Scalable Many-Core Memory Systems

Topic 3: Memory Interference and QoS-Aware Memory Systems

Prof. Onur Mutlu

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

HiPEAC ACACES Summer School 2013
July 15-19, 2013

Carnegie Mellon
What Will You Learn in This Course?

- **Scalable Many-Core Memory Systems**
  - July 15-19, 2013

- Topic 1: Main memory basics, DRAM scaling
- Topic 2: Emerging memory technologies and hybrid memories
- Topic 3: Main memory interference and QoS
- Topic 4 (unlikely): Cache management
- Topic 5 (unlikely): Interconnects

- Major Overview Reading:
Readings and Videos
Memory Lecture Videos

- Memory Hierarchy (and Introduction to Caches)
  - http://www.youtube.com/watch?v=JBdfZ5i21cs&list=PL5PHm2jkkXmidJOd59REog9jDnPDTG6IJ&index=22

- Main Memory
  - http://www.youtube.com/watch?v=ZLCy3pG7Rc0&list=PL5PHm2jkkXmidJOd59REog9jDnPDTG6IJ&index=25

- Memory Controllers, Memory Scheduling, Memory QoS
  - http://www.youtube.com/watch?v=ZSotvL3WXmA&list=PL5PHm2jkkXmidJOd59REog9jDnPDTG6IJ&index=26
  - http://www.youtube.com/watch?v=1xe2w3_NzmI&list=PL5PHm2jkkXmidJOd59REog9jDnPDTG6IJ&index=27

- Emerging Memory Technologies
  - http://www.youtube.com/watch?v=LzfOghMKyA0&list=PL5PHm2jkkXmidJOd59REog9jDnPDTG6IJ&index=35

- Multiprocessor Correctness and Cache Coherence
  - http://www.youtube.com/watch?v=U-VZKMgItDM&list=PL5PHm2jkkXmidJOd59REog9jDnPDTG6IJ&index=32
Readings for Topic 1 (DRAM Scaling)

Readings for Topic 2 (Emerging Technologies)

Readings for Topic 3 (Memory QoS)

Readings for Topic 3 (Memory QoS)

- Ebrahimi et al., “Parallel Application Memory Scheduling,” MICRO 2011.
Readings in Flash Memory


Online Lectures and More Information

- **Online Computer Architecture Lectures**
  - [http://www.youtube.com/playlist?list=PL5PHm2jkkXmidJOD59REog9jDnPDTG6IJ](http://www.youtube.com/playlist?list=PL5PHm2jkkXmidJOD59REog9jDnPDTG6IJ)

- **Online Computer Architecture Courses**
  - Advanced: [http://www.ece.cmu.edu/~ece742/doku.php](http://www.ece.cmu.edu/~ece742/doku.php)

- **Recent Research Papers**
  - [http://users.ece.cmu.edu/~omutlu/projects.htm](http://users.ece.cmu.edu/~omutlu/projects.htm)
  - [http://scholar.google.com/citations?user=7XyGUGkAAAAJ&hl=en](http://scholar.google.com/citations?user=7XyGUGkAAAAJ&hl=en)
Main Memory Interference
Trend: Many Cores on Chip

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

**AMD Barcelona**
4 cores

**Intel Core i7**
8 cores

**IBM Cell BE**
8+1 cores

**IBM POWER7**
8 cores

**Sun Niagara II**
8 cores

**Nvidia Fermi**
448 “cores”

**Intel SCC**
48 cores, networked

**Tilera TILE Gx**
100 cores, networked
Many Cores on Chip

- What we want:
  - N times the system performance with N times the cores

- What do we get today?
Unfair Slowdowns due to Interference

Uncontrolled Interference: An Example

Multi-Core Chip

Shared DRAM Memory System

unfairness
Memory System is the Major Shared Resource

![Diagram of Memory System]

- Core 0, Core 1, Core 2, ..., Core N
- Shared Cache
- Memory Controller
- On-chip
- Off-chip
- Chip Boundary
- Threads' requests interfere
- Shared Memory Resources

(SAFARI)
Much More of a Shared Resource in Future
Inter-Thread/Application Interference

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

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

Uncontrolled Interference: An Example

Multi-Core Chip

Shared DRAM Memory System

Unfairness

DRAM MEMORY CONTROLLER

L2 CACHE

INTERCONNECT

DRAM Bank 0

DRAM Bank 1

DRAM Bank 2

DRAM Bank 3
A Memory Performance Hog

// initialize large arrays A, B
for (j=0; j<N; j++) {
    index = j*linesize; /* streaming */
    A[index] = B[index];
    ...
}

STREAM
- Sequential memory access
- Very high row buffer locality (96% hit rate)
- Memory intensive

RANDOM
- Random memory access
- Very low row buffer locality (3% hit rate)
- Similarly memory intensive

What Does the Memory Hog Do?

Memory Request Buffer

Row size: 8KB, cache block size: 64B
128 \((8KB/64B)\) requests of T0 serviced before T1


SAFARI
DRAM Controllers

- A row-conflict memory access takes significantly longer than a row-hit access

- Current controllers take advantage of the row buffer

- Commonly used scheduling policy (FR-FCFS) [Rixner 2000]*
  
  (1) Row-hit first: Service row-hit memory accesses first

  (2) Oldest-first: Then service older accesses first

- This scheduling policy aims to maximize DRAM throughput
  
  - But, it is unfair when multiple threads share the DRAM system

---


Effect of the Memory Performance Hog

Results on Intel Pentium D running Windows XP
(Similar results for Intel Core Duo and AMD Turion, and on Fedora Linux)

Greater Problem with More Cores

- Vulnerable to denial of service (DoS) [Usenix Security’07]
- Unable to enforce priorities or SLAs [MICRO’07,’10,’11, ISCA’08’11’12, ASPLOS’10]
- Low system performance [IEEE Micro Top Picks ’09,’11a,’11b,’12]

Uncontrollable, unpredictable system
Greater Problem with More Cores

- Vulnerable to denial of service (DoS) [Usenix Security’07]
- Unable to enforce priorities or SLAs [MICRO’07,’10,’11, ISCA’08’11’12, ASPLOS’10]
- Low system performance [IEEE Micro Top Picks ’09,’11a,’11b,’12]

Uncontrollable, unpredictable system
Distributed DoS in Networked Multi-Core Systems

Cores connected via packet-switched routers on chip

~5000X latency increase

How Do We Solve The Problem?

- Inter-thread interference is uncontrolled in all memory resources
  - Memory controller
  - Interconnect
  - Caches

- We need to control it
  - i.e., design an interference-aware (QoS-aware) memory system
QoS-Aware Memory Systems: Challenges

- How do we **reduce inter-thread interference**?
  - Improve system performance and core utilization
  - Reduce request serialization and core starvation

- How do we **control inter-thread interference**?
  - Provide mechanisms to enable system software to enforce QoS policies
  - While providing high system performance

- How do we **make the memory system configurable/flexible**?
  - Enable flexible mechanisms that can achieve many goals
    - Provide fairness or throughput when needed
    - Satisfy performance guarantees when needed
Designing QoS-Aware Memory Systems: Approaches

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

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

- How to schedule requests to provide
  - High system performance
  - High fairness to applications
  - Configurability to system software

- Memory controller needs to be aware of threads

Resolves memory contention by scheduling requests
QoS-Aware Memory Scheduling: Evolution
QoS-Aware Memory Scheduling: Evolution

- **Stall-time fair memory scheduling** [Mutlu+ MICRO’07]
  - Idea: Estimate and balance thread slowdowns
  - Takeaway: Proportional thread progress improves performance, especially when threads are “heavy” (memory intensive)

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

- **ATLAS memory scheduler** [Kim+ HPCA’10]
Within-Thread Bank Parallelism

Key Idea:

memory service timeline

thread A

WAIT

thread B

WAIT

thread execution timeline

SAVED CYCLES

Parallelism-Aware Batch Scheduling [ISCA’08]

- **Principle 1:** Schedule requests from a thread back to back
  - Preserves each thread’s bank parallelism
  - But, this can cause starvation...

- **Principle 2:** Group a fixed number of oldest requests from each thread into a “batch”
  - Service the batch before all other requests
  - Form a new batch when the current batch is done
  - Eliminates starvation, provides fairness
QoS-Aware Memory Scheduling: Evolution

- **Stall-time fair memory scheduling** [Mutlu+ MICRO’07]
  - Idea: Estimate and balance thread slowdowns
  - Takeaway: *Proportional thread progress improves performance, especially when threads are “heavy”* (memory intensive)

- **Parallelism-aware batch scheduling** [Mutlu+ ISCA’08, Top Picks’09]
  - Idea: Rank threads and service in rank order (to preserve bank parallelism); batch requests to prevent starvation
  - Takeaway: *Preserving within-thread bank-parallelism improves performance*; request batching improves fairness

- **ATLAS memory scheduler** [Kim+ HPCA’10]
  - Idea: Prioritize threads that have attained the least service from the memory scheduler
  - Takeaway: *Prioritizing “light” threads improves performance*
QoS-Aware Memory Scheduling: Evolution

- **Thread cluster memory scheduling** [Kim+ MICRO’10]
  - Idea: Cluster threads into two groups (latency vs. bandwidth sensitive); prioritize the latency-sensitive ones; employ a fairness policy in the bandwidth sensitive group
  - Takeaway: Heterogeneous scheduling policy that is different based on thread behavior maximizes both performance and fairness

- **Integrated Memory Channel Partitioning and Scheduling** [Muralidhara+ MICRO’11]
  - Idea: Only prioritize very latency-sensitive threads in the scheduler; mitigate all other applications’ interference via channel partitioning
  - Takeaway: Intelligently combining application-aware channel partitioning and memory scheduling provides better performance than either
QoS-Aware Memory Scheduling: Evolution

- **Parallel application memory scheduling** [Ebrahimi+ MICRO’11]
  - Idea: Identify and prioritize limiter threads of a multithreaded application in the memory scheduler; provide fast and fair progress to non-limiter threads
  - Takeaway: Carefully prioritizing between limiter and non-limiter threads of a parallel application improves performance

- **Staged memory scheduling** [Ausavarungnirun+ ISCA’12]
  - Idea: Divide the functional tasks of an application-aware memory scheduler into multiple distinct stages, where each stage is significantly simpler than a monolithic scheduler
  - Takeaway: Staging enables the design of a scalable and relatively simpler application-aware memory scheduler that works on very large request buffers
QoS-Aware Memory Scheduling: Evolution

- **MISE** [Subramanian+ HPCA’13]
  - Idea: Estimate the performance of a thread by estimating its change in memory request service rate when run alone vs. shared → use this simple model to estimate slowdown to design a scheduling policy that provides predictable performance or fairness
  - Takeaway: Request service rate of a thread is a good proxy for its performance; alone request service rate can be estimated by giving high priority to the thread in memory scheduling for a while
QoS-Aware Memory Scheduling: Evolution

- **Prefetch-aware shared resource management** [Ebrahimi+ ISCA’12] [Ebrahimi+ MICRO’09] [Lee+ MICRO’08]
  - Idea: Prioritize prefetches depending on how they affect system performance; even accurate prefetches can degrade performance of the system
  - Takeaway: Carefully controlling and prioritizing prefetch requests improves performance and fairness

- **DRAM-Aware last-level cache policies** [Lee+ HPS Tech Report’10] [Lee+ HPS Tech Report’10]
  - Idea: Design cache eviction and replacement policies such that they proactively exploit the state of the memory controller and DRAM (e.g., proactively evict data from the cache that hit in open rows)
  - Takeaway: Coordination of last-level cache and DRAM policies improves performance and fairness
Stall-Time Fair Memory Scheduling

Onur Mutlu and Thomas Moscibroda,
"Stall-Time Fair Memory Access Scheduling for Chip Multiprocessors"
40th International Symposium on Microarchitecture (MICRO),
pages 146-158, Chicago, IL, December 2007. Slides (ppt)

STFM Micro 2007 Talk
The Problem: Unfairness

- Vulnerable to denial of service (DoS) [Usenix Security'07]
- Unable to enforce priorities or SLAs [MICRO’07,’10,’11, ISCA’08’11’12, ASPLOS’10]
- Low system performance [IEEE Micro Top Picks ’09,’11a,’11b,’12]

Uncontrollable, unpredictable system
How Do We Solve the Problem?

- **Stall-time fair memory scheduling** [Mutlu+ MICRO’07]

- **Goal:** Threads sharing main memory should experience similar slowdowns compared to when they are run alone → fair scheduling
  - Also improves overall system performance by ensuring cores make “proportional” progress

- **Idea:** Memory controller estimates each thread’s slowdown due to interference and schedules requests in a way to balance the slowdowns

A DRAM system is fair if it equalizes the slowdown of equal-priority threads relative to when each thread is run alone on the same system.

DRAM-related stall-time: The time a thread spends waiting for DRAM memory

\( ST_{\text{shared}} \): DRAM-related stall-time when the thread runs with other threads

\( ST_{\text{alone}} \): DRAM-related stall-time when the thread runs alone

**Memory-slowdown** = \( \frac{ST_{\text{shared}}}{ST_{\text{alone}}} \)
- Relative increase in stall-time

*Stall-Time Fair Memory scheduler (STFM)* aims to equalize **Memory-slowdown** for interfering threads, without sacrificing performance
- Considers inherent DRAM performance of each thread
- Aims to allow proportional progress of threads
STFM Scheduling Algorithm [MICRO’ 07]

- For each thread, the DRAM controller
  - Tracks \( ST_{\text{shared}} \)
  - Estimates \( ST_{\text{alone}} \)

- Each cycle, the DRAM controller
  - Computes \( \text{Slowdown} = \frac{ST_{\text{shared}}}{ST_{\text{alone}}} \) for threads with legal requests
  - Computes \( \text{unfairness} = \frac{\text{MAX Slowdown}}{\text{MIN Slowdown}} \)

- If \( \text{unfairness} < \alpha \)
  - Use DRAM throughput oriented scheduling policy

- If \( \text{unfairness} \geq \alpha \)
  - Use fairness-oriented scheduling policy
    - (1) requests from thread with MAX Slowdown first
    - (2) row-hit first, (3) oldest-first
How Does STFM Prevent Unfairness?

<table>
<thead>
<tr>
<th>T0: Row 0</th>
<th>T1: Row 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>T0: Row 0</td>
<td>T1: Row 111</td>
</tr>
<tr>
<td>T0: Row 0</td>
<td>T0: Row 06</td>
</tr>
</tbody>
</table>

T0 Slowdown 1.00
T1 Slowdown 1.06
Unfairness 1.06
\( \alpha \) 1.05

Row Buffer
Row 161
Data
STFM Pros and Cons

- **Upsides:**
  - First work on fair multi-core memory scheduling
  - Good at providing fairness
  - Being fair improves performance

- **Downsides:**
  - Does not handle all types of interference
  - Somewhat complex to implement
  - Slowdown estimations can be incorrect
Parallelism-Aware Batch Scheduling

Onur Mutlu and Thomas Moscibroda,
"Parallelism-Aware Batch Scheduling: Enhancing both Performance and Fairness of Shared DRAM Systems"
35th International Symposium on Computer Architecture (ISCA), pages 63-74, Beijing, China, June 2008. Slides (ppt)
Another Problem due to Interference

- Processors try to tolerant the latency of DRAM requests by generating multiple outstanding requests
  - Memory-Level Parallelism (MLP)
  - Out-of-order execution, non-blocking caches, runahead execution

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

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

**Single Thread:**

Thread A: Bank 0, Row 1

Bank access latencies of the two requests overlapped
Thread stalls for ~ONE bank access latency
Bank Parallelism Interference in DRAM

Baseline Scheduler:
2 DRAM Requests

A: Compute | Stall | Stall | Compute
Bank 0

B: Compute | Stall | Stall | Compute
Bank 1

2 DRAM Requests

Thread A: Bank 0, Row 1
Thread B: Bank 1, Row 99
Thread B: Bank 0, Row 99
Thread A: Bank 1, Row 1

Bank access latencies of each thread serialized
Each thread stalls for ~TWO bank access latencies
Parallelism-Aware Scheduler

**Baseline Scheduler:**

- 2 DRAM Requests

A: Compute | Stall | Stall | Compute

Bank 0

Bank 1

B: Compute | Stall | Stall | Compute

Bank 1

Bank 0

**Parallelism-aware Scheduler:**

- 2 DRAM Requests

A: Compute | Stall | Compute

Bank 0

Bank 1

B: Compute | Stall | Stall | Compute

Bank 0

Bank 1

Saved Cycles

Average stall-time: ~1.5 bank access latencies
Parallelism-Aware Batch Scheduling (PAR-BS)

- **Principle 1: Parallelism-awareness**
  - Schedule requests from a thread (to different banks) back to back
  - Preserves each thread’s bank parallelism
  - But, this can cause starvation...

- **Principle 2: Request Batching**
  - Group a fixed number of oldest requests from each thread into a “batch”
  - Service the batch before all other requests
  - Form a new batch when the current one is done
  - Eliminates starvation, provides fairness
  - Allows parallelism-awareness within a batch

PAR-BS Components

- Request batching

- Within-batch scheduling
  - Parallelism aware
Request Batching

- Each memory request has a bit \((marked)\) associated with it

- **Batch formation:**
  - Mark up to \(Marking-Cap\) oldest requests per bank for each thread
  - Marked requests constitute the batch
  - Form a new batch when no marked requests are left

- **Marked requests are prioritized over unmarked ones**
  - No reordering of requests across batches: no starvation, high fairness

- **How to prioritize requests within a batch?**
Within-Batch Scheduling

- Can use any existing DRAM scheduling policy
  - FR-FCFS (row-hit first, then oldest-first) exploits row-buffer locality
- But, we also want to preserve intra-thread bank parallelism
  - Service each thread’s requests back to back

**HOW?**

- Scheduler computes a **ranking of threads** when the batch is formed
  - Higher-ranked threads are prioritized over lower-ranked ones
  - Improves the likelihood that requests from a thread are serviced in parallel by different banks
    - Different threads prioritized in the same order across ALL banks
How to Rank Threads within a Batch

- Ranking scheme affects system throughput and fairness

- **Maximize system throughput**
  - Minimize average stall-time of threads within the batch

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

- **Shortest stall-time first (shortest job first) ranking**
  - Provides optimal system throughput [Smith, 1956]*
  - Controller estimates each thread’s stall-time within the batch
  - Ranks threads with shorter stall-time higher

---

Shortest Stall-Time First Ranking

- **Maximum number of marked requests to any bank** (max-bank-load)
  - Rank thread with lower max-bank-load higher (~ low stall-time)
- **Total number of marked requests** (total-load)
  - Breaks ties: rank thread with lower total-load higher

Ranking:

<table>
<thead>
<tr>
<th>Bank 0</th>
<th>Bank 1</th>
<th>Bank 2</th>
<th>Bank 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>T3</td>
<td>T2</td>
<td>T3</td>
<td>T3</td>
</tr>
<tr>
<td>T1</td>
<td>T0</td>
<td>T2</td>
<td>T0</td>
</tr>
<tr>
<td>T2</td>
<td>T2</td>
<td>T1</td>
<td>T2</td>
</tr>
<tr>
<td>T3</td>
<td>T1</td>
<td>T0</td>
<td>T3</td>
</tr>
<tr>
<td>T1</td>
<td>T3</td>
<td>T2</td>
<td>T3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>max-bank-load</th>
<th>total-load</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Ranking:**

T0 > T1 > T2 > T3
Example Within-Batch Scheduling Order

Baseline Scheduling Order (Arrival order)

<table>
<thead>
<tr>
<th>Time</th>
<th>Bank 0</th>
<th>Bank 1</th>
<th>Bank 2</th>
<th>Bank 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>T1</td>
<td>T3</td>
<td>T2</td>
<td>T0</td>
</tr>
<tr>
<td>2</td>
<td>T3</td>
<td>T1</td>
<td>T2</td>
<td>T0</td>
</tr>
<tr>
<td>3</td>
<td>T1</td>
<td>T3</td>
<td>T2</td>
<td>T0</td>
</tr>
<tr>
<td>4</td>
<td>T3</td>
<td>T1</td>
<td>T2</td>
<td>T0</td>
</tr>
<tr>
<td>5</td>
<td>T3</td>
<td>T1</td>
<td>T2</td>
<td>T0</td>
</tr>
<tr>
<td>6</td>
<td>T3</td>
<td>T1</td>
<td>T2</td>
<td>T0</td>
</tr>
<tr>
<td>7</td>
<td>T3</td>
<td>T1</td>
<td>T2</td>
<td>T0</td>
</tr>
</tbody>
</table>

Par-Bs Scheduling Order

<table>
<thead>
<tr>
<th>Time</th>
<th>Bank 0</th>
<th>Bank 1</th>
<th>Bank 2</th>
<th>Bank 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>T0</td>
<td>T1</td>
<td>T2</td>
<td>T3</td>
</tr>
<tr>
<td>2</td>
<td>T1</td>
<td>T0</td>
<td>T2</td>
<td>T3</td>
</tr>
<tr>
<td>3</td>
<td>T2</td>
<td>T1</td>
<td>T0</td>
<td>T3</td>
</tr>
<tr>
<td>4</td>
<td>T2</td>
<td>T1</td>
<td>T0</td>
<td>T3</td>
</tr>
<tr>
<td>5</td>
<td>T2</td>
<td>T1</td>
<td>T0</td>
<td>T3</td>
</tr>
<tr>
<td>6</td>
<td>T2</td>
<td>T1</td>
<td>T0</td>
<td>T3</td>
</tr>
<tr>
<td>7</td>
<td>T2</td>
<td>T1</td>
<td>T0</td>
<td>T3</td>
</tr>
</tbody>
</table>

Ranking: T0 > T1 > T2 > T3

Stall times

<table>
<thead>
<tr>
<th>AVG: 5 bank access latencies</th>
</tr>
</thead>
<tbody>
<tr>
<td>T0</td>
</tr>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

Stall times

<table>
<thead>
<tr>
<th>AVG: 3.5 bank access latencies</th>
</tr>
</thead>
<tbody>
<tr>
<td>T0</td>
</tr>
<tr>
<td>1</td>
</tr>
</tbody>
</table>
Putting It Together: PAR-BS Scheduling Policy

- **PAR-BS Scheduling Policy**
  - (1) Marked requests first
  - (2) Row-hit requests first
  - (3) Higher-rank thread first (shortest stall-time first)
  - (4) Oldest first

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

- **Many more trade-offs analyzed in the paper**
Hardware Cost

- <1.5KB storage cost for
  - 8-core system with 128-entry memory request buffer

- No complex operations (e.g., divisions)

- Not on the critical path
  - Scheduler makes a decision only every DRAM cycle
Unfairness on 4-, 8-, 16-core Systems

Unfairness = MAX Memory Slowdown / MIN Memory Slowdown [MICRO 2007]
System Performance (Hmean-speedup)
PAR-BS Pros and Cons

- **Upsides:**
  - First work to identify the notion of bank parallelism destruction across multiple threads
  - Simple mechanism

- **Downsides:**
  - Implementation in multiple controllers needs coordination for best performance → too frequent coordination since batching is done frequently
  - Does not always prioritize the latency-sensitive applications
ATLAS Memory Scheduler

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

A thread alternates between two states (episodes)

- **Compute episode**: Zero outstanding memory requests → **High IPC**
- **Memory episode**: Non-zero outstanding memory requests → **Low IPC**

**Goal**: Minimize time spent in memory episodes
How to Minimize Memory Episode Time

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

Remaining length of a memory episode?

Outstanding memory requests

How much longer?

Time
Predicting Memory Episode Lengths

We discovered: past is excellent predictor for future

- Large **attained service** $\Rightarrow$ Large expected **remaining service**

Q: Why?
A: Memory episode lengths are **Pareto distributed**...
Pareto Distribution of Memory Episode Lengths

Favoring **least-attained-service** memory episode

\[ \text{Favoring memory episode which will end the soonest} \]
Least Attained Service (LAS) Memory Scheduling

**Our Approach**

Prioritize the memory episode with least-remaining-service

- Remaining service: Correlates with attained service
- Attained service: Tracked by per-thread counter

Prioritize the memory episode with least-attained-service

Least-attained-service (LAS) scheduling:
Minimize memory episode time

**Queueing Theory**

Prioritize the job with shortest-remaining-processing-time

Provably optimal

However, LAS does not consider long-term thread behavior
### Long-Term Thread Behavior

<table>
<thead>
<tr>
<th>Short-term thread behavior</th>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short memory episode</td>
<td>&gt; priority</td>
<td>Mem. episode</td>
</tr>
<tr>
<td>Long memory episode</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Prioritizing Thread 2 is more beneficial: results in very long stretches of compute episodes.
Quantum-Based Attained Service of a Thread

We divide time into large, fixed-length intervals: **quanta** (millions of cycles)
LAS Thread Ranking

**During a quantum**

Each thread’s attained service (AS) is tracked by MCs

\[ AS_i = A \text{ thread’s AS during only the } i\text{-th quantum} \]

**End of a quantum**

Each thread’s TotalAS computed as:

\[ TotalAS_i = \alpha \cdot TotalAS_{i-1} + (1 - \alpha) \cdot AS_i \]

High \( \alpha \Rightarrow \text{More bias towards history} \)

Threads are ranked, favoring threads with lower TotalAS

**Next quantum**

Threads are serviced according to their ranking
ATLAS Scheduling Algorithm

ATLAS

- Adaptive per-Thread Least Attained Service

- Request prioritization order

1. Prevent starvation: Over threshold request
2. Maximize performance: Higher LAS rank
3. Exploit locality: Row-hit request
4. Tie-breaker: Oldest request

How to coordinate MCs to agree upon a consistent ranking?
ATLAS consistently provides higher system throughput than all previous scheduling algorithms.
System Throughput: 4-MC System

# of cores increases ➔ ATLAS performance benefit increases
Properties of ATLAS

Goals

- Maximize system performance
- Scalable to large number of controllers
- Configurable by system software

Properties of ATLAS

- LAS-ranking
- Bank-level parallelism
- Row-buffer locality
- Very infrequent coordination
- Scale attained service with thread weight (in paper)

**Low complexity**: Attained service requires a single counter per thread in each MC
ATLAS Pros and Cons

- **Upsides:**
  - Good at improving performance
  - Low complexity
  - Coordination among controllers happens infrequently

- **Downsides:**
  - Lowest ranked threads get delayed significantly → high unfairness
Yoongu Kim, Michael Papamichael, Onur Mutlu, and Mor Harchol-Balter, "Thread Cluster Memory Scheduling: Exploiting Differences in Memory Access Behavior" 43rd International Symposium on Microarchitecture (MICRO), pages 65-76, Atlanta, GA, December 2010. Slides (pptx) (pdf)
Previous Scheduling Algorithms are Biased

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

<table>
<thead>
<tr>
<th>Previous Scheduling Algorithms</th>
<th>System Throughput Bias</th>
<th>Fairness Bias</th>
</tr>
</thead>
<tbody>
<tr>
<td>24 cores, 4 memory controllers, 96 workloads</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Better fairness

Better system throughput
Throughput vs. Fairness

**Throughput biased approach**
Prioritize less memory-intensive threads

**Fairness biased approach**
Take turns accessing memory

- **Good for throughput**
  - less memory intensive
  - thread A
  - thread B
  - thread C
  - higher priority

- **Does not starve**
  - thread C

- **starvation ➔ unfairness**

- **not prioritized ➔ reduced throughput**

Single policy for all threads is insufficient
Achieving the Best of Both Worlds

**For Throughput**
- Prioritize memory-non-intensive threads

**For Fairness**
- Unfairness caused by memory-intensive being prioritized over each other
  - Shuffle thread ranking
- Memory-intensive threads have different vulnerability to interference
  - Shuffle asymmetrically
1. Group threads into two *clusters*
2. Prioritize non-intensive cluster
3. Different policies for each cluster
1. Clustering
Clustering Threads

**Step 1** Sort threads by **MPKI** (misses per kiloinstruction)

- **Non-intensive cluster**
  - $\alpha T$
- **Intensive cluster**
  - Higher MPKI

$T = \text{Total memory bandwidth usage}$

**Step 2** Memory bandwidth usage $\alpha T$ divides clusters

- $\alpha < 10\%$
- **ClusterThreshold**
TCM Outline

1. Clustering

2. Between Clusters
Prioritization Between Clusters

Prioritize non-intensive cluster

- Increases system throughput
  - Non-intensive threads have greater potential for making progress
- Does not degrade fairness
  - Non-intensive threads are “light”
  - Rarely interfere with intensive threads
TCM Outline

1. Clustering
2. Between Clusters
3. Non-Intensive Cluster

Throughput
Non-Intensive Cluster

Prioritize threads according to MPKI

• Increases system throughput
  – Least intensive thread has the greatest potential for making progress in the processor
TCM Outline

1. Clustering
2. Between Clusters
3. Non-Intensive Cluster
4. Intensive Cluster
Periodically shuffle the priority of threads

- Is treating all threads equally good enough?
- **BUT:** Equal turns ≠ Same slowdown
Case Study: A Tale of Two Threads

Case Study: Two intensive threads contending

1. random-access
2. streaming

Which is slowed down more easily?

Prioritize random-access

Prioritize streaming

random-access thread is more easily slowed down
Why are Threads Different?

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

**streaming**
- All requests ➞ Same row
- High row-buffer locality

Vulnerable to interference
1. Clustering

2. Between Clusters

3. Non-Intensive Cluster

4. Intensive Cluster

Throughput

Fairness
Niceness

How to quantify difference between threads?

High Niceness Low

Bank-level parallelism
Vulnerability to interference

Row-buffer locality
Causes interference

Niceness

Shuffling: Round-Robin vs. Niceness-Aware

1. **Round-Robin** shuffling

2. **Niceness-Aware** shuffling

**What can go wrong?**

**GOOD:** Each thread prioritized once

- **Most prioritized**
  - D
  - A
  - B
  - C
  - D

- **Priority**
  - D
  - C
  - B
  - A

- **Nice thread**
  - A

- **Least nice thread**
  - D

- **ShuffleInterval**

- **Time**
Shuffling: Round-Robin vs. Niceness-Aware

1. Round-Robin shuffling

What can go wrong?

2. Niceness-Aware shuffling

GOOD: Each thread prioritized once

BAD: Nice threads receive lots of interference

ShuffleInterval

Most prioritized

<table>
<thead>
<tr>
<th>Priority</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td></td>
</tr>
</tbody>
</table>

Nice thread

Least nice thread
Shuffling: Round-Robin vs. Niceness-Aware

1. **Round-Robin** shuffling

2. **Niceness-Aware** shuffling

**GOOD**: Each thread prioritized once

<table>
<thead>
<tr>
<th>Priority</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td></td>
</tr>
</tbody>
</table>

**ShuffleInterval**
Shuffling: Round-Robin vs. Niceness-Aware

1. **Round-Robin** shuffling

2. **Niceness-Aware** shuffling

**GOOD:** Each thread prioritized once

**GOOD:** Least nice thread stays mostly deprioritized

**ShuffleInterval**
TCM Outline

1. Clustering

2. Between Clusters

3. Non-Intensive Cluster

4. Intensive Cluster

Fairness

Throughput
During quantum:
• Monitor thread behavior
  1. Memory intensity
  2. Bank-level parallelism
  3. Row-buffer locality

Beginning of quantum:
• Perform clustering
• Compute niceness of intensive threads
TCM: Scheduling Algorithm

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

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

3. **Oldest**: Older requests are prioritized
TCM: Implementation Cost

**Required storage at memory controller** (24 cores)

<table>
<thead>
<tr>
<th>Thread memory behavior</th>
<th>Storage</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPKI</td>
<td>~0.2kb</td>
</tr>
<tr>
<td>Bank-level parallelism</td>
<td>~0.6kb</td>
</tr>
<tr>
<td>Row-buffer locality</td>
<td>~2.9kb</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>&lt; 4kbits</td>
</tr>
</tbody>
</table>

- No computation is on the critical path
Previous Work

**FRFCFS** [Rixner et al., ISCA00]: Prioritizes row-buffer hits
   – Thread-oblivious $\rightarrow$ **Low throughput & Low fairness**

**STFM** [Mutlu et al., MICRO07]: Equalizes thread slowdowns
   – Non-intensive threads not prioritized $\rightarrow$ **Low throughput**

**PAR-BS** [Mutlu et al., ISCA08]: Prioritizes oldest batch of requests while preserving bank-level parallelism
   – Non-intensive threads not always prioritized $\rightarrow$ **Low throughput**

**ATLAS** [Kim et al., HPCA10]: Prioritizes threads with less memory service
   – Most intensive thread starves $\rightarrow$ **Low fairness**
TCM: Throughput and Fairness

Better fairness

Maximum Slowdown

Weighted Speedup

Better system throughput

TCM, a heterogeneous scheduling policy, provides best fairness and system throughput
TCM: Fairness-Throughput Tradeoff

When configuration parameter is varied...

Better fairness

Adjusted ClusterThreshold

Better system throughput

TCM allows robust fairness-throughput tradeoff
Operating System Support

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

• Enforcing thread weights
  – OS assigns weights to threads
  – TCM enforces thread weights within each cluster
Conclusion

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

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

• *TCM provides the best system throughput and fairness*
TCM Pros and Cons

- **Upsides:**
  - Provides both high fairness and high performance

- **Downsides:**
  - Scalability to large buffer sizes?
  - Effectiveness in a heterogeneous system?
Staged Memory Scheduling

Executive Summary

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

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

- **Solution:** Staged Memory Scheduling (SMS)
  decomposes the memory controller into three simple stages:
  1) Batch formation: maintains row buffer locality
  2) Batch scheduler: reduces interference between applications
  3) DRAM command scheduler: issues requests to DRAM

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

- Background
- Motivation
- Our Goal
- Observations
- Staged Memory Scheduling
  1) Batch Formation
  2) Batch Scheduler
  3) DRAM Command Scheduler
- Results
- Conclusion
Main Memory is a Bottleneck

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

<table>
<thead>
<tr>
<th>Memory Request Buffer</th>
<th>Core 1</th>
<th>Core 2</th>
<th>Core 3</th>
<th>Core 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Req</td>
<td>Req</td>
<td>Req</td>
<td>Req</td>
<td>Req</td>
</tr>
<tr>
<td>Req</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Memory Scheduler
Three Principles of Memory Scheduling

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

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

- Ensure that no application is starved [Mutlu and Moscibroda, MICRO’07]
  - To minimize unfairness

Table:

<table>
<thead>
<tr>
<th>Application</th>
<th>Memory Intensity (MPKI)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>10</td>
</tr>
</tbody>
</table>

Older

Newer
Outline

- Background
- Motivation: CPU-GPU Systems
- Our Goal
- Observations
- Staged Memory Scheduling
  1) Batch Formation
  2) Batch Scheduler
  3) DRAM Command Scheduler
- Results
- Conclusion
Memory Scheduling for CPU-GPU Systems

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

- GPU shares the main memory with the CPU cores

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

- How should memory scheduling be done when GPU is integrated on-chip?
Introducing the GPU into the System

- GPU occupies a significant portion of the request buffers
  - Limits the MC’s visibility of the CPU applications’ differing memory behavior → can lead to a poor scheduling decision
Naïve Solution: Large Monolithic Buffer

Memory Scheduler

To DRAM
Problems with Large Monolithic Buffer

- A large buffer requires more complicated logic to:
  - Analyze memory requests (e.g., determine row buffer hits)
  - Analyze application characteristics
  - Assign and enforce priorities

This leads to high complexity, high power, large die area

More Complex Memory Scheduler
Our Goal

- Design a new memory scheduler that is:
  - **Scalable** to accommodate a large number of requests
  - **Easy to implement**
  - Application-aware
  - Able to provide high *performance and fairness*, especially in heterogeneous CPU-GPU systems
Outline

- Background
- Motivation: CPU-GPU Systems
- Our Goal
- **Observations**
  - Staged Memory Scheduling
    1) Batch Formation
    2) Batch Scheduler
    3) DRAM Command Scheduler
- Results
- Conclusion
Key Functions of a Memory Controller

- Memory controller must consider three different things concurrently when choosing the next request:

1) Maximize row buffer hits
   - Maximize memory bandwidth

2) Manage contention between applications
   - Maximize system throughput and fairness

3) Satisfy DRAM timing constraints

- Current systems use a **centralized memory controller design** to accomplish these functions
  - Complex, especially with large request buffers
Key Idea: Decouple Tasks into Stages

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

1) Maximize row buffer hits

- **Stage 1:** Batch formation
  - Within each application, groups requests to the same row into batches

2) Manage contention between applications

- **Stage 2:** Batch scheduler
  - Schedules batches from different applications

3) Satisfy DRAM timing constraints

- **Stage 3:** DRAM command scheduler
  - Issues requests from the already-scheduled order to each bank
Outline

- Background
- Motivation: CPU-GPU Systems
- Our Goal
- Observations
- Staged Memory Scheduling
  1) Batch Formation
  2) Batch Scheduler
  3) DRAM Command Scheduler
- Results
- Conclusion
SMS: Staged Memory Scheduling

Stage 1

Batch Formation

Stage 2

Batch Scheduler

Stage 3

DRAM Command Scheduler

Bank 1
Bank 2
Bank 3
Bank 4

To DRAM
Stage 1: Batch Formation

- **Goal:** *Maximize row buffer hits*

- At each core, we want to *batch requests that access the same row* within a *limited time window*

- A batch is ready to be scheduled under two conditions
  1) When the next request accesses a different row
  2) When the time window for batch formation expires

- Keep this stage simple by using *per-core FIFOs*
Stage 1: Batch Formation Example

Time window expires

Batch Boundary

Next request goes to a different row

To Stage 2 (Batch Scheduling)
SMS: Staged Memory Scheduling

Stage 1
- Batch Formation

Stage 2
- Batch Scheduler

Stage 3
- DRAM Command Scheduler
  - Bank 1
  - Bank 2
  - Bank 3
  - Bank 4

To DRAM
Stage 2: Batch Scheduler

- **Goal:** Minimize interference between applications

- Stage 1 forms batches **within each application**
- Stage 2 schedules batches **from different applications**
  - Schedules the oldest batch from each application

- Question: Which application’s batch should be scheduled next?

- **Goal:** Maximize system performance and fairness
  - To achieve this goal, the batch scheduler chooses between two different policies
Stage 2: Two Batch Scheduling Algorithms

- **Shortest Job First (SJF)**
  - Prioritize the applications with the fewest outstanding memory requests because *they make fast forward progress*
  - **Pro:** Good system performance and fairness
  - **Con:** GPU and memory-intensive applications get deprioritized

- **Round-Robin (RR)**
  - Prioritize the applications in a round-robin manner to ensure *memory-intensive applications can make progress*
  - **Pro:** GPU and memory-intensive applications are treated fairly
  - **Con:** GPU and memory-intensive applications significantly slow down others
Stage 2: Batch Scheduling Policy

- The importance of the GPU varies between systems and over time → Scheduling policy needs to adapt to this

- **Solution**: Hybrid Policy

- At every cycle:
  - With probability $p$: **Shortest Job First** → Benefits the CPU
  - With probability $1-p$: **Round-Robin** → Benefits the GPU

- System software can configure $p$ based on the importance/weight of the GPU
  - Higher GPU importance → Lower $p$ value
SMS: Staged Memory Scheduling

Stage 1
- Batch Formation

Stage 2
- Batch Scheduler

Stage 3
- DRAM Command Scheduler
  - Bank 1
  - Bank 2
  - Bank 3
  - Bank 4
  - To DRAM
Stage 3: DRAM Command Scheduler

- High level policy decisions have already been made by:
  - Stage 1: Maintains row buffer locality
  - Stage 2: Minimizes inter-application interference

- Stage 3: No need for further scheduling

- Only goal: *service requests while satisfying DRAM timing constraints*

- Implemented as *simple per-bank FIFO queues*
Putting Everything Together

Stage 1: Batch Formation

Stage 2: Batch Scheduler

Stage 3: DRAM Command Scheduler

Current Batch Scheduling Policy: RR
Complexity

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

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

* Based on a Verilog model using 180nm library
Outline

- Background
- Motivation: CPU-GPU Systems
- Our Goal
- Observations
- Staged Memory Scheduling
  1) Batch Formation
  2) Batch Scheduler
  3) DRAM Command Scheduler
- Results
- Conclusion
Methodology

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

- **Workloads**
  - CPU: SPEC CPU 2006
  - GPU: Recent games and GPU benchmarks
  - 7 workload categories based on the memory-intensity of CPU applications
    - Low memory-intensity (L)
    - Medium memory-intensity (M)
    - High memory-intensity (H)
Comparison to Previous Scheduling Algorithms

- **FR-FCFS** [Rixner+, ISCA’00]
  - Prioritizes row buffer hits
  - Maximizes DRAM throughput
  - Low multi-core performance \(\Leftarrow\) Application unaware

- **ATLAS** [Kim+, HPCA’10]
  - Prioritizes latency-sensitive applications
  - Good multi-core performance
  - Low fairness \(\Leftarrow\) Deprioritizes memory-intensive applications

- **TCM** [Kim+, MICRO’10]
  - Clusters low and high-intensity applications and treats each separately
  - Good multi-core performance and fairness
  - Not robust \(\Leftarrow\) Misclassifies latency-sensitive applications
Evaluation Metrics

- CPU performance metric: Weighted speedup

\[ CPU_{WS} = \sum \frac{IPC_{Shared}}{IPC_{Alone}} \]

- GPU performance metric: Frame rate speedup

\[ GPU_{Speedup} = \frac{FrameRate_{Shared}}{FrameRate_{Alone}} \]

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

\[ CGWS = CPU_{WS} + GPU_{Speedup} \times GPU_{Weight} \]
Evaluated System Scenarios

- CPU-focused system
- GPU-focused system
Evaluated System Scenario: CPU Focused

- GPU has low weight (weight = 1)

\[
CGWS = CPU_{WS} + GPU_{Speedup} \times GPU_{Weight}
\]

- Configure SMS such that \( p \), SJF probability, is set to 0.9
  - Mostly uses SJF batch scheduling \( \rightarrow \) prioritizes latency-sensitive applications (mainly CPU)
SJF batch scheduling policy allows latency-sensitive applications to get serviced as fast as possible.

SMS is much less complex than previous schedulers.

Performance: CPU-Focused System

+17.2% over ATLAS
Evaluated System Scenario: GPU Focused

- GPU has high weight (weight = 1000)

\[
CGWS = CPU_{WS} + GPU_{Speedup} \times GPU_{Weight}
\]

- Configure SMS such that \( p \), SJF probability, is set to 0
  - Always uses round-robin batch scheduling \( \rightarrow \) prioritizes memory-intensive applications (GPU)
Round-robin batch scheduling policy schedules GPU requests more frequently

Performance: GPU-Focused System

SMS is much less complex than previous schedulers

+1.6% over FR-FCFS
Performance at Different GPU Weights

![Graph showing system performance at different GPU weights for various schedulers: ATLAS, TCM, FR-FCFS. The graph highlights the best previous scheduler.](image-url)
At every GPU weight, SMS outperforms the best previous scheduling algorithm for that weight.
Additional Results in the Paper

- Fairness evaluation
  - 47.6% improvement over the best previous algorithms

- Individual CPU and GPU performance breakdowns

- CPU-only scenarios
  - Competitive performance with previous algorithms

- Scalability results
  - SMS’ performance and fairness scales better than previous algorithms as the number of cores and memory channels increases

- Analysis of SMS design parameters
Outline

- Background
- Motivation: CPU-GPU Systems
- Our Goal
- Observations
- Staged Memory Scheduling
  1) Batch Formation
  2) Batch Scheduler
  3) DRAM Command Scheduler
- Results
- Conclusion
Conclusion

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

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

- **Solution:** Staged Memory Scheduling (SMS) decomposes the memory controller into three simple stages:
  1) Batch formation: maintains row buffer locality
  2) Batch scheduler: reduces interference between applications
  3) DRAM command scheduler: issues requests to DRAM

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

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

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

Lavanya Subramanian, Vivek Seshadri,
Yoongu Kim, Ben Jaiyen, Onur Mutlu
Main Memory Interference is a Problem
An application’s performance depends on which application it is running with
Need for Predictable Performance

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

Example 1: In mobile systems
- Interactive applications run with non-interactive applications
- Need to guarantee performance for interactive applications

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

Our Goal: Predictable performance in the presence of memory interference
1. Estimate Slowdown

2. Control Slowdown
Outline

1. Estimate Slowdown
   - Key Observations
   - Implementation
   - MISE Model: Putting it All Together
   - Evaluating the Model

2. Control Slowdown
   - Providing Soft Slowdown Guarantees
   - Minimizing Maximum Slowdown
Slowdown: Definition

\[
\text{Slowdown} = \frac{\text{Performance}_{\text{Alone}}}{\text{Performance}_{\text{Shared}}}
\]
Key Observation 1

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

Slowdown = \( \frac{\text{Normalized Performance}_{\text{Shared}}}{\text{Normalized Performance}_{\text{Alone}}} \)

- Easy
- Harder

Intel Core i7, 4 cores
Mem. Bandwidth: 8.5 GB/s
Key Observation 2

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

Highest priority $\rightarrow$ 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

Request Buffer State

Main Memory

Time units

Service order

Main Memory

Main Memory

Main Memory

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

\[
\text{Slowdown} = \frac{\text{Request Service Rate \text{ Alone} (RSR_{\text{Alone}})}}{\text{Request Service Rate \text{ Shared} (RSR_{\text{Shared}})}}
\]
Key Observation 3

- Memory-bound application

![Diagram showing compute phase and memory phase with and without interference]

- Memory phase slowdown dominates overall slowdown
Key Observation 3

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

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

1. Estimate Slowdown
   - Key Observations
   - Implementation
   - MISE Model: Putting it All Together
   - Evaluating the Model

2. Control Slowdown
   - Providing Soft Slowdown Guarantees
   - Minimizing Maximum Slowdown
Interval Based Operation

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

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

Estimate slowdown

Estimate slowdown
Measuring $RSR_{\text{Shared}}$ and $\alpha$

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

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

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

- Divide each interval into shorter epochs

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

- At the end of an interval, for each application, estimate highest priority in accessing memory

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

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

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

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

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

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

- A cycle is an interference cycle if
  - a request from the highest priority application is waiting in the request buffer \textit{and}
  - another application’s request was issued previously
Outline

1. Estimate Slowdown
   - Key Observations
   - Implementation
   - MISE Model: Putting it All Together
   - Evaluating the Model

2. Control Slowdown
   - Providing Soft Slowdown Guarantees
   - Minimizing Maximum Slowdown
MISE Model: Putting it All Together

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

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

Estimate slowdown

Estimate slowdown
Outline

1. Estimate Slowdown
   - Key Observations
   - Implementation
   - MISE Model: Putting it All Together
   - Evaluating the Model

2. Control Slowdown
   - Providing Soft Slowdown Guarantees
   - Minimizing Maximum Slowdown
Previous Work on Slowdown Estimation

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

- Basic Idea:

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

Count number of cycles application receives interference
Two Major Advantages of MISE Over STFM

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

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

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

- Workloads
  - SPEC CPU2006
  - 300 multi programmed workloads
Quantitative Comparison

SPEC CPU 2006 application
leslie3d

Slowdown vs. Million Cycles

- Actual

SAFARI
Comparison to STFM

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

1. Estimate Slowdown
   - Key Observations
   - Implementation
   - MISE Model: Putting it All Together
   - Evaluating the Model

2. Control Slowdown
   - Providing Soft Slowdown Guarantees
   - Minimizing Maximum Slowdown
Providing “Soft” Slowdown Guarantees

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

- **Basic Idea**
  - Allocate **just enough bandwidth to QoS-critical application**
  - Assign **remaining bandwidth to other applications**
MISE-QoS: Mechanism to Provide Soft QoS

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

- Each application (25 applications in total) considered the QoS-critical application
- Run with 12 sets of co-runners of different memory intensities
- Total of 300 multiprogrammed workloads
- Each workload run with 10 slowdown bound values
- Baseline memory scheduling mechanism
  - Always prioritize QoS-critical application
    [Iyer+, SIGMETRICS 2007]
  - Other applications’ requests scheduled in FRFCFS order
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.
When slowdown bound is 10/3, MISE-QoS improves system performance by 10%
Outline

1. Estimate Slowdown
   - Key Observations
   - Implementation
   - MISE Model: Putting it All Together
   - Evaluating the Model

2. Control Slowdown
   - Providing Soft Slowdown Guarantees
   - Minimizing Maximum Slowdown
Other Results in the Paper

- Sensitivity to model parameters
  - Robust across different values of model parameters

- Comparison of STFM and MISE models in enforcing soft slowdown guarantees
  - MISE significantly more effective in enforcing guarantees

- Minimizing maximum slowdown
  - MISE improves fairness across several system configurations
Summary

- Uncontrolled memory interference slows down applications unpredictably
- Goal: Estimate and control slowdowns
- Key contribution
  - MISE: An accurate slowdown estimation model
  - Average error of MISE: 8.2%
- Key Idea
  - Request Service Rate is a proxy for performance
  - Request Service Rate \(^{\text{Alone}}\) estimated by giving an application highest priority in accessing memory
- Leverage slowdown estimates to control slowdowns
  - Providing soft slowdown guarantees
  - Minimizing maximum slowdown
MISE: Providing Performance Predictability in Shared Main Memory Systems

Lavanya Subramanian, Vivek Seshadri, Yoongu Kim, Ben Jaiyen, Onur Mutlu
Some Current Directions

- Building a comprehensive slowdown estimation model
  - Performance predictability with other shared resources
  - Performance predictability in heterogeneous systems
  - Interaction with power and energy consumption

- Integrated techniques for enforcing performance levels
  - Scheduling, partitioning, prioritization, interleaving, ...

- Exploiting slowdown information in software
  - Admission control
  - Migration policies
  - Billing policies
Designing QoS-Aware Memory Systems: Approaches

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

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

Eiman Ebrahimi, Chang Joo Lee, Onur Mutlu, and Yale N. Patt,
"Fairness via Source Throttling: A Configurable and High-Performance Fairness Substrate for Multi-Core Memory Systems"
Many Shared Resources

Core 0

Core 1

Core 2

... Core N

Shared Cache

Memory Controller

DRAM Bank 0

DRAM Bank 1

DRAM Bank 2

... DRAM Bank K

On-chip

Off-chip

Chip Boundary

Shared Memory Resources
The Problem with “Smart Resources”

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

- Explicitly coordinating mechanisms for different resources requires complex implementation.

- How do we enable fair sharing of the entire memory system by controlling interference in a coordinated manner?
An Alternative Approach: Source Throttling

- Manage inter-thread interference at the **cores**, not at the shared resources
- Dynamically estimate unfairness in the memory system
- Feed back this information into a controller
- Throttle cores’ memory access rates accordingly
  - Whom to throttle and by how much depends on performance target (throughput, fairness, per-thread QoS, etc)
  - E.g., if unfairness > system-software-specified target then **throttle down** core causing unfairness & **throttle up** core that was unfairly treated

Queue of requests to shared resources

Unmanaged Interference

Oldest

Shared Memory Resources

Request Generation Order:
A1, A2, A3, A4, B1

Core A’s stall time
Core B’s stall time

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

Save Cycles Core B

Extra Cycles Core A

Queue of requests to shared resources

Fair Source Throttling

Compute Stall on A1 Stall on A2 Stall on A3 Stall on A4

A: Compute Stall wait waiting for shared resources

B: Stall on B1

Dynamically detect application A’s interference for application B and throttle down application A
Fairness via Source Throttling (FST)

- Two components (interval-based)

- Run-time unfairness evaluation (in hardware)
  - Dynamically estimates the unfairness in the memory system
  - Estimates which application is slowing down which other

- Dynamic request throttling (hardware or software)
  - Adjusts how aggressively each core makes requests to the shared resources
  - Throttles down request rates of cores causing unfairness
    - Limit miss buffers, limit injection rate
Fairness via Source Throttling (FST)

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

if (Unfairness Estimate > Target)
{ 1-Throttle down App-interfering 
  2-Throttle up App-slowest
}
Fairness via Source Throttling (FST)

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

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

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

- Slowdown of application \( i \) = \frac{T_{i}^{\text{Shared}}}{T_{i}^{\text{Alone}}}

- How can \( T_{i}^{\text{Alone}} \) be estimated in shared mode?

- \( T_{i}^{\text{Excess}} \) is the number of extra cycles it takes application \( i \) to execute due to interference.

- \( T_{i}^{\text{Alone}} = T_{i}^{\text{Shared}} - T_{i}^{\text{Excess}} \)
Tracking Inter-Core Interference

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

- Shadow Row Address Register (SRAR) Core 1: Row B
- Shadow Row Address Register (SRAR) Core 0: Row A

Interference induced row conflict

Interference per core bit vector

Queue of requests to bank 2

Winograd - 202
Tracking Inter-Core Interference

Interference per core

Excess Cycles Counters per core

Core 0
Core 1
Core 2
Core 3

Shared Cache

Memory Controller

Bank 0
Bank 1
Bank 2
Bank 7

Cycle Count

FST hardware

Core #

T

i

Excess

T

i

Shared

Alone

T

i

= T

i

- T

i

Excess
Fairness via Source Throttling (FST)

FST

Runtime Unfairness Evaluation

Unfairness Estimate

App-slowest

App-interfering

Dynamic Request Throttling

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

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

204
To identify App-interfering, for each core $i$

- FST separately tracks interference caused by each core $j$ ($j \neq i$)

**Pairwise interference bit matrix**

<table>
<thead>
<tr>
<th>Interfering core</th>
<th>Interfered with core</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core # 0</td>
<td>0 - 0 - 0 - 0</td>
</tr>
<tr>
<td>Core # 1</td>
<td>0 - 0 - 1 - 0</td>
</tr>
<tr>
<td>Core # 2</td>
<td>0 - 1 - 0 - 0</td>
</tr>
<tr>
<td>Core # 3</td>
<td>0 - 0 - 0 - 0</td>
</tr>
</tbody>
</table>

**Pairwise excess cycles count per core**

<table>
<thead>
<tr>
<th></th>
<th>Cnt 0,1</th>
<th>Cnt 0,2</th>
<th>Cnt 0,3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cnt 1,0</td>
<td>-</td>
<td>-</td>
<td>Cnt 1,3</td>
</tr>
<tr>
<td>Cnt 2,0</td>
<td>Cnt 2,1</td>
<td>-</td>
<td>Cnt 2,3</td>
</tr>
<tr>
<td>Cnt 3,0</td>
<td>Cnt 3,1</td>
<td>Cnt 3,2</td>
<td>-</td>
</tr>
</tbody>
</table>

Core 2 interfered with core 1

Row with largest count determines App-interfering

App-slowest = 2
Fairness via Source Throttling (FST)

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

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

FST

Runtime Unfairness Evaluation

Unfairness Estimate

App-slowest

App-interfering

Dynamic Request Throttling
Dynamic Request Throttling

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

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

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

<table>
<thead>
<tr>
<th>Throttling level</th>
<th>MSHR quota</th>
<th>Request Injection Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>100%</td>
<td>128</td>
<td>Every cycle</td>
</tr>
<tr>
<td>50%</td>
<td>64</td>
<td>Every other cycle</td>
</tr>
<tr>
<td>25%</td>
<td>32</td>
<td>Once every 4 cycles</td>
</tr>
<tr>
<td>10%</td>
<td>12</td>
<td>Once every 10 cycles</td>
</tr>
<tr>
<td>5%</td>
<td>6</td>
<td>Once every 20 cycles</td>
</tr>
<tr>
<td>4%</td>
<td>5</td>
<td>Once every 25 cycles</td>
</tr>
<tr>
<td>3%</td>
<td>3</td>
<td>Once every 30 cycles</td>
</tr>
<tr>
<td>2%</td>
<td>2</td>
<td>Once every 50 cycles</td>
</tr>
</tbody>
</table>

Total # of MSHRs: 128
FST at Work

FST

Runtime Unfairness Evaluation

Unfairness Estimate
App-slowest
App-interfering

Core 0
Core 1
Core 2
Core 3

System software fairness goal: 1.4

Dynamic Request Throttling

Throttle down
Throttle up

Interval i
Interval i + 1
Interval i + 2

Throttling Levels

Interval i  50%  100%  10%  100%
Interval i + 1  25%  100%  25%  100%
Interval i + 2  25%  50%  50%  100%
System Software Support

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

- Support for thread priorities
  - Weighted Slowdown(i) = Estimated Slowdown(i) \times Weight(i)
FST Hardware Cost

- Total storage cost required for 4 cores is ~12KB
- FST does not require any structures or logic that are on the processor’s critical path
FST Evaluation Methodology

- x86 cycle accurate simulator
- Baseline processor configuration
  - Per-core
    - 4-wide issue, out-of-order, 256 entry ROB
  - Shared (4-core system)
    - 128 MSHRs
    - 2 MB, 16-way L2 cache
  - Main Memory
    - DDR3 1333 MHz
    - Latency of 15ns per command (tRP, tRCD, CL)
    - 8B wide core to memory bus
FST: System Unfairness Results

System Unfairness

- No Fairness
- Fair Cache Capacity (VPC)
- Parallelism-Aware Batch Scheduling + VPC
- Fairness via Source Throttling (FST)

44.4%
36%
FST: System Performance Results

- Fair Cache Capacity (VPC)
- Parallelism-Aware Batch Scheduling + VPC
- Fairness via Source Throttling (FST)

System Perf. Normalized to No Fairness

- 2.0
- 1.6
- 1.2
- 0.8
- 0.4
- 0.0

14%
25.6%
Source Throttling Results: Takeaways

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

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

- **Combined approaches** are even more powerful
  - Source throttling and resource-based interference control
Designing QoS-Aware Memory Systems: Approaches

**Smart resources:** Design each shared resource to have a configurable interference control/reduction mechanism

- **QoS-aware memory controllers** [Mutlu+ MICRO’07] [Moscibroda+, Usenix Security’07] [Mutlu+ ISCA’08, Top Picks’09] [Kim+ HPCA’10] [Kim+ MICRO’10, Top Picks’11] [Ebrahimi+ ISCA’11, MICRO’11] [Ausavarungnirun+, ISCA’12]
- **QoS-aware interconnects** [Das+ MICRO’09, ISCA’10, Top Picks ’11] [Grot+ MICRO’09, ISCA’11, Top Picks ’12]
- **QoS-aware caches**

**Dumb resources:** Keep each resource free-for-all, but reduce/control interference by injection control or data mapping

- **Source throttling to control access to memory system** [Ebrahimi+ ASPLOS’10, ISCA’11, TOCS’12] [Ebrahimi+ MICRO’09] [Nychis+ HotNets’10]
- **QoS-aware data mapping to memory controllers** [Muralidhara+ MICRO’11]
- **QoS-aware thread scheduling to cores**
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
Previous Approach

Goal:
Mitigate
Inter-Application Interference

Previous Approach:
Application-Aware Memory Request Scheduling

Our First Approach:
Application-Aware Memory Channel Partitioning

Our Second Approach:
Integrated Memory Partitioning and Scheduling
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

- Memory-non-intensive
- Memory-intensive

Threads in the system

Non-intensive cluster
- Prioritized

Intensive cluster
- Higher priority
- Fairness

Figure: Kim et al., MICRO 2010
Application-Aware Memory Request Scheduling

Advantages

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

Disadvantages

- Requires modifications to memory scheduling logic for ranking and prioritization
- Cannot completely eliminate interference by request reordering
Our Approach

Goal: Mitigate Inter-Application Interference

Previous Approach: Application-Aware Memory Request Scheduling

Our First Approach: Application-Aware Memory Channel Partitioning

Our Second Approach: Integrated Memory Partitioning and Scheduling
Observation: Modern Systems Have Multiple Channels

A new degree of freedom
Mapping data across multiple channels
Data Mapping in Current Systems

Causes interference between applications’ requests
Partitioning Channels Between Applications

Eliminates interference between applications’ requests
Overview: Memory Channel Partitioning (MCP)

- **Goal**
  - Eliminate harmful interference between applications

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

- **Key Principles**
  - Separate *low and high memory-intensity applications*
  - Separate *low and high row-buffer locality applications*
Key Insight 1: Separate by Memory Intensity

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

Map data of low and high memory-intensity applications to different channels.
Key Insight 2: Separate by Row-Buffer Locality

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

Map data of low and high row-buffer locality applications to different channels.
<table>
<thead>
<tr>
<th></th>
<th><strong>Memory Channel Partitioning (MCP) Mechanism</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td><strong>Profile</strong> applications</td>
</tr>
<tr>
<td>2.</td>
<td><strong>Classify</strong> applications into groups</td>
</tr>
<tr>
<td>3.</td>
<td><strong>Partition channels</strong> between application groups</td>
</tr>
<tr>
<td>4.</td>
<td><strong>Assign a preferred channel</strong> to each application</td>
</tr>
<tr>
<td>5.</td>
<td><strong>Allocate application pages</strong> to preferred channel</td>
</tr>
</tbody>
</table>

**Hardware**

**System**

**Software**
1. Profile Applications

- Hardware counters collect application memory access characteristics

- Memory access characteristics
  - Memory intensity:
    - Last level cache Misses Per Kilo Instruction (MPKI)
  - Row-buffer locality:
    - Row-buffer Hit Rate (RBH) - percentage of accesses that hit in the row buffer
2. Classify Applications

Test MPKI

Low

Low Intensity

High

High Intensity

Test RBH

Low

High Intensity Low Row-Buffer Locality

High

High Intensity Low Row-Buffer Locality
3. Partition Channels Among Groups: Step 1

Assign number of channels proportional to number of applications in group

Low Intensity

High Intensity
Low Row-Buffer Locality

High Intensity
High Row-Buffer Locality

Channel 1
Channel 2
Channel 3
...
Channel N-1
Channel N
3. Partition Channels Among Groups: Step 2

Assign number of channels proportional to bandwidth demand of group.
4. Assign Preferred Channel to Application

- Assign each application a preferred channel from its group’s allocated channels
- Distribute applications to channels such that group’s bandwidth demand is balanced across its channels
5. Allocate Page to Preferred Channel

- Enforce channel preferences computed in the previous step

- On a page fault, the operating system
  - allocates page to preferred channel if free page available in preferred channel
  - if free page not available, replacement policy tries to allocate page to preferred channel
  - if it fails, allocate page to another channel
Interval Based Operation

1. Profile applications
2. Classify applications into groups
3. Partition channels between groups
4. Assign preferred channel to applications
5. Enforce channel preferences
Integrating Partitioning and Scheduling

Goal:
Mitigate Inter-Application Interference

Previous Approach: Application-Aware Memory Request Scheduling

Our First Approach: Application-Aware Memory Channel Partitioning

Our Second Approach: Integrated Memory Partitioning and Scheduling
Observations

- Applications with very low memory-intensity rarely access memory
  → Dedicating channels to them results in precious memory bandwidth waste

- They have the most potential to keep their cores busy
  → We would really like to prioritize them

- They interfere minimally with other applications
  → Prioritizing them does not hurt others
Integrated Memory Partitioning and Scheduling (IMPS)

- Always prioritize very low memory-intensity applications in the memory scheduler.

- Use memory channel partitioning to mitigate interference between other applications.
Hardware Cost

- **Memory Channel Partitioning (MCP)**
  - Only profiling counters in hardware
  - No modifications to memory scheduling logic
  - 1.5 KB storage cost for a 24-core, 4-channel system

- **Integrated Memory Partitioning and Scheduling (IMPS)**
  - A single bit per request
  - Scheduler prioritizes based on this single bit
Methodology

- Simulation Model
  - 24 cores, 4 channels, 4 banks/channel
  - Core Model
    - Out-of-order, 128-entry instruction window
    - 512 KB L2 cache/core
  - Memory Model – DDR2

- Workloads
  - 240 SPEC CPU 2006 multiprogrammed workloads (categorized based on memory intensity)

- Metrics
  - System Performance $Weighted\; Speedup = \sum_i \frac{IPC_i^{shared}}{IPC_i^{alone}}$
Previous Work on Memory Scheduling

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

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

- **TCM** [Kim et al., MICRO 2010]
  - Always prioritizes low memory-intensity applications
  - Shuffles request priorities of high memory-intensity applications
Comparison to Previous Scheduling Policies

Averaged over 240 workloads

Normalized System Performance

FRFCFS
ATLAS
TCM
MCP
IMPS

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

Significant performance improvement over baseline FRFCFS
Interaction with Memory Scheduling

IMPS improves performance regardless of scheduling policy
Highest improvement over FRFCF as IMPS designed for FRFCFS
MCP Summary

- Uncontrolled inter-application interference in main memory degrades system performance

- **Application-aware memory channel partitioning (MCP)**
  - Separates the data of badly-interfering applications to different channels, eliminating interference

- **Integrated memory partitioning and scheduling (IMPS)**
  - Prioritizes very low memory-intensity applications in scheduler
  - Handles other applications’ interference by partitioning

- MCP/IMPS provide better performance than application-aware memory request scheduling at lower hardware cost
Summary: Memory QoS Approaches and Techniques

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

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

- **Combined approaches and techniques are the most powerful**
  - Integrated Memory Channel Partitioning and Scheduling [MICRO’11]
Handling Interference in Parallel Applications

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

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

- Hardware/software cooperative limiter thread estimation:
  - Thread executing the most contended critical section
  - Thread that is falling behind the most in a parallel for loop
Designing QoS-Aware Memory Systems: Approaches

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

- **Dumb resources:** Keep each resource free-for-all, but reduce/control interference by injection control or data mapping
  - Source throttling to control access to memory system [Ebrahimi+ ASPLOS’10, ISCA’11, TOCS’12, Ebrahimi+ MICRO’09, Nychis+ HotNets’10]
  - QoS-aware data mapping to memory controllers [Muralidhara+ MICRO’11]
  - QoS-aware thread scheduling to cores [Das+ HPCA’13]
Conclusions: Topic 3

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

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

- QoS-unaware memory: uncontrollable and unpredictable

- Providing QoS awareness improves performance, predictability, fairness, and utilization of the memory system
Scalable Many-Core Memory Systems

Topic 3: Memory Interference and QoS-Aware Memory Systems

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

HiPEAC ACACES Summer School 2013
July 15-19, 2013

Carnegie Mellon

SAFARI
Additional Material
Two Works


Application-to-Core Mapping Policies to Reduce Memory System Interference

Reetuparna Das*  Rachata Ausavarungnirun$  Onur Mutlu$
Akhilesh Kumar§  Mani Azimi§

*University of Michigan  $Carnegie Mellon University  §Intel
Multi-Core to Many-Core

Multi-Core

Many-Core
Many-Core On-Chip Communication

Applications

- Light
- Heavy

Memory Controller

Shared Cache Bank

$\$
Task Scheduling

- **Traditional**
  
  When to schedule a task? – Temporal

- **Many-Core**
  
  When to schedule a task? – Temporal
  + Where to schedule a task? – **Spatial**

- **Spatial scheduling impacts performance of memory hierarchy**
  - Latency and interference in interconnect, memory, caches
Problem: Spatial Task Scheduling

Applications

Cores

How to map applications to cores?
Challenges in Spatial Task Scheduling

How to reduce destructive interference between applications?

How to reduce communication distance?

How to prioritize applications to improve throughput?
Application-to-Core Mapping

- **Balancing**
  - Improve Bandwidth Utilization

- **Clustering**
  - Improve Locality
  - Reduce Interference

- **Isolation**
  - Reduce Interference

- **Radial Mapping**
  - Improve Bandwidth Utilization
Step 1 — Clustering

Inefficient data mapping to memory and caches
Step 1 — Clustering

Cluster 0

Cluster 1

Cluster 2

Cluster 3

Improved Locality

Reduced Interference
Step 1 — Clustering

- Clustering memory accesses
  - Locality aware page replacement policy (cluster-CLOCK)
    - When allocating free page, give preference to pages belonging to the cluster’s memory controllers (MCs)
    - Look ahead “N” pages beyond the default replacement candidate to find page belonging to cluster’s MC

- Clustering cache accesses
  - Private caches automatically enforce clustering
  - Shared caches can use Dynamic Spill Receive* mechanism

*Qureshi et al, HPCA 2009
Step 2 — Balancing

Applications

Heavy

Light

Cores

Too much load in clusters with heavy applications
Step 2 — Balancing

Applications

- Heavy
- Light

Cores

Better bandwidth utilization

Is this the best we can do? Let’s take a look at application characteristics
Application Types

YOUR THESIS COMMITTEE

Also known as: an impossibly difficult group to get together in one room but who nevertheless hold your future in their hands depending on their ability to reach a civilized consensus.

Your Professor
Simultaneously your biggest ally and your worst enemy. Will be the first to suggest you do more work.

The Guru
Only here for the free cookies. Don’t forget to bring cookies.

Adversary
Has bitter rivalry with your Professor and will argue the exact opposite view. Work this to your advantage.

The Strawman/woman
Nice guy. No opinions.

The Assistant Professor
Still doesn’t believe just a few months ago they were on the other side just like you. Pretends to be an adult.

NONE OF THEM WILL ACTUALLY READ YOUR ENTIRE THESIS.
Application Types

**Sensitive**
- High Miss Rate
- Low MLP

**Medium**
- Med Miss Rate
- High MLP

**Heavy**
- High Miss Rate
- High MLP

**Light**
- Low Miss Rate

Identify and *isolate sensitive applications* while ensuring load balance

Thesis Committee
- Advisor: Sensitive
- Guru: There for cookies
- Adversary: Bitter rival
- Nice Guy: No opinions
- Asst. Professor

(c) PHD Comics 267
Step 3 — Isolation

**Applications**
- Sensitive
- Light
- Medium
- Heavy

**Cores**

- Isolate *sensitive* applications to a cluster
- Balance load for remaining applications across clusters
Step 3 — Isolation

- How to estimate sensitivity?
  - High Miss — high misses per kilo instruction (MPKI)
  - Low MLP — high relative stall cycles per miss (STPM)
  - Sensitive if MPKI > Threshold and relative STPM is high

- Whether to or not to allocate cluster to sensitive applications?

- How to map sensitive applications to their own cluster?
  - Knap-sack algorithm
Step 4 — Radial Mapping

Map applications that benefit most from being close to memory controllers close to these resources.
Step 4 — Radial Mapping

- What applications benefit most from being close to the memory controller?
  - High memory bandwidth demand
  - Also affected by network performance
  - Metric => Stall time per thousand instructions
Putting It All Together

Inter-Cluster Mapping

Intra-Cluster Mapping

Clustering
Balancing
Isolation
Radial Mapping

Improve Locality
Reduce Interference
Improve Shared Resource Utilization
Evaluation Methodology

- **60-core system**
  - x86 processor model based on Intel Pentium M
  - 2 GHz processor, 128-entry instruction window
  - 32KB private L1 and 256KB per core private L2 caches
  - 4GB DRAM, 160 cycle access latency, 4 on-chip DRAM controllers
  - CLOCK page replacement algorithm

- **Detailed Network-on-Chip model**
  - 2-stage routers (with speculation and look ahead routing)
  - Wormhole switching (4 flit data packets)
  - Virtual channel flow control (4 VCs, 4 flit buffer depth)
  - 8x8 Mesh (128 bit bi-directional channels)
Configurations

- Evaluated configurations
  - BASE—Random core mapping
  - BASE+CLS—Baseline with clustering
  - A2C

- Benchmarks
  - Scientific, server, desktop benchmarks (35 applications)
  - 128 multi-programmed workloads
  - 4 categories based on aggregate workload MPKI
    - MPKI500, MPKI1000, MPKI1500, MPKI2000
System Performance

The system performance improves by 17%.
Network Power

Average network power consumption reduces by 52%
Summary of Other Results

- A2C can reduce page fault rate
Summary of Other Results

- A2C can reduce page faults
- **Dynamic A2C also improves system performance**
  - Continuous “Profiling” + “Enforcement” intervals
  - Retains clustering benefits
  - Migration overheads are minimal
- A2C complements application-aware packet prioritization* in NoCs
- A2C is effective for a variety of system parameters
  - Number of and placement of memory controllers
  - Size and organization of last level cache

*Das et al, MICRO 2009
**Conclusion**

- **Problem:** Spatial scheduling for Many-Core processors
  - Develop fundamental insights for core mapping policies
- **Solution:** Application-to-Core (A2C) mapping policies

- A2C improves system performance, system fairness and network power significantly
Application-to-Core Mapping Policies to Reduce Memory System Interference

Reetuparna Das* Rachata Ausavarungnirun$ Onur Mutlu$
Akhilesh Kumar§ Mani Azimi§

*University of Michigan §Carnegie Mellon University $Intel
Background

Shared Memory Resources

On-chip

Off-chip

Chip Boundary
Memory requests from different cores interfere in shared memory resources

Multi-programmed workloads

- System Performance and Fairness

A single multi-threaded application?
Memory System Interference in A Single Multi-Threaded Application

- *Inter-dependent threads* from the same application slow each other down

- Most importantly the *critical path* of execution can be significantly slowed down

- Problem and goal are very different from interference between independent applications
  - Interdependence between threads
  - Goal: *Reduce execution time* of a single application
  - *No notion of fairness* among the threads of the same application
Potential in A Single Multi-Threaded Application

If all main-memory related interference is ideally eliminated, execution time is reduced by 45% on average.
Outline

- Problem Statement
- Parallel Application Memory Scheduling
- Evaluation
- Conclusion
Outline

- Problem Statement
- Parallel Application Memory Scheduling
- Evaluation
- Conclusion
Parallel Application Memory Scheduler

- Identify the set of threads likely to be on the critical path as limiter threads
  - Prioritize requests from limiter threads

- Among limiter threads:
  - Prioritize requests from latency-sensitive threads (those with lower MPKI)

- Among non-limiter threads:
  - Shuffle priorities of non-limiter threads to reduce inter-thread memory interference
  - Prioritize requests from threads falling behind others in a parallel for-loop
Parallel Application Memory Scheduler

- Identify the set of threads likely to be on the critical path as limiter threads
  - Prioritize requests from limiter threads

- Among limiter threads:
  - Prioritize requests from latency-sensitive threads (those with lower MPKI)

- Among non-limiter threads:
  - Shuffle priorities of non-limiter threads to reduce inter-thread memory interference
  - Prioritize requests from threads falling behind others in a parallel for-loop
Contended critical sections are often on the critical path of execution.

Extend runtime system to identify thread executing the most contended critical section as the limiter thread:
- Track total amount of time all threads wait on each lock in a given interval.
- Identify the lock with largest waiting time as the most contended.
- Thread holding the most contended lock is a limiter and this information is exposed to the memory controller.
Prioritizing Requests from Limiter Threads

Limiter Thread Identification

Most Contended Critical Section: 🤔

Saved Cycles

Thread A
Thread B
Thread C
Thread D

Barrier
Time

Critical Section 1
Critical Section 2
Critical Path
Barrier
Waiting for Sync or Lock

Non-Critical Section
Parallel Application Memory Scheduler

- Identify the set of threads likely to be on the critical path as limiter threads
  - Prioritize requests from limiter threads

- Among limiter threads:
  - Prioritize requests from latency-sensitive threads (those with lower MPKI)

- Among non-limiter threads:
  - Shuffle priorities of non-limiter threads to reduce inter-thread memory interference
  - Prioritize requests from threads falling behind others in a parallel for-loop
Time-based classification of threads as latency- vs. BW-sensitive

Thread Cluster Memory Scheduling (TCM) [Kim et. al., MICRO’10]
Terminology

- A code-segment is defined as:
  - A program region between two consecutive synchronization operations
  - Identified with a 2-tuple: \(<\text{beginning IP, lock address}>\)

- Important for classifying threads as latency- vs. bandwidth-sensitive
  - Time-based vs. code-segment based classification
Code-segment based classification of threads as latency- vs. BW-sensitive
Parallel Application Memory Scheduler

- Identify the set of threads **likely to be on the critical path as limiter threads**
  - Prioritize requests from limiter threads

- Among limiter threads:
  - Prioritize requests from latency-sensitive threads (those with lower MPKI)

- Among non-limiter threads:
  - **Shuffle priorities of non-limiter threads to reduce inter-thread memory interference**
  - Prioritize requests from threads falling behind others in a *parallel for-loop*
Shuffling Priorities of Non-Limiter Threads

Goal:
- Reduce inter-thread interference among a set of threads with the same importance in terms of our estimation of the critical path
- Prevent any of these threads from becoming new bottlenecks

Basic Idea:
- Give each thread a chance to be high priority in the memory system and exploit intra-thread bank parallelism and row-buffer locality
- Every interval assign a set of random priorities to the threads and shuffle priorities at the end of the interval
Shuffling Priorities of Non-Limiter Threads

<table>
<thead>
<tr>
<th>Threads with similar memory behavior</th>
<th>Threads with different memory behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread A</td>
<td>Thread A</td>
</tr>
<tr>
<td>Thread B</td>
<td>Thread B</td>
</tr>
<tr>
<td>Thread C</td>
<td>Thread C</td>
</tr>
<tr>
<td>Thread D</td>
<td>Thread D</td>
</tr>
</tbody>
</table>

**Legend**
- Active
- Waiting

**Baseline (No shuffling)**

<table>
<thead>
<tr>
<th>Thread A</th>
<th>Thread B</th>
<th>Thread C</th>
<th>Thread D</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td>Time</td>
<td>Time</td>
<td>Time</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

**Shuffling**

**Policy 1**

<table>
<thead>
<tr>
<th>Thread A</th>
<th>Thread B</th>
<th>Thread C</th>
<th>Thread D</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td>Time</td>
<td>Time</td>
<td>Time</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

**Policy 2**

**Legend**
- Threads with similar memory behavior
- Threads with different memory behavior

PAMS dynamically monitors memory intensities and chooses appropriate shuffling policy for non-limiter threads at runtime.
Outline

- Problem Statement
- Parallel Application Memory Scheduling
- Evaluation
- Conclusion
Evaluation Methodology

- x86 cycle accurate simulator

- Baseline processor configuration
  - Per-core
    - 4-wide issue, out-of-order, 64 entry ROB
  - Shared (16-core system)
    - 128 MSHRs
    - 4MB, 16-way L2 cache
  - Main Memory
    - DDR3 1333 MHz
    - Latency of 15ns per command (tRP, tRCD, CL)
    - 8B wide core to memory bus
PAMS Evaluation

- Thread cluster memory scheduler [Kim+, MICRO'10]

```
| Thread cluster memory scheduler [Kim+, MICRO'10] |
```

```
Thread criticality predictors (TCP) [Bhattacherjee+, ISCA'09]
```

![Bar chart showing normalized execution time normalized to FR-FCFS for different threads: hist, mg, cg, is, bt, ft, and gmean. The chart highlights 13% and 7% improvements.]
Sensitivity to system parameters

<table>
<thead>
<tr>
<th></th>
<th>L2 Cache Size</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>4 MB</td>
<td>8 MB</td>
<td>16 MB</td>
</tr>
<tr>
<td>Δ FR-FCFS</td>
<td>Δ FR-FCFS</td>
<td>Δ FR-FCFS</td>
<td></td>
</tr>
<tr>
<td>-16.7%</td>
<td>-15.9%</td>
<td>-10.5%</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Number of Memory Channels</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1 Channel</td>
<td>2 Channels</td>
<td>4 Channels</td>
</tr>
<tr>
<td>Δ FR-FCFS</td>
<td>Δ FR-FCFS</td>
<td>Δ FR-FCFS</td>
<td></td>
</tr>
<tr>
<td>-16.7%</td>
<td>-11.6%</td>
<td>-10.4%</td>
<td></td>
</tr>
</tbody>
</table>
Conclusion

- Inter-thread main memory interference within a multi-threaded application increases execution time

- Parallel Application Memory Scheduling (PAMS) improves a single multi-threaded application’s performance by
  - Identifying a set of threads likely to be on the critical path and prioritizing requests from them
  - Periodically shuffling priorities of non-likely critical threads to reduce inter-thread interference among them

- PAMS significantly outperforms
  - Best previous memory scheduler designed for multi-programmed workloads
  - A memory scheduler that uses a state-of-the-art thread criticality predictor (TCP)
Related Works
Some Related Past Work

- That I could not cover...
- How to handle prefetch requests in a QoS-aware multi-core memory system?
  - Prefetch-aware shared resource management, ISCA’11. ISCA 2011 Talk
  - Prefetch-aware memory controllers, MICRO’08, IEEE-TC’11. Micro 2008 Talk
  - Coordinated control of multiple prefetchers, MICRO’09. Micro 2009 Talk
- How to design QoS mechanisms in the interconnect?
  - Slack-based packet scheduling, ISCA’10, IEEE Micro’11.
  - Efficient bandwidth guarantees, MICRO’09.
  - Application-aware request prioritization, MICRO’09.
Some Issues in Cache Design
DRAM-Aware LLC Writeback

- **Problem 1:** Writebacks to DRAM interfere with reads and cause additional performance penalty
  - Write-to-read turnaround time in DRAM bus
  - Write-recovery latency in DRAM bank
  - Change of row buffer → reduced row-buffer locality for read requests

- **Problem 2:** Writebacks that occur once in a while have low row buffer locality

- **Idea:** When evicting a dirty cache block to a row, proactively search the cache for other dirty blocks to the same row → evict them → write to DRAM in a batch
  - Improves row buffer locality
  - Reduces write-to-read switching penalties on DRAM bus
  - Improves performance on both single-core and multi-core systems
More Information

- Chang Joo Lee, Veynu Narasiman, Eiman Ebrahimi, Onur Mutlu, and Yale N. Patt,
  "DRAM-Aware Last-Level Cache Writeback: Reducing Write-Caused Interference in Memory Systems"

DRAM-Aware Last-Level Cache Writeback:
Reducing Write-Caused Interference in Memory Systems

Abstract

Read and write requests from a processor contend for the main memory data bus. System performance depends heavily on when read requests are serviced since they are required for an application's forward progress whereas writes do not need to be performed immediately. However, writes eventually have to be written to memory because the storage required to buffer them on-chip is limited.

In modern high bandwidth DDR (Double Data Rate)-based memory systems write requests significantly interfere with the servicing of read requests by delaying the more critical read requests and by causing the memory bus to become idle when switching between the servicing of a write and read request. This interference significantly degrades overall system performance. We call this phenomenon write-caused interference. To reduce write-caused interference, this paper proposes a new last-level cache writeback policy, called DRAM-aware writeback. The key idea of the proposed technique is to aggressively send out writeback requests that are expected to hit in DRAM row buffers before they would normally be evicted by the last-level cache replacement policy and have the DRAM controller service as many writes as possible together. Doing so not only reduces the amount of time to service writes by improving their row buffer locality but also reduces the idle bus cycles wasted due to switching between the servicing of a write and a read request.

DRAM-aware writeback improves system performance by 7.1% and 12.8% on single and 4-core systems respectively. The performance benefits of the mechanism increases in systems with prefetching since such systems have higher contention between reads and writes in the DRAM system.
DRAM-Aware Cache Design: An Example of Resource Coordination
DRAM-Aware Cache Design

- Coordination of cache policies with memory controllers

  Chang Joo Lee, Veynu Narasiman, Eiman Ebrahimi, Onur Mutlu, and Yale N. Patt,
  "DRAM-Aware Last-Level Cache Writeback: Reducing Write-Caused Interference in Memory Systems"

  Chang Joo Lee, Eiman Ebrahimi, Veynu Narasiman, Onur Mutlu, and Yale N. Patt,
  "DRAM-Aware Last-Level Cache Replacement"
Write-Caused Interference: Read-Write Switching

- Row-hit read-to-read (write-to-write) to any bank: back-to-back data transfer

  Command: Read A  Read B
  Data bus: Data A  Data B

- Read-write switching penalty for requests to any bank

  Command: Read A  Write C  Read B
  Data bus: Data A  Data C  Data B

Frequent read-write switching incurs many idle cycles
Write-Caused Interference: Write-to-Row-Conflict

- Row-conflict after read (in the same bank)

Command:

Data bus:

- Read A  Precharge  Activate B  Read or write B

Row-conflict

Idle (~120 processor cycles)

- Row-conflict after write (in the same bank)

Command:

Data bus:

- Write A

Precharge  Activate B  Read or write B

Row-conflict

Idle (~225 processor cycles)

No precharge

Row-conflict after a write causes more idle cycles
Write-Caused Interference

• Read-Write Switching
  – Frequent read-write switching incurs many idle cycles

• Write-to-Row-Conflict
  – A row-conflict after a write causes more idle cycles

Generating many row-hit writes rather than row-conflict writes is preferred
LRU vs. Interference-Aware Replacement

All requests are to the same cache set

Conventional LRU:
Write B (row-hit), Write C (row-conflict)

Read A
Write B
Write C

Reading A
Writing B
Writing C

Row B
Row Buffer in Bank 0

Servicing

Read buffer
Write buffer

Clean A
Clean
Dirty B
Dirty C

Set X

Less recently used

Dirty C

Clean
Clean

Last-level cache

Controller

DRAM

DRAM
LRU vs. Interference-Aware Replacement

All requests are to the same cache set

- **Conventional LRU:**
  - Write B (row-hit), Write C (row-conflict)

- **Interference-aware:**
  - Write B (row-hit), Write B (row-hit)

**Reduced idle cycles**

A simple policy can reduce write service time
Performance of DRAM-Aware Replacement

1-core: 11.4%

4-core: 12.3%
Outline

• Problem

• Solutions
  – Prefetch-Aware DRAM Controller
  – BLP-Aware Request Issue Policies
  – DRAM-Aware Cache Replacement
  – DRAM-Aware Writeback

• Combination of Solutions
• Related Work
• Conclusion
DRAM-Aware Writeback

• Write-caused interference-aware replacement is not enough
  – Row-hit writebacks are sent only when a replacement occurs
    • Lose opportunities to service more writes quickly

• To minimize write-caused interference, proactively clean row-hit dirty lines
  → Reads are serviced without write-caused interference for a longer period
DRAM-Aware Writeback

1. When a dirty line is evicted for the last-level cache, store its address

2. Using the stored address, search all possible sets for row-hit dirty lines and clean them whenever the cache bank is idle

- Many row-hit writes (up to the row size) are serviced quickly
  - Reads can be serviced for a longer time without being interfered with by writes
Performance of DRAM-Aware Writeback

Performance normalized to baseline

1-core  7.1%

4-core  12.8%