Scalable Many-Core Memory Systems Topic 3: Memory Interference and QoS-Aware Memory Systems

Prof. Onur Mutlu
http://www.ece.cmu.edu/~omutlu
onur@cmu.edu

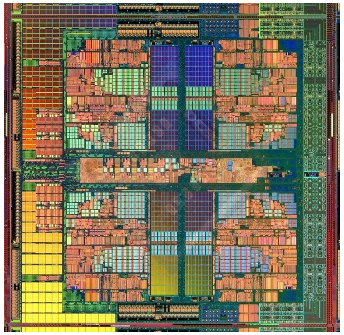
HiPEAC ACACES Summer School 2013
July 15-19, 2013

Carnegie Mellon

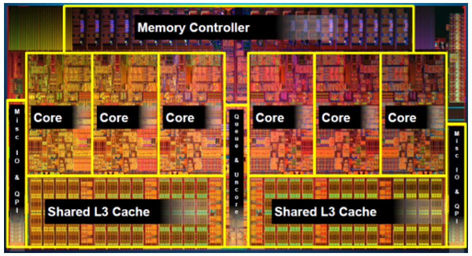
SAFARI

Trend: Many Cores on Chip

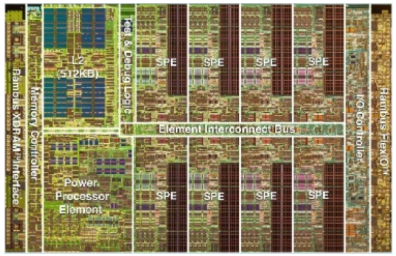
- Simpler and lower power than a single large core
- Large scale parallelism on chip



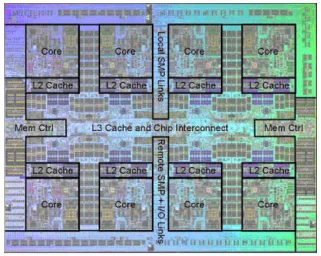
AMD Barcelona
4 cores



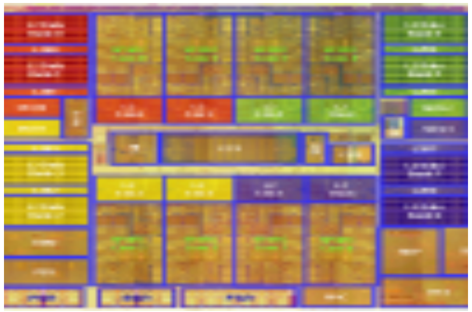
Intel Core i7
8 cores



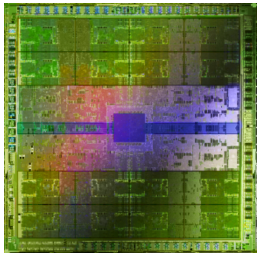
IBM Cell BE
8+1 cores



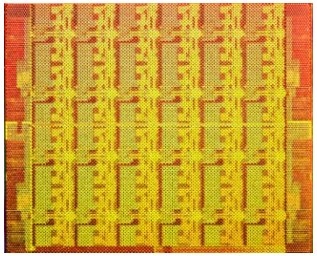
IBM POWER7
8 cores



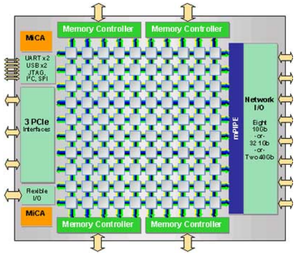
Sun Niagara II
8 cores



Nvidia Fermi
448 “cores”



Intel SCC
48 cores, networked



Tilera TILE Gx
100 cores, networked

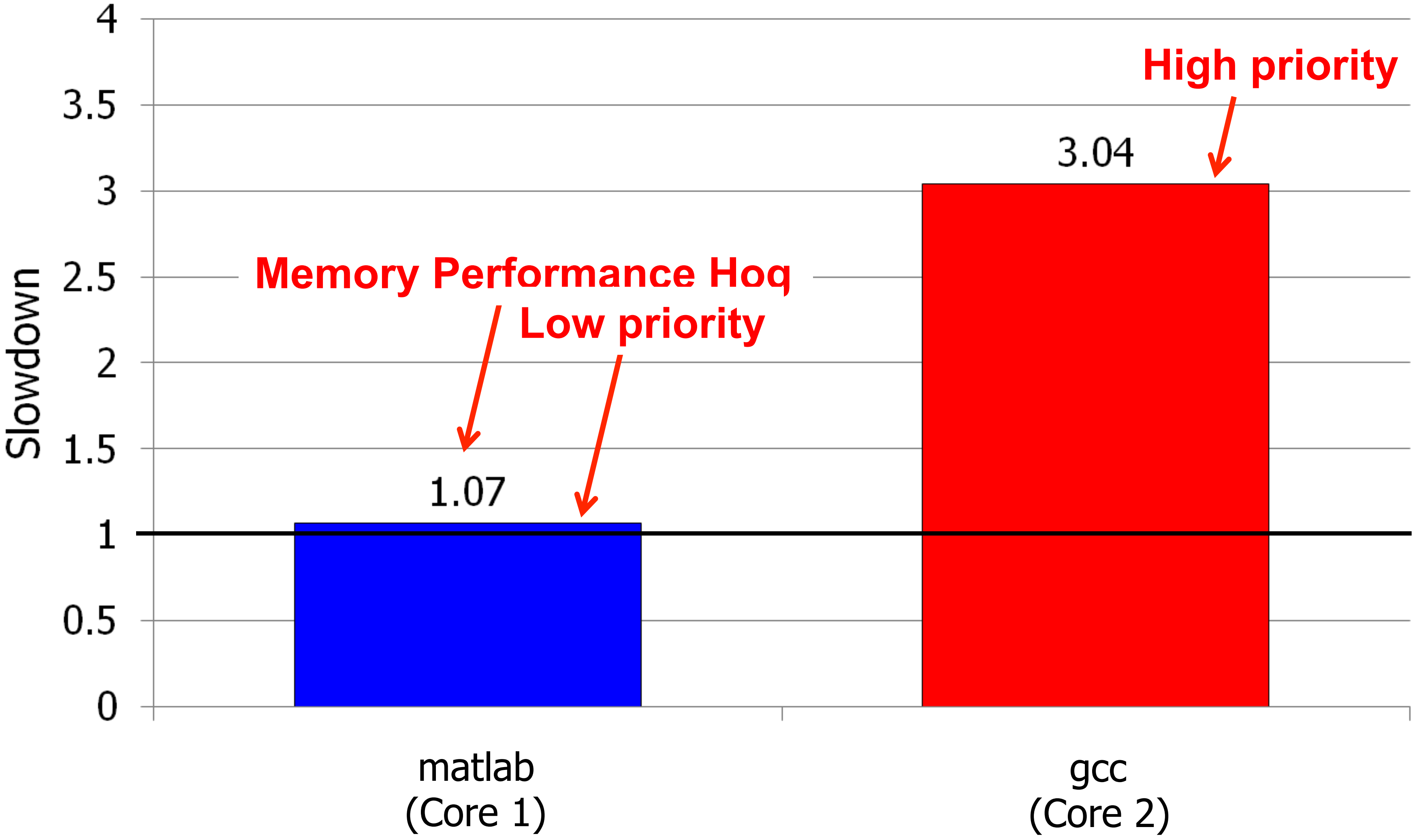
SAFARI

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



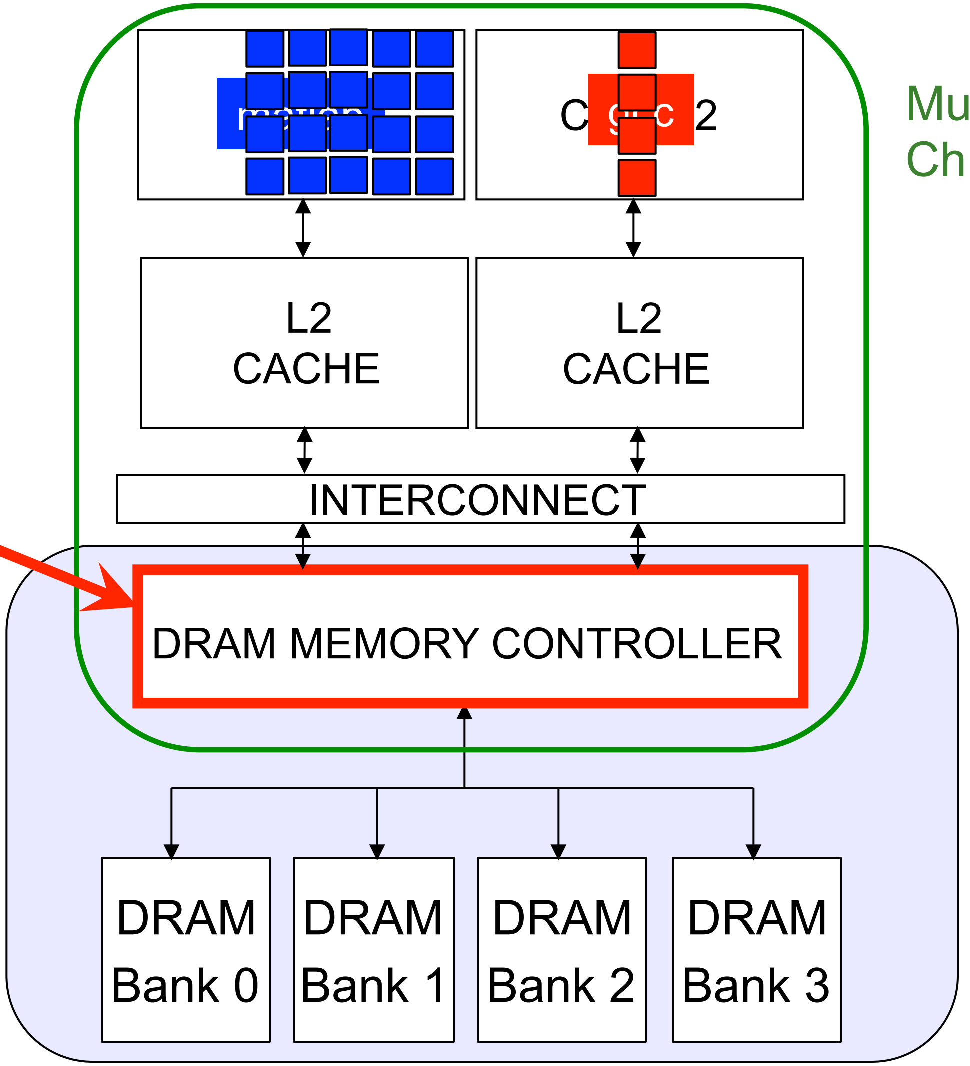
High priority

Memory Performance Hog

Low priority

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

Uncontrolled Interference: An Example

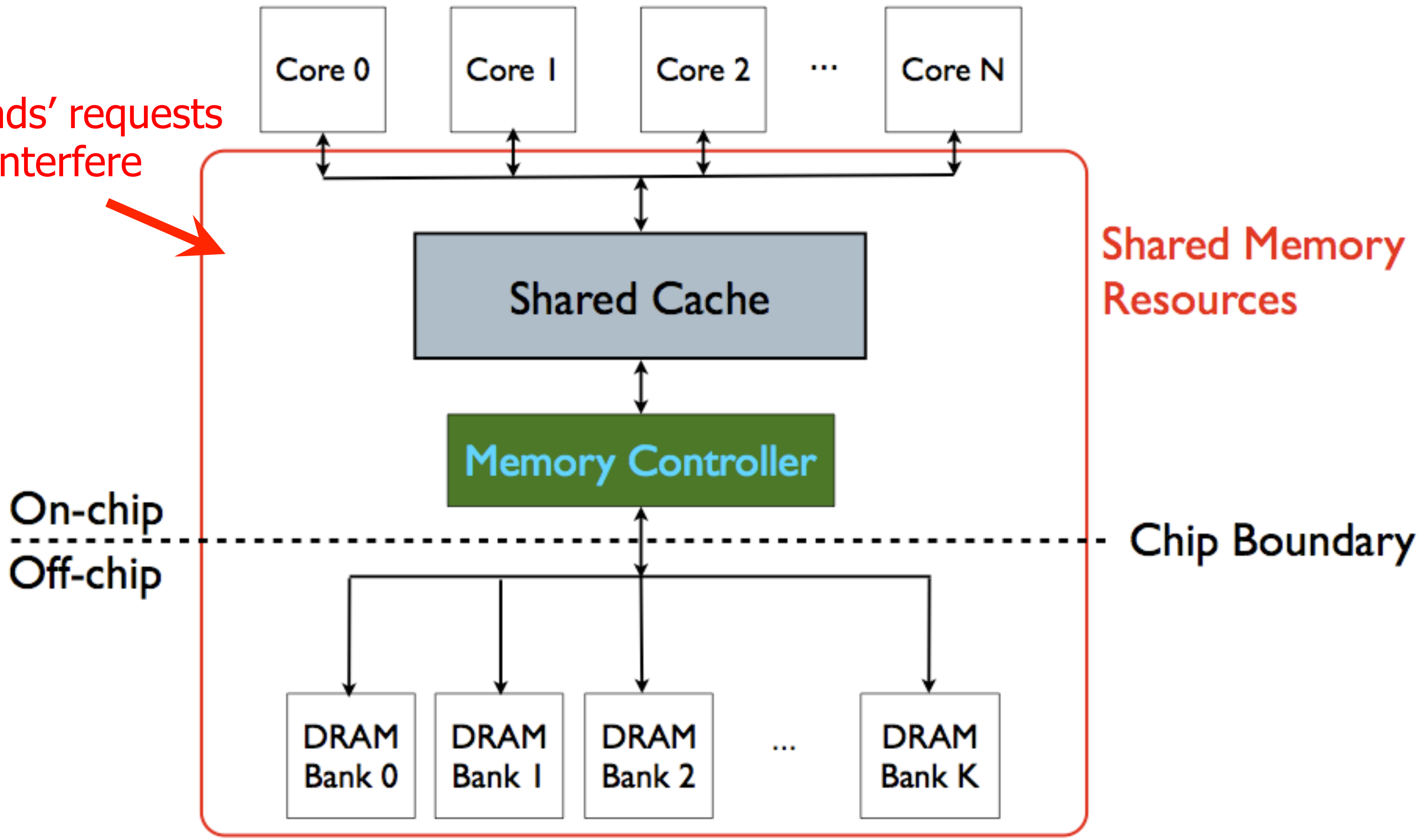


Multi-Core Chip

Shared DRAM Memory System

unfairness

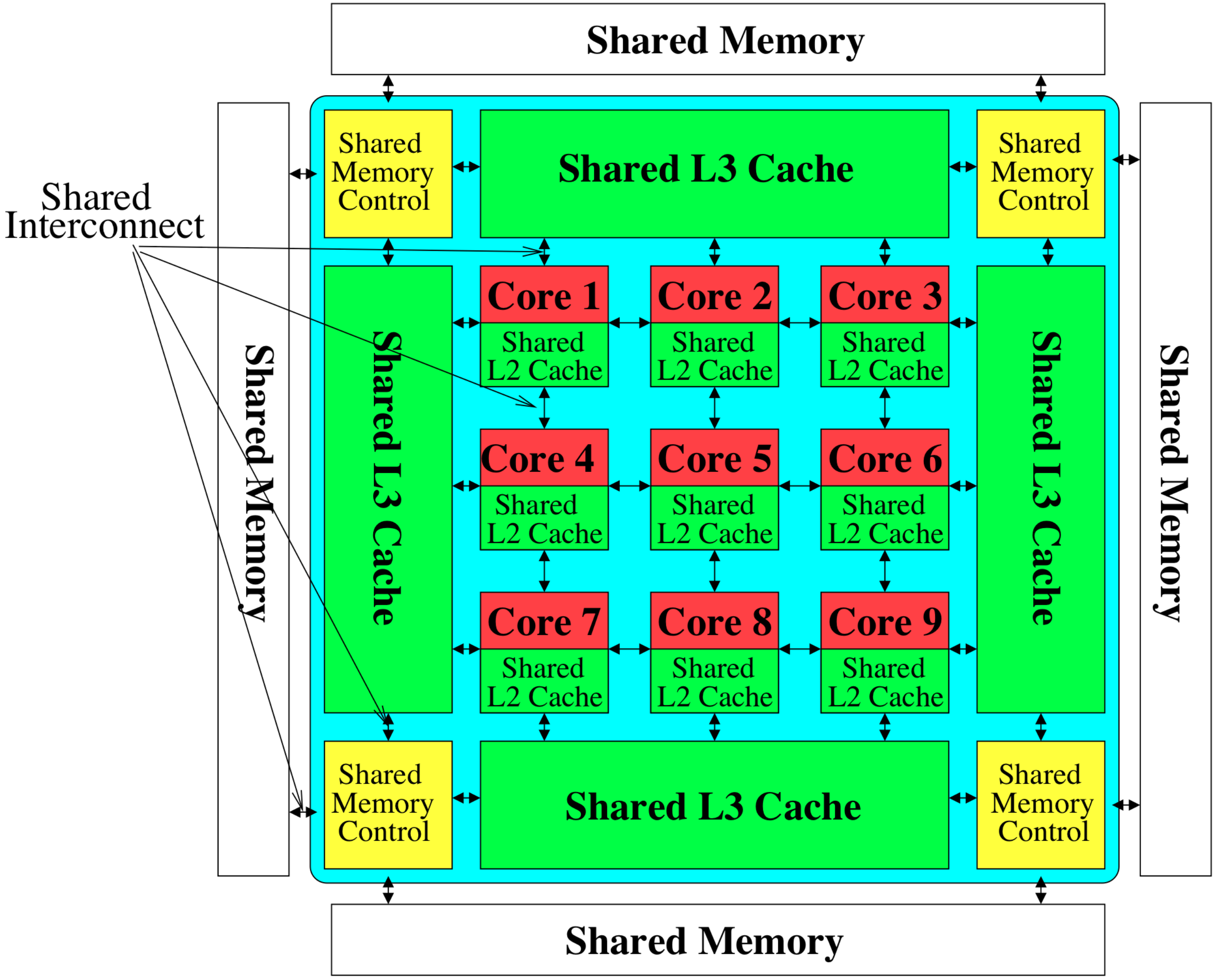
Memory System is the Major Shared Resource



threads' requests interfere

Shared Memory Resources

Much More of a Shared Resource in Future

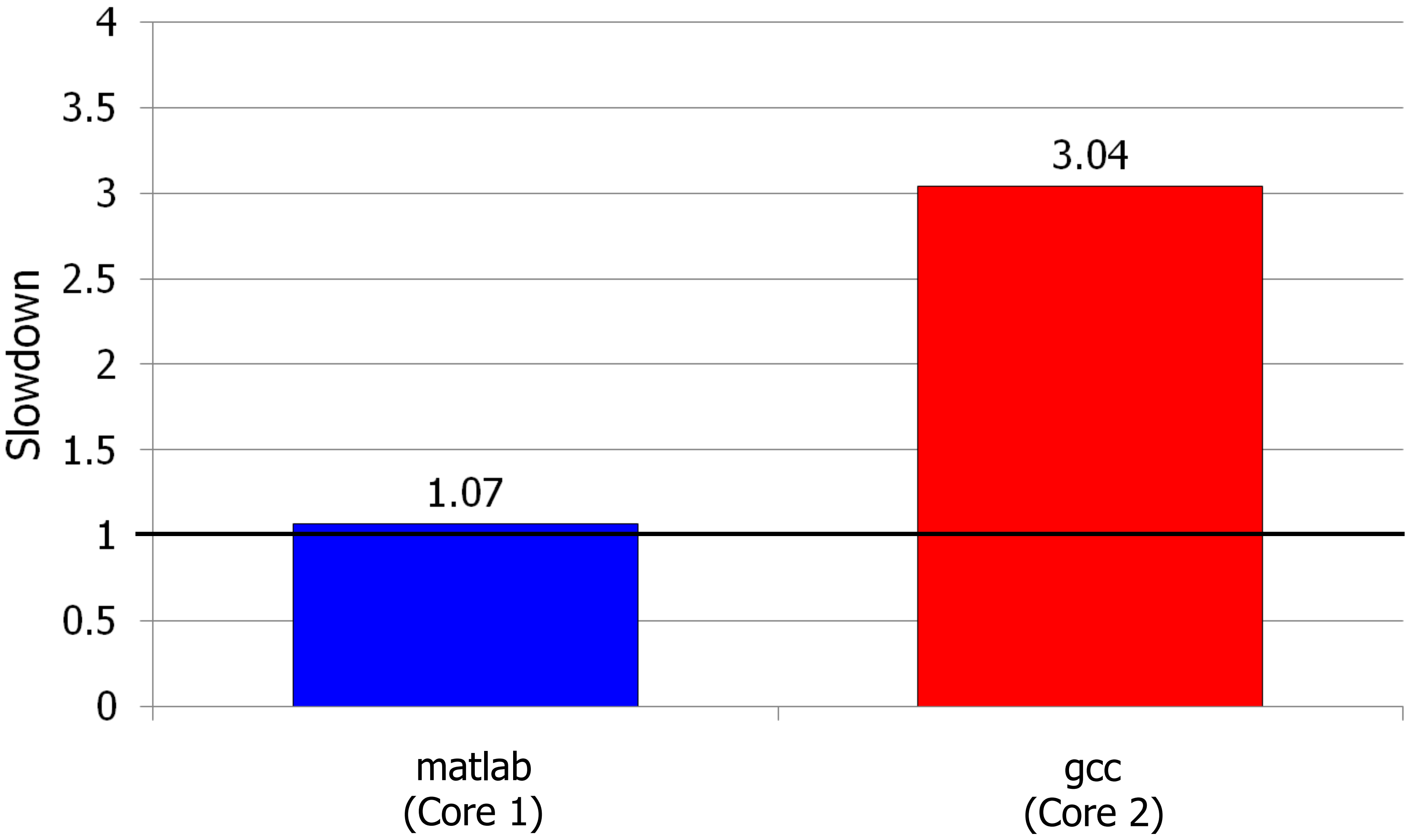


Inter-Thread/Application Interference

- Problem: Threads share the memory system, but memory system does not distinguish between threads' requests

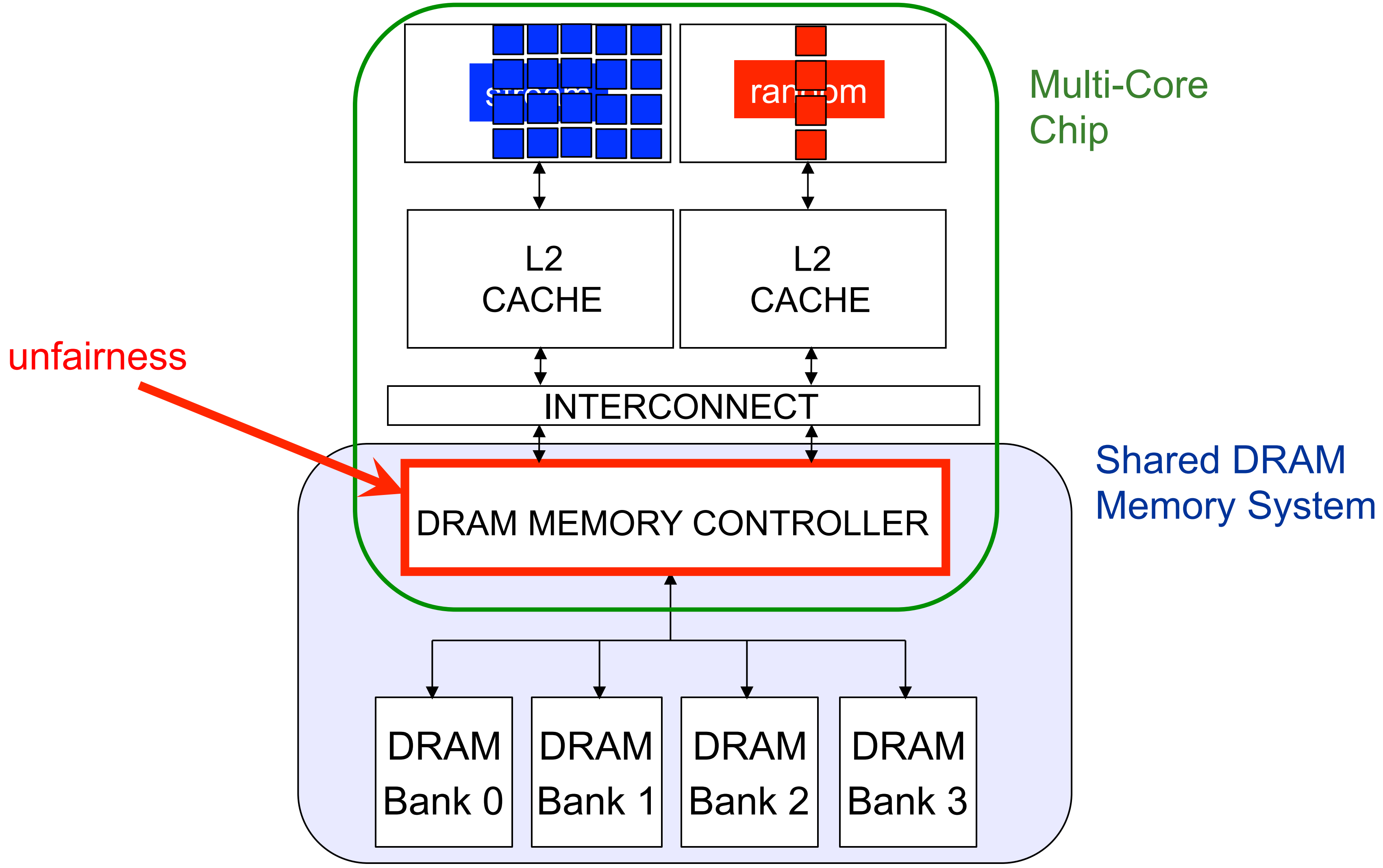
- Existing memory systems
  - Free-for-all, shared based on demand
  - Control algorithms thread-unaware and thread-unfair
  - Aggressive threads can deny service to others
  - Do not try to reduce or control inter-thread interference

Unfair Slowdowns due to Interference



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

Uncontrolled Interference: An Example



A Memory Performance Hog

```
// initialize large arrays A, B

for (j=0; j<N; j++) {
    index = j*linesize;
    A[index] = B[index];
    …
}
```

streaming

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();
    A[index] = B[index];
    …
}
```

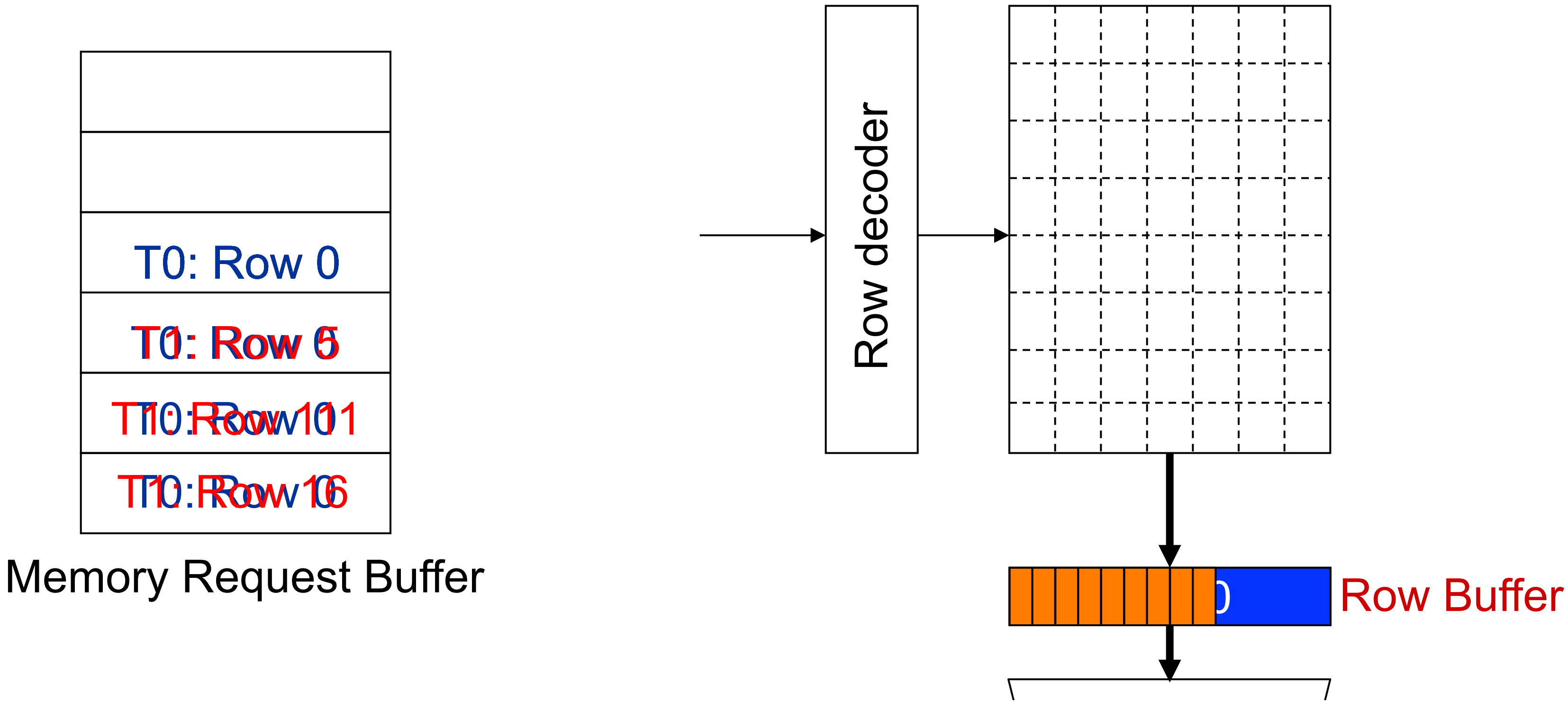
random

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]*

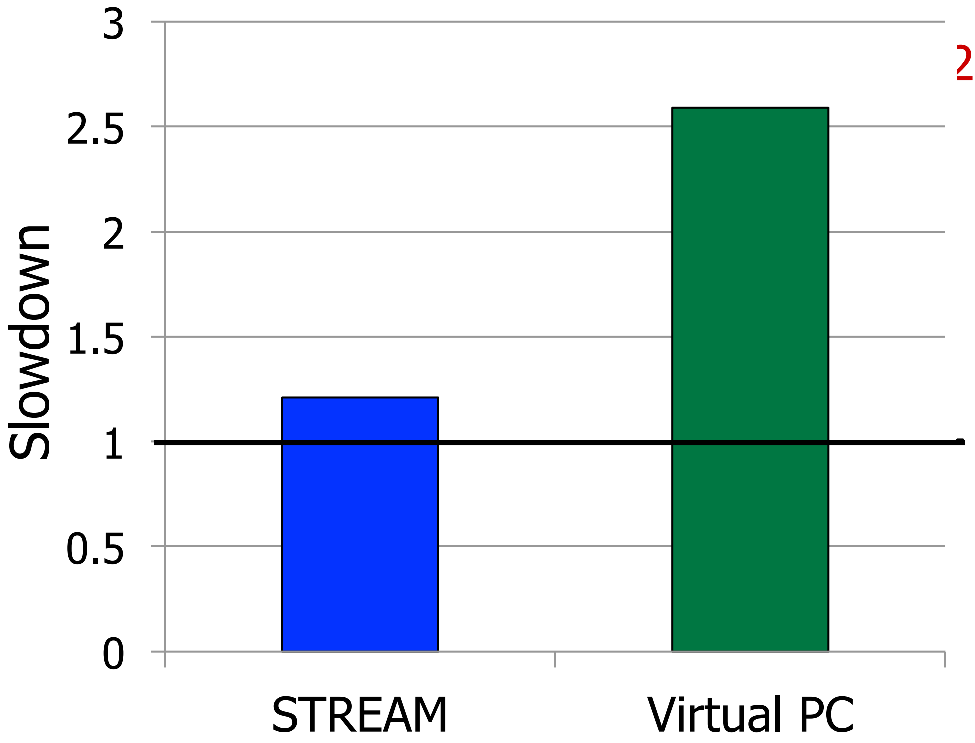
  (1) Row-hit first: Service row-hit memory accesses first

  (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

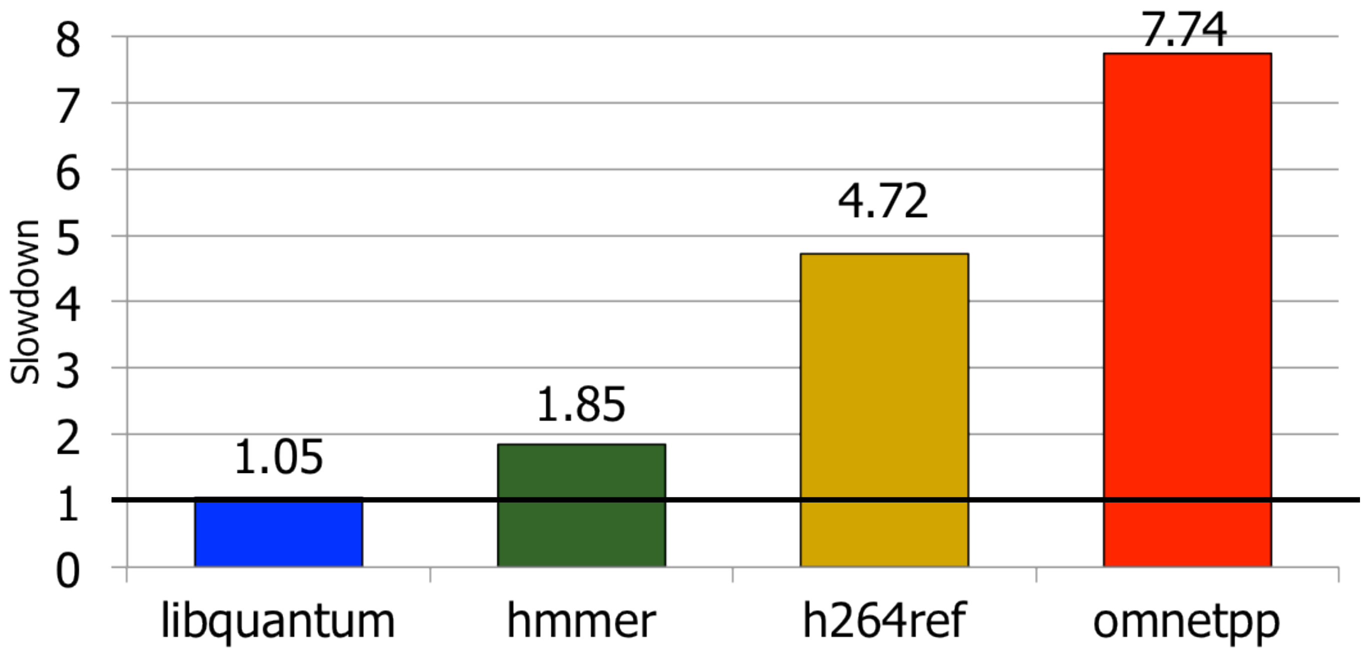


2X slowdown

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.

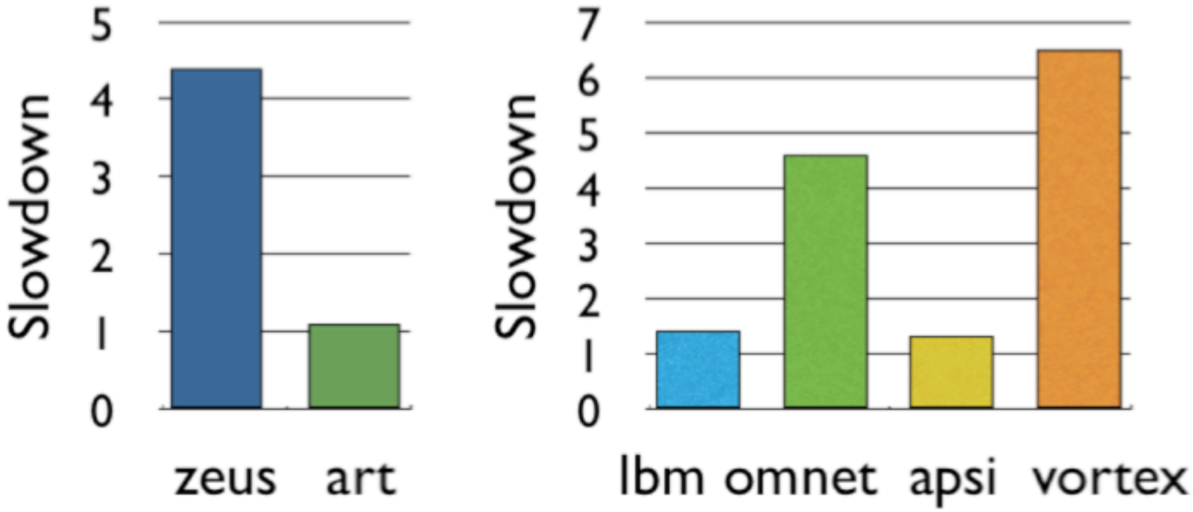
Greater Problem with More Cores



- Vulnerable to denial of service (DoS) [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

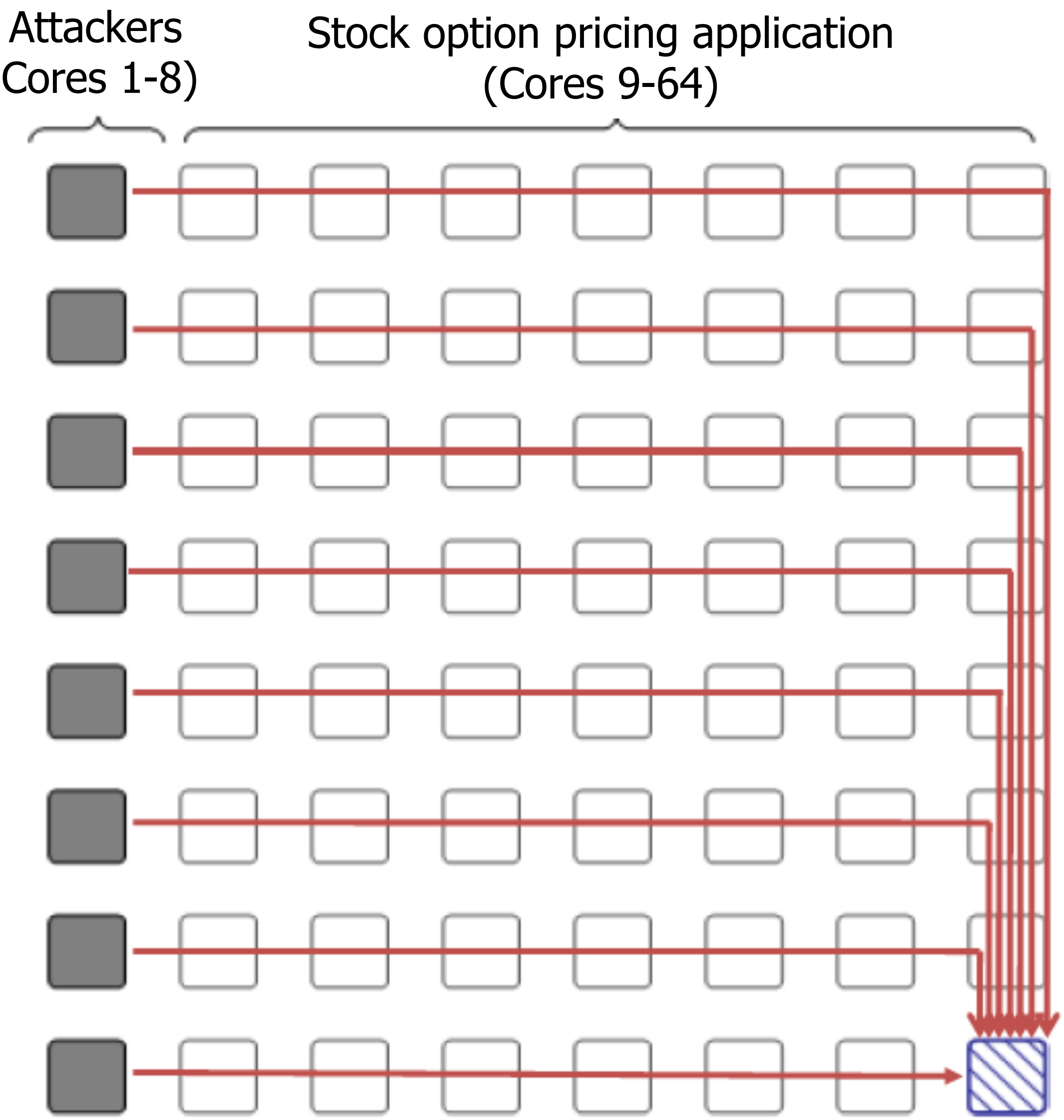
Greater Problem with More Cores



- 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

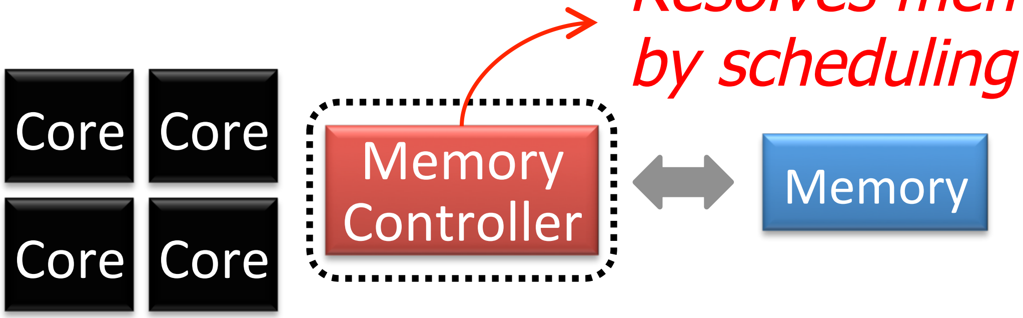
- How do we reduce inter-thread interference?
  - Improve system performance and core utilization
  - Reduce request serialization and core starvation
- How do we control inter-thread interference?
  - Provide mechanisms to enable system software to enforce QoS policies
  - While providing high system 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]
  - 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



Resolves memory contention by scheduling requests

- 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

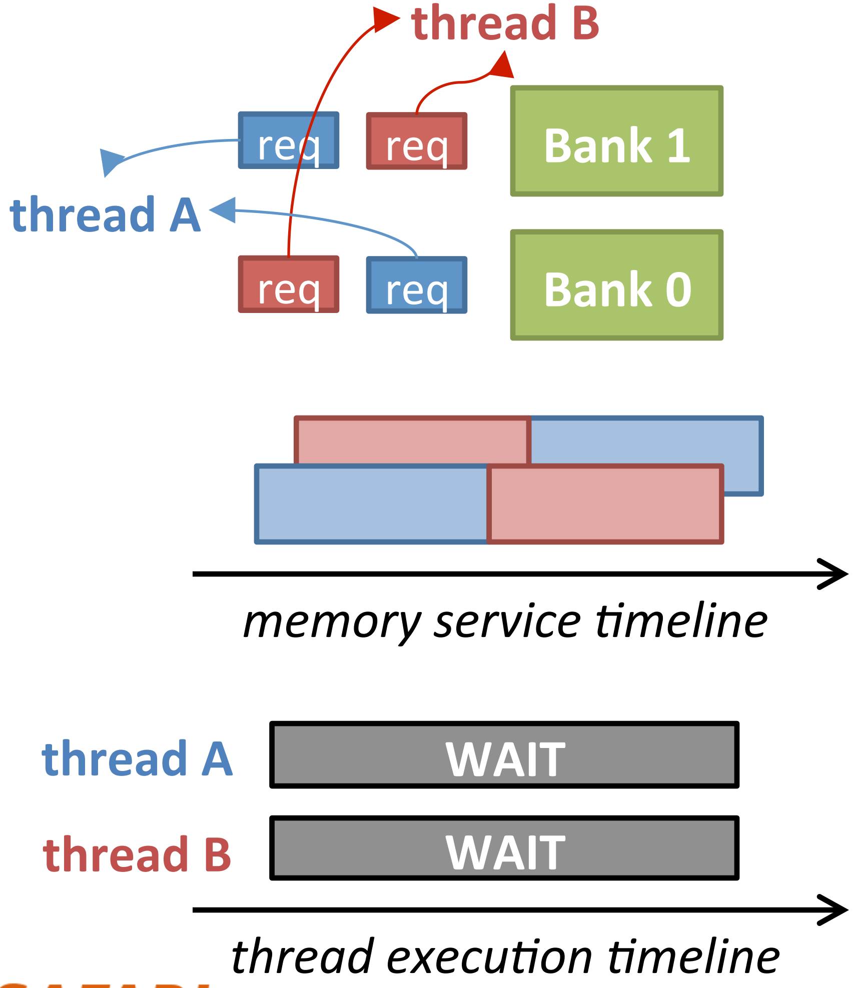
QoS-Aware Memory Scheduling: Evolution

- 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)

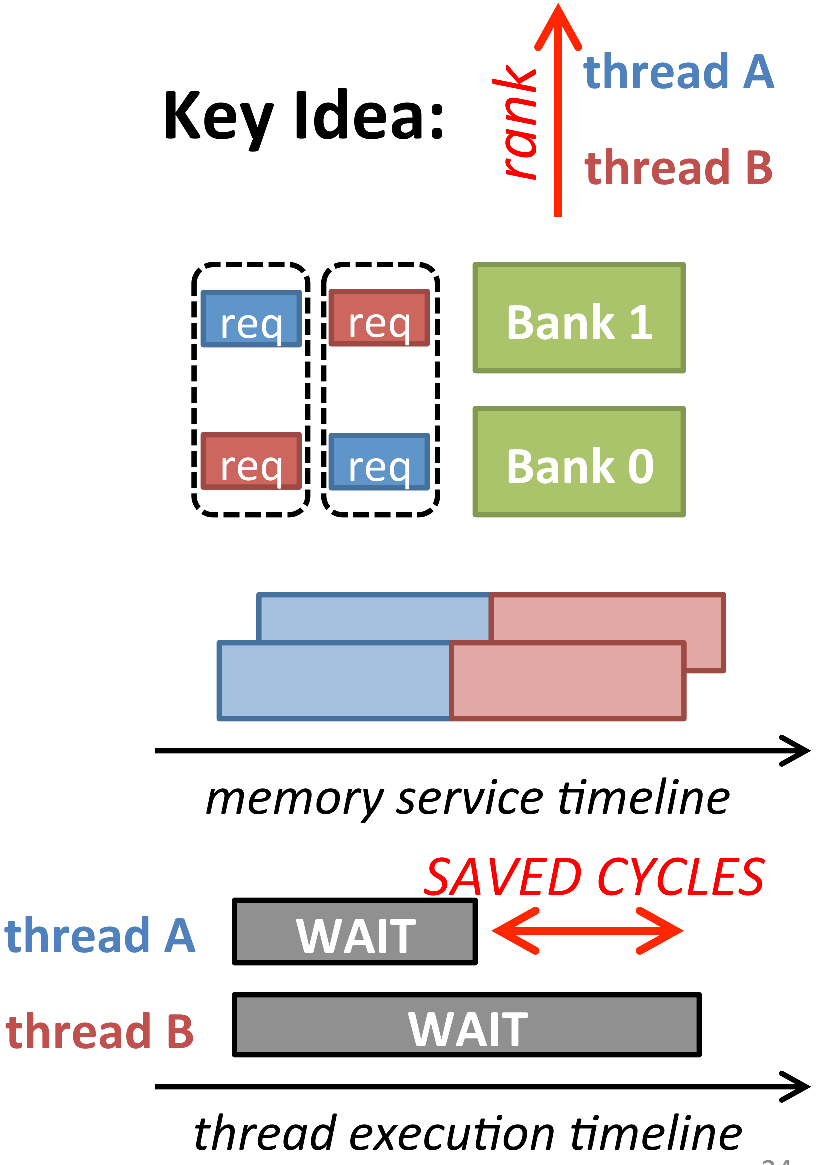
- Parallelism-aware batch scheduling [Mutlu+ ISCA'08, Top Picks'09]
  - Idea: Rank threads and service in rank order (to preserve bank parallelism); batch requests to prevent starvation

- ATLAS memory scheduler [Kim+ HPCA'10]

Within-Thread Bank Parallelism



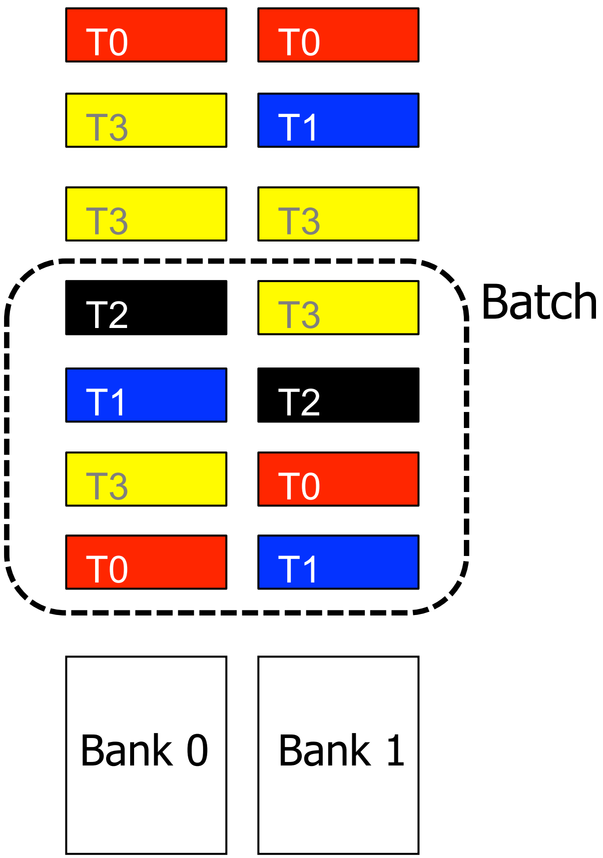
Key Idea:



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

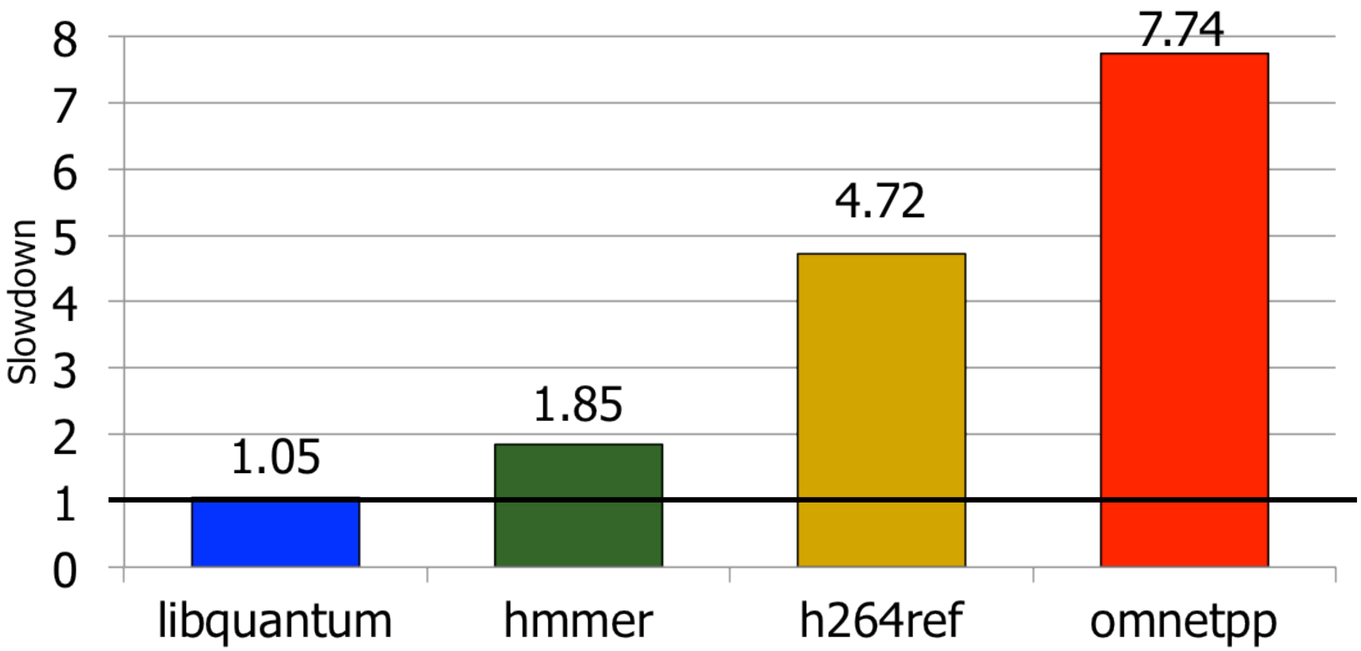
- DRAM-Aware last-level cache policies [Lee+ HPS Tech Report'10] [Lee+ HPS Tech Report'10]
  - Idea: Design cache eviction and replacement policies such that they proactively exploit the state of the memory controller and DRAM (e.g., proactively evict data from the cache that hit in open rows)
  - Takeaway: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

STFM Micro 2007 Talk

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_{shared}/ST_{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]

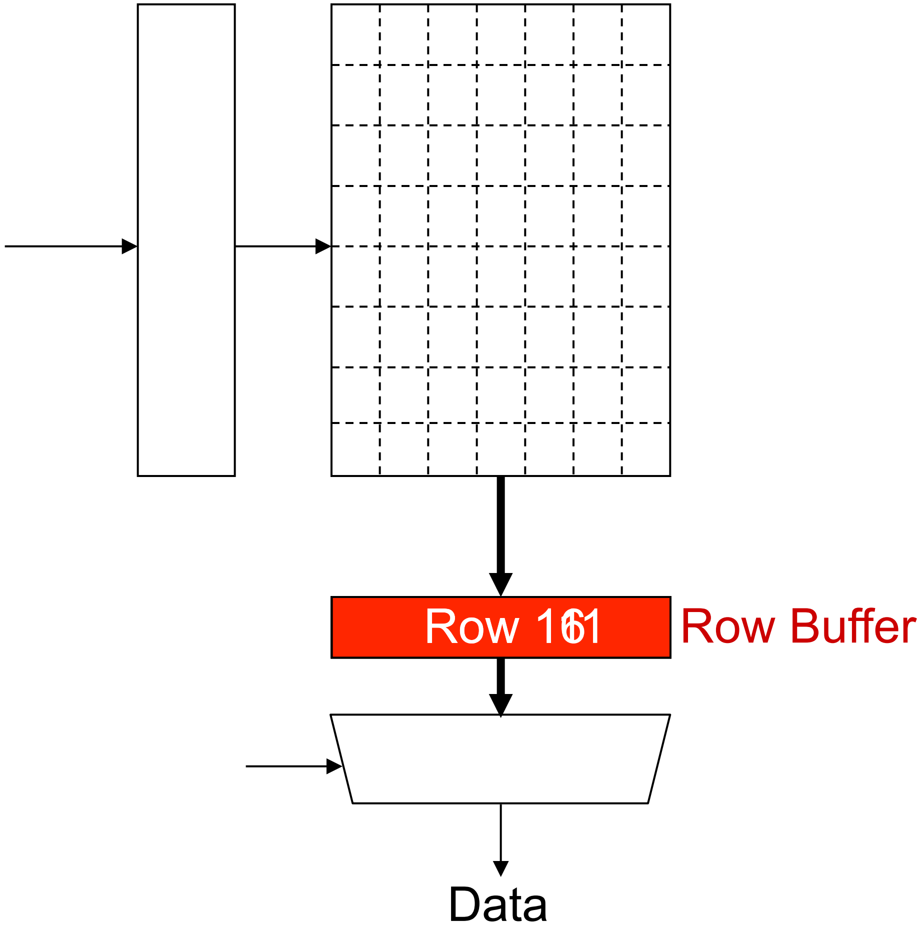
- For each thread, the DRAM controller
  - Tracks ST_{shared}
  - Estimates ST_{alone}

- Each cycle, the DRAM controller
  - Computes Slowdown = ST_{shared}/ST_{alone} for threads with legal requests
  - Computes unfairness = MAX Slowdown / MIN Slowdown

- If 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
T1: Row 16



T0 Slowdown	1.04
T1 Slowdown	1.06
Unfairness	1.06
α	1.05

STFM Pros and Cons

- Upsides:
  - First work on fair 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)

PAR-BS ISCA 2008 Talk

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

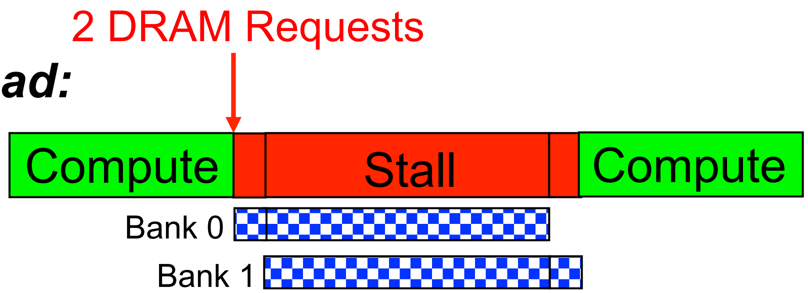
- Effective only if the DRAM controller actually services the multiple requests in parallel in DRAM banks

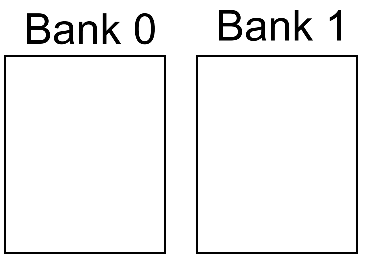
- Multiple threads share the DRAM controller
- DRAM controllers are not aware of a thread's MLP
  - Can service each thread's outstanding requests serially, not in parallel

Bank Parallelism of a Thread

Single Thread:

Thread A :





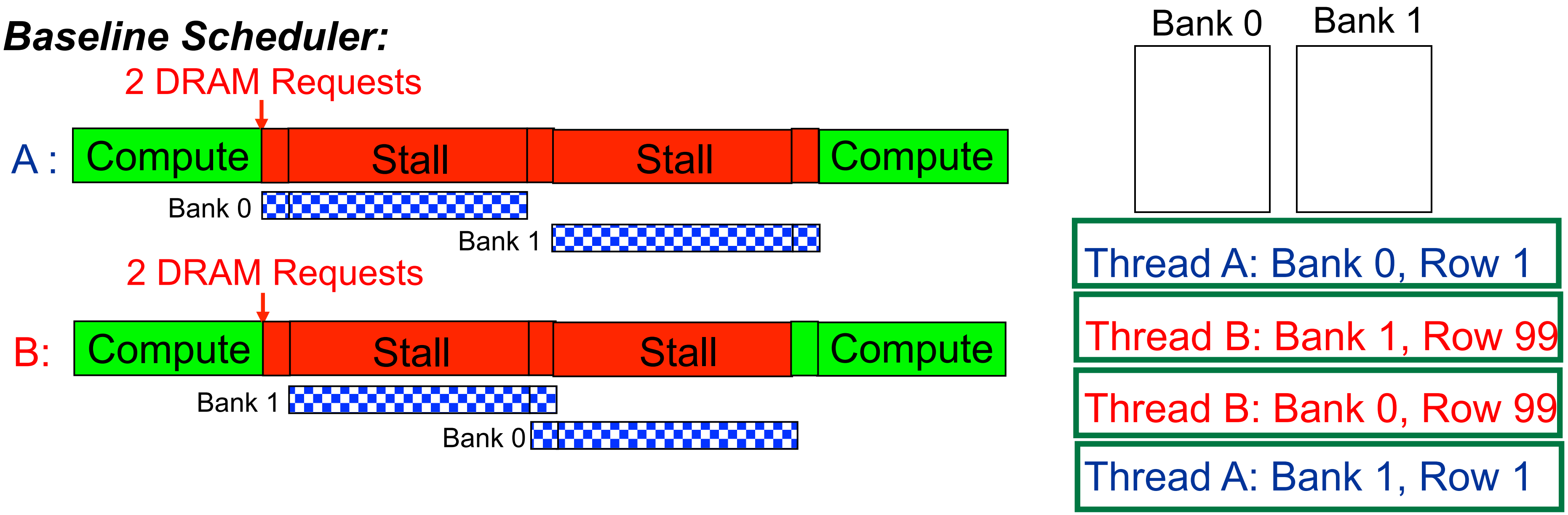
Thread A: Bank 0, Row 1

Thread A: Bank 1, Row 1

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

Bank Parallelism Interference in DRAM

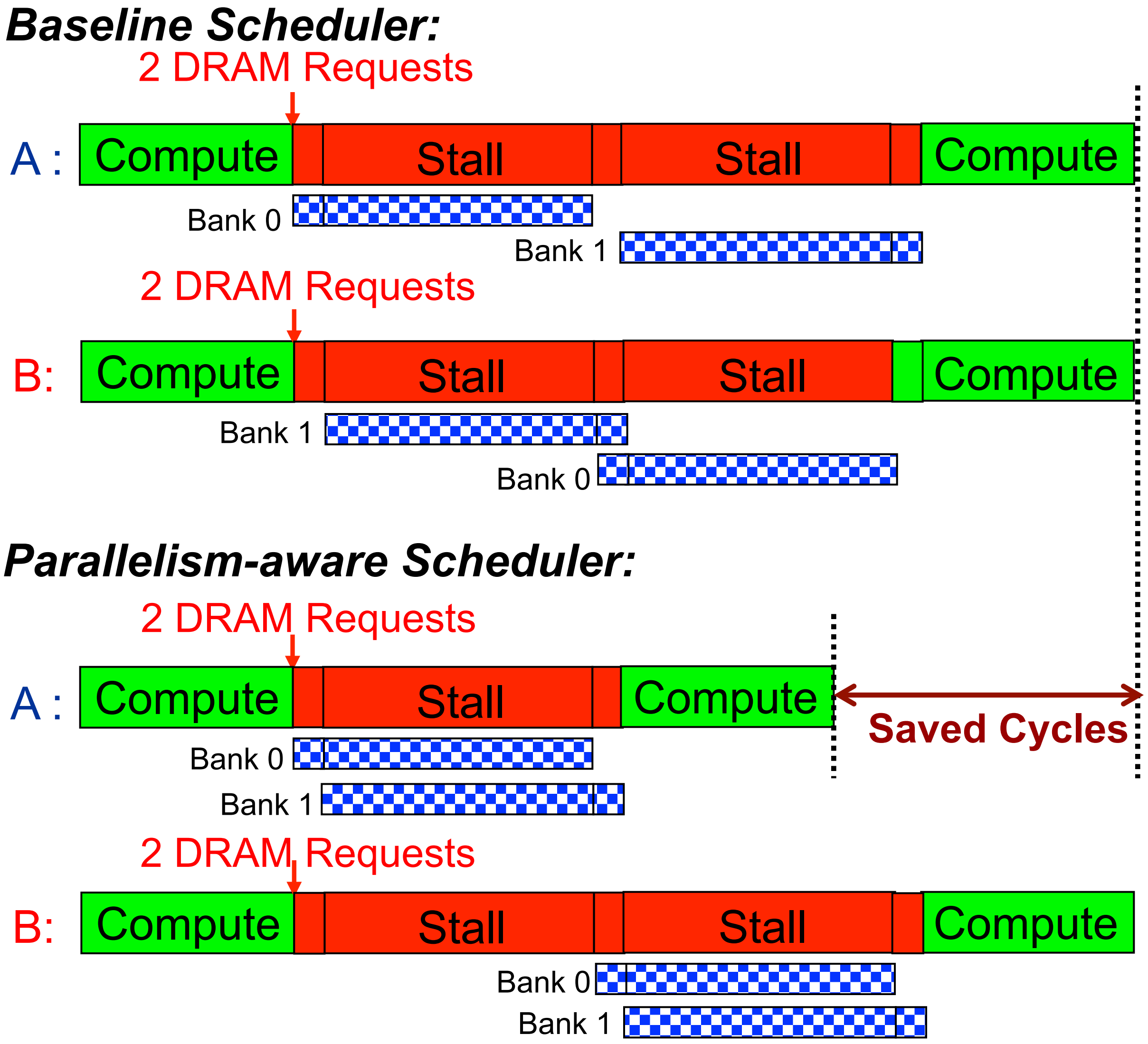
Baseline Scheduler:



Bank access latencies of each thread serialized

Each thread stalls for ~TWO bank access latencies

Parallelism-Aware Scheduler



Baseline Scheduler:

2 DRAM Requests

A : Compute | Stall | Stall | Compute

Bank 0

Bank 1

2 DRAM Requests

B: Compute | Stall | Stall | Compute

Bank 1

Bank 0

Parallelism-aware Scheduler:

2 DRAM Requests

A : Compute | Stall | Compute

Bank 0

Bank 1

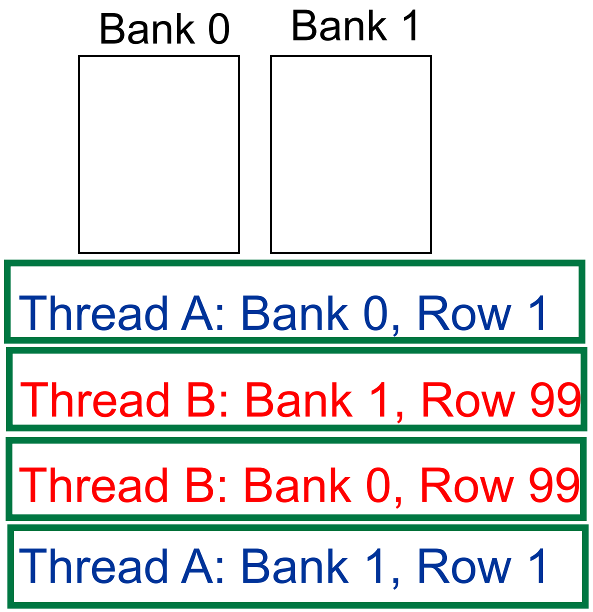
Saved Cycles

2 DRAM Requests

B: Compute | Stall | Stall | Compute

Bank 0

Bank 1



Bank 0

Bank 1

Thread A: Bank 0, Row 1

Thread B: Bank 1, Row 99

Thread B: Bank 0, Row 99

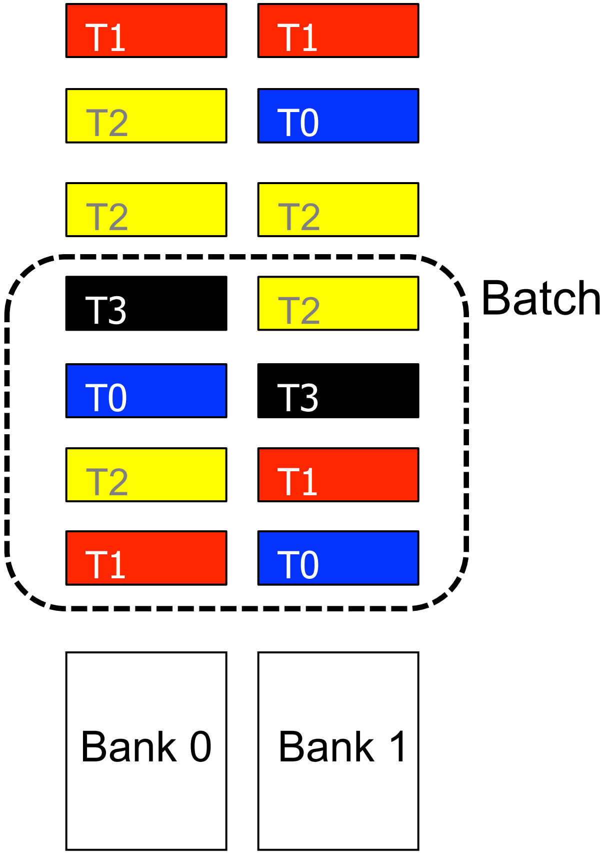
Thread A: Bank 1, Row 1

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



Mutlu and Moscibroda, "Parallelism-Aware Batch Scheduling," ISCA 2008.

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

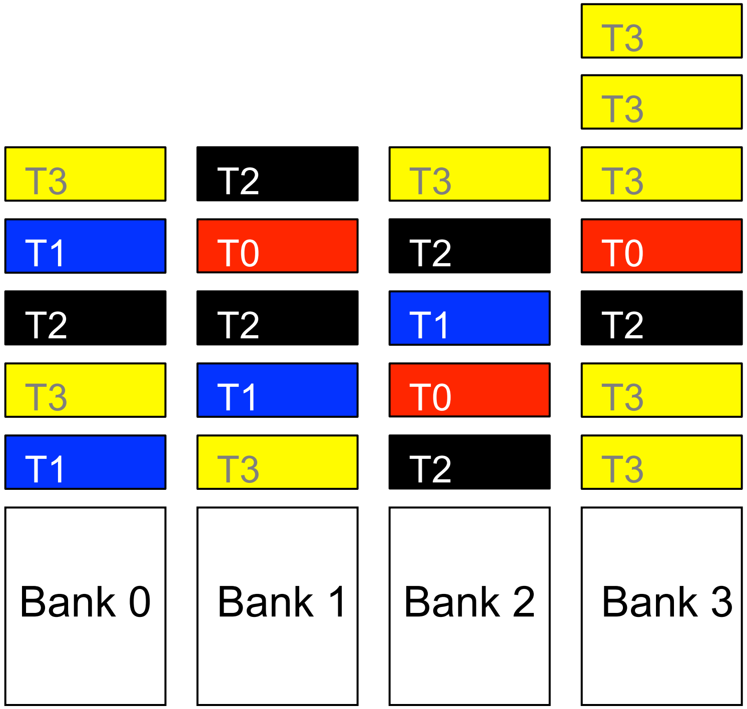
- Maximize system throughput
  - Minimize average stall-time of threads within the batch
- Minimize unfairness (Equalize the slowdown of threads)
  - Service threads with inherently low stall-time early in the batch
  - Insight: delaying memory non-intensive threads results in high slowdown

- 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

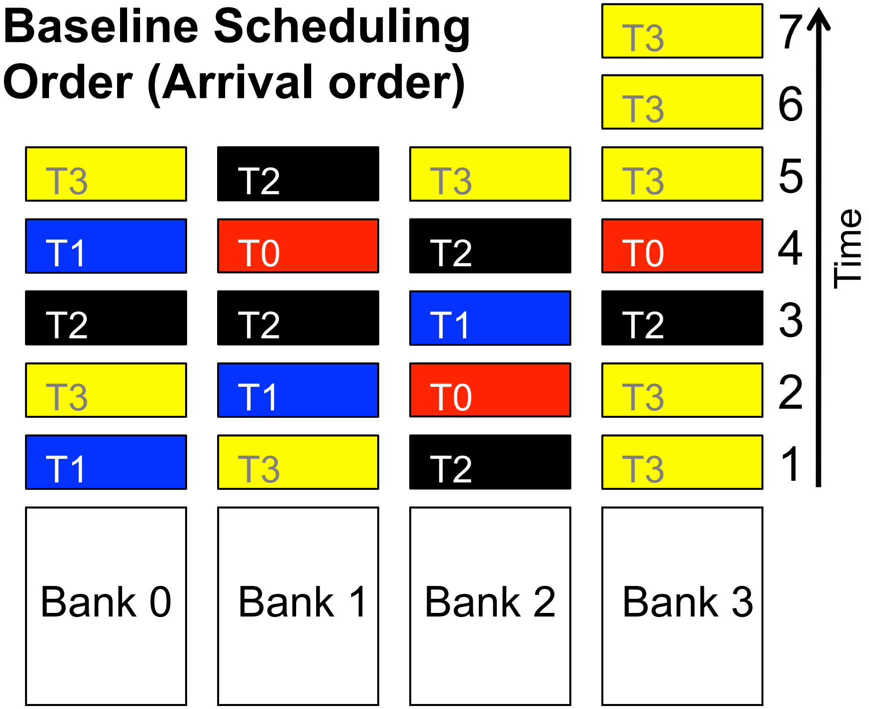


	max-bank-load	total-load

Ranking:
T0 > T1 > T2 > T3

Example Within-Batch Scheduling Order

Baseline Scheduling Order (Arrival order)

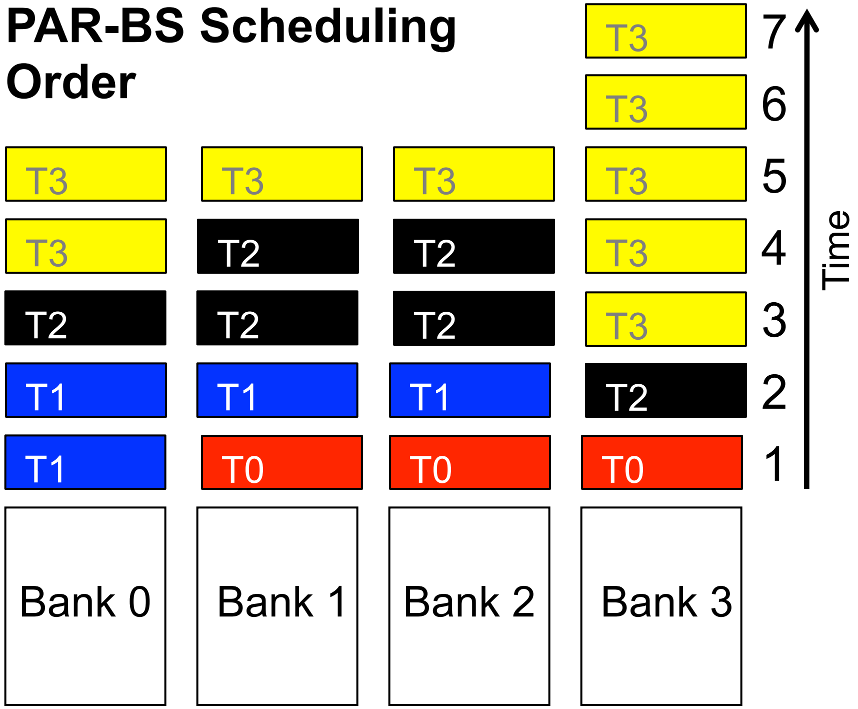


Time	Bank 0	Bank 1	Bank 2	Bank 3
7				T3
6				T3
5	T3	T2	T3	T3
4	T1	T0	T2	T0
3	T2	T2	T1	T2
2	T3	T1	T0	T3
1	T1	T3	T2	T3

	T0	T1	T2	T3
Stall times				

AVG: 5 bank access latencies

PAR-BS Scheduling Order



Time	Bank 0	Bank 1	Bank 2	Bank 3
7				T3
6				T3
5	T3	T3	T3	T3
4	T3	T2	T2	T3
3	T2	T2	T2	T3
2	T1	T1	T1	T2
1	T1	T0	T0	T0

Ranking: T0 > T1 > T2 > T3

	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)

  (4) Oldest first

  Parallelism-aware within-batch scheduling

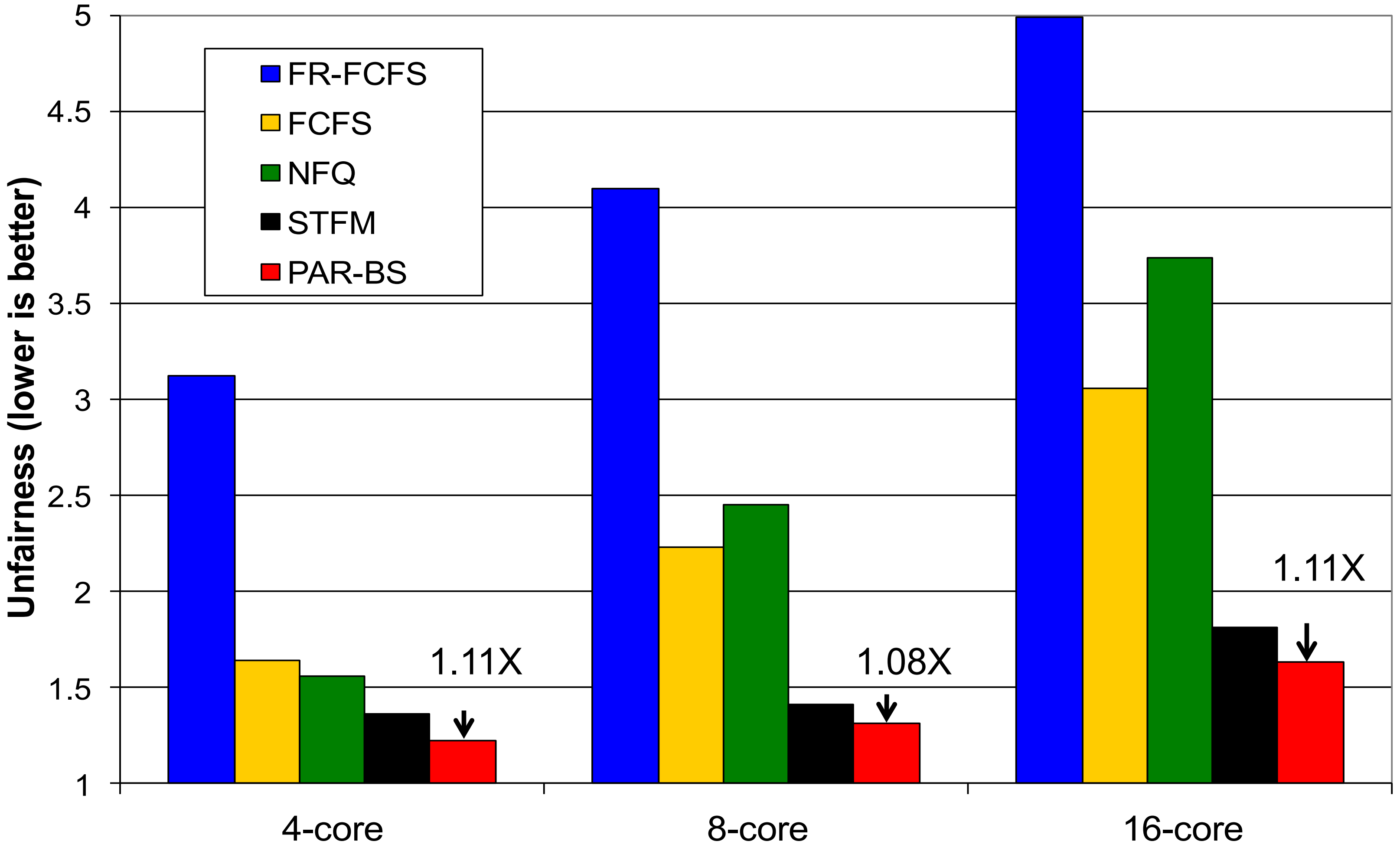
- Three properties:
  - Exploits row-buffer locality **and** intra-thread bank parallelism
  - Work-conserving
    - Services unmarked requests to banks without marked requests
  - Marking-Cap is important
    - Too small cap: destroys row-buffer locality
    - Too large cap: penalizes memory non-intensive threads
- Many more trade-offs analyzed in the paper

Hardware Cost

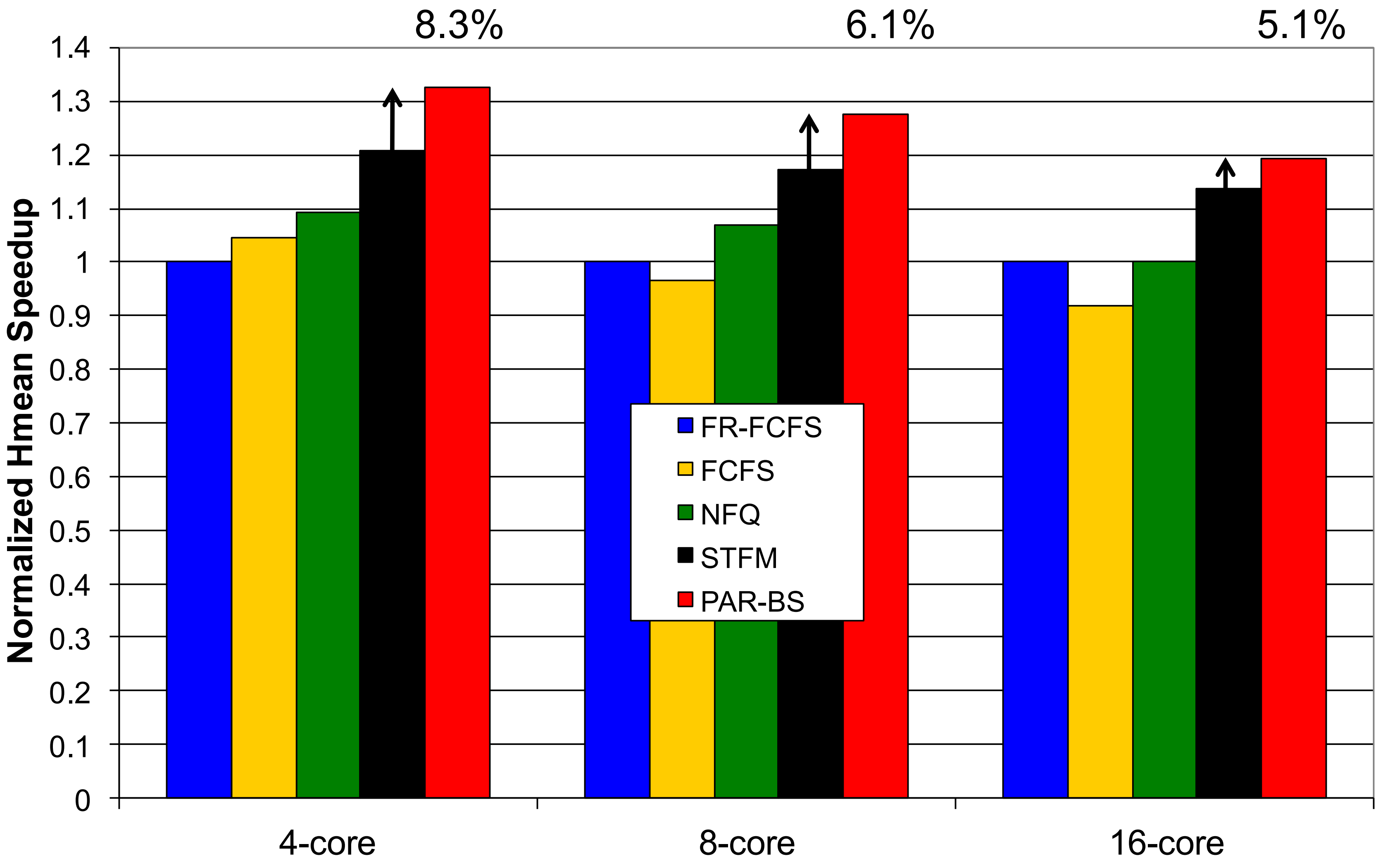
- <1.5KB storage cost for
  - 8-core system with 128-entry memory request buffer
- No complex operations (e.g., divisions)
- Not on the critical path
  - Scheduler makes a decision only every DRAM cycle

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:
  - First work to identify the notion of bank parallelism destruction across multiple threads
  - Simple mechanism
- Downsides:
  - Implementation in multiple controllers needs coordination for best performance → too frequent coordination since batching is done frequently
  - Does not always prioritize the latency-sensitive applications

ATLAS Memory Scheduler

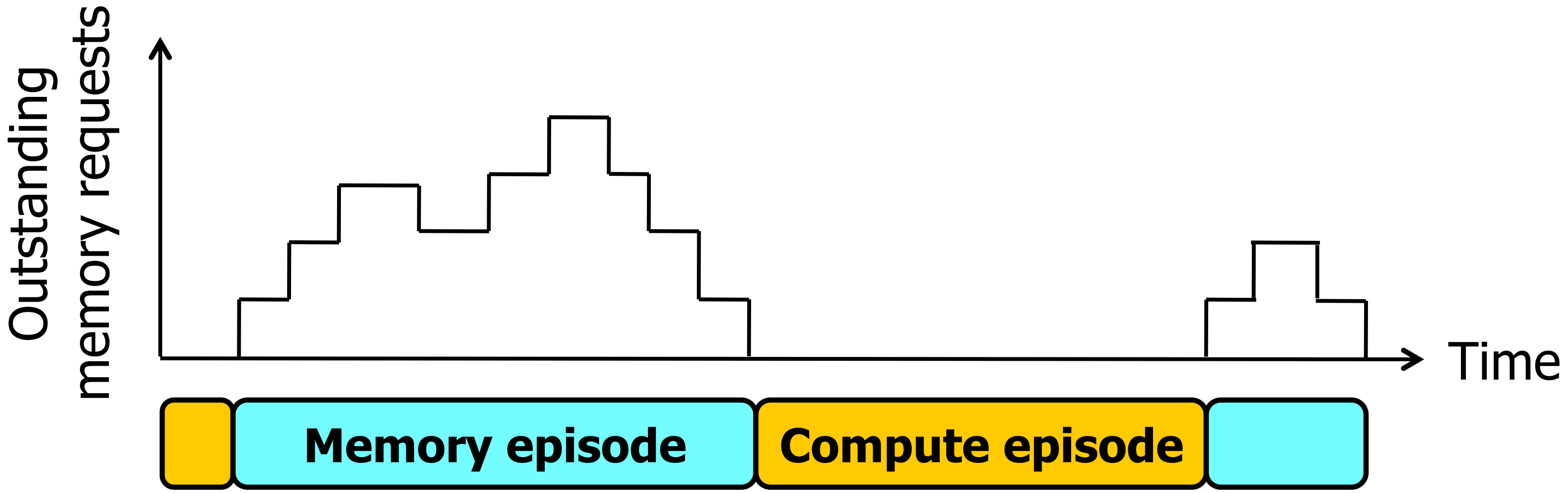
Yoongu Kim, Dongsu Han, Onur Mutlu, and Mor Harchol-Balter,
"ATLAS: A Scalable and High-Performance Scheduling Algorithm for Multiple Memory Controllers"
16th International Symposium on High-Performance Computer Architecture (***HPCA***),
Bangalore, India, January 2010. Slides (pptx)

ATLAS HPCA 2010 Talk

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**



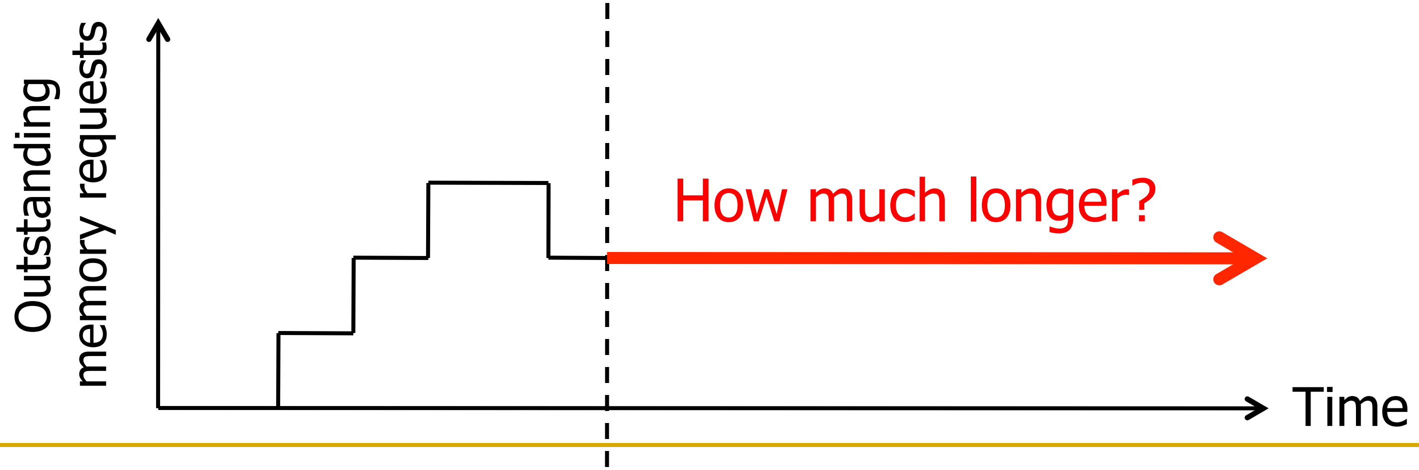
<u>Goal</u>: Minimize time spent in memory episodes

How to Minimize Memory Episode Time

Prioritize thread whose memory episode will end the soonest

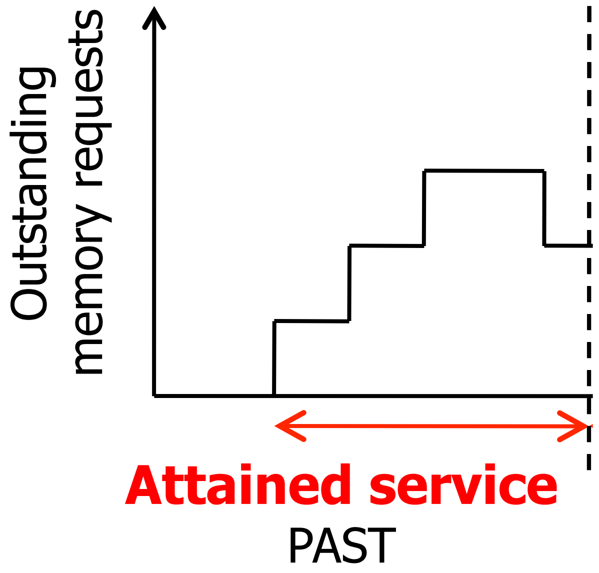
- Minimizes time spent in memory episodes across all threads
- Supported by queueing theory:
  - Shortest-Remaining-Processing-Time scheduling is optimal in single-server queue

Remaining length of a memory episode?



Predicting Memory Episode Lengths

We discovered: past is excellent predictor for future

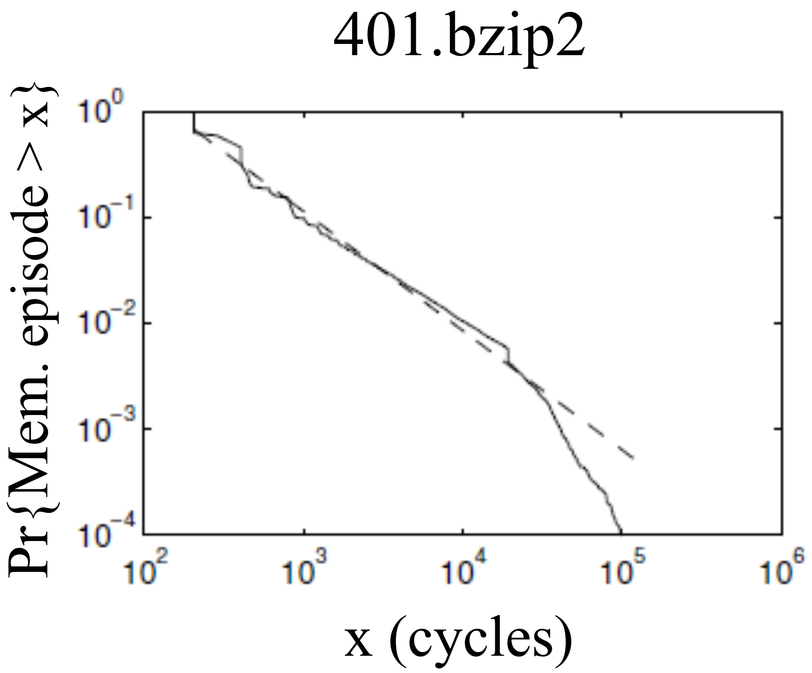


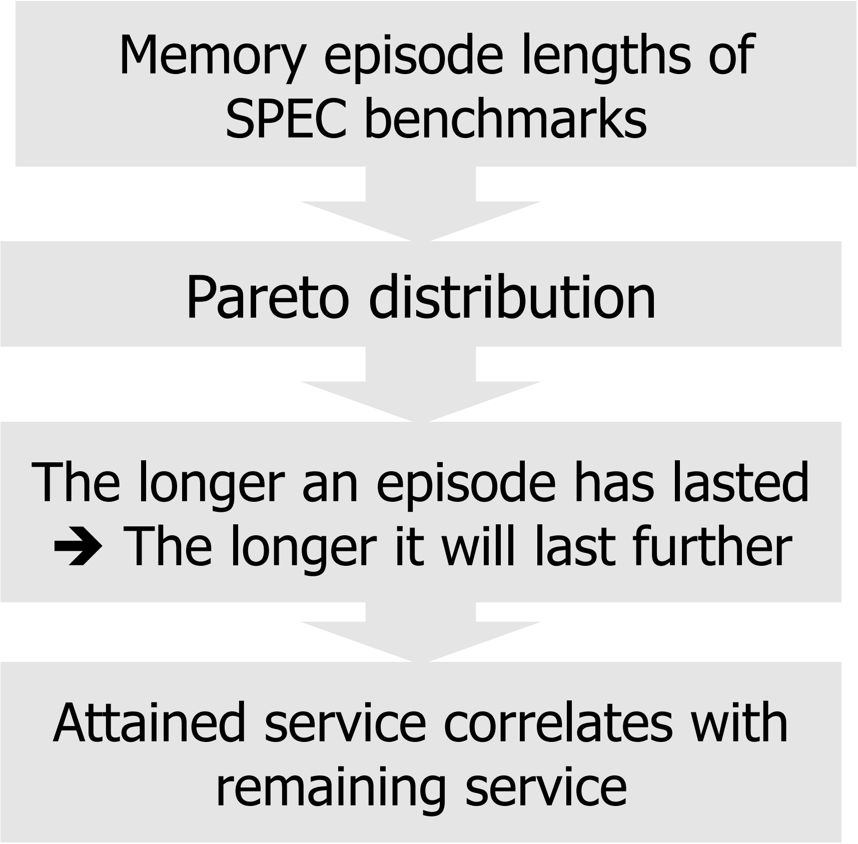
Large **attained service** ➔ Large expected **remaining service**

Q: Why?

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

Pareto Distribution of Memory Episode Lengths





Memory episode lengths of SPEC benchmarks

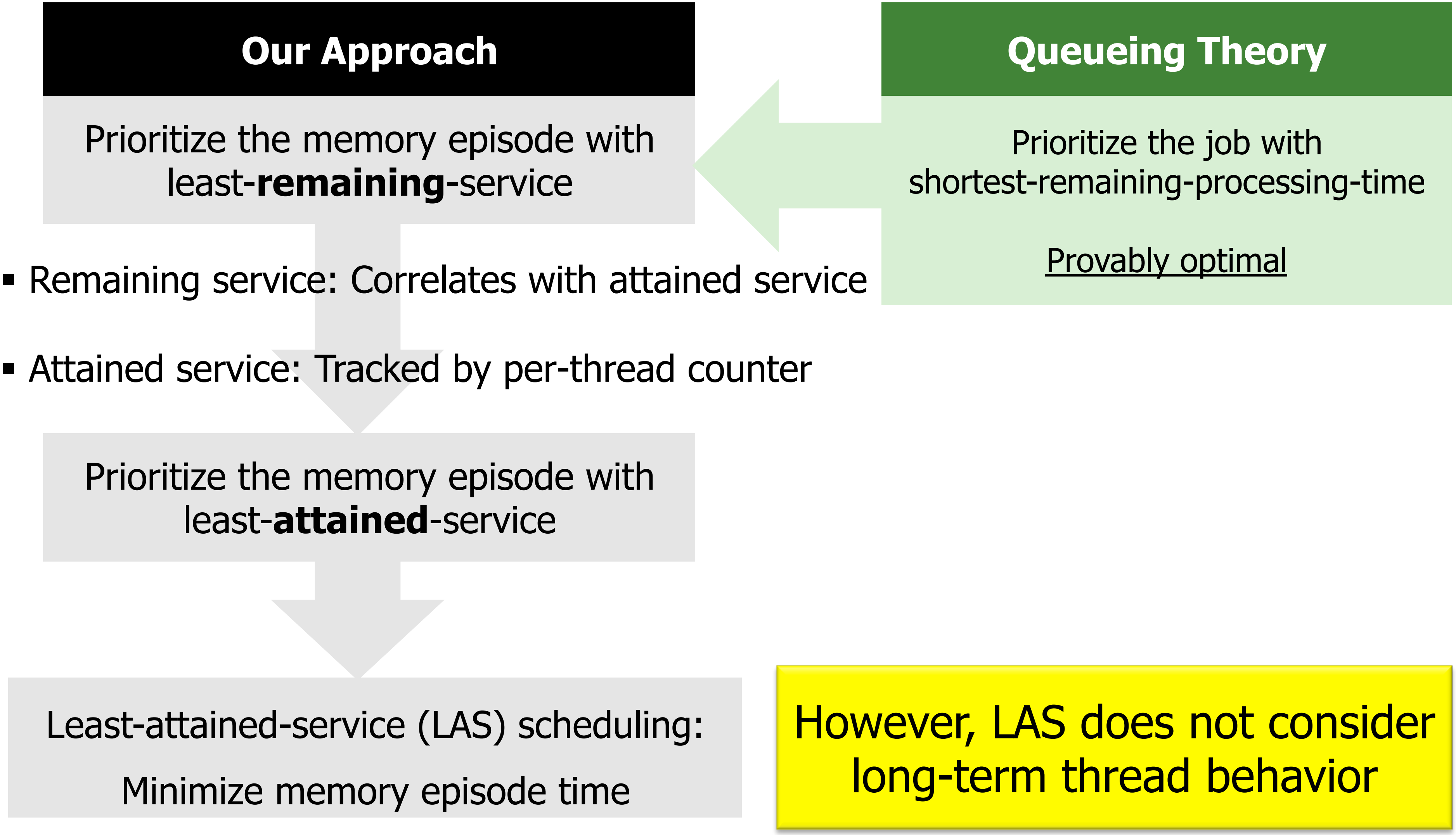
Pareto distribution

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

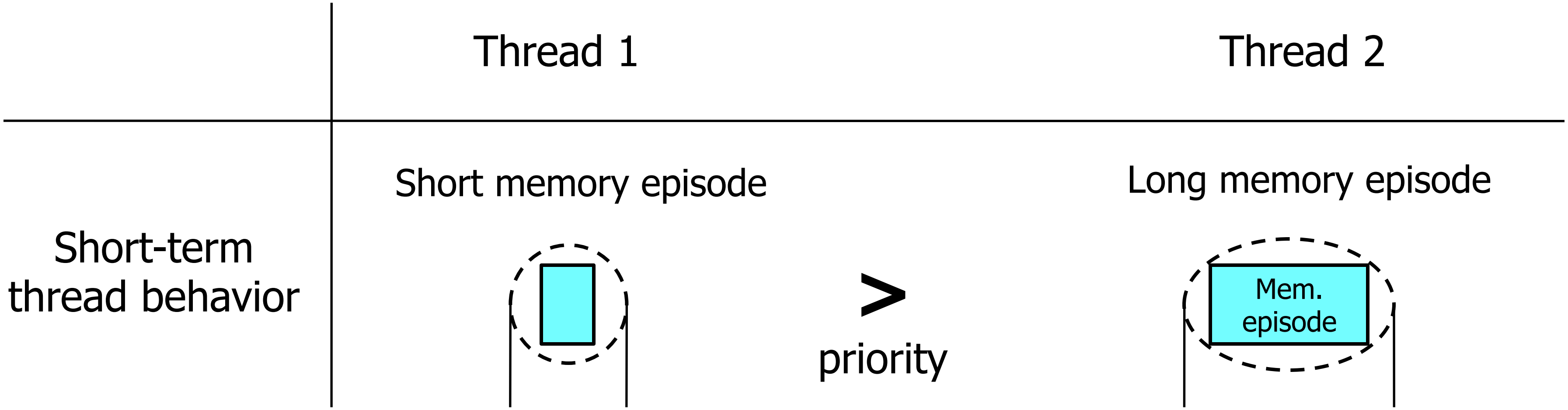
Queueing Theory

Prioritize the job with shortest-remaining-processing-time

Provably optimal

However, LAS does not consider long-term thread behavior

Long-Term Thread Behavior



Thread 1

Thread 2

Short-term thread behavior

Short memory episode

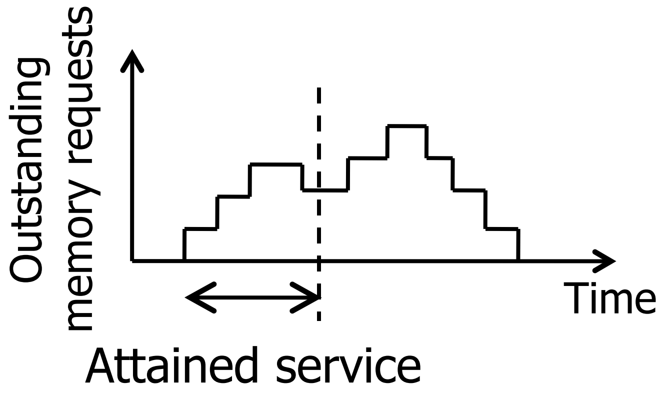
Long memory episode

> priority

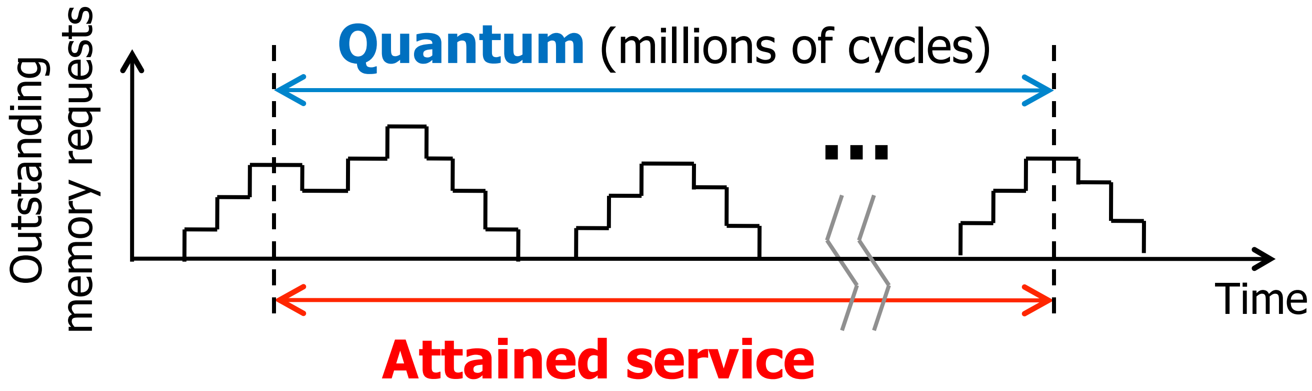
Prioritizing Thread 2 is more beneficial:
results in very long stretches of compute episodes

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 thread’s AS during only the i-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 α ➔ *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

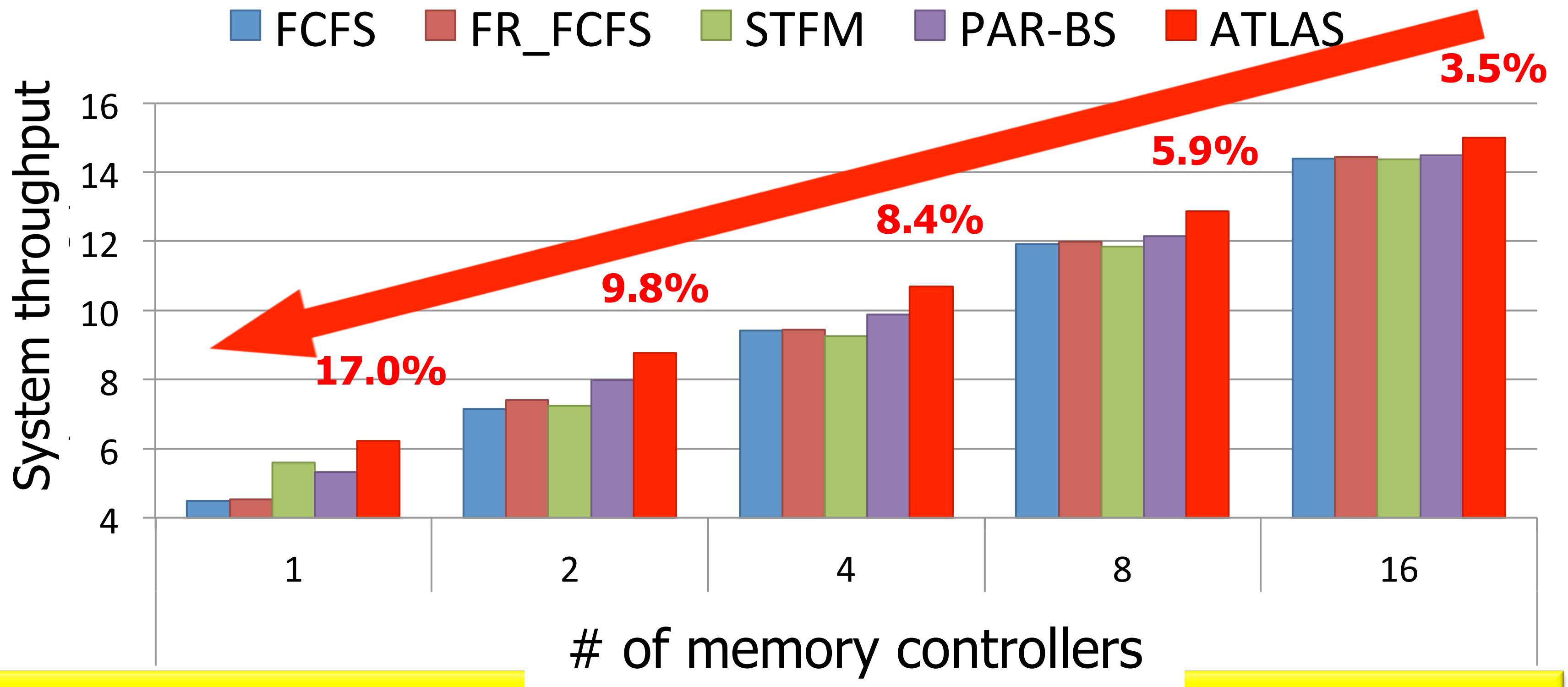
- Request prioritization order

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?

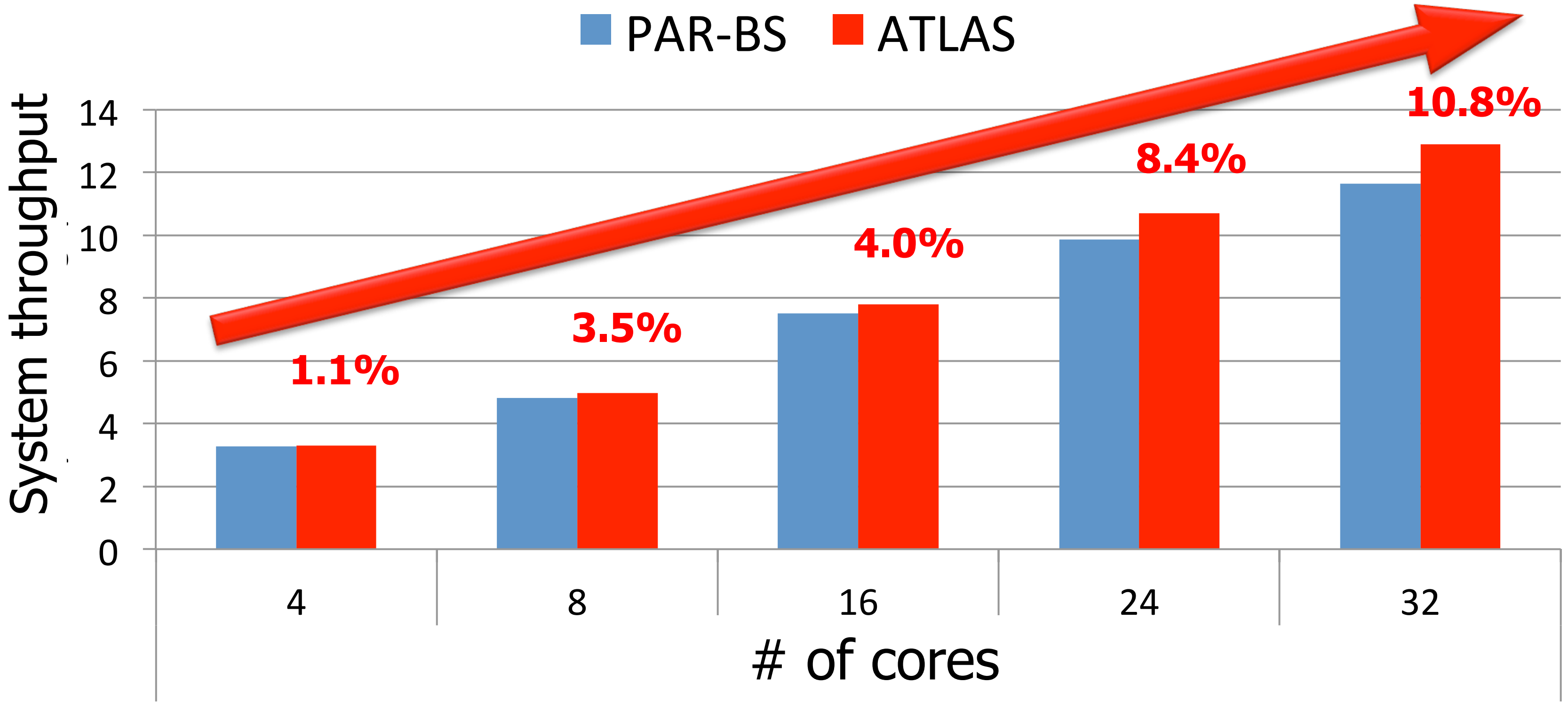
System Throughput: 24-Core System

System throughput = ∑ 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	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

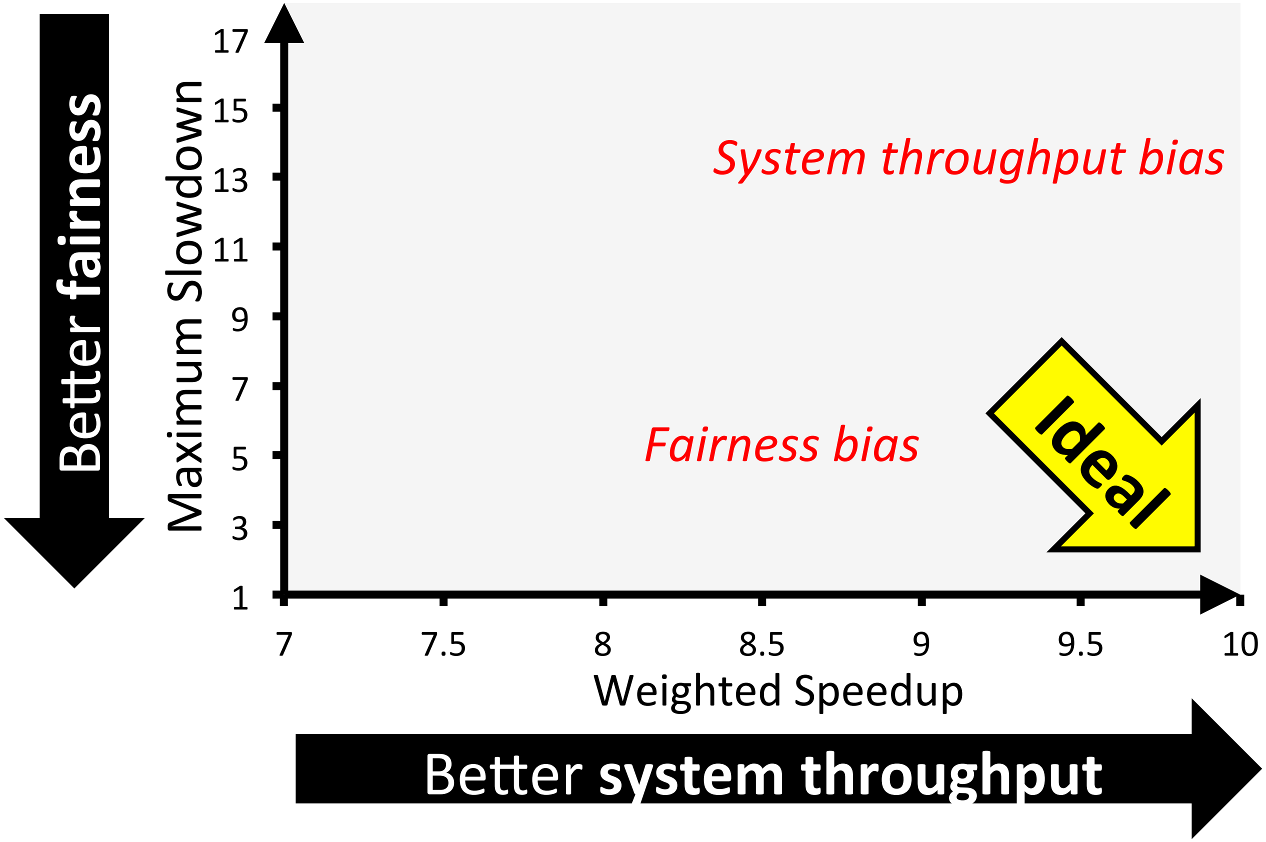
TCM:
Thread Cluster Memory Scheduling

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

Previous Scheduling Algorithms are Biased

24 cores, 4 memory controllers, 96 workloads



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

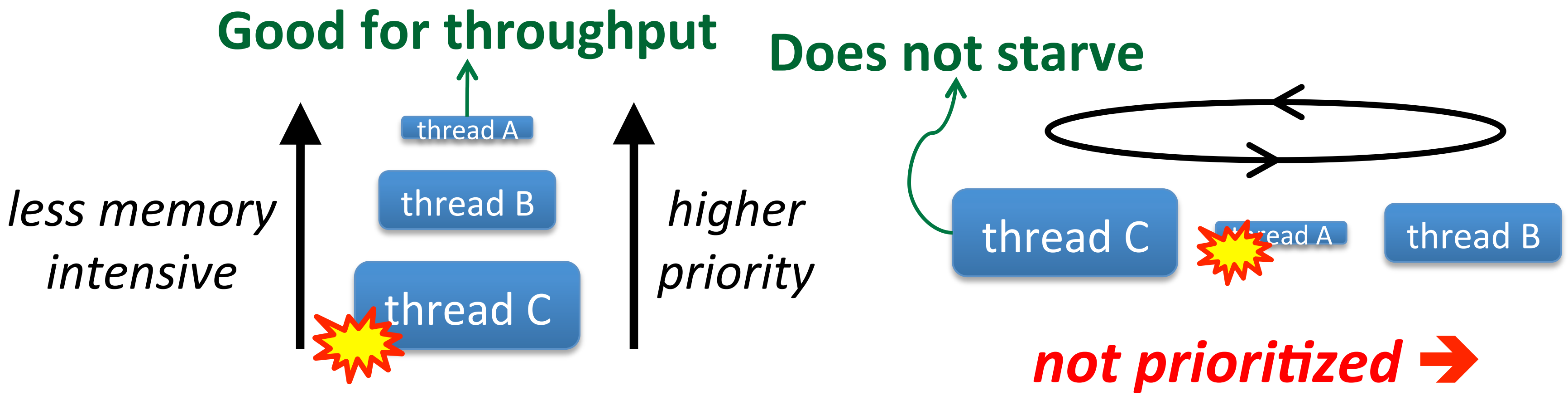
Throughput vs. Fairness

***Throughput biased** approach*

Prioritize less memory-intensive threads

***Fairness biased** approach*

Take turns accessing memory

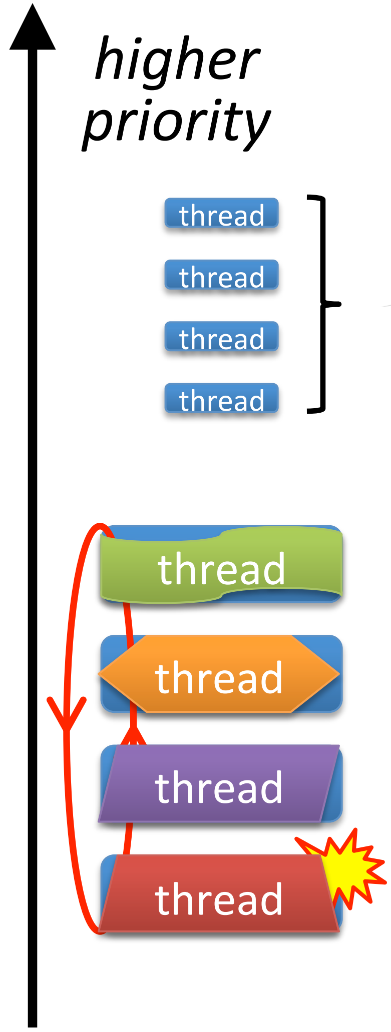


starvation → unfairness

not prioritized → reduced throughput

Single policy for all threads is insufficient

Achieving the Best of Both Worlds



For Throughput

Prioritize memory-non-intensive threads

For Fairness

Unfairness caused by memory-intensive being prioritized over each other

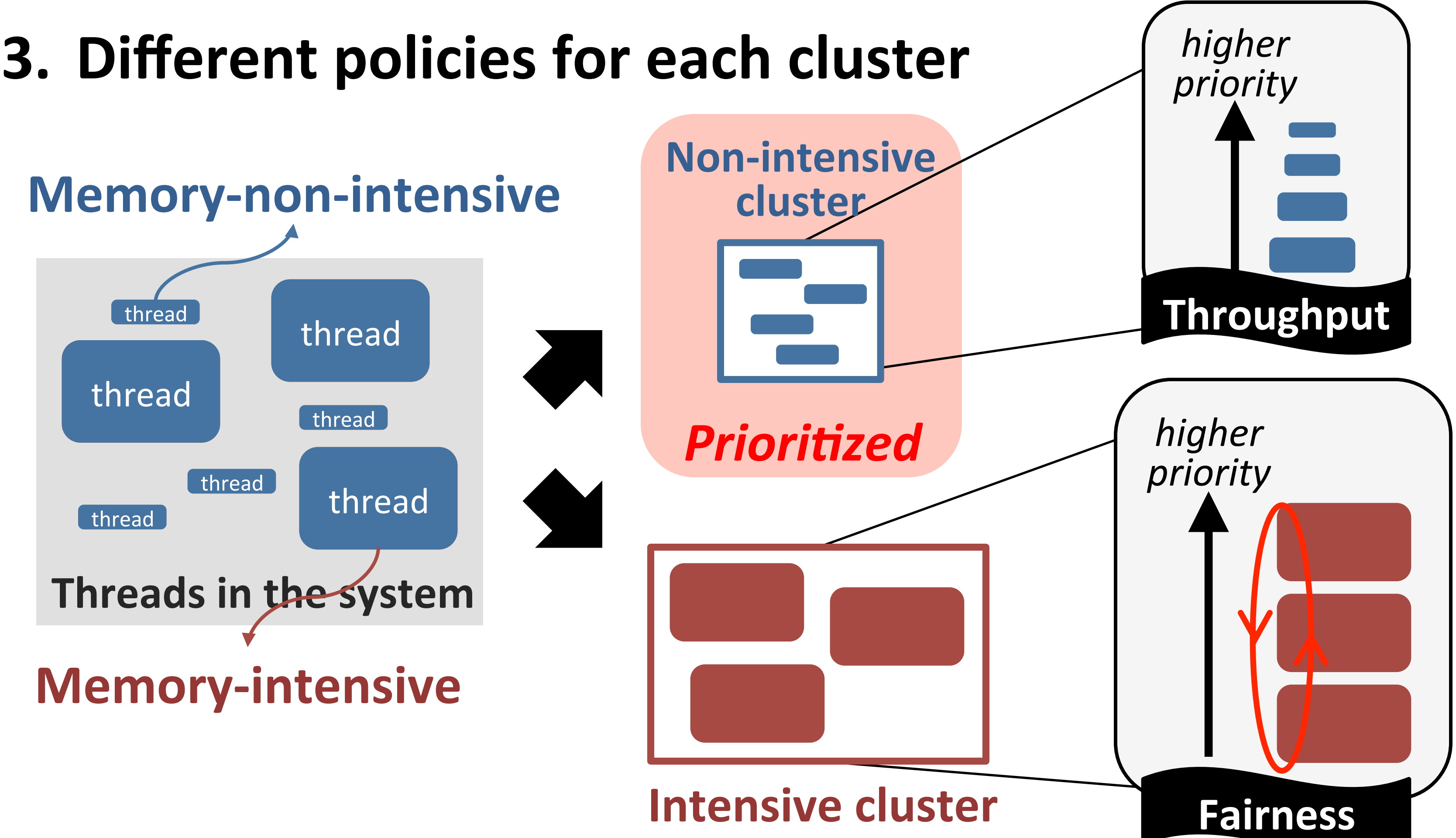
- Shuffle thread ranking

Memory-intensive threads have different vulnerability to interference

- Shuffle asymmetrically

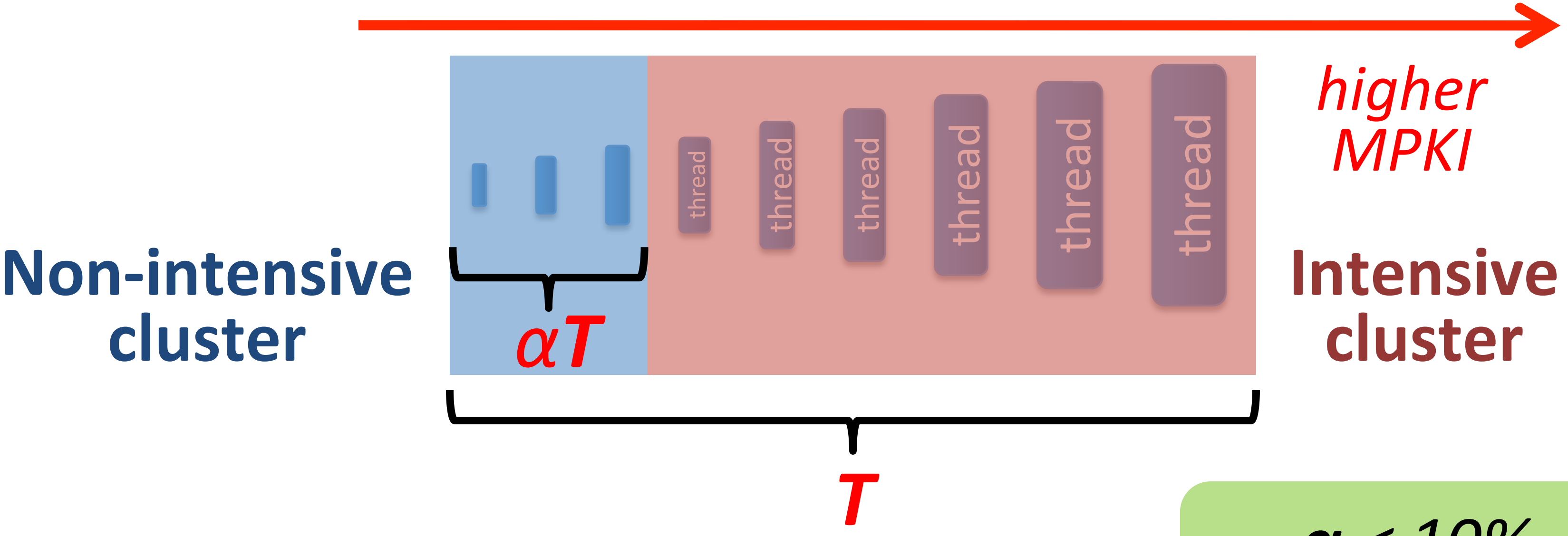
Thread Cluster Memory Scheduling [Kim+ MICRO'10]

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



Clustering Threads

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



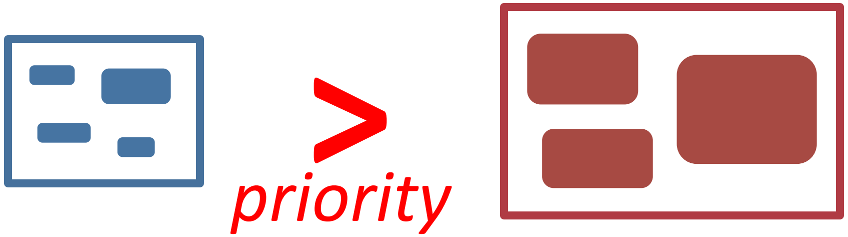
T **=** Total ***memory bandwidth usage***

***α** < 10%*
ClusterThreshold

Step2 Memory bandwidth usage *αT* divides clusters

Prioritization Between Clusters

Prioritize non-intensive cluster

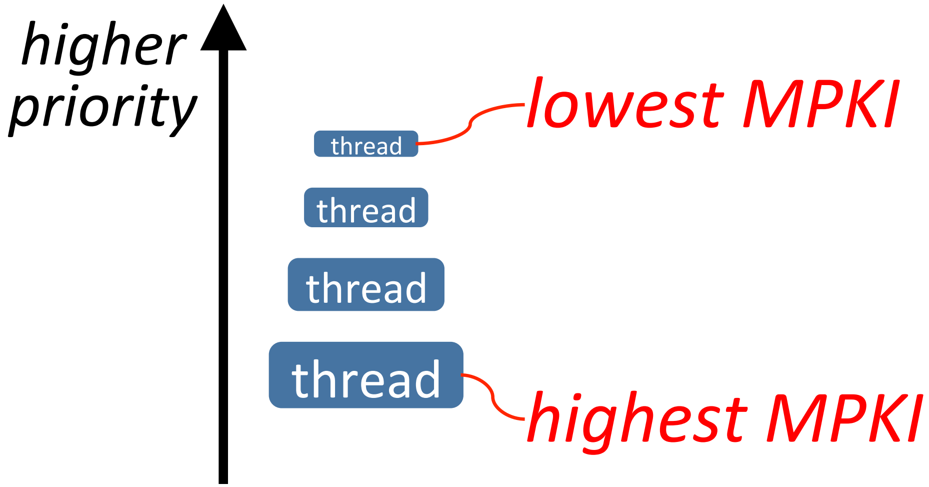


- **Increases system throughput**
  - Non-intensive threads have greater potential for making progress

- **Does not degrade fairness**
  - Non-intensive threads are "light"
  - Rarely interfere with intensive threads

Non-Intensive Cluster

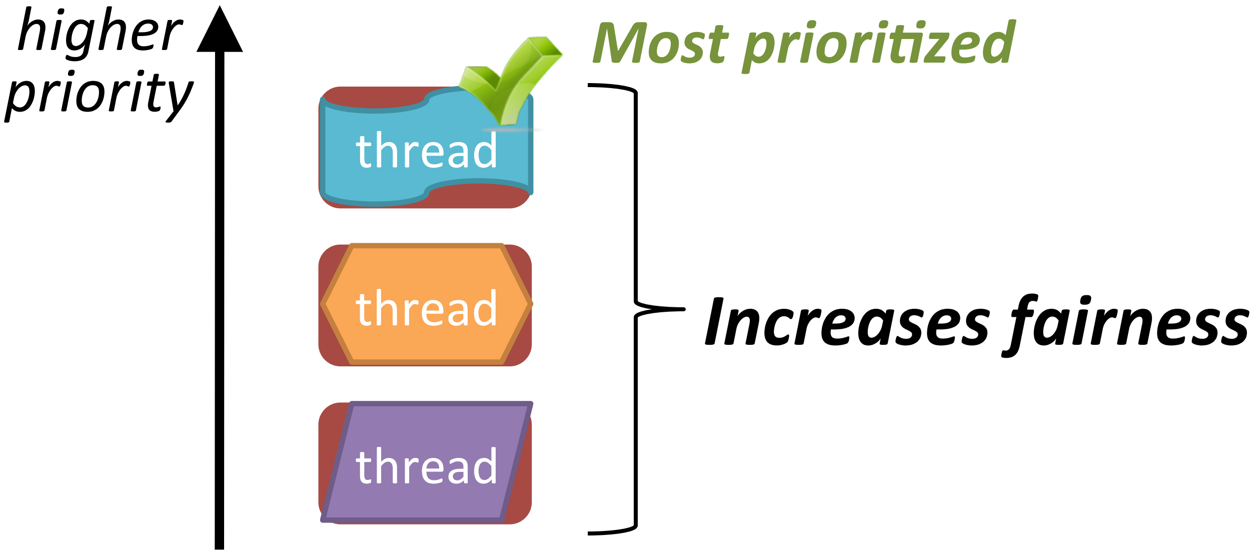
Prioritize threads according to MPKI



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

Intensive Cluster

Periodically shuffle the priority of threads



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

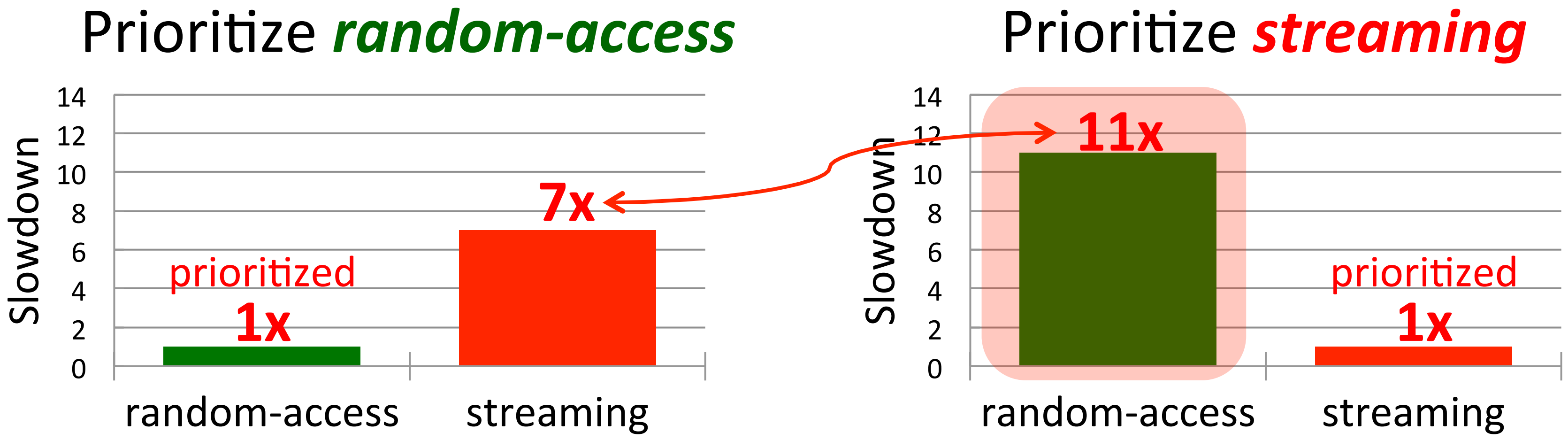
Case Study: A Tale of Two Threads

Case Study: Two intensive threads contending

1.random-access

2.streaming

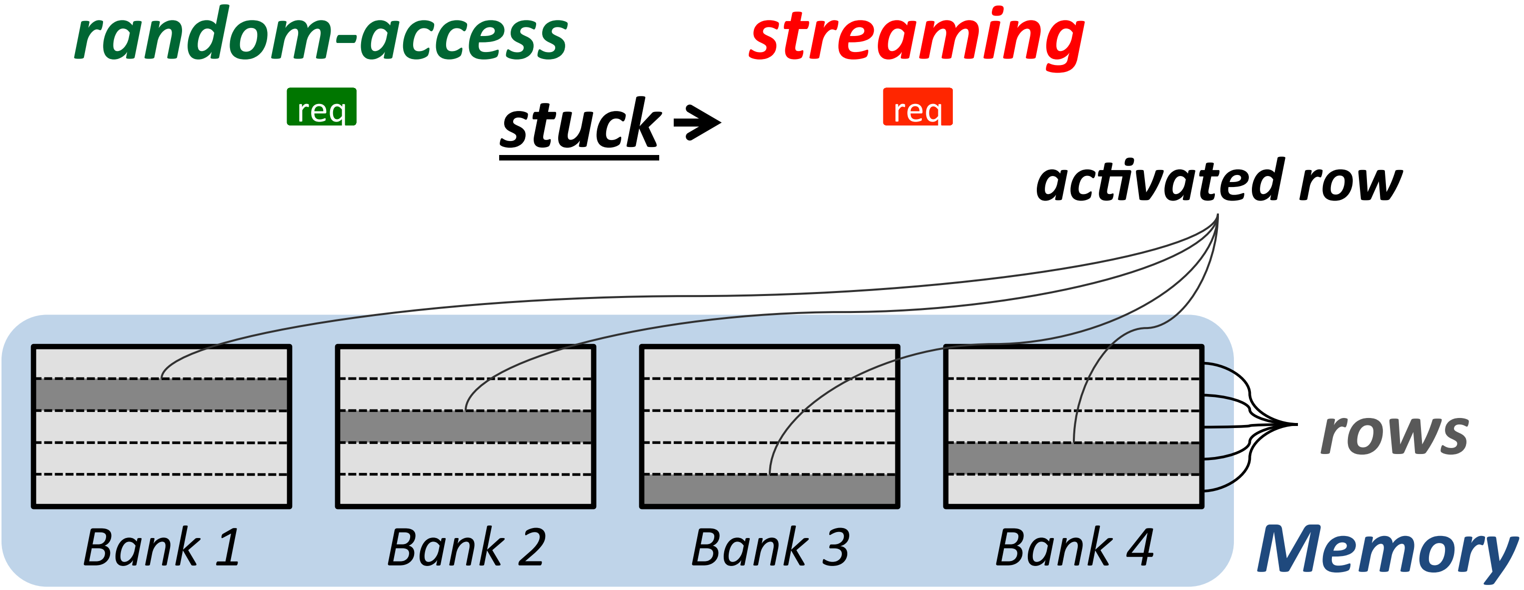
Which is slowed down more easily?



random-access thread is more easily slowed down

SAFARI

Why are Threads Different?



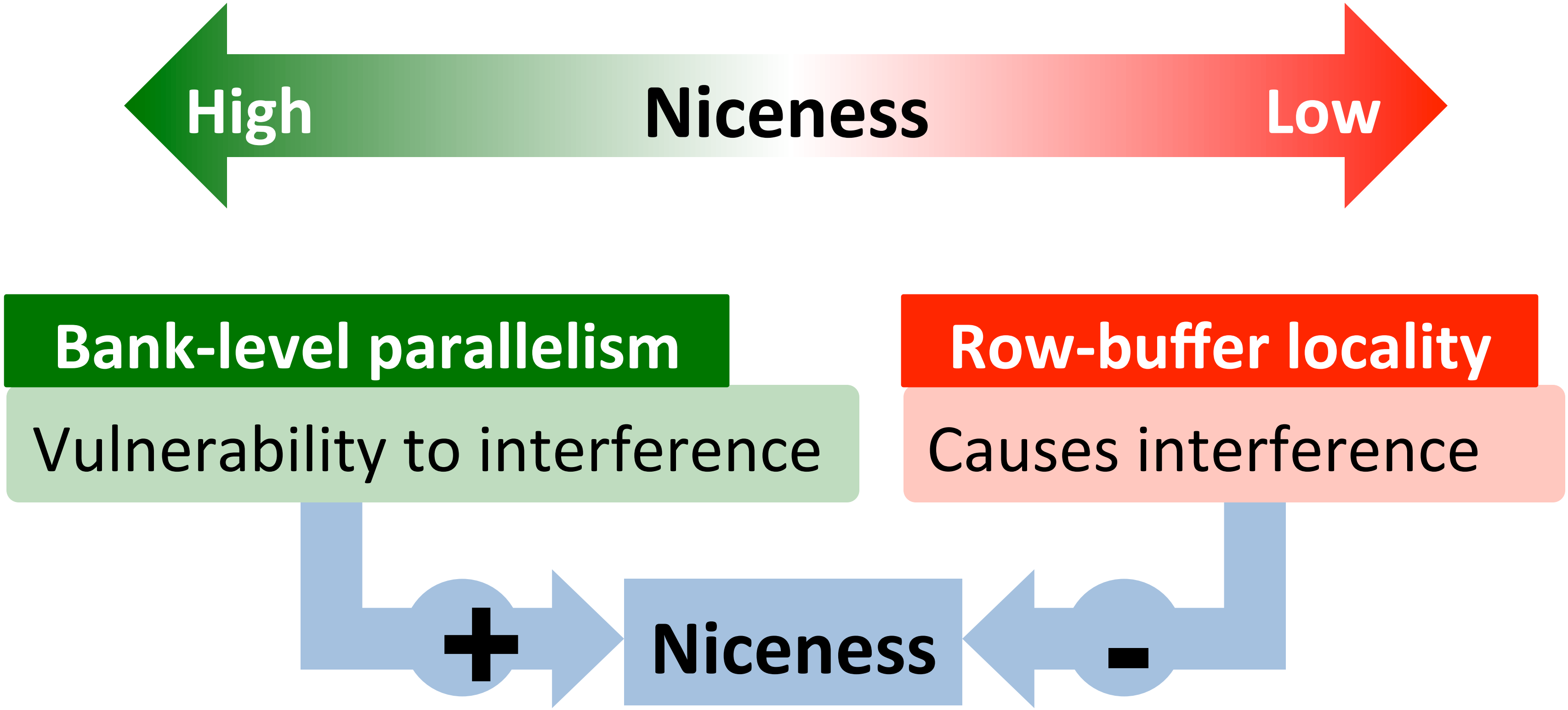
- All requests parallel
- High **bank-level parallelism**

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

Vulnerable to interference

Niceness

How to quantify difference between threads?



High

Niceness

Low

Bank-level parallelism

Vulnerability to interference

Row-buffer locality

Causes interference

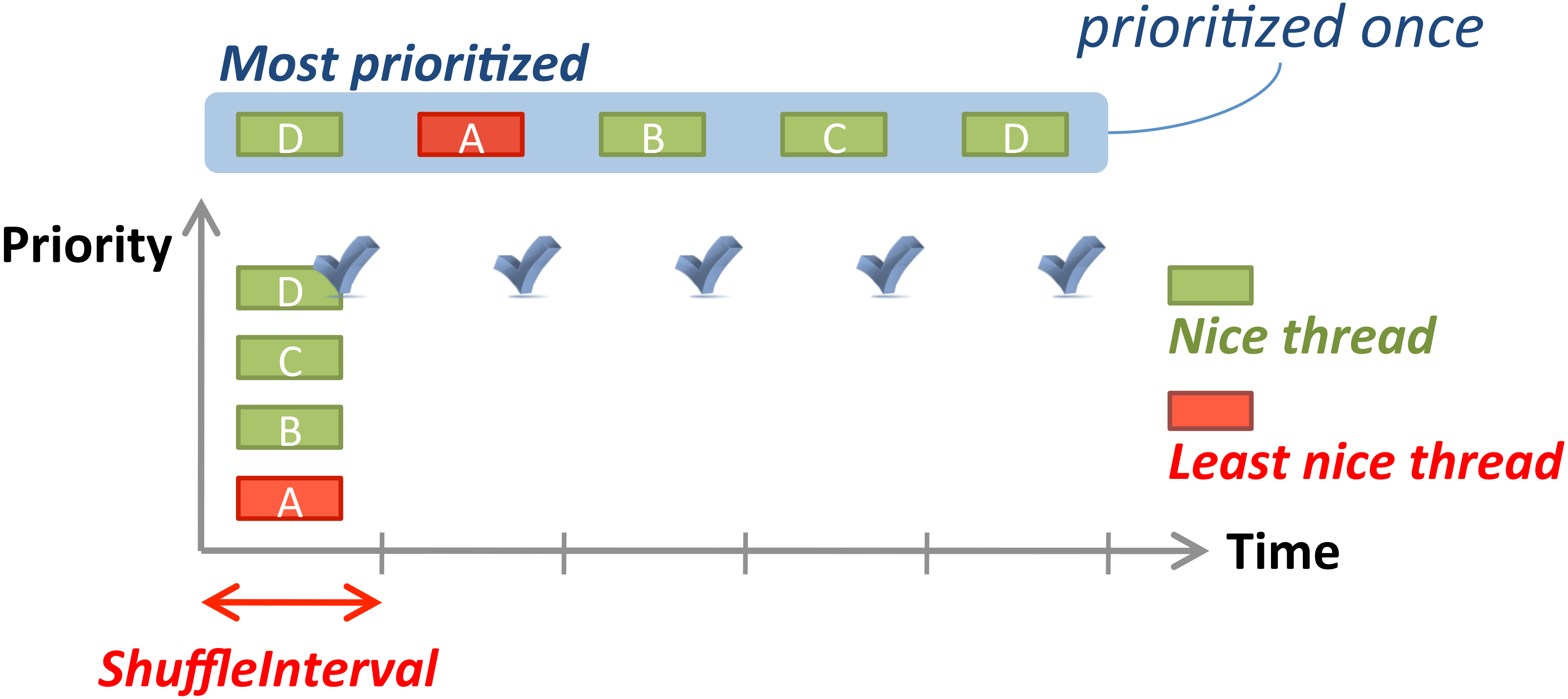
+

Niceness

-

Shuffling: Round-Robin vs. Niceness-Aware

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

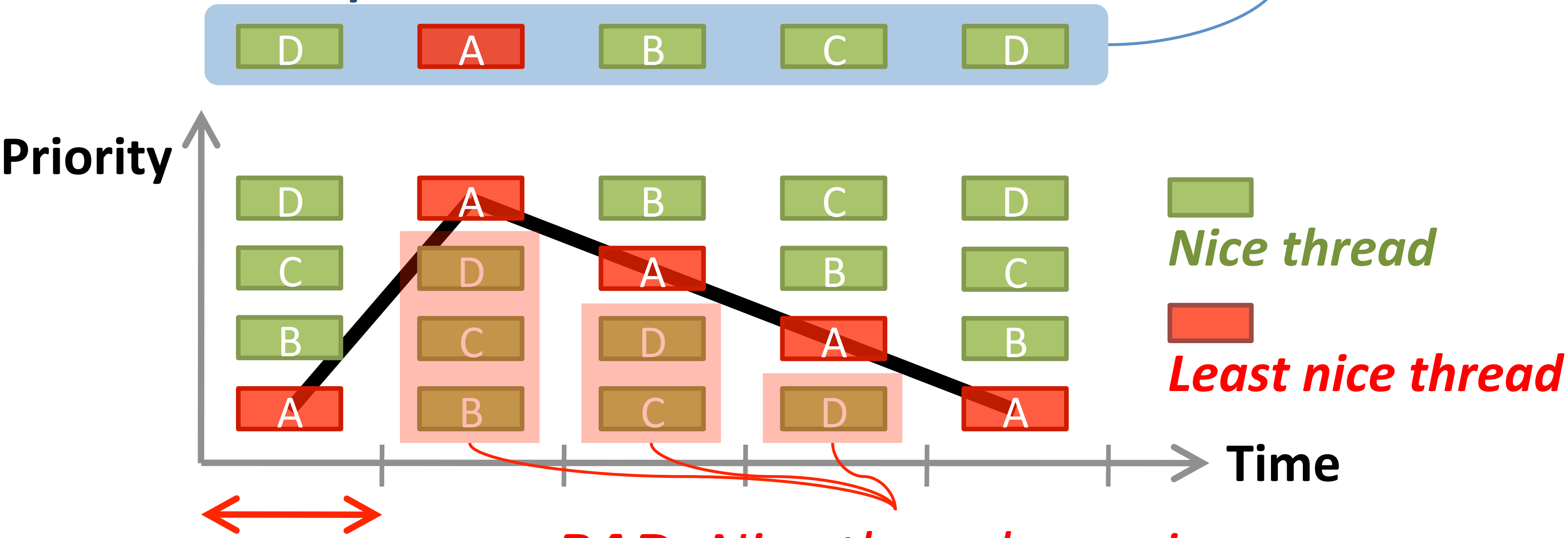


Shuffling: Round-Robin vs. Niceness-Aware

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

GOOD: *Each thread prioritized once*

Most prioritized



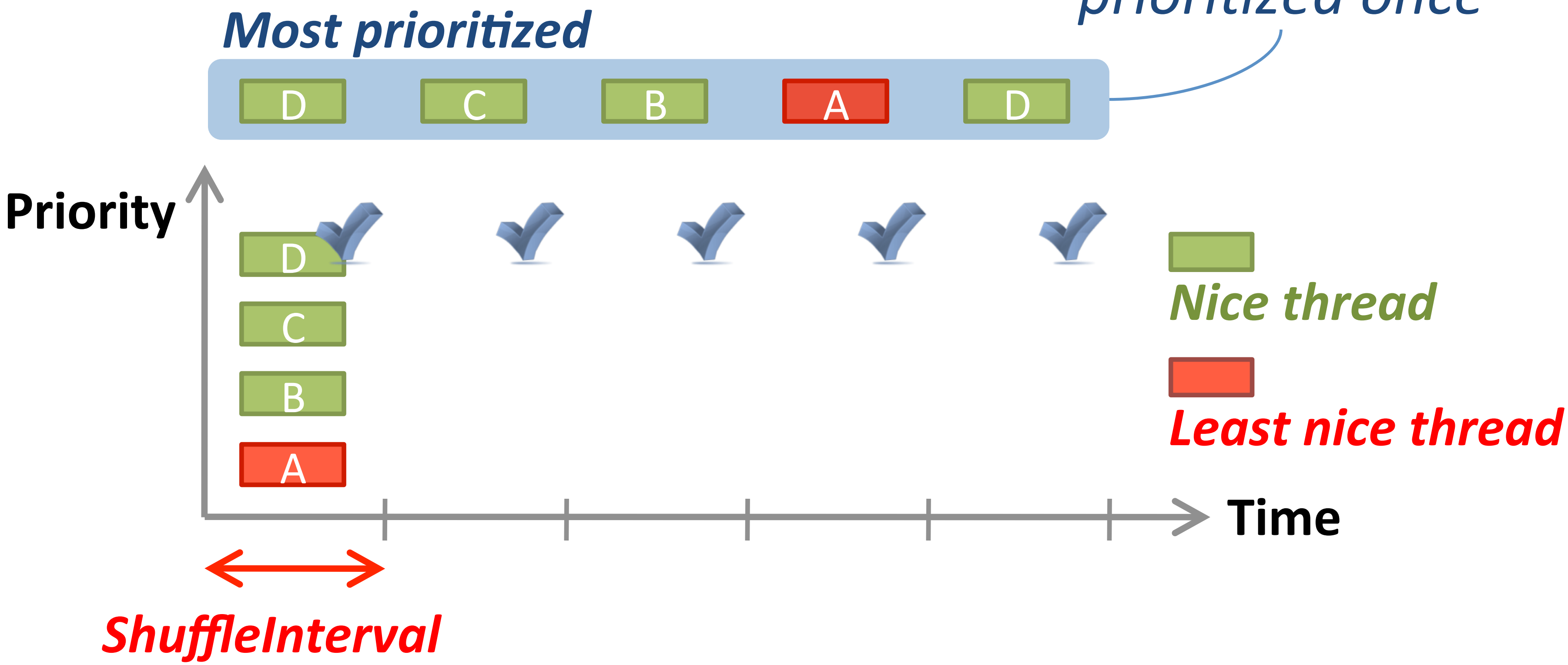
ShuffleInterval

BAD: *Nice threads receive lots of interference*

Shuffling: Round-Robin vs. Niceness-Aware

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

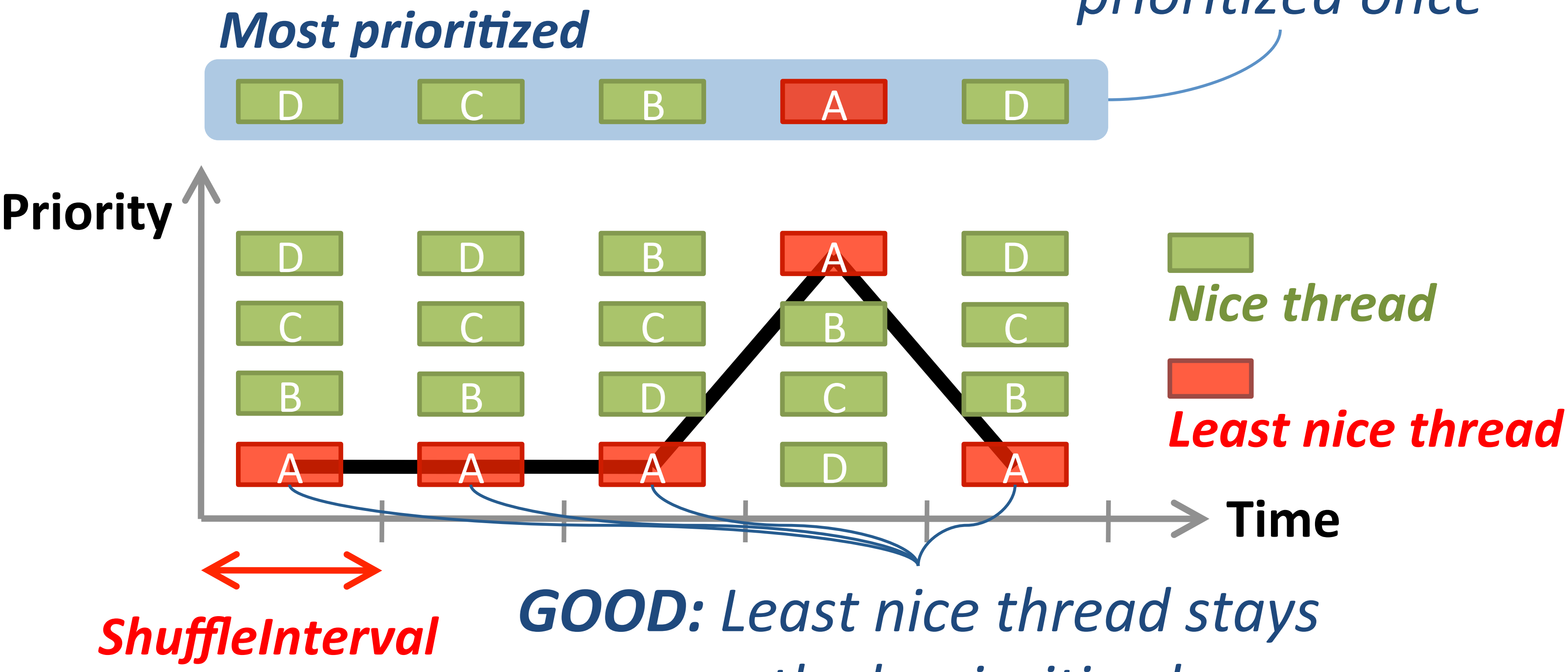
GOOD: *Each thread prioritized once*



Shuffling: Round-Robin vs. Niceness-Aware

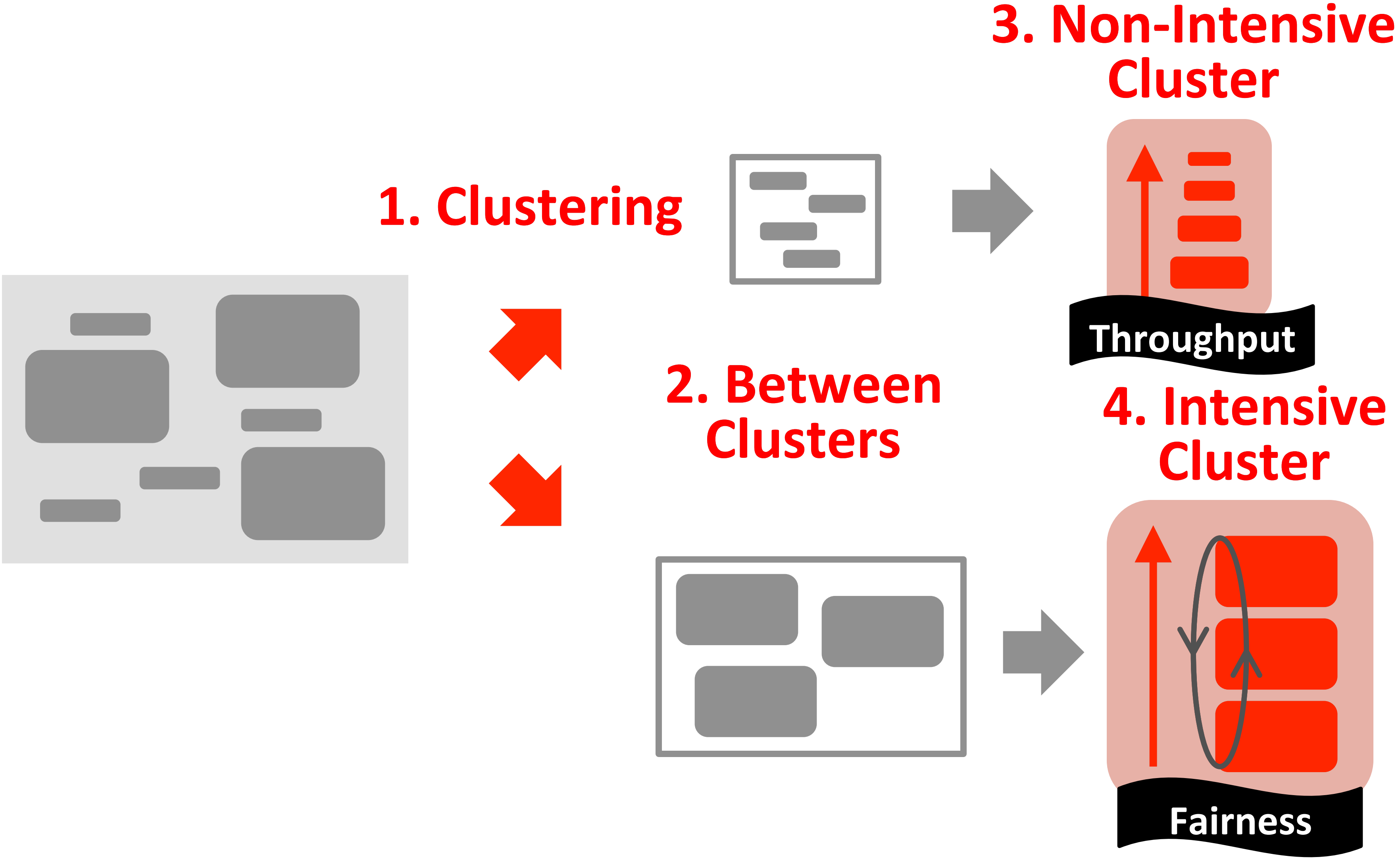
1. ***Round-Robin*** *shuffling*
2. ***Niceness-Aware*** *shuffling*

GOOD: *Each thread prioritized once*

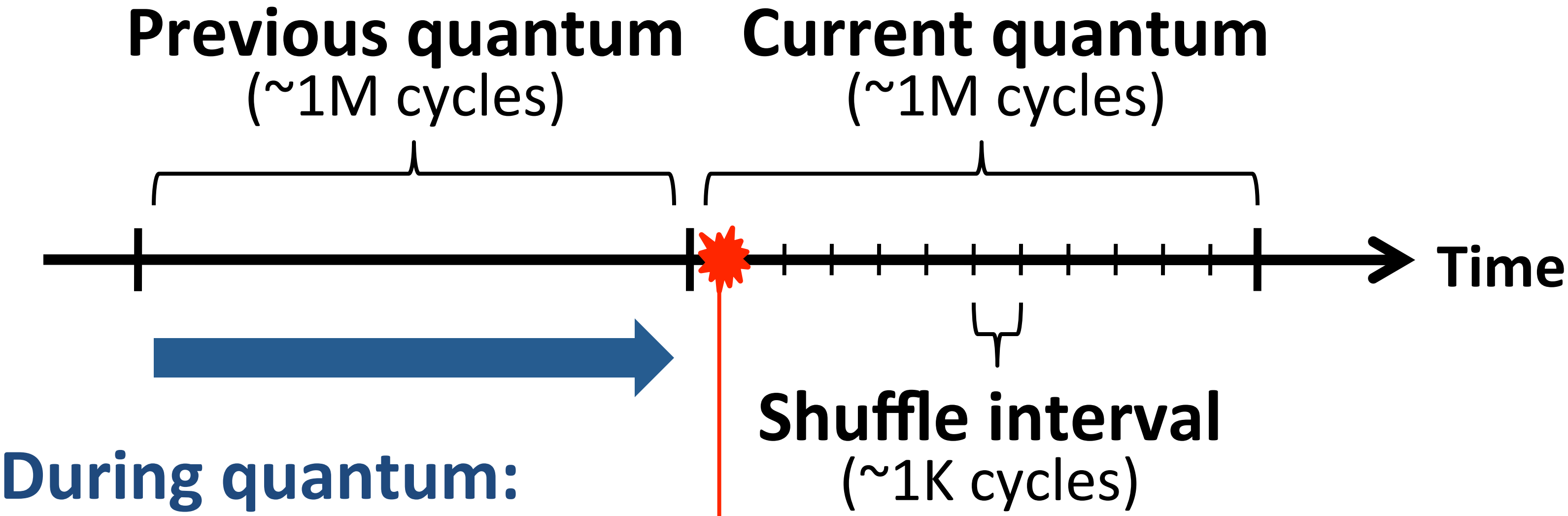


GOOD: *Least nice thread stays mostly deprioritized*

TCM Outline


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

TCM: Quantum-Based Operation



During quantum:

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

Beginning of quantum:

- Perform clustering
- Compute niceness of intensive threads

SAFARI

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

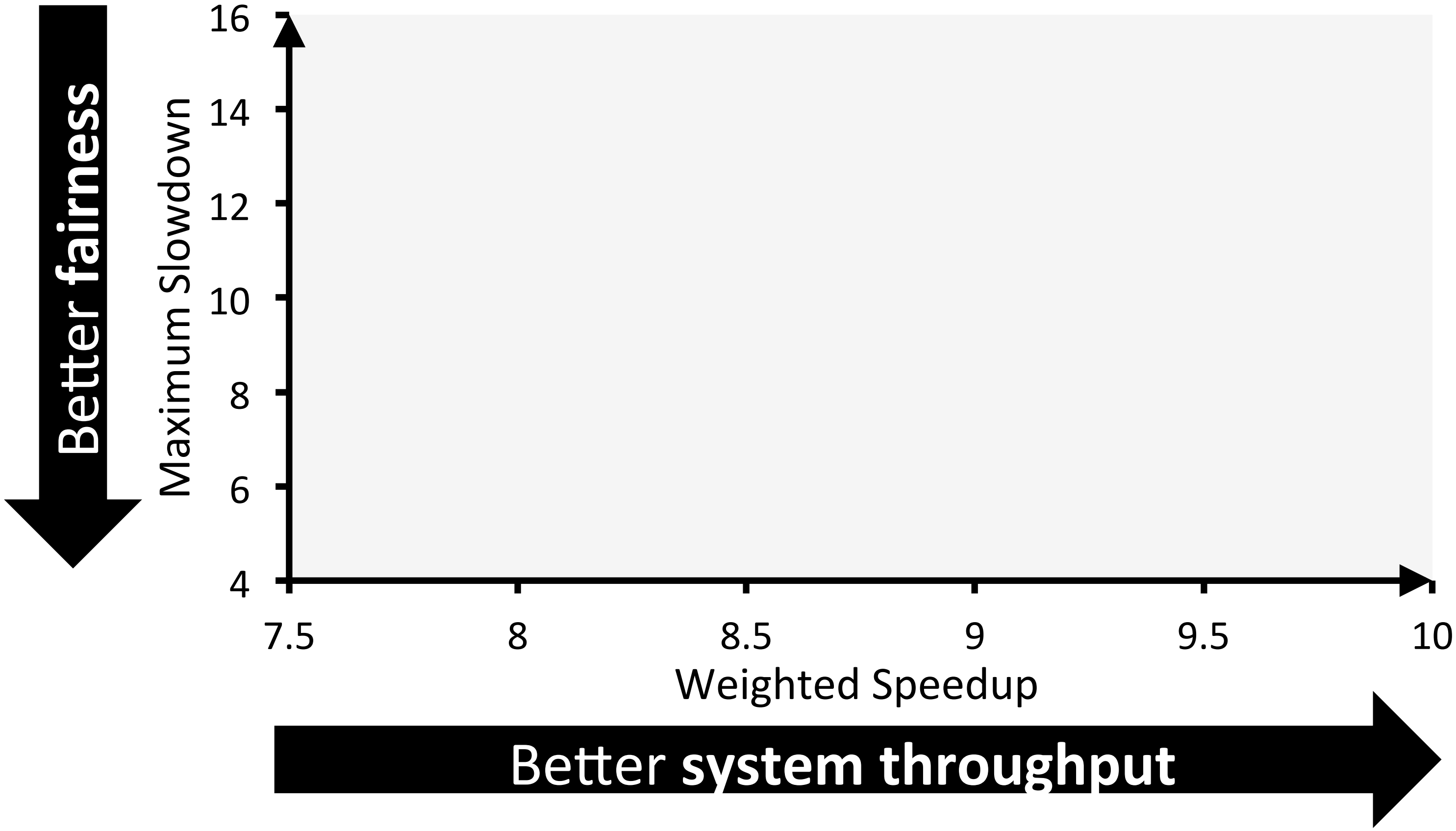
- Non-intensive threads not always prioritized → Low throughput

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

- Most intensive thread starves → Low fairness

TCM: Throughput and Fairness

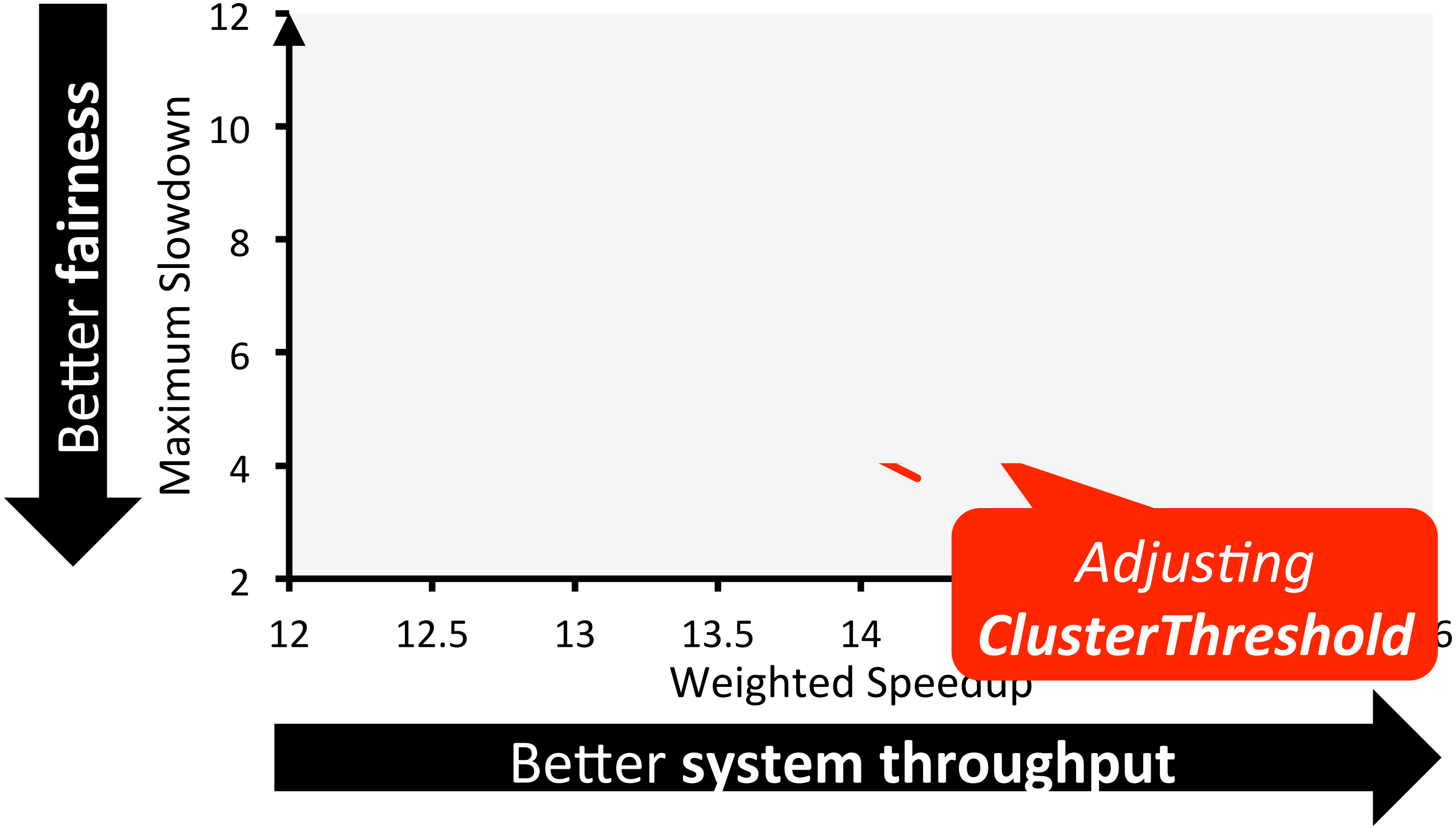
24 cores, 4 memory controllers, 96 workloads



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

TCM: Fairness-Throughput Tradeoff

When configuration parameter is varied…



Better fairness

Better **system throughput**

Adjusting ClusterThreshold

TCM allows robust fairness-throughput tradeoff

Operating System Support

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

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

Conclusion

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

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

- *TCM provides the best system throughput and fairness*

TCM Pros and Cons

- Upsides:
  - Provides both high fairness and high performance

- Downsides:
  - Scalability to large buffer sizes?
  - Effectiveness in a heterogeneous system?

Staged Memory Scheduling

Rachata Ausavarungnirun, Kevin Chang, Lavanya Subramanian, Gabriel Loh, and Onur Mutlu,
"Staged Memory Scheduling: Achieving High Performance and Scalability in Heterogeneous Systems"
39th International Symposium on Computer Architecture (***ISCA***),
Portland, OR, June 2012.

SMS ISCA 2012 Talk

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)

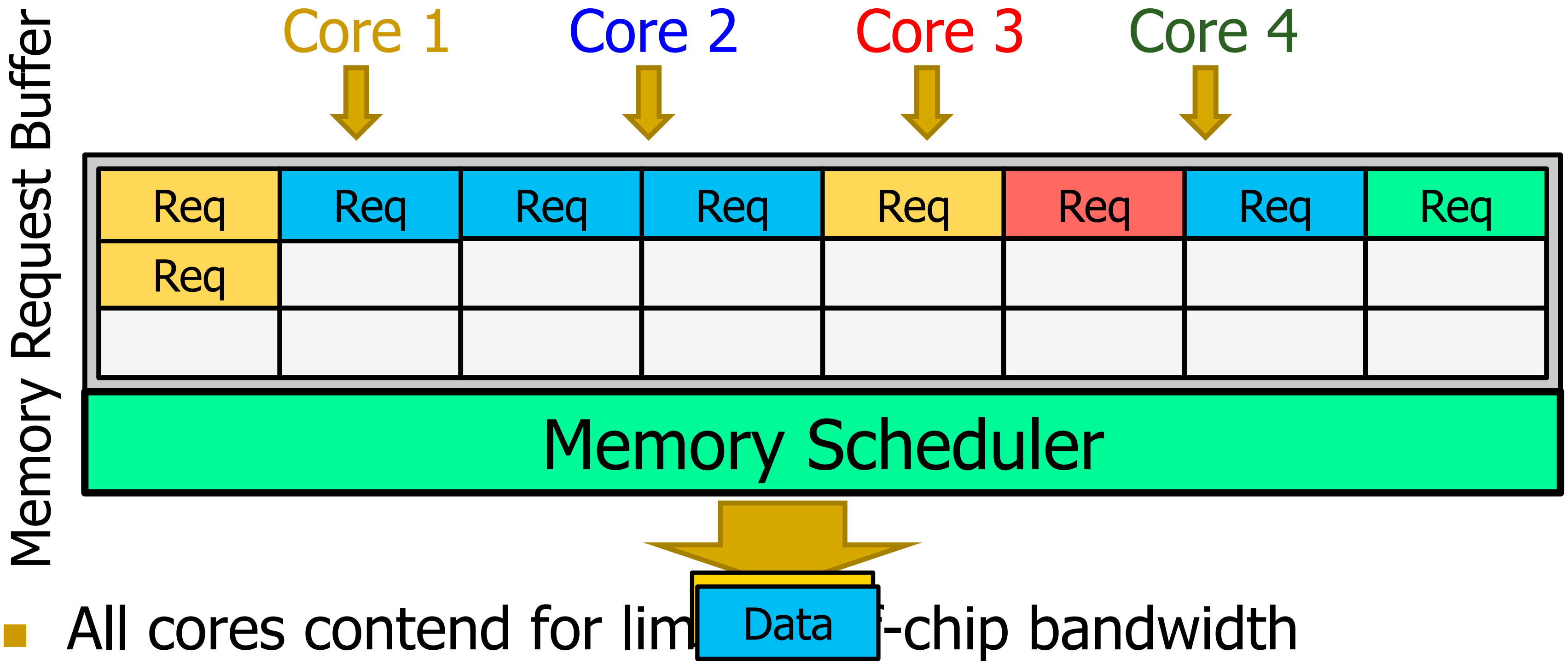
  decomposes the memory controller into three simple stages:

  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



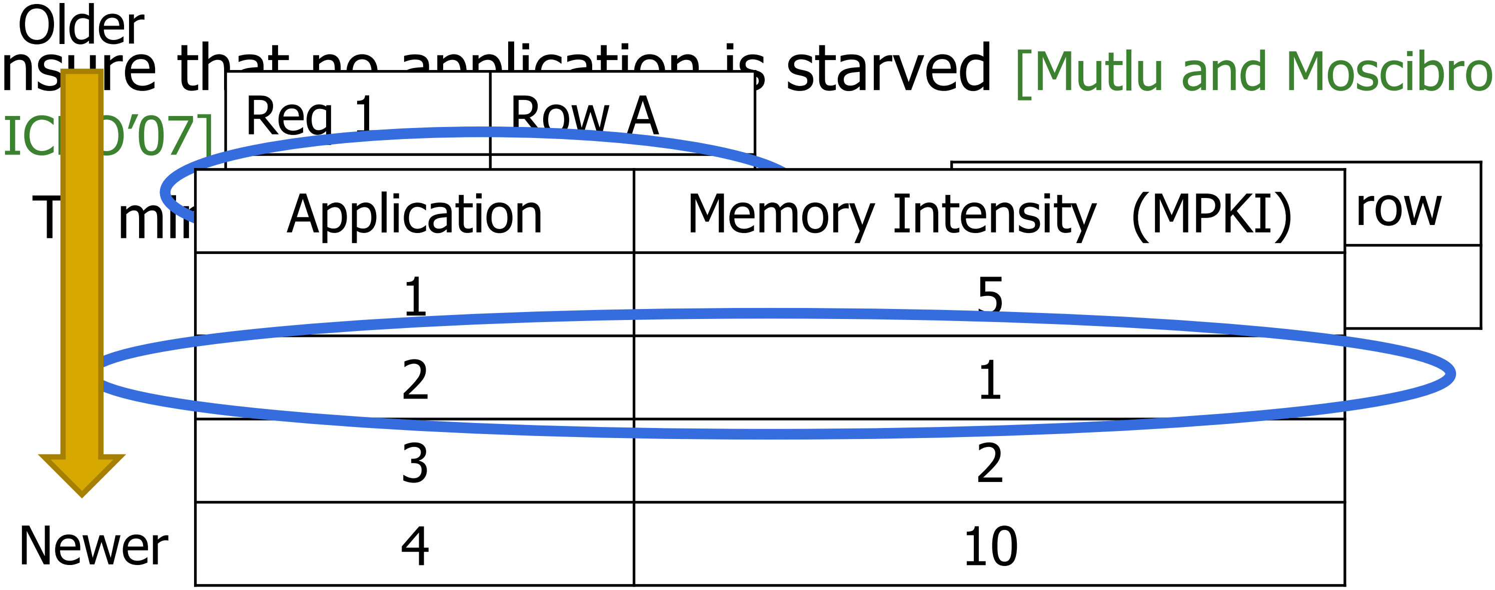
- All cores contend for limited off-chip bandwidth
  - Inter-application interference degrades system performance
  - The memory scheduler can help mitigate the problem
- How does the memory scheduler deliver good performance and fairness?

Three Principles of Memory Scheduling

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

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

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



Req 1	Row A

row

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

Memory Scheduling for CPU-GPU Systems

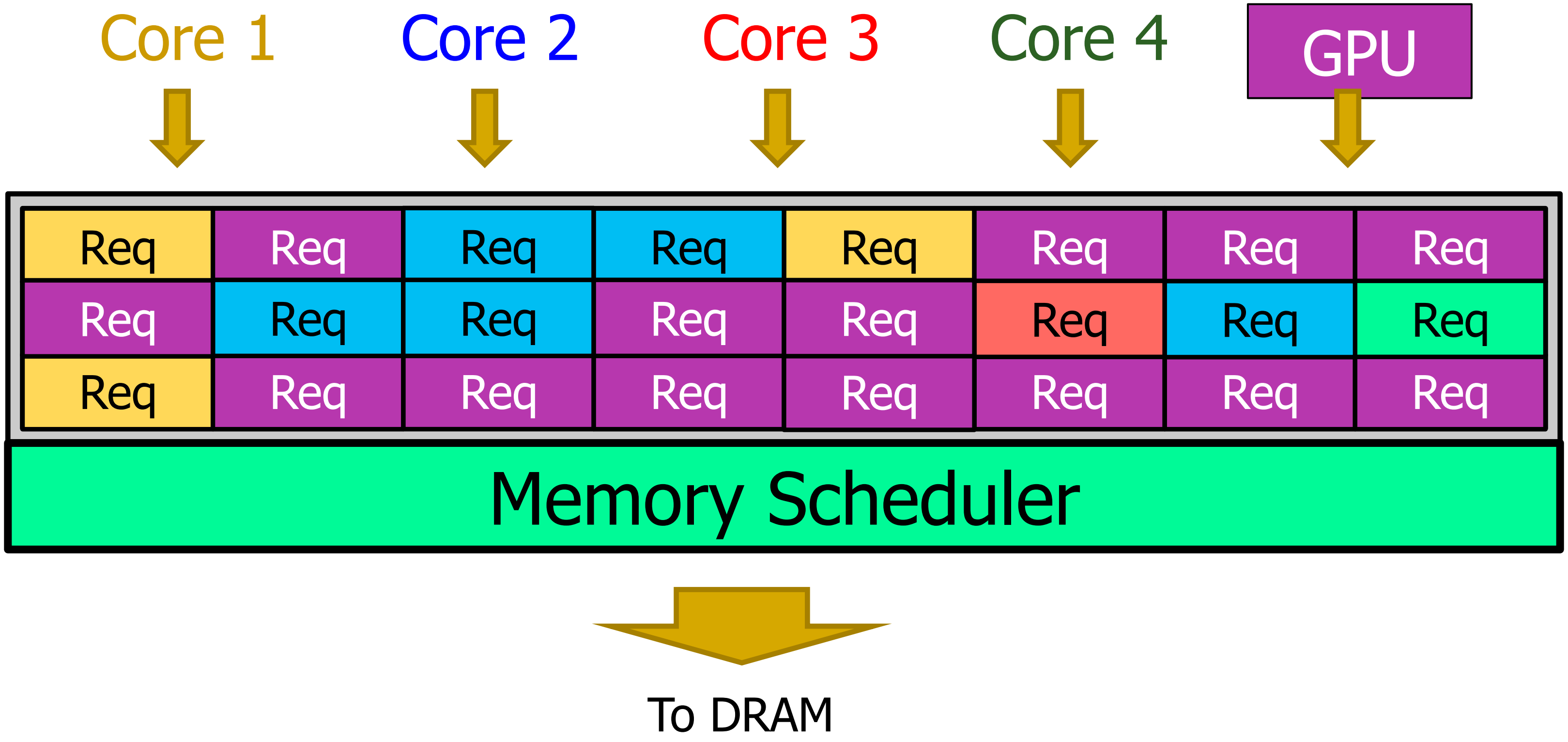
- Current and future systems integrate a GPU along with multiple cores

- GPU shares the main memory with the CPU cores

- GPU is much more (4x-20x) memory-intensive than CPU

- How should memory scheduling be done when GPU is integrated on-chip?

Introducing the GPU into the System



Core 1 Core 2 Core 3 Core 4 GPU

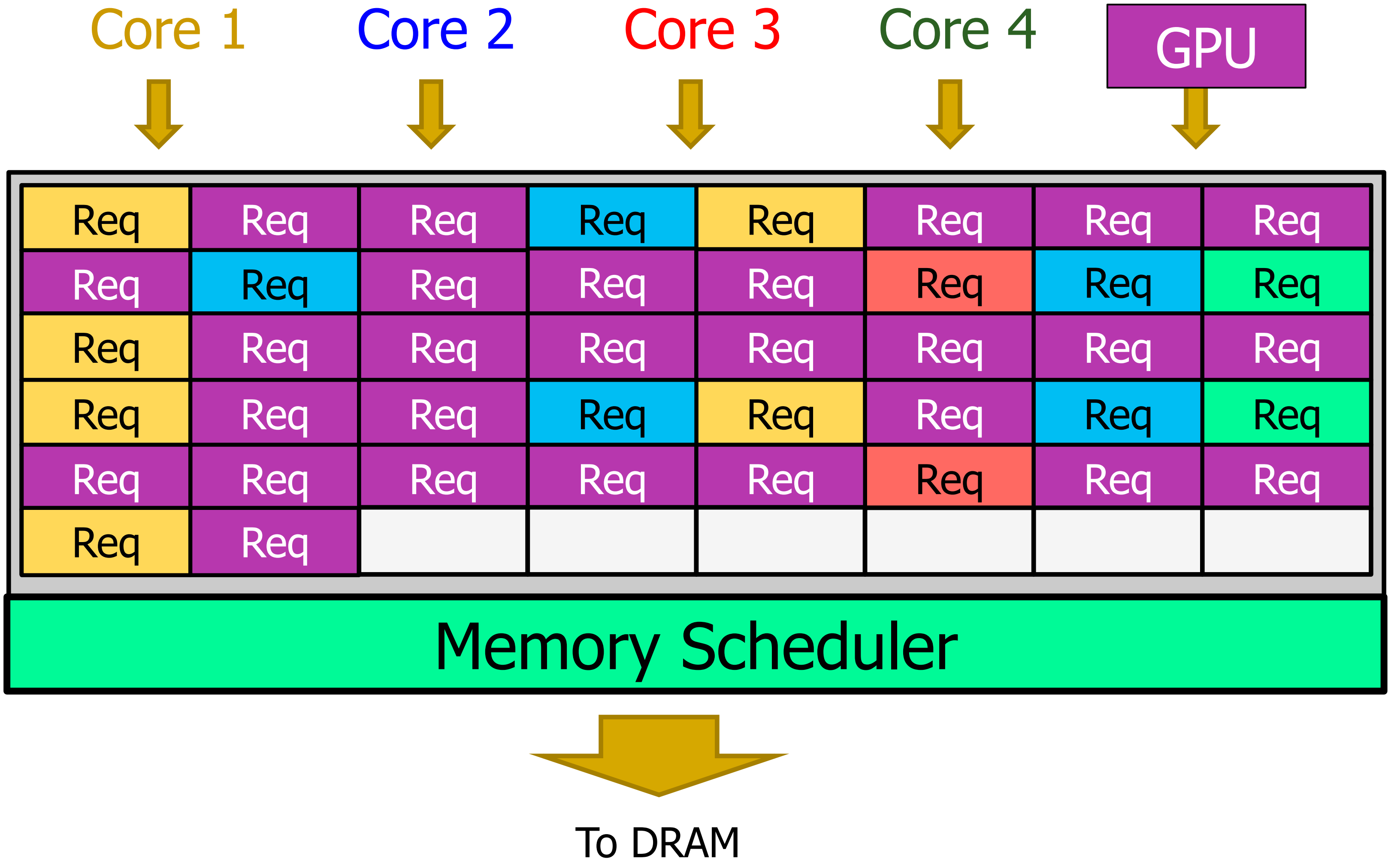
Req	Req	Req	Req	Req	Req	Req	Req
Req	Req	Req	Req	Req	Req	Req	Req
Req	Req	Req	Req	Req	Req	Req	Req

Memory Scheduler

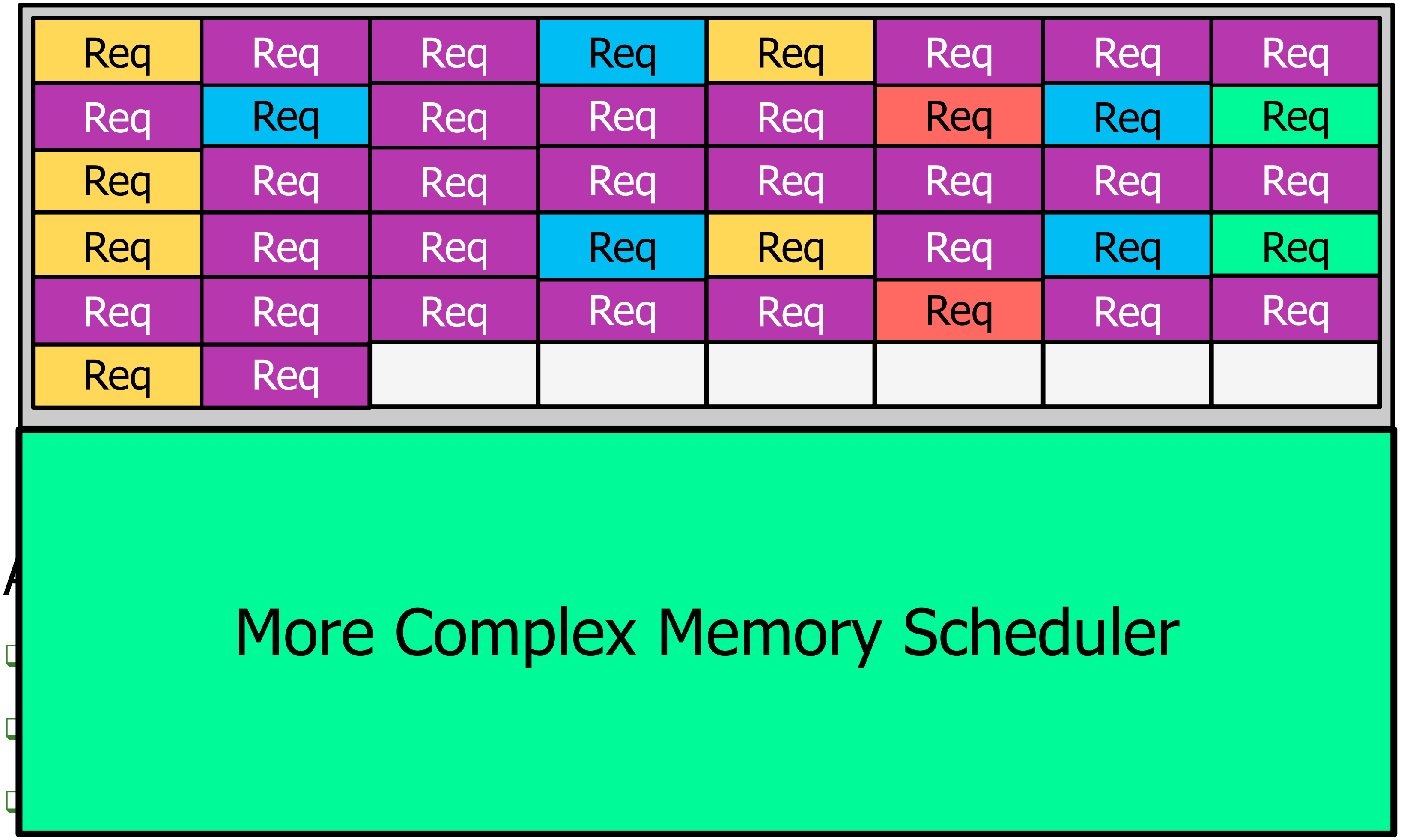
To DRAM

- 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


Core 1
Core 2
Core 3
Core 4
GPU
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
Memory Scheduler
To DRAM

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

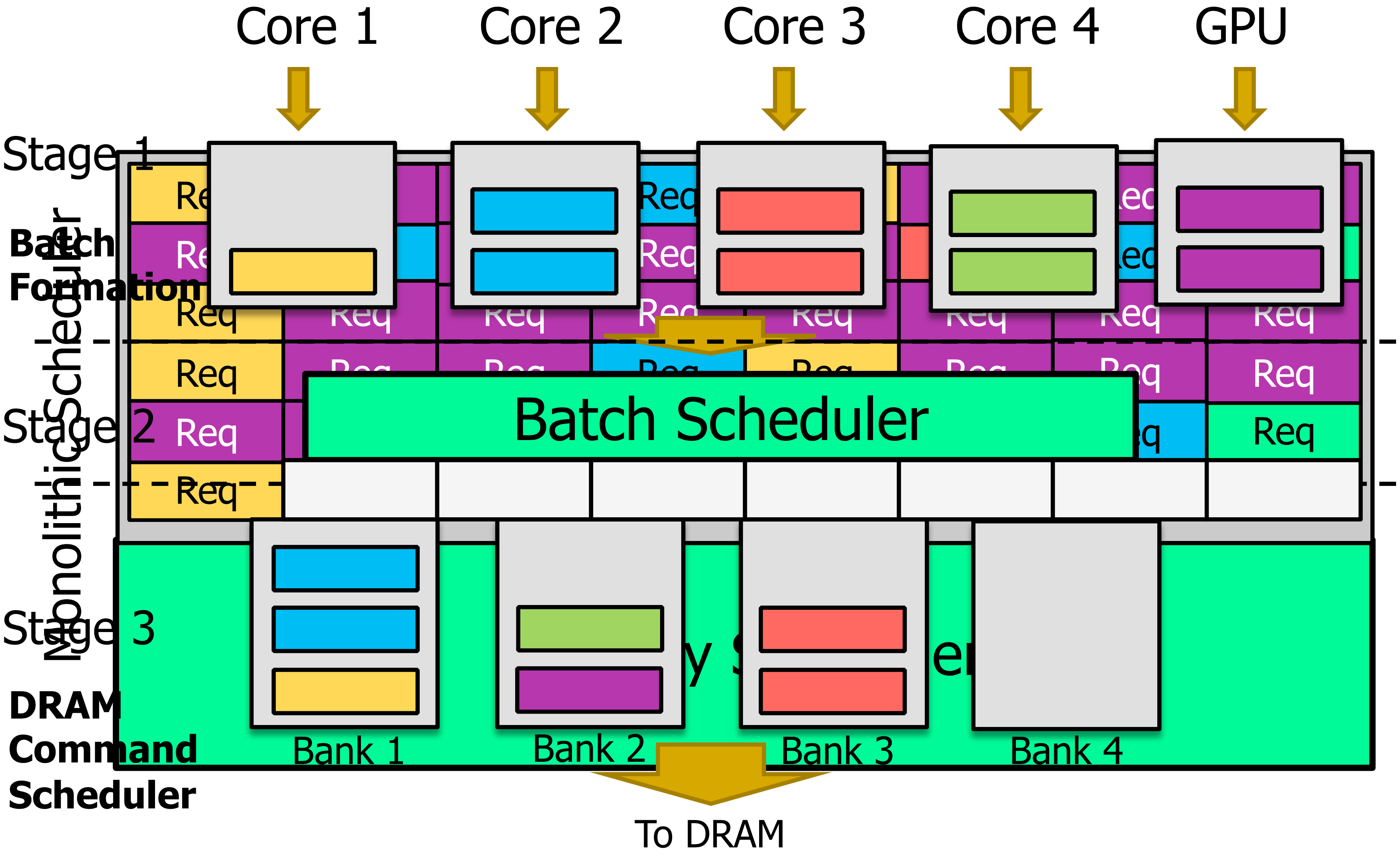
- Idea: Decouple the functional tasks of the memory controller
  - Partition tasks across several simpler HW structures (stages)

1) Maximize row buffer hits
  - Stage 1: Batch formation
  - Within each application, groups requests to the same row into batches

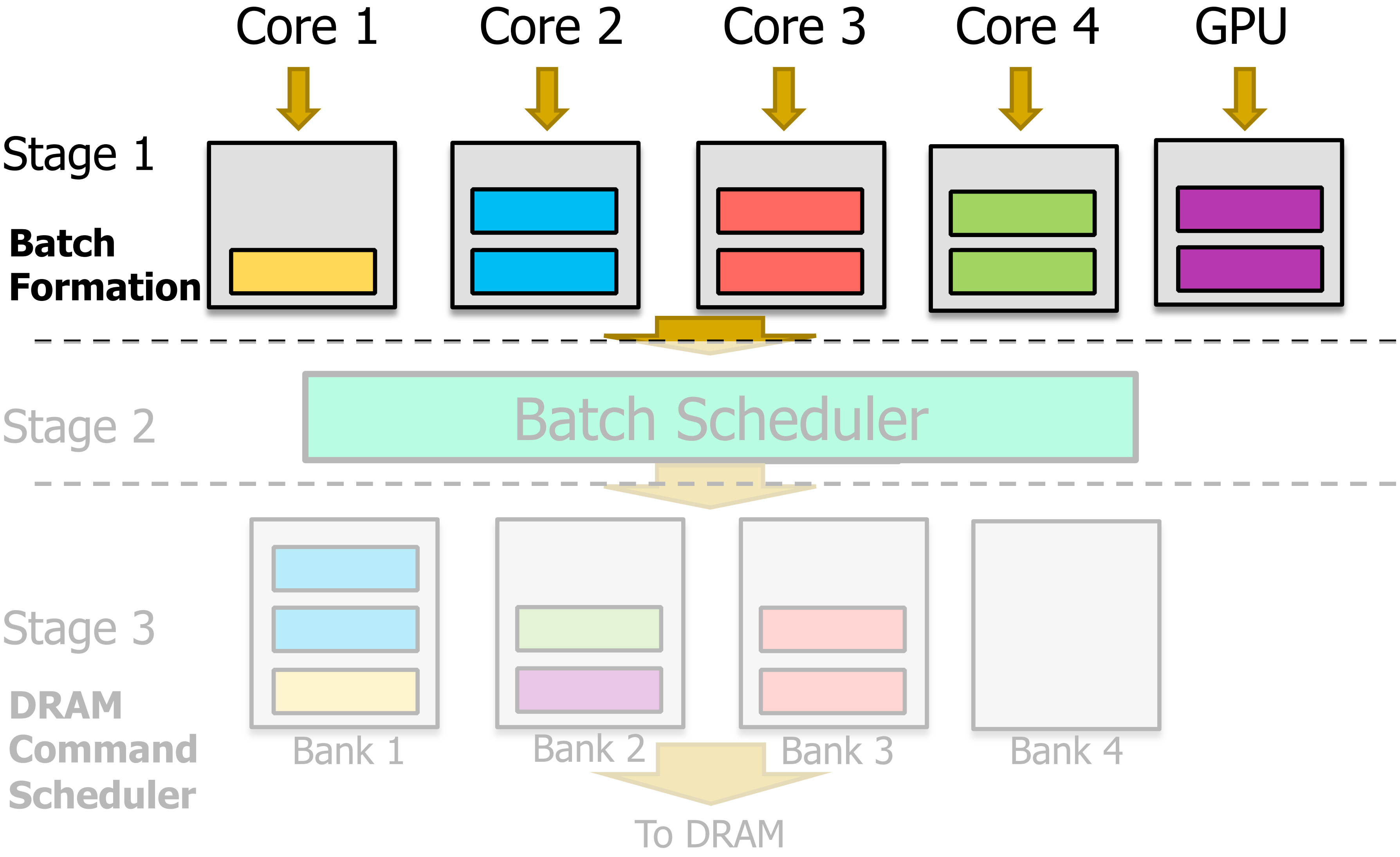
2) Manage contention between applications
  - Stage 2: Batch scheduler
  - Schedules batches from different applications

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

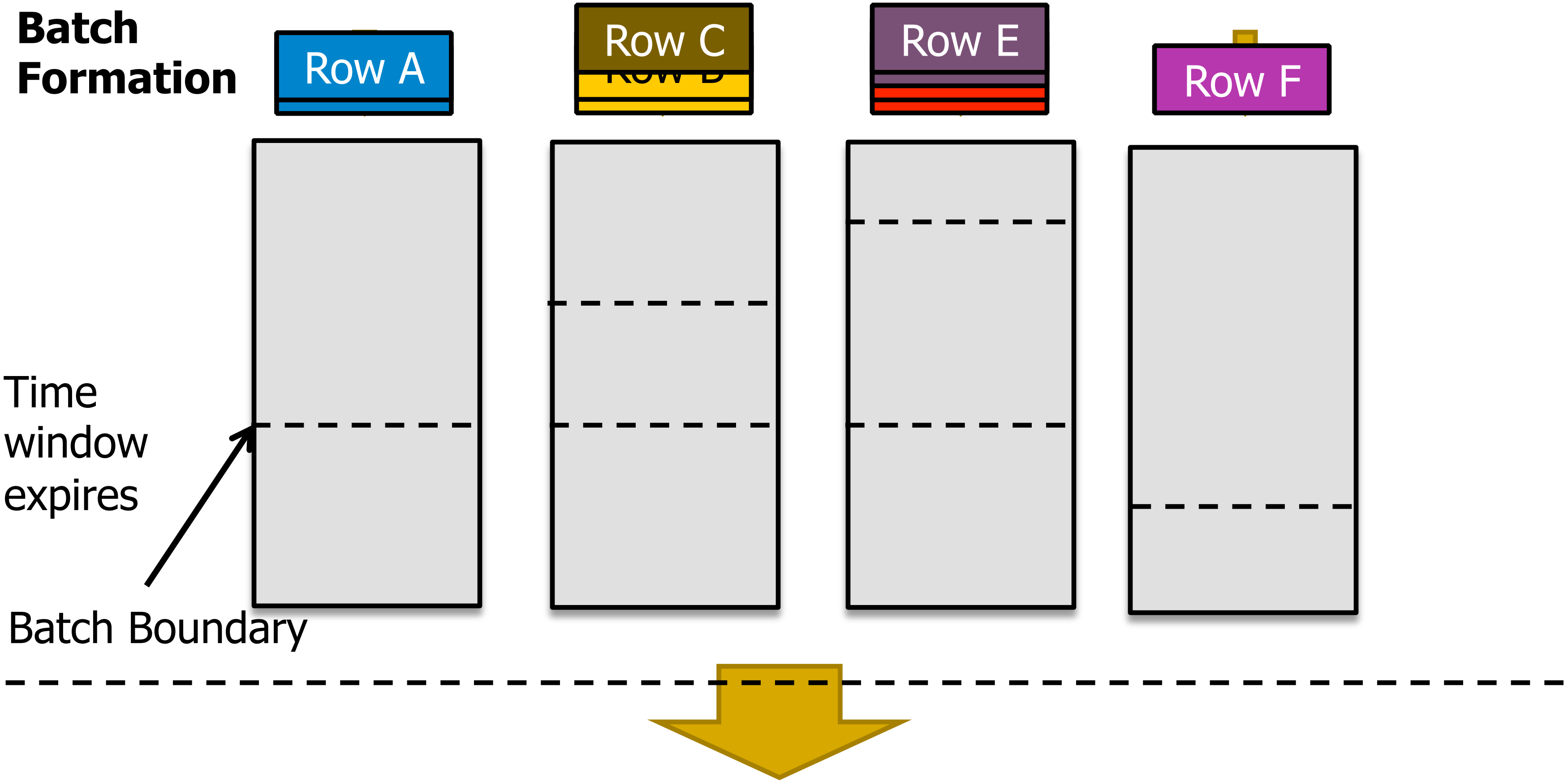
Stage 1: Batch Formation Example

Stage 1

Next request goes to a different row

Core 1 Core 2 Core 3 Core 4

Batch Formation



Row A

Row C

Row E

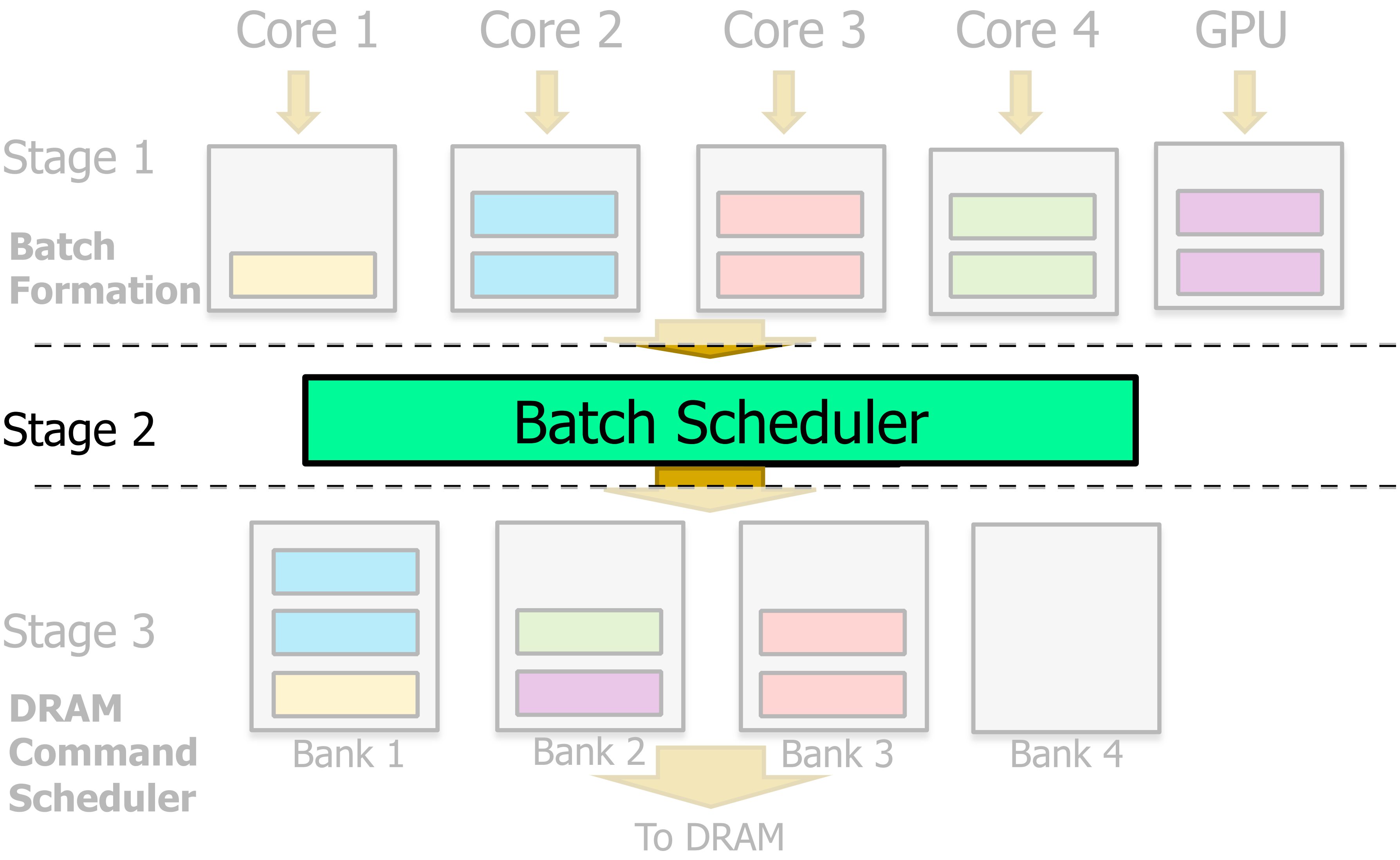
Row F

Time window expires

Batch Boundary

To Stage 2 (Batch Scheduling)

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

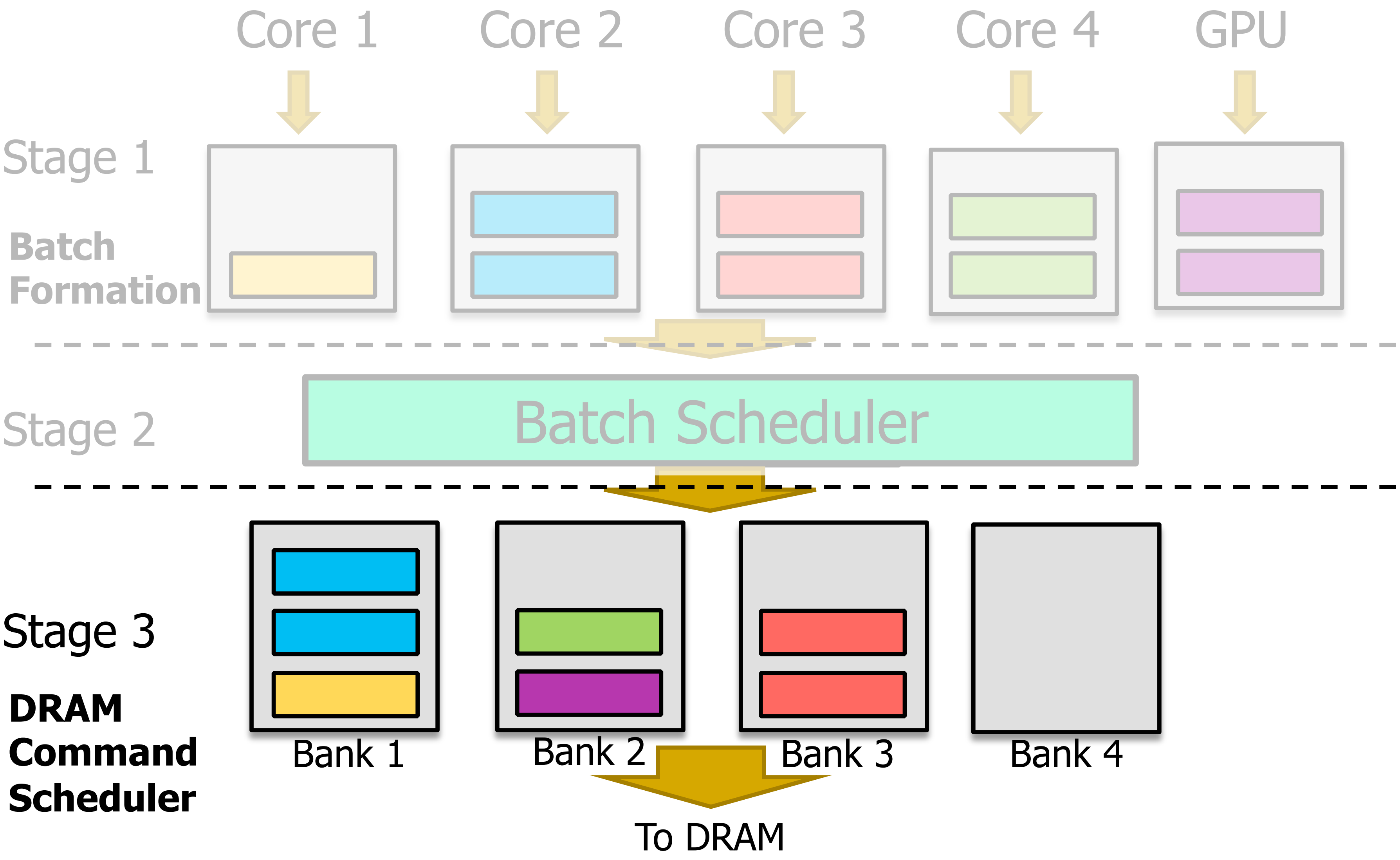
Stage 2: Batch Scheduling Policy

- 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



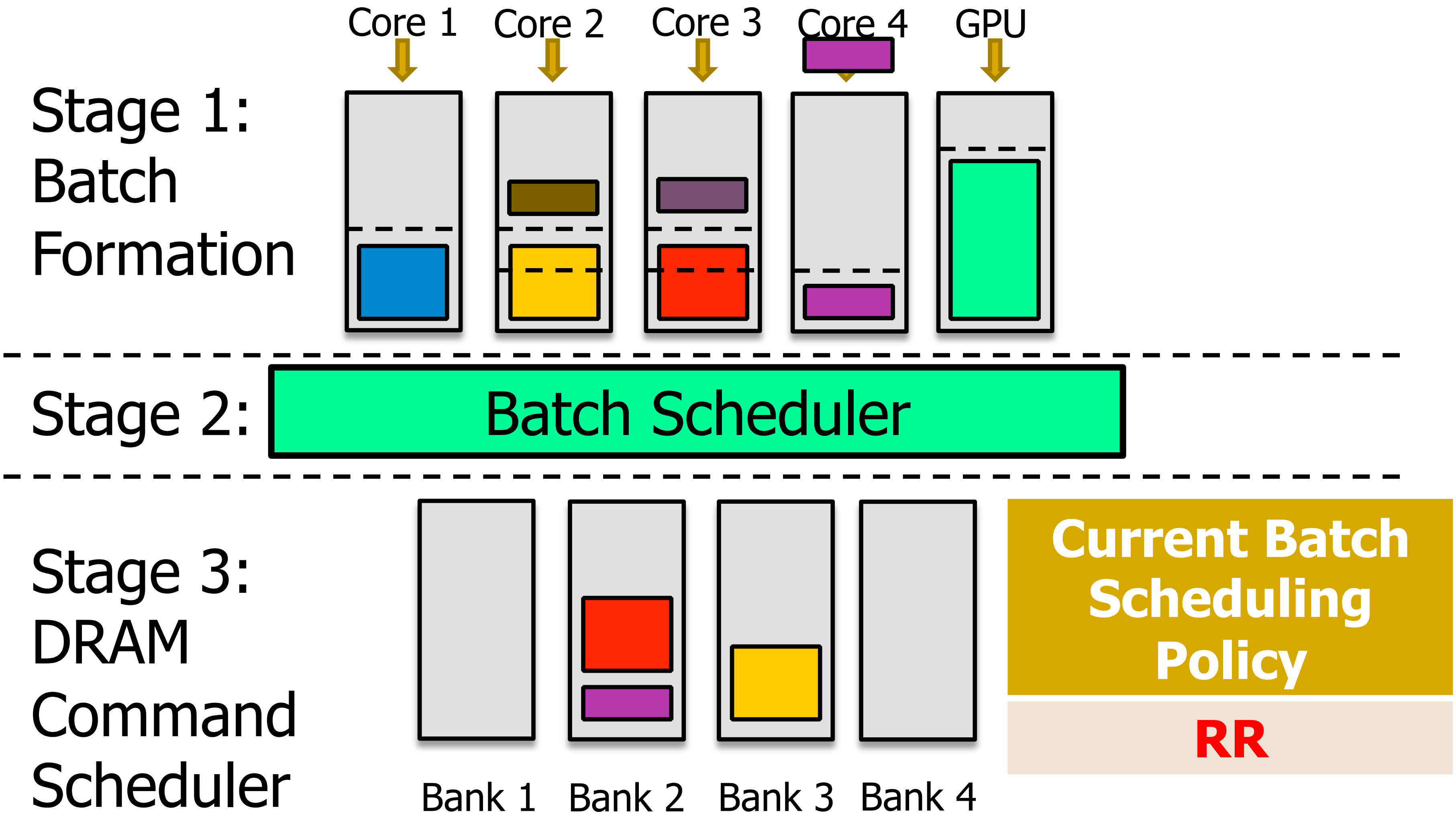
Stage 3: DRAM Command Scheduler

- 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



Stage 1: Batch Formation

Core 1 Core 2 Core 3 Core 4 GPU

Stage 2: Batch Scheduler

Stage 3: DRAM Command Scheduler

Bank 1 Bank 2 Bank 3 Bank 4

Current Batch Scheduling Policy

RR

Complexity

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

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

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

Methodology

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

- Workloads
  - CPU: SPEC CPU 2006
  - GPU: Recent games and GPU benchmarks
  - 7 workload categories based on the memory-intensity of CPU applications

    → Low memory-intensity (L)

    → Medium memory-intensity (M)

    → High memory-intensity (H)

Comparison to Previous Scheduling Algorithms

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

Evaluation Metrics

- CPU performance metric: Weighted speedup

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

- GPU performance metric: Frame rate speedup

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

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

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

Evaluated System Scenario: CPU Focused

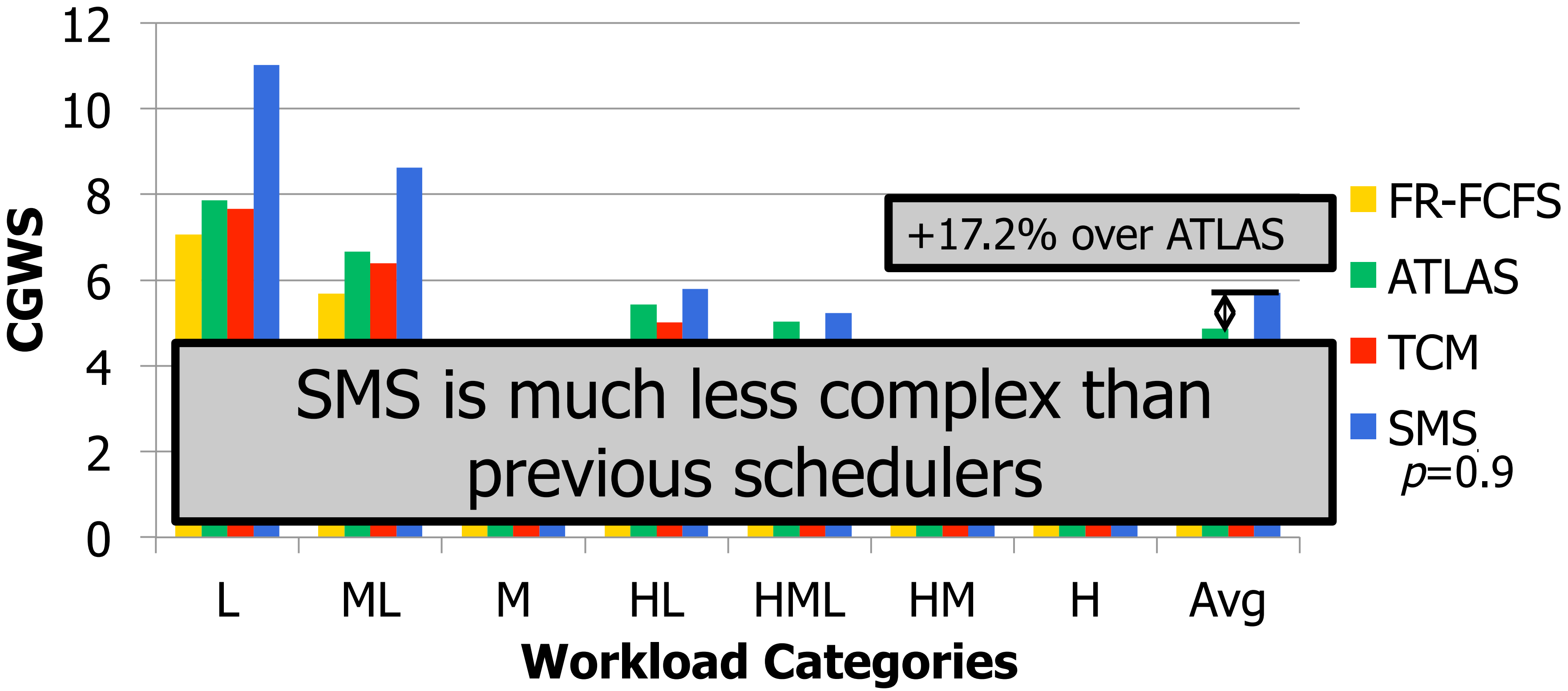
- GPU has low weight (weight = 1)

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

1

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

Performance: CPU-Focused System



+17.2% over ATLAS

SMS is much less complex than previous schedulers

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

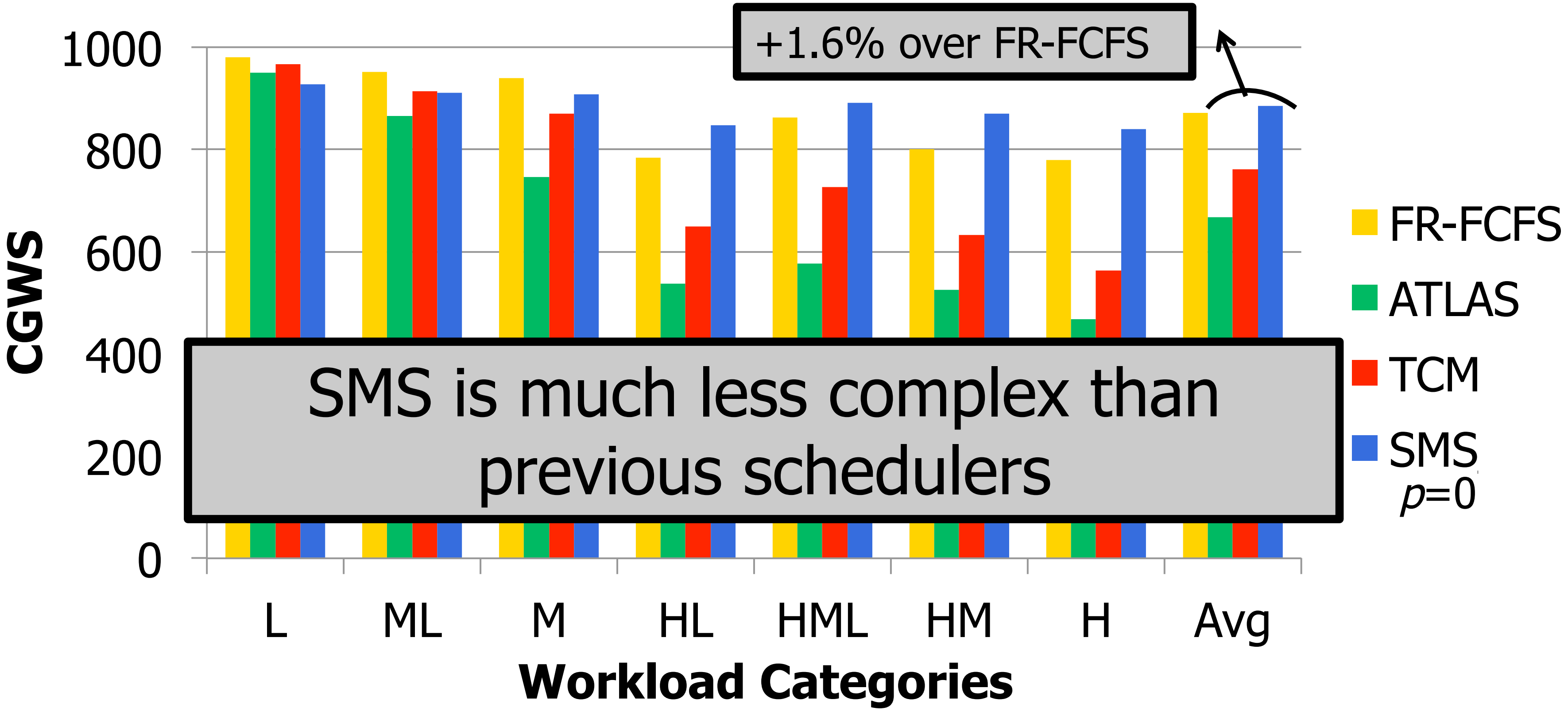
Evaluated System Scenario: GPU Focused

- GPU has high weight (weight = 1000)

$$CGWS = CPU_{WS} + GPU_{Speedup} * \underbrace{GPU_{Weight}}_{1000}$$

- 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

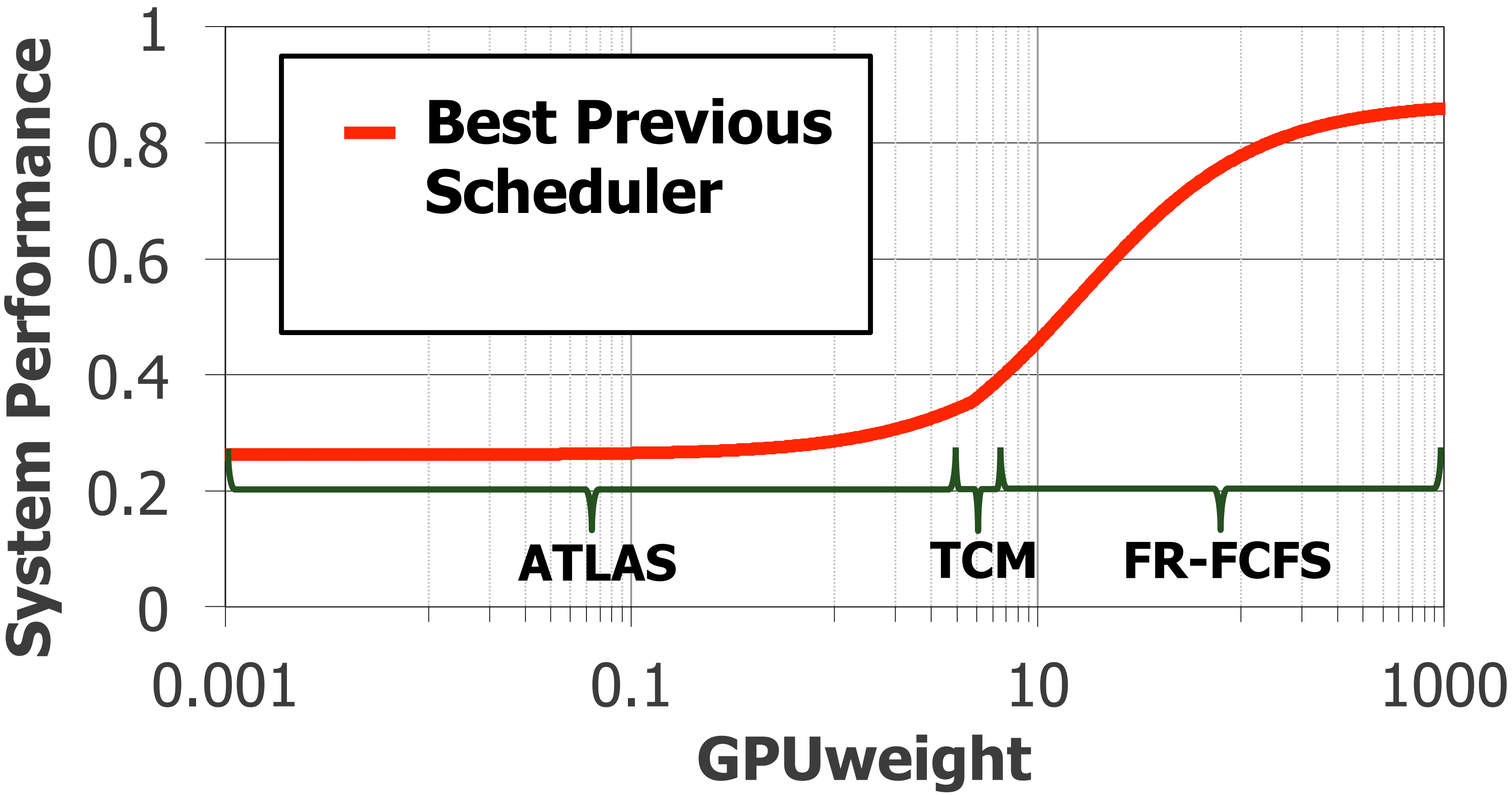


+1.6% over FR-FCFS

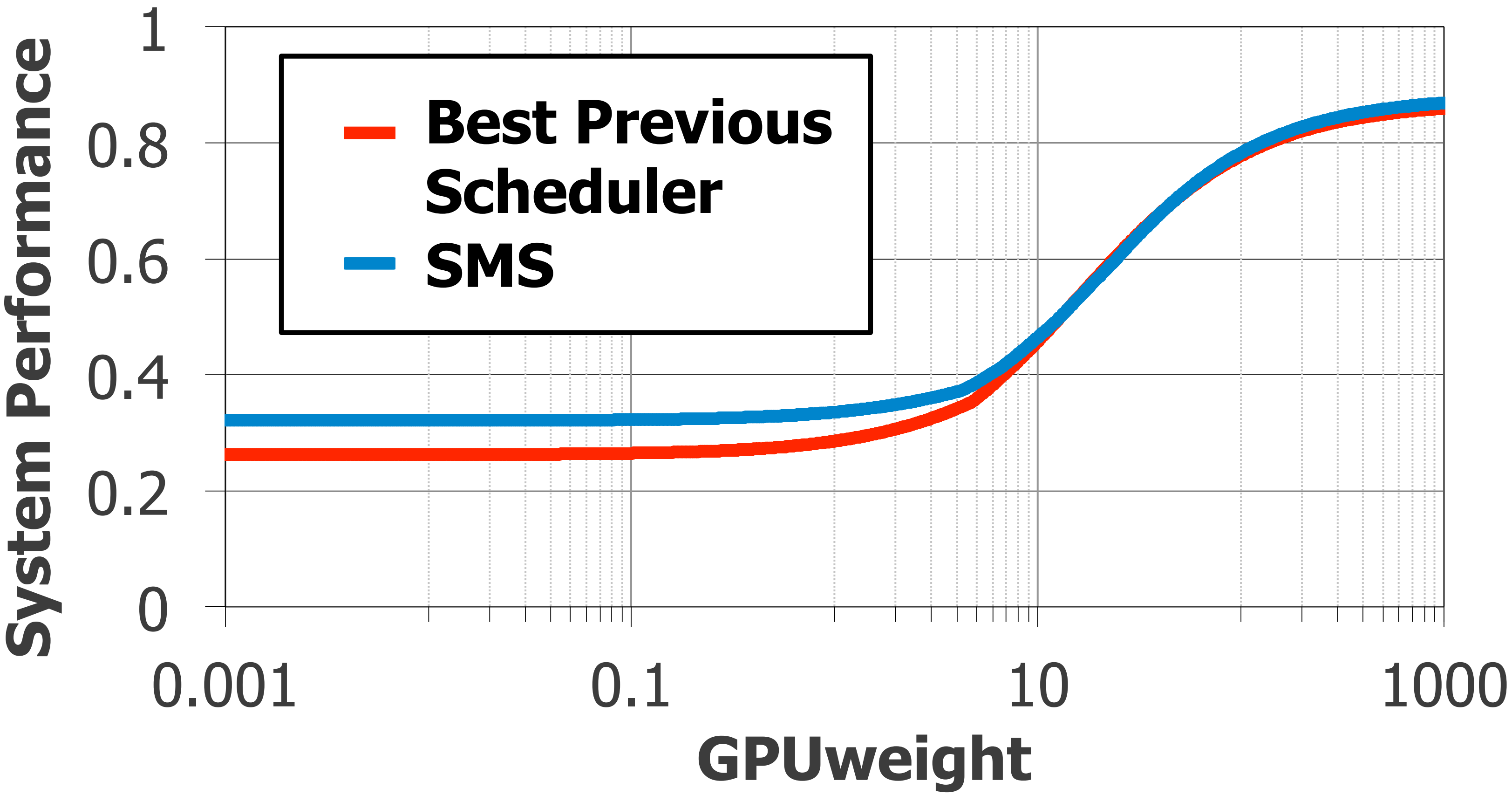
SMS is much less complex than previous schedulers

- 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

Strong Memory Service Guarantees

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

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

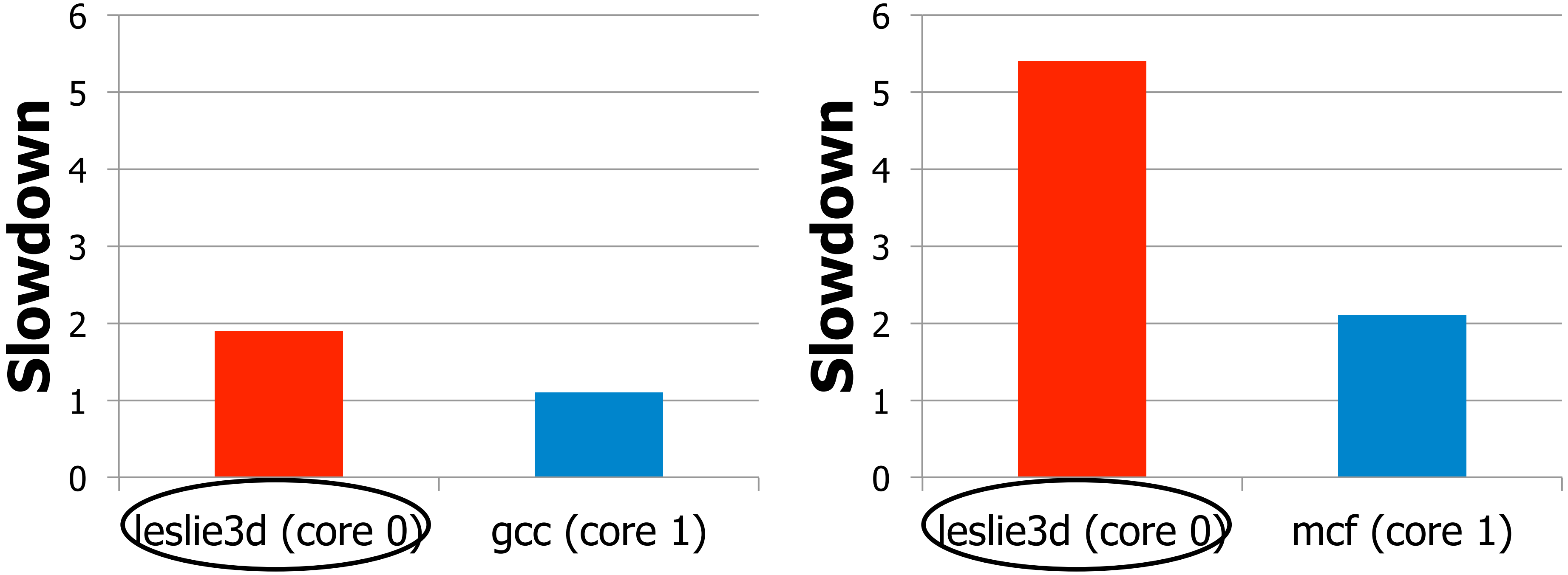
MISE: Providing Performance Predictability in Shared Main Memory Systems

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

SAFARI

Carnegie Mellon

Unpredictable Application Slowdowns



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

Need for Predictable Performance

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

Our Goal: Predictable performance
in the presence of memory interference

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

Outline

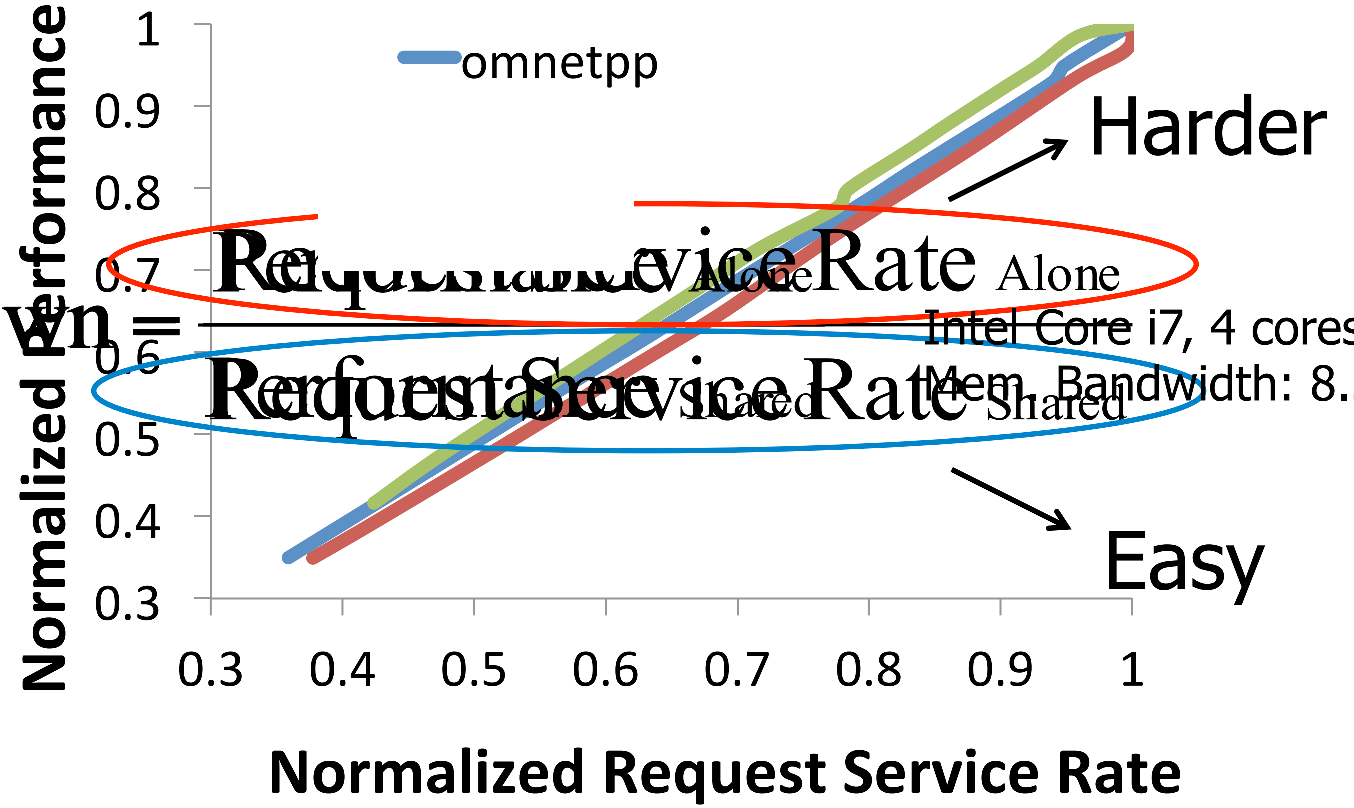
1. Estimate Slowdown
   - Key Observations
   - Implementation
   - MISE Model: Putting it All Together
   - Evaluating the Model
2. Control Slowdown
   - Providing Soft Slowdown Guarantees
   - Minimizing Maximum Slowdown

Slowdown: Definition

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

Key Observation 1

For a memory bound application,
Performance ∝ Memory request service rate



Intel Core i7, 4 cores
Mem. Bandwidth: 8.5 GB/s

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

Key Observation 2

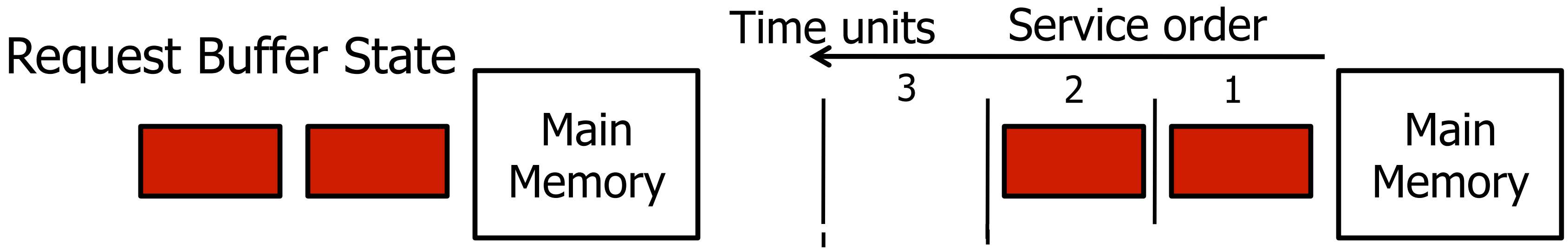
Request Service Rate $_{\text{Alone}}$ ($\text{RSR}_{\text{Alone}}$) of an application can be estimated by giving the application highest priority in accessing memory

Highest priority → Little interference

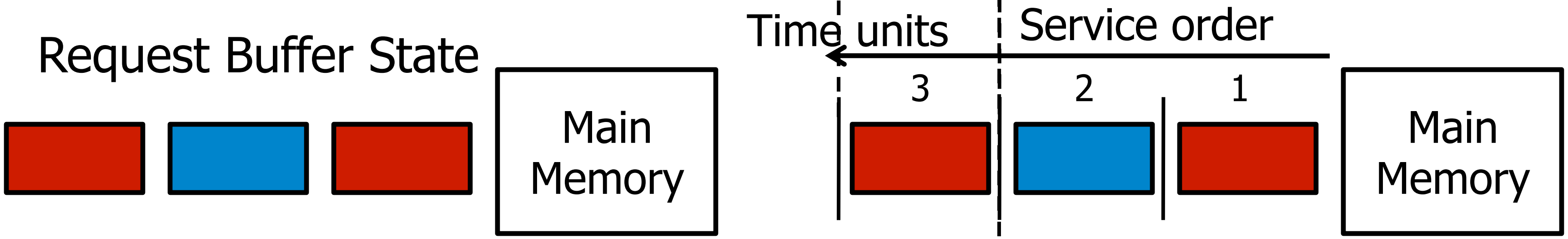
(almost as if the application were run alone)

Key Observation 2

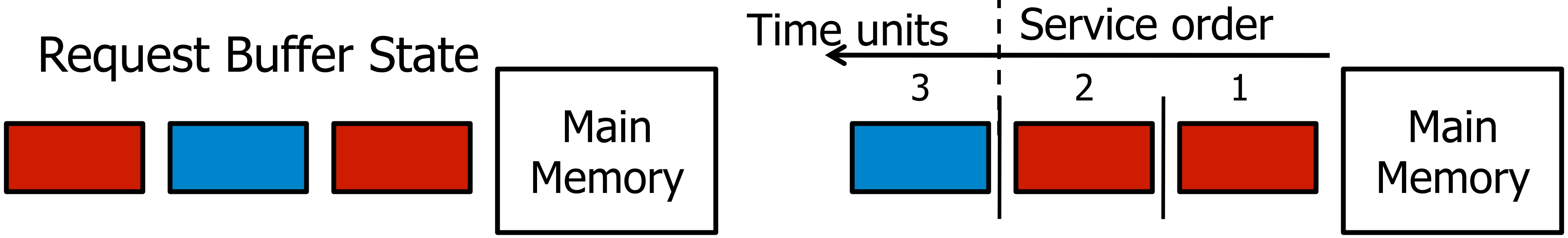
1. Run alone



2. Run with another application



3. Run with another application: highest priority

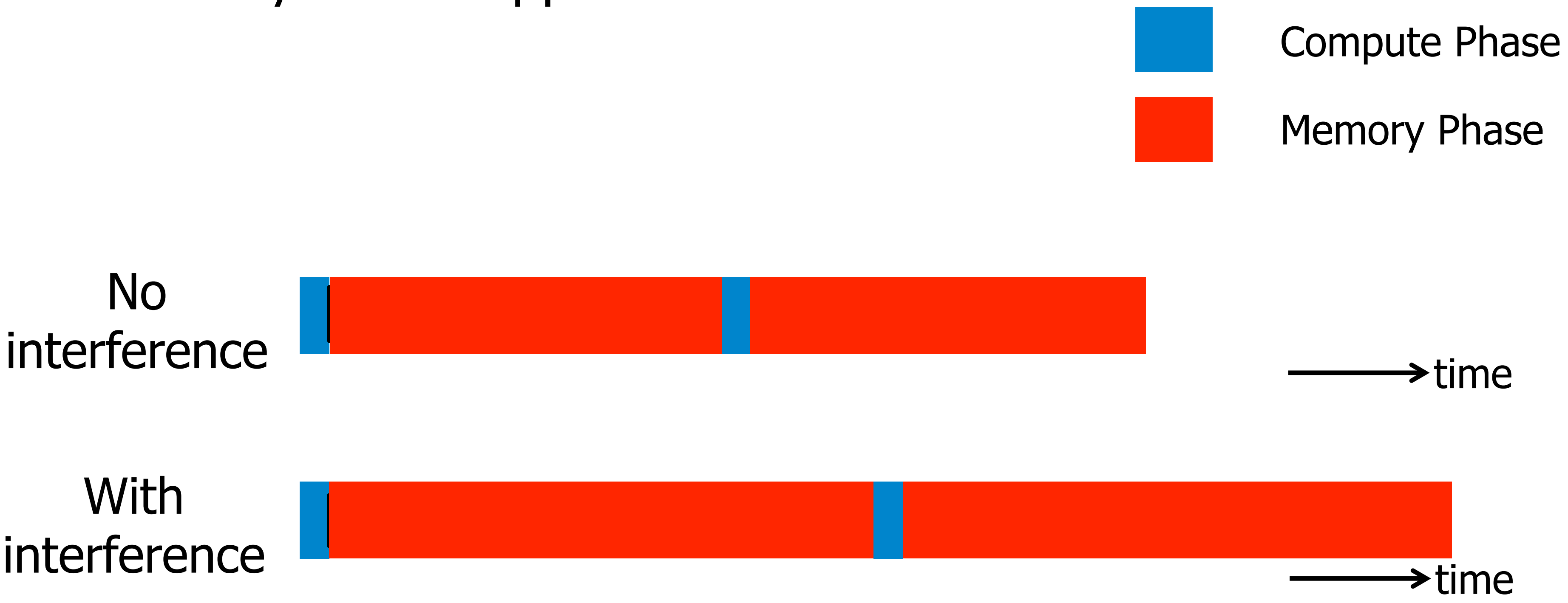


Memory Interference-induced Slowdown Estimation (MISE) model for memory bound applications

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

Key Observation 3

- Memory-bound application



🟦 Compute Phase

🟥 Memory Phase

No interference

With interference

time

time

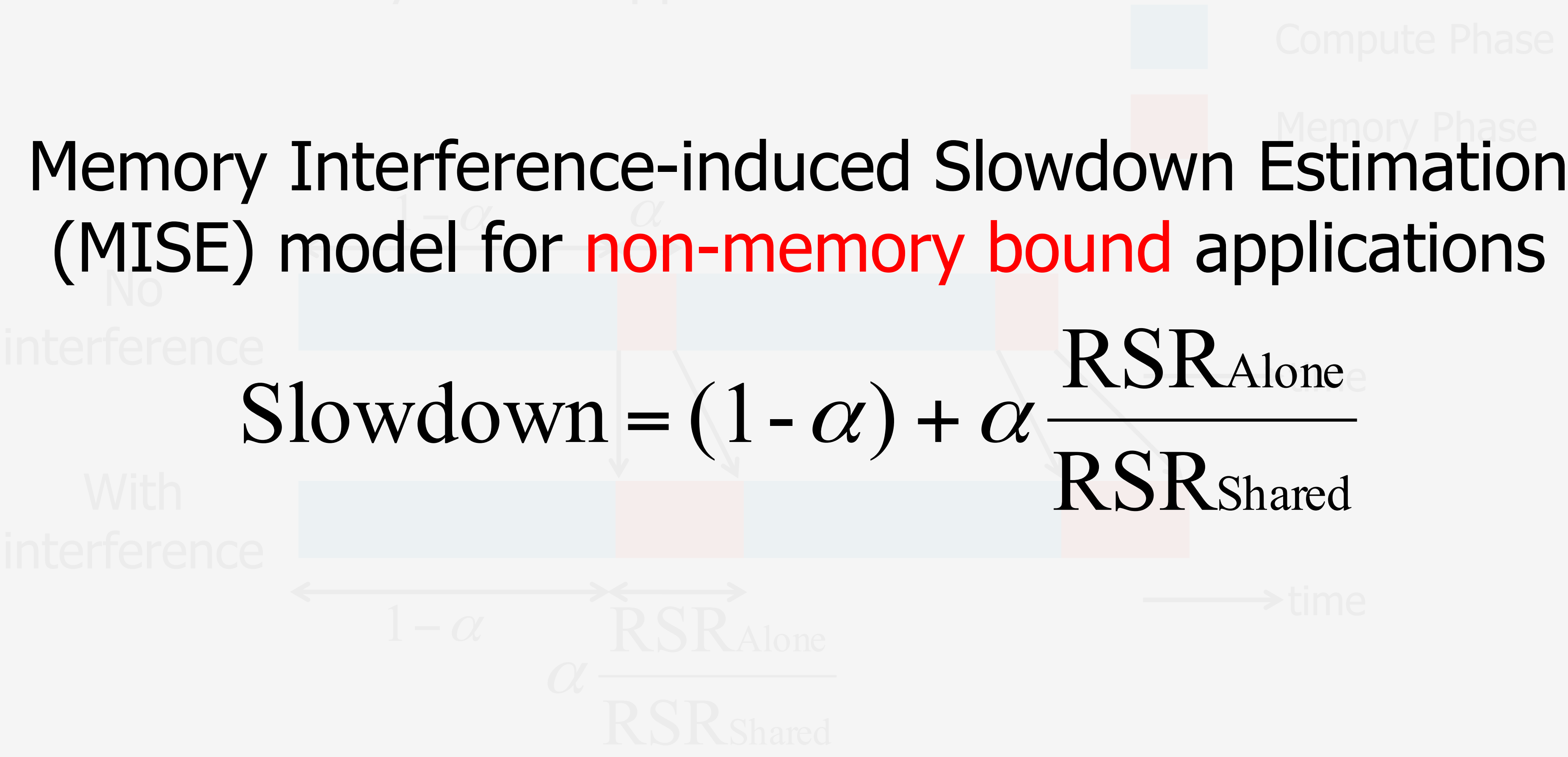
Memory phase slowdown dominates overall slowdown

Key Observation 3

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

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

- Non-memory-bound application



Only memory fraction (α) slows down with interference

Measuring $\text{RSR}_{\text{Shared}}$ and α

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

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

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

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

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

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

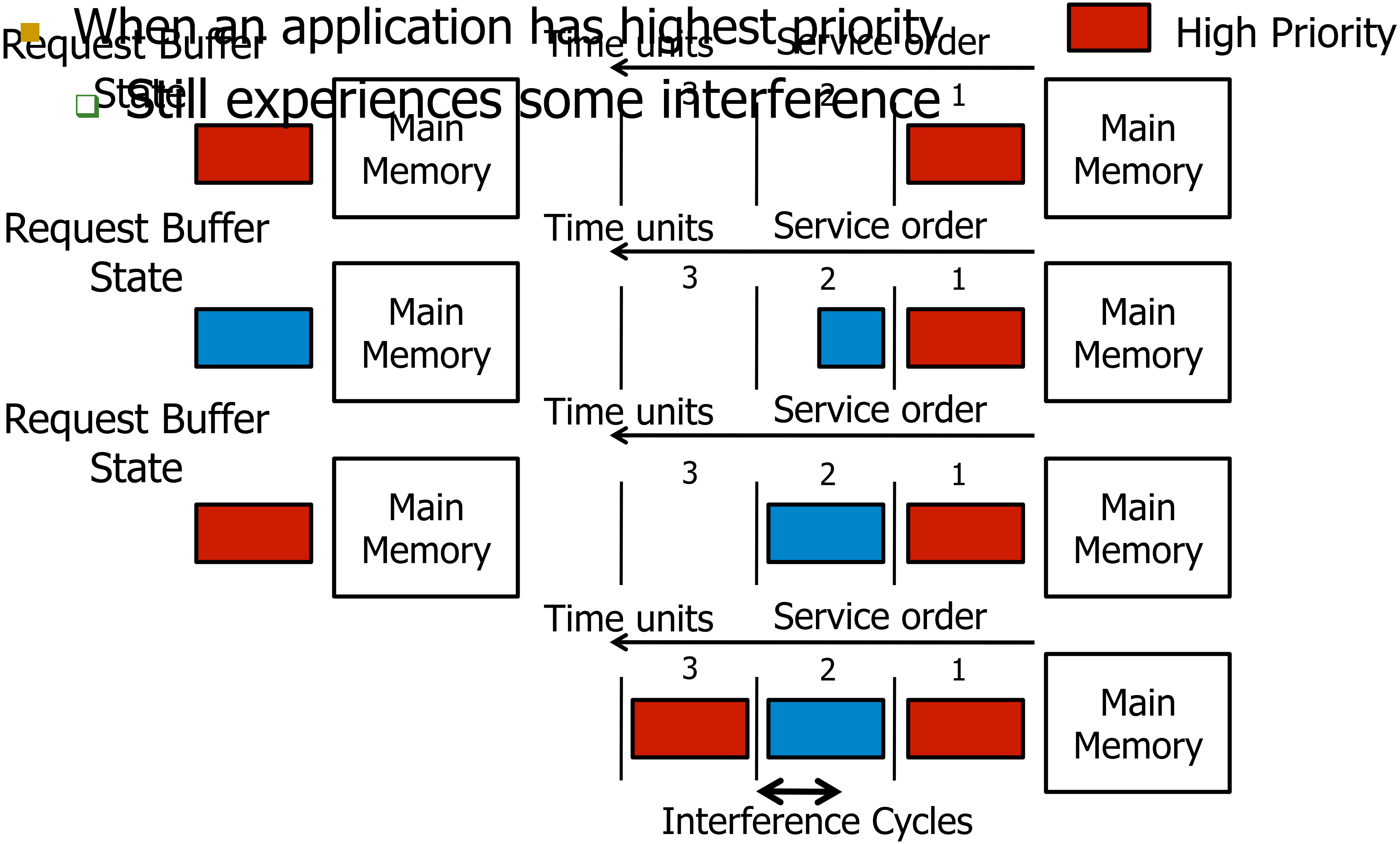
How: Periodically give each application highest priority in accessing memory

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

SAFARI

Inaccuracy in Estimating $\text{RSR}_{\text{Alone}}$

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



Accounting for Interference in $\text{RSR}_{\text{Alone}}$ Estimation

- Solution: Determine and remove interference cycles from $\text{RSR}_{\text{Alone}}$ calculation

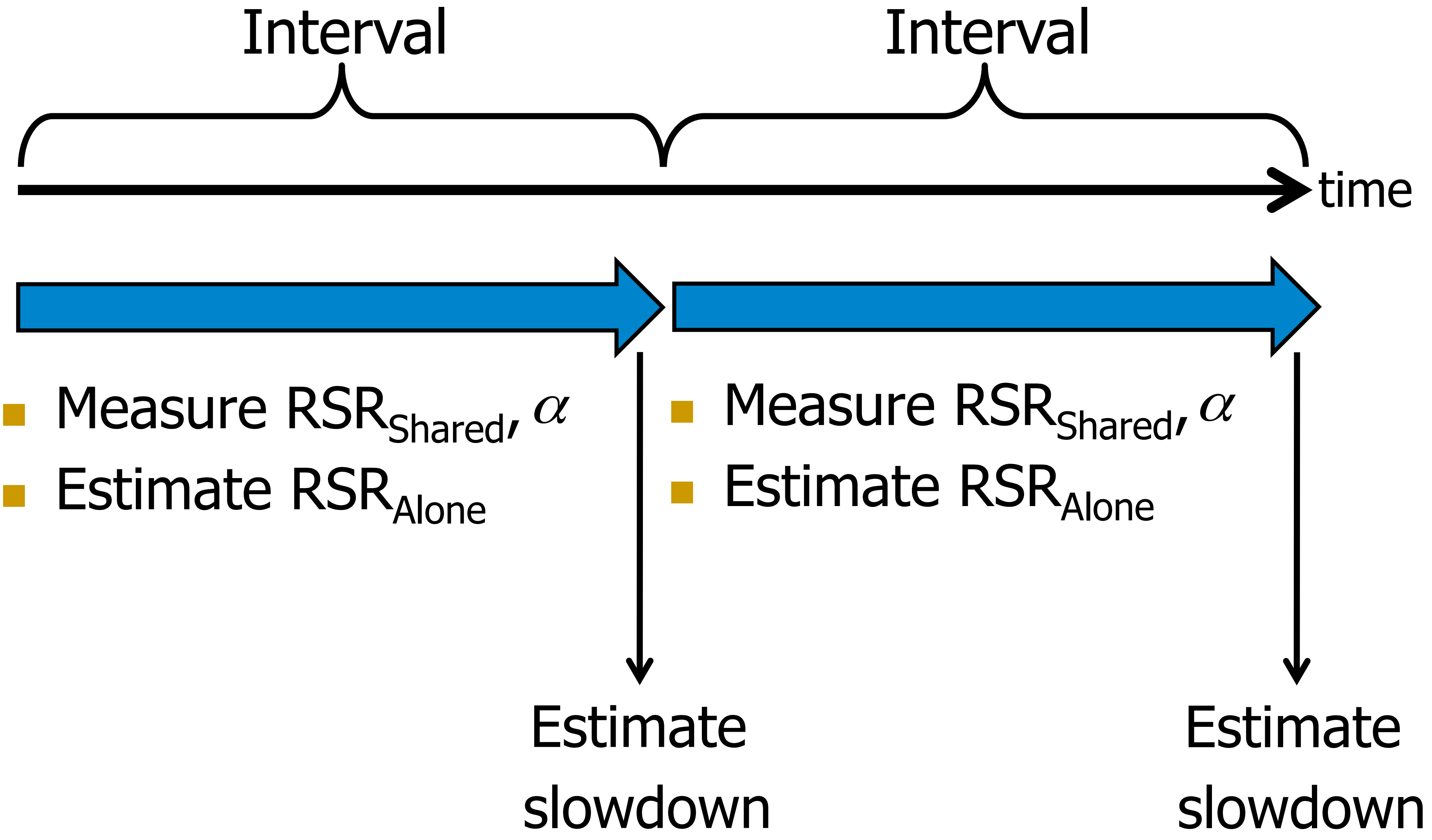
$$\text{RSR}_{\text{Alone}} = \frac{\text{Number of Requests During High Priority Epochs}}{\text{Number of Cycles Application Given High Priority} \boxed{- \text{Interference Cycles}}}$$

- 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



Interval

Interval

time

- Measure $\mathrm{RSR}_{\mathrm{Shared}}$, α
- Estimate $\mathrm{RSR}_{\mathrm{Alone}}$

- Measure $\mathrm{RSR}_{\mathrm{Shared}}$, α
- Estimate $\mathrm{RSR}_{\mathrm{Alone}}$

Estimate slowdown

Estimate slowdown

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:

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

Hard

Easy

Count number of cycles application receives interference

Two Major Advantages of MISE Over STFM

- Advantage 1:
  - STFM estimates alone performance while an application is receiving interference → Hard
  - MISE estimates alone performance while giving an application the highest priority → Easier

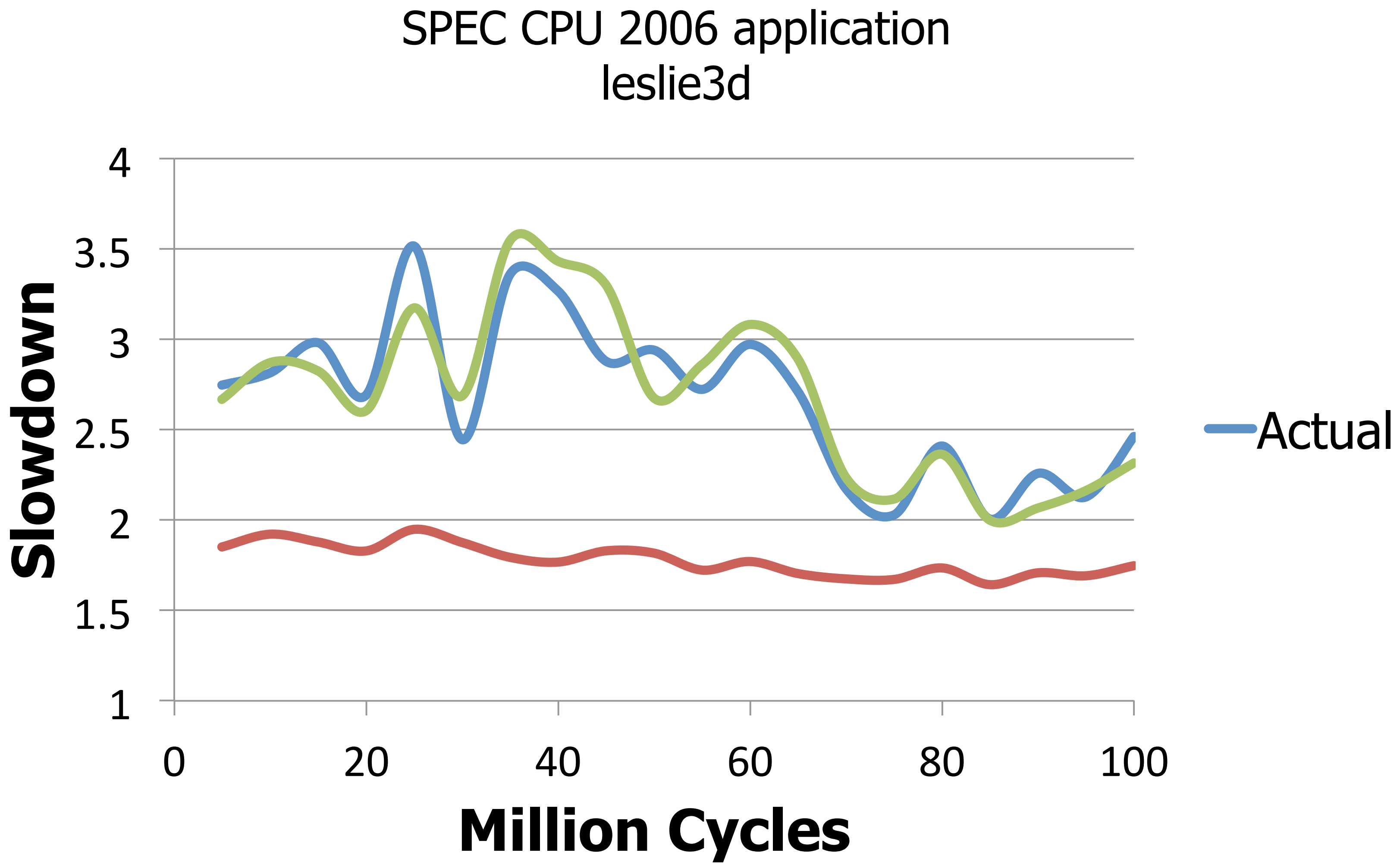
- Advantage 2:
  - STFM does not take into account compute phase for non-memory-bound applications
  - MISE accounts for compute phase → Better accuracy

Methodology

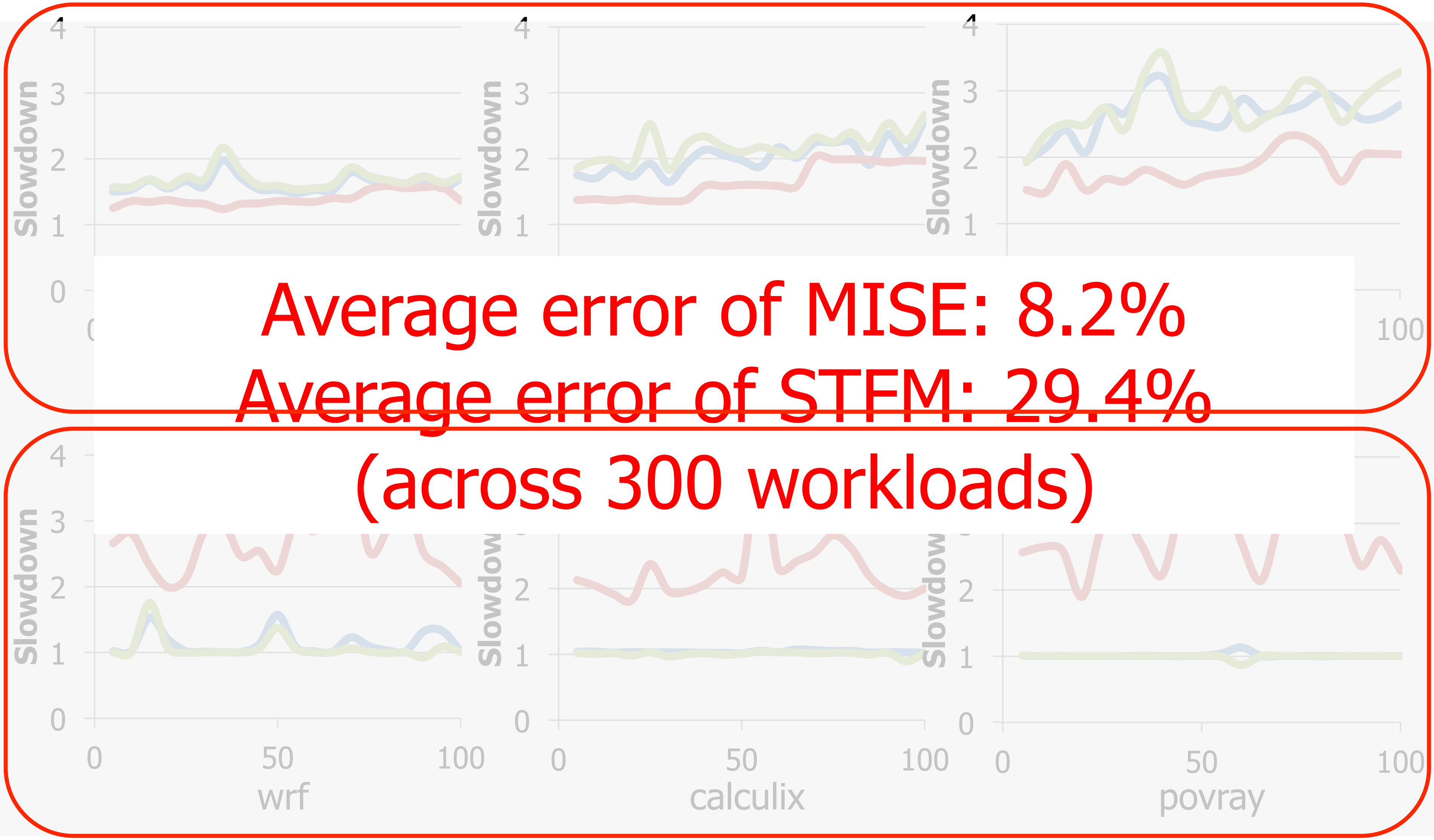
- Configuration of our simulated system
  - 4 cores
  - 1 channel, 8 banks/channel
  - DDR3 1066 DRAM
  - 512 KB private cache/core

- Workloads
  - SPEC CPU2006
  - 300 multi programmed workloads

Quantitative Comparison



Comparison to STFM



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

Providing “Soft” Slowdown Guarantees

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

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

MISE-QoS: Mechanism to Provide Soft QoS

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

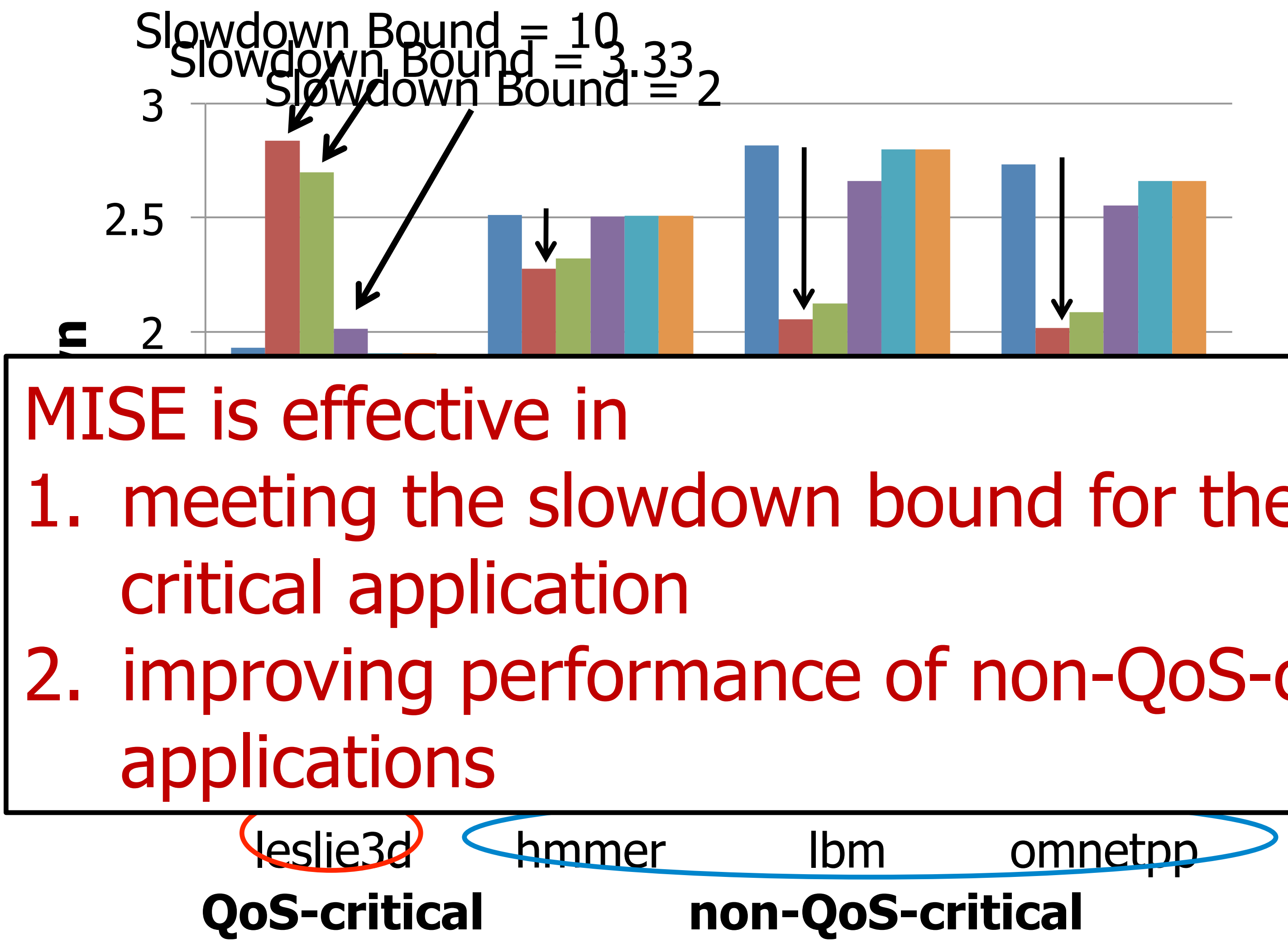
Methodology

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

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

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

A Look at One Workload



MISE is effective in

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

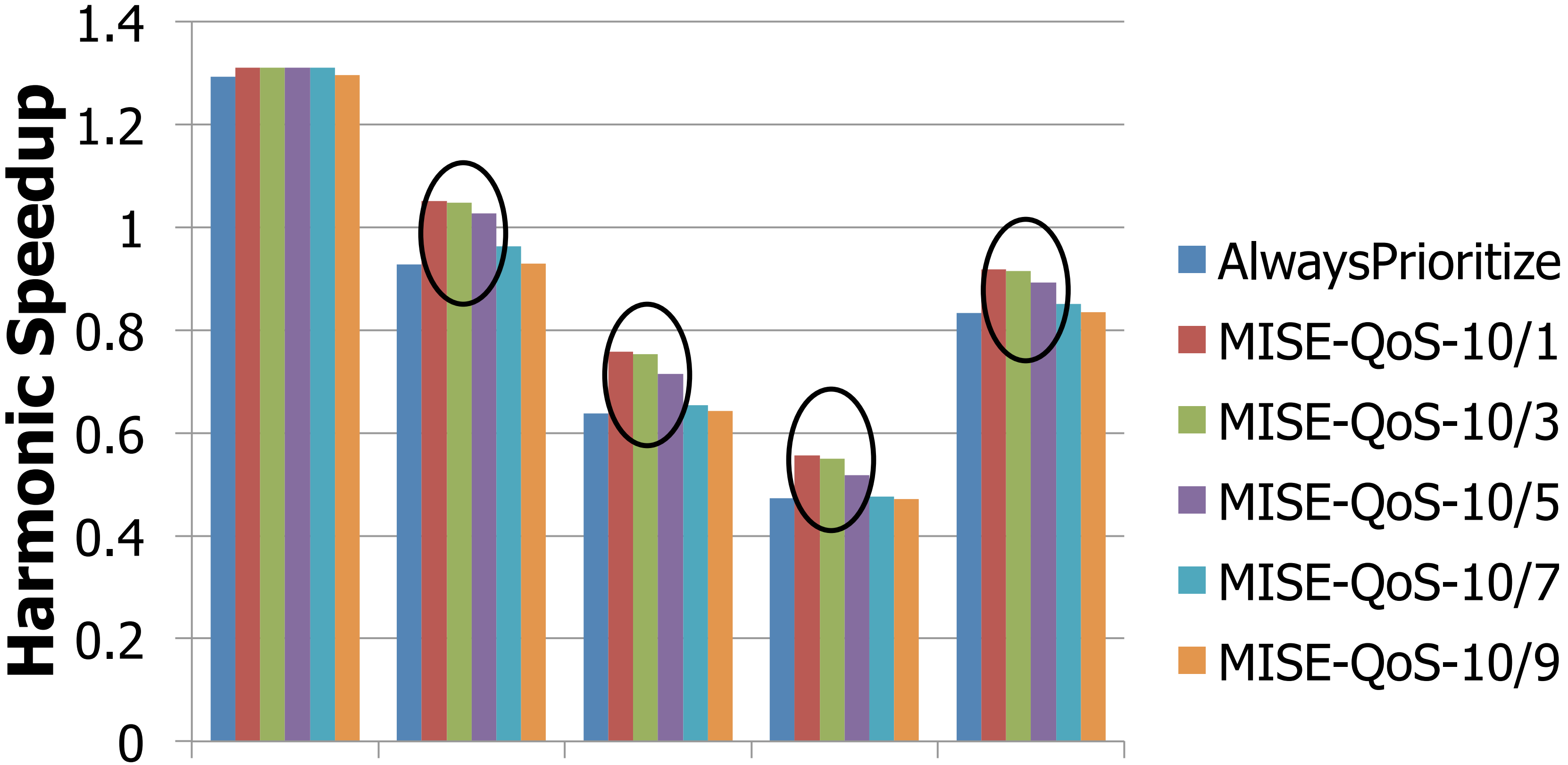
Effectiveness of MISE in Enforcing QoS

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 $_{\text{Alone}}$ 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

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

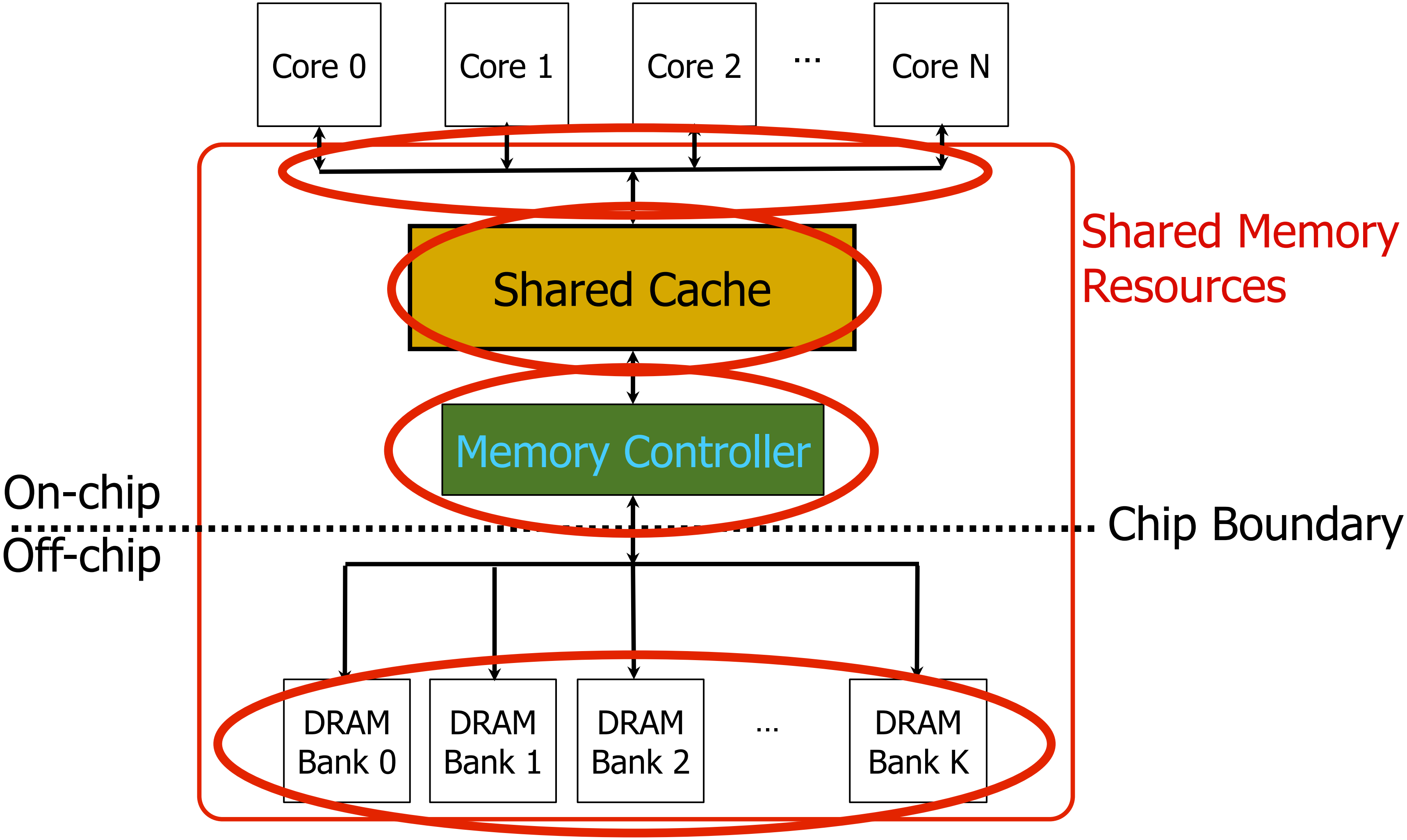
- Dumb resources: Keep each resource free-for-all, but reduce/control interference by injection control or data mapping
  - Source throttling to control access to memory system [Ebrahimi+ ASPLOS'10, ISCA'11, TOCS'12] [Ebrahimi+ MICRO'09] [Nychis+ HotNets'10]
  - QoS-aware data mapping to memory controllers [Muralidhara+ MICRO'11]
  - QoS-aware thread scheduling to cores

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

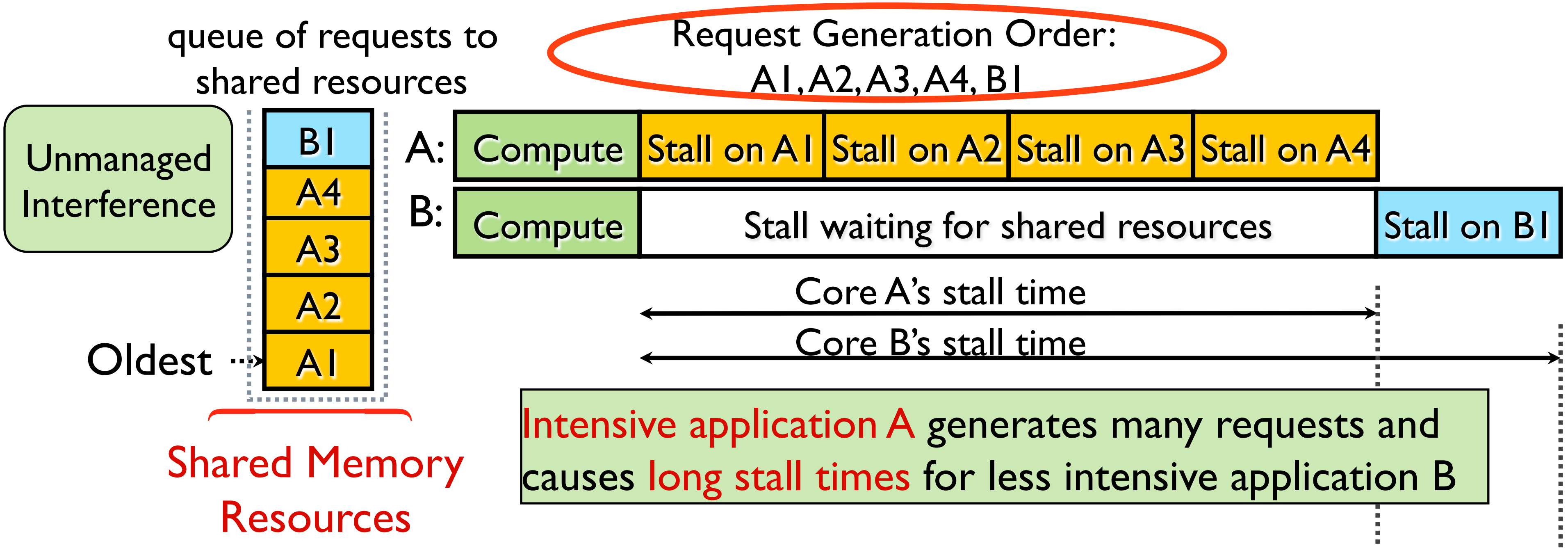
- How do we enable fair sharing of the entire memory system by controlling interference in a coordinated manner?

An Alternative Approach: Source Throttling

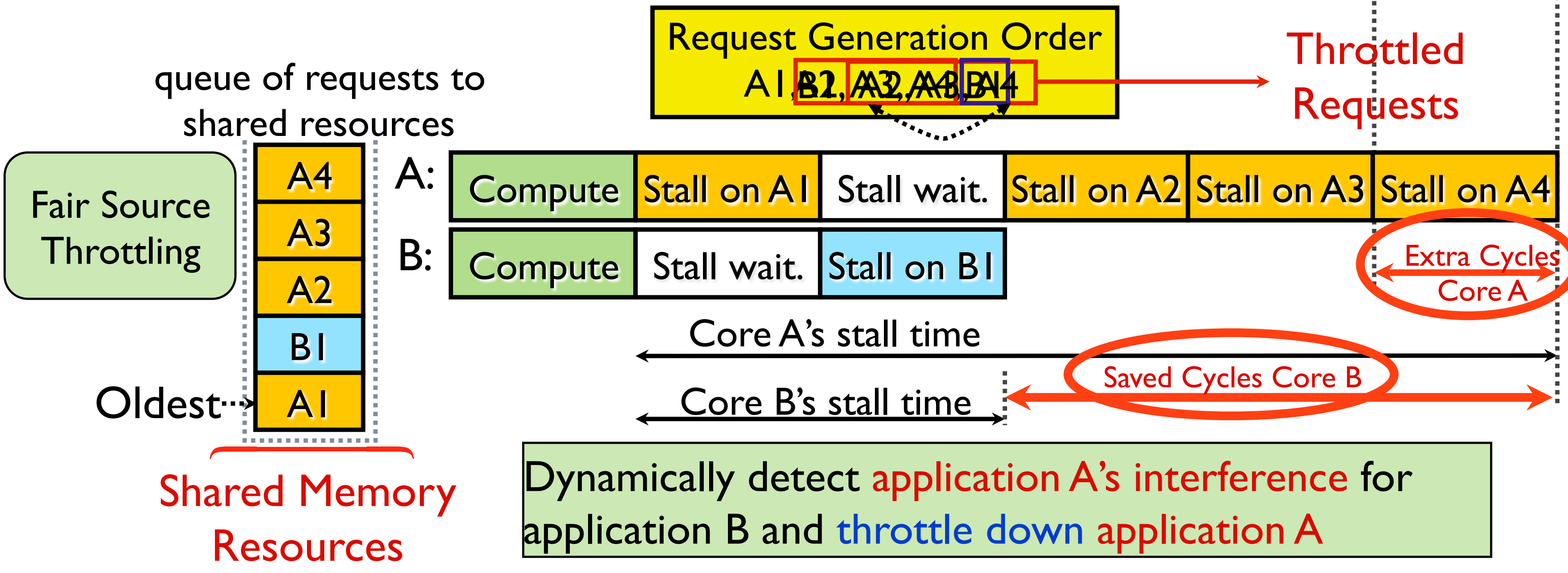
- Manage inter-thread interference at the cores, not at the shared resources

- 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.



Intensive application A generates many requests and causes long stall times for less intensive application B



Dynamically detect application A's interference for application B and throttle down application A

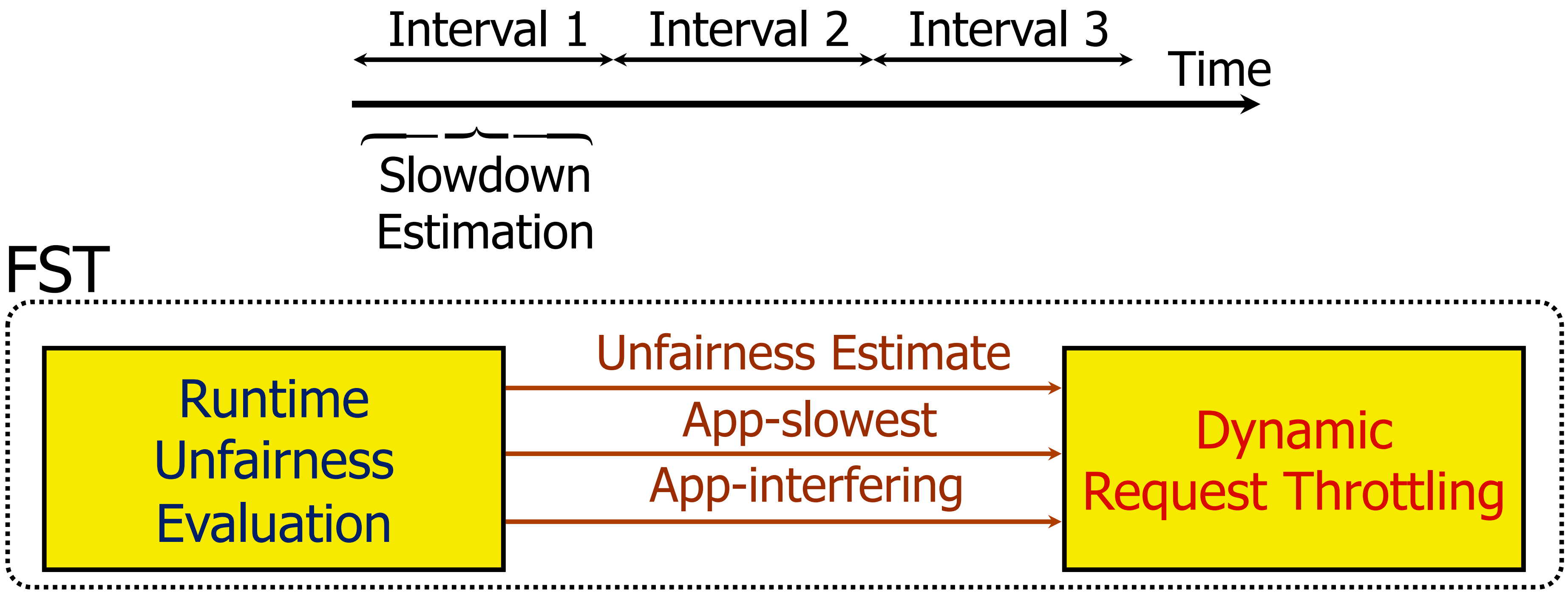
Fairness via Source Throttling (FST)

- 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)



Interval 1

Interval 2

Interval 3

Time

Slowdown Estimation

FST

Runtime Unfairness Evaluation

Unfairness Estimate

App-slowest

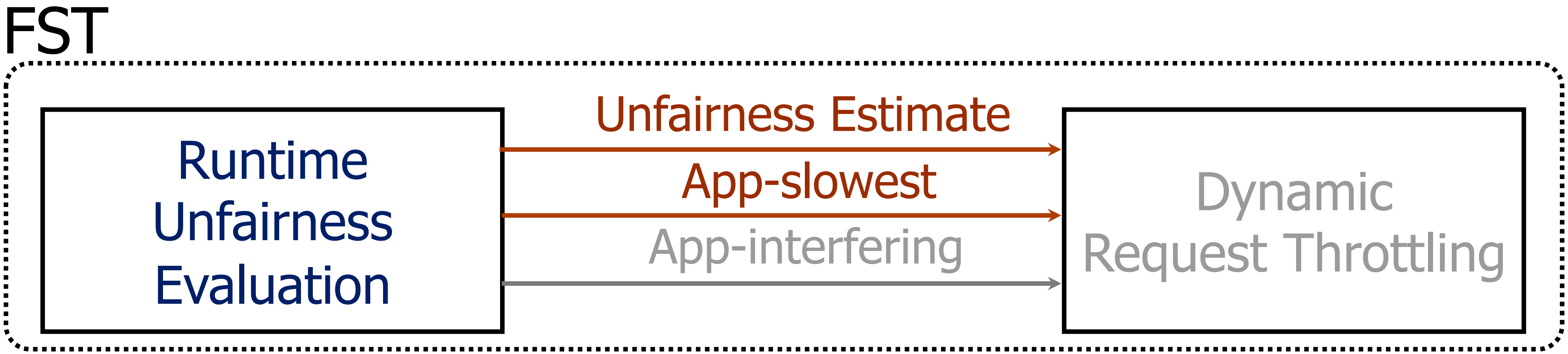
App-interfering

Dynamic Request Throttling

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)



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

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

Estimating System Unfairness

- Unfairness = $\dfrac{\text{Max\{Slowdown i\} over all applications i}}{\text{Min\{Slowdown i\} over all applications i}}$

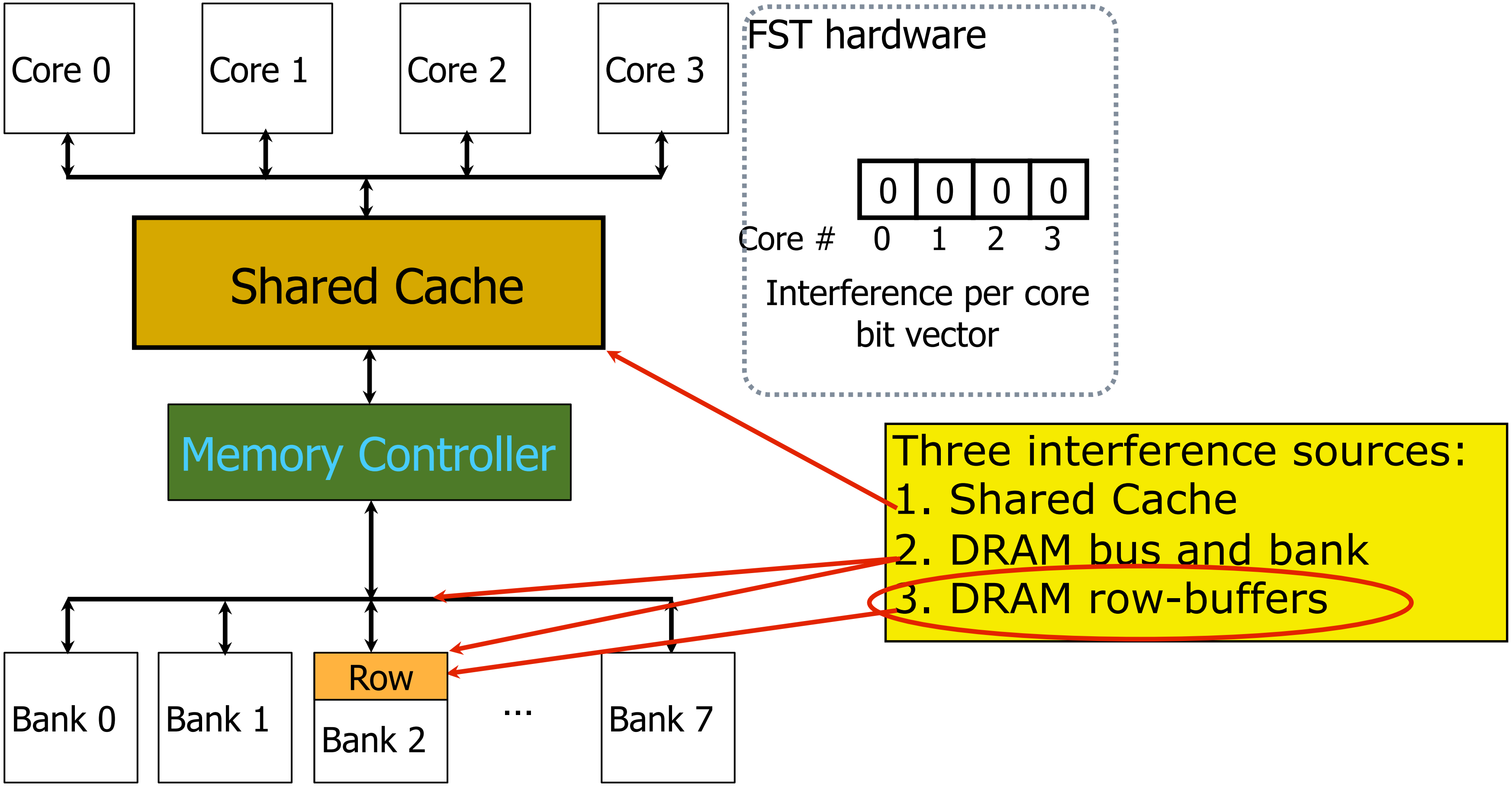
- Slowdown of application i = $\dfrac{Ti^{Shared}}{Ti^{Alone}}$

- How can Ti^{Alone} be estimated in shared mode?

- Ti^{Excess} is the number of extra cycles it takes application i to execute due to interference

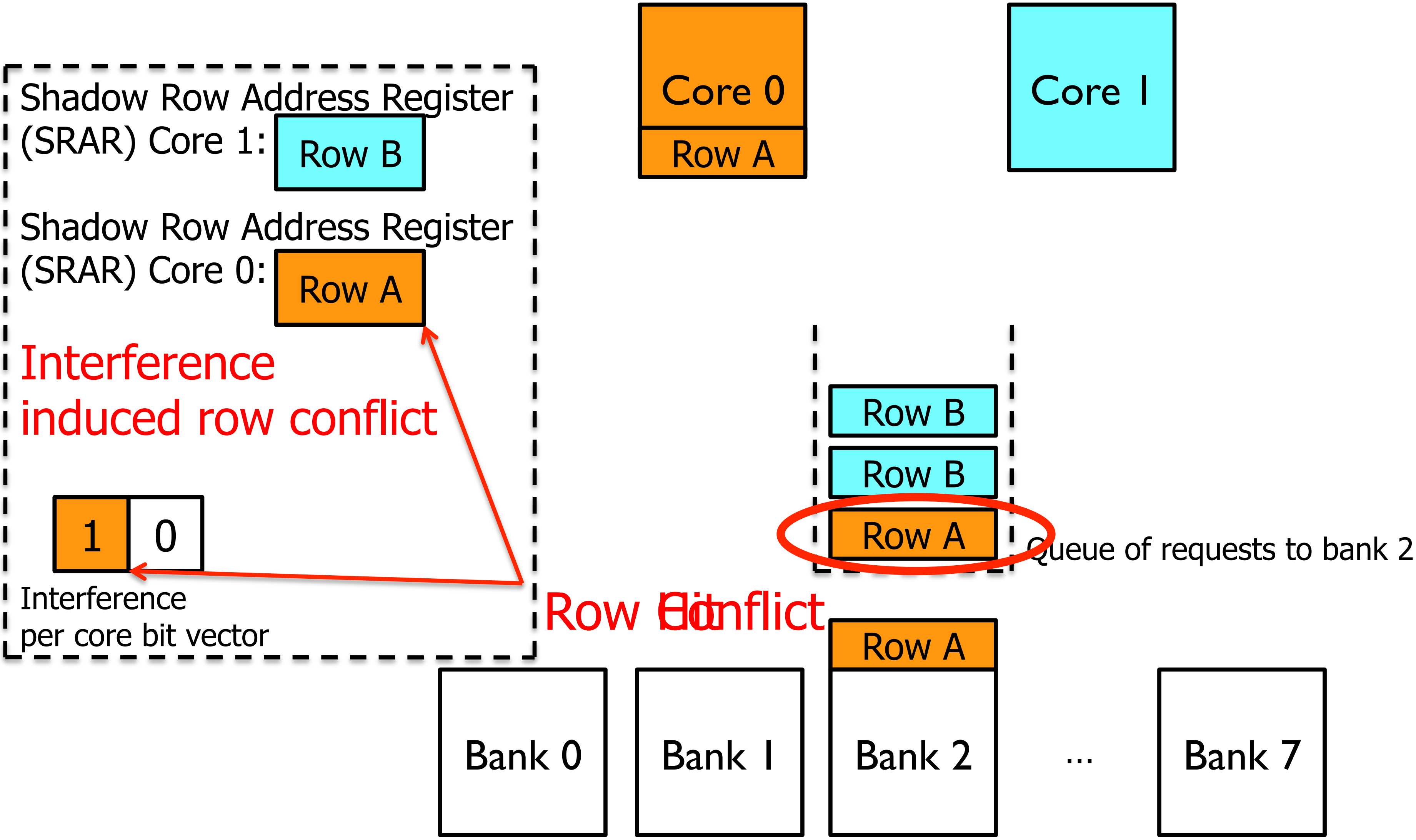
- $Ti^{Alone} = Ti^{Shared} - \boxed{Ti^{Excess}}$

Tracking Inter-Core Interference



Three interference sources:
1. Shared Cache
2. DRAM bus and bank
3. DRAM row-buffers

Tracking DRAM Row-Buffer Interference



Core 0

Row A

Core 1

Shadow Row Address Register (SRAR) Core 1: Row B

Shadow Row Address Register (SRAR) Core 0: Row A

Interference induced row conflict

1 0

Interference per core bit vector

Row B

Row B

Row A

Queue of requests to bank 2

Row Hit

Row Conflict

Row A

Bank 0

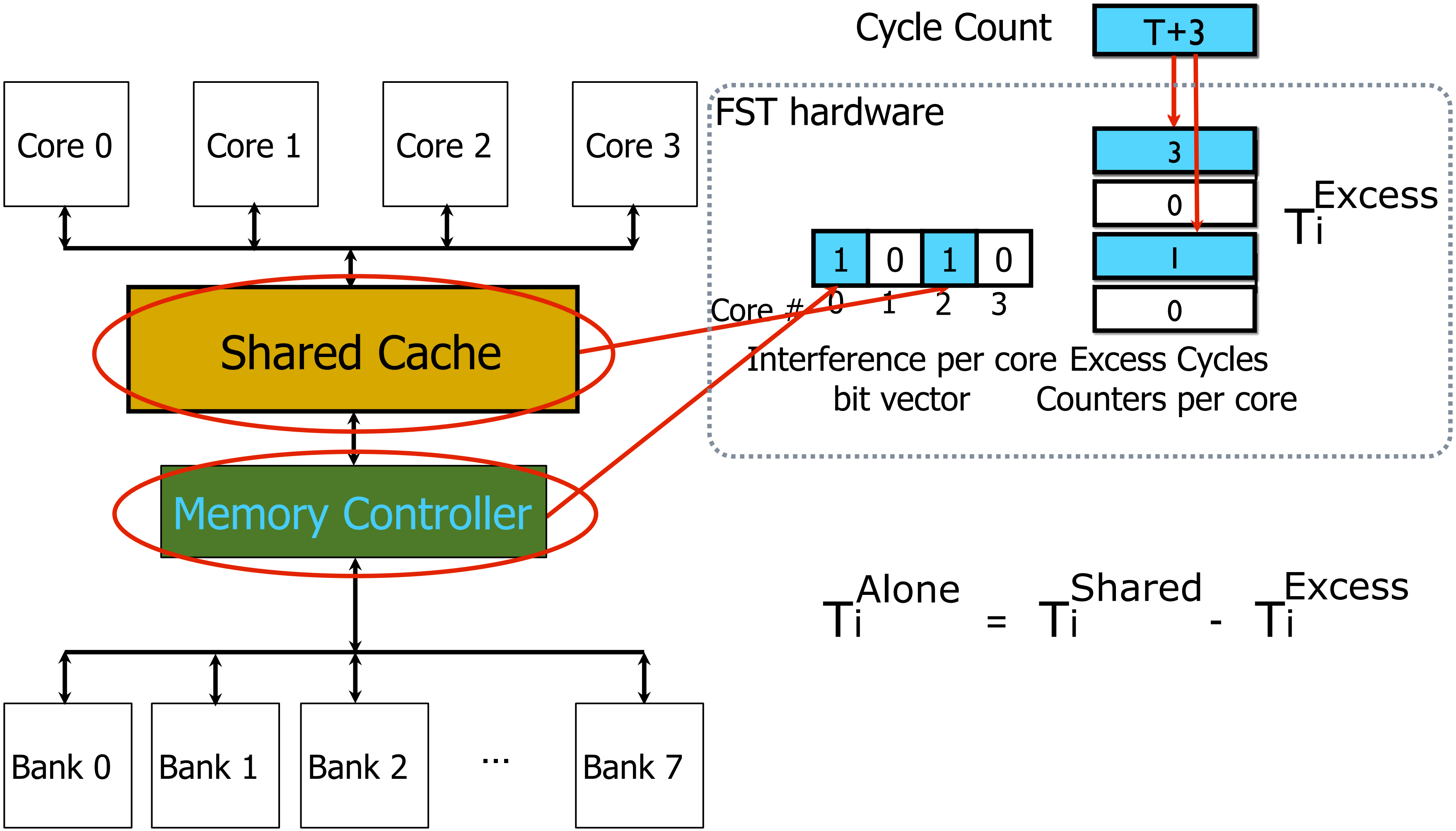
Bank 1

Bank 2

…

Bank 7

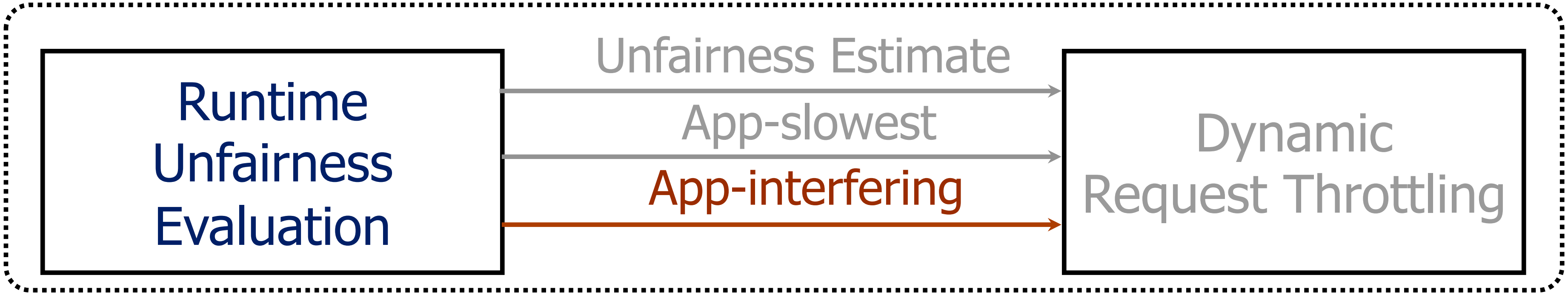
Tracking Inter-Core Interference



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

Fairness via Source Throttling (FST)

FST

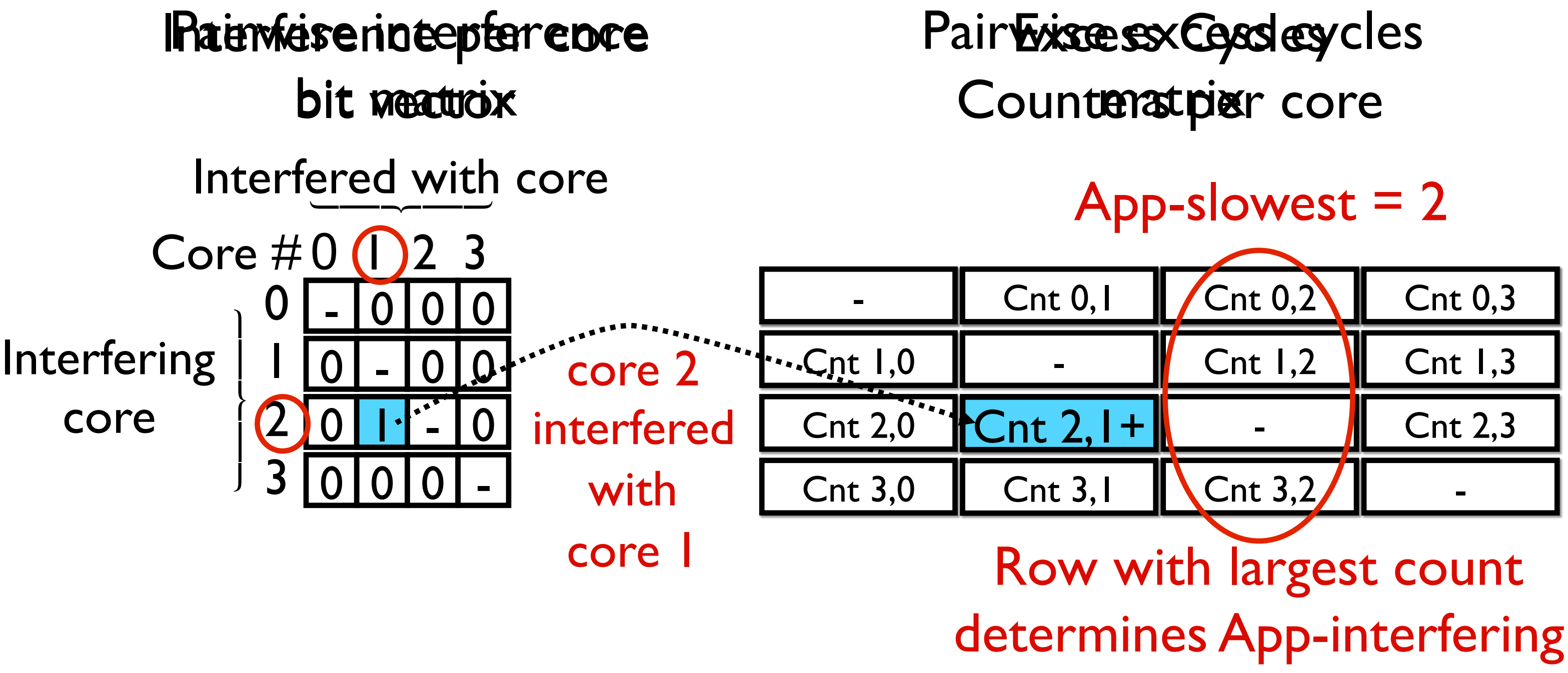


1- Estimating system unfairness
2- Find app. with the highest slowdown (App-slowest)
3- Find app. causing most interference for App-slowest (App-interfering)

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

Tracking Inter-Core Interference

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



Pairwise interference bit matrix

Interference per core bit vector

Interfered with core

Core #	0	1	2	3
0	-	0	0	0
1	0	-	0	0
2	0	1	-	0
3	0	0	0	-

core 2 interfered with core 1

Pairwise excess cycles matrix

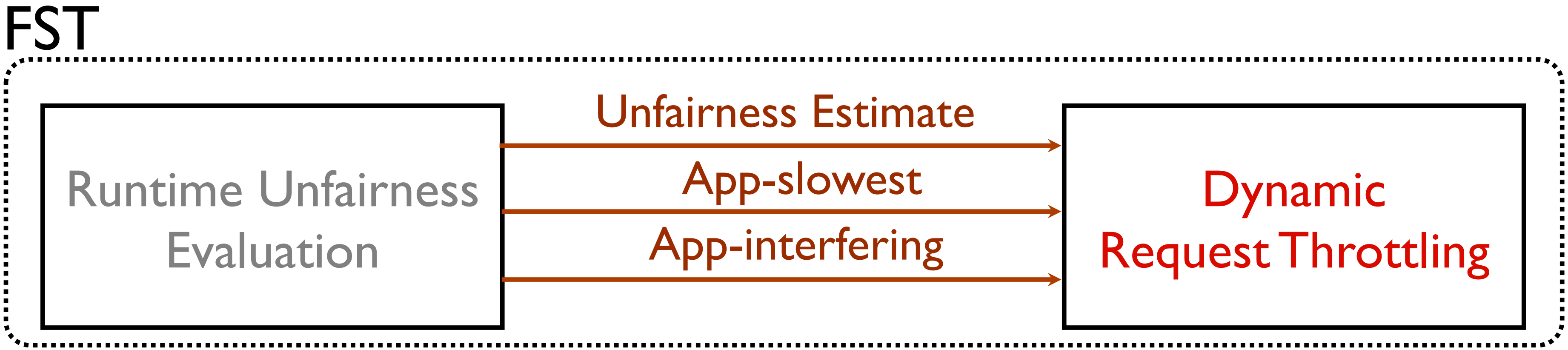
Excess Cycles Counters per core

App-slowest = 2

-	Cnt 0,1	Cnt 0,2	Cnt 0,3
Cnt 1,0	-	Cnt 1,2	Cnt 1,3
Cnt 2,0	Cnt 2,1+	-	Cnt 2,3
Cnt 3,0	Cnt 3,1	Cnt 3,2	-

Row with largest count determines App-interfering

Fairness via Source Throttling (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
}
```

Dynamic Request Throttling

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

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

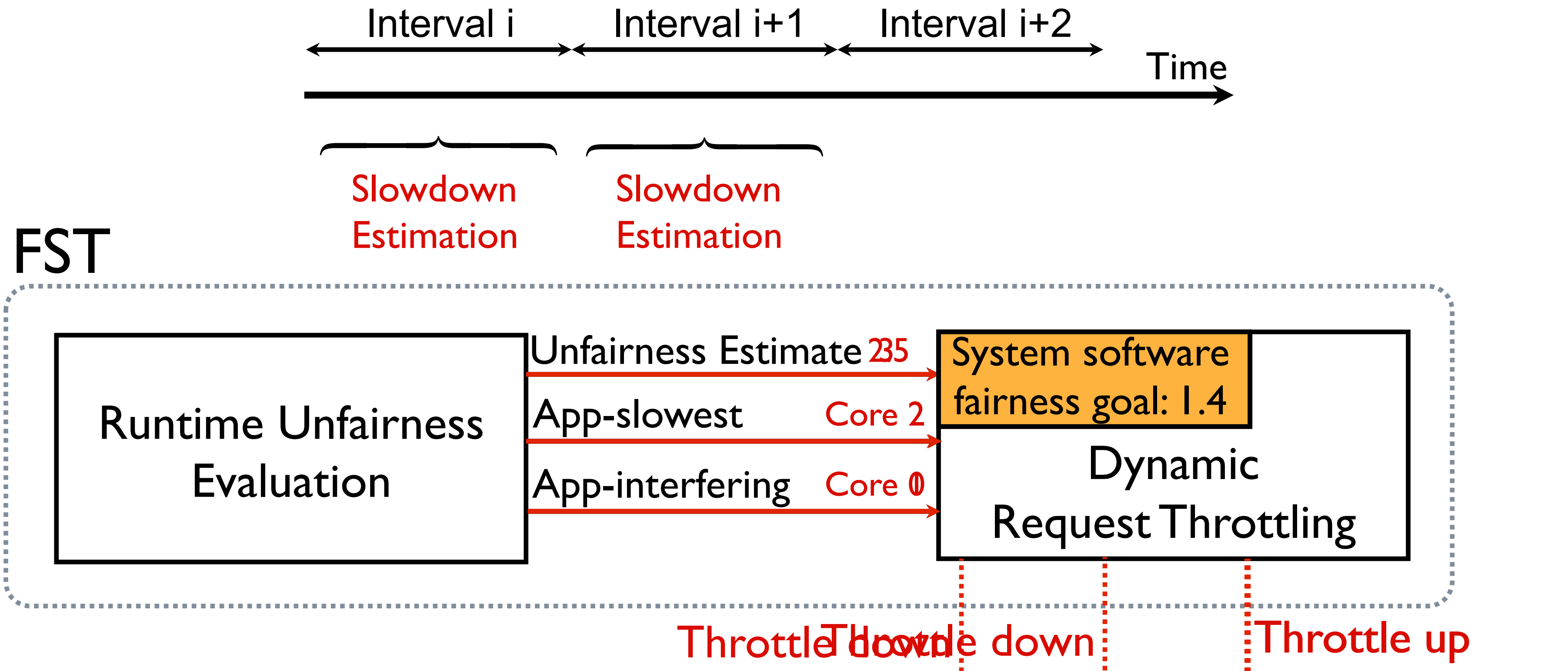
Dynamic Request Throttling

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

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

Total # of MSHRs: 128

FST at Work


Interval i
Interval i+1
Interval i+2
Time
Slowdown Estimation
Slowdown Estimation
FST
Runtime Unfairness Evaluation
Unfairness Estimate 2.5 3
App-slowest Core 2
App-interfering Core 0 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
  - 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

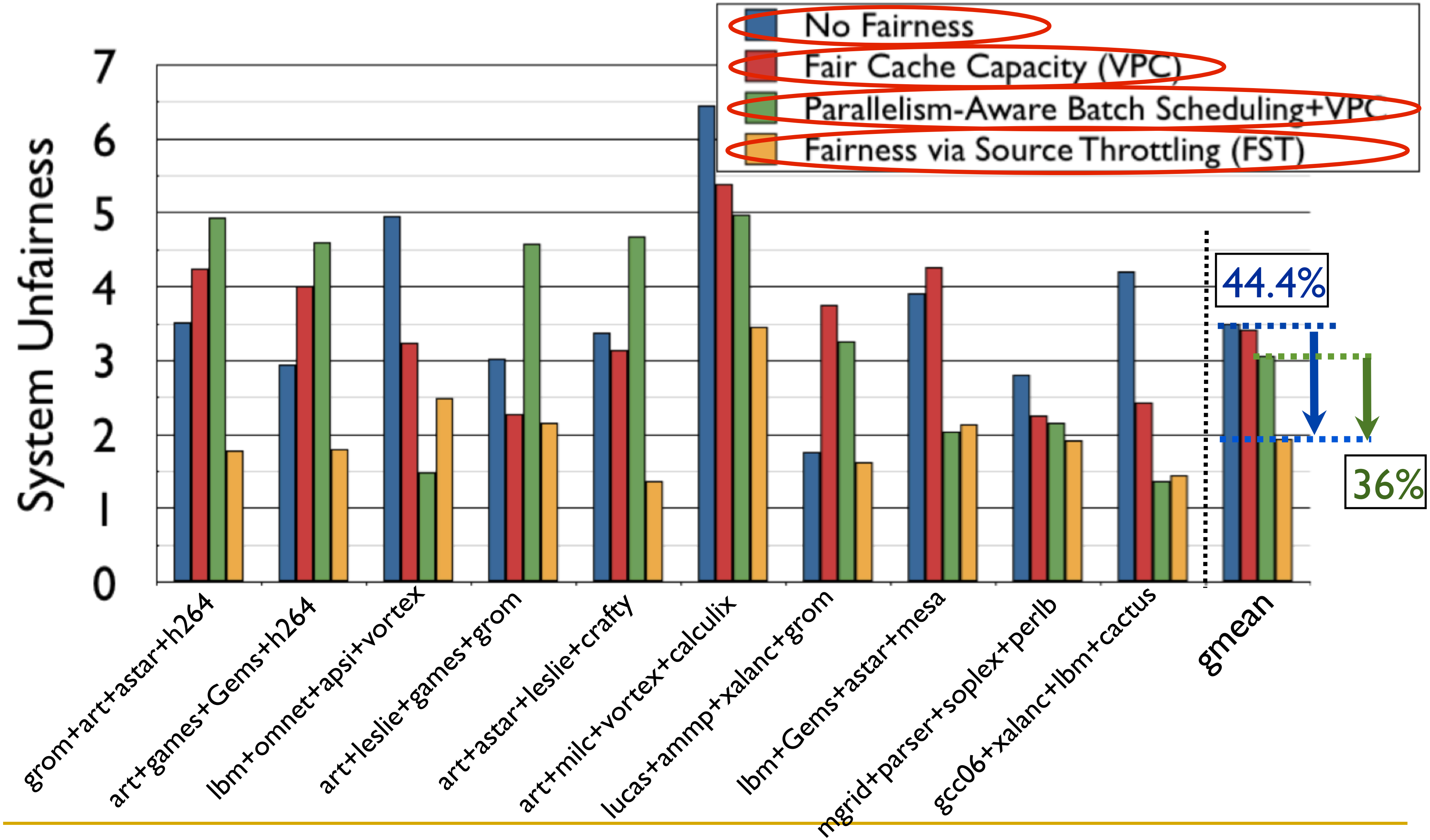
- Total storage cost required for 4 cores is ~12KB

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

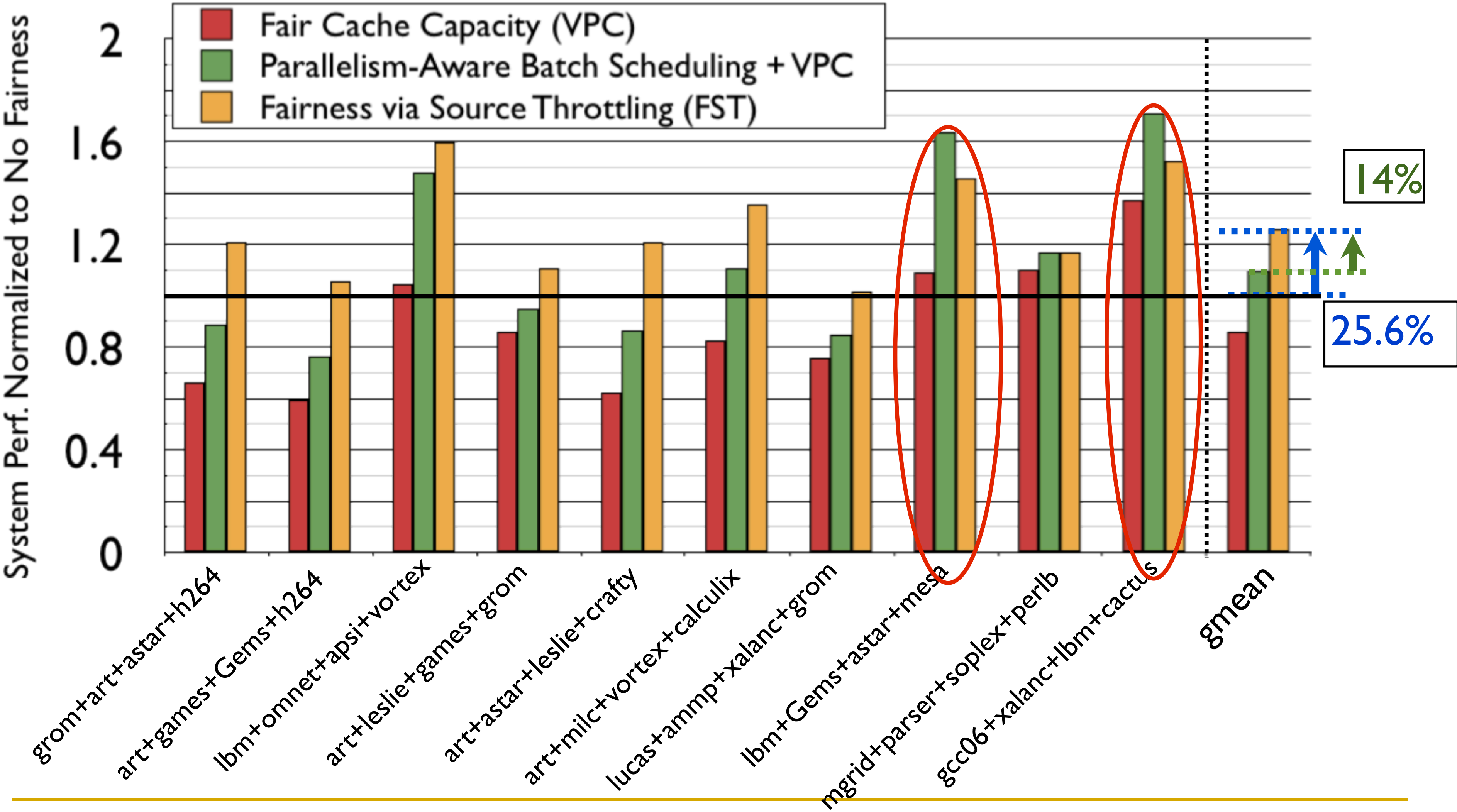
FST Evaluation Methodology

- 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

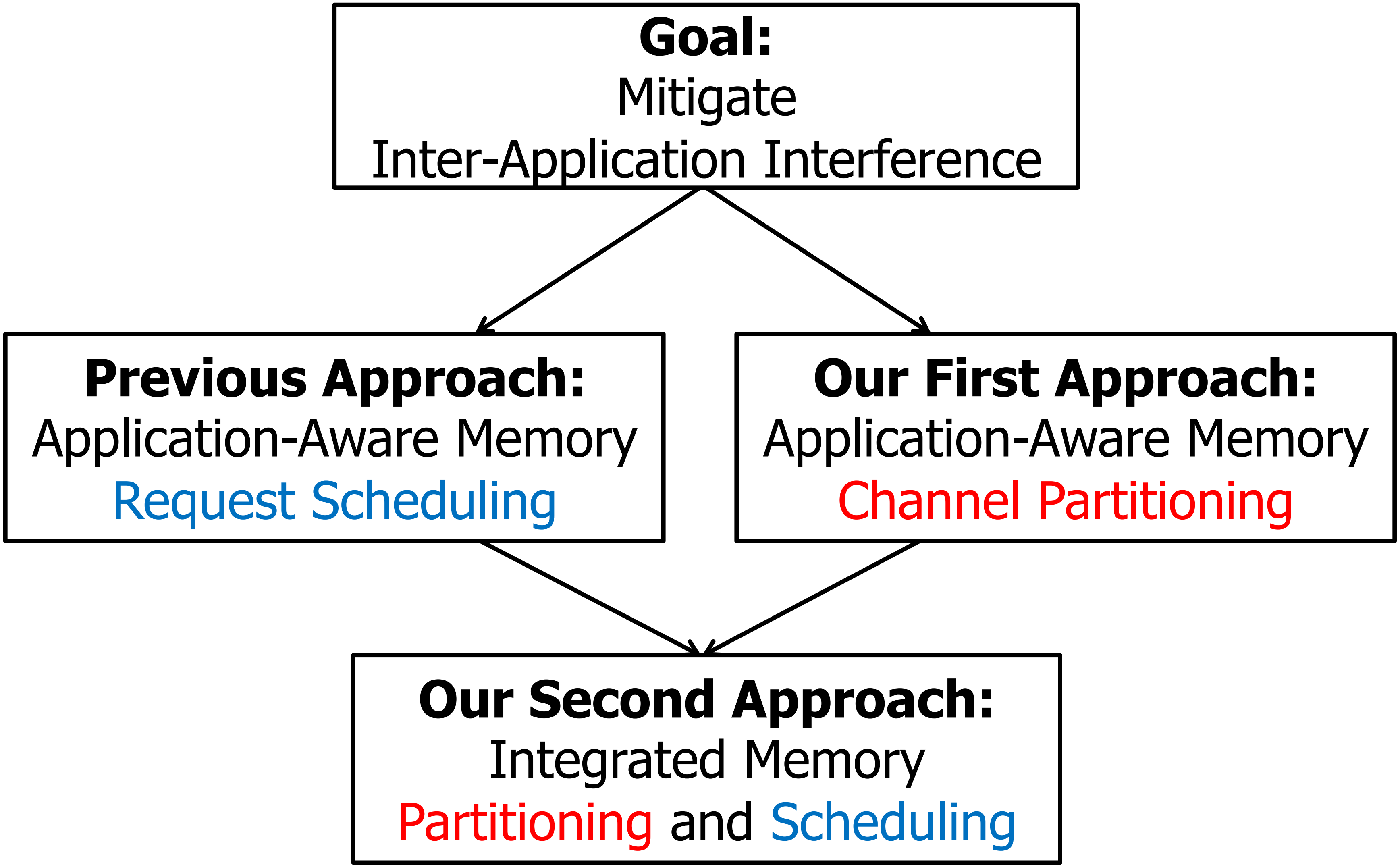
SAFARI

Memory Channel Partitioning

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

MCP Micro 2011 Talk

Outline



Goal:
Mitigate
Inter-Application Interference

Previous Approach:
Application-Aware Memory
Request Scheduling

Our First Approach:
Application-Aware Memory
Channel Partitioning

Our Second Approach:
Integrated Memory
Partitioning and Scheduling

Application-Aware Memory Request Scheduling

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

An Example: Thread Cluster Memory Scheduling

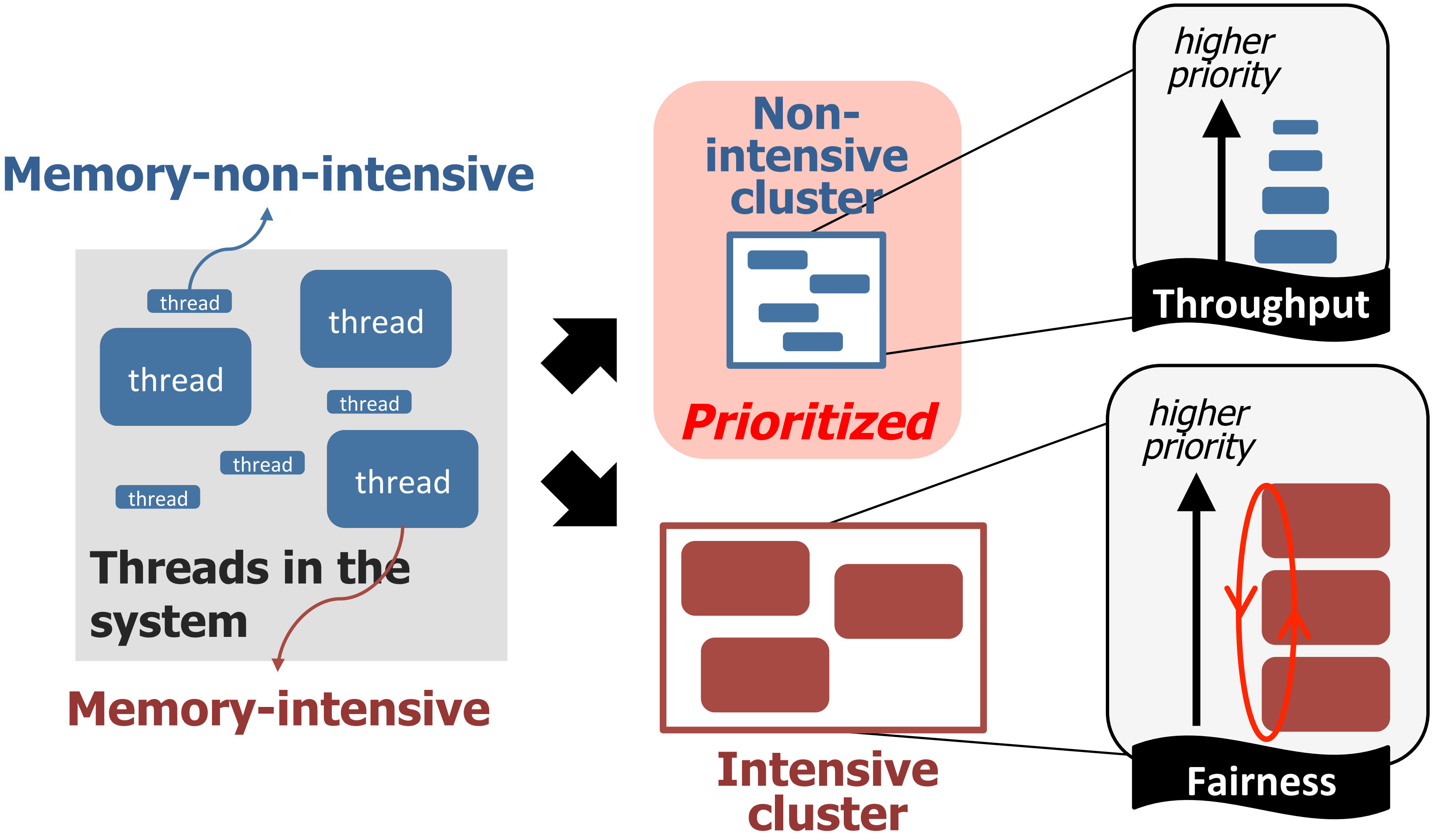


Figure: Kim et al., MICRO 2010

Application-Aware Memory Request Scheduling

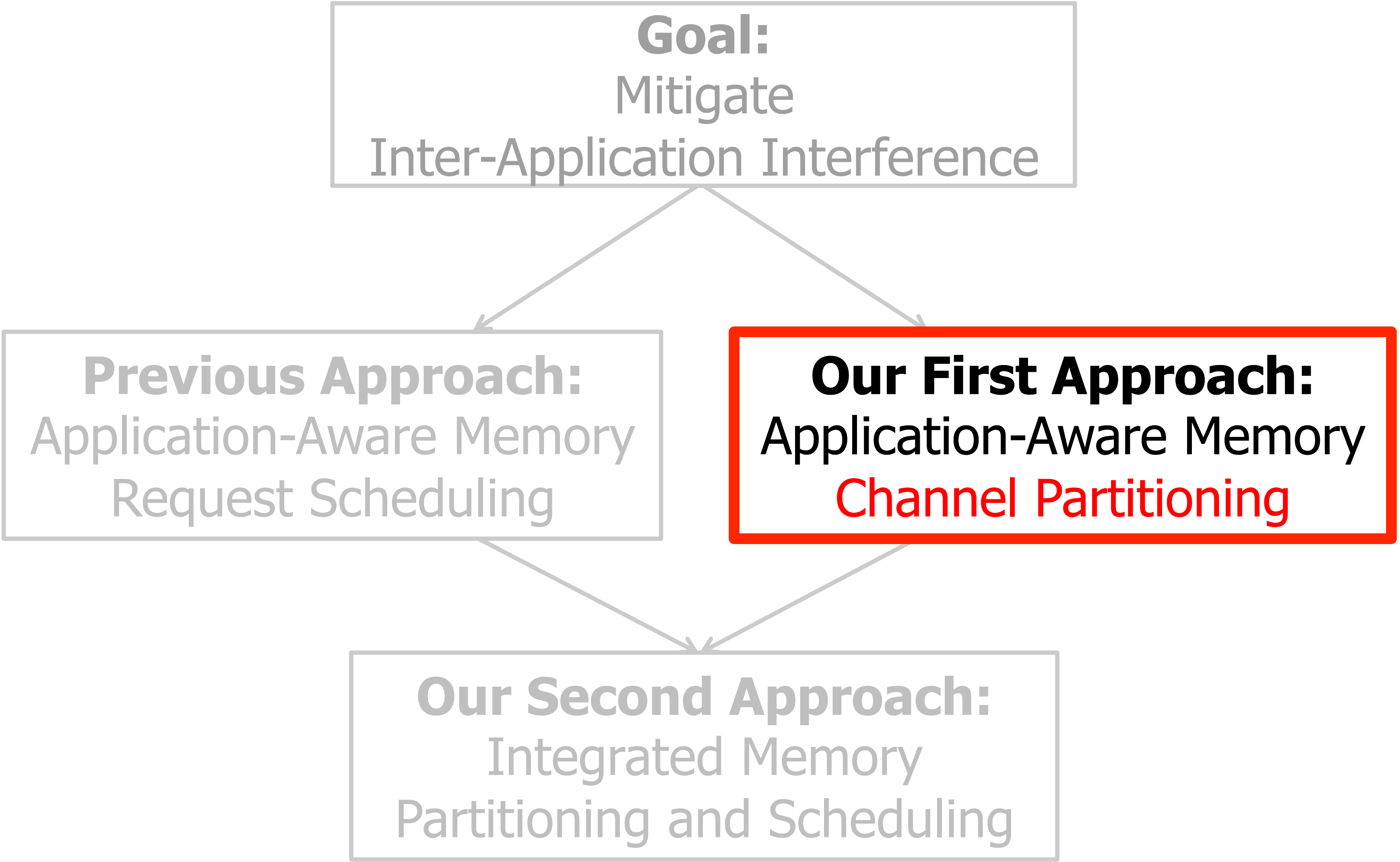
Advantages

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

Disadvantages

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

Our Approach



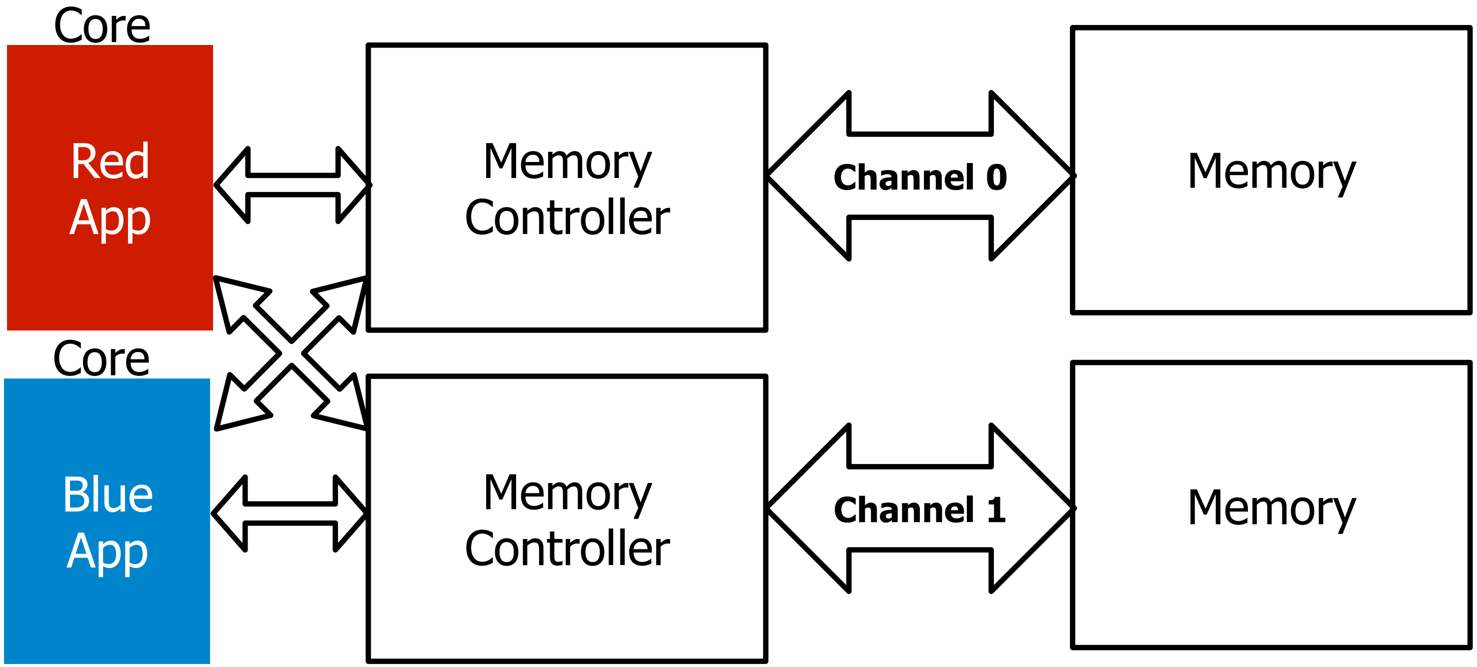
Goal:
Mitigate
Inter-Application Interference

Previous Approach:
Application-Aware Memory
Request Scheduling

Our First Approach:
Application-Aware Memory
Channel Partitioning

Our Second Approach:
Integrated Memory
Partitioning and Scheduling

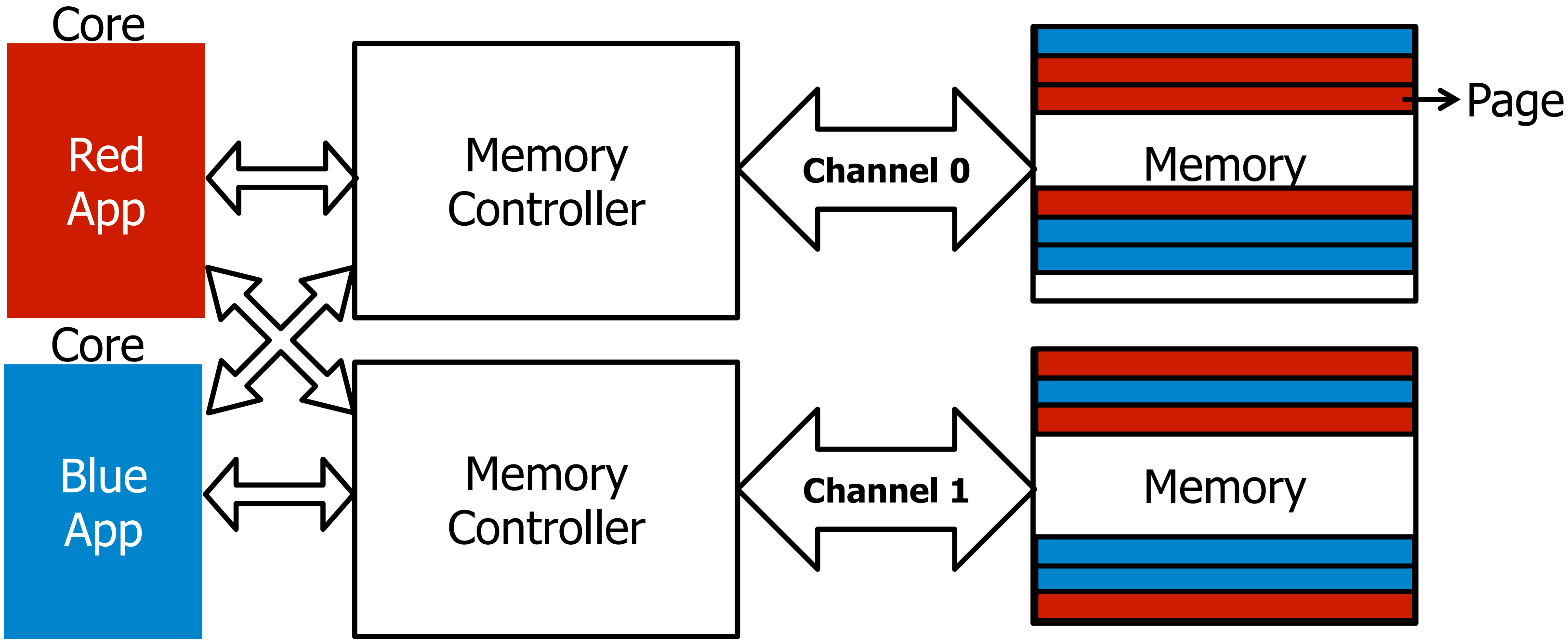
Observation: Modern Systems Have Multiple Channels



A new degree of freedom

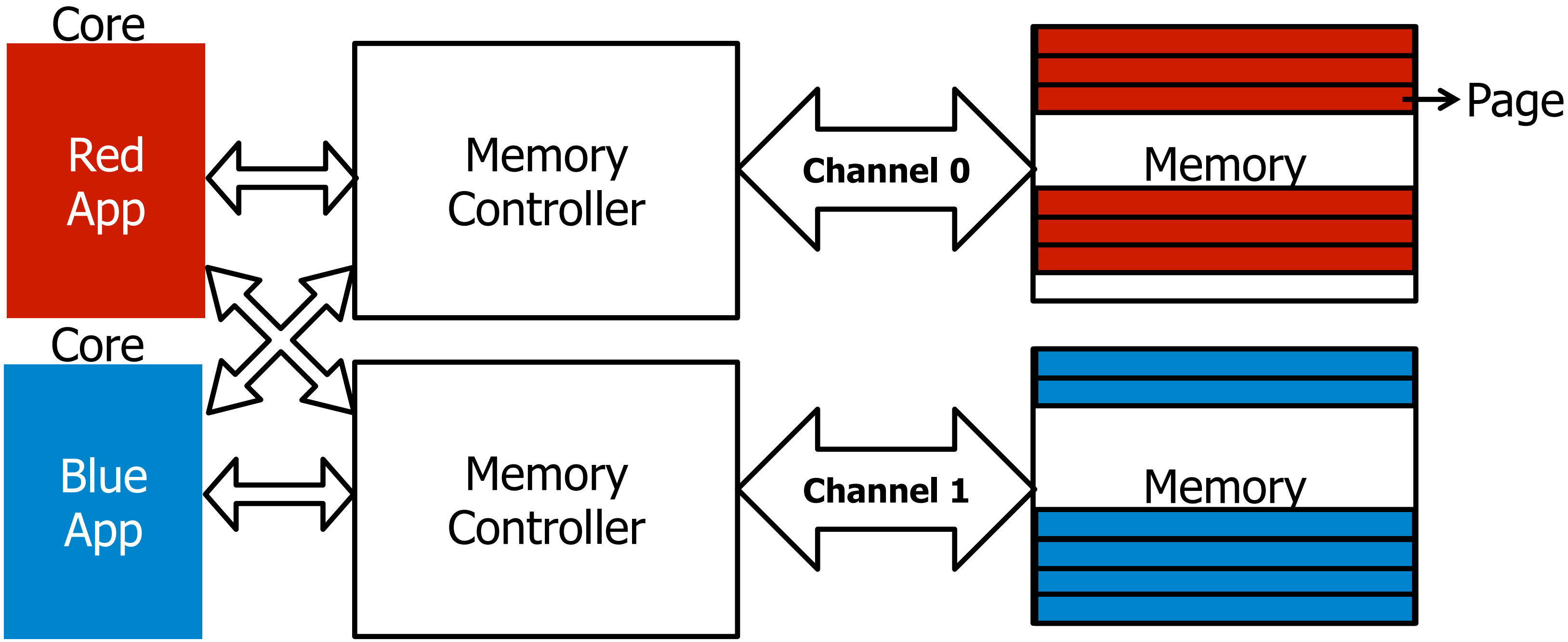
Mapping data across multiple channels

Data Mapping in Current Systems



Causes interference between applications' requests

Partitioning Channels Between Applications



Eliminates interference between applications' requests

Overview: Memory Channel Partitioning (MCP)

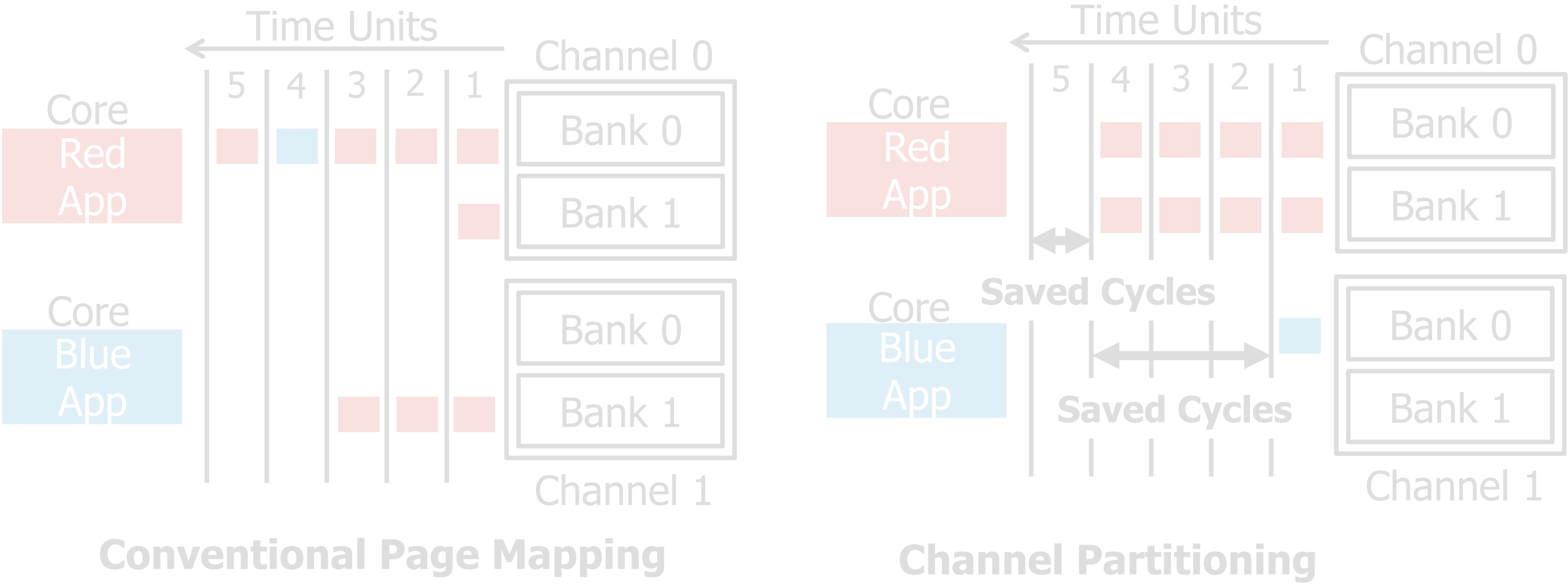
- Goal
  - Eliminate harmful interference between applications

- Basic Idea
  - Map the data of badly-interfering applications to different channels

- Key Principles
  - Separate low and high memory-intensity applications
  - Separate low and high row-buffer locality applications

Key Insight 1: Separate by Memory Intensity

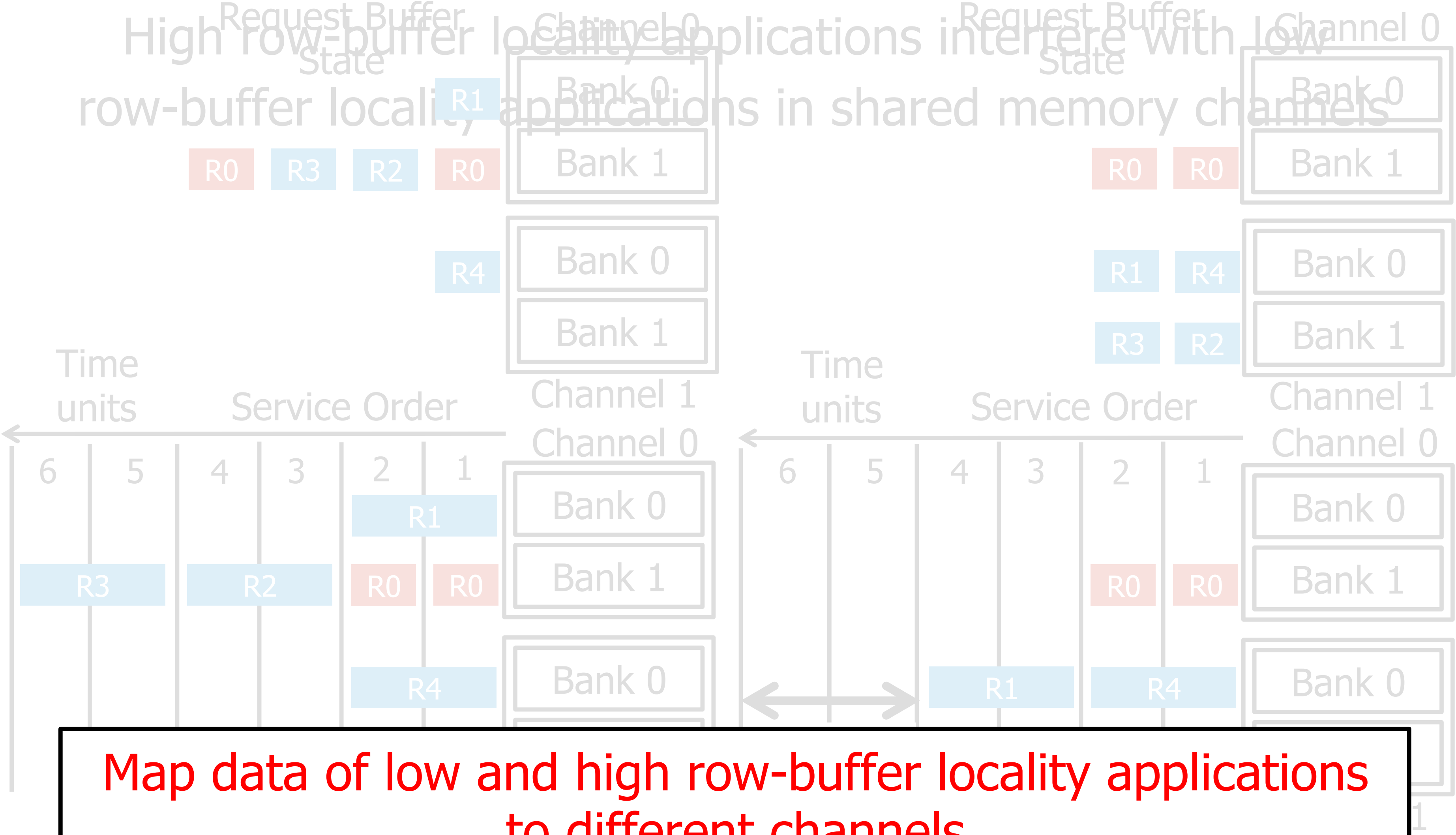
High memory-intensity applications interfere with low memory-intensity applications in shared memory channels



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

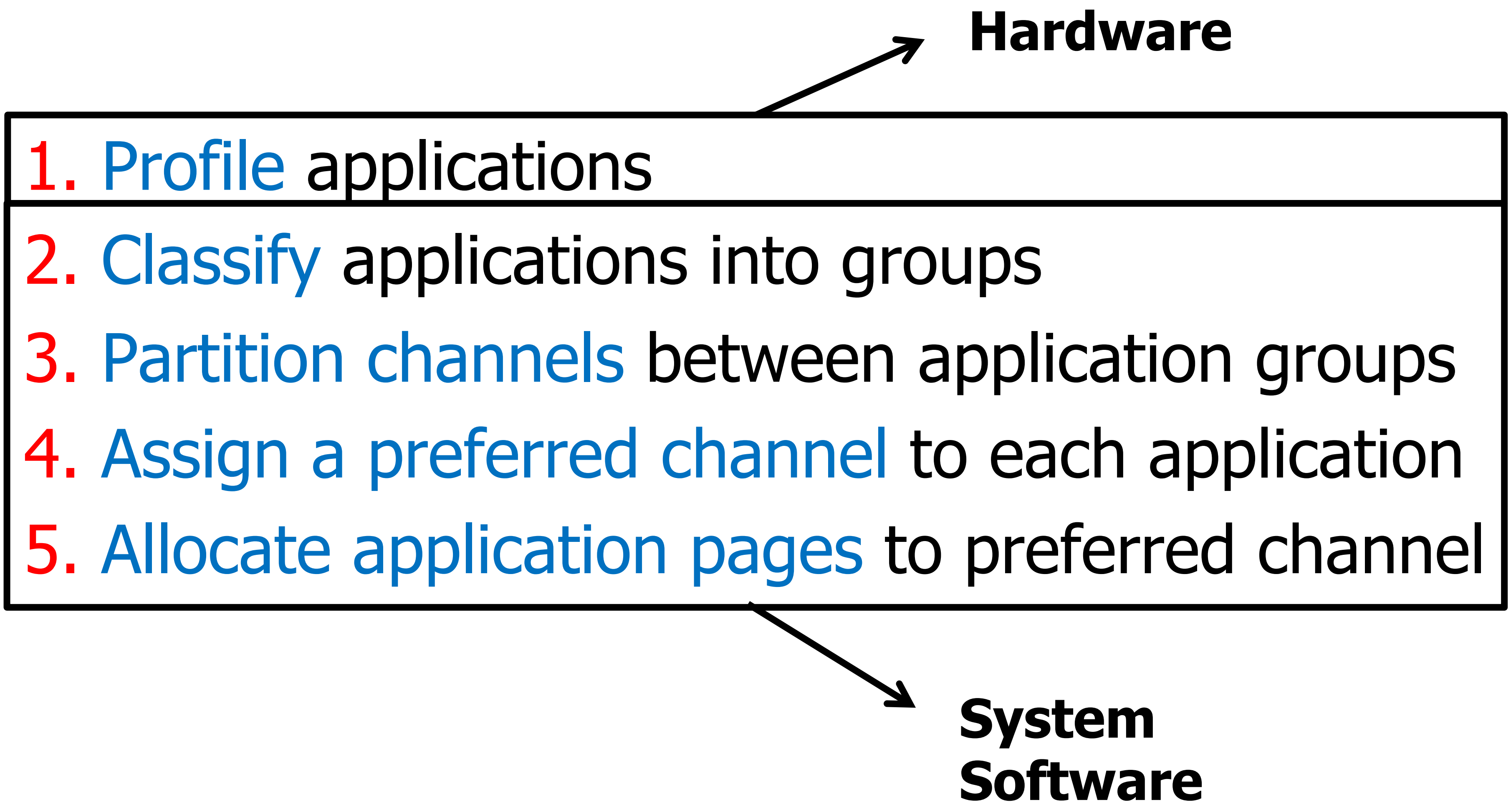
Key Insight 2: Separate by Row-Buffer Locality

High row-buffer locality applications interfere with low row-buffer locality applications in shared memory channels



Map data of low and high row-buffer locality applications to different channels

Memory Channel Partitioning (MCP) Mechanism



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

Hardware

System Software

1. Profile Applications

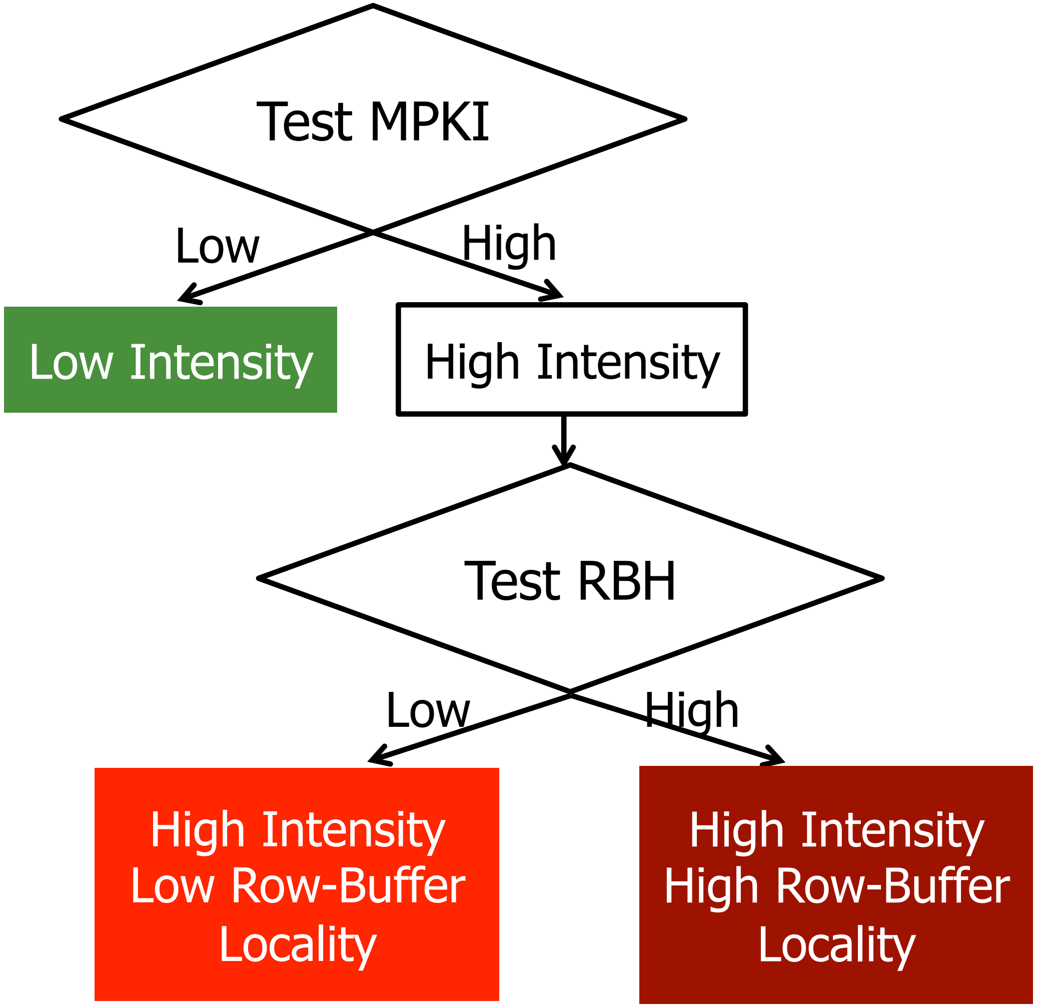
- Hardware counters collect application memory access characteristics

- Memory access characteristics
  - Memory intensity:

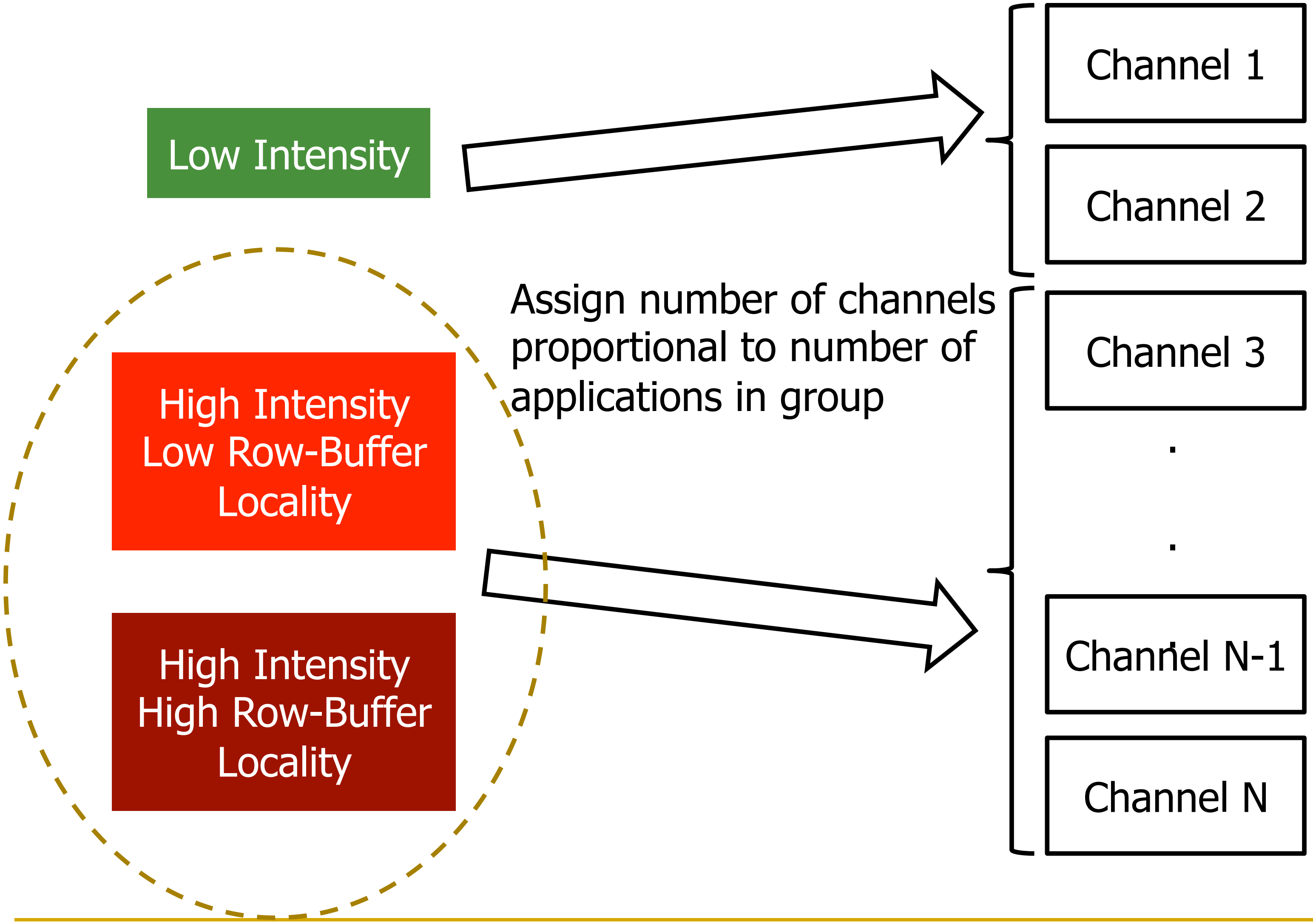
    Last level cache Misses Per Kilo Instruction (MPKI)
  - Row-buffer locality:

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

2. Classify Applications

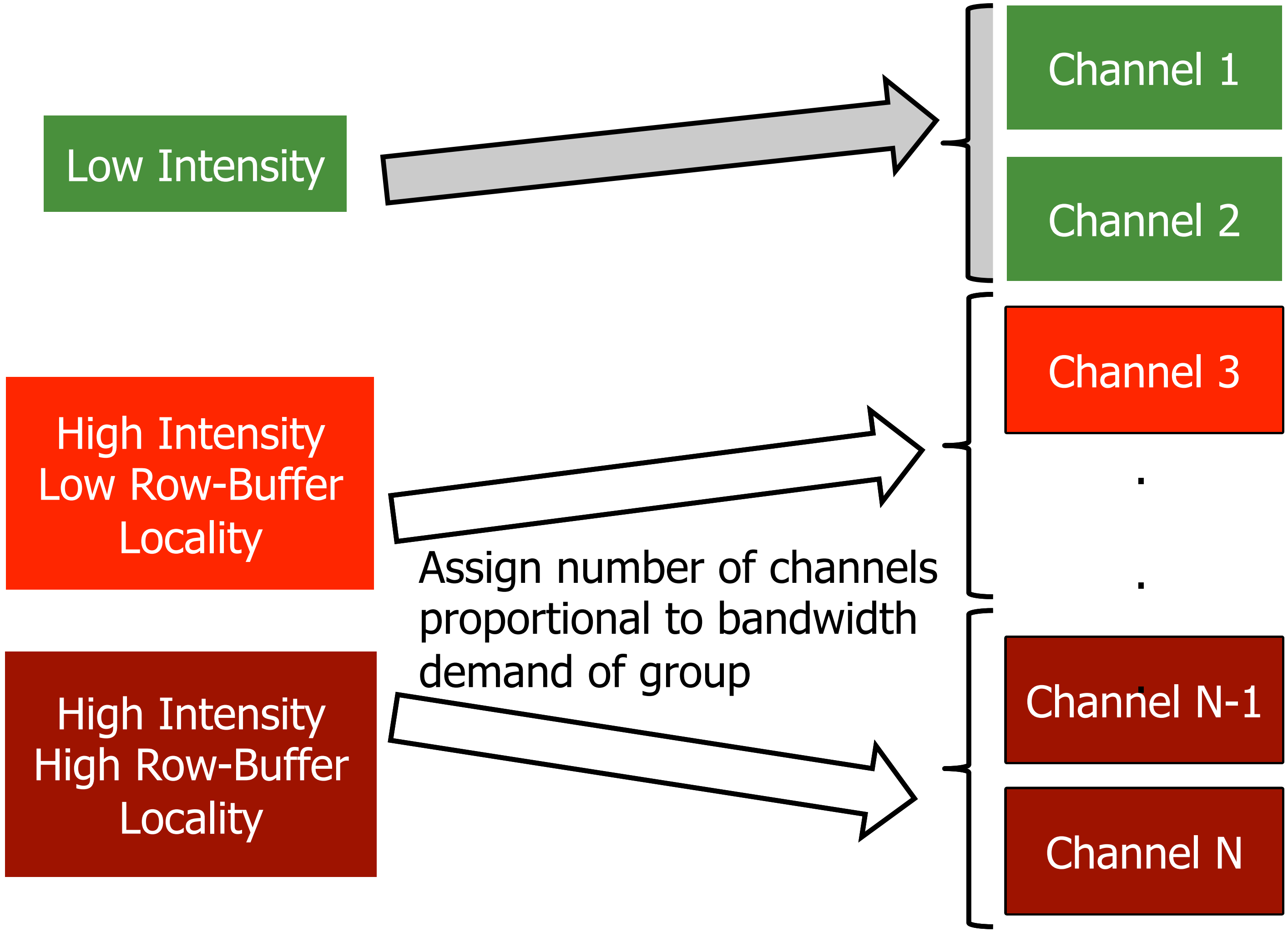

Test MPKI
Low
High
Low Intensity
High Intensity
Test RBH
Low
High
High Intensity Low Row-Buffer Locality
High Intensity High Row-Buffer Locality

3. Partition Channels Among Groups: Step 1


Low Intensity
High Intensity Low Row-Buffer Locality
High Intensity High Row-Buffer Locality
Channel 1
Channel 2
Channel 3
.
.
Channel N-1
Channel N

Assign number of channels proportional to number of applications in group

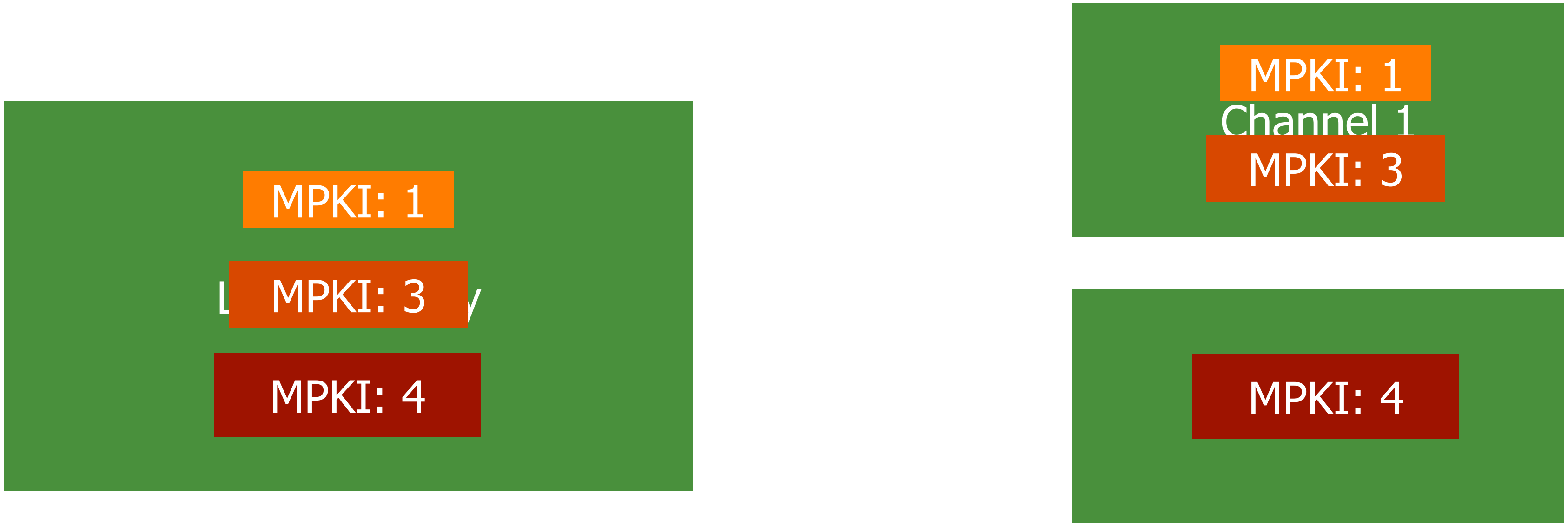
3. Partition Channels Among Groups: Step 2



Assign number of channels proportional to bandwidth demand of group

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

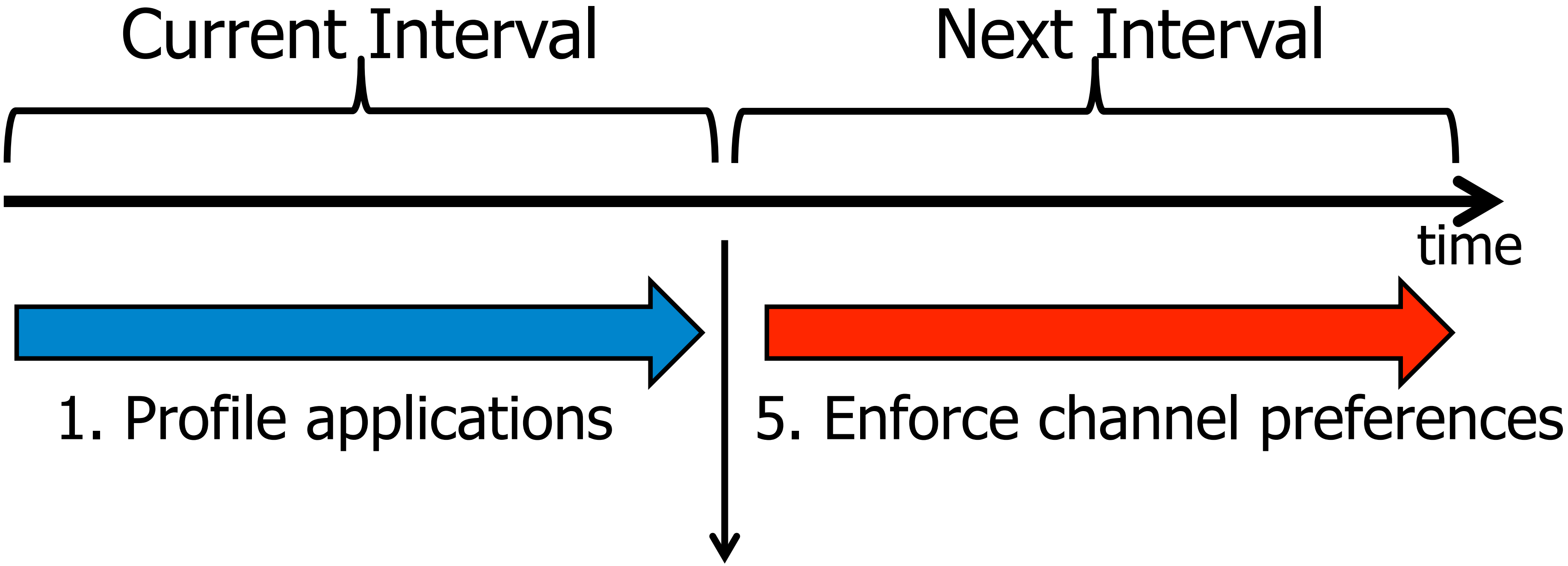


5. Allocate Page to Preferred Channel

- 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



Current Interval

Next Interval

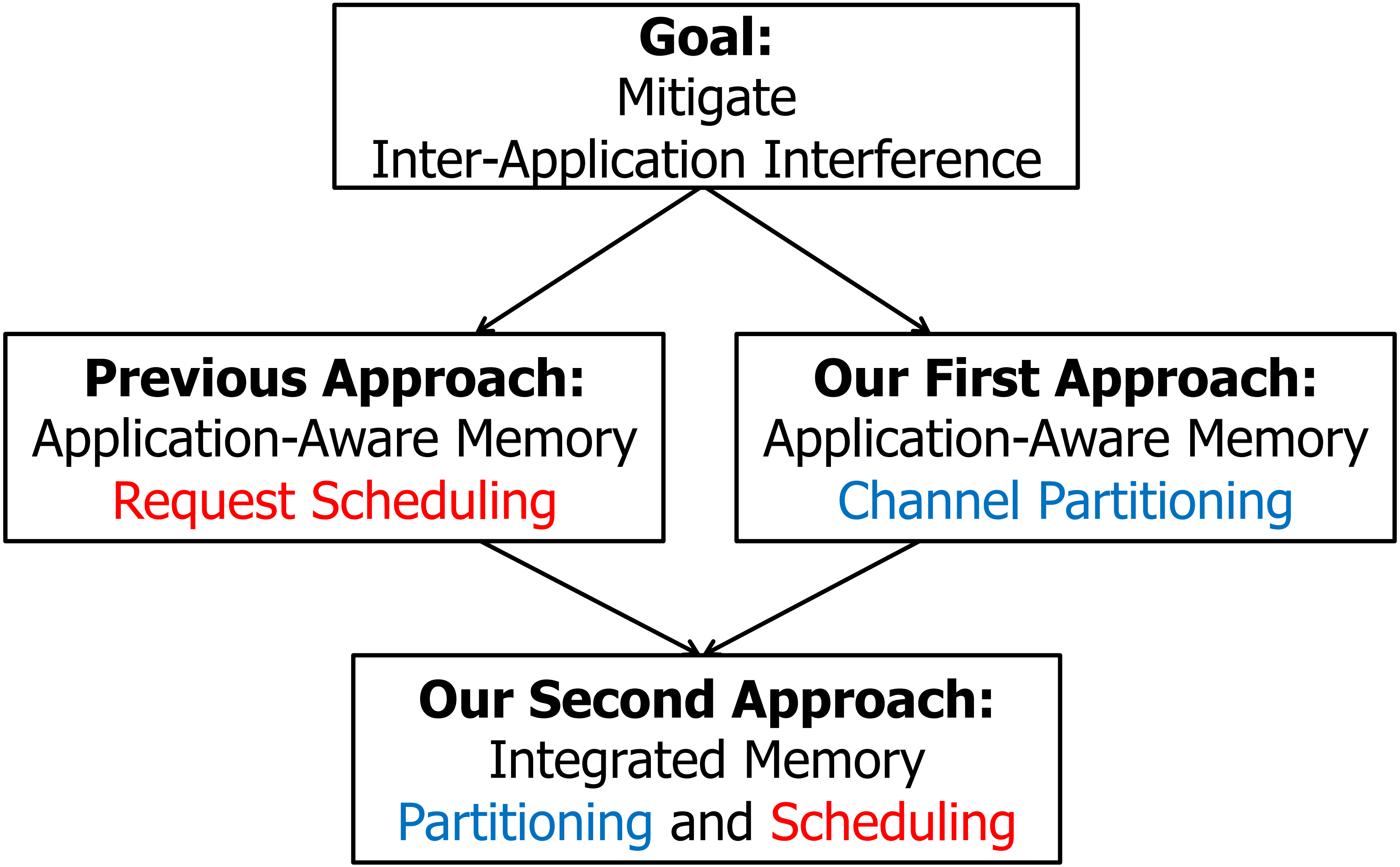
time

1. Profile applications

5. Enforce channel preferences

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

Integrating Partitioning and Scheduling



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

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

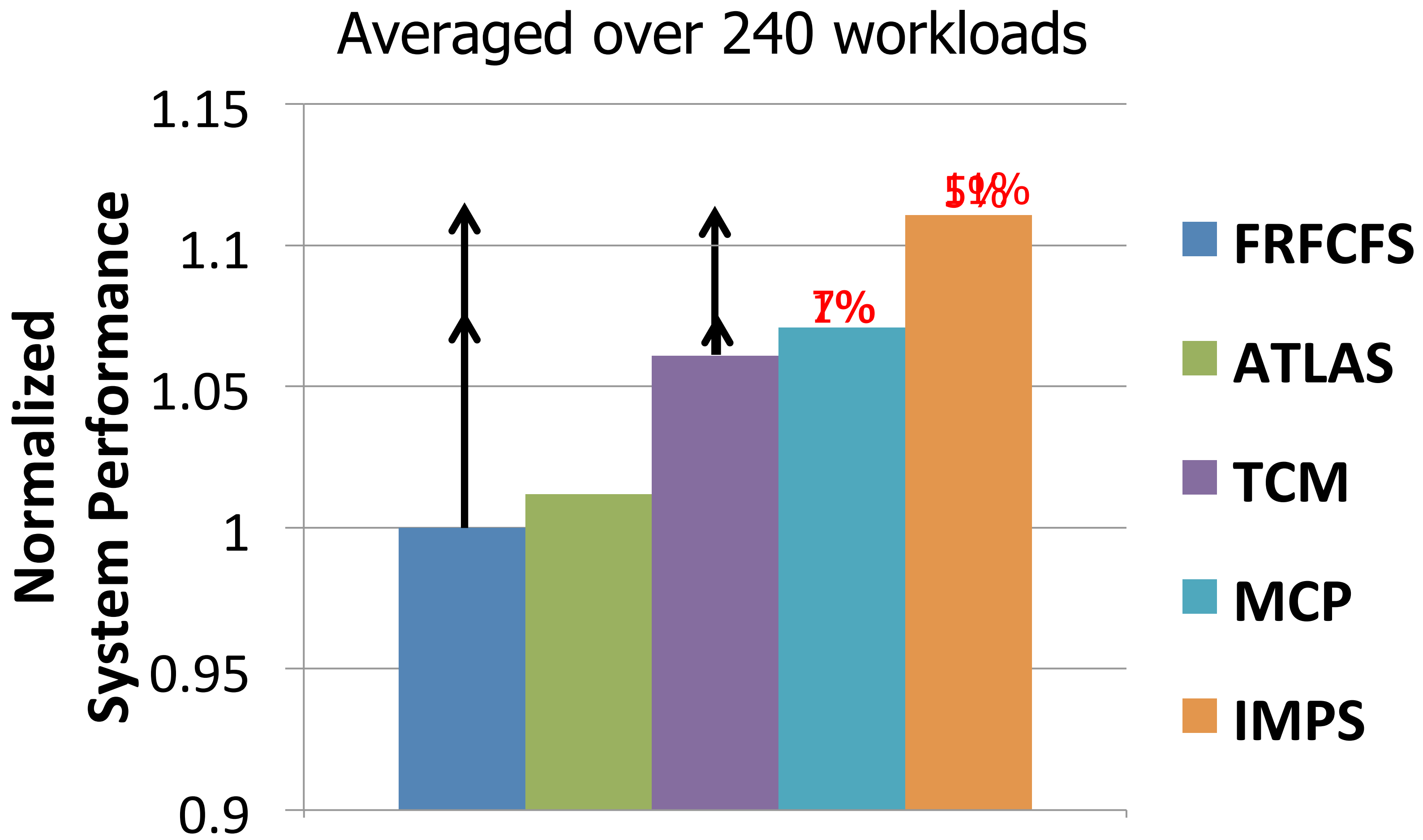
Previous Work on Memory Scheduling

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

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

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

Comparison to Previous Scheduling Policies

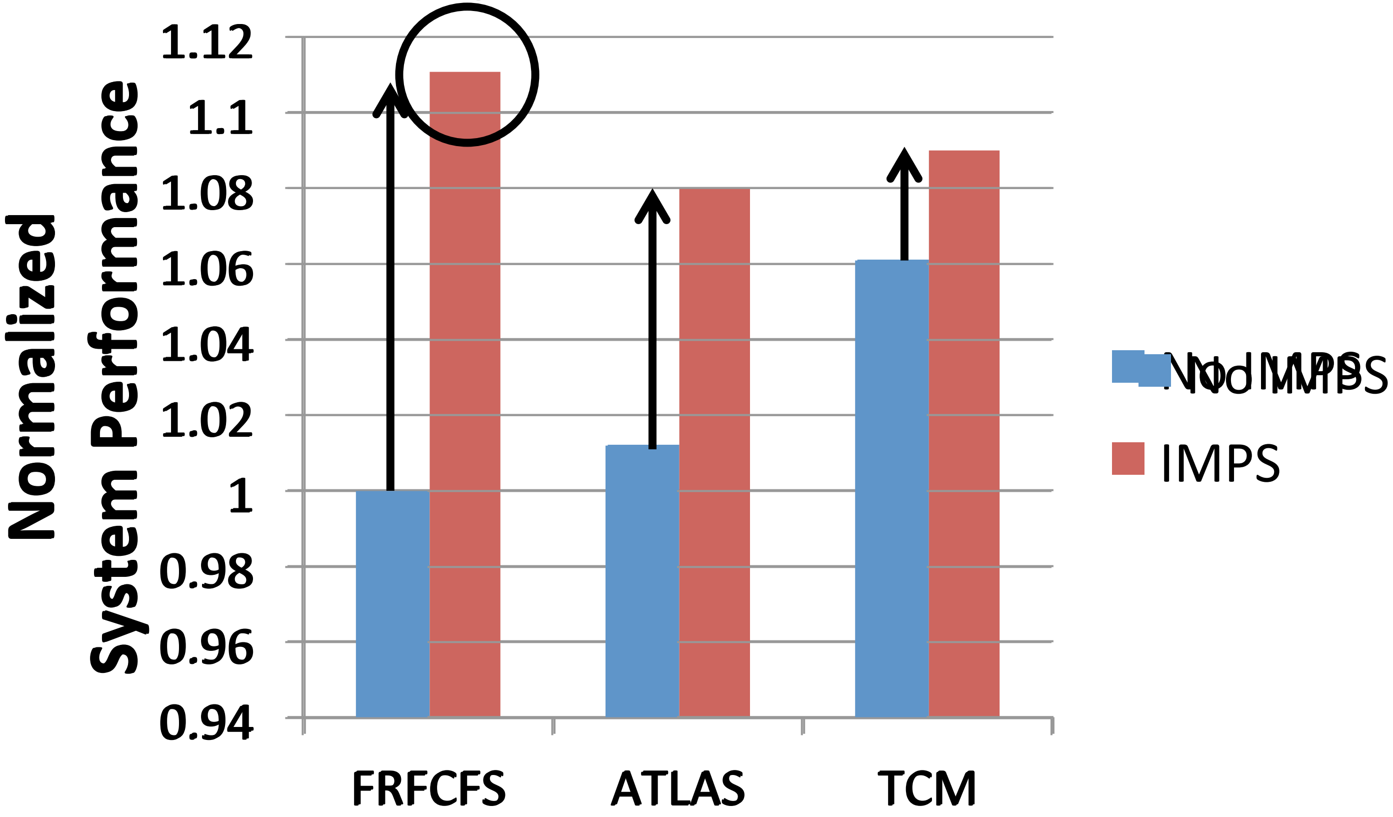


Better system performance than the best previous scheduler at lower hardware cost

Significant performance improvement over baseline FRFCFS

Interaction with Memory Scheduling

Averaged over 240 workloads



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]

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

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 [Das+ HPCA'13]

Conclusions: Topic 3

- 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

Scalable Many-Core Memory Systems Topic 3: Memory Interference and QoS-Aware Memory Systems

Prof. Onur Mutlu
http://www.ece.cmu.edu/~omutlu
onur@cmu.edu

HiPEAC ACACES Summer School 2013
July 15-19, 2013

Carnegie Mellon

SAFARI

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)