Course Logistics (I)

- 27 registered students fulfilled the "talk slot assignment" requirements
  - Requirement 1: Submit HW0
  - Requirement 2: Explicitly provide paper preferences

- We have assigned papers based on their preferences
  - Everyone gets one of their top choices

- We have assigned mentors based on paper topic

- We will provide these assignments later this week

- If you drop the course, do it soon and let us know ASAP
Course Logistics (II)

- We will have 9 sessions of presentations + 1 extra wrap-up session at the end

- 3 presentations in each of the 9 sessions
  - Max 35 minutes total for each presentation+discussion
  - We will take the entire 2 hours in each meeting

- Each presentation
  - One student presents one paper and leads discussion
  - Max 25 minute summary+analysis
  - Max 10 minute discussion+brainstorming+feedback
  - Should follow the suggested guidelines
Algorithm for Presentation Preparation

- Study Lecture 1 again for presentation guidelines
- Read and analyze your paper thoroughly
  - Discuss with anyone you wish + use any resources
- Prepare a draft presentation based on guidelines
- Meet mentor(s) and get feedback
  - Revise the presentation and delivery
- Meet mentor(s) again and get further feedback
  - Revise the presentation and delivery
- Meetings are mandatory – you have to schedule them with your assigned mentor(s). We may suggest meeting times.
- Practice, practice, practice
Example Paper Presentations
Learning by Example

- A great way of learning
- We already did one example last time
  - Memory Channel Partitioning
- We will do at least one more today
Structure of the Presentation

- Background, Problem & Goal
- Novelty
- Key Approach and Ideas
- Mechanisms (in some detail)
- Key Results: Methodology and Evaluation
- Summary
- Strengths
- Weaknesses
- Thoughts and Ideas
- Takeaways
- Open Discussion
Background, Problem & Goal
Novelty
Key Approach and Ideas
Mechanisms (in some detail)
Key Results: Methodology and Evaluation
Summary
Strengths
Weaknesses
Thoughts and Ideas
Takeaways
Open Discussion
Example Paper Presentation
Let’s Review This Paper

- Vivek Seshadri, Yoongu Kim, Chris Fallin, Donghyuk Lee, Rachata Ausavarungnirun, Gennady Pekhimenko, Yixin Luo, Onur Mutlu, Michael A. Kozuch, Phillip B. Gibbons, and Todd C. Mowry,

"RowClone: Fast and Energy-Efficient In-DRAM Bulk Data Copy and Initialization"

Proceedings of the 46th International Symposium on Microarchitecture (MICRO), Davis, CA, December 2013. [Slides (pptx) (pdf)] [Lightning Session Slides (pptx) (pdf)] [Poster (pptx) (pdf)]
RowClone

Fast and Energy-Efficient In-DRAM Bulk Data Copy and Initialization

Vivek Seshadri

Background, Problem & Goal
Limited Bandwidth

High Energy
Goal: Reduce Memory Bandwidth Demand

Reduce unnecessary data movement
Bulk Data Copy and Initialization

Bulk Data Copy

Bulk Data Initialization

\[ \text{src} \rightarrow \text{dst} \]

\[ \text{val} \rightarrow \text{dst} \]
Bulk Data Copy and Initialization

The Impact of Architectural Trends on Operating System Performance
Mendel Rosenblum, Edouard Bugnion, Stephen Alan Herrod, Emmett Witchel, and Anoop Gupta

Hardware Support for Bulk Data Movement in Server Platforms
Li Zhao†, Ravi Iyer‡ Srijani Makineni‡, Laxmi Bhuyan† and Don Newell‡
†Department of Computer Science and Engineering, University of California, Riverside, CA 92521
Email: {zhao, bhuyan}@cs.ucr.edu
‡Communications Technology Lab, Intel Corporation

Architecture Support for Improving Bulk Memory Copying and Initialization Performance
Xiaowei Jiang, Yan Solihin
Dept. of Electrical and Computer Engineering
North Carolina State University
Raleigh, USA

Li Zhao, Ravishankar Iyer
Intel Labs
Intel Corporation
Hillsboro, USA
Bulk Data Copy and Initialization

*memmove* & *memcpy*: 5% cycles in Google’s datacenter [Kanev+ ISCA’15]

Forking  Zero initialization (e.g., security)  Checkpointing

VM Cloning  Deduplication  Page Migration

Many more
Shortcomings of Today’s Systems

1) High latency
2) High bandwidth utilization
3) Cache pollution
4) Unwanted data movement

1046ns, 3.6uJ (for 4KB page copy via DMA)
Novelty, Key Approach, and Ideas
RowClone: In-Memory Copy

1) Low latency
2) Low bandwidth utilization
3) No cache pollution
4) No unwanted data movement

1046ns, 3.6uJ → 90ns, 0.04uJ
RowClone: In-DRAM Row Copy

Idea: Two consecutive ACTivates
Negligible HW cost

Step 1: Activate row A
Step 2: Activate row B

DRAM subarray

Row Buffer (4 Kbytes)

Transfer row

4 Kbytes

8 bits

Data Bus
Mechanisms (in some detail)
DRAM Chip Organization

- Chip I/O
- Memory Channel
- Bank
- Subarray
- Bank I/O
- Row of DRAM Cells
- Row Buffer
RowClone Types

- Intra-subarray RowClone (row granularity)
  - Fast Parallel Mode (FPM)

- Inter-bank RowClone (byte granularity)
  - Pipelined Serial Mode (PSM)

- Inter-subarray RowClone
RowClone: Fast Parallel Mode (FPM)

1. Source row to row buffer
2. Row buffer to destination row
RowClone: Intra-Subarray (I)

$V_{DD}/2 \pm \delta$

Data gets copied

Sensing Amplifier (Row Buffer)

Amplify the difference

$V_{DD}/2$
RowClone: Intra-Subarray (II)

1. **Activate** src row (copy data from src to row buffer)

2. **Activate** dst row (disconnect src from row buffer, connect dst – copy data from row buffer to dst)
Fast Parallel Mode: Benefits

Bulk Data Copy

Latency: 11x
1046ns to 90ns

Energy: 74x
3600nJ to 40nJ

No bandwidth consumption
Very little changes to the DRAM chip
Fast Parallel Mode: Constraints

- Location of source/destination
  - Both should be in the same subarray

- Size of the copy
  - Copies *all* the data from source row to destination
RowClone: Inter-Bank

Overlap the latency of the read and the write

1.9X latency reduction, 3.2X energy reduction
Generalized RowClone

0.01% area cost

Inter Subarray Copy
(Use Inter-Bank Copy Twice)

Inter Bank Copy
(Pipelined Internal RD/WR)

Intra Subarray Copy
(2 ACTs)
RowClone: Fast Row Initialization

Fix a row at Zero
(0.5% loss in capacity)
RowClone: Bulk Initialization

- Initialization with arbitrary data
  - Initialize one row
  - Copy the data to other rows

- Zero initialization (most common)
  - Reserve a row in each subarray (always zero)
  - Copy data from reserved row (FPM mode)
  - 6.0x lower latency, 41.5x lower DRAM energy
  - 0.2% loss in capacity
RowClone: Latency & Energy Benefits

Latency Reduction

<table>
<thead>
<tr>
<th></th>
<th>Intra-Subarray</th>
<th>Inter-Bank</th>
<th>Inter-Subarray</th>
<th>Intra-Subarray</th>
</tr>
</thead>
<tbody>
<tr>
<td>Copy</td>
<td>11.6x</td>
<td>1.9x</td>
<td>1.0x</td>
<td>6.0x</td>
</tr>
</tbody>
</table>

Energy Reduction

<table>
<thead>
<tr>
<th></th>
<th>Intra-Subarray</th>
<th>Inter-Bank</th>
<th>Inter-Subarray</th>
<th>Intra-Subarray</th>
</tr>
</thead>
<tbody>
<tr>
<td>Copy</td>
<td>74.4x</td>
<td>3.2x</td>
<td>1.5x</td>
<td>41.5x</td>
</tr>
</tbody>
</table>
RowClone: Latency & Energy Benefits

**Latency Reduction**
- Intra-Subarray: 11.6x
- Inter-Bank: 1.9x
- Inter-Subarray: 6.0x

**Energy Reduction**
- Intra-Subarray: 74.4x
- Inter-Bank: 3.2x
- Inter-Subarray: 41.5x

Very low cost: 0.01% increase in die area
System Design to Enable RowClone
End-to-End System Design

- Application
- Operating System
- ISA
- Microarchitecture
- DRAM (RowClone)

How to communicate occurrences of bulk copy/initialization across layers?

How to ensure cache coherence?

How to maximize latency and energy savings?

How to handle data reuse?
1. Hardware/Software Interface

- Two new instructions
  - memcopy and meminit
  - Similar instructions present in existing ISAs

- Microarchitecture Implementation
  - Checks if instructions can be sped up by RowClone
  - Export instructions to the memory controller
2. Managing Cache Coherence

- RowClone modifies data in memory
  - Need to maintain coherence of cached data

- Similar to DMA
  - Source and destination in memory
  - Can leverage hardware support for DMA

- Additional optimizations
3. Maximizing Use of the Fast Parallel Mode

- Make operating system subarray-aware

- Primitives amenable to use of FPM
  - **Copy-on-Write**
    - Allocate destination in same subarray as source
    - Use FPM to copy
  - **Bulk Zeroing**
    - Use FPM to copy data from reserved zero row
4. Handling Data Reuse After Zeroing

- Data reuse after zero initialization
  - Phase 1: OS zeroes out the page
  - Phase 2: Application uses cachelines of the page

- RowClone
  - Avoids misses in phase 1
  - But incurs misses in phase 2

- RowClone-Zero-Insert (RowClone-ZI)
  - Insert clean zero cachelines
Key Results:
Methodology and Evaluation
Methodology

- Out-of-order multi-core simulator
- 1MB/core last-level cache
- Cycle-accurate DDR3 DRAM simulator
- 6 Copy/Initialization intensive applications + SPEC CPU2006 for multi-core

Performance
- Instruction throughput for single-core
- Weighted Speedup for multi-core
Copy/Initialization Intensive Applications

- **System** **bootup** (Booting the Debian OS)
- **Compile** (GNU C compiler – executing `cc1`)
- **Forkbench** (A fork microbenchmark)
- **Memcached** (Inserting a large number of objects)
- **MySql** (Loading a database)
- **Shell** script (find with `ls` on each subdirectory)
Copy and Initialization in Workloads

![Graph showing fractions and workloads.]

- Fraction of Memory Traffic
- Zero, Copy, Write, Read

Workloads Include:
- bootup
- compile
- forkbench
- mcached
- mysql
- shell
Single-Core – Performance and Energy

Compared to Baseline

- IPC Improvement
- Memory Energy Reduction

<table>
<thead>
<tr>
<th>Activity</th>
<th>IPC Improvement</th>
<th>Memory Energy Reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>bootup</td>
<td>10%</td>
<td>40%</td>
</tr>
<tr>
<td>compile</td>
<td>5%</td>
<td>30%</td>
</tr>
<tr>
<td>forkbench</td>
<td>40%</td>
<td>70%</td>
</tr>
<tr>
<td>mcached</td>
<td>0%</td>
<td>10%</td>
</tr>
<tr>
<td>mysql</td>
<td>5%</td>
<td>20%</td>
</tr>
<tr>
<td>shell</td>
<td>60%</td>
<td>60%</td>
</tr>
</tbody>
</table>
Single-Core – Performance and Energy

Improvements correlate with fraction of memory traffic due to copy-initialization
Multi-Core Systems

- Reduced bandwidth consumption benefits all applications.

- Run copy/initialization intensive applications with memory intensive SPEC applications.

- Half the cores run copy/initialization intensive applications. Remaining half run SPEC applications.
Multi-Core Results: Summary

- System Performance
- Memory Energy Efficiency

Improvement over Baseline

- 2-Core
- 4-Core
- 8-Core
Multi-Core Results: Summary

- System Performance
- Memory Energy Efficiency

Performance improvement increases with increasing core count
Multi-Core Results: Summary

- System Performance
- Memory Energy Efficiency

Consistent improvement in energy/instruction
Summary
Executive Summary

- Bulk data copy and initialization
  - Unnecessarily move data on the memory channel
  - Degrade system performance and energy efficiency
- **RowClone** – perform copy in DRAM with low cost
  - Uses row buffer to copy large quantity of data
  - **Source row → row buffer → destination row**
  - 11X lower latency and 74X lower energy for a bulk copy
- Accelerate Copy-on-Write and Bulk Zeroing
  - Forking, checkpointing, zeroing (security), VM cloning
- Improves performance and energy efficiency at low cost
  - 27% and 17% for 8-core systems (0.01% DRAM chip area)
Strengths
Strengths of the Paper

- Simple, novel mechanism to solve an important problem
- Effective and low hardware overhead
- Intuitive idea!
- Greatly improves performance and efficiency (assuming data is mapped nicely)
- Seems like a clear win for data initialization (without mapping requirements)
- Makes software designer’s life easier
  - If copies are 10x-100x cheaper, how to design software?

- Paper tackles many low-level and system-level issues
- Well-written, insightful paper
Weaknesses
Weaknesses

- Requires data to be mapped in the same subarray to deliver the largest benefits
  - Helps less if data movement is not within a subarray
  - Does not help if data movement is across DRAM channels

- Inter-subarray copy is very inefficient

- Causes many changes in the system stack
  - End-to-end design spans applications to circuits
  - Software-hardware cooperative solution might not always be easy to adopt

- Cache coherence and data reuse cause real overheads

- Evaluation is done solely in simulation

- Evaluation does not consider multi-chip systems

- Are these the best workloads to evaluate?
Recall: Try to Avoid Rat Holes

Performance Analysis Rat Holes

- Workload
- Metrics
- Configuration
- Details

Source: https://www.cse.wustl.edu/~jain/iucee/ftp/k_10adp.pdf
Thoughts and Ideas
Extensions

- Can this be improved to do faster inter-subarray copy?
  - Yes, see the LISA paper [Chang et al., HPCA 2016]

- Can this be extended to move data at smaller granularities?

- Can we have more efficient solutions to
  - Cache coherence (minimize overhead)
  - Data reuse after copy and initialization

- Can this idea be evaluated on a real system? How?

- Can similar ideas and DRAM properties be used to perform computation on data?
  - Yes, see the Ambit paper [Seshadri et al., MICRO 2017]
LISA: Fast Inter-Subarray Data Movement

Kevin K. Chang, Prashant J. Nair, Saugata Ghose, Donghyuk Lee, Moinuddin K. Qureshi, and Onur Mutlu,
"Low-Cost Inter-Linked Subarrays (LISA): Enabling Fast Inter-Subarray Data Movement in DRAM"
[Slides (pptx) (pdf)]
[Source Code]

Low-Cost Inter-Linked Subarrays (LISA): Enabling Fast Inter-Subarray Data Movement in DRAM

Kevin K. Chang†, Prashant J. Nair*, Donghyuk Lee†, Saugata Ghose†, Moinuddin K. Qureshi*, and Onur Mutlu†
†Carnegie Mellon University  *Georgia Institute of Technology

SAFARI
In-DRAM Bulk AND/OR

- Vivek Seshadri, Kevin Hsieh, Amirali Boroumand, Donghyuk Lee, Michael A. Kozuch, Onur Mutlu, Phillip B. Gibbons, and Todd C. Mowry,

"Fast Bulk Bitwise AND and OR in DRAM"


Fast Bulk Bitwise AND and OR in DRAM

Vivek Seshadri*, Kevin Hsieh*, Amirali Boroumand*, Donghyuk Lee*, Michael A. Kozuch†, Onur Mutlu*, Phillip B. Gibbons†, Todd C. Mowry*

*Carnegie Mellon University  †Intel Pittsburgh

SAFARI
Ambit: Bulk-Bitwise in-DRAM Computation

Takeaways
Key Takeaways

- A novel method to accelerate data copy and initialization
- Simple and effective
- Hardware/software cooperative
- Good potential for work building on it to extend it
  - To different granularities
  - To make things more efficient and effective
  - Multiple works have already built on the paper (see LISA, Ambit, and other works in Google Scholar)
- Easy to read and understand paper
Open Discussion
Discussion Starters

- Thoughts on the previous ideas?
- How practical is this?
- Will the problem become bigger and more important over time?
- Will the solution become more important over time?
- Are other solutions better?
- Is this solution clearly advantageous in some cases?
Vivek Seshadri, Yoongu Kim, Chris Fallin, Donghyuk Lee, Rachata Ausavarungnirun, Gennady Pekhimenko, Yixin Luo, Onur Mutlu, Michael A. Kozuch, Phillip B. Gibbons, and Todd C. Mowry,

"RowClone: Fast and Energy-Efficient In-DRAM Bulk Data Copy and Initialization"

Proceedings of the 46th International Symposium on Microarchitecture (MICRO), Davis, CA, December 2013. [Slides (pptx) (pdf)] [Lightning Session Slides (pptx) (pdf)] [Poster (pptx) (pdf)]
RowClone

Fast and Energy-Efficient In-DRAM Bulk Data Copy and Initialization

Vivek Seshadri

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

- I will give a “conference talk”
- You can ask questions and analyze what I described
Parallelism-Aware Batch Scheduling
Enhancing both Performance and Fairness of Shared DRAM Systems

Onur Mutlu and Thomas Moscibroda
Computer Architecture Group
Microsoft Research
Outline

- Background and Goal
- Motivation
  - Destruction of Intra-thread DRAM Bank Parallelism
- Parallelism-Aware Batch Scheduling
  - Batching
  - Within-batch Scheduling
- System Software Support
- Evaluation
- Summary
The DRAM System

FR-FCFS policy
1) Row-hit first
2) Oldest first
Multi-Core Systems

 threads’ requests interfere

Multi-Core Chip

Shared DRAM Memory System
Inter-thread Interference in the DRAM System

- Threads delay each other by causing resource contention:
  - Bank, bus, row-buffer conflicts [MICRO 2007]
- Threads can also destroy each other’s DRAM bank parallelism
  - Otherwise parallel requests can become serialized

- Existing DRAM schedulers are unaware of this interference
- 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 [MICRO 2007]
- System performance loss [MICRO 2007]
- Vulnerability to denial of service [USENIX Security 2007]
- Inability to enforce system-level thread priorities [MICRO 2007]
Our Goal

- Control inter-thread interference in DRAM

- Design a shared DRAM scheduler that
  - provides high system performance
    - preserves each thread’s DRAM bank parallelism
  - provides fairness to threads sharing the DRAM system
    - equalizes memory-slowdowns of equal-priority threads
  - is controllable and configurable
    - enables different service levels for threads with different priorities
Outline

- Background and Goal
- **Motivation**
  - Destruction of Intra-thread DRAM Bank Parallelism
- Parallelism-Aware Batch Scheduling
  - Batching
  - Within-batch Scheduling
- System Software Support
- Evaluation
- Summary
The Problem

- 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
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

**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

Saved Cycles

Average stall-time: ~1.5 bank access latencies
Outline

- Background and Goal
- Motivation
  - Destruction of Intra-thread DRAM Bank Parallelism
- Parallelism-Aware Batch Scheduling (PAR-BS)
  - Request Batching
  - Within-batch Scheduling
- System Software Support
- Evaluation
- Summary
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>max-bank-load</th>
<th>total-load</th>
</tr>
</thead>
<tbody>
<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>T3</td>
<td>T3</td>
</tr>
<tr>
<td>T1</td>
<td>T0</td>
<td>T2</td>
<td>T1</td>
</tr>
<tr>
<td>T2</td>
<td>T2</td>
<td>T1</td>
<td>T2</td>
</tr>
<tr>
<td>T3</td>
<td>T1</td>
<td>T0</td>
<td>T3</td>
</tr>
<tr>
<td>T1</td>
<td>T3</td>
<td>T2</td>
<td>T3</td>
</tr>
</tbody>
</table>

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>T1</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>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>
</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>
</table>

AVG: 3.5 bank access latencies
### Putting It Together: PAR-BS Scheduling Policy

<table>
<thead>
<tr>
<th>PAR-BS Scheduling Policy</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1) Marked requests first</td>
</tr>
<tr>
<td>(2) Row-hit requests first</td>
</tr>
<tr>
<td>(3) Higher-rank thread first (shortest stall-time first)</td>
</tr>
<tr>
<td>(4) Oldest first</td>
</tr>
</tbody>
</table>

#### Batching
- 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

#### Parallelism-aware within-batch scheduling

#### Three properties:
- Exploits row-buffer locality and intra-thread bank parallelism
- Work-conserving
  - Services unmarked requests to banks without marked requests
- Marking-Cap is important
  - Too small cap: destroys row-buffer locality
  - Too large cap: penalizes memory non-intensive threads
- Many more trade-offs analyzed in the paper
Hardware Cost

- <1.5KB storage cost for
  - 8-core system with 128-entry memory request buffer

- No complex operations (e.g., divisions)

- Not on the critical path
  - Scheduler makes a decision only every DRAM cycle
Outline

- Background and Goal
- Motivation
  - Destruction of Intra-thread DRAM Bank Parallelism
- Parallelism-Aware Batch Scheduling
  - Batching
  - Within-batch Scheduling
- System Software Support
- Evaluation
- Summary
System Software Support

- OS conveys each thread’s priority level to the controller
  - Levels 1, 2, 3, ... (highest to lowest priority)
- Controller enforces priorities in two ways
  - Mark requests from a thread with priority X only every Xth batch
  - Within a batch, higher-priority threads’ requests are scheduled first

- Purely opportunistic service
  - Special very low priority level L
  - Requests from such threads never marked

- Quantitative analysis in paper
Outline

- Background and Goal
- Motivation
  - Destruction of Intra-thread DRAM Bank Parallelism
- Parallelism-Aware Batch Scheduling
  - Batching
  - Within-batch Scheduling
- System Software Support
- Evaluation
- Summary
Evaluation Methodology

- 4-, 8-, 16-core systems
  - x86 processor model based on Intel Pentium M
  - 4 GHz processor, 128-entry instruction window
  - 512 Kbyte per core private L2 caches, 32 L2 miss buffers

- Detailed DRAM model based on Micron DDR2-800
  - 128-entry memory request buffer
  - 8 banks, 2Kbyte row buffer
  - 40ns (160 cycles) row-hit round-trip latency
  - 80ns (320 cycles) row-conflict round-trip latency

- Benchmarks
  - Multiprogrammed SPEC CPU2006 and Windows Desktop applications
  - 100, 16, 12 program combinations for 4-, 8-, 16-core experiments
Comparison with Other DRAM Controllers

- **Baseline FR-FCFS** [Zuravleff and Robinson, US Patent 1997; Rixner et al., ISCA 2000]
  - Prioritizes row-hit requests, older requests
  - Unfairly penalizes threads with low row-buffer locality, memory non-intensive threads

- **FCFS** [Intel Pentium 4 chipsets]
  - Oldest-first; low DRAM throughput
  - Unfairly penalizes memory non-intensive threads

- **Network Fair Queueing (NFQ)** [Nesbit et al., MICRO 2006]
  - Equally partitions DRAM bandwidth among threads
  - Does not consider inherent (baseline) DRAM performance of each thread
  - Unfairly penalizes threads with high bandwidth utilization [MICRO 2007]
  - Unfairly prioritizes threads with bursty access patterns [MICRO 2007]

- **Stall-Time Fair Memory Scheduler (STFM)** [Mutlu & Moscibroda, MICRO 2007]
  - Estimates and balances thread slowdowns relative to when run alone
  - Unfairly treats threads with inaccurate slowdown estimates
  - Requires multiple (approximate) arithmetic operations
Unfairness on 4-, 8-, 16-core Systems

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

- 8.3% improvement in normalized Hmean speedup for 4-core systems
- 6.1% improvement in normalized Hmean speedup for 8-core systems
- 5.1% improvement in normalized Hmean speedup for 16-core systems
Outline

- Background and Goal
- Motivation
  - Destruction of Intra-thread DRAM Bank Parallelism
- Parallelism-Aware Batch Scheduling
  - Batching
  - Within-batch Scheduling
- System Software Support
- Evaluation
- Summary
Summary

- Inter-thread interference can destroy each thread’s DRAM bank parallelism
  - Serializes a thread’s requests → reduces system throughput
  - Makes techniques that exploit memory-level parallelism less effective
  - Existing DRAM controllers unaware of intra-thread bank parallelism

- A new approach to fair and high-performance DRAM scheduling
  - **Batching**: Eliminates starvation, allows fair sharing of the DRAM system
  - **Parallelism-aware thread ranking**: Preserves each thread’s bank parallelism
  - **Flexible and configurable**: Supports system-level thread priorities → QoS policies

- PAR-BS provides better fairness and system performance than previous DRAM schedulers
Thank you. Questions?
Parallelism-Aware Batch Scheduling
Enhancing both Performance and Fairness of Shared DRAM Systems

Onur Mutlu and Thomas Moscibroda
Computer Architecture Group
Microsoft Research
Backup Slides
Multiple Memory Controllers (I)

- **Local ranking**: Each controller uses PAR-BS independently
  - Computes its own ranking based on its local requests

- **Global ranking**: Meta controller that computes a global ranking across all controllers based on global information
  - Only needs to track bookkeeping info about each thread’s requests to the banks in each controller

- The difference between the ranking computed by each scheme depends on the balance of the distribution of requests to each controller
  - Balanced → Local and global rankings are similar
Multiple Memory Controllers (II)

16-core system, 4 memory controllers

Unfairness

Normalized Hmean-Speedup

- FR-FCFS
- FCFS
- NFQ
- STFM
- PAR-BS Local
- PAR-BS Global

1.18X 1.33X

7.4% 11.5%
### Example with Row Hits

<table>
<thead>
<tr>
<th></th>
<th>Stall time</th>
<th></th>
<th>Stall time</th>
<th></th>
<th>Stall time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread 1</td>
<td>4</td>
<td>Thread 1</td>
<td>5.5</td>
<td>Thread 1</td>
<td>1</td>
</tr>
<tr>
<td>Thread 2</td>
<td>4</td>
<td>Thread 2</td>
<td>3</td>
<td>Thread 2</td>
<td>2</td>
</tr>
<tr>
<td>Thread 3</td>
<td>5</td>
<td>Thread 3</td>
<td>4.5</td>
<td>Thread 3</td>
<td>4</td>
</tr>
<tr>
<td>Thread 4</td>
<td>7</td>
<td>Thread 4</td>
<td>4.5</td>
<td>Thread 4</td>
<td>5.5</td>
</tr>
<tr>
<td>AVG</td>
<td>5</td>
<td>AVG</td>
<td>4.375</td>
<td>AVG</td>
<td>3.125</td>
</tr>
</tbody>
</table>

(a) Arrival order (and FCFS schedule)

(b) FR-FCFS schedule

(c) PAR-BS schedule
End of Backup Slides
Now Your Turn to Analyze…

- Background, Problem & Goal
- Novelty
- Key Approach and Ideas
- Mechanisms (in some detail)
- Key Results: Methodology and Evaluation
- Summary
- Strengths
- Weaknesses
- Thoughts and Ideas
- Takeaways
- Open Discussion
PAR-BS Pros and Cons

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

- **Downsides:**
  - Does not always prioritize the latency-sensitive applications
  - Deadline guarantees?
  - Complexity?

- Some ideas implemented in real SoC memory controllers
More on PAR-BS

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