740: Computer Architecture
Architecting and Exploiting Asymmetry (in Multi-Core Architectures)

Prof. Onur Mutlu
Carnegie Mellon University

Three Key Problems in Future Systems

- Memory system
  - Many important existing and future applications are increasingly data intensive → require bandwidth and capacity
  - Data storage and movement limits performance & efficiency

- Efficiency (performance and energy) → scalability
  - Enables scalable systems → new applications
  - Enables better user experience → new usage models

- Predictability and robustness
  - Resource sharing and unreliable hardware causes QoS issues
  - Predictable performance and QoS are first class constraints

Readings for Today

- Required – Symmetric and Asymmetric Multi-Core Systems
  - Suleman et al., "Accelerating Critical Section Execution with Asymmetric Multi-Core Architectures," ASPLOS 2009, IEEE Micro 2010.
  - Suleman et al., "Data Marshaling for Multi-Core Architectures," ISCA 2010, IEEE Micro 2011.
  - Joao et al., "Bottleneck Identification and Scheduling for Multithreaded Applications," ASPLOS 2012.
  - Joao et al., "Utility-Based Acceleration of Multithreaded Applications on Asymmetric CMPs," ISCA 2013.

- Recommended
  - Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," AFIPS 1967.
  - Olukotun et al., "The Case for a Single-Chip Multiprocessor," ASPLOS 1996.
  - Mutlu et al., "Runahead Execution: An Alternative to Very Large Instruction Windows for Out-of-order Processors," HPCA 2003, IEEE Micro 2003.
  - Mutlu et al., "Techniques for Efficient Processing in Runahead Execution Engines," ISCA 2005, IEEE Micro 2006.

Warning

- This is an asymmetric lecture
- But, we do not need to cover all of it…
- Component 1: A case for asymmetry *everywhere*
- Component 2: A deep dive into mechanisms to exploit asymmetry in processing cores
- Component 3: Asymmetry in memory controllers
- Asymmetry = heterogeneity
    - A way to enable specialization/customization

The Setting

- Hardware resources are shared among many threads/apps in a many-core 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 consumption 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 computer architectures should be designed to
  - Minimize programmer effort to optimize (parallel) programs
  - Maximize runtime system's effectiveness in automatic shared resource management

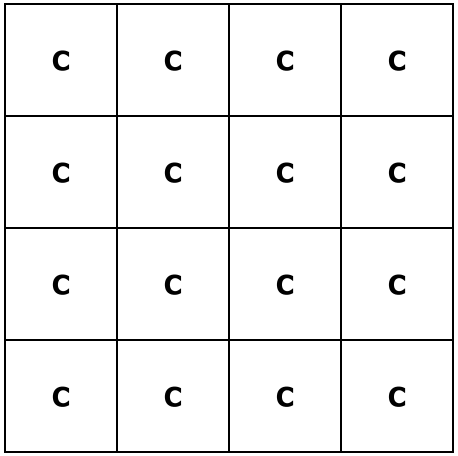
Shared Resource Management: Goals

- 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

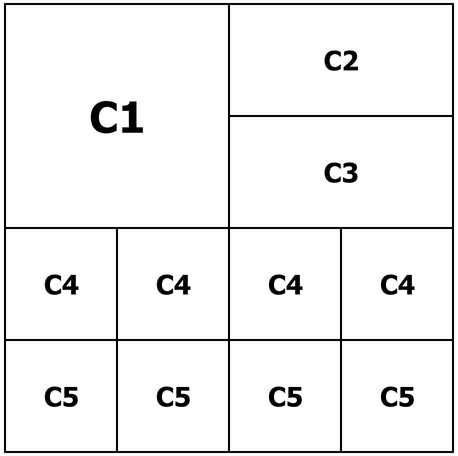
- Asymmetry and configurability in system resources essential to achieve these goals

Asymmetry Enables Customization



C	C	C	C
C	C	C	C
C	C	C	C
C	C	C	C

Symmetric



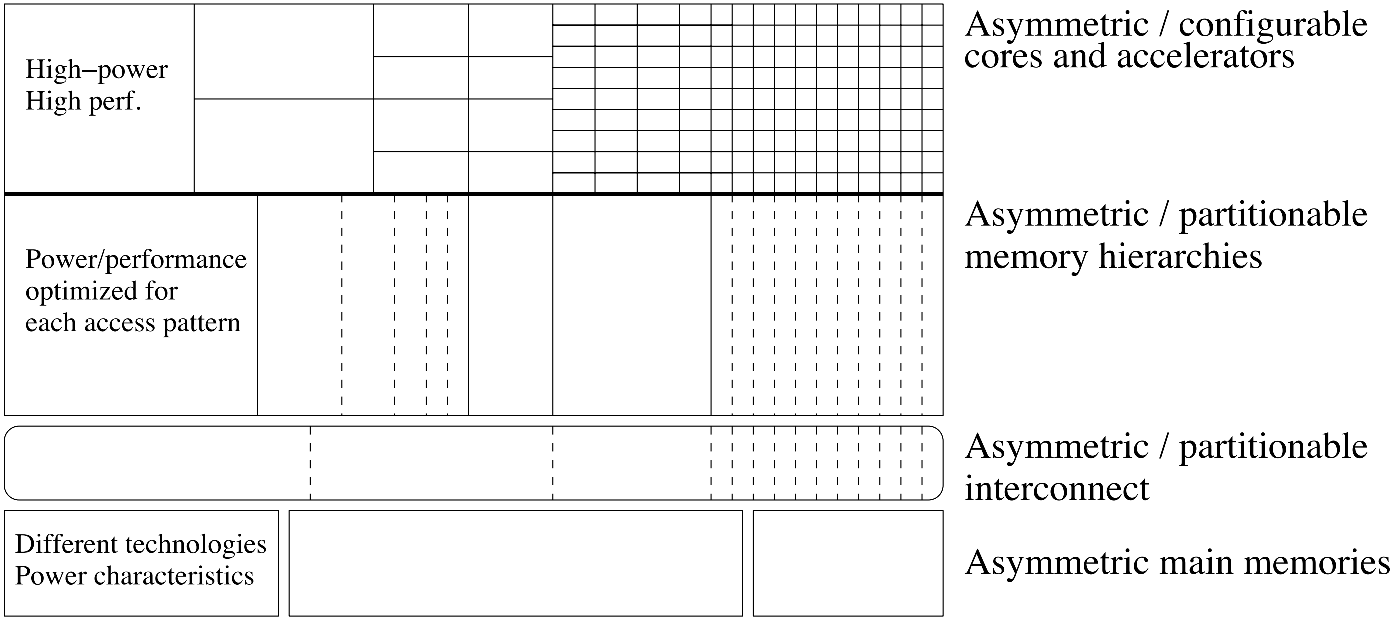
<table>
<tr><td colspan="2" rowspan="2">C1</td><td colspan="2">C2</td></tr>
<tr><td colspan="2">C3</td></tr>
<tr><td>C4</td><td>C4</td><td>C4</td><td>C4</td></tr>
<tr><td>C5</td><td>C5</td><td>C5</td><td>C5</td></tr>
</table>

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.)

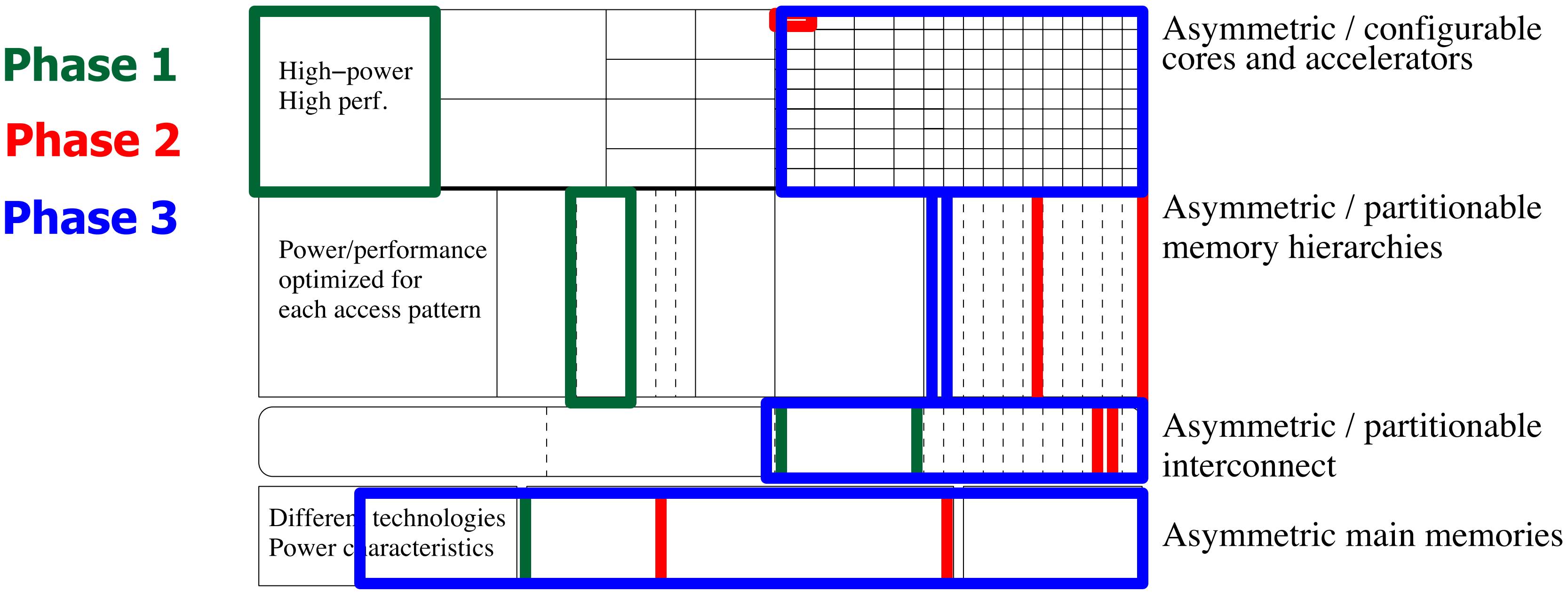
Thought Experiment: Asymmetry Everywhere

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



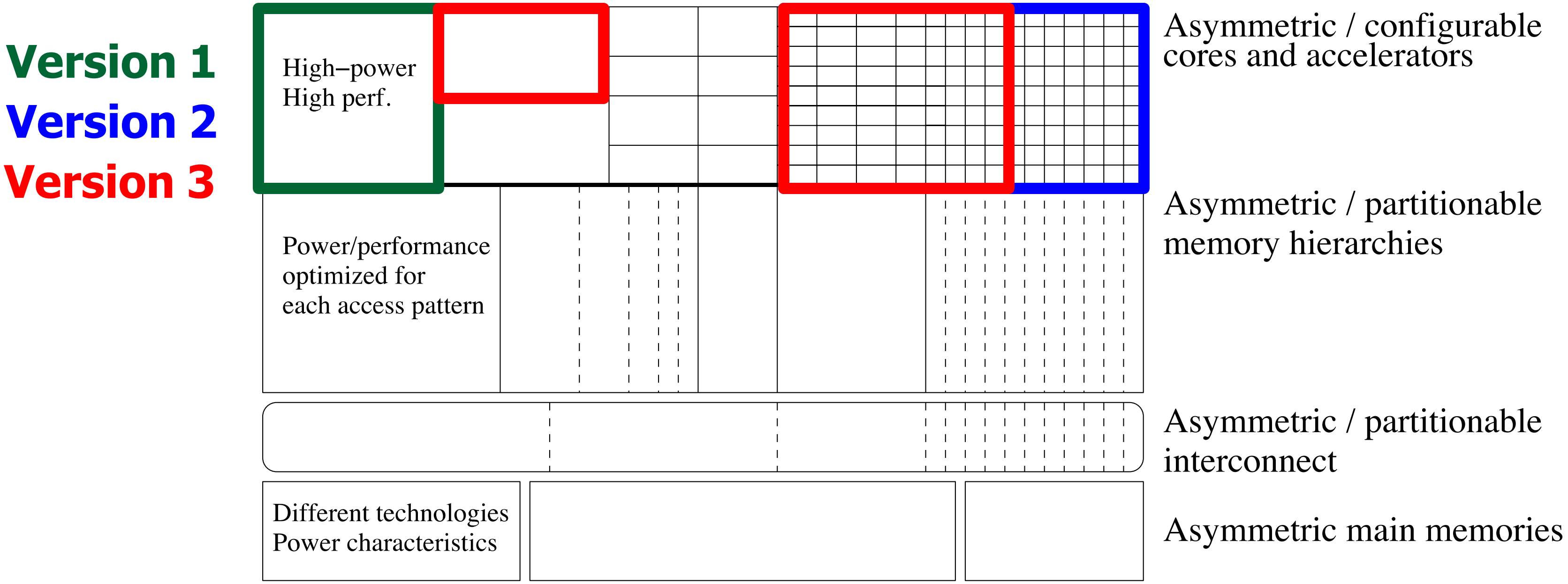
Thought Experiment: Asymmetry Everywhere

- Design the runtime system (HW & SW) to automatically choose the best-fit components for each 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

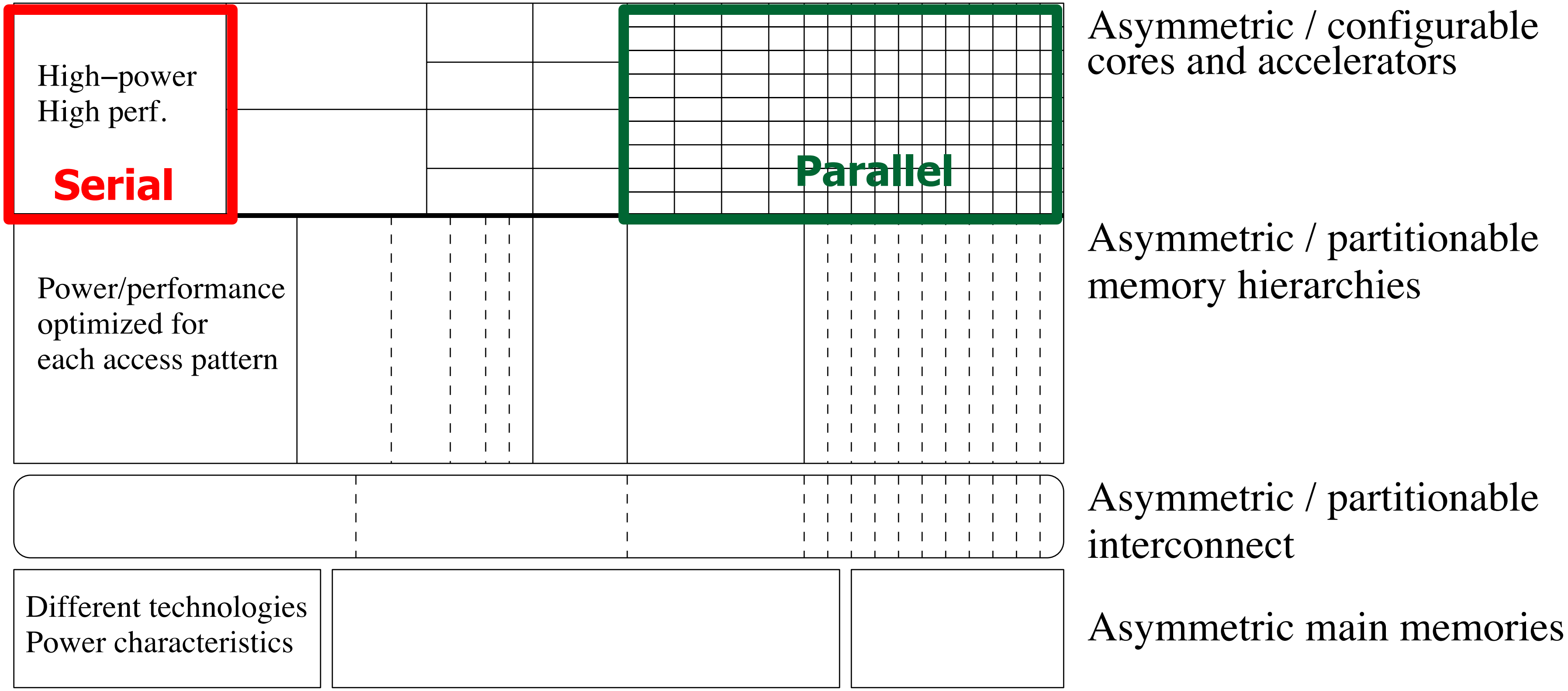
- How to design feedback/control loop between components and runtime system software?

- How to design the runtime to automatically manage resources?
    - Track task behavior, pick "best-fit" components for the entire workload

Talk Outline

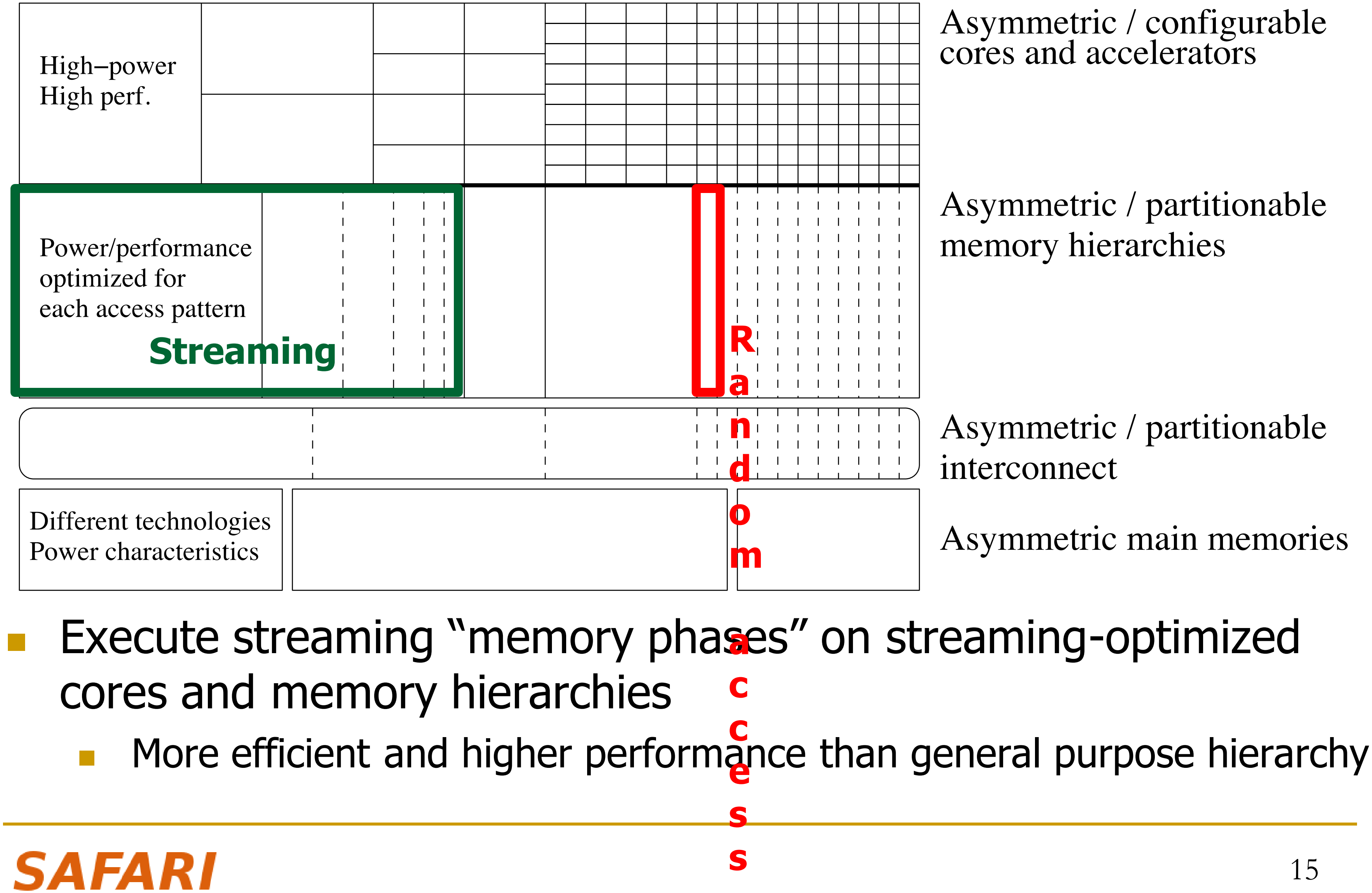
- Problem and Motivation
- How Do We Get There: Examples
- Accelerated Critical Sections (ACS)
- Bottleneck Identification and Scheduling (BIS)
- Staged Execution and Data Marshaling
- Thread Cluster Memory Scheduling (if time permits)
- Ongoing/Future Work
- Conclusions

Exploiting Asymmetry: Simple Examples


High–power
High perf.
Serial
Parallel
Asymmetric / configurable cores and accelerators
Power/performance optimized for each access pattern
Asymmetric / partitionable memory hierarchies
Asymmetric / partitionable interconnect
Different technologies
Power characteristics
Asymmetric main memories

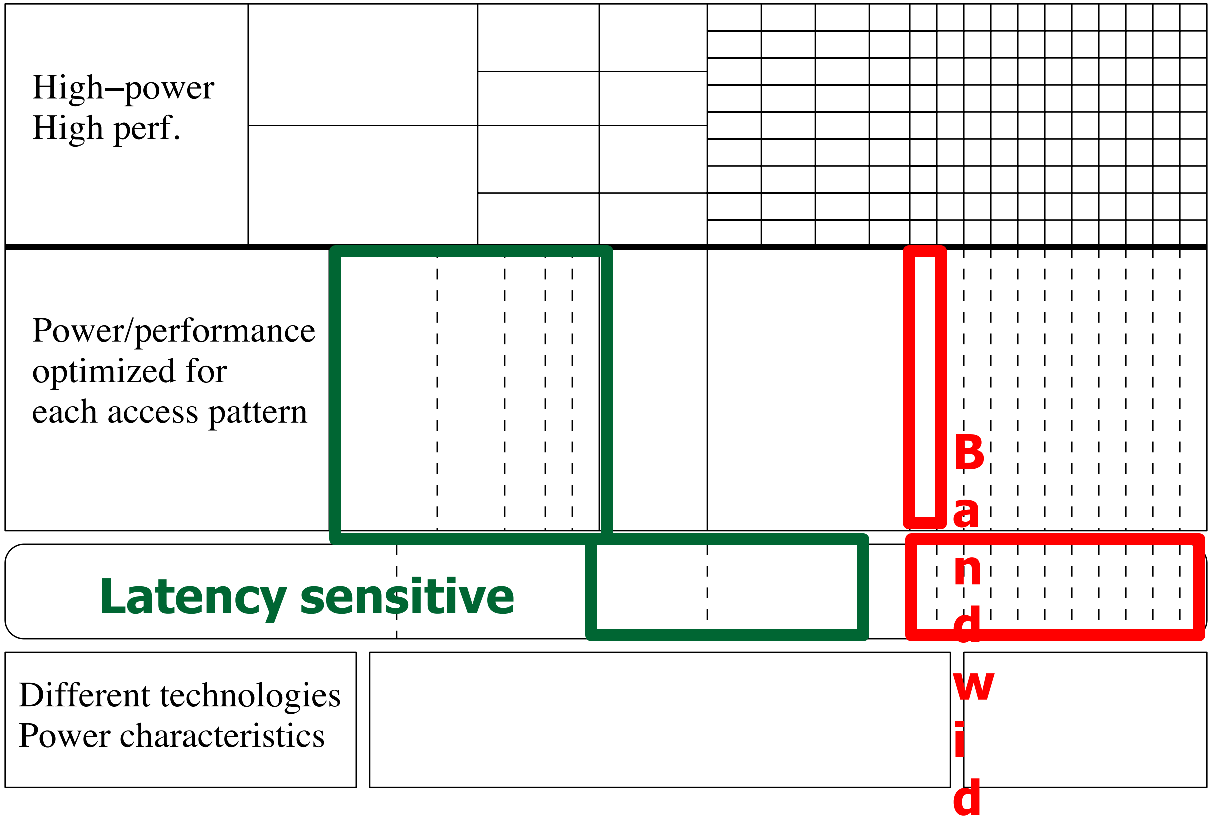
- Execute critical/serial sections on high-power, high-performance cores/resources [Suleman+ ASPLOS'09, ISCA'10, Top Picks'10'11, Joao+ ASPLOS'12]
  - Programmer can write less optimized, but more likely correct programs

Exploiting Asymmetry: Simple Examples


High–power
High perf.
Asymmetric / configurable cores and accelerators
Power/performance optimized for each access pattern
Streaming
Random access
Asymmetric / partitionable memory hierarchies
Asymmetric / partitionable interconnect
Different technologies
Power characteristics
Asymmetric main memories

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

Exploiting Asymmetry: Simple Examples



Asymmetric / configurable cores and accelerators

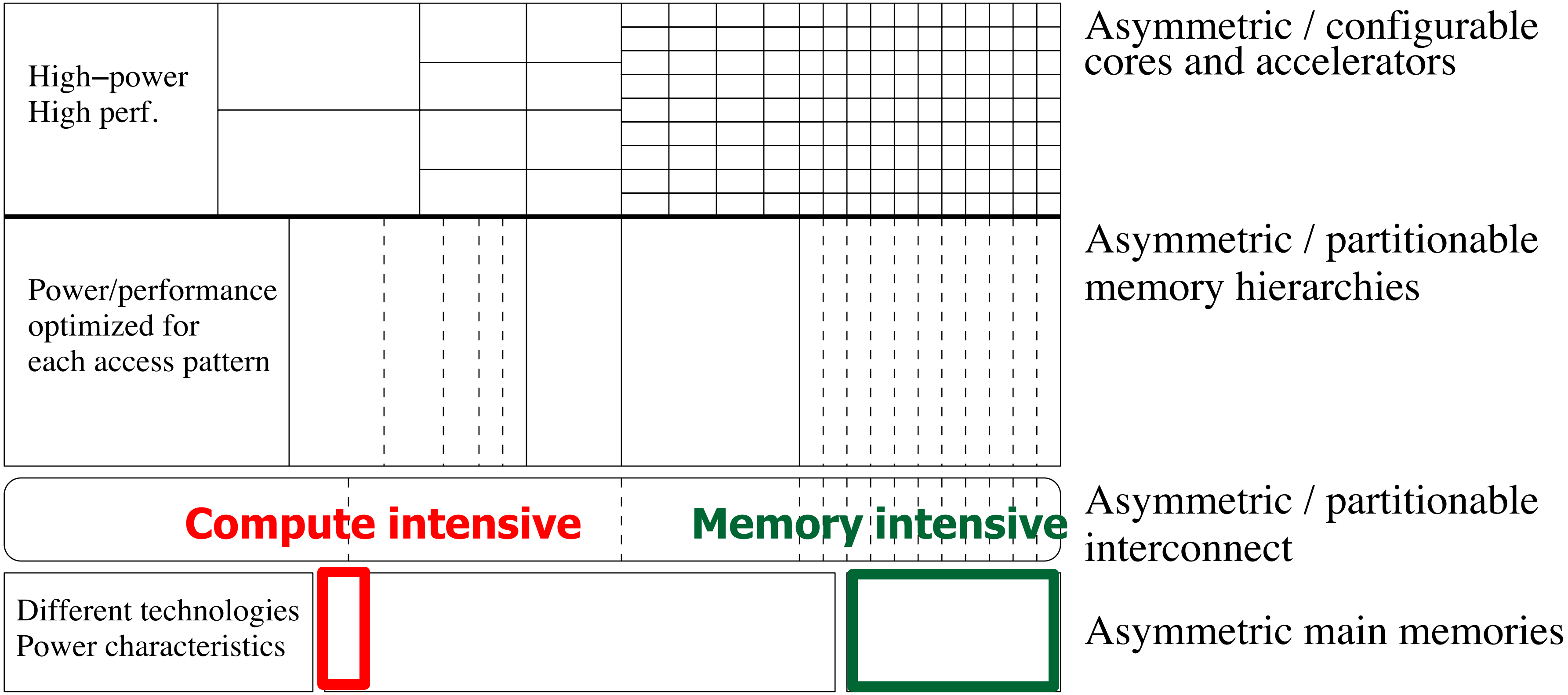
Asymmetric / partitionable memory hierarchies

Asymmetric / partitionable interconnect

Asymmetric main memories

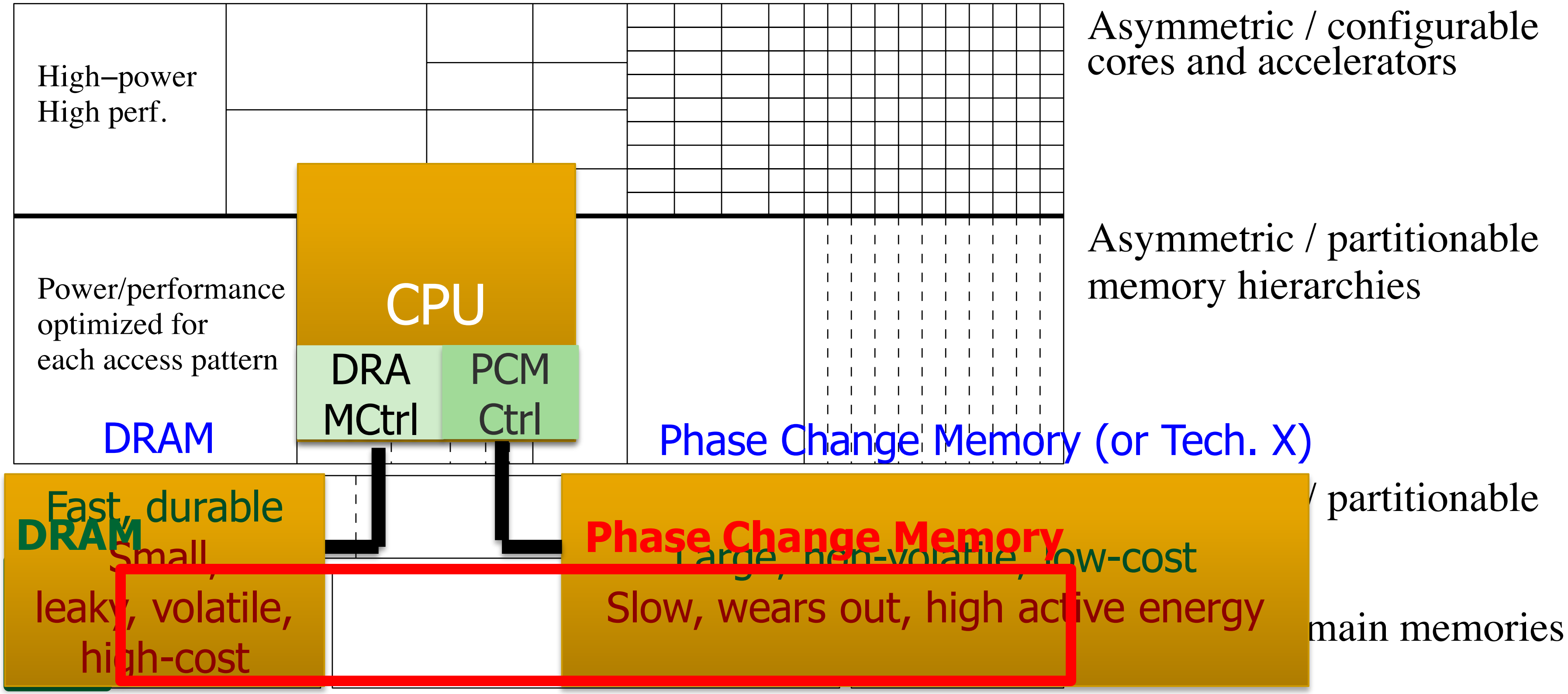
- Partition memory controller and on-chip network bandwidth asymmetrically among threads [Kim+ HPCA 2010, MICRO 2010, Top Picks 2011] [Nychis+ HotNets 2010] [Das+ MICRO 2009, ISCA 2010, Top Picks 2011]
  - Higher performance and energy-efficiency than symmetric/free-for-all

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


High–power
High perf.
Power/performance optimized for each access pattern
DRAM
CPU
DRAMCtrl
PCM Ctrl
Phase Change Memory (or Tech. X)
Fast, durable
DRAM
Small,
leaky, volatile, high-cost
Phase Change Memory
Large, non-volatile, low-cost
Slow, wears out, high active energy

Asymmetric / configurable cores and accelerators

Asymmetric / partitionable memory hierarchies

/ partitionable

main memories

- 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

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

Serialized Code Sections in Parallel Applications

- Multithreaded applications:
  - Programs split into threads

- Threads execute concurrently on multiple cores

- Many parallel programs cannot be parallelized completely

- Serialized code sections:
  - Reduce performance
  - Limit scalability
  - Waste energy

Causes of Serialized Code Sections

- Sequential portions (Amdahl's "serial part")
- Critical sections
- Barriers
- Limiter stages in pipelined programs

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

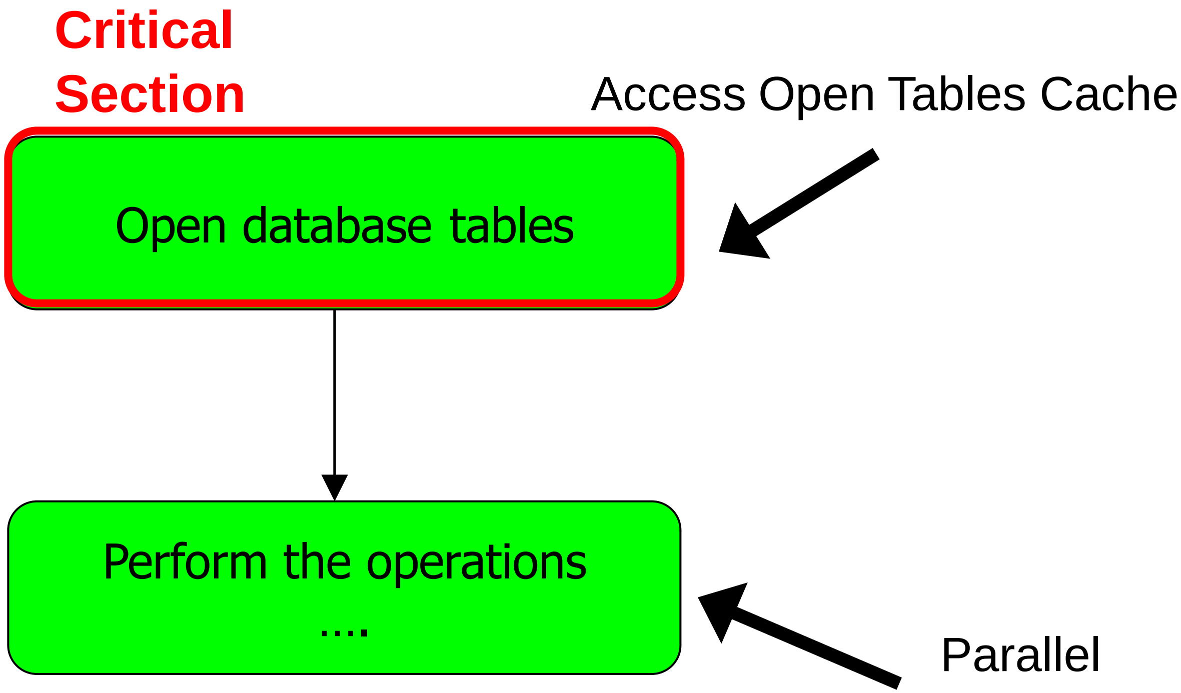
Critical Sections

- Threads are not allowed to update shared data concurrently
  - For correctness (mutual exclusion principle)

- Accesses to shared data are encapsulated inside ***critical sections***

- Only one thread can execute a critical section at a given time

Example from MySQL



Critical Section

Access Open Tables Cache

Open database tables

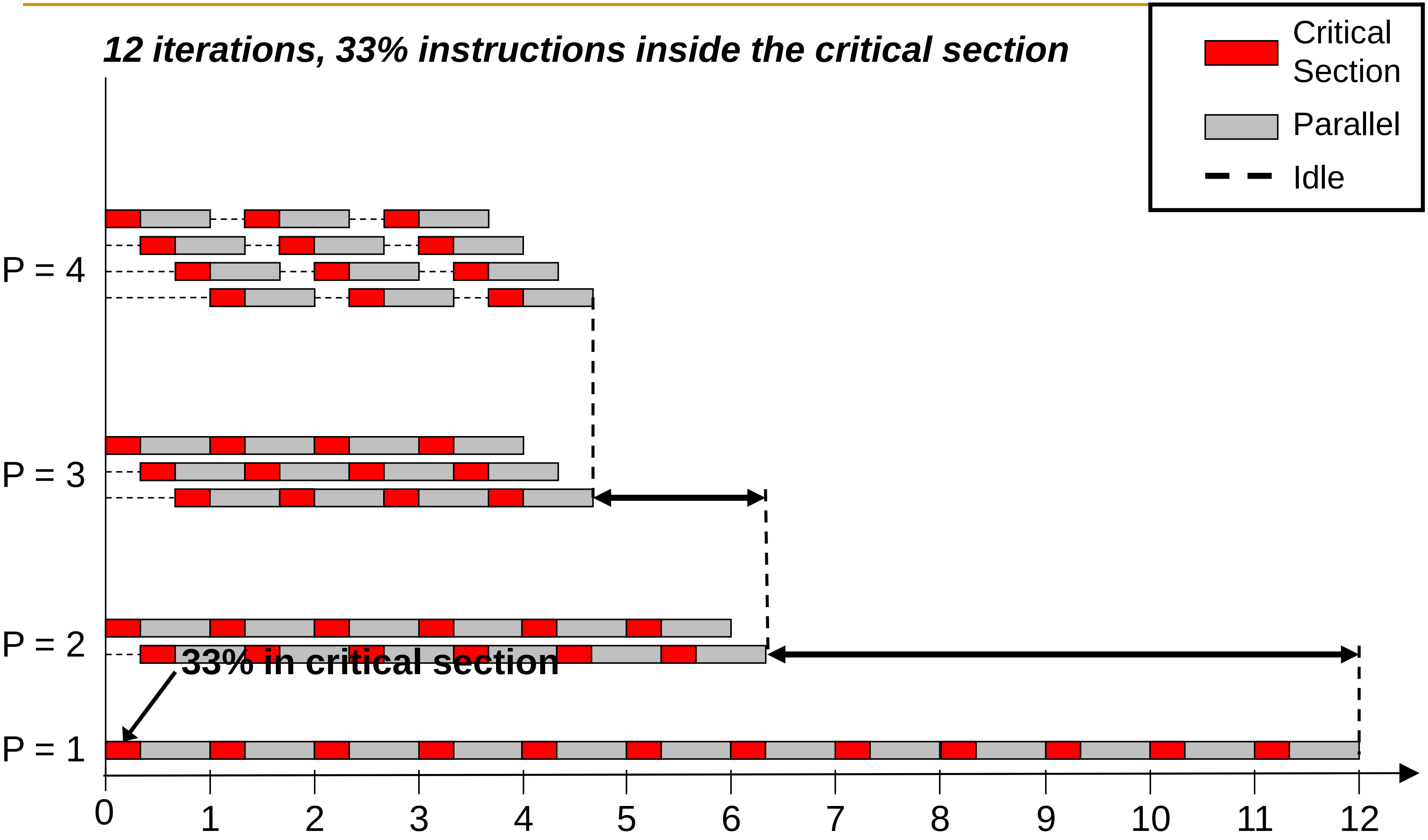
Perform the operations

….

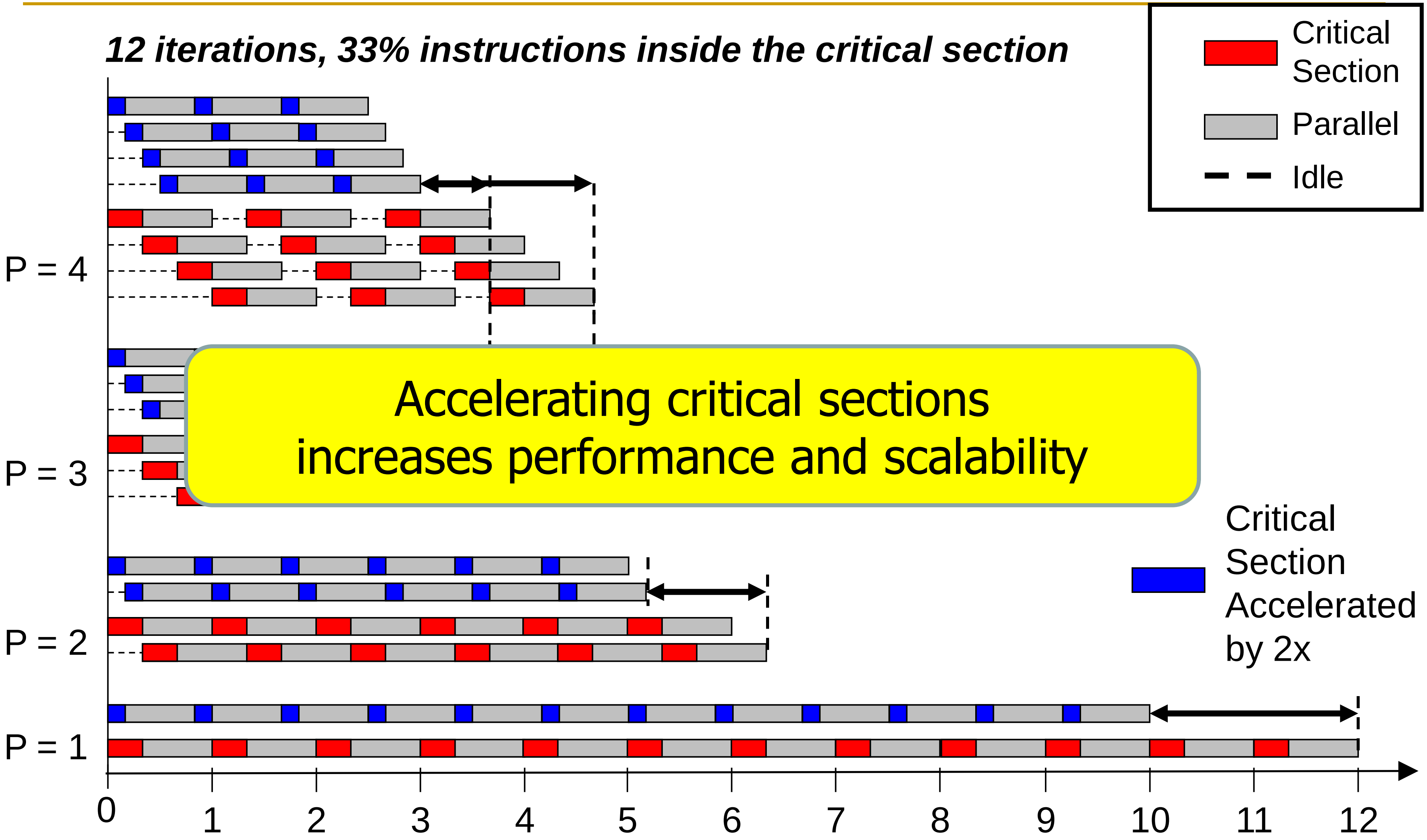
Parallel

SAFARI

Contention for Critical Sections



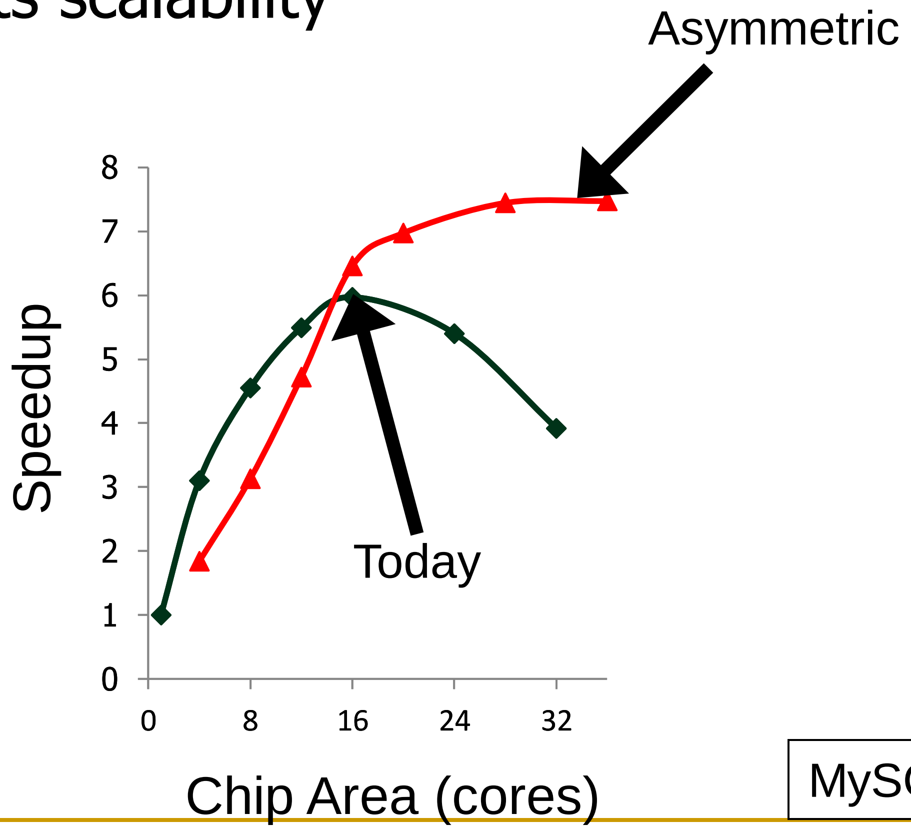
Contention for Critical Sections



Accelerating critical sections increases performance and scalability

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

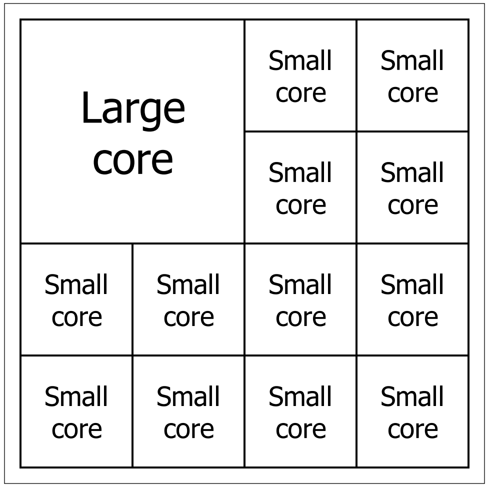
- Execution time of sequential kernels, critical sections, and limiter stages must be short

- 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)

ACMP



Large 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

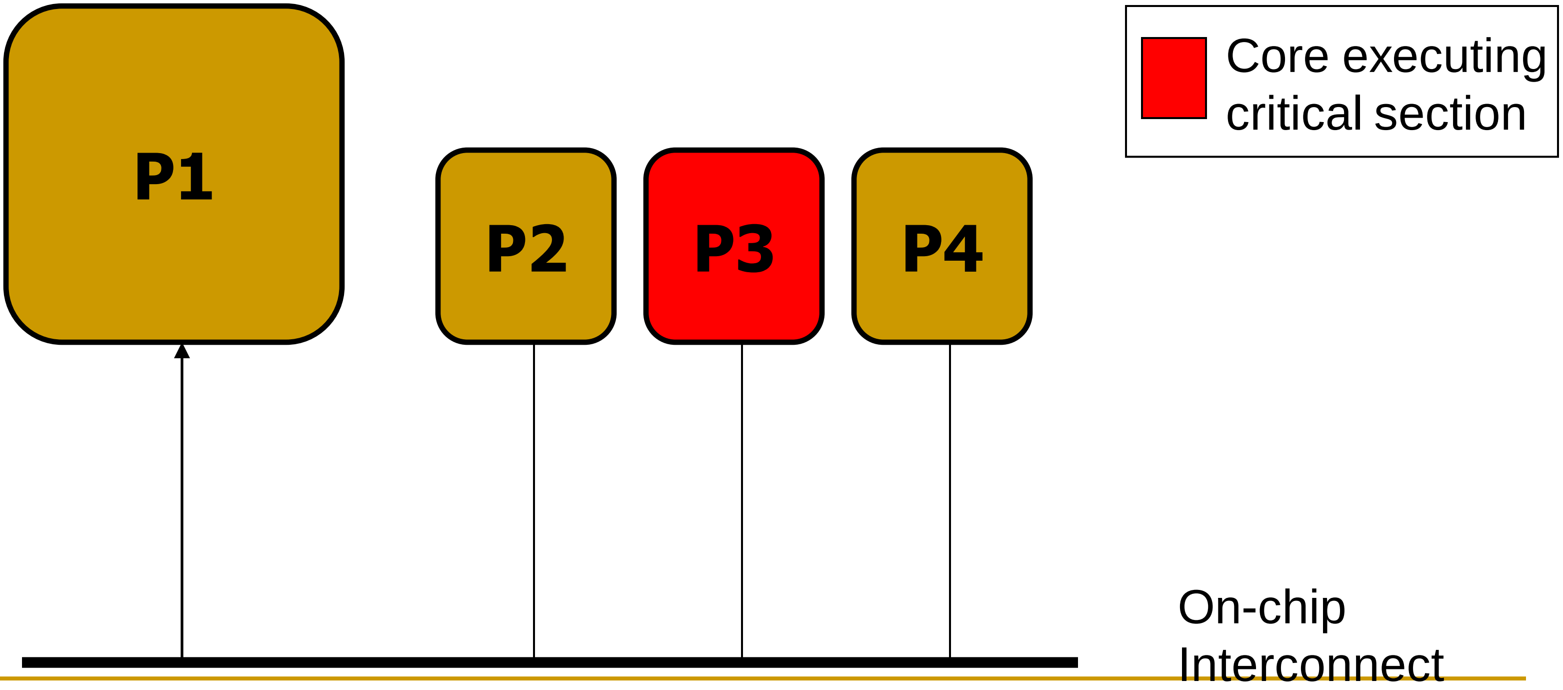
ACMP

- Provide one large core and many small cores
- Execute parallel part on small cores for high throughput
- Accelerate serialized sections using the large core
  - Baseline: Amdahl’s serial part accelerated [Morad+ CAL 2006, Suleman+, UT-TR 2007]

Conventional ACMP

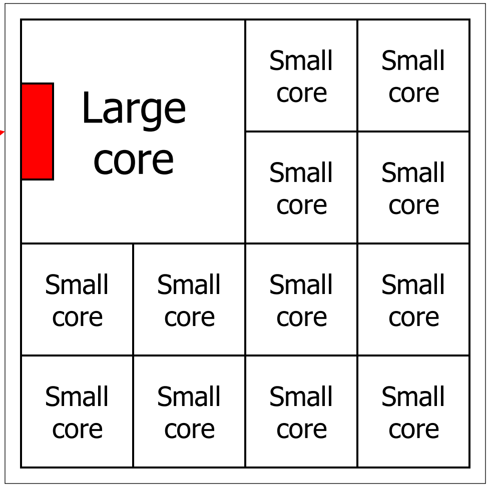
```
EnterCS()
          PriorityQ.insert(…)
LeaveCS()
```

1. **P2 encounters a Critical Section**
2. **Sends a request for the lock**
3. **Acquires the lock**
4. **Executes Critical Section**
5. **Releases the lock**



Accelerated Critical Sections (ACS)

Critical Section Request Buffer (CSRB)



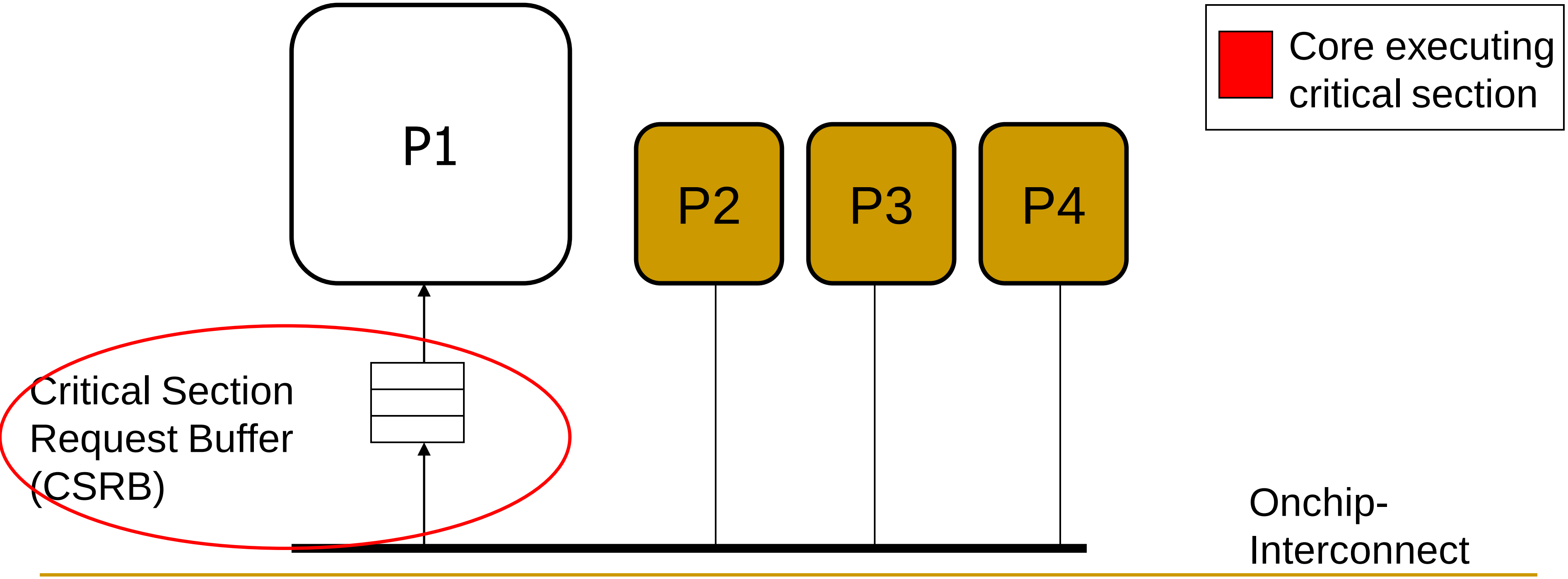
ACMP

- Accelerate Amdahl’s serial part **and critical sections** using the large core
  - Suleman et al., “Accelerating Critical Section Execution with Asymmetric Multi-Core Architectures,” ASPLOS 2009, IEEE Micro Top Picks 2010.

Accelerated Critical Sections (ACS)

```
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


🟥 Core executing critical section
P1
P2
P3
P4
Critical Section Request Buffer (CSRB)
Onchip-Interconnect

SAFARI

ACS Architecture Overview

- ISA extensions
  - CSCALL *LOCK_ADDR, TARGET_PC*
  - CSRET *LOCK_ADDR*

- Compiler/Library inserts CSCALL/CSRET

- On a CSCALL, the small core:
  - Sends a CSCALL request to the large core
    - Arguments: Lock address, Target PC, Stack Pointer, Core ID
  - Stalls and waits for CSDONE

- Large Core
  - Critical Section Request Buffer (CSRB)
  - Executes the critical section and sends CSDONE to the requesting core

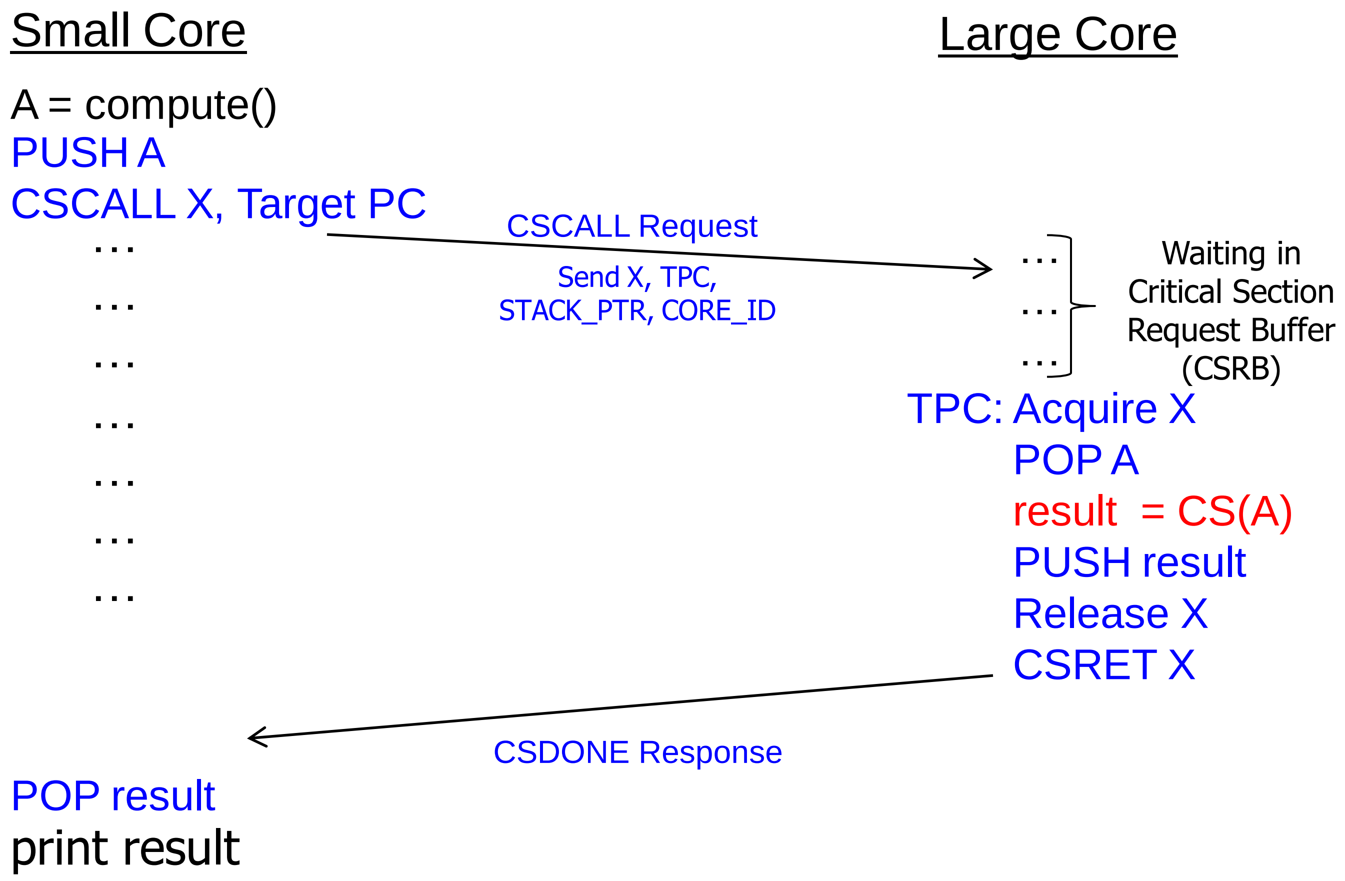
Accelerated Critical Sections (ACS)

Small Core

```
A = compute()

LOCK X
    result = CS(A)
UNLOCK X

print result
```



Small Core

```
A = compute()
PUSH A
CSCALL X, Target PC
   …
   …
   …
   …
   …
   …
   …
POP result
print result
```

CSCALL Request

Send X, TPC,
STACK_PTR, CORE_ID

CSDONE Response

Large Core

```
…
…
…
```

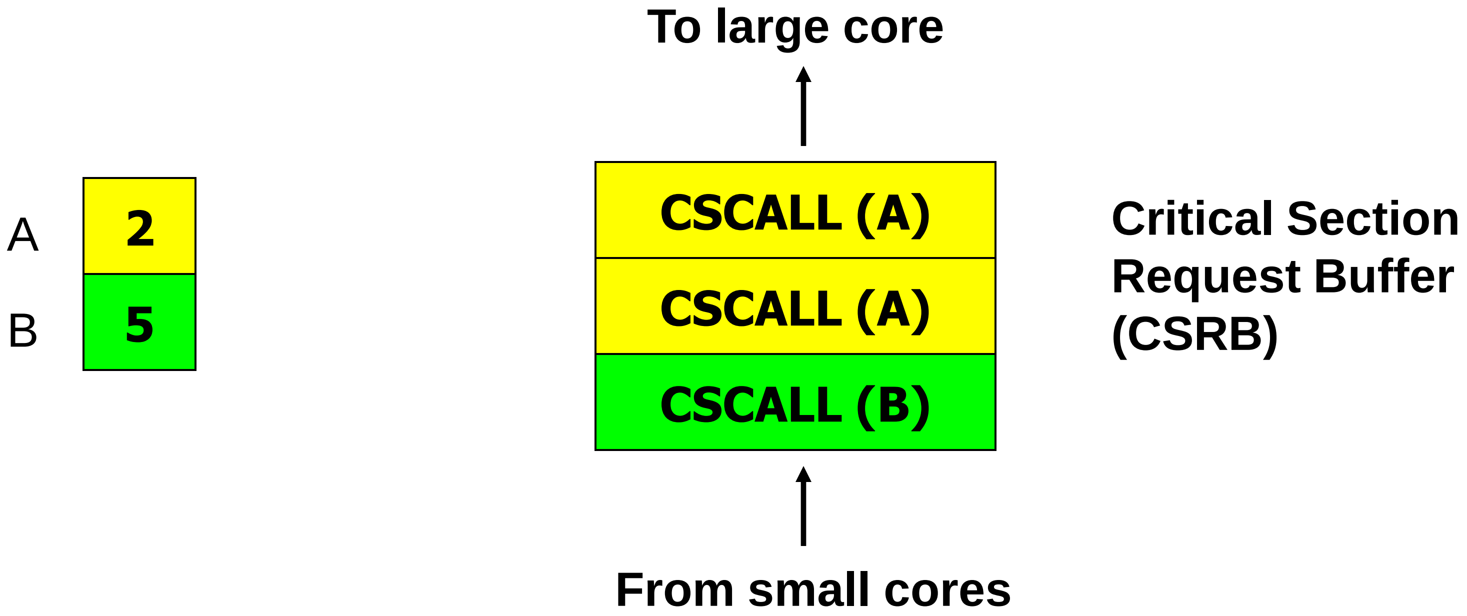
Waiting in Critical Section Request Buffer (CSRB)

```
TPC: Acquire X
     POP A
     result  = CS(A)
     PUSH result
     Release X
     CSRET X
```

False Serialization

- ACS can serialize independent critical sections

- Selective Acceleration of Critical Sections (SEL)
  - Saturating counters to track false serialization



A 2

B 5

To large core

CSCALL (A)

CSCALL (A)

CSCALL (B)

From small cores

Critical Section Request Buffer (CSRB)

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

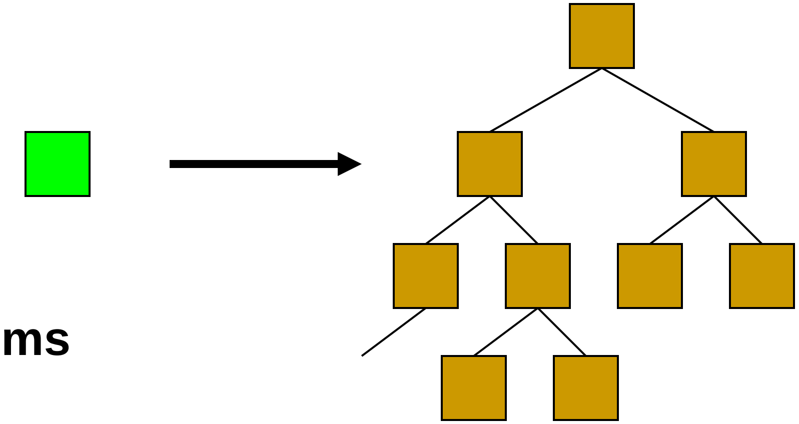
- ***Fewer parallel threads vs. accelerated critical sections***
  - Accelerating critical sections offsets loss in throughput
  - As the number of cores (threads) on chip increase:
    - Fractional loss in parallel performance decreases
    - Increased contention for critical sections makes acceleration 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

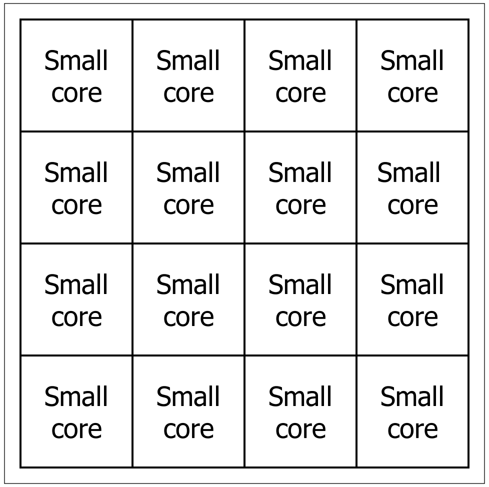
- ***Fewer parallel threads vs. accelerated critical sections***
  - Accelerating critical sections offsets loss in throughput
  - As the number of cores (threads) on chip increase:
    - Fractional loss in parallel performance decreases
    - Increased contention for critical sections
      makes acceleration 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

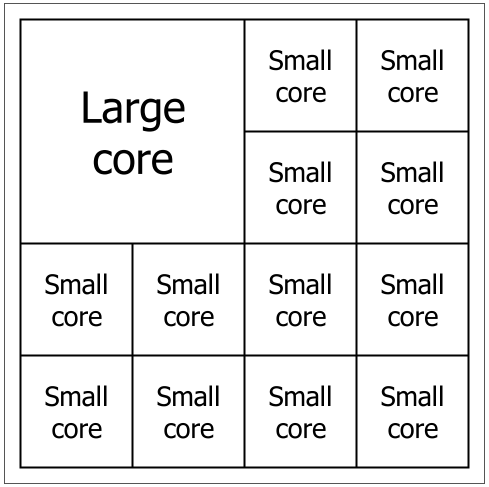
We will get back to this

ACS Comparison Points


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

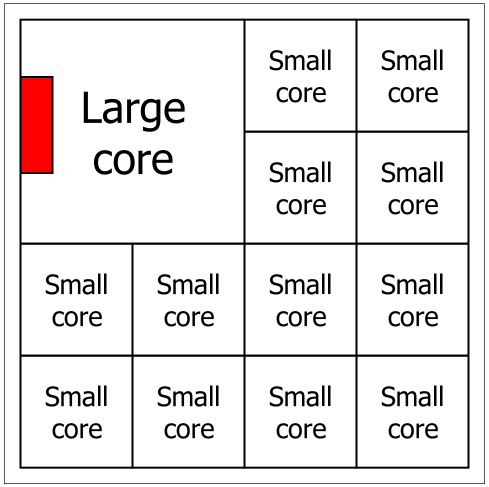
SCMP

- Conventional locking


Large 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

ACMP

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


Large 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

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

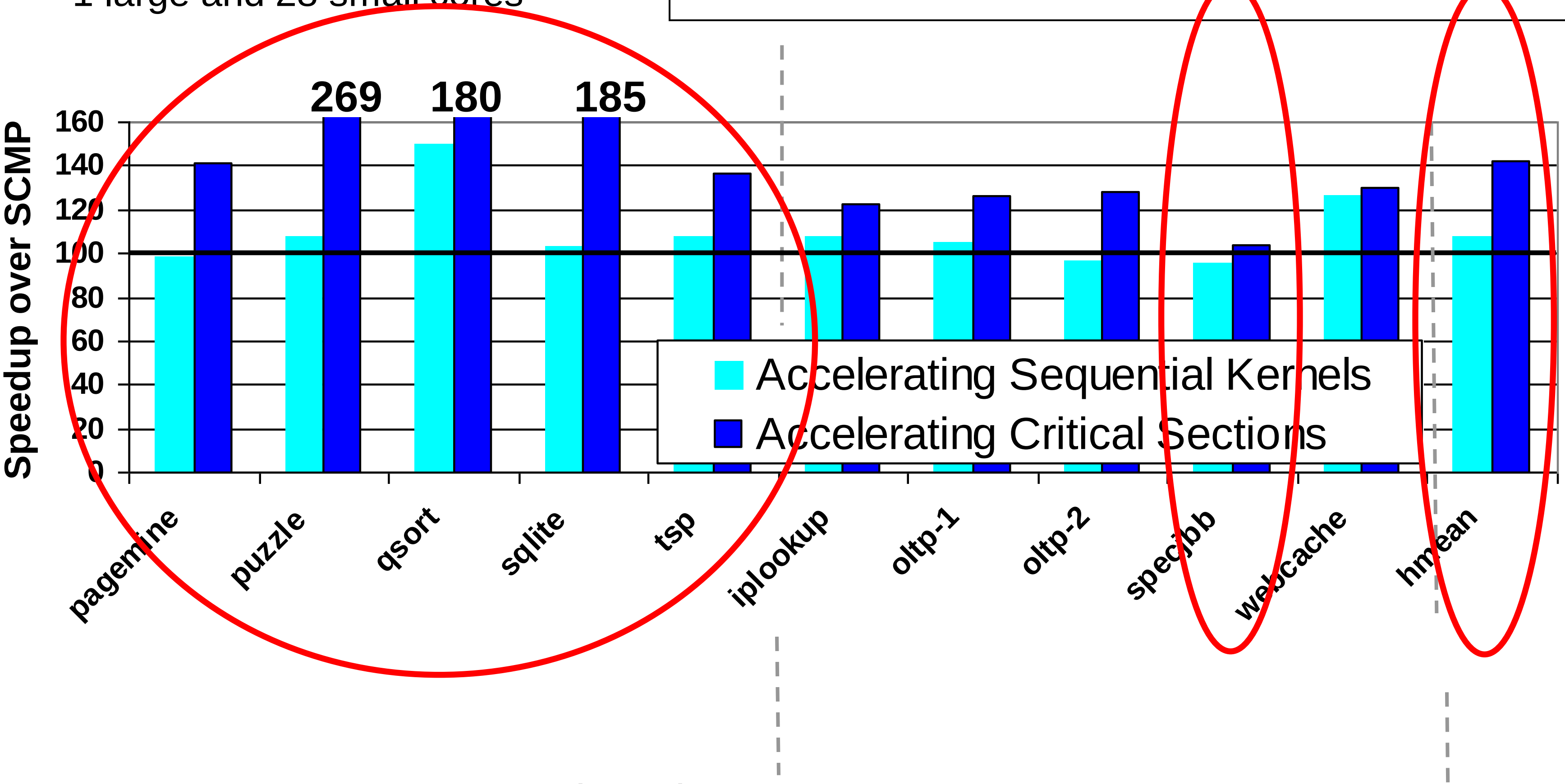
Chip Area = 32 small cores

SCMP = 32 small cores

ACMP = 1 large and 28 small cores

Equal-area comparison

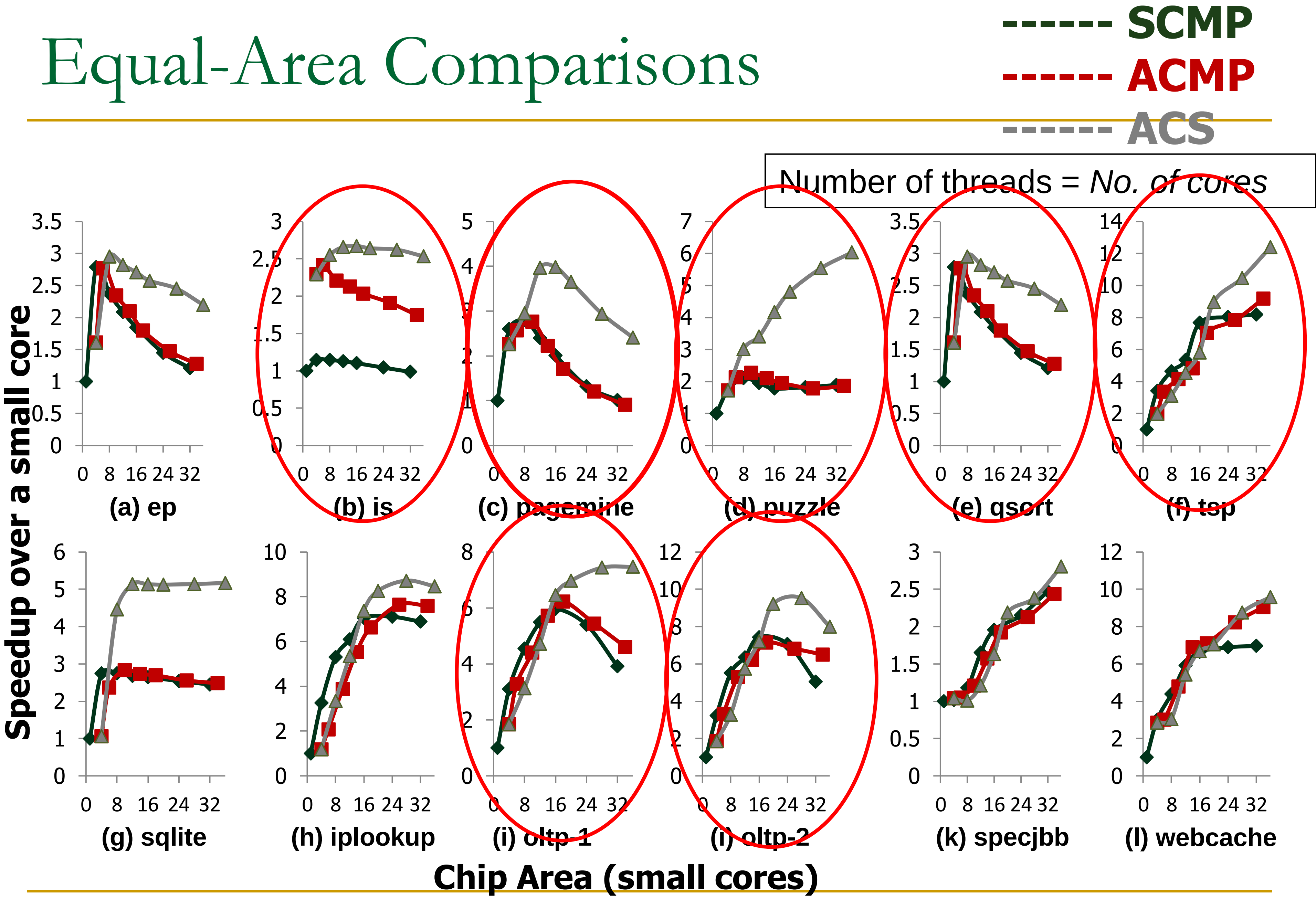
Number of threads = *Best threads*



Coarse-grain locks

Fine-grain locks

Equal-Area Comparisons



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

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

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

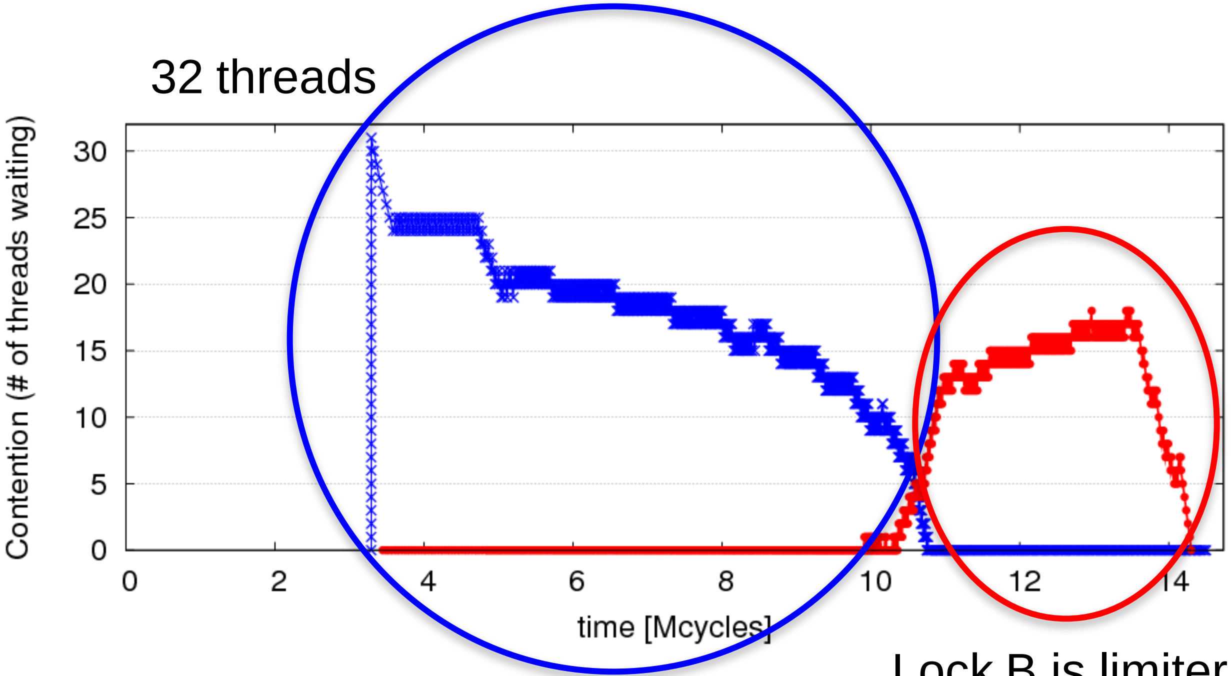
Definition: any code segment for which threads contend (i.e. wait)

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

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
```

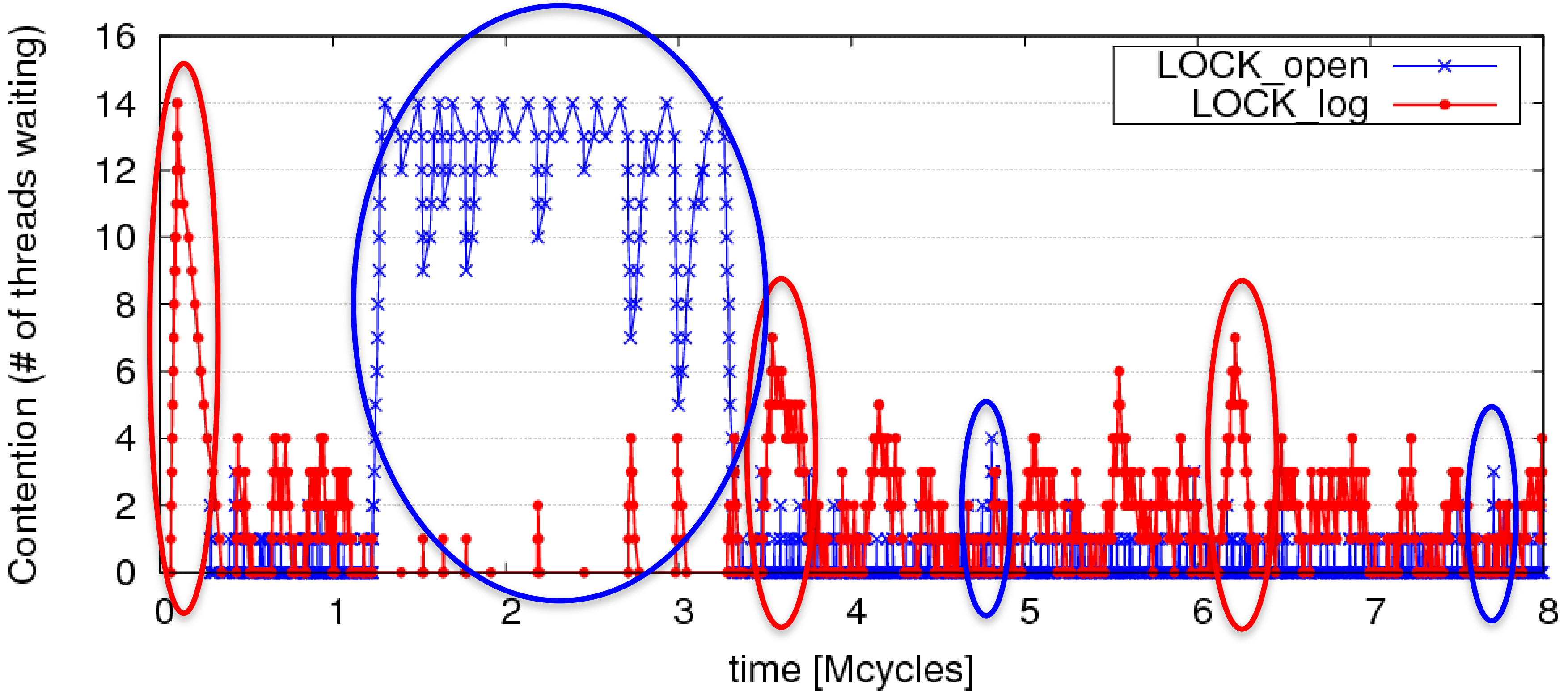

32 threads

Lock B is limiter

Lock A is limiter

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]
  - Accelerate only the Amdahl's bottleneck
- Accelerated Critical Sections (ACS) [Suleman+, ASPLOS'09]
  - Accelerate only critical sections
  - Does not take into account importance of critical sections
- Feedback-Directed Pipelining (FDP) [Suleman+, PACT'10 and PhD thesis'11]
  - Accelerate only stages with lowest throughput
  - Slow to adapt to phase changes (software based library)

No previous work can accelerate all three types of bottlenecks or quickly adapts to fine-grain changes in the *importance* of bottlenecks

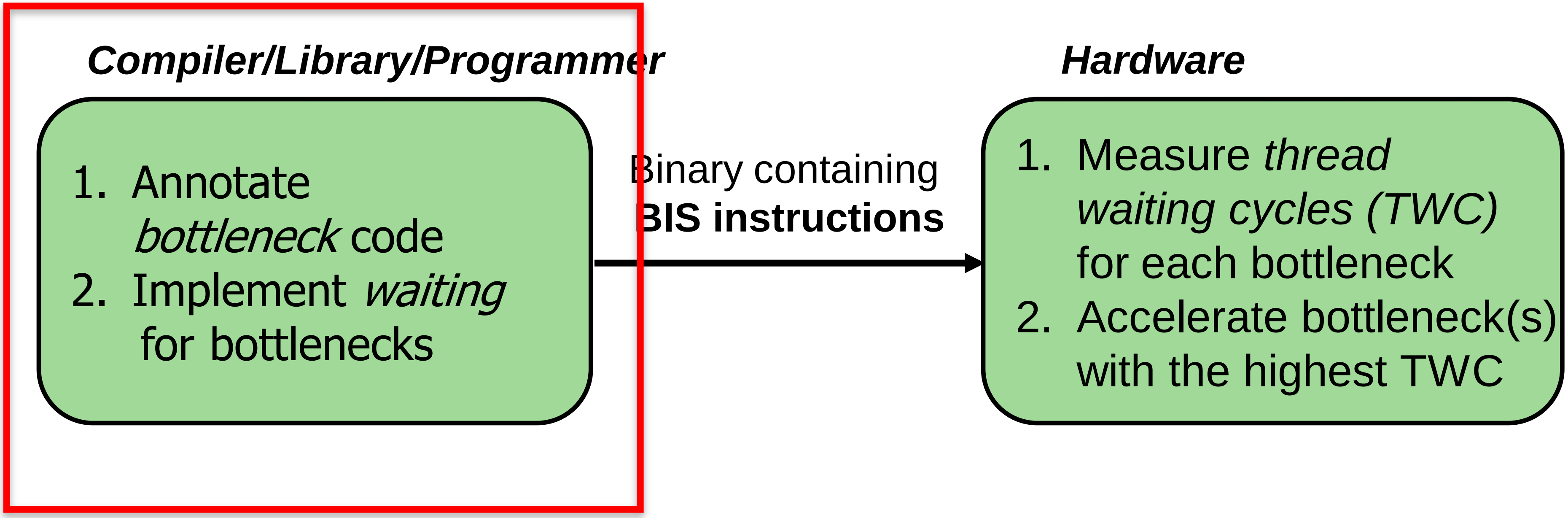
Our goal: general mechanism to identify performance-limiting bottlenecks of any type and accelerate them on an ACMP

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

```
...
while cannot acquire lock
        Wait loop for watch_addr
acquire lock
...
release lock
```

```
            ...
            BottleneckCall bid, targetPC
            ...
targetPC:   while cannot acquire lock
                Wait loop for watch_addr
                BottleneckWait bid, watch_addr
            acquire lock
            ...
            release lock
            BottleneckReturn bid
```

Used to enable acceleration

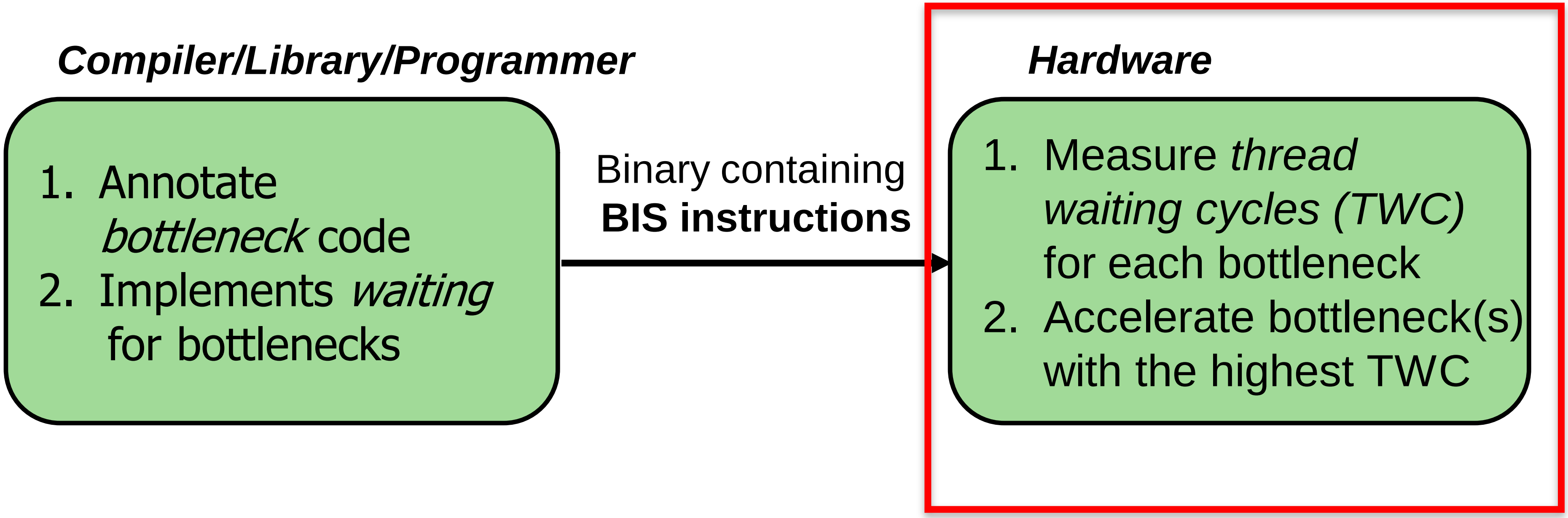
Barriers: Code Modifications

```
          …
          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)



Compiler/Library/Programmer

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

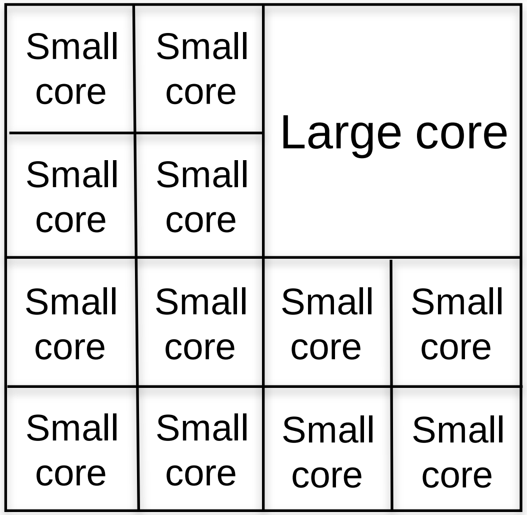
Binary containing **BIS instructions**

Hardware

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

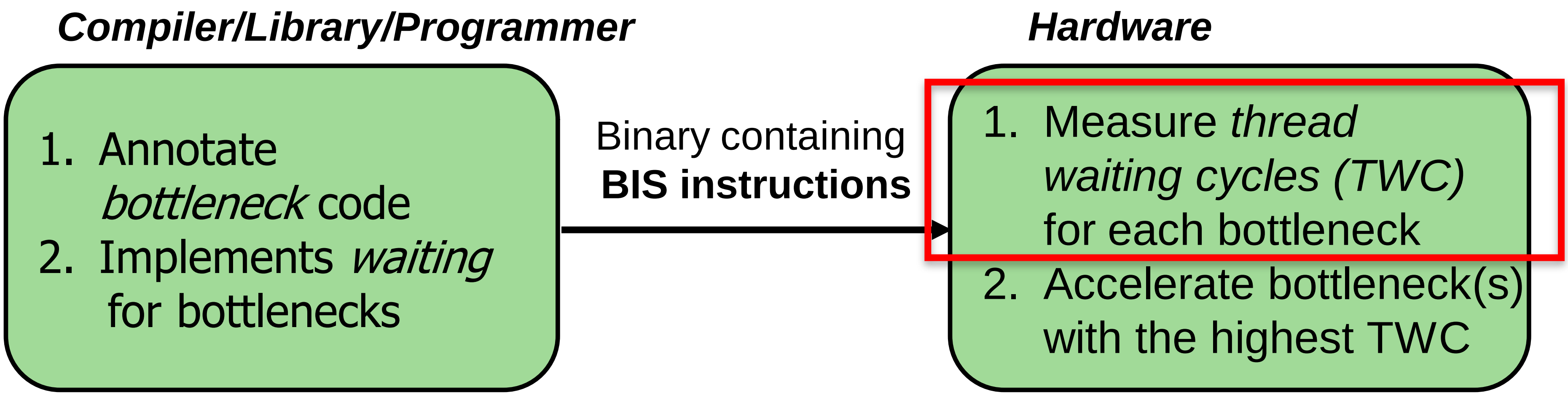
BIS: Hardware Overview

- 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

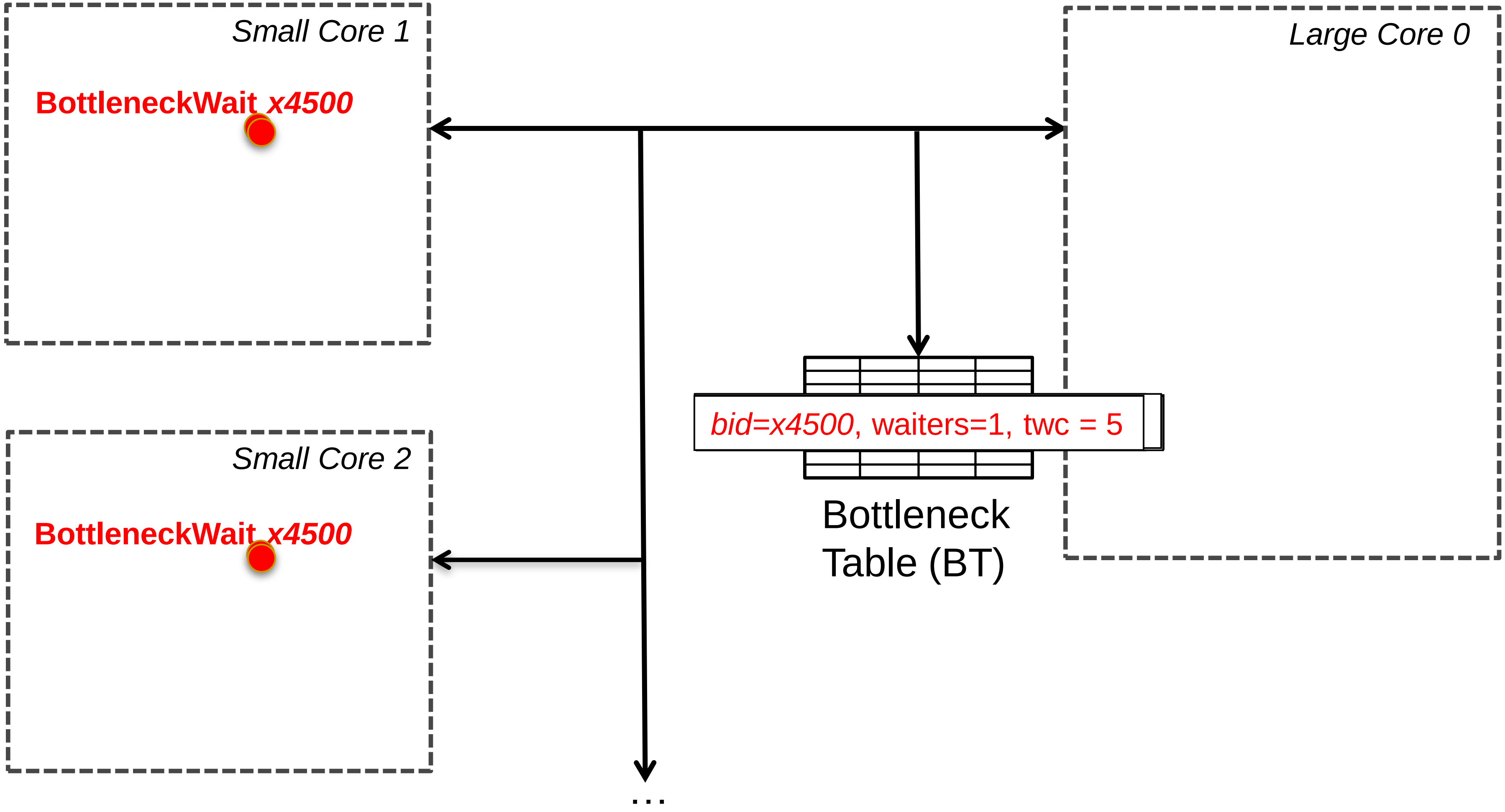
1. Annotate *bottleneck* code
2. Implements *waiting* for bottlenecks

Binary containing **BIS instructions**

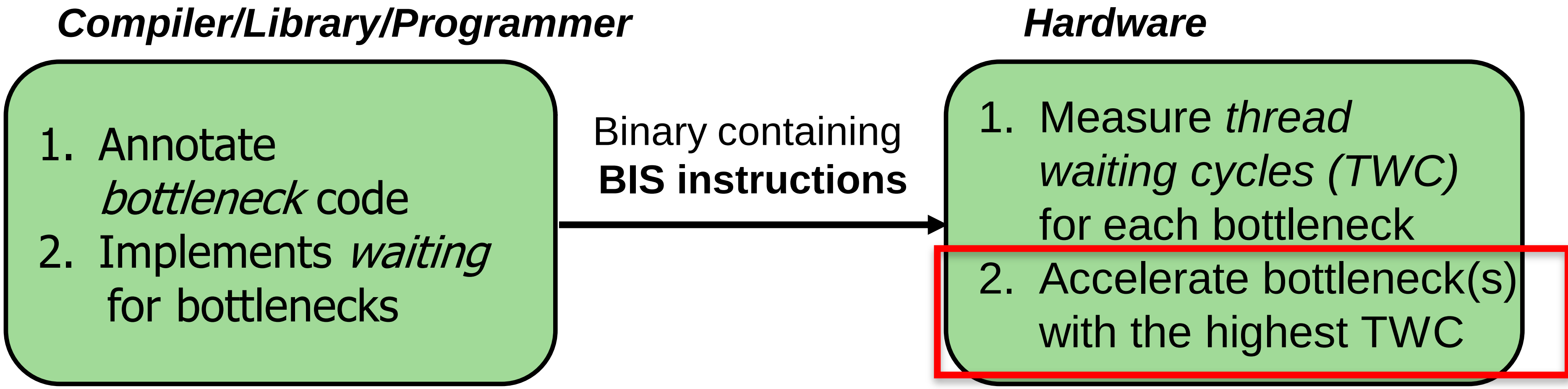
Hardware

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)



Compiler/Library/Programmer

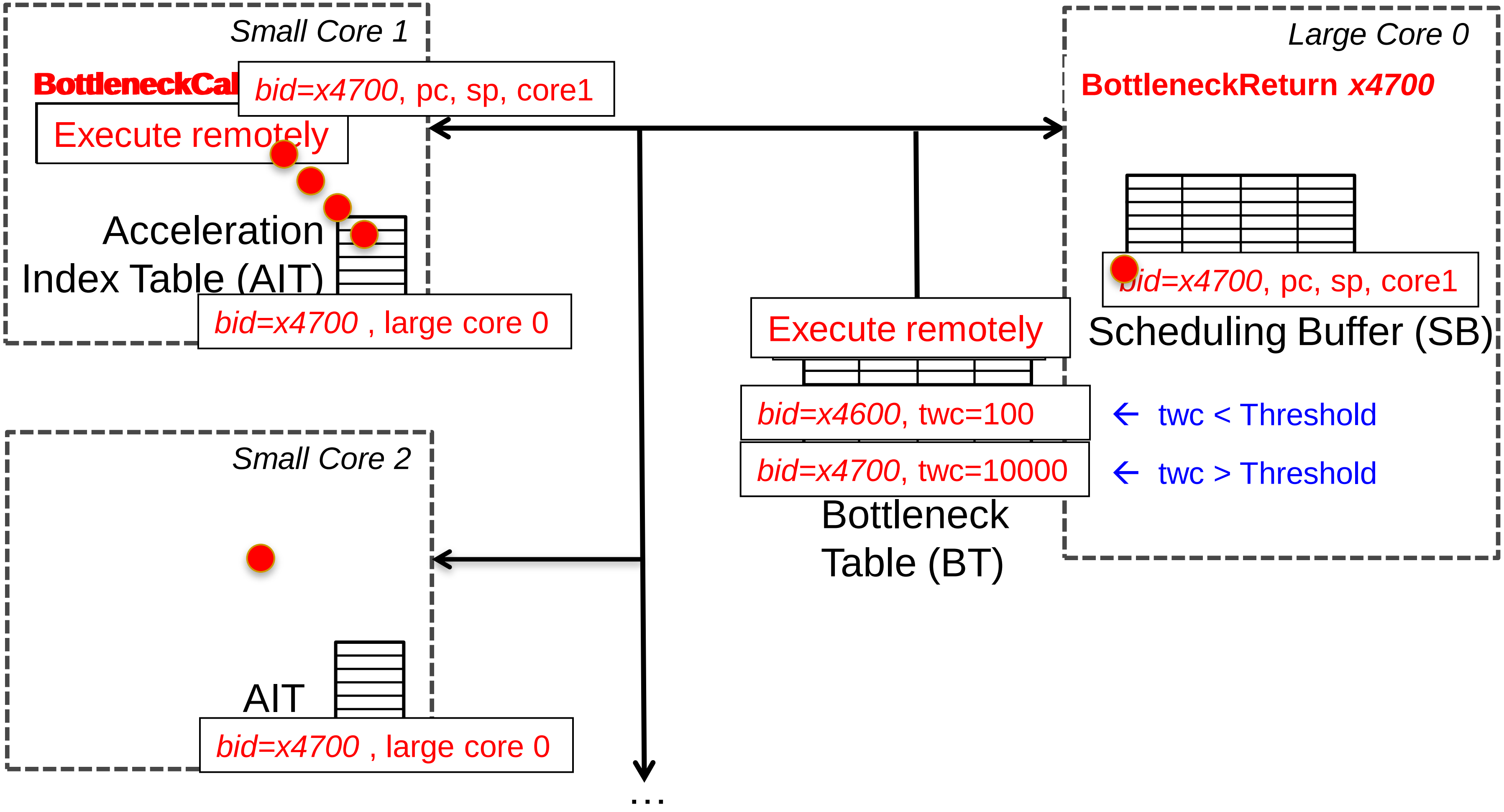
1. Annotate *bottleneck* code
2. Implements *waiting* for bottlenecks

Binary containing **BIS instructions**

Hardware

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

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

- **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)

Methodology

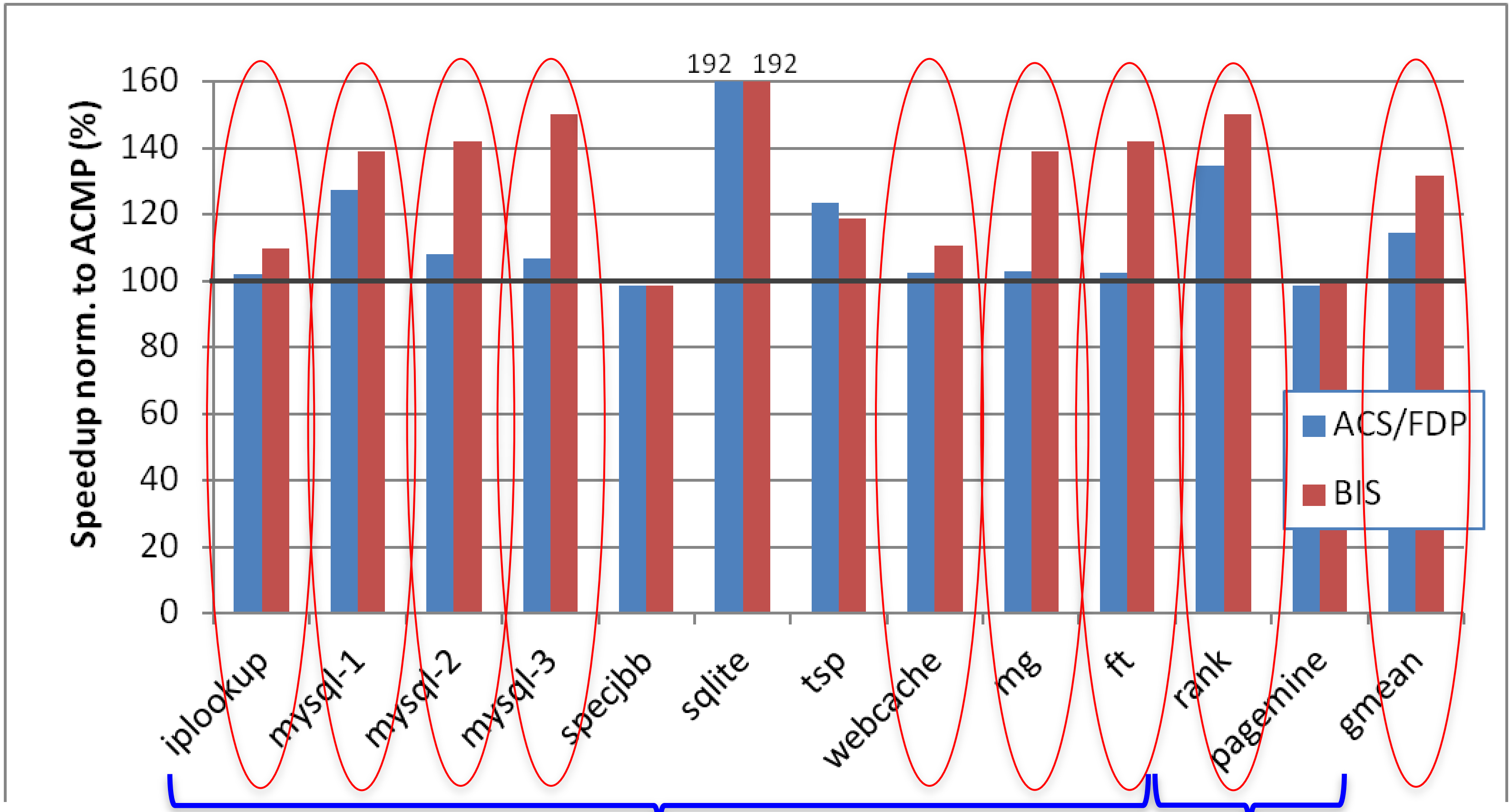
- Workloads: 8 critical section intensive, 2 barrier intensive and 2 pipeline-parallel applications
  - Data mining kernels, scientific, database, web, networking, specjbb
- Cycle-level multi-core x86 simulator
  - 8 to 64 small-core-equivalent area, 0 to 3 large cores, SMT
  - 1 large core is area-equivalent 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

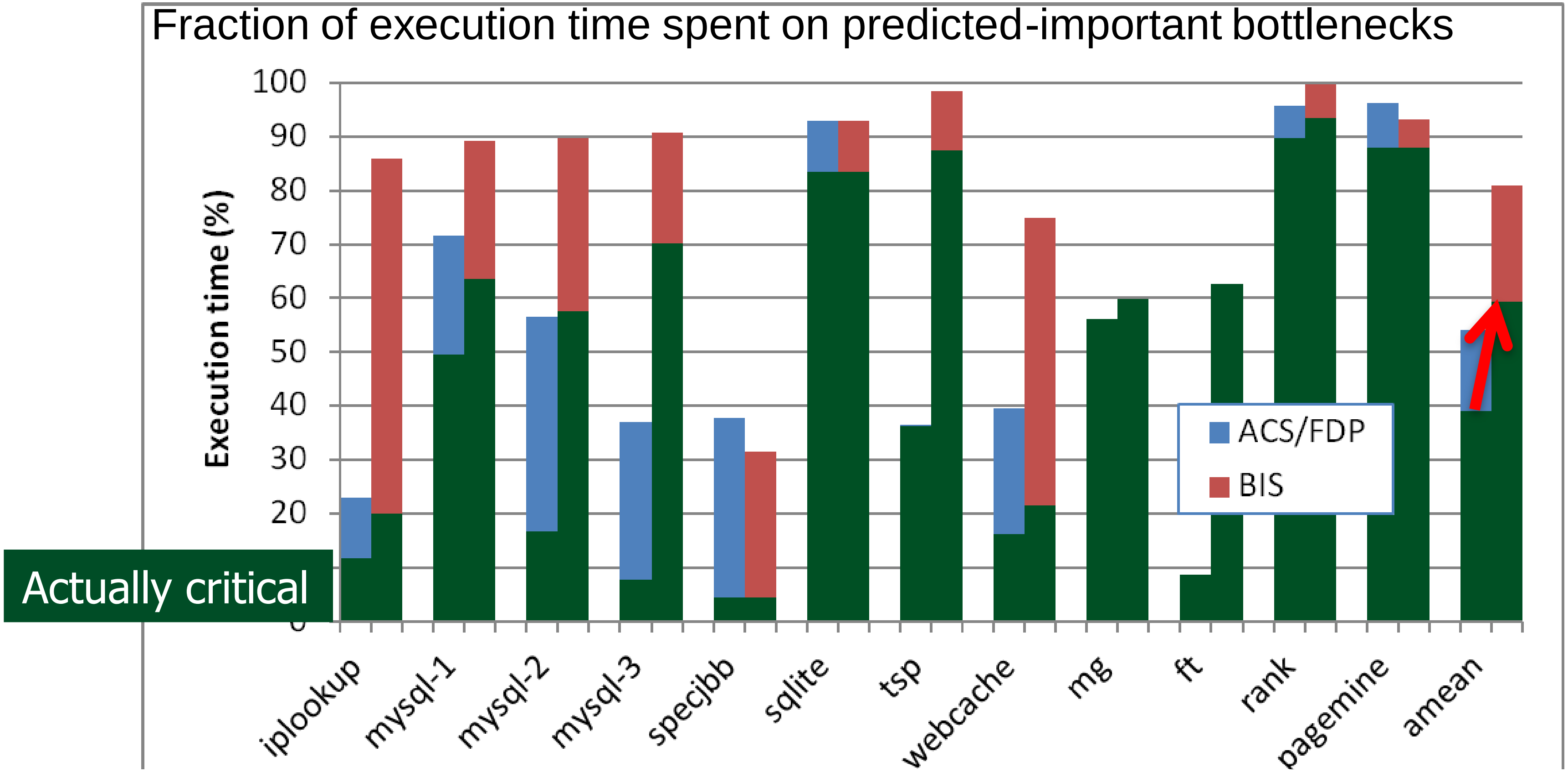
barriers, which ACS cannot accelerate

FDP

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

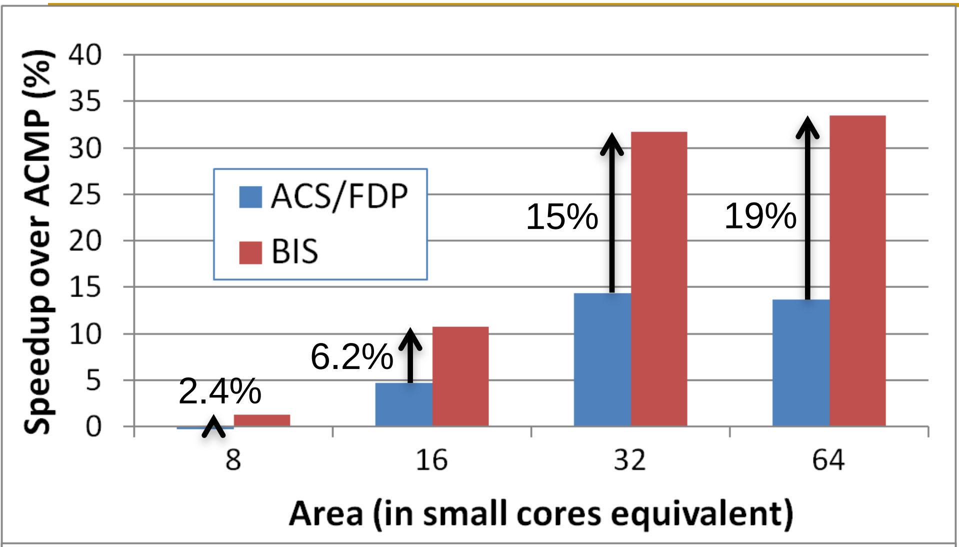
SAFARI

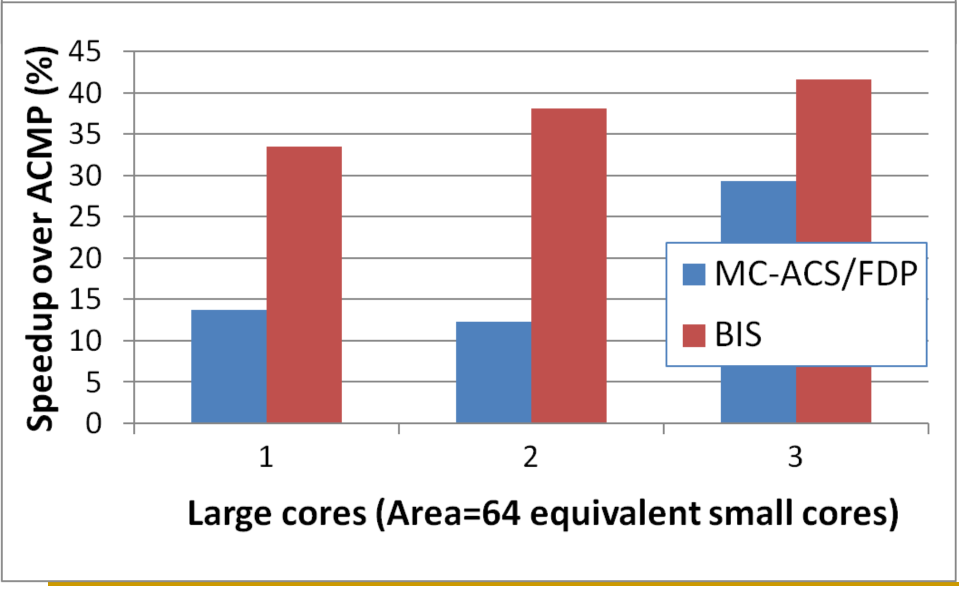
Why Does BIS Work?



- 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

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

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)

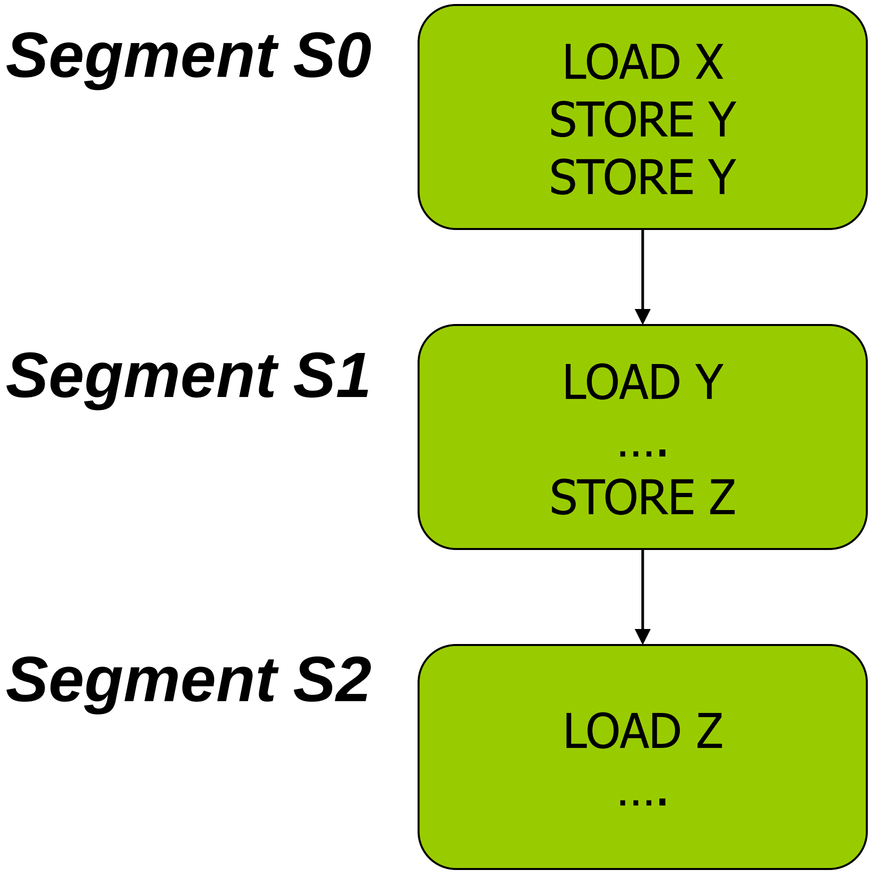
```
LOAD X
STORE Y
STORE Y

LOAD Y
….
STORE Z

LOAD Z
….
```

Staged Execution Model (III)

Split code into segments



Segment S0

LOAD X
STORE Y
STORE Y

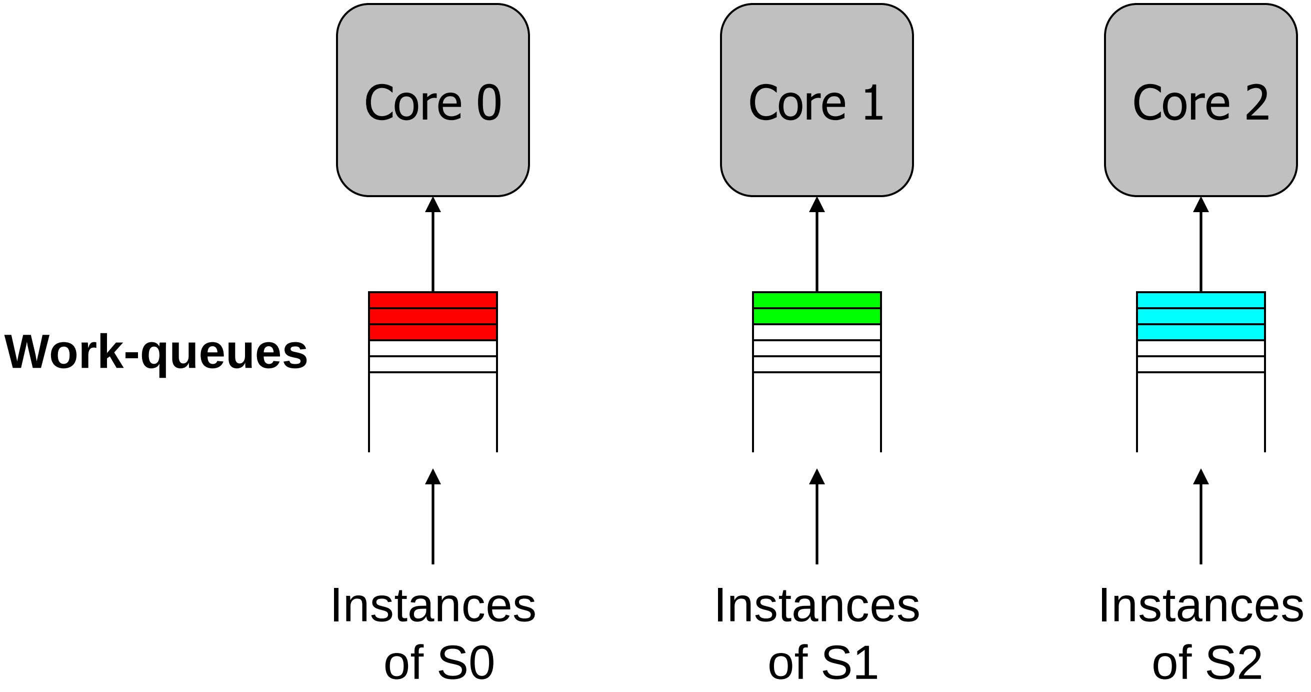
Segment S1

LOAD Y
....
STORE Z

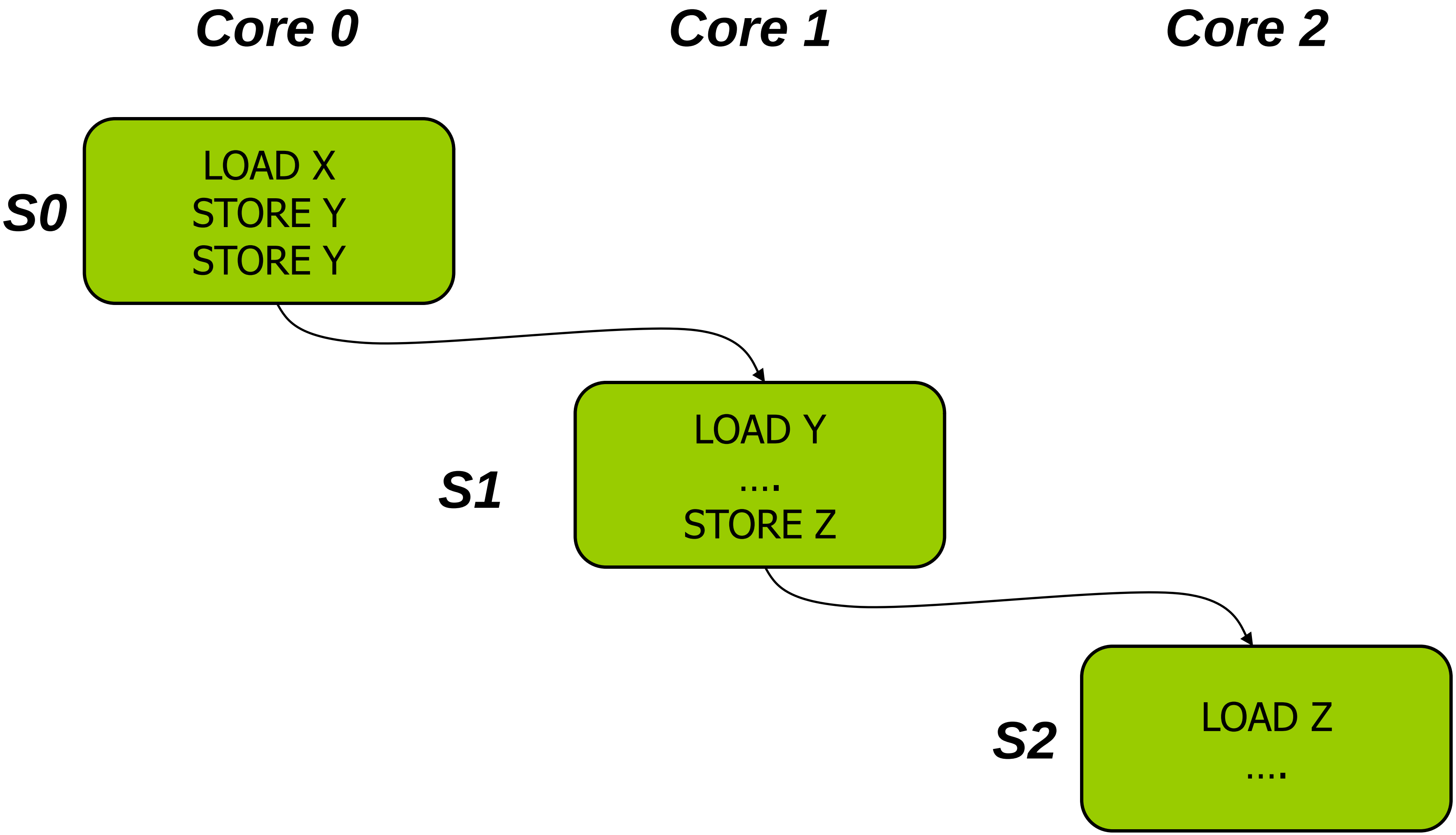
Segment S2

LOAD Z
....

Staged Execution Model (IV)



Staged Execution Model: Segment Spawning



Core 0

Core 1

Core 2

S0

LOAD X
STORE Y
STORE Y

S1

LOAD Y
....
STORE Z

S2

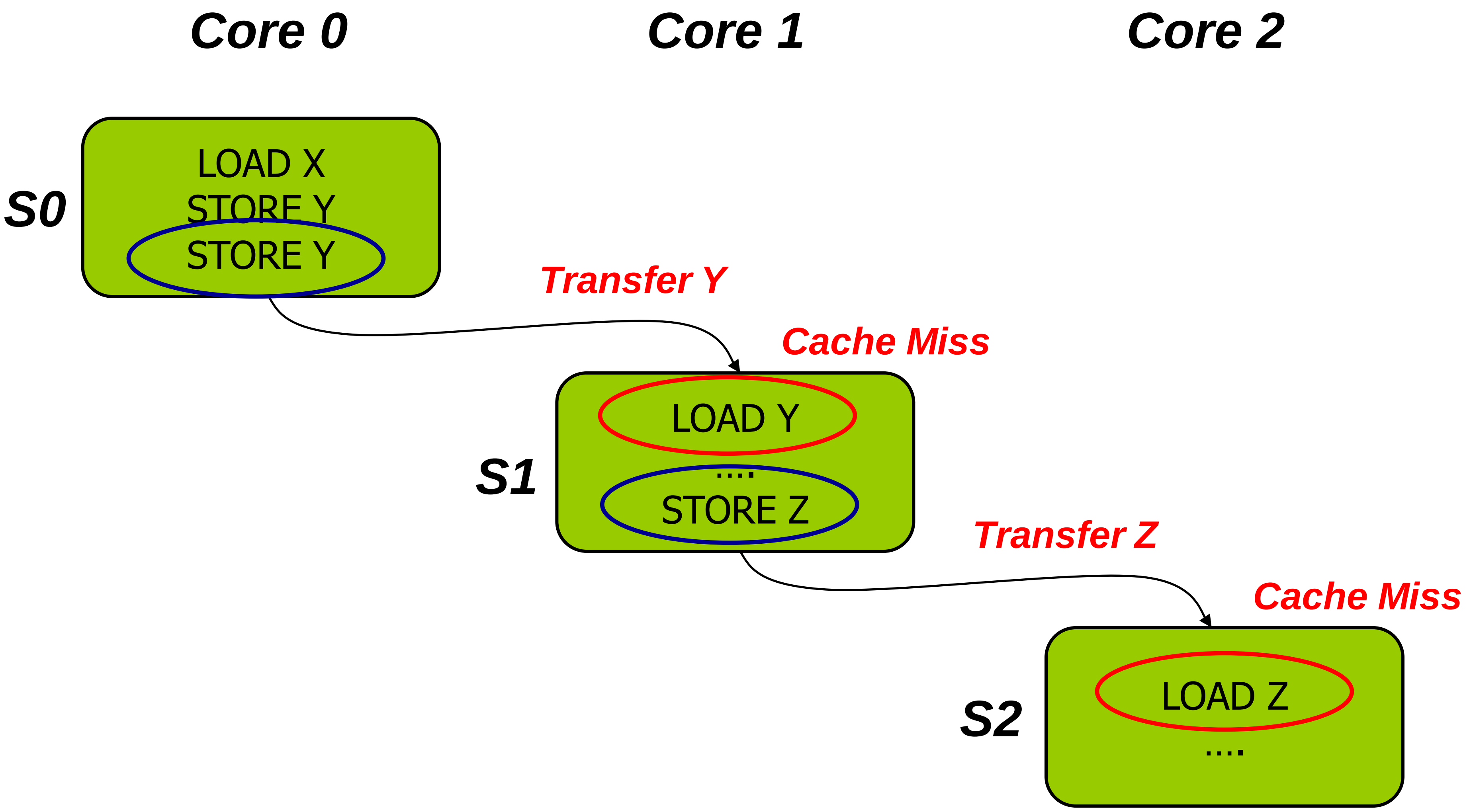
LOAD Z
....

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



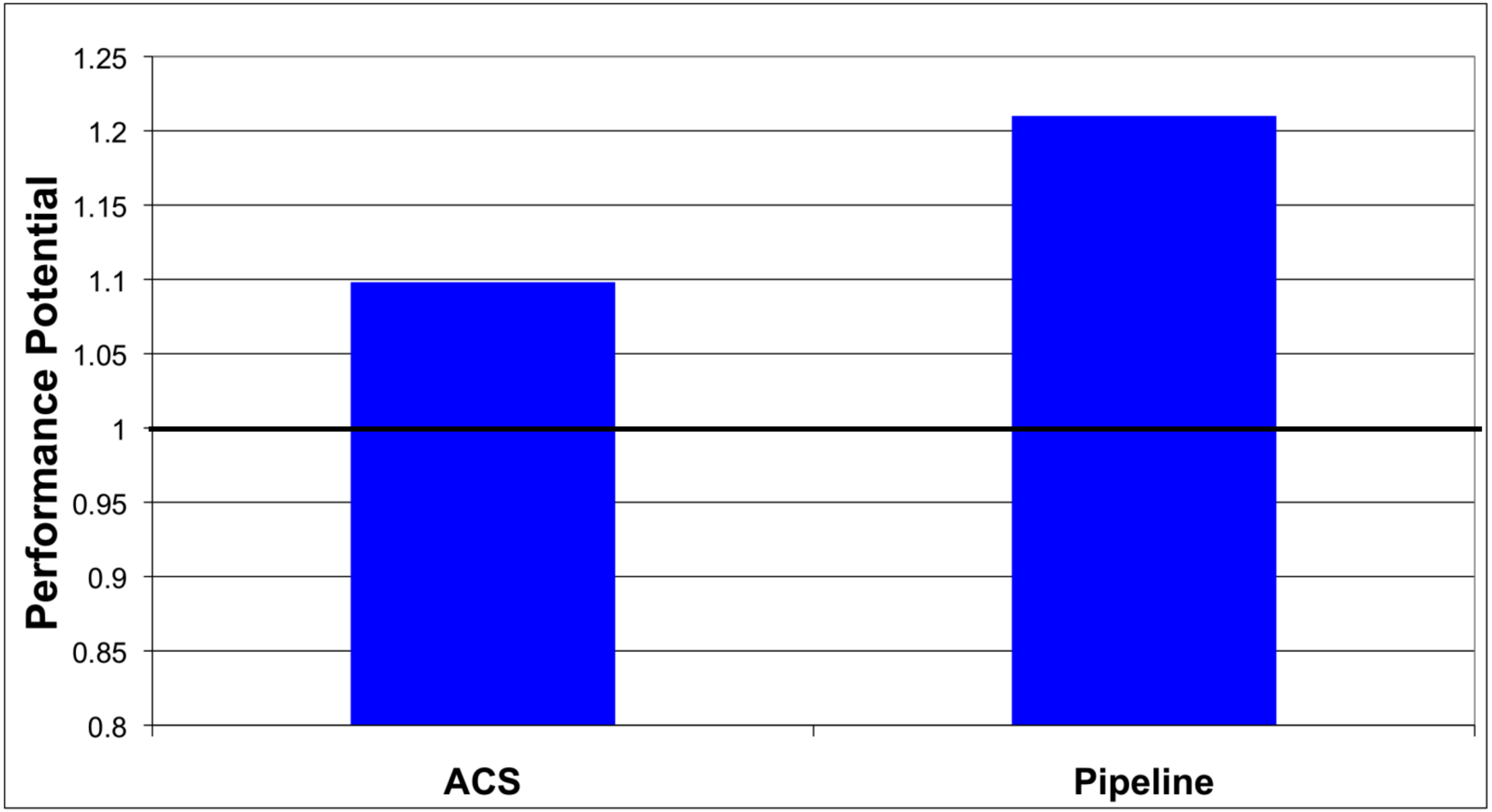
Problem: Locality of Inter-segment Data

- Accelerated Critical Sections [Suleman et al., ASPLOS 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

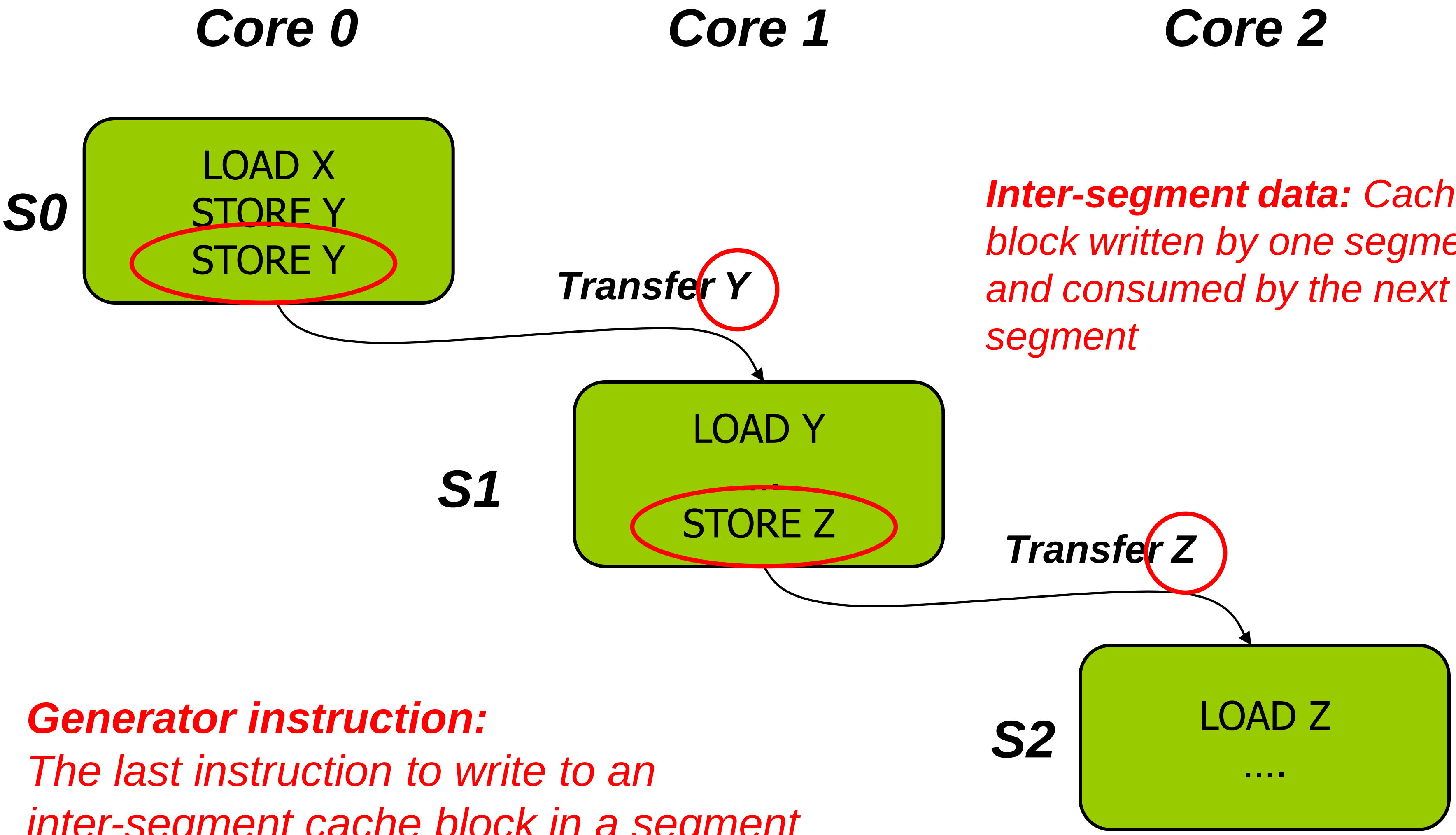
What if We Eliminated All Inter-segment Misses?



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



Inter-segment data: *Cache block written by one segment and consumed by the next segment*

Generator instruction:
The last instruction to write to an inter-segment cache block in a segment

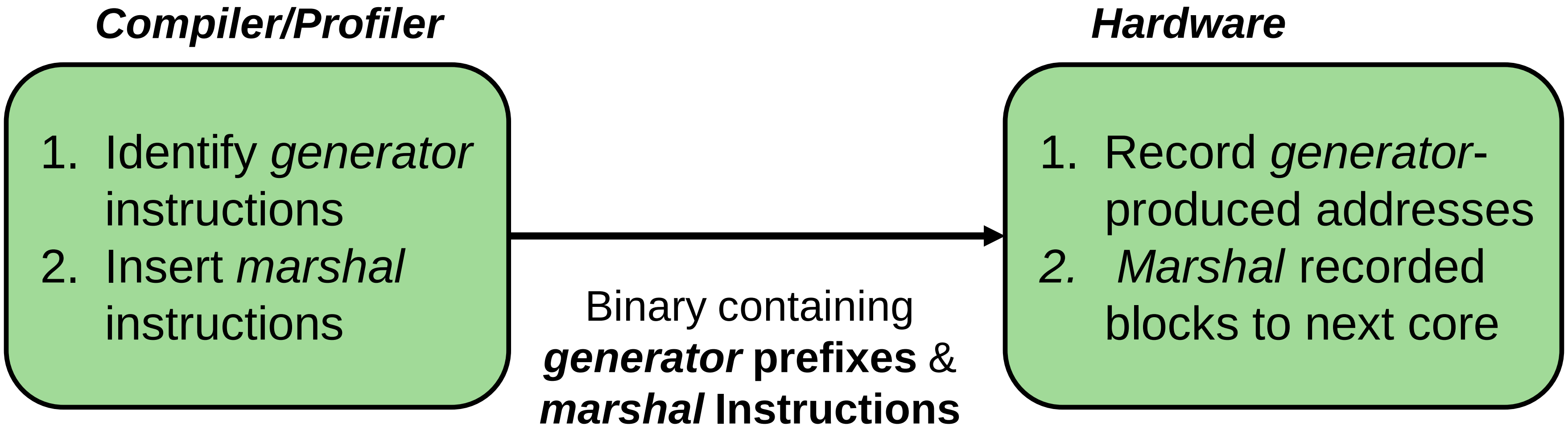
Key Observation and Idea

- 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



Compiler/Profiler

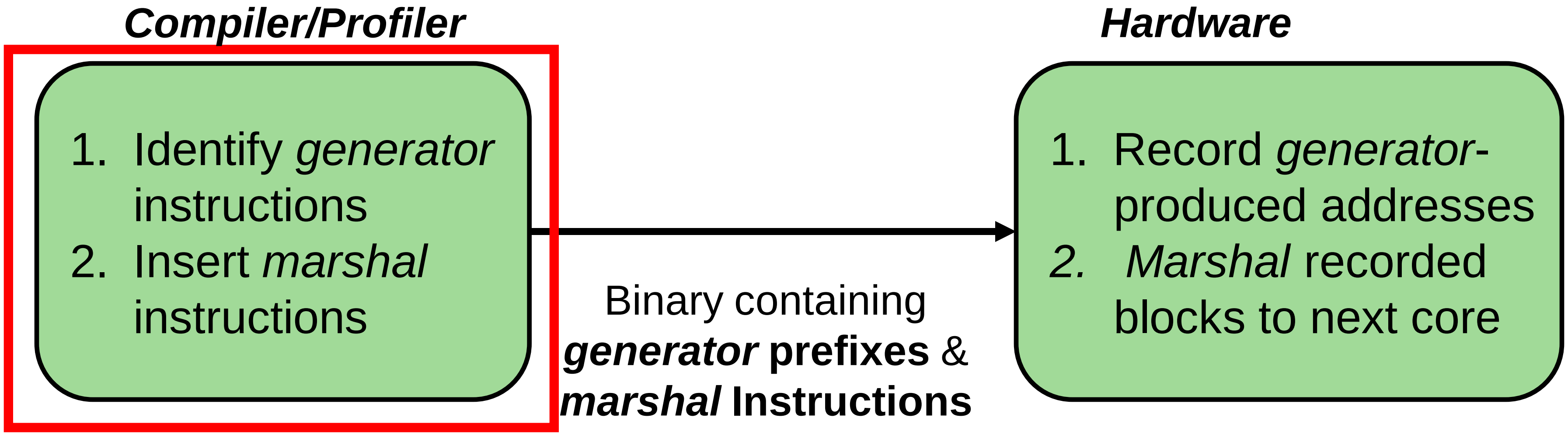
1. Identify *generator* instructions
2. Insert *marshal* instructions

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

Hardware

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

Data Marshaling



Compiler/Profiler

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

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

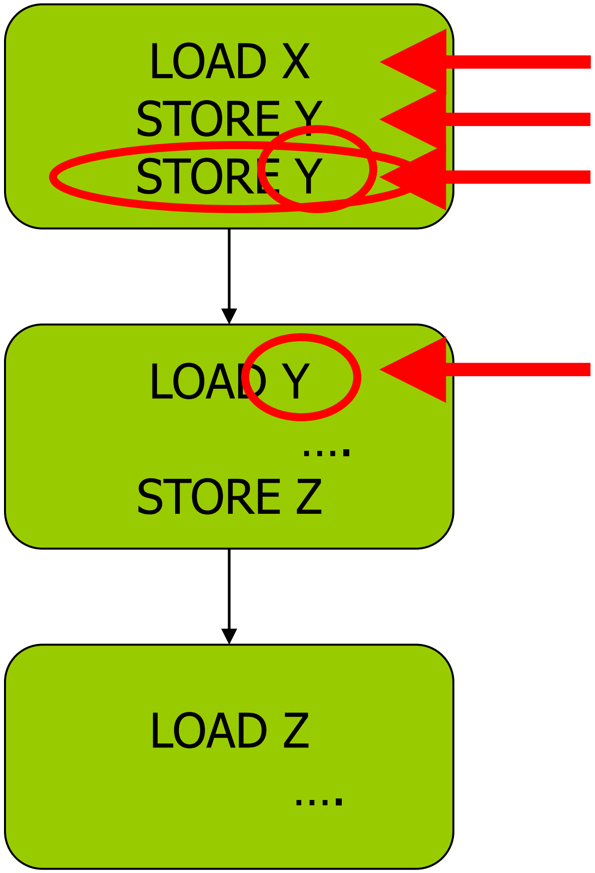
Hardware

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

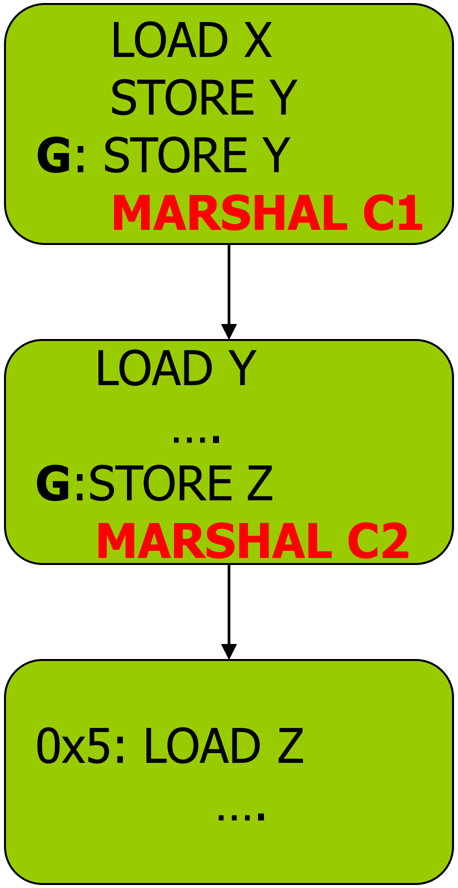
Profiling Algorithm

Inter-segment data

Mark as Generator Instruction



Marshal Instructions



LOAD X
STORE Y
G: STORE Y
MARSHAL C1

LOAD Y
....
G:STORE Z
MARSHAL C2

0x5: LOAD Z
....

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

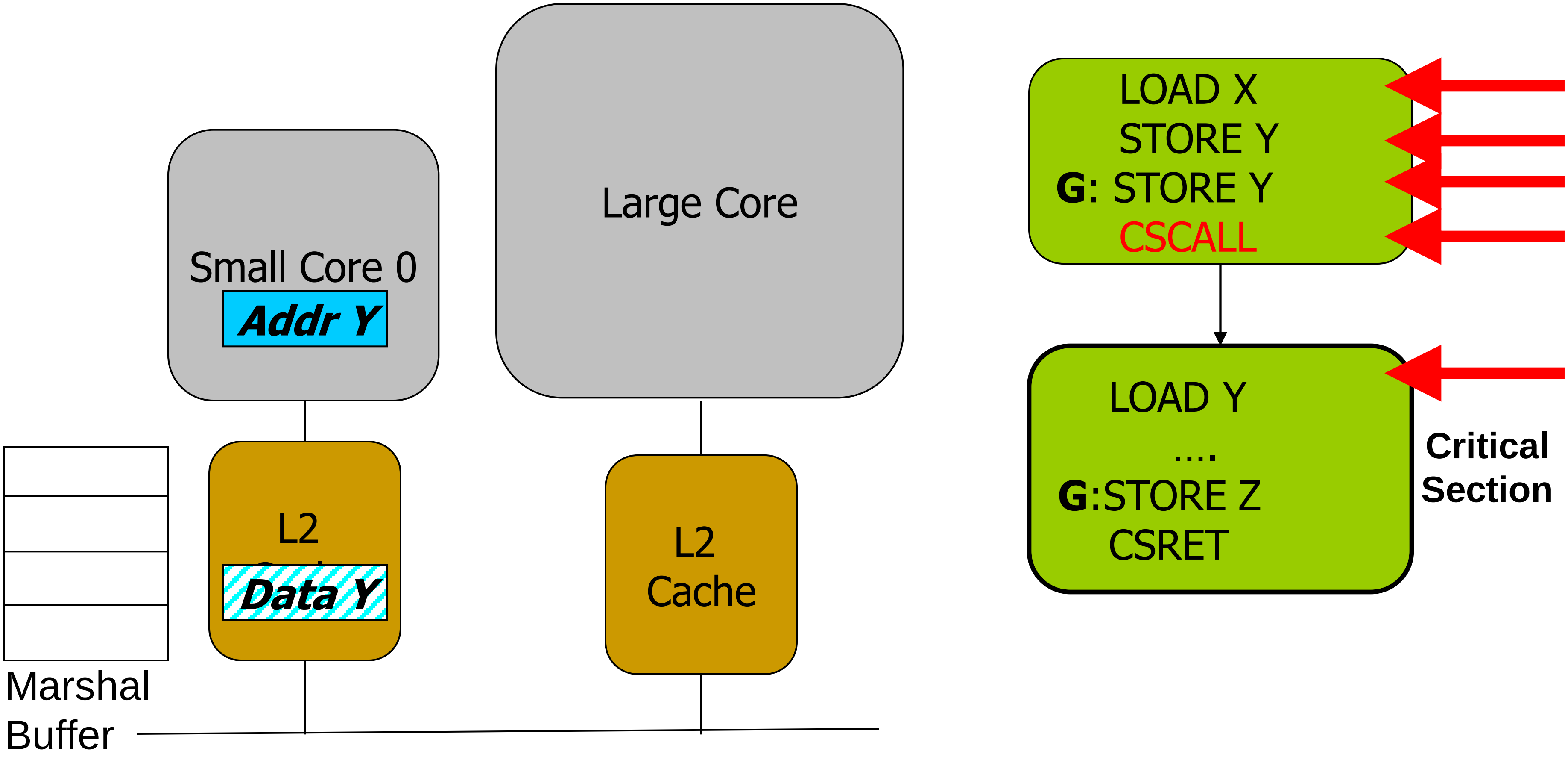
- Hardware
  - Marshal Buffer
    - Stores physical addresses of cache blocks to be marshaled
    - 16 entries enough for almost all workloads → 96 bytes per core
  - Ability to execute generator prefixes and marshal instructions
  - Ability to push data to another cache

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


Small Core 0

Addr Y

L2

Data Y

Marshal Buffer

Large Core

L2 Cache

LOAD X
STORE Y
G: STORE Y
CSCALL

LOAD Y
....
G:STORE Z
CSRET

Critical Section

Cache Hit!

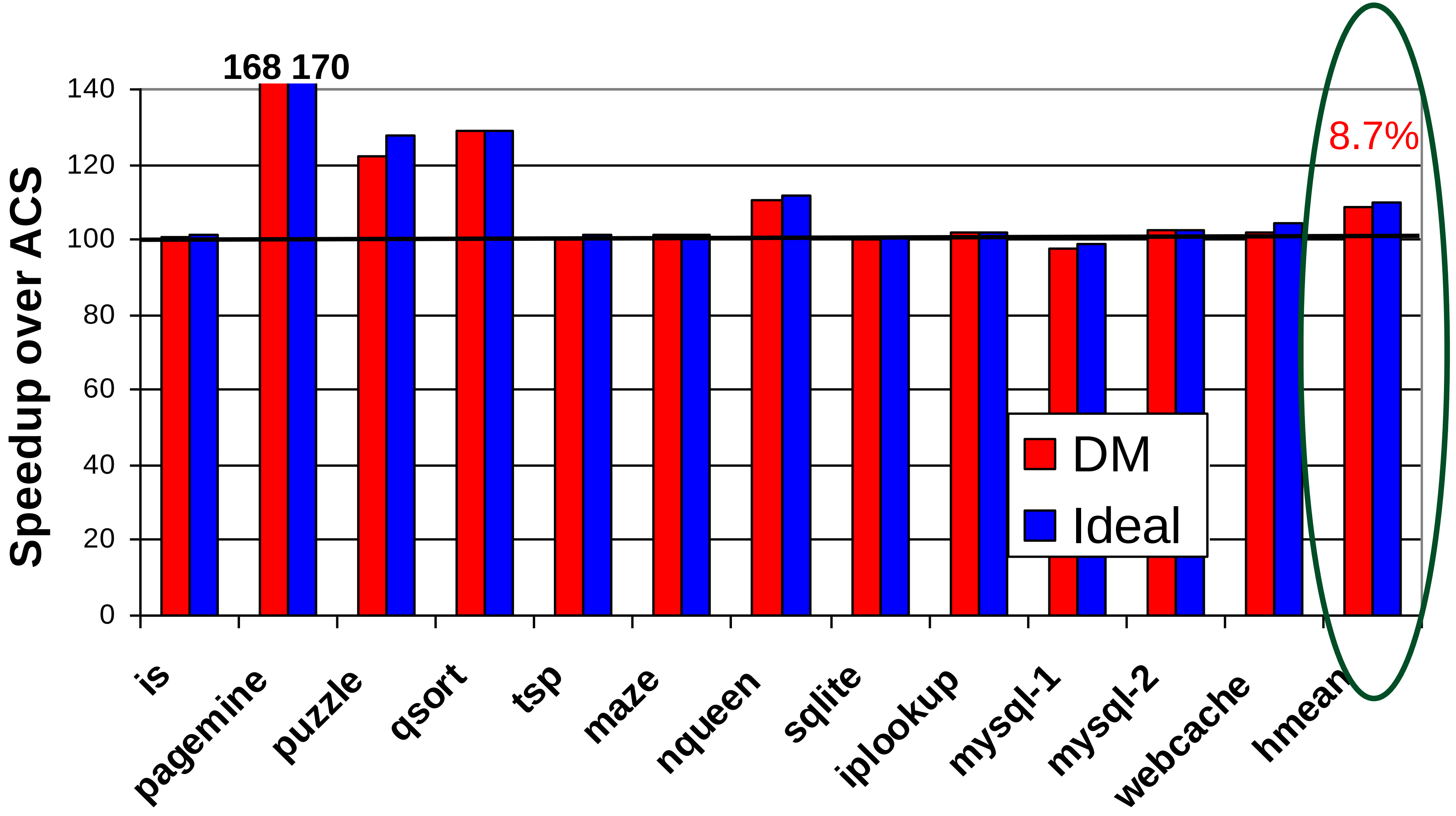
Accelerated Critical Sections: Methodology

- Workloads: 12 critical section intensive applications
  - Data mining kernels, sorting, database, web, networking
  - Different training and simulation input sets

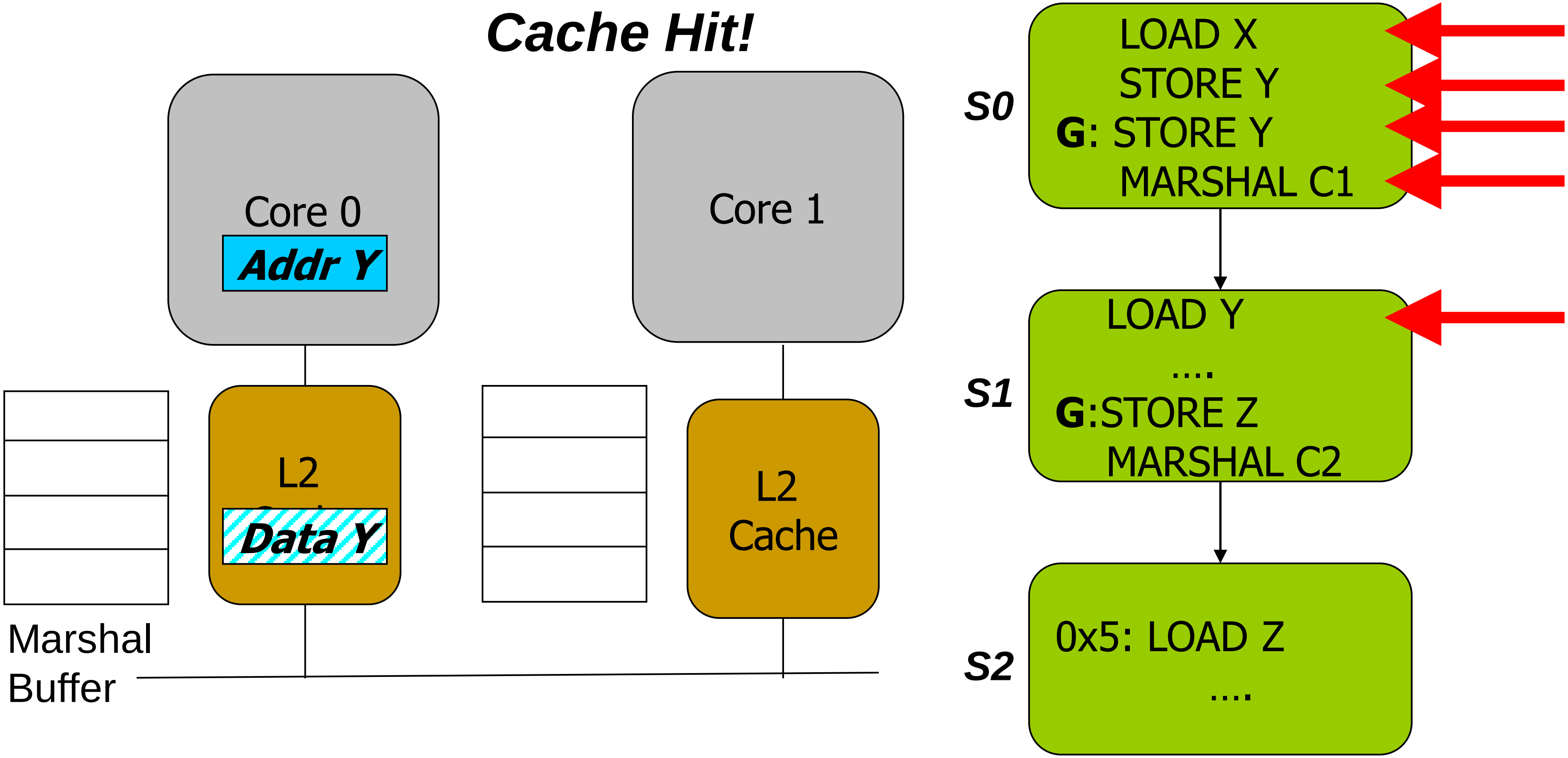
- Multi-core x86 simulator
  - 1 large and 28 small cores
  - Aggressive stream prefetcher employed at each core

- 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!

Core 0

Addr Y

Core 1

L2

Data Y

L2
Cache

Marshal
Buffer

S0

LOAD X
STORE Y
G: STORE Y
MARSHAL C1

S1

LOAD Y
....
G:STORE Z
MARSHAL C2

S2

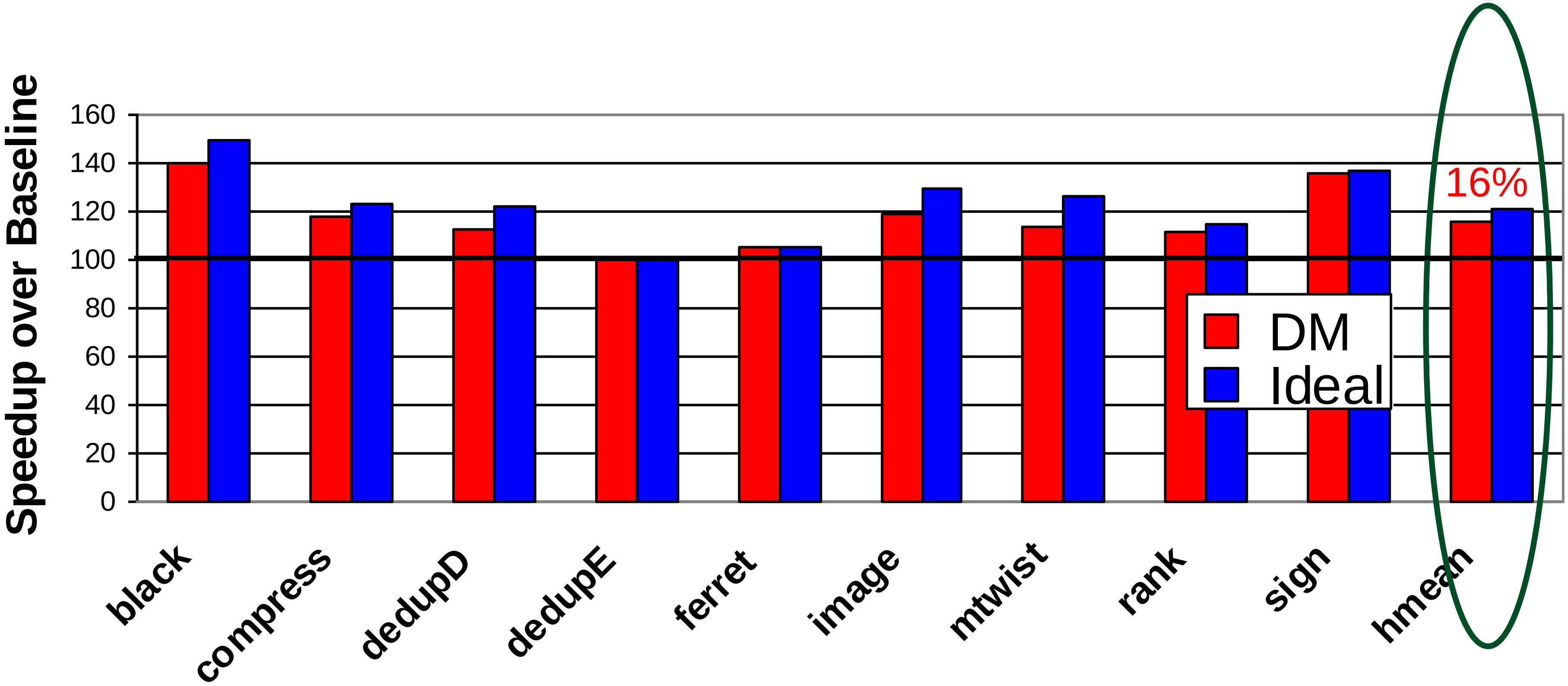
0x5: LOAD Z
....

Pipeline Parallelism: Methodology

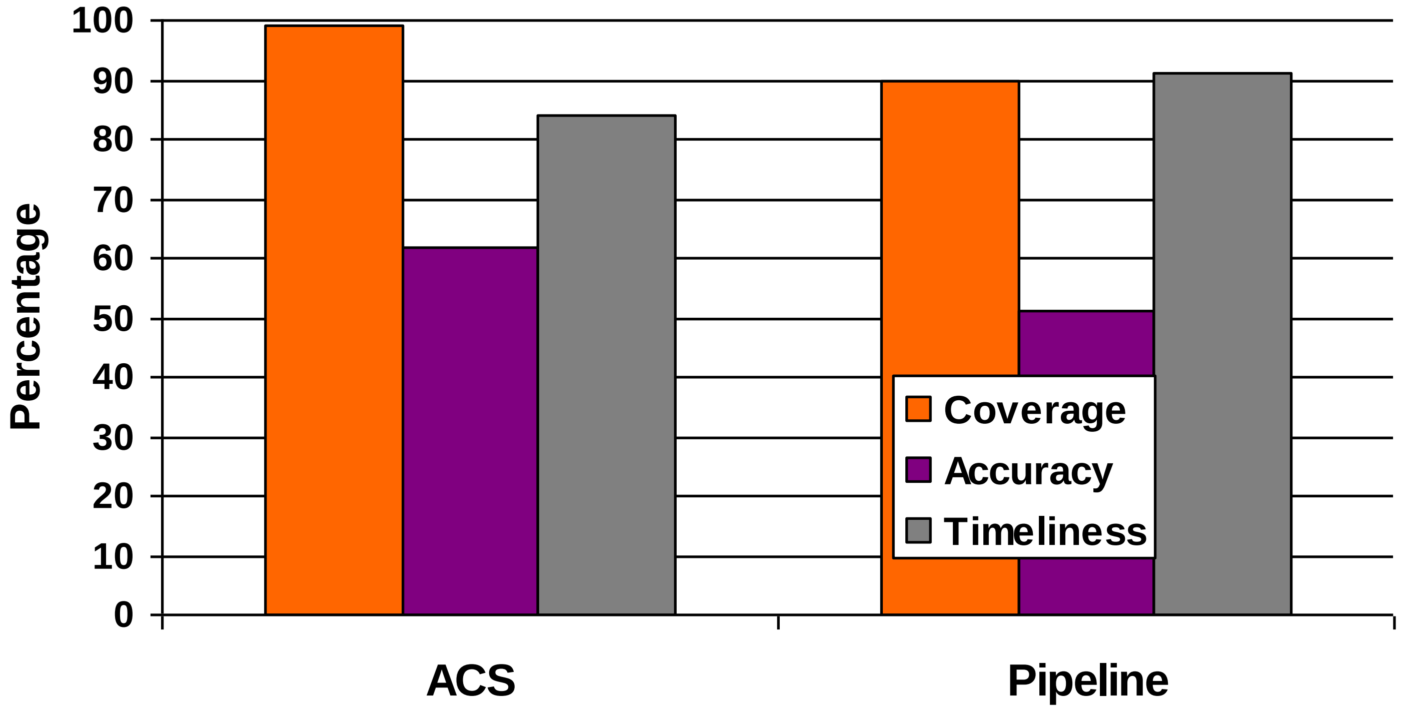
- Workloads: 9 applications with pipeline parallelism
  - Financial, compression, multimedia, encoding/decoding
  - Different training and simulation input sets

- 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

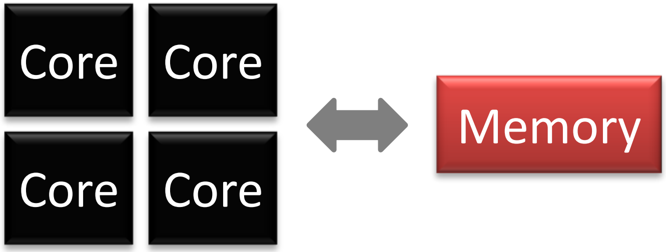
- Applicable to several existing Staged Execution models
  - Accelerated Critical Sections: 9% performance benefit
  - Pipeline Parallelism: 16% performance benefit
- Can enable new models→ very fine-grained remote execution

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

Motivation

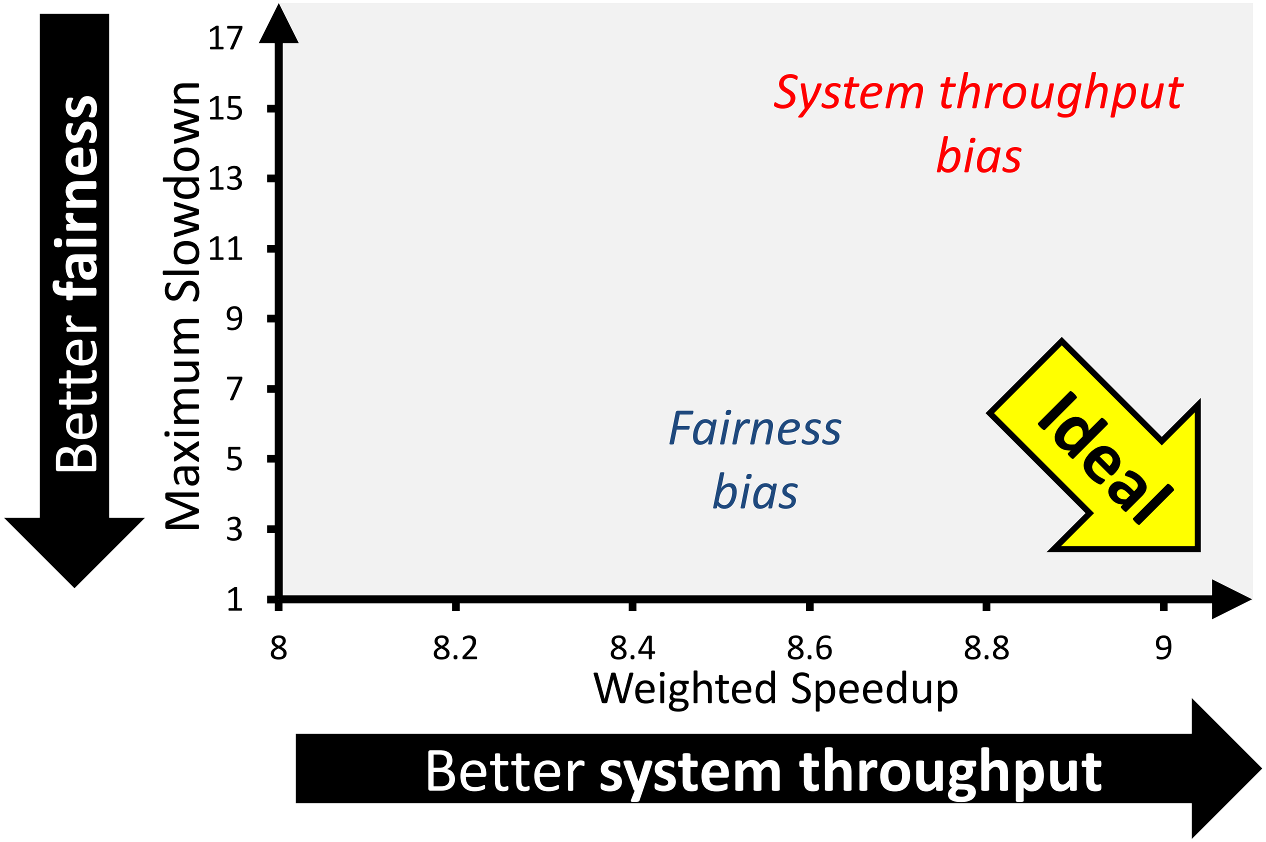
- Memory is a shared resource



- 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?

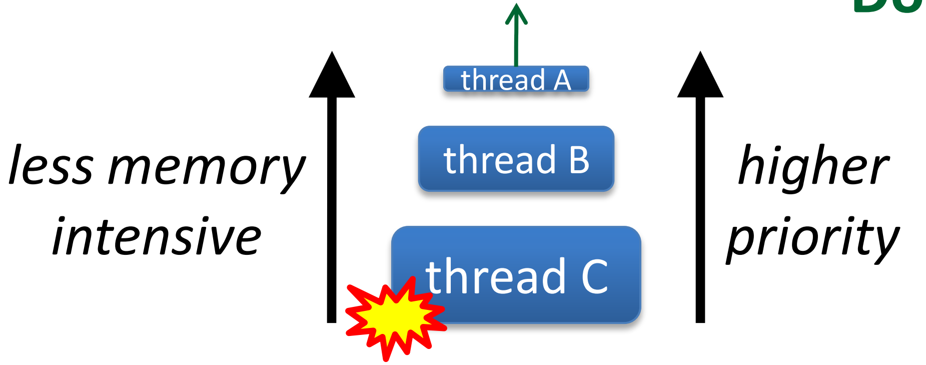
***Throughput biased** approach*

Prioritize less memory-intensive threads

***Fairness biased** approach*

Take turns accessing memory

Good for throughput

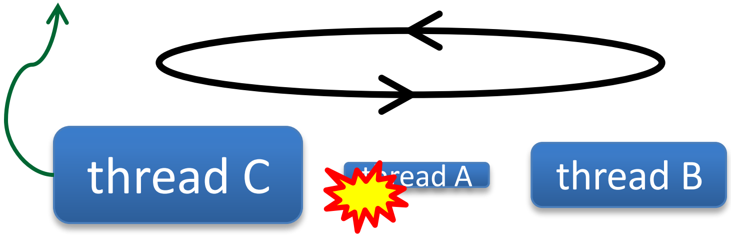


less memory intensive

higher priority

starvation → unfairness

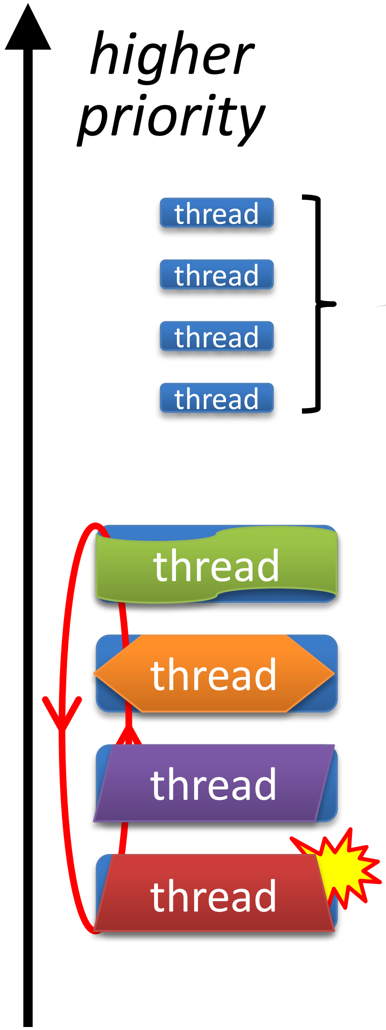
Does not starve



not prioritized → reduced throughput

Single policy for all threads is insufficient

Insight: Achieving Best of Both Worlds



For Throughput

Prioritize memory-non-intensive threads

For Fairness

Unfairness caused by memory-intensive being prioritized over each other

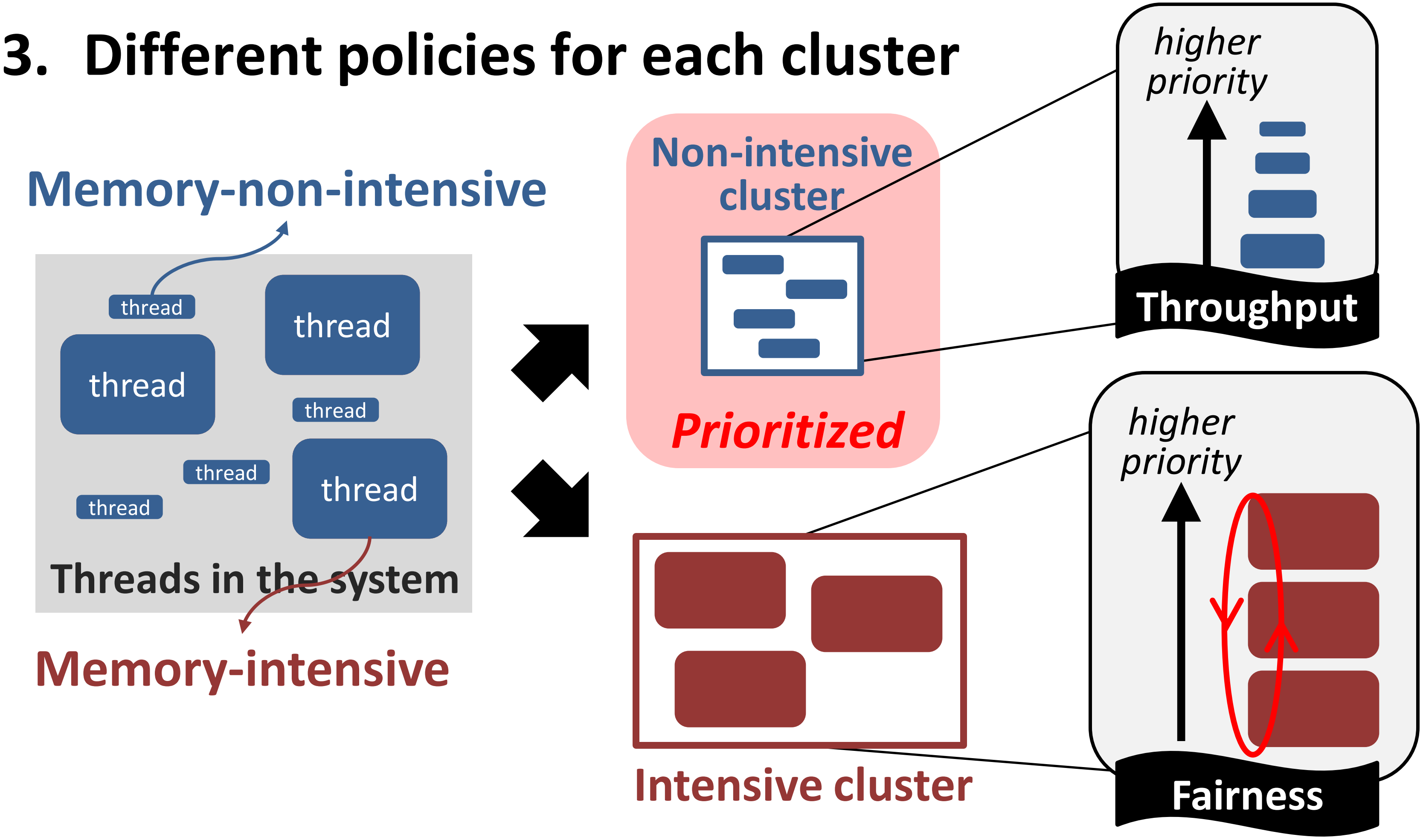
- Shuffle threads

Memory-intensive threads have different vulnerability to interference

- Shuffle asymmetrically

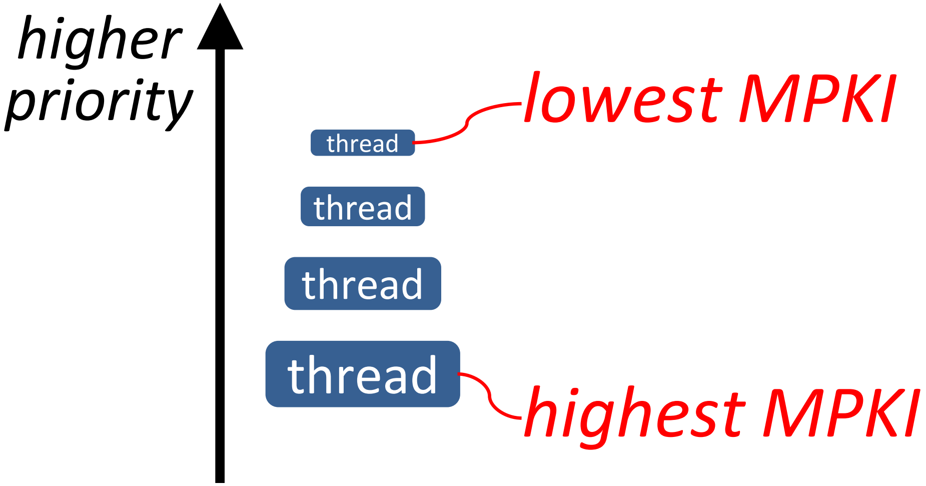
Overview: Thread Cluster Memory Scheduling

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



Non-Intensive Cluster

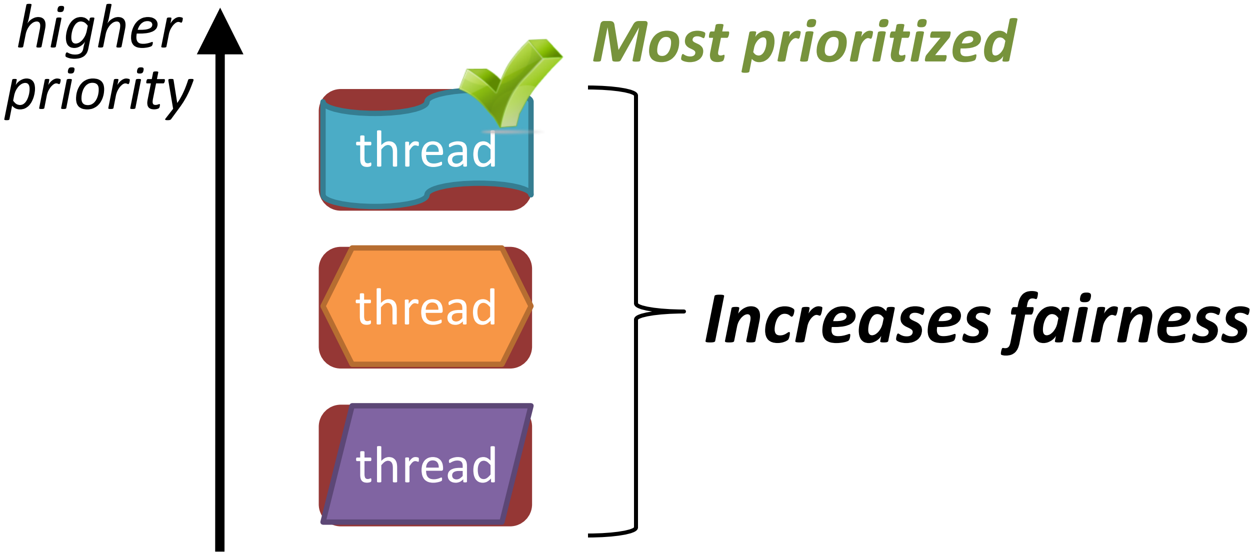
Prioritize threads according to MPKI



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

Intensive Cluster

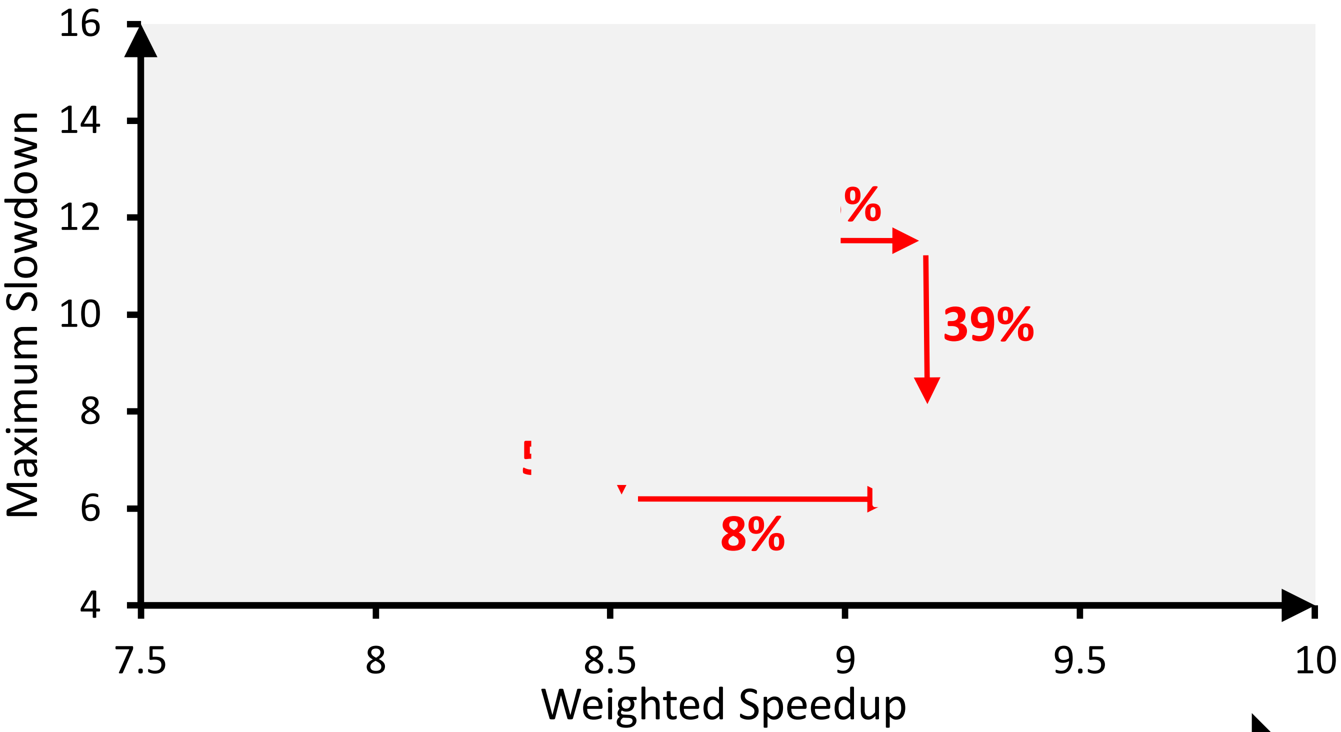
Periodically shuffle the priority of threads



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

Results: Fairness vs. Throughput

Averaged over 96 workloads



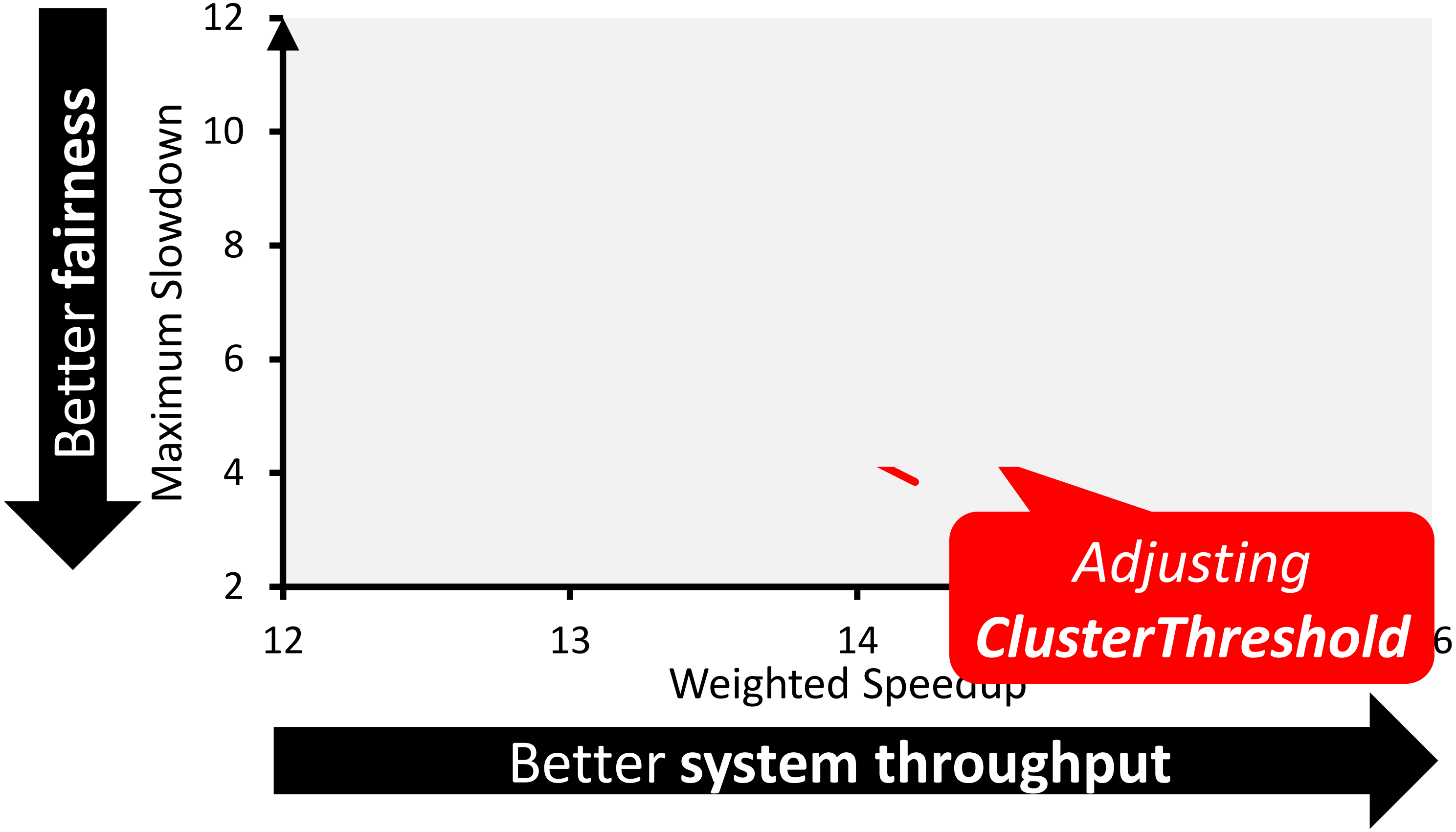
Better fairness

Better **system throughput**

TCM provides best fairness and system throughput

Results: Fairness-Throughput Tradeoff

When configuration parameter is varied…



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.

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

Related Ongoing/Future Work

- Dynamically asymmetric cores
- Memory system design for asymmetric cores
- Asymmetric memory systems
  - Phase Change Memory (or Technology X) + DRAM
  - Hierarchies optimized for different access patterns
- Asymmetric on-chip interconnects
  - Interconnects optimized for different application requirements
- Asymmetric resource management algorithms
  - E.g., network congestion control
- Interaction of multiprogrammed multithreaded workloads

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

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

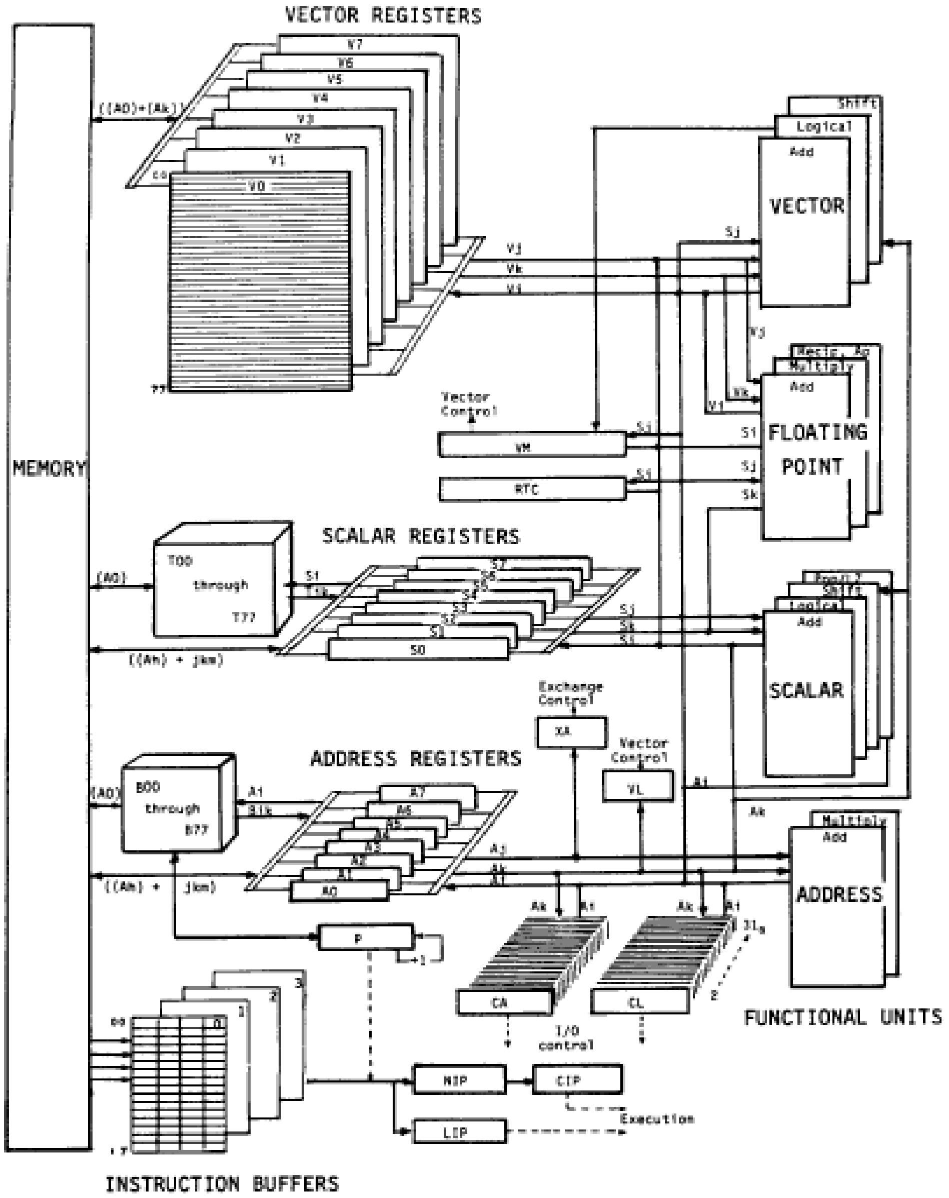
- Asymmetry in design enables tradeoffs: can get the best of all worlds
  - Asymmetry in core microarch. → Accelerated Critical Sections, BIS, DM → Good parallel performance + Good serialized performance
  - Asymmetry in memory scheduling → Thread Cluster Memory Scheduling → Good throughput + good fairness

- Simple asymmetric designs can be effective and low-cost

740: Computer Architecture
Architecting and Exploiting Asymmetry (in Multi-Core Architectures)

Prof. Onur Mutlu
Carnegie Mellon University

Vector Machine Organization (CRAY-1)



- CRAY-1
- Russell, "The CRAY-1 computer system," CACM 1978.

- Scalar and vector modes
- 8 64-element vector registers
- 64 bits per element
- 16 memory banks
- 8 64-bit scalar registers
- 8 24-bit address registers

Identifying and Accelerating Resource Contention Bottlenecks

Thread Serialization

- Three fundamental causes

    1. Synchronization

    2. Load imbalance

    3. Resource contention

Memory Contention as a Bottleneck

- Problem:
  - Contended memory regions cause serialization of threads
  - Threads accessing such regions can form the critical path
  - Data-intensive workloads (MapReduce, GraphLab, Graph500) can be sped up by 1.5 to 4X by ideally removing contention

- Idea:
  - Identify contended regions dynamically
  - Prioritize caching the data from threads which are slowed down the most by such regions in faster DRAM/eDRAM

- Benefits:
  - Reduces contention, serialization, critical path

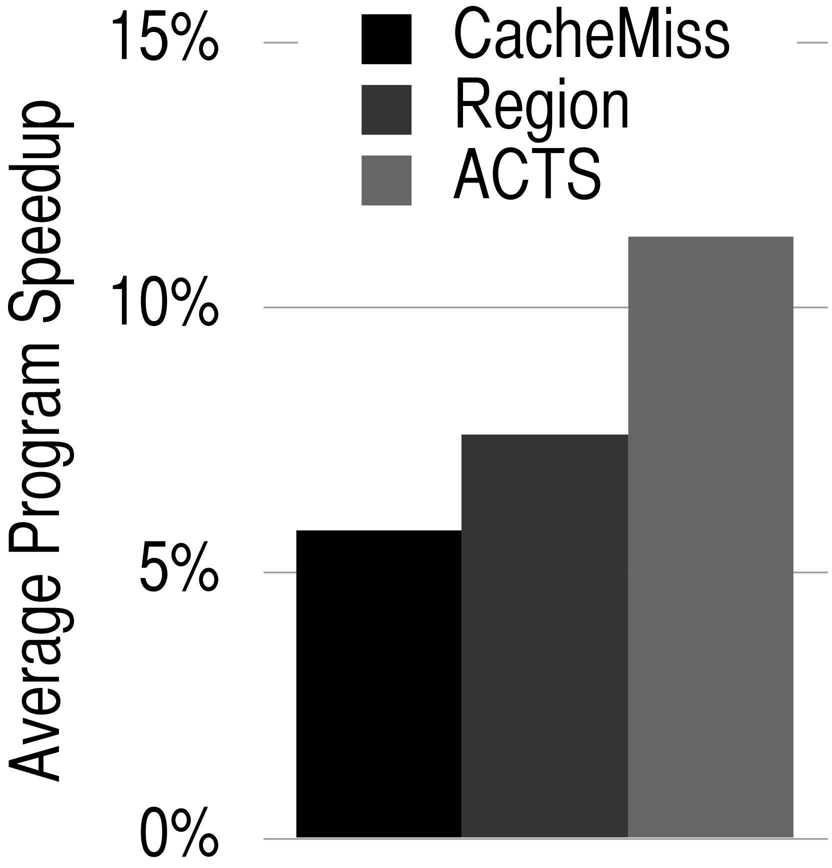
Evaluation

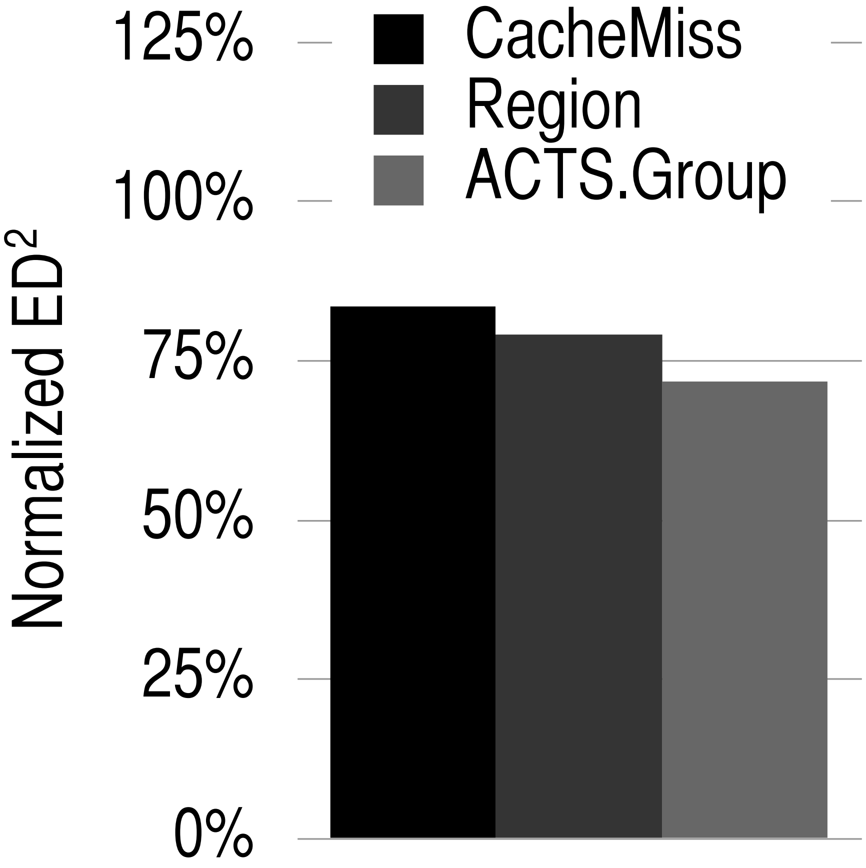
- Workloads: MapReduce, GraphLab, Graph500

- Cycle-level x86 platform simulator
  - **CPU**: 8 out-of-order cores, 32KB private L1, 512KB shared L2
  - **Hybrid Memory**: DDR3 1066 MT/s, 32MB DRAM, 8GB PCM

- Mechanisms
  - Baseline: DRAM as a conventional cache to PCM
  - CacheMiss: Prioritize caching data from threads with highest cache miss latency
  - Region: Cache data from most contended memory regions
  - ACTS: Prioritize caching data from threads most slowed down due to memory region contention

Caching Results





Partial List of Referenced/Related Papers

Heterogeneous Cores

- 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***), pages 253-264, Washington, DC, March 2009. Slides (ppt)

- 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. Slides (ppt)

- 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. Slides (ppt) (pdf)

- Jose A. Joao, M. Aater Suleman, Onur Mutlu, and Yale N. Patt,
  "Utility-Based Acceleration of Multithreaded Applications on Asymmetric CMPs"
  Proceedings of the 40th International Symposium on Computer Architecture (***ISCA***), Tel-Aviv, Israel, June 2013. Slides (ppt) Slides (pdf)

QoS-Aware Memory Systems (I)

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

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

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

- Eiman Ebrahimi, Chang Joo Lee, Onur Mutlu, and Yale N. Patt,
"Fairness via Source Throttling: A Configurable and High-Performance Fairness Substrate for Multi-Core Memory Systems"
ACM Transactions on Computer Systems (***TOCS***), April 2012.

QoS-Aware Memory Systems (II)

- Onur Mutlu and Thomas Moscibroda,
"Parallelism-Aware Batch Scheduling: Enabling High-Performance and Fair Memory Controllers"
*IEEE Micro, Special Issue: Micro's Top Picks from 2008 Computer Architecture Conferences (**MICRO TOP PICKS**)*, Vol. 29, No. 1, pages 22-32, January/February 2009.

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

- Thomas Moscibroda and Onur Mutlu,
"Memory Performance Attacks: Denial of Memory Service in Multi-Core Systems"
*Proceedings of the 16th USENIX Security Symposium (**USENIX SECURITY**)*, pages 257-274, Boston, MA, August 2007. Slides (ppt)

QoS-Aware Memory Systems (III)

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

- Boris Grot, Joel Hestness, Stephen W. Keckler, and Onur Mutlu,
  "Kilo-NOC: A Heterogeneous Network-on-Chip Architecture for Scalability and Service Guarantees"
  Proceedings of the 38th International Symposium on Computer Architecture (***ISCA***), San Jose, CA, June 2011. Slides (pptx)

- Reetuparna Das, Onur Mutlu, Thomas Moscibroda, and Chita R. Das,
  "Application-Aware Prioritization Mechanisms for On-Chip Networks"
  Proceedings of the 42nd International Symposium on Microarchitecture (***MICRO***), pages 280-291, New York, NY, December 2009. Slides (pptx)

Heterogeneous Memory

- Justin Meza, Jichuan Chang, HanBin Yoon, Onur Mutlu, and Parthasarathy Ranganathan, **"Enabling Efficient and Scalable Hybrid Memories Using Fine-Granularity DRAM Cache Management"** *IEEE Computer Architecture Letters* (***CAL***), May 2012.

- HanBin Yoon, Justin Meza, Rachata Ausavarungnirun, Rachael Harding, and Onur Mutlu, **"Row Buffer Locality-Aware Data Placement in Hybrid Memories"** SAFARI Technical Report, TR-SAFARI-2011-005, Carnegie Mellon University, September 2011.

- Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug Burger, **"Architecting Phase Change Memory as a Scalable DRAM Alternative"** *Proceedings of the 36th International Symposium on Computer Architecture* (***ISCA***), pages 2-13, Austin, TX, June 2009. Slides (pdf)

- Benjamin C. Lee, Ping Zhou, Jun Yang, Youtao Zhang, Bo Zhao, Engin Ipek, Onur Mutlu, and Doug Burger, **"Phase Change Technology and the Future of Main Memory"** *IEEE Micro, Special Issue: Micro's Top Picks from 2009 Computer Architecture Conferences* (***MICRO TOP PICKS***), Vol. 30, No. 1, pages 60-70, January/February 2010.