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

Core <-> Cache <-> Main Memory
The Multicore Era

Multiple applications execute in parallel
High throughput and efficiency
Challenge:
Interference at Shared Resources

Main Memory

Shared Cache

Interconnect

Core

Core

Core

Core
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

High and predictable performance can be achieved in multicore systems through simple/implementable mechanisms to mitigate and quantify shared resource interference.

Goals

Approaches
Goals and Approaches

Goals:
1. High Performance
2. Predictable Performance

Approaches:

Mitigate Interference

Quantify Interference
Focus Shared Resources in This Thesis

Main Memory

Shared Cache Capacity

Interconnect

Main Memory Bandwidth
Related Prior Work

**Mitigate Interference**
- Much explored
- Not our focus

**Quantify Interference**
- Not our focus

<table>
<thead>
<tr>
<th>Cache Capacity</th>
<th>Memory Bandwidth</th>
</tr>
</thead>
<tbody>
<tr>
<td>STFM (MICRO ’07), PARBS (ISCA ’08), ATLAS (HPCA ’10), TCM (MICRO ’11), Criticality-aware (ISCA ’13)</td>
<td>STFM (MICRO ’07)</td>
</tr>
</tbody>
</table>

**Challenge:**
- High complexity
- High inaccuracy

**FST (ASPLOS ’10), PTCA (TACO ’13)**

**Challenge:**
- High inaccuracy
<table>
<thead>
<tr>
<th>Cache Capacity</th>
<th>Memory Bandwidth</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mitigate Interference</td>
<td>Blacklisting Memory Scheduler</td>
</tr>
<tr>
<td>Much explored Not our focus</td>
<td></td>
</tr>
<tr>
<td>Quantify Interference</td>
<td></td>
</tr>
<tr>
<td>Not our focus</td>
<td>Memory Interference induced Slowdown Estimation Model and its uses</td>
</tr>
<tr>
<td>Application Slowdown Model and its uses</td>
<td></td>
</tr>
</tbody>
</table>
Outline

Mitigate Interference

- Much explored
- Not our focus

Quantify Interference

- Not our focus

Cache Capacity

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 hits

- FR-FCFS Memory Scheduler [Zuravlev 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)

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

| Req 1 | 1 |
| Req 2 | 4 |
| Req 3 | 1 |
| Req 4 | 1 |
| Req 5 | 3 |
| Req 7 | 1 |
| Req 8 | 3 |

Highest Ranked AID
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 1: Low complexity compared to 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* $\rightarrow$ *Blacklisting*
- **Deprioritize** blacklisted applications
- **Clear** blacklist periodically (1000s of cycles)

**Benefits:**
- **Lower complexity**
- **Finer grained grouping decisions** $\rightarrow$ **Lower unfairness**
The Blacklisting Memory Scheduler (ICCD ‘14)

1. Monitor

2. Blacklist

3. Prioritize

4. Clear Periodically

Simple and scalable design

Controller

<table>
<thead>
<tr>
<th>Last Req AID</th>
<th># Consecutive Requests</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>

Request Buffer

<table>
<thead>
<tr>
<th>Req 4</th>
<th>Blacklist</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Req 5</th>
<th>Blacklist</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Req 6</th>
<th>Blacklist</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Req 7</th>
<th>Blacklist</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Req 8</th>
<th>Blacklist</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
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
1. Blacklisting achieves the highest performance
2. Blacklisting balances performance and fairness
Blacklisting reduces complexity significantly
Outline

Mitigate Interference

- Cache Capacity
  - Much explored
  - Not our focus

- Memory Bandwidth
  - Blacklisting
  - Memory Scheduler

Quantify Interference

- Not our focus

Application Slowdown Model and its uses

Memory Interference induced Slowdown Estimation Model and its uses
Impact of Interference on Performance

*Alone (No interference)*

*Shared (With interference)*

**Execution time**

**Impact of Interference**
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*
<table>
<thead>
<tr>
<th>Cache Capacity</th>
<th>Mitigate Interference</th>
<th>Quantify Interference</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Much explored</td>
<td>Not our focus</td>
</tr>
<tr>
<td>Memory Bandwidth</td>
<td>Blacklisting Memory Scheduler</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Memory Interference induced Slowdown Estimation Model and its uses</td>
<td></td>
</tr>
</tbody>
</table>

Application Slowdown Model and its uses
Observation: Request Service Rate is a Proxy for Performance

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

Normalized Performance

Slowdown = \frac{Normalized Performance}{Normalized Request Service Rate}

omnetpp

mcf

astar

Shared

Alone

Rate Service Request

Rate Service Request

Slowdown

Easy

Difficult

Intel Core i7, 4 cores

29
Observation: Highest Priority Enables Request Service Rate \( RSR_{\text{Alone}} \) Estimation

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

Highest priority \( \rightarrow \) Little interference

(almost as if the application were run alone)
Observation: Highest Priority Enables Request Service Rate Estimation

1. Run alone
   - Request Buffer State
   - Time units: 3
   - Service order: 1, 2, 3
   - Main Memory

2. Run with another application
   - Request Buffer State
   - Time units: 3
   - Service order: 1, 2, 3
   - Main Memory

3. Run with another application: highest priority
   - Request Buffer State
   - Time units: 3
   - Service order: 1, 2, 3
   - 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

- No interference: Memory phase slowdown dominates overall slowdown
- With interference: 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

\[
\text{Slowdown} = (1 - \alpha) + \alpha \frac{RSR_{\text{RSR Alone}}}{RSR_{\text{RSR Shared}}}
\]
Interval Based Operation

Interval

Measure \( RSR_{\text{Shared}} \), \( \alpha \)
Estimate \( RSR_{\text{Alone}} \)

Estimate slowdown

Interval

Measure \( RSR_{\text{Shared}} \), \( \alpha \)
Estimate \( RSR_{\text{Alone}} \)

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:

\[
\text{Slowdown} = \frac{\text{Stall Time Alone}}{\text{Stall Time Shared}}
\]

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

SPEC CPU 2006 application
leslie3d

Slowdown vs Million Cycles

- Actual
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
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

<table>
<thead>
<tr>
<th></th>
<th>Predicted Met</th>
<th>Predicted Not Met</th>
</tr>
</thead>
<tbody>
<tr>
<td>QoS Bound Met</td>
<td>78.8%</td>
<td>2.1%</td>
</tr>
<tr>
<td>QoS Bound Not Met</td>
<td>2.2%</td>
<td>16.9%</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Cache Capacity</th>
<th>Mitigate Interference</th>
<th>Quantify Interference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Much explored</td>
<td>Not our focus</td>
<td>Not our focus</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Memory Bandwidth</th>
<th>Blacklisting Memory Scheduler</th>
<th>Memory Interference induced Slowdown Estimation Model and its uses</th>
</tr>
</thead>
</table>

**Application Slowdown Model and its uses**
Shared Cache Capacity Contention

Core

Core

Shared Cache Capacity

Core

Core

Main Memory
Applications evict each other’s blocks from the shared cache
Outline

Mitigate Interference

- Cache Capacity
  - Much explored
  - Not our focus

- Memory Bandwidth
  - ✔ Blacklisting Memory Scheduler

Quantify Interference

- Not our focus

✔ Memory Interference induced Slowdown Estimation Model and its uses

Application Slowdown Model and its uses
Estimating Cache and Memory Slowdowns

![Diagram showing the relationship between cores, shared cache, and main memory, with service rates indicated.](Diagram.png)
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

![Graph showing the relationship between cache access rate and slowdown for different programs: astart, lbm, and bzip2. The x-axis represents the cache access rate ratio, and the y-axis represents the slowdown. The graph shows a linear increase in slowdown as the cache access rate ratio increases.]
Challenge

How to estimate alone cache access rate?

Cache Access Rate

Main Memory

Priority

Auxiliary Tag Store

Shared Cache
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

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

**Cache Contention Cycles**: Cycles spent serving contention misses

\[
\text{Cache Contention Cycles} = \# \text{ Contention Misses} \times \text{Average Memory Service Time}
\]

*From auxiliary tag store when given high priority*

*Measured when given high priority*
Application Slowdown Model (ASM)

The diagram illustrates the relationship between core, cache access rate, and main memory. The formula for slowdown is:

$$ \text{Slowdown} = \frac{\text{Cache Access Rate}_{\text{Alone}}}{\text{Cache Access Rate}_{\text{Shared}}} $$
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
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
Goal: Partition the shared cache among applications to mitigate contention
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
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

Cache capacity-aware bandwidth allocation
Fairness and Performance Results

100 workloads

16-core system

Significant fairness benefits across different channel counts
Outline

<table>
<thead>
<tr>
<th>Mitigate Interference</th>
<th>Quantify Interference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Much explored</td>
<td>Not our focus</td>
</tr>
<tr>
<td>Not our focus</td>
<td></td>
</tr>
</tbody>
</table>

- **Cache Capacity**
  - Much explored
  - 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

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

Ranking-based application-aware schedulers cause unfair slowdowns

GemsFDTD *(high memory intensity)*

sjeng *(low memory intensity)*
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

  - Prioritizes row-buffer hits and older requests

- **FRFCFS-Cap** [Mutlu and Moscibroda, MICRO 2007]
  - Caps the number of consecutive row-buffer hits

- **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-unaware**

+ **Low complexity**
- **Low performance and fairness**

**Application-aware**

+ **High performance and fairness**
- **High complexity**
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
5% higher system performance and 21% lower maximum slowdown than TCM
Blacklisting achieves 43% lower area than TCM
Understanding Why Blacklisting Works

Blacklisting shifts the request distribution towards the right.
Harmonic Speedup

![Harmonic Speedup Graph]

- FRFCFS
- FRFCFS-Cap
- PARBS
- ATLAS
- TCM
- Blacklisting
Effect of Workload Memory Intensity

Weighted Speedup (Normalized)

Maximum Slowdown (Normalized)

FRFCFS
FRFCFS-Cap
PARBS
ATLAS
TCM
Blacklisting
Combining FRFCFS-Cap and Blacklisting

- Weighted Speedup
- Maximum Slowdown

- FRFCFS
- FRFCFS-Cap
- Blacklisting
- FRFCFS-Cap-Blacklisting
Sensitivity to Blacklisting Threshold

![Weighted Speedup Graph]

![Maximum Slowdown Graph]

- **Weighted Speedup**
  - FRFCFS
  - Blacklisting-1
  - Blacklisting-2
  - Blacklisting-4
  - Blacklisting-8
  - Blacklisting-16

- **Maximum Slowdown**
  - FRFCFS
  - Blacklisting-1
  - Blacklisting-2
  - Blacklisting-4
  - Blacklisting-8
  - Blacklisting-16
Sensitivity to Clearing Interval

![Bar Chart 1: Weighted Speedup](chart1.png)

- **Weighted Speedup**
  - FRFCFS
  - Blacklisting-1000
  - Blacklisting-10000
  - Blacklisting-100000

![Bar Chart 2: Maximum Slowdown](chart2.png)

- **Maximum Slowdown**
  - FRFCFS
  - Blacklisting-1000
  - Blacklisting-10000
  - Blacklisting-100000
Sensitivity to Core Count

Performance vs Core Count

Unfairness vs Core Count

- FRFCFS
- FRFCFS-Cap
- PARBS
- ATLAS
- TCM
- Blacklisting
Sensitivity to Channel Count

**Performance**

Channel Count

**Unfairness**

Channel Count

- FRFCFS
- FRFCFS-Cap
- PARBS
- ATLAS
- TCM
- Blacklisting
Sensitivity to Cache Size

- **Performance**
  - 512KB
  - 1MB
  - 2MB

- **Unfairness**
  - FRFCFS
  - FRFCFS-Cap
  - PARBS
  - ATLAS
  - TCM
  - Blacklisting
Performance and Fairness with Shared Cache

![Performance and Unfairness Bar Chart](chart.png)
Breakdown of Benefits

- Weighted Speedup
- Maximum Slowdown

<table>
<thead>
<tr>
<th>Method</th>
<th>FRFCFS</th>
<th>TCM</th>
<th>Grouping</th>
<th>Blacklisting</th>
</tr>
</thead>
<tbody>
<tr>
<td>Weighted Speedup</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum Slowdown</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
BLISS vs. Criticality-aware Scheduling

Weighted Speedup

Maximum Slowdown

- FRFCFS
- PARBS
- TCM
- Crit-MaxStall
- Crit-TotalStall
- Blacklisting
Sub-row Interleaving

- Maximum Slowdown
  - FRFCFS-Row
  - FRFCFS
  - FRFCFS-Cap
  - PARBS
  - ATLAS
  - TCM
  - Blacklisting

- Weighted Speedup

Graph showing performance metrics for different algorithms with bars indicating performance metrics.
MISE
Measuring $\text{RSR}_{\text{Shared}}$ and $\alpha$

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

\[
\text{RSR}_{\text{Shared}} = \frac{\text{Number of Requests Served}}{\text{Interval Length}}
\]

- **Memory Phase Fraction** ($\alpha$)
  - Count number of stall cycles at the core
  - Compute fraction of cycles stalled for memory
Estimating Request Service Rate $\text{RSR}_\text{Alone}$

- Divide each interval into shorter epochs

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

**How:** Periodically give each application highest priority in accessing memory

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

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

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

- When an application has highest priority, it still experiences some interference.
Accounting for Interference in $\text{RSR}_{\text{Alone}}$ Estimation

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

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

- **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

- Measure $\text{RSR}_{\text{Shared}}$
- Estimate $\text{RSR}_{\text{Alone}}$

$\alpha$

Estimate slowdown

Interval

- Measure $\text{RSR}_{\text{Shared}}$
- Estimate $\text{RSR}_{\text{Alone}}$

$\alpha$

Estimate slowdown

108
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 $> b +/\epsilon$, increase bandwidth allocation
  - If slowdown $< b +/\epsilon$, 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

![Bar Chart]

- Core Count: 4, 8, 16
- Maximum Slowdown:
  - FRFCFS
  - ATLAS
  - TCM
  - STFM
  - MISE-Fair

The chart shows the comparison of maximum slowdown across different core counts and various scheduling algorithms.
Sensitivity to Memory Intensity (16 cores)
# MISE: Per-Application Error

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

<table>
<thead>
<tr>
<th>Epoch Length</th>
<th>Interval Length</th>
<th>1 mil.</th>
<th>5 mil.</th>
<th>10 mil.</th>
<th>25 mil.</th>
<th>50 mil.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>120</td>
<td>65.1%</td>
<td>9.1%</td>
<td>11.5%</td>
<td>10.7%</td>
<td>8.2%</td>
</tr>
<tr>
<td>10000</td>
<td>120</td>
<td>64.1%</td>
<td>8.1%</td>
<td>9.6%</td>
<td>8.6%</td>
<td>8.5%</td>
</tr>
<tr>
<td>100000</td>
<td>120</td>
<td>64.3%</td>
<td>11.2%</td>
<td>9.1%</td>
<td>8.9%</td>
<td>9%</td>
</tr>
<tr>
<td>1000000</td>
<td>120</td>
<td>64.5%</td>
<td>31.3%</td>
<td>14.8%</td>
<td>14.9%</td>
<td>11.7%</td>
</tr>
</tbody>
</table>
## Workload Mixes

<table>
<thead>
<tr>
<th>Mix No.</th>
<th>Benchmark 1</th>
<th>Benchmark 2</th>
<th>Benchmark 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>sphinx3</td>
<td>leslie3d</td>
<td>milc</td>
</tr>
<tr>
<td>2</td>
<td>sjeng</td>
<td>gcc</td>
<td>perlbench</td>
</tr>
<tr>
<td>3</td>
<td>tonto</td>
<td>povray</td>
<td>wrf</td>
</tr>
<tr>
<td>4</td>
<td>perlbench</td>
<td>gcc</td>
<td>povray</td>
</tr>
<tr>
<td>5</td>
<td>gcc</td>
<td>povray</td>
<td>leslie3d</td>
</tr>
<tr>
<td>6</td>
<td>perlbench</td>
<td>namd</td>
<td>lbm</td>
</tr>
<tr>
<td>7</td>
<td>h264ref</td>
<td>bzip2</td>
<td>libquantum</td>
</tr>
<tr>
<td>8</td>
<td>hmmer</td>
<td>lbm</td>
<td>omnetpp</td>
</tr>
<tr>
<td>9</td>
<td>sjeng</td>
<td>libquantum</td>
<td>cactusADM</td>
</tr>
<tr>
<td>10</td>
<td>namd</td>
<td>libquantum</td>
<td>mcf</td>
</tr>
<tr>
<td>11</td>
<td>xalancbmk</td>
<td>mcf</td>
<td>astar</td>
</tr>
<tr>
<td>12</td>
<td>mcf</td>
<td>libquantum</td>
<td>leslie3d</td>
</tr>
</tbody>
</table>
## STFM’s Effectiveness in Enforcing QoS

Across 3000 data points

<table>
<thead>
<tr>
<th>QoS Bound Met</th>
<th>Predicted Met</th>
<th>Predicted Not Met</th>
</tr>
</thead>
<tbody>
<tr>
<td>QoS Bound Met</td>
<td>63.7%</td>
<td>16%</td>
</tr>
<tr>
<td>QoS Bound Not Met</td>
<td>2.4%</td>
<td>17.9%</td>
</tr>
</tbody>
</table>
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 ($\alpha$)

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

<table>
<thead>
<tr>
<th>Workload Category (Number of memory intensive applications)</th>
<th>Average Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4.3%</td>
</tr>
<tr>
<td>1</td>
<td>8.9%</td>
</tr>
<tr>
<td>2</td>
<td>21.2%</td>
</tr>
<tr>
<td>3</td>
<td>18.4%</td>
</tr>
</tbody>
</table>
Impact of Cache Capacity Contention

Cache capacity interference causes high application slowdowns
Error with Sampling
Error Distribution

![Graph showing error distribution for FST, PTCA, and ASM](chart.png)
Impact of Prefetching

![Bar chart showing impact of different prefetching methods. The Y-axis represents the slowdown estimation error, and the X-axis represents different methods: FST, PTCA, ASM. The chart shows that ASM has the lowest error, followed by FST, and PTCA has the highest error.]
Sensitivity to Epoch and Quantum Lengths

<table>
<thead>
<tr>
<th>Quantum Length</th>
<th>Epoch Length</th>
<th>10000</th>
<th>50000</th>
<th>100000</th>
</tr>
</thead>
<tbody>
<tr>
<td>10000000</td>
<td>12%</td>
<td>14%</td>
<td>16.6%</td>
<td></td>
</tr>
<tr>
<td>5000000</td>
<td>9.9%</td>
<td>10.6%</td>
<td>11.5%</td>
<td></td>
</tr>
<tr>
<td>1000000000</td>
<td>9.2%</td>
<td>9.9%</td>
<td>10.5%</td>
<td></td>
</tr>
</tbody>
</table>
Sensitivity to Core Count

![Bar chart showing sensitivity to core count for FST, PTCA, ASM, and another category. The x-axis represents the number of cores (4, 8, 16), and the y-axis represents slowdown estimation error (0% to 50%). Different categories are distinguished by different colors: FST is white, PTCA is light gray, ASM is dark gray, and another category is black. The error bars show the variation in slowdown estimation across different core counts.](image-url)
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

The graph compares the slowdown of different benchmarks under various QoS conditions. The benchmarks are h264ref, mcf, sphinx3, and soplex. The QoS conditions are Naive-QoS, ASM-QoS-2.5, ASM-QoS-3, ASM-QoS-3.5, and ASM-QoS-4. The y-axis represents the slowdown, and the x-axis represents the benchmarks.
Previous Approach: Estimate Interference Experienced Per-Request

Request Overlap Makes Interference Estimation Per-Request Difficult
Estimating Performance *Alone*

*Shared (With interference)*

![Execution time diagram with requests]

**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
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*
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