Computer Architecture:
Memory Interference and QoS (Part II)

Prof. Onur Mutlu
Carnegie Mellon University

Memory Interference and QoS Lectures

- These slides are from a lecture delivered at INRIA (July 8, 2013)

- Similar slides were used at the ACACES 2013 course
    - http://users.ece.cmu.edu/~omutlu/acaces2013-memory.html

QoS-Aware Memory Systems (Wrap Up)

Onur Mutlu
onur@cmu.edu

July 9, 2013
INRIA

Carnegie Mellon

SAFARI

Slides for These Lectures

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

Videos for Similar Lectures

- Basics (of Computer Architecture)
    - http://www.youtube.com/playlist?list=PL5PHm2jkkXmidJOd59REog9jDnPDTG6IJ

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

Designing QoS-Aware Memory Systems: Approaches

- Smart resources: Design each shared resource to have a configurable interference control/reduction mechanism
  - QoS-aware memory controllers [Mutlu+ MICRO'07] [Moscibroda+, Usenix Security'07] [Mutlu+ ISCA'08, Top Picks'09] [Kim+ HPCA'10] [Kim+ MICRO'10, Top Picks'11] [Ebrahimi+ ISCA'11, MICRO'11] [Ausavarungnirun+, ISCA'12][Subramanian+, HPCA'13]
  - QoS-aware interconnects [Das+ MICRO'09, ISCA'10, Top Picks '11] [Grot+ MICRO'09, ISCA'11, Top Picks '12]
  - QoS-aware caches

- Dumb resources: Keep each resource free-for-all, but reduce/control interference by injection control or data mapping
  - Source throttling to control access to memory system [Ebrahimi+ ASPLOS'10, ISCA'11, TOCS'12] [Ebrahimi+ MICRO'09] [Nychis+ HotNets'10] [Nychis+ SIGCOMM'12]
  - QoS-aware data mapping to memory controllers [Muralidhara+ MICRO'11]
  - QoS-aware thread scheduling to cores [Das+ HPCA'13]

ATLAS Pros and Cons

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

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

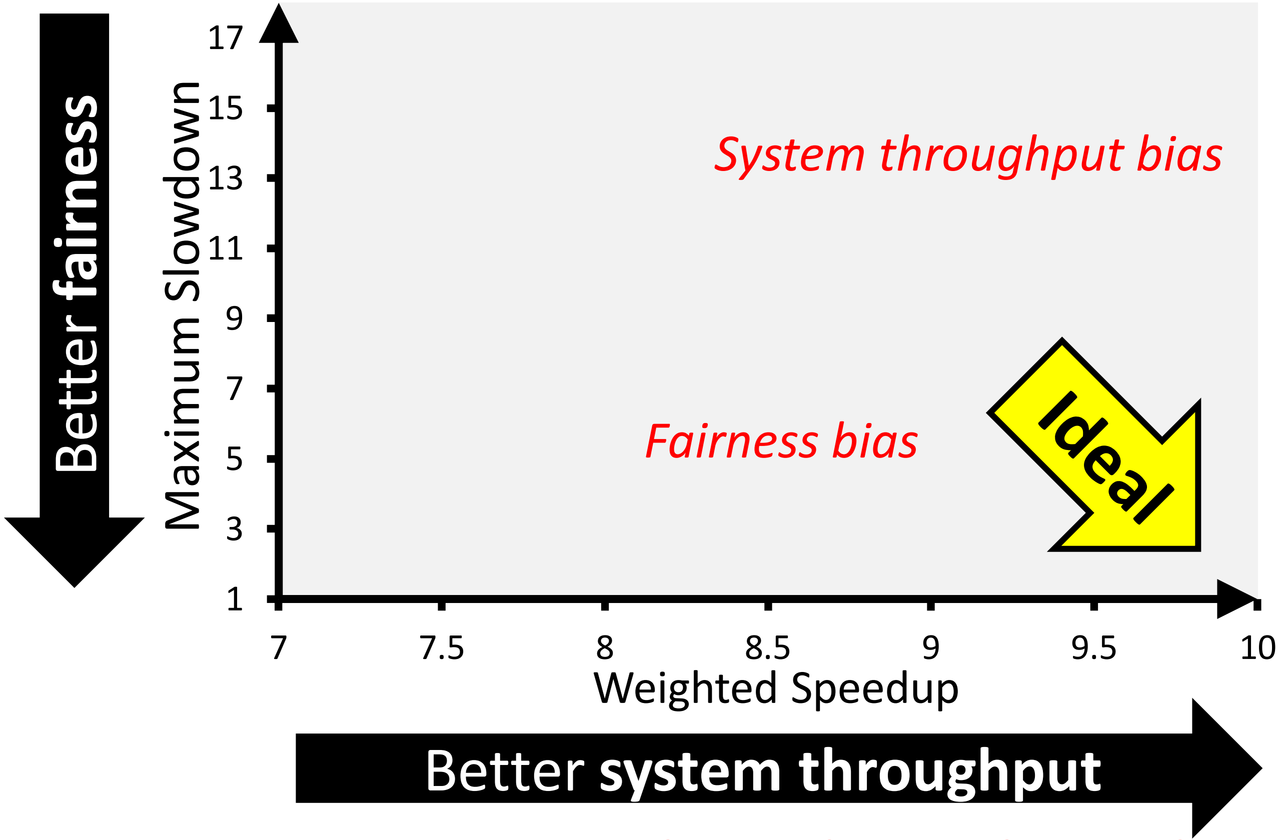
TCM:
Thread Cluster Memory Scheduling

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

TCM Micro 2010 Talk

Previous Scheduling Algorithms are Biased

24 cores, 4 memory controllers, 96 workloads



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

Throughput vs. Fairness

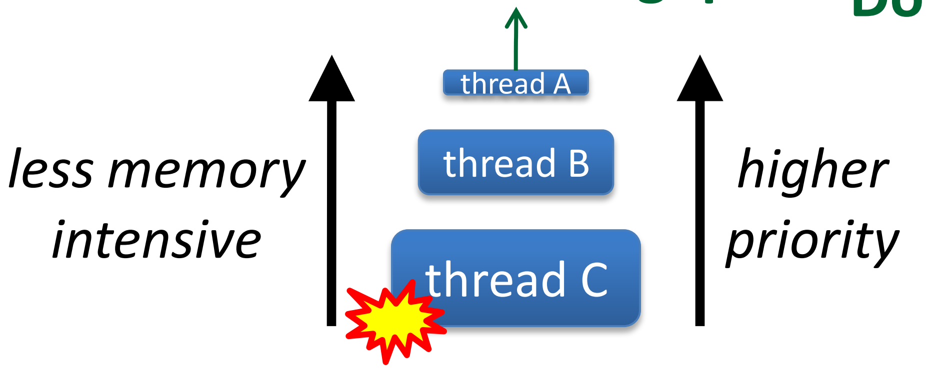
***Throughput biased** approach*

Prioritize less memory-intensive threads

***Fairness biased** approach*

Take turns accessing memory

Good for throughput

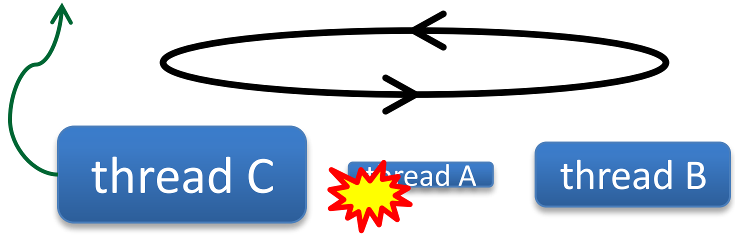


less memory intensive

higher priority

starvation → unfairness

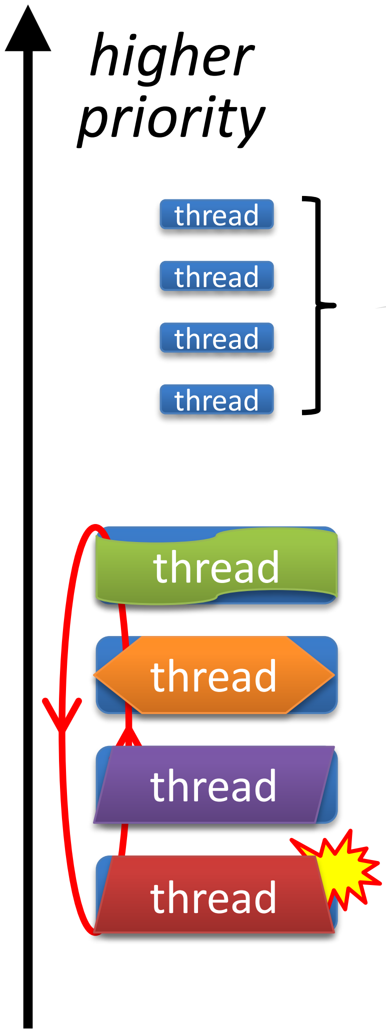
Does not starve



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

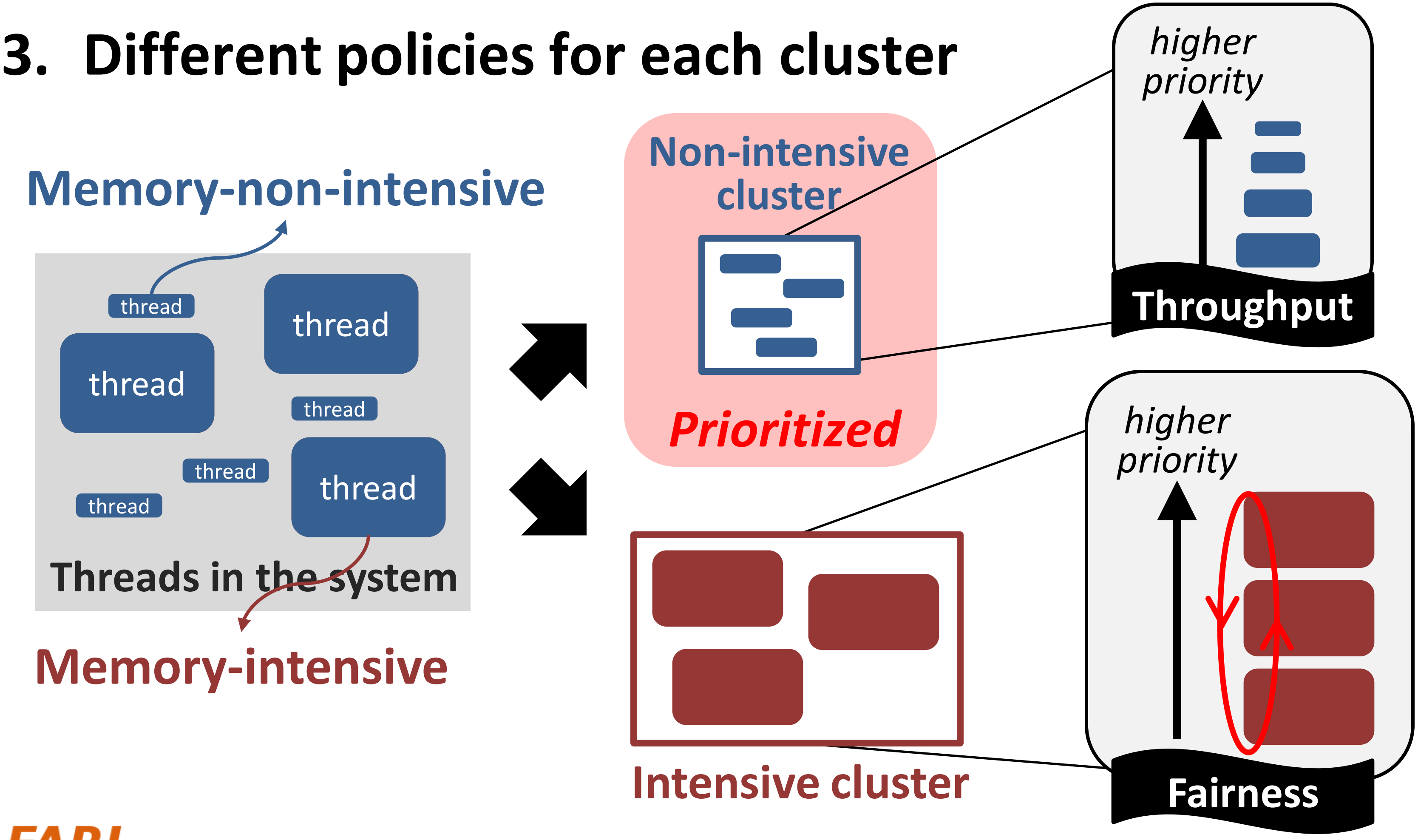
- Shuffle thread ranking

Memory-intensive threads have different vulnerability to interference

- Shuffle asymmetrically

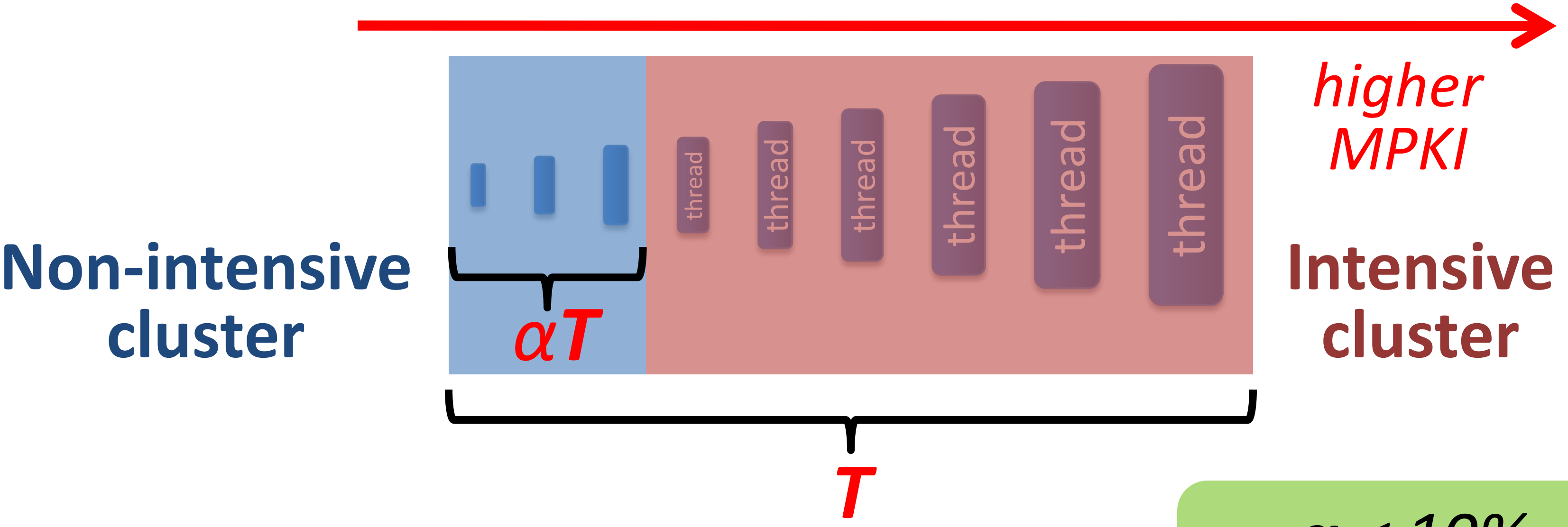
Thread Cluster Memory Scheduling [Kim+ MICRO'10]

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

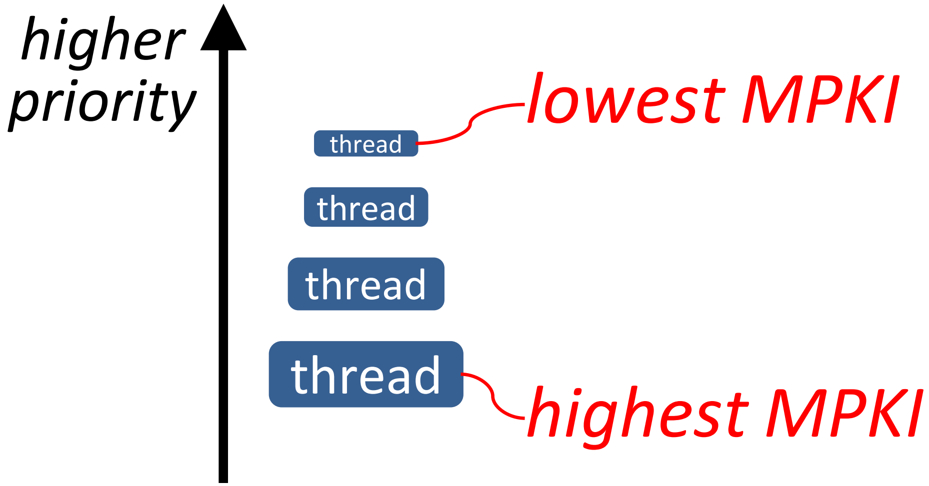
Prioritization Between Clusters

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

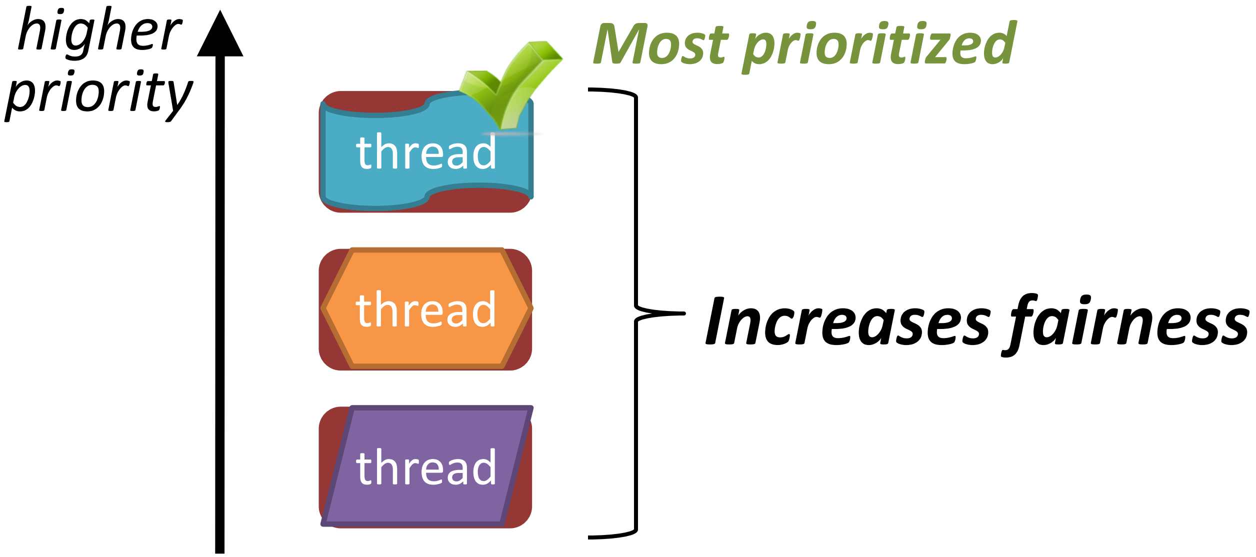
Prioritize threads according to MPKI



- **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*

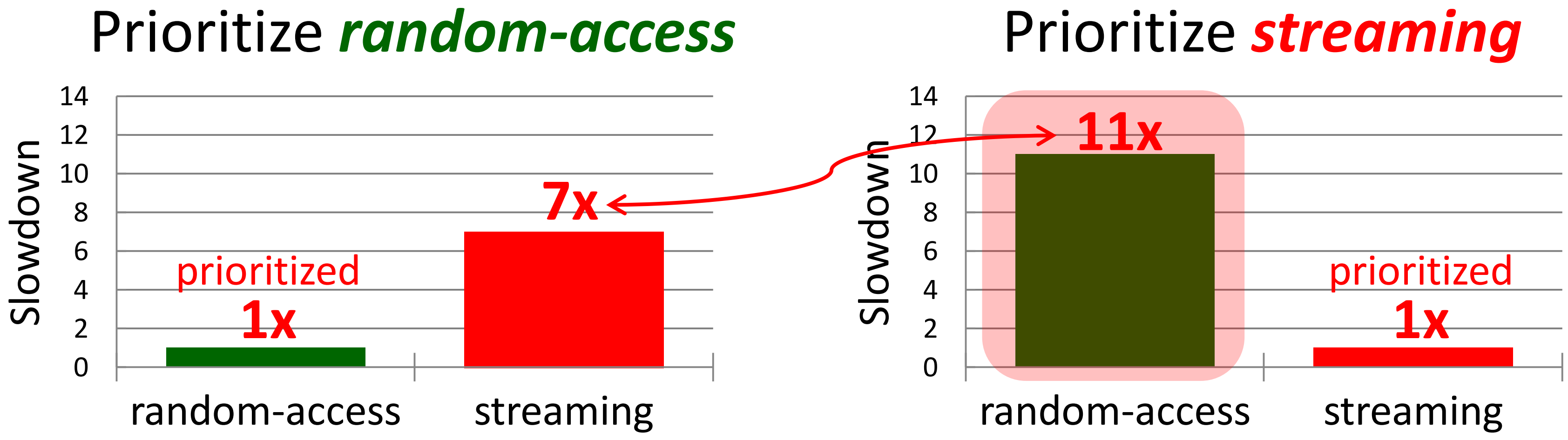
Case Study: A Tale of Two Threads

Case Study: Two intensive threads contending

1. random-access

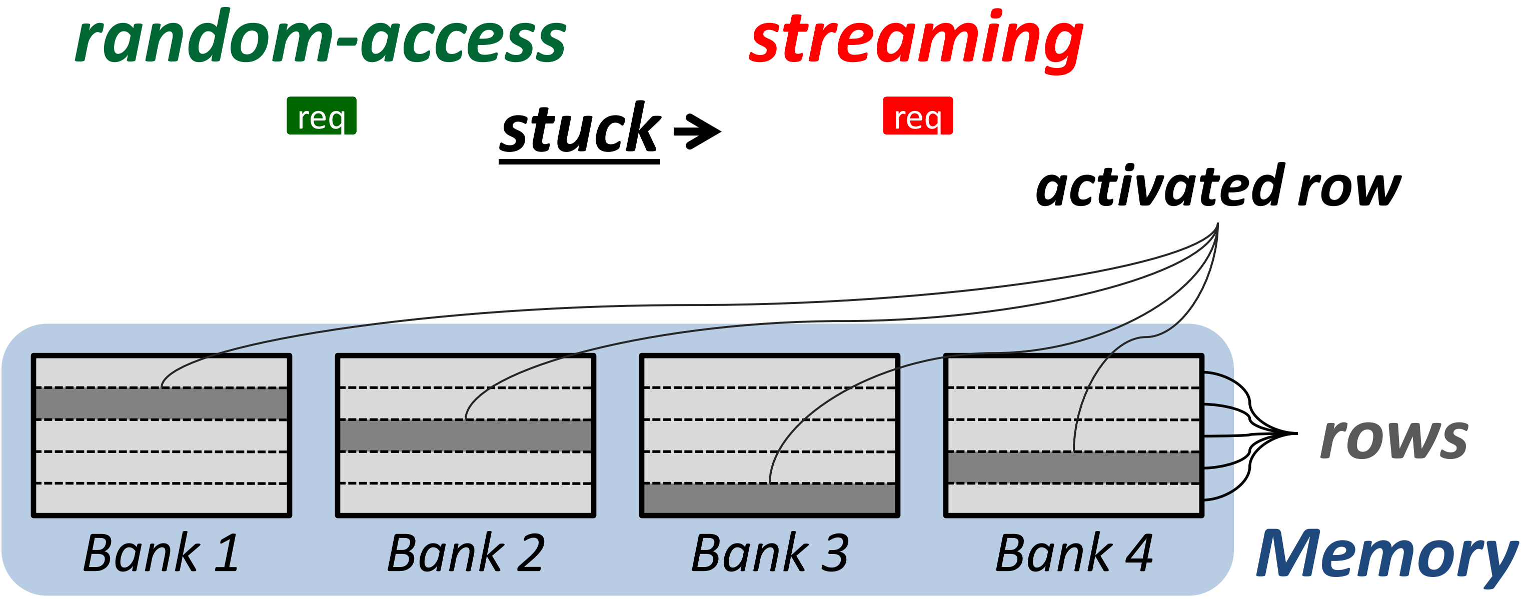
2. streaming

Which is slowed down more easily?



random-access thread is more easily slowed down

Why are Threads Different?



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

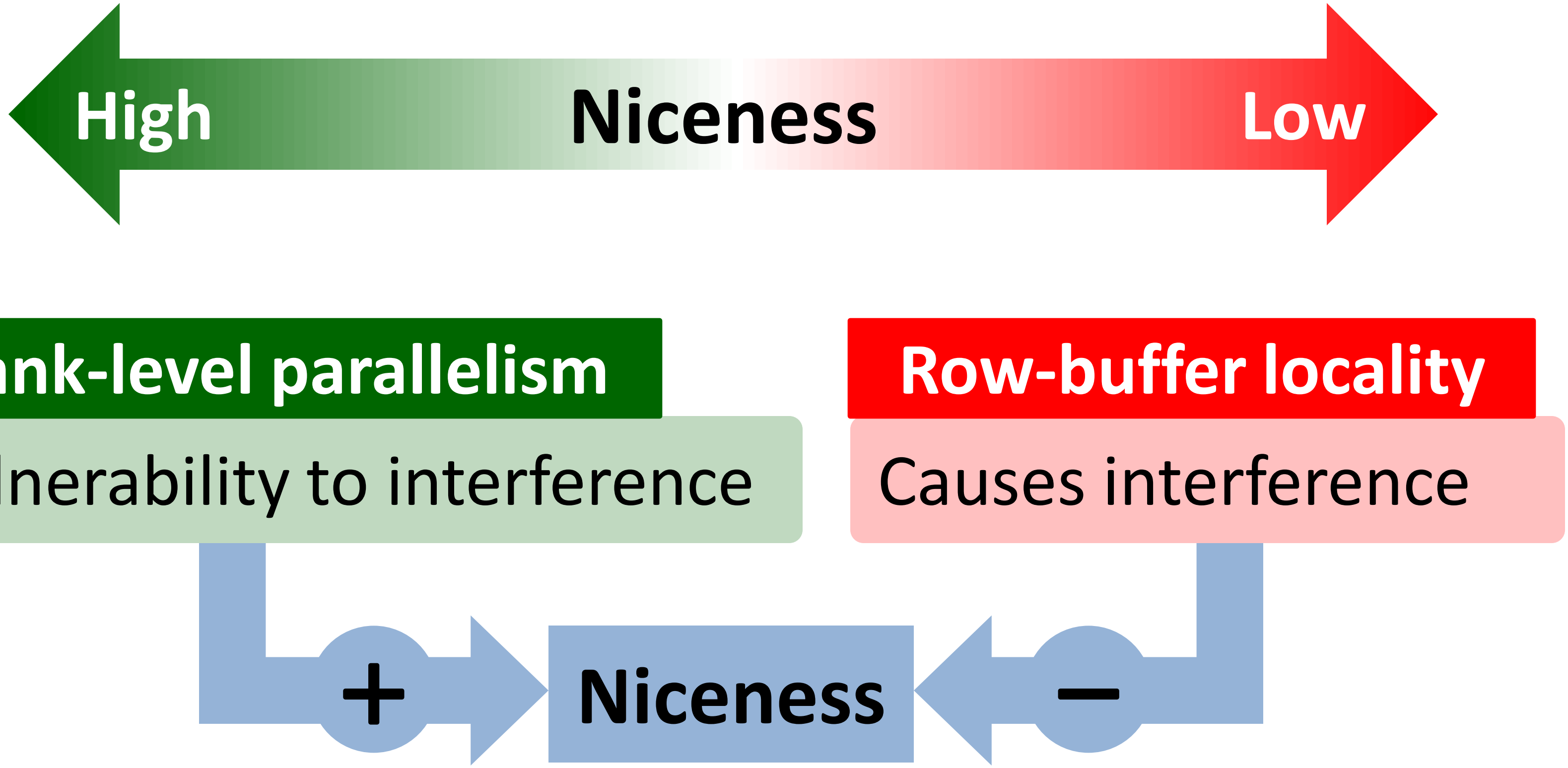
- All requests → Same row
- High **row-buffer locality**

Vulnerable to interference

SAFARI

Niceness

How to quantify difference between threads?



High **Niceness** **Low**

Bank-level parallelism

Vulnerability to interference

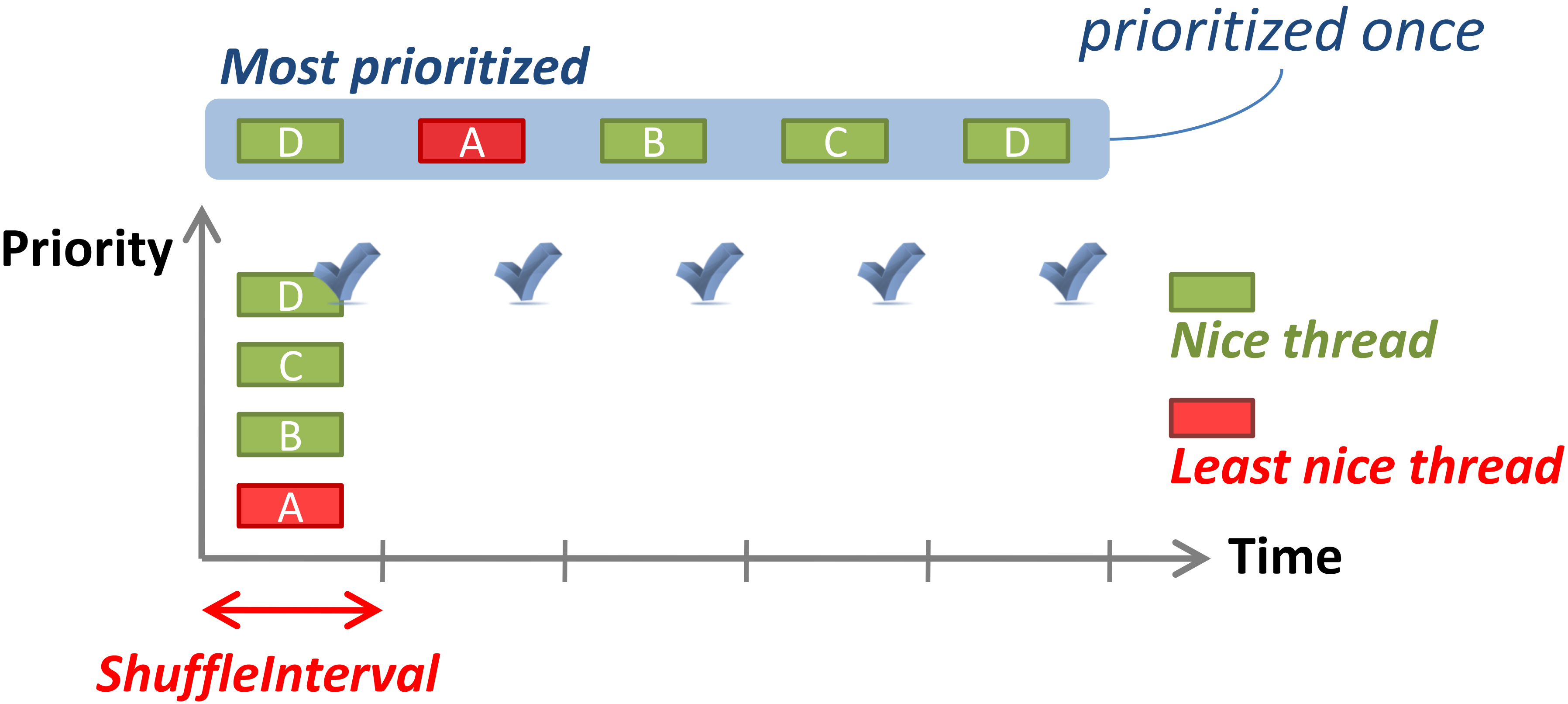
Row-buffer locality

Causes interference

+ **Niceness** **−**

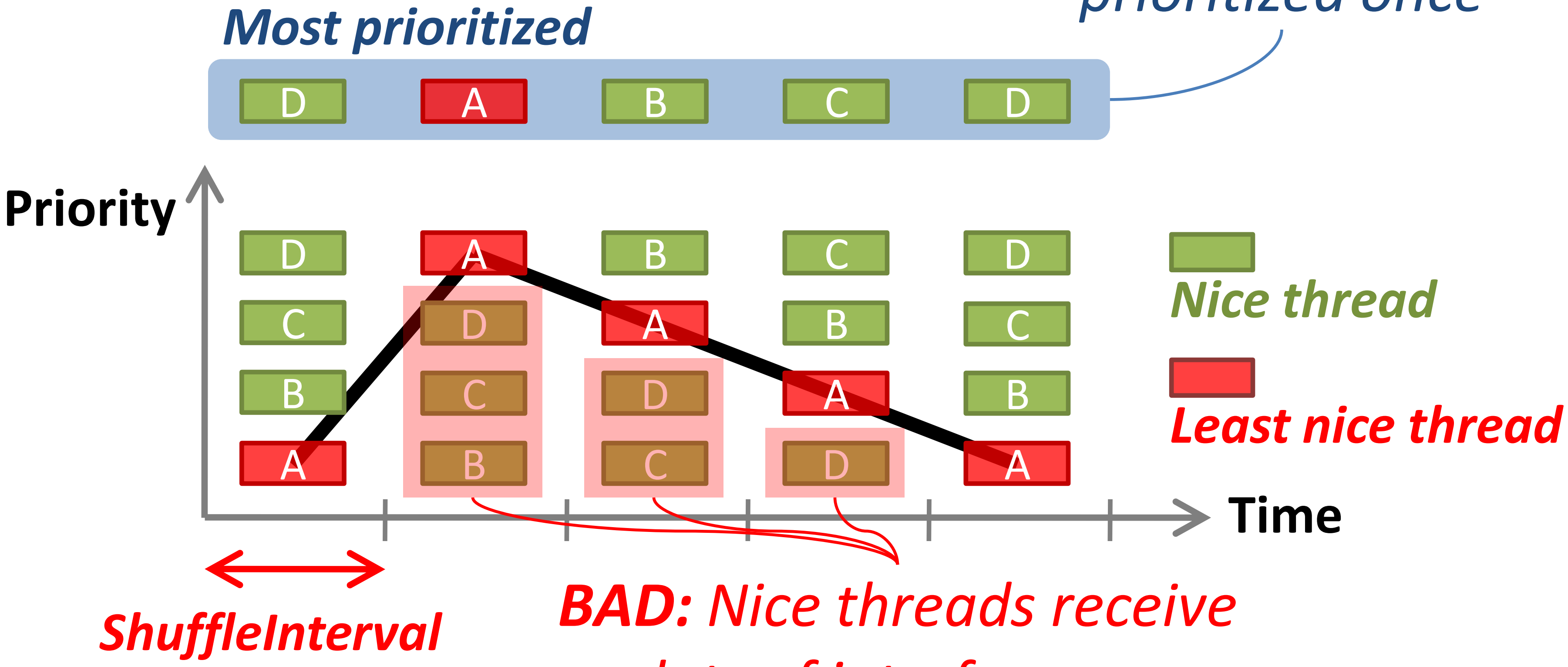
Shuffling: Round-Robin vs. Niceness-Aware

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*

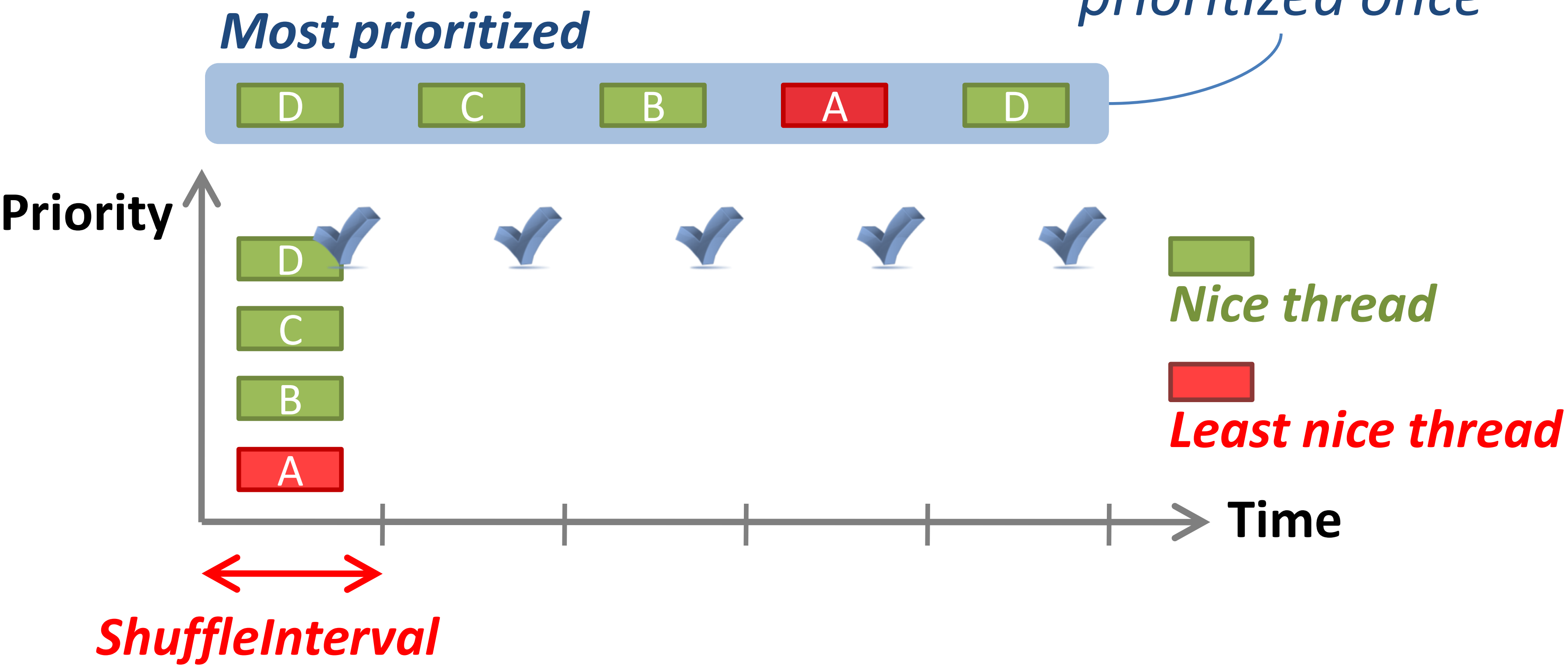


***GOOD:** Each thread prioritized once*

***BAD:** Nice threads receive lots of interference*

Shuffling: Round-Robin vs. Niceness-Aware

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

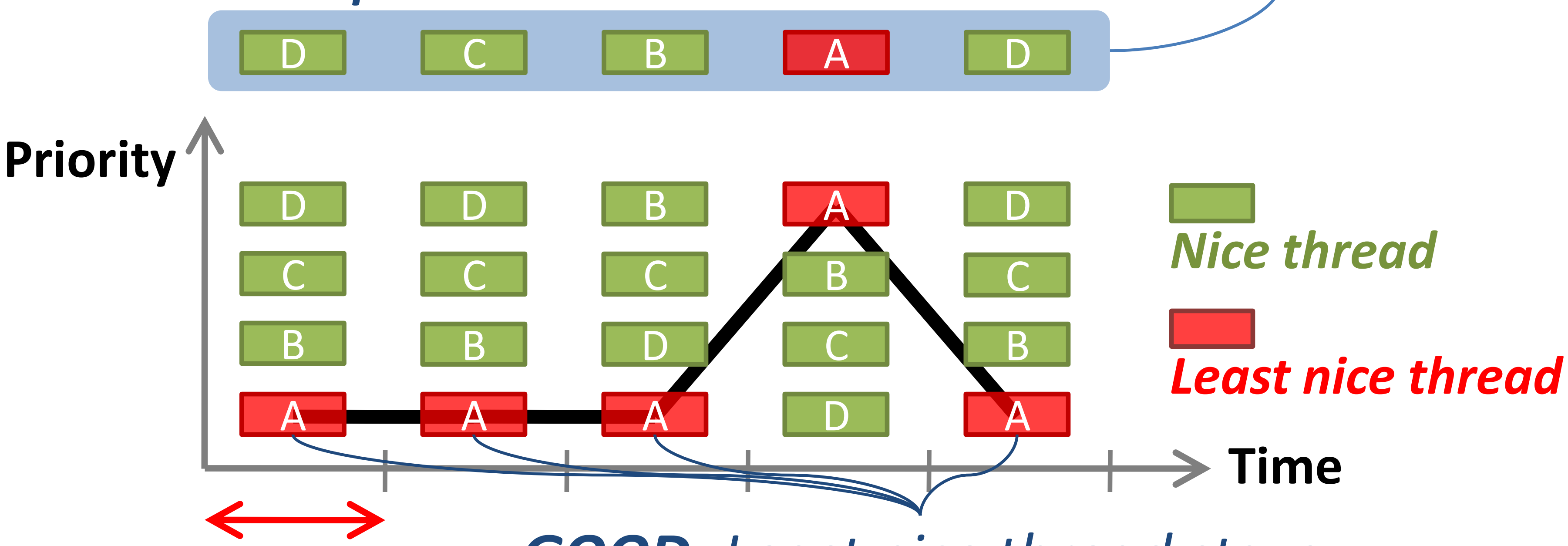


Shuffling: Round-Robin vs. Niceness-Aware

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

GOOD: Each thread prioritized once

Most prioritized



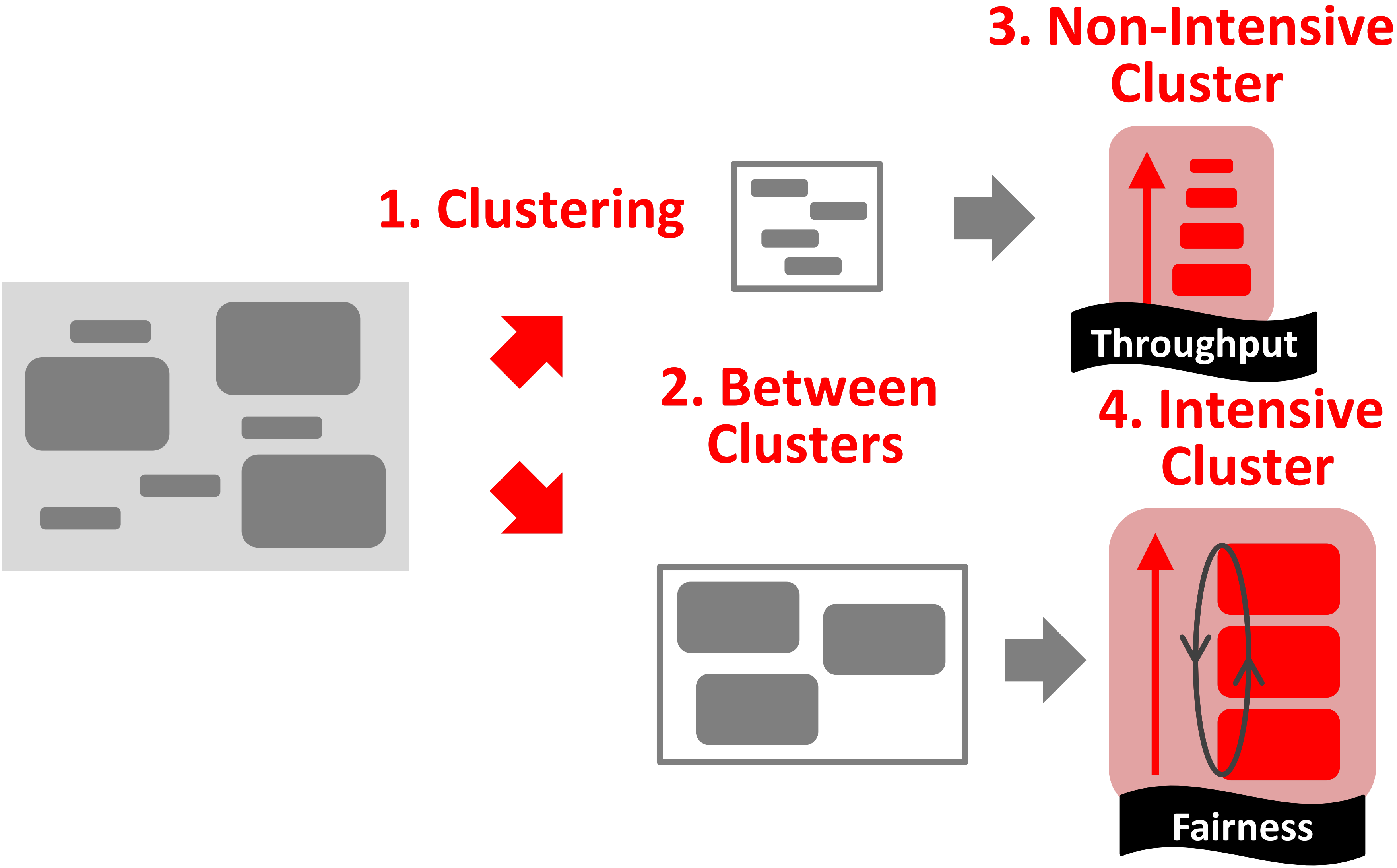
🟩 *Nice thread*

🟥 *Least nice thread*

ShuffleInterval

GOOD: Least nice thread stays mostly deprioritized

TCM Outline



1. Clustering

2. Between Clusters

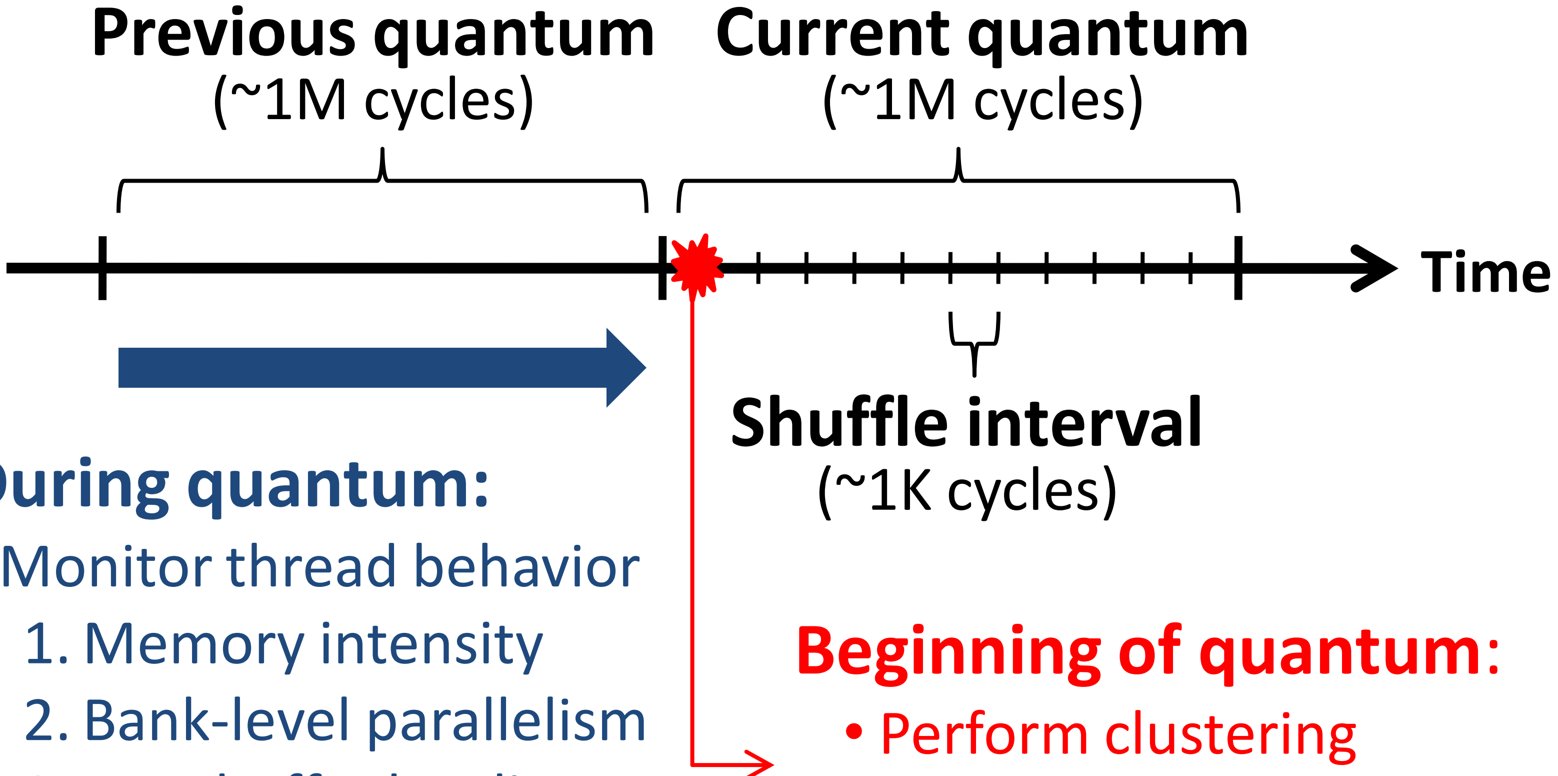
3. Non-Intensive Cluster

Throughput

4. Intensive Cluster

Fairness

TCM: Quantum-Based Operation



During quantum:

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

Beginning of quantum:

- Perform clustering
- Compute niceness of intensive threads

TCM: Scheduling Algorithm

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

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

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

TCM: Implementation Cost

Required storage at memory controller *(24 cores)*

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

- No computation is on the critical path

Previous Work

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

- Thread-oblivious ➔ Low throughput & Low fairness

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

- Non-intensive threads not prioritized ➔ Low throughput

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

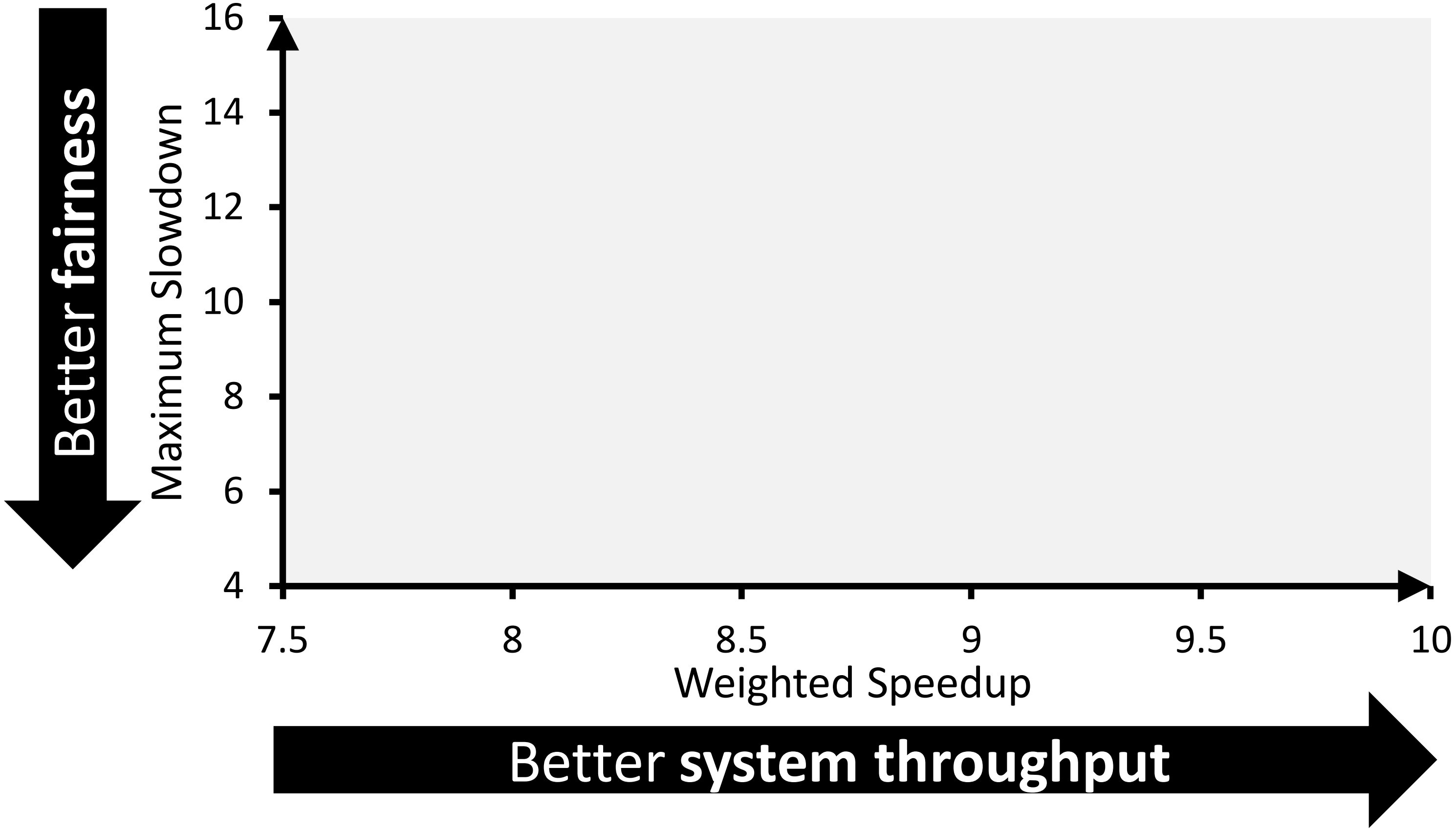
- Non-intensive threads not always prioritized ➔ Low throughput

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

- Most intensive thread starves ➔ Low fairness

TCM: Throughput and Fairness

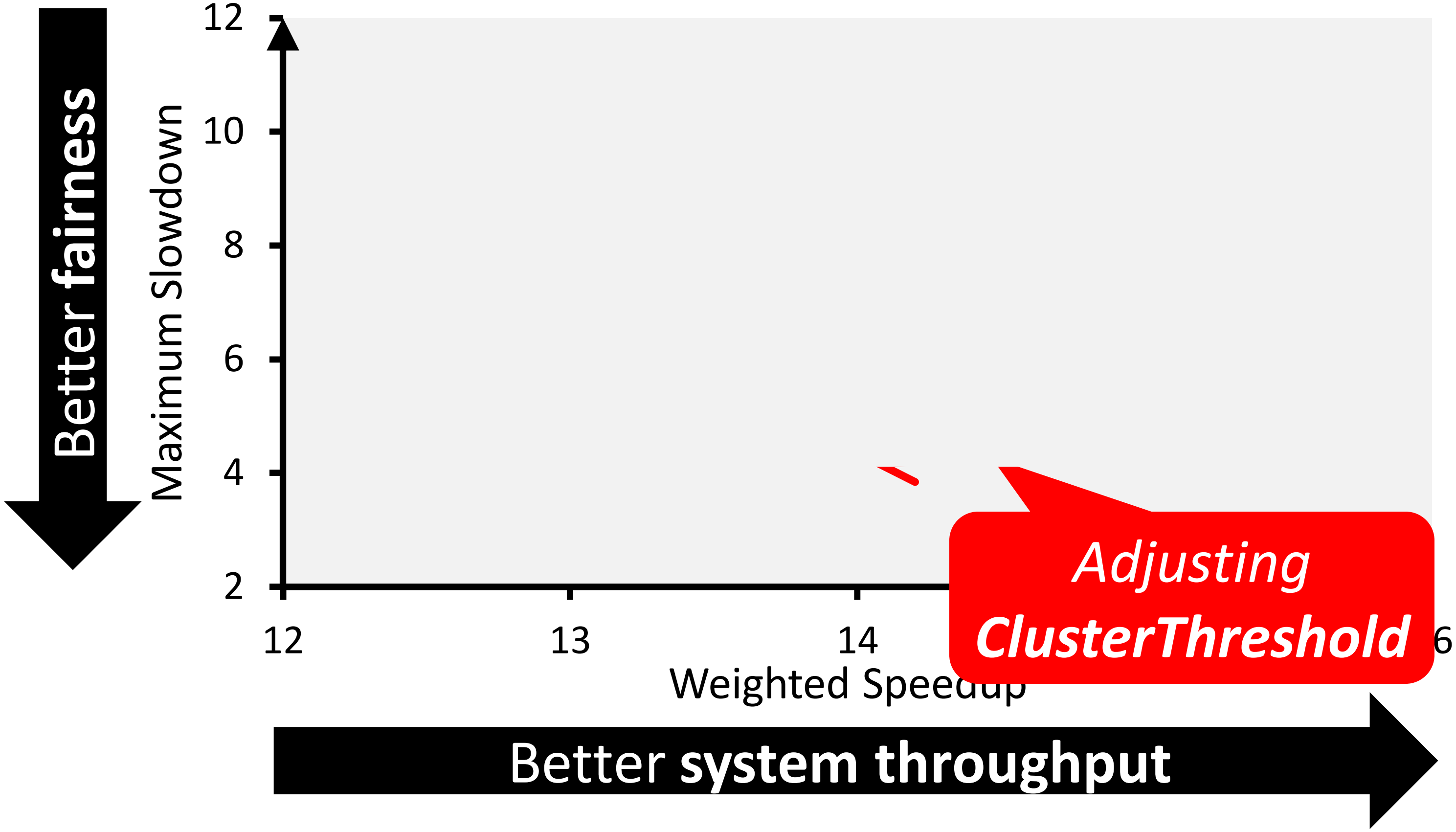
24 cores, 4 memory controllers, 96 workloads



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

TCM: Fairness-Throughput Tradeoff

When configuration parameter is varied…



Better **fairness**

Adjusting
ClusterThreshold

Better **system throughput**

TCM allows robust fairness-throughput tradeoff

Operating System Support

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

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

Conclusion

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

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

- *TCM provides the best system throughput and fairness*

TCM Pros and Cons

- Upsides:
  - Provides both high fairness and high performance

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

Staged Memory Scheduling

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

SMS ISCA 2012 Talk

SMS: Executive Summary

- **Observation:** Heterogeneous CPU-GPU systems require memory schedulers with large request buffers

- **Problem:** Existing monolithic application-aware memory scheduler designs are hard to scale to large request buffer sizes

- **Solution:** Staged Memory Scheduling (SMS)

  decomposes the memory controller into three simple stages:

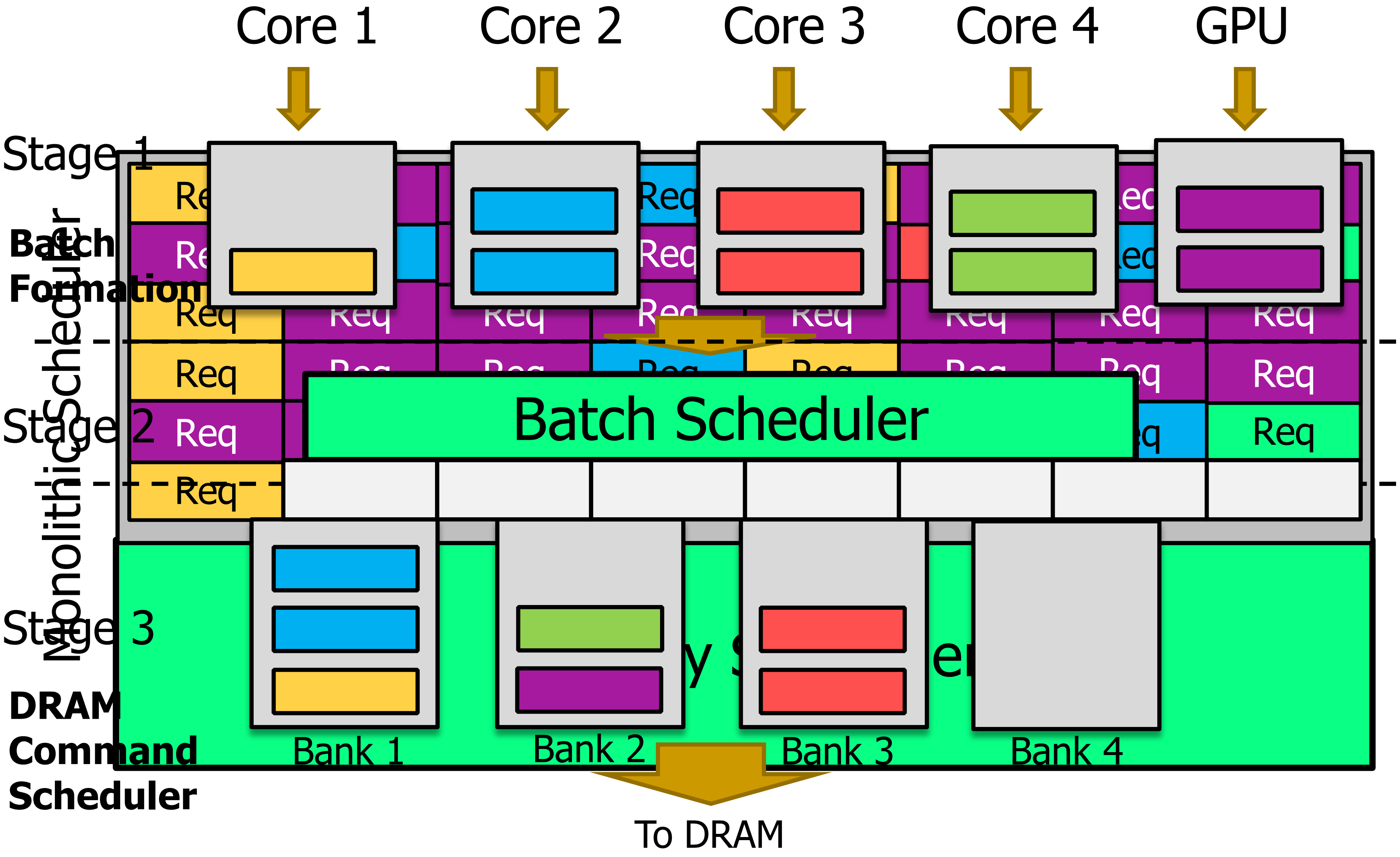
  1) Batch formation: maintains row buffer locality

  2) Batch scheduler: reduces interference between applications

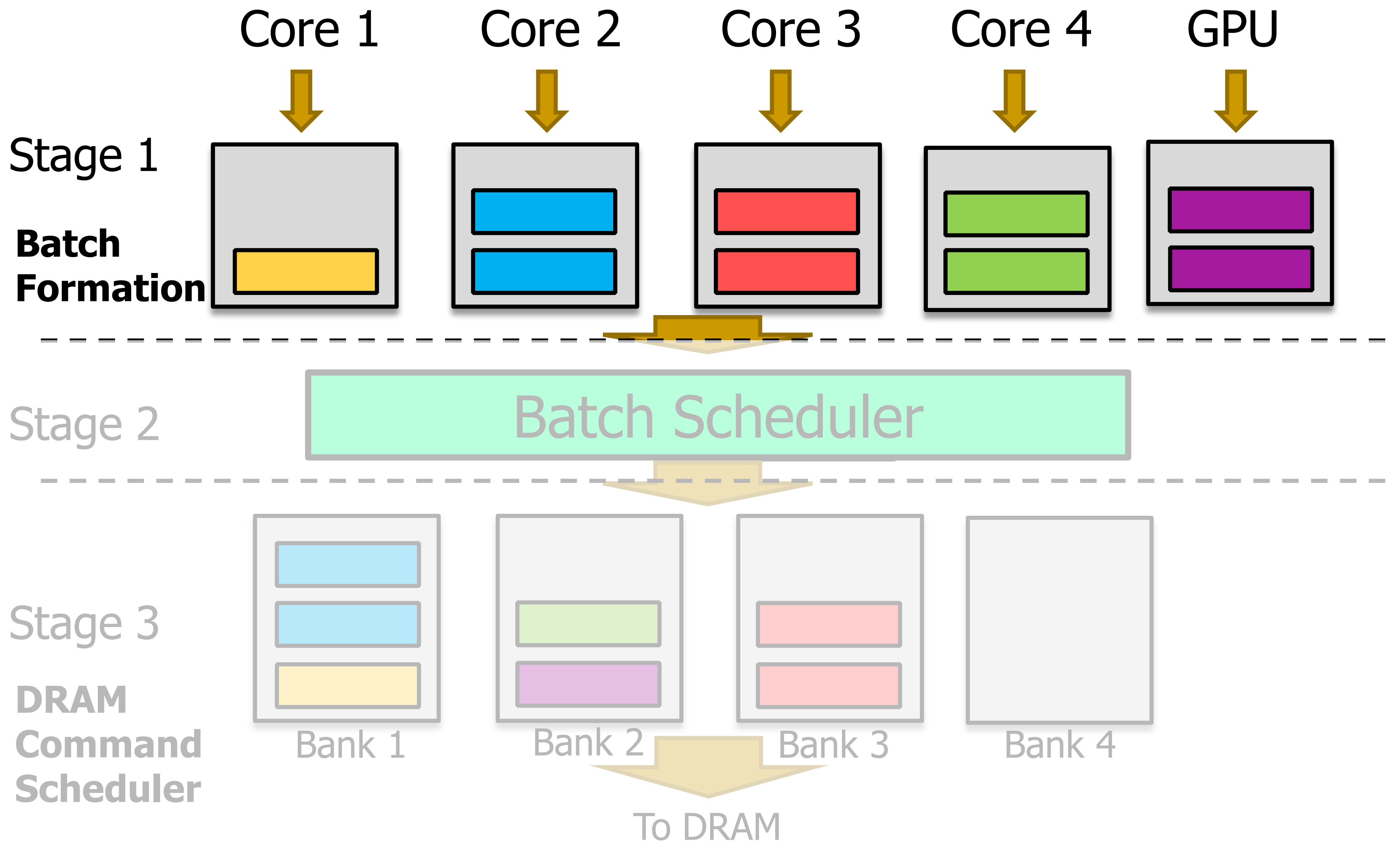
  3) DRAM command scheduler: issues requests to DRAM

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

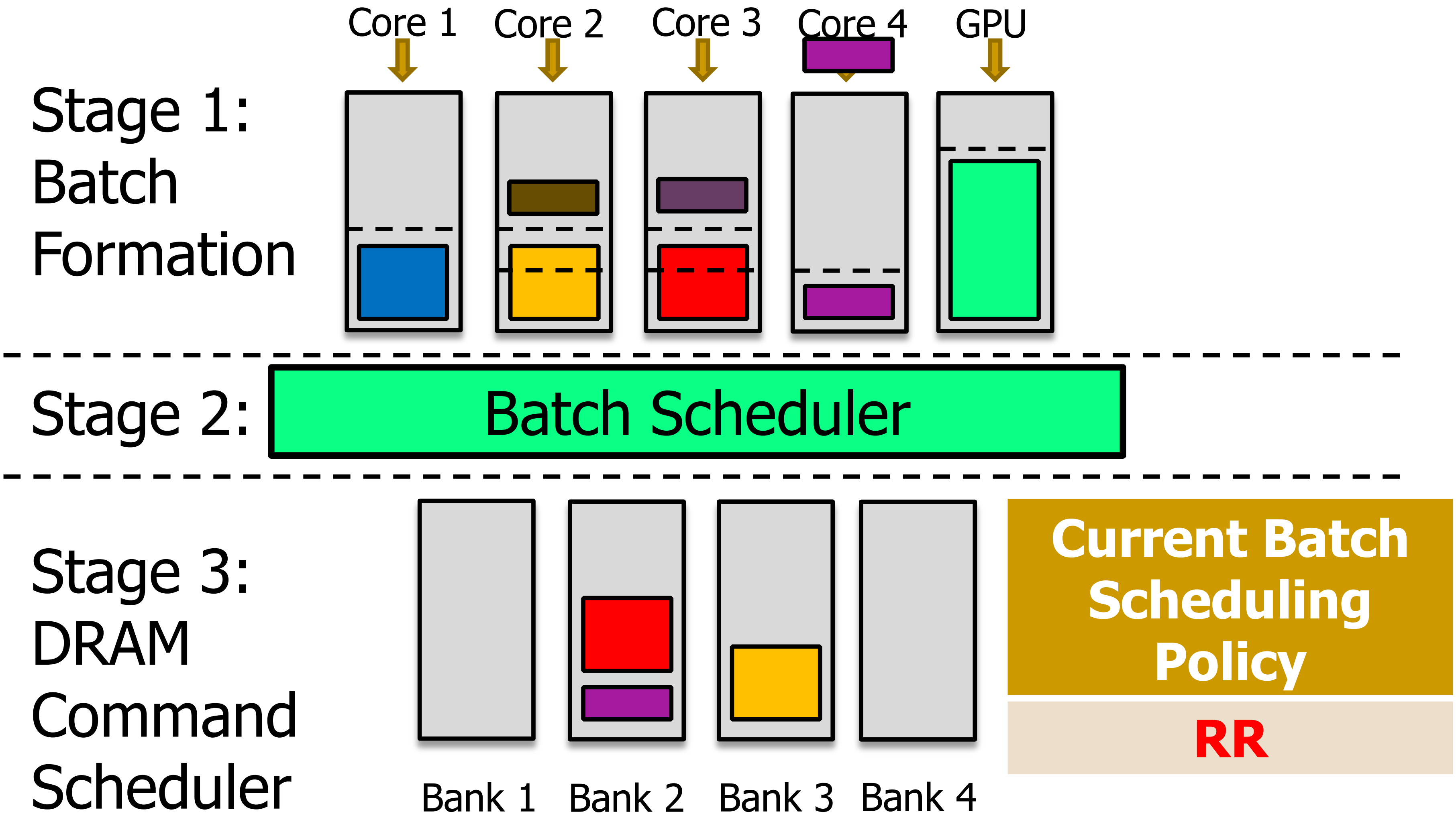
SMS: Staged Memory Scheduling



SMS: Staged Memory Scheduling



Putting Everything Together



Stage 1: Batch Formation

Stage 2: Batch Scheduler

Stage 3: DRAM Command Scheduler

Current Batch Scheduling Policy

RR

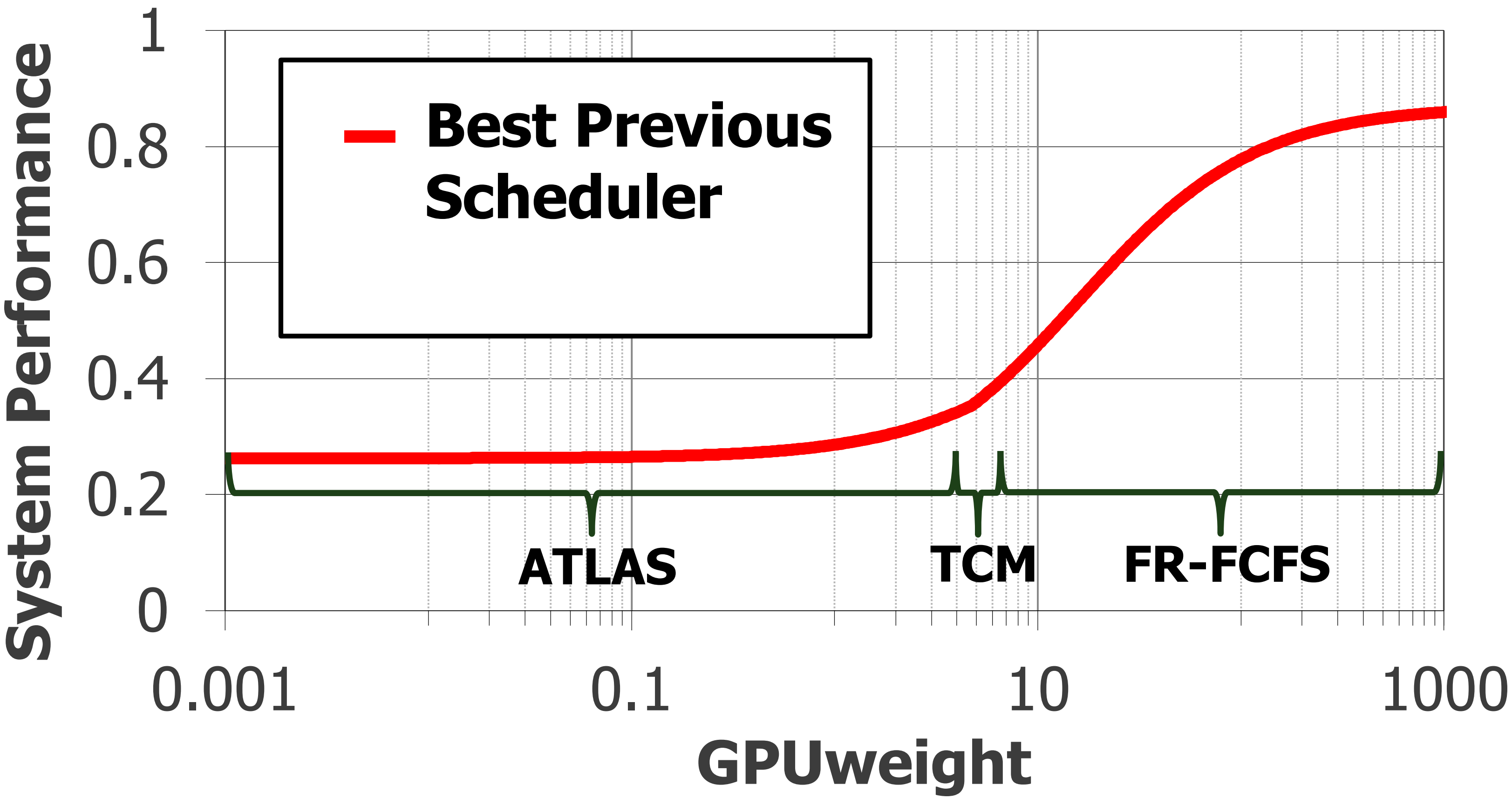
Complexity

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

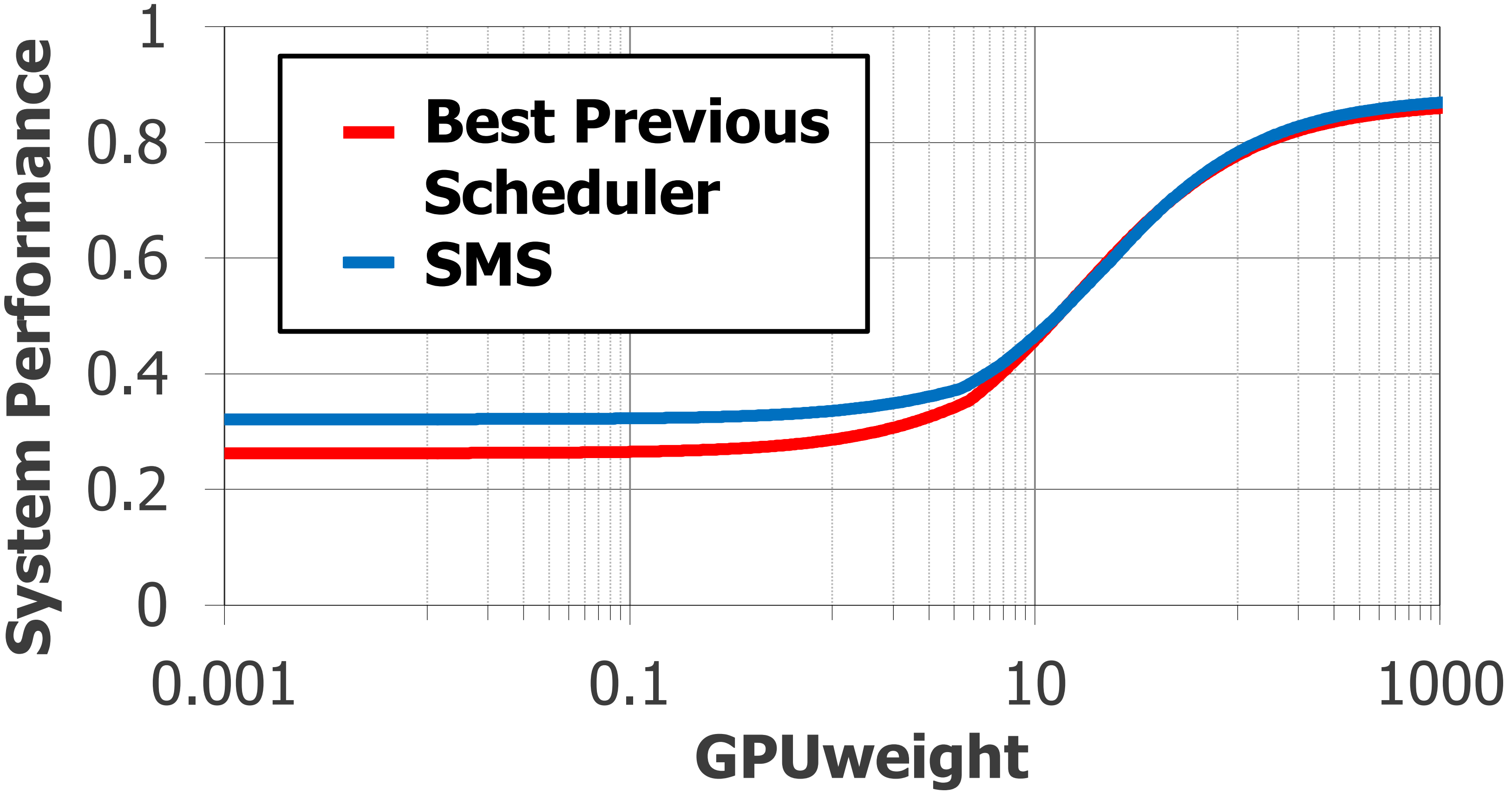
- Reduction comes from:
  - Monolithic scheduler → stages of simpler schedulers
  - Each stage has a simpler scheduler (considers fewer properties at a time to make the scheduling decision)
  - Each stage has simpler buffers (FIFO instead of out-of-order)
  - Each stage has a portion of the total buffer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

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

Performance at Different GPU Weights



Performance at Different GPU Weights



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

Stronger Memory Service Guarantees

Lavanya Subramanian, Vivek Seshadri, Yoongu Kim, Ben Jaiyen, and Onur Mutlu,
"MISE: Providing Performance Predictability and Improving Fairness in Shared Main Memory Systems"
Proceedings of the 19th International Symposium on High-Performance Computer Architecture (***HPCA***),
Shenzhen, China, February 2013. Slides (pptx)

Strong Memory Service Guarantees

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

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

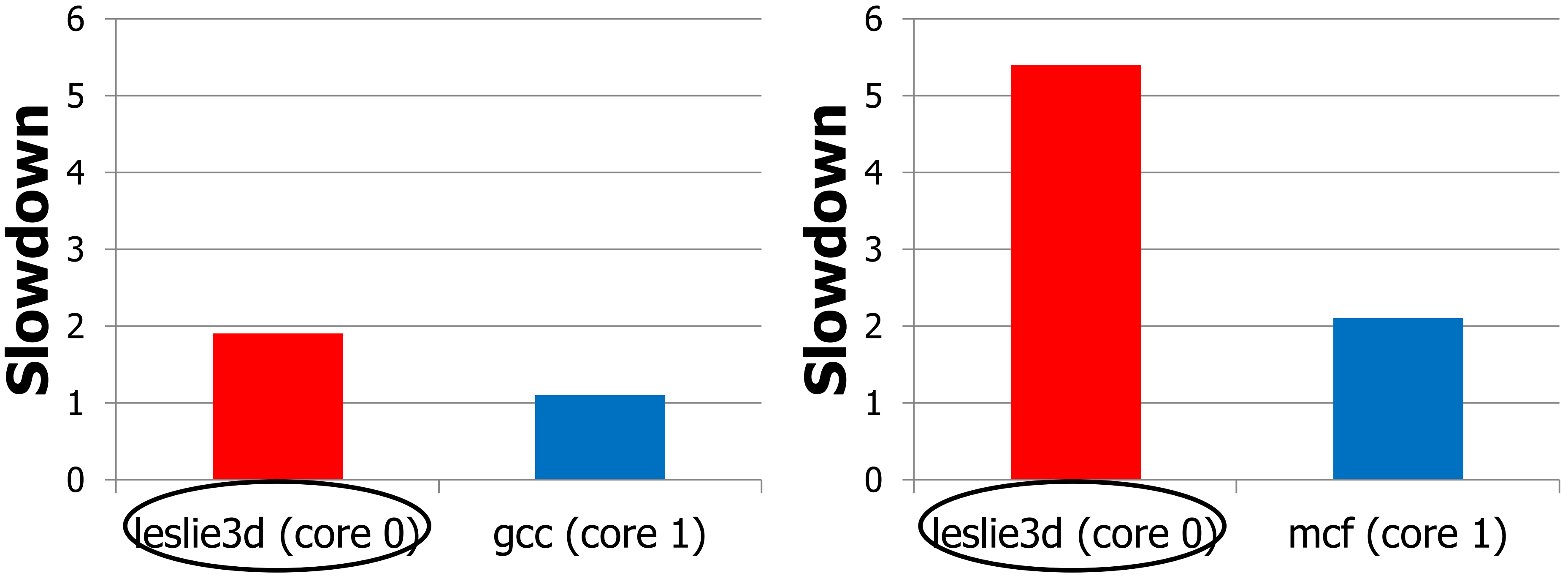
MISE: Providing Performance Predictability in Shared Main Memory Systems

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

SAFARI

Carnegie Mellon

Unpredictable Application Slowdowns



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

Need for Predictable Performance

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

Our Goal: Predictable performance
in the presence of memory interference

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

Outline

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

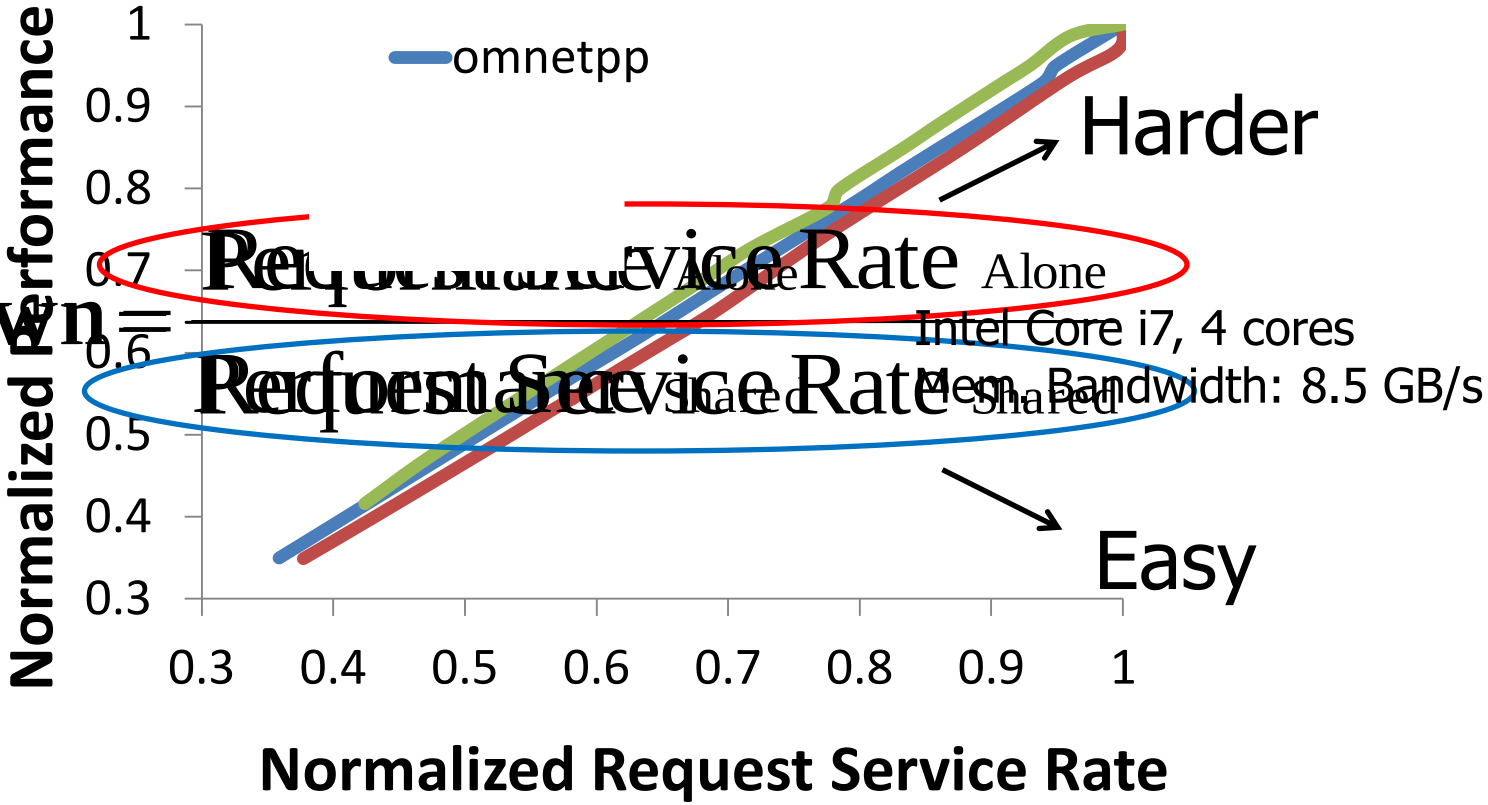
Slowdown: Definition

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

Key Observation 1

For a memory bound application,

Performance ∝ Memory request service rate



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

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

Intel Core i7, 4 cores

Mem. Bandwidth: 8.5 GB/s

Key Observation 2

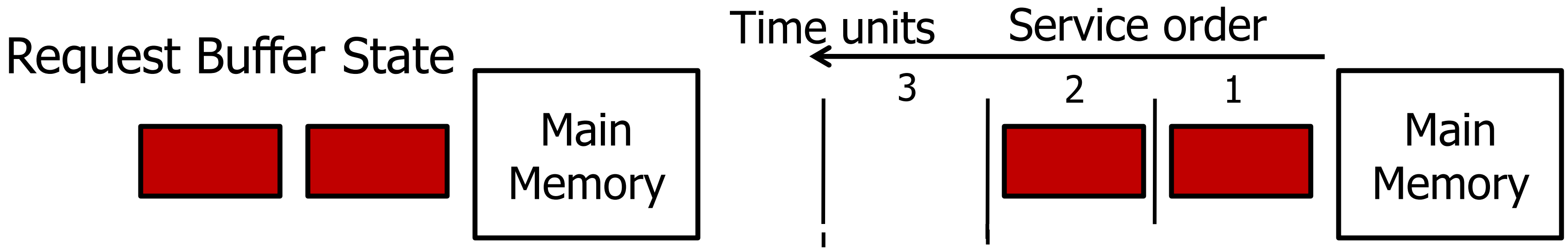
Request Service Rate $_{\text{Alone}}$ (RSR$_{\text{Alone}}$) of an application can be estimated by giving the application highest priority in accessing memory

Highest priority → Little interference

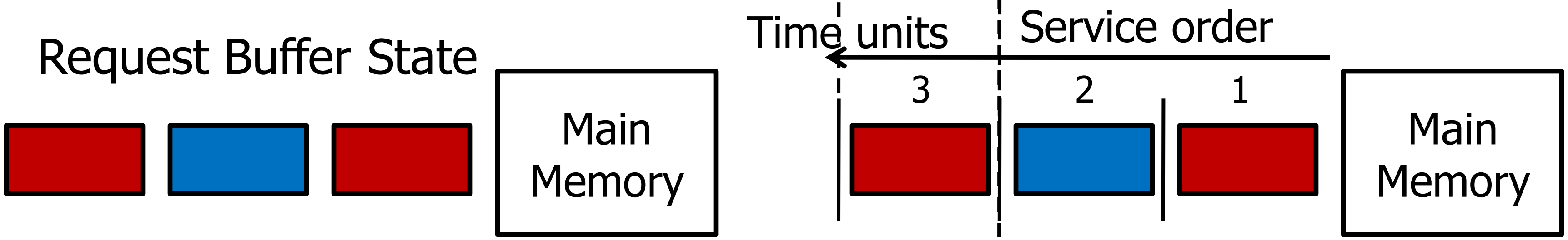
(almost as if the application were run alone)

Key Observation 2

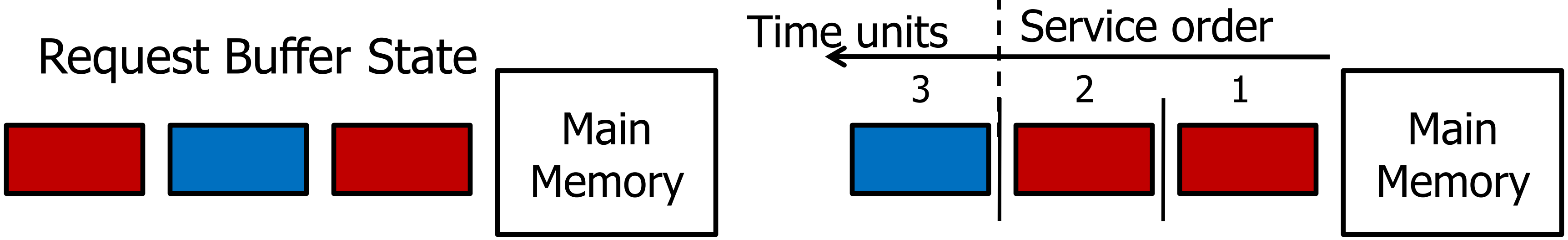
1. Run alone



2. Run with another application



3. Run with another application: highest priority

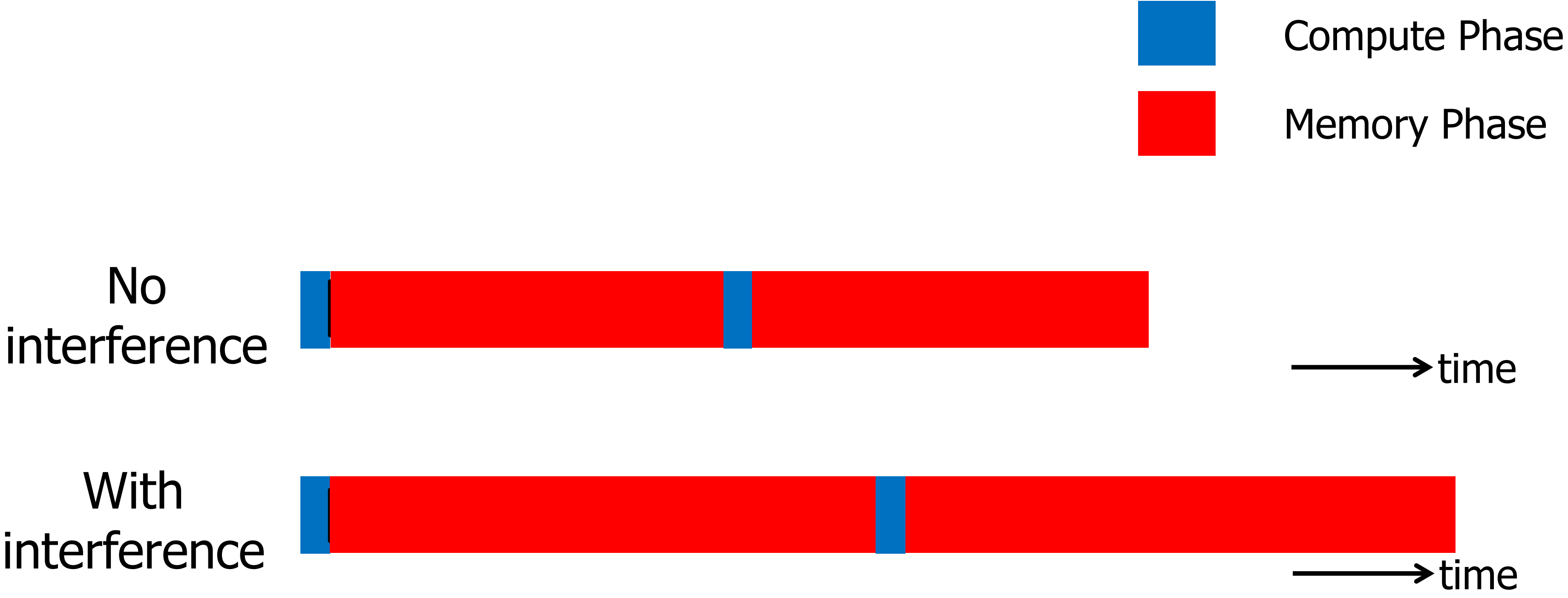


Memory Interference-induced Slowdown Estimation (MISE) model for memory bound applications

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

Key Observation 3

- Memory-bound application



🟦 Compute Phase

🟥 Memory Phase

No interference

time

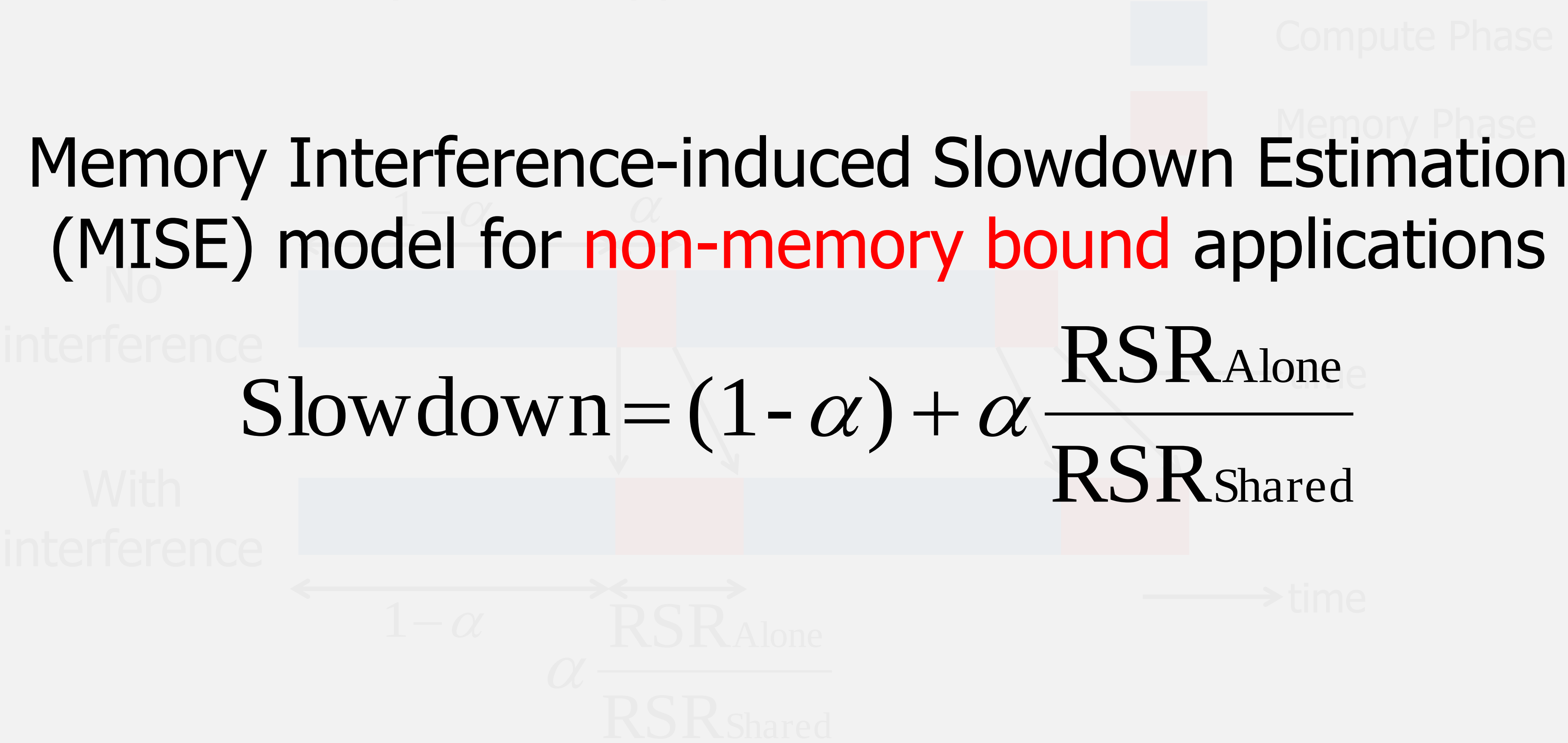
With interference

time

Memory phase slowdown dominates overall slowdown

Key Observation 3

- Non-memory-bound application



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

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

Only memory fraction (α) slows down with interference

Measuring $\text{RSR}_{\text{Shared}}$ 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 (a)
  - 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

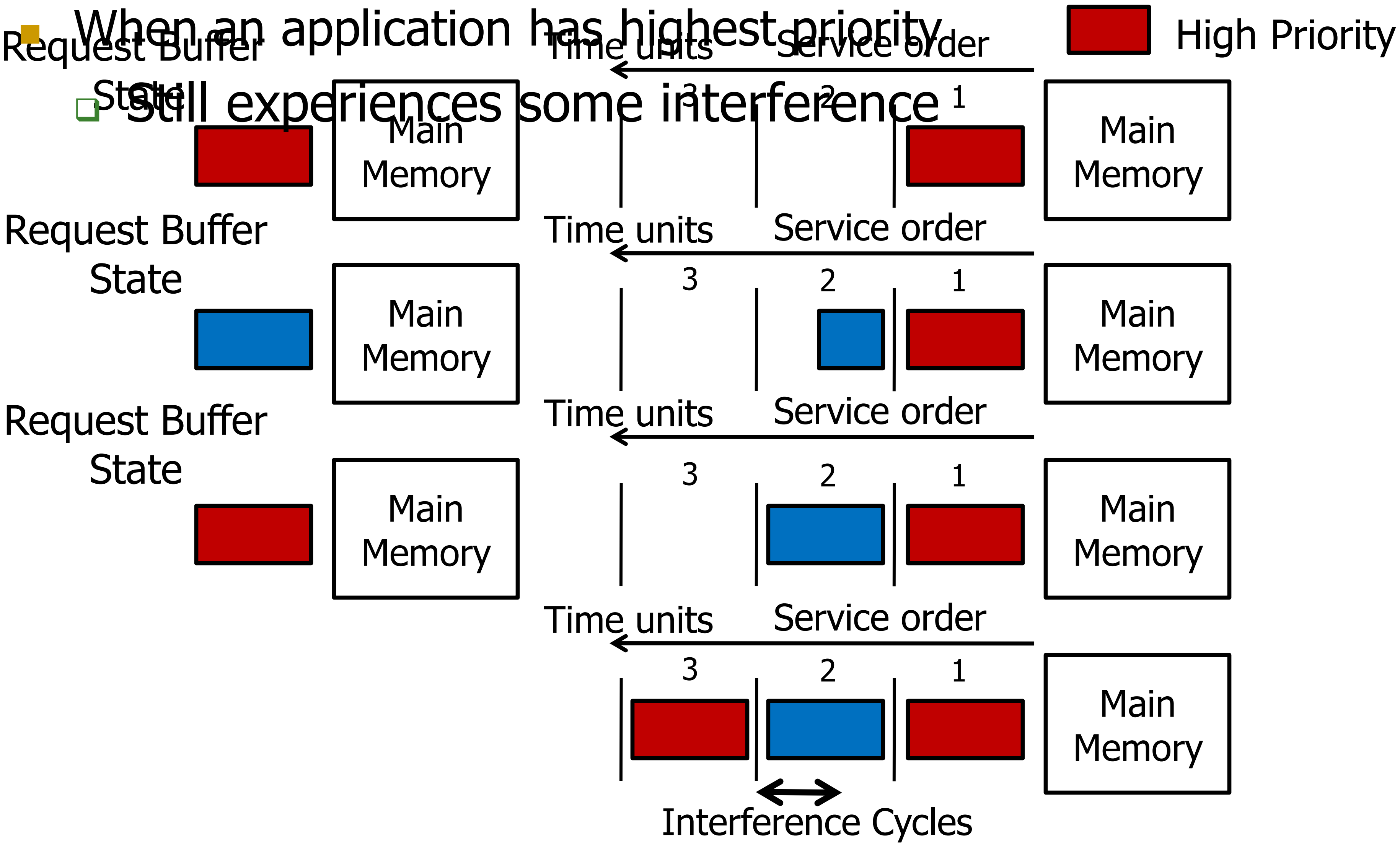
Goal: Estimate RSR$_{\text{Alone}}$

How: Periodically give each application highest priority in accessing memory

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

Inaccuracy in Estimating RSR_{Alone}

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



Accounting for Interference in RSR_{Alone} Estimation

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

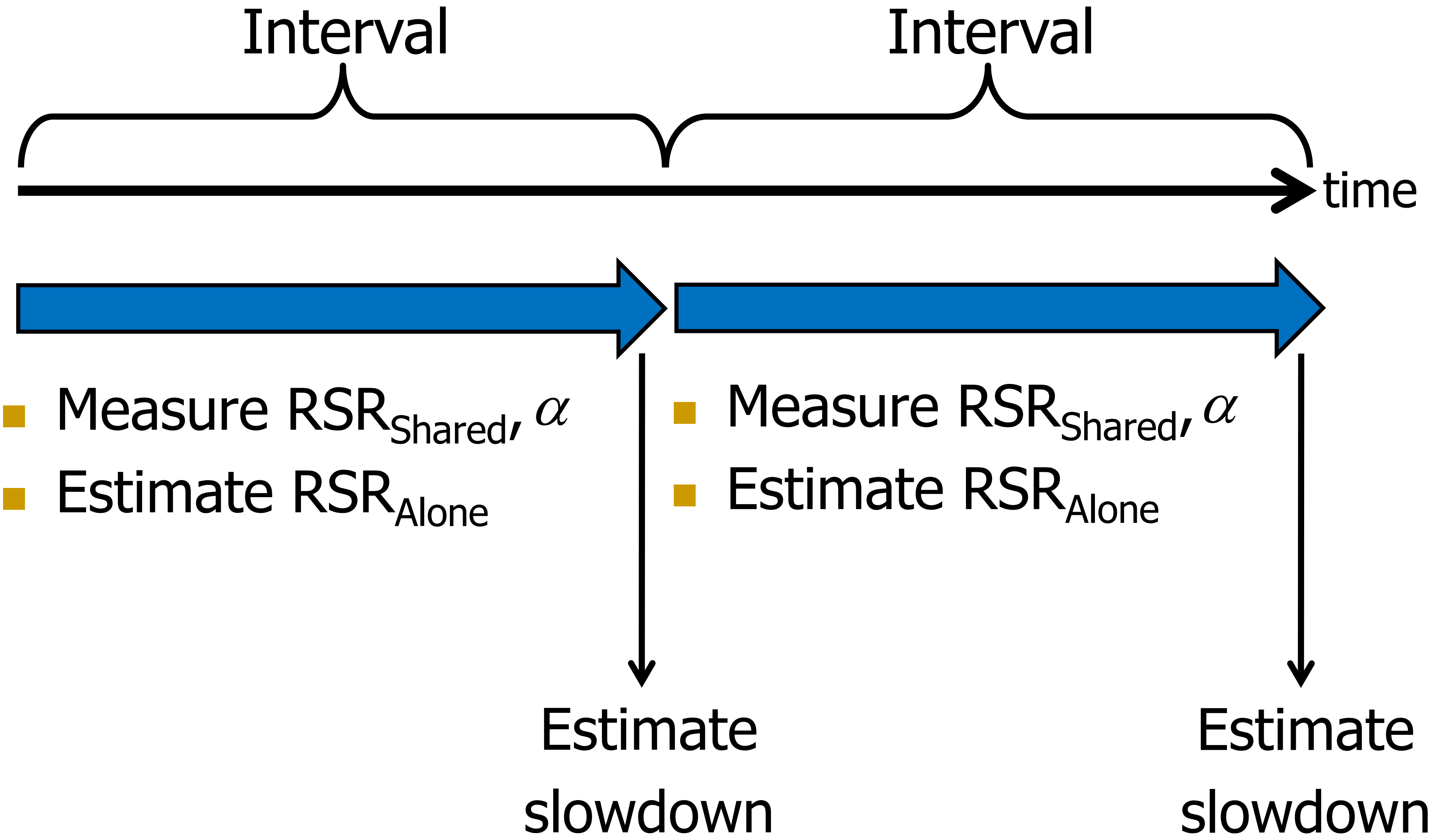
$$RSR_{Alone} = \frac{\text{Number of Requests During High Priority Epochs}}{\text{Number of Cycles Application Given High Priority} \boxed{- \text{Interference Cycles}}}$$

- 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



- Measure RSR_{Shared}, α
- Estimate RSR_{Alone}

- Measure RSR_{Shared}, α
- Estimate RSR_{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

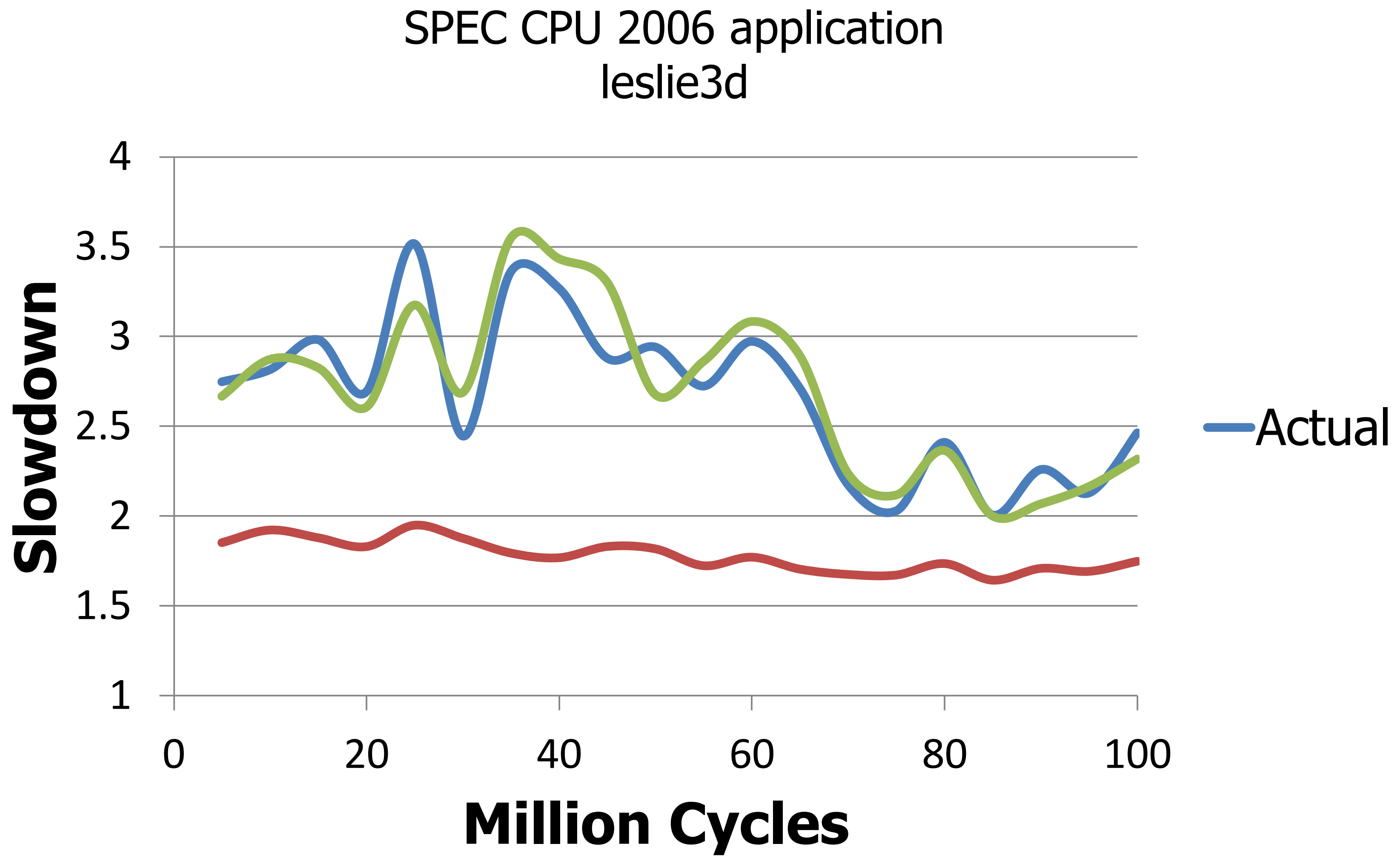
- Advantage 2:
    - STFM does not take into account compute phase for non-memory-bound applications
    - MISE accounts for compute phase → Better accuracy

Methodology

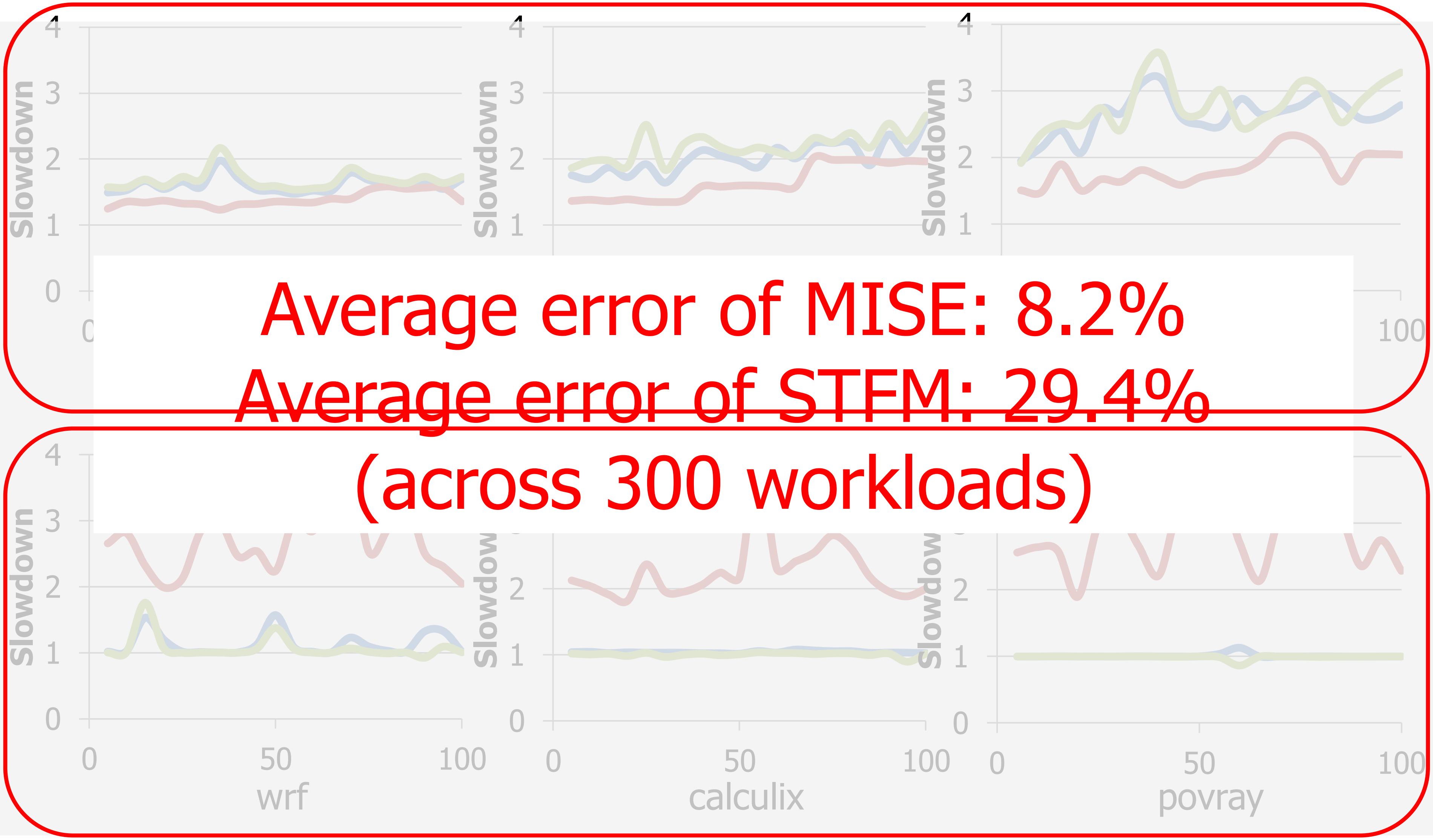
- Configuration of our simulated system
  - 4 cores
  - 1 channel, 8 banks/channel
  - DDR3 1066 DRAM
  - 512 KB private cache/core

- 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 allocatio
- When slowdown bound not met for N intervals
  - Notify the OS so it can migrate/de-schedule jobs

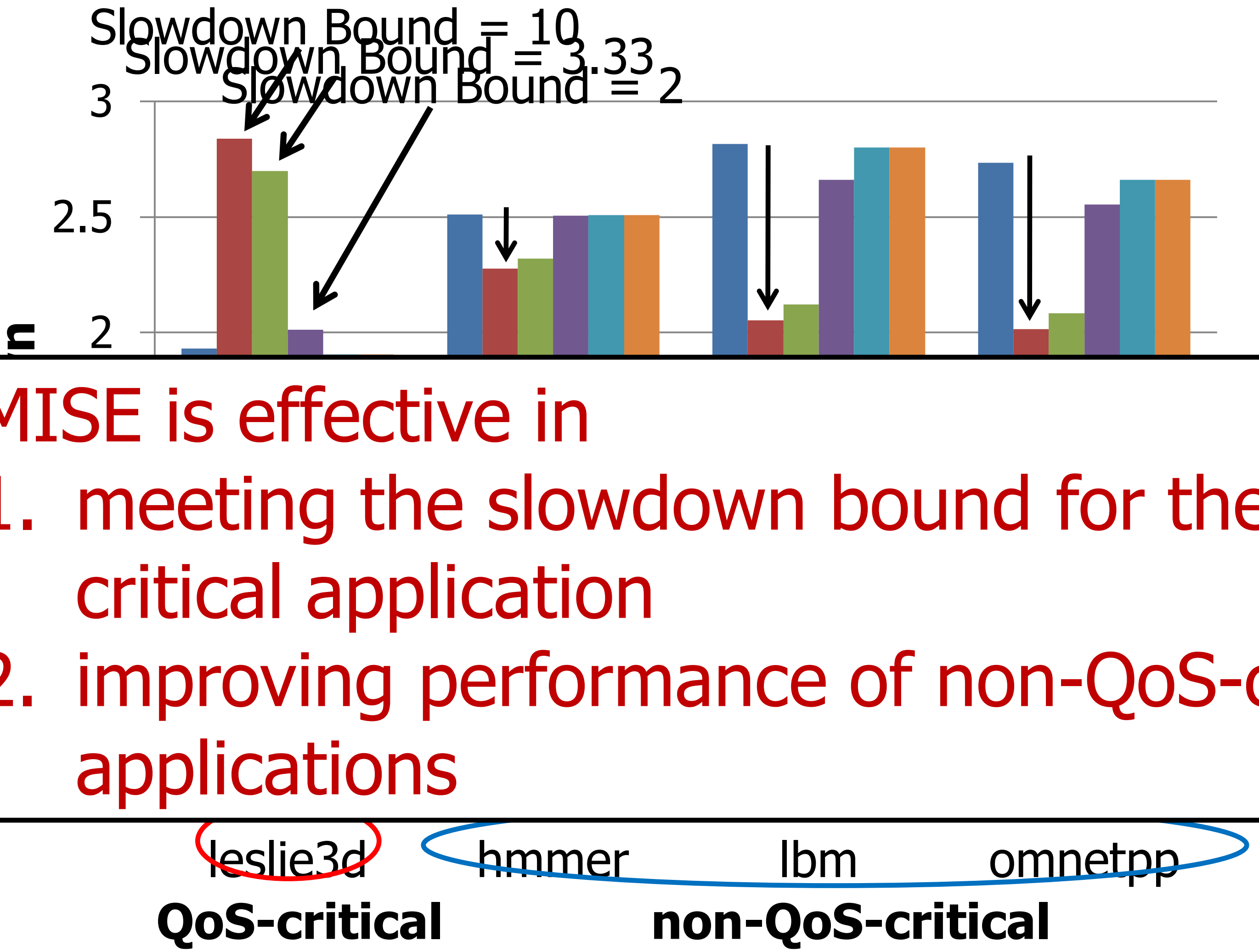
Methodology

- 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

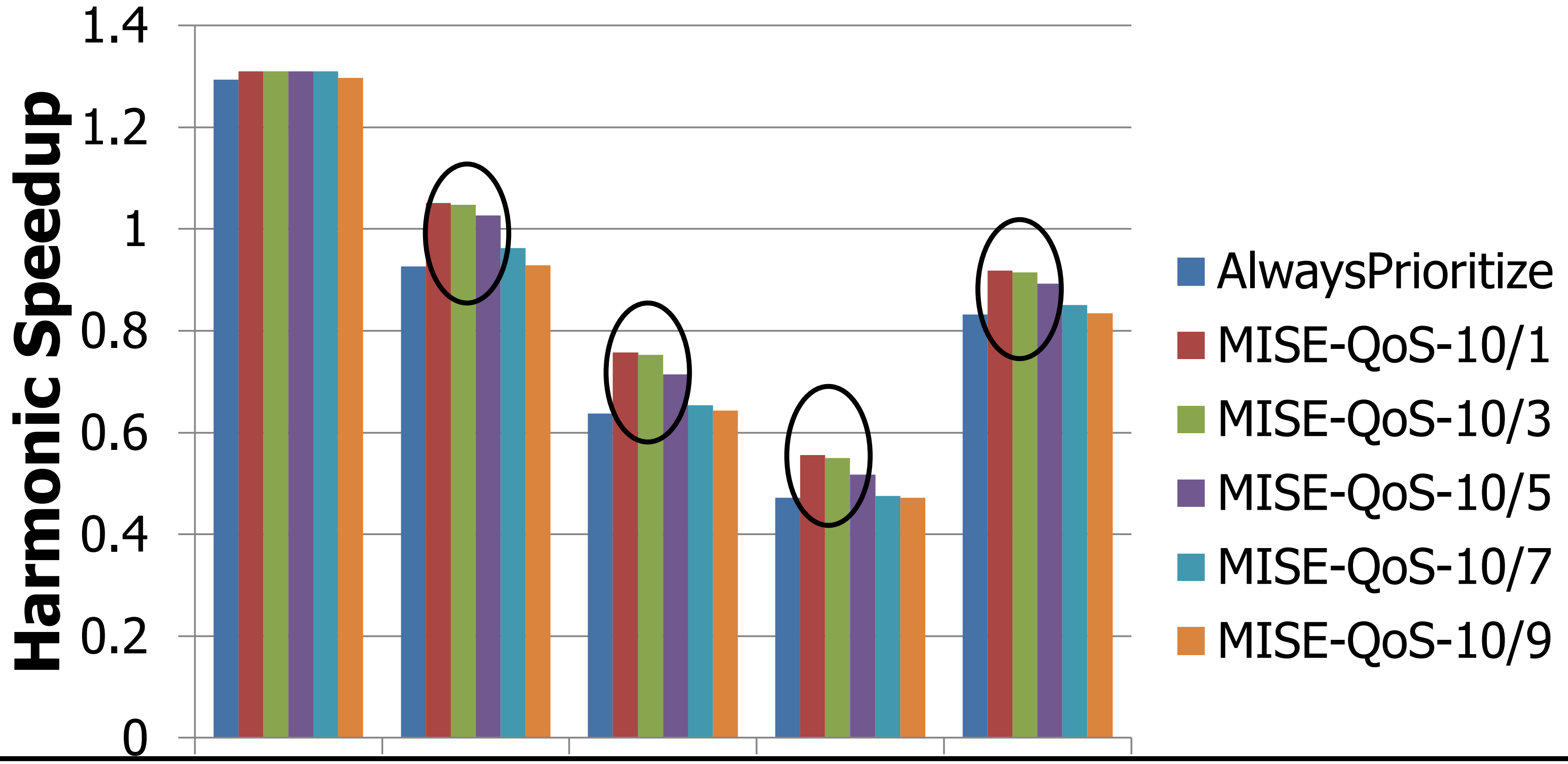
Effectiveness of MISE in Enforcing QoS

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

Memory Scheduling for Parallel Applications

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

Handling Interference in Parallel Applications

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

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

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

PAMS Micro 2011 Talk

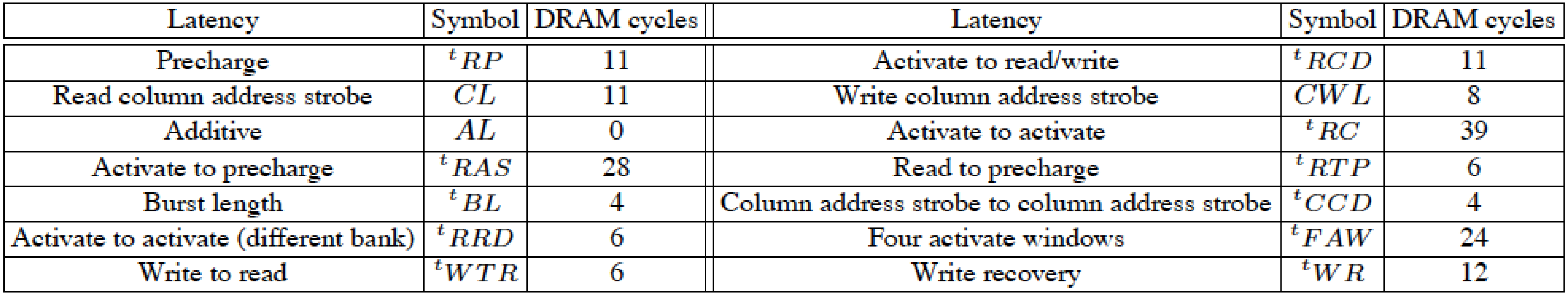
Aside:
Self-Optimizing Memory Controllers

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

Why are DRAM Controllers Difficult to Design?

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

Many DRAM Timing Constraints



Latency	Symbol	DRAM cycles	Latency	Symbol	DRAM cycles
Precharge	^{t}RP	11	Activate to read/write	^{t}RCD	11
Read column address strobe	CL	11	Write column address strobe	CWL	8
Additive	AL	0	Activate to activate	^{t}RC	39
Activate to precharge	^{t}RAS	28	Read to precharge	^{t}RTP	6
Burst length	^{t}BL	4	Column address strobe to column address strobe	^{t}CCD	4
Activate to activate (different bank)	^{t}RRD	6	Four activate windows	^{t}FAW	24
Write to read	^{t}WTR	6	Write recovery	^{t}WR	12

Table 4. DDR3 1600 DRAM timing specifications

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

More on DRAM Operation and Constraints

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

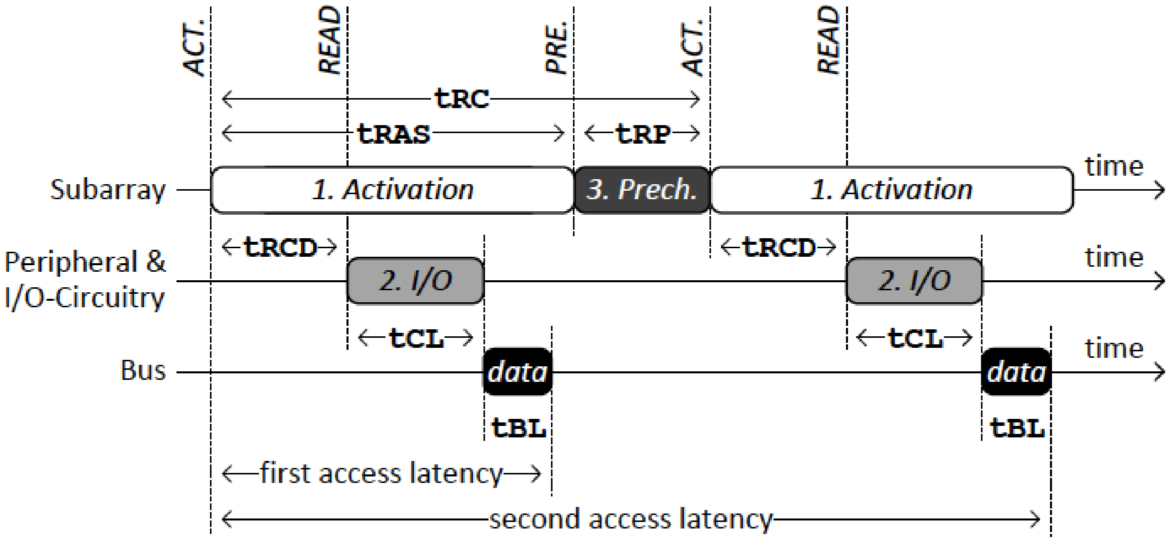
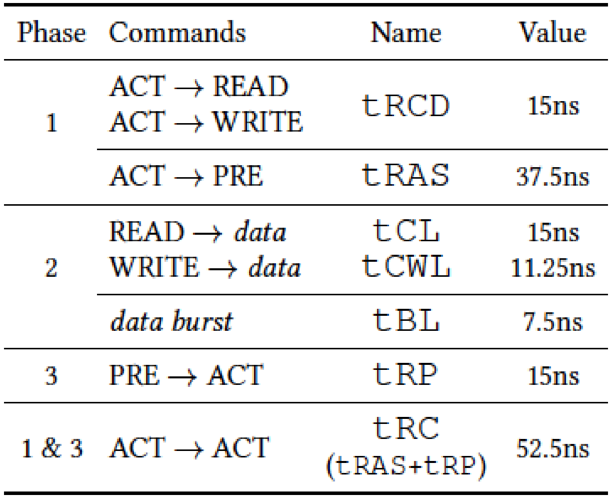


Figure 5. Three Phases of DRAM Access

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



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

Self-Optimizing DRAM Controllers

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

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

- Observation: Reinforcement learning maps nicely to memory control.

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

Self-Optimizing DRAM Controllers

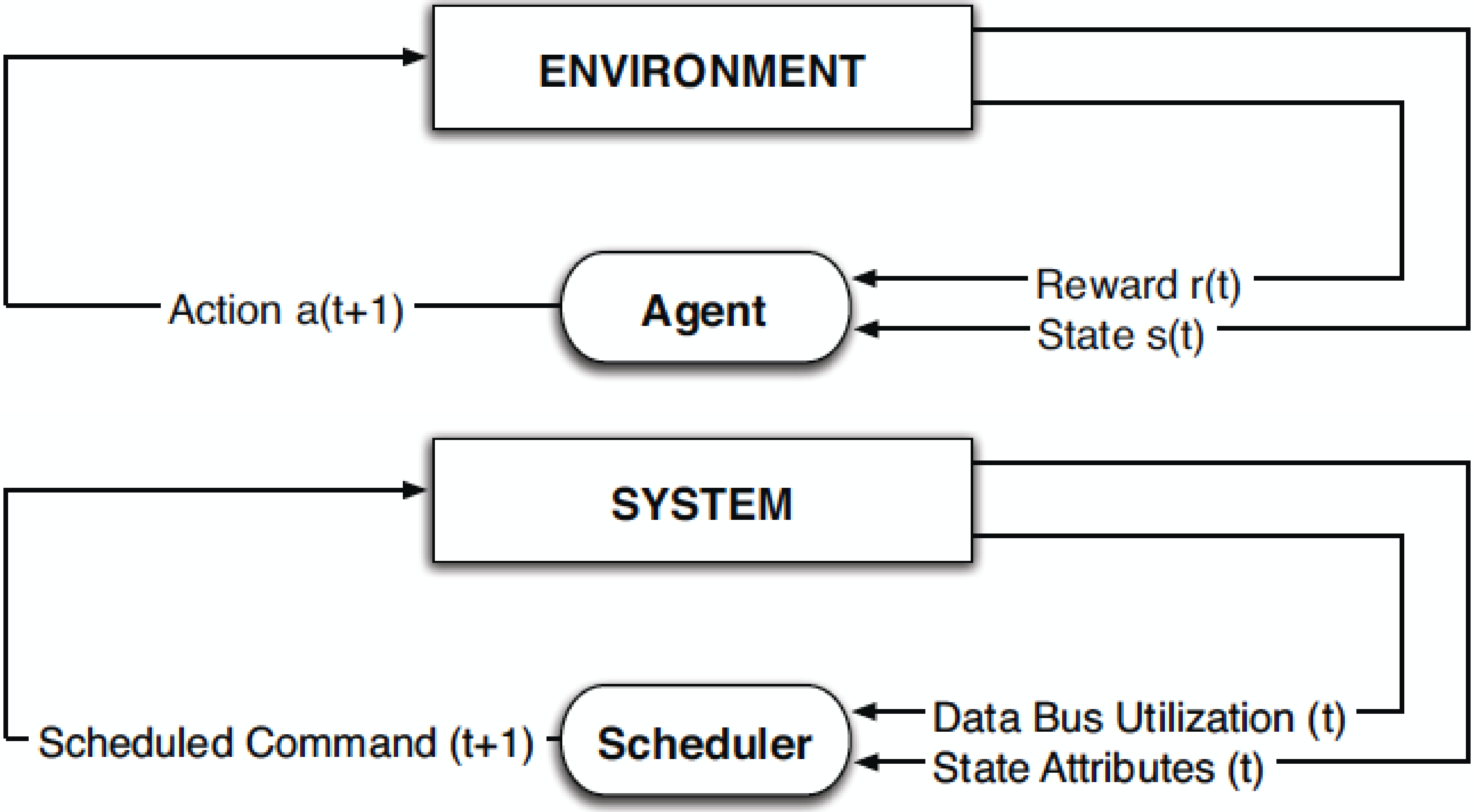


Figure 2: (a) Intelligent agent based on reinforcement learning principles; (b) DRAM scheduler as an RL-agent

Self-Optimizing DRAM Controllers

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

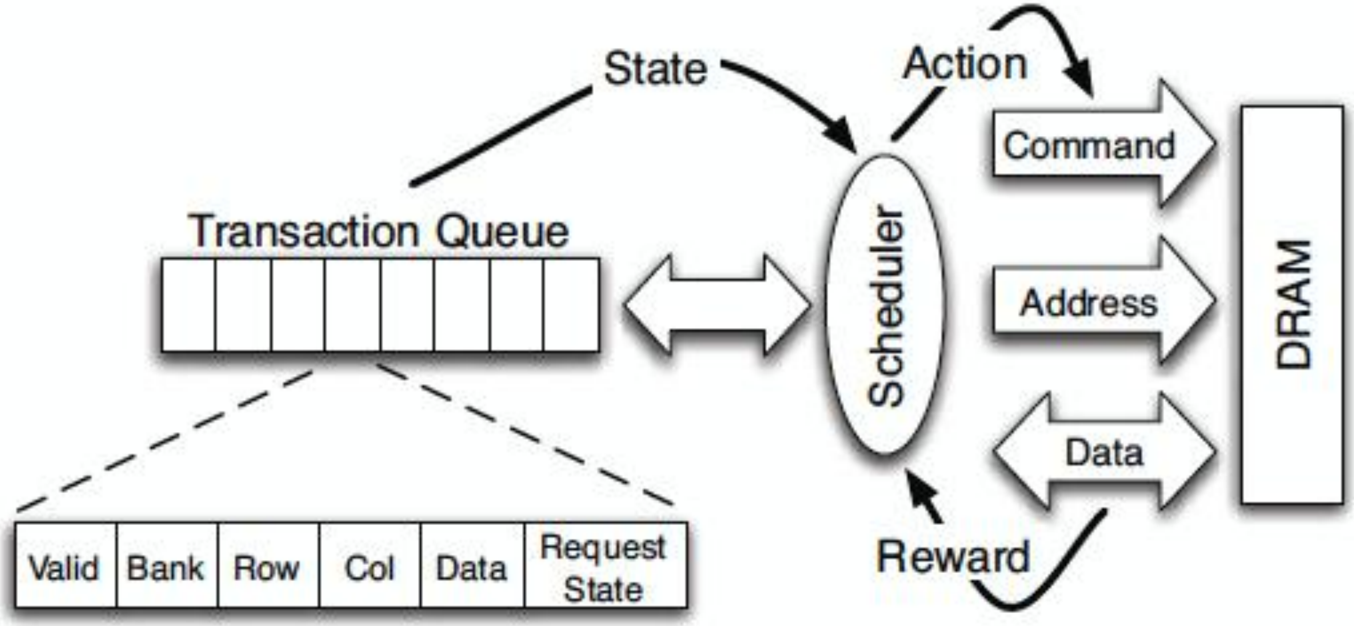


Figure 4: High-level overview of an RL-based scheduler.

Performance Results

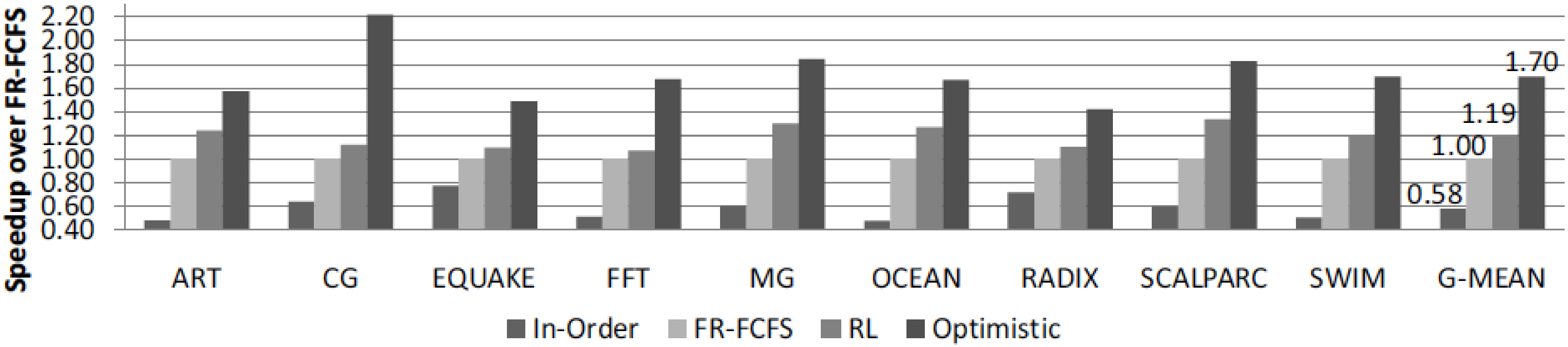


Figure 7: Performance comparison of in-order, FR-FCFS, RL-based, and optimistic memory controllers

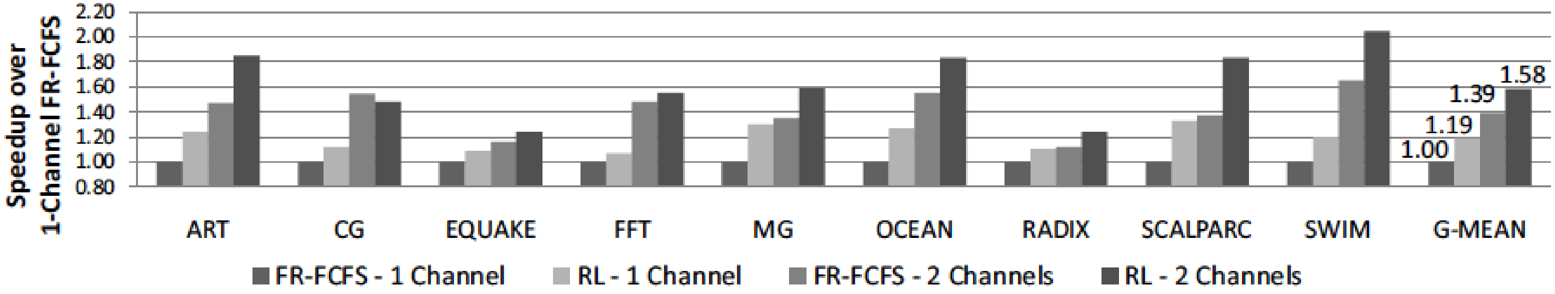


Figure 15: Performance comparison of FR-FCFS and RL-based memory controllers on systems with 6.4GB/s and 12.8GB/s peak DRAM bandwidth

QoS-Aware Memory Systems: The Dumb Resources Approach

Designing QoS-Aware Memory Systems: Approaches

- Smart resources: Design each shared resource to have a configurable interference control/reduction mechanism
  - QoS-aware memory controllers [Mutlu+ MICRO'07] [Moscibroda+, Usenix Security'07] [Mutlu+ ISCA'08, Top Picks'09] [Kim+ HPCA'10] [Kim+ MICRO'10, Top Picks'11] [Ebrahimi+ ISCA'11, MICRO'11] [Ausavarungnirun+, ISCA'12] [Subramanian+, HPCA'13]
  - QoS-aware interconnects [Das+ MICRO'09, ISCA'10, Top Picks '11] [Grot+ MICRO'09, ISCA'11, Top Picks '12]
  - QoS-aware caches

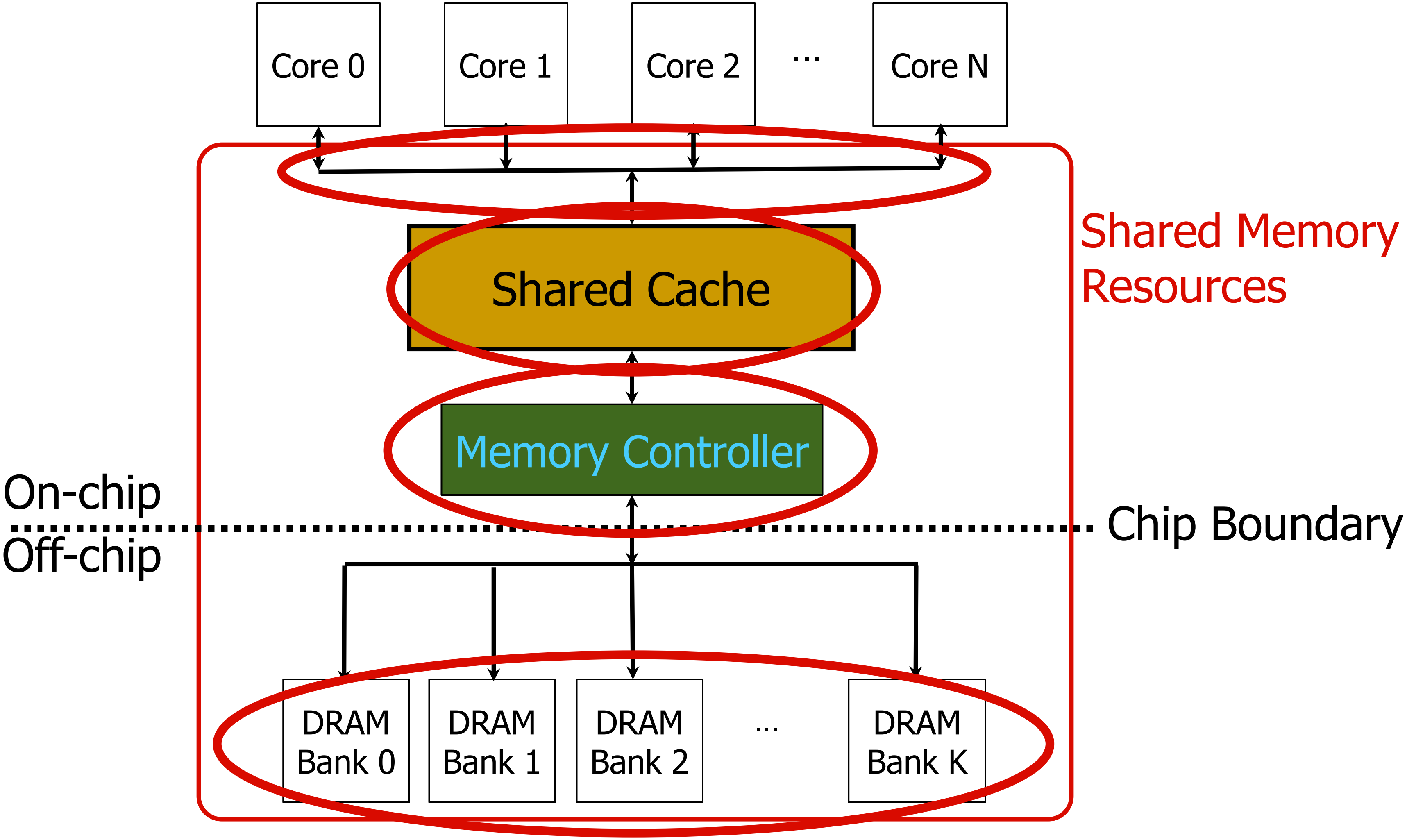
- Dumb resources: Keep each resource free-for-all, but reduce/control interference by injection control or data mapping
  - Source throttling to control access to memory system [Ebrahimi+ ASPLOS'10, ISCA'11, TOCS'12] [Ebrahimi+ MICRO'09] [Nychis+ HotNets'10]
  - QoS-aware data mapping to memory controllers [Muralidhara+ MICRO'11]
  - QoS-aware thread scheduling to cores [Das+ HPCA'13]

Fairness via Source Throttling

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

FST ASPLOS 2010 Talk

Many Shared Resources



The Problem with "Smart Resources"

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

- Explicitly coordinating mechanisms for different resources requires complex implementation

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

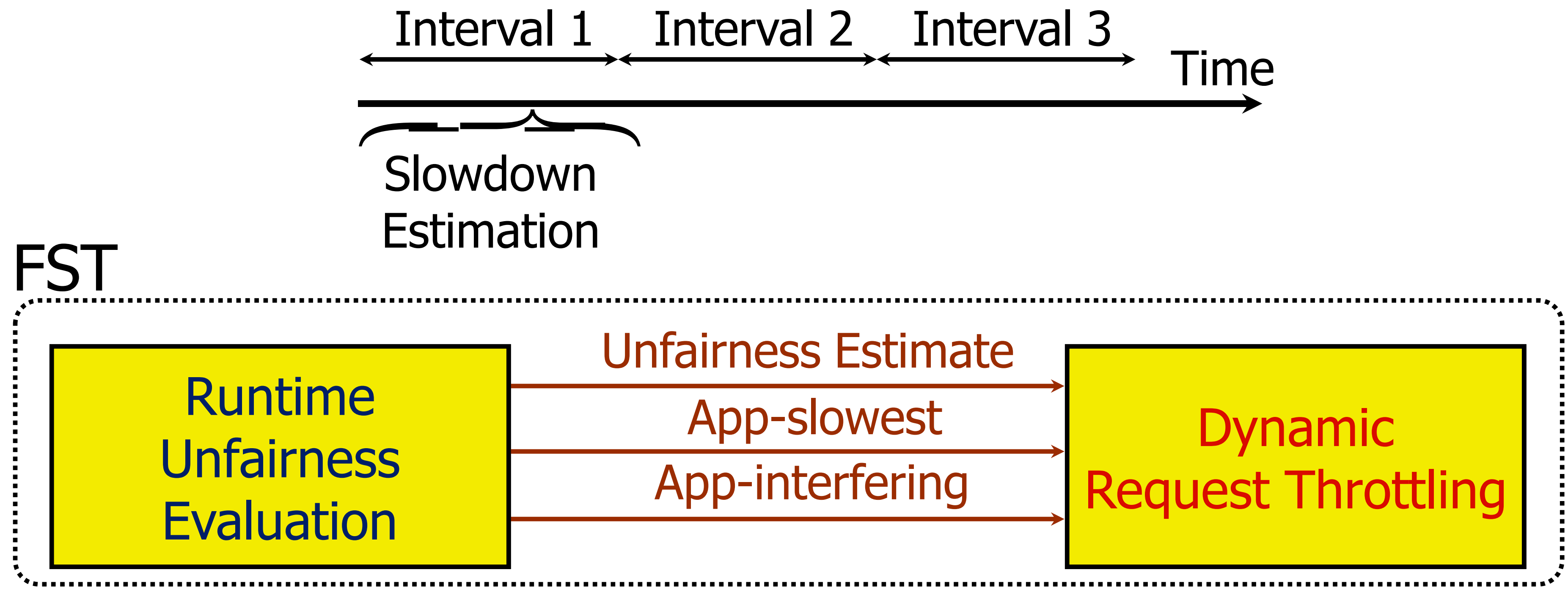
An Alternative Approach: Source Throttling

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

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

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

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



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
   (limit injection rate and parallelism)
 2-Throttle up App-slowest
}
```

System Software Support

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

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

Source Throttling Results: Takeaways

- Source throttling alone provides better performance than a combination of "smart" memory scheduling and fair caching
    - Decisions made at the memory scheduler and the cache sometimes contradict each other

- Neither source throttling alone nor "smart resources" alone provides the best performance

- Combined approaches are even more powerful
    - Source throttling and resource-based interference control

Designing QoS-Aware Memory Systems: Approaches

- Smart resources: Design each shared resource to have a configurable interference control/reduction mechanism
  - QoS-aware memory controllers [Mutlu+ MICRO'07] [Moscibroda+, Usenix Security'07] [Mutlu+ ISCA'08, Top Picks'09] [Kim+ HPCA'10] [Kim+ MICRO'10, Top Picks'11] [Ebrahimi+ ISCA'11, MICRO'11] [Ausavarungnirun+, ISCA'12] [Subramanian+, HPCA'13]
  - QoS-aware interconnects [Das+ MICRO'09, ISCA'10, Top Picks '11] [Grot+ MICRO'09, ISCA'11, Top Picks '12]
  - QoS-aware caches

- Dumb resources: Keep each resource free-for-all, but reduce/control interference by injection control or data mapping
  - Source throttling to control access to memory system [Ebrahimi+ ASPLOS'10, ISCA'11, TOCS'12] [Ebrahimi+ MICRO'09] [Nychis+ HotNets'10] [Nychis+ SIGCOMM'12]
  - QoS-aware data mapping to memory controllers [Muralidhara+ MICRO'11]
  - QoS-aware thread scheduling to cores [Das+ HPCA'13]

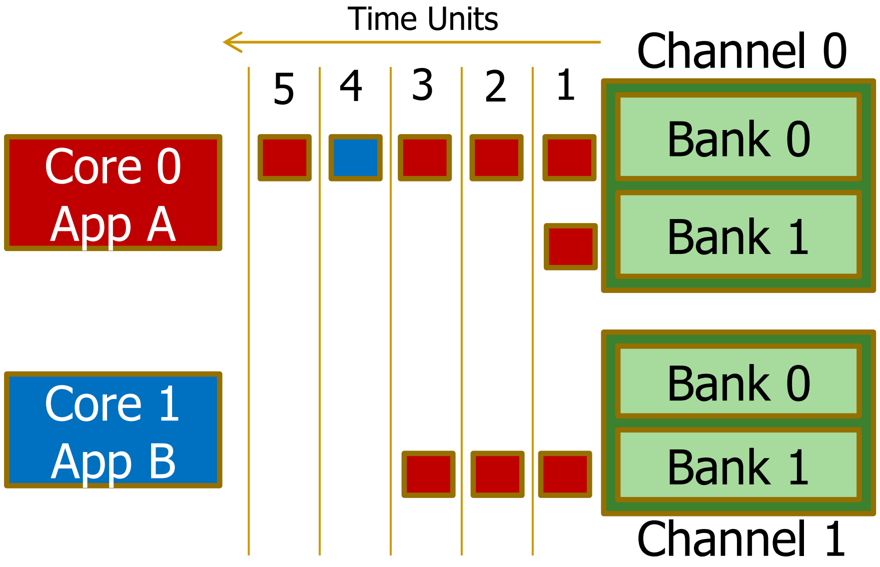
Memory Channel Partitioning

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

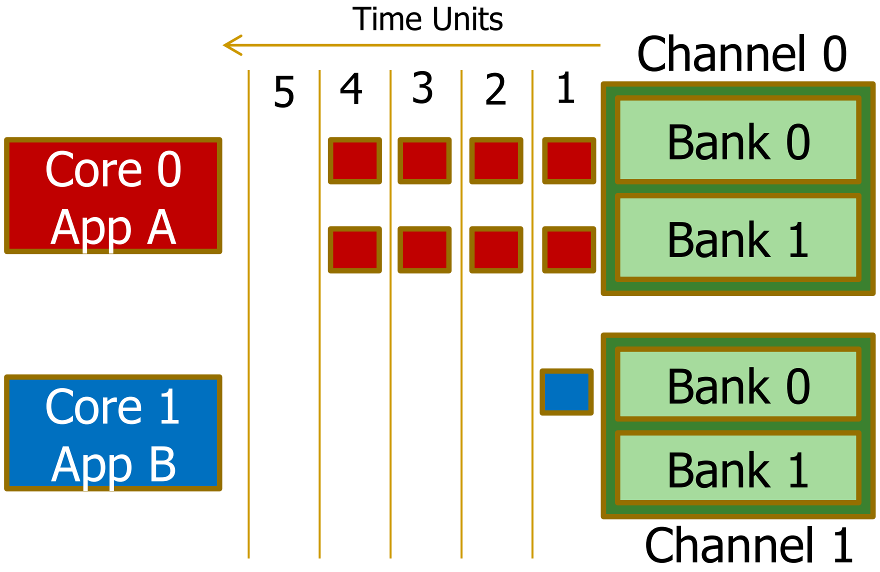
MCP Micro 2011 Talk

Another Way to Reduce Memory Interference

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



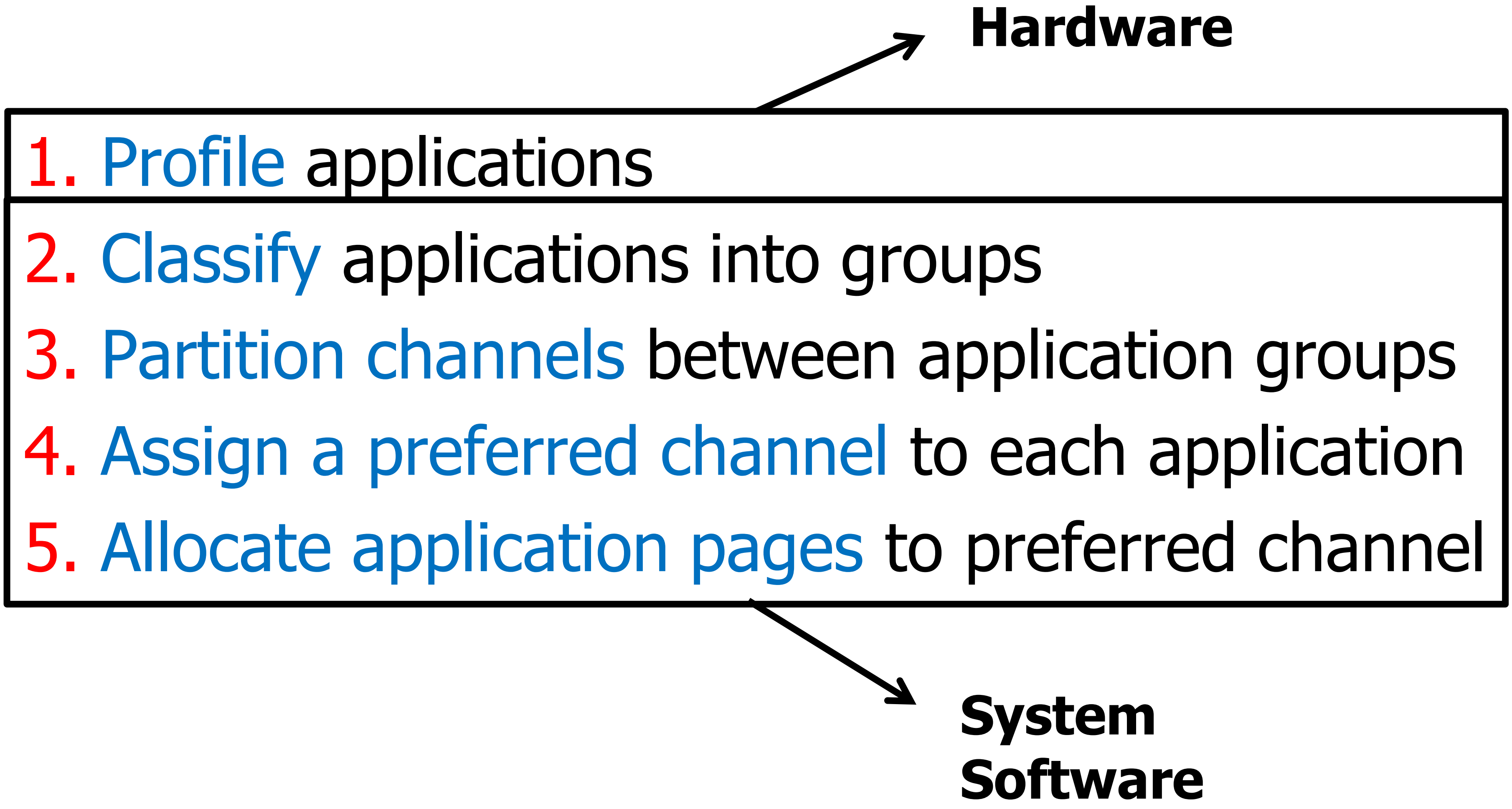
Conventional Page Mapping



Channel Partitioning

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

Memory Channel Partitioning (MCP) Mechanism



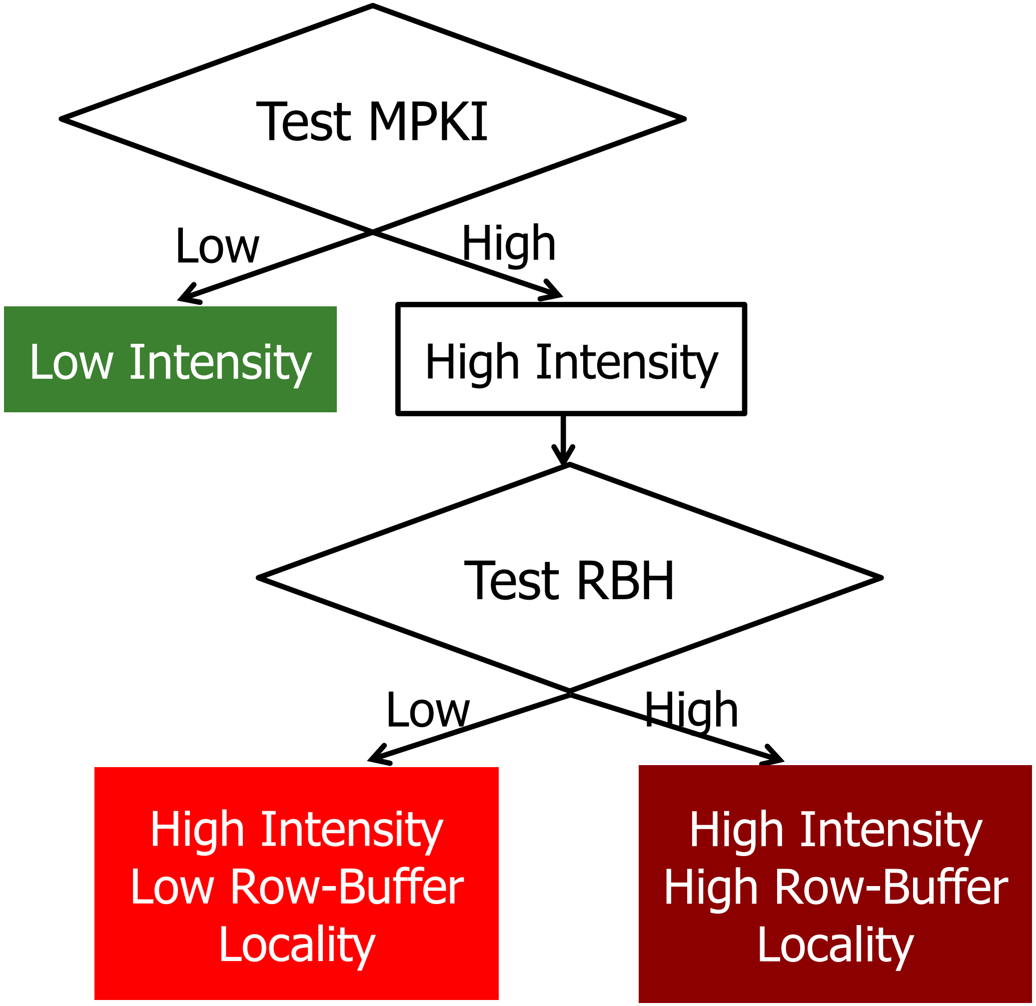
Hardware

1. Profile applications

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

System Software

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

Summary: Memory QoS

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

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

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

Summary: Memory QoS Approaches and Techniques

- Approaches: Smart vs. dumb resources
  - Smart resources: QoS-aware memory scheduling
  - Dumb resources: Source throttling; channel partitioning
  - Both approaches are effective in reducing interference
  - No single best approach for all workloads

- Techniques: Request/thread scheduling, source throttling, memory partitioning
  - All approaches are effective in reducing interference
  - Can be applied at different levels: hardware vs. software
  - No single best technique for all workloads

- Combined approaches and techniques are the most powerful
  - Integrated Memory Channel Partitioning and Scheduling [MICRO'11]

Computer Architecture: Memory Interference and QoS (Part II)

Prof. Onur Mutlu
Carnegie Mellon University