18-740/640
Computer Architecture
Lecture 15: Memory Resource Management II

Prof. Onur Mutlu
Carnegie Mellon University

Fall 2015, 11/2/2015

Required Readings

- Required Reading Assignment:
  - Mutlu and Moscibroda, "Parallelism-Aware Batch Scheduling: Enhancing both Performance and Fairness of Shared DRAM Systems," ISCA 2008.

- Recommended References:
  - Muralidhara et al., "Reducing Memory Interference in Multicore Systems via Application-Aware Memory Channel Partitioning," MICRO 2011.
  - Ebrahimi et al., "Parallel Application Memory Scheduling," MICRO 2011.
  - Wang et al., "A-DRM: Architecture-aware Distributed Resource Management of Virtualized Clusters," VEE 2015.

Guest Lecture Tomorrow (11/3, Tuesday)

- Mike O’Connor, NVIDIA
  - Advances in GPU architecture and GPU memory systems
  - HH 1107, 7:30pm Pittsburgh Time

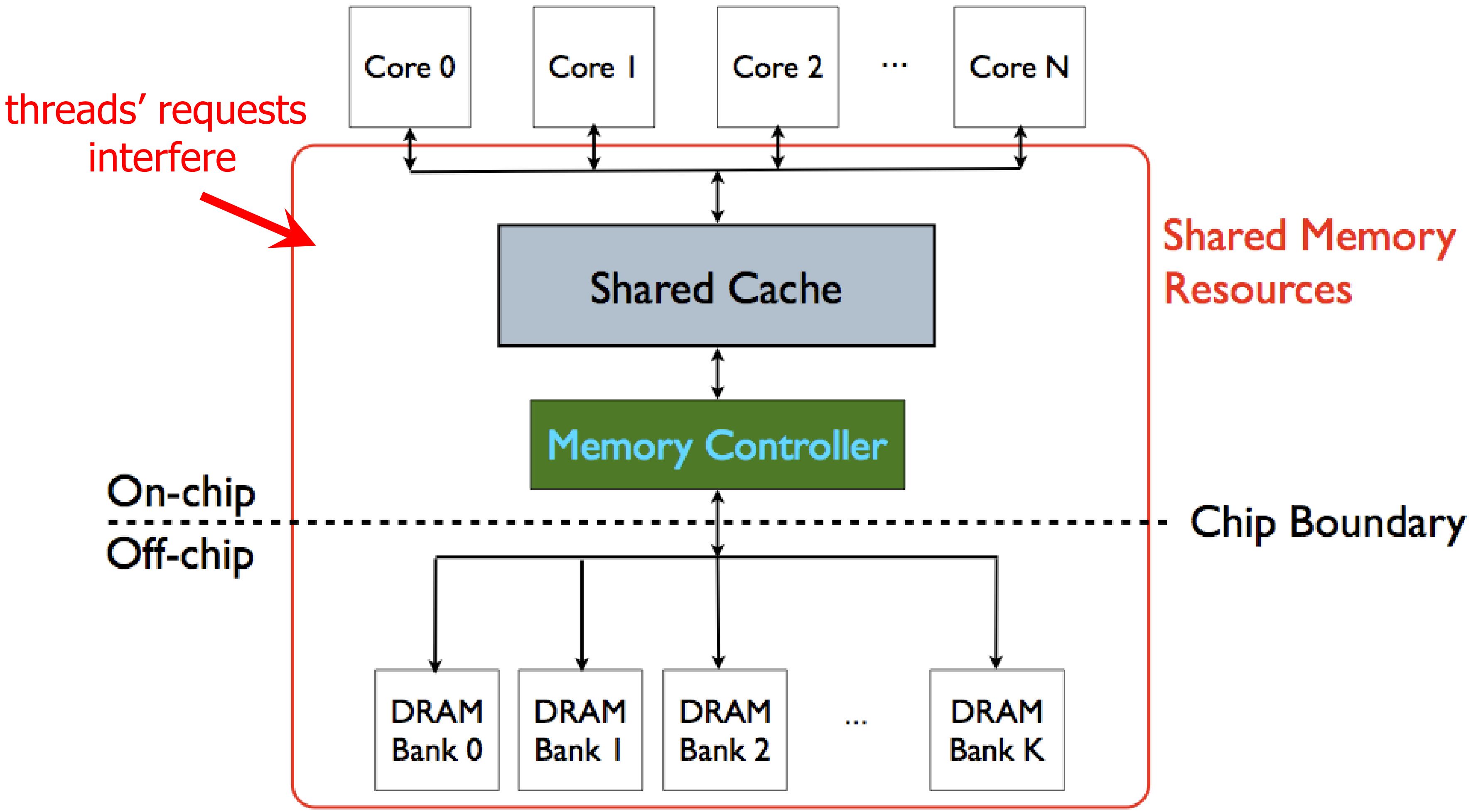
CALCM Seminar Tomorrow (11/3)

- High-bandwidth, Energy-efficient DRAM Architectures for GPU Systems
- Mike O'Connor, NVIDIA
  - CIC Panther Hollow Room (4th Floor), 4:30pm

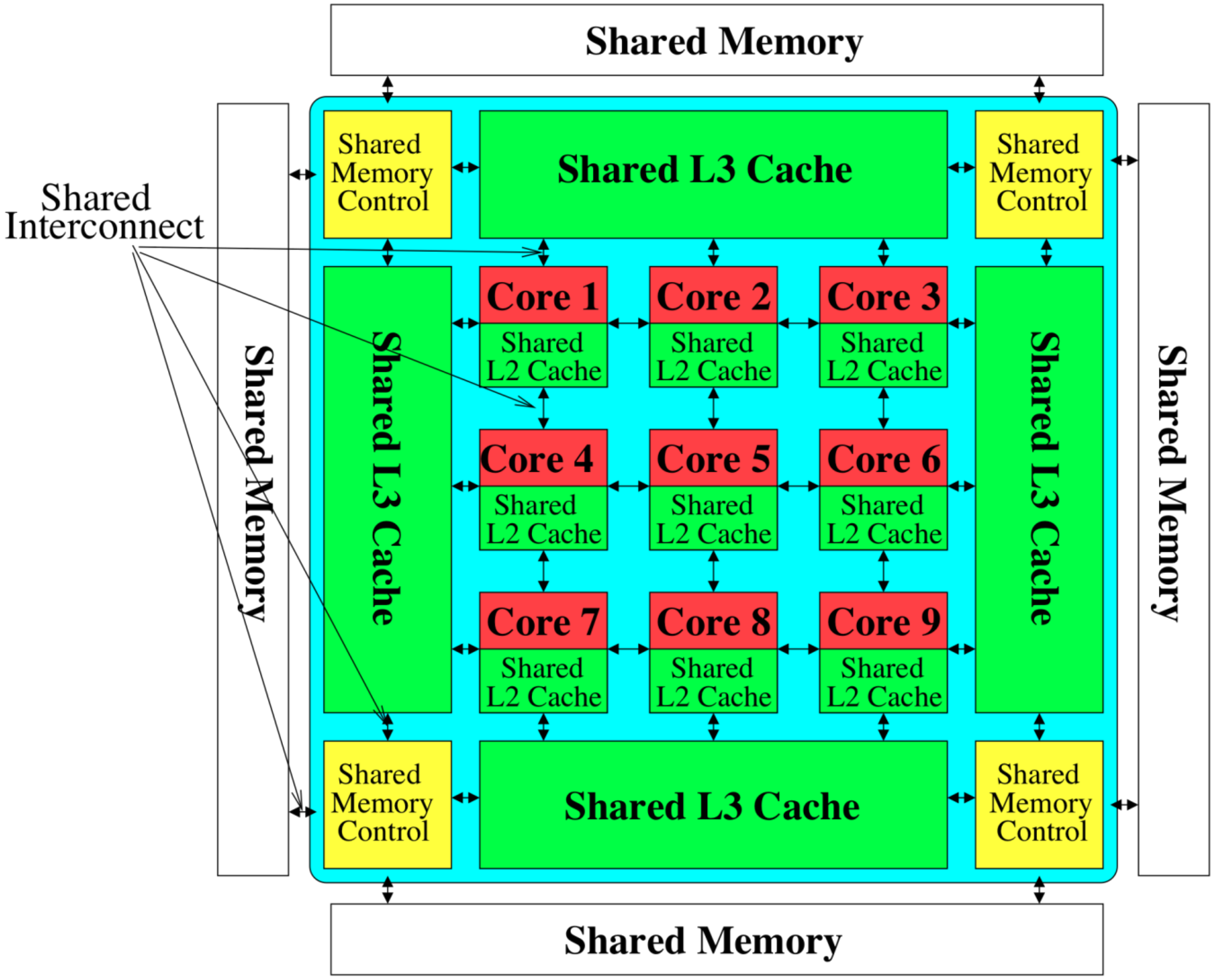
- https://www.ece.cmu.edu/~calcm/doku.php?id=seminars:seminar_11_03_15

Shared Resource Design for Multi-Core Systems

Memory System is the Major Shared Resource



Much More of a Shared Resource in Future

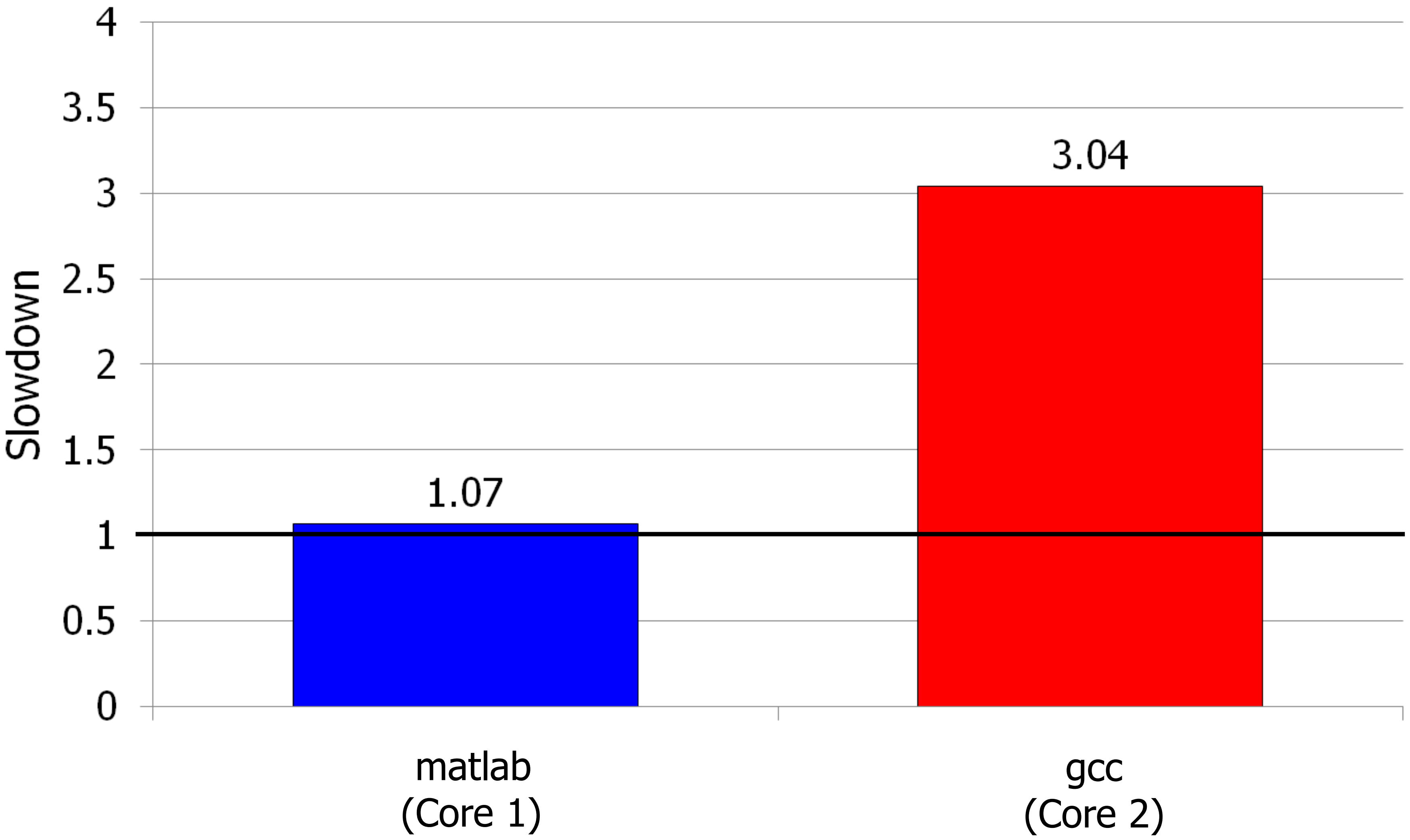


Inter-Thread/Application Interference

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

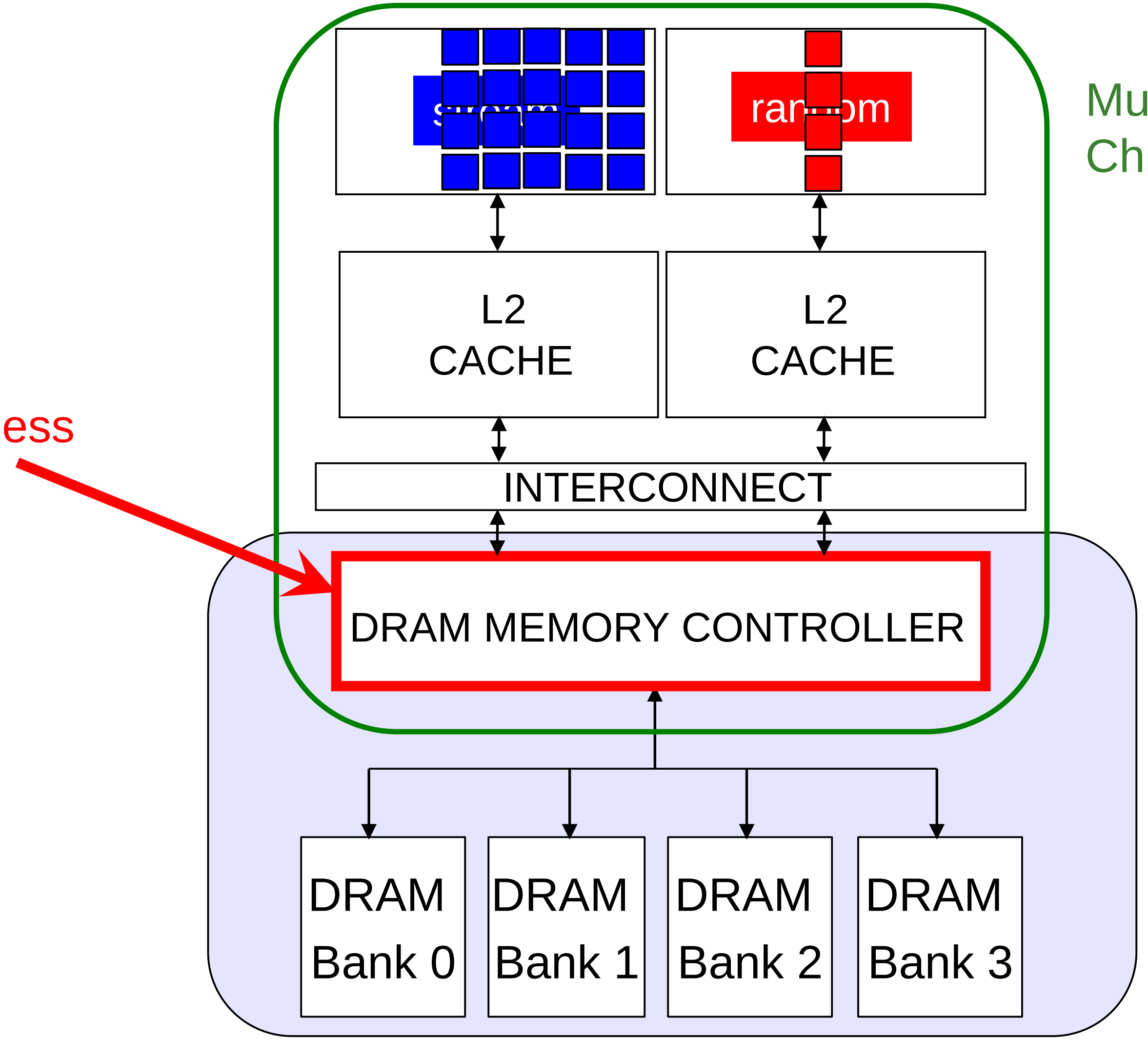
- Existing memory systems
  - Free-for-all, shared based on demand
  - Control algorithms thread-unaware and thread-unfair
  - Aggressive threads can deny service to others
  - Do not try to reduce or control inter-thread interference

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



stream

random

L2
CACHE

L2
CACHE

INTERCONNECT

DRAM MEMORY CONTROLLER

DRAM
Bank 0

DRAM
Bank 1

DRAM
Bank 2

DRAM
Bank 3

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

```
// initialize large arrays A, B

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

random

STREAM

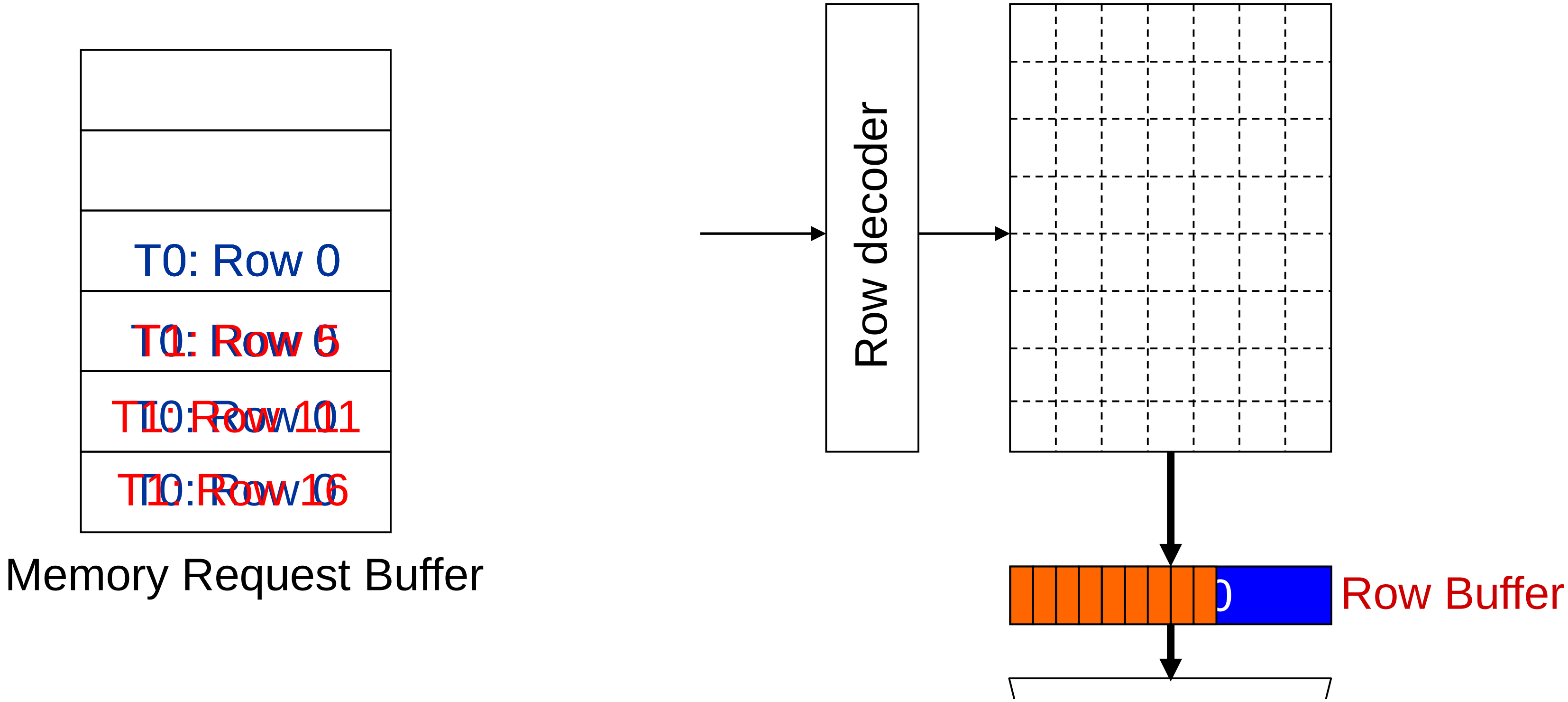
- Sequential memory access
- Very high row buffer locality (96% hit rate)
- Memory intensive

RANDOM

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

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

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]*

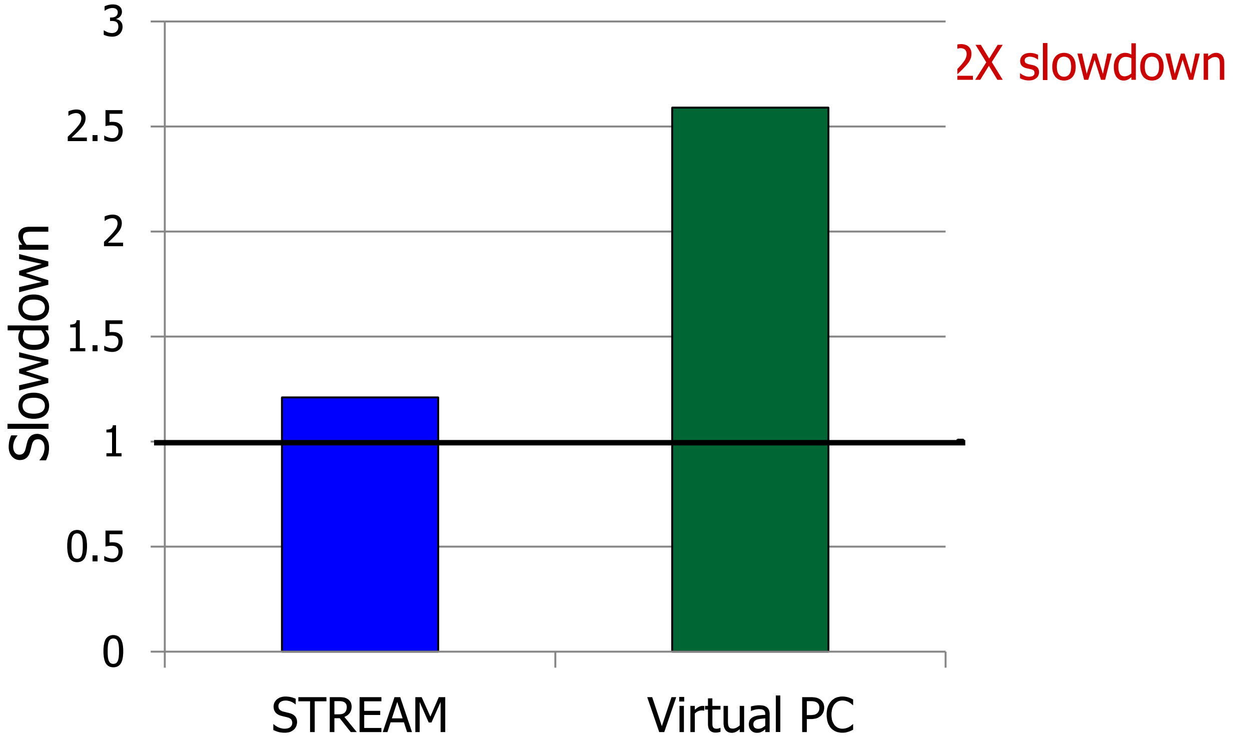
  (1) Row-hit first: Service row-hit memory accesses first

  (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

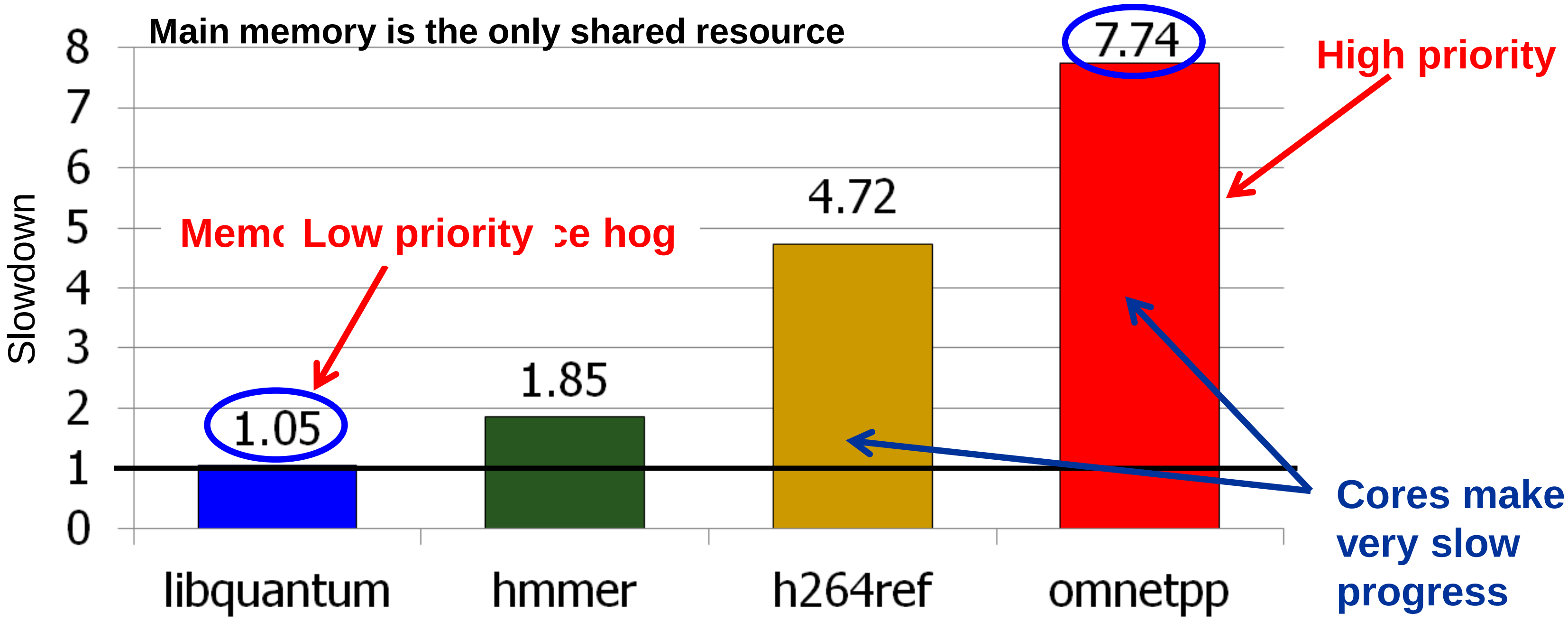


2X slowdown

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.

Problems due to Uncontrolled Interference



Main memory is the only shared resource

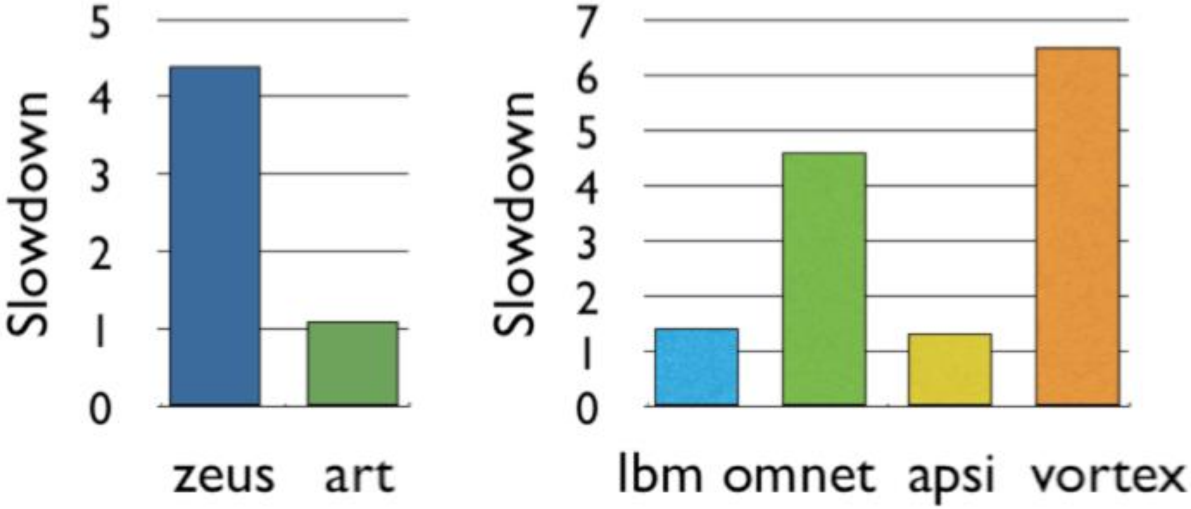
Memc Low priority ce hog

High priority

Cores make very slow progress

- 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)

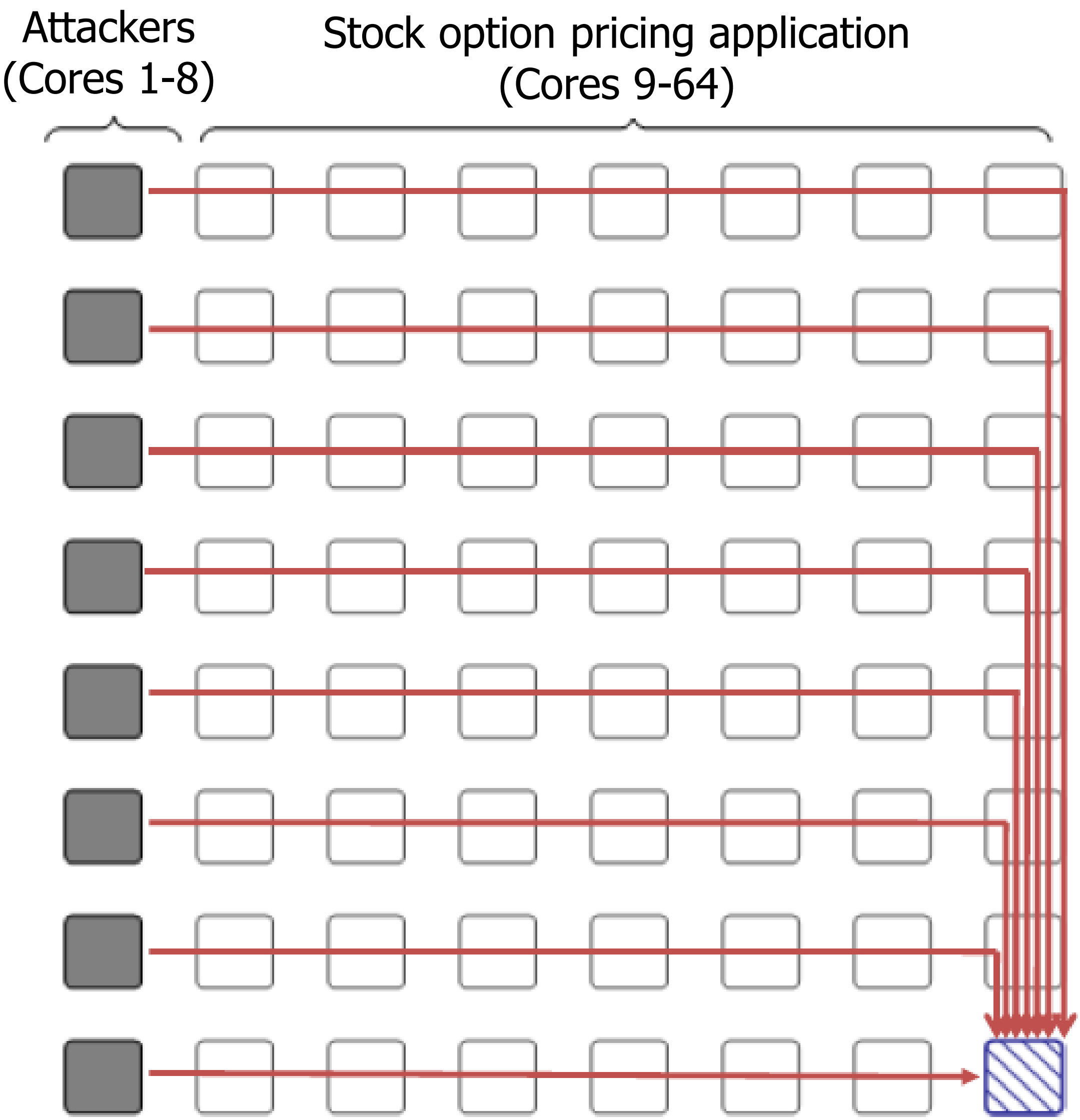
Uncontrollable, unpredictable system

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
  - QoS-aware interconnects
  - 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
  - QoS-aware data mapping to memory controllers
  - QoS-aware thread scheduling to cores

Fundamental Interference Control Techniques

- Goal: to reduce/control inter-thread memory interference

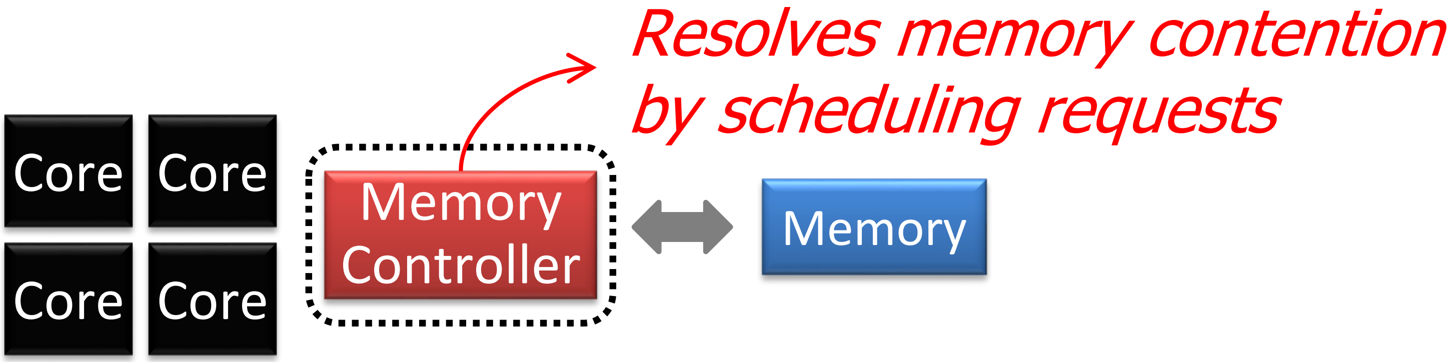
1. Prioritization or request scheduling

2. Data mapping to banks/channels/ranks

3. Core/source throttling

4. Application/thread scheduling

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
  - 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

- BLISS: Blacklisting Memory Scheduler [Subramanian+ ICCD'14]
  - Idea: Deprioritize (i.e., blacklist) a thread that has consecutively serviced a large number of requests
  - Takeaway: Blacklisting greatly reduces interference enables the scheduler to be simple without requiring full thread ranking

QoS-Aware Memory Scheduling: Evolution

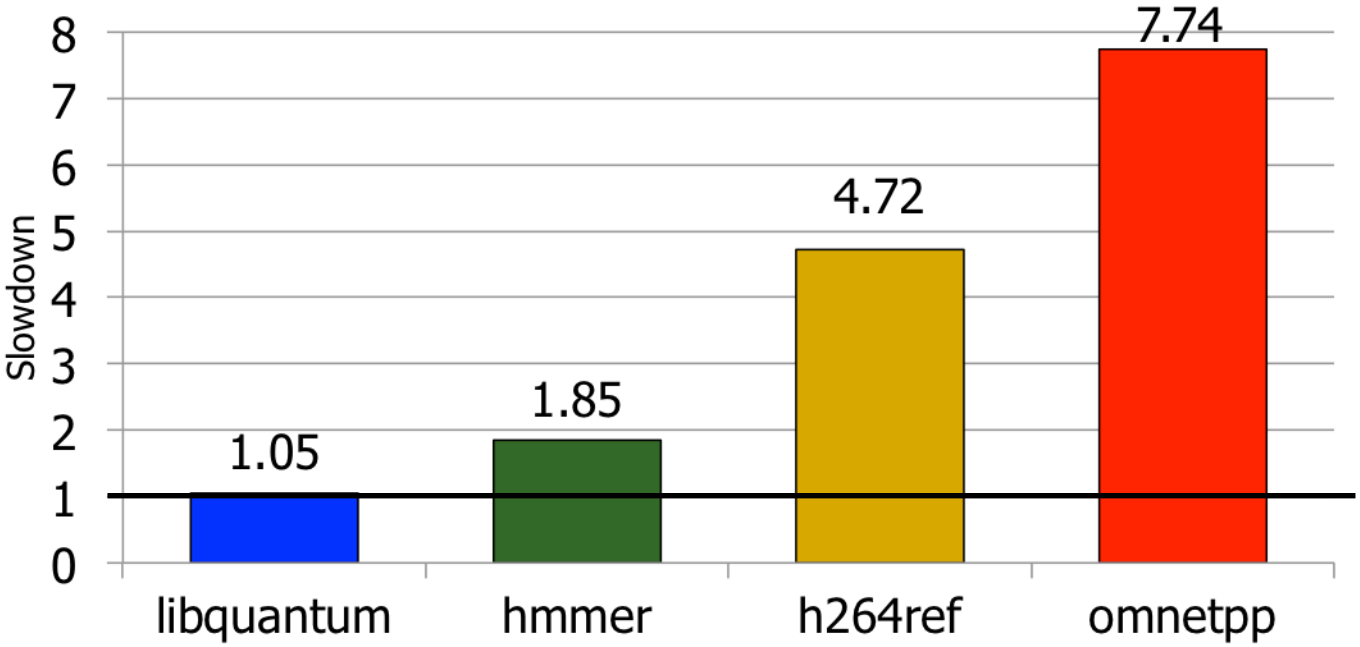
- Prefetch-aware shared resource management [Ebrahimi+ ISCA'11] [Ebrahimi+ MICRO'09] [Ebrahimi+ HPCA'09] [Lee+ MICRO'08]
    - Idea: Prioritize prefetches depending on how they affect system performance; even accurate prefetches can degrade performance of the system
    - Takeaway: Carefully controlling and prioritizing prefetch requests improves performance and fairness

- DRAM-Aware last-level cache policies and write scheduling [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)

The Problem: Unfairness



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

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

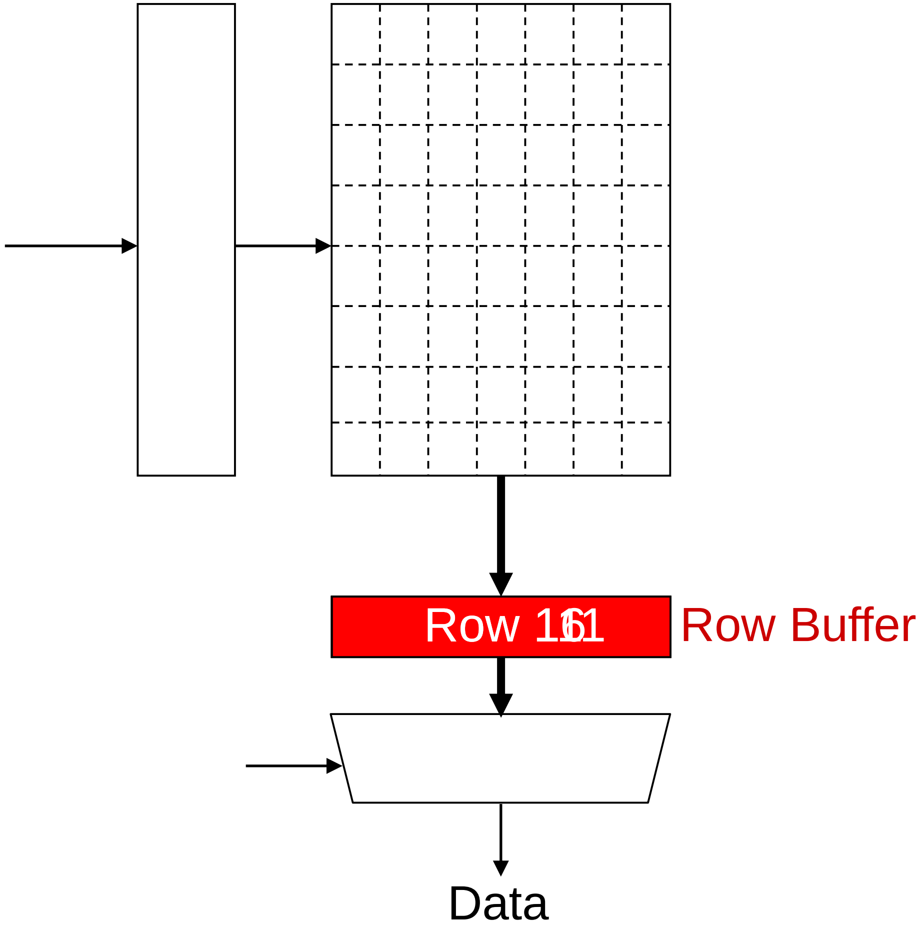
T0: Row 0

T1: Row 111

T0: Row 0

T1: Row 16

T0 Slowdown	1.04
T1 Slowdown	1.08
Unfairness	1.06
α	1.05



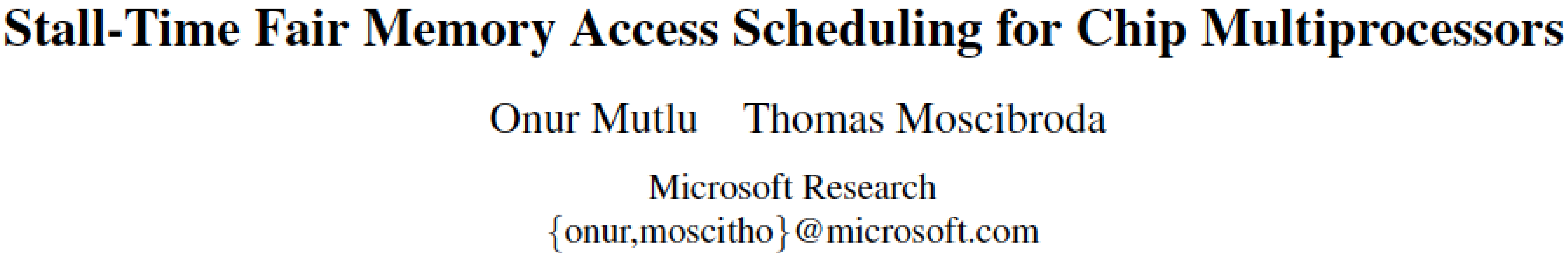
STFM Pros and Cons

- 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

More on STFM

- Onur Mutlu and Thomas Moscibroda, **"Stall-Time Fair Memory Access Scheduling for Chip Multiprocessors"** *Proceedings of the 40th International Symposium on Microarchitecture (**MICRO**)*, pages 146-158, Chicago, IL, December 2007. [Summary] [Slides (ppt)]



Stall-Time Fair Memory Access Scheduling for Chip Multiprocessors

Onur Mutlu   Thomas Moscibroda

Microsoft Research
{onur,moscitho}@microsoft.com

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

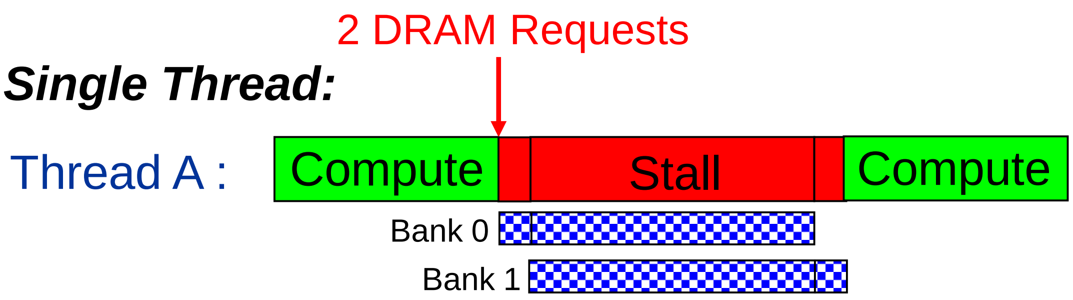
Another Problem due to Memory Interference

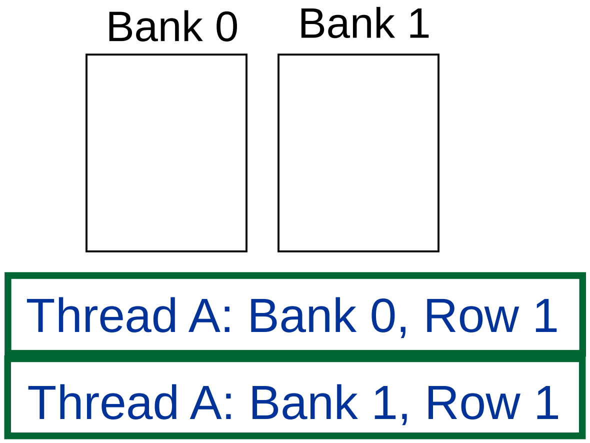
- Processors try to tolerate the latency of DRAM requests by generating multiple outstanding requests
  - Memory-Level Parallelism (MLP)
  - Out-of-order execution, non-blocking caches, runahead execution

- 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

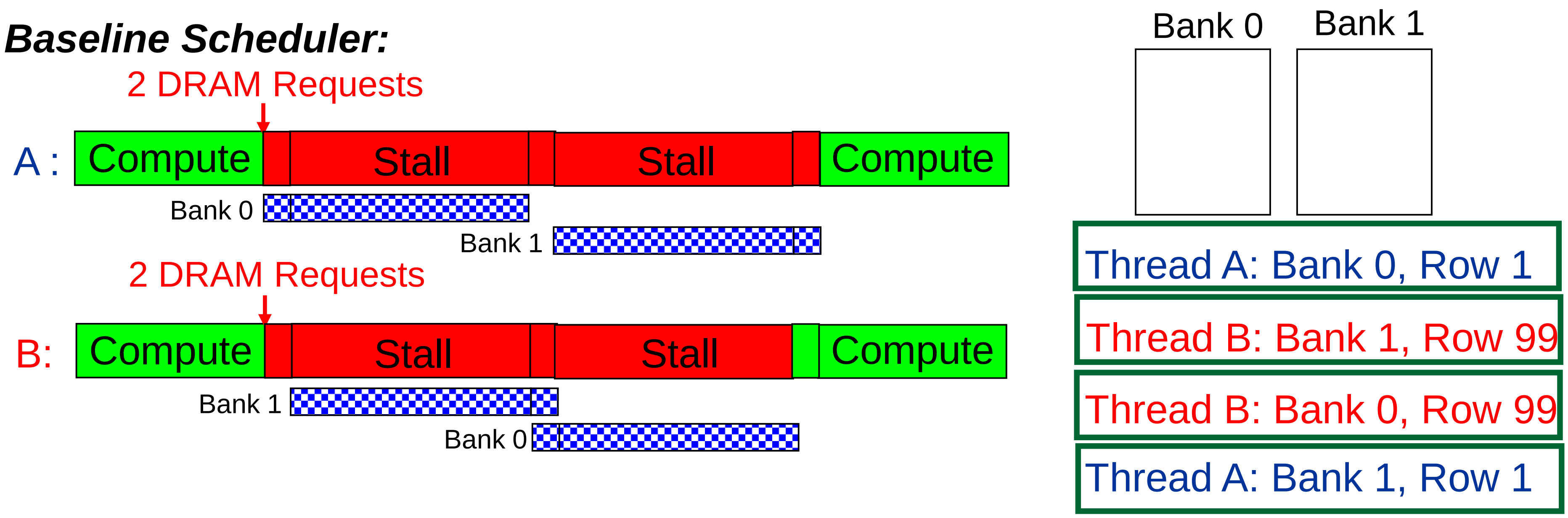
Bank Parallelism of a Thread





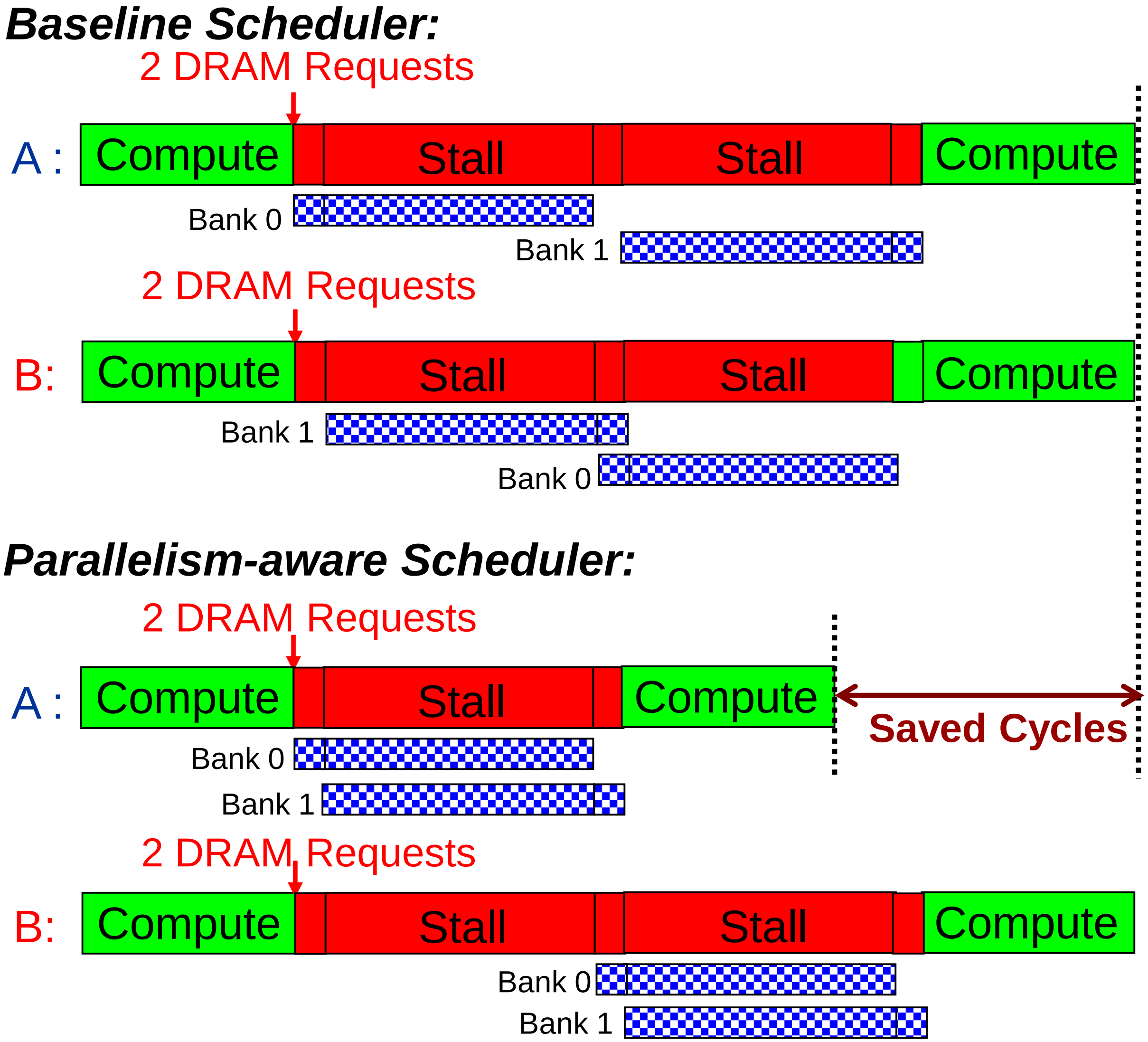
Bank access latencies of the two requests overlapped
Thread stalls for ~ONE bank access latency

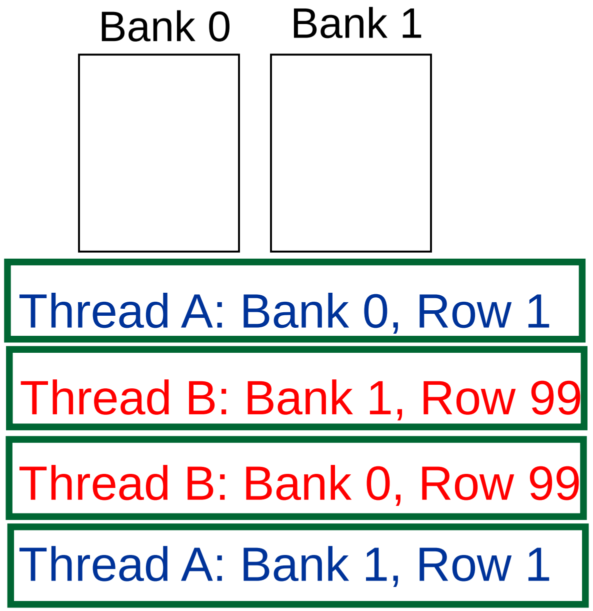
Bank Parallelism Interference in DRAM



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

Parallelism-Aware Scheduler





Thread A: Bank 0, Row 1

Thread B: Bank 1, Row 99

Thread B: Bank 0, Row 99

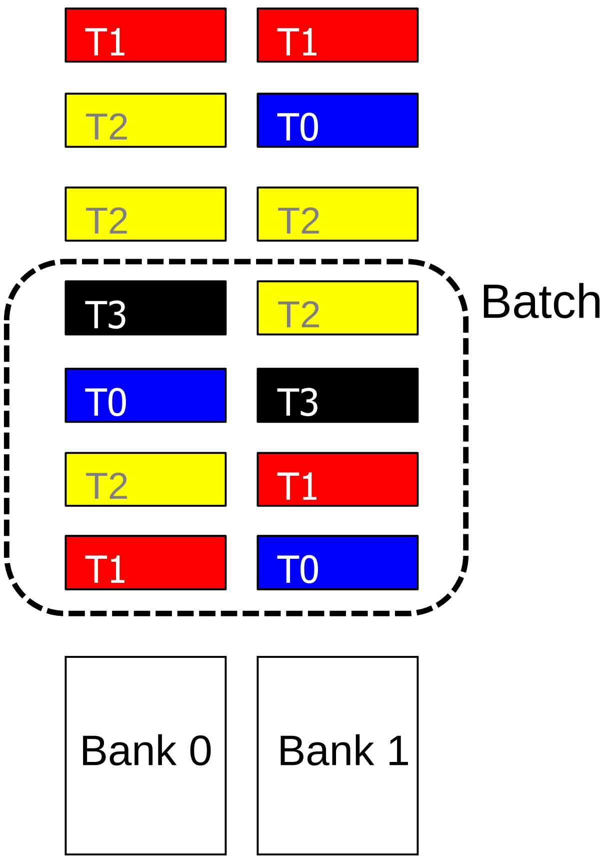
Thread A: Bank 1, Row 1

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?

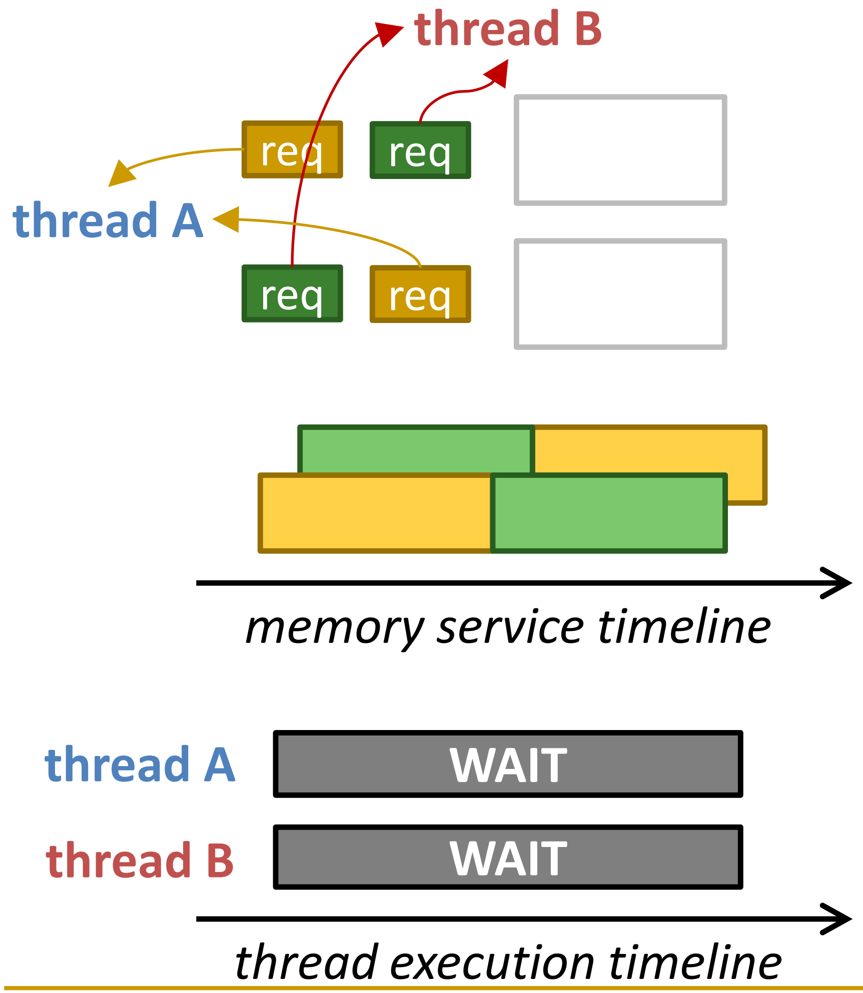
Within-Batch Scheduling

- 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

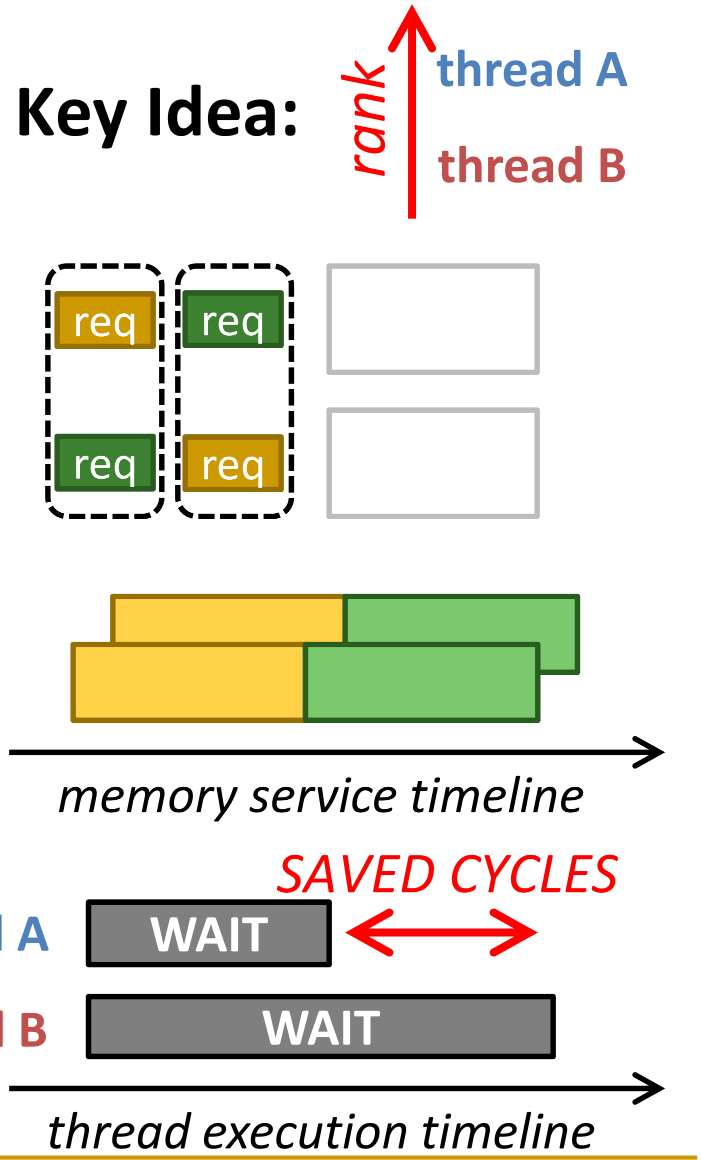
HOW?

- 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

Thread Ranking



Key Idea:



How to Rank Threads within a Batch

- Ranking scheme affects system throughput and fairness

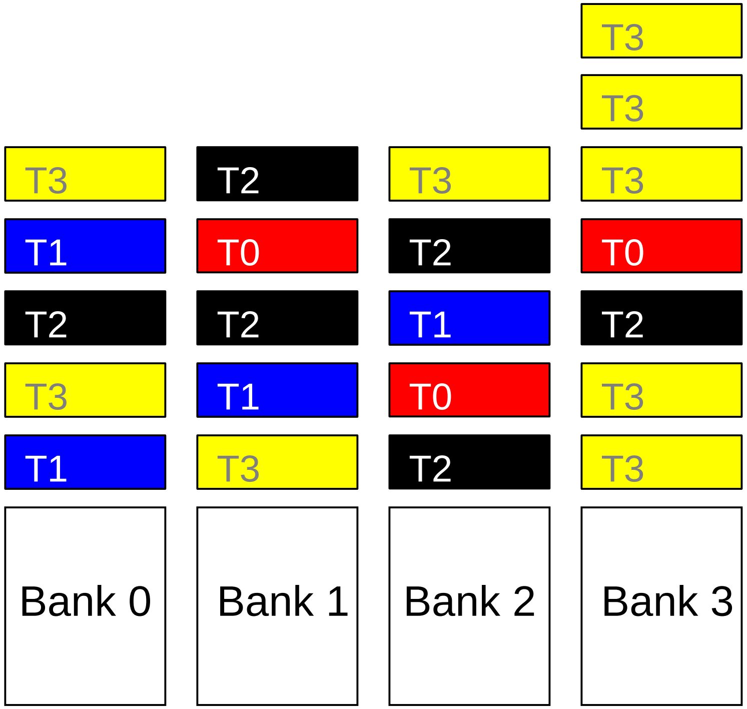
- Maximize system throughput
  - Minimize average stall-time of threads within the batch
- Minimize unfairness (Equalize the slowdown of threads)
  - Service threads with inherently low stall-time early in the batch
  - Insight: delaying memory non-intensive threads results in high slowdown

- 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

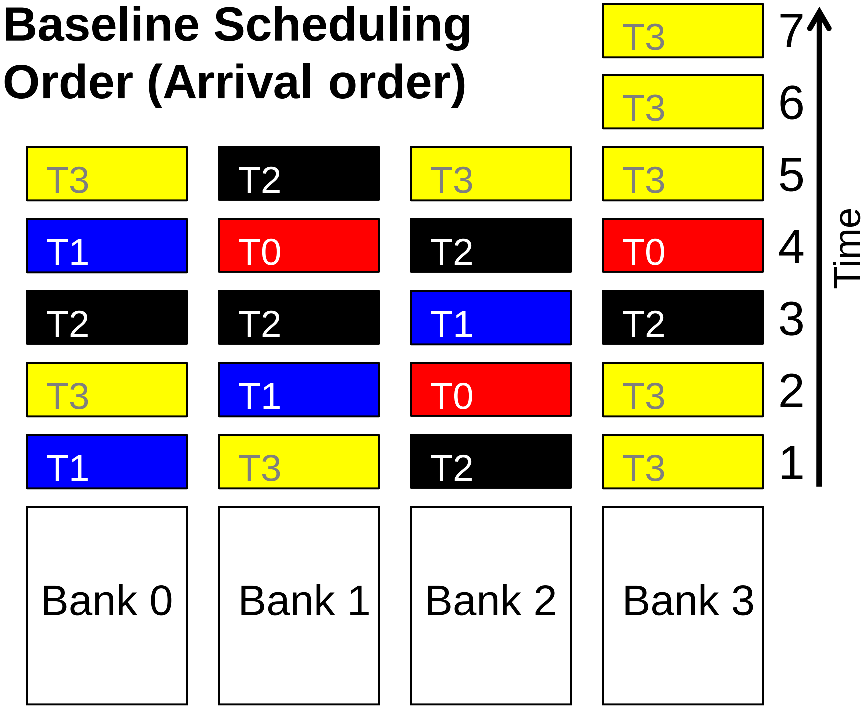


	max-bank-load	total-load

Ranking:
T0 > T1 > T2 > T3

Example Within-Batch Scheduling Order

Baseline Scheduling Order (Arrival order)

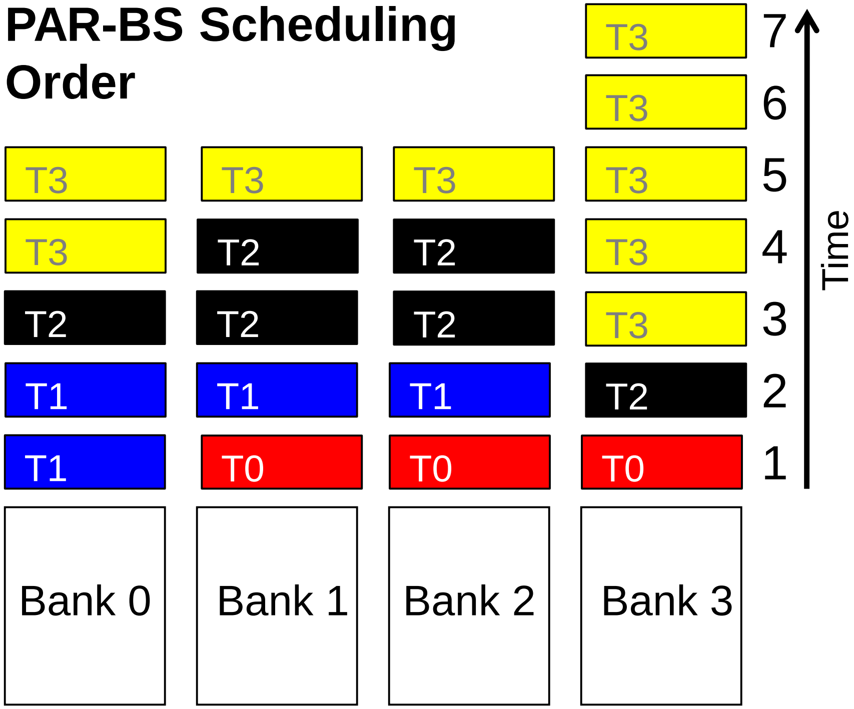


Time	Bank 0	Bank 1	Bank 2	Bank 3
7				T3
6				T3
5	T3	T2	T3	T3
4	T1	T0	T2	T0
3	T2	T2	T1	T2
2	T3	T1	T0	T3
1	T1	T3	T2	T3

	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

  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

Hardware Cost

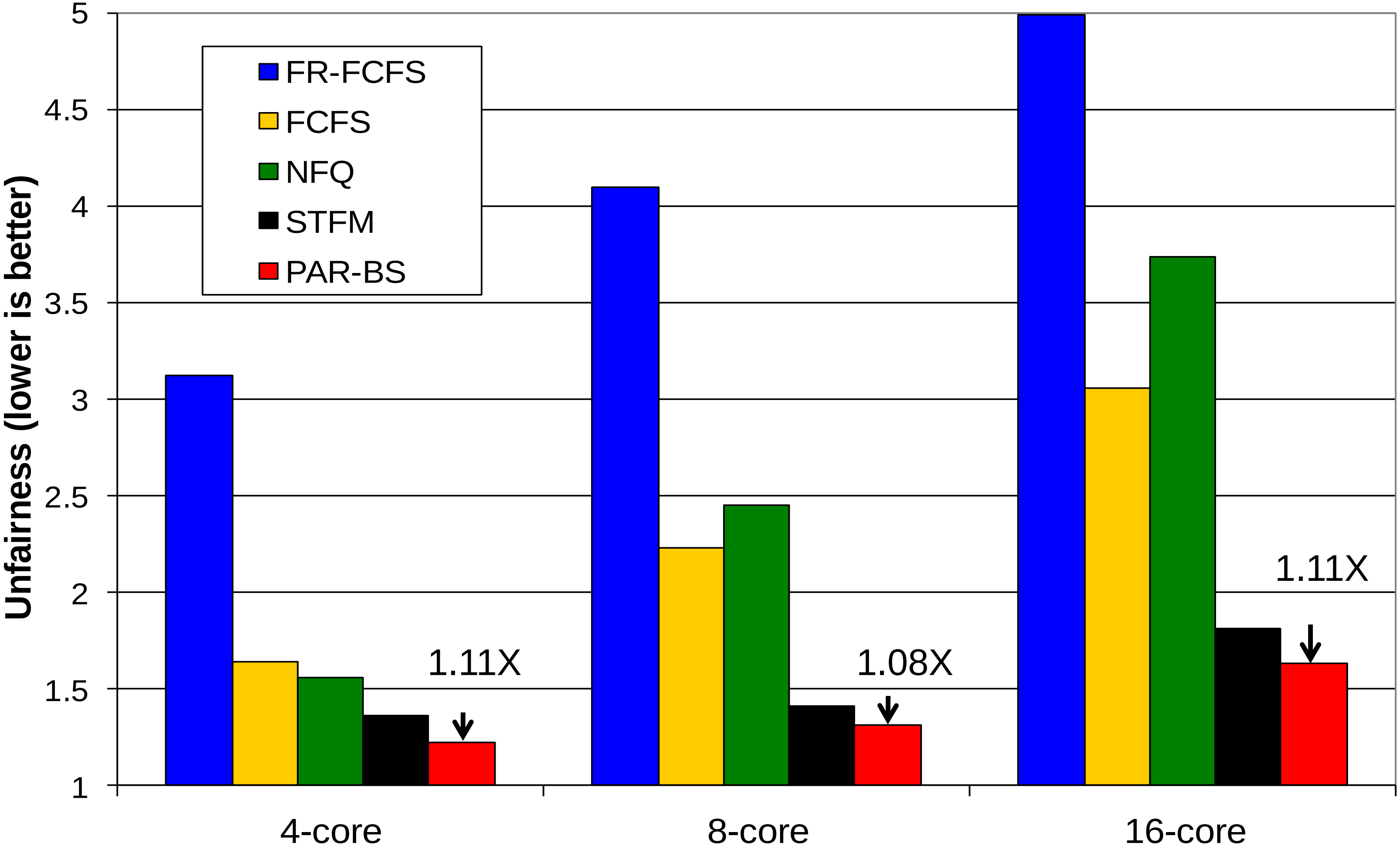
- <1.5KB storage cost for
  - 8-core system with 128-entry memory request buffer

- No complex operations (e.g., divisions)

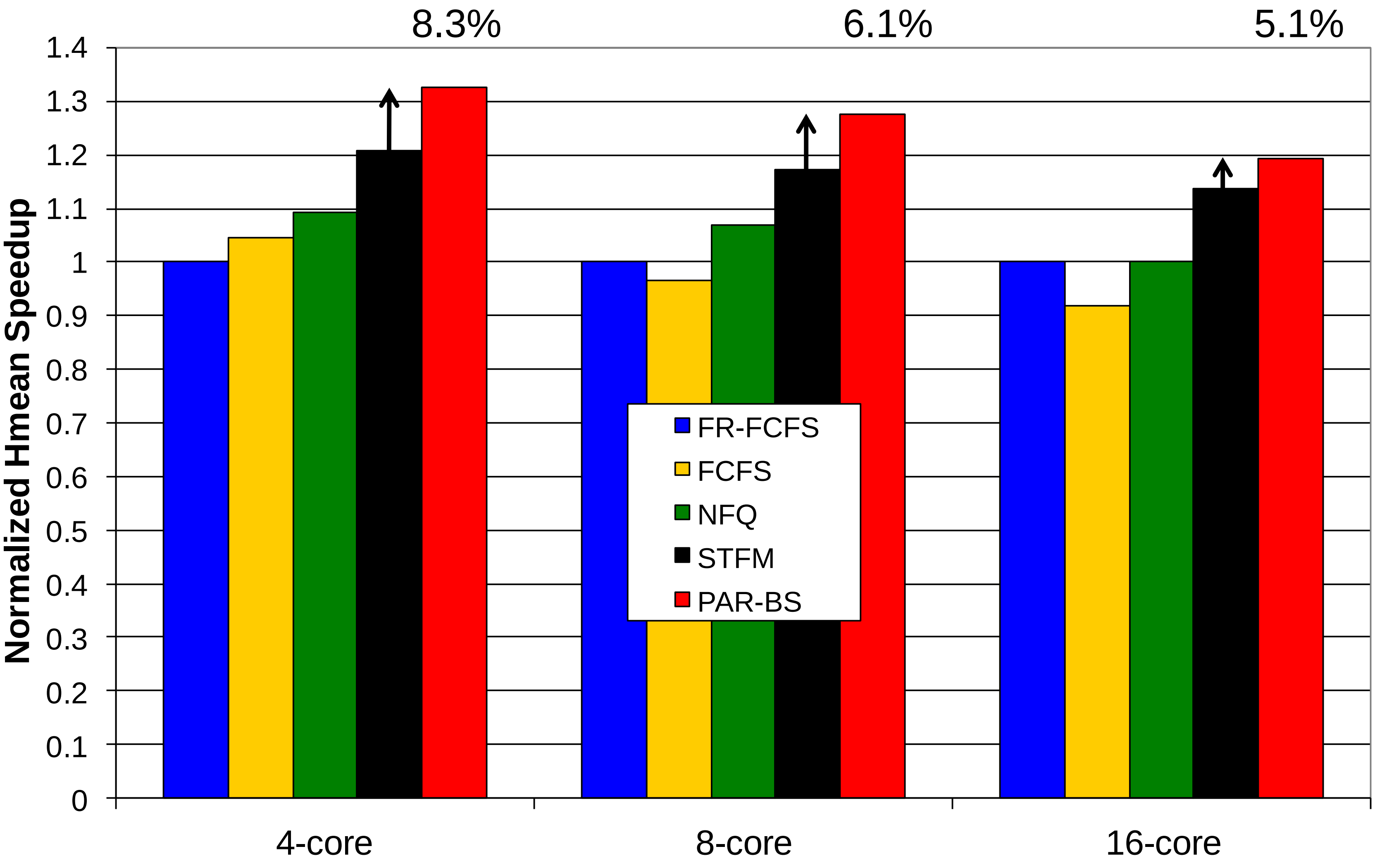
- Not on the critical path
  - Scheduler makes a decision only every DRAM cycle

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

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



System Performance (Hmean-speedup)



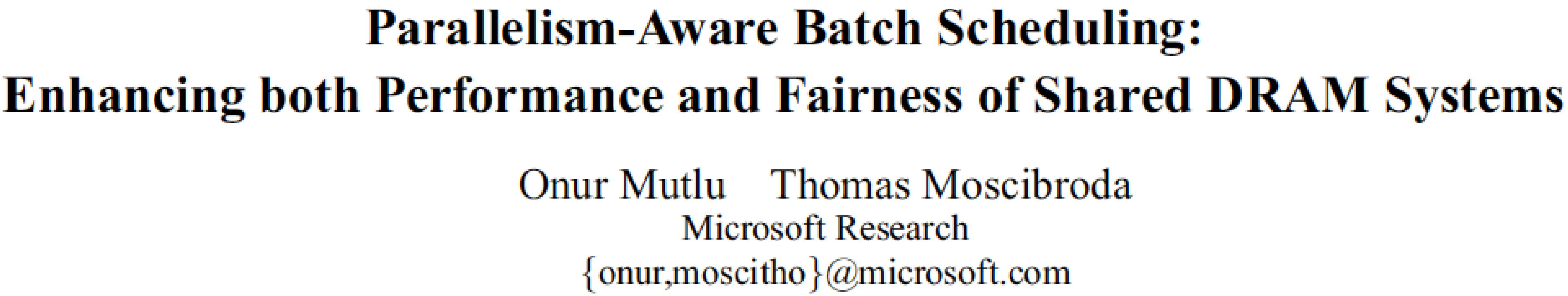
PAR-BS Pros and Cons

- Upsides:
  - 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

More on PAR-BS

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



Parallelism-Aware Batch Scheduling:
Enhancing both Performance and Fairness of Shared DRAM Systems

Onur Mutlu Thomas Moscibroda
Microsoft Research
{onur,moscitho}@microsoft.com

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

ATLAS: Summary

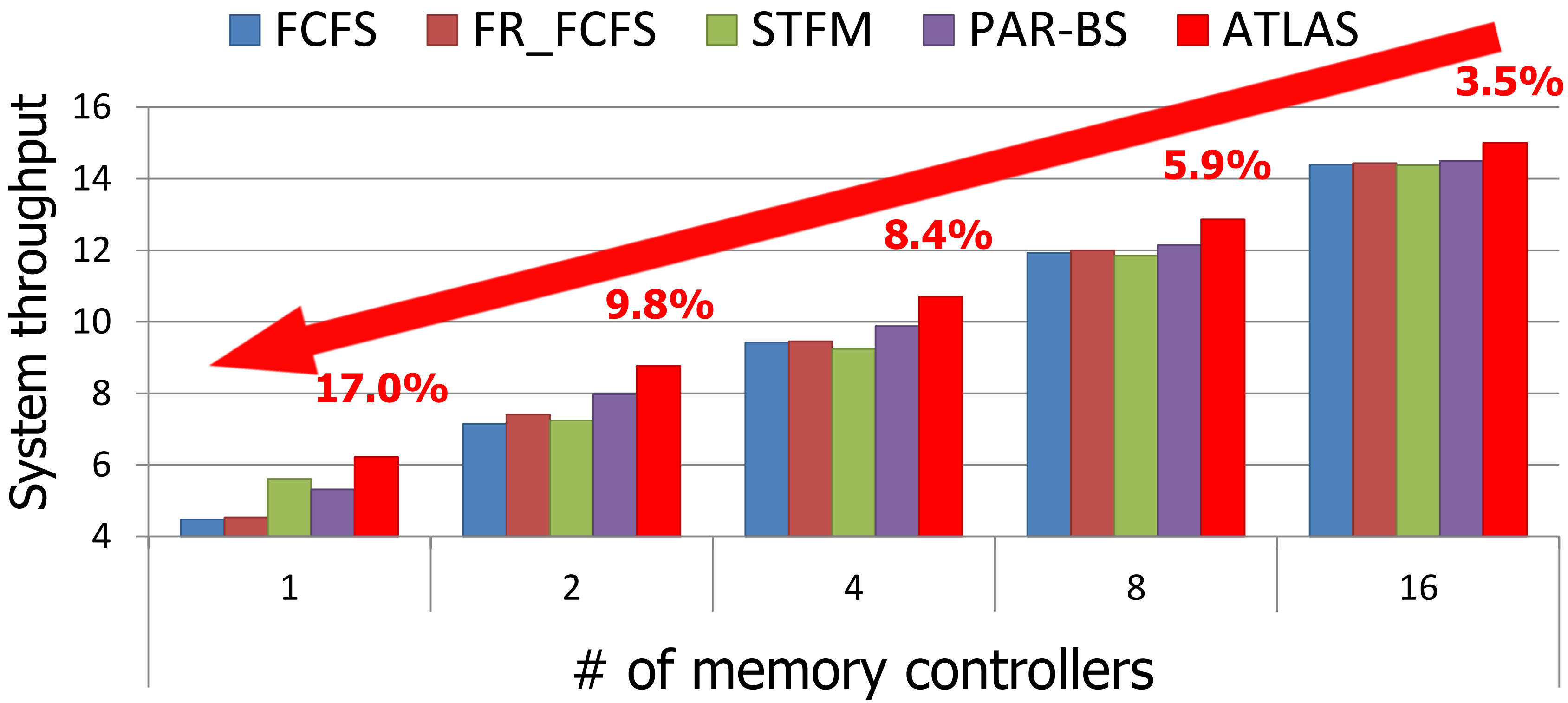
- Goal: To maximize system performance

- Main idea: Prioritize the thread that has attained the least service from the memory controllers (Adaptive per-Thread Least Attained Service Scheduling)
  - Rank threads based on attained service in the past time interval(s)
  - Enforce thread ranking in the memory scheduler during the current interval

- Why it works: Prioritizes “light” (memory non-intensive) threads that are more likely to keep their cores busy

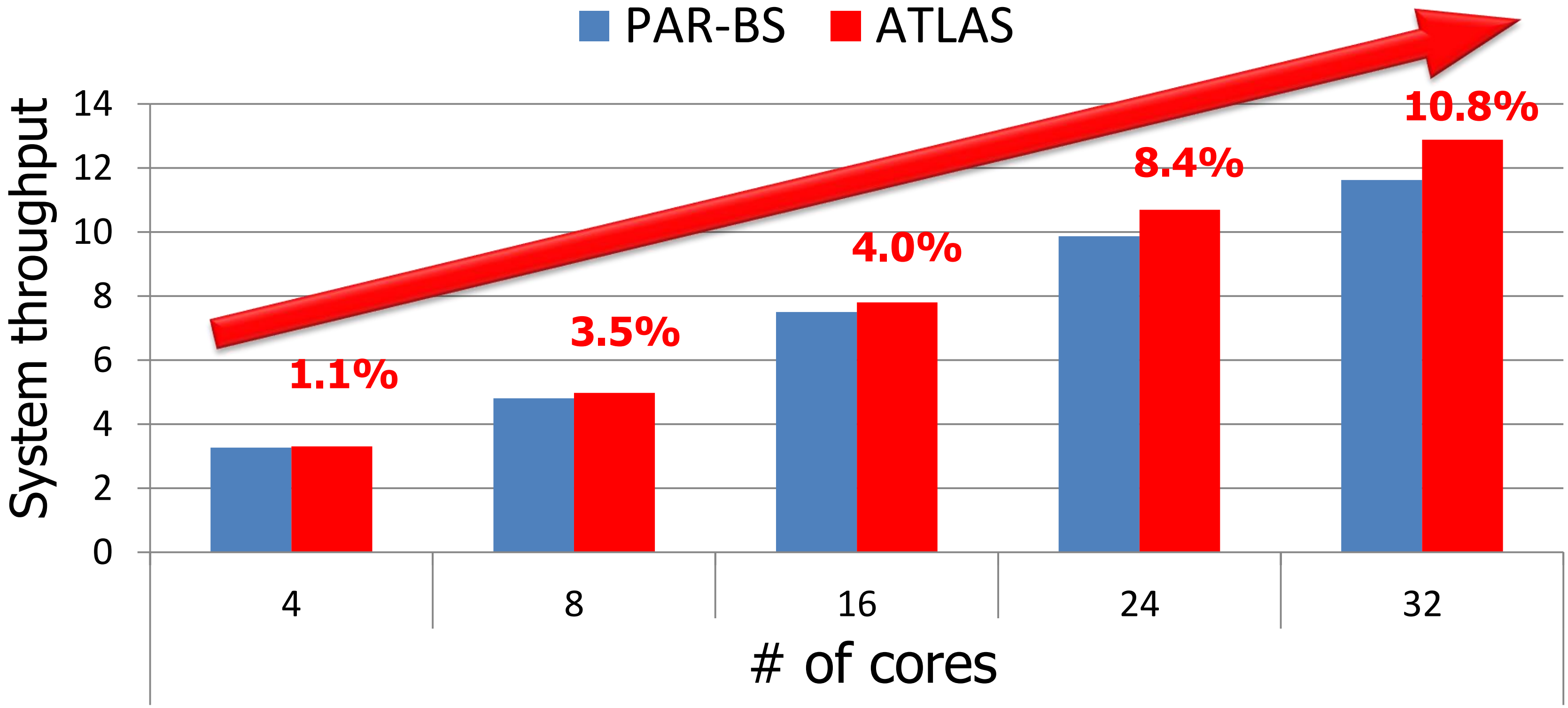
System Throughput: 24-Core System

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

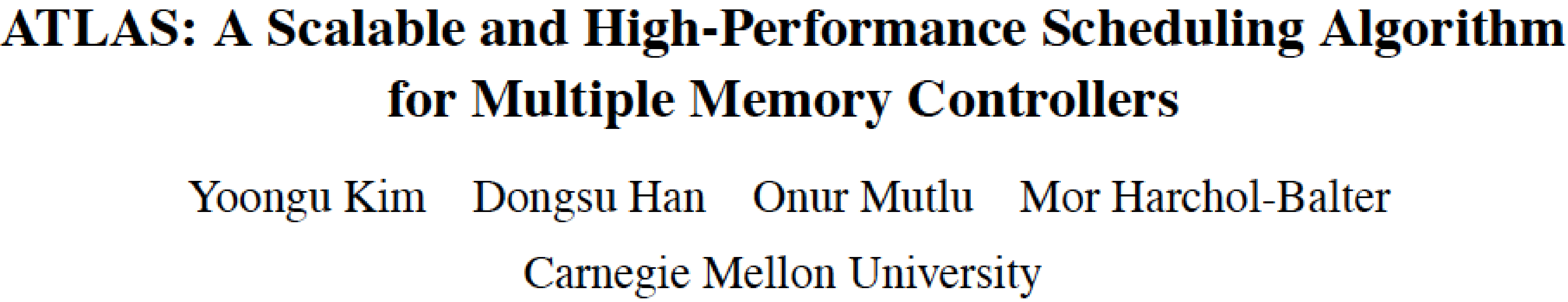
ATLAS Pros and Cons

- 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

More on 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"** *Proceedings of the 16th International Symposium on High-Performance Computer Architecture* (***HPCA***), Bangalore, India, January 2010. Slides (pptx)



ATLAS: A Scalable and High-Performance Scheduling Algorithm for Multiple Memory Controllers

Yoongu Kim   Dongsu Han   Onur Mutlu   Mor Harchol-Balter

Carnegie Mellon University

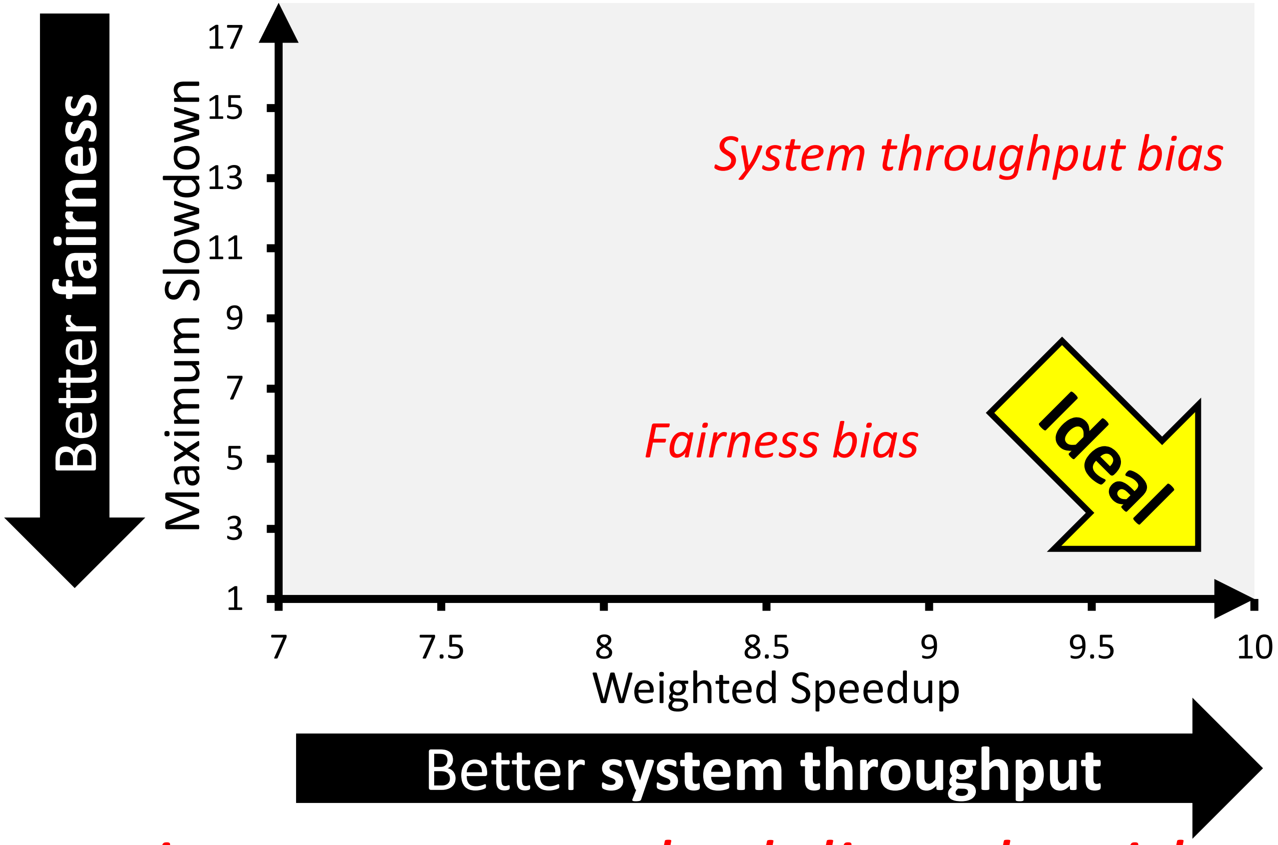
TCM:
Thread Cluster Memory Scheduling

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

Previous Scheduling Algorithms are Biased

24 cores, 4 memory controllers, 96 workloads



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

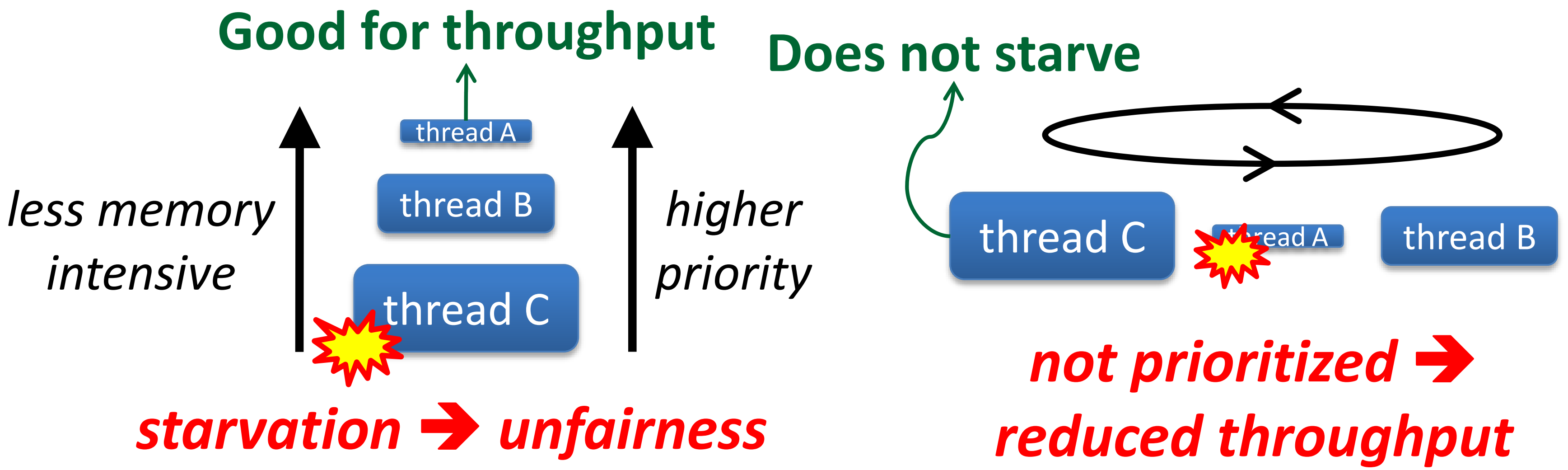
Throughput vs. Fairness

***Throughput biased** approach*

Prioritize less memory-intensive threads

***Fairness biased** approach*

Take turns accessing memory



Good for throughput

less memory intensive

thread A

thread B

thread C

higher priority

starvation → unfairness

Does not starve

thread C

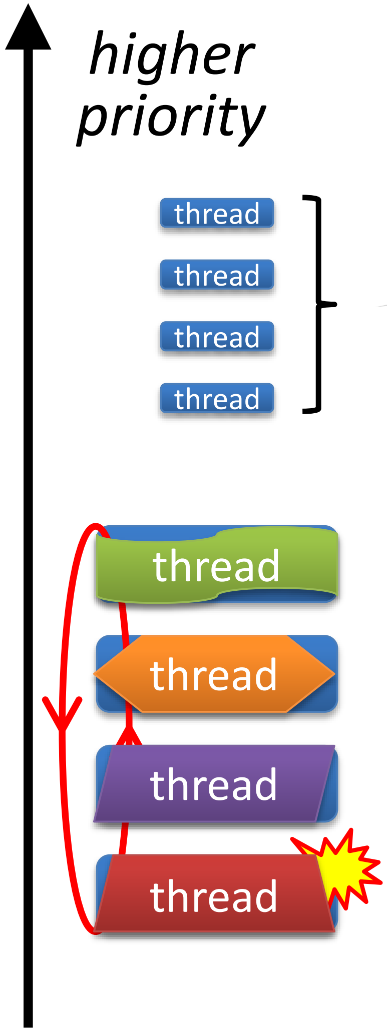
thread A

thread B

not prioritized → reduced throughput

Single policy for all threads is insufficient

Achieving the Best of Both Worlds



For Throughput

Prioritize memory-non-intensive threads

For Fairness

Unfairness caused by memory-intensive being prioritized over each other

- Shuffle thread ranking

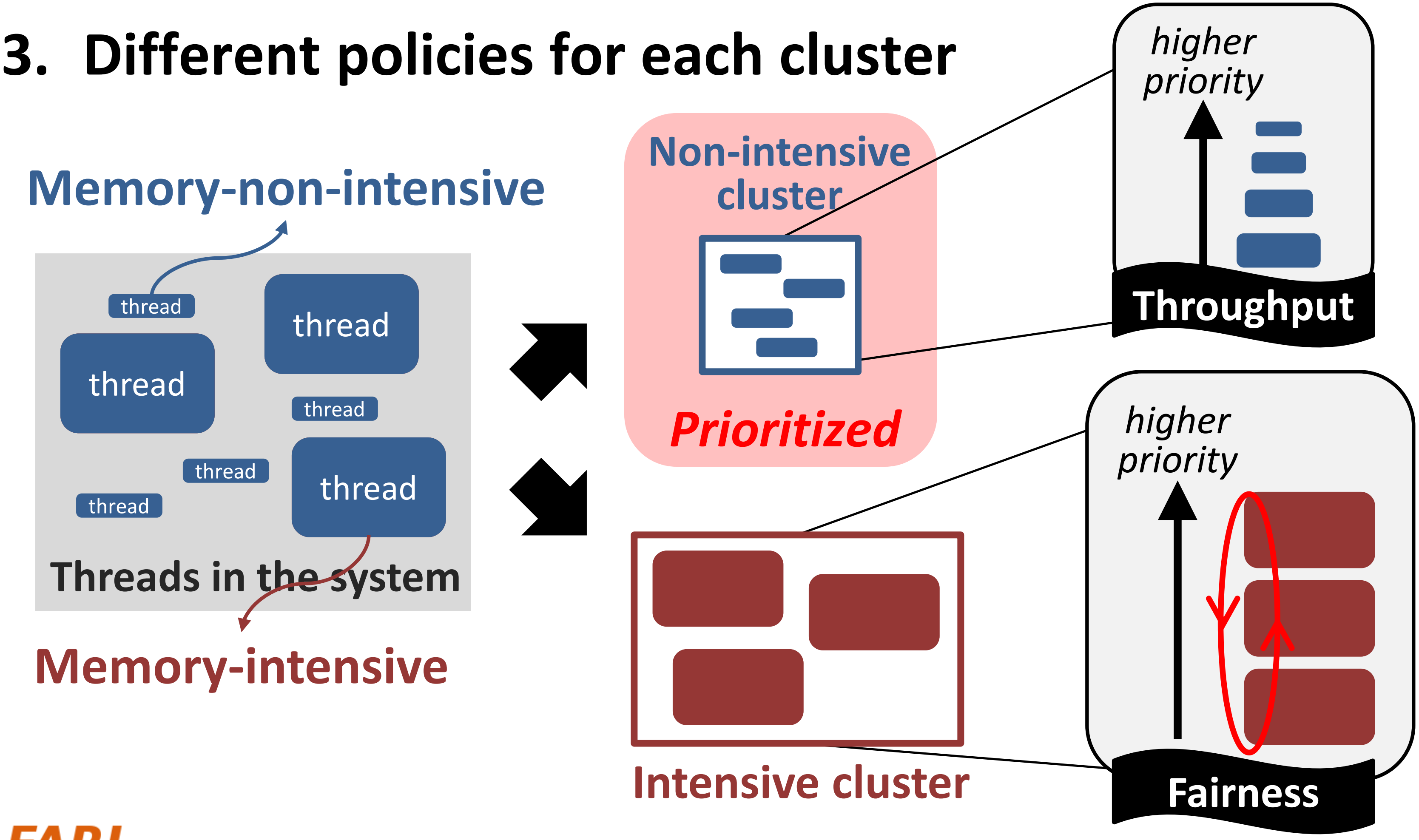
Memory-intensive threads have different vulnerability to interference

- Shuffle asymmetrically

SAFARI

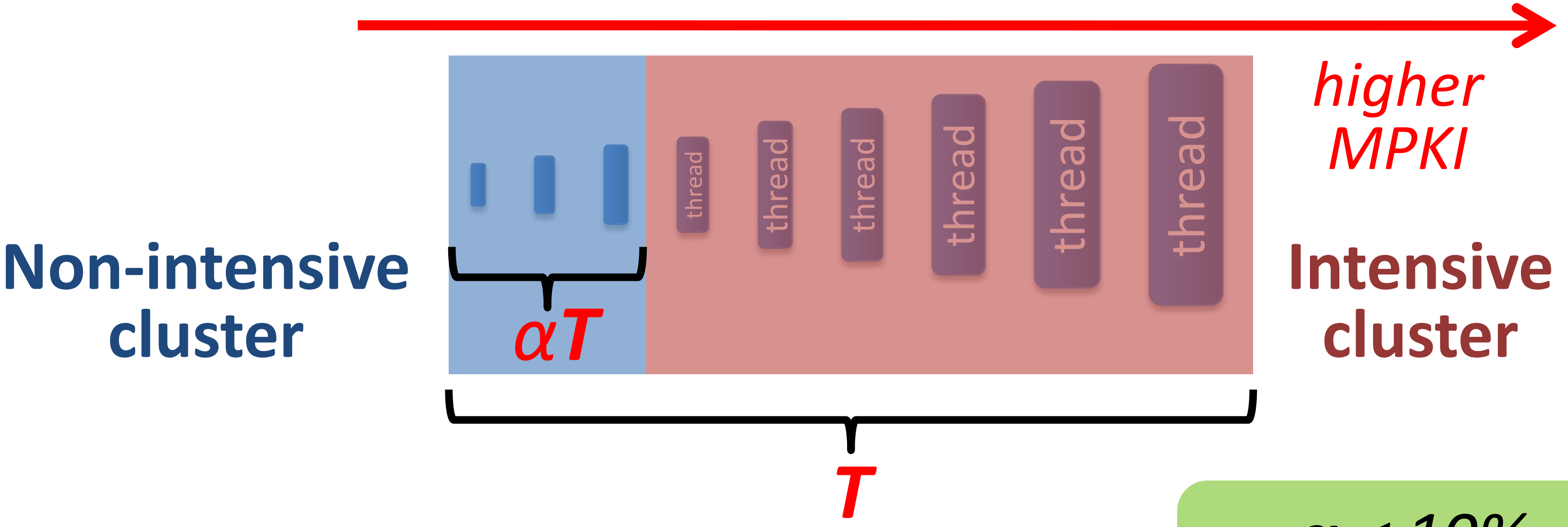
Thread Cluster Memory Scheduling [Kim+ MICRO'10]

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)



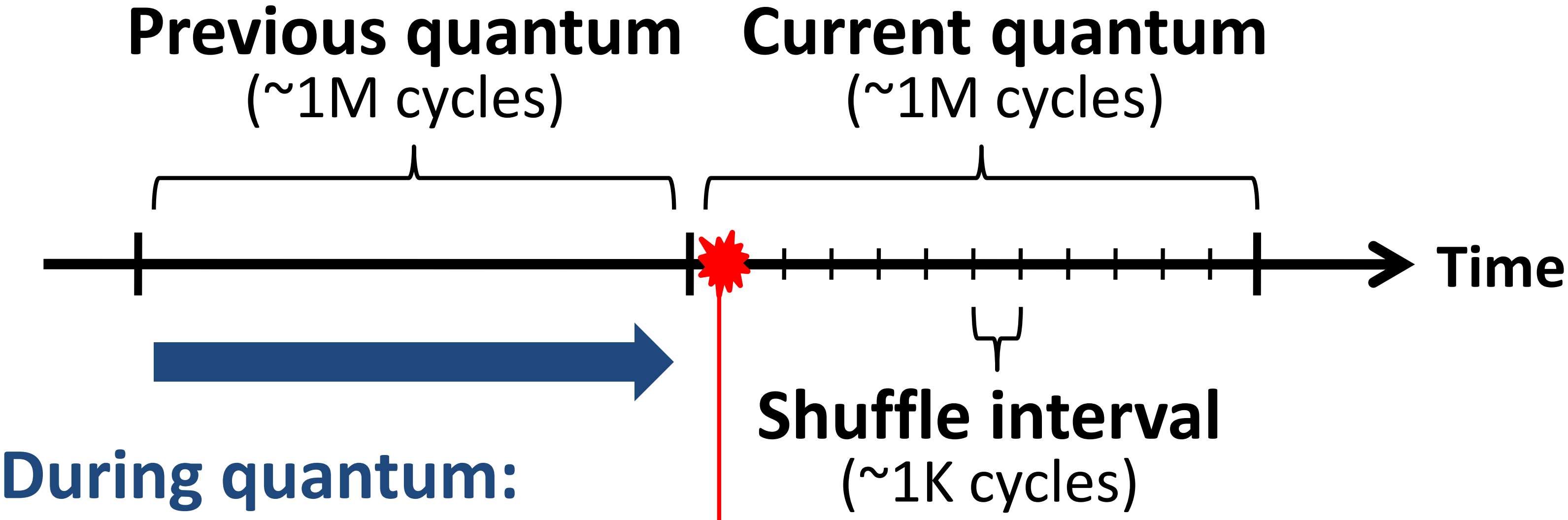
T **=** Total ***memory bandwidth usage***

***α** < 10%*
ClusterThreshold

Step2 Memory bandwidth usage ***αT*** divides clusters

SAFARI

TCM: Quantum-Based Operation



Previous quantum
(~1M cycles)

Current quantum
(~1M cycles)

Time

Shuffle interval
(~1K cycles)

During quantum:

- Monitor thread behavior
  1. Memory intensity
  2. Bank-level parallelism
  3. Row-buffer locality

Beginning of quantum:

- Perform clustering
- Compute niceness of intensive threads

TCM: Scheduling Algorithm

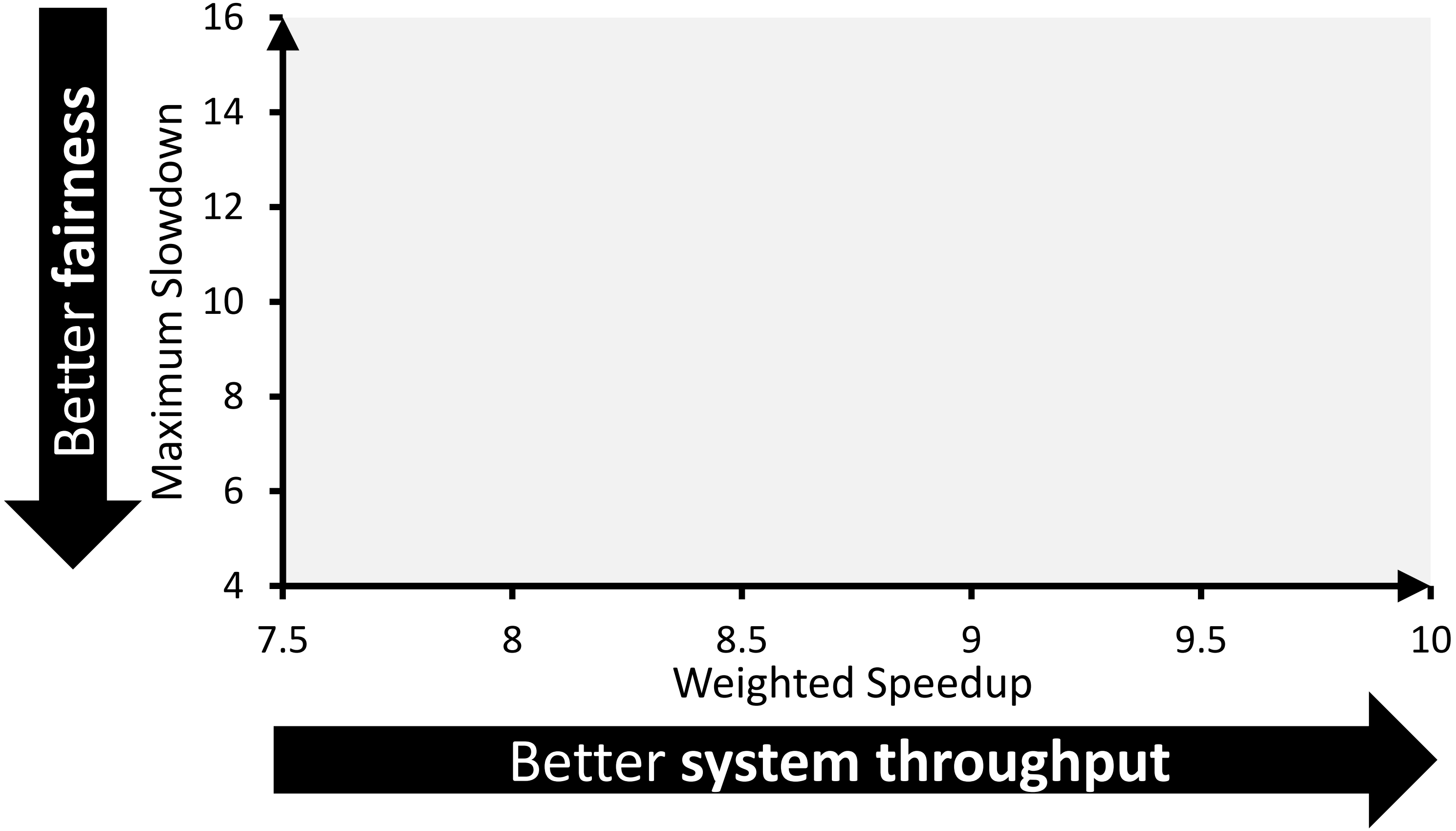
1. ***Highest-rank***: Requests from higher ranked threads prioritized
   - **Non-Intensive** cluster > **Intensive** cluster
   - **Non-Intensive** cluster: lower intensity → higher rank
   - **Intensive** cluster: rank shuffling

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



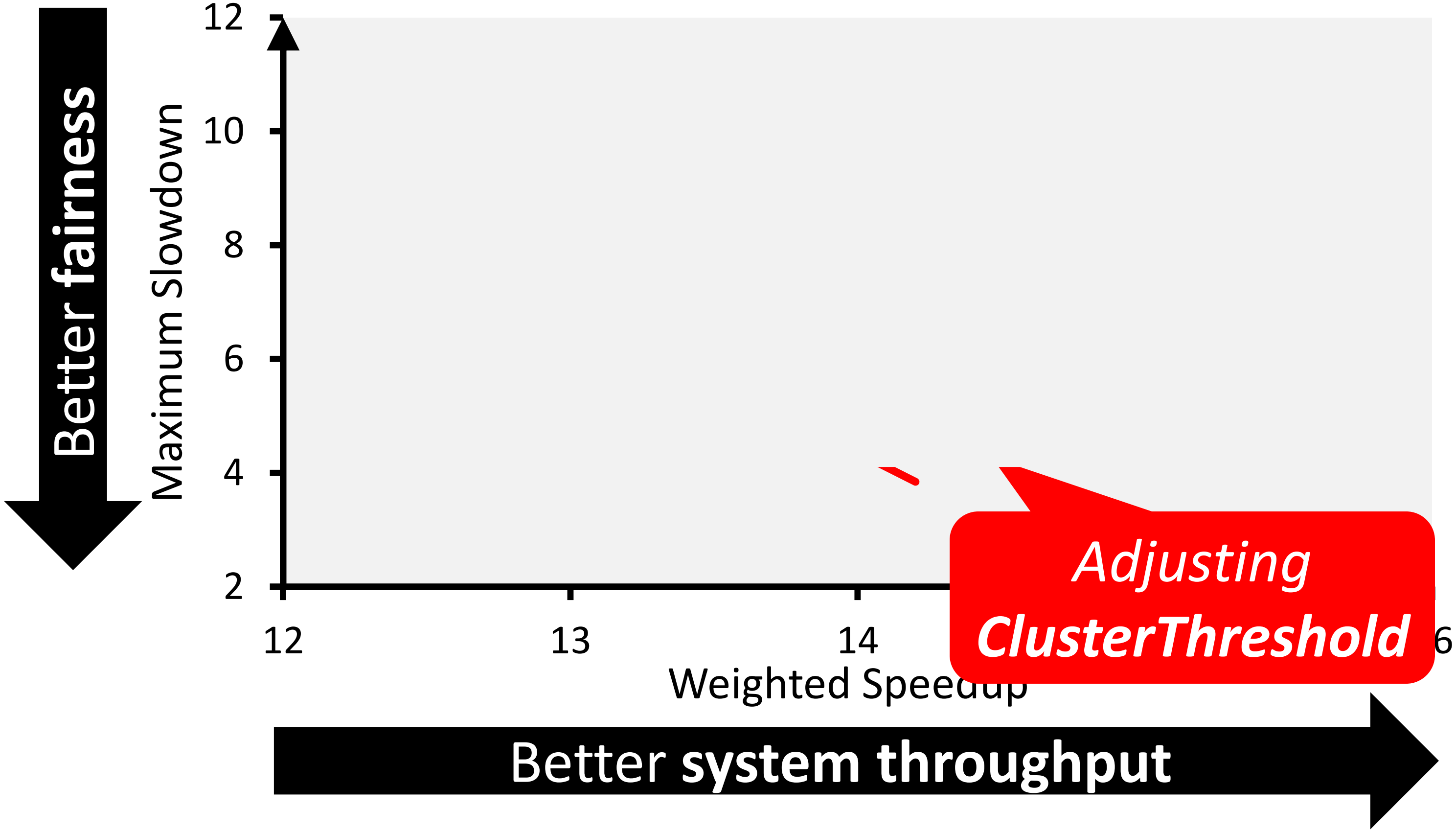
Better **fairness**

Better **system throughput**

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

TCM: Fairness-Throughput Tradeoff

When configuration parameter is varied…



Adjusting ***ClusterThreshold***

TCM allows robust fairness-throughput tradeoff

Operating System Support

- ***ClusterThreshold*** is a tunable knob
  - OS can trade off between fairness and throughput

- Enforcing thread weights
  - OS assigns weights to threads
  - TCM enforces thread weights within each cluster

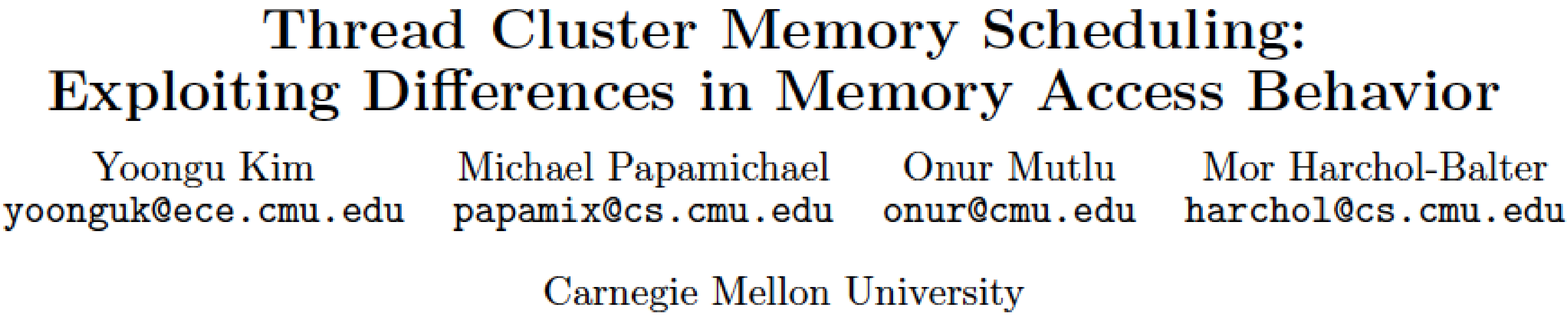
TCM Pros and Cons

- 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?
  - Ranking is still too complex?

More on TCM

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



Thread Cluster Memory Scheduling: Exploiting Differences in Memory Access Behavior

Yoongu Kim yoonguk@ece.cmu.edu
Michael Papamichael papamix@cs.cmu.edu
Onur Mutlu onur@cmu.edu
Mor Harchol-Balter harchol@cs.cmu.edu

Carnegie Mellon University

Handling Memory Interference In Multithreaded Applications with Memory Scheduling

Eiman Ebrahimi, Rustam Miftakhutdinov, Chris Fallin,
Chang Joo Lee, Onur Mutlu, and Yale N. Patt,
"Parallel Application Memory Scheduling"
Proceedings of the 44th International Symposium on Microarchitecture (***MICRO***),
Porto Alegre, Brazil, December 2011. Slides (pptx)

Multithreaded (Parallel) Applications

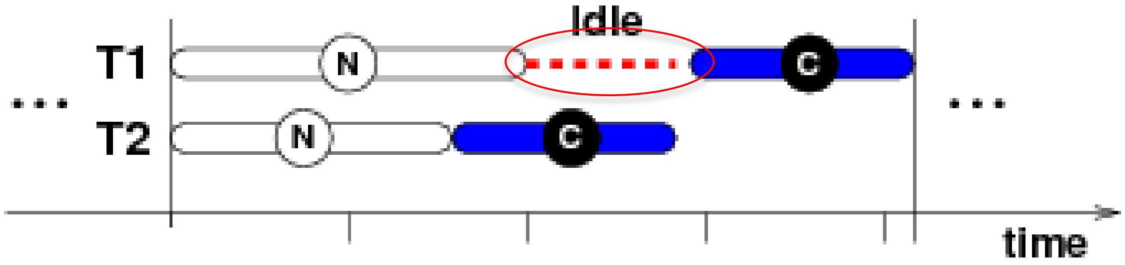
- Threads in a multi-threaded application can be inter-dependent
  - As opposed to threads from different applications
- Such threads can synchronize with each other
  - Locks, barriers, pipeline stages, condition variables, semaphores, …
- Some threads can be on the critical path of execution due to synchronization; some threads are not
- Even within a thread, some "code segments" may be on the critical path of execution; some are not

Critical Sections

- Enforce mutually exclusive access to shared data
- Only one thread can be executing it at a time
- Contended critical sections make threads wait → threads causing serialization can be on the critical path

Each thread:

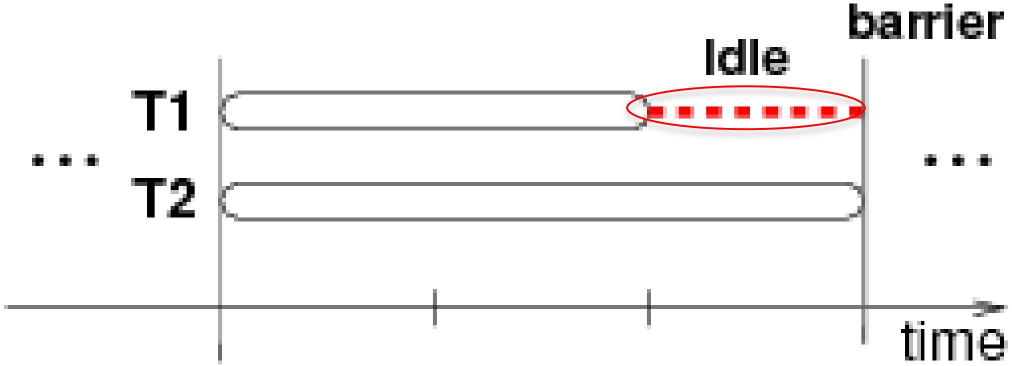
```
loop {
  Compute                   N
  lock(A)
     Update shared data
  unlock(A)                 C
}
```



Barriers

- Synchronization point
- Threads have to wait until all threads reach the barrier
- Last thread arriving to the barrier is on the critical path

```
Each thread:
  loop1 {
    Compute
  }
  barrier
  loop2 {
    Compute
  }
```



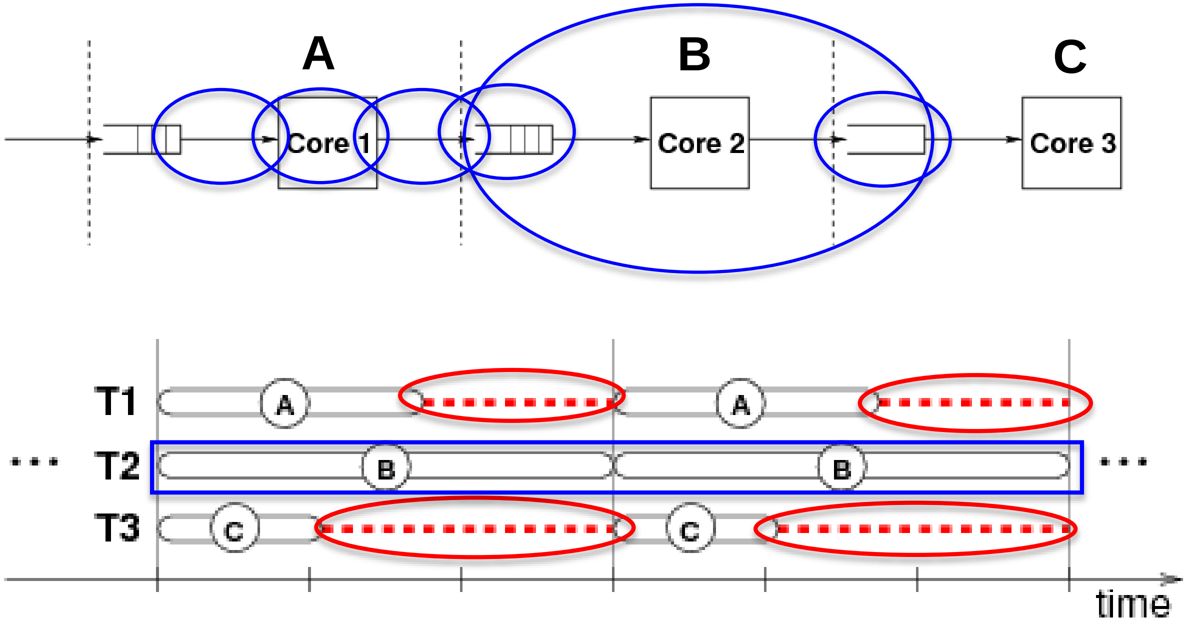
Stages of Pipelined Programs

- Loop iterations are statically divided into code segments called *stages*
- Threads execute stages on different cores
- Thread executing the slowest stage is on the critical path

```
loop {
  Compute1    A

  Compute2    B

  Compute3    C
}
```



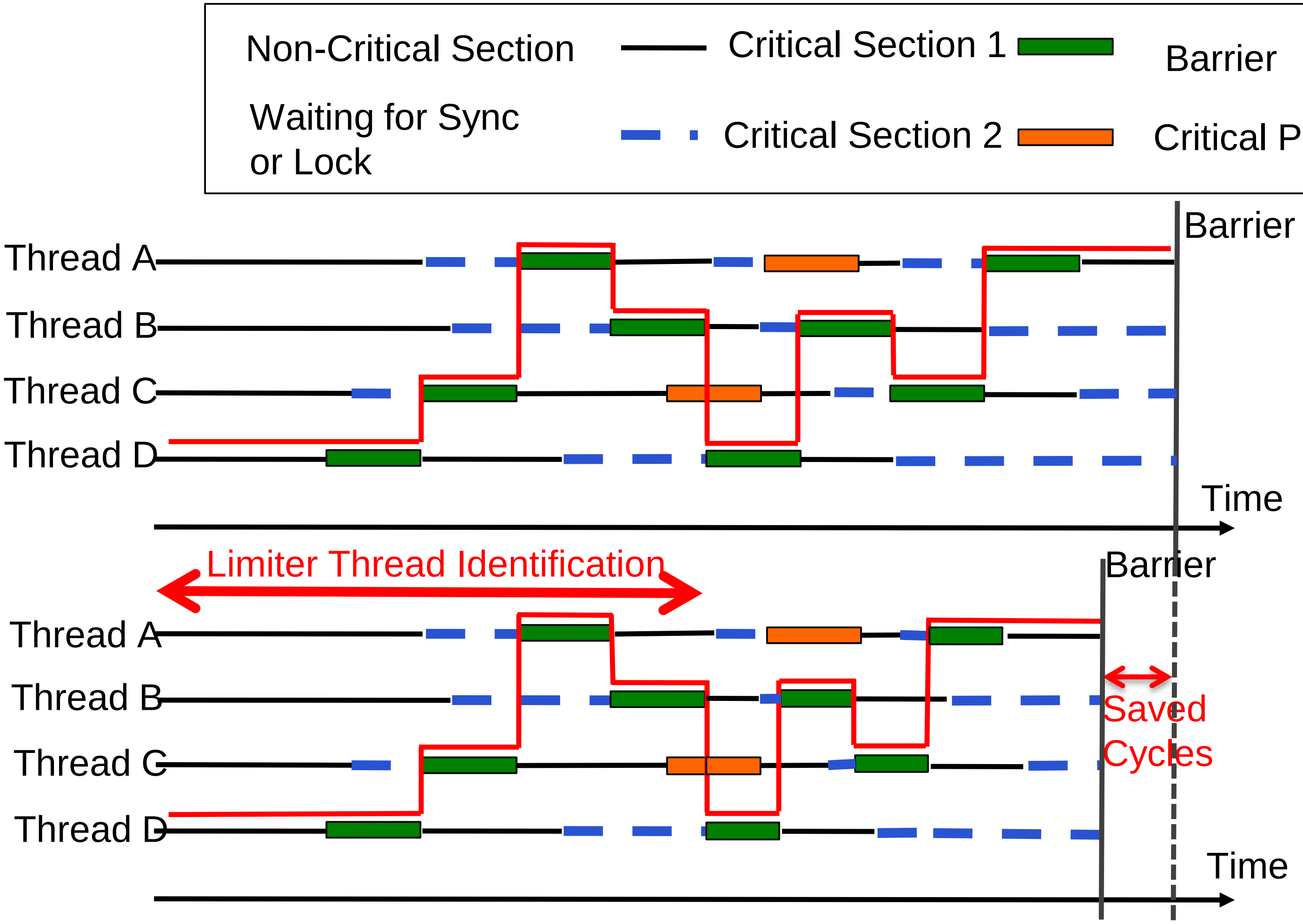
Handling Interference in Parallel Applications

- Threads in a multithreaded application are inter-dependent
- Some threads can be on the critical path of execution due to synchronization; some threads are not
- How do we schedule requests of inter-dependent threads to maximize multithreaded application performance?

- Idea: Estimate limiter threads likely to be on the critical path and prioritize their requests; shuffle priorities of non-limiter threads to reduce memory interference among them [Ebrahimi+, MICRO'11]

- Hardware/software cooperative limiter thread estimation:
  - Thread executing the most contended critical section
  - Thread executing the slowest pipeline stage
  - Thread that is falling behind the most in reaching a barrier

Prioritizing Requests from Limiter Threads

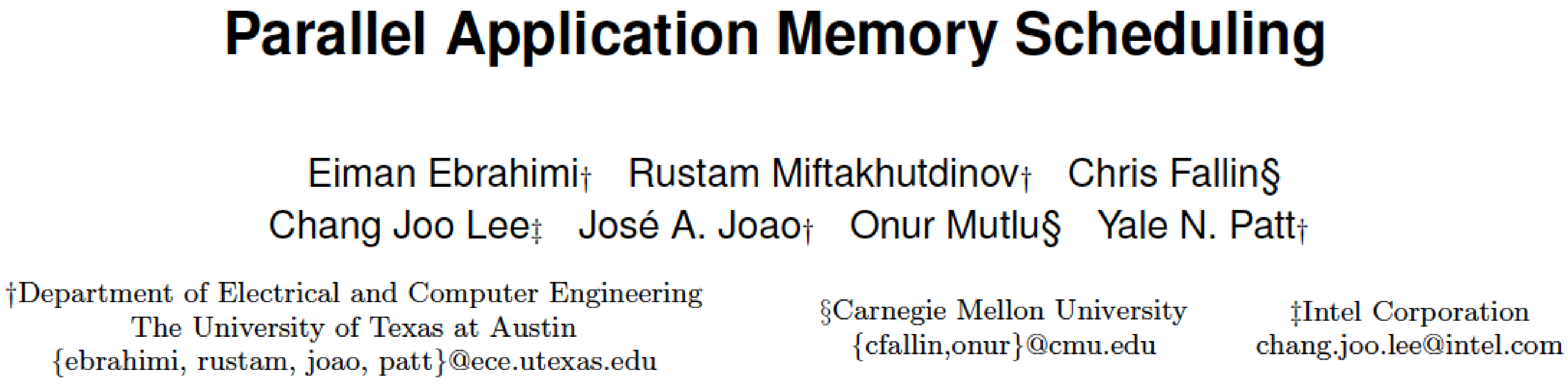


Most Contended Critical Section: 1 🟩

Limiter Thread: D

More on PAMS

- Eiman Ebrahimi, Rustam Miftakhutdinov, Chris Fallin, Chang Joo Lee, Onur Mutlu, and Yale N. Patt, **"Parallel Application Memory Scheduling"** *Proceedings of the 44th International Symposium on Microarchitecture* (***MICRO***), Porto Alegre, Brazil, December 2011. Slides (pptx)



Parallel Application Memory Scheduling

Eiman Ebrahimi† Rustam Miftakhutdinov† Chris Fallin§
Chang Joo Lee‡ José A. Joao† Onur Mutlu§ Yale N. Patt†

†Department of Electrical and Computer Engineering
The University of Texas at Austin
{ebrahimi, rustam, joao, patt}@ece.utexas.edu

§Carnegie Mellon University
{cfallin,onur}@cmu.edu

‡Intel Corporation
chang.joo.lee@intel.com

Other Ways of Handling Memory Interference

Fundamental Interference Control Techniques

- Goal: to reduce/control inter-thread memory interference

1. Prioritization or request scheduling

2. Data mapping to banks/channels/ranks

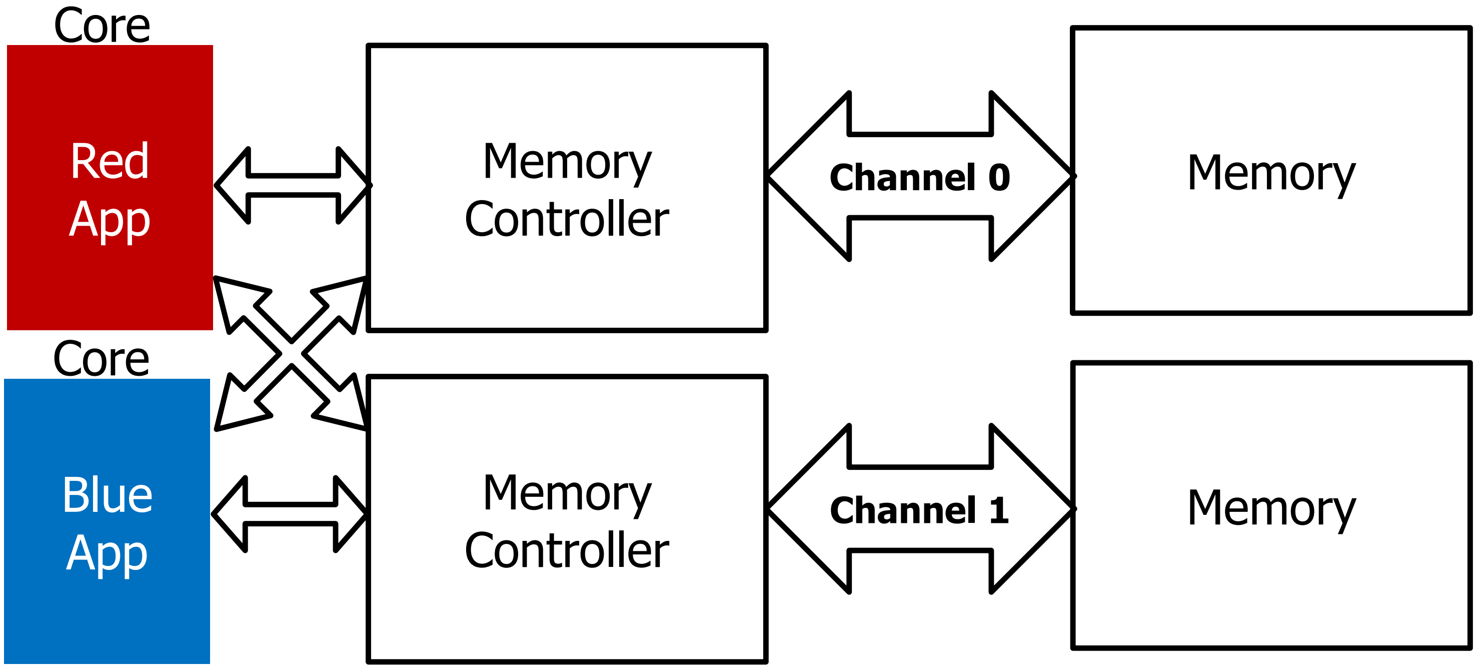
3. Core/source throttling

4. Application/thread scheduling

Memory Channel Partitioning

Sai Prashanth Muralidhara, Lavanya Subramanian, Onur Mutlu, Mahmut Kandemir, and Thomas Moscibroda,
"Reducing Memory Interference in Multicore Systems via Application-Aware Memory Channel Partitioning"
44th International Symposium on Microarchitecture (***MICRO***),
Porto Alegre, Brazil, December 2011. Slides (pptx)

Observation: Modern Systems Have Multiple Channels

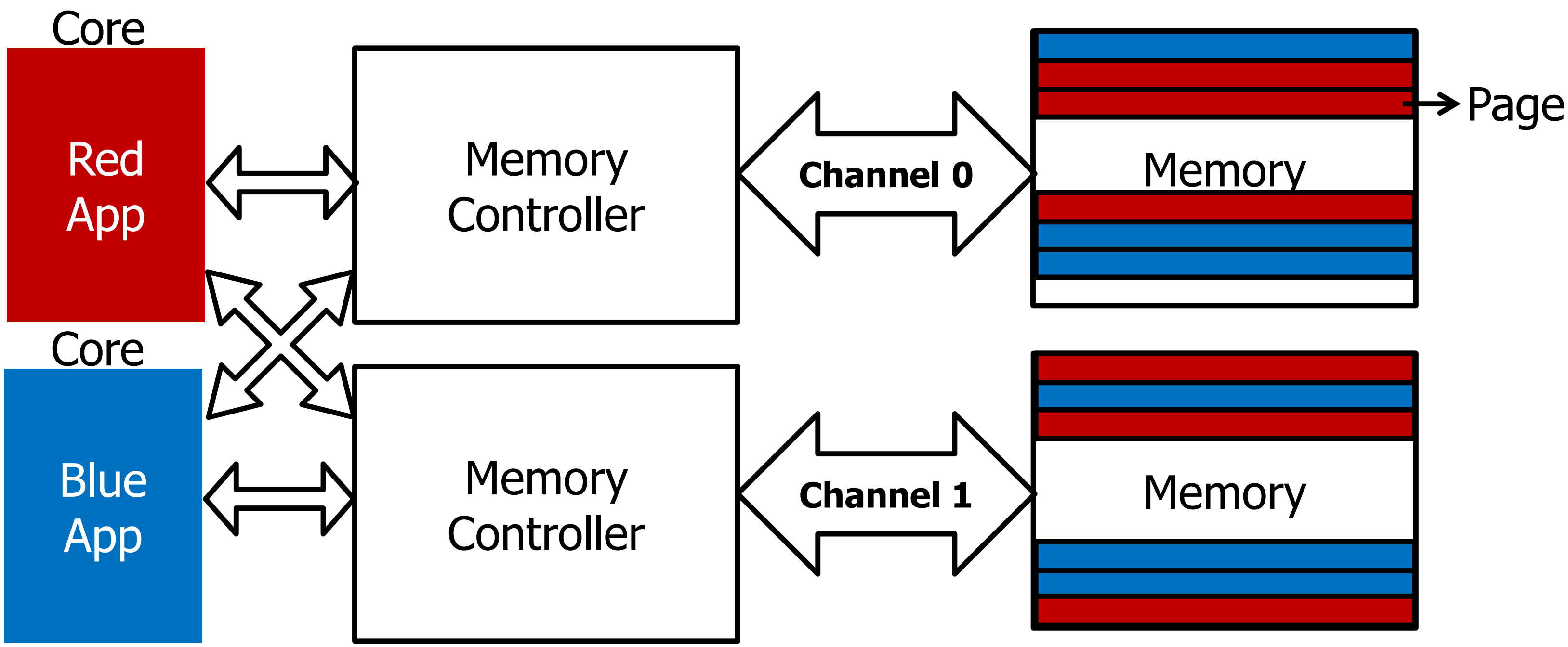


A new degree of freedom

Mapping data across multiple channels

Muralidhara et al., “Memory Channel Partitioning,” MICRO’11.

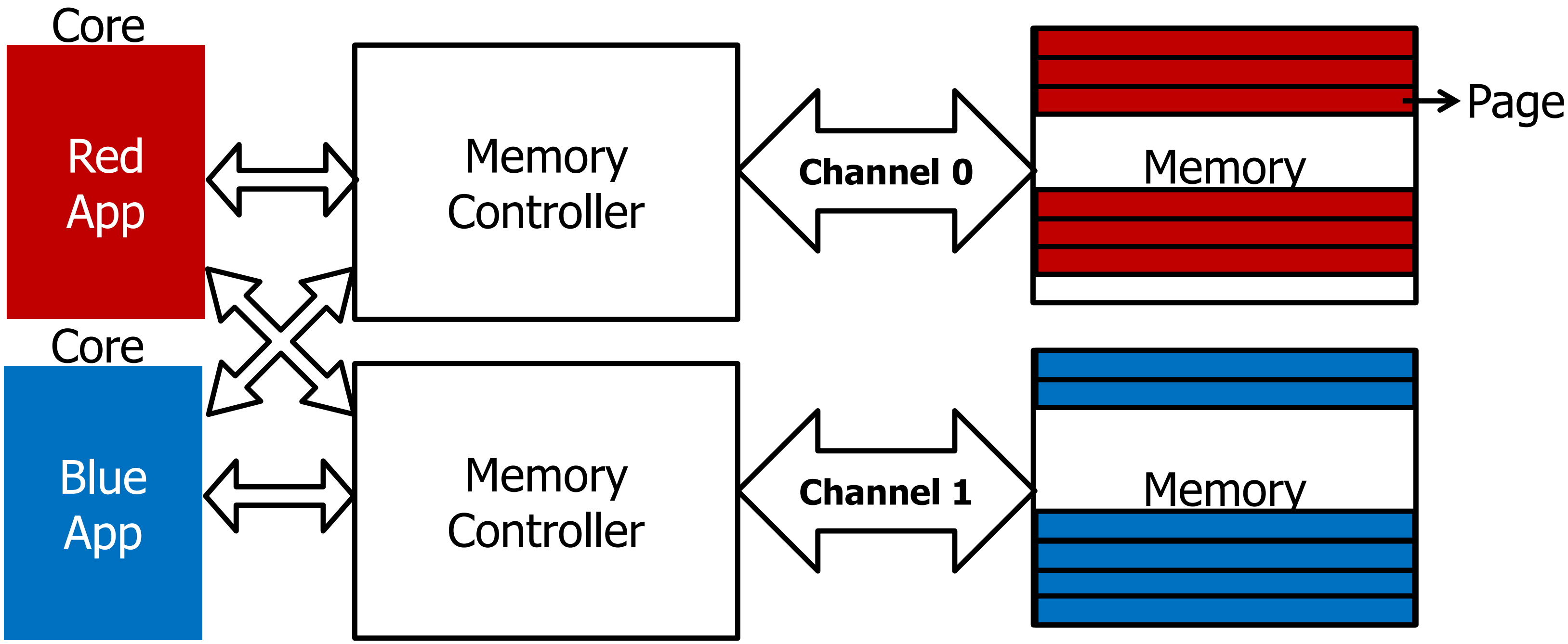
Data Mapping in Current Systems



Causes interference between applications' requests

Muralidhara et al., "Memory Channel Partitioning," MICRO'11.

Partitioning Channels Between Applications



Eliminates interference between applications' requests

Muralidhara et al., "Memory Channel Partitioning," MICRO'11.

Overview: Memory Channel Partitioning (MCP)

- Goal
  - Eliminate harmful interference between applications

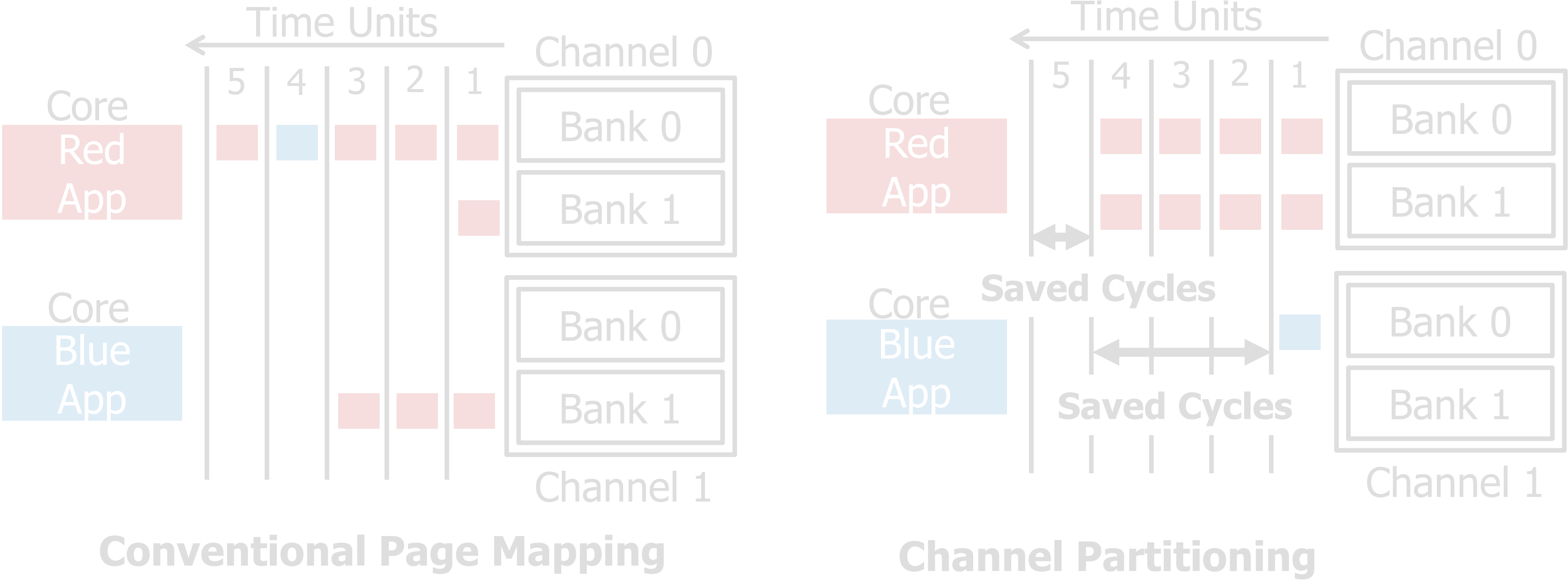
- Basic Idea
  - Map the data of badly-interfering applications to different channels

- Key Principles
  - Separate low and high memory-intensity applications
  - Separate low and high row-buffer locality applications

Muralidhara et al., "Memory Channel Partitioning," MICRO'11.

Key Insight 1: Separate by Memory Intensity

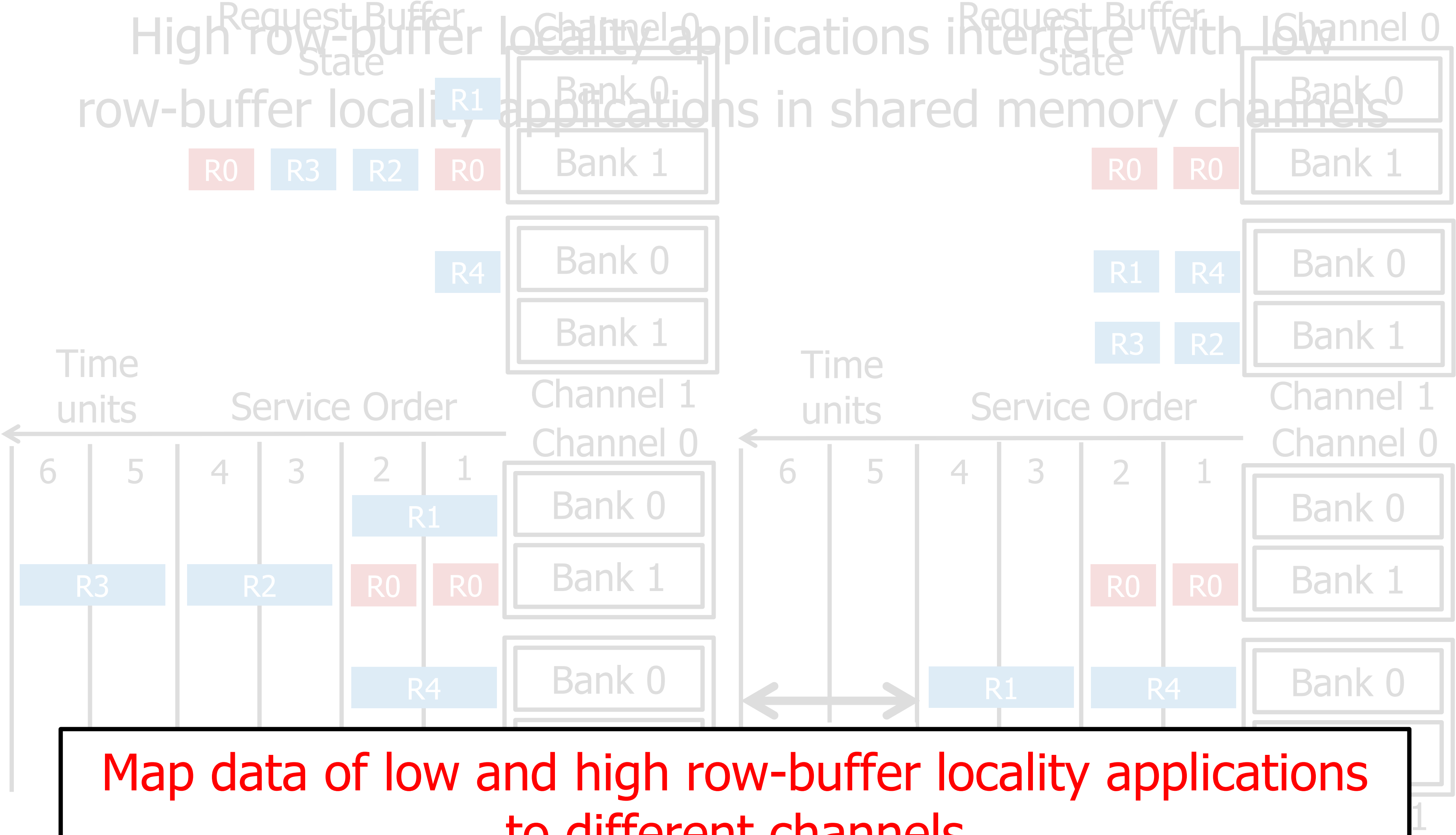
High memory-intensity applications interfere with low memory-intensity applications in shared memory channels



Map data of low and high memory-intensity applications to different channels

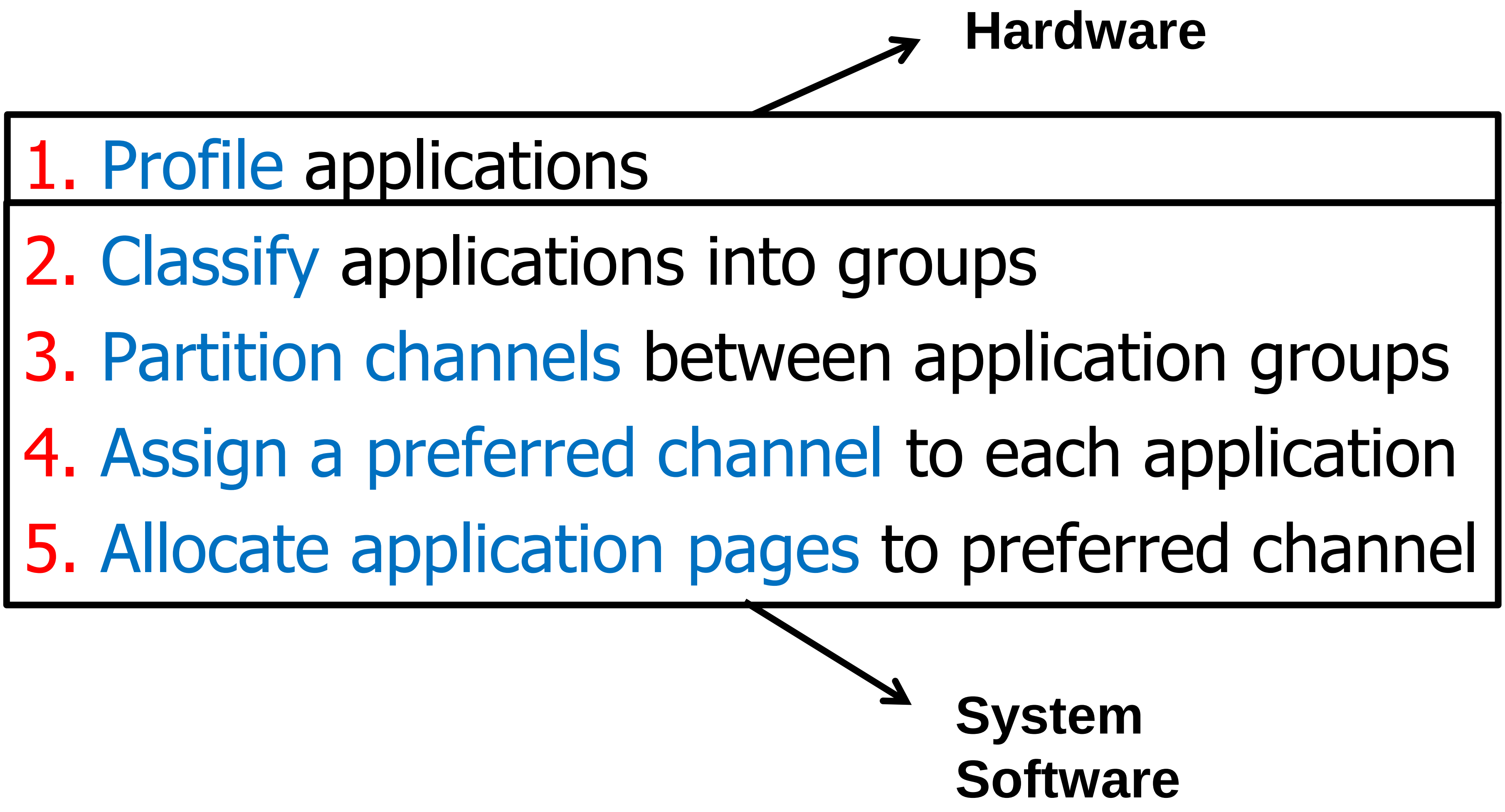
Key Insight 2: Separate by Row-Buffer Locality

High row-buffer locality applications interfere with low row-buffer locality applications in shared memory channels



Map data of low and high row-buffer locality applications to different channels

Memory Channel Partitioning (MCP) Mechanism



Hardware

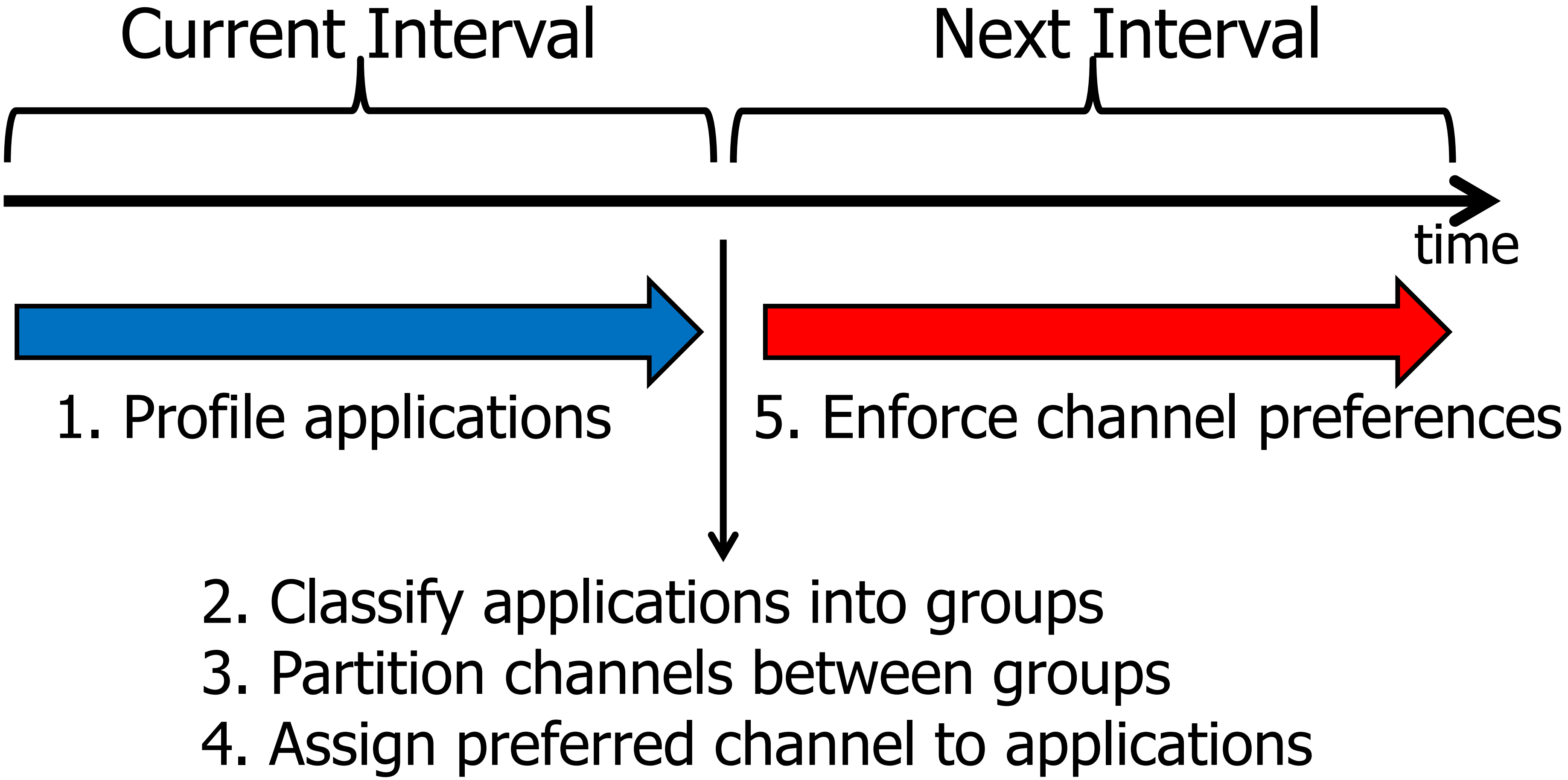
1. Profile applications

2. Classify applications into groups
3. Partition channels between application groups
4. Assign a preferred channel to each application
5. Allocate application pages to preferred channel

System Software

Muralidhara et al., “Memory Channel Partitioning,” MICRO’11.

Interval Based Operation



Current Interval

Next Interval

time

1. Profile applications

5. Enforce channel preferences

2. Classify applications into groups
3. Partition channels between groups
4. Assign preferred channel to applications

Observations

- Applications with very low memory-intensity rarely access memory
  → Dedicating channels to them results in precious memory bandwidth waste

- They have the most potential to keep their cores busy
  → We would really like to prioritize them

- They interfere minimally with other applications
  → Prioritizing them does not hurt others

Integrated Memory Partitioning and Scheduling (IMPS)

- Always prioritize very low memory-intensity applications in the memory scheduler

- Use memory channel partitioning to mitigate interference between other applications

Muralidhara et al., "Memory Channel Partitioning," MICRO'11.

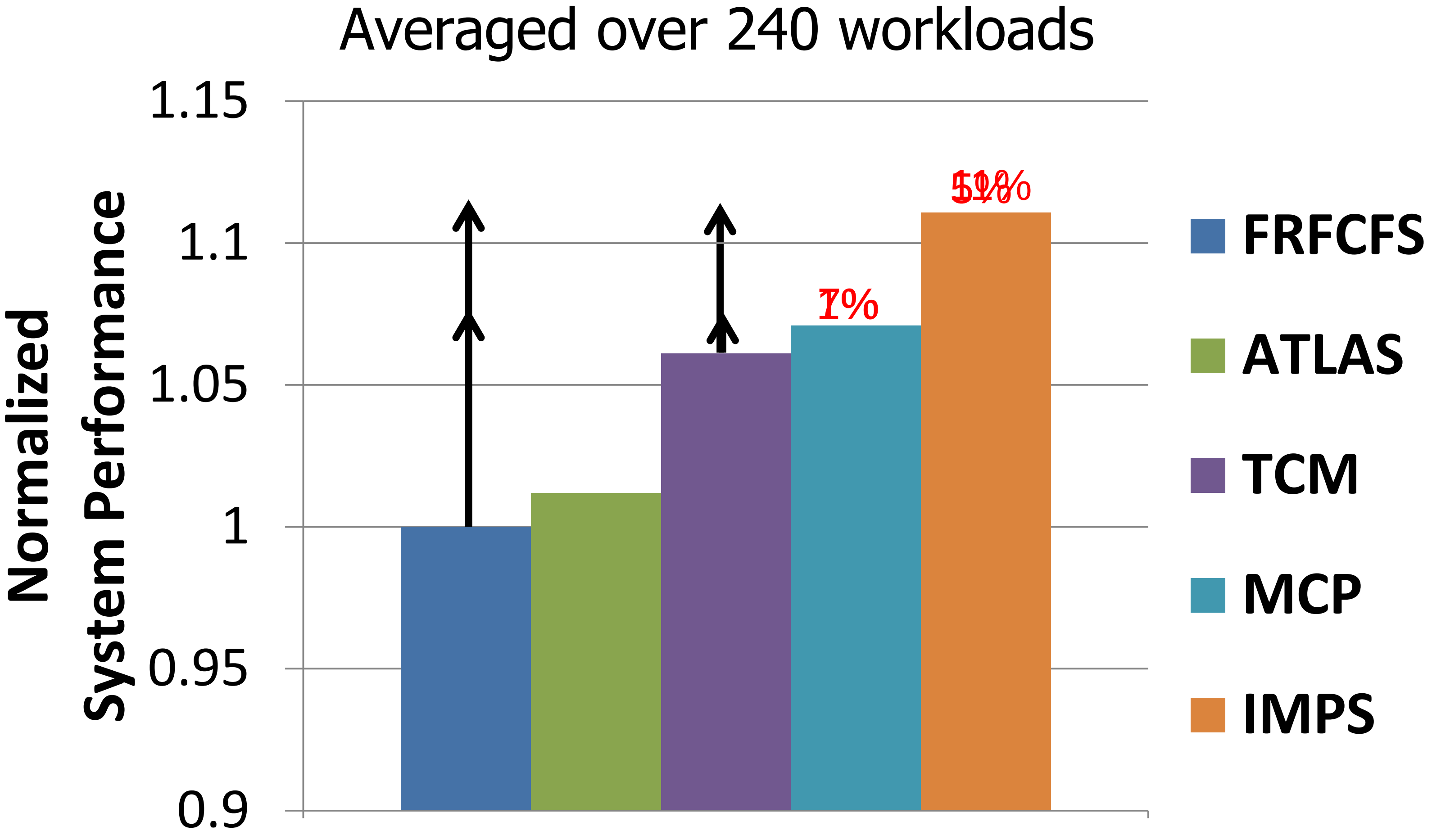
Hardware Cost

- Memory Channel Partitioning (MCP)
  - Only profiling counters in hardware
  - No modifications to memory scheduling logic
  - 1.5 KB storage cost for a 24-core, 4-channel system

- Integrated Memory Partitioning and Scheduling (IMPS)
  - A single bit per request
  - Scheduler prioritizes based on this single bit

Muralidhara et al., "Memory Channel Partitioning," MICRO'11.

Performance of Channel Partitioning



Better system performance than the best previous scheduler at lower hardware cost

Combining Multiple Interference Control Techniques

- Combined interference control techniques can mitigate interference much more than a single technique alone can do

- The key challenge is:
  - Deciding what technique to apply when
  - Partitioning work appropriately between software and hardware

Fundamental Interference Control Techniques

- Goal: to reduce/control inter-thread memory interference

1. Prioritization or request scheduling
2. Data mapping to banks/channels/ranks
3. Core/source throttling
4. Application/thread scheduling

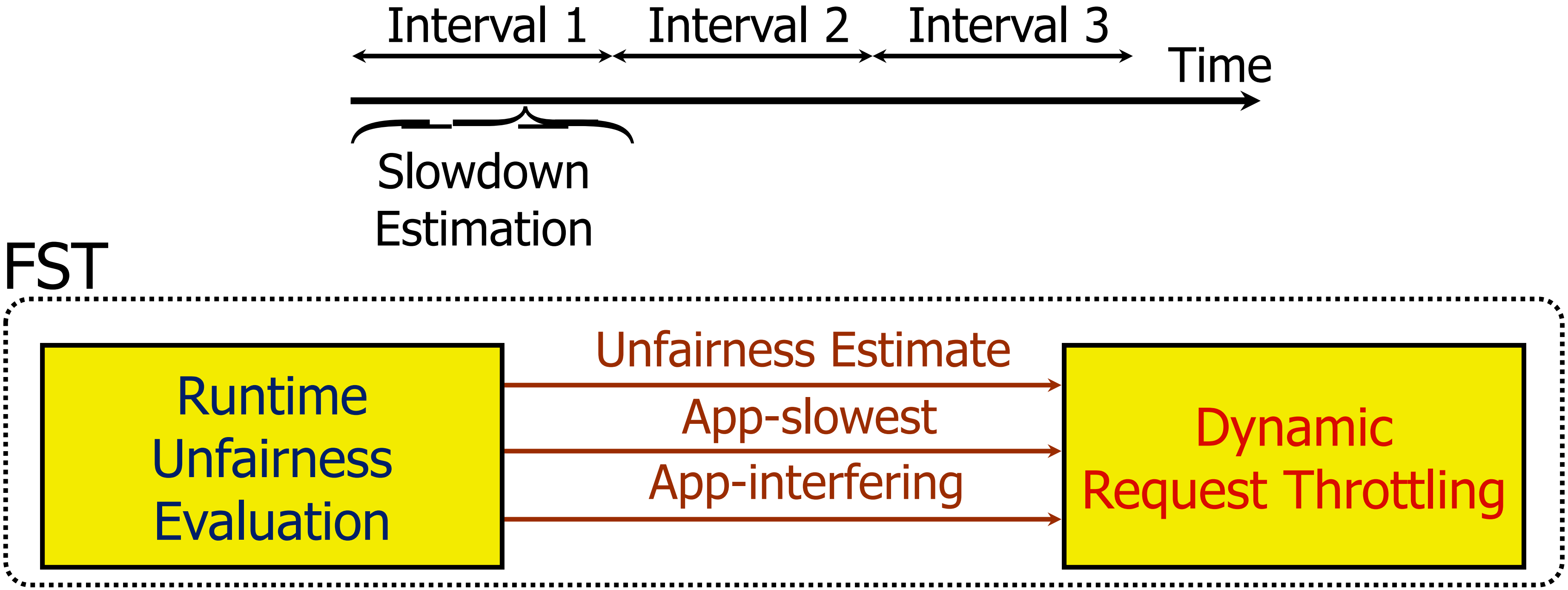
Source Throttling: A Fairness Substrate

- Key idea: Manage inter-thread interference at the cores (sources), not at the shared resources

- Dynamically estimate unfairness in the memory system
- Feed back this information into a controller
- Throttle cores' memory access rates accordingly
  - Whom to throttle and by how much depends on performance target (throughput, fairness, per-thread QoS, etc)
  - E.g., if unfairness > system-software-specified target then throttle down core causing unfairness &
    throttle up core that was unfairly treated

- Ebrahimi et al., "Fairness via Source Throttling," ASPLOS'10, TOCS'12.

Fairness via Source Throttling (FST) [ASPLOS'10]



1- Estimating system unfairness
2- Find app. with the highest slowdown (App-slowest)
3- Find app. causing most interference for App-slowest (App-interfering)

```
if (Unfairness Estimate >Target)
{
 1-Throttle down App-interfering
    (limit injection rate and parallelism)
 2-Throttle up App-slowest
}
```

Core (Source) Throttling

- Idea: Estimate the slowdown due to (DRAM) interference and throttle down threads that slow down others
  - Ebrahimi et al., "Fairness via Source Throttling: A Configurable and High-Performance Fairness Substrate for Multi-Core Memory Systems," ASPLOS 2010.

- Advantages

  + Core/request throttling is easy to implement: no need to change the memory scheduling algorithm

  + Can be a general way of handling shared resource contention

  + Can reduce overall load/contention in the memory system

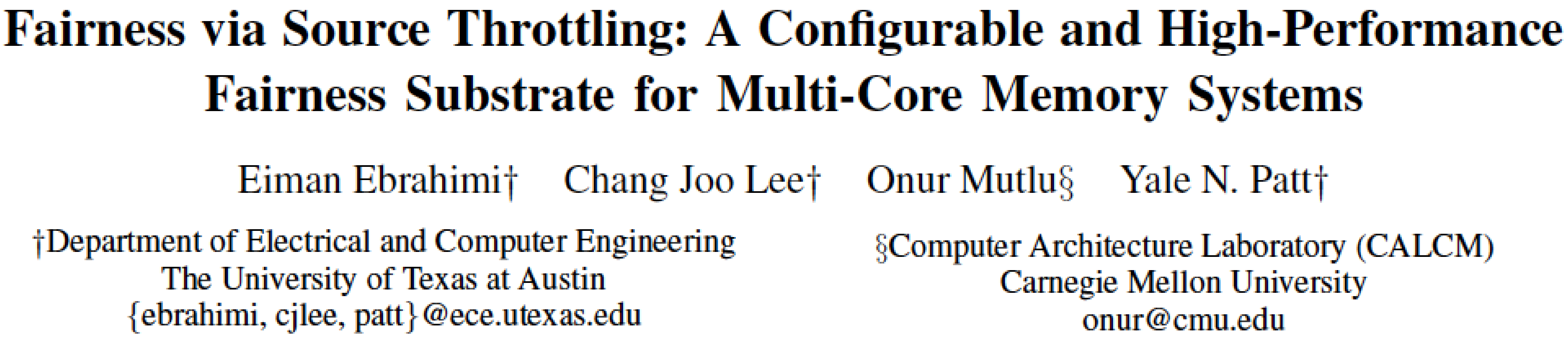
- Disadvantages

  - Requires interference/slowdown estimations → difficult to estimate

  - Thresholds can become difficult to optimize → throughput loss

More on Source Throttling (I)

- Eiman Ebrahimi, Chang Joo Lee, Onur Mutlu, and Yale N. Patt, **"Fairness via Source Throttling: A Configurable and High-Performance Fairness Substrate for Multi-Core Memory Systems"** *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems* (***ASPLOS***), pages 335-346, Pittsburgh, PA, March 2010. Slides (pdf)



Fairness via Source Throttling: A Configurable and High-Performance Fairness Substrate for Multi-Core Memory Systems

Eiman Ebrahimi† Chang Joo Lee† Onur Mutlu§ Yale N. Patt†

†Department of Electrical and Computer Engineering
The University of Texas at Austin
{ebrahimi, cjlee, patt}@ece.utexas.edu

§Computer Architecture Laboratory (CALCM)
Carnegie Mellon University
onur@cmu.edu

Fundamental Interference Control Techniques

- Goal: to reduce/control interference

1. Prioritization or request scheduling

2. Data mapping to banks/channels/ranks

3. Core/source throttling

4. Application/thread scheduling

Idea: Pick threads that do not badly interfere with each other to be scheduled together on cores sharing the memory system

Interference-Aware Thread Scheduling

- Advantages

  + Can eliminate/minimize interference by scheduling "symbiotic applications" together (as opposed to just managing the interference)

  + Less intrusive to hardware (less need to modify the hardware resources)

- Disadvantages and Limitations

  -- High overhead to migrate threads between cores and machines

  -- Does not work (well) if all threads are similar and they interfere

Summary: Fundamental Interference Control Techniques

- Goal: to reduce/control interference

1. Prioritization or request scheduling

2. Data mapping to banks/channels/ranks

3. Core/source throttling

4. Application/thread scheduling

Best is to combine all. How would you do that?

Required Readings for Wednesday

- Required Reading Assignment:
  - Dubois, Annavaram, Stenstrom, Chapter 6.

- Recommended References:
  - Moscibroda and Mutlu, "A Case for Bufferless Routing in On-Chip Networks," ISCA 2009.
  - Das et al., "Application-Aware Prioritization Mechanisms for On-Chip Networks," MICRO 2009.

18-740/640
Computer Architecture
Lecture 15: Memory Resource Management II

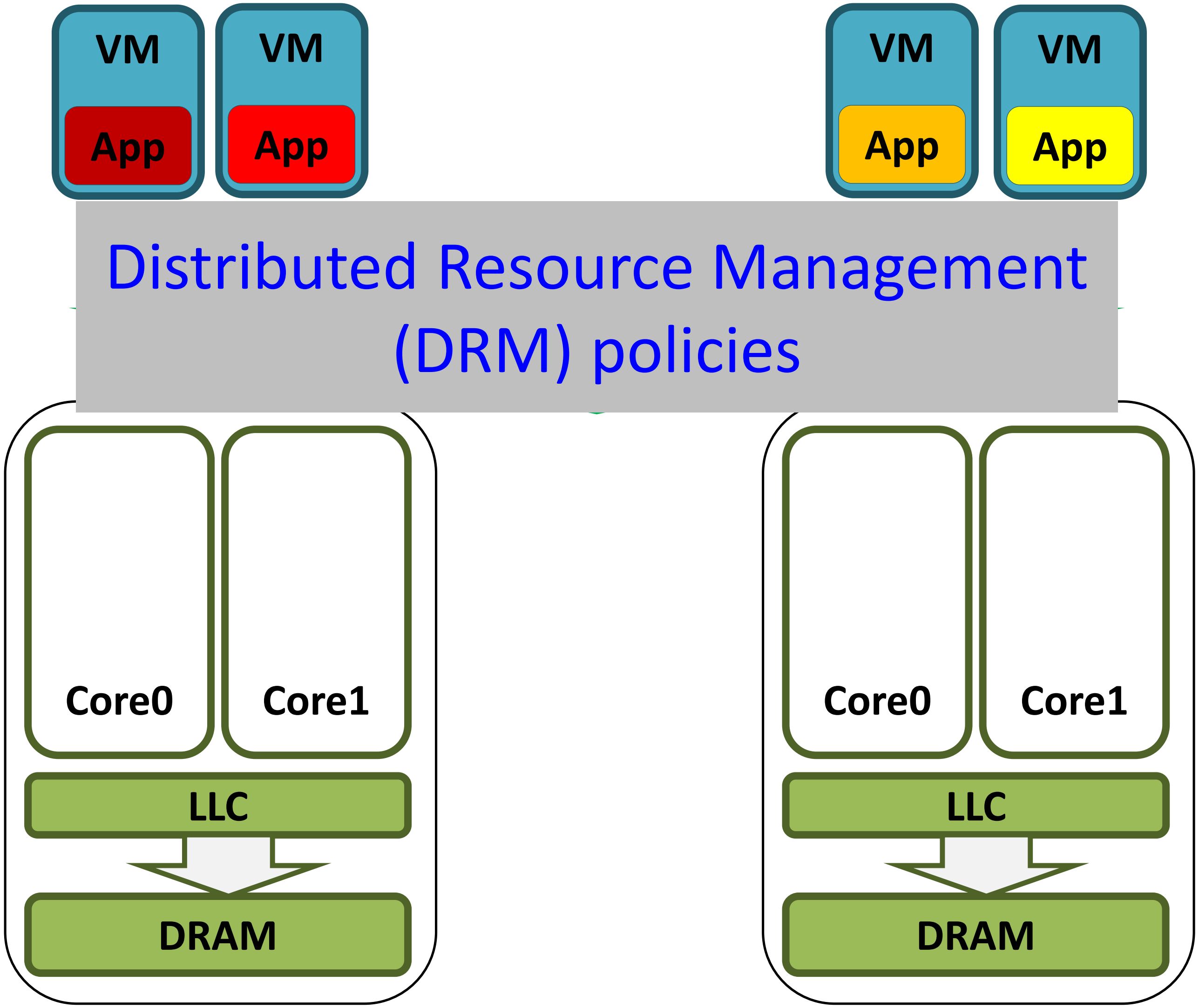
Prof. Onur Mutlu
Carnegie Mellon University

Fall 2015, 11/2/2015

Interference-Aware Thread Scheduling

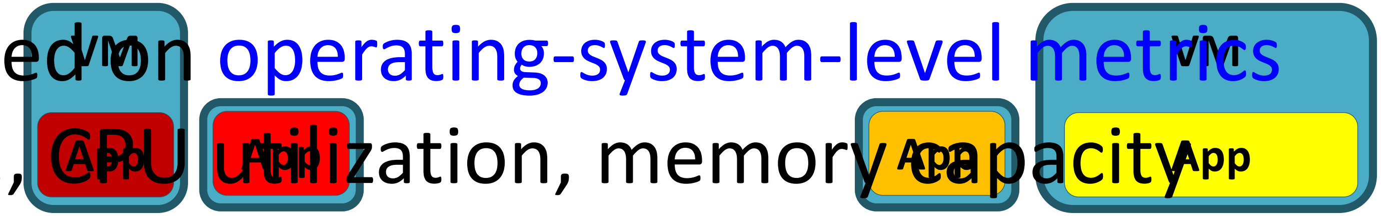
- An example from scheduling in clusters (data centers)
- Clusters can be running virtual machines

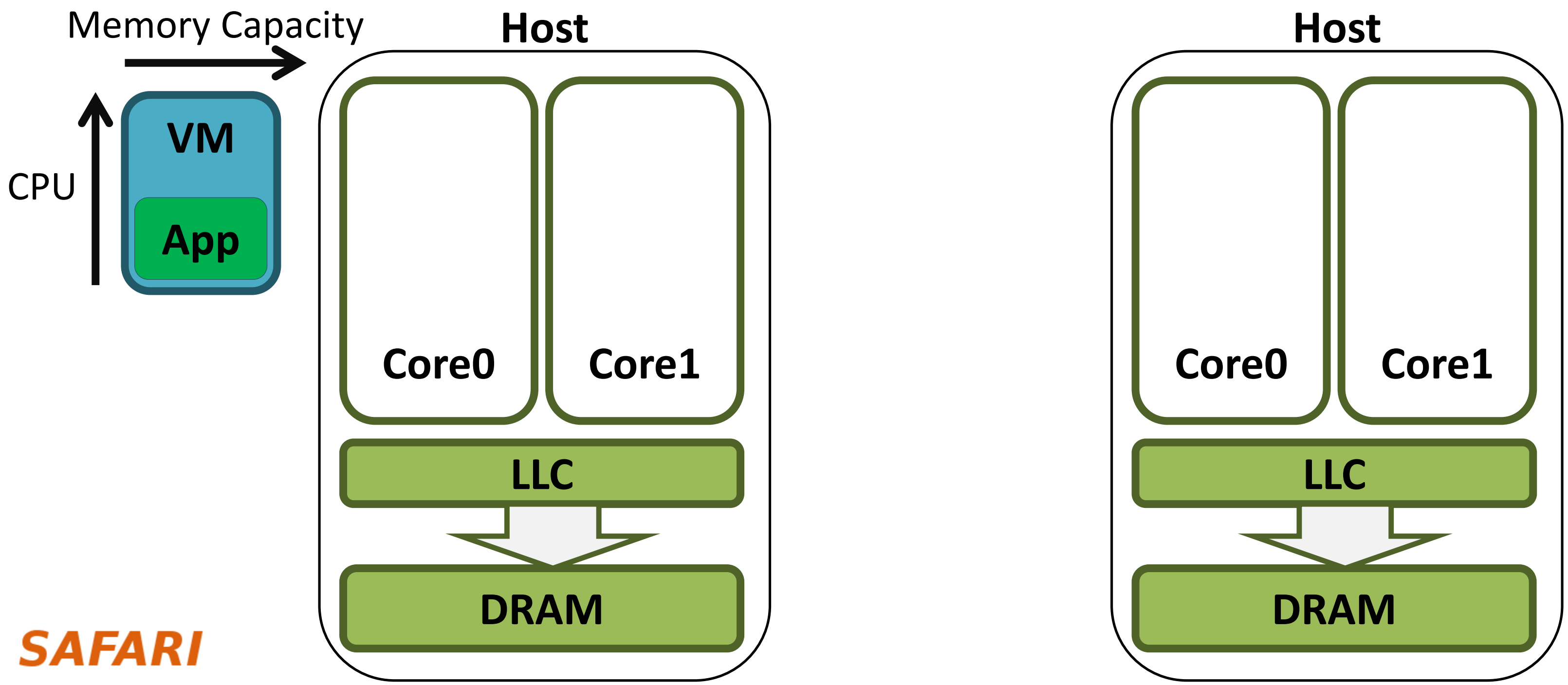
Virtualized Cluster



Conventional DRM Policies

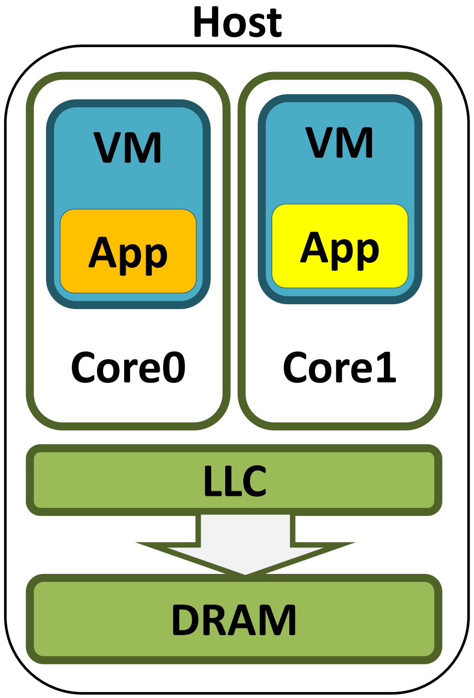
Based on operating-system-level metrics
e.g., CPU utilization, memory capacity demand





Microarchitecture-level Interference

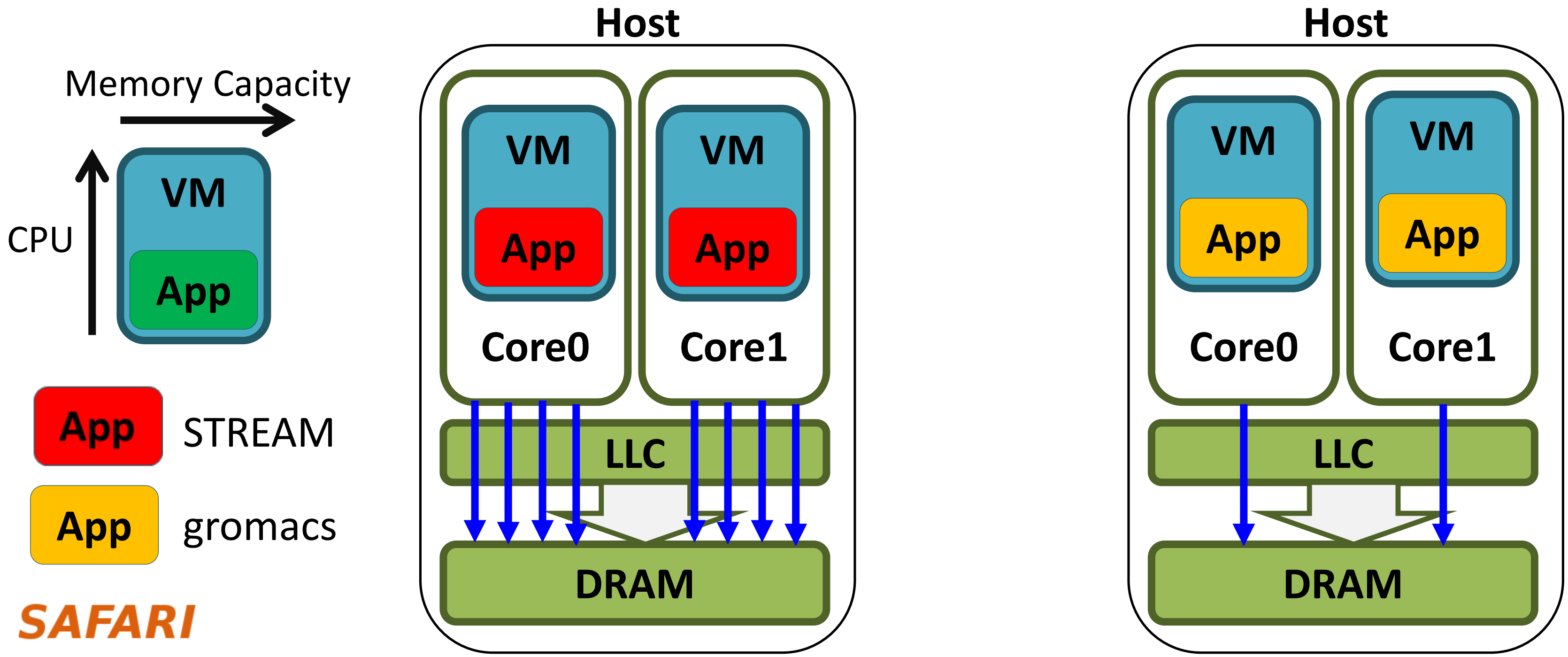
- VMs within a host compete for:
  - Shared cache capacity
  - Shared memory bandwidth



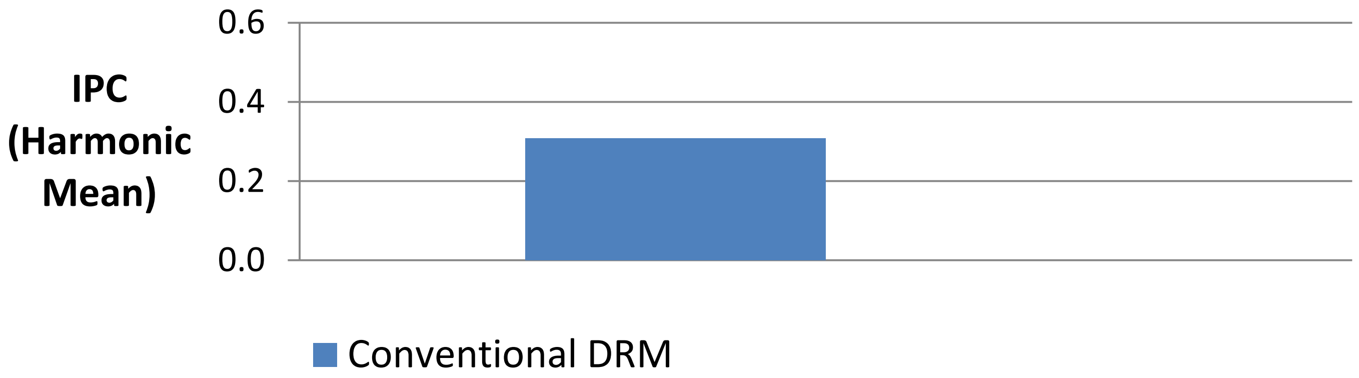
Can operating-system-level metrics capture the microarchitecture-level resource interference?

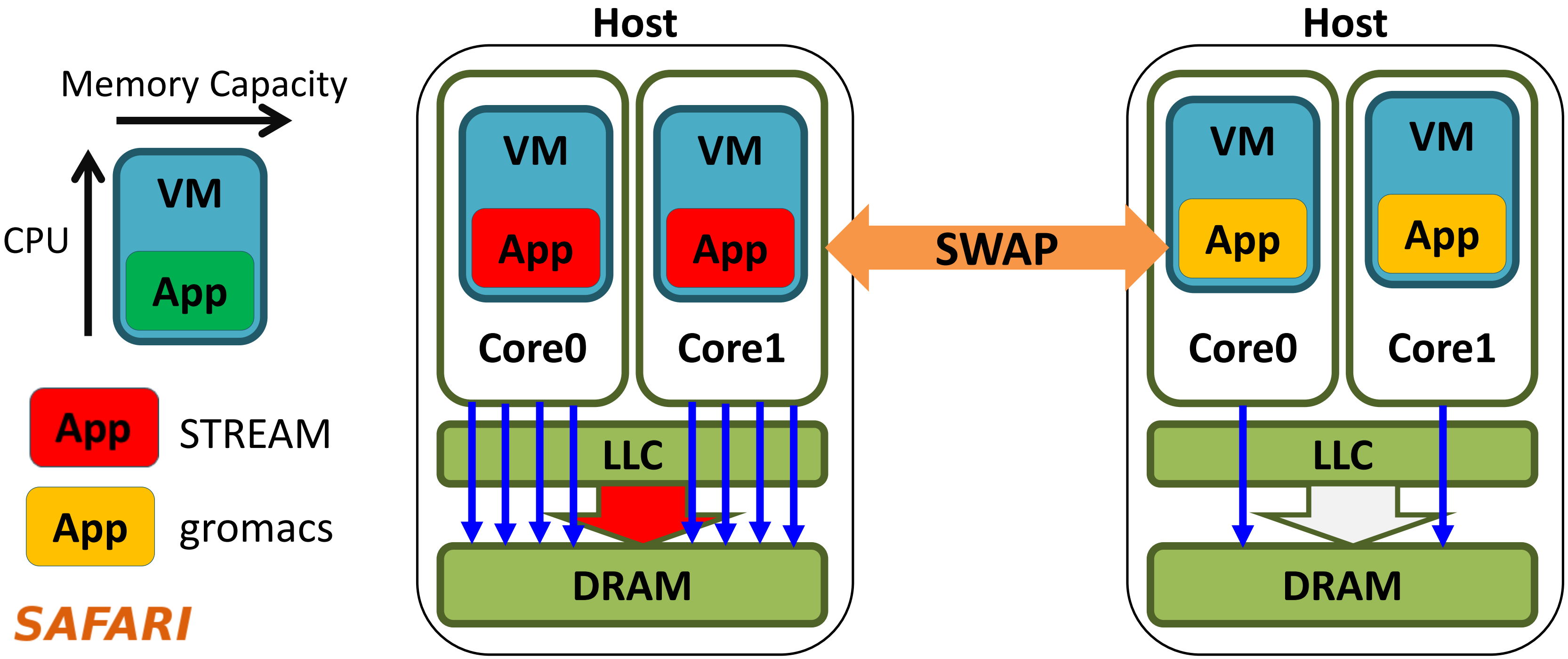
Microarchitecture Unawareness

VM	Operating-system-level metrics		Microarchitecture-level metrics	
	CPU Utilization	Memory Capacity	LLC Hit Ratio	Memory Bandwidth
🟥 App	92%	369 MB	2%	2267 MB/s
🟨 App	93%	348 MB	98%	1 MB/s

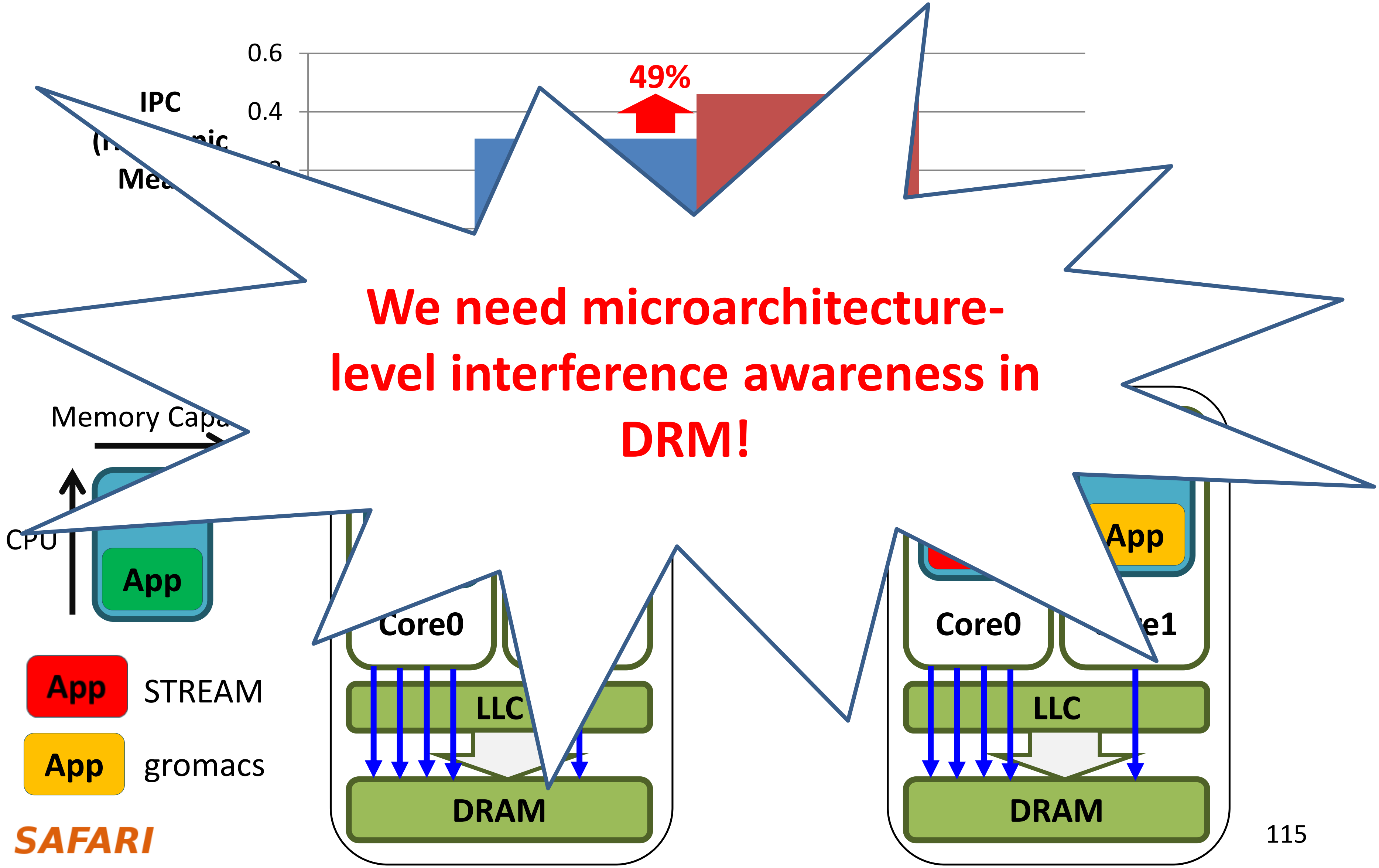


Impact on Performance





Impact on Performance



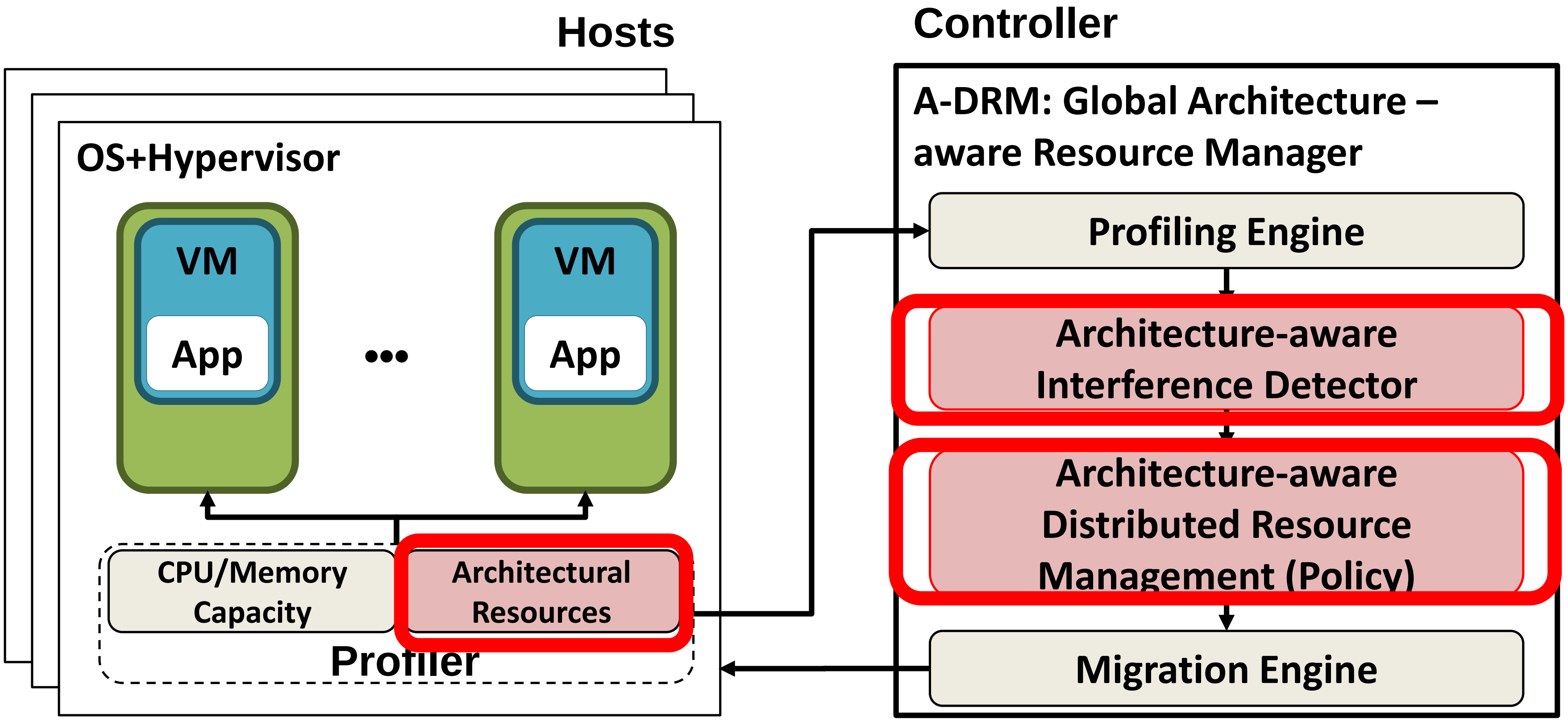
We need microarchitecture-level interference awareness in DRM!

A-DRM: Architecture-aware DRM

- **Goal**: Take into account microarchitecture-level shared resource interference
  - Shared cache capacity
  - Shared memory bandwidth

- **Key Idea**:
  - Monitor and detect microarchitecture-level shared resource interference
  - Balance microarchitecture-level resource usage across cluster to minimize memory interference while maximizing system performance

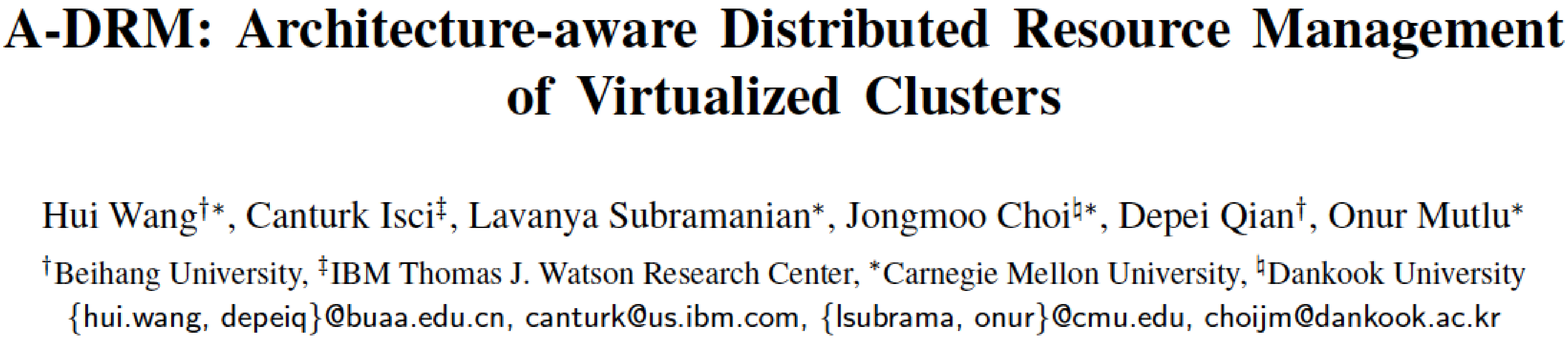
A-DRM: Architecture-aware DRM


Hosts
OS+Hypervisor
VM
App
•••
VM
App
CPU/Memory Capacity
Architectural Resources
Profiler
Controller
A-DRM: Global Architecture –aware Resource Manager
Profiling Engine
Architecture-aware Interference Detector
Architecture-aware Distributed Resource Management (Policy)
Migration Engine

SAFARI

More on Architecture-Aware DRM

- Hui Wang, Canturk Isci, Lavanya Subramanian, Jongmoo Choi, Depei Qian, and Onur Mutlu,
"A-DRM: Architecture-aware Distributed Resource Management of Virtualized Clusters"
*Proceedings of the 11th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (**VEE**)*, Istanbul, Turkey, March 2015.
[Slides (pptx) (pdf)]



A-DRM: Architecture-aware Distributed Resource Management of Virtualized Clusters

Hui Wang[†*], Canturk Isci[‡], Lavanya Subramanian[*], Jongmoo Choi[♮*], Depei Qian[†], Onur Mutlu[*]
[†]Beihang University, [‡]IBM Thomas J. Watson Research Center, [*]Carnegie Mellon University, [♮]Dankook University
{hui.wang, depeiq}@buaa.edu.cn, canturk@us.ibm.com, {lsubrama, onur}@cmu.edu, choijm@dankook.ac.kr

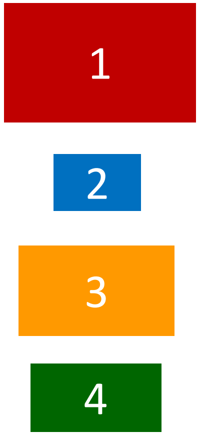
The Blacklisting Memory Scheduler

Rachata Ausavarungnirun, Kevin Chang, Lavanya Subramanian, Gabriel Loh, and Onur Mutlu,
"Staged Memory Scheduling: Achieving High Performance and Scalability in Heterogeneous Systems"
39th International Symposium on Computer Architecture (***ISCA***),
Portland, OR, June 2012.

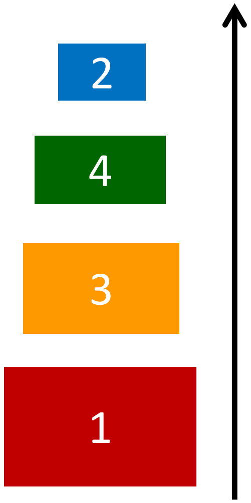
SMS ISCA 2012 Talk

Tackling Inter-Application Interference: Application-aware Memory Scheduling

Monitor

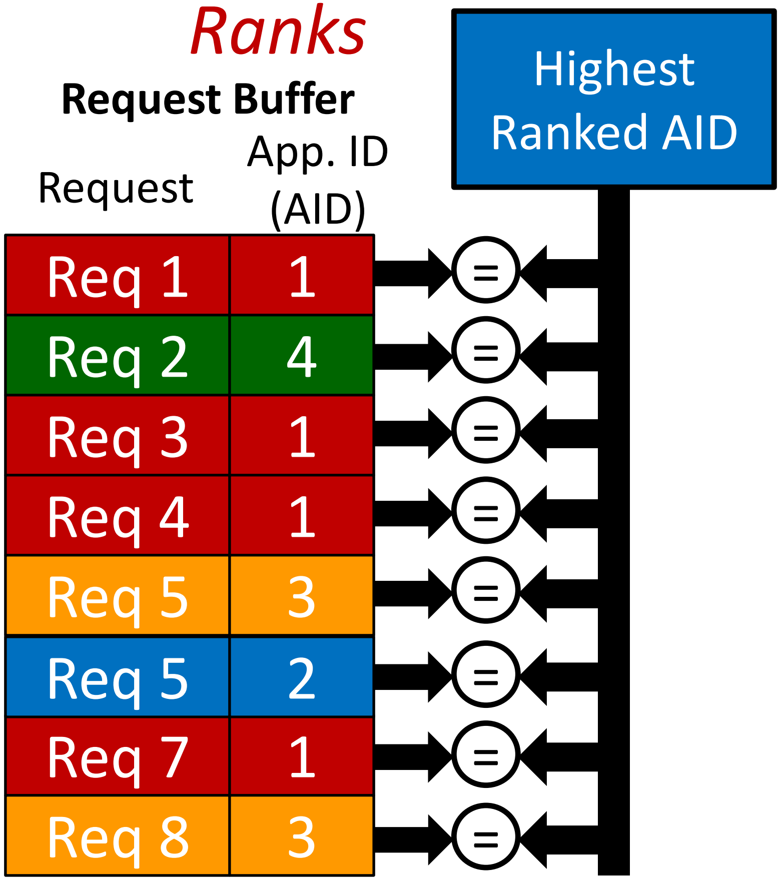


Rank



Full ranking increases critical path latency and area significantly to improve performance and fairness

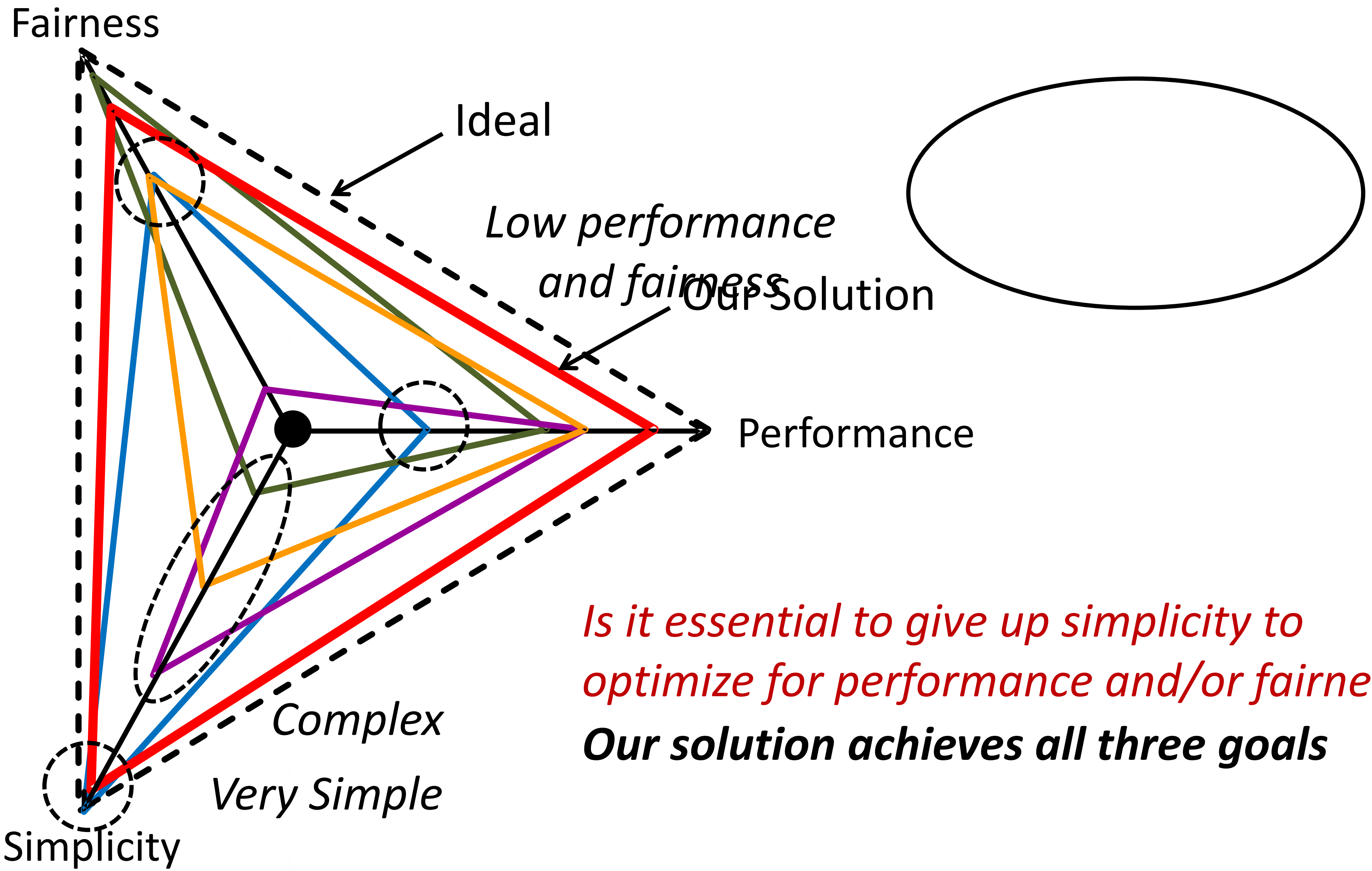
Enforce Ranks



Request Buffer

Request	App. ID (AID)
Req 1	1
Req 2	4
Req 3	1
Req 4	1
Req 5	3
Req 5	2
Req 7	1
Req 8	3

Performance vs. Fairness vs. Simplicity

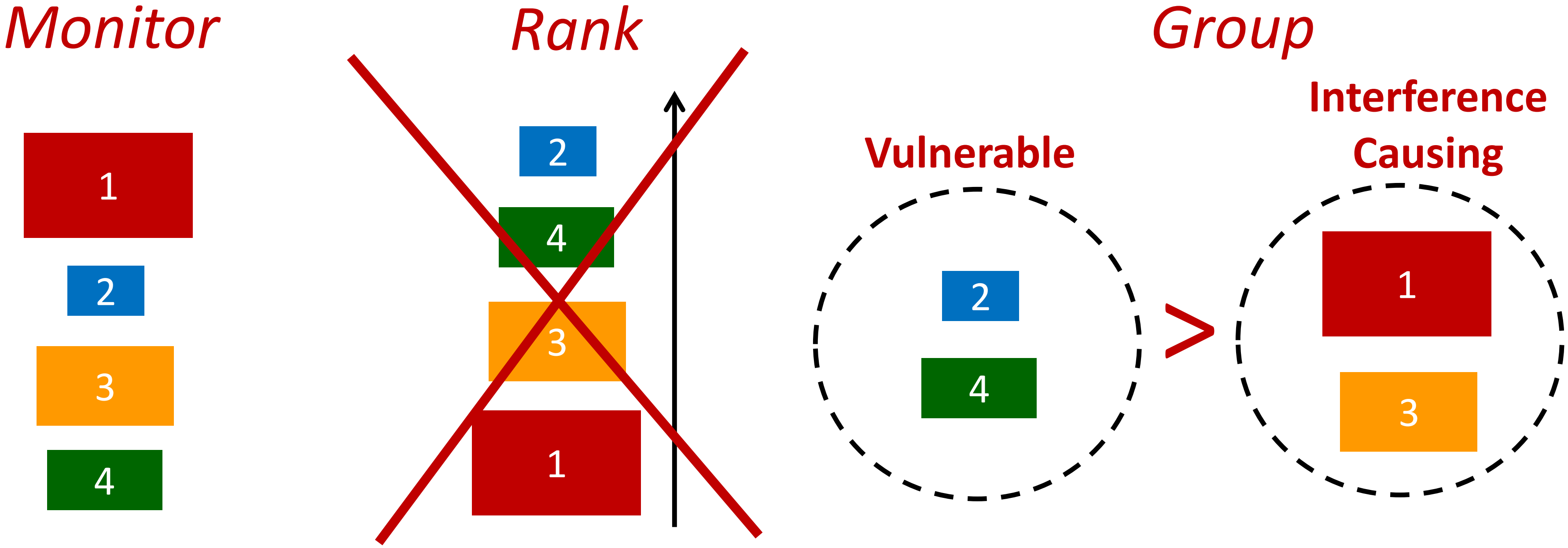


Is it essential to give up simplicity to optimize for performance and/or fairness?

Our solution achieves all three goals

Key Observation 1: Group Rather Than Rank

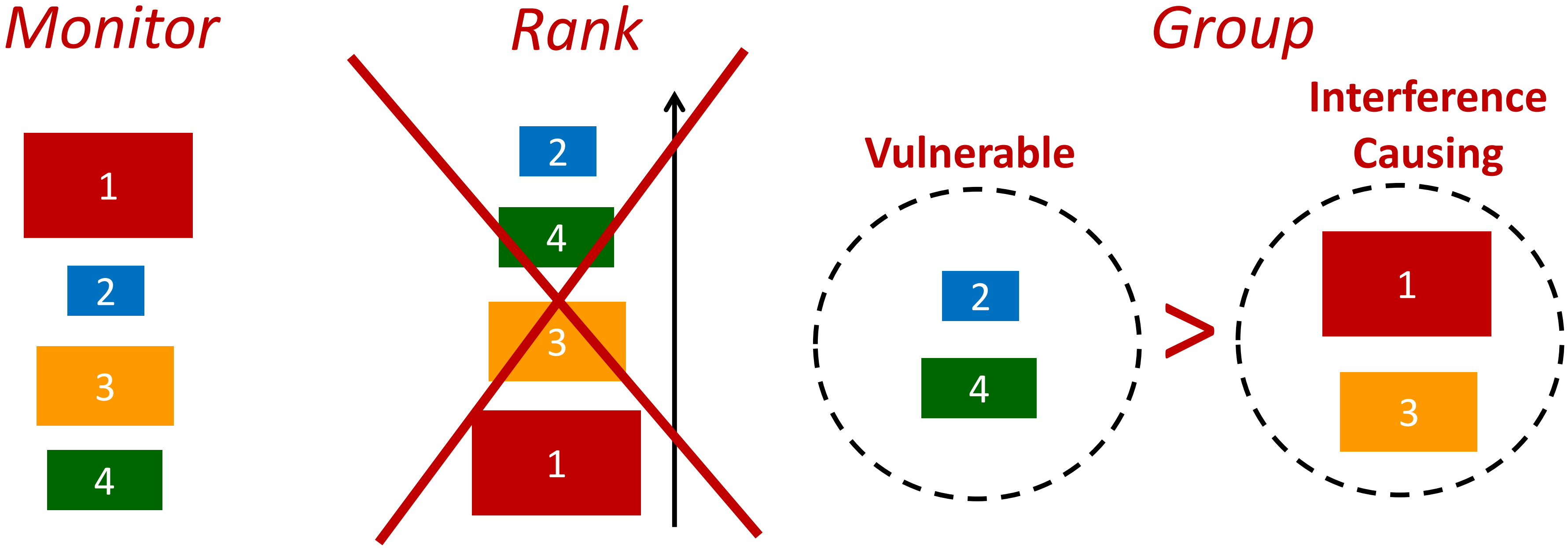
Observation 1: Sufficient to separate applications into two groups, rather than do full ranking



Benefit 2: Lower slowdowns than ranking

Key Observation 1: Group Rather Than Rank

Observation 1: Sufficient to separate applications into two groups, rather than do full ranking



How to classify applications into groups?

Key Observation 2

Observation 2: Serving a large number of consecutive requests from an application causes interference

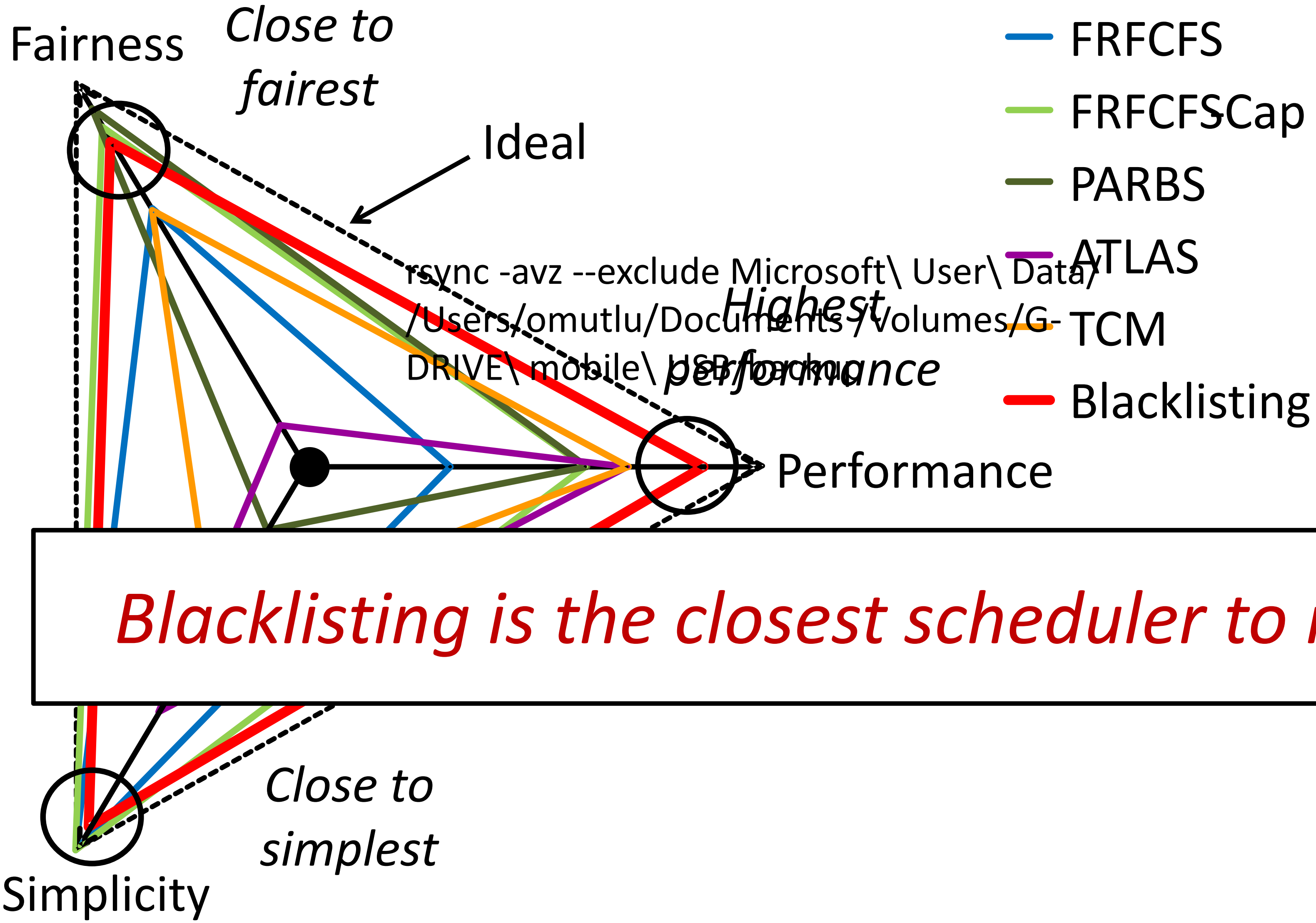
Basic Idea:

- *Group* applications with a large number of consecutive requests as *interference-causing* → *Blacklisting*
- *Deprioritize* blacklisted applications
- *Clear* blacklist periodically (1000s of cycles)

Benefits:

- *Lower complexity*
- *Finer grained grouping decisions → Lower unfairness*

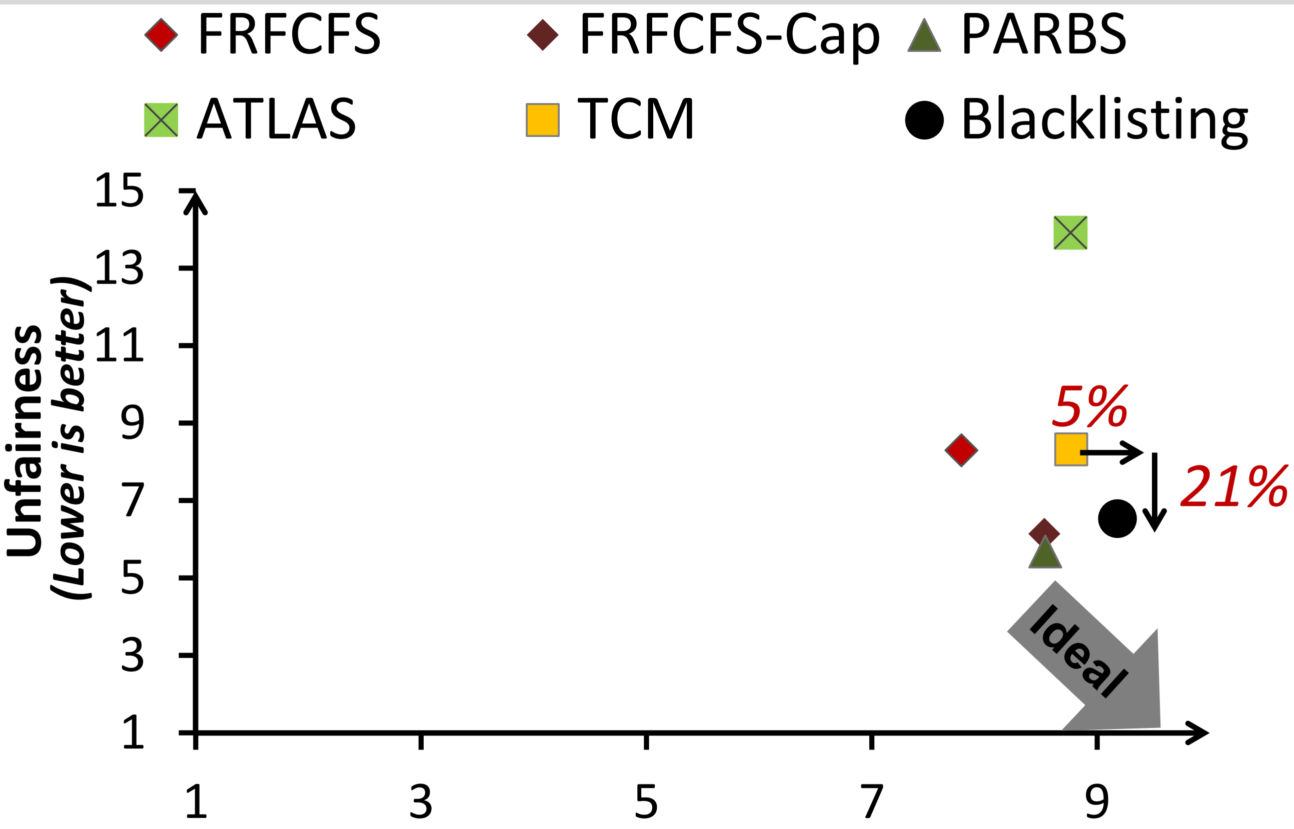
Performance vs. Fairness vs. Simplicity



rsync -avz --exclude Microsoft\ User\ Data/ /Users/omutlu/Documents /Volumes/G-DRIVE\ mobile\ USB/backup

Blacklisting is the closest scheduler to ideal

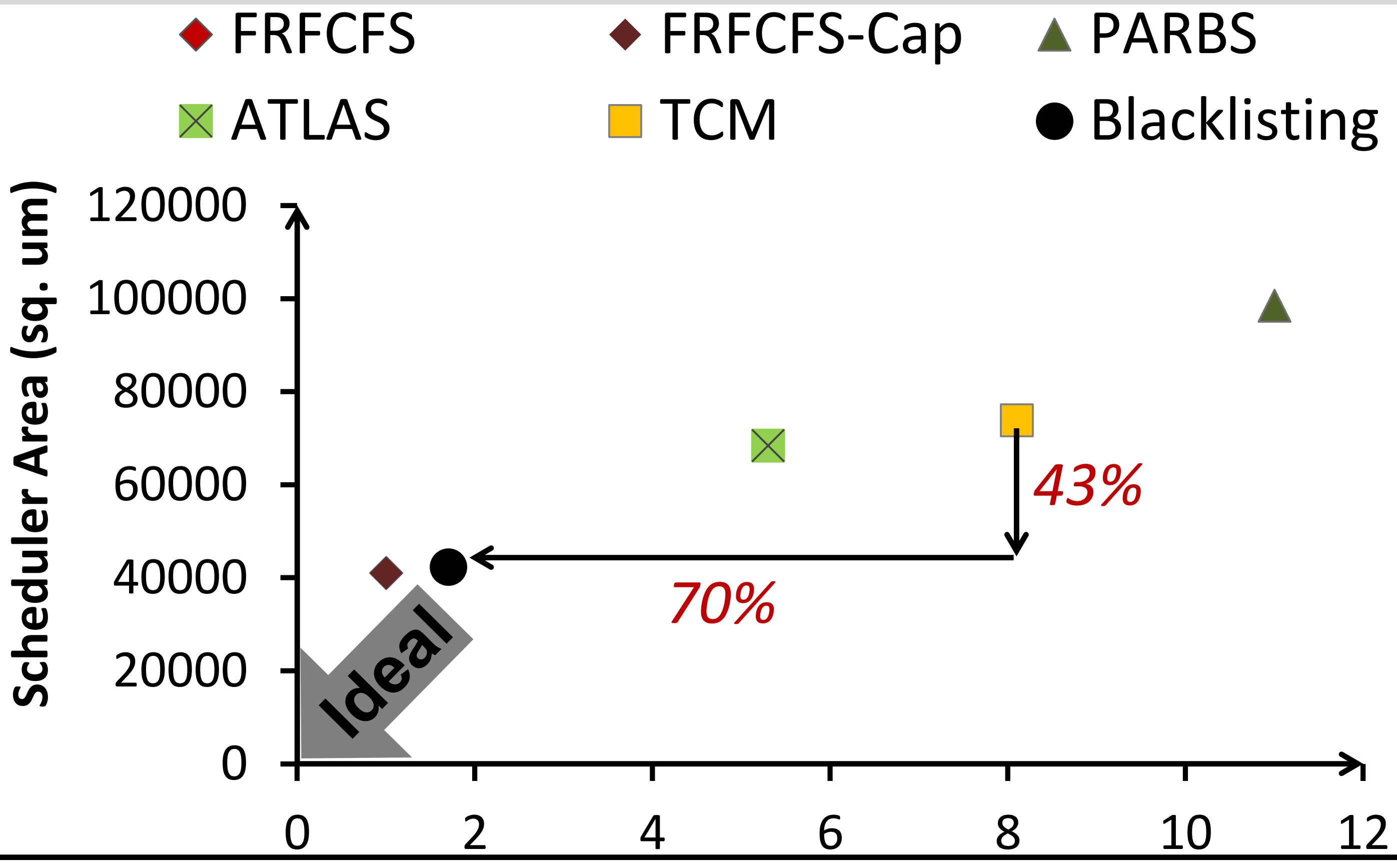
Performance and Fairness



1. Blacklisting achieves the highest performance

2. Blacklisting balances performance and fairness

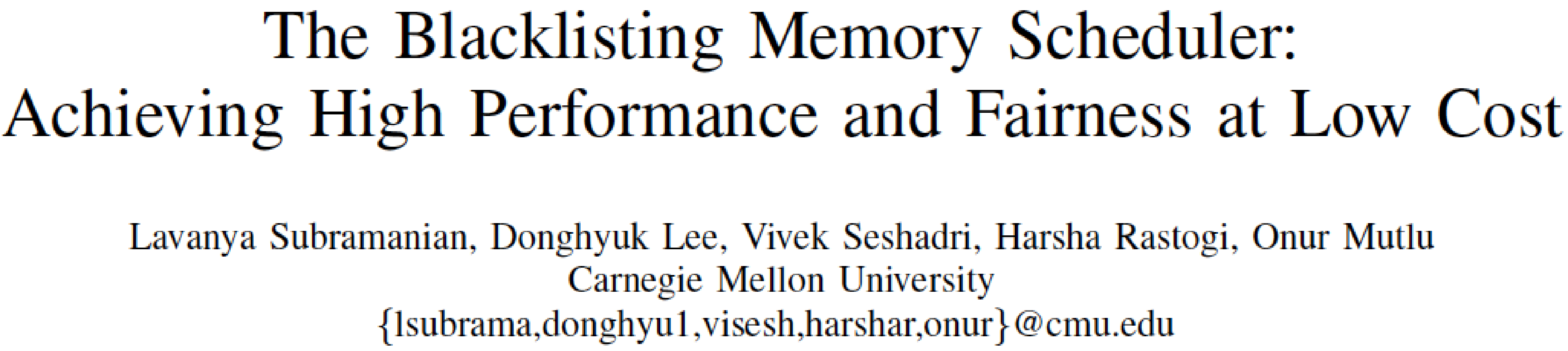
Complexity



Blacklisting reduces complexity significantly

More on BLISS (I)

- Lavanya Subramanian, Donghyuk Lee, Vivek Seshadri, Harsha Rastogi, and Onur Mutlu, **"The Blacklisting Memory Scheduler: Achieving High Performance and Fairness at Low Cost"** *Proceedings of the 32nd IEEE International Conference on Computer Design (**ICCD**)*, Seoul, South Korea, October 2014. [Slides (pptx) (pdf)]



The Blacklisting Memory Scheduler:
Achieving High Performance and Fairness at Low Cost

Lavanya Subramanian, Donghyuk Lee, Vivek Seshadri, Harsha Rastogi, Onur Mutlu
Carnegie Mellon University
{lsubrama,donghyu1,visesh,harshar,onur}@cmu.edu

More on BLISS: Longer Version

- Lavanya Subramanian, Donghyuk Lee, Vivek Seshadri, Harsha Rastogi, and Onur Mutlu, **"The Blacklisting Memory Scheduler: Balancing Performance, Fairness and Complexity"** *SAFARI Technical Report*, TR-SAFARI-2015-004, Carnegie Mellon University, March 2015.

The Blacklisting Memory Scheduler: Balancing Performance, Fairness and Complexity

Lavanya Subramanian, Donghyuk Lee, Vivek Seshadri, Harsha Rastogi, Onur Mutlu
Carnegie Mellon University
{lsubrama,donghyu1,visesh,harshar,onur}@cmu.edu
SAFARI Technical Report No. 2015-004