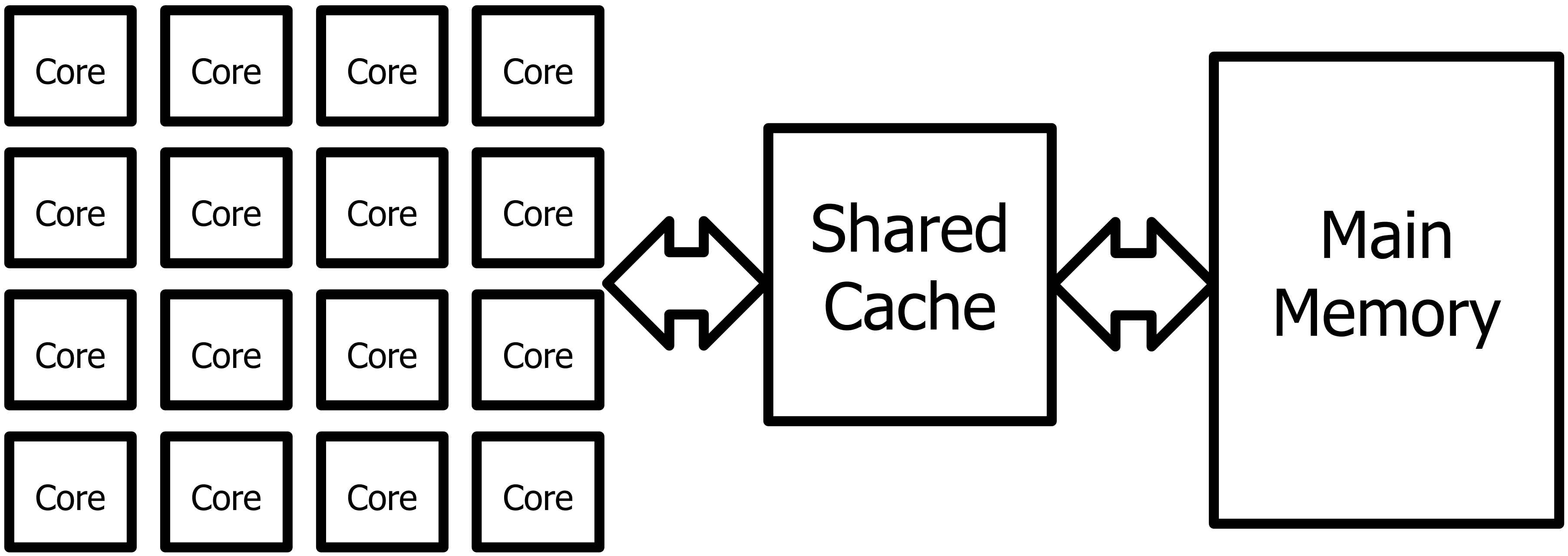
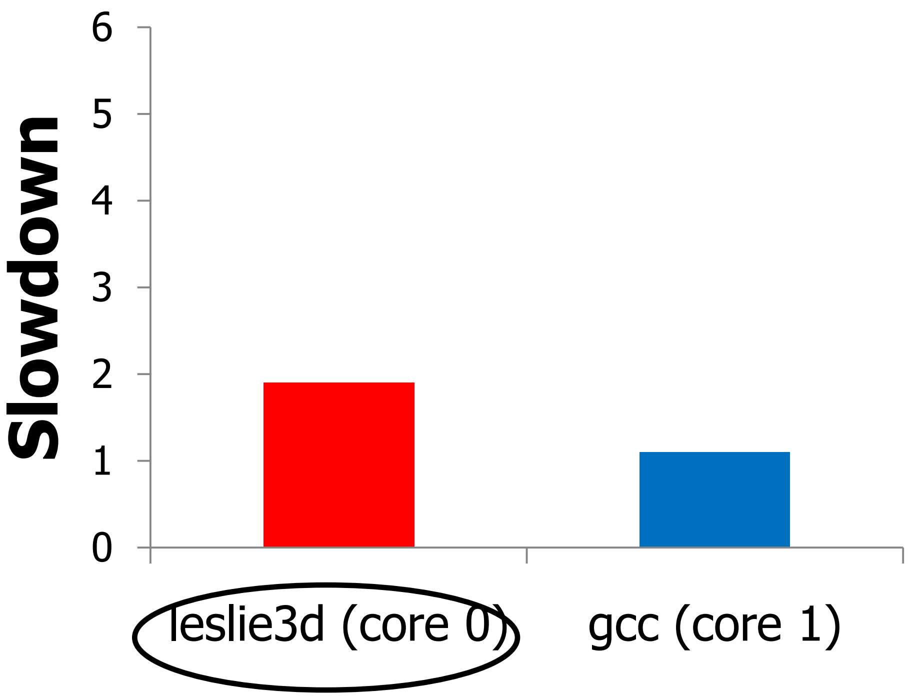
Providing High and Predictable Performance in Multicore Systems Through Shared Resource Management

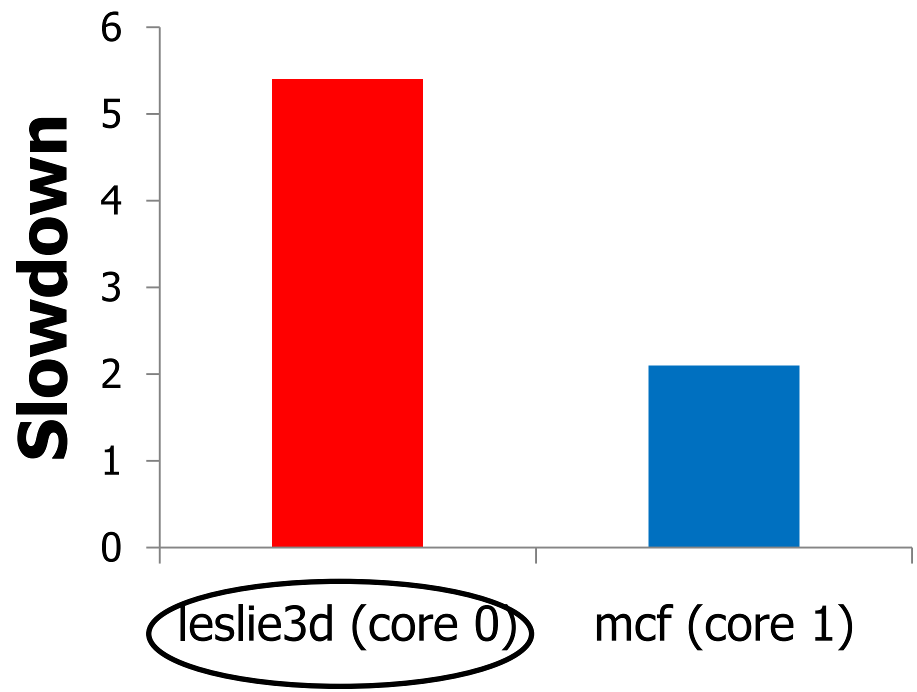
Lavanya Subramanian

Shared Resource Interference



High and Unpredictable Application Slowdowns





1. High application slowdowns due to shared resource interference

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

Outline

Goals:

1. High performance
2. Predictable performance

- Blacklisting memory scheduler

- Predictability with memory interference

Outline

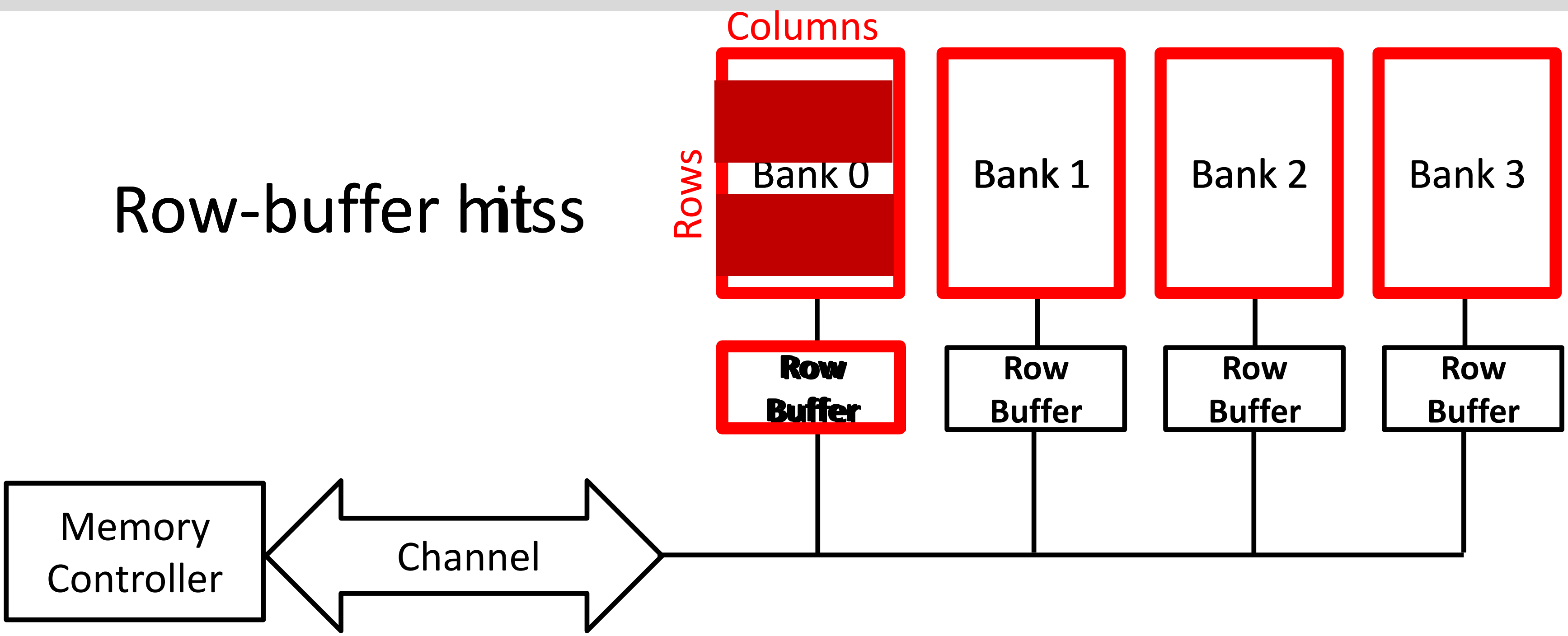
Goals:

1. High performance
2. Predictable performance

- Blacklisting memory scheduler

- Predictability with memory interference

Background: Main Memory



- 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: Memory Request Scheduling

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

An Example: Thread Cluster Memory Scheduling

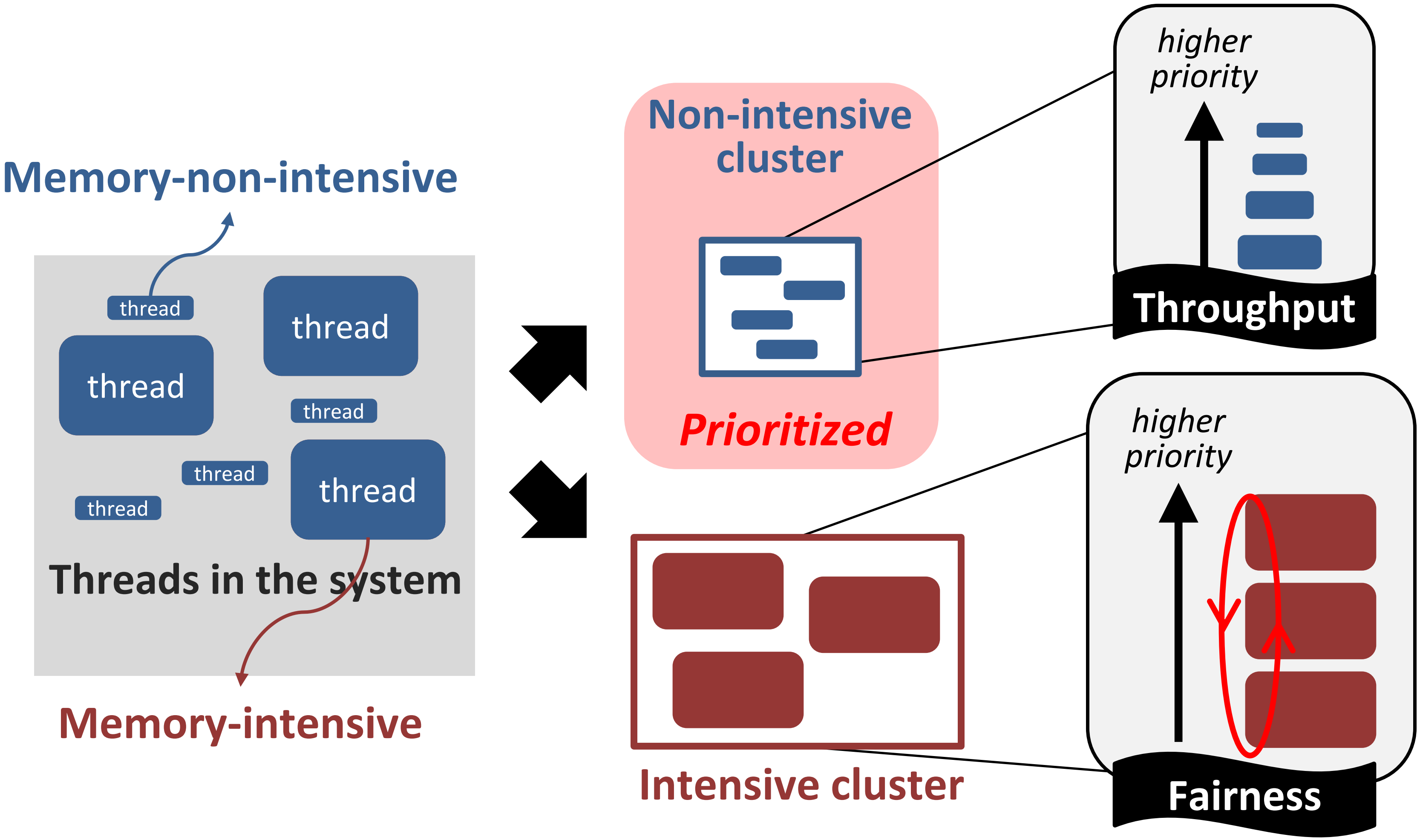


Figure: Kim et al., MICRO 2010

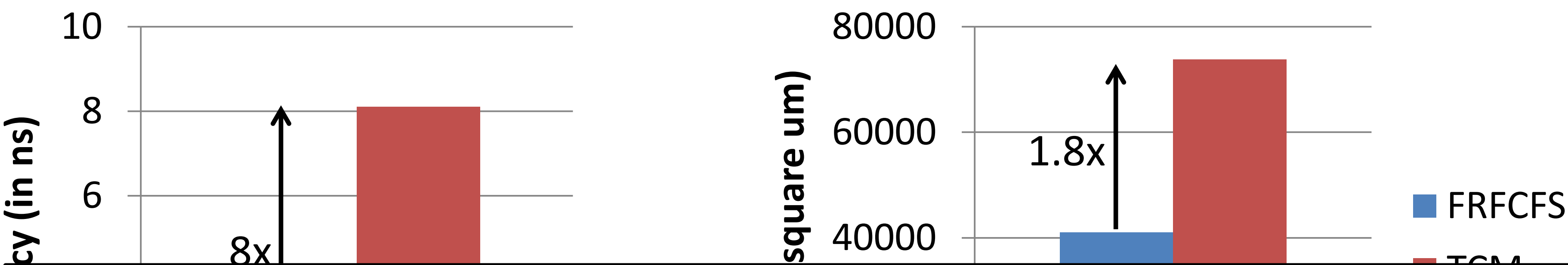
Problems with Previous Application-aware Memory Schedulers

- Hardware Complexity
  – Ranking incurs high hardware cost

- Unfair slowdowns of some applications
  – Ranking causes unfairness

High Hardware Complexity

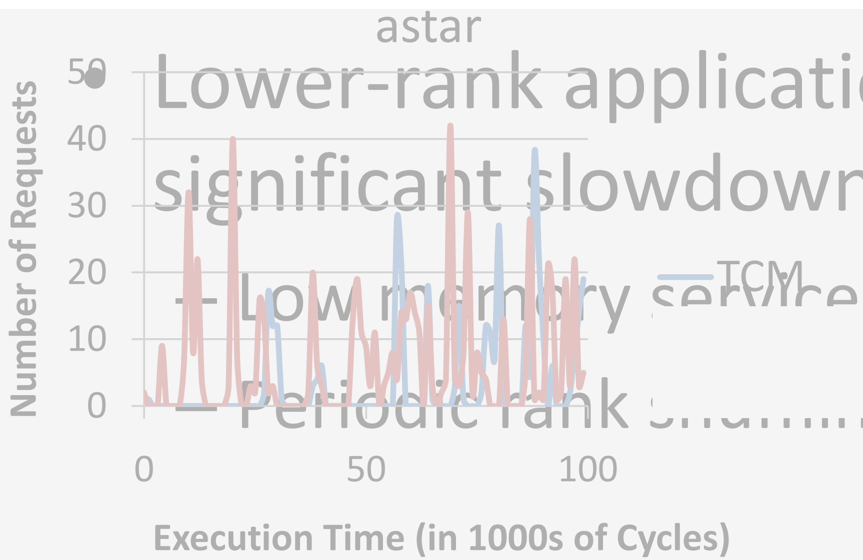
- Ranking incurs high hardware cost
  - Rank computation incurs logic/storage cost
  - Rank enforcement requires comparison logic

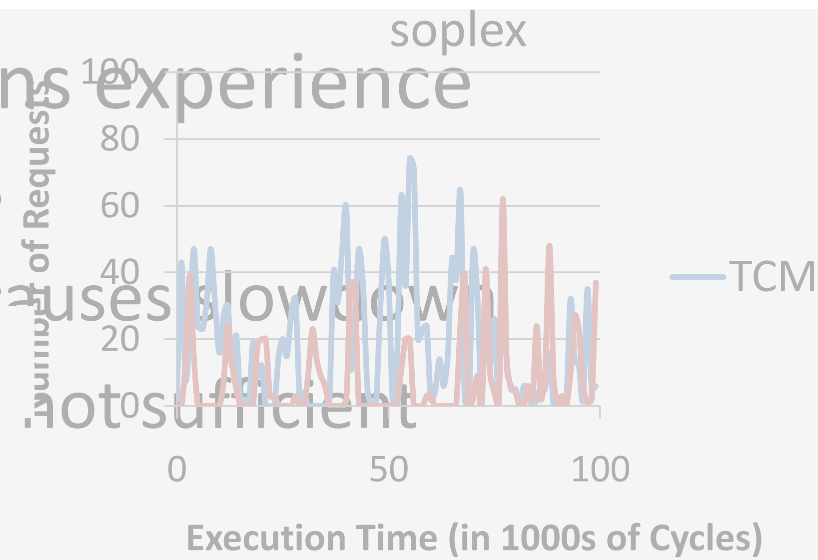


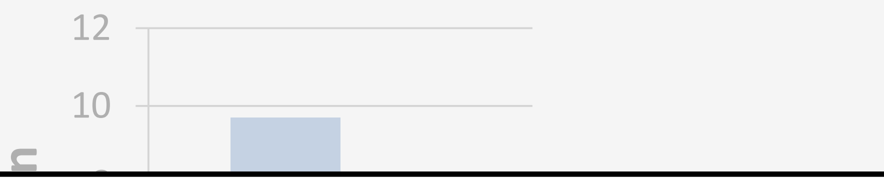
Avoid ranking to achieve low hardware cost

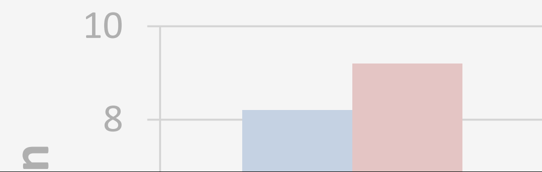
Ranking Causes Unfair Application Slowdowns

- Lower-rank applications experience significant slowdowns
  - Low memory service causes slowdown
  - Periodic rank shuffling not sufficient









Grouping offers lower unfairness than ranking

Problems with Previous Application-Aware Memory Schedulers

- Hardware Complexity
  - Ranking incurs high hardware cost
- Unfair slowdowns of some applications
  - Ranking causes unfairness

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

Towards a New Scheduler Design

- Monitor applications that have a number of consecutive requests served

1. Simple Grouping Mechanism

- Blacklist such applications

- Prioritize requests of non-blacklisted applications

2. Enforcing Priorities Based On Grouping

- Periodically clear blacklists

Methodology

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

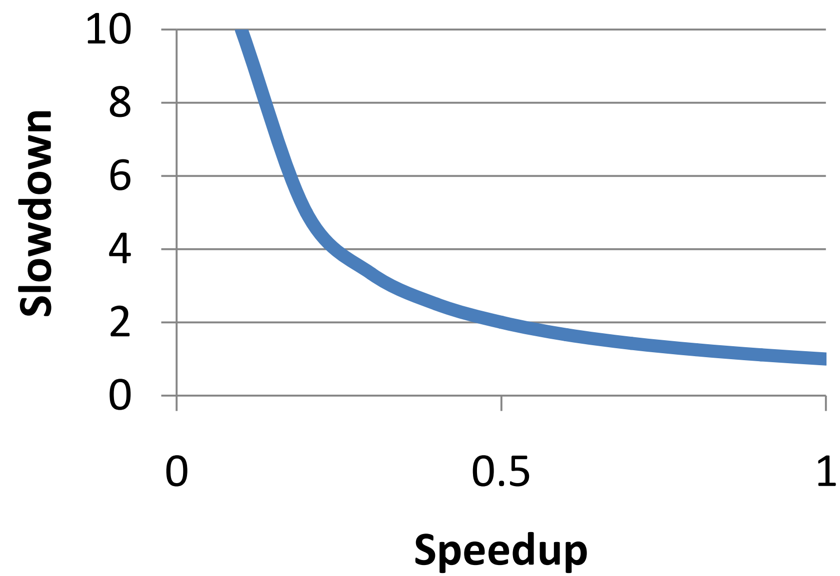
- Workloads
  - SPEC CPU2006, TPCC, Matlab
  - 80 multi programmed workloads

Metrics

- System Performance:

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

$$\text{Harmonic Speedup} = \frac{N}{\sum_i \frac{IPC_i^{alone}}{IPC_i^{shared}}}$$



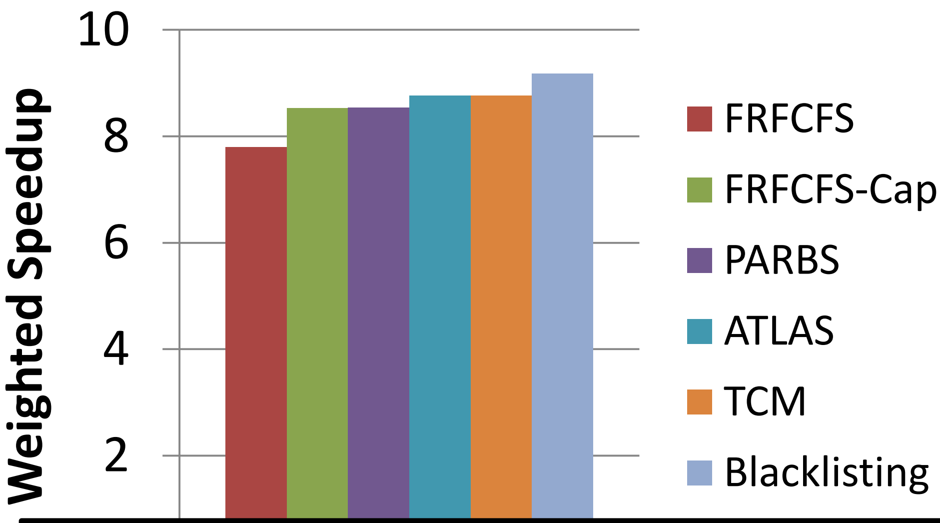
- Fairness:

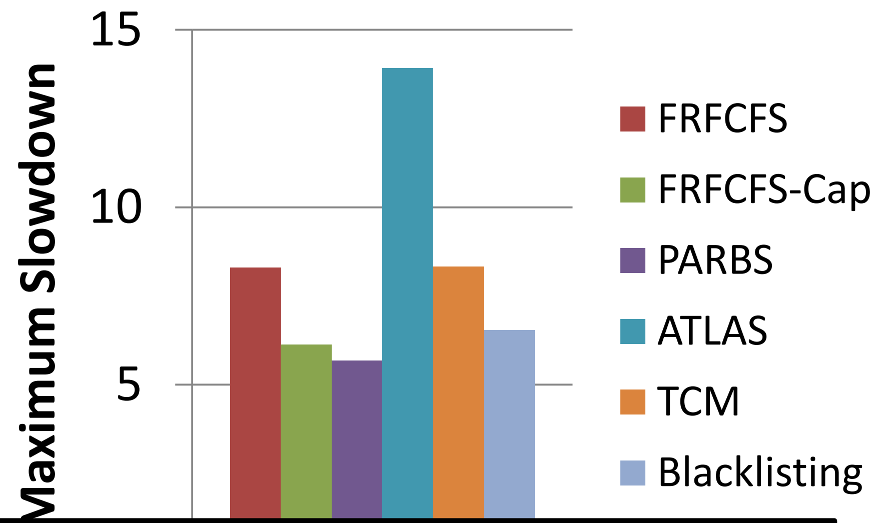
$$\text{Maximum Slowdown} = \max \frac{IPC_i^{alone}}{IPC_i^{shared}}$$

Previous Memory Schedulers

- FR-FCFS [Zuravleff and Robinson, US Patent 1997, Rixner et al., ISCA 2000]
  - Prioritizes row-buffer hits and older requests
  - Application-unaware
- 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 request priorities of high memory-intensity applications

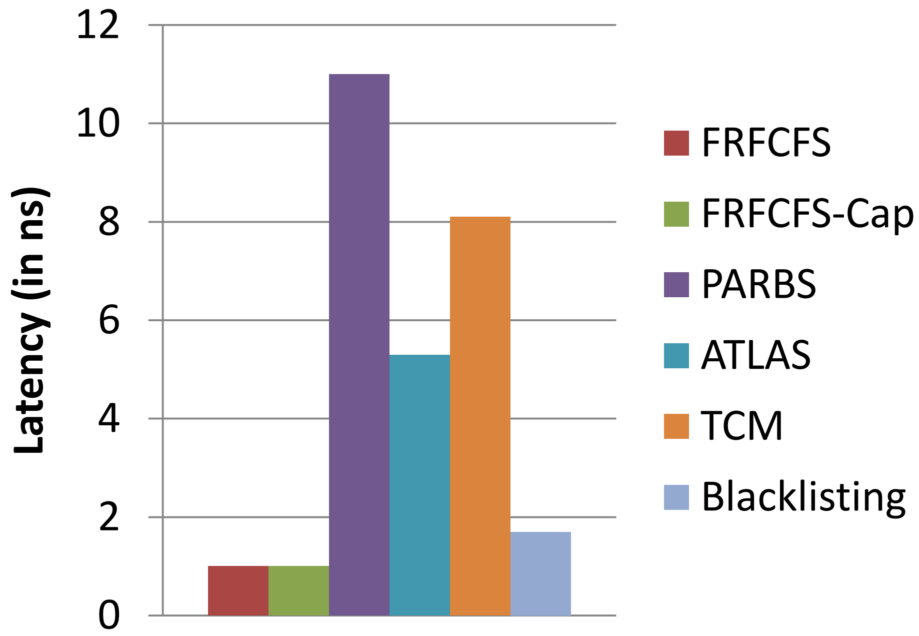
Performance Results

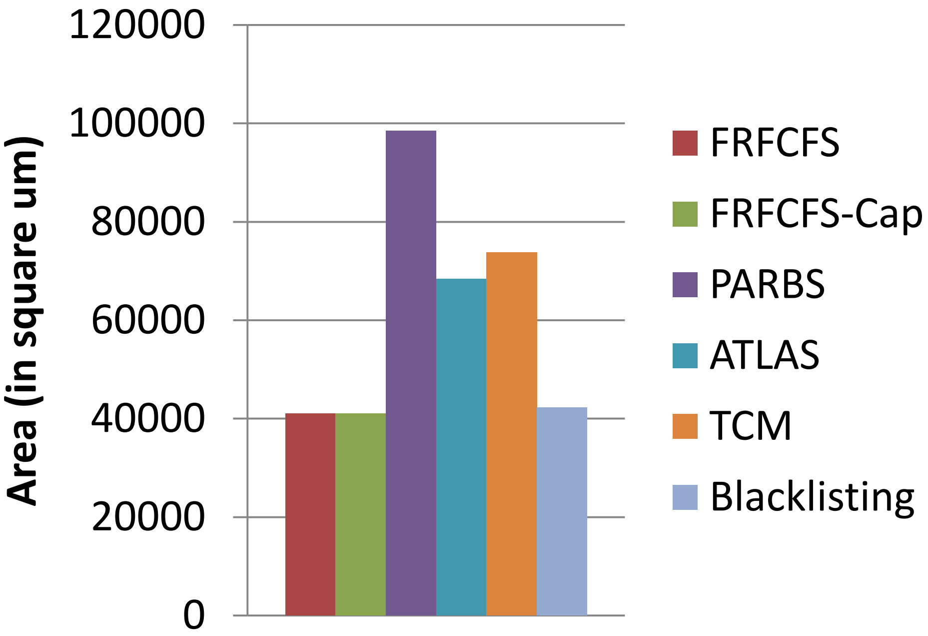




Approaches fairness of PARBS and FRFCFS-Cap achieving better performance than TCM

Complexity Results





Blacklisting achieves 43% lower area than TCM

Outline

Goals:

1. High performance
2. Predictable performance

- Blacklisting memory scheduler

- Predictability with memory interference

Need for Predictable Performance

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

As a first step: Predictable performance in the presence of memory interference

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

Outline

Goals:

1. High performance
2. Predictable performance

- Blacklisting memory scheduler

- Predictability with memory interference

Predictability in the Presence of Memory Interference

1. Estimate Slowdown

2. Control Slowdown

Predictability in the Presence of Memory Interference

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

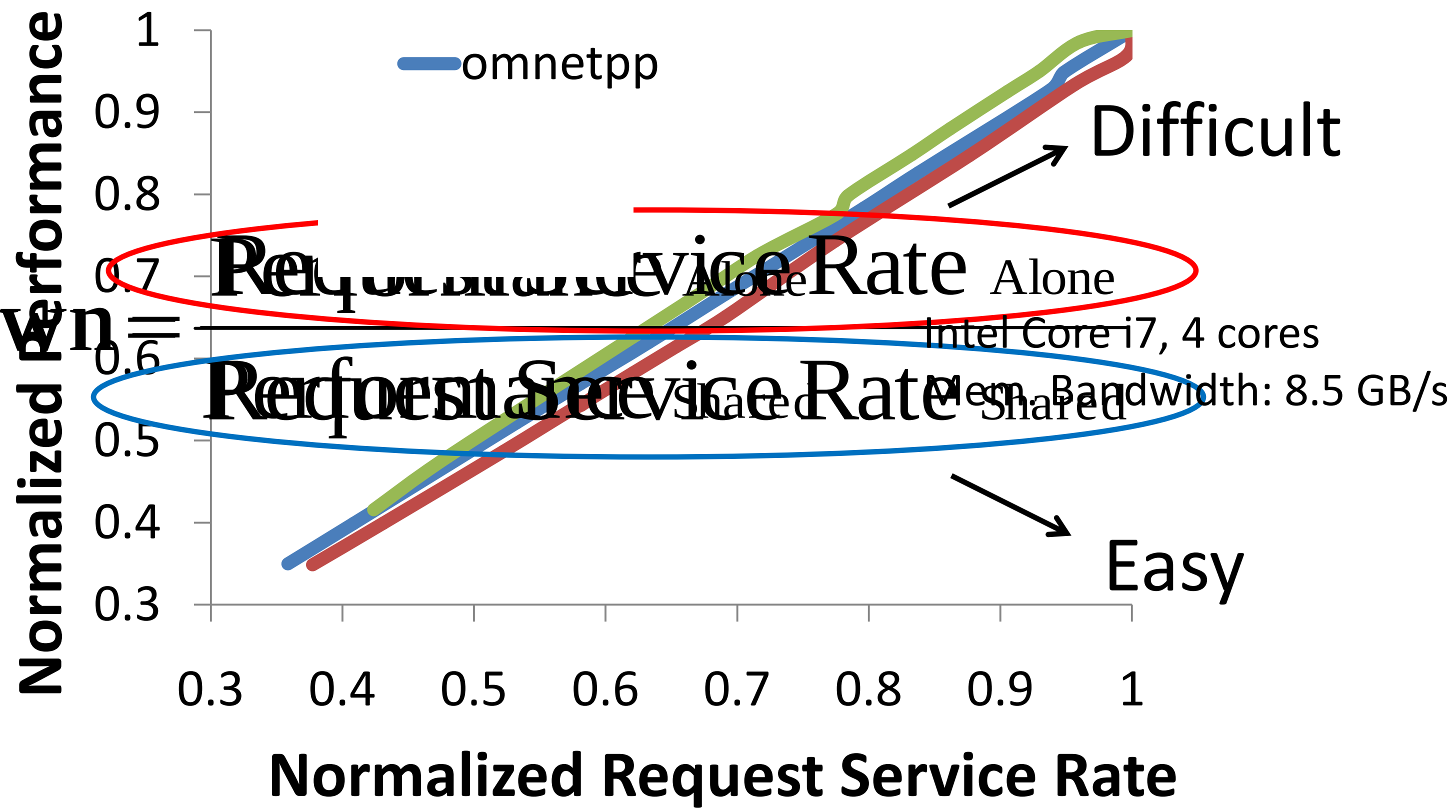
Slowdown: Definition

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

Key Observation 1

For a memory bound application,

Performance ∝ Memory request service rate



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

Key Observation 2

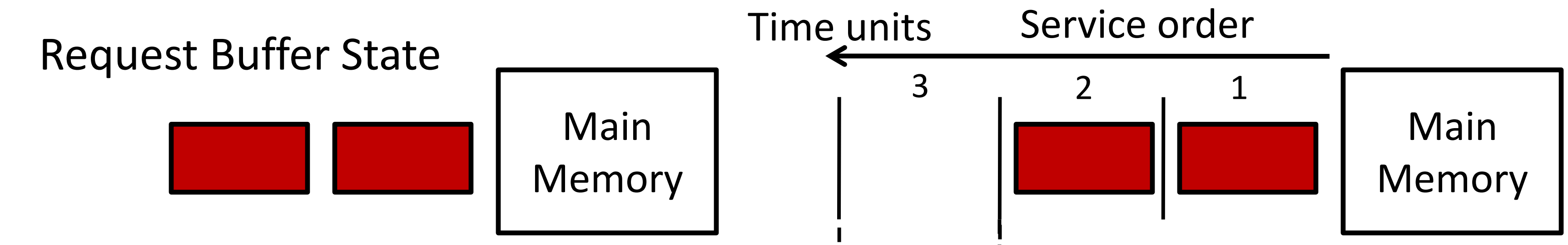
Request Service Rate $_{\text{Alone}}$ (RSR$_{\text{Alone}}$) of an application can be estimated by giving the application highest priority in accessing memory

Highest priority → Little interference

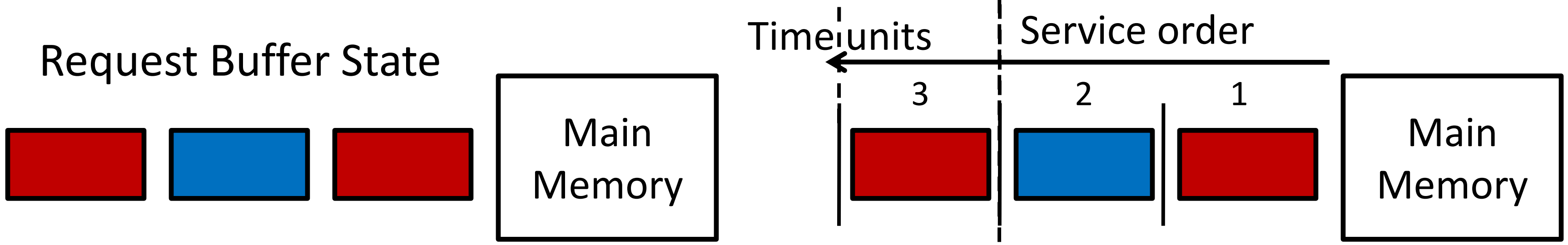
(almost as if the application were run alone)

Key Observation 2

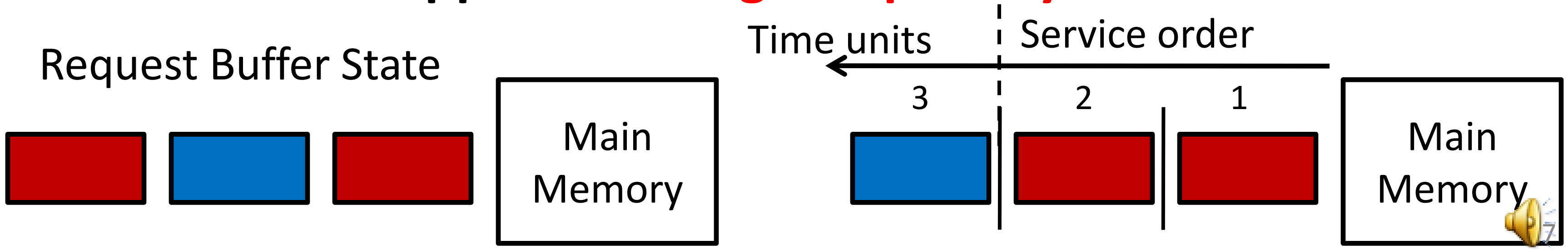
1. Run alone



2. Run with another application



3. Run with another application: highest priority

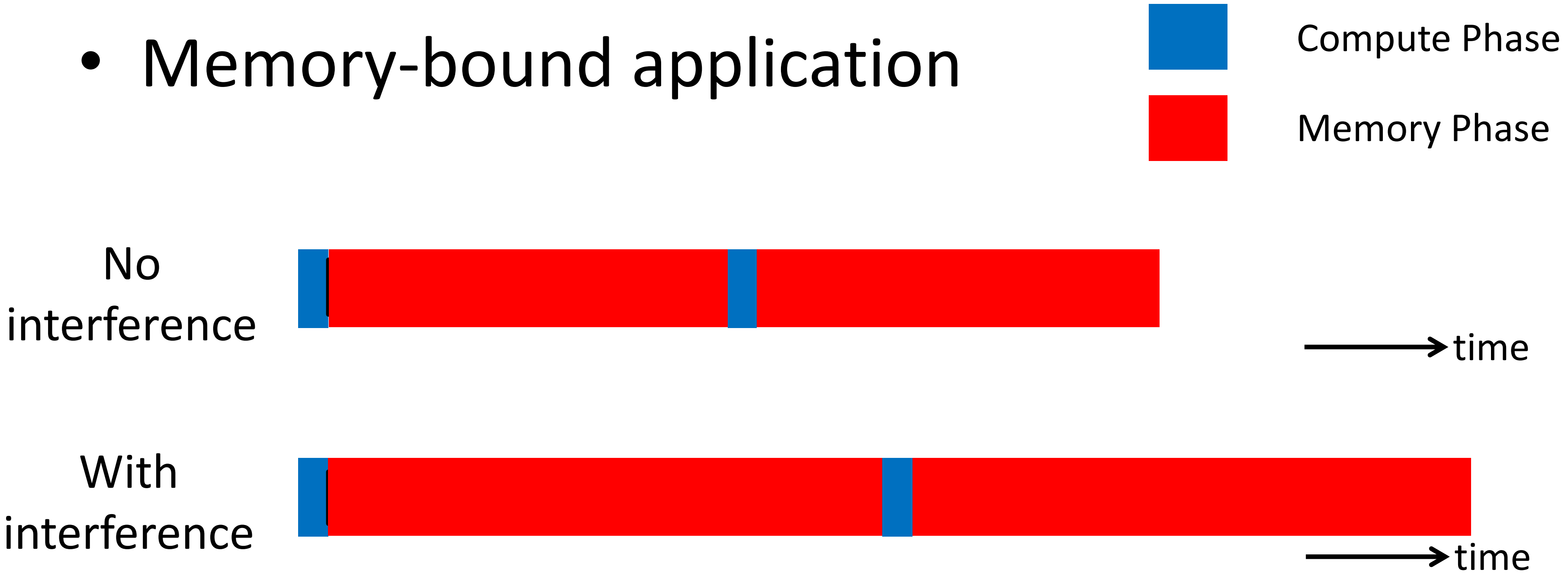


Memory Interference-induced Slowdown Estimation (MISE) model for memory bound applications

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

Key Observation 3

- Memory-bound application



Memory phase slowdown dominates overall slowdown

Key Observation 3

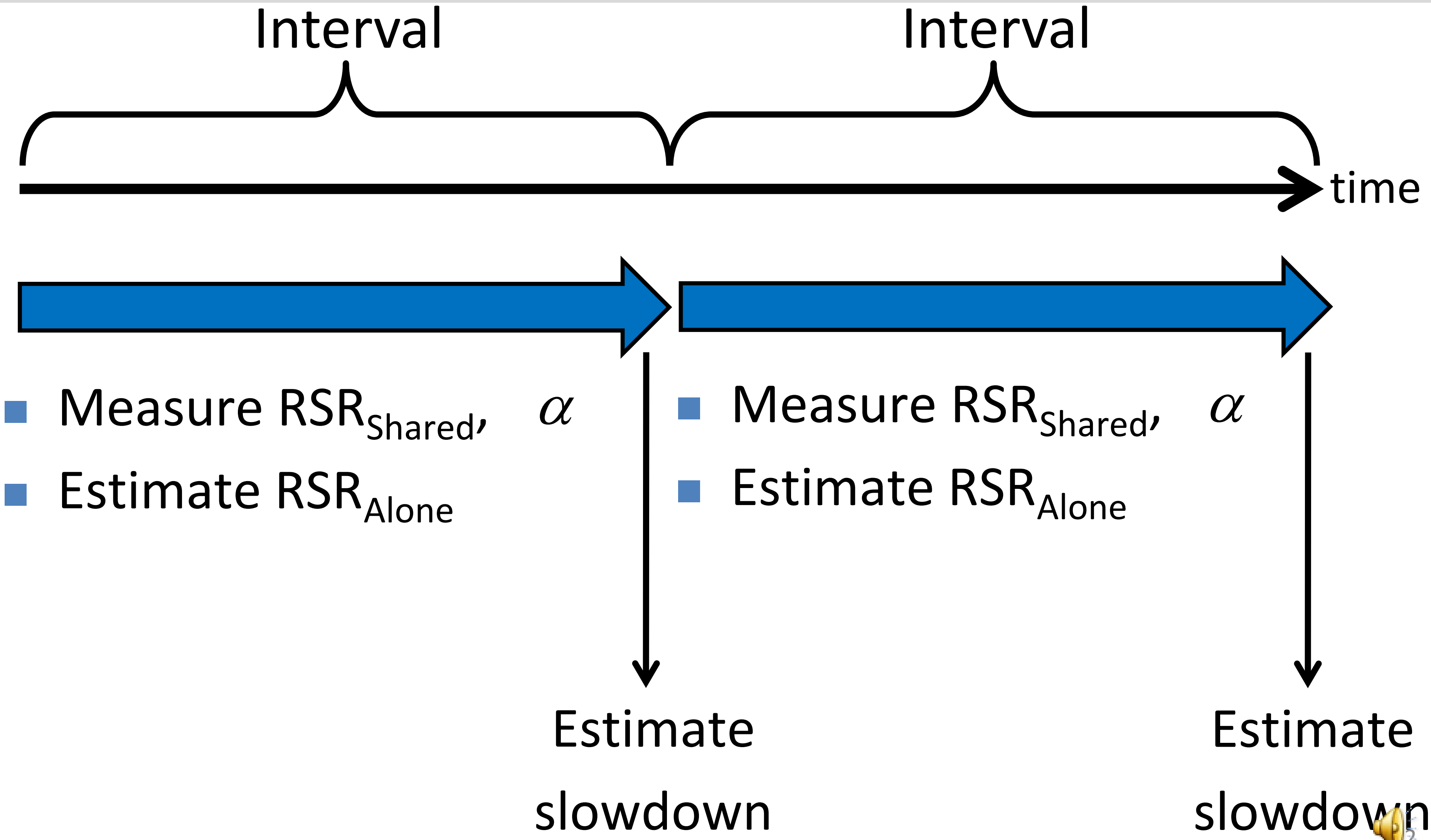
Memory Interference-induced Slowdown Estimation (MISE) model for non-memory bound applications

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

Predictability in the Presence of Memory Interference

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

MISE Operation: Putting it All Together



Predictability in the Presence of Memory Interference

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

- Basic Idea:

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

Difficult

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 → Difficult
  - MISE estimates alone performance while giving an application the highest priority → Easier

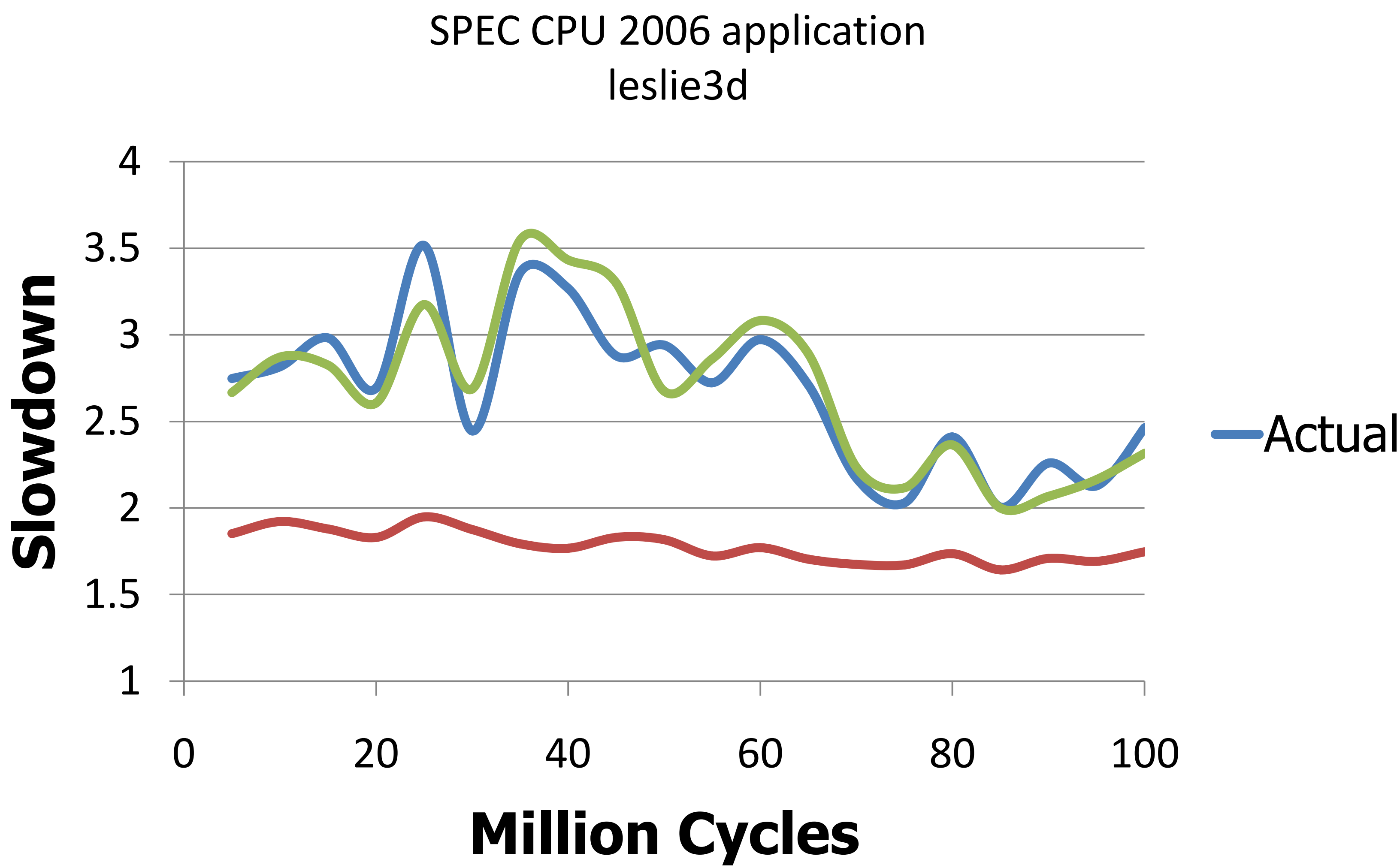
- Advantage 2:
  - STFM does not take into account compute phase for non-memory-bound applications
  - MISE accounts for compute phase → Better accuracy

Methodology

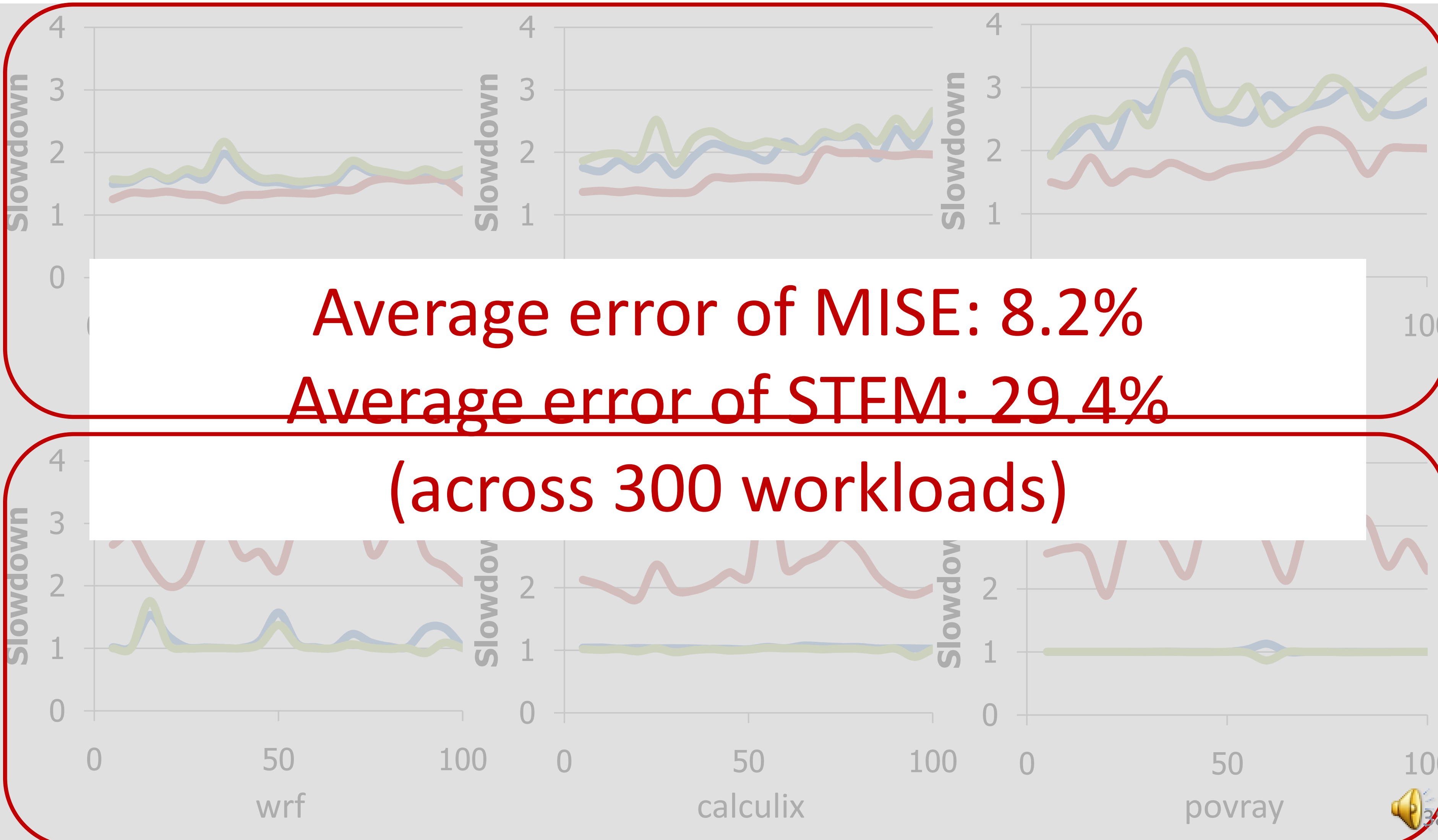
- Configuration of our simulated system
  - 4 cores
  - 1 channel, 8 banks/channel
  - DDR3 1066 DRAM
  - 512 KB private cache/core

- 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)

Predictability in the Presence of Memory Interference

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

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

Outline

Goals:

1. High performance
2. Predictable performance

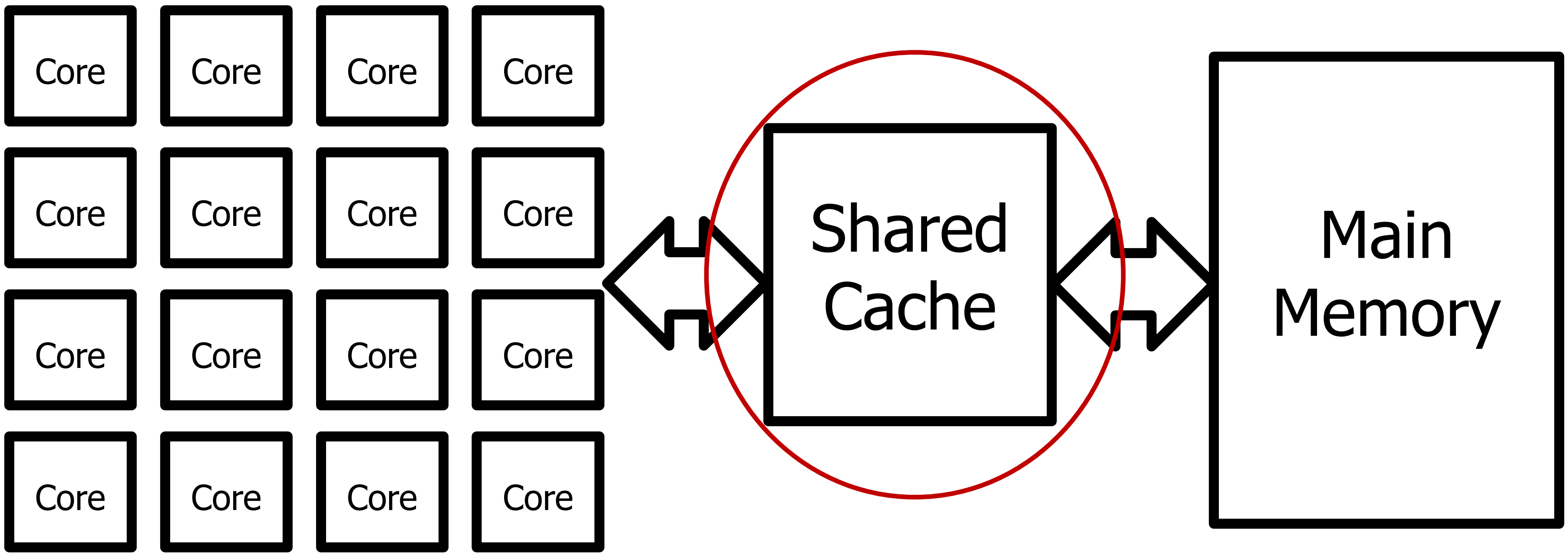
- Blacklisting memory scheduler

- Predictability with memory interference

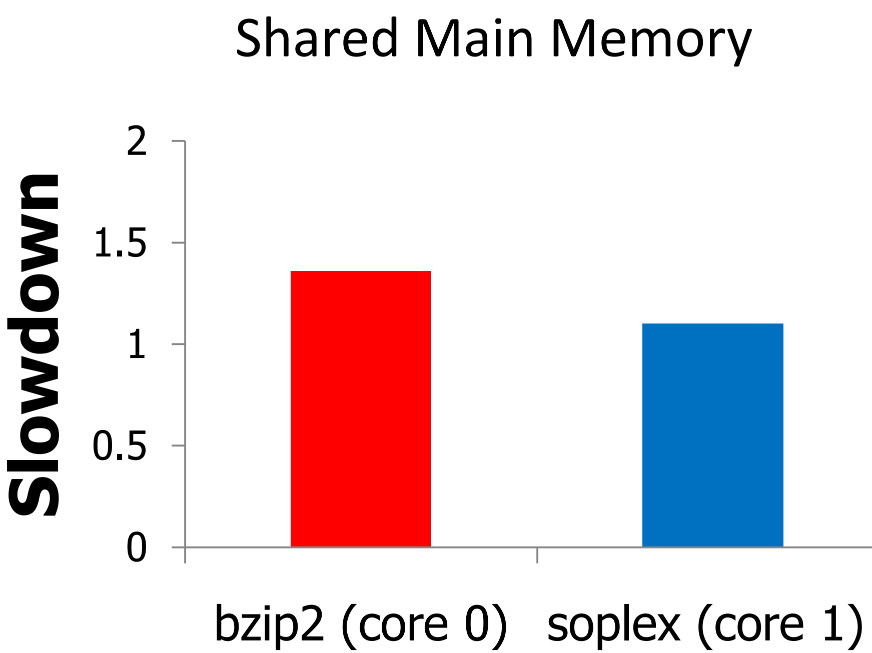
A Recap

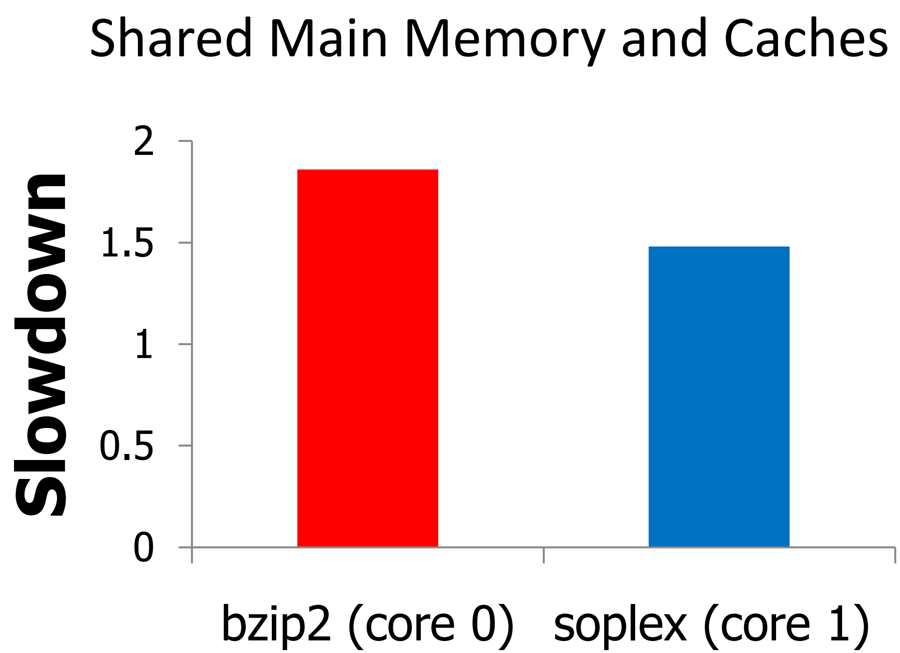
- Problem: Shared resource interference causes high and unpredictable application slowdowns
- Approach:
  - Simple mechanisms to mitigate interference
  - Slowdown estimation models
  - Slowdown control mechanisms
- Future Work:
  - Extending to shared caches

Shared Cache Interference



Impact of Cache Capacity Contention





Cache capacity interference causes high application slowdowns

Backup Slides

Outline

Goals:
1. High performance
2. Predictable performance

- Blacklisting memory scheduler

- Predictability with memory interference

- *Coordinated cache/memory management for performance*

- *Cache slowdown estimation*
- *Coordinated cache/memory management for predictability*

Outline

Goals:

1. High performance
2. Predictable performance

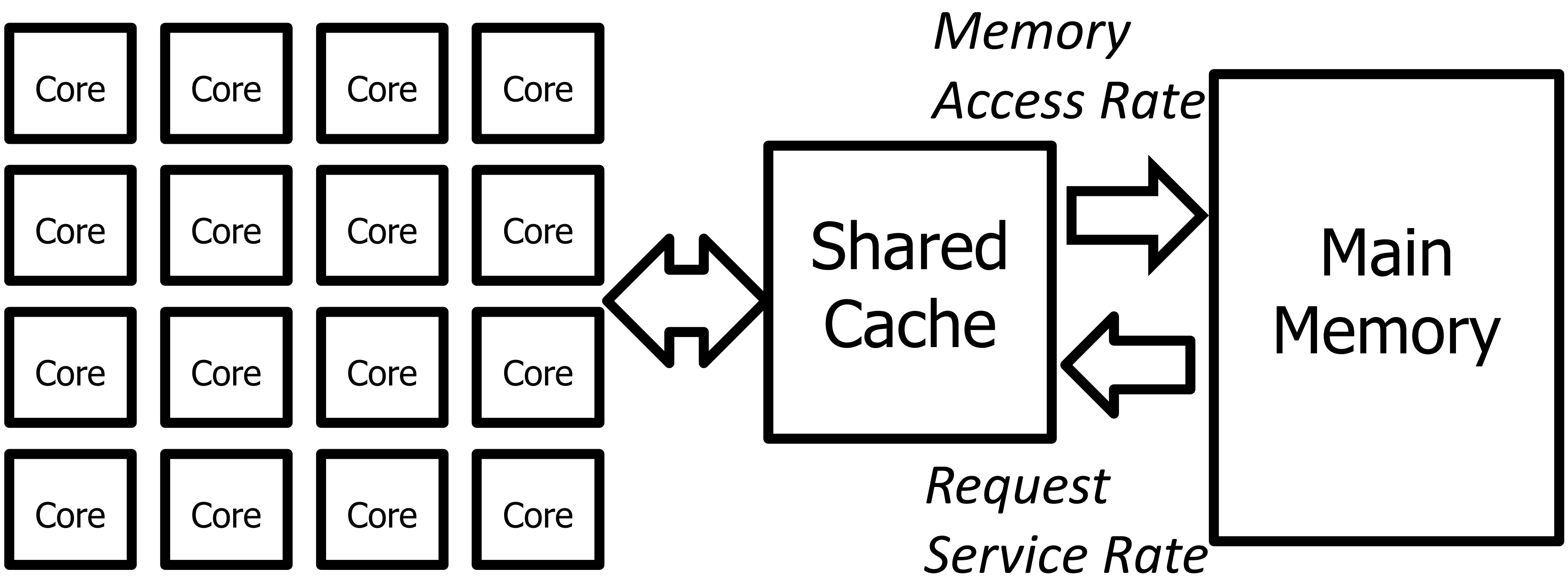
- Blacklisting memory scheduler

- Predictability with memory interference

- *Coordinated cache/memory management for performance*

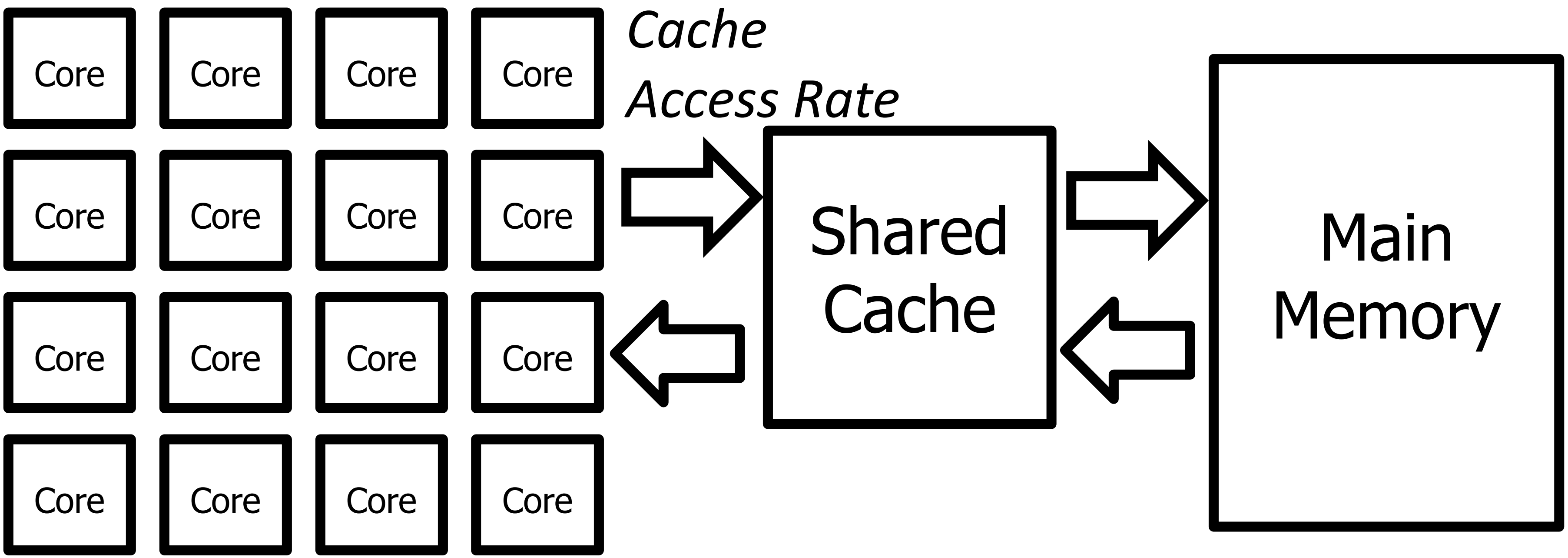
- *Cache slowdown estimation*
- *Coordinated cache/memory management for predictability*

Request Service vs. Memory Access



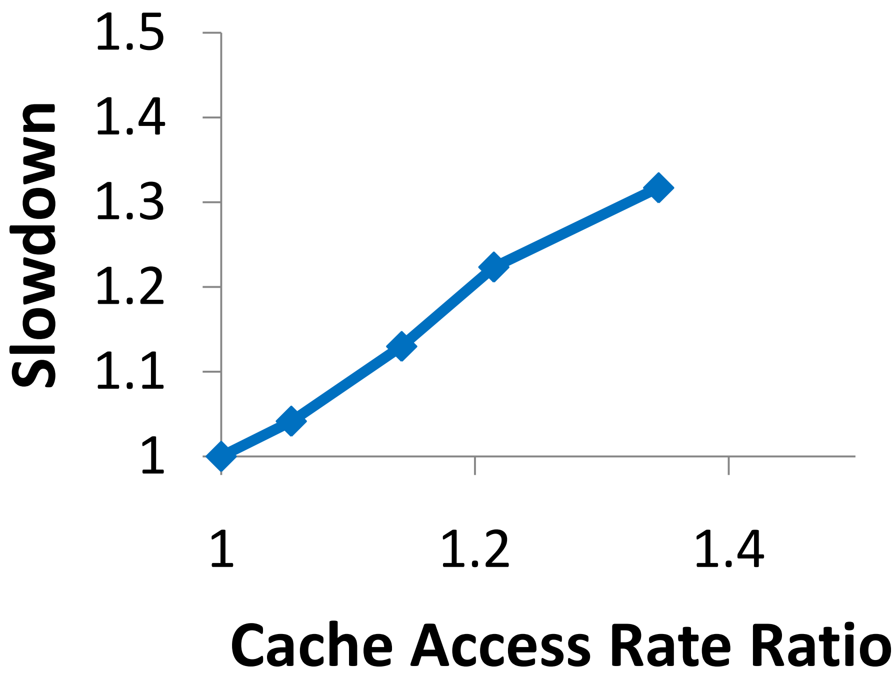
Request service and access rates tightly coupled

Estimating Cache and Memory Slowdowns Through Cache Access Rates

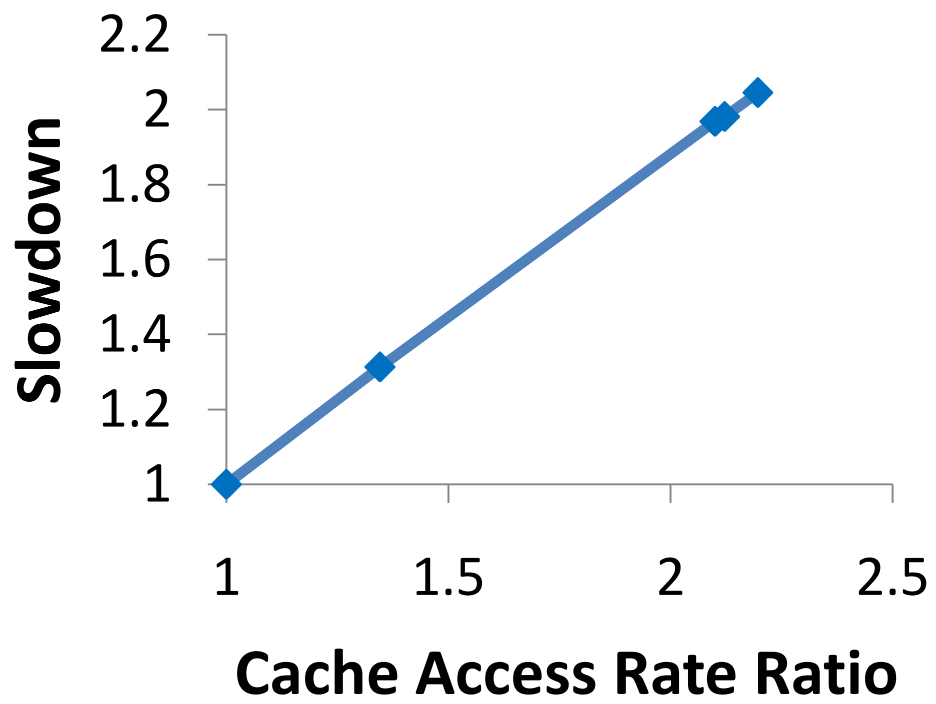


$$\text{Slowdown} = \frac{\text{Cache Access Rate}_{\text{Alone}}}{\text{Cache Access Rate}_{\text{Shared}}}$$

Cache Access Rate vs. Slowdown



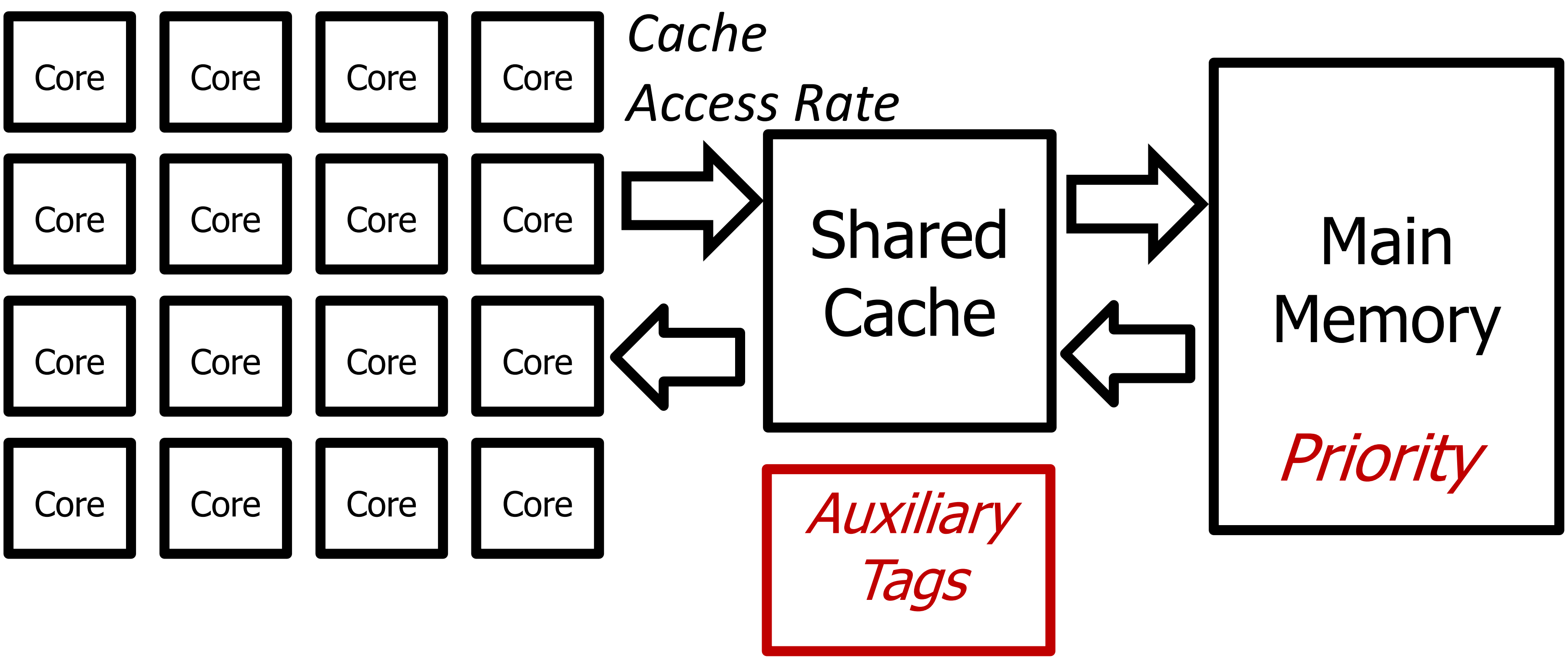
bzip2



xalancbmk

Challenge

How to estimate alone cache access rate?



Outline

Goals:

1. High performance
2. Predictable performance

- Blacklisting memory scheduler

- Predictability with memory interference

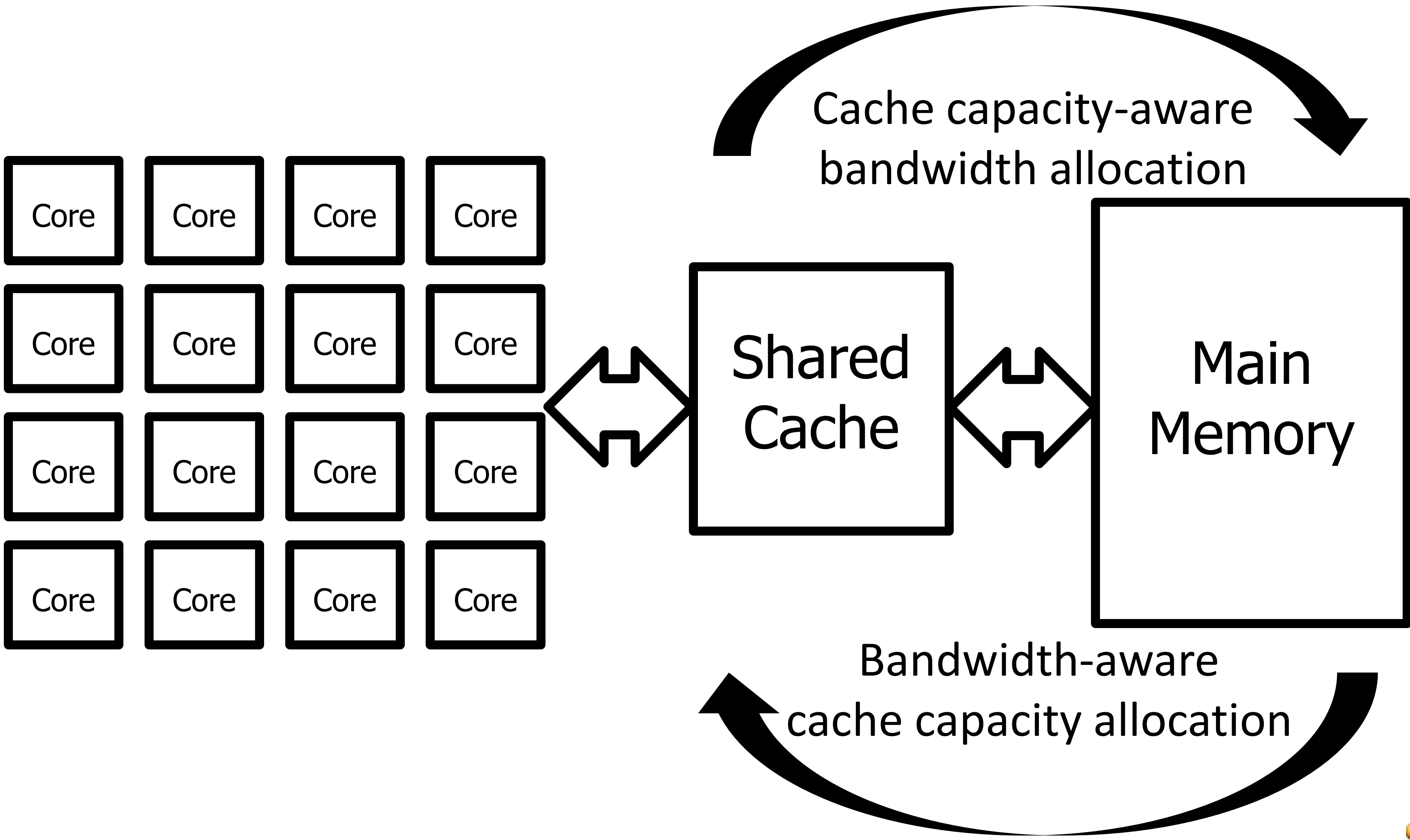
- *Coordinated cache/memory management for performance*

- *Cache slowdown estimation*
- *Coordinated cache/memory management for predictability*

Leveraging Slowdown Estimates for Performance Optimization

- How do we leverage slowdown estimates to achieve high performance by allocating
  - Memory bandwidth?
  - Cache capacity?
- Leverage other metrics along with slowdowns
  - Memory intensity
  - Cache miss rates

Coordinated Resource Allocation Schemes



Outline

Goals:

1. High performance
2. Predictable performance

- Blacklisting memory scheduler

- Predictability with memory interference

- *Coordinated cache/memory management for performance*

- *Cache slowdown estimation*
- *Coordinated cache/memory management for predictability*

Coordinated Resource Management Schemes for Predictable Performance

Goal: Cache capacity and memory bandwidth allocation for an application to meet a bound

Challenges:

- Large search space of potential cache capacity and memory bandwidth allocations
- Multiple possible combinations of cache/memory allocations for each application

Outline

Goals:

1. High performance
2. Predictable performance

- Blacklisting memory scheduler

- Predictability with memory interference

- *Coordinated cache/memory management for performance*

- *Cache slowdown estimation*
- *Coordinated cache/memory management for predictability*

Timeline

	2014									2015				
	Apr.	May	Jun.	Jul.	Aug.	Sep.	Oct.	Nov.	Dec.	Jan.	Feb.	Mar.	Apr.	May
Cache slowdown estimation (75% Goal)														
Coordinated cache/memory management for performance (100% Goal)														
Coordinated cache/memory management for predictability (125% Goal)														
Writeup thesis and defend														

Summary

- Problem: Shared resource interference causes high and unpredictable application slowdowns
- Goals: High and predictable performance
- Our Approach:
  - Simple mechanisms to mitigate interference
  - Slowdown estimation models
  - Coordinated cache/memory management

Thank You!

Backup Slides

Prior Work: Cache and Memory QoS

- Resource QoS (SIGMETRICS 2007, ICS 2009, TACO 2010)
- Bubble up (MICRO 2011)
- Cache capacity QoS (ASPLOS 2014)

- Provide QoS on resource allocations
- Complementary to our slowdown-based mechanisms

Prior Work: Memory Interference Mitigation

- Source throttling (ASPLOS 2010)
  - Throttle interfering applications at the core
- Memory interleaving (MICRO 2011)
  - Map data to banks to exploit parallelism and row locality

Complementary to our proposals

Prior Work: Shared Cache Management

- Cache replacement policies (PACT 2008, ISCA 2010)
- Insertion policies (ACSAC 2007, MICRO 2011)
- Partitioning (ICS 2004, MICRO 2006, ASPLOS 2014)

- Focused on a single resource: shared cache
- Goal of these policies: Improve performance

Prior Work: DRAM Optimizations

Related Work

- Main memory interference mitigation
- Slowdown estimation
- Shared cache capacity management
- Cache and memory QoS

Main Memory Interference Mitigation

- Source throttling (Ebrahimi et al., ASPLOS 2010)
  – Throttle interfering applications at the core

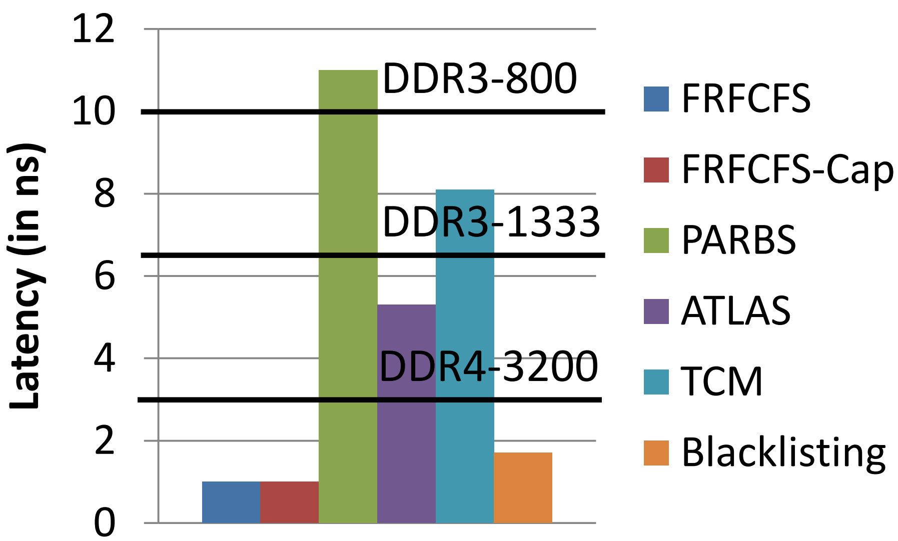
- Memory interleaving (Kaseridis et al., MICRO 2011)
  – Map data to banks to exploit parallelism and row locality

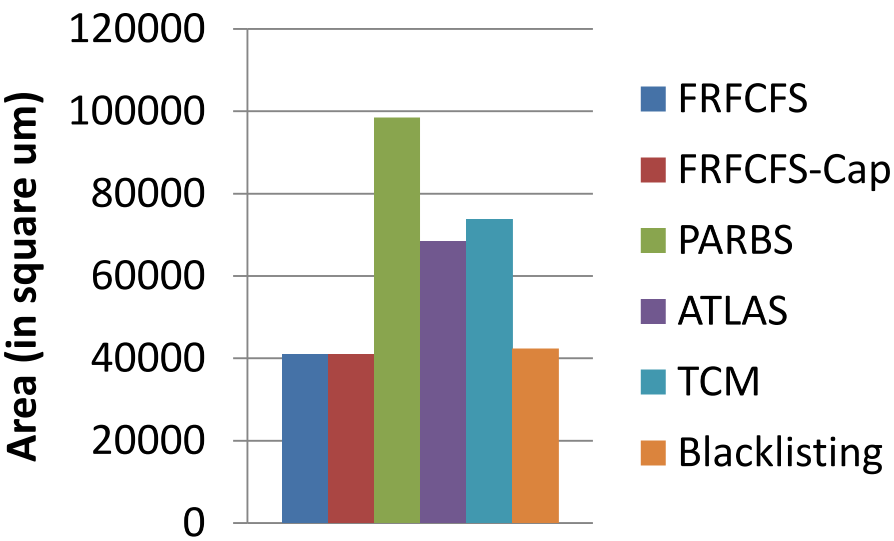
Blacklisting

Blacklisting Scheduler Mechanism

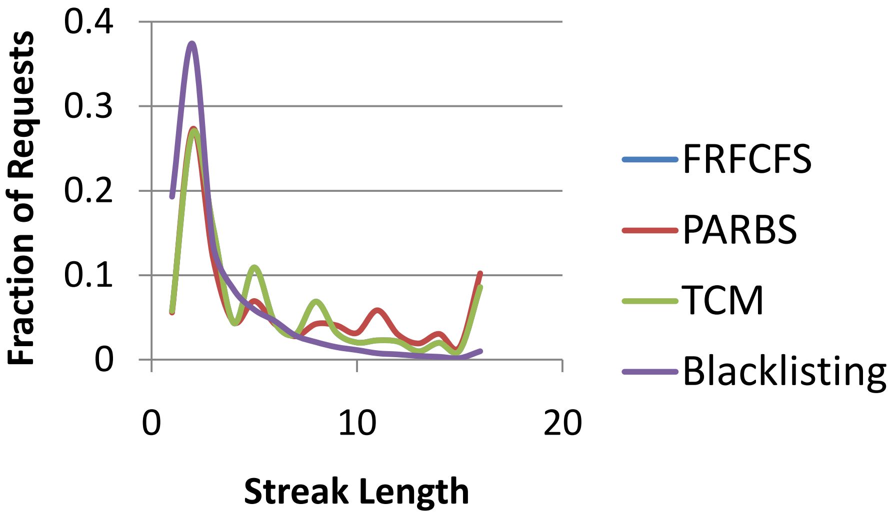
- *At each channel,*
  - Count the number of consecutive requests served from an application
- *The counter is cleared*
  - when a request belongs to a different application than the previous one is served
- *When count equals N*
  - clear the counter
  - blacklist the application
- *Periodically clear blacklists*

Complexity Results

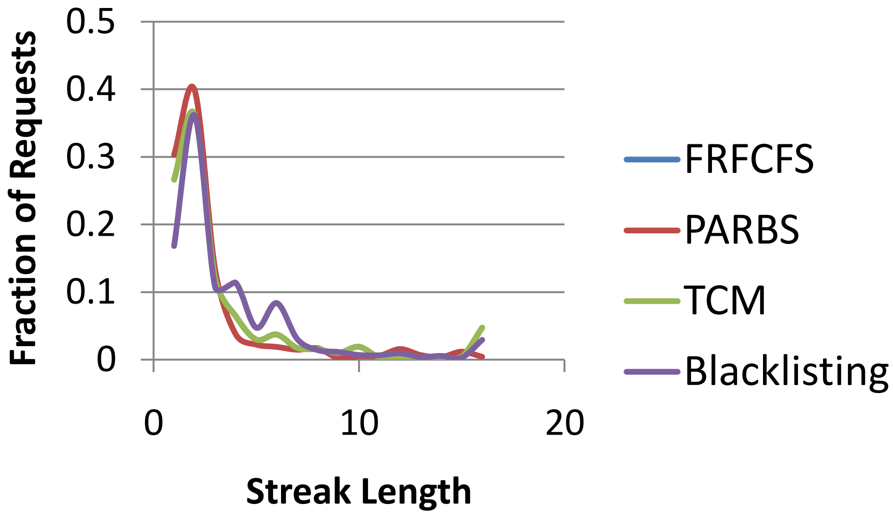




Understanding Why Blacklisting Works



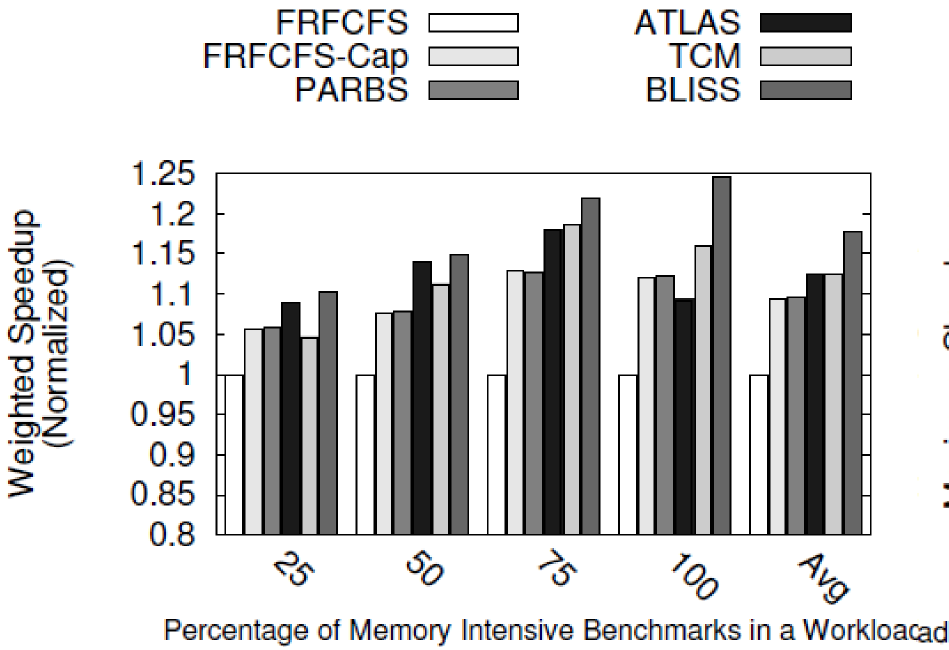
libquantum
(High memory-intensity application)

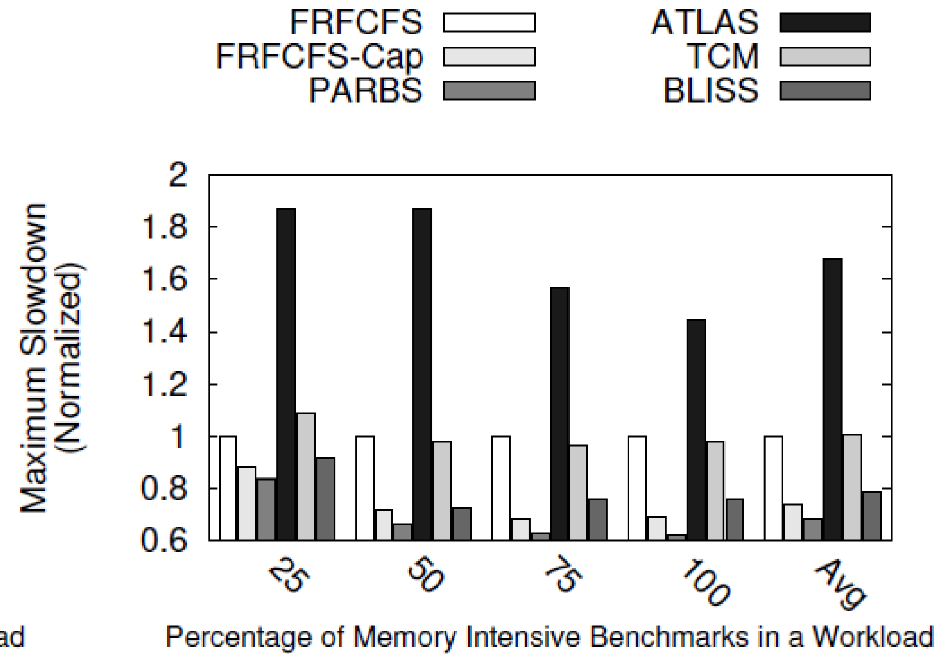


calculix
(Low memory-intensity application)

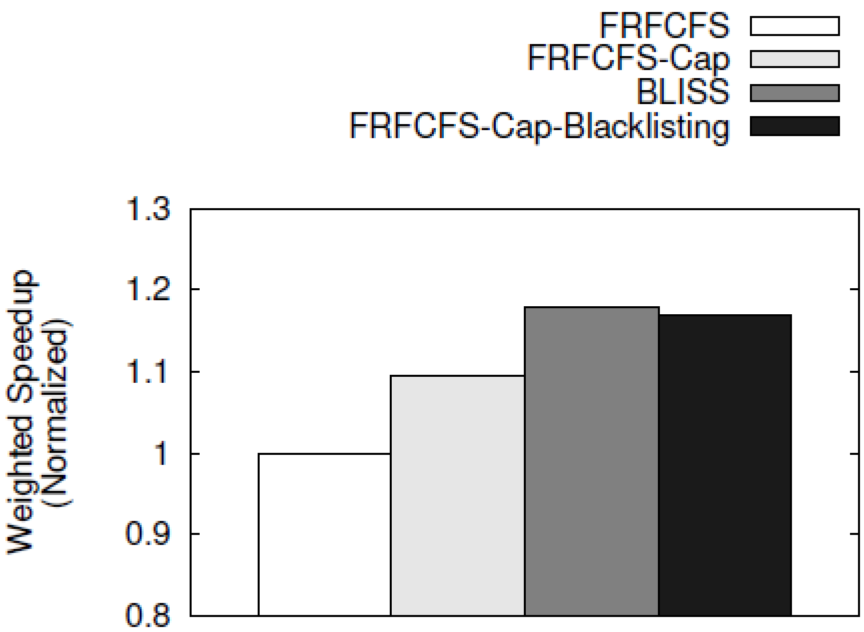
Blacklisting shifts the request distribution towards the right

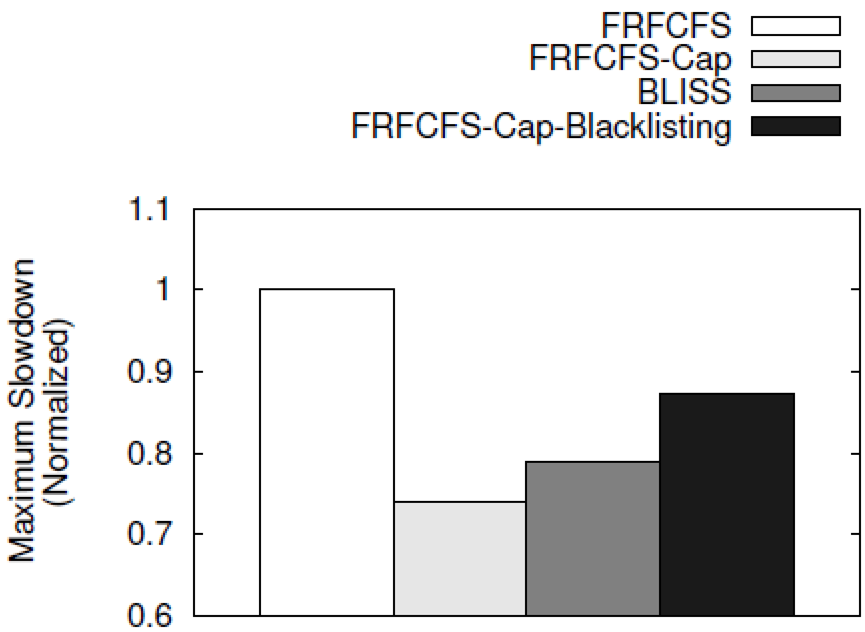
Effect of Workload Memory Intensity



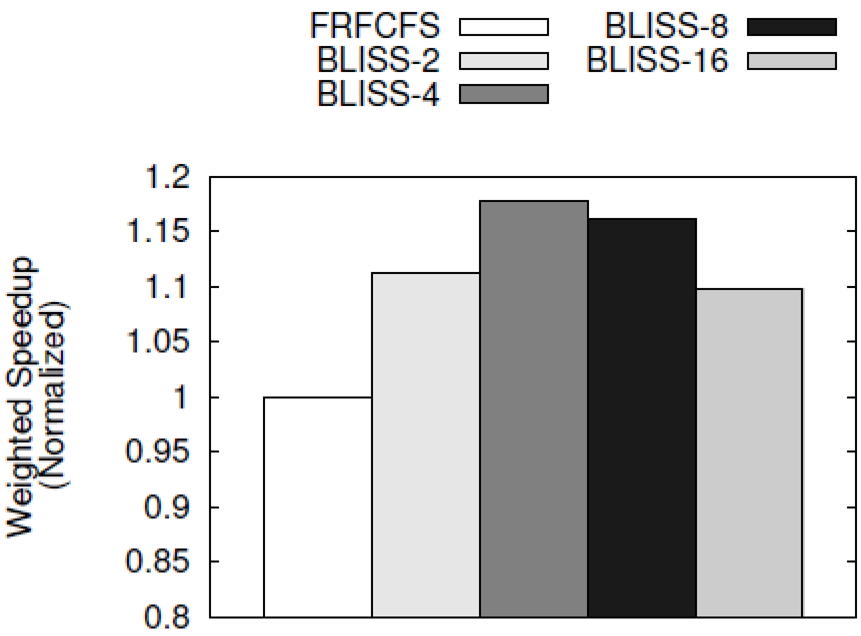


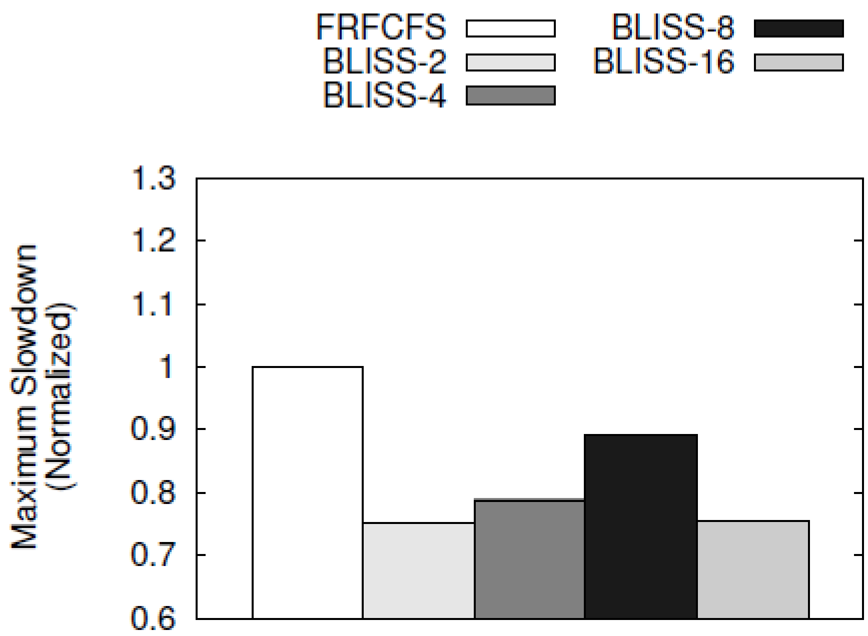
Combining FRFCFS-Cap and Blacklisting



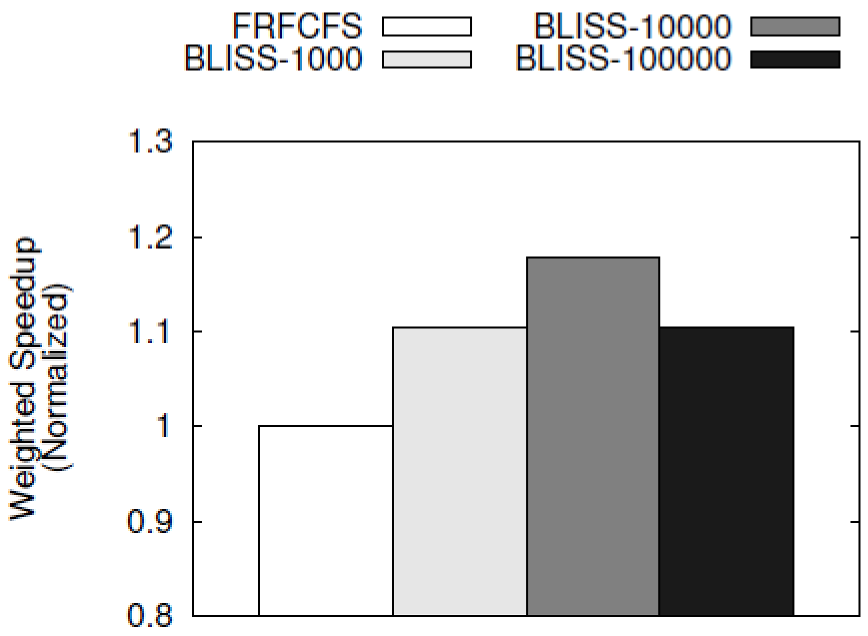


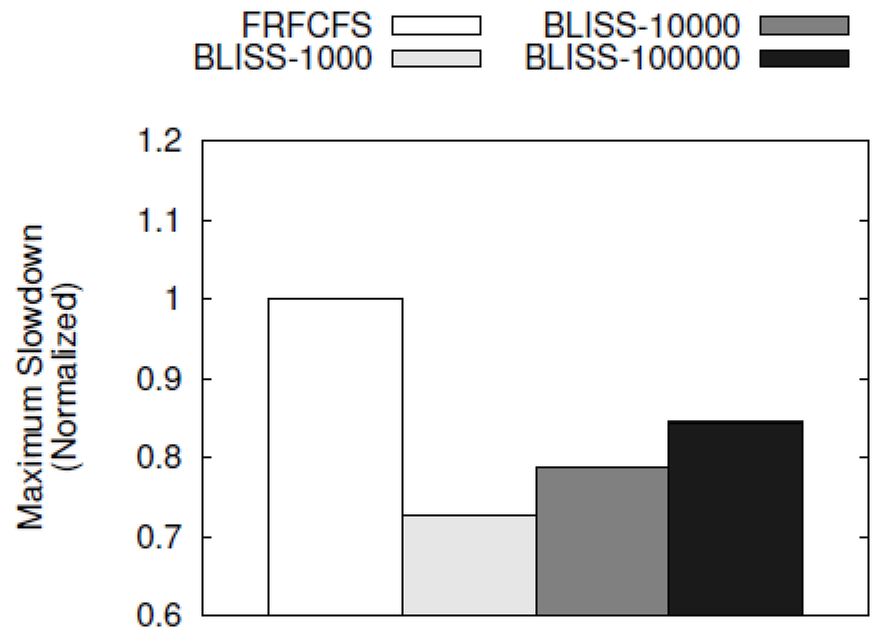
Sensitivity to Blacklisting Threshold



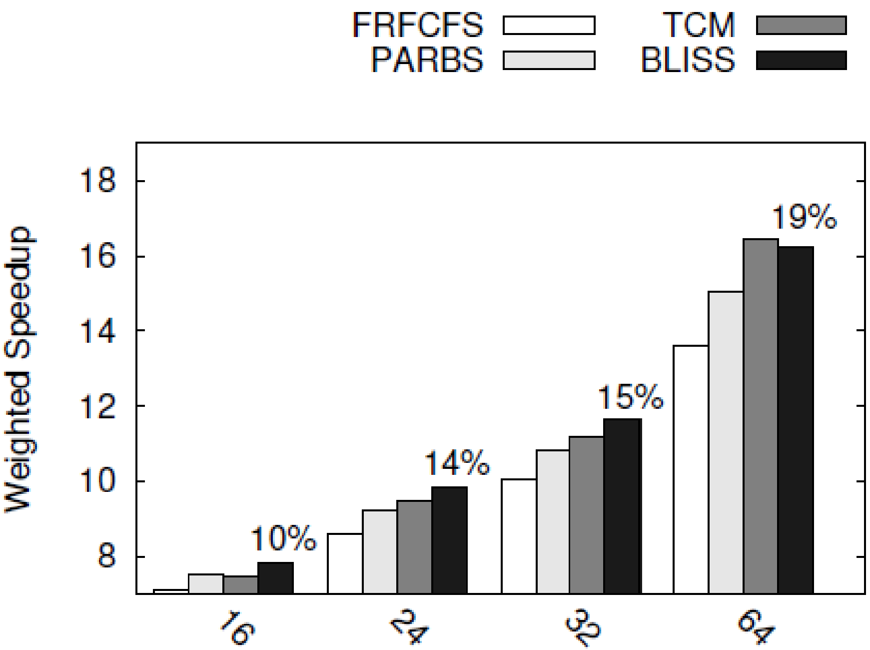


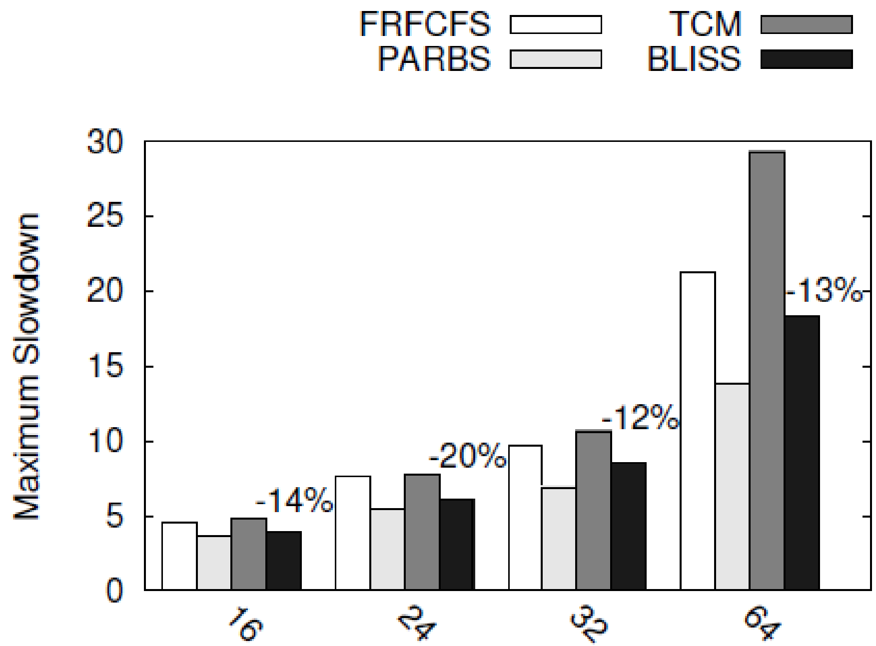
Sensitivity to Clearing Interval



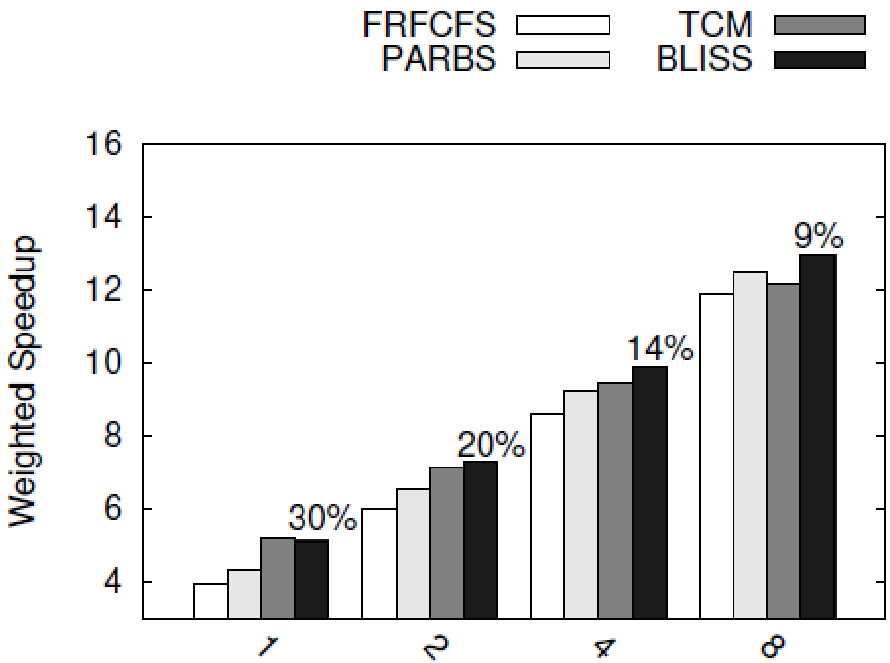


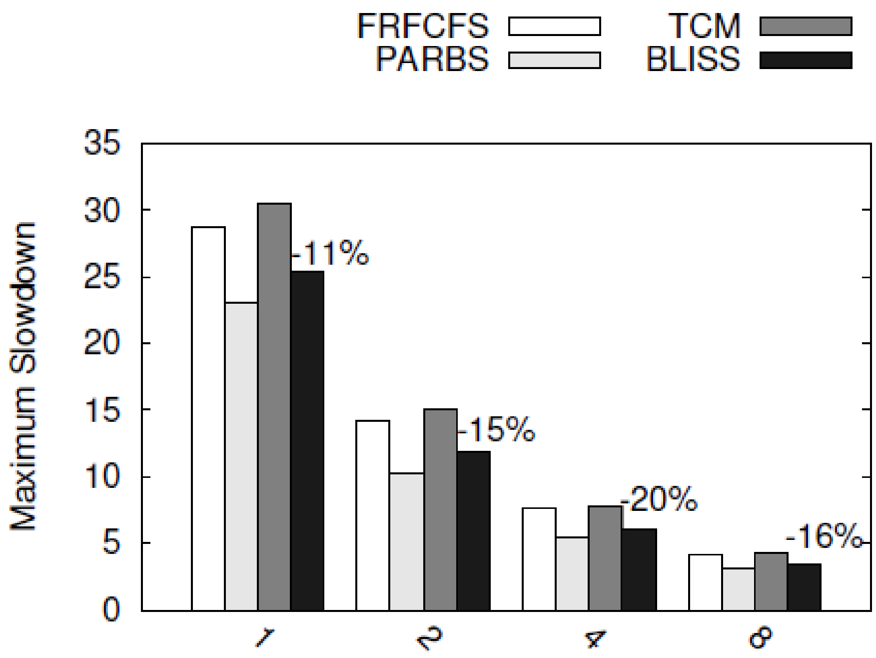
Sensitivity to Core Count



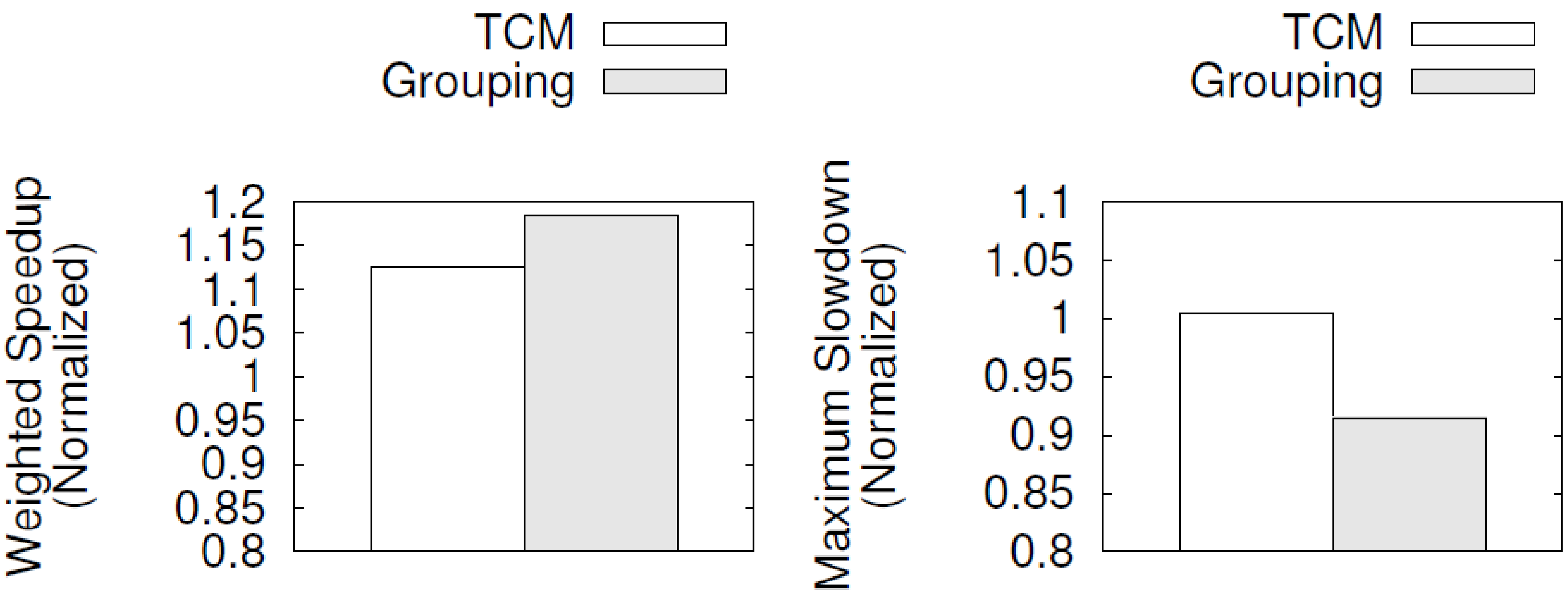


Sensitivity to Channel Count

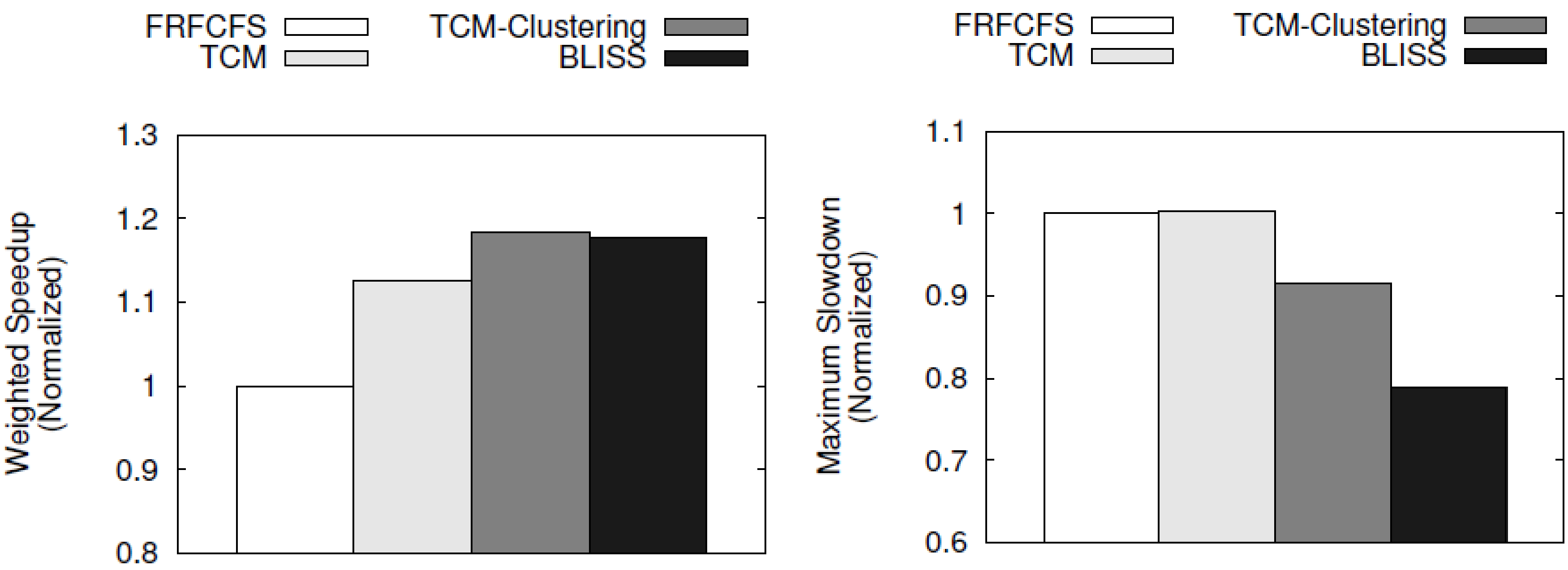




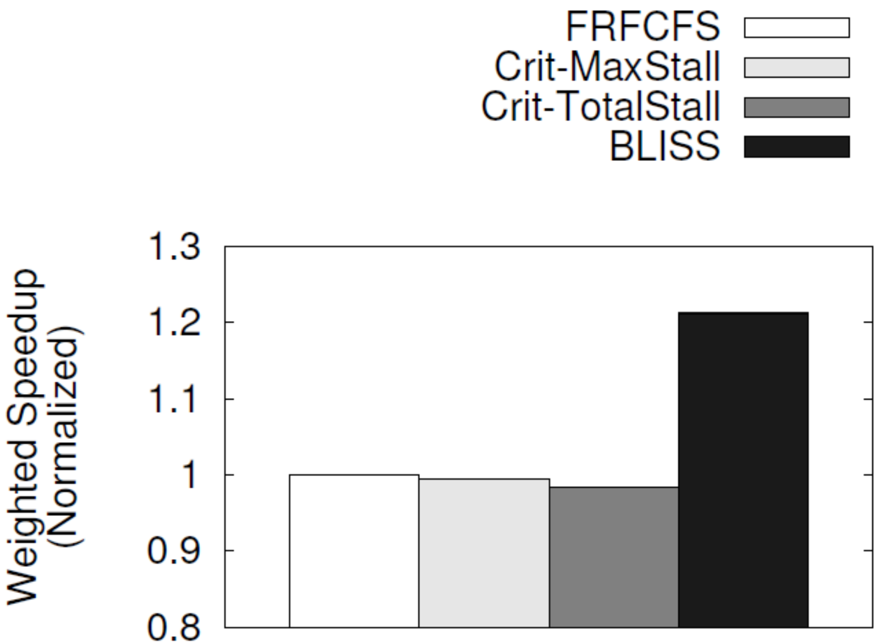
TCM vs. Grouping

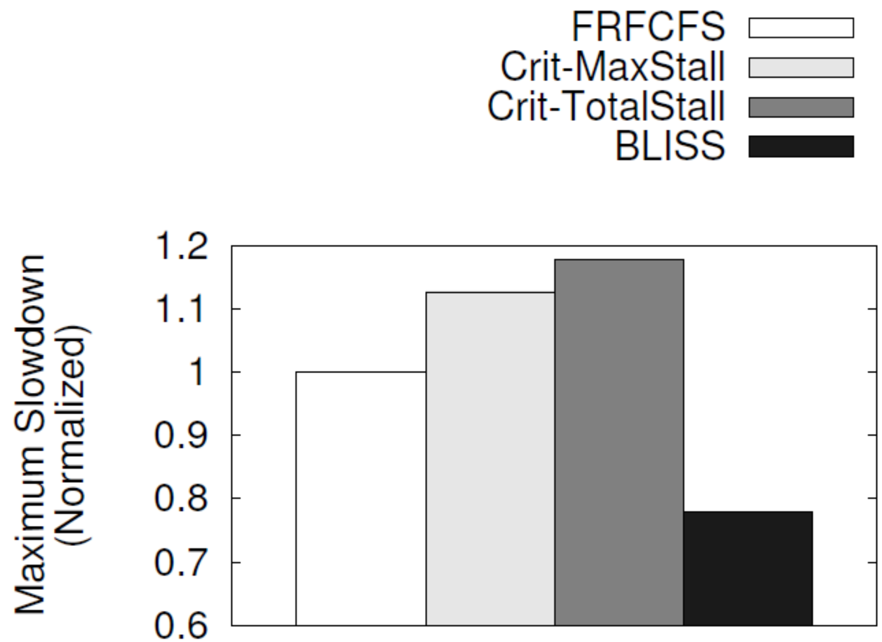


Comparison with TCM-like Clustering

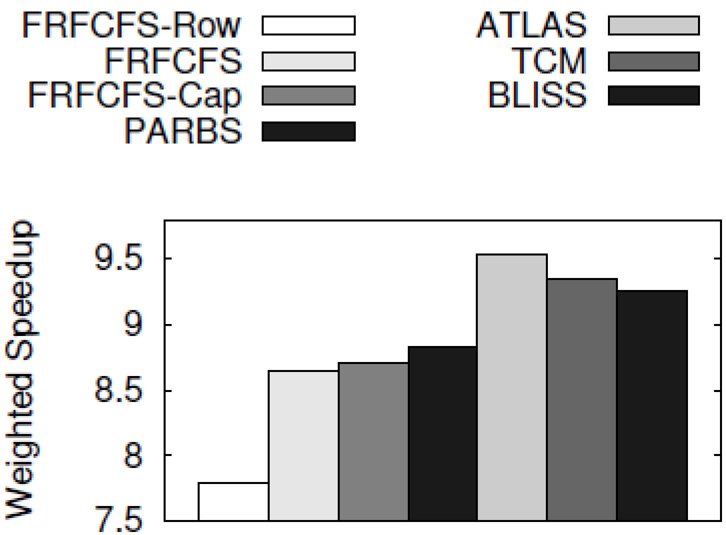


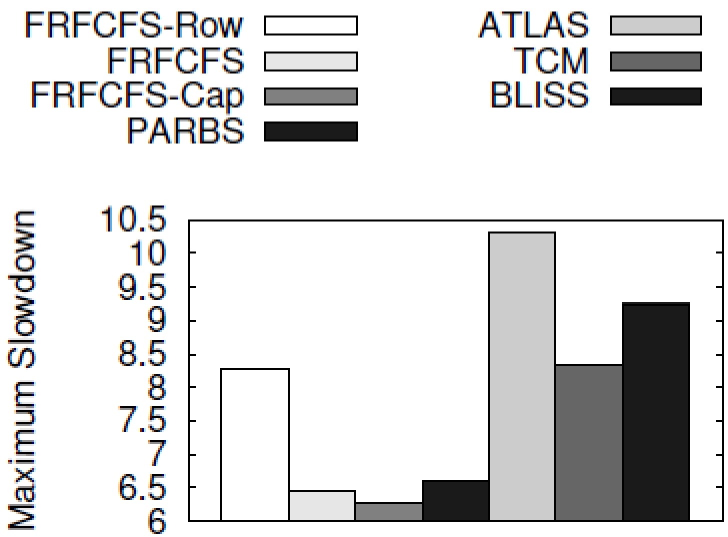
BLISS vs. Criticality-aware Scheduling





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

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

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

Estimating Request Service Rate $_{Alone}$ (RSR_{Alone})

- Divide each interval into shorter epochs

Goal: Estimate RSR_{Alone}

- At the beginning of each epoch
  - Memory controller randomly picks an application as the highest priority application

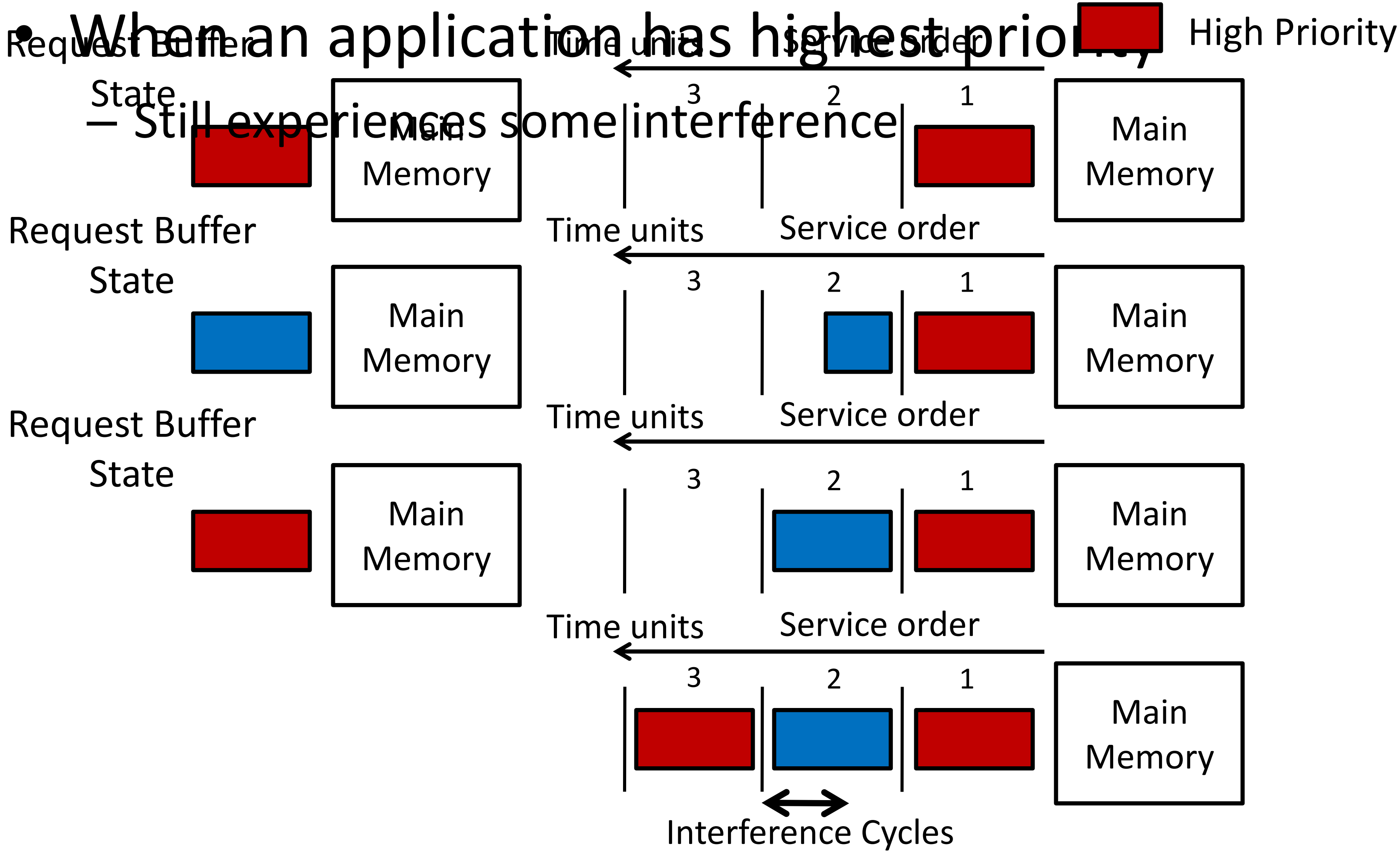
How: Periodically give each application highest priority in accessing memory

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

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

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

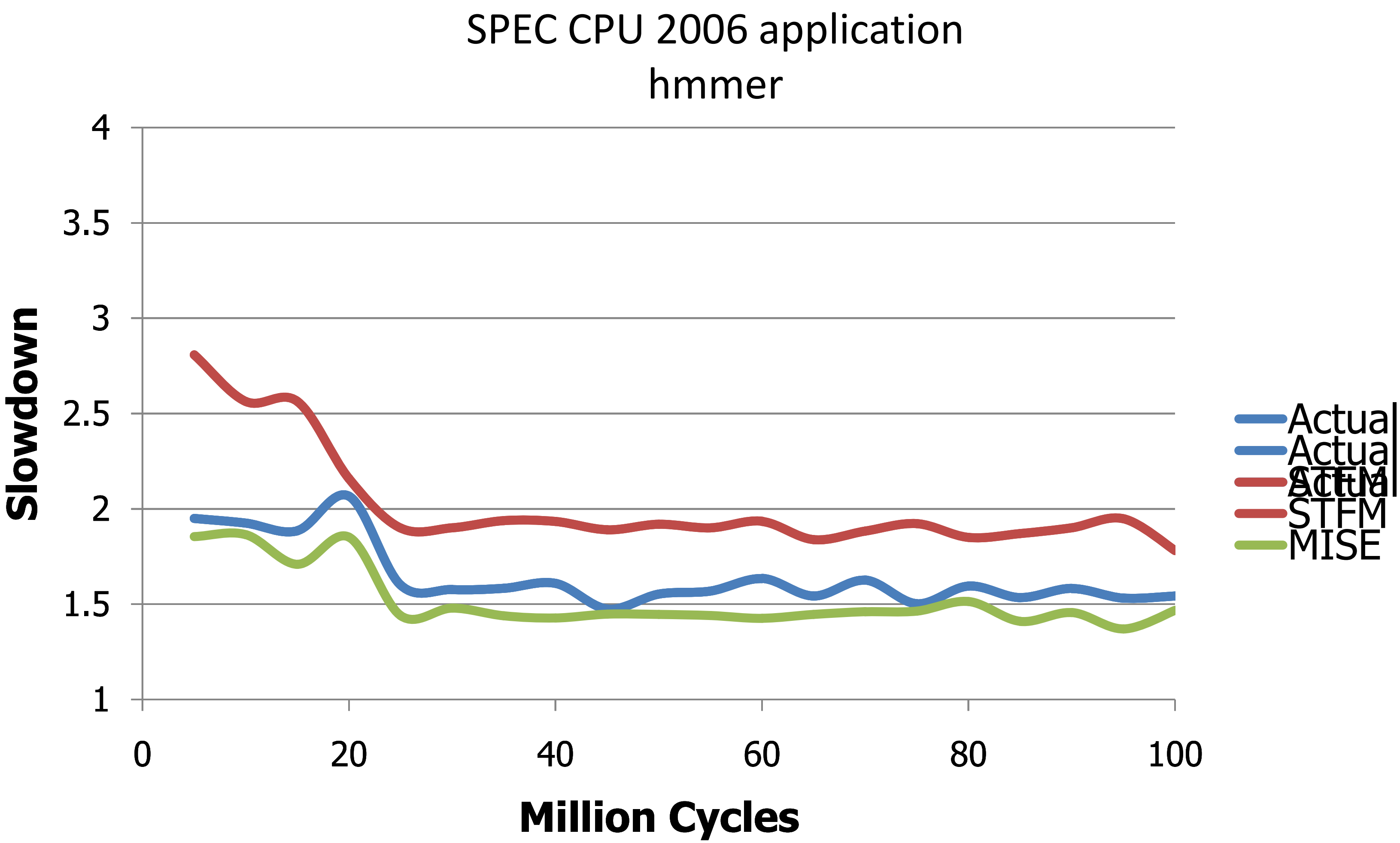
$$RSR_{Alone} = \frac{\text{Number of Requests During High Priority Epochs}}{\text{Number of Cycles Application Given High Priority} - \text{Interference Cycles}}$$

- 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

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

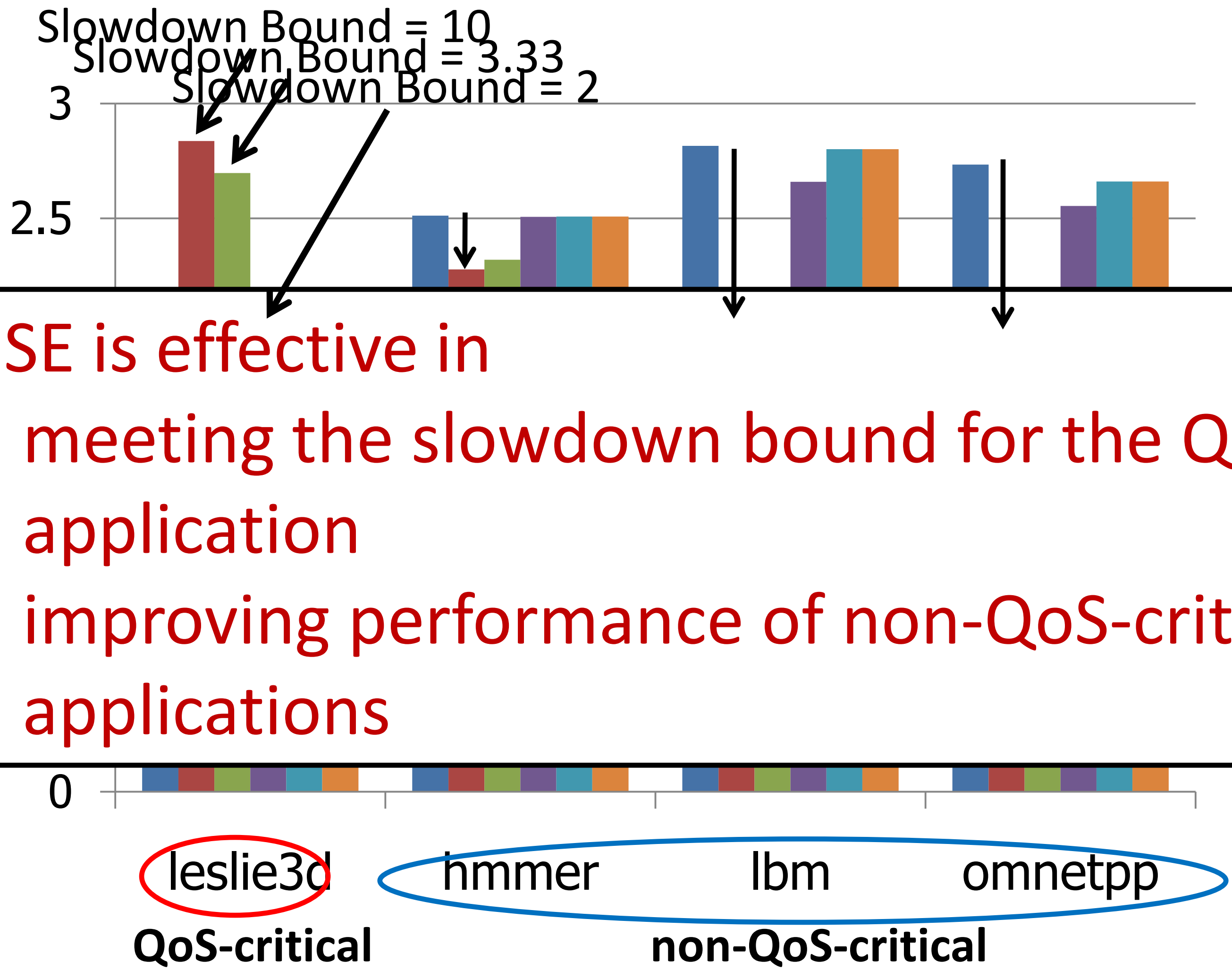
Quantitative Comparison



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

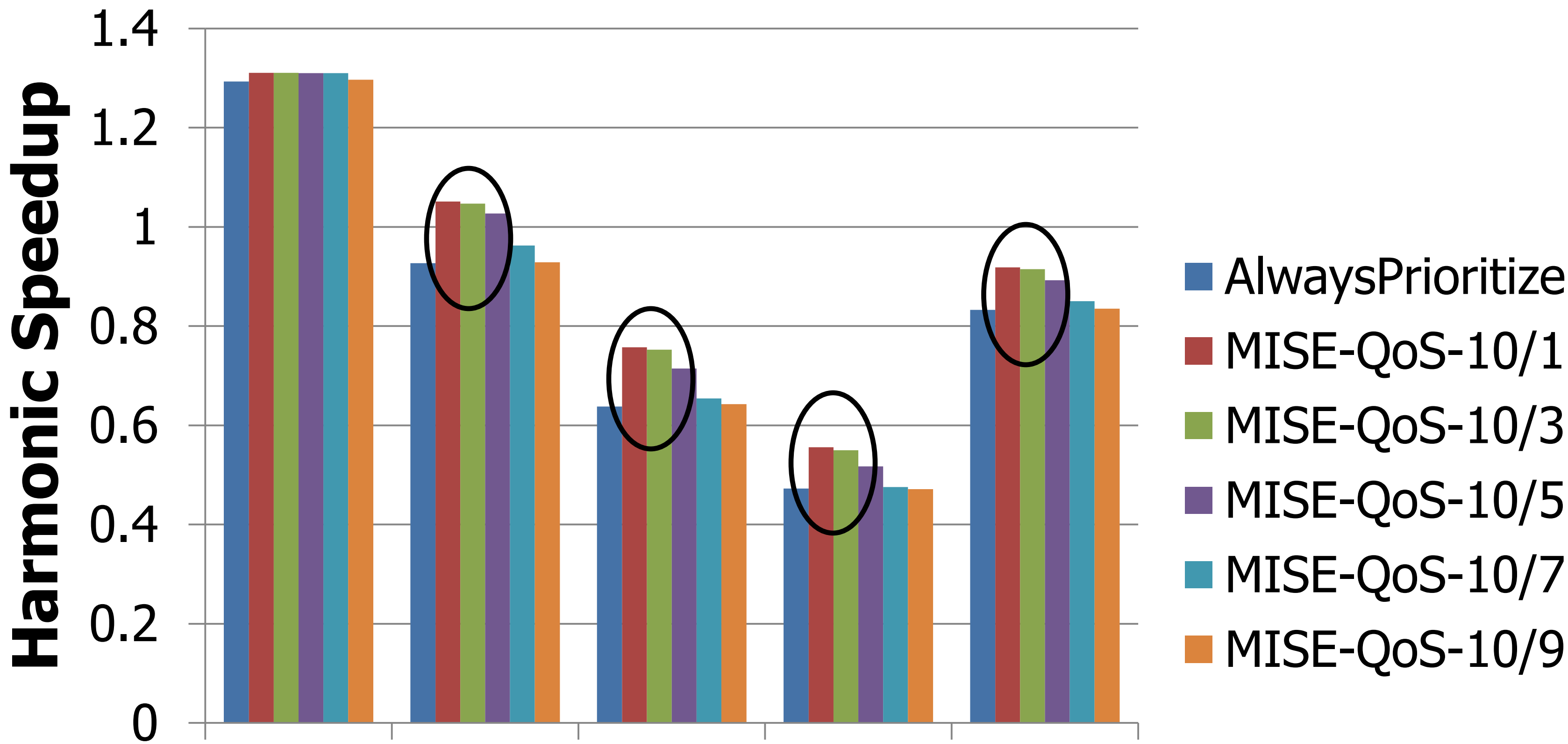
A Look at One Workload



MISE is effective in

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

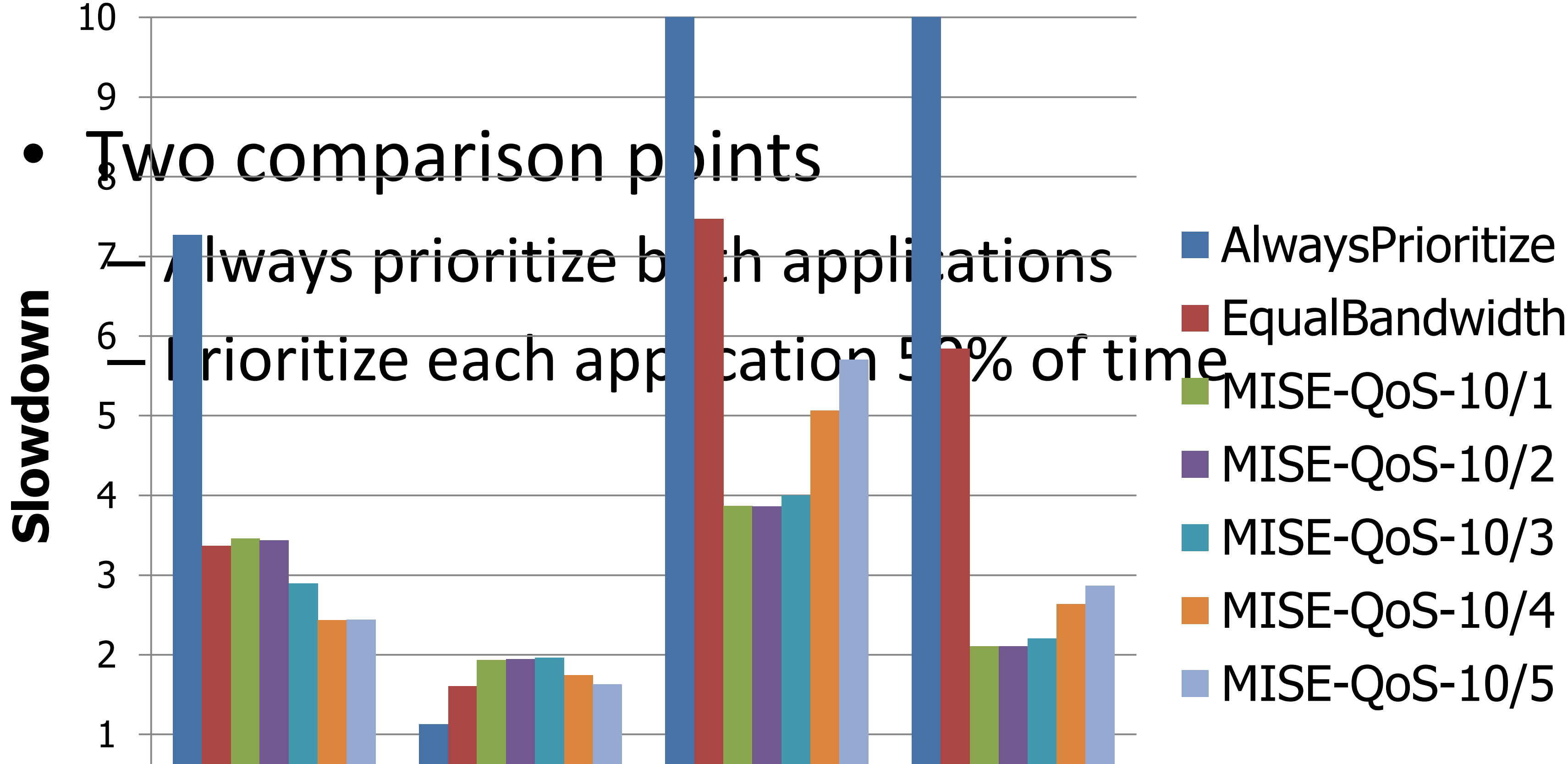
Performance of Non-QoS-Critical Applications



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

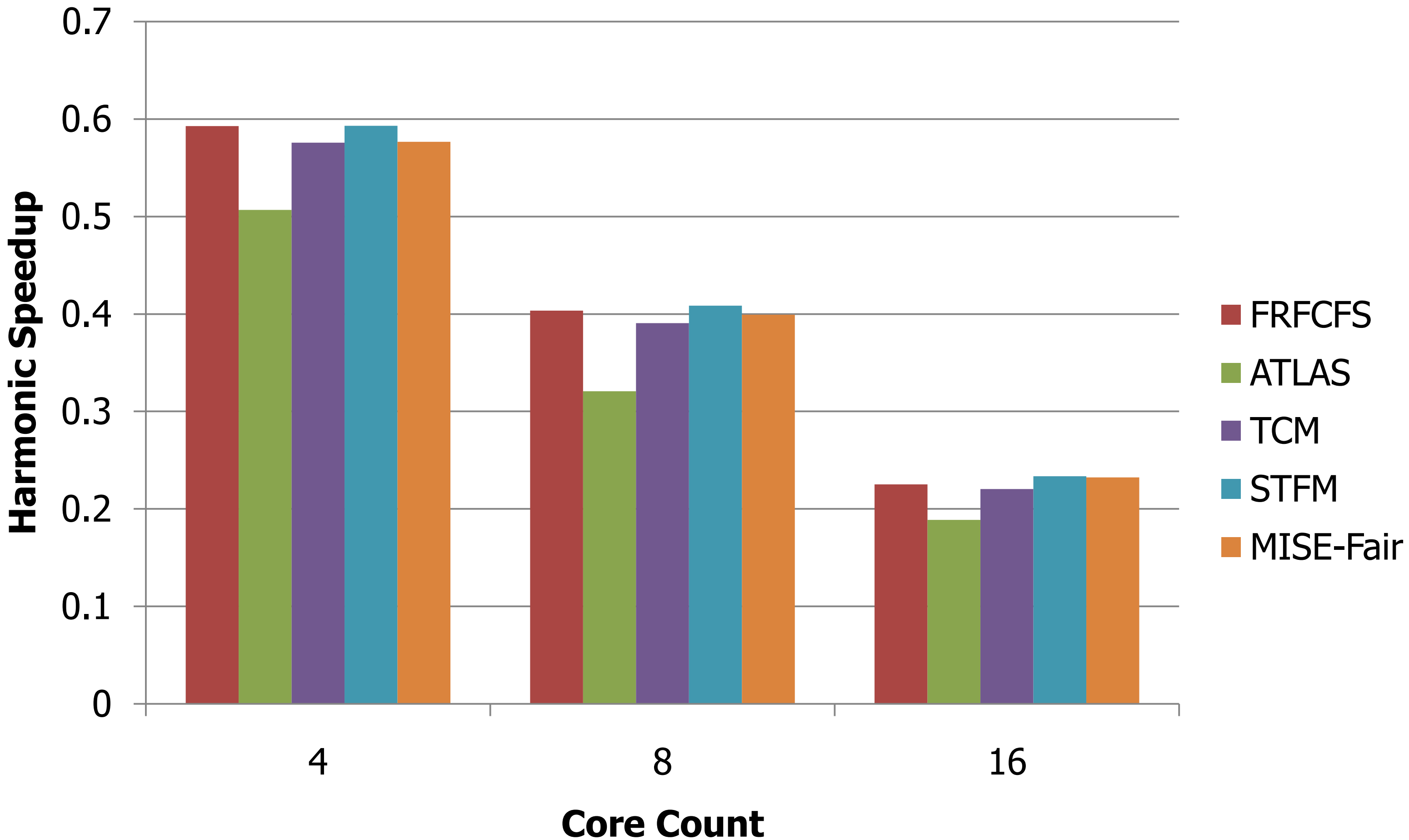
Bandwidth Redistribution

- 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

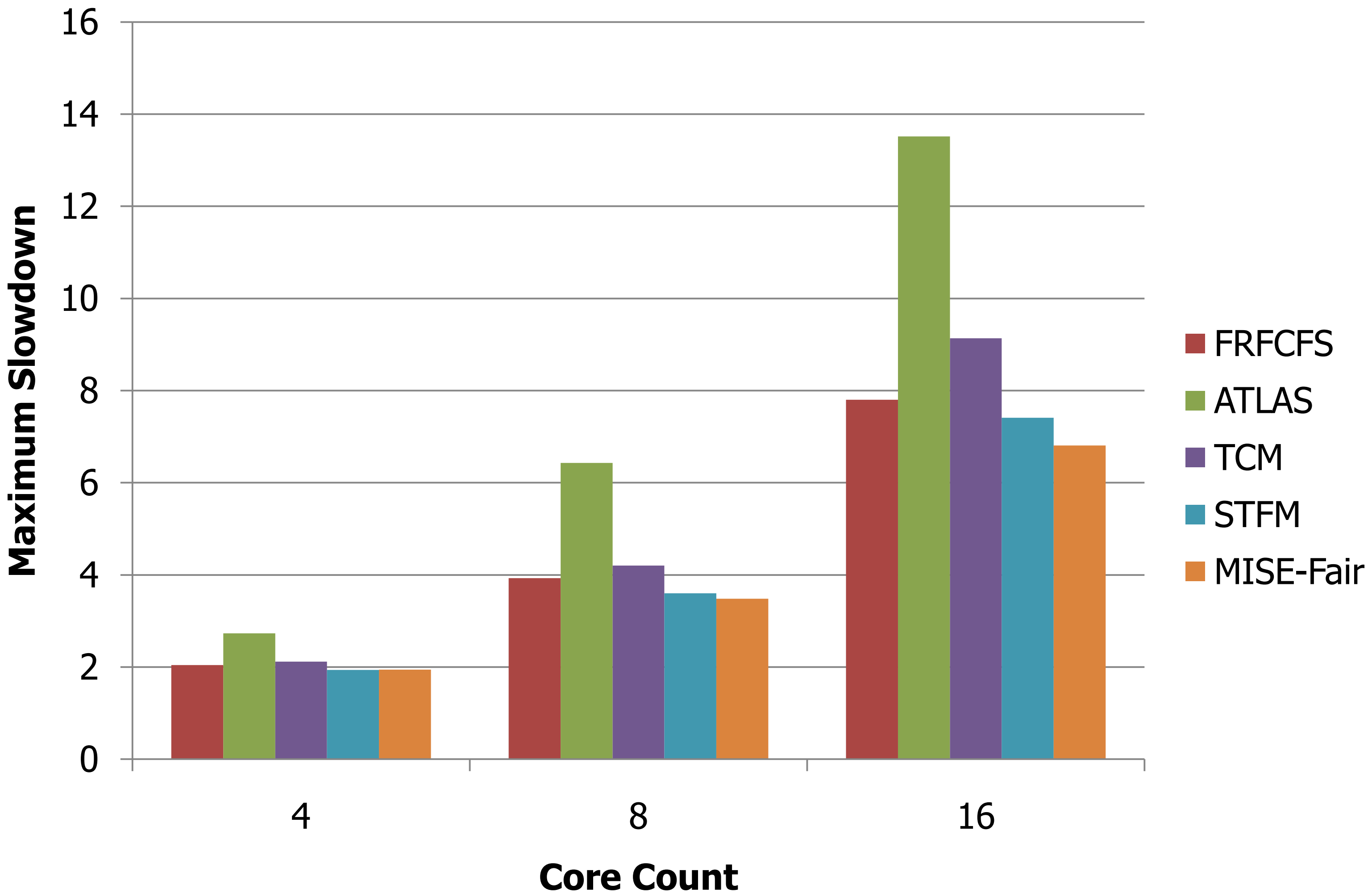
Modifying Target Bound

- 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

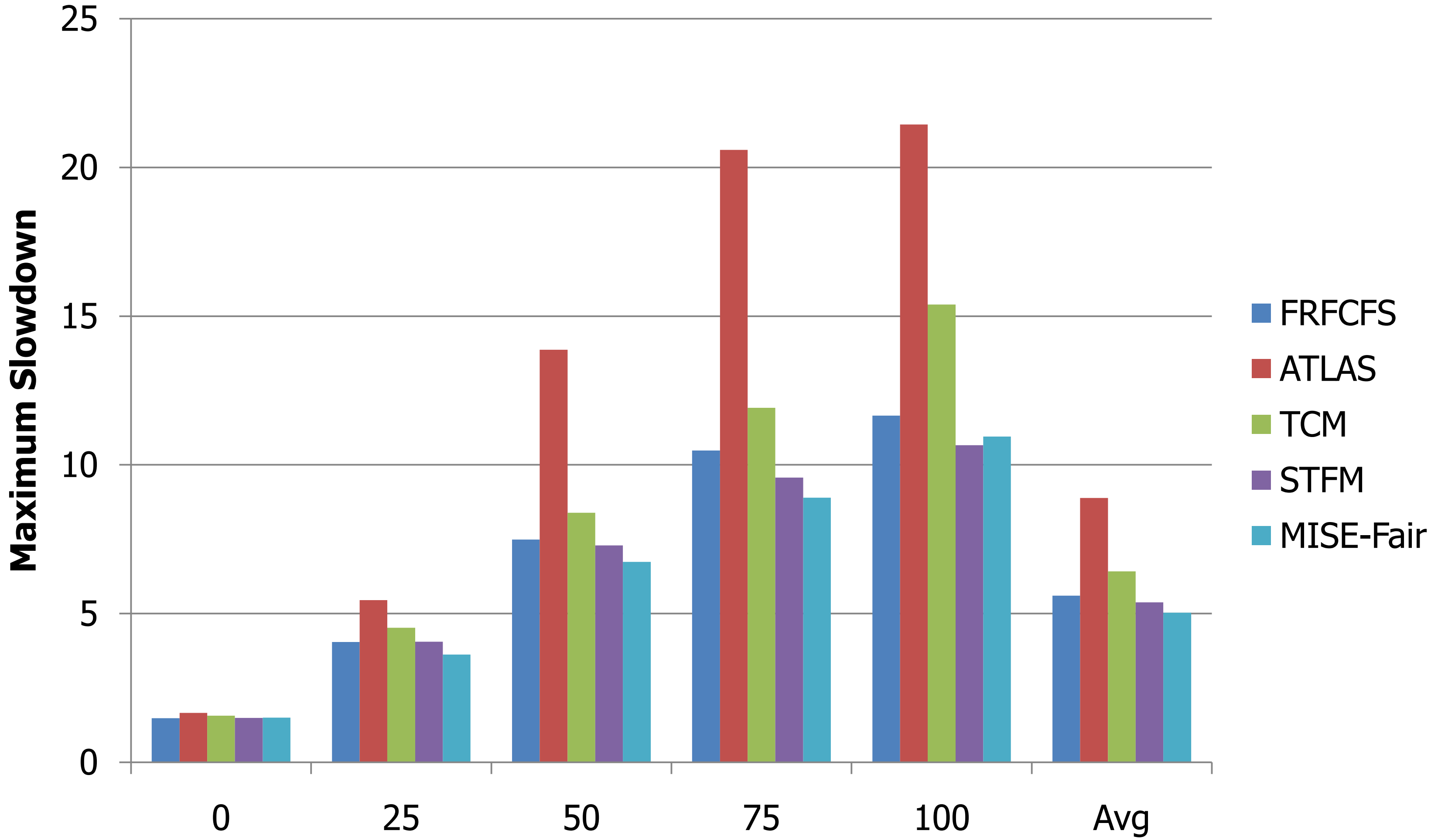
Results: Harmonic Speedup



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

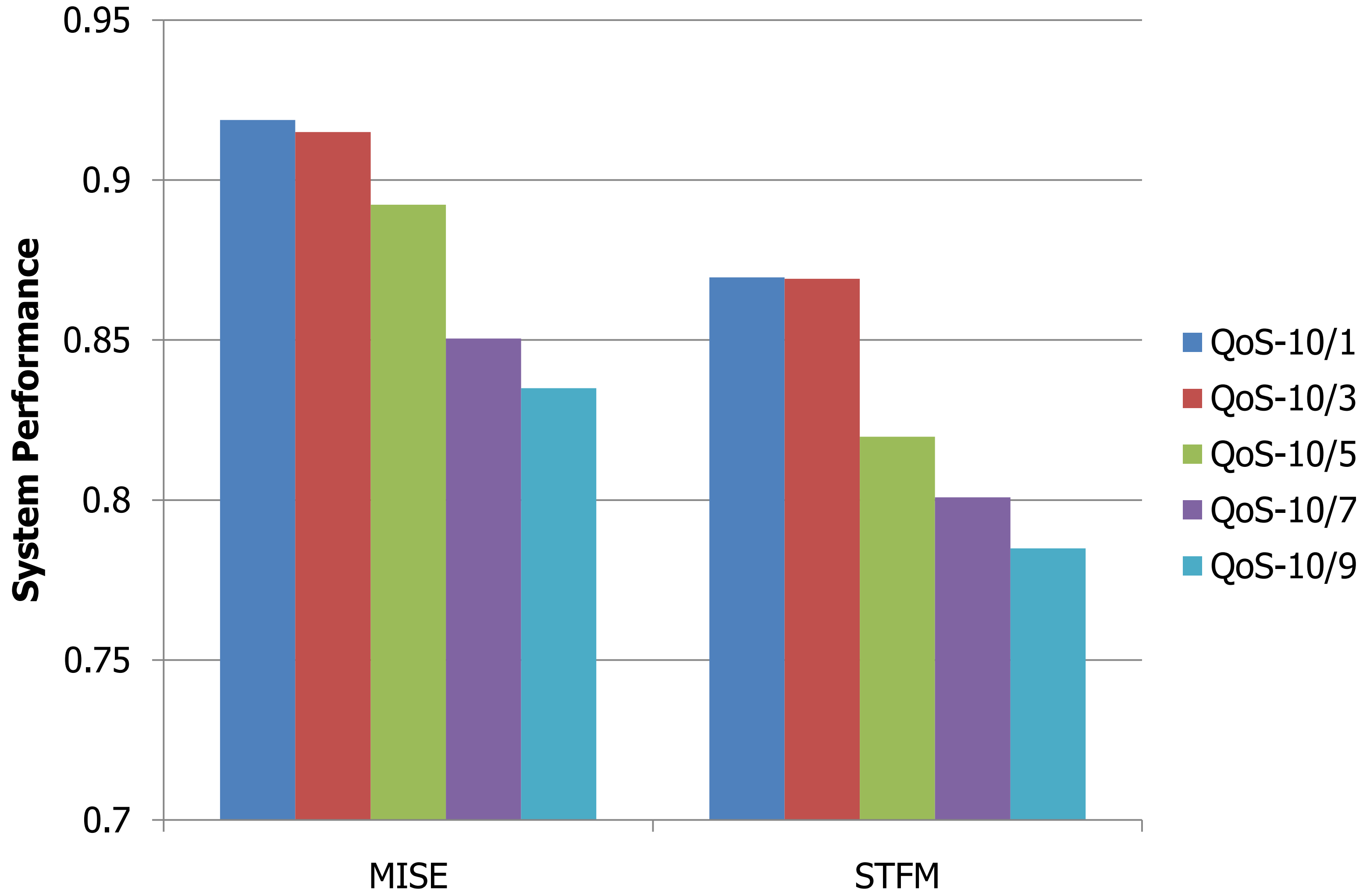
Mix No.	Benchmark 1	Benchmark 2	Benchmark 3
1	sphinx3	leslie3d	milc
2	sjeng	gcc	perlbench
3	tonto	povray	wrf
4	perlbench	gcc	povray
5	gcc	povray	leslie3d
6	perlbench	namd	lbm
7	h264ref	bzip2	libquantum
8	hmmer	lbm	omnetpp
9	sjeng	libquantum	cactusADM
10	namd	libquantum	mcf
11	xalancbmk	mcf	astar
12	mcf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

Initial Ideas

- Separate slowdown into cache and memory slowdowns
- Determine resource allocations based on cache, memory and overall slowdowns

QoS in Heterogeneous Systems

- Staged memory scheduling
  - In collaboration with Rachata Ausavarungnirun, Kevin Chang and Gabriel Lob
  - 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*

Publications