18-742 Fall 2012
Parallel Computer Architecture
Lecture 26: Memory Interference Mitigation

Prof. Onur Mutlu
Carnegie Mellon University
11/14/2012

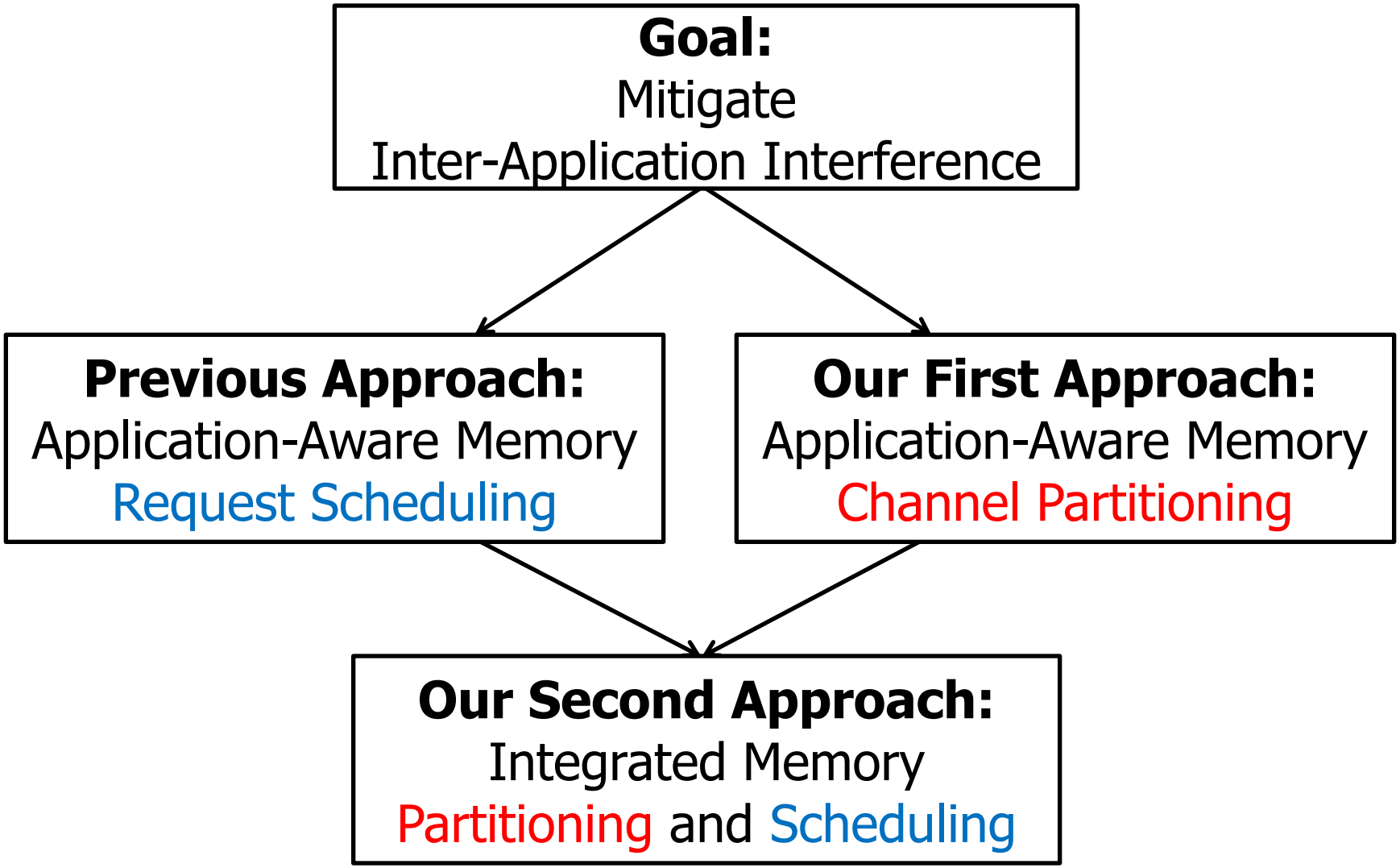
Designing QoS-Aware Memory Systems: Approaches

- 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]
  - 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
  - QoS-aware data mapping to memory controllers [Muralidhara+ MICRO'11]
  - Source throttling to control access to memory system [Ebrahimi+ ASPLOS'10, ISCA'11, TOCS'12] [Ebrahimi+ MICRO'09] [Nychis+ HotNets'10]
  - QoS-aware thread scheduling to cores

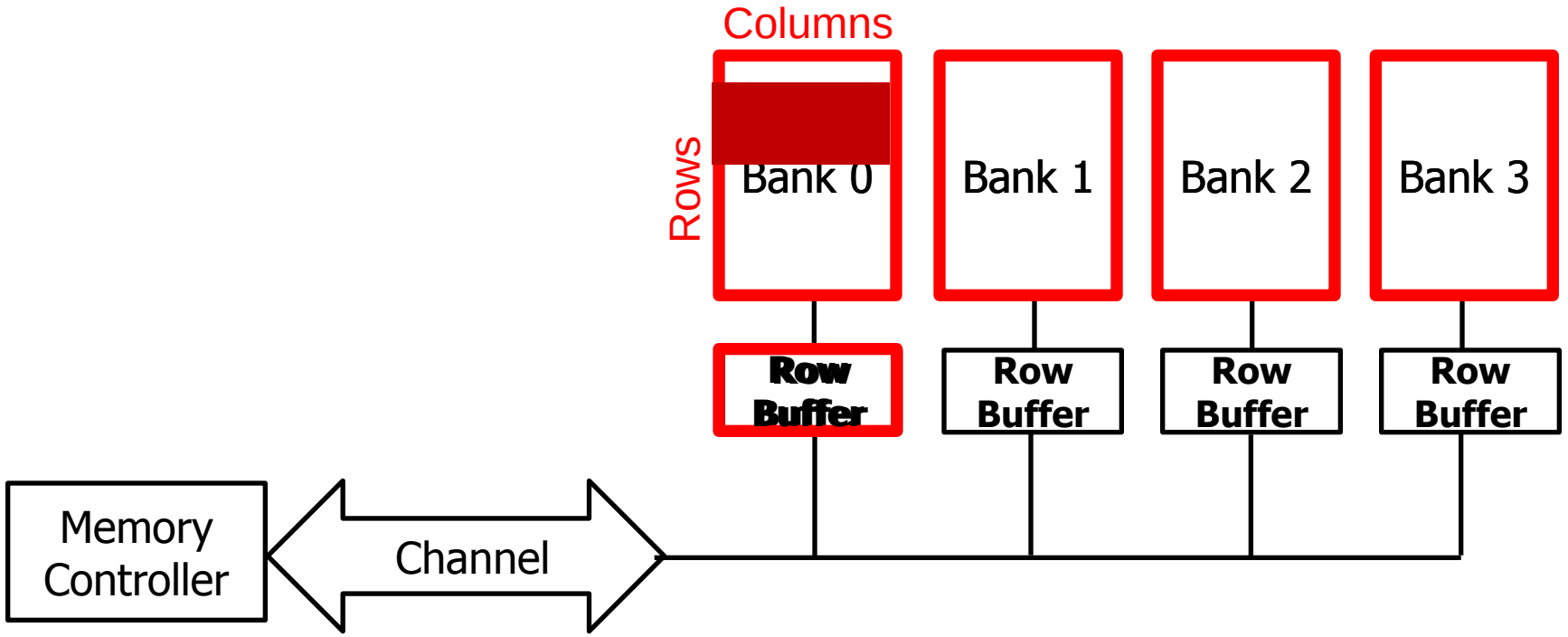
Memory Channel Partitioning

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)

Outline

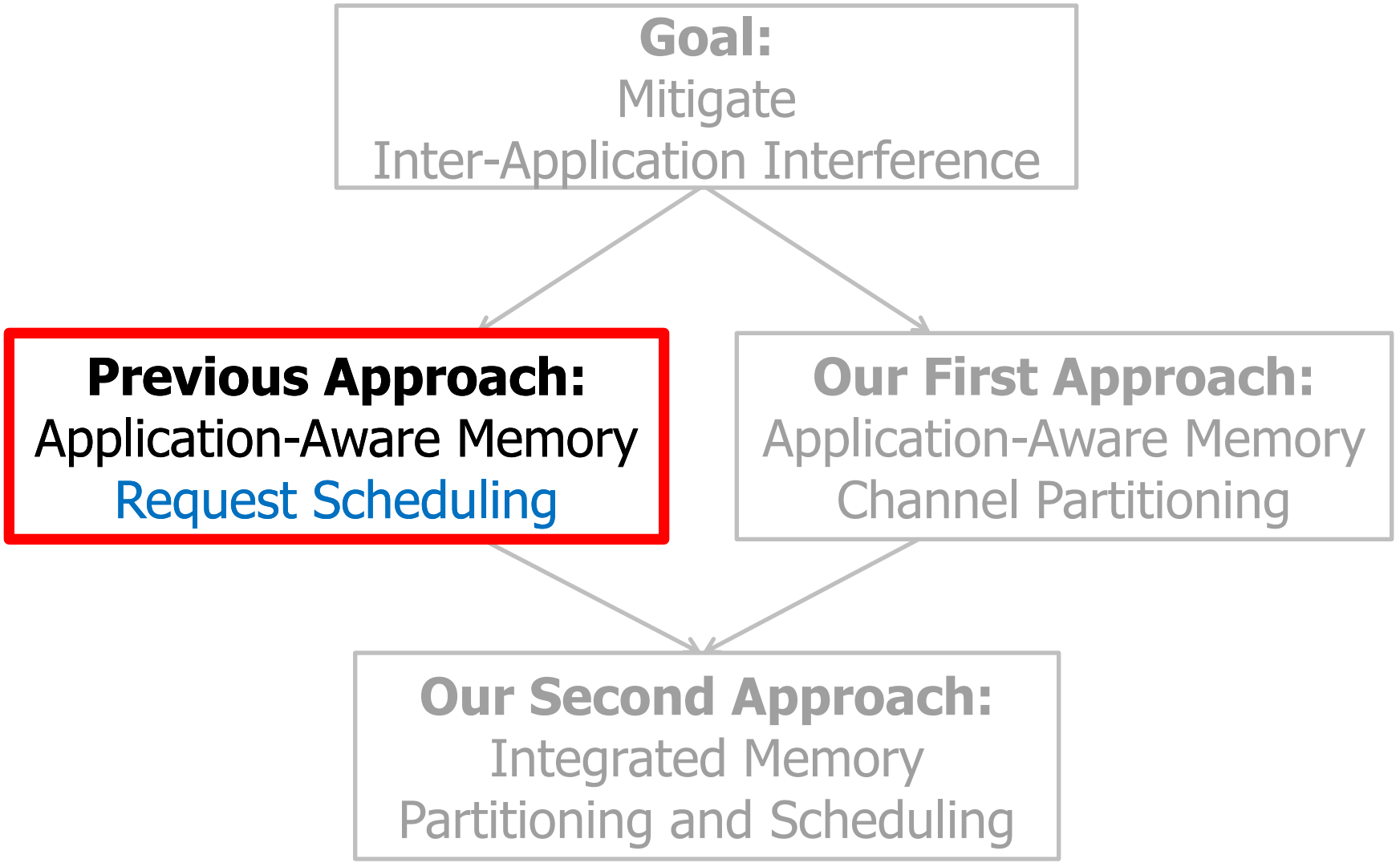

Goal:
Mitigate
Inter-Application Interference
Previous Approach:
Application-Aware Memory
Request Scheduling
Our First Approach:
Application-Aware Memory
Channel Partitioning
Our Second Approach:
Integrated Memory
Partitioning and Scheduling

Background: Main Memory



- FR-FCFS memory scheduling policy [Zuravleff et al., US Patent '97; Rixner et al., ISCA '00]
  - Row-buffer hit first
  - Oldest request first
- Unaware of inter-application interference

Previous Approach



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

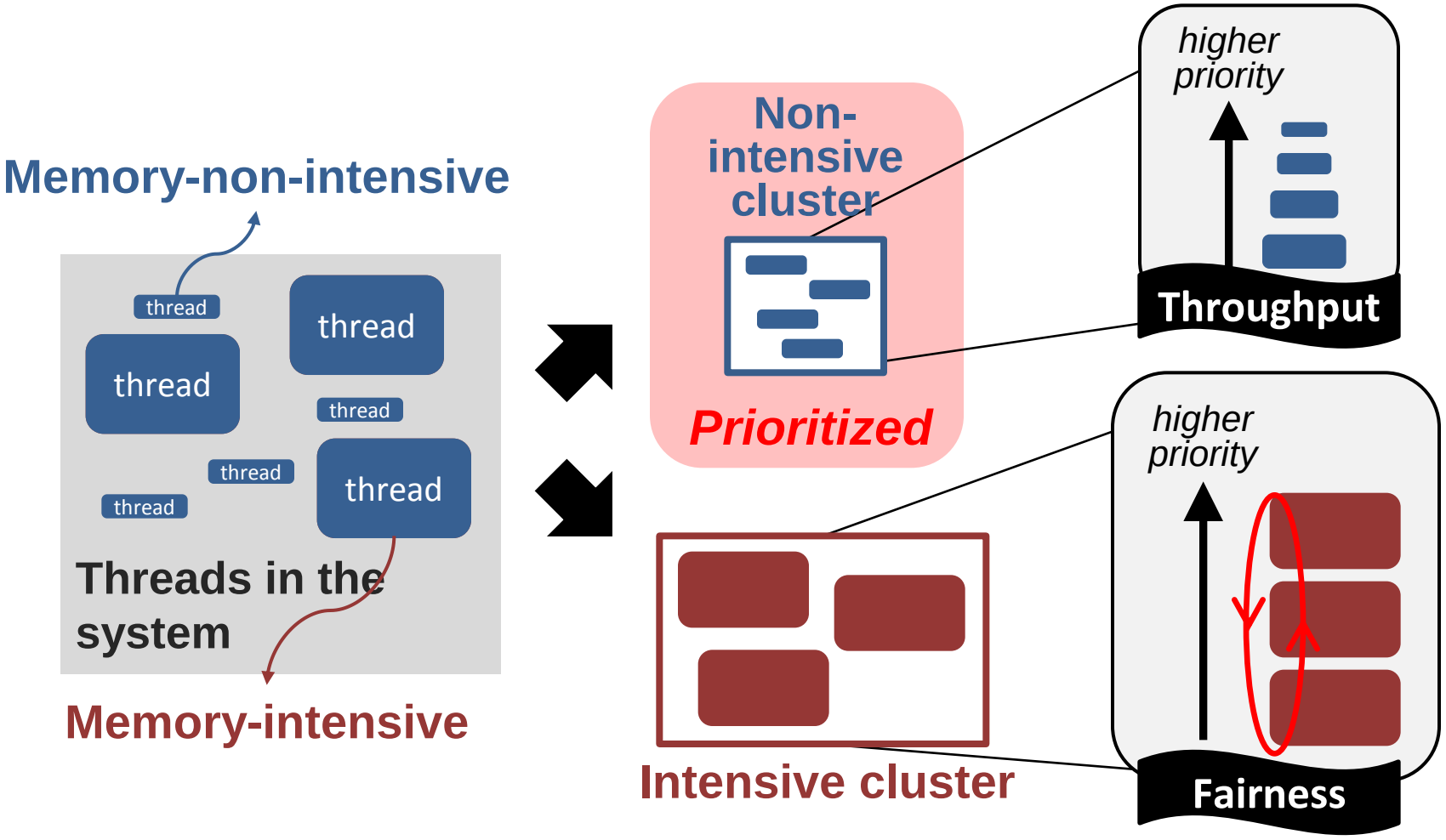


Figure: Kim et al., MICRO 2010

Application-Aware Memory Request Scheduling

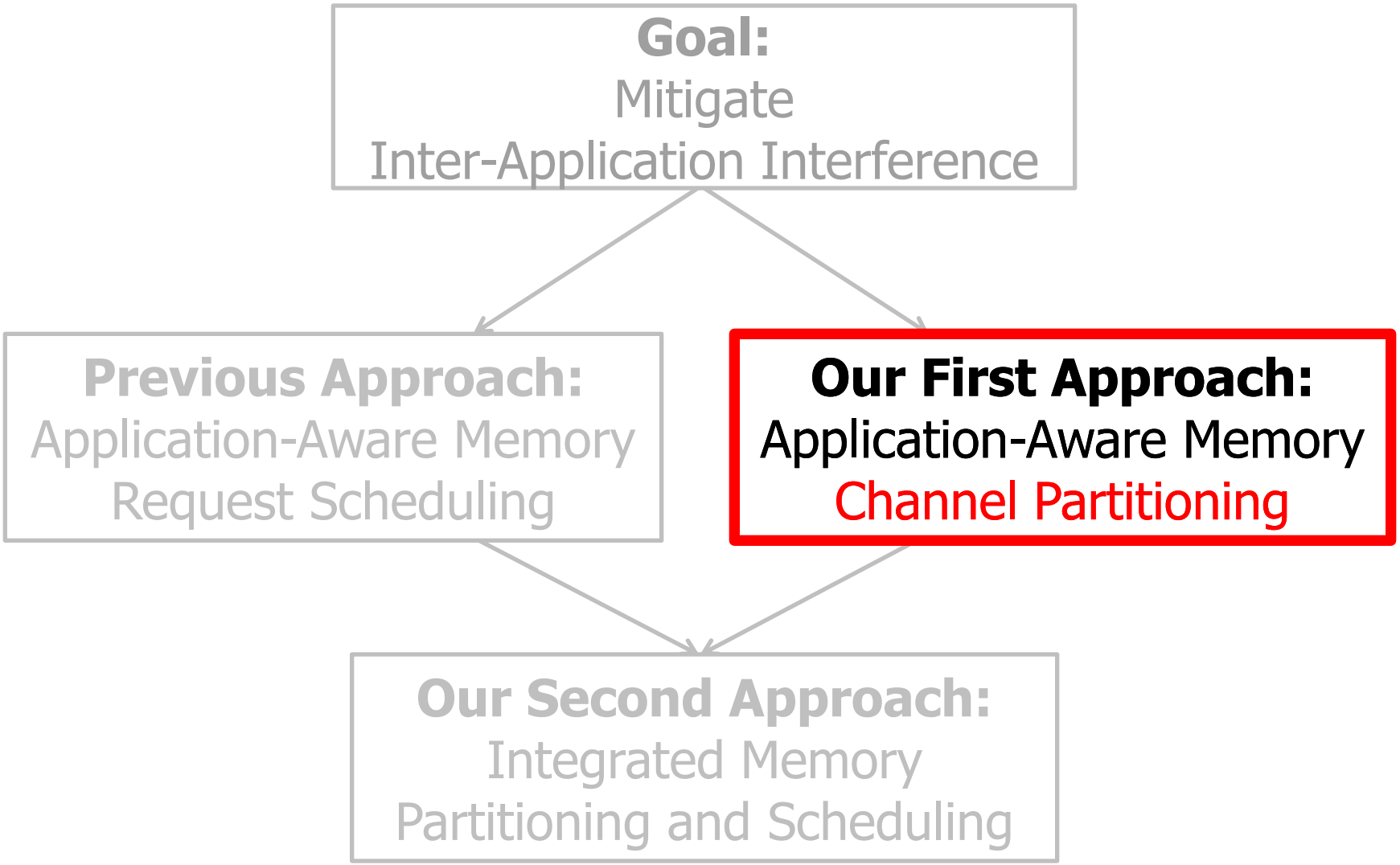
Advantages

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

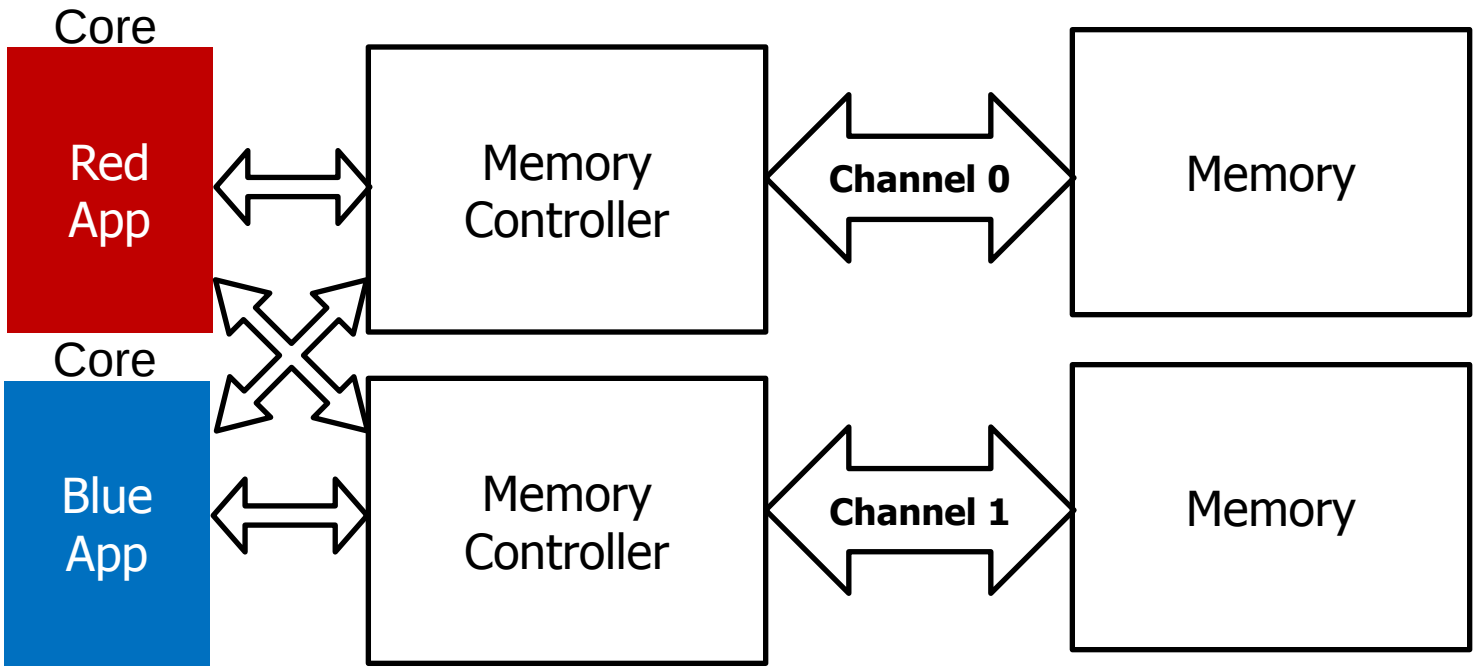
Disadvantages

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

Our Approach



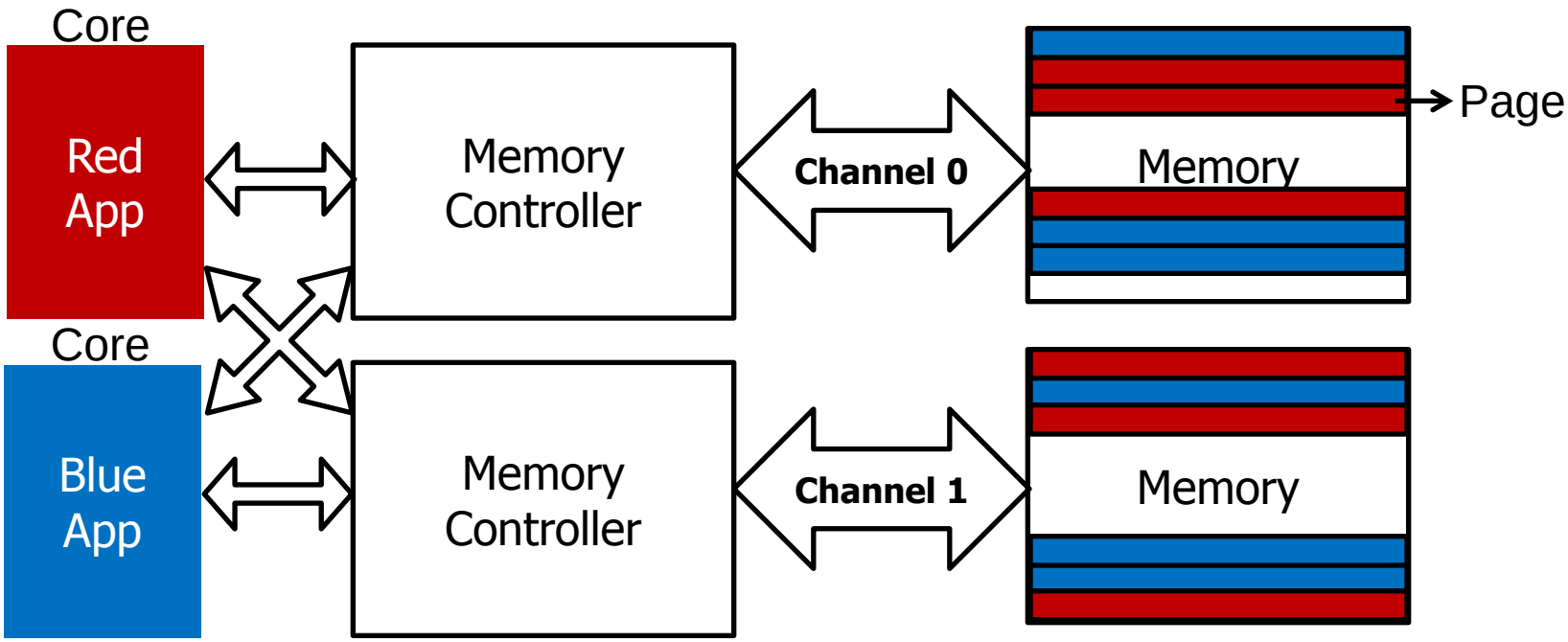
Observation: Modern Systems Have Multiple Channels



A new degree of freedom

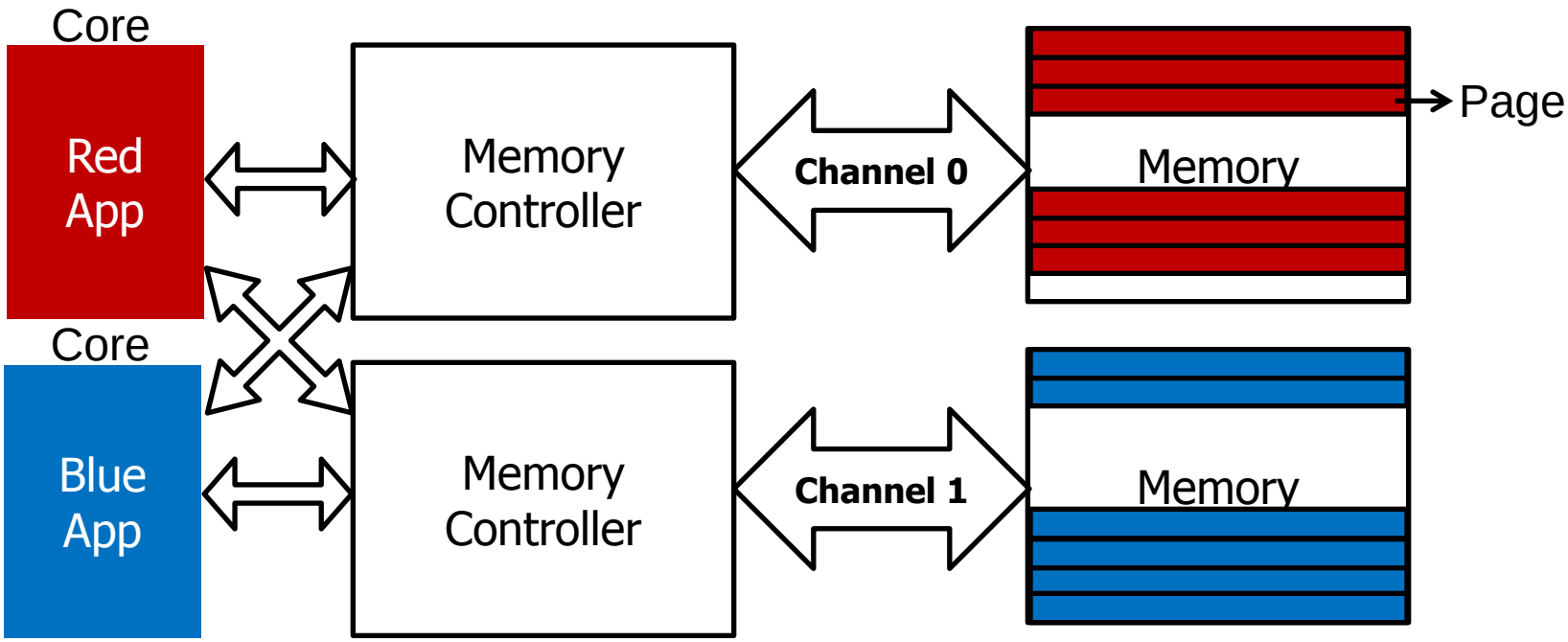
Mapping data across multiple channels

Data Mapping in Current Systems



Causes interference between applications' requests

Partitioning Channels Between Applications



Eliminates interference between applications' requests

Overview: Memory Channel Partitioning (MCP)

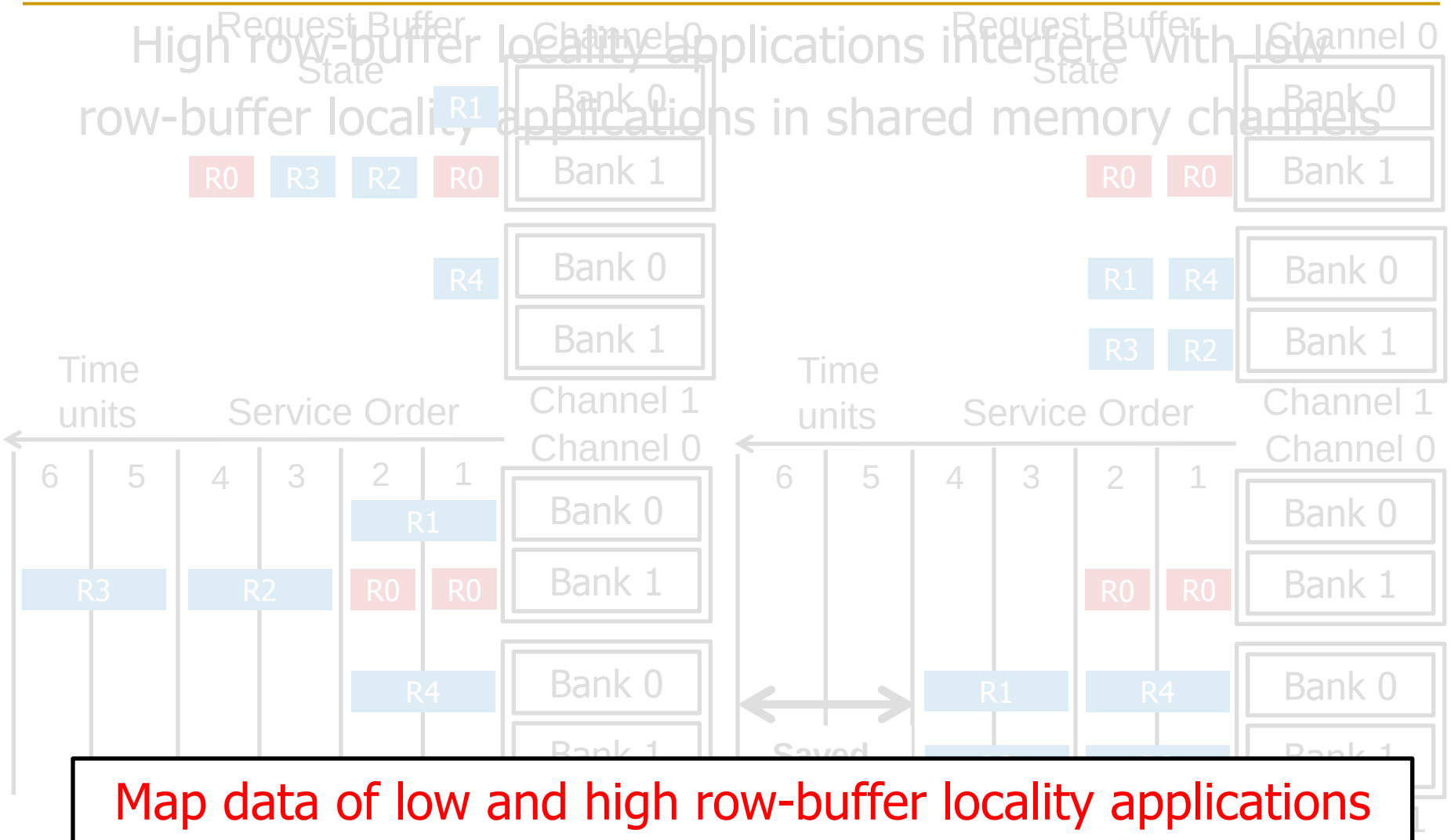
- Goal
  - Eliminate harmful interference between applications
- Basic Idea
  - Map the data of badly-interfering applications to different channels
- Key Principles
  - Separate low and high memory-intensity applications
  - Separate low and high row-buffer locality applications

Key Insight 1: Separate by Memory Intensity

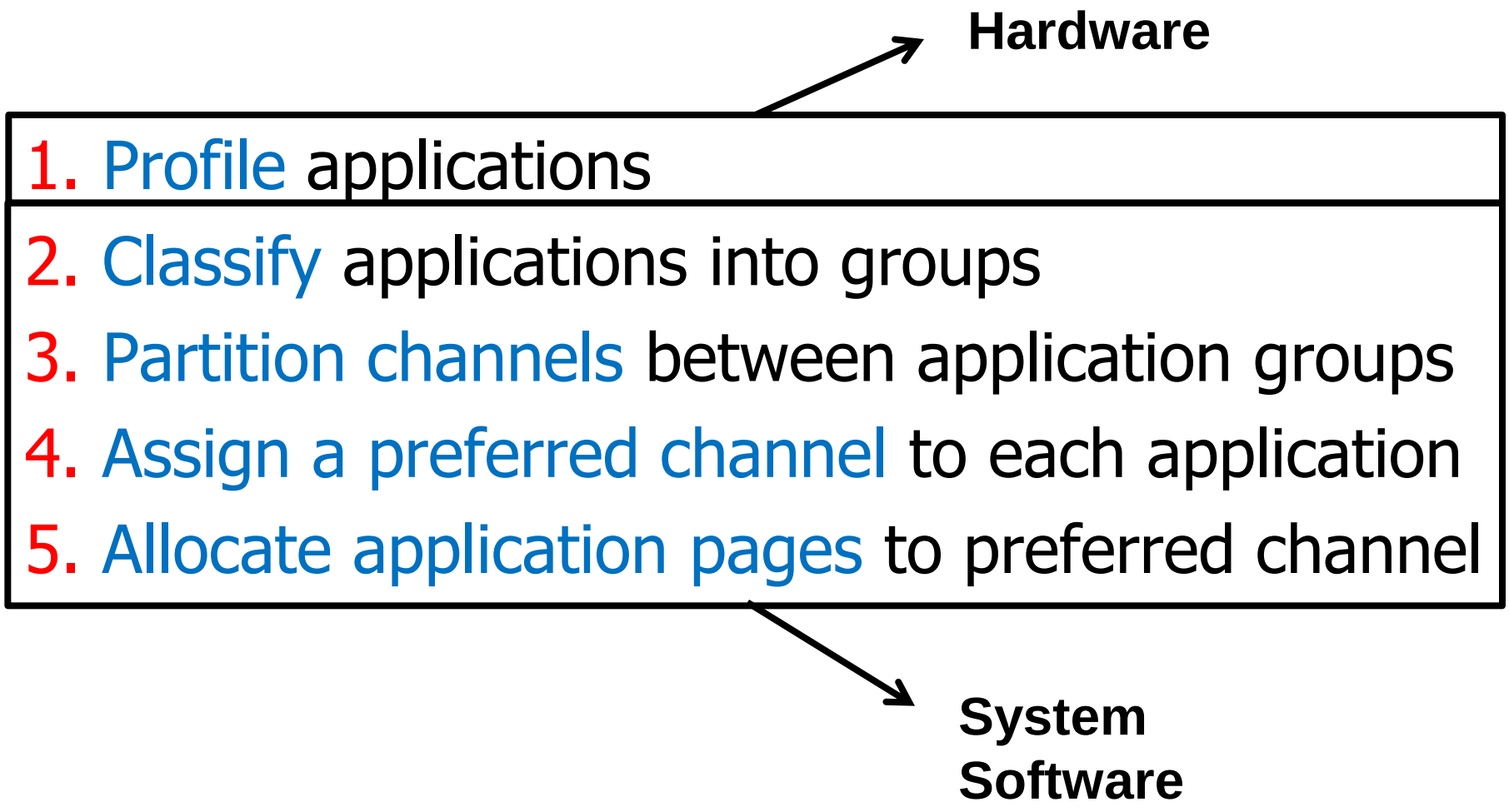
High memory-intensity applications interfere with low memory-intensity applications in shared memory channels

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

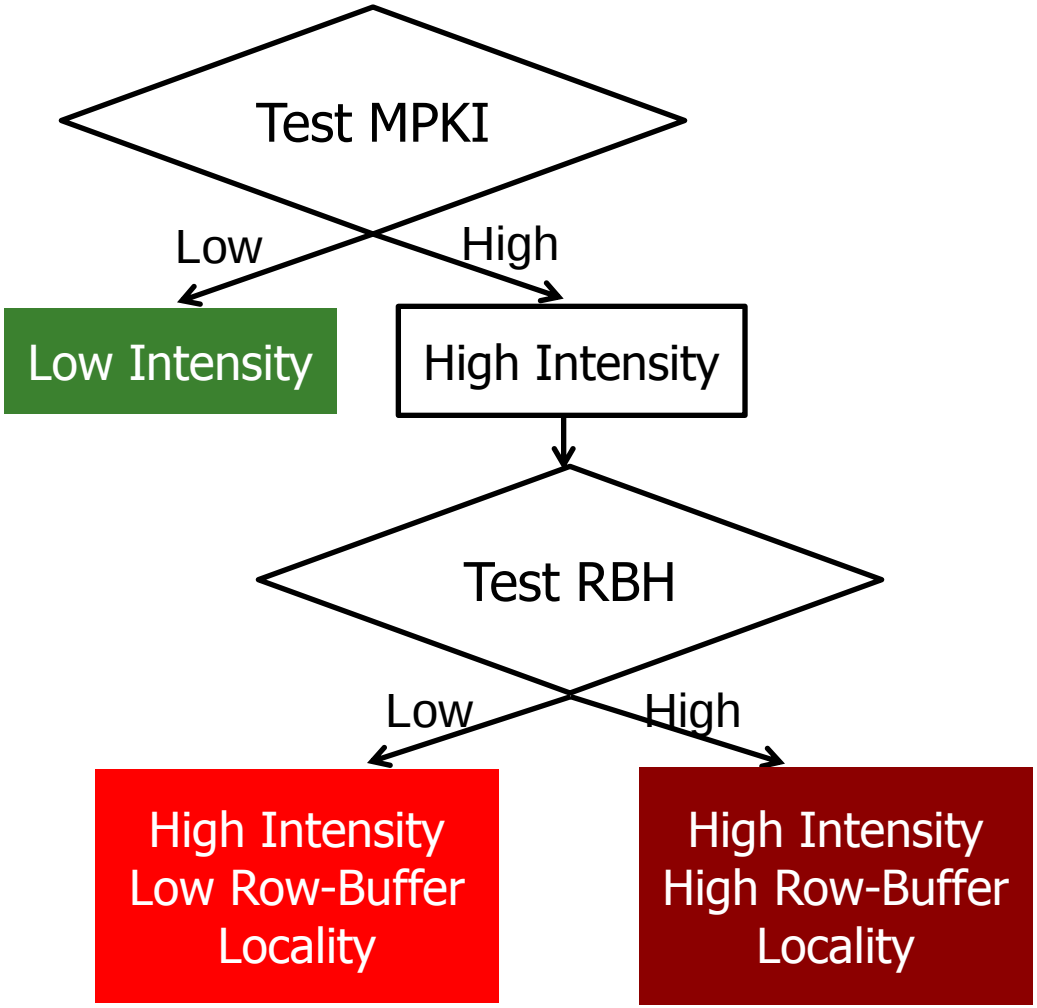
- Hardware counters collect application memory access characteristics

- Memory access characteristics
  - Memory intensity:

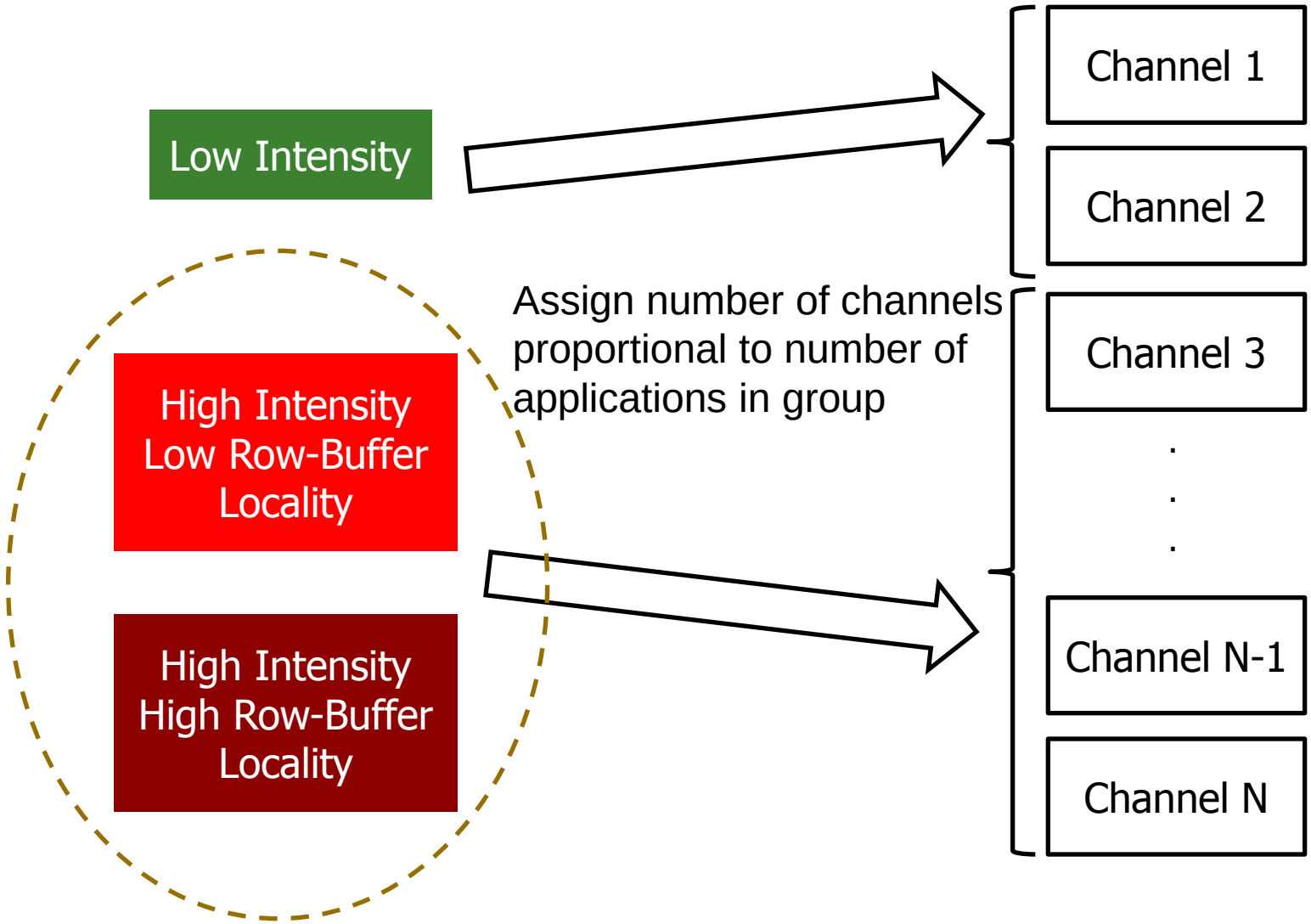
    Last level cache Misses Per Kilo Instruction (MPKI)
  - Row-buffer locality:

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

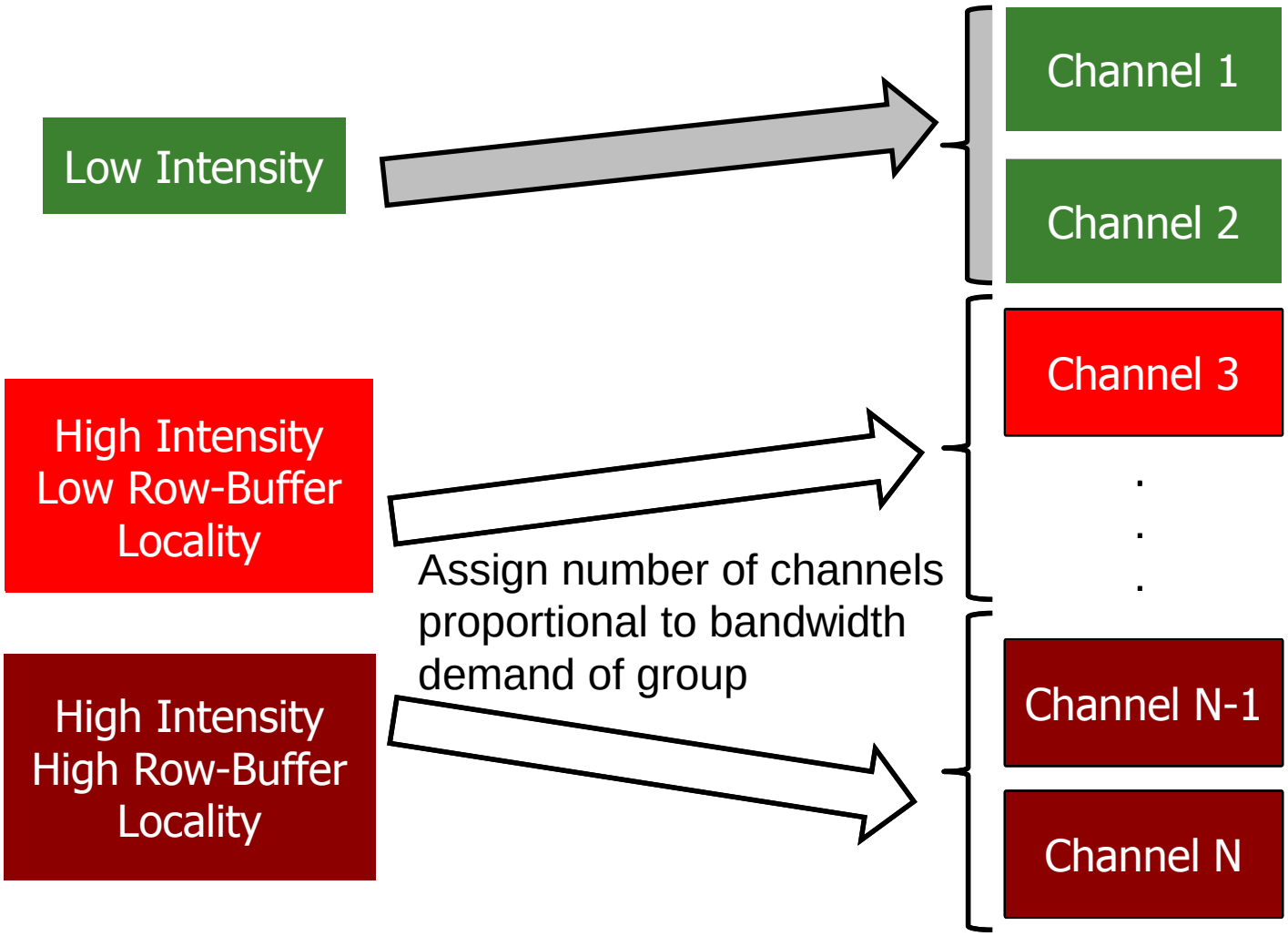
2. Classify Applications



3. Partition Channels Among Groups: Step 1

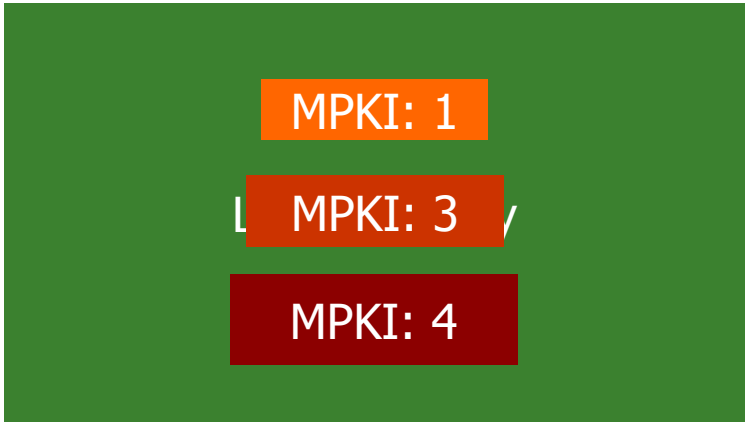


3. Partition Channels Among Groups: Step 2



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

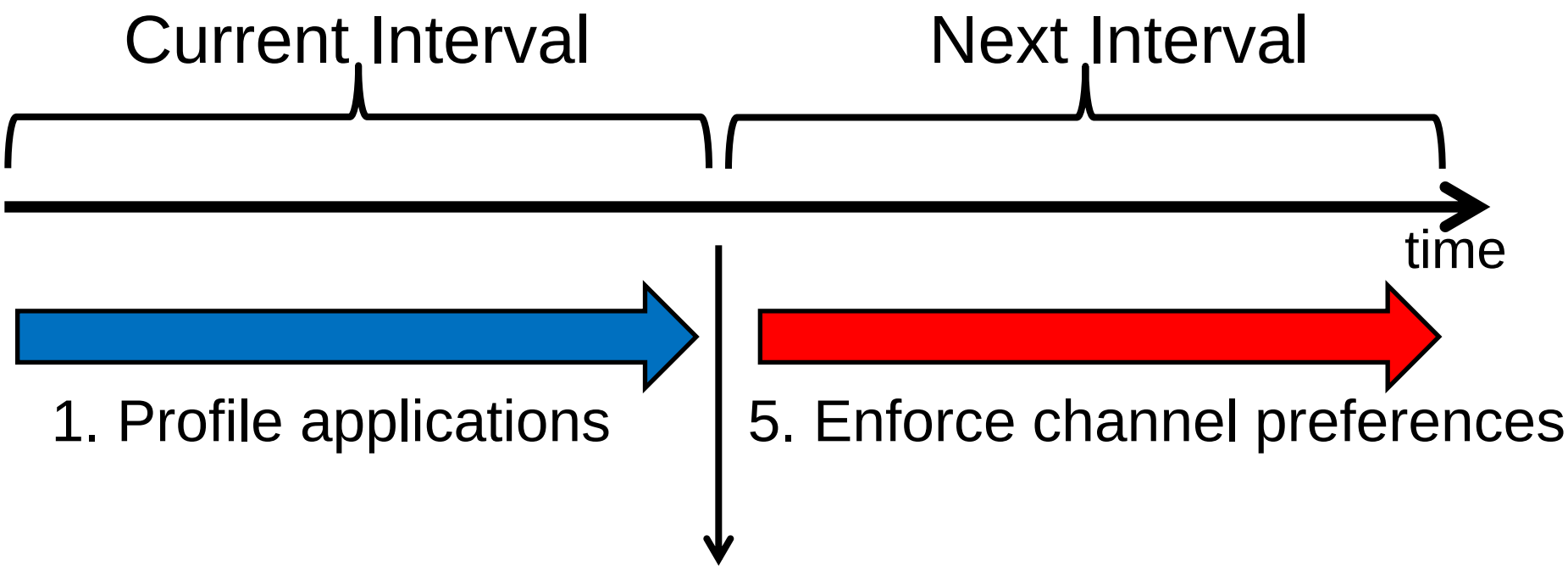


5. Allocate Page to Preferred Channel

- Enforce channel preferences
  computed in the previous step

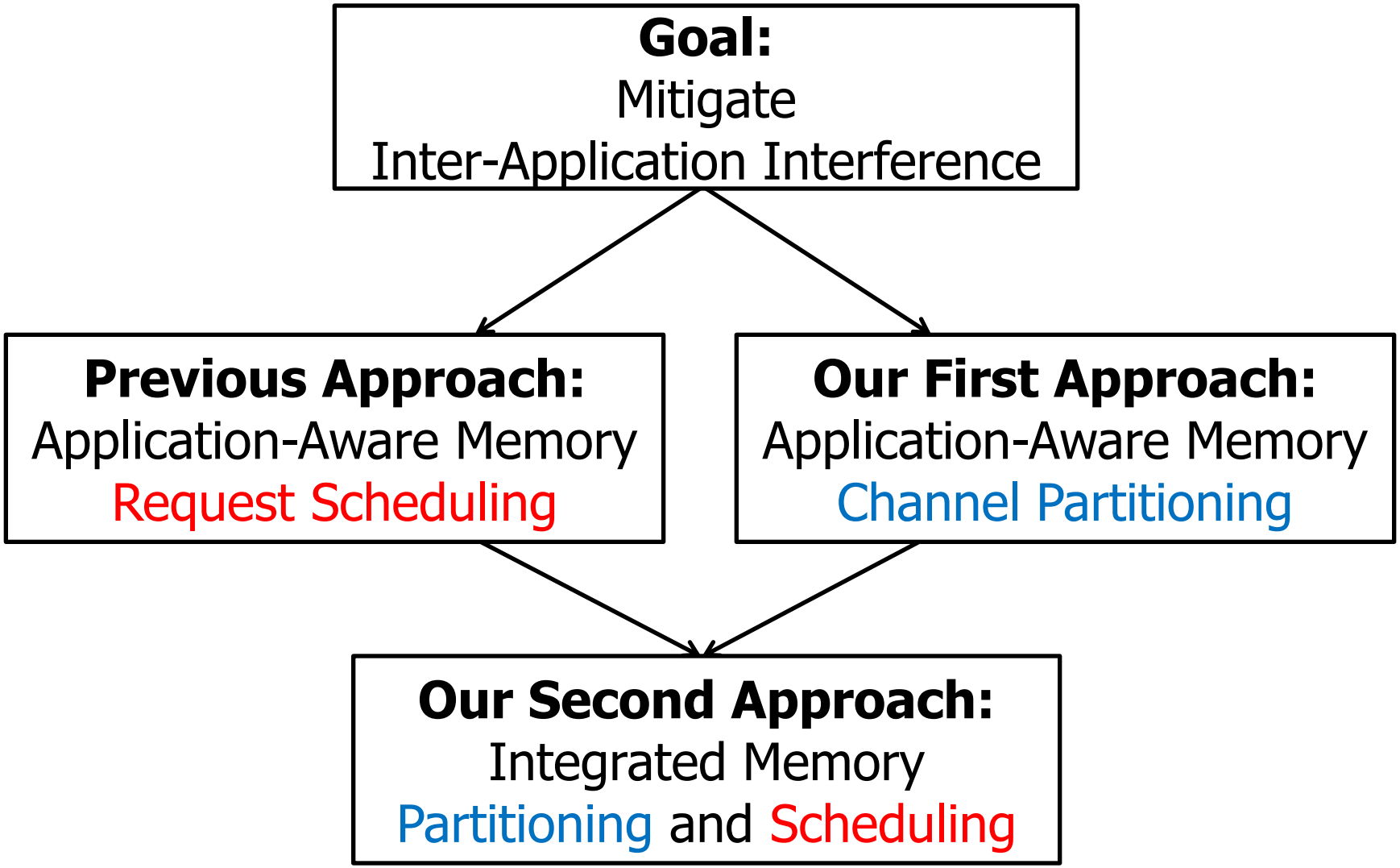
- On a page fault, the operating system
  - allocates page to preferred channel if free page available in preferred channel
  - if free page not available, replacement policy tries to allocate page to preferred channel
  - if it fails, allocate page to another channel

Interval Based Operation



2. Classify applications into groups
3. Partition channels between groups
4. Assign preferred channel to applications

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 $Weighted\ Speedup = \sum_i \frac{IPC_i^{shared}}{IPC_i^{alone}}$

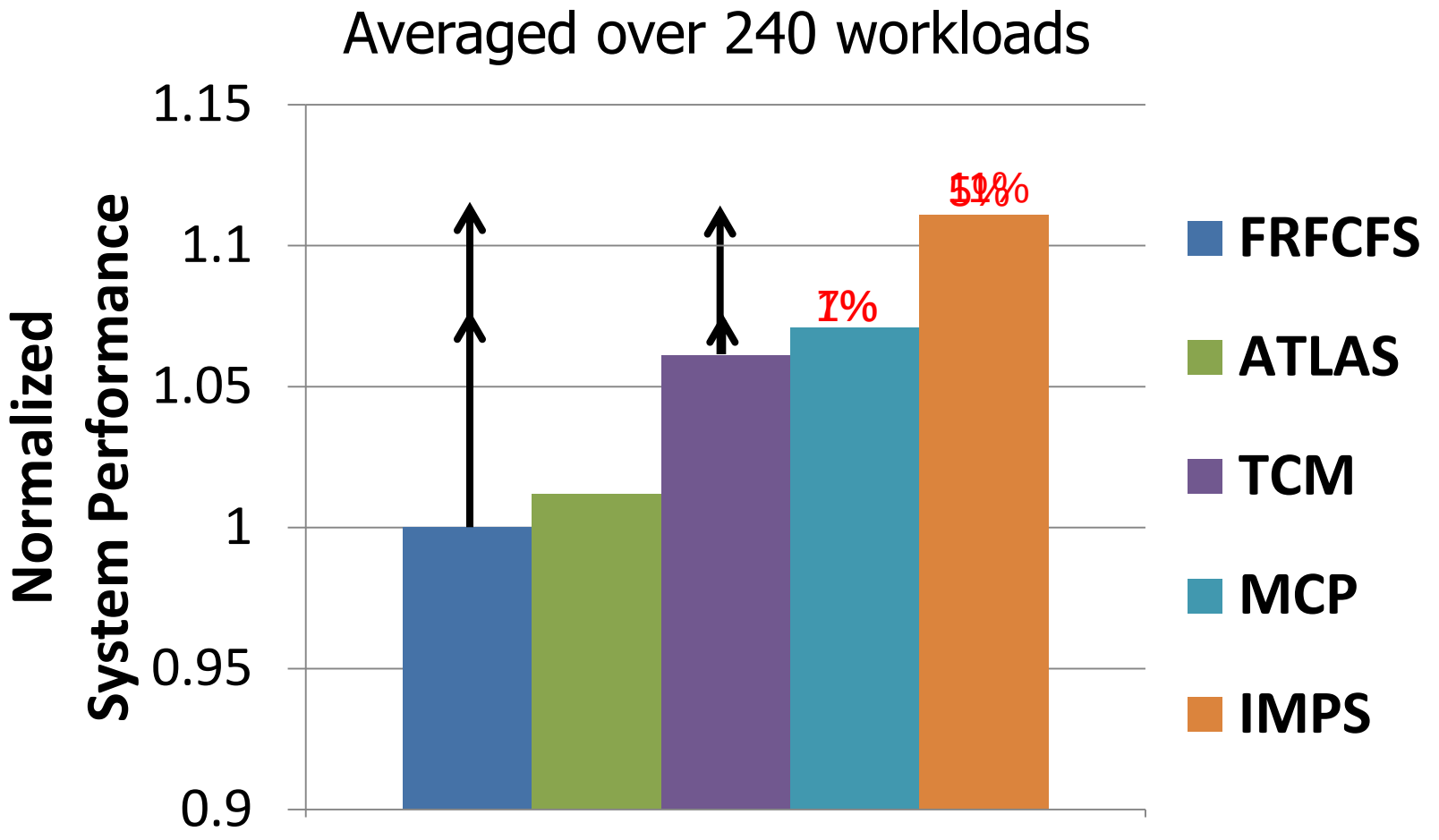
Previous Work on Memory Scheduling

- 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

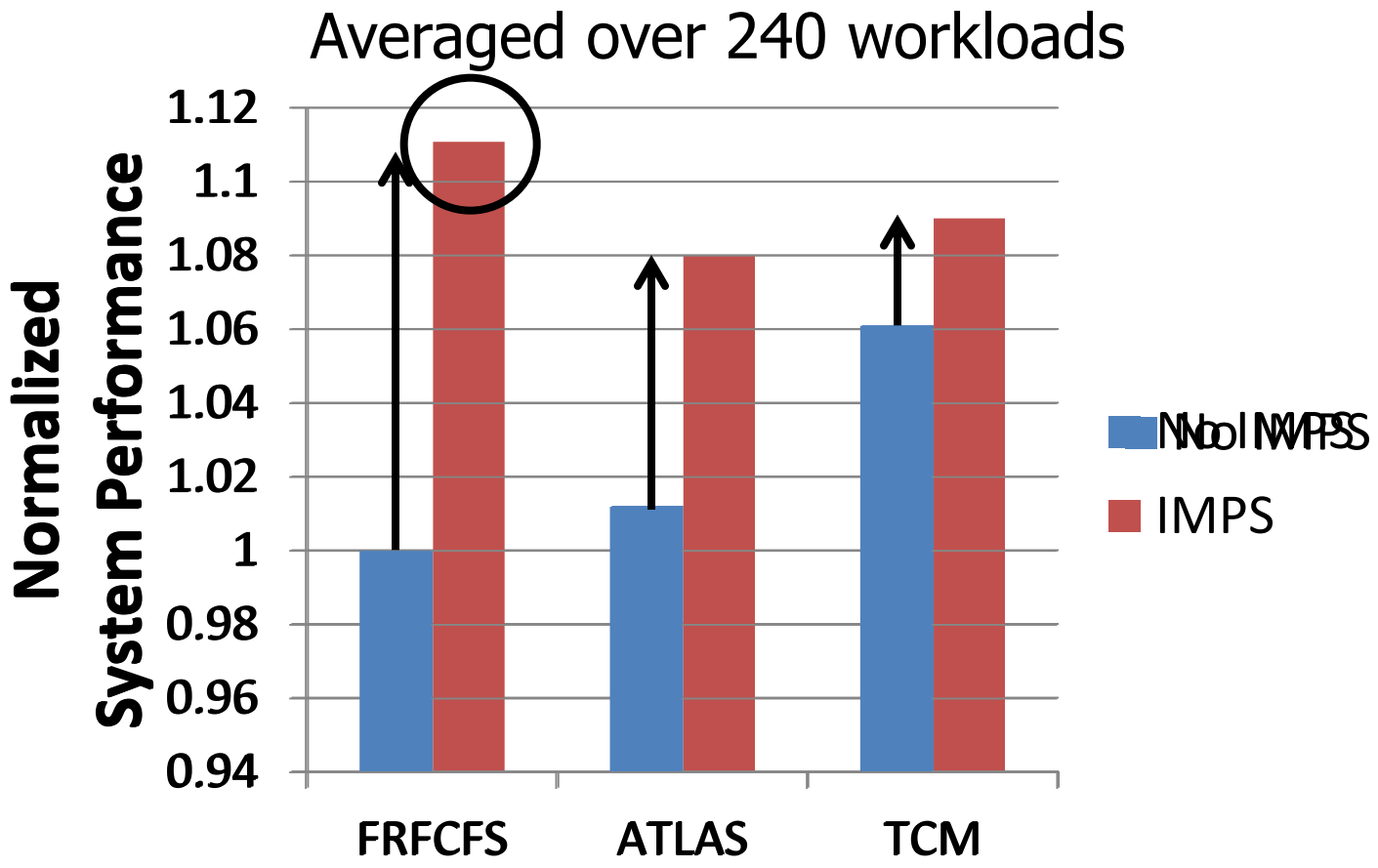
Comparison to Previous Scheduling Policies



Better system performance than the best previous scheduler

Significant performance improvement over baseline FRFCFS at lower hardware cost

Interaction with Memory Scheduling



IMPS improves performance regardless of scheduling policy

Highest improvement over FRFCFS as IMPS designed for FRFCFS

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

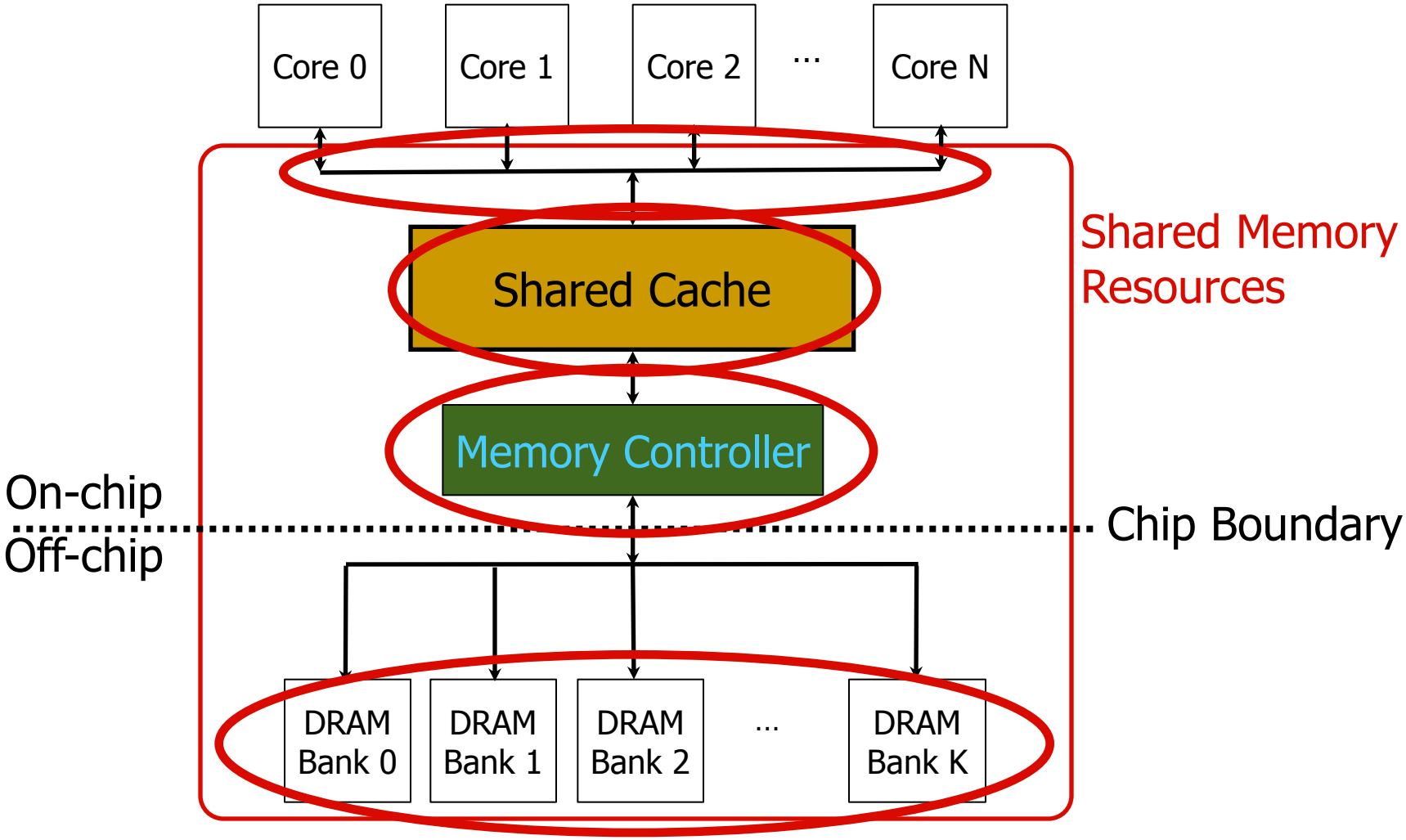
Designing QoS-Aware Memory Systems: Approaches

- 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]
  - 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
  - QoS-aware data mapping to memory controllers [Muralidhara+ MICRO'11]
  - Source throttling to control access to memory system [Ebrahimi+ ASPLOS'10, ISCA'11, TOCS'12] [Ebrahimi+ MICRO'09] [Nychis+ HotNets'10]
  - QoS-aware thread scheduling to cores

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)

Many Shared Resources



The Problem with "Smart Resources"

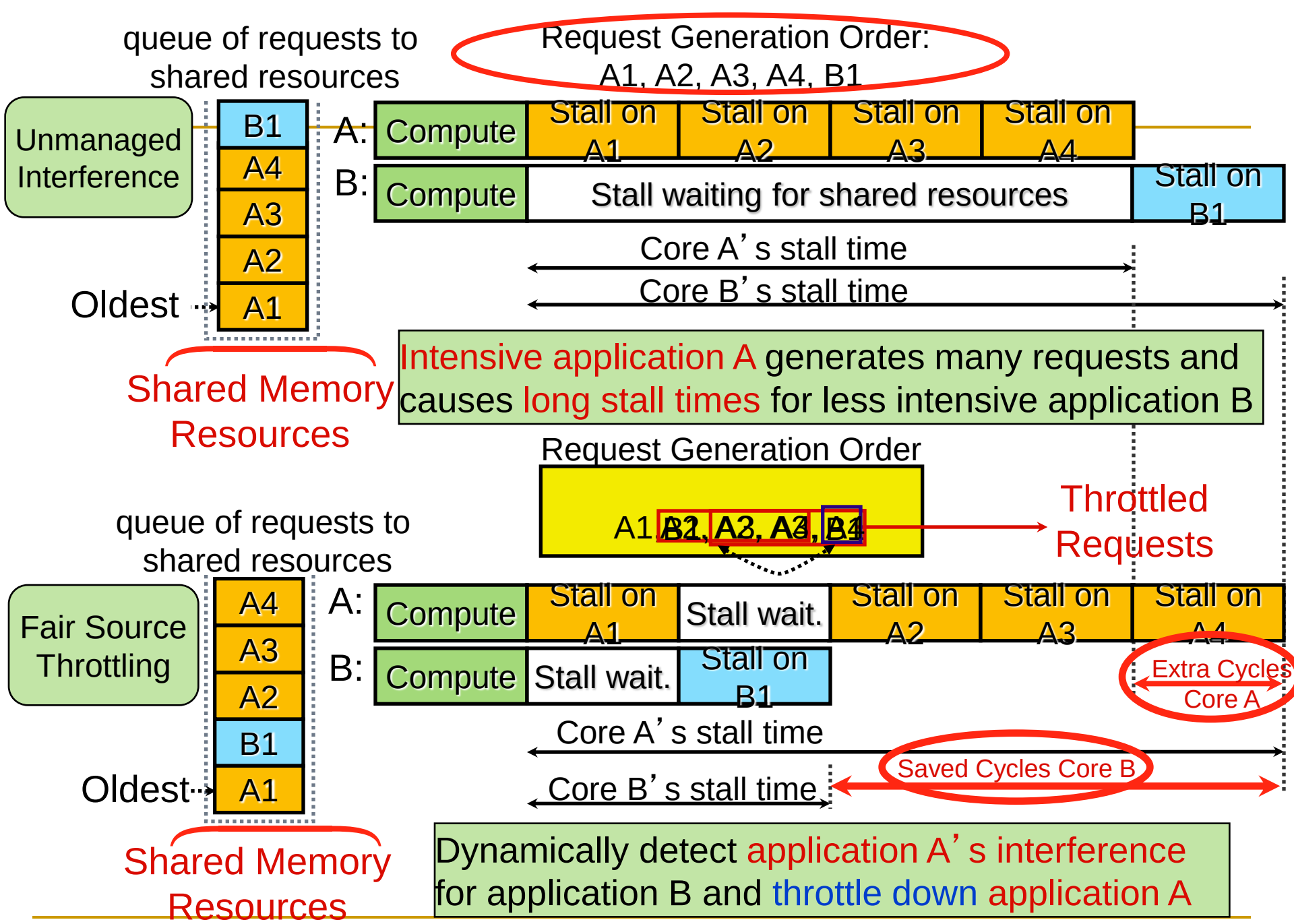
- Independent interference control mechanisms in caches, interconnect, and memory can contradict each other
- Explicitly coordinating mechanisms for different resources requires complex implementation
- How do we enable fair sharing of the entire memory system by controlling interference in a coordinated 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, 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

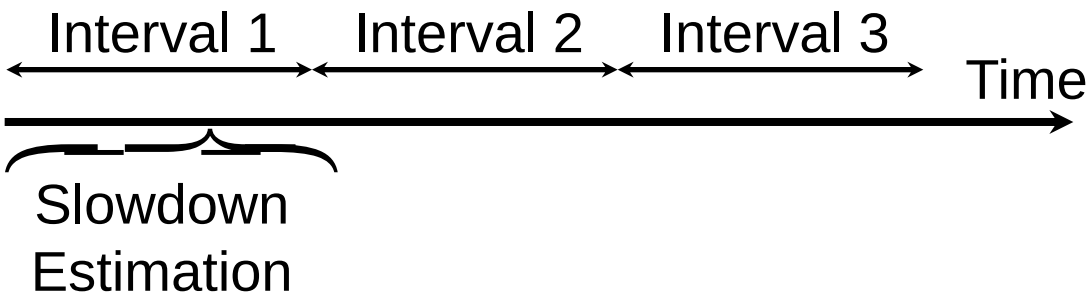
Fairness via Source Throttling (FST)

- 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/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)

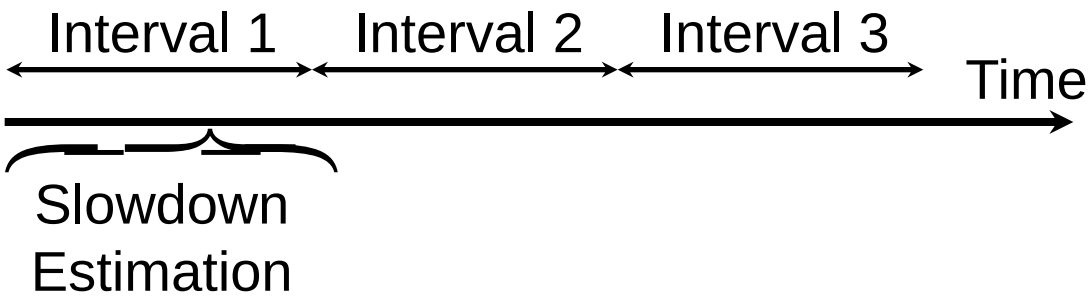


FST

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)



FST

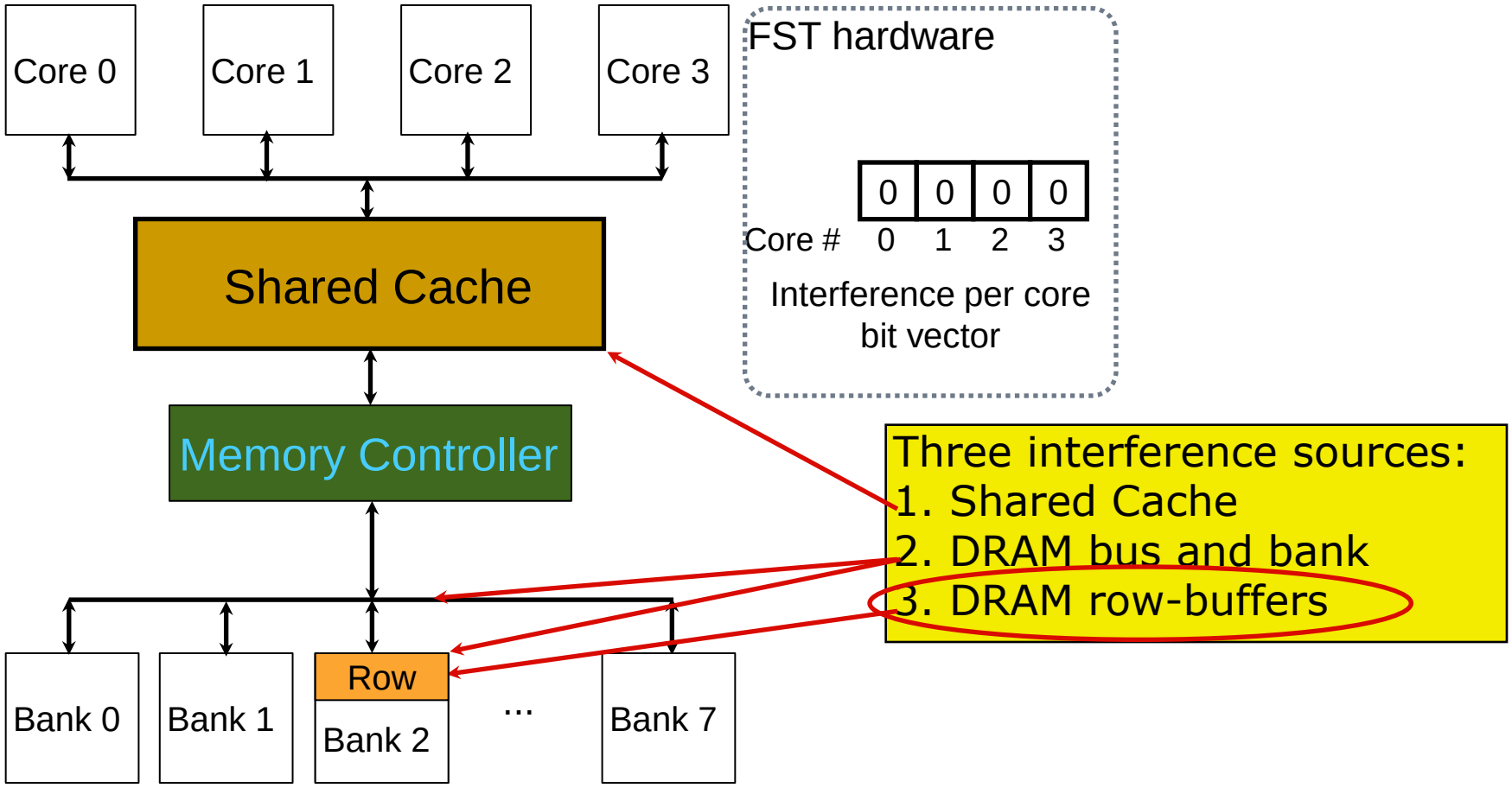
1- Estimating system unfairness
2- Find app. with the highest slowdown (App-slowest)
3- Find app. causing most interference for App-slowest (App-interfering)

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

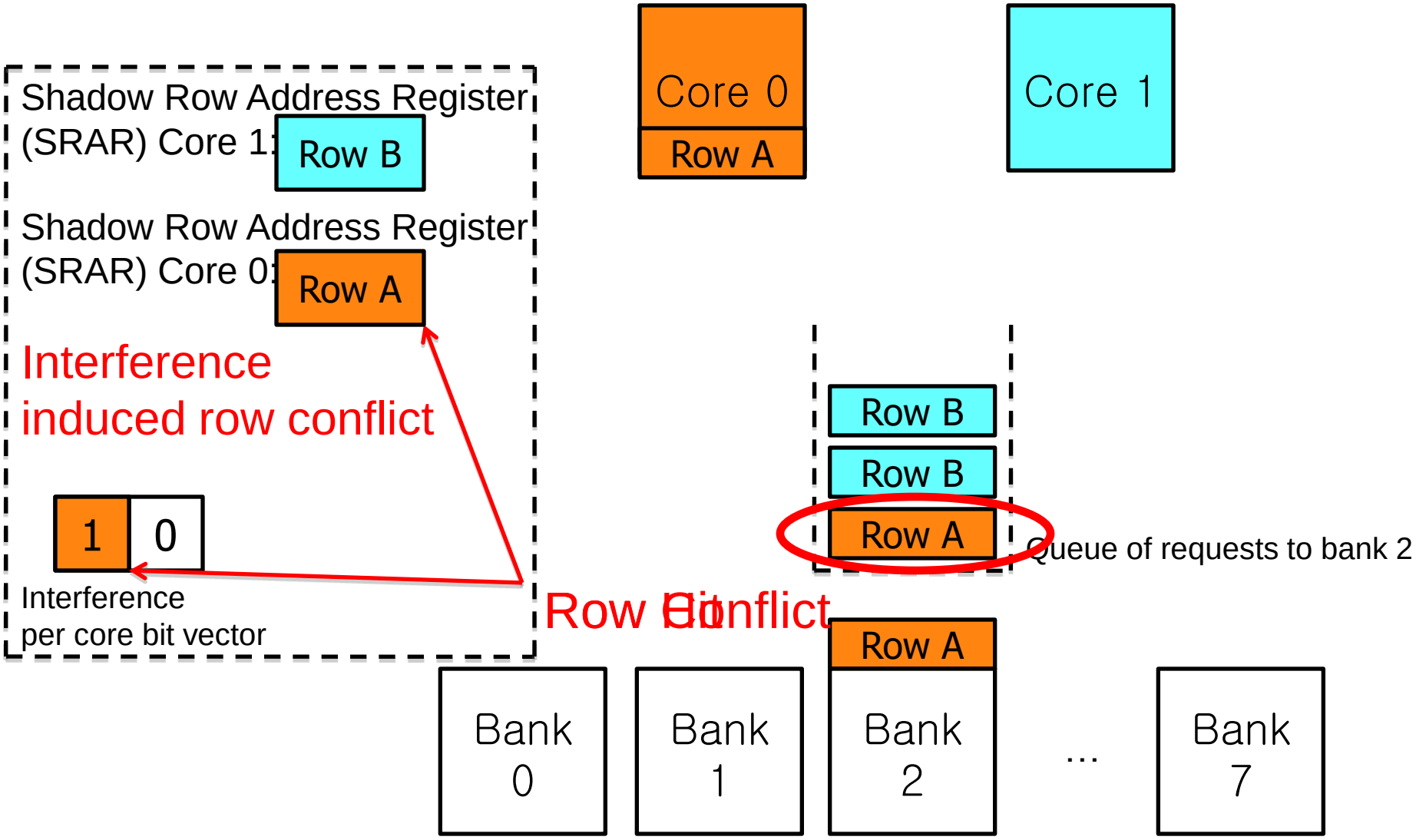
Estimating System Unfairness

- Unfairness = $\frac{\text{Max\{Slowdown i\} over all applications i}}{\text{Min\{Slowdown i\} over all applications i}}$
- 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
- $T_i^{Alone} = T_i^{Shared} - T_i^{Excess}$

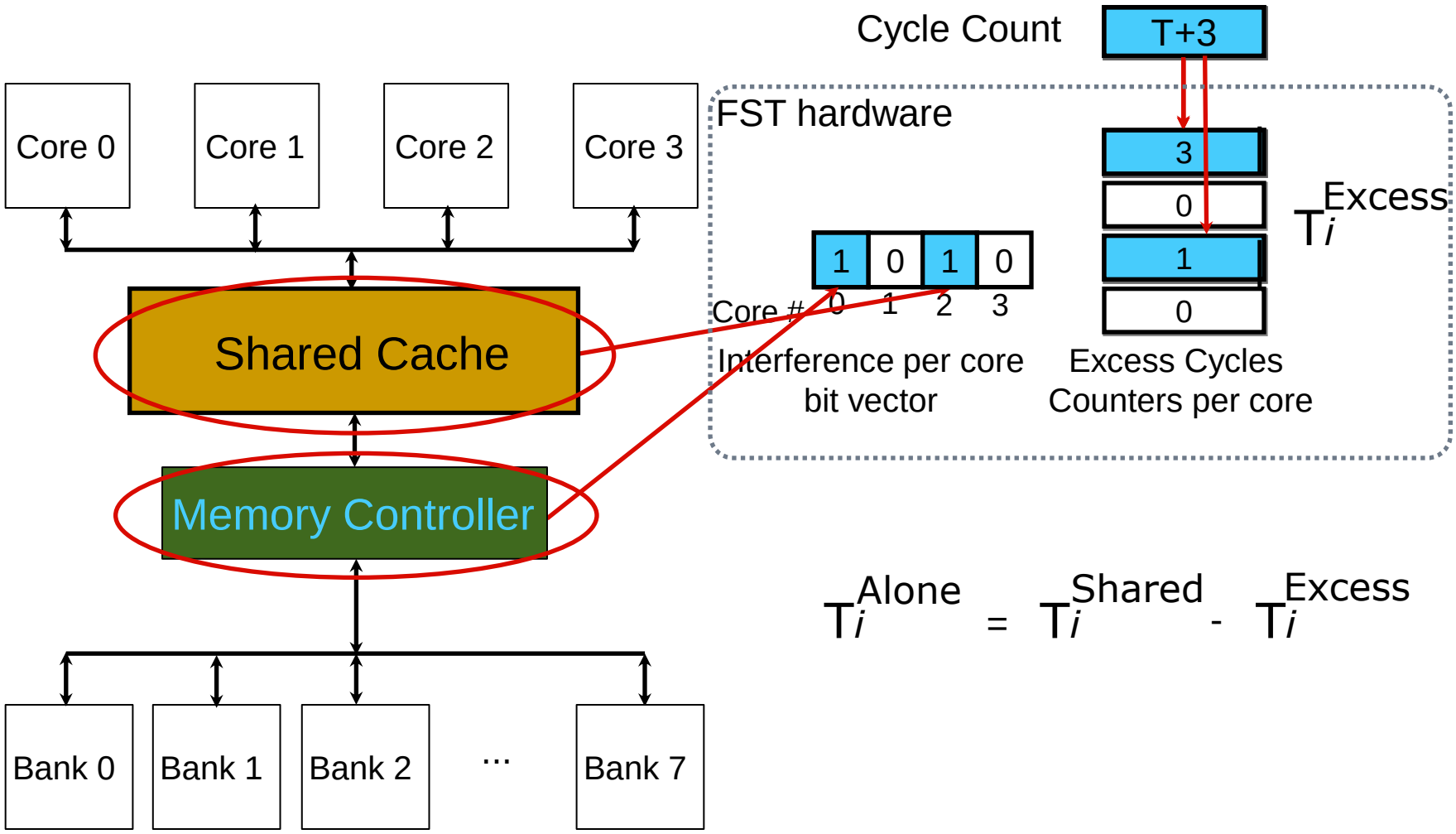
Tracking Inter-Core Interference



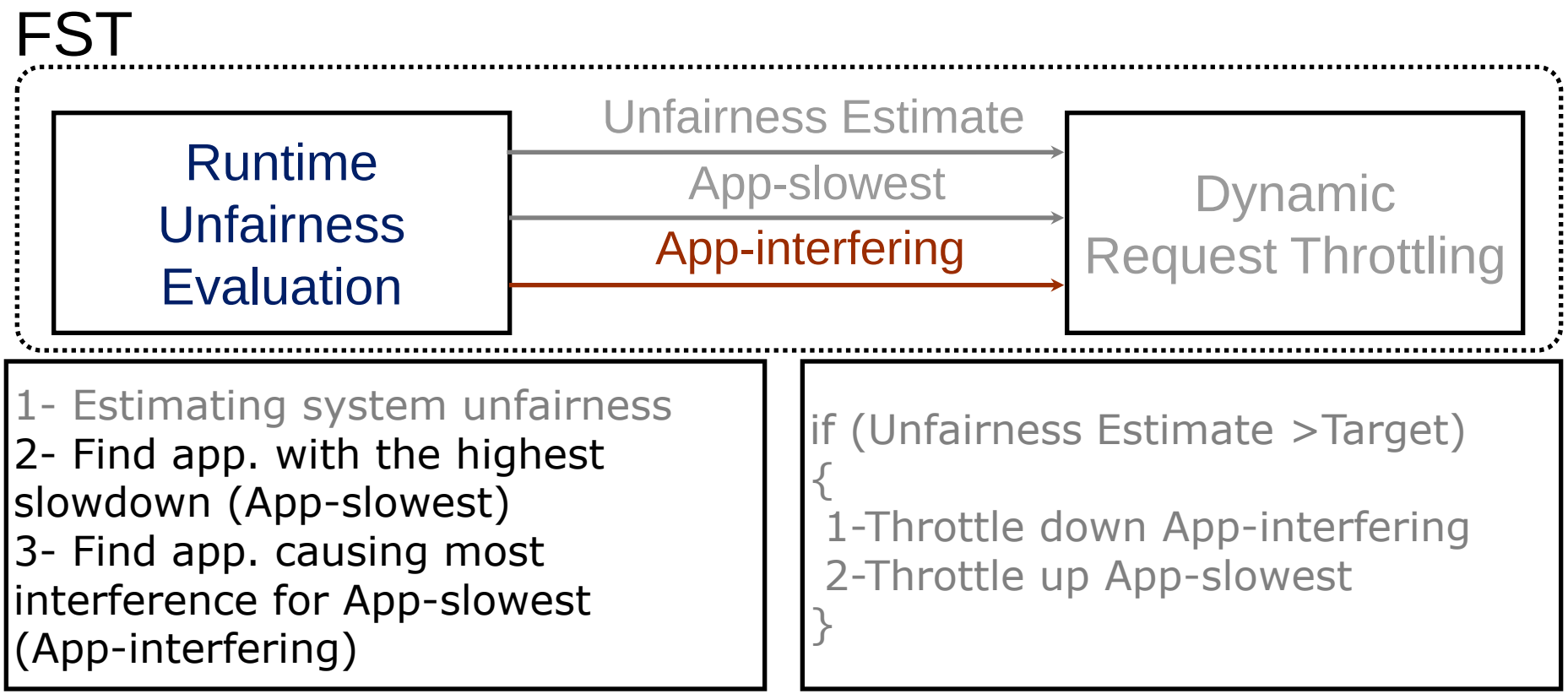
Tracking DRAM Row-Buffer Interference



Tracking Inter-Core Interference



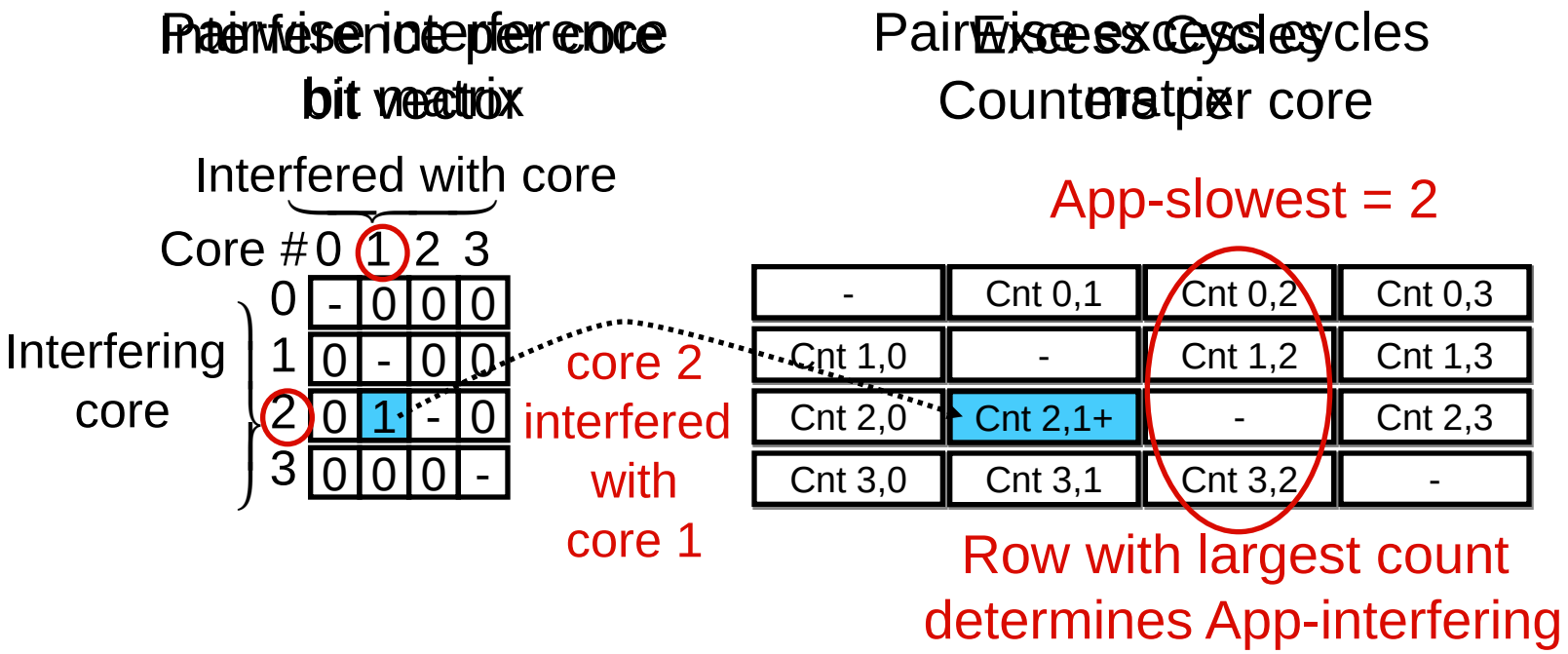
Fairness via Source Throttling (FST)



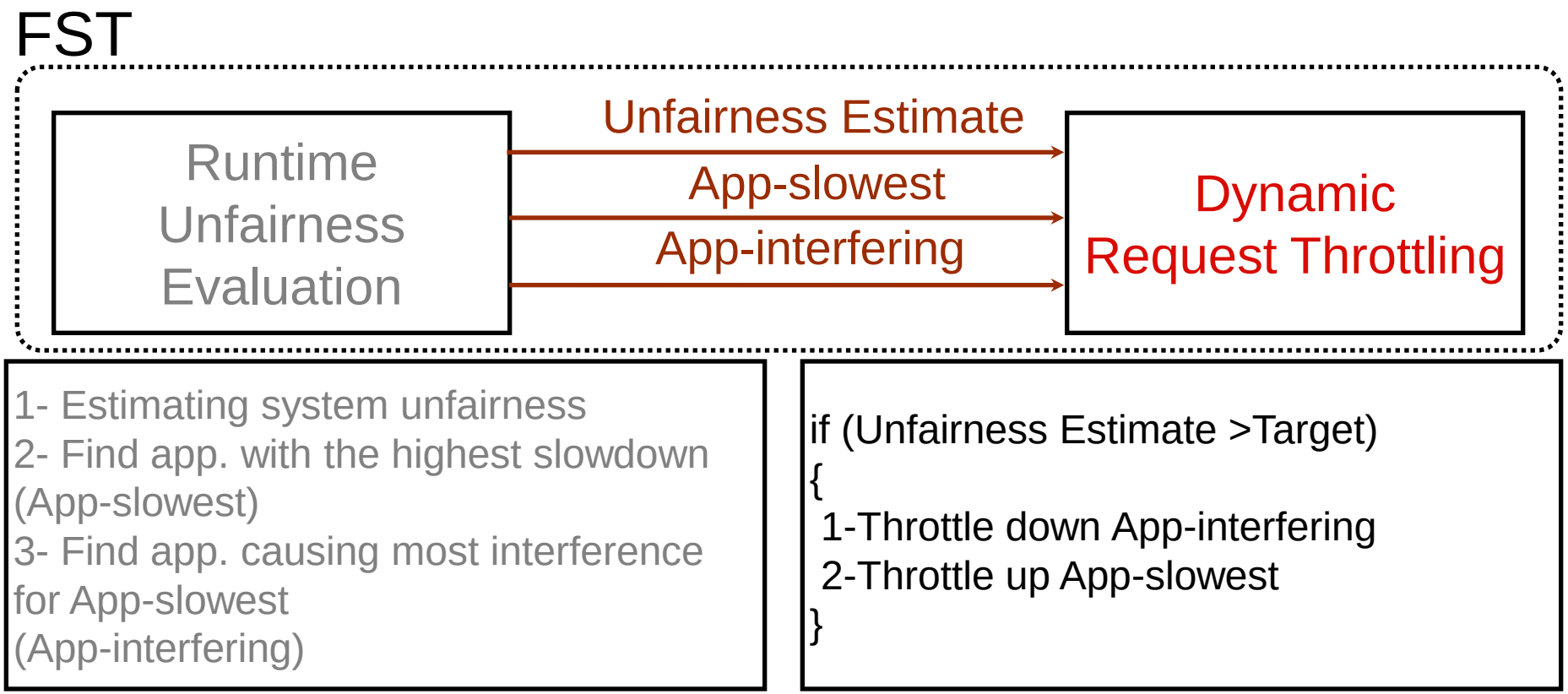
47

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

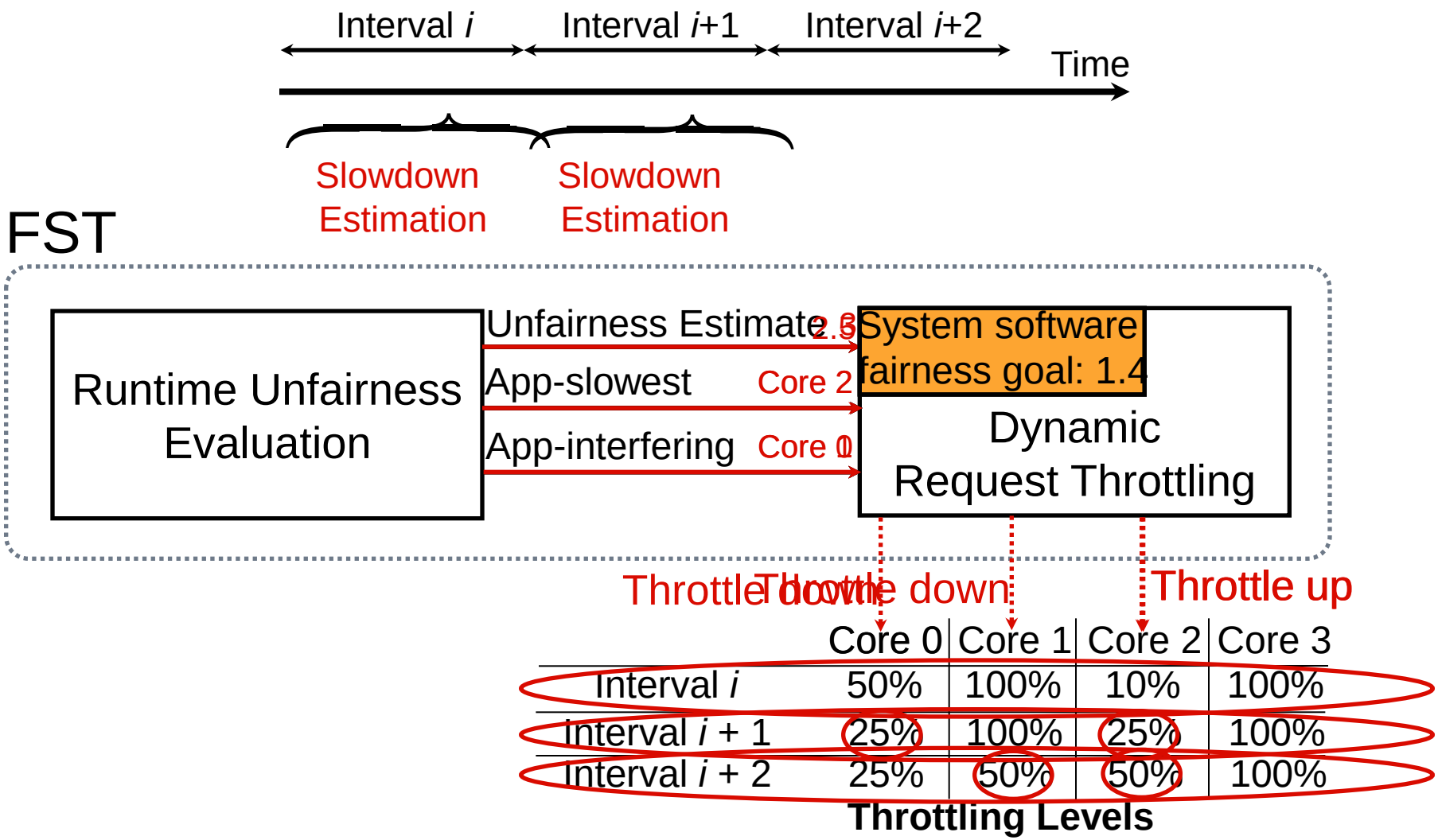
Dynamic Request Throttling

- 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
App-slowest
App-interfering
Core 2
Core 1
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
  - Estimated Unfairness > Target Unfairness
  - 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

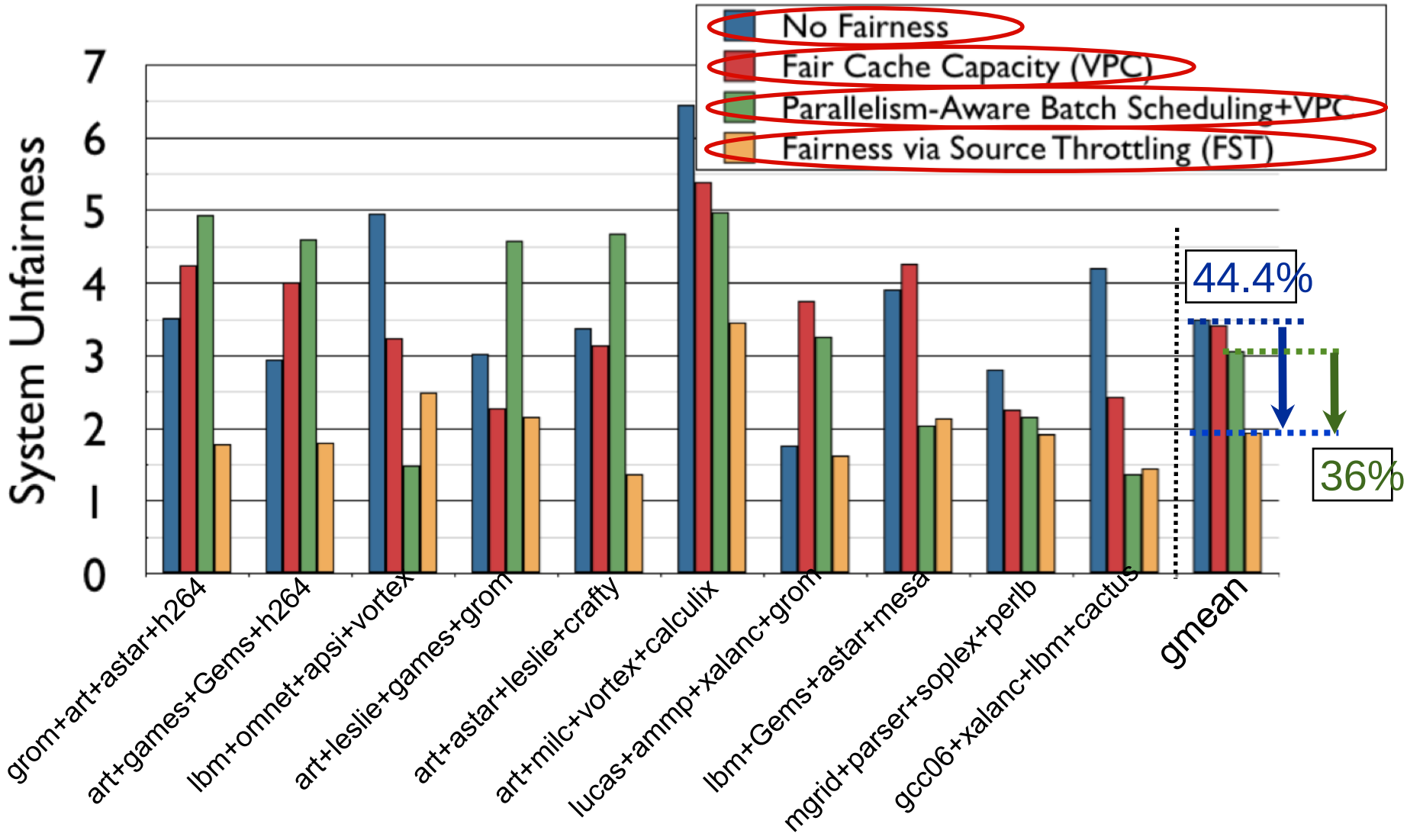
- Total storage cost required for 4 cores is ~12KB

- FST does not require any structures or logic that are on the processor's critical path

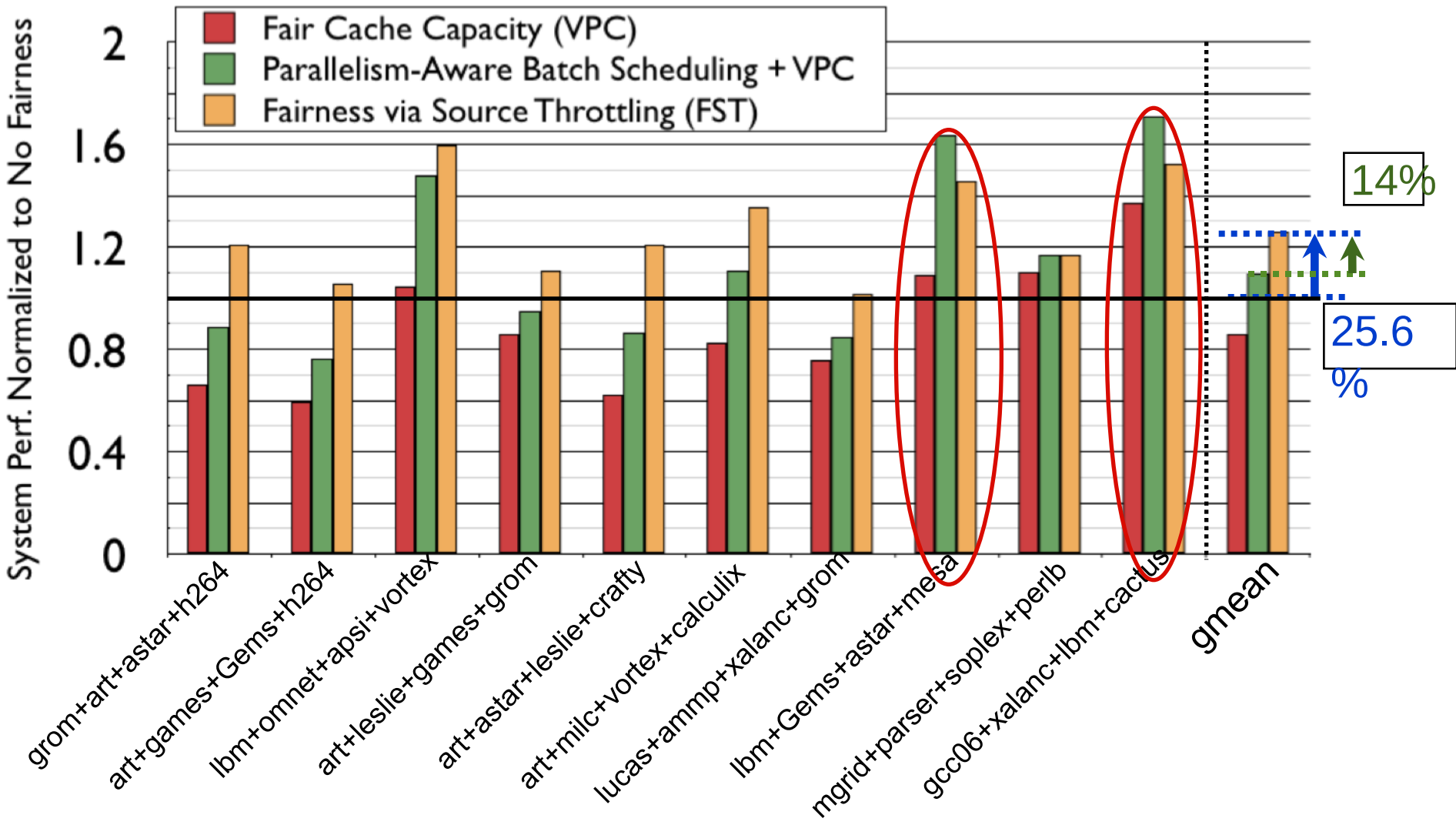
FST Evaluation Methodology

- 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

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 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]

Smart Resources vs. Source Throttling

- Advantages of "smart resources"
  - Each resource is designed to be as efficient as possible → more efficient design using custom techniques for each resource
  - No need for estimating interference across the entire system (to feed a throttling algorithm).
  - Does not lose throughput by possibly overthrottling

- Advantages of source throttling
  - Prevents overloading of any or all resources (if employed well)
  - Can keep each resource simple; no need to redesign each resource
  - Provides prioritization of threads in the entire memory system; instead of per resource
  - Eliminates conflicting decision making between resources

QoS Work So Far

- Major Goals
  - System performance
  - Fairness
- New challenge in today's clouds, clusters
  - Need for guarantees on performance
  - Need for accurate performance prediction
- Fairness via Source Throttling
  - A step in the direction of performance (slowdown) prediction
  - But, slowdown estimates not very accurate

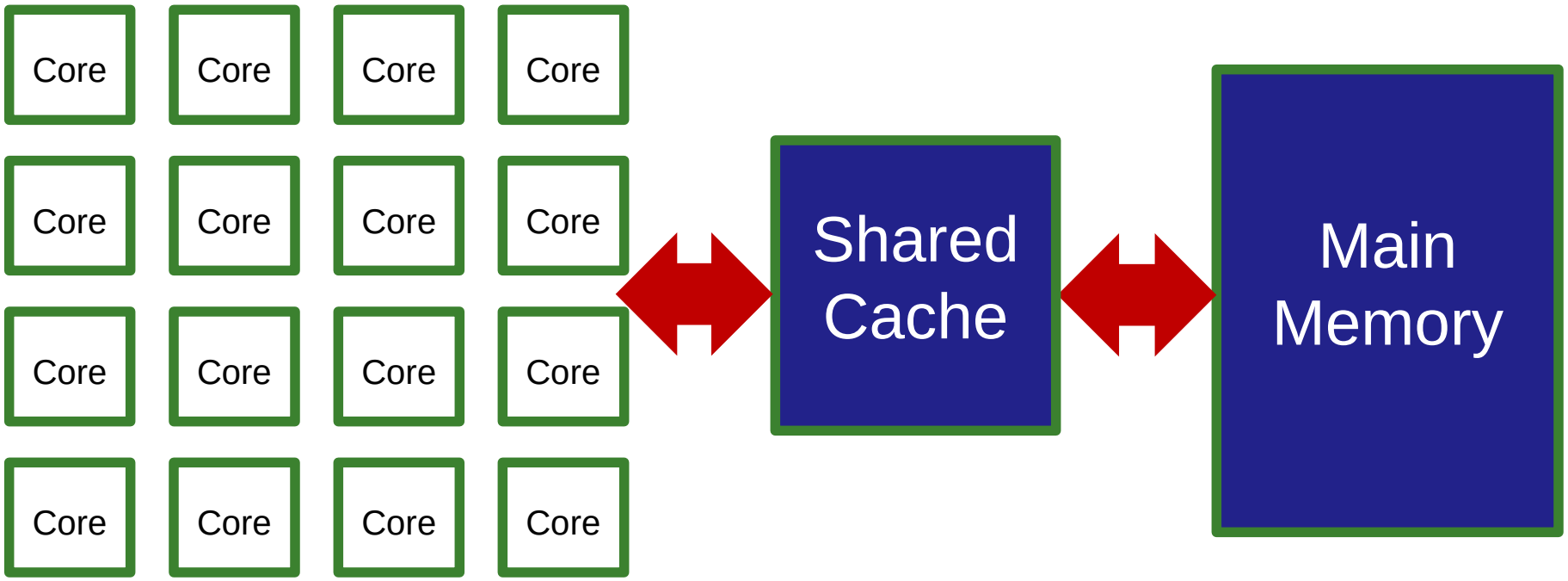
MISE: Providing Performance Predictability in Shared Main Memory Systems

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

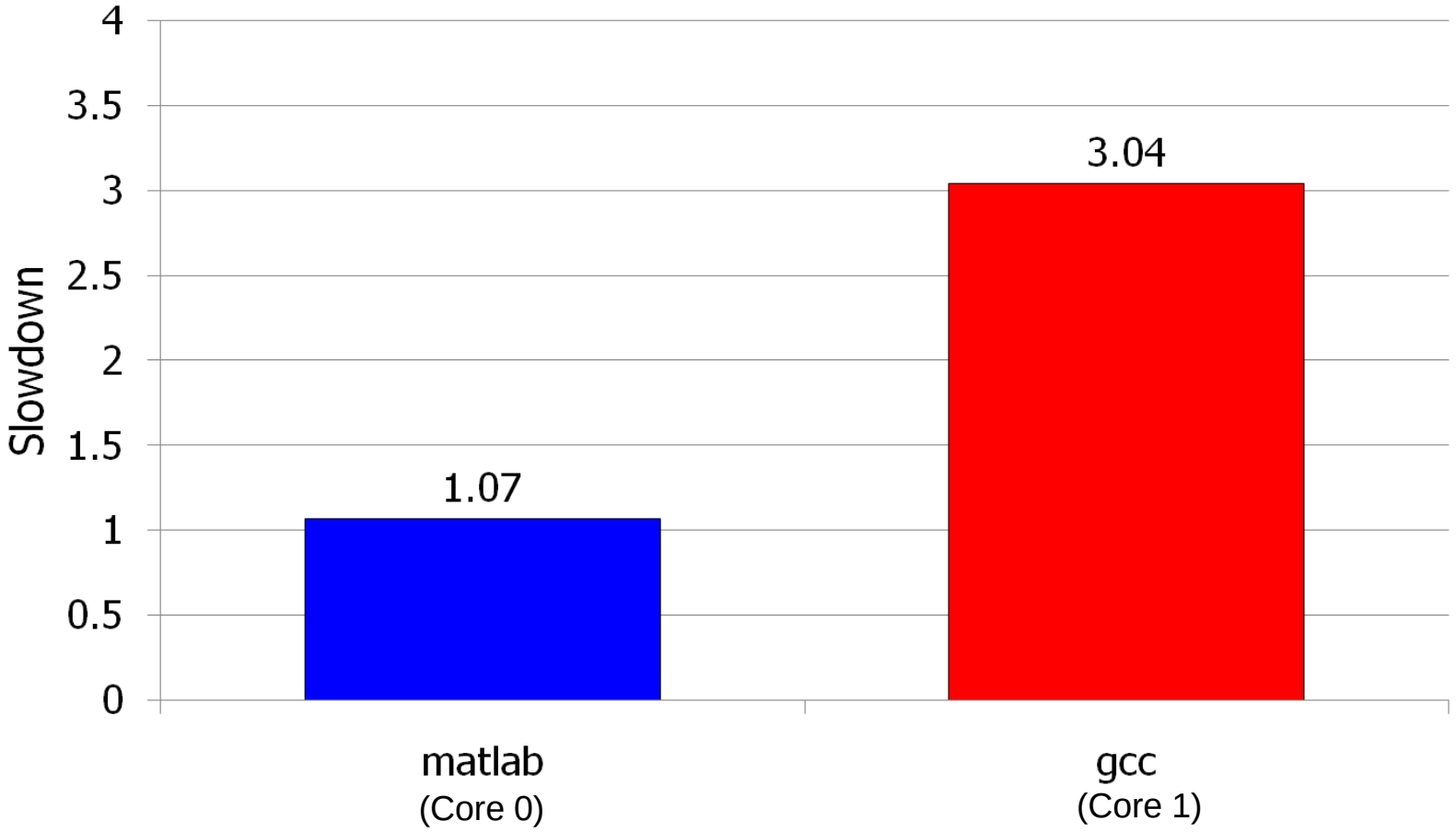
"MISE: Providing Performance Predictability and Fairness in Shared Main Memory Systems"

19th International Symposium on High Performance Computer Architecture (HPCA), Shen Zhen, China, February 2013

Shared Resource Interference is a Problem



Unpredictable Slowdowns

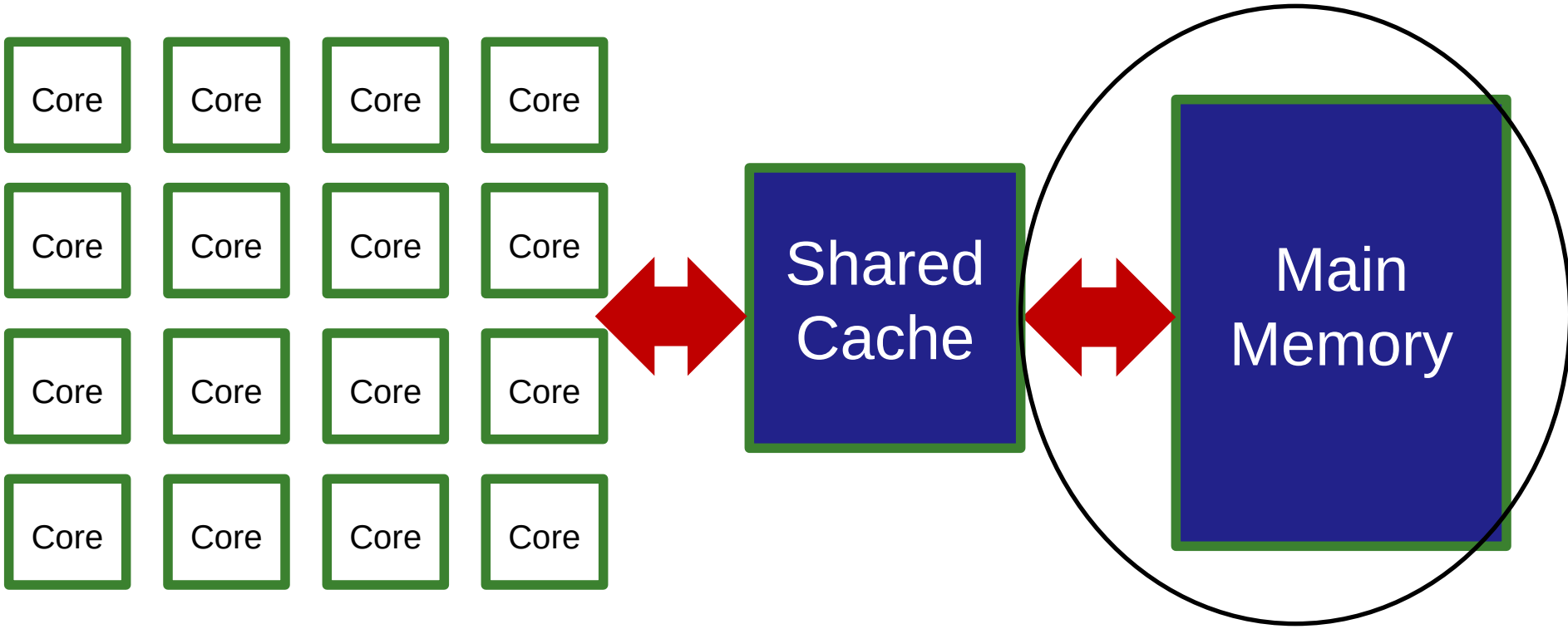


Moscibroda and Mutlu, "Memory performance attacks: Denial of memory service in multi-core systems," USENIX Security 2007.

Need for Predictable Performance

- Billing in a cloud
  - Billing by time?
  - More interference → Longer runtime → Pay more
  - Knowledge of slowdown enables smarter billing
- Server consolidation
  - Multiple applications consolidated on a server
  - Need to provide bounded performance

Towards a Predictable Performance Substrate



Memory Interference-induced Slowdown Estimation (MISE)

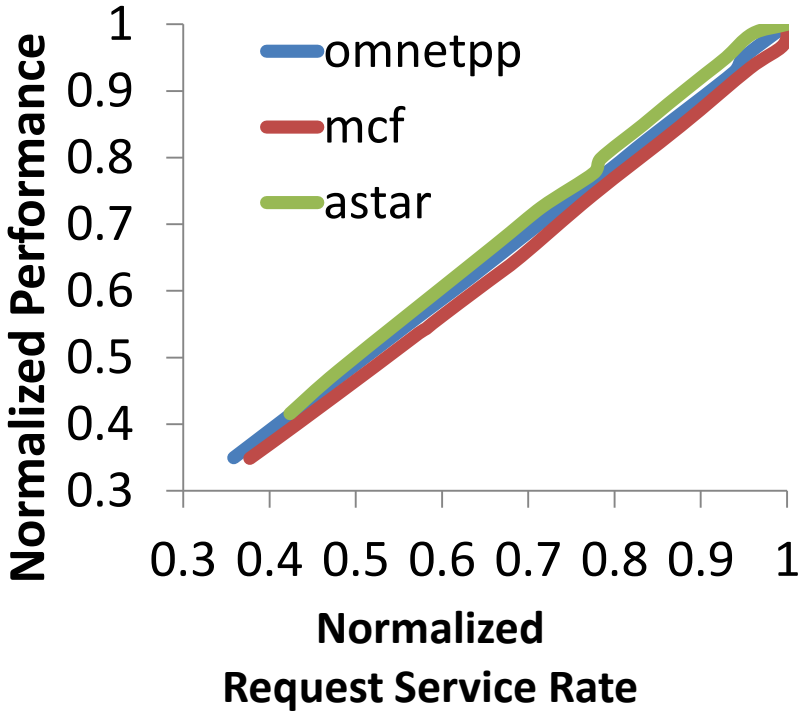
Outline

- Introduction and Motivation
- Slowdown Estimation Model
- Comparison to Prior Work
- An Application of Our Model

Key Observation 1

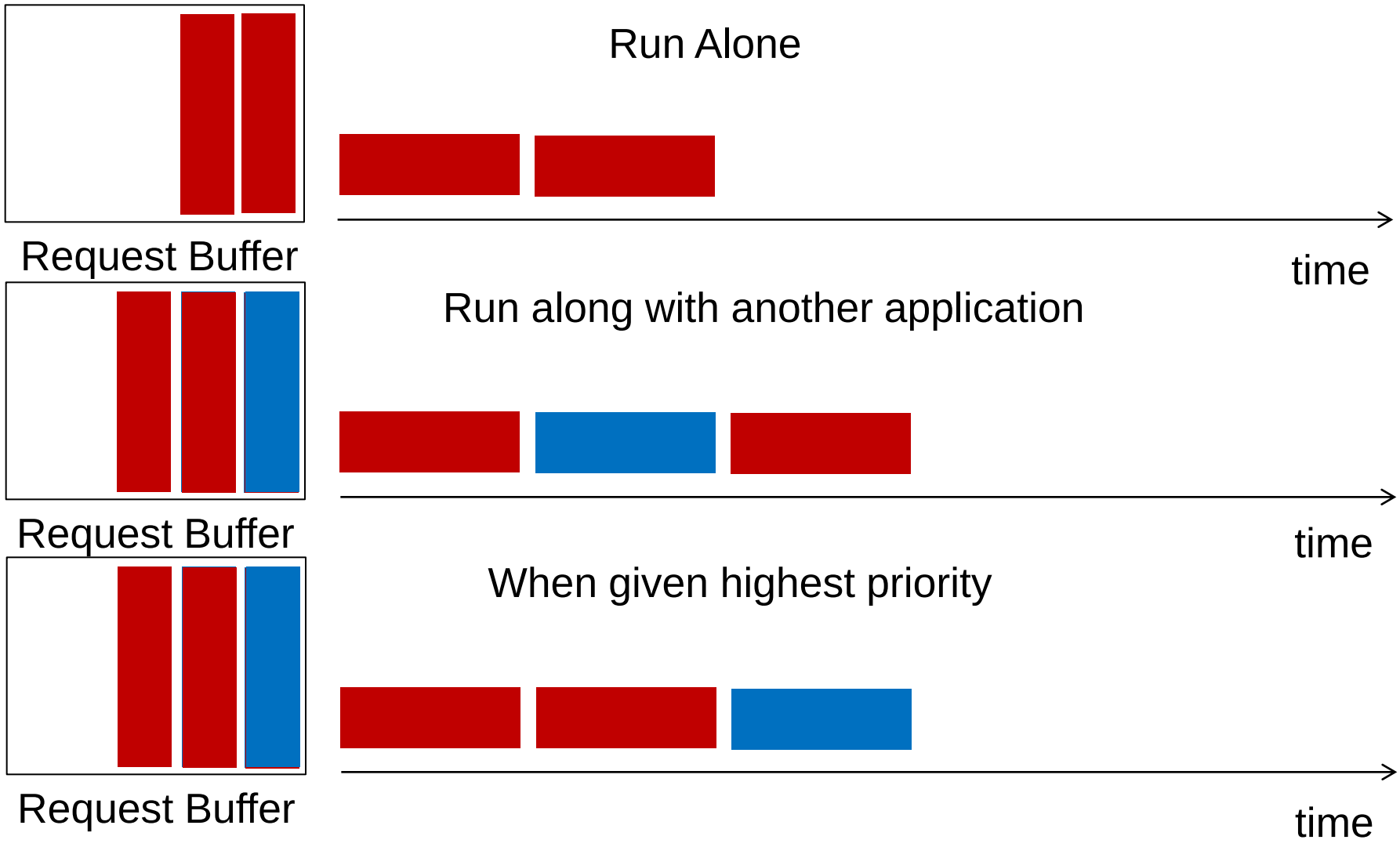
For a memory bound application,

Performance α Request service rate



$$Slowdown = \frac{\text{Alone Request Service Rate (ARSR)}}{\text{Shared Request Service Rate (SRSR)}}$$

Key Observation 2



Key Observation 2

Alone Request Service Rate of an application can be measured by giving the application highest priority in accessing memory

Highest priority → Little interference

Key Observation 3

- Memory-bound applications
  - Spend significant time stalling for memory
- Non-memory-bound applications
  - Spend significant time in compute phase
  - Compute phase length unchanged by request service rate variation

$$\text{Slowdown} = (1 - \alpha) + \alpha \frac{\text{ARSR}}{\text{SRSR}}$$

α – fraction of time in memory phase

Interval Based Implementation

- Divide execution time into intervals
- Slowdown is estimated at end of each interval
- To estimate slowdown: Measure/estimate three major components at the end of each interval
  - Alone Request Service Rate (ARSR)
  - Shared Request Service Rate (SRSR)
  - Memory Phase Fraction (α)

Measuring SRSR and α

- Shared Request Service Rate (SRSR)
  - Per-core counter to track number of requests serviced of each core
  - At the end of each interval, measure

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

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

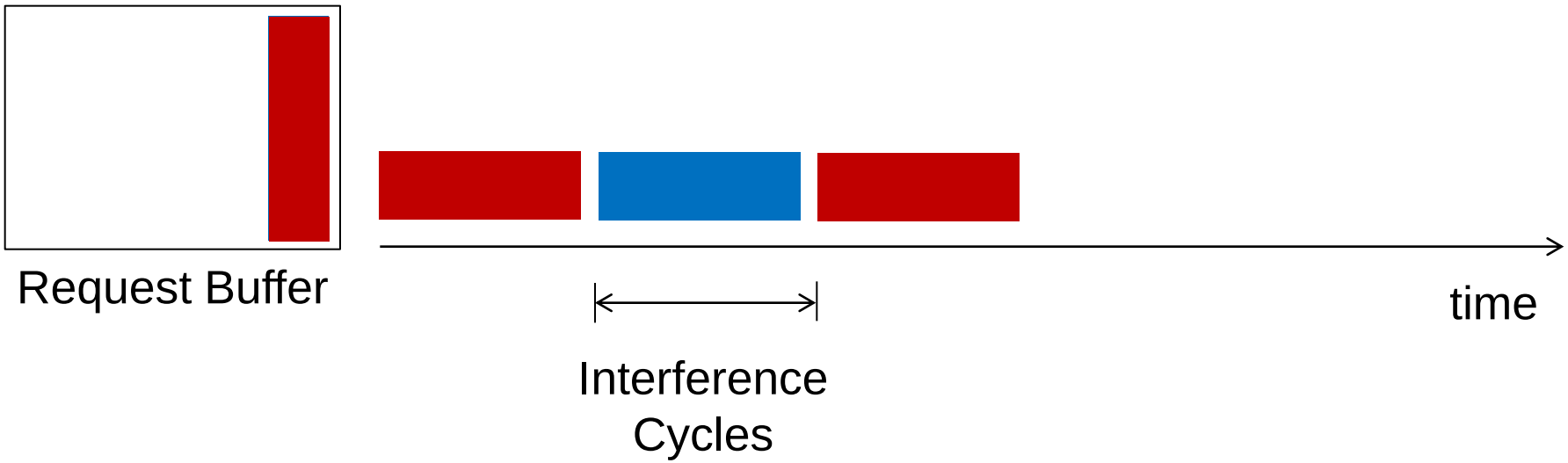
ARSR Estimation Mechanism

- Divide each interval into shorter epochs
- At the beginning of each epoch
  - Randomly pick a highest priority application
  - Probability of picking an application is proportional to its bandwidth allocation
- At the end of an interval, for each application, estimate

$$\text{ARSR} = \frac{\text{Number of High Priority Epoch Requests}}{\text{Number of High Priority Epoch Cycles}}$$

Tackling Inaccuracy in ARSR Estimation

- When an application has highest priority
  - Little Interference
  - Not Zero Interference



Tackling Inaccuracy in ARSR Estimation

- Solution: Factor out 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

$$\text{ARSR} = \frac{\text{Number of High Priority Epoch Requests}}{\text{Number of High Priority Epoch Cycles} - \text{Interference Cycles}}$$

Putting it all Together

- Divide execution time into intervals
- Measure/estimate three major components at the end of each interval
  - Alone Request Service Rate (ARSR)
  - Shared Request Service Rate (SRSR)
  - Memory Phase Fraction (α)
- Estimate slowdown as $\text{Slowdown} = (1 - \alpha) + \alpha \frac{\text{ARSR}}{\text{SRSR}}$

MISE Hardware Cost

- Total storage cost required for 4 cores is ~96 bytes
- Simple changes to memory scheduler

Outline

- Introduction and Motivation
- Slowdown Estimation Model
- Comparison to Prior Work
- An Application of Our Model

Previous Work on Slowdown Estimation

- Major previous work on slowdown estimation
  - STFM (Mutlu+, MICRO 2007)
  - FST (Ebrahimi+, ASPLOS 2010)

- Basic Idea:
  - Estimate slowdown as ratio of uninterfered to interfered memory stall cycles
  - Interfered stall cycles - easy to measure
  - Uninterfered stall cycles - estimated by factoring out stall cycles due to interference

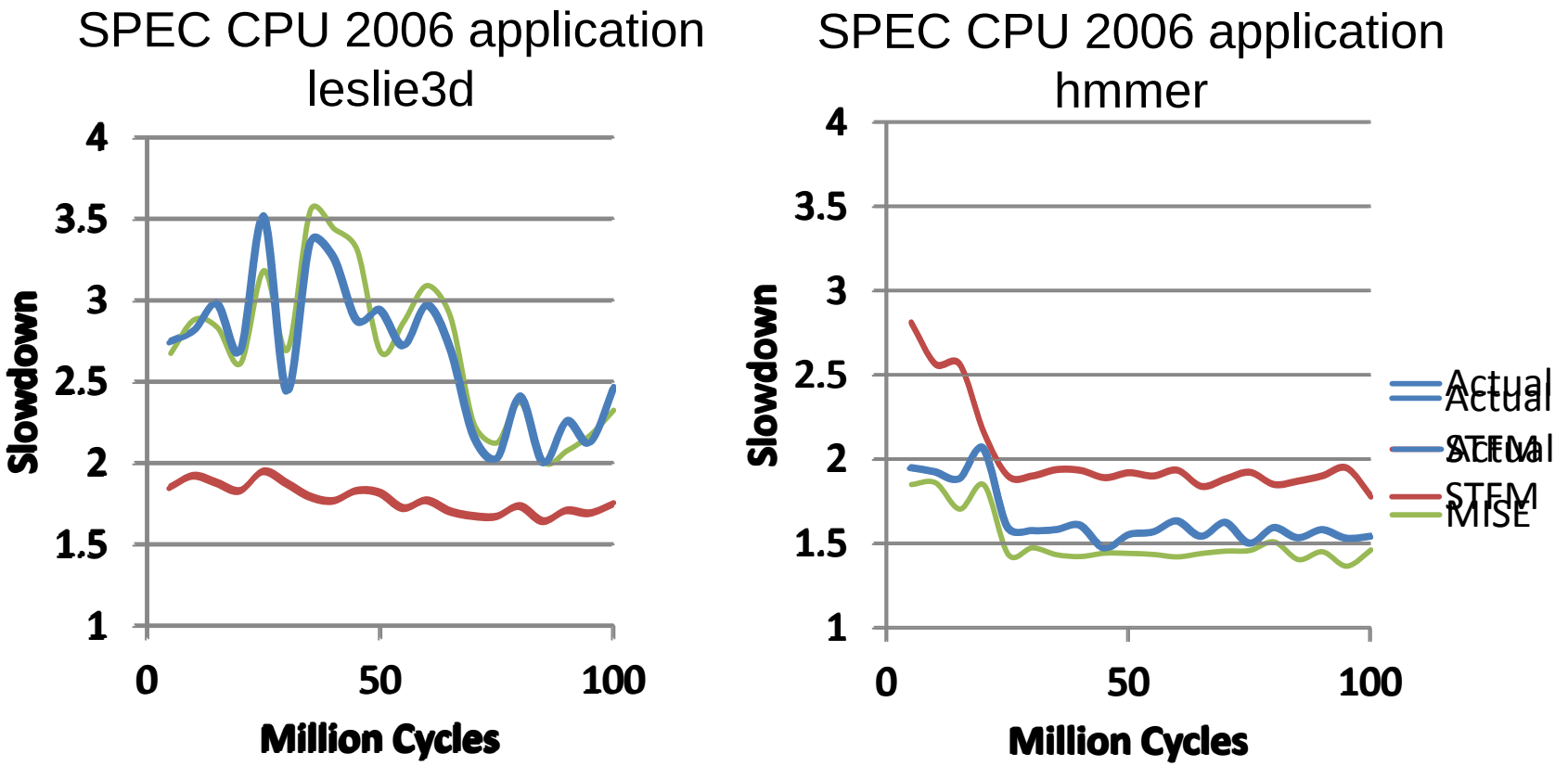
Two Major Advantages of MISE Over STFM

- Advantage 1:
  - STFM tries to estimate uninterfered stall time in the presence of interference
  - MISE eliminates significant portion of interference by giving highest priority
- Advantage 2:
  - STFM's slowdown estimation mechanism is inaccurate for low intensity applications
  - MISE accounts for compute phase providing better accuracy

Methodology

- Configuration of our simulated system
  - 4 cores
  - 1 channel
  - DDR3 1066 DRAM
  - 512 KB private cache/core
  - Data interleaving policy: row interleaving
  - Thread unaware memory scheduling policy
- Workloads
  - 300 multiprogrammed workloads
  - Built using SPEC CPU2006 benchmarks

Quantitative Comparison



Average error of MISE: 8.8%
Average error of STFM/FST: 35.4%
(across 300 multiprogrammed workloads)

Outline

- Introduction and Motivation
- Slowdown Estimation Model
- Comparison to Prior Work
- An Application of Our Model

Providing "Soft" Slowdown Guarantees

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

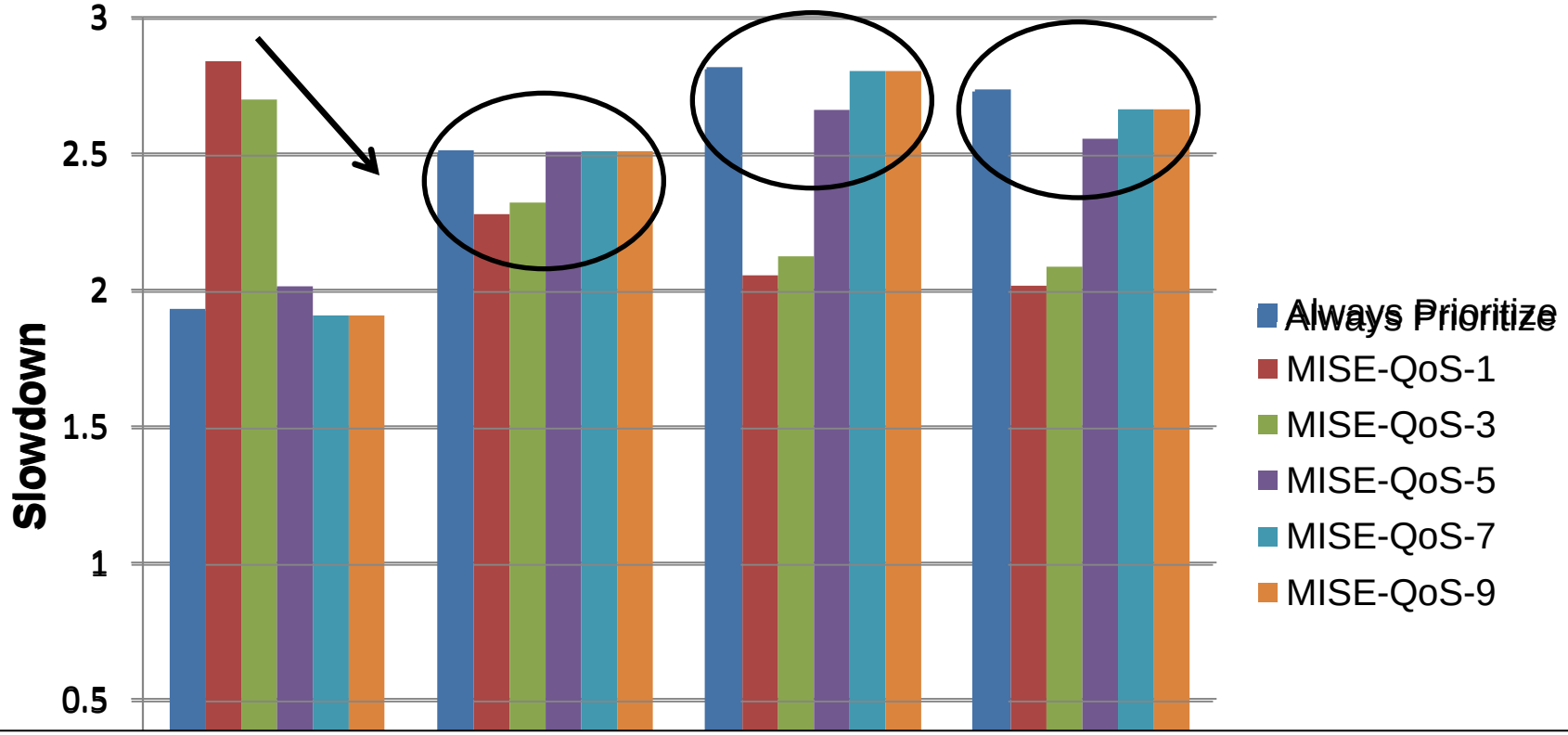
- Basic Idea
  - Allocate just enough bandwidth to QoS-critical application
  - Assign remaining bandwidth to other applications

Mechanism to Provide Soft QoS (For One QoS-Critical Application)

- Estimate slowdown of QoS-critical application
- At the end of each interval
  - If slowdown > bound B, increase bandwidth allocation
  - If slowdown < bound B, decrease bandwidth allocation
- When slowdown bound not met
  - Notify the OS

A Sample Workload

QoS-Critical vs Non-QoS-Critical Application Performance



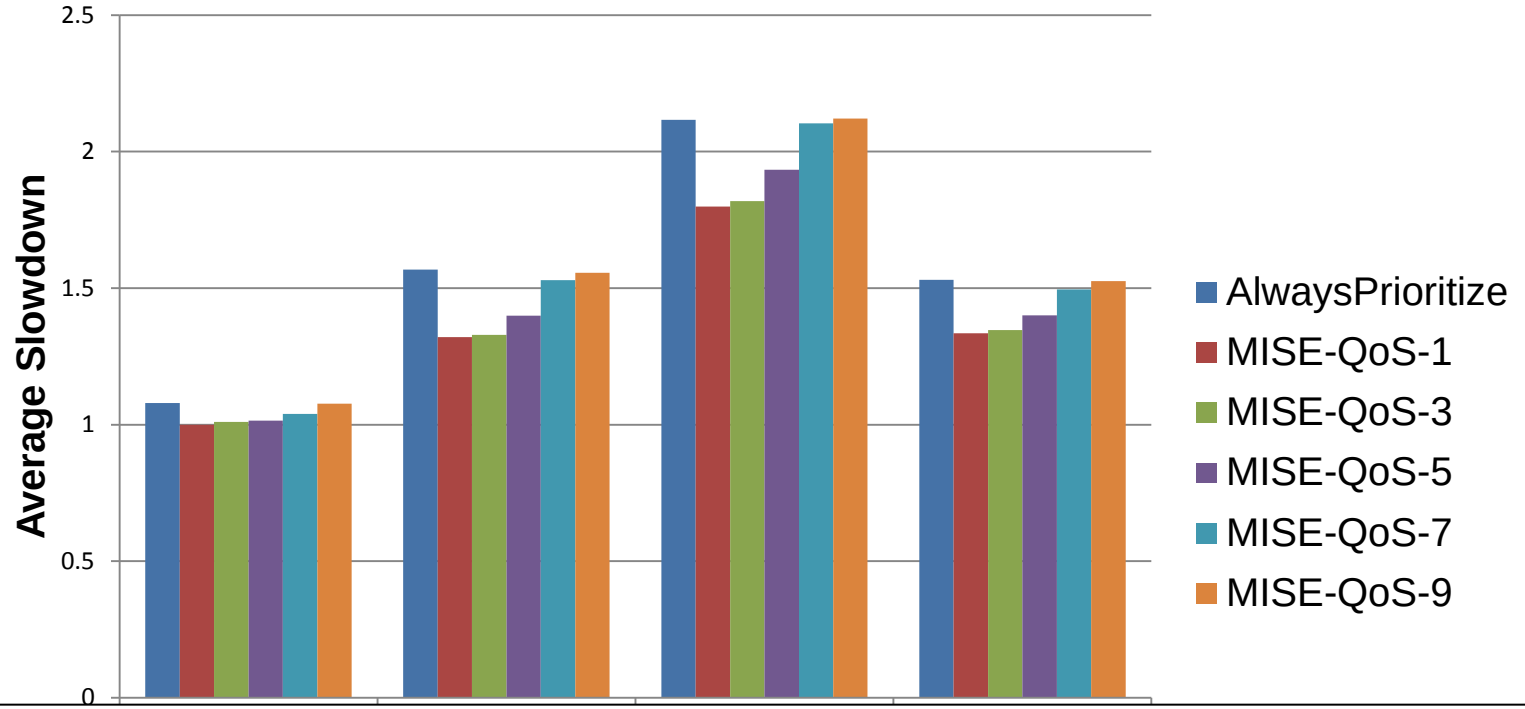
QoS-Critical application's slowdown decreases as the bound becomes tighter

Effectiveness of MISE in Enforcing QoS

	Predicted Right	Predicted Wrong
QoS Bound Met	78.8%	12.1%
QoS Bound Not Met	6.9%	2.2%

Only for 2.2% of workloads is a violated bound predicted as met

Performance of Non-QoS-Critical Applications

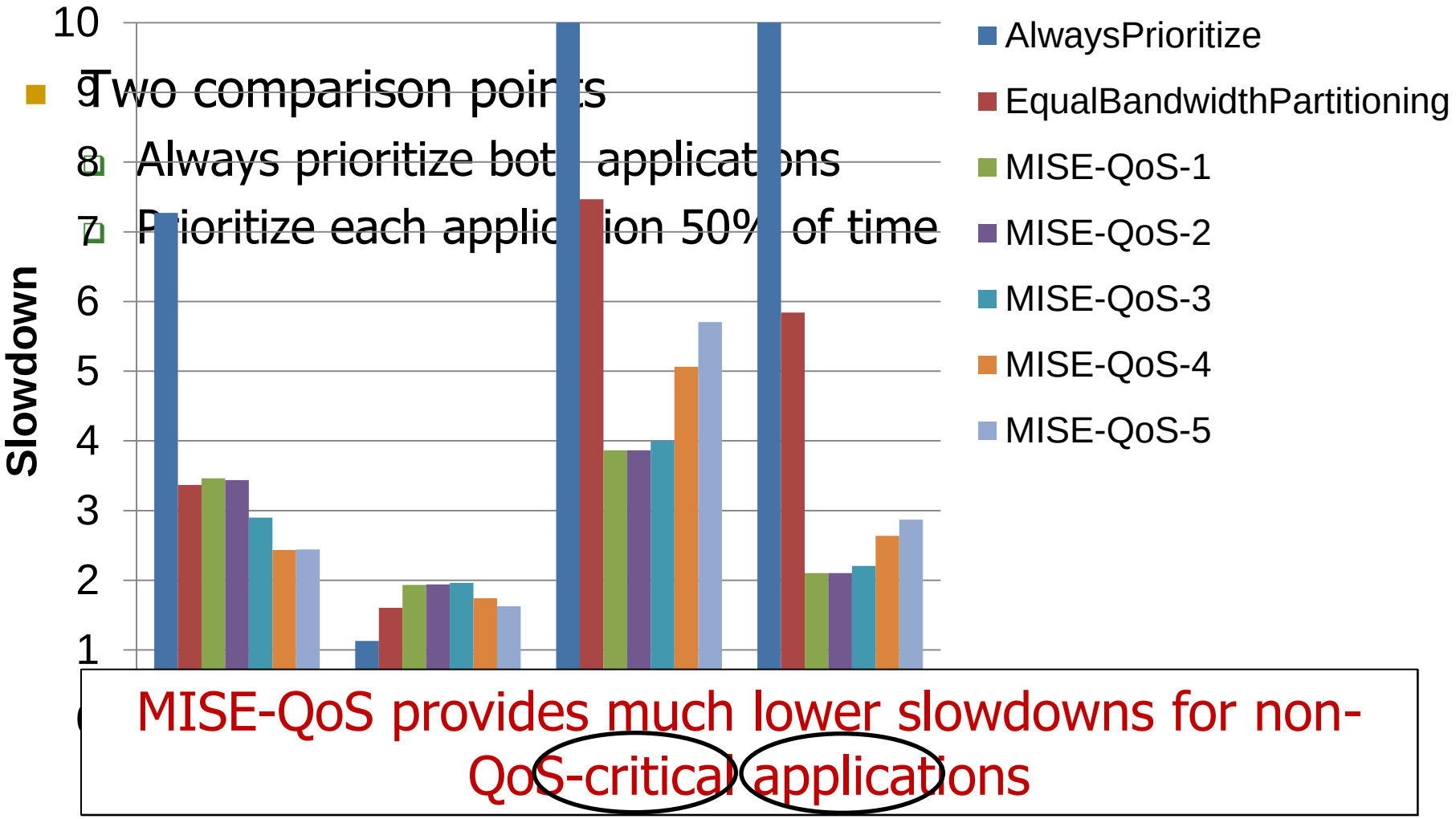


Lower average slowdown when bound is loose

MISE-QoS-3 has 10% lower average slowdown than Always Prioritize

Case Study with Two QoS-Critical Applications

- Two comparison points
  - Always prioritize both applications
  - Prioritize each application 50% of time



MISE-QoS provides much lower slowdowns for non-QoS-critical applications

Future Work

- Exploiting slowdown information in software
  - Admission control
  - Migration policies
  - Billing policies
- Building a comprehensive model
  - Performance predictability with other shared resources
  - Performance predictability in heterogeneous systems

Summary

- Problem
  - Memory interference slows down different applications to different degrees
  - Need to provide predictable performance in the presence of memory interference
- Solution
  - New slowdown estimation model
  - Accurate slowdown estimates: 8.8% error
- Our model enables better QoS-enforcement policies
- We presented one application of our model
  - Providing soft "QoS" guarantees

Research Topics in Main Memory Management

- Abundant

- Interference reduction via different techniques
- Distributed memory controller management
- Co-design with on-chip interconnects and caches
- Reducing waste, minimizing energy, minimizing cost
- Enabling new memory technologies
  - Die stacking
  - Non-volatile memory
  - Latency tolerance