

CSCI 564 Advanced Computer Architecture

Lecture 10: Main Memory System

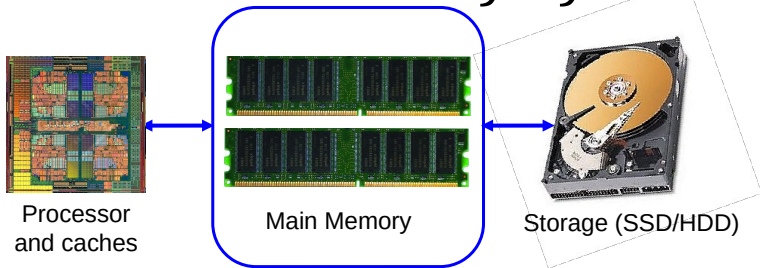
Dr. Bo Wu (with modifications by Sumner Evans)

April 18, 2021

Colorado School of Mines

The Main Memory System

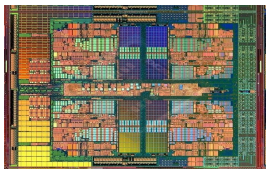
The Main Memory System



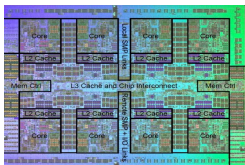
- Main memory is a critical component of all computing systems: server, mobile, embedded, desktop, sensor
- Main memory system must scale (in size, technology, efficiency, cost, and management algorithms) to maintain performance growth and technology scaling benefits

Demand for Memory Capacity

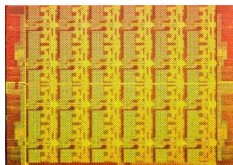
More cores \Rightarrow More concurrency \Rightarrow Larger working set



AMD Barcelona: 4 cores



IBM Power7: 8 cores



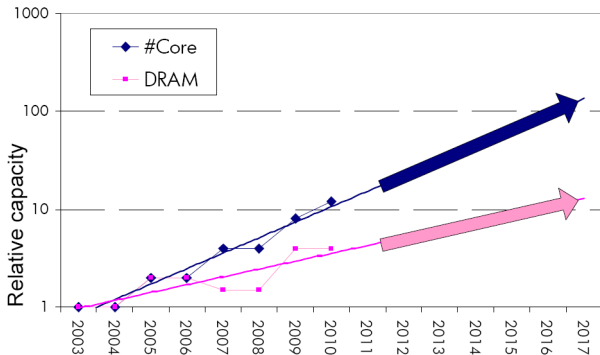
Intel SCC: 48 cores

- Modern applications are (increasingly) data-intensive
- Many applications/virtual machines (will) share main memory
 - **Cloud computing/servers**: Consolidation to improve efficiency
 - **Many-core CPUs**: Many threads from multiple parallel applications
 - **Mobile**: Interactive + non-interactive consolidation
 - ...

Example: The Memory Capacity Gap

Core count doubling ~ every 2 years

DRAM DIMM capacity doubling ~ every 3 years



- *Memory capacity per core* expected to drop by 30% every two years
- Trends worse for *memory bandwidth per core*!

Major Trends Affecting Main Memory (I)

- Need for main memory capacity, bandwidth, QoS increasing
 - **Multi-core**: increasing number of cores/agents
 - **Data-intensive applications**: increasing demand/hunger for data
 - **Consolidation**: Cloud computing, GPUs, mobile, heterogeneity
- Main memory energy/power is a key system design concern
- DRAM technology scaling is ending

Major Trends Affecting Main Memory (II)

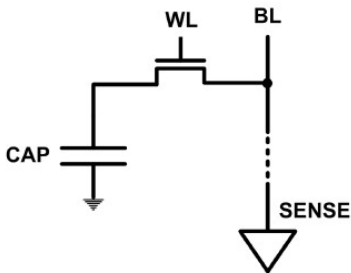
- Need for main memory capacity, bandwidth, QoS increasing
- Main memory energy/power is a key system design concern
 - IBM servers: ~50% energy spent in off-chip memory hierarchy [Lefurgy, IEEE Computer 2003]
 - DRAM consumes power when idle and needs periodic refresh
- DRAM technology scaling is ending

Major Trends Affecting Main Memory (III)

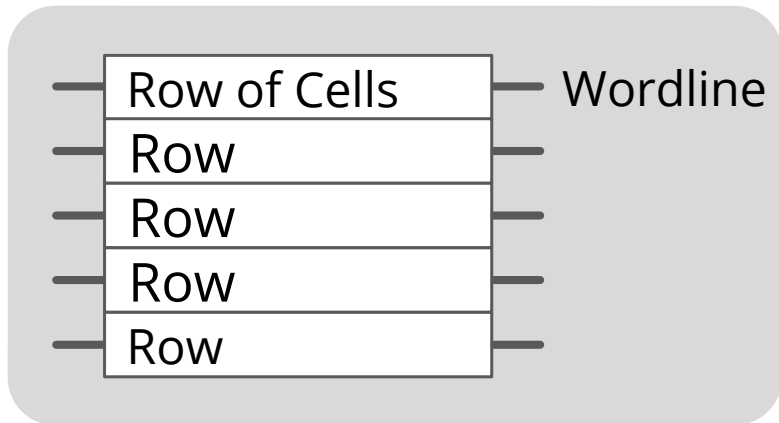
- Need for main memory capacity, bandwidth, QoS increasing
- Main memory energy/power is a key system design concern
- DRAM technology scaling is ending
 - ITRS projects DRAM will not scale easily below X nm
 - Scaling has provided many benefits:
 - higher capacity, higher density, lower cost, lower energy

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]

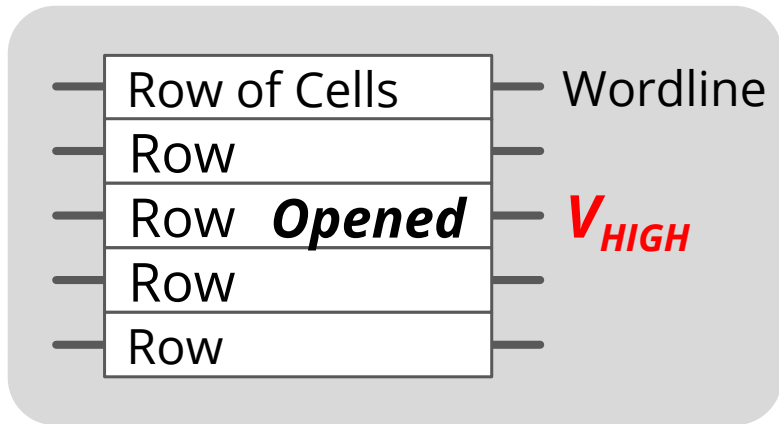


Evidence of the DRAM Scaling Problem



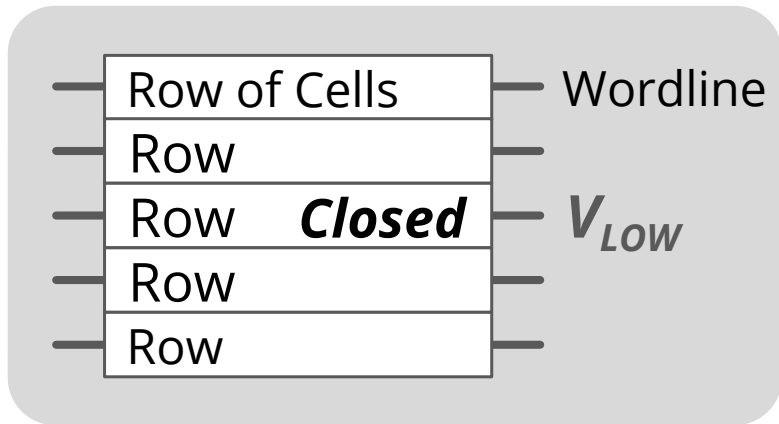
Kim+, "Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors," ISCA 2014.

Evidence of the DRAM Scaling Problem



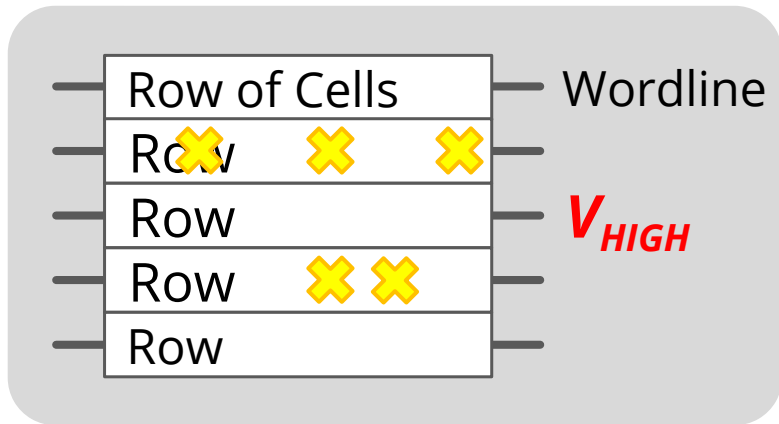
Kim+, "Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors," ISCA 2014.

Evidence of the DRAM Scaling Problem



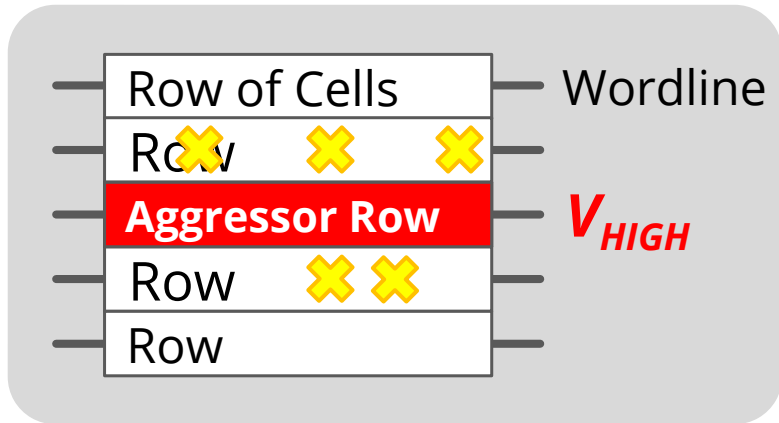
Kim+, "Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors," ISCA 2014.

Evidence of the DRAM Scaling Problem



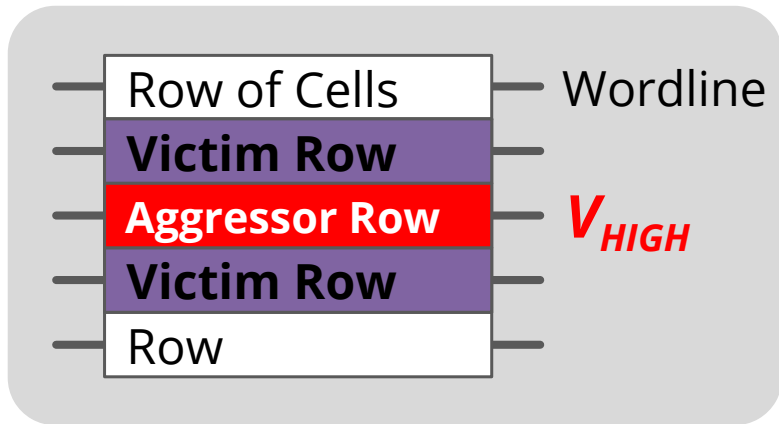
Kim+, "Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors," ISCA 2014.

Evidence of the DRAM Scaling Problem



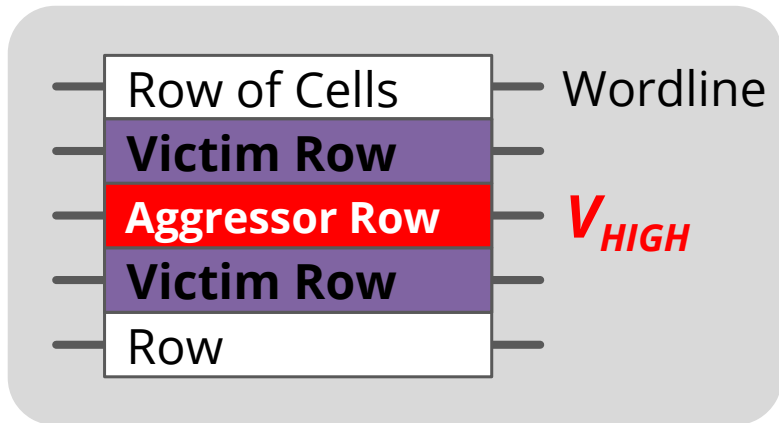
Kim+, "Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors," ISCA 2014.

Evidence of the DRAM Scaling Problem



Kim+, "Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors," ISCA 2014.

Evidence of the DRAM Scaling Problem



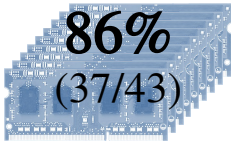
Repeatedly opening and closing a row enough times within a refresh interval induces **disturbance errors** in adjacent rows in **most real DRAM chips you can buy today**

Kim+, "Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors," ISCA 2014.

Most DRAM Modules Are At Risk

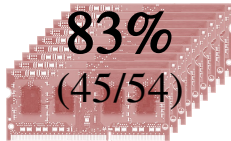
A

company



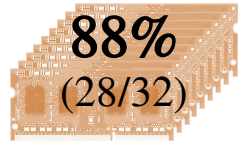
B

company



C

company

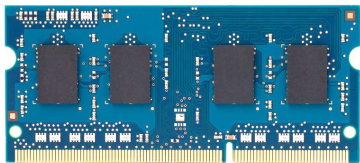


Up to
 1.0×10^7
errors

Up to
 2.7×10^6
errors

Up to
 3.3×10^5
errors

x86 CPU DRAM Module

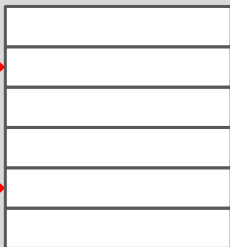


loop:

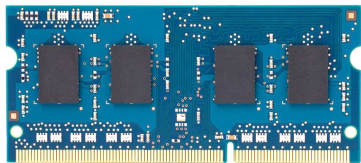
```
mov (X), %eax  
mov (Y), %ebx  
clflush (X)  
clflush (Y)  
mfence  
jmp loop
```

X →

Y →



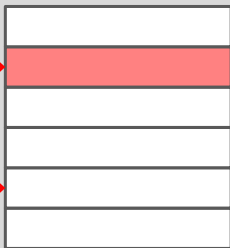
x86 CPU DRAM Module



loop:

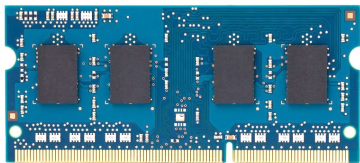
```
mov (X), %eax  
mov (Y), %ebx  
clflush (X)  
clflush (Y)  
mfence  
jmp loop
```

X →



Y →

x86 CPU DRAM Module

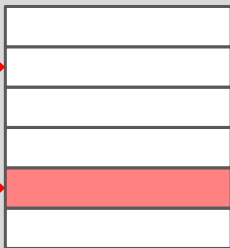


loop:

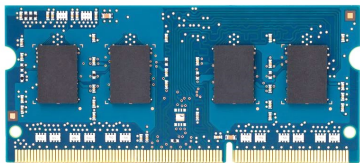
```
mov (X), %eax  
mov (Y), %ebx  
clflush (X)  
clflush (Y)  
mfence  
jmp loop
```

X →

Y →

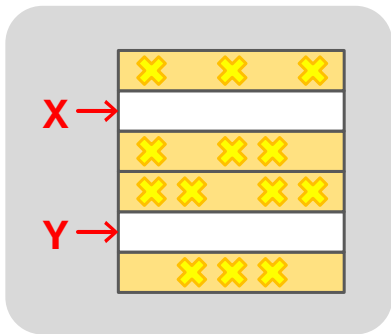


x86 CPU DRAM Module



loop:

```
mov (X), %eax  
mov (Y), %ebx  
clflush (X)  
clflush (Y)  
mfence  
jmp loop
```

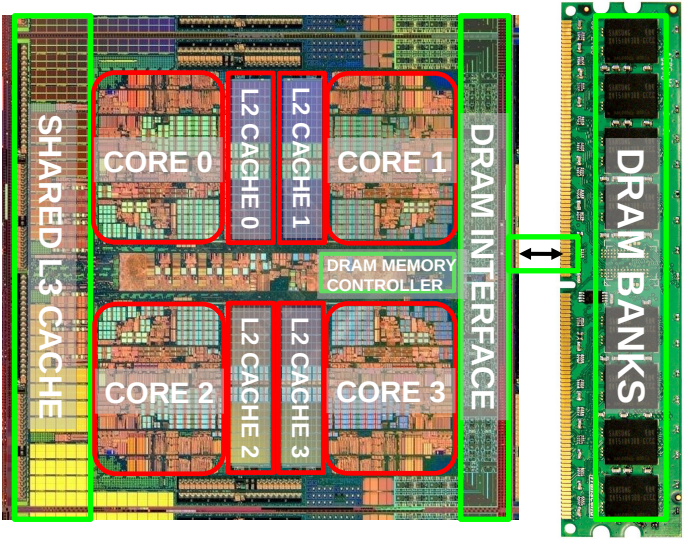


Participation Homework (25 points!)

Read: *Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors.*

Write: a 250 word summary of the paper and submit on Gradescope (as a .txt or PDF, typed, not handwritten).

Main Memory in the System



Main Memory in a Real System



This is Sumner's custom-built desktop. You can see the large cooling element on top of the CPU, the DRAM sticks to the right, and the NVMe SSD below.

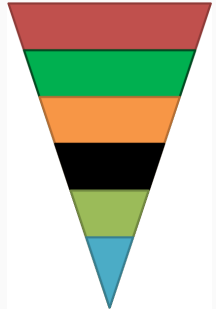
Some Fundamental Concepts

- **Physical address space**
 - Maximum size of main memory: total number of uniquely identifiable locations
- **Physical addressability**
 - Minimum size of data in memory that can be addressed
 - Examples: byte-addressable, word-addressable, 64-bit addressable
 - Microarchitectural addressability depends on the abstraction level of the implementation

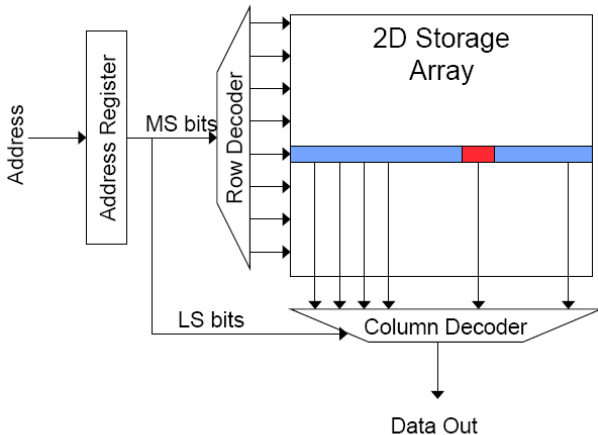
The DRAM Subsystem: A Bottom-Up View

DRAM Subsystem Organization

- Channel
- DIMM (Dual Inline Memory Module)
- Rank
- Chip
- Bank
- Row/Column



Memory Bank Organization



- Read access sequence:
 1. Decode row address & drive word-lines
 2. Selected bits drive bit-lines
 - Entire row read
 3. Amplify row data
 4. Decode column address & select subset of row
 - Send to output
 5. Precharge bit-lines
 - For next access

Interleaving

- Interleaving (banking)

- Problem: a single monolithic memory array takes long to access and does not enable multiple accesses in parallel
- Goal: Reduce the latency of memory array access and enable multiple accesses in parallel
- Idea: Divide the array into multiple banks that can be accessed independently (in the same cycle or in consecutive cycles)
 - Each bank is smaller than the entire memory storage
 - Accesses to different banks can be overlapped
- A Key Issue: How do you map data to different banks? (i.e., how do you interleave data across banks?)

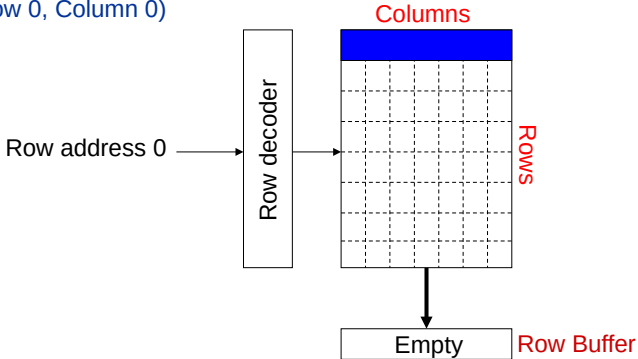
Page Mode DRAM

- A DRAM bank is a 2D array of cells: rows x columns
- A “DRAM row” is also called a “DRAM page”
- The activated row is put in a “row buffer”

- Each address is a <row,column> pair
- Access to a “closed row”
 - **Activate** command opens row (placed into row buffer)
 - **Read/write** command reads/writes column in the row buffer
 - **Precharge** command closes the row and prepares the bank for next access
- Access to an “open row”
 - No need for activate command

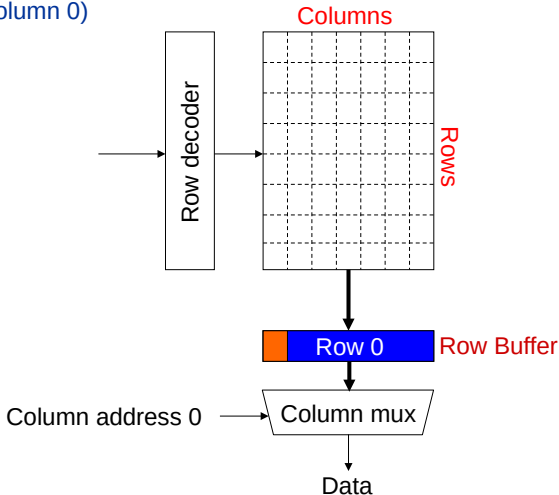
DRAM Bank Operation

Access Address:
(Row 0, Column 0)



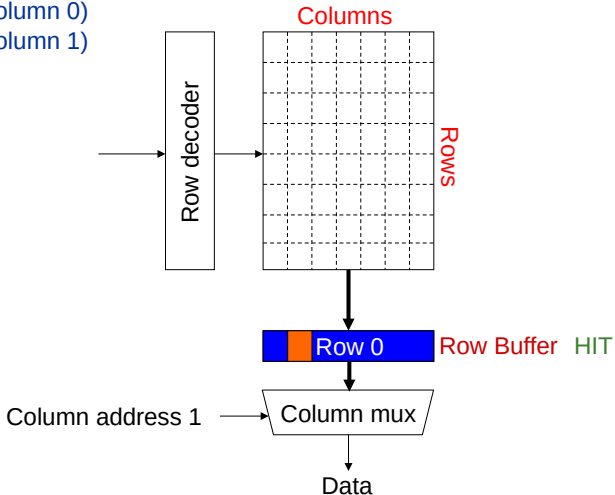
DRAM Bank Operation

Access Address:
(Row 0, Column 0)



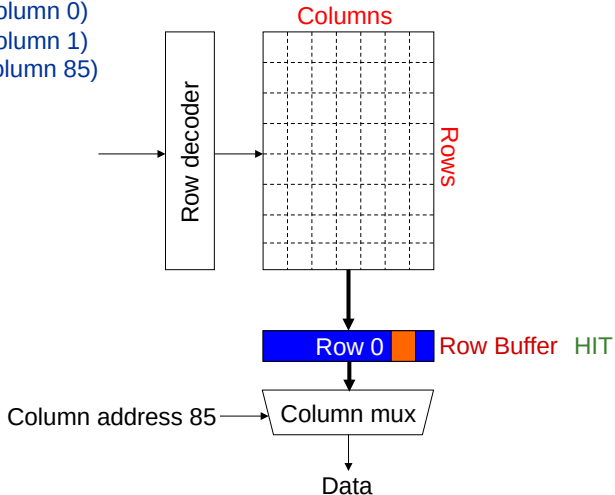
DRAM Bank Operation

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



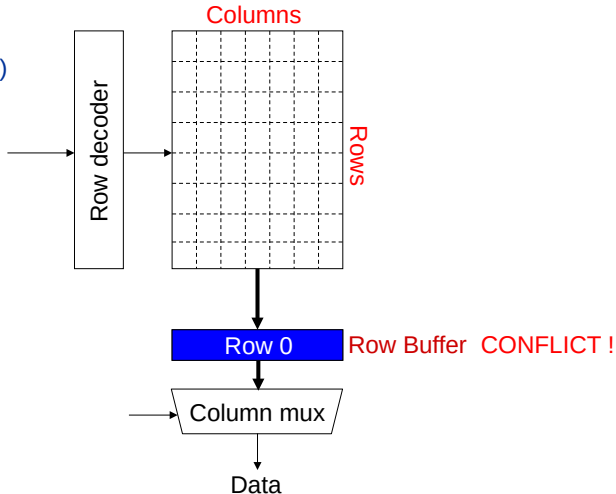
DRAM Bank Operation

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



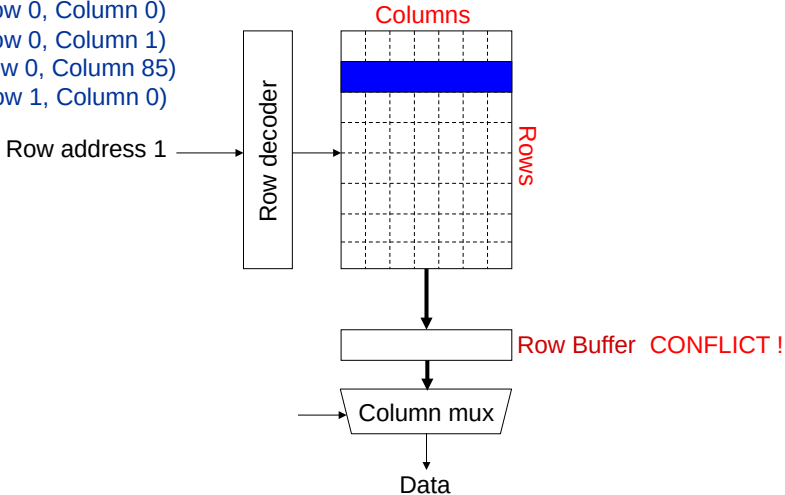
DRAM Bank Operation

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



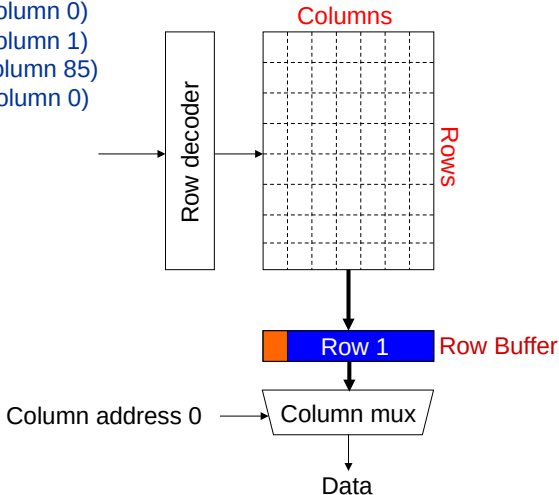
DRAM Bank Operation

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



DRAM Bank Operation

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



The DRAM Chip

- Consists of multiple *banks* (2-16 in DRAM)
- Banks share command, address, and data busses
- The chip itself has a narrow interface (4-16 bits per read)

The DRAM Rank and Module

A DRAM **rank** consists of multiple *chips* operated together to form a wide interface.

All chips comprising a rank are controlled *at the same time*.

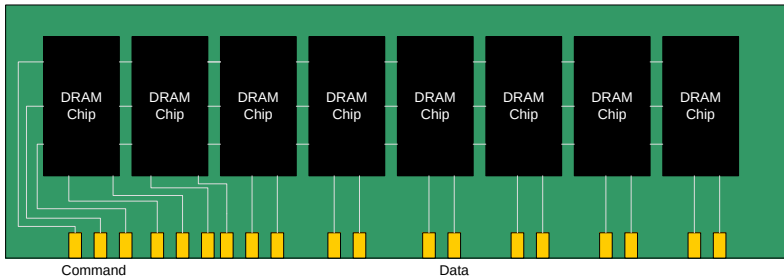
- They all respond to a single command.
- They share address and command buses, but provide different data.

A DRAM *module* consists of one or more ranks.

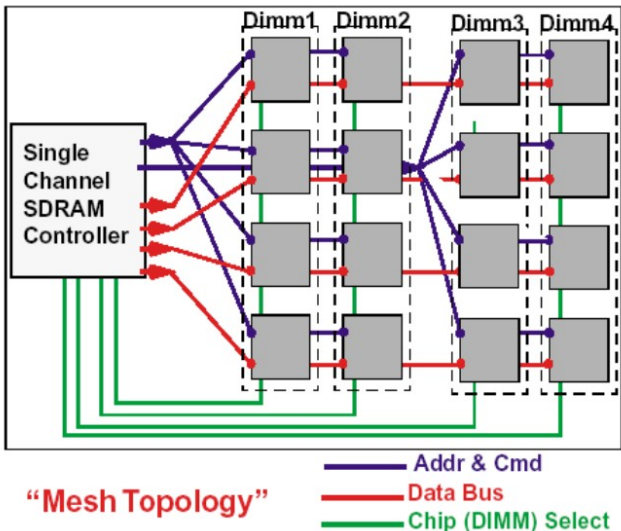
- For example DIMM (dual inline memory module)
- This is what you plug into your motherboard

If we have chips with 8-bit interfaces, to read 8 bytes in a single access, we would want 8 chips in a DIMM.

A 64-bit Wide DIMM (One Rank)

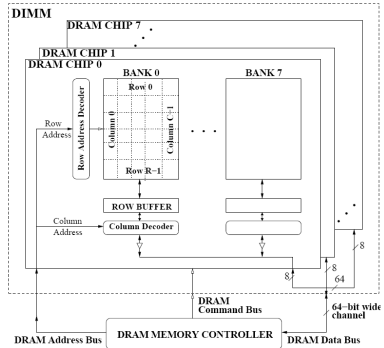
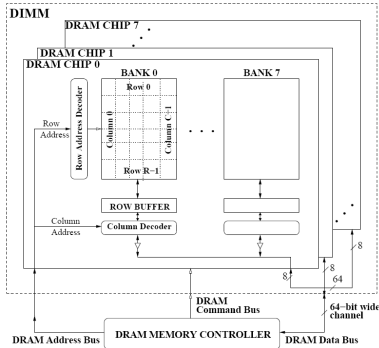


Multiple DIMMs



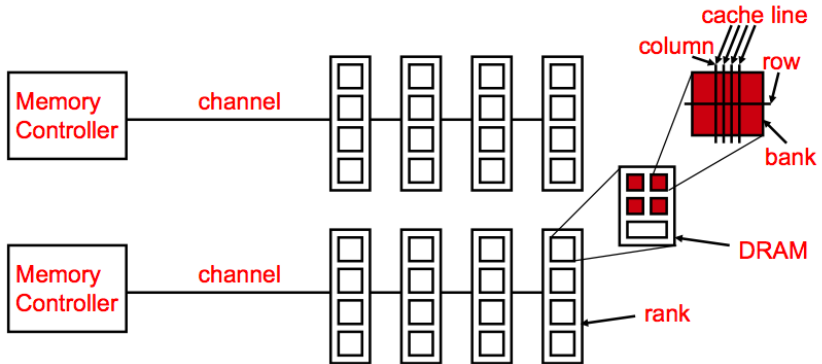
- Advantages:
 - Enables even higher capacity
- Disadvantages:
 - Interconnect complexity and energy consumption can be high

DRAM Channels

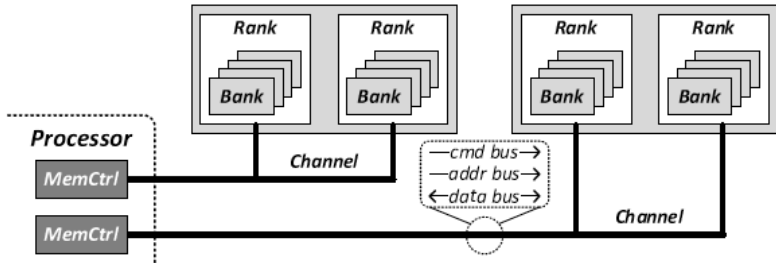


- 2 Independent Channels: 2 Memory Controllers (Above)

Generalized Memory Structure



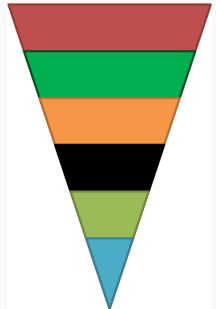
Generalized Memory Structure



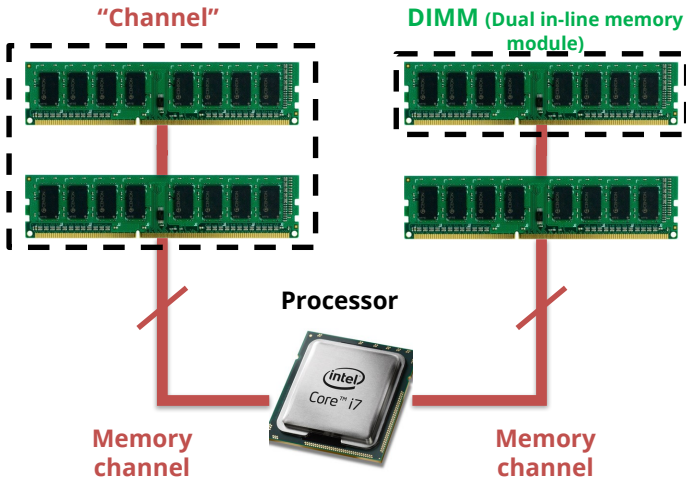
The DRAM Subsystem: A Top-Down View

DRAM Subsystem Organization

- Channel
- DIMM (Dual Inline Memory Module)
- Rank
- Chip
- Bank
- Row/Column

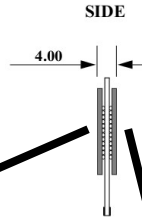
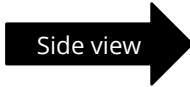


The DRAM subsystem

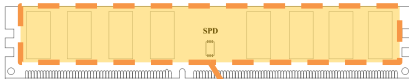


Breaking down a DIMM

DIMM (Dual in-line memory module)



Front of DIMM



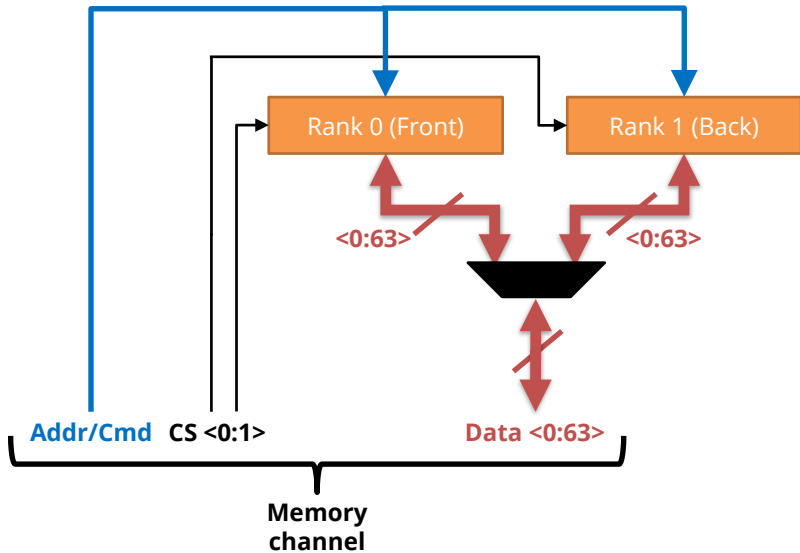
Rank 0: collection of 8 chips

Back of DIMM

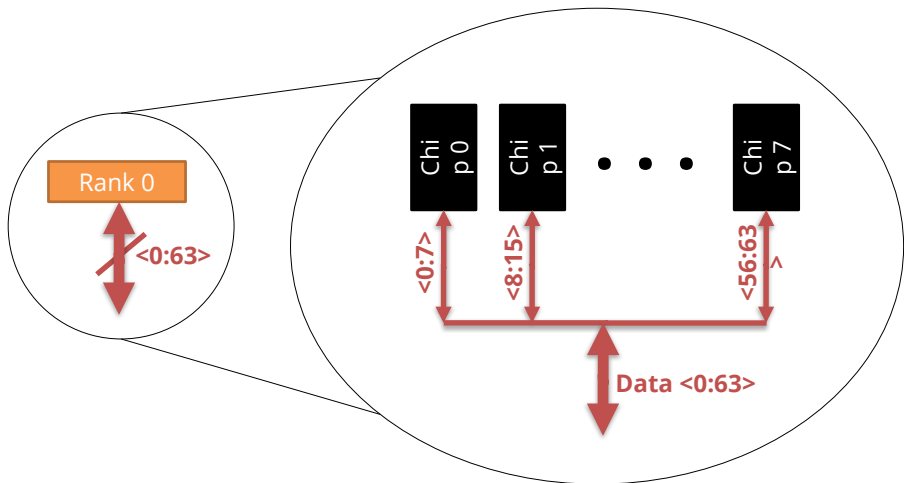


Rank 1

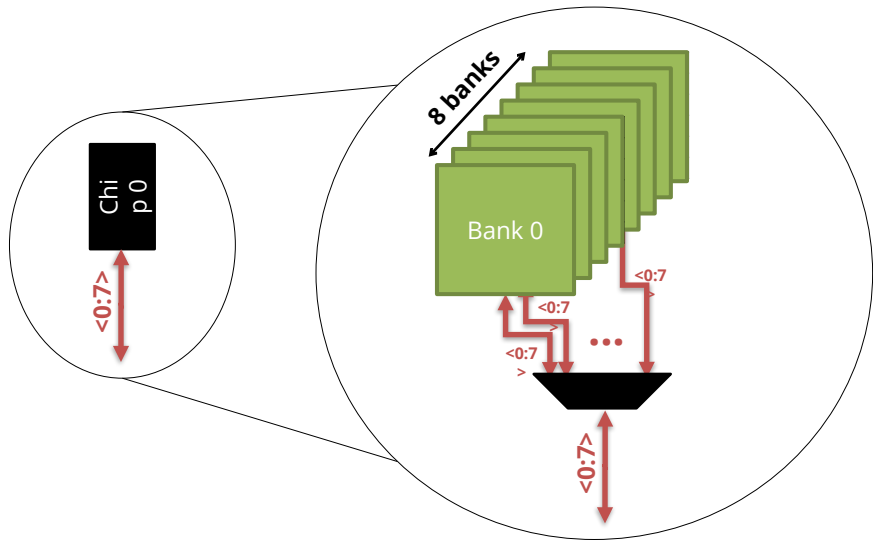
Rank



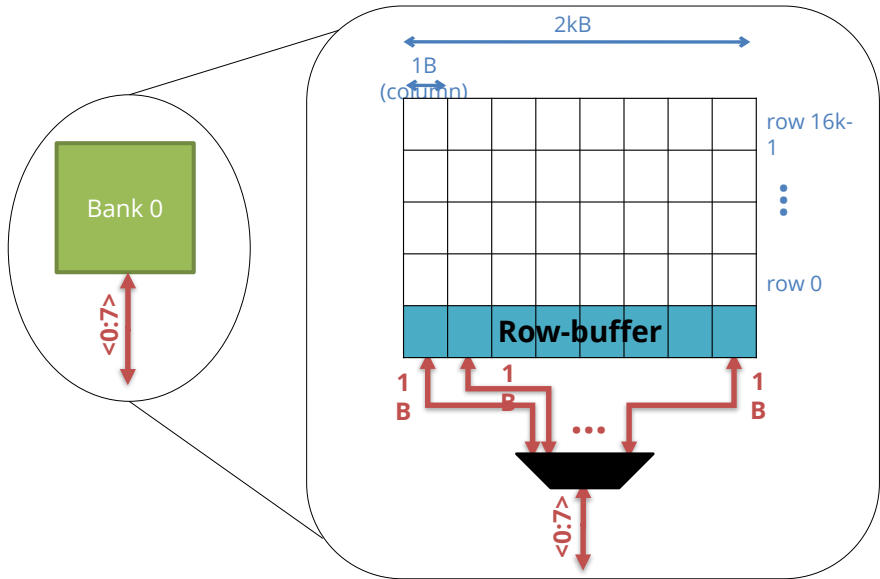
Breaking down a Rank



Breaking down a Chip

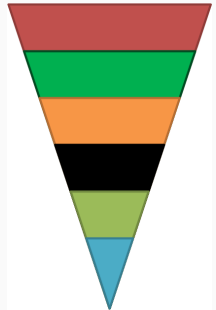


Breaking down a Bank



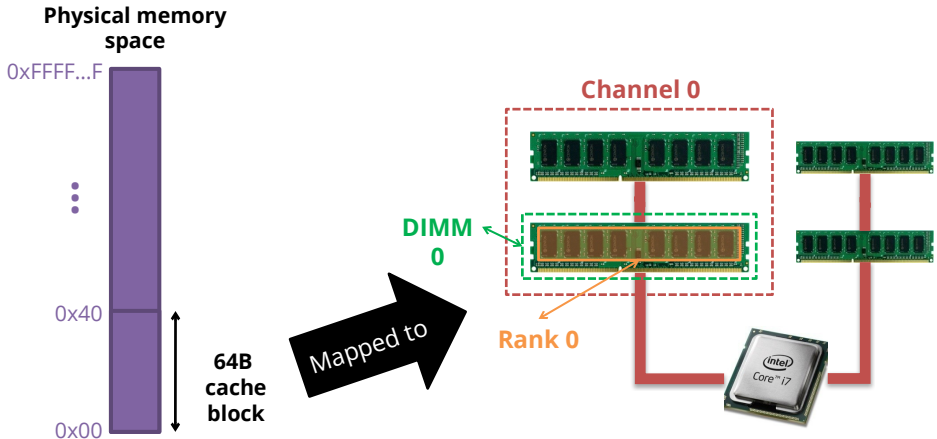
DRAM Subsystem Organization

- Channel
- DIMM (Dual Inline Memory Module)
- Rank
- Chip
- Bank
- Row/Column

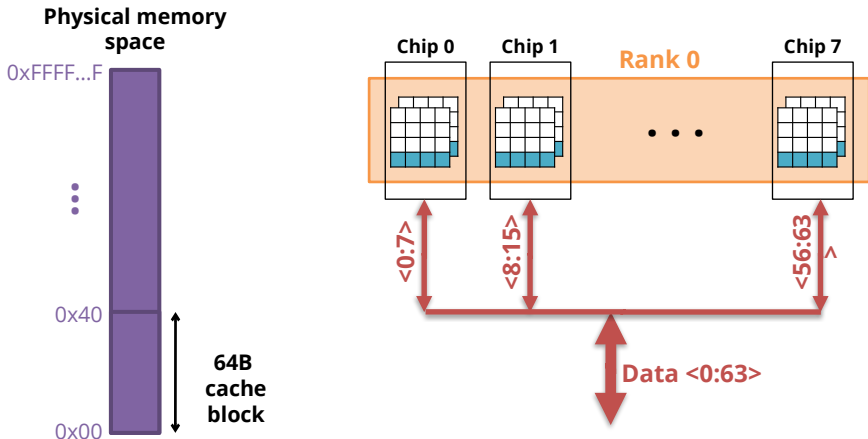


DRAM Subsystem Operation

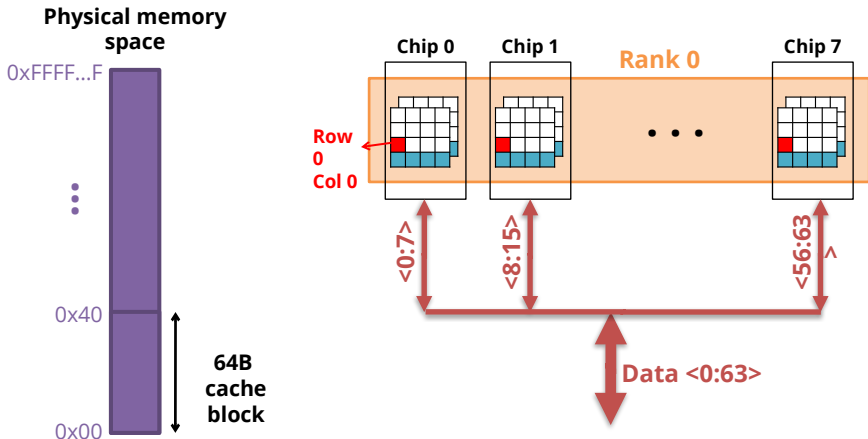
Example: Transferring a cache block



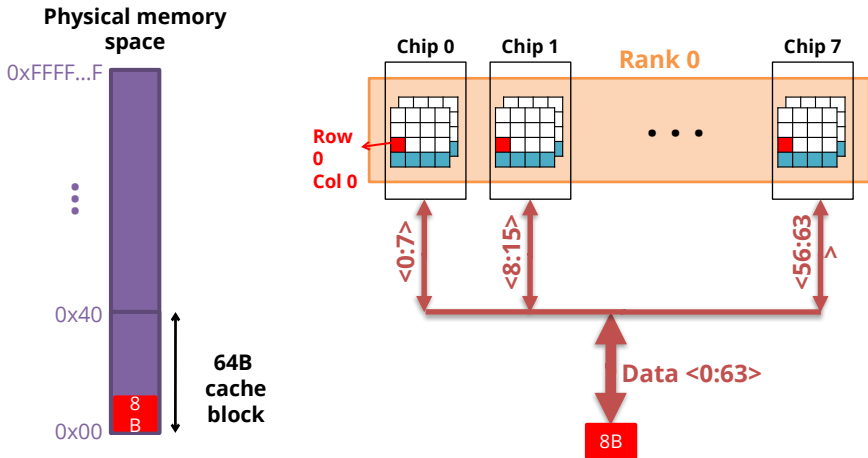
Example: Transferring a cache block



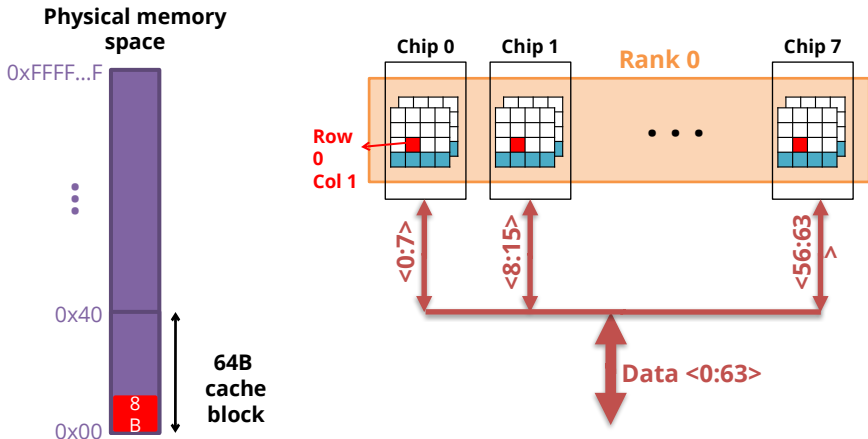
Example: Transferring a cache block



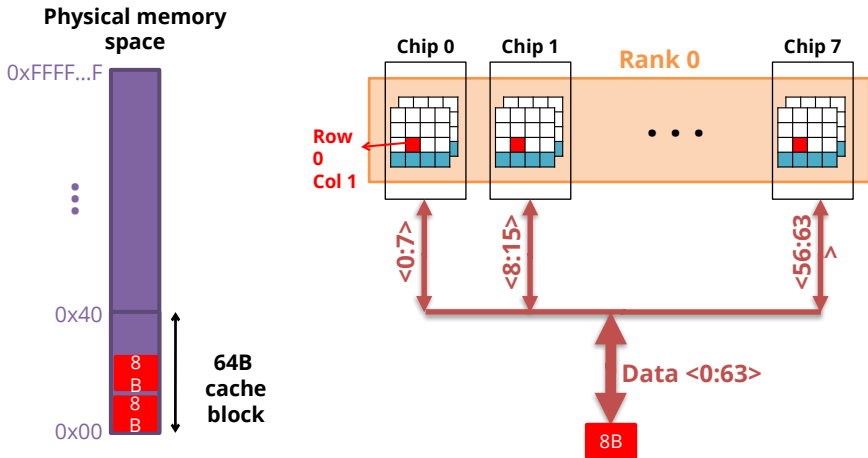
Example: Transferring a cache block



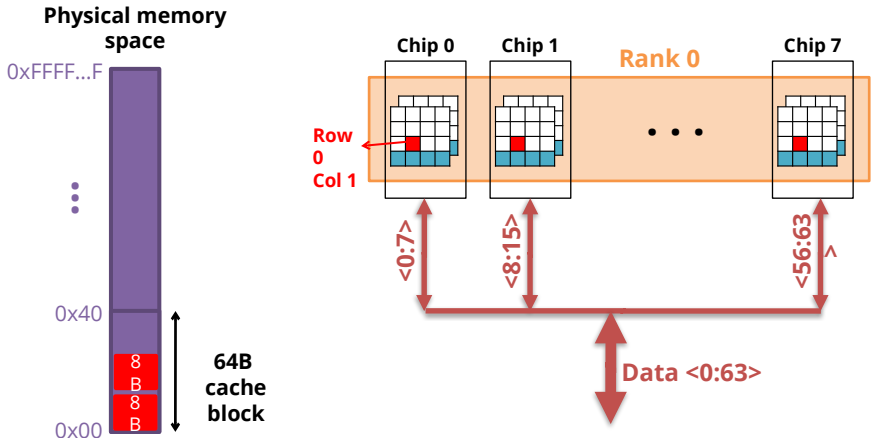
Example: Transferring a cache block



Example: Transferring a cache block



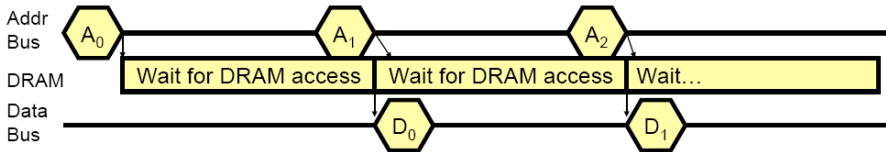
Example: Transferring a cache block



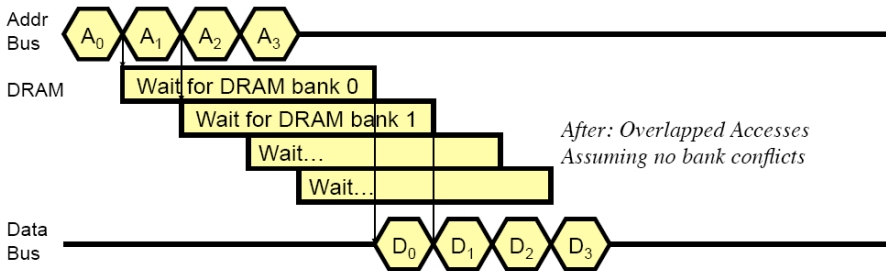
A 64B cache block takes 8 I/O cycles to transfer.

During the process, 8 columns are read sequentially.

How Multiple Banks/Channels Help

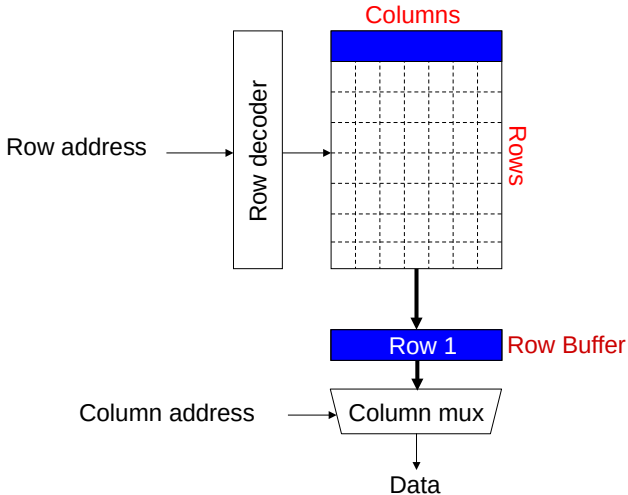


*Before: No Overlapping
Assuming accesses to different DRAM rows*



*After: Overlapped Accesses
Assuming no bank conflicts*

DRAM Bank Operation



DRAM Scheduling

Row Buffer Management Policy

Row buffers can act as a cache within DRAM.

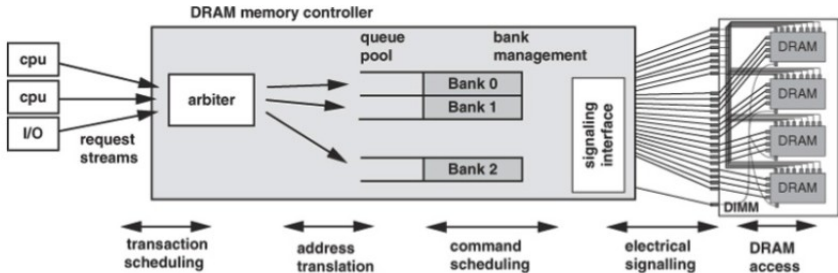
- **Open Row:** keep the row open after access
 - **Pro:** next access might need the same row → row hit
 - **Con:** next access might need a different row → row conflict
- **Closed Row:** close the row open after an access
 - **Pro:** next access might need a different row → no row conflict
 - **Con:** next access might need the same row → extra activation

Practice: DRAM Operation: Worksheet Problem 2

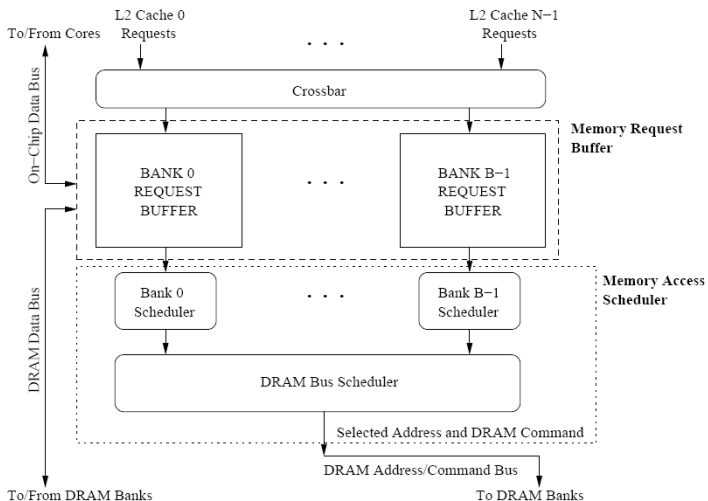
- **Row buffer hit:** only need to move data from row buffer to pins (~20 ns access time)
- **Empty row buffer access:** must first read read the row, the move data from row buffer to pins (~40 ns access time)
- **Row buffer conflict:** must first precharge the bitlines, then read the other row, the move data from row buffer to pins (~60 ns access time)

Requested	Time of Arrival	Time of Service	
		Open	Closed
X	0		
Y	10		
X + 1	100		
X + 2	200		
Y + 1	250		
X + 3	300		

A Modern DRAM Controller (I)



A Modern DRAM Controller (II)



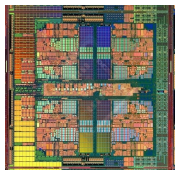
DRAM Scheduling Policies

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

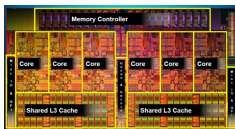
Goal: maximize row buffer hit rate → maximize DRAM throughput

Trend: Many Cores on Chip

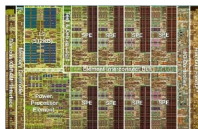
- Simpler and lower power than a single large core
- Large scale parallelism on chip



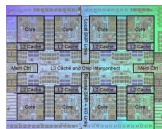
AMD Barcelona
4 cores



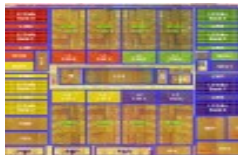
Intel Core i7
8 cores



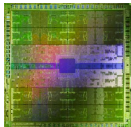
IBM Cell BE
8+1 cores



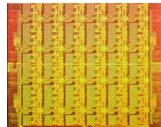
IBM POWER7
8 cores



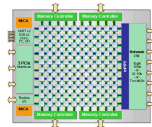
Sun Niagara II
8 cores



Nvidia Fermi
448 "cores"



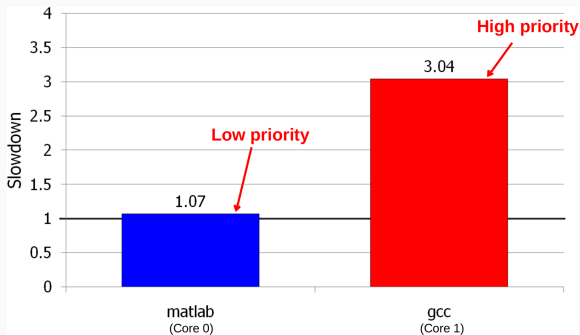
Intel SCC
48 cores, networked



Tiler TILE Gx
100 cores, networked

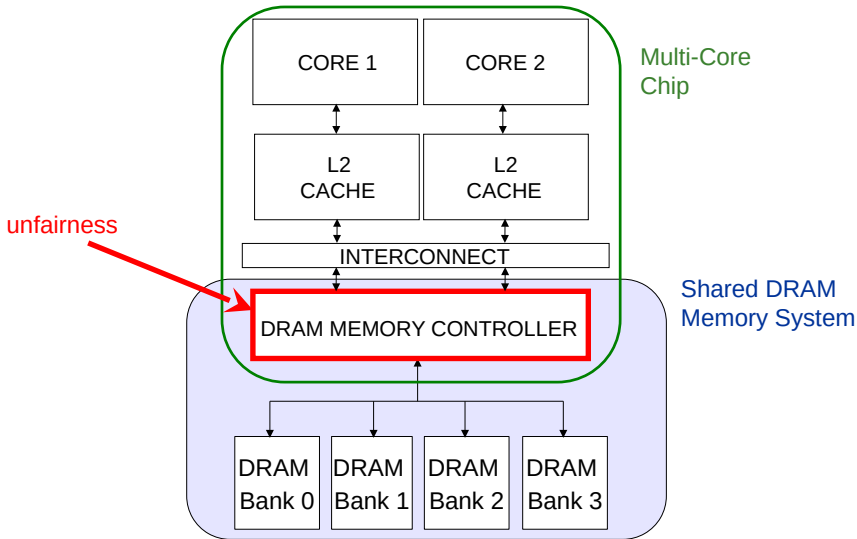
(Un)expected Slowdowns with Multi-Core Systems

- What we want:
 - N times the system performance with N times the cores.
- What do we get today?

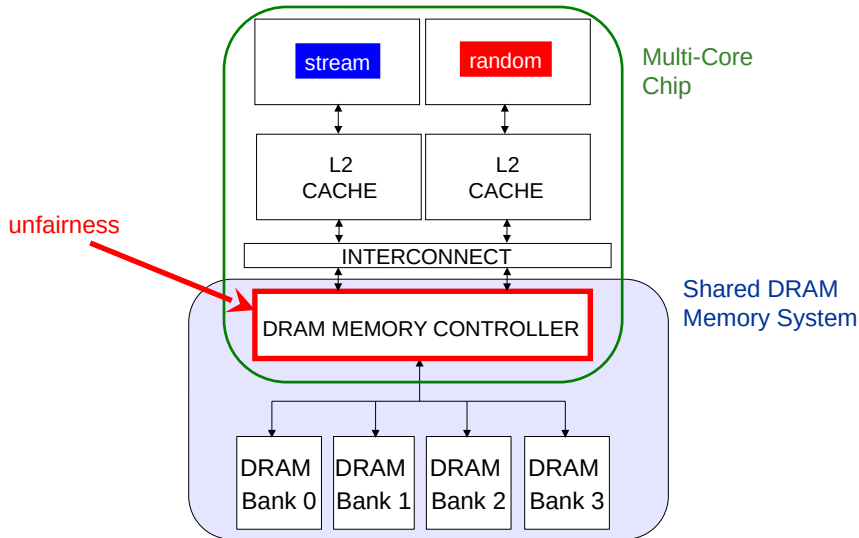


Moscibroda and Mutlu, *Memory performance attacks: Denial of memory service in multi-core systems*, USENIX Security 2007.

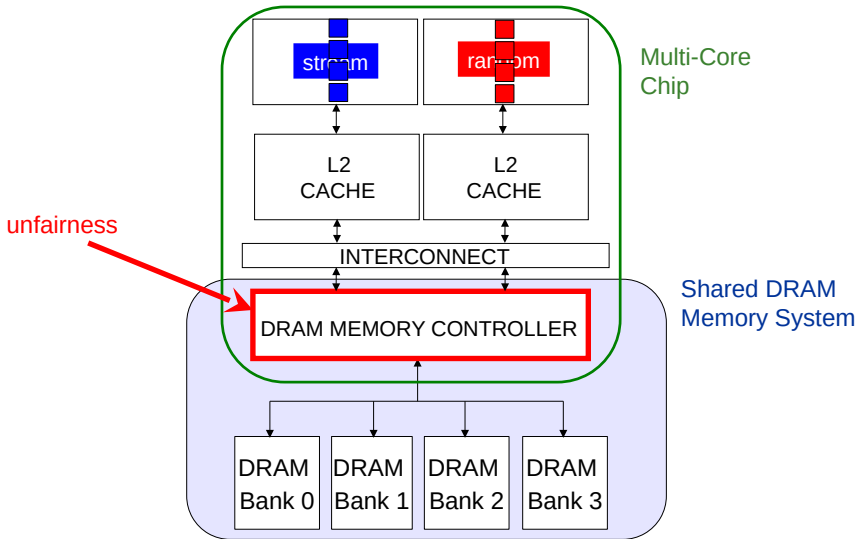
Uncontrolled Interference: An Example



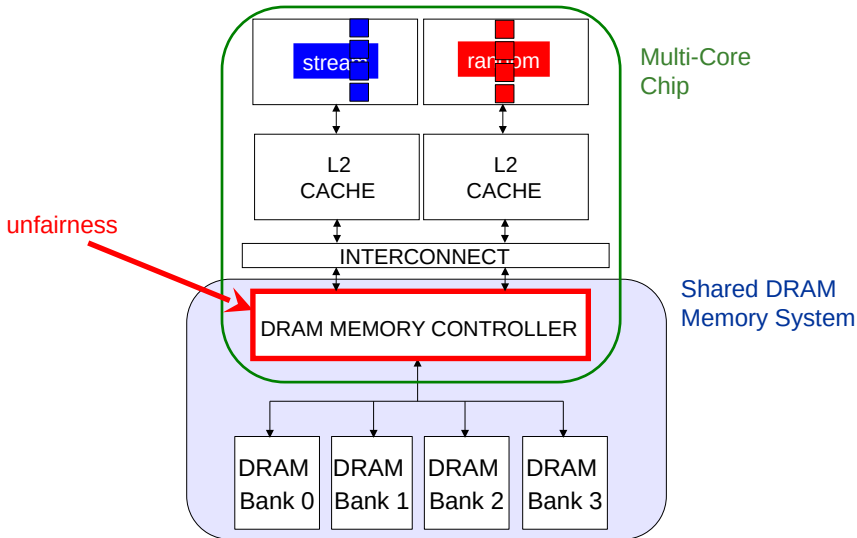
Uncontrolled Interference: An Example



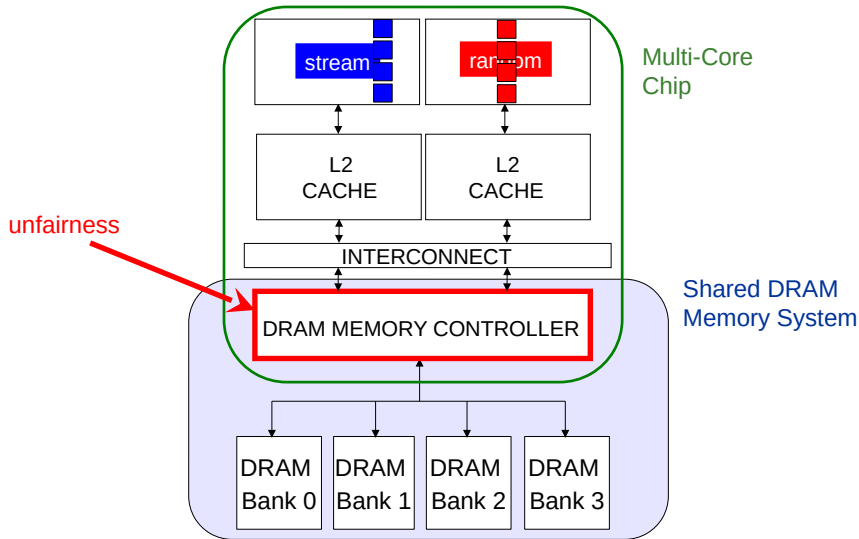
Uncontrolled Interference: An Example



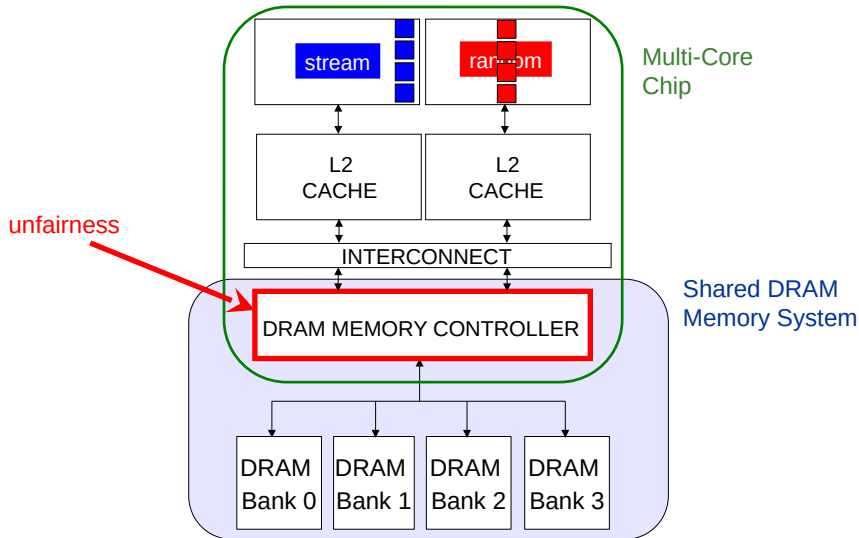
Uncontrolled Interference: An Example



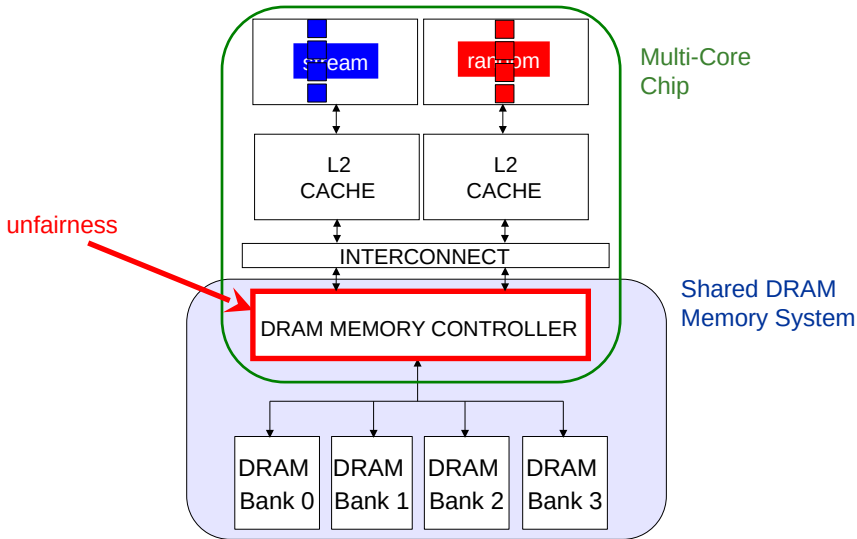
Uncontrolled Interference: An Example



Uncontrolled Interference: An Example



Uncontrolled Interference: An Example



A Memory Performance Hog

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

STREAM

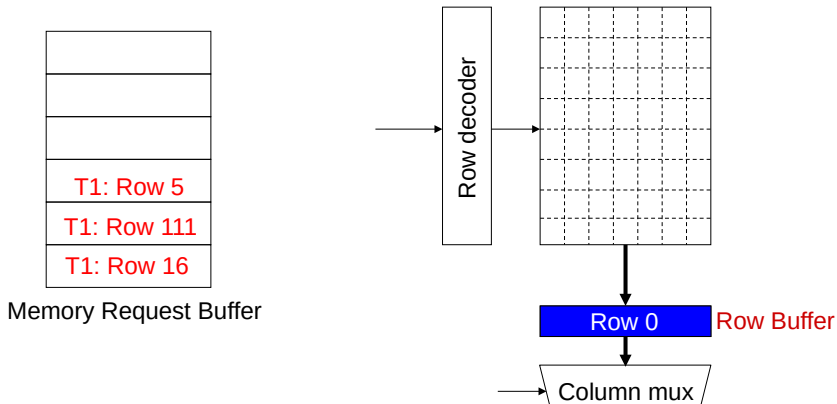
- Sequential memory access
- Very high row buffer locality (96% hit rate)
- Memory intensive

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

RANDOM

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

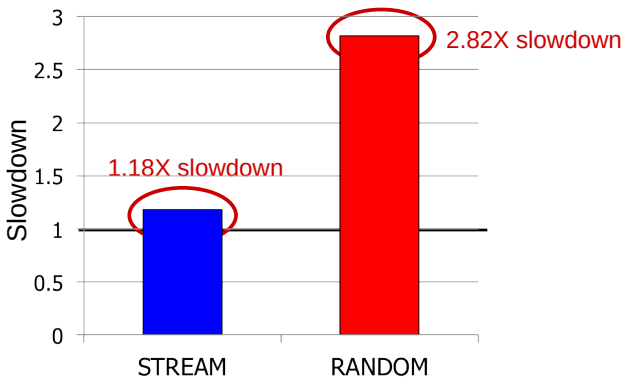
What Does the Memory Hog Do?



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

Moscibroda and Mutlu, "[Memory Performance Attacks](#)," USENIX Security 2007.

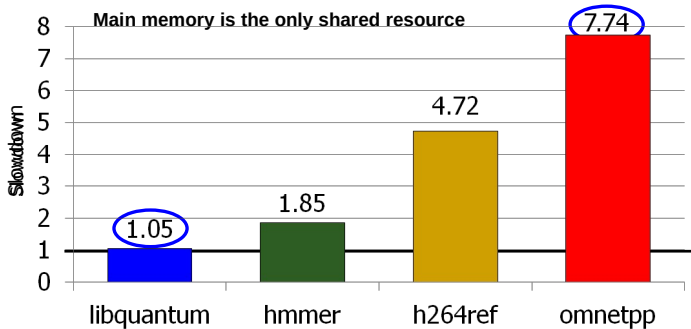
Effect of Memory Performance Hog



Results on Intel Pentium D running Windows XP
(Similar results for Intel Core Duo and AMD Turion, and on Fedora Linux)

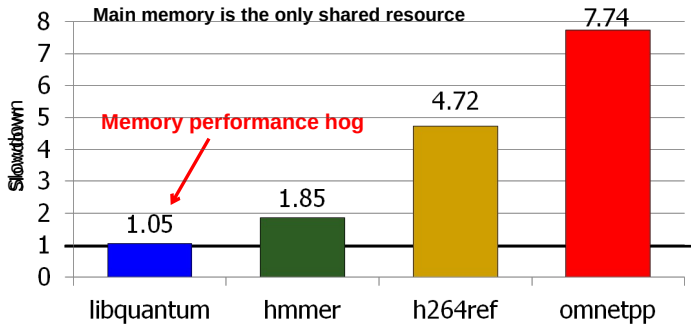
Moscibroda and Mutlu, "[Memory Performance Attacks](#)," USENIX Security 2007.

Problems due to Uncontrolled Interference



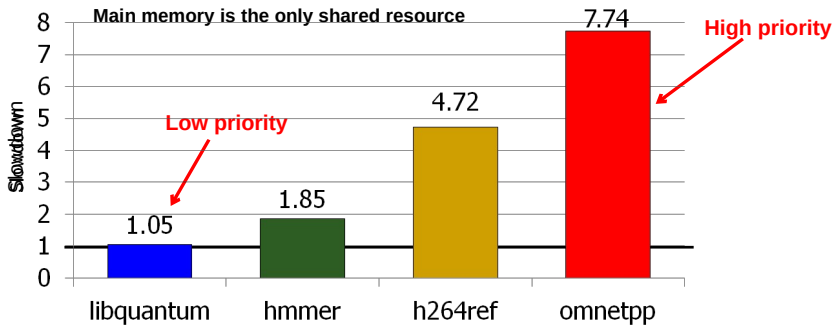
- Unfair slowdown of different threads

Problems due to Uncontrolled Interference



- Unfair slowdown of different threads

Problems due to Uncontrolled Interference



- Unfair slowdown of different threads
- Priority inversion: unable to enforce priorities/SLAs

How do we Solve the Problem?

- Stall-time fair memory (STFM) 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.

Mutlu and Moscibroda, *Stall-Time Fair Memory Access Scheduling for Chip Multiprocessors*, MICRO 2007.

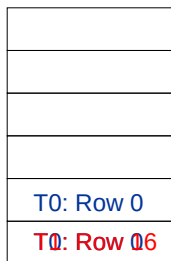
Stall-Time Fairness in Shared DRAM Systems

- 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_{shared} — DRAM-related stall-time when the thread runs with other threads
- ST_{alone} — DRAM-related stall-time when the thread runs with other threads
- Memory-slowdown = $ST_{\text{shared}}/ST_{\text{alone}}$ — relative increase in stall time
- The *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

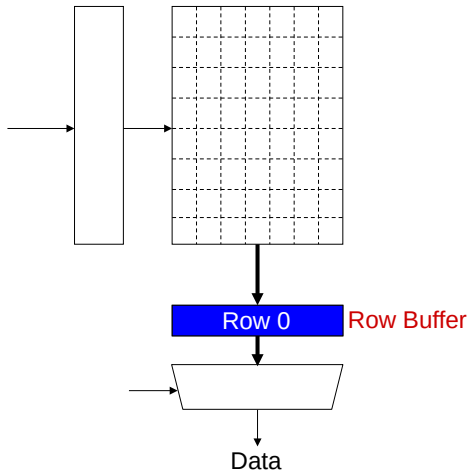
STFM Scheduling Algorithm

- For each thread, the DRAM controller
 - Keeps track of ST_{shared}
 - Estimates ST_{alone}
- Each cycle, the DRAM controller
 - Computes memory slowdown for threads with legal requests
 - Computes **unfairness = max(slowdowns) / min(slowdowns)**
- If $\text{unfairness} < \alpha$, use DRAM throughput-oriented scheduling policy
- If $\text{unfairness} > \alpha$, use fairness-oriented scheduling policy
 1. requests from thread with maximum slowdown first
 2. row-hit first
 3. oldest-first

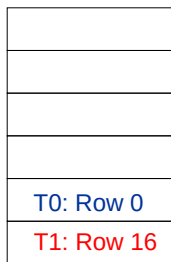
How Does STFM Prevent Unfairness?



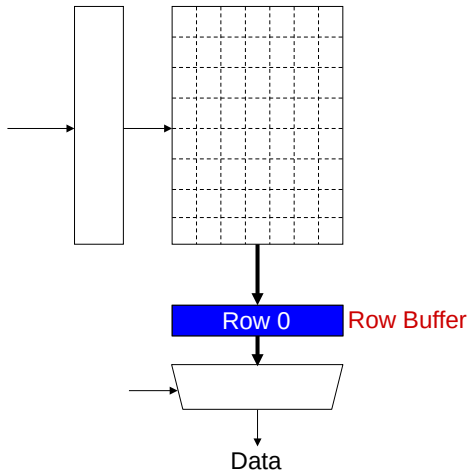
T0 Slowdown	1.00
T1 Slowdown	1.00
Unfairness	1.00
α	1.05



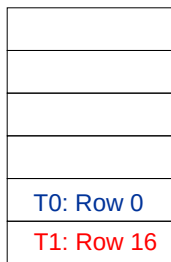
How Does STFM Prevent Unfairness?



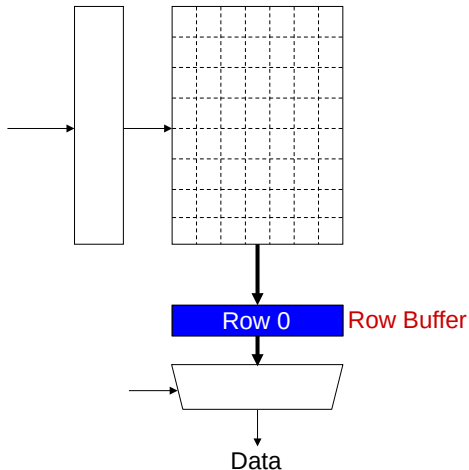
T0 Slowdown	1.00
T1 Slowdown	1.00
Unfairness	1.00
α	1.05



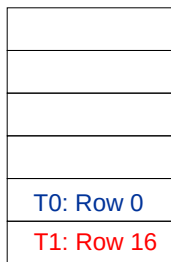
How Does STFM Prevent Unfairness?



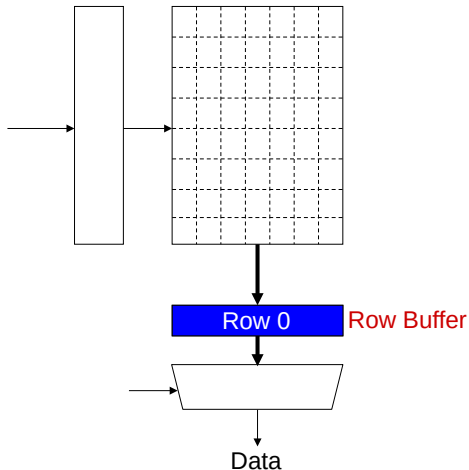
T0 Slowdown	1.00
T1 Slowdown	1.00
Unfairness	1.00
α	1.05



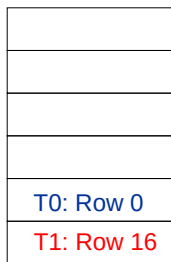
How Does STFM Prevent Unfairness?



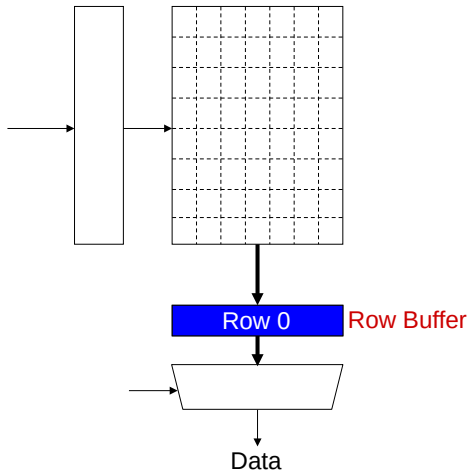
T0 Slowdown	1.00
T1 Slowdown	1.03
Unfairness	1.03
α	1.05



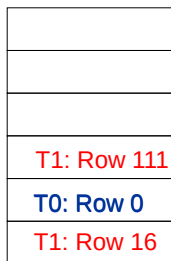
How Does STFM Prevent Unfairness?



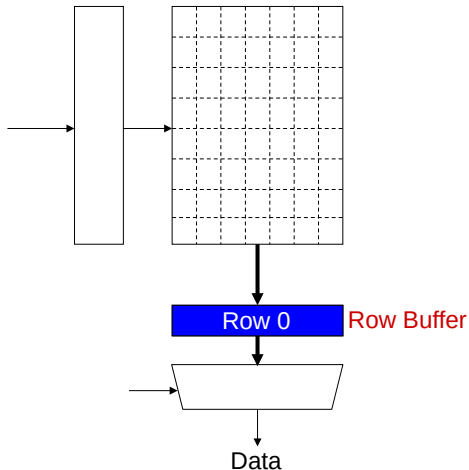
T0 Slowdown	1.00
T1 Slowdown	1.03
Unfairness	1.03
α	1.05



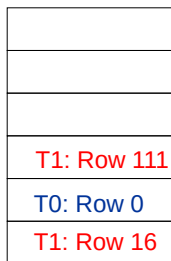
How Does STFM Prevent Unfairness?



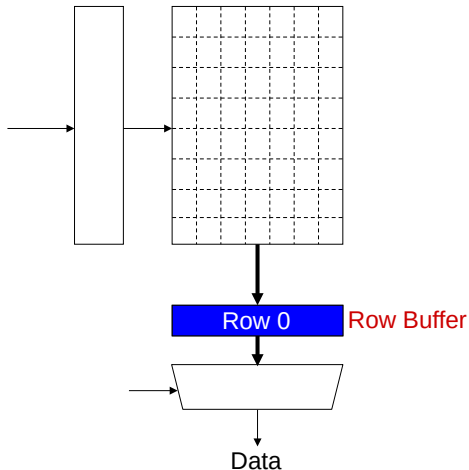
T0 Slowdown	1.00
T1 Slowdown	1.06
Unfairness	1.06
α	1.05



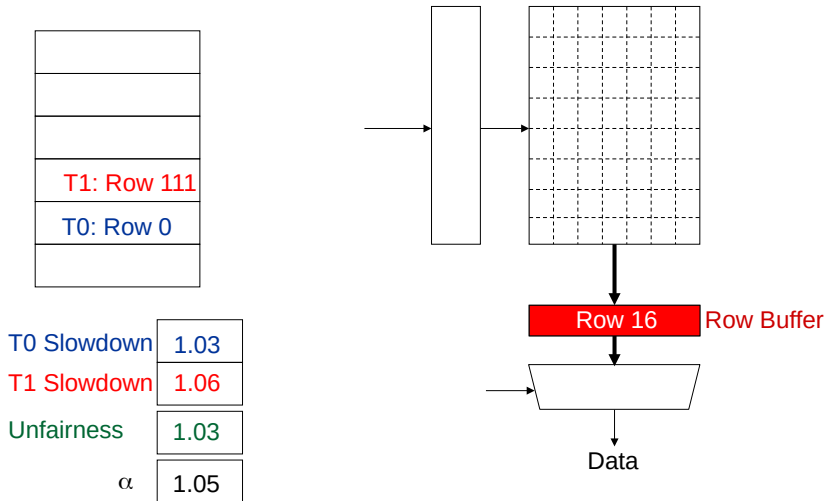
How Does STFM Prevent Unfairness?



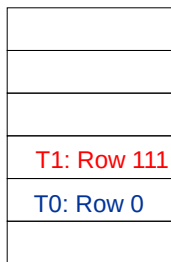
T0 Slowdown	1.00
T1 Slowdown	1.06
Unfairness	1.06
α	1.05



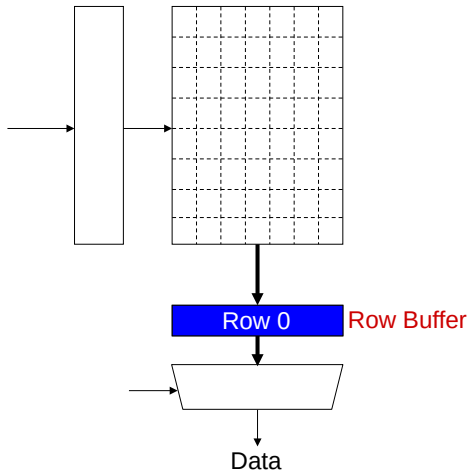
How Does STFM Prevent Unfairness?



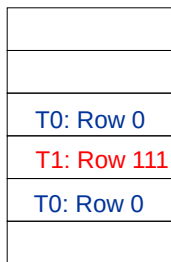
How Does STFM Prevent Unfairness?



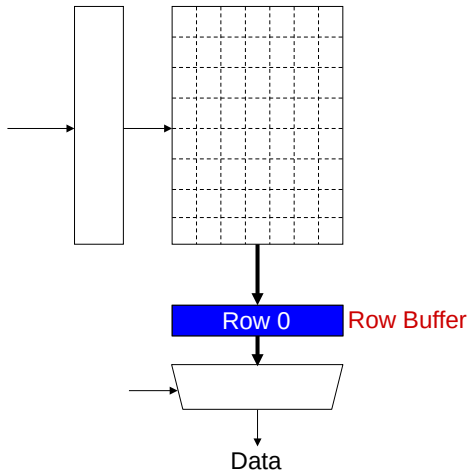
T0 Slowdown	1.03
T1 Slowdown	1.06
Unfairness	1.03
α	1.05



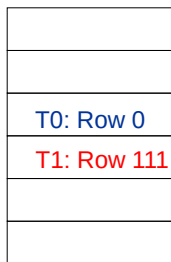
How Does STFM Prevent Unfairness?



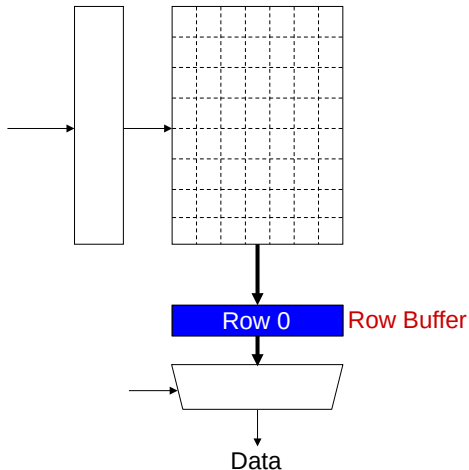
T0 Slowdown	1.04
T1 Slowdown	1.08
Unfairness	1.04
α	1.05



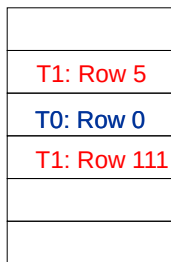
How Does STFM Prevent Unfairness?



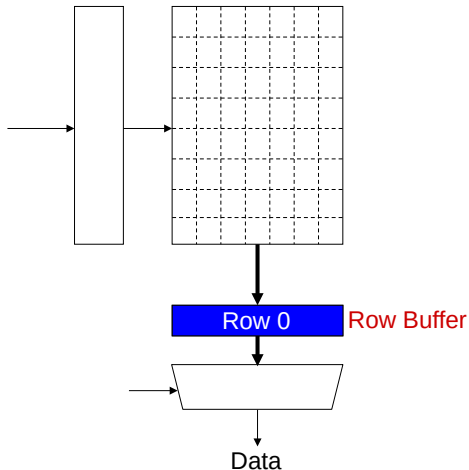
T0 Slowdown	1.04
T1 Slowdown	1.08
Unfairness	1.04
α	1.05



How Does STFM Prevent Unfairness?



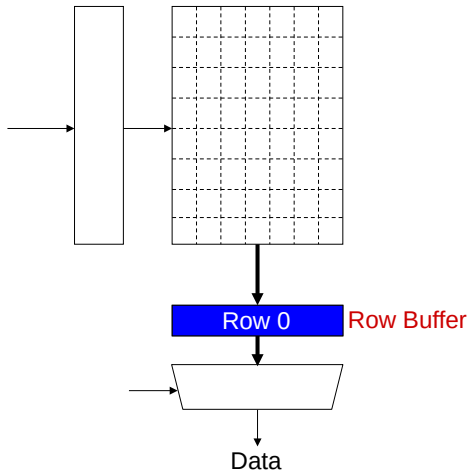
T0 Slowdown	1.04
T1 Slowdown	1.11
Unfairness	1.06
α	1.05



How Does STFM Prevent Unfairness?

T0: Row 0
T1: Row 5
T0: Row 0
T1: Row 111

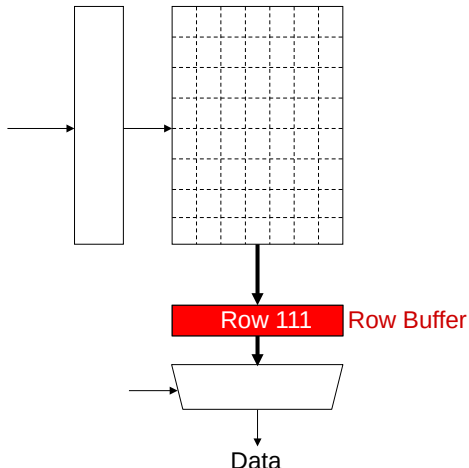
T0 Slowdown	1.04
T1 Slowdown	1.11
Unfairness	1.06
α	1.05



How Does STFM Prevent Unfairness?

T0: Row 0
T1: Row 5
T0: Row 0
T1: Row 111

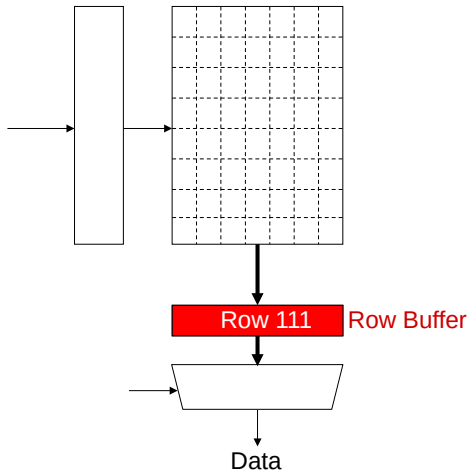
T0 Slowdown	1.04
T1 Slowdown	1.11
Unfairness	1.06
α	1.05



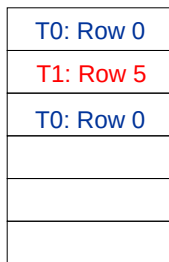
How Does STFM Prevent Unfairness?

T0: Row 0
T1: Row 5
T0: Row 0

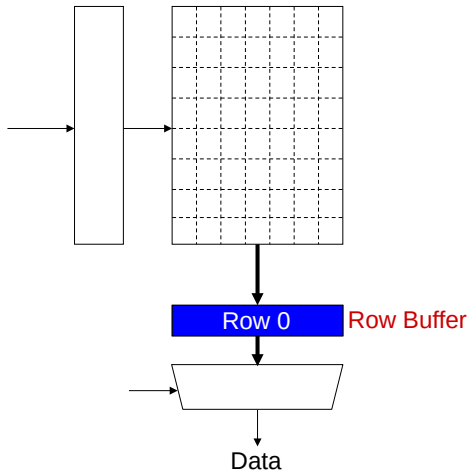
T0 Slowdown	1.07
T1 Slowdown	1.11
Unfairness	1.04
α	1.05



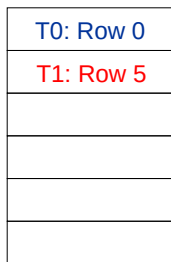
How Does STFM Prevent Unfairness?



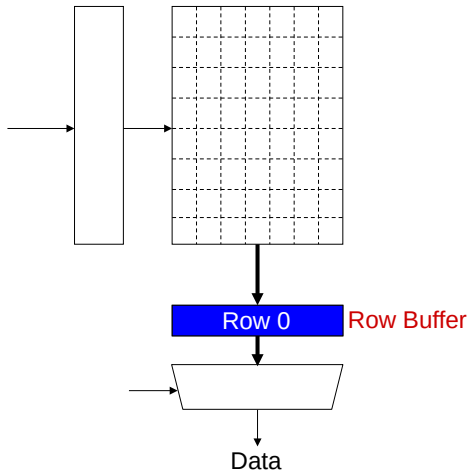
T0 Slowdown	1.07
T1 Slowdown	1.11
Unfairness	1.04
α	1.05



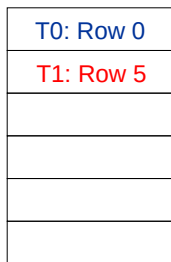
How Does STFM Prevent Unfairness?



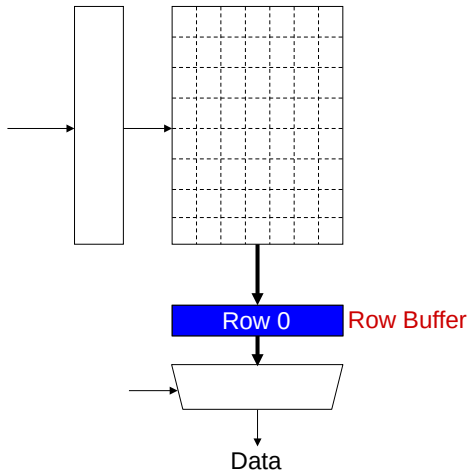
T0 Slowdown	1.10
T1 Slowdown	1.14
Unfairness	1.03
α	1.05



How Does STFM Prevent Unfairness?



T0 Slowdown	1.10
T1 Slowdown	1.14
Unfairness	1.03
α	1.05



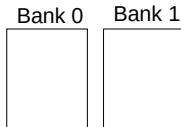
Another Problem due to Interference

- 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

Bank Parallelism of a Thread

Single Thread:

Thread A : Compute



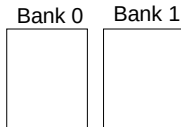
Bank Parallelism Interference

in DRAM

Baseline Scheduler:

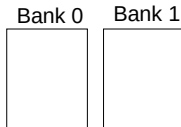
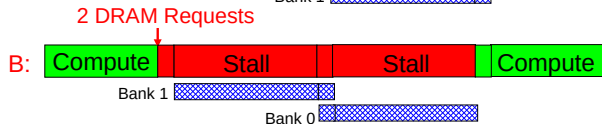
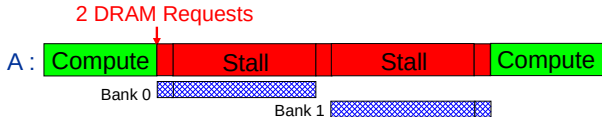
A: Compute

B: Compute



Parallelism-Aware Scheduler

Baseline Scheduler:



Parallelism-aware Scheduler:

A:

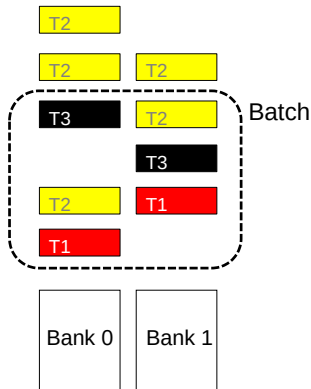
B:

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...

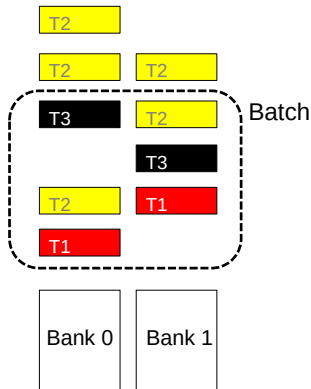
Parallelism-Aware Batch Scheduling (PAR-BS)

- 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



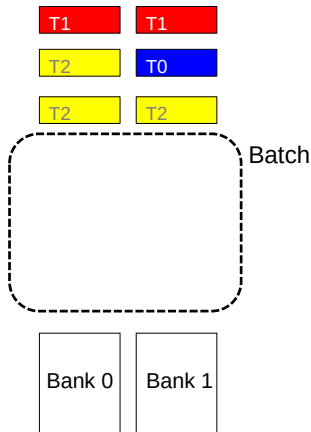
Parallelism-Aware Batch Scheduling (PAR-BS)

- 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



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

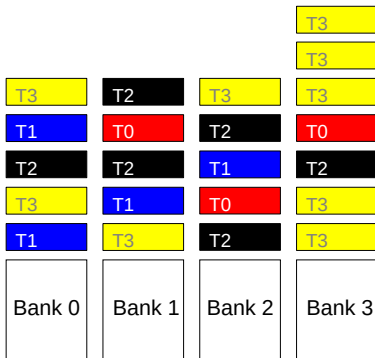
- Can use any DRAM scheduling policy
 - FR-FCFS (row-hit first, then oldest-first) exploits row-buffer locality
- But, we also want to preserve intra-thread bank parallelism
 - Service each thread's requests back to back

How to Rank Threads within a Batch

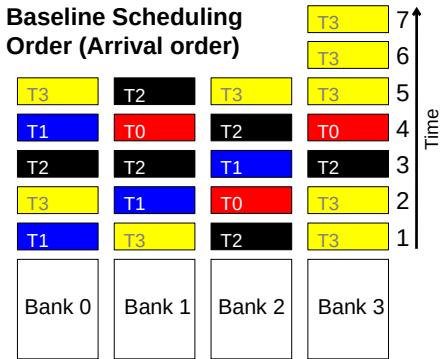
- Ranking scheme affects system throughput and fairness
- Maximize system throughput

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

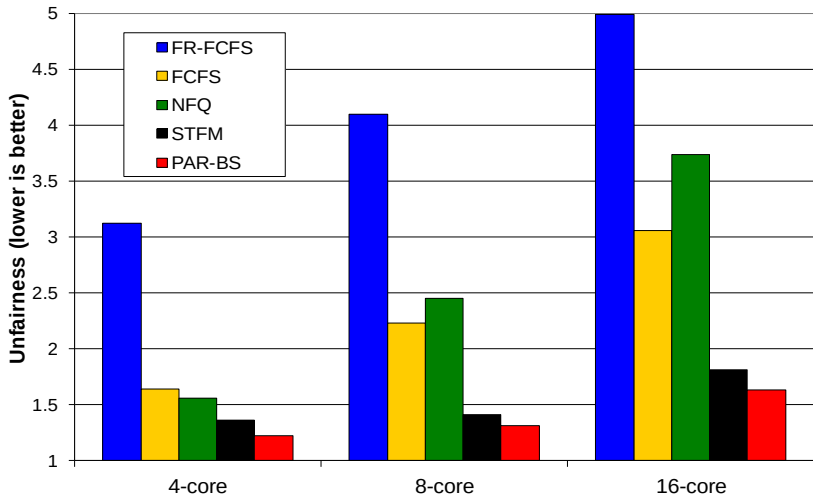


Example Within-Batch Scheduling Order

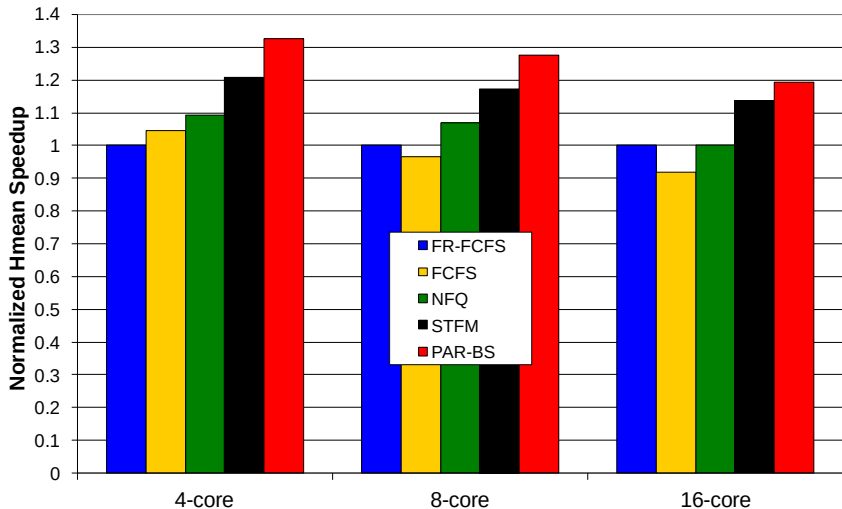


Unfairness on 4-, 8-, 16-core Systems

Unfairness = MAX Memory Slowdown / MIN Memory Slowdown [MICRO 2007]



System Performance



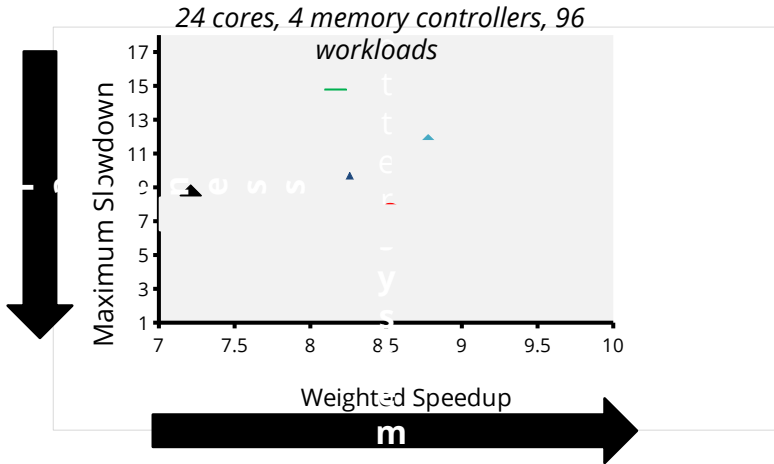
PAR-BS Pros and Cons

- Upsides:

PAR-BS Pros and Cons

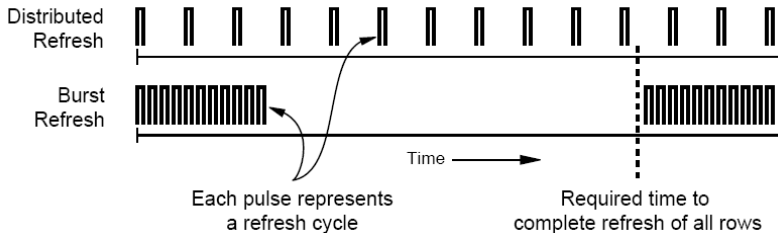
- Upsides:
 - First scheduler to address bank parallelism destruction across multiple threads

Throughput vs. Fairness



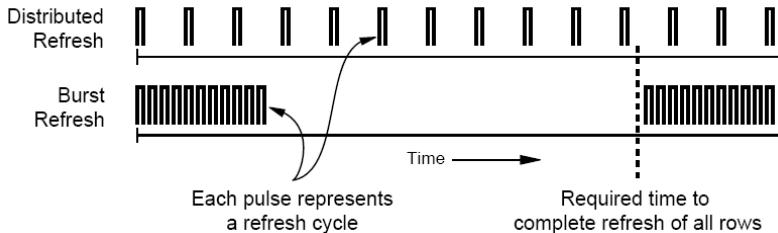
DRAM Refresh

Distributed Refresh



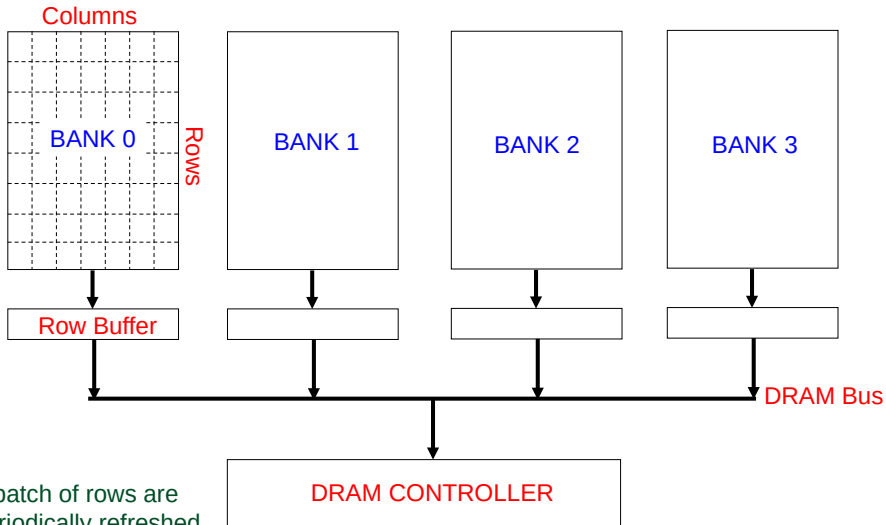
- Distributed refresh eliminates long pause times

Distributed Refresh



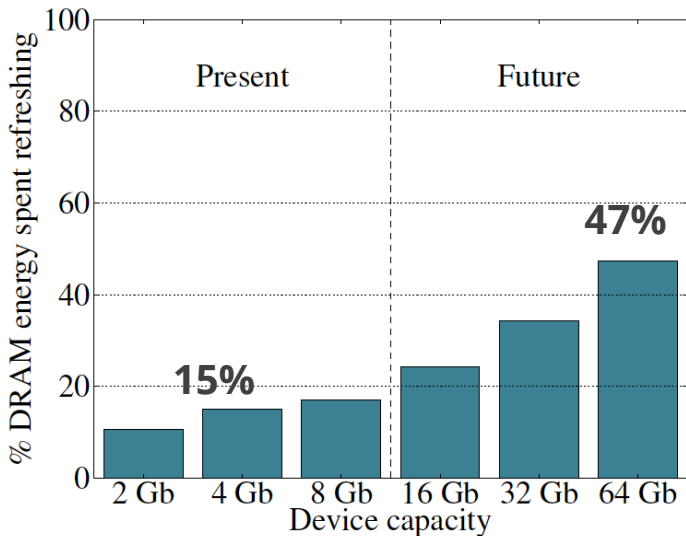
- Distributed refresh eliminates long pause times
- Does distributed refresh reduce refresh impact on energy?

Refresh Today: Auto Refresh



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

Refresh Overhead: Energy





Problem with Conventional Refresh

Problem with Conventional Refresh

- Today: Every row is refreshed at the same rate

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 items  Low HW overhead to achieve large reductions in refresh operations
- Liu et al., "RAIDR: Retention-Aware Intelligent DRAM Refresh," ISCA 2012.