

QoS-Aware Memory Systems

Onur Mutlu

onur@cmu.edu

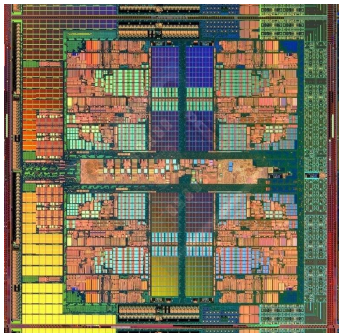
July 8, 2013

INRIA

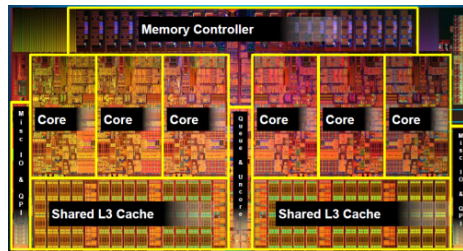
Carnegie Mellon

Trend: Many Cores on Chip

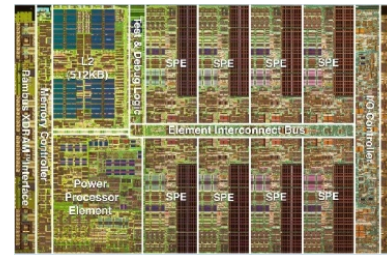
- Simpler and lower power than a single large core
- Large scale parallelism on chip



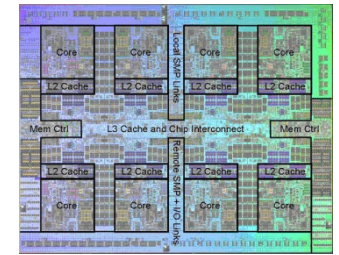
AMD Barcelona
4 cores



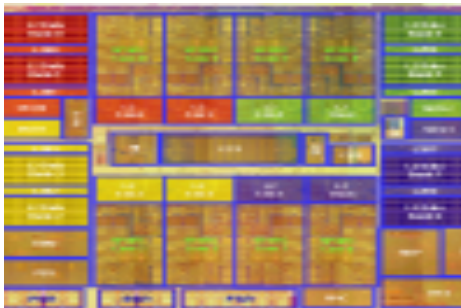
Intel Core i7
8 cores



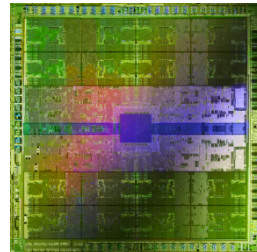
IBM Cell BE
8+1 cores



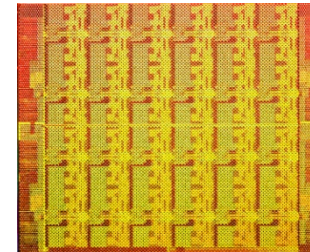
IBM POWER7
8 cores



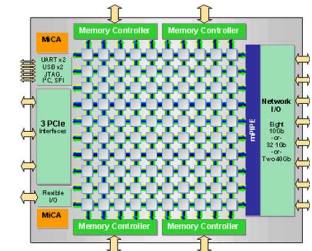
Sun Niagara II
8 cores



Nvidia Fermi
448 "cores"



Intel SCC
48 cores, networked

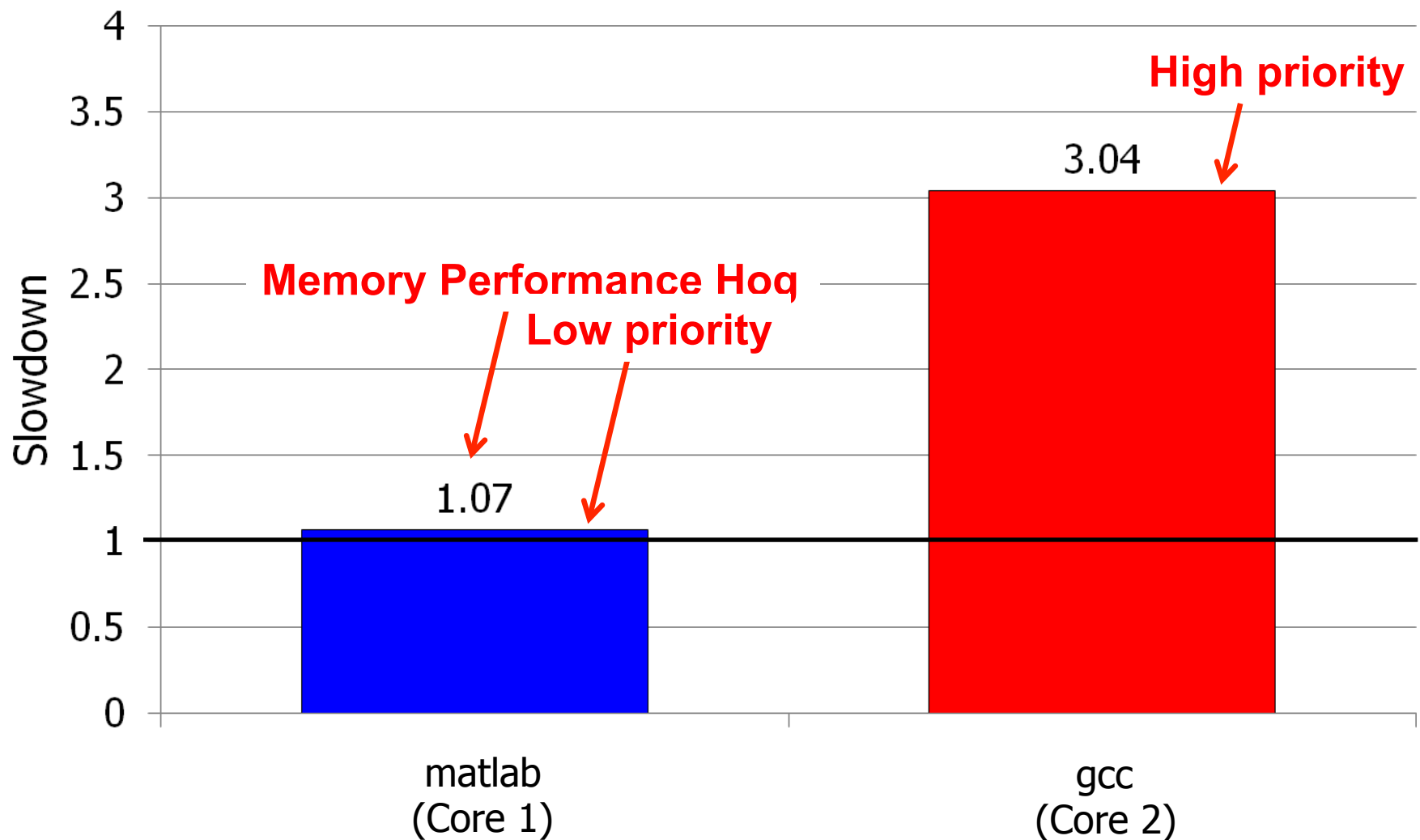


Tiler TILE Gx
100 cores, networked

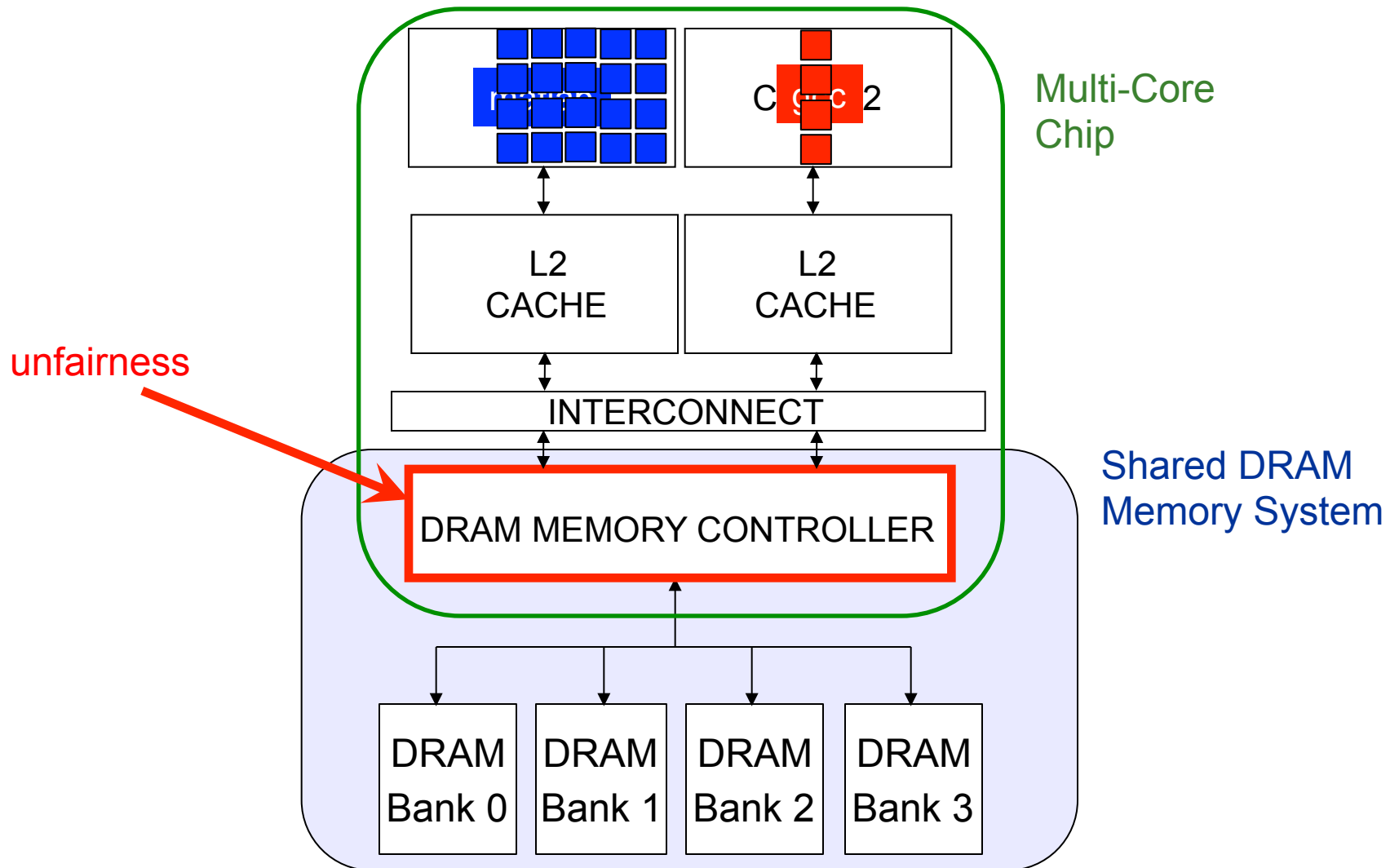
Many Cores on Chip

- What we want:
 - N times the system performance with N times the cores
- What do we get today?

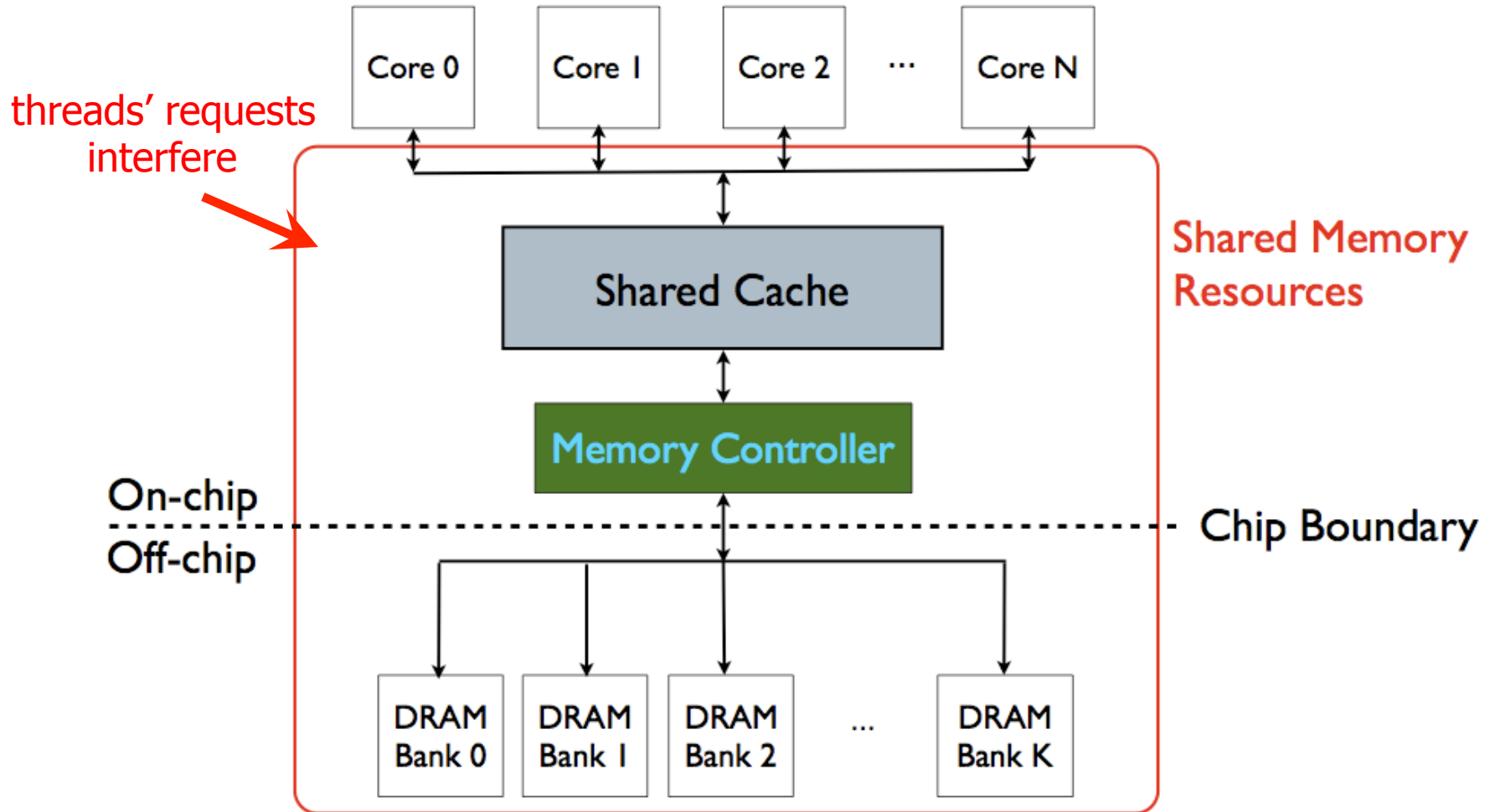
Unfair Slowdowns due to Interference



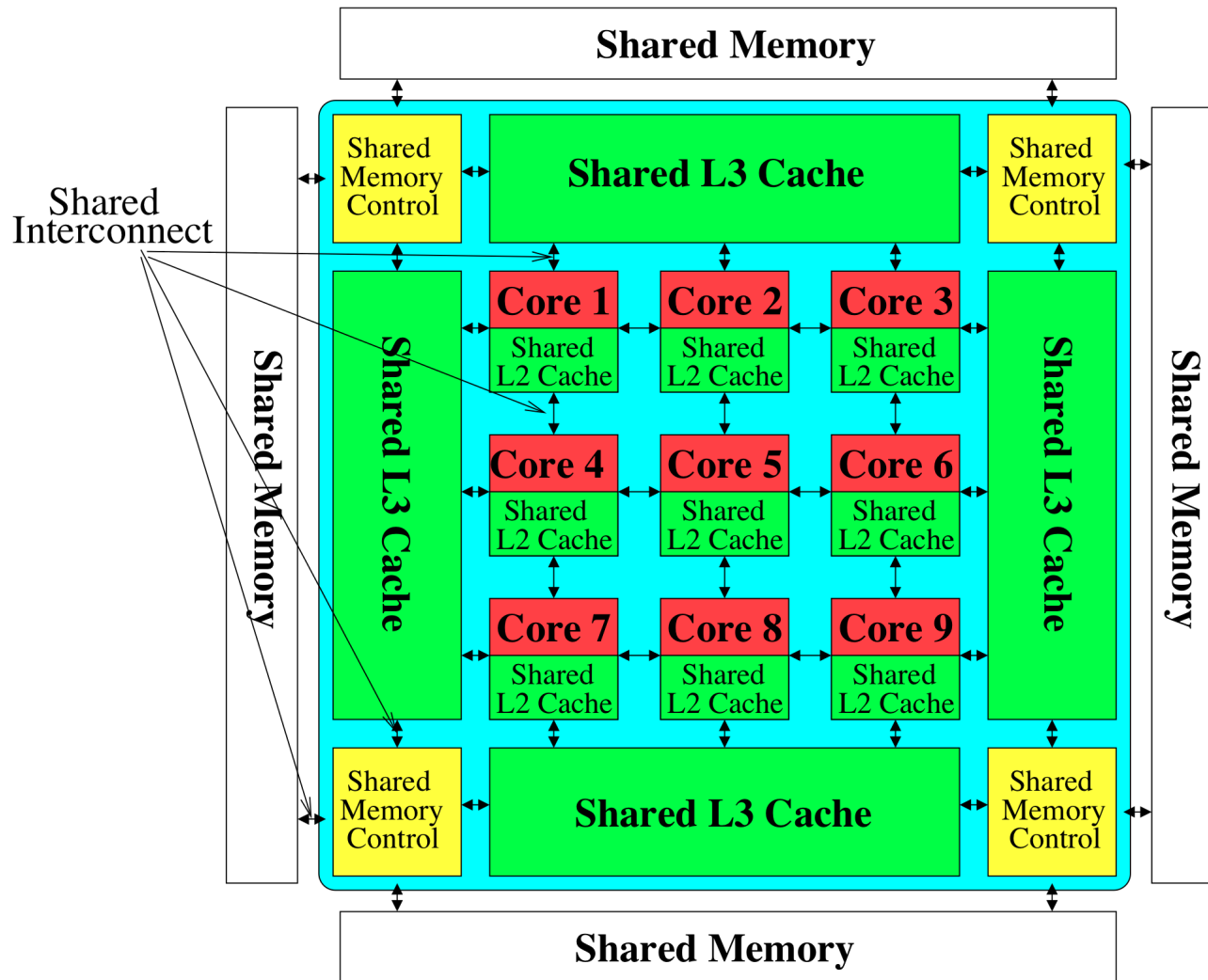
Uncontrolled Interference: An Example



Memory System is the Major Shared Resource



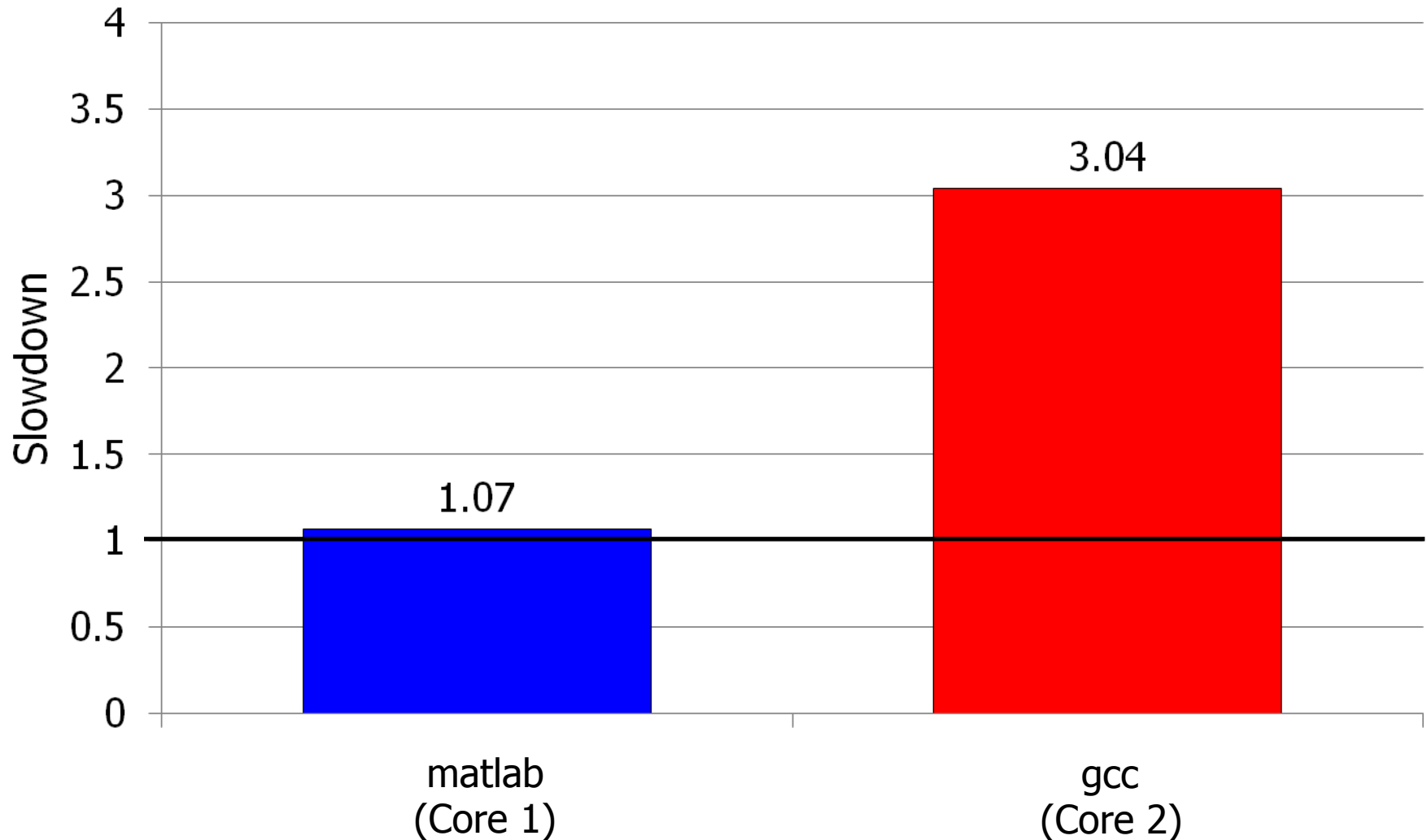
Much More of a Shared Resource in Future



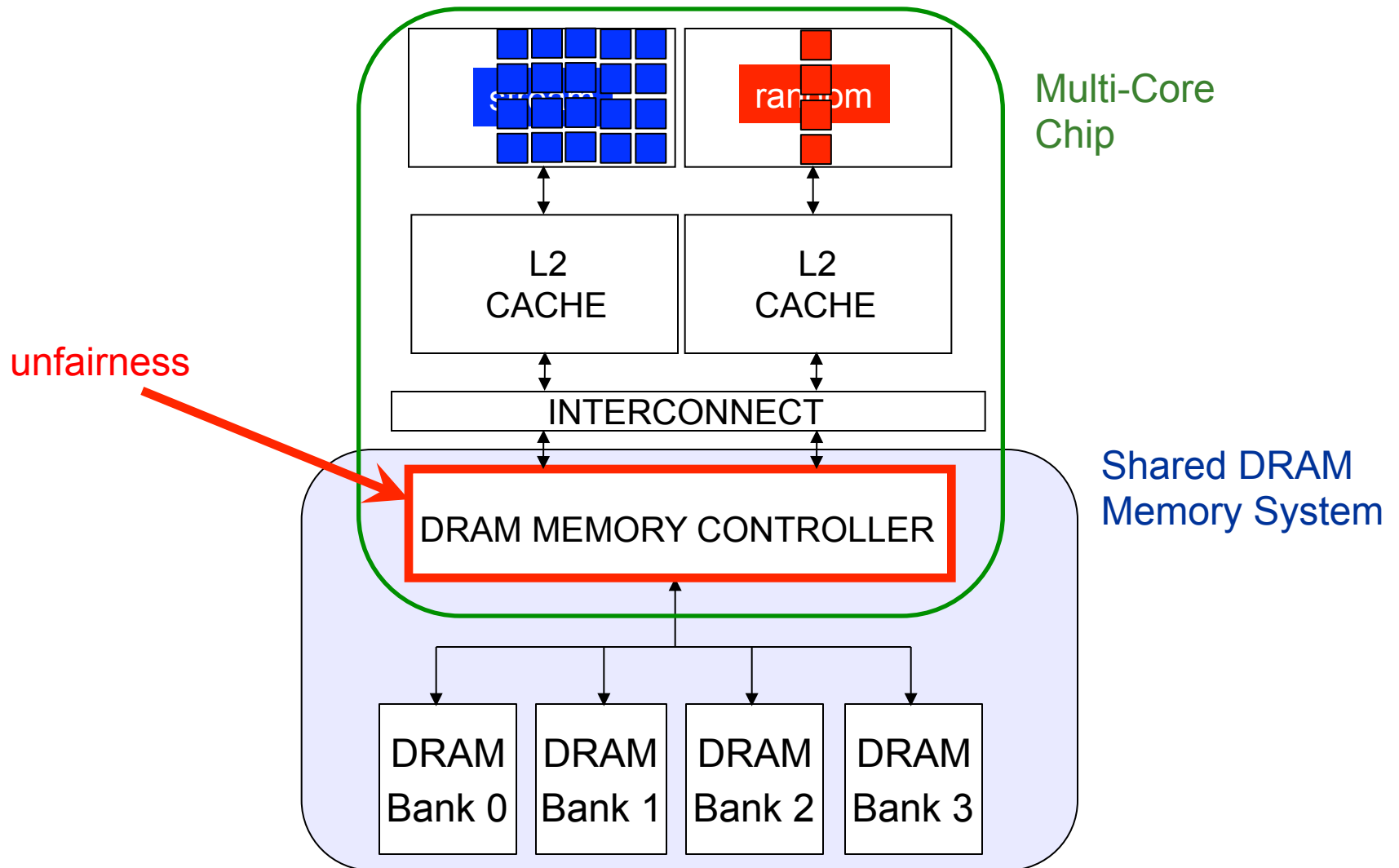
Inter-Thread/ Application Interference

- Problem: Threads share the memory system, but memory system does not distinguish between threads' requests
- Existing memory systems
 - ❑ Free-for-all, shared based on demand
 - ❑ Control algorithms thread-unaware and thread-unfair
 - ❑ Aggressive threads can deny service to others
 - ❑ Do not try to reduce or control inter-thread interference

Unfair Slowdowns due to Interference



Uncontrolled Interference: An Example



A Memory Performance Hog

```
// initialize large arrays A, B  
for (j=0; j<N; j++) {  
    index = j*linesize; streaming  
    A[index] = B[index];  
    ...  
}
```

STREAM

- Sequential memory access
- Very high row buffer locality (96% hit rate)
- Memory intensive

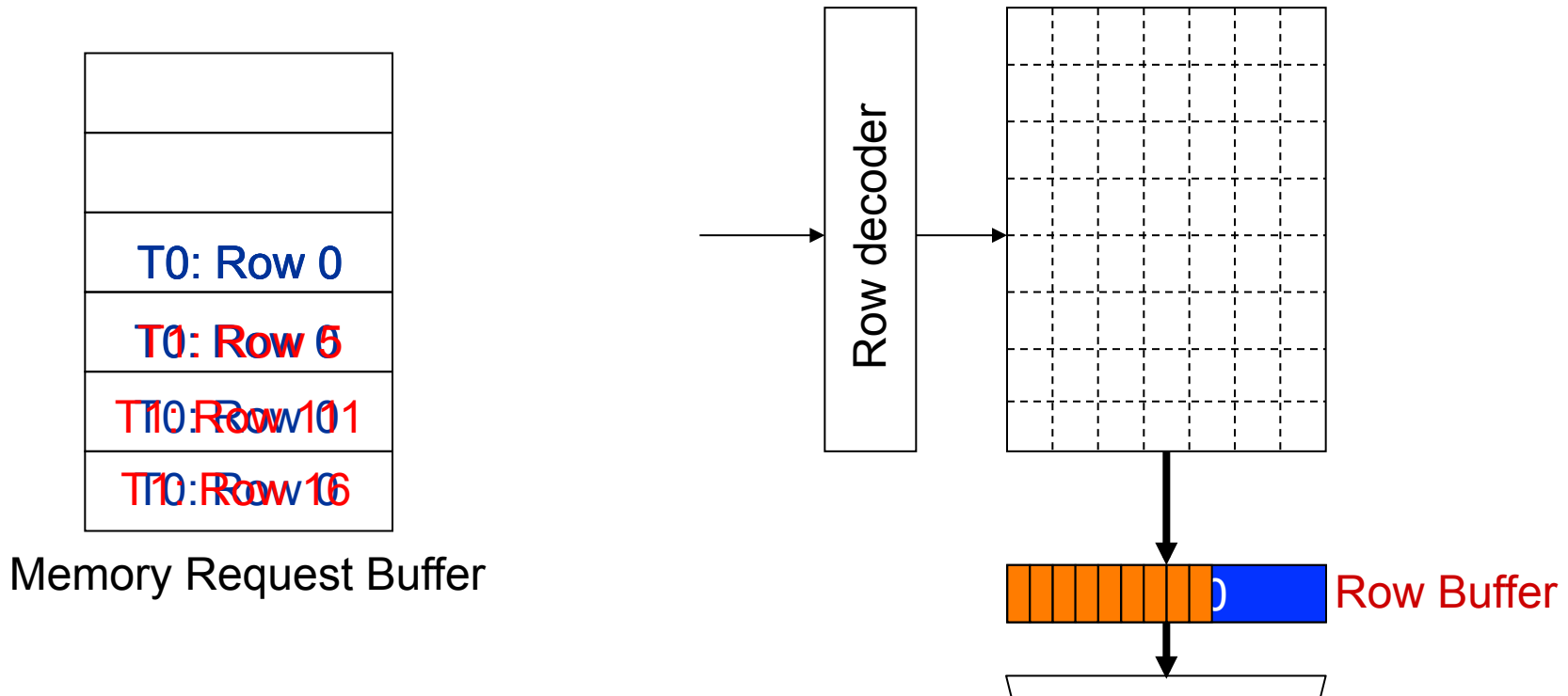
```
// initialize large arrays A, B  
for (j=0; j<N; j++) {  
    index = rand(); random  
    A[index] = B[index];  
    ...  
}
```

RANDOM

- Random memory access
- Very low row buffer locality (3% hit rate)
- Similarly memory intensive

Moscibroda and Mutlu, “[Memory Performance Attacks](#),” USENIX Security 2007.

What Does the Memory Hog Do?



Row size: 8KB, cache block size: 64B
128 (8KB/64B) requests of T0 serviced before T1

Moscibroda and Mutlu, “[Memory Performance Attacks](#),” USENIX Security 2007.

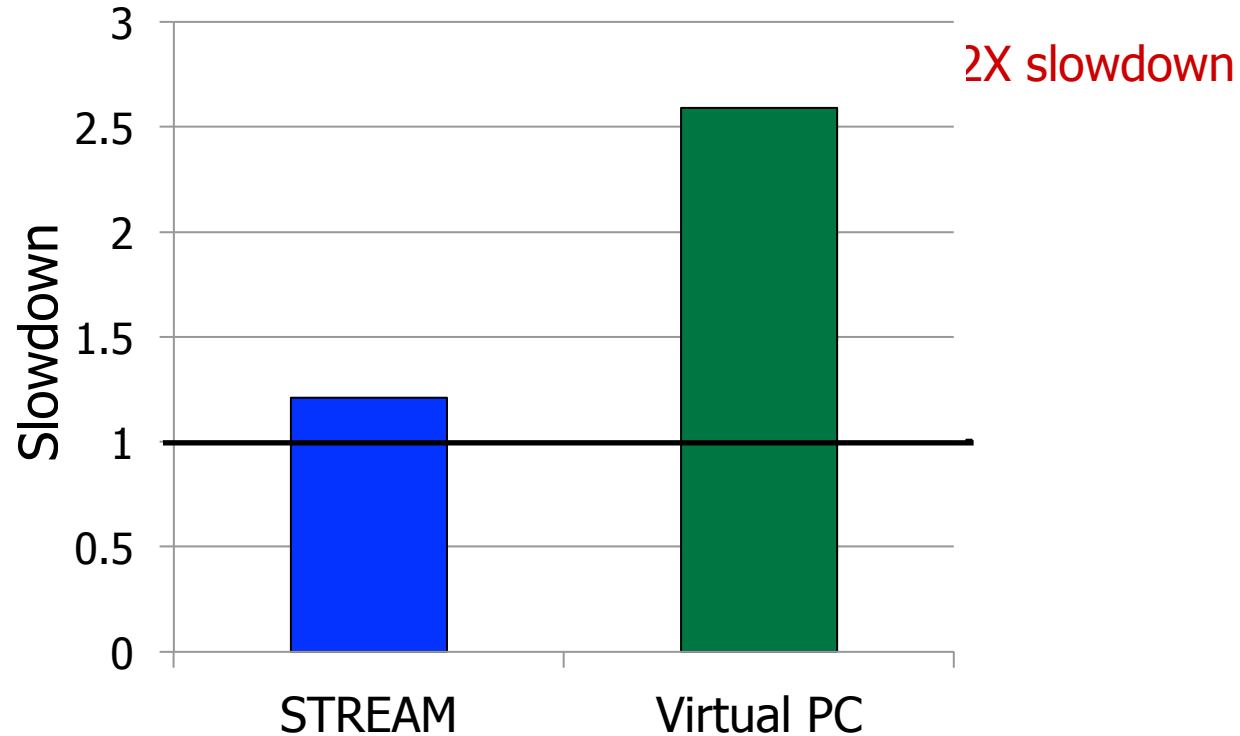
DRAM Controllers

- A row-conflict memory access takes significantly longer than a row-hit access
- Current controllers take advantage of the row buffer
- Commonly used scheduling policy (FR-FCFS) [Rixner 2000]*
 - (1) Row-hit first: Service row-hit memory accesses first
 - (2) Oldest-first: Then service older accesses first
- This scheduling policy aims to maximize DRAM throughput
 - But, it is unfair when multiple threads share the DRAM system

*Rixner et al., “Memory Access Scheduling,” ISCA 2000.

*Zuravleff and Robinson, “Controller for a synchronous DRAM ...,” US Patent 5,630,096, May 1997.

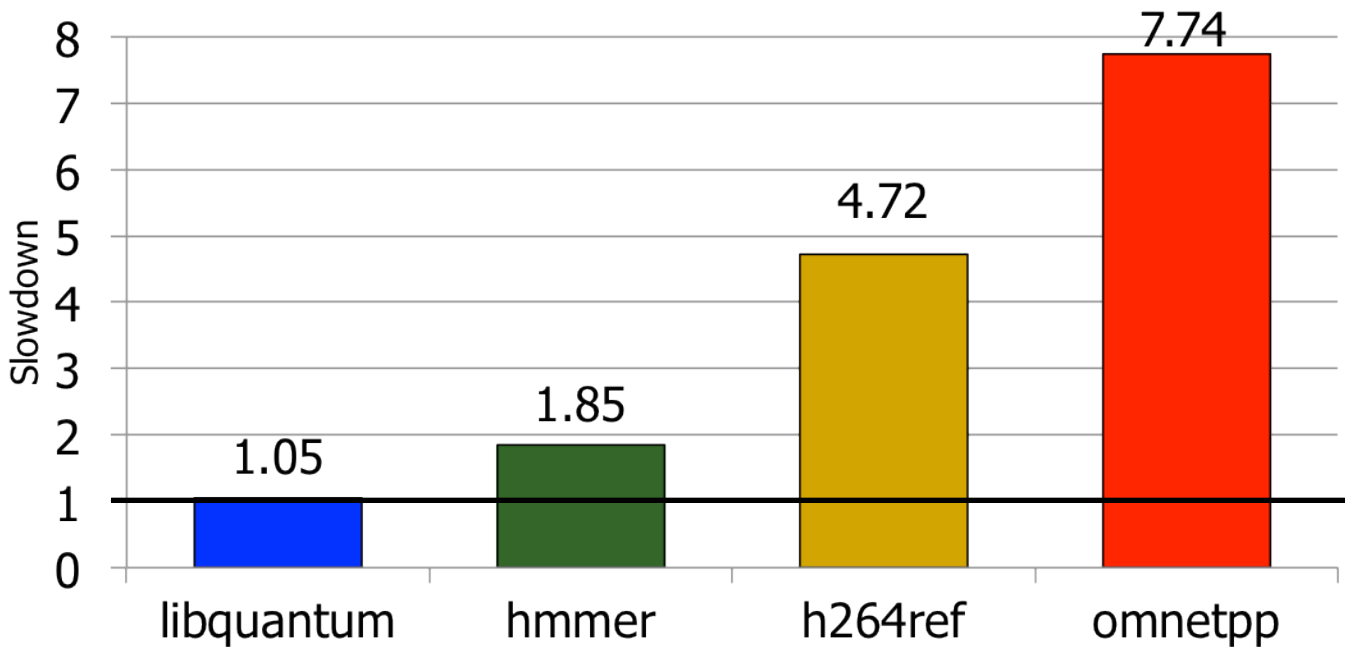
Effect of the Memory Performance Hog



Results on Intel Pentium D running Windows XP
(Similar results for Intel Core Duo and AMD Turion, and on Fedora Linux)

Moscibroda and Mutlu, “[Memory Performance Attacks](#),” USENIX Security 2007.

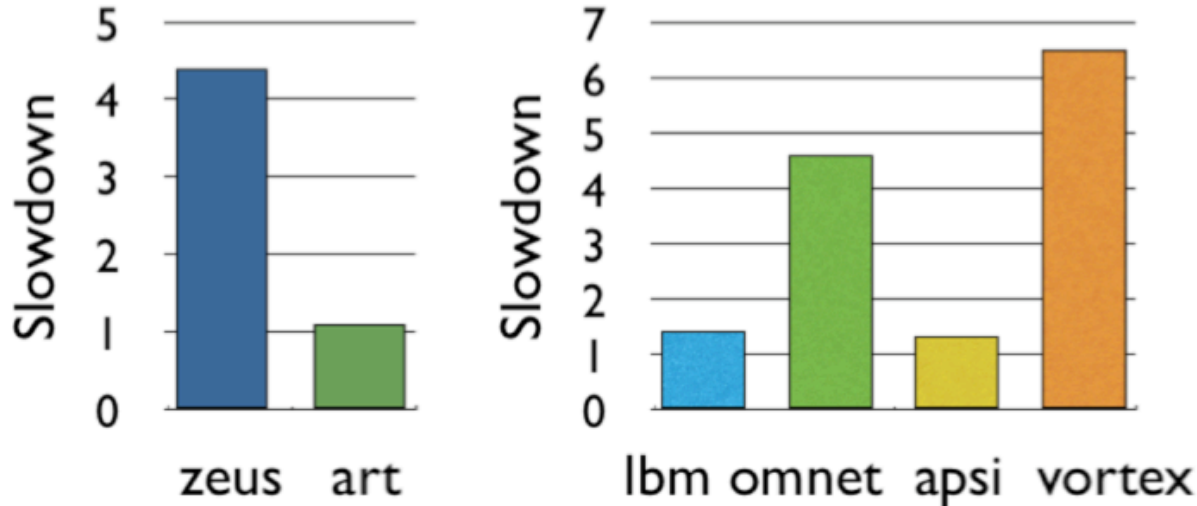
Greater Problem with More Cores



- Vulnerable to denial of service (DoS)
- Unable to enforce priorities or SLAs
- Low system performance

Uncontrollable, unpredictable system

Greater Problem with More Cores



- Vulnerable to denial of service (DoS) [Usenix Security'07]
- Unable to enforce priorities or SLAs [MICRO'07,'10,'11, ISCA'08'11'12, ASPLOS'10]
- Low system performance [IEEE Micro Top Picks '09,'11a,'11b,'12]

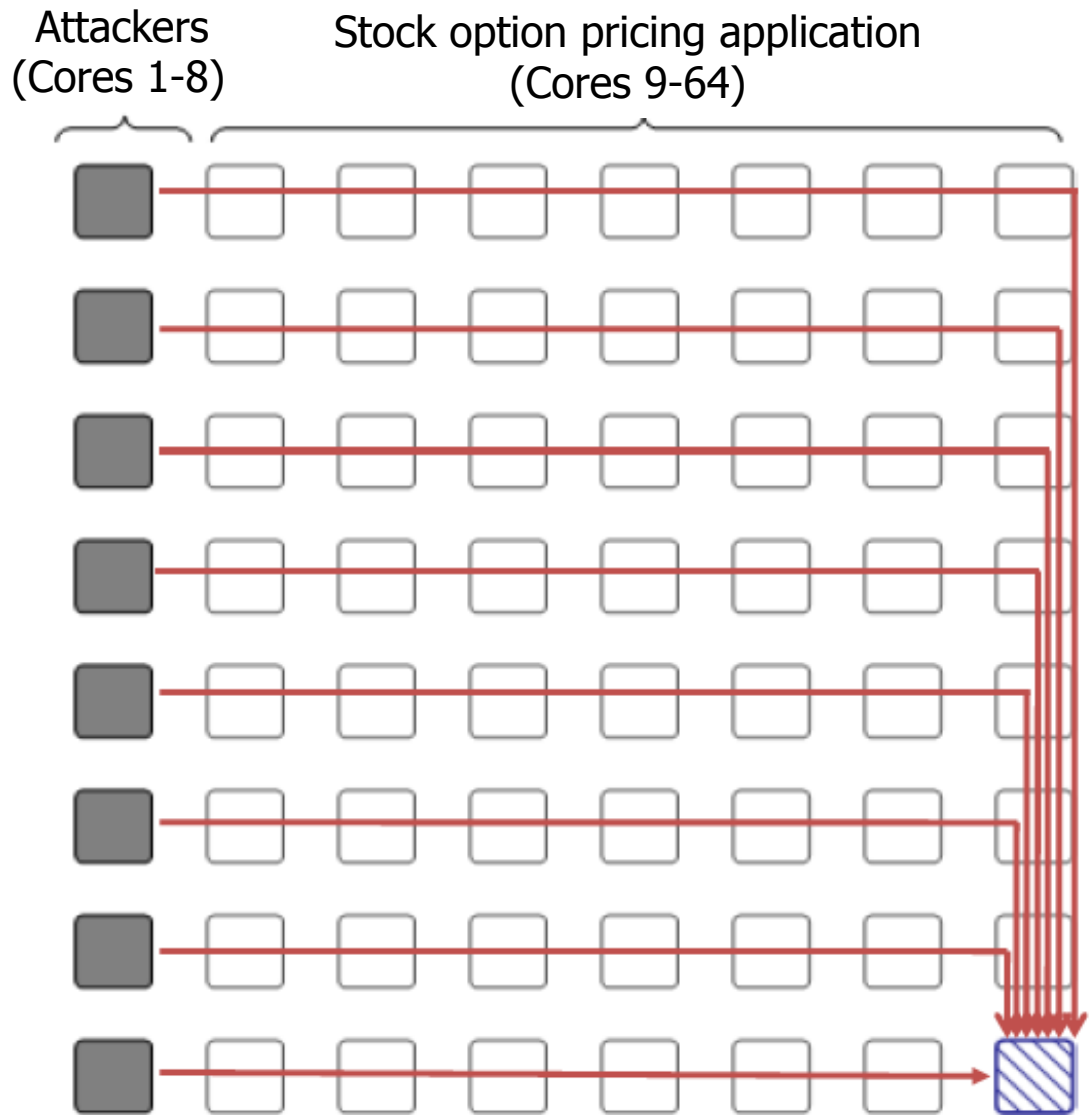
Uncontrollable, unpredictable system

Distributed DoS in Networked Multi-Core Systems

Cores connected via
packet-switched
routers on chip

~5000X latency increase

Grot, Hestness, Keckler, Mutlu,
“Preemptive virtual clock: A Flexible,
Efficient, and Cost-effective QOS
Scheme for Networks-on-Chip,”
MICRO 2009.



How Do We Solve The Problem?

- Inter-thread interference is uncontrolled in all memory resources
 - Memory controller
 - Interconnect
 - Caches
- We need to control it
 - i.e., design an interference-aware (QoS-aware) memory system

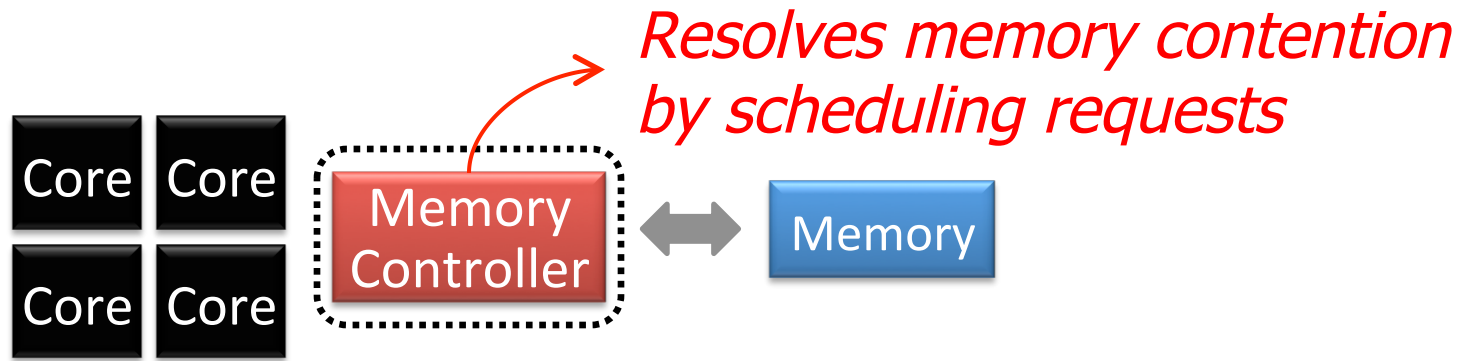
QoS-Aware Memory Systems: Challenges

- How do we **reduce inter-thread interference**?
 - Improve system performance and core utilization
 - Reduce request serialization and core starvation
- How do we **control inter-thread interference**?
 - Provide mechanisms to enable system software to enforce QoS policies
 - While providing high system performance
- How do we **make the memory system configurable/flexible**?
 - Enable flexible mechanisms that can achieve many goals
 - Provide fairness or throughput when needed
 - Satisfy performance guarantees when needed

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]

QoS-Aware Memory Scheduling



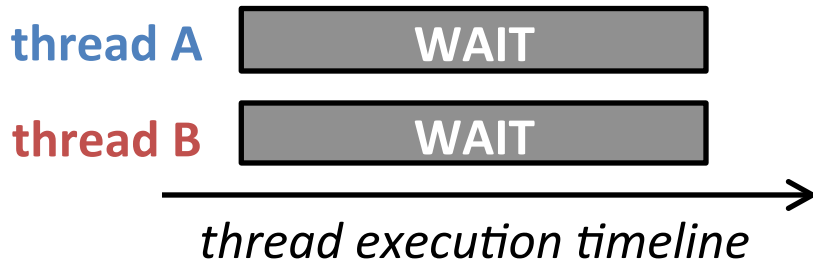
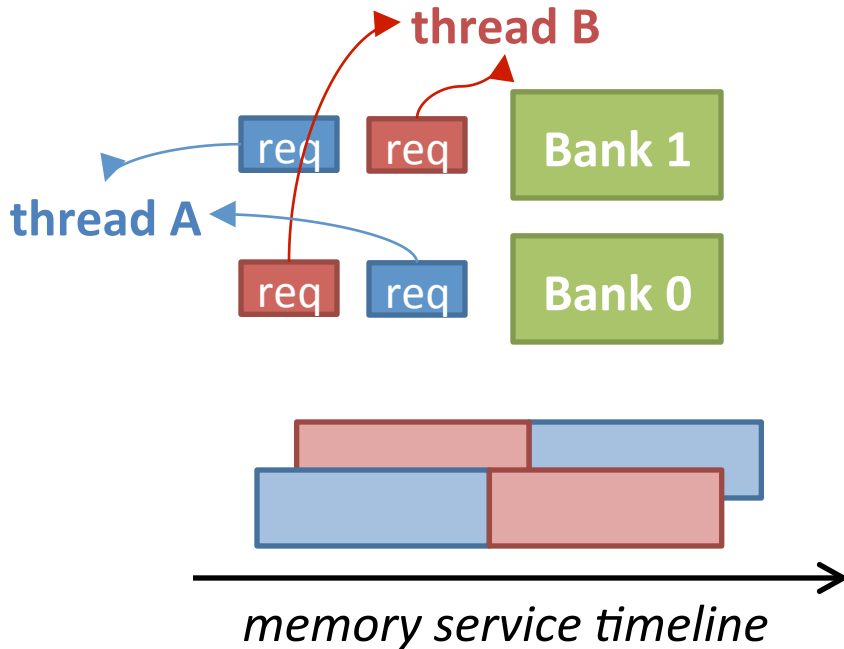
- How to schedule requests to provide
 - ❑ High system performance
 - ❑ High fairness to applications
 - ❑ Configurability to system software
- Memory controller needs to be aware of threads

QoS-Aware Memory Scheduling: Evolution

QoS-Aware Memory Scheduling: Evolution

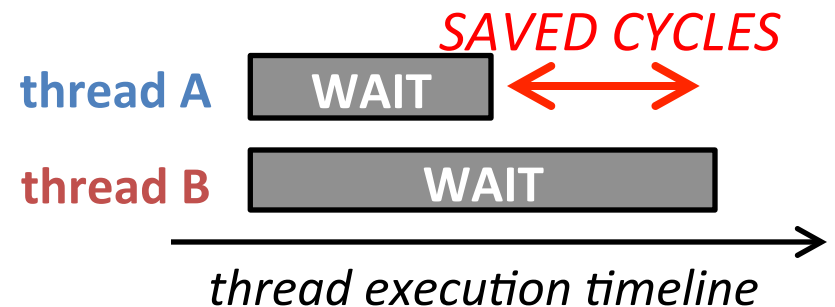
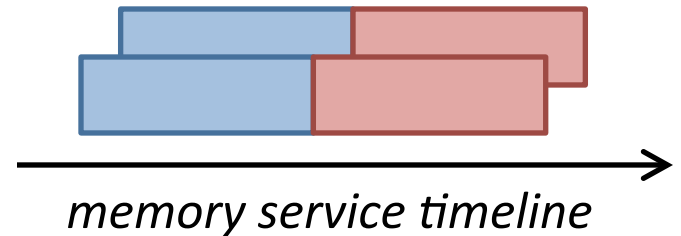
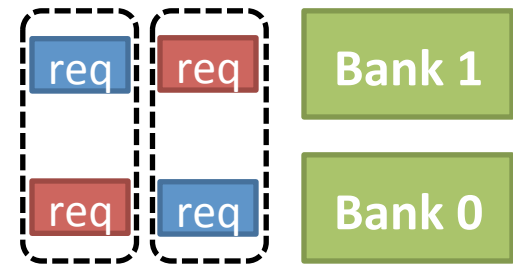
- **Stall-time fair memory scheduling** [Mutlu+ MICRO'07]
 - Idea: Estimate and balance thread slowdowns
 - Takeaway: **Proportional thread progress improves performance, especially when threads are "heavy"** (memory intensive)
- **Parallelism-aware batch scheduling** [Mutlu+ ISCA'08, Top Picks'09]
 - Idea: Rank threads and service in rank order (to preserve bank parallelism); batch requests to prevent starvation
- **ATLAS memory scheduler** [Kim+ HPCA'10]

Within-Thread Bank Parallelism



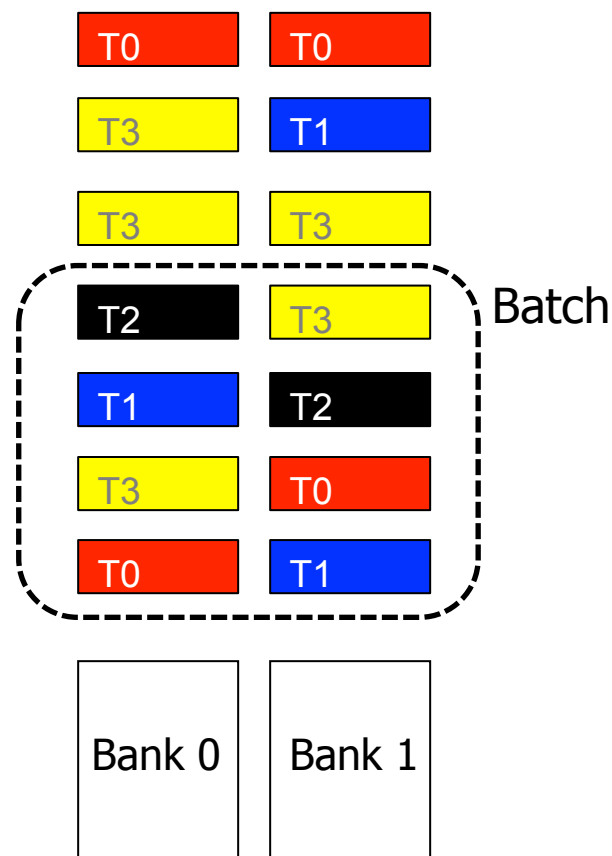
Key Idea:

rank ↑ thread A
thread B



Parallelism-Aware Batch Scheduling [ISCA'08]

- Principle 1: Schedule requests from a thread back to back
 - ❑ Preserves each thread's bank parallelism
 - ❑ But, this can cause starvation...
- Principle 2: Group a fixed number of oldest requests from each thread into a "batch"
 - ❑ Service the batch before all other requests
 - ❑ Form a new batch when the current batch is done
 - ❑ Eliminates starvation, provides fairness



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
- 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 combining application-aware channel partitioning and memory scheduling provides better performance than either

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
- **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

- **MISE** [Subramanian+ HPCA'13]
 - Idea: Estimate the performance of a thread by estimating its change in memory request service rate when run alone vs. shared → use this simple model to estimate slowdown to design a scheduling policy that provides predictable performance or fairness
 - Takeaway: Request service rate of a thread is a good proxy for its performance; alone request service rate can be estimated by giving high priority to the thread in memory scheduling for a while

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
- **DRAM-Aware last-level cache policies** [Lee+ HPS Tech Report'10] [Lee+ HPS Tech Report'10]
 - Idea: Design cache eviction and replacement policies such that they proactively exploit the state of the memory controller and DRAM (e.g., proactively evict data from the cache that hit in open rows)
 - Takeaway: Coordination of last-level cache and DRAM policies improves performance and fairness

Stall-Time Fair Memory Scheduling

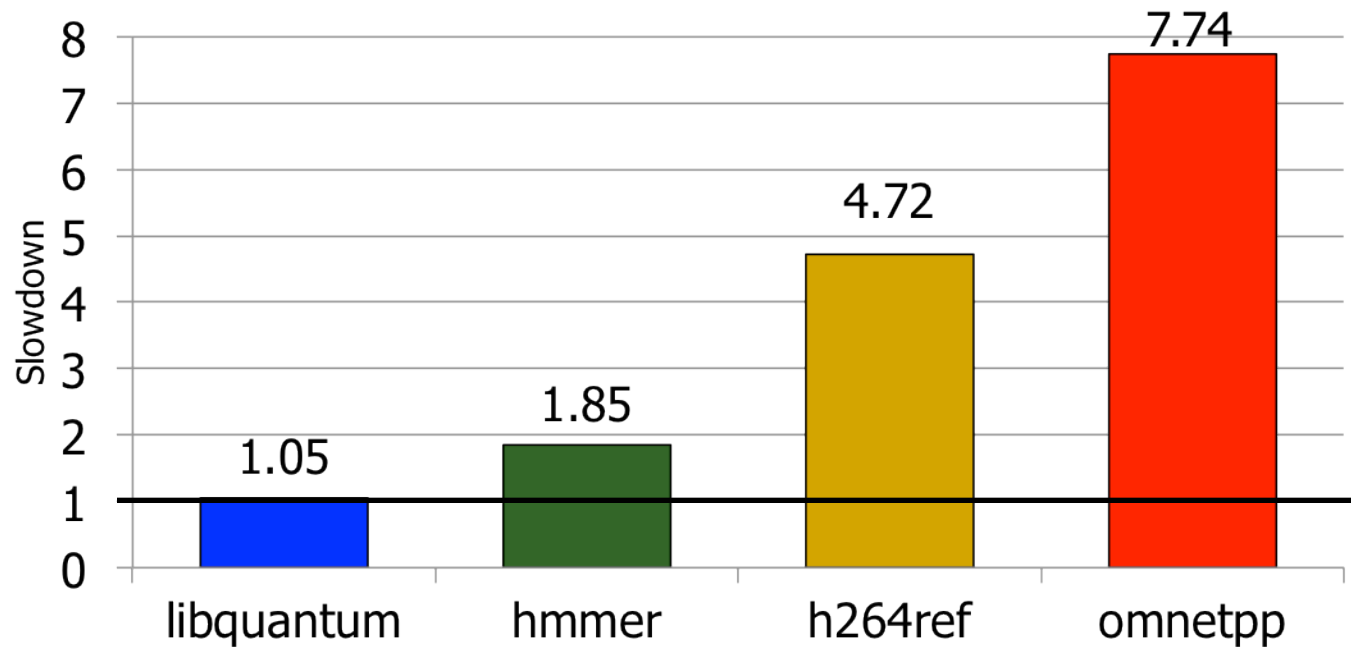
Onur Mutlu and Thomas Moscibroda,

"Stall-Time Fair Memory Access Scheduling for Chip Multiprocessors"

40th International Symposium on Microarchitecture (**MICRO**),

pages 146-158, Chicago, IL, December 2007. Slides (ppt)

The Problem: Unfairness



- Vulnerable to denial of service (DoS) [Usenix Security'07]
- Unable to enforce priorities or SLAs [MICRO'07,'10,'11, ISCA'08'11'12, ASPLOS'10]
- Low system performance [IEEE Micro Top Picks '09,'11a,'11b,'12]

Uncontrollable, unpredictable system

How Do We Solve the Problem?

- Stall-time fair memory scheduling [Mutlu+ MICRO'07]
- Goal: Threads sharing main memory should experience similar slowdowns compared to when they are run alone → fair scheduling
 - Also improves overall system performance by ensuring cores make “proportional” progress
- Idea: Memory controller estimates each thread's slowdown due to interference and schedules requests in a way to balance the slowdowns
- Mutlu and Moscibroda, “Stall-Time Fair Memory Access Scheduling for Chip Multiprocessors,” MICRO 2007.

Stall-Time Fairness in Shared DRAM Systems

- A DRAM system is fair if it equalizes the slowdown of equal-priority threads relative to when each thread is run alone on the same system
- DRAM-related stall-time: The time a thread spends waiting for DRAM memory
- ST_{shared} : DRAM-related stall-time when the thread runs with other threads
- ST_{alone} : DRAM-related stall-time when the thread runs alone
- **Memory-slowdown** = $ST_{\text{shared}}/ST_{\text{alone}}$
 - Relative increase in stall-time
- *Stall-Time Fair Memory scheduler (STFM)* aims to equalize **Memory-slowdown** for interfering threads, without sacrificing performance
 - Considers inherent DRAM performance of each thread
 - Aims to allow proportional progress of threads

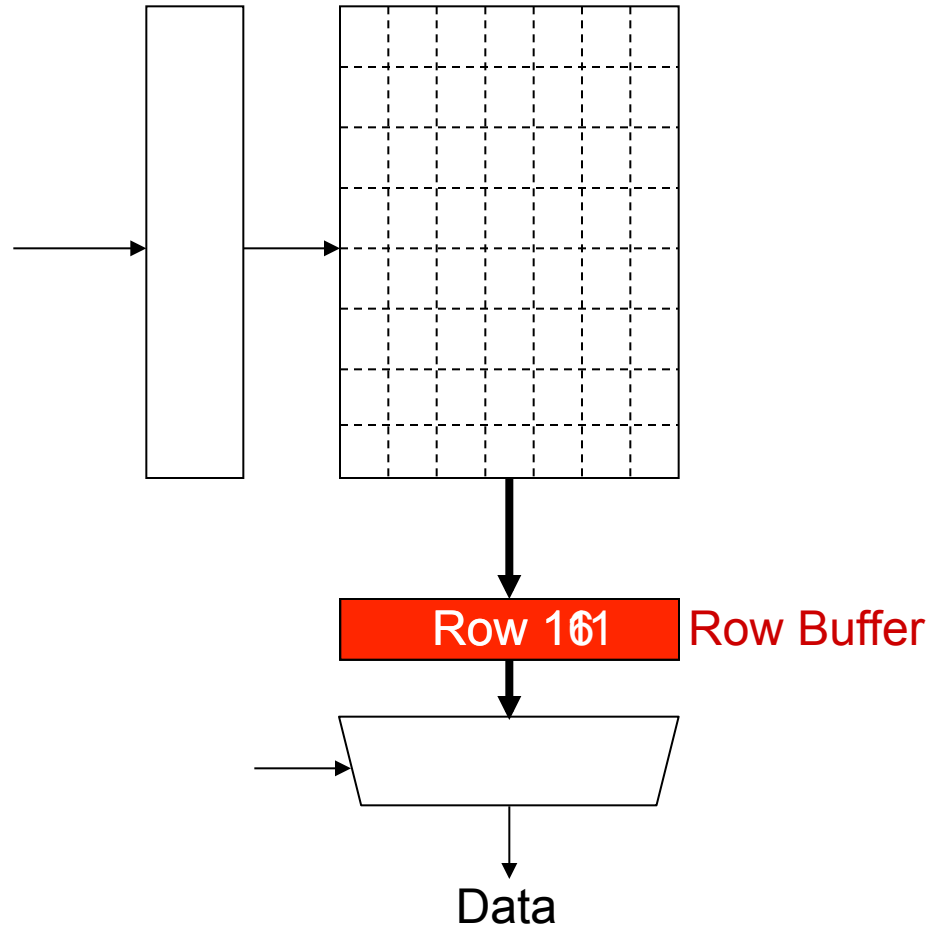
STFM Scheduling Algorithm [MICRO' 07]

- For each thread, the DRAM controller
 - Tracks ST_{shared}
 - Estimates ST_{alone}
- Each cycle, the DRAM controller
 - Computes $\text{Slowdown} = ST_{\text{shared}} / ST_{\text{alone}}$ for threads with legal requests
 - Computes **unfairness = MAX Slowdown / MIN Slowdown**
- If $\text{unfairness} < \alpha$
 - Use DRAM throughput oriented scheduling policy
- **If unfairness $\geq \alpha$**
 - Use fairness-oriented scheduling policy
 - **(1) requests from thread with MAX Slowdown first**
 - (2) row-hit first , (3) oldest-first

How Does STFM Prevent Unfairness?

T0: Row 0
T1: Row 5
T0: Row 0
T1: Row 111
T0: Row 0
T0: Row 06

T0 Slowdown	1.00
T1 Slowdown	1.00
Unfairness	1.00
α	1.05



STFM Pros and Cons

- Upsides:
 - ❑ Identifies fairness as an issue in multi-core memory scheduling
 - ❑ Good at providing fairness
 - ❑ Being fair improves performance

- Downsides:
 - ❑ Does not handle all types of interference
 - ❑ Somewhat complex to implement
 - ❑ Slowdown estimations can be incorrect

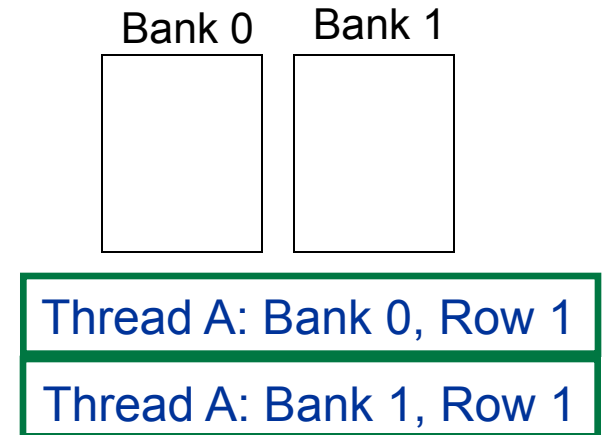
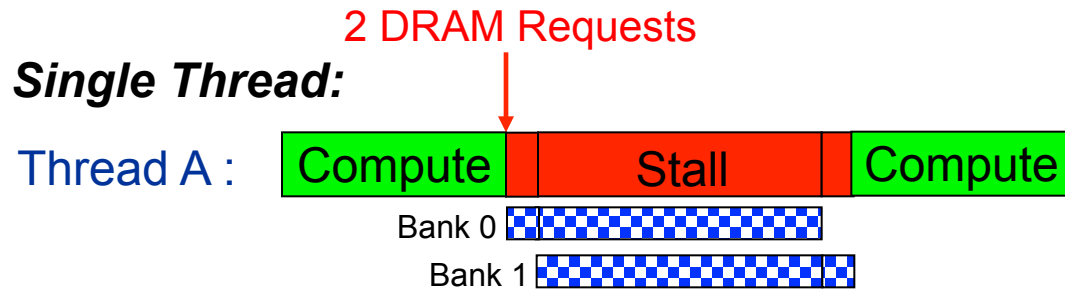
Parallelism-Aware Batch Scheduling

Onur Mutlu and Thomas Moscibroda,
**"Parallelism-Aware Batch Scheduling: Enhancing both
Performance and Fairness of Shared DRAM Systems"**
35th International Symposium on Computer Architecture (ISCA),
pages 63-74, Beijing, China, June 2008. [Slides \(ppt\)](#)

Another Problem due to Interference

- Processors try to tolerate the latency of DRAM requests by generating multiple outstanding requests
 - Memory-Level Parallelism (MLP)
 - Out-of-order execution, non-blocking caches, runahead execution
- Effective only if the DRAM controller actually services the multiple requests in parallel in DRAM banks
- Multiple threads share the DRAM controller
- DRAM controllers are not aware of a thread's MLP
 - Can service each thread's outstanding requests serially, not in parallel

Bank Parallelism of a Thread



Bank access latencies of the two requests overlapped
Thread stalls for ~ONE bank access latency

Bank Parallelism Interference in DRAM

Baseline Scheduler:

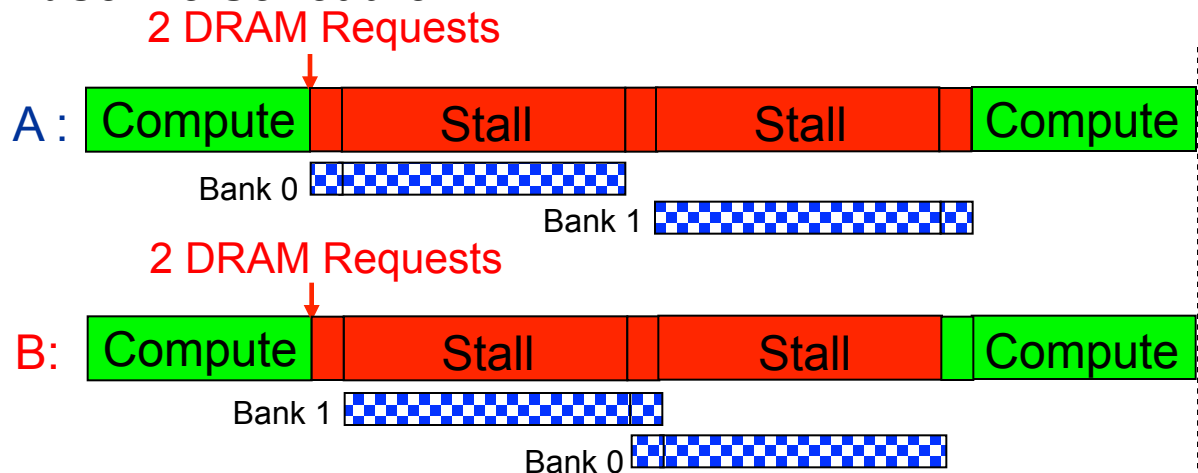


Bank 0	Bank 1
Thread A: Bank 0, Row 1	
Thread B: Bank 1, Row 99	
Thread B: Bank 0, Row 99	
Thread A: Bank 1, Row 1	

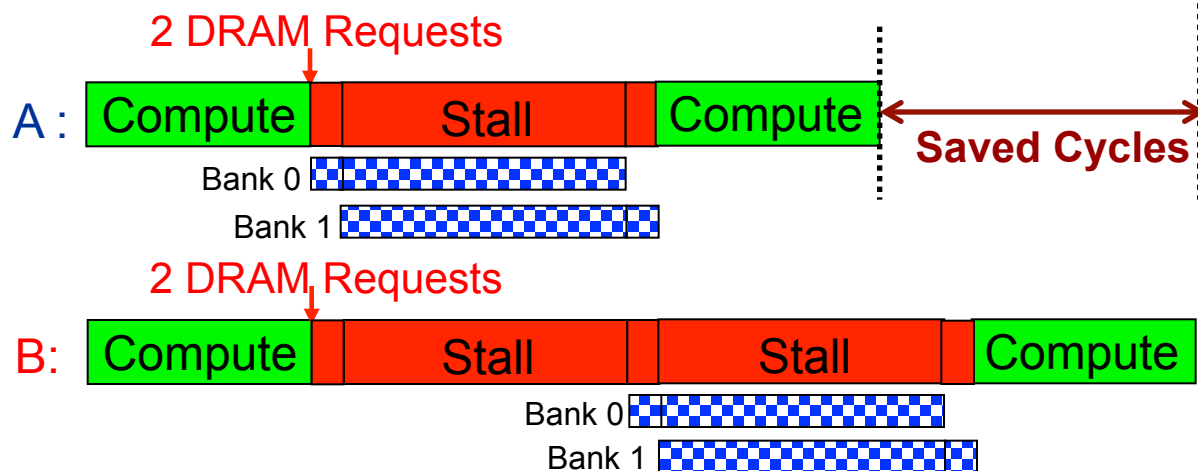
Bank access latencies of each thread serialized
Each thread stalls for ~TWO bank access latencies

Parallelism-Aware Scheduler

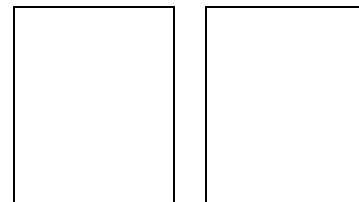
Baseline Scheduler:



Parallelism-aware Scheduler:



Bank 0 Bank 1



Thread A: Bank 0, Row 1

Thread B: Bank 1, Row 99

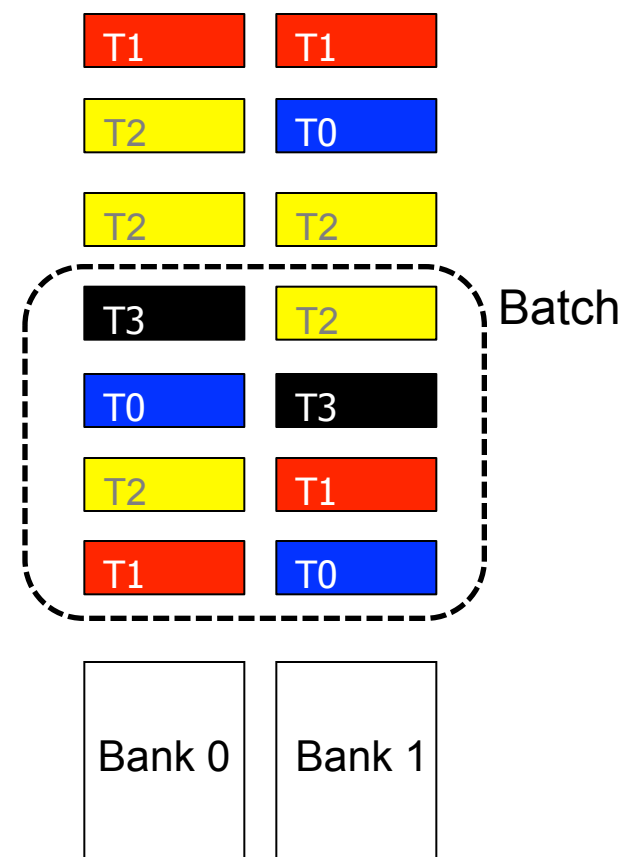
Thread B: Bank 0, Row 99

Thread A: Bank 1, Row 1

**Average stall-time:
~1.5 bank access
latencies**

Parallelism-Aware Batch Scheduling (PAR-BS)

- Principle 1: Parallelism-awareness
 - ❑ Schedule requests from a thread (to different banks) back to back
 - ❑ Preserves each thread's bank parallelism
 - ❑ But, this can cause starvation...
- Principle 2: Request Batching
 - ❑ Group a fixed number of oldest requests from each thread into a "batch"
 - ❑ Service the batch before all other requests
 - ❑ Form a new batch when the current one is done
 - ❑ Eliminates starvation, provides fairness
 - ❑ Allows parallelism-awareness within a batch



PAR-BS Components

- Request batching
- Within-batch scheduling
 - Parallelism aware

Request Batching

- Each memory request has a bit (*marked*) associated with it
- Batch formation:
 - Mark up to *Marking-Cap* oldest requests per bank for each thread
 - Marked requests constitute the batch
 - Form a new batch when no marked requests are left
- Marked requests are prioritized over unmarked ones
 - No reordering of requests across batches: **no starvation, high fairness**
- **How to prioritize requests within a batch?**

Within-Batch Scheduling

- Can use any existing DRAM scheduling policy
 - FR-FCFS (row-hit first, then oldest-first) exploits row-buffer locality
- But, we also want to preserve intra-thread bank parallelism
 - Service each thread's requests back to back

HOW?

- Scheduler computes a **ranking of threads** when the batch is formed
 - Higher-ranked threads are prioritized over lower-ranked ones
 - Improves the likelihood that requests from a thread are serviced in parallel by different banks
 - Different threads prioritized in the same order across ALL banks

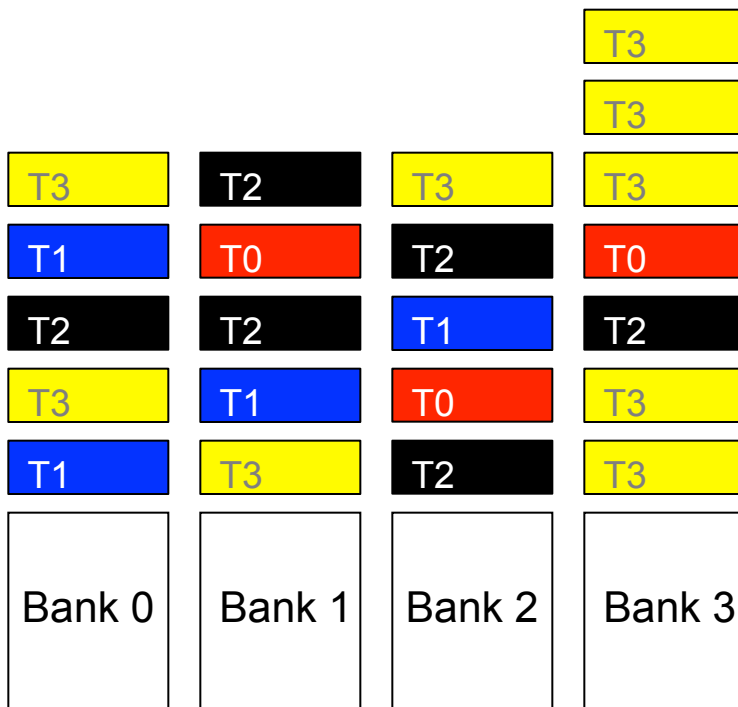
How to Rank Threads within a Batch

- Ranking scheme affects system throughput and fairness
- Maximize system throughput
 - Minimize average stall-time of threads within the batch
- Minimize unfairness (Equalize the slowdown of threads)
 - Service threads with inherently low stall-time early in the batch
 - Insight: delaying memory non-intensive threads results in high slowdown
- Shortest stall-time first (shortest job first) ranking
 - Provides optimal system throughput [Smith, 1956]*
 - Controller estimates each thread's stall-time within the batch
 - Ranks threads with shorter stall-time higher

* W.E. Smith, "Various optimizers for single stage production," Naval Research Logistics Quarterly, 1956.

Shortest Stall-Time First Ranking

- Maximum number of marked requests to any bank (max-bank-load)
 - Rank thread with lower max-bank-load higher (~ low stall-time)
- Total number of marked requests (total-load)
 - Breaks ties: rank thread with lower total-load higher

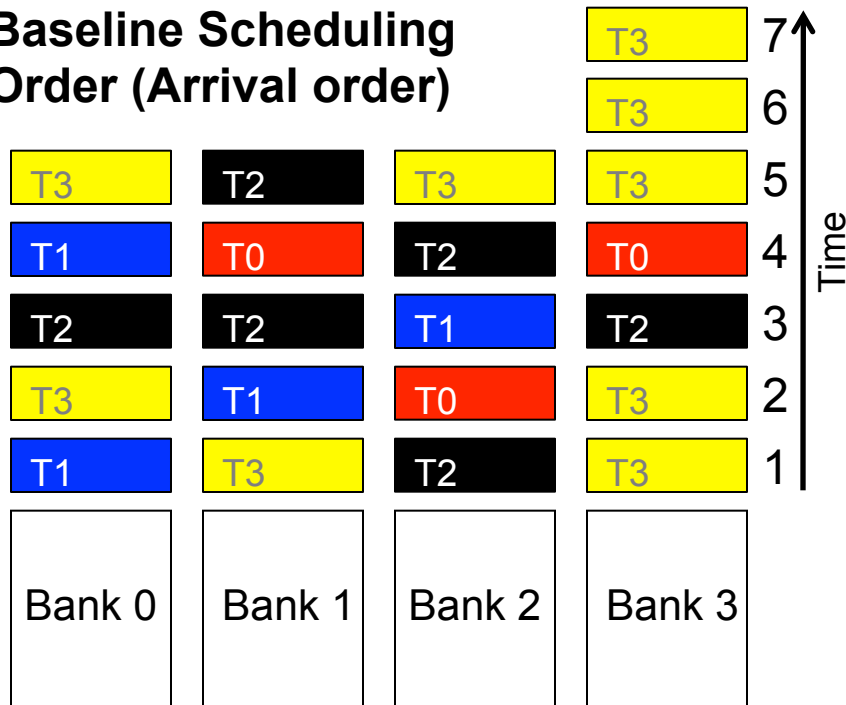


	max-bank-load	total-load

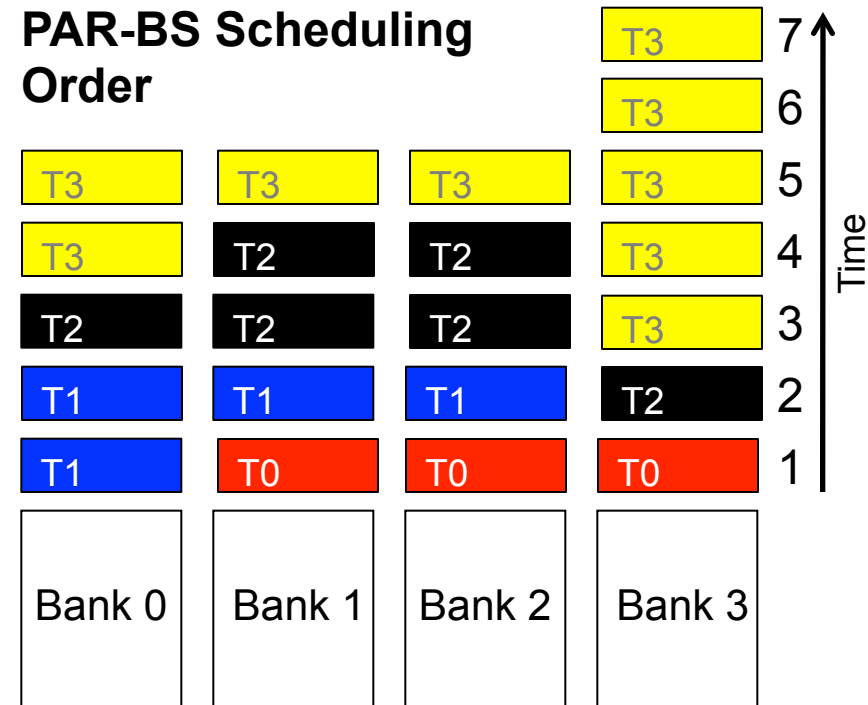
Ranking:
T0 > T1 > T2 > T3

Example Within-Batch Scheduling Order

Baseline Scheduling Order (Arrival order)



PAR-BS Scheduling Order



Ranking: T0 > T1 > T2 > T3

	T0	T1	T2	T3
Stall times				

AVG: 5 bank access latencies

	T0	T1	T2	T3
Stall times				

AVG: 3.5 bank access latencies

Putting It Together: PAR-BS Scheduling Policy

■ PAR-BS Scheduling Policy

(1) Marked requests first

Batching

(2) Row-hit requests first

(3) Higher-rank thread first (shortest stall-time first)

Parallelism-aware
within-batch
scheduling

(4) Oldest first

■ Three properties:

- ❑ Exploits row-buffer locality **and** intra-thread bank parallelism
- ❑ Work-conserving
 - Services unmarked requests to banks without marked requests
- ❑ Marking-Cap is important
 - Too small cap: destroys row-buffer locality
 - Too large cap: penalizes memory non-intensive threads

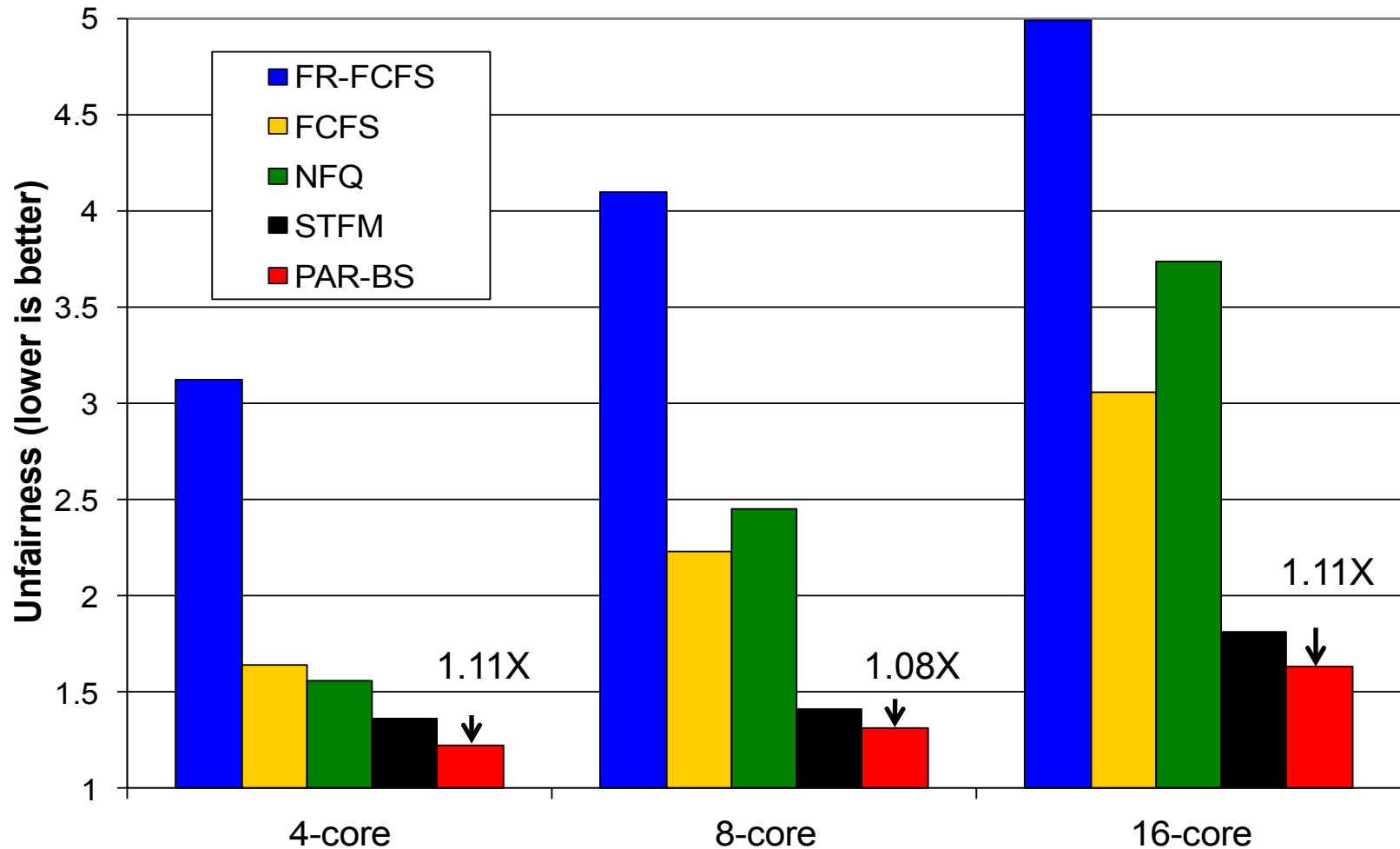
■ Many more trade-offs analyzed in the paper

Hardware Cost

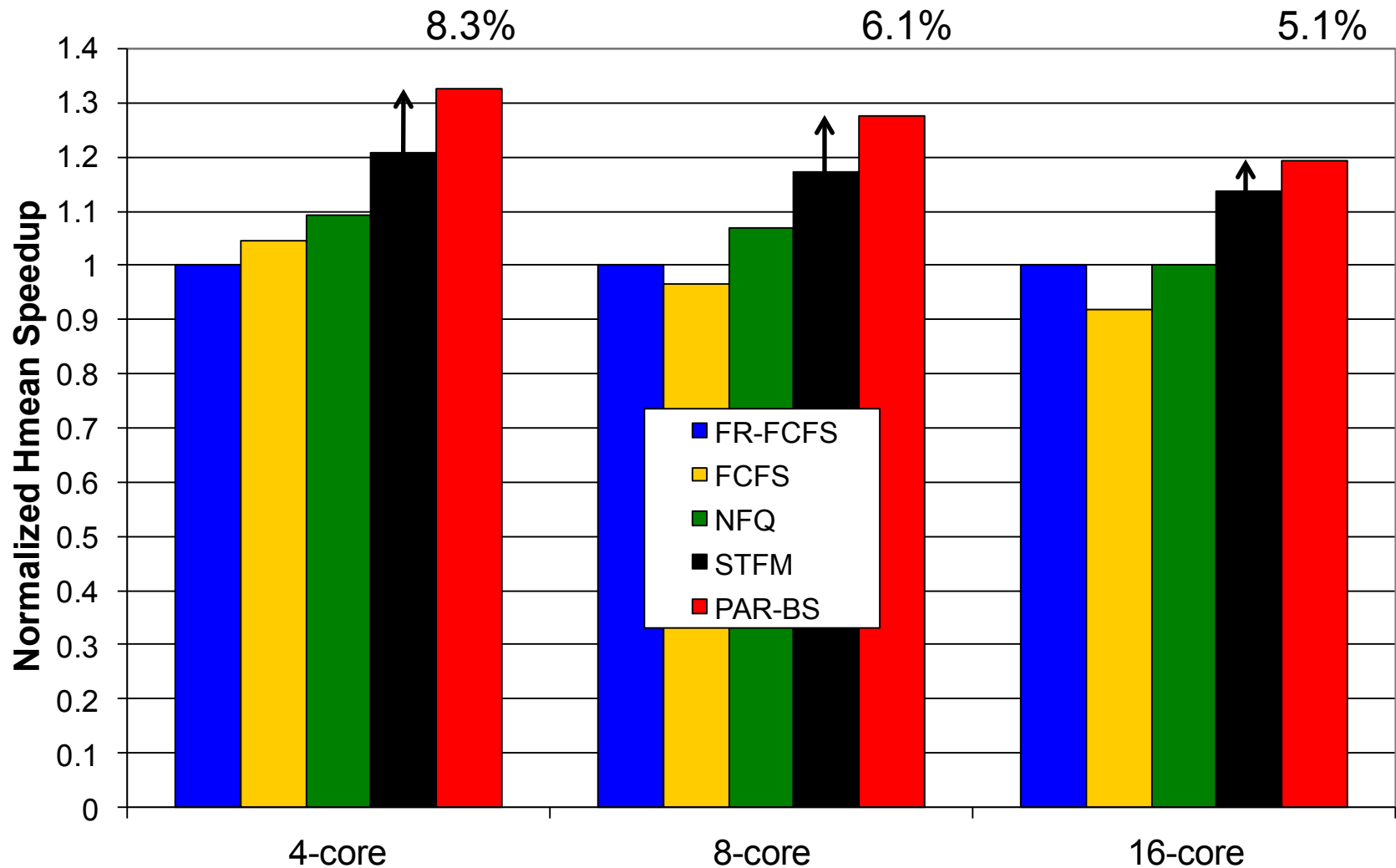
- <1.5KB storage cost for
 - 8-core system with 128-entry memory request buffer
- No complex operations (e.g., divisions)
- Not on the critical path
 - Scheduler makes a decision only every DRAM cycle

Unfairness on 4-, 8-, 16-core Systems

Unfairness = MAX Memory Slowdown / MIN Memory Slowdown [MICRO 2007]



System Performance (Hmean-speedup)



PAR-BS Pros and Cons

- Upsides:
 - Identifies the problem of bank parallelism destruction across multiple threads
 - Simple mechanism

- Downsides:
 - Does not always prioritize the latency-sensitive applications → lower overall throughput
 - Implementation in multiple controllers needs coordination for best performance → too frequent coordination since batching is done frequently

ATLAS Memory Scheduler

Yoongu Kim, Dongsu Han, Onur Mutlu, and Mor Harchol-Balter,

"ATLAS: A Scalable and High-Performance

Scheduling Algorithm for Multiple Memory Controllers"

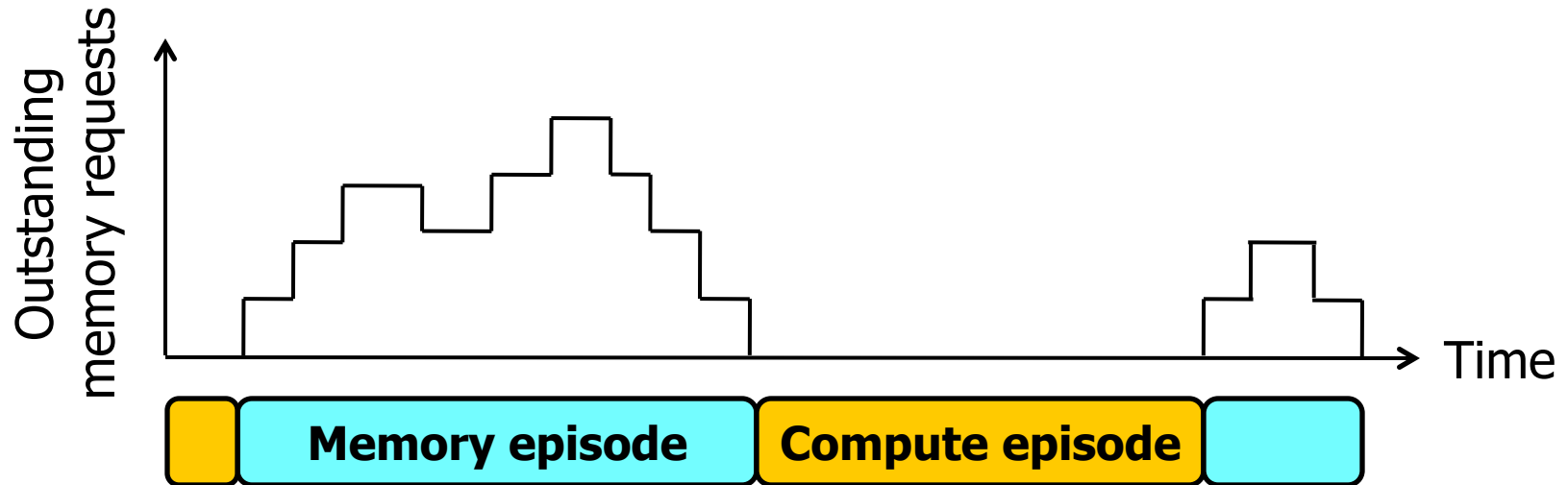
16th International Symposium on High-Performance Computer Architecture (HPCA),

Bangalore, India, January 2010. [Slides \(pptx\)](#)

Rethinking Memory Scheduling

A thread alternates between two states (episodes)

- **Compute episode:** Zero outstanding memory requests → **High IPC**
- **Memory episode:** Non-zero outstanding memory requests → **Low IPC**



Goal: Minimize time spent in memory episodes

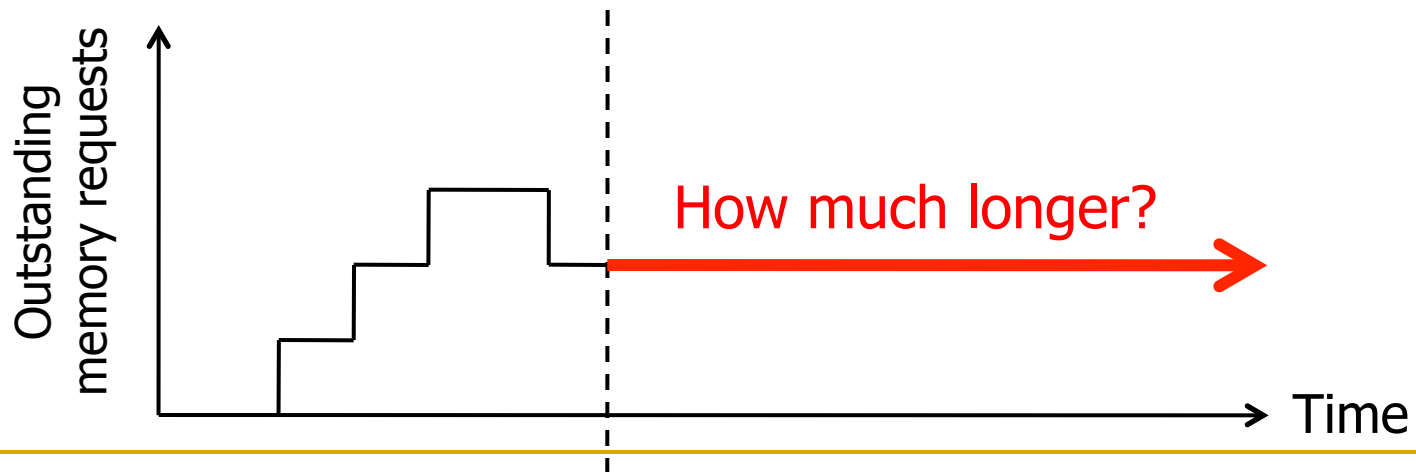
How to Minimize Memory Episode Time



Prioritize thread whose memory episode will end the soonest

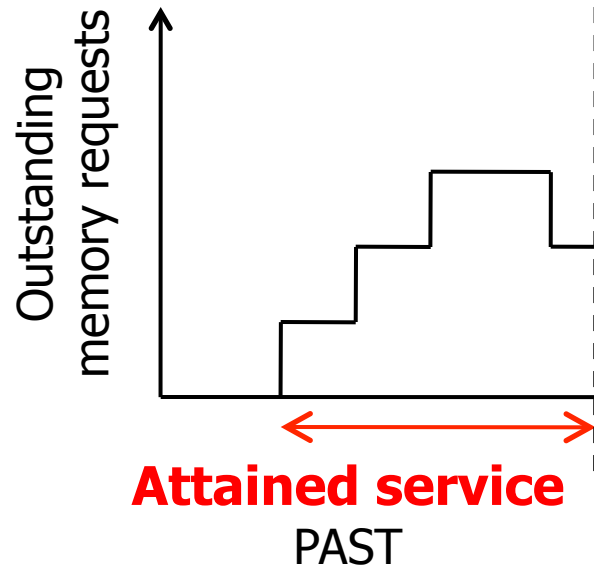
- Minimizes time spent in memory episodes across all threads
- Supported by queueing theory:
 - **Shortest-Remaining-Processing-Time** scheduling is optimal in single-server queue

Remaining length of a memory episode?



Predicting Memory Episode Lengths

We discovered: past is excellent predictor for future

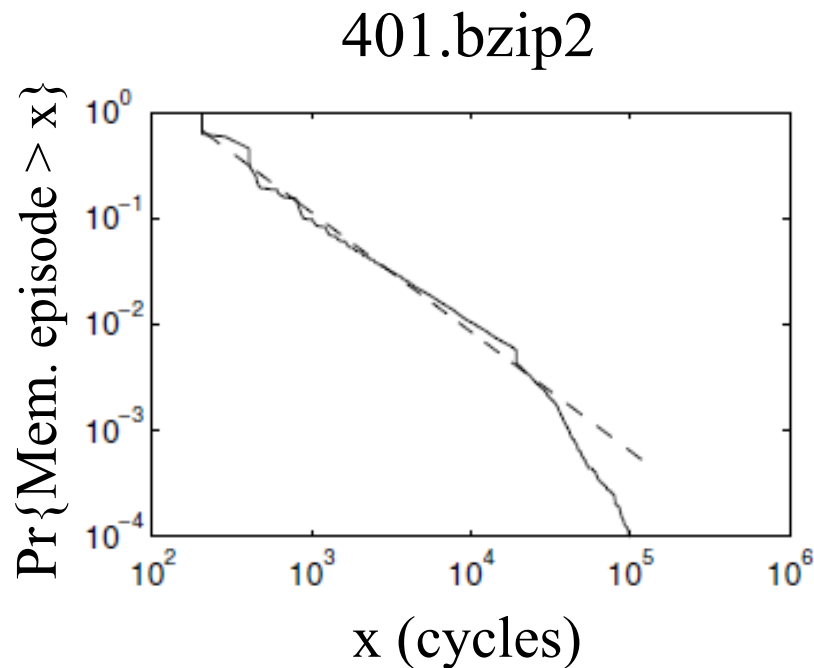


Large **attained service** → Large expected **remaining service**

Q: Why?

A: Memory episode lengths are **Pareto distributed...**

Pareto Distribution of Memory Episode Lengths



Memory episode lengths of
SPEC benchmarks

Pareto distribution

The longer an episode has lasted
→ The longer it will last further

Attained service correlates with
remaining service

Favoring **least-attained-service** memory episode
= Favoring memory episode which will **end the soonest**

Least Attained Service (LAS) Memory Scheduling

Our Approach

Prioritize the memory episode with least-**remaining**-service

- Remaining service: Correlates with attained service
- Attained service: Tracked by per-thread counter

Prioritize the memory episode with least-**attained**-service

Least-attained-service (LAS) scheduling:
Minimize memory episode time

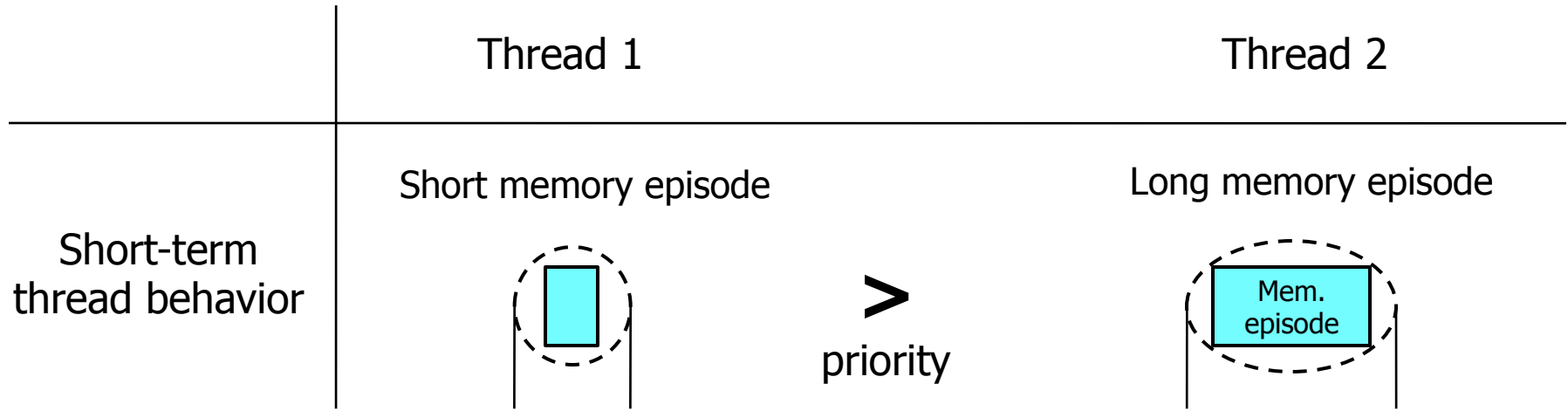
Queueing Theory

Prioritize the job with shortest-remaining-processing-time

Provably optimal

However, LAS does not consider long-term thread behavior

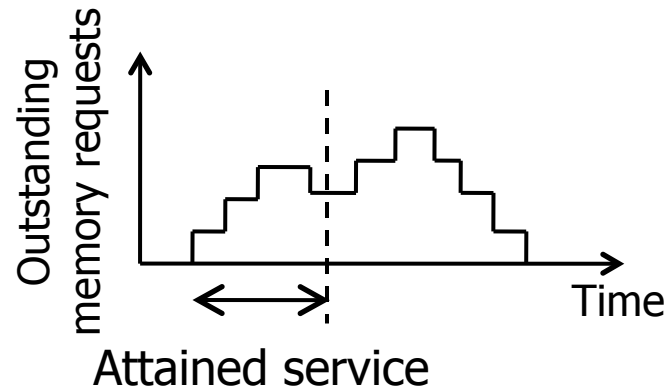
Long-Term Thread Behavior



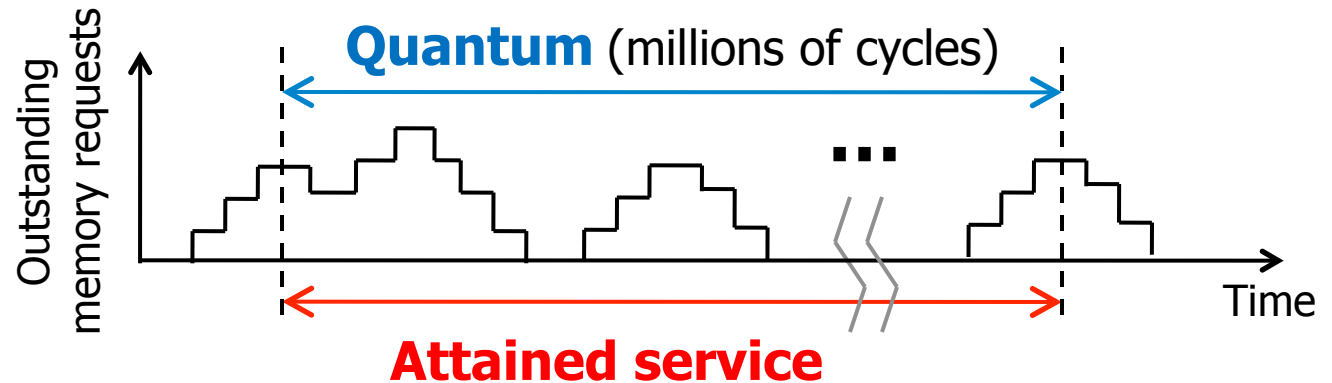
Prioritizing Thread 2 is more beneficial:
results in very long stretches of compute episodes

Quantum-Based Attained Service of a Thread

Short-term
thread behavior



Long-term
thread behavior



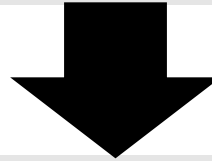
We divide time into large, fixed-length intervals:
quanta (millions of cycles)

LAS Thread Ranking

During a quantum

Each thread's attained service (AS) is tracked by MCs

$AS_i = A \text{ thread's AS during only the } i\text{-th quantum}$



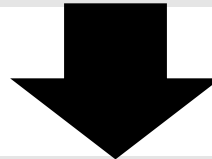
End of a quantum

Each thread's **TotalAS** computed as:

$$TotalAS_i = \alpha \cdot TotalAS_{i-1} + (1 - \alpha) \cdot AS_i$$

High $\alpha \rightarrow$ More bias towards history

Threads are ranked, favoring threads with lower TotalAS



Next quantum

Threads are serviced according to their ranking

ATLAS Scheduling Algorithm

ATLAS

- **A**daptive per-**T**hread **L**east **A**ttained **S**ervice
- Request prioritization order

1. **Prevent starvation**: Over threshold request

2. **Maximize performance**: Higher LAS rank

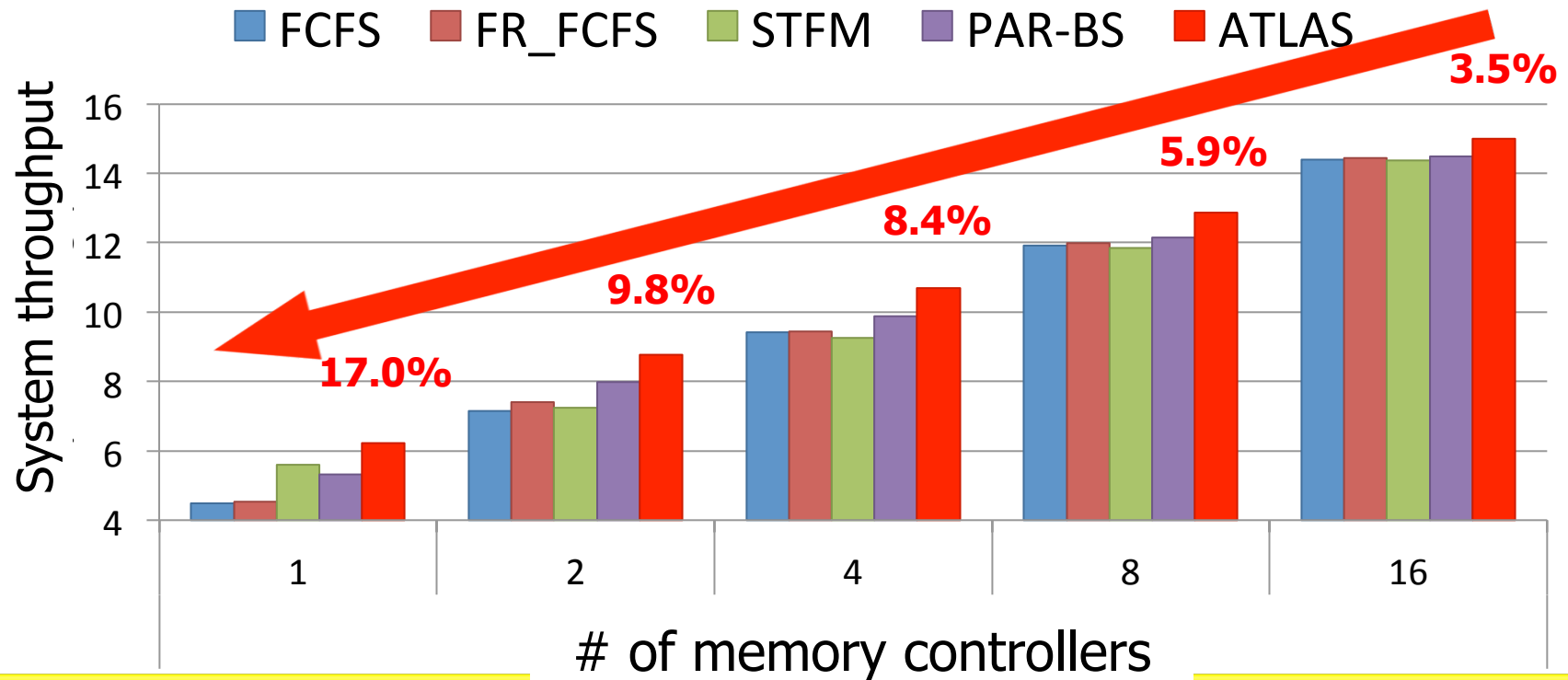
3. **Exploit locality**: Row-hit request

4. **Tie-breaker**: Oldest request

How to coordinate MCs to agree upon a consistent ranking?

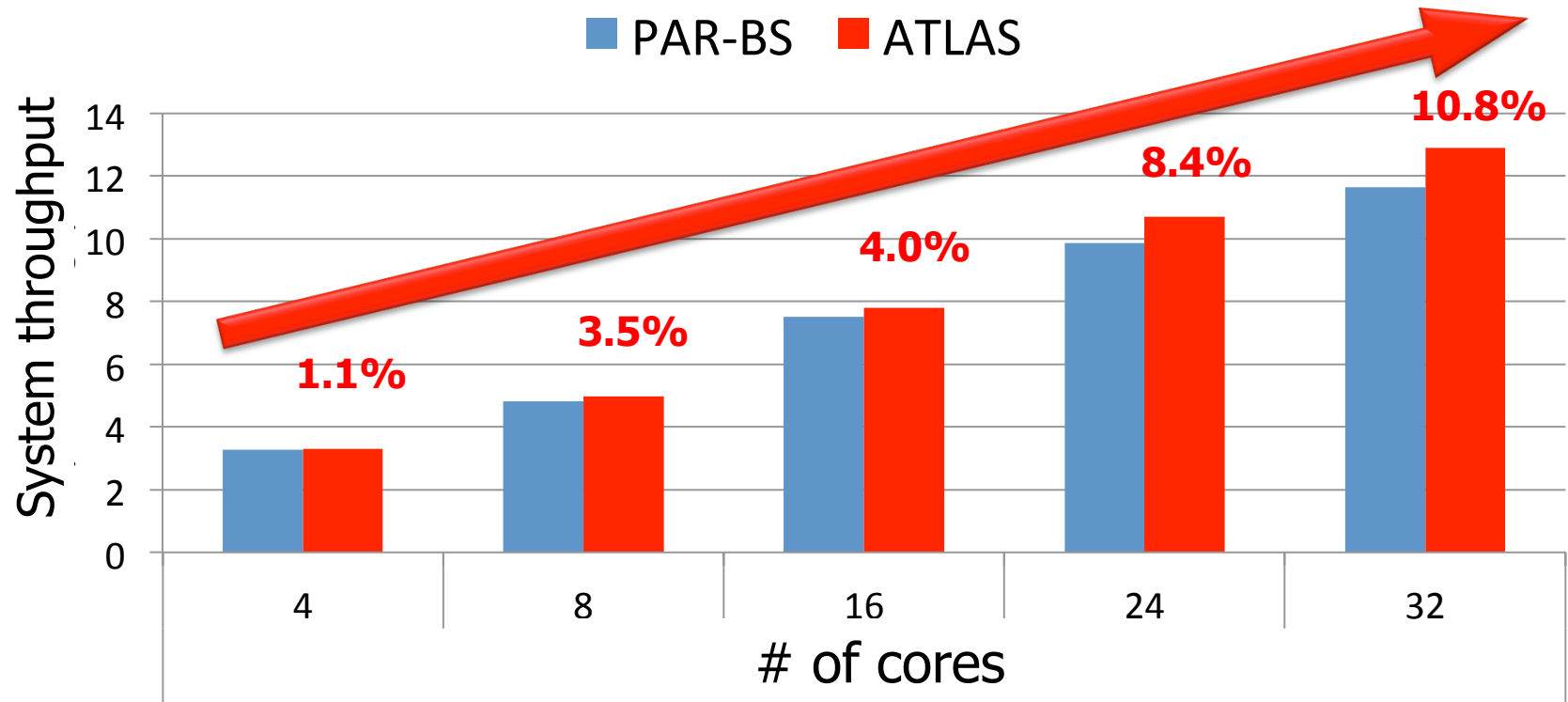
System Throughput: 24-Core System

$$\text{System throughput} = \sum \text{Speedup}$$



ATLAS consistently provides higher system throughput than all previous scheduling algorithms

System Throughput: 4-MC System



of cores increases → ATLAS performance benefit increases

Properties of ATLAS

Goals

- Maximize system performance
- Scalable to large number of controllers
- Configurable by system software

Properties of ATLAS

- LAS-ranking
- Bank-level parallelism
- Row-buffer locality
- Very infrequent coordination
- 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 overall throughput (compute-intensive threads are prioritized)
- ❑ Low complexity
- ❑ Coordination among controllers happens infrequently

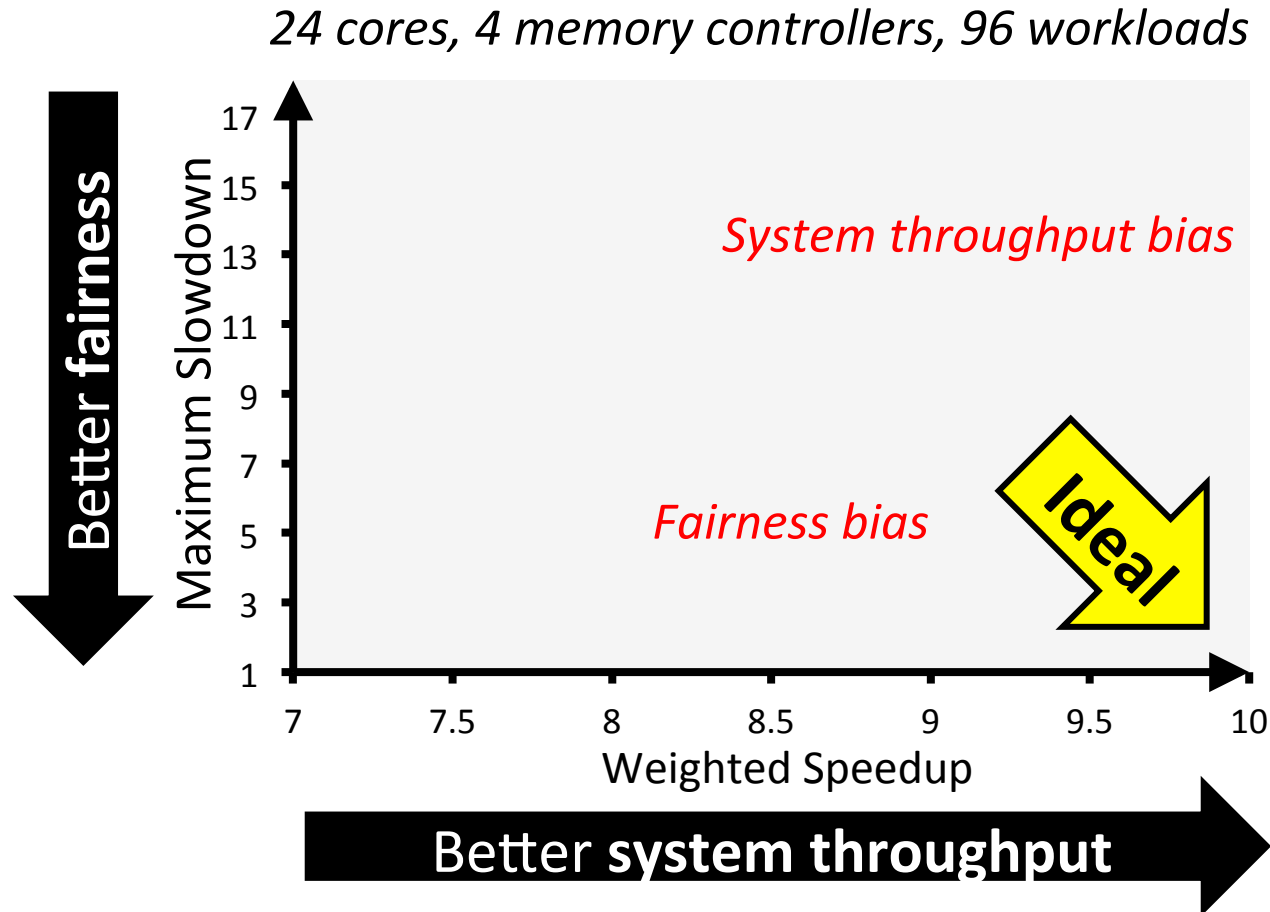
■ Downsides:

- ❑ 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 \(pptx\)](#) [\(pdf\)](#)

Previous Scheduling Algorithms are Biased

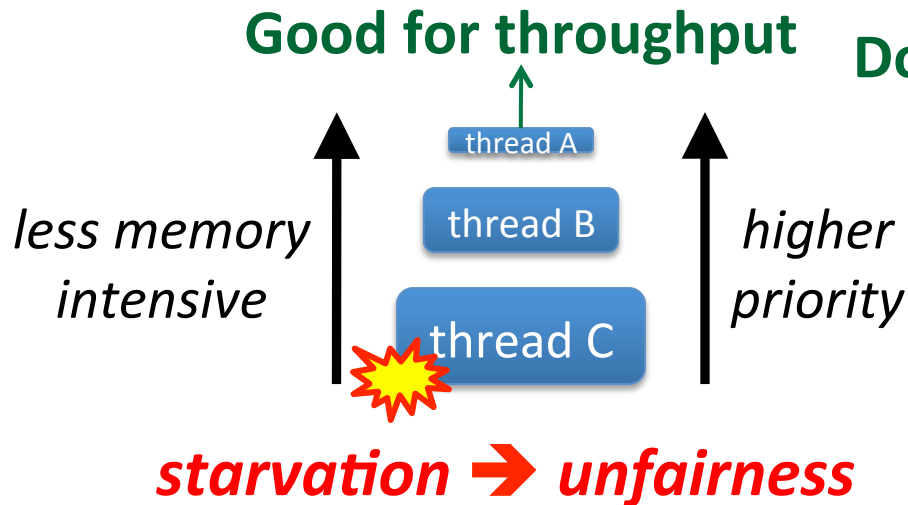


No previous memory scheduling algorithm provides both the best fairness and system throughput

Throughput vs. Fairness

Throughput biased approach

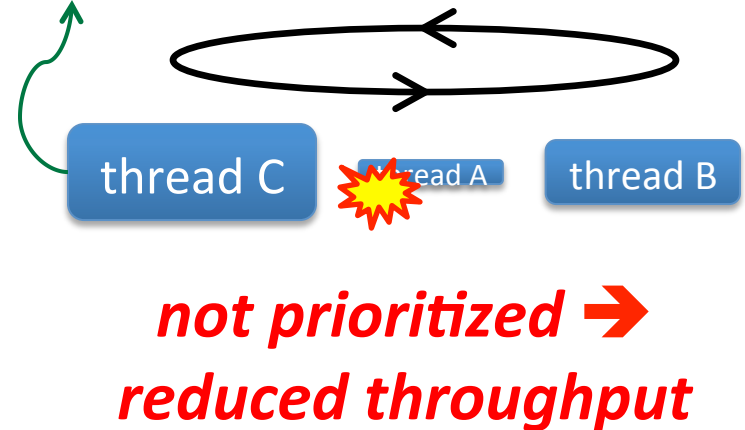
Prioritize less memory-intensive threads



Fairness biased approach

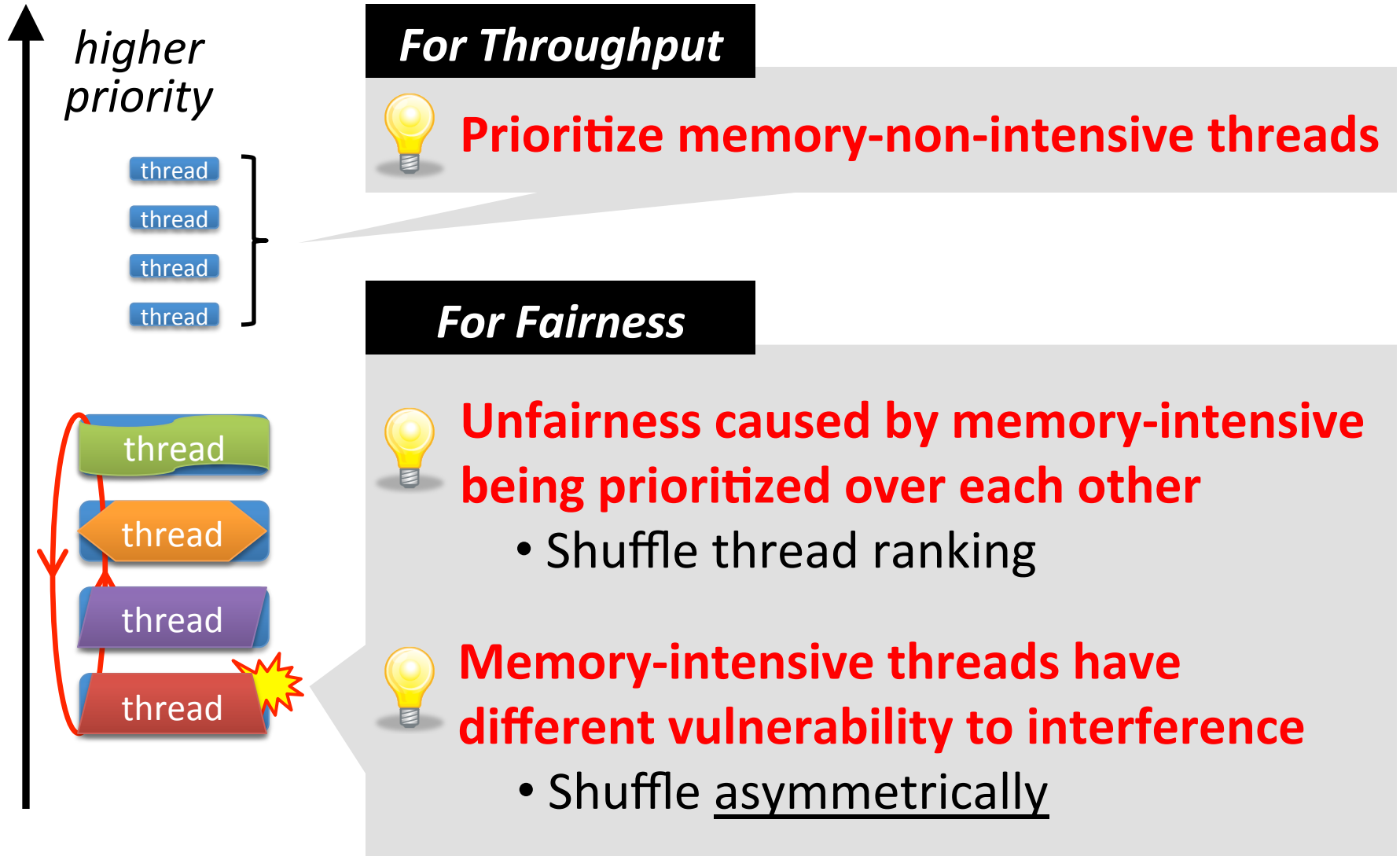
Take turns accessing memory

Does not starve



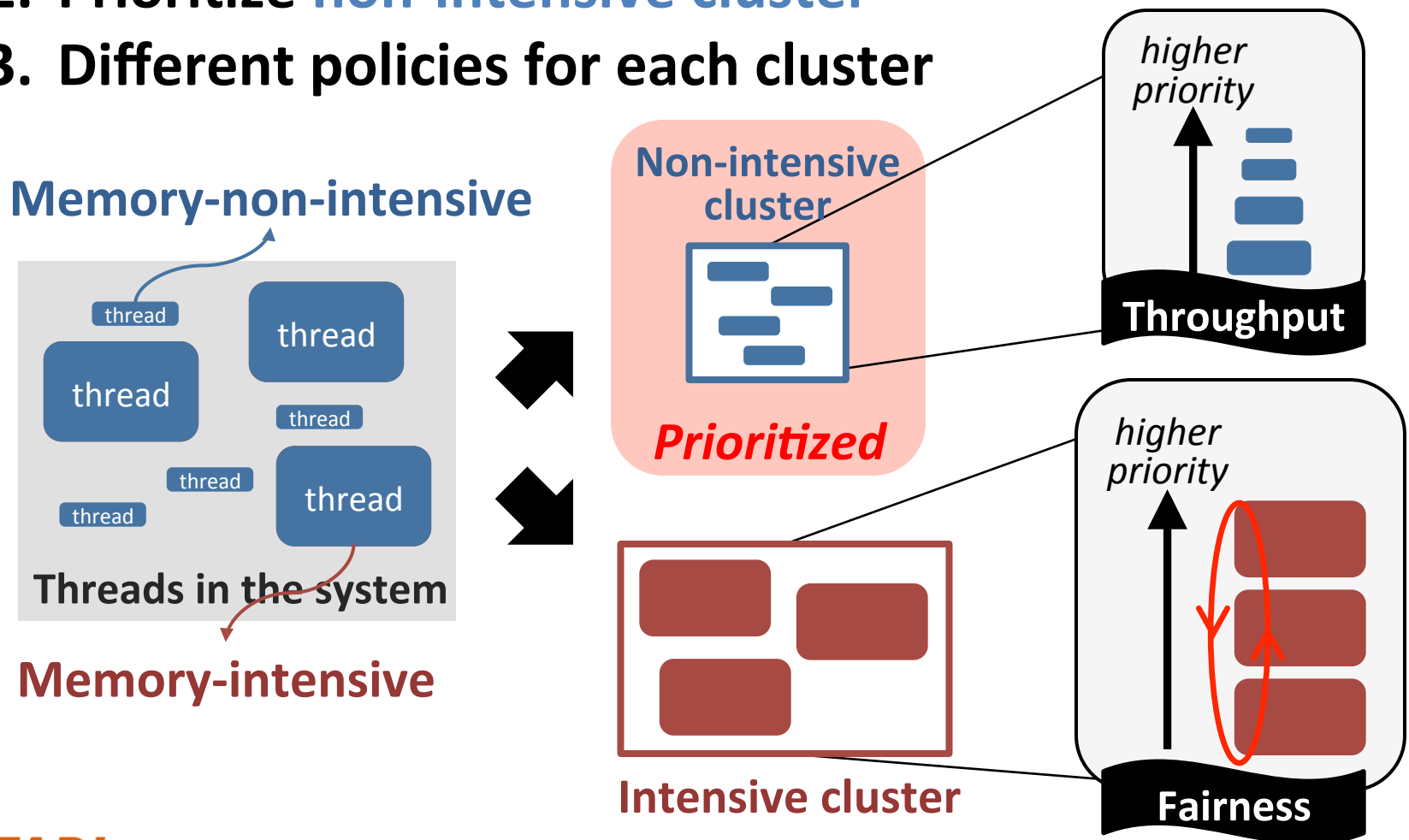
Single policy for all threads is insufficient

Achieving the Best of Both Worlds



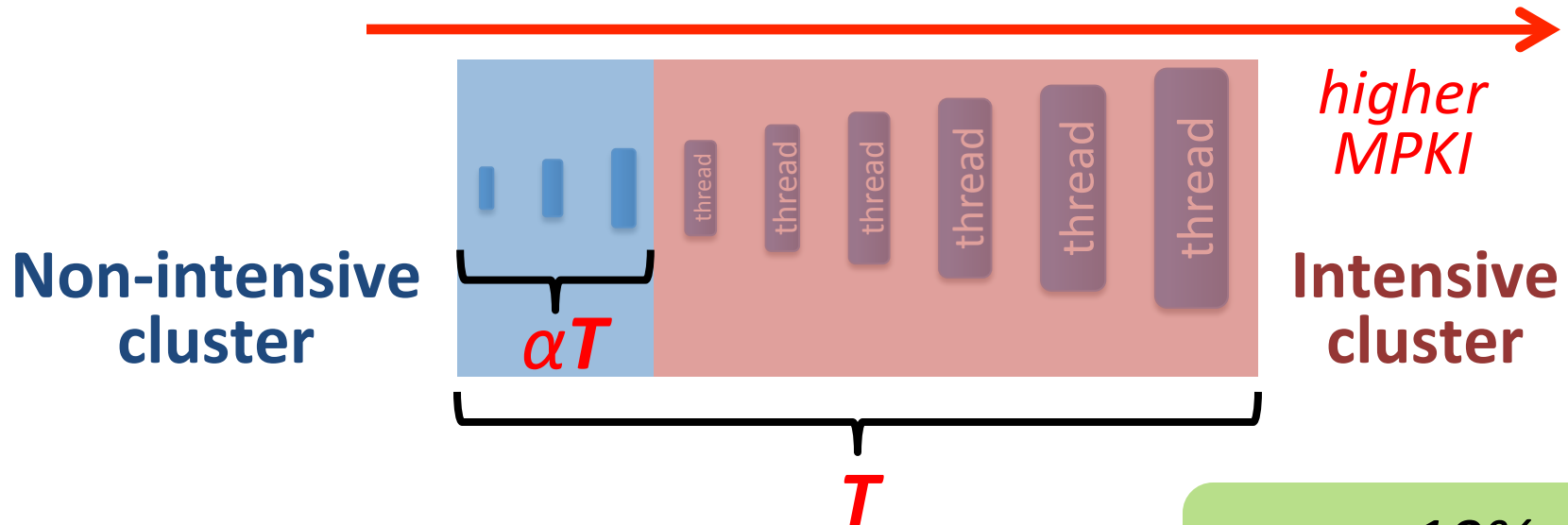
Thread Cluster Memory Scheduling [Kim+ MICRO'10]

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*

$\alpha < 10\%$
ClusterThreshold

Step2 Memory bandwidth usage αT divides clusters

Prioritization Between Clusters

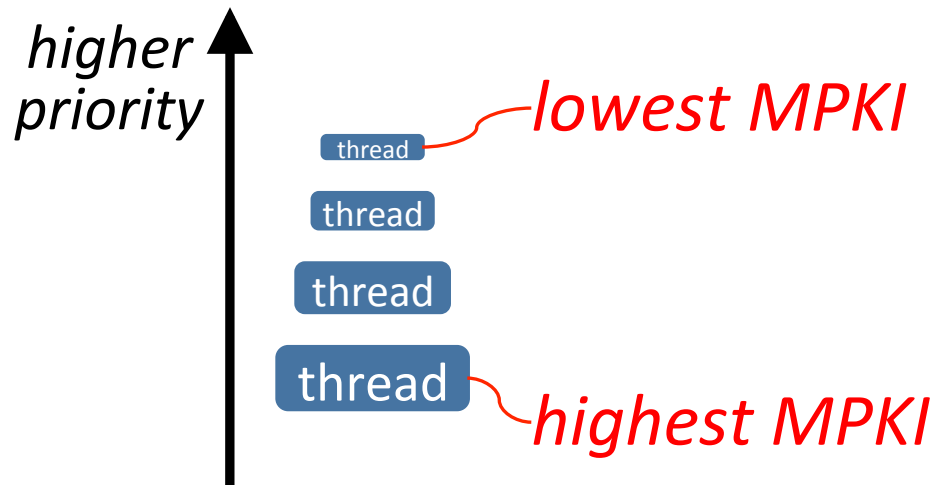
Prioritize *non-intensive* cluster



- **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

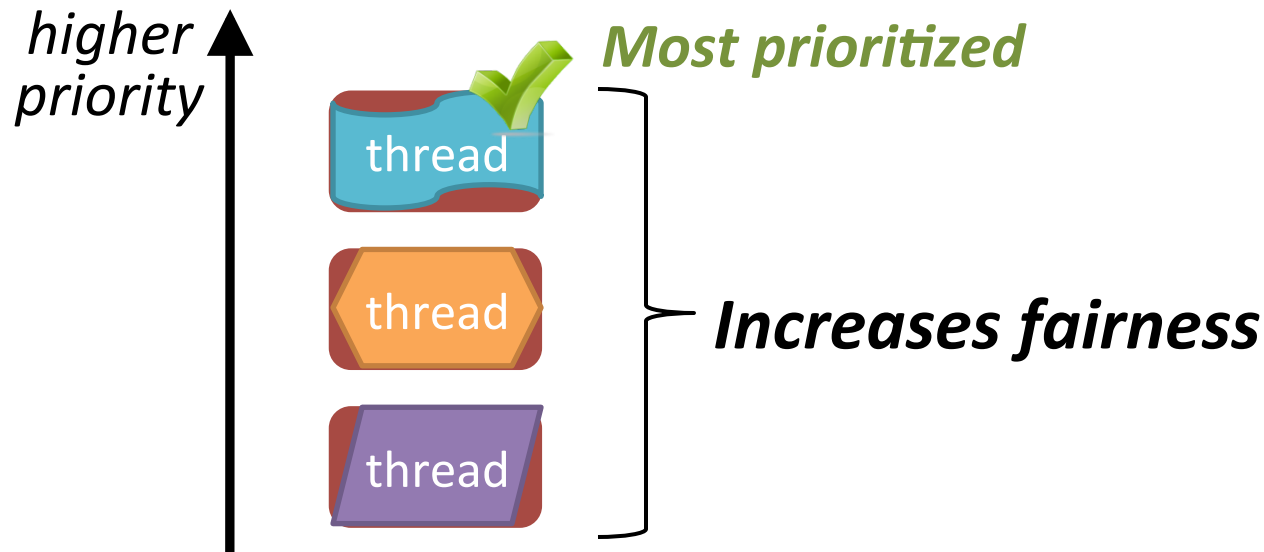
Non-Intensive Cluster

Prioritize threads according to MPKI



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

Intensive Cluster



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

Case Study: A Tale of Two Threads

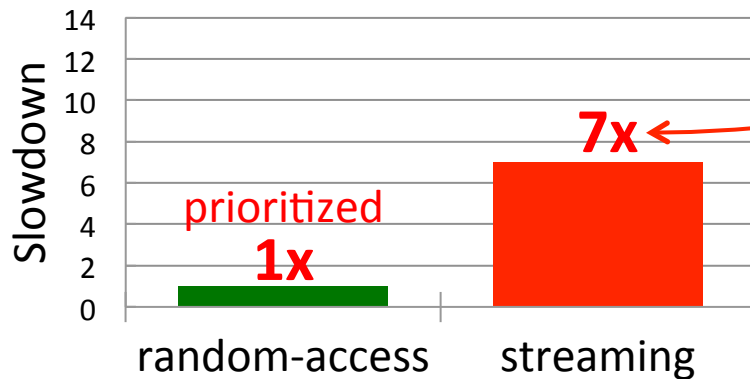
Case Study: Two intensive threads contending

1.random-access

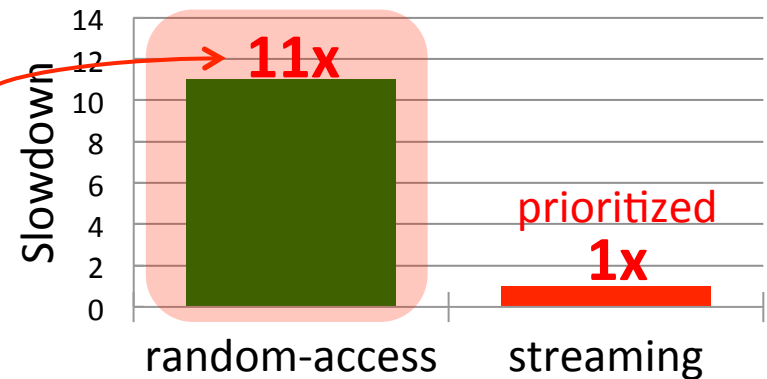
2.streaming

} *Which is slowed down more easily?*

Prioritize *random-access*



Prioritize *streaming*



random-access thread is more easily slowed down

Why are Threads Different?

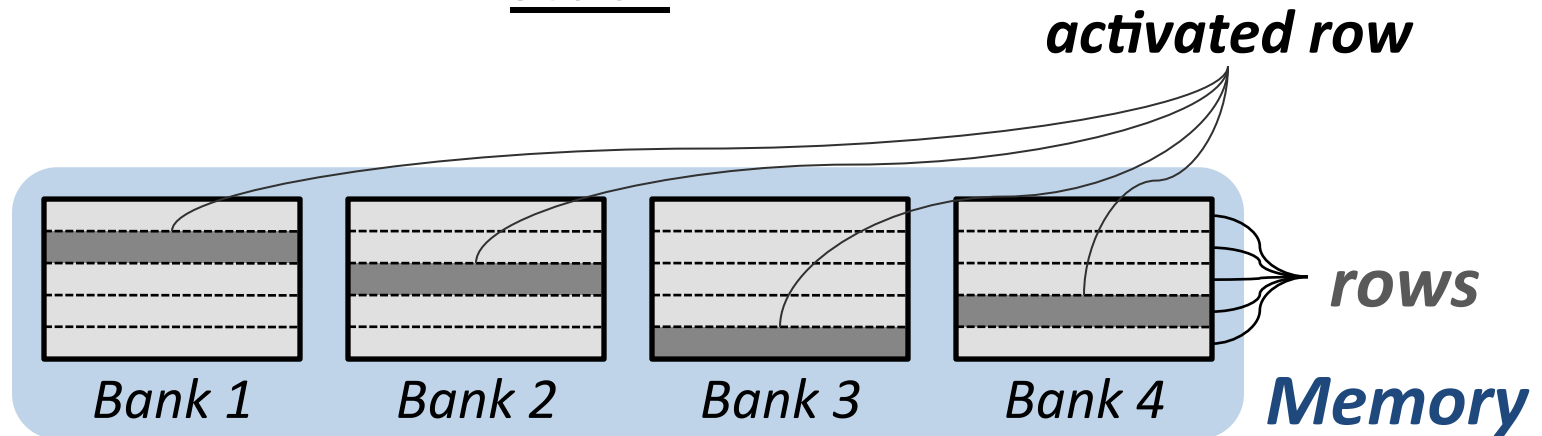
random-access

req

stuck →

streaming

req



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

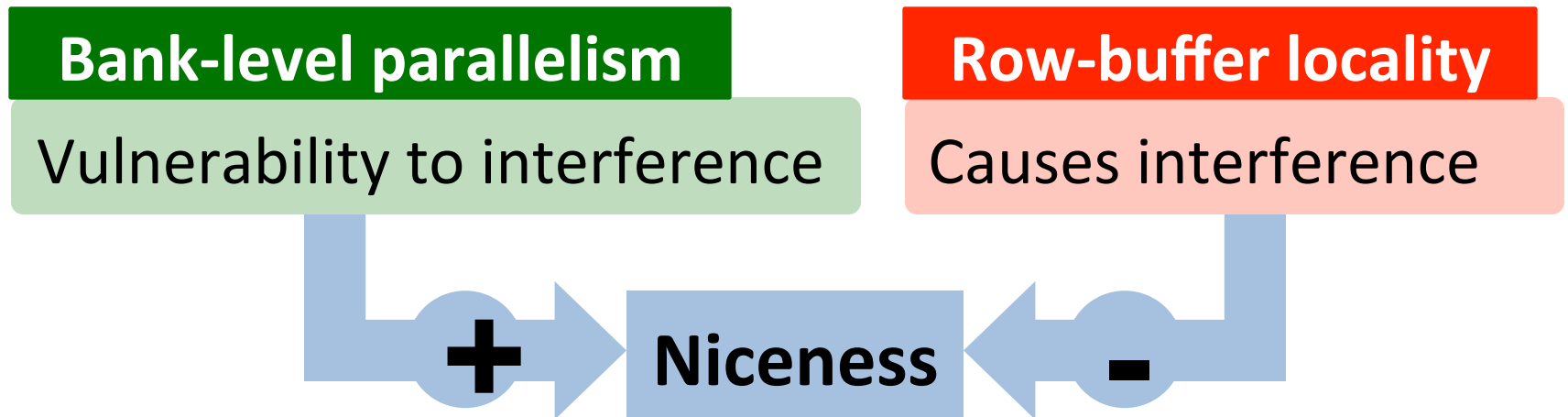
- All requests → Same row
- High **row-buffer locality**



Vulnerable to interference

Niceness

How to quantify difference between threads?



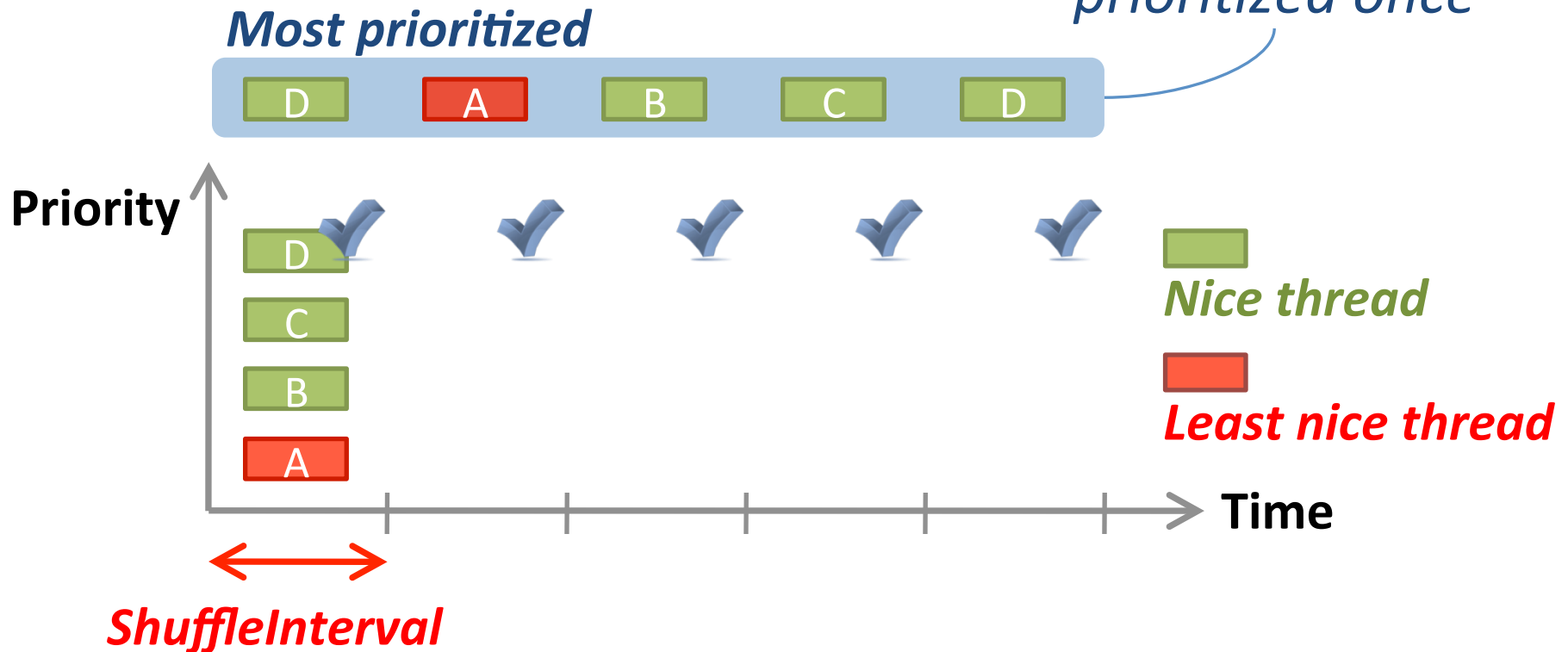
Shuffling: Round-Robin vs. Niceness-Aware

1. Round-Robin shuffling

← *What can go wrong?*

2. Niceness-Aware shuffling

GOOD: *Each thread prioritized once*



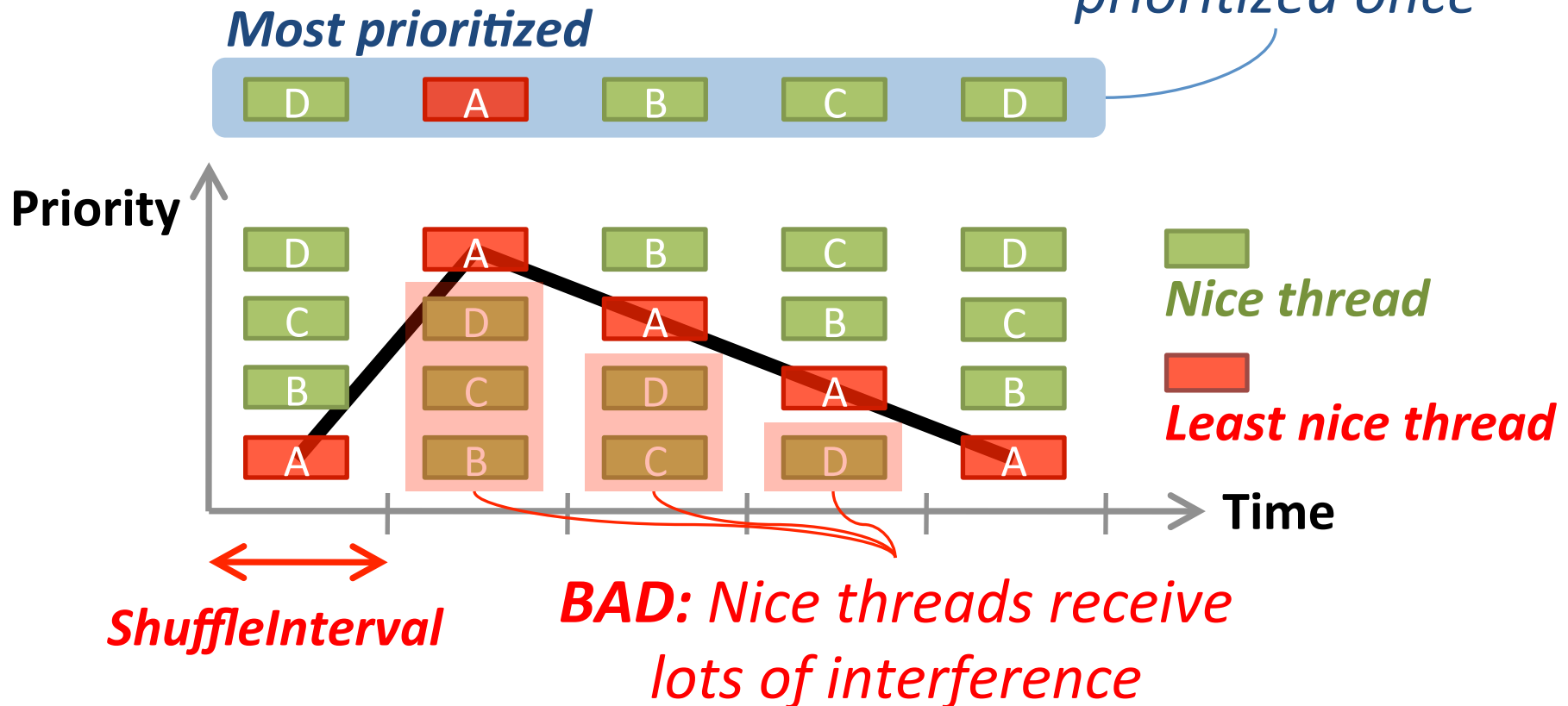
Shuffling: Round-Robin vs. Niceness-Aware

1. Round-Robin shuffling

← What can go wrong?

2. Niceness-Aware shuffling

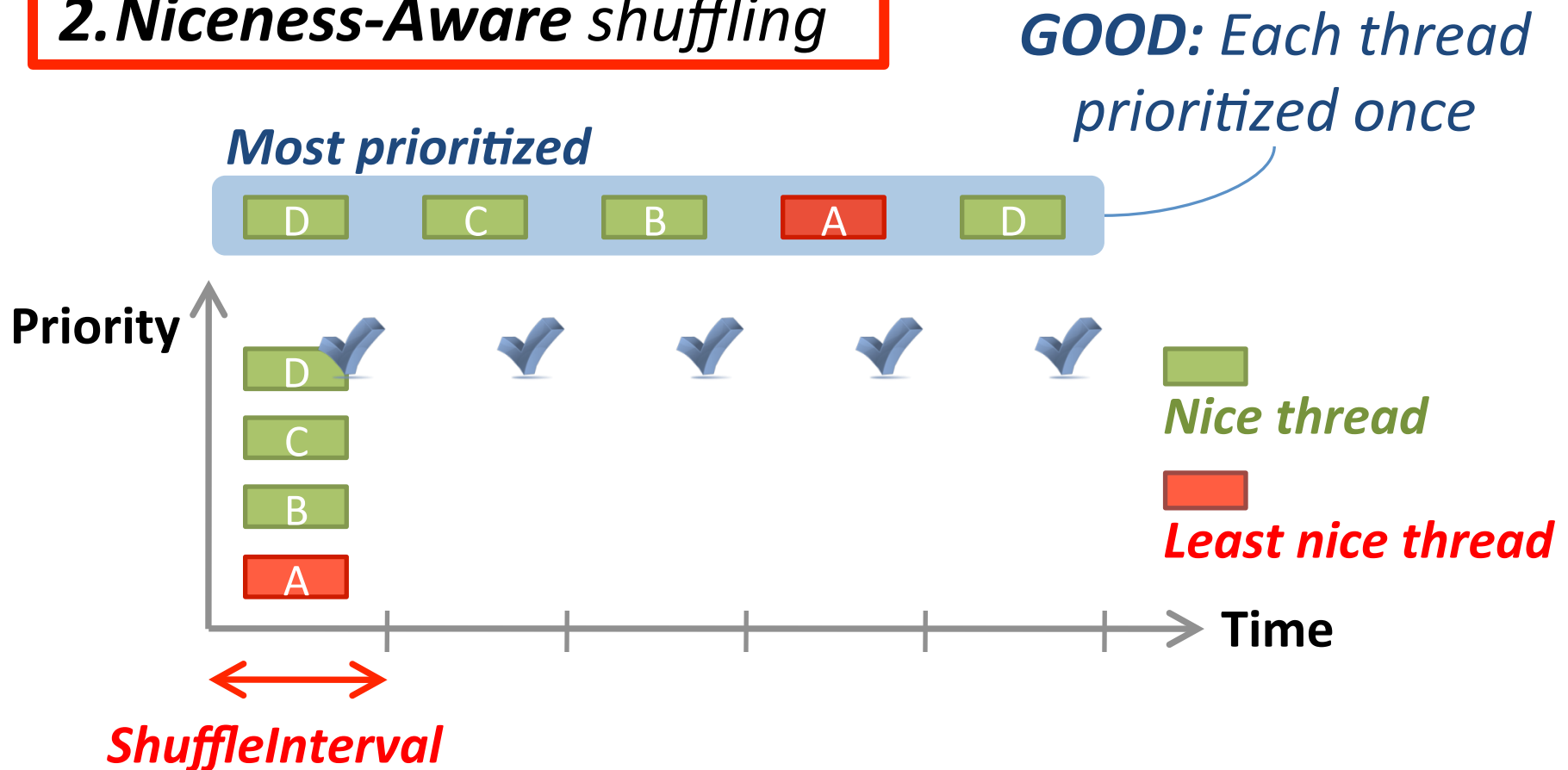
GOOD: Each thread prioritized once



Shuffling: Round-Robin vs. Niceness-Aware

1. Round-Robin shuffling

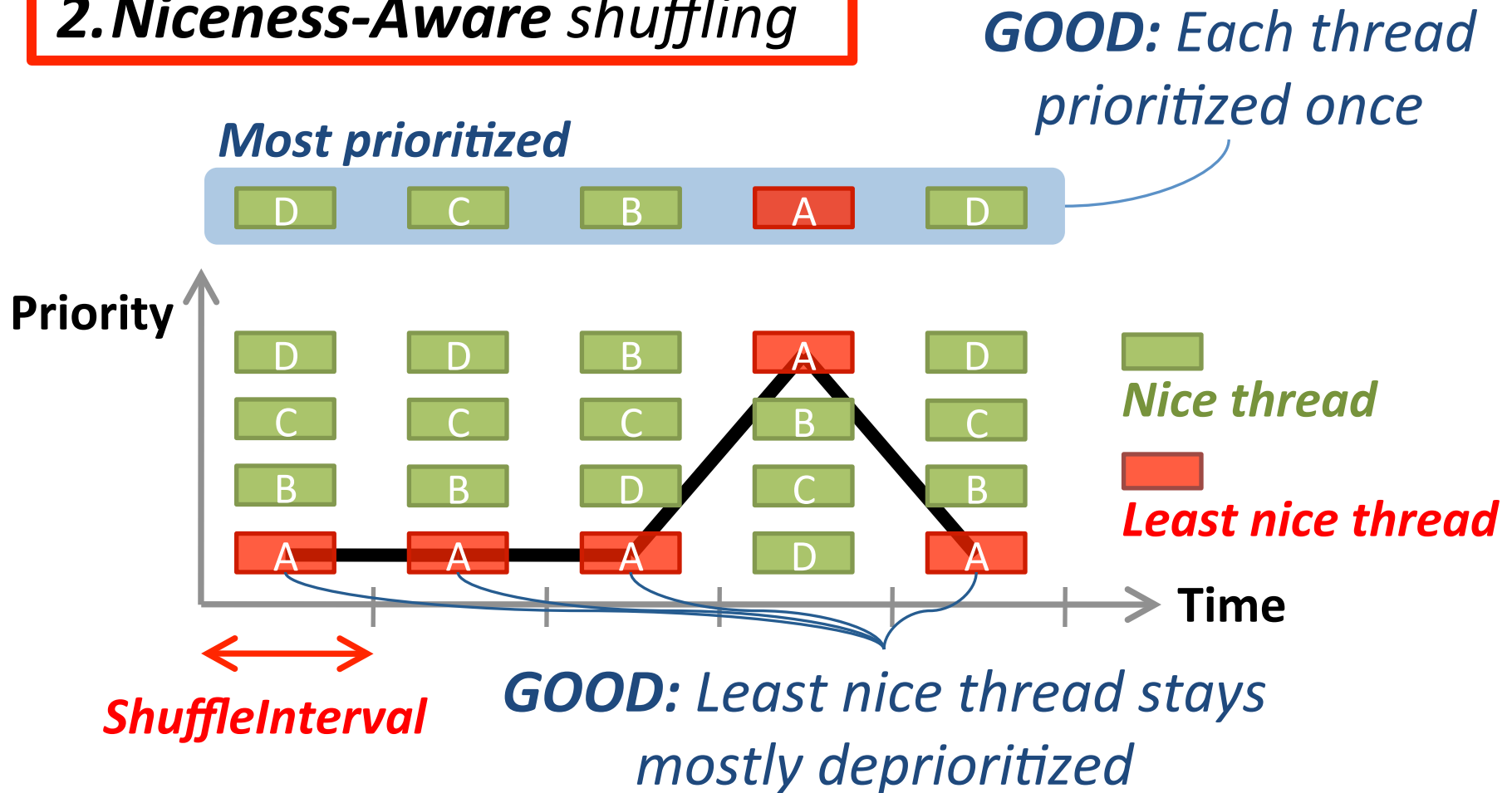
2. Niceness-Aware shuffling



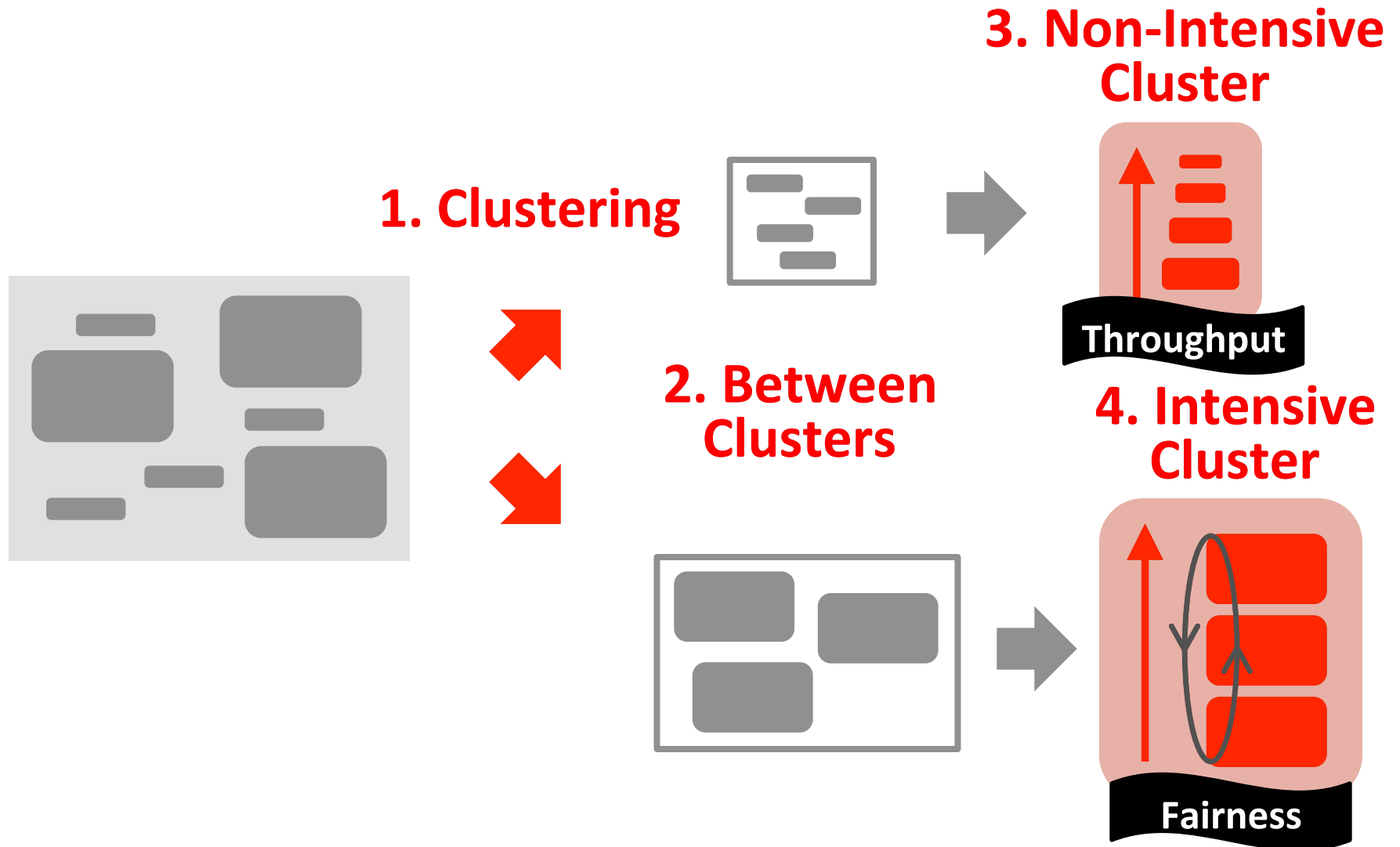
Shuffling: Round-Robin vs. Niceness-Aware

1. Round-Robin shuffling

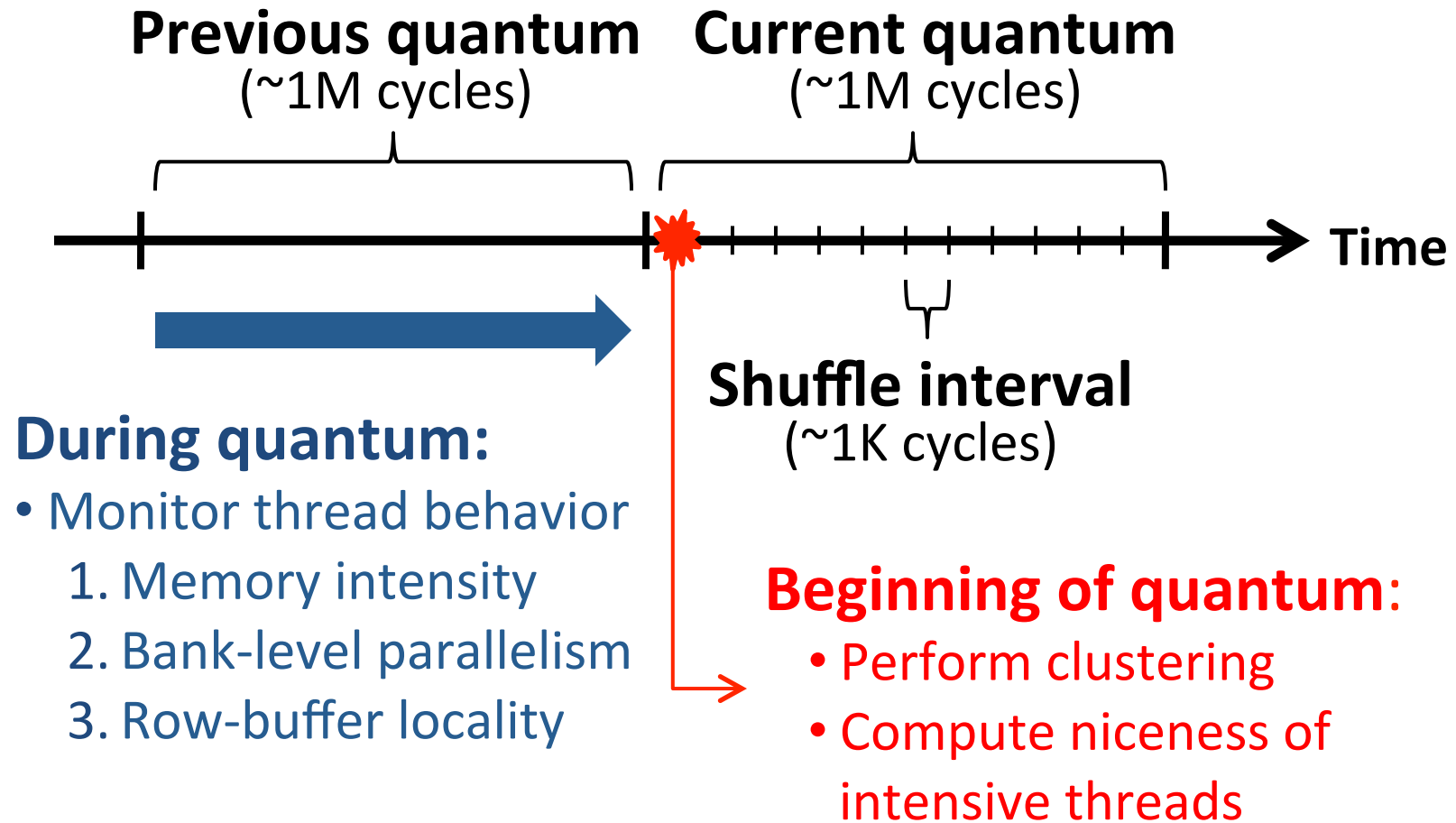
2. Niceness-Aware shuffling



TCM Outline



TCM: Quantum-Based Operation



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

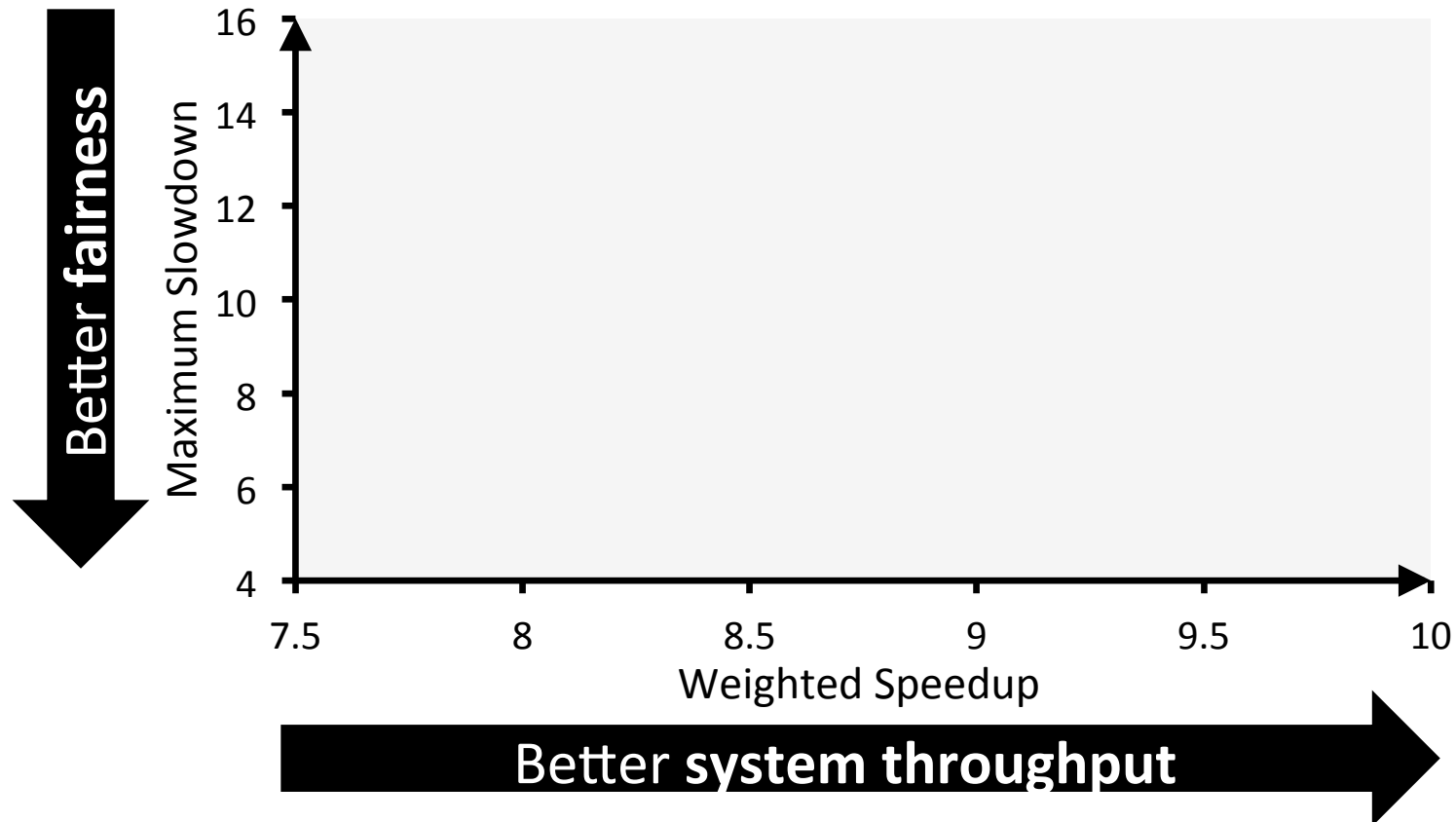
- Non-intensive threads not always prioritized → Low throughput

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

- Most intensive thread starves → Low fairness

TCM: Throughput and Fairness

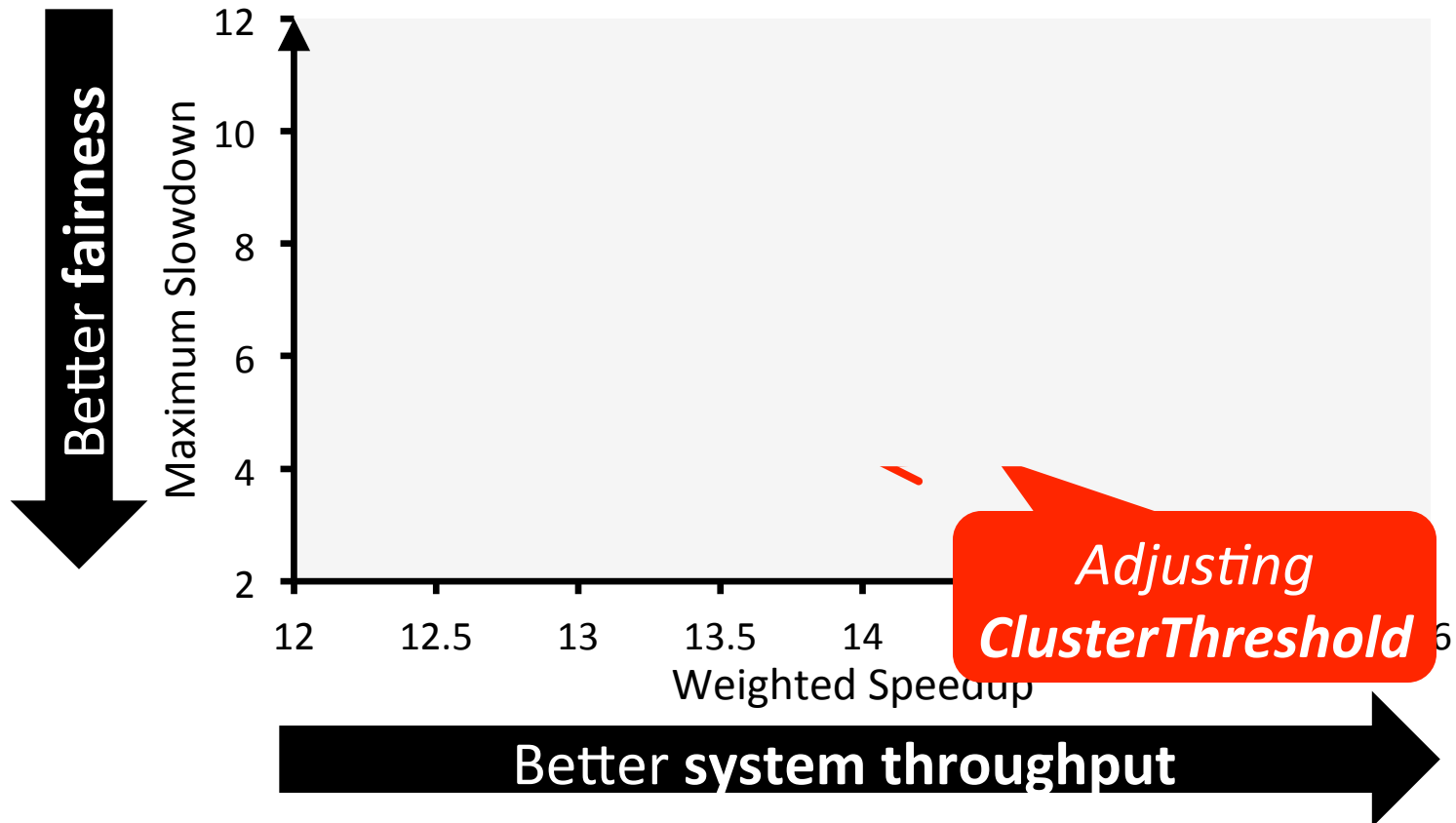
24 cores, 4 memory controllers, 96 workloads



*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.

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
-

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:

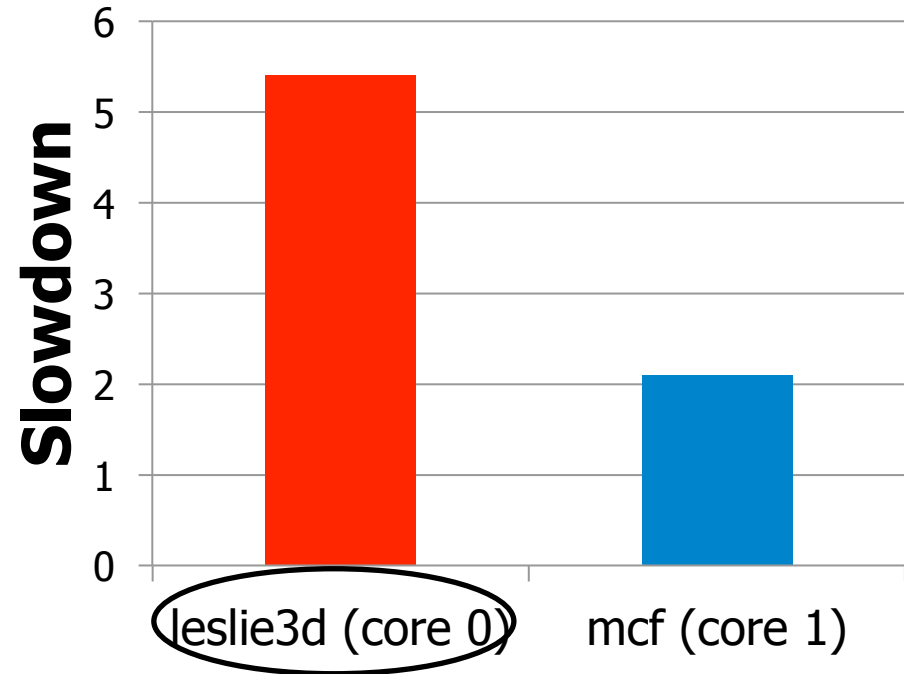
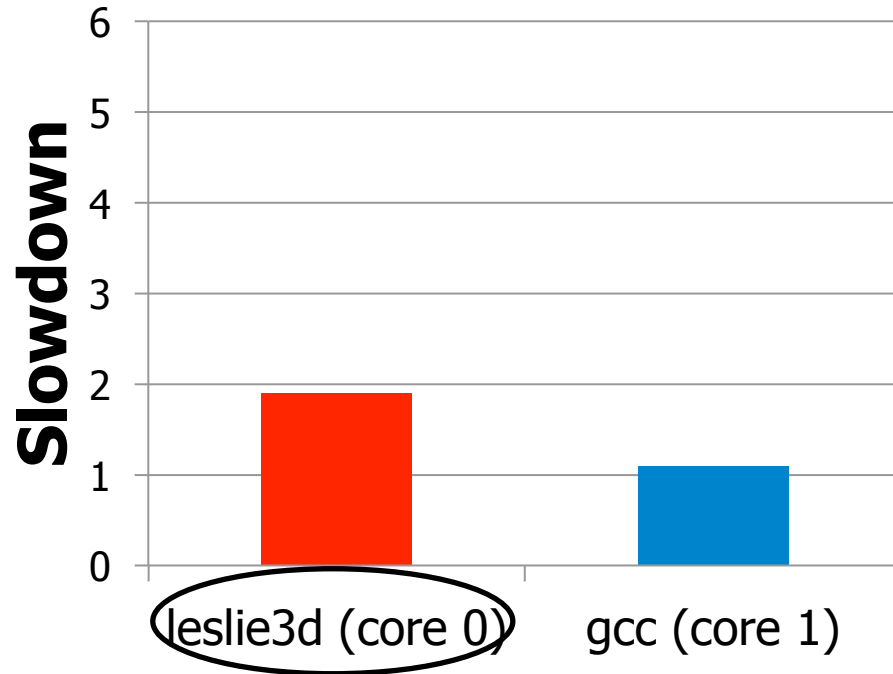
Providing Performance Predictability in Shared Main Memory Systems

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

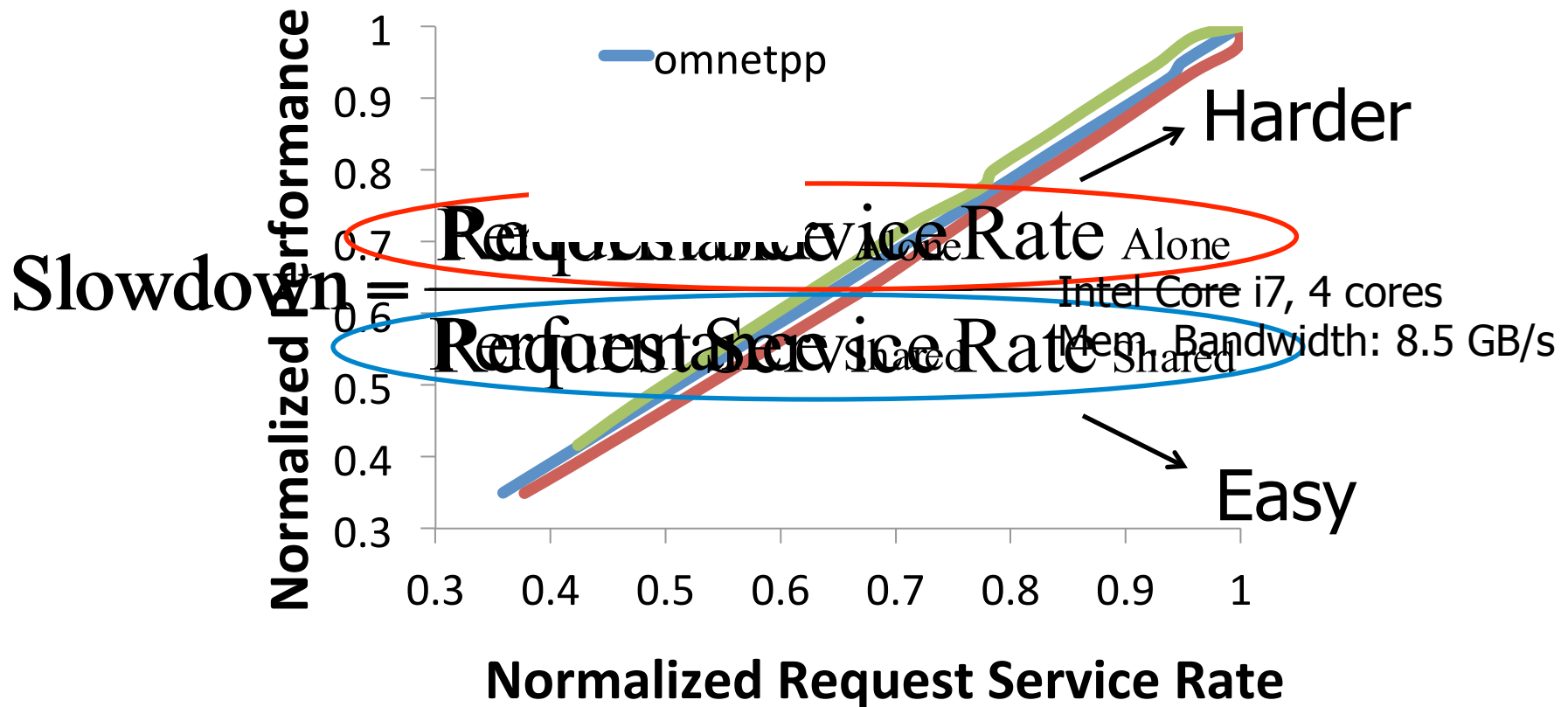
- ❑ Providing Soft Slowdown Guarantees
- ❑ Minimizing Maximum Slowdown

Slowdown: Definition

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

Key Observation 1

For a memory bound application,
Performance \propto Memory request service rate



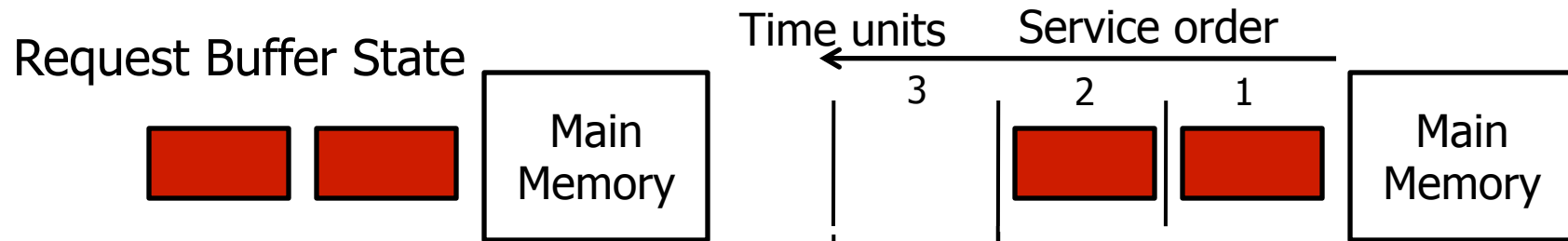
Key Observation 2

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

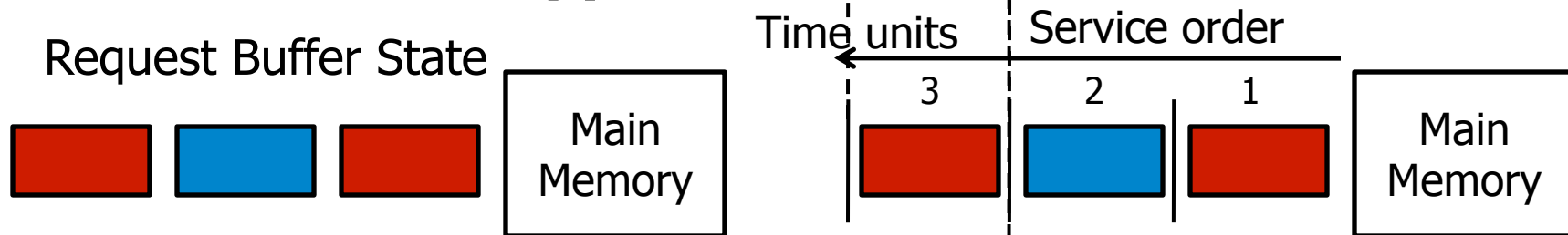
Highest priority → Little interference
(almost as if the application were run alone)

Key Observation 2

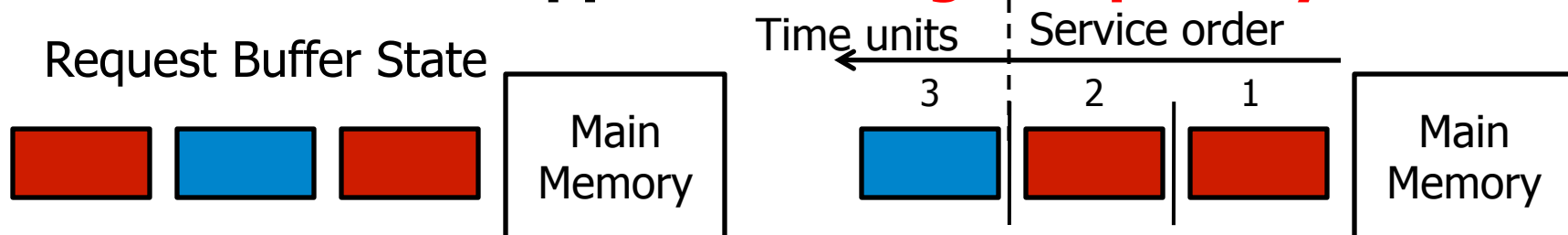
1. Run alone



2. Run with another application



3. Run with another application: **highest priority**

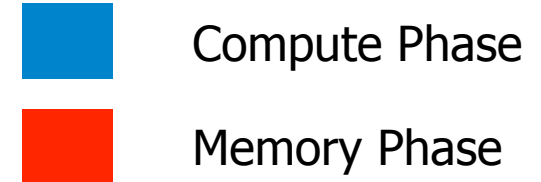


Memory Interference-induced Slowdown Estimation (MISE) model for **memory bound** applications

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

Key Observation 3

- Memory-bound application



Memory phase slowdown dominates overall slowdown

Key Observation 3

■ Non-memory-bound application

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

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

Only memory fraction (α) slows down with interference

Measuring RSR_{Shared} and α

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

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

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

Estimating Request Service Rate $_{\text{Alone}}$ ($\text{RSR}_{\text{Alone}}$)

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

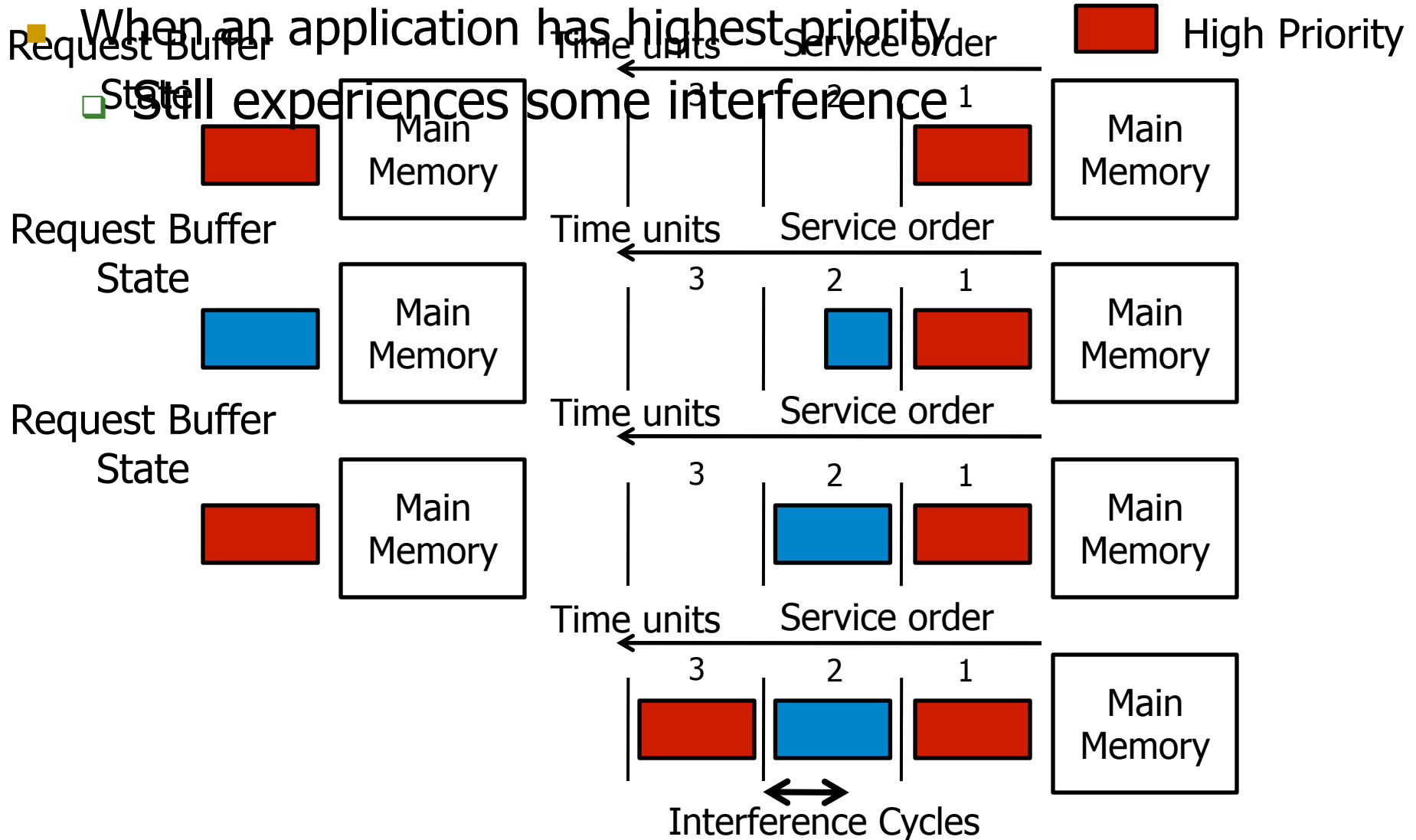
Goal: Estimate $\text{RSR}_{\text{Alone}}$

How: Periodically give each application

- At the end of an interval, for each application, estimate highest priority in accessing memory

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

Inaccuracy in Estimating RSR_{Alone}



Accounting for Interference in RSR_{Alone} Estimation

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

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

- 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

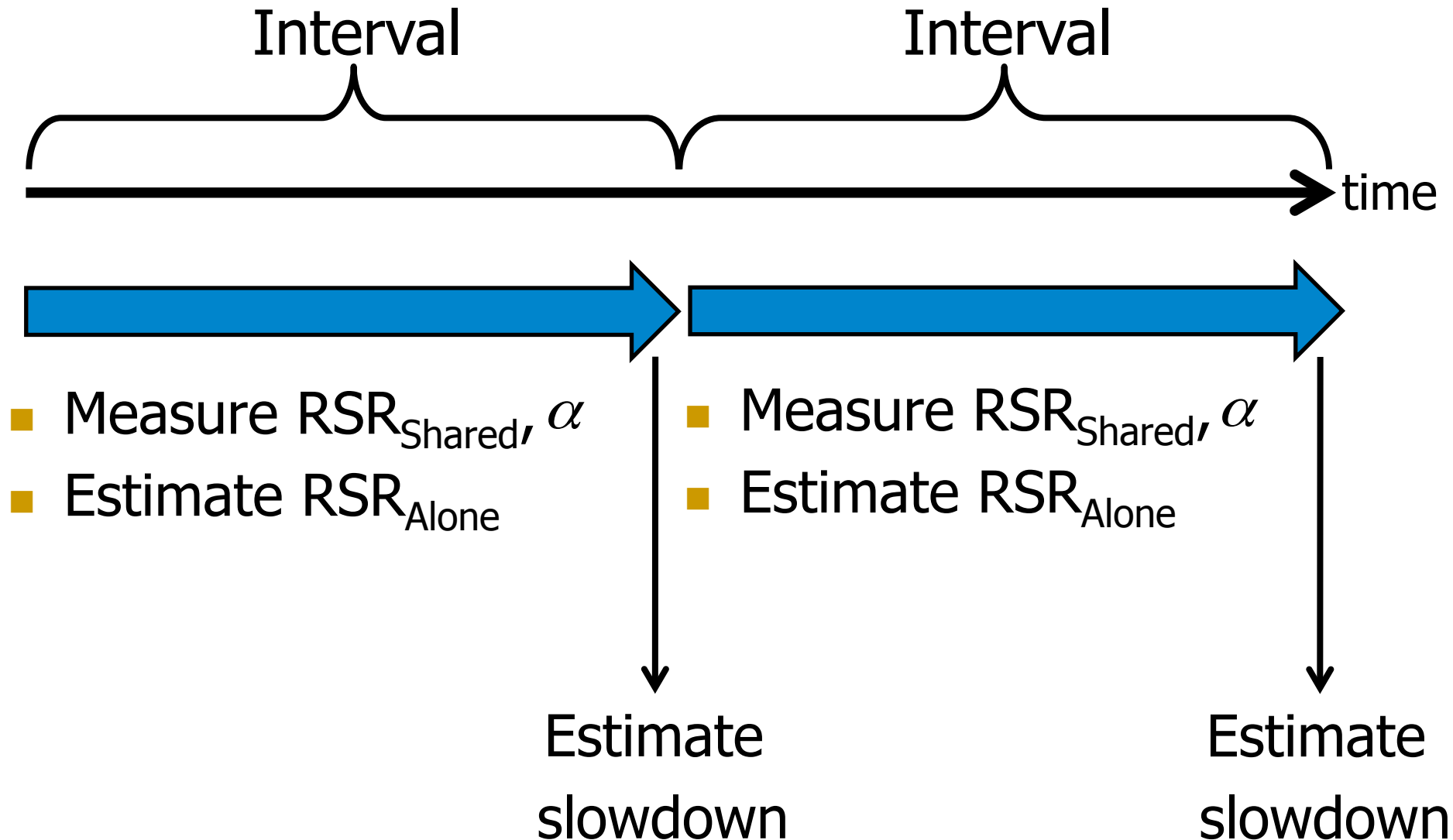
1. Estimate Slowdown

- ❑ 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:

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

Diagram illustrating the components of the slowdown formula:

- The numerator, **Stall Time Alone**, is circled and labeled **Hard** (indicating a difficult or high-cost scenario).
- The denominator, **Stall Time Shared**, is labeled **Easy** (indicating an easier or lower-cost scenario).

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

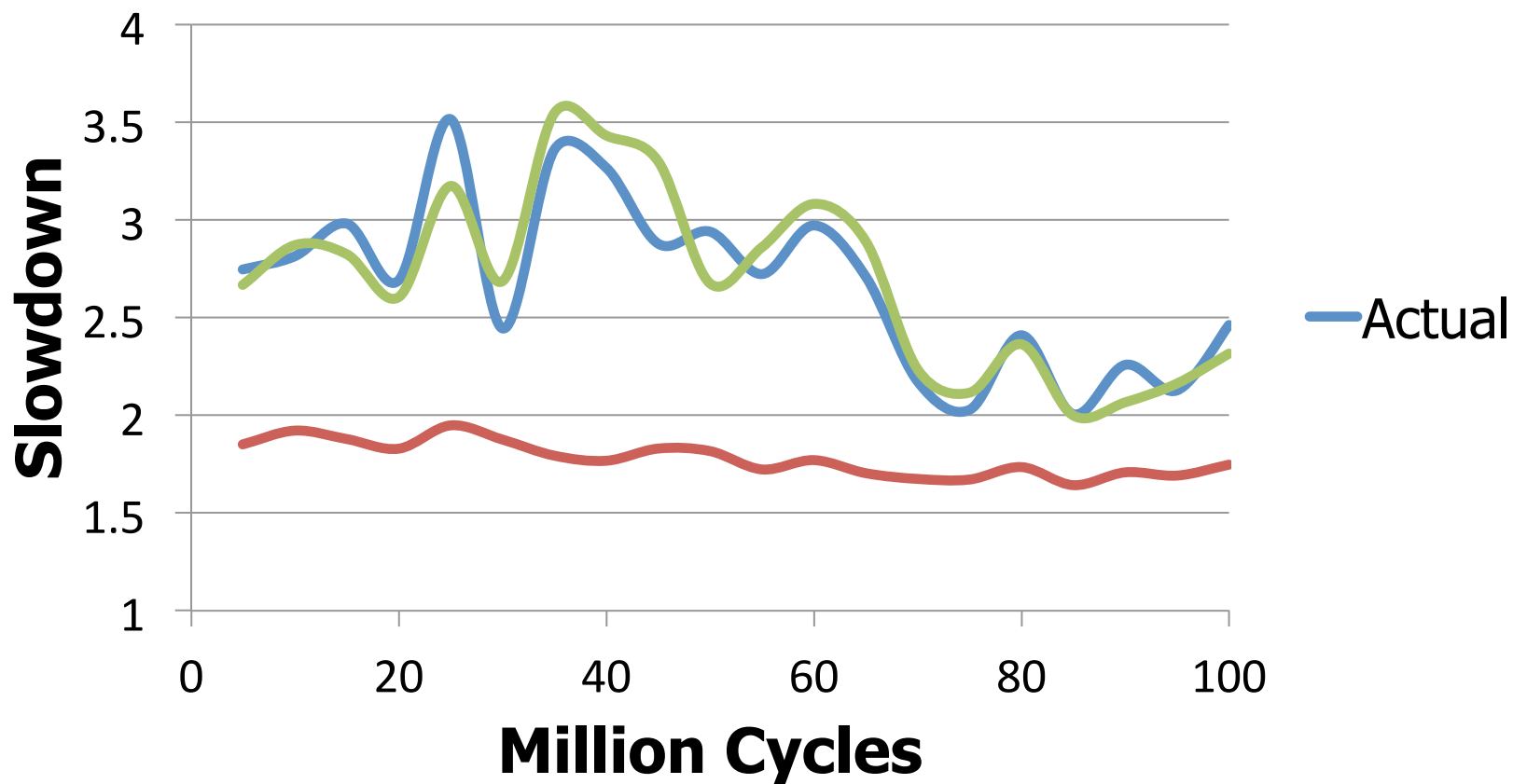
Methodology

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

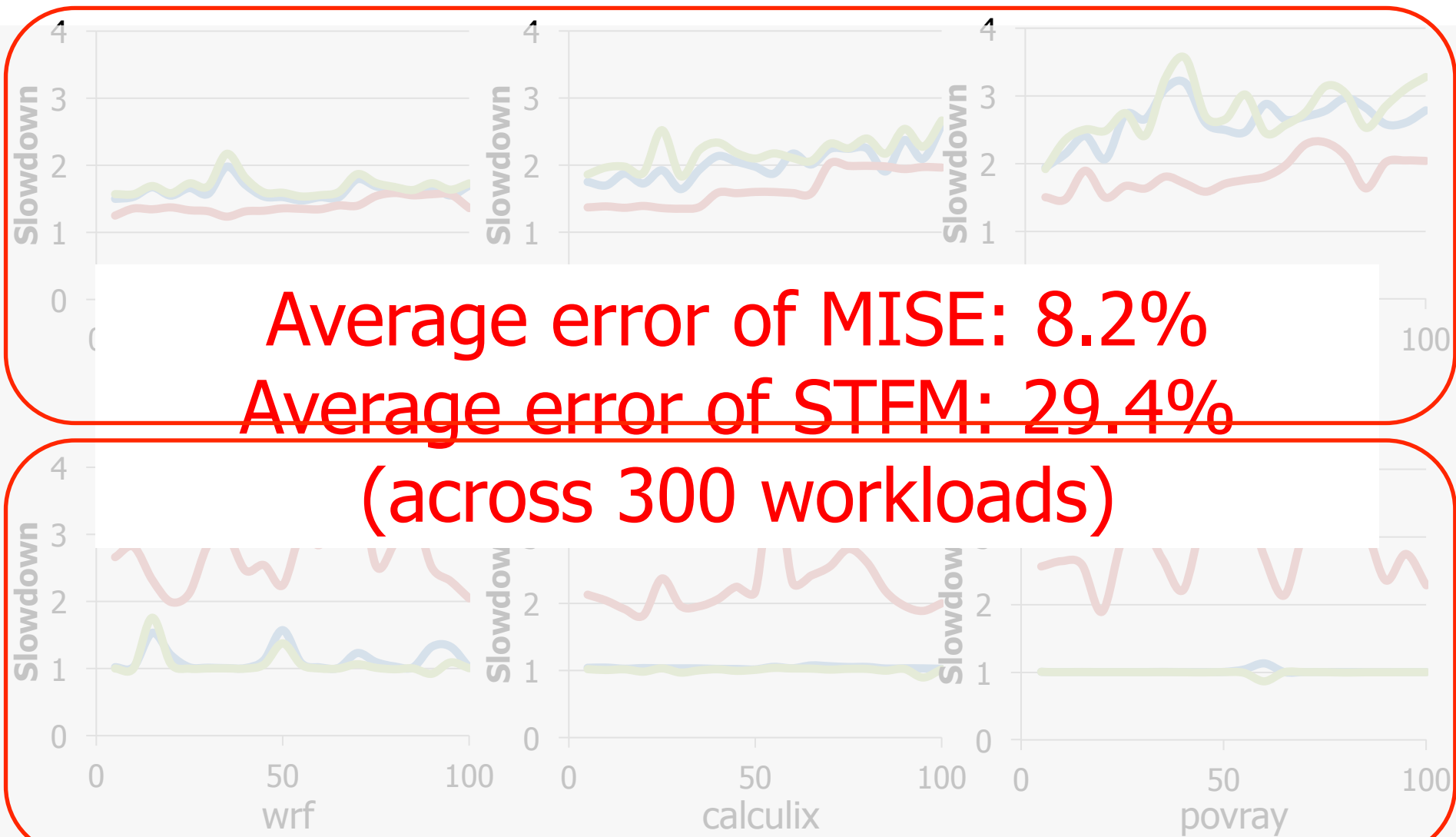
- Workloads
 - ❑ SPEC CPU2006
 - ❑ 300 multi programmed workloads

Quantitative Comparison

SPEC CPU 2006 application
leslie3d



Comparison to STFM



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

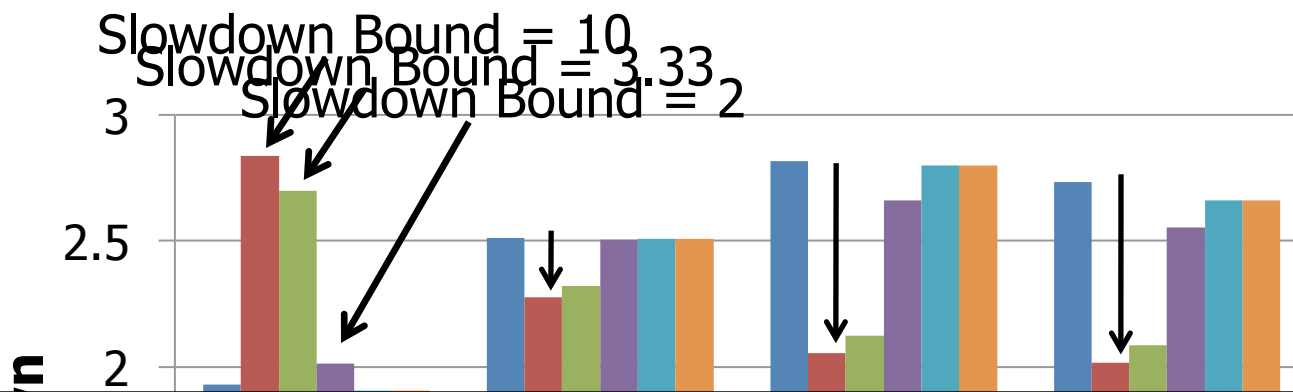
MISE-QoS: Mechanism to Provide Soft QoS

- 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 $B \pm \epsilon$, increase bandwidth allocation
 - If slowdown $<$ bound $B \pm \epsilon$, decrease bandwidth allocation
- When slowdown bound not met for N intervals
 - Notify the OS so it can migrate/de-schedule jobs

Methodology

- 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

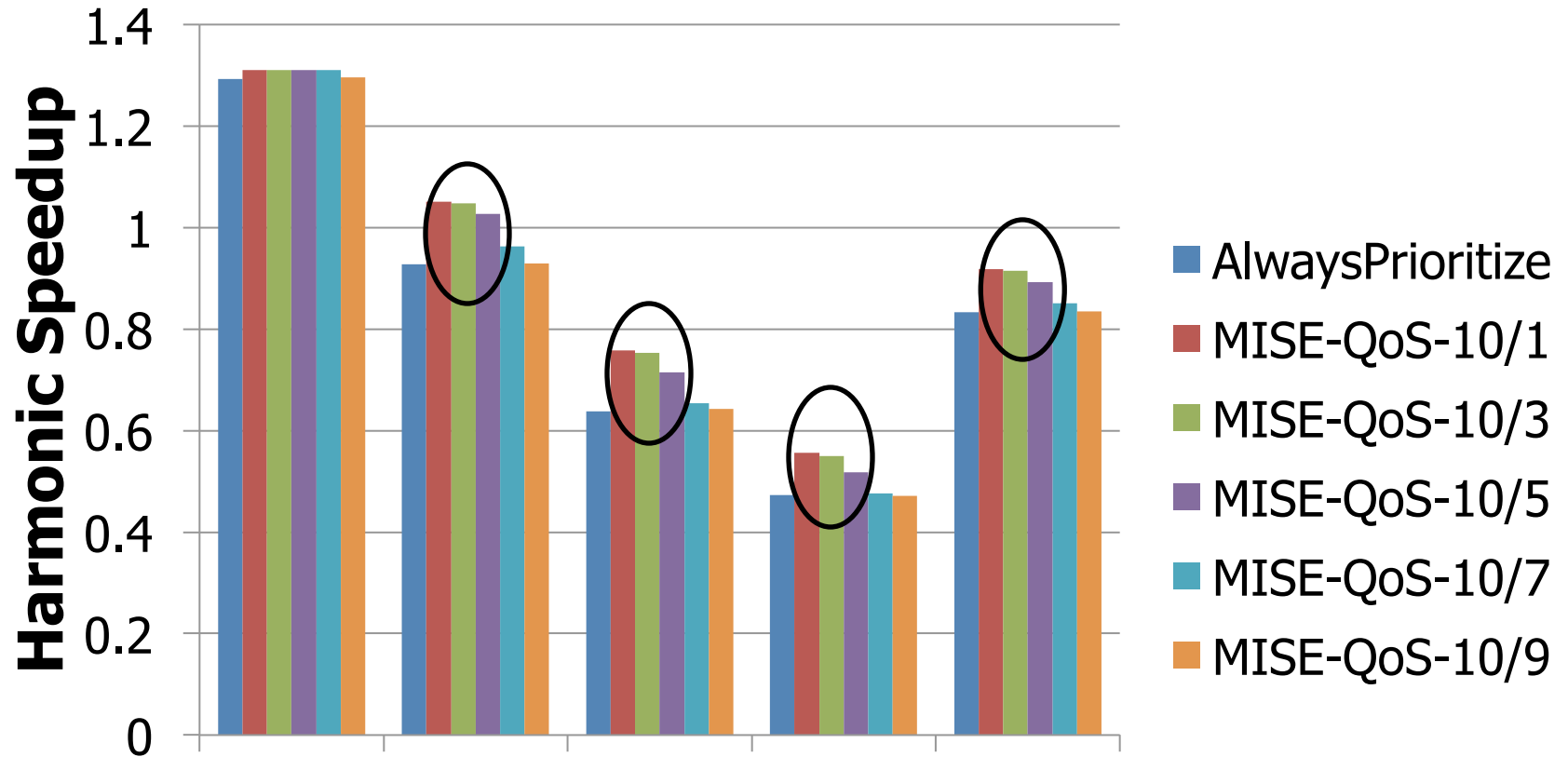
Effectiveness of MISE in Enforcing QoS

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	t_{RP}	11	Activate to read/write	t_{RCD}	11
Read column address strobe	CL	11	Write column address strobe	CWL	8
Additive	AL	0	Activate to activate	t_{RC}	39
Activate to precharge	t_{RAS}	28	Read to precharge	t_{RTP}	6
Burst length	t_{BL}	4	Column address strobe to column address strobe	t_{CCD}	4
Activate to activate (different bank)	t_{RRD}	6	Four activate windows	t_{FAW}	24
Write to read	t_{WTR}	6	Write recovery	t_{WR}	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.

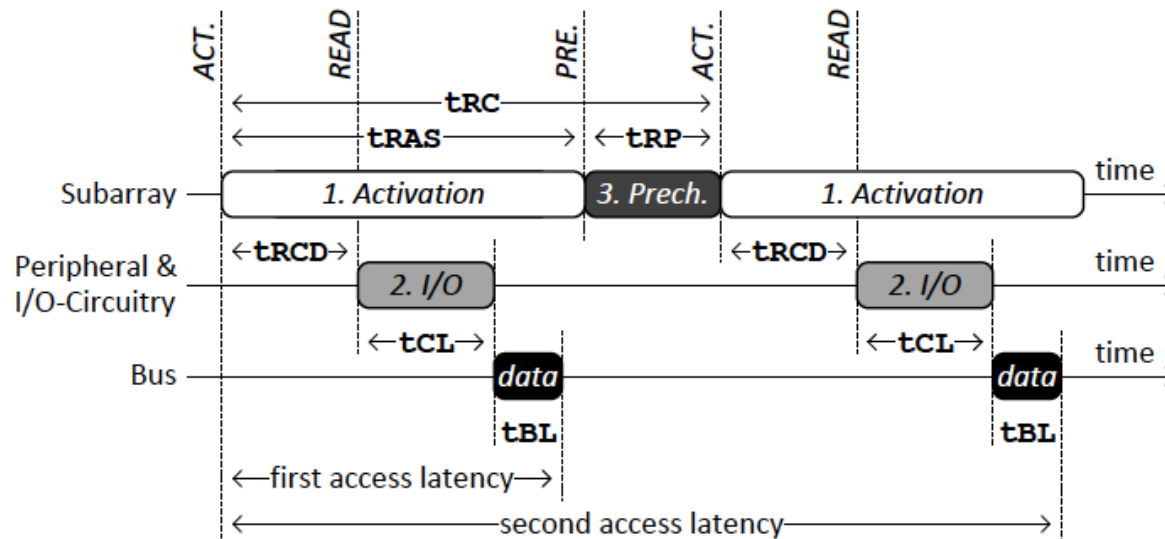


Figure 5. Three Phases of DRAM Access

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

Phase	Commands	Name	Value
1	ACT → READ	tRCD	15ns
	ACT → WRITE		
	ACT → PRE	tRAS	37.5ns
2	READ → data	tCL	15ns
	WRITE → data	tCWL	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

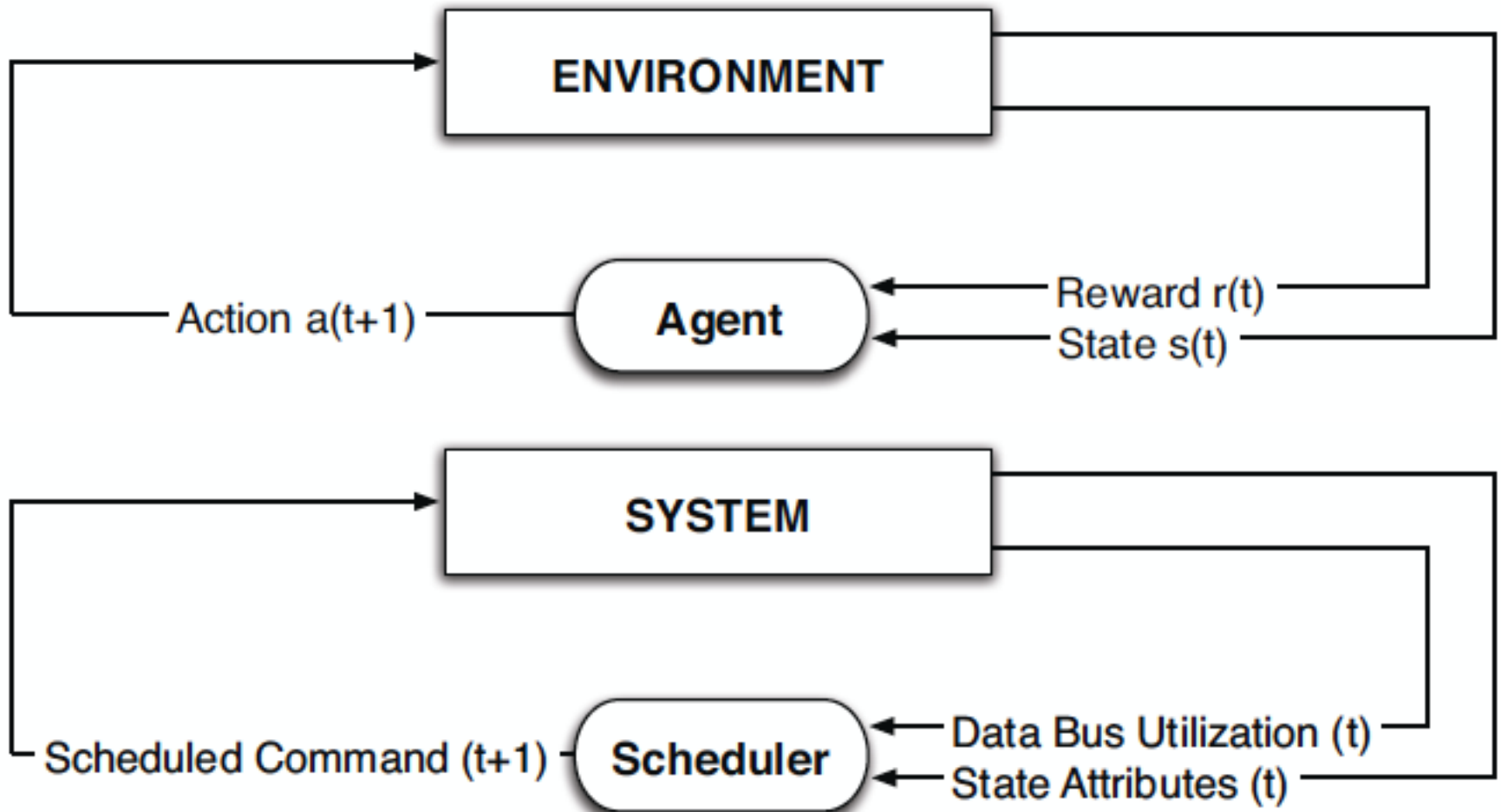


Figure 2: (a) Intelligent agent based on reinforcement learning principles; (b) DRAM scheduler as an RL-agent

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.

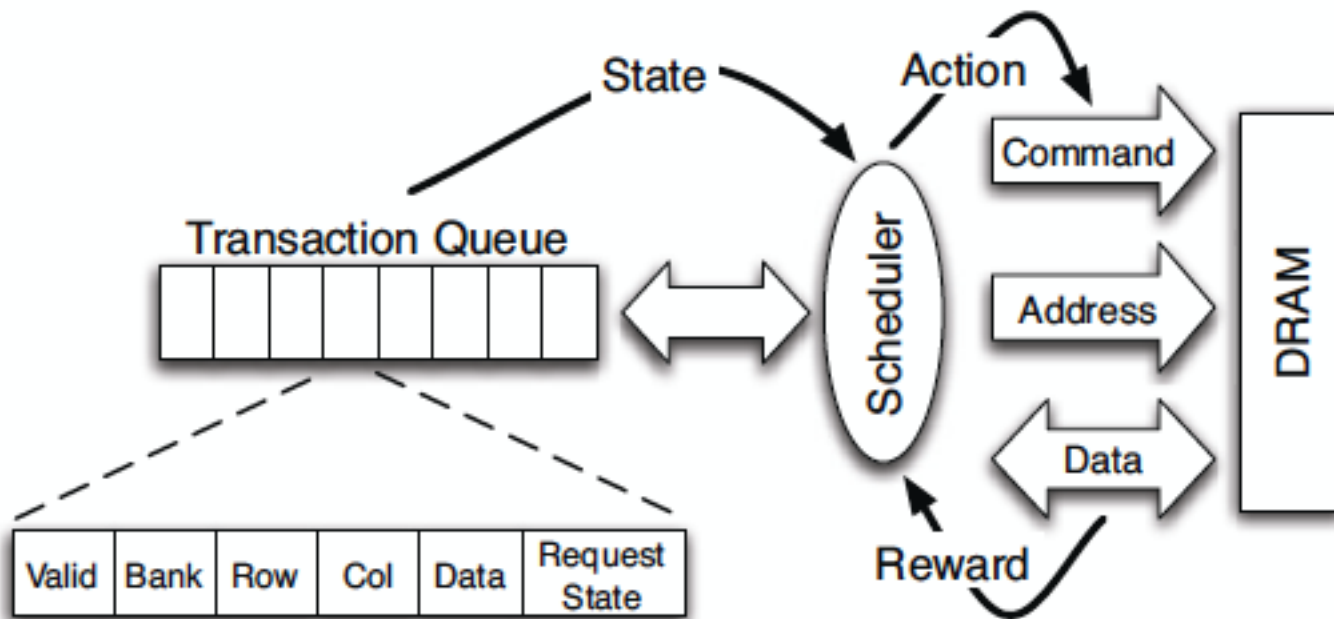


Figure 4: High-level overview of an RL-based scheduler.

Performance Results

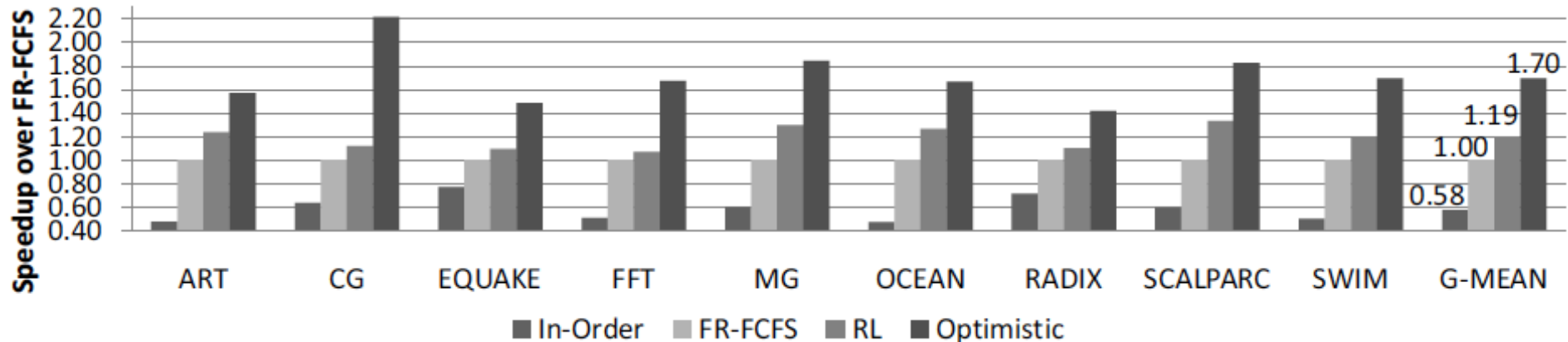


Figure 7: Performance comparison of in-order, FR-FCFS, RL-based, and optimistic memory controllers

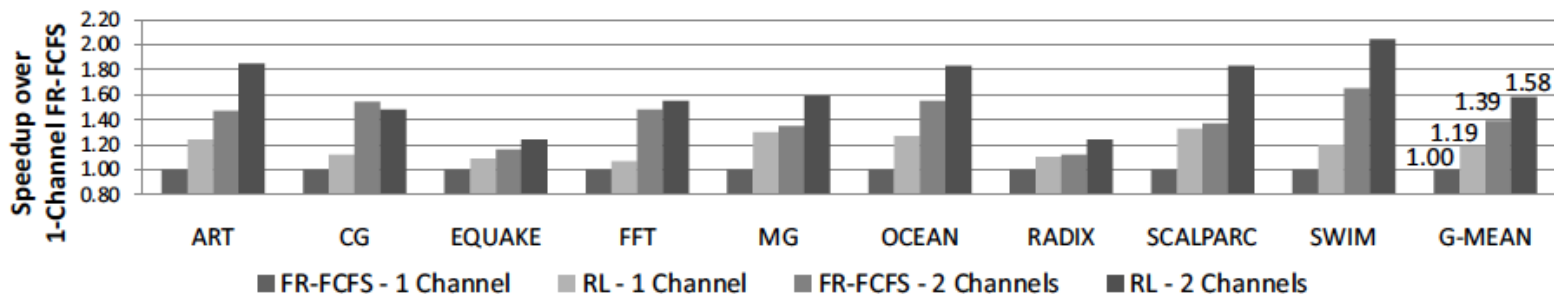


Figure 15: Performance comparison of FR-FCFS and RL-based memory controllers on systems with 6.4GB/s and 12.8GB/s peak DRAM bandwidth

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]

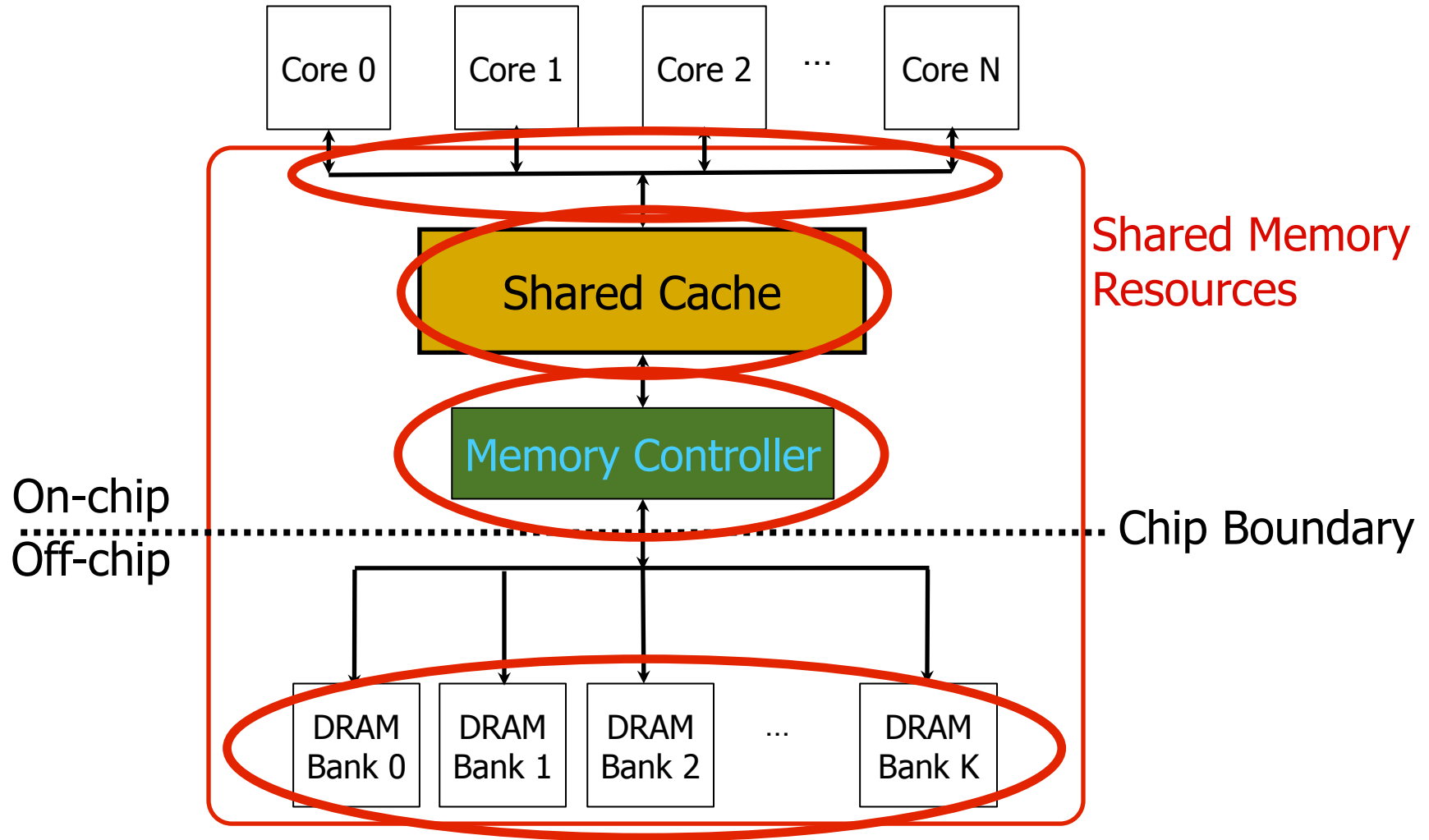
Fairness via Source Throttling

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)

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

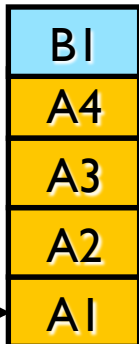
- Manage inter-thread interference at the **cores**, **not** at the **shared resources**
- **Dynamically estimate unfairness** in the memory system
- Feed back this information into a controller
- **Throttle cores' memory access rates** accordingly
 - Whom to throttle and by how much depends on performance 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



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

queue of requests to shared resources

Request Generation Order

A1, ~~B1~~, ~~A2~~, ~~A3~~, ~~A4~~, B1

Throttled Requests

Fair Source Throttling



A: Compute Stall on A1 Stall wait. Stall on A2 Stall on A3 Stall on A4

B: Compute Stall wait. Stall on B1

Core A's stall time

Core B's stall time

Extra Cycles Core A

Saved Cycles Core B

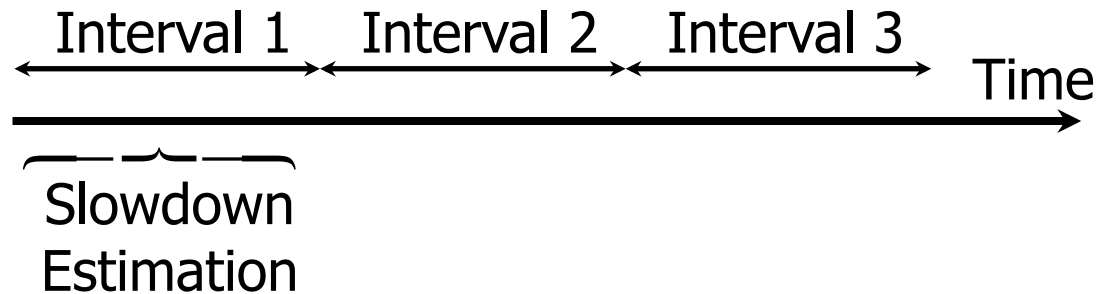
Shared Memory Resources

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

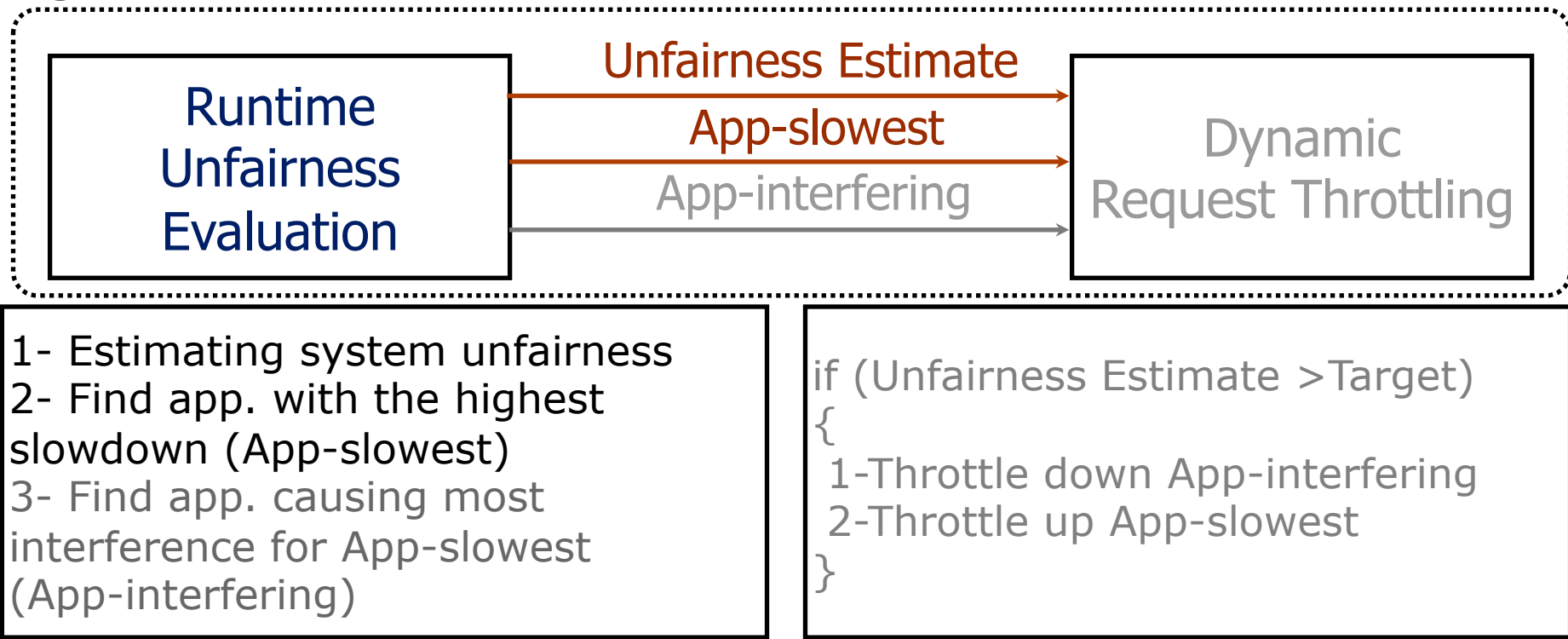


- 1- Estimating system unfairness
- 2- Find app. with the highest slowdown (App-slowest)
- 3- Find app. causing most interference for App-slowest (App-interfering)

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

Fairness via Source Throttling (FST)

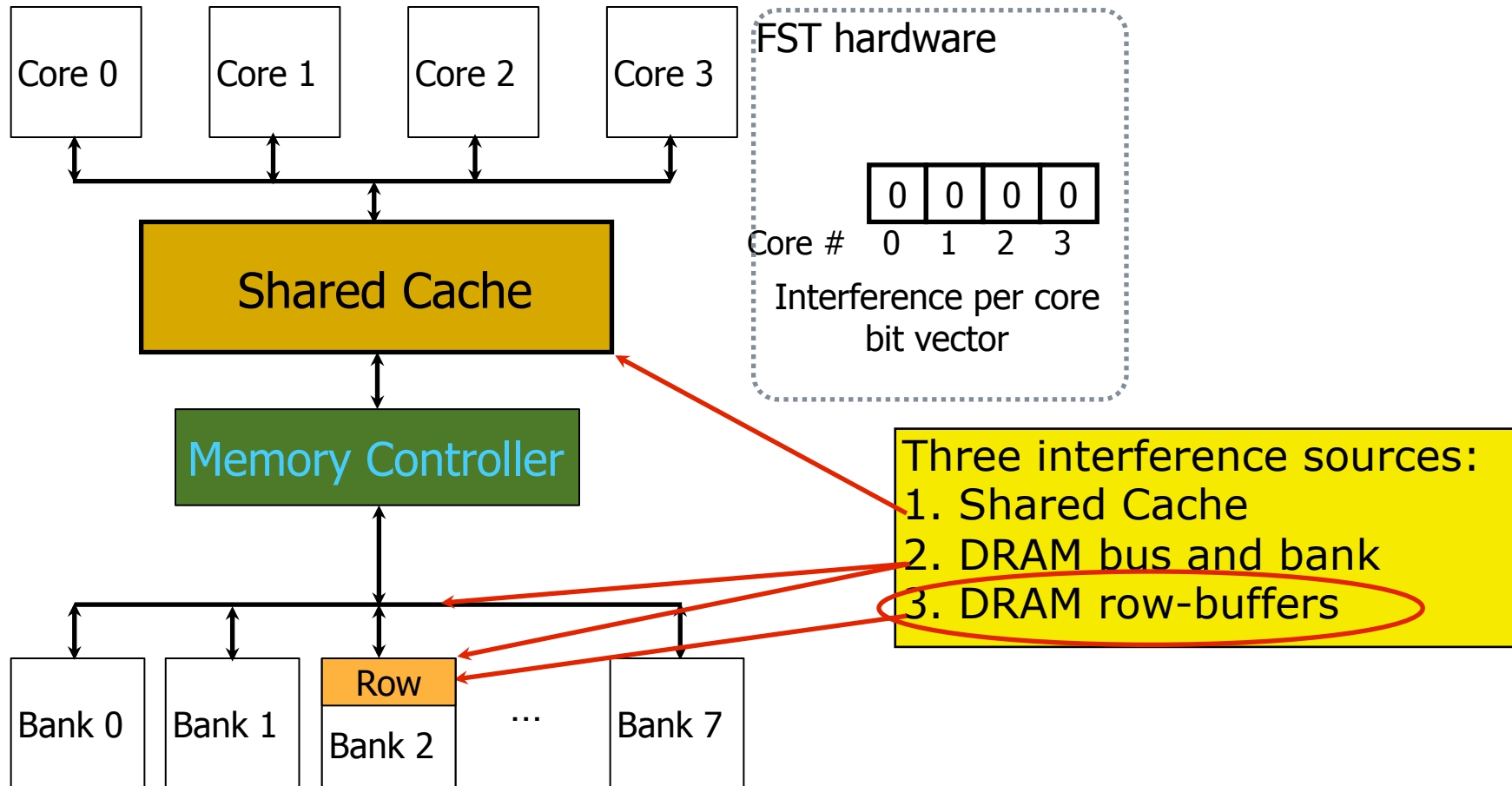
FST



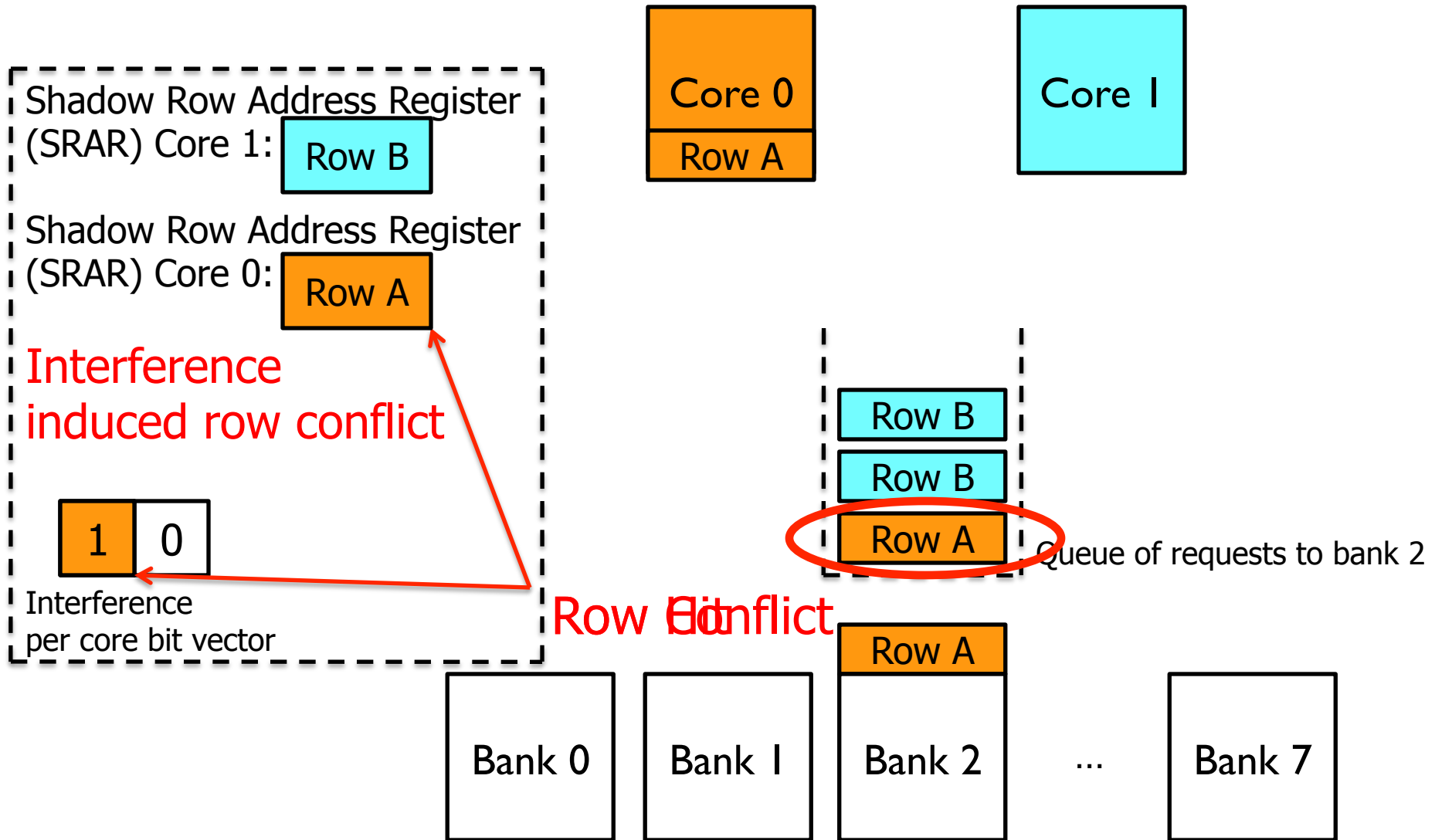
Estimating System Unfairness

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

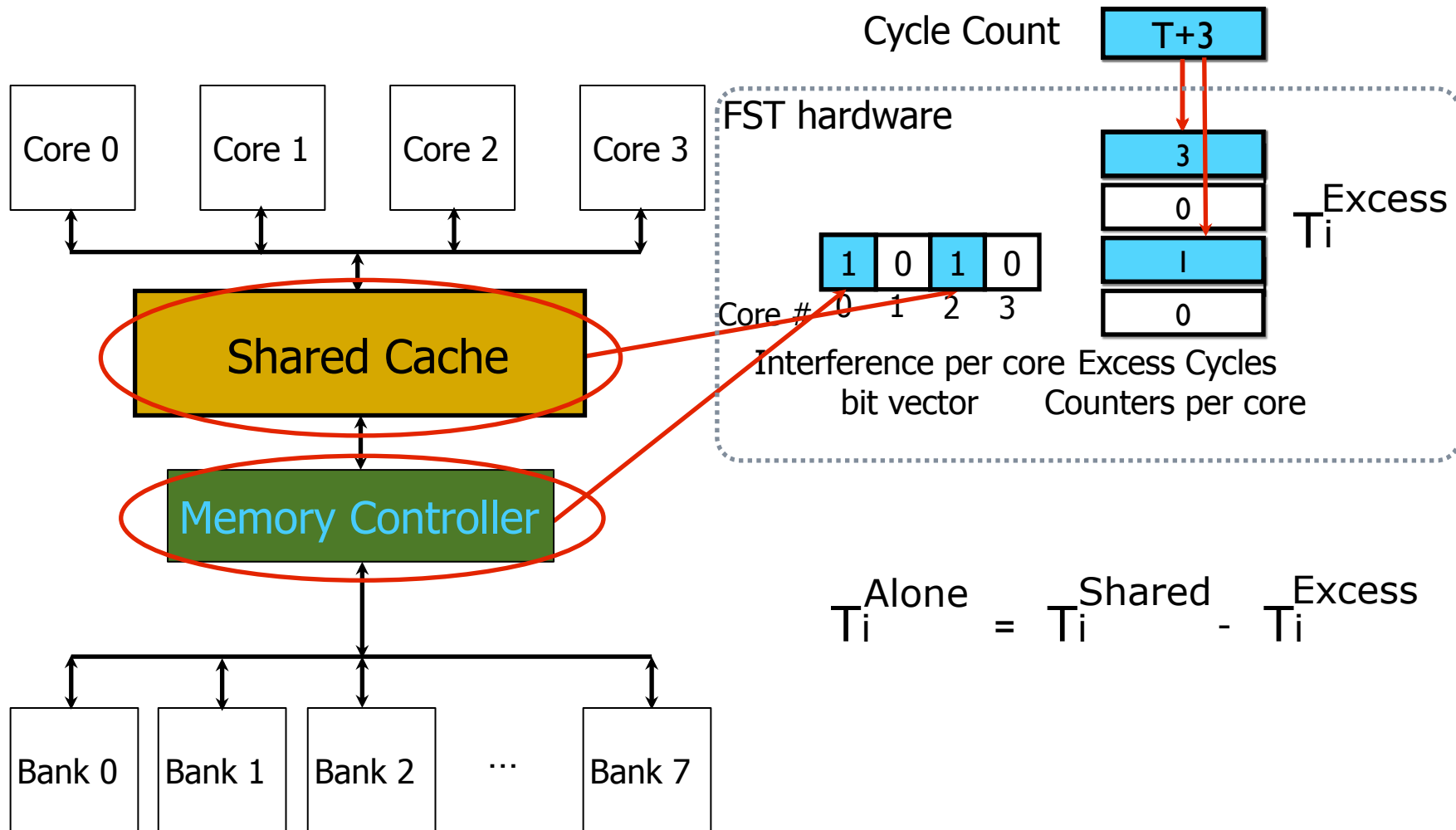
Tracking Inter-Core Interference



Tracking DRAM Row-Buffer Interference

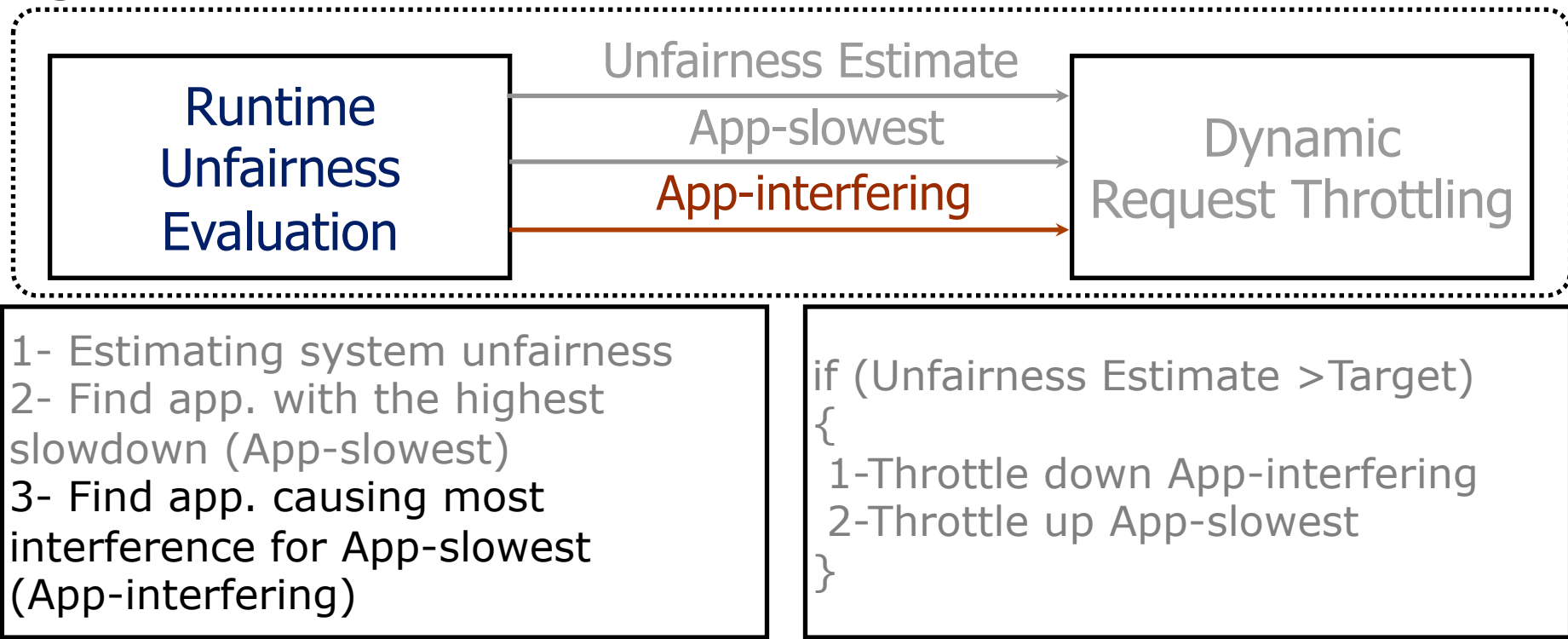


Tracking Inter-Core Interference



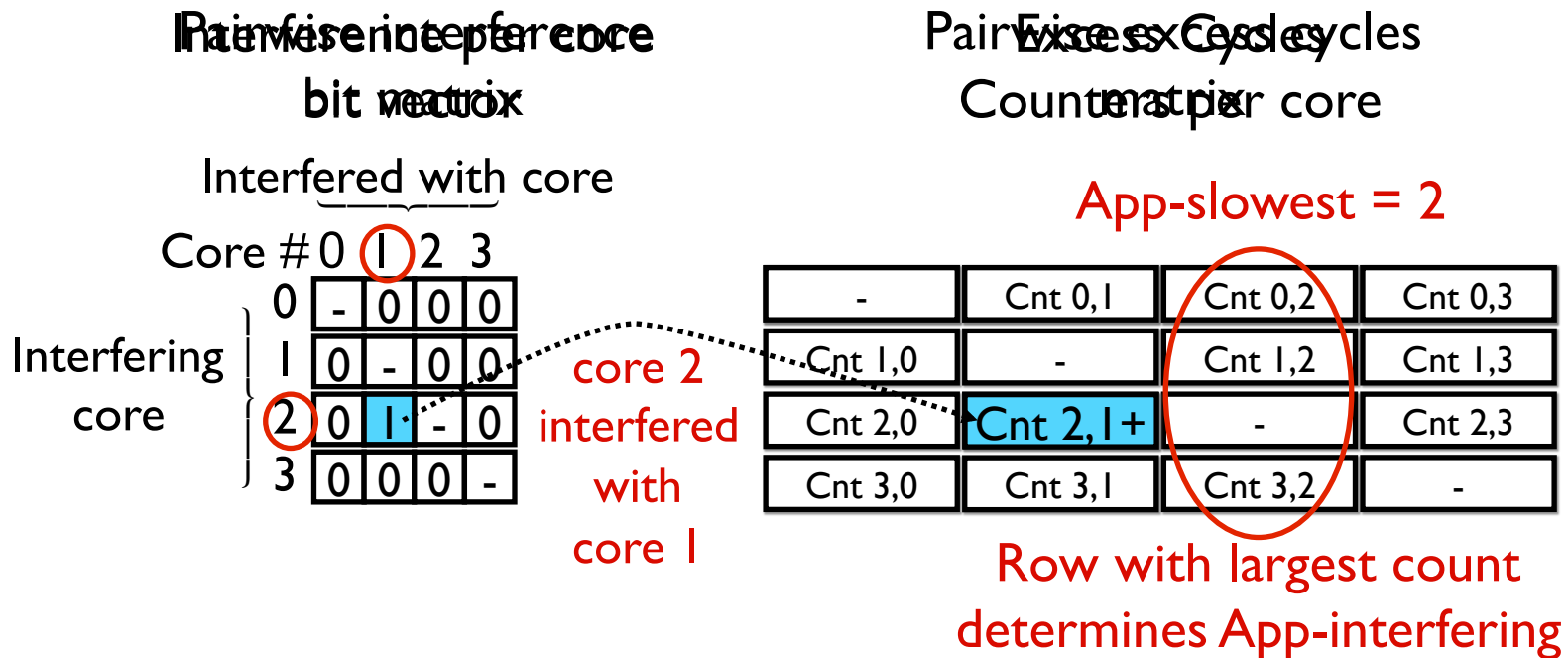
Fairness via Source Throttling (FST)

FST



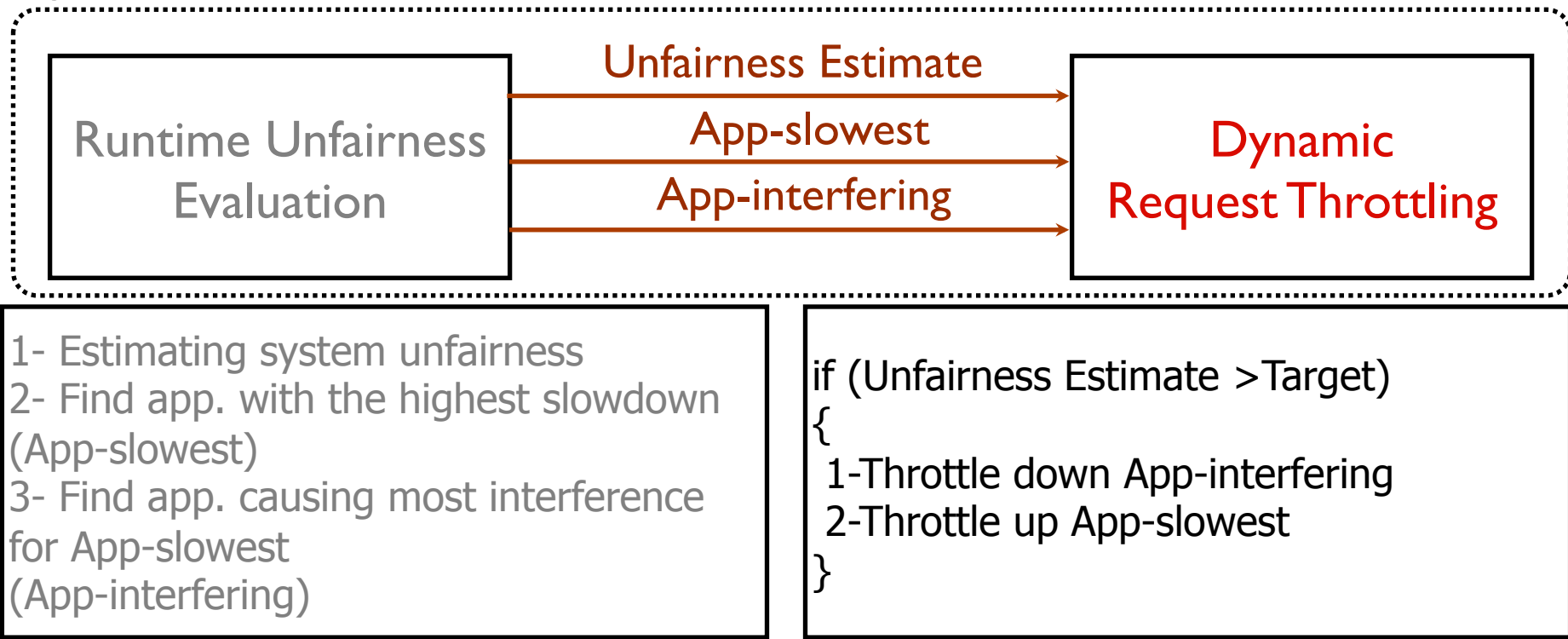
Tracking Inter-Core Interference

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



Fairness via Source Throttling (FST)

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

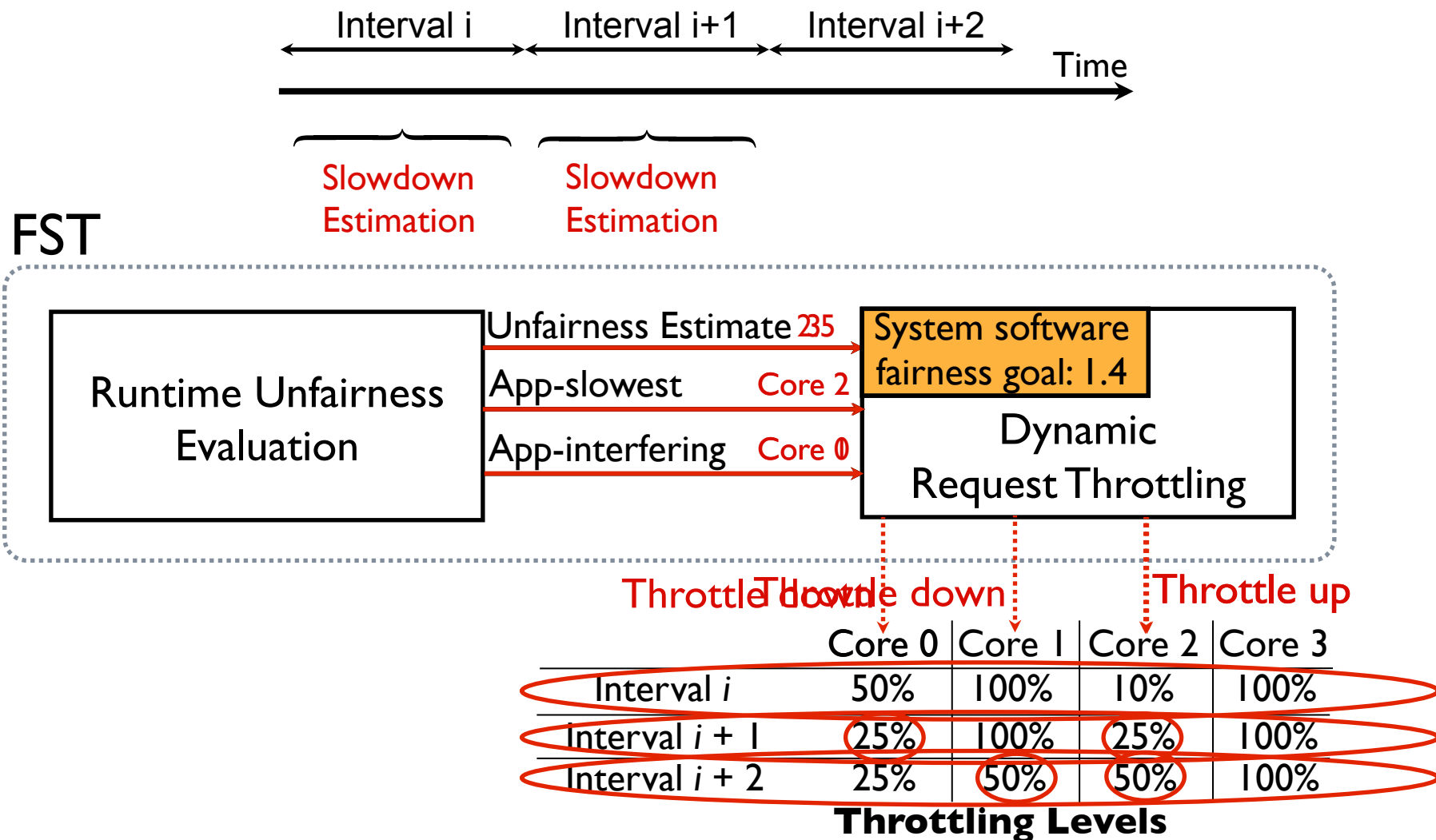
Dynamic Request Throttling

- **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



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
 - $\text{Weighted Slowdown}(i) = \text{Estimated Slowdown}(i) \times \text{Weight}(i)$

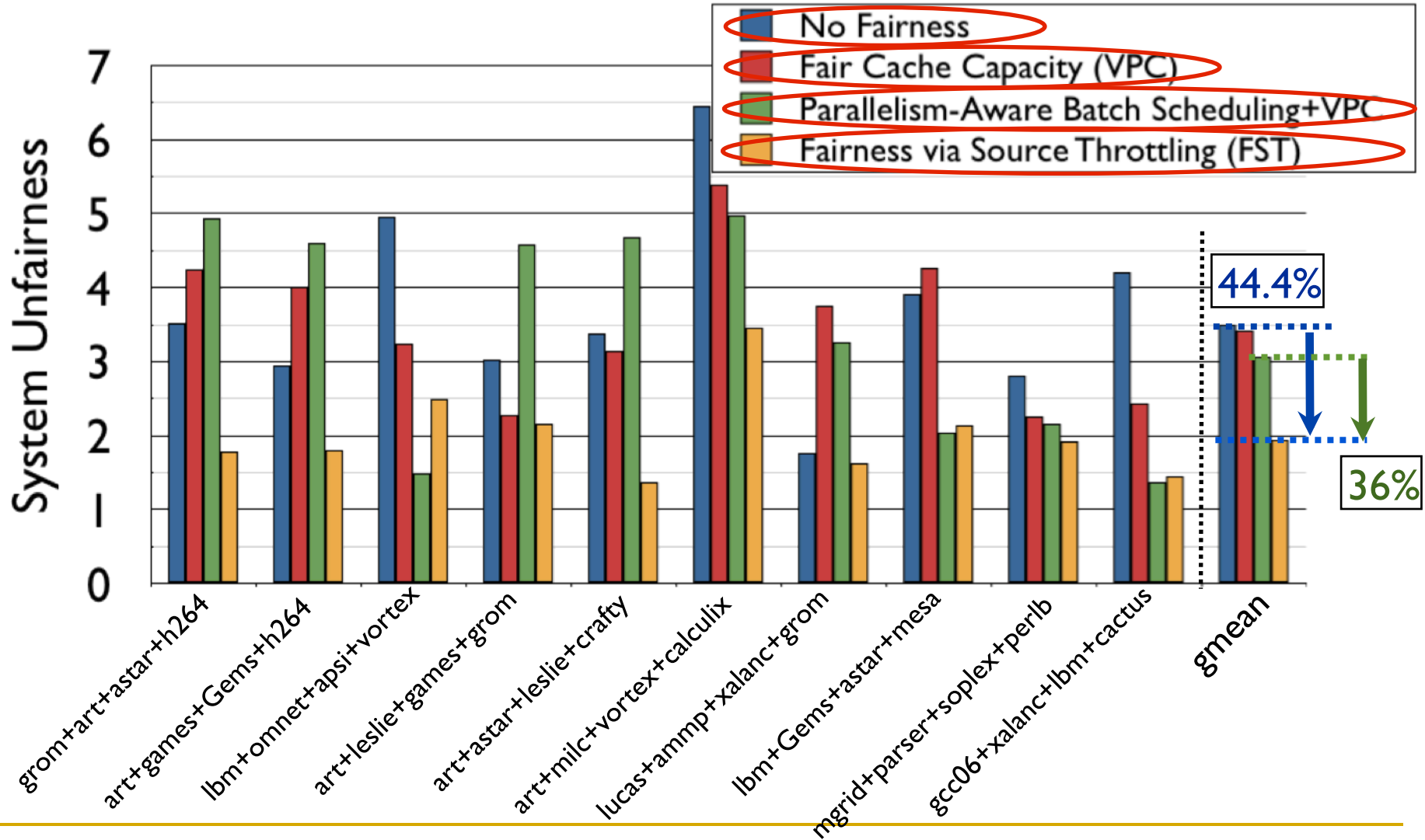
FST Hardware Cost

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

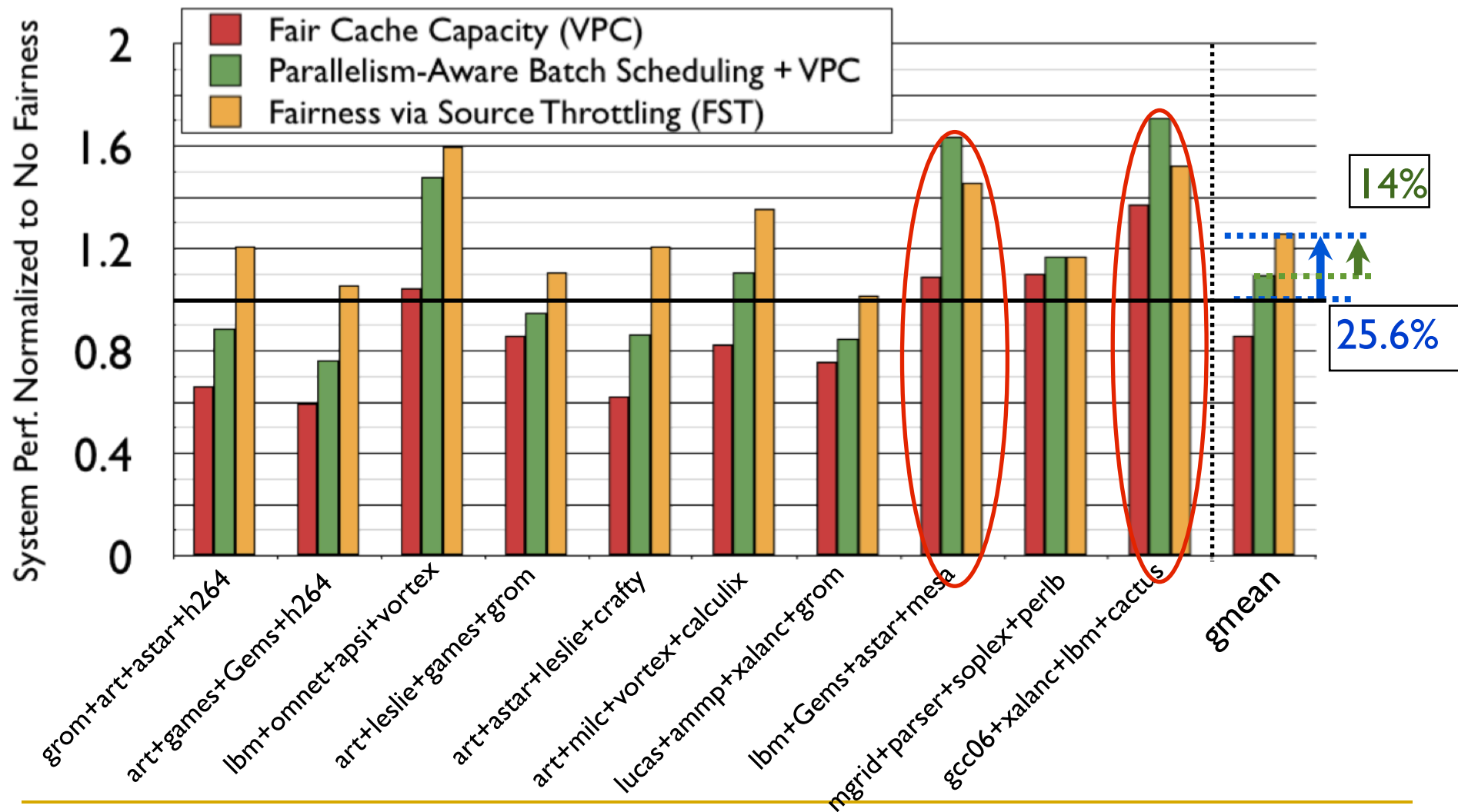
FST Evaluation Methodology

- 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

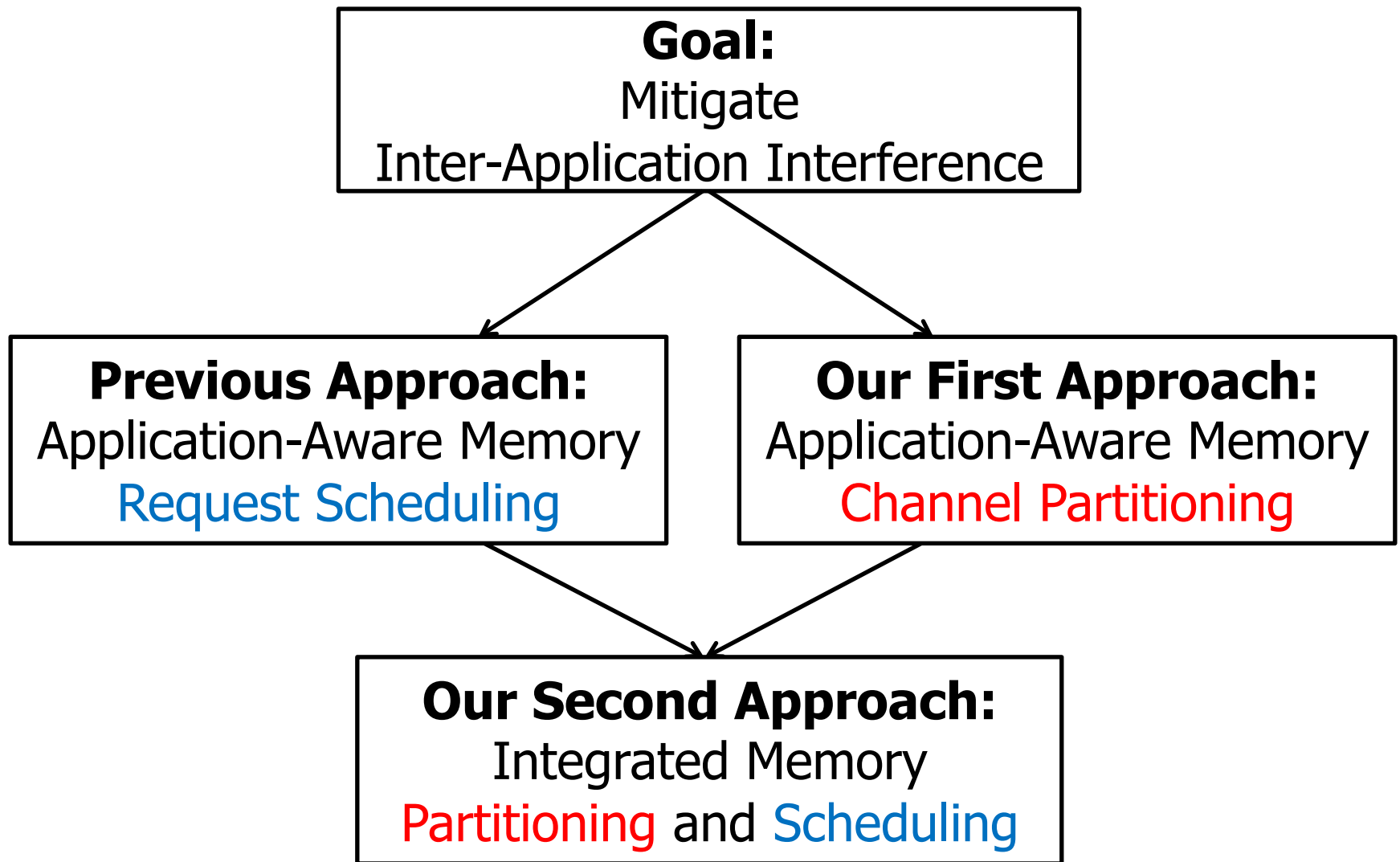
Designing QoS-Aware Memory Systems: Approaches

- **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)

Outline



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

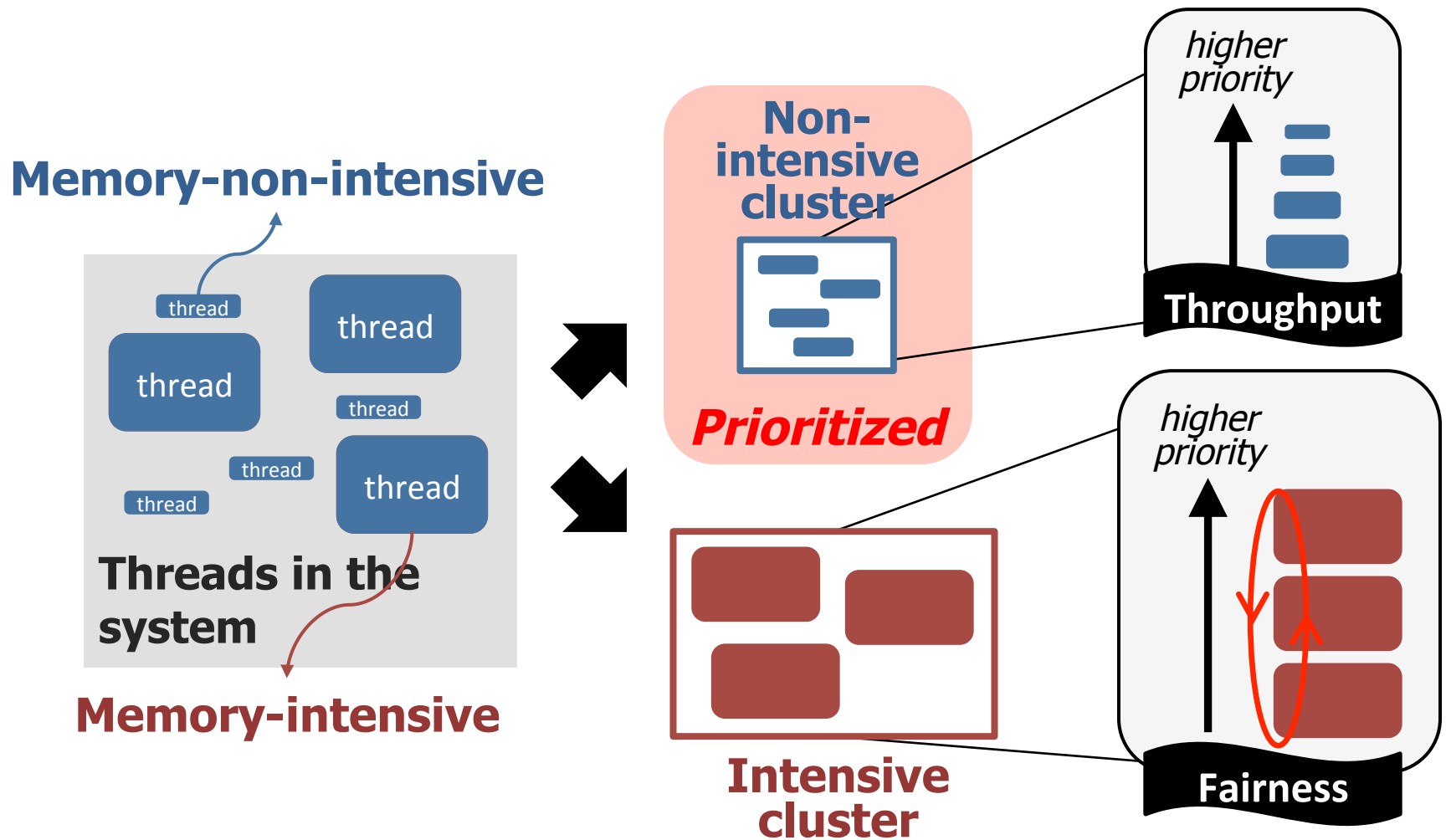


Figure: Kim et al., MICRO 2010

Application-Aware Memory Request Scheduling

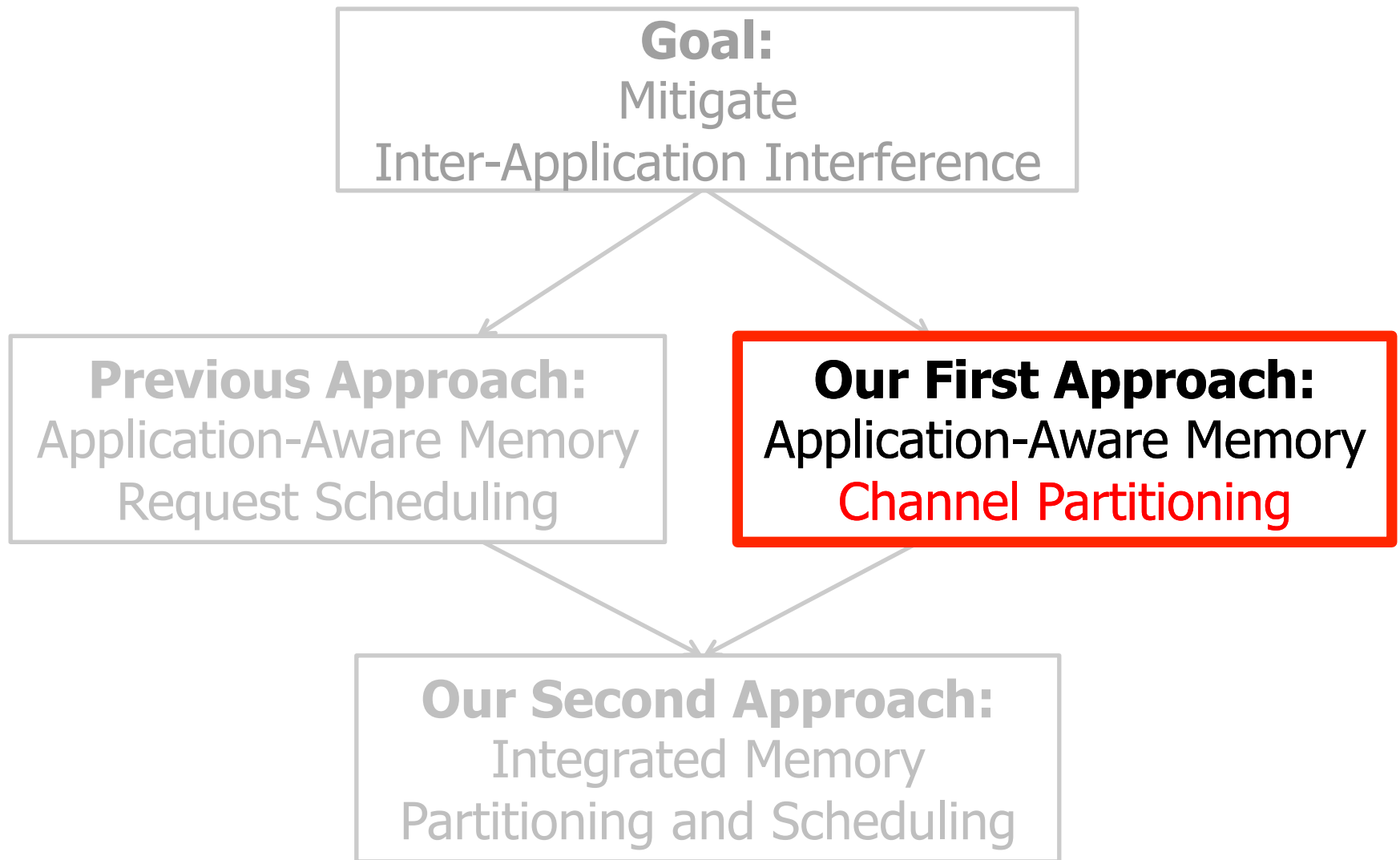
Advantages

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

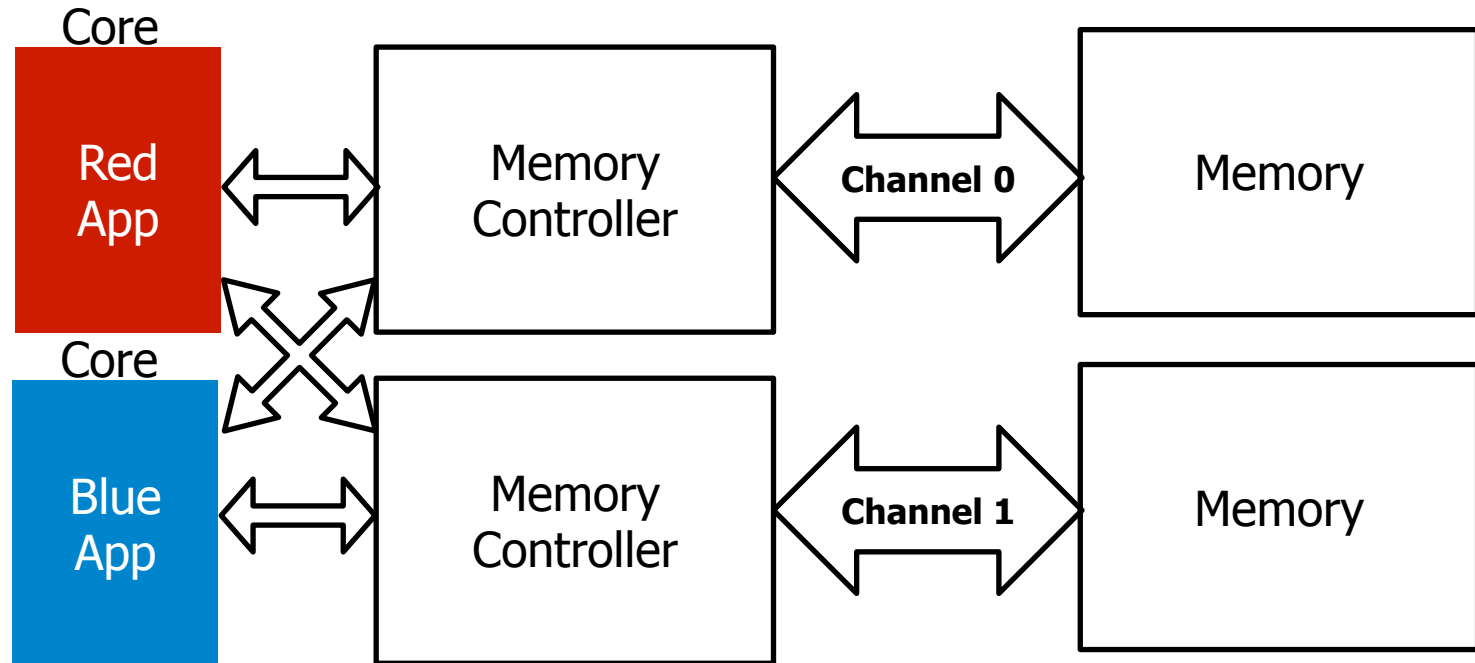
Disadvantages

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

Our Approach

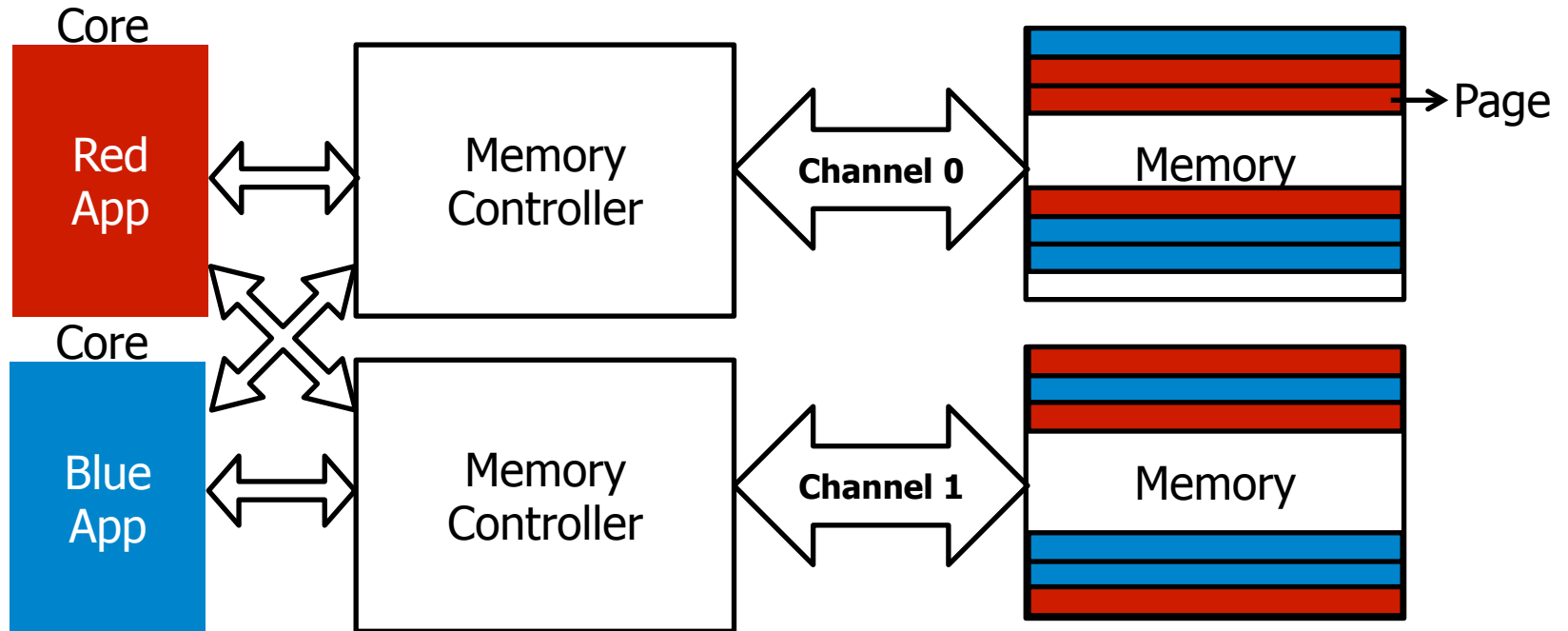


Observation: Modern Systems Have Multiple Channels



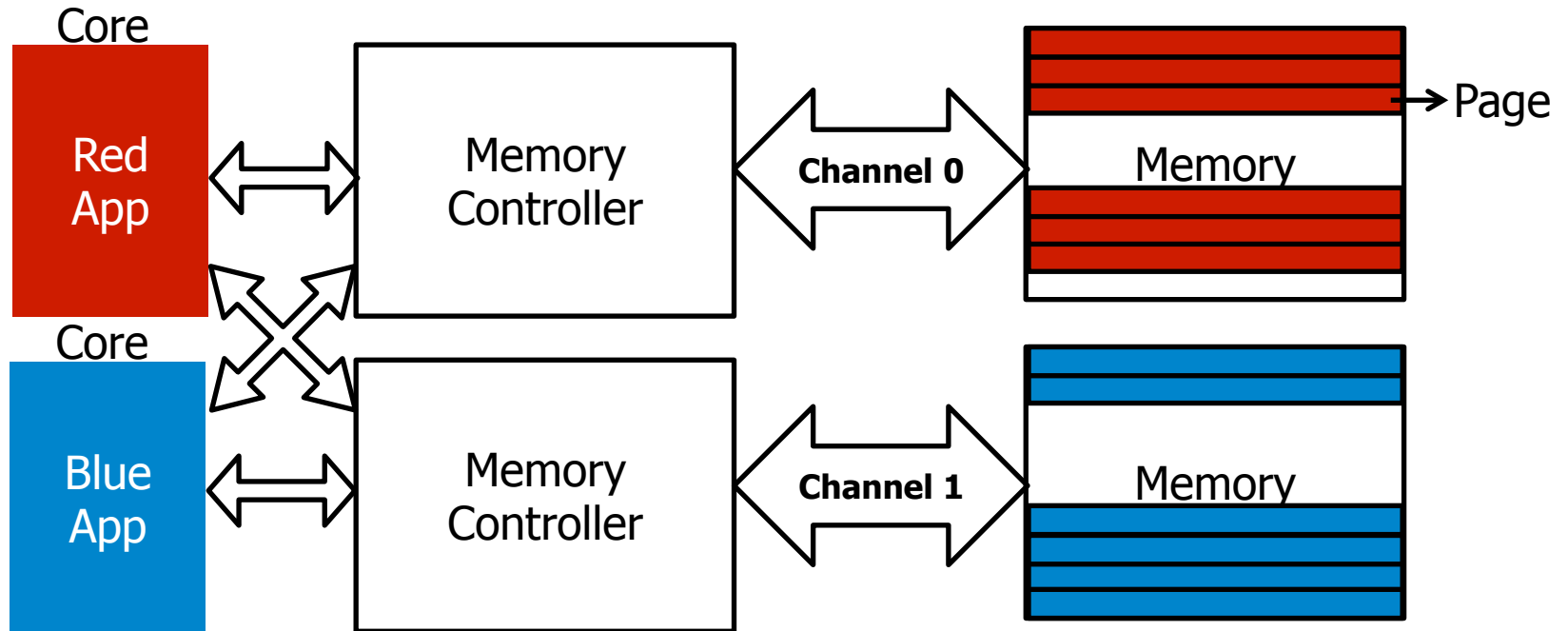
A new degree of freedom
Mapping data across multiple channels

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

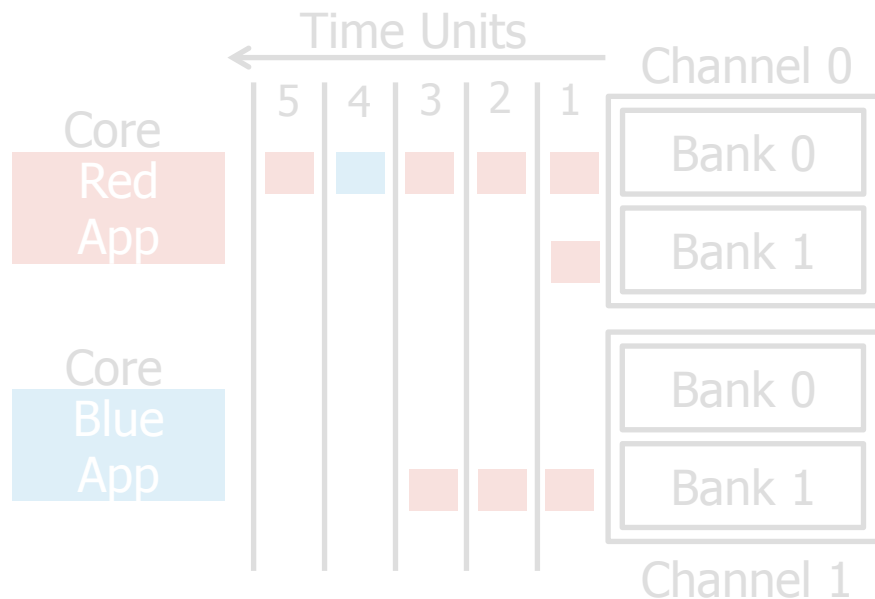
- Map the data of **badly-interfering applications** to different channels

■ Key Principles

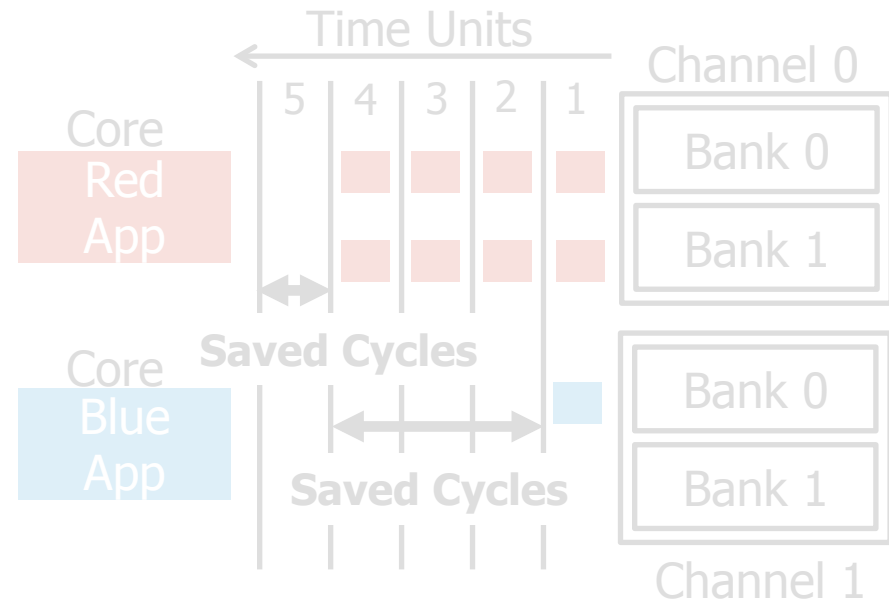
- Separate **low and high memory-intensity applications**
- Separate **low and high row-buffer locality applications**

Key Insight 1: Separate by Memory Intensity

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



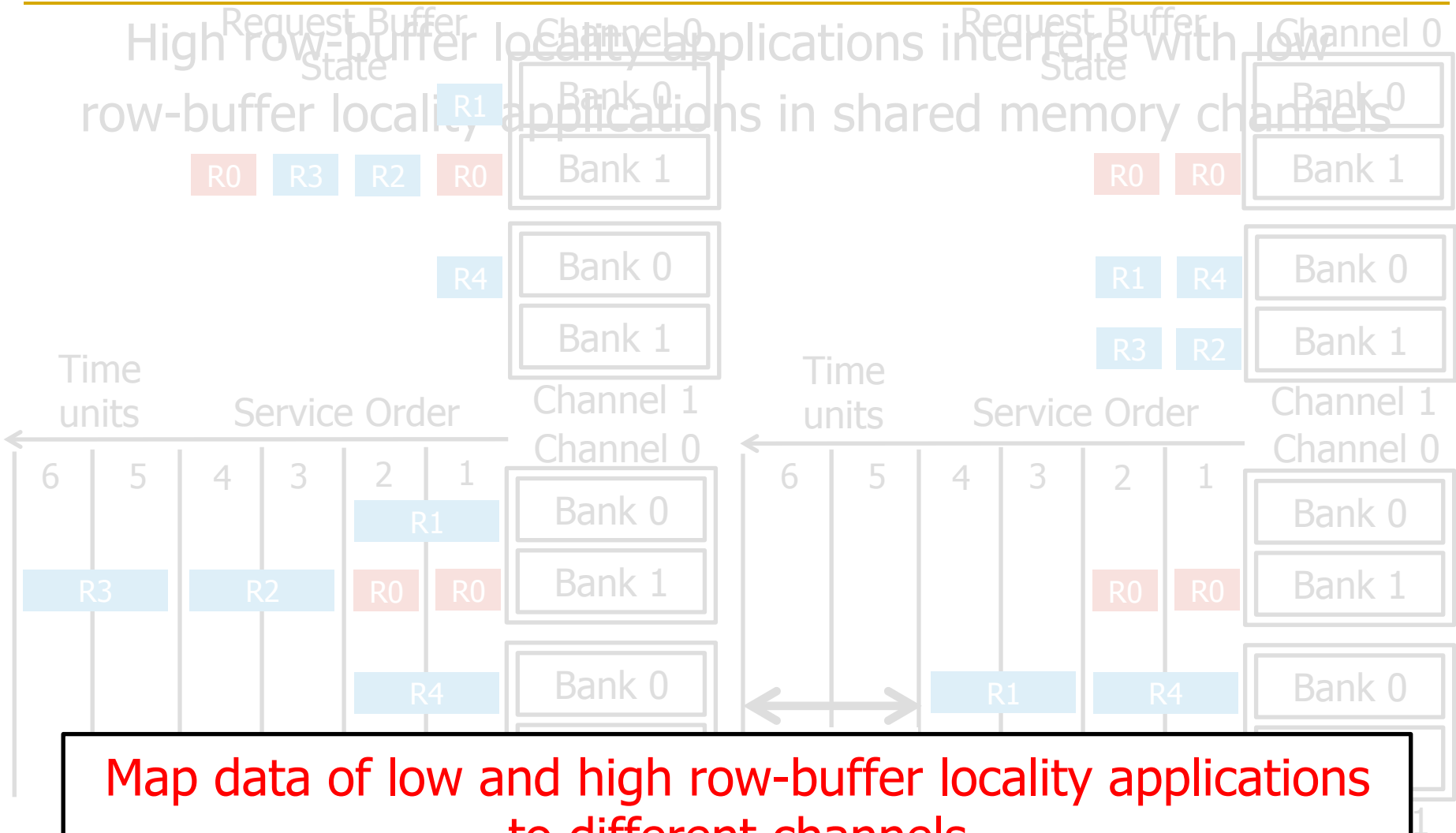
Conventional Page Mapping



Channel Partitioning

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

Hardware

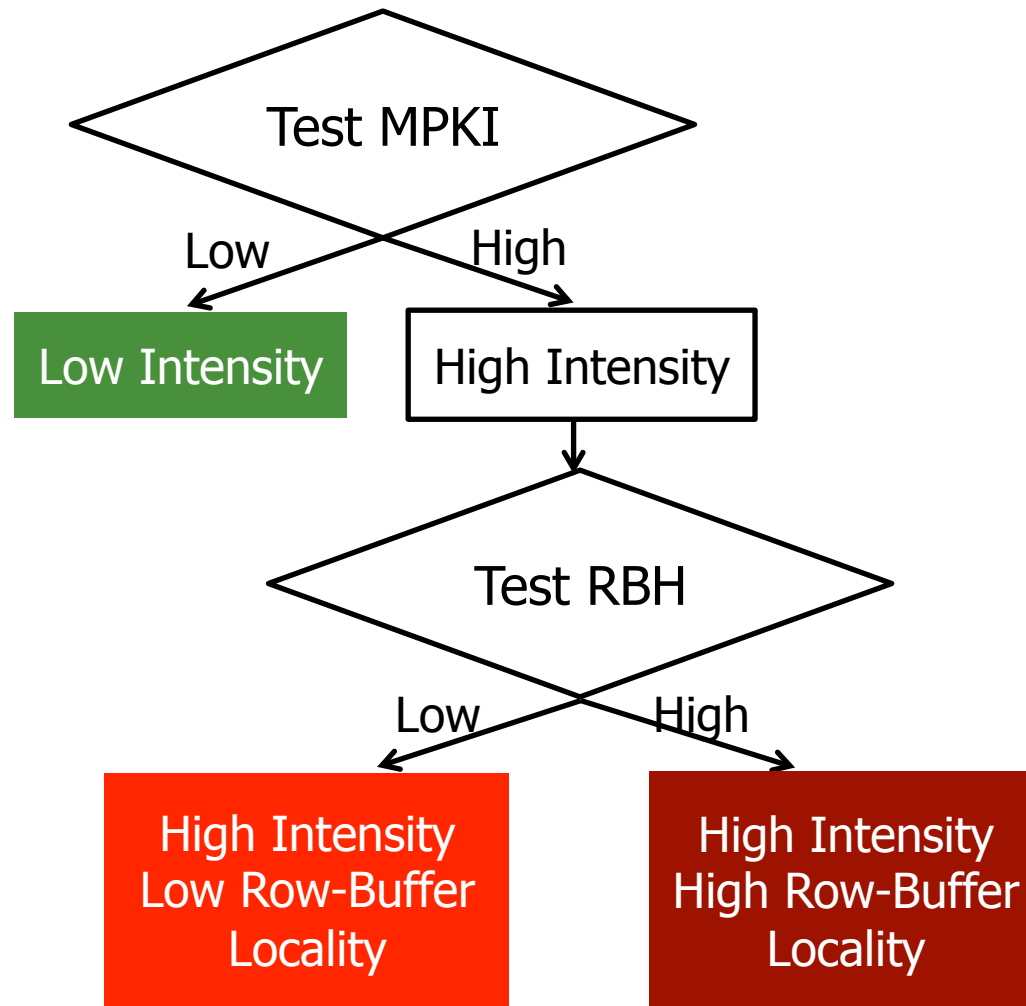
1. Profile applications
2. Classify applications into groups
3. Partition channels between application groups
4. Assign a preferred channel to each application
5. Allocate application pages to preferred channel

**System
Software**

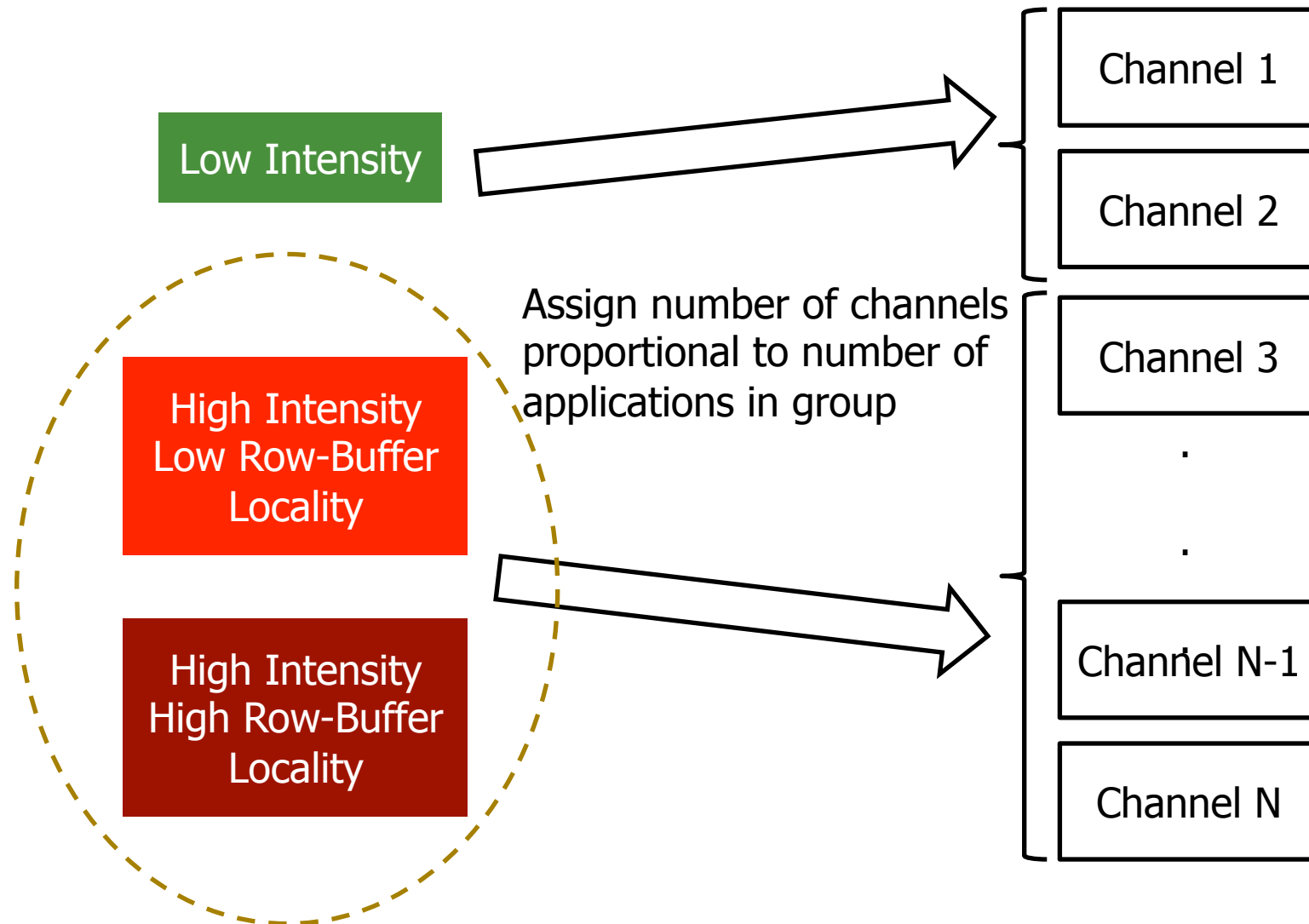
1. Profile Applications

- 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

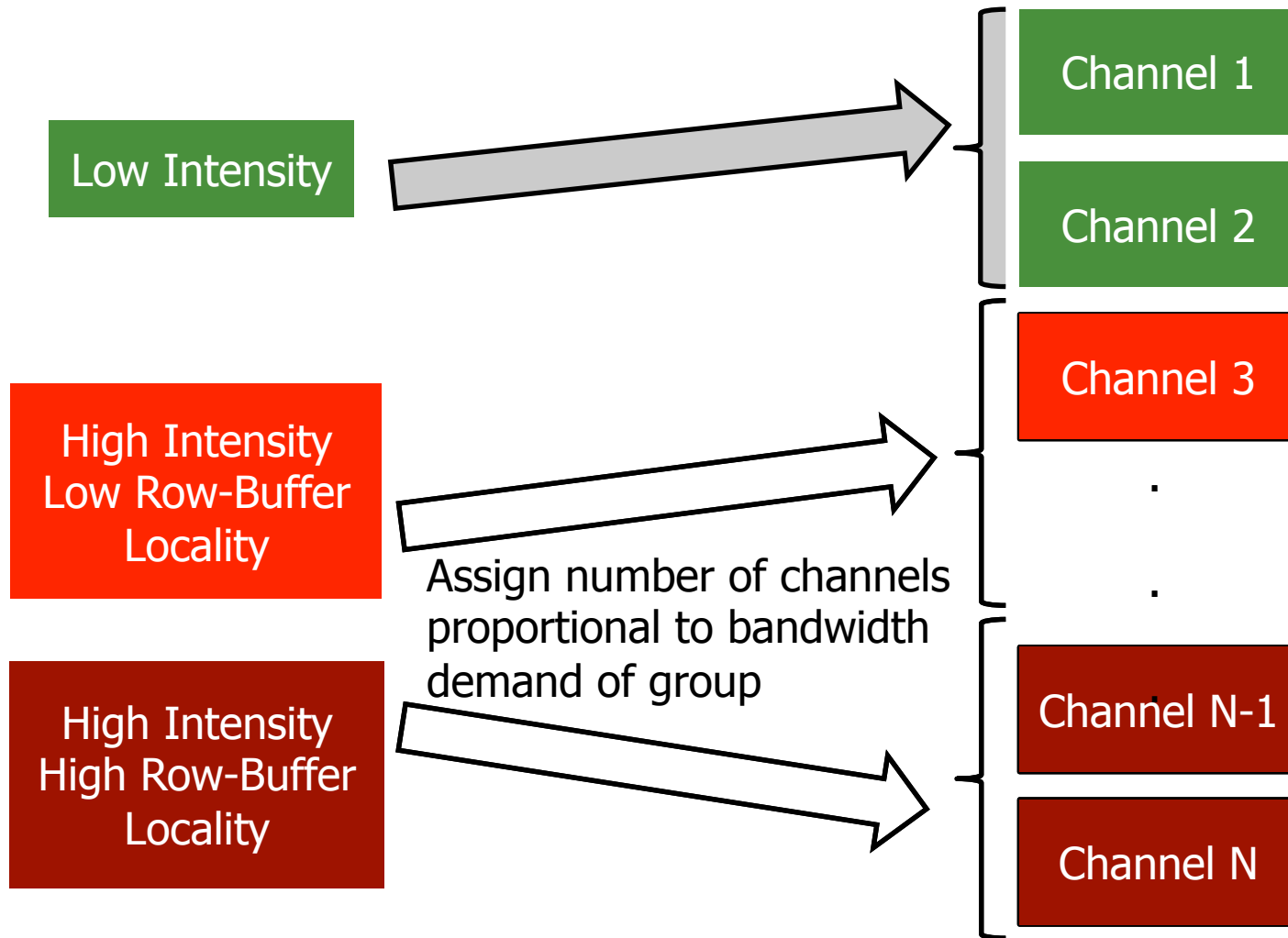
2. Classify Applications



3. Partition Channels Among Groups: Step 1

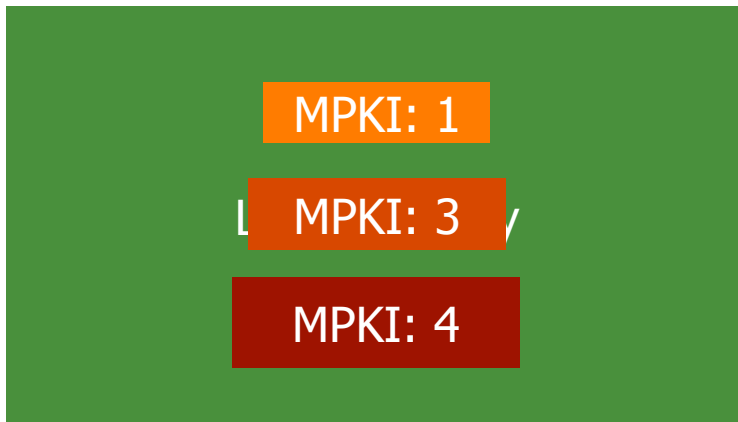


3. Partition Channels Among Groups: Step 2



4. Assign Preferred Channel to Application

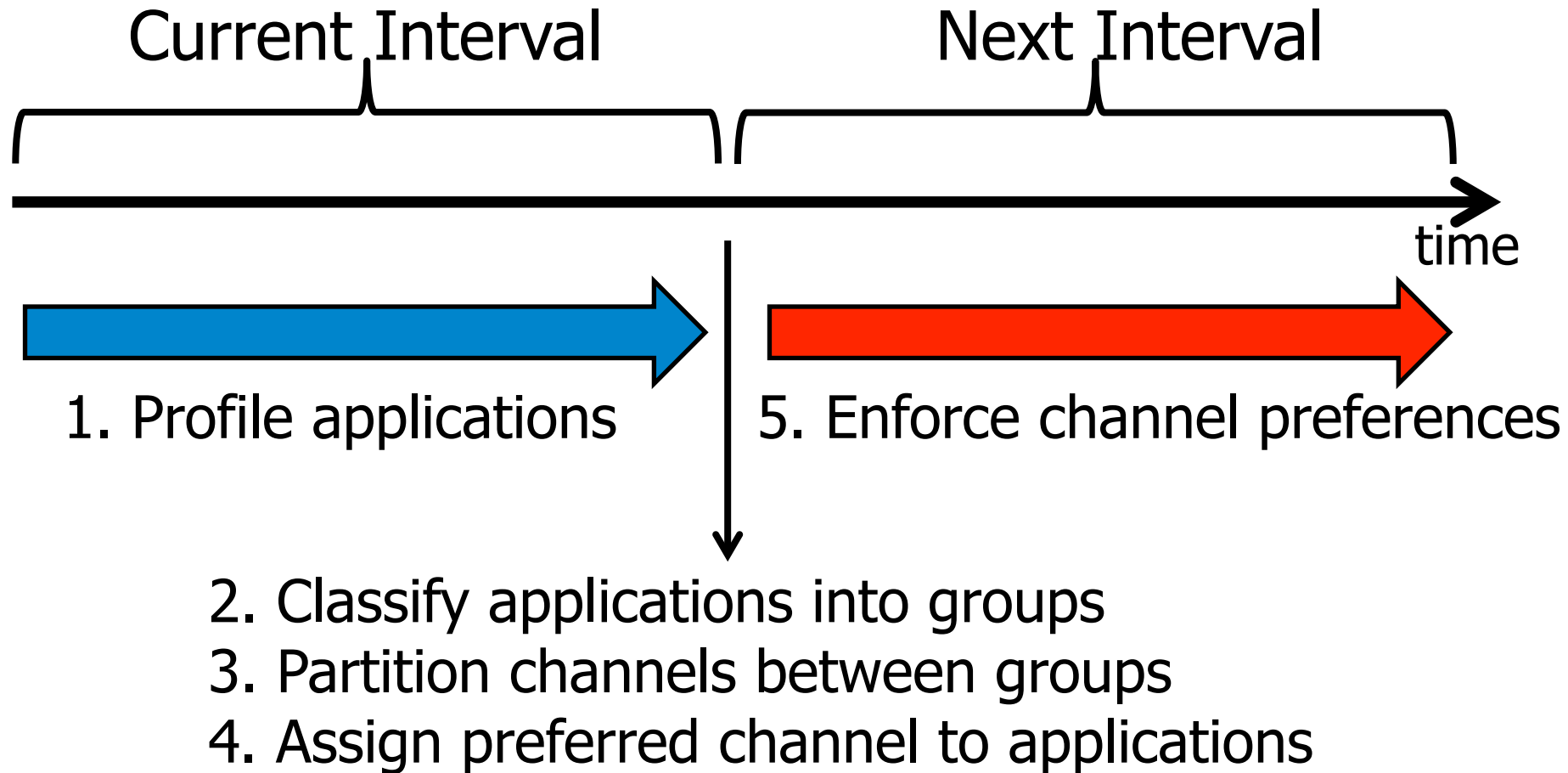
- Assign **each application a preferred channel** from its group's allocated channels
- Distribute applications to channels such that **group's bandwidth demand is balanced** across its channels



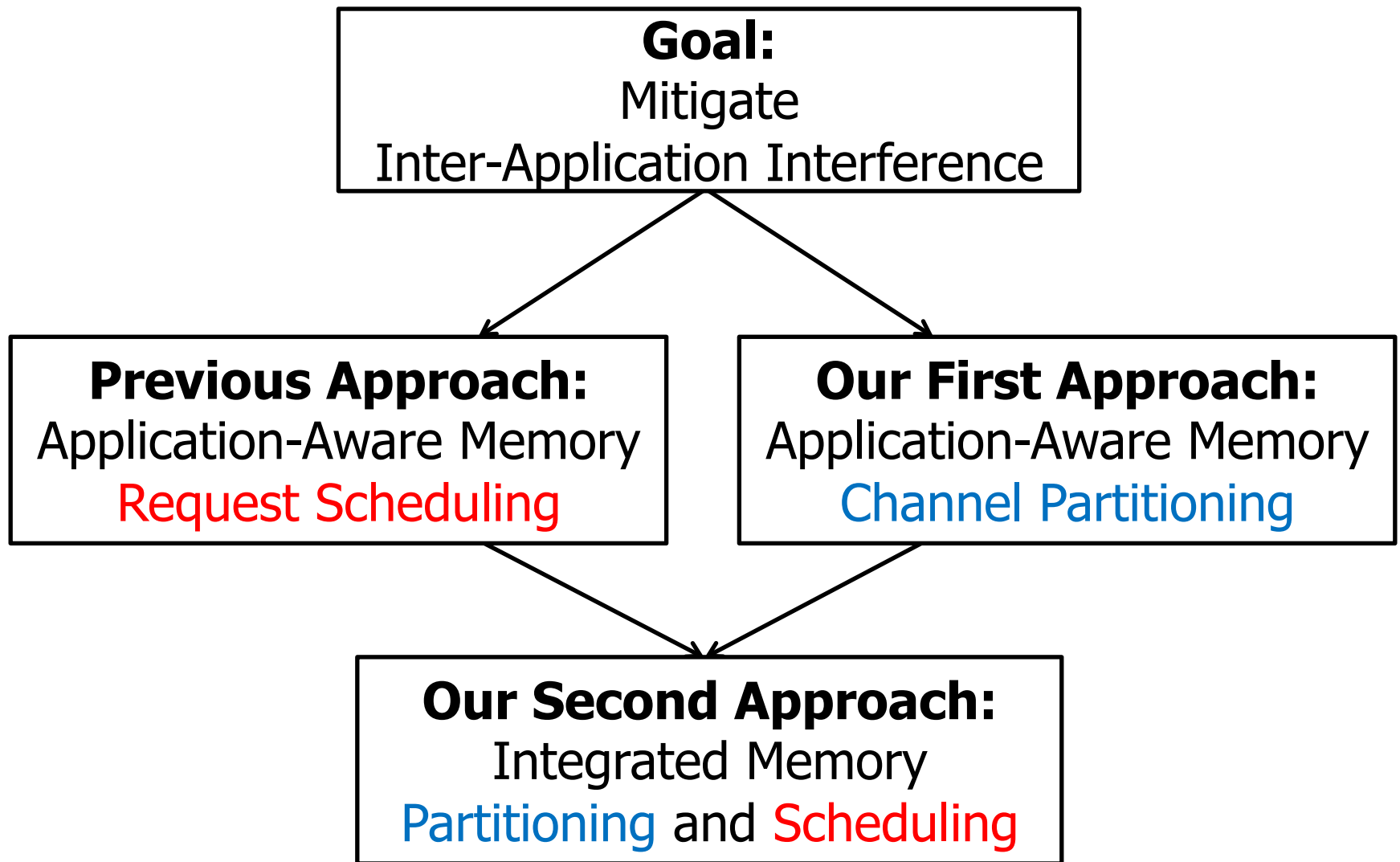
5. Allocate Page to Preferred Channel

- **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

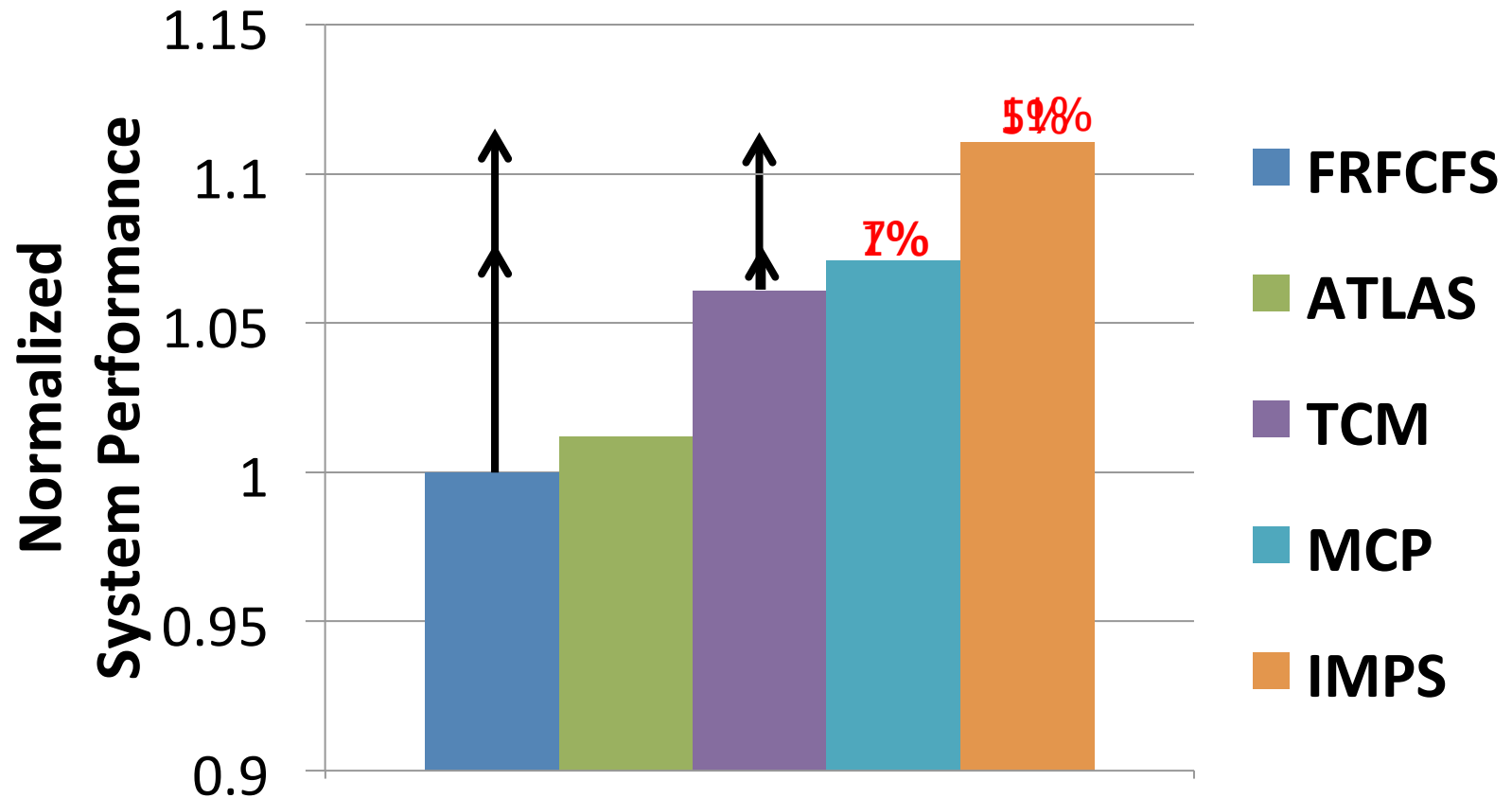
- ❑ System Performance $Weighted\ Speedup = \sum_i \frac{IPC_i^{shared}}{IPC_i^{alone}}$

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

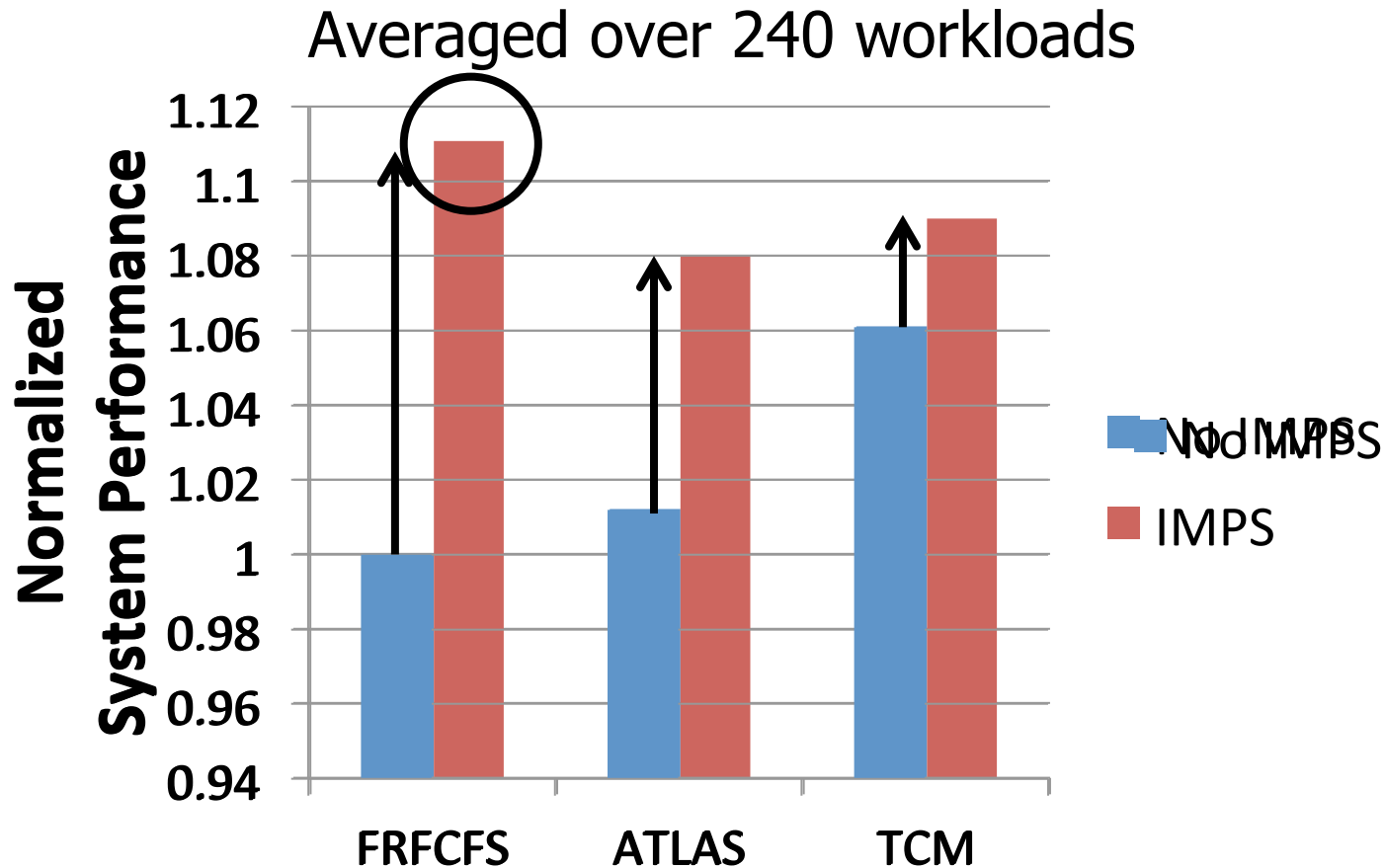
Comparison to Previous Scheduling Policies

Averaged over 240 workloads



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

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
-

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]

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]**

QoS-Aware Memory Systems

Onur Mutlu

onur@cmu.edu

July 8, 2013

INRIA

Carnegie Mellon

Additional Material

Two Works

- 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)

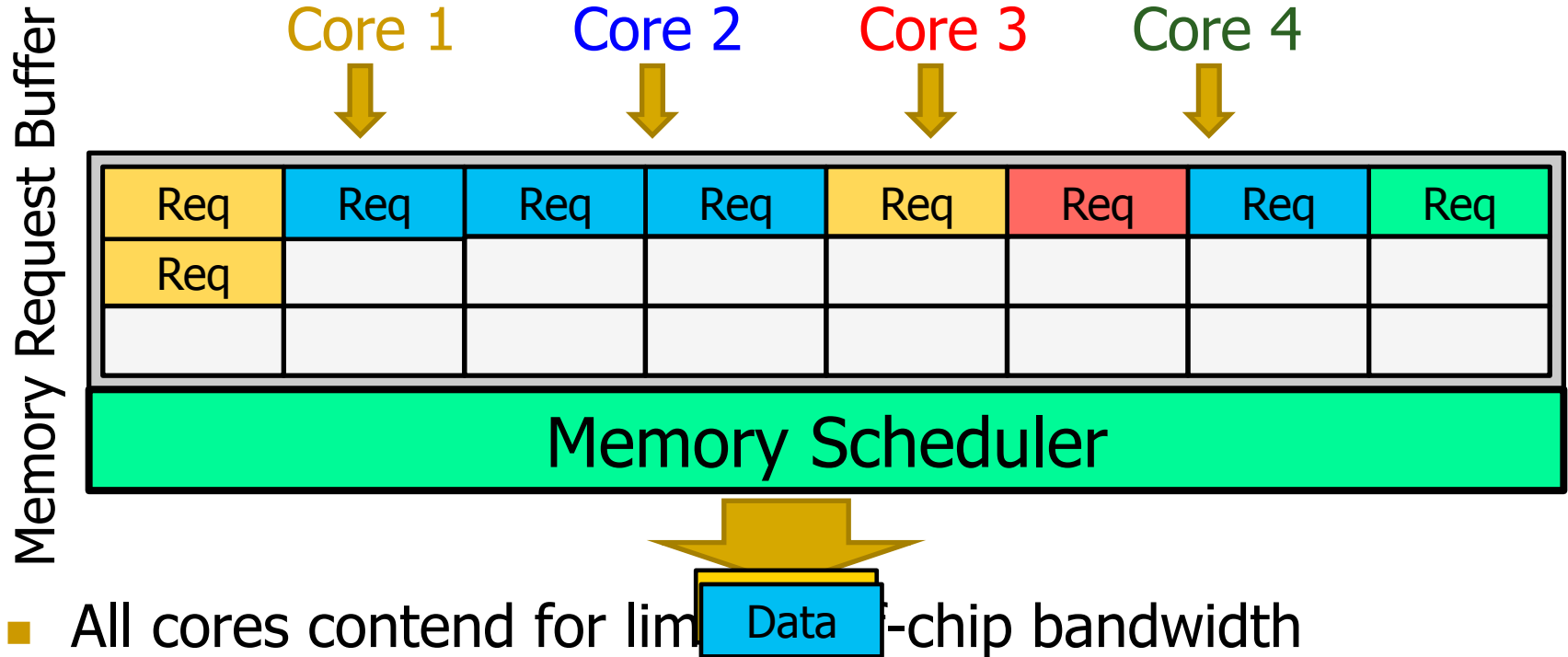
Staged Memory Scheduling

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.

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 limited on-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 minimize starvation

Older

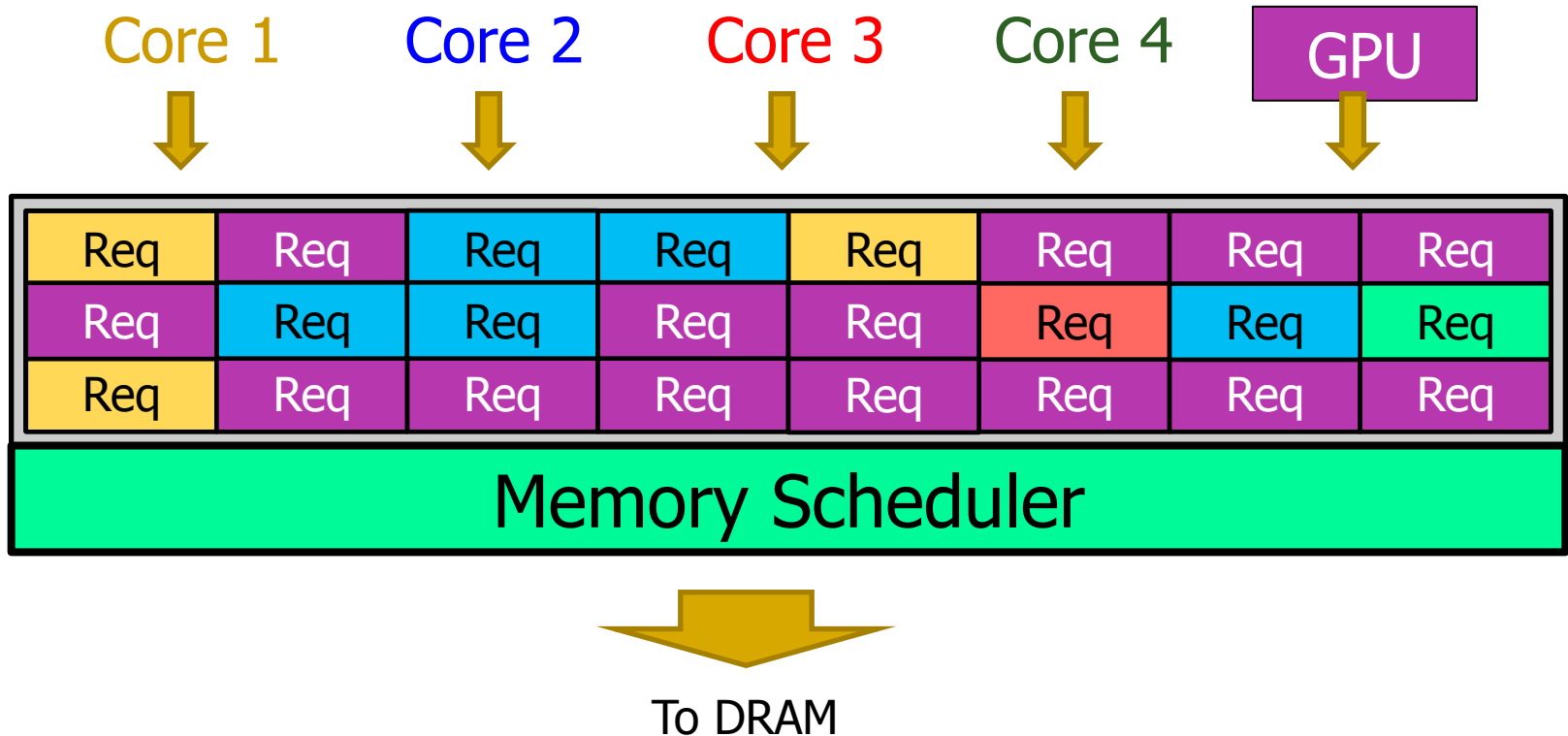
Application	Memory Intensity (MPKI)	row
1	5	
2	1	
3	2	
4	10	

Newer

Memory Scheduling for CPU-GPU Systems

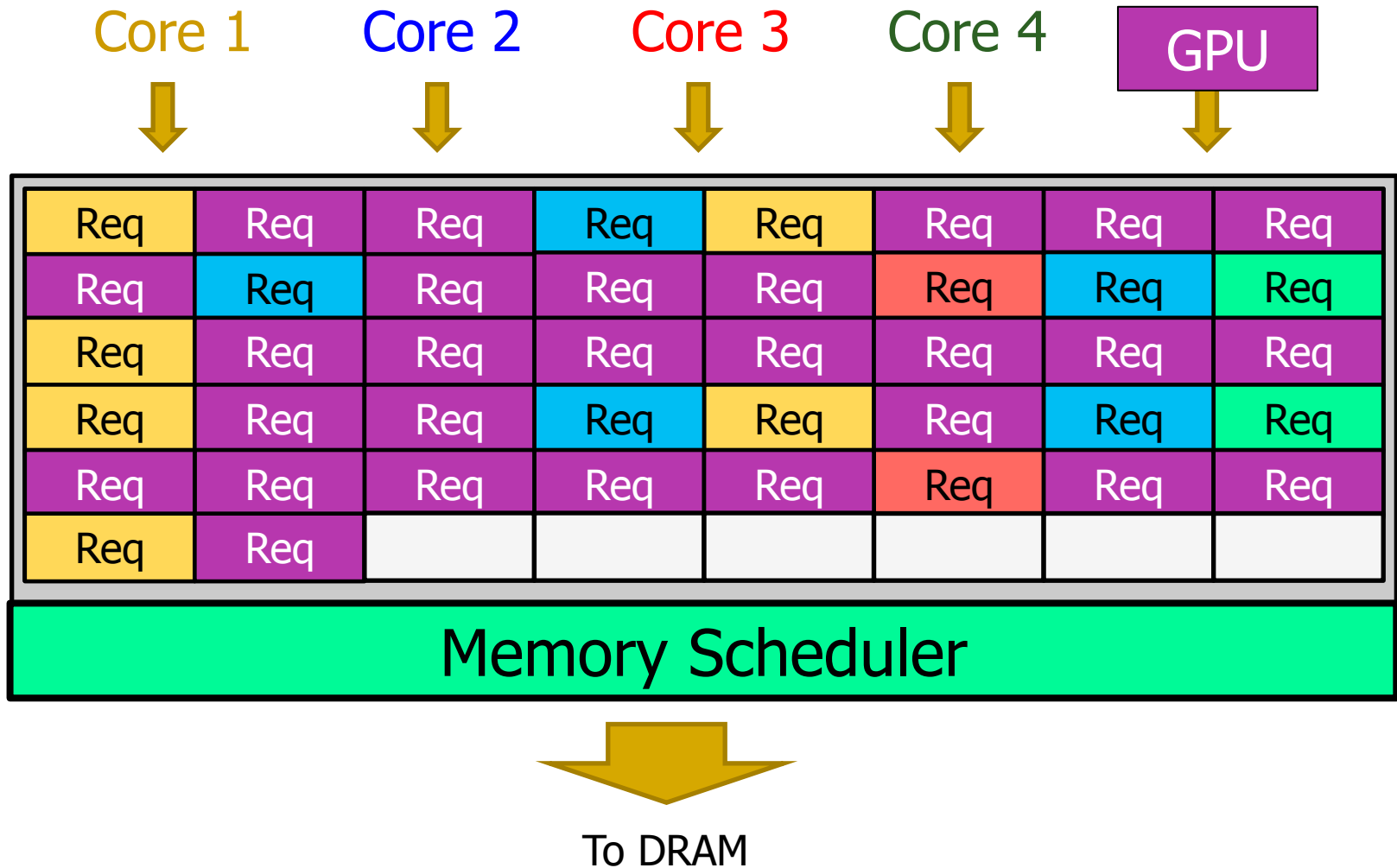
- Current and future systems integrate a GPU along with multiple cores
- GPU shares the main memory with the CPU cores
- GPU is **much more (4x-20x) memory-intensive** than CPU
- How should memory scheduling be done when GPU is integrated on-chip?

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

Req	Req	Req	Req	Req	Req	Req	Req
Req	Req	Req	Req	Req	Req	Req	Req
Req	Req	Req	Req	Req	Req	Req	Req
Req	Req	Req	Req	Req	Req	Req	Req
Req	Req	Req	Req	Req	Req	Req	Req
Req	Req						

More Complex Memory Scheduler

- 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

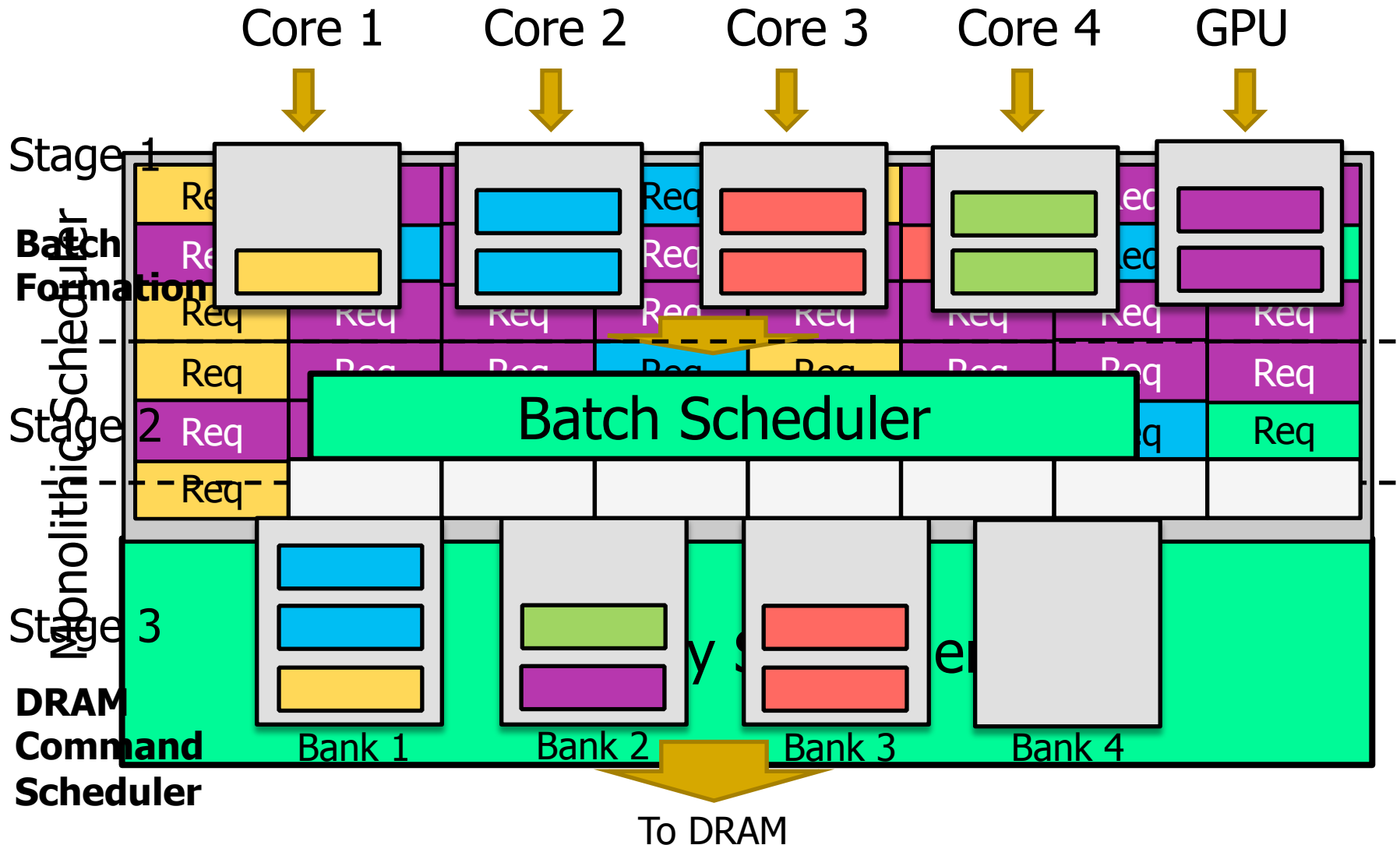
Key Functions of a Memory Controller

- 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**

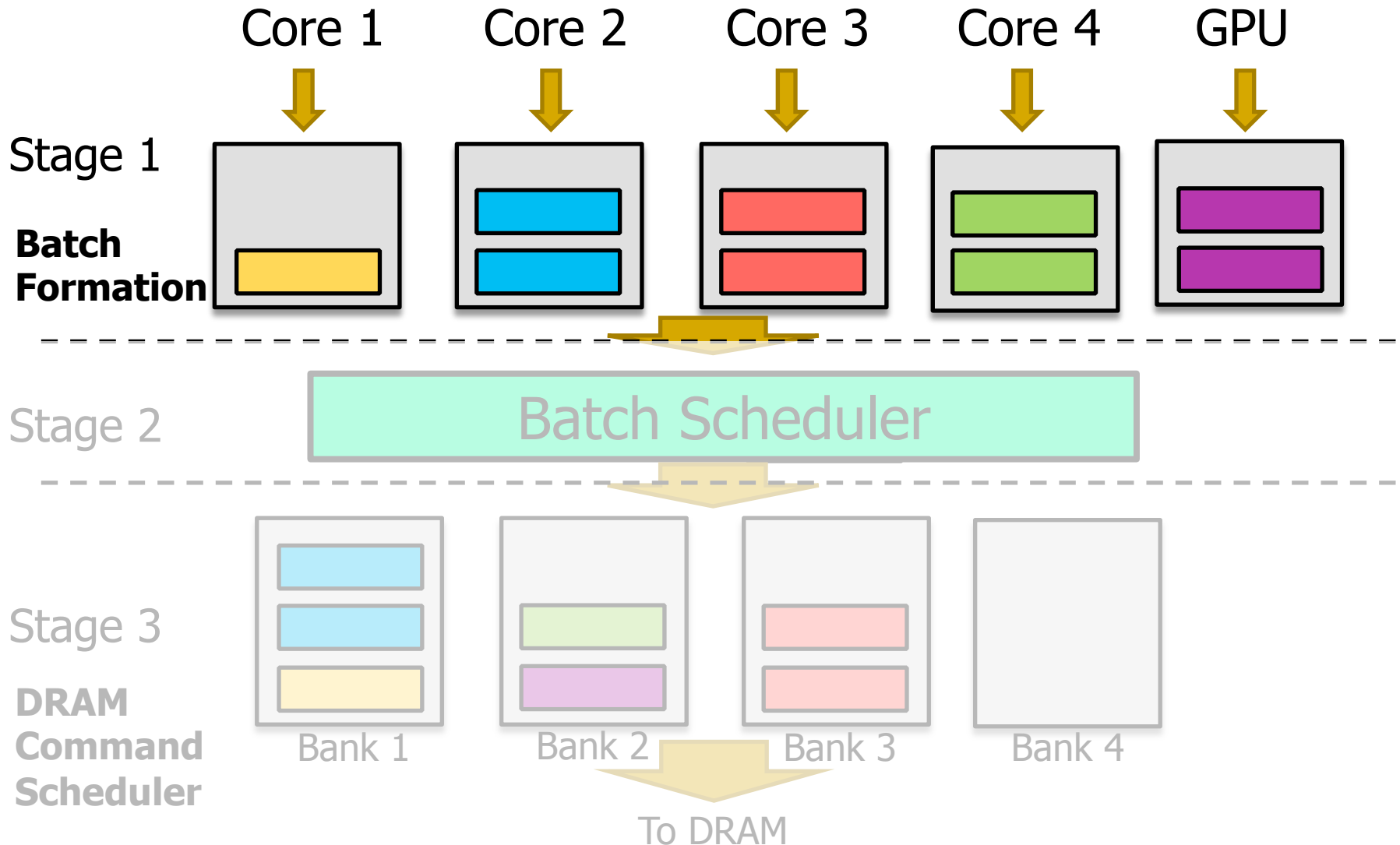
Key Idea: Decouple Tasks into Stages

- 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



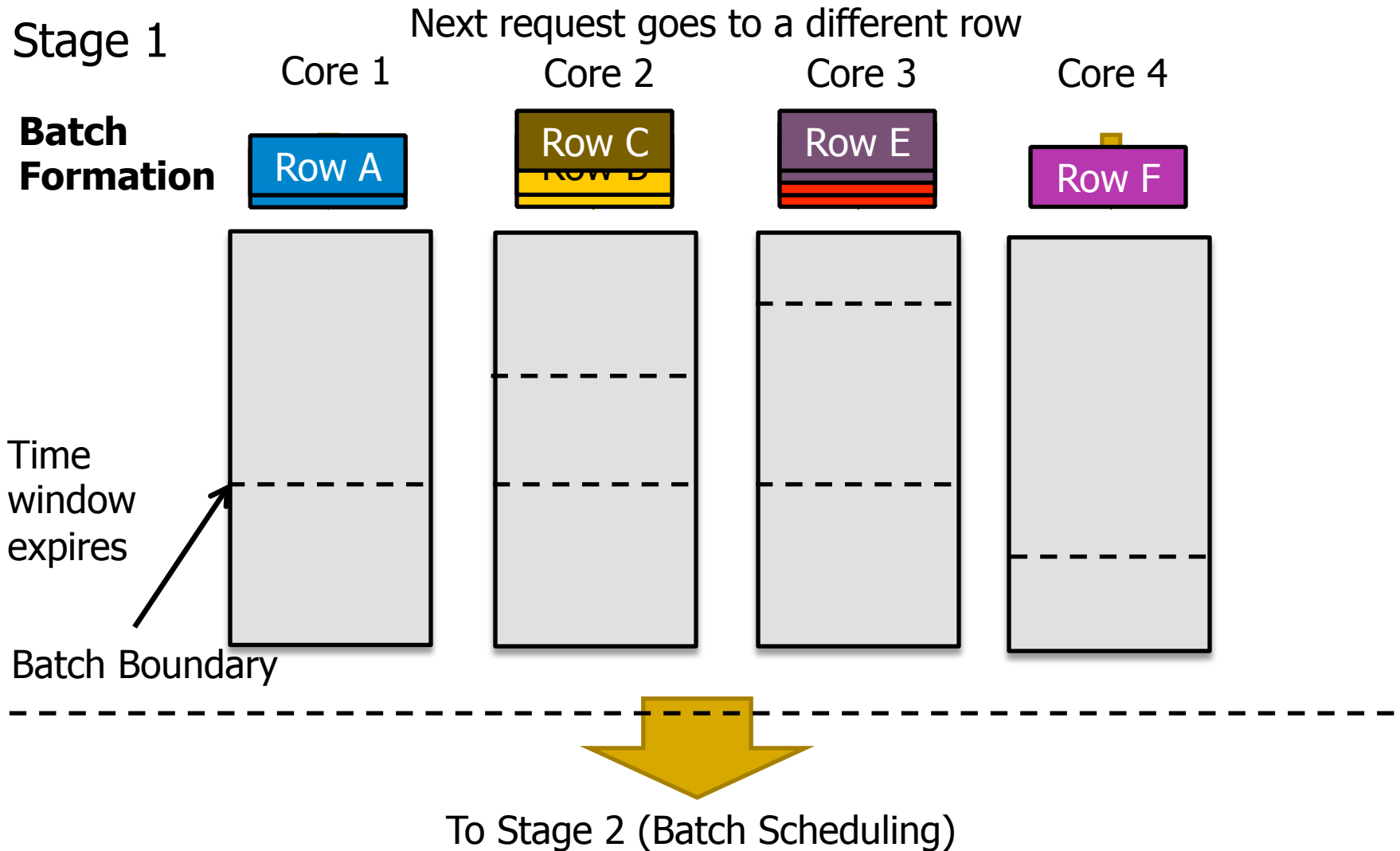
SMS: Staged Memory Scheduling



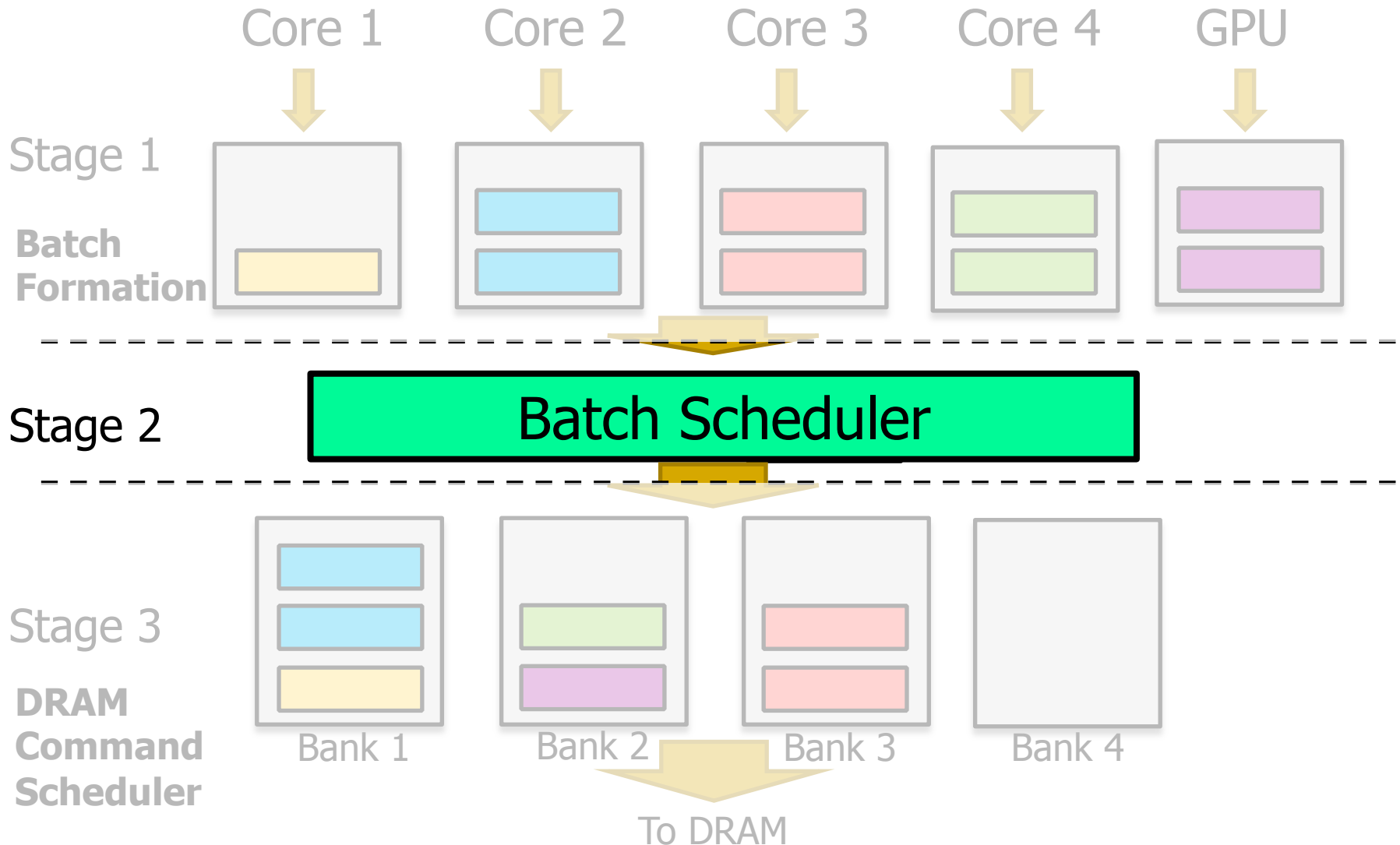
Stage 1: Batch Formation

- 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

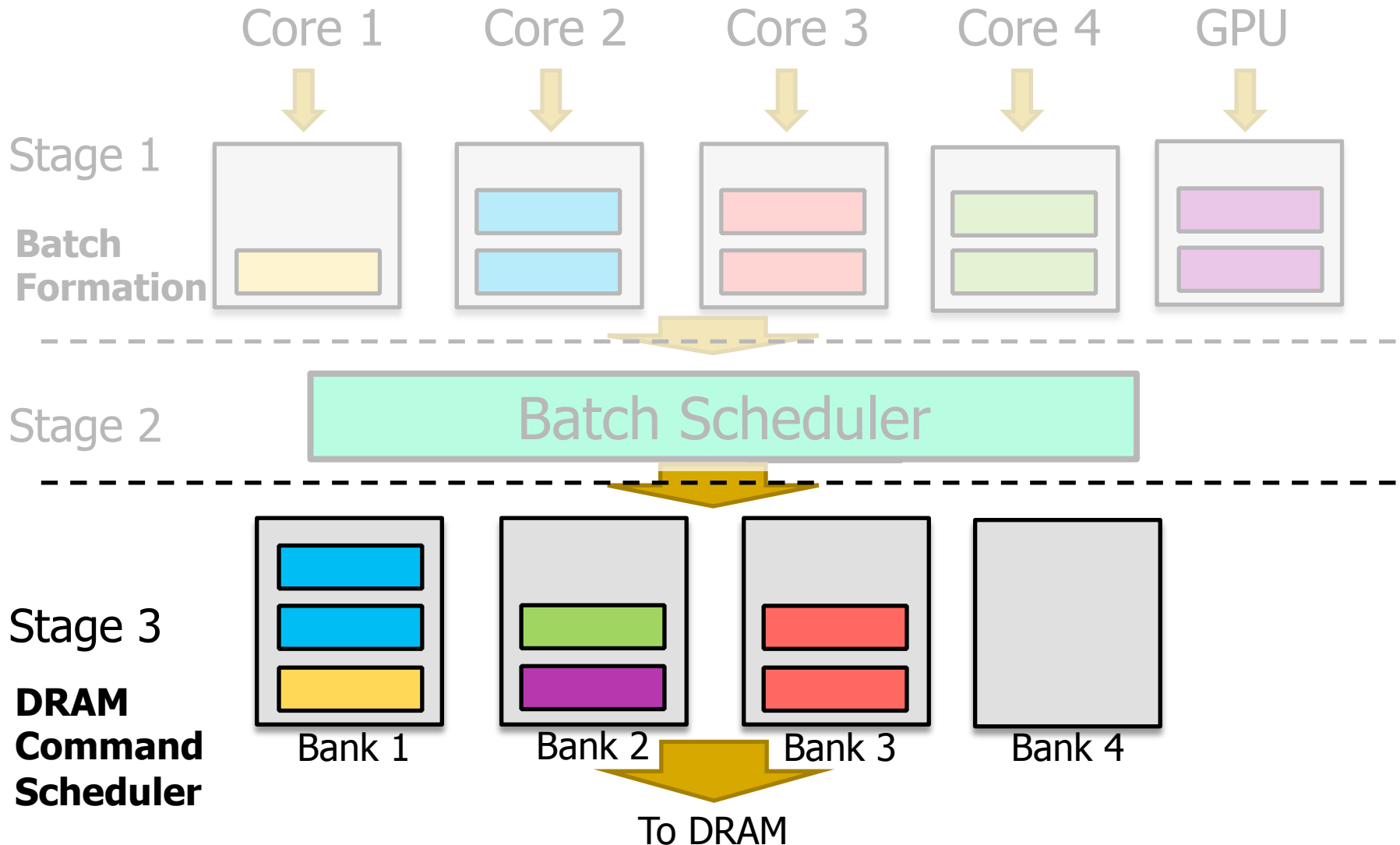
■ **Round-Robin (RR)**

- ❑ 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

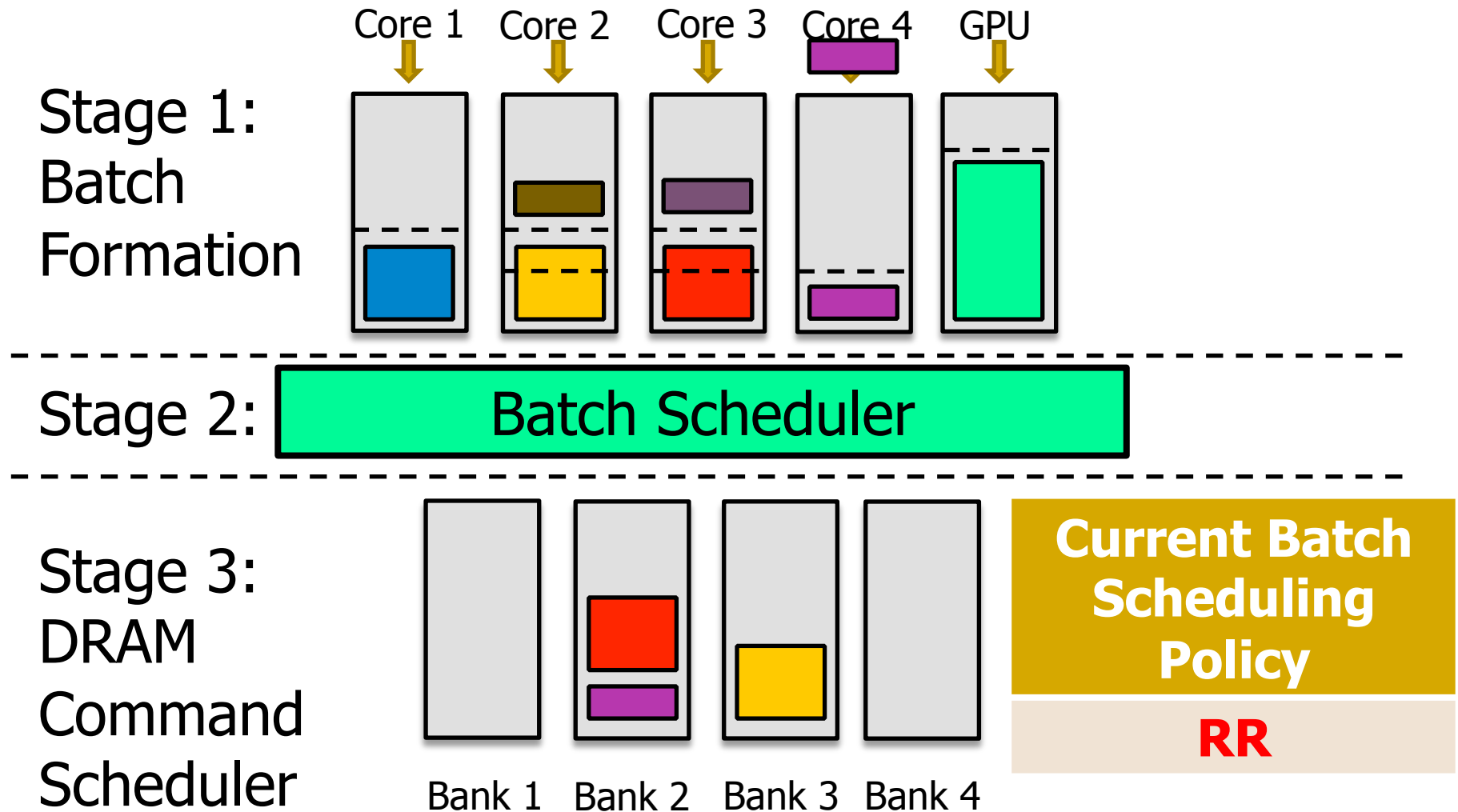
SMS: Staged Memory Scheduling



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)

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

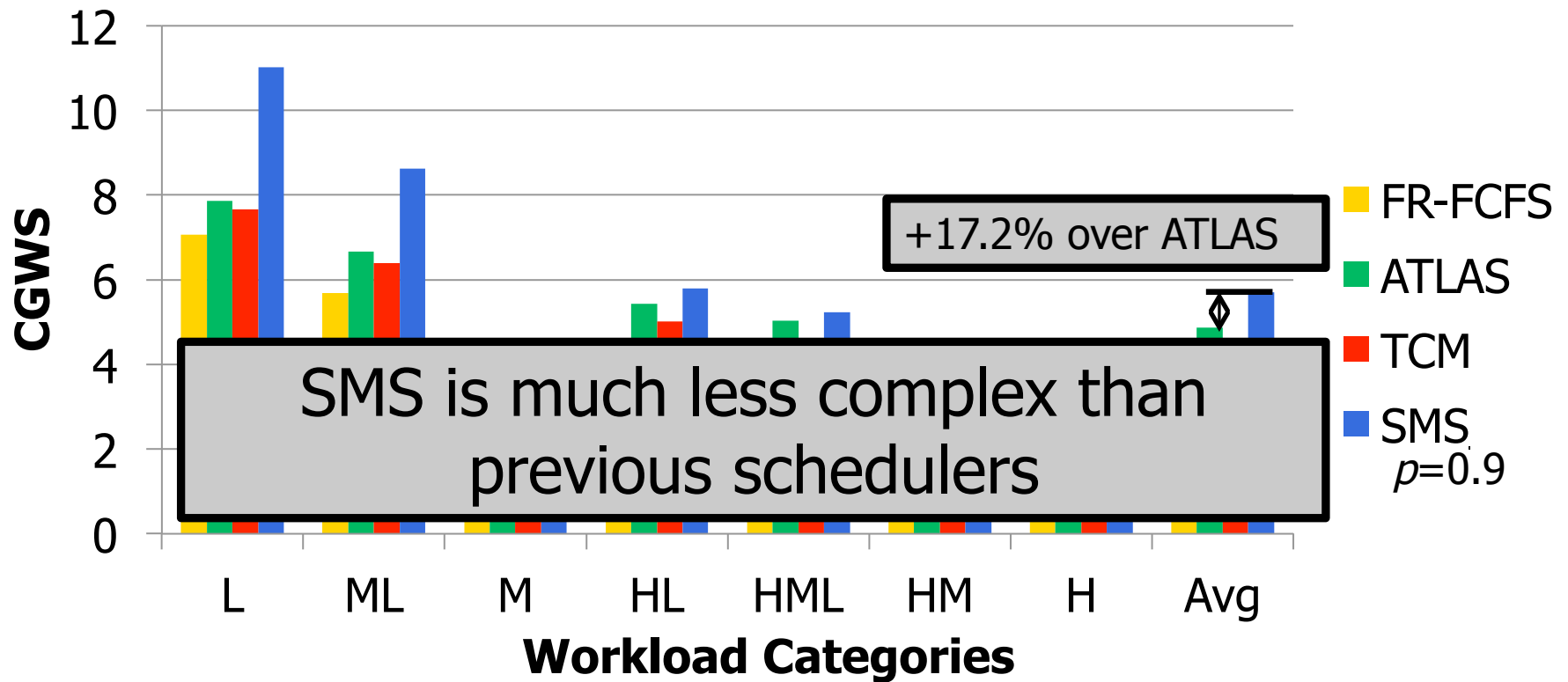
- GPU has **low** weight (weight = 1)

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

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

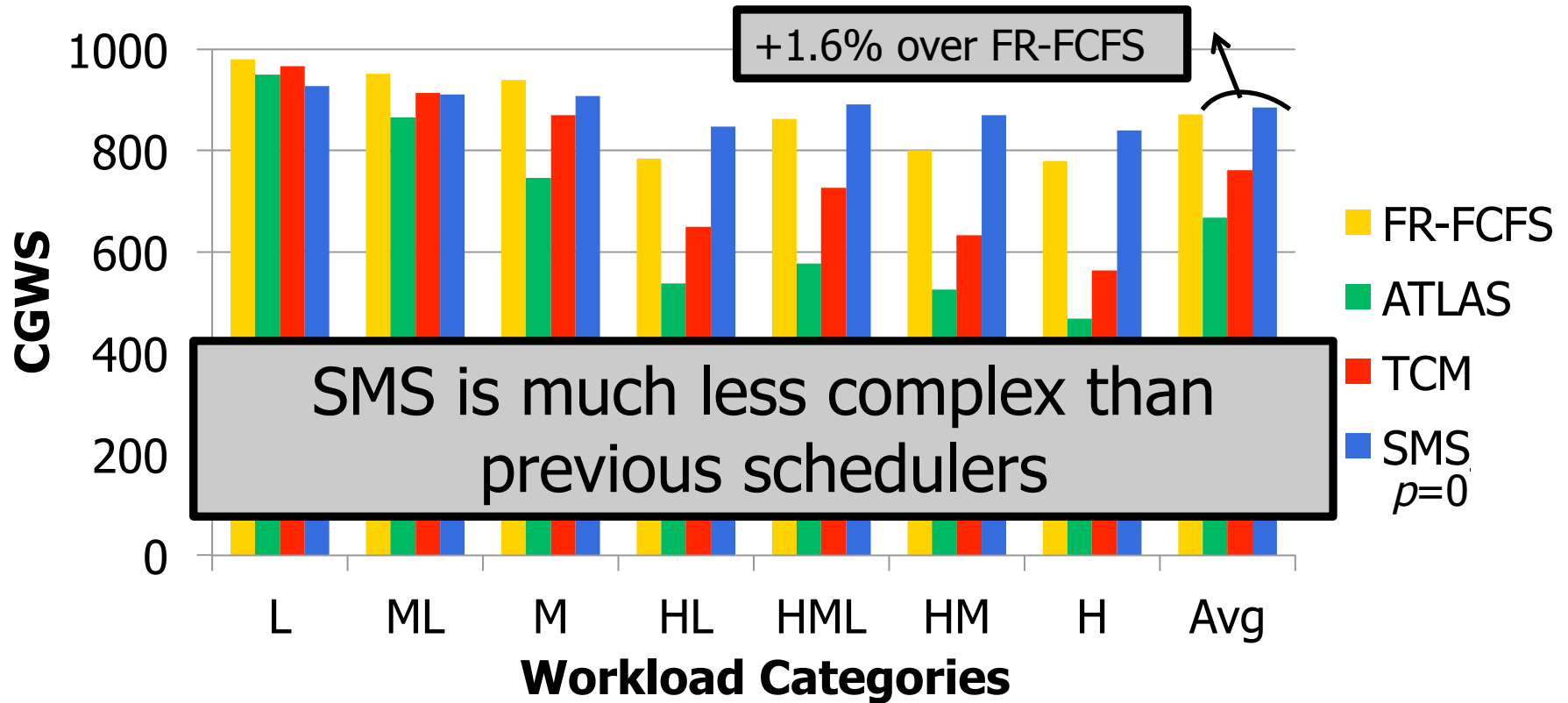
- GPU has **high** weight (weight = 1000)

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

1000

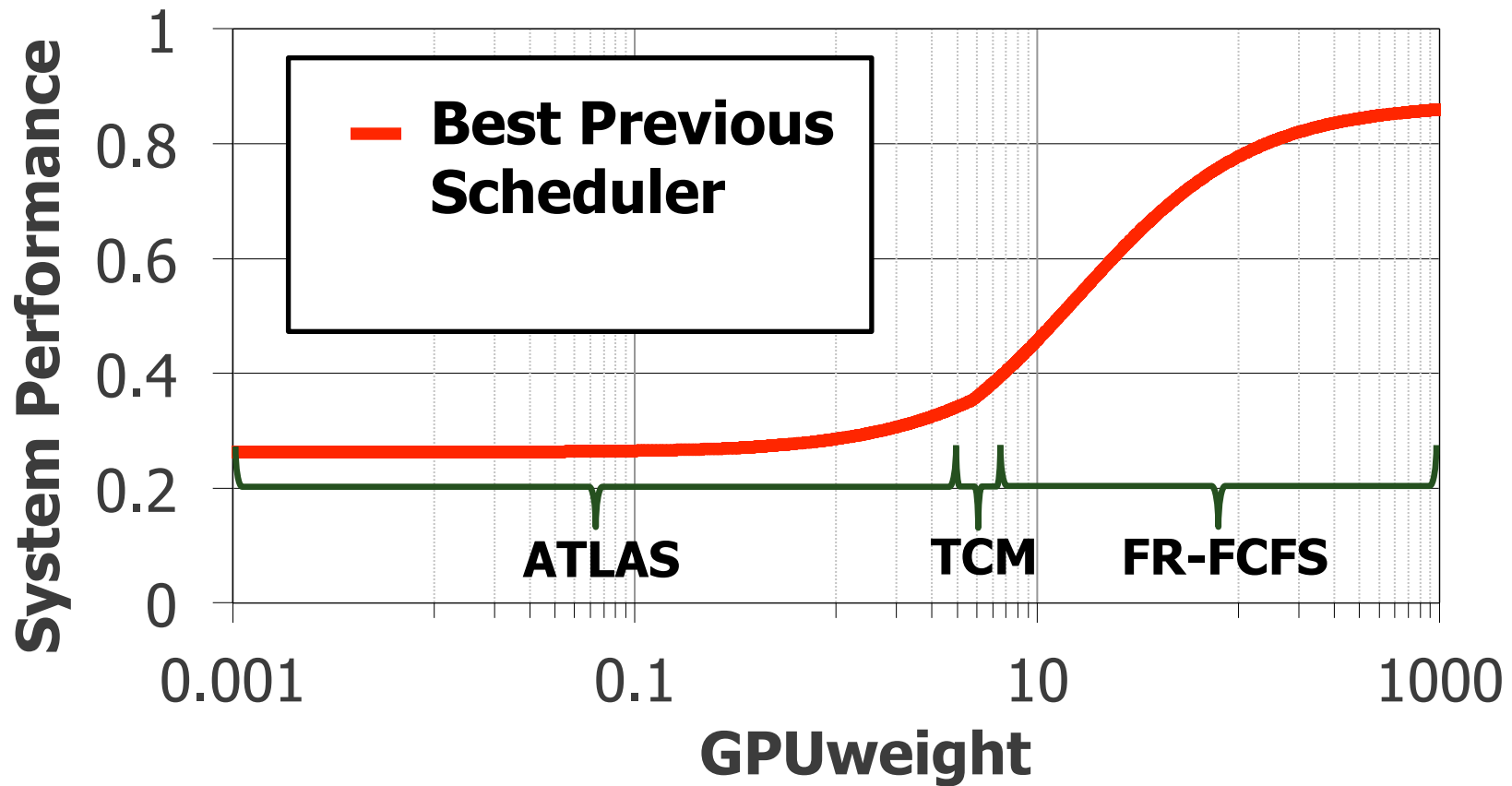
- Configure SMS such that p , SJF probability, is set to 0
 - **Always uses round-robin** batch scheduling → prioritizes memory-intensive applications (GPU)

Performance: GPU-Focused System

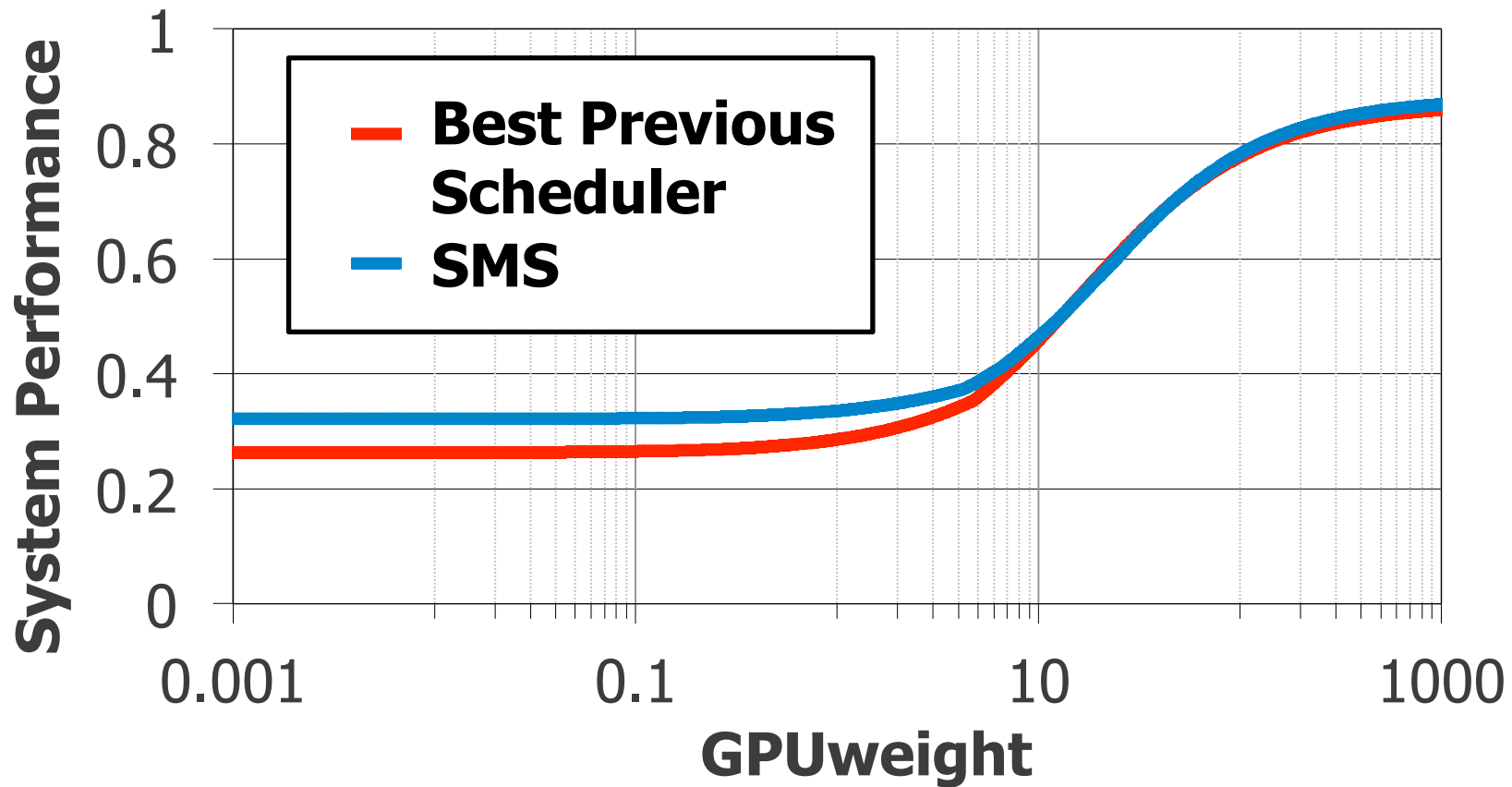


- Round-robin batch scheduling policy schedules GPU requests more frequently

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
-