

18-447

# Computer Architecture

## Lecture 31: Asymmetric Multi-Core

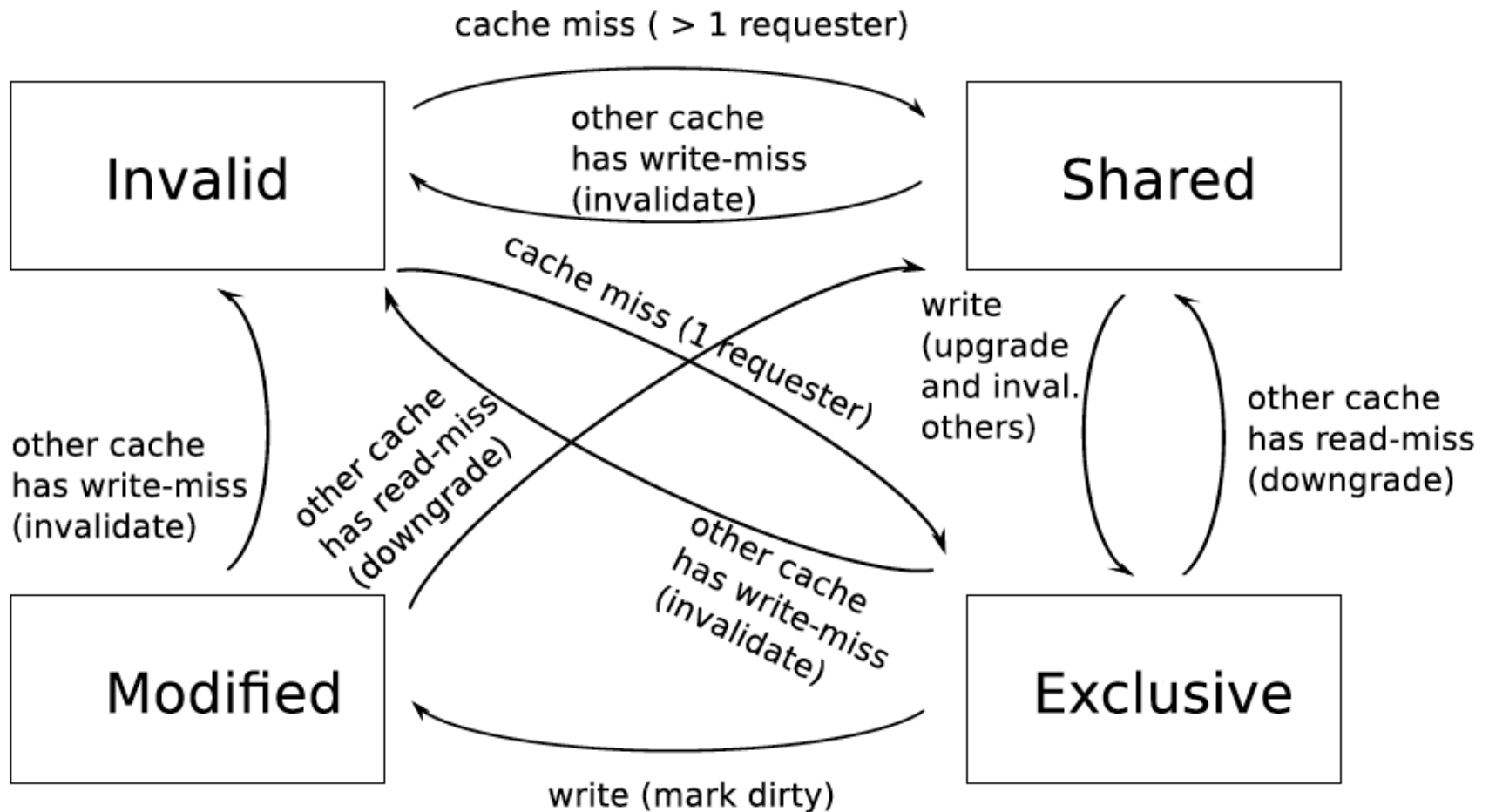
Prof. Onur Mutlu

Carnegie Mellon University

Spring 2014, 4/30/2014

# Lab 7: Multi-Core Cache Coherence

- Due May 2; Last submission accepted on May 9, 11:59pm
- Cycle-level modeling of the MESI cache coherence protocol



# A Note on Testing Your Own Code

---

- We provide the reference simulator to aid you
- Do not expect it to be given, and do not rely on it much
- In real life, there are no reference simulators
- The architect designs the reference simulator
- The architect verifies it
- The architect tests it
- The architect fixes it
- The architect makes sure there are no bugs
- The architect ensures the simulator matches the specification

# Lab 7 Grading

---

- Lab 7 is worth 5% of your entire course grade
- 3% of this is my gift to each of you 😊
- 2% of this you can earn by getting a grade between 0/100 and 40/100 (if you submit)
- An additional 3% will be extra credit you can earn by getting a grade between 40/100 and 100/100 (if you submit)
  
- Lab 7 grade is calculated as
  - Effect on total course grade =  $3\% + [5\% * \text{grade}/100]$
  - Everyone gets 3% on the lab without even submitting it
  - Those who submit the lab can get the remaining 2% and up to 3% more extra credit.

# Final Exam: May 6

---

- May 6, 8:30-11:30am, Hamerschlag Hall B103
- Comprehensive (over **all topics** in course)
- Three cheat sheets allowed
- We might have a review session
- Remember this is 25% of your grade
  - I will take into account your improvement over the course
  - Know all concepts, especially the previous midterm concepts
  - Same advice as before for Midterms I and II

# A Note on 742, Research, Jobs

---

- I am teaching **Parallel Computer Architecture** next semester (Fall 2014)
  - Deep dive into many topics we covered
  - And, many topics we did not cover
  - Research oriented with an open-ended research project
  - Cutting edge research and topics in HW/SW interface
- If you are enjoying 447 and are doing well, you can take it
  - no need to have taken 640/740
  - talk with me
- If you are excited about Computer Architecture research or looking for a job/internship in this area
  - talk with me

# Course Evaluations (due May 12)

---

- Due May 12
- Please do not forget to fill out the course evaluations
  - <http://www.cmu.edu/hub/fce/>
- Your feedback is very important
  
- I read these very carefully, and take into account every piece of feedback
  - And, improve the course for the future
- Please take the time to write out feedback
  - State the things you liked, topics you enjoyed, and what we can improve on
  - Don't just say "the course is hard" because you knew that from the very beginning!

# Last Lecture

---

- Wrap up cache coherence
  - VI → MSI → MESI → MOESI → ?
  - Directory vs. snooping tradeoffs
  - Scaling the directory based protocols
  
- Interconnects
  - Why important?
  - Topologies
  - Routing algorithms
  - Handling contention
  - On-chip interconnects



# Today

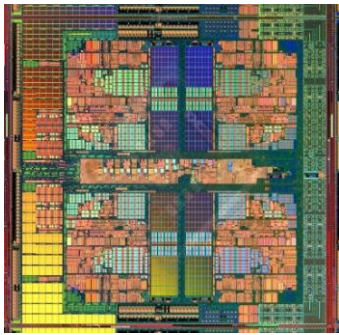
---

- Evolution of multi-core systems
- Handling serial and parallel bottlenecks better
- Heterogeneous multi-core systems

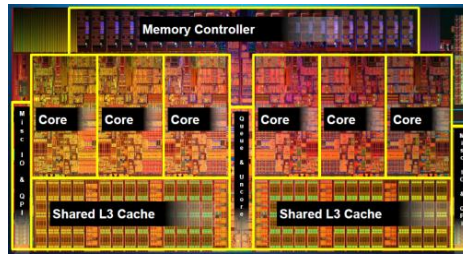
# Multi-Core Design

# Many Cores on Chip

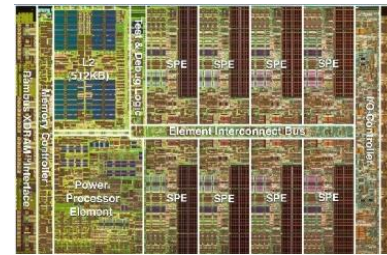
- Simpler and lower power than a single large core
- Large scale parallelism on chip



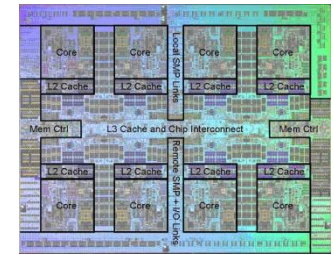
AMD Barcelona  
4 cores



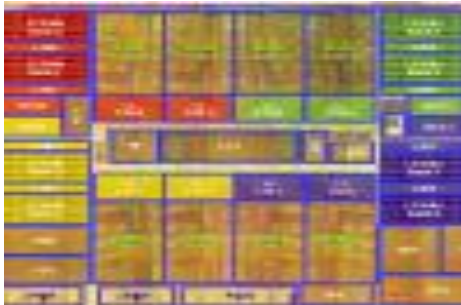
Intel Core i7  
8 cores



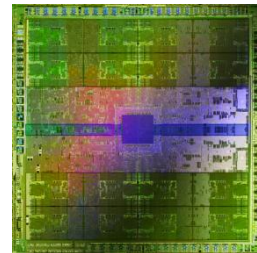
IBM Cell BE  
8+1 cores



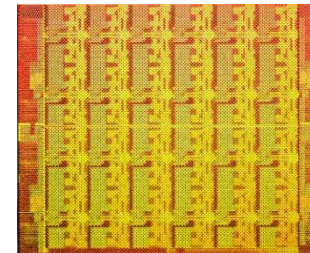
IBM POWER7  
8 cores



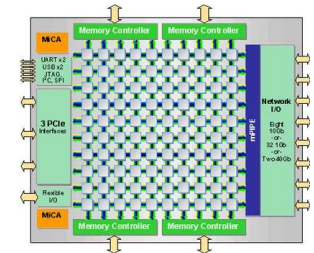
Sun Niagara II  
8 cores



Nvidia Fermi  
448 "cores"



Intel SCC  
48 cores, networked



Tiler TILE Gx  
100 cores, networked

# With Many Cores on Chip

---

- What we want:
  - N times the performance with N times the cores when we parallelize an application on N cores
  
- What we get:
  - Amdahl's Law (serial bottleneck)
  - Bottlenecks in the parallel portion

# Caveats of Parallelism

---

## ■ Amdahl's Law

- f: Parallelizable fraction of a program
- N: Number of processors

$$\text{Speedup} = \frac{1}{1 - f + \frac{f}{N}}$$

- Amdahl, “Validity of the single processor approach to achieving large scale computing capabilities,” AFIPS 1967.
- **Maximum speedup limited by serial portion: Serial bottleneck**
- **Parallel portion is usually not perfectly parallel**
  - **Synchronization** overhead (e.g., updates to shared data)
  - **Load imbalance** overhead (imperfect parallelization)
  - **Resource sharing** overhead (contention among N processors)

# The Problem: Serialized Code Sections

---

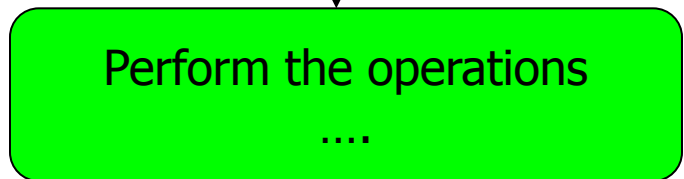
- Many parallel programs cannot be parallelized completely
- Causes of serialized code sections
  - Sequential portions (Amdahl's "serial part")
  - Critical sections
  - Barriers
  - Limiter stages in pipelined programs
- Serialized code sections
  - Reduce performance
  - Limit scalability
  - Waste energy

# Example from MySQL

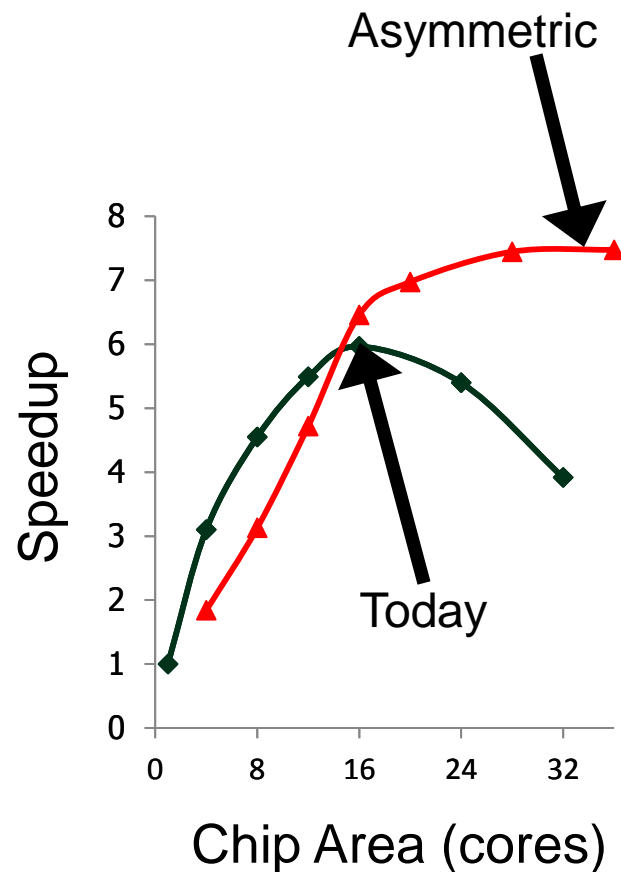
**Critical Section**



Access Open Tables Cache



Parallel



# Demands in Different Code Sections

---

- What we want:
- In a serialized code section → one powerful “large” core
- In a parallel code section → many wimpy “small” cores
- These two conflict with each other:
  - If you have a single powerful core, you cannot have many cores
  - A small core is much more energy and area efficient than a large core



# “Large” vs. “Small” Cores

## Large Core

- *Out-of-order*
- *Wide fetch e.g. 4-wide*
- *Deeper pipeline*
- *Aggressive branch predictor (e.g. hybrid)*
- *Multiple functional units*
- *Trace cache*
- *Memory dependence speculation*

## Small Core

- *In-order*
- *Narrow Fetch e.g. 2-wide*
- *Shallow pipeline*
- *Simple branch predictor (e.g. Gshare)*
- *Few functional units*

**Large Cores are power inefficient:  
e.g., 2x performance for 4x area (power)**

# Large vs. Small Cores

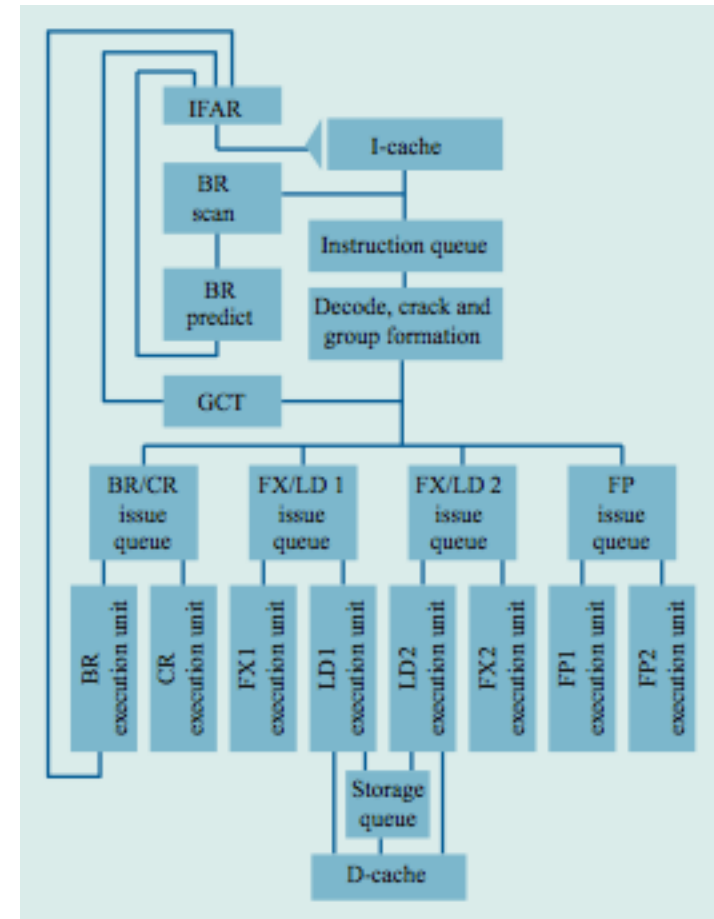
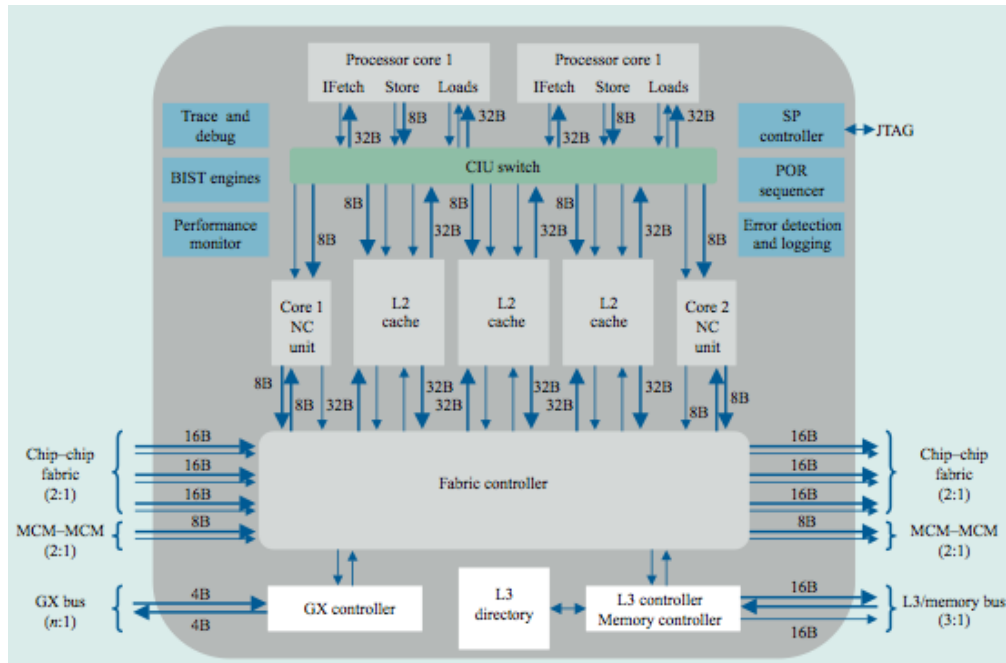
---

- Grochowski et al., “Best of both Latency and Throughput,” ICCD 2004.

	Large core	Small core
Microarchitecture	Out-of-order, 128-256 entry ROB	In-order
Width	3-4	1
Pipeline depth	20-30	5
Normalized performance	5-8x	1x
Normalized power	20-50x	1x
Normalized energy/instruction	4-6x	1x

# Meet Large: IBM POWER4

- Tendler et al., “POWER4 system microarchitecture,” IBM J R&D, 2002.
- Another symmetric multi-core chip...
- But, fewer and more powerful cores



# IBM POWER4

---

- 2 cores, out-of-order execution
- 100-entry instruction window in each core
- 8-wide instruction fetch, issue, execute
- Large, local+global hybrid branch predictor
- 1.5MB, 8-way L2 cache
- Aggressive stream based prefetching

# IBM POWER5

- Kalla et al., “IBM Power5 Chip: A Dual-Core Multithreaded Processor,” IEEE Micro 2004.

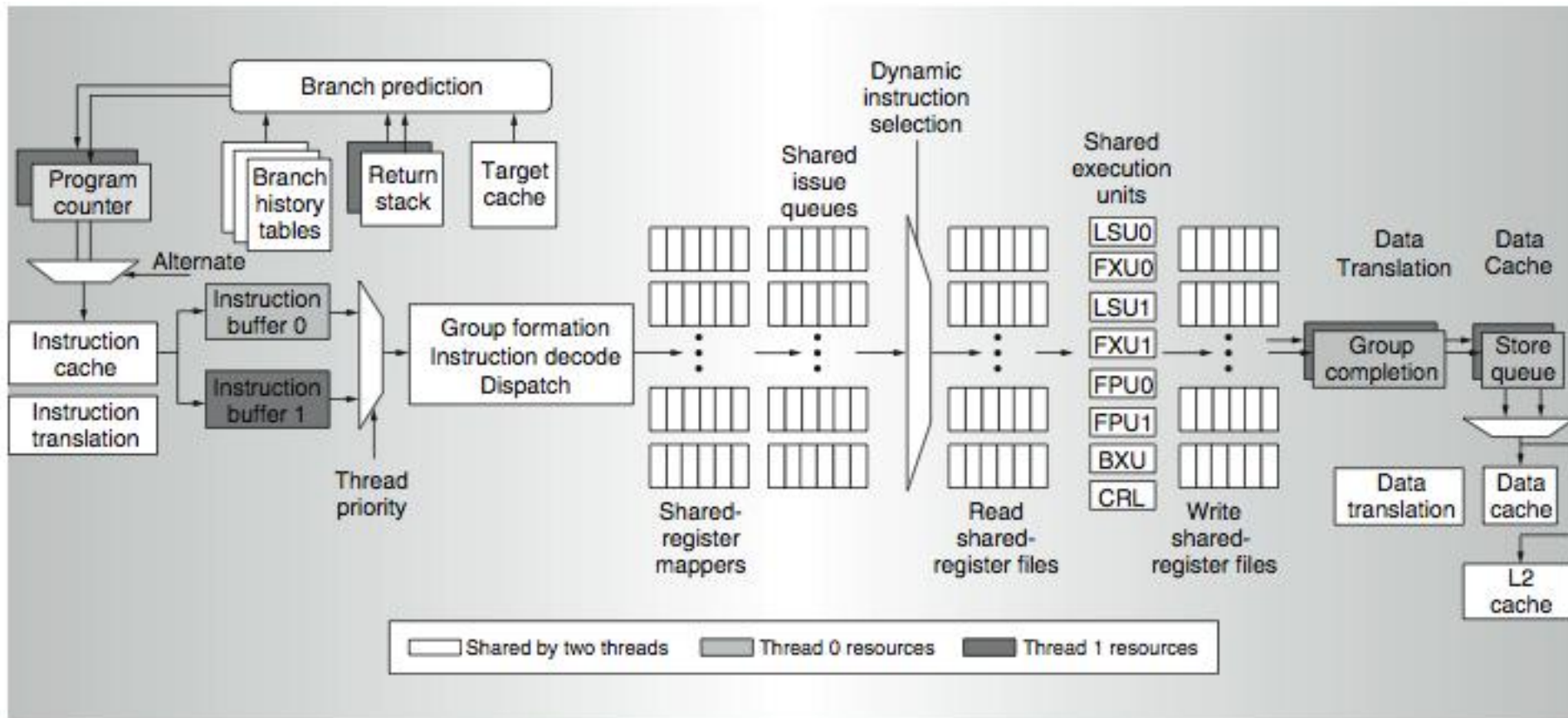
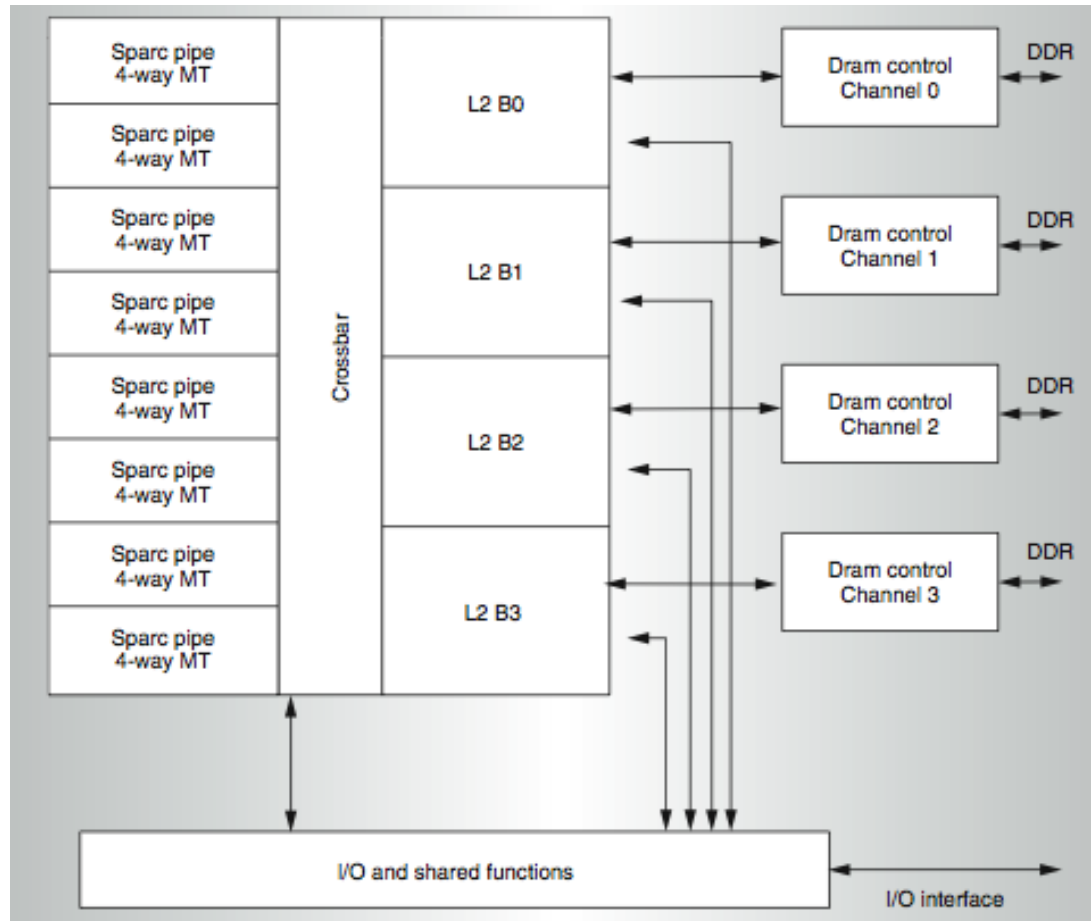


Figure 4. Power5 instruction data flow (BXU = branch execution unit and CRL = condition register logical execution unit).

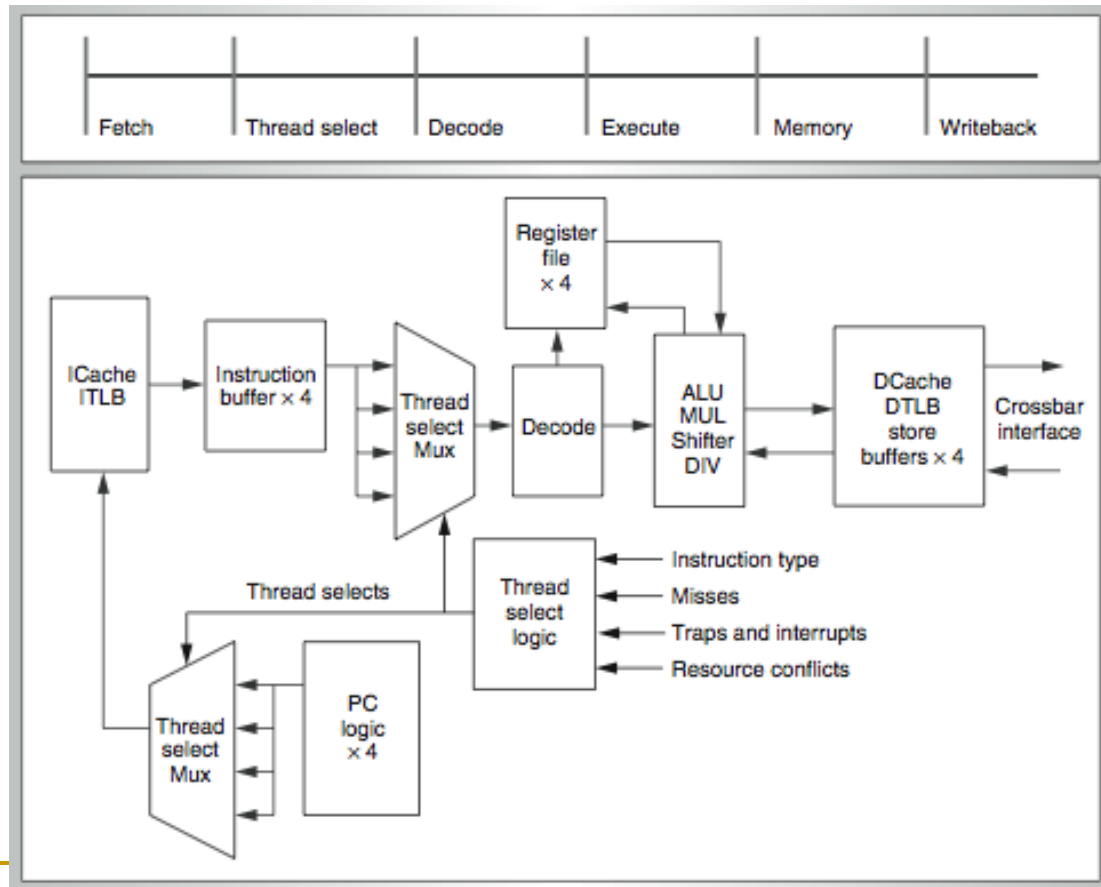
# Meet Small: Sun Niagara (UltraSPARC T1)

- Kongetira et al., “Niagara: A 32-Way Multithreaded SPARC Processor,” IEEE Micro 2005.



# Niagara Core

- 4-way fine-grain multithreaded, 6-stage, dual-issue in-order
- Round robin thread selection (unless cache miss)
- Shared FP unit among cores



# Remember the Demands

---

- What we want:
- In a serialized code section → one powerful “large” core
- In a parallel code section → many wimpy “small” cores
- These two conflict with each other:
  - If you have a single powerful core, you cannot have many cores
  - A small core is much more energy and area efficient than a large core
- Can we get the best of both worlds?



# Performance vs. Parallelism

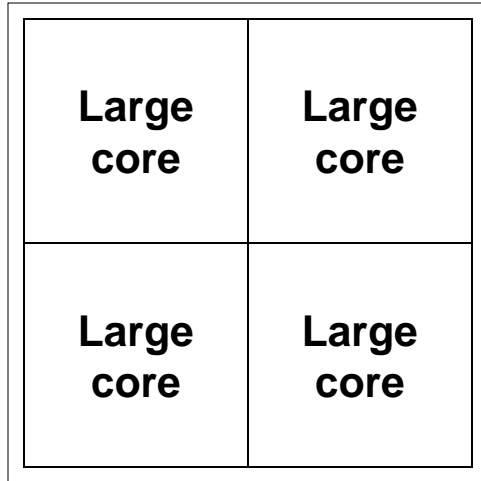
---

## *Assumptions:*

- 1. Small cores takes an area budget of 1 and has performance of 1*
- 2. Large core takes an area budget of 4 and has performance of 2*

# Tile-Large Approach

---



“Tile-Large”

- Tile a few large cores
- IBM Power 5, AMD Barcelona, Intel Core2Quad, Intel Nehalem
- + High performance on single thread, serial code sections (2 units)
- Low throughput on parallel program portions (8 units)

# Tile-Small Approach

---

Small core	Small core	Small core	Small core
Small core	Small core	Small core	Small core
Small core	Small core	Small core	Small core
Small core	Small core	Small core	Small core

“Tile-Small”

- Tile many small cores
- Sun Niagara, Intel Larrabee, Tiler TILE (tile ultra-small)
  - + High throughput on the parallel part (16 units)
  - Low performance on the serial part, single thread (1 unit)

# Can we get the best of both worlds?

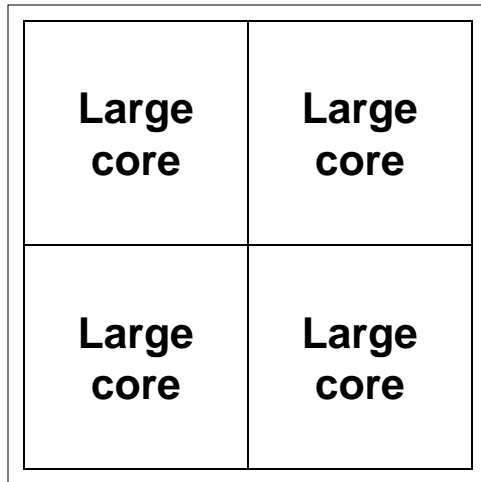
---

- Tile Large
  - + High performance on single thread, serial code sections (2 units)
  - Low throughput on parallel program portions (8 units)
- Tile Small
  - + High throughput on the parallel part (16 units)
  - Low performance on the serial part, single thread (1 unit),  
reduced single-thread performance compared to existing single thread processors
- Idea: Have both large and small on the same chip →  
Performance asymmetry

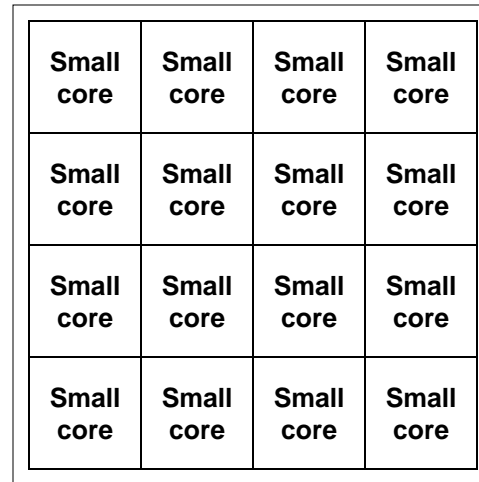
# Asymmetric Multi-Core

# Asymmetric Chip Multiprocessor (ACMP)

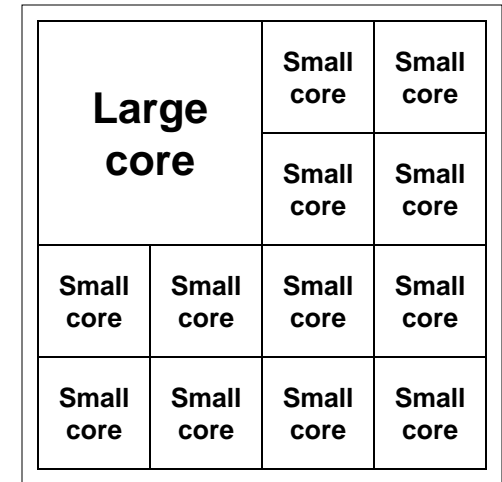
---



“Tile-Large”



“Tile-Small”



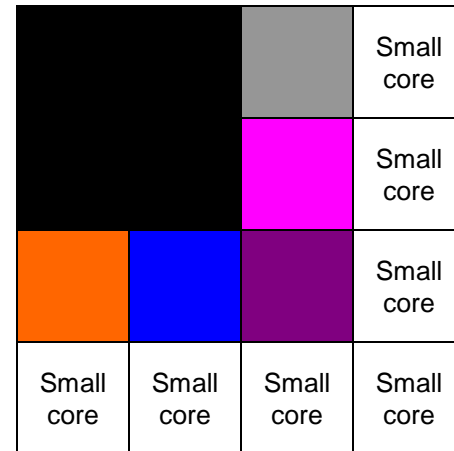
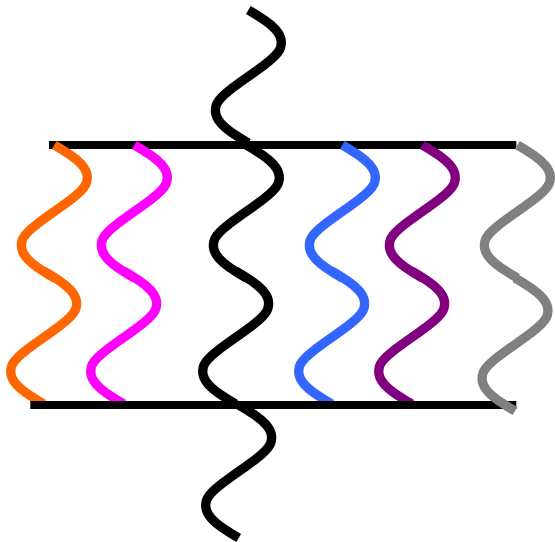
ACMP

- Provide one large core and many small cores
- + Accelerate serial part using the large core (2 units)
- + Execute parallel part on small cores and large core for high throughput (12+2 units)

# Accelerating Serial Bottlenecks

---

Single thread  $\rightarrow$  Large core



ACMP Approach

# Performance vs. Parallelism

---

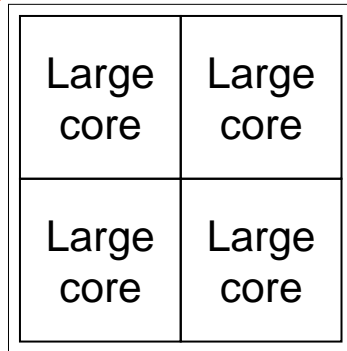
## *Assumptions:*

- 1. Small cores takes an area budget of 1 and has performance of 1*
- 2. Large core takes an area budget of 4 and has performance of 2*

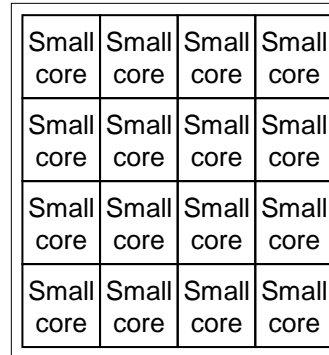


# ACMP Performance vs. Parallelism

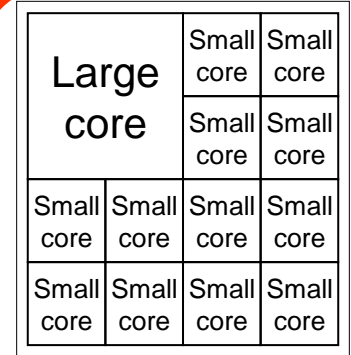
Area-budget = 16 small cores



“Tile-Large”



“Tile-Small”



ACMP

Large Cores	4	0	1
Small Cores	0	16	12
Serial Performance	2	1	2
Parallel Throughput	$2 \times 4 = 8$	$1 \times 16 = 16$	$1 \times 2 + 1 \times 12 = 14$

# Caveats of Parallelism, Revisited

---

## ■ Amdahl's Law

- f: Parallelizable fraction of a program
- N: Number of processors

$$\text{Speedup} = \frac{1}{1 - f + \frac{f}{N}}$$

- Amdahl, “Validity of the single processor approach to achieving large scale computing capabilities,” AFIPS 1967.
- **Maximum speedup limited by serial portion: Serial bottleneck**
- **Parallel portion is usually not perfectly parallel**
  - **Synchronization** overhead (e.g., updates to shared data)
  - **Load imbalance** overhead (imperfect parallelization)
  - **Resource sharing** overhead (contention among N processors)

# Accelerating Parallel Bottlenecks

---

- Serialized or imbalanced execution in the parallel portion can also benefit from a large core
- Examples:
  - Critical sections that are contended
  - Parallel stages that take longer than others to execute
- Idea: Dynamically identify these code portions that cause serialization and execute them on a large core

# Accelerated Critical Sections

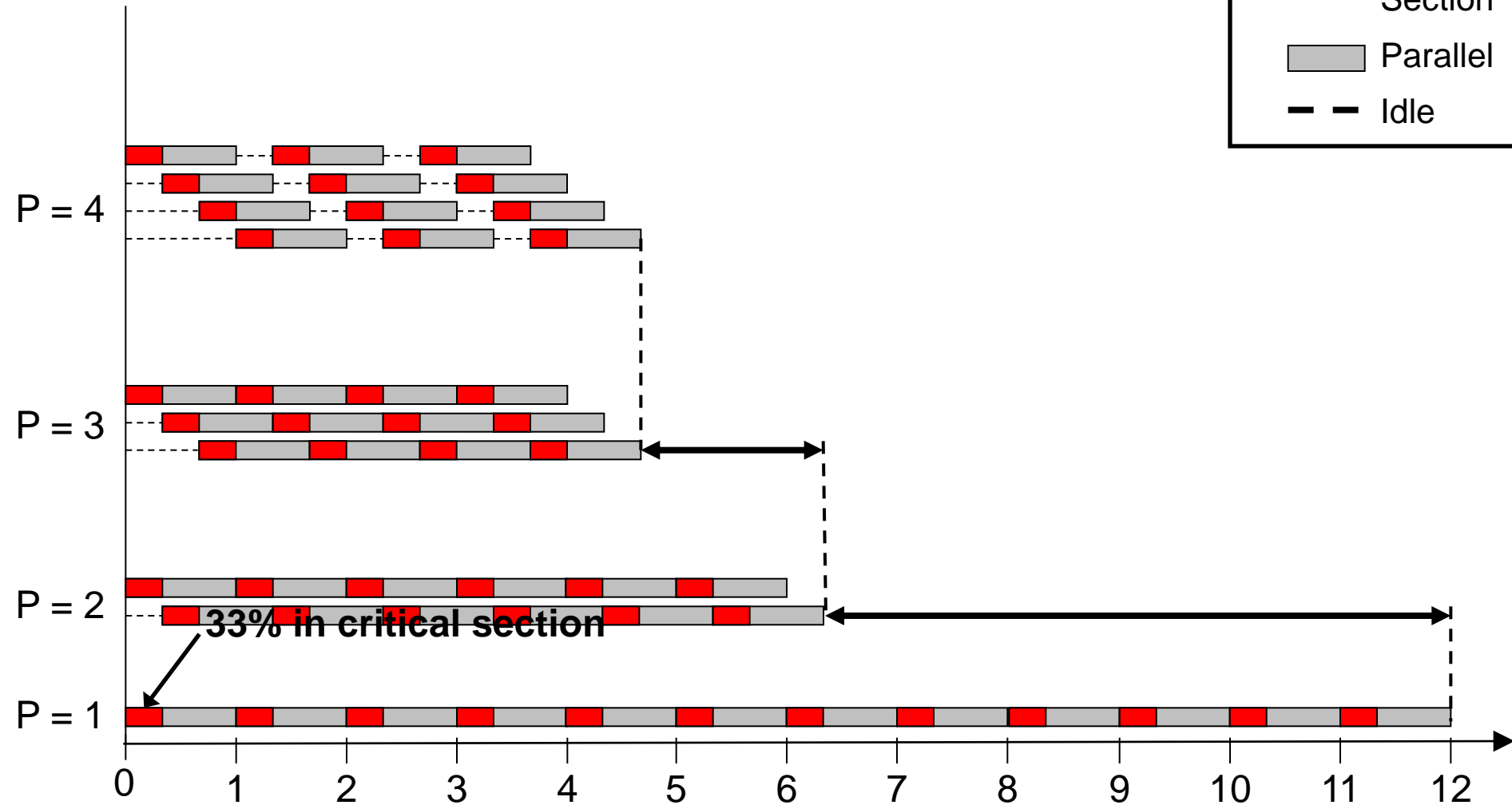
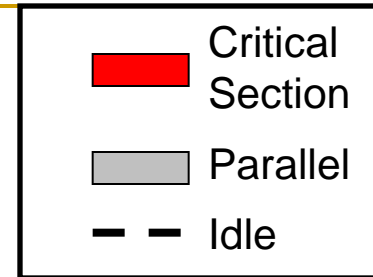
M. Aater Suleman, Onur Mutlu, Moinuddin K. Qureshi, and Yale N. Patt,

**"Accelerating Critical Section Execution with Asymmetric Multi-Core Architectures"**

*Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2009*

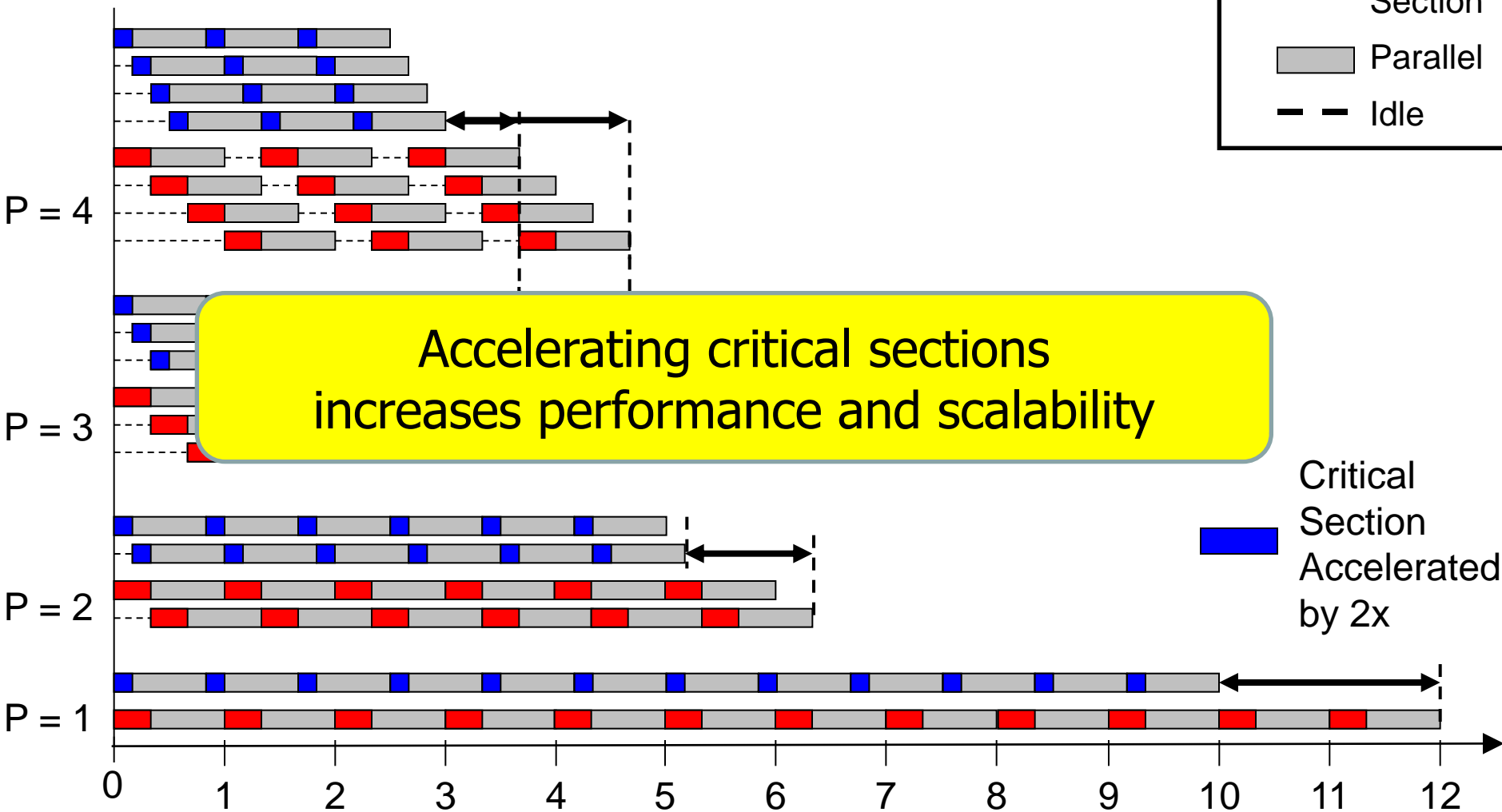
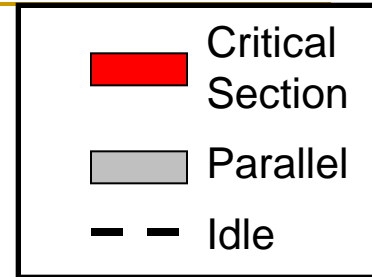
# Contention for Critical Sections

12 iterations, 33% instructions inside the critical section



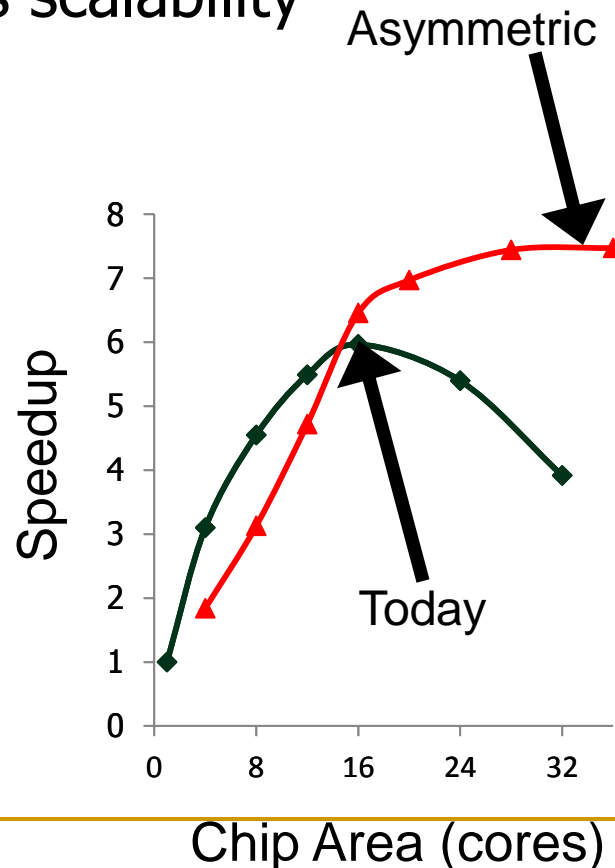
# Contention for Critical Sections

12 iterations, 33% instructions inside the critical section



# Impact of Critical Sections on Scalability

- Contention for critical sections leads to serial execution (serialization) of threads in the parallel program portion
- Contention for critical sections increases with the number of threads and limits scalability



MySQL (oltp-1)

# A Case for Asymmetry

---

- Execution time of sequential kernels, critical sections, and limiter stages must be short
- It is difficult for the programmer to shorten these serialized sections
  - Insufficient domain-specific knowledge
  - Variation in hardware platforms
  - Limited resources
- Goal: A mechanism to shorten serial bottlenecks without requiring programmer effort
- Idea: Accelerate serialized code sections by shipping them to powerful cores in an asymmetric multi-core (ACMP)



# An Example: Accelerated Critical Sections

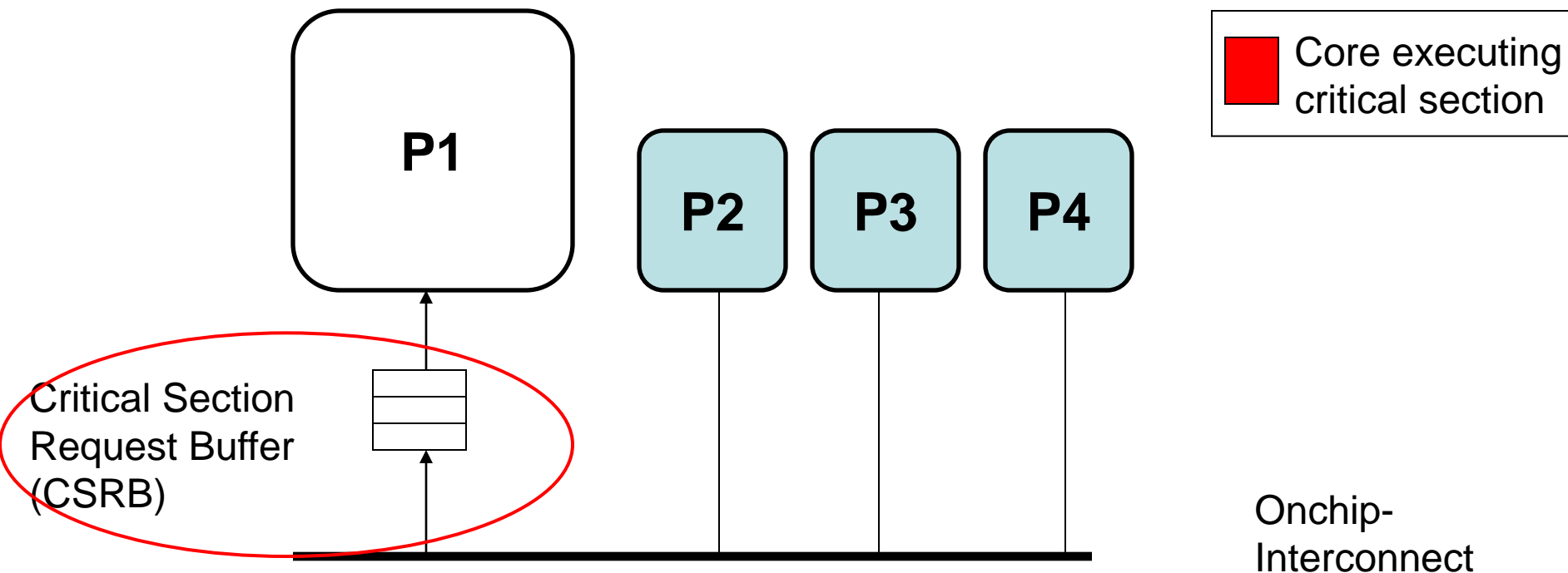
---

- Idea: HW/SW ships critical sections to a large, powerful core in an asymmetric multi-core architecture
- Benefit:
  - Reduces serialization due to contended locks
  - Reduces the performance impact of hard-to-parallelize sections
  - Programmer does not need to (heavily) optimize parallel code → fewer bugs, improved productivity
- Suleman et al., “Accelerating Critical Section Execution with Asymmetric Multi-Core Architectures,” ASPLOS 2009, IEEE Micro Top Picks 2010.
- Suleman et al., “Data Marshaling for Multi-Core Architectures,” ISCA 2010, IEEE Micro Top Picks 2011.

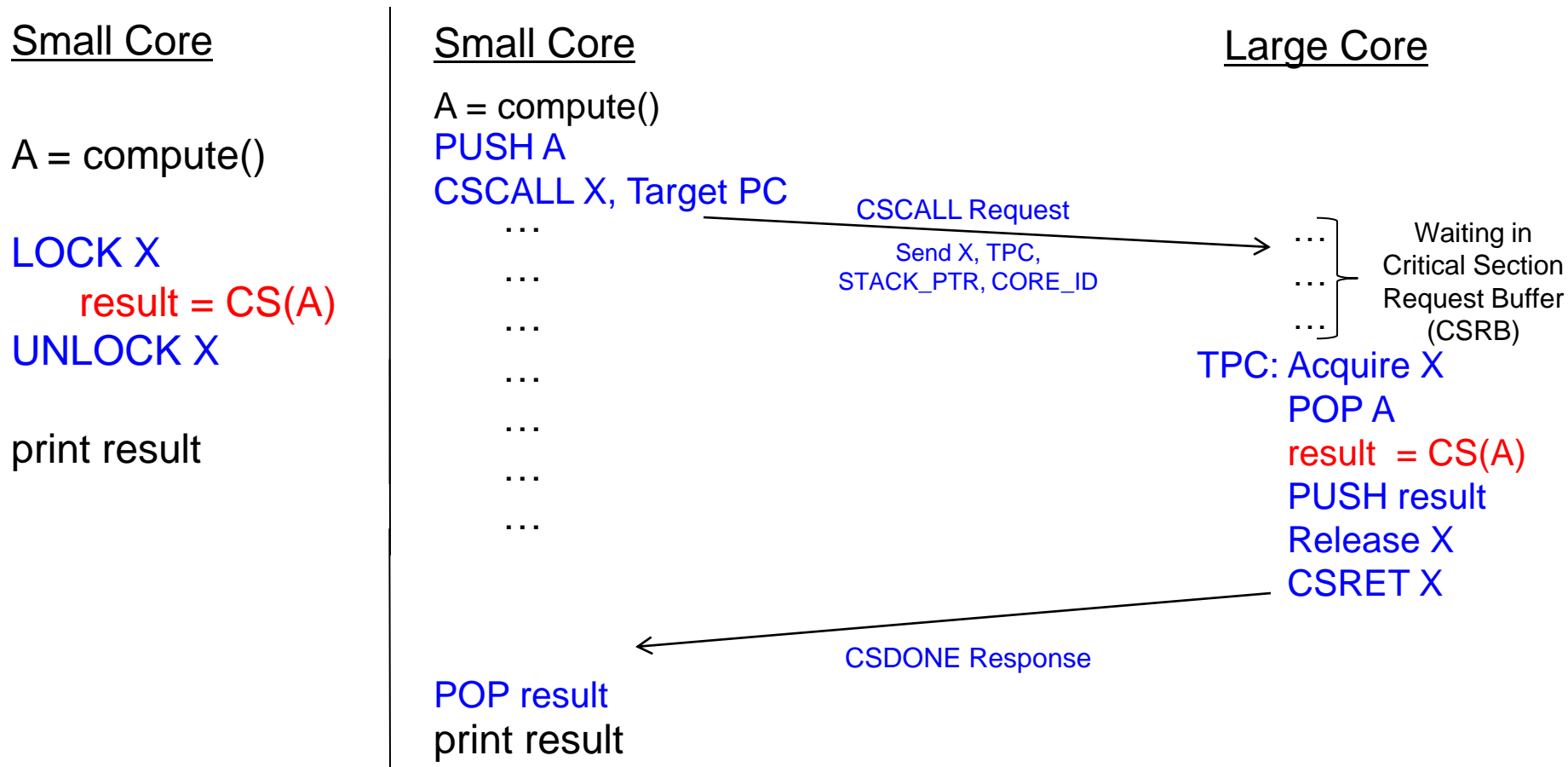
# Accelerated Critical Sections

```
EnterCS()  
    PriorityQ.insert(...)  
LeaveCS()
```

1. P2 encounters a critical section (CSCALL)
2. P2 sends CSCALL Request to CSRB
3. P1 executes Critical Section
4. P1 sends CSDONE signal



# Accelerated Critical Sections (ACS)

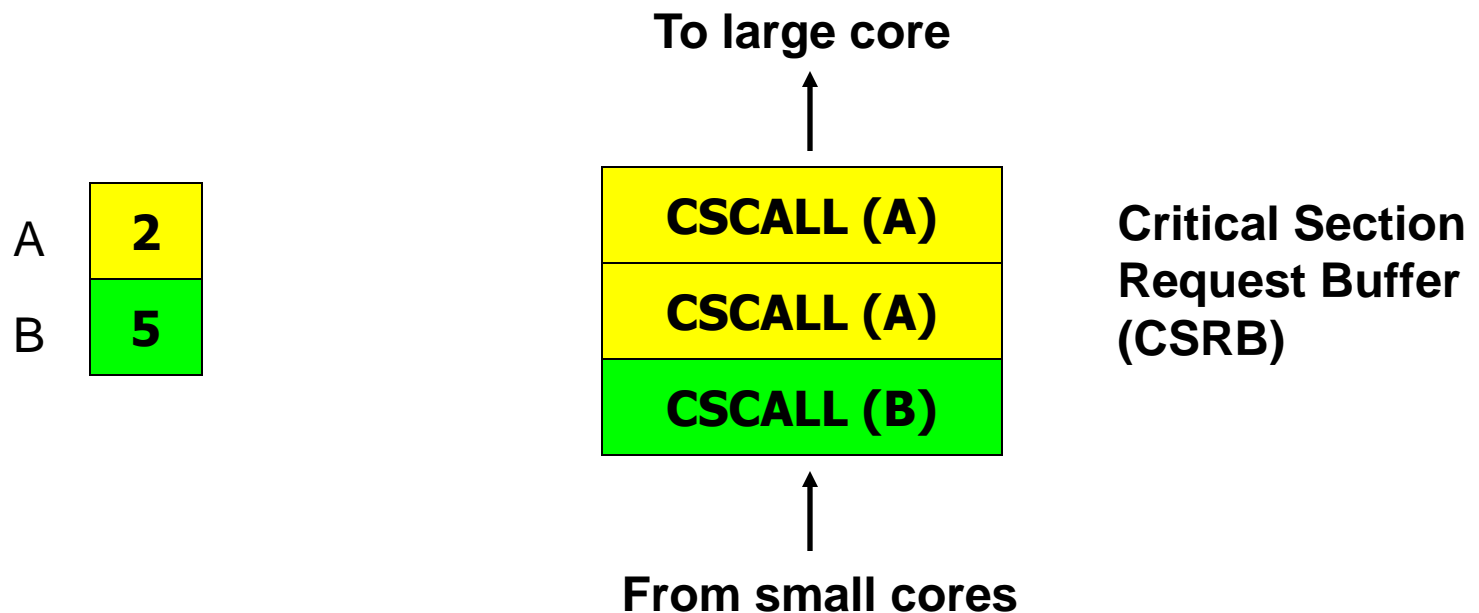


- Suleman et al., “Accelerating Critical Section Execution with Asymmetric Multi-Core Architectures,” ASPLOS 2009.

# False Serialization

---

- ACS can serialize independent critical sections
- Selective Acceleration of Critical Sections (SEL)
  - Saturating counters to track false serialization



# ACS Performance Tradeoffs

---

## ■ Pluses

- + Faster critical section execution
- + Shared locks stay in one place: better lock locality
- + Shared data stays in large core's (large) caches: better shared data locality, less ping-ponging

## ■ Minuses

- Large core dedicated for critical sections: reduced parallel throughput
- CSCALL and CSDONE control transfer overhead
- Thread-private data needs to be transferred to large core: worse private data locality

# ACS Performance Tradeoffs

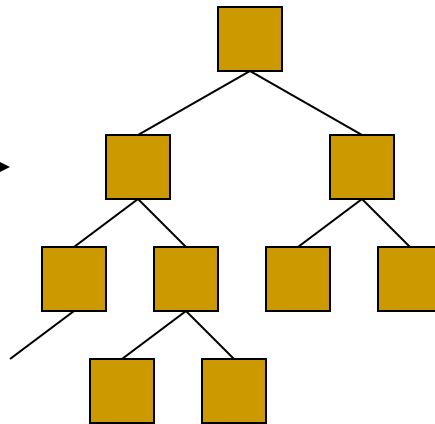
---

- ***Fewer parallel threads vs. accelerated critical sections***
  - Accelerating critical sections offsets loss in throughput
  - As the number of cores (threads) on chip increase:
    - Fractional loss in parallel performance decreases
    - Increased contention for critical sections makes acceleration more beneficial
- ***Overhead of CSCALL/CSDONE vs. better lock locality***
  - ACS avoids “ping-ponging” of locks among caches by keeping them at the large core
- ***More cache misses for private data vs. fewer misses for shared data***

# Cache Misses for Private Data

**PriorityHeap.insert(NewSubProblems)**

Private Data:  
NewSubProblems



Shared Data:  
The priority heap

**Puzzle Benchmark**

# ACS Performance Tradeoffs

---

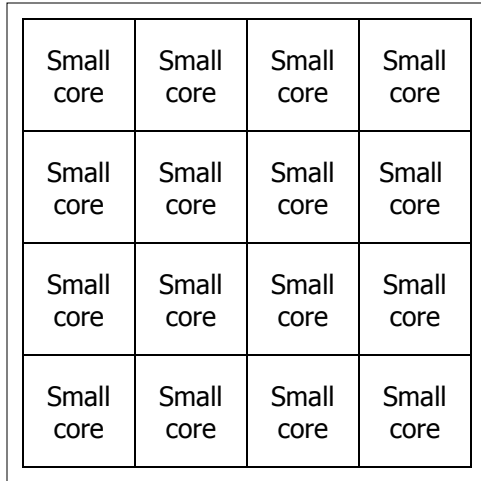
- ***Fewer parallel threads vs. accelerated critical sections***
  - Accelerating critical sections offsets loss in throughput
  - As the number of cores (threads) on chip increase:
    - Fractional loss in parallel performance decreases
    - Increased contention for critical sections makes acceleration more beneficial
- ***Overhead of CSCALL/CSDONE vs. better lock locality***
  - ACS avoids “ping-ponging” of locks among caches by keeping them at the large core
- ***More cache misses for private data vs. fewer misses for shared data***
  - Cache misses reduce if shared data > private data

**This problem can be solved**



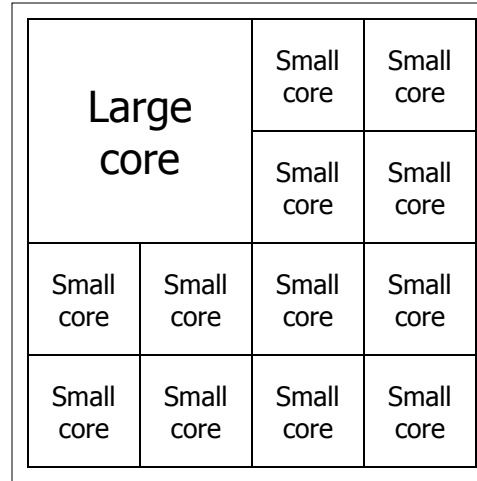
# ACS Comparison Points

---



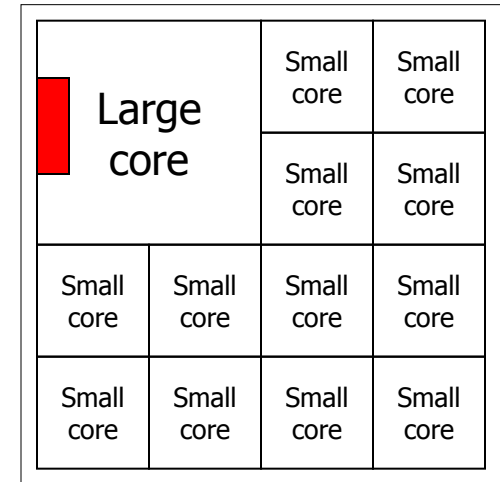
SCMP

- Conventional locking



ACMP

- Conventional locking
- Large core executes Amdahl's serial part



ACS

- Large core executes Amdahl's serial part and critical sections

# Accelerated Critical Sections: Methodology

---

- Workloads: 12 critical section intensive applications
  - Data mining kernels, sorting, database, web, networking
- 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

# ACS Performance

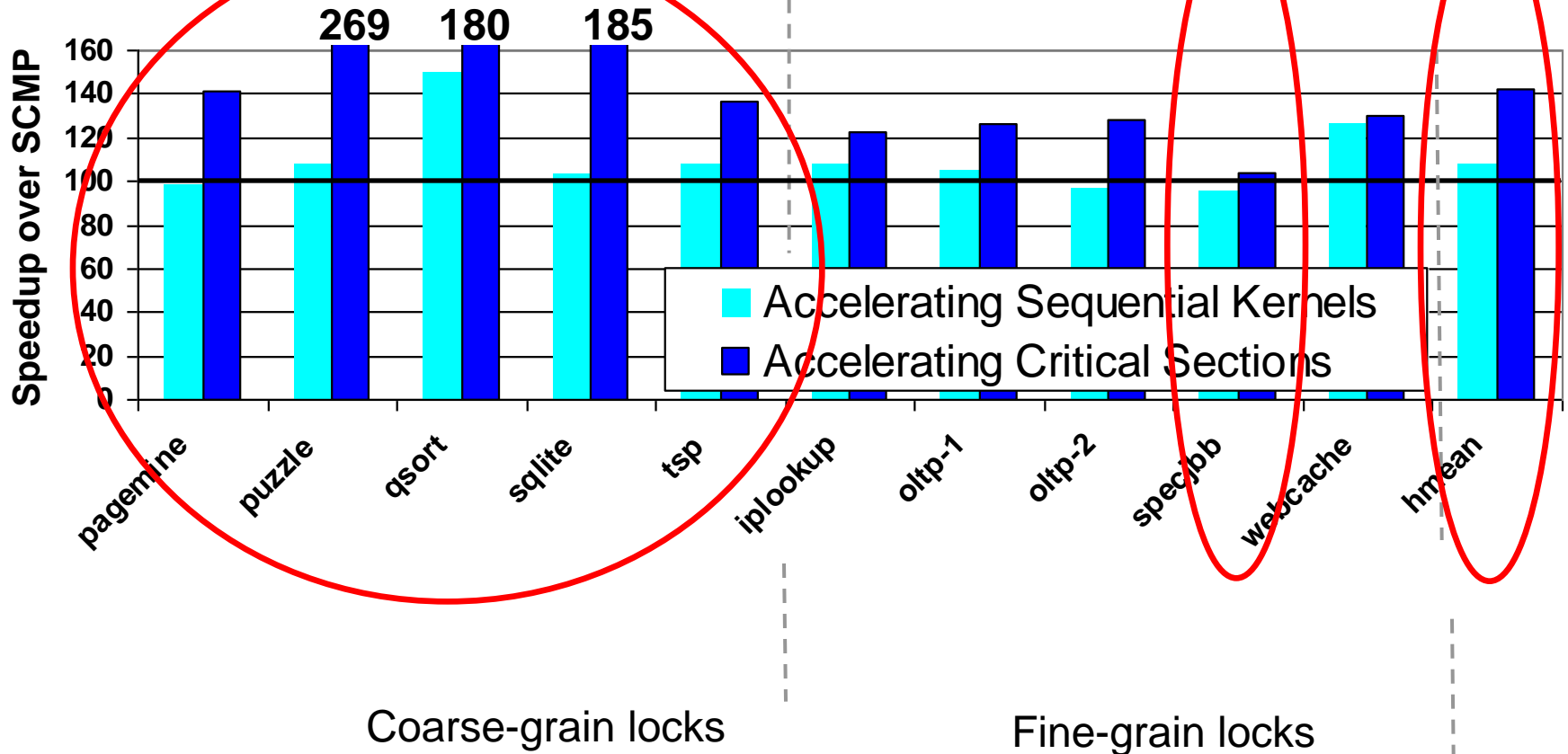
Chip Area = 32 small cores

SCMP = 32 small cores

ACMP = 1 large and 28 small cores

Equal-area comparison

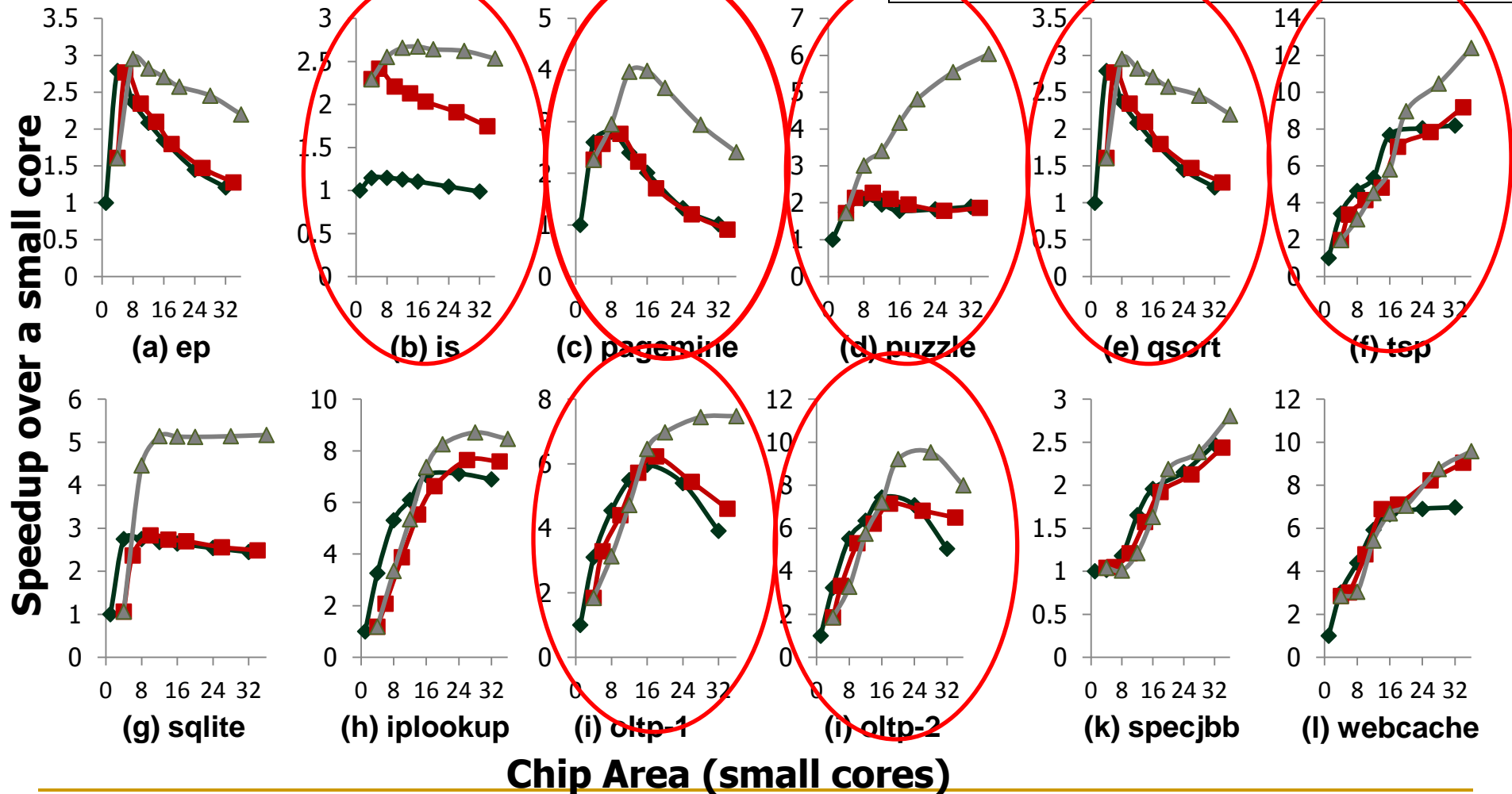
Number of threads = *Best threads*



# Equal-Area Comparisons

----- **SCMP**  
 ----- **ACMP**  
 ----- **ACS**

Number of threads = No. of cores



# ACS Summary

---

- Critical sections reduce performance and limit scalability
- Accelerate critical sections by executing them on a powerful core
- ACS reduces average execution time by:
  - 34% compared to an equal-area SCMP
  - 23% compared to an equal-area ACMP
- ACS improves scalability of 7 of the 12 workloads
- Generalizing the idea: Accelerate all bottlenecks (“critical paths”) by executing them on a powerful core

# Bottleneck Identification and Scheduling

Jose A. Joao, M. Aater Suleman, Onur Mutlu, and Yale N. Patt,

**"Bottleneck Identification and Scheduling in Multithreaded Applications"**

*Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), London, UK, March 2012.*

# BIS Summary

---

- **Problem:** Performance and scalability of multithreaded applications are limited by serializing synchronization 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



# Observation: Limiting Bottlenecks Change Over Time

A=full linked list; B=empty linked list

repeat

Lock A

    Traverse list A

    Remove X from A

Unlock A

Compute on X

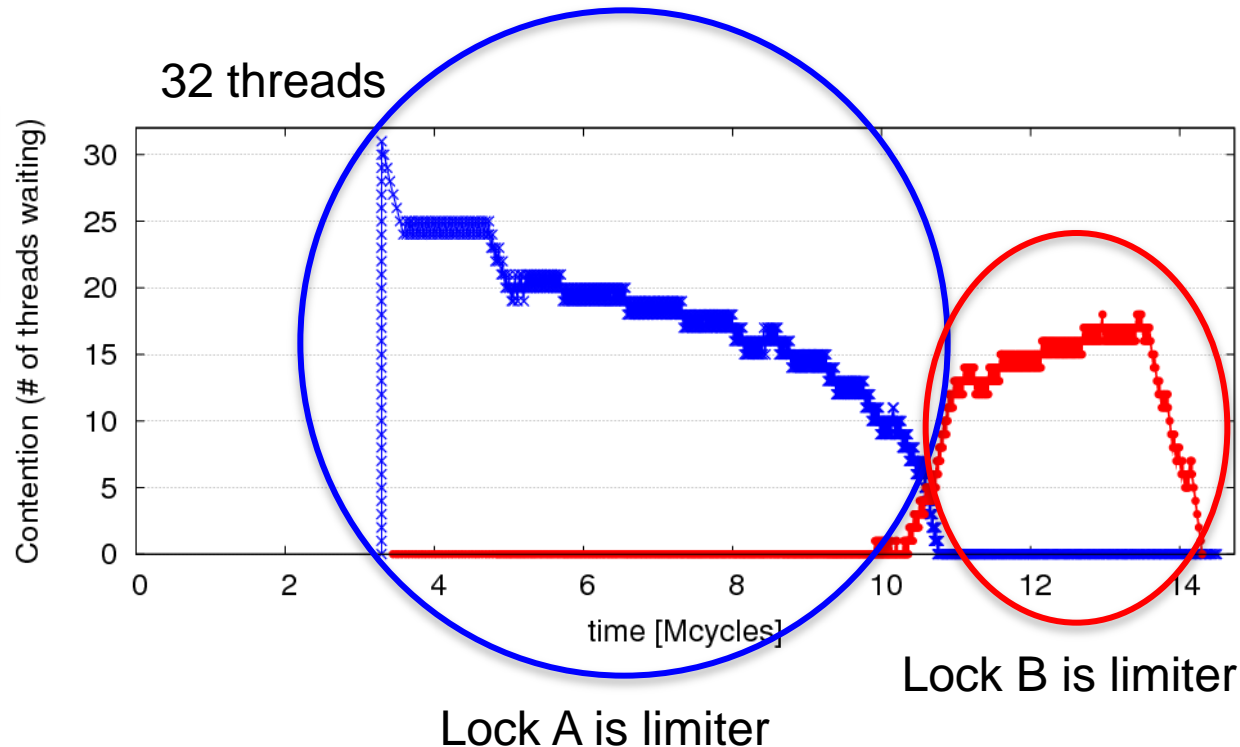
Lock B

    Traverse list B

    Insert X into B

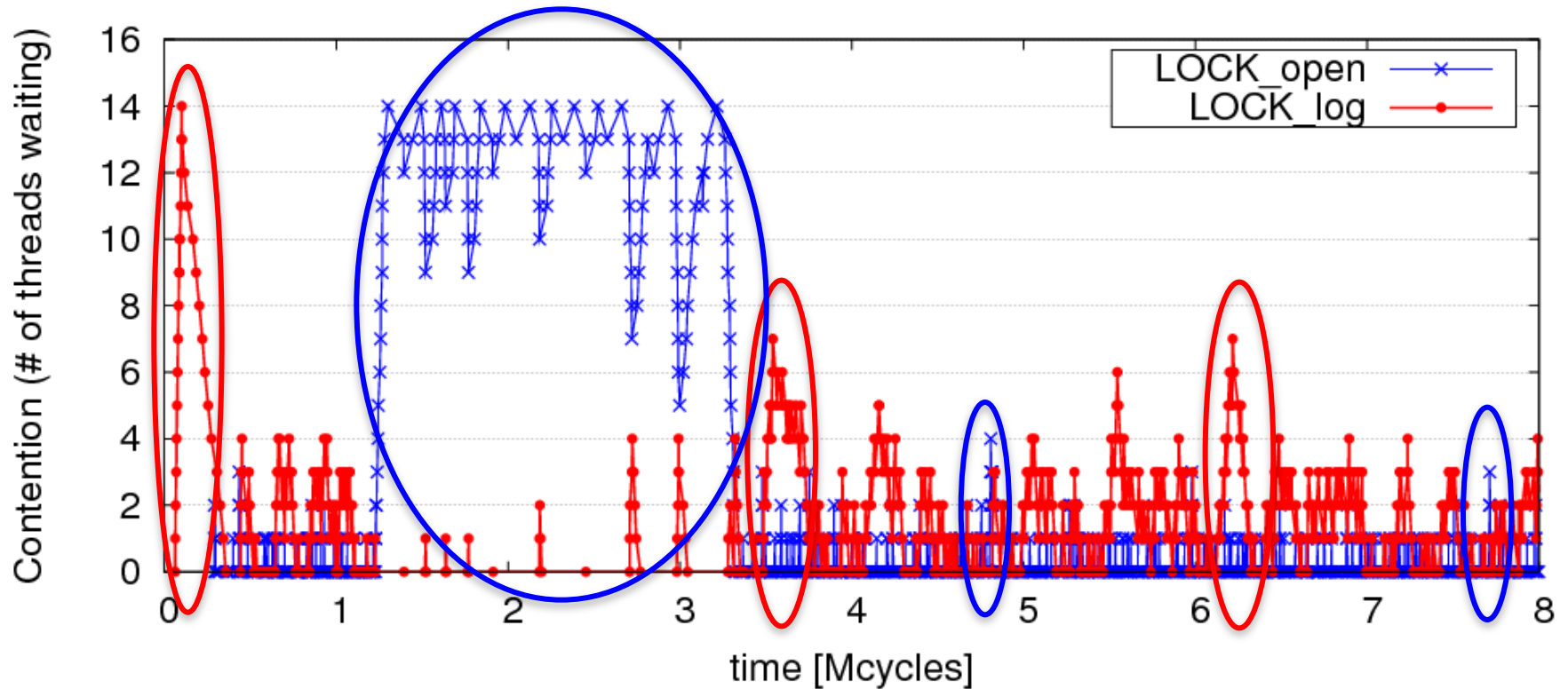
Unlock B

until A is empty



# Limiting Bottlenecks Do Change on Real Applications

MySQL running Sysbench queries, 16 threads



# Previous Work on Bottleneck Acceleration

---

- Asymmetric CMP (ACMP) proposals [Annavaram+, ISCA'05] [Morad+, Comp. Arch. Letters'06] [Suleman+, Tech. Report'07]
- Accelerated Critical Sections (ACS) [Suleman+, ASPLOS'09, Top Picks'10]
- Feedback-Directed Pipelining (FDP) [Suleman+, PACT'10 and PhD thesis'11]

No previous work

- can accelerate all types of bottlenecks or
- adapts to fine-grain changes in the *importance* of bottlenecks

*Our goal:*

*general mechanism to identify and accelerate performance-limiting bottlenecks of any type*

# 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

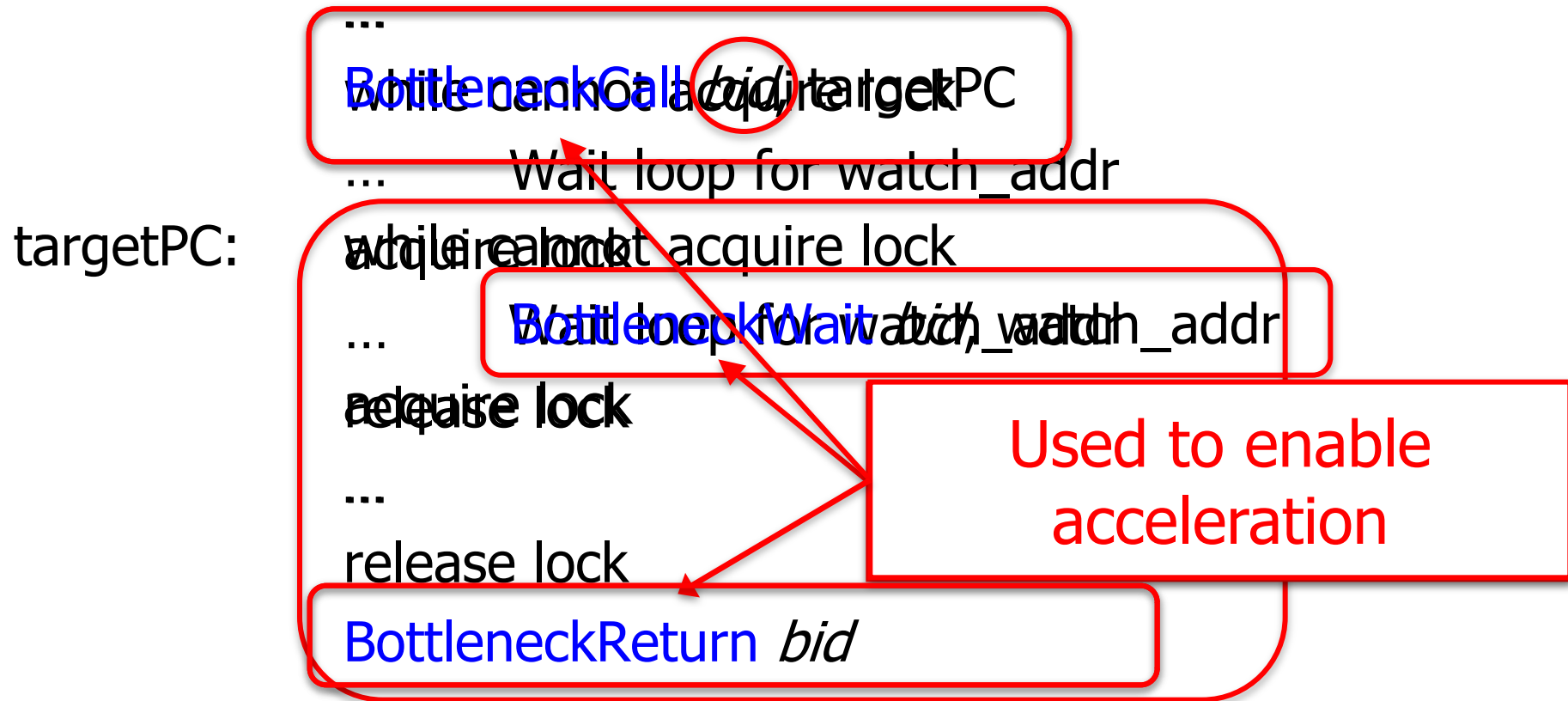
1. Annotate *bottleneck* code
2. Implement *waiting* for bottlenecks

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

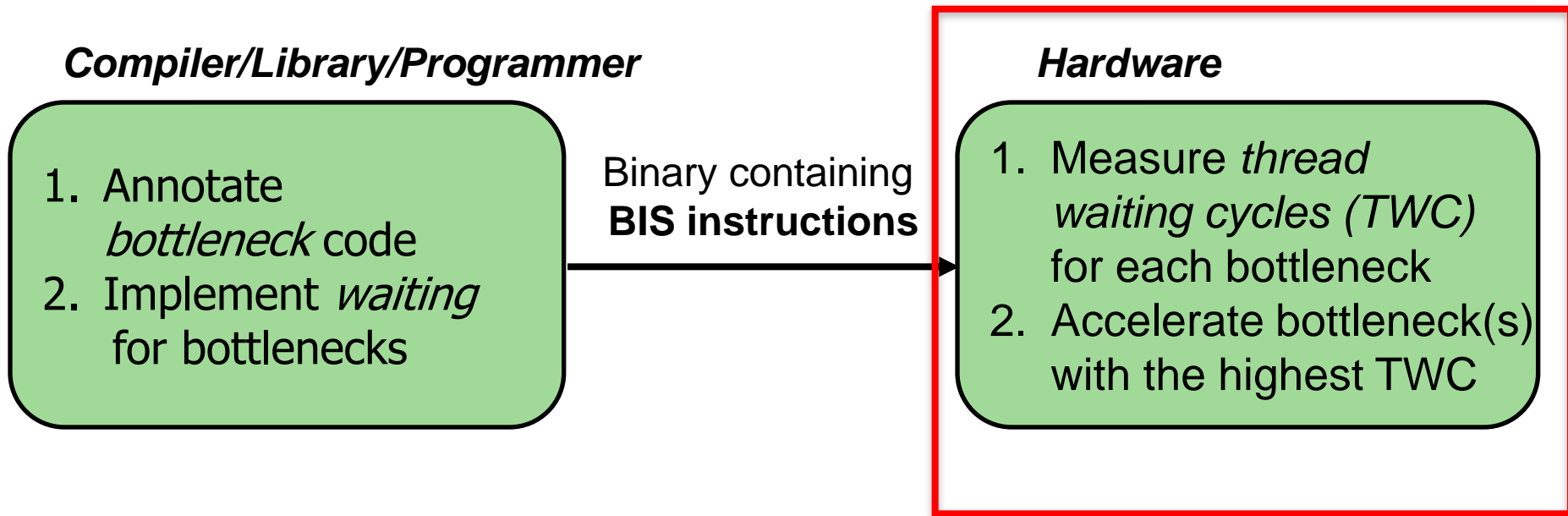
enqueue next work

**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

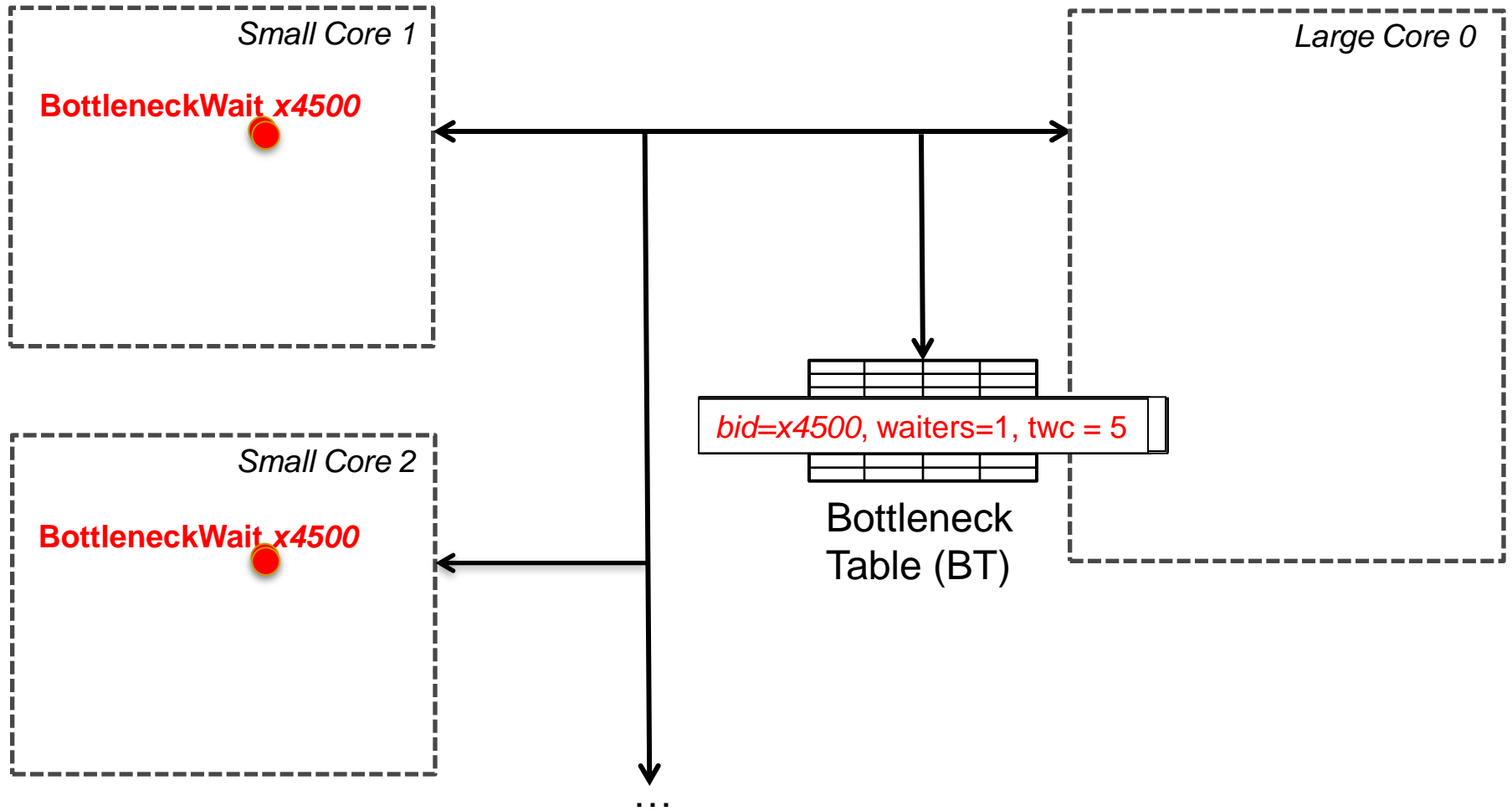
1. Annotate *bottleneck* code
2. Implement *waiting* for bottlenecks

Binary containing  
**BIS instructions**

## Hardware

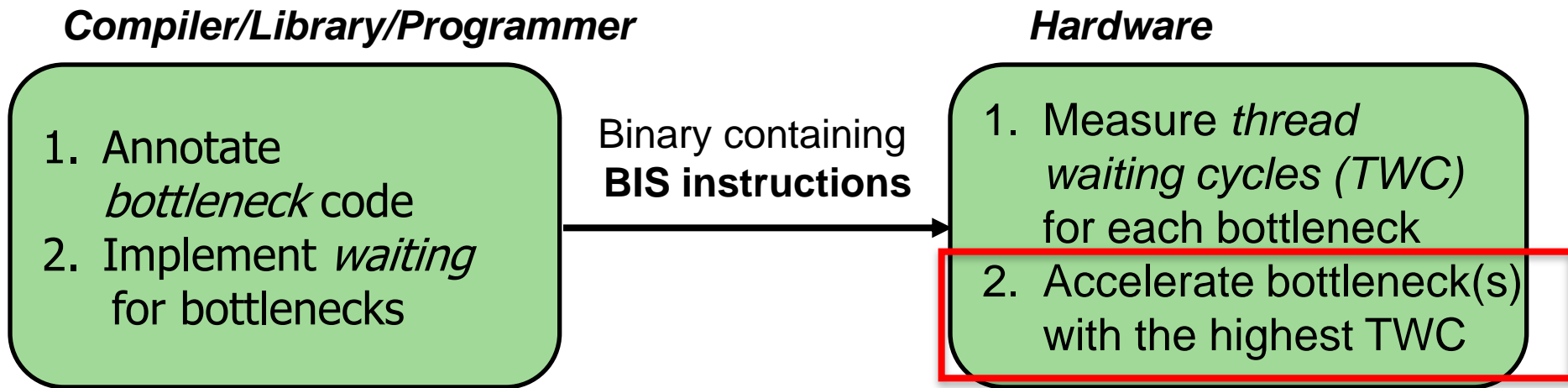
1. Measure *thread waiting cycles (TWC)* for each bottleneck
2. Accelerate bottleneck(s) with the highest TWC

# Determining Thread Waiting Cycles for Each Bottleneck

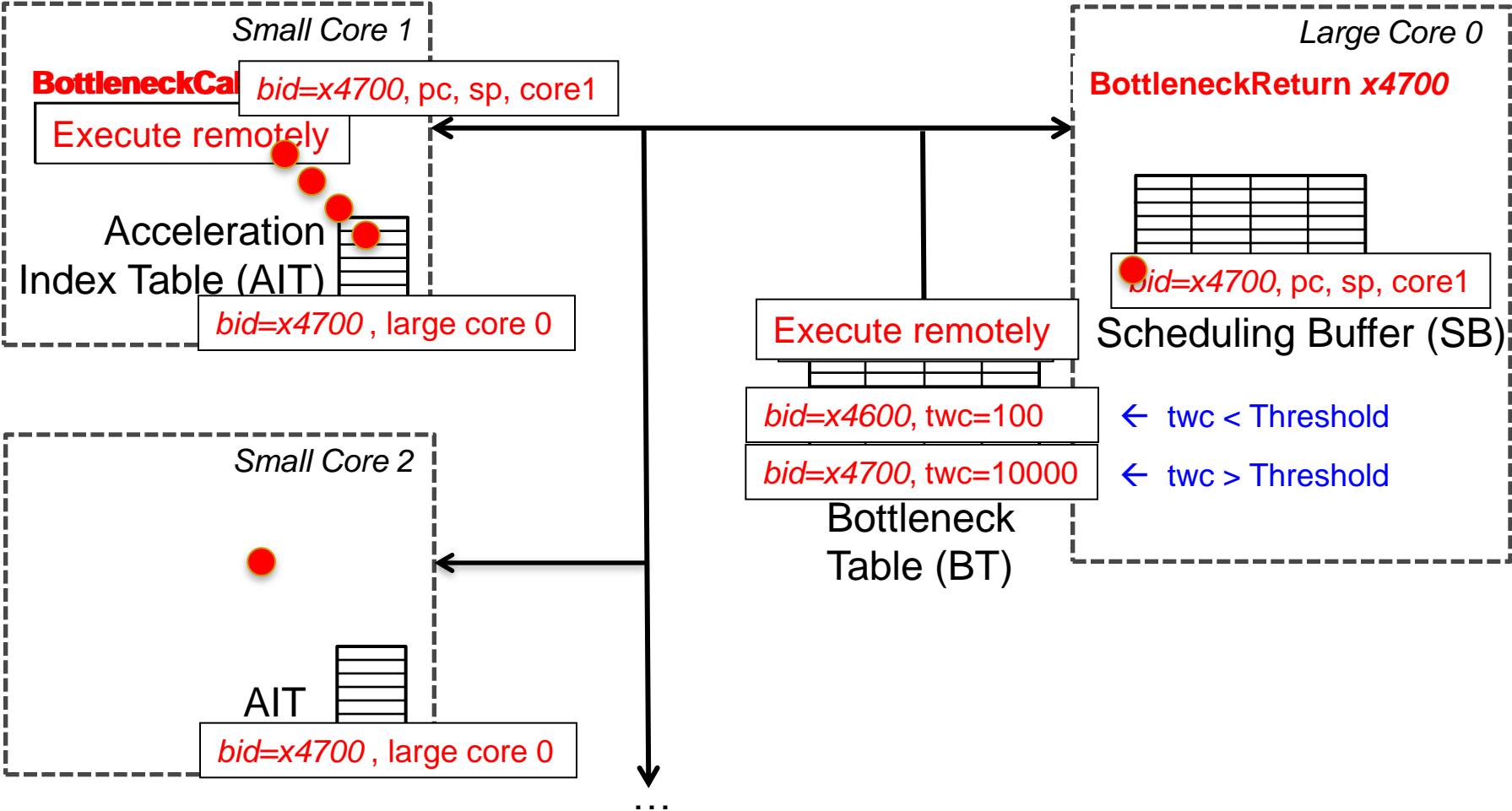


# Bottleneck Identification and Scheduling (BIS)

---



# 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

# 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

---

- **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 not on critical path)

# Evaluation 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

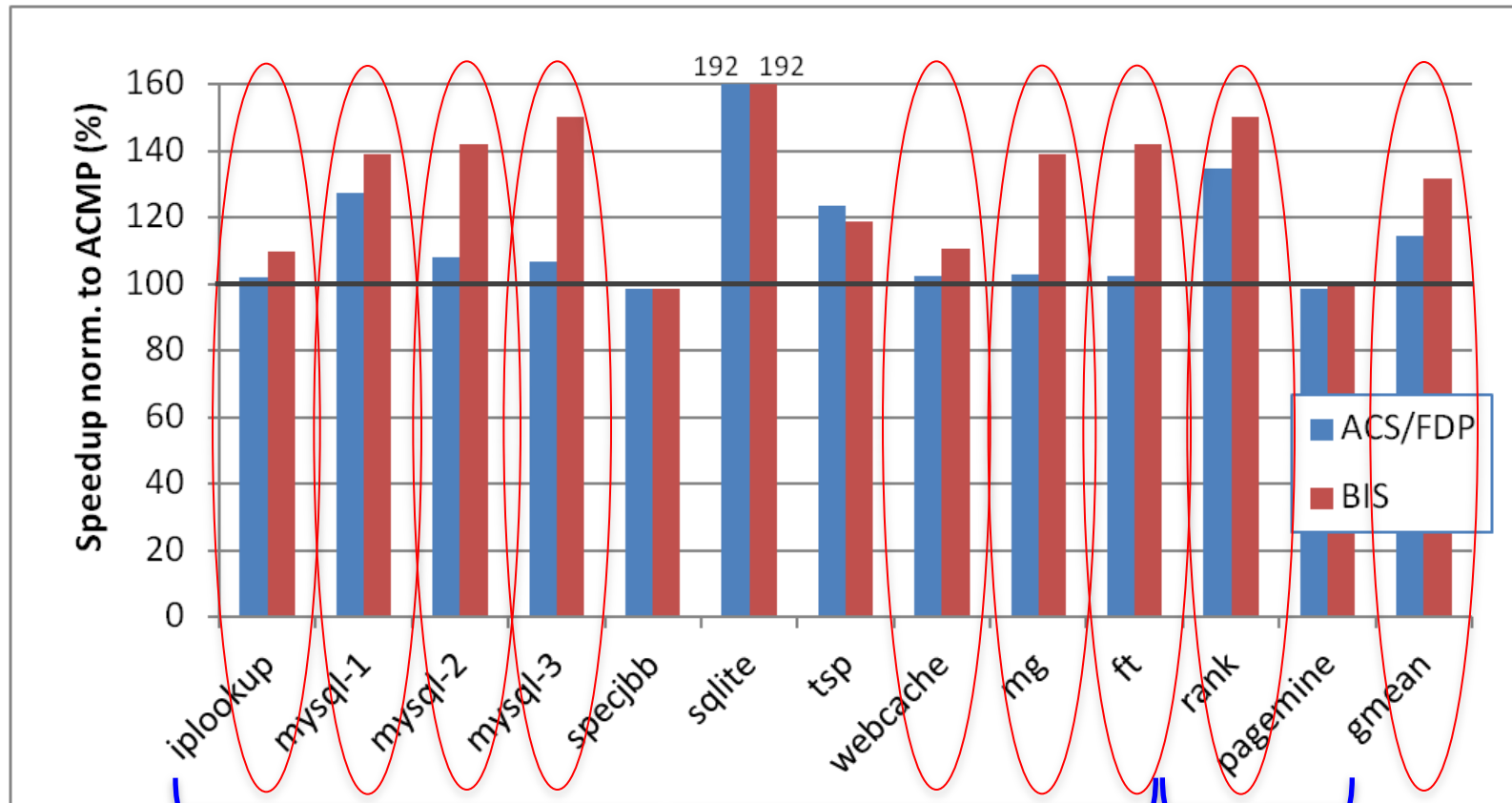
# BIS Comparison Points (Area-Equivalent)

---

- SCMP (Symmetric CMP)
  - All small cores
- 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

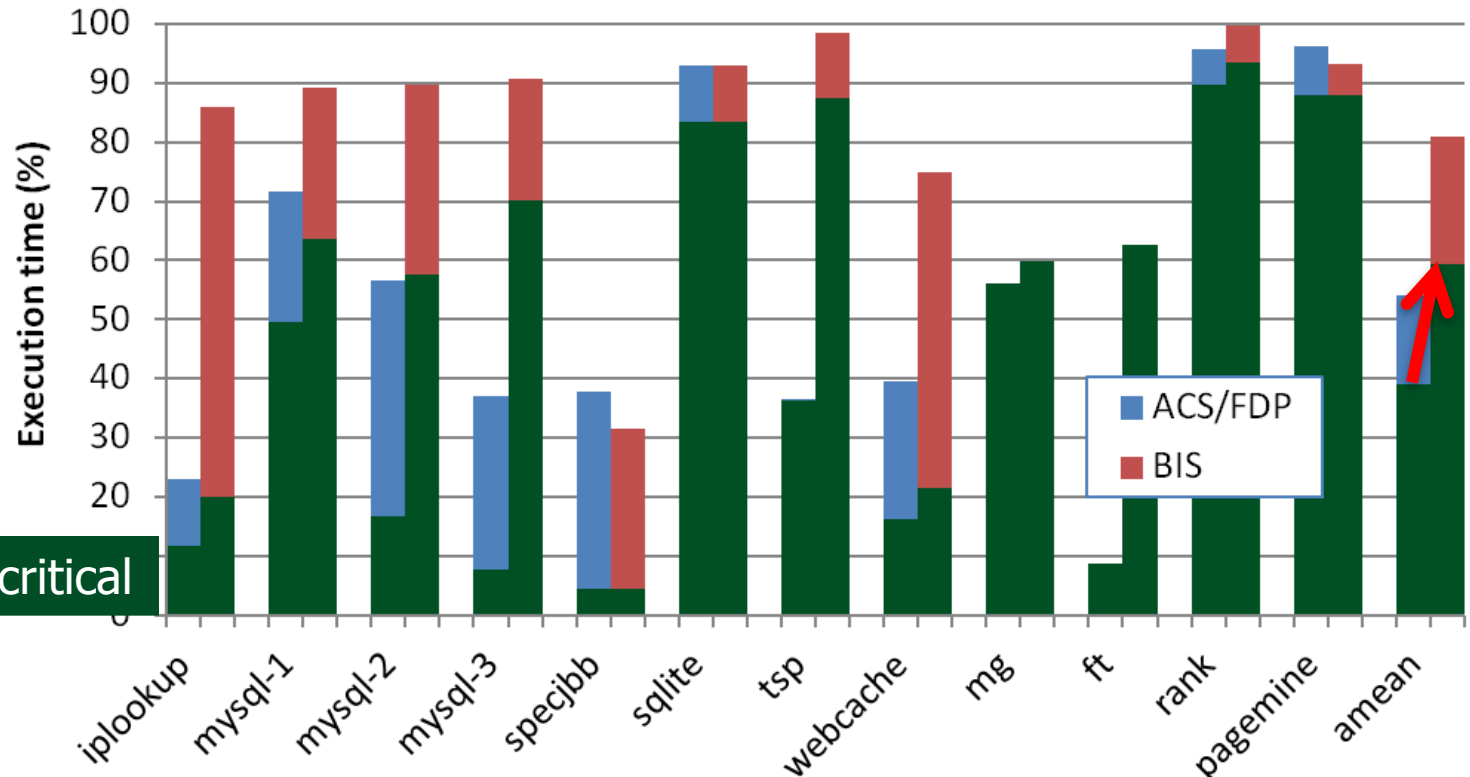
Optimal number of threads, 28 small cores, 1 large core



- BIS outperforms ACS/FDP by 15% and ACMP by 32%  
limiting bottlenecks change over barriers, which ACS cannot accelerate
- BIS improves scalability on 4 of the benchmarks

# Why Does BIS Work?

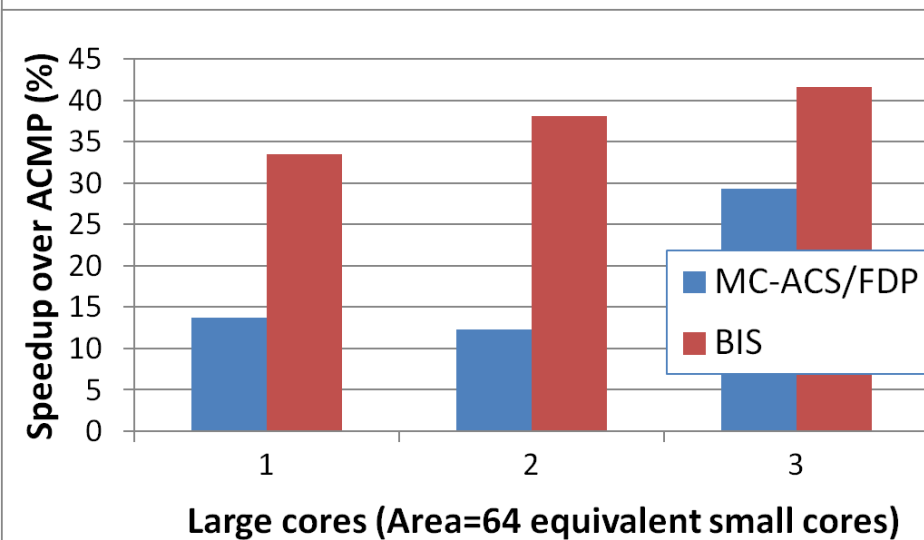
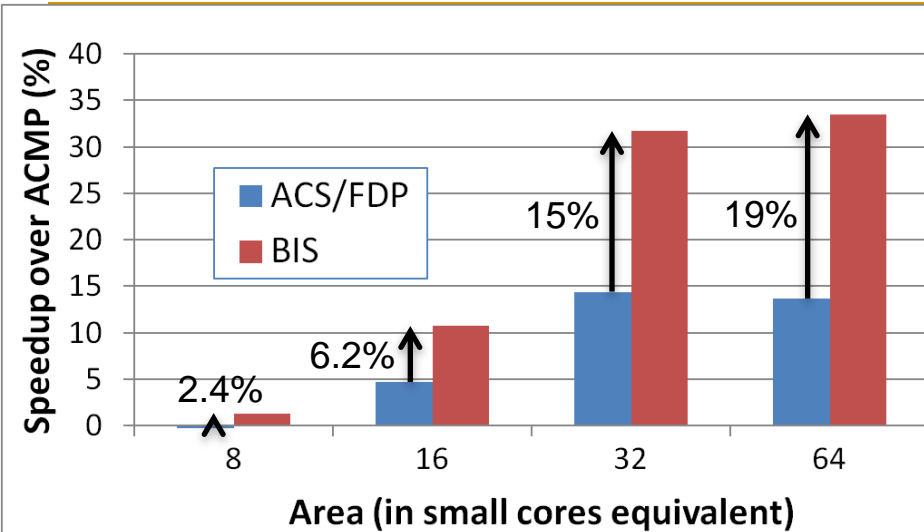
Fraction of execution time spent on predicted-important bottlenecks



Actually critical

- 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:
  - Performance benefits increase with more cores
- Provides **comprehensive fine-grained bottleneck acceleration** with no programmer effort

We did not cover the remaining slides.  
These are for your benefit.



# Handling Private Data Locality: Data Marshaling

M. Aater Suleman, Onur Mutlu, Jose A. Joao, Khubaib, and Yale N. Patt,  
**"Data Marshaling for Multi-core Architectures"**  
*Proceedings of the 37th International Symposium on Computer Architecture (ISCA)*,  
pages 441-450, Saint-Malo, France, June 2010.

# 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)

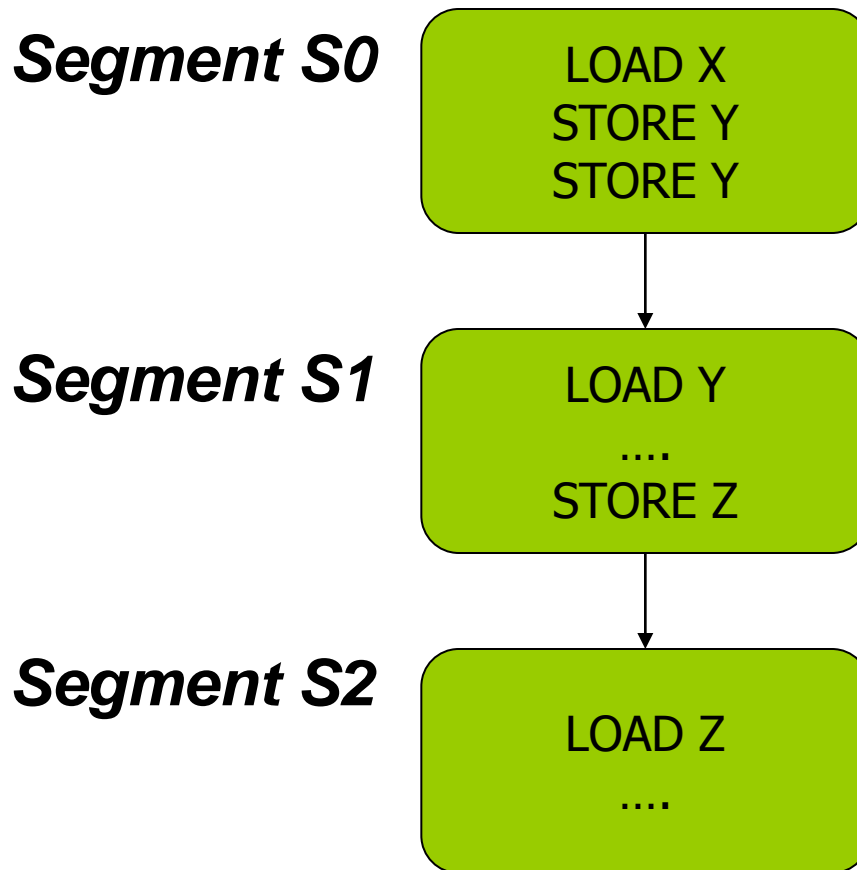
---



# Staged Execution Model (III)

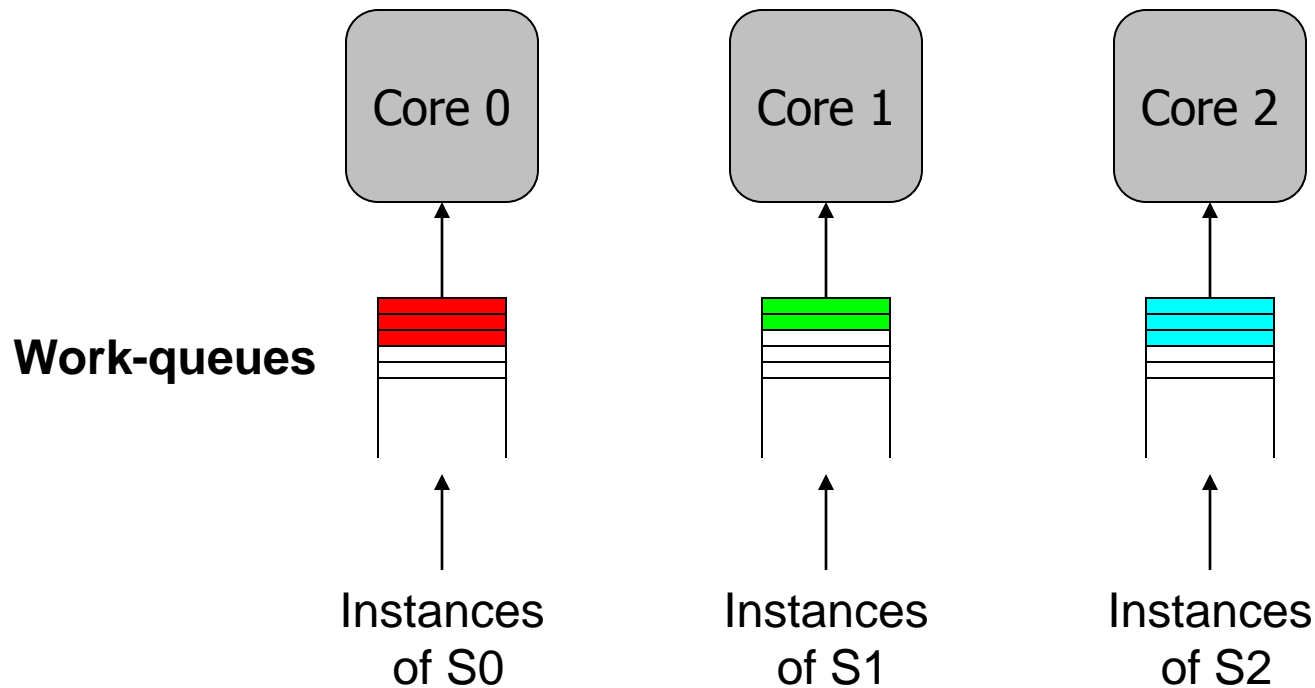
---

*Split code into segments*



# Staged Execution Model (IV)

---



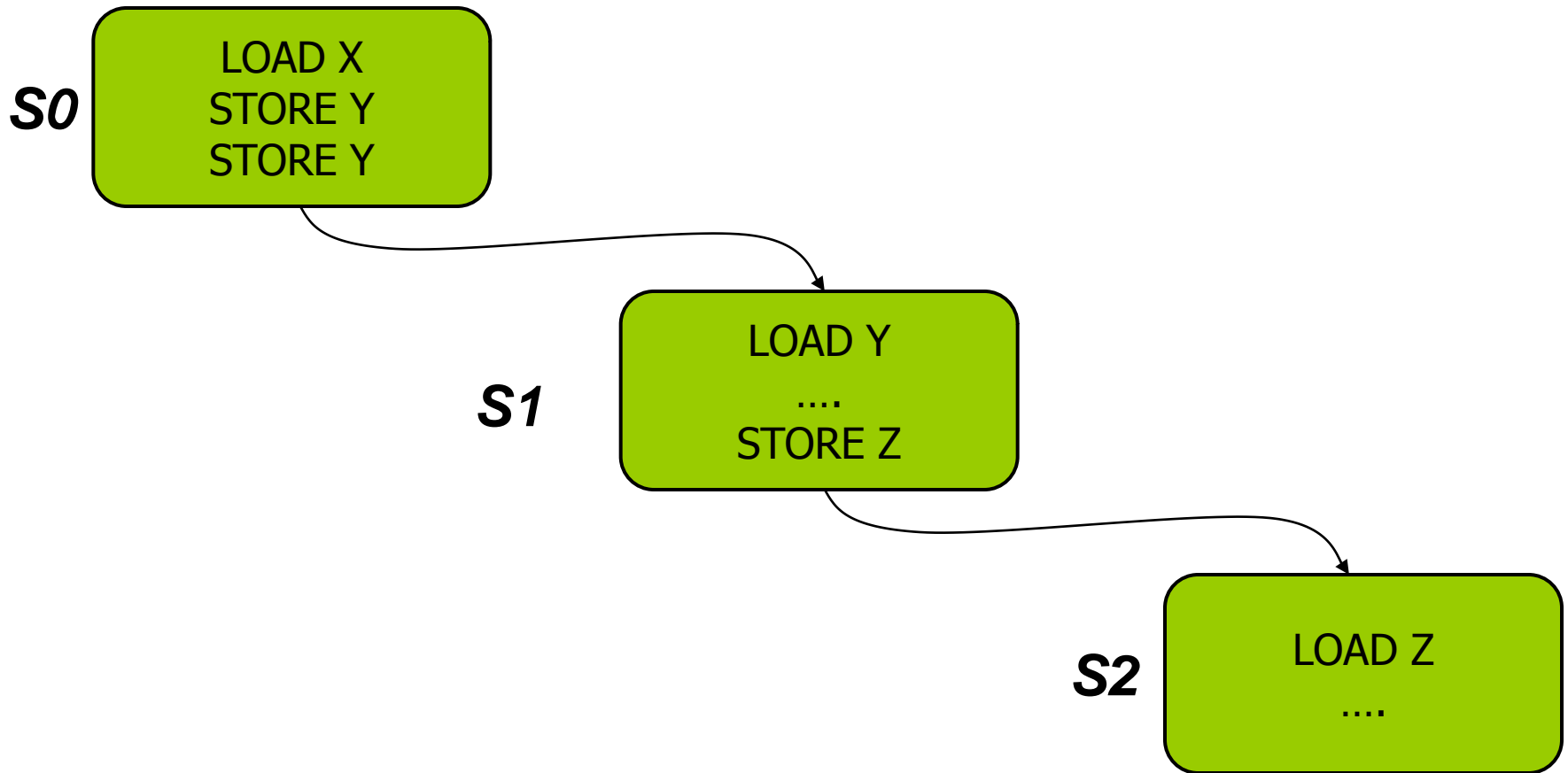
# Staged Execution Model: Segment Spawning

---

**Core 0**

**Core 1**

**Core 2**



# 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

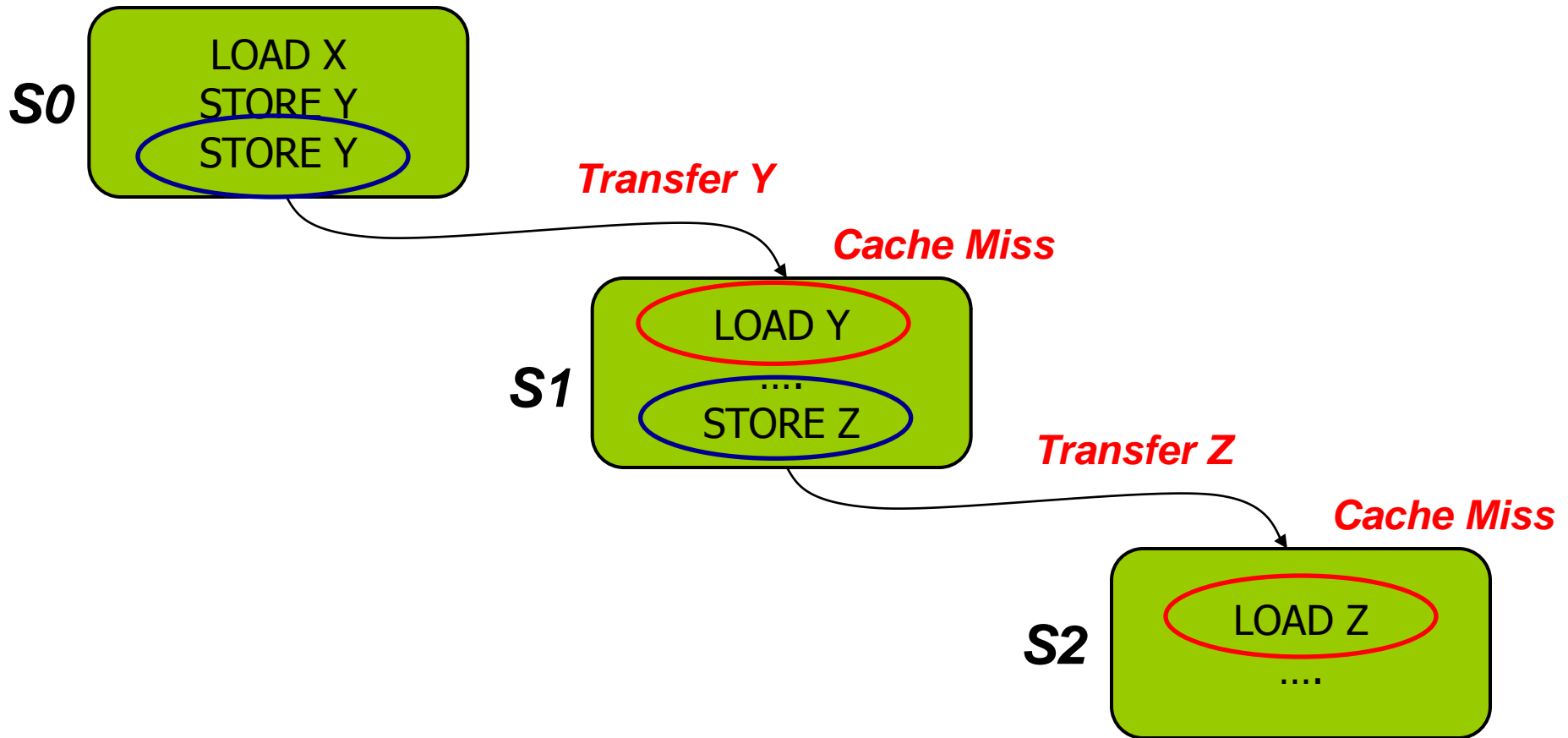
# Problem: Locality of Inter-segment Data

---

**Core 0**

**Core 1**

**Core 2**





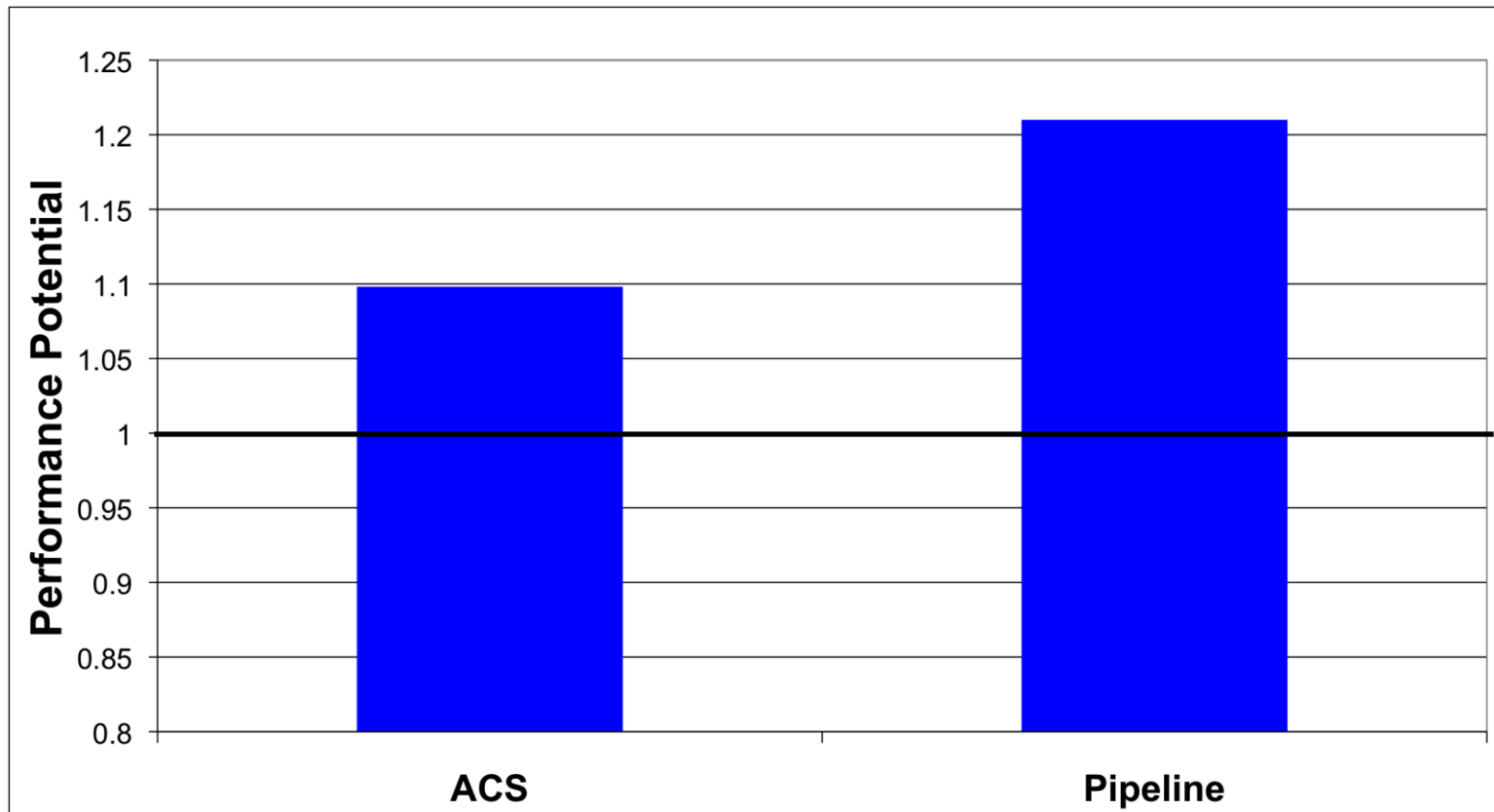
# Problem: Locality of Inter-segment Data

---

- Accelerated Critical Sections [Suleman et al., ASPLOS 2010]
  - 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?

---



# Talk Outline

---

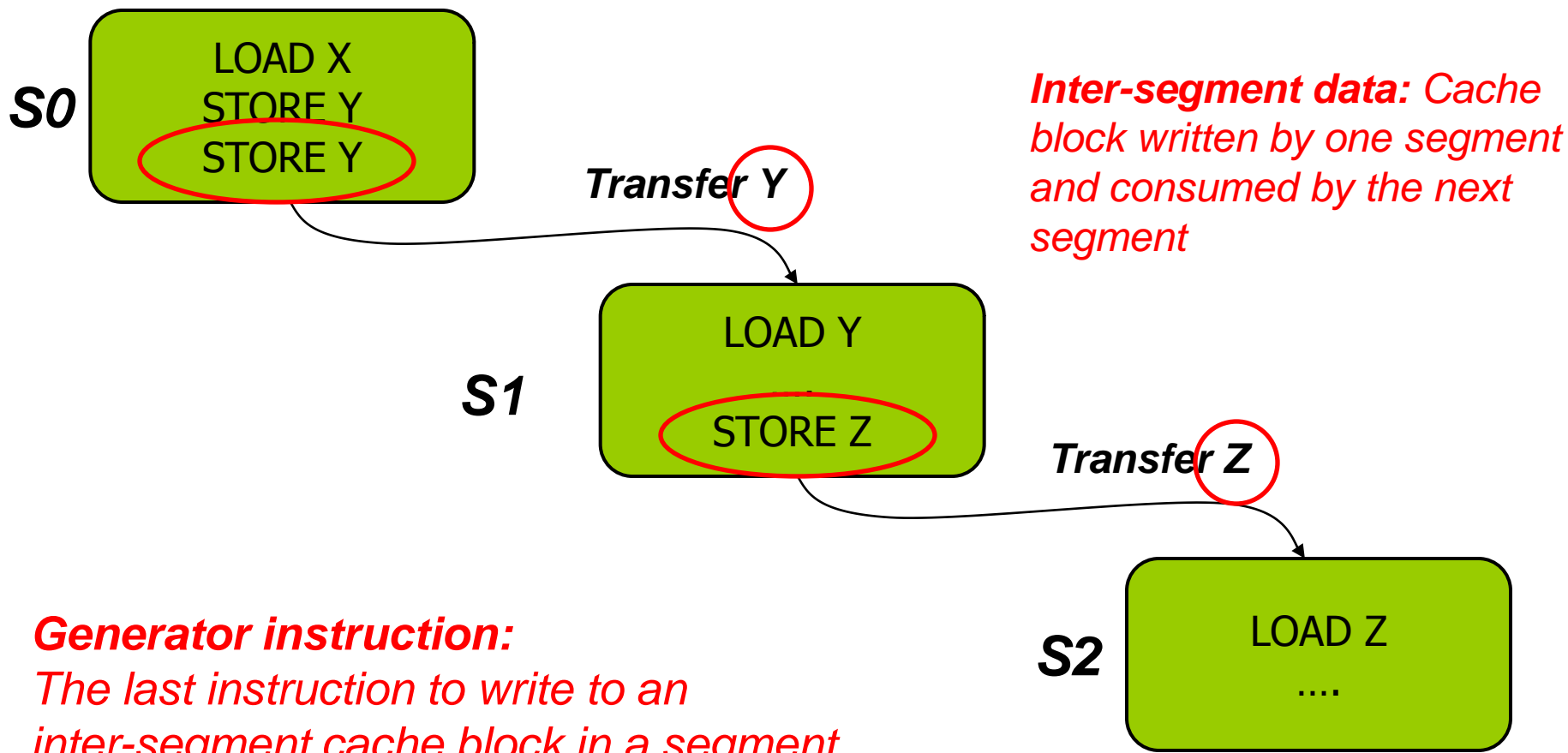
- Problem and Motivation
- How Do We Get There: Examples
- Accelerated Critical Sections (ACS)
- Bottleneck Identification and Scheduling (BIS)
- Staged Execution and [Data Marshaling](#)
- Thread Cluster Memory Scheduling (if time permits)
- Ongoing/Future Work
- Conclusions

# Terminology

Core 0

Core 1

Core 2



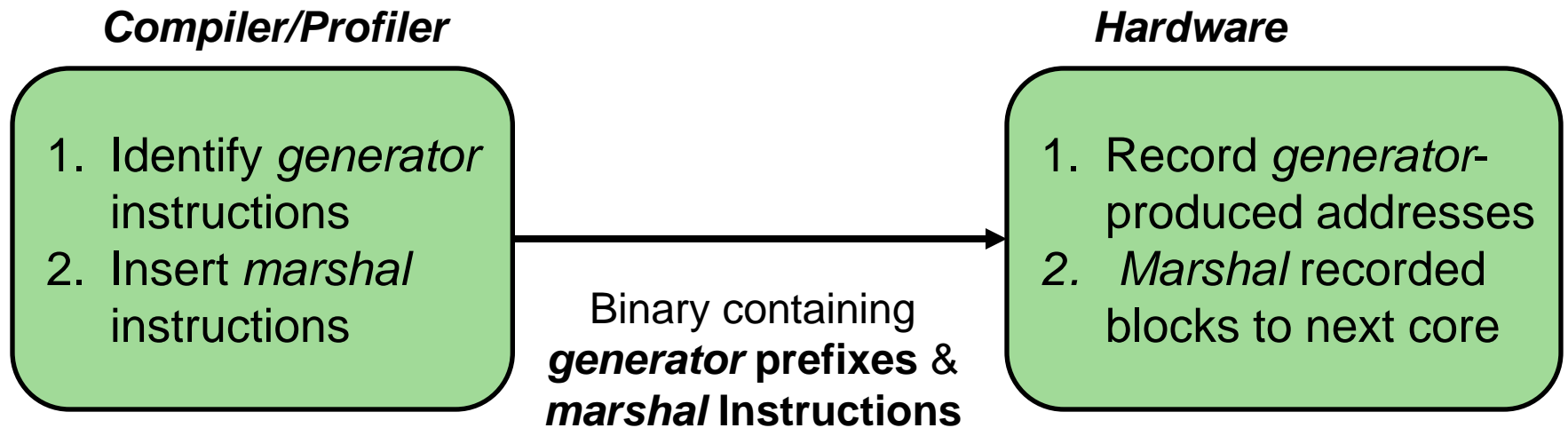
# Key Observation and Idea

---

- 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.

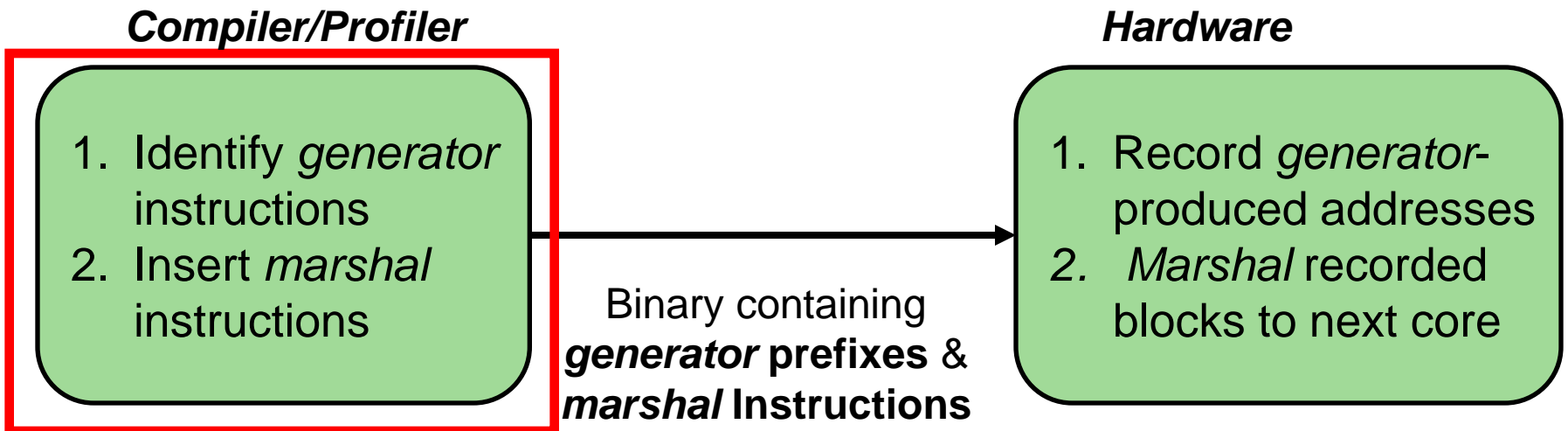
# Data Marshaling

---



# Data Marshaling

---

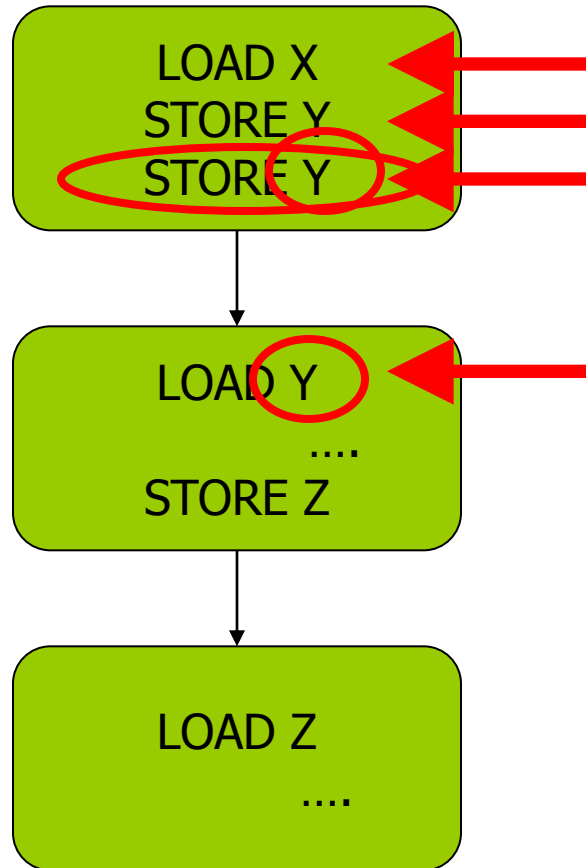


# Profiling Algorithm

---

*Inter-segment data*

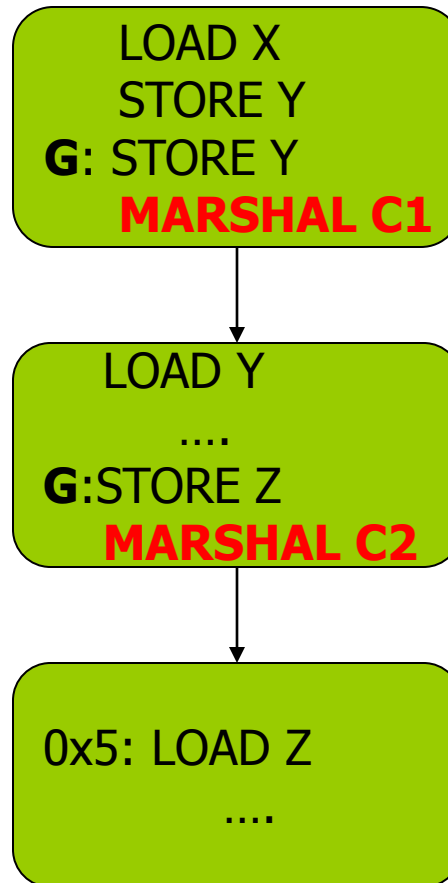
*Mark as Generator  
Instruction*





# 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

---

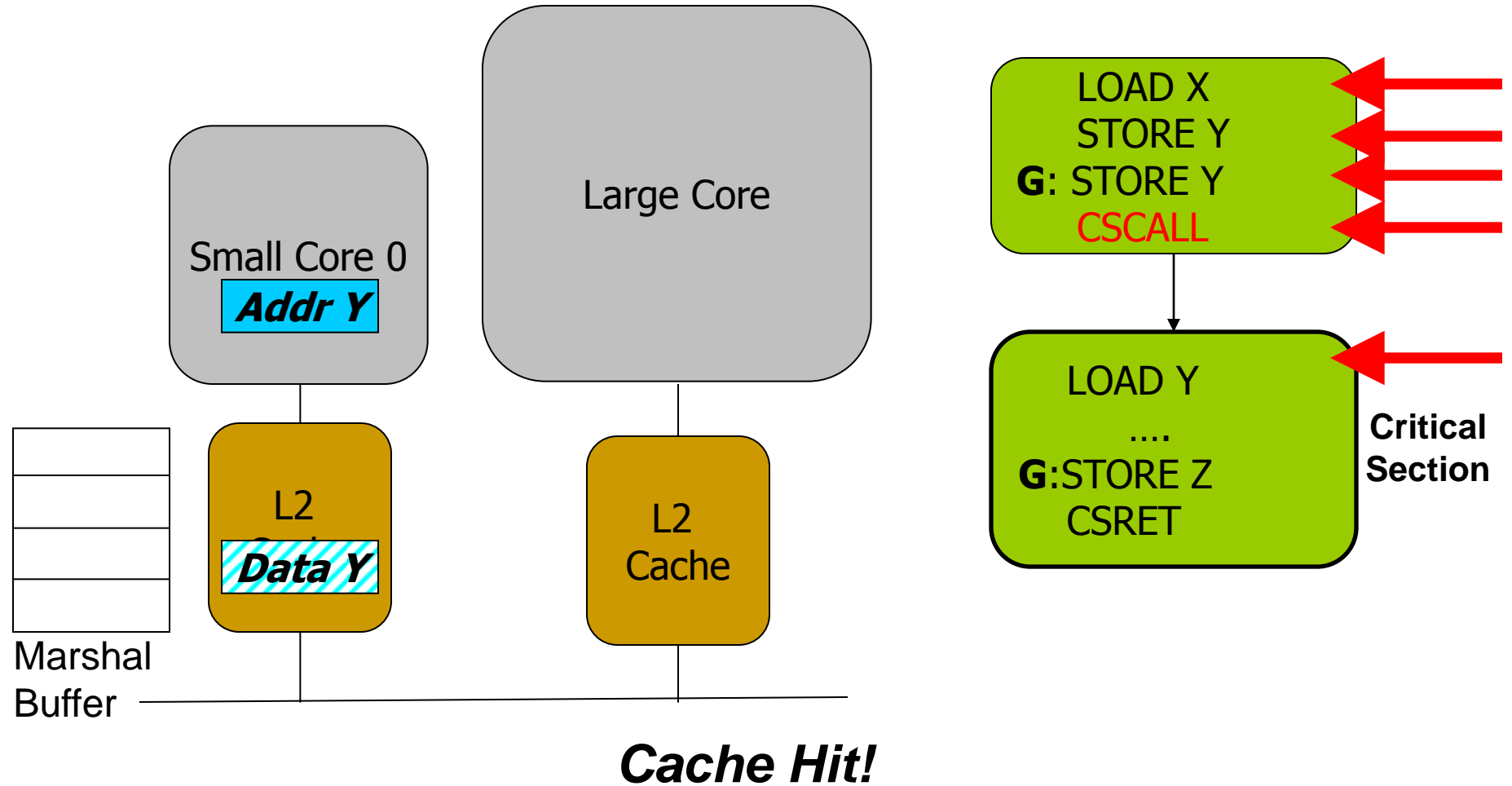
## ■ Advantages

- **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

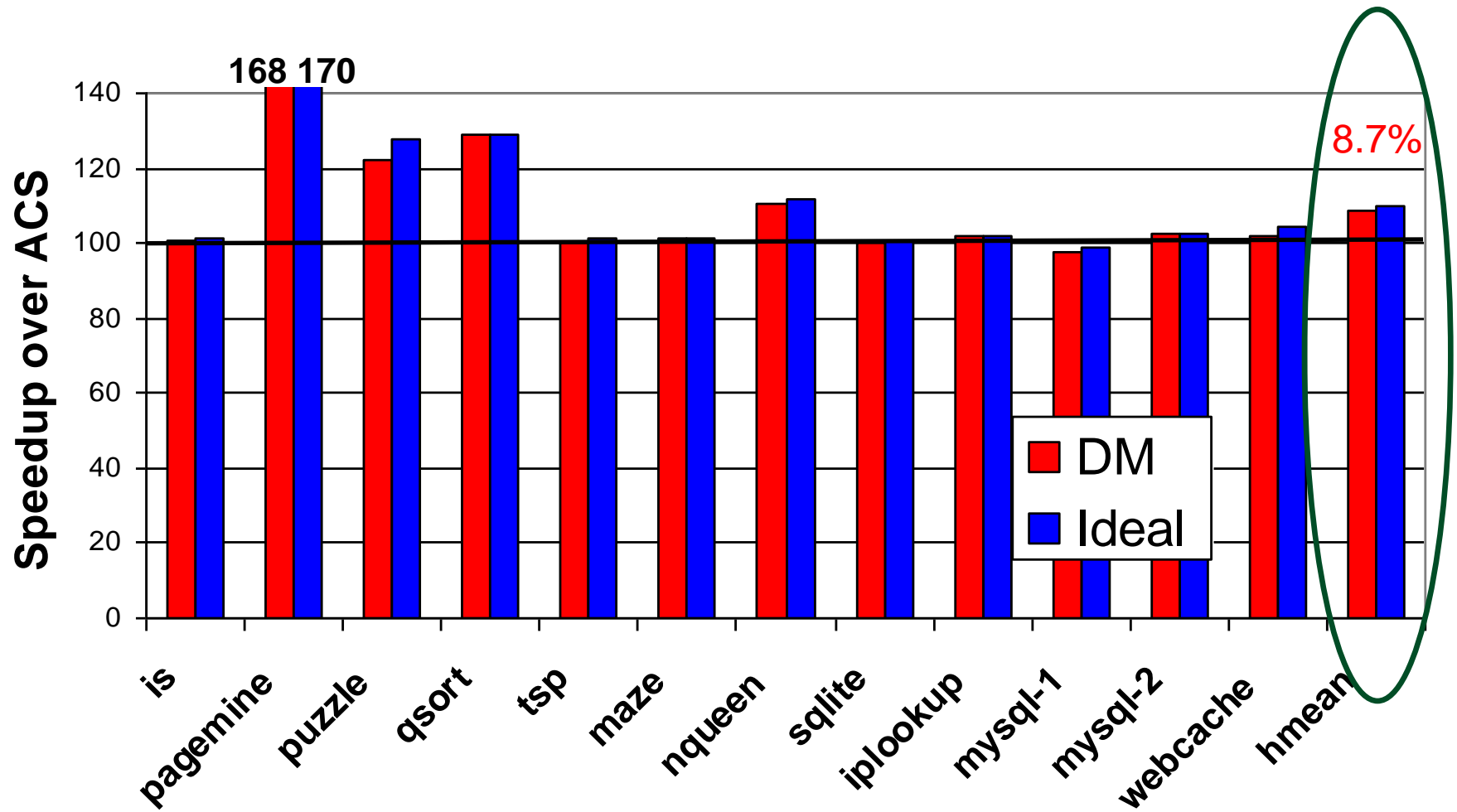


# Accelerated Critical Sections: Methodology

---

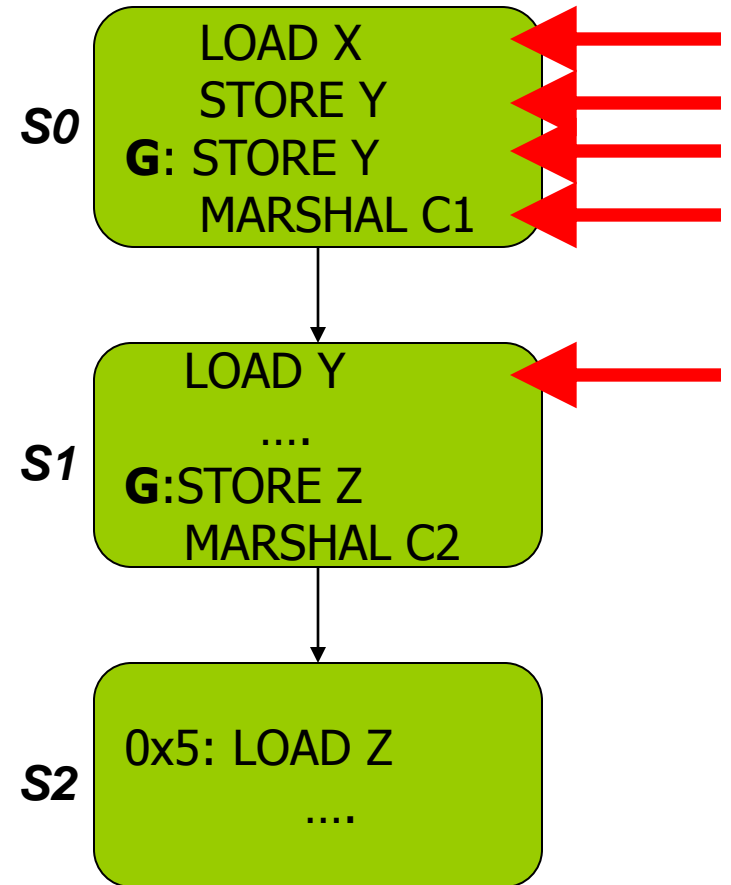
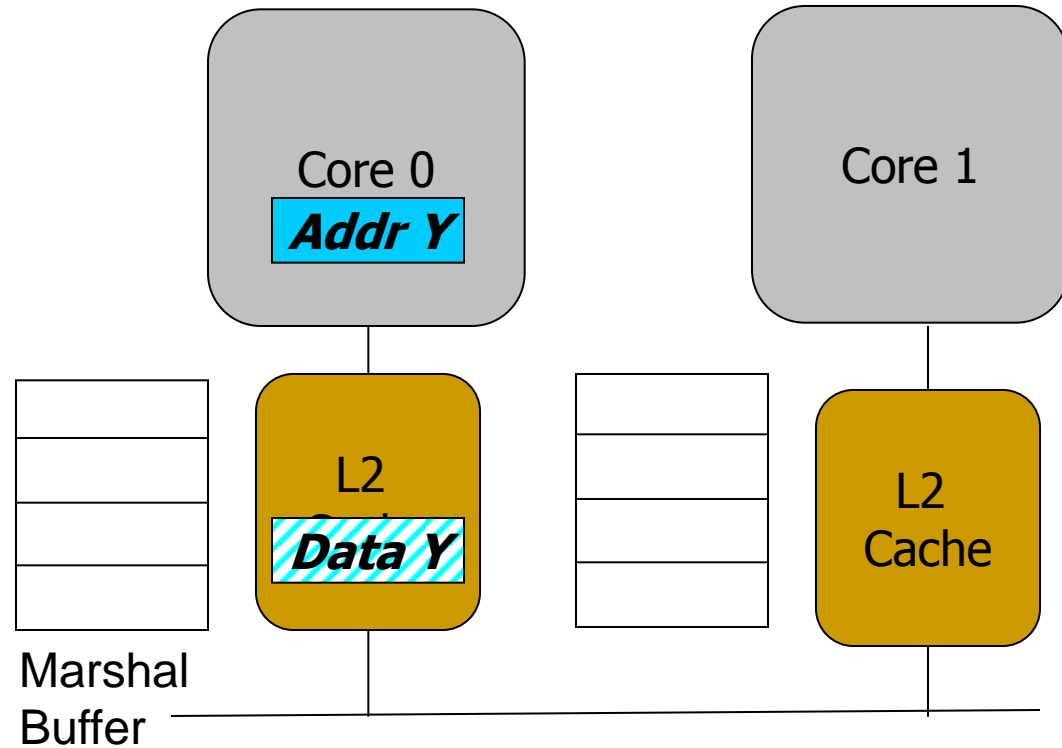
- Workloads: 12 critical section intensive applications
  - Data mining kernels, sorting, database, web, networking
  - Different training and simulation input sets
- Multi-core x86 simulator
  - 1 large and 28 small cores
  - Aggressive stream prefetcher employed at each core
- Details:
  - Large core: 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

**Cache Hit!**



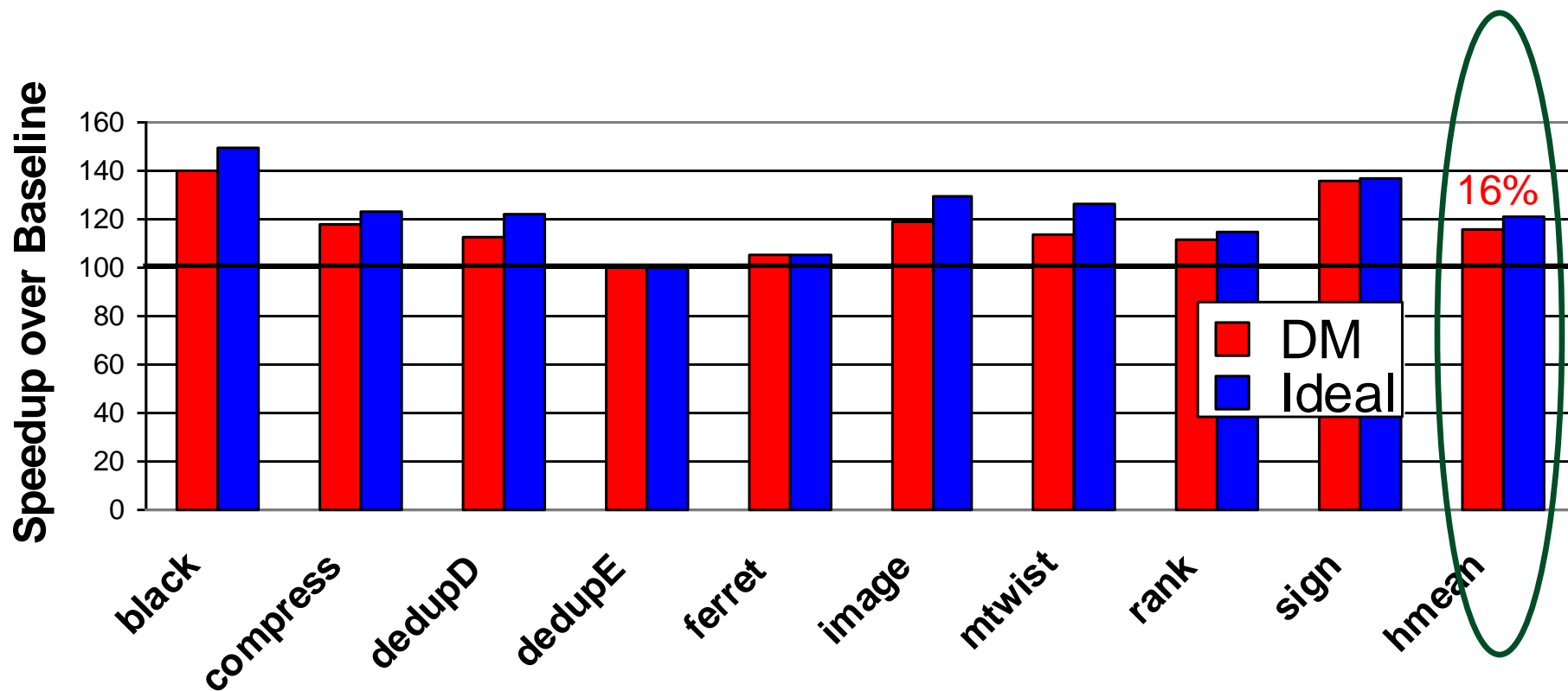
# Pipeline Parallelism: Methodology

---

- 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

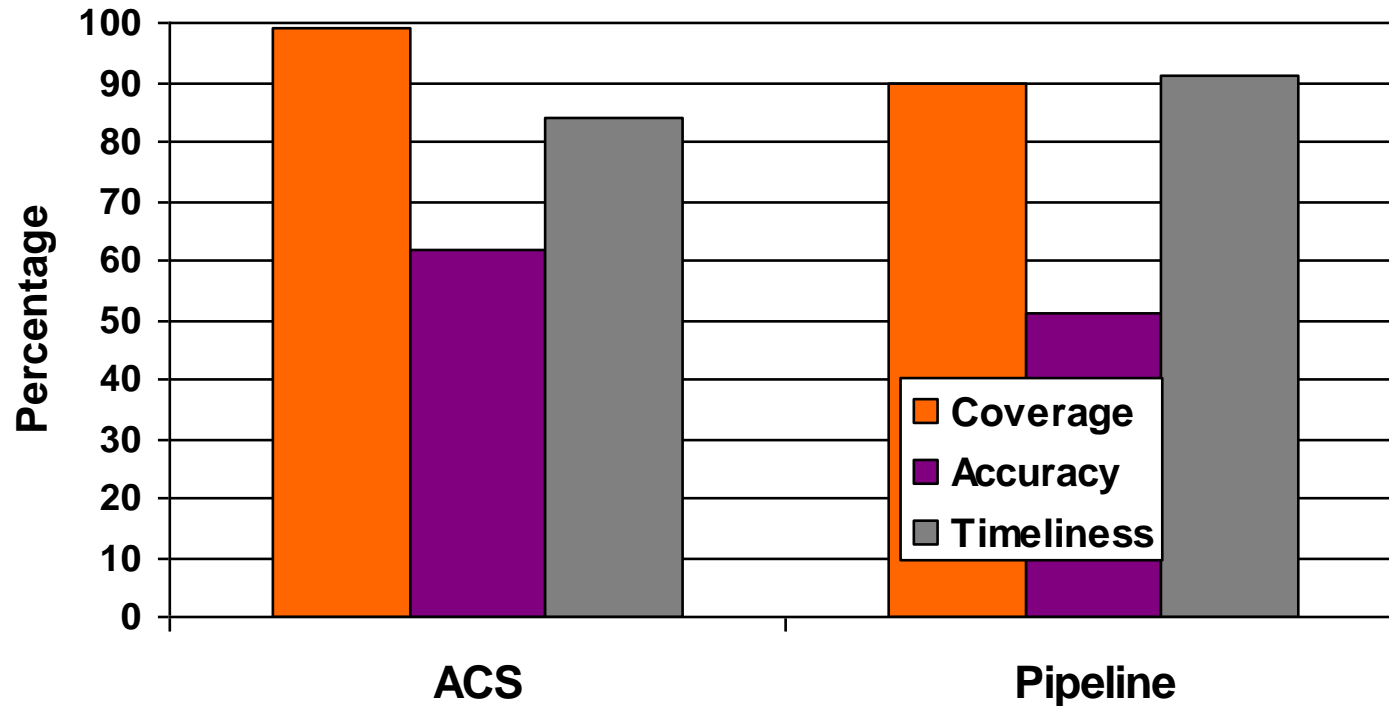


# DM on Pipeline Parallelism: Results



# 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**

# A Case for Asymmetry Everywhere

Onur Mutlu,

**"Asymmetry Everywhere (with Automatic Resource Management)"**

*CRA Workshop on Advancing Computer Architecture Research: Popular  
Parallel Programming*, San Diego, CA, February 2010.

Position paper

# The Setting

---

- Hardware resources are shared among many threads/apps in a many-core based system
  - Cores, caches, interconnects, memory, disks, power, lifetime, ...
- Management of these resources is a very difficult task
  - When optimizing parallel/multiprogrammed workloads
  - Threads interact unpredictably/unfairly in shared resources
- **Power/energy** is arguably the most valuable shared resource
  - Main limiter to efficiency and performance

# Shield the Programmer from Shared Resources

---

- Writing even sequential software is hard enough
  - Optimizing code for a complex shared-resource parallel system will be a nightmare for most programmers
- Programmer should not worry about (hardware) resource management
  - What should be executed where with what resources
- Future cloud computer architectures should be designed to
  - Minimize programmer effort to optimize (parallel) programs
  - Maximize runtime system's effectiveness in automatic shared resource management



# Shared Resource Management: Goals

---

- Future many-core systems should manage power and performance automatically across threads/applications
- Minimize energy/power consumption
- While satisfying performance/SLA requirements
  - Provide predictability and Quality of Service
- Minimize programmer effort
  - In creating optimized parallel programs
- Asymmetry and configurability in system resources essential to achieve these goals

# Asymmetry Enables Customization

---

c	c	c	c
c	c	c	c
c	c	c	c
c	c	c	c

Symmetric

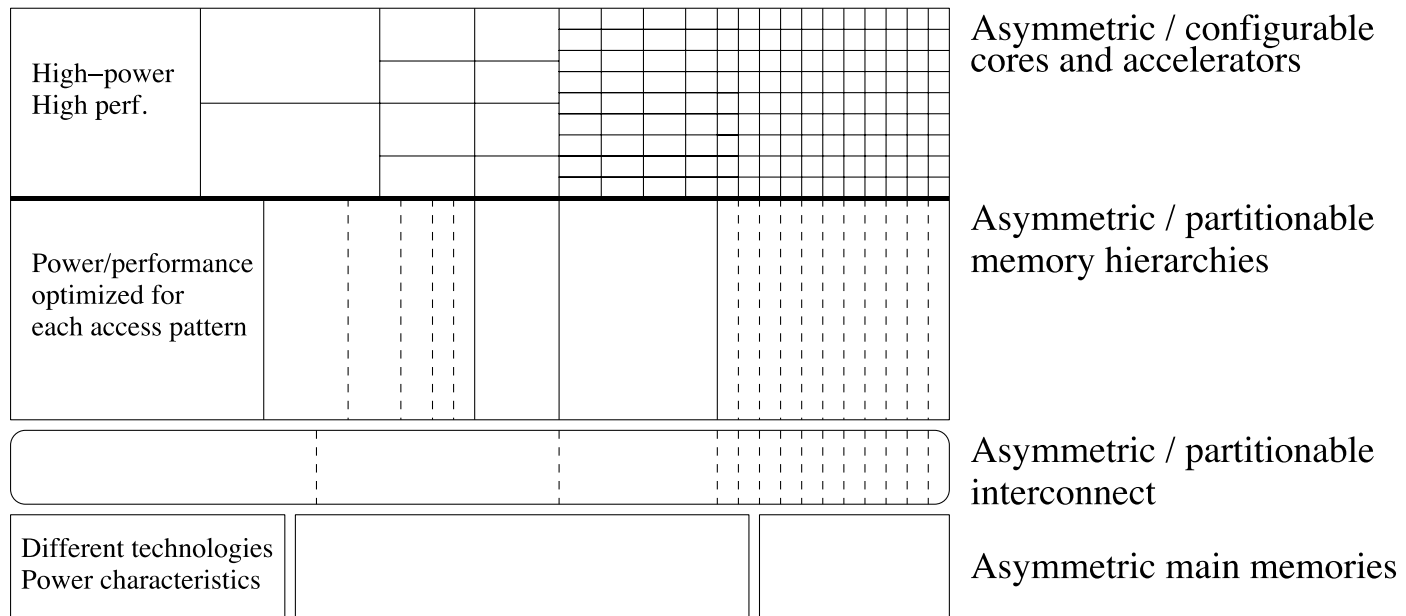
<b>C1</b>		C2	
		C3	
C4	C4	C4	C4
C5	C5	C5	C5

Asymmetric

- Symmetric: One size fits all
  - Energy and performance suboptimal for different phase behaviors
- Asymmetric: Enables tradeoffs and customization
  - Processing requirements vary across applications and phases
  - Execute code on best-fit resources (minimal energy, adequate perf.)

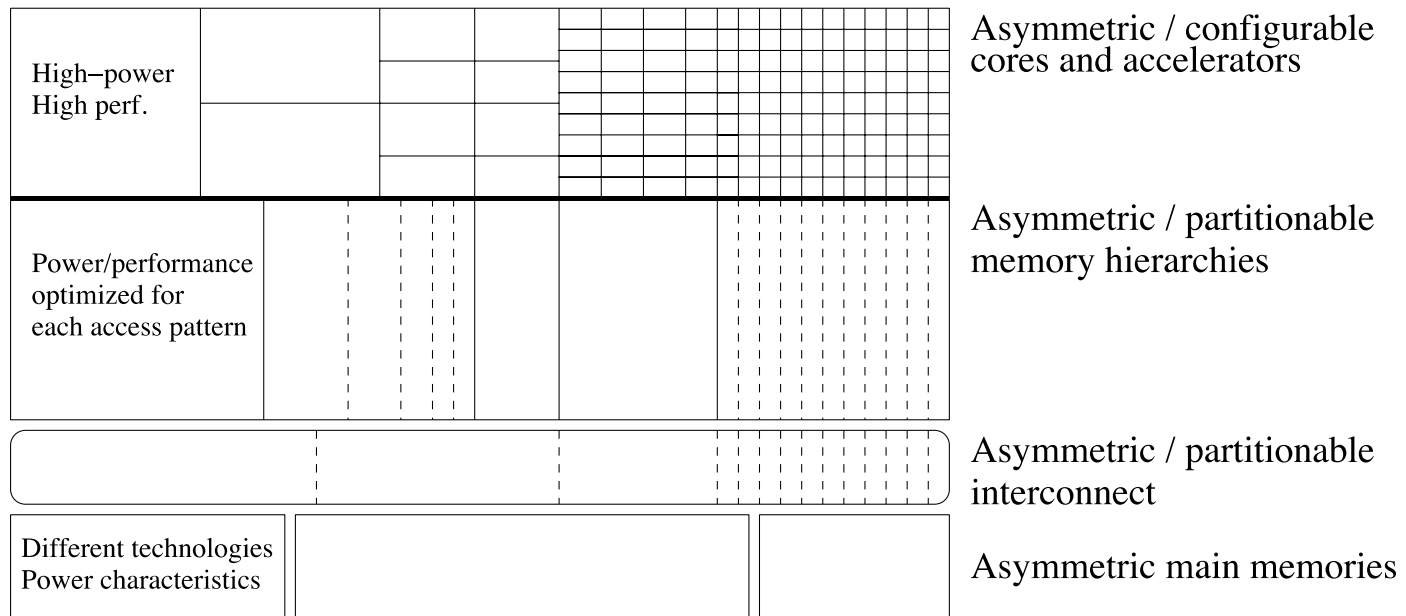
# Thought Experiment: Asymmetry Everywhere

- Design each hardware resource with **asymmetric, (re-)configurable, partitionable components**
  - Different power/performance/reliability characteristics
  - To fit different computation/access/communication patterns



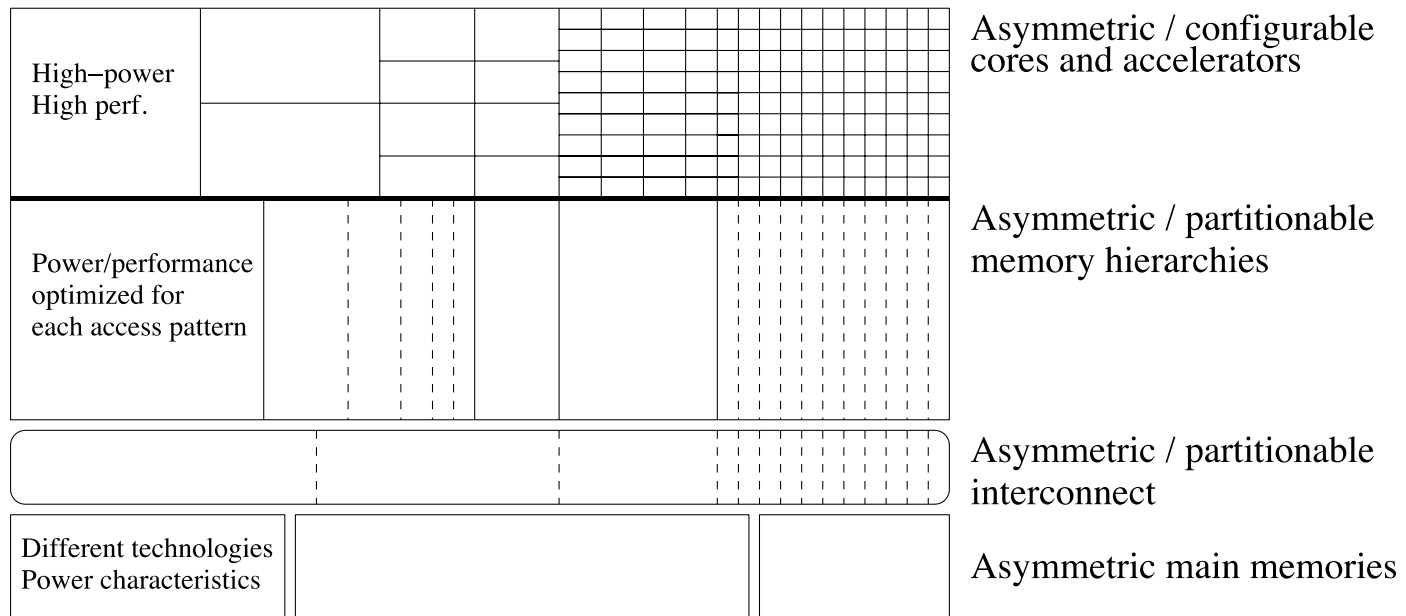
# Thought Experiment: Asymmetry Everywhere

- Design the **runtime system (HW & SW)** to **automatically choose** the best-fit components for each workload/phase
  - Satisfy performance/SLA with minimal energy
  - **Dynamically stitch together the “best-fit” chip for each phase**



# Thought Experiment: Asymmetry Everywhere

- **Morph software components** to match asymmetric HW components
  - Multiple versions for different resource characteristics

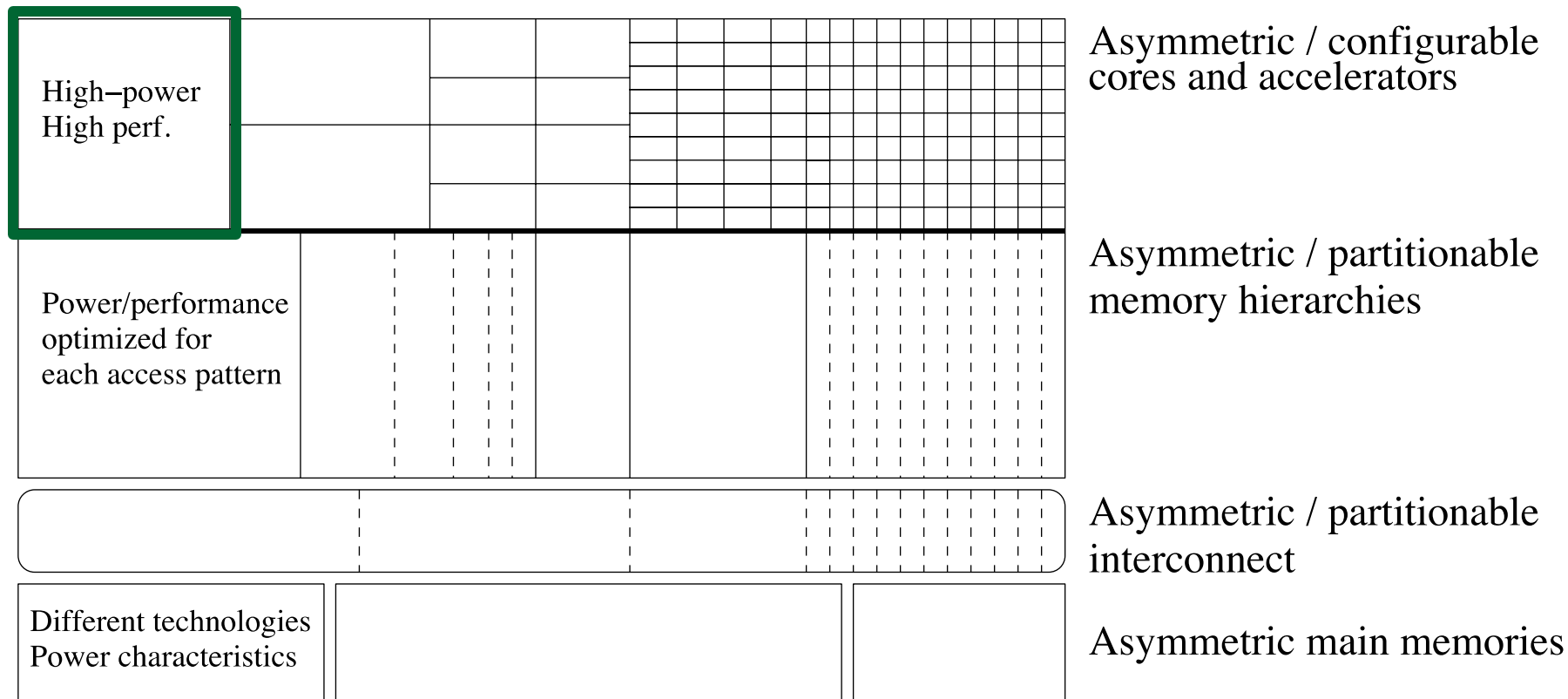


# Many Research and Design Questions

---

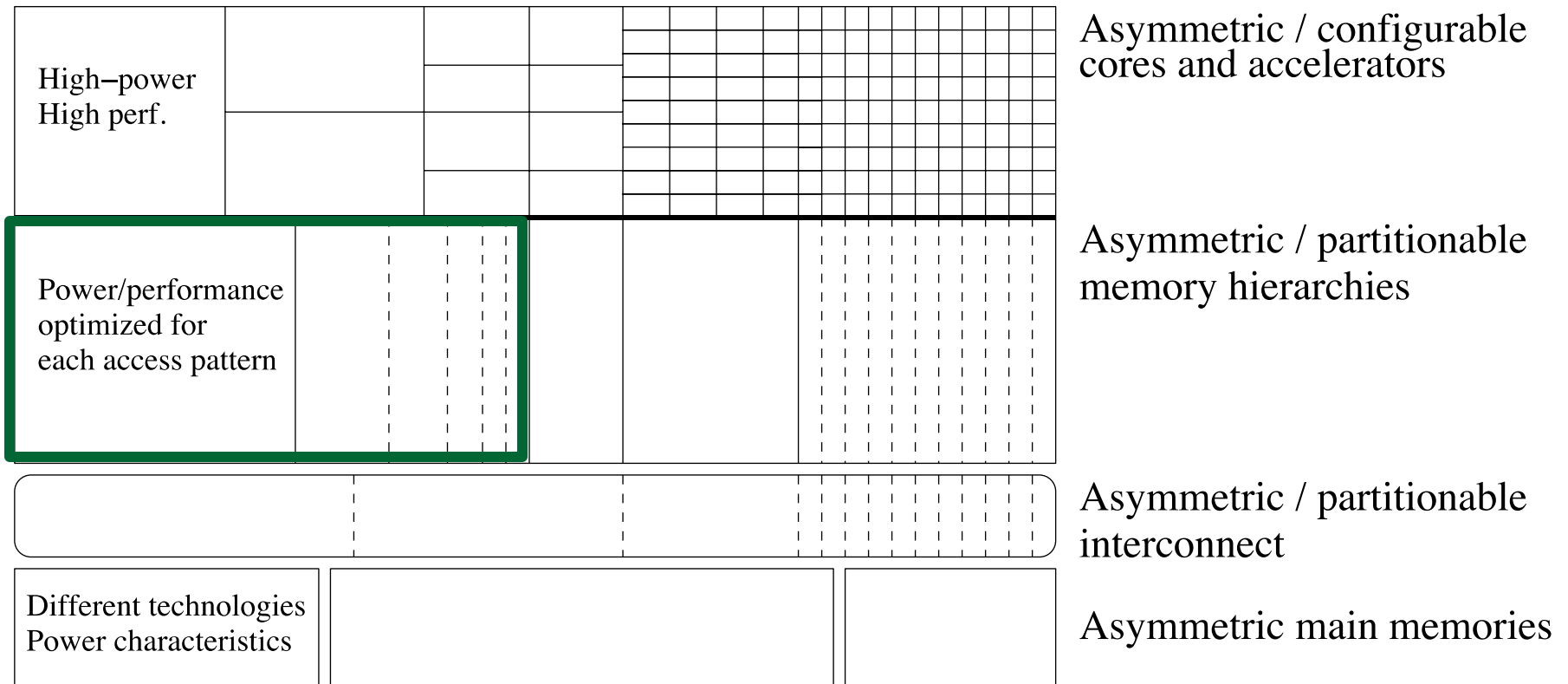
- How to design asymmetric components?
  - Fixed, partitionable, reconfigurable components?
  - What types of asymmetry? Access patterns, technologies?
- What monitoring to perform cooperatively in HW/SW?
  - Automatically discover phase/task requirements
- How to design feedback/control loop between components and runtime system software?
- How to design the runtime to automatically manage resources?
  - Track task behavior, pick “best-fit” components for the entire workload

# Exploiting Asymmetry: Simple Examples



- Execute critical/serial sections on high-power, high-performance cores/resources [Suleman+ ASPLOS'09, ISCA'10, Top Picks'10'11, Joao+ ASPLOS'12]
  - Programmer can write less optimized, but more likely correct programs

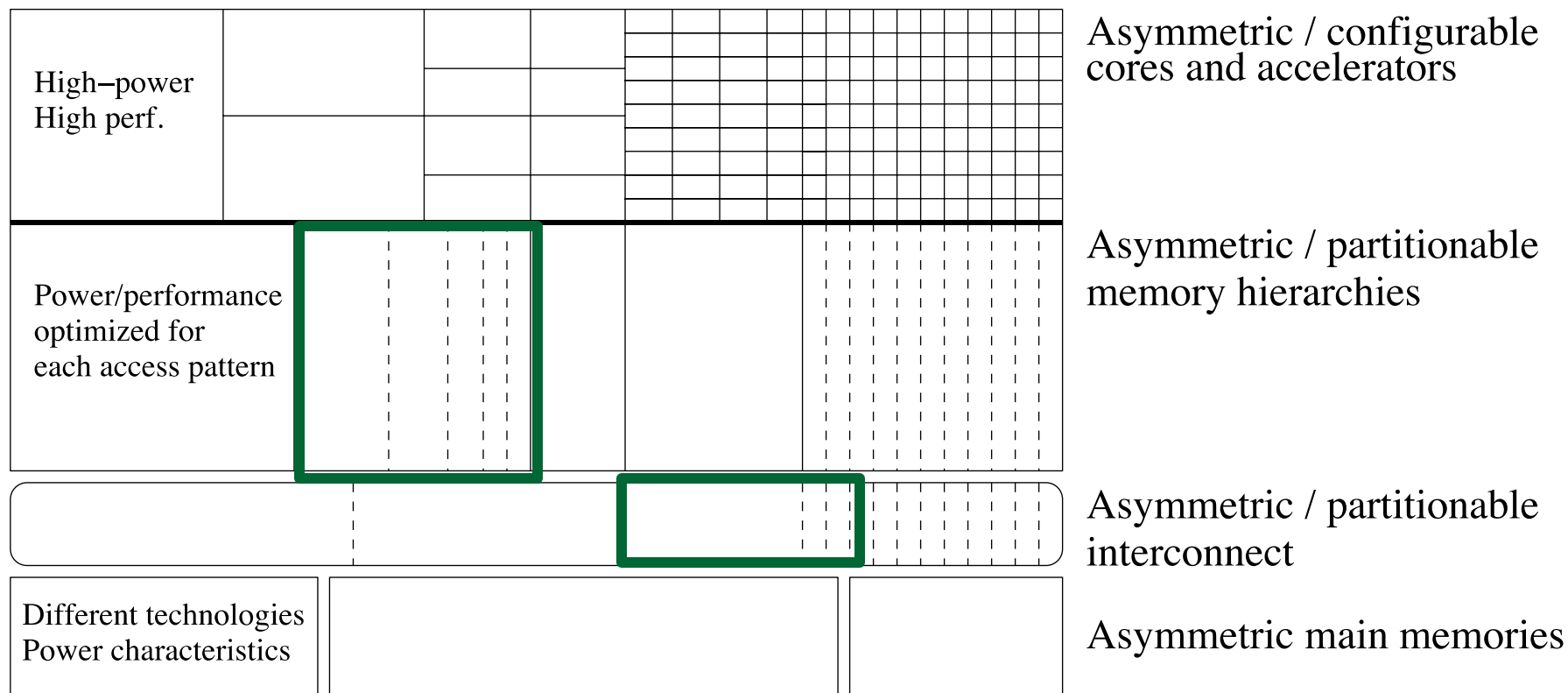
# Exploiting Asymmetry: Simple Examples



- Execute streaming “memory phases” on streaming-optimized cores and memory hierarchies
  - More efficient and higher performance than general purpose hierarchy

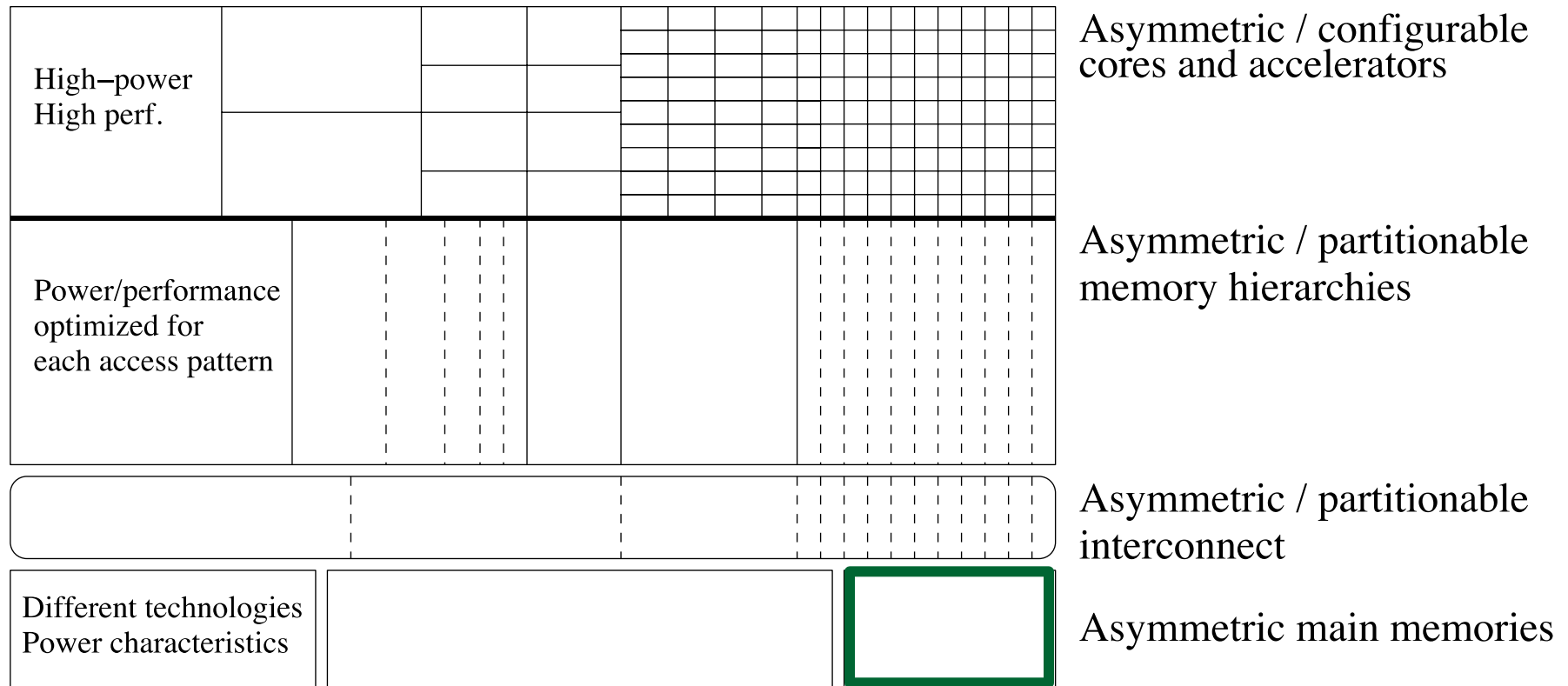


# Exploiting Asymmetry: Simple Examples



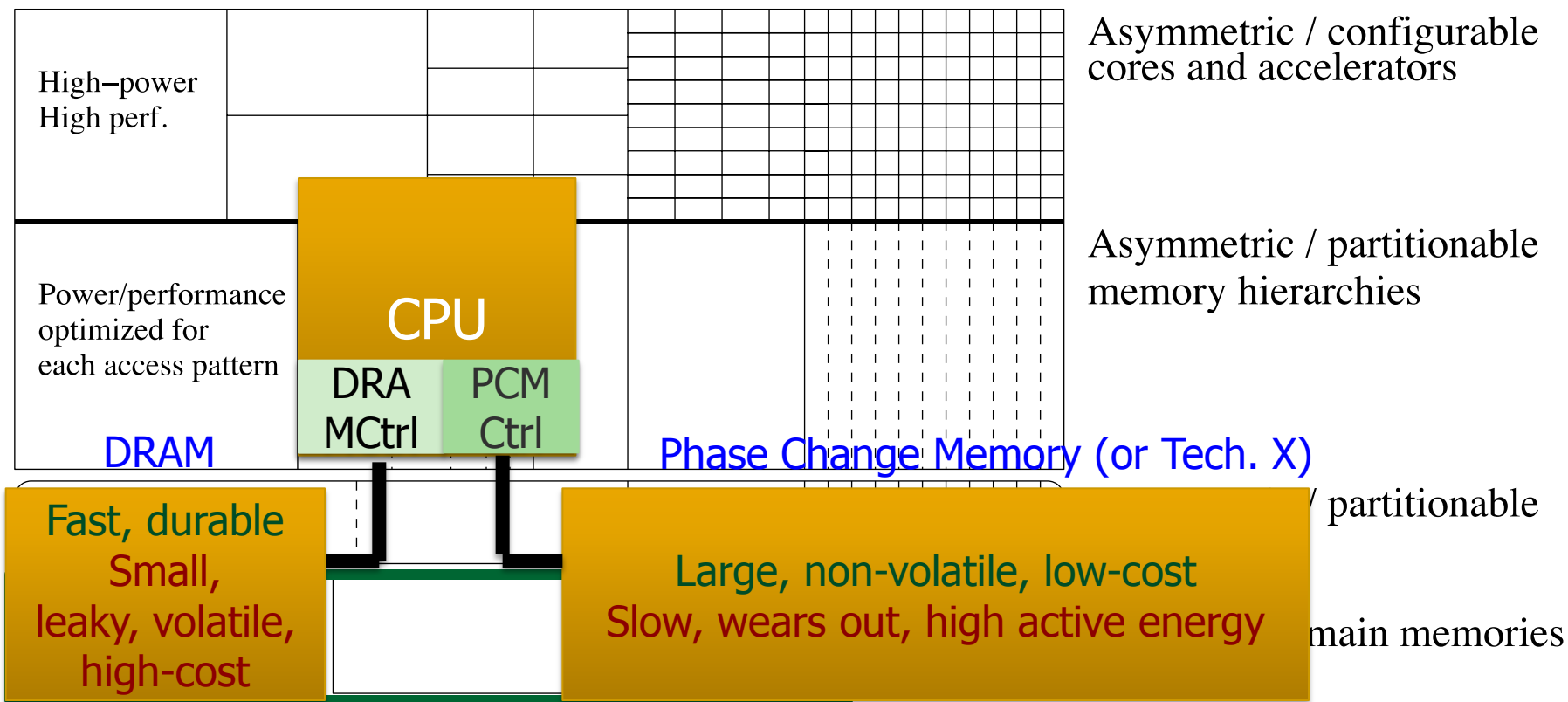
- Partition memory controller and on-chip network bandwidth asymmetrically among threads [Kim+ HPCA 2010, MICRO 2010, Top Picks 2011] [Nychis+ HotNets 2010] [Das+ MICRO 2009, ISCA 2010, Top Picks 2011]
  - Higher performance and energy-efficiency than symmetric/free-for-all

# Exploiting Asymmetry: Simple Examples



- Have multiple different memory scheduling policies; apply them to different sets of threads based on thread behavior [Kim+ MICRO 2010, Top Picks 2011] [Ausavarungnirun, ISCA 2012]
  - Higher performance and fairness than a homogeneous policy

# Exploiting Asymmetry: Simple Examples



- Build main memory with different technologies with different characteristics (energy, latency, wear, bandwidth) [Meza+ IEEE CAL'12]

- Map pages/applications to the best-fit memory resource

Higher performance and energy-efficiency than single-level memory