Computer Architecture: Memory Interference and QoS (Part I)

Prof. Onur Mutlu

Carnegie Mellon University

Memory Interference and QoS Lectures

- These slides are from a lecture delivered at INRIA (July 8, 2013)

- Similar slides were used at the ACACES 2013 course
    - http://users.ece.cmu.edu/~omutlu/acaces2013-memory.html

QoS-Aware Memory Systems

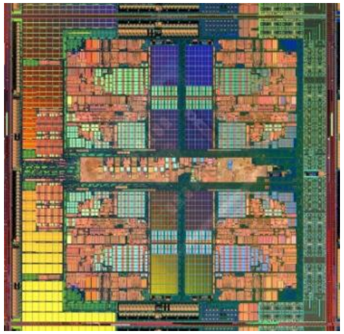
Onur Mutlu
onur@cmu.edu
July 8, 2013
INRIA

Carnegie Mellon

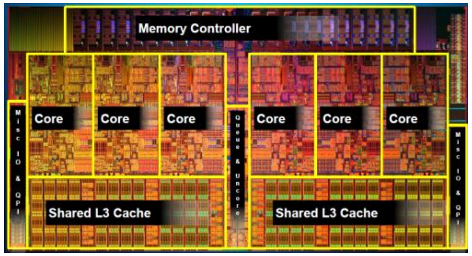
SAFARI

Trend: Many Cores on Chip

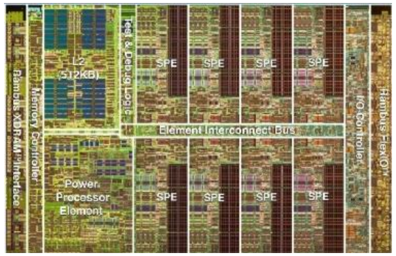
- Simpler and lower power than a single large core
- Large scale parallelism on chip



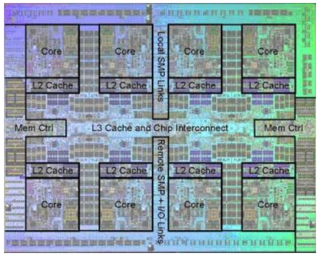
AMD Barcelona
4 cores



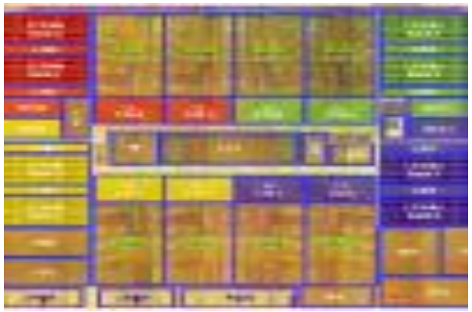
Intel Core i7
8 cores



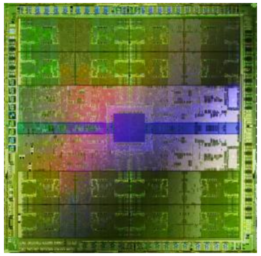
IBM Cell BE
8+1 cores



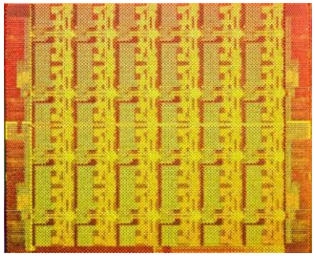
IBM POWER7
8 cores



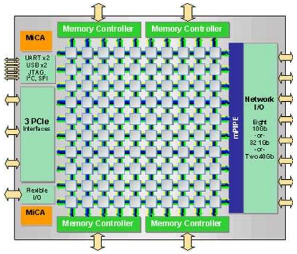
Sun Niagara II
8 cores



Nvidia Fermi
448 “cores”



Intel SCC
48 cores, networked



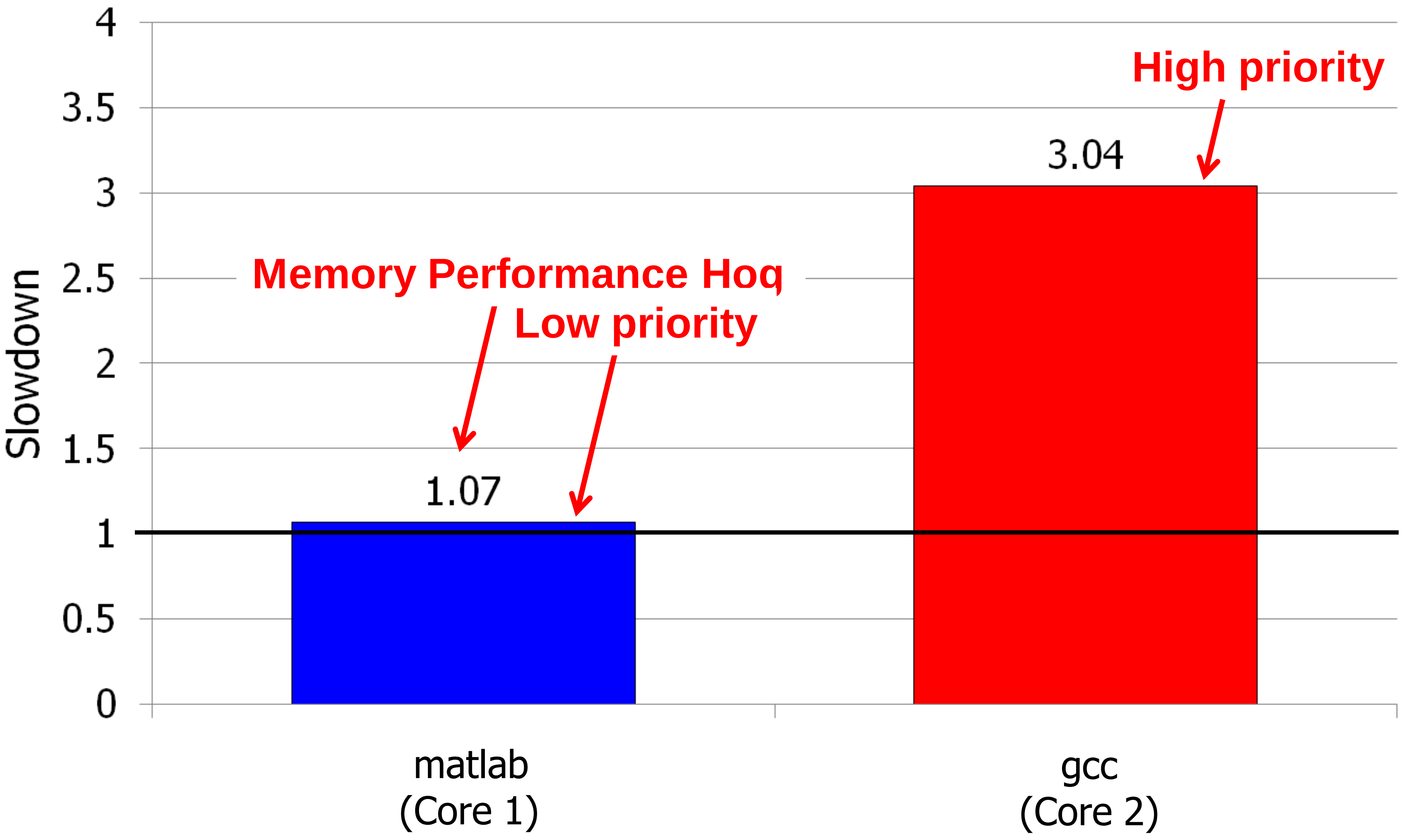
Tilera TILE Gx
100 cores, networked

Many Cores on Chip

- What we want:
    - N times the system performance with N times the cores

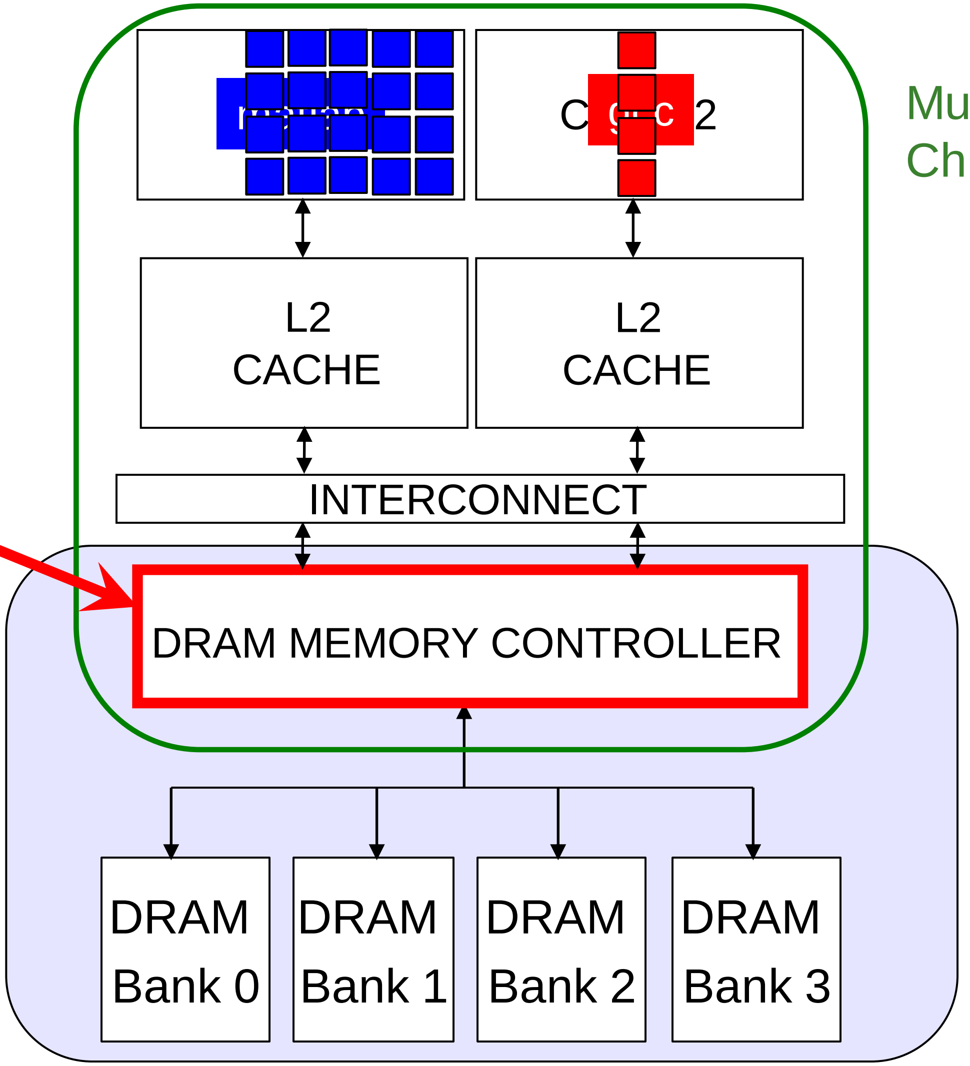
- What do we get today?

Unfair Slowdowns due to Interference



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

Uncontrolled Interference: An Example

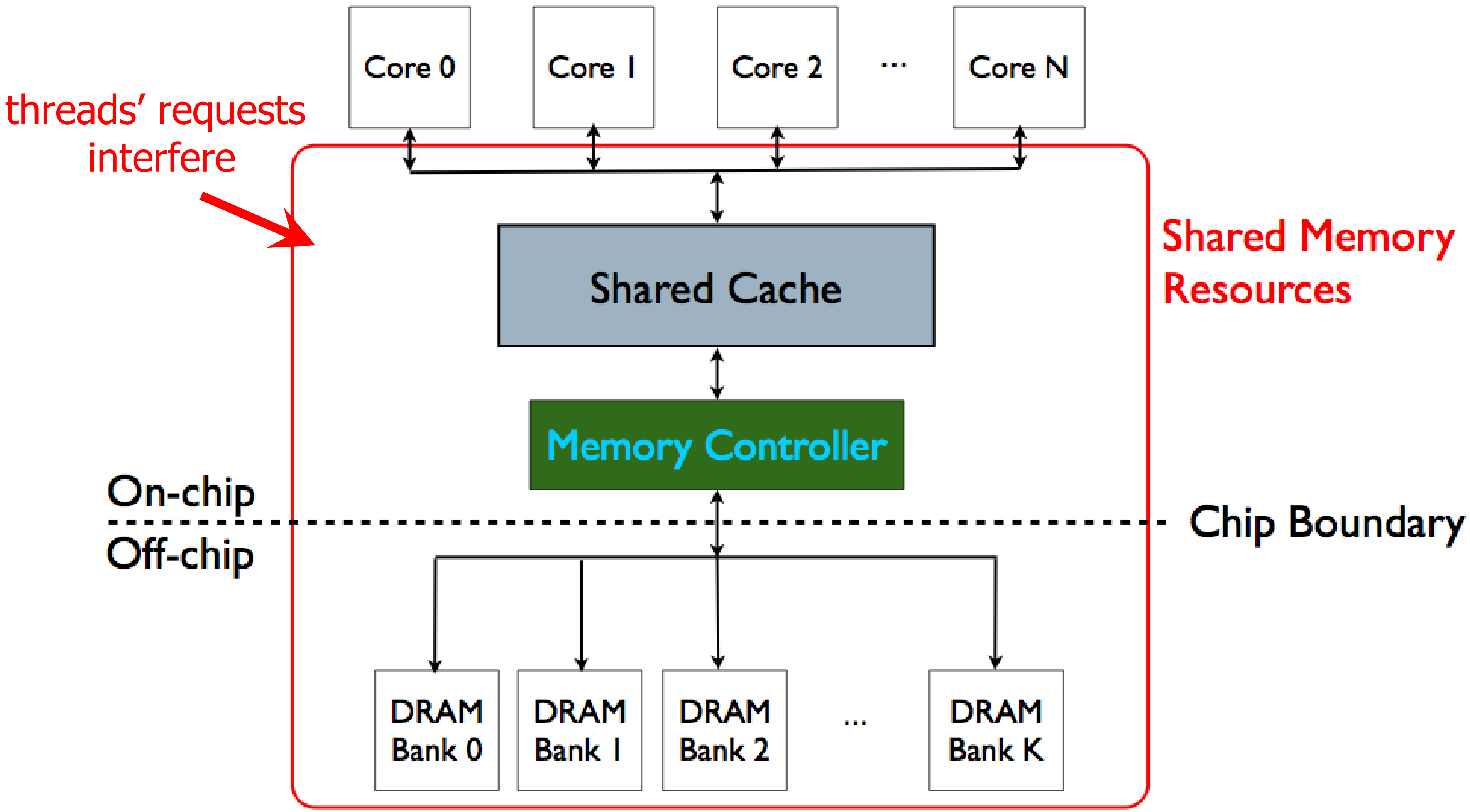


Multi-Core Chip

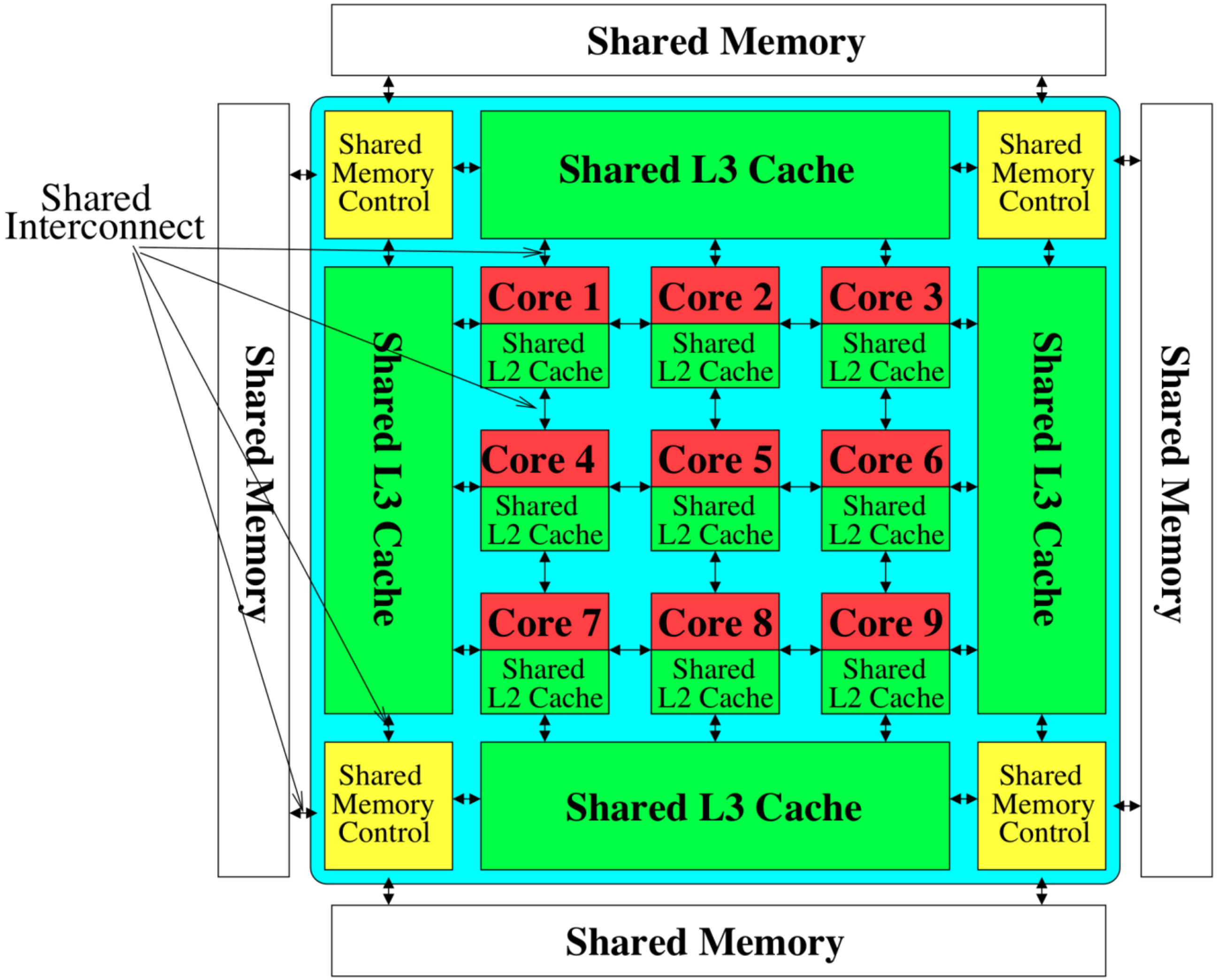
unfairness

Shared DRAM Memory System

Memory System is the Major Shared Resource



Much More of a Shared Resource in Future

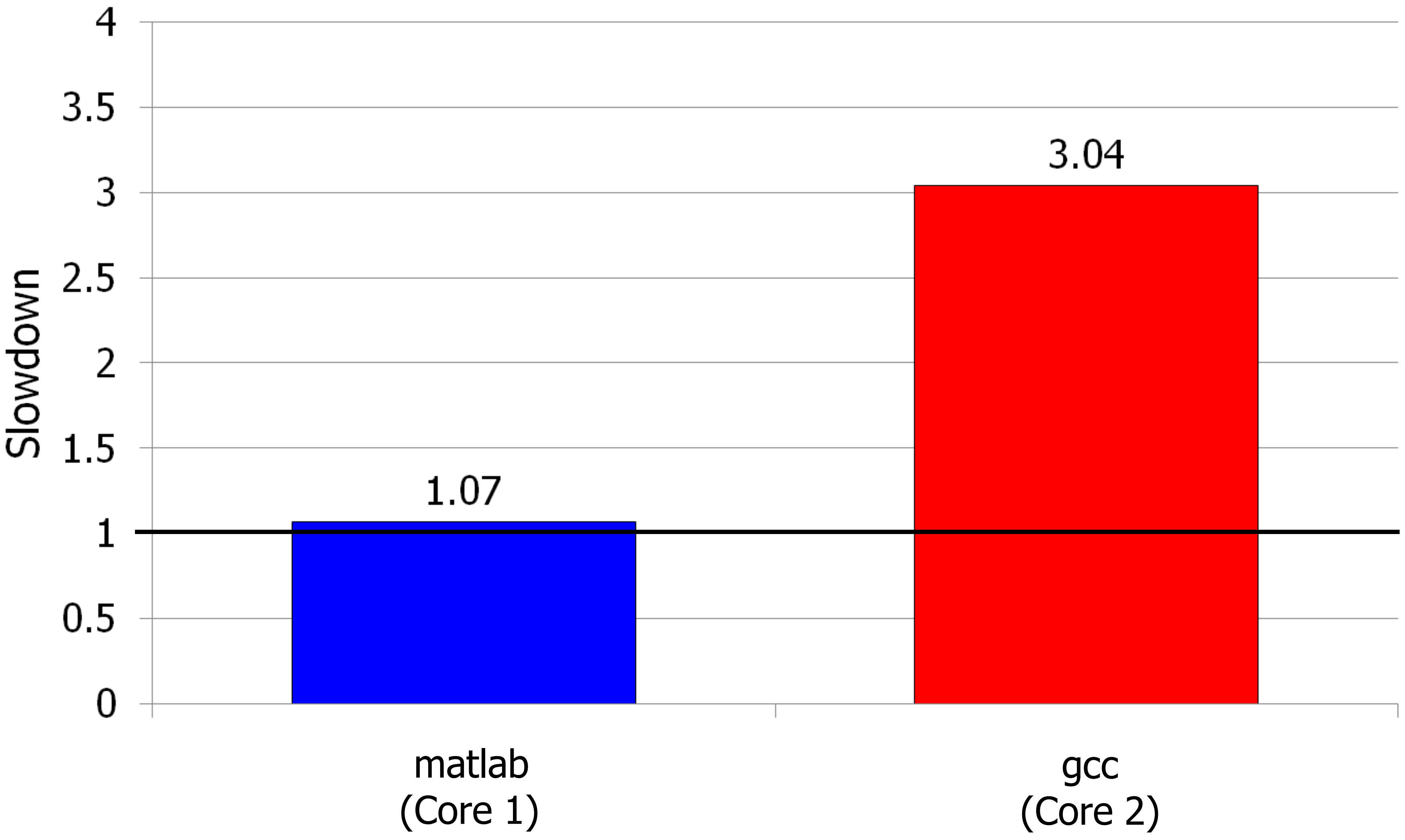


Inter-Thread/Application Interference

- Problem: Threads share the memory system, but memory system does not distinguish between threads' requests

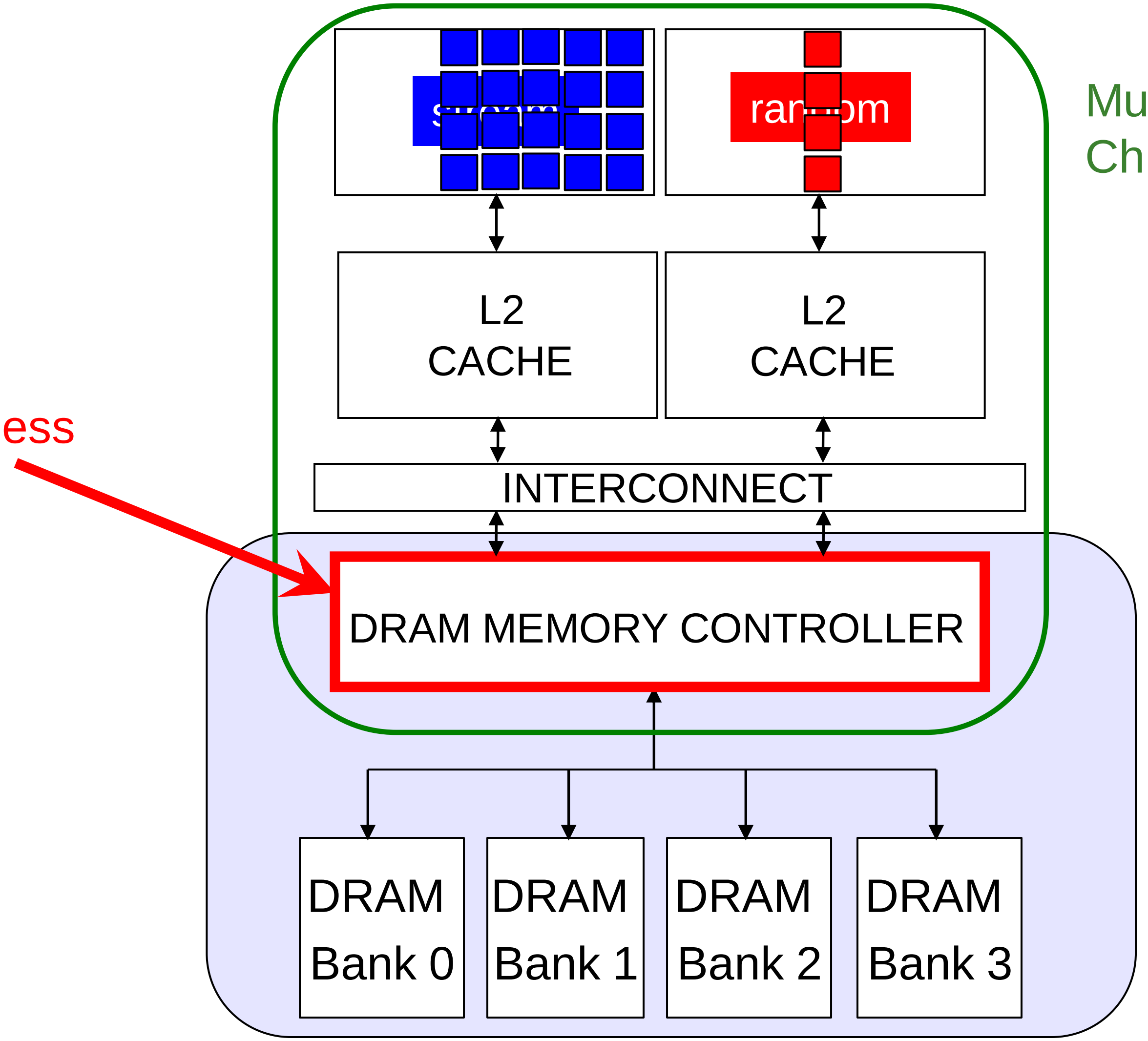
- Existing memory systems
  - Free-for-all, shared based on demand
  - Control algorithms thread-unaware and thread-unfair
  - Aggressive threads can deny service to others
  - Do not try to reduce or control inter-thread interference

Unfair Slowdowns due to Interference



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

Uncontrolled Interference: An Example



unfairness

Multi-Core Chip

Shared DRAM Memory System

A Memory Performance Hog

```
// initialize large arrays A, B

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

streaming

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();
    A[index] = B[index];
    …
}
```

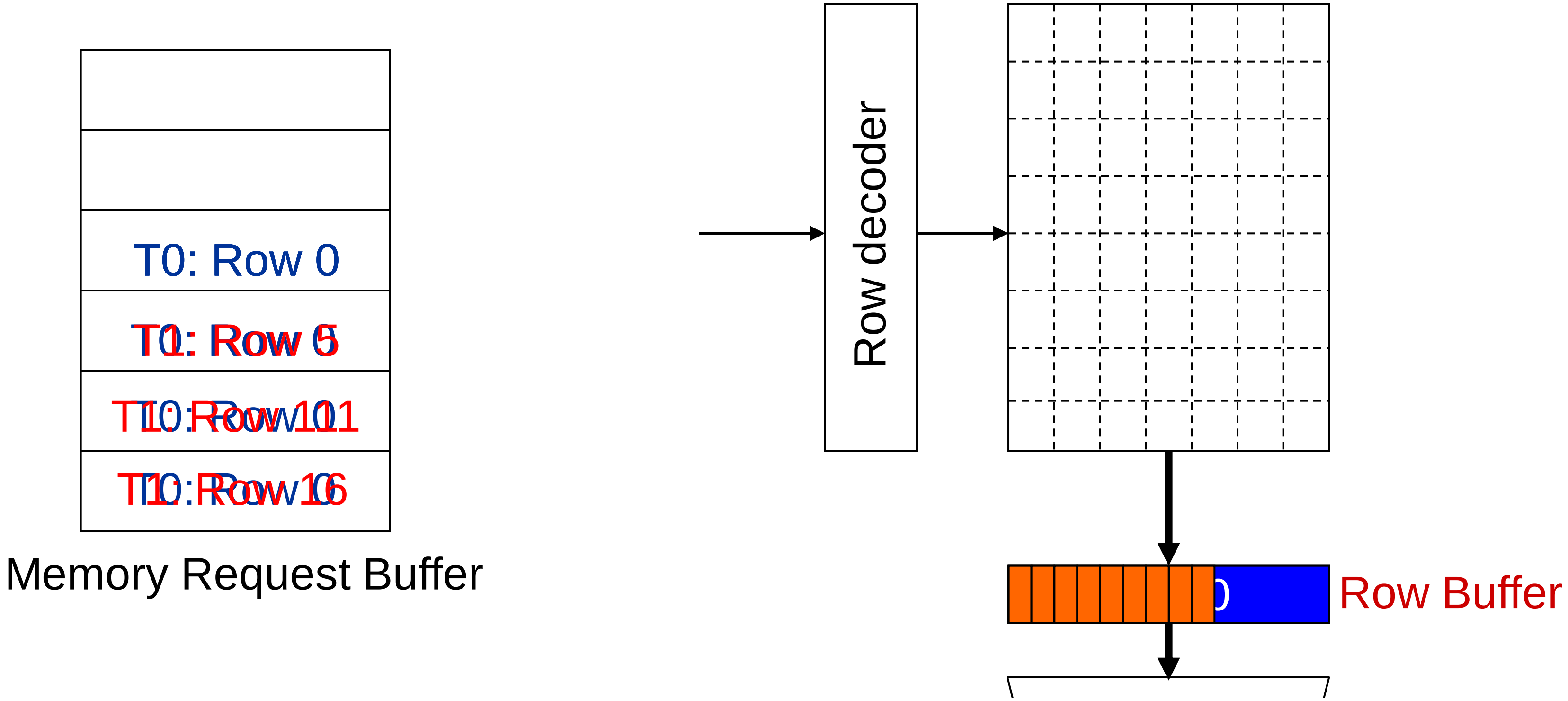
random

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.

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]*

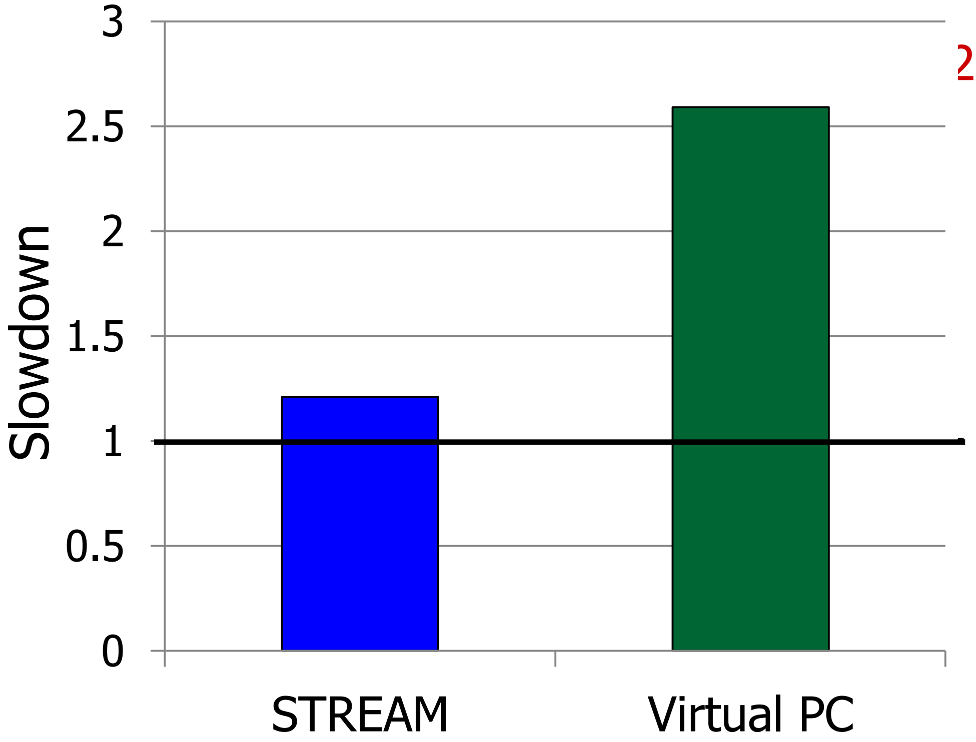
  (1) Row-hit first: Service row-hit memory accesses first

  (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.

Effect of the Memory Performance Hog

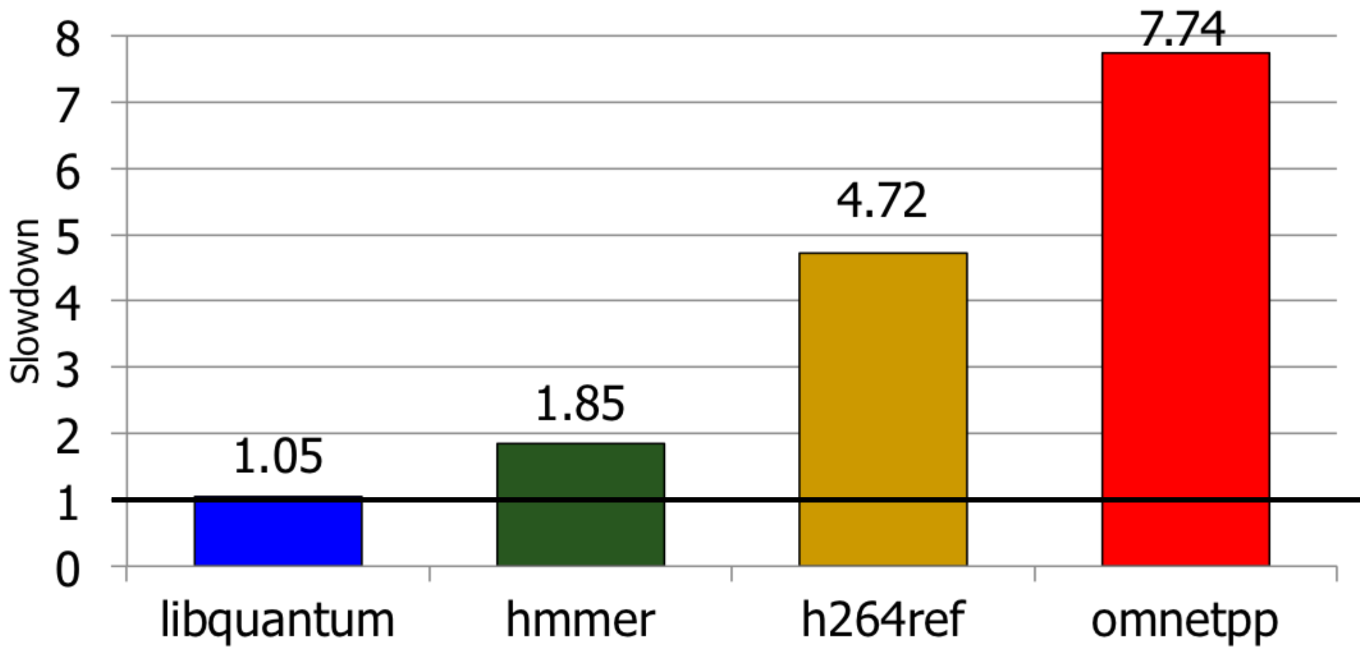


2X slowdown

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.

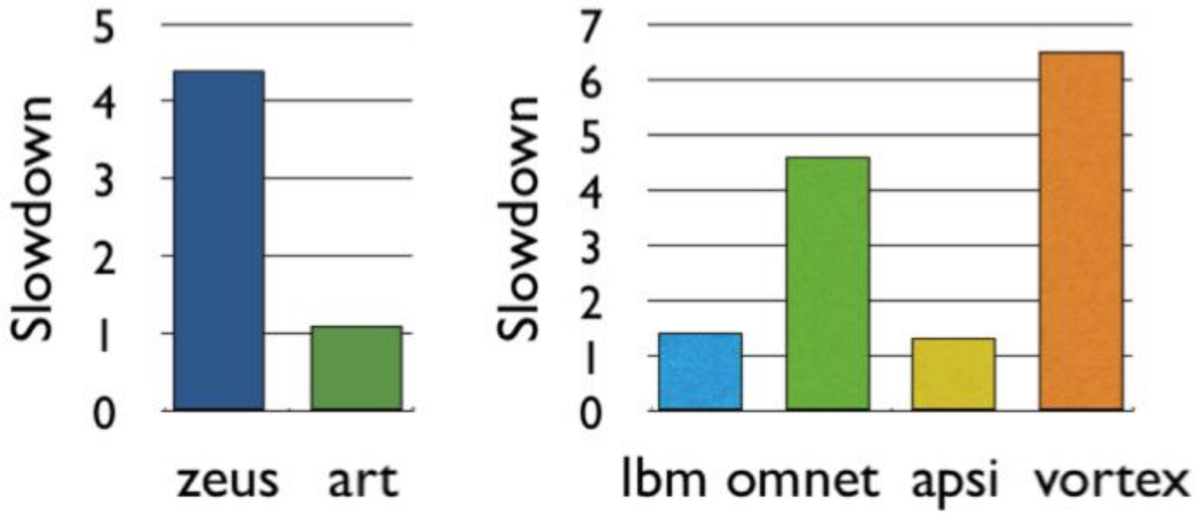
Greater Problem with More Cores



- Vulnerable to denial of service (DoS)
- Unable to enforce priorities or SLAs
- Low system performance

Uncontrollable, unpredictable system

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]

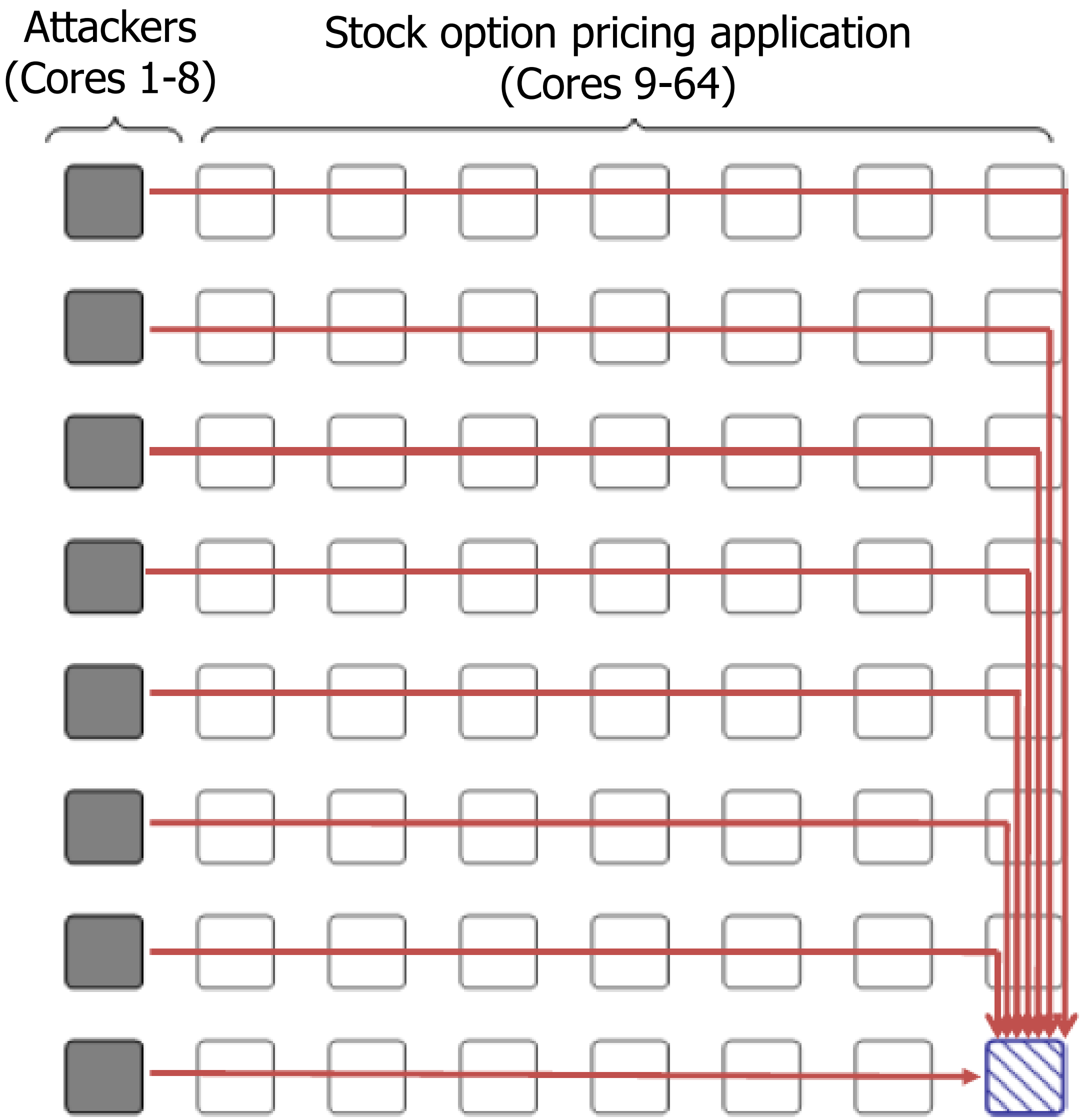
Uncontrollable, unpredictable system

Distributed DoS in Networked Multi-Core Systems

Cores connected via packet-switched routers on chip

~5000X latency increase

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



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

QoS-Aware Memory Systems: Challenges

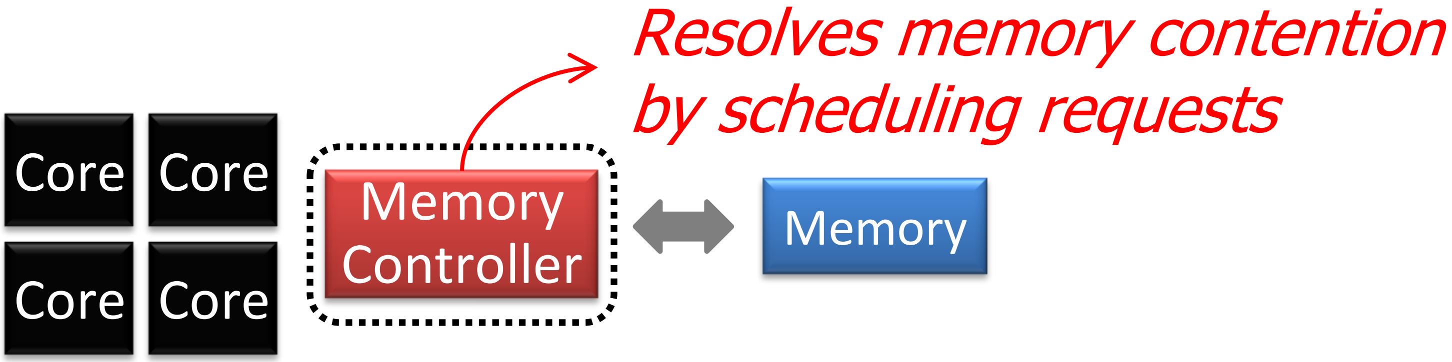
- How do we reduce inter-thread interference?
  - Improve system performance and core utilization
  - Reduce request serialization and core starvation
- How do we control inter-thread interference?
  - Provide mechanisms to enable system software to enforce QoS policies
  - While providing high system 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][Subramanian+, HPCA'13]
  - 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] [Nychis+ SIGCOMM'12]
  - QoS-aware data mapping to memory controllers [Muralidhara+ MICRO'11]
  - QoS-aware thread scheduling to cores [Das+ HPCA'13]

QoS-Aware Memory Scheduling



Resolves memory contention by scheduling requests

- 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

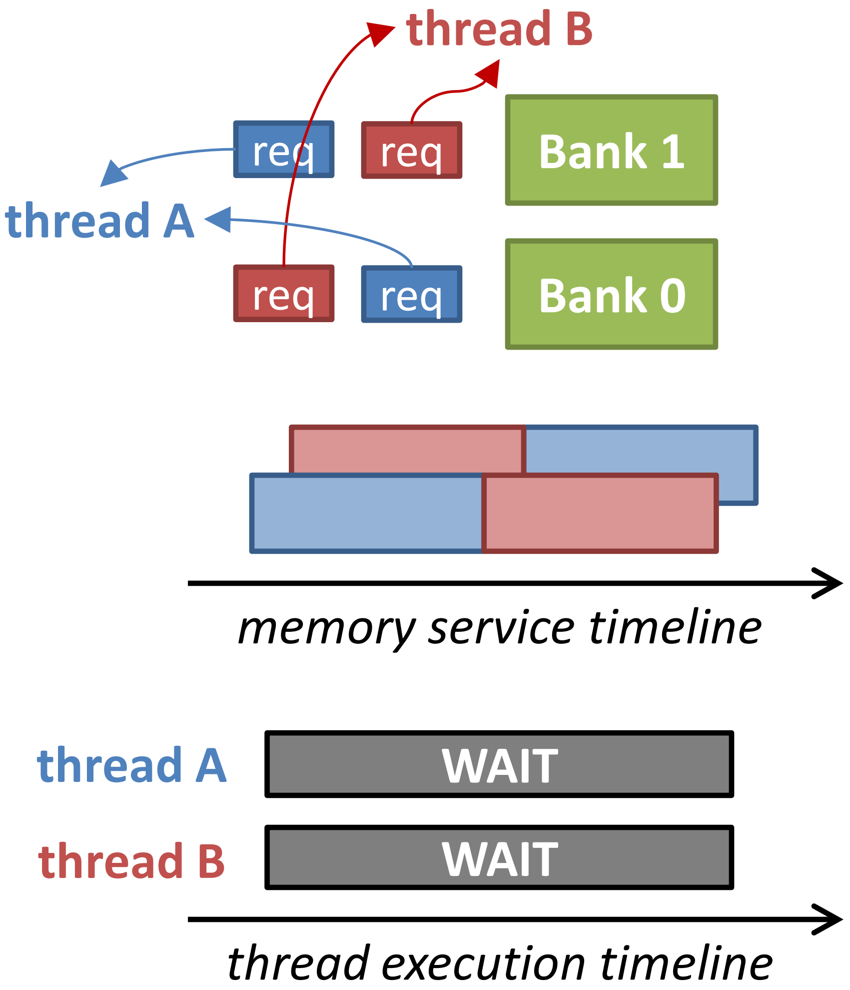
QoS-Aware Memory Scheduling: Evolution

- 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

- ATLAS memory scheduler [Kim+ HPCA'10]

Within-Thread Bank Parallelism



thread B

req

req

Bank 1

thread A

req

req

Bank 0

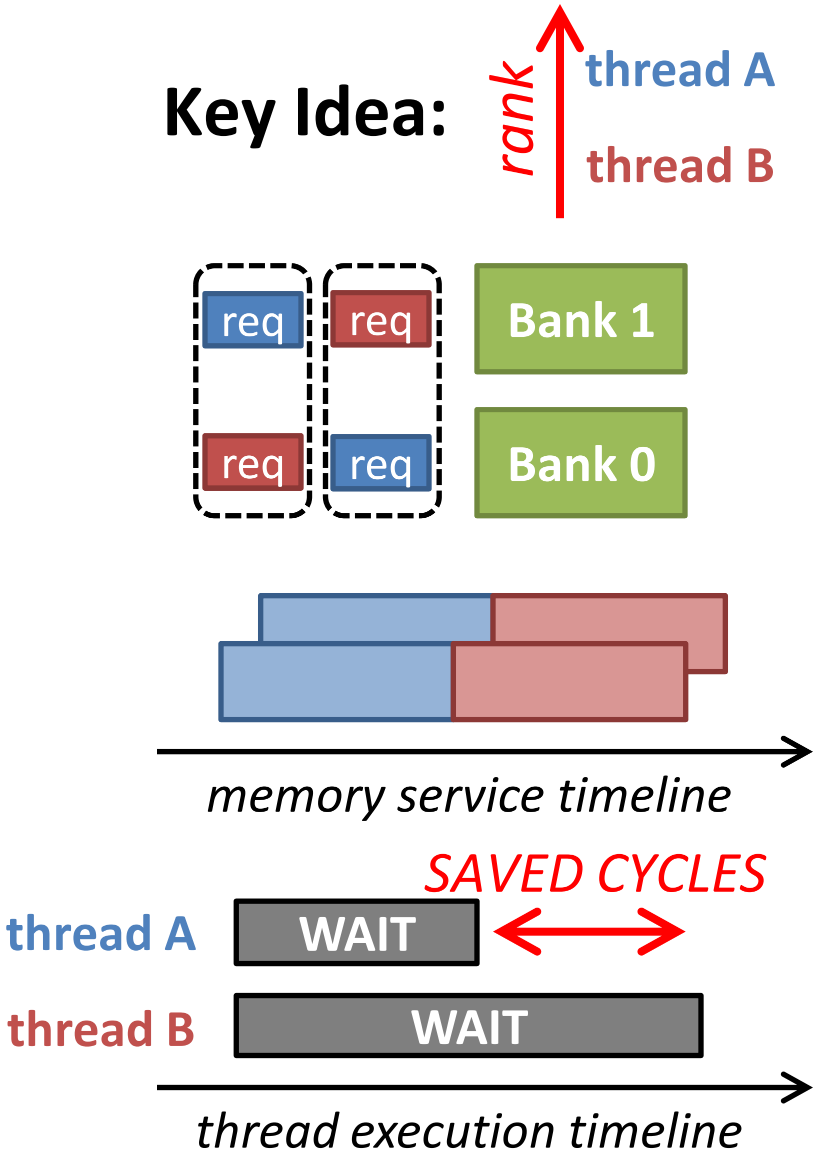
memory service timeline

thread A WAIT

thread B WAIT

thread execution timeline

Key Idea:



rank

thread A

thread B

req

req

Bank 1

req

req

Bank 0

memory service timeline

SAVED CYCLES

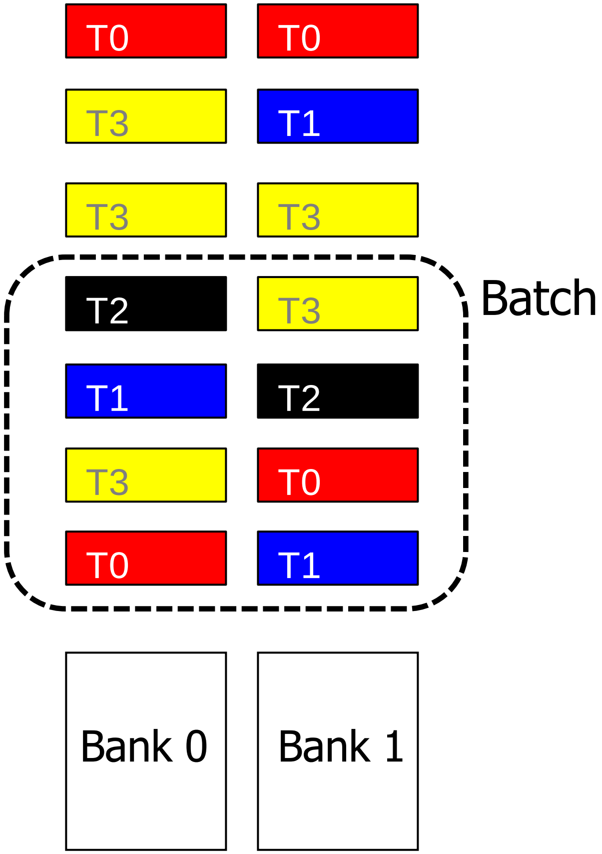
thread A WAIT

thread B WAIT

thread execution timeline

Parallelism-Aware Batch Scheduling [ISCA'08]

- Principle 1: Schedule requests from a thread back to back
  - Preserves each thread's bank parallelism
  - But, this can cause starvation…
- Principle 2: 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 batch is done
  - Eliminates starvation, provides fairness



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 combining 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

- MISE [Subramanian+ HPCA'13]
  - Idea: Estimate the performance of a thread by estimating its change in memory request service rate when run alone vs. shared → use this simple model to estimate slowdown to design a scheduling policy that provides predictable performance or fairness
  - Takeaway: Request service rate of a thread is a good proxy for its performance; alone request service rate can be estimated by giving high priority to the thread in memory scheduling for a while

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

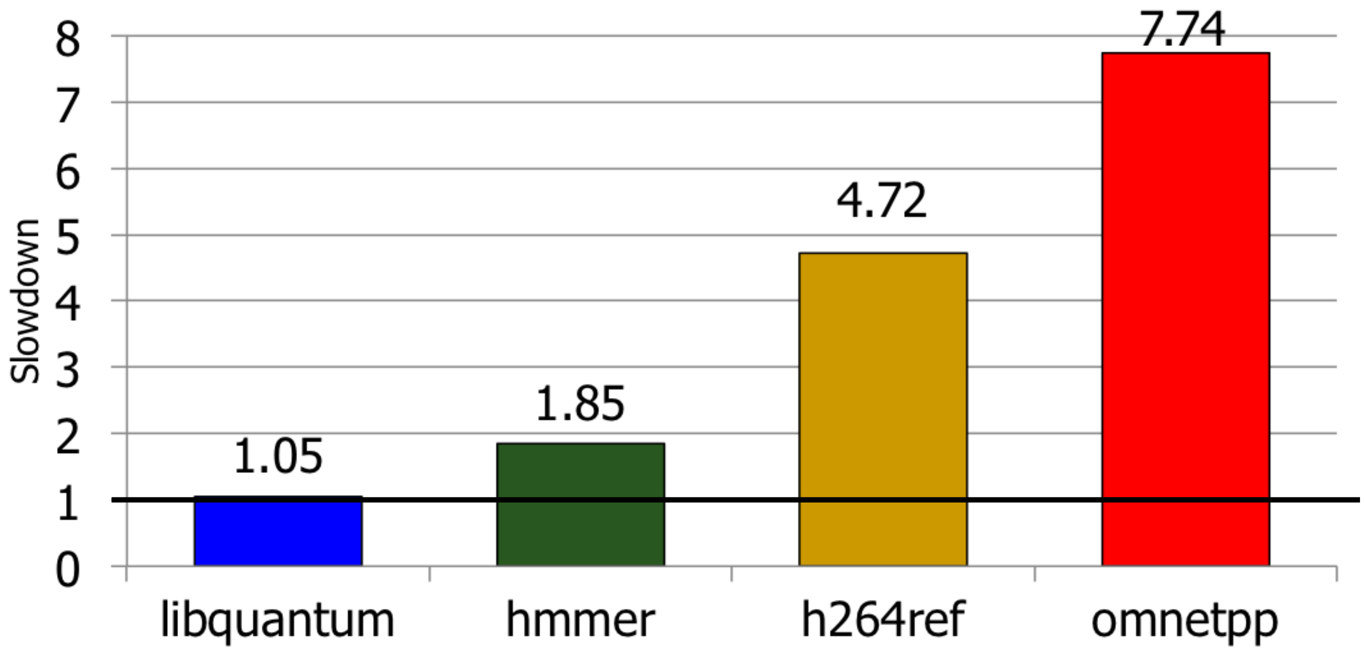
- DRAM-Aware last-level cache policies [Lee+ HPS Tech Report'10] [Lee+ HPS Tech Report'10]
  - Idea: Design cache eviction and replacement policies such that they proactively exploit the state of the memory controller and DRAM (e.g., proactively evict data from the cache that hit in open rows)
  - Takeaway: Coordination of last-level cache and DRAM policies 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

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}

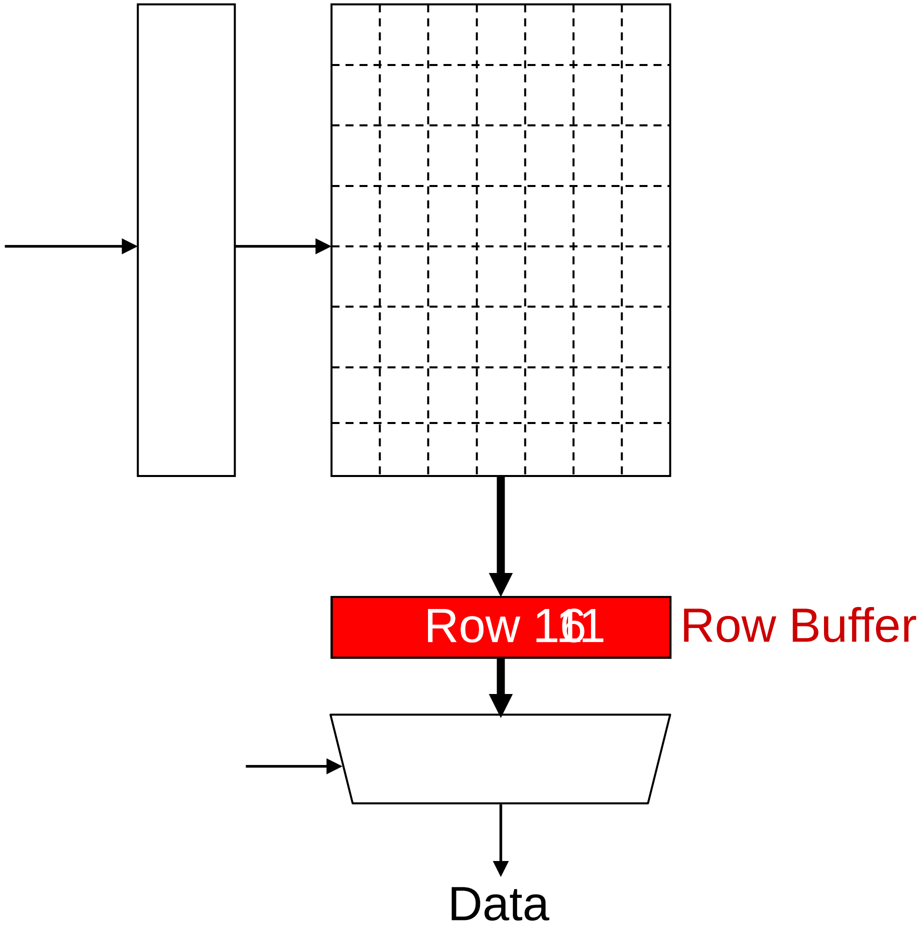
- Each cycle, the DRAM controller
  - Computes Slowdown = ST_{shared}/ST_{alone} for threads with legal requests
  - Computes unfairness = MAX Slowdown / MIN Slowdown

- 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
T1: Row 16

T0 Slowdown	1.04
T1 Slowdown	1.08
Unfairness	1.04
α	1.05



STFM Pros and Cons

- Upsides:
  - Identifies fairness as an issue in 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

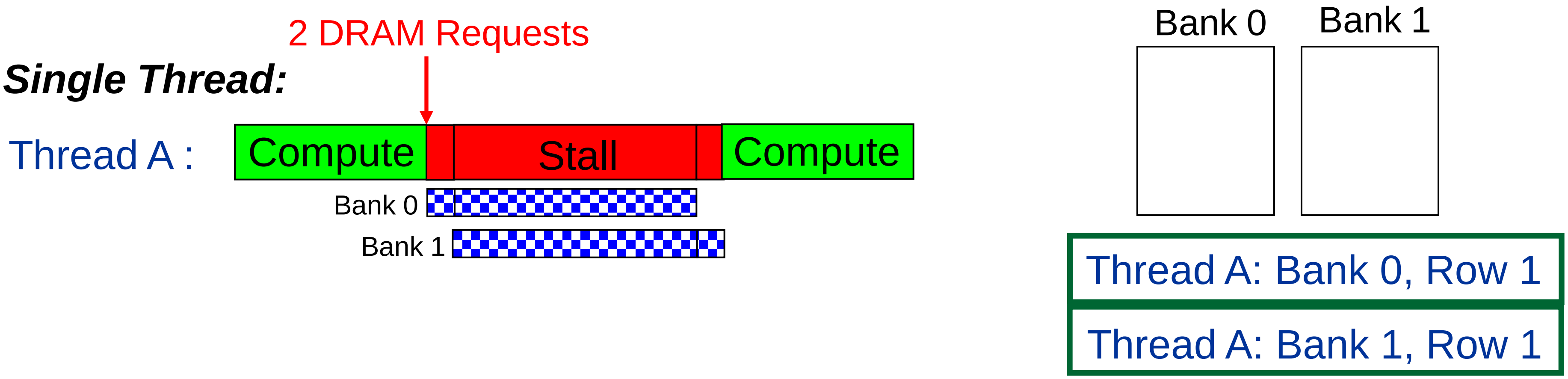
Another Problem due to Interference

- 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

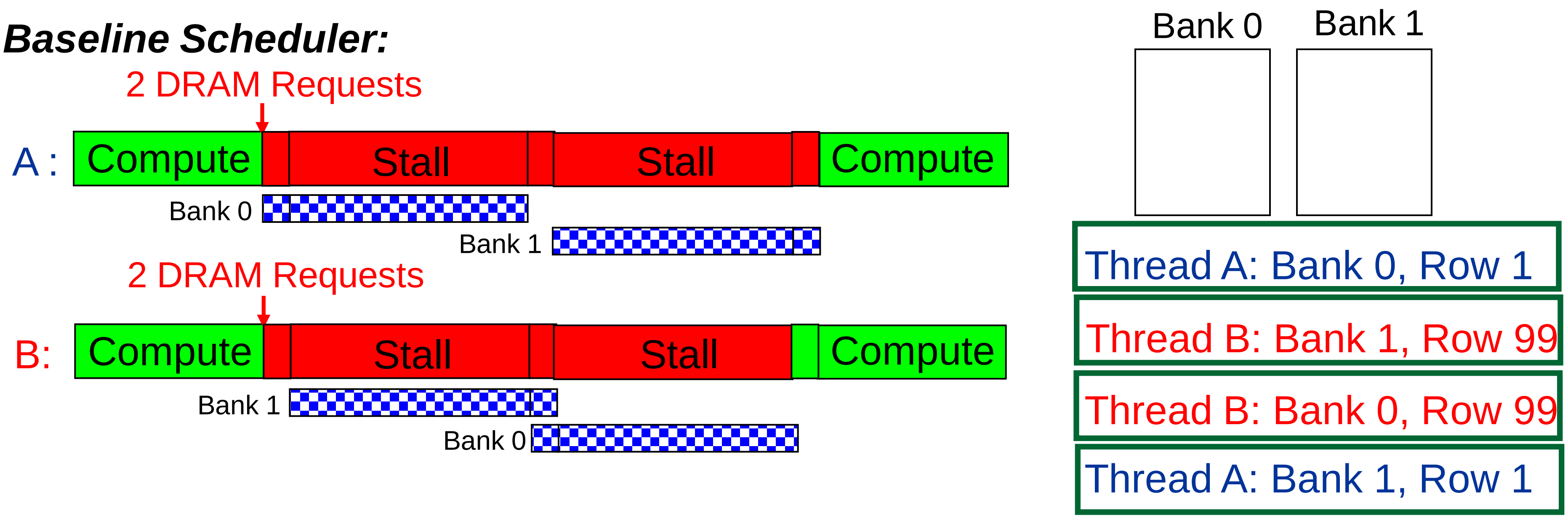
Bank Parallelism of a Thread



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

Bank Parallelism Interference in DRAM

Baseline Scheduler:



Thread A: Bank 0, Row 1

Thread B: Bank 1, Row 99

Thread B: Bank 0, Row 99

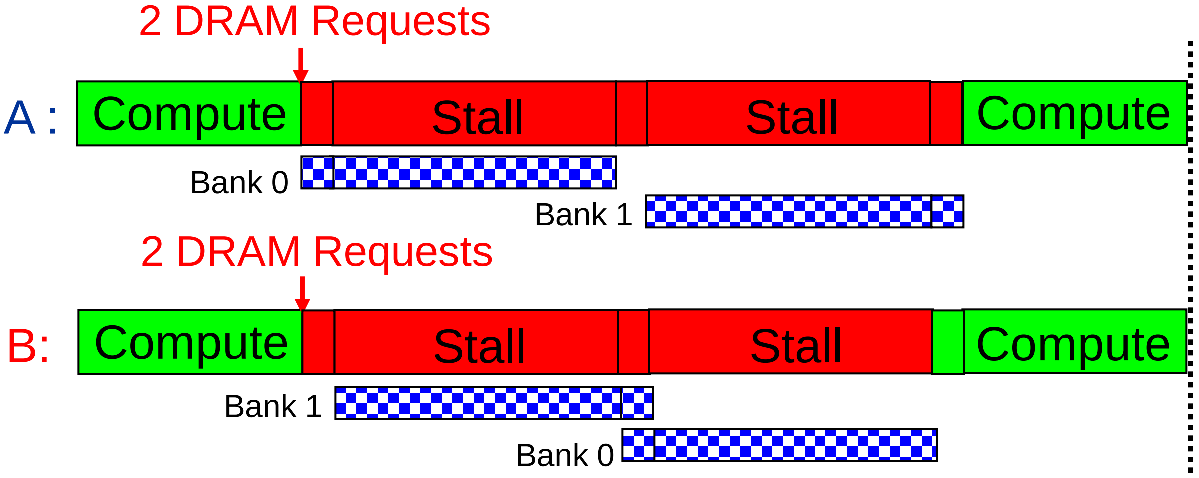
Thread A: Bank 1, Row 1

Bank access latencies of each thread serialized

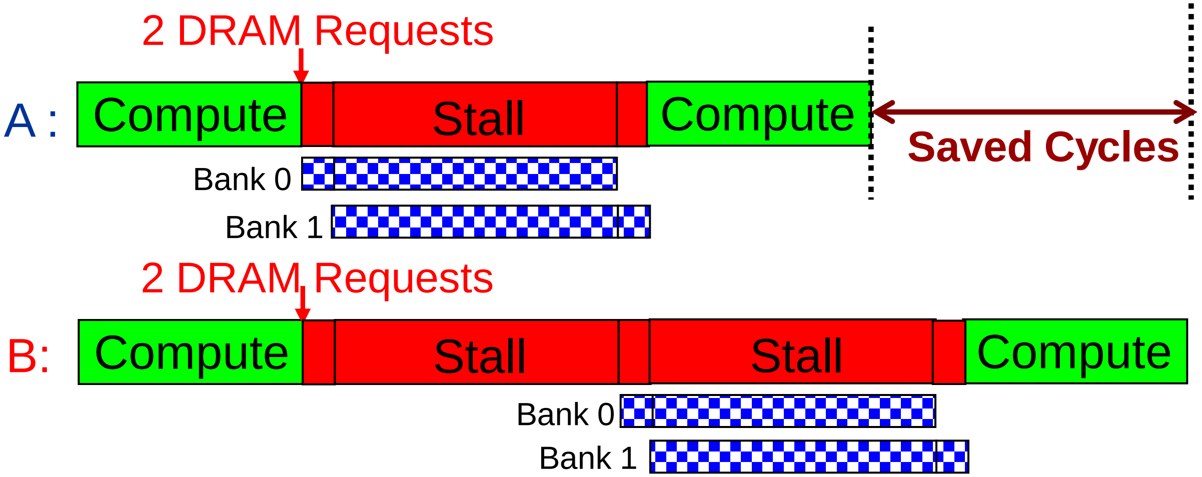
Each thread stalls for ~TWO bank access latencies

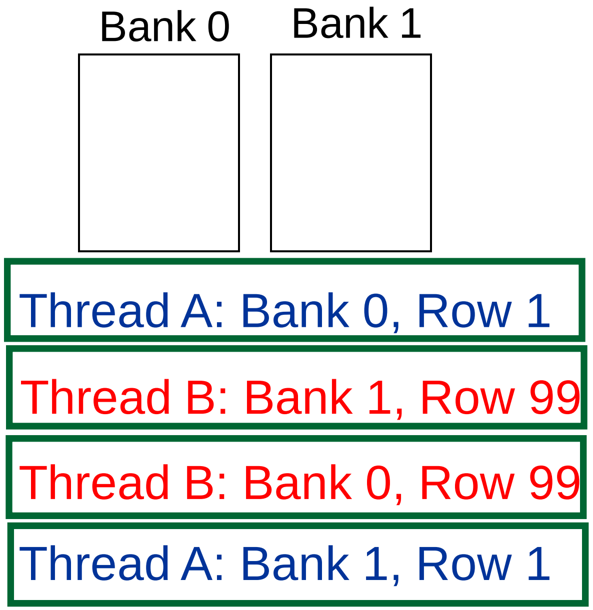
Parallelism-Aware Scheduler

Baseline Scheduler:



Parallelism-aware Scheduler:





Thread A: Bank 0, Row 1

Thread B: Bank 1, Row 99

Thread B: Bank 0, Row 99

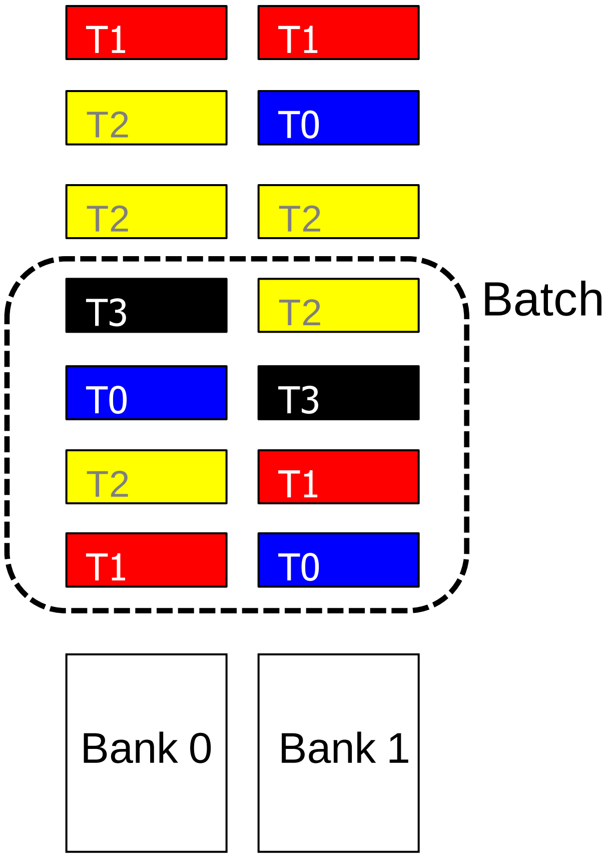
Thread A: Bank 1, Row 1

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



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 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

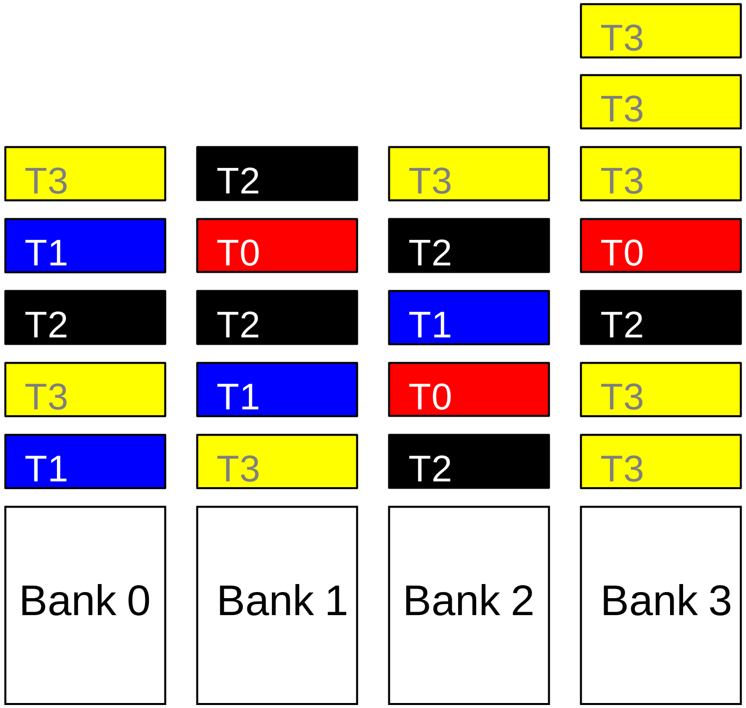
- Shortest stall-time first (shortest job first) ranking
  - Provides optimal system throughput [Smith, 1956]*
  - Controller estimates each thread's stall-time within the batch
  - Ranks threads with shorter stall-time higher

---

* 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

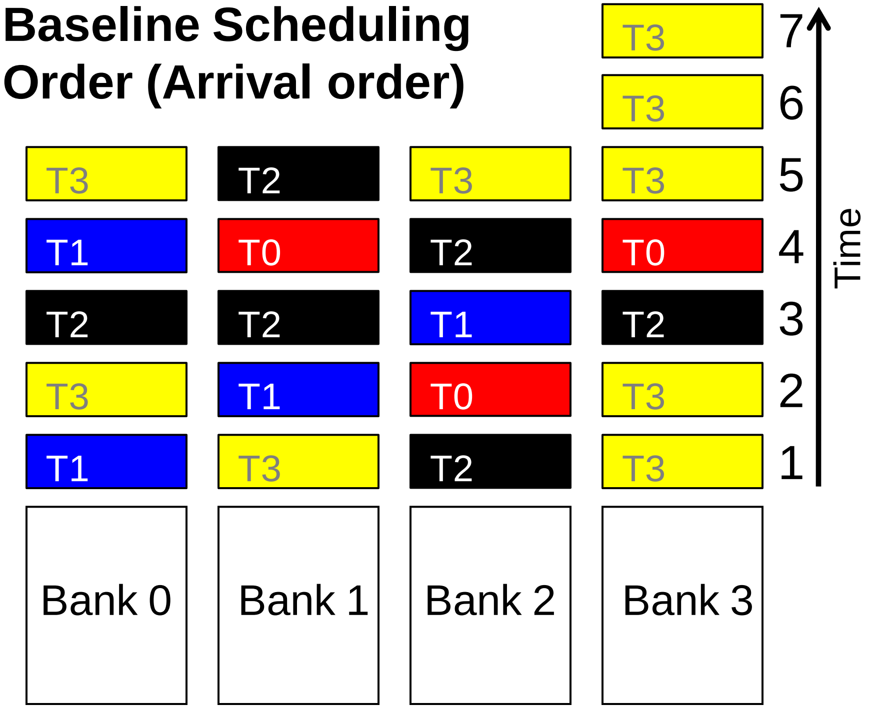


	max-bank-load	total-load

Ranking:
T0 > T1 > T2 > T3

Example Within-Batch Scheduling Order

Baseline Scheduling Order (Arrival order)

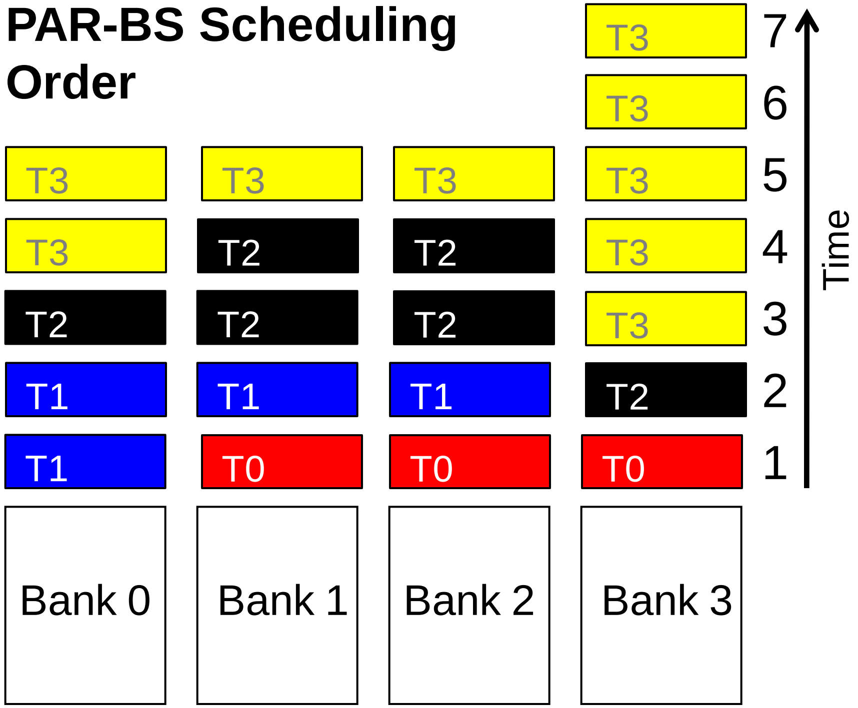


Time	Bank 0	Bank 1	Bank 2	Bank 3
7				T3
6				T3
5	T3	T2	T3	T3
4	T1	T0	T2	T0
3	T2	T2	T1	T2
2	T3	T1	T0	T3
1	T1	T3	T2	T3

	T0	**T1**	**T2**	**T3**
Stall times				

AVG: 5 bank access latencies

PAR-BS Scheduling Order



Time	Bank 0	Bank 1	Bank 2	Bank 3
7				T3
6				T3
5	T3	T3	T3	T3
4	T3	T2	T2	T3
3	T2	T2	T2	T3
2	T1	T1	T1	T2
1	T1	T0	T0	T0

Ranking: T0 > T1 > T2 > T3

	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
  - (2) Row-hit requests first
  - (3) Higher-rank thread first (shortest stall-time first)
  - (4) Oldest first

Batching

Parallelism-aware within-batch scheduling

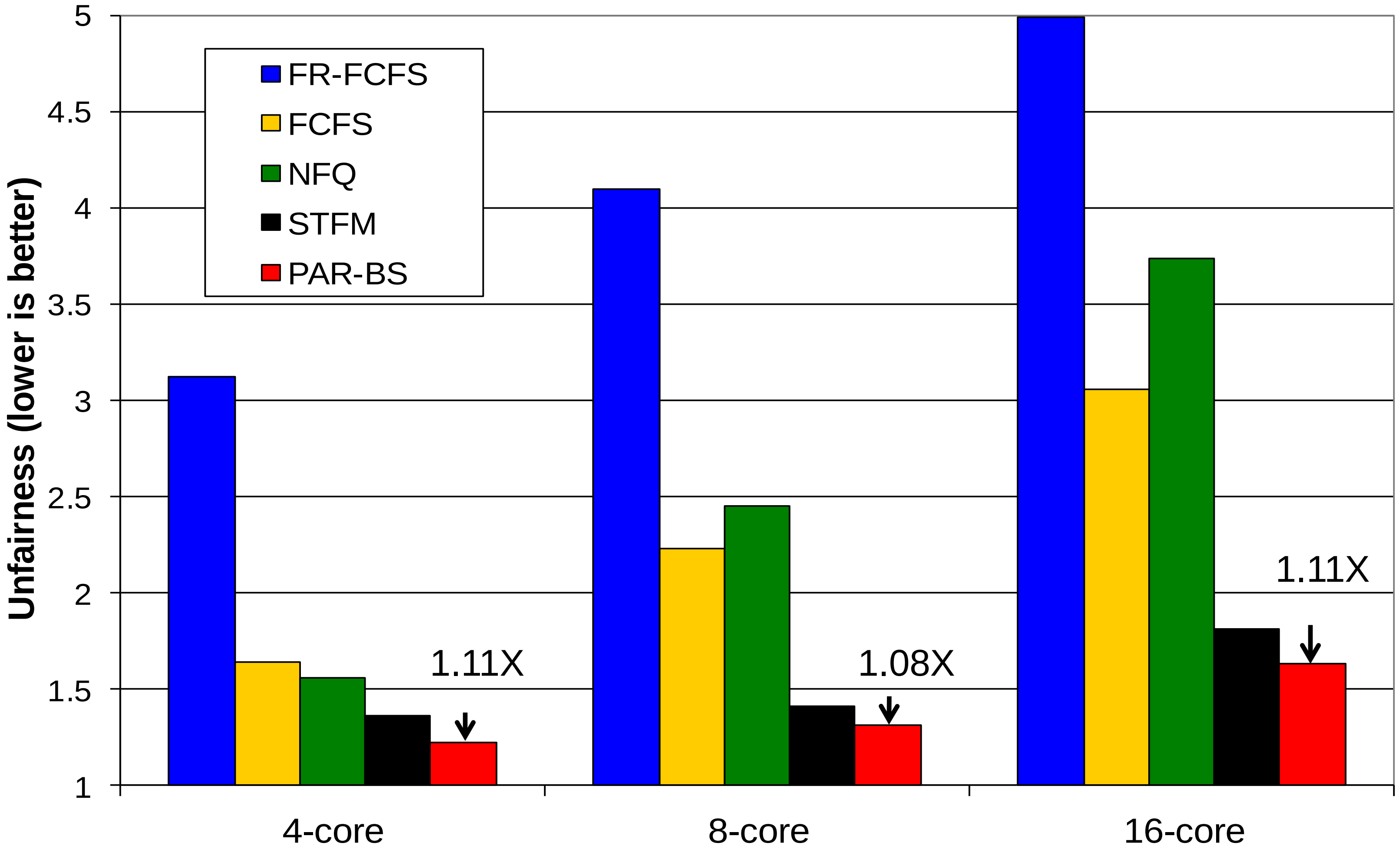
- Three properties:
  - Exploits row-buffer locality **and** intra-thread bank parallelism
  - Work-conserving
    - Services unmarked requests to banks without marked requests
  - Marking-Cap is important
    - Too small cap: destroys row-buffer locality
    - Too large cap: penalizes memory non-intensive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

Hardware Cost

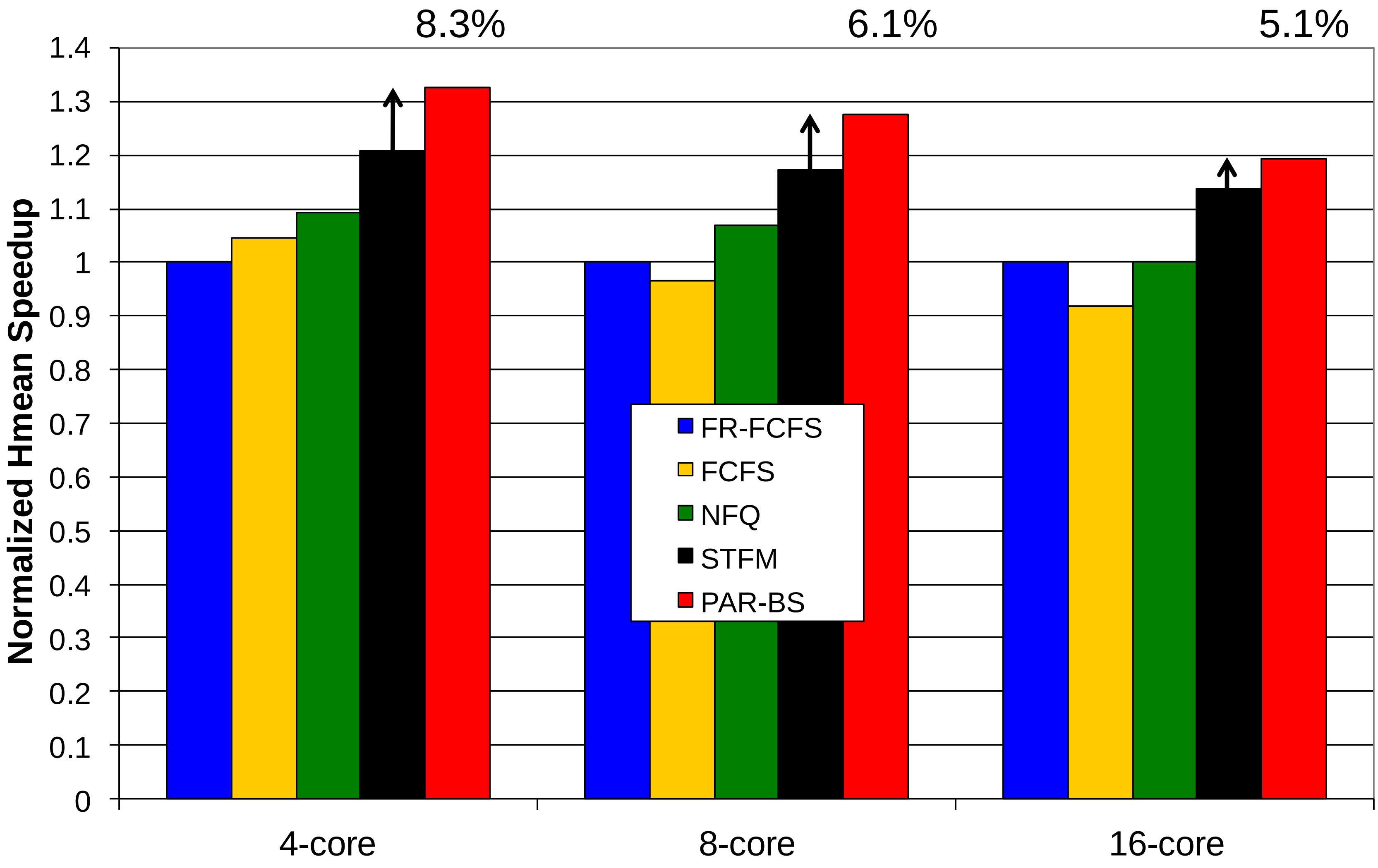
- <1.5KB storage cost for
  - 8-core system with 128-entry memory request buffer
- No complex operations (e.g., divisions)
- Not on the critical path
  - Scheduler makes a decision only every DRAM cycle

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

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



System Performance (Hmean-speedup)



PAR-BS Pros and Cons

- Upsides:
  - Identifies the problem of bank parallelism destruction across multiple threads
  - Simple mechanism

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

ATLAS Memory Scheduler

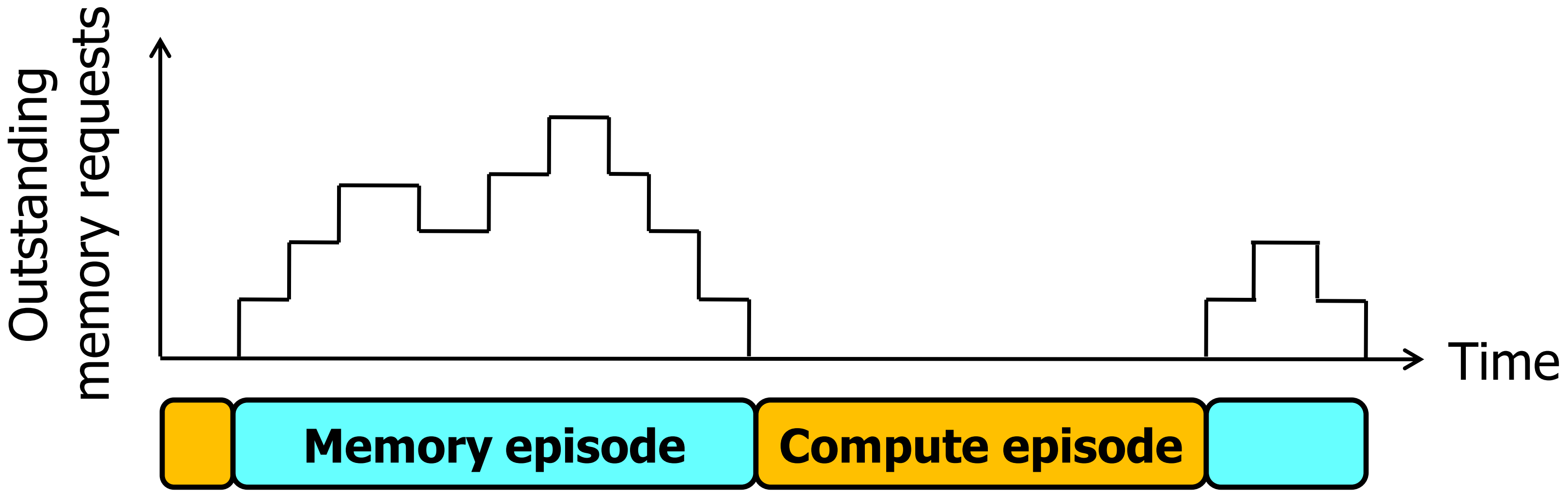
Yoongu Kim, Dongsu Han, Onur Mutlu, and Mor Harchol-Balter,
"ATLAS: A Scalable and High-Performance Scheduling Algorithm for Multiple Memory Controllers"
16th International Symposium on High-Performance Computer Architecture (***HPCA***),
Bangalore, India, January 2010. Slides (pptx)

ATLAS HPCA 2010 Talk

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**



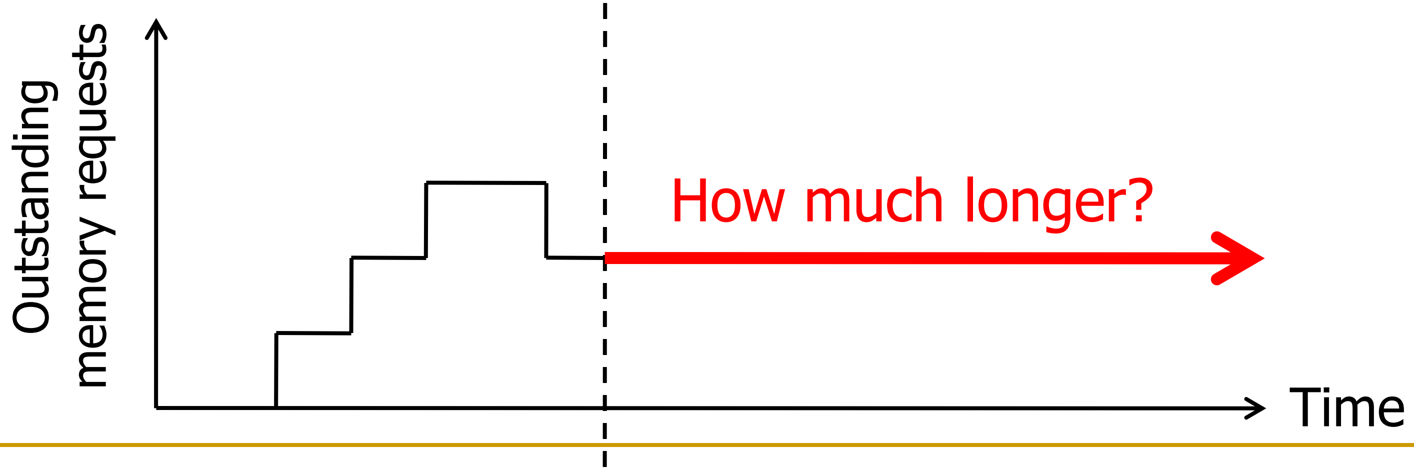
Goal: Minimize time spent in memory episodes

How to Minimize Memory Episode Time

Prioritize thread whose memory episode will end the soonest

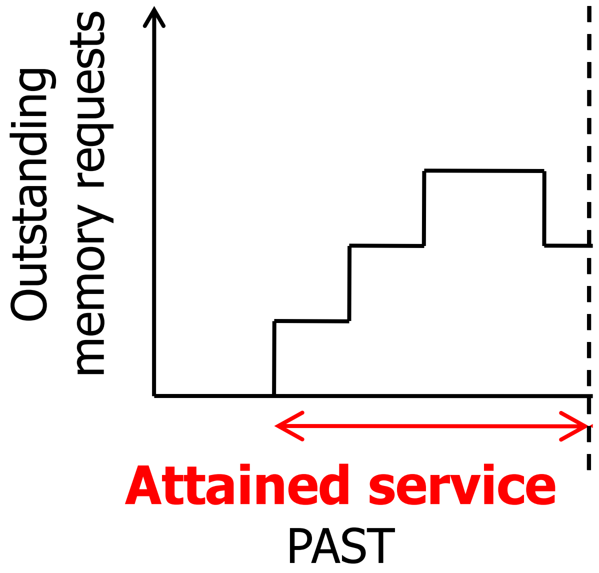
- Minimizes time spent in memory episodes across all threads
- Supported by queueing theory:
  - Shortest-Remaining-Processing-Time scheduling is optimal in single-server queue

Remaining length of a memory episode?



Predicting Memory Episode Lengths

We discovered: past is excellent predictor for future

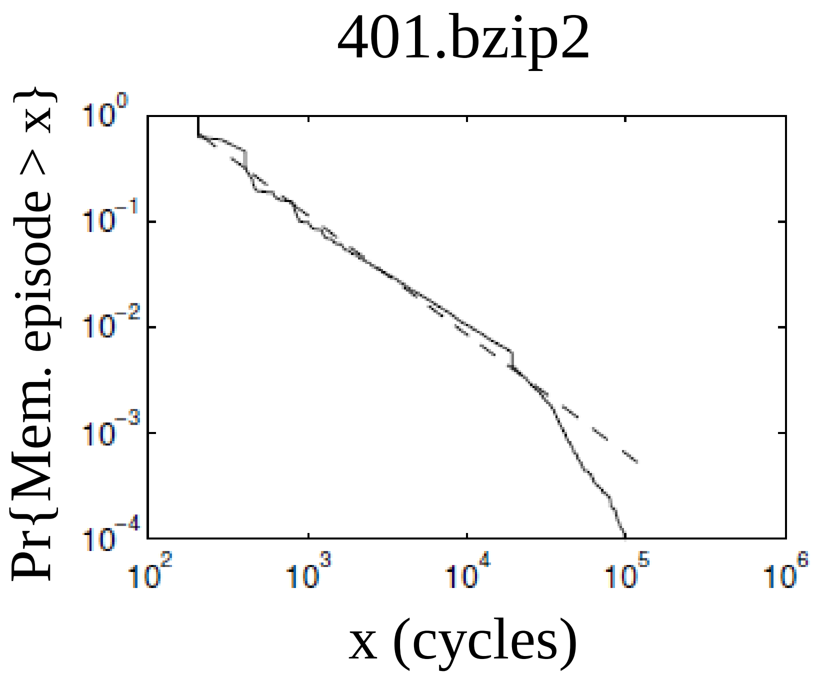


Large **attained service** ➔ Large expected **remaining service**

Q: Why?

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

Pareto Distribution of Memory Episode Lengths



Memory episode lengths of SPEC benchmarks

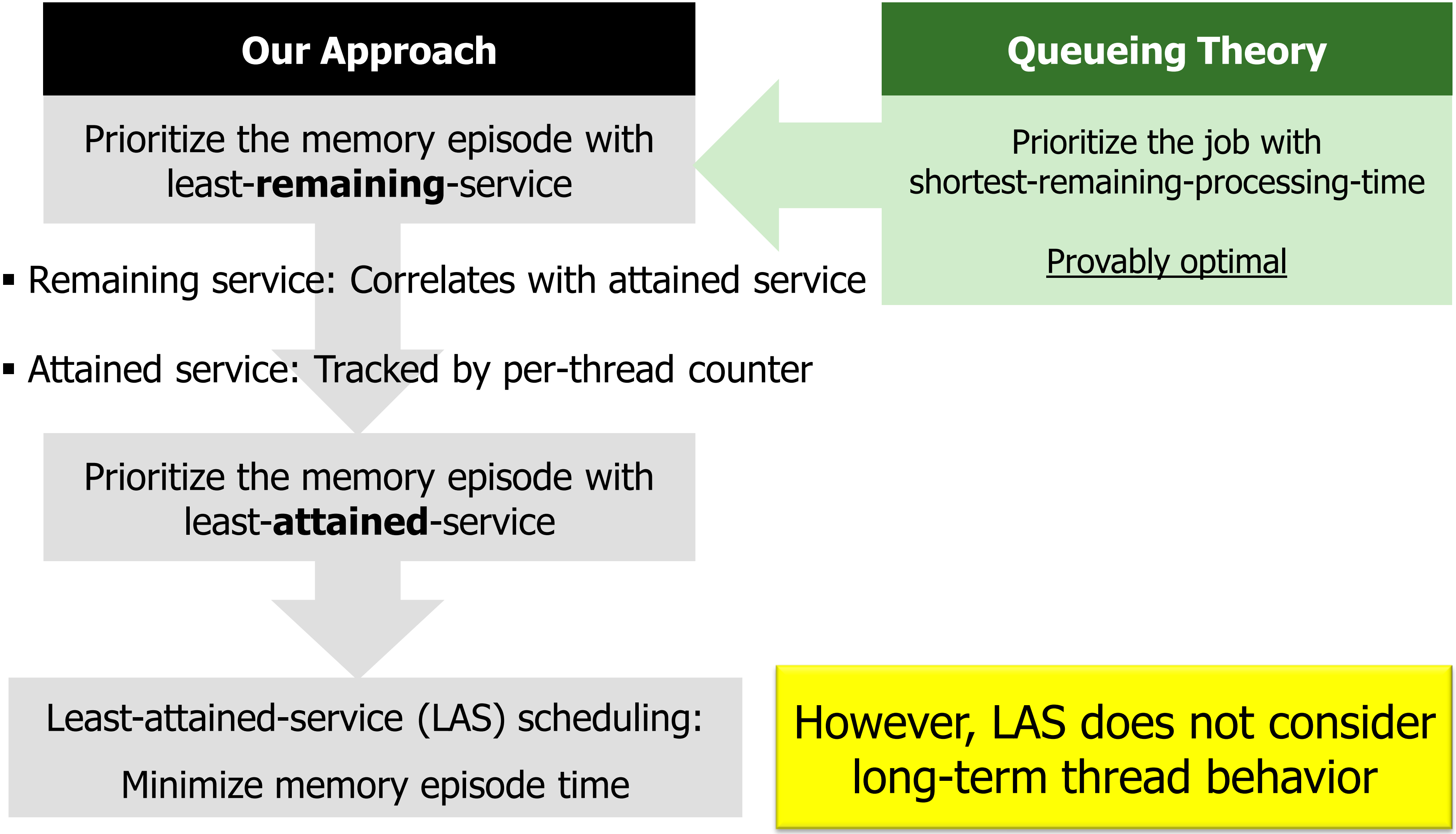
Pareto distribution

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**

Least Attained Service (LAS) Memory Scheduling



Queueing Theory

Prioritize the job with
shortest-remaining-processing-time

Provably optimal

Our Approach

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

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

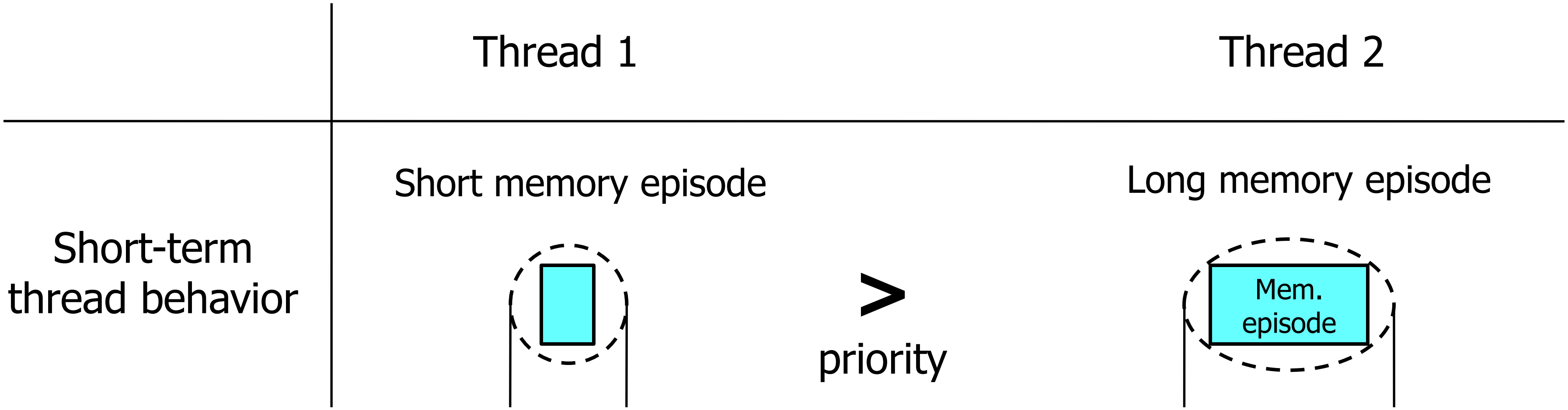
Prioritize the memory episode with
least-**attained**-service

Least-attained-service (LAS) scheduling:

Minimize memory episode time

However, LAS does not consider long-term thread behavior

Long-Term Thread Behavior



Thread 1

Thread 2

Short-term thread behavior

Short memory episode

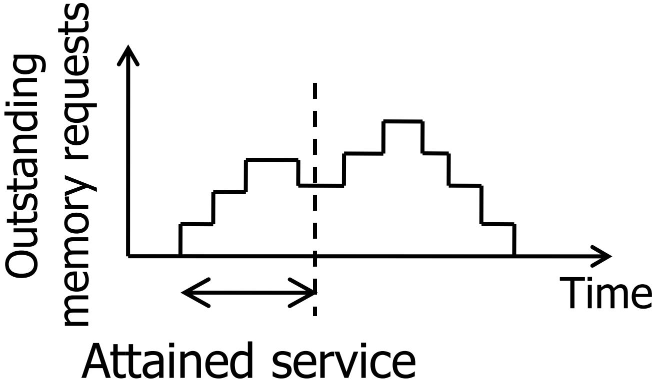
\> priority

Long memory episode

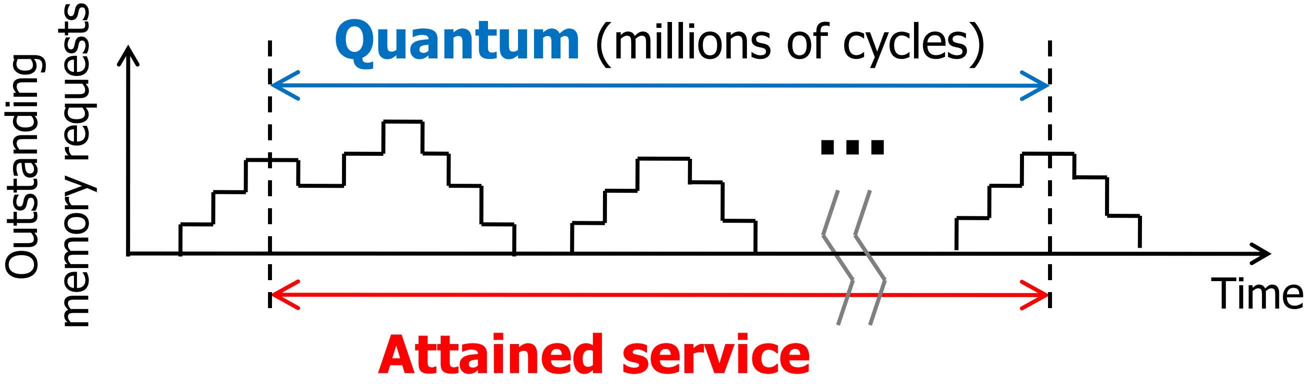
Prioritizing Thread 2 is more beneficial:
results in very long stretches of compute episodes

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

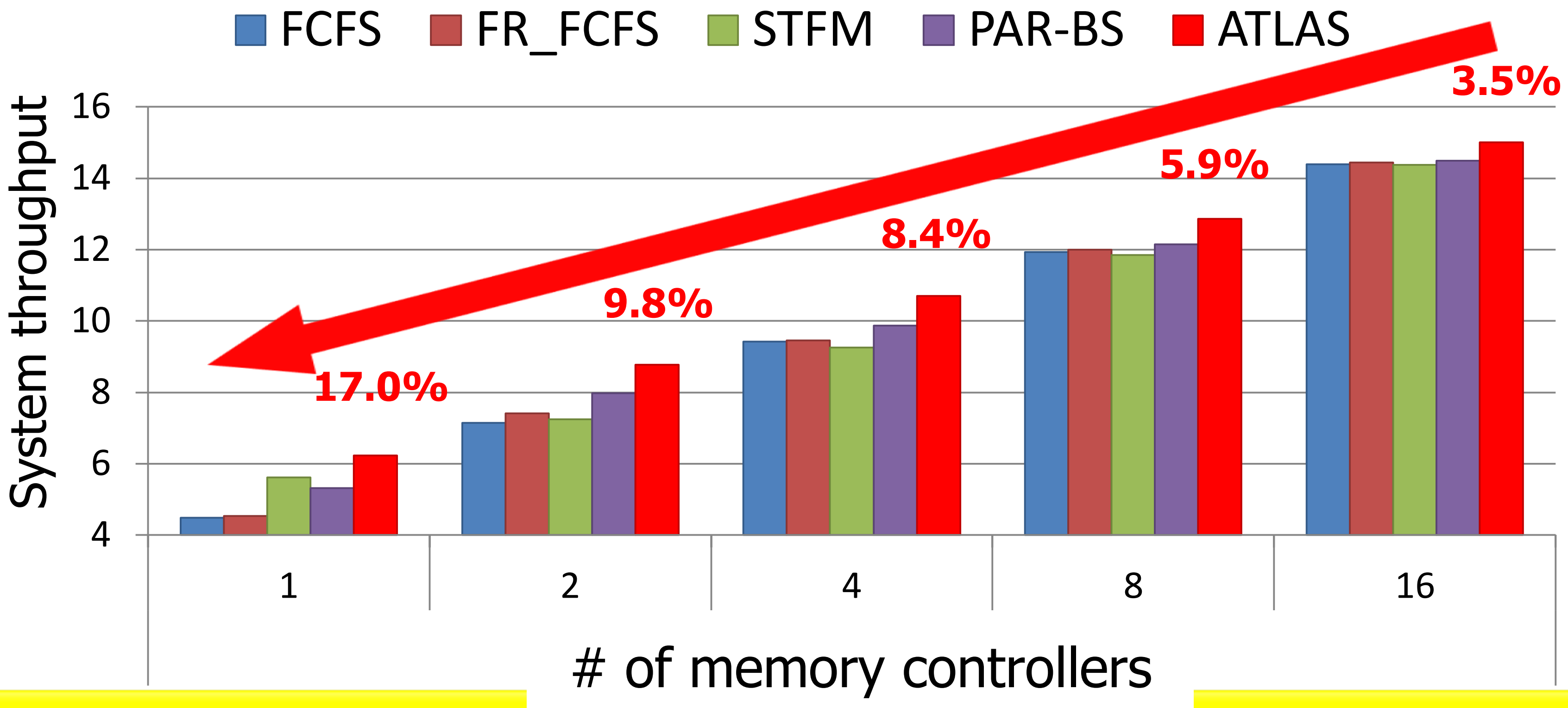
- Request prioritization order

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?

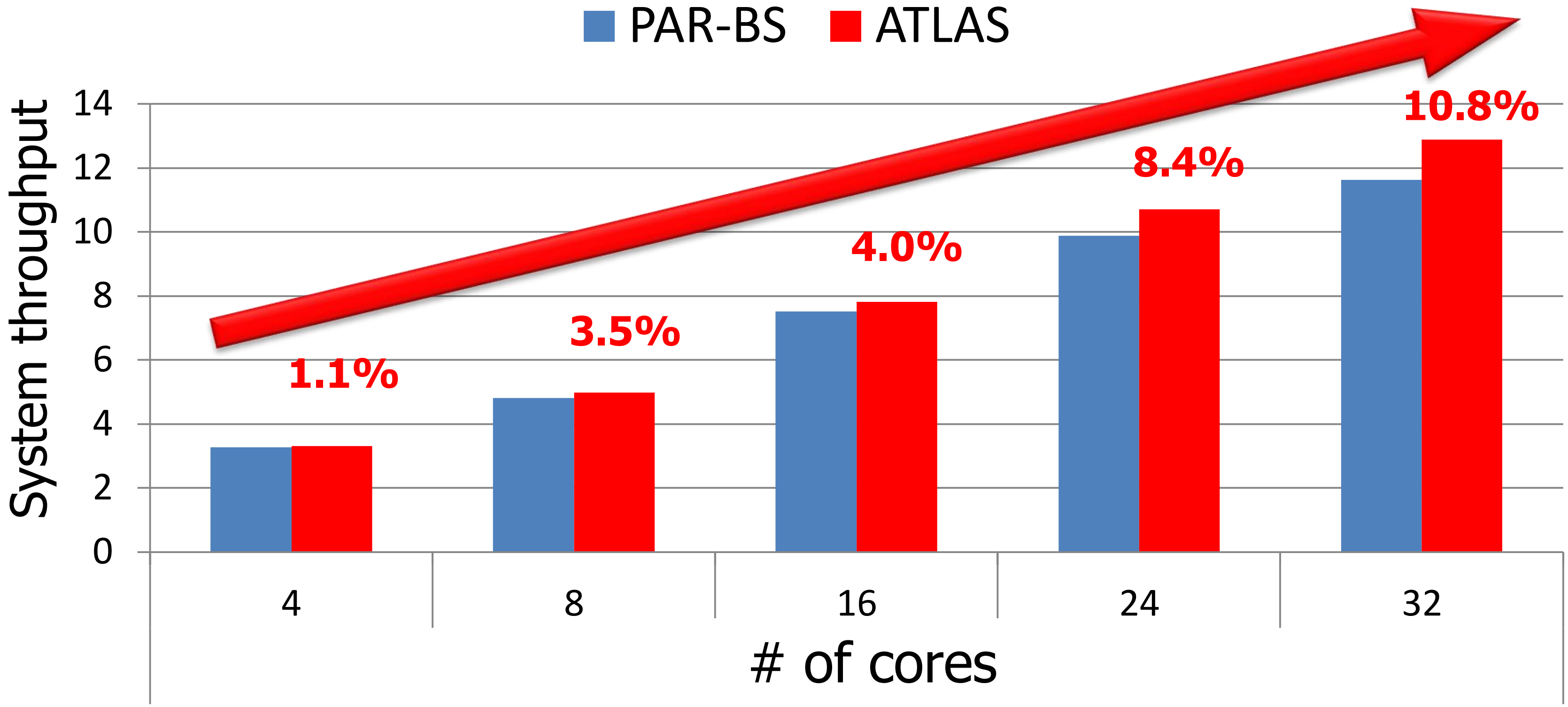
System Throughput: 24-Core System

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

ATLAS Pros and Cons

- Upsides:
  - Good at improving overall throughput (compute-intensive threads are prioritized)
  - Low complexity
  - Coordination among controllers happens infrequently

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

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)

TCM Micro 2010 Talk

Computer Architecture: Memory Interference and QoS (Part I)

Prof. Onur Mutlu
Carnegie Mellon University