Computer Architecture
Lecture 15: Memory Contention: Performance, QoS, Complexity

Prof. Onur Mutlu
ETH Zürich
Fall 2022
17 November 2022
Recall: Memory Controller Design Is Becoming More Difficult

- Heterogeneous agents: CPUs, GPUs, HWAs, DMA engine, ...
- Main memory interference between CPUs, GPUs, HWAs
- Many timing constraints for various memory types
- Many goals at the same time: performance, fairness, QoS, energy efficiency, ...
Recall: Memory Performance Attacks


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

Thomas Moscibroda  Onur Mutlu
Microsoft Research
{moscitho,onur}@microsoft.com

http://www.youtube.com/watch?v=VJzZbwgBfy8
Recall: Interconnect Based Starvation

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
Lecture on Other QoS Techniques

Application-to-Core Mapping

Improve Bandwidth Utilization
- Balancing
- Radial Mapping

Improve Locality
- Clustering
- Isolation

Reduce Interference
Lecture on Other QoS Techniques
Memory Channel Partitioning

Partitioning Channels Between Applications

Eliminates interference between applications’ requests

https://www.youtube.com/watch?v=rjmVKDdl8Jc&list=PL5Q2soXY2Zi_7UBNmC9B8Yr5JSwTG9yH4&index=5
Handling CPU-IO Interference

Donghyuk Lee, Lavanya Subramanian, Rachata Ausavarungrinr, Jongmoo Choi, and Onur Mutlu,
"Decoupled Direct Memory Access: Isolating CPU and IO Traffic by Leveraging a Dual-Data-Port DRAM"
Proceedings of the 24th International Conference on Parallel Architectures and Compilation Techniques (PACT), San Francisco, CA, USA, October 2015.
[Slides (pptx) (pdf)]

Decoupled Direct Memory Access: Isolating CPU and IO Traffic by Leveraging a Dual-Data-Port DRAM

Donghyuk Lee*  Lavanya Subramanian*  Rachata Ausavarungrinr*  Jongmoo Choi†  Onur Mutlu*

*Carnegie Mellon University
†Dankook University
{donghyu1, lsubrama, rachata, onur}@cmu.edu  choijm@dankook.ac.kr
QoS-Aware Memory Scheduling: Revolution & Evolution
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

- **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, Top Picks’11]
  - 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, TPDS’16]
  - 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’08’09]
- 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] [Seshadri+ ISCA’14]
- 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

- **Criticality-Aware Memory Scheduling for GPUs** [Jog+ SIGMETRICS’16]
  - Idea: Prioritize latency-critical cores’ requests in a GPU system
  - Takeaway: Need to carefully balance locality and criticality to make sure performance improves by taking advantage of both

- **Worst-case Execution Time Based Memory Scheduling for Real-Time Systems** [Kim+ RTAS’14, JRTS’16]
Recall: 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)]
Recall: 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
  - Scalability to many threads and heterogeneous cores unclear
More on PAR-BS

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

One of the 12 computer architecture papers of 2008 selected as Top Picks by IEEE Micro.

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

Onur Mutlu Thomas Moscibroda
Microsoft Research
{onur,moscitho}@microsoft.com

http://www.youtube.com/watch?v=UB1kgYR-4V0
More on PAR-BS

Onur Mutlu and Thomas Moscibroda,
"Parallelism-Aware Batch Scheduling: Enabling High-Performance and Fair Memory Controllers"


**Parallelism-Aware Batch Scheduling: Enabling High-Performance and Fair Shared Memory Controllers**

Uncontrolled interthread interference in main memory can destroy individual threads’ memory-level parallelism, effectively serializing the memory requests of a thread whose latencies would otherwise have largely overlapped, thereby reducing single-thread performance. The parallelism-aware batch scheduler preserves each thread’s memory-level parallelism, ensures fairness and starvation freedom, and supports system-level thread priorities.
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
ATLAS consistently provides higher system throughput than all previous scheduling algorithms.

System throughput = \( \sum \text{Speedup} \)
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)

*Best paper session. One of the four papers nominated for the Best Paper Award by the Program Committee.*
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"
43rd International Symposium on Microarchitecture (MICRO),
No previous memory scheduling algorithm provides both the best fairness and system throughput
Throughput vs. Fairness

**Throughput biased approach**

Prioritize less memory-intensive threads

**Fairness biased approach**

Take turns accessing memory

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

- Does not starve
  - thread C
  - not prioritized
  - reduced throughput

- Starvation → unfairness

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
Thread Cluster Memory Scheduling [Kim+ MICRO’10]

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

Memory-non-intensive

Threads in the system

Memory-intensive

Non-intensive cluster

Prioritized

Throughput

higher priority

Intensive cluster

Fairness

higher priority

SAFARI
1. Clustering
Clustering Threads

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

- **Higher MPKI** threads are clustered
- **Non-intensive** cluster
- **Intensive** cluster

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

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

**ClusterThreshold**

\[ \alpha < 10\% \]
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
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

**Intensive Cluster**

Increases fairness

higher priority

Most prioritized

thread

thread

thread
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*  
 Req  

*streaming*  
 Stuck 
 Req

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

- 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

+ Niceness

- Niceness
**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 $\rightarrow$ 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

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

When configuration parameter is varied...

- Adjusting ClusterThreshold
- Better system throughput
- Better fairness
- 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 $\Rightarrow$ throughput
  2. Shuffle priorities in *intensive* cluster $\Rightarrow$ fairness
  3. Shuffling should favor *nice* threads $\Rightarrow$ 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


One of the 11 computer architecture papers of 2010 selected as Top Picks by IEEE Micro.

Thread Cluster Memory Scheduling: Exploiting Differences in Memory Access Behavior

Yoongu Kim yoonguk@ece.cmu.edu
Michael Papamichael papamix@cs.cmu.edu
Onur Mutlu onur@cmu.edu
Mor Harchol-Balter harchol@cs.cmu.edu

Carnegie Mellon University
More on TCM


**Thread Cluster Memory Scheduling**

Memory schedulers in multicore systems should carefully schedule memory requests from different threads to ensure high system performance and fair, fast progress of each thread. No existing memory scheduler provides both the highest system performance and highest fairness. Thread Cluster Memory scheduling is a new algorithm that achieves the best of both worlds by differentiating latency-sensitive threads from bandwidth-sensitive ones and employing different scheduling policies for each.
The Blacklisting Memory Scheduler

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

- Performance
- Fairness
- Simplicity

FRFCFS
FRFCFSCap
PARBS
ATLAS
TCM
Blacklisting

Ideal

Highest performance

Close to fairest

Close to simplest

Blacklisting is the closest scheduler to ideal
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,donghyu1,visesh,harshar,onur}@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]

---

BLISS: Balancing Performance, Fairness and Complexity in Memory Access Scheduling

Lavanya Subramanian, Donghyuk Lee, Vivek Seshadri, Harsha Rastogi, and Onur Mutlu
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)
Lecture on Parallel Application Scheduling

Prioritizing Requests from Limiter Threads

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

Thread A
Thread B
Thread C
Thread D

Limiter Thread Identification

Most Contended Critical Section: B

ETH ZENTRUM
Computer Arch - Lecture 13: Memory Interference and Quality of Service II (ETH Zürich, Spring 2020)

https://www.youtube.com/watch?v=Axye9VqQT7w&list=PL5Q2soXY2Zi9xdy1qBxUz7xRPS-wisBN&index=26
Observation: Limiting Bottlenecks Change Over Time

A = full linked list; B = empty linked list
repeat

Lock A
  Traverse list A
  Remove X from A
Unlock A
Compute on X
Lock B
  Traverse list B
  Insert X into B
Unlock B
until A is empty
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:

```python
    loop1 {
        Compute
    }
    barrier
    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
}
```
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 executing the slowest pipeline stage
- Thread that is falling behind the most in reaching a barrier
Prioritizing Requests from Limiter Threads

Critical Section 1 Barrier
Non-Critical Section
Waiting for Sync or Lock

Thread D
Thread C
Thread B
Thread A

Critical Section 2
Critical Path
Saved Cycles

Limiter Thread Identification

Most Contended Critical Section: 1
Limiter Thread: 🧨
Parallel App Mem Scheduling: Pros and Cons

- **Upsides:**
  - Improves the performance of multi-threaded applications
  - Provides a mechanism for estimating “limiter threads”
  - Opens a path for slowdown estimation for multi-threaded applications

- **Downsides:**
  - What if there are multiple multi-threaded applications running together?
  - Limiter thread estimation can be complex
  - Joao+, “Bottleneck Identification and Scheduling,” ASPLOS 2012
More on PAMS

- 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)

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
Prioritizing Requests from Limiter Threads

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

Thread A
Thread B
Thread C
Thread D

Limiter Thread Identification

Most Contended Critical Section: 1
Limiter Thread: B
Bottleneck Identification & Scheduling

- Jose A. Joao, M. Aater Suleman, Onur Mutlu, and Yale N. Patt, "Bottleneck Identification and Scheduling in Multithreaded Applications"

Bottleneck Identification and Scheduling in Multithreaded Applications

José A. Joao  
ECE Department  
The University of Texas at Austin  
joao@ece.utexas.edu

M. Aater Suleman  
Calxeda Inc.  
aater.suleman@calxeda.com

Onur Mutlu  
Computer Architecture Lab.  
Carnegie Mellon University  
onur@cmu.edu

Yale N. Patt  
ECE Department  
The University of Texas at Austin  
patt@ece.utexas.edu
Utility-Based Bottleneck Acceleration

- Jose A. Joao, M. Aater Suleman, Onur Mutlu, and Yale N. Patt, "Utility-Based Acceleration of Multithreaded Applications on Asymmetric CMPs"

*Proceedings of the 40th International Symposium on Computer Architecture (ISCA),* Tel-Aviv, Israel, June 2013. [Slides (ppt)](slides_ppt) [Slides (pdf)](slides_pdf)

Utility-Based Acceleration of Multithreaded Applications on Asymmetric CMPs

José A. Joao † M. Aater Suleman †† Onur Mutlu § Yale N. Patt †

† ECE Department
The University of Texas at Austin
Austin, TX, USA
{joao, patt}@ece.utexas.edu

†† Flux7 Consulting
Austin, TX, USA
suleman@hps.utexas.edu

§ Computer Architecture Laboratory
Carnegie Mellon University
Pittsburgh, PA, USA
onur@cmu.edu
Lecture on Bottleneck Acceleration

Observation: Limiting Bottlenecks Change Over Time

A=full linked list; B=empty linked list
repeat
  Lock A
  Traverse list A
  Remove X from A
  Unlock A
  Compute on X
  Lock B
  Traverse list B
  Insert X into B
  Unlock B
until A is empty
Memory Scheduling for Heterogeneous Systems
Heterogeneous agents: CPUs, GPUs, HWAs, DMA engine, ...
Memory resources shared by CPUs/GPUs/HWAs → Interference

How to allocate resources to heterogeneous agents to mitigate interference and provide predictable performance?
Current SoC Architectures: Heterogeneity

Source: https://www.anandtech.com/show/16252/mac-mini-apple-m1-tested

Apple M1, 2021
Lecture on Heterogeneous System Scheduling

SMS: Staged Memory Scheduling

Stage 1
Batch Formation

Stage 2
Batch Scheduler

Stage 3
DRAM Command Scheduler

To DRAM

Core 1  Core 2  Core 3  Core 4  GPU

Batch 1  Batch 2  Batch 3  Batch 4

Bank 1  Bank 2  Bank 3  Bank 4

https://www.youtube.com/watch?v=Axye9VqQT7w&list=PL5Q2soXY2Zi9xydyIgBxUz7xRPS-wisBN&index=26
Asymmetry Enables Customization

Symmetry: One size fits all
- Energy and performance suboptimal for different “workload” behaviors

Asymmetric: Enables customization and adaptation
- Processing requirements vary across workloads (applications and phases)
- Execute code on best-fit resources (minimal energy, adequate perf.)
Staged Memory Scheduling

- 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)

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

Rachata Ausavarungnirun†, Kevin Kai-Wei Chang†, Lavanya Subramanian†, Gabriel H. Loh‡, Onur Mutlu‡

†Carnegie Mellon University
{rachata, kevincha, lsubrama, onur}@cmu.edu

‡Advanced Micro Devices, Inc.
gabe.loh@amd.com
DASH: Deadline-Aware Memory Scheduler

- 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

SAFARI
Handling CPU-IO Interference

Donghyuk Lee, Lavanya Subramanian, Rachata Ausavarungnirun, Jongmoo Choi, and Onur Mutlu,
"Decoupled Direct Memory Access: Isolating CPU and IO Traffic by Leveraging a Dual-Data-Port DRAM"
Proceedings of the 24th International Conference on Parallel Architectures and Compilation Techniques (PACT), San Francisco, CA, USA, October 2015.
[Slides (pptx) (pdf)]

Decoupled Direct Memory Access: Isolating CPU and IO Traffic by Leveraging a Dual-Data-Port DRAM

Donghyuk Lee*, Lavanya Subramanian*, Rachata Ausavarungnirun*, Jongmoo Choi†, Onur Mutlu*
* Carnegie Mellon University
{donghyu1, lsubrama, rachata, onur}@cmu.edu
† Dankook University
choijm@dankook.ac.kr

SAFARI
Isolating CPU and IO Traffic by Leveraging a Dual-Data-Port DRAM

Decoupled Direct Memory Access

Donghyuk Lee
Lavanya Subramanian, Rachata Ausavarungnirun, Jongmoo Choi, Onur Mutlu

SAFARI
Main memory connects processor and IO devices as an *intermediate layer*
Physical System Implementation

- Processor
- Main memory
- IO devices
- CPU access
- IO access
- High Pin Cost in Processor
- High Contention in Memory Channel
Our Approach

Enabling IO channel, \textit{decoupled} \& \textit{isolated} from CPU channel
Executive Summary

- **Problem**
  - CPU and IO accesses contend for the shared memory channel

- **Our Approach:** *Decoupled Direct Memory Access (DDMA)*
  - Design new DRAM architecture with two independent data ports
    - *Dual-Data-Port DRAM*
  - Connect one port to CPU and the other port to IO devices
    - *Decouple CPU and IO accesses*

- **Applications**
  - Communication between compute units (e.g., CPU – GPU)
  - In-memory communication (e.g., bulk in-memory copy/init.)
  - Memory-storage communication (e.g., page fault, IO prefetch)

- **Result**
  - Significant *performance improvement* (20% in 2 ch. & 2 rank system)
  - *CPU pin count reduction* (4.5%)
Problem 1: Memory Channel Contention

Memory Channel Contention

Processor Chip

DRAM Chip

CPU

memory controller

DMA

IO interface

main memory

graphics

network

storage

USB
Problem 1: Memory Channel Contention

A large fraction of the execution time is spent on IO accesses.
Problem 2: High Cost for IO Interfaces

Integrating IO interface on the processor chip leads to high area cost.

- Processor Pin Count (w/ power pins): 959 pins in total
- Processor Pin Count (w/o power pins): 359 pins in total

IO interface (10.6%)

IO interface (28.4%)
Shared Memory Channel

• **Memory channel contention** for IO access and CPU access

• **High area cost** for integrating IO interfaces on processor chip
Decoupled DMA Approach

Processor Chip

DRAM Chip

DMA Chip

CPU

memory controller

DMA IO interface

Dual-Data-Port DRAM

DMA IO interface

DMA control

graphics

network

storage

USB
Decoupled DMA Approach

**Decoupled Direct Memory Access**

![Diagram](image)

**CPU ACCESS**
- Processor Chip
- Dual-Data-Port DRAM
  - Port 1
  - Port 2

**IO ACCESS**
- DMA Chip
- DMA IO interface
  - graphics
  - network
  - storage
  - USB

SAFARI
Understanding Host Interconnect Congestion

Saksham Agarwal  
Cornell University

Rachit Agarwal  
Cornell University

Behnam Montazeri  
Google

Masoud Moshref  
Google

Khaled Elmeleegy  
Google

Luigi Rizzo  
Google

Marc Asher de Kruijf  
Google

Gautam Kumar  
Google

Sylvia Ratnasamy  
Google & UC Berkeley

David Culler  
Google

Amin Vahdat  
Google

We present evidence and characterization of host congestion in production clusters: adoption of high-bandwidth access links leading to emergence of bottlenecks within the host interconnect (NIC-to-CPU data path). We demonstrate that contention on existing IO memory management units and/or the memory subsystem can significantly reduce the available NIC-to-CPU bandwidth, resulting in hundreds of microseconds of queueing delays and eventual packet drops at hosts (even when running a state-of-the-art congestion control protocol that accounts for CPU-induced host congestion). We also discuss implications of host interconnect congestion to design of future host architecture, network stacks and network protocols.
3.2 Memory bus induced host congestion

**Intuition.** Another root cause of host congestion is the rapidly reducing gap between access link bandwidth and memory bus bandwidth: in large servers, applications that perform large volumes of memory operations can lead to starvation of memory requests coming from the NIC. Specifically, within the host interconnect, CPUs reading/writing data to memory share the memory bus with the NIC performing DMA operations; when memory bus is contended, per-DMA latency increases for memory requests coming from the NIC. This increase leads to an effect similar to the IOMMU case discussed in the previous subsection: increase in latency eventually results in PCIe bandwidth underutilization, and in-flight packets resulting in NIC buffers quickly building up. This, in turn, results in large host delays and packet drops even when host is receiving data at rate lower than the access link bandwidth; these delays and drops can result in subsequent rate reduction and link underutilization.
Predictable Performance: Strong Memory Service Guarantees
Goal: Predictable Performance in Complex Systems

- Heterogeneous agents: CPUs, GPUs, HWAs, DMA engine, ...
- Memory resources shared by CPUs/GPUs/HWAs → Interference

How to allocate resources to heterogeneous agents to mitigate interference and provide predictable performance?
Lecture on Predictable Performance

Key Observation 1

For a memory bound application, Performance ∝ Memory request service rate

Normalized Performance

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

Normalized Request Service Rate
Lecture on Predictable Performance

Effectiveness of MISE in Enforcing QoS

Across 3000 data points

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

Memory Systems - Lecture 6.4: Memory Interface and QoS (Technion, Summer 2018)

https://www.youtube.com/watch?v=15hRJhGWGA&list=PL5Q2soXY2Zi-IymxXpH_9vIvZC0eA7Yfn9&index=19
Predictable Performance Readings (I)


Best paper award.
Predictable Performance Readings (II)

- 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)
Predictable Performance Readings (III)

- 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
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]
Predictable Performance Readings (V)

Donghyuk Lee, Lavanya Subramanian, Rachata Ausavarungnirun, Jongmoo Choi, and Onur Mutlu,

"Decoupled Direct Memory Access: Isolating CPU and IO Traffic by Leveraging a Dual-Data-Port DRAM"

Proceedings of the 24th International Conference on Parallel Architectures and Compilation Techniques (PACT), San Francisco, CA, USA, October 2015.

[Slides (pptx) (pdf)]

Decoupled Direct Memory Access: Isolating CPU and IO Traffic by Leveraging a Dual-Data-Port DRAM

Donghyuk Lee*  Lavanya Subramanian*  Rachata Ausavarungnirun*  Jongmoo Choi†  Onur Mutlu*

*Carnegie Mellon University  †Dankook University  
{donghyu1, lsubrama, rachata, onur}@cmu.edu  choijm@dankook.ac.kr

Figure 1: Methods to Connect Cores and Data Sources
Other QoS Approaches
Recall: 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**
Lecture on Other QoS Techniques

Application-to-Core Mapping

Improve Bandwidth Utilization
- Balancing
- Radial Mapping

Improve Locality
- Clustering

Reduce Interference
- Isolation

ETH ZENTRUM
Computer Arch - Lecture 13: Memory Interference and Quality of Service II (ETH Zürich, Spring 2020)

https://www.youtube.com/watch?v=Axye9VqQT7w&list=PL5Q2soXY2Zi9xidy1gBxUz7xRPS-wisBN&index=26
Lecture on Other QoS Techniques

Computer Arch - Lecture 13: Memory Interference and Quality of Service II (ETH Zürich, Spring 2020)

https://www.youtube.com/watch?v=Axye9VqQT7w&list=PL5Q2soXY2Zi9xidyIqBxUz7xRPS-wisBN&index=26
Memory Channel Partitioning

Partitioning Channels Between Applications

Eliminates interference between applications’ requests

https://www.youtube.com/watch?v=rjmVKDdl8Jc&list=PL5Q2soXY2Zi_7UBNmC9B8Yr5JSwTG9yH4&index=5
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"
  Proceedings of the 44th International Symposium on Microarchitecture (MICRO), Porto Alegre, Brazil, December 2011. Slides (pptx)

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  
lsubraman@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

https://www.youtube.com/watch?v=yEYEzFwAY9g
Source Throttling (I)


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

SAFARI
Source Throttling (II)

- Kevin Chang, Rachata Ausavarungnirun, Chris Fallin, and Onur Mutlu, "HAT: Heterogeneous Adaptive Throttling for On-Chip Networks"

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
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
gnychis,cfallin,onur,srini}@cmu.edu
§ Microsoft Research Asia
moscitho@microsoft.com
Application-to-Core Mapping to Reduce Interference

- Reetuparna Das, Rachata Ausavarungrunirun, Onur Mutlu, Akhilesh Kumar, and Mani Azimi,

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


Slides (pptx)
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, depei.q}@buaa.edu.cn, canturk@us.ibm.com, {lsubrama, onur}@cmu.edu, choijm@dankook.ac.kr
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)

MCP Micro 2011 Talk
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"

Proceedings of the 44th International Symposium on Microarchitecture (MICRO), Porto Alegre, Brazil, December 2011. Slides (pptx)

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

https://www.youtube.com/watch?v=yEYEzFwAY9g
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.
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

![Diagram](image-url)

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
Performance of Channel Partitioning

Better system performance than the best previous scheduler at lower hardware cost
An Example of Poor Channel Partitioning
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
MCP and IMPS: Pros and Cons

- Upsides:
  - Keeps the memory scheduling hardware simple
  - Combines multiple interference reduction techniques
  - Can provide performance isolation across applications mapped to different channels
  - General idea of partitioning can be extended to smaller granularities in the memory hierarchy: banks, subarrays, etc.

- Downsides:
  - Reacting is difficult if workload changes behavior after profiling
  - Overhead of moving pages between channels restricts benefits
More on 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"

Proceedings of the 44th International Symposium on Microarchitecture (MICRO), Porto Alegre, Brazil, December 2011. Slides (pptx)

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

https://www.youtube.com/watch?v=yEYEzFwAY9g
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"

FST ASPLOS 2010 Talk
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

Shared Memory Resources

Chip Boundary
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

Fairness via Source Throttling (FST)

- Two components (interval-based)

- Run-time unfairness evaluation (in hardware)
  - Dynamically estimates the unfairness (application slowdowns) 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) [ASPLOS’10]

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
}
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>
</tbody>
</table>

Total # of MSHRs: 128
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) 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
Source Throttling: Ups and Downs

**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 slowdown estimations → difficult to estimate
- Thresholds can become difficult to optimize
  → throughput loss due to too much throttling
  → can be difficult to find an overall-good configuration
More on Source Throttling (I)


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)

- Kevin Chang, Rachata Ausavarungrunrun, Chris Fallin, and Onur Mutlu, "HAT: Heterogeneous Adaptive Throttling for On-Chip Networks"

HAT: Heterogeneous Adaptive Throttling for On-Chip Networks

Kevin Kai-Wei Chang, Rachata Ausavarungrunrun, Chris Fallin, Onur Mutlu
Carnegie Mellon University
{kevincha, rachata, cfallin, onur}@cmu.edu
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)
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"
  
  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
Multi-Core to Many-Core

Multi-Core

Many-Core
Many-Core On-Chip Communication

Applications
Light
Heavy

Memory Controller
Shared Cache Bank
Problem: Spatial Task Scheduling

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

- Clustering
- Balancing
- Isolation
- Radial Mapping

- Improve Locality
- Reduce Interference
- Improve Bandwidth Utilization
- Reduce Interference
- Improve Bandwidth Utilization

SAFARI
Step 1 — Clustering

Inefficient data mapping to memory and caches
Step 1 — Clustering

Improved Locality

Reduced Interference

Cluster 0

Cluster 1

Cluster 2

Cluster 3

Improved Locality

Reduced Interference
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"
  *Proceedings of the 19th International Symposium on High-Performance Computer Architecture (HPCA)*, Shenzhen, China, February 2013. Slides (pptx)
Interference-Aware Thread Scheduling

- An example from scheduling in compute clusters (data centers)
  - Data centers can be running virtual machines

- How to co-schedule virtual machines such that system throughput in the entire data center is maximized?

- Interference in a single node matters
A-DRM: Architecture-aware DRM

• **Goal**: Take into account microarchitecture-level shared resource interference in cluster scheduling
  – 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
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*

*Proceedings of the 11th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE), Istanbul, Turkey, March 2015.*

[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 and data between cores and machines
  -- Does not work (well) if all threads are similar and they interfere
Summary
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 at reducing interference
  - No single best approach for all workloads

- **Techniques:** Request/thread scheduling, source throttling, memory partitioning
  - All approaches are effective at 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
- **DRAM designs to reduce interference**
- Interference & QoS in processing-in-memory
- ...

SAFARI
What the Future May Bring

- Memory QoS techniques for heterogeneous SoC systems
  - Many accelerators, processing in/near memory, better predictability, higher performance

- Combinations of memory QoS/performance techniques
  - E.g., data mapping and scheduling

- Use of machine learning techniques to manage resources

- End-to-end QoS, predictability, and performance in complex systems
Backup Slides
Memory Scheduling for Heterogeneous Systems
Lecture on Heterogeneous System Scheduling

SMS: Staged Memory Scheduling

Core 1  Core 2  Core 3  Core 4  GPU

Stage 1
Batch Formation

Stage 2
Batch Scheduler

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

To DRAM

https://www.youtube.com/watch?v=Axye9VqQT7w&list=PL5Q2soXY2Zi9xyd1qBxUz7xRPS-wisBN&index=26
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  Stage 2  Stage 3

Batch Scheduler

Batch Formation

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

Bank 1
Bank 2
Bank 3
Bank 4

To DRAM
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 $\rightarrow$ 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 against GPU weight with different schedulers: ATLAS, TCM, FR-FCFS. The graph plots system performance on the y-axis against GPU weight on the x-axis. The best previous scheduler is highlighted in red, showing an upward trend. Other schedulers are shown in green, maintaining a lower performance level.
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)

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

Rachata Ausavarungnirun† Kevin Kai-Wei Chang† Lavanya Subramanian† Gabriel H. Loh‡ Onur Mutlu‡

†Carnegie Mellon University
{rachata,kevincha,lsubrama,onur}@cmu.edu

‡Advanced Micro Devices, Inc.
gabe.loh@amd.com
DASH Memory Scheduler

[TACO 2016]
Current SoC Architectures

- Heterogeneous agents: CPUs and HWAs
  - HWA : Hardware Accelerator
- Main memory is shared by CPUs and HWAs \(\rightarrow\) 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

Switch probabilistically
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
Predictable Performance: 
Strong Memory Service Guarantees
Lecture on Predictable Performance

Key Observation 1

For a memory bound application, 
Performance ∝ Memory request service rate

Normalized Performance

Normalized Request Service Rate

omnetpp

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

SAFARI

https://www.youtube.com/watch?v=Axye9VqQT7w&list=PL5Q2soXY2Zi9xidyIgBxUz7xRPS-wisBN&index=26
Lecture on Predictable Performance

Effectiveness of MISE in Enforcing QoS

Across 3000 data points

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

Memory Systems - Lecture 6.4: Memory Interface and QoS (Technion, Summer 2018)
163 views • Oct 12, 2018

https://www.youtube.com/watch?v=15hRLhGWGA&list=PL5Q2soXY2Zi-IymxXpH_9vZCOeA7Yfn9&index=19
Goal: Predictable Performance in Complex Systems

- Heterogeneous agents: CPUs, GPUs, HWAs, DMA engine, ...
- Memory resources shared by CPUs/GPUs/HWAs → Interference

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


Predictable Performance Readings (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"
Best paper award.
Predictable Performance Readings (II)

- 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)](#)*
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
MISE: Providing Performance Predictability in Shared Main Memory Systems

Lavanya Subramanian, Vivek Seshadri, Yoongu Kim, Ben Jaiyen, Onur Mutlu
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
Outline

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

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

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

For a memory bound application, the performance is proportional to the memory request service rate. The normalized performance is shown in the graph below. The graph compares the performance of different applications (omnetpp, mcf, astar) under shared and alone conditions. The shared conditions show a slower performance due to increased contention for memory resources.

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

Request Buffer State

Main Memory

2. Run with another application

Request Buffer State

Main Memory

3. Run with another application: **highest priority**

Request Buffer State

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

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

- Memory-bound application

No interference

With interference

Memory phase slowdown dominates overall slowdown
Key Observation 3

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

\[
\text{Slowdown} = (1 - \alpha) + \alpha \frac{R_{SR, \text{Alone}}}{R_{SR, \text{Shared}}}
\]
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 $\text{RSR}_{\text{Shared}}$ and $\alpha$

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

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

- Memory Phase Fraction ($\lambda$)
  - 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

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\textsubscript{Alone}

- When an application has highest priority, it still experiences some interference.
Solution: Determine and remove interference cycles from \( RSR_{\text{Alone}} \) calculation

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

A cycle is an interference cycle if

- a request from the highest priority application is waiting in the request buffer \( \text{and} \)
- 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
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 +/- $\varepsilon$, increase bandwidth allocation.
  - If slowdown < bound B +/- $\varepsilon$, 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
MISE is effective in
1. meeting the slowdown bound for the QoS-critical application
2. improving performance of non-QoS-critical applications
Effectiveness of MISE in Enforcing QoS

Across 3000 data points

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

MISE-QoS correctly predicts whether or not the bound is met for 95.7% of workloads.
Performance of Non-QoS-Critical Applications

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

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: Pros and Cons

- **Upsides:**
  - Simple new insight to estimate slowdown
  - Much more accurate slowdown estimations than prior techniques (STFM, FST)
  - Enables a number of QoS mechanisms that can use slowdown estimates to satisfy performance requirements

- **Downsides:**
  - Slowdown estimation is not perfect - there are still errors
  - Does not take into account caches and other shared resources in slowdown estimation
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)
Extending MISE to Shared Caches: ASM

- 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]
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
More on 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"

Proceedings of the 44th International Symposium on Microarchitecture (MICRO), Porto Alegre, Brazil, December 2011. Slides (pptx)

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

https://www.youtube.com/watch?v=yEYEzFwAY9g
More on Source Throttling (I)


Best paper award.

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)

- Kevin Chang, Rachata Ausavarungnirun, Chris Fallin, and Onur Mutlu, "HAT: Heterogeneous Adaptive Throttling for On-Chip Networks".

HAT: Heterogeneous Adaptive Throttling for On-Chip Networks

Kevin Kai-Wei Chang, Rachata Ausavarungnirun, Chris Fallin, Onur Mutlu
Carnegie Mellon University
{kevincha,rachata,cfallin,onor}@cmu.edu

SAFARI
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
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"
  Proceedings of the 19th International Symposium on High-Performance Computer Architecture (HPCA), Shenzhen, China, February 2013. Slides (pptx)
Interference-Aware Thread Scheduling

- An example from scheduling in compute clusters (data centers)
- Data centers 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.
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>

CPU Utilization: 92% for App, 93% for App
Memory Capacity: 369 MB for App, 348 MB for App

Microarchitecture-level metrics:
- LLC Hit Ratio: 2%
- Memory Bandwidth: 2267 MB/s for App, 1 MB/s for App

Diagrams showing the memory capacity flow and CPU to memory connections.
Impact on Performance

IPC (Harmonic Mean)

Conventional DRM with Microarchitecture Awareness

Memory Capacity

CPU

VM

App

Core0

Core1

DRAM

LLC

App

STREAM

gromacs

SAFARI
Impact on Performance

We need microarchitecture-level interference awareness in DRM!

Memory Capacity

CPU

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

VM

App

CPU/Memory Capacity

Architectural Resources

Profiler

VM

App

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 and data between cores and machines
  -- Does not work (well) if all threads are similar and they interfere