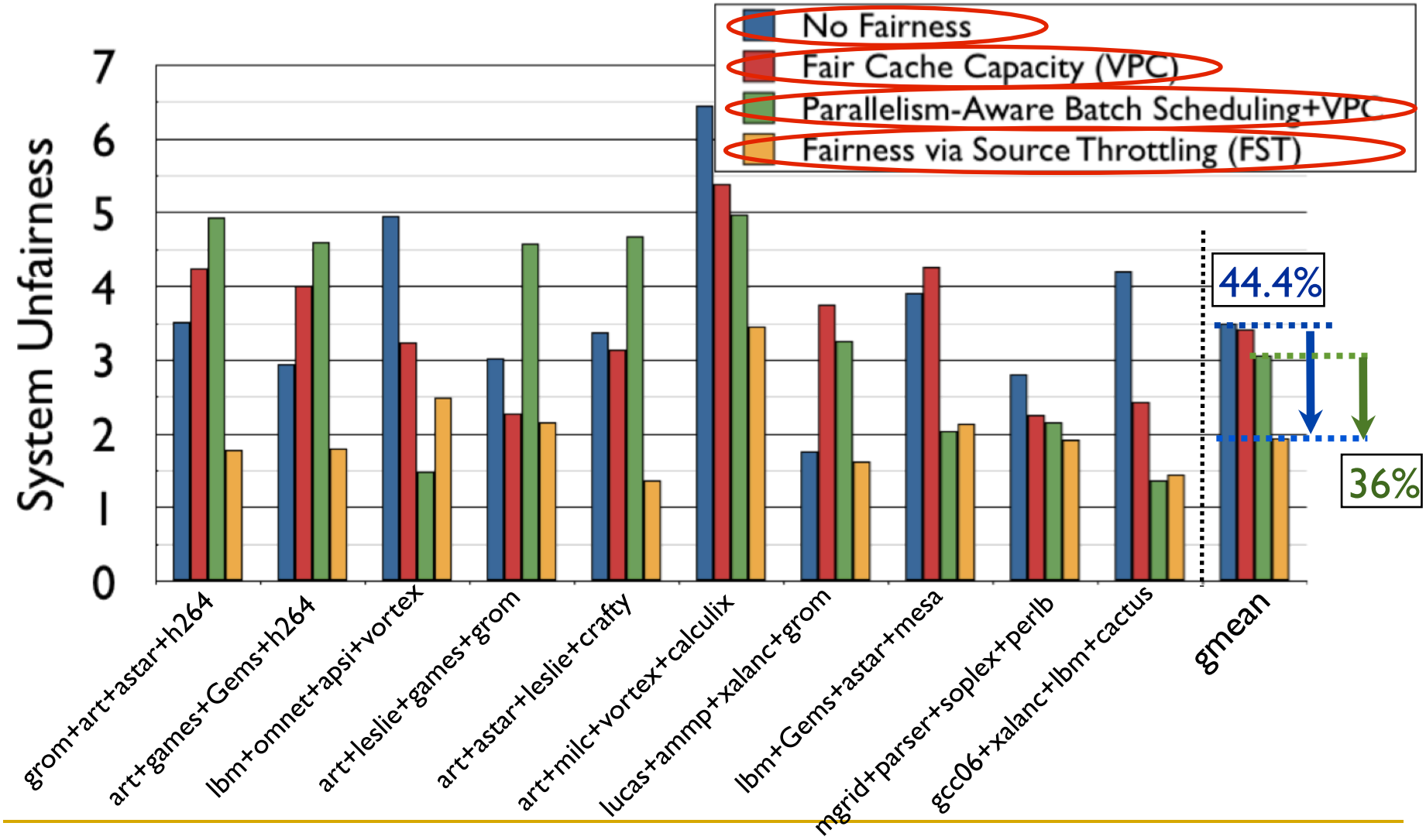
FST Hardware Cost

- Total storage cost required for 4 cores is ~12KB
- FST does not require any structures or logic that are on the processor's critical path

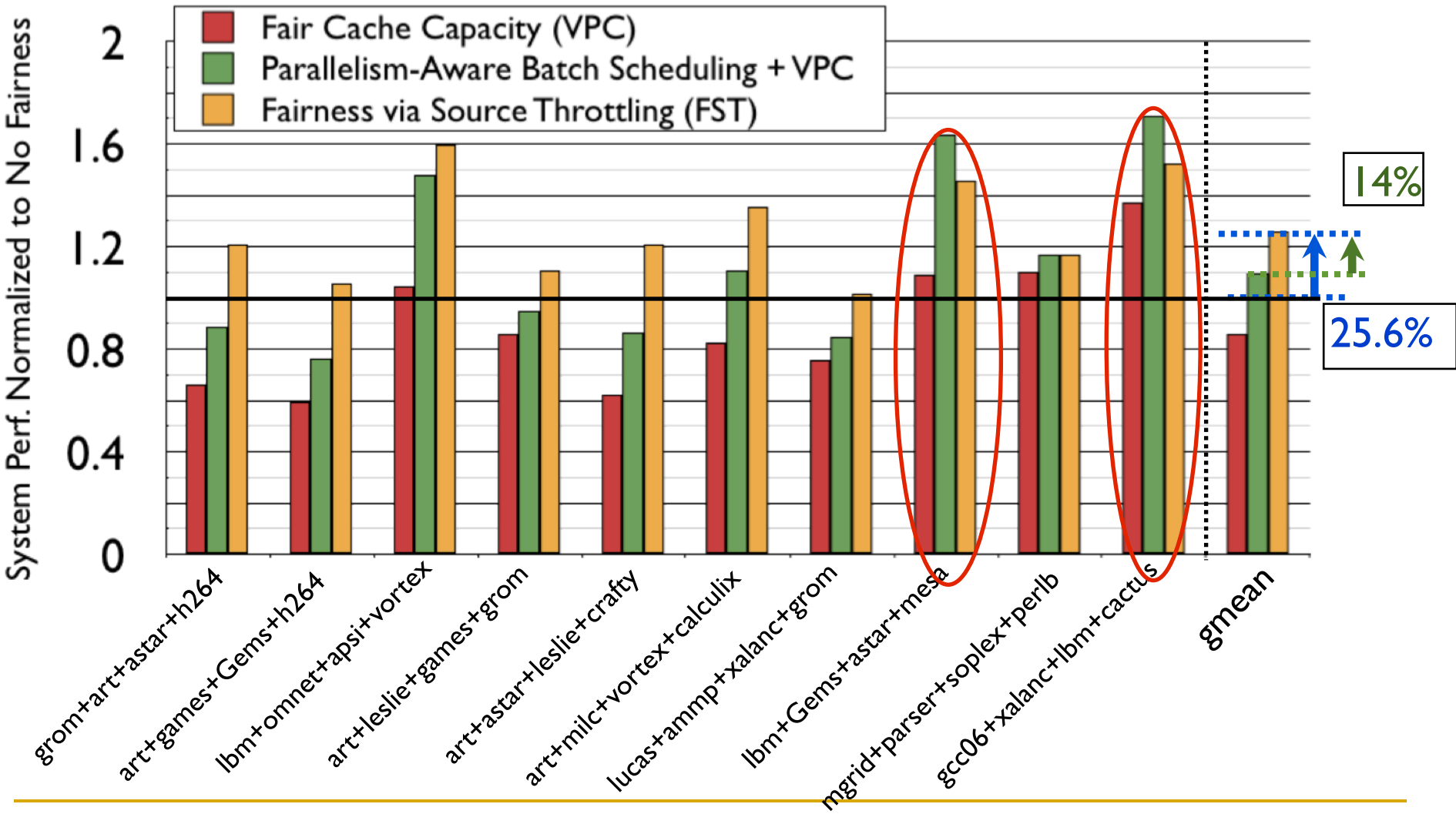
FST Evaluation Methodology

- x86 cycle accurate simulator
- Baseline processor configuration
  - Per-core
    - 4-wide issue, out-of-order, 256 entry ROB
  - Shared (4-core system)
    - 128 MSHRs
    - 2 MB, 16-way L2 cache
  - Main Memory
    - DDR3 1333 MHz
    - Latency of 15ns per command (tRP, tRCD, CL)
    - 8B wide core to memory bus

FST: System Unfairness Results



FST: System Performance Results



Source Throttling Results: Takeaways

- Source throttling alone provides better performance than a combination of "smart" memory scheduling and fair caching
  - Decisions made at the memory scheduler and the cache sometimes contradict each other
- Neither source throttling alone nor "smart resources" alone provides the best performance
- Combined approaches are even more powerful
  - Source throttling and resource-based interference control

FST ASPLOS 2010 Talk

Designing QoS-Aware Memory Systems: Approaches

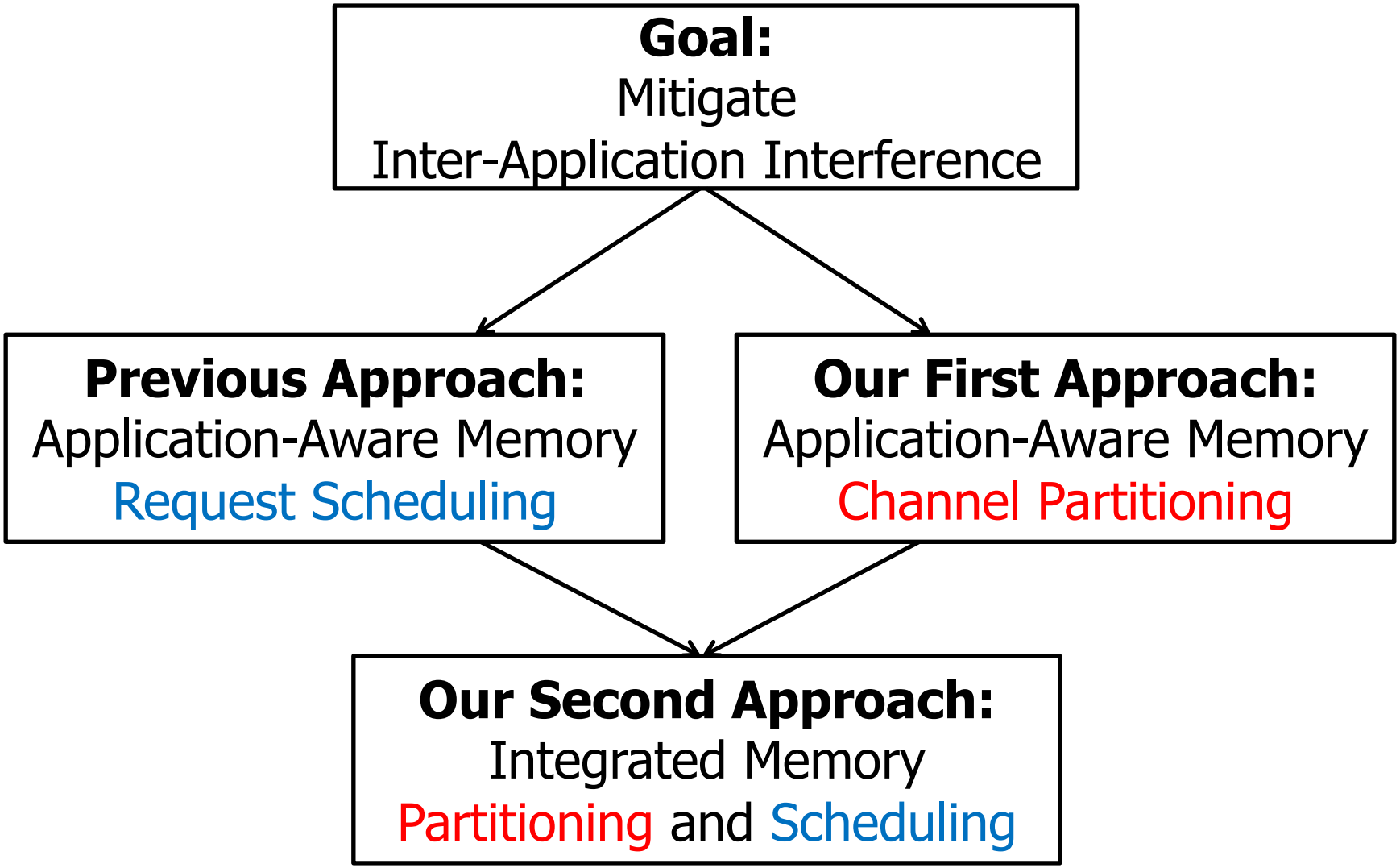
- Smart resources: Design each shared resource to have a configurable interference control/reduction mechanism
  - QoS-aware memory controllers [Mutlu+ MICRO'07] [Moscibroda+, Usenix Security'07] [Mutlu+ ISCA'08, Top Picks'09] [Kim+ HPCA'10] [Kim+ MICRO'10, Top Picks'11] [Ebrahimi+ ISCA'11, MICRO'11] [Ausavarungnirun+, ISCA'12] [Subramanian+, HPCA'13]
  - QoS-aware interconnects [Das+ MICRO'09, ISCA'10, Top Picks '11] [Grot+ MICRO'09, ISCA'11, Top Picks '12]
  - QoS-aware caches
- Dumb resources: Keep each resource free-for-all, but reduce/control interference by injection control or data mapping
  - Source throttling to control access to memory system [Ebrahimi+ ASPLOS'10, ISCA'11, TOCS'12] [Ebrahimi+ MICRO'09] [Nychis+ HotNets'10] [Nychis+ SIGCOMM'12]
  - QoS-aware data mapping to memory controllers [Muralidhara+ MICRO'11]
  - QoS-aware thread scheduling to cores [Das+ HPCA'13]

Memory Channel Partitioning

Sai Prashanth Muralidhara, Lavanya Subramanian, Onur Mutlu, Mahmut Kandemir, and Thomas Moscibroda,
"Reducing Memory Interference in Multicore Systems via Application-Aware Memory Channel Partitioning"
44th International Symposium on Microarchitecture (***MICRO***),
Porto Alegre, Brazil, December 2011. Slides (pptx)

MCP Micro 2011 Talk

Outline


Goal:
Mitigate
Inter-Application Interference
Previous Approach:
Application-Aware Memory
Request Scheduling
Our First Approach:
Application-Aware Memory
Channel Partitioning
Our Second Approach:
Integrated Memory
Partitioning and Scheduling

Application-Aware Memory Request Scheduling

- Monitor application memory access characteristics
- Rank applications based on memory access characteristics
- Prioritize requests at the memory controller, based on ranking

An Example: Thread Cluster Memory Scheduling

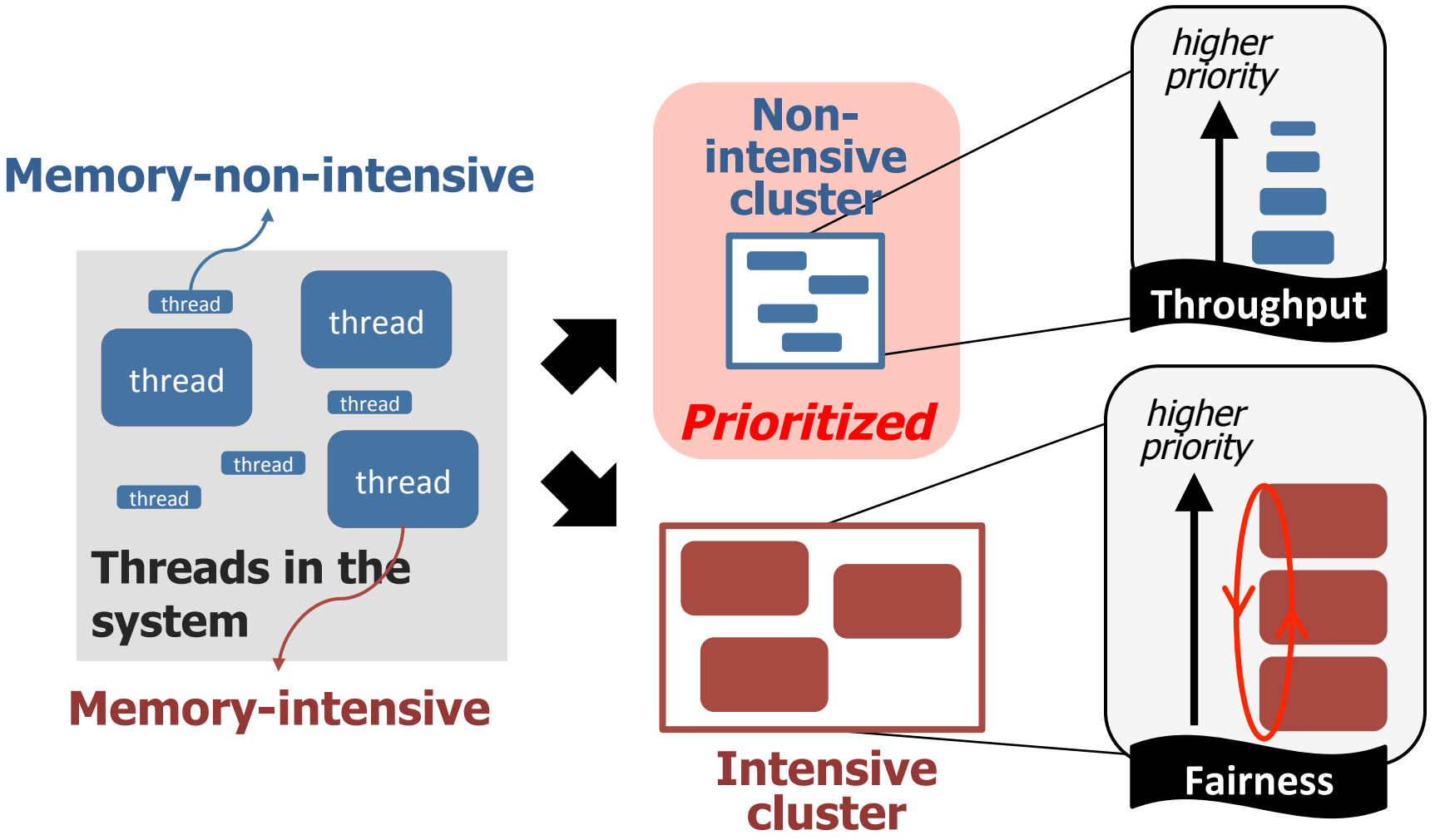


Figure: Kim et al., MICRO 2010

Application-Aware Memory Request Scheduling

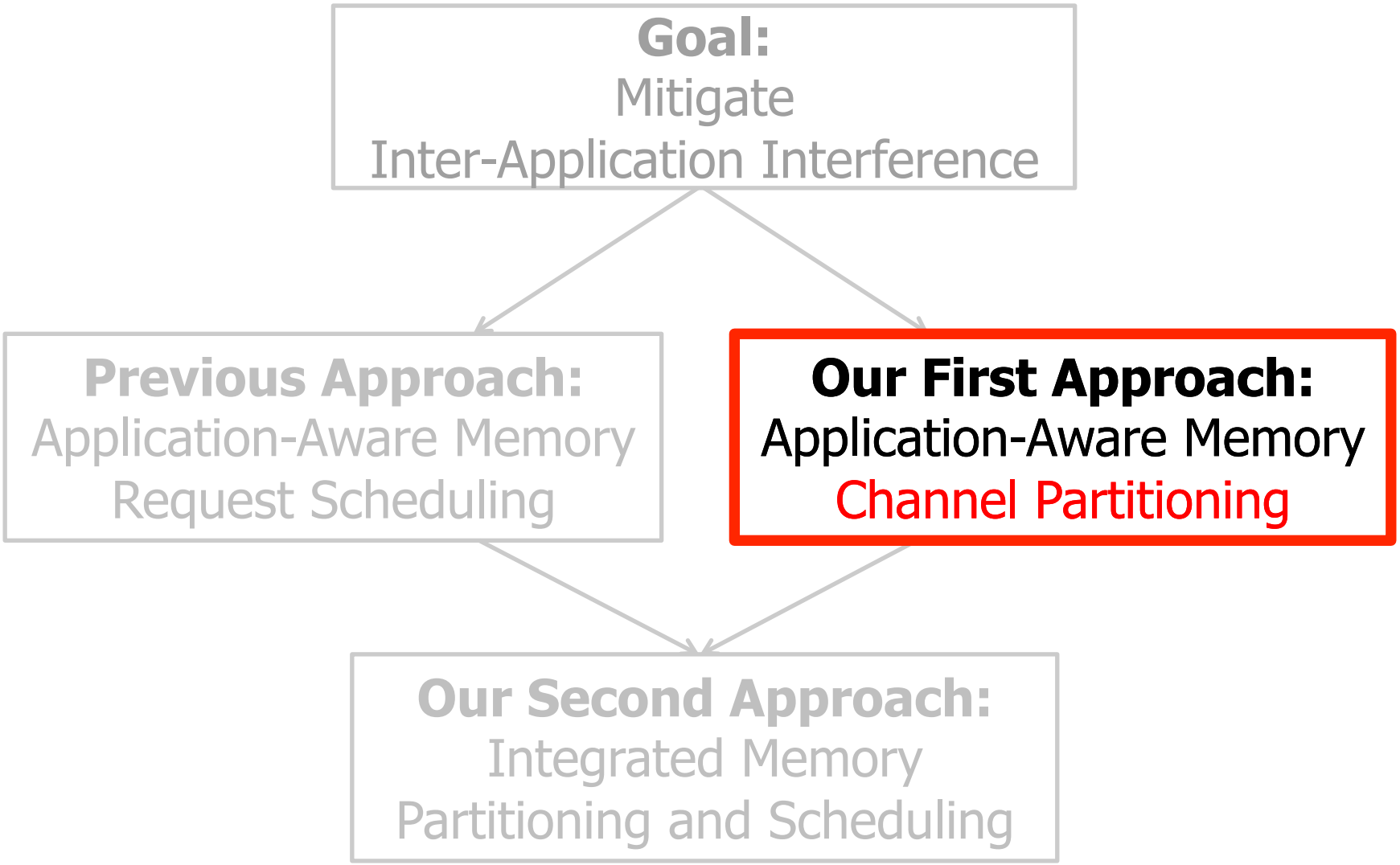
Advantages

- Reduces interference between applications by request reordering
- Improves system performance

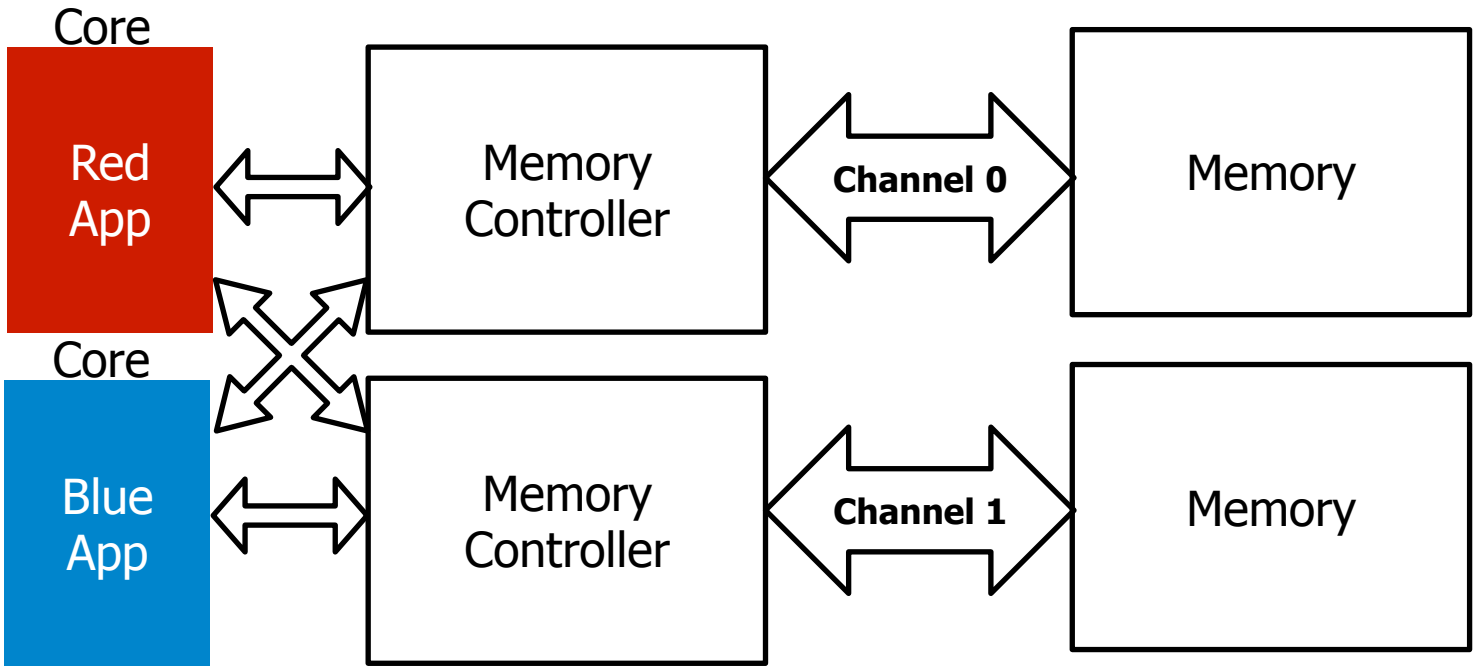
Disadvantages

- Requires modifications to memory scheduling logic for
  - Ranking
  - Prioritization
- Cannot completely eliminate interference by request reordering

Our Approach



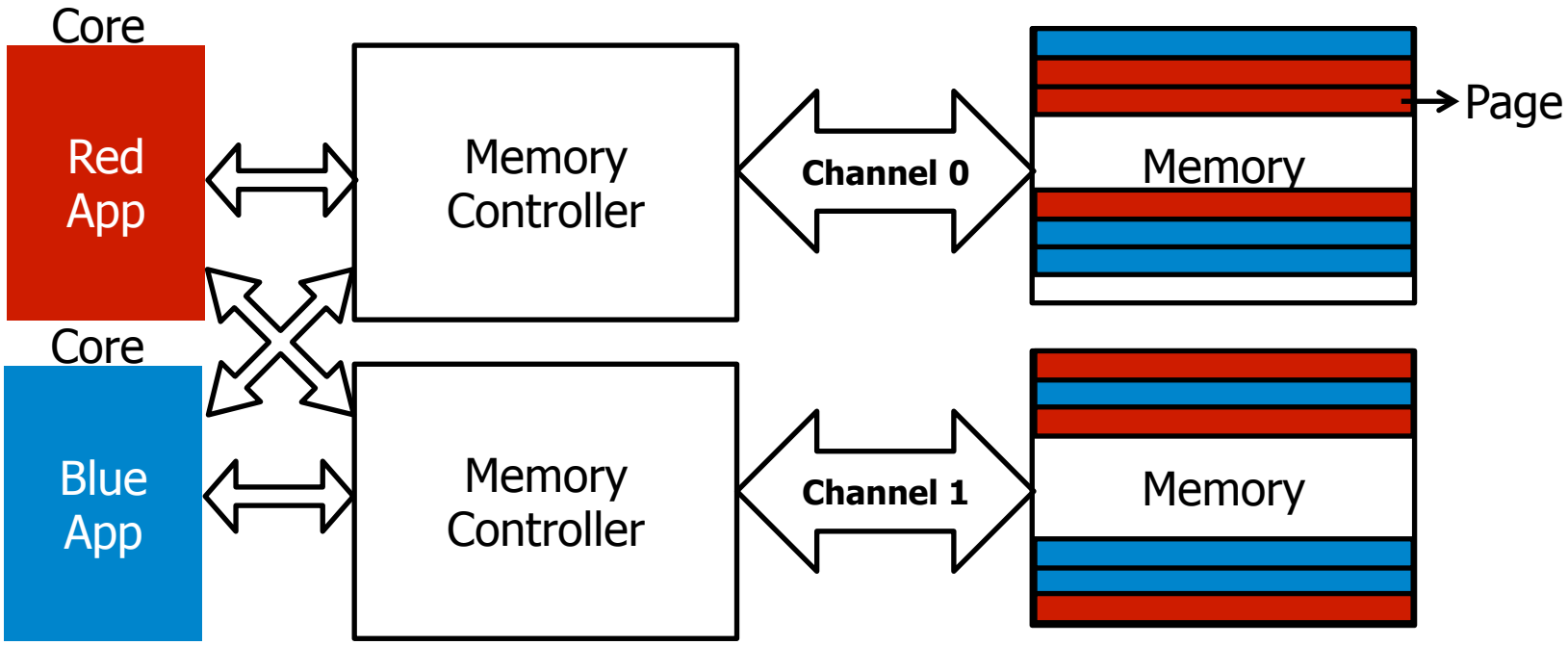
Observation: Modern Systems Have Multiple Channels



A new degree of freedom

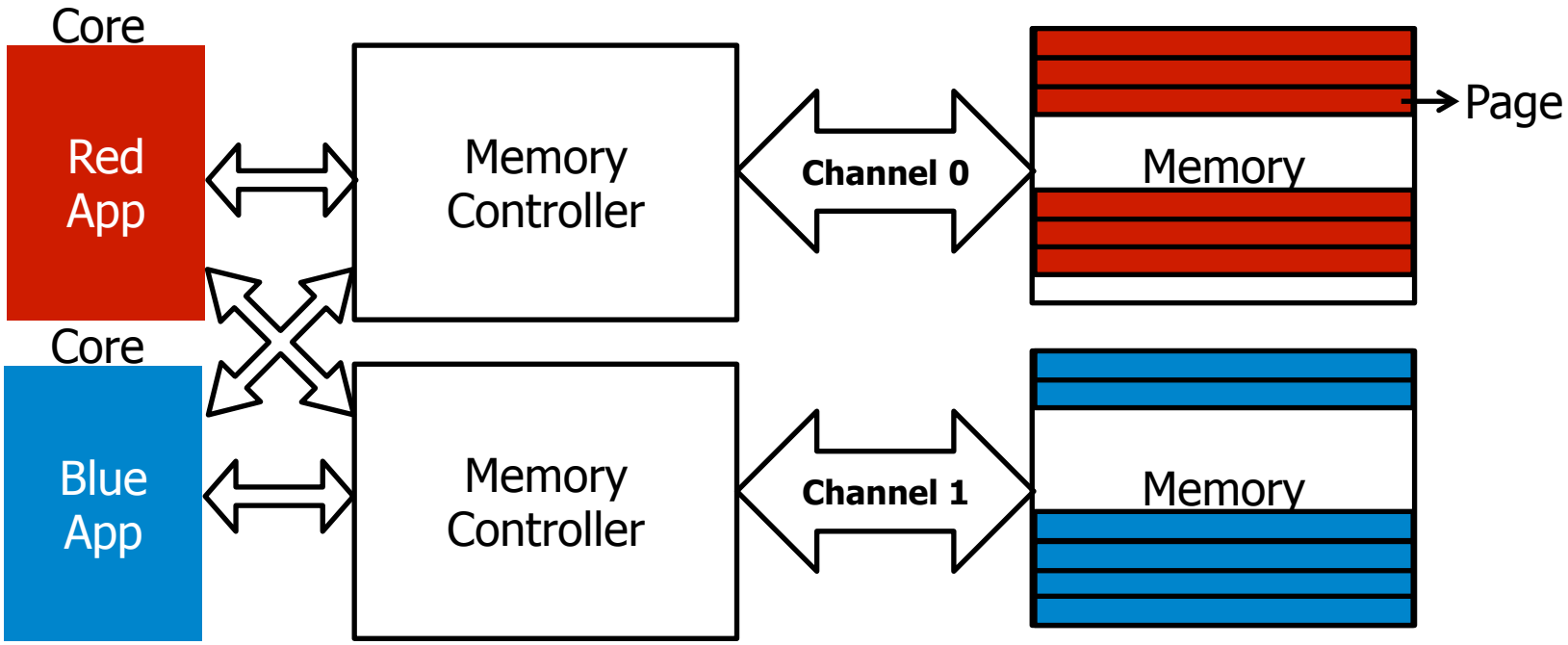
Mapping data across multiple channels

Data Mapping in Current Systems



Causes interference between applications' requests

Partitioning Channels Between Applications



Eliminates interference between applications' requests

Overview: Memory Channel Partitioning (MCP)

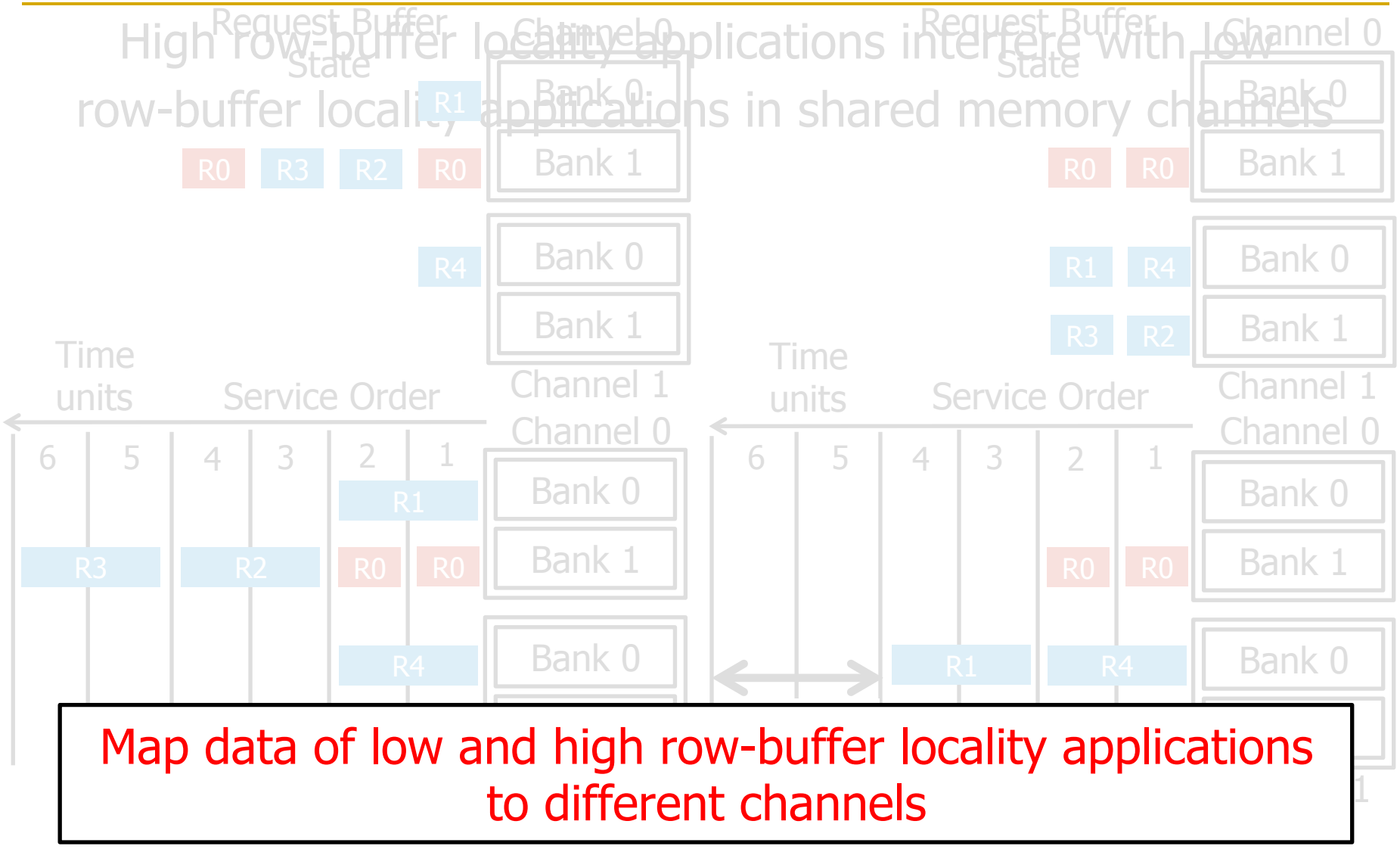
- Goal
  - Eliminate harmful interference between applications
- Basic Idea
  - Map the data of badly-interfering applications to different channels
- Key Principles
  - Separate low and high memory-intensity applications
  - Separate low and high row-buffer locality applications

Key Insight 1: Separate by Memory Intensity

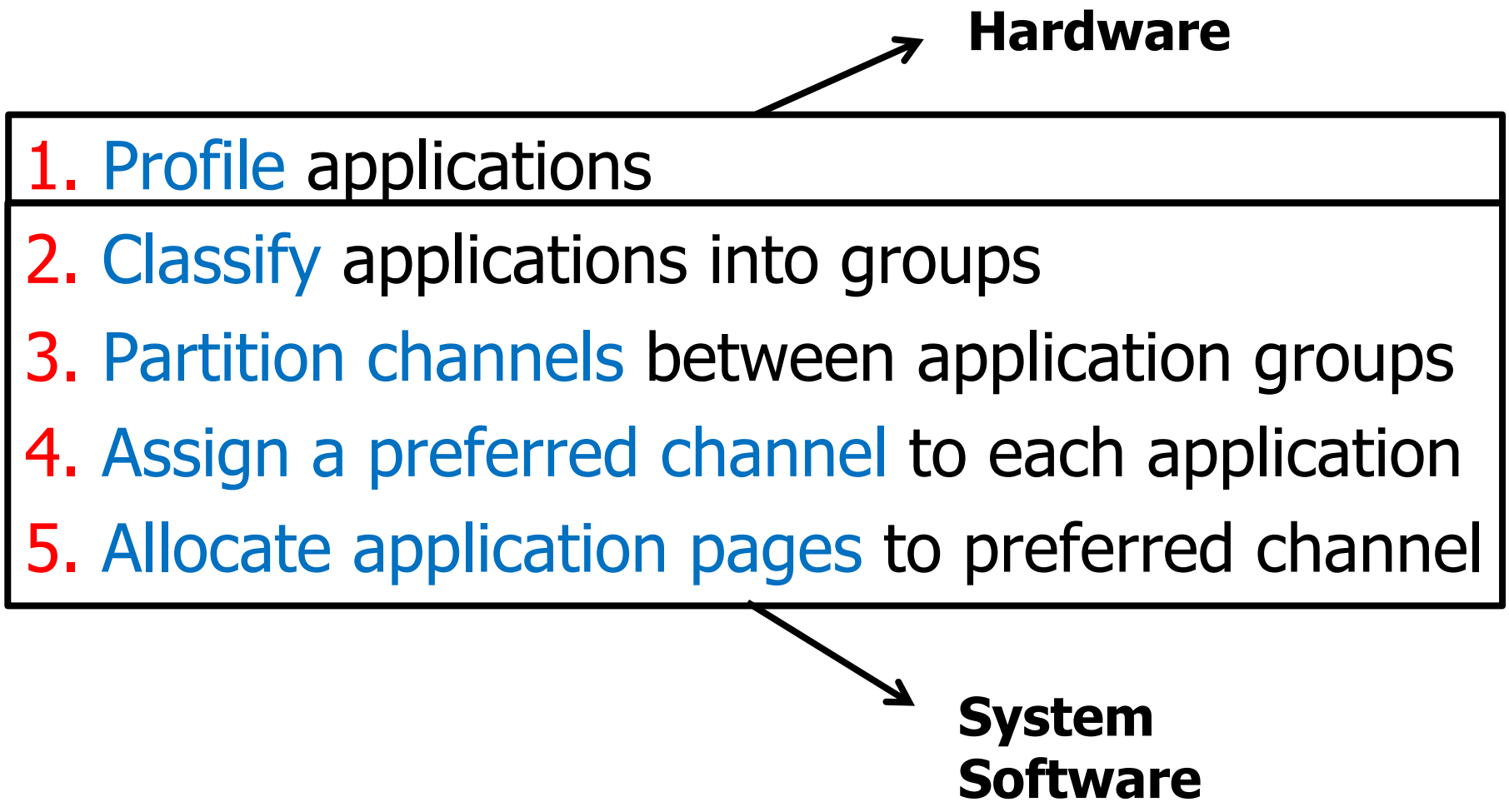
High memory-intensity applications interfere with low memory-intensity applications in shared memory channels

Map data of low and high memory-intensity applications to different channels

Key Insight 2: Separate by Row-Buffer Locality



Memory Channel Partitioning (MCP) Mechanism



1. Profile Applications

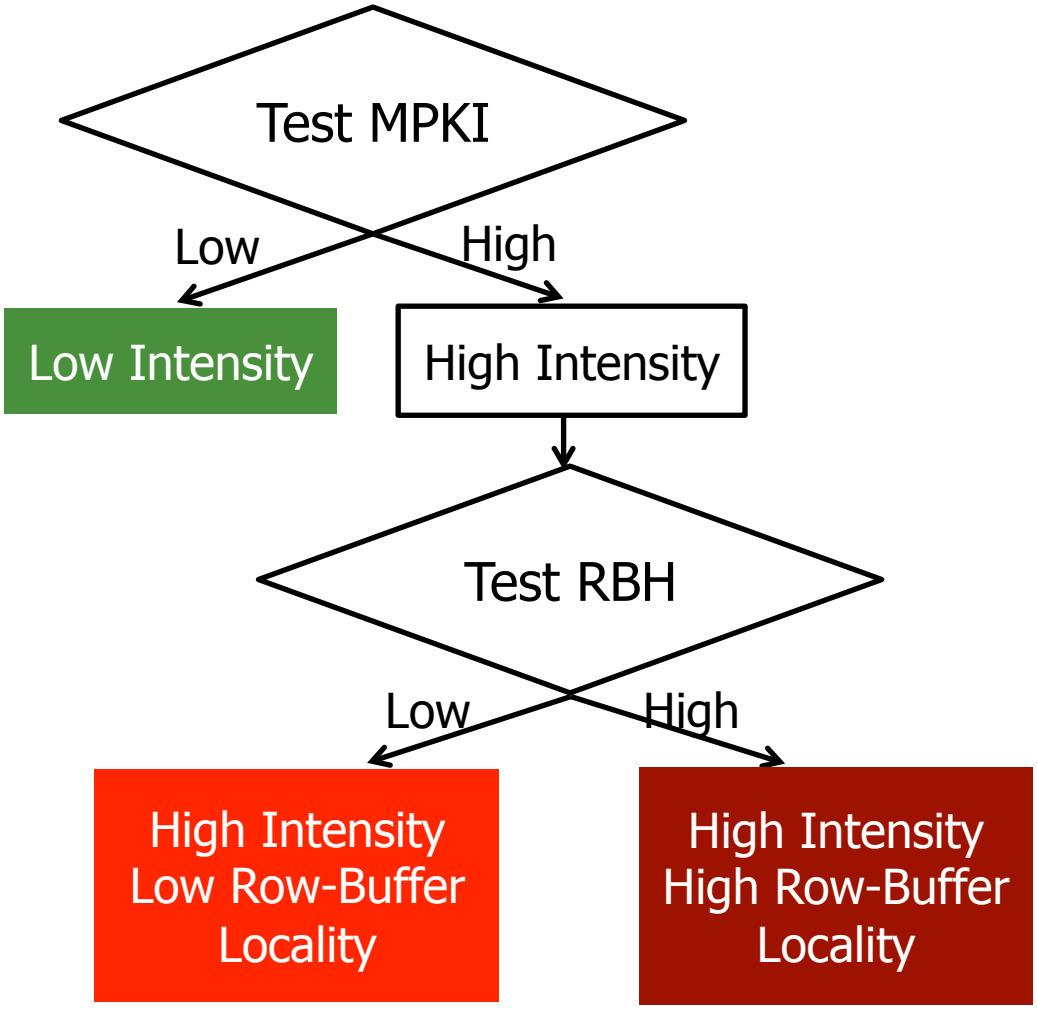
- Hardware counters collect application memory access characteristics

- Memory access characteristics
  - Memory intensity:

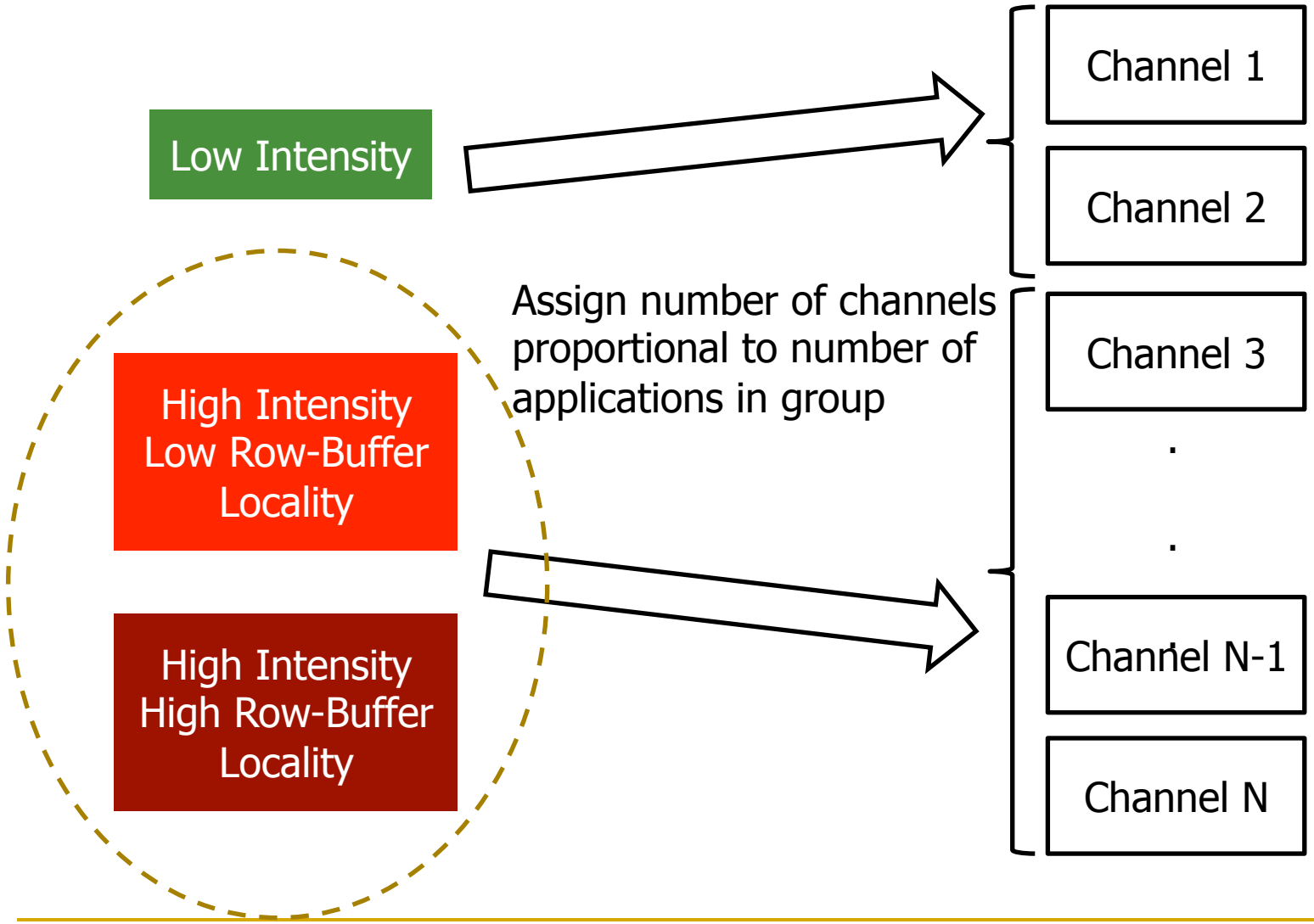
    Last level cache Misses Per Kilo Instruction (MPKI)
  - Row-buffer locality:

    Row-buffer Hit Rate (RBH) - percentage of accesses that hit in the row buffer

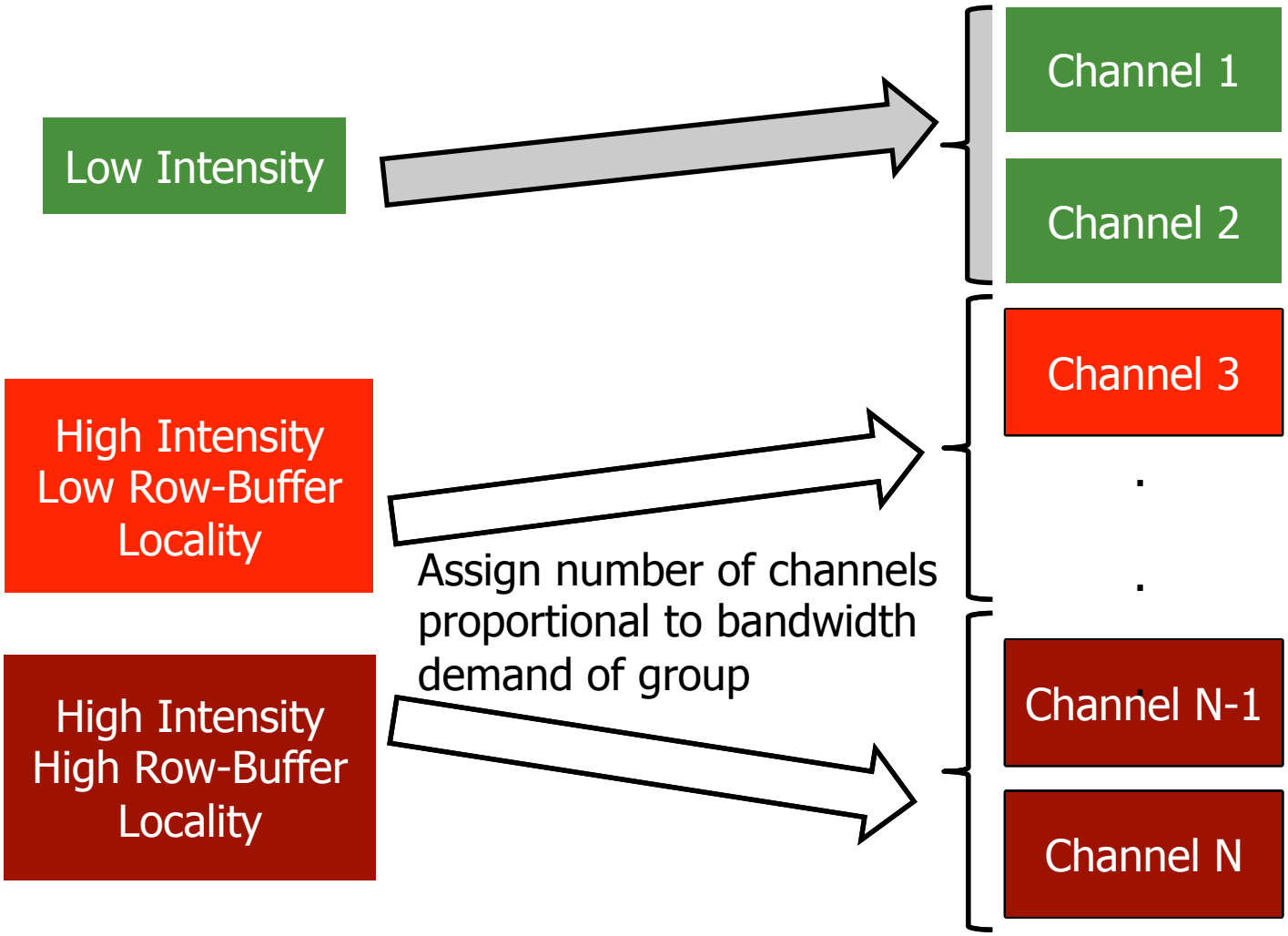
2. Classify Applications



3. Partition Channels Among Groups: Step 1

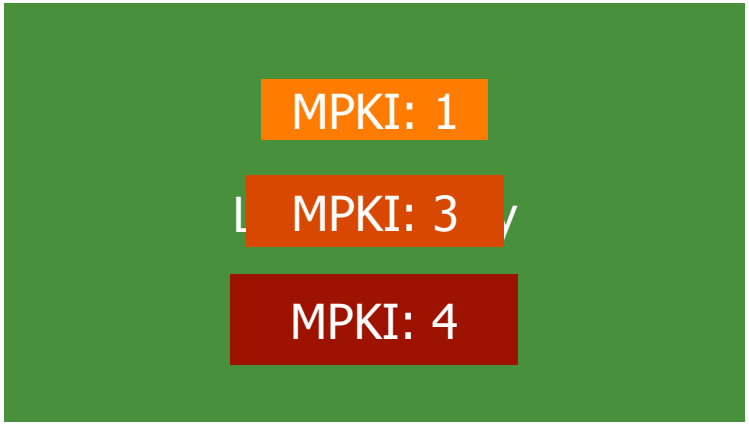


3. Partition Channels Among Groups: Step 2



4. Assign Preferred Channel to Application

- Assign each application a preferred channel from its group's allocated channels
- Distribute applications to channels such that group's bandwidth demand is balanced across its channels

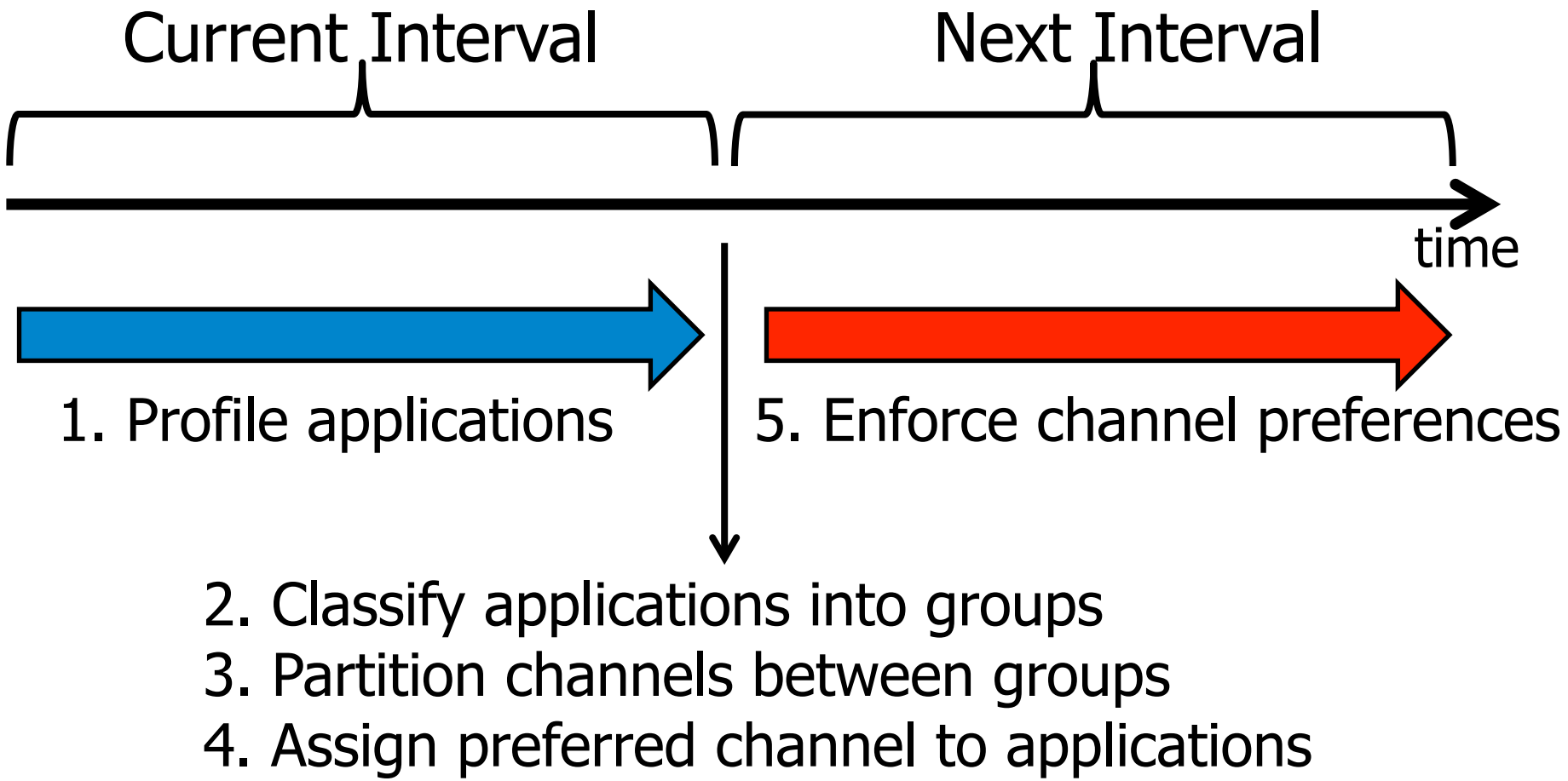


5. Allocate Page to Preferred Channel

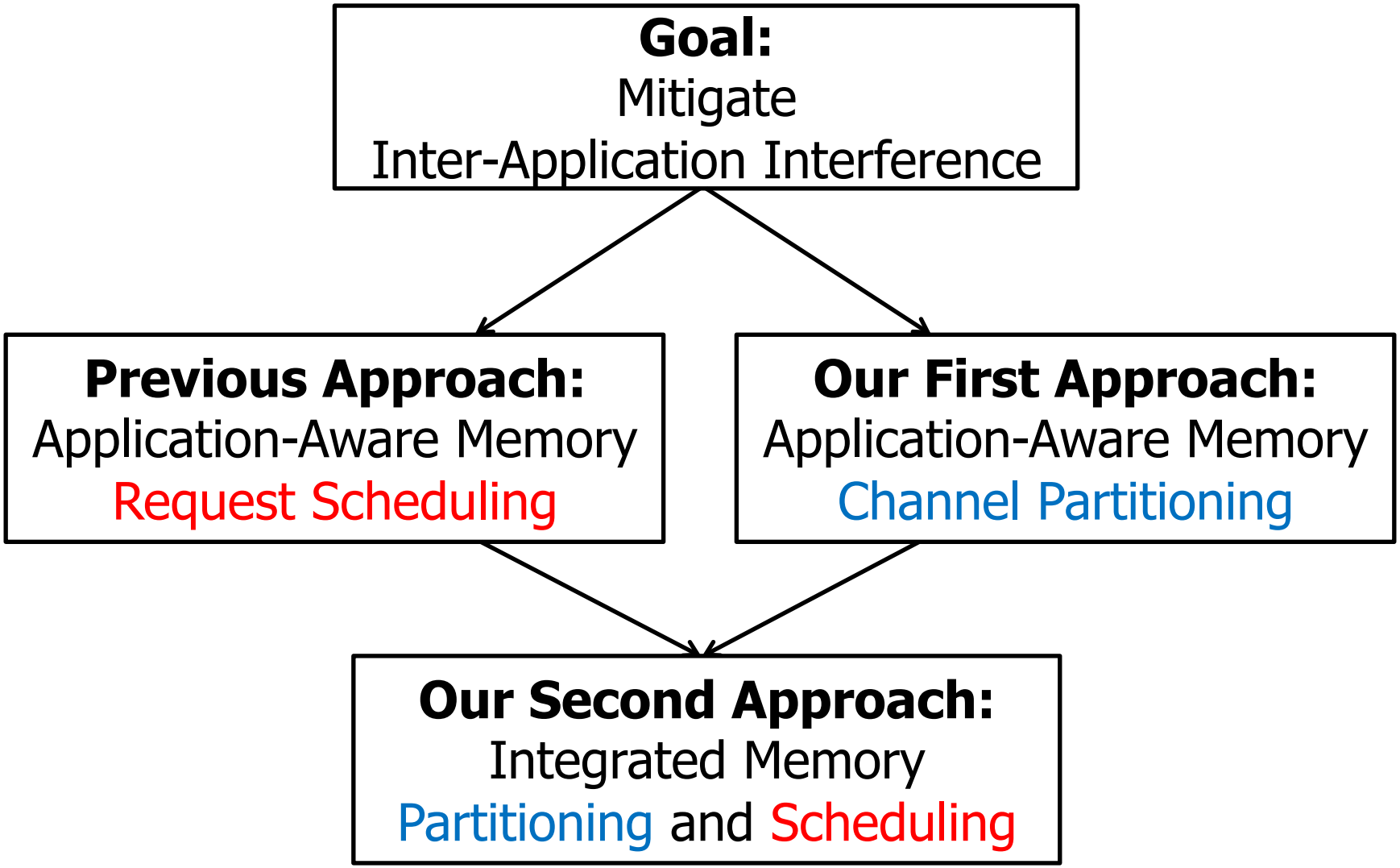
- Enforce channel preferences
  computed in the previous step

- On a page fault, the operating system
  - allocates page to preferred channel if free page available in preferred channel
  - if free page not available, replacement policy tries to allocate page to preferred channel
  - if it fails, allocate page to another channel

Interval Based Operation



Integrating Partitioning and Scheduling



Observations

- Applications with very low memory-intensity rarely access memory
  → Dedicating channels to them results in precious memory bandwidth waste

- They have the most potential to keep their cores busy
  → We would really like to prioritize them

- They interfere minimally with other applications
  → Prioritizing them does not hurt others

Integrated Memory Partitioning and Scheduling (IMPS)

- Always prioritize very low memory-intensity applications in the memory scheduler

- Use memory channel partitioning to mitigate interference between other applications

Hardware Cost

- Memory Channel Partitioning (MCP)
  - Only profiling counters in hardware
  - No modifications to memory scheduling logic
  - 1.5 KB storage cost for a 24-core, 4-channel system

- Integrated Memory Partitioning and Scheduling (IMPS)
  - A single bit per request
  - Scheduler prioritizes based on this single bit

Methodology

- Simulation Model
  - 24 cores, 4 channels, 4 banks/channel
  - Core Model
    - Out-of-order, 128-entry instruction window
    - 512 KB L2 cache/core
  - Memory Model – DDR2
- Workloads
  - 240 SPEC CPU 2006 multiprogrammed workloads (categorized based on memory intensity)
- Metrics
  - System Performance $Weighted\ Speedup = \sum_i \frac{IPC_i^{shared}}{IPC_i^{alone}}$

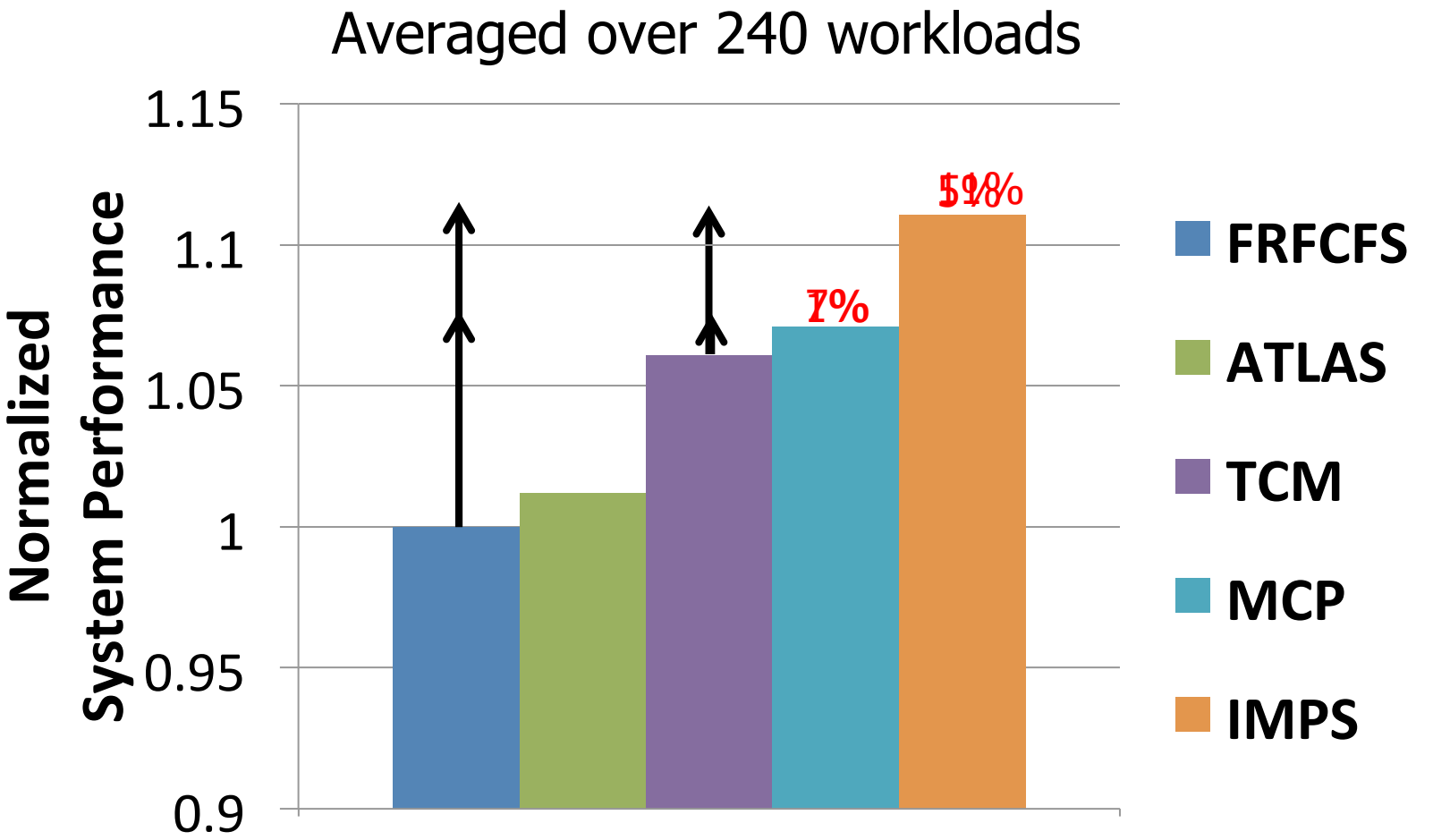
Previous Work on Memory Scheduling

- FR-FCFS [Zuravleff et al., US Patent 1997, Rixner et al., ISCA 2000]
  - Prioritizes row-buffer hits and older requests
  - Application-unaware

- ATLAS [Kim et al., HPCA 2010]
  - Prioritizes applications with low memory-intensity

- TCM [Kim et al., MICRO 2010]
  - Always prioritizes low memory-intensity applications
  - Shuffles request priorities of high memory-intensity applications

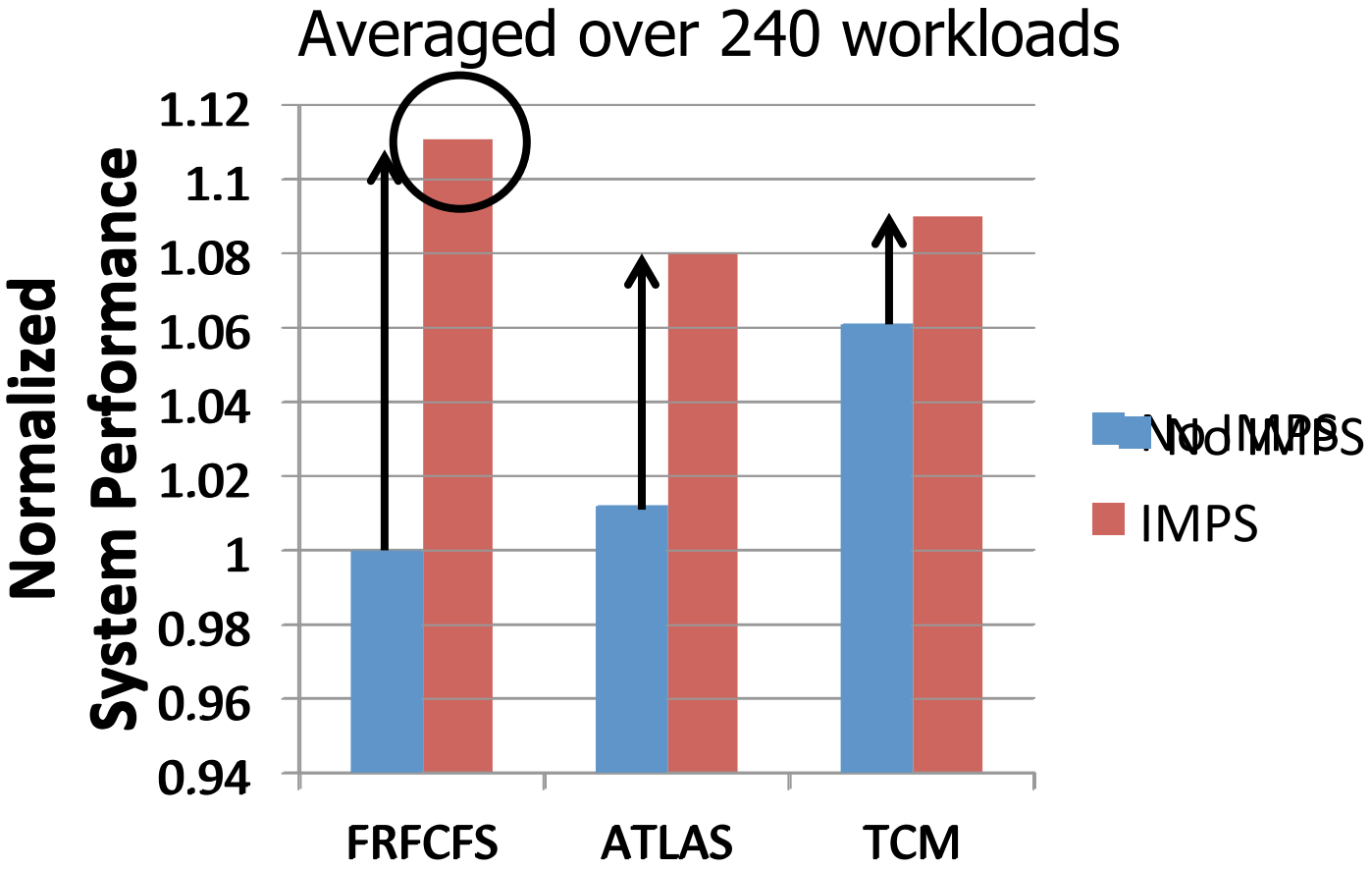
Comparison to Previous Scheduling Policies



Better system performance than the best previous scheduler at lower hardware cost

Significant performance improvement over baseline FRFCFS

Interaction with Memory Scheduling



IMPS improves performance regardless of scheduling policy

Highest improvement over FRFCFS as IMPS designed for FRFCFS

MCP Summary

- Uncontrolled inter-application interference in main memory degrades system performance

- Application-aware memory channel partitioning (MCP)
  - Separates the data of badly-interfering applications to different channels, eliminating interference

- Integrated memory partitioning and scheduling (IMPS)
  - Prioritizes very low memory-intensity applications in scheduler
  - Handles other applications' interference by partitioning

- MCP/IMPS provide better performance than application-aware memory request scheduling at lower hardware cost

Staged Memory Scheduling

Rachata Ausavarungnirun, Kevin Chang, Lavanya Subramanian, Gabriel Loh, and Onur Mutlu,

"Staged Memory Scheduling: Achieving High Performance and Scalability in Heterogeneous Systems"

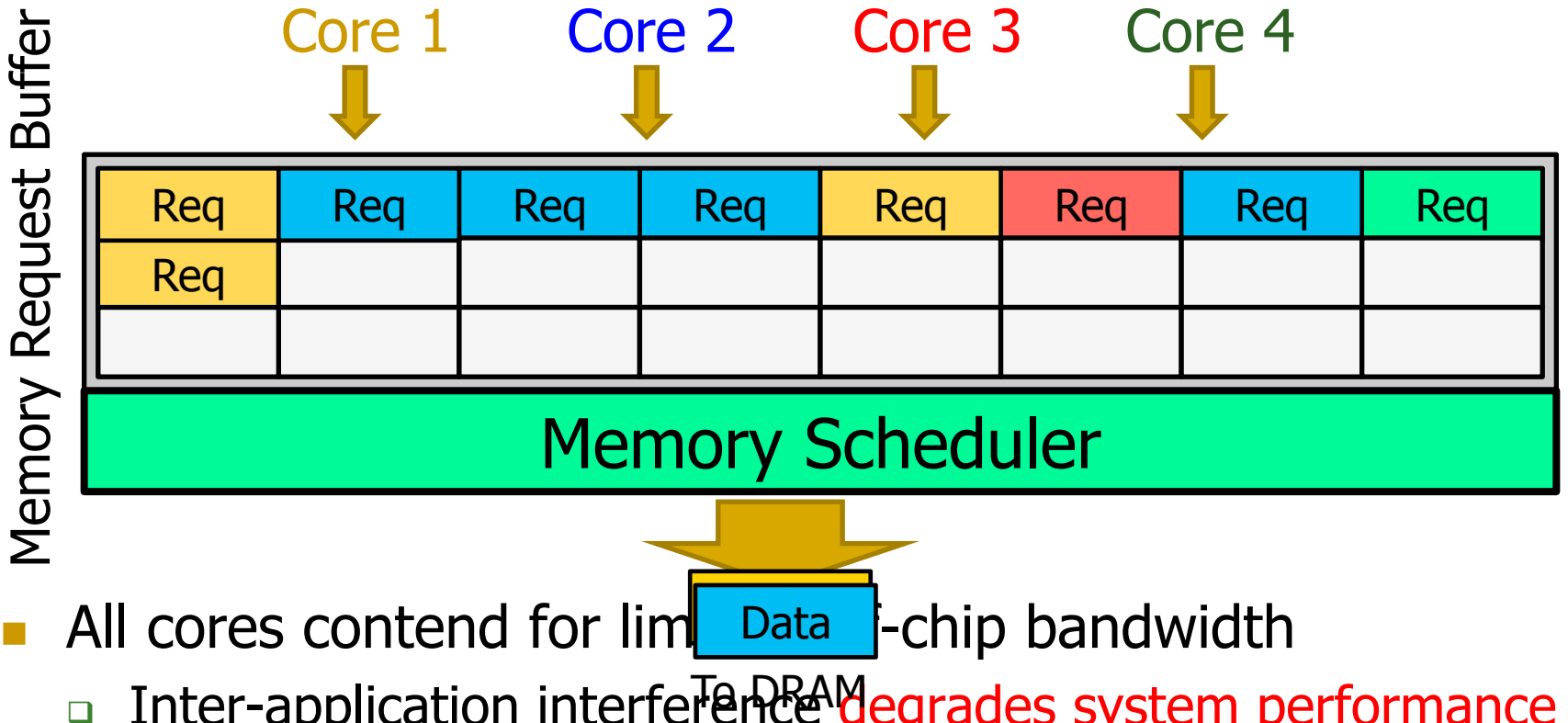
39th International Symposium on Computer Architecture (***ISCA***),
Portland, OR, June 2012.

SMS ISCA 2012 Talk

Executive Summary

- **Observation:** Heterogeneous CPU-GPU systems require memory schedulers with large request buffers
- **Problem:** Existing monolithic application-aware memory scheduler designs are hard to scale to large request buffer sizes
- **Solution:** Staged Memory Scheduling (SMS)
  decomposes the memory controller into three simple stages:
  1) Batch formation: maintains row buffer locality
  2) Batch scheduler: reduces interference between applications
  3) DRAM command scheduler: issues requests to DRAM
- Compared to state-of-the-art memory schedulers:
  - SMS is significantly simpler and more scalable
  - SMS provides higher performance and fairness

Main Memory is a Bottleneck



- All cores contend for lim -chip bandwidth
  - Inter-application interference degrades system performance
  - The memory scheduler can help mitigate the problem
- How does the memory scheduler deliver good performance and fairness?

Three Principles of Memory Scheduling

- Prioritize row-buffer-hit requests [Rixner+, ISCA'00]
  - To maximize memory bandwidth

- Prioritize latency-sensitive applications [Kim+, HPCA'10]
  - To maximize system throughput

- Ensure that no application is starved [Mutlu and Moscibroda, MICRO'07]
  - To mi

Older

Newer

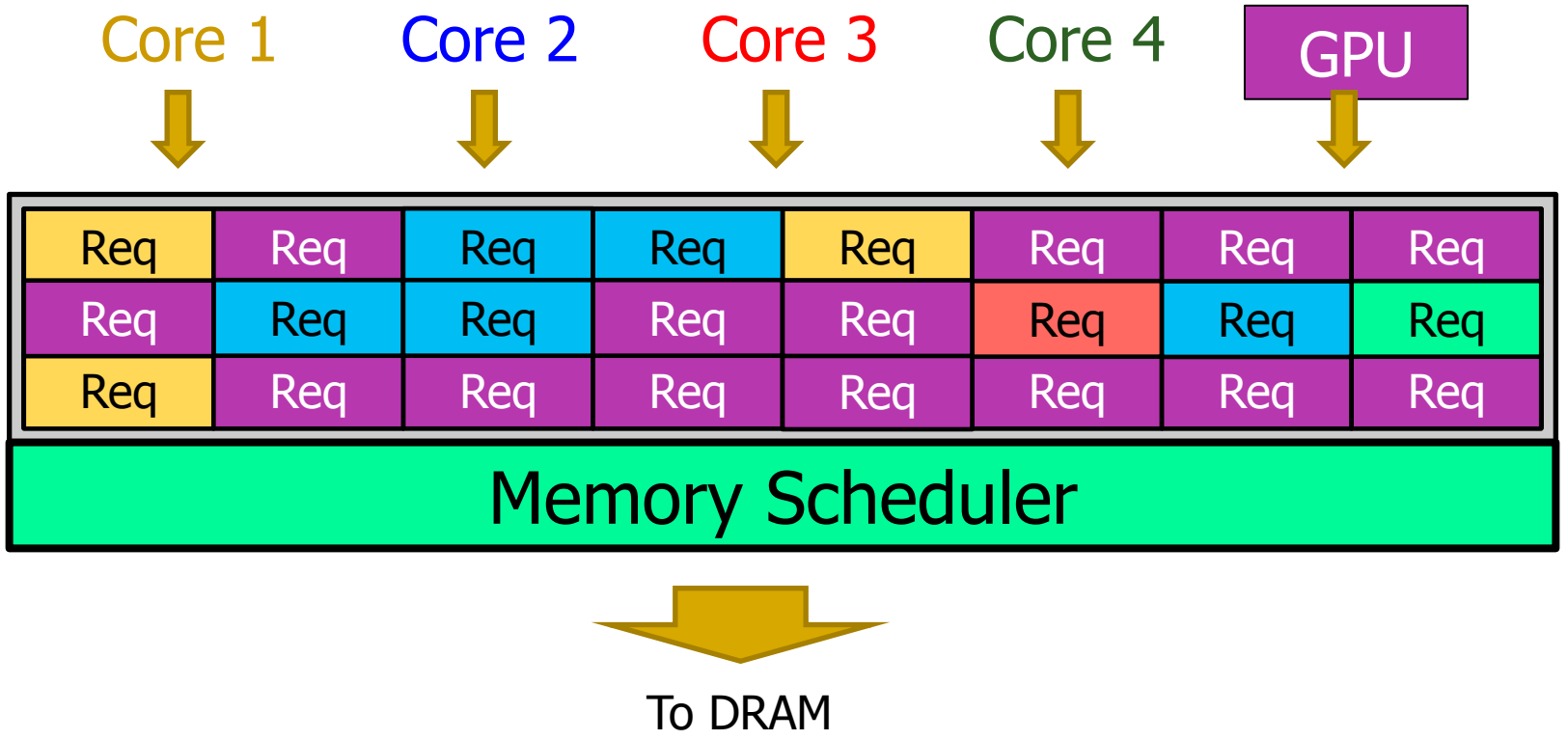
Reg 1 | Row A

Application	Memory Intensity (MPKI)	row
1	5	
2	1	
3	2	
4	10	

Memory Scheduling for CPU-GPU Systems

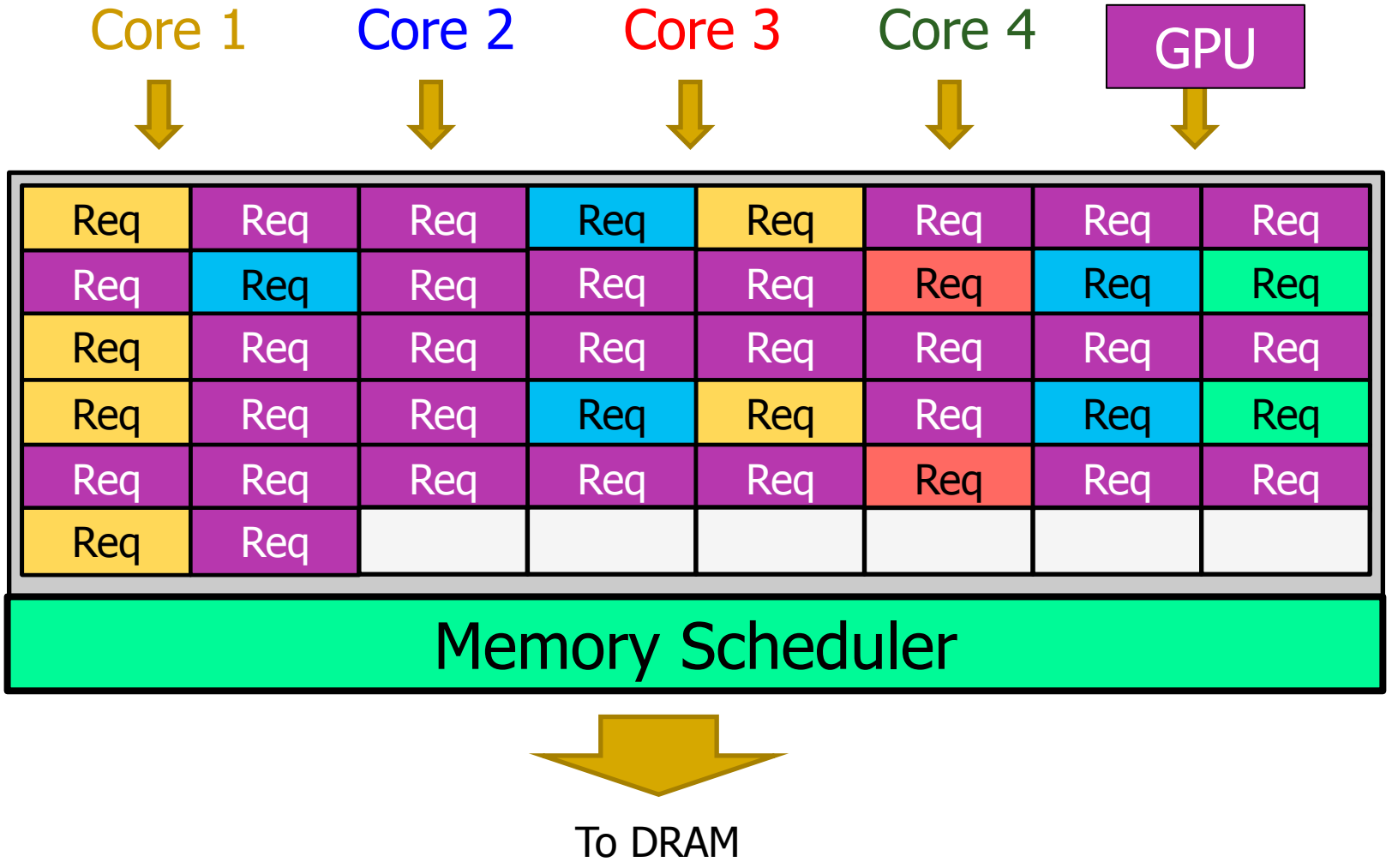
- Current and future systems integrate a GPU along with multiple cores
- GPU shares the main memory with the CPU cores
- GPU is much more (4x-20x) memory-intensive than CPU
- How should memory scheduling be done when GPU is integrated on-chip?

Introducing the GPU into the System

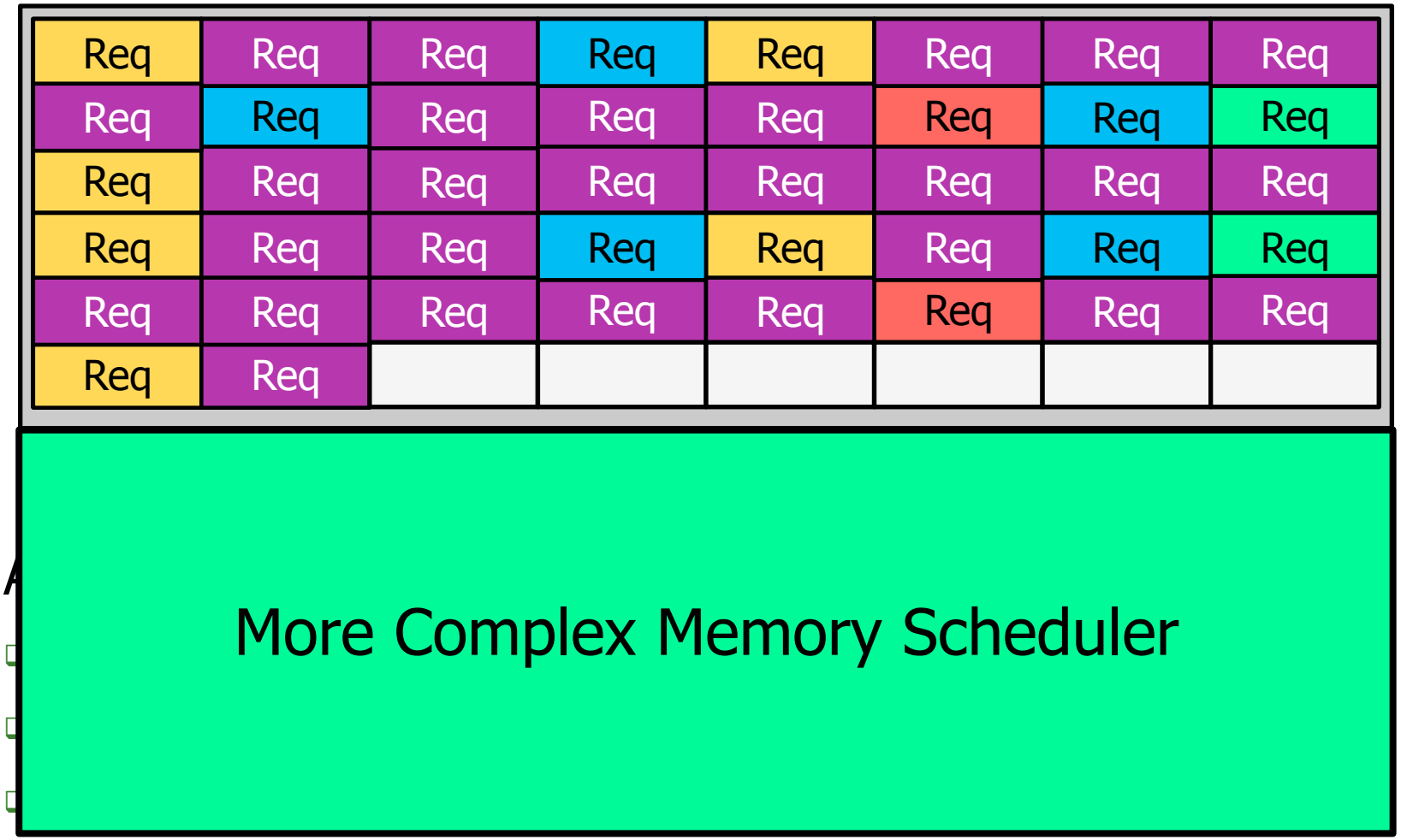


- GPU occupies a significant portion of the request buffers
  - Limits the MC's visibility of the CPU applications' differing memory behavior → can lead to a poor scheduling decision

Naïve Solution: Large Monolithic Buffer



Problems with Large Monolithic Buffer



- This leads to high complexity, high power, large die area

Our Goal

- Design a new memory scheduler that is:
  - Scalable to accommodate a large number of requests
  - Easy to implement
  - Application-aware
  - Able to provide high performance and fairness, especially in heterogeneous CPU-GPU systems

Key Functions of a Memory Controller

- Memory controller must consider three different things concurrently when choosing the next request:

1) Maximize row buffer hits
   - Maximize memory bandwidth
2) Manage contention between applications
   - Maximize system throughput and fairness
3) Satisfy DRAM timing constraints

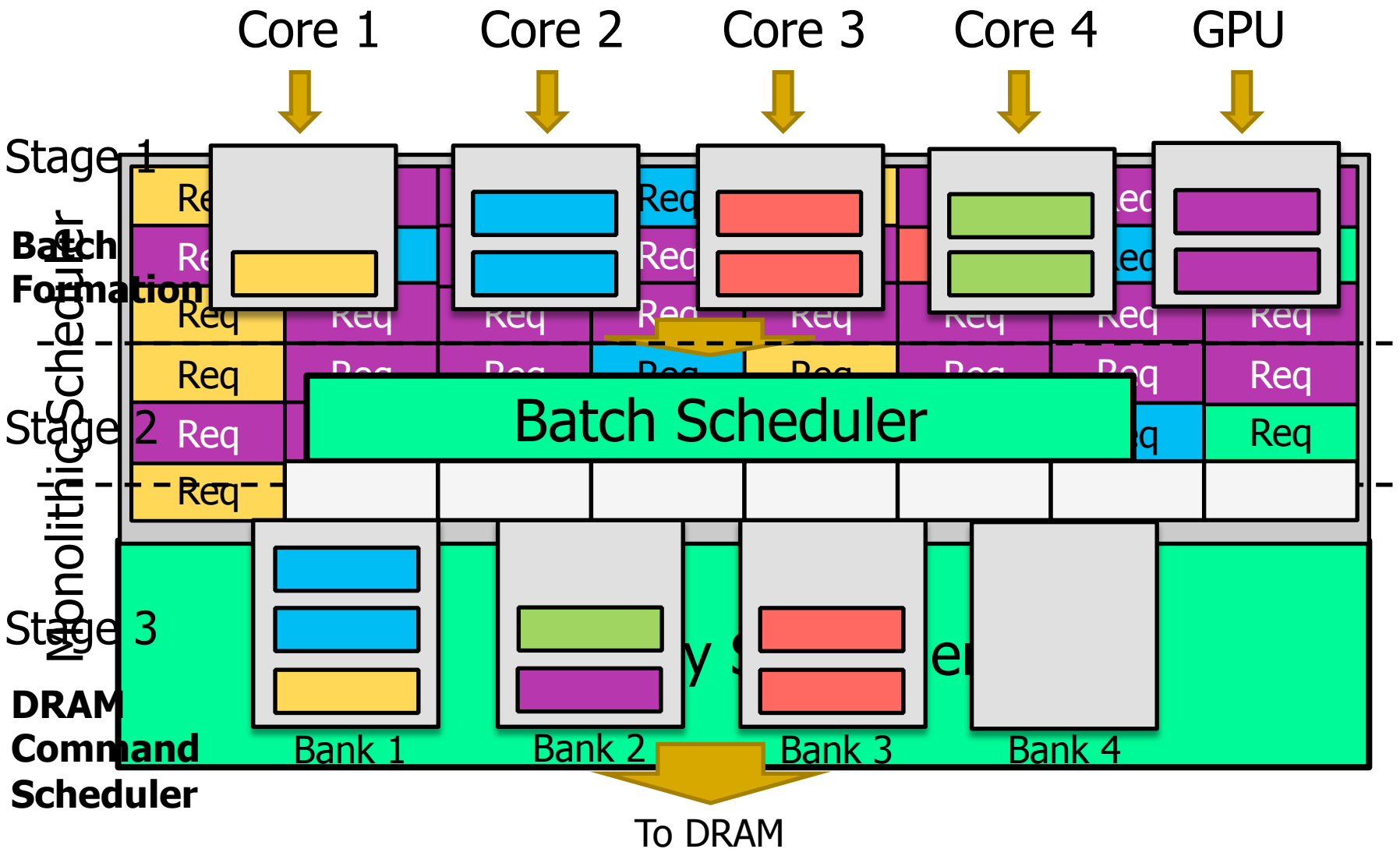
- Current systems use a **centralized memory controller design** to accomplish these functions
  - **Complex, especially with large request buffers**

Key Idea: Decouple Tasks into Stages

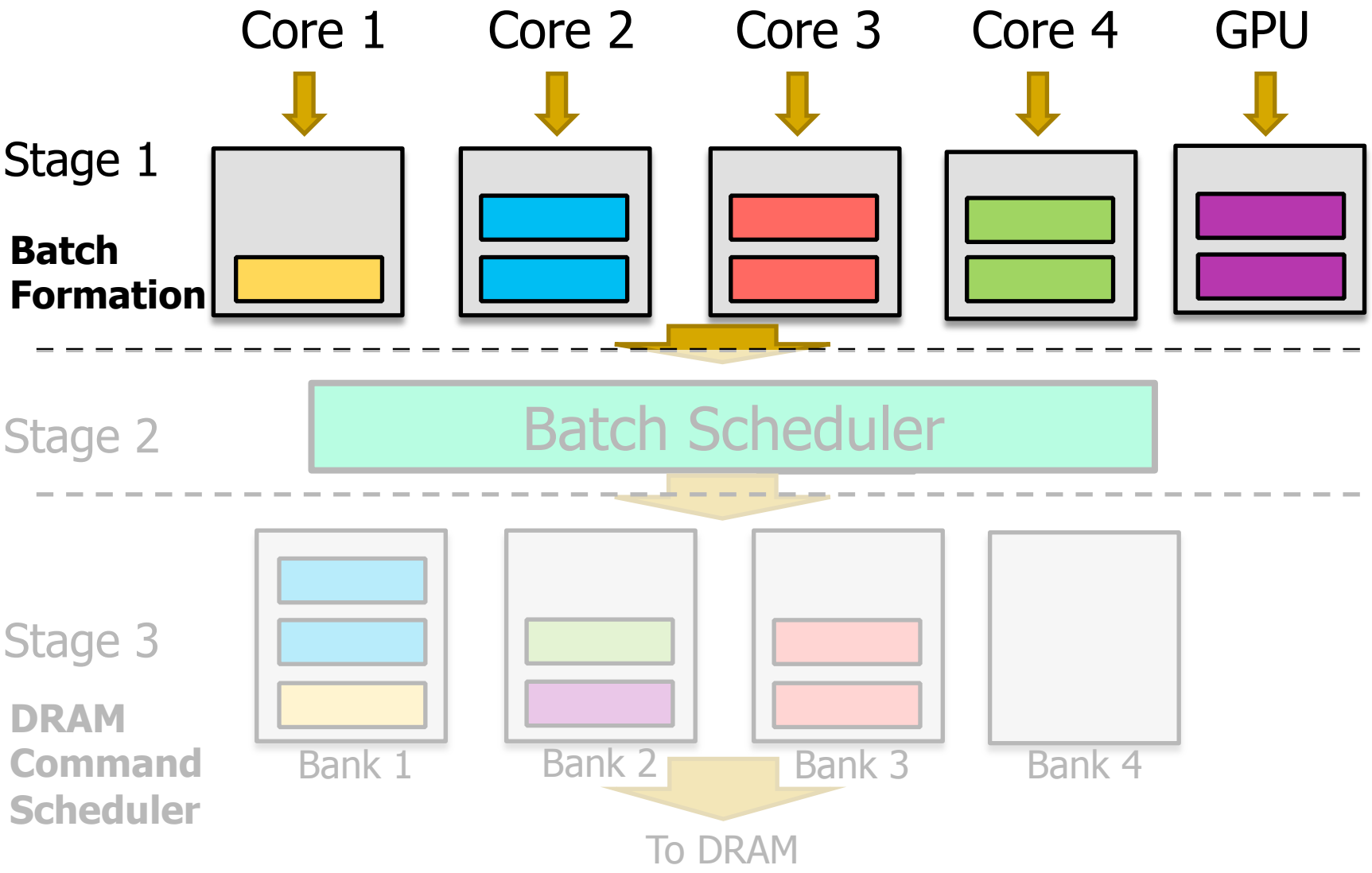
- Idea: Decouple the functional tasks of the memory controller
  - Partition tasks across several simpler HW structures (stages)

1) Maximize row buffer hits
   - Stage 1: Batch formation
   - Within each application, groups requests to the same row into batches
2) Manage contention between applications
   - Stage 2: Batch scheduler
   - Schedules batches from different applications
3) Satisfy DRAM timing constraints
   - Stage 3: DRAM command scheduler
   - Issues requests from the already-scheduled order to each bank

SMS: Staged Memory Scheduling



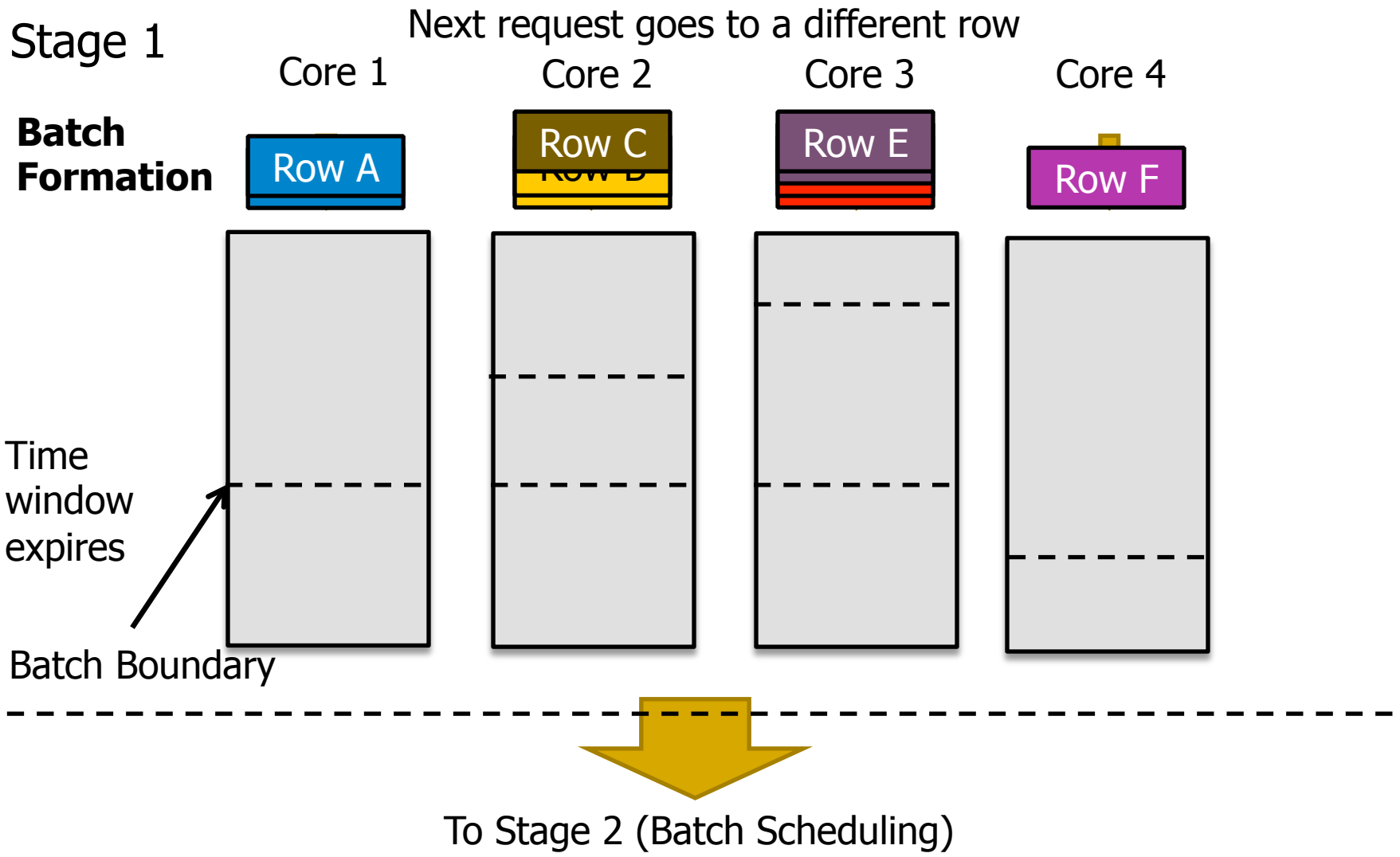
SMS: Staged Memory Scheduling



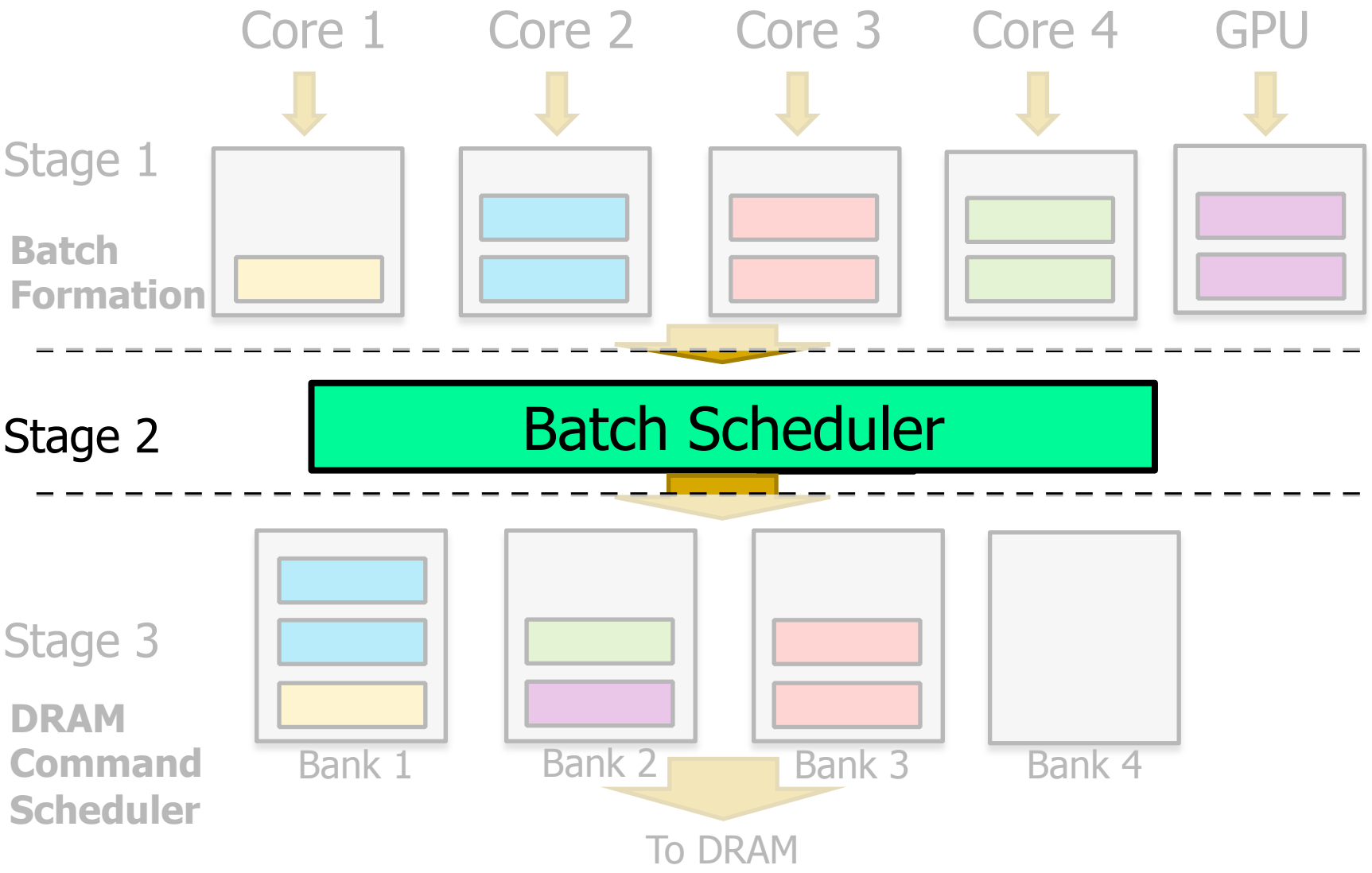
Stage 1: Batch Formation

- Goal: **Maximize row buffer hits**
- At each core, we want to batch requests that access the same row within a limited time window
- A batch is ready to be scheduled under two conditions
  1) When the next request accesses a different row
  2) When the time window for batch formation expires
- Keep this stage simple by using per-core FIFOs

Stage 1: Batch Formation Example



SMS: Staged Memory Scheduling



Stage 2: Batch Scheduler

- Goal: **Minimize interference between applications**

- Stage 1 forms batches within each application
- Stage 2 schedules batches from different applications
  - Schedules the oldest batch from each application

- Question: Which application's batch should be scheduled next?
- Goal: Maximize system performance and fairness
  - To achieve this goal, the batch scheduler chooses between two different policies

Stage 2: Two Batch Scheduling Algorithms

- **Shortest Job First (SJF)**
  - Prioritize the applications with the fewest outstanding memory requests because they make fast forward progress
  - **Pro:** Good system performance and fairness
  - **Con:** GPU and memory-intensive applications get deprioritized

- **Round-Robin (RR)**
  - Prioritize the applications in a round-robin manner to ensure that memory-intensive applications can make progress
  - **Pro:** GPU and memory-intensive applications are treated fairly
  - **Con:** GPU and memory-intensive applications significantly slow down others

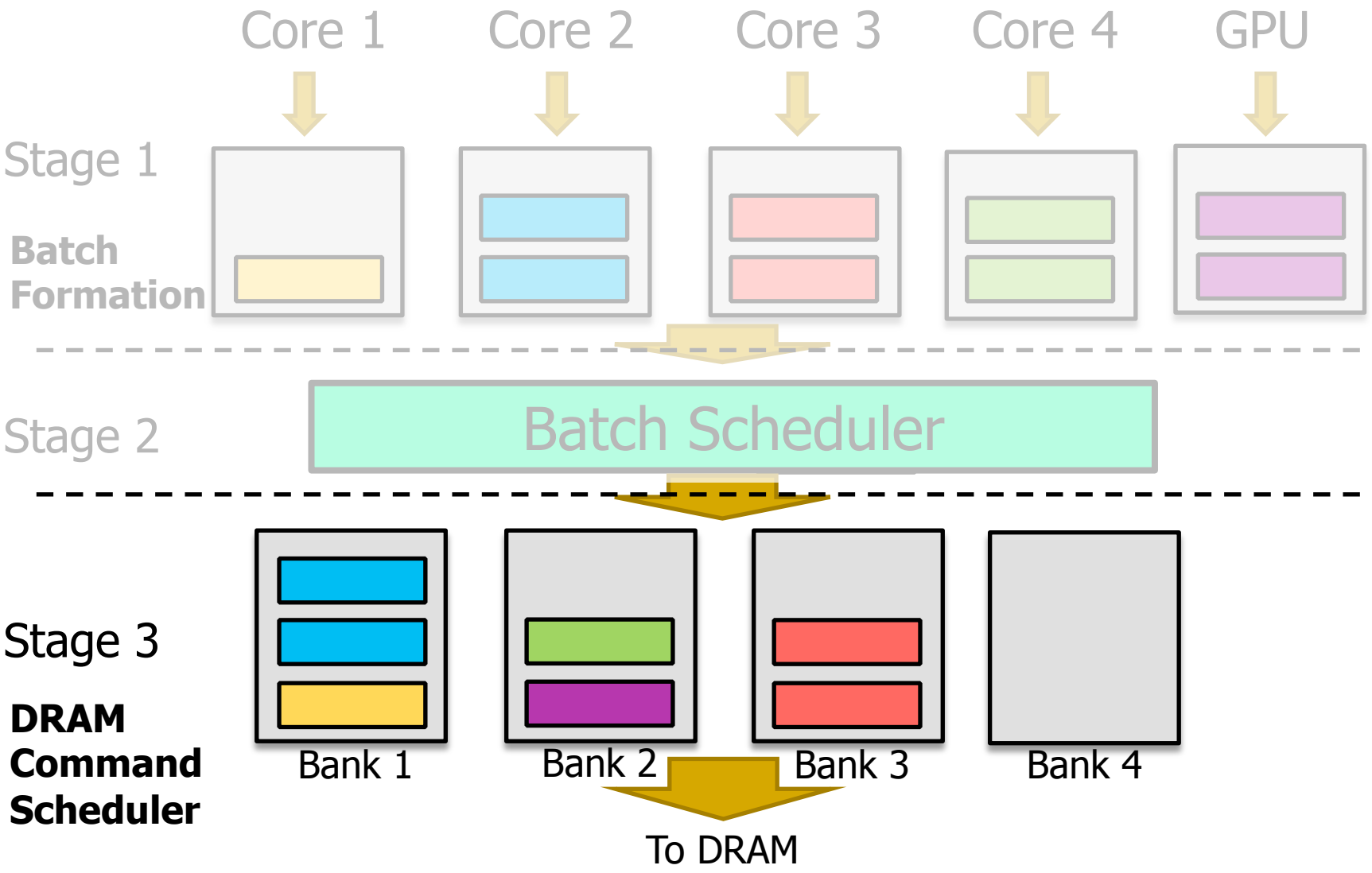
Stage 2: Batch Scheduling Policy

- The importance of the GPU varies between systems and over time → Scheduling policy needs to adapt to this

- Solution: Hybrid Policy
- At every cycle:
  - With probability *p* : Shortest Job First → Benefits the CPU
  - With probability *1-p* : Round-Robin → Benefits the GPU

- System software can configure *p* based on the importance/ weight of the GPU
  - Higher GPU importance → Lower *p* value

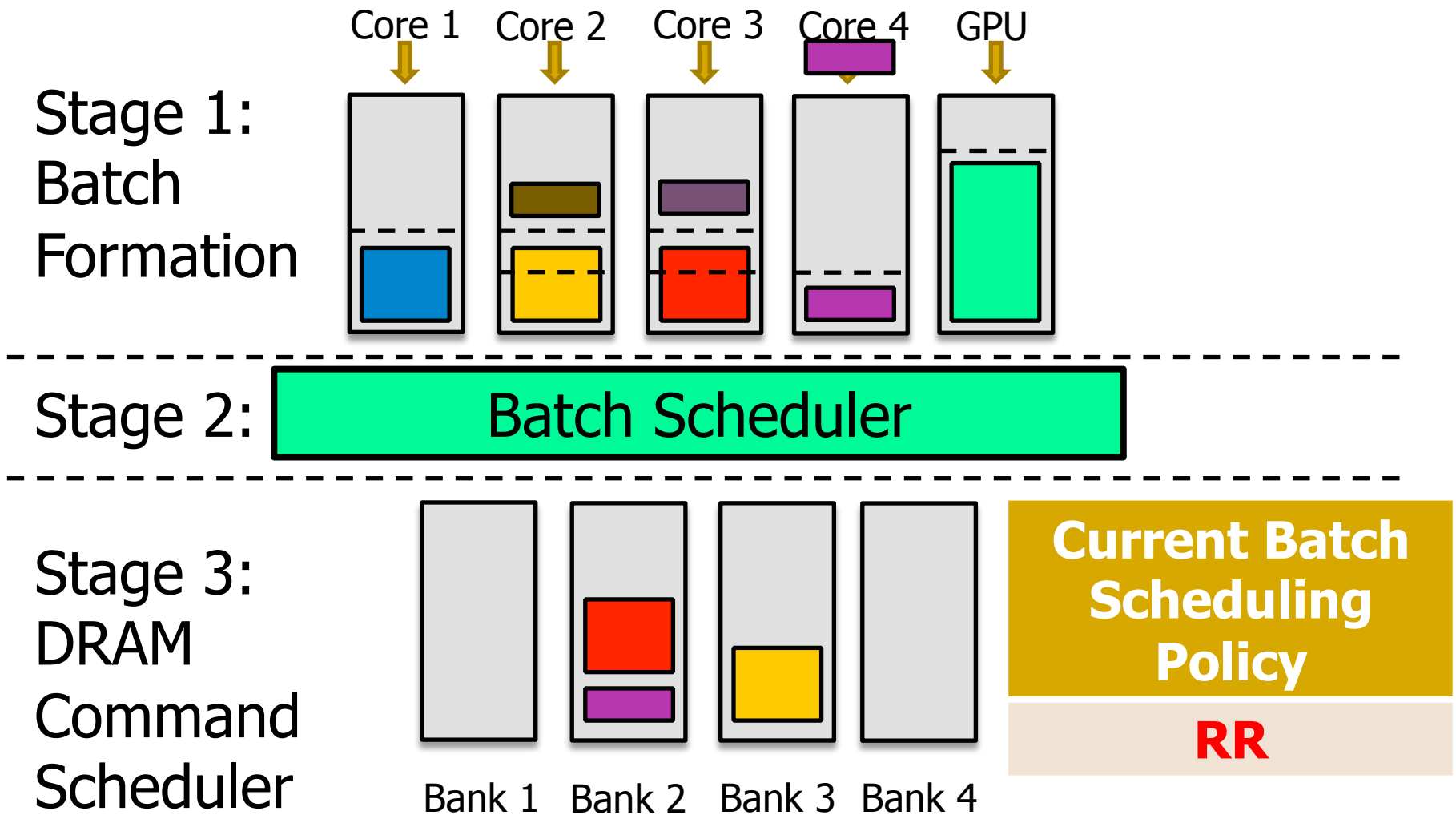
SMS: Staged Memory Scheduling



Stage 3: DRAM Command Scheduler

- High level policy decisions have already been made by:
  - Stage 1: Maintains row buffer locality
  - Stage 2: Minimizes inter-application interference
- Stage 3: No need for further scheduling
- Only goal: **service requests while satisfying DRAM timing constraints**
- Implemented as **simple per-bank FIFO queues**

Putting Everything Together



Complexity

- Compared to a row hit first scheduler, SMS consumes*
  - 66% less area
  - 46% less static power

- Reduction comes from:
  - Monolithic scheduler → stages of simpler schedulers
  - Each stage has a simpler scheduler (considers fewer properties at a time to make the scheduling decision)
  - Each stage has simpler buffers (FIFO instead of out-of-order)
  - Each stage has a portion of the total buffer size (buffering is distributed across stages)

*** Based on a Verilog model using 180nm library**

Methodology

- Simulation parameters
  - 16 OoO CPU cores, 1 GPU modeling AMD Radeon™ 5870
  - DDR3-1600 DRAM 4 channels, 1 rank/channel, 8 banks/channel

- Workloads
  - CPU: SPEC CPU 2006
  - GPU: Recent games and GPU benchmarks
  - 7 workload categories based on the memory-intensity of CPU applications
    → Low memory-intensity (L)
    → Medium memory-intensity (M)
    → High memory-intensity (H)

Comparison to Previous Scheduling Algorithms

- FR-FCFS [Rixner+, ISCA'00]
  - Prioritizes row buffer hits
  - Maximizes DRAM throughput
  - Low multi-core performance ← Application unaware
- ATLAS [Kim+, HPCA'10]
  - Prioritizes latency-sensitive applications
  - Good multi-core performance
  - Low fairness ← Deprioritizes memory-intensive applications
- TCM [Kim+, MICRO'10]
  - Clusters low and high-intensity applications and treats each separately
  - Good multi-core performance and fairness
  - Not robust ← Misclassifies latency-sensitive applications

Evaluation Metrics

- CPU performance metric: Weighted speedup

$$CPU_{WS} = \sum \frac{IPC_{Shared}}{IPC_{Alone}}$$

- GPU performance metric: Frame rate speedup

$$GPU_{Speedup} = \frac{FrameRate_{Shared}}{FrameRate_{Alone}}$$

- CPU-GPU system performance: CPU-GPU weighted speedup

$$CGWS = CPU_{WS} + GPU_{Speedup} * GPU_{Weight}$$

Evaluated System Scenario: CPU Focused

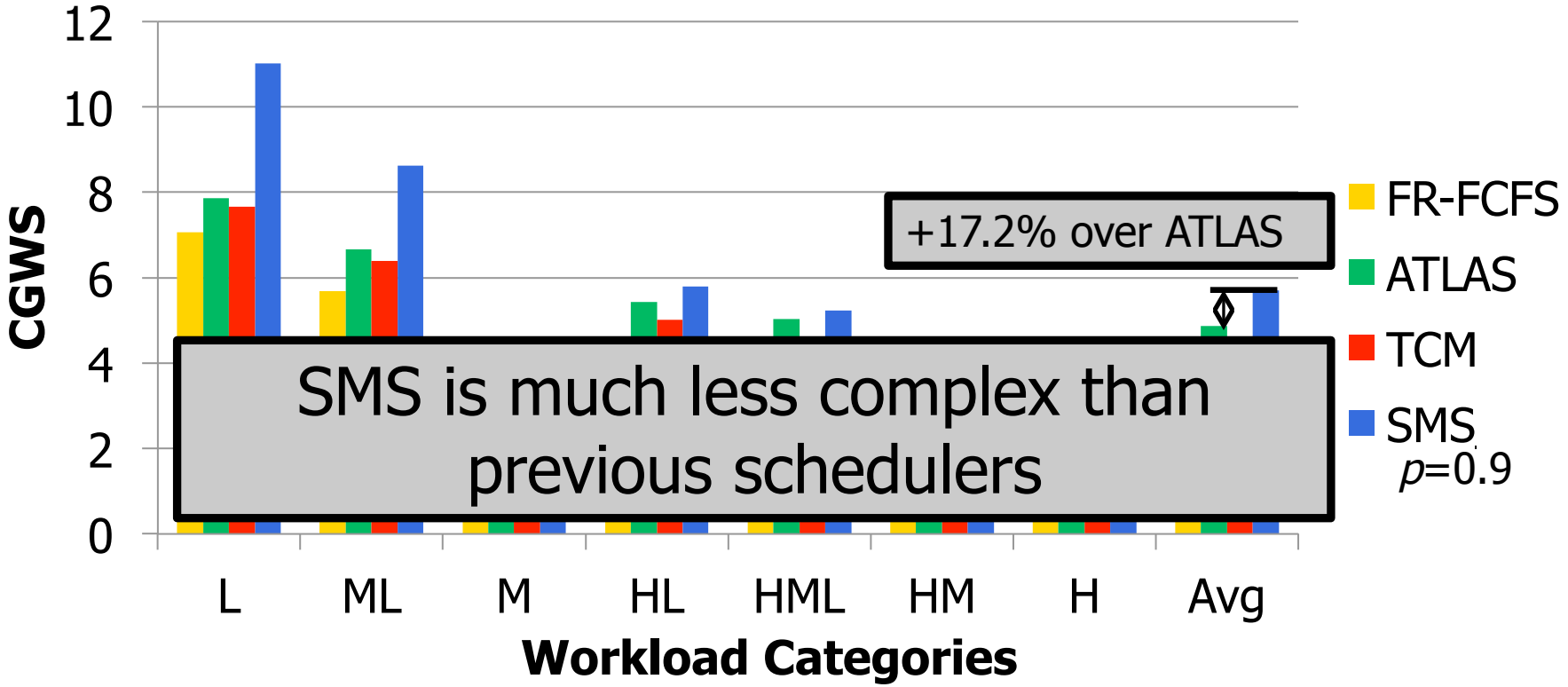
- GPU has low weight (weight = 1)

$$CGWS = CPU_{WS} + GPU_{Speedup} * GPU_{Weight}$$

1

- Configure SMS such that *p*, SJF probability, is set to 0.9
  - Mostly uses SJF batch scheduling → prioritizes latency-sensitive applications (mainly CPU)

Performance: CPU-Focused System



- SJF batch scheduling policy allows latency-sensitive applications to get serviced as fast as possible

Evaluated System Scenario: GPU Focused

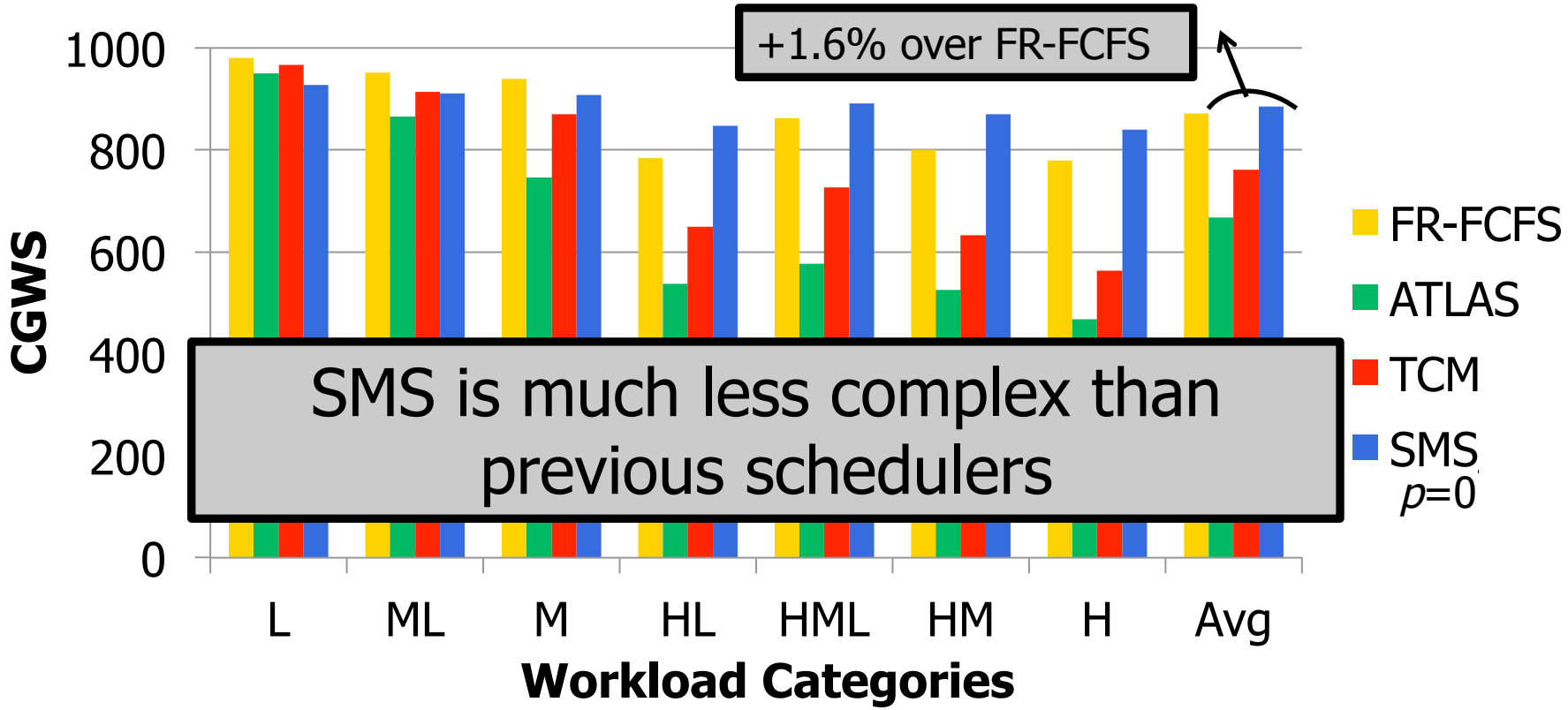
- GPU has high weight (weight = 1000)

$$CGWS = CPU_{WS} + GPU_{Speedup} * GPU_{Weight}$$

1000

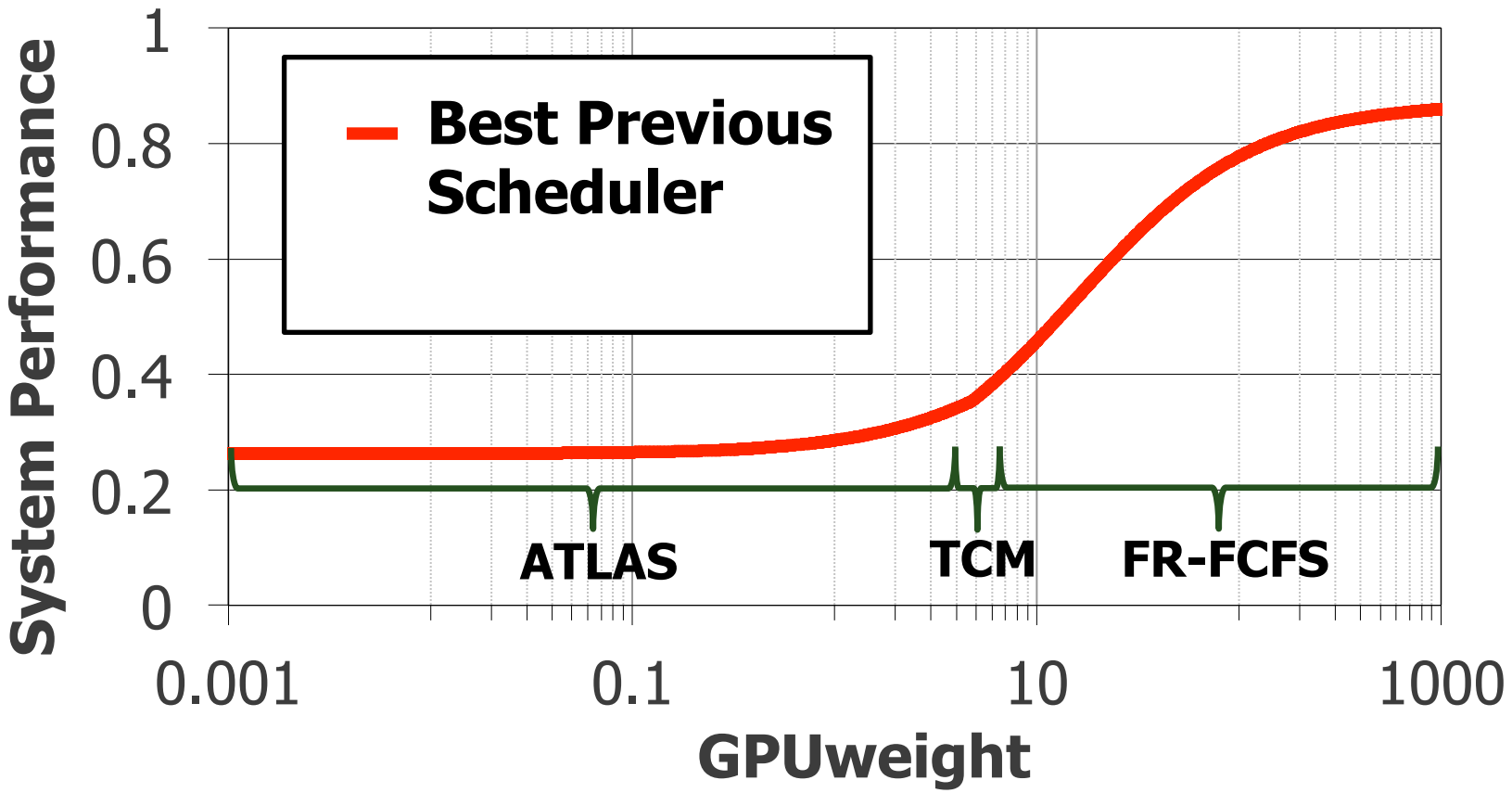
- Configure SMS such that *p*, SJF probability, is set to 0
  - Always uses round-robin batch scheduling → prioritizes memory-intensive applications (GPU)

Performance: GPU-Focused System

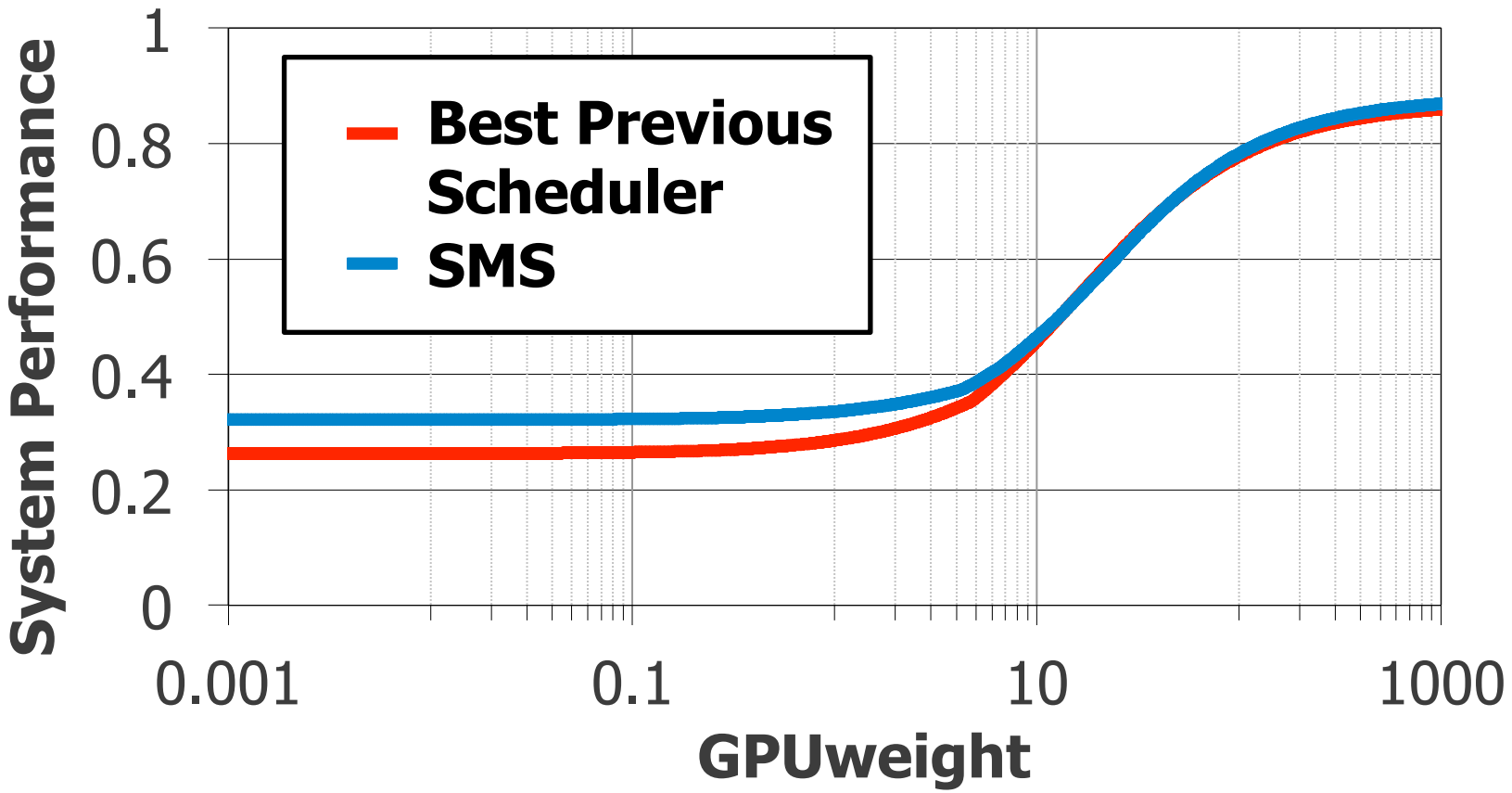


- Round-robin batch scheduling policy schedules GPU requests more frequently

Performance at Different GPU Weights



Performance at Different GPU Weights



- At every GPU weight, SMS outperforms the best previous scheduling algorithm for that weight

Additional Results in the Paper

- Fairness evaluation
  - 47.6% improvement over the best previous algorithms
- Individual CPU and GPU performance breakdowns
- CPU-only scenarios
  - Competitive performance with previous algorithms
- Scalability results
  - SMS' performance and fairness scales better than previous algorithms as the number of cores and memory channels increases
- Analysis of SMS design parameters

Conclusion

- **Observation:** Heterogeneous CPU-GPU systems require memory schedulers with large request buffers
- **Problem:** Existing monolithic application-aware memory scheduler designs are hard to scale to large request buffer size
- **Solution:** Staged Memory Scheduling (SMS)

  decomposes the memory controller into three simple stages:

  1) Batch formation: maintains row buffer locality

  2) Batch scheduler: reduces interference between applications

  3) DRAM command scheduler: issues requests to DRAM
- Compared to state-of-the-art memory schedulers:
  - SMS is significantly simpler and more scalable
  - SMS provides higher performance and fairness