Lecture 27: Shared Memory Management

Prof. Onur Mutlu
Carnegie Mellon University
Announcements

- Last year’s written exam posted online

- April 27 (Wednesday): Oral Exam
  - 30 minutes per person; in my office; closed book/notes
  - All content covered could be part of the exam

- May 6: Project poster session
  - HH 1112, 2-6pm

- May 10: Project report due
Readings: Shared Main Memory

Required


Recommended

Readings in Other Topics (Not Covered)

- Shared memory consistency
Readings in Other Topics (Not Covered)

- Rethinking main memory organization

- SIMD processing
Shared Main Memory Systems
Sharing in Main Memory

- **Bandwidth sharing**
  - Which thread/core to prioritize?
  - How to schedule requests?
  - How much bandwidth to allocate to each thread?

- **Capacity sharing**
  - How much memory capacity to allocate to which thread?
  - Where to map that memory? (row, bank, rank, channel)

- **Metrics for optimization**
  - System performance
  - Fairness, QoS
  - Energy/power consumption
DRAM Bank Operation

Access Address:
(Row 0, Column 0)
(Row 0, Column 1)
(Row 0, Column 85)
(Row 1, Column 0)

Row address 0

Row decoder

Columns

Rows

Row 1

Row Buffer

CONFLICT!

Column address 85

Column mux

Data
Generalized Memory Structure
Memory Controller
Inter-Thread Interference in DRAM

- 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
Inter-Thread Interference in DRAM

- 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
Consequences of Inter-Thread Interference in DRAM

- Unfair slowdown of different threads
- System performance loss
- Vulnerability to denial of service
- Inability to enforce system-level thread priorities
Why the Disparity in Slowdowns?

Multi-Core Chip

Shared DRAM Memory System

unfairness
QoS-Aware Memory Systems: Challenges

- How do we reduce inter-thread interference?
  - Improve system performance and utilization
  - Preserve the benefits of single-thread performance techniques

- How do we control inter-thread interference?
  - Provide scalable mechanisms to enable system software to enforce a variety of 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

- Fair/QoS-aware memory schedulers, interconnects, caches, arbiters

**Dumb resources:** Keep each resource free-for-all, but control access to memory system at the cores/sources

- Estimate interference/slowdown in the entire system and throttle cores that slow down others
  - Fairness via Source Throttling [Ebrahimi+, ASPLOS 2010, ISCA 2011]
  - Coordinated Prefetcher Throttling [Ebrahimi+, MICRO 2009]
Stall-Time Fair
Memory Access Scheduling

Stall-Time Fairness in Shared 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 $\text{Slowdown} = ST_{\text{shared}}/ST_{\text{alone}}$ for threads with legal requests
  - Computes unfairness $= \text{MAX Slowdown} / \text{MIN Slowdown}$

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

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

| T0: Row 0     | 1.00 |
| T1: Row 5     | 1.00 |
| T0: Row 0     | 1.00 |
| T1: Row 111   | 1.00 |
| T0: Row 0     | 1.00 |
| T0: Row 06    | 1.00 |

T0 Slowdown: 1.00
T1 Slowdown: 1.00
Unfairness: 1.00
\[ \alpha = 1.05 \]
STFM Implementation

- **Tracking** $ST_{\text{shared}}$
  - Increase $ST_{\text{shared}}$ if the thread cannot commit instructions due to an outstanding DRAM access

- **Estimating** $ST_{\text{alone}}$
  - Difficult to estimate directly because thread not running alone
    - Observation: $ST_{\text{alone}} = ST_{\text{shared}} - ST_{\text{interference}}$
    - Estimate $ST_{\text{interference}}$: Extra stall-time due to interference
  - Update $ST_{\text{interference}}$ when a thread incurs delay due to other threads
    - When a row buffer hit turns into a row-buffer conflict
      - (keep track of the row that would have been in the row buffer)
    - When a request is delayed due to bank or bus conflict
Support for System Software

- System-level thread weights (priorities)
  - OS can choose thread weights to satisfy QoS requirements
  - Larger-weight threads should be slowed down less
  - OS communicates thread weights to the memory controller
  - Controller scales each thread’s slowdown by its weight
  - Controller uses weighted slowdown used for scheduling
    - Favors threads with larger weights

- \( \alpha \): Maximum tolerable unfairness set by system software
  - Don’t need fairness? Set \( \alpha \) large.
  - Need strict fairness? Set \( \alpha \) close to 1.
  - Other values of \( \alpha \): trade off fairness and throughput
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
Parallelism-Aware Batch Scheduling

Bank Parallelism of a Thread

**Single Thread:**

2 DRAM Requests

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
Bank Parallelism Interference in DRAM

Baseline Scheduler:
2 DRAM Requests

A: Compute | Stall | Stall | Compute
Bank 0
2 DRAM Requests
Bank 1

B: Compute | Stall | Stall | Compute
Bank 1

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

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

**Baseline Scheduler:**
- 2 DRAM Requests
- A: Compute | Stall | Stall | Compute
- Bank 0
- Bank 1
- B: Compute | Stall | Stall | Compute
- Bank 1
- Bank 0

**Parallelism-aware Scheduler:**
- 2 DRAM Requests
- A: Compute | Stall | Compute
- Bank 0
- Bank 1
- B: Compute | Stall | Stall | Compute
- Bank 0
- Bank 1

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
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>max-bank-load</th>
<th>total-load</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</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>T1</td>
<td>T3</td>
</tr>
<tr>
<td>T1</td>
<td>T0</td>
<td>T0</td>
<td>T0</td>
</tr>
<tr>
<td>T2</td>
<td>T2</td>
<td>T2</td>
<td>T2</td>
</tr>
<tr>
<td>T3</td>
<td>T1</td>
<td>T0</td>
<td>T3</td>
</tr>
<tr>
<td>T1</td>
<td>T3</td>
<td>T2</td>
<td>T3</td>
</tr>
</tbody>
</table>

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>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>T1</td>
</tr>
<tr>
<td>T1</td>
<td>T0</td>
<td>T0</td>
<td>T0</td>
</tr>
<tr>
<td>T0</td>
<td>T0</td>
<td>T0</td>
<td>T0</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
  - Many more trade-offs analyzed in the paper
Unfairness on 4-, 8-, 16-core Systems

Unfairness = MAX Memory Slowdown / MIN Memory Slowdown [MICRO 2007]
System Performance (Hmean-speedup)
ATLAS Memory Scheduler

Yoongu Kim et al., “ATLAS: A Scalable and High-Performance Scheduling Algorithm for Multiple Memory Controllers,”

HPCA 2010, Best Paper Session.
Desired Properties of Memory Scheduling Algorithm

- **Maximize system performance**
  - Without starving any cores

- **Configurable by system software**
  - To enforce thread priorities and QoS/fairness policies

- **Scalable to a large number of controllers**
  - Should not require significant coordination between controllers

Multiple memory controllers

No previous scheduling algorithm satisfies all these requirements
Multiple Memory Controllers

**Single-MC system**

<table>
<thead>
<tr>
<th>Core</th>
<th>MC</th>
<th>Memory</th>
</tr>
</thead>
</table>

**Multiple-MC system**

<table>
<thead>
<tr>
<th>Core</th>
<th>MC</th>
<th>Memory</th>
</tr>
</thead>
</table>

**Difference?**

The need for coordination
Thread Ranking in Single-MC

Assume all requests are to the same bank

Memory service timeline

Thread 1's request
Thread 2's request

[Diagram showing memory service timeline with MC 1, Thread 1 (T1), Thread 2 (T2), and execution timeline with Thread 1 and Thread 2 with stall markers]

Optimal average stall time: $2T$

# of requests: Thread 1 < Thread 2
Thread 1 ➔ Shorter job

Thread ranking: Thread 1 > Thread 2
Thread 1 ➔ Assigned higher rank
Thread Ranking in Multiple-MC

Uncoordinated

<table>
<thead>
<tr>
<th>MC 1</th>
<th>T1</th>
<th>T2</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>T1</td>
<td>T1</td>
<td>T1</td>
</tr>
</tbody>
</table>

Coordination

<table>
<thead>
<tr>
<th>MC 1</th>
<th>MC 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>T2</td>
<td>T2</td>
</tr>
<tr>
<td>T1</td>
<td>T1</td>
</tr>
<tr>
<td>T1</td>
<td>T1</td>
</tr>
</tbody>
</table>

Avg. stall time: 3T

Coordination

Thread 1

GLOBAL shorter job: Thread 2

MC 1 incorrectly assigns higher rank to Thread 1

Thread 2

GLOBAL shorter job: Thread 2

MC 1 correctly assigns higher rank to Thread 2

Coordination ➜ Better scheduling decisions

Avg. stall time: 2.5T
Coordination Limits Scalability

To be scalable, coordination should:
- exchange little information
- occur infrequently
The Problem and Our Goal

**Problem:**
- Previous best memory scheduling algorithms are not scalable to many controllers
  - Not designed for multiple MCs
  - Low performance or require significant coordination

**Our Goal:**
- Fundamentally redesign the memory scheduling algorithm such that it
  - Provides high system throughput
  - Requires little or no coordination among MCs
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

Prioritize thread whose memory episode will end the soonest

Remaining length of a memory episode?

Outstanding memory requests

Time

How much longer?
Predicting Memory Episode Lengths

We discovered: past is excellent predictor for future

Outstanding memory requests

Attained service

PAST

Large **attained service** \(\rightarrow\) Large expected **remaining service**

Q: Why?
A: Memory episode lengths are **Pareto distributed**...
Favoring **least-attained-service** memory episode

\[ \Pr\{\text{Mem. episode} > x\} \]

- 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 memory episode which will **end the soonest**
Prioritize the job with shortest-remaining-processing-time

Provenably optimal

- Remaining service: Correlates with attained service

- Attained service: Tracked by per-thread counter

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

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

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

We divide time into large, fixed-length intervals: **quanta** (millions of cycles)
Quantum-Based 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 \( \alpha \rightarrow \text{More bias towards history} \)

Threads are ranked, favoring threads with lower TotalAS

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

ATLAS

- Adaptive per-Thread Least Attained Service

- Request prioritization order

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

---

How to coordinate MCs to agree upon a consistent ranking?
ATLAS Coordination Mechanism

During a quantum:
- Each MC increments the local AS of each thread

End of a quantum:
- Each MC sends local AS of each thread to centralized meta-MC
- Meta-MC accumulates local AS and calculates ranking
- Meta-MC broadcasts ranking to all MCs

→ Consistent thread ranking
## Coordination Cost in ATLAS

How costly is coordination in ATLAS?

<table>
<thead>
<tr>
<th></th>
<th>ATLAS</th>
<th>PAR-BS (previous best work [ISCA08])</th>
</tr>
</thead>
<tbody>
<tr>
<td>How often?</td>
<td>Very infrequently</td>
<td>Every quantum boundary (10 M cycles)</td>
</tr>
<tr>
<td></td>
<td>Frequently</td>
<td>Every batch boundary (thousands of cycles)</td>
</tr>
<tr>
<td>Sensitive to coordination latency?</td>
<td>Insensitive</td>
<td>Coordination latency $&lt;&lt;$ Quantum length</td>
</tr>
<tr>
<td></td>
<td>Sensitive</td>
<td>Coordination latency $\approx$ Batch length</td>
</tr>
</tbody>
</table>
Properties of ATLAS

**Goals**

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

**Properties of ATLAS**

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

**Low complexity**: Attained service requires a single counter per thread in each MC (<9K bits for 24-core, 4-MC)
ATLAS Evaluation Methodology

- **4, 8, 16, 24, 32-core systems**
  - 5 GHz processor, 128-entry instruction window
  - 512 Kbyte per-core private L2 caches

- **1, 2, 4, 8, 16-MC systems**
  - 128-entry memory request buffer
  - 4 banks, 2Kbyte row buffer
  - 40ns (200 cycles) row-hit round-trip latency
  - 80ns (400 cycles) row-conflict round-trip latency

- **Workloads**
  - Multiprogrammed SPEC CPU2006 applications
  - 32 program combinations for 4, 8, 16, 24, 32-core experiments
Comparison to Previous Scheduling Algorithms

- **FCFS, FR-FCFS** [Rixner+, ISCA00]
  - Oldest-first, row-hit first
  - **Low multi-core performance** \(\Leftarrow\) Do not distinguish between threads

- **Network Fair Queueing** [Nesbit+, MICRO06]
  - Partitions memory bandwidth equally among threads
  - **Low system performance** \(\Leftarrow\) Bank-level parallelism, locality not exploited

- **Stall-time Fair Memory Scheduler** [Mutlu+, MICRO07]
  - Balances thread slowdowns relative to when run alone
  - **High coordination costs** \(\Leftarrow\) Requires heavy cycle-by-cycle coordination

- **Parallelism-Aware Batch Scheduler** [Mutlu+, ISCA08]
  - Batches requests and performs thread ranking to preserve bank-level parallelism
  - **High coordination costs** \(\Leftarrow\) Batch duration is very short
System Throughput: 24-Core System

System throughput = $\sum$ 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 enforces system priorities, or thread weights
- Linear relationship between thread weight and speedup

\[
Total AS_i = \alpha Total AS_{i-1} + \frac{(1 - \alpha)}{\text{thread weight}} AS_i
\]

Figure 14. Evaluation of ATLAS vs. PAR-BS and STFM with different thread weights
STFM, PAR-BS, ATLAS Comparison

- What are the relative advantages and disadvantages of each in different dimensions?
  - System Performance
  - System Fairness
  - QoS Guarantees
  - Scalability to Controllers
  - Hardware Cost

- Can we do better than all of these?
System Performance vs. Fairness
Achieving the Best of Both Worlds


- Idea: Dynamically cluster threads based on their memory intensity and employ a different scheduling algorithm in each cluster
  - Low intensity cluster: Prioritize over high intensity threads \(\rightarrow\) maximize system throughput
  - High intensity cluster: Shuffle the priority order of threads periodically \(\rightarrow\) maximize system fairness
Thread Cluster Memory Scheduling

![Graph showing maximum slowdown vs weighted speedup for different scheduling methods.]

- **FRFCFS**
- **STFM**
- **PAR_BS**
- **ATLAS**
- **TCM**
Thread Cluster Memory Scheduling

Motivation

• Memory is a shared resource

• Threads’ requests contend for memory
  – Degradation in single thread performance
  – Can even lead to starvation

• How to schedule memory requests to increase both system throughput and fairness?
Previous Scheduling Algorithms are Biased

No previous memory scheduling algorithm provides both the best fairness and system throughput.
Why do Previous Algorithms Fail?

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

- Good for throughput
- less memory intensive
- thread A
- thread B
- thread C
- starvation ➔ unfairness

**Fairness biased approach**
Take turns accessing memory

- Does not starve
- higher priority
- thread C
- thread A
- thread B
- not prioritized ➔ reduced throughput

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

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

### For Fairness
- Unfairness caused by memory-intensive being prioritized over each other
  - Shuffle threads
- Memory-intensive threads have different vulnerability to interference
  - Shuffle asymmetrically
Overview: Thread Cluster Memory Scheduling

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

![Diagram showing thread clusters and priority](image-url)
1. Clustering
Clustering Threads

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

- **Non-intensive cluster**
  - \( \alpha T \)
- **Intensive cluster**
  - \( \alpha < 10\% \)
  - **ClusterThreshold**

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

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

2. Between Clusters

3. Non-Intensive Cluster

Throughput
Non-Intensive Cluster

Prioritize threads according to MPKI

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

1. Clustering

2. Between Clusters

3. Non-Intensive Cluster
   - Throughput
   - Fairness

4. Intensive Cluster
Periodically shuffle the priority of threads

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

Case Study: Two intensive threads contending

1. random-access
2. streaming

Which is slowed down more easily?

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

random-access  

streaming

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

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

Vulnerable to interference
Niceness

How to quantify difference between threads?

Bank-level parallelism
Vulnerability to interference

Row-buffer locality
Causes interference

Niceness

High

Low
Shuffling: Round-Robin vs. Niceness-Aware

1. **Round-Robin** shuffling

2. **Niceness-Aware** shuffling

**What can go wrong?**

**GOOD:** Each thread prioritized once

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

1. **Round-Robin** shuffling

What can go wrong?

BAD: Nice threads receive lots of interference

GOOD: Each thread prioritized once

2. **Niceness-Aware** shuffling

Most prioritized

BAD: Nice threads receive lots of interference

ShuffleInterval

Priority

Time

Nice thread

Least nice thread

A

B

C

D
Shuffling: Round-Robin vs. Niceness-Aware

1. **Round-Robin** shuffling

2. **Niceness-Aware** shuffling

**GOOD**: Each thread prioritized once

Priority

ShuffleInterval

Most prioritized

| A | B | C | D |

| D | C | B | A |

| Nice thread |

| Least nice thread |

Time
Shuffling: Round-Robin vs. Niceness-Aware

1. **Round-Robin** shuffling

2. **Niceness-Aware** shuffling

**GOOD:** Each thread prioritized once

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

---

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

**ShuffleInterval**
TCM Outline

1. Clustering
2. Between Clusters
3. Non-Intensive Cluster
4. Intensive Cluster
Quantum-Based Operation

During quantum:
- Monitor thread behavior
  1. Memory intensity
  2. Bank-level parallelism
  3. Row-buffer locality
- Shuffle interval
  (~1K cycles)

Beginning of quantum:
- Perform clustering
- Compute niceness of intensive threads
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
### Implementation Costs

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

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

- No computation is on the critical path
Metrics & Methodology

• **Metrics**

    **System throughput**
    \[
    \text{Weighted Speedup} = \sum_i \frac{IPC_i^{\text{shared}}}{IPC_i^{\text{alone}}}
    \]

    **Unfairness**
    \[
    \text{Maximum Slowdown} = \max_i \frac{IPC_i^{\text{alone}}}{IPC_i^{\text{shared}}}
    \]

• **Methodology**

    – Core model
      • 4 GHz processor, 128-entry instruction window
      • 512 KB/core L2 cache
    – Memory model: DDR2
    – 96 multiprogrammed SPEC CPU2006 workloads
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
Results: Fairness vs. Throughput

Averaged over 96 workloads

TCM provides best fairness and system throughput
Results: Fairness-Throughput Tradeoff

When configuration parameter is varied...

Better fairness

Adjusting ClusterThreshold

Better system throughput

TCM allows robust fairness-throughput tradeoff
Operating System Support

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

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

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

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

• **TCM provides the best system throughput and fairness**
Designing QoS-Aware Memory Systems: Approaches

- **Smart resources:** Design each shared resource to have a configurable interference control/reduction mechanism
  - Fair/QoS-aware memory schedulers, interconnects, caches, arbiters

- **Dumb resources:** Keep each resource free-for-all, but control access to memory system at the cores/sources
  - Estimate interference/slowdown in the entire system and throttle cores that slow down others
    - Fairness via Source Throttling [Ebrahimi+, ASPLOS 2010, ISCA 2011]
    - Coordinated Prefetcher Throttling [Ebrahimi+, MICRO 2009]
Fairness via Source Throttling

Eiman Ebrahimi et al., “Fairness via Source Throttling: A Configurable and High-Performance Fairness Substrate for Multi-Core Memory Systems,”
ASPLOS 2010, Best Paper Award.
Many Shared Resources

Shared Cache

Memory Controller

DRAM Bank 0
DRAM Bank 1
DRAM Bank 2
...
DRAM Bank K

Shared Memory Resources

On-chip

Off-chip

Chip Boundary
Motivation for Source Throttling

- Partitioning (fairness/QoS) mechanisms in each resource might be difficult to get right (initially)

- Independent partitioning mechanisms in caches, interconnect, and memory can contradict each other

- Approaches that coordinate interaction among techniques for different resources require complex implementations

Our Goal: Enable fair sharing of the entire memory system by dynamically detecting and controlling interference in a coordinated manner
An Alternative Approach

- Manage inter-thread interference at the cores, not at the shared resources

- **Dynamically estimate unfairness** in the memory system
- Feed back this information into a controller
- **Throttle cores’ memory access rates** accordingly
  - Whom to throttle and by how much depends on performance target (throughput, fairness, per-thread QoS, etc)
  - E.g., if unfairness > system-software-specified target then **throttle down** core causing unfairness & **throttle up** core that was unfairly treated
Intensive application A generates many requests and causes long stall times for less intensive application B.

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

- Two components (interval-based)

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

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

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

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

FST

- Runtime Unfairness Evaluation
- Unfairness Estimate
  - App-slowest
  - App-interfering
- Dynamic Request Throttling

Interval 1  Interval 2  Interval 3

Time

Slowdown Estimation
Fairness via Source Throttling (FST)

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

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

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

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

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

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

- \[
T_i^{\text{Alone}} = T_i^{\text{Shared}} - T_i^{\text{Excess}}
\]
Tracking Inter-Core Interference

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

Interference induced row conflict

Row Conflict

Interference per core bit vector

Queue of requests to bank 2

Row A

Bank 0

Bank 1

Bank 2

... Bank 7

Core 0

Row A

Core 1

Row B

Row A

Shadow Row Address Register (SRAR) Core 1

Shadow Row Address Register (SRAR) Core 0

Row A

Row B
Tracking Inter-Core Interference

- Core 0, Core 1, Core 2, Core 3
- Shared Cache
- Memory Controller
- Bank 0, Bank 1, Bank 2, ..., Bank 7

Cycle Count
- $T+3$
- $T+2$
- $T+1$
- $T$

FST hardware
- Core # 0 1 2 3
- Excess Cycles Counters per core

$T_i$ = $T_{i, \text{Shared}}$ - $T_{i, \text{Excess}}$

Interference per core bit vector
- $1$ $0$ $1$ $0$
- $1$
- $0$
Fairness via Source Throttling (FST)

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

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

- To identify App-interfering, for each core $i$
  - FST separately tracks interference caused by each core $j$ ($j \neq i$)

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

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

Row with largest count determines App-interfering

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

FST

Runtime Unfairness Evaluation

Unfairness Estimate

App-slowest

App-interfering

Dynamic Request Throttling

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

if (Unfairness Estimate >Target)
{
  1-Throttle down App-interfering
  2-Throttle up App-slowest
}
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><strong>10%</strong></td>
<td><strong>12</strong></td>
<td><strong>Once every 10 cycles</strong></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
# FST at Work

## Runtime Unfairness Evaluation

- **System software fairness goal:** 1.4

### Dynamic Request Throttling

<table>
<thead>
<tr>
<th>Interval</th>
<th>Core 0</th>
<th>Core 1</th>
<th>Core 2</th>
<th>Core 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>$i$</td>
<td>50%</td>
<td>100%</td>
<td>10%</td>
<td>100%</td>
</tr>
<tr>
<td>$i+1$</td>
<td>25%</td>
<td>100%</td>
<td>25%</td>
<td>100%</td>
</tr>
<tr>
<td>$i+2$</td>
<td>25%</td>
<td>50%</td>
<td>50%</td>
<td>100%</td>
</tr>
</tbody>
</table>

### Throttling Levels

- Throttle down
- Throttle up
System Software Support

- **Different fairness objectives** can be configured by system software
  - Estimated Unfairness > Target Unfairness
  - Estimated Max Slowdown > Target Max Slowdown
  - Estimated Slowdown(i) > Target Slowdown(i)

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

- Total storage cost required for 4 cores is ~12KB

- FST does not require any structures or logic that are on the processor’s critical path
FST Evaluation Methodology

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

![Graph showing system unfairness results with different fairness methods]

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

System Unfairness

- grom+art+astar+h264
- art+games+Gems+omnet+apis+vortex
- art+leslie+games+grom
- art+astar+leslie+crafty
- lucas+ammp+vortex+calculix
- ibm+Gems+astar+mesa
- mgrid+parser+soplex+perl
- gcc06+xalan+ibm+cactus
- gmean

44.4% reduction in unfairness
36% reduction in unfairness
FST: System Performance Results

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

System Perf. Normalized to No Fairness

- grom+art+astar+h264
- art+games+Gems+h264
- art+leslie+games+grom
- art+mic+Vortex+calculix
- lucas+ammp+xalanc+grom
- ibm+Gems+astar+mega
- mgrid+parser+soplex+perl
- gcc+6+xalanc+ibm+cactus
- gmean

14% 25.6%
FST Summary

- **Fairness via Source Throttling (FST)** is a *new* fair and high-performance shared resource management approach for CMPs.

- **Dynamically monitors unfairness** and throttles down sources of interfering memory requests.

- Reduces the need for multiple per-resource interference reduction/control techniques.

- Improves both *system fairness* and *performance*.

- Incorporates *thread weights* and enables different fairness objectives.
Designing QoS-Aware Memory Systems: Approaches

- **Smart resources:** Design each shared resource to have a configurable interference control/reduction mechanism
  - Fair/QoS-aware memory schedulers, interconnects, caches, arbiters

- **Dumb resources:** Keep each resource free-for-all, but control access to memory system at the cores/sources
  - Estimate interference/slowdown in the entire system and throttle cores that slow down others
    - Fairness via Source Throttling [Ebrahimi+, ASPLOS 2010, ISCA 2011]
    - Coordinated Prefetcher Throttling [Ebrahimi+, MICRO 2009]
Smart Resources vs. Source Throttling

- Advantages of “smart resources”
  - Each resource is designed to be as efficient as possible → more efficient design using custom techniques for each resource
  - No need for estimating interference across the entire system (to feed a throttling algorithm).
  - Does not lose throughput by possibly overthrottling

- Advantages of source throttling
  - Prevents overloading of any or all resources (if employed well)
  - Can keep each resource simple; no need to redesign each resource
  - Provides prioritization of threads in the entire memory system; instead of per resource
  - Eliminates conflicting decision making between resources
Other Ways of Reducing (DRAM) Interference

- DRAM bank/channel partitioning among threads
- Interference-aware address mapping/remapping
- Core/request throttling: How?
- Interference-aware thread scheduling: How?
- Better/Interference-aware caching
- Interference-aware scheduling in the interconnect
- Randomized address mapping
- DRAM architecture/microarchitecture changes?

These are general techniques that can be used to improve

- System throughput
- QoS/fairness
- Power/energy consumption?
DRAM Partitioning Among Threads

- **Idea:** Map competing threads’ physical pages to different channels (or banks/ranks)
- Essentially, physically partition DRAM channels/banks/ranks among threads
  - Can be static or dynamic
- A research topic (no known publications)

**Advantages**
- Reduces interference
- No/little need for hardware changes

**Disadvantages**
- Causes fragmentation in physical memory → possibly more page faults
- Can reduce each thread’s bank parallelism/locality
- Scalability?
Core/Request 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 scheduling algorithm
+ Can be a general way of handling shared resource contention

**Disadvantages**

- Requires interference/slowdown estimations
- Thresholds can become difficult to optimize
Research Topics in Main Memory Management

- Abundant
- Interference reduction via different techniques
- Distributed memory controller management
- Co-design with on-chip interconnects and caches
- Reducing waste, minimizing energy, minimizing cost
- Enabling new memory technologies
  - Die stacking
  - Non-volatile memory
  - Latency tolerance
- You can come up with great solutions that will significantly impact computing industry