18-742 Fall 2012
Parallel Computer Architecture
Lecture 25: Main Memory Management II

Prof. Onur Mutlu
Carnegie Mellon University
11/12/2012

Reminder: New Review Assignments

- Due: Tuesday, November 13, 11:59pm.
  - Mutlu and Moscibroda, "Parallelism-Aware Batch Scheduling: Enhancing both Performance and Fairness of Shared DRAM Systems," ISCA 2008.
  - Kim et al., "Thread Cluster Memory Scheduling: Exploiting Differences in Memory Access Behavior," MICRO 2010.

- Due: Thursday, November 15, 11:59pm.
  - Ebrahimi et al., "Fairness via Source Throttling: A Configurable and High-Performance Fairness Substrate for Multi-Core Memory Systems," ASPLOS 2010.
  - Muralidhara et al., "Reducing Memory Interference in Multicore Systems via Application-Aware Memory Channel Partitioning," MICRO 2011.

Reminder: Literature Survey Process

- Done in groups: your research project group is likely ideal
- Step 1: Pick 3 or more research papers
  - Broadly related to your research project
- Step 2: Send me the list of papers with links to pdf copies (by Sunday, November 11)
  - I need to approve the 3 papers
  - We will iterate to ensure convergence on the list
- Step 3: Prepare a 2-page writeup on the 3 papers
- Step 3: Prepare a 15-minute presentation on the 3 papers
  - Total time: 15-minute talk + 5-minute Q&A
  - Talk should focus on insights and tradeoffs
- Step 4: Deliver the presentation in front of class (dates: November 26-28 or December 3-7) and turn in your writeup (due date: December 1)

Last Lecture

- Begin shared resource management

- Main memory as a shared resource
  - QoS-aware memory systems
  - Memory request scheduling
    - Memory performance attacks
    - STFM
    - PAR-BS
    - ATLAS

Today

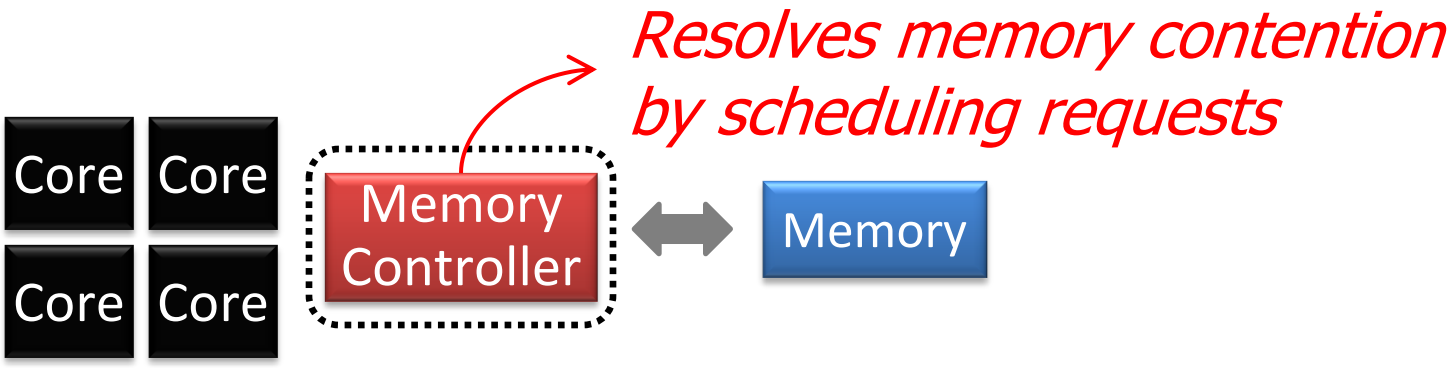
- End QoS-aware Memory Request Scheduling

More on QoS-Aware Memory Request Scheduling

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
  - 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

QoS-Aware Memory Scheduling



- How to schedule requests to provide
  - High system performance
  - High fairness to applications
  - Configurability to system software
- Memory controller needs to be aware of threads

QoS-Aware Memory Scheduling: Evolution

- Stall-time fair memory scheduling [Mutlu+ MICRO'07]
  - Idea: Estimate and balance thread slowdowns
  - Takeaway: Proportional thread progress improves performance, especially when threads are "heavy" (memory intensive)

- Parallelism-aware batch scheduling [Mutlu+ ISCA'08, Top Picks'09]
  - Idea: Rank threads and service in rank order (to preserve bank parallelism); batch requests to prevent starvation
  - Takeaway: Preserving within-thread bank-parallelism improves performance; request batching improves fairness

- ATLAS memory scheduler [Kim+ HPCA'10]
  - Idea: Prioritize threads that have attained the least service from the memory scheduler
  - Takeaway: Prioritizing "light" threads improves performance

QoS-Aware Memory Scheduling: Evolution

- Thread cluster memory scheduling [Kim+ MICRO'10]
  - Idea: Cluster threads into two groups (latency vs. bandwidth sensitive); prioritize the latency-sensitive ones; employ a fairness policy in the bandwidth sensitive group
  - Takeaway: Heterogeneous scheduling policy that is different based on thread behavior maximizes both performance and fairness

- Staged memory scheduling [Ausavarungnirun+ ISCA'12]
  - Idea: Divide the functional tasks of an application-aware memory scheduler into multiple distinct stages, where each stage is significantly simpler than a monolithic scheduler
  - Takeaway: Staging enables the design of a scalable and relatively simpler application-aware memory scheduler that works on very large request buffers

QoS-Aware Memory Scheduling: Evolution

- Parallel application memory scheduling [Ebrahimi+ MICRO'11]
  - Idea: Identify and prioritize limiter threads of a multithreaded application in the memory scheduler; provide fast and fair progress to non-limiter threads
  - Takeaway: Carefully prioritizing between limiter and non-limiter threads of a parallel application improves performance

- Integrated Memory Channel Partitioning and Scheduling [Muralidhara+ MICRO'11]
  - Idea: Only prioritize very latency-sensitive threads in the scheduler; mitigate all other applications' interference via channel partitioning
  - Takeaway: Intelligently ombining application-aware channel partitioning and memory scheduling provides better performance than either

QoS-Aware Memory Scheduling: Evolution

- Prefetch-aware shared resource management [Ebrahimi+ ISCA'12] [Ebrahimi+ MICRO'09] [Lee+ MICRO'08]
  - Idea: Prioritize prefetches depending on how they affect system performance; even accurate prefetches can degrade performance of the system
  - Takeaway: Carefully controlling and prioritizing prefetch requests improves performance and fairness

Properties of ATLAS

Goals	Properties of ATLAS
▪ Maximize system performance	▪ LAS-ranking ▪ Bank-level parallelism ▪ Row-buffer locality
▪ Scalable to large number of controllers	▪ Very infrequent coordination
▪ Configurable by system software	▪ Scale attained service with thread weight (in paper)
	▪ **Low complexity**: Attained service requires a single counter per thread in each MC

ATLAS Pros and Cons

- Upsides:
  - Good at improving performance
  - Low complexity
  - Coordination among controllers happens infrequently

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

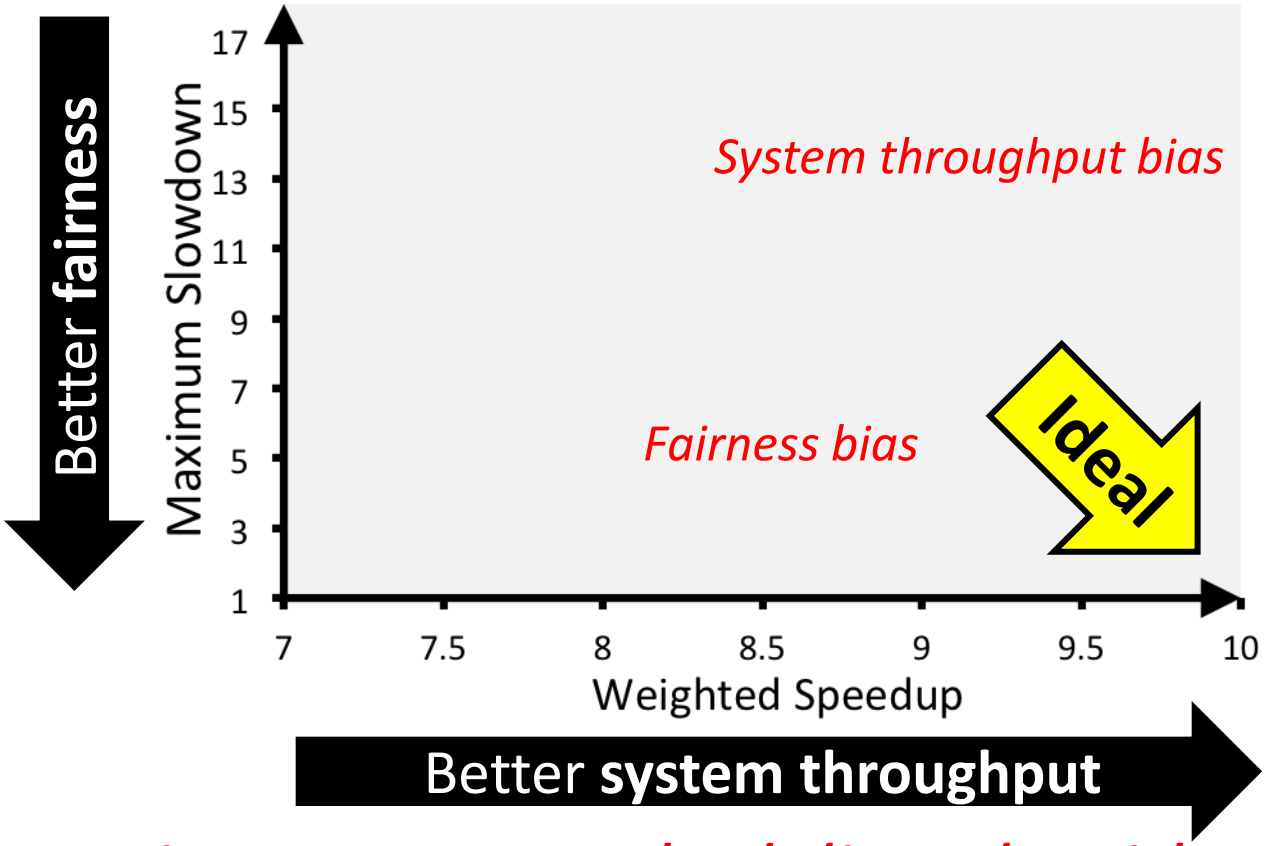
TCM:
Thread Cluster Memory Scheduling

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

Throughput vs. Fairness

24 cores, 4 memory controllers, 96 workloads



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

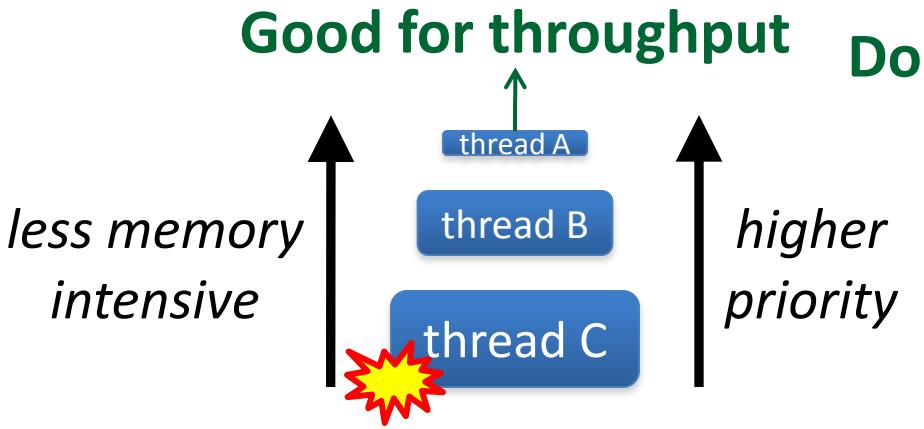
Throughput vs. Fairness

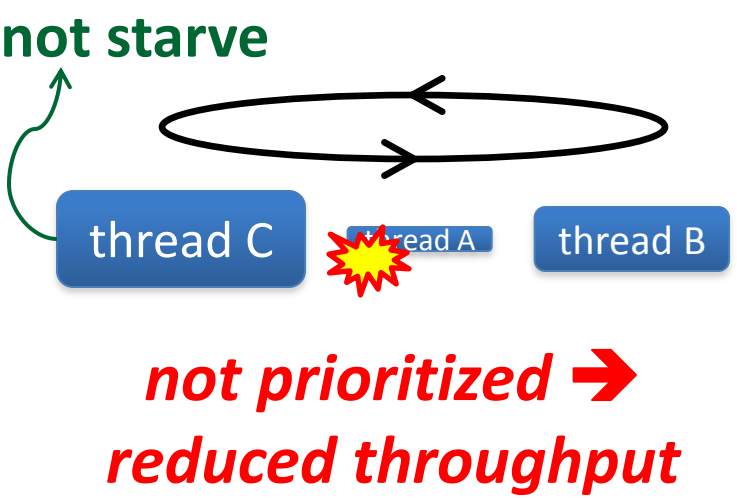
***Throughput biased** approach*

Prioritize less memory-intensive threads

***Fairness biased** approach*

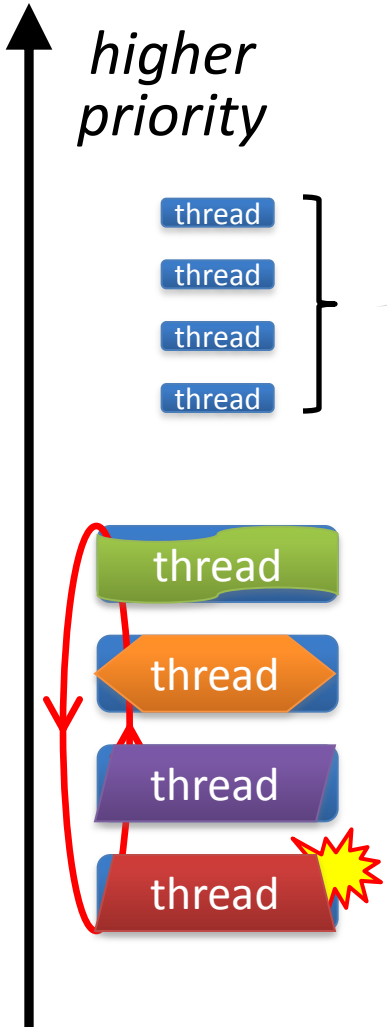
Take turns accessing memory





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

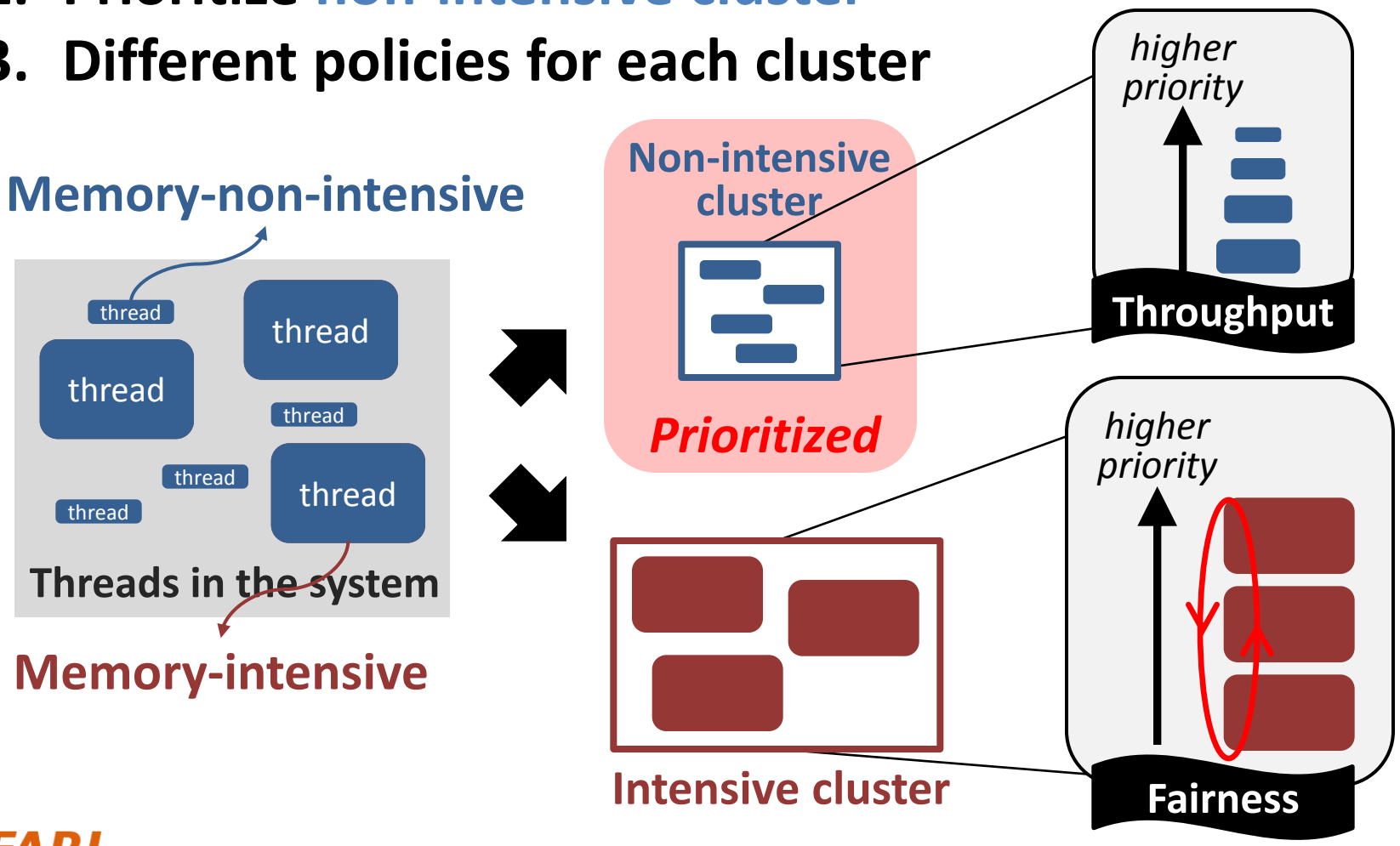
- Shuffle thread ranking

Memory-intensive threads have different vulnerability to interference

- Shuffle asymmetrically

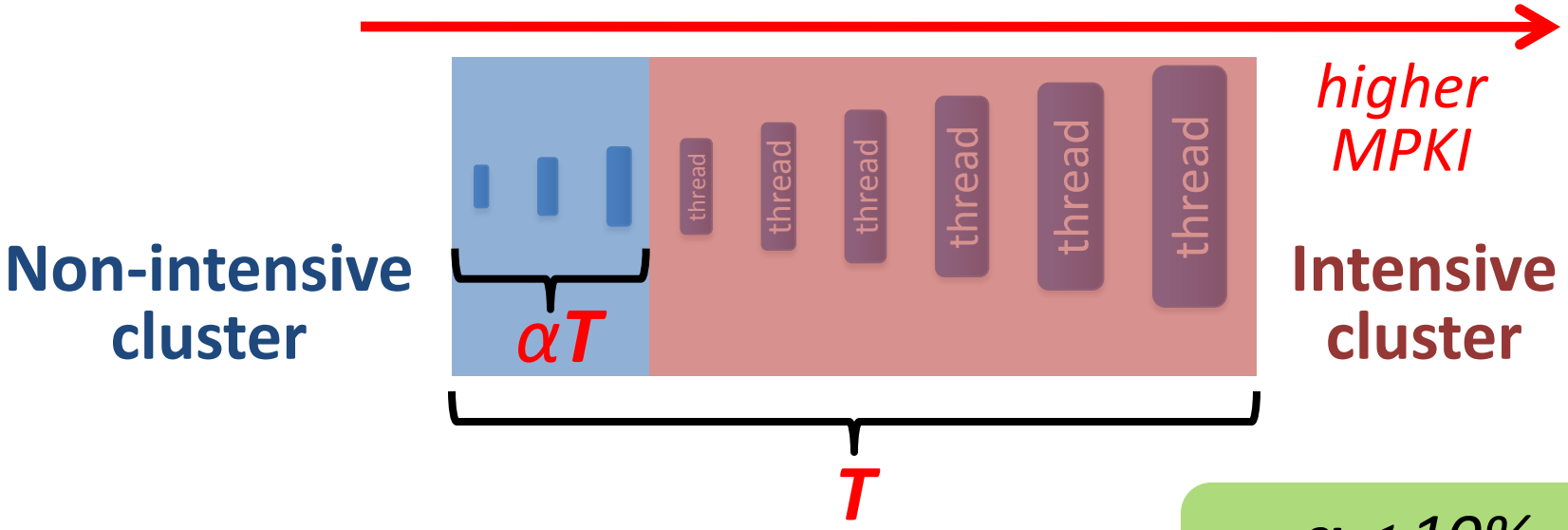
Thread Cluster Memory Scheduling [Kim+ MICRO'10]

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)

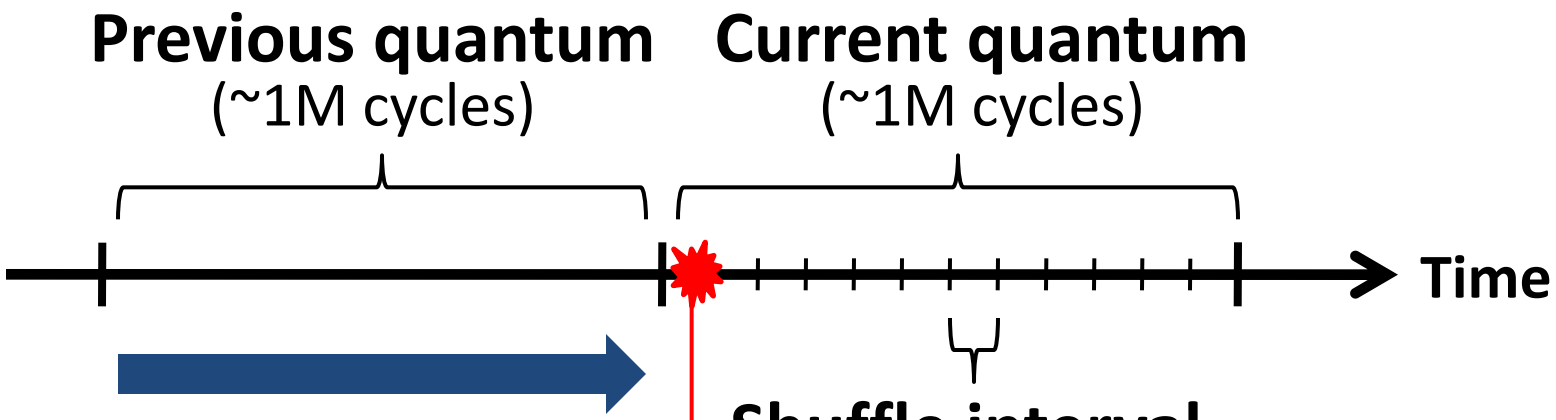


T **=** Total ***memory bandwidth usage***

α *< 10%*
ClusterThreshold

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

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

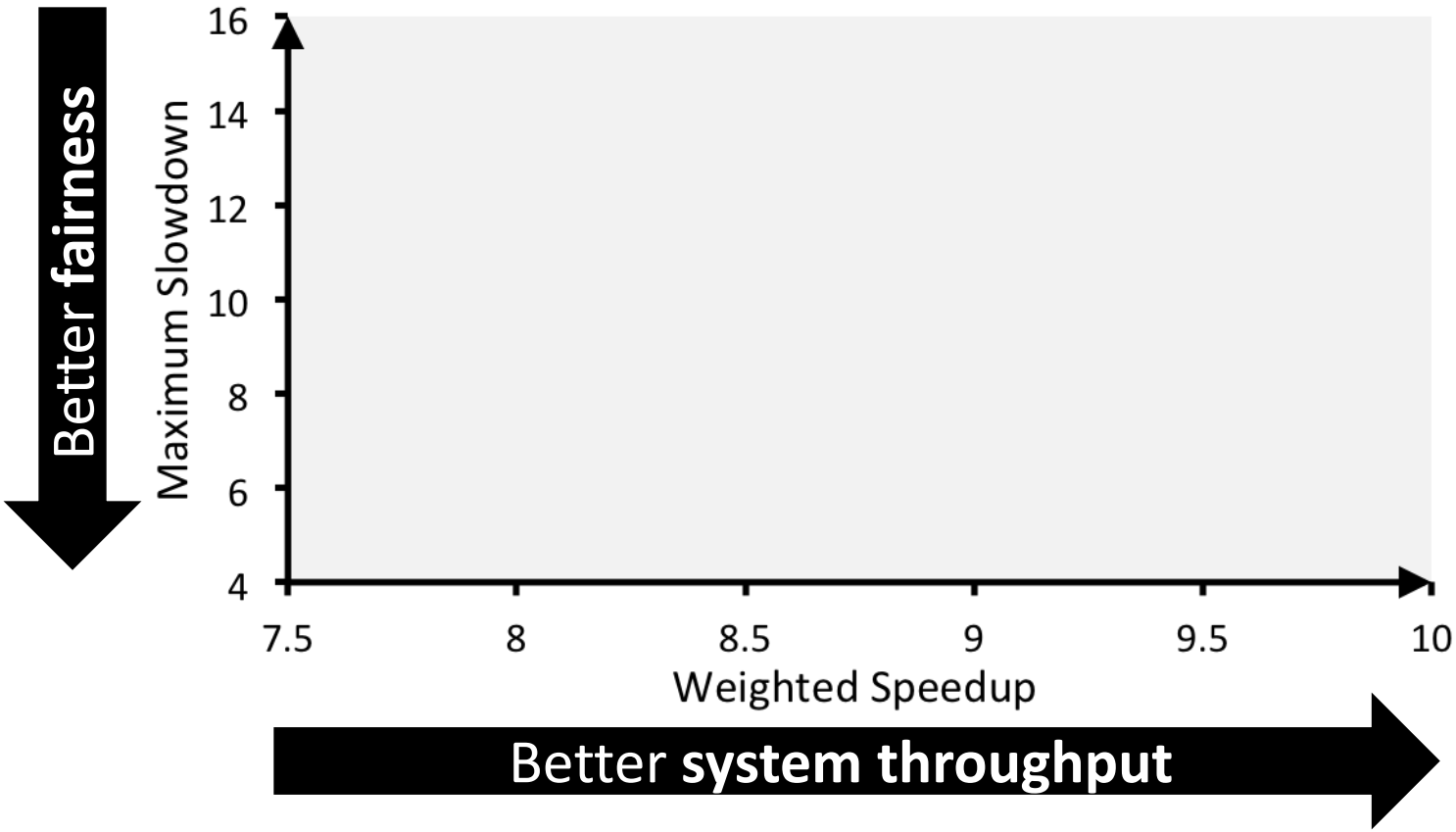
- **Non-Intensive** cluster **>** **Intensive** cluster
- **Non-Intensive** cluster: lower intensity ➔ higher rank
- **Intensive** cluster: rank shuffling

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

3. Oldest: Older requests are prioritized

TCM: Throughput and Fairness

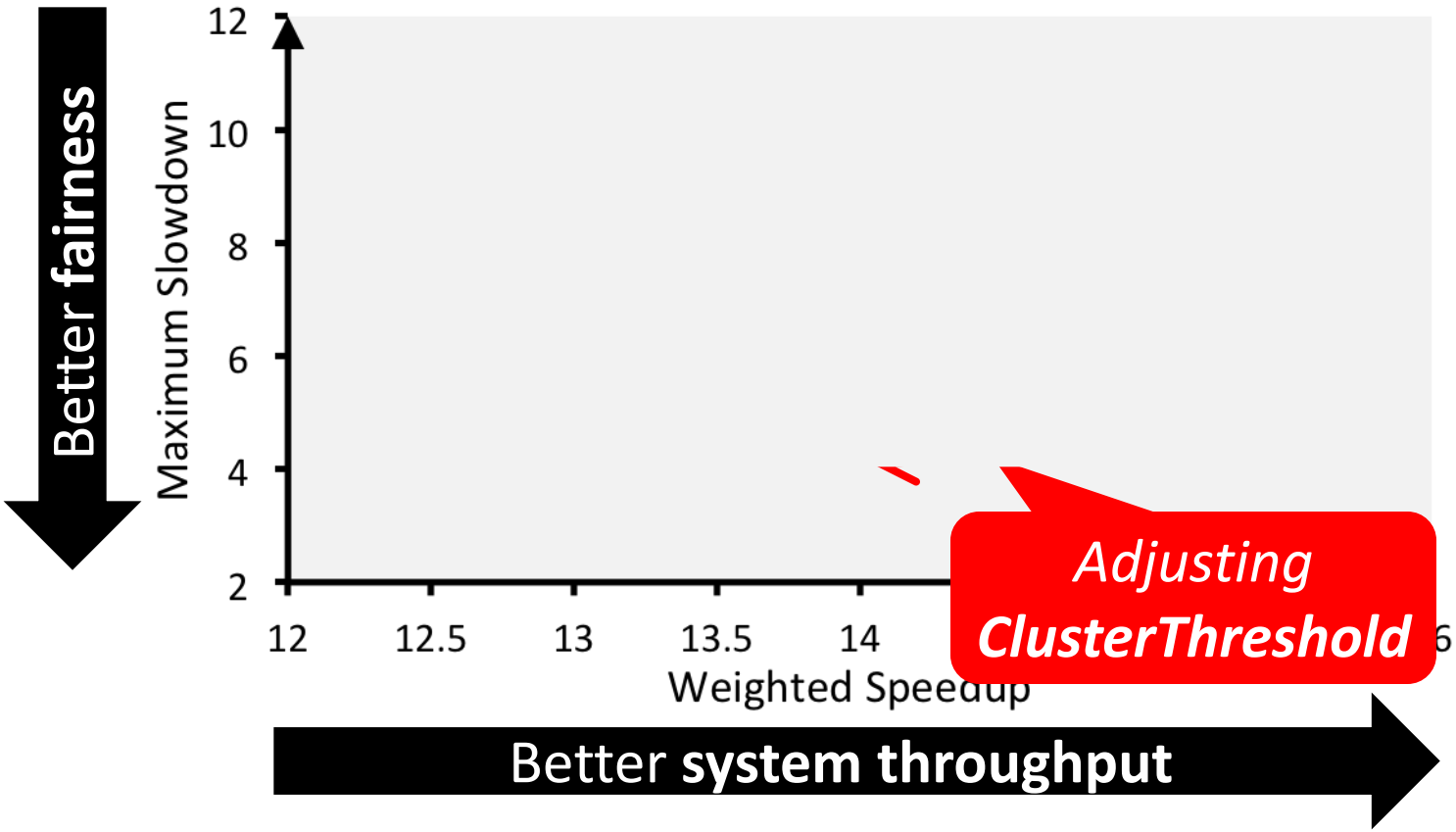
24 cores, 4 memory controllers, 96 workloads



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

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

Memory Control in CPU-GPU Systems

- **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

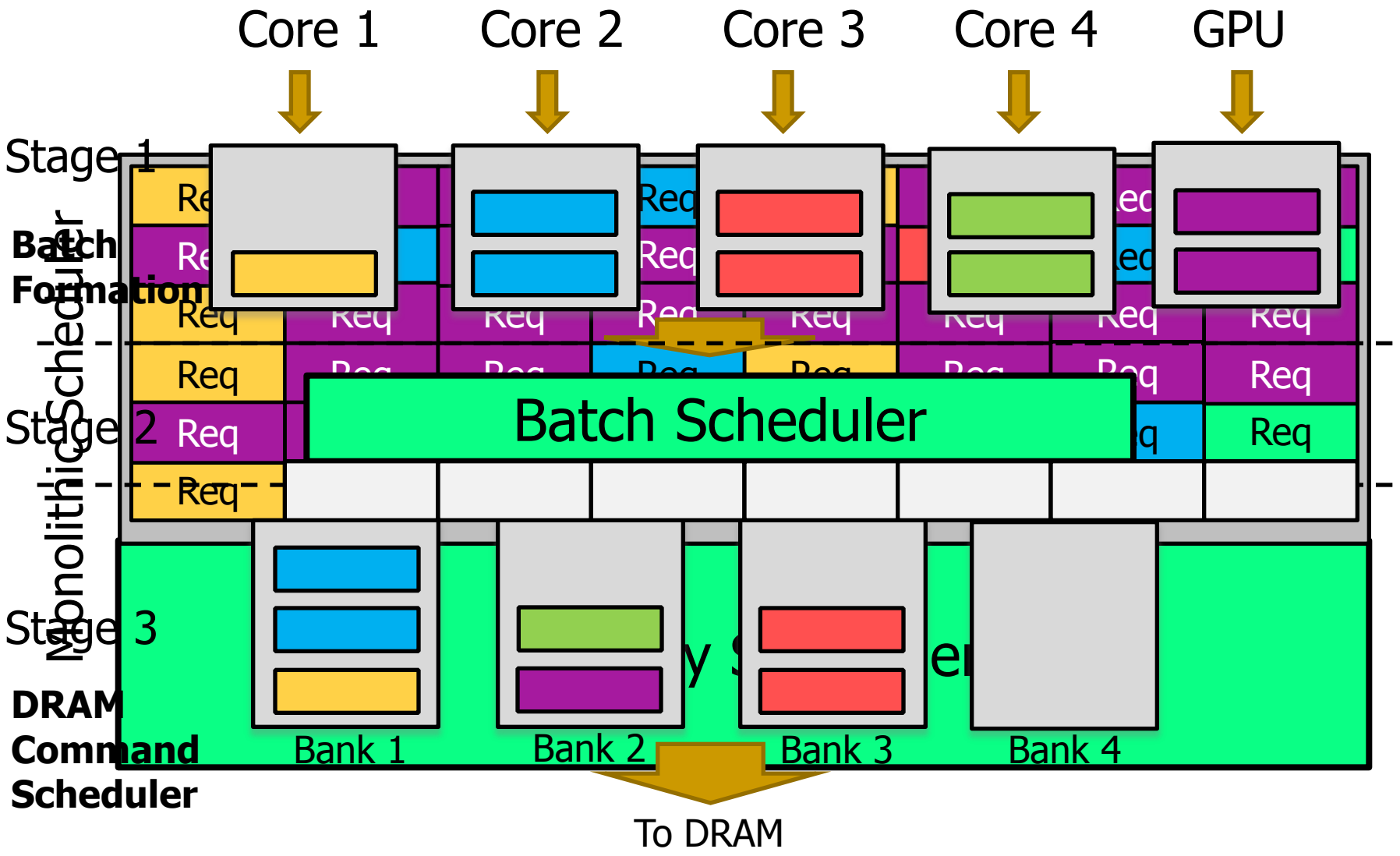
Ausavarungnirun et al., "Staged Memory Scheduling," ISCA 2012.

Key Idea: Decouple Tasks into Stages

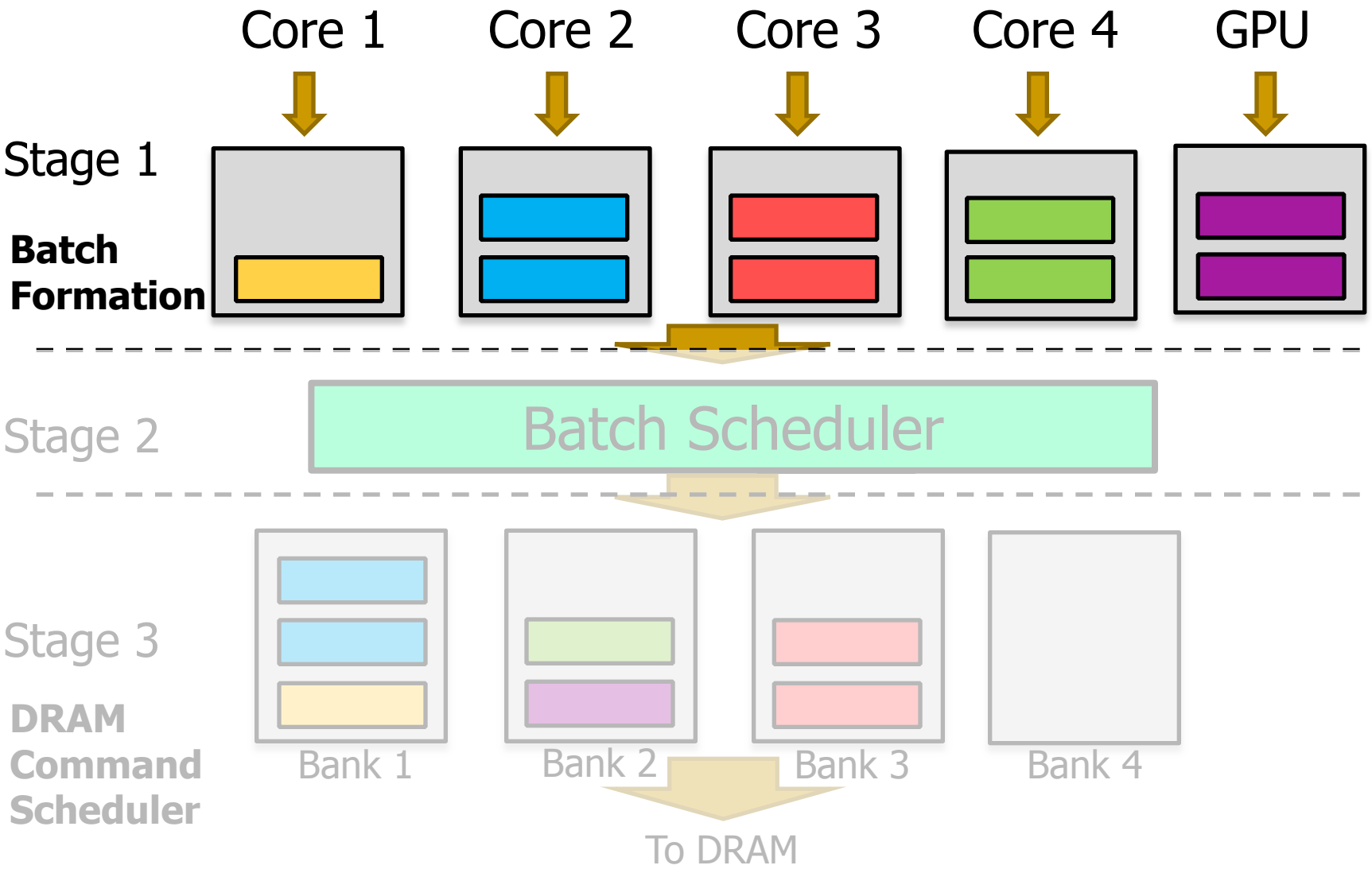
- Idea: Decouple the functional tasks of the memory controller
  - Partition tasks across several simpler HW structures (stages)

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

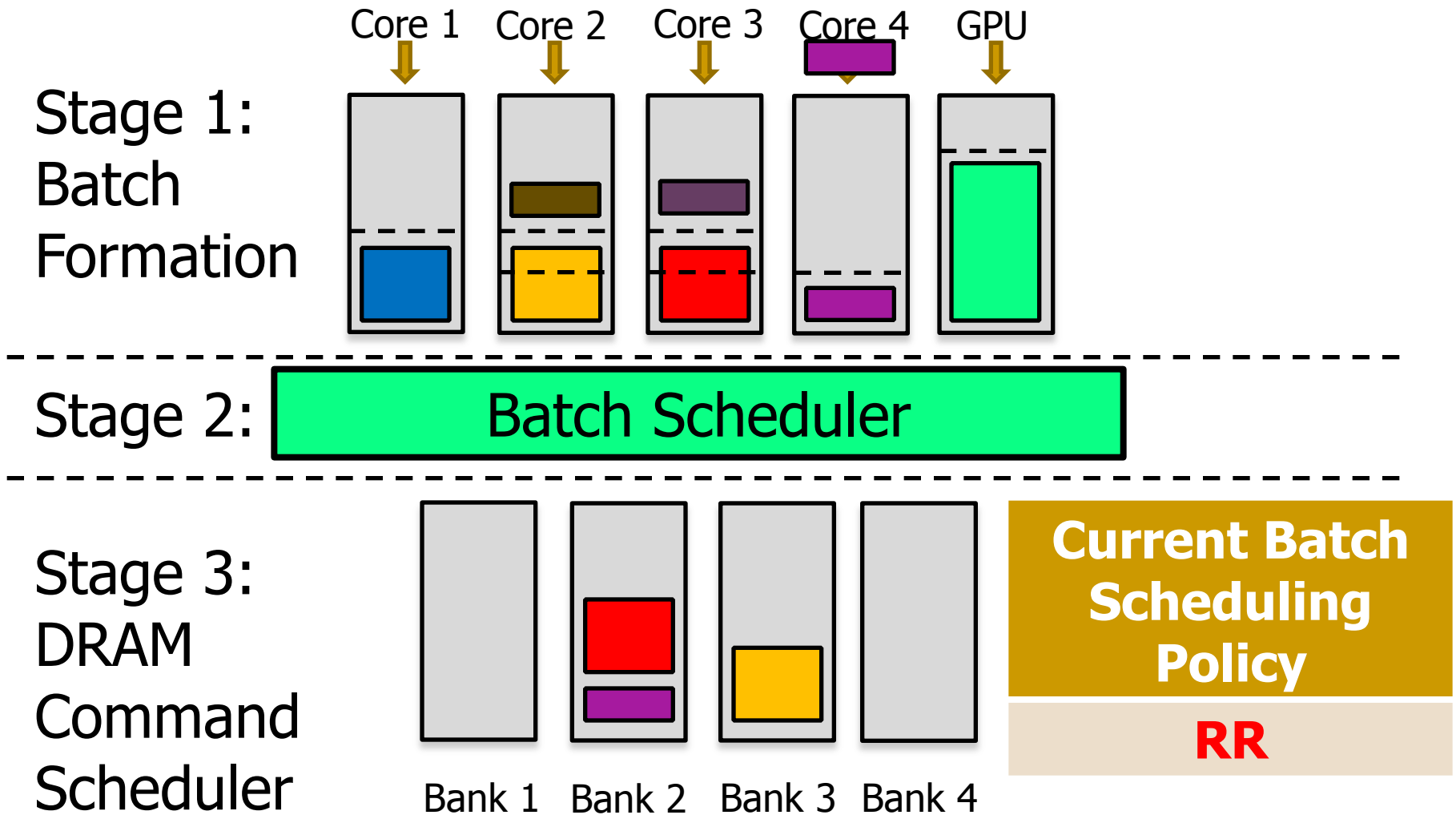
SMS: Staged Memory Scheduling



SMS: Staged Memory Scheduling



SMS: Staged Memory Scheduling



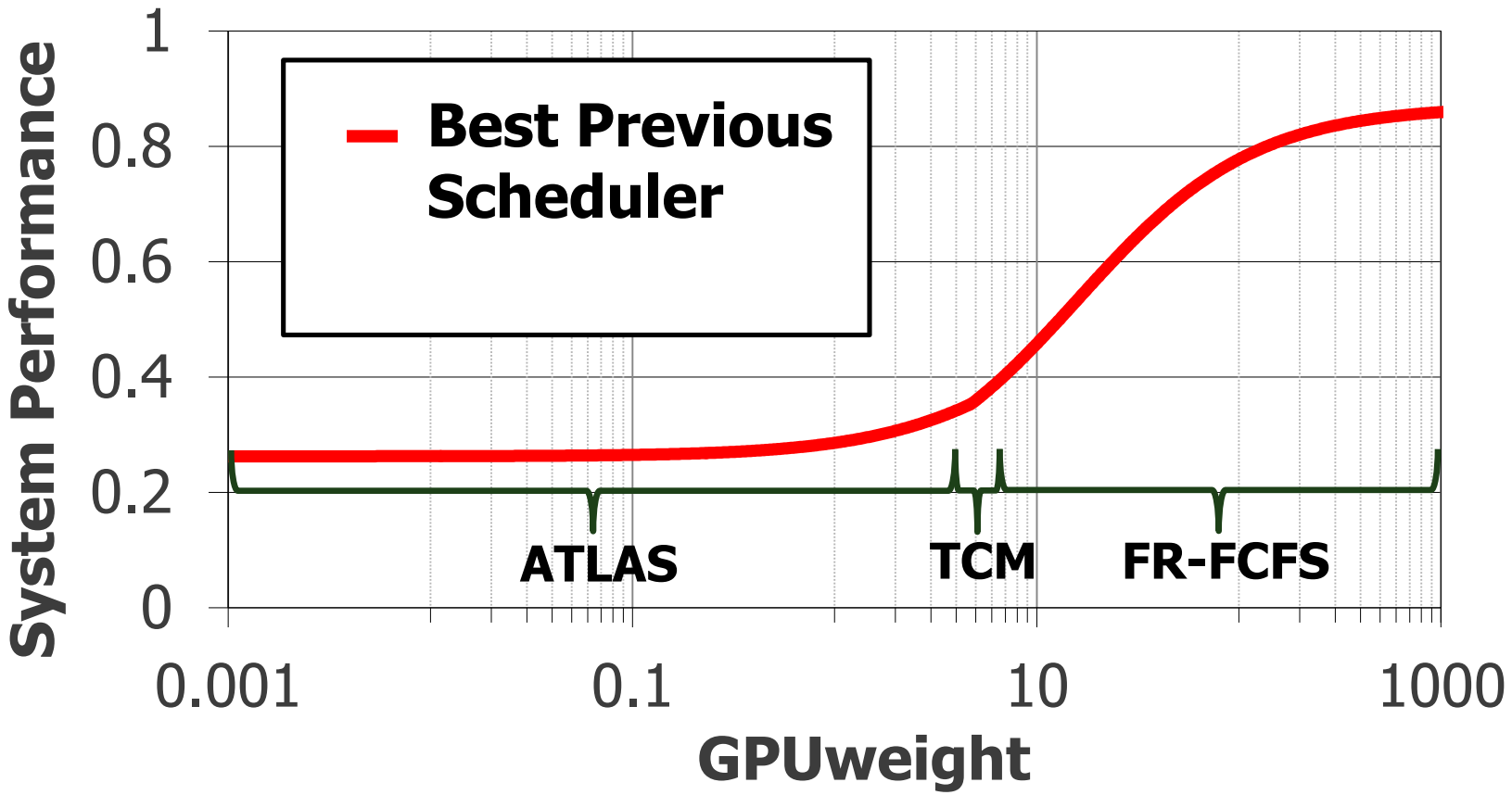
SMS Complexity

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

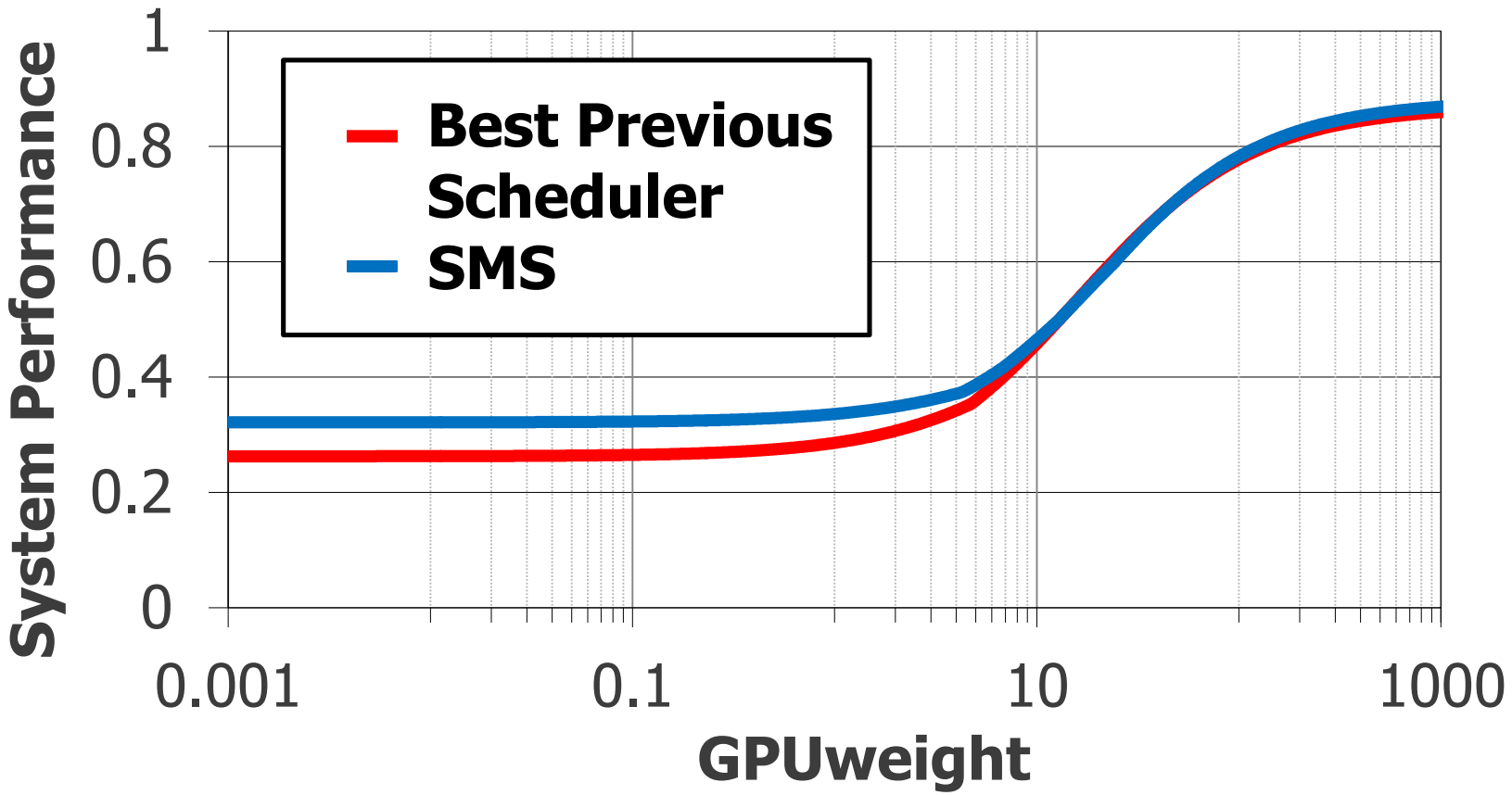
- Reduction comes from:
  - Monolithic scheduler → stages of simpler schedulers
  - Each stage has a simpler scheduler (considers fewer properties at a time to make the scheduling decision)
  - Each stage has simpler buffers (FIFO instead of out-of-order)
  - Each stage has a portion of the total buffer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

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

SMS Performance



SMS Performance



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

Memory QoS in a Parallel Application

- 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

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
  - 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

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

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)***,
Beijing, China, June 2008.

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?

Why are DRAM Controllers Difficult to Design?

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.

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

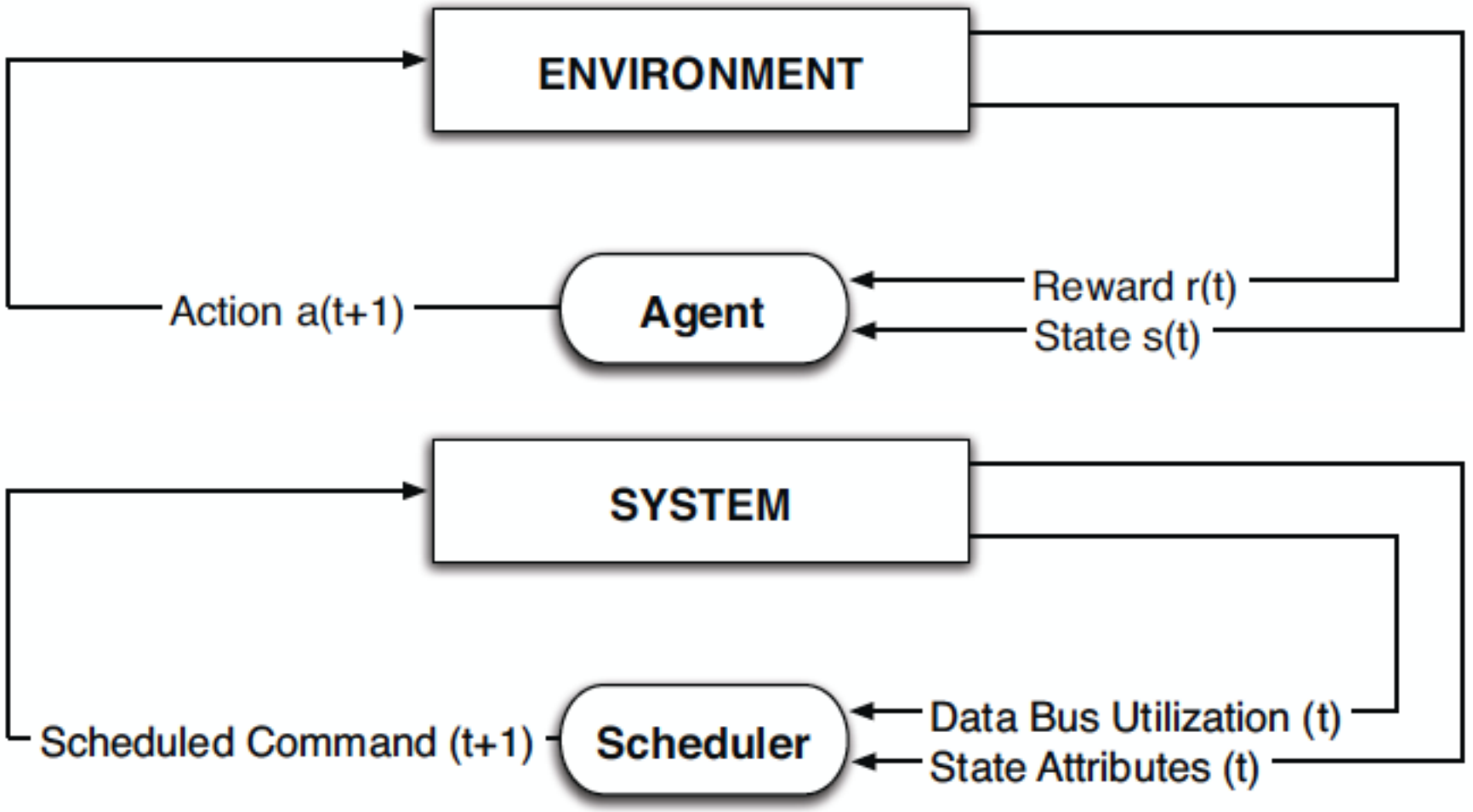


Figure 2: (a) Intelligent agent based on reinforcement learning principles; (b) DRAM scheduler as an RL-agent

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.

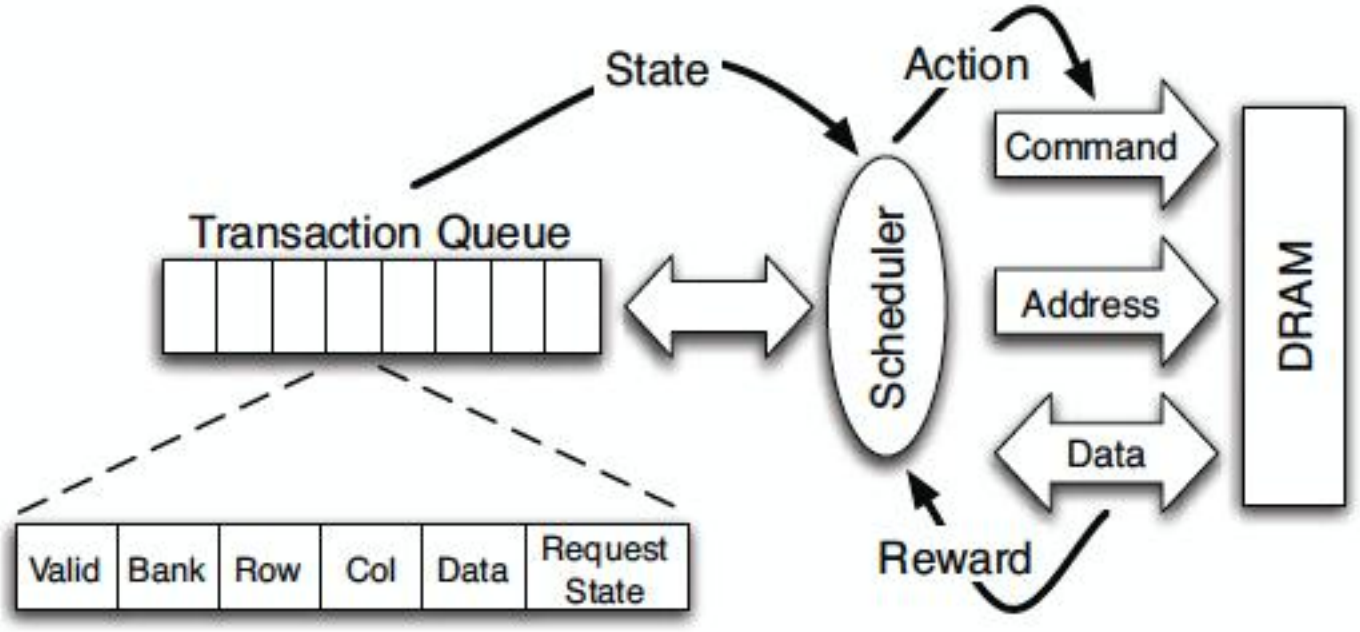


Figure 4: High-level overview of an RL-based scheduler.

Performance Results

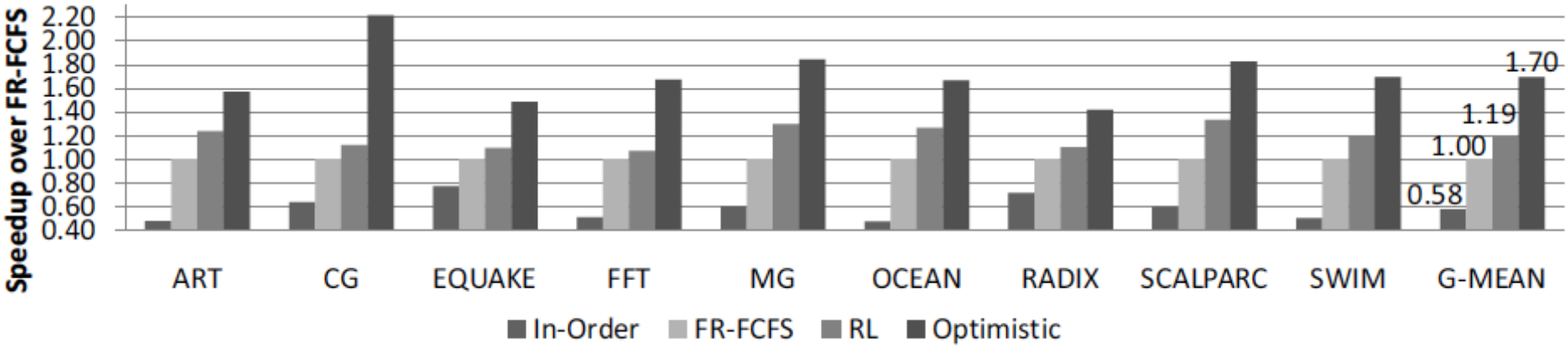


Figure 7: Performance comparison of in-order, FR-FCFS, RL-based, and optimistic memory controllers

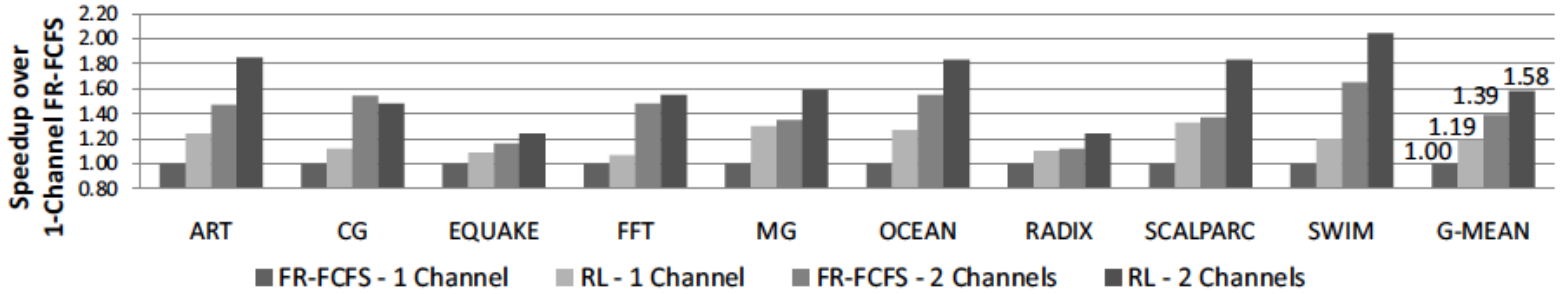


Figure 15: Performance comparison of FR-FCFS and RL-based memory controllers on systems with 6.4GB/s and 12.8GB/s peak DRAM bandwidth

DRAM-Aware Cache Design: An Example of Resource Coordination

Chang Joo Lee, Veynu Narasiman, Eiman Ebrahimi, Onur Mutlu, and Yale N. Patt,
"DRAM-Aware Last-Level Cache Writeback: Reducing Write-Caused Interference in Memory Systems"
HPS Technical Report, TR-HPS-2010-002, April 2010.

DRAM-Aware LLC Writeback

- Problem 1: Writebacks to DRAM interfere with reads and cause additional performance penalty
  - Write-to-read turnaround time in DRAM bus
  - Write-recovery latency in DRAM bank
  - Change of row buffer → reduced row-buffer locality for read requests
- Problem 2: Writebacks that occur once in a while have low row buffer locality
- Idea: When evicting a dirty cache block to a row, proactively search the cache for other dirty blocks to the same row → evict them → write to DRAM in a batch
  - Improves row buffer locality
  - Reduces write-to-read switching penalties on DRAM bus
  - Improves performance on both single-core and multi-core systems

More Information

- Chang Joo Lee, Veynu Narasiman, Eiman Ebrahimi, Onur Mutlu, and Yale N. Patt,
  "DRAM-Aware Last-Level Cache Writeback: Reducing Write-Caused Interference in Memory Systems"
  HPS Technical Report, TR-HPS-2010-002, April 2010.

DRAM-Aware Last-Level Cache Writeback: Reducing Write-Caused Interference in Memory Systems

Abstract

Read and write requests from a processor contend for the main memory data bus. System performance depends heavily on when read requests are serviced since they are required for an application's forward progress whereas writes do not need to be performed immediately. However, writes eventually have to be written to memory because the storage required to buffer them on-chip is limited.

In modern high bandwidth DDR (Double Data Rate)-based memory systems write requests significantly interfere with the servicing of read requests by delaying the more critical read requests and by causing the memory bus to become idle when switching between the servicing of a write and read request. This interference significantly degrades overall system performance. We call this phenomenon write-caused interference. To reduce write-caused interference, this paper proposes a new last-level cache writeback policy, called DRAM-aware writeback. The key idea of the proposed technique is to aggressively send out writeback requests that are expected to hit in DRAM row buffers before they would normally be evicted by the last-level cache replacement policy and have the DRAM controller service as many writes as possible together. Doing so not only reduces the amount of time to service writes by improving their row buffer locality but also reduces the idle bus cycles wasted due to switching between the servicing of a write and a read request.

DRAM-aware writeback improves system performance by 7.1% and 12.8% on single and 4-core systems respectively. The performance benefits of the mechanism increases in systems with prefetching since such systems have higher contention between reads and writes in the DRAM system.

DRAM-aware Cache Design

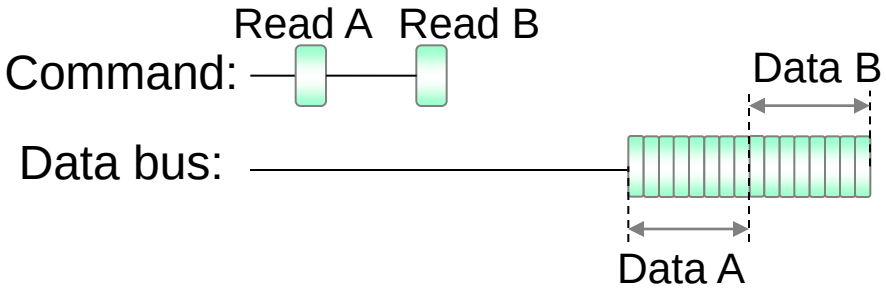
- Coordination of cache policies with memory controllers

- Chang Joo Lee, Veynu Narasiman, Eiman Ebrahimi, Onur Mutlu, and Yale N. Patt,
  "DRAM-Aware Last-Level Cache Writeback: Reducing Write-Caused Interference in Memory Systems"
  HPS Technical Report, TR-HPS-2010-002, April 2010.

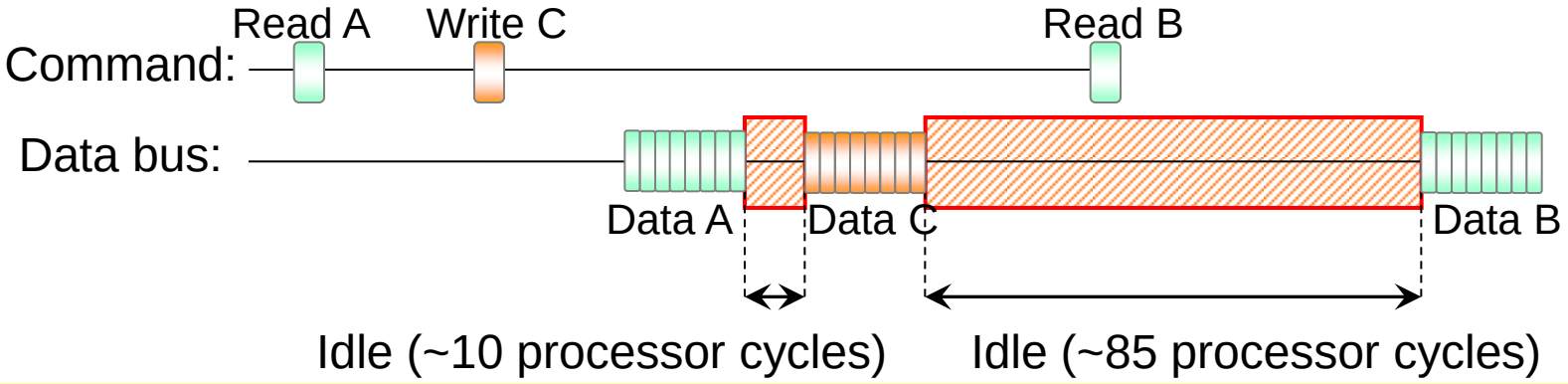
- Chang Joo Lee, Eiman Ebrahimi, Veynu Narasiman, Onur Mutlu, and Yale N. Patt,
  "DRAM-Aware Last-Level Cache Replacement"
  HPS Technical Report, TR-HPS-2010-007, December 2010.

Write-Caused Interference: Read-Write Switching

- Row-hit read-to-read (write-to-write) to ***any*** bank: back-to-back data transfer



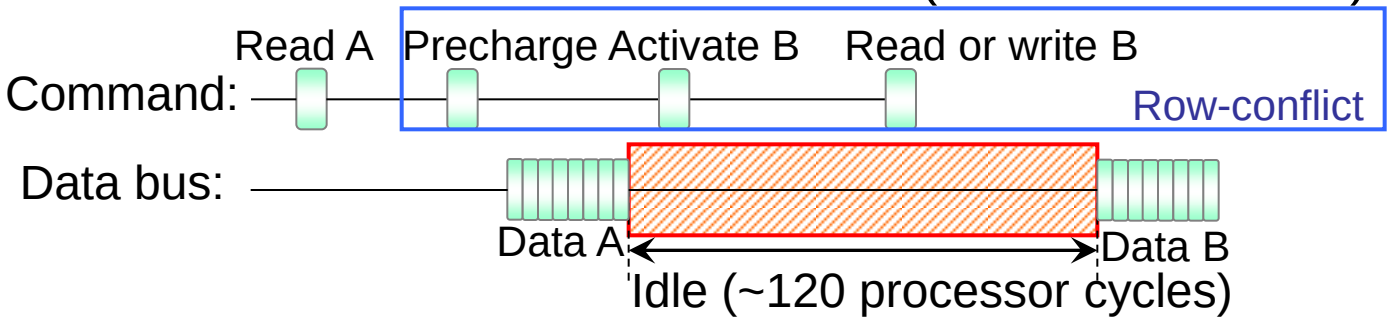
- Read-write switching penalty for requests to ***any*** bank



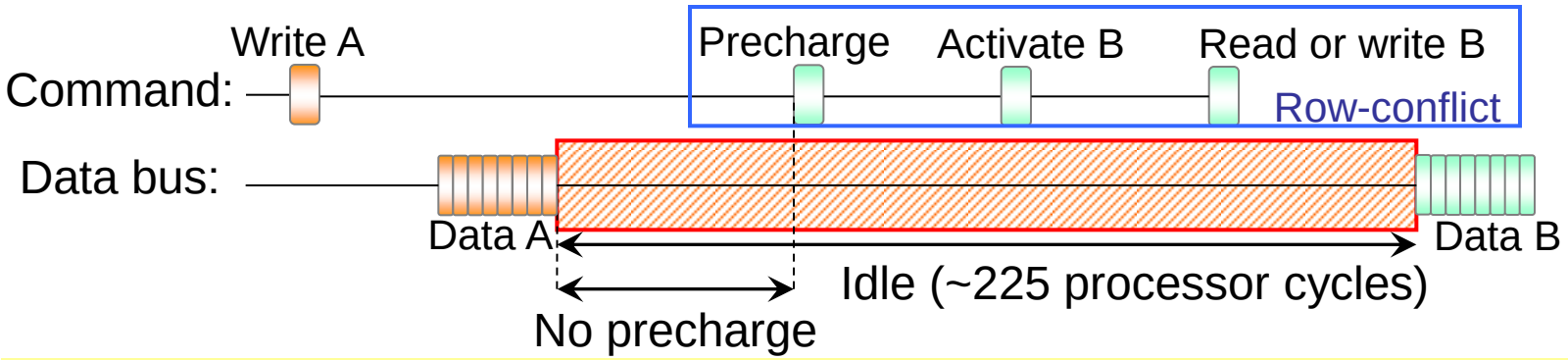
Frequent read-write switching incurs many idle cycles

Write-Caused Interference: Write-to-Row-Conflict

- Row-conflict after read (in the ***same*** bank)



- Row-conflict after write (in the ***same*** bank)



Row-conflict after a write causes more idle cycles

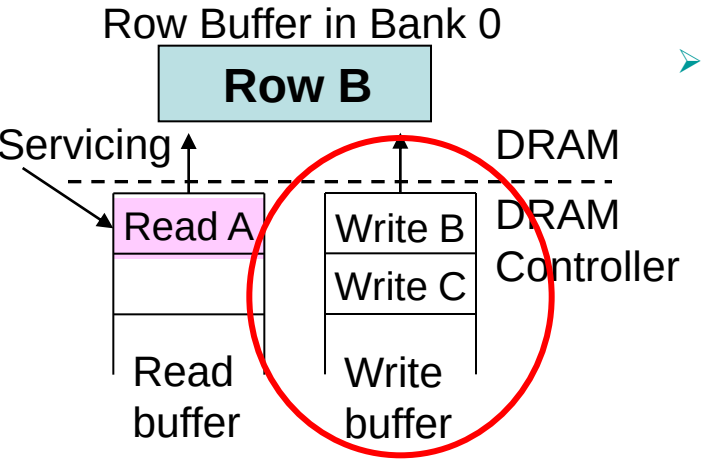
Write-Caused Interference

- **Read-Write Switching**
  - **Frequent read-write switching incurs many idle cycles**
- **Write-to-Row-Conflict**
  - **A row-conflict after a write causes more idle cycles**

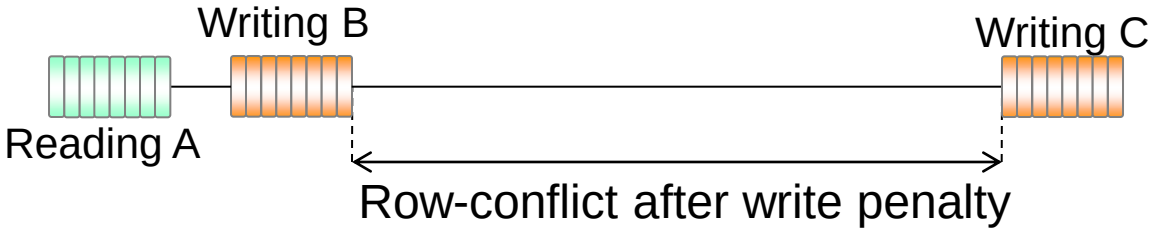
Generating many row-hit writes rather than row-conflict writes is preferred

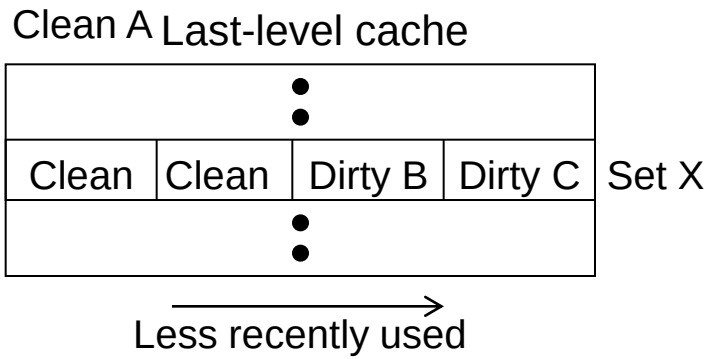
LRU vs. Interference-Aware Replacement

All requests are to the *same* cache set



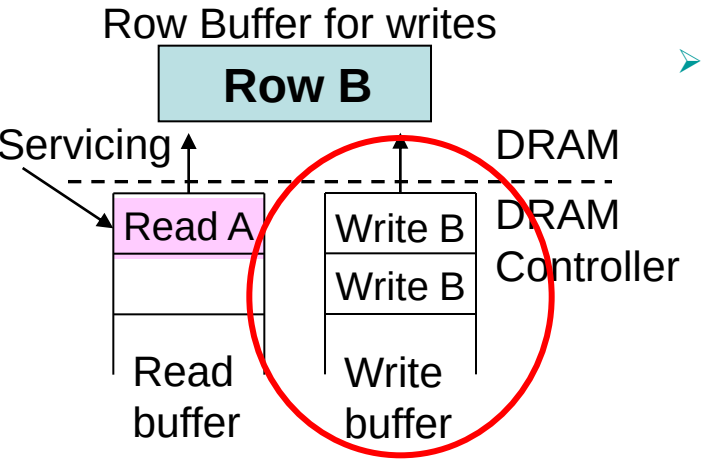
- **Conventional LRU:**
  Write B (row-hit), Write C (row-conflict)





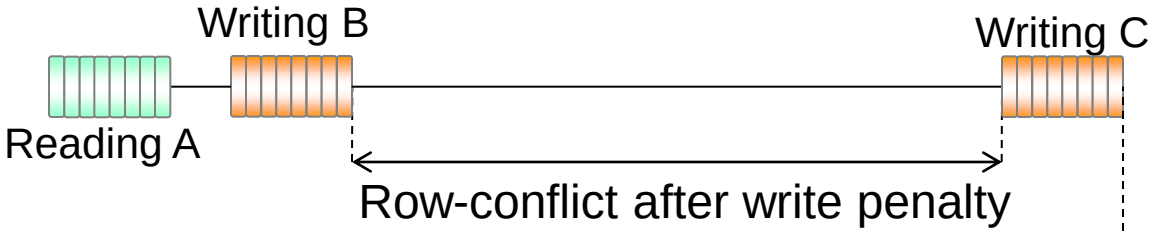
LRU vs. Interference-Aware Replacement

All requests are to the *same* cache set



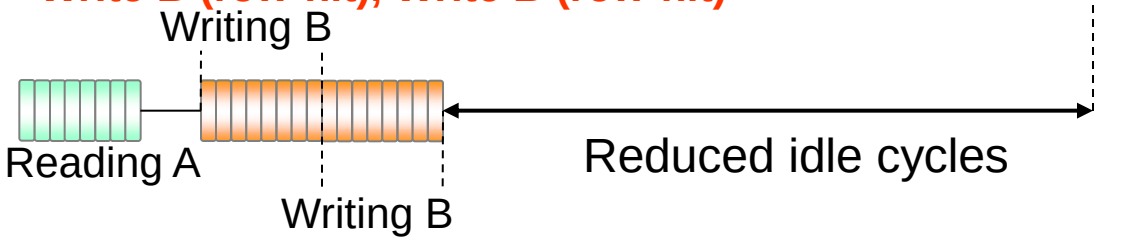
- **Conventional LRU:**

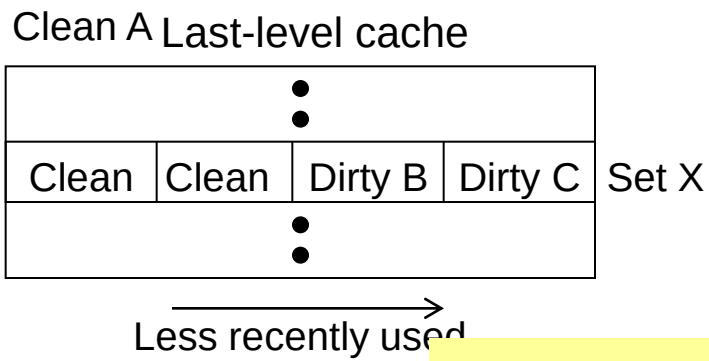
Write B (row-hit), Write C (row-conflict)



- **Interference-aware:**

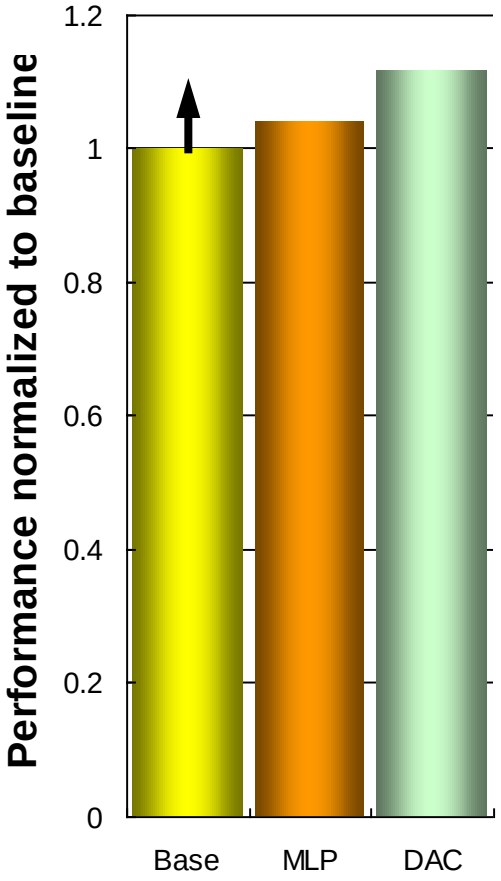
Write B (row-hit), Write B (row-hit)



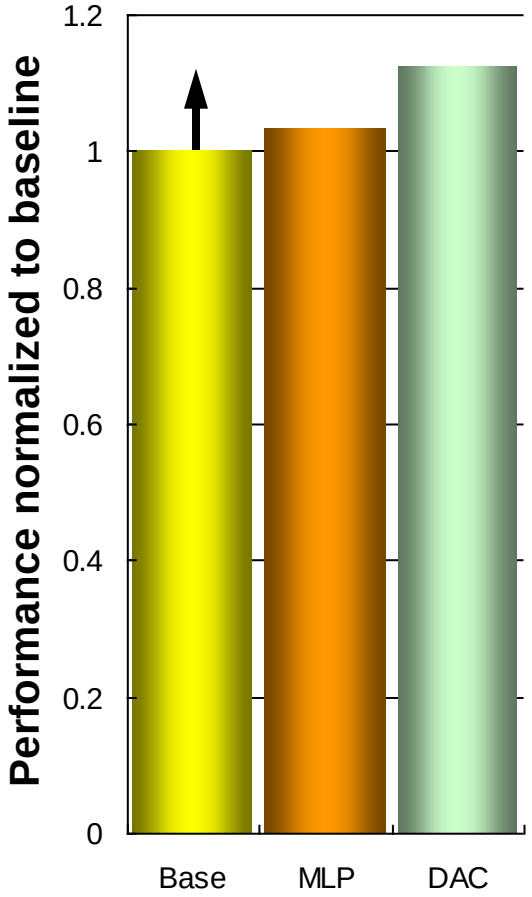


A simple policy can reduce write service time

Performance of DRAM-Aware Replacement



1-core **11.4%**



4-core **12.3%**

Outline

- Problem
- **Solutions**
  - Prefetch-Aware DRAM Controller
  - BLP-Aware Request Issue Policies
  - DRAM-Aware Cache Replacement
  - **DRAM-Aware Writeback**
- Combination of Solutions
- Related Work
- Conclusion

DRAM-Aware Writeback

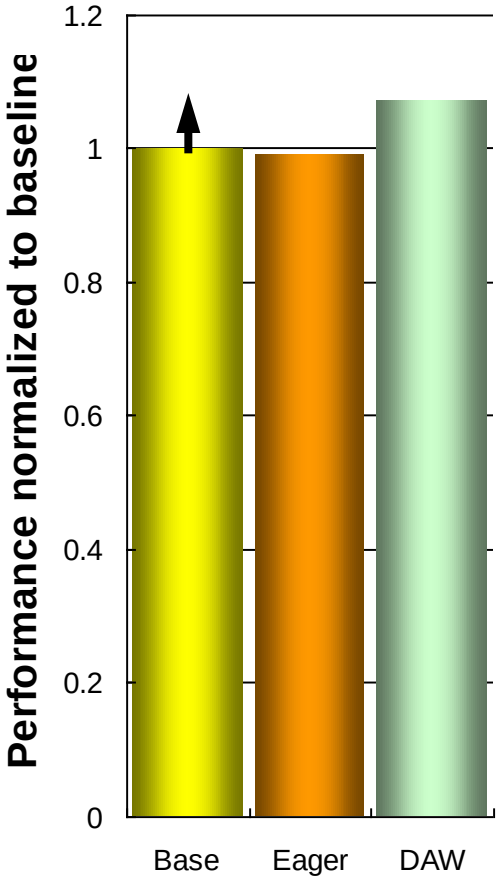
- Write-caused interference-aware replacement is not enough
  - Row-hit writebacks are sent only when a replacement occurs
    - Lose opportunities to service more writes quickly
- **To minimize write-caused interference, *proactively* clean row-hit dirty lines**
  → Reads are serviced without write-caused interference for a longer period

DRAM-Aware Writeback

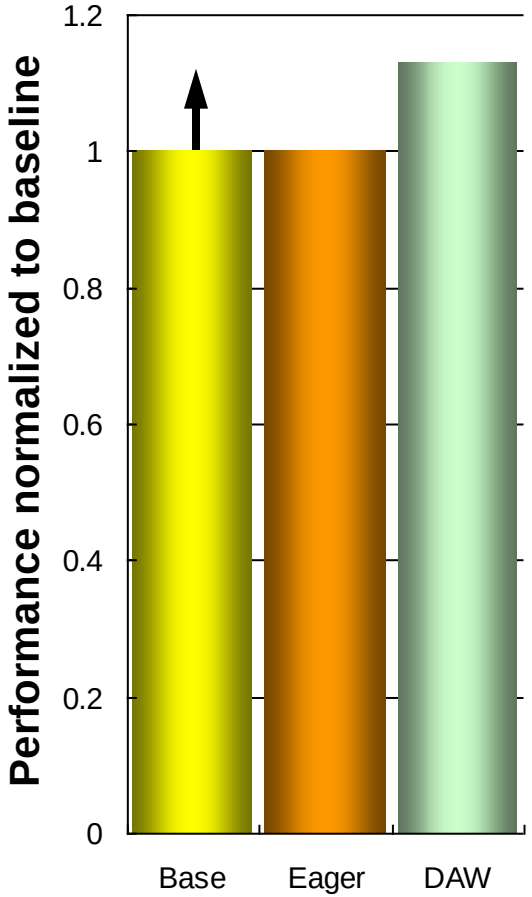
1. When a dirty line is evicted for the last-level cache, store its address
2. Using the stored address, search all possible sets for row-hit dirty lines and clean them whenever the cache bank is idle

- Many row-hit writes (up to the row size) are m serviced quickly
  - Reads can be serviced for a longer time without being interfered with by writes

Performance of DRAM-Aware Writeback



1-core **7.1%**



4-core **12.8%**

Fairness via Source Throttling

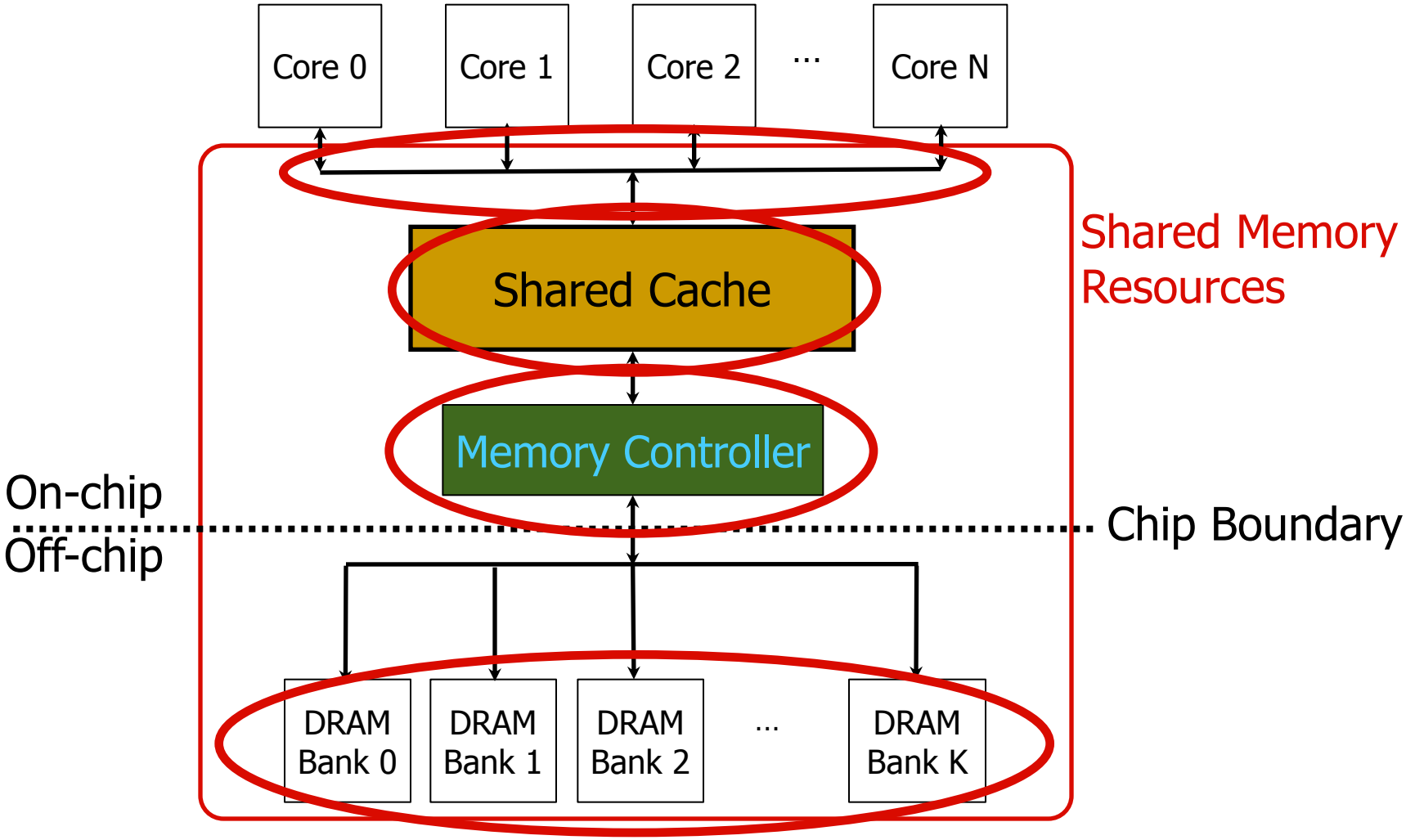
Eiman Ebrahimi, Chang Joo Lee, Onur Mutlu, and Yale N. Patt,

"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

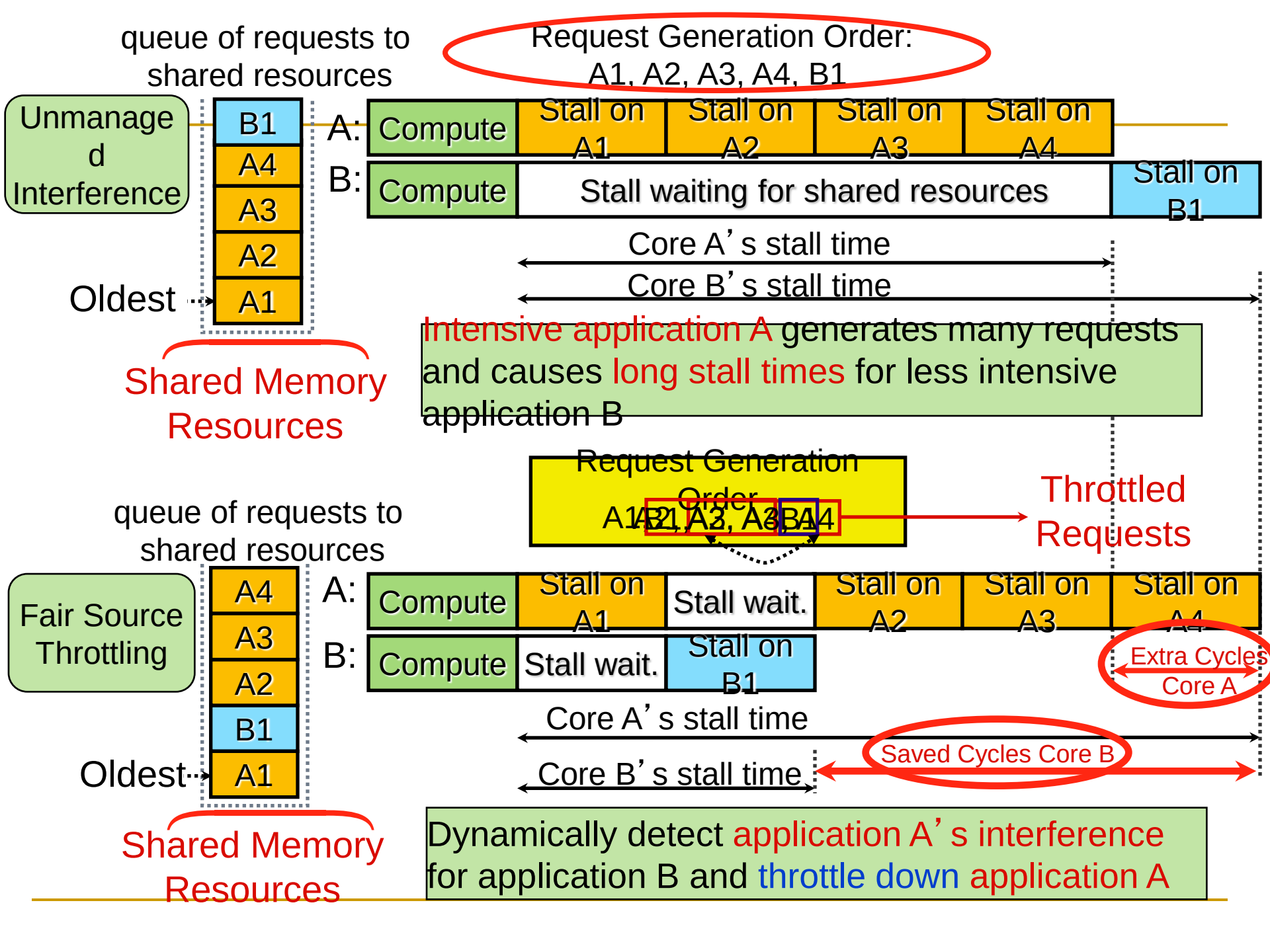
- How do we enable fair sharing of the entire memory system by controlling interference in a coordinated manner?

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, B1, A4
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

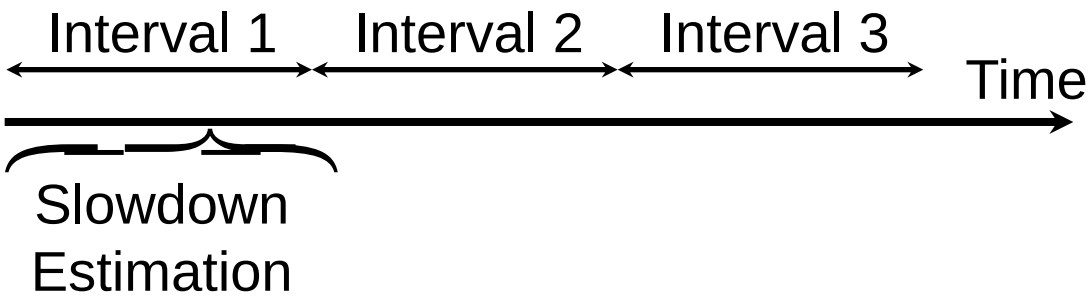
Fairness via Source Throttling (FST)

- 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)

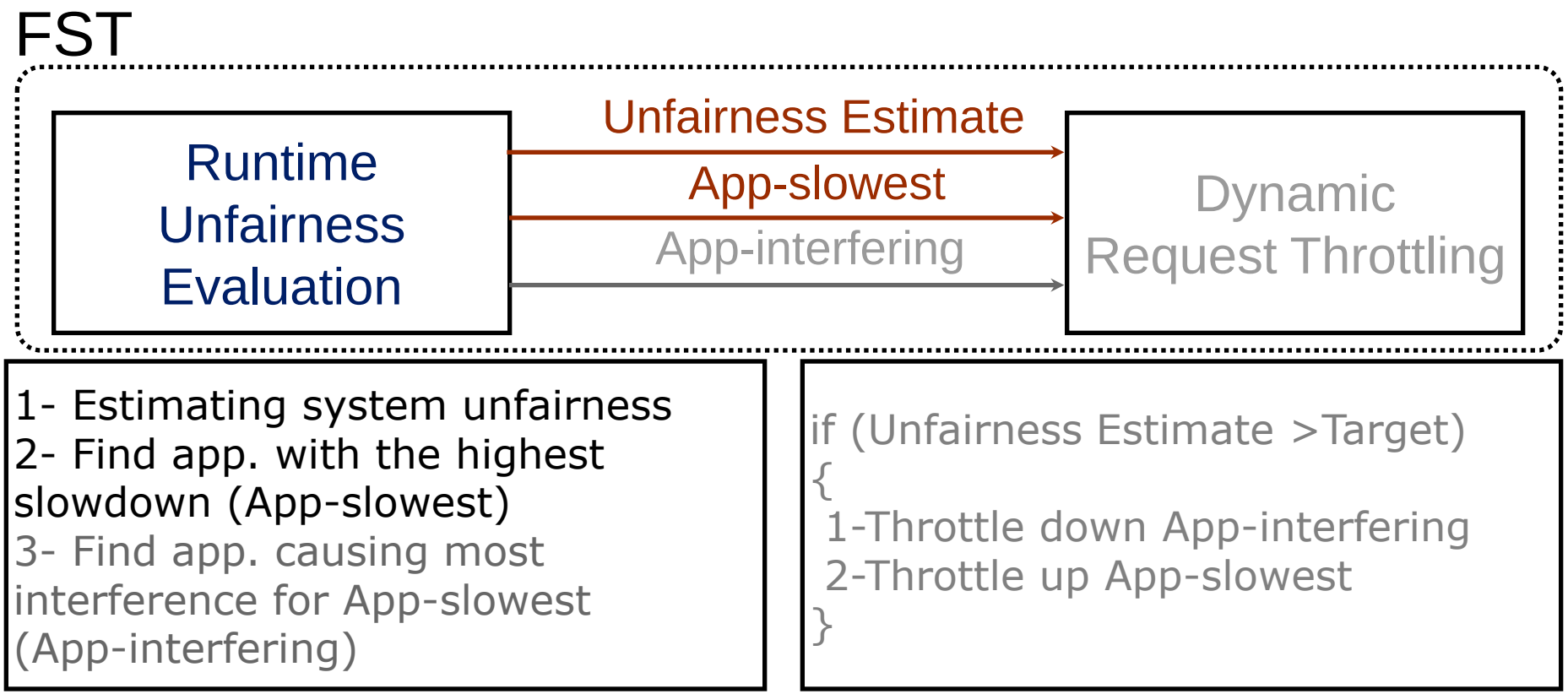


FST

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
}
```

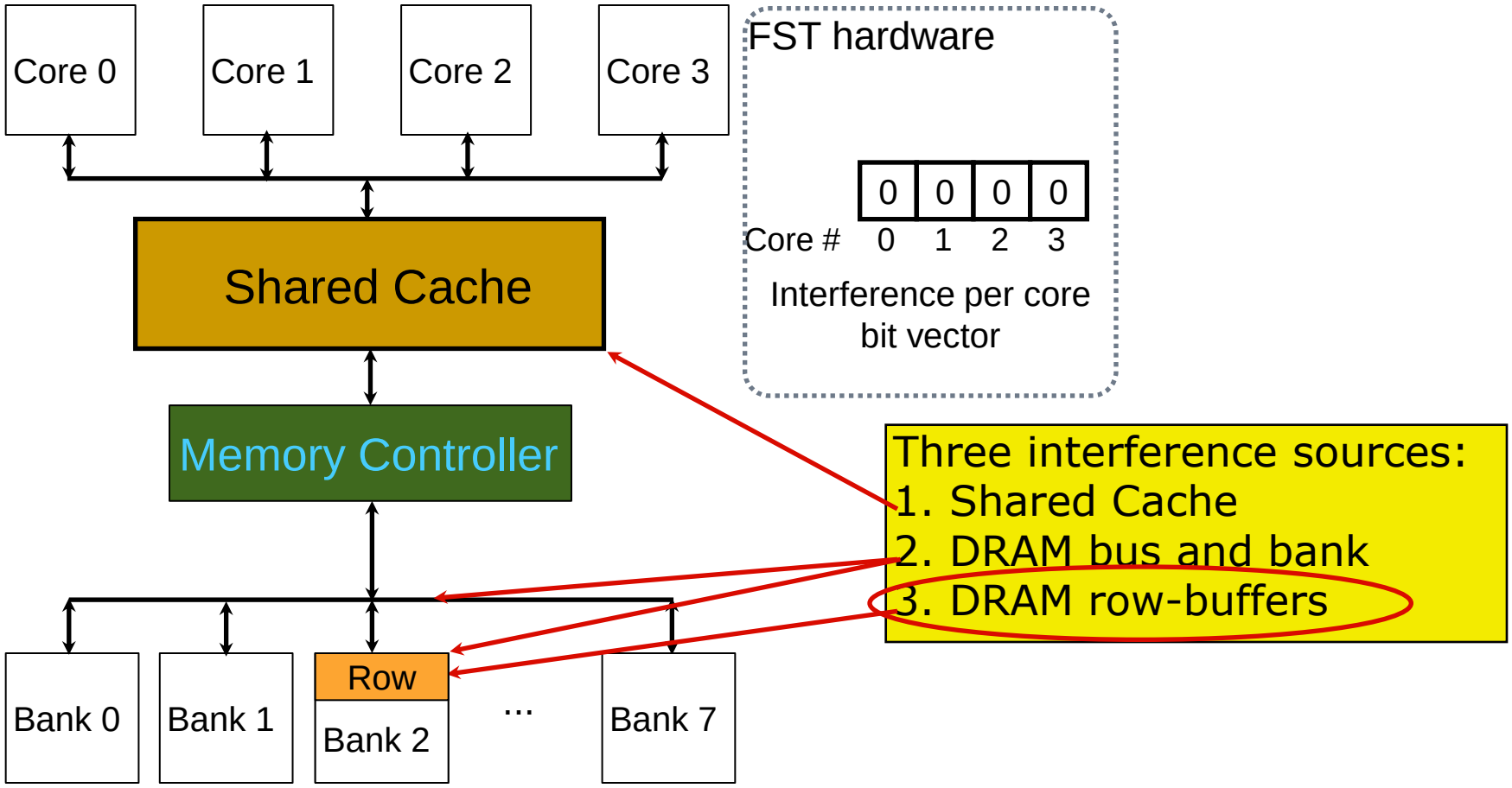
Fairness via Source Throttling (FST)



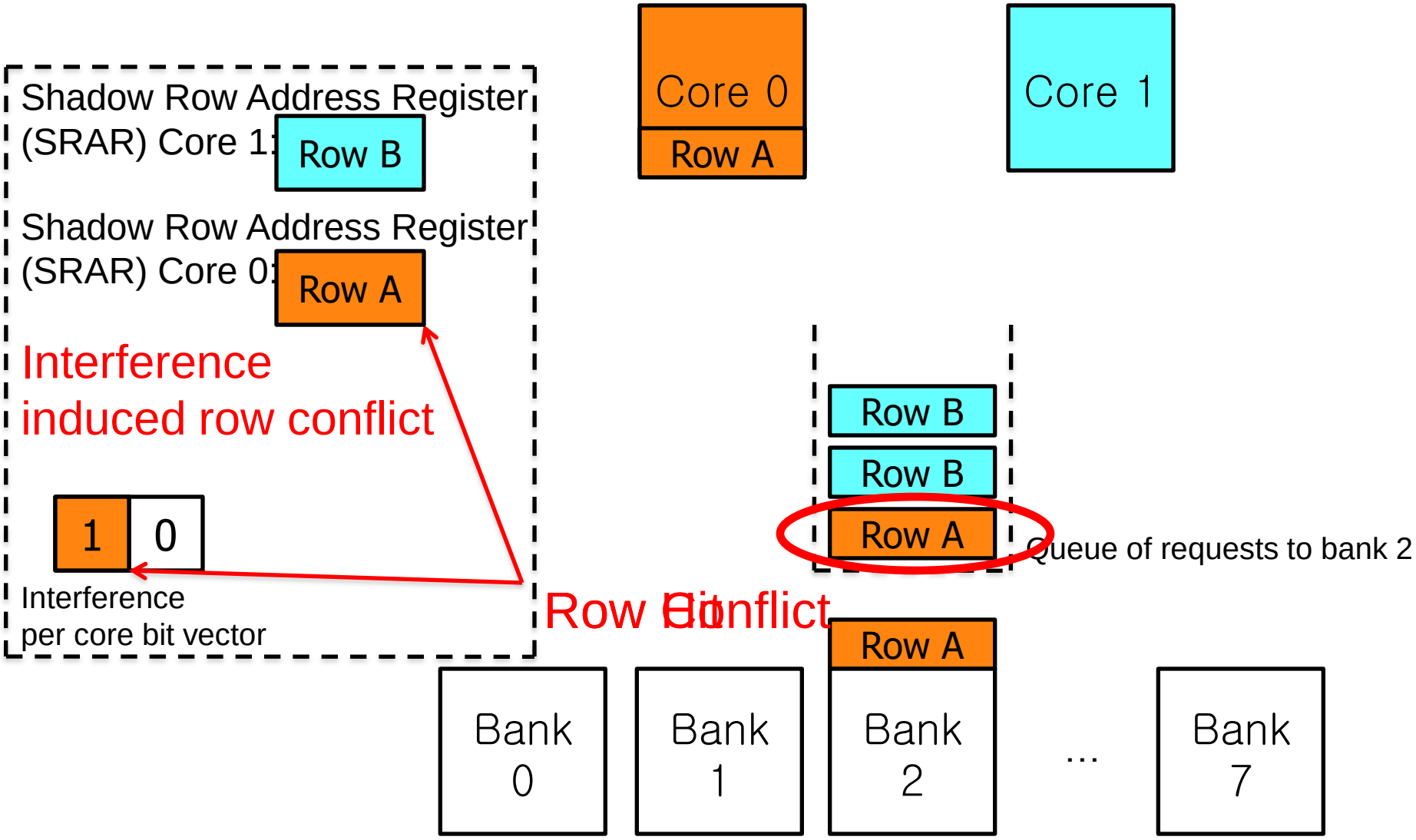
Estimating System Unfairness

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

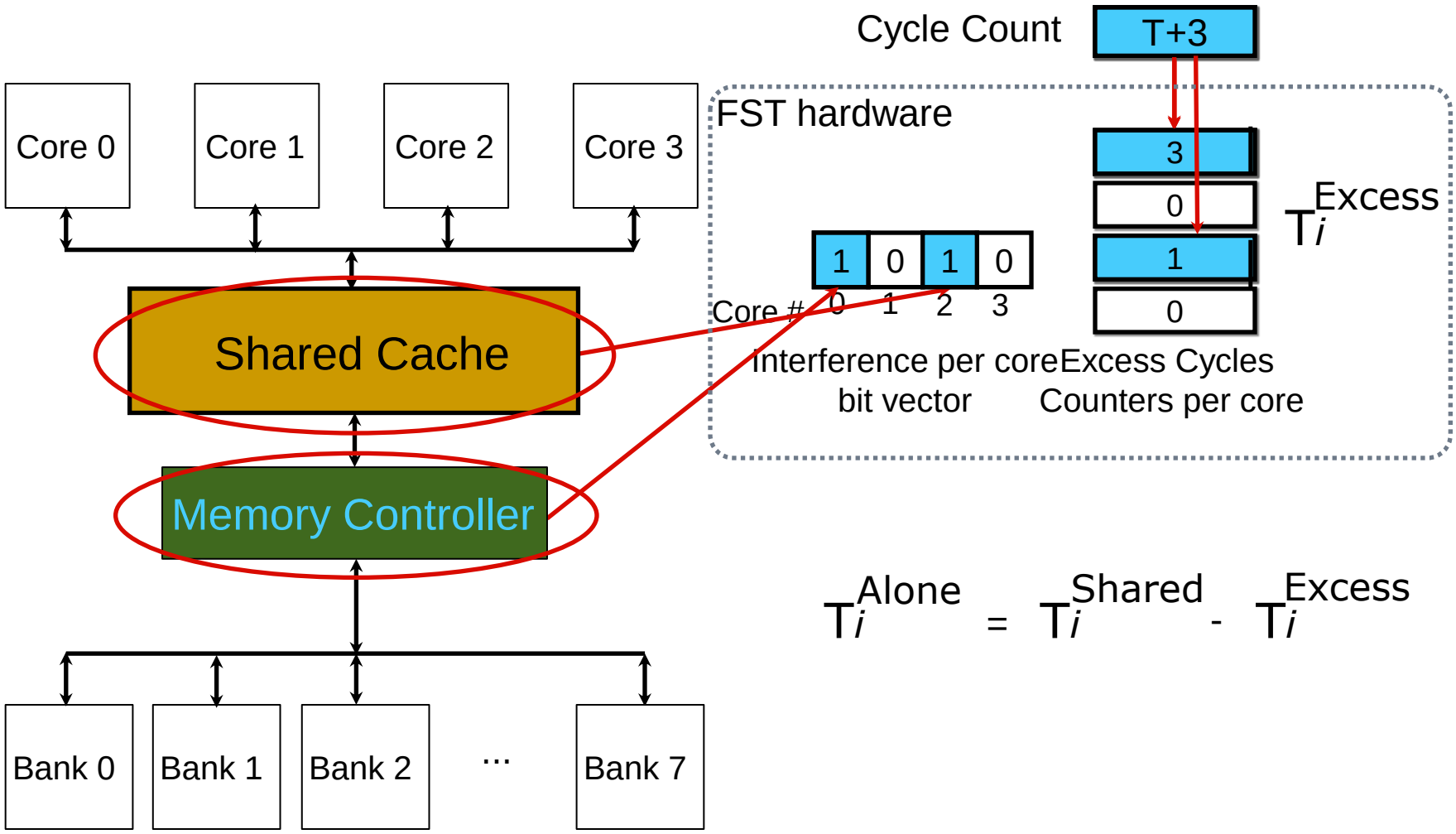
Tracking Inter-Core Interference



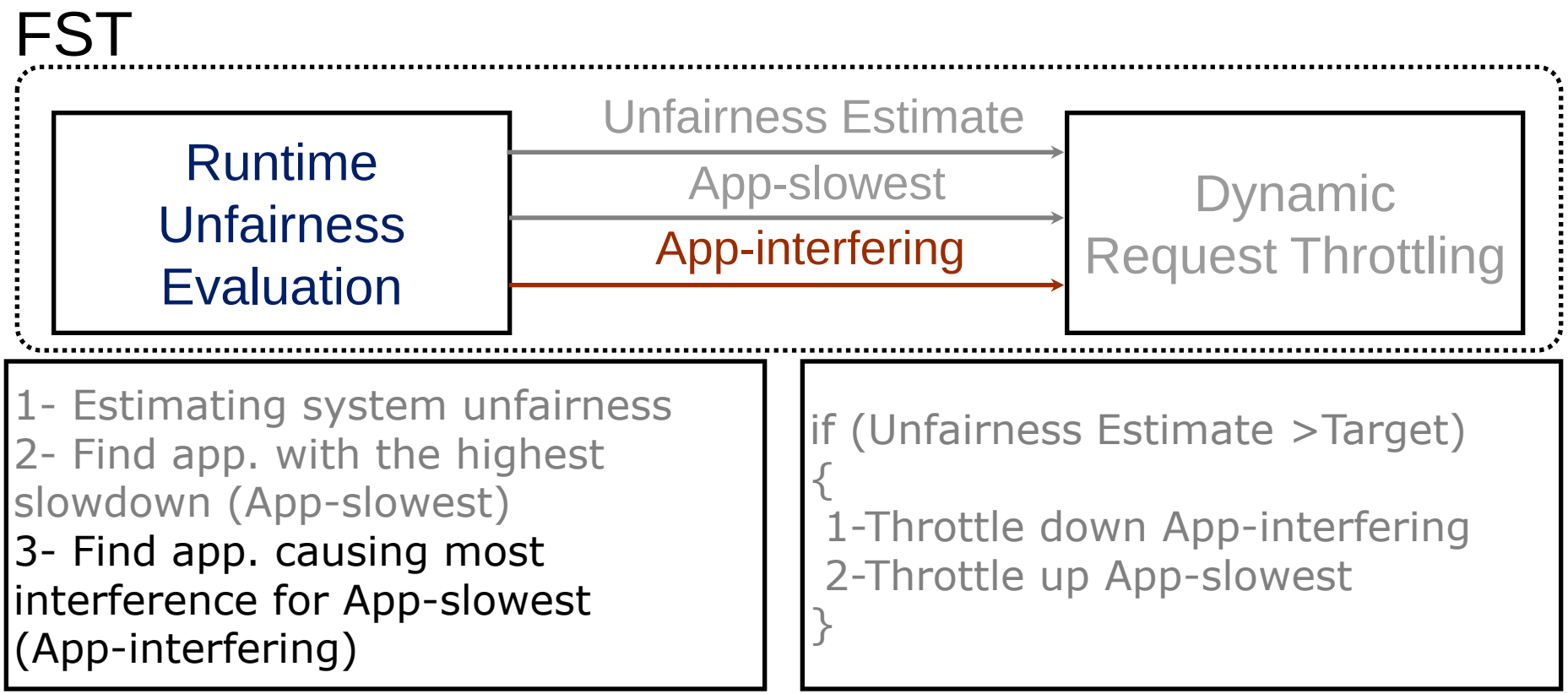
Tracking DRAM Row-Buffer Interference



Tracking Inter-Core Interference

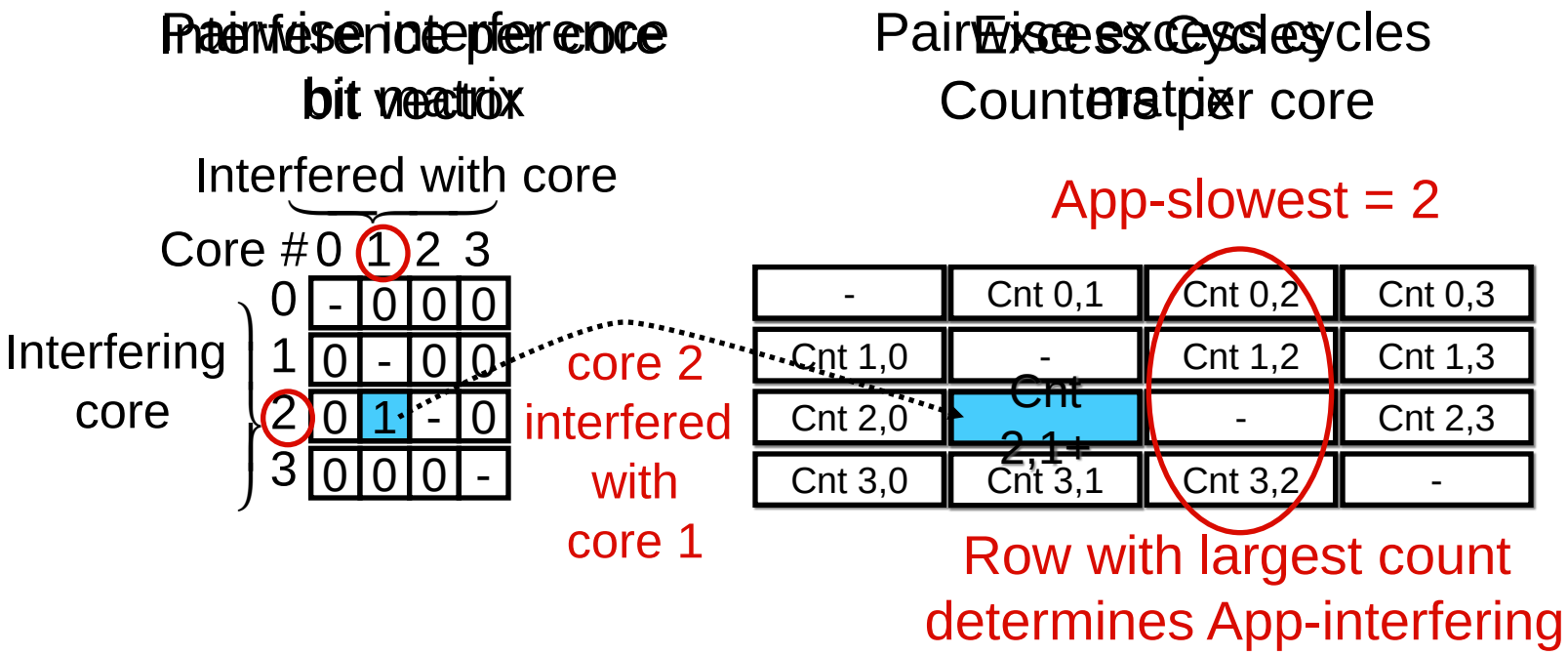


Fairness via Source Throttling (FST)

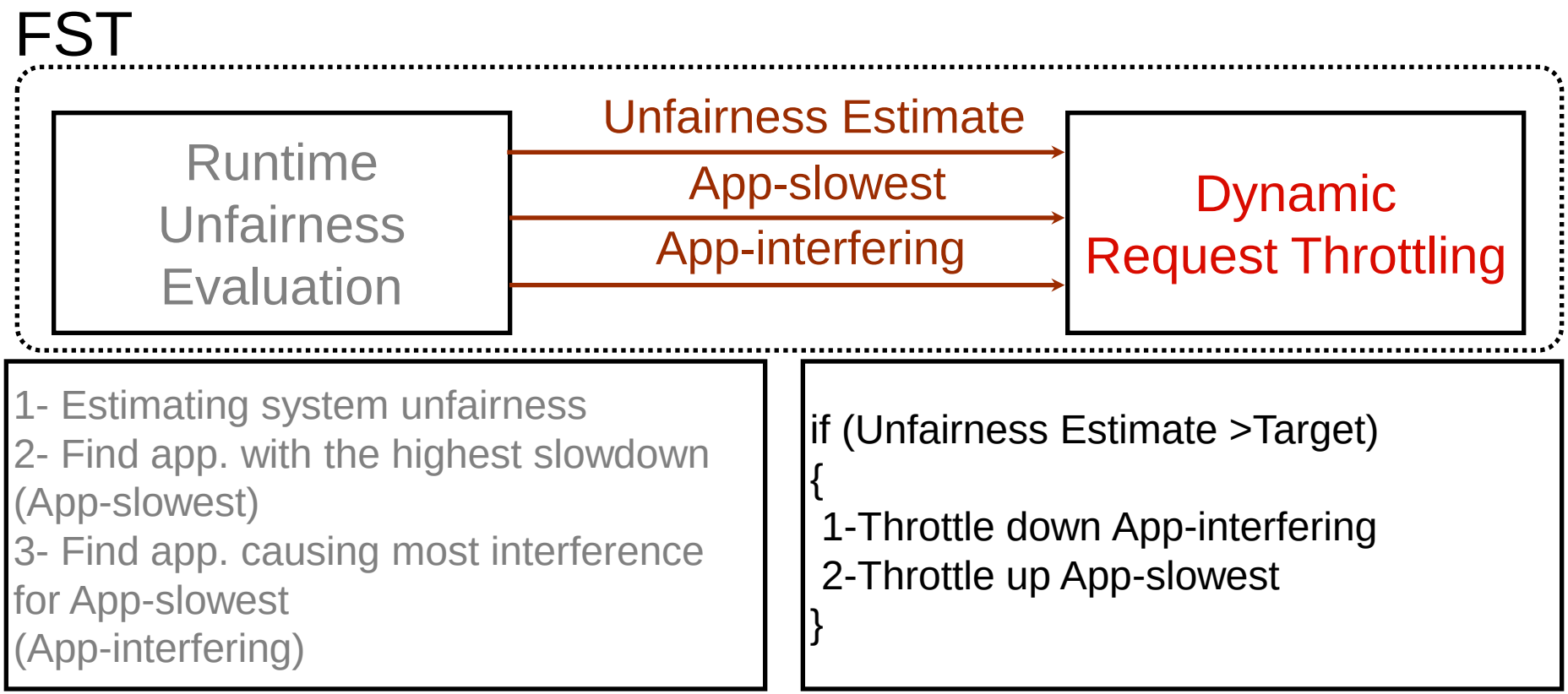


Tracking Inter-Core Interference

- 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

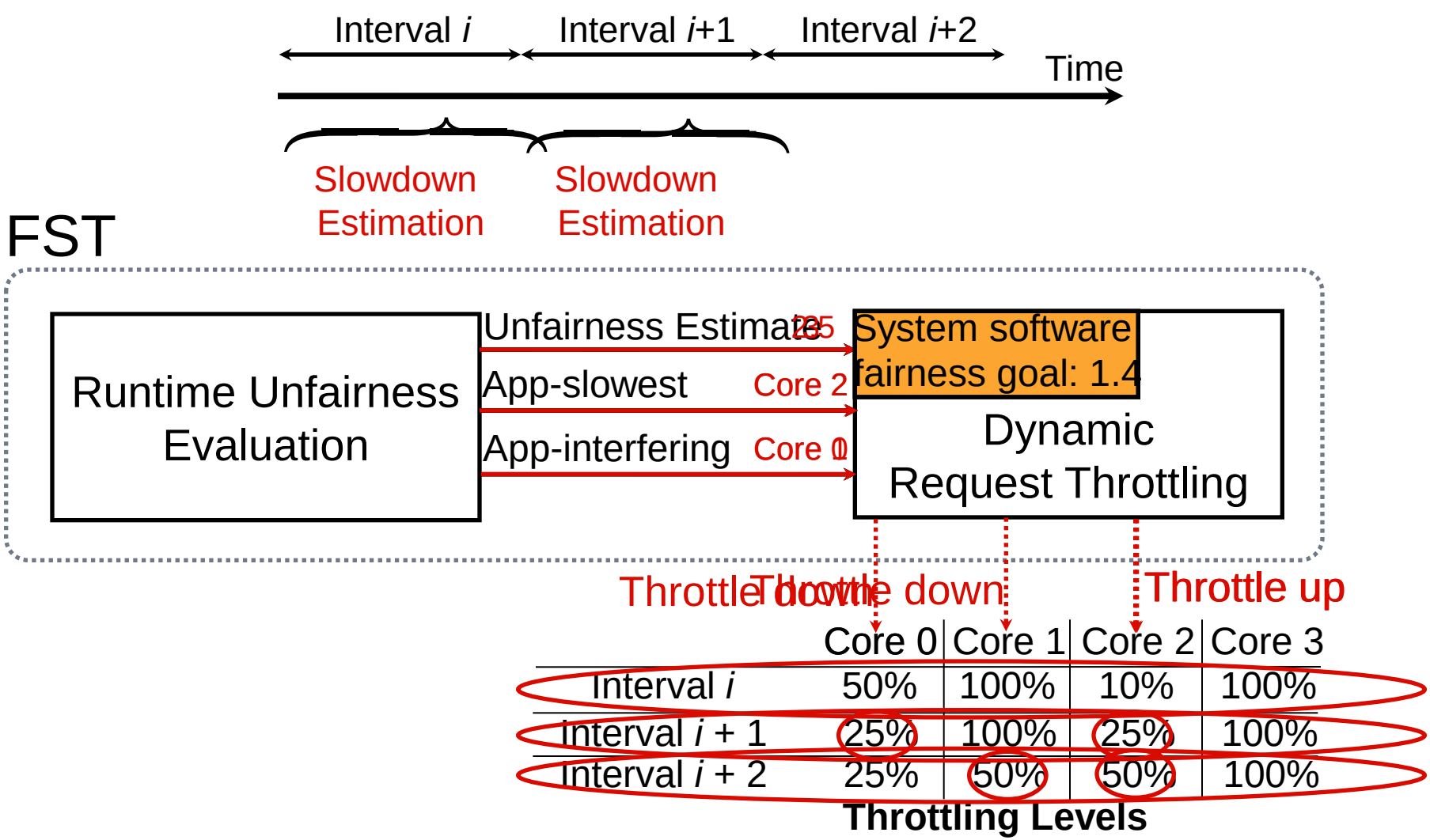
Dynamic Request Throttling

- 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

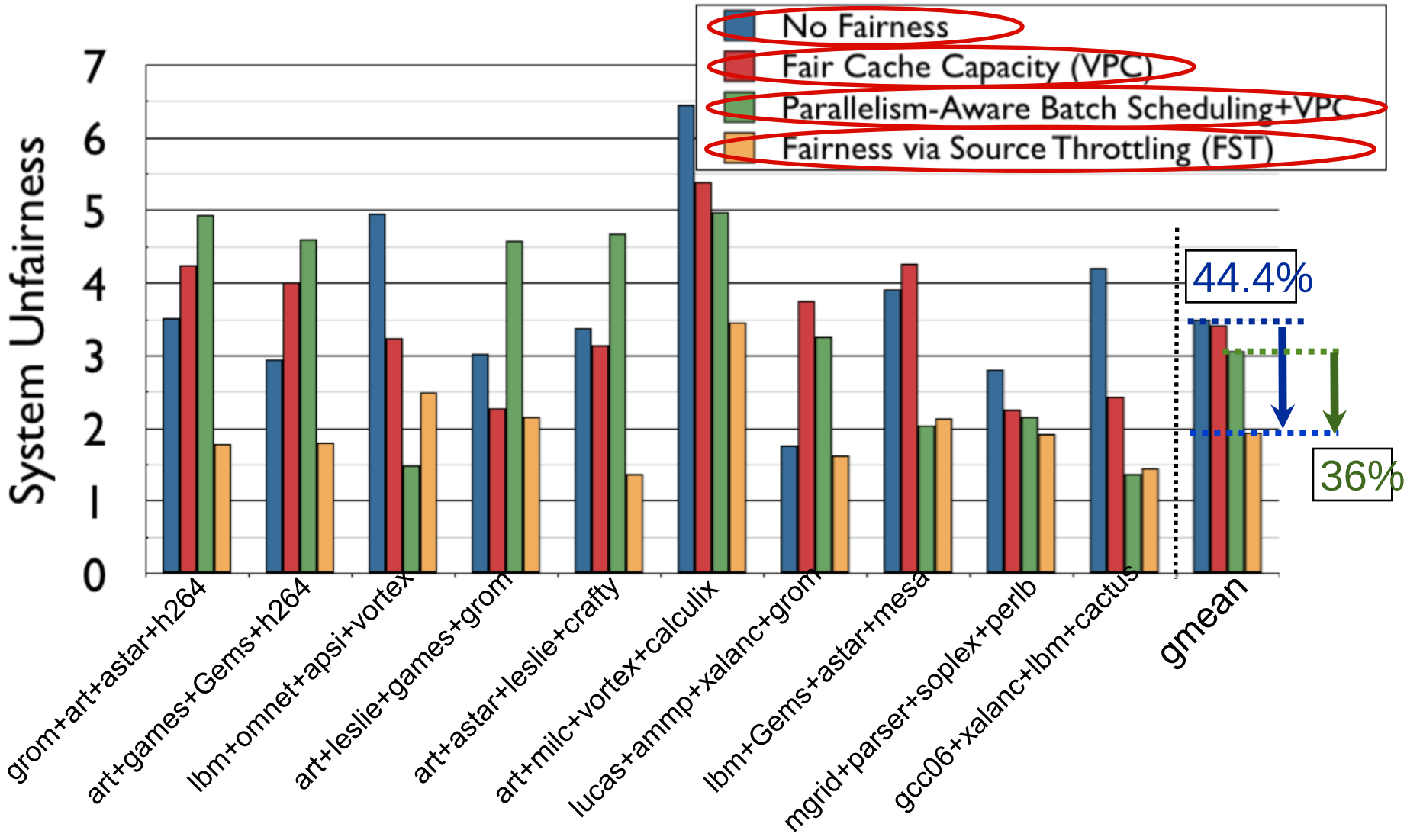
- Total storage cost required for 4 cores is ~12KB

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

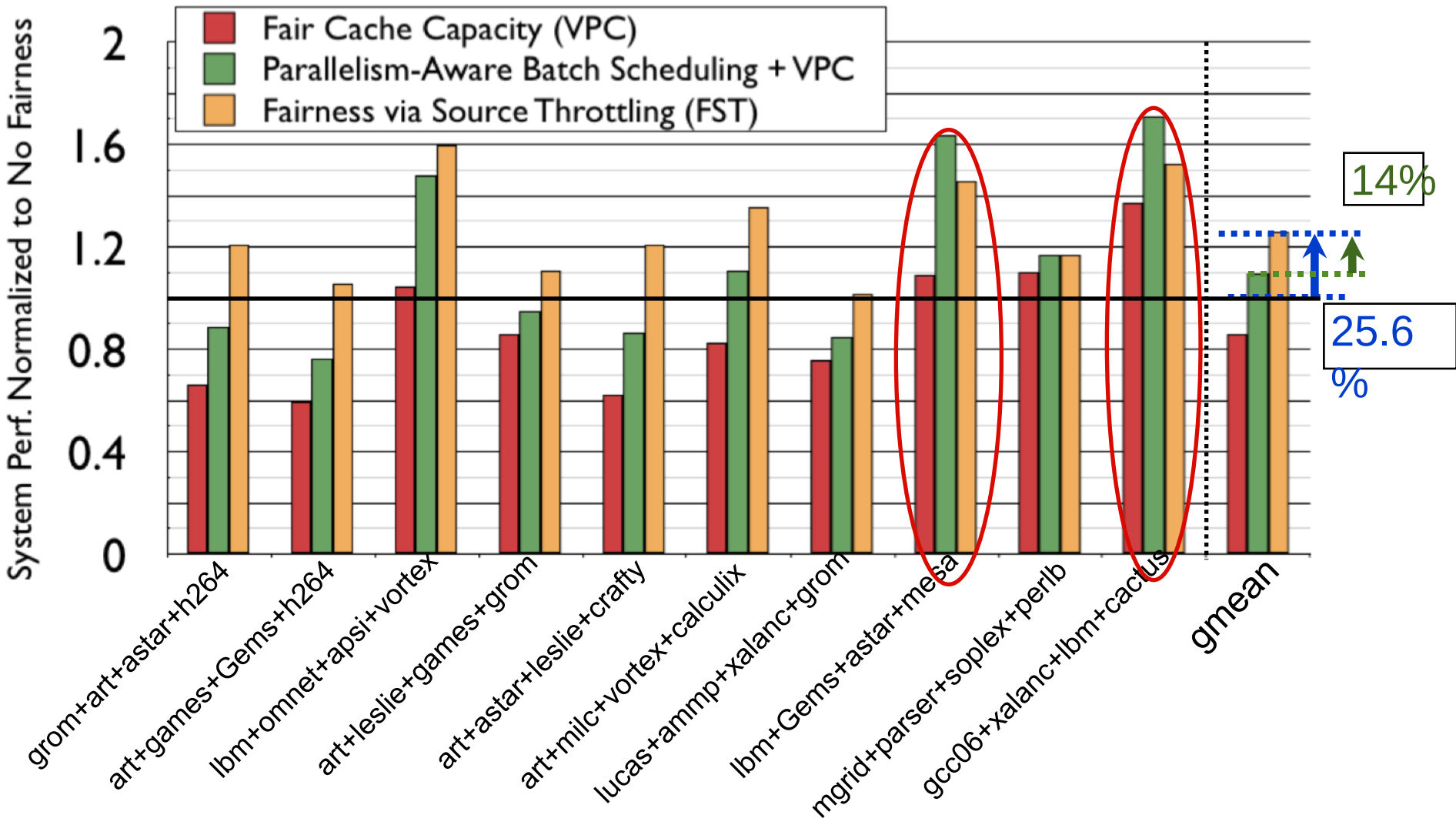
FST Evaluation Methodology

- 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

Designing QoS-Aware Memory Systems: Approaches

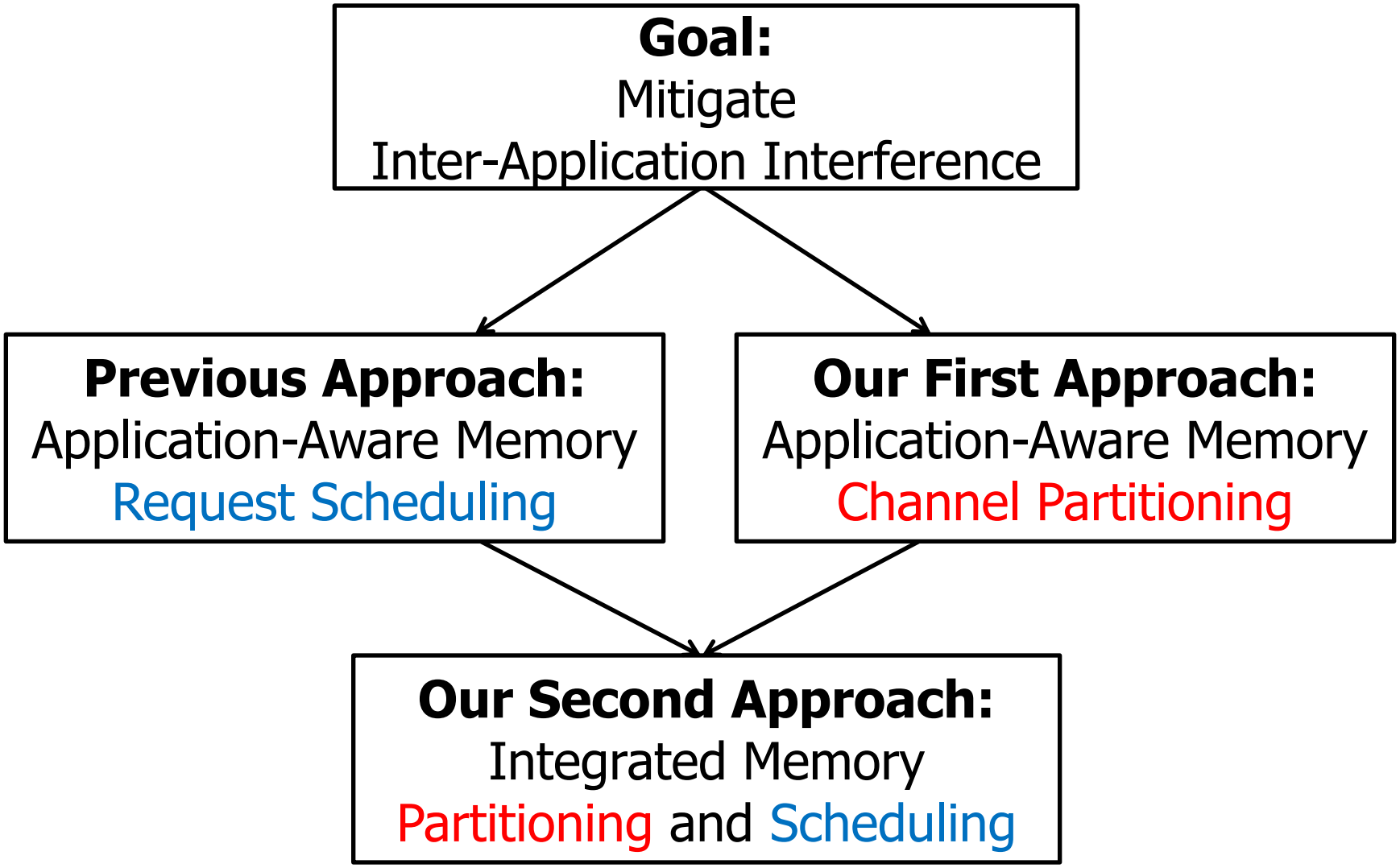
- Smart resources: Design each shared resource to have a configurable interference control/reduction mechanism
  - QoS-aware memory controllers [Mutlu+ MICRO'07] [Moscibroda+, Usenix Security'07] [Mutlu+ ISCA'08, Top Picks'09] [Kim+ HPCA'10] [Kim+ MICRO'10, Top Picks'11] [Ebrahimi+ ISCA'11, MICRO'11] [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
  - 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

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

Overview: Memory Channel Partitioning (MCP)

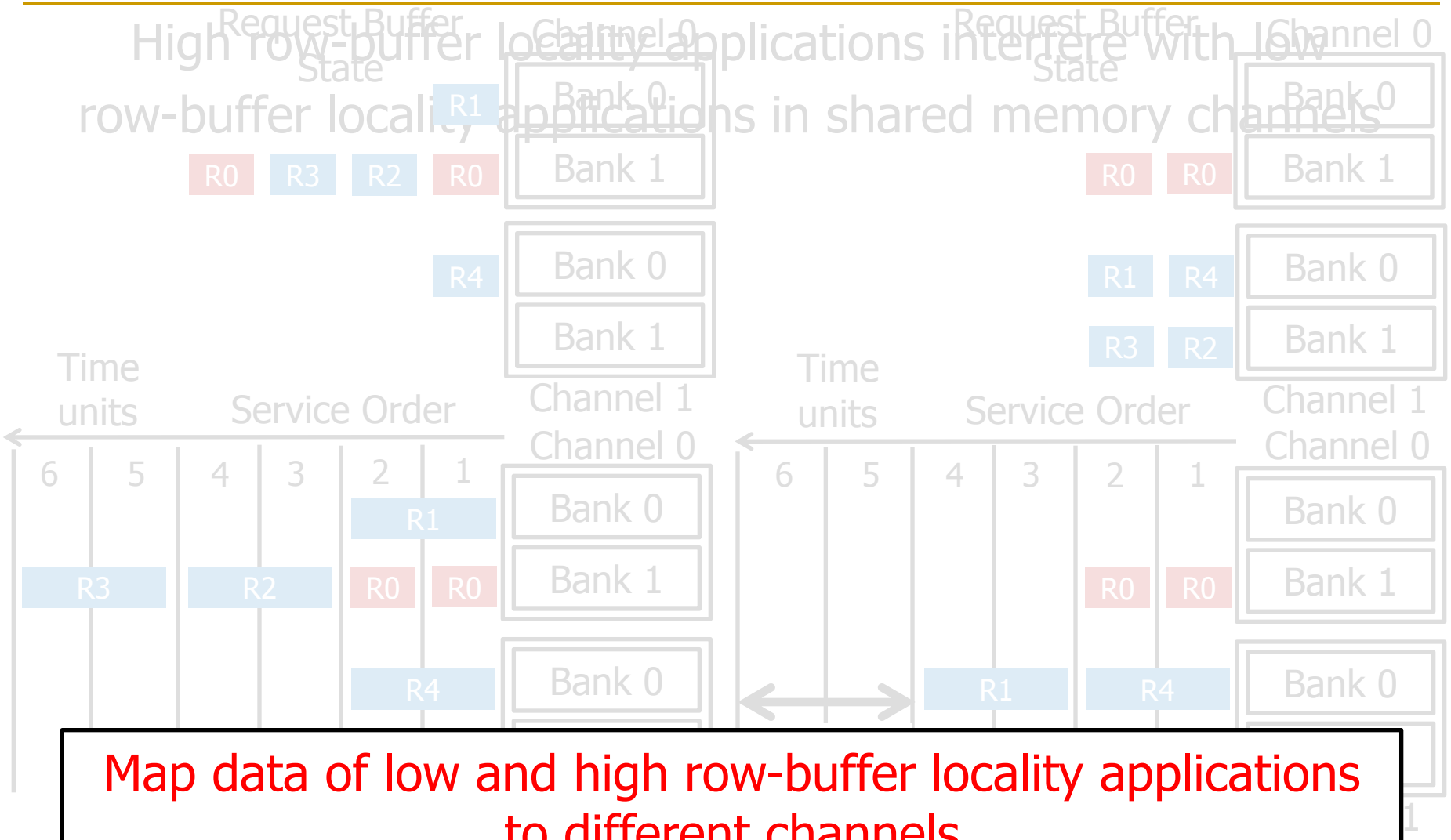
- Goal
  - Eliminate harmful interference between applications
- Basic Idea
  - Map the data of badly-interfering applications to different channels
- Key Principles
  - Separate low and high memory-intensity applications
  - Separate low and high row-buffer locality applications

Key Insight 1: Separate by Memory Intensity

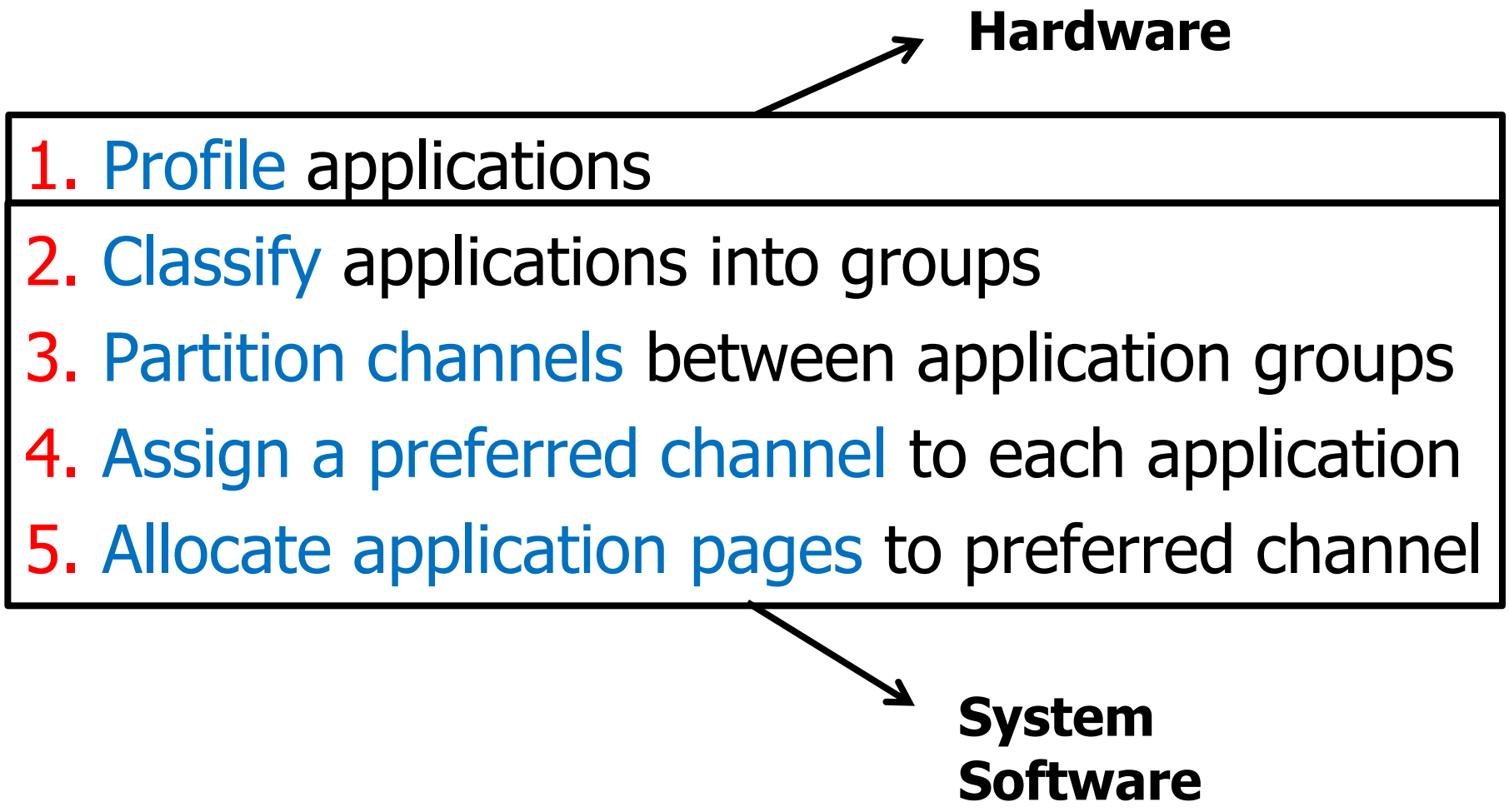
High memory-intensity applications interfere with low memory-intensity applications in shared memory channels

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

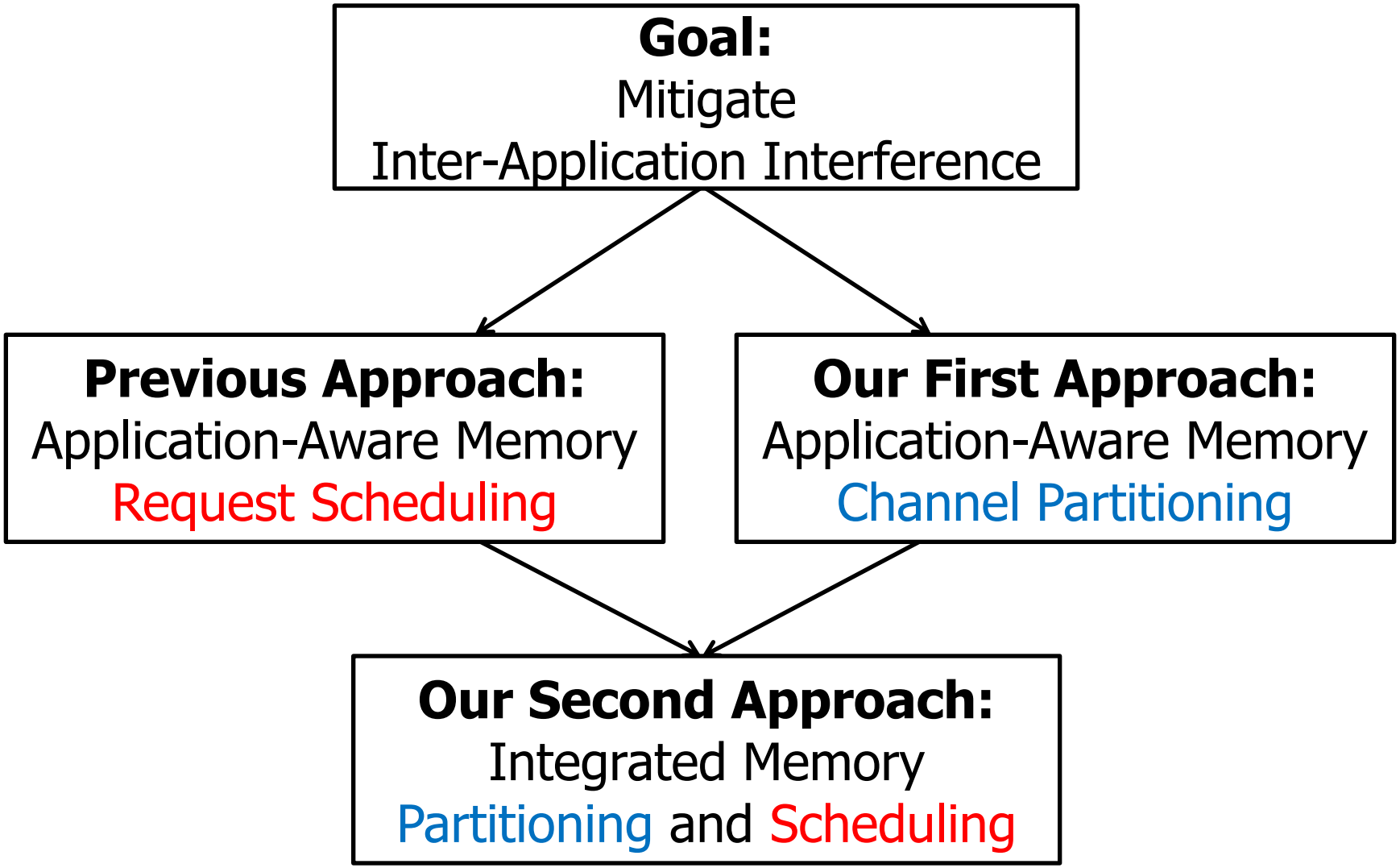
Key Insight 2: Separate by Row-Buffer Locality



Memory Channel Partitioning (MCP) Mechanism



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

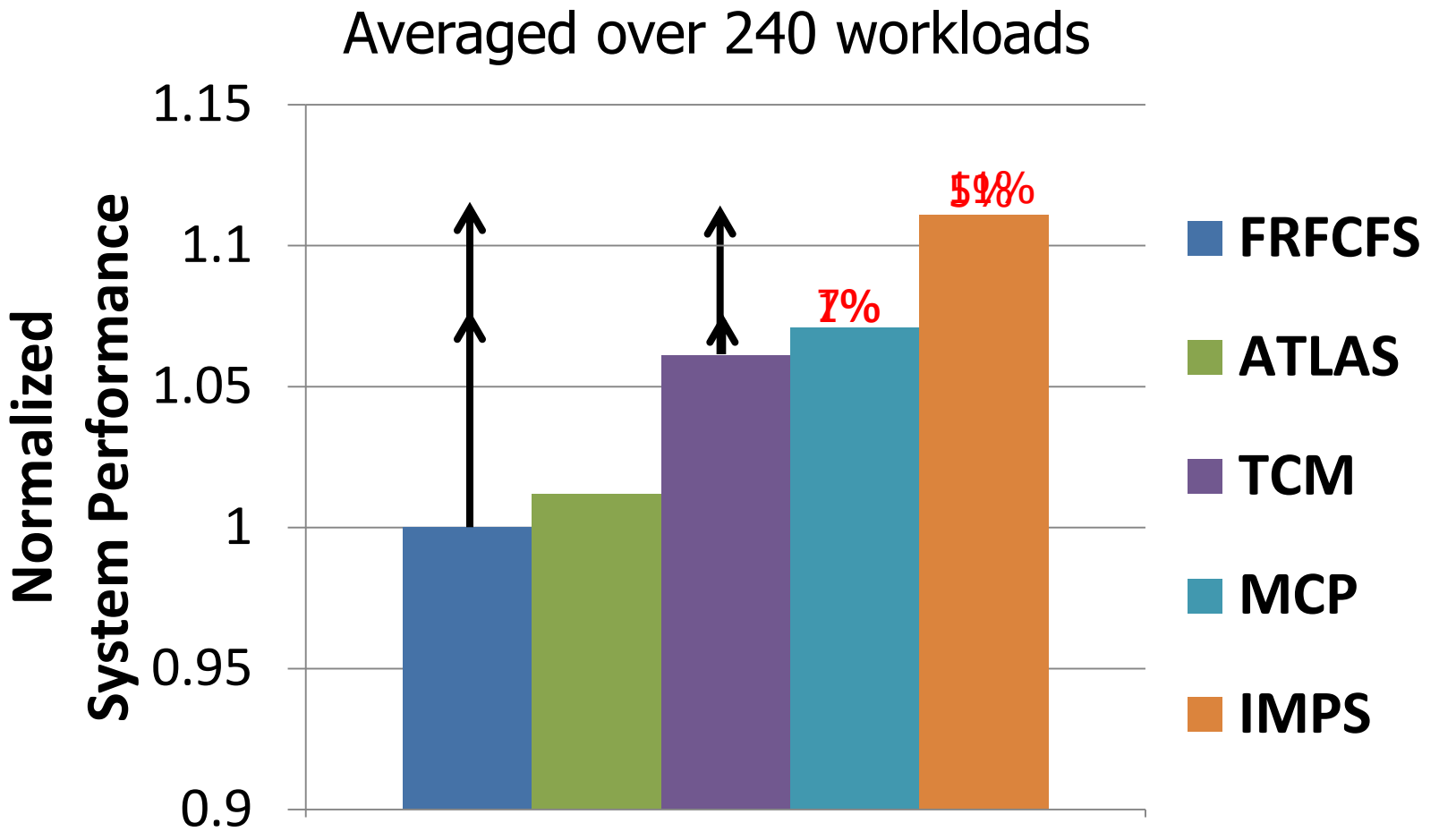
- Use memory channel partitioning to mitigate interference between other applications

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

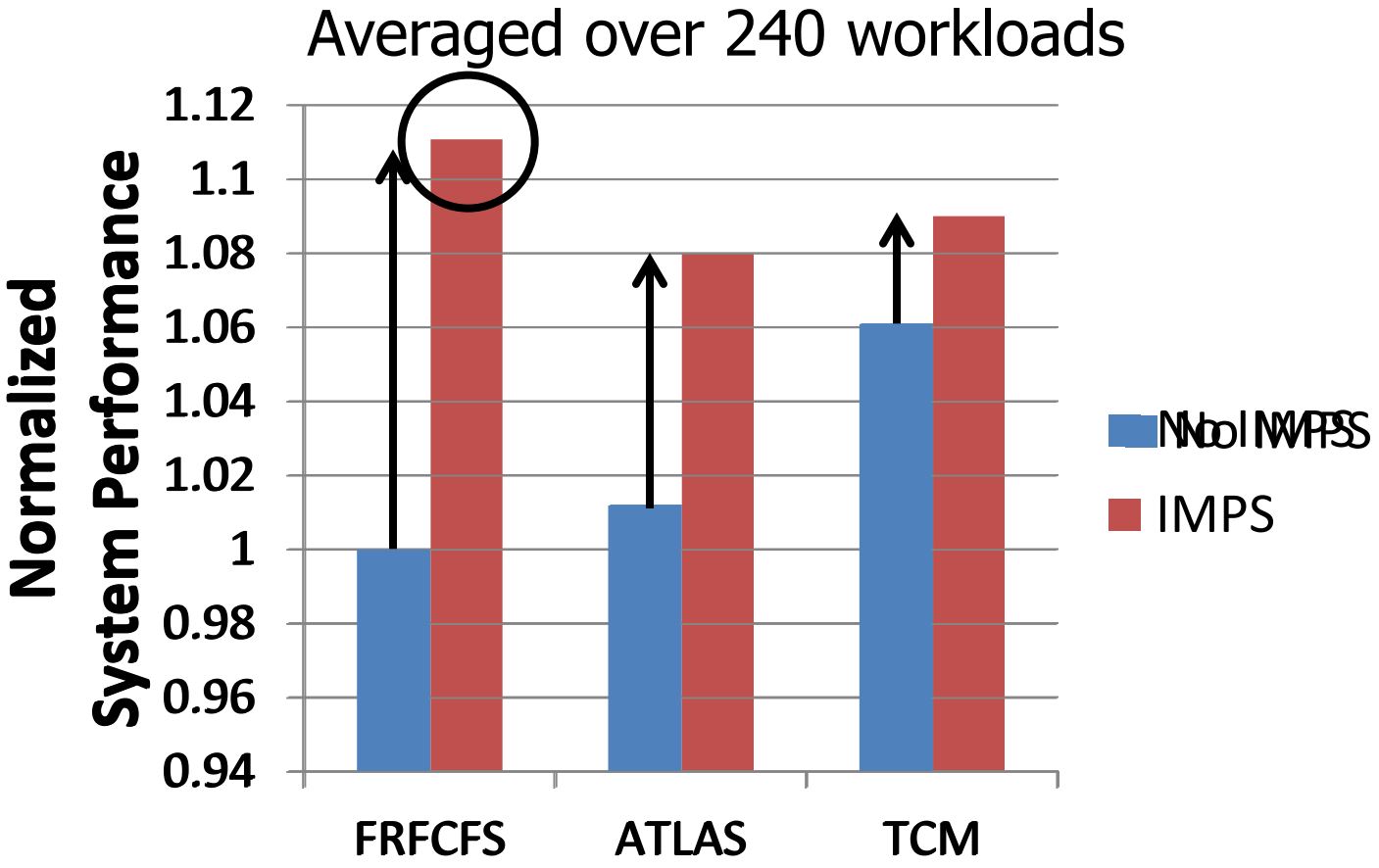
Comparison to Previous Scheduling Policies



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

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; inst ead of per resource
  - Eliminates conflicting decision making between resources

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

Other Ways of Reducing (DRAM) Interference

- DRAM bank/subarray partitioning among threads
- Interference-aware address mapping/remapping
- Core/request throttling: How?
- Interference-aware thread scheduling: How?
- Better/Interference-aware caching
- Interference-aware scheduling in the interconnect
- Randomized address mapping
- DRAM architecture/microarchitecture changes?

- These are general techniques that can be used to improve
  - System throughput
  - QoS/fairness
  - Power/energy consumption?

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

- You can come up with great solutions that will significantly impact computing industry