18-447
Computer Architecture
Lecture 24: Memory Scheduling

Prof. Onur Mutlu
Presented by Justin Meza
Carnegie Mellon University
Spring 2014, 3/31/2014
Last Two Lectures

- Main Memory
  - Organization and DRAM Operation
  - Memory Controllers

- DRAM Design and Enhancements
  - More Detailed DRAM Design: Subarrays
  - RowClone and In-DRAM Computation
  - Tiered-Latency DRAM

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

- Row Buffer Management Policies
- Memory Interference (and Techniques to Manage It)
  - With a focus on Memory Request Scheduling
Review: DRAM Scheduling Policies (I)

- **FCFS** (first come first served)
  - Oldest request first

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

Goal: Maximize row buffer hit rate → maximize DRAM throughput

- Actually, scheduling is done at the **command level**
  - Column commands (read/write) prioritized over row commands (activate/precharge)
  - Within each group, older commands prioritized over younger ones
A scheduling policy is essentially a prioritization order

Prioritization can be based on
- Request age
- Row buffer hit/miss status
- Request type (prefetch, read, write)
- Requestor type (load miss or store miss)
- Request criticality
  - Oldest miss in the core?
  - How many instructions in core are dependent on it?
Row Buffer Management Policies

- Open row
  - Keep the row open after an access
    + Next access might need the same row → row hit
    -- Next access might need a different row → row conflict, wasted energy

- Closed row
  - Close the row after an access (if no other requests already in the request buffer need the same row)
    + Next access might need a different row → avoid a row conflict
    -- Next access might need the same row → extra activate latency

- Adaptive policies
  - Predict whether or not the next access to the bank will be to the same row
# Open vs. Closed Row Policies

<table>
<thead>
<tr>
<th>Policy</th>
<th>First access</th>
<th>Next access</th>
<th>Commands needed for next access</th>
</tr>
</thead>
<tbody>
<tr>
<td>Open row</td>
<td>Row 0</td>
<td>Row 0 (row hit)</td>
<td>Read</td>
</tr>
<tr>
<td>Open row</td>
<td>Row 0</td>
<td>Row 1 (row conflict)</td>
<td>Precharge + Activate Row 1 + Read</td>
</tr>
<tr>
<td>Closed row</td>
<td>Row 0</td>
<td>Row 0 – access in request buffer (row hit)</td>
<td>Read</td>
</tr>
<tr>
<td>Closed row</td>
<td>Row 0</td>
<td>Row 0 – access not in request buffer (row closed)</td>
<td>Activate Row 0 + Read + Precharge</td>
</tr>
<tr>
<td>Closed row</td>
<td>Row 0</td>
<td>Row 1 (row closed)</td>
<td>Activate Row 1 + Read + Precharge</td>
</tr>
</tbody>
</table>
Memory Interference and Scheduling in Multi-Core Systems
Review: A Modern DRAM Controller
Review: 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
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

- **Nvidia Fermi**
  - 448 “cores”

- **Sun Niagara II**
  - 8 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

Uncontrolled Interference: An Example

Multi-Core Chip

Shared DRAM Memory System

Unfairness
A Memory Performance Hog

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

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

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

What Does the Memory Hog Do?

T0: Row 0
T0: Row 6
T0: Row 101
T1: Row 101
T1: Row 16

Memory Request Buffer

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

Row decoder

Row Buffer

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

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

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

- $ST_{\text{shared}}$: DRAM-related stall-time when the thread runs with other threads.
- $ST_{\text{alone}}$: DRAM-related stall-time when the thread runs alone.

**Memory-slowdown** = $\frac{ST_{\text{shared}}}{ST_{\text{alone}}}$

- Relative increase in stall-time

*Stall-Time Fair Memory scheduler (STFM)* aims to equalize Memory-slowdown for interfering threads, without sacrificing performance.

- Considers inherent DRAM performance of each thread.
- Aims to allow proportional progress of threads.
STFM Scheduling Algorithm [MICRO’07]

- For each thread, the DRAM controller
  - Tracks $ST_{shared}$
  - Estimates $ST_{alone}$

- Each cycle, the DRAM controller
  - Computes Slowdown = $ST_{shared}/ST_{alone}$ for threads with legal requests
  - Computes 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?

<table>
<thead>
<tr>
<th></th>
<th>T0: Row 0</th>
<th>T1: Row 5</th>
<th>T0: Row 0</th>
<th>T1: Row 111</th>
<th>T0: Row 0</th>
<th>T0: Row 0</th>
<th>T0: Row 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>T0 Slowdown</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>T1 Slowdown</td>
<td>1.06</td>
<td>1.06</td>
<td>1.06</td>
<td>1.06</td>
<td>1.06</td>
<td>1.06</td>
<td>1.06</td>
</tr>
<tr>
<td>Unfairness</td>
<td>1.08</td>
<td>1.08</td>
<td>1.08</td>
<td>1.08</td>
<td>1.08</td>
<td>1.08</td>
<td>1.08</td>
</tr>
</tbody>
</table>

Row Buffer

Data

Row 161
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

- 2 DRAM Requests

**Bank access latencies of the two requests overlapped**

**Thread stalls for ~ONE bank access latency**
Bank Parallelism Interference in DRAM

Baseline Scheduler:

2 DRAM Requests

A: Compute | Stall | Stall | Compute
Bank 0

2 DRAM Requests

B: Compute | Stall | Stall | Compute
Bank 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

2 DRAM Requests

B: Compute | Stall | Stall | Compute

Bank 1

Bank 0

2 DRAM Requests

**Parallelism-aware Scheduler:**

A: Compute | Stall | Compute

Bank 0

Bank 1

2 DRAM Requests

B: Compute | Stall | Stall | Compute

Bank 0

Bank 1

Saved Cycles

Average stall-time: ~1.5 bank access latencies

Baseline Scheduler:

Thread A: Bank 0, Row 1

Thread B: Bank 1, Row 99

Parallelism-aware Scheduler:

Thread B: Bank 0, Row 99

Thread A: Bank 1, Row 1
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>T1</td>
<td>T0</td>
<td>T2</td>
<td>T3</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>

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

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

Baseline Scheduling Order (Arrival order)

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

PAR-BS Scheduling Order

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

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
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 → too frequent coordination since batching is done frequently
- Does not always prioritize the latency-sensitive applications