Providing High and Predictable Performance in Multicore Systems Through Shared Resource Management

Thesis Defense

Lavanya Subramanian

Committee:

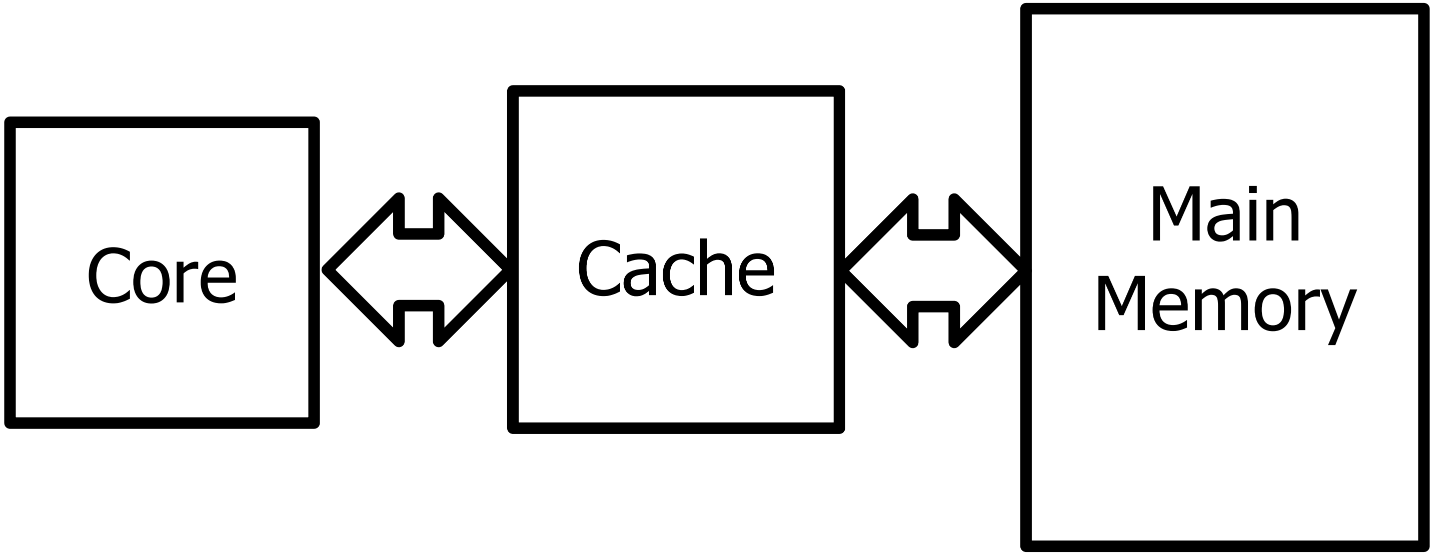
Advisor: Onur Mutlu

Greg Ganger

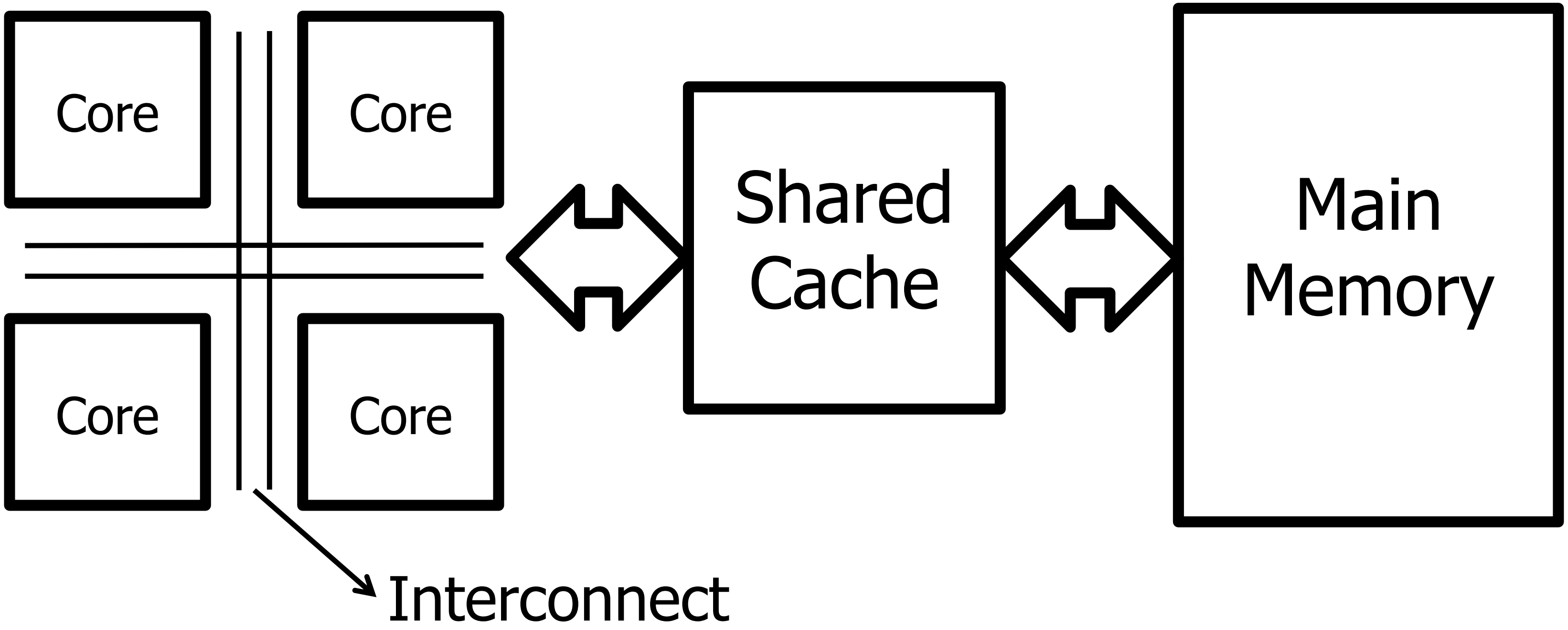
James Hoe

Ravi Iyer (Intel)

The Multicore Era



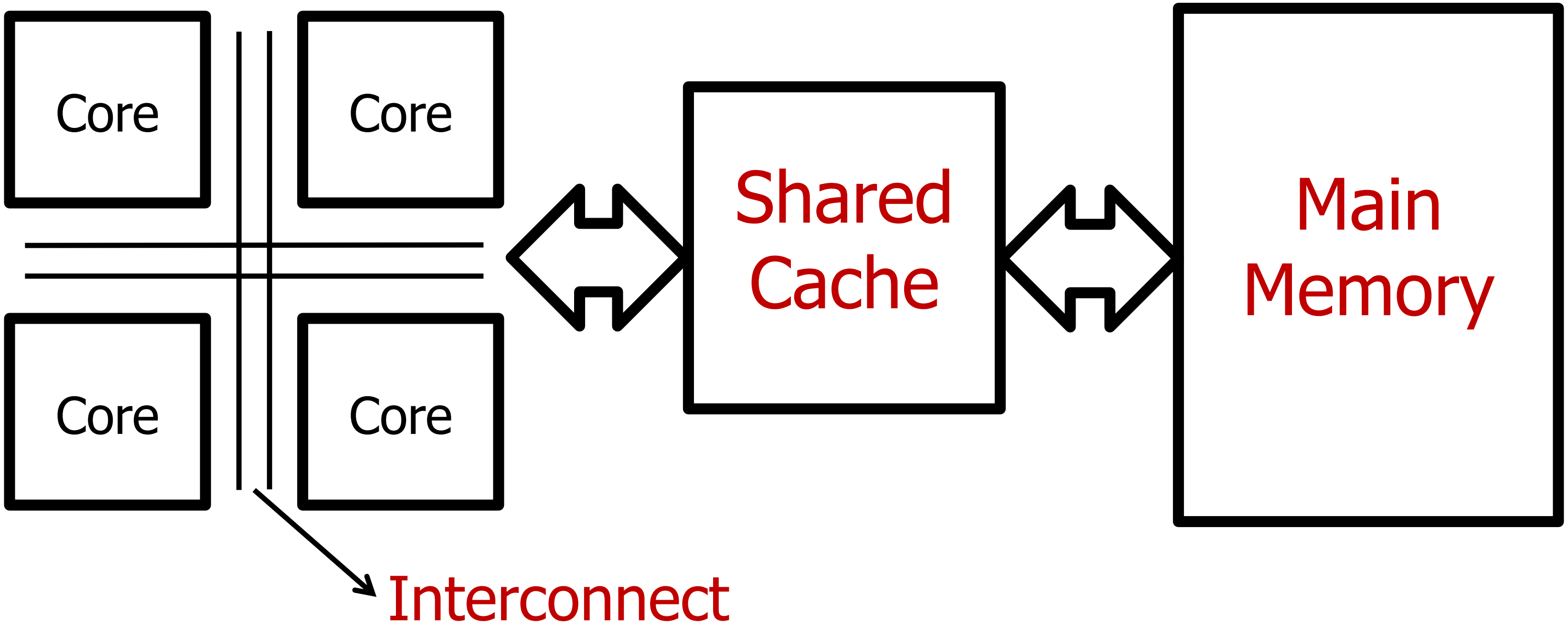
The Multicore Era



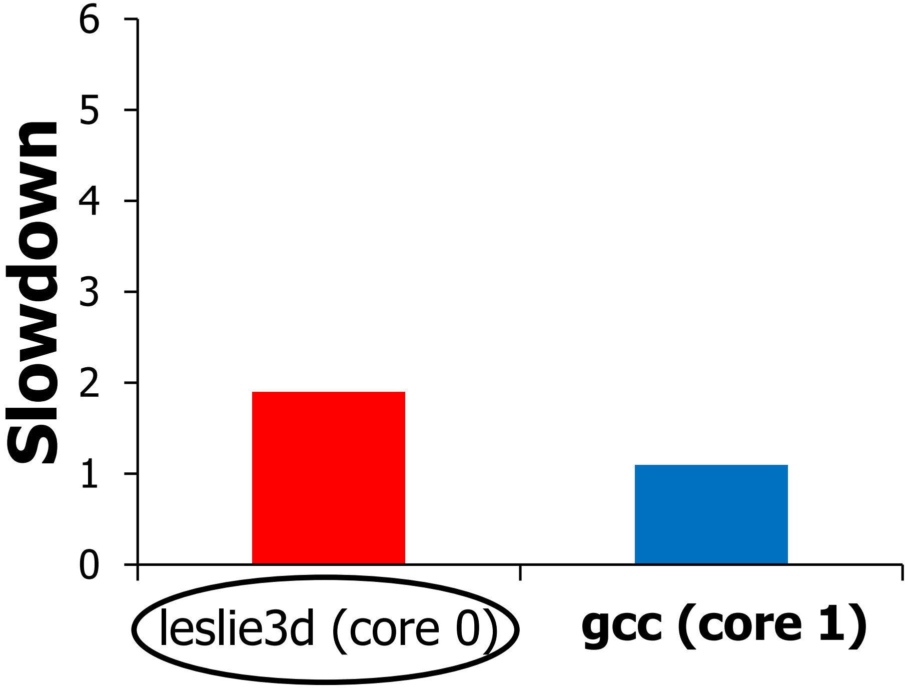
Multiple applications execute in parallel

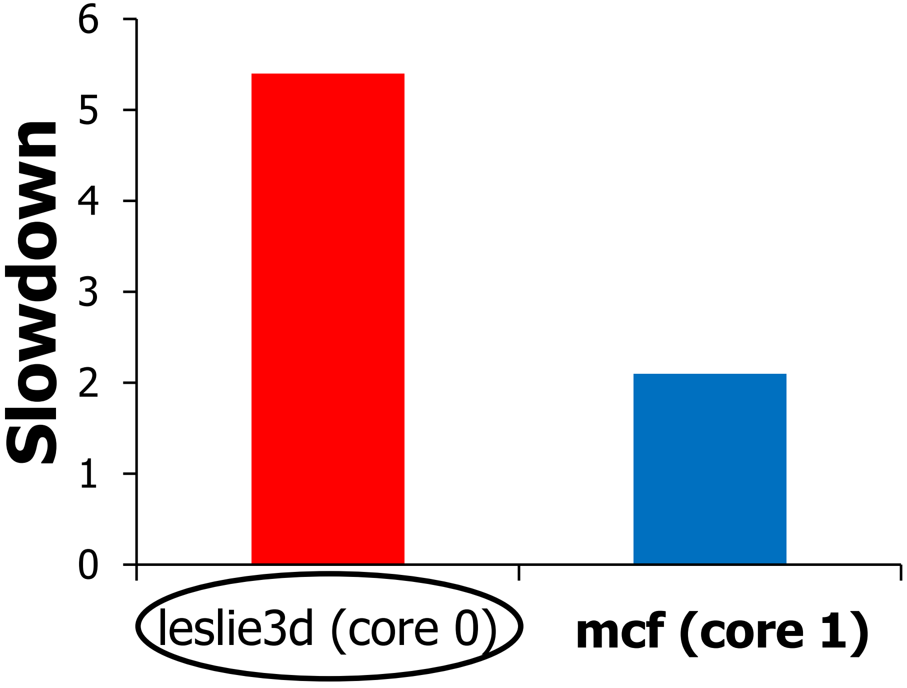
High throughput and efficiency

Challenge:
Interference at Shared Resources



Impact of Shared Resource Interference





1. High application slowdowns

2. Unpredictable application slowdowns

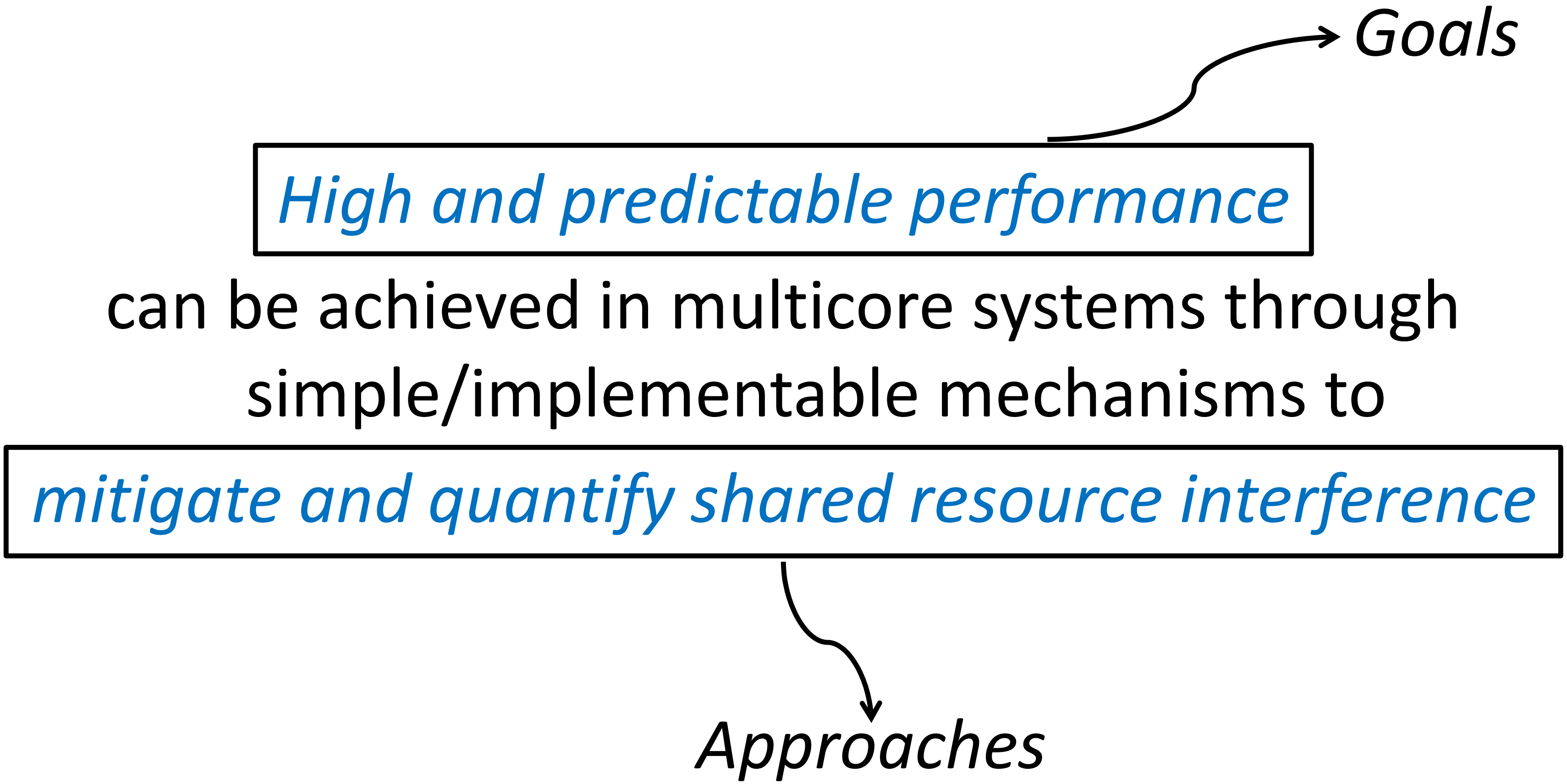
Why Predictable Performance?

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

- Example 1: In server systems
  - Different users’ jobs consolidated onto the same server
  - Need to provide bounded slowdowns to critical jobs

- Example 2: In mobile systems
  - Interactive applications run with non-interactive applications
  - Need to guarantee performance for interactive applications

Thesis Statement



Goals

High and predictable performance

can be achieved in multicore systems through simple/implementable mechanisms to

mitigate and quantify shared resource interference

Approaches

Goals and Approaches

Goals:

1. High Performance

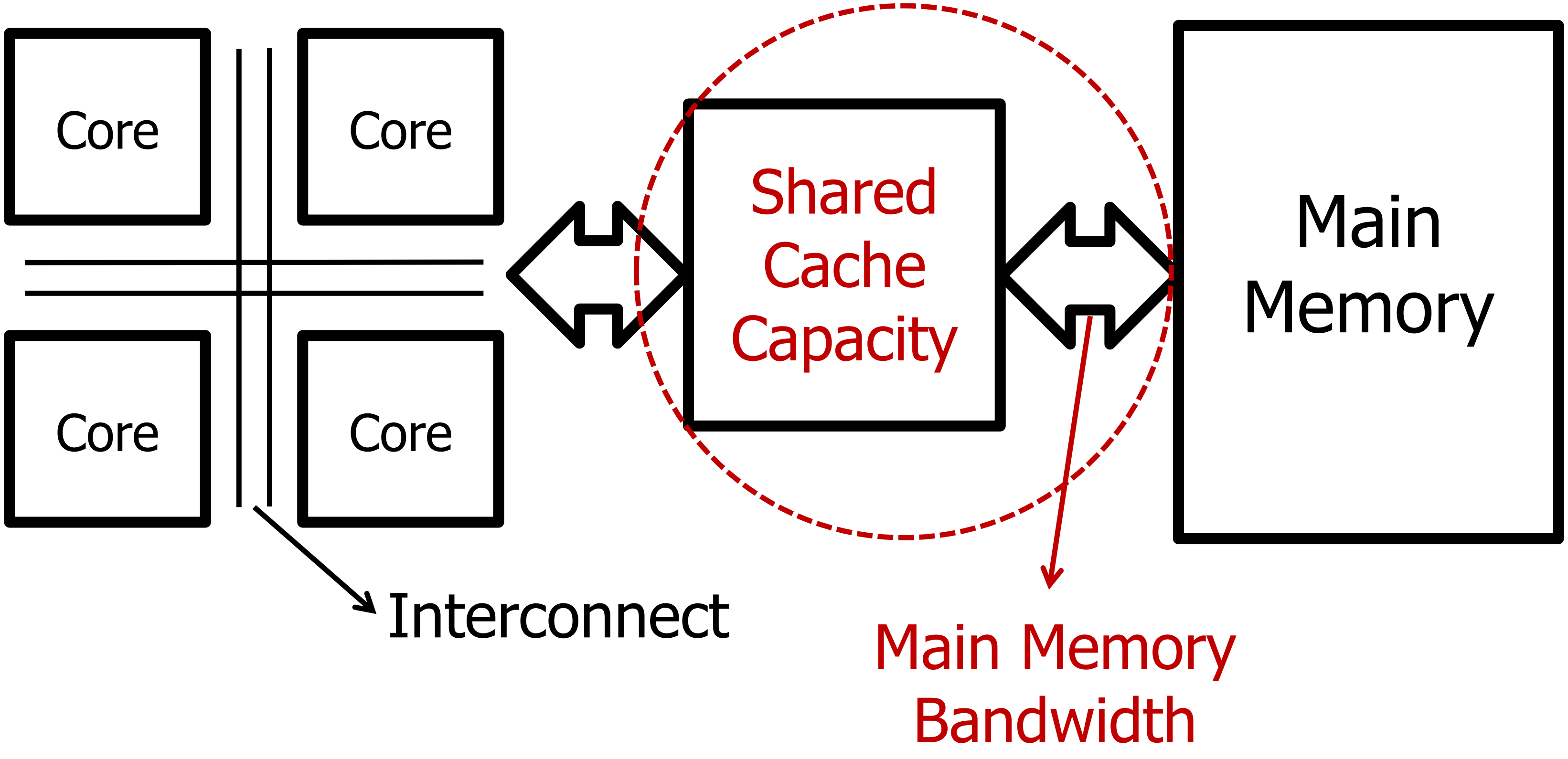
2. Predictable Performance

Approaches:

Mitigate Interference

Quantify Interference

Focus Shared Resources in This Thesis



Related Prior Work

	Mitigate Interference	*Quantify Interference*
Cache Capacity	**Much explored** ***Not our focus***	***Not our focus***
Memory Bandwidth	STFM (MICRO ’07), PARBS (ISCA ’08), ATLAS (HPCA ’10), TCM (MICRO ’11), Criticality-aware (ISCA ‘’13) ***Challenge: High complexity***	*STFM (MICRO ’07)* ***Challenge: High inaccuracy***

FST (ASPLOS ’10),
PTCA (TACO ’13)

Challenge: High inaccuracy

Outline

	Mitigate Interference	*Quantify Interference*
Cache Capacity	Much explored *Not our focus*	*Not our focus*
Memory Bandwidth	***Blacklisting Memory Scheduler***	***Memory Interference induced Slowdown Estimation Model and its uses***

Application Slowdown Model and its uses

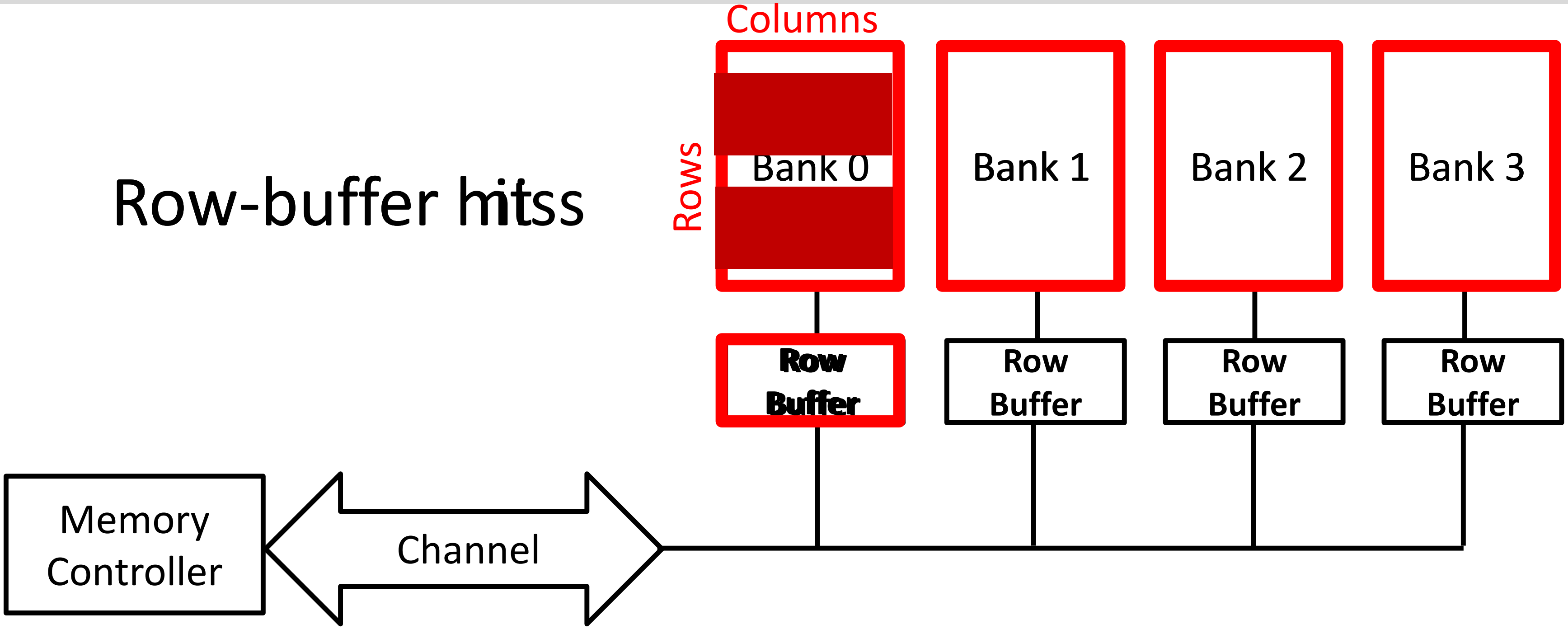
Outline

	Mitigate Interference	*Quantify Interference*
Cache Capacity	Much explored *Not our focus*	*Not our focus*
Memory Bandwidth	***Blacklisting Memory Scheduler***	***Memory Interference induced Slowdown Estimation Model and its uses***

Application Slowdown Model and its uses

Background: Main Memory

Row-buffer hitss

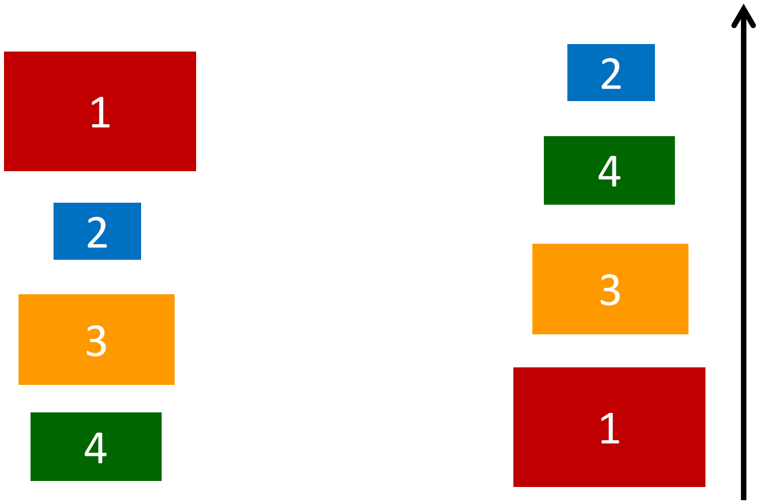


- FR-FCFS Memory Scheduler [Zuravleff and Robinson, US Patent '97; Rixner et al., ISCA '00]
  - – Row-buffer hit first
  - – Older request first
- Unaware of inter-application interference

Tackling Inter-Application Interference: Application-aware Memory Scheduling

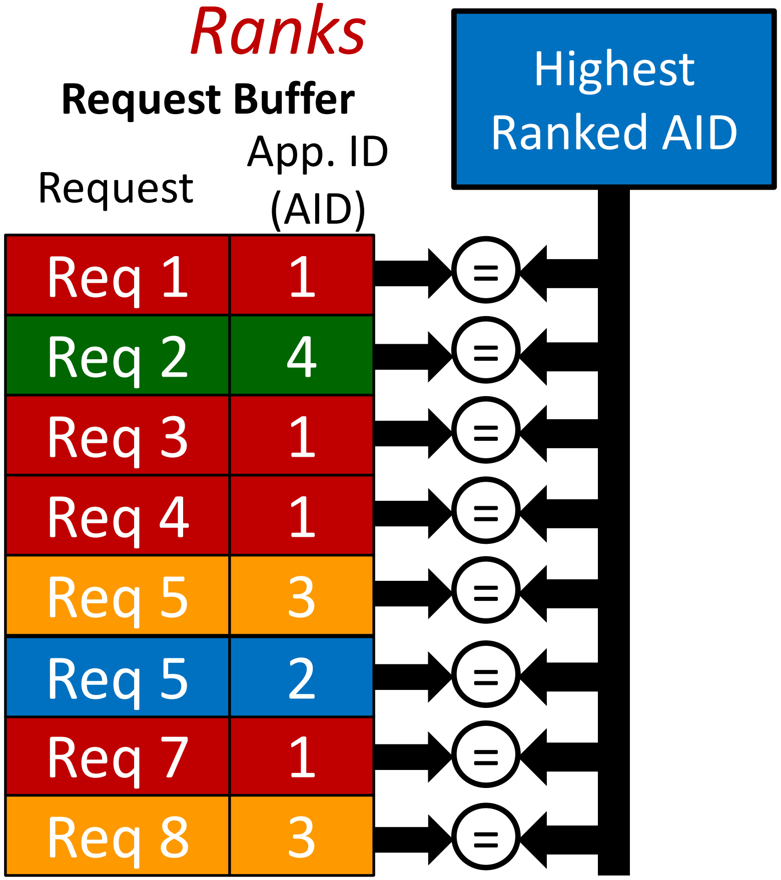
Monitor

Rank



Enforce Ranks

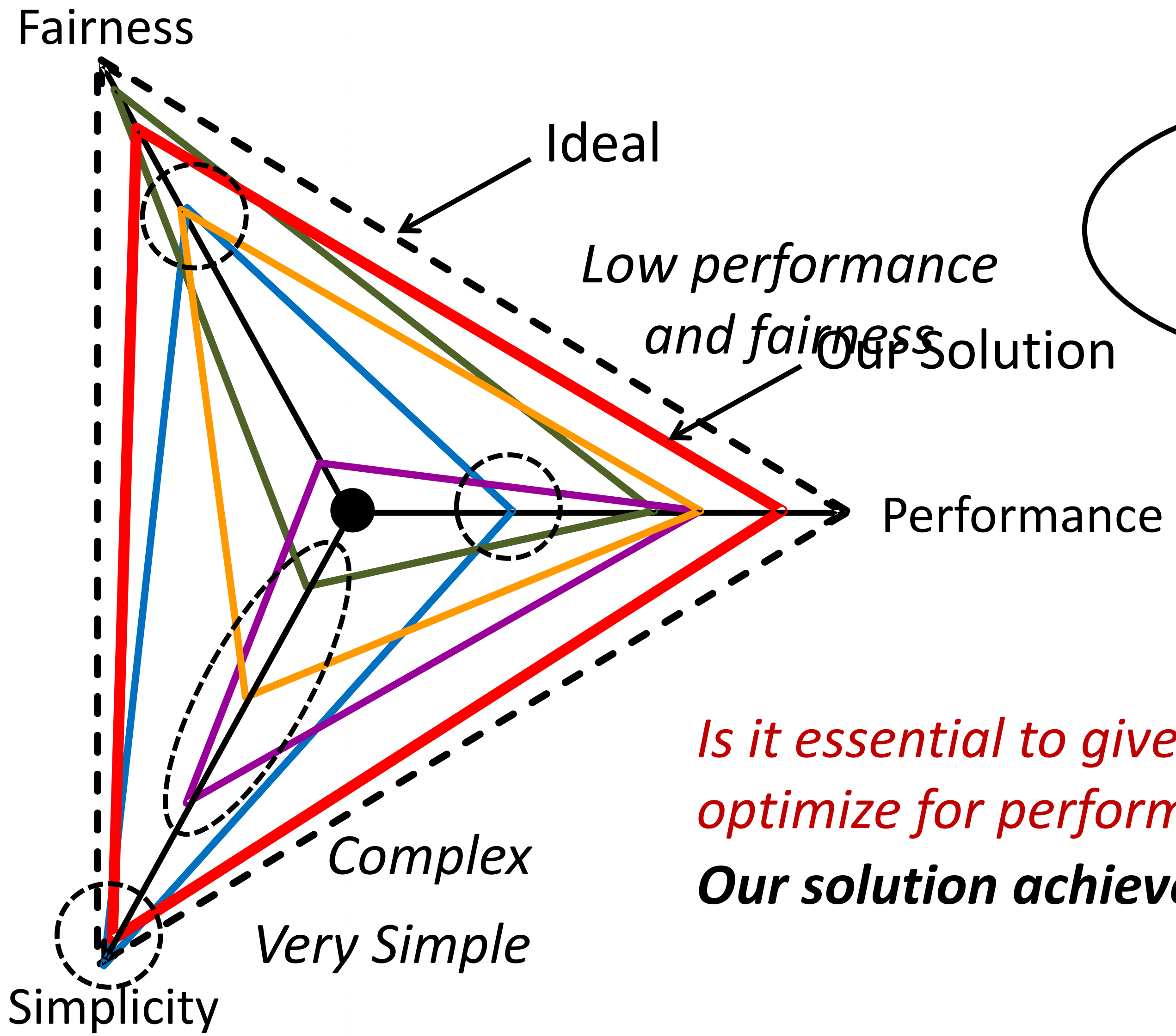
Request Buffer



Request	App. ID (AID)
Req 1	1
Req 2	4
Req 3	1
Req 4	1
Req 5	3
Req 5	2
Req 7	1
Req 8	3

Full ranking increases critical path latency and area significantly to improve performance and fairness

Performance vs. Fairness vs. Simplicity



Is it essential to give up simplicity to optimize for performance and/or fairness?

Our solution achieves all three goals

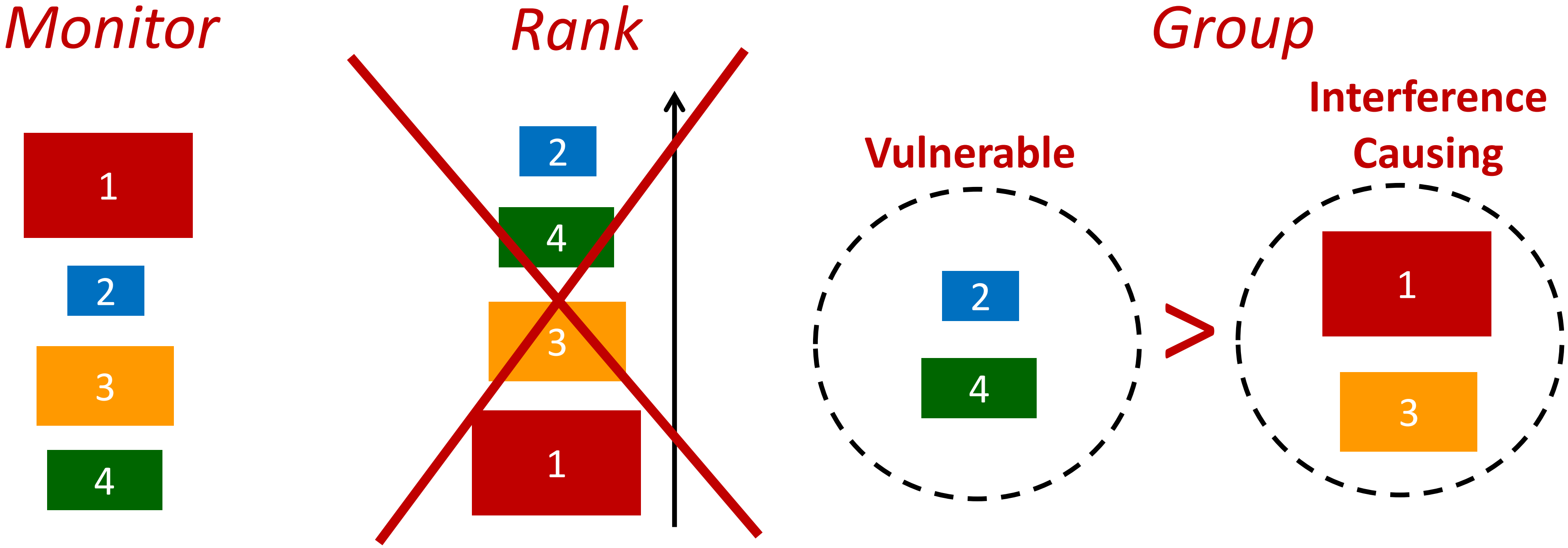
Problems with Previous Application-aware Memory Schedulers

1. Full ranking increases hardware complexity
2. Full ranking causes unfair slowdowns

Our Goal: Design a memory scheduler with
Low Complexity, High Performance, and Fairness

Key Observation 1: Group Rather Than Rank

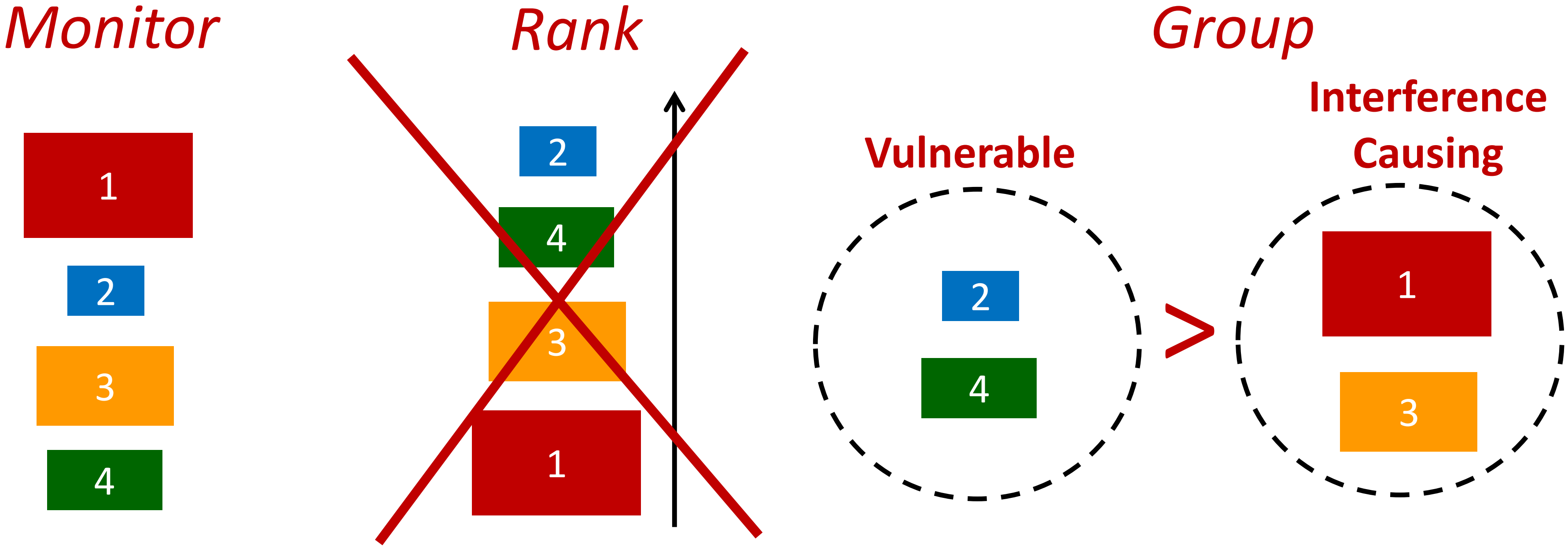
Observation 1: Sufficient to separate applications into two groups, rather than do full ranking



Benefit 2: Lower slowdowns than ranking

Key Observation 1: Group Rather Than Rank

Observation 1: Sufficient to separate applications into two groups, rather than do full ranking



How to classify applications into groups?

Key Observation 2

Observation 2: Serving a large number of consecutive requests from an application causes interference

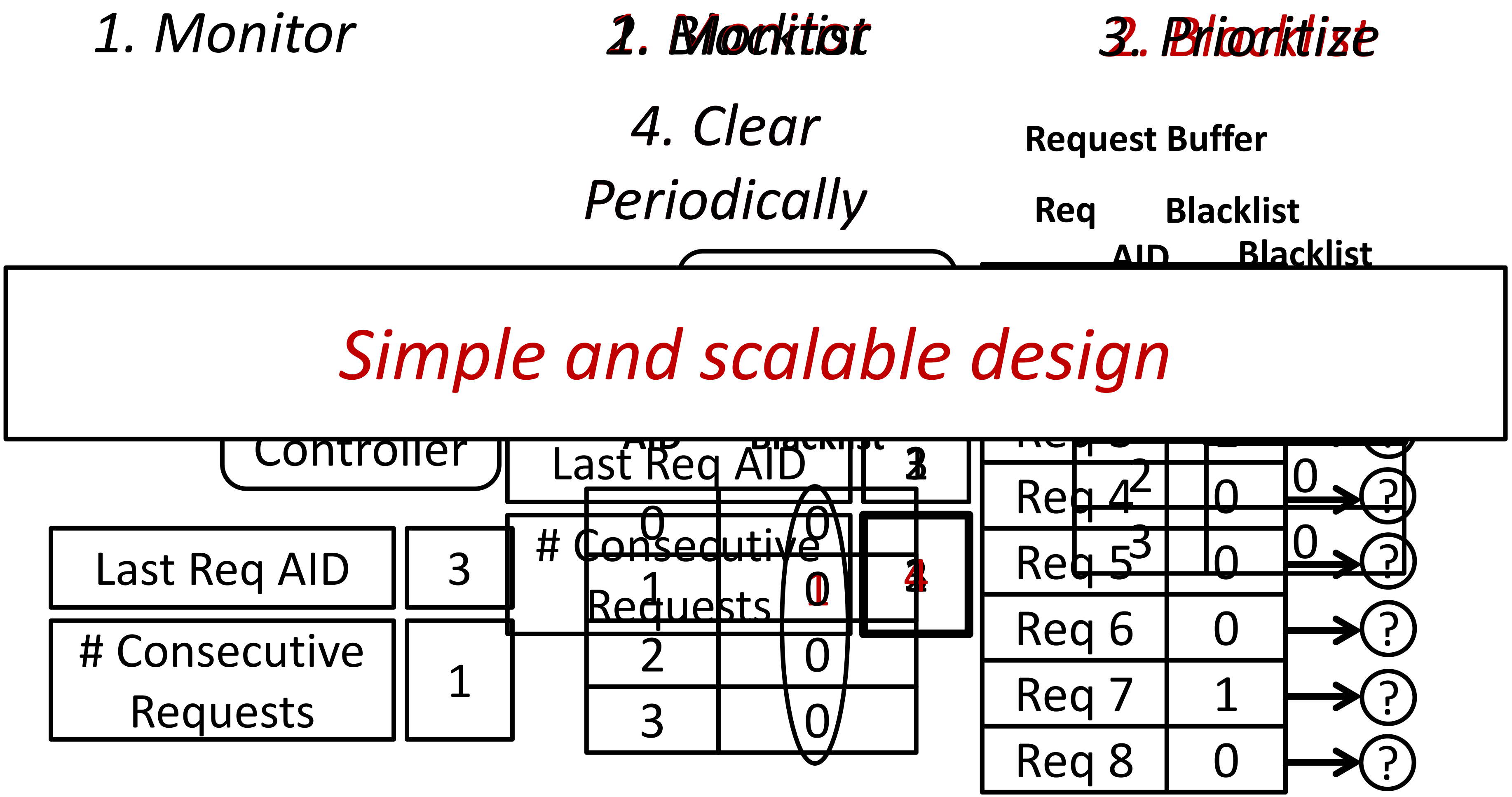
Basic Idea:

- *Group* applications with a large number of consecutive requests as *interference-causing* → *Blacklisting*
- *Deprioritize* blacklisted applications
- *Clear* blacklist periodically (1000s of cycles)

Benefits:

- *Lower complexity*
- *Finer grained grouping decisions* → *Lower unfairness*

The Blacklisting Memory Scheduler (ICCD '14)



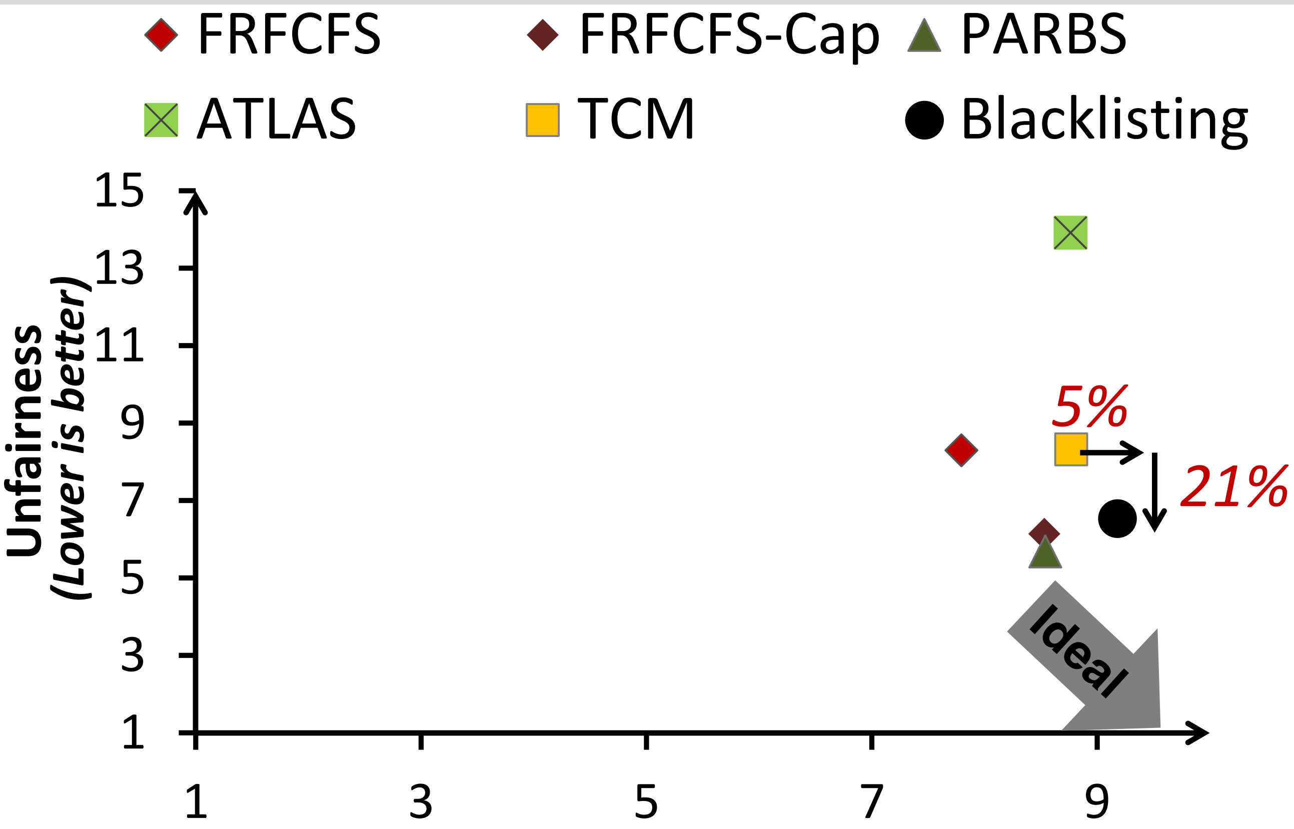
Simple and scalable design

Methodology

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

- Workloads
  - SPEC CPU2006, TPC-C, Matlab , NAS
  - 80 multiprogrammed workloads

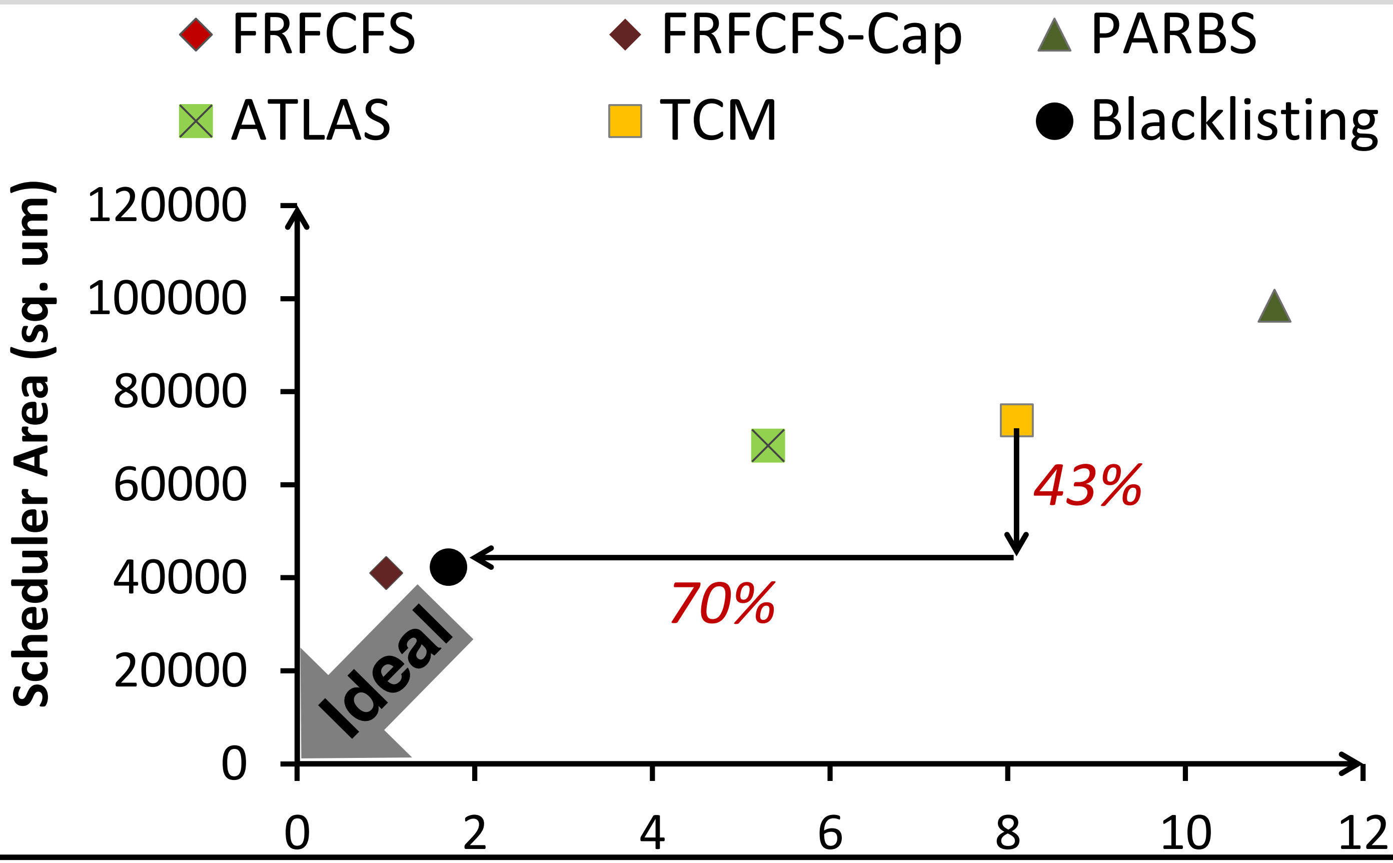
Performance and Fairness



1. Blacklisting achieves the highest performance

2. Blacklisting balances performance and fairness

Complexity



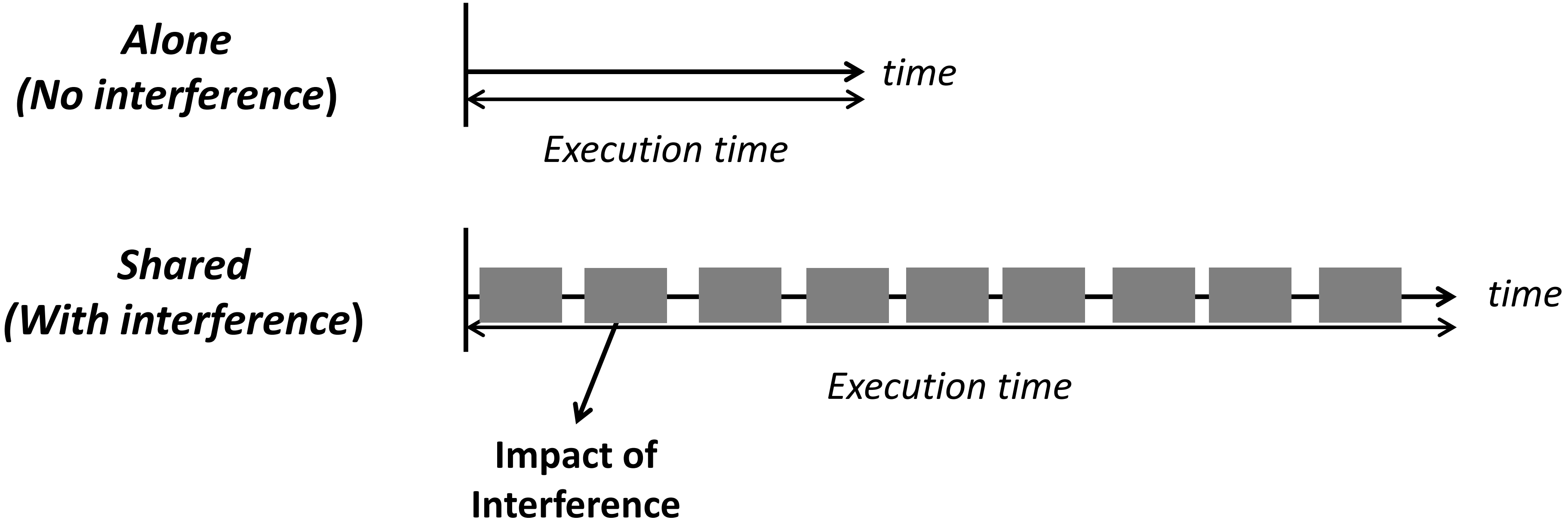
Blacklisting reduces complexity significantly

Outline

	Mitigate Interference	***Quantify Interference***
Cache Capacity	**Much explored** ***Not our focus***	***Not our focus***
Memory Bandwidth	✓ ***Blacklisting Memory Scheduler***	***Memory Interference induced Slowdown Estimation Model and its uses***

Application Slowdown Model and its uses

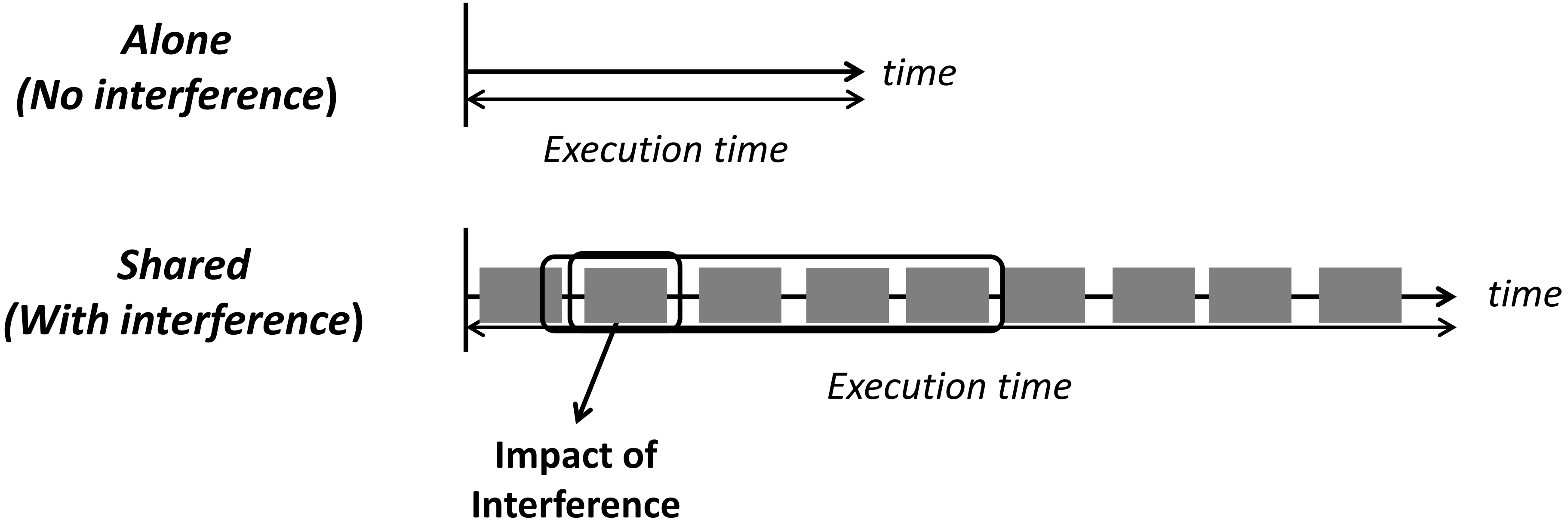
Impact of Interference on Performance



Slowdown: Definition

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

Impact of Interference on Performance



Previous Approach: Estimate impact of interference at a per-request granularity

Difficult to estimate due to request overlap

Outline

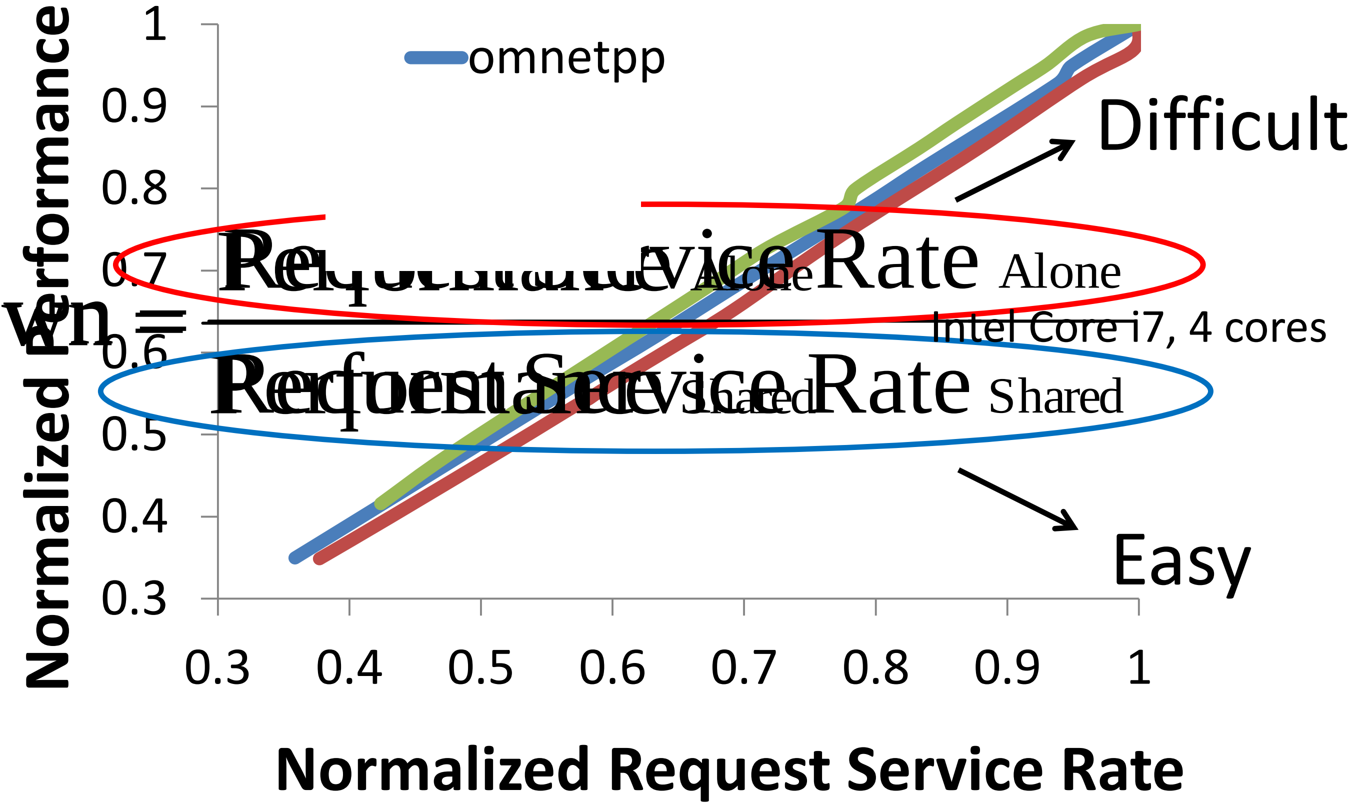
	Mitigate Interference	*Quantify Interference*
Cache Capacity	Much explored *Not our focus*	*Not our focus*
Memory Bandwidth	✓ ***Blacklisting Memory Scheduler***	***Memory Interference induced Slowdown Estimation Model and its uses***

Application Slowdown Model and its uses

Observation: Request Service Rate is a Proxy for Performance

For a memory bound application,

Performance ∝ Memory request service rate



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

Observation: Highest Priority Enables Request Service Rate $_{\text{Alone}}$ Estimation

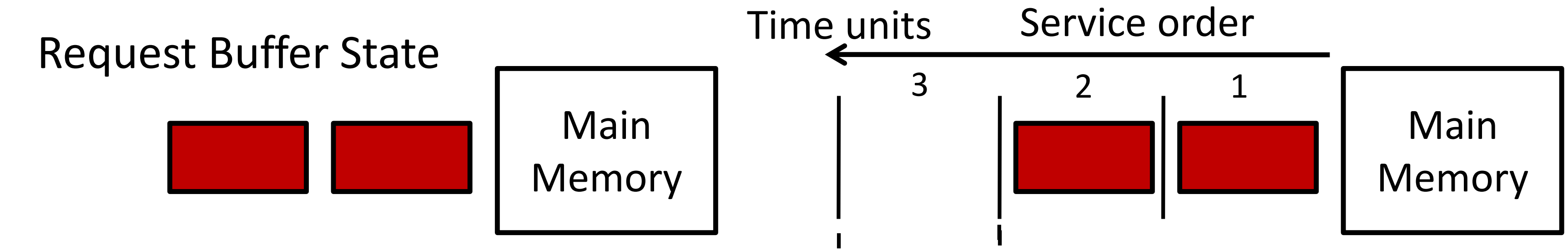
Request Service Rate $_{\text{Alone}}$ ($\text{RSR}_{\text{Alone}}$) of an application can be estimated by giving the application highest priority at the *memory controller*

Highest priority → Little interference

(almost as if the application were run alone)

Observation: Highest Priority Enables Request Service Rate$_{\text{Alone}}$ Estimation

1. Run alone



Request Buffer State

Main Memory

Time units

Service order

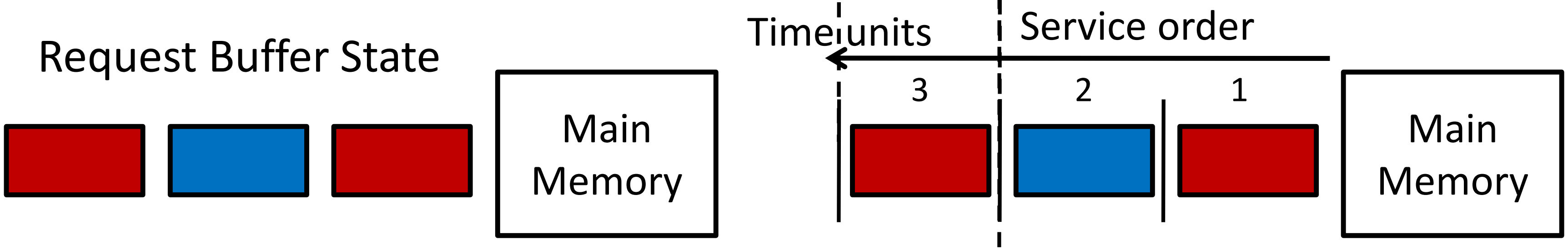
3

2

1

Main Memory

2. Run with another application



Request Buffer State

Main Memory

Time units

Service order

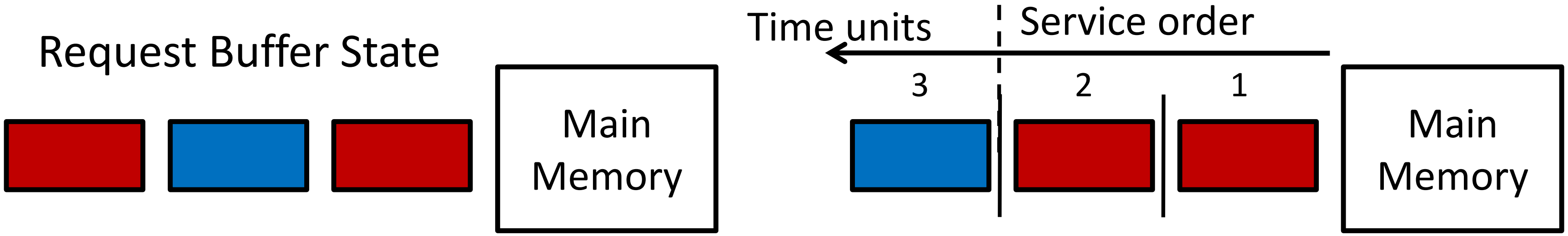
3

2

1

Main Memory

3. Run with another application: highest priority



Request Buffer State

Main Memory

Time units

Service order

3

2

1

Main Memory

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

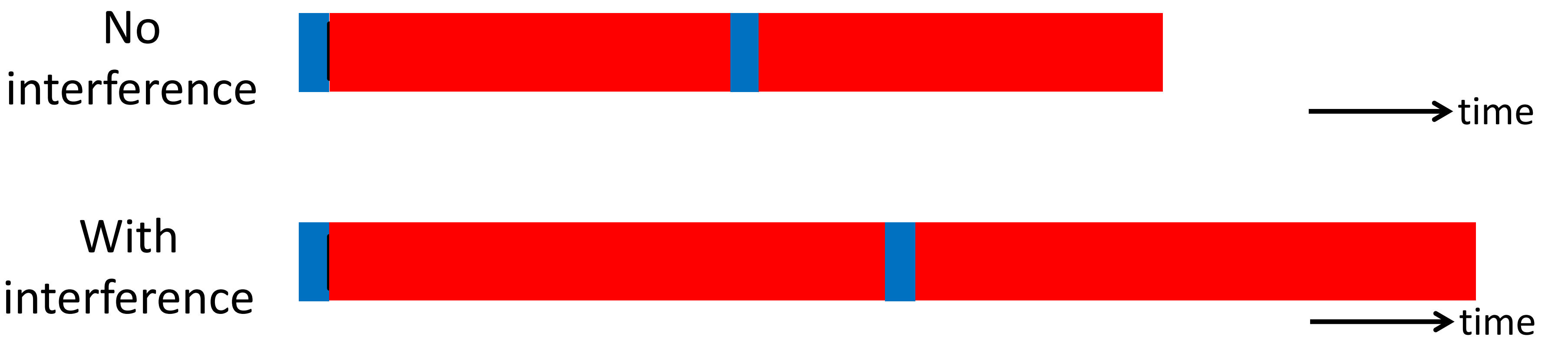
$$\text{Slowdown} = \frac{\text{Request Service Rate}_{\text{Alone}}\ (\text{RSR}_{\text{Alone}})}{\text{Request Service Rate}_{\text{Shared}}\ (\text{RSR}_{\text{Shared}})}$$

Observation: Memory Bound vs. Non-Memory Bound

- Memory-bound application

🟦 Compute Phase

🟥 Memory Phase



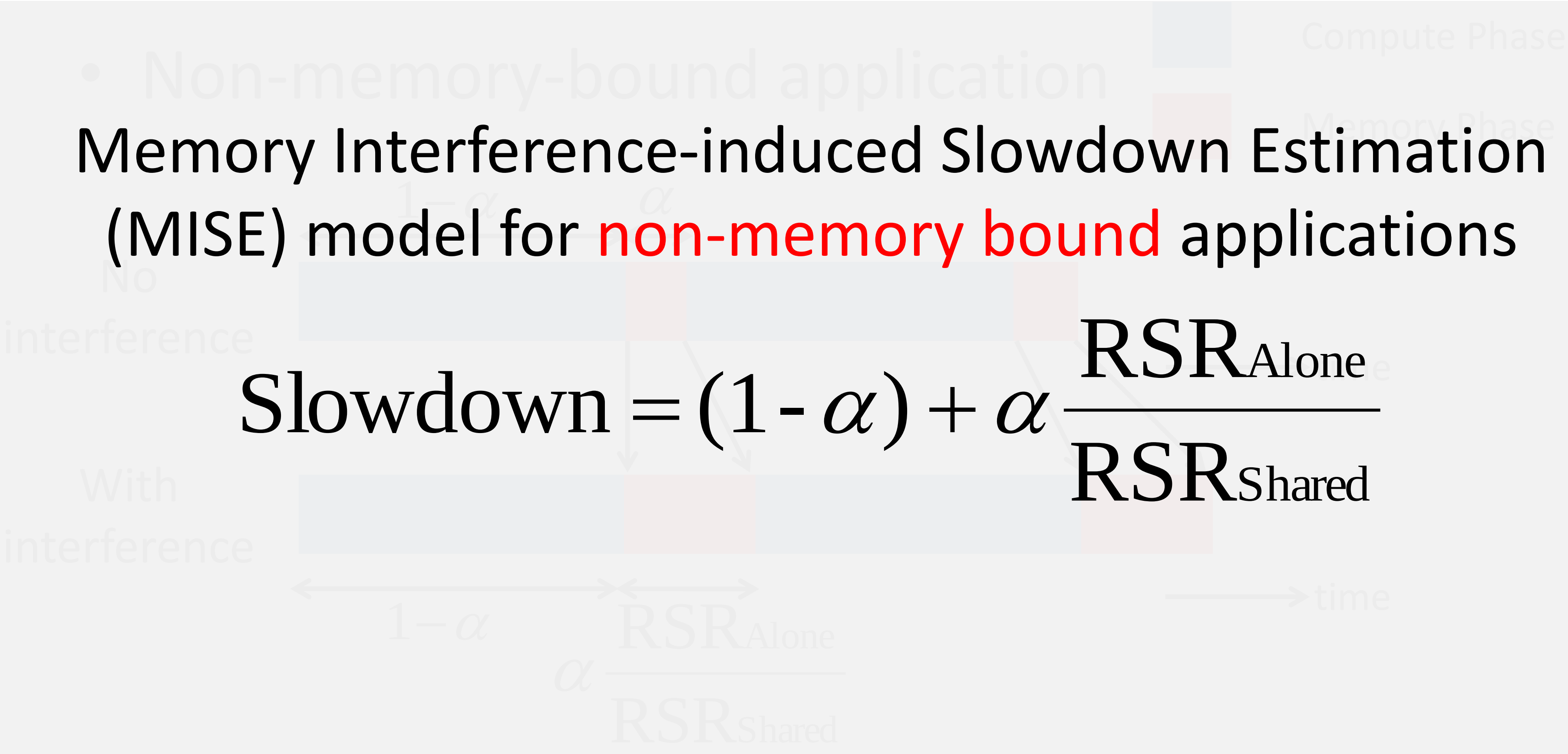
Memory phase slowdown dominates overall slowdown

Observation: Memory Bound vs. Non-Memory Bound

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

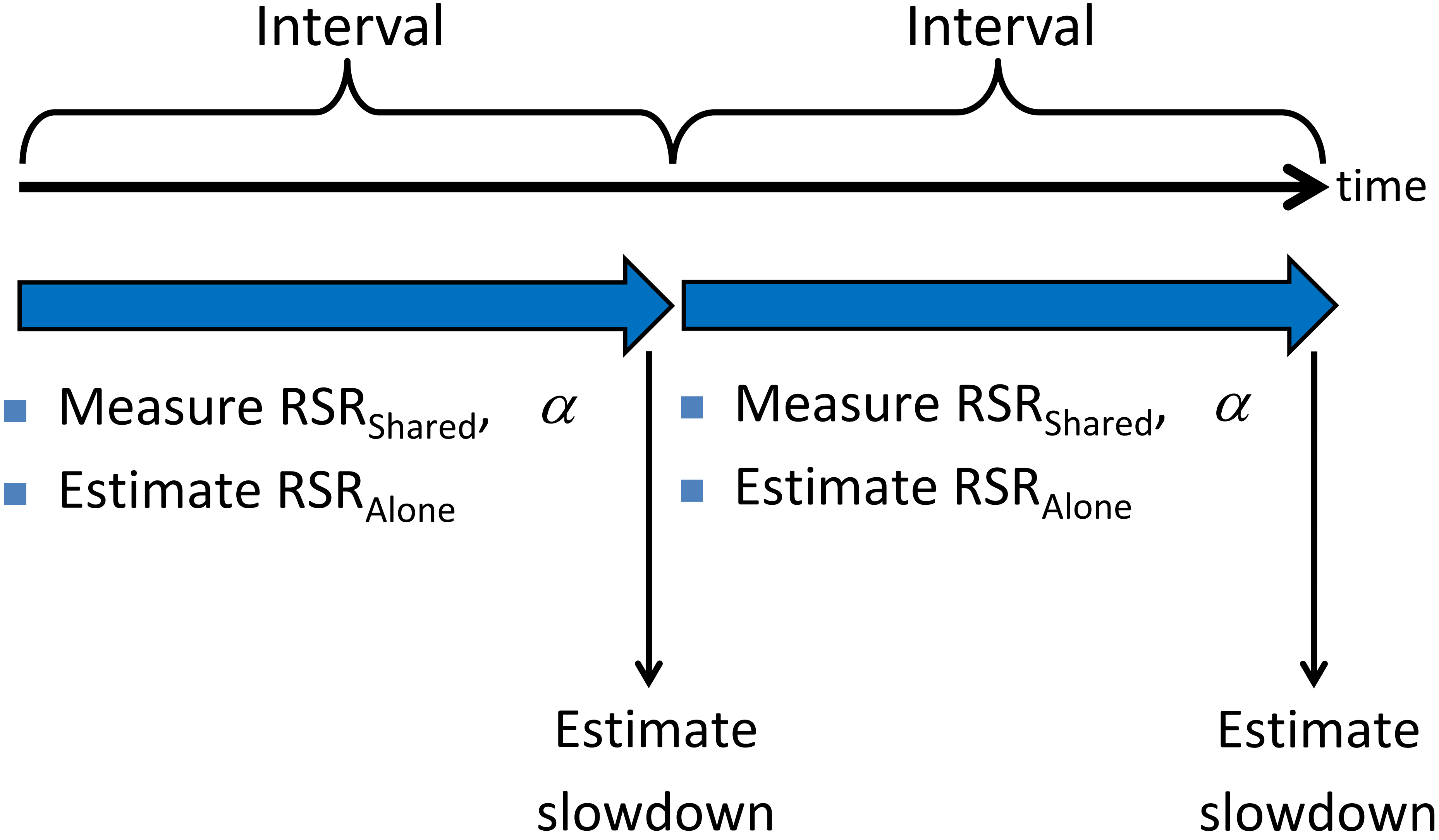
$$\text{Slowdown} = (1-\alpha) + \alpha \frac{\text{RSR}_{\text{Alone}}}{\text{RSR}_{\text{Shared}}}$$

- Non-memory-bound application



Only memory fraction (α) slows down with interference

Interval Based Operation



Interval

Interval

time

- 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 et al., MICRO '07]
  - **FST** (Fairness via Source Throttling) [Ebrahimi et al., ASPLOS '10]
  - **Per-thread Cycle Accounting** [Du Bois et al., HiPEAC '13]

- Basic Idea:

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

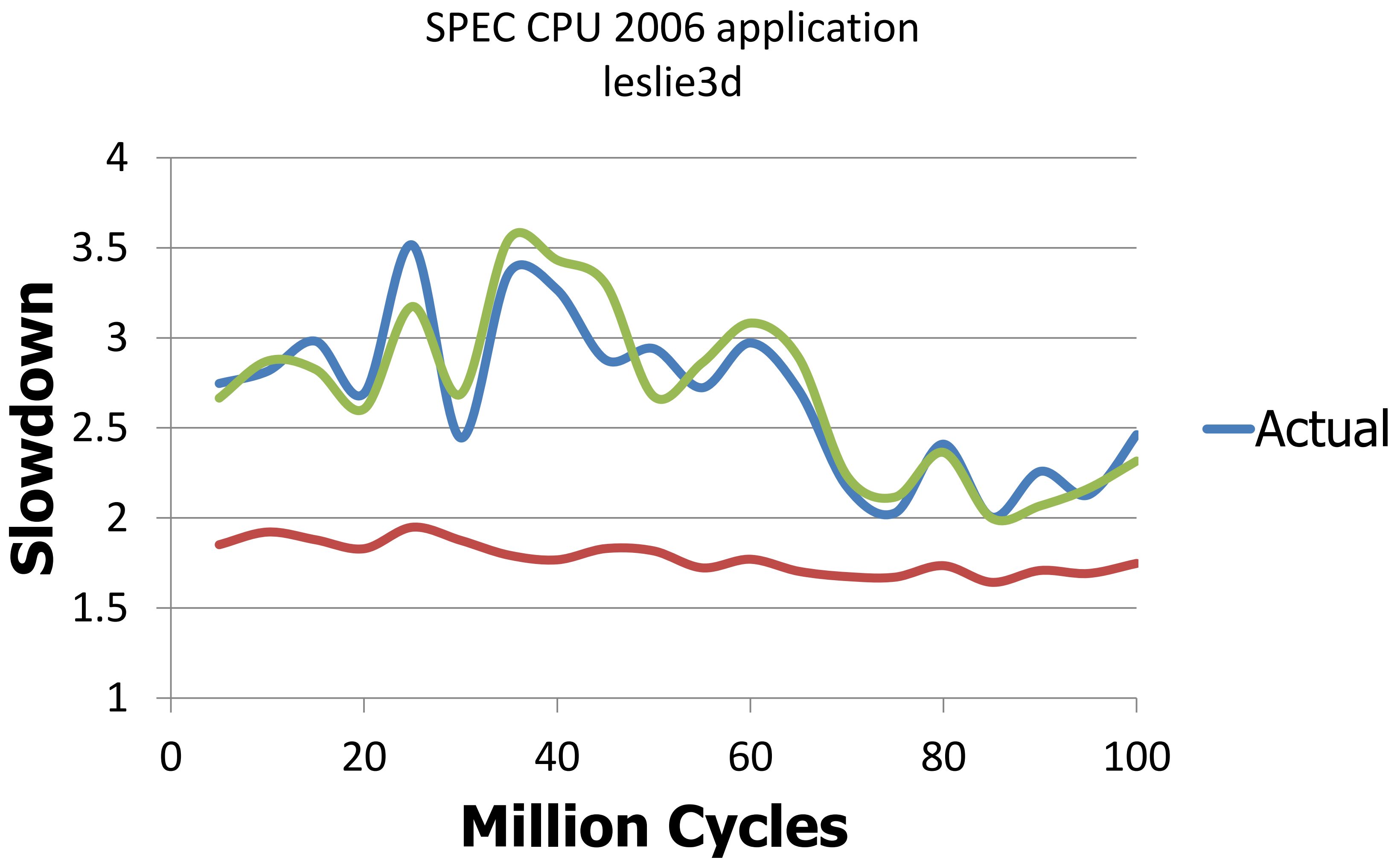
Count number of cycles application receives interference

Methodology

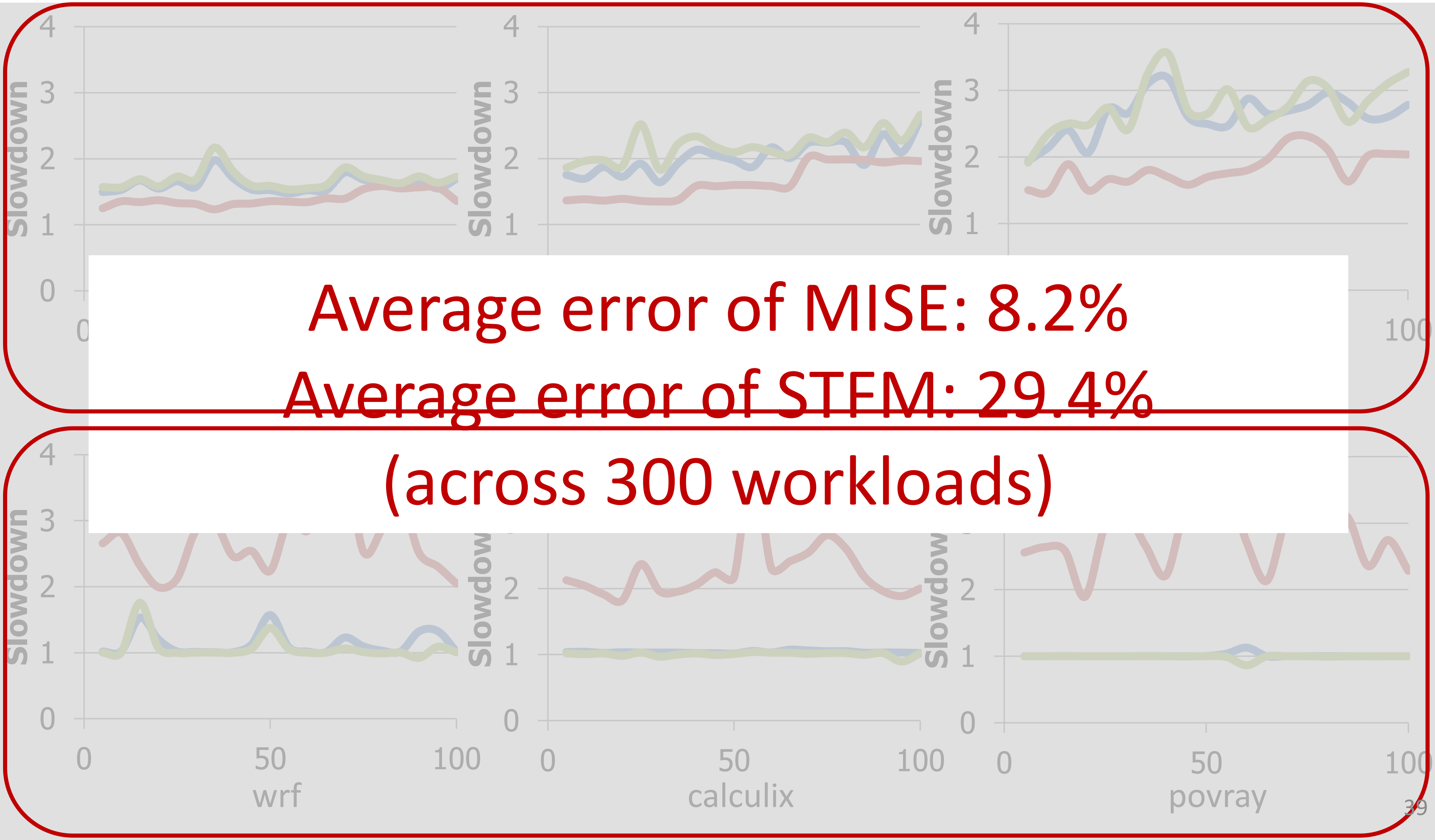
- Configuration of our simulated system
  - 4 cores
  - 1 channel, 8 banks/channel
  - DDR3 1066 DRAM
  - 512 KB private cache/core

- 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)

Possible Use Cases of the MISE Model

- *Bounding application slowdowns* ***[HPCA ’13]***

- *Achieving high system fairness and performance* ***[HPCA ’13]***

- *VM migration and admission control schemes* ***[VEE ’15]***

- *Fair billing schemes in a commodity cloud*

MISE-QoS: 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

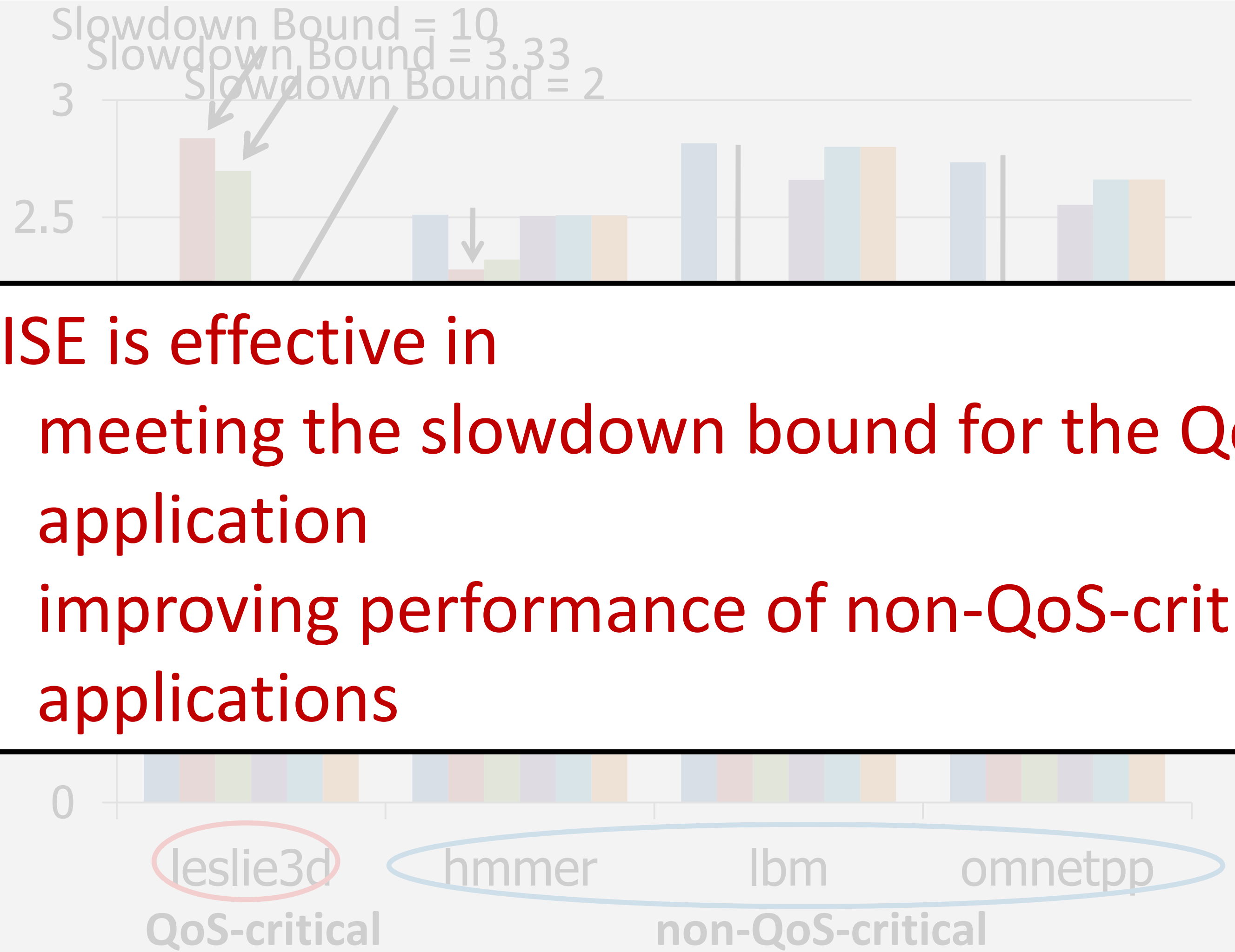
Methodology

- 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 multi programmed workloads
- Each workload run with 10 slowdown bound values
- Baseline memory scheduling mechanism
  - Always prioritize QoS-critical application

    [Iyer et al., SIGMETRICS 2007]
  - Other applications' requests scheduled in FR-FCFS order

    [Zuravleff and Robinson, 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

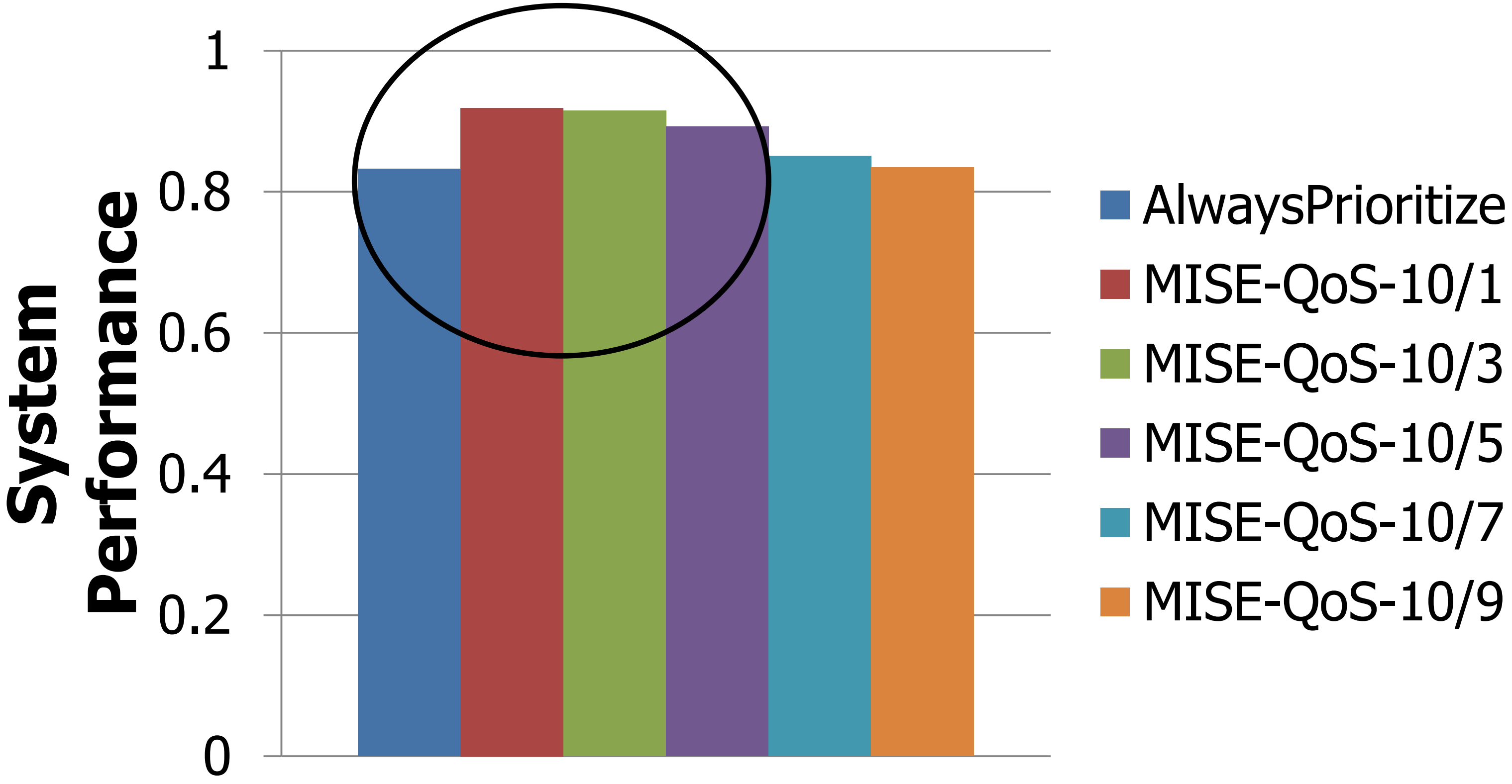
Effectiveness of MISE in Enforcing QoS

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



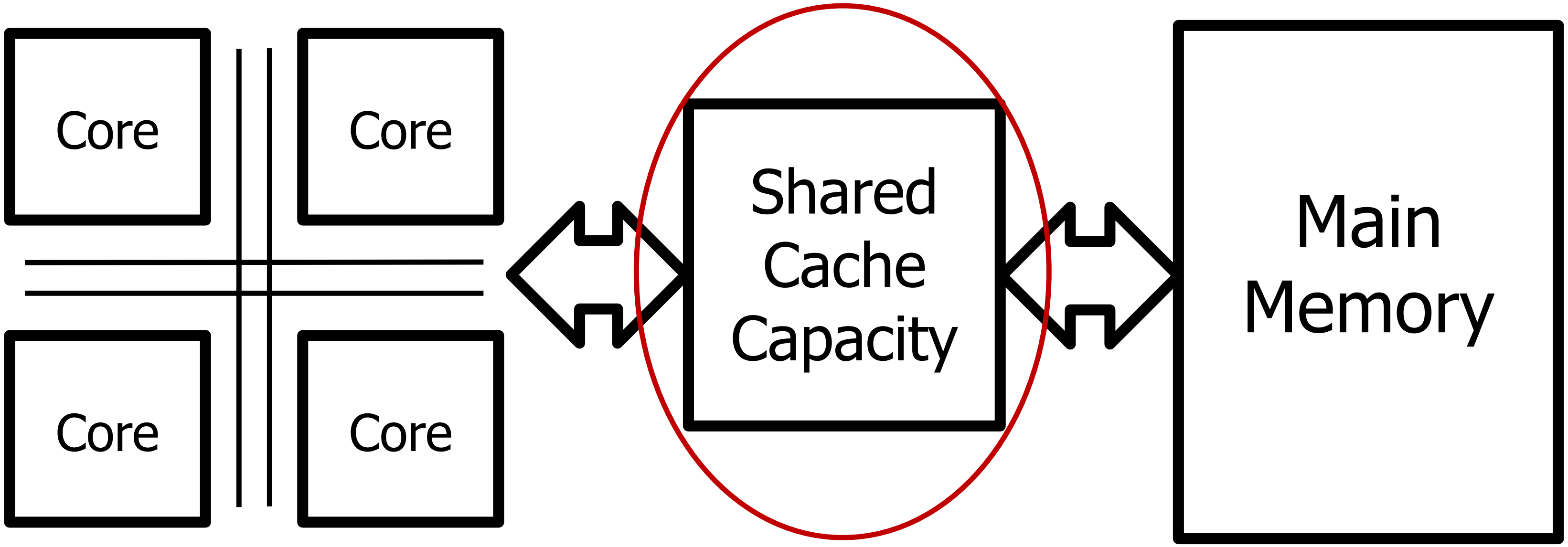
When slowdown bound is 10/3
MISE-QoS improves system performance by 10%

Outline

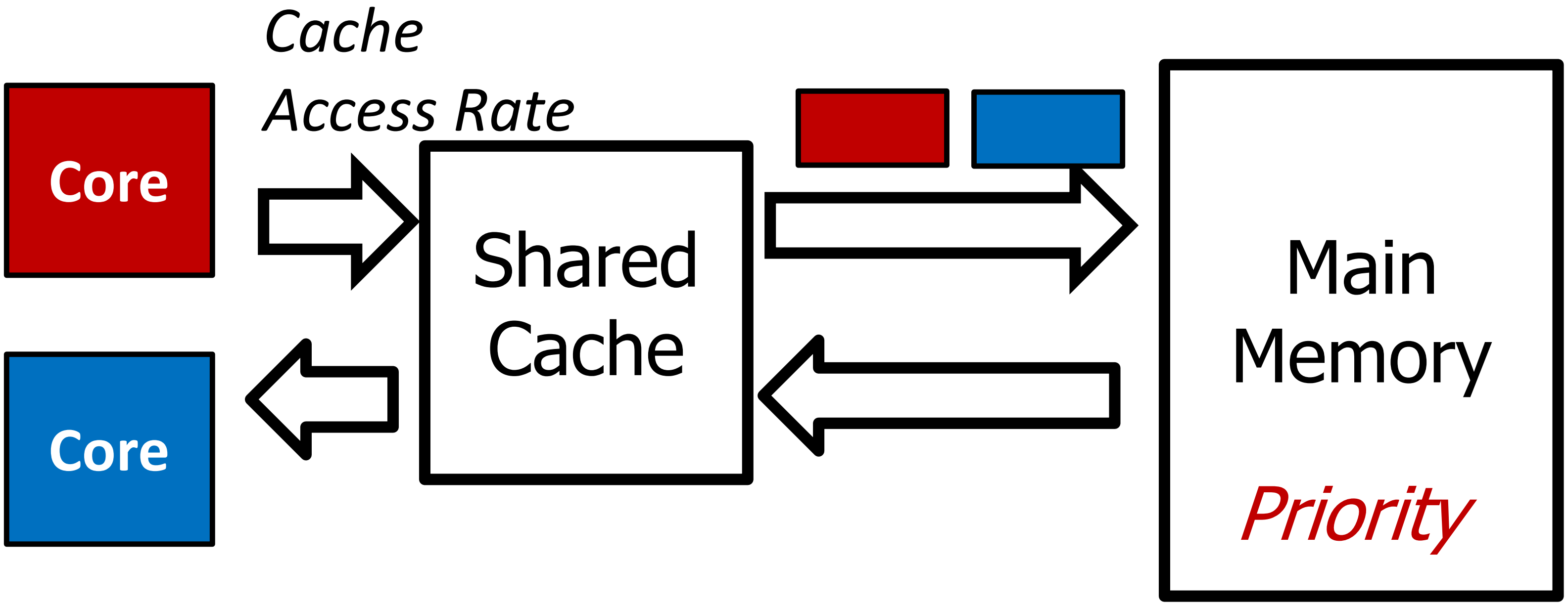
	Mitigate Interference	*Quantify Interference*
Cache Capacity	**Much explored** ***Not our focus***	***Not our focus***
Memory Bandwidth	✓***Blacklisting Memory Scheduler***	✓***Memory Interference induced Slowdown Estimation Model and its uses***

Application Slowdown Model and its uses

Shared Cache Capacity Contention



Cache Capacity Contention



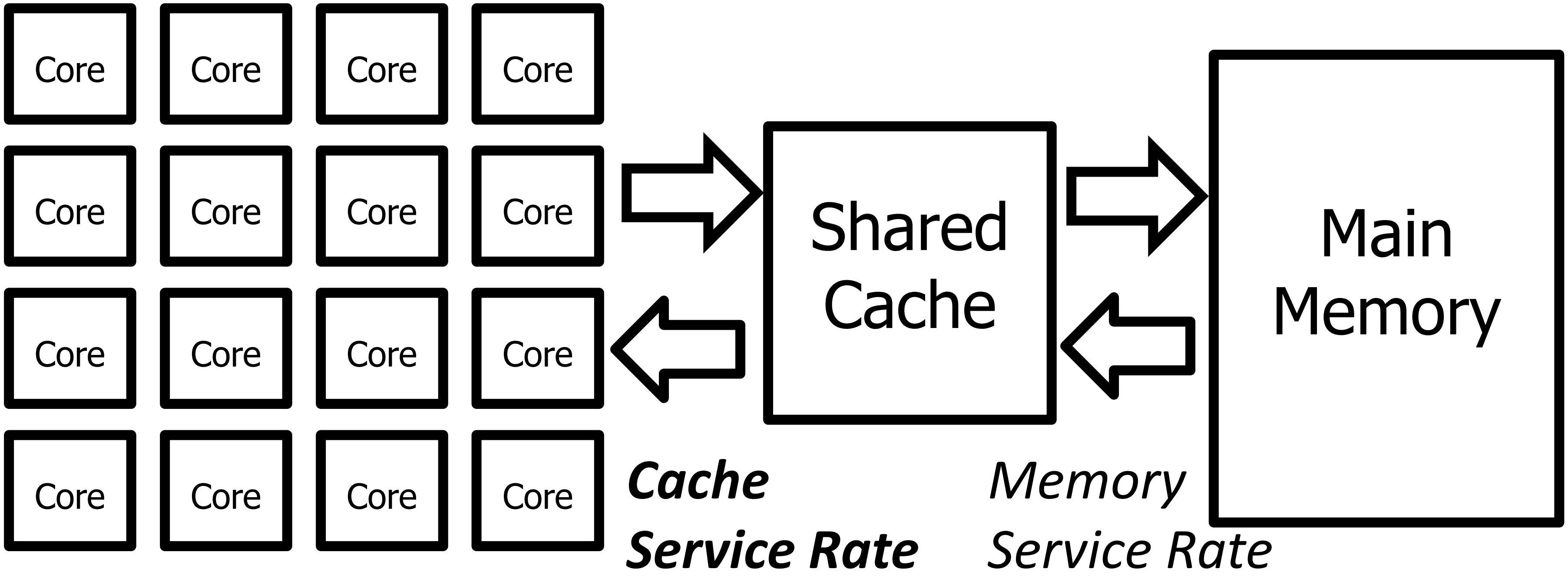
Applications evict each other's blocks from the shared cache

Outline

	Mitigate Interference	*Quantify Interference*
Cache Capacity	Much explored *Not our focus*	*Not our focus*
Memory Bandwidth	✓ ***Blacklisting Memory Scheduler***	✓ ***Memory Interference induced Slowdown Estimation Model and its uses***

Application Slowdown Model and its uses

Estimating Cache and Memory Slowdowns



Core Core Core Core

Core Core Core Core

Core Core Core Core

Core Core Core Core

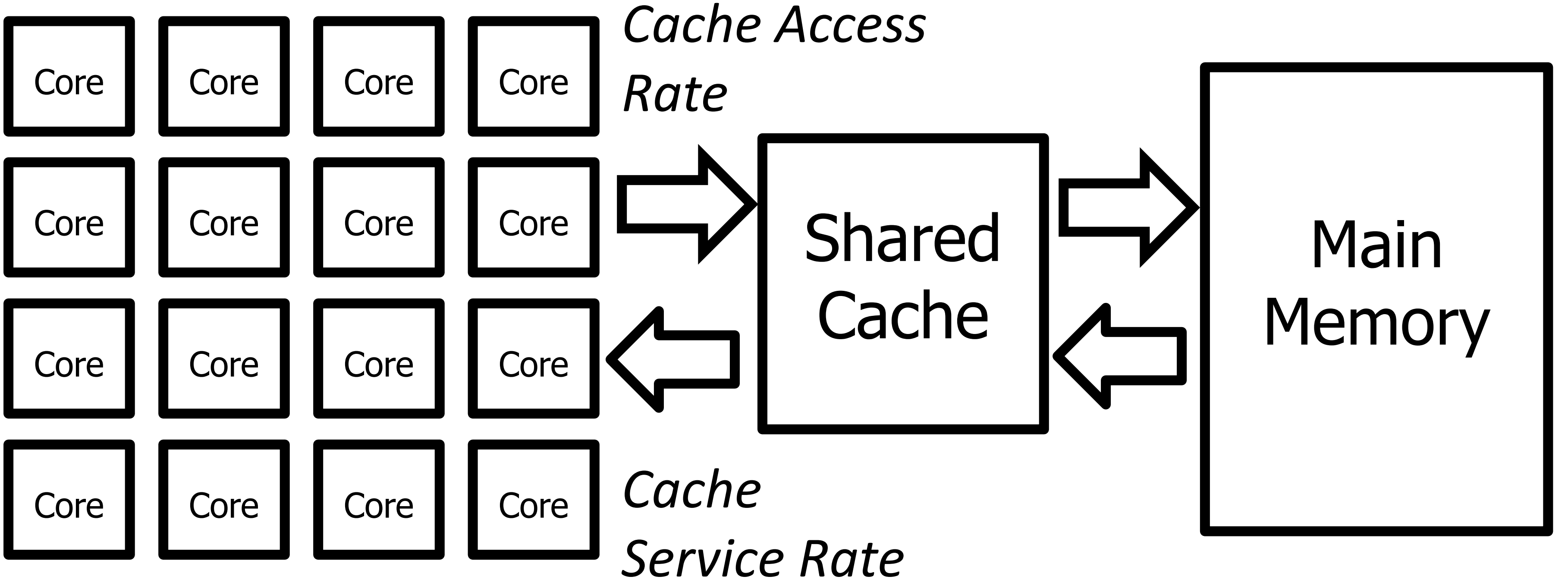
Shared Cache

Main Memory

Cache Service Rate

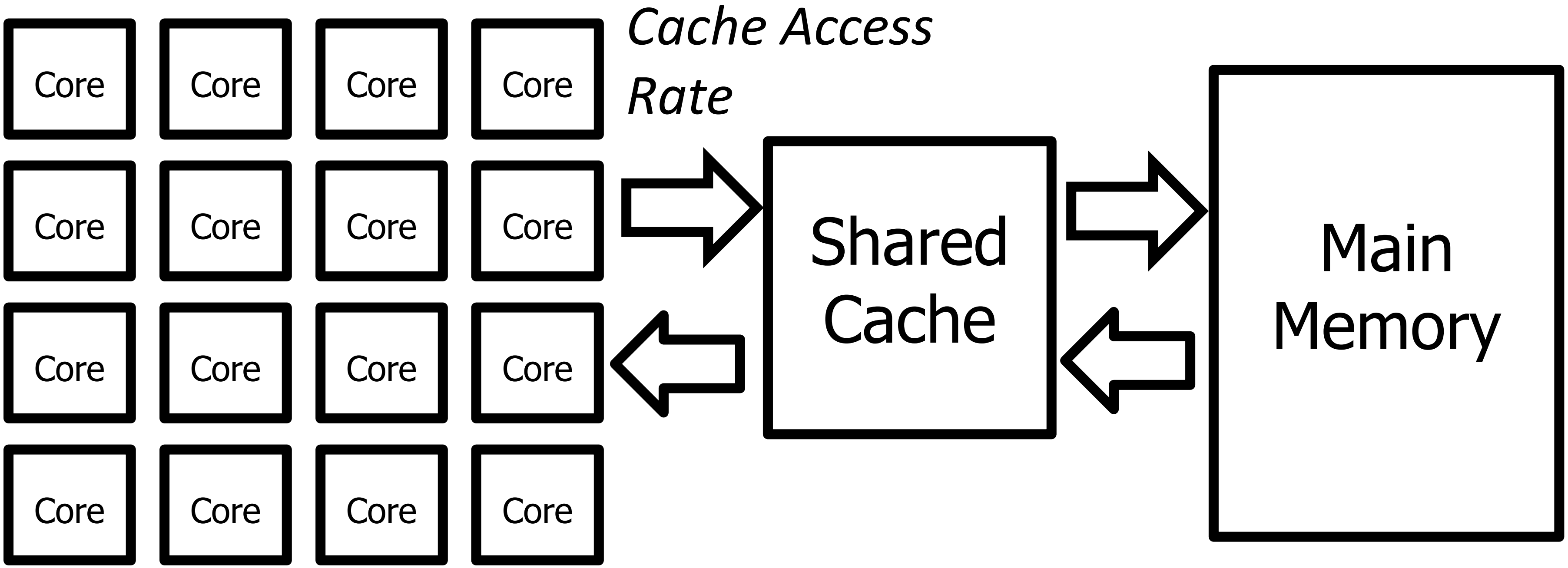
Memory Service Rate

Service Rates vs. Access Rates



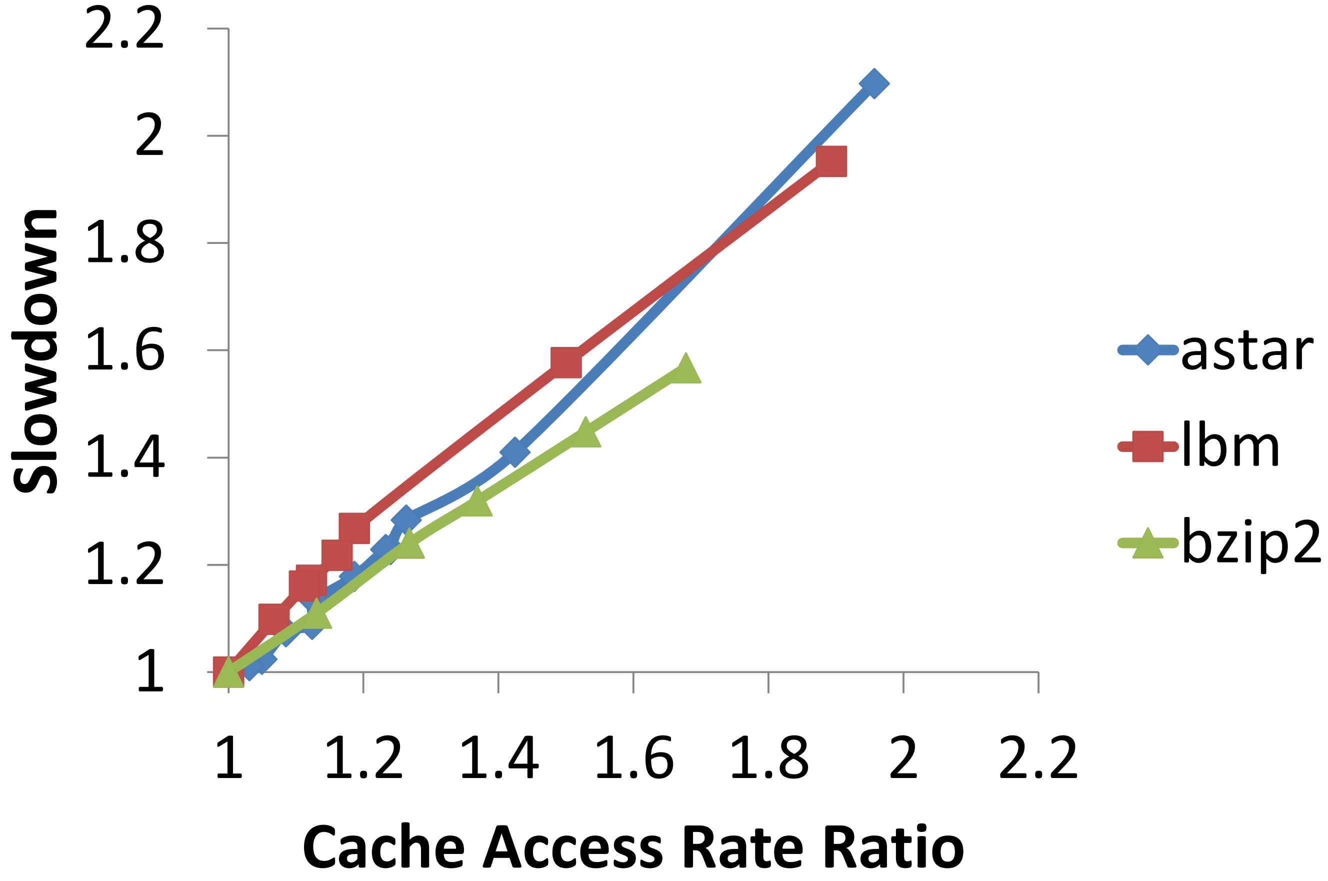
Request service and access rates are tightly coupled

The Application Slowdown Model



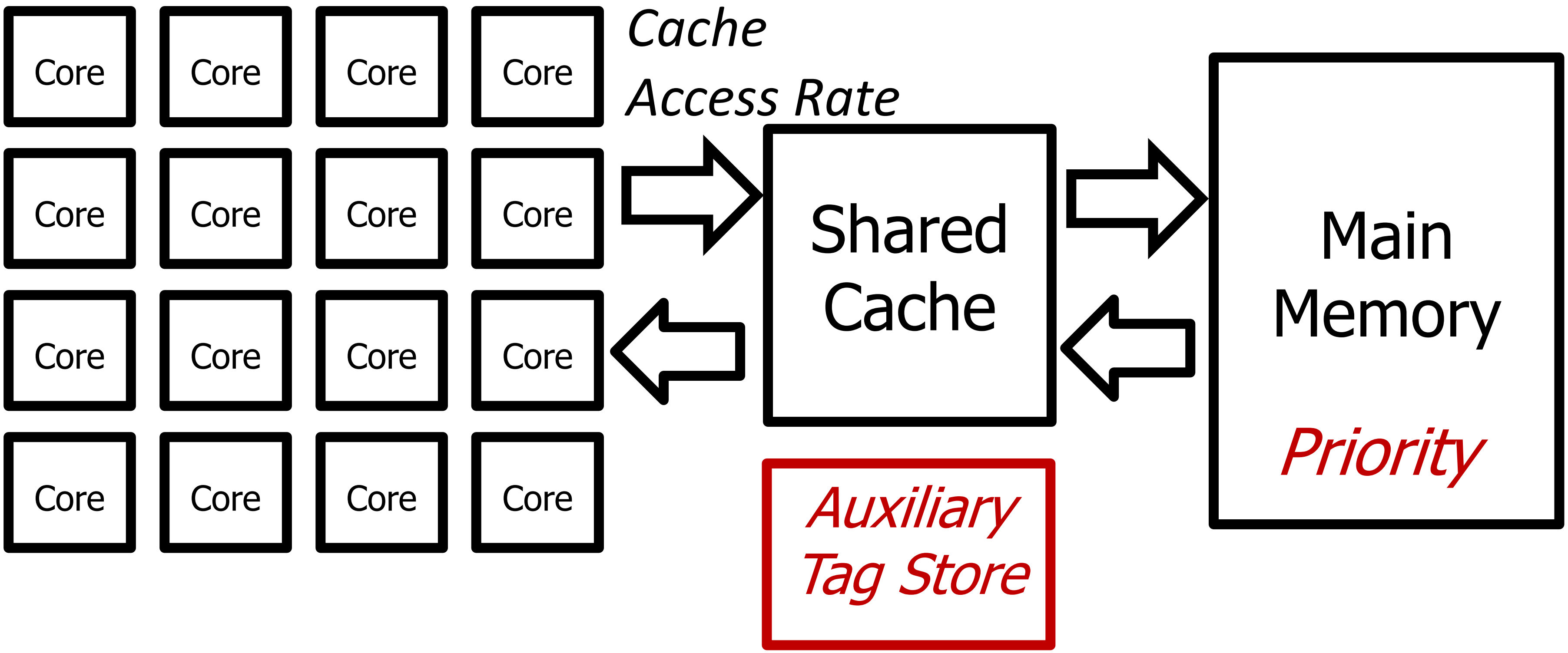
$$\text{Slowdown} = \frac{\text{Cache Access Rate}_{\text{Alone}}}{\text{Cache Access Rate}_{\text{Shared}}}$$

Real System Studies: Cache Access Rate vs. Slowdown

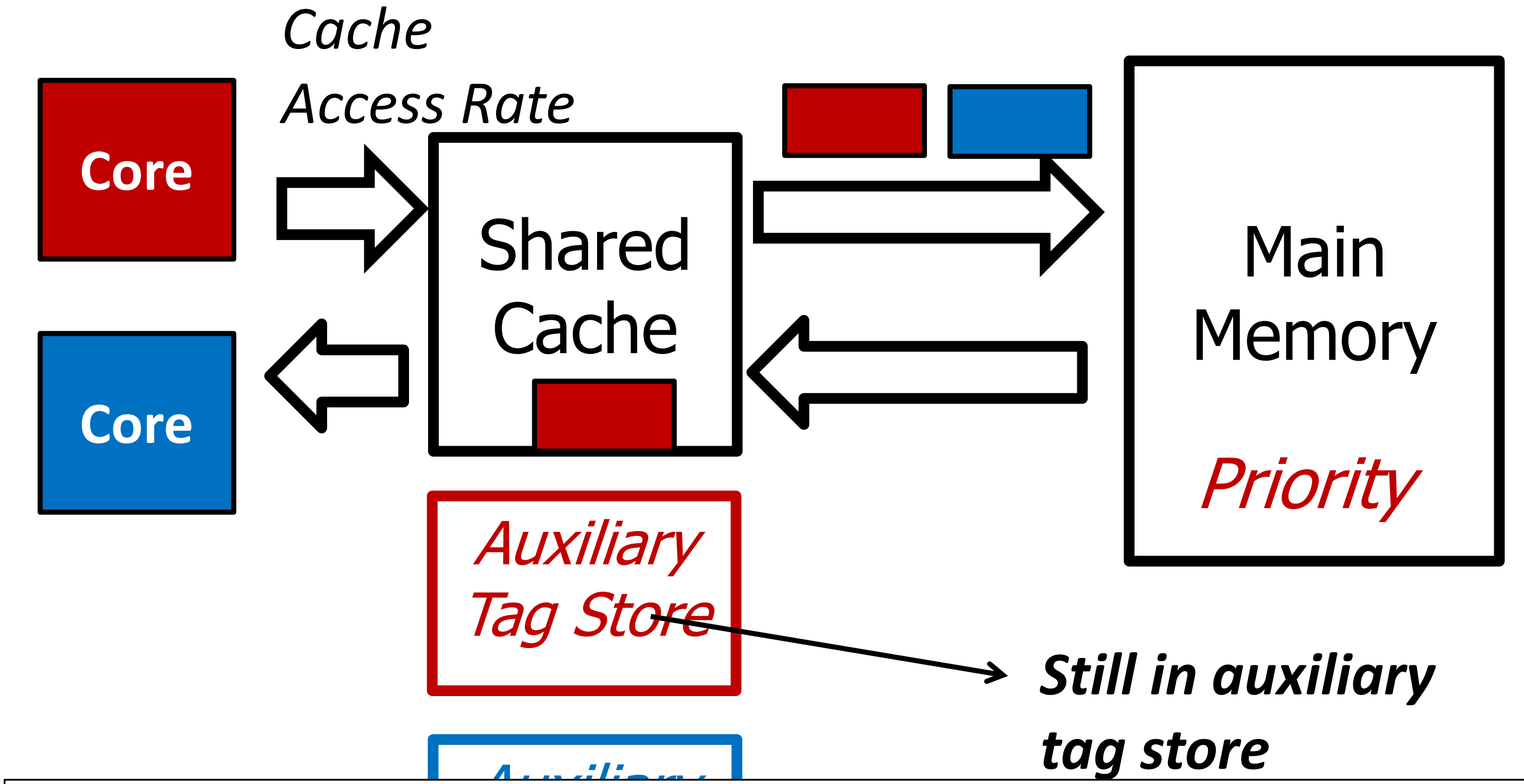


Challenge

How to estimate alone cache access rate?



Auxiliary Tag Store



Still in auxiliary tag store

Auxiliary tag store tracks such ***contention misses***

Accounting for Contention Misses

- Revisiting alone memory request service rate

$$\text{Alone Request Service Rate of an Application} = \frac{\#\,\text{Requests During High Priority Epochs}}{\#\,\text{High Priority Cycles}}$$

Cycles serving contention misses should not count as high priority cycles

Alone Cache Access Rate Estimation

$$\text{Cache Access Rate}_{\text{Alone}} \text{ of an Application} = \frac{\text{\# Requests During High Priority Epochs}}{\text{\# High Priority Cycles} - \boxed{\text{\#Cache Contention Cycles}}}$$

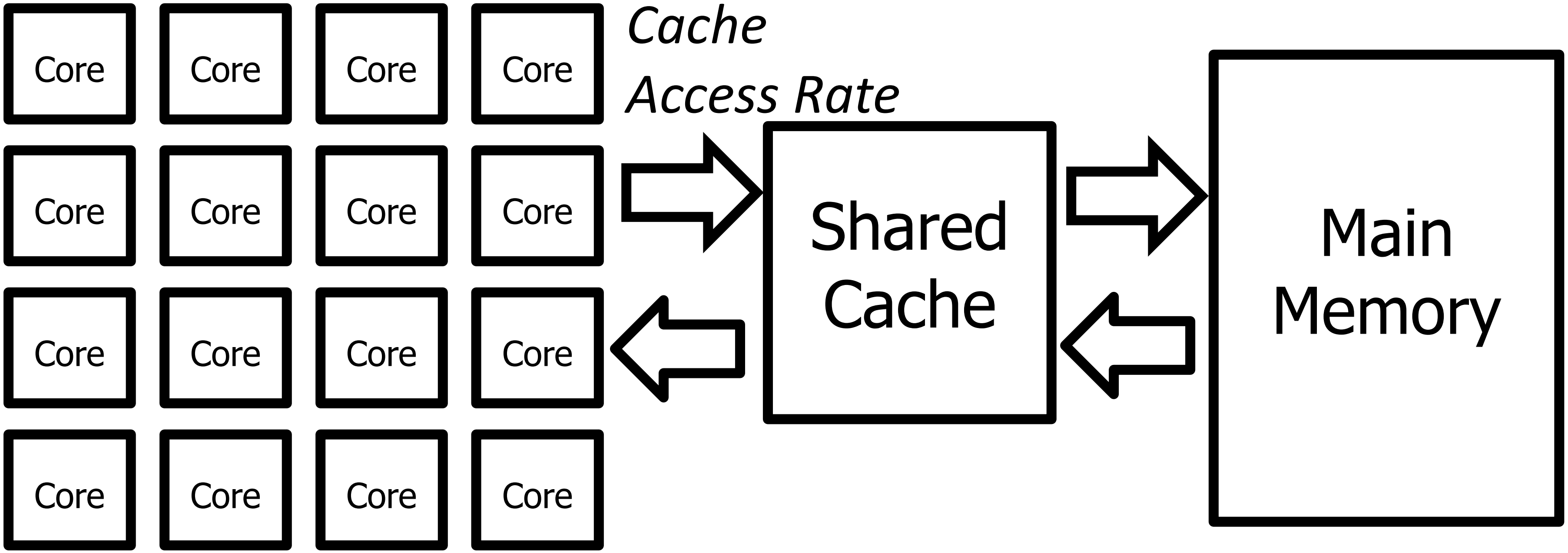
Cache Contention Cycles: Cycles spent serving contention misses

$$\text{Cache Contention Cycles} = \text{\# Contention Misses x Average Memory Service Time}$$

From auxiliary tag store when given high priority

Measured when given high priority

Application Slowdown Model (ASM)



$$\text{Slowdown} = \frac{\text{Cache Access Rate}_{\text{Alone}}}{\text{Cache Access Rate}_{\text{Shared}}}$$

Previous Work on Slowdown Estimation

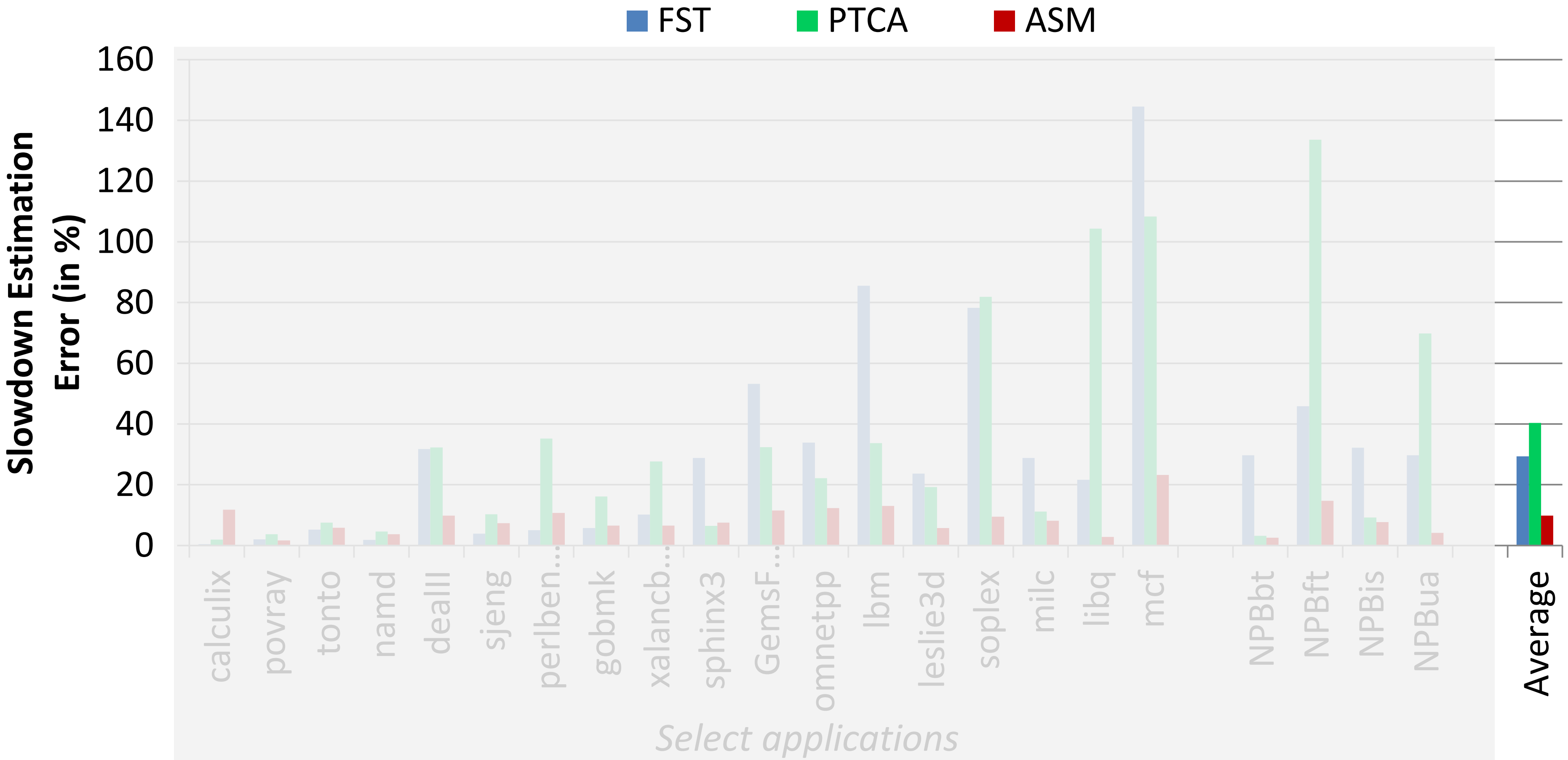
- Previous work on slowdown estimation
  - **STFM** (Stall Time Fair Memory) Scheduling [Mutlu et al., MICRO '07]
  - **FST** (Fairness via Source Throttling) [Ebrahimi et al., ASPLOS '10]
  - **Per-thread Cycle Accounting** [Du Bois et al., HiPEAC '13]

- Basic Idea:

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

Count interference experienced by each request

Model Accuracy Results



Average error of ASM's slowdown estimates: 10%

Leveraging ASM's Slowdown Estimates

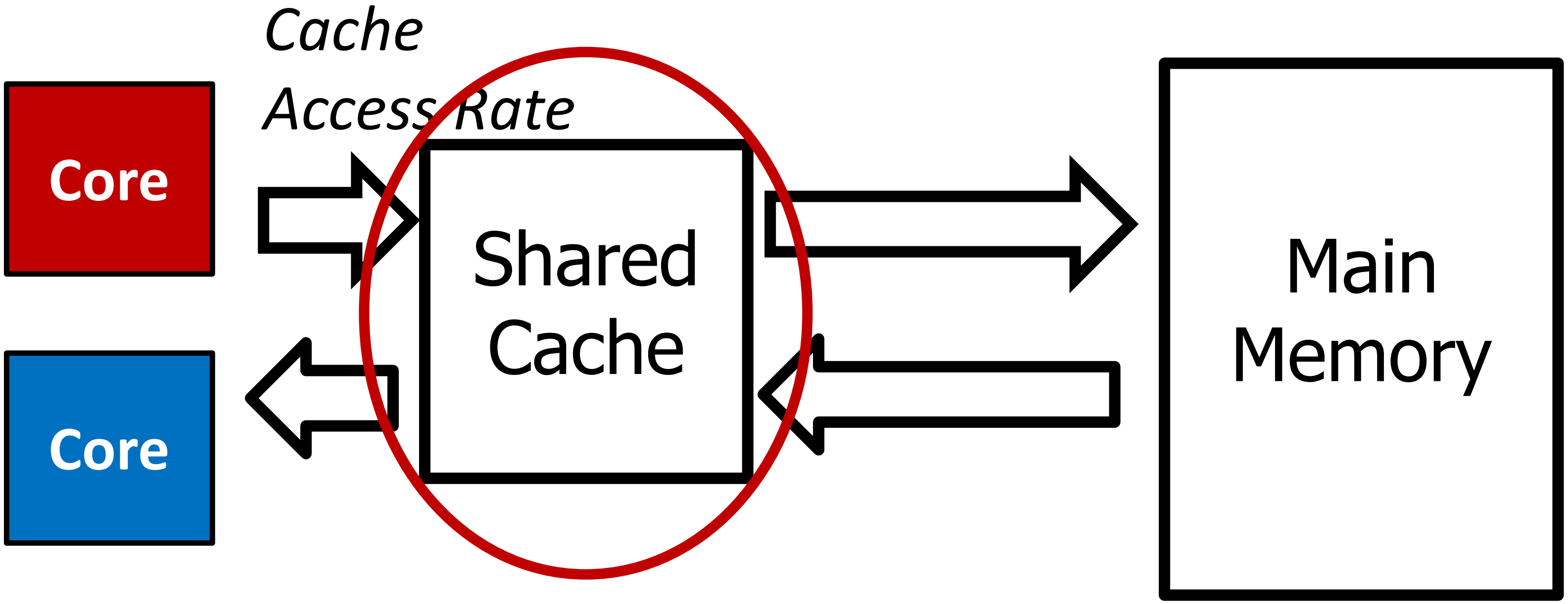
- *Slowdown-aware resource allocation for high performance and fairness*

- *Slowdown-aware resource allocation to bound application slowdowns*

- *VM migration and admission control schemes* ***[VEE '15]***

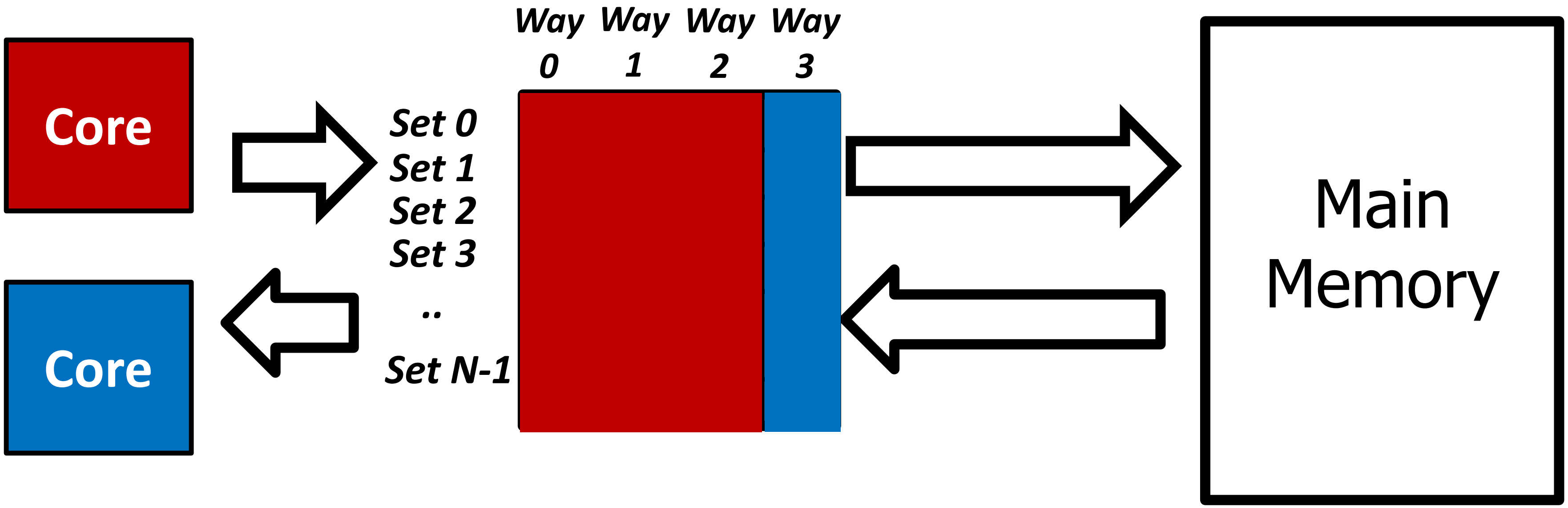
- *Fair billing schemes in a commodity cloud*

Cache Capacity Partitioning



Goal: Partition the shared cache among applications to mitigate contention

Cache Capacity Partitioning



Previous partitioning schemes optimize for miss count

Problem: Not aware of performance and slowdowns

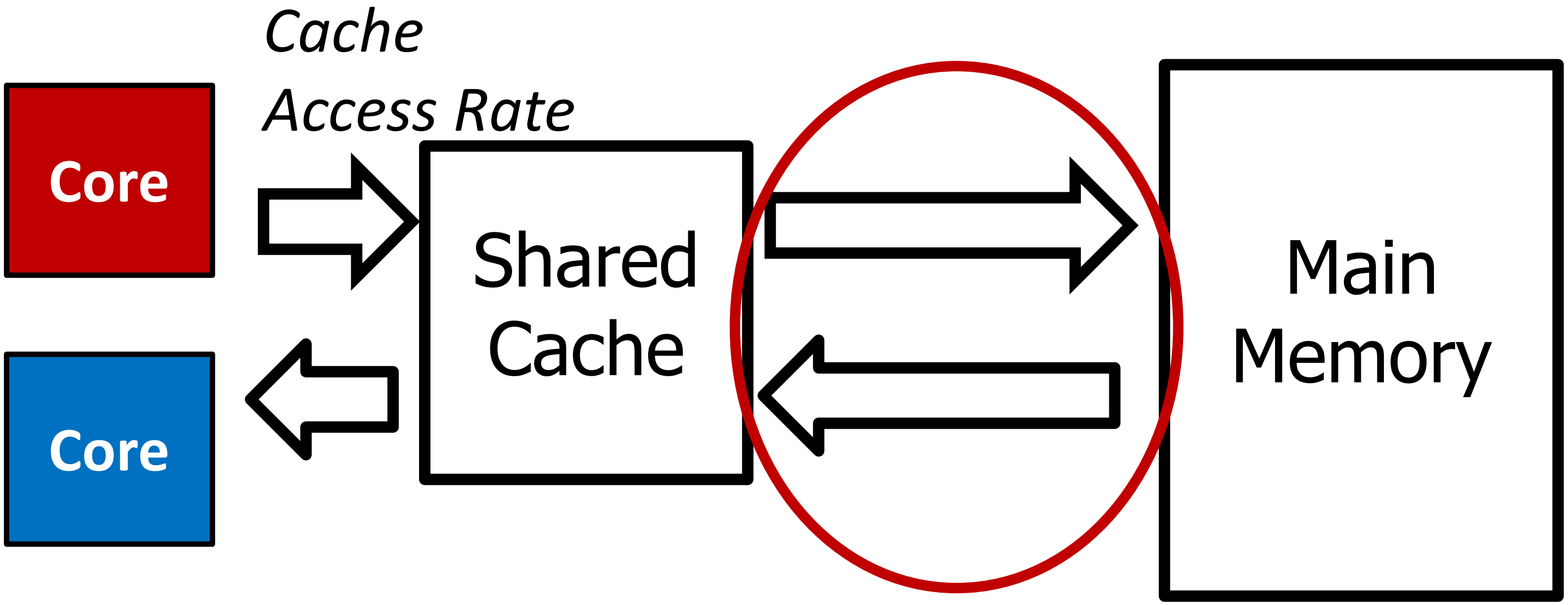
ASM-Cache: Slowdown-aware Cache Way Partitioning

- *Key Requirement: Slowdown estimates for all possible way partitions*

- *Extend ASM to estimate slowdown for all possible cache way allocations*

- *Key Idea: Allocate each way to the application whose slowdown reduces the most*

Memory Bandwidth Partitioning



Goal: Partition the main memory bandwidth among applications to mitigate contention

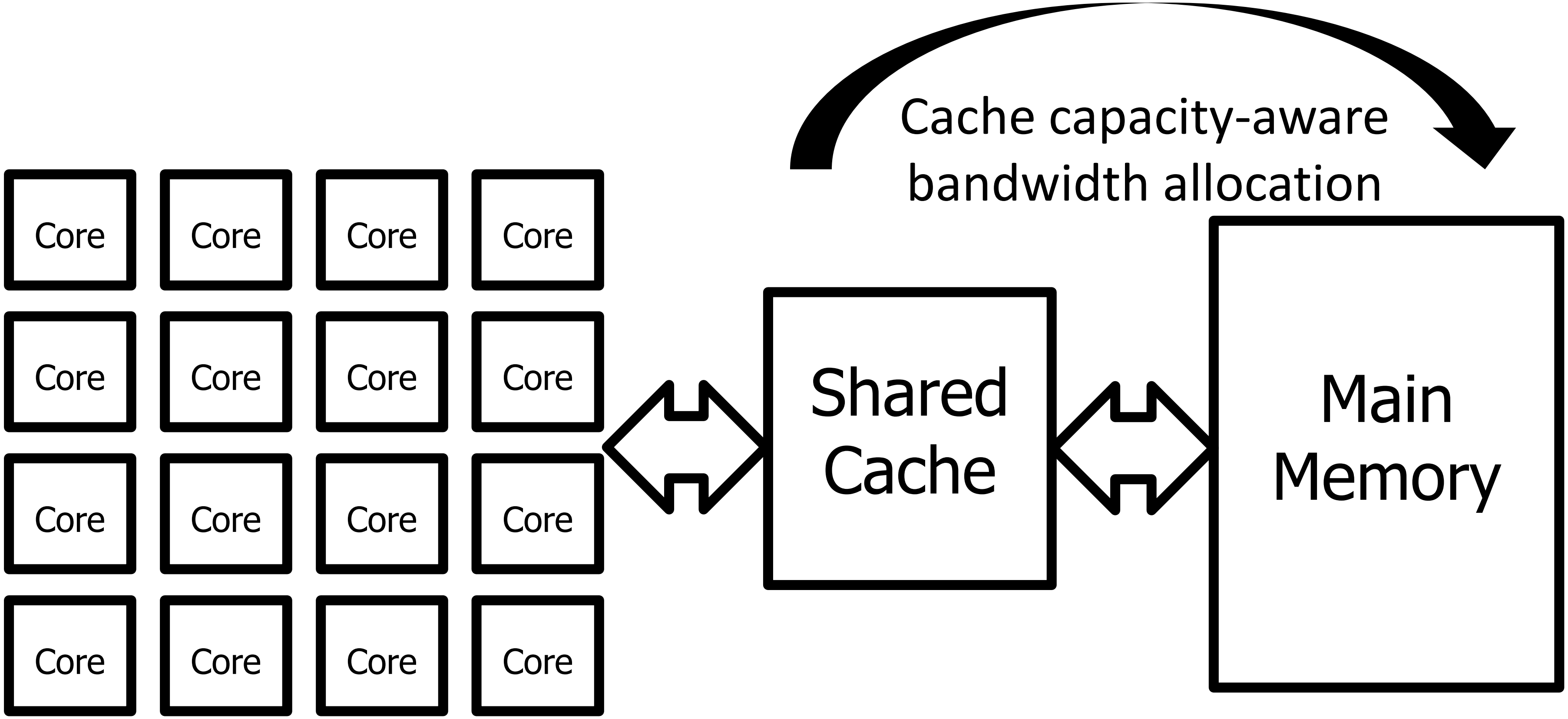
ASM-Mem: Slowdown-aware Memory Bandwidth Partitioning

- *Key Idea: Allocate high priority proportional to an application's slowdown*

$$\text{High Priority Fraction}_i = \frac{\text{Slowdown}_i}{\sum_j \text{Slowdown}_j}$$

- *Application i's requests given highest priority at the memory controller for its fraction*

Coordinated Resource Allocation Schemes

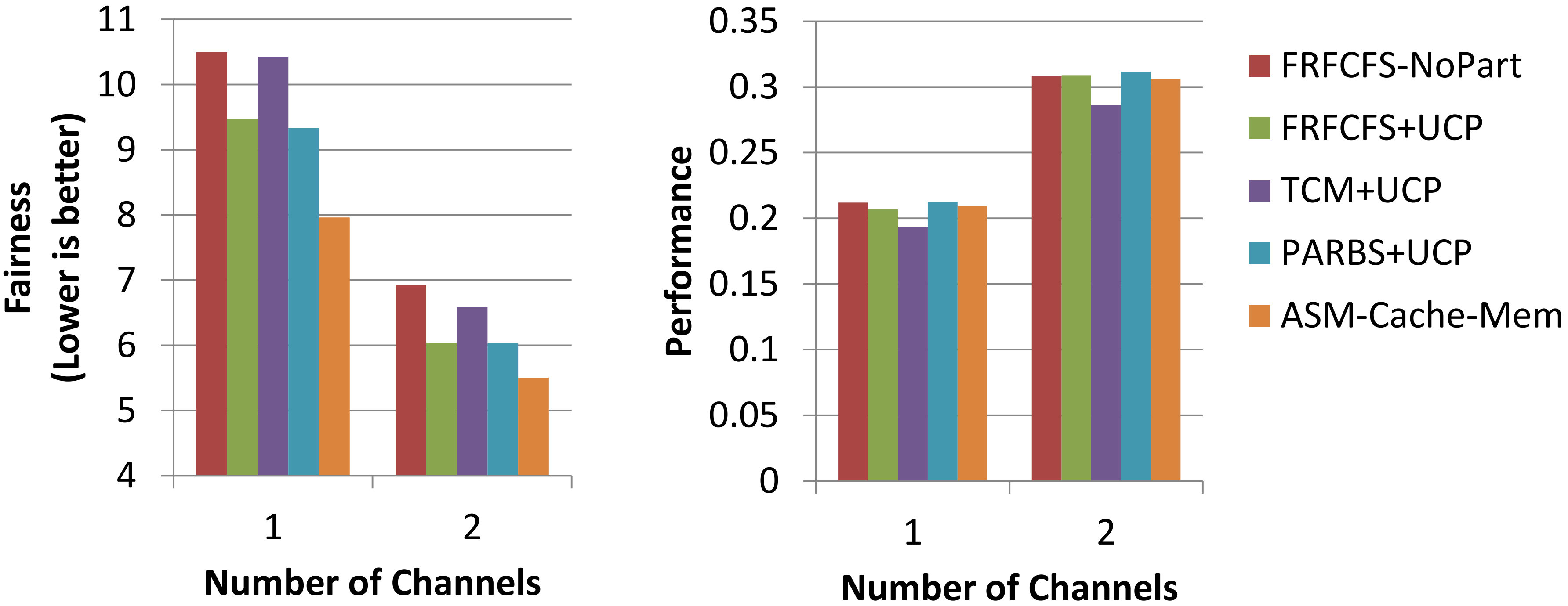


1. Employ ASM-Cache to partition cache capacity

2. Drive ASM-Mem with slowdowns from ASM-Cache

Fairness and Performance Results

16-core system
100 workloads



Significant fairness benefits across different channel counts

Outline

	Mitigate Interference	*Quantify Interference*
Cache Capacity	Much explored *Not our focus*	*Not our focus*
Memory Bandwidth	✓ ***Blacklisting Memory Scheduler***	✓ ***Memory Interference induced Slowdown Estimation Model and its uses***

✓ ***Application Slowdown Model and its uses***

Thesis Contributions

- *Principles behind our scheduler and models*
  - Simple two-level prioritization sufficient to mitigate interference
  - Request service rate a proxy for performance
- *Simple and high-performance memory scheduler design*
- *Accurate slowdown estimation models*
- *Mechanisms that leverage our slowdown estimates*

Summary

- Problem: Shared resource interference causes high and unpredictable application slowdowns

- Goals: High and predictable performance

- Approaches: Mitigate and quantify interference

- Thesis Contributions:
  1. Principles behind our scheduler and models
  2. Simple and high-performance memory scheduler
  3. Accurate slowdown estimation models
  4. Mechanisms that leverage our slowdown estimates

Future Work

- *Leveraging slowdown estimates at the system and cluster level*

- *Interference estimation and performance predictability for multithreaded applications*

- *Performance predictability in heterogeneous systems*

- *Coordinating the management of main memory and storage*

Research Summary

- Predictable performance in multicore systems *[HPCA ’13, SuperFri ’14, KIISE ’15]*
- High and predictable performance in heterogeneous systems *[ISCA ’12, SAFARI Tech Report ’15]*
- Low-complexity memory scheduling *[ICCD ’14]*
- Memory channel partitioning *[MICRO ’11]*
- Architecture-aware cluster management *[VEE ’15]*
- Low-latency DRAM architectures *[HPCA ’13]*

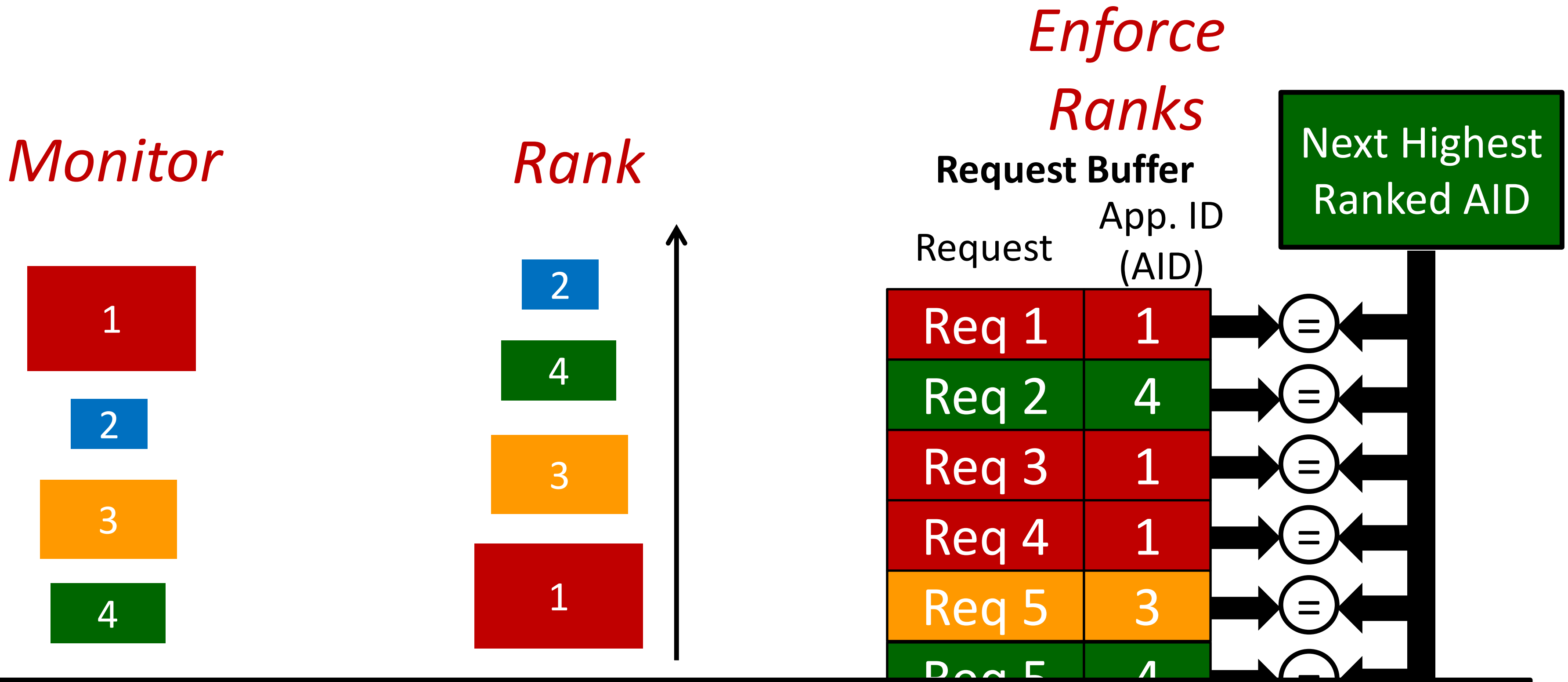
Backup Slides

Blacklisting

Problems with Previous Application-aware Memory Schedulers

1. **Full ranking increases hardware complexity**
2. Full ranking causes unfair slowdowns

Ranking Increases Hardware Complexity



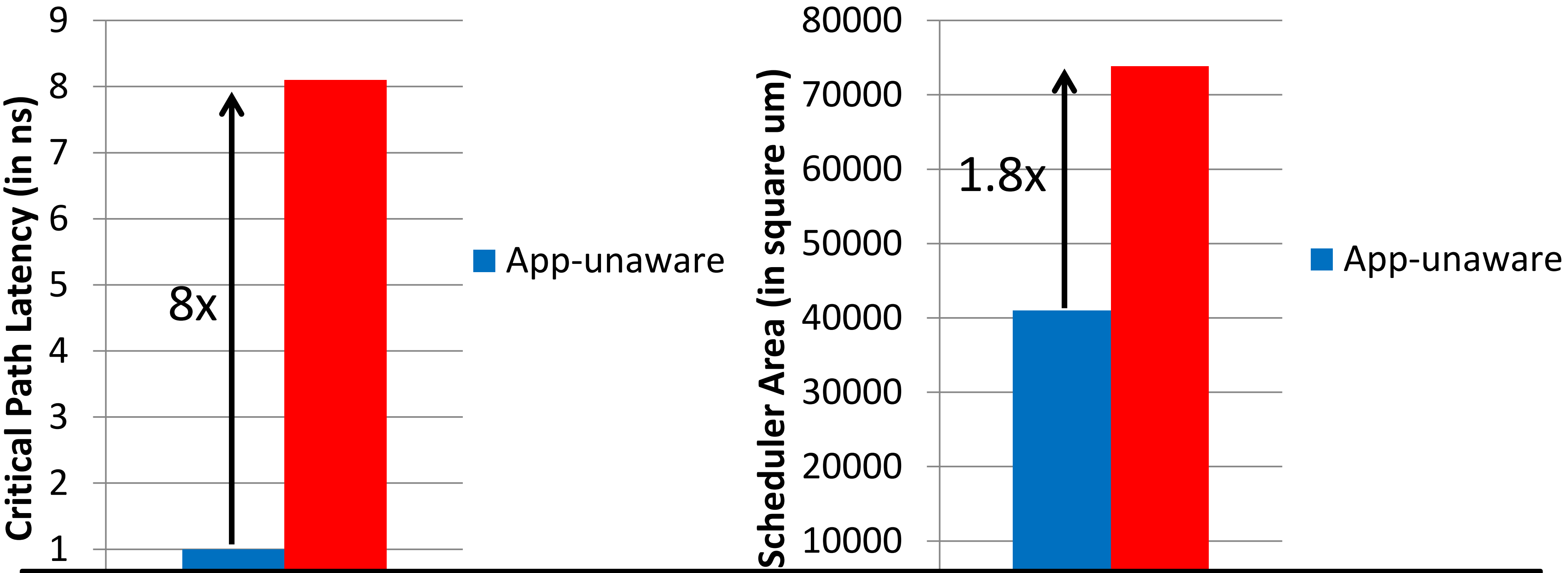
Request Buffer

Request	App. ID (AID)
Req 1	1
Req 2	4
Req 3	1
Req 4	1
Req 5	3
Req 5	4

Hardware complexity increases with application/core count

Ranking Increases Hardware Complexity

From synthesis of RTL implementations using a 32nm library



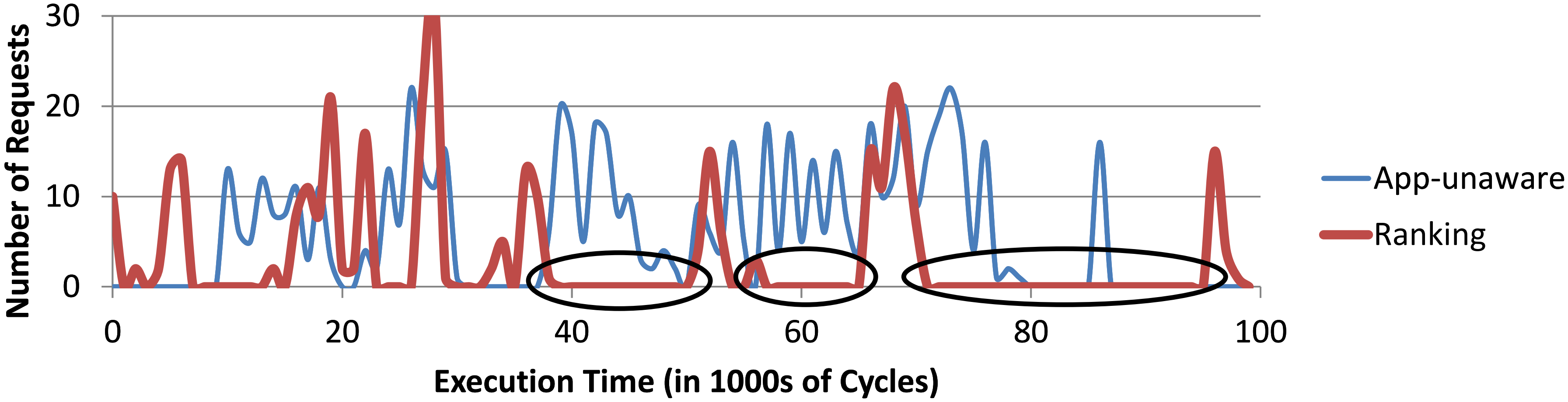
Ranking-based application-aware schedulers incur high hardware cost

Problems with Previous Application-aware Memory Schedulers

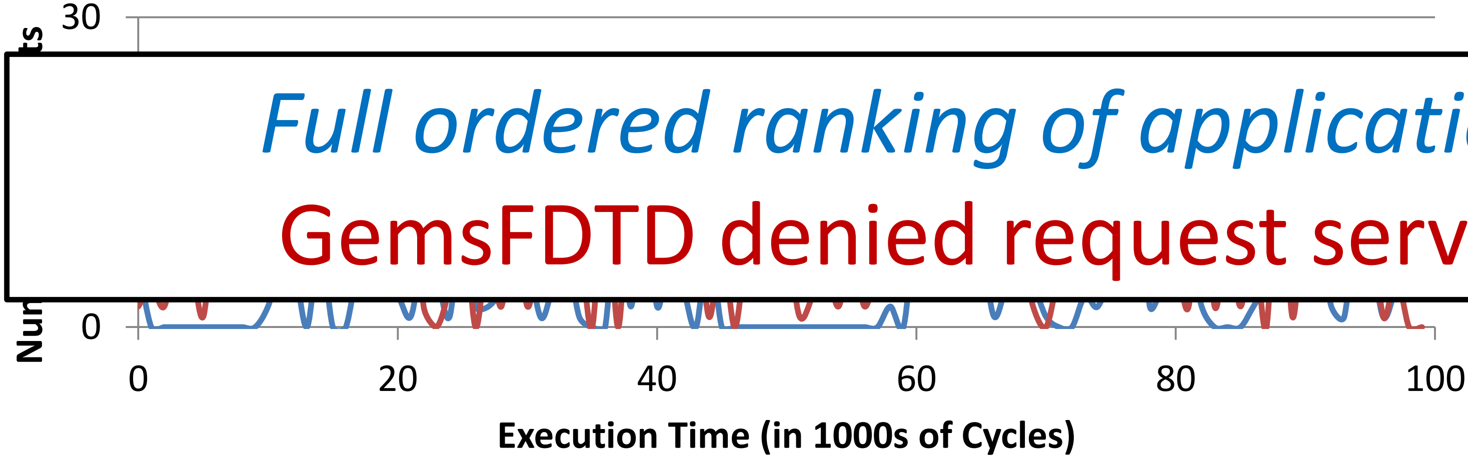
1. Full ranking increases hardware complexity
2. **Full ranking causes unfair slowdowns**

Ranking Causes Unfair Slowdowns

GemsFDTD *(high memory intensity)*



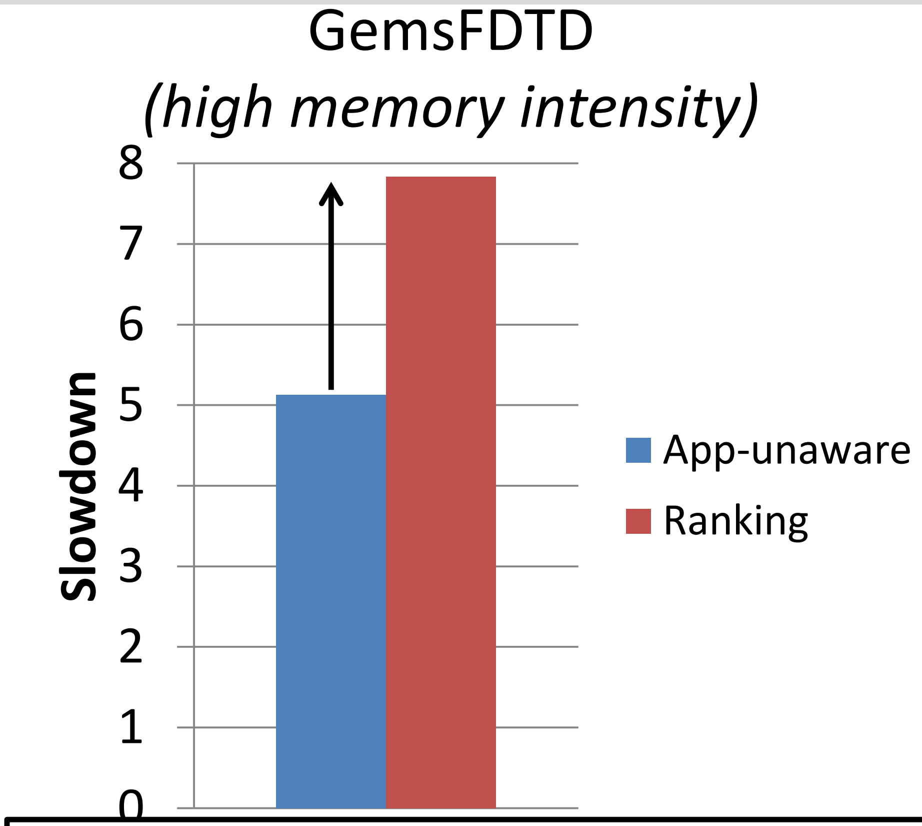
sjeng *(low memory intensity)*

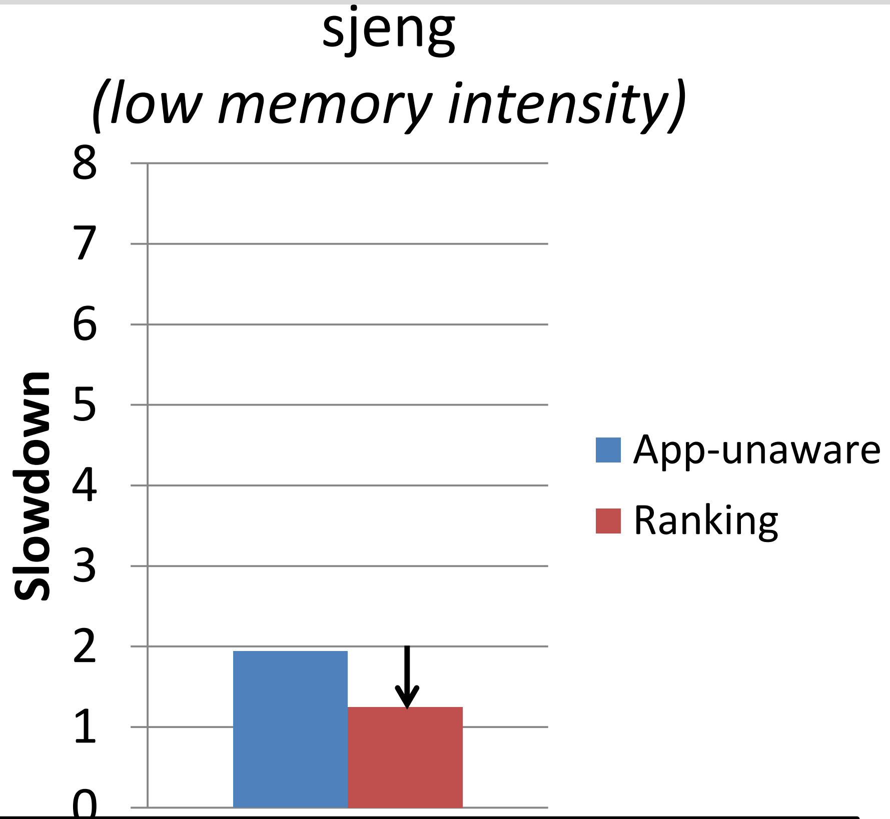


Full ordered ranking of applications

GemsFDTD denied request service

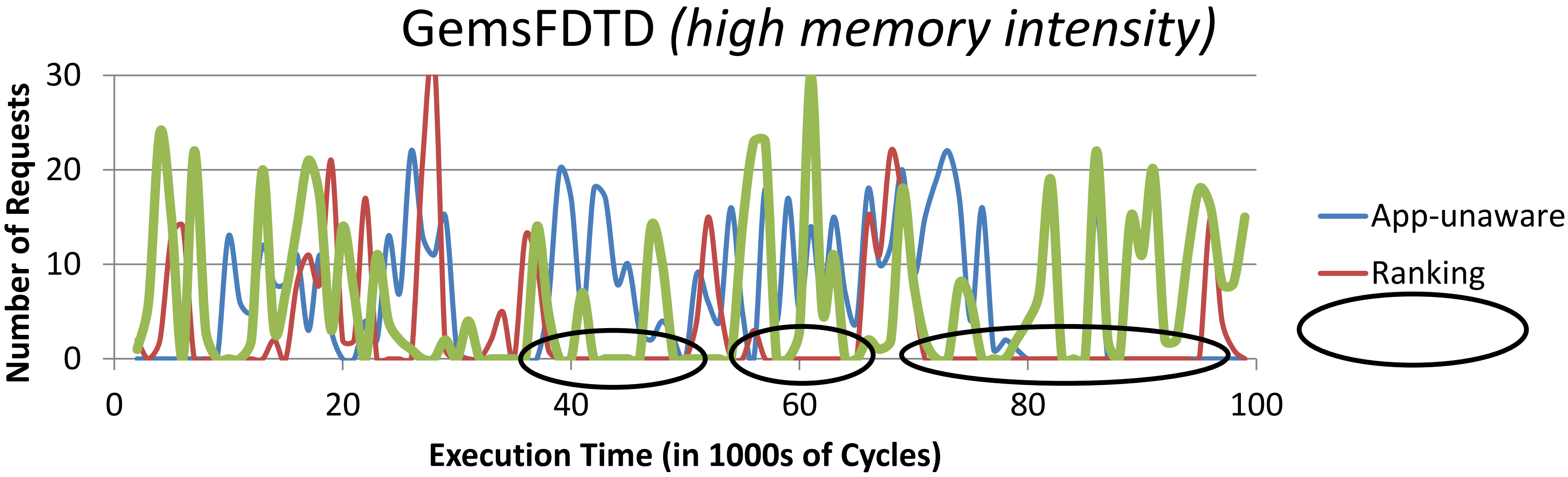
Ranking Causes Unfair Slowdowns


GemsFDTD
(high memory intensity)


sjeng
(low memory intensity)

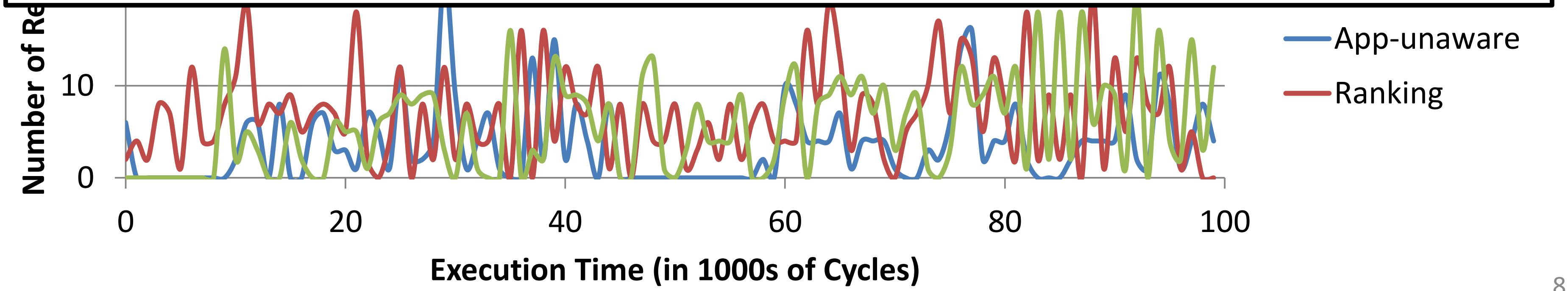
Ranking-based application-aware schedulers cause unfair slowdowns

Key Observation 1: Group Rather Than Rank

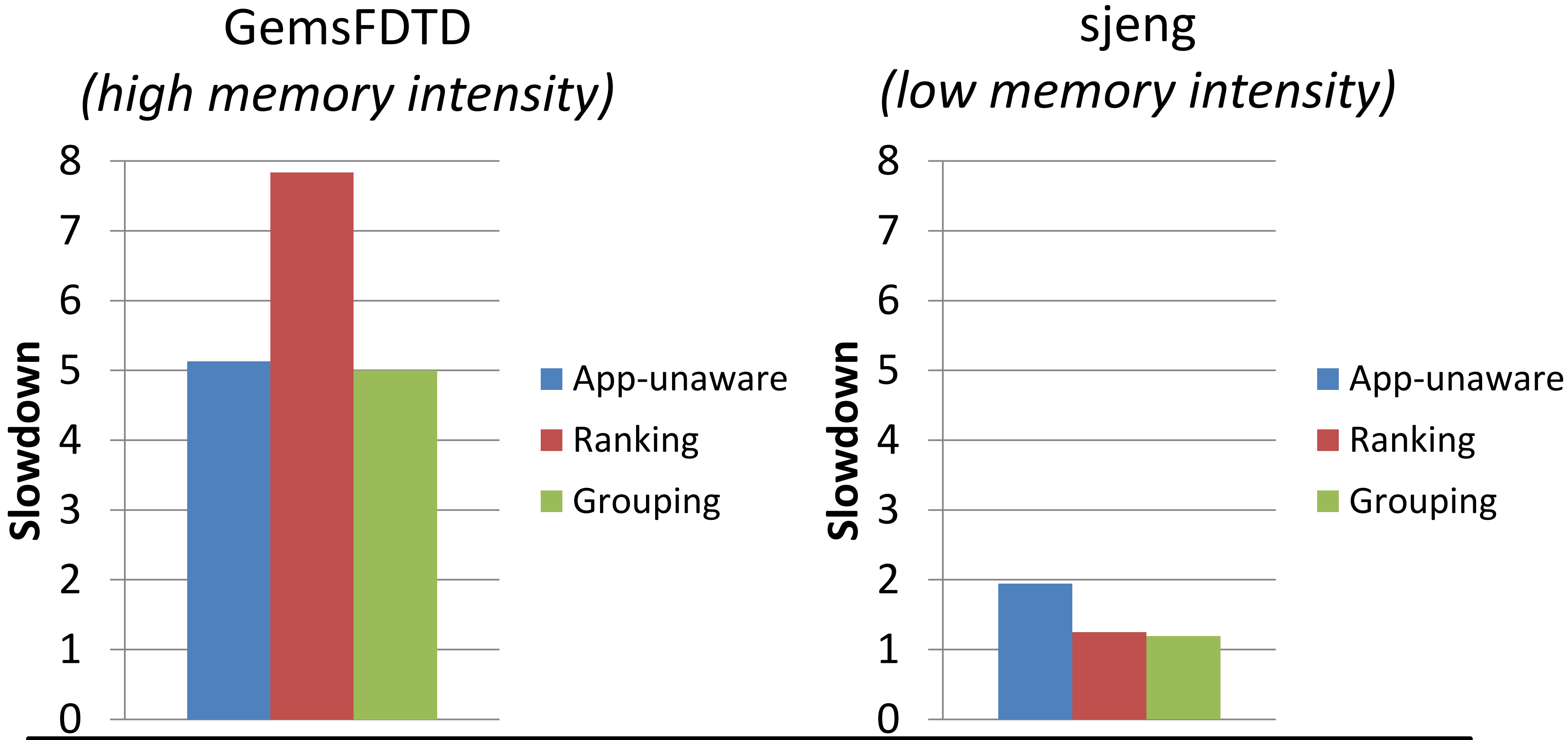


GemsFDTD *(high memory intensity)*

No unfairness due to denial of request service



Key Observation 1: Group Rather Than Rank



Benefit 2: Lower slowdowns than ranking

Previous Memory Schedulers

- FRFCFS [Zuravleff and Robinson, US Patent 1997, Rixner et al., ISCA 2000]
  - Prioritizes row-buffer hits and older requests
- FRFCFS-Cap [Mutlu and Moscibroda, MICRO 2007]
  - Caps number of consecutive row-buffer hits

Application-unaware

+ Low complexity

- Low performance and fairness

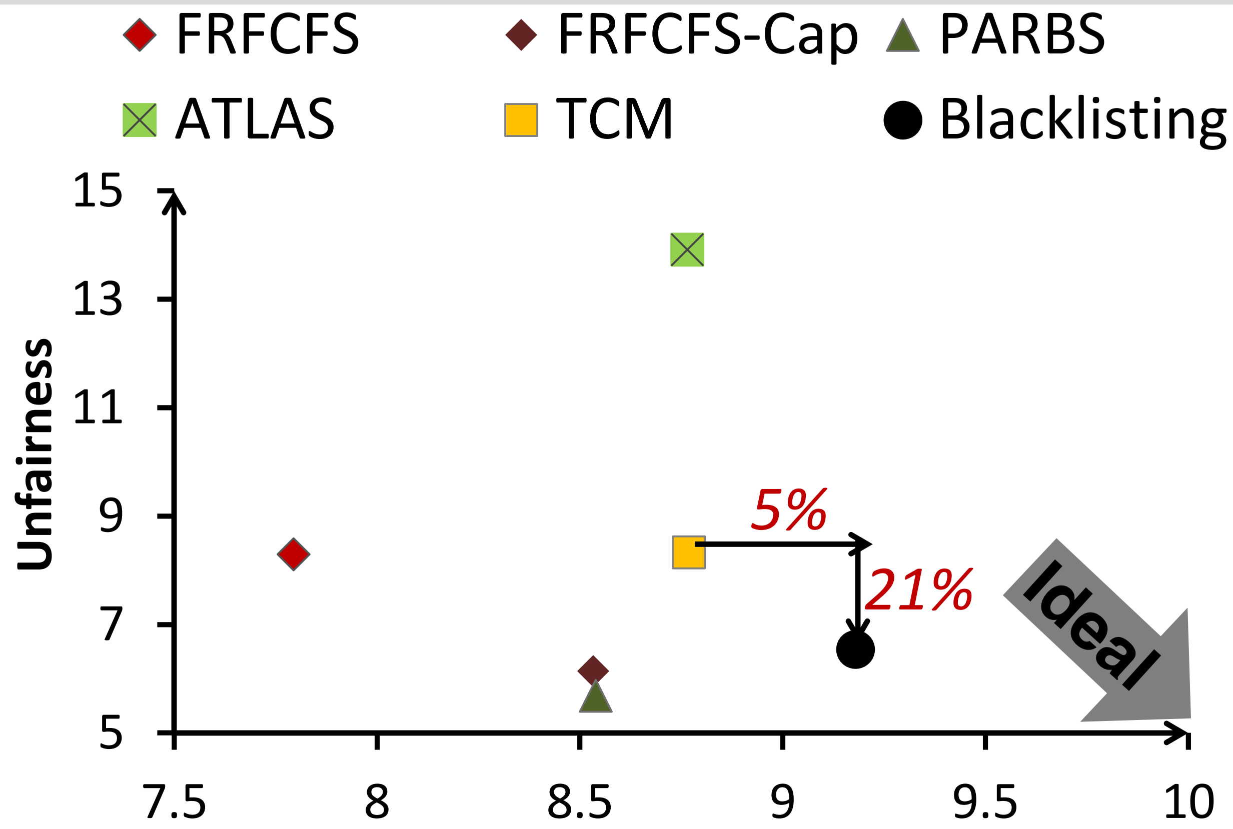
- PARBS [Mutlu and Moscibroda, ISCA 2008]
  - Batches oldest requests from each application; prioritizes batch
  - Employs ranking within a batch
- ATLAS [Kim et al., HPCA 2010]
  - Prioritizes applications with low memory-intensity
- TCM [Kim et al., MICRO 2010]
  - Always prioritizes low memory-intensity applications
  - Shuffles thread ranks of high memory-intensity applications

Application-aware

+ High performance and fairness

- High complexity

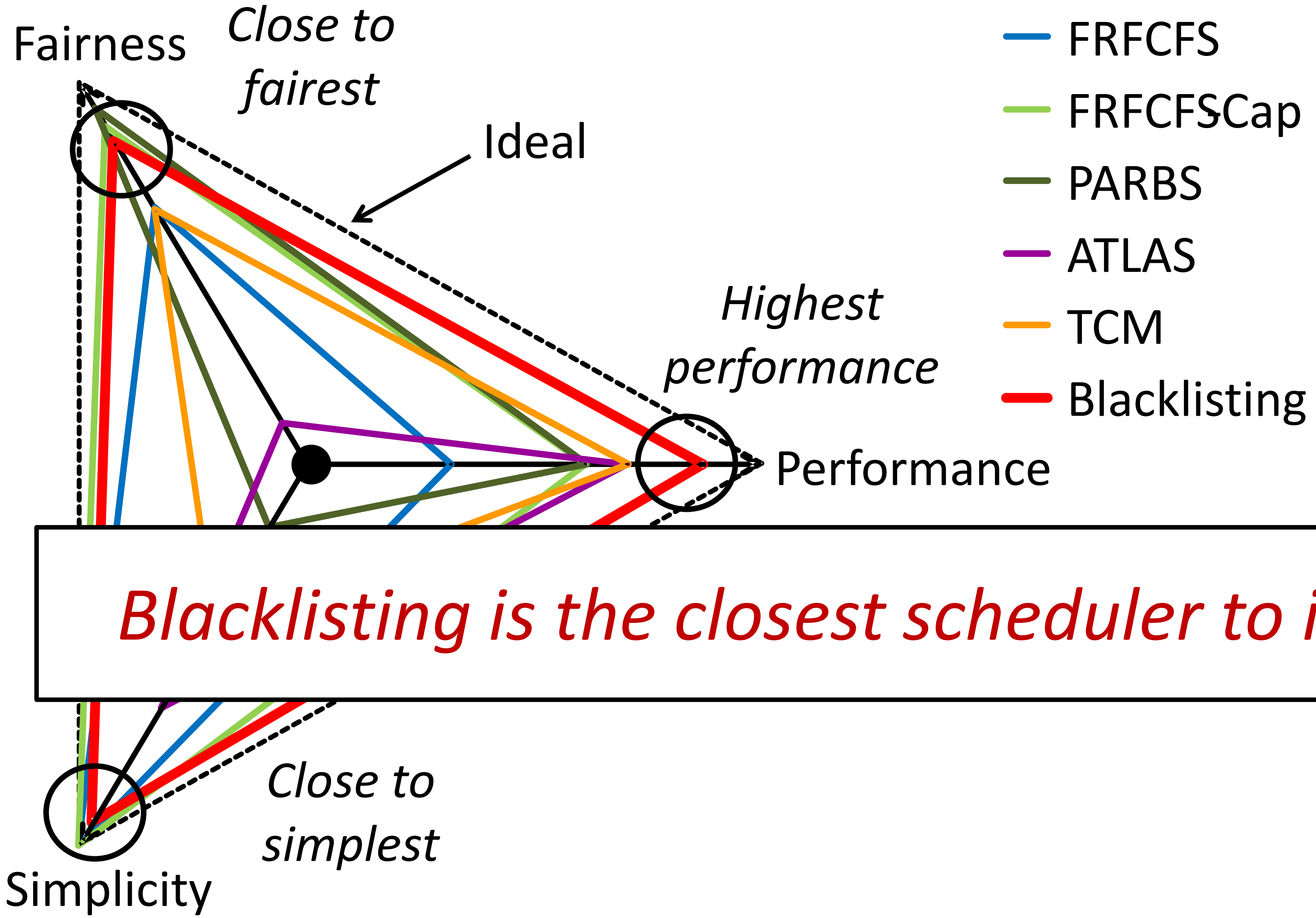
Performance and Fairness



1. Blacklisting achieves the highest performance

2. Blacklisting balances performance and fairness

Performance vs. Fairness vs. Simplicity



Blacklisting is the closest scheduler to ideal

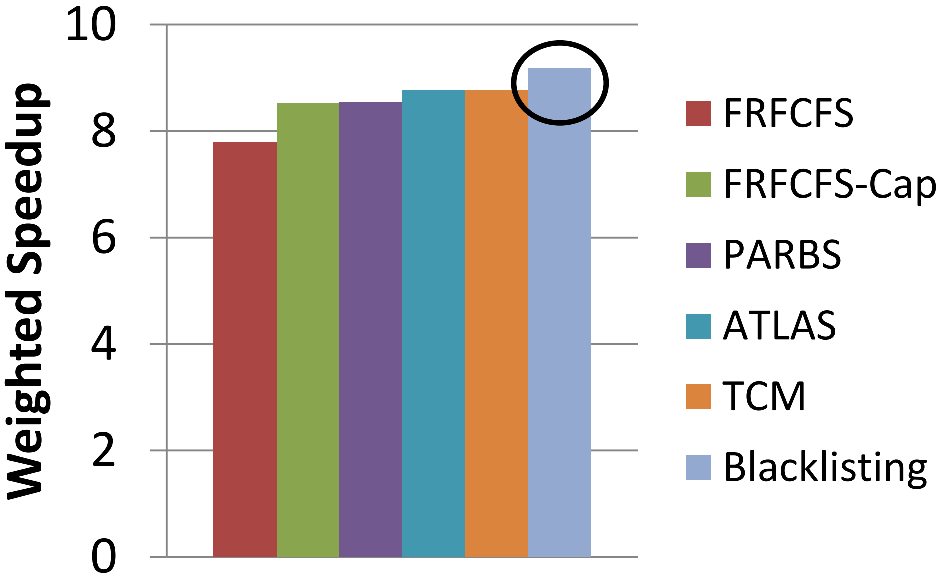
Summary

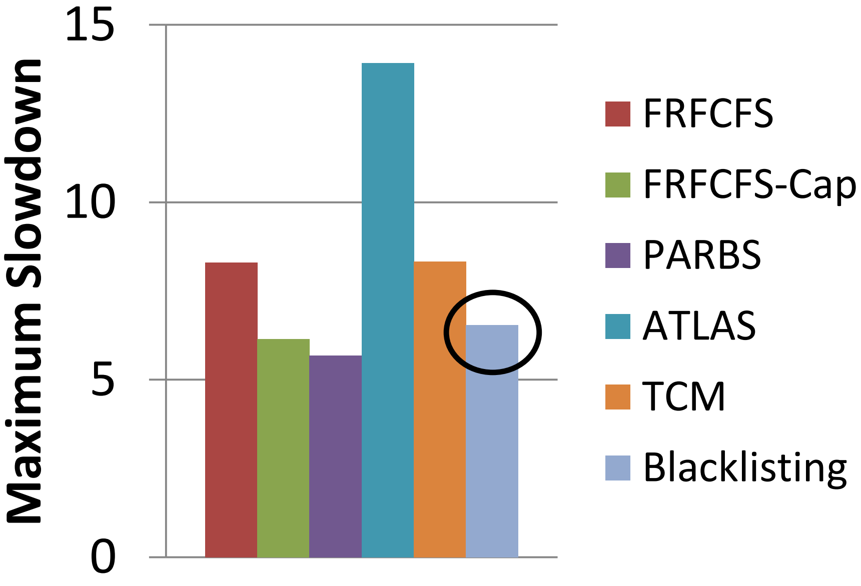
- Applications' requests interfere at main memory
- Prevalent solution approach
  - *Application-aware memory request scheduling*
- Key shortcoming of previous schedulers: *Full ranking*
  - *High hardware complexity*
  - *Unfair application slowdowns*

- Our Solution: Blacklisting memory scheduler
  - *Sufficient to group applications rather than rank*
  - *Group by tracking number of consecutive requests*

- *Much simpler than application-aware schedulers at higher performance and fairness*

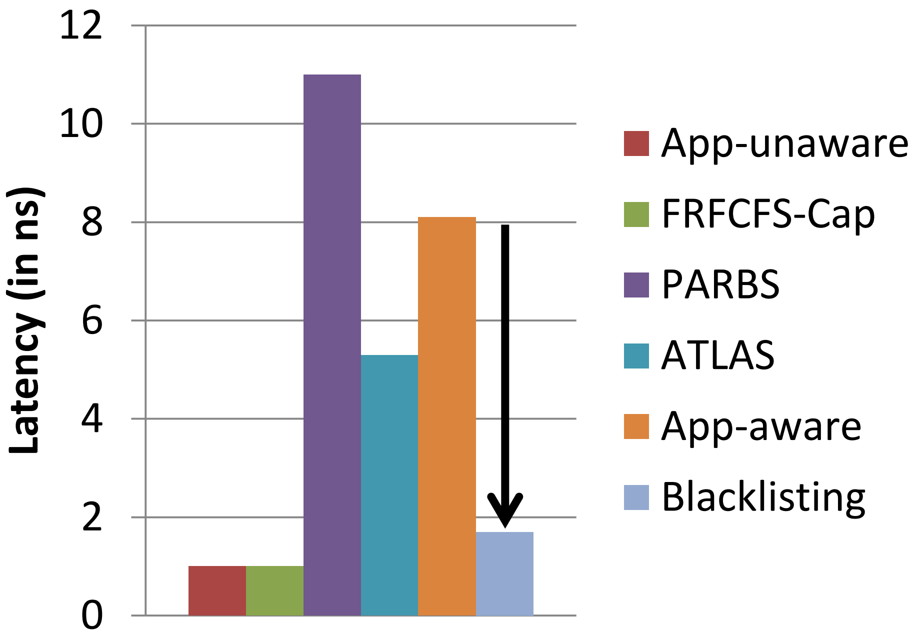
Performance and Fairness

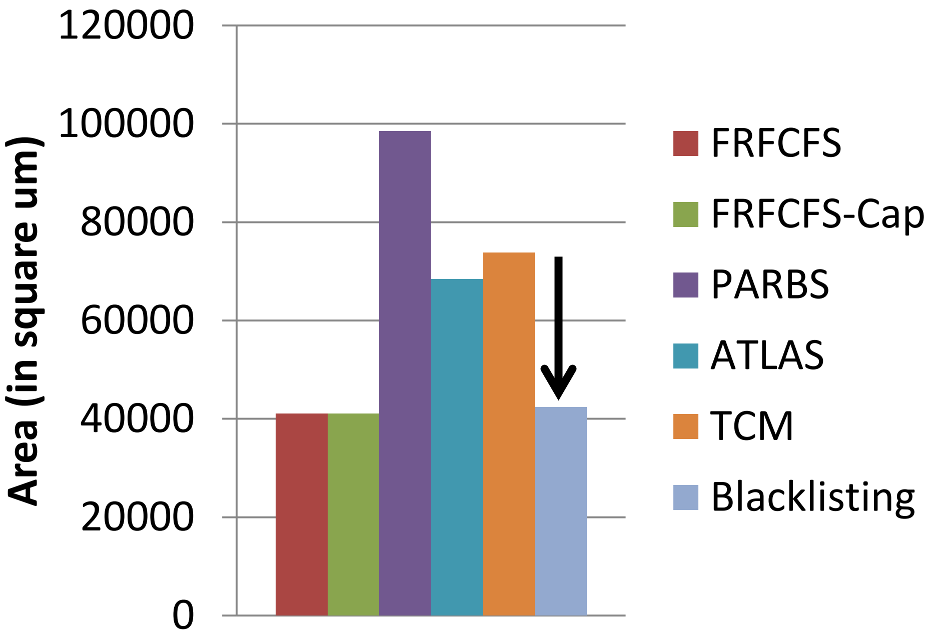




5% higher system performance and 21% lower maximum slowdown than TCM

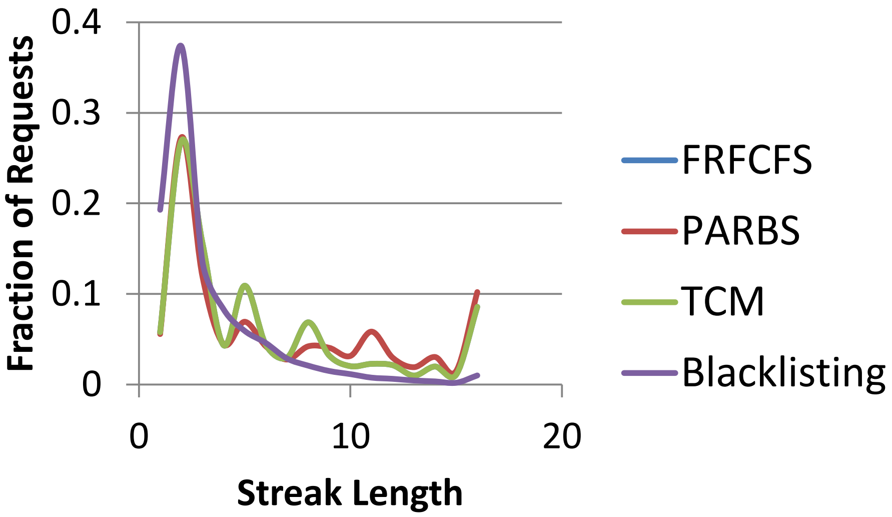
Complexity Results



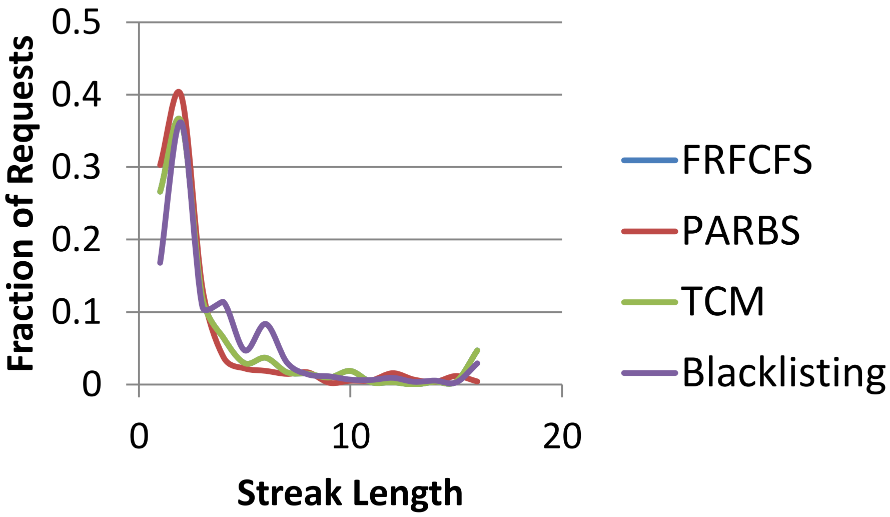


Blacklisting achieves
43% lower area than TCM

Understanding Why Blacklisting Works



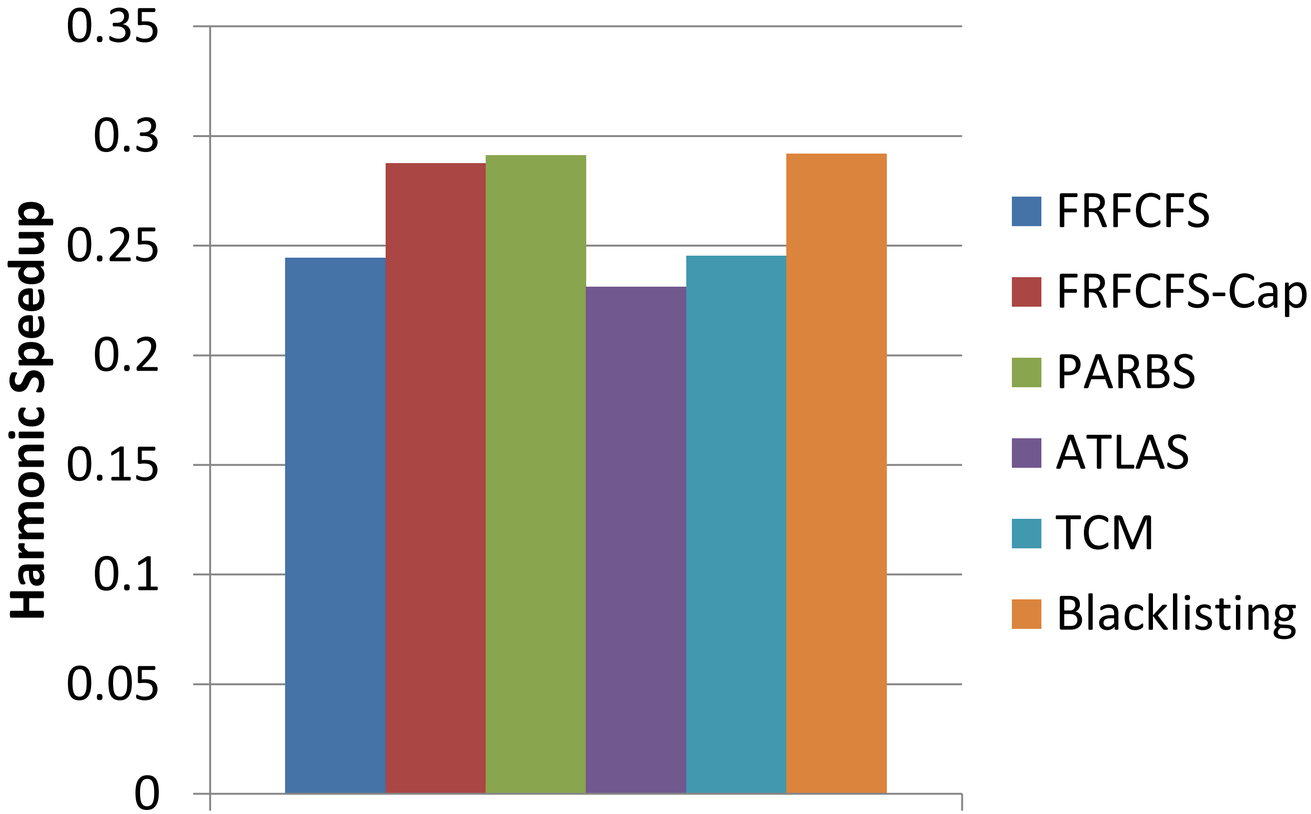
libquantum
(High memory-intensity
application)



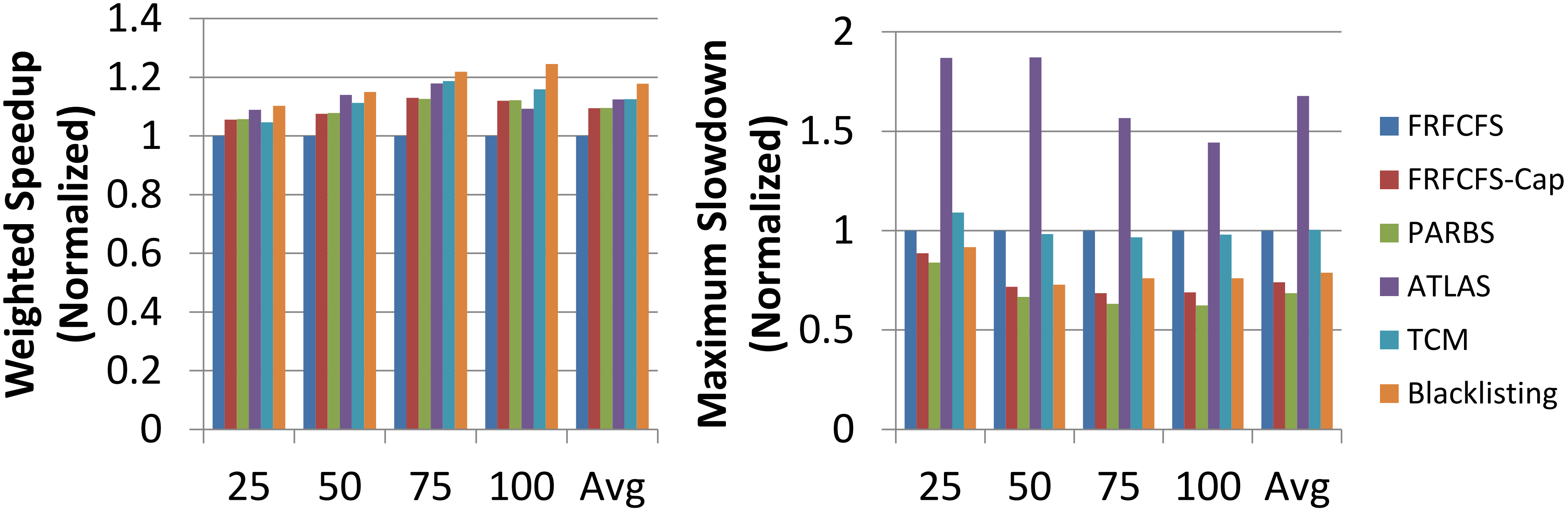
calculix
(Low memory-intensity
application)

Blacklisting shifts the request distribution towards the right

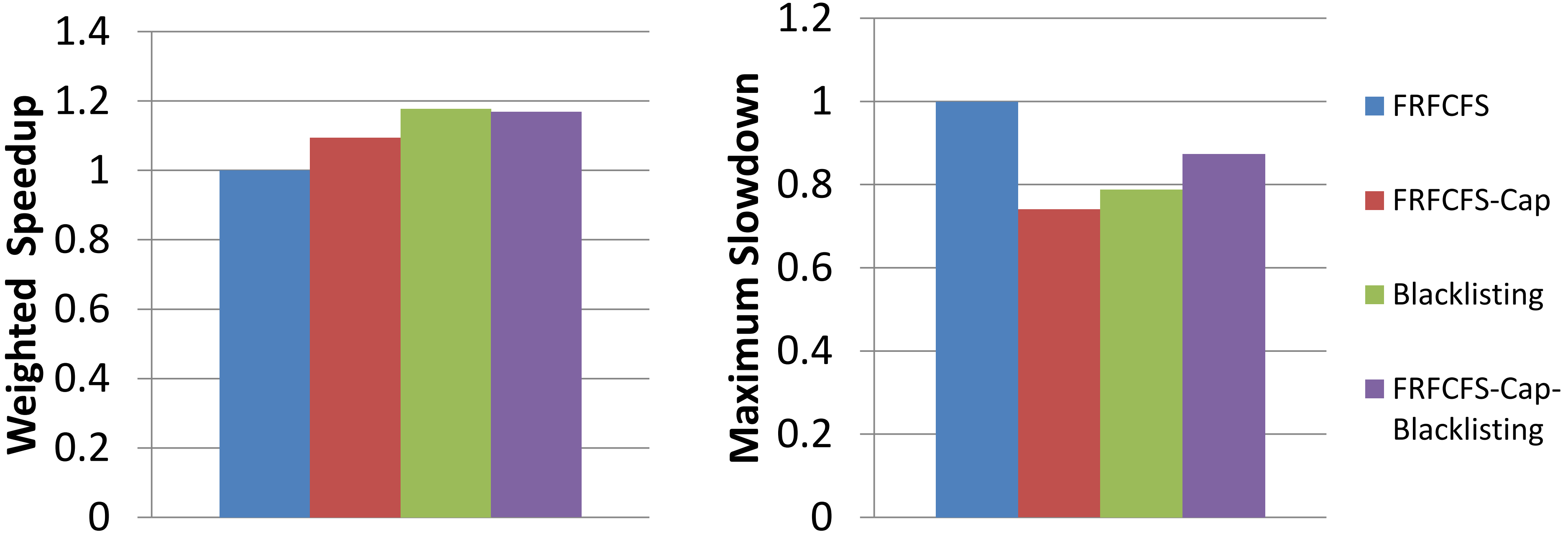
Harmonic Speedup



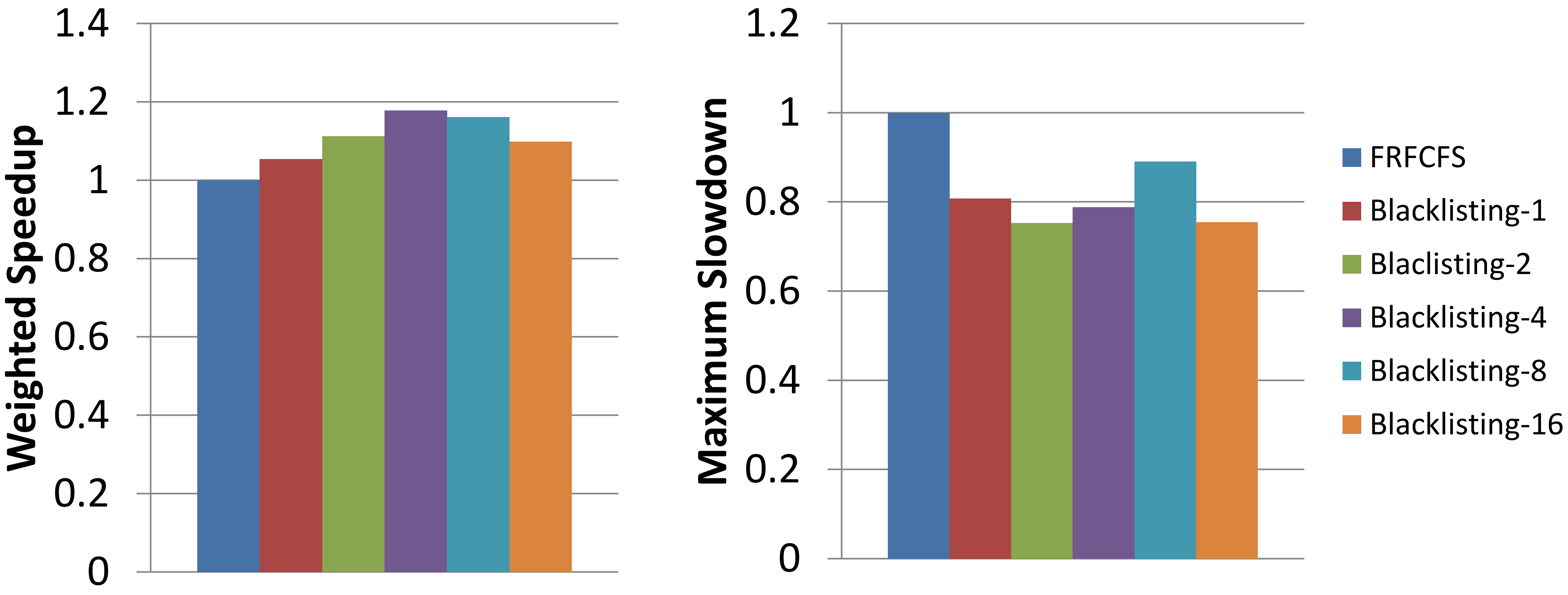
Effect of Workload Memory Intensity



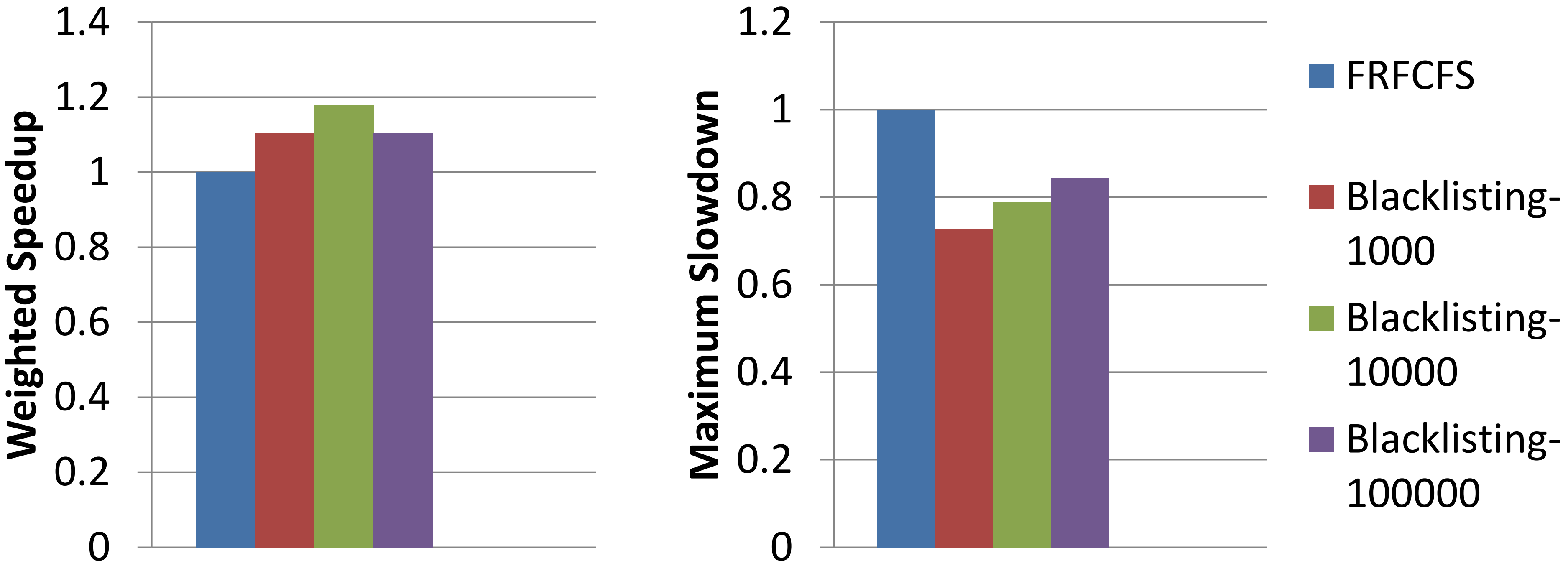
Combining FRFCFS-Cap and Blacklisting



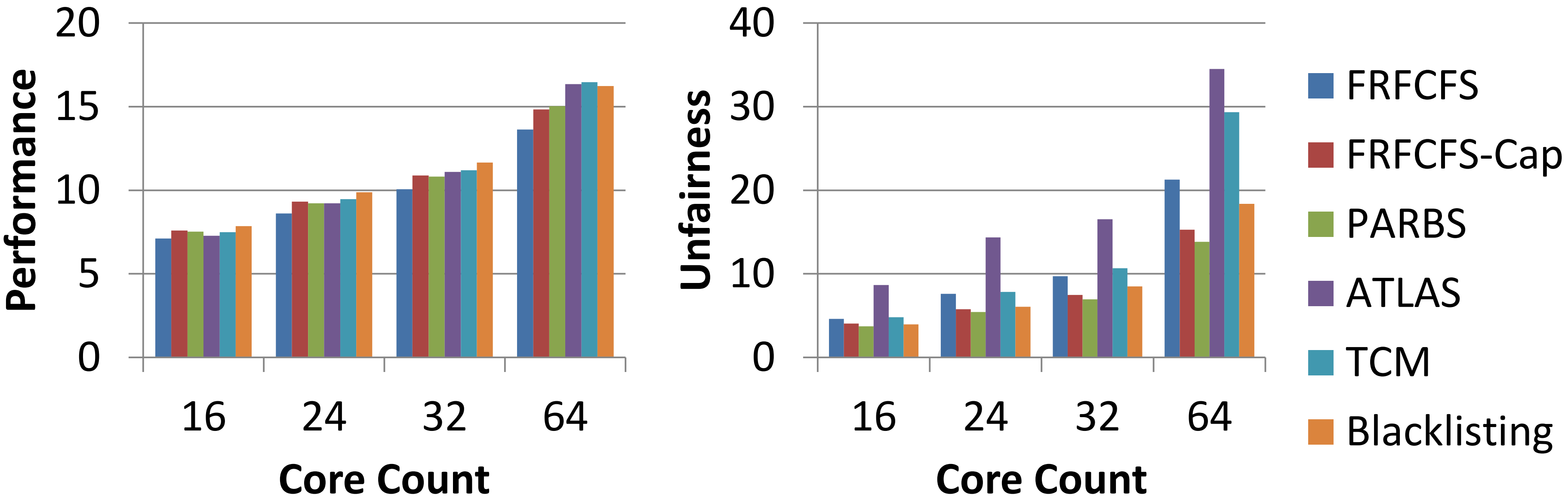
Sensitivity to Blacklisting Threshold



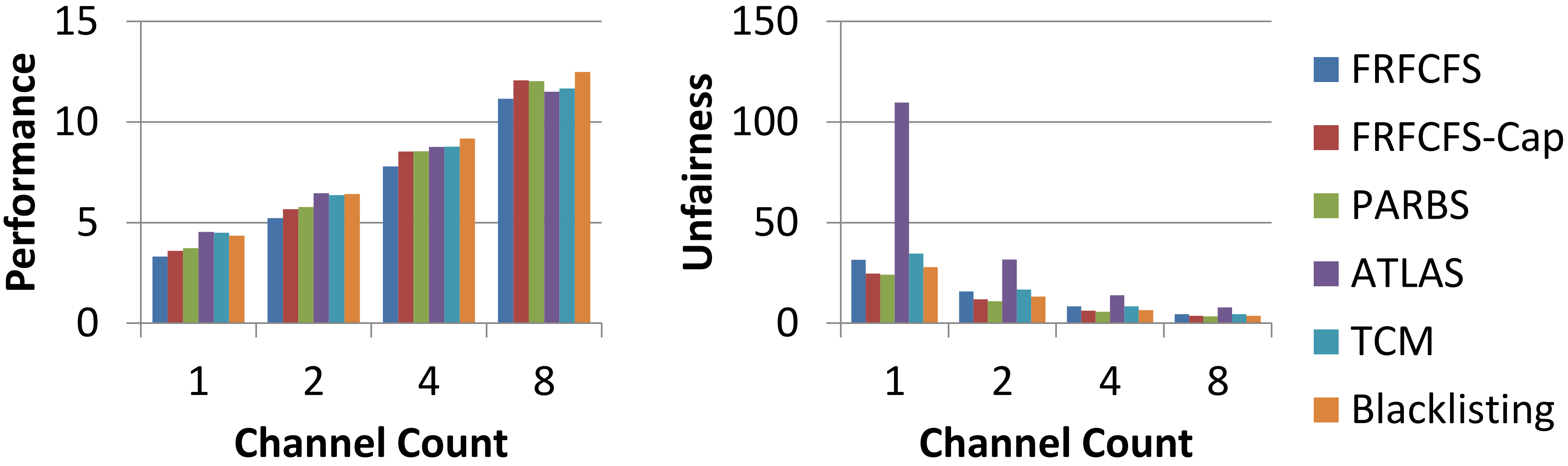
Sensitivity to Clearing Interval



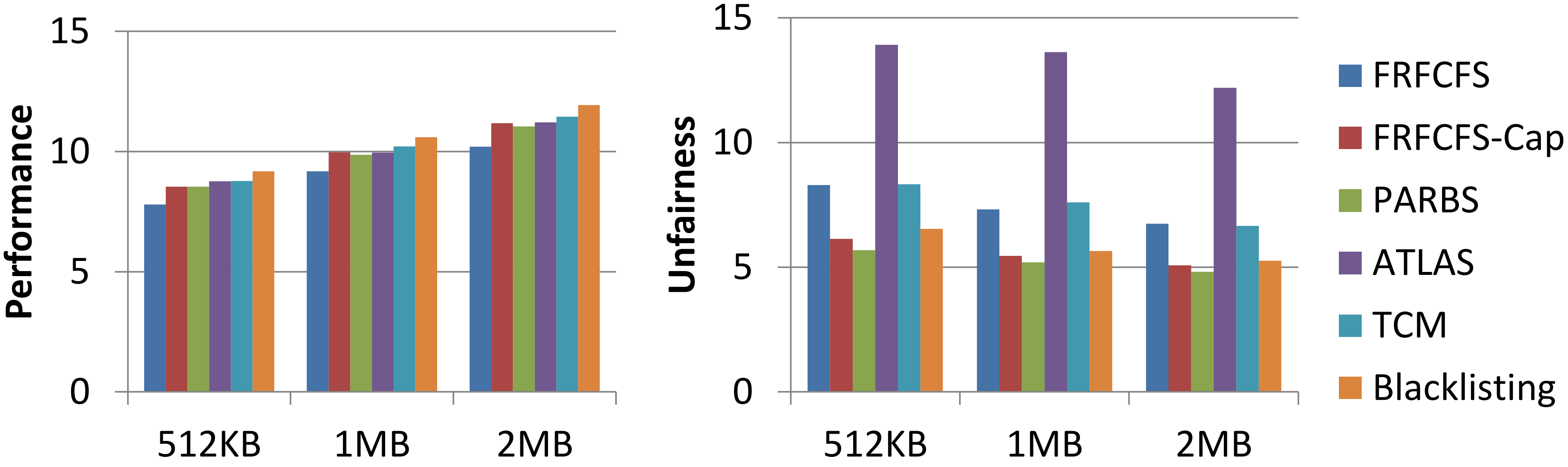
Sensitivity to Core Count



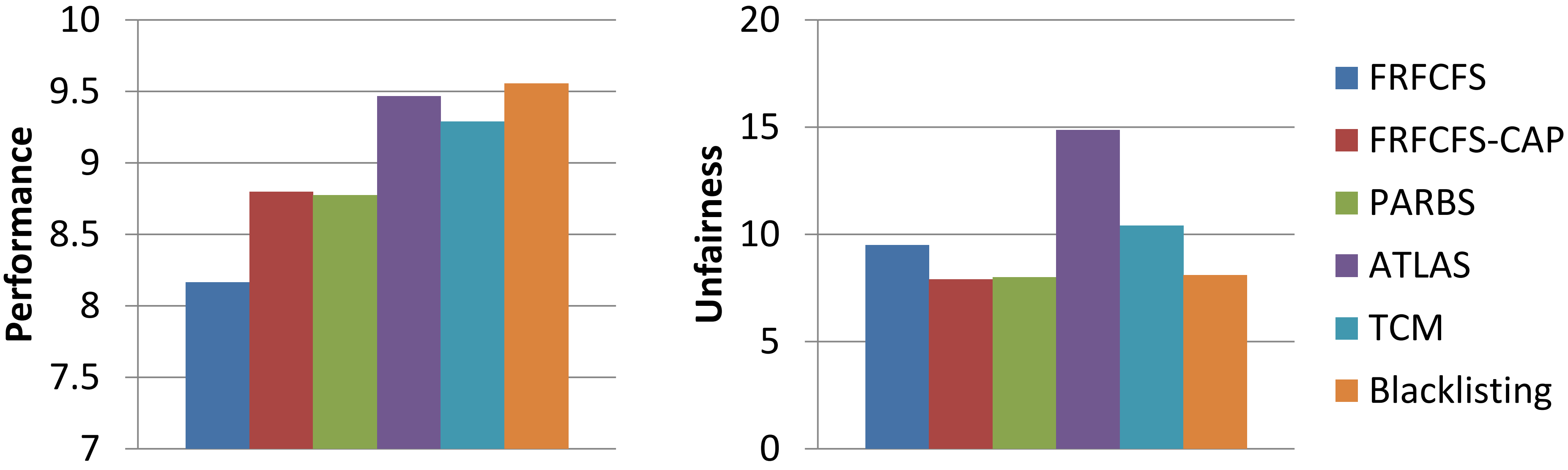
Sensitivity to Channel Count



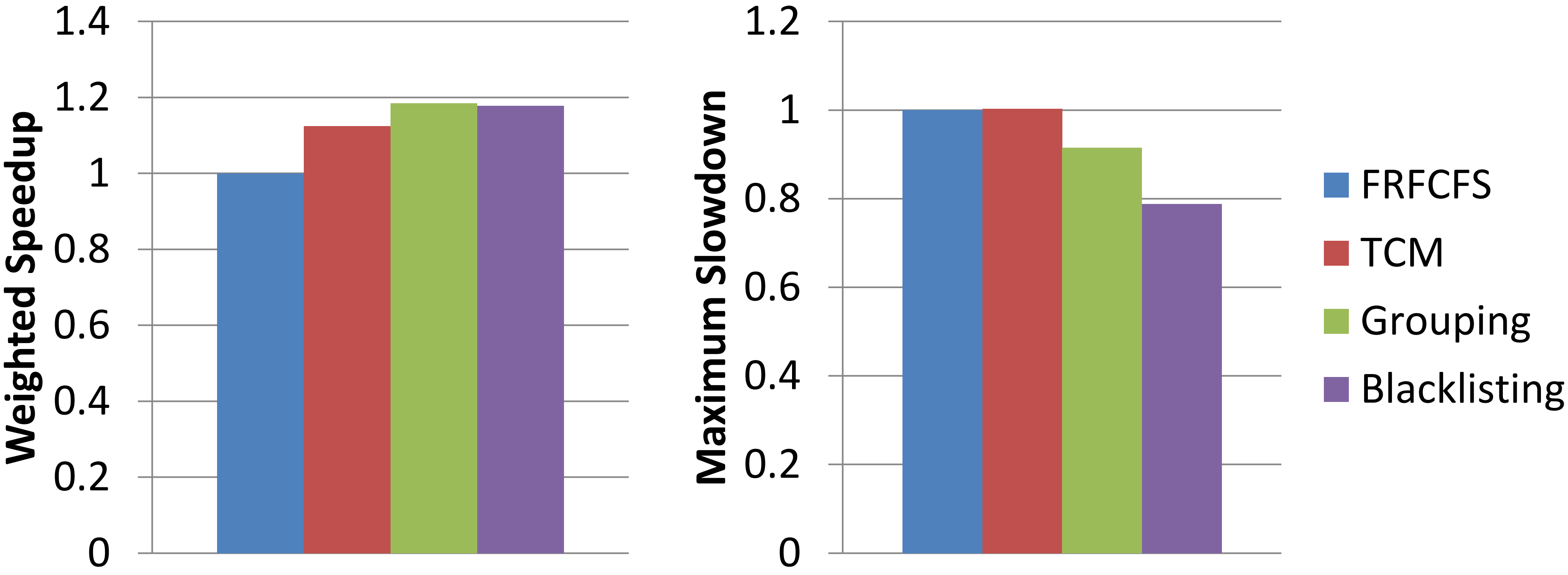
Sensitivity to Cache Size



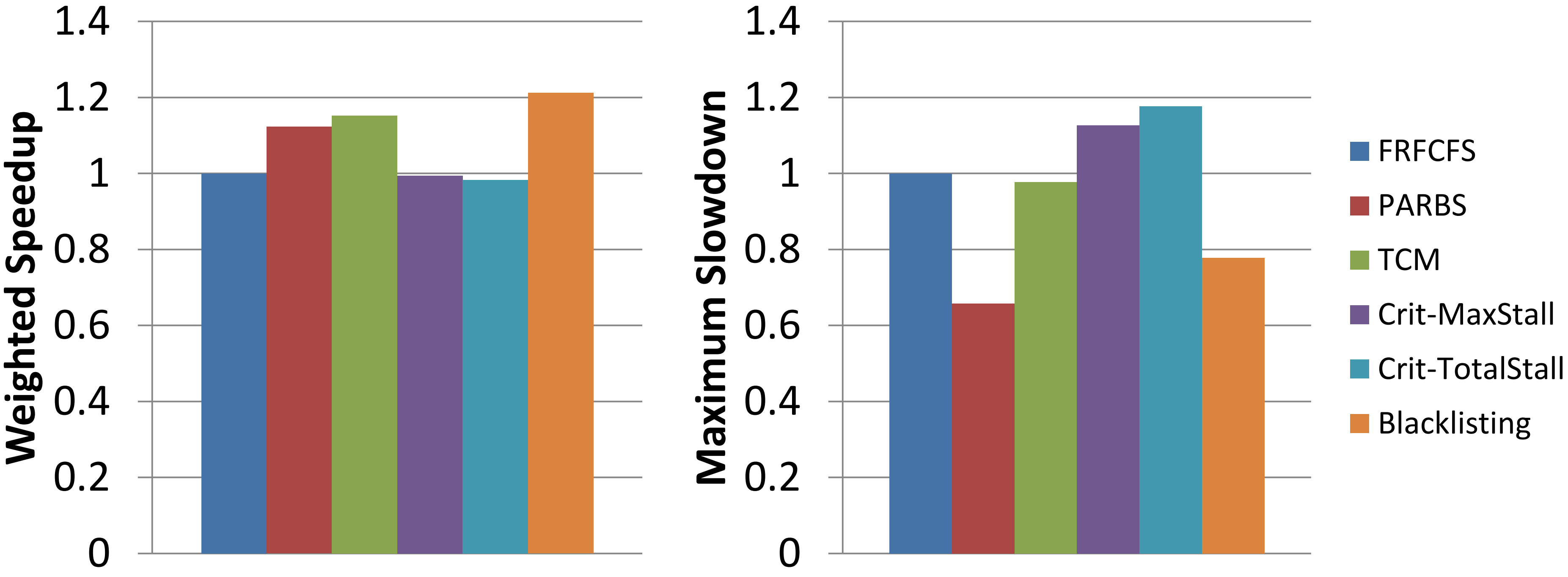
Performance and Fairness with Shared Cache



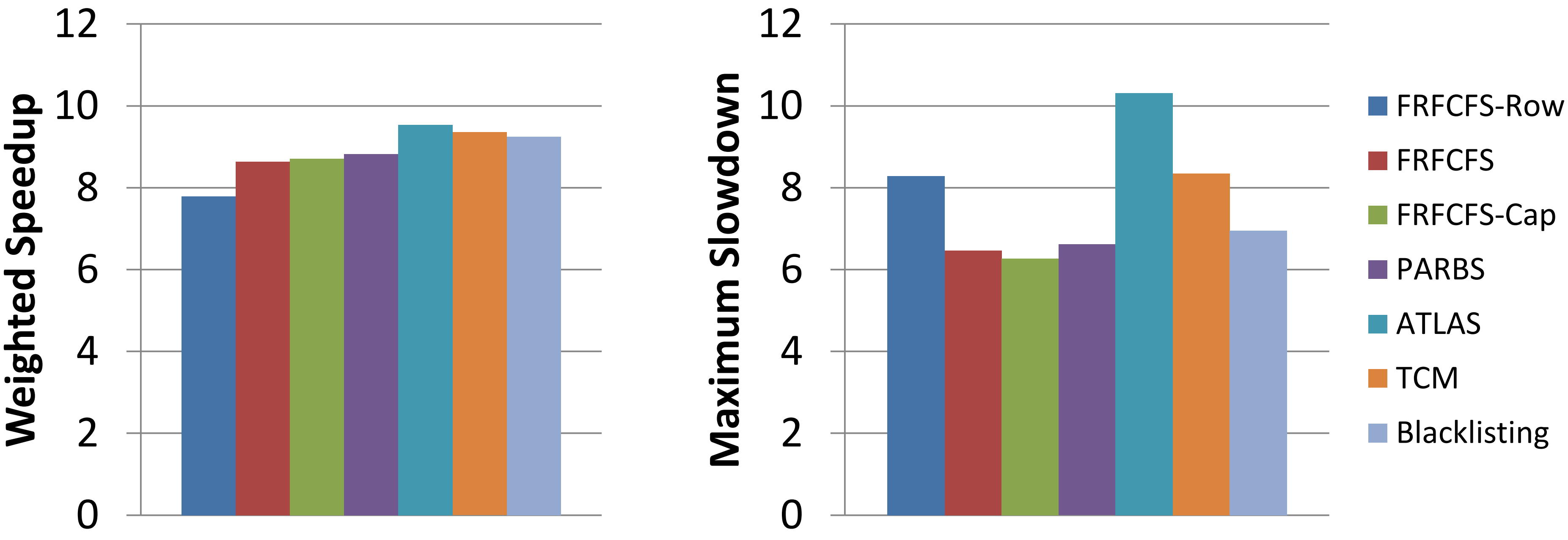
Breakdown of Benefits



BLISS vs. Criticality-aware Scheduling



Sub-row Interleaving



MISE

Measuring RSR_{Shared} and α

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

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

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

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

- Divide each interval into shorter epochs

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

- At the beginning of each epoch
  - Randomly pick an application as the highest priority application

How: Periodically give each application highest priority in accessing memory

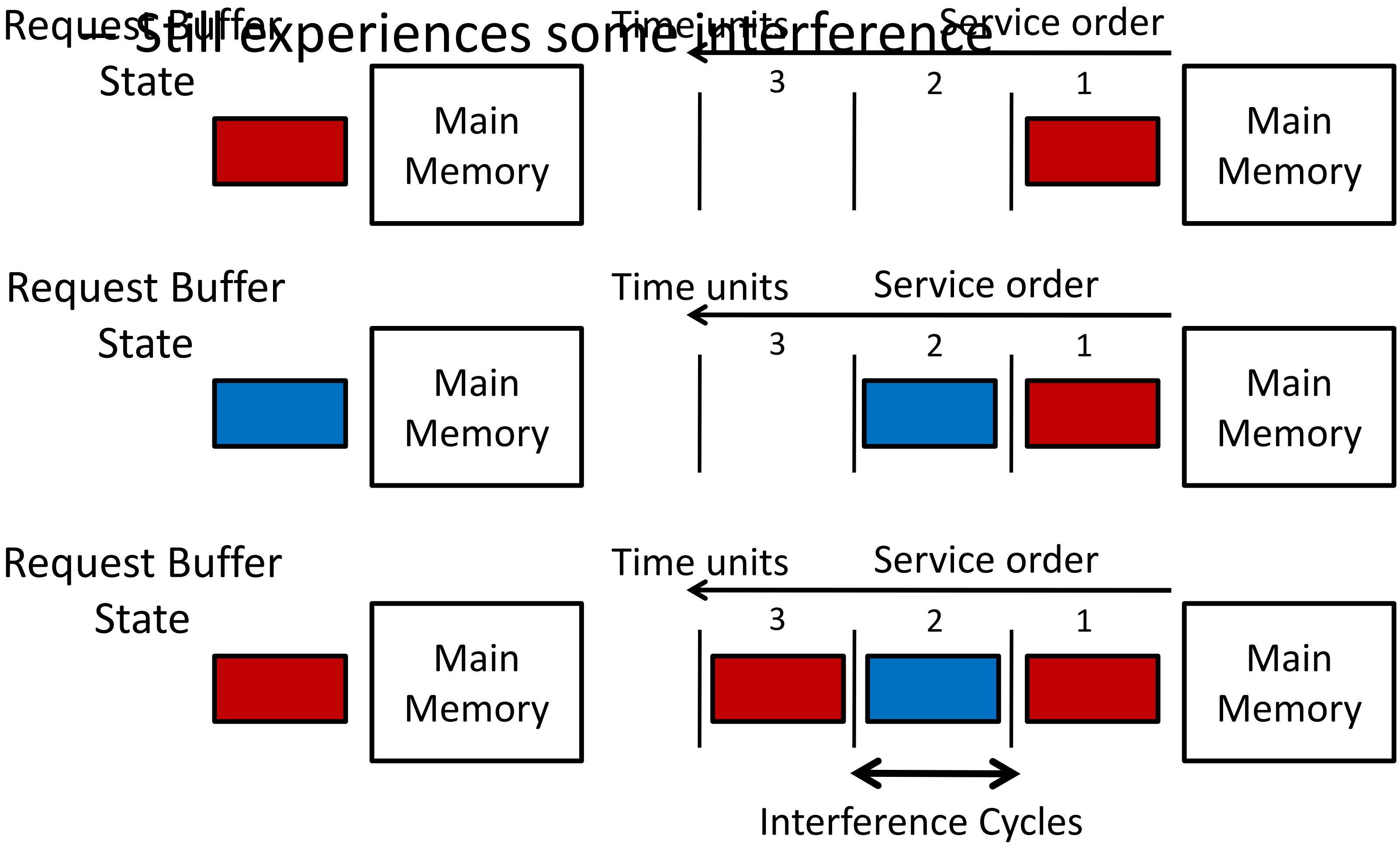
- At the end of an interval, for each application, estimate

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

Inaccuracy in Estimating RSR_{Alone}

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

🟥 High Priority



Request Buffer State

Time units

Service order

3 2 1

Main Memory

Main Memory

Request Buffer State

Time units

Service order

3 2 1

Main Memory

Main Memory

Request Buffer State

Time units

Service order

3 2 1

Main Memory

Main Memory

Interference Cycles

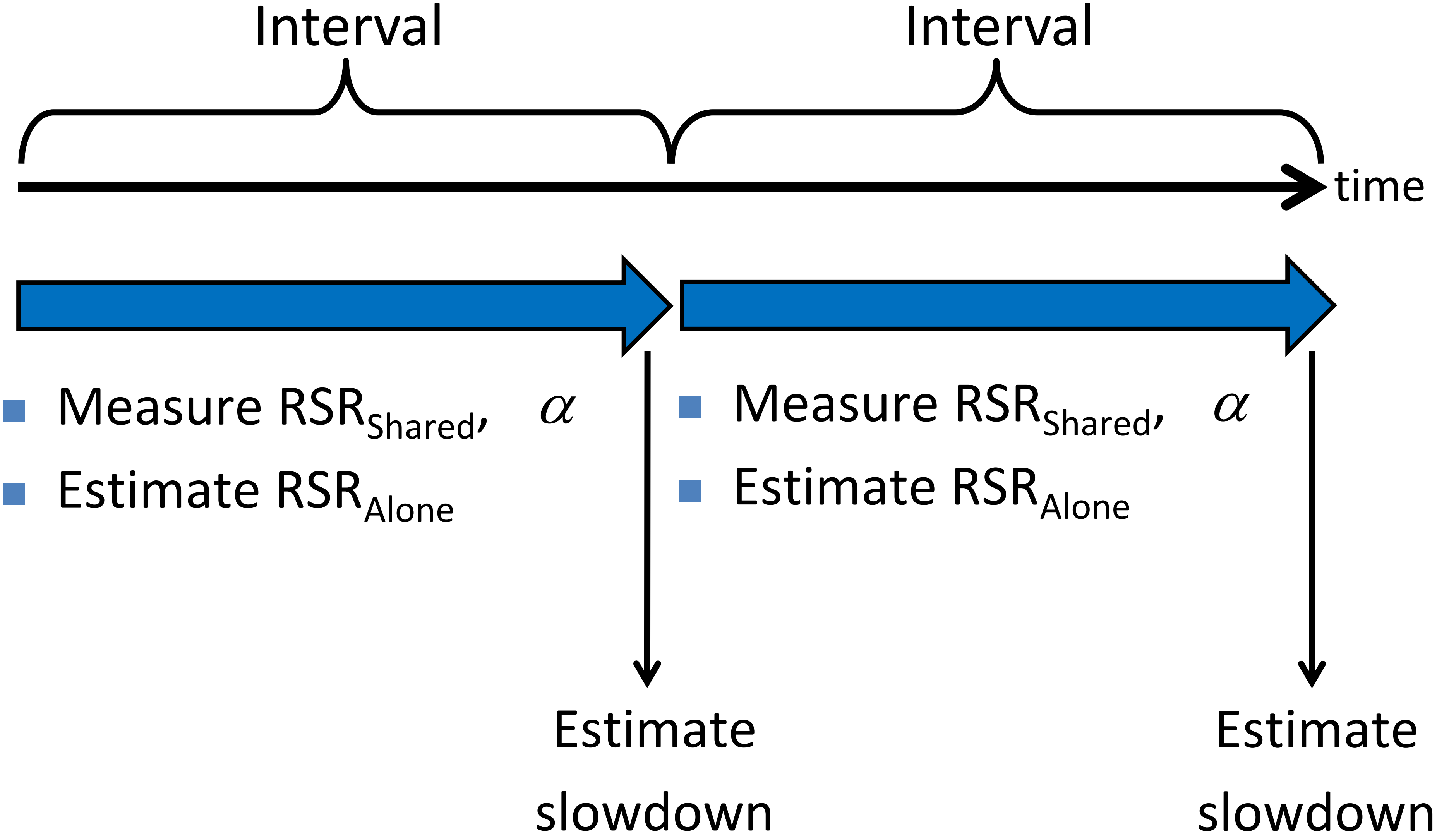
Accounting for Interference in $\text{RSR}_{\text{Alone}}$ Estimation

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

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

- A cycle is an interference cycle if
  - a request from the highest priority application is waiting in the request buffer *and*
  - another application's request was issued previously

MISE Operation: Putting it All Together



Interval

Interval

time

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

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

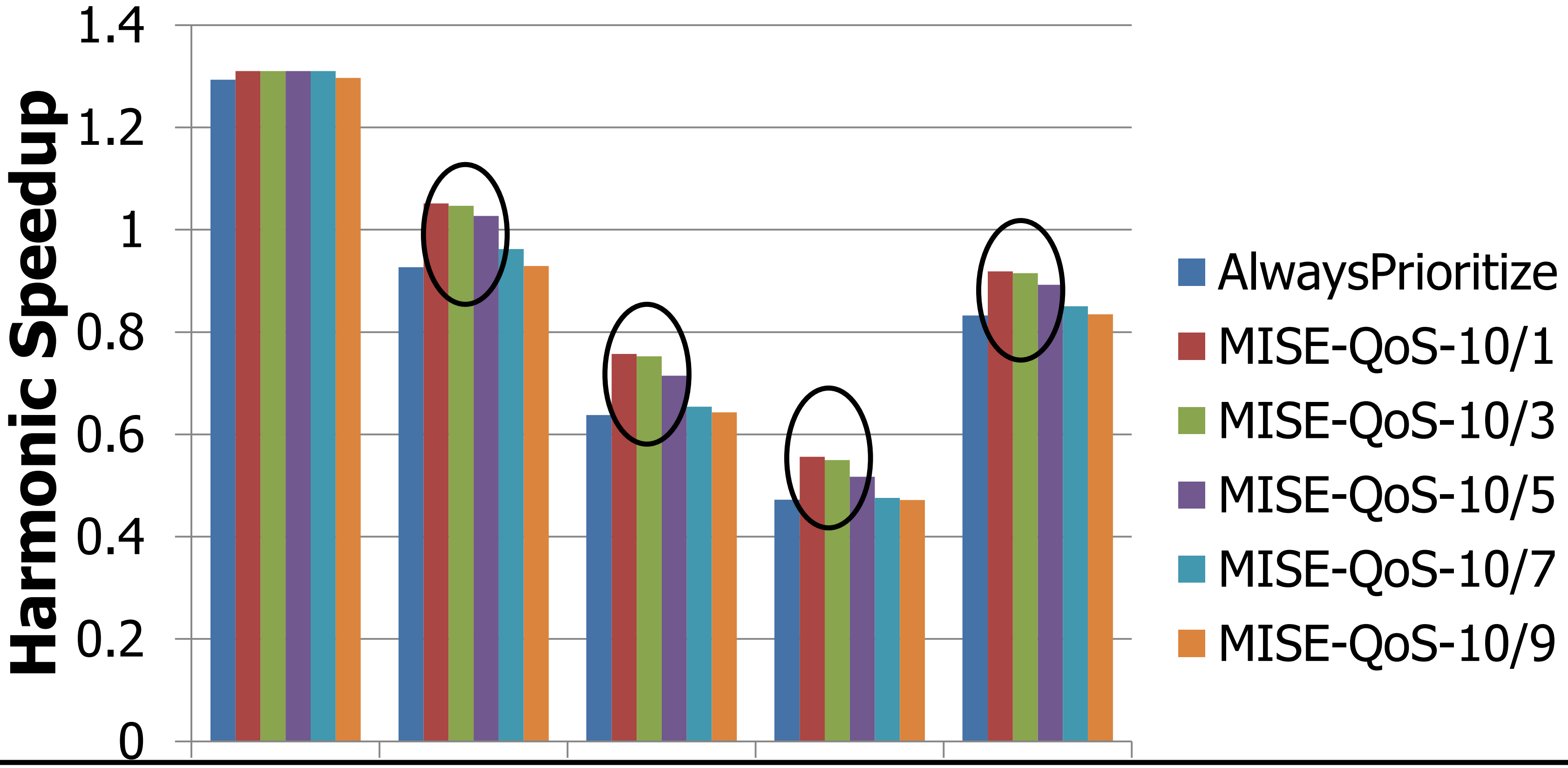
Estimate slowdown

Estimate slowdown

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

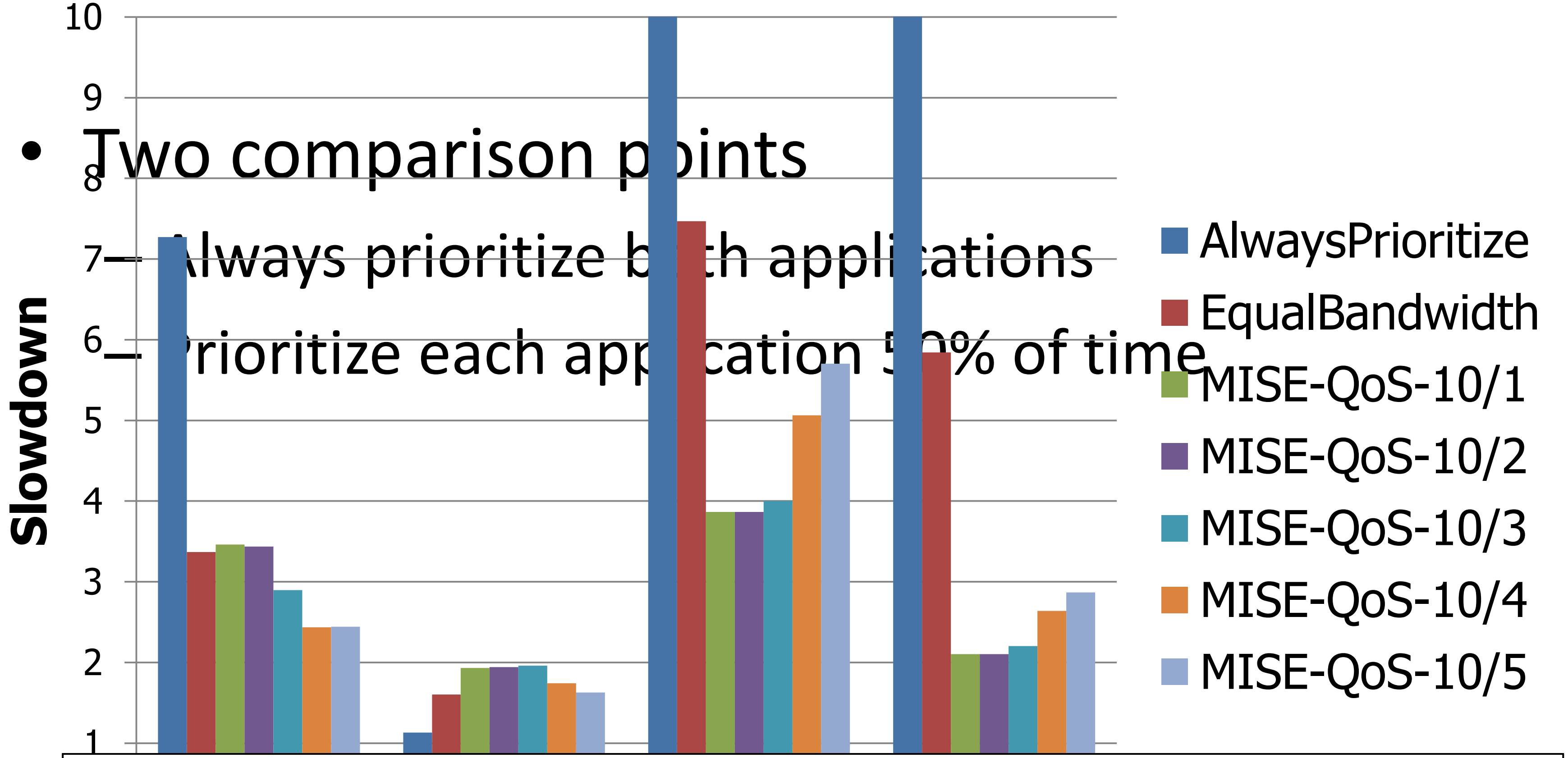
Performance of Non-QoS-Critical Applications



When slowdown bound is 10/3
MISE-QoS improves system performance by 10%

Case Study with Two QoS-Critical Applications

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



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

Minimizing Maximum Slowdown

- Goal
  - Minimize the maximum slowdown experienced by any application

- Basic Idea
  - Assign more memory bandwidth to the more slowed down application

Mechanism

- Memory controller tracks
  - Slowdown bound B
  - Bandwidth allocation of all applications

- Different components of mechanism
  - Bandwidth redistribution policy
  - Modifying target bound
  - Communicating target bound to OS periodically

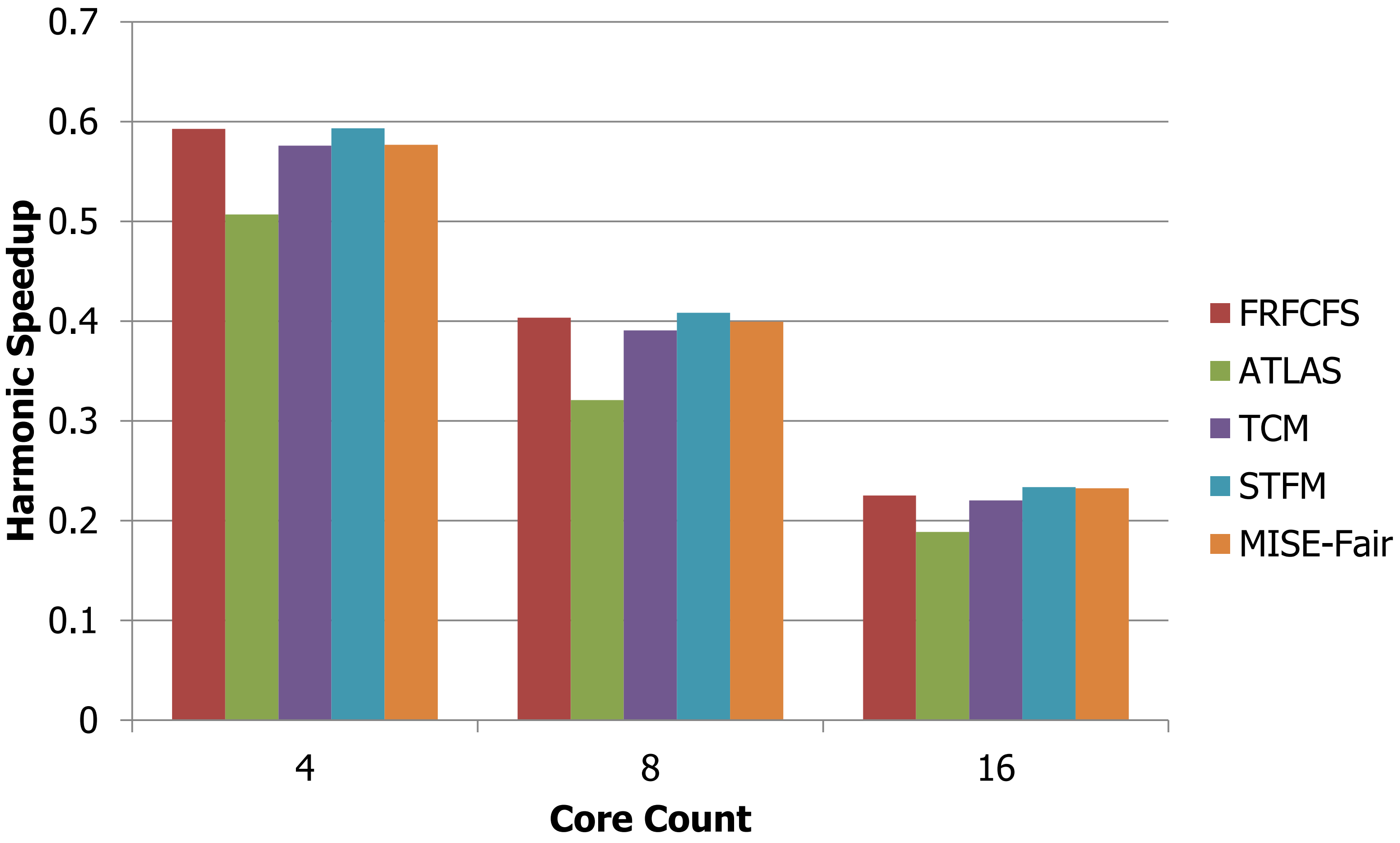
Bandwidth Redistribution

- At the end of each interval,
  - Group applications into two clusters
  - Cluster 1: applications that meet bound
  - Cluster 2: applications that don’t meet bound
  - Steal small amount of bandwidth from each application in cluster 1 and allocate to applications in cluster 2

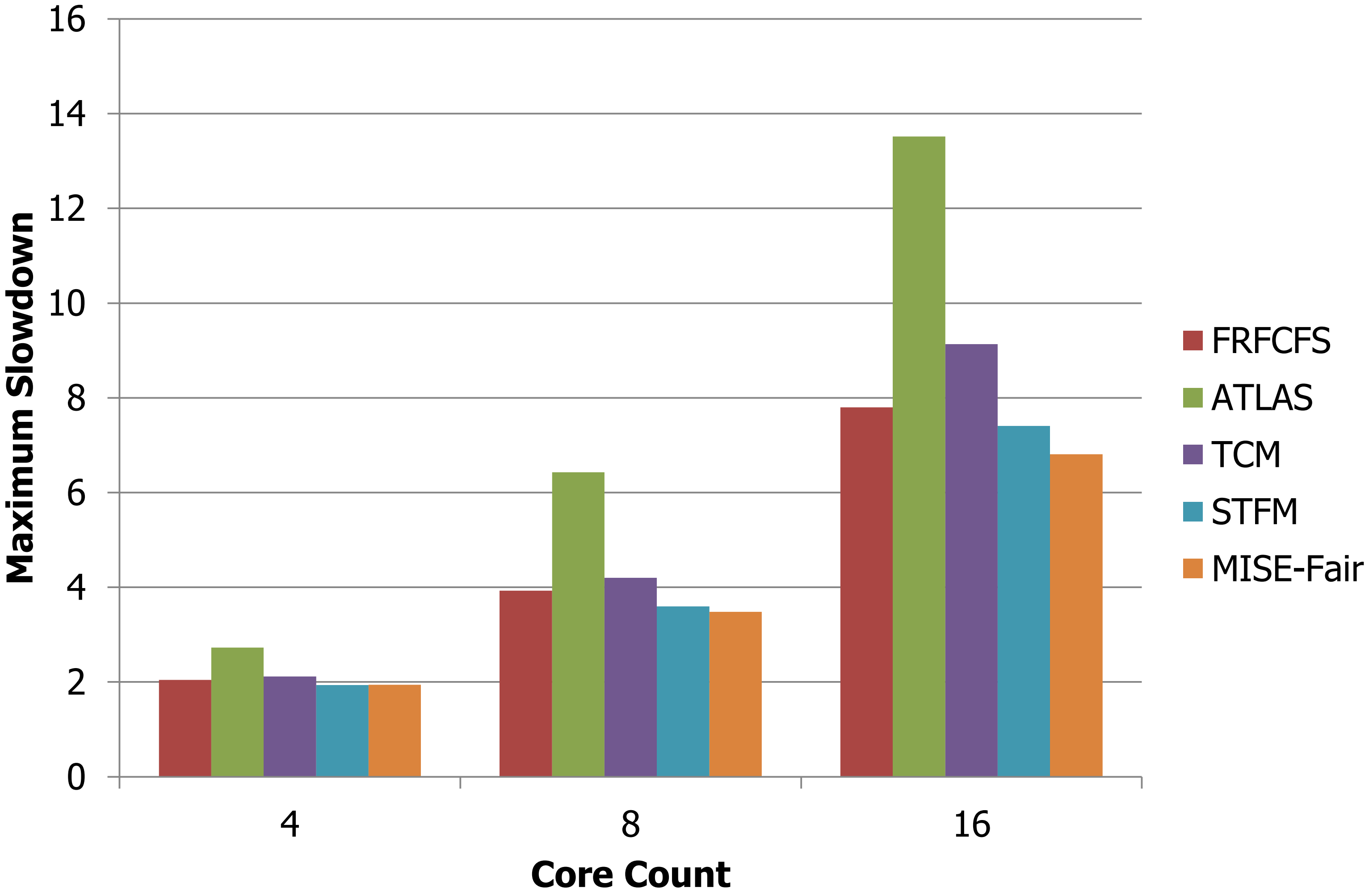
Modifying Target Bound

- If bound B is met for past N intervals
  - Bound can be made more aggressive
  - Set bound higher than the slowdown of most slowed down application
- If bound B not met for past N intervals by more than half the applications
  - Bound should be more relaxed
  - Set bound to slowdown of most slowed down application

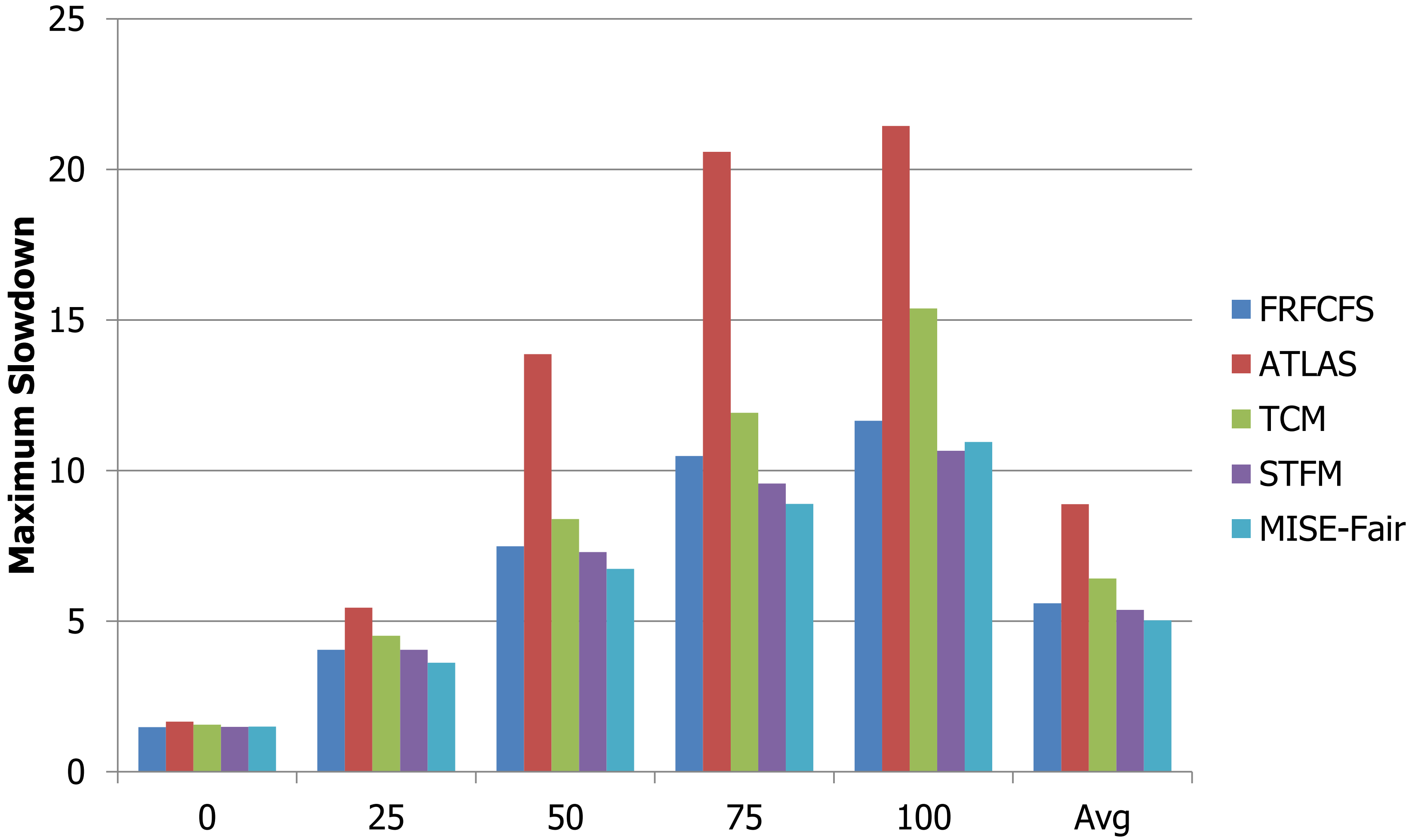
Results: Harmonic Speedup



Results: Maximum Slowdown



Sensitivity to Memory Intensity (16 cores)



MISE: Per-Application Error

Benchmark	STFM	MISE	Benchmark	STFM	MISE
453.povray	56.3	0.1	473.astar	12.3	8.1
454.calculix	43.5	1.3	456.hmmer	17.9	8.1
400.perlbench	26.8	1.6	464.h264ref	13.7	8.3
447.dealII	37.5	2.4	401.bzip2	28.3	8.5
436.cactusADM	18.4	2.6	458.sjeng	21.3	8.8
450.soplex	29.8	3.5	433.milc	26.4	9.5
444.namd	43.6	3.7	481.wrf	33.6	11.1
437.leslie3d	26.4	4.3	429.mcf	83.74	11.5
403.gcc	25.4	4.5	445.gobmk	23.1	12.5
462.libquantum	48.9	5.3	483.xalancbmk	18	13.6
459.GemsFDTD	21.6	5.5	435.gromacs	31.4	15.6
470.lbm	6.9	6.3	482.sphinx3	21	16.8
473.astar	12.3	8.1	471.omnetpp	26.2	17.5
456.hmmer	17.9	8.1	465.tonto	32.7	19.5

Sensitivity to Epoch and Interval Lengths

Interval Length

Epoch Length

	1 mil.	5 mil.	10 mil.	25 mil.	50 mil.
1000	65.1%	9.1%	11.5%	10.7%	8.2%
10000	64.1%	8.1%	9.6%	8.6%	8.5%
100000	64.3%	11.2%	9.1%	8.9%	9%
1000000	64.5%	31.3%	14.8%	14.9%	11.7%

Workload Mixes

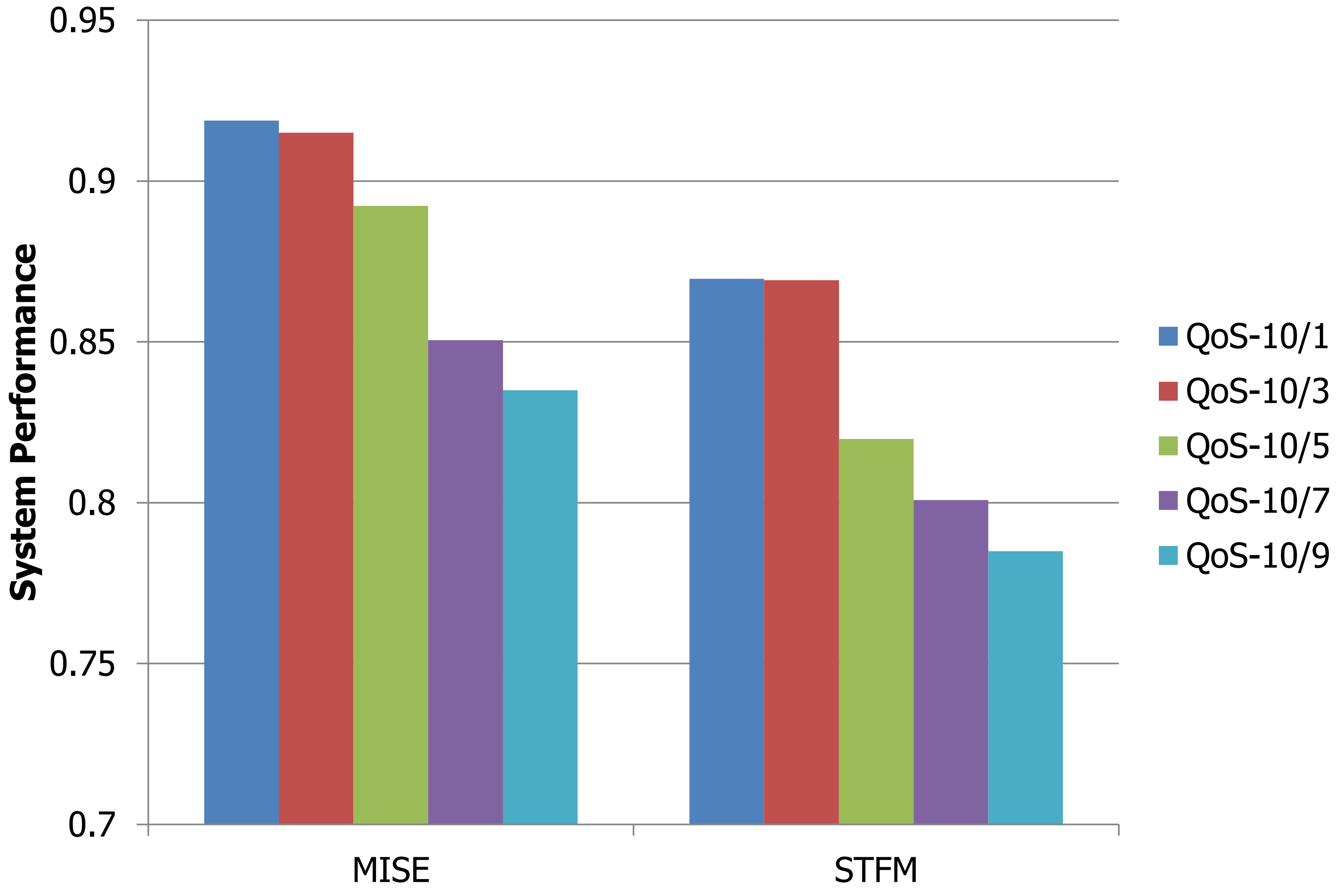
Mix No.	Benchmark 1	Benchmark 2	Benchmark 3
1	sphinx3	leslie3d	milc
2	sjeng	gcc	perlbench
3	tonto	povray	wrf
4	perlbench	gcc	povray
5	gcc	povray	leslie3d
6	perlbench	namd	lbm
7	h264ref	bzip2	libquantum
8	hmmer	lbm	omnetpp
9	sjeng	libquantum	cactusADM
10	namd	libquantum	mcf
11	xalancbmk	mcf	astar
12	mcf	libquantum	leslie3d

STFM's Effectiveness in Enforcing QoS

Across 3000 data points

	Predicted Met	Predicted Not Met
QoS Bound Met	63.7%	16%
QoS Bound Not Met	2.4%	17.9%

STFM vs. MISE's System Performance



MISE’s Implementation Cost

1. Per-core counters worth 20 bytes
- Request Service Rate Shared
- Request Service Rate Alone
  - 1 counter for number of high priority epoch requests
  - 1 counter for number of high priority epoch cycles
  - 1 counter for interference cycles
- Memory phase fraction (α)

2. Register for current bandwidth allocation – 4 bytes
3. Logic for prioritizing an application in each epoch

MISE Accuracy w/o Interference Cycles

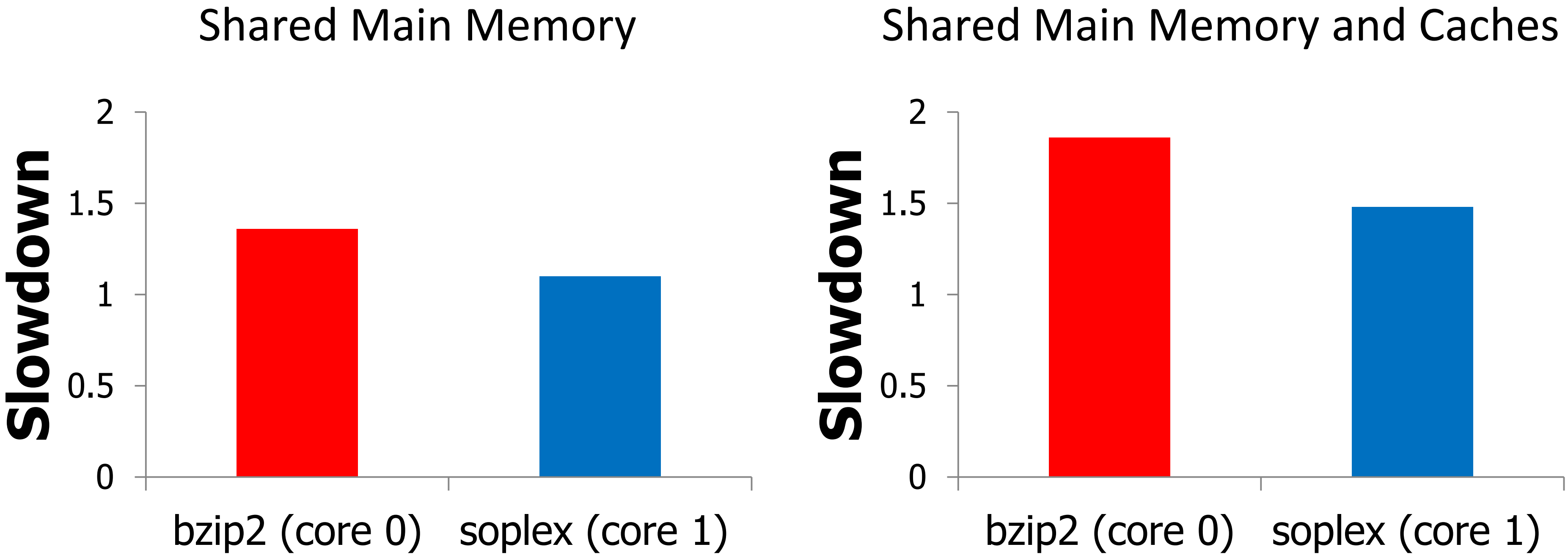
- Average error – 23%

MISE Average Error by Workload Category

Workload Category (Number of memory intensive applications)	**Average Error**
0	4.3%
1	8.9%
2	21.2%
3	18.4%

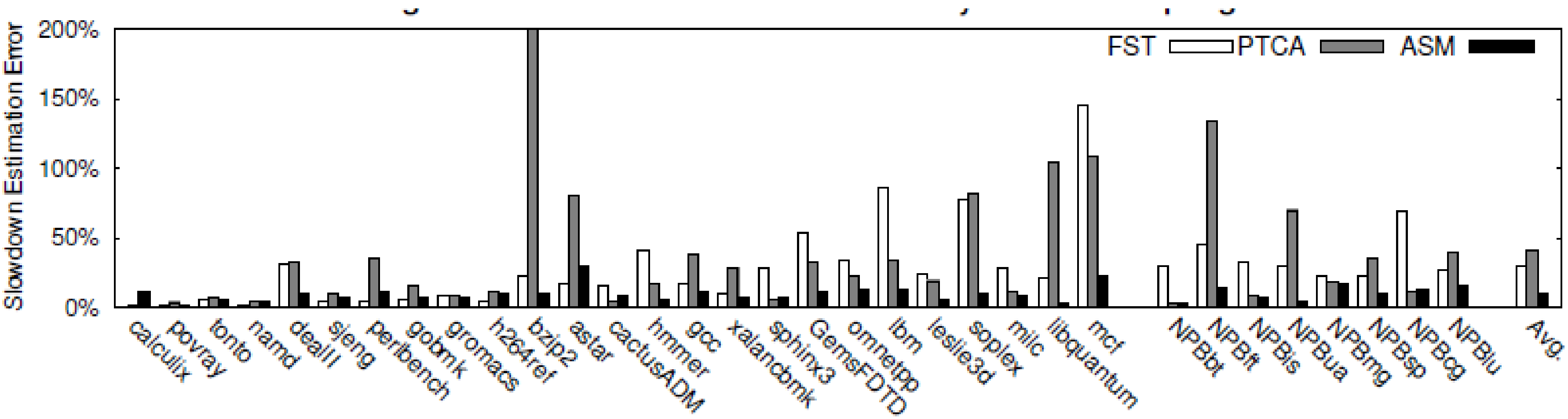
ASM

Impact of Cache Capacity Contention

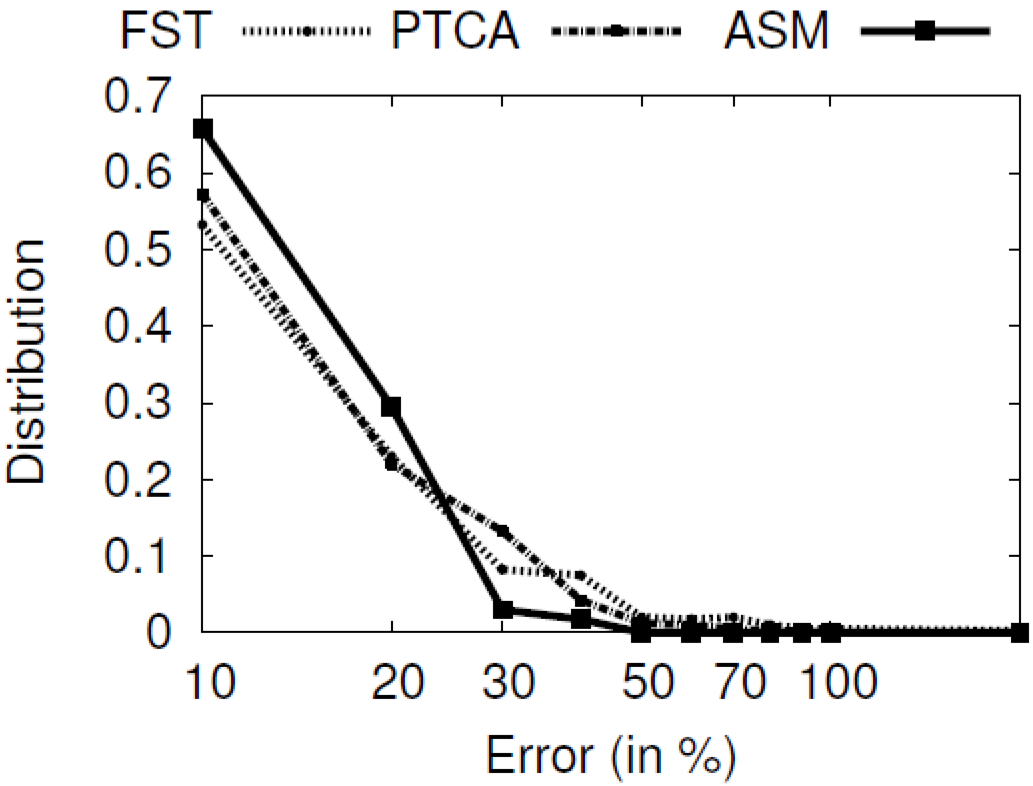


Cache capacity interference causes high application slowdowns

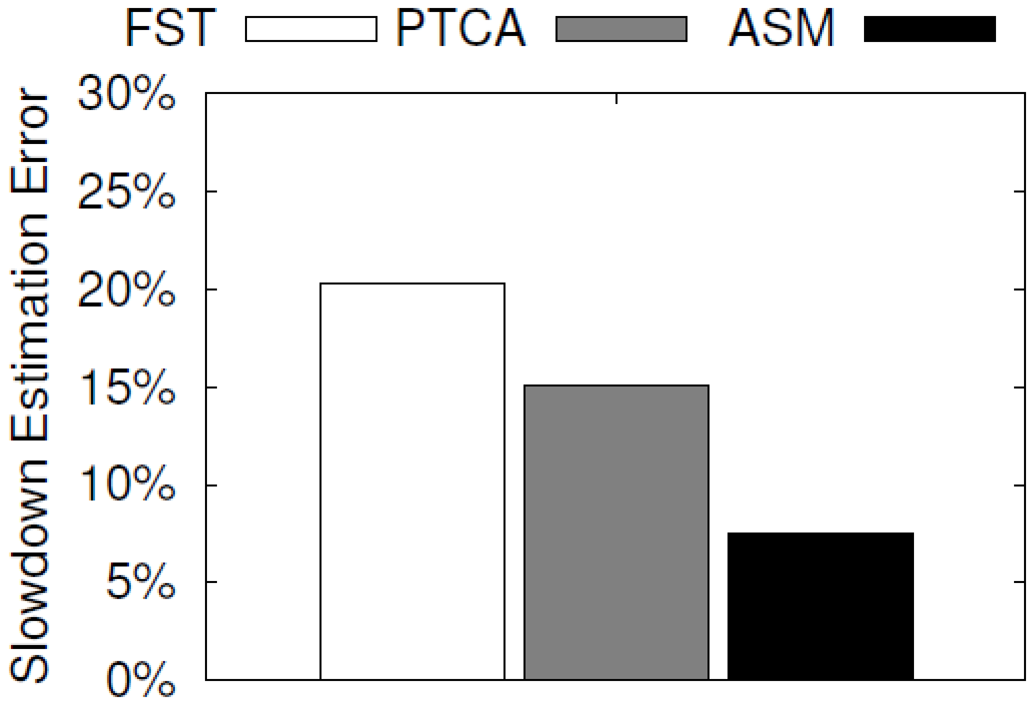
Error with Sampling



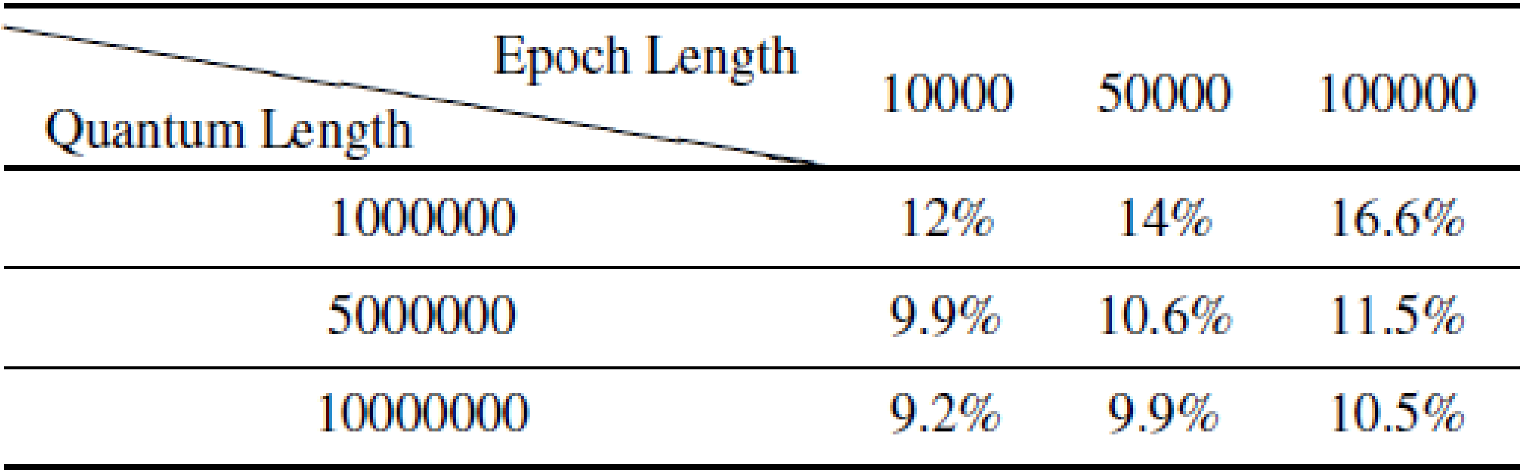
Error Distribution



Impact of Prefetching

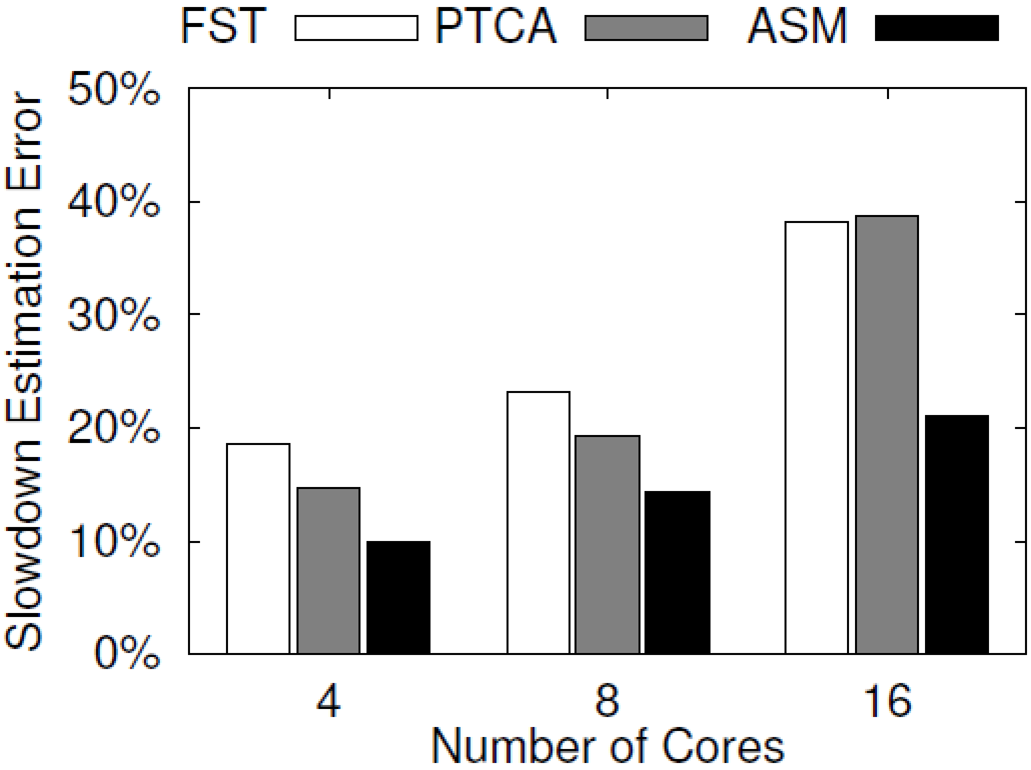


Sensitivity to Epoch and Quantum Lengths

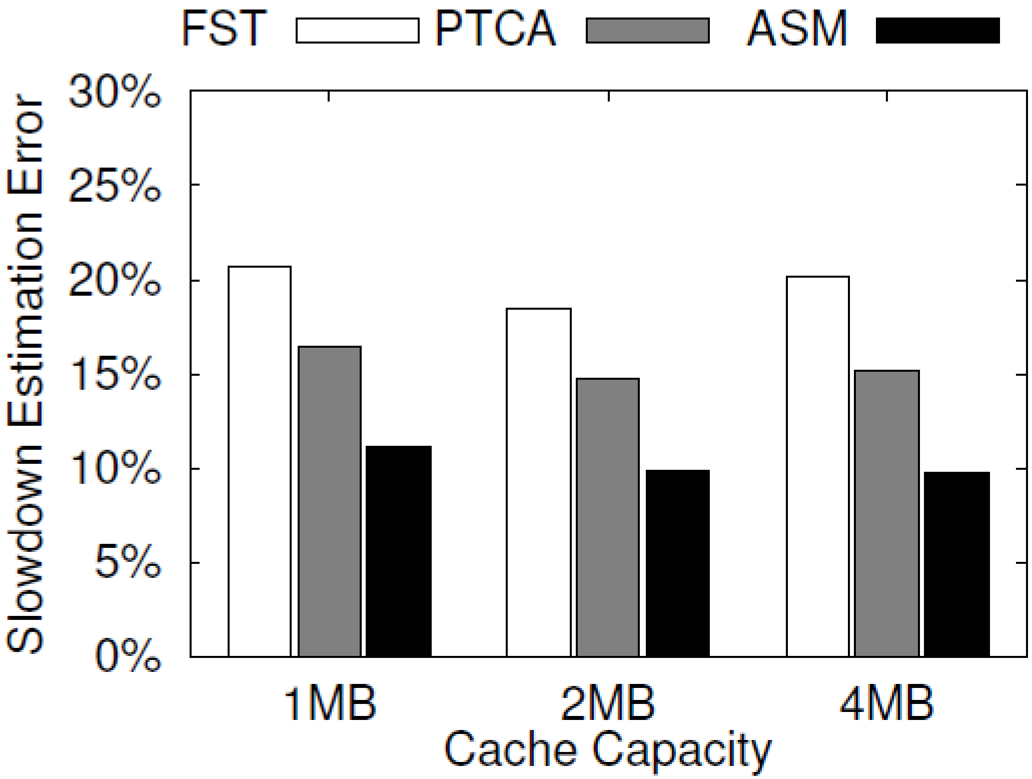


Epoch Length Quantum Length	10000	50000	100000
1000000	12%	14%	16.6%
5000000	9.9%	10.6%	11.5%
10000000	9.2%	9.9%	10.5%

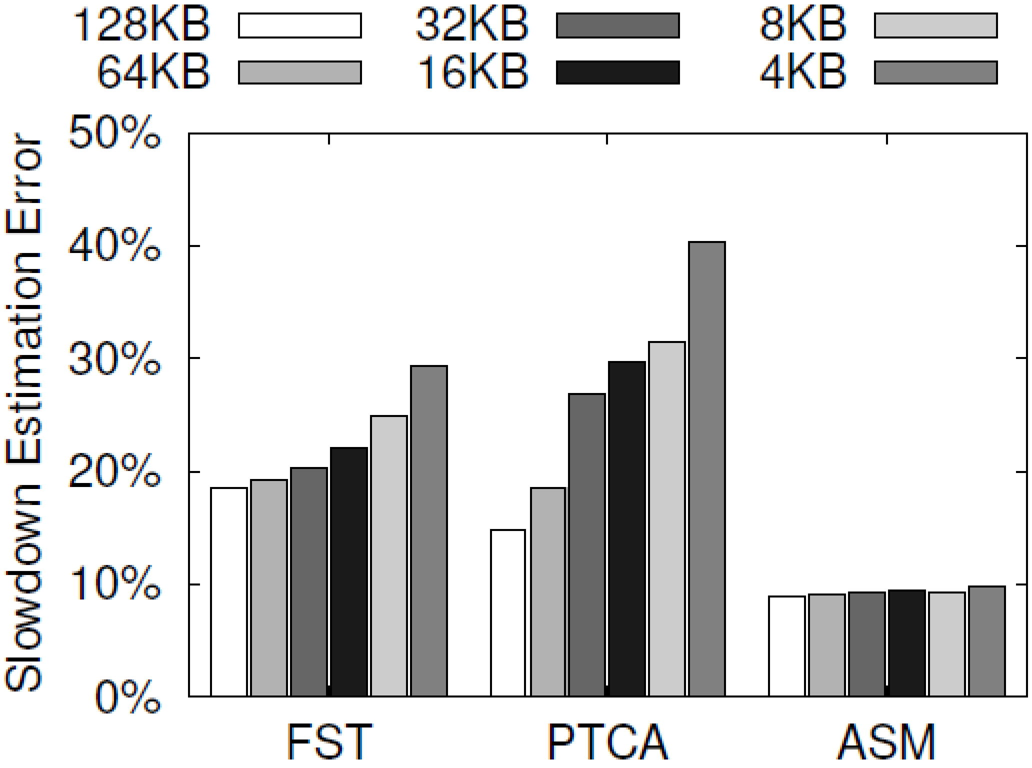
Sensitivity to Core Count



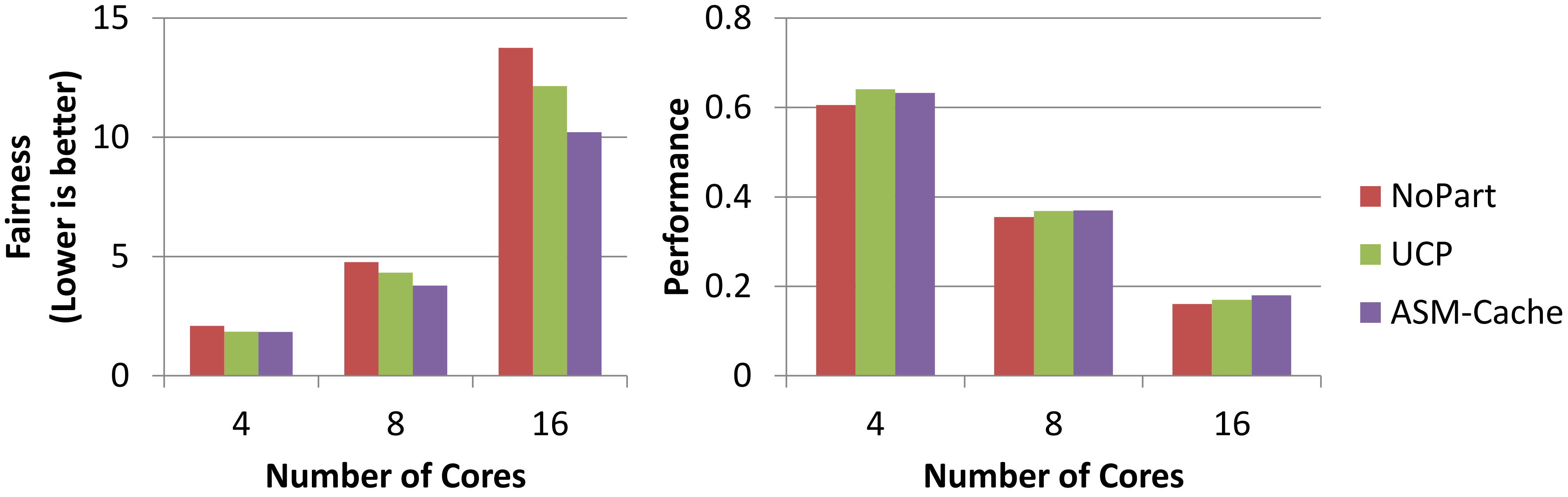
Sensitivity to Cache Capacity



Sensitivity to Auxiliary Tag Store Sampling

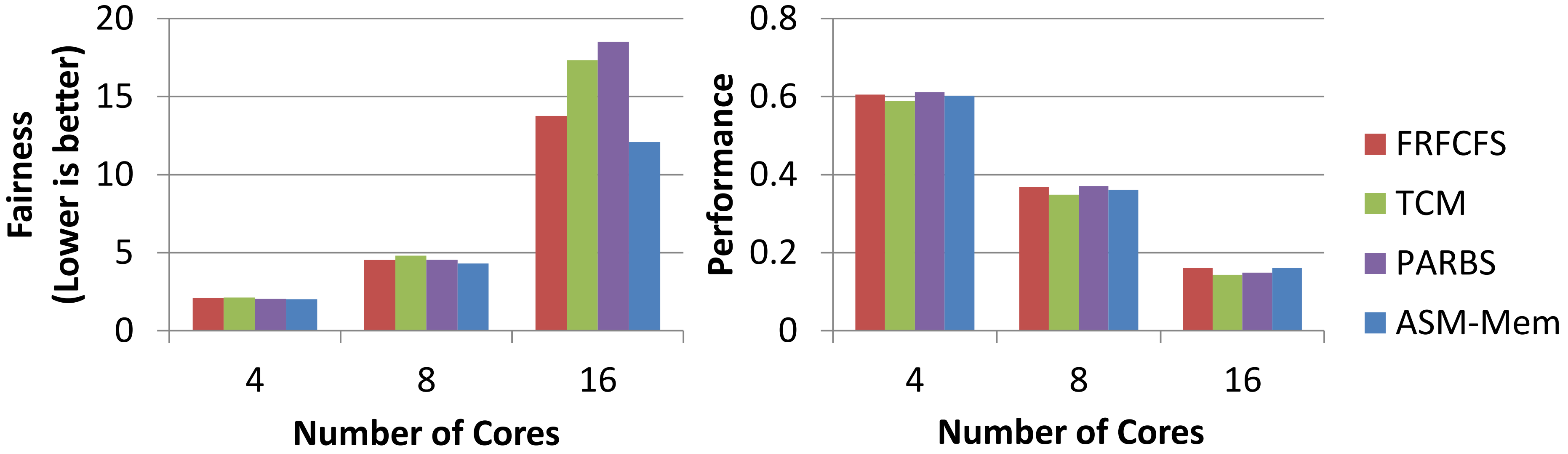


ASM-Cache:
Fairness and Performance Results



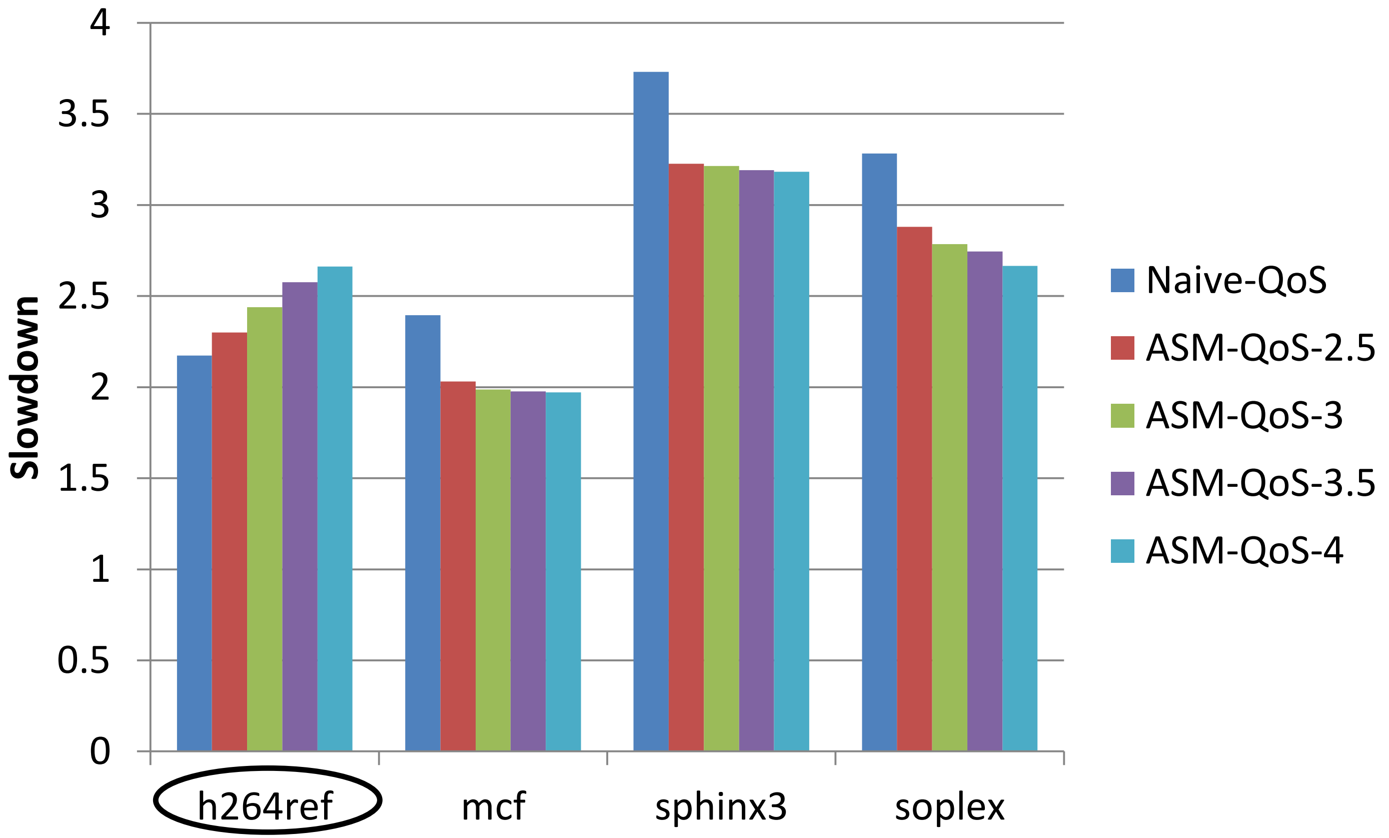
Significant fairness benefits across different systems

ASM-Mem:
Fairness and Performance Results

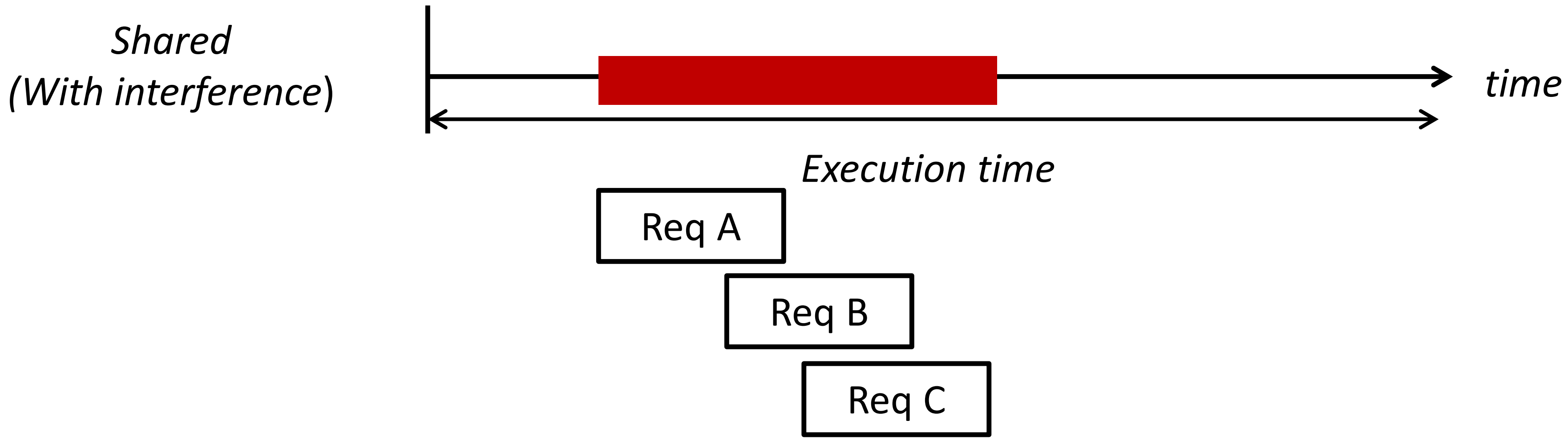


Significant fairness benefits across different systems

ASM-QoS: Meeting Slowdown Bounds

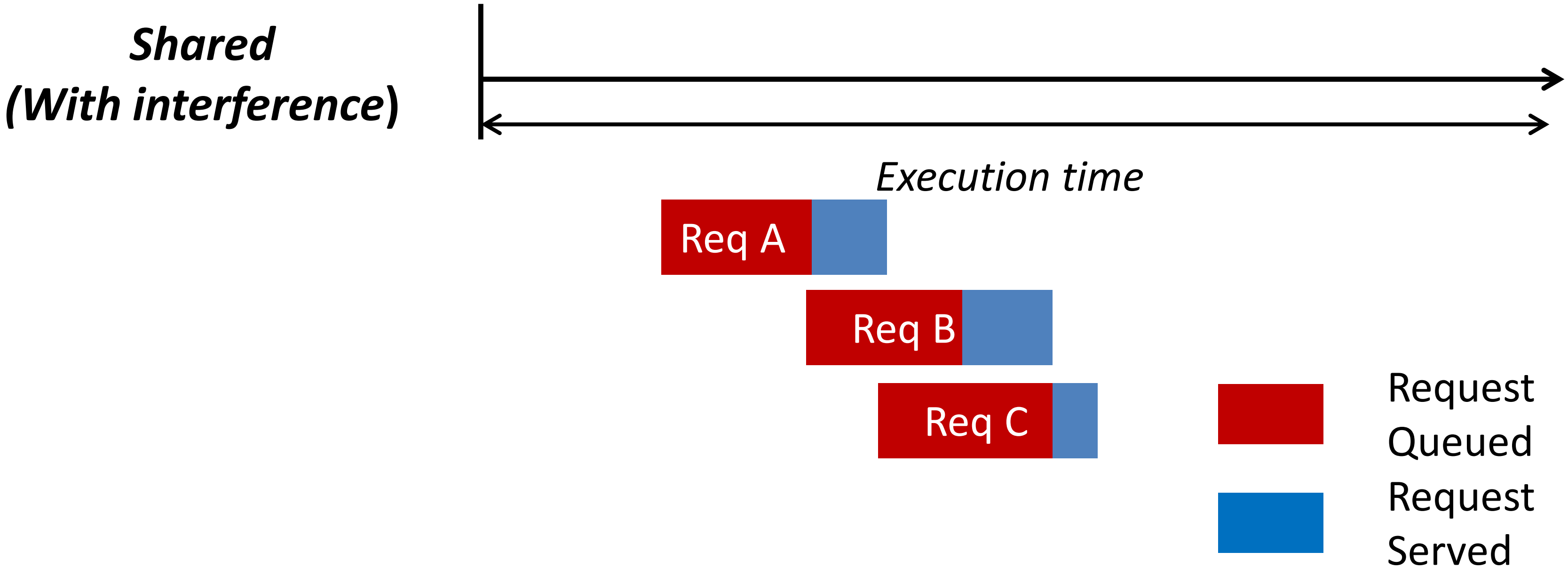


Previous Approach: Estimate Interference Experienced Per-Request



Request Overlap Makes Interference Estimation Per-Request Difficult

Estimating Performance$_{Alone}$



Shared
(With interference)

Execution time

Req A

Req B

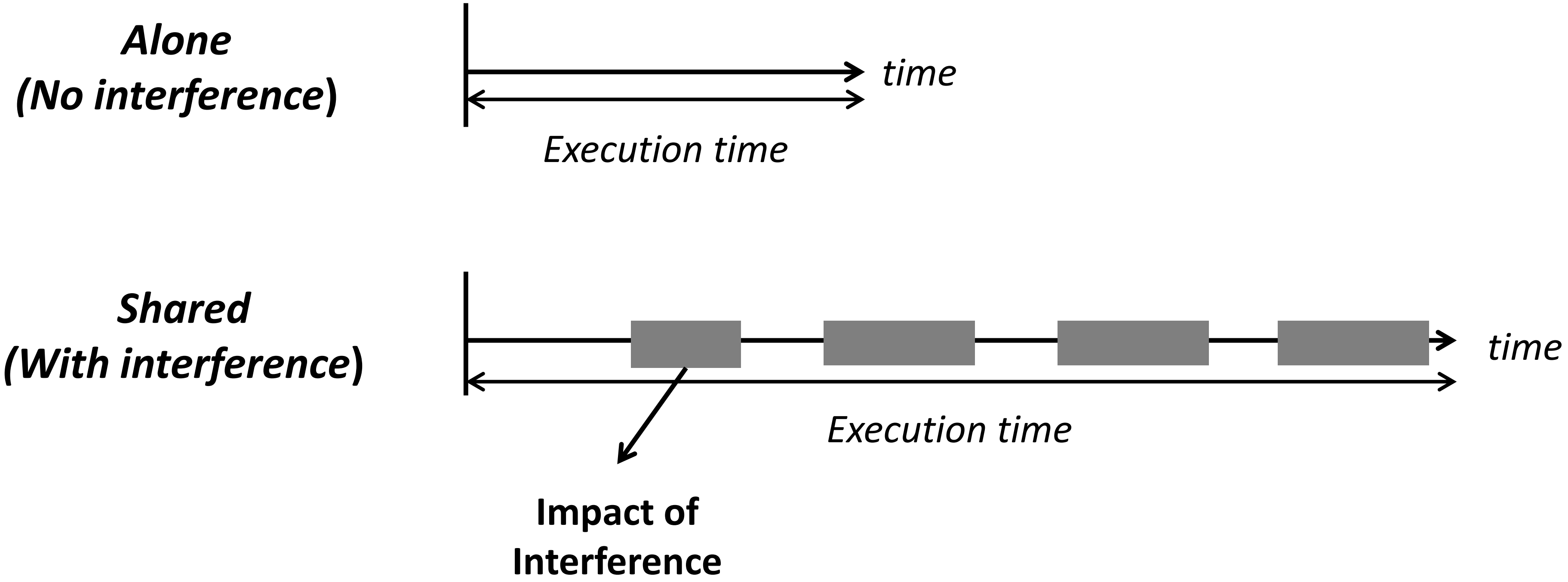
Req C

🟥 Request Queued

🟦 Request Served

Difficult to estimate impact of interference per-request due to request overlap

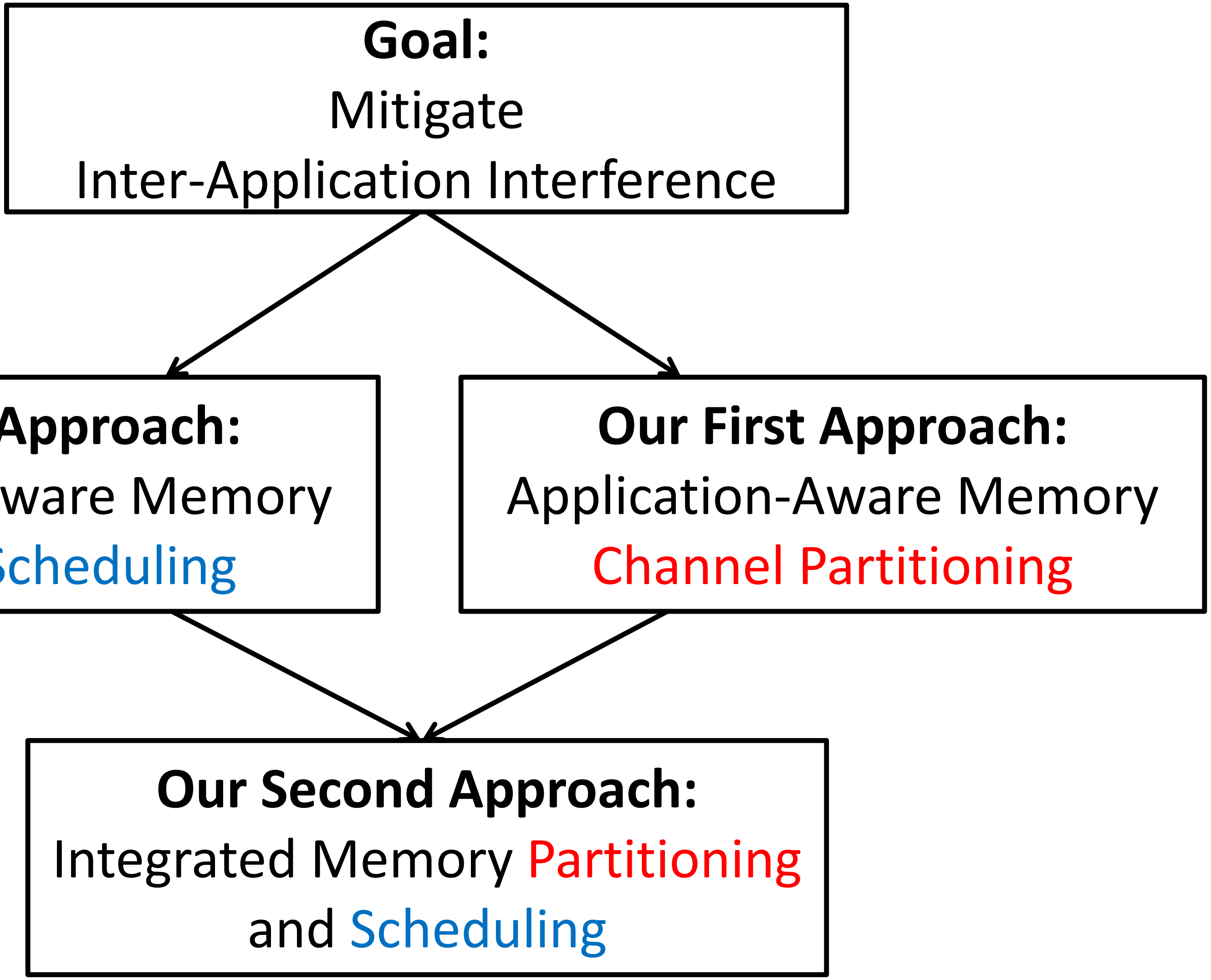
Impact of Interference on Performance



Previous Approach: Estimate impact of interference at a per-request granularity

Difficult to estimate due to request overlap

Application-aware Memory Channel Partitioning



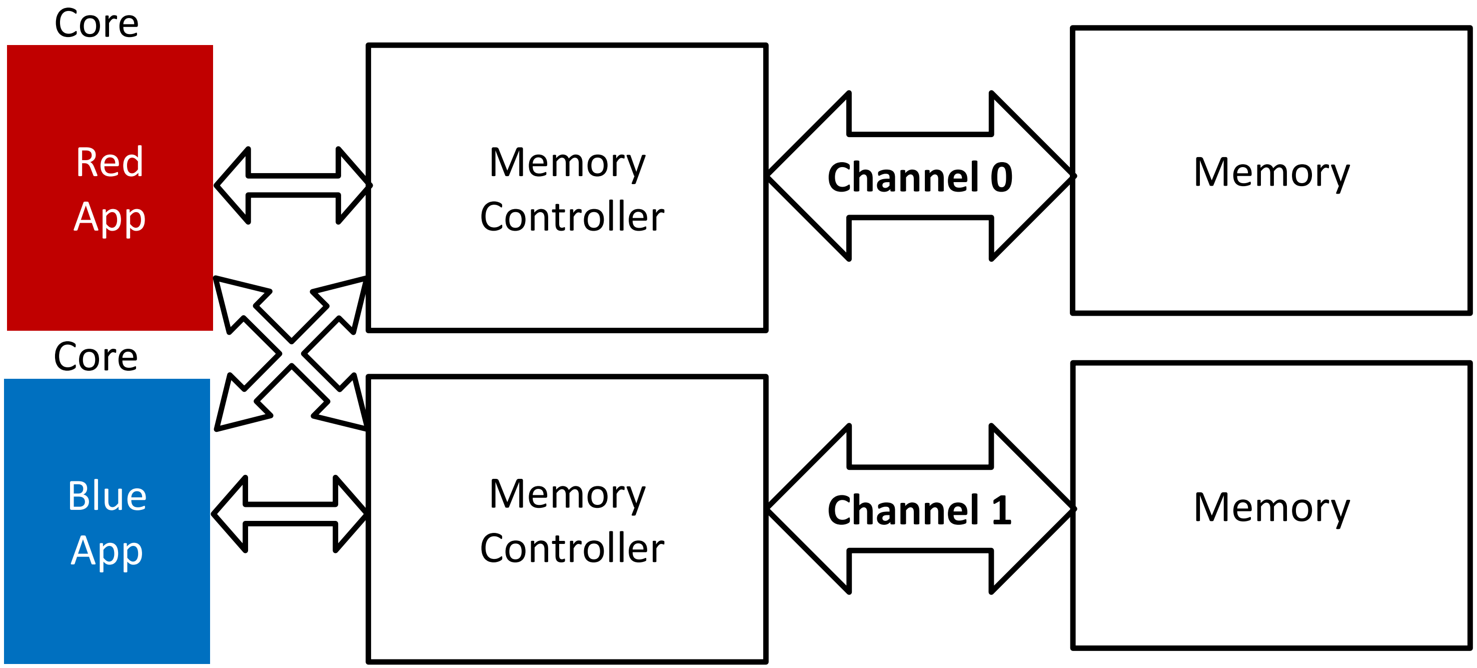
Goal:
Mitigate
Inter-Application Interference

Previous Approach:
Application-Aware Memory Request Scheduling

Our First Approach:
Application-Aware Memory Channel Partitioning

Our Second Approach:
Integrated Memory Partitioning and Scheduling

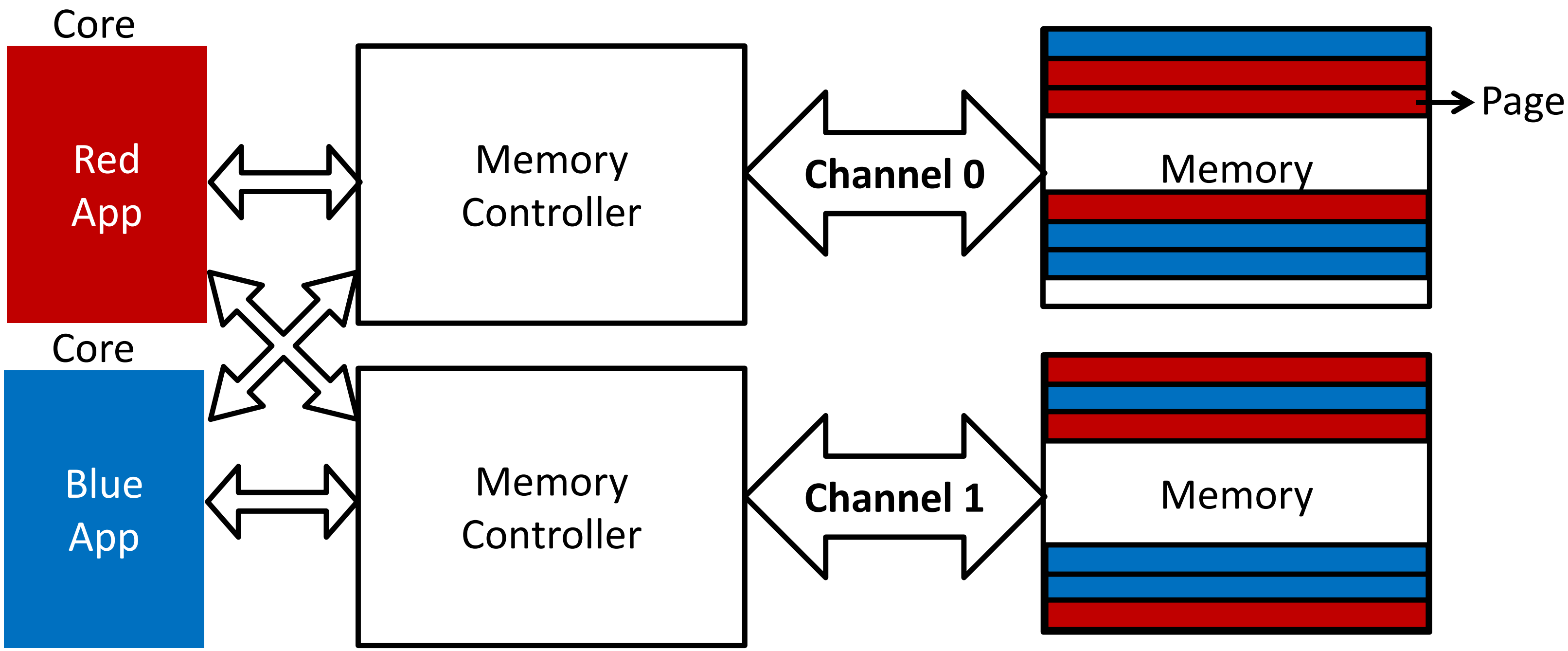
Observation: Modern Systems Have Multiple Channels



A new degree of freedom

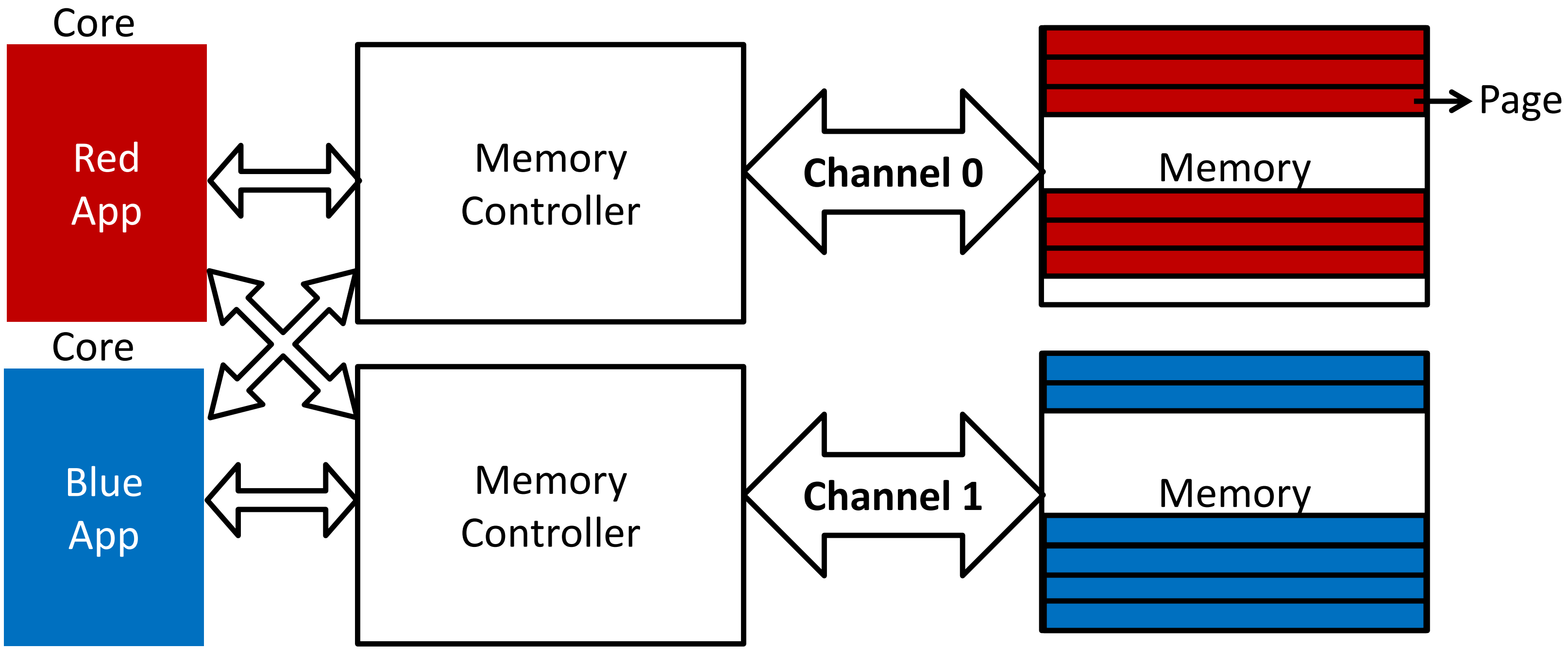
Mapping data across multiple channels

Data Mapping in Current Systems



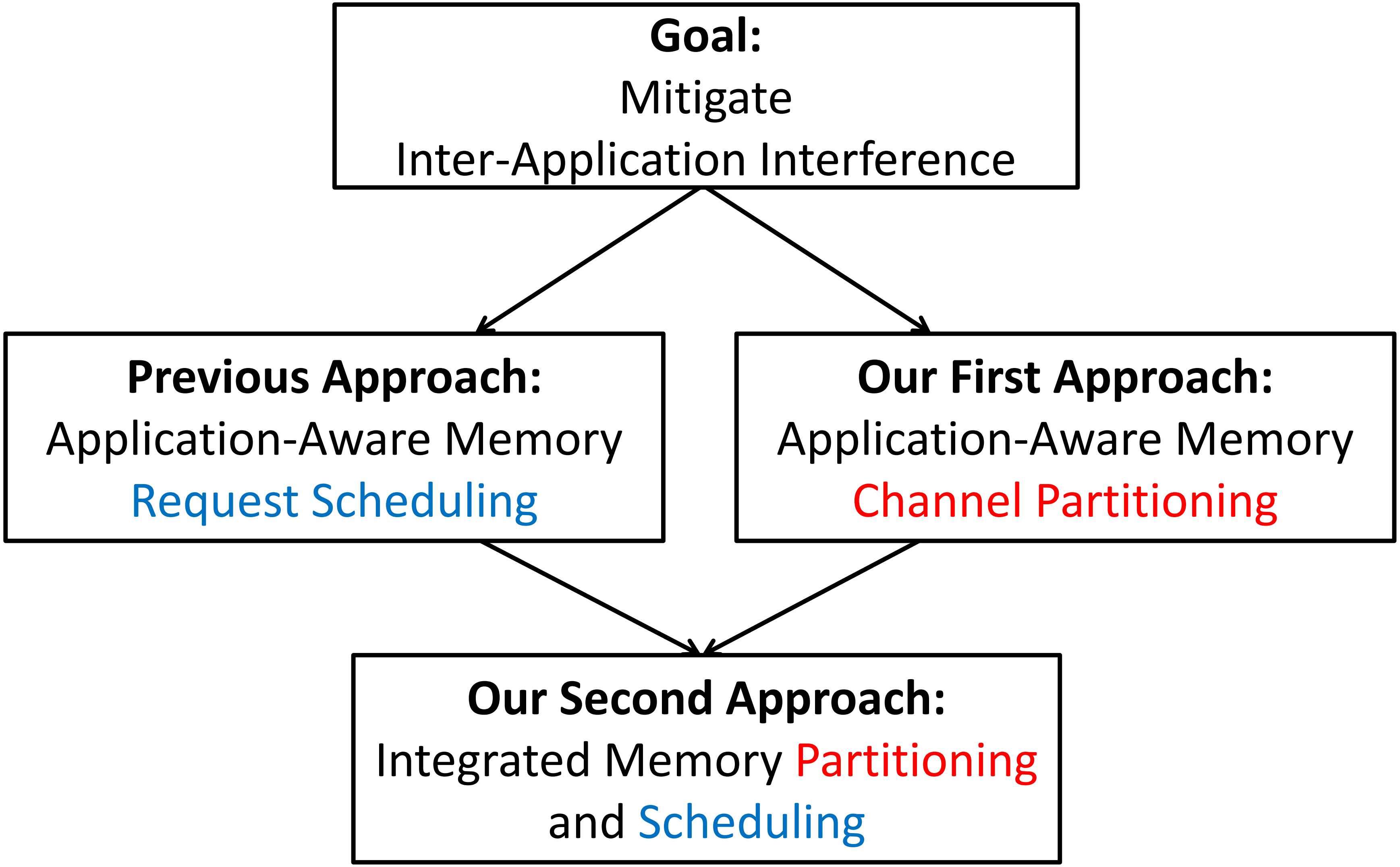
Causes interference between applications' requests

Partitioning Channels Between Applications



Eliminates interference between applications' requests

Integrated Memory Partitioning and Scheduling



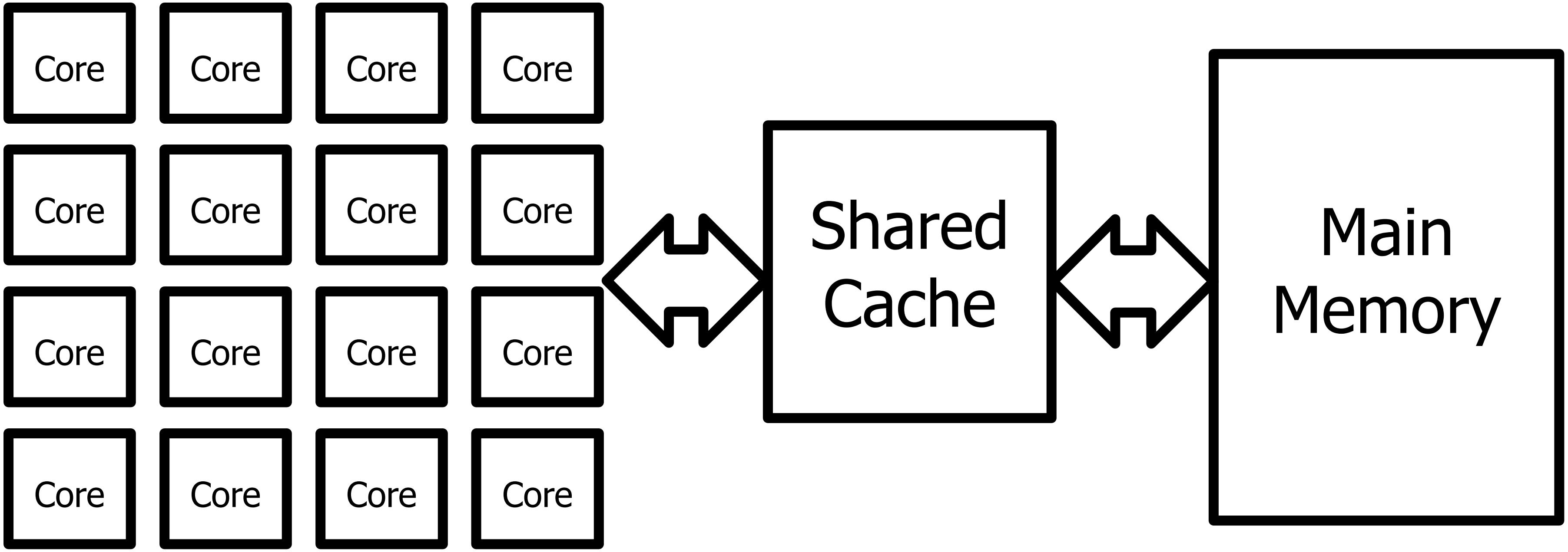
Goal:
Mitigate
Inter-Application Interference

Previous Approach:
Application-Aware Memory
Request Scheduling

Our First Approach:
Application-Aware Memory
Channel Partitioning

Our Second Approach:
Integrated Memory Partitioning
and Scheduling

Slowdown/Interference Estimation in Existing Systems



How do we detect/mitigate the impact of interference on a real system using existing performance counters?

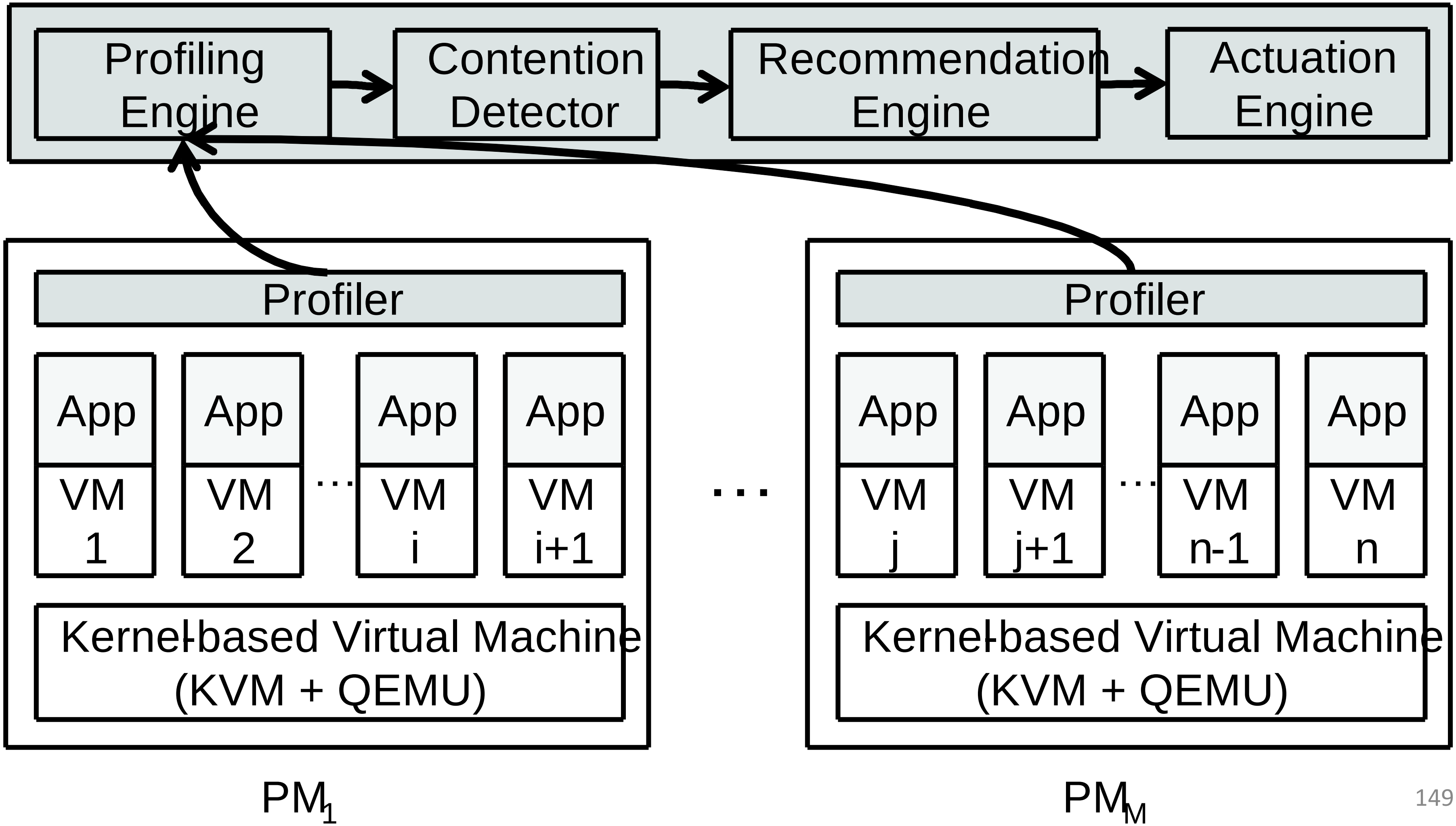
Our Approach: Mitigating Interference in a Cluster

1. *Detect memory bandwidth contention* at each host

2. *Estimate impact* of moving each VM to a non-contended host *(cost-benefit analysis)*

3. *Execute the migrations* that provide the most benefit

Architecture-aware DRM – ADRM (VEE 2015)



ADRM: Key Ideas and Results

- Key Ideas:
  - Memory bandwidth captures impact of shared cache and memory bandwidth interference
  - Model degradation in performance as linearly proportional to bandwidth increase/decrease

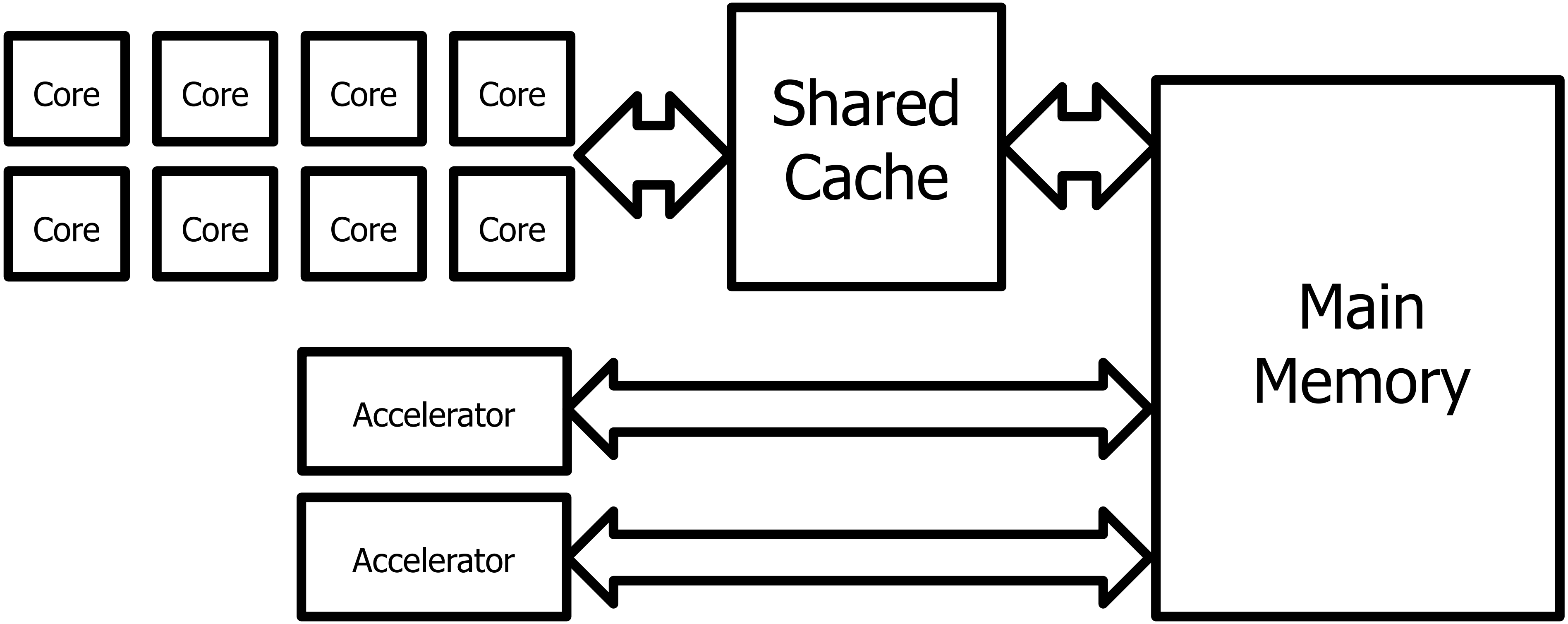
- Key Results:
  - Average performance improvement of 9.67% on a 4-node cluster

QoS in Heterogeneous Systems

- Staged memory scheduling
  - In collaboration with Rachata Ausavarungnirun, Kevin Chang and Gabriel Loh
  - Goal: *High performance in CPU-GPU systems*

- Memory scheduling in heterogeneous systems
  - In collaboration with Hiroukui Usui
  - Goal: *Meet deadlines for accelerators while improving performance*

Performance Predictability in Heterogeneous Systems



Goal of our Scheduler (SQUASH)

- Goal: Design a memory scheduler that
  - Meets accelerators' deadlines *and*
  - Achieves high CPU performance

- Basic Idea:
  - *Different CPU applications and hardware accelerators have different memory requirements*
  - Track progress of different agents and prioritize accordingly

Key Observation: Distribute Priority for Accelerators

- Accelerators need priority to meet deadlines
- Worst case prioritization not always the best
- Prioritize accelerators when they are not on track to meet a deadline

Distributing priority mitigates impact of accelerators on CPU cores' requests

Key Observation: Not All Accelerators are Equal

- Long-deadline accelerators are more likely to meet their deadlines
- Short-deadline accelerators are more likely to miss their deadlines

Schedule short-deadline accelerators based on worst-case memory access time

Key Observation:
Not All CPU cores are Equal

- Memory-intensive cores are much less vulnerable to interference
- Memory non-intensive cores are much more vulnerable to interference

Prioritize accelerators over memory-intensive cores to ensure accelerators do not become urgent

SQUASH: Key Ideas and Results

- *Distribute priority for HWAs*
- *Prioritize HWAs over memory-intensive CPU cores even when not urgent*
- *Prioritize short-deadline-period HWAs based on worst case estimates*

Improves CPU performance by 7-21%
Meets 99.9% of deadlines for HWAs