18-742 Fall 2012
Parallel Computer Architecture
Lecture 24: Main Memory Management

Prof. Onur Mutlu
Carnegie Mellon University
11/9/2012

New Review Assignments

- Due: Tuesday, November 13, 11:59pm.
  - Mutlu and Moscibroda, "Parallelism-Aware Batch Scheduling: Enhancing both Performance and Fairness of Shared DRAM Systems," ISCA 2008.
  - Kim et al., "Thread Cluster Memory Scheduling: Exploiting Differences in Memory Access Behavior," MICRO 2010.

- Due: Thursday, November 15, 11:59pm.
  - Ebrahimi et al., "Fairness via Source Throttling: A Configurable and High-Performance Fairness Substrate for Multi-Core Memory Systems," ASPLOS 2010.
  - Muralidhara et al., "Reducing Memory Interference in Multicore Systems via Application-Aware Memory Channel Partitioning," MICRO 2011.

Reminder: Old Review Assignment

- Was Due: Sunday, November 4, 11:59pm.
  - H. T. Kung, "Why Systolic Architectures?," IEEE Computer 1982.

Reminder: Literature Survey Process

- Done in groups: your research project group is likely ideal
- Step 1: Pick 3 or more research papers
  - Broadly related to your research project
- Step 2: Send me the list of papers with links to pdf copies (by Sunday, November 11)
  - I need to approve the 3 papers
  - We will iterate to ensure convergence on the list
- Step 3: Prepare a 2-page writeup on the 3 papers
- Step 3: Prepare a 15-minute presentation on the 3 papers
  - Total time: 15-minute talk + 5-minute Q&A
  - Talk should focus on insights and tradeoffs
- Step 4: Deliver the presentation in front of class (dates: November 26-28 or December 3-7) and turn in your writeup (due date: December 1)

Reminder: Literature Survey Guidelines

- The goal is to
  - Understand the solution space and tradeoffs
  - Deeply analyze and synthesize three papers
    - Analyze: Describe individual strengths and weaknesses
    - Synthesize: Find commonalities and common strengths and weaknesses, categorize the solutions with respect to criteria
  - Explain how they relate to your project, how they can enhance it, or why your solution will be better

- Read the papers very carefully
  - Attention to detail is important

Reminder: Literature Survey Talk

- The talk should clearly convey at least the following:
  - The problem: What is the general problem targeted by the papers and what are the specific problems?
  - The solutions: What are the key ideas and solution approaches of the proposed papers?
  - Key results and insights: What are the key results, insights, and conclusions of the papers?
  - Tradeoffs and analyses: How do the solutions differ or interact with each other? Can they be combined? What are the tradeoffs between them? This is where you will need to analyze the approaches and find a way to synthesize a common framework to describe and qualitatively compare&contrast the approaches.
  - Comparison to your project: How do these approaches relate to your project? Why is your approach novel, different, better, or complementary?
  - Key conclusions and new ideas: What have you learned? Do you have new ideas/approaches based on what you have learned?

Last Lecture

- End Dataflow
- Systolic Arrays

Today

- Begin shared resource management

- Main memory as a shared resource
  - QoS-aware memory systems
  - Memory request scheduling
  - Memory channel partitioning
  - Source throttling

Other Readings: Shared Main Memory

- Required
  - Moscibroda and Mutlu, "Memory Performance Attacks," USENIX Security 2007.
  - Mutlu and Moscibroda, "Stall-Time Fair Memory Access Scheduling for Chip Multiprocessors," MICRO 2007.
  - Kim et al., "ATLAS: A Scalable and High-Performance Scheduling Algorithm for Multiple Memory Controllers," HPCA 2010.
  - Muralidhara et al., "Reducing Memory Interference in Multicore Systems via Application-Aware Memory Channel Partitioning," MICRO 2011.
  - Ausavarungnirun et al., "Staged Memory Scheduling: Achieving High Performance and Scalability in Heterogeneous Systems," ISCA 2012.
  - Lee et al., "Prefetch-Aware DRAM Controllers," MICRO 2008.

- Recommended
  - Rixner et al., "Memory Access Scheduling," ISCA 2000.
  - Zheng et al., "Mini-Rank: Adaptive DRAM Architecture for Improving Memory Power Efficiency," MICRO 2008.
  - Ipek et al., "Self Optimizing Memory Controllers: A Reinforcement Learning Approach," ISCA 2008.

Resource Sharing

Resource Sharing Concept

- Idea: Instead of dedicating a hardware resource to a hardware context, allow multiple contexts to use it
  - Example resources: functional units, pipeline, caches, buses, memory
- Why?

+ Resource sharing improves utilization/efficiency → throughput
  - As we saw with (simultaneous) multithreading
  - When a resource is left idle by one thread, another thread can use it; no need to replicate shared data

+ Reduces communication latency
  - For example, shared data kept in the same cache in SMT

+ Compatible with the shared memory model

Resource Sharing Disadvantages

- Resource sharing results in contention for resources
  - When the resource is not idle, another thread cannot use it
  - If space is occupied by one thread, another thread needs to re-occupy it

- Sometimes reduces each or some thread's performance

  - Thread performance can be worse than when it is run alone

- Eliminates performance isolation → inconsistent performance across runs

  - Thread performance depends on co-executing threads

- Uncontrolled (free-for-all) sharing degrades QoS

  - Causes unfairness, starvation

Need for QoS and Shared Resource Mgmt.

- Why is unpredictable performance (or lack of QoS) bad?

- Makes programmer's life difficult
  - An optimized program can get low performance (and performance varies widely depending on co-runners)

- Causes discomfort to user
  - An important program can starve
  - Examples from shared software resources

- Makes system management difficult
  - How do we enforce a Service Level Agreement when hardware resources are sharing is uncontrollable?

Resource Sharing vs. Partitioning

- Sharing improves throughput
  - Better utilization of space

- Partitioning provides performance isolation (predictable performance)
  - Dedicated space

- Can we get the benefits of both?

- Idea: Design shared resources in a controllable/partitionable way

Shared Hardware Resources

- Memory subsystem (in both MT and CMP)
  - Non-private caches
  - Interconnects
  - Memory controllers, buses, banks

- I/O subsystem (in both MT and CMP)
  - I/O, DMA controllers
  - Ethernet controllers

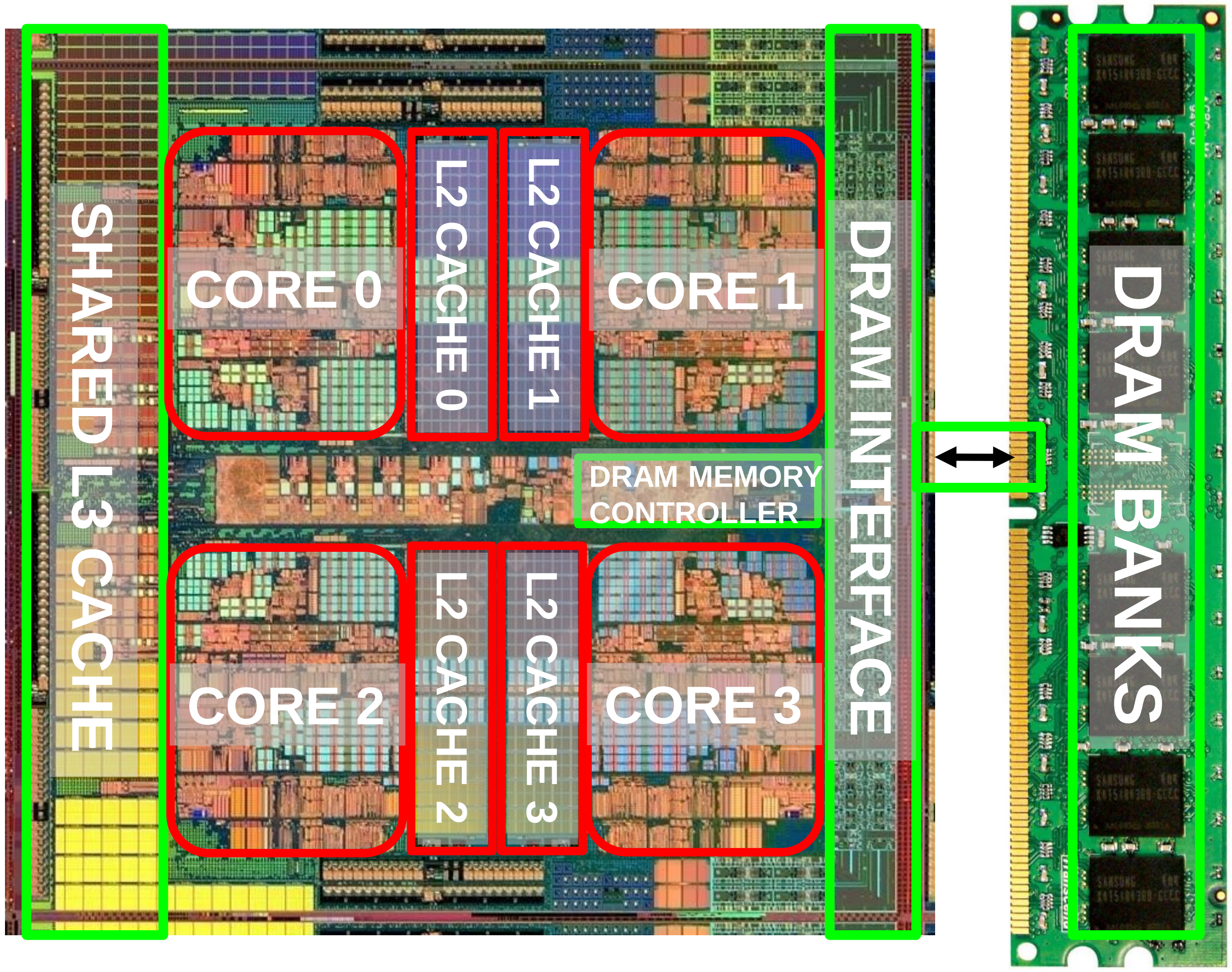
- Processor (in MT)
  - Pipeline resources
  - L1 caches

Resource Sharing Issues and Related Metrics

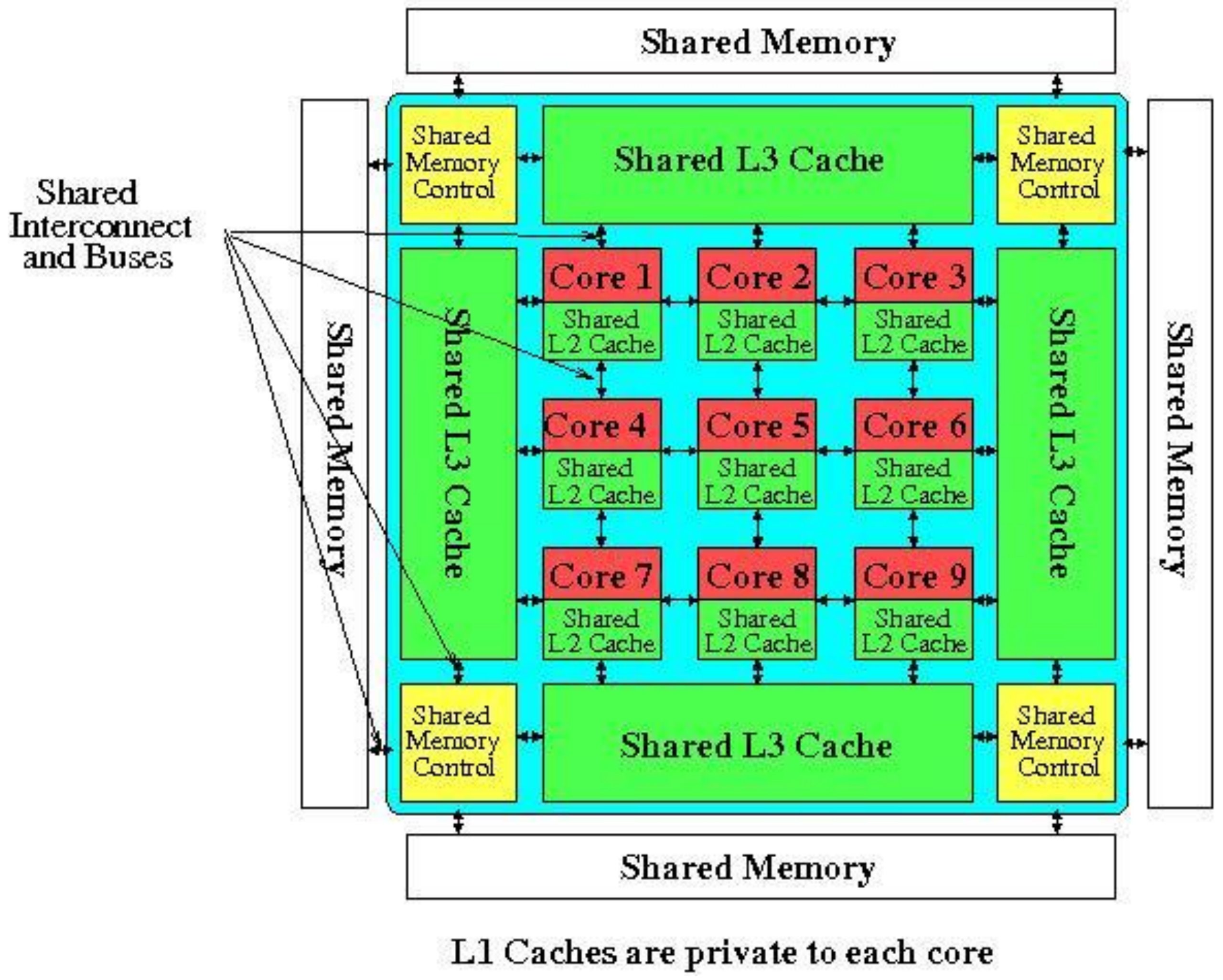
- System performance
- Fairness
- Per-application performance (QoS)
- Power
- Energy
- System cost
- Lifetime
- Reliability, effect of faults
- Security, information leakage

- Partitioning: Isolation between apps/threads
- Sharing (free for all): No isolation

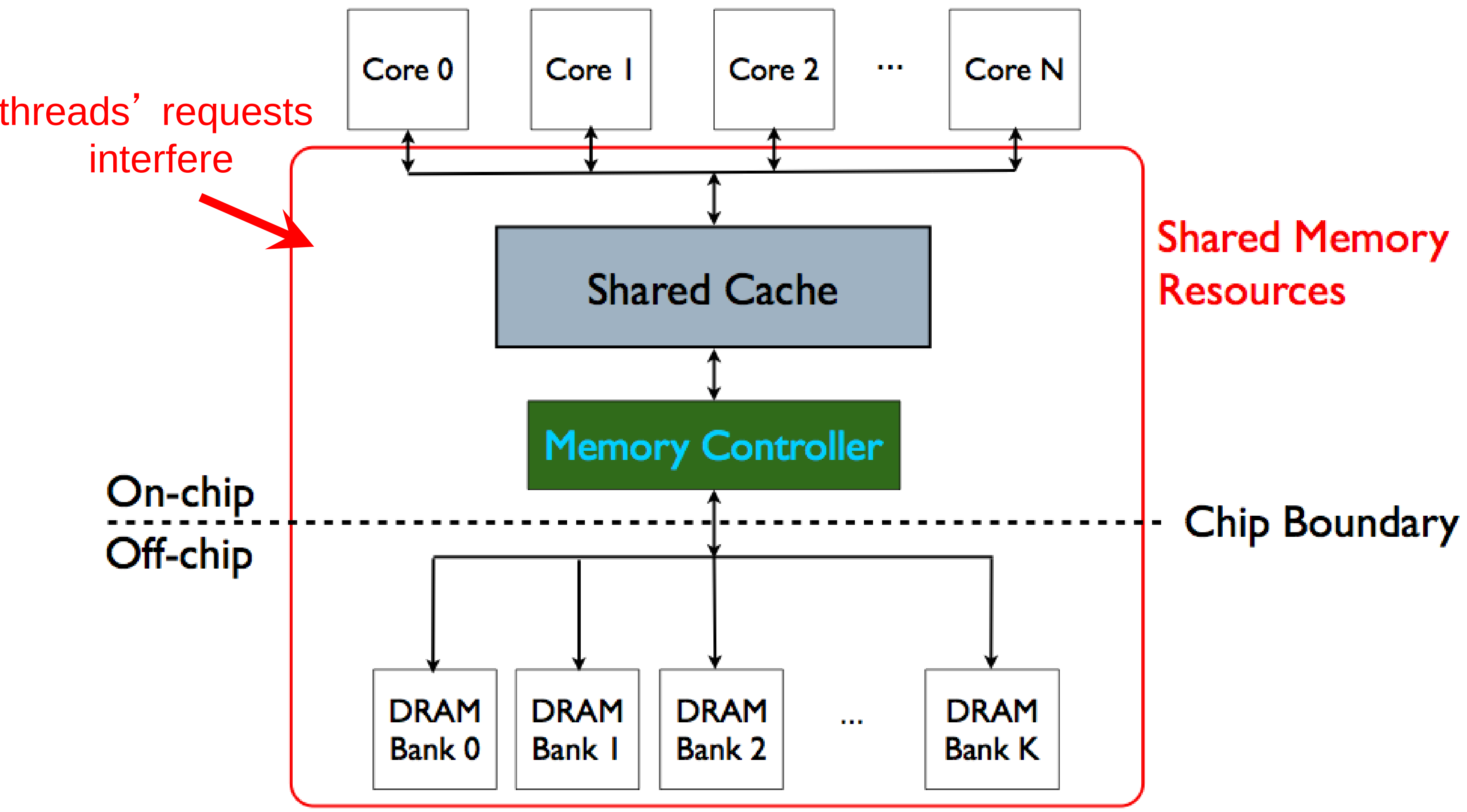
Main Memory in the System



Modern Memory Systems (Multi-Core)

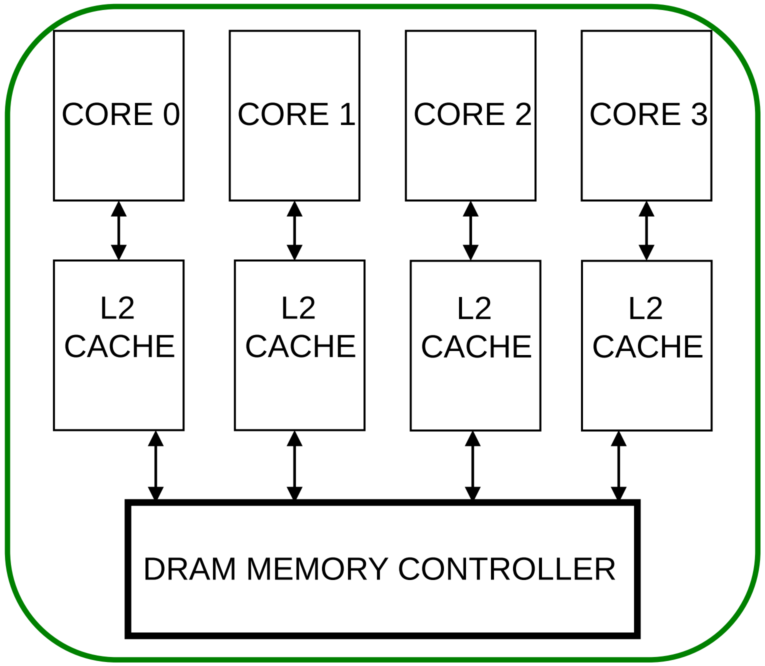


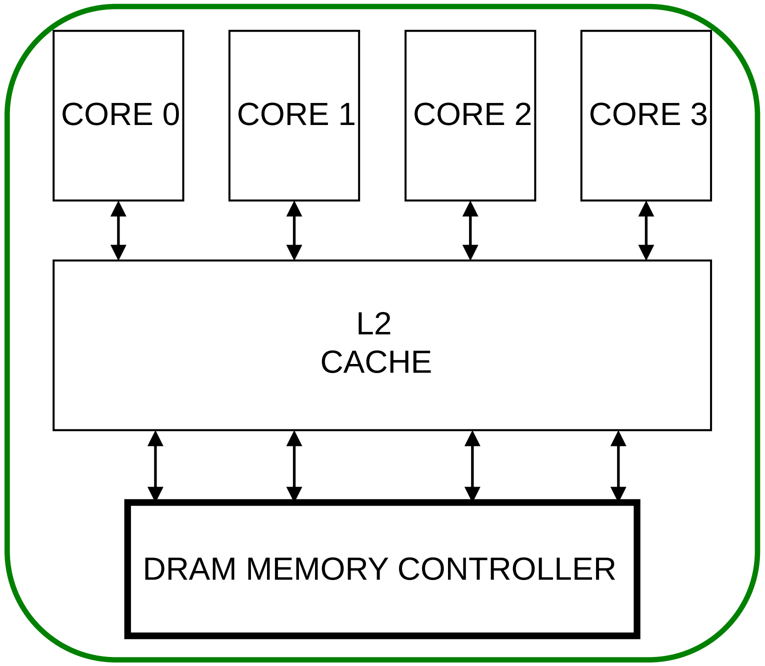
Memory System is the Major Shared Resource



Multi-core Issues in Caching

- How does the cache hierarchy change in a multi-core system?
- Private cache: Cache belongs to one core (a shared block can be in multiple caches)
- Shared cache: Cache is shared by multiple cores





Shared Caches Between Cores

- Advantages:
  - High effective capacity
  - Dynamic partitioning of available cache space
    - No fragmentation due to static partitioning
  - Easier to maintain coherence (a cache block is in a single location)
  - Shared data and locks do not ping pong between caches

- Disadvantages
  - Slower access
  - Cores incur conflict misses due to other cores' accesses
    - Misses due to inter-core interference
    - Some cores can destroy the hit rate of other cores
  - Guaranteeing a minimum level of service (or fairness) to each core is harder (how much space, how much bandwidth?)

Shared Caches: How to Share?

- Free-for-all sharing
  - Placement/replacement policies are the same as a single core system (usually LRU or pseudo-LRU)
  - Not thread/application aware
  - An incoming block evicts a block regardless of which threads the blocks belong to

- Problems
  - A cache-unfriendly application can destroy the performance of a cache friendly application
  - Not all applications benefit equally from the same amount of cache: free-for-all might prioritize those that do not benefit
  - Reduced performance, reduced fairness

Controlled Cache Sharing

- Utility based cache partitioning
  - Qureshi and Patt, "Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches," MICRO 2006.
  - Suh et al., "A New Memory Monitoring Scheme for Memory-Aware Scheduling and Partitioning," HPCA 2002.

- Fair cache partitioning
  - Kim et al., "Fair Cache Sharing and Partitioning in a Chip Multiprocessor Architecture," PACT 2004.

- Shared/private mixed cache mechanisms
  - Qureshi, "Adaptive Spill-Receive for Robust High-Performance Caching in CMPs," HPCA 2009.
  - Hardavellas et al., "Reactive NUCA: Near-Optimal Block Placement and Replication in Distributed Caches," ISCA 2009.

Readings: Shared Cache Management

- Required
  - Qureshi and Patt, "Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches," MICRO 2006.
  - Kim et al., "Fair Cache Sharing and Partitioning in a Chip Multiprocessor Architecture," PACT 2004.
  - Qureshi, "Adaptive Spill-Receive for Robust High-Performance Caching in CMPs," HPCA 2009.
  - Hardavellas et al., "Reactive NUCA: Near-Optimal Block Placement and Replication in Distributed Caches," ISCA 2009.
- Recommended
  - Kim et al., "An Adaptive, Non-Uniform Cache Structure for Wire-Delay Dominated On-Chip Caches," ASPLOS 2002.
  - Qureshi et al., "Adaptive Insertion Policies for High-Performance Caching," ISCA 2007.
  - Lin et al., "Gaining Insights into Multi-Core Cache Partitioning: Bridging the Gap between Simulation and Real Systems," HPCA 2008.

Main Memory As a Shared Resource

Sharing in Main Memory

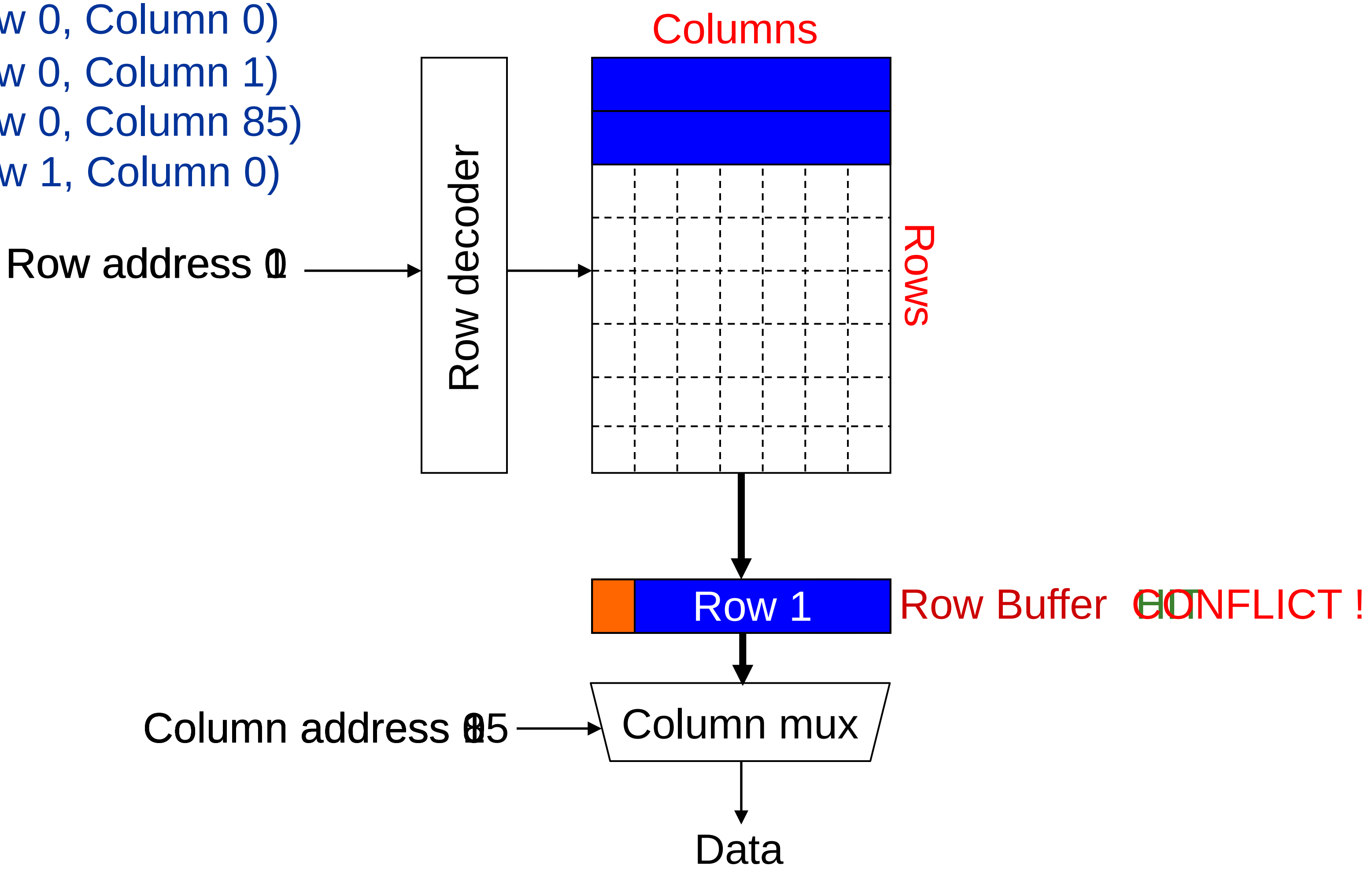
- Bandwidth sharing
  - Which thread/core to prioritize?
  - How to schedule requests?
  - How much bandwidth to allocate to each thread?

- Capacity sharing
  - How much memory capacity to allocate to which thread?
  - Where to map that memory? (row, bank, rank, channel)

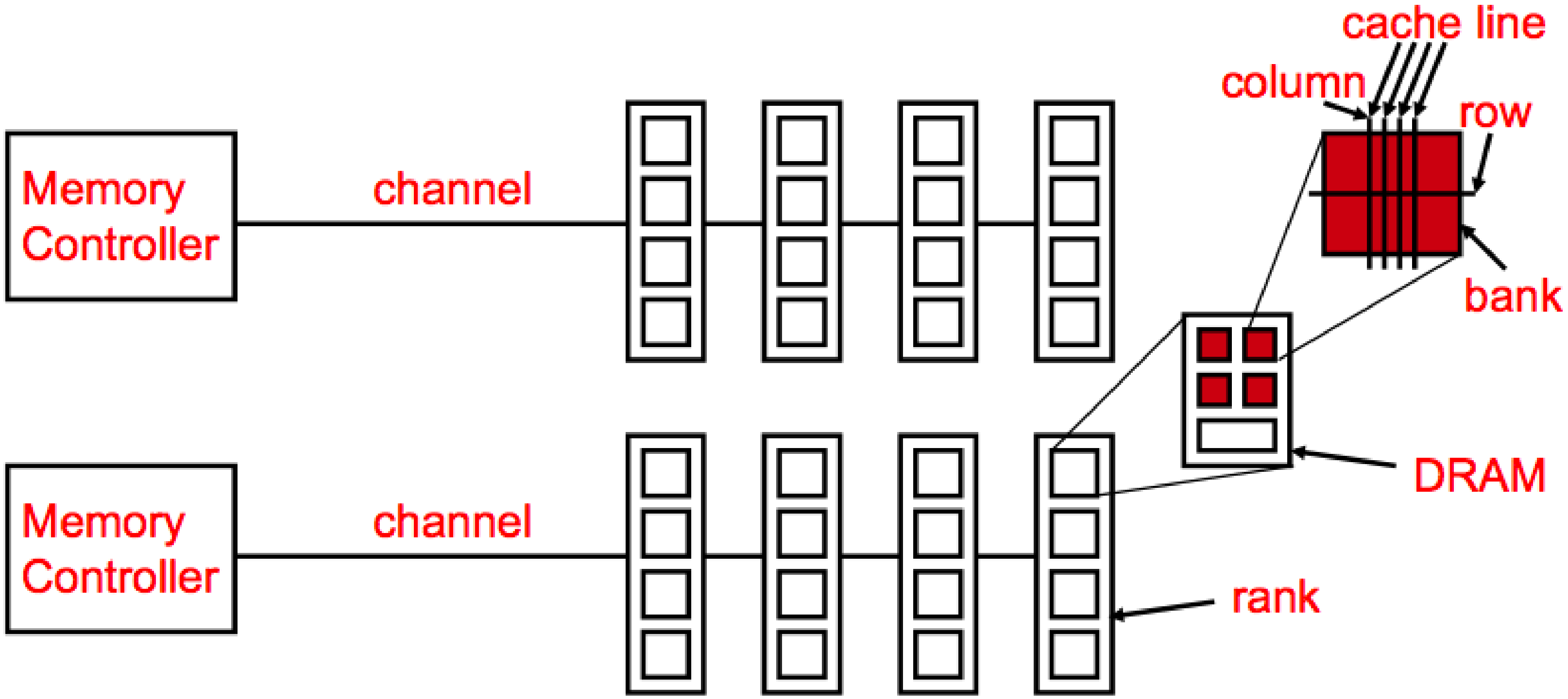
- Metrics for optimization
  - System performance
  - Fairness, QoS
  - Energy/power consumption

DRAM Bank Operation

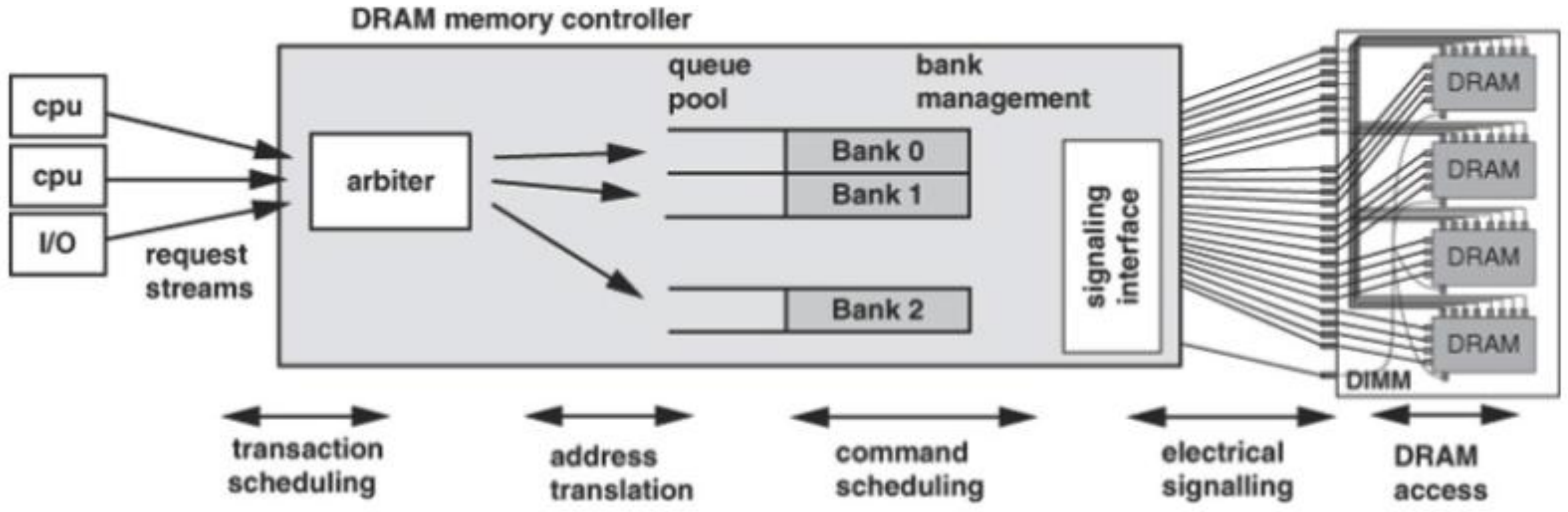
Access Address:
(Row 0, Column 0)
(Row 0, Column 1)
(Row 0, Column 85)
(Row 1, Column 0)



Generalized Memory Structure



Memory Controller



Inter-Thread Interference in DRAM

- 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

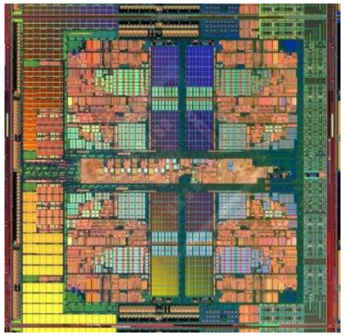
- Queueing/contention delays
  - Bank conflict, bus conflict, channel conflict, …

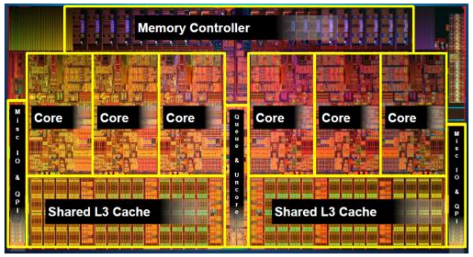
- Additional delays due to DRAM constraints
  - Called "protocol overhead"
  - Examples
    - Row conflicts
    - Read-to-write and write-to-read delays

- Loss of intra-thread parallelism

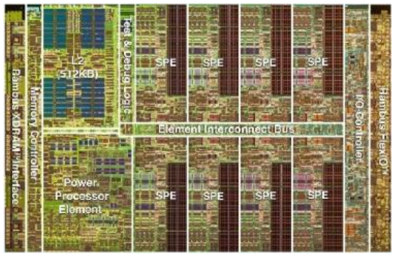
Trend: Many Cores on Chip

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

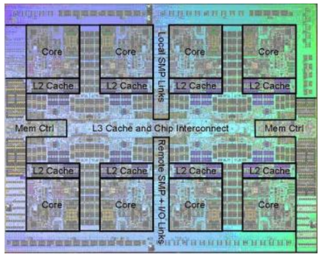

AMD Barcelona
4 cores



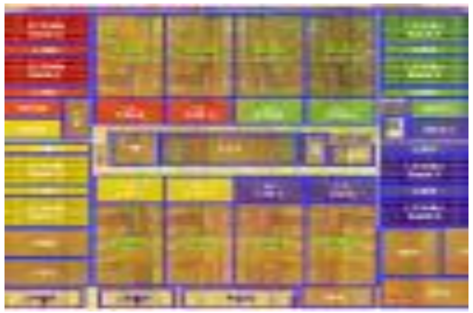
Intel Core i7
8 cores

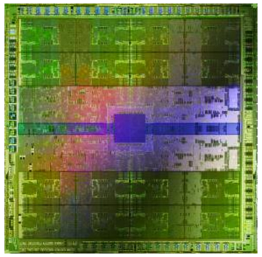


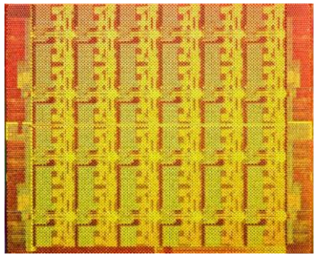
IBM Cell BE
8+1 cores

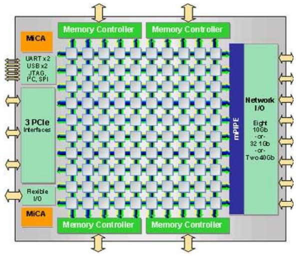


IBM POWER7
8 cores


Sun Niagara II
8 cores


Nvidia Fermi
448 "cores"


Intel SCC
48 cores, networked

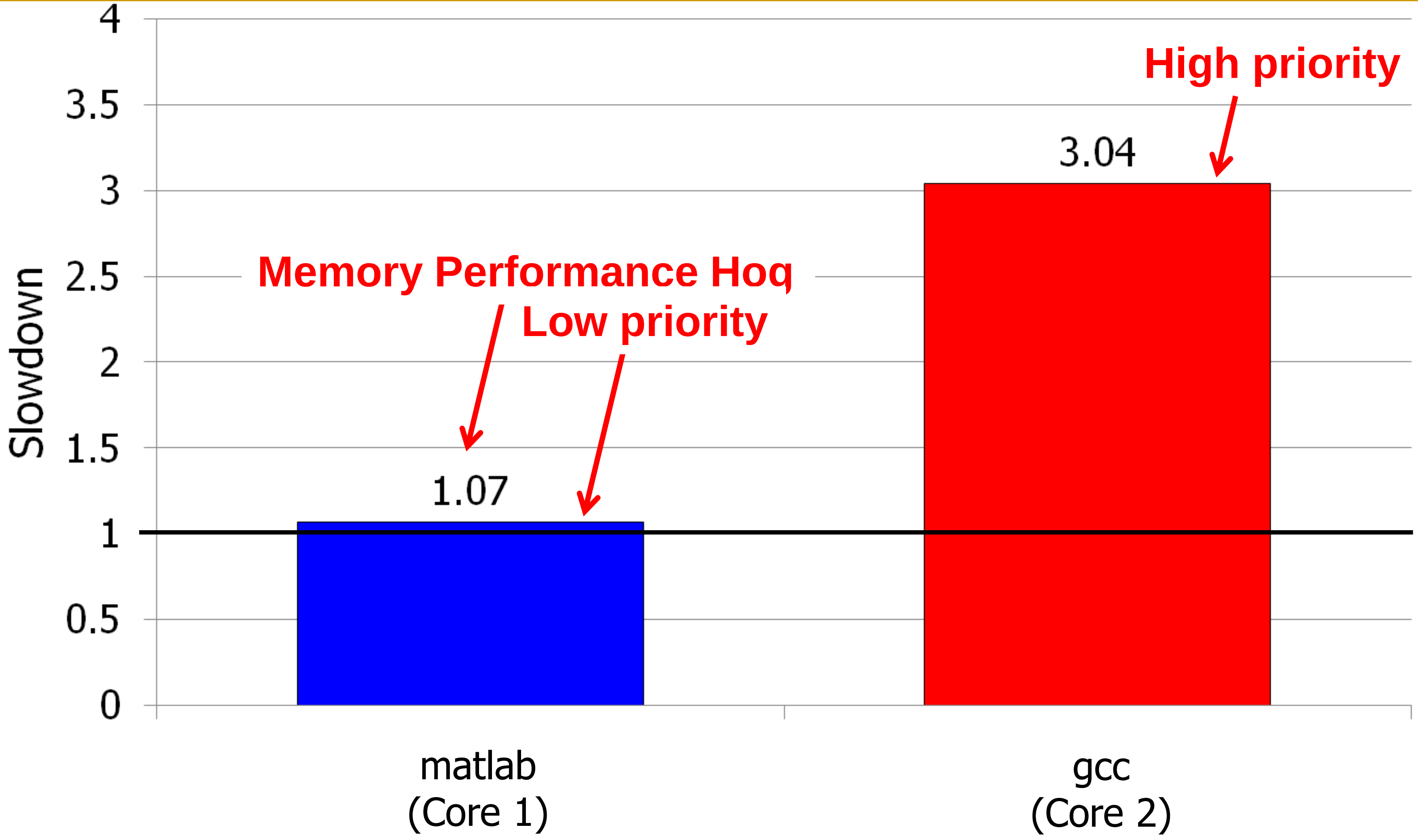


Tilera TILE Gx
100 cores, networked

Many Cores on Chip

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

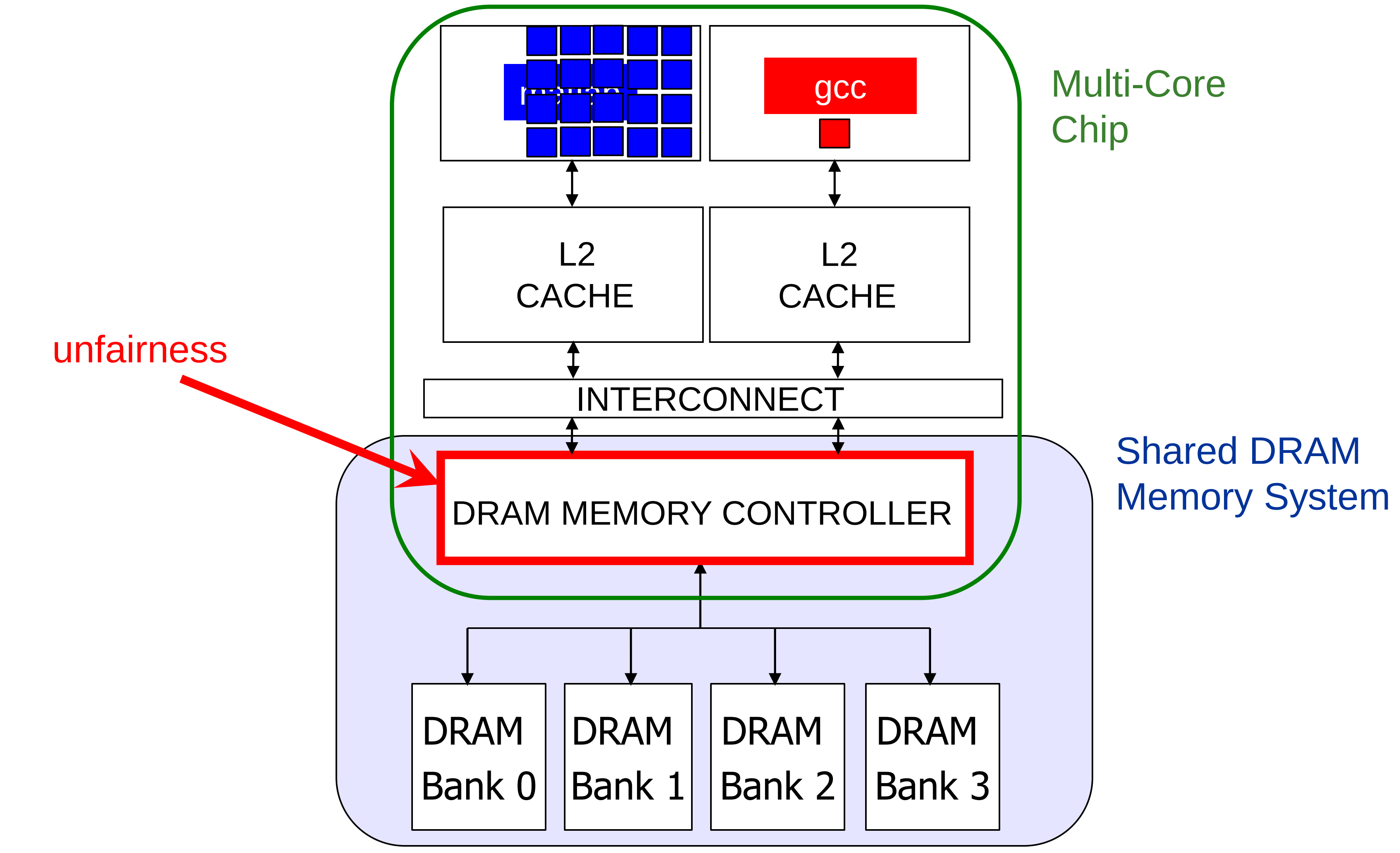
(Un)expected Slowdowns



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

SAFARI

Why? Uncontrolled Memory Interference



A Memory Performance Hog

```
// initialize large arrays A, B

for (j=0; j<N; j++) {
    index = j*linesize;  streaming
    A[index] = B[index];
    …
}
```

```
// initialize large arrays A, B

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

STREAM

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

RANDOM

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

Moscibroda and Mutlu, "Memory Performance Attacks," USENIX Security 2007.

DRAM Controllers

- A row-conflict memory access takes significantly longer than a row-hit access

- Current controllers take advantage of the row buffer

- Commonly used scheduling policy (FR-FCFS) [Rixner 2000]*

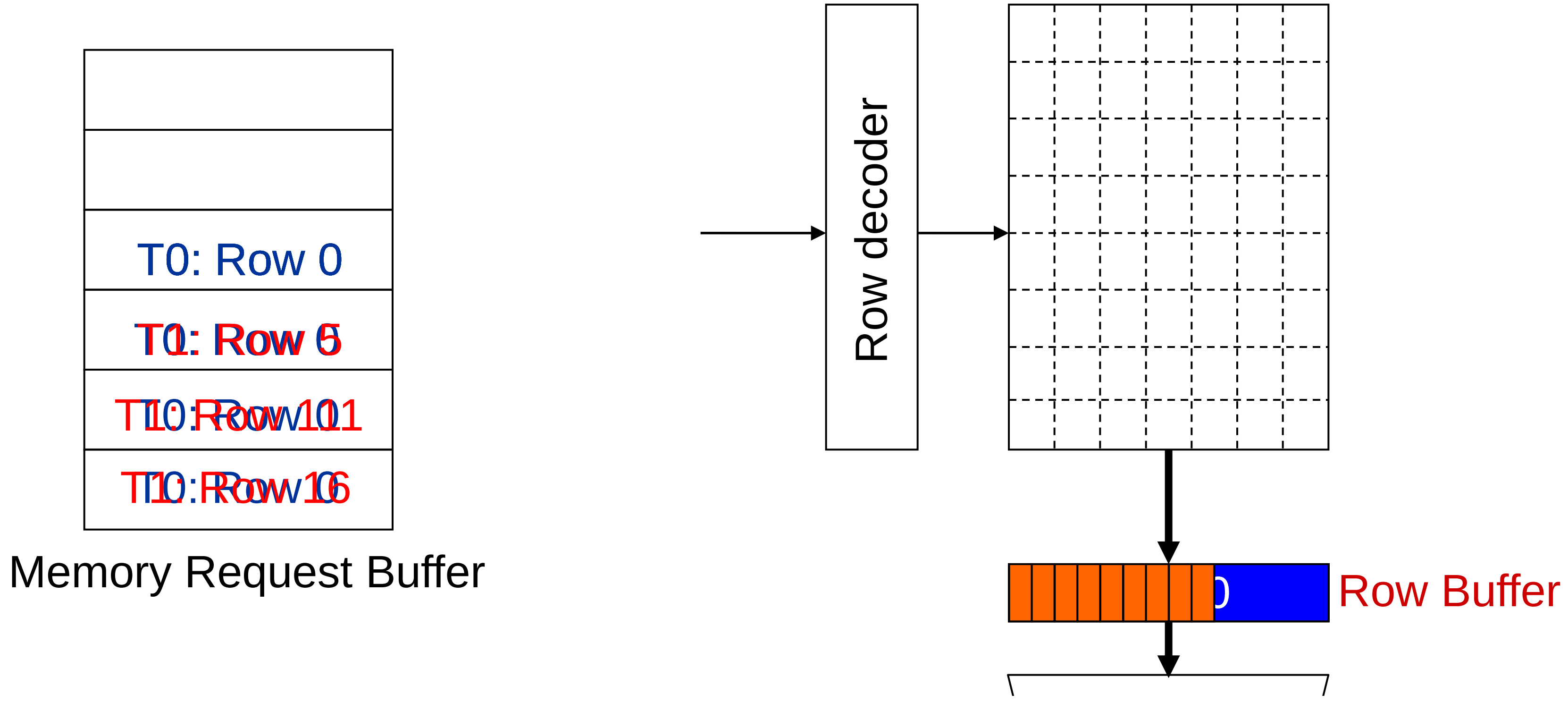
  (1) Row-hit first: Service row-hit memory accesses first

  (2) Oldest-first: Then service older accesses first

- This scheduling policy aims to maximize DRAM throughput
  - But, it is unfair when multiple threads share the DRAM system

*Rixner et al., "Memory Access Scheduling," ISCA 2000.
*Zuravleff and Robinson, "Controller for a synchronous DRAM …," US Patent 5,630,096, May 1997.

What Does the Memory Hog Do?

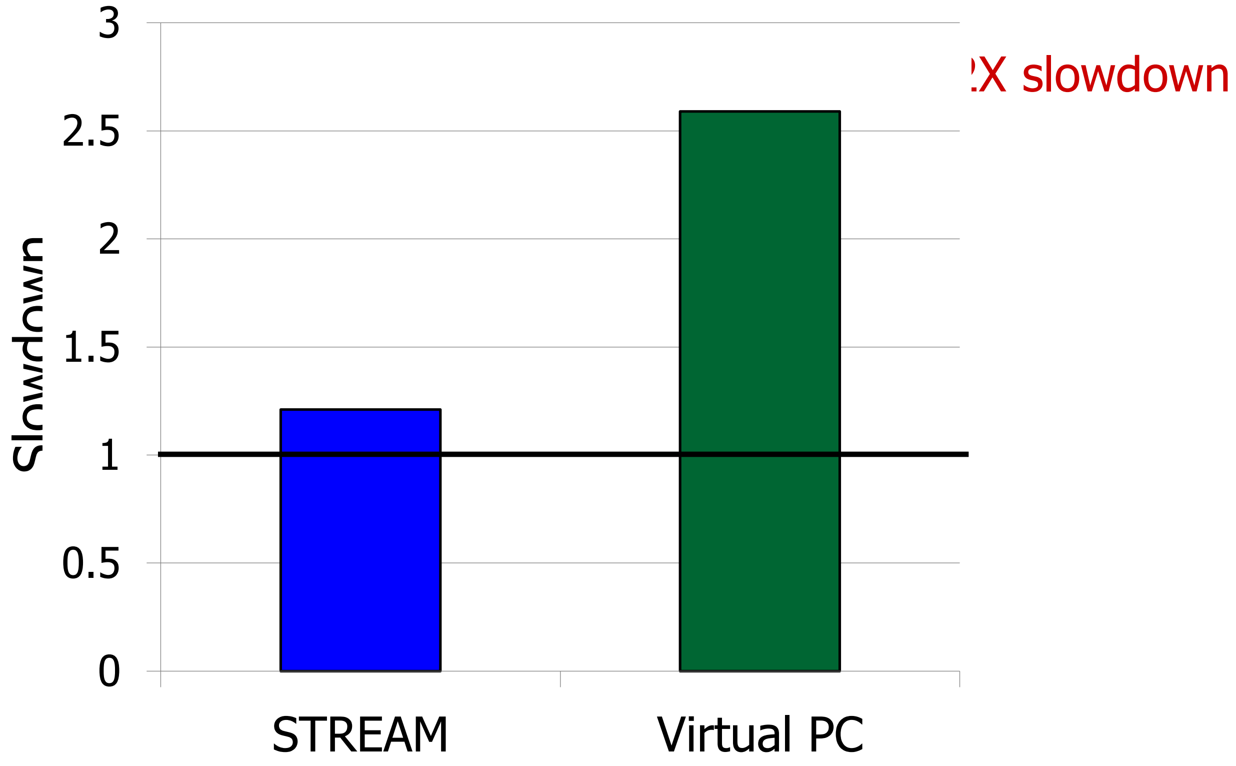


Row size: 8KB, cache block size: 64B

128 (8KB/64B) requests of T0 serviced before T1

Moscibroda and Mutlu, "Memory Performance Attacks," USENIX Security 2007.

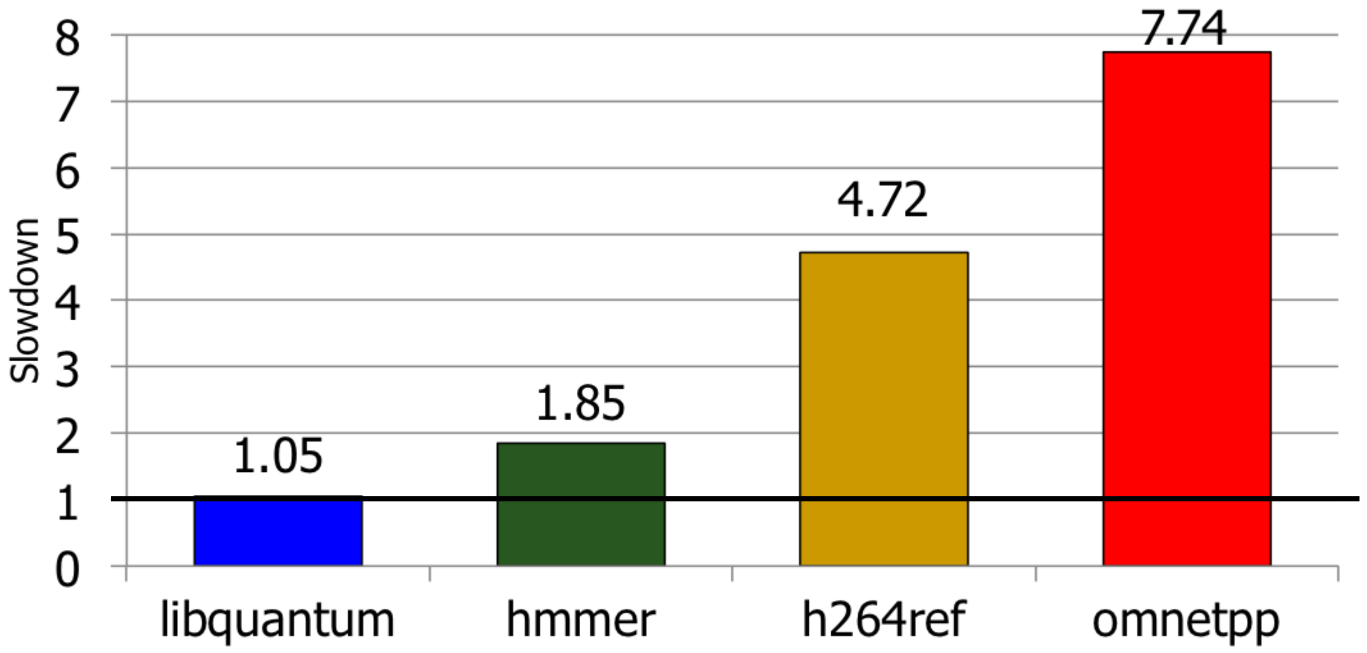
Effect of the Memory Performance Hog



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.

Greater Problem with More Cores

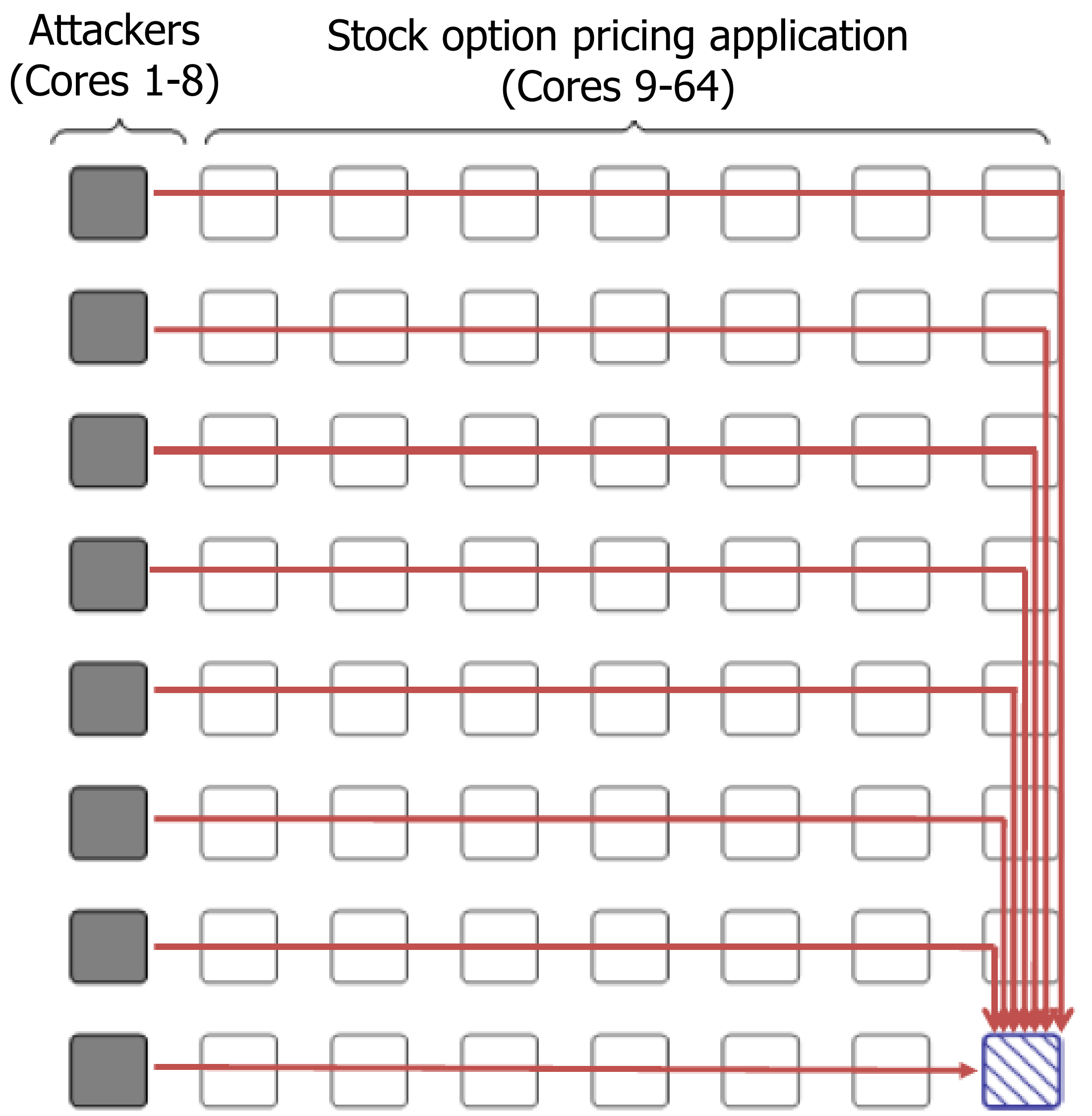


- Vulnerable to denial of service (DoS) [Usenix Security'07]
- Unable to enforce priorities or SLAs [MICRO'07,'10,'11, ISCA'08'11'12, ASPLOS'10]
- Low system performance [IEEE Micro Top Picks '09,'11a,'11b,'12]

Uncontrollable, unpredictable system

SAFARI

Distributed DoS in Networked Multi-Core Systems



Cores connected via packet-switched routers on chip

~5000X slowdown

Grot, Hestness, Keckler, Mutlu, "Preemptive virtual clock: A Flexible, Efficient, and Cost-effective QOS Scheme for Networks-on-Chip," MICRO 2009.

SAFARI

How Do We Solve The Problem?

- Inter-thread interference is uncontrolled in all memory resources
  - Memory controller
  - Interconnect
  - Caches

- We need to control it
  - i.e., design an interference-aware (QoS-aware) memory system

Solution: QoS-Aware, Predictable Memory

- Problem: Memory interference is uncontrolled → uncontrollable, unpredictable, vulnerable system

- Goal: We need to control it → Design a QoS-aware system

- Solution: Hardware/software cooperative memory QoS
  - Hardware designed to provide a configurable fairness substrate
    - Application-aware memory scheduling, partitioning, throttling
  - Software designed to configure the resources to satisfy different QoS goals

  - E.g., fair, programmable memory controllers and on-chip networks provide QoS and predictable performance
    [2007-2012, Top Picks'09,'11a,'11b,'12]

QoS-Aware Memory Systems: Challenges

- How do we reduce inter-thread interference?
  - Improve system performance and core utilization
  - Reduce request serialization and core starvation

- How do we control inter-thread interference?
  - Provide mechanisms to enable system software to enforce QoS policies
  - While providing high system performance

- How do we make the memory system configurable/flexible?
  - Enable flexible mechanisms that can achieve many goals
    - Provide fairness or throughput when needed
    - Satisfy performance guarantees when needed

Designing QoS-Aware Memory Systems: Approaches

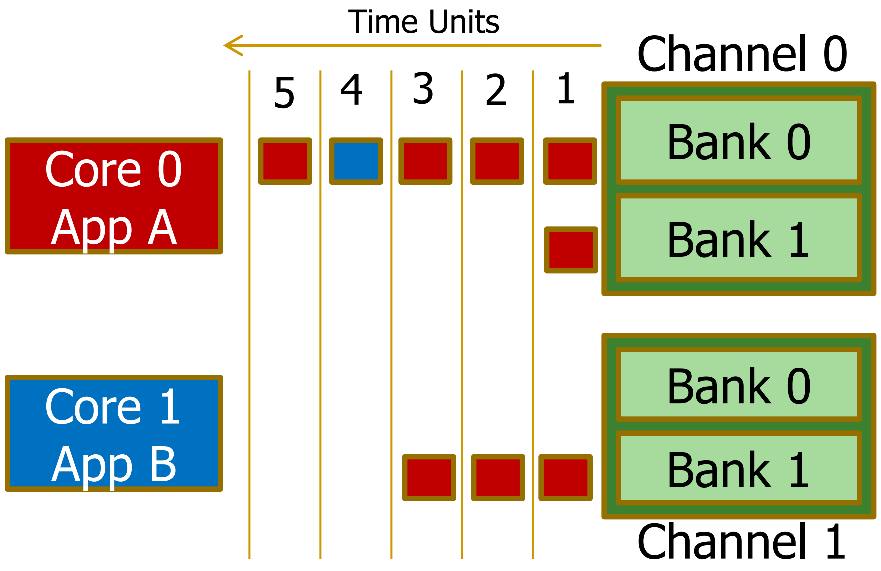
- Smart resources: Design each shared resource to have a configurable interference control/reduction mechanism
  - QoS-aware memory controllers [Mutlu+ MICRO'07] [Moscibroda+, Usenix Security'07] [Mutlu+ ISCA'08, Top Picks'09] [Kim+ HPCA'10] [Kim+ MICRO'10, Top Picks'11] [Ebrahimi+ ISCA'11, MICRO'11] [Ausavarungnirun+, ISCA'12]
  - QoS-aware interconnects [Das+ MICRO'09, ISCA'10, Top Picks '11] [Grot+ MICRO'09, ISCA'11, Top Picks '12]
  - QoS-aware caches

- Dumb resources: Keep each resource free-for-all, but reduce/control interference by injection control or data mapping
  - Source throttling to control access to memory system [Ebrahimi+ ASPLOS'10, ISCA'11, TOCS'12] [Ebrahimi+ MICRO'09] [Nychis+ HotNets'10]
  - QoS-aware data mapping to memory controllers [Muralidhara+ MICRO'11]
  - QoS-aware thread scheduling to cores

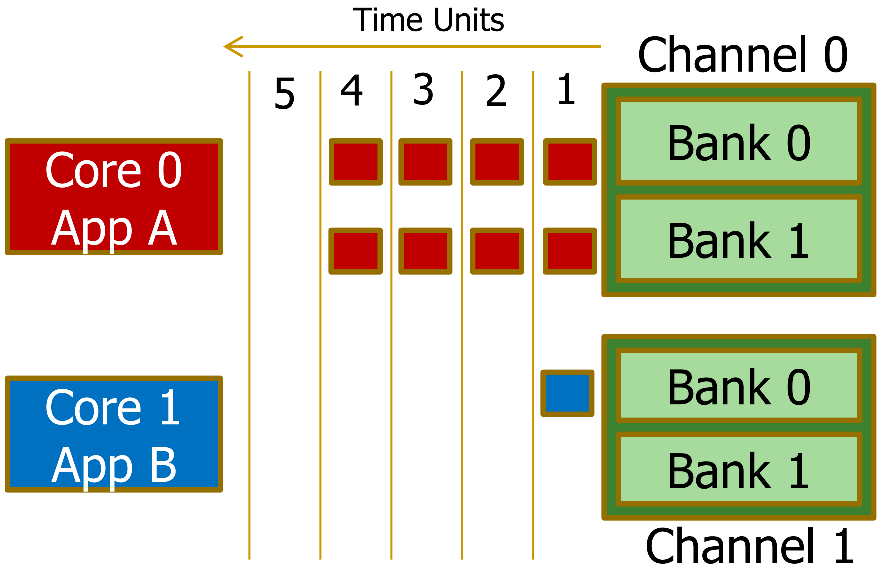
SAFARI

A Mechanism to Reduce Memory Interference

- Memory Channel Partitioning
  - Idea: System software maps badly-interfering applications' pages to different channels [Muralidhara+, MICRO'11]



Conventional Page Mapping



Channel Partitioning

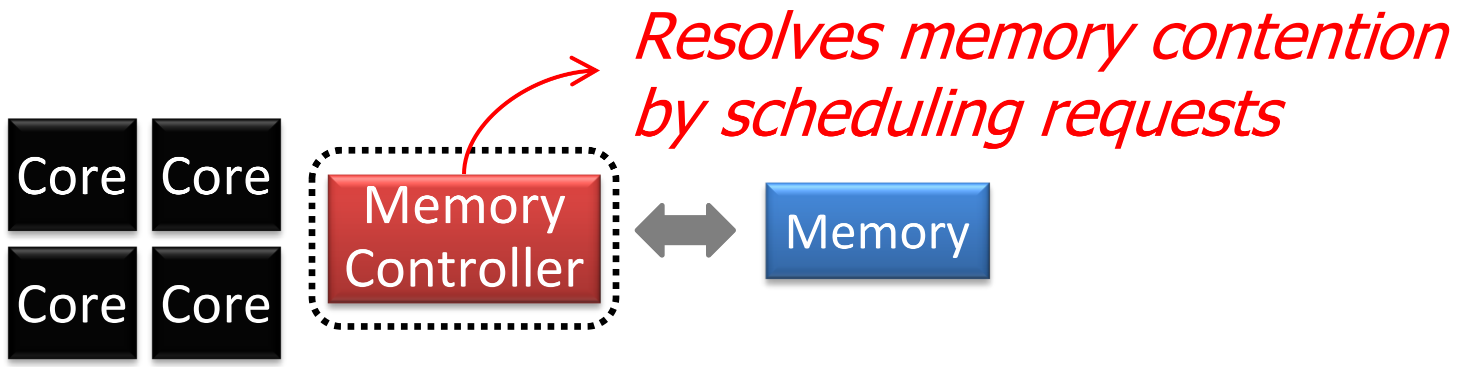
- Separate data of low/high intensity and low/high row-locality applications
- Especially effective in reducing interference of threads with "medium" and "heavy" memory intensity
  - 11% higher performance over existing systems (200 workloads)

Designing QoS-Aware Memory Systems: Approaches

- Smart resources: Design each shared resource to have a configurable interference control/reduction mechanism
  - QoS-aware memory controllers [Mutlu+ MICRO'07] [Moscibroda+, Usenix Security'07] [Mutlu+ ISCA'08, Top Picks'09] [Kim+ HPCA'10] [Kim+ MICRO'10, Top Picks'11] [Ebrahimi+ ISCA'11, MICRO'11] [Ausavarungnirun+, ISCA'12]
  - QoS-aware interconnects [Das+ MICRO'09, ISCA'10, Top Picks '11] [Grot+ MICRO'09, ISCA'11, Top Picks '12]
  - QoS-aware caches

- Dumb resources: Keep each resource free-for-all, but reduce/control interference by injection control or data mapping
  - Source throttling to control access to memory system [Ebrahimi+ ASPLOS'10, ISCA'11, TOCS'12] [Ebrahimi+ MICRO'09] [Nychis+ HotNets'10]
  - QoS-aware data mapping to memory controllers [Muralidhara+ MICRO'11]
  - QoS-aware thread scheduling to cores

QoS-Aware Memory 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

QoS-Aware Memory Scheduling: Evolution

- Stall-time fair memory scheduling [Mutlu+ MICRO'07]
  - Idea: Estimate and balance thread slowdowns
  - Takeaway: Proportional thread progress improves performance, especially when threads are "heavy" (memory intensive)
- Parallelism-aware batch scheduling [Mutlu+ ISCA'08, Top Picks'09]
  - Idea: Rank threads and service in rank order (to preserve bank parallelism); batch requests to prevent starvation
  - Takeaway: Preserving within-thread bank-parallelism improves performance; request batching improves fairness
- ATLAS memory scheduler [Kim+ HPCA'10]
  - Idea: Prioritize threads that have attained the least service from the memory scheduler
  - Takeaway: Prioritizing "light" threads improves performance

QoS-Aware Memory Scheduling: Evolution

- Thread cluster memory scheduling [Kim+ MICRO'10]
  - Idea: Cluster threads into two groups (latency vs. bandwidth sensitive); prioritize the latency-sensitive ones; employ a fairness policy in the bandwidth sensitive group
  - Takeaway: Heterogeneous scheduling policy that is different based on thread behavior maximizes both performance and fairness

- Integrated Memory Channel Partitioning and Scheduling [Muralidhara+ MICRO'11]
  - Idea: Only prioritize very latency-sensitive threads in the scheduler; mitigate all other applications' interference via channel partitioning
  - Takeaway: Intelligently ombining application-aware channel partitioning and memory scheduling provides better performance than either

QoS-Aware Memory Scheduling: Evolution

- Parallel application memory scheduling [Ebrahimi+ MICRO'11]
  - Idea: Identify and prioritize limiter threads of a multithreaded application in the memory scheduler; provide fast and fair progress to non-limiter threads
  - Takeaway: Carefully prioritizing between limiter and non-limiter threads of a parallel application improves performance

- Staged memory scheduling [Ausavarungnirun+ ISCA'12]
  - Idea: Divide the functional tasks of an application-aware memory scheduler into multiple distinct stages, where each stage is significantly simpler than a monolithic scheduler
  - Takeaway: Staging enables the design of a scalable and relatively simpler application-aware memory scheduler that works on very large request buffers

QoS-Aware Memory Scheduling: Evolution

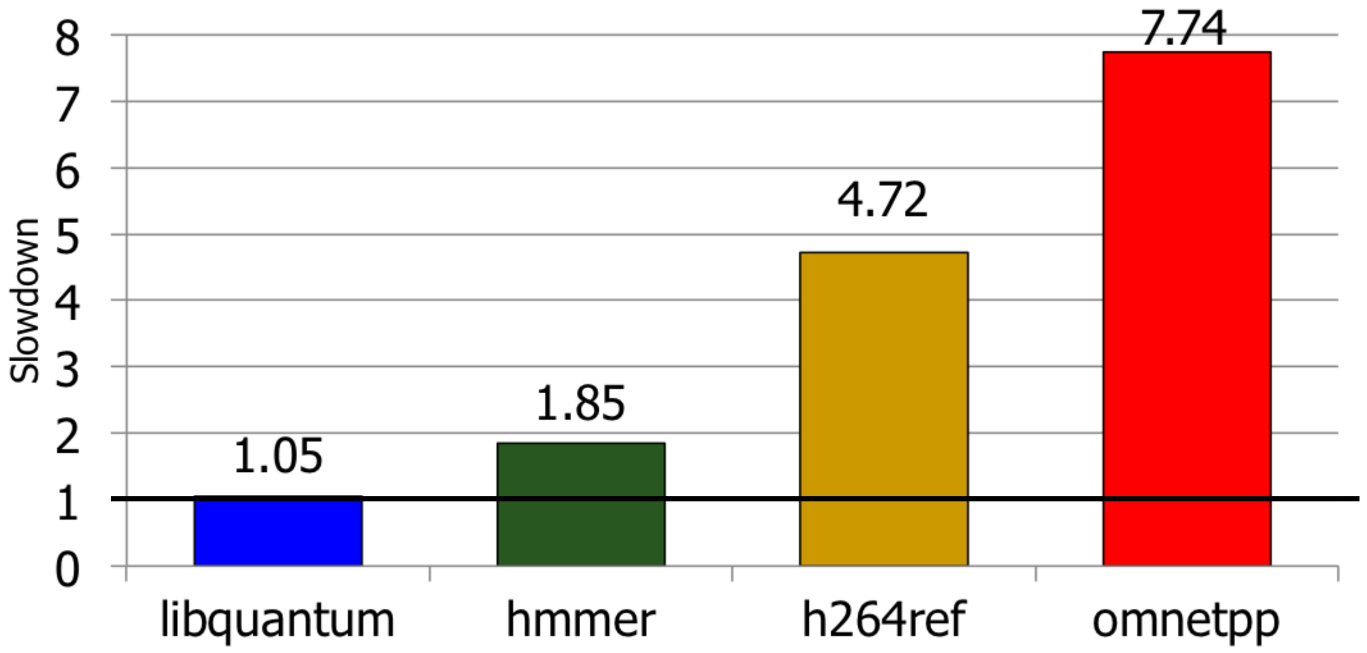
- Prefetch-aware shared resource management [Ebrahimi+ ISCA'12] [Ebrahimi+ MICRO'09] [Lee+ MICRO'08]
  - Idea: Prioritize prefetches depending on how they affect system performance; even accurate prefetches can degrade performance of the system
  - Takeaway: Carefully controlling and prioritizing prefetch requests improves performance and fairness

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)

STFM Micro 2007 Talk

The Problem: Unfairness



- Vulnerable to denial of service (DoS) [Usenix Security'07]
- Unable to enforce priorities or SLAs [MICRO'07,'10,'11, ISCA'08'11'12, ASPLOS'10]
- Low system performance [IEEE Micro Top Picks '09,'11a,'11b,'12]

Uncontrollable, unpredictable system

SAFARI

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_{shared}/ST_{alone}**
  - Relative increase in stall-time

- *Stall-Time Fair Memory scheduler (STFM)* aims to equalize Memory-slowdown for interfering threads, without sacrificing performance
  - Considers inherent DRAM performance of each thread
  - Aims to allow proportional progress of threads

STFM Scheduling Algorithm [MICRO' 07]

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

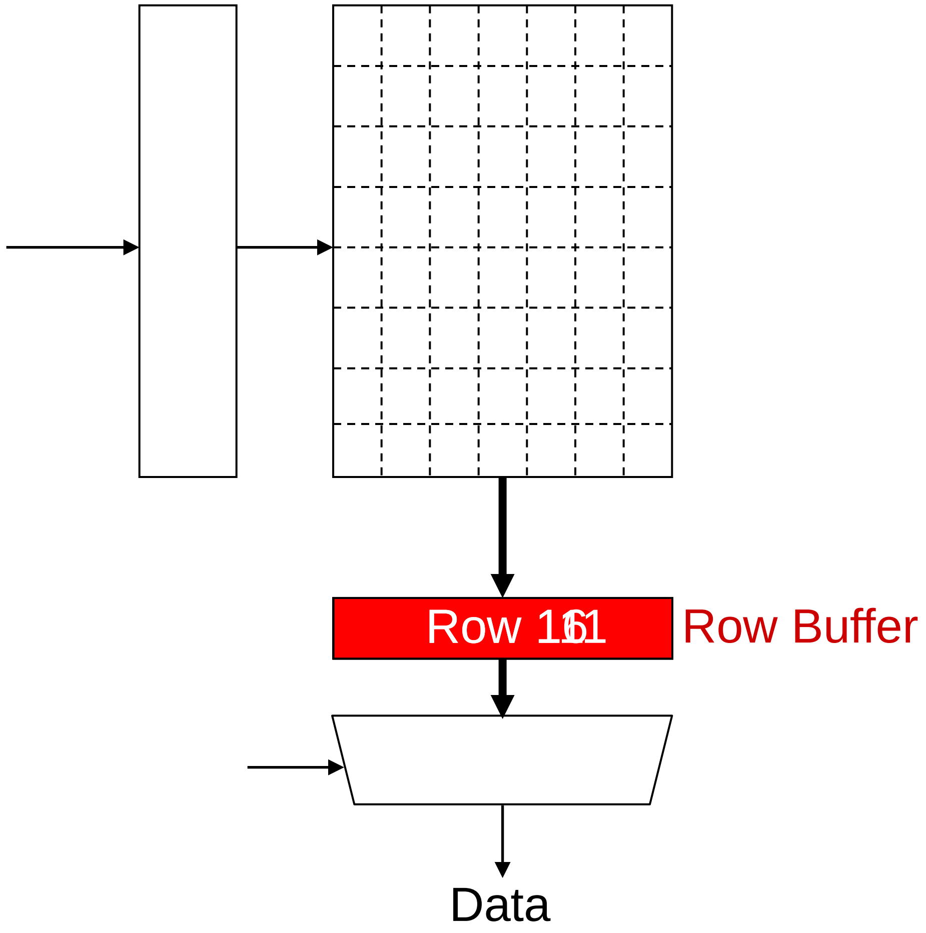
- Each cycle, the DRAM controller
  - Computes Slowdown = ST_{shared}/ST_{alone} for threads with legal requests
  - Computes unfairness = MAX Slowdown / MIN Slowdown

- If unfairness $< \alpha$
  - Use DRAM throughput oriented scheduling policy
- If unfairness $\geq \alpha$
  - Use fairness-oriented scheduling policy
    - (1) requests from thread with MAX Slowdown first
    - (2) row-hit first , (3) oldest-first

How Does STFM Prevent Unfairness?

T0: Row 0
T1: Row 5
T0: Row 0
T1: Row 111
T0: Row 0
T0: Row 0 / T1: Row 16

T0 Slowdown	1.04
T1 Slowdown	1.06
Unfairness	1.03
α	1.05



STFM Pros and Cons

- Upsides:
  - First work on fair multi-core memory scheduling
  - Good at providing fairness
  - Being fair improves 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)

PAR-BS ISCA 2008 Talk

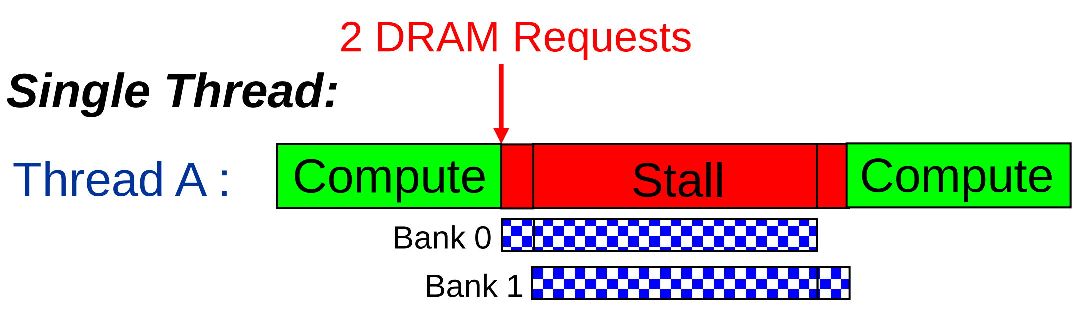
Another Problem due to Interference

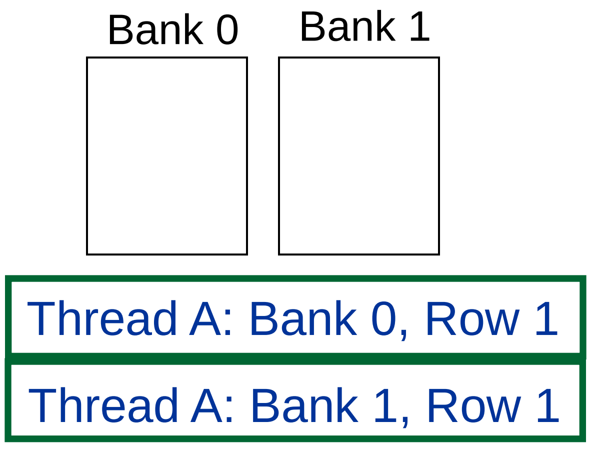
- Processors try to tolerate the latency of DRAM requests by generating multiple outstanding requests
  - Memory-Level Parallelism (MLP)
  - Out-of-order execution, non-blocking caches, runahead execution

- 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

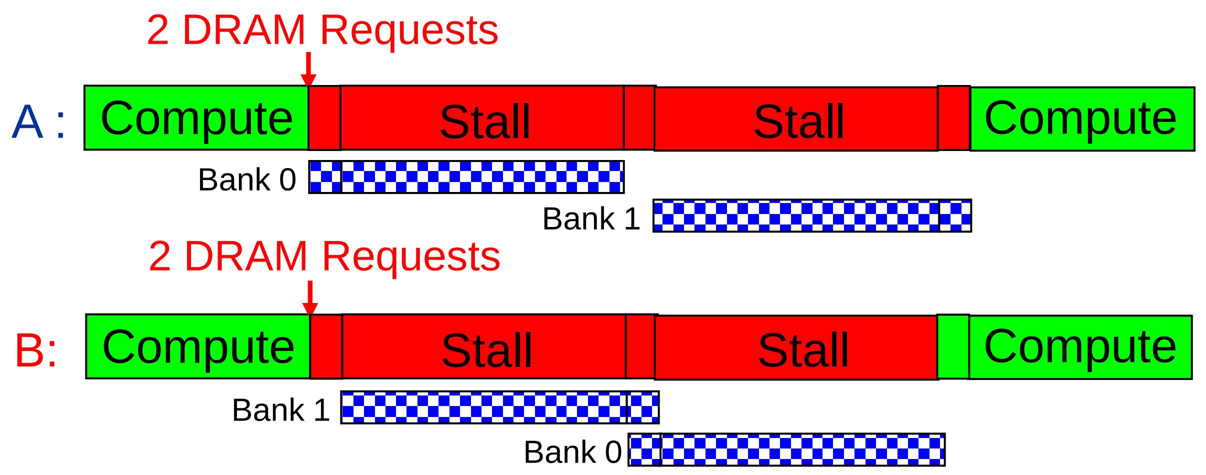


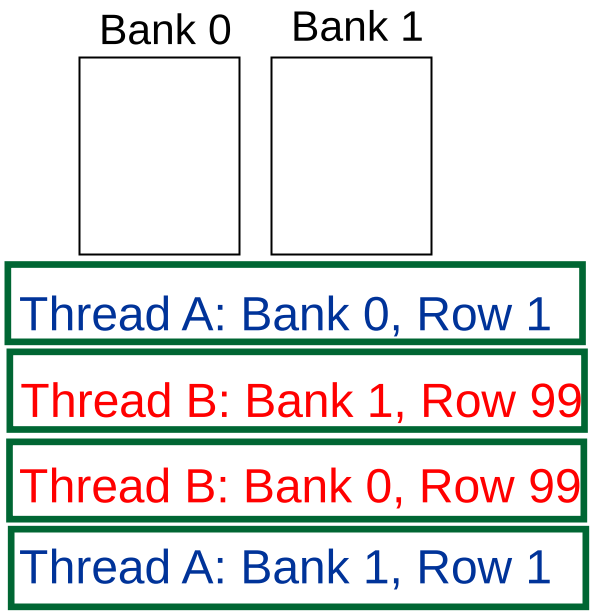


Bank access latencies of the two requests overlapped
Thread stalls for ~ONE bank access latency

Bank Parallelism Interference in DRAM

Baseline Scheduler:

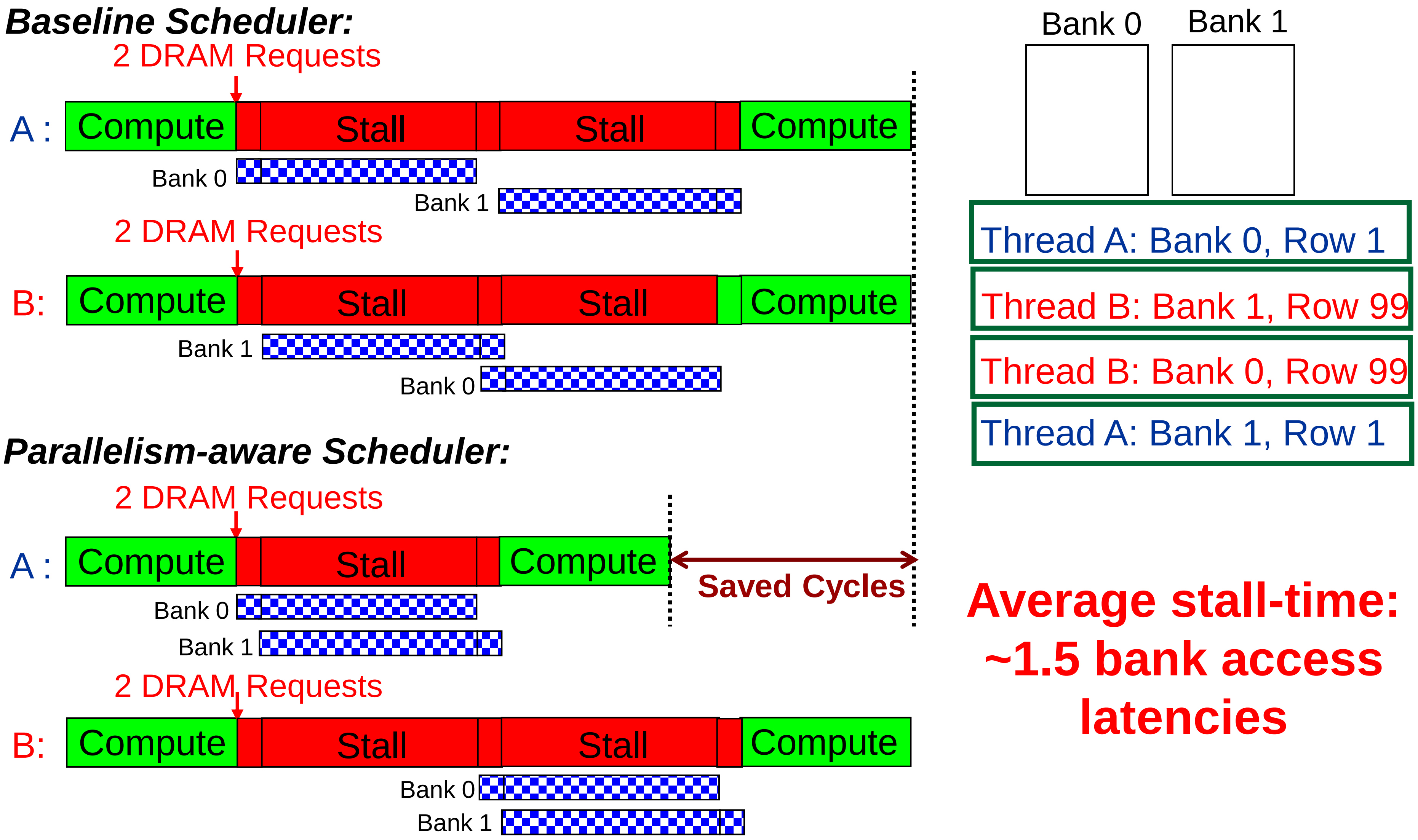




Bank access latencies of each thread serialized
Each thread stalls for ~TWO bank access latencies

SAFARI

Parallelism-Aware Scheduler

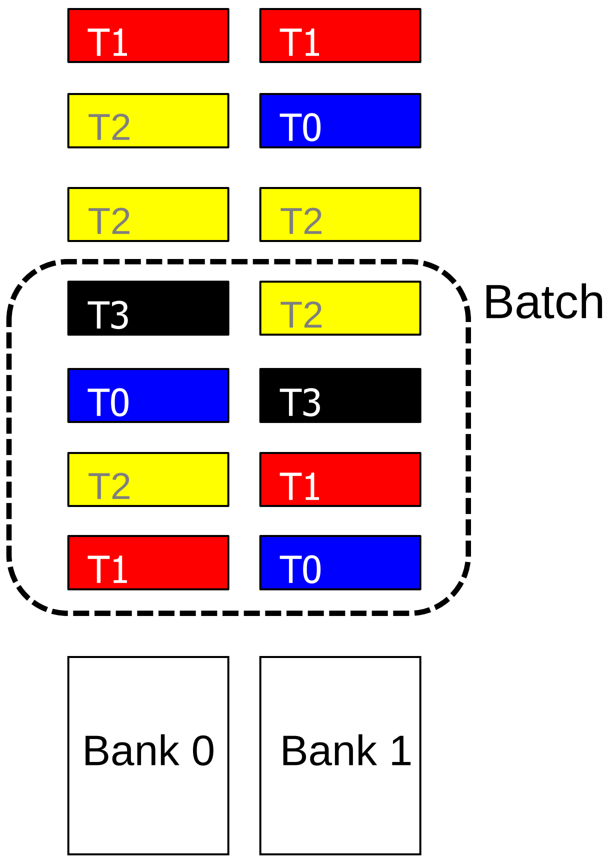


Average stall-time: ~1.5 bank access latencies

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



Mutlu and Moscibroda, "Parallelism-Aware Batch Scheduling," ISCA 2008.

PAR-BS Components

- Request batching

- Within-batch scheduling
  - Parallelism aware

Request Batching

- Each memory request has a bit (*marked)* associated with it
- Batch formation:
  - Mark up to *Marking-Cap* oldest requests per bank for each thread
  - Marked requests constitute the batch
  - Form a new batch when no marked requests are left
- Marked requests are prioritized over unmarked ones
  - No reordering of requests across batches: no starvation, high fairness
- How to prioritize requests within a batch?

Within-Batch Scheduling

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

HOW?

- Scheduler $\boxed{\text{computes a ranking of threads}}$ 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

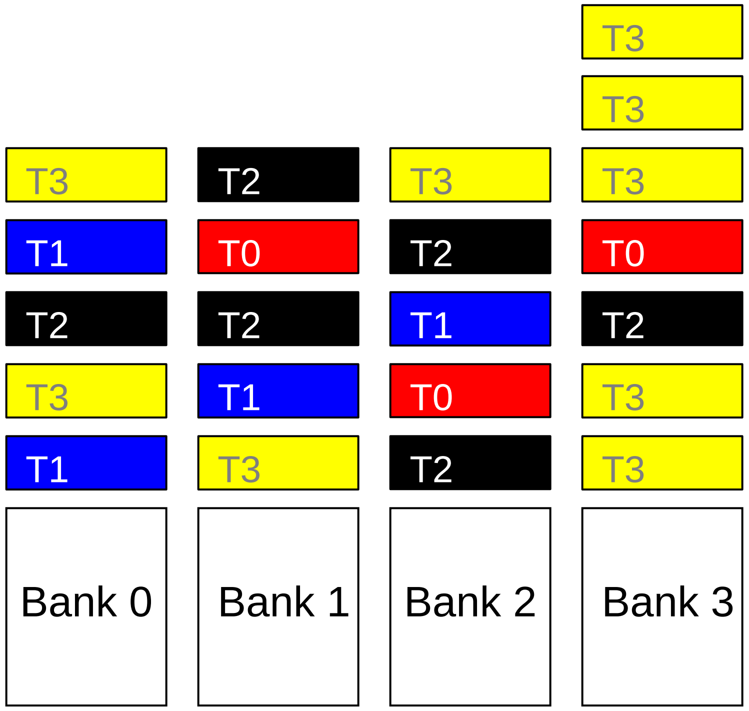
- Maximize system throughput
  - Minimize average stall-time of threads within the batch
- Minimize unfairness (Equalize the slowdown of threads)
  - Service threads with inherently low stall-time early in the batch
  - Insight: delaying memory non-intensive threads results in high slowdown

- 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 (~ low stall-time)
- Total number of marked requests (total-load)
  - Breaks ties: rank thread with lower total-load higher

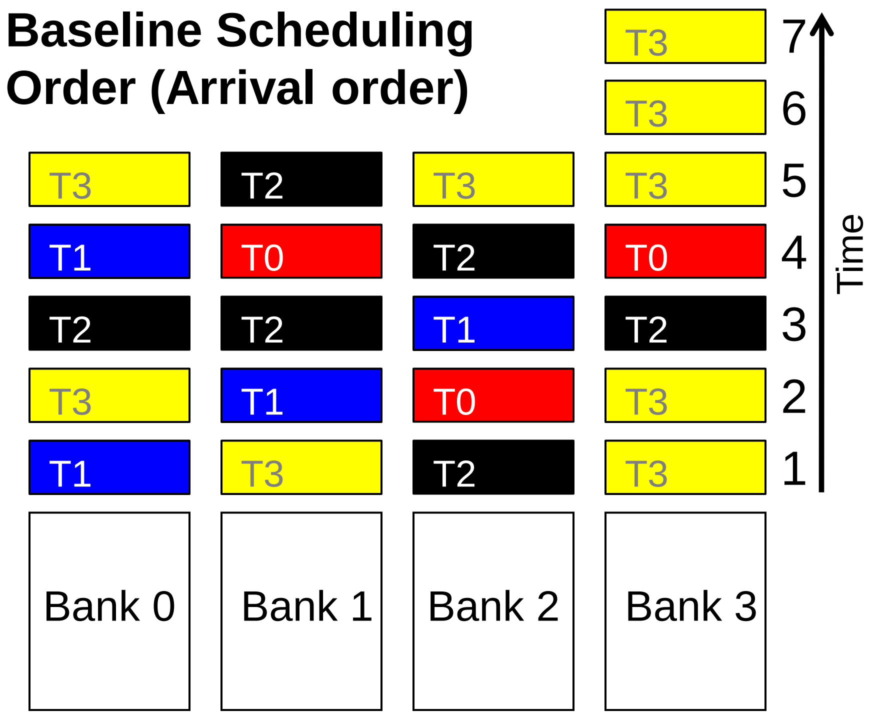


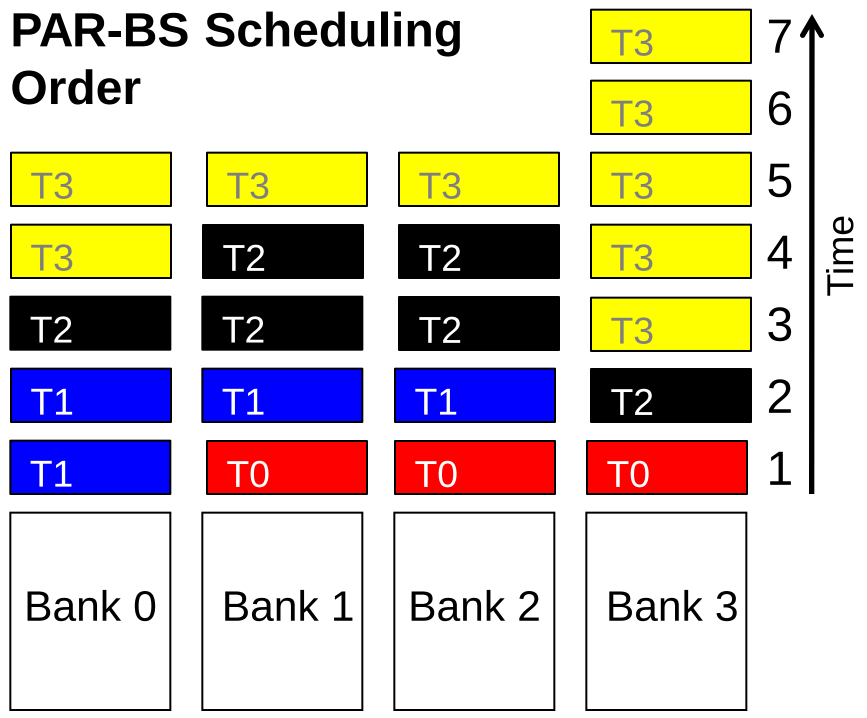
	max-bank-load	total-load

Ranking:
T0 > T1 > T2 > T3

SAFARI

Example Within-Batch 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

SAFARI

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)
  (4) Oldest first — Parallelism-aware within-batch scheduling

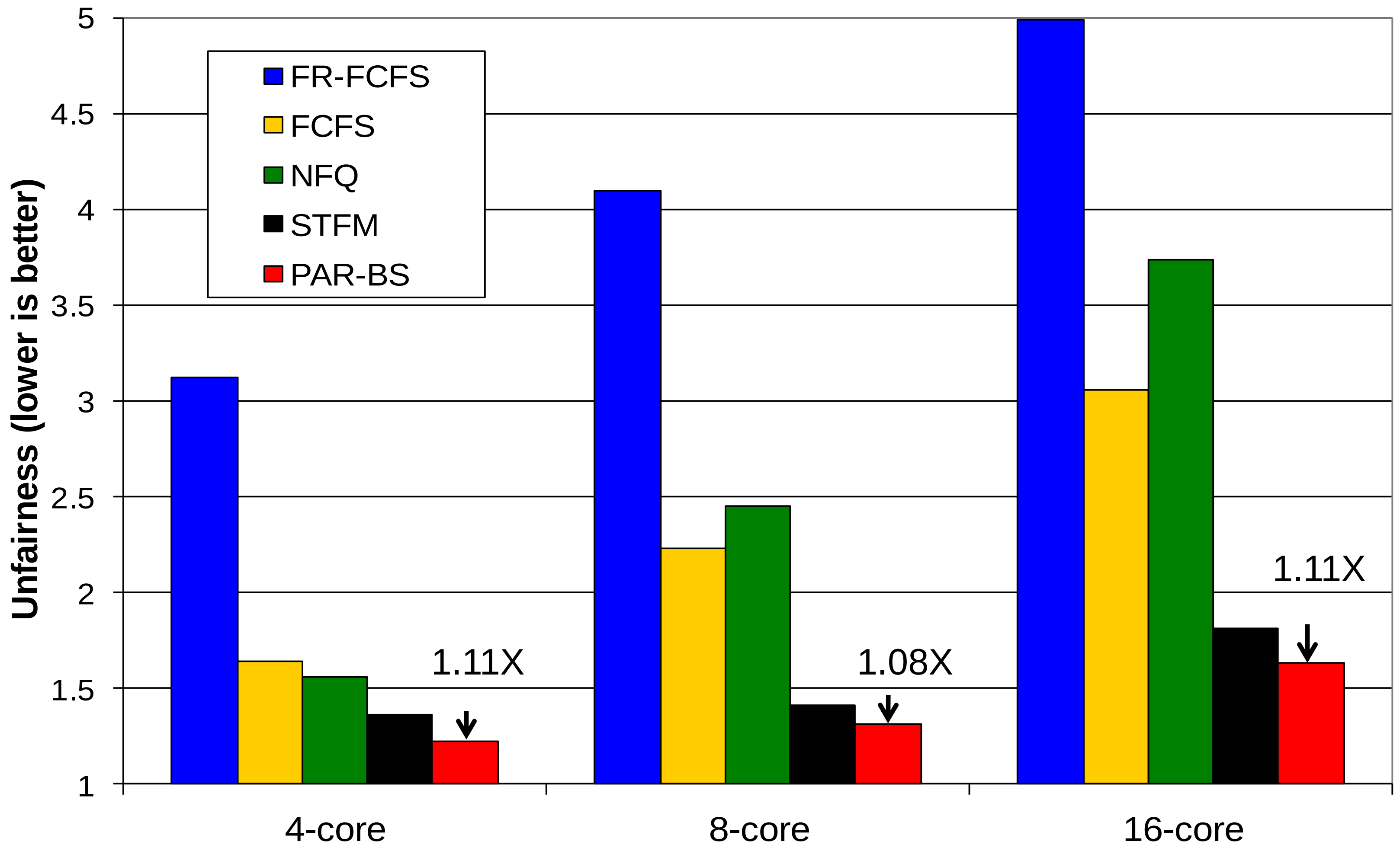
- Three properties:
  - Exploits row-buffer locality **and** intra-thread bank parallelism
  - Work-conserving
    - Services unmarked requests to banks without marked requests
  - Marking-Cap is important
    - Too small cap: destroys row-buffer locality
    - Too large cap: penalizes memory non-intensive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

Hardware Cost

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

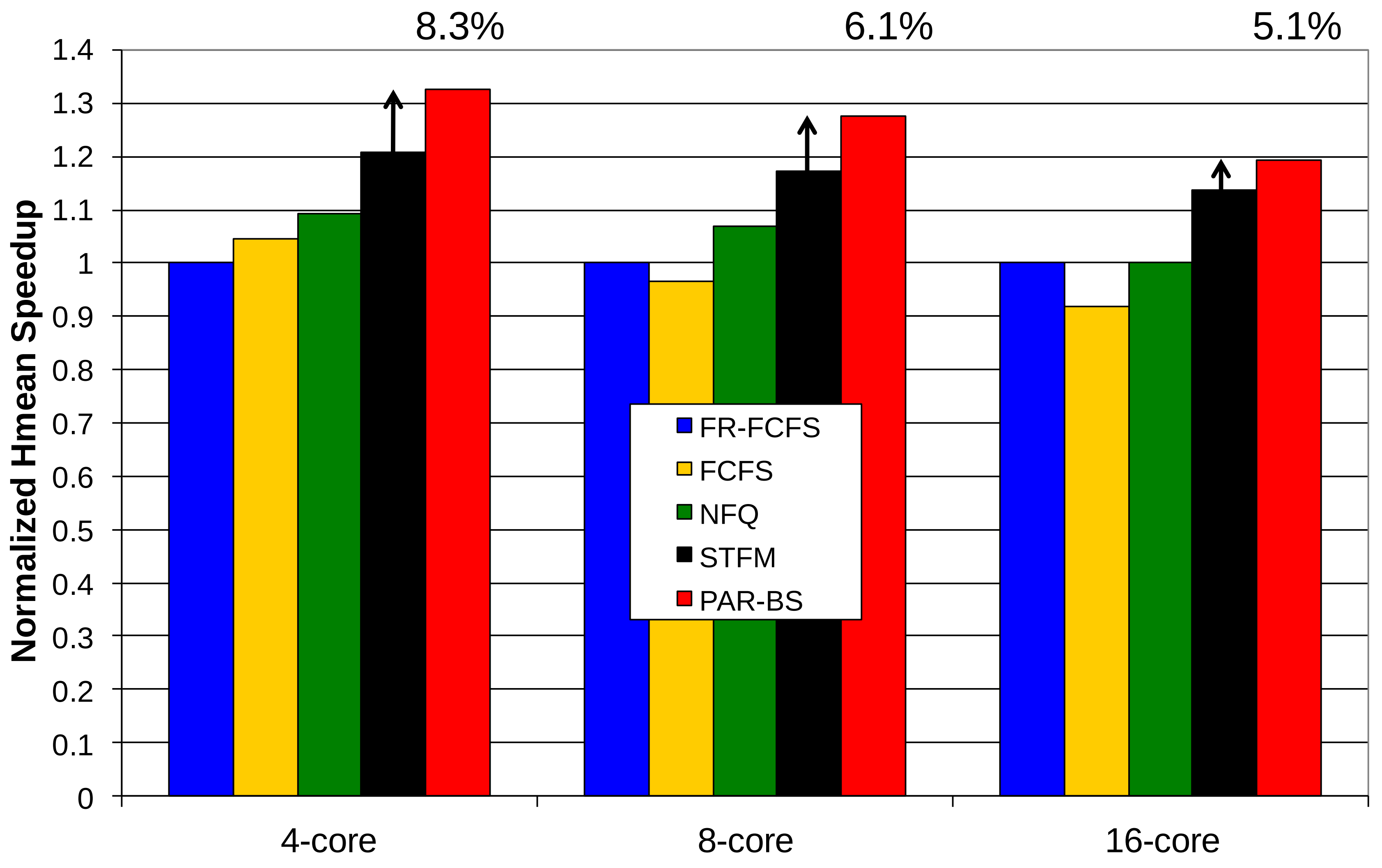
Unfairness on 4-, 8-, 16-core Systems

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



SAFARI

System Performance (Hmean-speedup)



SAFARI

PAR-BS Pros and Cons

- Upsides:
  - First work to identify the notion of bank parallelism destruction across multiple threads
  - Simple mechanism

- Downsides:
  - Implementation in multiple controllers needs coordination for best performance → too frequent coordination since batching is done frequently
  - Does not always prioritize the latency-sensitive applications

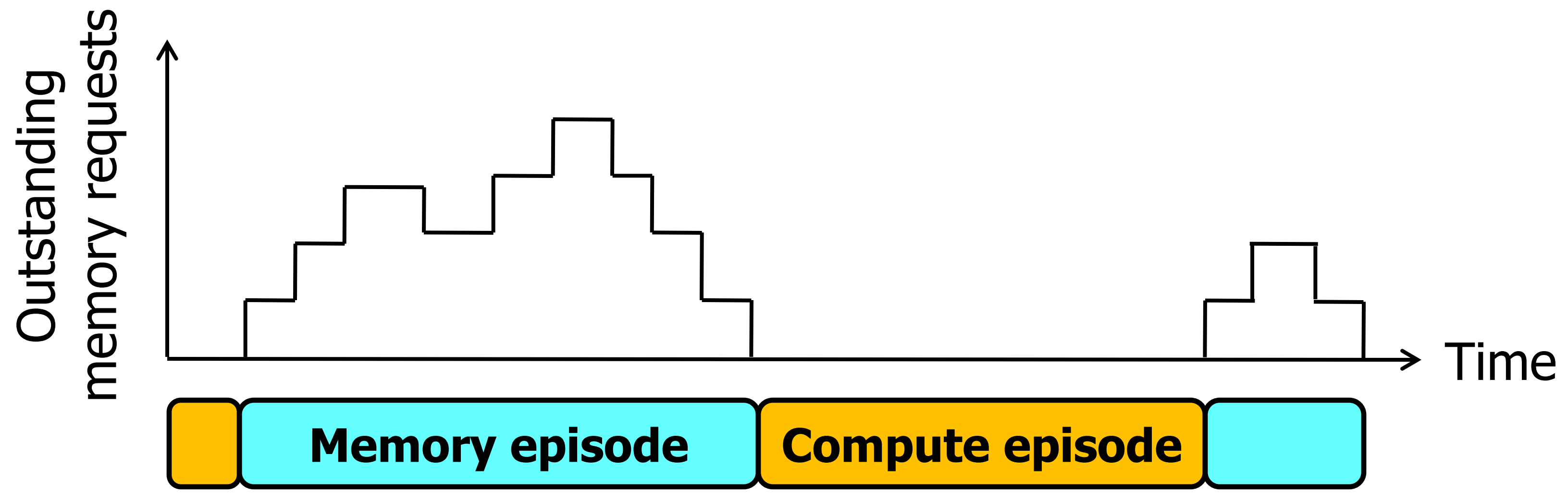
ATLAS Memory Scheduler

Yoongu Kim, Dongsu Han, Onur Mutlu, and Mor Harchol-Balter,
"ATLAS: A Scalable and High-Performance Scheduling Algorithm for Multiple Memory Controllers"
16th International Symposium on High-Performance Computer Architecture (***HPCA***),
Bangalore, India, January 2010. Slides (pptx)

Rethinking Memory Scheduling

A thread alternates between two states (episodes)

- **Compute episode**: Zero outstanding memory requests → **High IPC**
- **Memory episode**: Non-zero outstanding memory requests → **Low IPC**



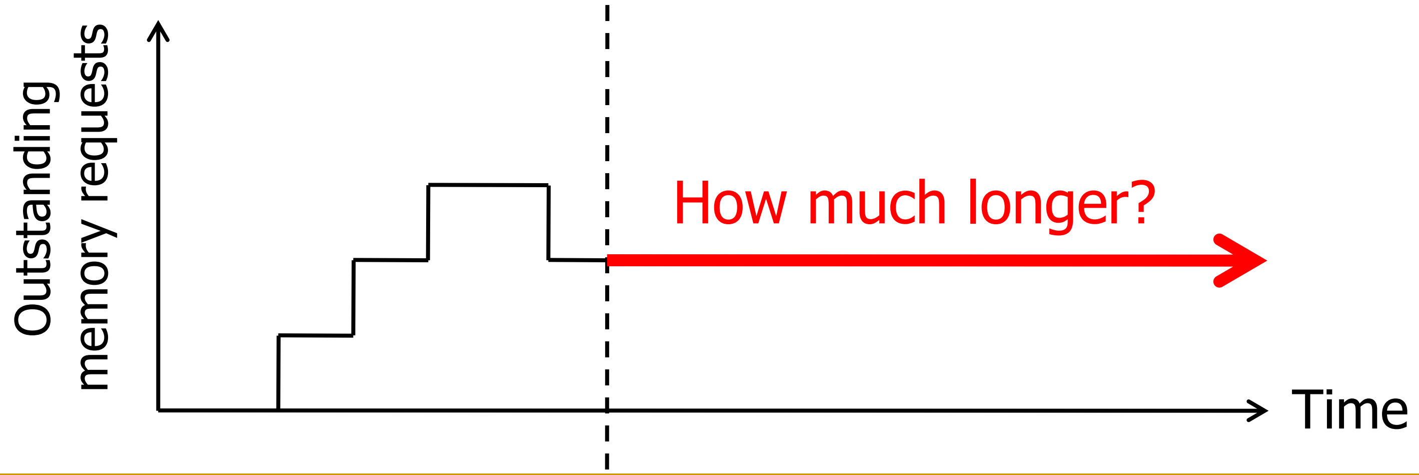
Goal: Minimize time spent in memory episodes

How to Minimize Memory Episode Time

Prioritize thread whose memory episode will end the soonest

- Minimizes time spent in memory episodes across all threads
- Supported by queueing theory:
  - Shortest-Remaining-Processing-Time scheduling is optimal in single-server queue

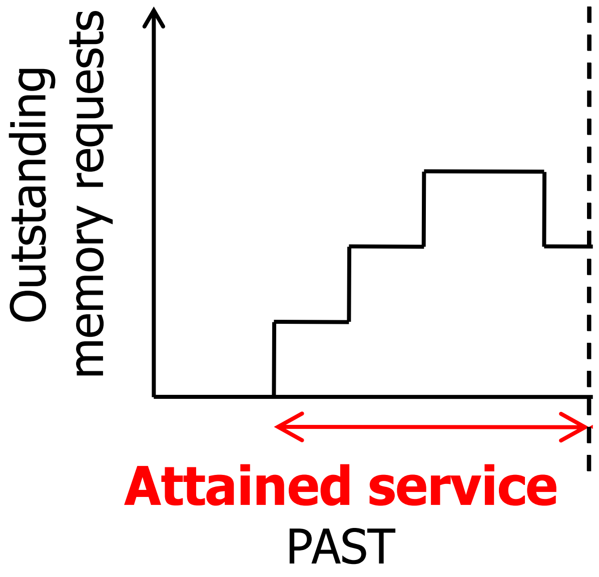
Remaining length of a memory episode?



SAFARI

Predicting Memory Episode Lengths

We discovered: past is excellent predictor for future

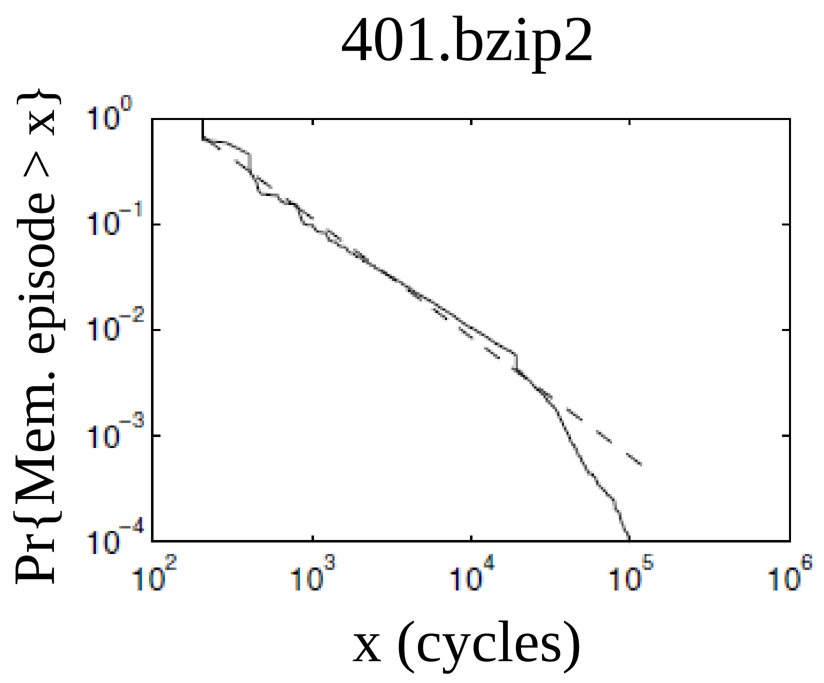


Large **attained service** ➔ Large expected **remaining service**

Q: Why?

A: Memory episode lengths are **Pareto distributed…**

Pareto Distribution of Memory Episode Lengths



Memory episode lengths of SPEC benchmarks

↓

Pareto distribution

↓

The longer an episode has lasted
➔ The longer it will last further

↓

Attained service correlates with remaining service

Favoring **least-attained-service** memory episode
= Favoring memory episode which will **end the soonest**

SAFARI

Least Attained Service (LAS) Memory Scheduling

Our Approach

Prioritize the memory episode with least-**remaining**-service

Queueing Theory

Prioritize the job with shortest-remaining-processing-time

Provably optimal

- Remaining service: Correlates with attained service
- Attained service: Tracked by per-thread counter

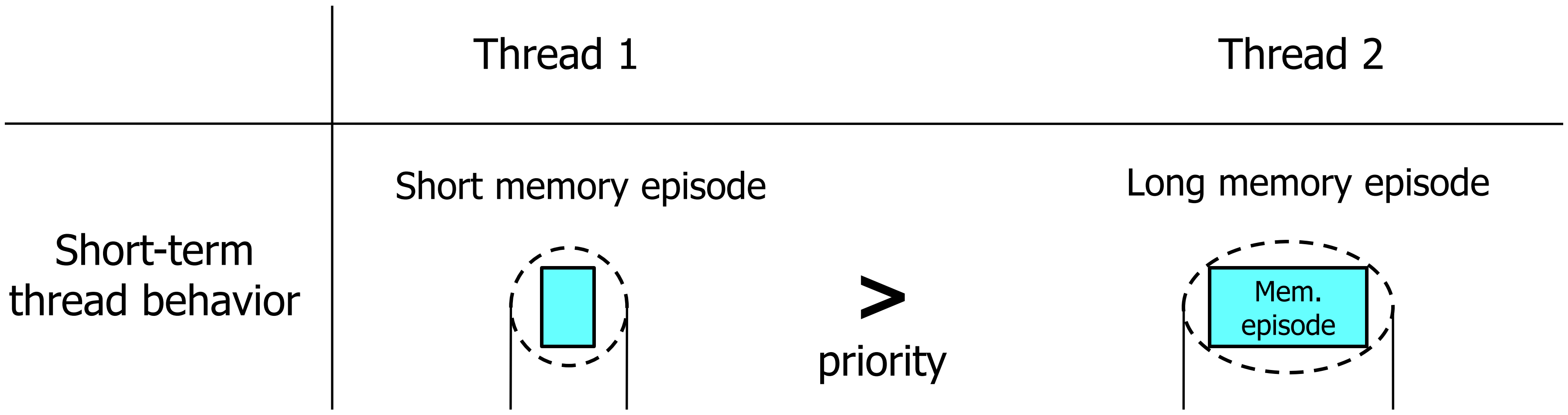
Prioritize the memory episode with least-**attained**-service

Least-attained-service (LAS) scheduling:

Minimize memory episode time

However, LAS does not consider long-term thread behavior

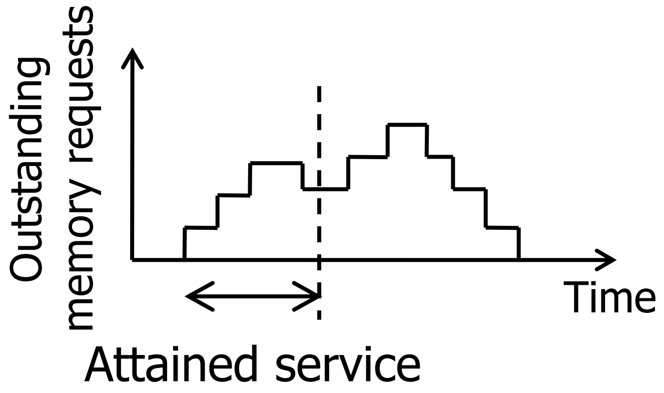
Long-Term Thread Behavior



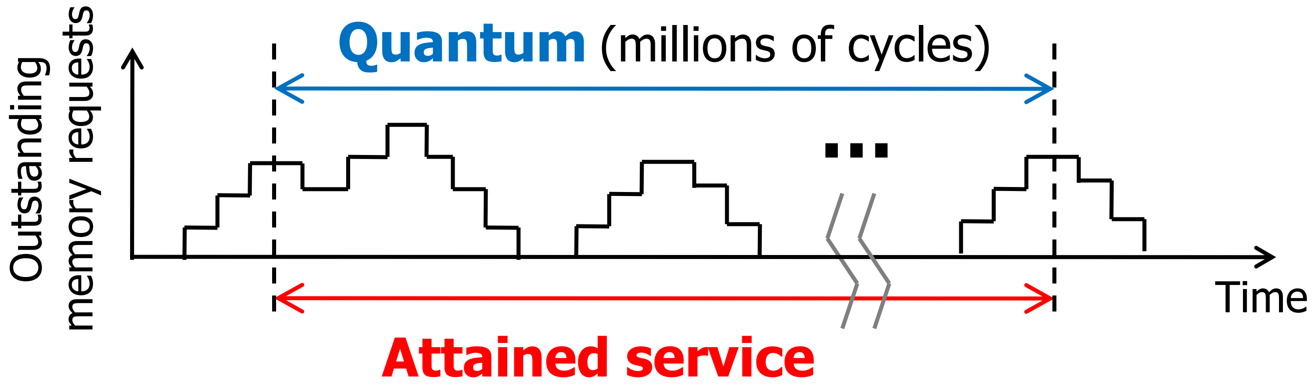
Prioritizing Thread 2 is more beneficial:
results in very long stretches of compute episodes

Quantum-Based Attained Service of a Thread

Short-term thread behavior



Long-term thread behavior



We divide time into large, fixed-length intervals: **quanta** (millions of cycles)

LAS Thread Ranking

During a quantum

Each thread's attained service (AS) is tracked by MCs

AS_i = A thread's AS during only the i-th quantum

End of a quantum

Each thread's **TotalAS** computed as:

$$TotalAS_i = \alpha \cdot TotalAS_{i-1} + (1-\alpha) \cdot AS_i$$

High α ➔ *More bias towards history*

Threads are ranked, favoring threads with lower TotalAS

Next quantum

Threads are serviced according to their ranking

ATLAS Scheduling Algorithm

ATLAS

- **A**daptive per-**T**hread **L**east **A**ttained **S**ervice

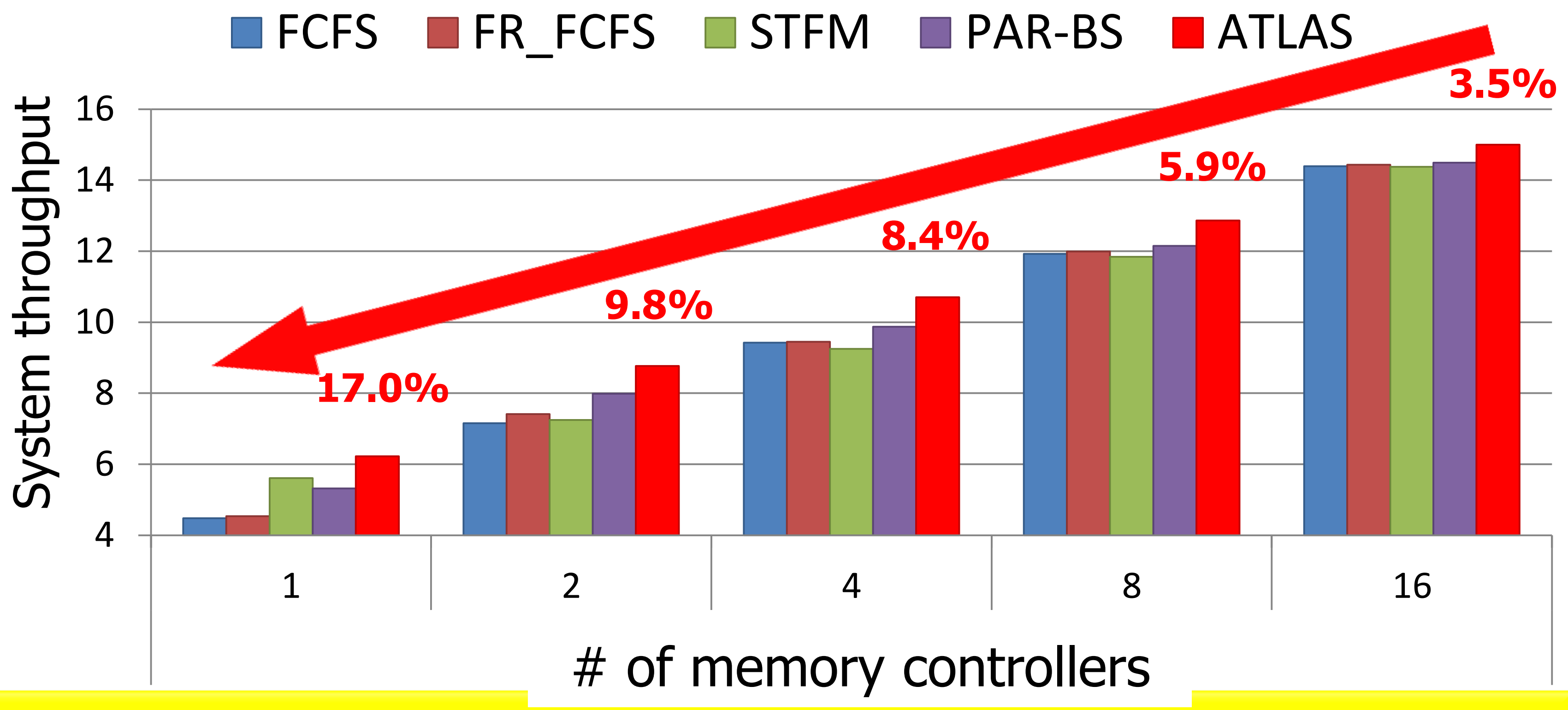
- Request prioritization order

1. **Prevent starvation**: Over threshold request
2. **Maximize performance**: Higher LAS rank
3. **Exploit locality**: Row-hit request
4. **Tie-breaker**: Oldest request

How to coordinate MCs to agree upon a consistent ranking?

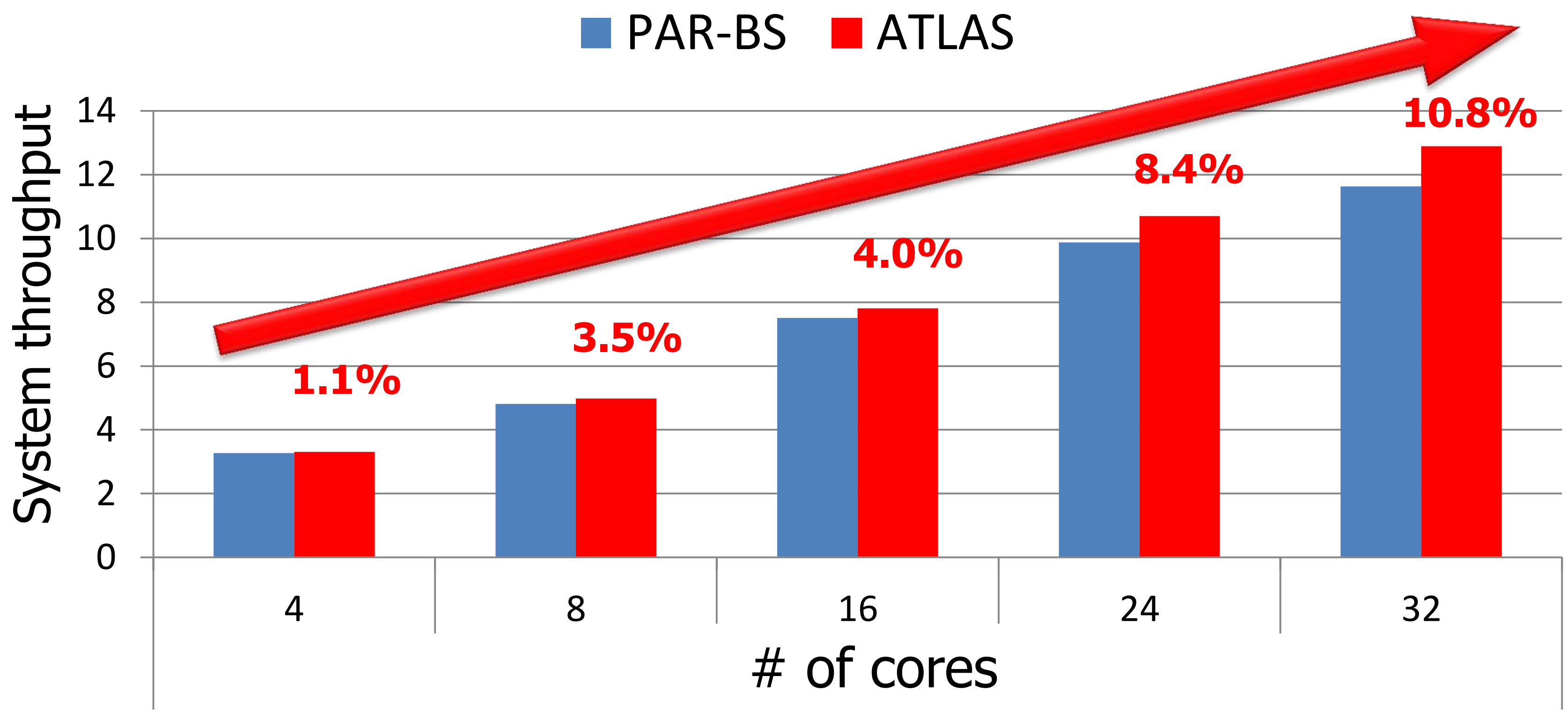
System Throughput: 24-Core System

System throughput = ∑ Speedup



ATLAS consistently provides higher system throughput than all previous scheduling algorithms

System Throughput: 4-MC System



# of cores increases → ATLAS performance benefit increases

Properties of ATLAS

Goals	Properties of ATLAS
▪ Maximize system performance	▪ LAS-ranking ▪ Bank-level parallelism ▪ Row-buffer locality
▪ Scalable to large number of controllers	▪ Very infrequent coordination
▪ Configurable by system software	▪ Scale attained service with thread weight (in paper)
	▪ **Low complexity**: Attained service requires a single counter per thread in each MC

SAFARI

ATLAS Pros and Cons

- Upsides:
  - Good at improving performance
  - Low complexity
  - Coordination among controllers happens infrequently

- Downsides:
  - Lowest ranked threads get delayed significantly → high unfairness