Computer Architecture
Lecture 12: Memory Interference and Quality of Service

Prof. Onur Mutlu
ETH Zürich
Fall 2017
1 November 2017
Summary of Last Week’s Lectures

- Control Dependence Handling
  - Problem
  - Six solutions

- Branch Prediction

- Trace Caches

- Other Methods of Control Dependence Handling
  - Fine-Grained Multithreading
  - Predicated Execution
  - Multi-path Execution
Agenda for Today

- Shared vs. private resources in multi-core systems
- Memory interference and the QoS problem
- Memory scheduling
- Other approaches to mitigate and control memory interference
Quick Summary Papers

- "Parallelism-Aware Batch Scheduling: Enhancing both Performance and Fairness of Shared DRAM Systems"

- "The Blacklisting Memory Scheduler: Achieving High Performance and Fairness at Low Cost"

- "Staged Memory Scheduling: Achieving High Performance and Scalability in Heterogeneous Systems"

- "Parallel Application Memory Scheduling"

- "Reducing Memory Interference in Multicore Systems via Application-Aware Memory Channel Partitioning"
Shared Resource Design for Multi-Core Systems
Memory System: A **Shared Resource** View
Resource Sharing Concept

- **Idea:** Instead of dedicating a hardware resource to a hardware context, allow multiple contexts to use it
  - Example resources: functional units, pipeline, caches, buses, memory

- **Why?**
  - Resource sharing *improves utilization/efficiency → throughput*
    - When a resource is left idle by one thread, another thread can use it; no need to replicate shared data
  - *Reduces communication latency*
    - For example, shared data kept in the same cache in SMT processors
  - *Compatible with the shared memory model*
Resource Sharing Disadvantages

- Resource sharing results in contention for resources
  - When the resource is not idle, another thread cannot use it
  - If space is occupied by one thread, another thread needs to re-occupy it

- Sometimes reduces each or some thread’s performance
  - Thread performance can be worse than when it is run alone

- Eliminates performance isolation → inconsistent performance across runs
  - Thread performance depends on co-executing threads

- Uncontrolled (free-for-all) sharing degrades QoS
  - Causes unfairness, starvation

Need to efficiently and fairly utilize shared resources
Example: Problem with Shared Caches

Example: Problem with Shared Caches

Example: Problem with Shared Caches

Processor Core 1 \( \rightarrow t_1 \) Processor Core 2

L1 $

L2 $

\ldots$

\[ \text{t2's throughput is significantly reduced due to unfair cache sharing.} \]

Need for QoS and Shared Resource Mgmt.

- Why is unpredictable performance (or lack of QoS) bad?

- Makes programmer’s life difficult
  - An optimized program can get low performance (and performance varies widely depending on co-runners)

- Causes discomfort to user
  - An important program can starve
  - Examples from shared software resources

- Makes system management difficult
  - How do we enforce a Service Level Agreement when hardware resources are sharing is uncontrollable?
Resource Sharing vs. Partitioning

- Sharing improves throughput
  - Better utilization of space

- Partitioning provides performance isolation (predictable performance)
  - Dedicated space

Can we get the benefits of both?

Idea: Design shared resources such that they are efficiently utilized, controllable and partitionable
  - No wasted resource + QoS mechanisms for threads
Shared Hardware Resources

- Memory subsystem (in both Multi-threaded and Multi-core)
  - Non-private caches
  - Interconnects
  - Memory controllers, buses, banks

- I/O subsystem (in both Multi-threaded and Multi-core)
  - I/O, DMA controllers
  - Ethernet controllers

- Processor (in Multi-threaded)
  - Pipeline resources
  - L1 caches
Memory System is the Major Shared Resource

- Threads' requests interfere.
- Shared Memory Resources
- On-chip
- Off-chip
- Chip Boundary
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

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?

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

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)
- Unable to enforce priorities or SLAs
- Low system performance

Uncontrollable, unpredictable system
Greater Problem with More Cores

- Vulnerable to denial of service (DoS)
- Unable to enforce priorities or SLAs
- Low system performance

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

Cores connected via packet-switched routers on chip

~5000X latency increase

More on Memory Performance Attacks


Memory Performance Attacks: Denial of Memory Service in Multi-Core Systems

Thomas Moscibroda  Onur Mutlu
Microsoft Research
{moscitho, onur}@microsoft.com
More on Interconnect Based Starvation

- Boris Grot, Stephen W. Keckler, and Onur Mutlu, "Preemptive Virtual Clock: A Flexible, Efficient, and Cost-effective QOS Scheme for Networks-on-Chip"

Preemptive Virtual Clock: A Flexible, Efficient, and Cost-effective QOS Scheme for Networks-on-Chip

Boris Grot  Stephen W. Keckler  Onur Mutlu†
Department of Computer Sciences  The University of Texas at Austin  {bgrot, skeckler@cs.utexas.edu}
†Computer Architecture Laboratory (CALCM)  Carnegie Mellon University  onur@cmu.edu
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
  - QoS-aware interconnects
  - 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
  - QoS-aware data mapping to memory controllers
  - QoS-aware thread scheduling to cores
Fundamental Interference Control Techniques

- **Goal:** to reduce/control inter-thread memory interference

1. Prioritization or request scheduling

2. Data mapping to banks/channels/ranks

3. Core/source throttling

4. Application/thread scheduling
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
  - 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: Memory Slowdown Model** [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

- **ASM: Application Slowdown Model** [Subramanian+ MICRO’15]
  - Idea: Extend MISE to take into account cache+memory interference
  - Takeaway: Cache access rate of an application can be estimated accurately and is a good proxy for application performance
QoS-Aware Memory Scheduling: Evolution

- **BLISS: Blacklisting Memory Scheduler** [Subramanian+ ICCD’14]
  - Idea: Deprioritize (i.e., blacklist) a thread that has consecutively serviced a large number of requests
  - Takeaway: Blacklisting greatly reduces interference enables the scheduler to be simple without requiring full thread ranking

- **DASH: Deadline-Aware Memory Scheduler** [Usui+ TACO’16]
  - Idea: Balance prioritization between CPUs, GPUs and Hardware Accelerators (HWA) by keeping HWA progress in check vs. deadlines such that HWAs do not hog performance and appropriately distinguishing between latency-sensitive vs. bandwidth-sensitive CPU workloads
  - Takeaway: Proper control of HWA progress and application-aware CPU prioritization leads to better system performance while meeting HWA deadlines
QoS-Aware Memory Scheduling: Evolution

- **Prefetch-aware shared resource management** [Ebrahimi+ ISCA’11] [Ebrahimi+ MICRO’09] [Ebrahimi+ HPCA’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 and write scheduling** [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; writes should not be ignored
QoS-Aware Memory Scheduling: Evolution

- **FIRM: Memory Scheduling for NVM** [Zhao+ MICRO’14]
  - Idea: Carefully handle write-read prioritization with coarse-grained batching and application-aware scheduling
  - Takeaway: Carefully controlling and prioritizing write requests improves performance and fairness; write requests are especially critical in NVMs
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)
The Problem: Unfairness

- Vulnerable to denial of service (DoS)
- Unable to enforce priorities or SLAs
- Low system performance

**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** = $ST_{\text{shared}}/ST_{\text{alone}}$
- 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 $\text{Slowdown} = ST_{\text{shared}}/ST_{\text{alone}}$ for threads with legal requests
  - Computes $\text{unfairness} = \text{MAX Slowdown} / \text{MIN Slowdown}$

- If unfairness $< \alpha$
  - Use DRAM throughput oriented scheduling policy

- If 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?

- T0: Row 0
- T1: Row 5
- T0: Row 0
- T1: Row 111
- T0: Row 0
- T0: Row 0

<table>
<thead>
<tr>
<th>T0 Slowdown</th>
<th>T1 Slowdown</th>
<th>Unfairness</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.04</td>
<td>1.08</td>
<td>1.08</td>
</tr>
</tbody>
</table>

α = 1.05
STFM Pros and Cons

■ Upsides:
  ● First algorithm for fair multi-core memory scheduling
  ● Provides a mechanism to estimate memory slowdown of a thread
  ● Good at providing fairness
  ● Being fair can improve performance

■ Downsides:
  ● Does not handle all types of interference
  ● (Somewhat) complex to implement
  ● Slowdown estimations can be incorrect
More on STFM

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

- Processors try to tolerate 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:

2 DRAM Requests

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

**Baseline Scheduler:**

- **A:**
  - Compute
  - Stall
  - Stall
  - Compute
  - Bank 0
  - Bank 1

- **B:**
  - Compute
  - Stall
  - Stall
  - Compute
  - Bank 1
  - Bank 0

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

**Parallelism-aware Scheduler:**

- 2 DRAM Requests
- 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
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
Thread Ranking

Key Idea:

SAFARI

SAVED CYCLES
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

<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>T1</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>T1</td>
<td>T3</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>T1</td>
<td>T3</td>
<td>T2</td>
<td>T0</td>
</tr>
<tr>
<td>5</td>
<td>T1</td>
<td>T3</td>
<td>T2</td>
<td>T0</td>
</tr>
<tr>
<td>6</td>
<td>T1</td>
<td>T3</td>
<td>T2</td>
<td>T0</td>
</tr>
<tr>
<td>7</td>
<td>T1</td>
<td>T3</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>T3</td>
<td>T3</td>
<td>T2</td>
<td>T0</td>
</tr>
<tr>
<td>2</td>
<td>T1</td>
<td>T3</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>T1</td>
<td>T3</td>
<td>T2</td>
<td>T0</td>
</tr>
<tr>
<td>5</td>
<td>T1</td>
<td>T3</td>
<td>T2</td>
<td>T0</td>
</tr>
<tr>
<td>6</td>
<td>T1</td>
<td>T3</td>
<td>T2</td>
<td>T0</td>
</tr>
<tr>
<td>7</td>
<td>T1</td>
<td>T3</td>
<td>T2</td>
<td>T0</td>
</tr>
</tbody>
</table>

Ranking: T0 > T1 > T2 > T3

Stall times

<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>T0</td>
<td>T1</td>
<td>T2</td>
<td>T3</td>
</tr>
<tr>
<td>3</td>
<td>T0</td>
<td>T1</td>
<td>T2</td>
<td>T3</td>
</tr>
<tr>
<td>4</td>
<td>T0</td>
<td>T1</td>
<td>T2</td>
<td>T3</td>
</tr>
<tr>
<td>5</td>
<td>T0</td>
<td>T1</td>
<td>T2</td>
<td>T3</td>
</tr>
<tr>
<td>6</td>
<td>T0</td>
<td>T1</td>
<td>T2</td>
<td>T3</td>
</tr>
<tr>
<td>7</td>
<td>T0</td>
<td>T1</td>
<td>T2</td>
<td>T3</td>
</tr>
</tbody>
</table>

AVG: 5 bank access latencies

AVG: 3.5 bank access latencies
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)

Normalized Hmean Speedup

FR-FCFS
FCFS
NFQ
STFM
PAR-BS

8.3%
6.1%
5.1%

4-core
8-core
16-core
PAR-BS Pros and Cons

- **Upsides:**
  - First scheduler to address bank parallelism destruction across multiple threads
  - Simple mechanism (vs. STFM)
  - Batching provides fairness
  - Ranking enables parallelism awareness

- **Downsides:**
  - Does not always prioritize the latency-sensitive applications
More on PAR-BS

- Onur Mutlu and Thomas Moscibroda,
  "Parallelism-Aware Batch Scheduling: Enhancing both Performance and Fairness of Shared DRAM Systems"
  [Summary] [Slides (ppt)]

Parallelism-Aware Batch Scheduling:
Enhancing both Performance and Fairness of Shared DRAM Systems

Onur Mutlu    Thomas Moscibroda
Microsoft Research
{onur,moscitho}@microsoft.com
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)
ATLAS: Summary

- Goal: To maximize system performance

- Main idea: Prioritize the thread that has attained the least service from the memory controllers (Adaptive per-Thread Least Attained Service Scheduling)
  - Rank threads based on attained service in the past time interval(s)
  - Enforce thread ranking in the memory scheduler during the current interval

- Why it works: Prioritizes “light” (memory non-intensive) threads that are more likely to keep their cores busy
System Throughput: 24-Core System

System throughput = \sum \text{Speedup}

ATLAS consistently provides higher system throughput than all previous scheduling algorithms
System Throughput: 4-MC System

# of cores increases ➔ ATLAS performance benefit increases
ATLAS Pros and Cons

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

- **Downsides:**
  - Lowest/medium ranked threads get delayed significantly → high unfairness
More on 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" Proceedings of the 16th International Symposium on High-Performance Computer Architecture (HPCA), Bangalore, India, January 2010. Slides (pptx)

ATLAS: A Scalable and High-Performance Scheduling Algorithm for Multiple Memory Controllers

Yoongu Kim  Dongsu Han  Onur Mutlu  Mor Harchol-Balter
Carnegie Mellon University
TCM: Thread Cluster Memory Scheduling

Yoongu Kim, Michael Papamichael, Onur Mutlu, and Mor Harchol-Balter,
"Thread Cluster Memory Scheduling: Exploiting Differences in Memory Access Behavior"
Previous Scheduling Algorithms are Biased

No previous memory scheduling algorithm provides both the best fairness and system throughput.
**Throughput vs. Fairness**

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

**Good for throughput**
- Less memory intensive
- Higher priority
- Starvation → unfairness

**Fairness biased approach**
- Take turns accessing memory

**Does not starve**
- Not prioritized → reduced throughput

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

For Throughput
Prioritize memory-non-intensive threads

For Fairness
- Unfairness caused by memory-intensive threads being prioritized over each other
  - Shuffle thread ranking
- Memory-intensive threads have different vulnerability to interference
  - Shuffle asymmetrically
Thread Cluster Memory Scheduling [Kim+ MICRO’10]

1. Group threads into two clusters
2. Prioritize non-intensive cluster
3. Different policies for each cluster
TCM Outline

1. Clustering
Clustering Threads

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

\[ T = \text{Total memory bandwidth usage} \]

\[ \alpha T \text{ divides clusters} \]

\[ \alpha < 10\% \text{ ClusterThreshold} \]
TCM Outline

1. Clustering

2. 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

Throughput

Fairness
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?

Bank-level parallelism
Vulnerability to interference

Row-buffer locality
Causes interference

Niceness

High
Low
**TCM: Quantum-Based Operation**

**Previous quantum**  
(~1M cycles)

**Current quantum**  
(~1M cycles)

**Shuffle interval**  
(~1K cycles)

**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><strong>&lt; 4kbits</strong></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

24 cores, 4 memory controllers, 96 workloads

Better fairness

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

Better system throughput

Adjusting ClusterThreshold

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
  - Caters to the needs for different types of threads (latency vs. bandwidth sensitive)
  - (Relatively) simple

- **Downsides:**
  - Scalability to large buffer sizes?
  - Robustness of clustering and shuffling algorithms?
  - Ranking is still too complex?
More on TCM

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

Lavanya Subramanian, Donghyuk Lee, Vivek Seshadri, Harsha Rastogi, and Onur Mutlu,
"The Blacklisting Memory Scheduler: Achieving High Performance and Fairness at Low Cost"
Proceedings of the 32nd IEEE International Conference on Computer Design (ICCD), Seoul, South Korea, October 2014. [Slides (pptx) (pdf)]
Tackling Inter-Application Interference: Application-aware Memory Scheduling

Full ranking increases critical path latency and area significantly to improve performance and fairness
Is it essential to give up simplicity to optimize for performance and/or fairness?

Our solution achieves all three goals.
Key Observation 1: Group Rather Than Rank

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

Benefit 2: Lower slowdowns than ranking
Key Observation 1: Group Rather Than Rank

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

How to classify applications into groups?
Key Observation 2

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

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

**Benefits:**
- **Lower complexity**
- **Finer grained grouping decisions** → **Lower unfairness**
Performance vs. Fairness vs. Simplicity

- FRFCFS
- FRFCFSCap
- PARBS
- ATLAS
- TCM
- Blacklisting

**Blacklisting is the closest scheduler to ideal**
Performance and Fairness

1. Blacklisting achieves the highest performance
2. Blacklisting balances performance and fairness
Blacklisting reduces complexity significantly
More on BLISS (I)

- Lavanya Subramanian, Donghyuk Lee, Vivek Seshadri, Harsha Rastogi, and Onur Mutlu,
"The Blacklisting Memory Scheduler: Achieving High Performance and Fairness at Low Cost"
Proceedings of the 32nd IEEE International Conference on Computer Design (ICCD), Seoul, South Korea, October 2014.
[Slides (pptx) (pdf)]

The Blacklisting Memory Scheduler:
Achieving High Performance and Fairness at Low Cost

Lavanya Subramanian, Donghyuk Lee, Vivek Seshadri, Harsha Rastogi, Onur Mutlu
Carnegie Mellon University
{lsubrama,donghyul,visesh,harshar,onor}@cmu.edu
More on BLISS: Longer Version

- Lavanya Subramanian, Donghyuk Lee, Vivek Seshadri, Harsha Rastogi, and Onur Mutlu,
  "BLISS: Balancing Performance, Fairness and Complexity in Memory Access Scheduling"
  [Source Code]
Staged Memory Scheduling

SMS: Executive Summary

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

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

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

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

Core 1  Core 2  Core 3  Core 4  GPU

Stage 1

Batch Formation

Stage 2

Monolithic Scheduler

Batch Scheduler

Stage 3

DRAM Command Scheduler

Bank 1  Bank 2  Bank 3  Bank 4

To DRAM
SMS: Staged Memory Scheduling

Stage 1

Batch Formation

Stage 2

Batch Scheduler

Stage 3

DRAM Command Scheduler

To DRAM

Bank 1
Bank 2
Bank 3
Bank 4
Putting Everything Together

Stage 1: Batch Formation

Stage 2: Batch Scheduler

Stage 3: DRAM Command Scheduler

Current Batch Scheduling Policy

RR
Complexity

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

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

*Based on a Verilog model using 180nm library*
Performance at Different GPU Weights

![Graph showing system performance at different GPU weights for different schedulers: ATLAS, TCM, FR-FCFS. The graph highlights the best previous scheduler with a red line, showing improved performance as GPU weight increases.](Image)
At every GPU weight, SMS outperforms the best previous scheduling algorithm for that weight.
More on SMS

- Rachata Ausavarungnirun, Kevin Chang, Lavanya Subramanian, Gabriel Loh, and Onur Mutlu,
  "Staged Memory Scheduling: Achieving High Performance and Scalability in Heterogeneous Systems"
  Proceedings of the 39th International Symposium on Computer Architecture (ISCA), Portland, OR, June 2012. Slides (pptx)
DASH Memory Scheduler
[TACO 2016]
Current SoC Architectures

- Heterogeneous agents: CPUs and HWAs
  - HWA : Hardware Accelerator
- Main memory is shared by CPUs and HWAs → Interference
  How to schedule memory requests from CPUs and HWAs to mitigate interference?
DASH Scheduler: Executive Summary

- **Problem**: Hardware accelerators (HWAs) and CPUs share the same memory subsystem and interfere with each other in main memory.
- **Goal**: Design a memory scheduler that improves CPU performance while meeting HWAs’ deadlines.
- **Challenge**: Different HWAs have different memory access characteristics and different deadlines, which current schedulers do not smoothly handle.
  - Memory-intensive and long-deadline HWAs significantly degrade CPU performance *when they become high priority* (due to slow progress).
  - Short-deadline HWAs sometimes miss their deadlines *despite high priority*.
- **Solution**: DASH Memory Scheduler
  - Prioritize HWAs over CPU anytime when the HWA is not making good progress.
  - Application-aware scheduling for CPUs and HWAs.
- **Key Results**:
  1) Improves CPU performance for a wide variety of workloads by 9.5%.
  2) Meets 100% deadline met ratio for HWAs.
- DASH source code freely available on our GitHub.
Goal of Our Scheduler (DASH)

• **Goal:** Design a memory scheduler that
  – Meets GPU/accelerators’ frame rates/deadlines *and*
  – Achieves high CPU performance

• **Basic Idea:**
  – *Different CPU applications and hardware accelerators have different memory requirements*
  – Track progress of different agents and prioritize accordingly
Key Observation: Distribute Priority for Accelerators

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

Distributing priority over time mitigates impact of accelerators on CPU cores’ requests
Key Observation: Not All Accelerators are Equal

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

Schedule short-deadline accelerators based on worst-case memory access time
Key Observation:
Not All CPU cores are Equal

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

Prioritize accelerators over memory-intensive cores to ensure accelerators do not become urgent
DASH Summary: Key Ideas and Results

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

Improves CPU performance by 7-21%
Meets (almost) 100% of deadlines for HWAs
DASH: Scheduling Policy

- DASH scheduling policy
  1. Short-deadline-period HWAs with high priority
  2. Long-deadline-period HWAs with high priority
  3. Memory non-intensive CPU applications
  4. Long-deadline-period HWAs with low priority
  5. Memory-intensive CPU applications
  6. Short-deadline-period HWAs with low priority
More on DASH

- Hiroyuki Usui, Lavanya Subramanian, Kevin Kai-Wei Chang, and Onur Mutlu,
  "DASH: Deadline-Aware High-Performance Memory Scheduler for Heterogeneous Systems with Hardware Accelerators"
  Presented at the 11th HiPEAC Conference, Prague, Czech Republic, January 2016.
  [Slides (pptx) (pdf)]
  [Source Code]

DASH: Deadline-Aware High-Performance Memory Scheduler for Heterogeneous Systems with Hardware Accelerators

HIROYUKI USUI, LAVANYA SUBRAMANIAN, KEVIN KAI-WEI CHANG, and ONUR MUTLU, Carnegie Mellon University
We did not cover the following slides in lecture. These are for your preparation for the next lecture.
Computer Architecture

Lecture 12: Memory Interference and Quality of Service

Prof. Onur Mutlu
ETH Zürich
Fall 2017
1 November 2017
Predictable Performance:
Strong Memory Service Guarantees
Goal: Predictable Performance in Complex Systems

- Heterogeneous agents: CPUs, GPUs, and HWAs
- Main memory interference between CPUs, GPUs, HWAs

How to allocate resources to heterogeneous agents to mitigate interference and provide predictable performance?
Strong Memory Service Guarantees

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

- **Approach:**
  - Develop techniques/models to accurately *estimate* the performance loss 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*

We will defer this for later...
More on MISE

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

MISE: Providing Performance Predictability and Improving Fairness in Shared Main Memory Systems

Lavanya Subramanian    Vivek Seshadri    Yoongu Kim    Ben Jaiyen    Onur Mutlu
Carnegie Mellon University
More on Application Slowdown Model

- Lavanya Subramanian, Vivek Seshadri, Arnab Ghosh, Samira Khan, and Onur Mutlu,
  "The Application Slowdown Model: Quantifying and Controlling the Impact of Inter-Application Interference at Shared Caches and Main Memory"
  Proceedings of the 48th International Symposium on Microarchitecture (MICRO), Waikiki, Hawaii, USA, December 2015.
  [Slides (pptx) (pdf)] [Lightning Session Slides (pptx) (pdf)] [Poster (pptx) (pdf)]
  [Source Code]

The Application Slowdown Model: Quantifying and Controlling the Impact of Inter-Application Interference at Shared Caches and Main Memory

Lavanya Subramanian*§  Vivek Seshadri*  Arnab Ghosh*†
Samira Khan*‡  Onur Mutlu*

*Carnegie Mellon University  §Intel Labs  †IIT Kanpur  ‡University of Virginia
Handling Memory Interference
In Multithreaded Applications

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

- Threads in a multi-threaded application can be inter-dependent
  - As opposed to threads from different applications

- Such threads can synchronize with each other
  - Locks, barriers, pipeline stages, condition variables, semaphores, ...

- Some threads can be on the critical path of execution due to synchronization; some threads are not

- Even within a thread, some “code segments” may be on the critical path of execution; some are not
Critical Sections

- Enforce mutually exclusive access to shared data
- Only one thread can be executing it at a time
- Contended critical sections make threads wait → threads causing serialization can be on the critical path

Each thread:
```
loop {
  Compute
  lock(A)
  Update shared data
  unlock(A)
}
```
Barriers

- Synchronization point
- Threads have to wait until all threads reach the barrier
- Last thread arriving at the barrier is on the critical path

Each thread:
```c
loop1 {
    Compute
}
```
```c
barrier
```
```c
loop2 {
    Compute
}
```
Stages of Pipelined Programs

- Loop iterations are statically divided into code segments called *stages*
- Threads execute stages on different cores
- Thread executing the slowest stage is on the critical path

```plaintext
loop {
    Compute1
    Compute2
    Compute3
}
```
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 executing the slowest pipeline stage
- Thread that is falling behind the most in reaching a barrier
Prioritizing Requests from Limiter Threads

Non-Critical Section  | Critical Section 1  | Barrier
Waiting for Sync or Lock  | Critical Section 2  | Critical Path

Time

Barrier

Thread A
Thread B
Thread C
Thread D

Limiter Thread Identification

Saved Cycles

Most Contended Critical Section: 1
Limiter Thread:  
More on PAMS


Parallel Application Memory Scheduling

Eiman Ebrahimi†, Rustam Miftakhutdinov†, Chris Fallin§
Chang Joo Lee†, José A. Joao†, Onur Mutlu§, Yale N. Patt†

†Department of Electrical and Computer Engineering
The University of Texas at Austin
{ebrahimi, rustam, joao, patt}@ece.utexas.edu

§Carnegie Mellon University
{cfallin, onur}@cmu.edu

‡Intel Corporation
chang.joo.lee@intel.com
Other Ways of Handling Memory Interference
Fundamental Interference Control Techniques

- **Goal**: to reduce/control inter-thread memory interference

1. Prioritization or request scheduling

2. **Data mapping** to banks/channels/ranks

3. Core/source throttling

4. Application/thread scheduling
Designing QoS-Aware Memory Systems: Approaches

- **Smart resources:** Design each shared resource to have a configurable interference control/reduction mechanism
  - QoS-aware memory controllers
  - QoS-aware interconnects
  - 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
  - **QoS-aware data mapping to memory controllers**
  - QoS-aware thread scheduling to cores
Memory Channel Partitioning

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

A new degree of freedom
Mapping data across multiple channels

Muralidhara et al., “Memory Channel Partitioning,” MICRO’11.
Data Mapping in Current Systems

Causes interference between applications’ requests

Muralidhara et al., “Memory Channel Partitioning,” MICRO’11.
Partitioning Channels Between Applications

Eliminates interference between applications’ requests

Muralidhara et al., “Memory Channel Partitioning,” MICRO’11.
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.
Memory Channel Partitioning (MCP) Mechanism

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

Muralidhara et al., “Memory Channel Partitioning,” MICRO’11.
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
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

Muralidhara et al., “Memory Channel Partitioning,” MICRO’11.
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

Muralidhara et al., “Memory Channel Partitioning,” MICRO’11.
Performance of Channel Partitioning

Averaged over 240 workloads

<table>
<thead>
<tr>
<th>Normalized System Performance</th>
<th>FRFCFS</th>
<th>ATLAS</th>
<th>TCM</th>
<th>MCP</th>
<th>IMPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.95</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.05</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.15</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Better system performance than the best previous scheduler at lower hardware cost.
Combined interference control techniques can mitigate interference much more than a single technique alone can do.

The key challenge is:
- Deciding what technique to apply when
- Partitioning work appropriately between software and hardware
More on Memory Channel Partitioning


Reducing Memory Interference in Multicore Systems via Application-Aware Memory Channel Partitioning

- Sai Prashanth Muralidhara, Pennsylvania State University, smuralid@cse.psu.edu
- Lavanya Subramanian, Carnegie Mellon University, lsubrama@ece.cmu.edu
- Onur Mutlu, Carnegie Mellon University, onur@cmu.edu
- Mahmut Kandemir, Pennsylvania State University, kandemir@cse.psu.edu
- Thomas Moscibroda, Microsoft Research Asia, moscitho@microsoft.com
Fundamental Interference Control Techniques

- **Goal**: to reduce/control inter-thread memory interference

1. Prioritization or request scheduling

2. Data mapping to banks/channels/ranks

3. Core/source throttling

4. Application/thread scheduling
Fairness via Source Throttling

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

Core 0  Core 1  Core 2  …  Core N

Shared Cache

Memory Controller

On-chip

Off-chip

Shared Memory Resources

Chip Boundary

DRAM Bank 0  DRAM Bank 1  DRAM Bank 2  …  DRAM Bank K
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?
Source Throttling: A Fairness Substrate

- Key idea: Manage inter-thread interference at the cores (sources), 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

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

if (Unfairness Estimate > Target)
{
  1- Throttle down App-interfering (limit injection rate and parallelism)
  2- Throttle up App-slowest
}
Different fairness objectives can be configured by system software

- Keep maximum slowdown in check
  - Estimated Max Slowdown < Target Max Slowdown
- Keep slowdown of particular applications in check to achieve a particular performance target
  - Estimated Slowdown(i) < Target Slowdown(i)

Support for thread priorities

- Weighted Slowdown(i) = Estimated Slowdown(i) x Weight(i)
Source Throttling Results: Takeaways

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

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

- **Combined approaches** are even more powerful
  - Source throttling and resource-based interference control
Core (Source) Throttling

- **Idea:** Estimate the slowdown due to (DRAM) interference and throttle down threads that slow down others

- **Advantages**
  + Core/request throttling is easy to implement: no need to change the memory scheduling algorithm
  + Can be a general way of handling shared resource contention
  + Can reduce overall load/contention in the memory system

- **Disadvantages**
  - Requires interference/slowdown estimations → difficult to estimate
  - Thresholds can become difficult to optimize → throughput loss
More on Source Throttling (I)

- 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"


Slides (pdf)

Fairness via Source Throttling: A Configurable and High-Performance Fairness Substrate for Multi-Core Memory Systems

Eiman Ebrahimi†  Chang Joo Lee†  Onur Mutlu§  Yale N. Patt†

†Department of Electrical and Computer Engineering
The University of Texas at Austin
{ebrahimi, cjlee, patt}@ece.utexas.edu

§Computer Architecture Laboratory (CALCM)
Carnegie Mellon University
onur@cmu.edu
More on Source Throttling (II)


HAT: Heterogeneous Adaptive Throttling for On-Chip Networks

Kevin Kai-Wei Chang, Rachata Ausavarungnirun, Chris Fallin, Onur Mutlu
Carnegie Mellon University
{kevincha, rachata, cfallin, onur}@cmu.edu
More on Source Throttling (III)

- George Nychis, Chris Fallin, Thomas Moscibroda, Onur Mutlu, and Srinivasan Seshan,

"On-Chip Networks from a Networking Perspective: Congestion and Scalability in Many-core Interconnects"
Proceedings of the 2012 ACM SIGCOMM Conference (SIGCOMM), Helsinki, Finland, August 2012. Slides (pptx)

On-Chip Networks from a Networking Perspective: Congestion and Scalability in Many-Core Interconnects

George Nychis†, Chris Fallin†, Thomas Moscibroda§, Onur Mutlu†, Srinivasan Seshan†
† Carnegie Mellon University
§ Microsoft Research Asia
{gnychis,cfallin,onur,srini}@cmu.edu
moscitho@microsoft.com

SAFARI
Fundamental Interference Control Techniques

- **Goal:** to reduce/control interference

1. **Prioritization** or request scheduling

2. **Data mapping** to banks/channels/ranks

3. **Core/source throttling**

4. **Application/thread scheduling**
   
   Idea: Pick threads that do not badly interfere with each other to be scheduled together on cores sharing the memory system
Application-to-Core Mapping to Reduce Interference

- Reetuparna Das, Rachata Ausavarungnirun, Onur Mutlu, Akhilesh Kumar, and Mani Azimi,
  "Application-to-Core Mapping Policies to Reduce Memory System Interference in Multi-Core Systems"
  Proceedings of the 19th International Symposium on High-Performance Computer Architecture (HPCA), Shenzhen, China, February 2013. Slides (pptx)

- Key ideas:
  - Cluster threads to memory controllers (to reduce across chip interference)
  - Isolate interference-sensitive (low-intensity) applications in a separate cluster (to reduce interference from high-intensity applications)
  - Place applications that benefit from memory bandwidth closer to the controller
Application-to-Core Mapping

- Improve Bandwidth Utilization
- Improve Bandwidth Utilization

- Clustering
- Balancing
- Isolation

- Improve Locality
- Reduce Interference
- Reduce Interference

SAFARI
System Performance

System performance improves by 17%
Average network power consumption reduces by 52%
More on App-to-Core Mapping

- Reetuparna Das, Rachata Ausavarungnirun, Onur Mutlu, Akhilesh Kumar, and Mani Azimi,
  "Application-to-Core Mapping Policies to Reduce Memory System Interference in Multi-Core Systems"
  Slides (pptx)

Application-to-Core Mapping Policies to Reduce Memory System Interference in Multi-Core Systems

Reetuparna Das* Rachata Ausavarungnirun† Onur Mutlu† Akhilesh Kumar‡ Mani Azimi‡
University of Michigan* Carnegie Mellon University† Intel Labs‡
Interference-Aware Thread Scheduling

- An example from scheduling in clusters (data centers)
- Clusters can be running virtual machines
Virtualized Cluster

Distributed Resource Management (DRM) policies

How to dynamically schedule VMs onto hosts?
Conventional DRM Policies

Based on operating-system-level metrics e.g., CPU utilization, memory capacity demand

SAFARI
Microarchitecture-level Interference

• VMs within a host compete for:
  – Shared cache capacity
  – Shared memory bandwidth

Can operating-system-level metrics capture the microarchitecture-level resource interference?
## Microarchitecture Unawareness

<table>
<thead>
<tr>
<th>VM</th>
<th>Operating-system-level metrics</th>
<th>Microarchitecture-level metrics</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CPU Utilization</td>
<td>Memory Capacity</td>
</tr>
<tr>
<td>App</td>
<td>92%</td>
<td>369 MB</td>
</tr>
<tr>
<td>App</td>
<td>93%</td>
<td>348 MB</td>
</tr>
</tbody>
</table>

The diagram illustrates the memory capacity and CPU utilization for different applications running in VMs across multiple cores and levels of cache and memory hierarchy.

**SAFARI**
Impact on Performance

IPC (Harmonic Mean)

0.0
0.2
0.4
0.6

Conventional DRM

SAFARI
We need microarchitecture-level interference awareness in DRM!
A-DRM: Architecture-aware DRM

• **Goal**: Take into account microarchitecture-level shared resource interference
  – Shared cache capacity
  – Shared memory bandwidth

• **Key Idea**:
  – Monitor and detect microarchitecture-level shared resource interference
  – Balance microarchitecture-level resource usage across cluster to minimize memory interference while maximizing system performance
A-DRM: Architecture-aware DRM

OS+Hypervisor

Hosts

Controller

A-DRM: Global Architecture – aware Resource Manager

Profiling Engine

Architecture-aware Interference Detector

Architecture-aware Distributed Resource Management (Policy)

Migration Engine
More on Architecture-Aware DRM

- Hui Wang, Canturk Isci, Lavanya Subramanian, Jongmoo Choi, Depei Qian, and Onur Mutlu,
"A-DRM: Architecture-aware Distributed Resource Management of Virtualized Clusters"
[Slides (pptx) (pdf)]

A-DRM: Architecture-aware Distributed Resource Management of Virtualized Clusters

Hui Wang†*, Canturk Isci‡, Lavanya Subramanian*, Jongmoo Choi‡*, Depei Qian†, Onur Mutlu*
†Beihang University, ‡IBM Thomas J. Watson Research Center, *Carnegie Mellon University, ‡Dankook University
{hui.wang, depeiq}@buaa.edu.cn, canturk@us.ibm.com, {lsubrama, onur}@cmu.edu, choijm@dankook.ac.kr
Interference-Aware Thread Scheduling

- Advantages
  - Can eliminate/minimize interference by scheduling “symbiotic applications” together (as opposed to just managing the interference)
  - Less intrusive to hardware (less need to modify the hardware resources)

- Disadvantages and Limitations
  -- High overhead to migrate threads between cores and machines
  -- Does not work (well) if all threads are similar and they interfere
Summary: Fundamental Interference Control Techniques

- **Goal**: to reduce/control interference

1. **Prioritization** or request scheduling

2. **Data mapping** to banks/channels/ranks

3. **Core/source throttling**

4. **Application/thread scheduling**

Best is to combine all. How would you do that?
Summary: Memory QoS Approaches and Techniques

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

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

- **Combined approaches and techniques are the most powerful**
  - *Integrated Memory Channel Partitioning and Scheduling* [MICRO’11]
Summary: Memory Interference and QoS

- QoS-unaware memory →
  uncontrollable and unpredictable system

- Providing QoS awareness improves performance, predictability, fairness, and utilization of the memory system

- Discussed many new techniques to:
  - Minimize memory interference
  - Provide predictable performance

- Many new research ideas needed for integrated techniques and closing the interaction with software
What Did We Not Cover?

- Prefetch-aware shared resource management
- DRAM-controller co-design
- Cache interference management
- Interconnect interference management
- Write-read scheduling
- ...

SAFARI