

18-447

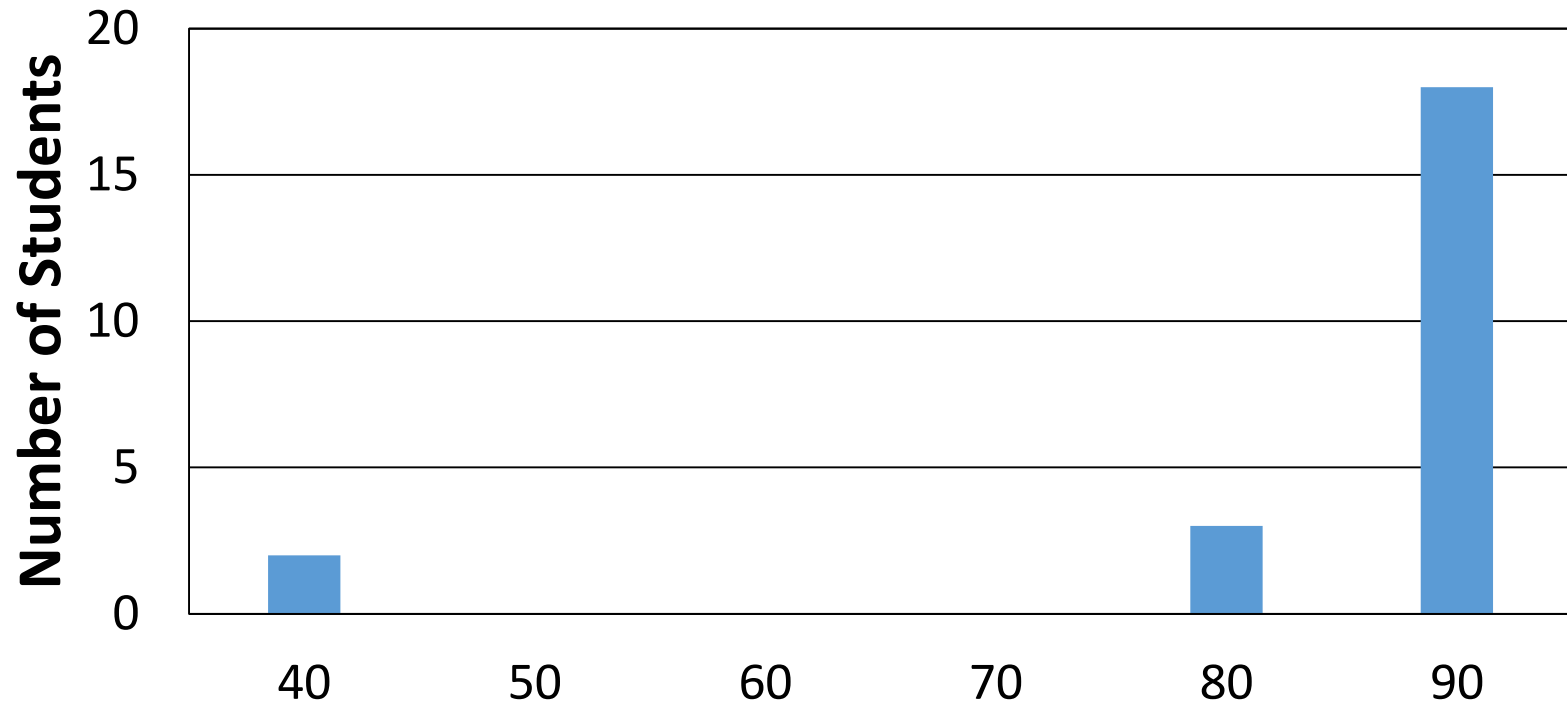
Computer Architecture
Lecture 22: Memory Controllers

Prof. Onur Mutlu

Carnegie Mellon University

Spring 2015, 3/25/2015

Lab 4 Grades



- Mean: 89.7
- Median: 94.3
- Standard Deviation: 16.2

Lab 4 Extra Credit

- Pete Ehrett (fastest) – 2%
- Navneet Saini (2nd fastest) – 1%

Announcements (I)

- No office hours today
 - Hosting a seminar in this room right after this lecture
 - Swarun Kumar, MIT, “Pushing the Limits of Wireless Networks: Interference Management and Indoor Positioning”
 - March 25, 2:30-3:30pm, HH 1107

- From talk abstract:

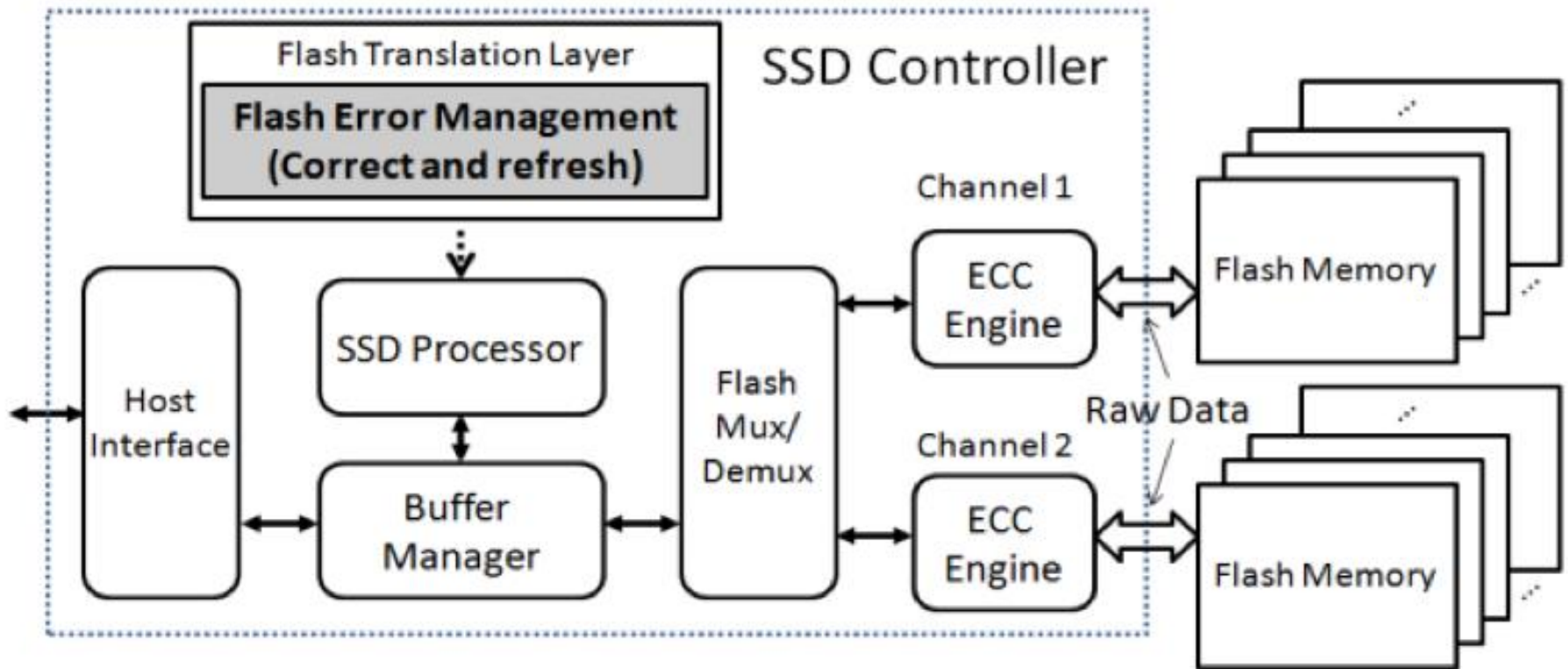
(...) perhaps our biggest expectation from modern wireless networks is faster communication speeds. However, state-of-the-art Wi-Fi networks continue to struggle in crowded environments — airports and hotel lobbies. The core reason is interference — Wi-Fi access points today avoid transmitting at the same time on the same frequency, since they would otherwise interfere with each other. I describe OpenRF, a novel system that enables today’s Wi-Fi access points to directly combat this interference and demonstrate significantly faster data-rates for real applications.

Today's Seminar on Flash Memory (4-5pm)

- March 25, Wednesday, CIC Panther Hollow Room, 4-5pm
- Yixin Luo, PhD Student, CMU
- Data Retention in MLC NAND Flash Memory: Characterization, Optimization and Recovery
- Yu Cai, Yixin Luo, Erich F. Haratsch, Ken Mai, and Onur Mutlu, **"Data Retention in MLC NAND Flash Memory: Characterization, Optimization and Recovery"**
Proceedings of the 21st International Symposium on High-Performance Computer Architecture (HPCA), Bay Area, CA, February 2015.
[Slides (pptx)] [pdf]
Best paper session.
 - http://users.ece.cmu.edu/~omutlu/pub/flash-memory-data-retention_hpca15.pdf

Flash Memory (SSD) Controllers

- Similar to DRAM memory controllers, except:
 - They are flash memory specific
 - They do much more: error correction, garbage collection, page remapping, ...



Where We Are in Lecture Schedule

- The memory hierarchy
 - Caches, caches, more caches
 - Virtualizing the memory hierarchy: Virtual Memory
 - Main memory: DRAM
 - Main memory control, scheduling
 - Memory latency tolerance techniques
 - Non-volatile memory
-
- Multiprocessors
 - Coherence and consistency
 - Interconnection networks
 - Multi-core issues

Required Reading (for the Next Few Lectures)

- Onur Mutlu, Justin Meza, and Lavanya Subramanian, **"The Main Memory System: Challenges and Opportunities"**

*Invited Article in Communications of the Korean Institute of Information Scientists and Engineers (**KIISE**), 2015.*

http://users.ece.cmu.edu/~omutlu/pub/main-memory-system_kiise15.pdf

Required Readings on DRAM

■ DRAM Organization and Operation Basics

- Sections 1 and 2 of: Lee et al., “Tiered-Latency DRAM: A Low Latency and Low Cost DRAM Architecture,” HPCA 2013.

http://users.ece.cmu.edu/~omutlu/pub/tldram_hpca13.pdf

- Sections 1 and 2 of Kim et al., “A Case for Subarray-Level Parallelism (SALP) in DRAM,” ISCA 2012.

http://users.ece.cmu.edu/~omutlu/pub/salp-dram_isca12.pdf

■ DRAM Refresh Basics

- Sections 1 and 2 of Liu et al., “RAIDR: Retention-Aware Intelligent DRAM Refresh,” ISCA 2012.

http://users.ece.cmu.edu/~omutlu/pub/raidr-dram-refresh_isca12.pdf

Memory Controllers

DRAM versus Other Types of Memories

- Long latency memories have similar characteristics that need to be controlled.
- The following discussion will use DRAM as an example, but many scheduling and control issues are similar in the design of controllers for other types of memories
 - Flash memory
 - Other emerging memory technologies
 - Phase Change Memory
 - Spin-Transfer Torque Magnetic Memory
 - These other technologies can place other demands on the controller

DRAM Types

- DRAM has different types with different interfaces optimized for different purposes
 - ❑ Commodity: DDR, DDR2, DDR3, DDR4, ...
 - ❑ Low power (for mobile): LPDDR1, ..., LPDDR5, ...
 - ❑ High bandwidth (for graphics): GDDR2, ..., GDDR5, ...
 - ❑ Low latency: eDRAM, RDRAM, ...
 - ❑ 3D stacked: WIO, HBM, HMC, ...
 - ❑ ...
- Underlying microarchitecture is fundamentally the same
- A flexible memory controller can support various DRAM types
- This complicates the memory controller
 - ❑ Difficult to support all types (and upgrades)

DRAM Types (II)

<i>Segment</i>	<i>DRAM Standards & Architectures</i>
Commodity	DDR3 (2007) [14]; DDR4 (2012) [18]
Low-Power	LPDDR3 (2012) [17]; LPDDR4 (2014) [20]
Graphics	GDDR5 (2009) [15]
Performance	eDRAM [28], [32]; RLDram3 (2011) [29]
3D-Stacked	WIO (2011) [16]; WIO2 (2014) [21]; MCDRAM (2015) [13]; HBM (2013) [19]; HMC1.0 (2013) [10]; HMC1.1 (2014) [11]
Academic	SBA/SSA (2010) [38]; Staged Reads (2012) [8]; RAIDR (2012) [27]; SALP (2012) [24]; TL-DRAM (2013) [26]; RowClone (2013) [37]; Half-DRAM (2014) [39]; Row-Buffer Decoupling (2014) [33]; SARP (2014) [6]; AL-DRAM (2015) [25]

Table 1. Landscape of DRAM-based memory

Kim et al., “[Ramulator: A Fast and Extensible DRAM Simulator](#),” IEEE Comp Arch Letters 2015.

DRAM Controller: Functions

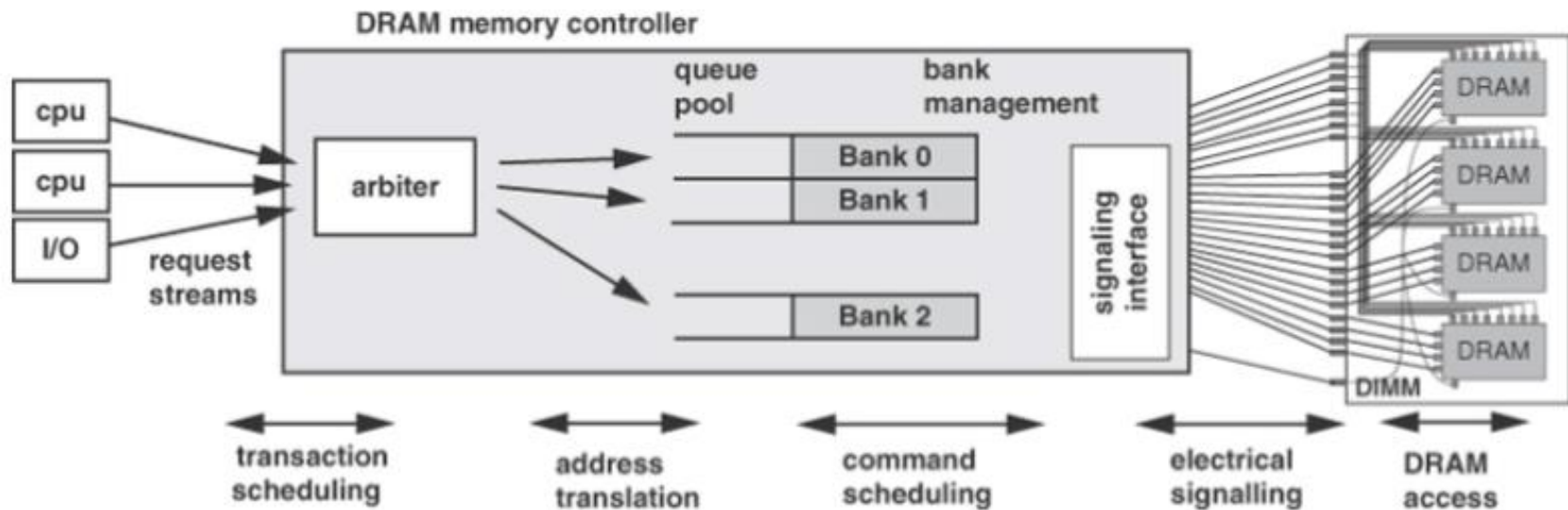
- Ensure correct operation of DRAM (refresh and timing)
- Service DRAM requests while obeying timing constraints of DRAM chips
 - Constraints: resource conflicts (bank, bus, channel), minimum write-to-read delays
 - Translate requests to DRAM command sequences
- Buffer and schedule requests to for high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

DRAM Controller: Where to Place

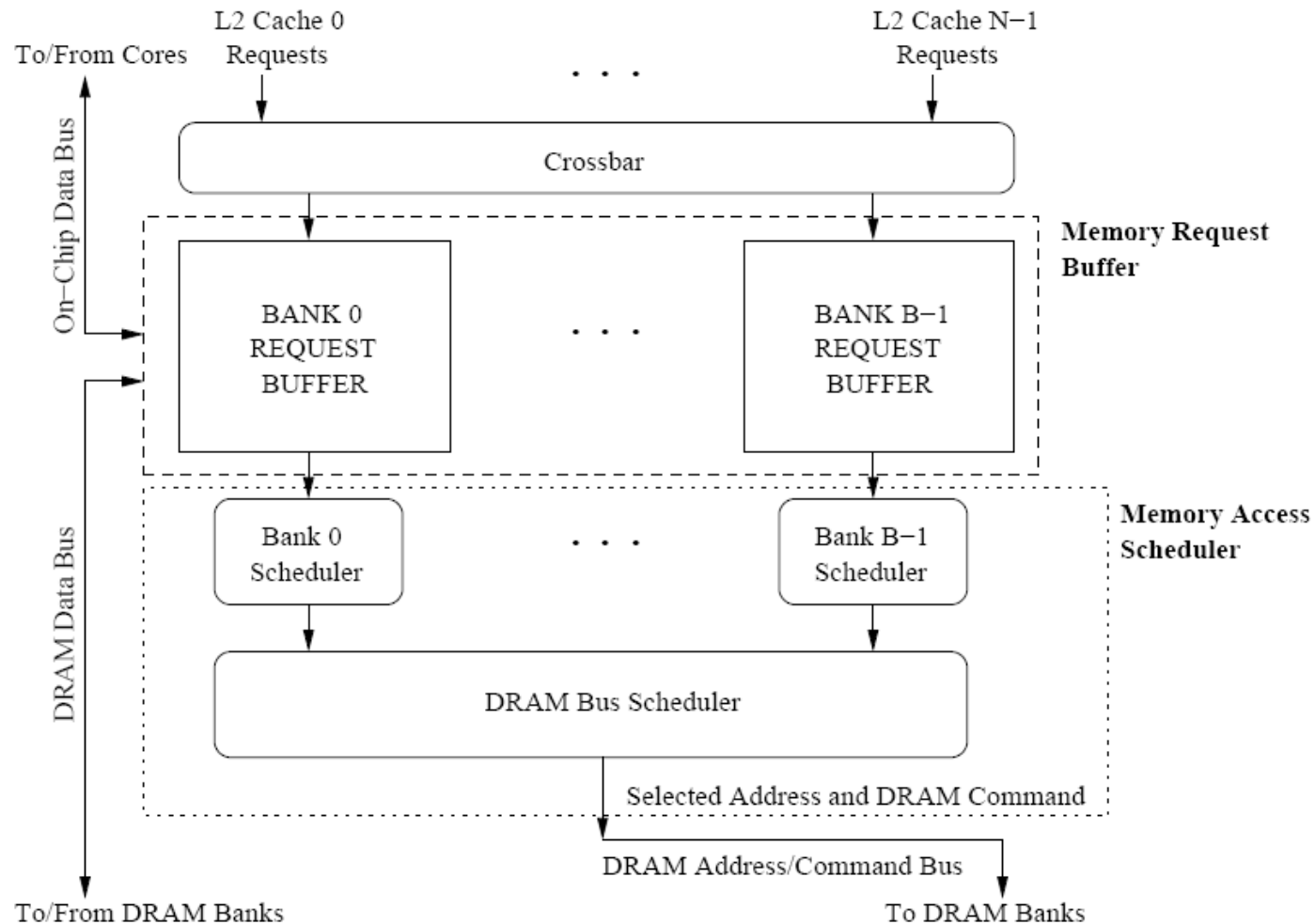
- In chipset
 - + More flexibility to plug different DRAM types into the system
 - + Less power density in the CPU chip

- On CPU chip
 - + Reduced latency for main memory access
 - + Higher bandwidth between cores and controller
 - More information can be communicated (e.g. request's importance in the processing core)

A Modern DRAM Controller (I)



A Modern DRAM Controller (II)



DRAM Scheduling Policies (I)

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

- Oldest request first

- **FR-FCFS** (first ready, first come first served)

1. Row-hit first
2. Oldest first

Goal: Maximize row buffer hit rate → **maximize DRAM throughput**

- Actually, scheduling is done at the **command level**

- Column commands (read/write) prioritized over row commands (activate/precharge)
- Within each group, older commands prioritized over younger ones

DRAM Scheduling Policies (II)

- A scheduling policy is a request prioritization order

- Prioritization can be based on
 - Request age
 - Row buffer hit/miss status
 - Request type (prefetch, read, write)
 - Requestor type (load miss or store miss)
 - Request criticality
 - Oldest miss in the core?
 - How many instructions in core are dependent on it?
 - Will it stall the processor?
 - Interference caused to other cores
 - ...

Row Buffer Management Policies

■ Open row

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

■ Closed row

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

■ Adaptive policies

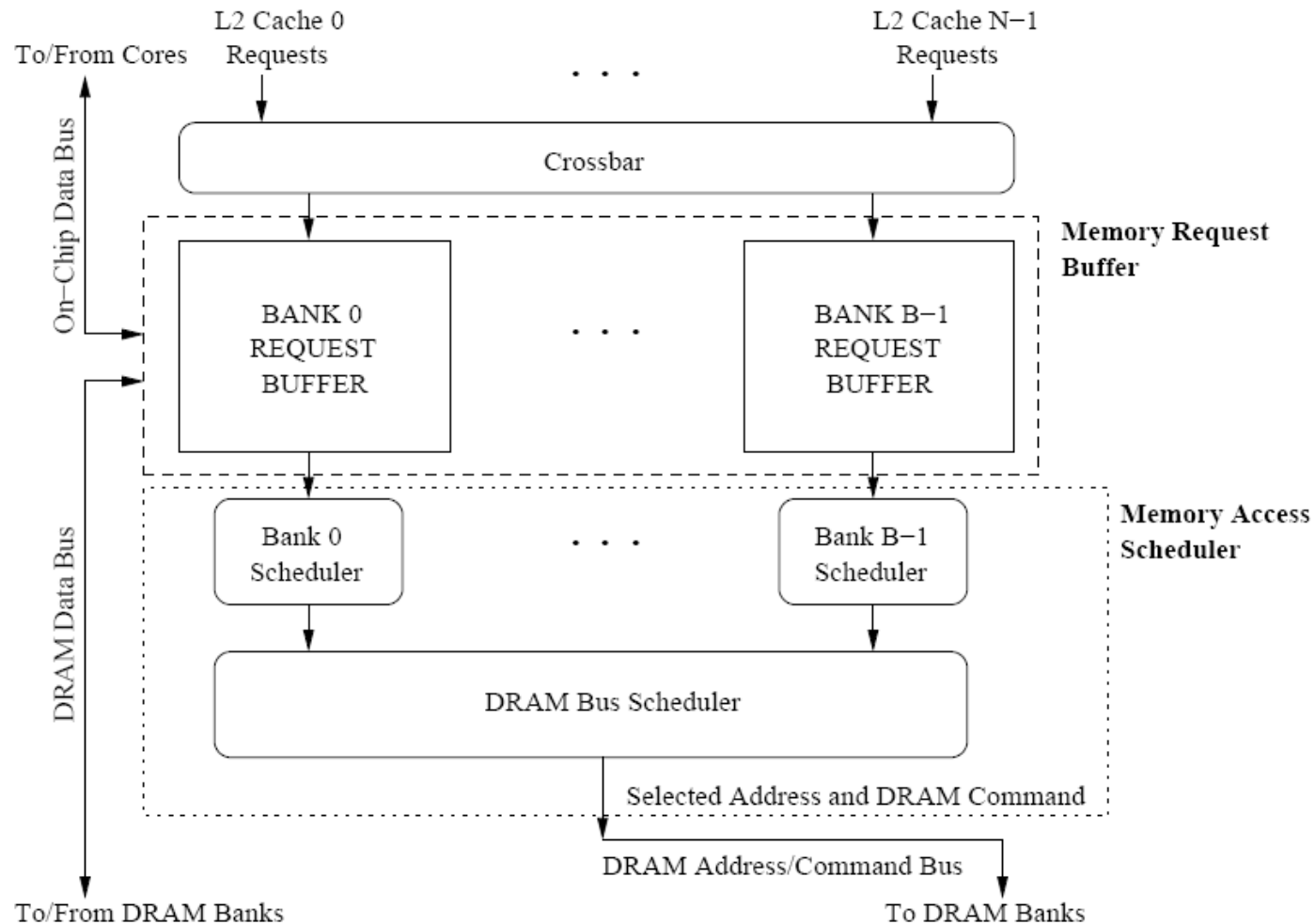
- Predict whether or not the next access to the bank will be to the same row

Open vs. Closed Row Policies

Policy	First access	Next access	Commands needed for next access
Open row	Row 0	Row 0 (row hit)	Read
Open row	Row 0	Row 1 (row conflict)	Precharge + Activate Row 1 + Read
Closed row	Row 0	Row 0 – access in request buffer (row hit)	Read
Closed row	Row 0	Row 0 – access not in request buffer (row closed)	Activate Row 0 + Read + Precharge
Closed row	Row 0	Row 1 (row closed)	Activate Row 1 + Read + Precharge

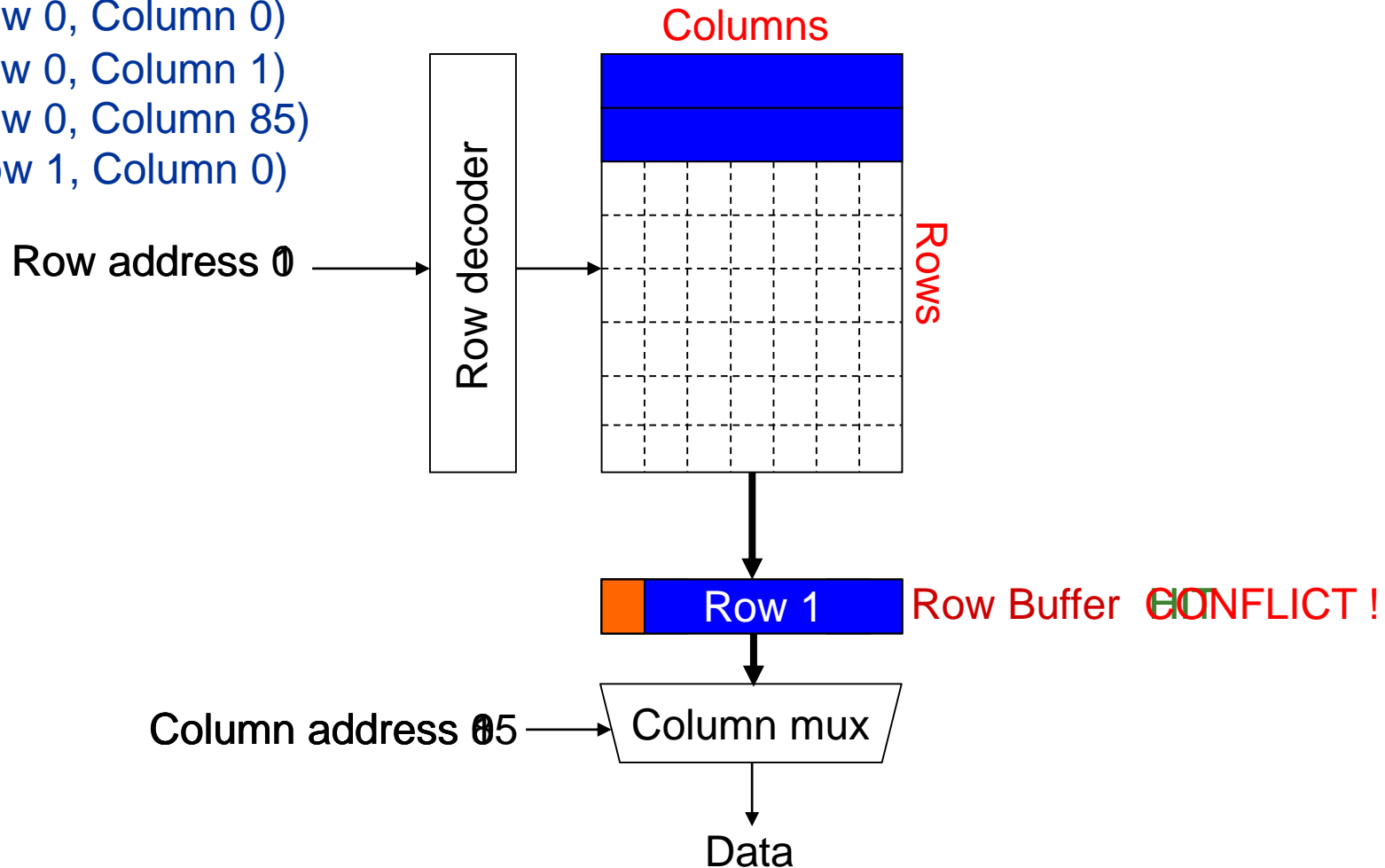
Memory Interference and Scheduling in Multi-Core Systems

Review: A Modern DRAM Controller



Review: DRAM Bank Operation

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

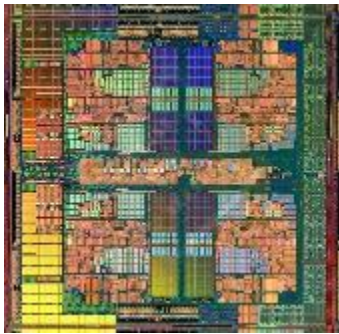


Scheduling Policy for Single-Core Systems

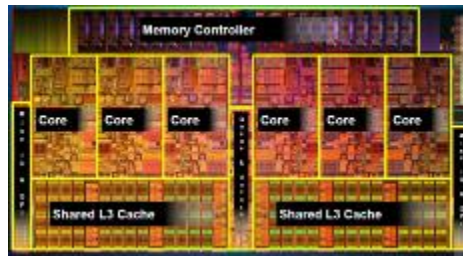
- A row-conflict memory access takes significantly longer than a row-hit access
 - Current controllers take advantage of the row buffer
 - **FR-FCFS** (first ready, first come first served) scheduling policy
 1. Row-hit first
 2. Oldest first
- Goal 1: Maximize row buffer hit rate → maximize DRAM throughput
- Goal 2: Prioritize older requests → ensure forward progress
- Is this a good policy in a multi-core system?

Trend: Many Cores on Chip

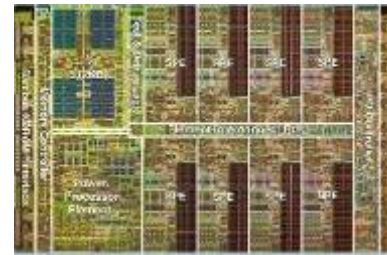
- Simpler and lower power than a single large core
- Large scale parallelism on chip



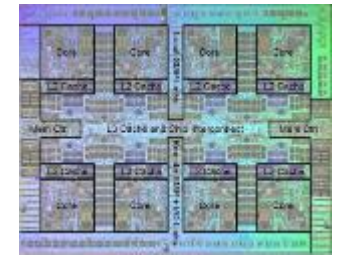
AMD Barcelona
4 cores



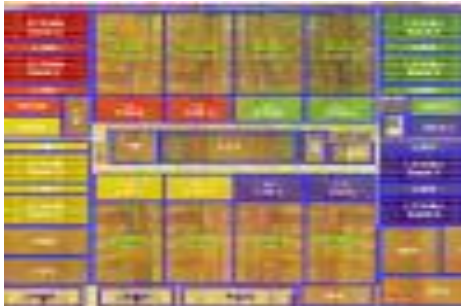
Intel Core i7
8 cores



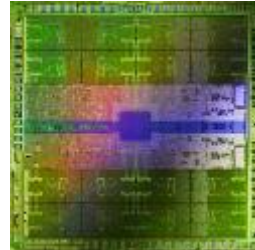
IBM Cell BE
8+1 cores



IBM POWER7
8 cores



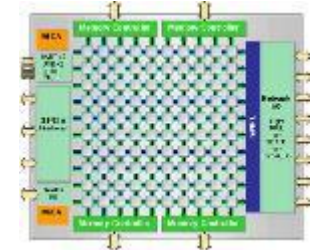
Sun Niagara II
8 cores



Nvidia Fermi
448 "cores"



Intel SCC
48 cores, networked

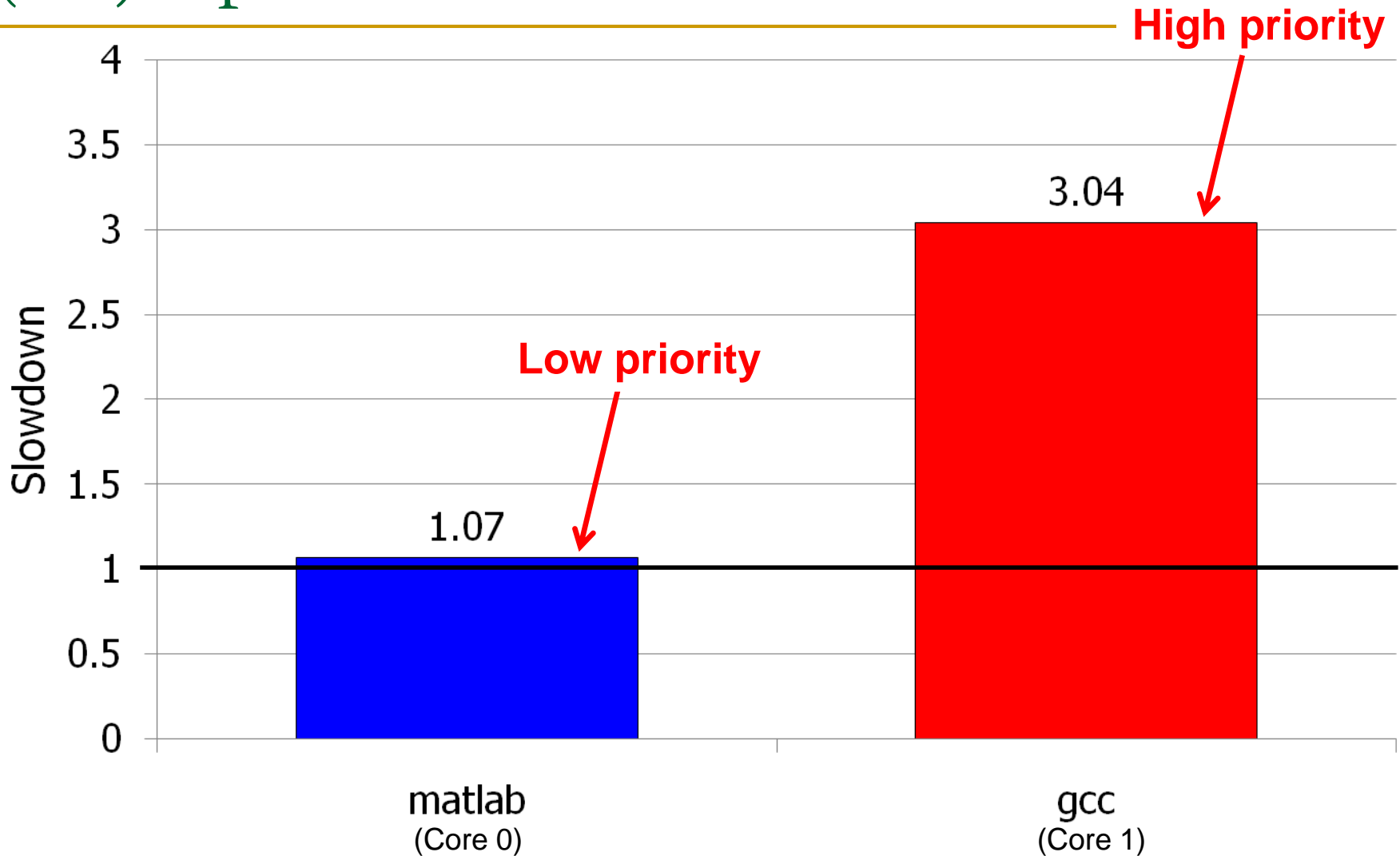


Tiler TILE Gx
100 cores, networked

Many Cores on Chip

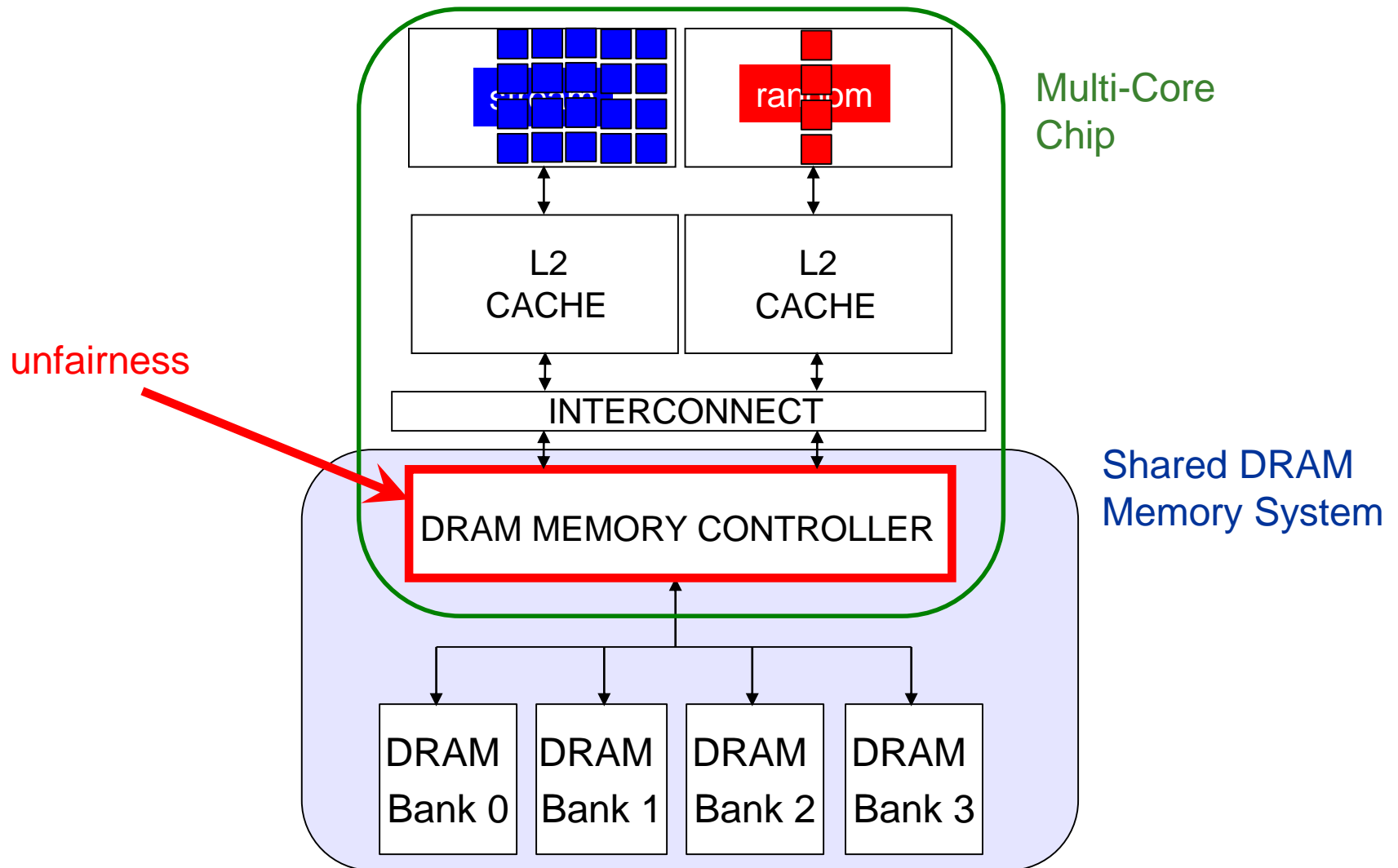
- What we want:
 - N times the system performance with N times the cores
- What do we get today?

(Un)expected Slowdowns in Multi-Core



Moscibroda and Mutlu, “[Memory performance attacks: Denial of memory service in multi-core systems](#),” USENIX Security 2007.

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

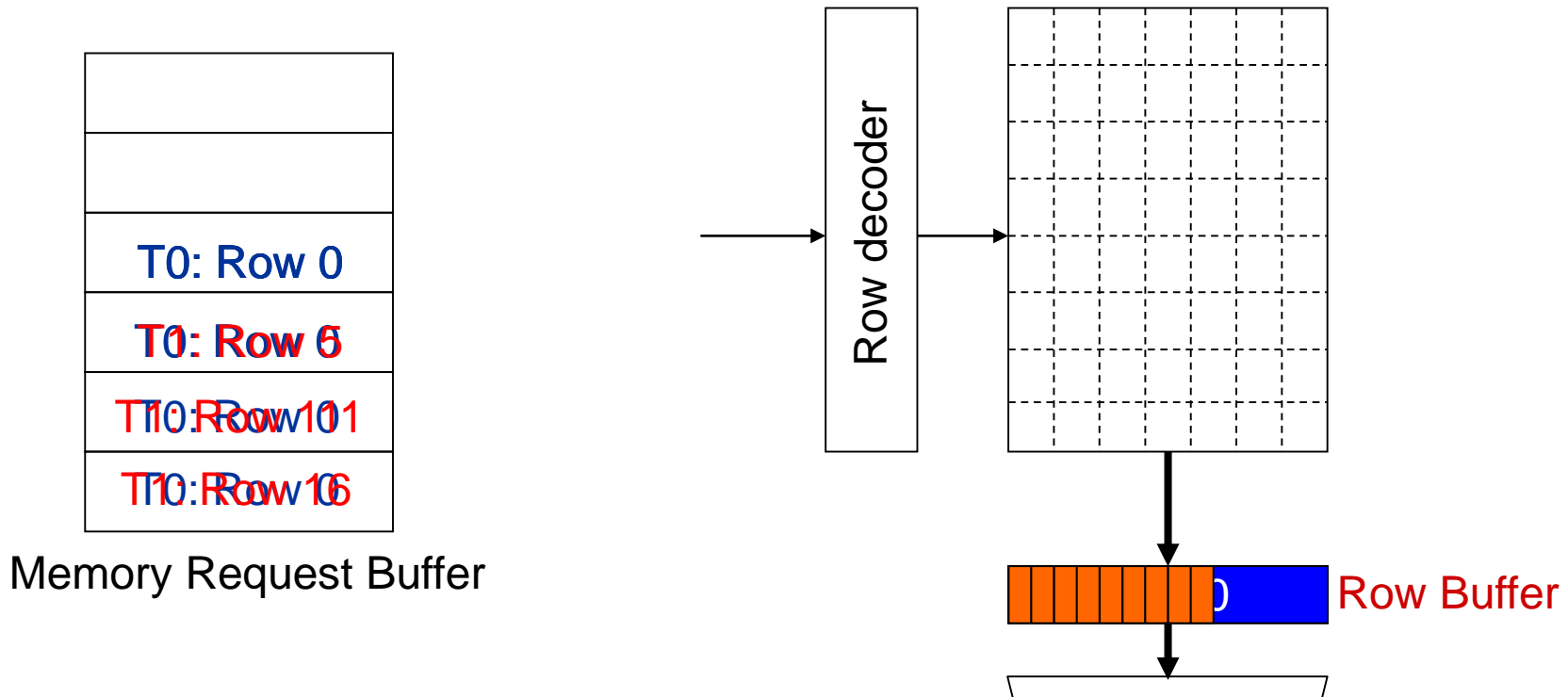
for (j=0; j<N; j++) {
    index = rand(); random
    A[index] = B[index];
    ...
}
```

RANDOM

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

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

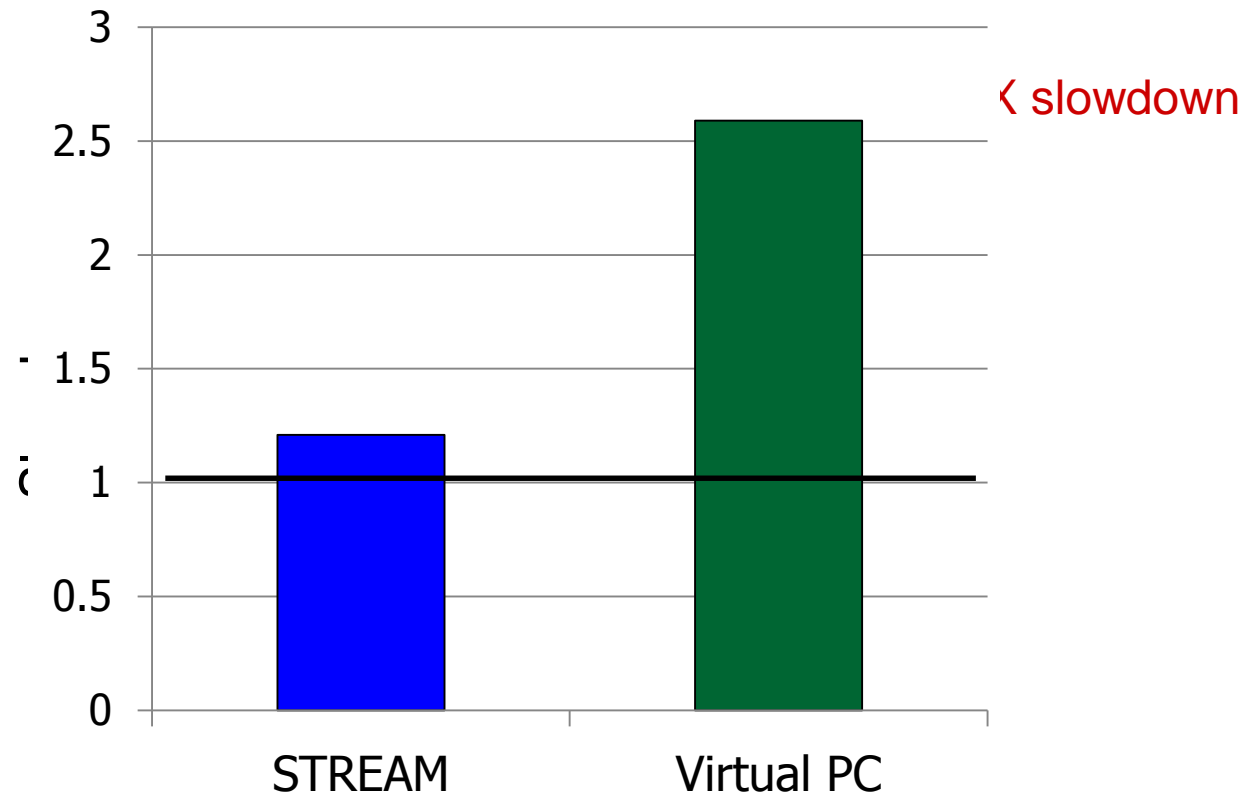
What Does the Memory Hog Do?



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.

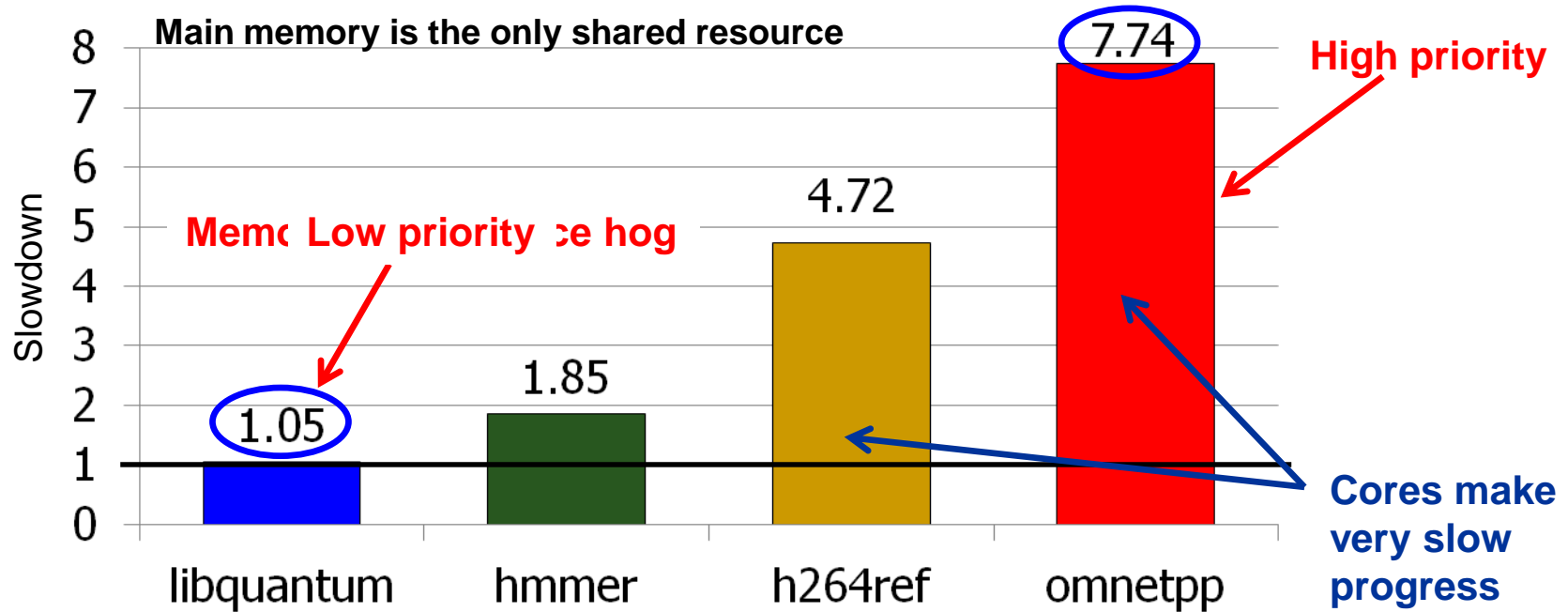
Effect of the Memory Performance Hog



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

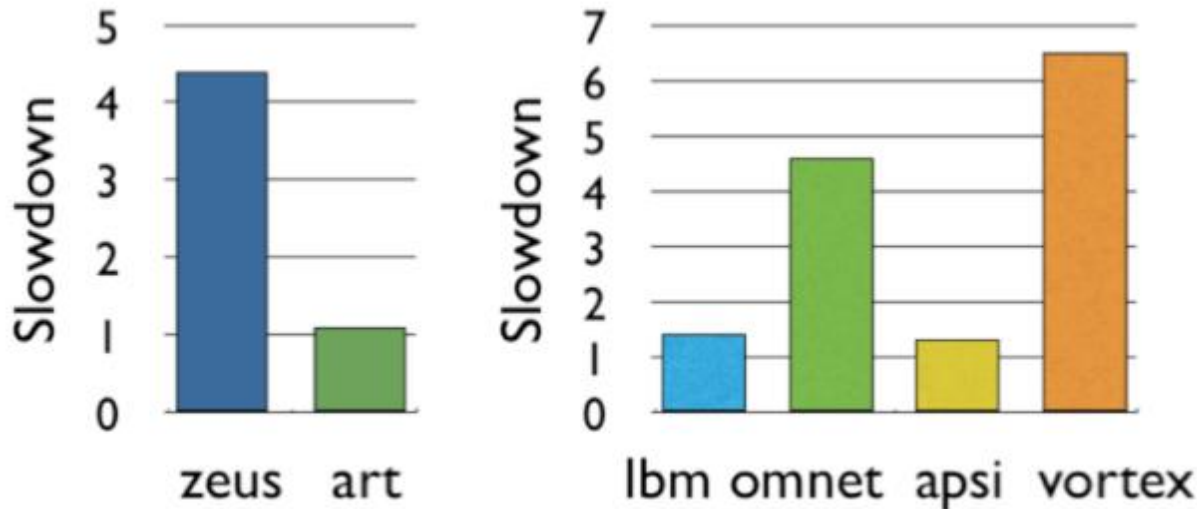
Moscibroda and Mutlu, “[Memory Performance Attacks](#),” USENIX Security 2007.

Problems due to Uncontrolled Interference



- Unfair slowdown of different threads
- Low system performance
- Vulnerability to denial of service
- Priority inversion: unable to enforce priorities/SLAs

Problems due to Uncontrolled Interference



- Unfair slowdown of different threads
- Low system performance
- Vulnerability to denial of service
- Priority inversion: unable to enforce priorities/SLAs
- Poor performance predictability (no performance isolation)

Uncontrollable, unpredictable system

Recap: Inter-Thread Interference in Memory

- Memory controllers, pins, and memory banks are shared
- Pin bandwidth is not increasing as fast as number of cores
 - Bandwidth per core reducing
- Different threads executing on different cores interfere with each other in the main memory system
- Threads delay each other by causing resource contention:
 - Bank, bus, row-buffer conflicts → reduced DRAM throughput
- Threads can also destroy each other's **DRAM bank parallelism**
 - Otherwise parallel requests can become serialized

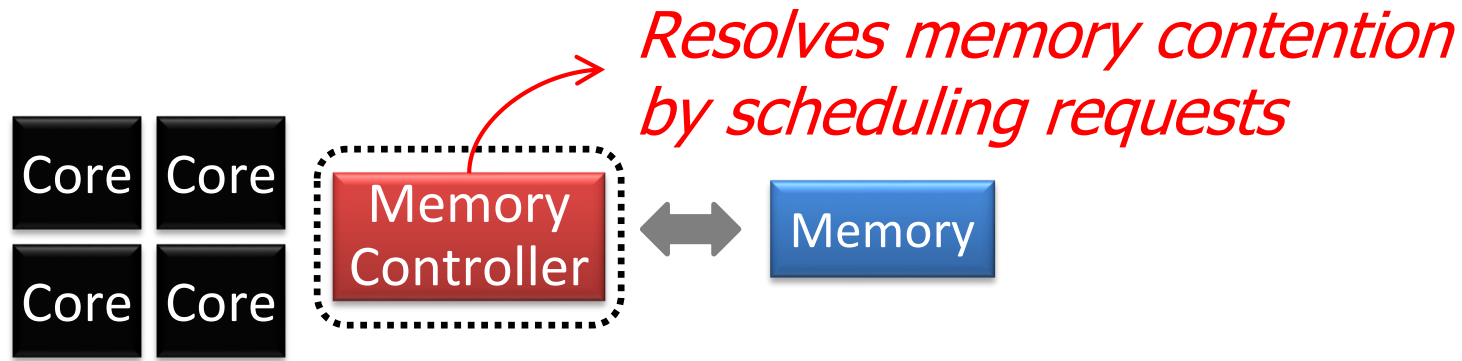
Effects of Inter-Thread Interference in DRAM

- Queueing/contention delays
 - Bank conflict, bus conflict, channel conflict, ...
- Additional delays due to DRAM constraints
 - Called “protocol overhead”
 - Examples
 - Row conflicts
 - Read-to-write and write-to-read delays
- Loss of intra-thread parallelism
 - A thread’s concurrent requests are serviced serially instead of in parallel

Problem: QoS-Unaware Memory Control

- Existing DRAM controllers are unaware of inter-thread interference in DRAM system
- They simply aim to maximize DRAM throughput
 - Thread-unaware and thread-unfair
 - No intent to service each thread's requests in parallel
 - FR-FCFS policy: 1) row-hit first, 2) oldest first
 - Unfairly prioritizes threads with high row-buffer locality
 - Unfairly prioritizes threads that are memory intensive (many outstanding memory accesses)

Solution: QoS-Aware Memory Request Scheduling



- How to schedule requests to provide
 - ❑ High system performance
 - ❑ High fairness to applications
 - ❑ Configurability to system software
- Memory controller needs to be aware of threads

Stall-Time Fair Memory Scheduling

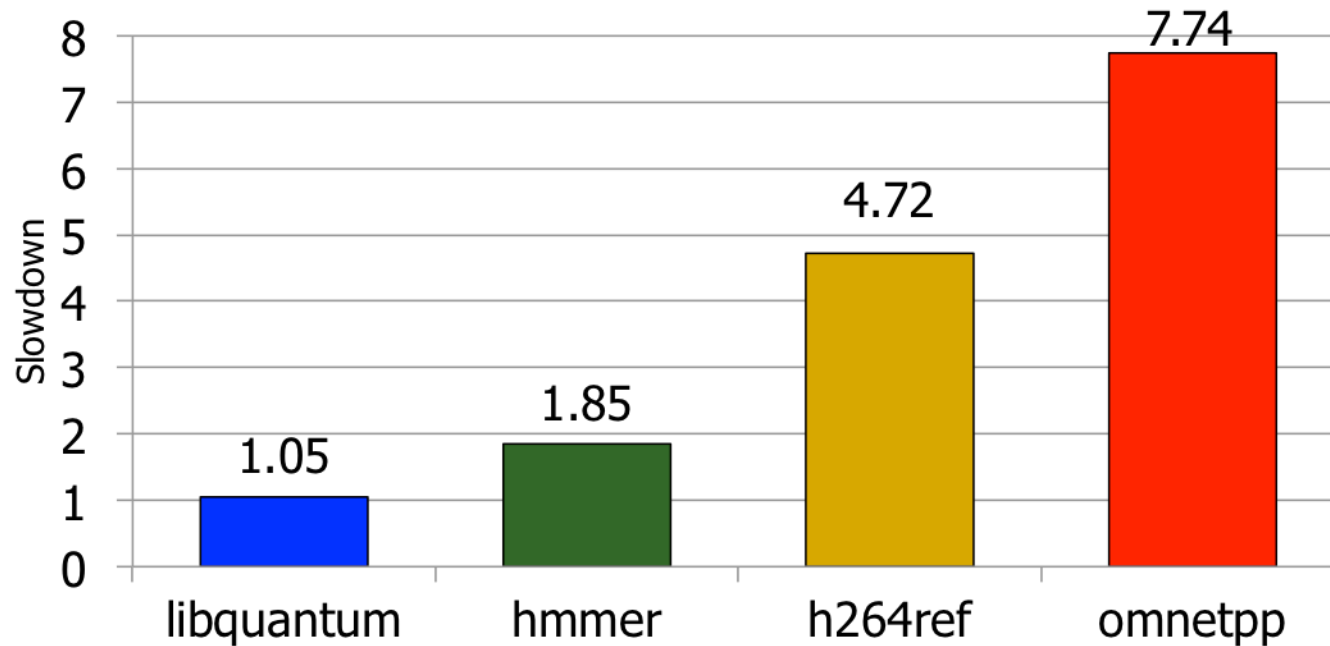
Onur Mutlu and Thomas Moscibroda,

"Stall-Time Fair Memory Access Scheduling for Chip Multiprocessors"

40th International Symposium on Microarchitecture (MICRO),

pages 146-158, Chicago, IL, December 2007. Slides (ppt)

The Problem: Unfairness



- Unfair slowdown of different threads
- Low system performance
- Vulnerability to denial of service
- Priority inversion: unable to enforce priorities/SLAs
- Poor performance predictability (no performance isolation)

Uncontrollable, unpredictable system

How Do We Solve the Problem?

- Stall-time fair memory scheduling [Mutlu+ MICRO'07]
- Goal: Threads sharing main memory should experience similar slowdowns compared to when they are run alone → fair scheduling
 - Also improves overall system performance by ensuring cores make “proportional” progress
- Idea: Memory controller estimates each thread’s slowdown due to interference and schedules requests in a way to balance the slowdowns
- 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 alone
- **Memory-slowdown** = $ST_{\text{shared}}/ST_{\text{alone}}$
 - Relative increase in stall-time
- *Stall-Time Fair Memory scheduler (STFM)* aims to equalize **Memory-slowdown** for interfering threads, without sacrificing performance
 - Considers inherent DRAM performance of each thread
 - Aims to allow proportional progress of threads

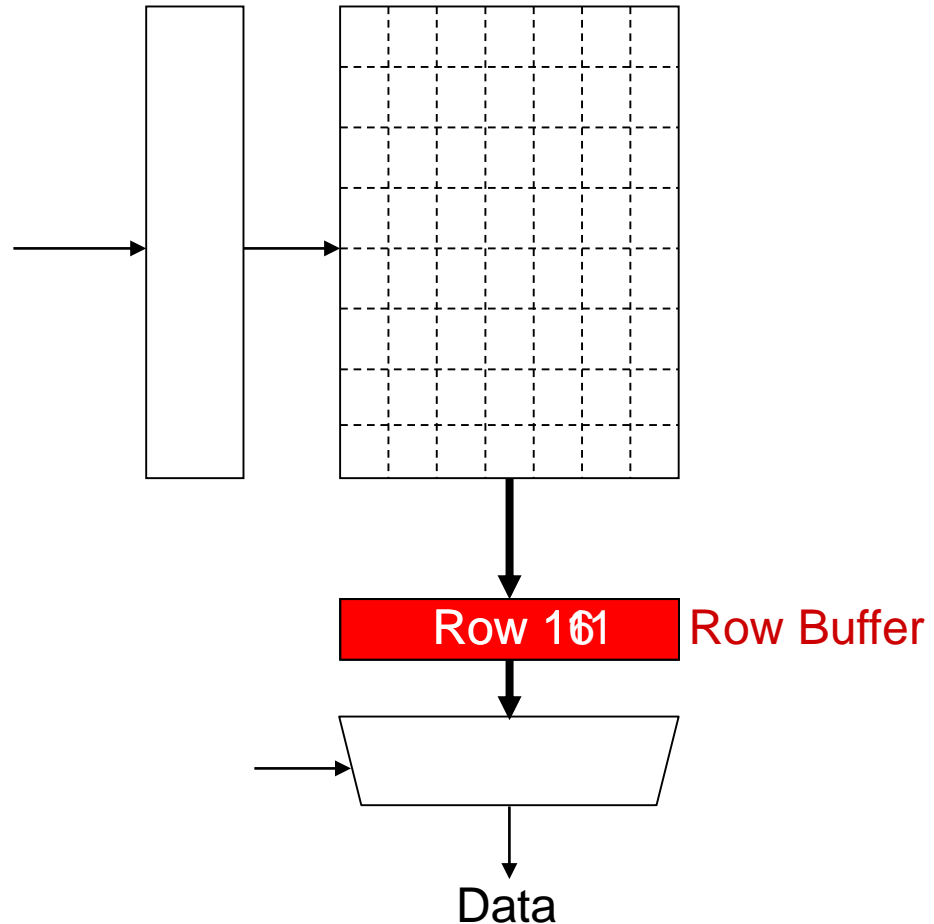
STFM Scheduling Algorithm [MICRO' 07]

- For each thread, the DRAM controller
 - Tracks ST_{shared}
 - Estimates ST_{alone}
- Each cycle, the DRAM controller
 - Computes $\text{Slowdown} = ST_{\text{shared}} / ST_{\text{alone}}$ for threads with legal requests
 - Computes **unfairness = MAX Slowdown / MIN Slowdown**
- If $\text{unfairness} < \alpha$
 - Use DRAM throughput oriented scheduling policy
- **If unfairness $\geq \alpha$**
 - Use fairness-oriented scheduling policy
 - **(1) requests from thread with MAX Slowdown first**
 - (2) row-hit first , (3) oldest-first

How Does STFMM Prevent Unfairness?

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

T0 Slowdown	1.00
T1 Slowdown	1.00
Unfairness	1.00
α	1.05



STFM Pros and Cons

■ Upsides:

- ❑ First algorithm for fair multi-core memory scheduling
- ❑ Provides a mechanism to estimate memory slowdown of a thread
- ❑ Good at providing fairness
- ❑ Being fair can improve performance

■ Downsides:

- ❑ Does not handle all types of interference
- ❑ (Somewhat) complex to implement
- ❑ Slowdown estimations can be incorrect

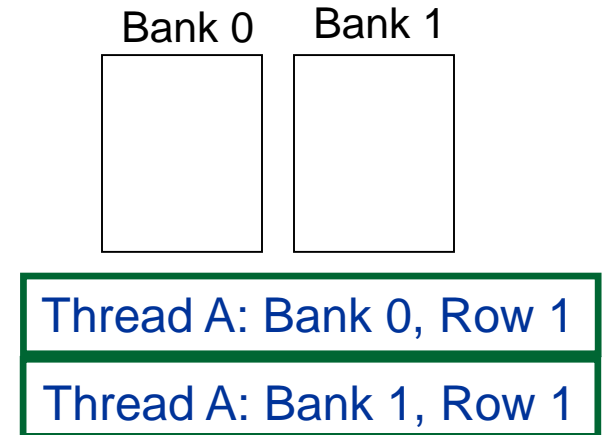
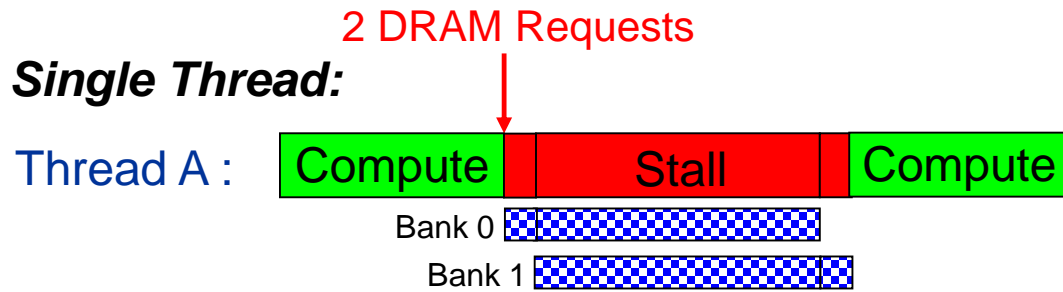
Parallelism-Aware Batch Scheduling

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

Another Problem due to Interference

- Processors try to tolerate the latency of DRAM requests by generating multiple outstanding requests
 - Memory-Level Parallelism (MLP)
 - Out-of-order execution, non-blocking caches, runahead execution
- Effective only if the DRAM controller actually services the multiple requests in parallel in DRAM banks
- Multiple threads share the DRAM controller
- DRAM controllers are not aware of a thread's MLP
 - Can service each thread's outstanding requests serially, not in parallel

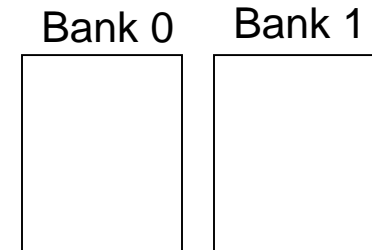
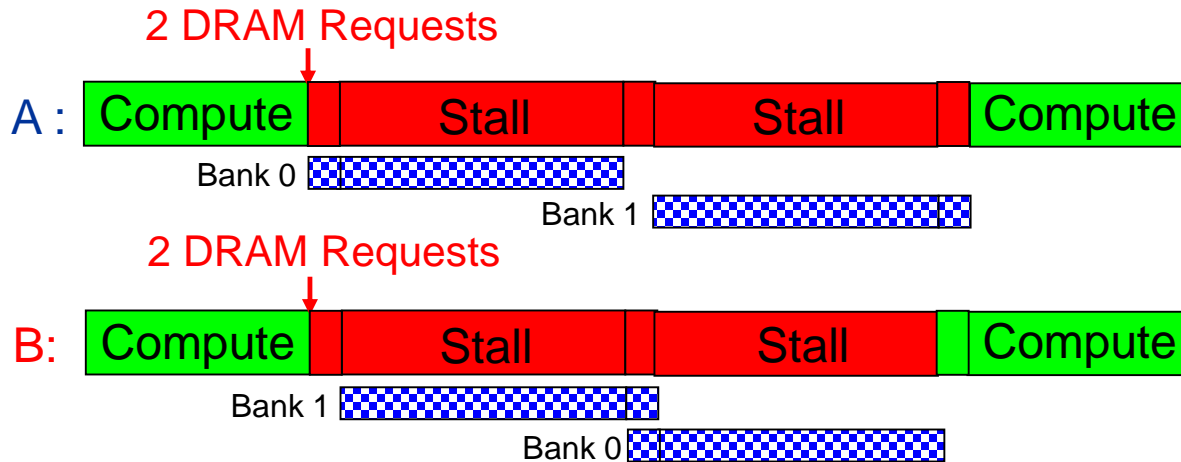
Bank Parallelism of a Thread



Bank access latencies of the two requests overlapped
Thread stalls for ~ONE bank access latency

Bank Parallelism Interference in DRAM

Baseline Scheduler:



Thread A: Bank 0, Row 1

Thread B: Bank 1, Row 99

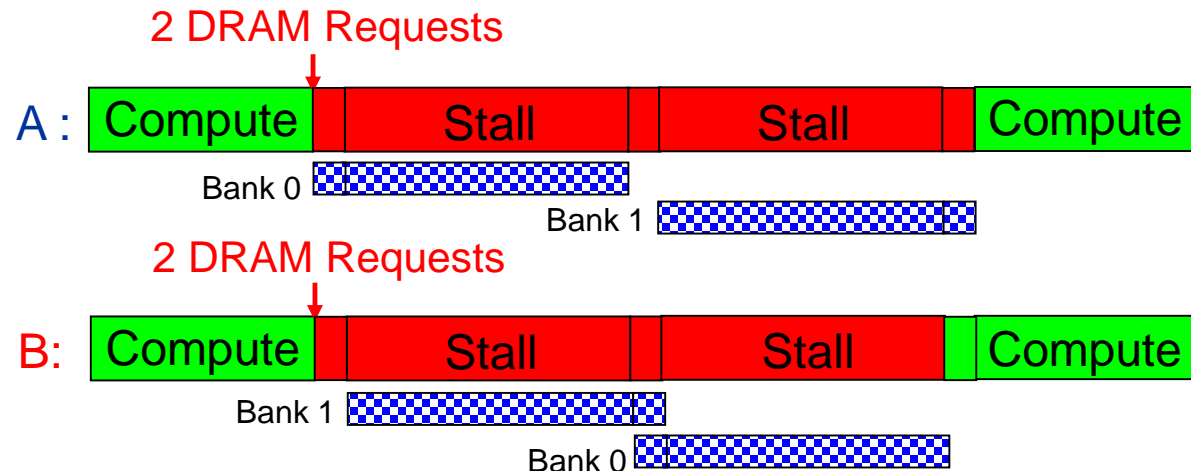
Thread B: Bank 0, Row 99

Thread A: Bank 1, Row 1

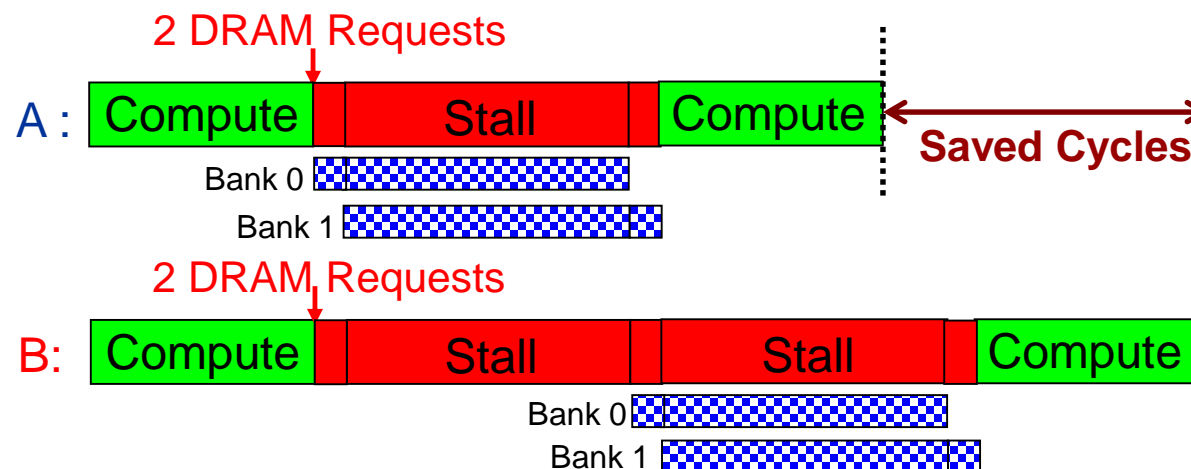
Bank access latencies of each thread serialized
Each thread stalls for ~TWO bank access latencies

Parallelism-Aware Scheduler

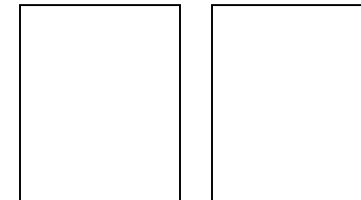
Baseline Scheduler:



Parallelism-aware Scheduler:



Bank 0 Bank 1



Thread A: Bank 0, Row 1

Thread B: Bank 1, Row 99

Thread B: Bank 0, Row 99

Thread A: Bank 1, Row 1

**Average stall-time:
~1.5 bank access
latencies**

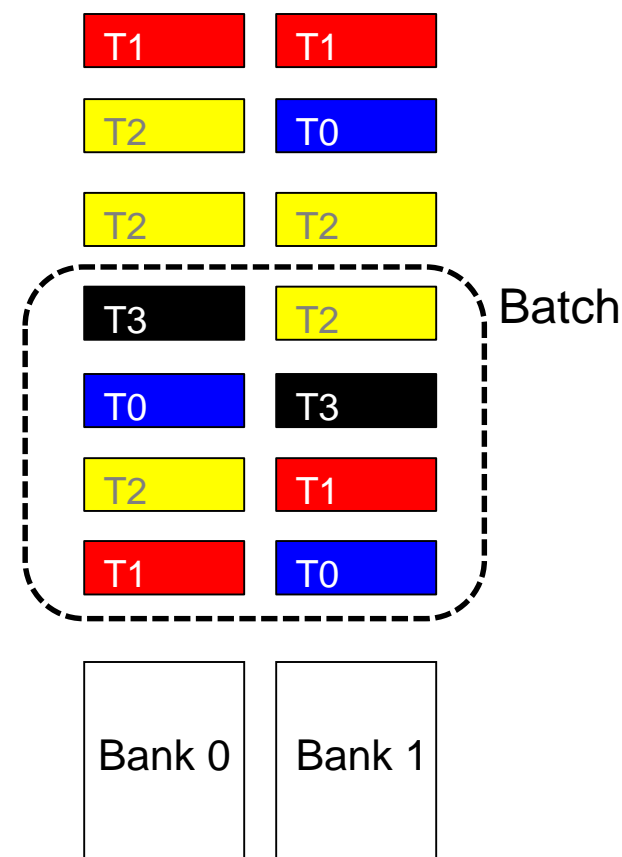
Parallelism-Aware Batch Scheduling (PAR-BS)

■ Principle 1: Parallelism-awareness

- ❑ Schedule requests from a thread (to different banks) back to back
- ❑ Preserves each thread's bank parallelism
- ❑ But, this can cause starvation...

■ Principle 2: Request Batching

- ❑ Group a fixed number of oldest requests from each thread into a “batch”
- ❑ Service the batch before all other requests
- ❑ Form a new batch when the current one is done
- ❑ Eliminates starvation, provides fairness
- ❑ Allows parallelism-awareness within a batch



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 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

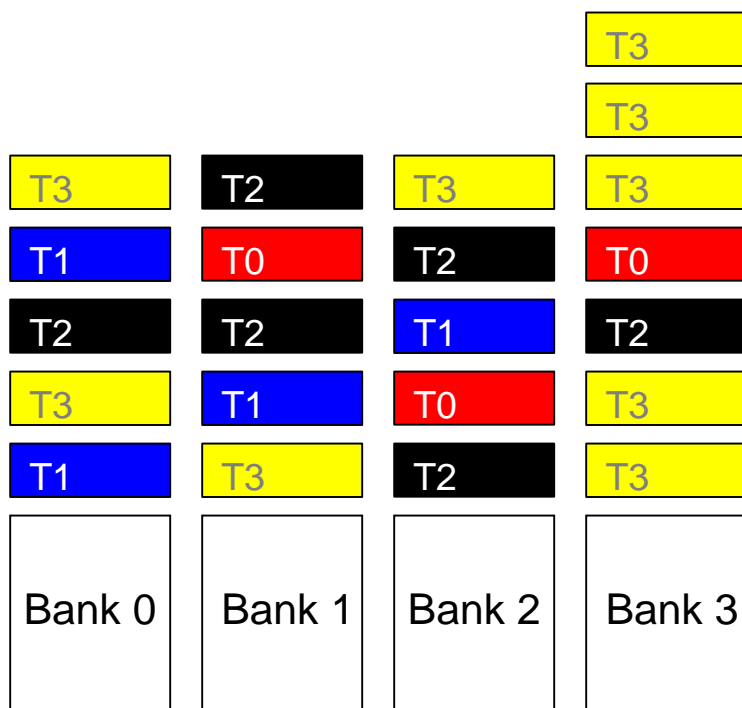
How to Rank Threads within a Batch

- 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

* W.E. Smith, "Various optimizers for single stage production," Naval Research Logistics Quarterly, 1956.

Shortest Stall-Time First Ranking

- Maximum number of marked requests to any bank (max-bank-load)
 - Rank thread with lower max-bank-load higher (\sim low stall-time)
- Total number of marked requests (total-load)
 - Breaks ties: rank thread with lower total-load higher

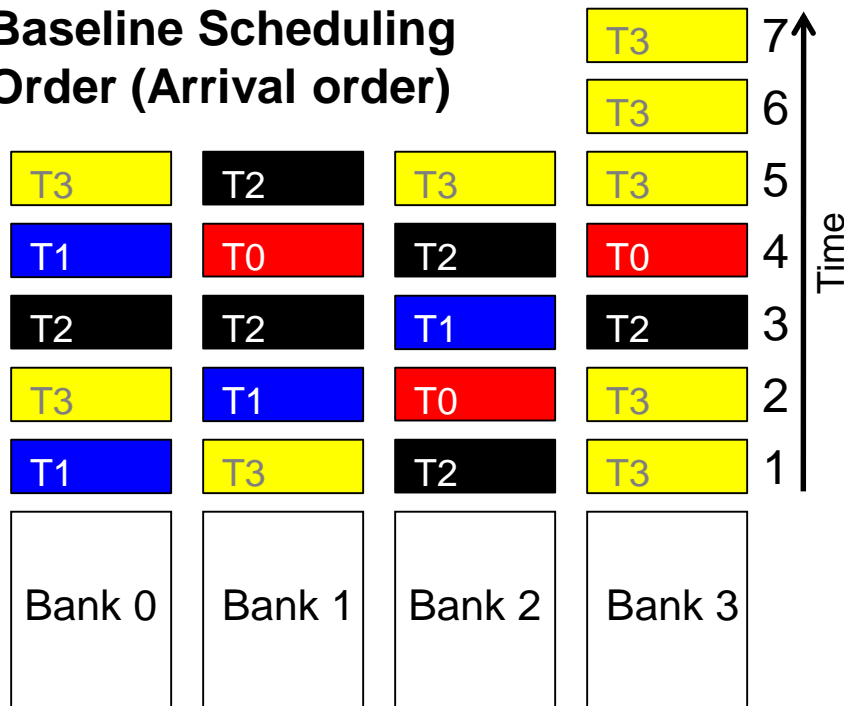


	max-bank-load	total-load

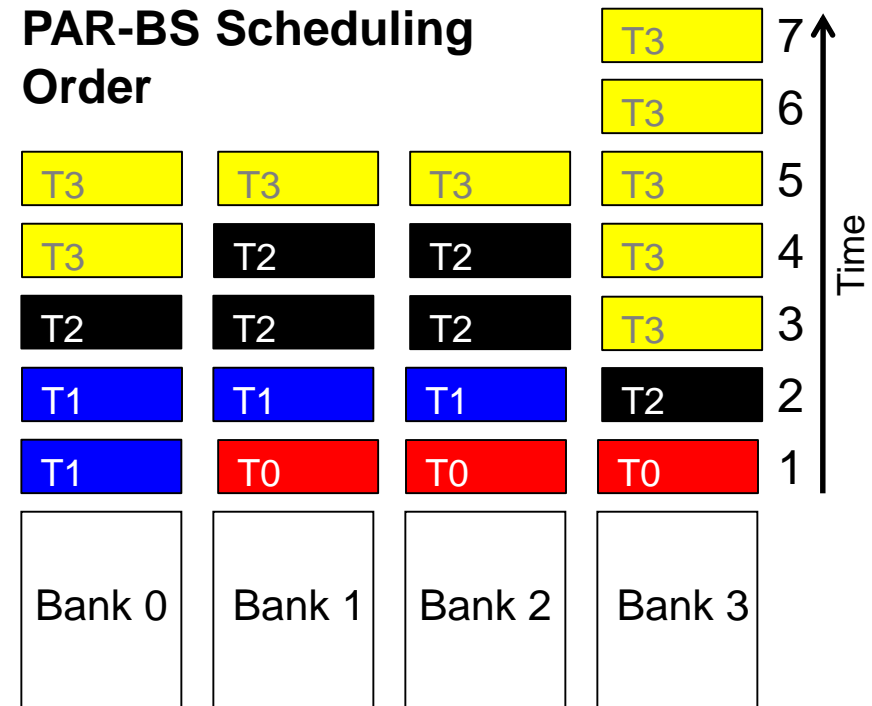
Ranking:
T0 > T1 > T2 > T3

Example Within-Batch Scheduling Order

Baseline Scheduling Order (Arrival order)



PAR-BS Scheduling Order



Ranking: T0 > T1 > T2 > T3

	T0	T1	T2	T3
Stall times				

AVG: 5 bank access latencies

	T0	T1	T2	T3
Stall times				

AVG: 3.5 bank access latencies

Putting It Together: PAR-BS Scheduling Policy

■ PAR-BS Scheduling Policy

(1) Marked requests first

Batching

(2) Row-hit requests first

(3) Higher-rank thread first (shortest stall-time first)

Parallelism-aware
within-batch
scheduling

(4) Oldest first

■ Three properties:

- Exploits row-buffer locality **and** intra-thread bank parallelism
- Work-conserving: does not waste bandwidth when it can be used
 - 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

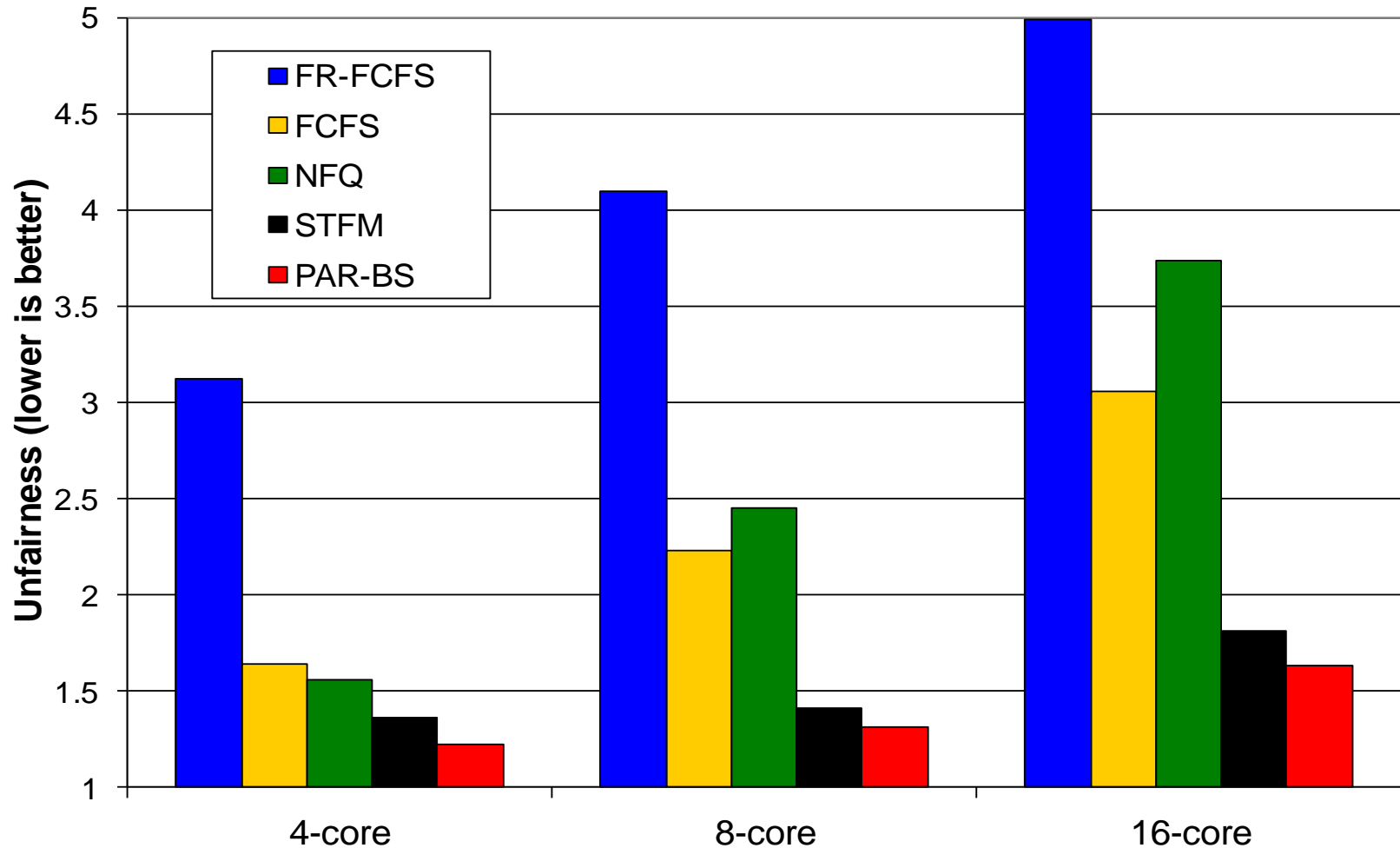
- Mutlu and Moscibroda, “[Parallelism-Aware Batch Scheduling](#),” ISCA 2008.

Hardware Cost

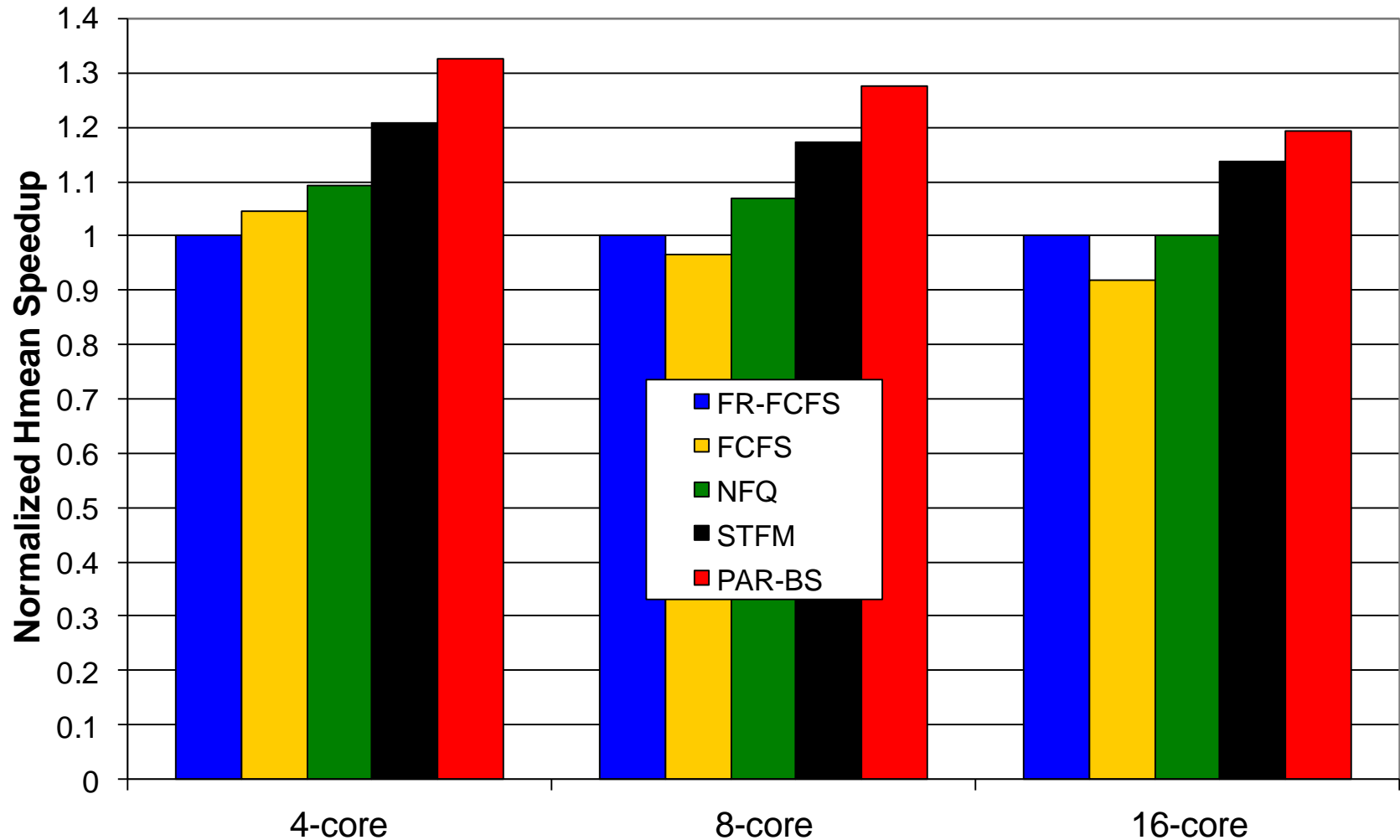
- <1.5KB storage cost for
 - 8-core system with 128-entry memory request buffer
- No complex operations (e.g., divisions)
- Not on the critical path
 - Scheduler makes a decision only every DRAM cycle

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

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



System Performance



PAR-BS Pros and Cons

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

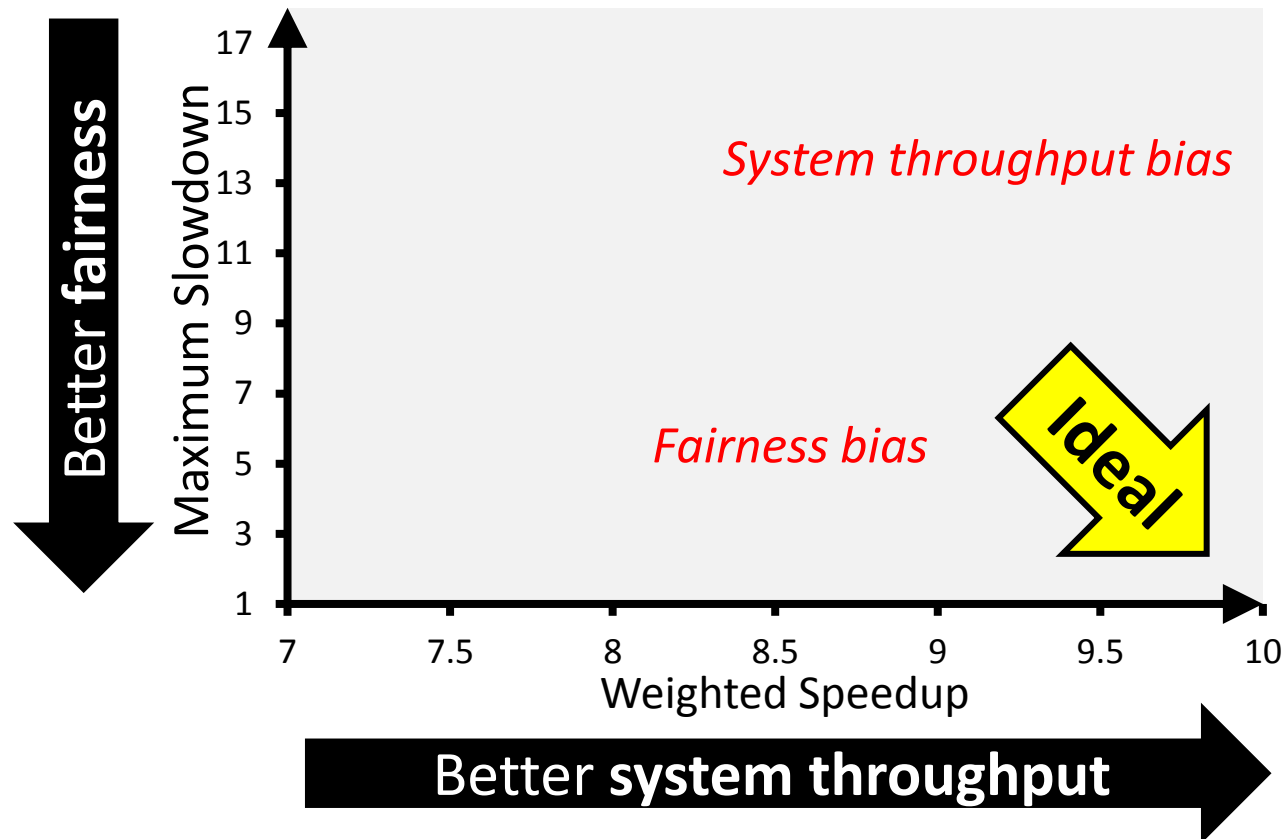
- Downsides:
 - ❑ Does not always prioritize the latency-sensitive applications

TCM: Thread Cluster Memory Scheduling

Yoongu Kim, Michael Papamichael, Onur Mutlu, and Mor Harchol-Balter,
**"Thread Cluster Memory Scheduling:
Exploiting Differences in Memory Access Behavior"**
43rd International Symposium on Microarchitecture (MICRO),
pages 65-76, Atlanta, GA, December 2010. [Slides \(pptx\)](#) [\(pdf\)](#)

Throughput vs. Fairness

24 cores, 4 memory controllers, 96 workloads



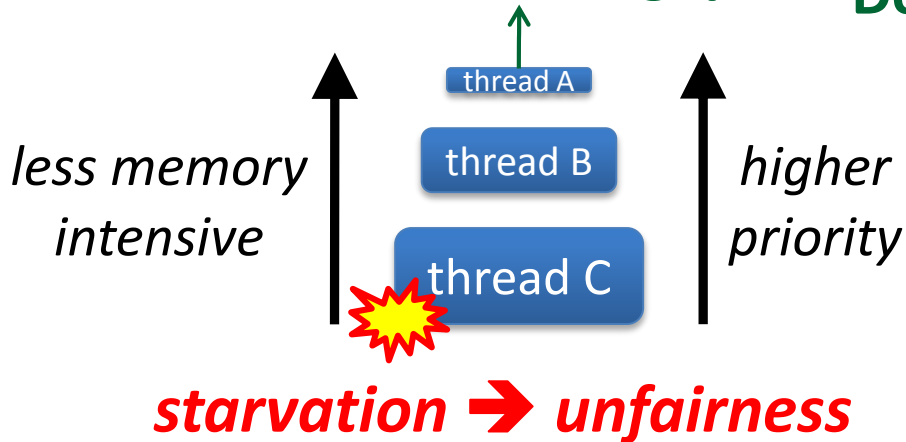
No previous memory scheduling algorithm provides both the best fairness and system throughput

Throughput vs. Fairness

Throughput biased approach

Prioritize less memory-intensive threads

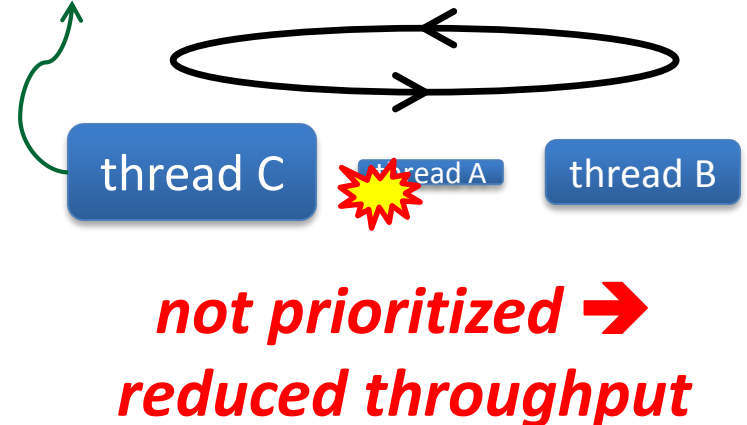
Good for throughput



Fairness biased approach

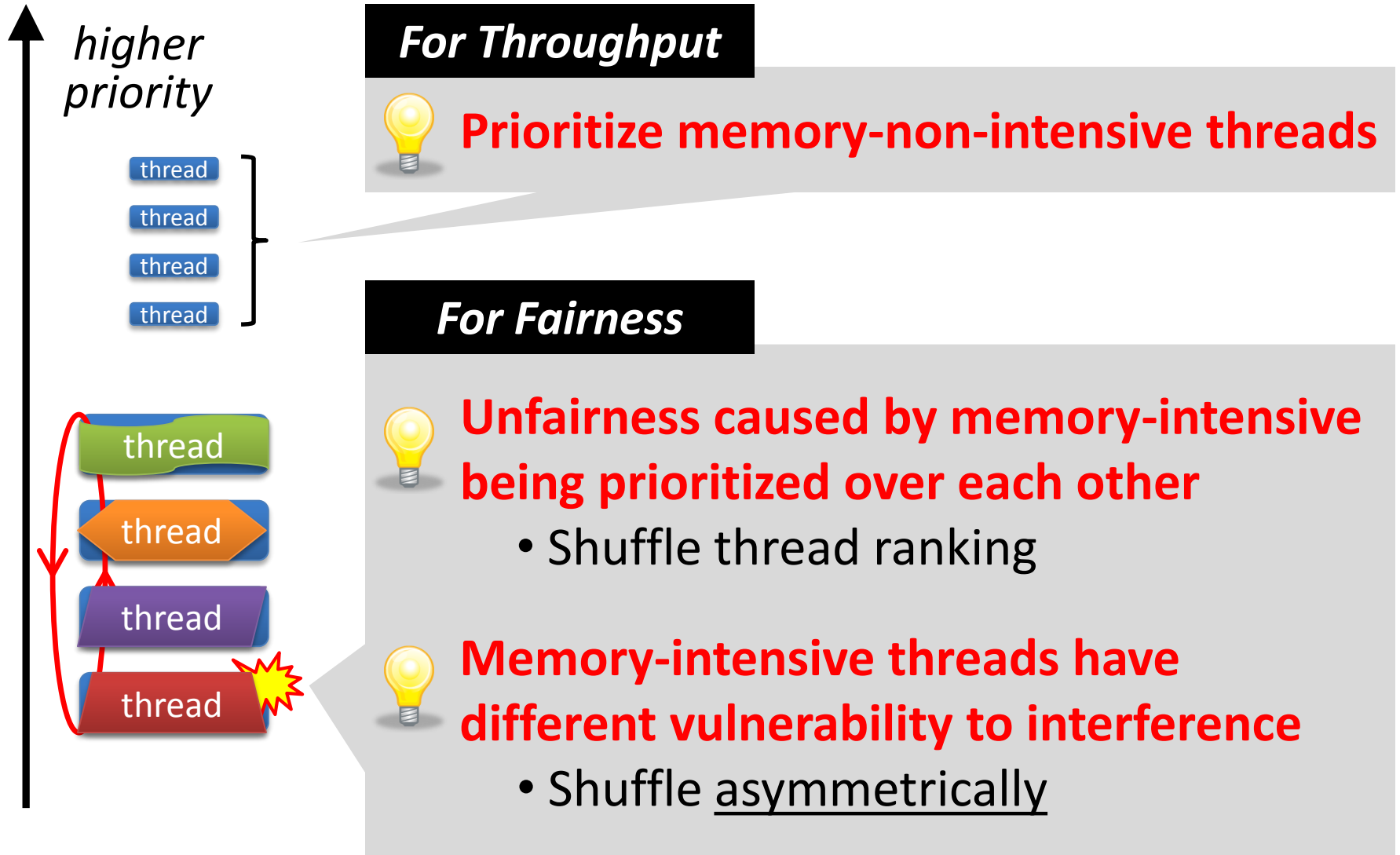
Take turns accessing memory

Does not starve



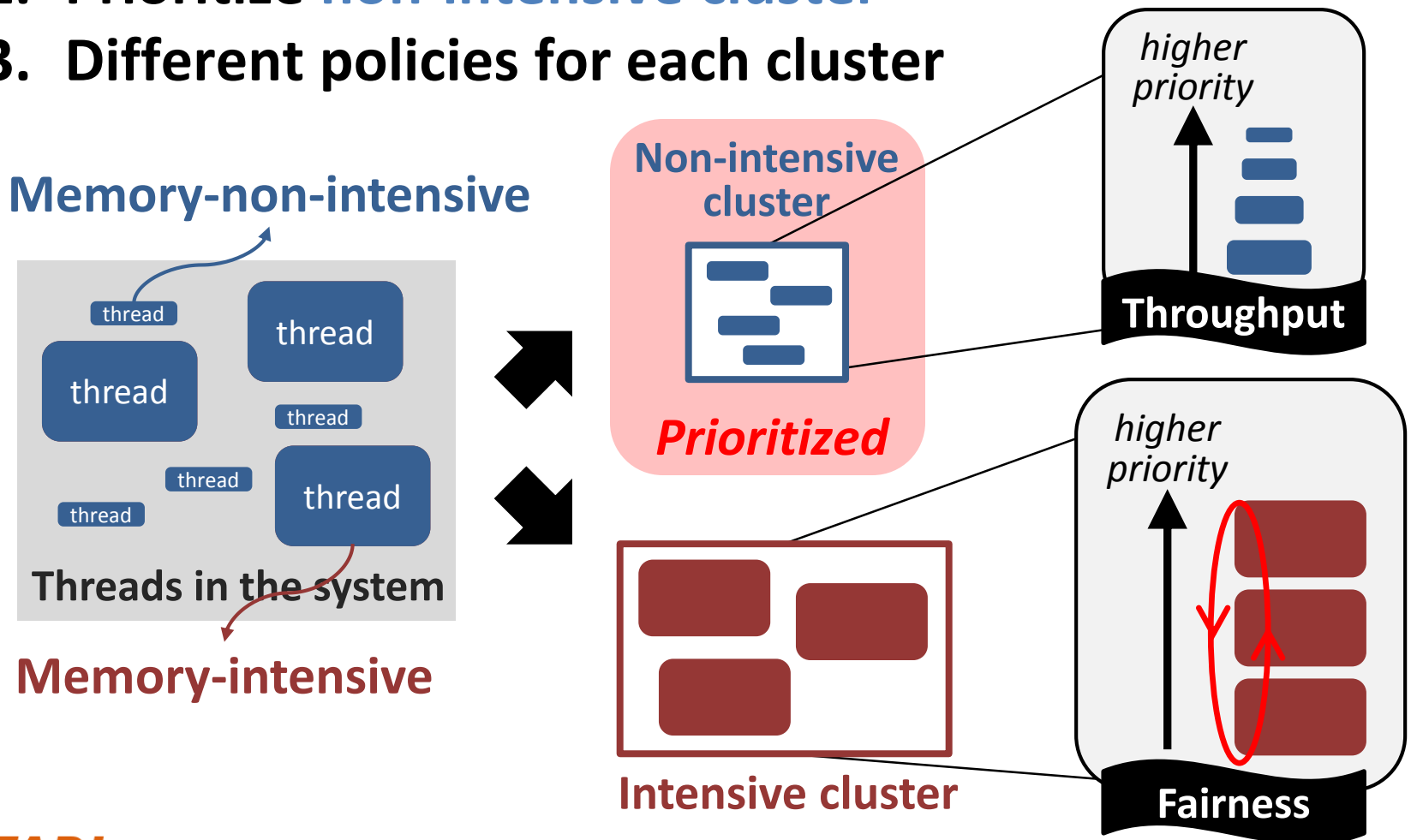
Single policy for all threads is insufficient

Achieving the Best of Both Worlds



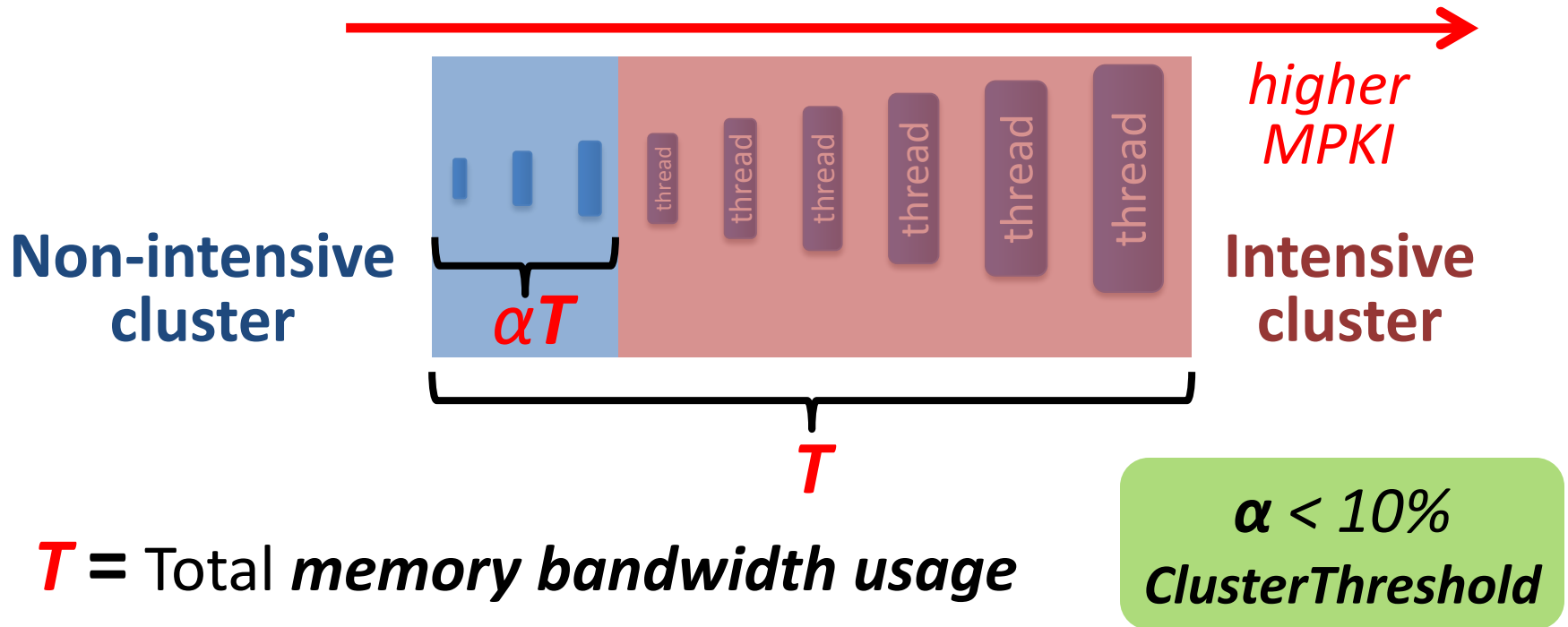
Thread Cluster Memory Scheduling [Kim+ MICRO'10]

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



Clustering Threads

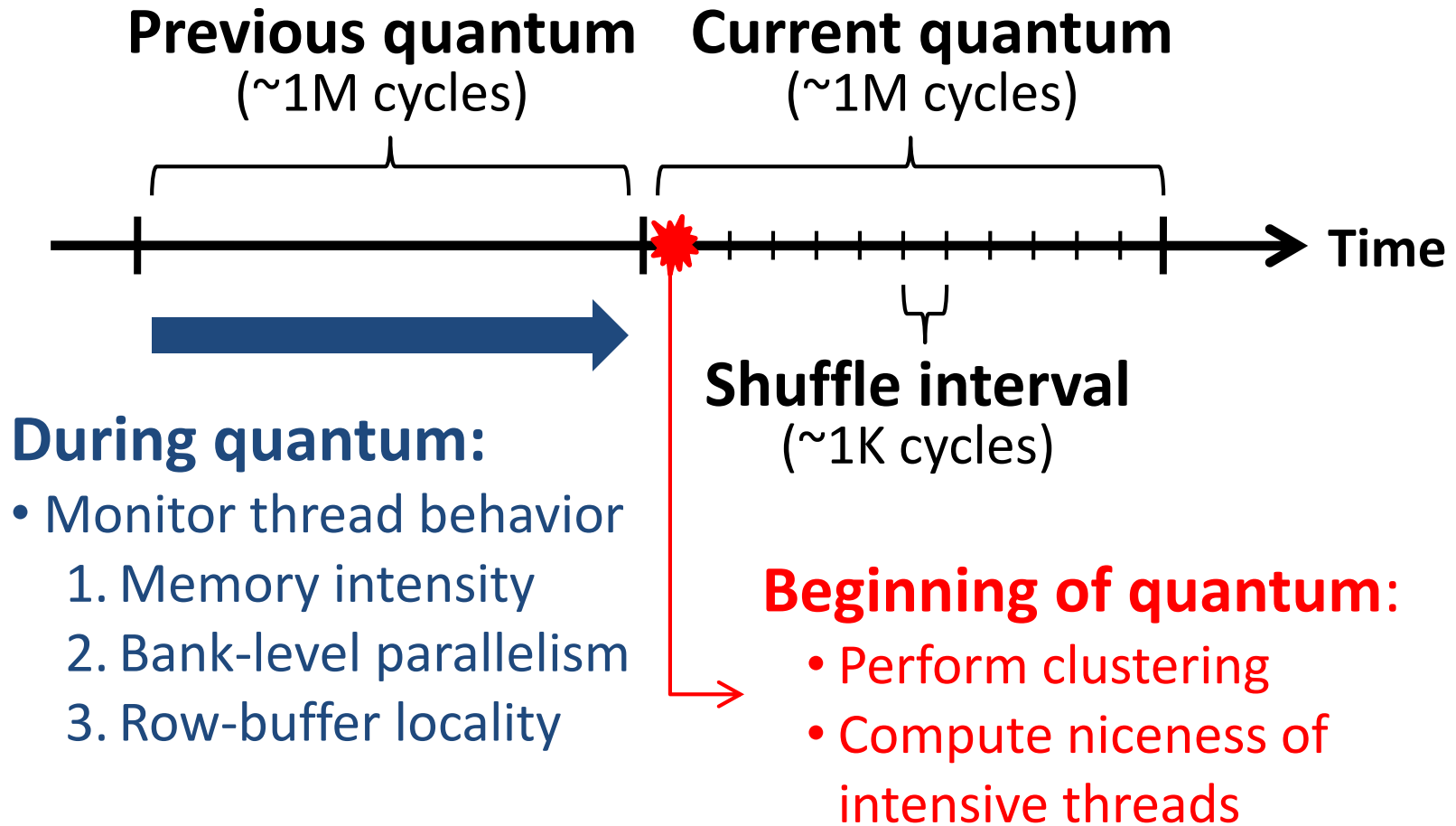
Step1 Sort threads by **MPKI** (misses per kiloinstruction)



T = Total *memory bandwidth usage*

Step2 Memory bandwidth usage αT divides clusters

TCM: Quantum-Based Operation



TCM: Scheduling Algorithm

1. Highest-rank: Requests from higher ranked threads prioritized

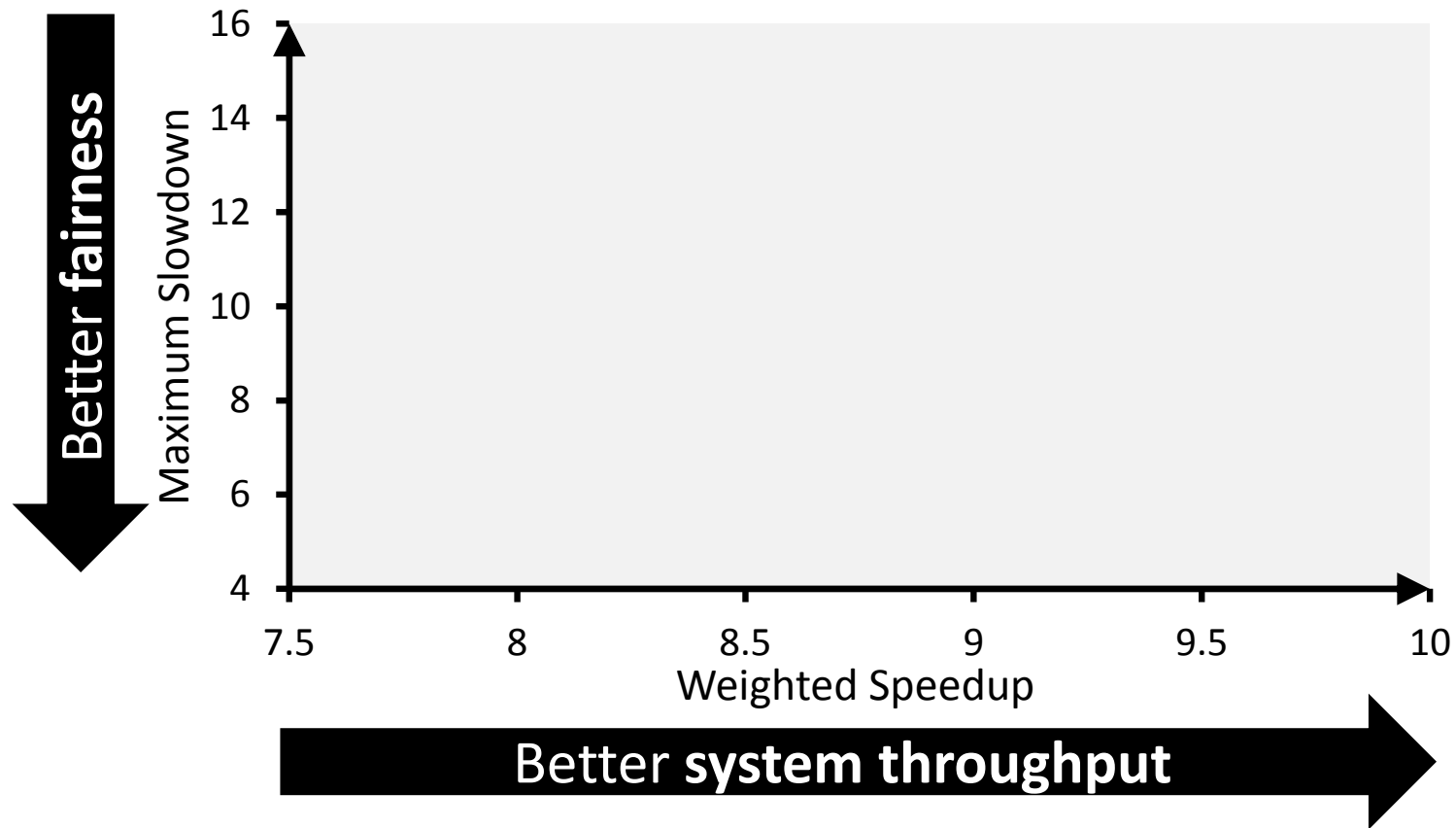
- **Non-Intensive** cluster > **Intensive** cluster
- **Non-Intensive** cluster: lower intensity → higher rank
- **Intensive** cluster: rank shuffling

2. Row-hit: Row-buffer hit requests are prioritized

3. Oldest: Older requests are prioritized

TCM: Throughput and Fairness

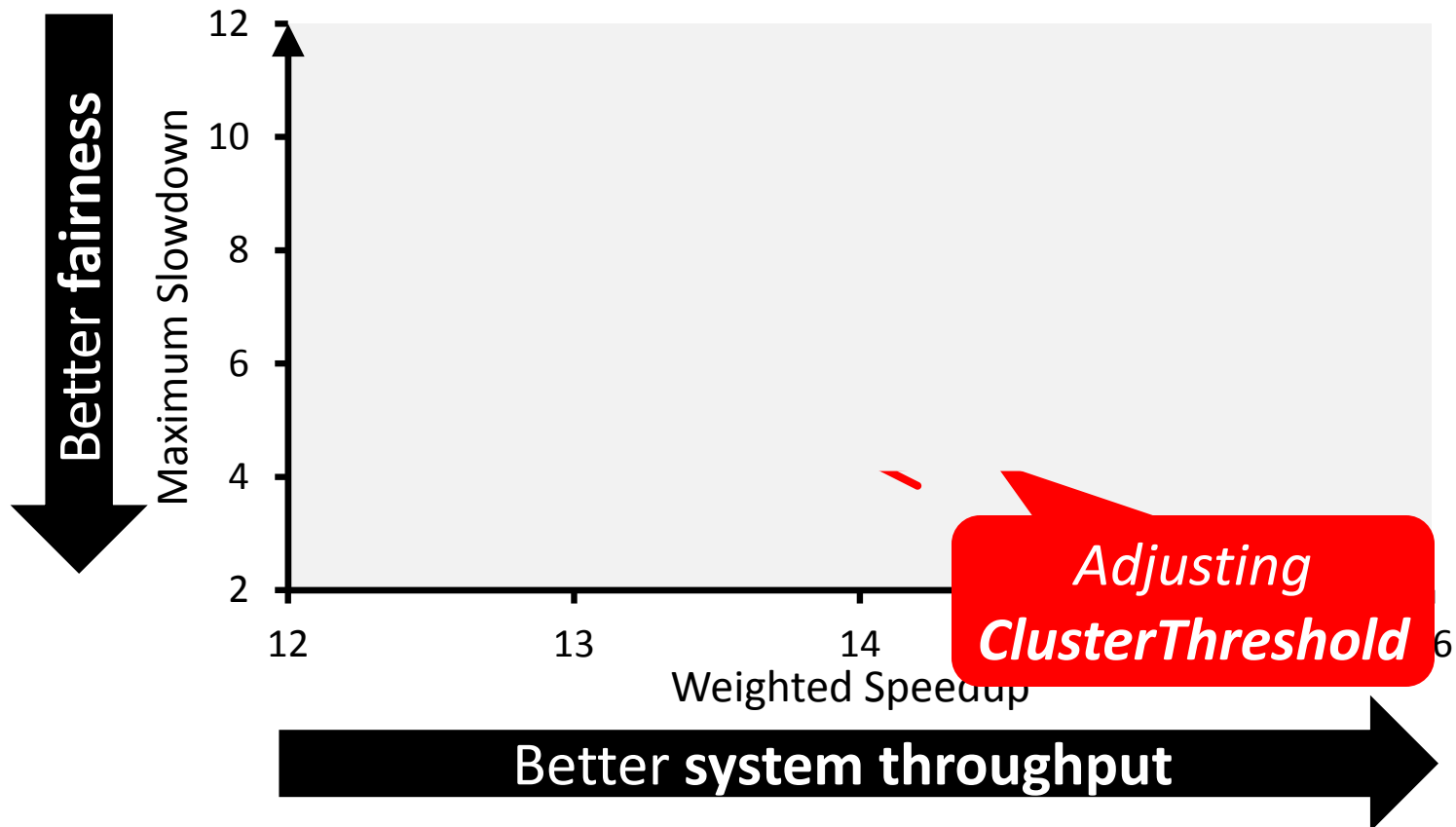
24 cores, 4 memory controllers, 96 workloads



*TCM, a heterogeneous scheduling policy,
provides best fairness and system throughput*

TCM: Fairness-Throughput Tradeoff

When configuration parameter is varied...



TCM allows robust fairness-throughput tradeoff

TCM Pros and Cons

■ Upsides:

- ❑ Provides both high fairness and high performance
- ❑ Caters to the needs for different types of threads (latency vs. bandwidth sensitive)
- ❑ (Relatively) simple

■ Downsides:

- ❑ Scalability to large buffer sizes?
- ❑ Robustness of clustering and shuffling algorithms?