Designing High-Performance and Fair Shared Multi-Core Memory Systems: Two Approaches

Onur Mutlu

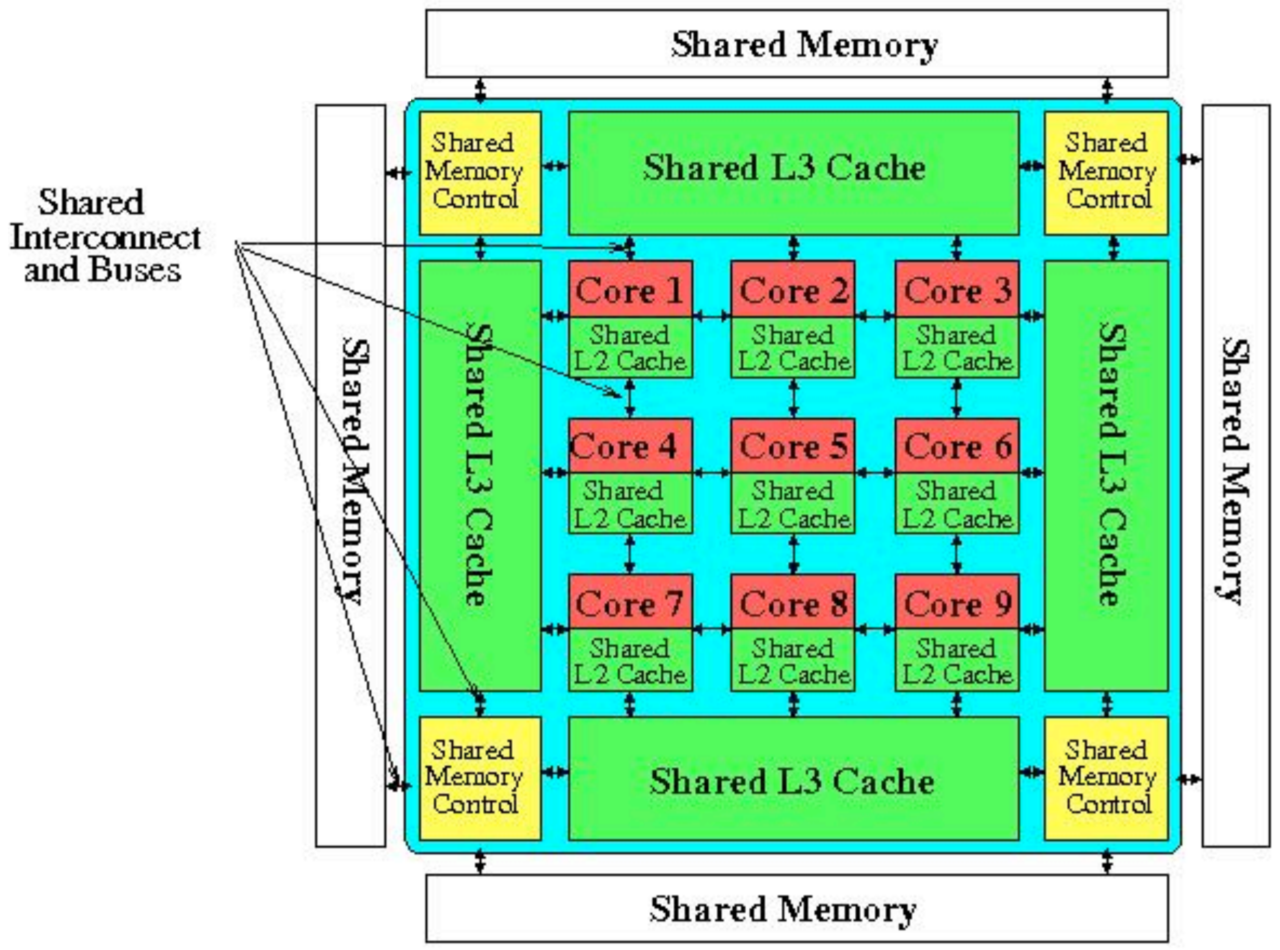
onur@cmu.edu

March 23, 2010

GSRC

Carnegie Mellon

Modern Memory Systems (Multi-Core)



L1 Caches are private to each core

The Memory System

- The memory system is a fundamental performance and power bottleneck in almost all computing systems

- Recent technology, architecture, and application trends lead to new requirements from the memory system:
  - Scalability (technology and algorithm)
  - Fairness and QoS-awareness
  - Energy/power efficiency

- Focus of this talk: enabling fair and high-performance sharing of the memory system among multiple cores/threads

Agenda

- Technology, Application, Architecture Trends
- Requirements from the Memory Hierarchy
- The Problem: Interference in Memory System
- Two Solution Approaches
  - Smart resources: ATLAS Memory Scheduler
  - Dumb resources: Fairness via Source Throttling
- Future Work
- Conclusions

Technology Trends

- DRAM does not scale well beyond N nm
  - Memory scaling benefits: density, capacity, cost

- Energy/power already key design limiters
  - Memory system responsible for a large fraction of power

- More transistors (cores) on chip (Moore's Law)
- Pin bandwidth not increasing as fast as number of transistors
  - Memory subsystem is a key shared resource among cores
  - More pressure on the memory hierarchy

Application/System Trends

- Many different threads/applications/virtual machines will share the memory system
  - Cloud computing/servers: Many workloads consolidated on-chip to improve efficiency
  - GP-GPUs: Many threads from multiple parallel applications
  - Mobile: Interactive + non-interactive consolidation

- Different applications with different requirements (SLAs)
  - Some applications/threads require performance guarantees
  - Memory system does not distinguish between applications
- Different goals for different systems/users
  - System throughput, fairness, per-application performance
  - Memory system does not control application interference, is not configurable

Architecture Trends

- More cores and components
    - More pressure on the memory hierarchy
- Asymmetric cores: Performance asymmetry, CPU+GPUs, accelerators, …
    - Motivated by energy efficiency and Amdahl’s Law
- Different cores have different performance requirements
    - Memory hierarchies do not distinguish between cores

Agenda

- Technology, Application, Architecture Trends
- Requirements from the Memory Hierarchy
- The Problem: Interference in Memory System
- Two Solution Approaches
  - Smart resources: ATLAS Memory Scheduler
  - Dumb resources: Fairness via Source Throttling
- Future Work
- Conclusions

Requirements from an Ideal Hierarchy

- Traditional
  - High system performance
  - Enough capacity
  - Low cost

- New
  - Technology scalability
  - QoS support and configurability
  - Energy (and power, bandwidth) efficiency

Requirements from an Ideal Hierarchy

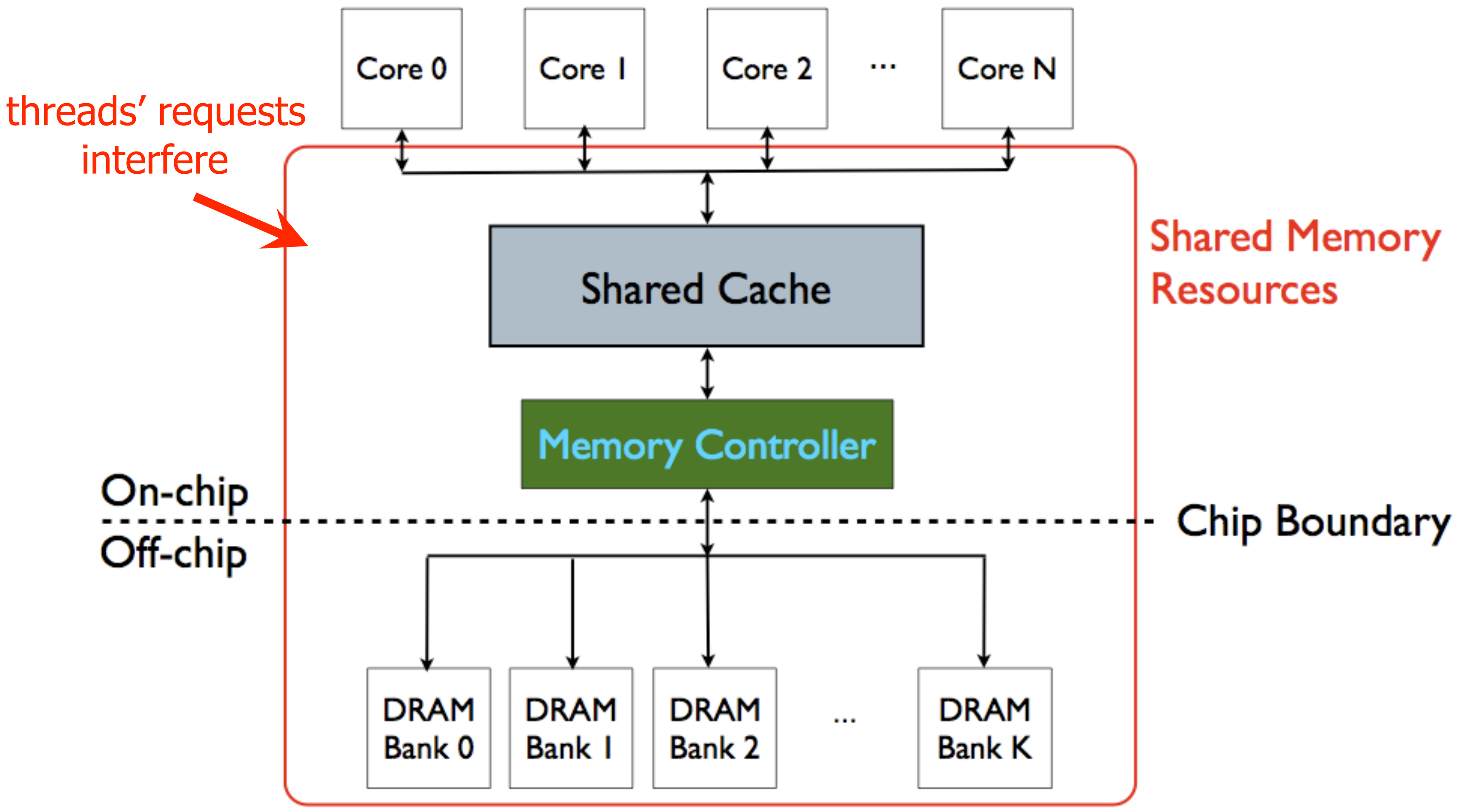
- Traditional
  - High system performance: Reduce inter-thread interference
  - Enough capacity
  - Low cost

- New
  - Technology scalability
    - Emerging non-volatile memory technologies (PCM) can help
  - QoS support and configurability
    - Need HW mechanisms to control interference and build QoS policies
  - Energy (and power, bandwidth) efficiency
    - One size fits all wastes energy, performance, bandwidth

Agenda

- Technology, Application, Architecture Trends
- Requirements from the Memory Hierarchy
- The Problem: Interference in Memory System
- Two Solution Approaches
  - Smart resources: ATLAS Memory Scheduler
  - Dumb resources: Fairness via Source Throttling
- Future Work
- Conclusions

Memory System is the Major Shared Resource

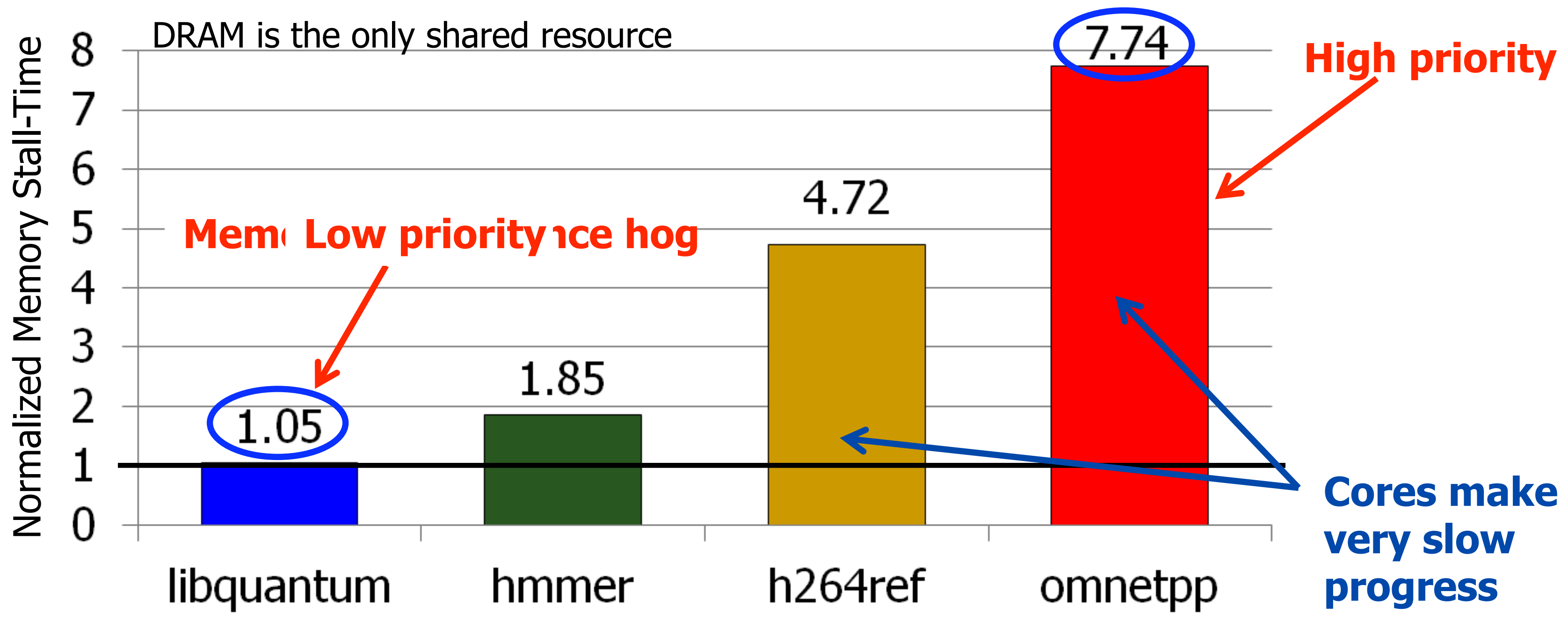


Inter-Thread/Application Interference

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

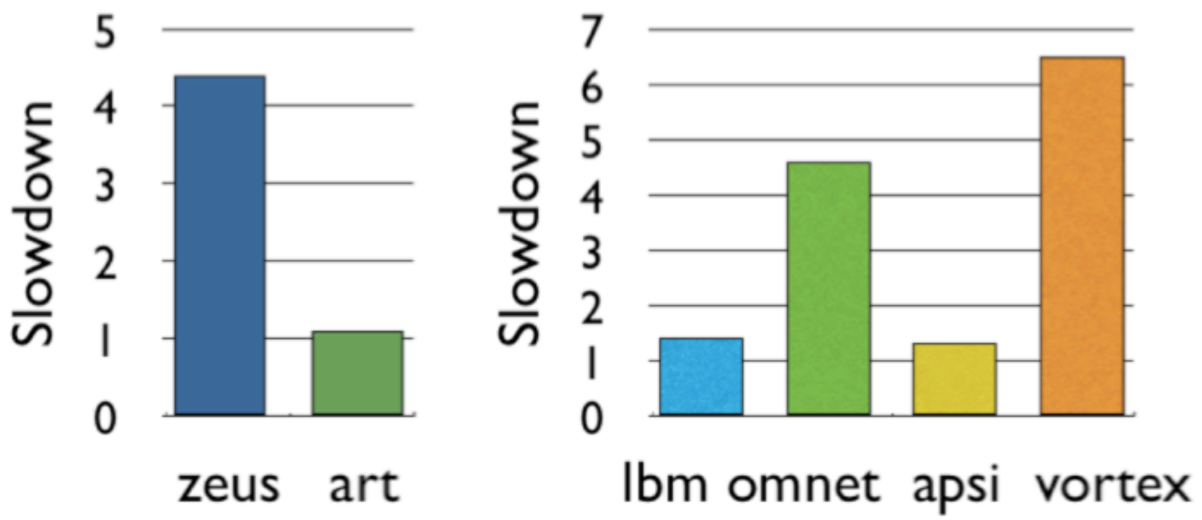
- Existing memory systems
  - Free-for-all, shared based on demand
  - Control algorithms thread-unaware and thread-unfair
  - Aggressive threads can deny service to others
  - Do not try to reduce or control inter-thread interference

Problems due to Uncontrolled Interference



- Unfair slowdown of different threads [MICRO'07, ISCA'08, ASPLOS'10]
- Low system performance [MICRO'07, ISCA'08, HPCA'10]
- Vulnerability to denial of service [USENIX Security'07]
- Priority inversion: unable to enforce priorities/SLAs [MICRO'07]
- Poor performance predictability (no performance isolation)

Problems due to Uncontrolled Interference



- Unfair slowdown of different threads [MICRO'07, ISCA'08, ASPLOS'10]
- Low system performance [MICRO'07, ISCA'08, HPCA'10]
- Vulnerability to denial of service [USENIX Security'07]
- Priority inversion: unable to enforce priorities/SLAs [MICRO'07]
- Poor performance predictability (no performance isolation)

QoS-Aware Memory Systems: Challenges

- How do we reduce inter-thread interference?
  - Improve system performance and utilization
  - Preserve the benefits of single-thread performance techniques
- How do we control inter-thread interference?
  - Provide scalable mechanisms to enable system software to enforce a variety of QoS policies
  - All the 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

Agenda

- Technology, Application, Architecture Trends
- Requirements from the Memory Hierarchy
- The Problem: Interference in Memory System
- Two Solution Approaches
    - Smart resources: ATLAS Memory Scheduler
    - Dumb resources: Fairness via Source Throttling
- Future Work
- Conclusions

Designing QoS-Aware Memory Systems: Approaches

- Smart resources: Design each shared resource to have a configurable interference control/reduction mechanism
  - Fair/QoS-aware memory schedulers, interconnects, caches, arbiters
    - Fair memory schedulers [Mutlu MICRO 2007], parallelism-aware memory schedulers [Mutlu ISCA 2008], ATLAS memory scheduler [Kim et al. HPCA 2010]
    - Application-aware on-chip networks [Das et al. MICRO 2009, ISCA 2010, Grot et al. MICRO 2009]

- Dumb resources: Keep each resource free-for-all, but control access to memory system at the cores/sources
  - Estimate interference/slowdown in the entire system and throttle cores that slow down others
    - Fairness via Source Throttling [Ebrahimi et al., ASPLOS 2010]
    - Coordinated Prefetcher Throttling [Ebrahimi et al., MICRO 2009]

Agenda

- Technology, Application, Architecture Trends
- Requirements from the Memory Hierarchy
- The Problem: Interference in Memory System
- Two Solution Approaches
  - Smart resources: ATLAS Memory Scheduler
  - Dumb resources: Fairness via Source Throttling
- Future Work
- Conclusions

ATLAS Memory Scheduler

Kim et al., "ATLAS: A Scalable and High-Performance Scheduling Algorithm for Multiple Memory Controllers," HPCA 2010.

Desired Properties of Memory Scheduling Algorithm

- **Maximize system performance**
  - Without starving any cores

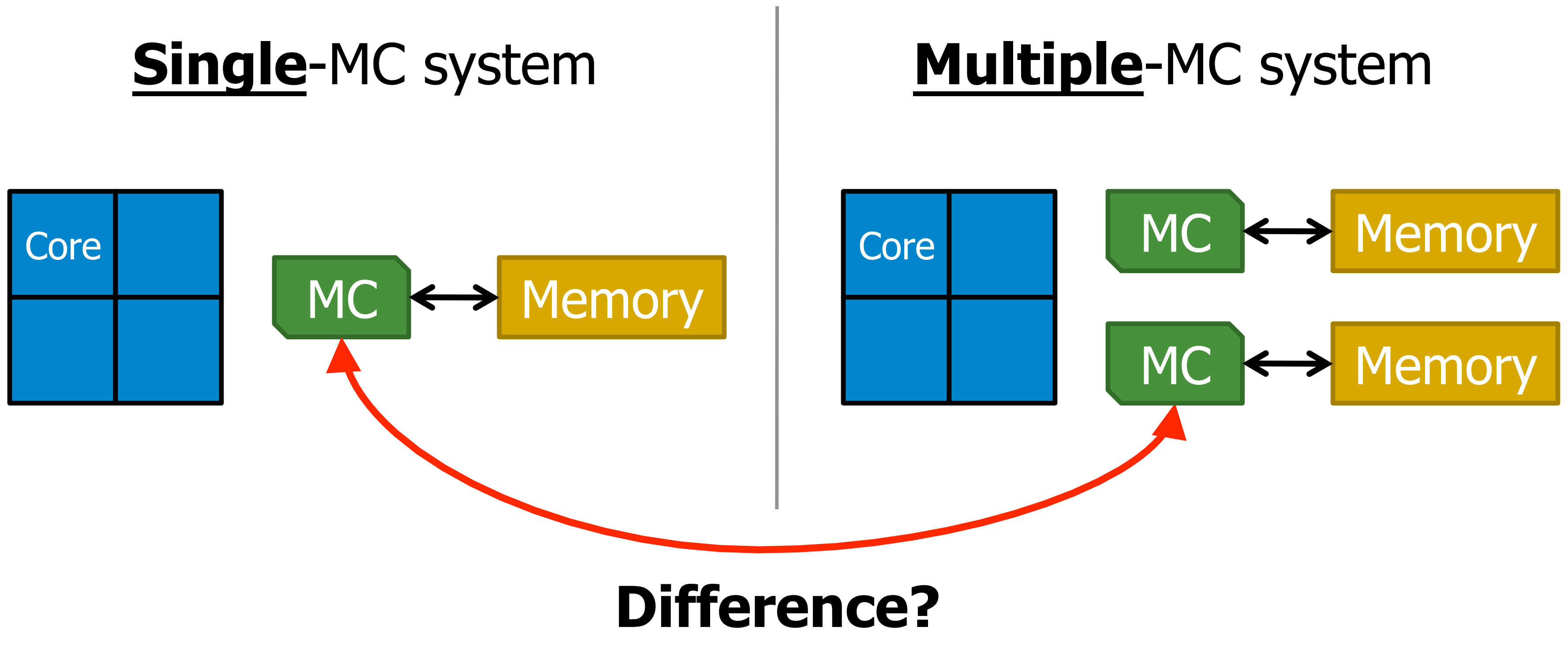
- **Configurable by system software**
  - To enforce thread priorities and QoS/fairness policies

Multiple memory controllers

- **Scalable to a large number of controllers**
  - Should not require significant coordination between controllers

No previous scheduling algorithm satisfies
all these requirements

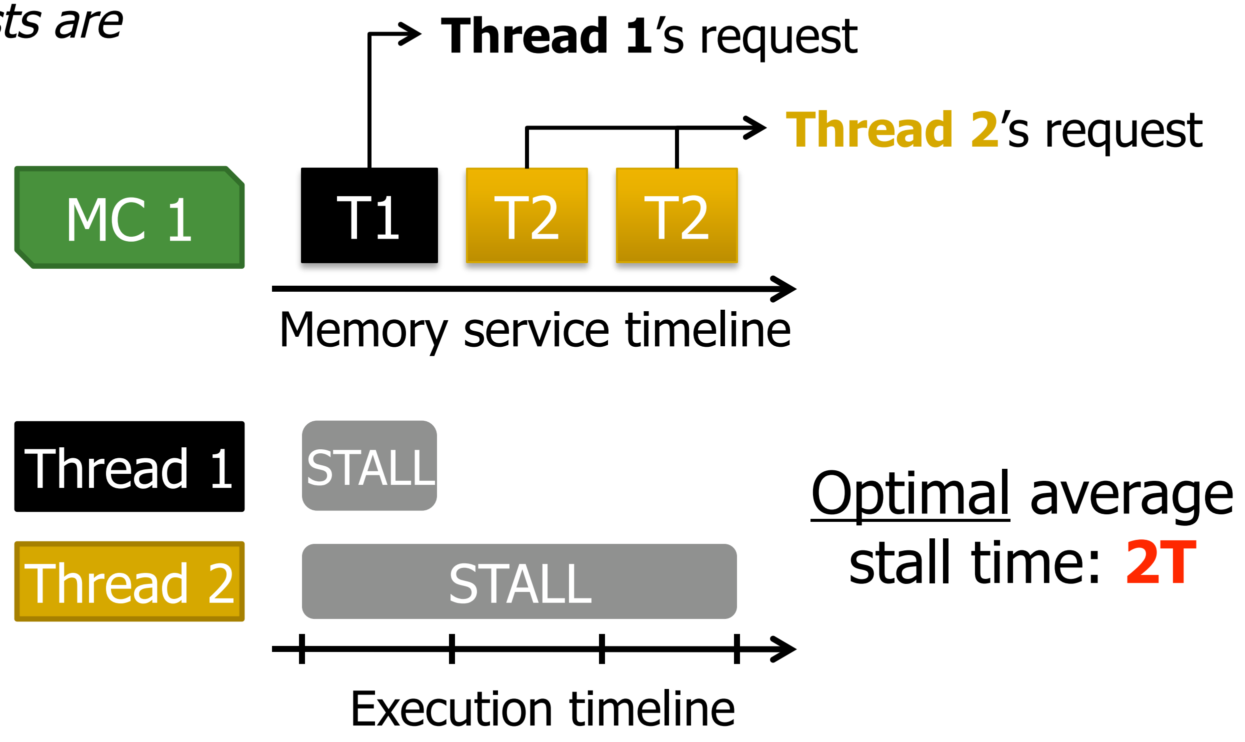
Multiple Memory Controllers



The need for coordination

Thread Ranking in Single-MC

Assume all requests are to the same bank



# of requests: Thread 1 < Thread 2

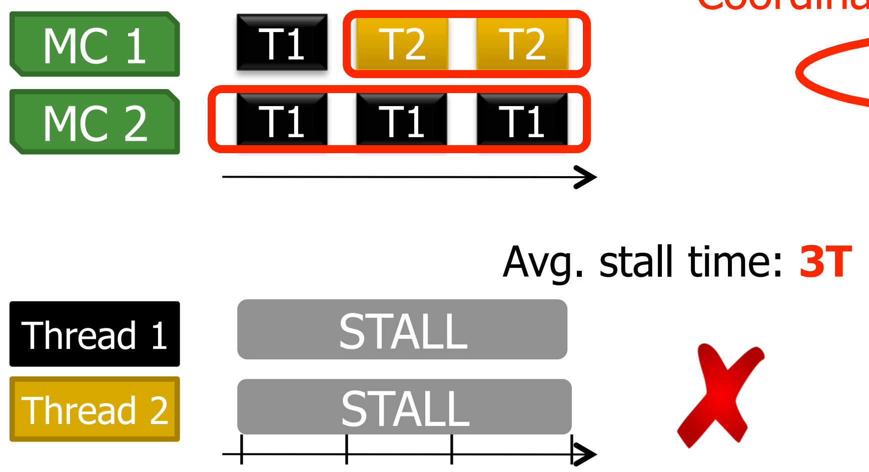
Thread 1 ➔ Shorter job

Thread ranking: Thread 1 > Thread 2

Thread 1 ➔ Assigned higher rank

Thread Ranking in Multiple-MC

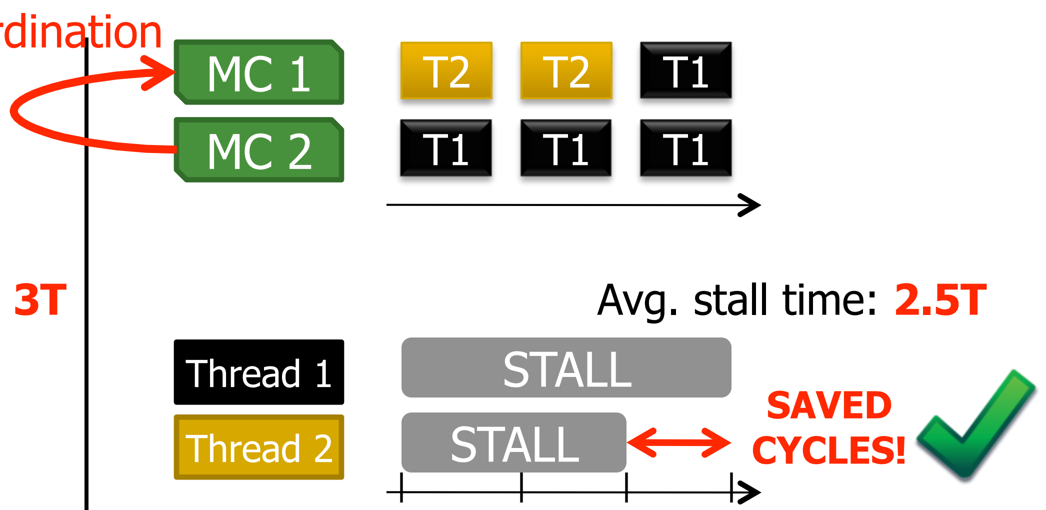
Uncoordinated



MC 1's shorter job: **Thread 1**
Global shorter job: **Thread 2**

MC 1 **incorrectly** assigns higher rank to **Thread 1**

Coordinated

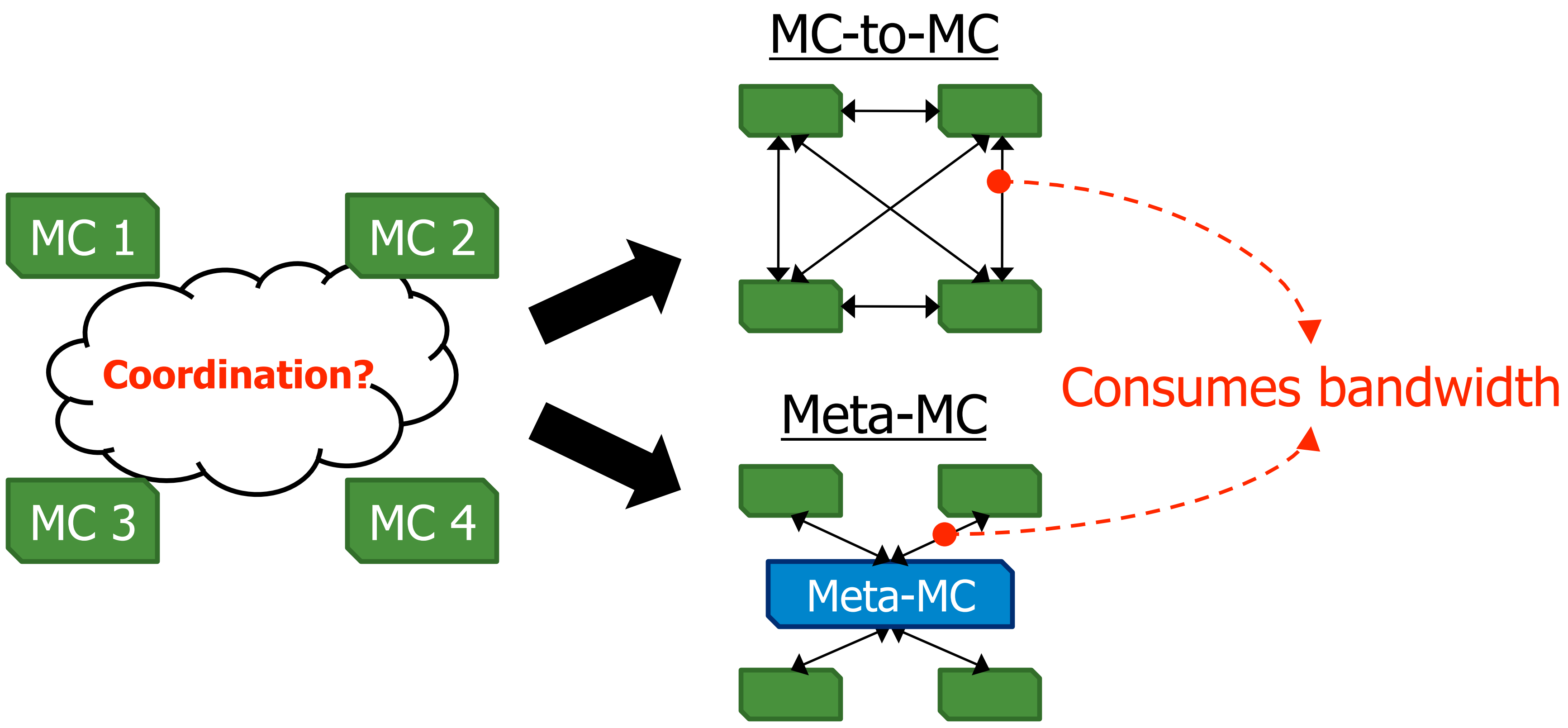


Global shorter job: **Thread 2**

MC 1 **correctly** assigns higher rank to **Thread 2**

Coordination → Better scheduling decisions

Coordination Limits Scalability



To be scalable, coordination should:
- exchange little information
- occur infrequently

The Problem and Our Goal

Problem:

- Previous best memory scheduling algorithms are not scalable to many controllers
  - Not designed for multiple MCs
  - Low performance or require significant coordination

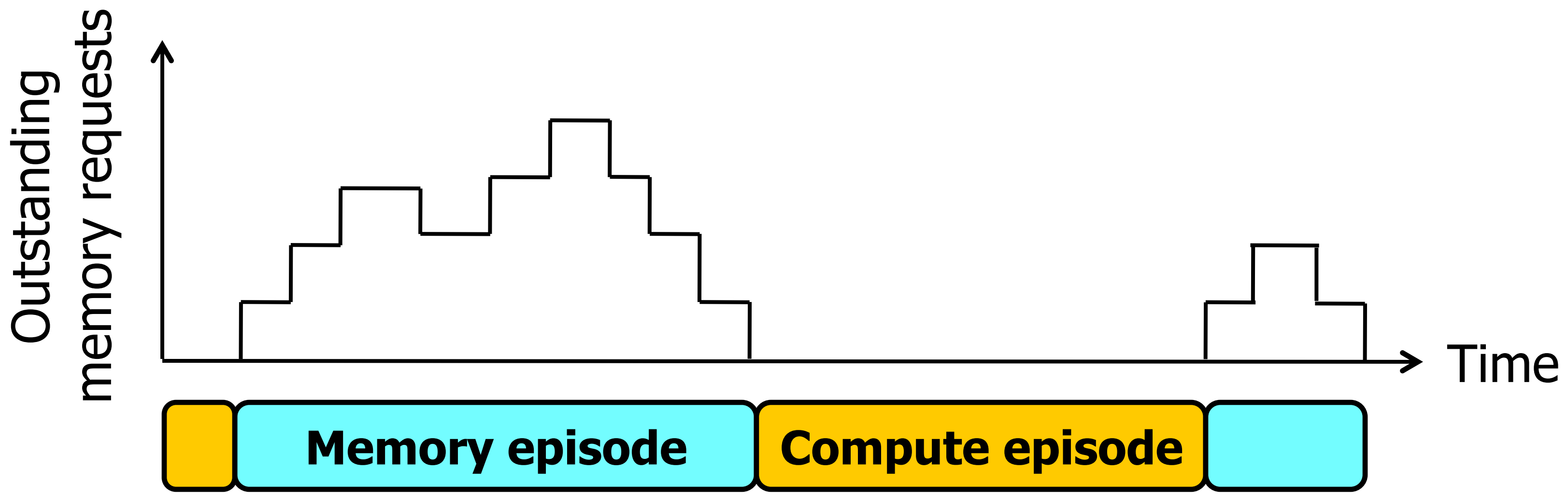
Our Goal:

- Fundamentally redesign the memory scheduling algorithm such that it
  - Provides high system throughput
  - Requires little or no coordination among MCs

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**



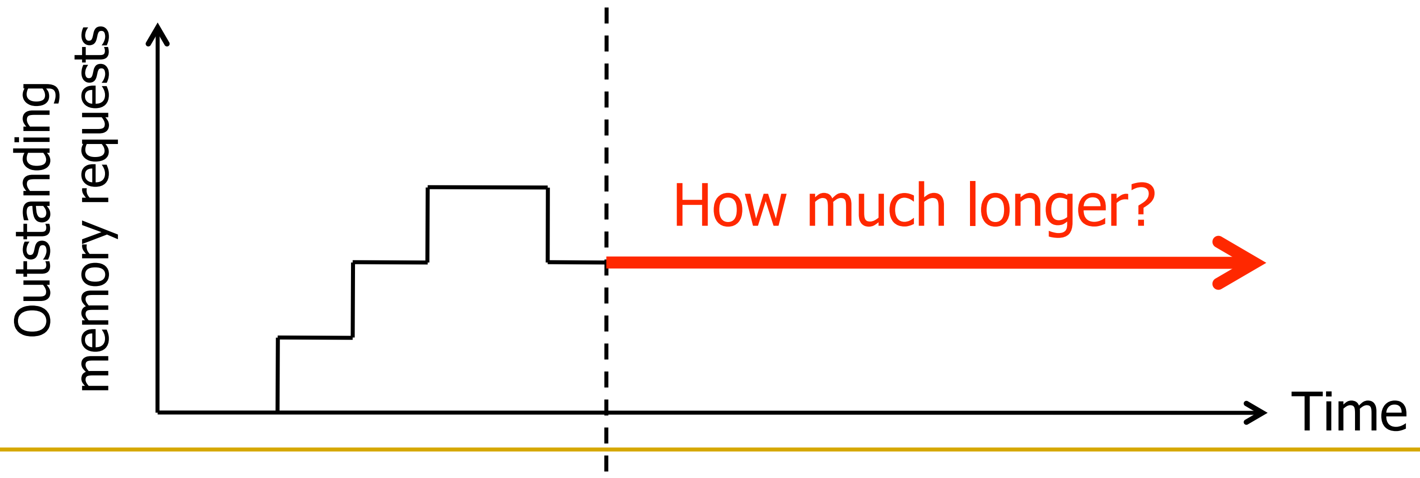
<u>Goal</u>: Minimize time spent in memory episodes

How to Minimize Memory Episode Time

Prioritize thread whose memory episode will end the soonest

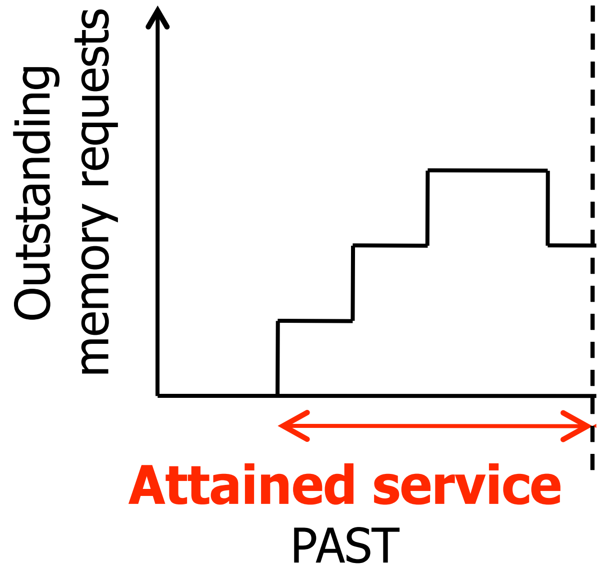
- Minimizes time spent in memory episodes across all threads
- Supported by queueing theory:
  - Shortest-Remaining-Processing-Time scheduling is optimal in single-server queue

Remaining length of a memory episode?



Predicting Memory Episode Lengths

We discovered: past is excellent predictor for future

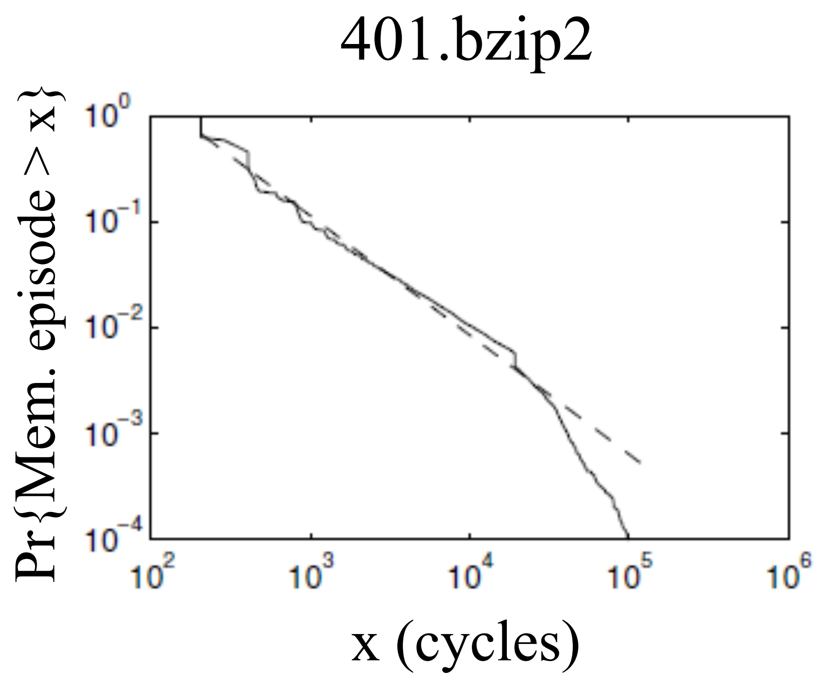


Large **attained service** ➔ Large expected **remaining service**

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

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

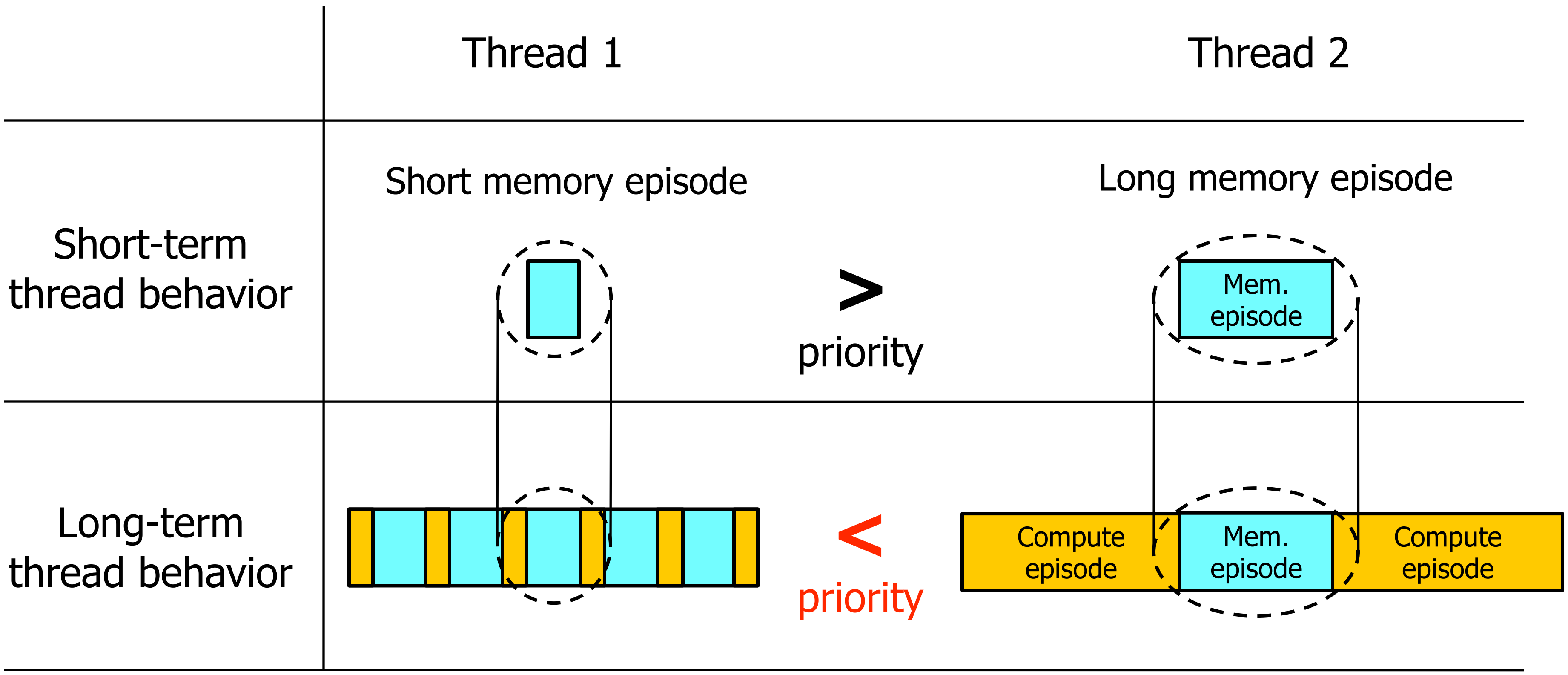
Prioritize the memory episode with least-**attained**-service

Least-attained-service (LAS) scheduling:

Minimize memory episode time

However, LAS does not consider long-term thread behavior

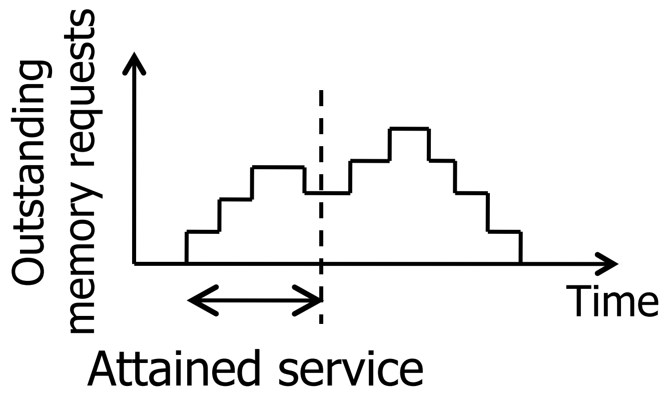
Long-Term Thread Behavior



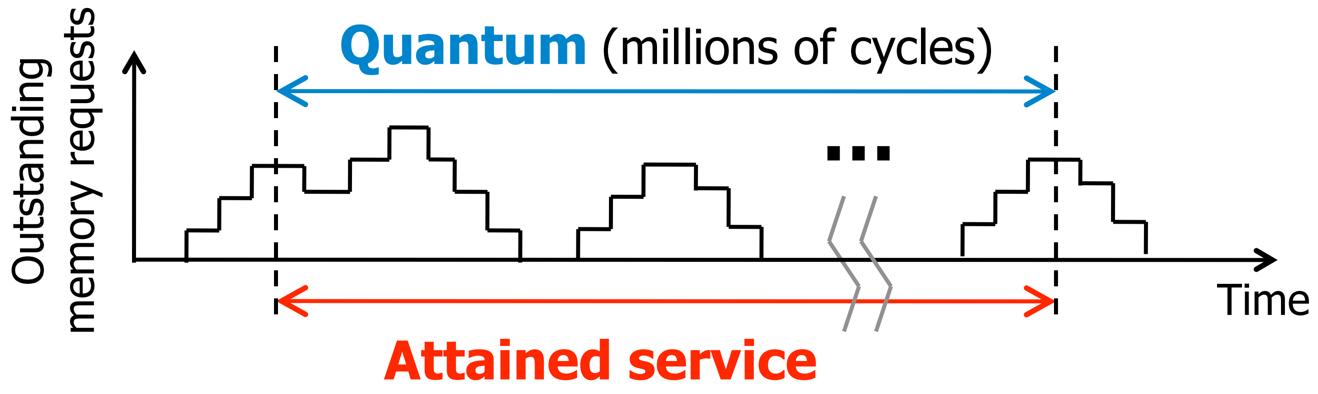
Prioritizing Thread 2 is more beneficial: results in very long stretches of compute episodes

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)

Quantum-Based 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?

ATLAS Coordination Mechanism

During a quantum:

- Each MC increments the local AS of each thread

End of a quantum:

- Each MC sends local AS of each thread to centralized meta-MC
- Meta-MC accumulates local AS and calculates ranking
- Meta-MC broadcasts ranking to all MCs
  - ➔ **Consistent thread ranking**

Coordination Cost in ATLAS

How costly is coordination in ATLAS?

	ATLAS	PAR-BS (previous best work [ISCA08])
How often?	**Very infrequently** ✓ Every quantum boundary (10 M cycles)	**Frequently** Every batch boundary (thousands of cycles)
Sensitive to coordination latency?	**Insensitive** ✓ Coordination latency << Quantum length	**Sensitive** Coordination latency ~ Batch length

Properties of ATLAS

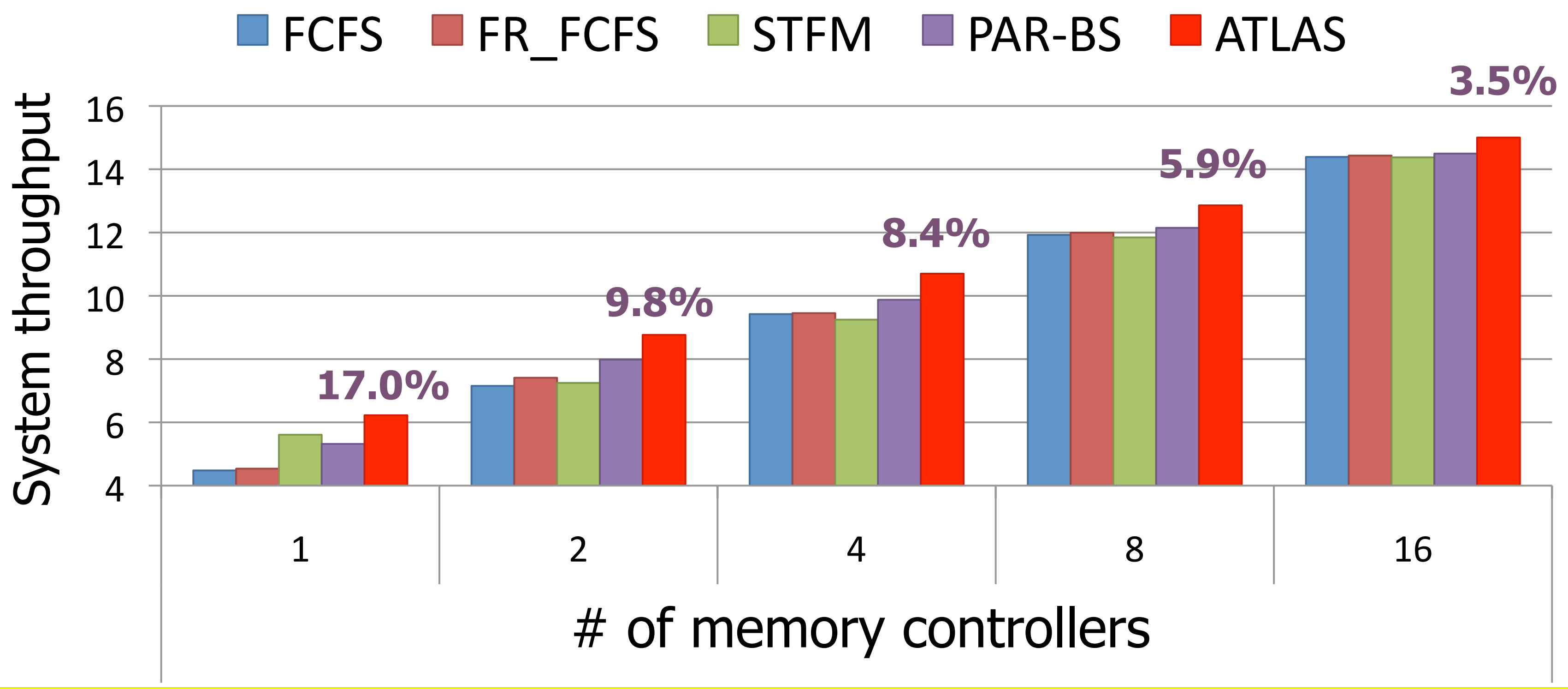
Goals	Properties of ATLAS
▪ Maximize system performance	▪ LAS-ranking ▪ Bank-level parallelism ▪ Row-buffer locality
▪ Scalable to large number of controllers	▪ Very infrequent coordination
▪ Configurable by system software	▪ Scale attained service with thread weight
	▪ **Low complexity**: Attained service requires a single counter per thread in each MC (<9K bits for 24-core, 4-MC)

ATLAS Evaluation Methodology

- 4, 8, 16, 24, 32-core systems
  - 5 GHz processor, 128-entry instruction window
  - 512 Kbyte per-core private L2 caches
- 1, 2, 4, 8, 16-MC systems
  - 128-entry memory request buffer
  - 4 banks, 2Kbyte row buffer
  - 40ns (200 cycles) row-hit round-trip latency
  - 80ns (400 cycles) row-conflict round-trip latency
- Workloads
  - Multiprogrammed SPEC CPU2006 applications
  - 32 program combinations for 4, 8, 16, 24, 32-core experiments

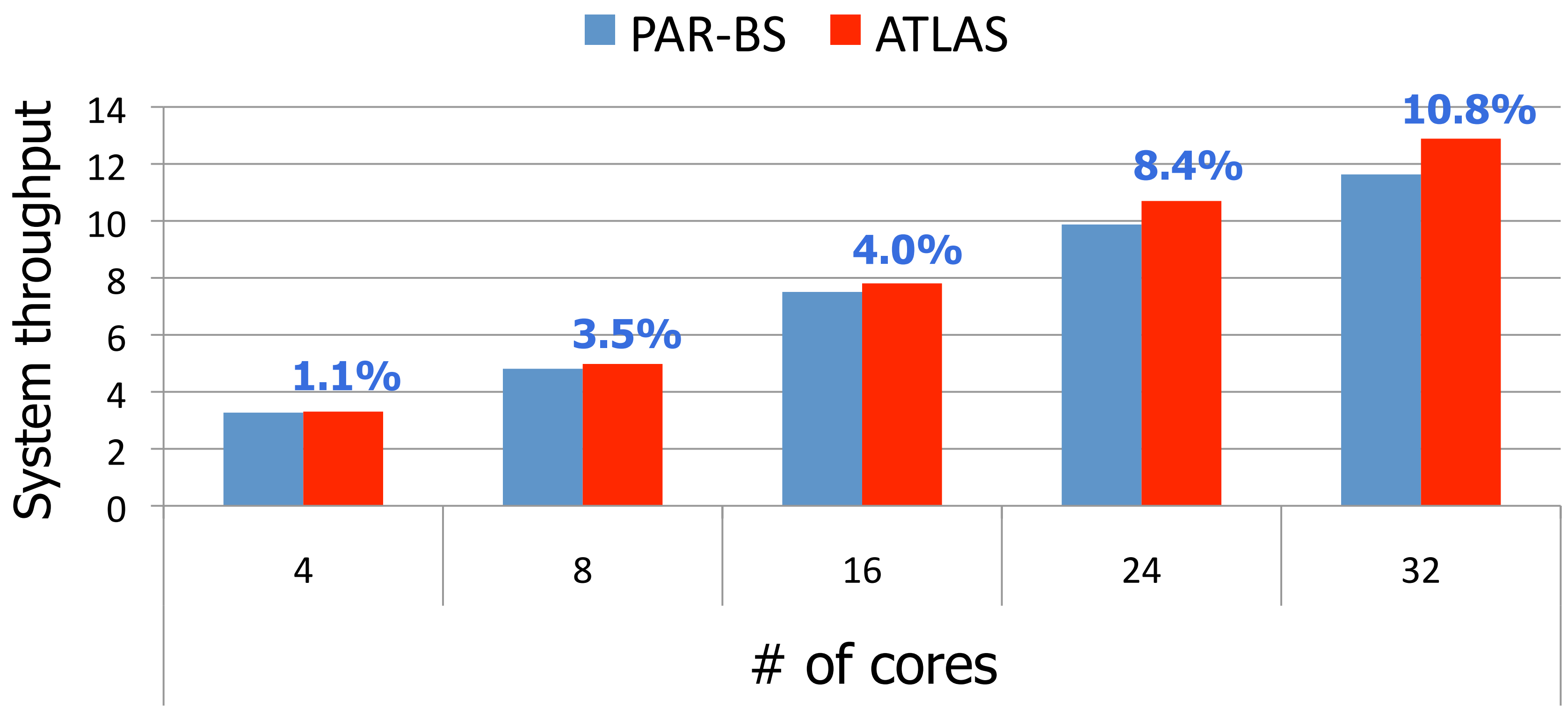
System Throughput: 24-Core System

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

System Software Support

- ATLAS enforces system priorities, or thread weights
  - Linear relationship between thread weight and speedup

$$TotalAS_i = \alpha TotalAS_{i-1} + \frac{(1-\alpha)}{thread\_weight} AS_i$$

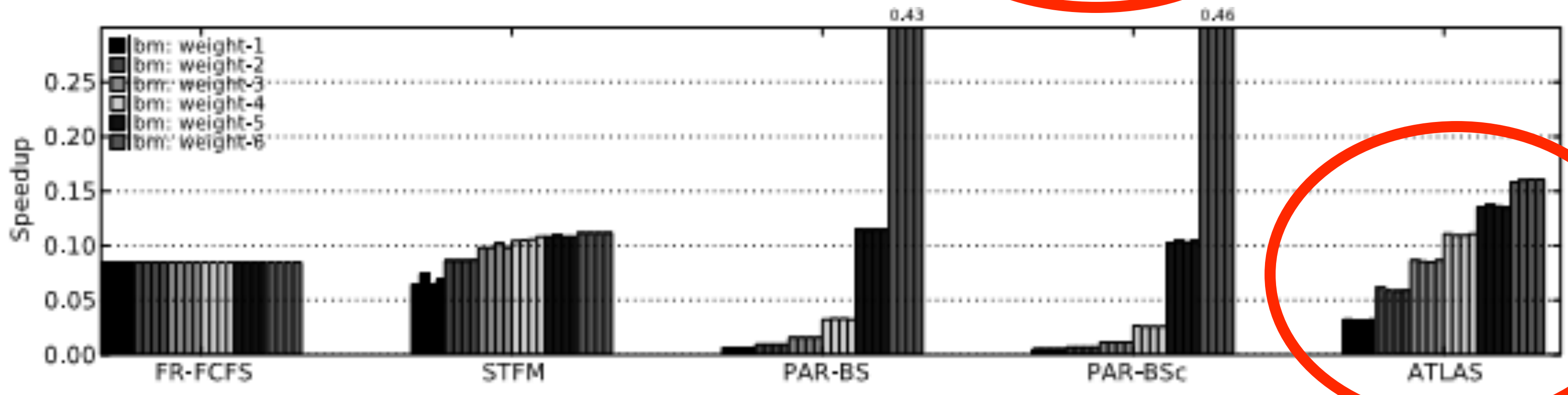


Figure 14. Evaluation of ATLAS vs. PAR-BS and STFM with different thread weights

ATLAS: Summary

- Existing memory scheduling algorithms are low performance
  - Especially with multiple memory controllers

- ATLAS is a fundamentally new approach to memory scheduling
  - **Scalable:** Thread ranking decisions at coarse-grained intervals
  - **High-performance:** Minimizes system time spent in memory episodes (Least Attained Service scheduling principle)
  - **Configurable:** Enforces thread priorities

- ATLAS provides the highest system throughput compared to five previous scheduling algorithms
  - Performance benefit increases as the number of cores increases

Agenda

- Technology, Application, Architecture Trends
- Requirements from the Memory Hierarchy
- The Problem: Interference in Memory System
- Two Solution Approaches
  - Smart resources: ATLAS Memory Scheduler
  - Dumb resources: Fairness via Source Throttling
- Future Work
- Conclusions

Designing QoS-Aware Memory Systems: Approaches

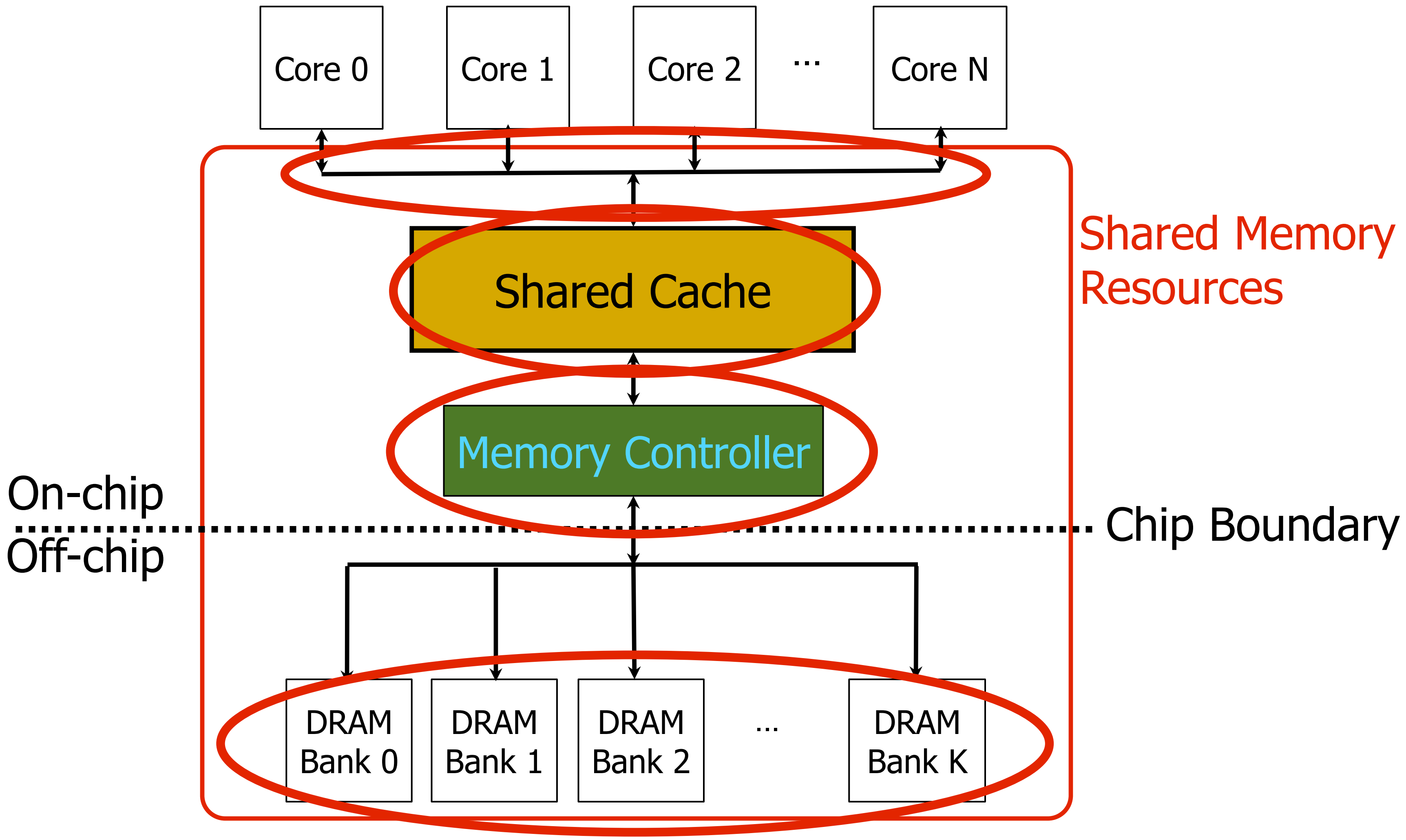
- Smart resources: Design each shared resource to have a configurable interference control/reduction mechanism
  - Fair/QoS-aware memory schedulers, interconnects, caches, arbiters
    - Fair memory schedulers [Mutlu MICRO 2007], parallelism-aware memory schedulers [Mutlu ISCA 2008], ATLAS memory scheduler [Kim et al. HPCA 2010]
    - Application-aware on-chip networks [Das et al. MICRO 2009, ISCA 2010, Grot et al. MICRO 2009]

- Dumb resources: Keep each resource free-for-all, but control access to memory system at the cores/sources
  - Estimate interference/slowdown in the entire system and throttle cores that slow down others
    - Fairness via Source Throttling [Ebrahimi et al., ASPLOS 2010]
    - Coordinated Prefetcher Throttling [Ebrahimi et al., MICRO 2009]

Fairness via Source Throttling

Ebrahimi et al., "Fairness via Source Throttling: A Configurable and High-Performance Fairness Substrate for Multi-Core Memory Systems," ASPLOS 2010.

Many Shared Resources



Motivation for Source Throttling

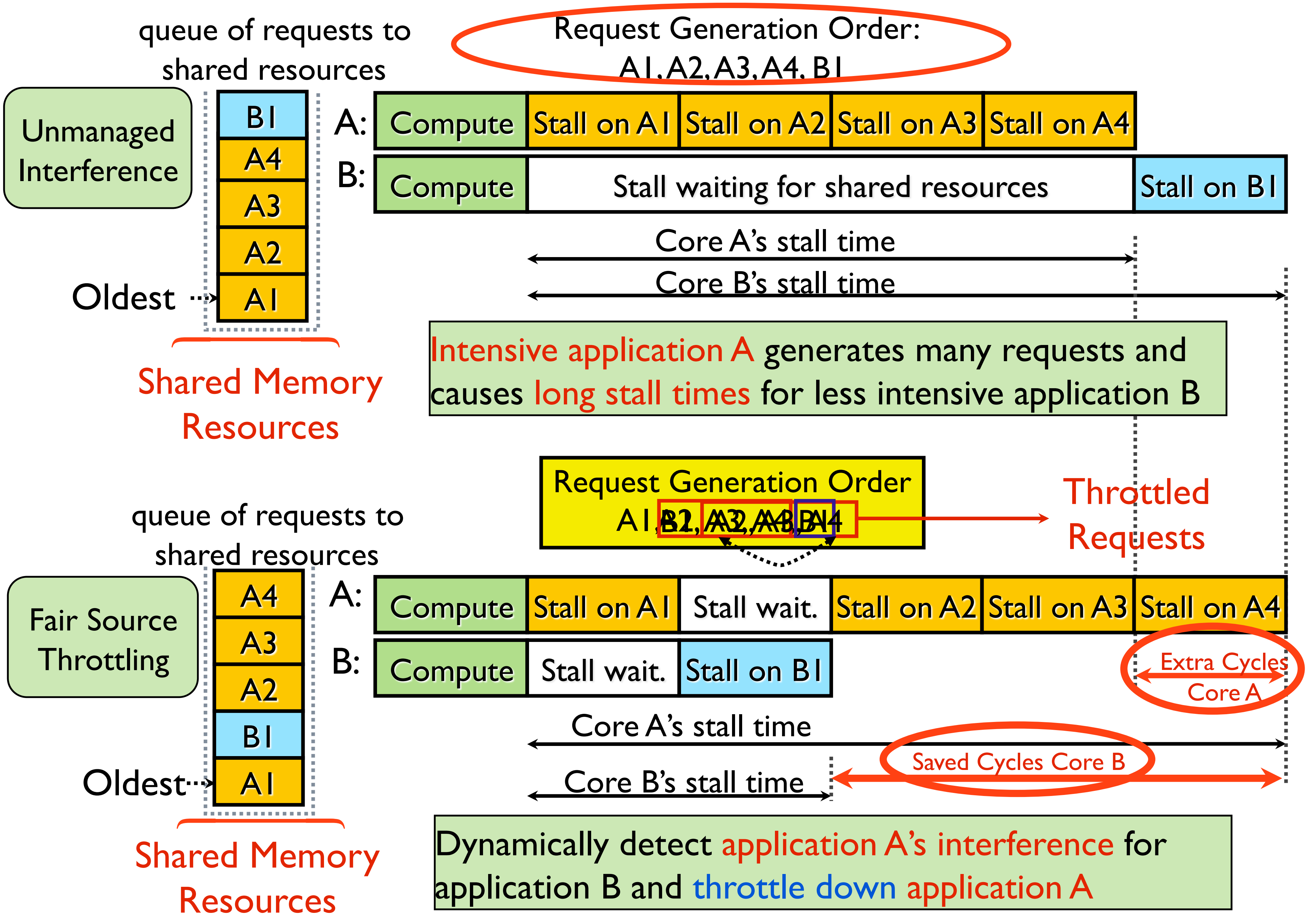
- Partitioning (fairness/QoS) mechanisms in each resource might be difficult to get right (initially)
- Independent partitioning mechanisms in caches, interconnect, and memory can contradict each other
- Approaches that coordinate interaction among techniques for different resources require complex implementations

Our Goal: Enable fair sharing of the entire memory system by dynamically detecting and controlling interference in a coordinated manner

An Alternative Approach

- Manage inter-thread interference at the cores, 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



Intensive application A generates many requests and causes long stall times for less intensive application B

Dynamically detect application A's interference for application B and throttle down application A

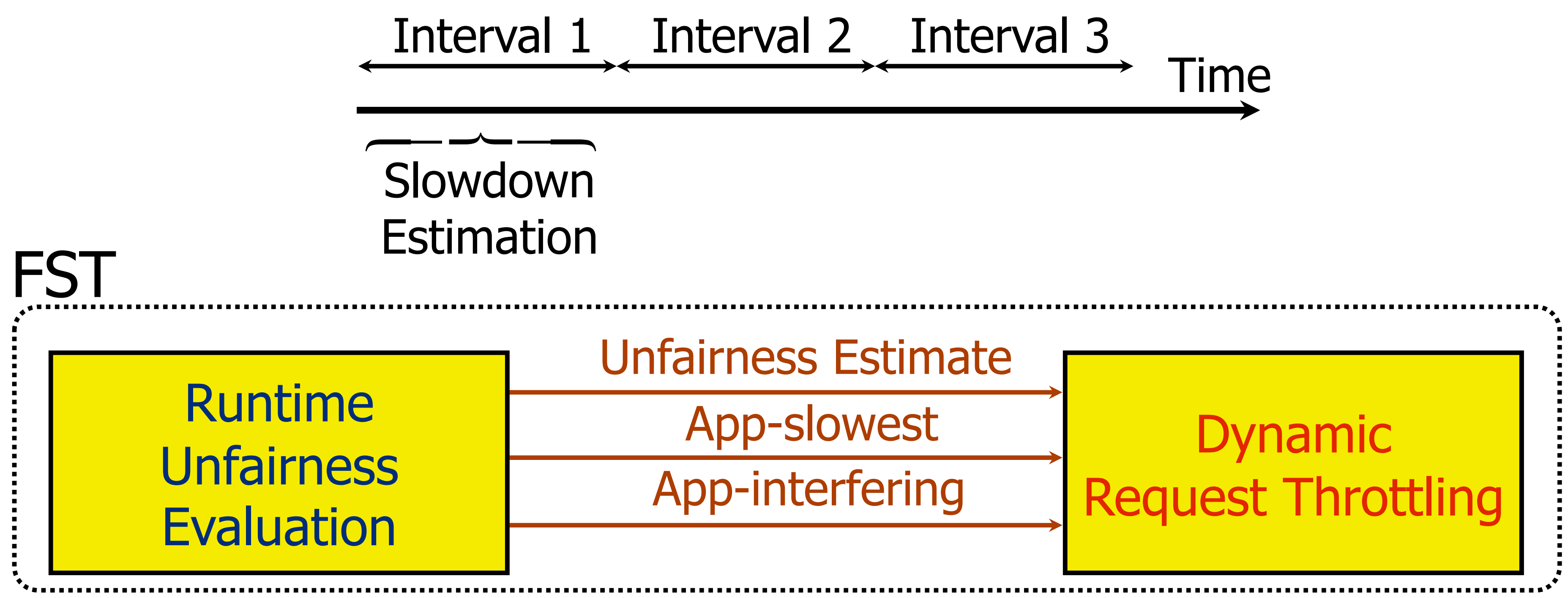
Fairness via Source Throttling (FST)

- Two components (interval-based)

- Run-time unfairness evaluation (in hardware)
  - Dynamically estimates the unfairness in the memory system
  - Estimates which application is slowing down which other

- Dynamic request throttling (hardware/software)
  - Adjusts how aggressively each core makes requests to the shared resources
  - Throttles down request rates of cores causing unfairness
    - Limit miss buffers, limit injection rate

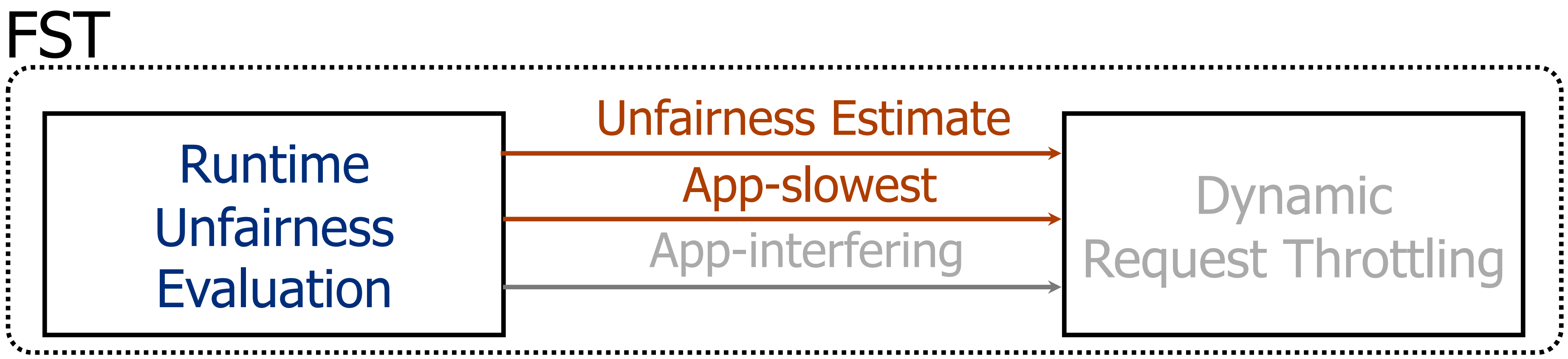
Fairness via Source Throttling (FST)



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
 2-Throttle up App-slowest
}
```

Fairness via Source Throttling (FST)



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
 2-Throttle up App-slowest
}
```

Estimating System Unfairness

- $$\text{Unfairness} = \frac{\text{Max\{Slowdown i\} over all applications i}}{\text{Min\{Slowdown i\} over all applications i}}$$

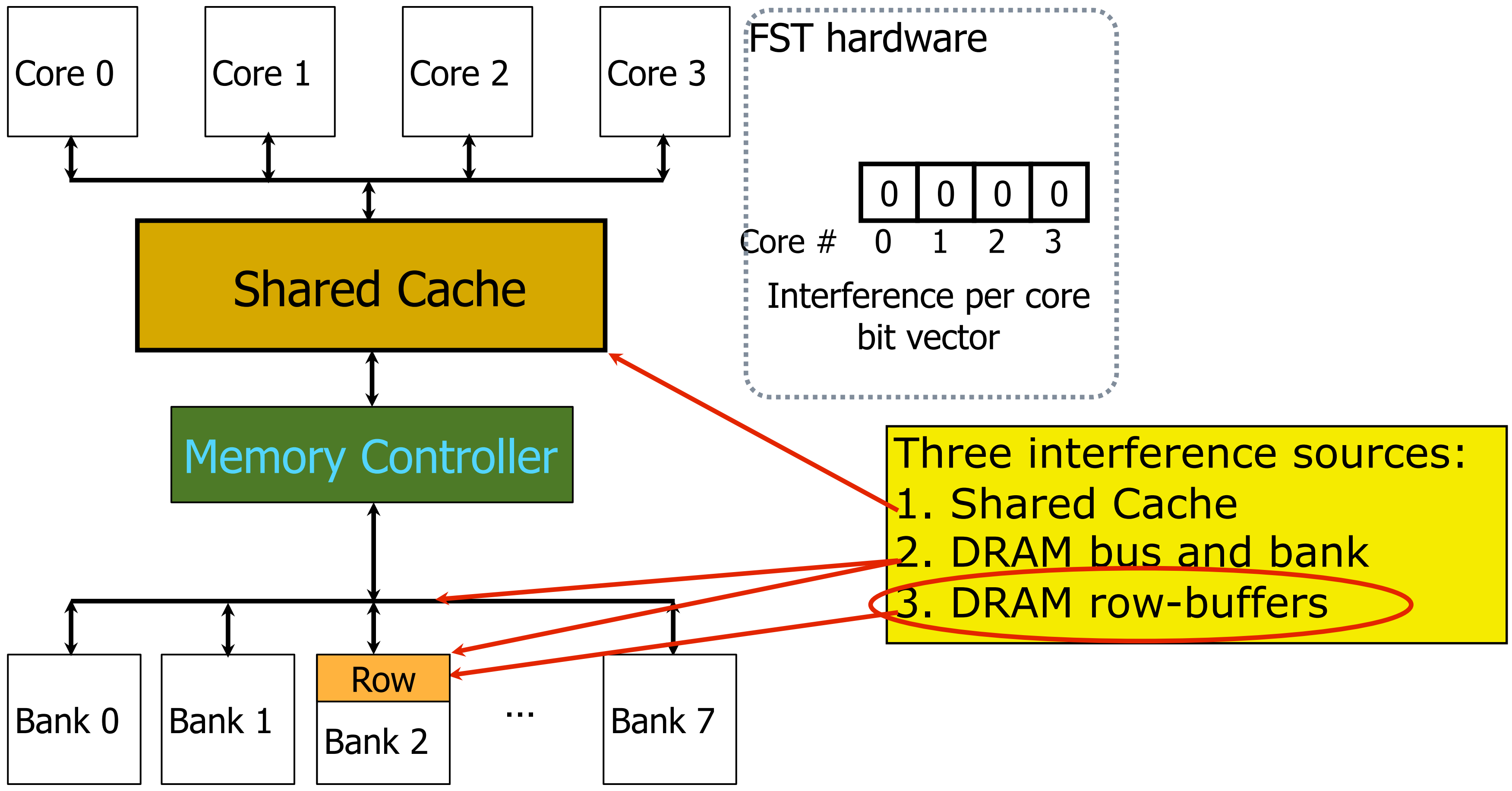
- $$\text{Slowdown of application i} = \frac{T_i^{Shared}}{T_i^{Alone}}$$

- How can T_i^{Alone} be estimated in shared mode?

- T_i^{Excess} is the number of extra cycles it takes application i to execute due to interference

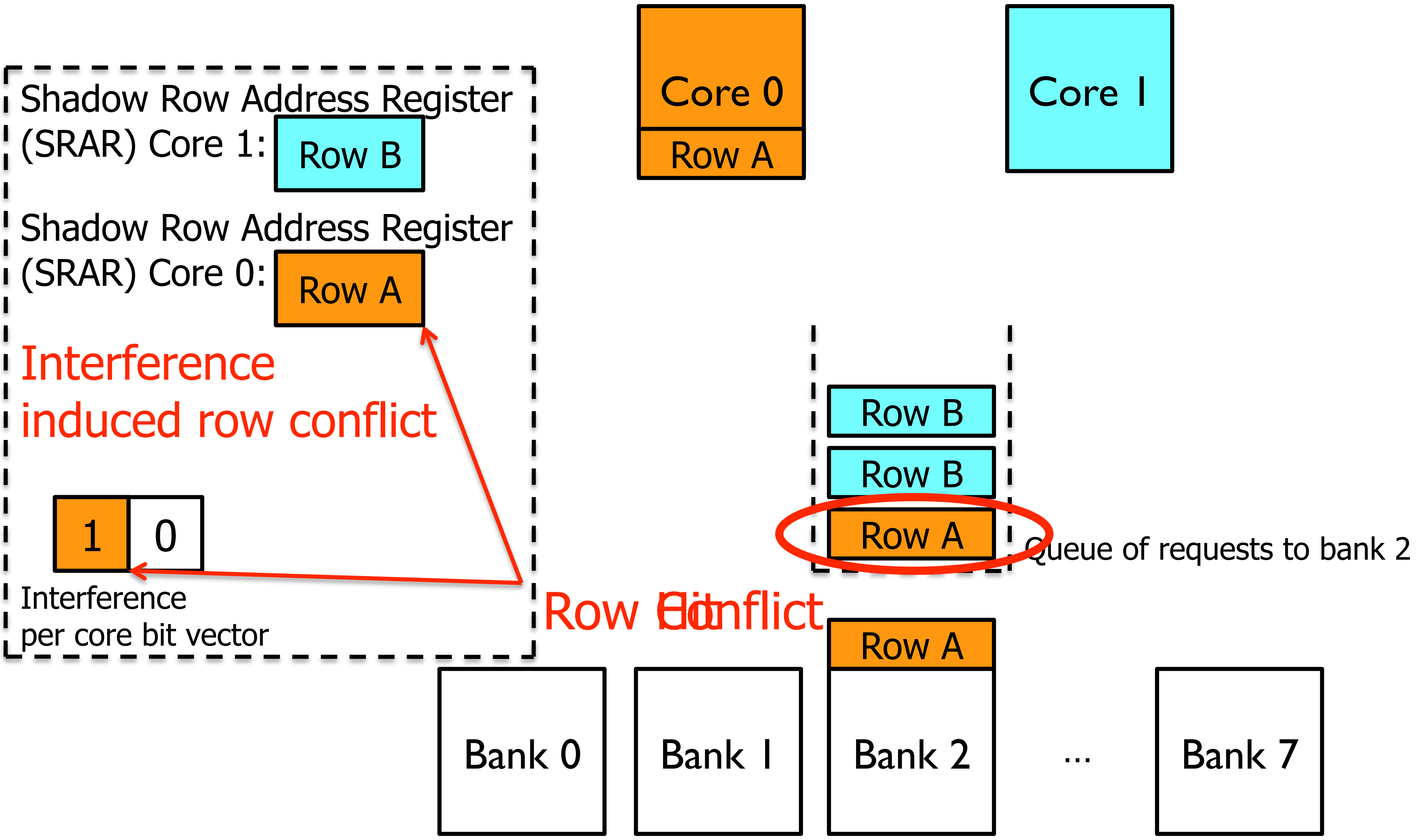
- $$T_i^{Alone} = T_i^{Shared} - \boxed{T_i^{Excess}}$$

Tracking Inter-Core Interference

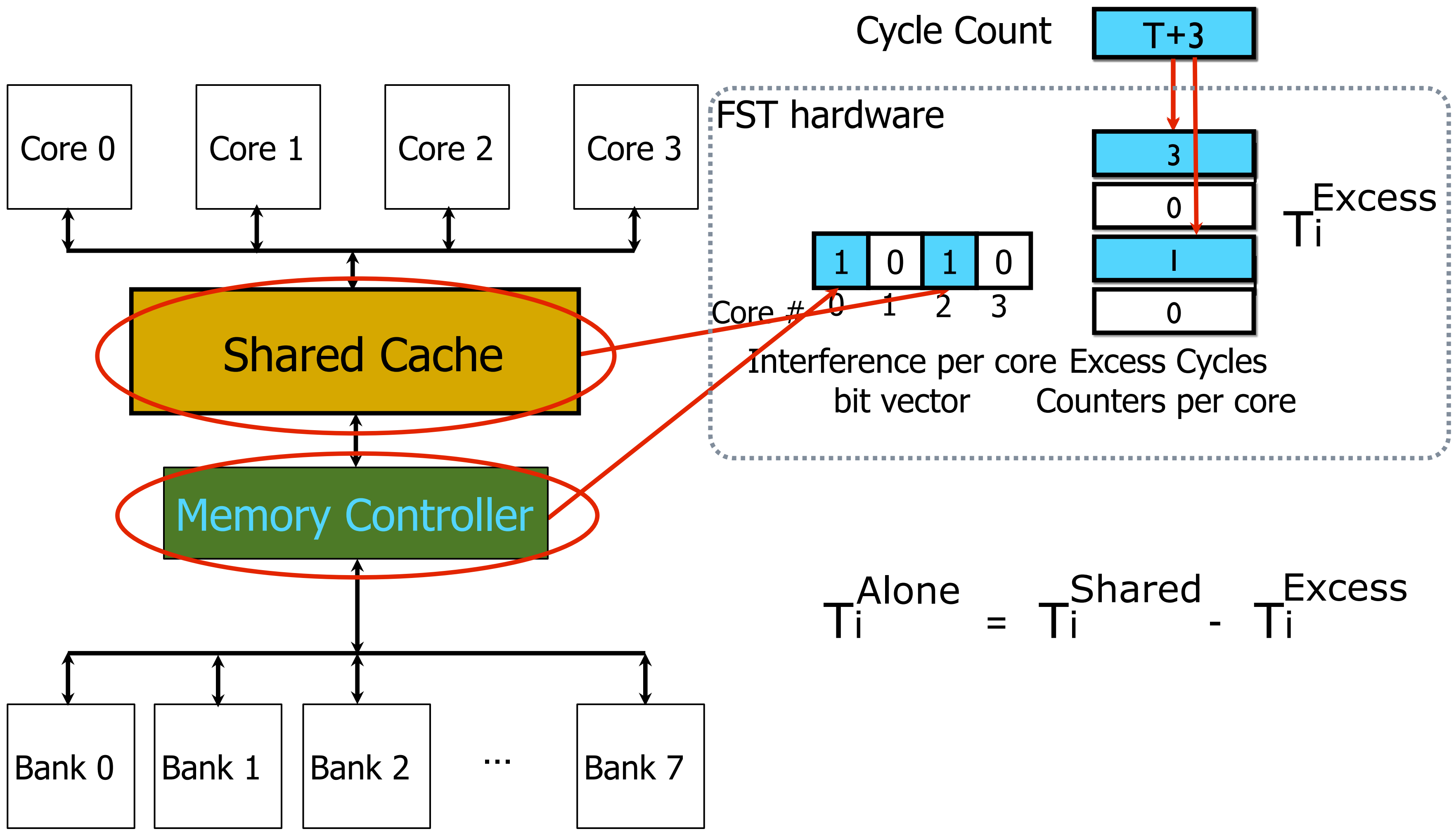


Three interference sources:
1. Shared Cache
2. DRAM bus and bank
3. DRAM row-buffers

Tracking DRAM Row-Buffer Interference

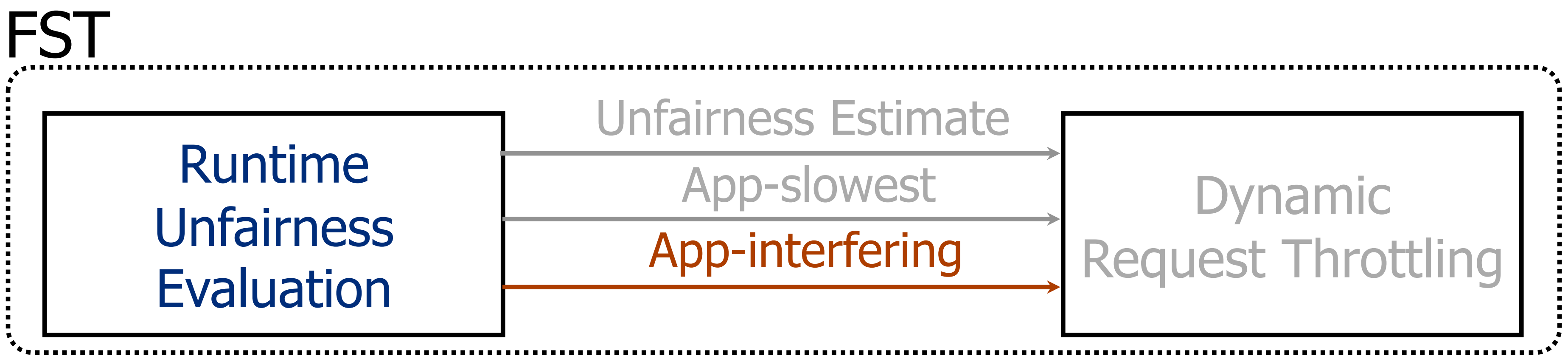

Shadow Row Address Register (SRAR) Core 1:
Row B
Shadow Row Address Register (SRAR) Core 0:
Row A
Interference induced row conflict
1
0
Interference per core bit vector
Core 0
Row A
Core 1
Row B
Row B
Row A
Queue of requests to bank 2
Row Hit
Row Conflict
Row A
Bank 0
Bank 1
Bank 2
…
Bank 7

Tracking Inter-Core Interference



$$T_i^{Alone} = T_i^{Shared} - T_i^{Excess}$$

Fairness via Source Throttling (FST)

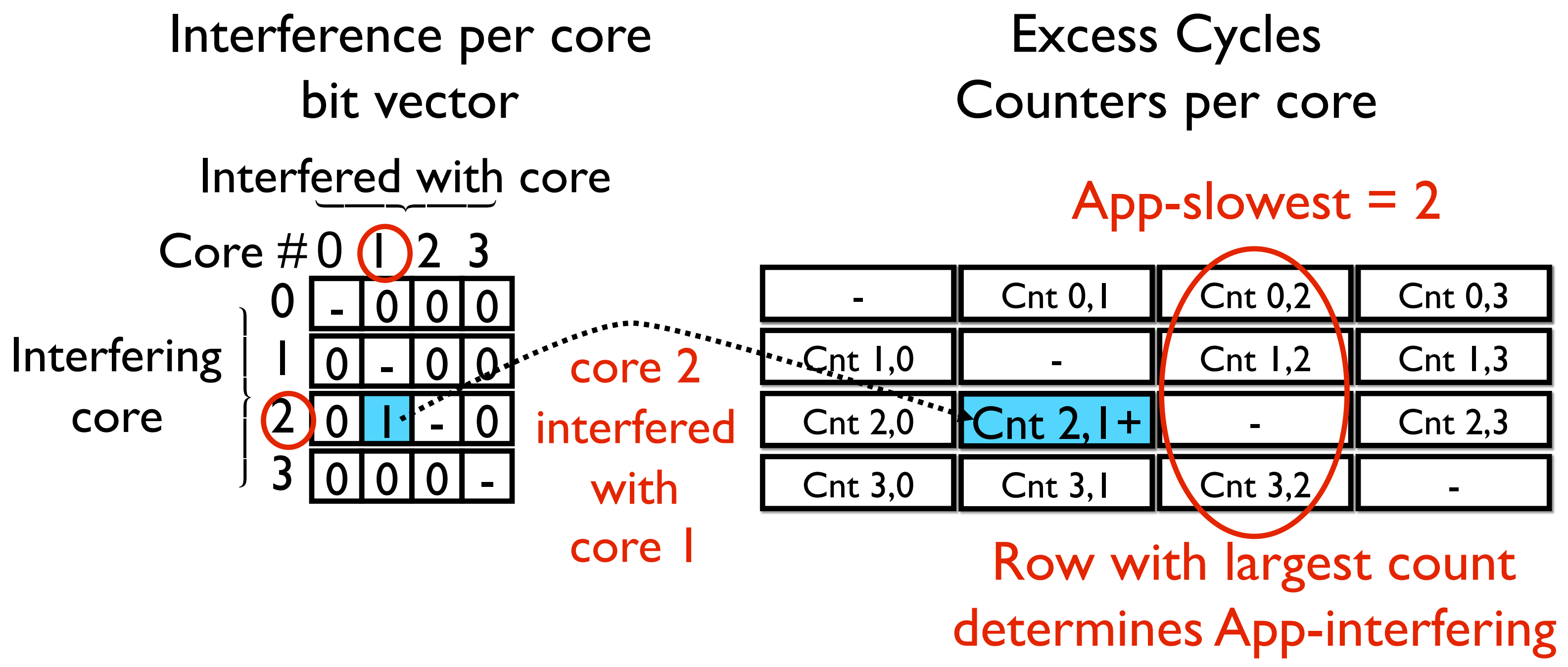


1- Estimating system unfairness
2- Find app. with the highest slowdown (App-slowest)
3- Find app. causing most interference for App-slowest (App-interfering)

```
if (Unfairness Estimate >Target)
{
 1-Throttle down App-interfering
 2-Throttle up App-slowest
}
```

Tracking Inter-Core Interference

- To identify App-interfering, for each core i
  - FST separately tracks interference caused by each core j (j ≠ i)

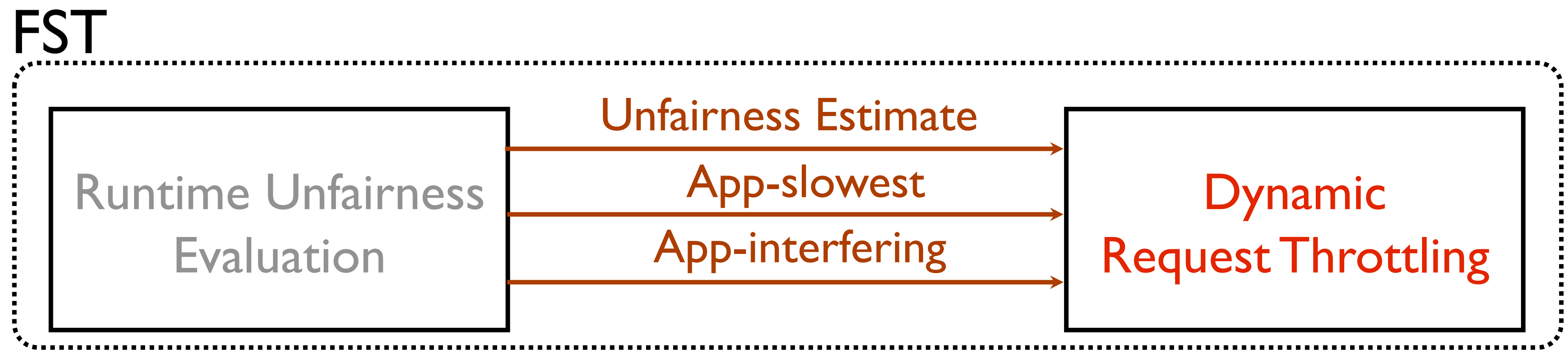


Interference per core bit vector
Interfered with core
Core #
Interfering core
core 2 interfered with core 1
Excess Cycles Counters per core
App-slowest = 2
Row with largest count determines App-interfering

Core #	0	1	2	3
0	-	0	0	0
1	0	-	0	0
2	0	1	-	0
3	0	0	0	-

-	Cnt 0,1	Cnt 0,2	Cnt 0,3
Cnt 1,0	-	Cnt 1,2	Cnt 1,3
Cnt 2,0	Cnt 2,1+	-	Cnt 2,3
Cnt 3,0	Cnt 3,1	Cnt 3,2	-

Fairness via Source Throttling (FST)



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
 2-Throttle up App-slowest
}
```

Dynamic Request Throttling

- Goal: Adjust how aggressively each core makes requests to the shared memory system

- Mechanisms:
  - Miss Status Holding Register (MSHR) quota
    - Controls the number of concurrent requests accessing shared resources from each application
  - Request injection frequency
    - Controls how often memory requests are issued to the last level cache from the MSHRs

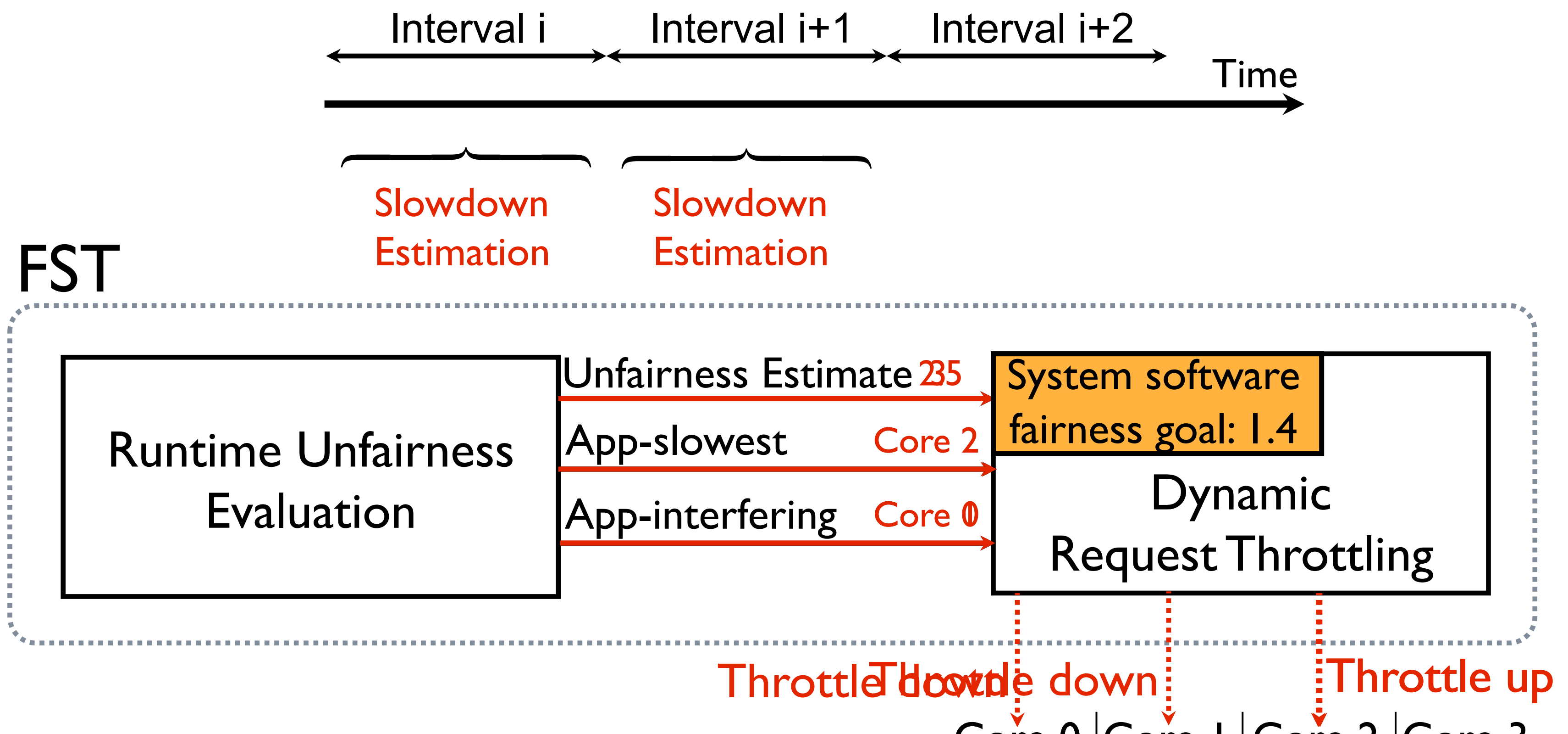
Dynamic Request Throttling

- Throttling level assigned to each core determines both MSHR quota and request injection rate

Throttling level	MSHR quota	Request Injection Rate
100%	128	Every cycle
50%	64	Every other cycle
25%	32	Once every 4 cycles
10%	12	Once every 10 cycles
5%	6	Once every 20 cycles
4%	5	Once every 25 cycles
3%	3	Once every 30 cycles
2%	2	Once every 50 cycles

Total # of MSHRs: 128

FST at Work



	Core 0	Core 1	Core 2	Core 3
Interval *i*	50%	100%	10%	100%
Interval *i* + 1	25%	100%	25%	100%
Interval *i* + 2	25%	50%	50%	100%

Throttling Levels

System Software Support

- Different fairness objectives can be configured by system software
  - Estimated Unfairness > Target Unfairness
  - Estimated Max Slowdown > Target Max Slowdown
  - Estimated Slowdown(i) > Target Slowdown(i)

- Support for thread priorities
  - Weighted Slowdown(i) =
    Estimated Slowdown(i) x Weight(i)

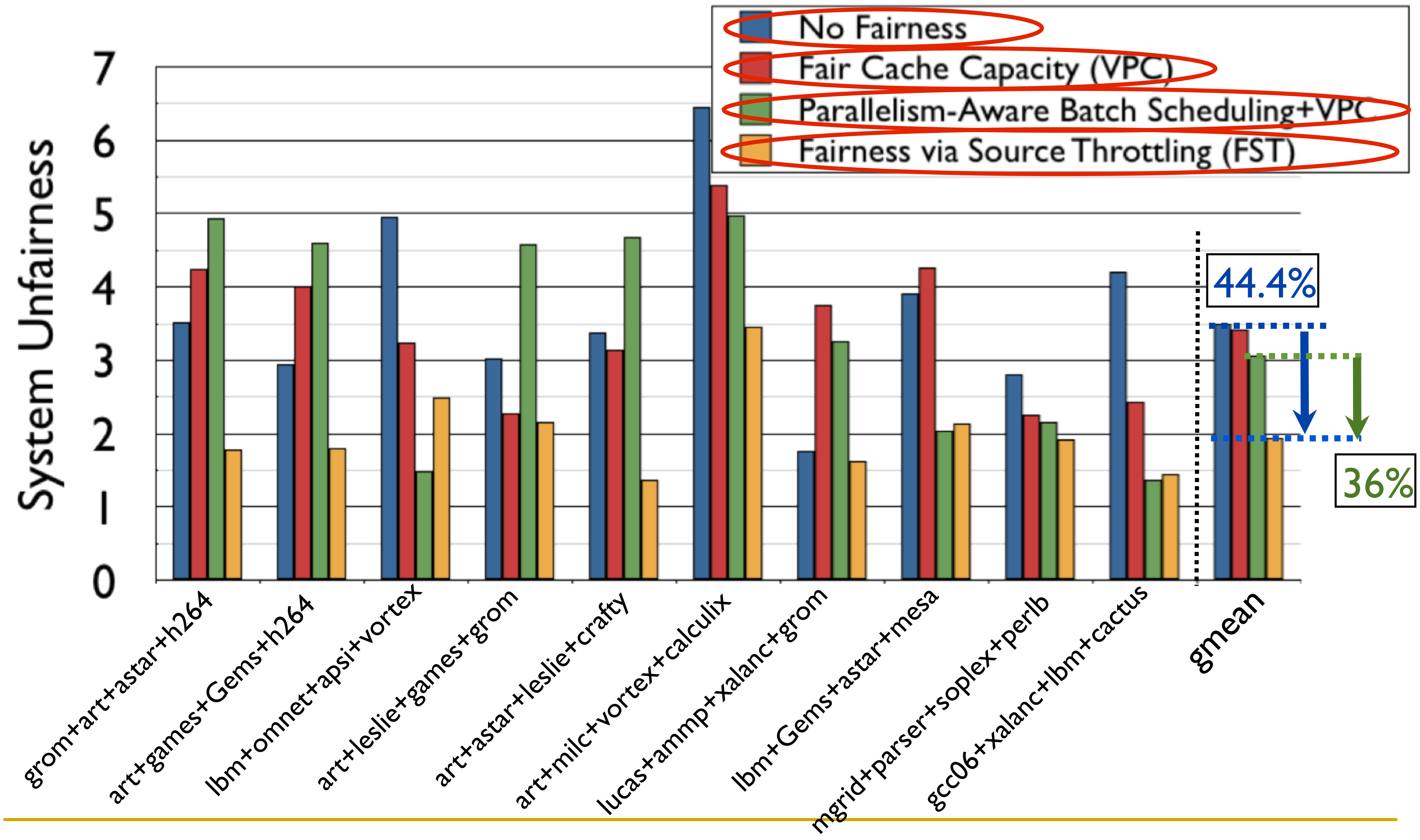
FST Hardware Cost

- Total storage cost required for 4 cores is ~12KB
- FST does not require any structures or logic that are on the processor’s critical path

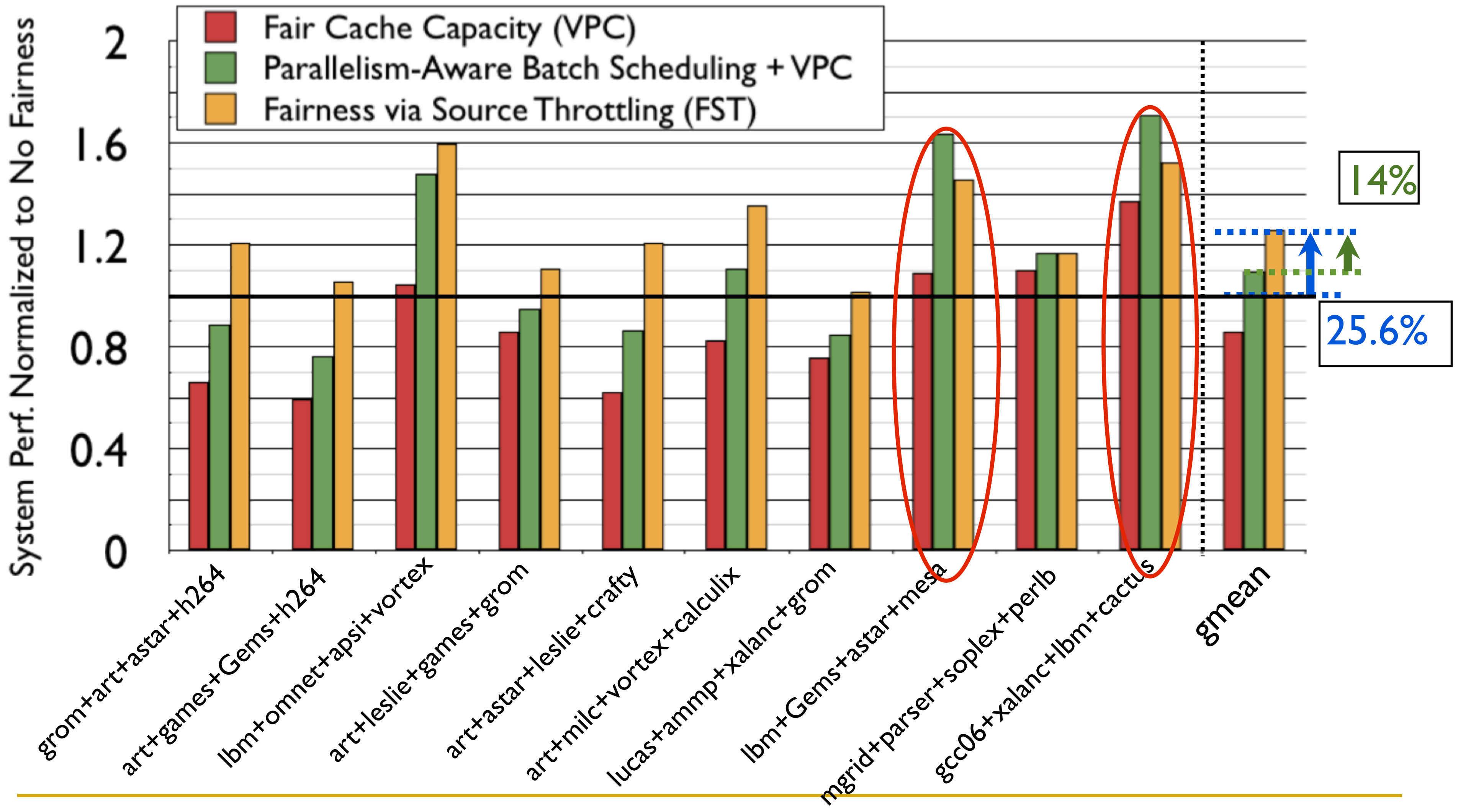
FST Evaluation Methodology

- x86 cycle accurate simulator
- Baseline processor configuration
    - Per-core
        - 4-wide issue, out-of-order, 256 entry ROB
    - Shared (4-core system)
        - 128 MSHRs
        - 2 MB, 16-way L2 cache
    - Main Memory
        - DDR3 1333 MHz
        - Latency of 15ns per command (tRP, tRCD, CL)
        - 8B wide core to memory bus

FST: System Unfairness Results



FST: System Performance Results



FST Summary

- Fairness via Source Throttling (FST) is a new fair and high-performance shared resource management approach for CMPs

- Dynamically monitors unfairness and throttles down sources of interfering memory requests

- Reduces the need for multiple per-resource interference reduction/control techniques

- Improves both system fairness and performance
- Incorporates thread weights and enables different fairness objectives

Agenda

- Technology, Application, Architecture Trends
- Requirements from the Memory Hierarchy
- The Problem: Interference in Memory System
- Two Solution Approaches
    - Smart resources: ATLAS Memory Scheduler
    - Dumb resources: Fairness via Source Throttling
- Future Work
- Conclusions

Ongoing/Future Work

- Combined approaches are even more powerful
  - Source throttling and resource-based interference control
- Interference control/reduction in on-chip networks
  - Application-aware prioritization mechanisms [Das et al., MICRO 2009, ISCA 2010]
  - Bandwidth partitioning mechanisms [Grot et al., MICRO 2009]
- Power partitioning in the shared memory system

Agenda

- Technology, Application, Architecture Trends
- Requirements from the Memory Hierarchy
- The Problem: Interference in Memory System
- Two Solution Approaches
  - Smart resources: ATLAS Memory Scheduler
  - Dumb resources: Fairness via Source Throttling
- Future Work
- Conclusions

Conclusions

- Many-core memory systems need scalable mechanisms to control and reduce application/thread interference

- Two approaches to solve this problem
  - Smart resources: ATLAS is a scalable memory access scheduling algorithm that intelligently prioritizes threads
  - Dumb resources: Fairness via Source Throttling is a generalized core throttling mechanism for fairness/performance

- Both approaches
  - Significantly improve system throughput
  - Configurable by the system software → enable QoS policies

Designing High-Performance and Fair Shared Multi-core Memory Systems: Two Approaches

Onur Mutlu
onur@cmu.edu

March 23, 2010

GSRC

Carnegie Mellon