18-447: Computer Architecture
Lecture 27: Memory Scheduling

Prof. Onur Mutlu
Carnegie Mellon University
Spring 2013, 4/10/2013
Homework 6

- Due April 19 (Friday)
- Topics: Virtual memory and cache interaction, main memory, memory scheduling

- Strong suggestion:
  - Please complete this before the exam to prepare for the exam

- Reminder:
  - Homeworks are mainly for your benefit and learning (and preparation for the exam).
  - They are not meant to be a large part of your grade
Lab 6: Memory Hierarchy

- Due April 22 (Monday)
- Cycle-level modeling of L2 cache and DRAM-based main memory

**Extra credit: Prefetching**
- Design your own hardware prefetcher to improve system performance

- HW 6 and Lab 6 are synergistic – work on them together
Heads Up: Midterm II Next Week

- April 17

- Similar format as Midterm I
- Suggestion: Do Homework 6 to prepare for the Midterm
Last Lecture

- DRAM Refresh
  - Ways of reducing refresh impact

- Memory Controllers
  - Structure and operation

- Memory Access Scheduling
  - FR-FCFS – row-hit-first scheduling
Today

- Memory Interference (and Techniques to Manage It)
  - With a focus on Memory Request Scheduling

- Emerging Memory Technologies and Hybrid Memory Systems (if time permits)

- Last 15 minutes: Q&A with Dr. William Strecker
Our Guests Today

- Dr. William Strecker and Nancy Strecker
- Architect of VAX
- SVP Corporate Strategy & Tech. and CTO, DEC
- CMU Alum (BS’66, MS’67, PhD’71)

- IEEE Wallace McDowell Award Recipient (1985)
  - For being principal designer of the VAX architecture and for contributions to local area networks, high-performance interconnects, caches, and memory hierarchies

Recommended Reading

- Gordon Bell and William D. Strecker, “What Have We Learned from the PDP-11 - What We Have Learned from VAX and Alpha,” 25 Years of ISCA, Retrospectives and Reprints, 1998.

- Original paper
Memory Interference and Scheduling in Multi-Core Systems
Review: A Modern DRAM Controller
Access Address:
(Row 0, Column 0)
(Row 0, Column 1)
(Row 0, Column 85)
(Row 1, Column 0)

Row address 0

Column address 0

Concurrent Row Access

Columns

Row decoder

Rows

Row Buffer

HIT

CONFLICT!
Scheduling Policy for Single-Core Systems

- A row-conflict memory access takes significantly longer than a row-hit access
- Current controllers take advantage of the row buffer

FR-FCFS (first ready, first come first served) scheduling policy
1. Row-hit first
2. Oldest first

Goal 1: Maximize row buffer hit rate → maximize DRAM throughput
Goal 2: Prioritize older requests → ensure forward progress

Is this a good policy in a multi-core system?
Trend: Many Cores on Chip

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

- **AMD Barcelona**
  - 4 cores

- **Intel Core i7**
  - 8 cores

- **IBM Cell BE**
  - 8+1 cores

- **IBM POWER7**
  - 8 cores

- **Sun Niagara II**
  - 8 cores

- **Nvidia Fermi**
  - 448 “cores”

- **Intel SCC**
  - 48 cores, networked

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

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

- What do we get today?
(Un)expected Slowdowns in Multi-Core

Uncontrolled Interference: An Example

Multi-Core Chip

Shared DRAM Memory System

unfairness

DRAM MEMORY CONTROLLER

L2 CACHE

INTERCONNECT

DRAM Bank 0

DRAM Bank 1

DRAM Bank 2

DRAM Bank 3
A Memory Performance Hog

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

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

What Does the Memory Hog Do?

Memory Request Buffer

Row decoder

Row Buffer

Row size: 8KB, cache block size: 64B

128 \((8\text{KB}/64\text{B})\) requests of T0 serviced before T1

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)

Problems due to Uncontrolled Interference

- Unfair slowdown of different threads
- Low system performance
- Vulnerability to denial of service
- Priority inversion: unable to enforce priorities/SLAs
Problems due to Uncontrolled Interference

- Unfair slowdown of different threads
- Low system performance
- Vulnerability to denial of service
- Priority inversion: unable to enforce priorities/SLAs
- Poor performance predictability (no performance isolation)

Uncontrollable, unpredictable system
Inter-Thread Interference in Memory

- Memory controllers, pins, and memory banks are shared

- Pin bandwidth is not increasing as fast as number of cores
  - Bandwidth per core reducing

- Different threads executing on different cores interfere with each other in the main memory system

- Threads delay each other by causing resource contention:
  - Bank, bus, row-buffer conflicts → reduced DRAM throughput

- Threads can also destroy each other’s DRAM bank parallelism
  - Otherwise parallel requests can become serialized
Effects of Inter-Thread Interference in DRAM

- Queueing/contention delays
  - Bank conflict, bus conflict, channel conflict, ...

- Additional delays due to DRAM constraints
  - Called “protocol overhead”
  - Examples
    - Row conflicts
    - Read-to-write and write-to-read delays

- Loss of intra-thread parallelism
  - A thread’s concurrent requests are serviced serially instead of in parallel
Problem: QoS-Unaware Memory Control

- Existing DRAM controllers are unaware of inter-thread interference in DRAM system

- They simply aim to maximize DRAM throughput
  - Thread-unaware and thread-unfair
  - No intent to service each thread’s requests in parallel
  - FR-FCFS policy: 1) row-hit first, 2) oldest first
    - Unfairly prioritizes threads with high row-buffer locality
    - Unfairly prioritizes threads that are memory intensive (many outstanding memory accesses)
Solution: QoS-Aware Memory Request 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
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
- Unable to enforce priorities or service-level agreements
- 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

Stall-Time Fairness inShared DRAM Systems

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

- 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

T0 Slowdown
T1 Slowdown
Unfairness
\( \alpha \)

1.00
1.06
1.08
1.05

Row Buffer
Row 161
Data
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
Parallelism-Aware Batch Scheduling

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

- Processors try to 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: Bank 0, Row 1
Thread A: Bank 1, Row 1

Bank access latencies of the two requests overlapped
Thread stalls for ~ONE bank access latency
**Baseline Scheduler:**

A: Compute | Stall | Stall | Compute
Bank 0 | Bank 1

B: Compute | Stall | Stall | Compute
Bank 1 | Bank 0

2 DRAM Requests

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

**Baseline Scheduler:**

2 DRAM Requests

A: Compute | Stall | Stall | Compute

Bank 0 Bank 1

2 DRAM Requests

B: Compute | Stall | Stall | Compute

Bank 1 Bank 0

**Parallelism-aware Scheduler:**

2 DRAM Requests

A: Compute | Stall | Compute

Bank 0 Bank 1

2 DRAM Requests

B: Compute | Stall | Stall | Compute

Bank 0 Bank 1

**Average stall-time:**

~1.5 bank access latencies
Parallelism-Aware Batch Scheduling (PAR-BS)

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

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

PAR-BS Components

- Request batching

- Within-batch scheduling
  - Parallelism aware
Request Batching

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

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

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

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

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

**HOW?**

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

- Ranking scheme affects system throughput and fairness

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

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

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

Shortest Stall-Time First Ranking

- Maximum number of marked requests to any bank (max-bank-load)
  - Rank thread with lower max-bank-load higher (~ low stall-time)

- Total number of marked requests (total-load)
  - Breaks ties: rank thread with lower total-load higher

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

<table>
<thead>
<tr>
<th></th>
<th>max-bank-load</th>
<th>total-load</th>
</tr>
</thead>
<tbody>
<tr>
<td>T0</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>T1</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>T2</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>T3</td>
<td>5</td>
<td>9</td>
</tr>
</tbody>
</table>

**Ranking:**

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

Baseline Scheduling Order (Arrival order)

<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>T0</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>

PAR-BS Scheduling Order

<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>T3</td>
<td>T3</td>
<td>T3</td>
</tr>
<tr>
<td>T3</td>
<td>T2</td>
<td>T2</td>
<td>T2</td>
</tr>
<tr>
<td>T2</td>
<td>T2</td>
<td>T2</td>
<td>T2</td>
</tr>
<tr>
<td>T1</td>
<td>T1</td>
<td>T1</td>
<td>T2</td>
</tr>
<tr>
<td>T1</td>
<td>T0</td>
<td>T0</td>
<td>T0</td>
</tr>
<tr>
<td>T1</td>
<td>T3</td>
<td>T3</td>
<td>T3</td>
</tr>
</tbody>
</table>

Ranking: T0 > T1 > T2 > T3

Stall times

<table>
<thead>
<tr>
<th>T0</th>
<th>T1</th>
<th>T2</th>
<th>T3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

AVG: 5 bank access latencies

Stall times

<table>
<thead>
<tr>
<th>T0</th>
<th>T1</th>
<th>T2</th>
<th>T3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

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

![Bar chart showing normalized Hmean speedup across different core counts and scheduling algorithms.](chart.png)
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:**
  - Implementation in multiple controllers needs coordination for best performance \(\rightarrow\) too frequent coordination since batching is done frequently
  - Does not always prioritize the latency-sensitive applications
Yoongu Kim, Michael Papamichael, Onur Mutlu, and Mor Harchol-Balter, "Thread Cluster Memory Scheduling: Exploiting Differences in Memory Access Behavior" 43rd International Symposium on Microarchitecture (MICRO), pages 65-76, Atlanta, GA, December 2010. Slides (pptx) (pdf)
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**

- Less memory intensive
- Higher priority
- Good for throughput

**Does not starve**

- thread C
- Not prioritized
- Does not starve
- 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

Threads in the system

Memory-non-intensive

Memory-intensive

Non-intensive cluster

Intensive cluster

Throughput

Fairness
Clustering Threads

**Step 1** Sort threads by **MPKI** (misses per kilo-instruction)

\[ \text{ClusteringThreshold} \]

**Non-intensive cluster**

**Intensive cluster**

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

**Step 2** Memory bandwidth usage \( \alpha T \) divides clusters

\( \alpha < 10\% \)

*ClusterThreshold*
TCM: Quantum-Based Operation

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: 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...

Adjusting ClusterThreshold

Better system throughput

TCM allows robust fairness-throughput tradeoff
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?
Other Ways of Handling Interference
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
Memory Channel Partitioning

- **Memory Channel Partitioning**
  - Idea: Map badly-interfering applications’ pages to different channels [Muralidhara+, MICRO’11]

- Separate data of low/high intensity and low/high row-locality applications
- Especially effective in reducing interference of threads with “medium” and “heavy” memory intensity

Muralidhara et al., “Memory Channel Partitioning,” MICRO’11.
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 showing the relationship between hardware and software in the MCP mechanism.](image-url)
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.
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**
An Alternative Approach: Source Throttling

- 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) [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
}
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

- **Disadvantages**
  - Requires interference/slowdown estimations
  - Thresholds can become difficult to optimize → throughput loss
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
Handling Interference in Parallel Applications

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

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

- Hardware/software cooperative limiter thread estimation:
  - Thread executing the most contended critical section
  - Thread that is falling behind the most in a parallel for loop
We did not cover the following slides in lecture. These are for your benefit.
ATLAS Memory Scheduler

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

A thread alternates between two states (episodes)

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

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

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

Remaining length of a memory episode?

![Diagram](image_url)
We discovered: past is excellent predictor for future

Large **attained service** ➞ Large expected **remaining service**

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

Memory episode lengths of SPEC benchmarks

Pareto distribution

The longer an episode has lasted \( \rightarrow \) The longer it will last further

Attained service correlates with remaining service

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

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

Our Approach

- Prioritize the memory episode with least-remaining-service
- Remaining service: Correlates with attained service
- Attained service: Tracked by per-thread counter
- Prioritize the memory episode with least-attained-service

Queueing Theory

- Prioritize the job with shortest-remaining-processing-time
  - Provably optimal

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

However, LAS does not consider long-term thread behavior
Prioritizing Thread 2 is more beneficial: results in very long stretches of compute episodes
Quantum-Based Attained Service of a Thread

Short-term thread behavior

Long-term thread behavior

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

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

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

End of a quantum
Each thread’s TotalAS computed as:

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

High \( \alpha \) \( \Rightarrow \) More bias towards history

Threads are ranked, favoring threads with lower TotalAS

Next quantum
Threads are serviced according to their ranking
ATLAS Scheduling Algorithm

**ATLAS**
- Adaptive per-Thread Least Attained Service

**Request prioritization order**

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

How to coordinate MCs to agree upon a consistent ranking?
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 performance
- Low complexity
- Coordination among controllers happens infrequently

**Downsides:**
- Lowest ranked threads get delayed significantly → high unfairness
Emerging Non-Volatile Memory Technologies
Aside: Non-Volatile Memory

- If memory were non-volatile...
  - there would be no need for refresh...
  - we would not lose data on power loss...
- Problem: non-volatile has traditionally been much slower than DRAM
  - Think hard disks... Even flash memory...
- Opportunity: there are some emerging memory technologies that are relatively fast, and non-volatile.
  - And, they seem more scalable than DRAM
- Question: Can we have emerging technologies as part of main memory?
Emerging Memory Technologies

- Some emerging resistive memory technologies seem more scalable than DRAM (and they are non-volatile)

- Example: Phase Change Memory
  - Data stored by changing phase of material
  - Data read by detecting material’s resistance
  - Expected to scale to 9nm (2022 [ITRS])
  - Prototyped at 20nm (Raoux+, IBM JRD 2008)
  - Expected to be denser than DRAM: can store multiple bits/cell

- But, emerging technologies have (many) shortcomings
  - Can they be enabled to replace/augment/surpass DRAM?
Emerging Resistive Memory Technologies

- **PCM**
  - Inject current to change material phase
  - Resistance determined by phase

- **STT-MRAM**
  - Inject current to change magnet polarity
  - Resistance determined by polarity

- **Memristors**
  - Inject current to change atomic structure
  - Resistance determined by atom distance
What is Phase Change Memory?

- Phase change material (chalcogenide glass) exists in two states:
  - Amorphous: Low optical reflexivity and high electrical resistivity
  - Crystalline: High optical reflexivity and low electrical resistivity

PCM is resistive memory: High resistance (0), Low resistance (1)

PCM cell can be switched between states reliably and quickly
How Does PCM Work?

- **Write**: change phase via current injection
  - **SET**: sustained current to heat cell above $T_{\text{cryst}}$
  - **RESET**: cell heated above $T_{\text{melt}}$ and quenched
- **Read**: detect phase via material resistance
  - amorphous/crystalline

![Diagram showing temperature vs time for SET and RESET processes.](image)

Photo Courtesy: Bipin Rajendran, IBM  
Slide Courtesy: Moinuddin Qureshi, IBM
Phase Change Memory: Pros and Cons

- **Pros over DRAM**
  - Better technology scaling (capacity and cost)
  - Non volatility
  - Low idle power (no refresh)

- **Cons**
  - Higher latencies: ~4-15x DRAM (especially write)
  - Higher active energy: ~2-50x DRAM (especially write)
  - Lower endurance (a cell dies after ~$10^8$ writes)

- **Challenges in enabling PCM as DRAM replacement/helper:**
  - Mitigate PCM shortcomings
  - Find the right way to place PCM in the system
PCM-based Main Memory (I)

- How should PCM-based (main) memory be organized?

- Hybrid PCM+DRAM [Qureshi+ ISCA’09, Dhiman+ DAC’09]:
  - How to partition/migrate data between PCM and DRAM
How should PCM-based (main) memory be organized?

Pure PCM main memory [Lee et al., ISCA’09, Top Picks’10]:
- How to redesign entire hierarchy (and cores) to overcome PCM shortcomings
PCM-Based Memory Systems: Research Challenges

- **Partitioning**
  - Should DRAM be a cache or main memory, or configurable?
  - What fraction? How many controllers?

- **Data allocation/movement (energy, performance, lifetime)**
  - Who manages allocation/movement?
  - What are good control algorithms?
  - How do we prevent degradation of service due to wearout?

- **Design of cache hierarchy, memory controllers, OS**
  - Mitigate PCM shortcomings, exploit PCM advantages

- **Design of PCM/DRAM chips and modules**
  - Rethink the design of PCM/DRAM with new requirements
An Initial Study: Replace DRAM with PCM

  - Surveyed prototypes from 2003-2008 (e.g. IEDM, VLSI, ISSCC)
  - Derived “average” PCM parameters for F=90nm

**Density**
- 9 - 12$F^2$ using BJT
- 1.5× DRAM

**Latency**
- 50ns Rd, 150ns Wr
- 4×, 12× DRAM

**Endurance**
- 1E+08 writes
- 1E-08× DRAM

**Energy**
- 40μA Rd, 150μA Wr
- 2×, 43× DRAM
Results: Naïve Replacement of DRAM with PCM

- Replace DRAM with PCM in a 4-core, 4MB L2 system
- PCM organized the same as DRAM: row buffers, banks, peripherals
- 1.6x delay, 2.2x energy, 500-hour average lifetime

Architecting PCM to Mitigate Shortcomings

- Idea 1: Use multiple narrow row buffers in each PCM chip
  → Reduces array reads/writes → better endurance, latency, energy

- Idea 2: Write into array at cache block or word granularity
  → Reduces unnecessary wear
Results: Architected PCM as Main Memory

- 1.2x delay, 1.0x energy, 5.6-year average lifetime
- Scaling improves energy, endurance, density

![PCM Performance](image1)

- Caveat 1: Worst-case lifetime is much shorter (no guarantees)
- Caveat 2: Intensive applications see large performance and energy hits
- Caveat 3: Optimistic PCM parameters?

![PCM Endurance](image2)
Hybrid Memory Systems

- Fast, durable
- Small, leaky, volatile, high-cost

CPU

DRAM MCtrl

Phase Change Memory (or Tech. X)

- Large, non-volatile, low-cost
- Slow, wears out, high active energy

Hardware/software manage data allocation and movement to achieve the best of multiple technologies

Yoon, Meza et al., “Row Buffer Locality Aware Caching Policies for Hybrid Memories,” ICCD 2012 Best Paper Award.
One Option: DRAM as a Cache for PCM

- PCM is main memory; DRAM caches memory rows/blocks
  - Benefits: Reduced latency on DRAM cache hit; write filtering

- Memory controller hardware manages the DRAM cache
  - Benefit: Eliminates system software overhead

Three issues:
- What data should be placed in DRAM versus kept in PCM?
- What is the granularity of data movement?
- How to design a low-cost hardware-managed DRAM cache?

Two idea directions:
- Locality-aware data placement [Yoon+ , ICCD 2012]
- Cheap tag stores and dynamic granularity [Meza+, IEEE CAL 2012]