Review: DRAM Controller: Functions

- Ensure correct operation of DRAM (refresh and timing)
- Service DRAM requests while obeying timing constraints of DRAM chips
  - Constraints: resource conflicts (bank, bus, channel), minimum write-to-read delays
  - Translate requests to DRAM command sequences
- Buffer and schedule requests to improve performance
  - Reordering, row-buffer, bank, rank, bus management
- Manage power consumption and thermals in DRAM
  - Turn on/off DRAM chips, manage power modes

DRAM Power Management

- DRAM chips have power modes
  - Idea: When not accessing a chip power it down
- Power states
  - Active (highest power)
  - All banks idle
  - Power-down
  - Self-refresh (lowest power)
- Tradeoff: State transitions incur latency during which the chip cannot be accessed

Review: Why are DRAM Controllers Difficult to Design?

- Need to obey DRAM timing constraints for correctness
  - There are many (50+) timing constraints in DRAM
  - tWTR: Minimum number of cycles to wait before issuing a read command after a write command is issued
  - tRC: Minimum number of cycles between the issuing of two consecutive activate commands to the same bank
  - ...
- Need to keep track of many resources to prevent conflicts
  - Channels, banks, ranks, data bus, address bus, row buffers
- Need to handle DRAM refresh
- Need to optimize for performance (in the presence of constraints)
  - Reordering is not simple
  - Predicting the future?
Review: Many DRAM Timing Constraints

<table>
<thead>
<tr>
<th>Latency</th>
<th>Symbol</th>
<th>DRAM cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Precharge</td>
<td>TLP</td>
<td>11</td>
</tr>
<tr>
<td>Read column addr strobe</td>
<td>CL</td>
<td>11</td>
</tr>
<tr>
<td>Address</td>
<td>TAL</td>
<td>0</td>
</tr>
<tr>
<td>Activate to precharge</td>
<td>TAMS</td>
<td>28</td>
</tr>
<tr>
<td>Burst length</td>
<td>TB</td>
<td>4</td>
</tr>
<tr>
<td>Activate to ready (different banks)</td>
<td>TTRD</td>
<td>6</td>
</tr>
<tr>
<td>Write to read</td>
<td>WTR</td>
<td>6</td>
</tr>
</tbody>
</table>

Table 4. DDR3 1600 DRAM timing specifications


Review: More on DRAM Operation


Self-Optimizing DRAM Controllers

- Problem: DRAM controllers difficult to design → It is difficult for human designers to design a policy that can adapt itself very well to different workloads and different system conditions
- Idea: Design a memory controller that adapts its scheduling policy decisions to workload behavior and system conditions using machine learning.
- Observation: Reinforcement learning maps nicely to memory control.
- Design: Memory controller is a reinforcement learning agent that dynamically and continuously learns and employs the best scheduling policy.


Goal: Learn to choose actions to maximize $r_0 + \gamma r_1 + \gamma^2 r_2 + \ldots$ ( $0 \leq \gamma < 1$)
Self-Optimizing DRAM Controllers

- Dynamically adapt the memory scheduling policy via interaction with the system at runtime
  - Associate system states and actions (commands) with long term reward values
  - Schedule command with highest estimated long-term value in each state
  - Continuously update state-action values based on feedback from system

States, Actions, Rewards

- Reward function
  - +1 for scheduling Read and Write commands
  - 0 at all other times
- State attributes
  - Number of reads, writes, and load misses in transaction queue
  - Number of pending writes and ROB heads waiting for referenced row
  - Request’s relative ROB order
- Actions
  - Activate
  - Write
  - Read - load miss
  - Read - store miss
  - Precharge - pending
  - Precharge - preemptive
  - NOP

Performance Results

Engin Ipek, Onur Mutlu, José F. Martínez, and Rich Caruana, “Self Optimizing Memory Controllers: A Reinforcement Learning Approach”

Figure 4: High-level overview of an RL-based scheduler.

Figure 7: Performance comparison of in-order, FR-FCFS, RL-based, and optimistic memory controllers

Figure 15: Performance comparison of FR-FCFS and RL-based memory controllers on systems with 6.4GB/s and 12.8GB/s peak DRAM bandwidth
Self Optimizing DRAM Controllers

- Advantages
  - Adapts the scheduling policy dynamically to changing workload behavior and to maximize a long-term target
  - Reduces the designer’s burden in finding a good scheduling policy. Designer specifies:
    1) What system variables might be useful
    2) What target to optimize, but not how to optimize it

- Disadvantages
  - Black box: designer much less likely to implement what she cannot easily reason about
  - How to specify different reward functions that can achieve different objectives? (e.g., fairness, QoS)

---

Major Trends Affecting Main Memory (I)

- Need for main memory capacity, bandwidth, QoS increasing

- Main memory energy/power is a key system design concern

- DRAM technology scaling is ending

---

Trends Affecting Main Memory

- Major Trends Affecting Main Memory (II)
  - Need for main memory capacity, bandwidth, QoS increasing
    - Multi-core: increasing number of cores
    - Data-intensive applications: increasing demand/hunger for data
    - Consolidation: cloud computing, GPUs, mobile

- Main memory energy/power is a key system design concern

- DRAM technology scaling is ending
Major Trends Affecting Main Memory (III)

- Need for main memory capacity, bandwidth, QoS increasing

- Main memory energy/power is a key system design concern
  - \( \sim 40-50\% \) energy spent in off-chip memory hierarchy [Lefurgy, IEEE Computer 2003]
  - DRAM consumes power even when not used (periodic refresh)

- DRAM technology scaling is ending

SAFARI 17

Major Trends Affecting Main Memory (IV)

- Need for main memory capacity, bandwidth, QoS increasing

- Main memory energy/power is a key system design concern

- DRAM technology scaling is ending
  - ITRS projects DRAM will not scale easily below X nm
  - Scaling has provided many benefits:
    - higher capacity (density), lower cost, lower energy

SAFARI 18

The DRAM Scaling Problem

- DRAM stores charge in a capacitor (charge-based memory)
  - Capacitor must be large enough for reliable sensing
  - Access transistor should be large enough for low leakage and high retention time
  - Scaling beyond 40-35nm (2013) is challenging [ITRS, 2009]

- DRAM capacity, cost, and energy/power hard to scale

SAFARI 19

Solution 1: Tolerate DRAM

- Overcome DRAM shortcomings with
  - System-DRAM co-design
  - Novel DRAM architectures, interface, functions
  - Better waste management (efficient utilization)

- Key issues to tackle
  - Reduce refresh energy
  - Improve bandwidth and latency
  - Reduce waste
  - Enable reliability at low cost

- Liu+, "An Experimental Study of Data Retention Behavior in Modern DRAM Devices" ISCA'13.
- Seshadri+, "RowClone: Fast and Efficient In-DRAM Copy and Initialization of Bulk Data," 2013.

SAFARI 20
Tolerating DRAM: System-DRAM Co-Design

New DRAM Architectures

- RAIDR: Reducing Refresh Impact
- TL-DRAM: Reducing DRAM Latency
- SALP: Reducing Bank Conflict Impact
- RowClone: Fast Bulk Data Copy and Initialization

DRAM Refresh

- DRAM capacitor charge leaks over time
- The memory controller needs to refresh each row periodically to restore charge
  - Activate + precharge each row every N ms
  - Typical N = 64 ms
- Downsides of refresh
  - Energy consumption: Each refresh consumes energy
  - Performance degradation: DRAM rank/bank unavailable while refreshed
  - QoS/predictability impact: (Long) pause times during refresh
  - Refresh rate limits DRAM density scaling
Refresh Today: Auto Refresh

A batch of rows are periodically refreshed via the auto-refresh command.

Refresh Overhead: Performance

- Present
- Future

- 2 Gb
- 4 Gb
- 8 Gb
- 16 Gb
- 32 Gb
- 64 Gb

Device capacity

Problem with Conventional Refresh

- Today: Every row is refreshed at the same rate

- Observation: Most rows can be refreshed much less often without losing data [Kim+, EDL’09]
- Problem: No support in DRAM for different refresh rates per row
Retention Time of DRAM Rows

- **Observation:** Only very few rows need to be refreshed at the worst-case rate

- Can we exploit this to reduce refresh operations at low cost?

---

Reducing DRAM Refresh Operations

- **Idea:** Identify the retention time of different rows and refresh each row at the frequency it needs to be refreshed

- **(Cost-conscious) Idea:** Bin the rows according to their minimum retention times and refresh rows in each bin at the refresh rate specified for the bin
  - e.g., a bin for 64-128ms, another for 128-256ms, ...

- **Observation:** Only very few rows need to be refreshed very frequently [64-128ms] → Have only a few bins → Low HW overhead to achieve large reductions in refresh operations


---

RAIDR: Mechanism

- **64-128ms**
  - >256ms
  - 1.25KB storage in controller for 32GB DRAM memory
  - bins at different rates
  → probe Bloom Filters to determine refresh rate of a row

---

1. Profiling

To profile a row:
1. Write data to the row
2. Prevent it from being refreshed
3. Measure time before data corruption

<table>
<thead>
<tr>
<th>Initially</th>
<th>Row 1</th>
<th>Row 2</th>
<th>Row 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>After 64 ms</td>
<td>11111111...</td>
<td>11111111...</td>
<td>11111111...</td>
</tr>
<tr>
<td>After 128 ms</td>
<td>11011111... (64–128ms)</td>
<td>11111111...</td>
<td>11111111...</td>
</tr>
<tr>
<td>After 256 ms</td>
<td>11111011... (128–256ms)</td>
<td>11111111... (128–256ms)</td>
<td>&gt;256ms</td>
</tr>
</tbody>
</table>
2. Binning

- How to efficiently and scalably store rows into retention time bins?
- Use Hardware Bloom Filters [Bloom, CACM 1970]

**Example with 64-128ms bin:**

```
0 0 1 0 1 0 0 0 0 1 0 0 0 0 0 0
```

- Hash function 1
- Hash function 2
- Hash function 3

Insert Row 1

---

**Example with 64-128ms bin:**

```
0 0 1 0 1 0 0 0 0 1 0 0 0 0 0 0
```

- Hash function 1
- Hash function 2
- Hash function 3

Row 1 present? Yes

---

**Example with 64-128ms bin:**

```
0 0 1 0 1 0 0 0 0 0 1 0 0 0 0 0 0
```

- Hash function 1
- Hash function 2
- Hash function 3

Row 2 present? No

---

**Example with 64-128ms bin:**

```
0 0 1 0 1 1 0 0 0 1 0 0 1 0 1 0
```

- Hash function 1
- Hash function 2
- Hash function 3

Insert Row 4
Bloom Filter Operation Example

Example with 64-128ms bin:

\[0 \ 0 \ 1 \ 0 \ 1 \ 1 \ 0 \ 0 \ 0 \ 1 \ 0 \ 0 \ 1 \ 0 \ 1 \ 0\]

Hash function 1
Hash function 2
Hash function 3

Row 5 present?
Yes (false positive)

Benefits of Bloom Filters as Bins

- **False positives**: a row may be declared present in the Bloom filter even if it was never inserted
  - Not a problem: Refresh some rows more frequently than needed
- **No false negatives**: rows are never refreshed less frequently than needed (no correctness problems)
- **Scalable**: a Bloom filter never overflows (unlike a fixed-size table)
- **Efficient**: No need to store info on a per-row basis; simple hardware \(\rightarrow\) 1.25 KB for 2 filters for 32 GB DRAM system

3. Refreshing (RAIDR Refresh Controller)

Choose a refresh candidate row

Determine which bin the row is in

Determine if refreshing is needed

Tolerating Temperature Changes

- Change in temperature causes retention time of all cells to change by a uniform and predictable factor

- Refresh rate scaling: increase the refresh rate for all rows uniformly, depending on the temperature

- Implementation: counter with programmable period
  - Lower temperature ⇒ longer period ⇒ less frequent refreshes
  - Higher temperature ⇒ shorter period ⇒ more frequent refreshes

RAIDR: Baseline Design

RAIDR in Memory Controller: Option 1

Overhead of RAIDR in DRAM controller:
1.25 KB Bloom Filters, 3 counters, additional commands issued for per-row refresh (all accounted for in evaluations)

RAIDR in DRAM Chip: Option 2

Overhead of RAIDR in DRAM chip:
Per-chip overhead: 20B Bloom Filters, 1 counter (4 Gbit chip)
Total overhead: 1.25KB Bloom Filters, 64 counters (32 GB DRAM)
RAIDR Results

- **Baseline:**
  - 32 GB DDR3 DRAM system (8 cores, 512KB cache/core)
  - 64ms refresh interval for all rows

- **RAIDR:**
  - 64–128ms retention range: 256 B Bloom filter, 10 hash functions
  - 128–256ms retention range: 1 KB Bloom filter, 6 hash functions
  - Default refresh interval: 256 ms

- Results on SPEC CPU2006, TPC-C, TPC-H benchmarks
  - 74.6% refresh reduction
  - ~16%/20% DRAM dynamic/idle power reduction
  - ~9% performance improvement

RAIDR Refresh Reduction

RAIDR: Performance

RAIDR: DRAM Energy Efficiency
DRAM Device Capacity Scaling: Performance

![Graph showing performance benefits increasing with DRAM chip capacity]

RAIDR performance benefits increase with DRAM chip capacity

More Readings Related to RAIDR


New DRAM Architectures

- RAIDR: Reducing Refresh Impact
- TL-DRAM: Reducing DRAM Latency
- SALP: Reducing Bank Conflict Impact
- RowClone: Fast Bulk Data Copy and Initialization
Tiered-Latency DRAM: Reducing DRAM Latency

What Causes the Long Latency?

Historical DRAM Latency-Capacity Trend

19th International Symposium on High-Performance Computer Architecture (HPCA), Shenzhen, China, February 2013. Slides (pptx)

DRAM latency continues to be a critical bottleneck

What Causes the Long Latency?

DRAM Latency = Subarray Latency + I/O Latency

Dominant
Why is the Subarray So Slow?

- Long bitline
  - Amortizes sense amplifier cost \(\rightarrow\) Small area
  - Large bitline capacitance \(\rightarrow\) High latency \& power

Trade-Off: Area (Die Size) vs. Latency

- Long Bitline
  - Faster
  - Trade-Off: Area vs. Latency
- Short Bitline
  - Smaller

Approximating the Best of Both Worlds

- Long Bitline: Small Area, High Latency
- Our Proposal: Large Area, Low Latency
- Short Bitline: Small Area, Low Latency
- Need Isolation
- Add Isolation Transistors
Approximating the Best of Both Worlds

Tiered-Latency DRAM

- Divide a bitline into two segments with an isolation transistor

Near Segment Access

- Turn **off** the isolation transistor

- Reduced bitline length
- Reduced bitline capacitance
- \( \Rightarrow \) Low latency & low power

Far Segment Access

- Turn **on** the isolation transistor

- Long bitline length
- Large bitline capacitance
- Additional resistance of isolation transistor
- \( \Rightarrow \) High latency & high power
Latency, Power, and Area Evaluation

- **Commodity DRAM**: 512 cells/bitline
- **TL-DRAM**: 512 cells/bitline
  - Near segment: 32 cells
  - Far segment: 480 cells

**Latency Evaluation**
- SPICE simulation using circuit-level DRAM model

**Power and Area Evaluation**
- DRAM area/power simulator from Rambus
- DDR3 energy calculator from Micron

---

**Commodity DRAM vs. TL-DRAM**

- **DRAM Latency (tRC)**
- **DRAM Power**

- **Latency**
  - Commodity DRAM
  - Near TL-DRAM: +23%  
  - Far TL-DRAM: +49%

- **Power**
  - Commodity DRAM
  - Near TL-DRAM: -51%

**DRAM Area Overhead**

~3%: mainly due to the isolation transistors

---

**Latency vs. Near Segment Length**

- Longer near segment length leads to higher near segment latency

**Latency vs. Near Segment Length**

- Far segment latency is higher than commodity DRAM latency

---

65 66 67 68
Trade-Off: Area (Die-Area) vs. Latency

Leveraging Tiered-Latency DRAM
- TL-DRAM is a *substrate* that can be leveraged by the hardware and/or software
- Many potential uses:
  1. Use near segment as hardware-managed *inclusive* cache to far segment
  2. Use near segment as hardware-managed *exclusive* cache to far segment
  3. Profile-based page mapping by operating system
  4. Simply replace DRAM with TL-DRAM

Near Segment as Hardware-Managed Cache

Inter-Segment Migration
- **Goal:** Migrate source row into destination row
- **Naïve way:** Memory controller reads the source row byte by byte and writes to destination row byte by byte → *High latency*

• Challenge 1: How to efficiently migrate a row between segments?
• Challenge 2: How to efficiently manage the cache?
Inter-Segment Migration

- **Our way:**
  - Source and destination cells **share bitlines**
  - Transfer data from source to destination across **shared bitlines concurrently**

**Near Segment**

**Far Segment**

**Isolation Transistor**

**Sense Amplifier**

**Challenge 1:** How to efficiently migrate a row between segments?
**Challenge 2:** How to efficiently manage the cache?

Evaluation Methodology

- **System simulator**
  - CPU: Instruction-trace-based x86 simulator
  - Memory: Cycle-accurate DDR3 DRAM simulator

- **Workloads**
  - 32 Benchmarks from TPC, STREAM, SPEC CPU2006

- **Performance Metrics**
  - Single-core: Instructions-Per-Cycle
  - Multi-core: Weighted speedup
Configurations
- **System configuration**
  - CPU: 5.3GHz
  - LLC: 512kB private per core
  - Memory: DDR3-1066
    - 1-2 channel, 1 rank/channel
    - 8 banks, 32 subarrays/bank, **512 cells/bitline**
    - Row-interleaved mapping & closed-row policy

- **TL-DRAM configuration**
  - Total bitline length: **512 cells/bitline**
  - Near segment length: 1-256 cells
  - Hardware-managed inclusive cache: near segment

Performance & Power Consumption
- **Normalized Performance**
  - Core-Count (Channel)
  - Core-Count (Channel)

Using near segment as a cache improves performance and reduces power consumption

Single-Core: Varying Near Segment Length
- **Maximum IPC Improvement**
  - Larger cache capacity
  - Higher cache access latency

By adjusting the near segment length, we can trade off cache capacity for cache latency

Other Mechanisms & Results
- **More mechanisms** for leveraging TL-DRAM
  - Hardware-managed **exclusive** caching mechanism
  - Profile-based page mapping to near segment
  - TL-DRAM improves performance and reduces power consumption with other mechanisms

- **More than two tiers**
  - Latency evaluation for three-tier TL-DRAM

- **Detailed circuit evaluation**
  - For DRAM latency and power consumption
  - Examination of tRC and tRCD

- **Implementation details** and **storage cost analysis** in memory controller
Summary of TL-DRAM

- **Problem:** DRAM latency is a critical performance bottleneck
- **Our Goal:** Reduce DRAM latency with low area cost
- **Observation:** Long bitlines in DRAM are the dominant source of DRAM latency
- **Key Idea:** Divide long bitlines into two shorter segments
  - Fast and slow segments
- **Tiered-latency DRAM:** Enables latency heterogeneity in DRAM
  - Can leverage this in many ways to improve performance and reduce power consumption
- **Results:** When the fast segment is used as a cache to the slow segment → Significant performance improvement (>12%) and power reduction (>23%) at low area cost (3%)

New DRAM Architectures

- RAIDR: Reducing Refresh Impact
- TL-DRAM: Reducing DRAM Latency
- SALP: Reducing Bank Conflict Impact
- RowClone: Fast Bulk Data Copy and Initialization

Subarray-Level Parallelism: Reducing Bank Conflict Impact

Yoongu Kim, Vivek Seshadri, Donghyuk Lee, Jamie Liu, and Onur Mutlu,
"A Case for Exploiting Subarray-Level Parallelism (SALP) in DRAM"
Proceedings of the 39th International Symposium on Computer Architecture (ISCA),
Portland, OR, June 2012. Slides (pptx)

The Memory Bank Conflict Problem

- Two requests to the same bank are serviced serially
- Problem: Costly in terms of performance and power
- Goal: We would like to reduce bank conflicts without increasing the number of banks (at low cost)

- Idea: Exploit the internal sub-array structure of a DRAM bank to parallelize bank conflicts
  - By reducing global sharing of hardware between sub-arrays
The Problem with Memory Bank Conflicts

- **Two Banks**

  ![Diagram of two banks with write (Wr) and read (Rd) operations served in parallel.]

- **One Bank**

  ![Diagram of one bank with write (Wr) and read (Rd) operations that are wasted.]

**Goal**

- **Goal:** Mitigate the detrimental effects of bank conflicts in a cost-effective manner.

  - **Naïve solution:** Add more banks
    - Very expensive
  - **Cost-effective solution:** Approximate the benefits of more banks without

**Key Observation #1**

A DRAM bank is divided into subarrays:

- Logical Bank
  - Rows
  - Row-Buffer
- Physical Bank
  - Subarray
    - 64
    - 1
  - 32k rows
  - Global Row-Buf

A single row-buffer cannot drive all rows. Many local row-buffers, one at each subarray.

**Key Observation #2**

Each subarray is mostly independent…

- except occasionally sharing global structures.
Key Idea: Reduce Sharing of Globals

1. Parallel access to subarrays
2. Utilize multiple local row-buffers

Overview of Our Mechanism

1. Parallelize
2. Utilize multiple local row-buffers but diff. subarrays

Challenges: Global Structures

1. Global Address Latch
2. Global Bitlines

Challenge #1. Global Address Latch
Solution #1. Subarray Address Latch

Challenges: Global Structures

1. Global Address Latch
   - Problem: Only *one* raised wordline
   - Solution: Subarray Address Latch

2. Global Bitlines

Challenge #2. Global Bitlines

Solution #2. Designated-Bit Latch

Selectively connect local to global
Challenges: Global Structures

1. Global Address Latch
   - Problem: Only one raised wordline
   - Solution: Subarray Address Latch

2. Global Bitlines
   - Problem: Collision during access
   - Solution: Designated-Bit Latch

MASA (Multitude of Activated Subarrays)

MASA: Advantages

- Baseline (Subarray-Oblivious)

MASA: Overhead

- DRAM Die Size: 0.15% increase
  - Subarray Address Latches
  - Designated-Bit Latches & Wire

- DRAM Static Energy: Small increase
  - 0.56mW for each activated subarray
  - But saves dynamic energy

- Controller: Small additional storage
  - Keep track of subarray status (<256B)
  - Keep track of new timing constraints

Cheaper Mechanisms

Latches

1. Serialization
2. Write Penalty
3. Thrashing
**System Configuration**

- **System Configuration**
  - CPU: 5.3GHz, 128 ROB, 8 MSHR
  - LLC: 512kB per-core slice

- **Memory Configuration**
  - DDR3-1066
  - *(default)* 1 channel, 1 rank, 8 banks, 8 subarrays-per-bank
  - *(sensitivity)* 1-8 chan, 1-8 ranks, 8-64 banks, 1-128 subarrays

- **Mapping & Row-Policy**
  - *(default)* Line-interleaved & Closed-row
  - *(sensitivity)* Row-interleaved & Open-row

- **DRAM Controller Configuration**
  - 64-/64-entry read/write queues per-channel
  - DRAM-FCFS, batch scheduling for writes

---

**SALP: Single-Core Results**

- SALP-1, SALP-2, MASA improve performance at low cost

- MASA achieves most of the benefit of having more banks *(“Ideal”)*

---

**Subarray-Level Parallelism: Results**

- MASA increases energy-efficiency

\[
\begin{array}{ccc}
\text{Normalized Dynamic Energy} & \text{Baseline} & \text{MASA} \\
0.0 & 0.2 & +13 \\
0.2 & 0.4 & \\
0.4 & 0.6 & \\
0.6 & 0.8 & \\
0.8 & 1.0 & \\
1.0 & 1.2 & \\
\end{array}
\]

\[
\begin{array}{ccc}
\text{Row-Buffer Hit Rate} & \text{Baseline} & \text{MASA} \\
0% & 20% & +13 \\
20% & 40% & \\
40% & 60% & \\
60% & 80% & \\
80% & 100% & \\
\end{array}
\]
New DRAM Architectures

- RAIDR: Reducing Refresh Impact
- TL-DRAM: Reducing DRAM Latency
- SALP: Reducing Bank Conflict Impact
- RowClone: Fast Bulk Data Copy and Initialization

RowClone: Fast Bulk Data Copy and Initialization


Today’s Memory: Bulk Data Copy

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

Future: RowClone (In-Memory Copy)

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

DRAM operation (load one byte)

1. Activate row
2. Transfer row
3. Transfer byte onto bus

RowClone: in-DRAM Row Copy (and Initialization)

1. Activate row A
2. Transfer row
3. Activate row B
4. Transfer row

RowClone: Key Idea

- DRAM banks contain
  1. Multiple rows of DRAM cells – row = 8KB
  2. A row buffer shared by the DRAM rows

- Large scale copy
  1. Copy data from source row to row buffer
  2. Copy data from row buffer to destination row

  Can be accomplished by two consecutive ACTIVATEs (if source and destination rows are in the same subarray)

RowClone: Intra-subarray Copy
RowClone: Inter-bank Copy

- **src**
- **dst**

1. **Transfer** (our proposal)
2. **Read** → I/O Bus
3. **Write** → I/O Bus

RowClone: Inter-subarray Copy

- **src** → **temp** → **dst**

1. **Transfer** (src to temp)
2. **Transfer** (temp to dst)

Fast Row Initialization

- Fix a row at Zero (0.5% loss in capacity)

RowClone: Latency and Energy Savings

- **Baseline**
- **Intra-Subarray**
- **Inter-Subarray**

- **Latency**
  - Baseline: 1.2
  - Intra-Subarray: 0.8
  - Inter-Subarray: 0.6

- **Energy**
  - Baseline: 1.2
  - Intra-Subarray: 0.8
  - Inter-Subarray: 0.6

Fix a row at Zero (0.5% loss in capacity)

RowClone: Latency and Energy Savings

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Absolute</th>
<th>Reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Latency (ms)</td>
<td>Energy (µJ)</td>
</tr>
<tr>
<td>4KB Copy</td>
<td>1046</td>
<td>3.6</td>
</tr>
<tr>
<td>Baseline</td>
<td>1046</td>
<td>3.6</td>
</tr>
<tr>
<td>Intra-subarray</td>
<td>90</td>
<td>0.04</td>
</tr>
<tr>
<td>Inter-Bank - PSM</td>
<td>540</td>
<td>1.1</td>
</tr>
<tr>
<td>Intra-Bank - PSM</td>
<td>1050</td>
<td>2.5</td>
</tr>
</tbody>
</table>

| 4KB Zeroing        |          |           |
| Baseline           | 546      | 2.0       | 1.0     | 1.0     |
| Intra-subarray     | 90       | 0.05      | 6.06    | 41.5    |

Table 3: Latency and energy reductions due to RowClone

Summary

- Major problems with DRAM scaling and design: high refresh rate, high latency, low parallelism, bulk data movement

- Four new DRAM designs
  - RAIDR: Reduces refresh impact
  - TL-DRAM: Reduces DRAM latency at low cost
  - SALP: Improves DRAM parallelism
  - RowClone: Reduces energy and performance impact of bulk data copy

- All four designs
  - Improve both performance and energy consumption
  - Are low cost (low DRAM area overhead)
  - Enable new degrees of freedom to software & controllers

- Rethinking DRAM interface and design essential for scaling
  - Co-design DRAM with the rest of the system

Computer Architecture: Main Memory (Part III)

Prof. Onur Mutlu
Carnegie Mellon University