# 18-742 Fall 2012
# Parallel Computer Architecture
# Lecture 8: More Asymmetry

Prof. Onur Mutlu
Carnegie Mellon University
9/24/2012

# Past Due: Review Assignments

- Due: Friday, September 21, 11:59pm.
- Smith, "Architecture and applications of the HEP multiprocessor computer system," SPIE 1981.
- Tullsen et al., "Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor," ISCA 1996.
- Chappell et al., "Simultaneous Subordinate Microthreading (SSMT)," ISCA 1999.
- Reinhardt and Mukherjee, "Transient Fault Detection via Simultaneous Multithreading," ISCA 2000.

# Reminder: Project Proposals

- Due: Tuesday, September 25, 11:59pm.

- What?
  - A clear, insightful writeup
  - Problem
  - Why is it important?
  - Your goal
  - Your solution idea
  - What have others done to solve the problem?
  - What are the advantages/disadvantages of your solution idea?
  - Your research and evaluation plan
- Clear goals for Milestones I, II, and final report

# New Review Assignments

- Due: Sunday, September 30, 11:59pm.
- Mutlu, "Some Ideas and Principles for Achieving Higher System Energy Efficiency," NSF Position Paper and Presentation 2012.
- Ebrahimi et al., "Parallel Application Memory Scheduling," MICRO 2011.
- Seshadri et al., "The Evicted-Address Filter: A Unified Mechanism to Address Both Cache Pollution and Thrashing," PACT 2012.
- Pekhimenko et al., "Linearly Compressed Pages: A Main Memory Compression Framework with Low Complexity and Low Latency," CMU SAFARI Technical Report 2012.

# Last Lecture

- Major Trends Affecting Main Memory
- Requirements from an Ideal Main Memory System
- Opportunity: Emerging Memory Technologies

# Today

- More Asymmetric Multi-Core
- Staged Execution
- Asymmetry in Memory Scheduling

# More Asymmetric Multi-Core

# Outline

- How Do We Get There: Examples

- Accelerated Critical Sections (ACS)
- Bottleneck Identification and Scheduling (BIS)
- Staged Execution and Data Marshaling

- Asymmetry in Memory
  - Thread Cluster Memory Scheduling
  - Heterogeneous DRAM+NVM Main Memory

# BIS Summary

- Problem: Performance and scalability of multithreaded applications are limited by serializing bottlenecks
  - different types: critical sections, barriers, slow pipeline stages
  - importance (criticality) of a bottleneck can change over time
- Our Goal: Dynamically identify the most important bottlenecks and accelerate them
  - How to identify the most critical bottlenecks
  - How to efficiently accelerate them
- Solution: Bottleneck Identification and Scheduling (BIS)
  - Software: annotate bottlenecks (BottleneckCall, BottleneckReturn) and implement waiting for bottlenecks with a special instruction (BottleneckWait)
  - Hardware: identify bottlenecks that cause the most thread waiting and accelerate those bottlenecks on large cores of an asymmetric multi-core system
- Improves multithreaded application performance and scalability, outperforms previous work, and performance improves with more cores

# Bottlenecks in Multithreaded Applications

Definition: any code segment for which threads contend (i.e. wait)

Examples:

- Amdahl's serial portions
  - Only one thread exists → on the critical path
- Critical sections
  - Ensure mutual exclusion → likely to be on the critical path if contended
- Barriers
  - Ensure all threads reach a point before continuing → the latest thread arriving is on the critical path
- Pipeline stages
  - Different stages of a loop iteration may execute on different threads, slowest stage makes other stages wait → on the critical path

# Bottleneck Identification and Scheduling (BIS)

- Key insight:
  - Thread waiting reduces parallelism and is likely to reduce performance
  - Code causing the most thread waiting → likely critical path
- Key idea:
  - Dynamically identify bottlenecks that cause the most thread waiting
  - Accelerate them (using powerful cores in an ACMP)

# Bottleneck Identification and Scheduling (BIS)

***Compiler/Library/Programmer***

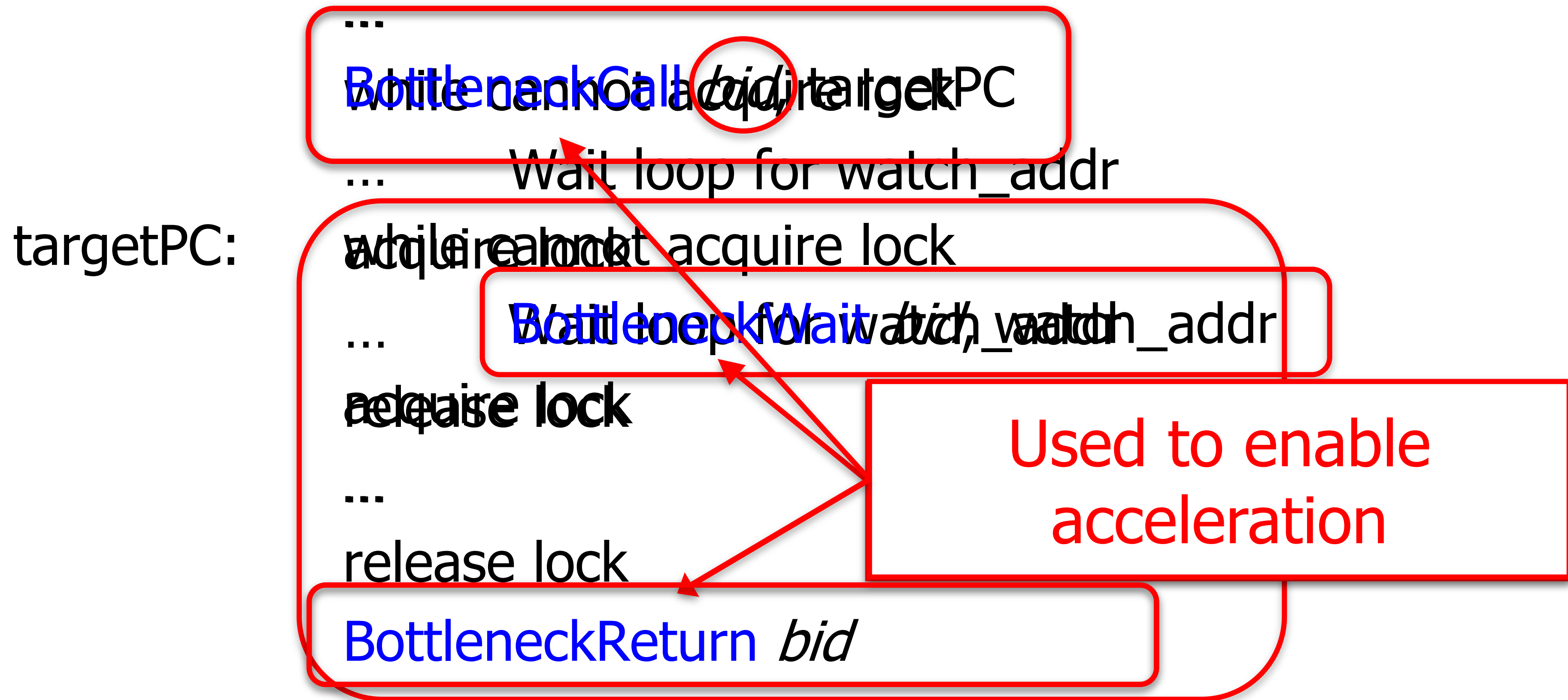
1. Annotate *bottleneck* code
2. Implement *waiting* for bottlenecks

Binary containing **BIS instructions**

***Hardware***

1. Measure *thread waiting cycles (TWC)* for each bottleneck
2. Accelerate bottleneck(s) with the highest TWC

# Critical Sections: Code Modifications



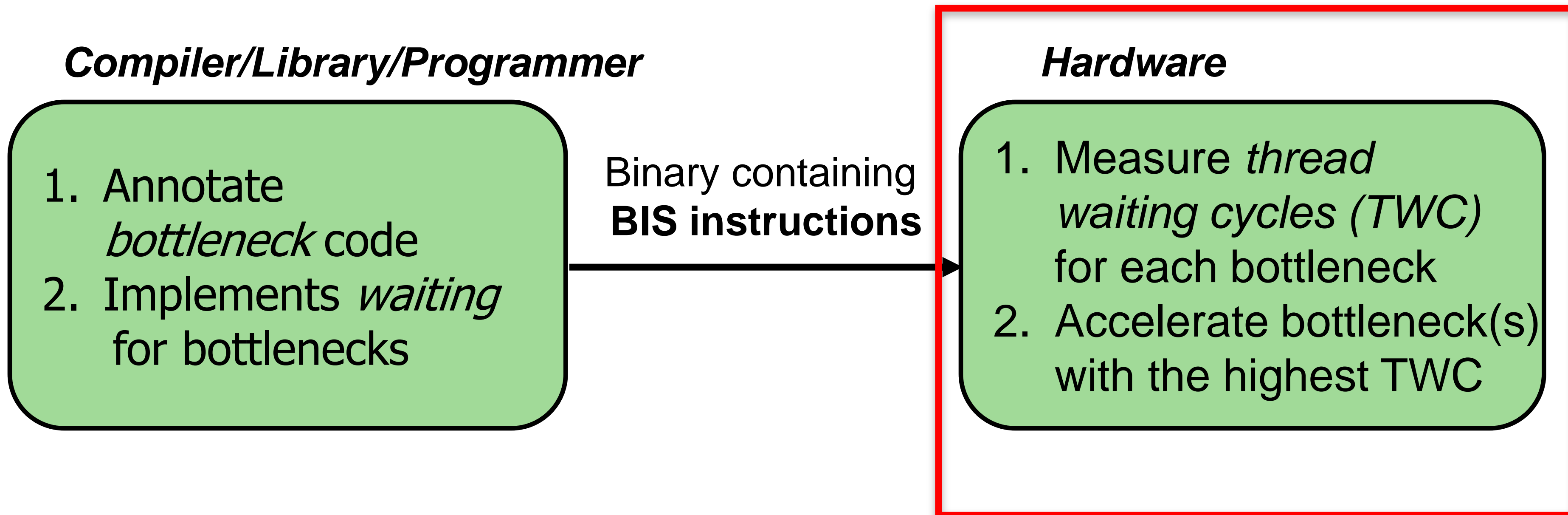
# Barriers: Code Modifications

```
                …
                BottleneckCall bid, targetPC
                enter barrier
                while not all threads in barrier
                        BottleneckWait bid, watch_addr
                exit barrier
                …
targetPC:       code running for the barrier
                …
                BottleneckReturn bid
```

# Pipeline Stages: Code Modifications

```
              BottleneckCall bid, targetPC
              …
targetPC:     while not done
                  while empty queue
                      BottleneckWait prev_bid
                  dequeue work
                  do the work …
                  while full queue
                      BottleneckWait next_bid
                  enqueue next work
              BottleneckReturn bid
```

# Bottleneck Identification and Scheduling (BIS)



# BIS: Hardware Overview

- Performance-limiting bottleneck identification and acceleration are independent tasks
- Acceleration can be accomplished in multiple ways
  - Increasing core frequency/voltage
  - Prioritization in shared resources [Ebrahimi+, MICRO'11]
  - Migration to faster cores in an Asymmetric CMP

| Small core | Small core | Large core | |
|---|---|---|---|
| Small core | Small core | | |
| Small core | Small core | Small core | Small core |
| Small core | Small core | Small core | Small core |

# Bottleneck Identification and Scheduling (BIS)

***Compiler/Library/Programmer***

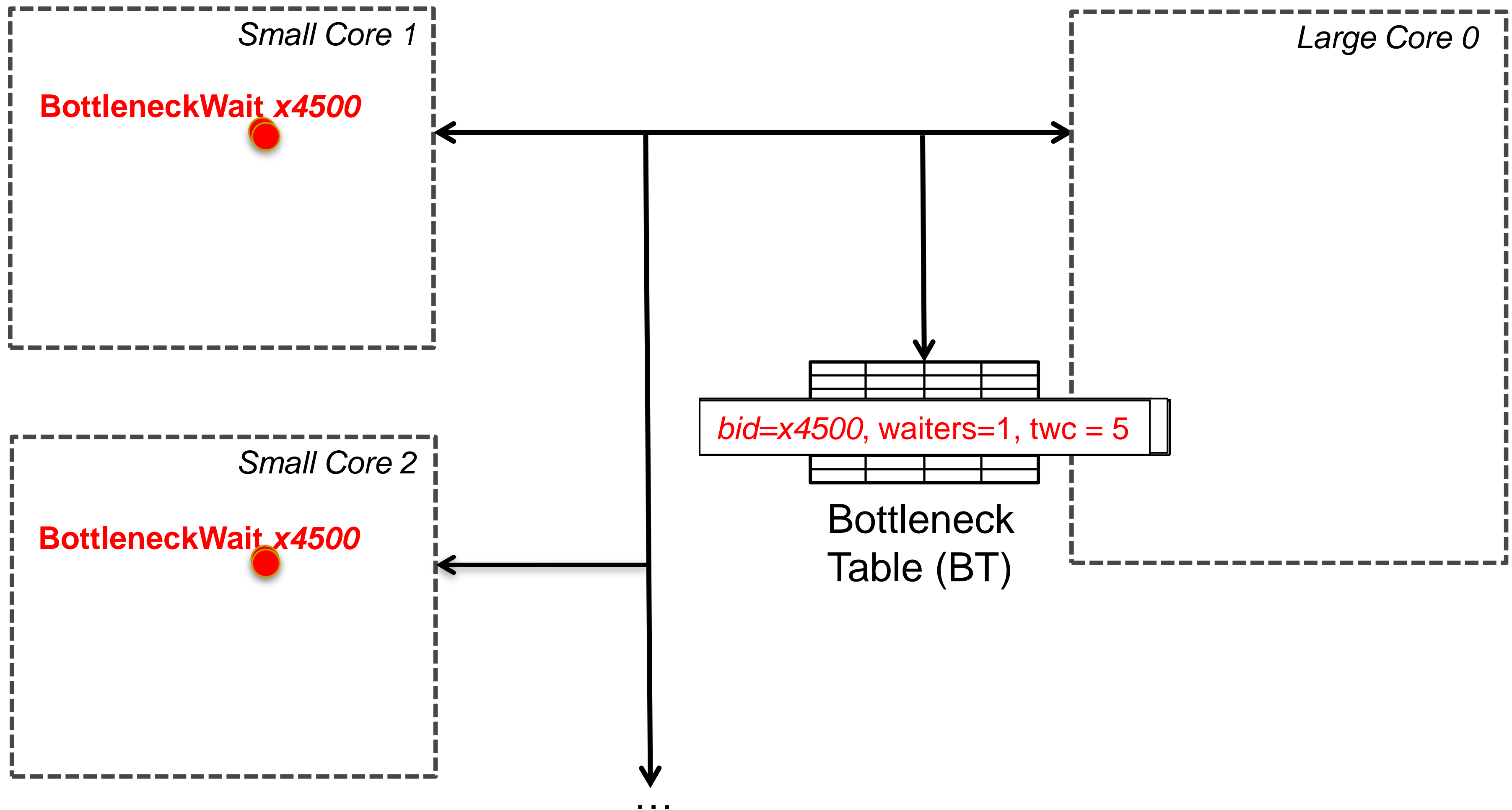
1. Annotate *bottleneck* code
2. Implements *waiting* for bottlenecks

Binary containing **BIS instructions**

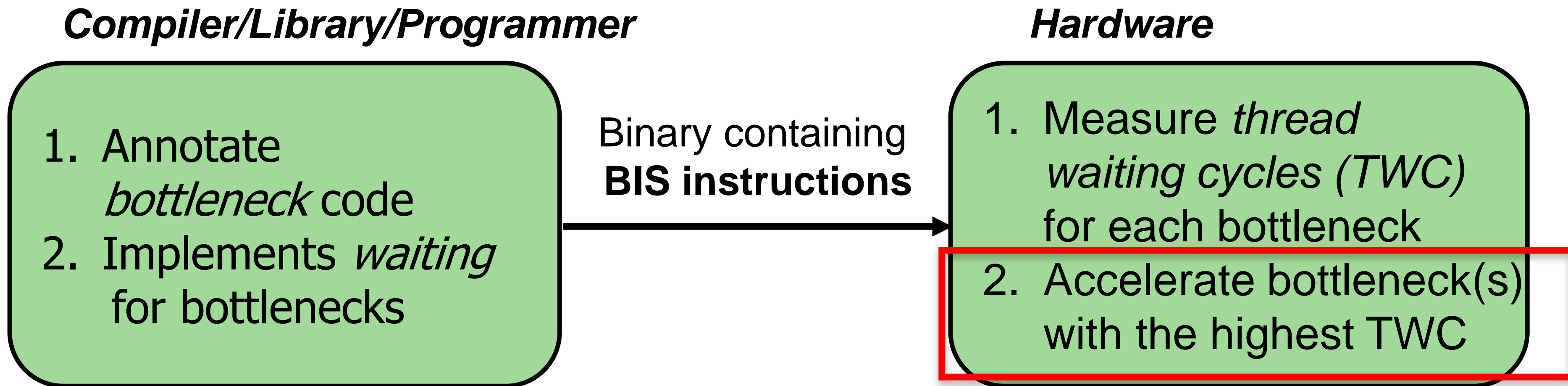
***Hardware***

1. Measure *thread waiting cycles (TWC)* for each bottleneck
2. Accelerate bottleneck(s) with the highest TWC

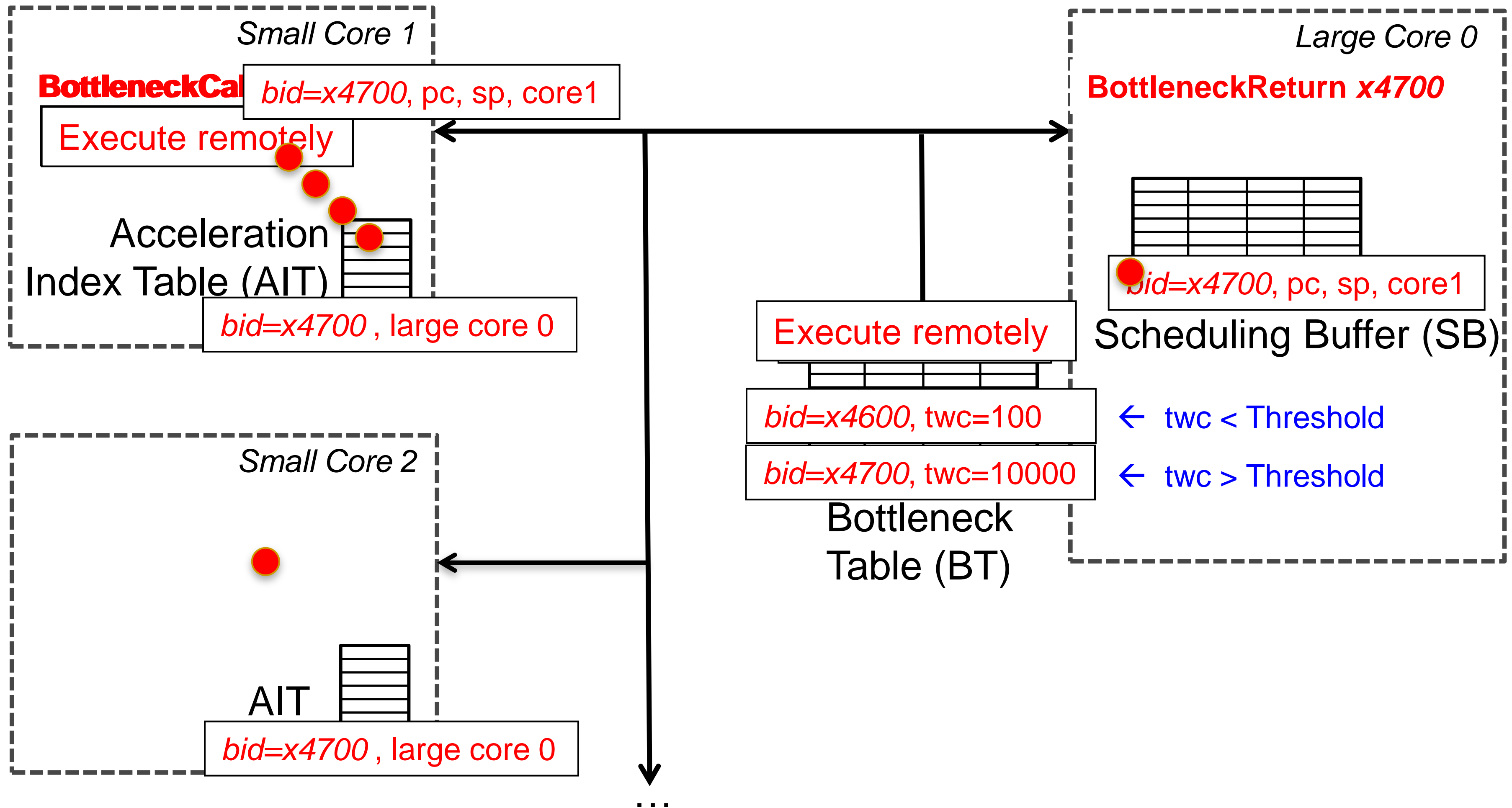
# Determining Thread Waiting Cycles for Each Bottleneck



# Bottleneck Identification and Scheduling (BIS)



# Bottleneck Acceleration



# BIS Mechanisms

- Basic mechanisms for BIS:
  - Determining Thread Waiting Cycles ✓
  - Accelerating Bottlenecks ✓

- Mechanisms to improve performance and generality of BIS:
  - Dealing with false serialization
  - Preemptive acceleration
  - Support for multiple large cores

# False Serialization and Starvation

- Observation: Bottlenecks are picked from Scheduling Buffer in Thread Waiting Cycles order
- Problem: An independent bottleneck that is ready to execute has to wait for another bottleneck that has higher thread waiting cycles → False serialization
- Starvation: Extreme false serialization
- Solution: Large core detects when a bottleneck is ready to execute in the Scheduling Buffer but it cannot → sends the bottleneck back to the small core

# Preemptive Acceleration

- Observation: A bottleneck executing on a small core can become the bottleneck with the highest thread waiting cycles

- Problem: This bottleneck should really be accelerated (i.e., executed on the large core)

- Solution: The Bottleneck Table detects the situation and sends a preemption signal to the small core. Small core:
  - saves register state on stack, ships the bottleneck to the large core

- Main acceleration mechanism for barriers and pipeline stages

# Support for Multiple Large Cores

- Objective: to accelerate independent bottlenecks
- Each large core has its own Scheduling Buffer (shared by all of its SMT threads)
- Bottleneck Table assigns each bottleneck to a fixed large core context to
  - preserve cache locality
  - avoid busy waiting
- Preemptive acceleration extended to send multiple instances of a bottleneck to different large core contexts

# Hardware Cost

- Main structures:
  - Bottleneck Table (BT): global 32-entry associative cache, minimum-Thread-Waiting-Cycle replacement
  - Scheduling Buffers (SB): one table per large core, as many entries as small cores
  - Acceleration Index Tables (AIT): one 32-entry table per small core
- Off the critical path
- Total storage cost for 56-small-cores, 2-large-cores < 19 KB

# BIS Performance Trade-offs

- Bottleneck identification:
  - Small cost: BottleneckWait instruction and Bottleneck Table
- Bottleneck acceleration on an ACMP (execution migration):
  - Faster bottleneck execution vs. fewer parallel threads
    - Acceleration offsets loss of parallel throughput with large core counts
  - Better shared data locality vs. worse private data locality
    - Shared data stays on large core (good)
    - Private data migrates to large core (bad, but latency hidden with Data Marshaling [Suleman+, ISCA' 10])
  - Benefit of acceleration vs. migration latency
    - Migration latency usually hidden by waiting (good)
    - Unless bottleneck not contended (bad, but likely to not be on critical path)

# Methodology

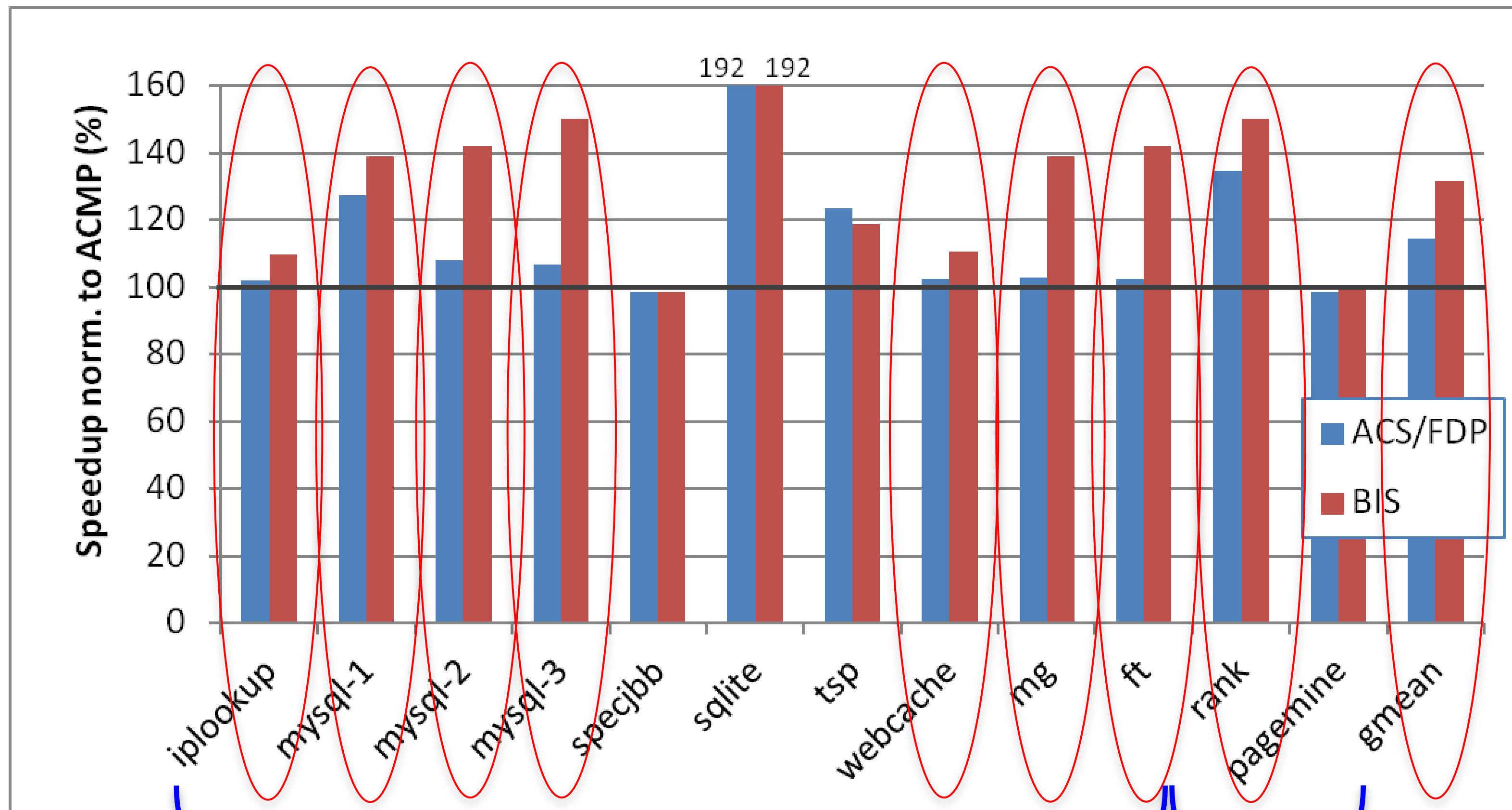
- Workloads: 8 critical section intensive, 2 barrier intensive and 2 pipeline-parallel applications
  - Data mining kernels, scientific, database, web, networking, specjbb
- Cycle-level multi-core x86 simulator
  - 8 to 64 small-core-equivalent area, 0 to 3 large cores, SMT
  - 1 large core is area-equivalent to 4 small cores
- Details:
  - Large core: 4GHz, out-of-order, 128-entry ROB, 4-wide, 12-stage
  - Small core: 4GHz, in-order, 2-wide, 5-stage
  - Private 32KB L1, private 256KB L2, shared 8MB L3
  - On-chip interconnect: Bi-directional ring, 2-cycle hop latency

# BIS Comparison Points (Area-Equivalent)

- SCMP (Symmetric CMP)
  - All small cores
  - Results in the paper
- ACMP (Asymmetric CMP)
  - Accelerates only Amdahl's serial portions
  - Our baseline
- ACS (Accelerated Critical Sections)
  - Accelerates only critical sections and Amdahl's serial portions
  - Applicable to multithreaded workloads (iplookup, mysql, specjbb, sqlite, tsp, webcache, mg, ft)
- FDP (Feedback-Directed Pipelining)
  - Accelerates only slowest pipeline stages
  - Applicable to pipeline-parallel workloads (rank, pagemine)

# BIS Performance Improvement

Optimal number of threads, 28 small cores, 1 large core




limiting bottlenecks change over time

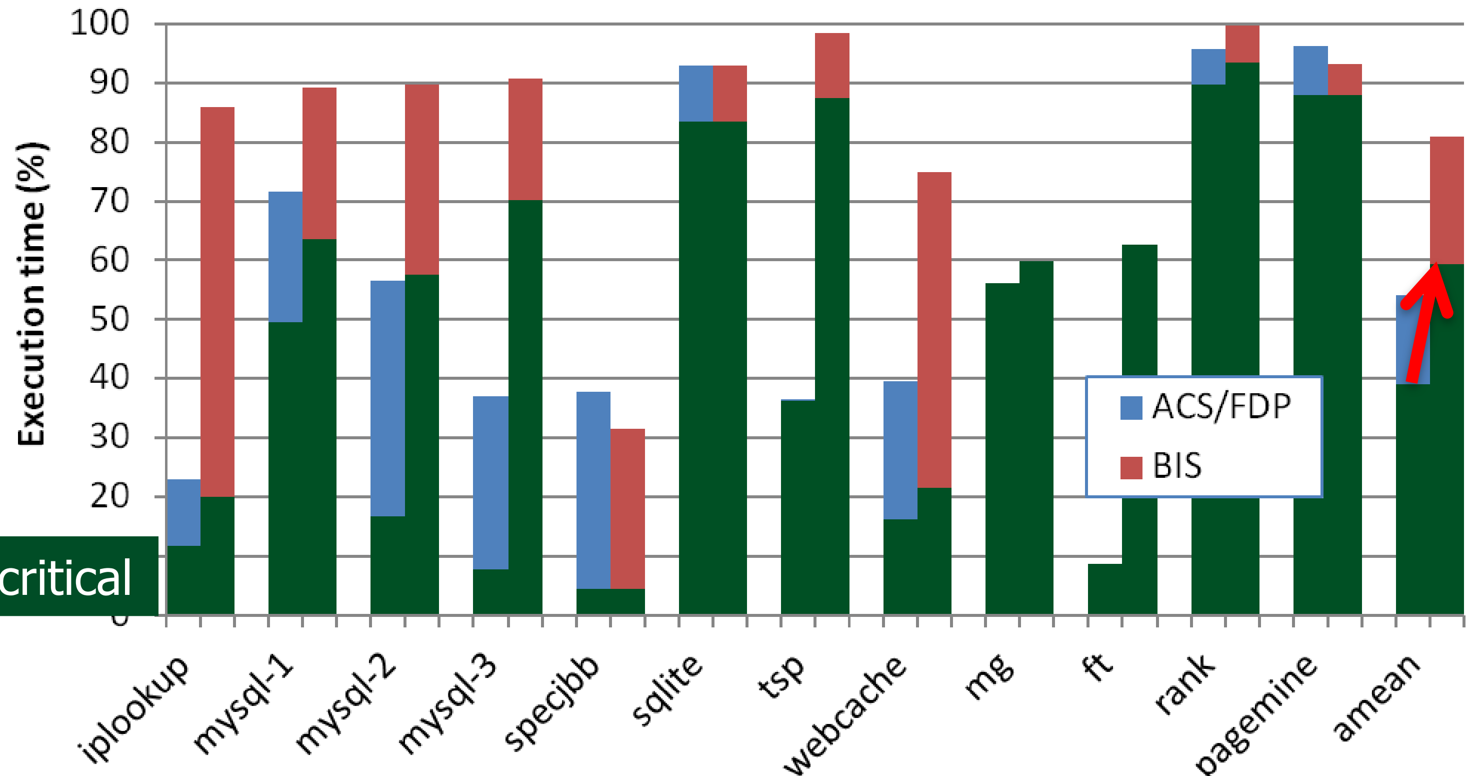
ACS

barriers, which ACS cannot accelerate

FDP

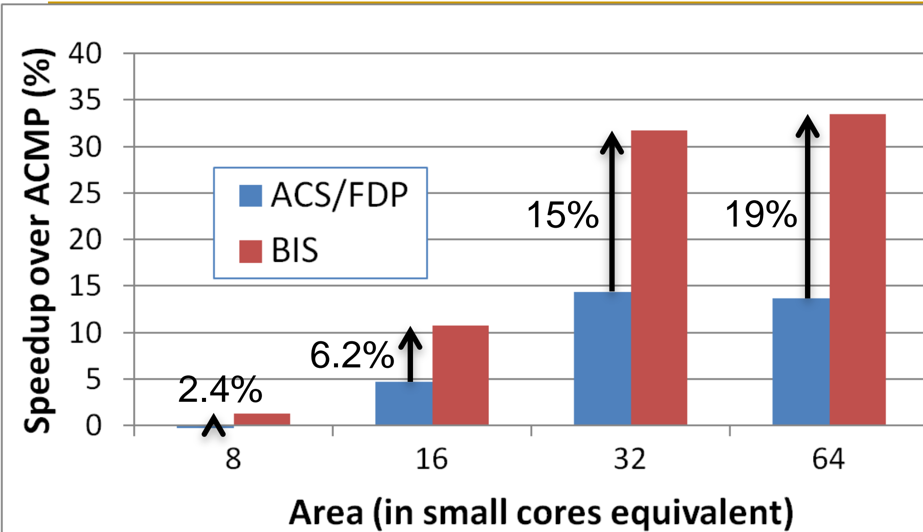
- BIS outperforms ACS/FDP by 15% and ACMP by 32%
- BIS improves scalability on 4 of the benchmarks

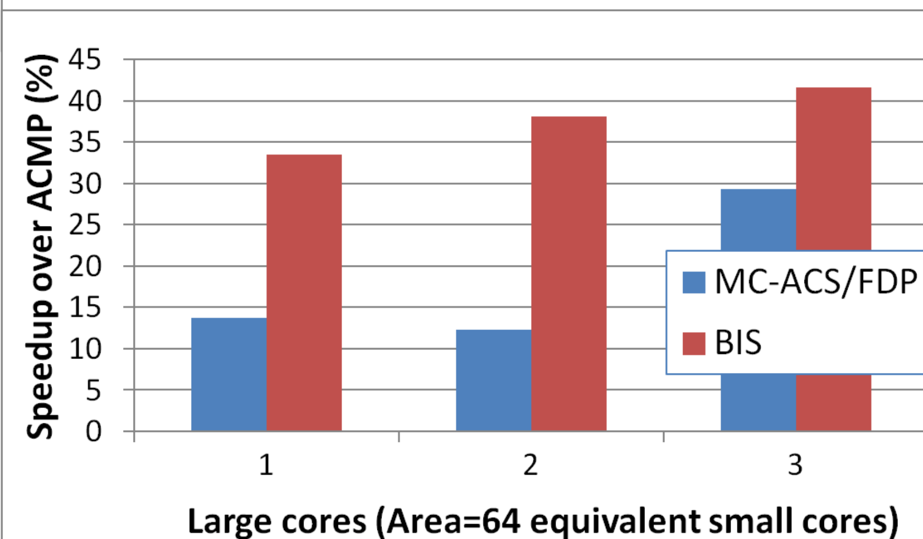
# Why Does BIS Work?




- Coverage: fraction of program critical path that is actually identified as bottlenecks
  - 39% (ACS/FDP) to 59% (BIS)
- Accuracy: identified bottlenecks on the critical path over total identified bottlenecks
  - 72% (ACS/FDP) to 73.5% (BIS)

# BIS Scaling Results




Performance increases with:

1) More small cores

- Contention due to bottlenecks increases
- Loss of parallel throughput due to large core reduces

2) More large cores

- Can accelerate independent bottlenecks
- *Without reducing parallel throughput (enough cores)*

# BIS Summary

- Serializing bottlenecks of different types limit performance of multithreaded applications: Importance changes over time
- BIS is a hardware/software cooperative solution:
  - Dynamically identifies bottlenecks that cause the most thread waiting and accelerates them on large cores of an ACMP
  - Applicable to critical sections, barriers, pipeline stages
- BIS improves application performance and scalability:
  - 15% speedup over ACS/FDP
  - Can accelerate multiple independent critical bottlenecks
  - Performance benefits increase with more cores
- Provides comprehensive fine-grained bottleneck acceleration for future ACMPs with little or no programmer effort

# Outline

- How Do We Get There: Examples

- Accelerated Critical Sections (ACS)
- Bottleneck Identification and Scheduling (BIS)
- Staged Execution and Data Marshaling

- Asymmetry in Memory
  - Thread Cluster Memory Scheduling
  - Heterogeneous DRAM+NVM Main Memory

# Staged Execution Model (I)

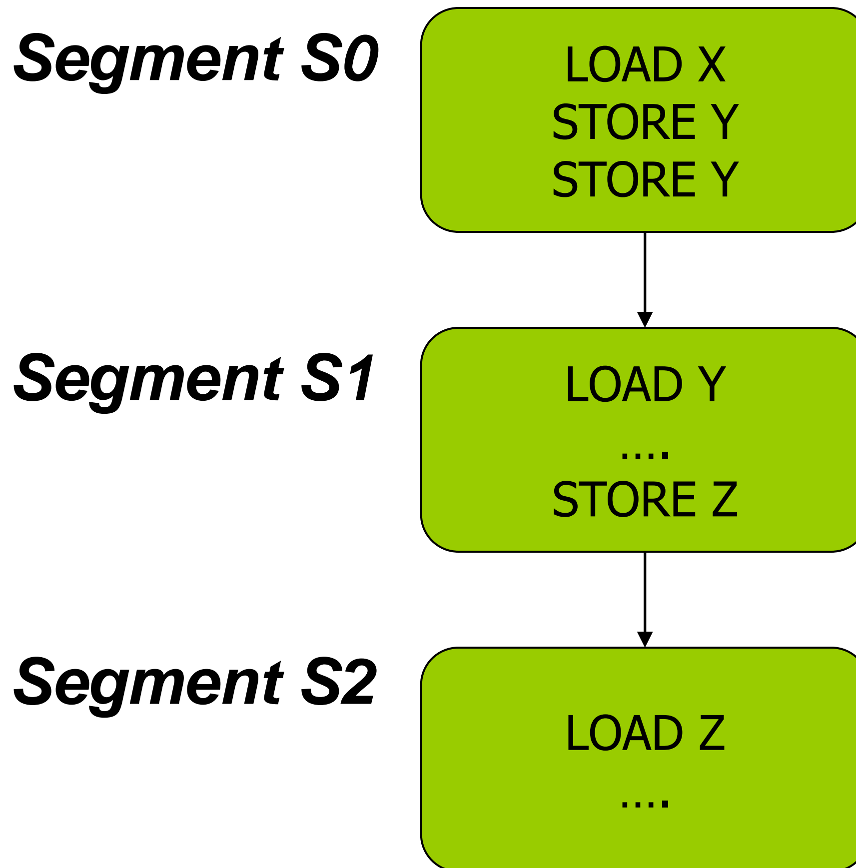
- Goal: speed up a program by dividing it up into pieces
- Idea
  - Split program code into ***segments***
  - Run each segment on the core best-suited to run it
  - Each core assigned a work-queue, storing segments to be run

- Benefits
  - Accelerates segments/critical-paths using specialized/heterogeneous cores
  - Exploits inter-segment parallelism
  - Improves locality of within-segment data

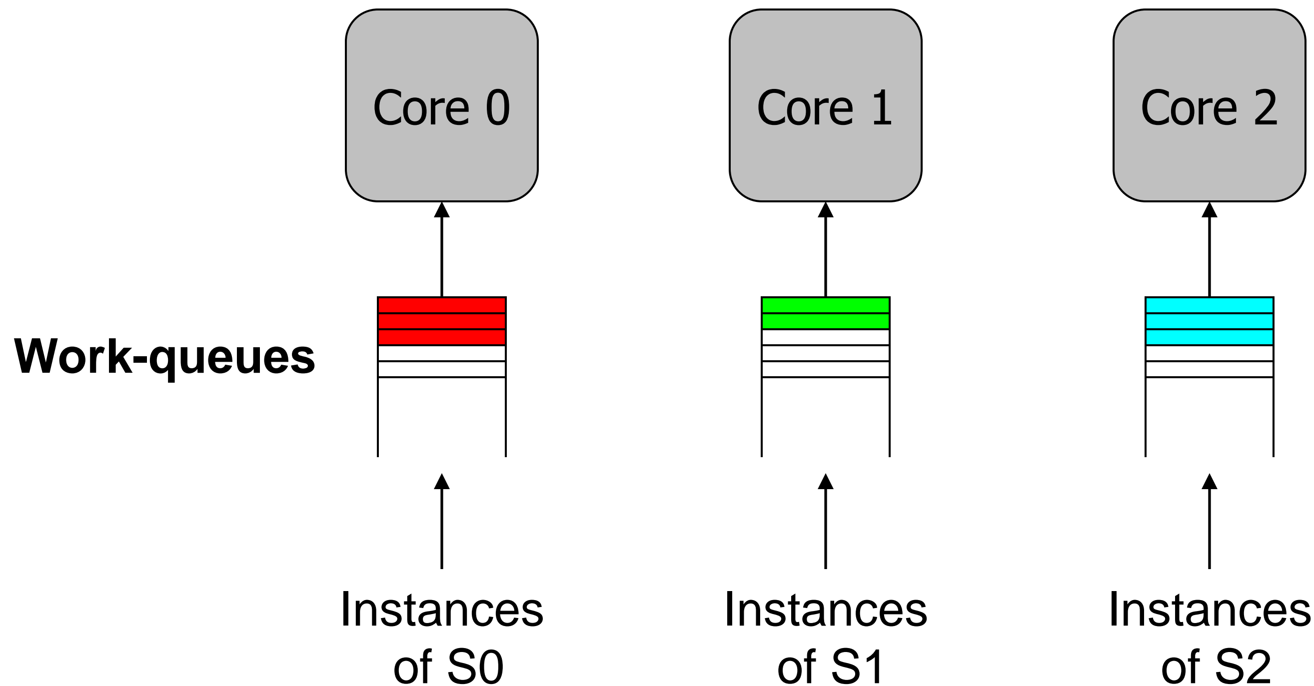
- Examples
  - Accelerated critical sections, Bottleneck identification and scheduling
  - Producer-consumer pipeline parallelism
  - Task parallelism (Cilk, Intel TBB, Apple Grand Central Dispatch)
  - Special-purpose cores and functional units

# Staged Execution Model (II)

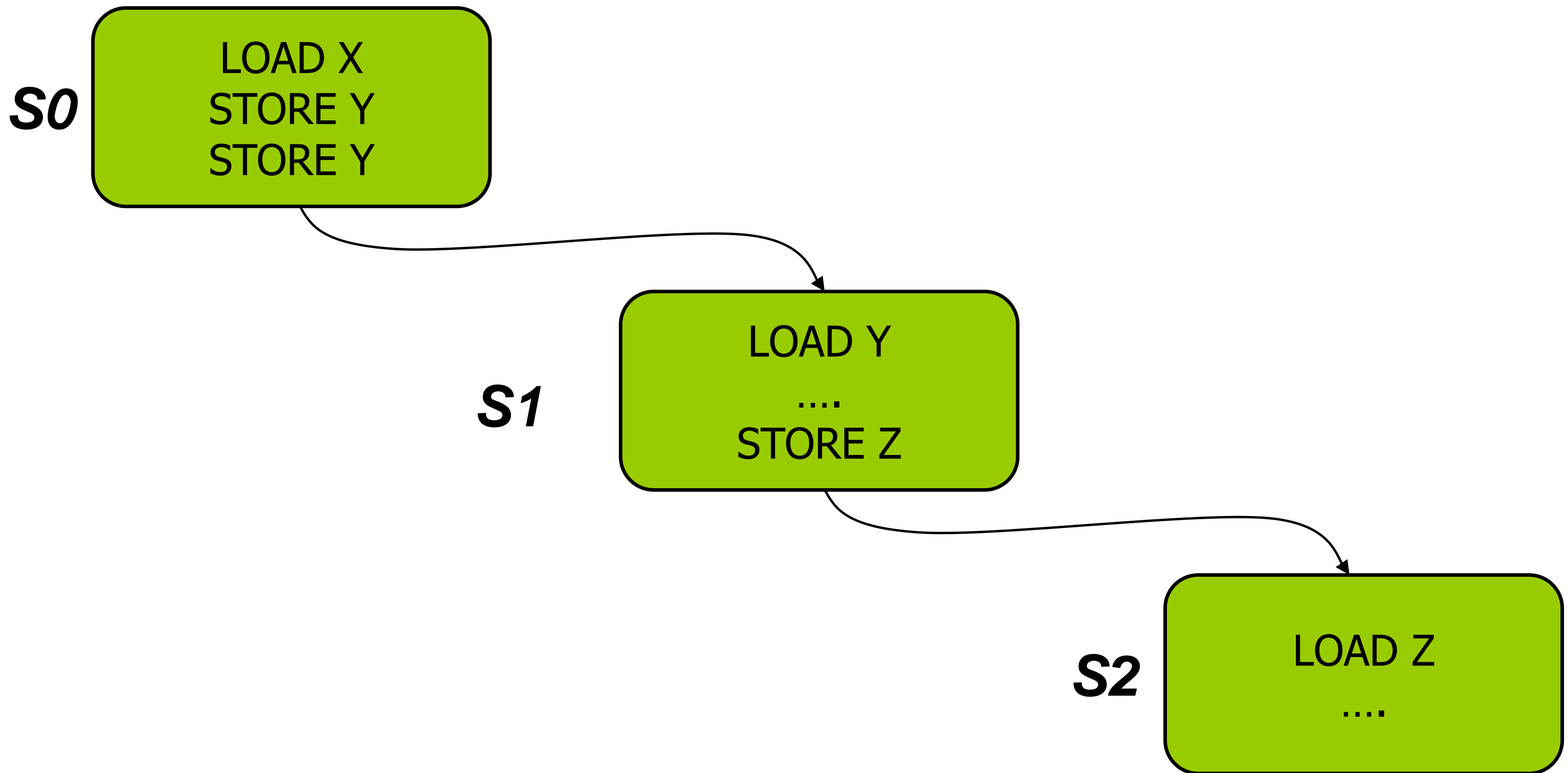
# Staged Execution Model (III)



# Staged Execution Model (IV)



# Staged Execution Model: Segment Spawning

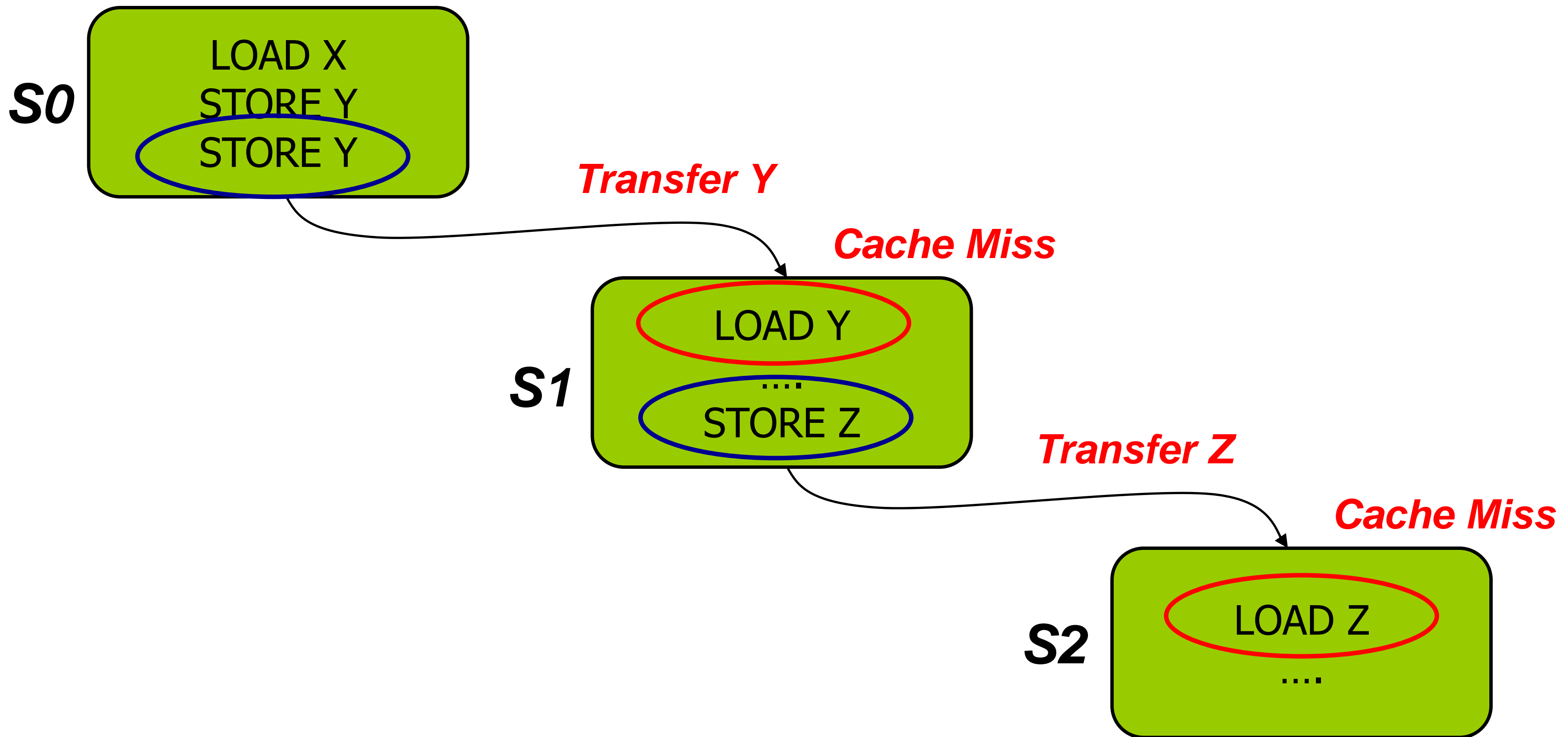


# Staged Execution Model: Two Examples

- Accelerated Critical Sections [Suleman et al., ASPLOS 2009]
  - Idea: Ship critical sections to a large core in an asymmetric CMP
    - Segment 0: Non-critical section
    - Segment 1: Critical section
  - Benefit: Faster execution of critical section, reduced serialization, improved lock and shared data locality

- Producer-Consumer Pipeline Parallelism
  - Idea: Split a loop iteration into multiple "pipeline stages" where one stage consumes data produced by the next stage → each stage runs on a different core
    - Segment N: Stage N
  - Benefit: Stage-level parallelism, better locality → faster execution

# Problem: Locality of Inter-segment Data



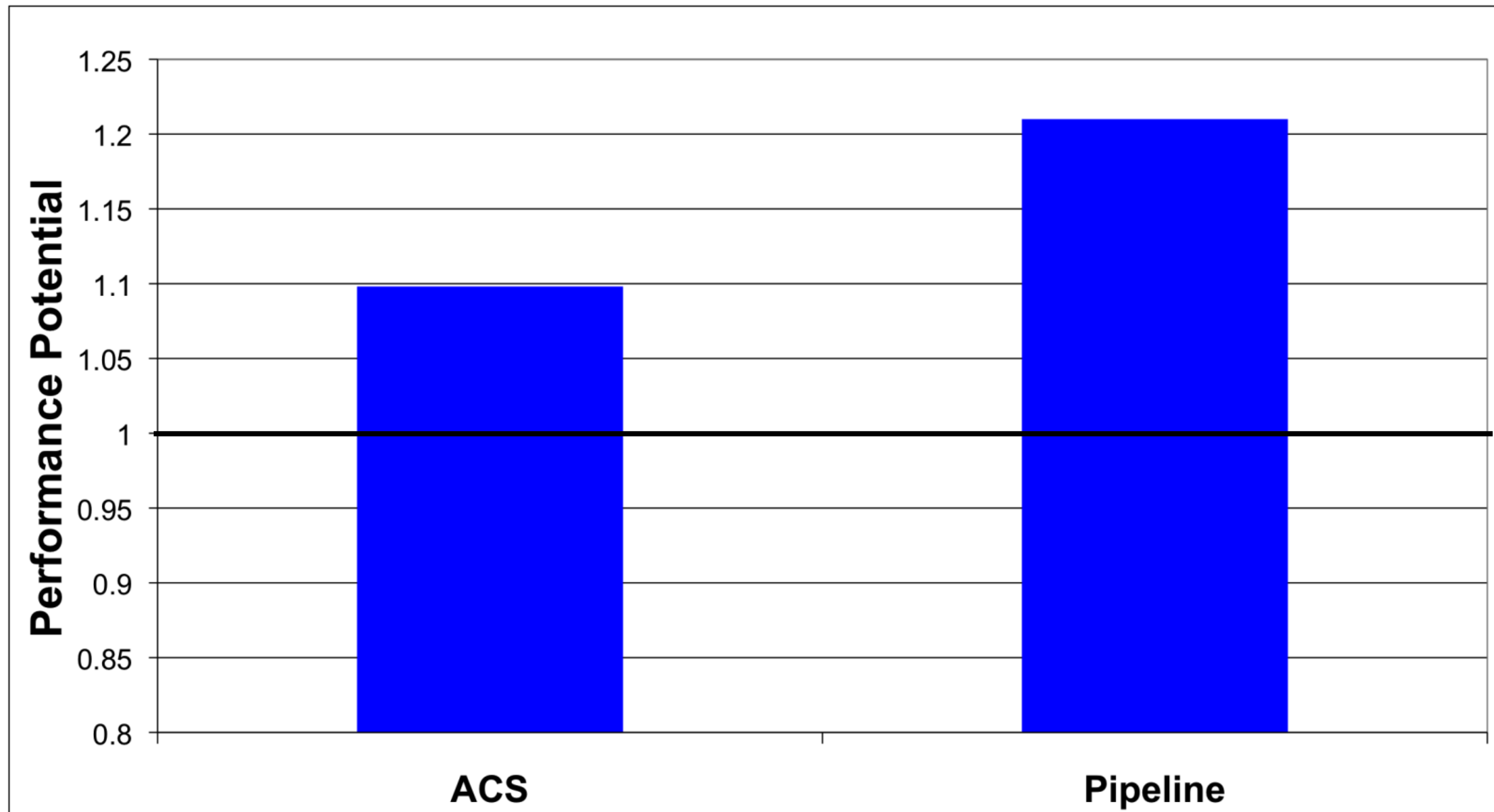
# Problem: Locality of Inter-segment Data

- Accelerated Critical Sections [Suleman et al., ASPLOS 2009]
  - Idea: Ship critical sections to a large core in an ACMP
  - Problem: Critical section incurs a cache miss when it touches data produced in the non-critical section (i.e., thread private data)

- Producer-Consumer Pipeline Parallelism
  - Idea: Split a loop iteration into multiple "pipeline stages" → each stage runs on a different core
  - Problem: A stage incurs a cache miss when it touches data produced by the previous stage

- Performance of Staged Execution limited by inter-segment cache misses

# What if We Eliminated All Inter-segment Misses?



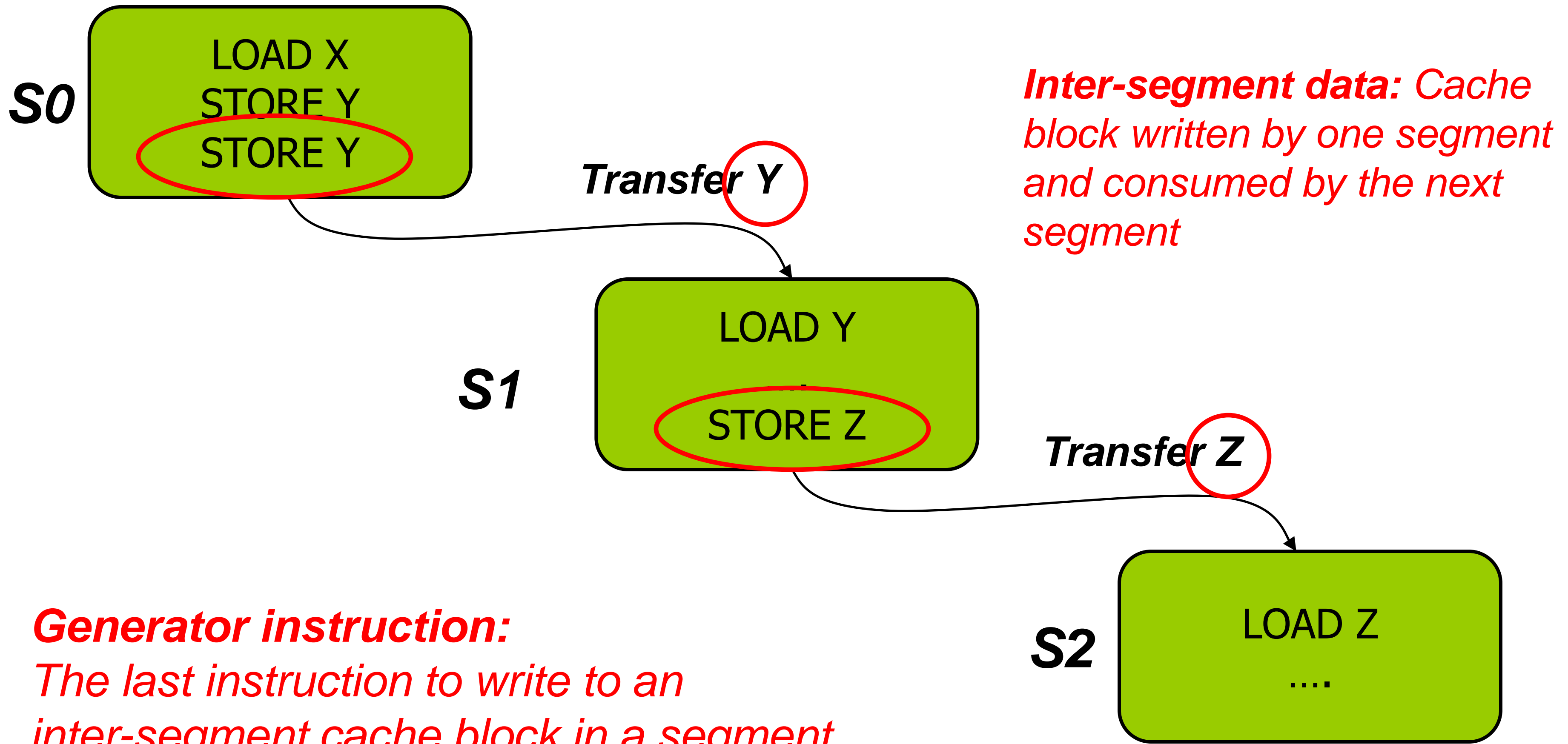
# Outline

- How Do We Get There: Examples

- Accelerated Critical Sections (ACS)
- Bottleneck Identification and Scheduling (BIS)
- Staged Execution and Data Marshaling

- Asymmetry in Memory
  - Thread Cluster Memory Scheduling
  - Heterogeneous DRAM+NVM Main Memory

# Terminology



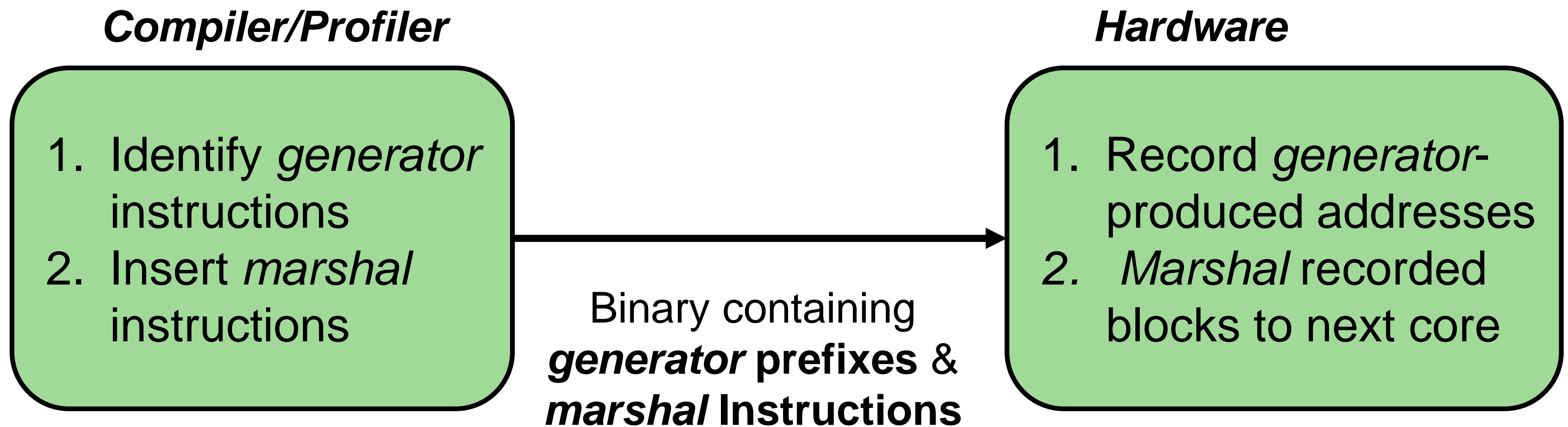
# Key Observation and Idea

- Observation: Set of generator instructions is stable over execution time and across input sets

- Idea:
  - Identify the generator instructions
  - Record cache blocks produced by generator instructions
  - Proactively send such cache blocks to the next segment's core before initiating the next segment

- Suleman et al., "Data Marshaling for Multi-Core Architectures," ISCA 2010, IEEE Micro Top Picks 2011.

# Data Marshaling



# Data Marshaling

**_Compiler/Profiler_**

1. Identify *generator* instructions
2. Insert *marshal* instructions

Binary containing ***generator* prefixes** & ***marshal* Instructions**

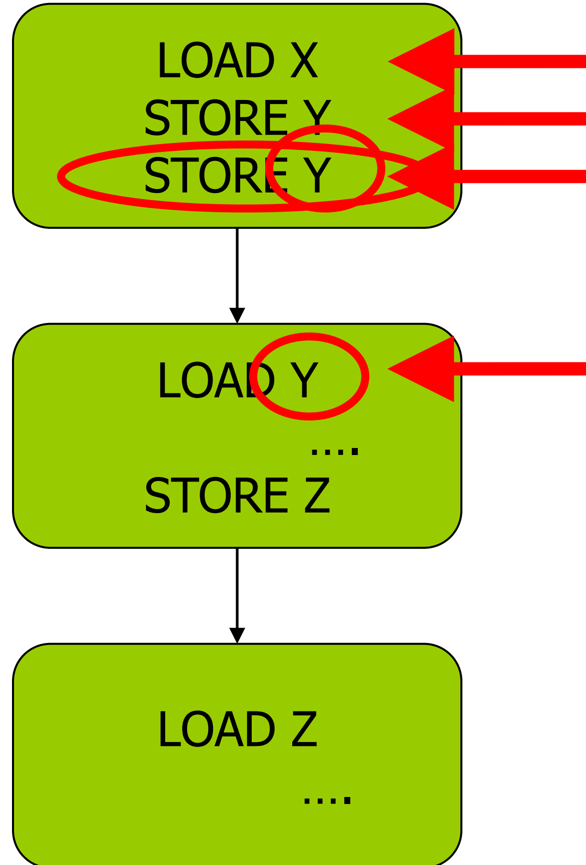
**_Hardware_**

1. Record *generator*-produced addresses
2. *Marshal* recorded blocks to next core

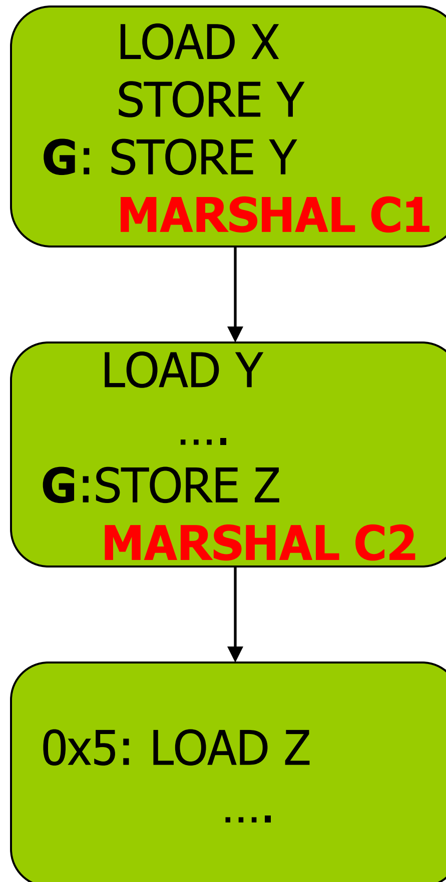
# Profiling Algorithm

***Inter-segment data***

***Mark as Generator Instruction***



# Marshal Instructions




***When to send (Marshal)***

***Where to send (C1)***

# DM Support/Cost

- Profiler/Compiler: Generators, marshal instructions
- ISA: Generator prefix, marshal instructions
- Library/Hardware: Bind next segment ID to a physical core

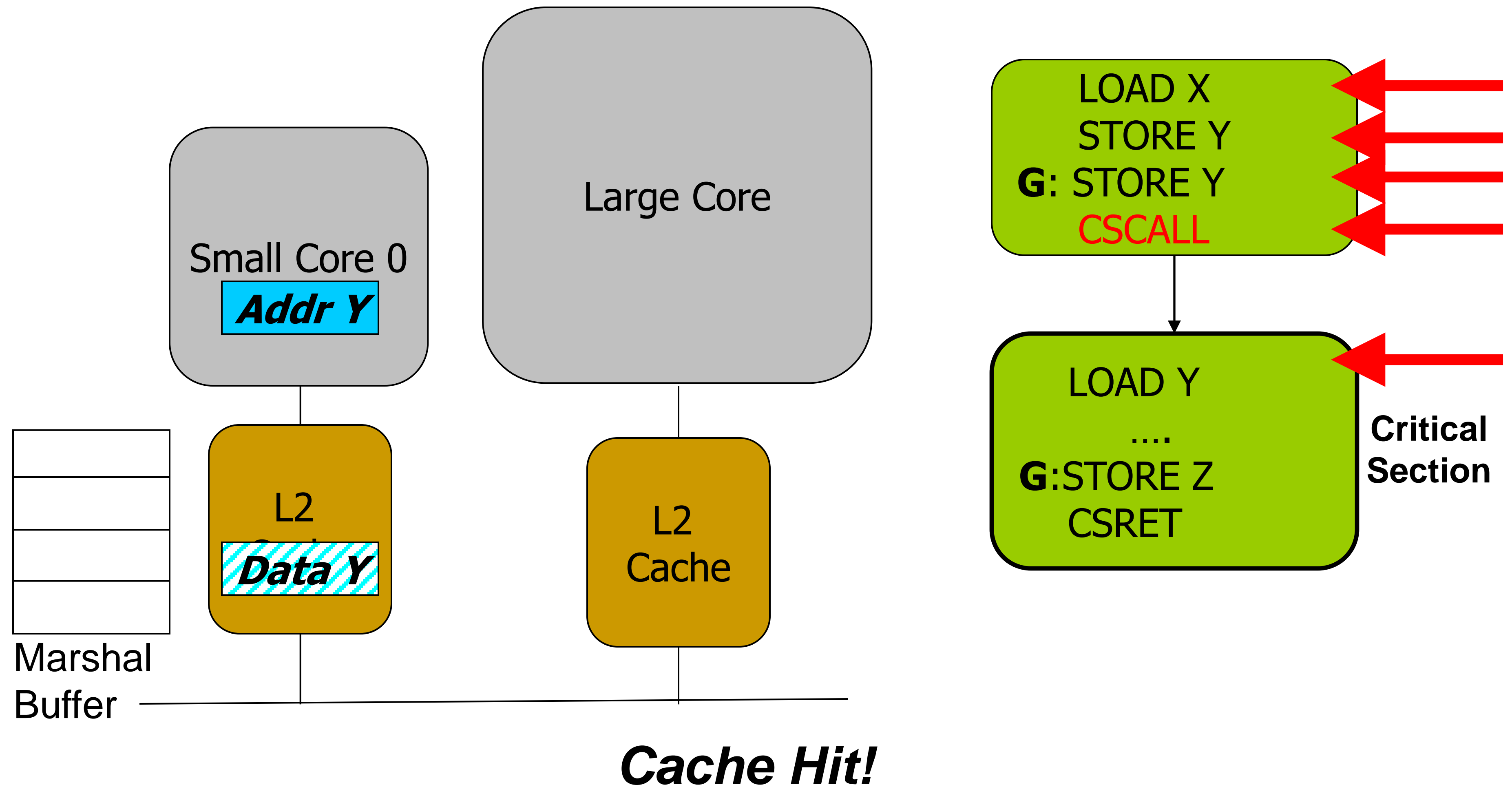
- Hardware
  - Marshal Buffer
    - Stores physical addresses of cache blocks to be marshaled
    - 16 entries enough for almost all workloads → 96 bytes per core
  - Ability to execute generator prefixes and marshal instructions
  - Ability to push data to another cache

# DM: Advantages, Disadvantages

- Advantages
  - Timely data transfer: Push data to core before needed
  - Can marshal any arbitrary sequence of lines: Identifies generators, not patterns
  - Low hardware cost: Profiler marks generators, no need for hardware to find them

- Disadvantages
  - Requires profiler and ISA support
  - Not always accurate (generator set is conservative): Pollution at remote core, wasted bandwidth on interconnect
    - Not a large problem as number of inter-segment blocks is small

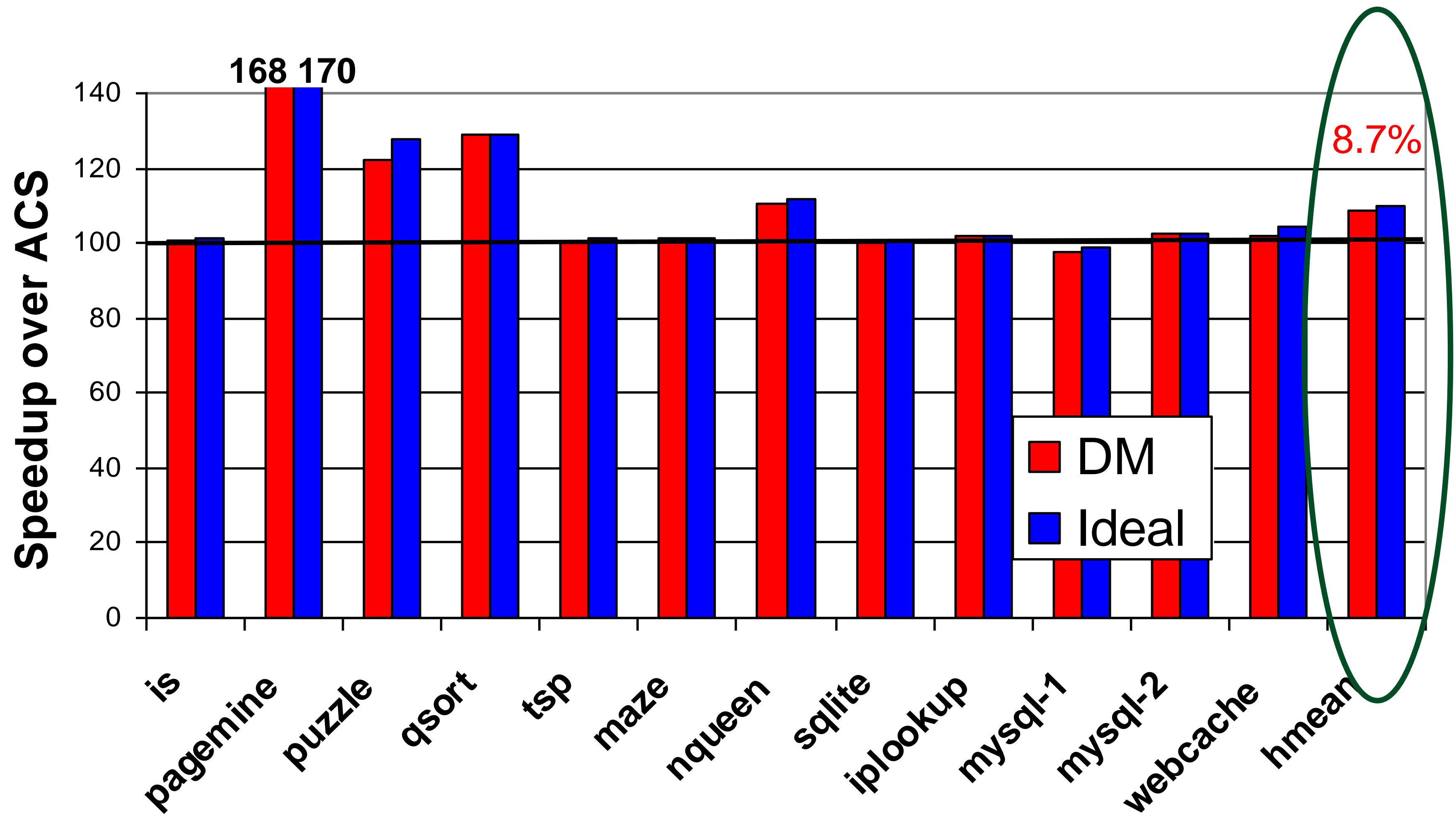
# Accelerated Critical Sections with DM



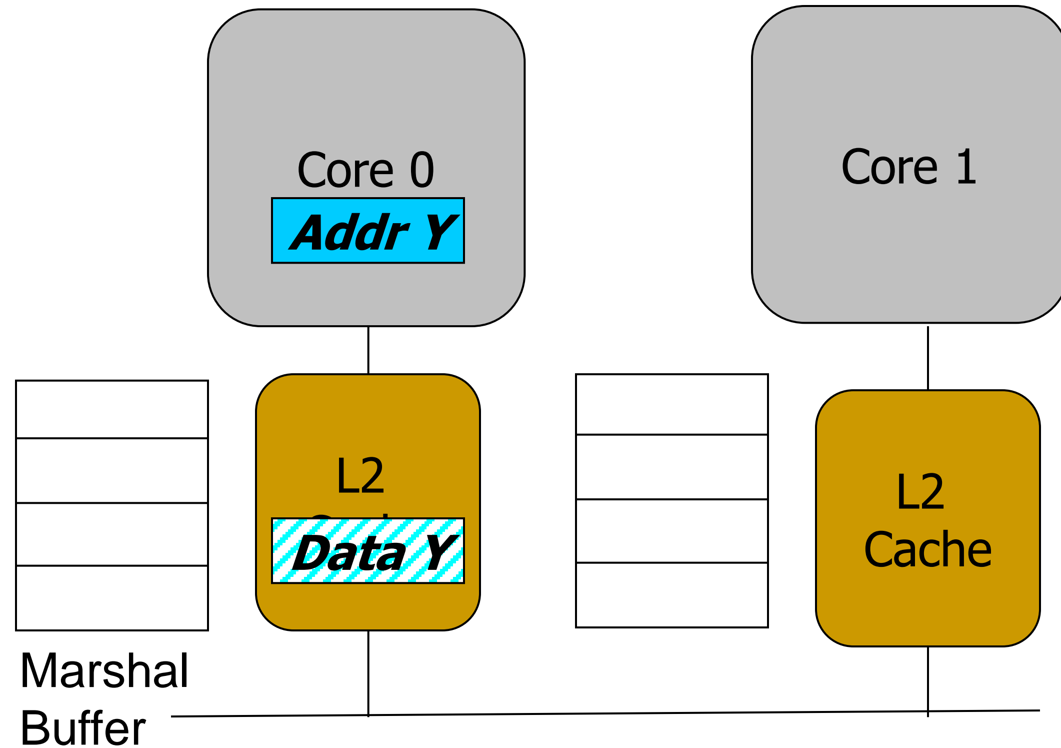
# Accelerated Critical Sections: Methodology

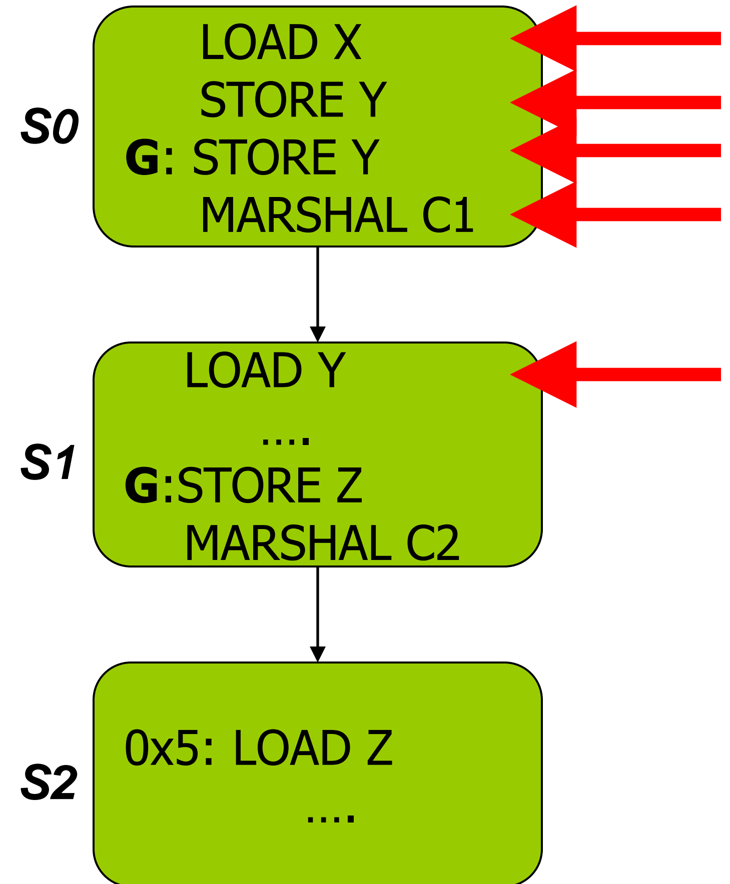
- Workloads: 12 critical section intensive applications
  - Data mining kernels, sorting, database, web, networking
  - Different training and simulation input sets
- Multi-core x86 simulator
  - 1 large and 28 small cores
  - Aggressive stream prefetcher employed at each core
- Details:
  - Large core: 2GHz, out-of-order, 128-entry ROB, 4-wide, 12-stage
  - Small core: 2GHz, in-order, 2-wide, 5-stage
  - Private 32 KB L1, private 256KB L2, 8MB shared L3
  - On-chip interconnect: Bi-directional ring, 5-cycle hop latency

# DM on Accelerated Critical Sections: Results



# Pipeline Parallelism



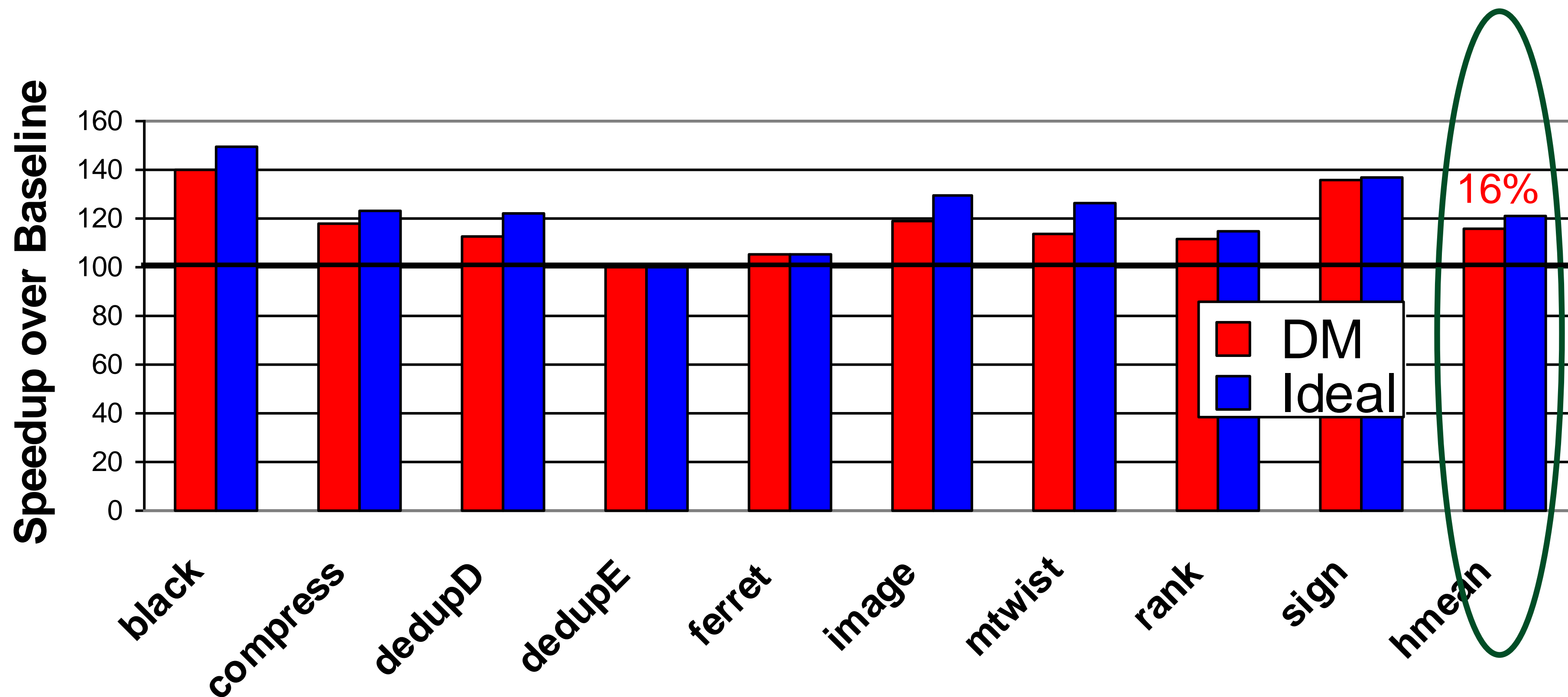



# Pipeline Parallelism: Methodology

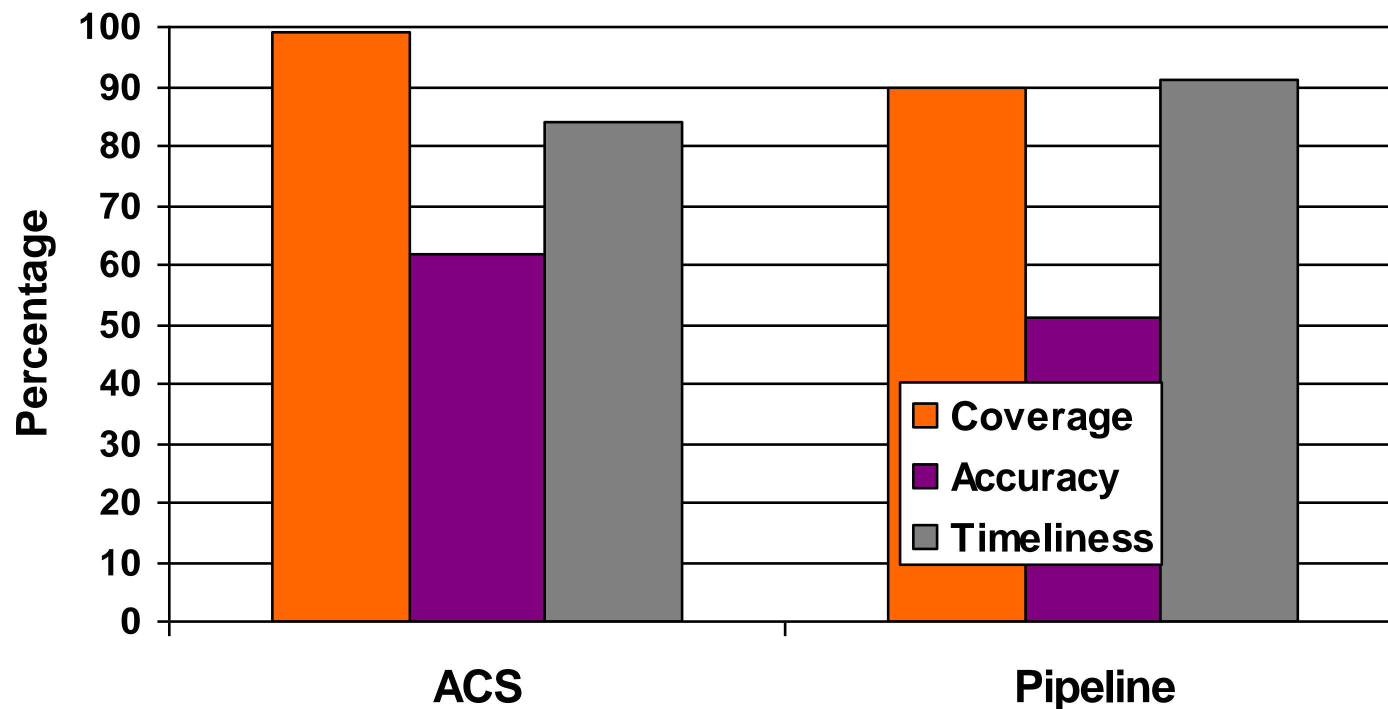
- Workloads: 9 applications with pipeline parallelism
  - Financial, compression, multimedia, encoding/decoding
  - Different training and simulation input sets

- Multi-core x86 simulator
  - 32-core CMP: 2GHz, in-order, 2-wide, 5-stage
  - Aggressive stream prefetcher employed at each core
  - Private 32 KB L1, private 256KB L2, 8MB shared L3
  - On-chip interconnect: Bi-directional ring, 5-cycle hop latency

# DM on Pipeline Parallelism: Results



# DM Coverage, Accuracy, Timeliness




- High coverage of inter-segment misses in a timely manner
- Medium accuracy does not impact performance
  - Only 5.0 and 6.8 cache blocks marshaled for average segment

# Scaling Results

- DM performance improvement increases with
  - More cores
  - Higher interconnect latency
  - Larger private L2 caches

- Why? Inter-segment data misses become a larger bottleneck
  - More cores → More communication
  - Higher latency → Longer stalls due to communication
  - Larger L2 cache → Communication misses remain

# Other Applications of Data Marshaling

- Can be applied to other Staged Execution models
  - Task parallelism models
    - Cilk, Intel TBB, Apple Grand Central Dispatch
  - Special-purpose remote functional units
  - Computation spreading [Chakraborty et al., ASPLOS' 06]
  - Thread motion/migration [e.g., Rangan et al., ISCA' 09]

- Can be an enabler for more aggressive SE models
  - Lowers the cost of data migration
    - an important overhead in remote execution of code segments
  - Remote execution of finer-grained tasks can become more feasible → finer-grained parallelization in multi-cores

# Data Marshaling Summary

- Inter-segment data transfers between cores limit the benefit of promising Staged Execution (SE) models
- Data Marshaling is a hardware/software cooperative solution: detect inter-segment data generator instructions and push their data to next segment's core
  - Significantly reduces cache misses for inter-segment data
  - Low cost, high-coverage, timely for arbitrary address sequences
  - Achieves most of the potential of eliminating such misses
- Applicable to several existing Staged Execution models
  - Accelerated Critical Sections: 9% performance benefit
  - Pipeline Parallelism: 16% performance benefit
- Can enable new models→ very fine-grained remote execution

# Outline

- How Do We Get There: Examples

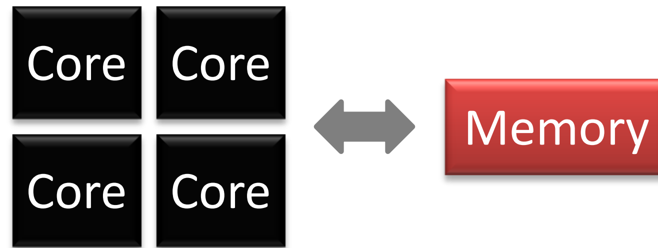
- Accelerated Critical Sections (ACS)
- Bottleneck Identification and Scheduling (BIS)
- Staged Execution and Data Marshaling

- Asymmetry in Memory
  - Thread Cluster Memory Scheduling
  - Heterogeneous DRAM+NVM Main Memory

We did not cover the following slides in lecture. These are for your preparation for the next lecture.

# Motivation

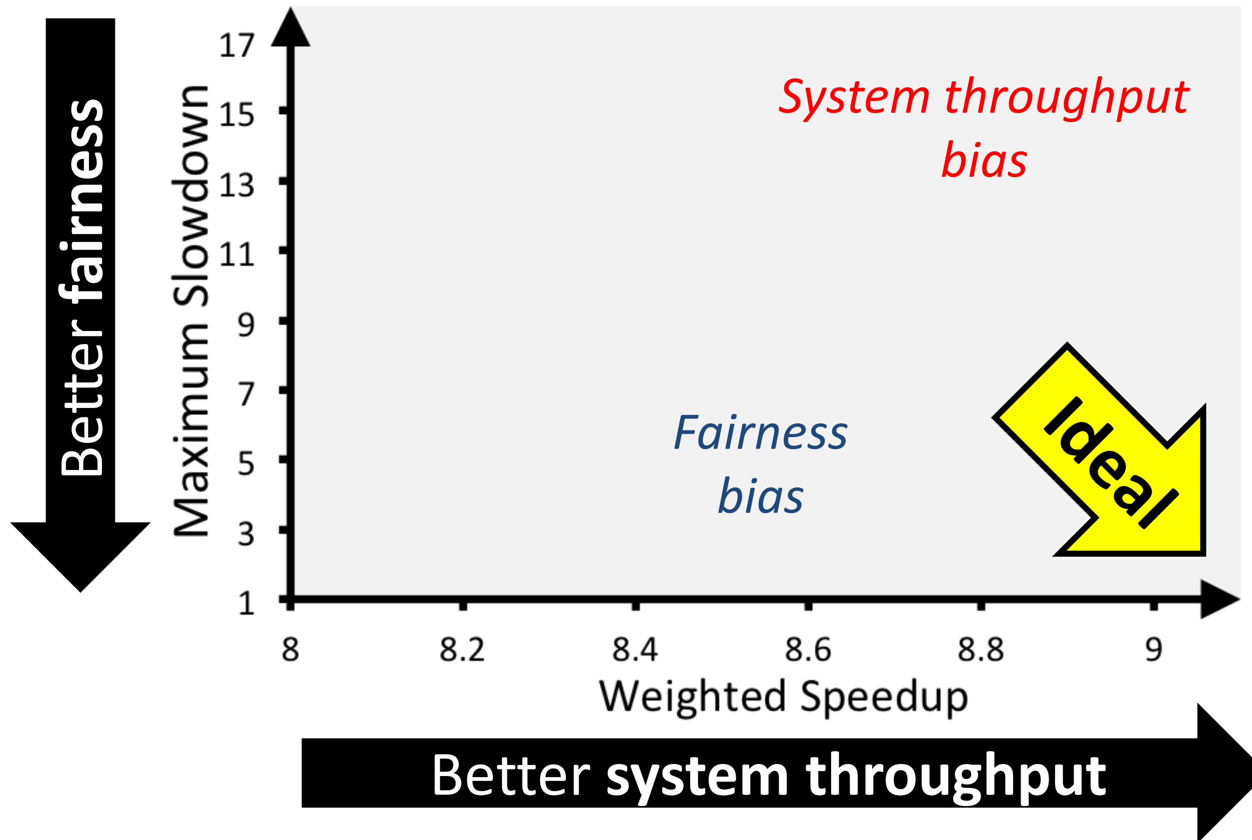
- Memory is a shared resource




- Threads' requests contend for memory
  - Degradation in single thread performance
  - Can even lead to starvation

- How to schedule memory requests to increase both system throughput and fairness?

# Previous Scheduling Algorithms are Biased




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

# Why do Previous Algorithms Fail?

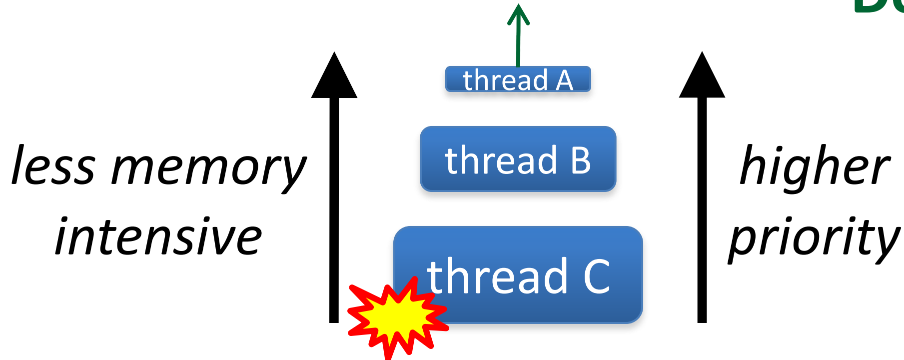
**Throughput biased** *approach*

Prioritize less memory-intensive threads

**Fairness biased** *approach*

Take turns accessing memory

**Good for throughput**



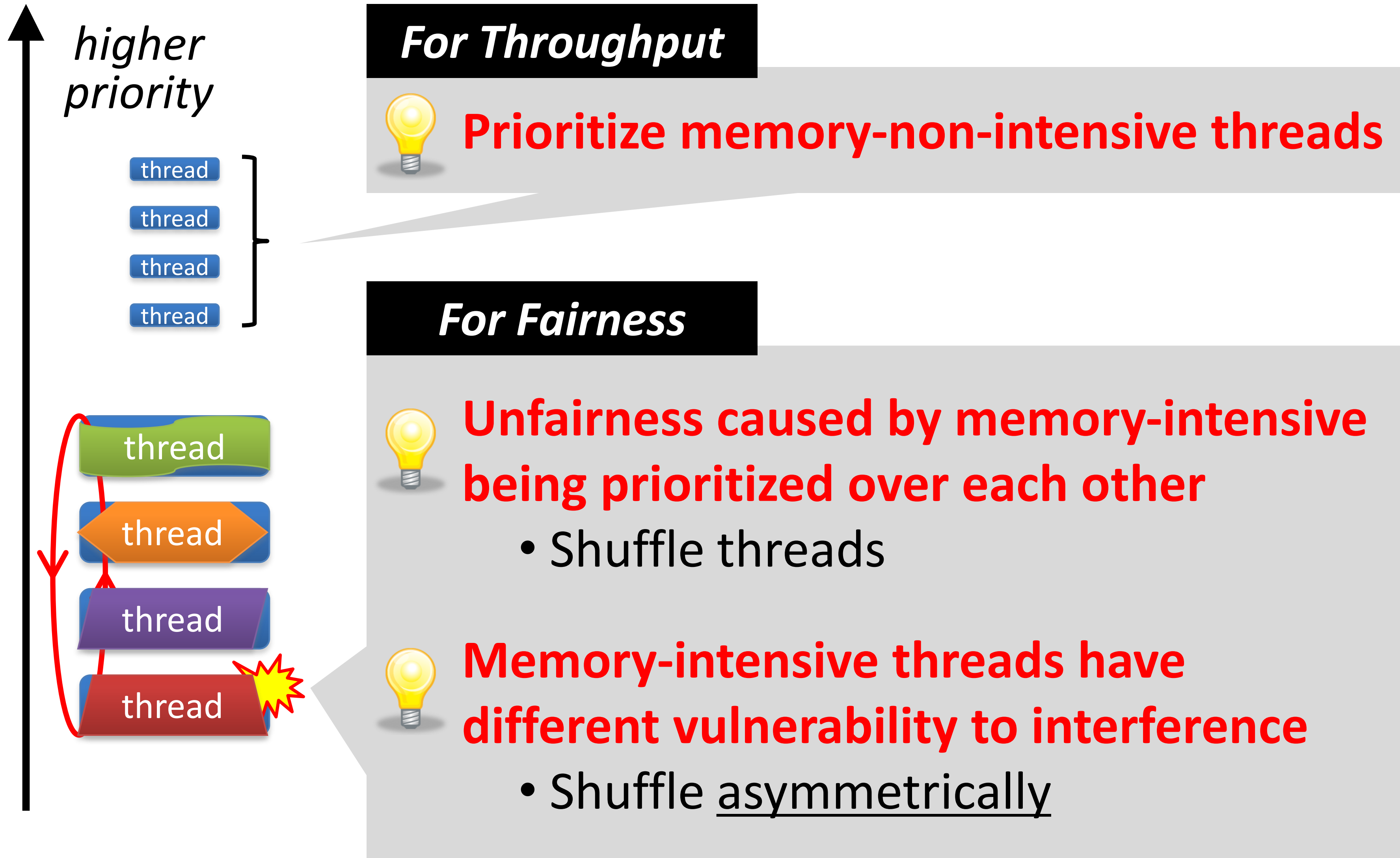

**Does not starve**

***starvation ➔ unfairness***

***not prioritized ➔ reduced throughput***

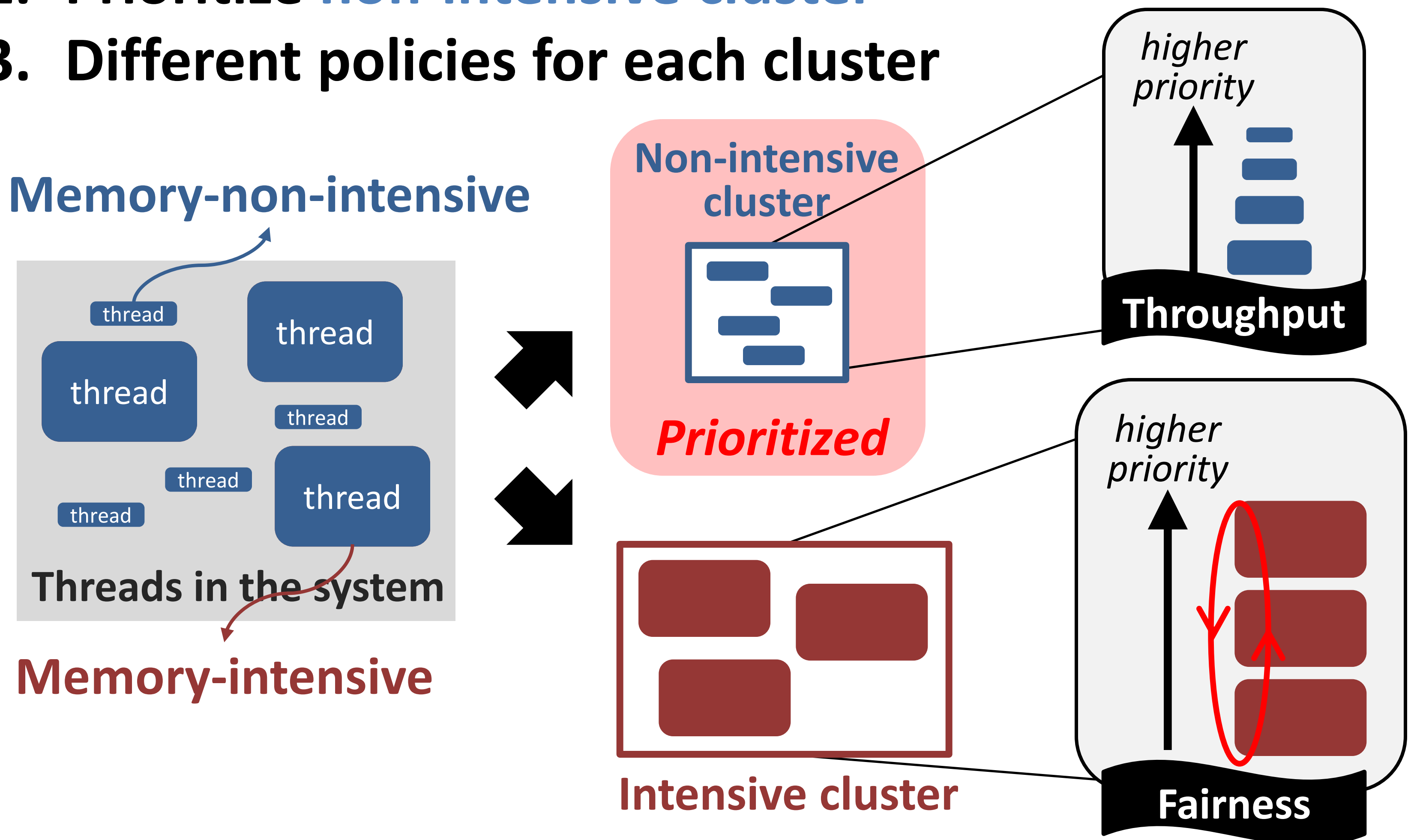
**Single policy for all threads is insufficient**

# Insight: Achieving Best of Both Worlds



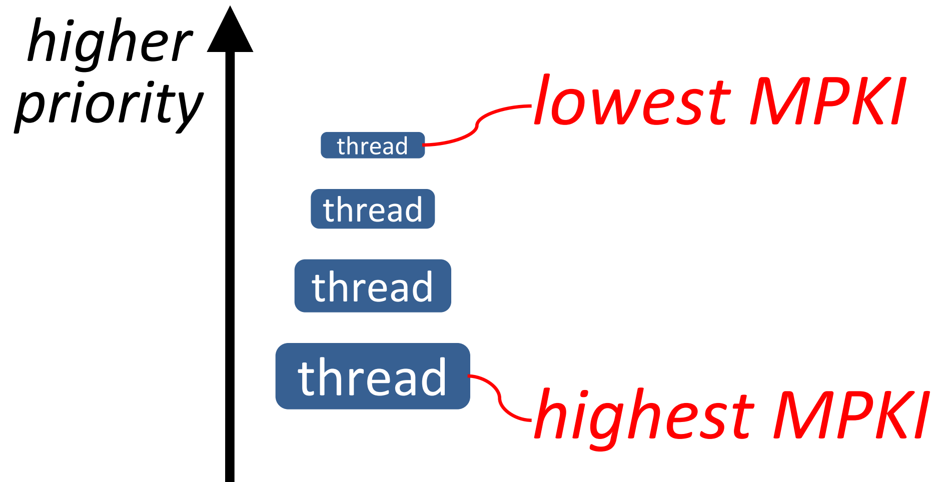
# Overview: Thread Cluster Memory Scheduling

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



# Non-Intensive Cluster

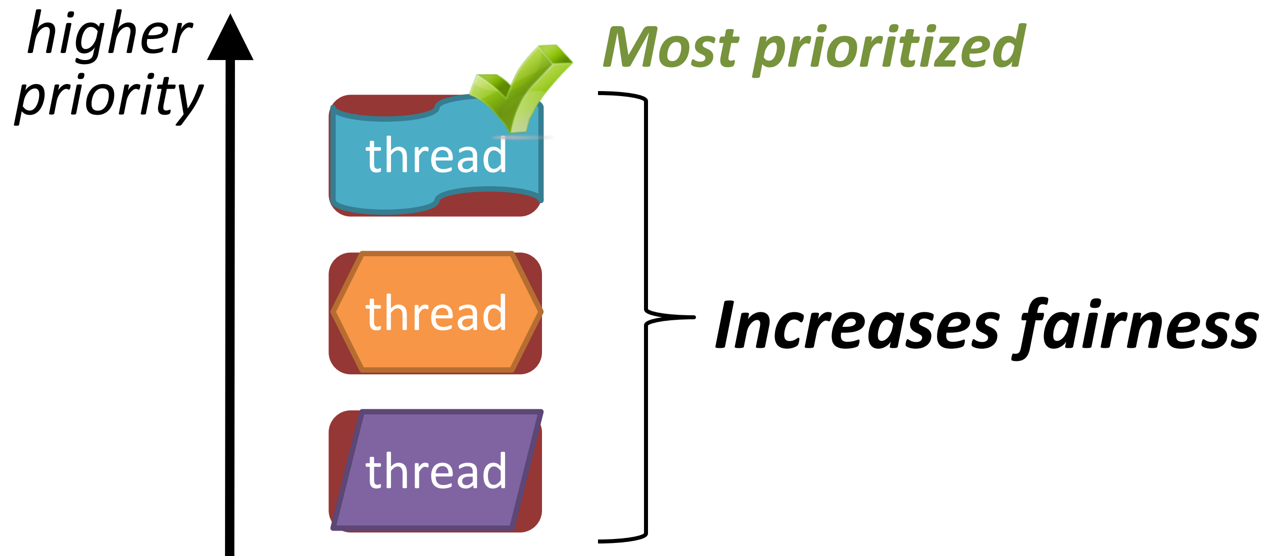
***Prioritize threads according to MPKI***




- **Increases system throughput**
  - Least intensive thread has the greatest potential for making progress in the processor

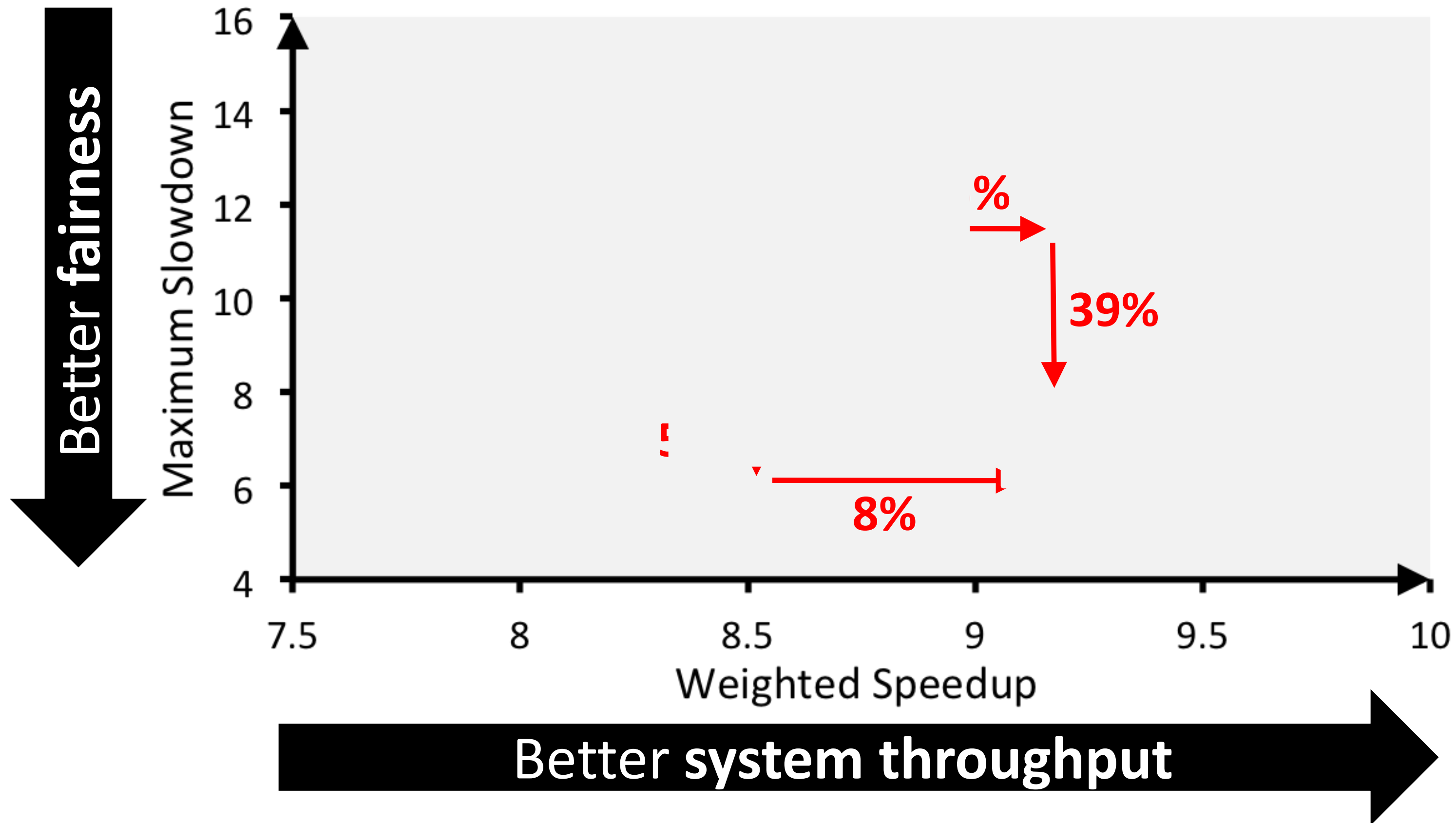
# Intensive Cluster

***Periodically shuffle the priority of threads***



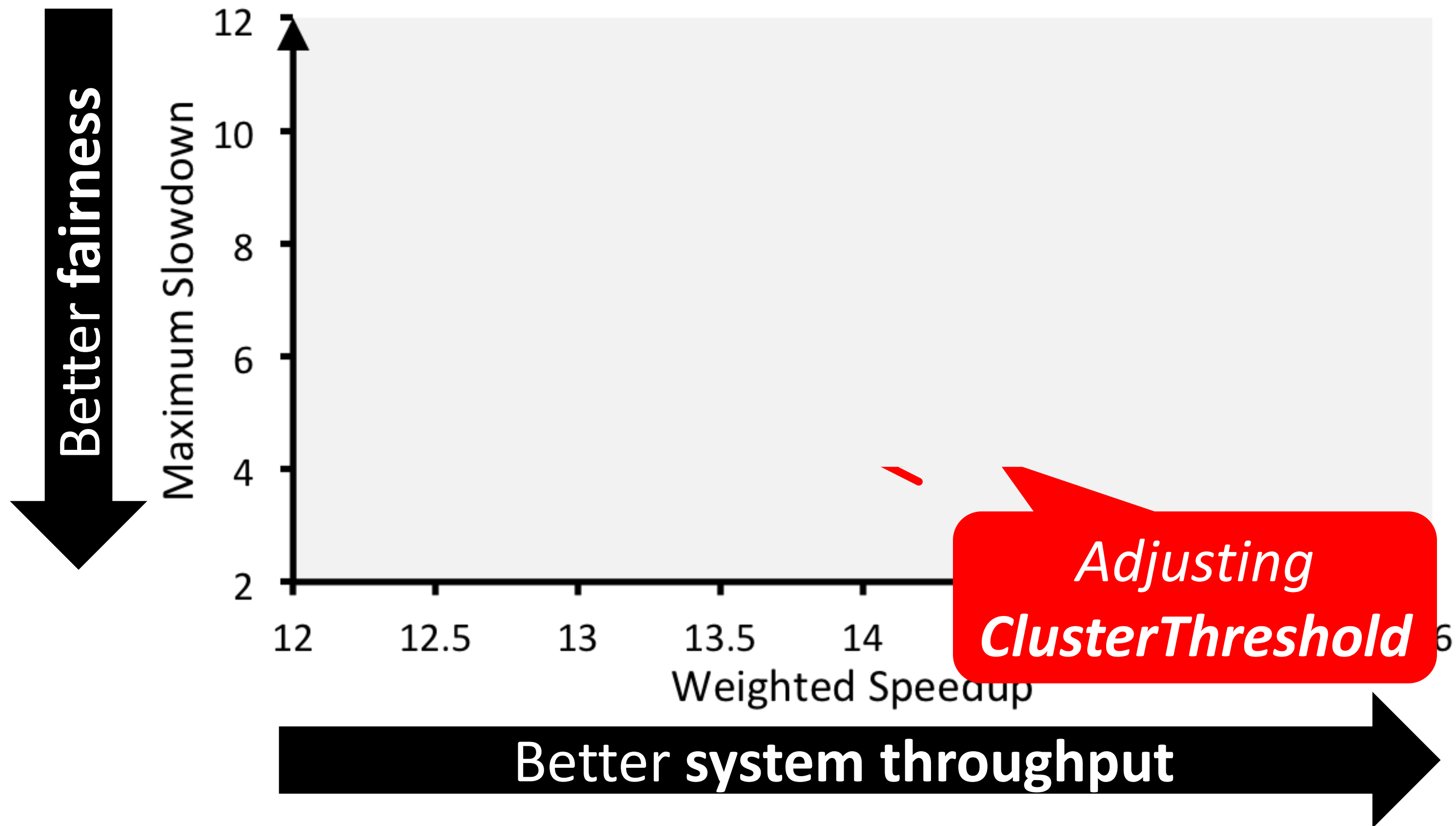

- Is treating all threads equally good enough?
- ***BUT:*** *Equal turns ≠ Same slowdown*

# Results: Fairness vs. Throughput




*TCM provides best fairness and system throughput*

# Results: Fairness-Throughput Tradeoff




*TCM allows robust fairness-throughput tradeoff*

# TCM Summary

- No previous memory scheduling algorithm provides both high ***system throughput*** and ***fairness***
  - **Problem:** They use a single policy for all threads

- TCM is a heterogeneous scheduling policy
  1. Prioritize ***non-intensive*** cluster → throughput
  2. Shuffle priorities in ***intensive*** cluster → fairness
  3. Shuffling should favor ***nice*** threads → fairness

- *Heterogeneity in memory scheduling provides the best system throughput and fairness*

# More Details on TCM

- Kim et al., "Thread Cluster Memory Scheduling: Exploiting Differences in Memory Access Behavior," MICRO 2010, Top Picks 2011.

# Memory Control in CPU-GPU Systems

- **Observation:** Heterogeneous CPU-GPU systems require memory schedulers with large request buffers
- **Problem:** Existing monolithic application-aware memory scheduler designs are hard to scale to large request buffer sizes
- **Solution:** Staged Memory Scheduling (SMS) decomposes the memory controller into three simple stages:
  1) Batch formation: maintains row buffer locality
  2) Batch scheduler: reduces interference between applications
  3) DRAM command scheduler: issues requests to DRAM
- Compared to state-of-the-art memory schedulers:
  - SMS is significantly simpler and more scalable
  - SMS provides higher performance and fairness

Ausavarungnirun et al., "Staged Memory Scheduling," ISCA 2012.

# Asymmetric Memory QoS in a Parallel Application

- 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 that is falling behind the most in a *parallel for* loop


Ebrahimi et al., "Parallel Application Memory Scheduling," MICRO 2011.

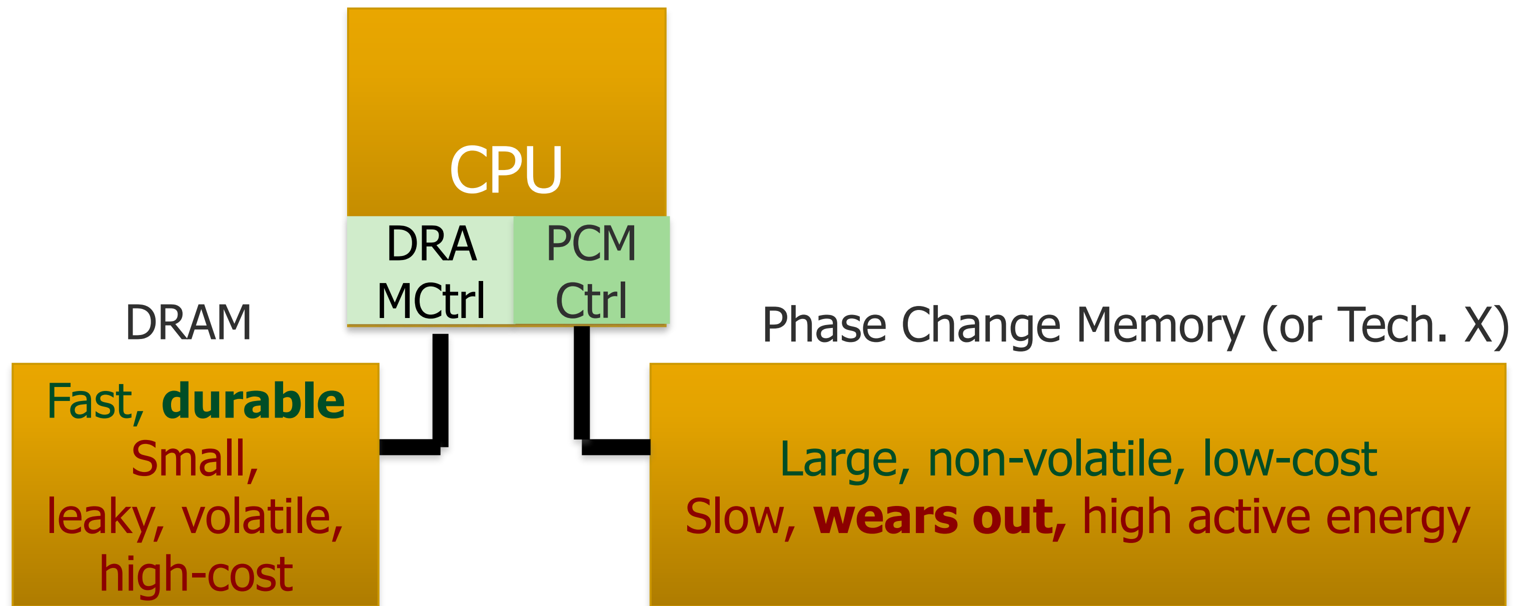
# Outline

- How Do We Get There: Examples

- Accelerated Critical Sections (ACS)
- Bottleneck Identification and Scheduling (BIS)
- Staged Execution and Data Marshaling

- Asymmetry in Memory
  - Thread Cluster Memory Scheduling
  - Heterogeneous DRAM+NVM Main Memory

# Heterogeneous Memory Systems




Hardware/software manage data allocation and movement
to achieve the best of multiple technologies

Meza, Chang, Yoon, Mutlu, Ranganathan, "Enabling Efficient and Scalable Hybrid Memories," IEEE Comp. Arch. Letters, 2012.

# One Option: DRAM as a Cache for PCM

- PCM is main memory; DRAM caches memory rows/blocks
  - Benefits: Reduced latency on DRAM cache hit; write filtering
- Memory controller hardware manages the DRAM cache
  - Benefit: Eliminates system software overhead

- Three issues:
  - What data should be placed in DRAM versus kept in PCM?
  - What is the granularity of data movement?
  - How to design a low-cost hardware-managed DRAM cache?

- Two idea directions:
  - Locality-aware data placement **[Yoon+ , ICCD 2012]**
  - Cheap tag stores and dynamic granularity **[Meza+, IEEE CAL 2012]**

# Summary

- Applications and phases have varying performance requirements
- Designs evaluated on multiple metrics/constraints: energy, performance, reliability, fairness, …
- One-size-fits-all design cannot satisfy all requirements and metrics: cannot get the best of all worlds
- Asymmetry enables tradeoffs: can get the best of all worlds
  - Asymmetry in core microarch. → Accelerated Critical Sections, BIS, DM → Good parallel performance + Good serialized performance
  - Asymmetry in memory scheduling → Thread Cluster Memory Scheduling → Good throughput + good fairness
  - Asymmetry in main memory → Data Management for DRAM-PCM Hybrid Memory → Good performance + good efficiency
- Simple asymmetric designs can be effective and low-cost