Computer Architecture
Lecture 17:
Bottleneck Acceleration

Prof. Onur Mutlu
ETH Zürich
Fall 2020
20 November 2020

Yesterday and Today

- Heterogeneous Multi-Core Systems
- Bottleneck Acceleration

Some Readings

- Suleman et al., "Accelerating Critical Section Execution with Asymmetric Multi-Core Architectures," ASPLOS 2009, IEEE Micro Top Picks 2010.

- Joao et al., "Bottleneck Identification and Scheduling in Multithreaded Applications," ASPLOS 2012.

- Joao et al., "Utility-Based Acceleration of Multithreaded Applications on Asymmetric CMPs," ISCA 2013.

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

- Grochowski et al., "Best of Both Latency and Throughput," ICCD 2004.

Recall: 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)

Recall: Accelerating Parallel Bottlenecks

- Serialized or imbalanced execution in the parallel portion can also benefit from a large core

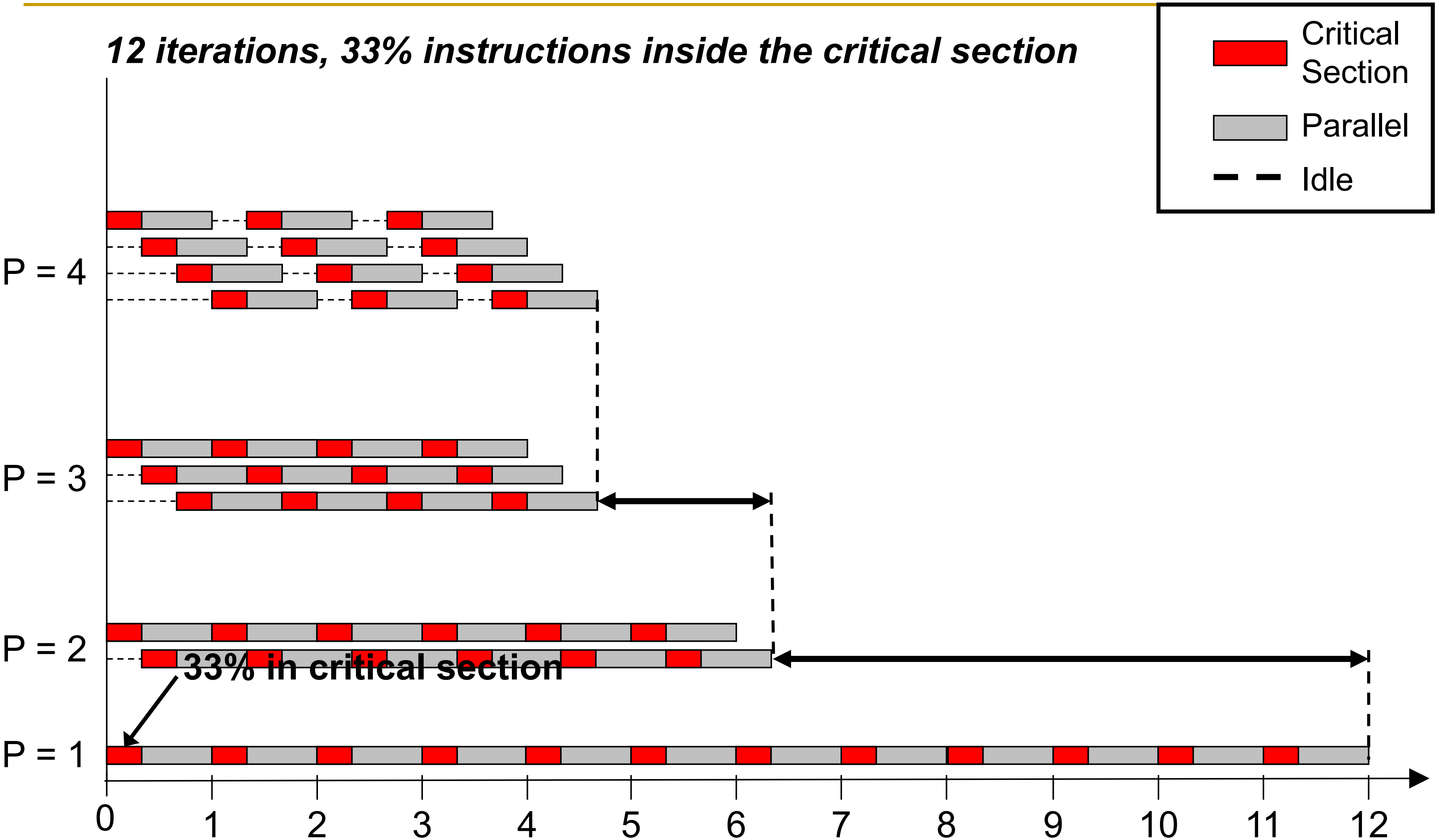
- Examples:
  - Critical sections that are contended
  - Parallel stages that take longer than others to execute

- Idea: Dynamically identify these code portions that cause serialization and execute them on a large core

Accelerated Critical Sections

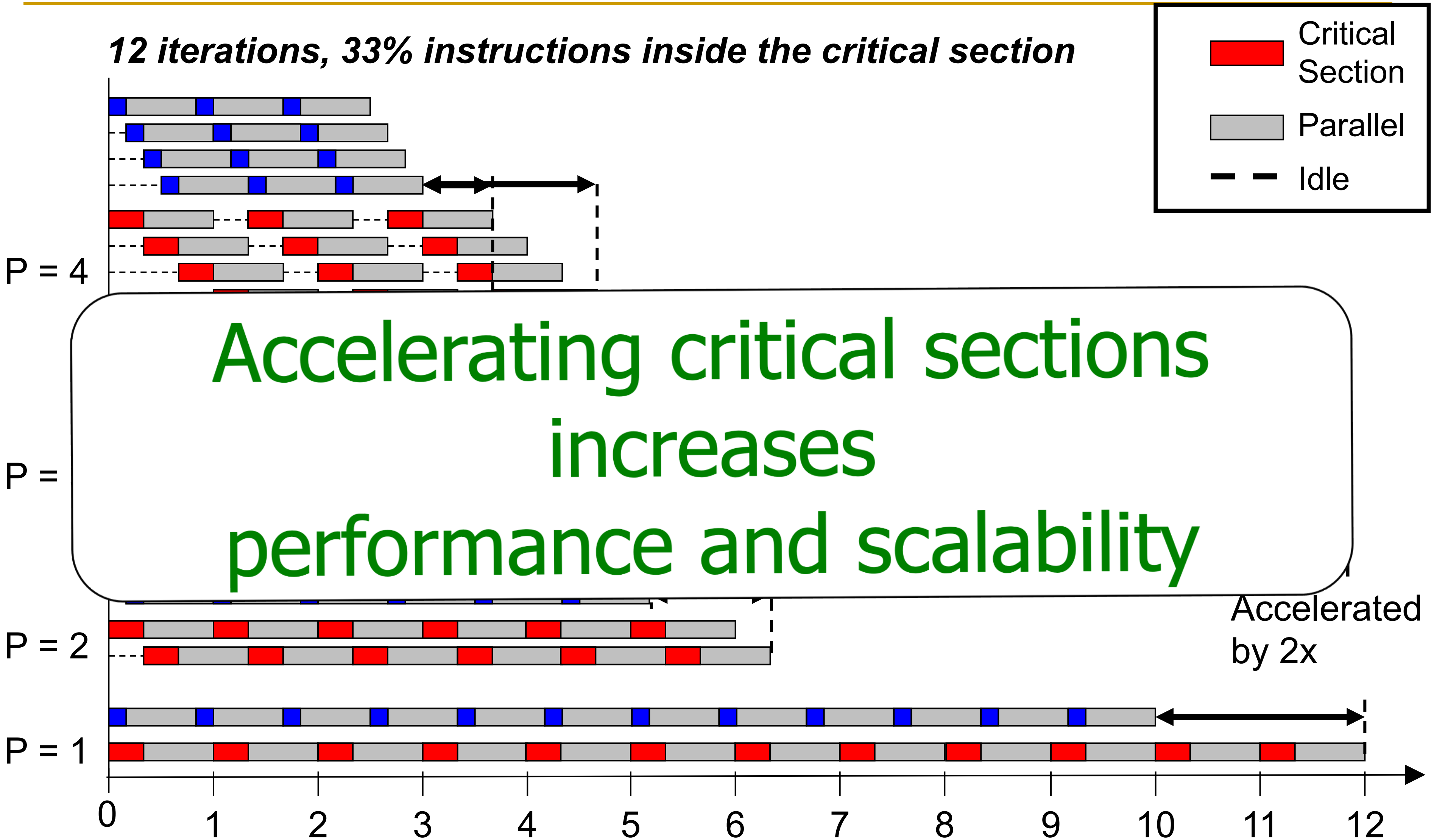
M. Aater Suleman, Onur Mutlu, Moinuddin K. Qureshi, and Yale N. Patt,
"Accelerating Critical Section Execution with Asymmetric Multi-Core Architectures"
*Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (**ASPLOS**), 2009*

Contention for Critical Sections



Contention for Critical Sections

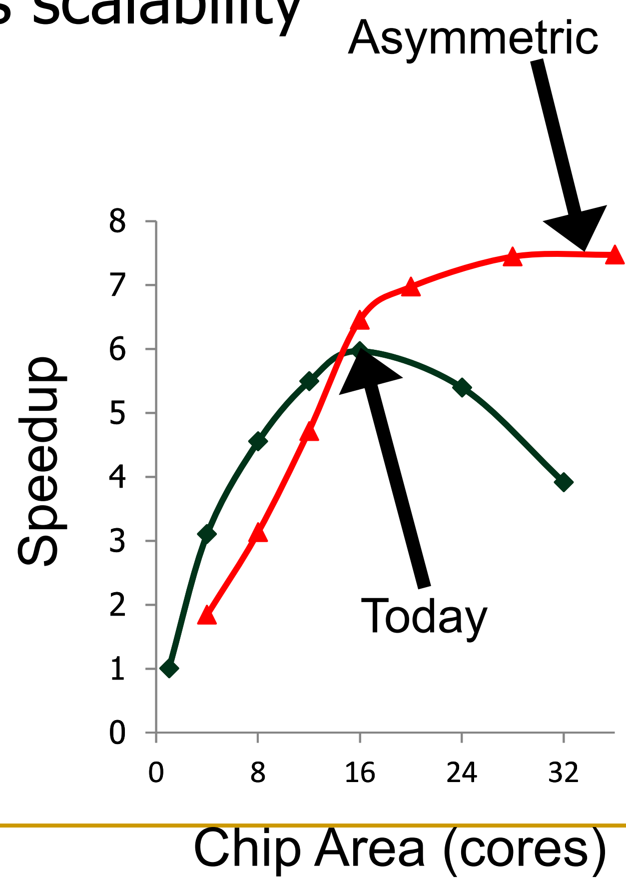
12 iterations, 33% instructions inside the critical section



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
  - Performance-debugging tradeoff
- 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

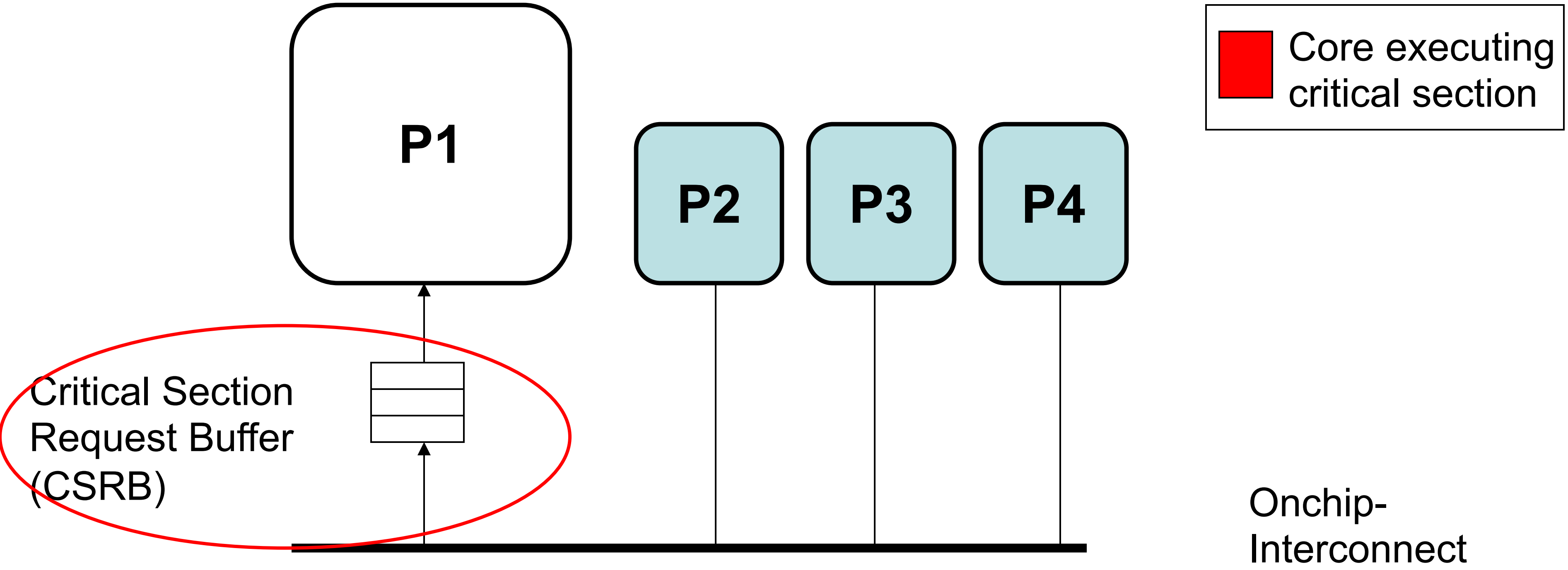
- Benefit:
  - Reduces serialization due to contended locks
  - Reduces the performance impact of hard-to-parallelize sections
  - Programmer does not need to (heavily) optimize parallel code → fewer bugs, improved productivity

- 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)

Small Core

A = compute()

LOCK X
    result = CS(A)
UNLOCK X

print result

Small Core

A = compute()
PUSH A
CSCALL X, Target PC
…
…
…
…
…
…
…

CSCALL Request
Send X, TPC,
STACK_PTR, CORE_ID

Large Core

…
…
…
Waiting in Critical Section Request Buffer (CSRB)

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

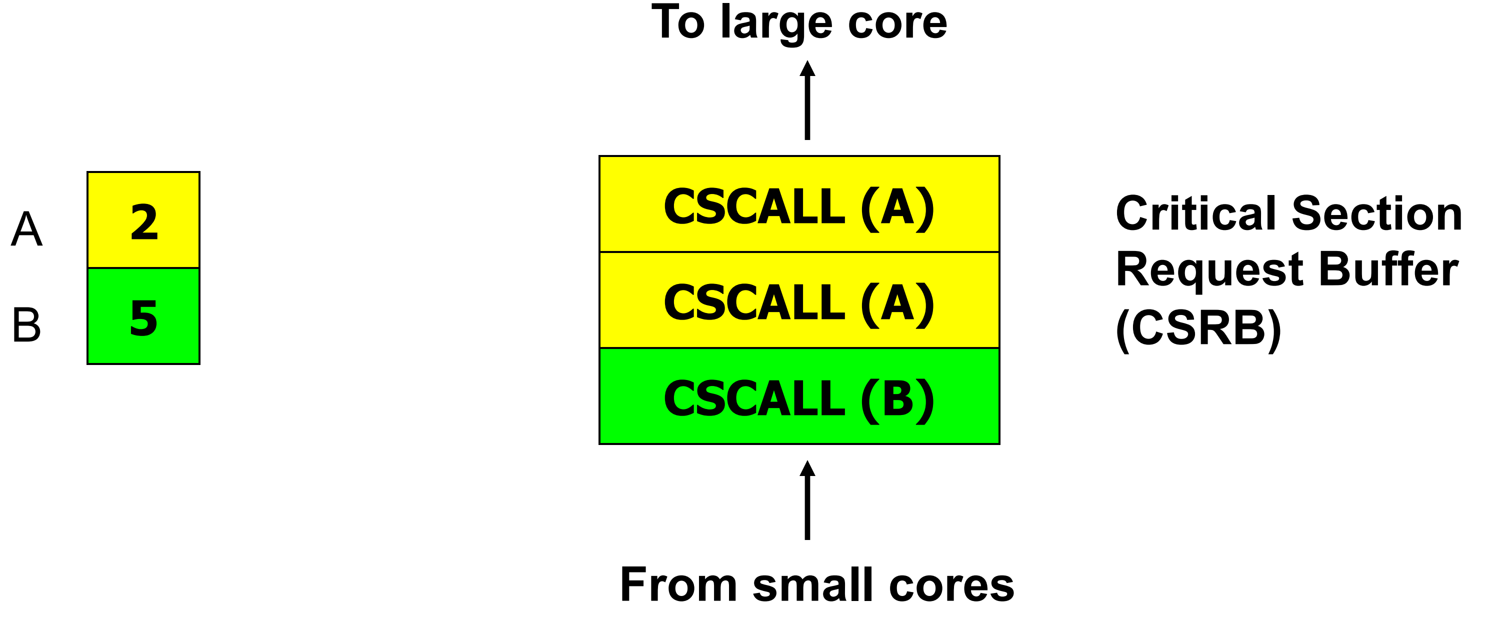
CSDONE Response

POP result
print result

- 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

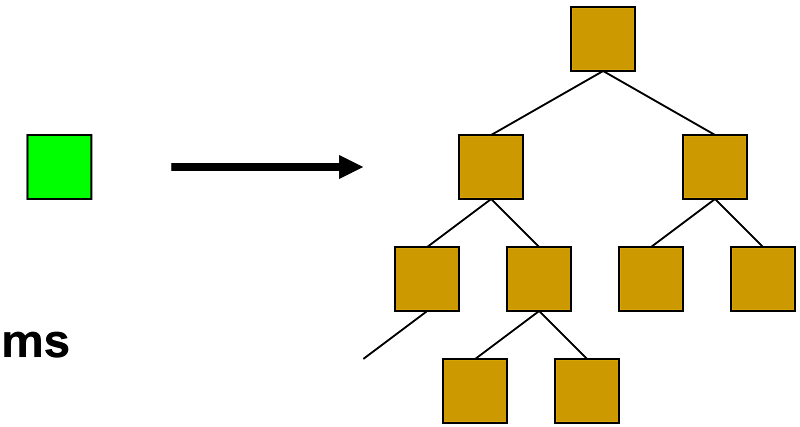
- ***Overhead of CSCALL/CSDONE vs. better lock locality***
  - ACS avoids "ping-ponging" of locks among caches by keeping them at the large core

- ***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

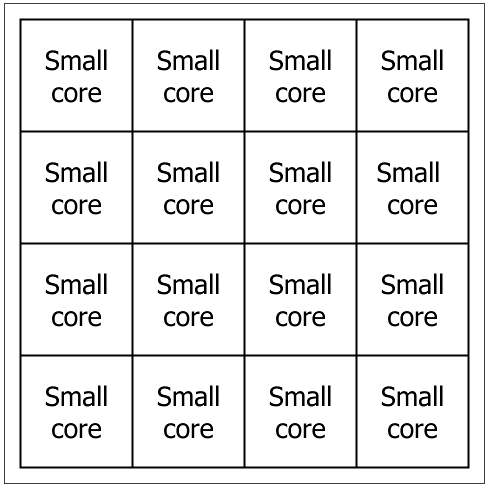
- ***Overhead of CSCALL/CSDONE vs. better lock locality***
  - ACS avoids "ping-ponging" of locks among caches by keeping them at the large core

- ***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

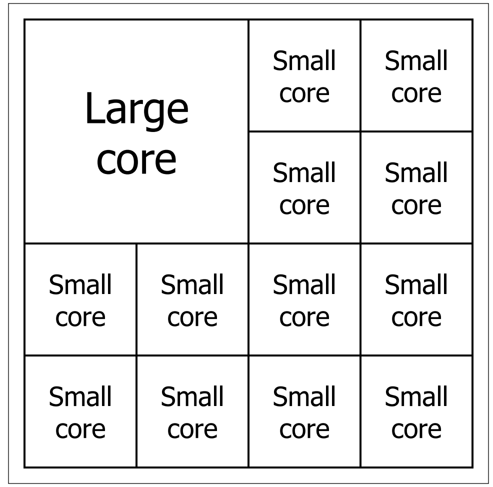
See Suleman et al., "Data Marshaling for Multi-Core Architectures," ISCA 2010.

ACS Comparison Points



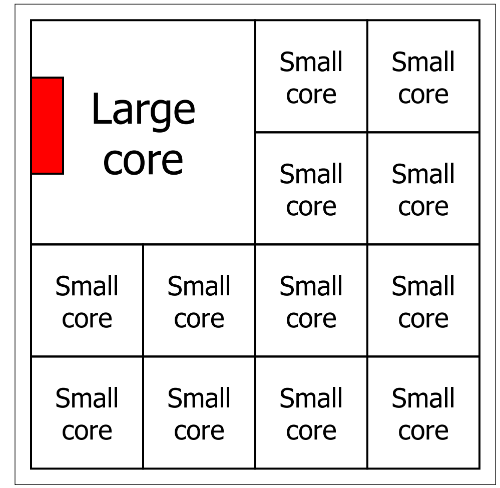
SCMP

- 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

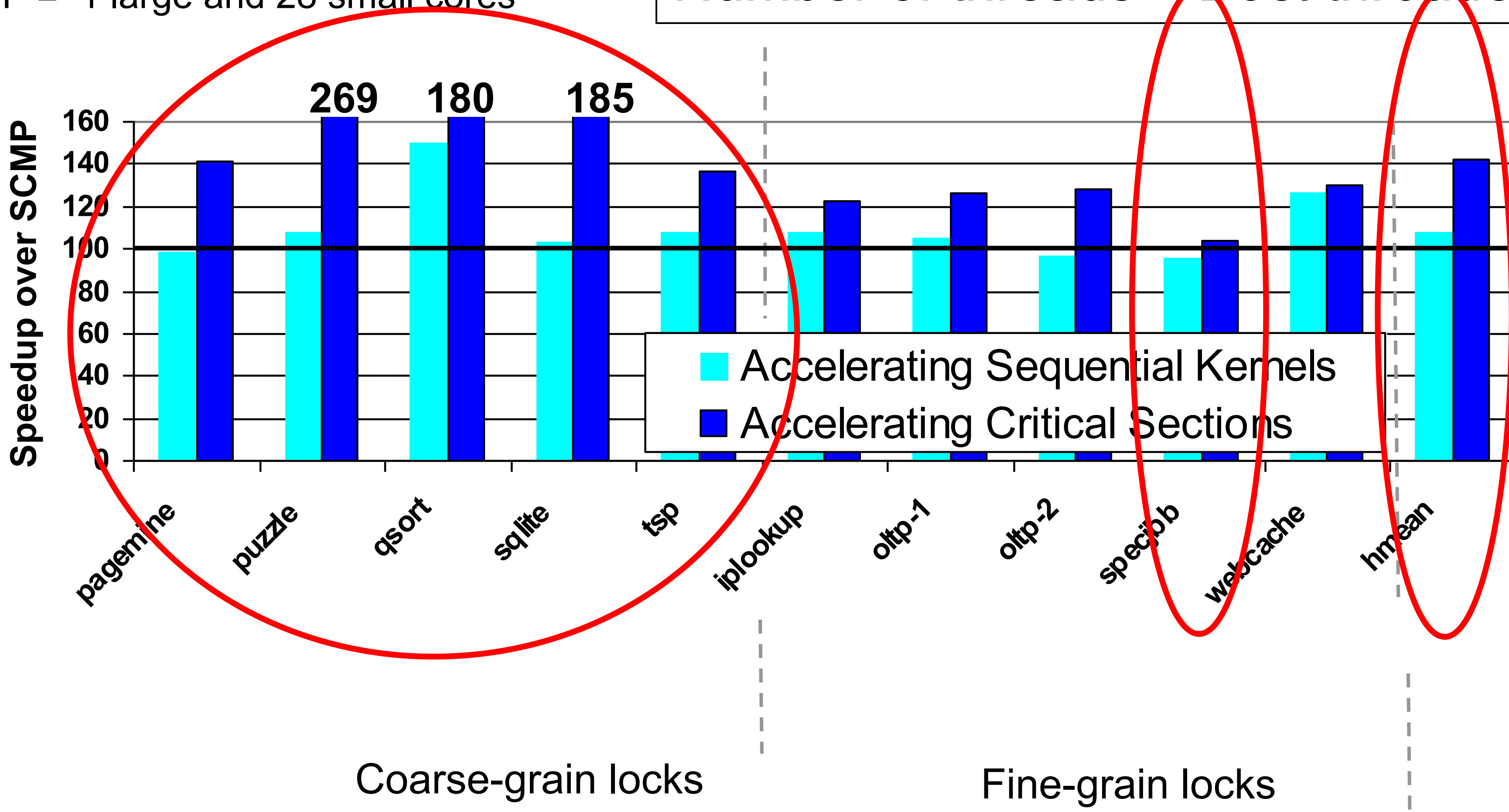
- Multi-core x86 simulator
  - 1 large and 28 small cores
  - Aggressive stream prefetcher employed at each core

- 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

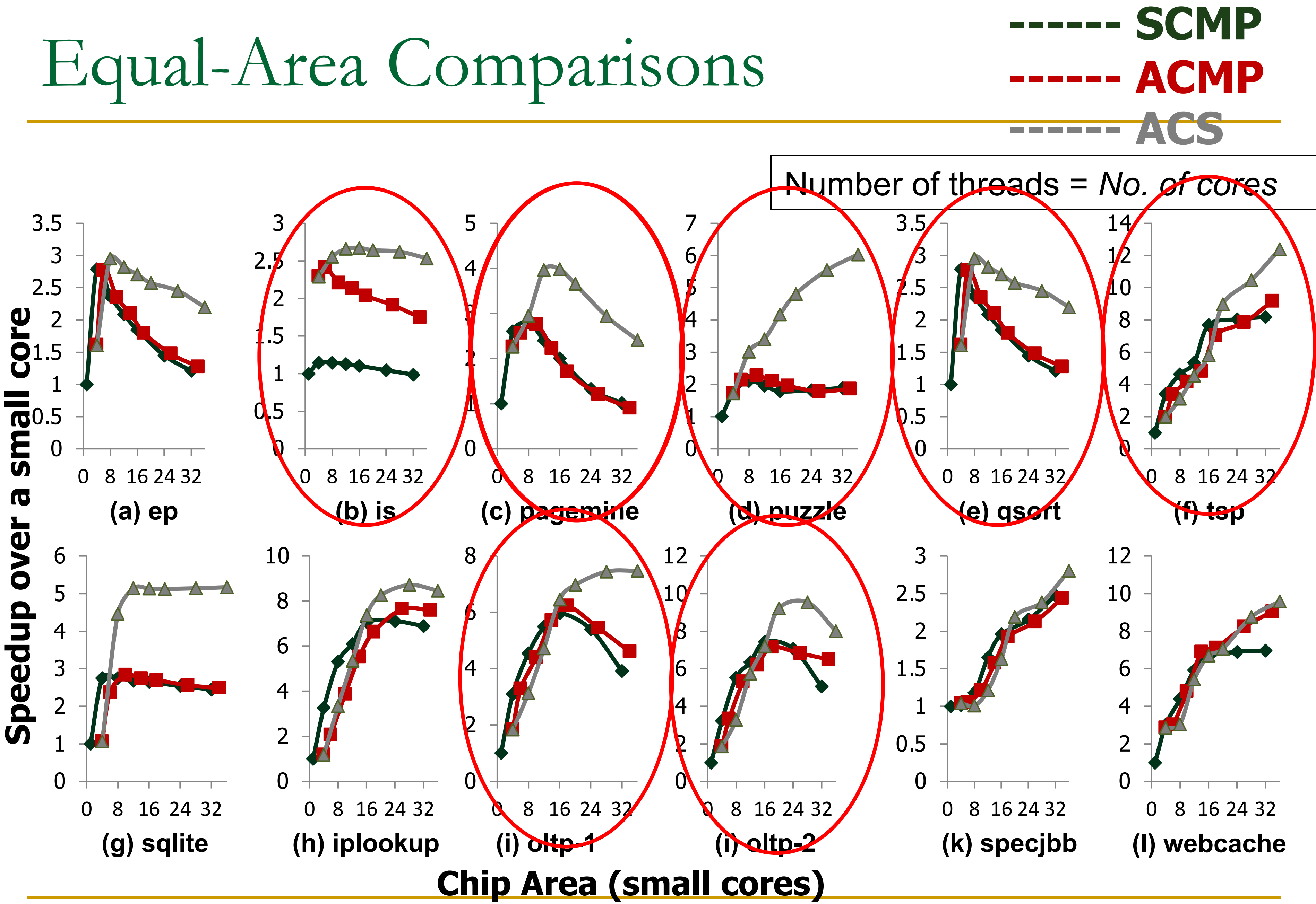
ACS Performance

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



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

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

Accelerating Critical Section Execution with Asymmetric Multi-Core Architectures

M. Aater Suleman
University of Texas at Austin
suleman@hps.utexas.edu

Onur Mutlu
Carnegie Mellon University
onur@cmu.edu

Moinuddin K. Qureshi
IBM Research
mkquresh@us.ibm.com

Yale N. Patt
University of Texas at Austin
patt@ece.utexas.edu

Generalization?

Can We Accelerate All Types of Synchronization Bottlenecks?

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.

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

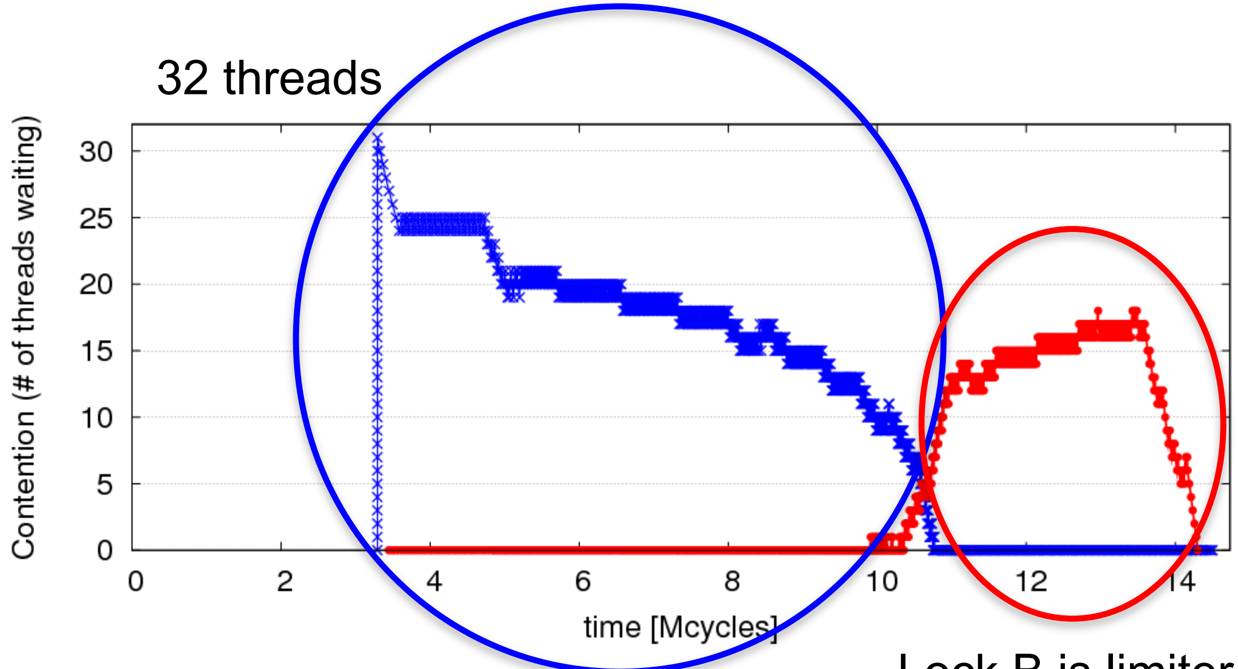
    Traverse list B

    Insert X into B

  Unlock B

until A is empty

32 threads

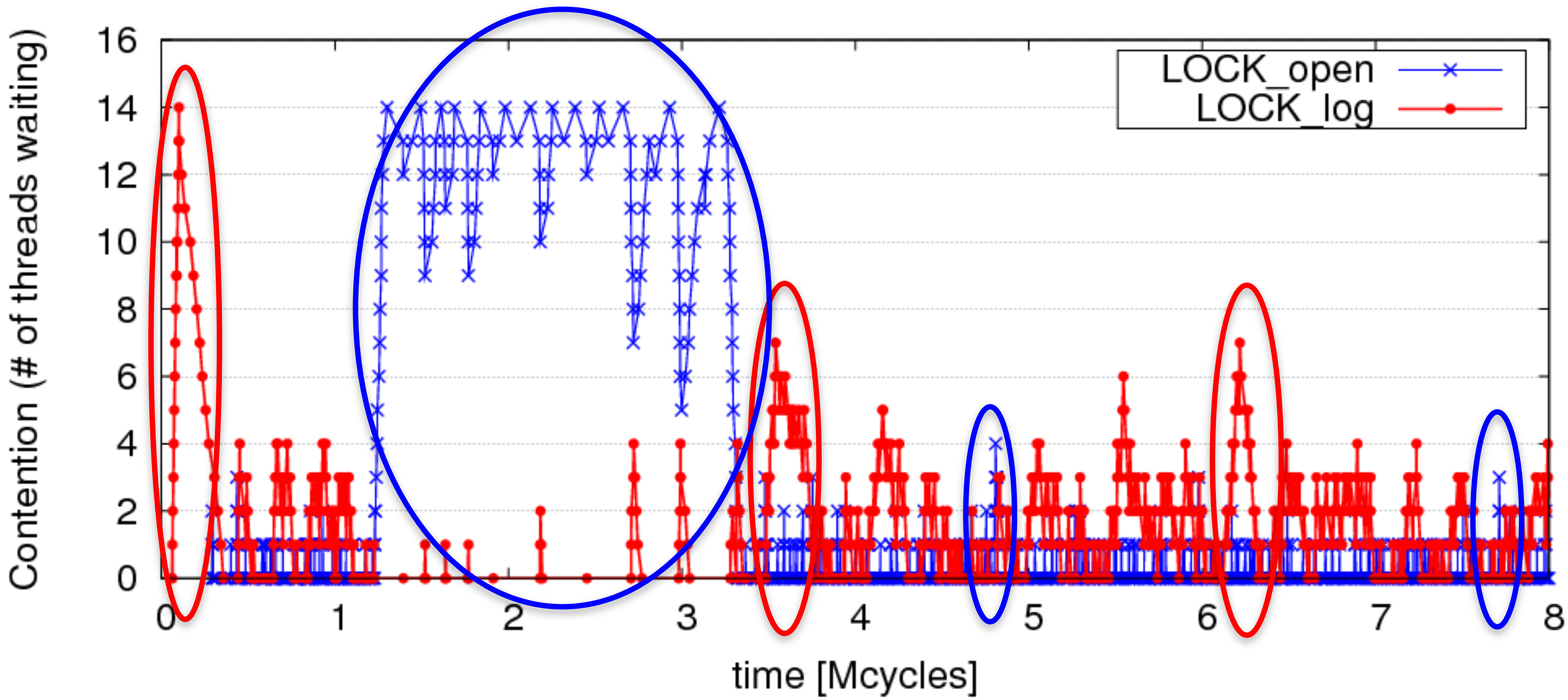


Lock A is limiter

Lock B is limiter

Limiting Bottlenecks Do Change on Real Applications

MySQL running Sysbench queries, 16 threads

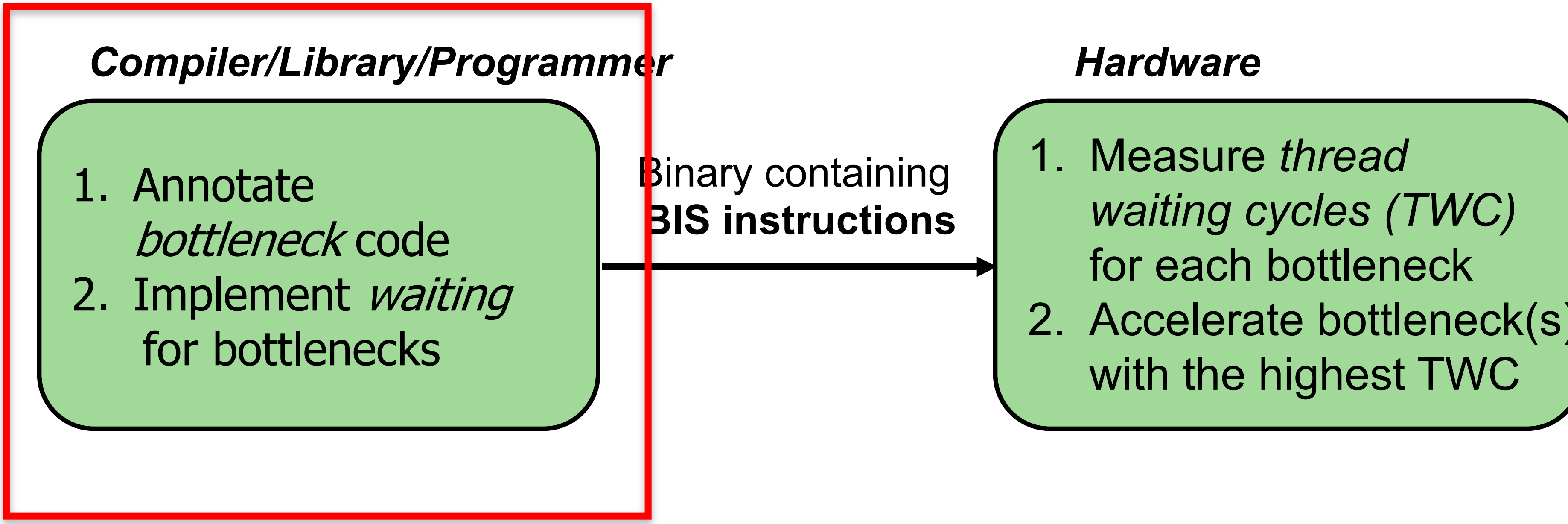


Bottleneck Identification and Scheduling (BIS)

- 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)



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
                BottleneckWait bid, watch_addr
            acquire lock
            ...
            release lock
            BottleneckReturn bid
```

Used to enable acceleration

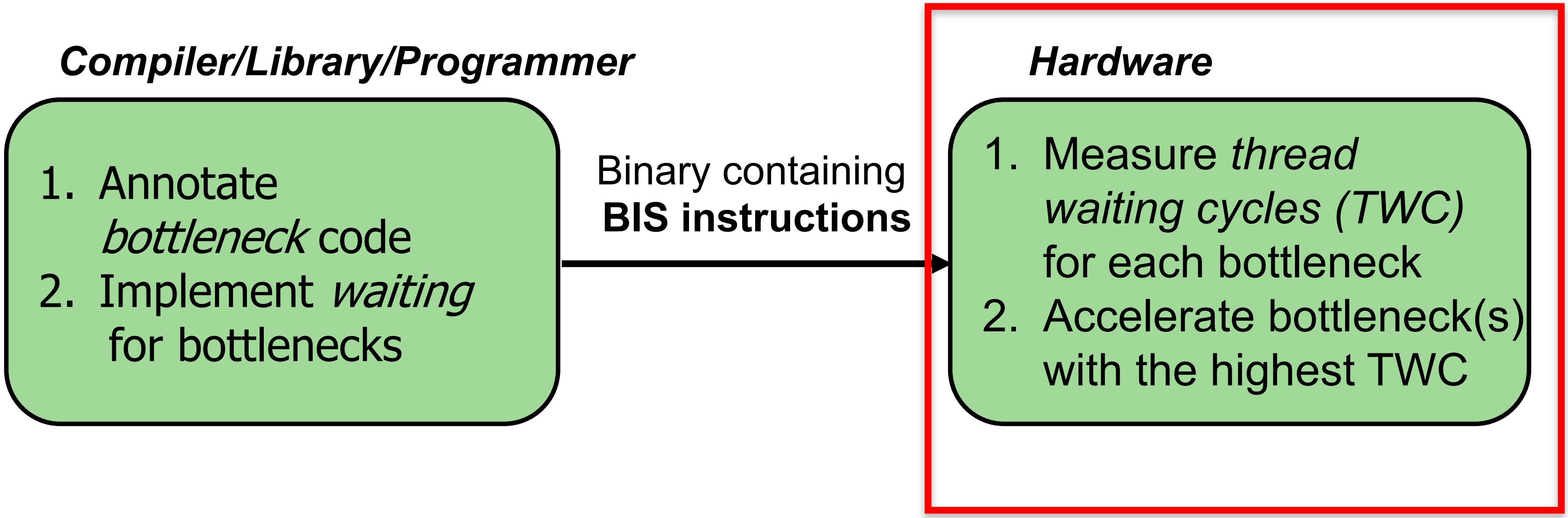
Barriers: Code Modifications

```
          …
          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

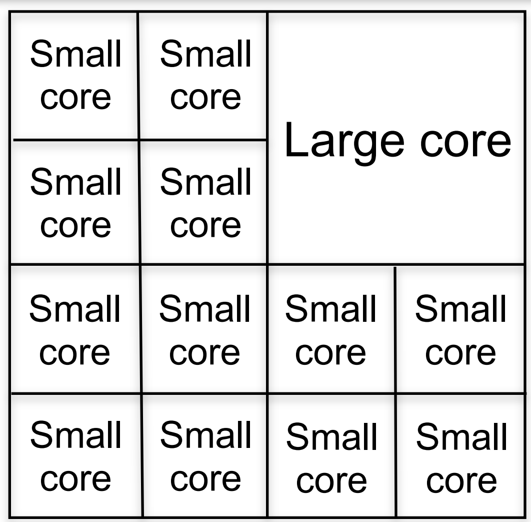
```
          BottleneckCall bid, targetPC
          …
targetPC: while not done
              while empty queue
                  BottleneckWait prev_bid
              dequeue work
              do the work …
              while full queue
                  BottleneckWait next_bid
              enqueue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

Bottleneck Identification and Scheduling (BIS)

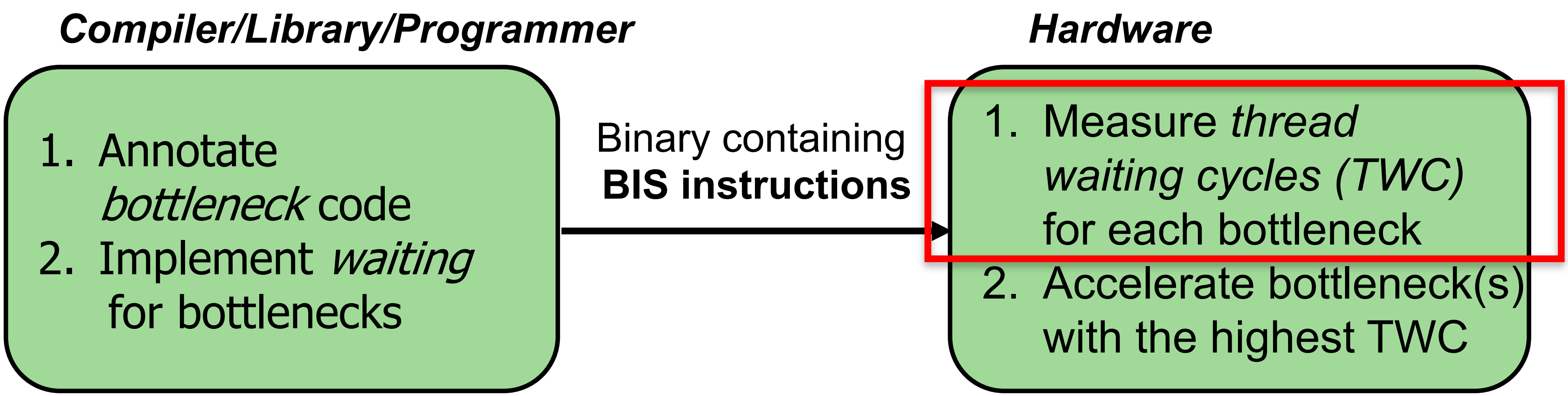


BIS: Hardware Overview

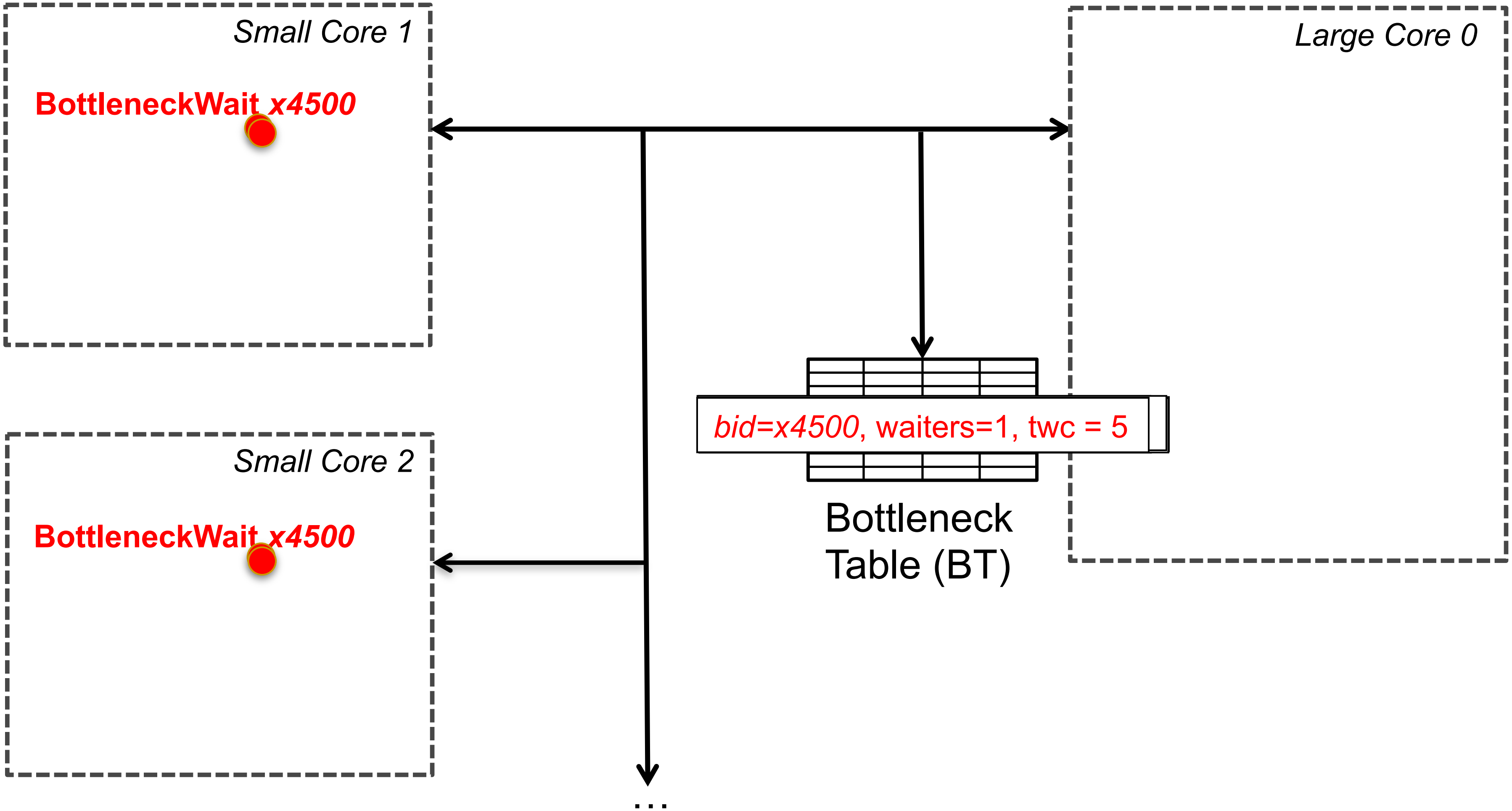
- Performance-limiting bottleneck identification and acceleration are independent tasks
- Acceleration can be accomplished in multiple ways
  - Increasing core frequency/voltage
  - Prioritization in shared resources [Ebrahimi+, MICRO'11]
  - Migration to faster cores in an Asymmetric CMP



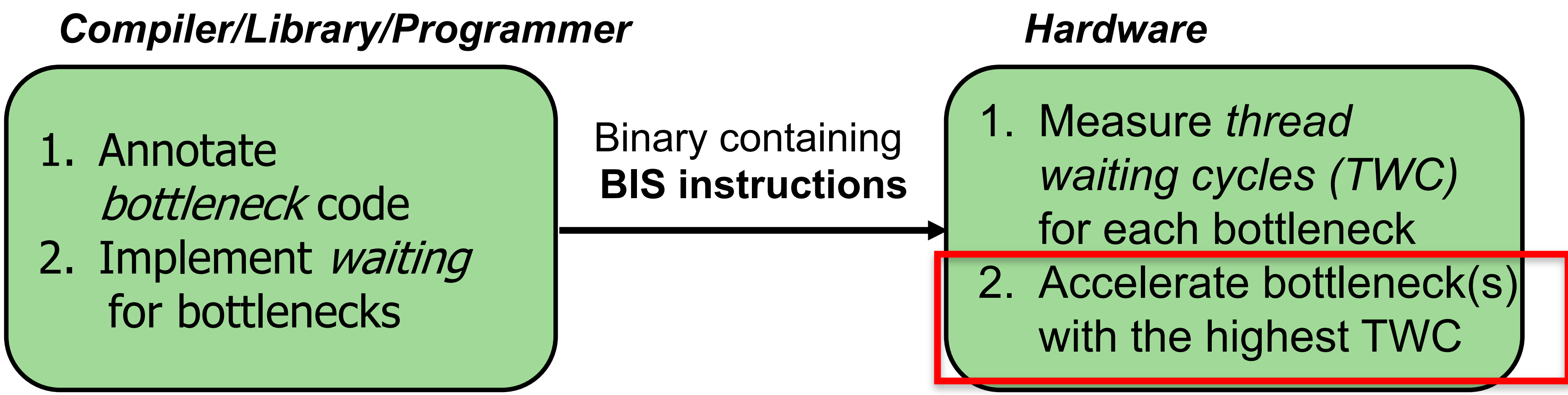
Bottleneck Identification and Scheduling (BIS)



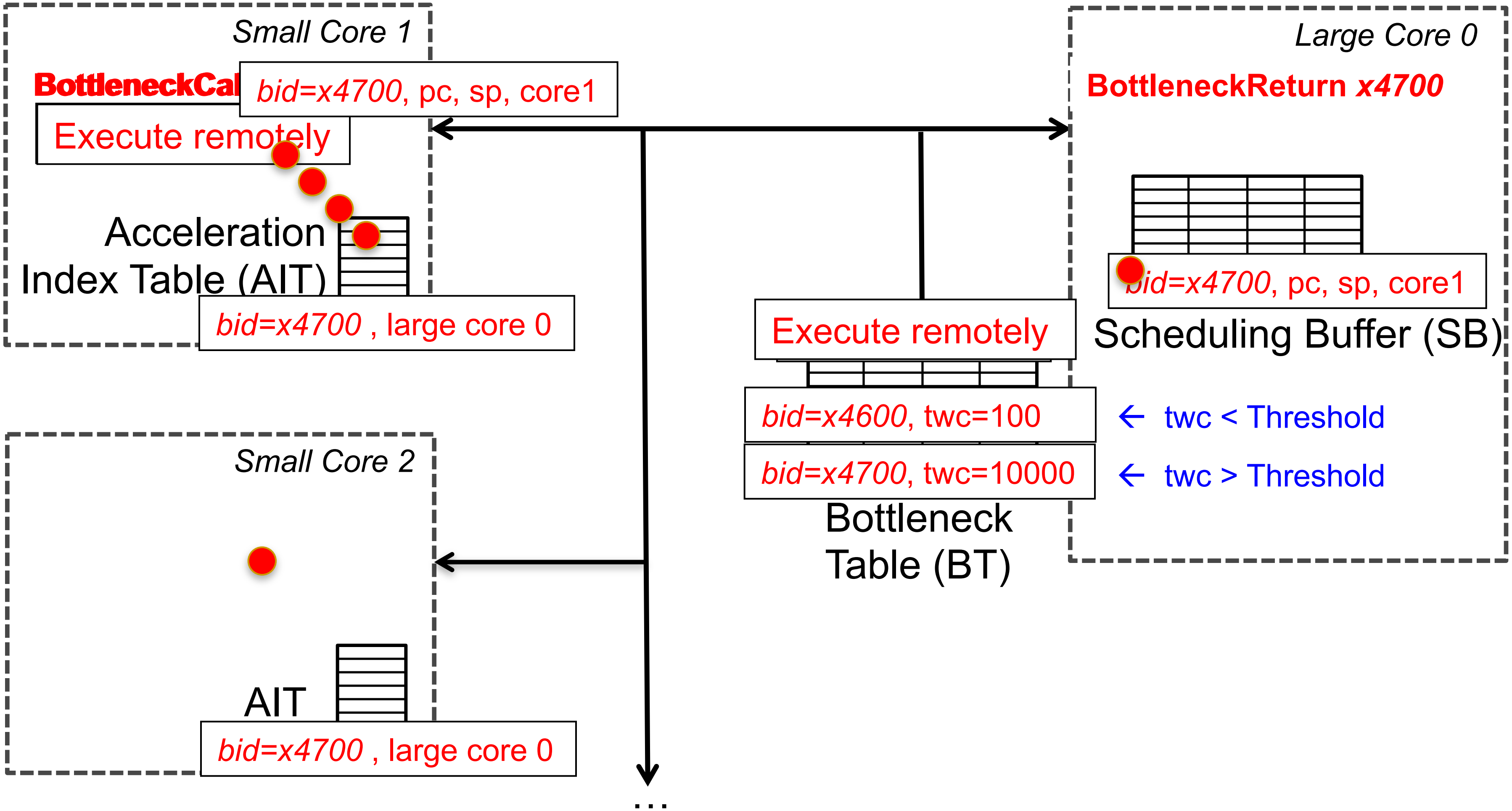
Determining Thread Waiting Cycles for Each Bottleneck



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

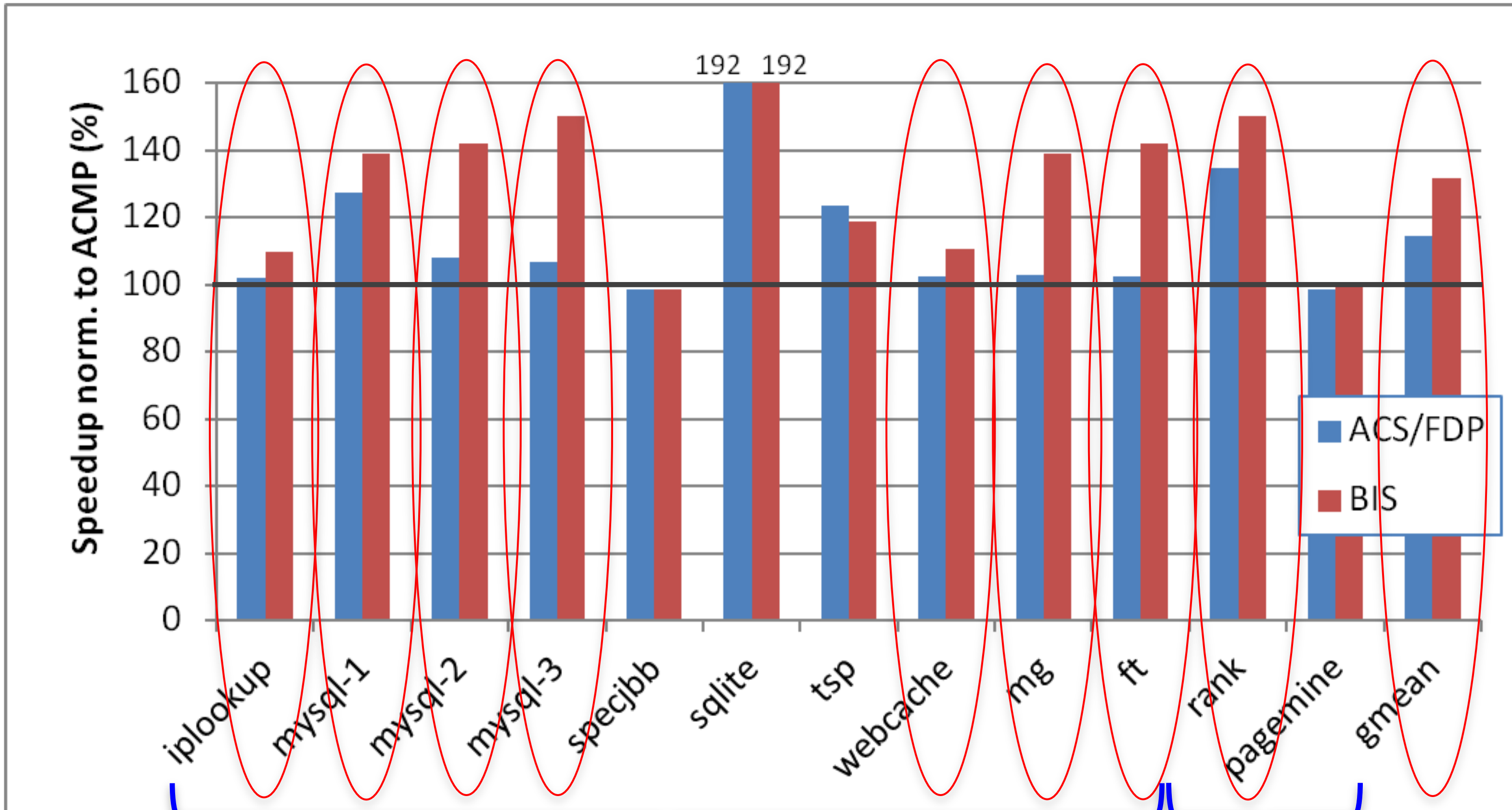
- ACMP (Asymmetric CMP)
  - Accelerates only Amdahl's serial portions
  - Our baseline

- 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)

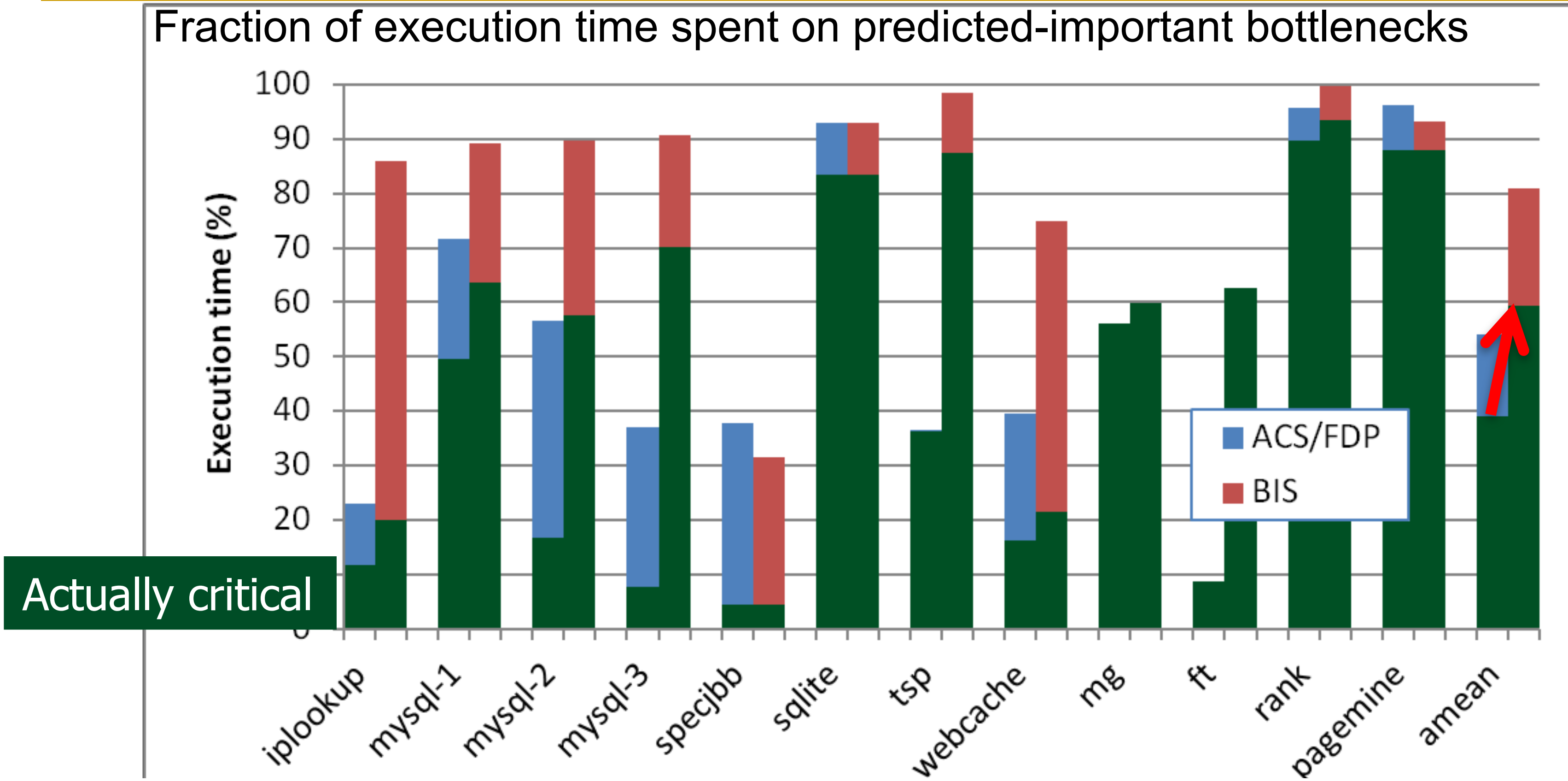
BIS Performance Improvement

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



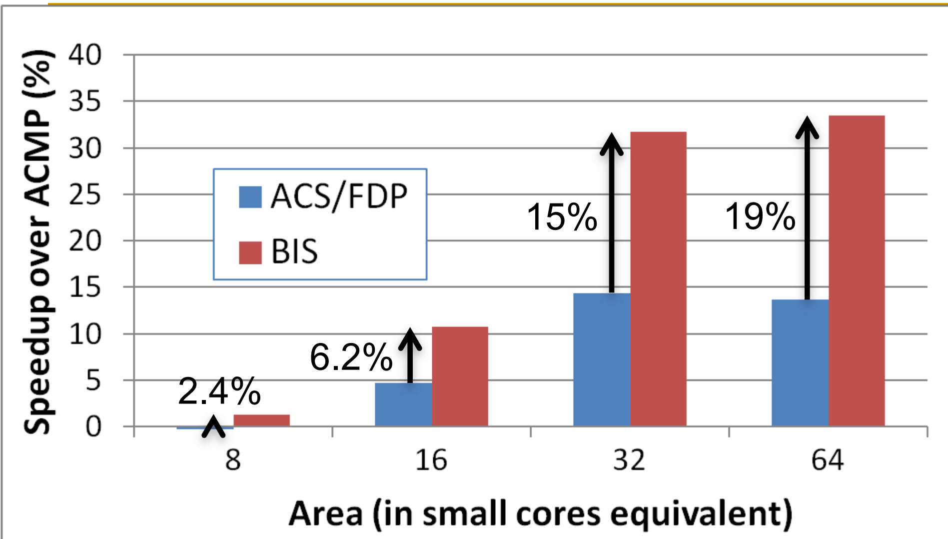
- BIS outperforms ACS/FDP by 15% and ACMP by 32%
- BIS improves scalability on 4 of the benchmarks

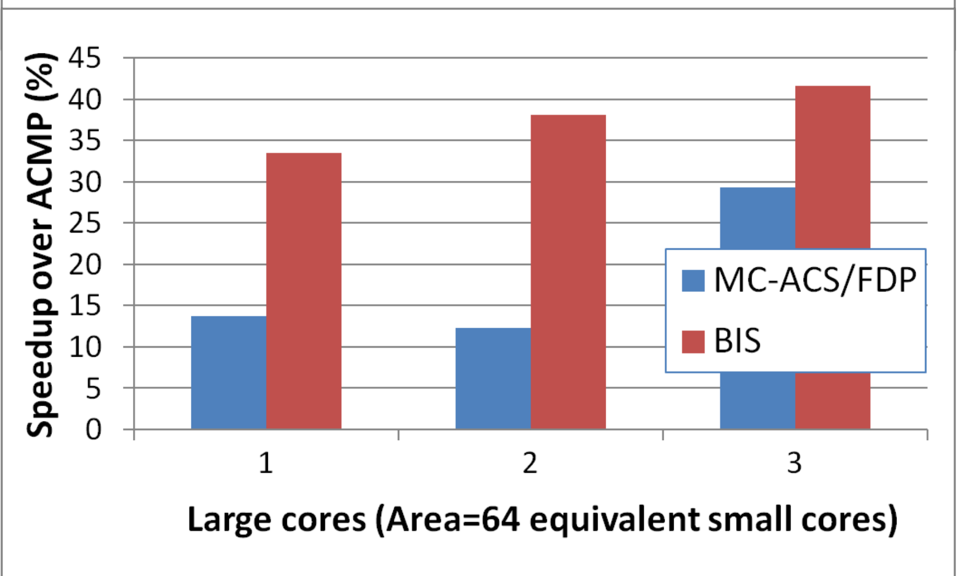
Why Does BIS Work?



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

BIS Scaling Results





Performance increases with:

1) More small cores
- Contention due to bottlenecks increases
- Loss of parallel throughput due to large core reduces

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

BIS Summary

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

More on Bottleneck Identification & 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. Slides (ppt) (pdf)

Bottleneck Identification and Scheduling in Multithreaded Applications

José A. Joao
ECE Department
The University of Texas at Austin
joao@ece.utexas.edu

M. Aater Suleman
Calxeda Inc.
aater.suleman@calxeda.com

Onur Mutlu
Computer Architecture Lab.
Carnegie Mellon University
onur@cmu.edu

Yale N. Patt
ECE Department
The University of Texas at Austin
patt@ece.utexas.edu

Improving on BIS?

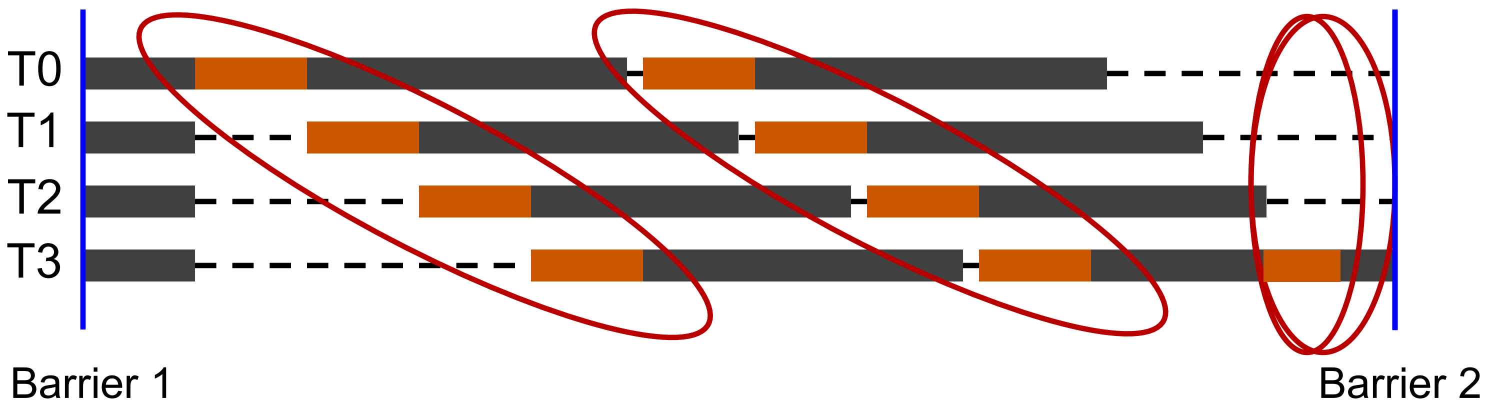
Can We Make
Better Acceleration Decisions?

Utility-Based Acceleration of Multithreaded Applications

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)

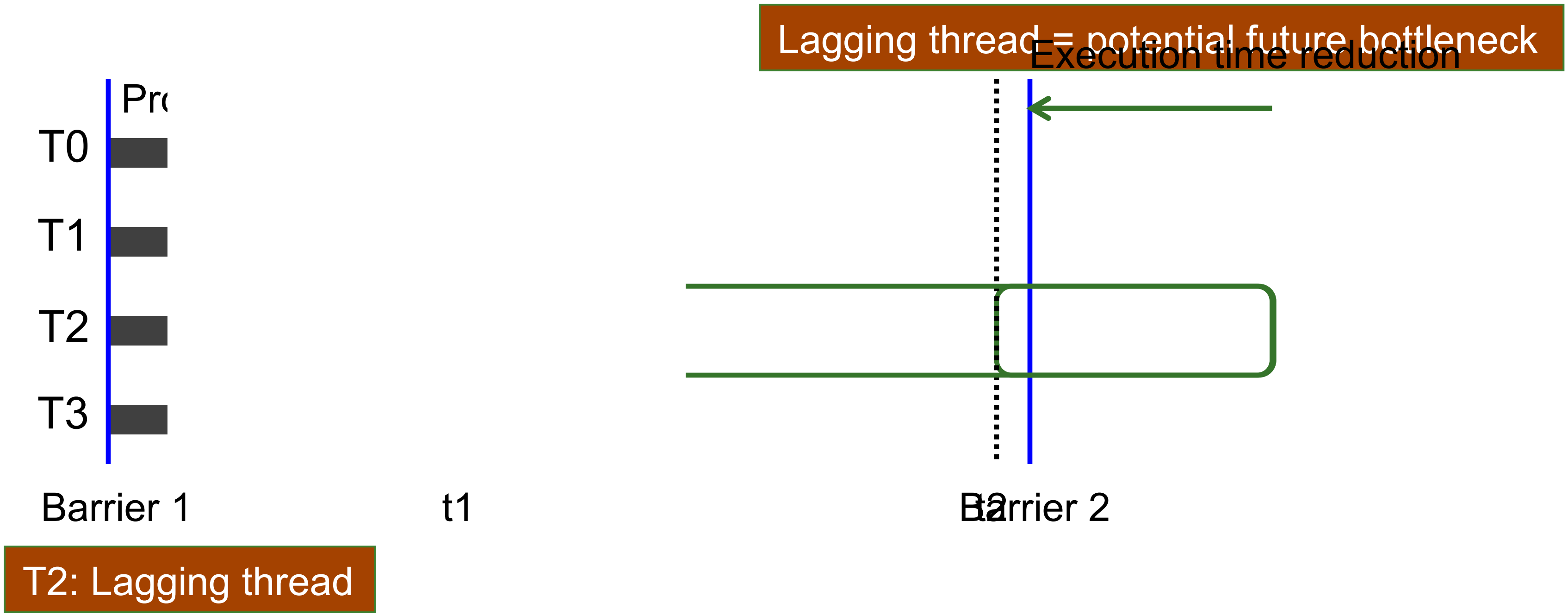
Bottlenecks

Accelerating Critical Sections (ACS), Suleman et al., ASPLOS' 09



Bottleneck Identification and Scheduling (BIS), Joao et al., ASPLOS' 12

Lagging Threads



Two Problems

1) Do we accelerate bottlenecks or lagging threads?

2) Multiple applications: which application do we accelerate?

Application 1

Acceleration decisions need to consider both:

- the **criticality** of code segments
- how much **speedup** they get

for bottlenecks and lagging threads

from any running application

Utility-Based Acceleration (UBA)

- *Goal:* identify performance-limiting bottlenecks or lagging threads from any running application and accelerate them on large cores of an ACMP
- *Key insight:* A New Utility of Acceleration metric that combines speedup and criticality of each code segment
- Utility of accelerating code segment *c* of length t on an application of length T:

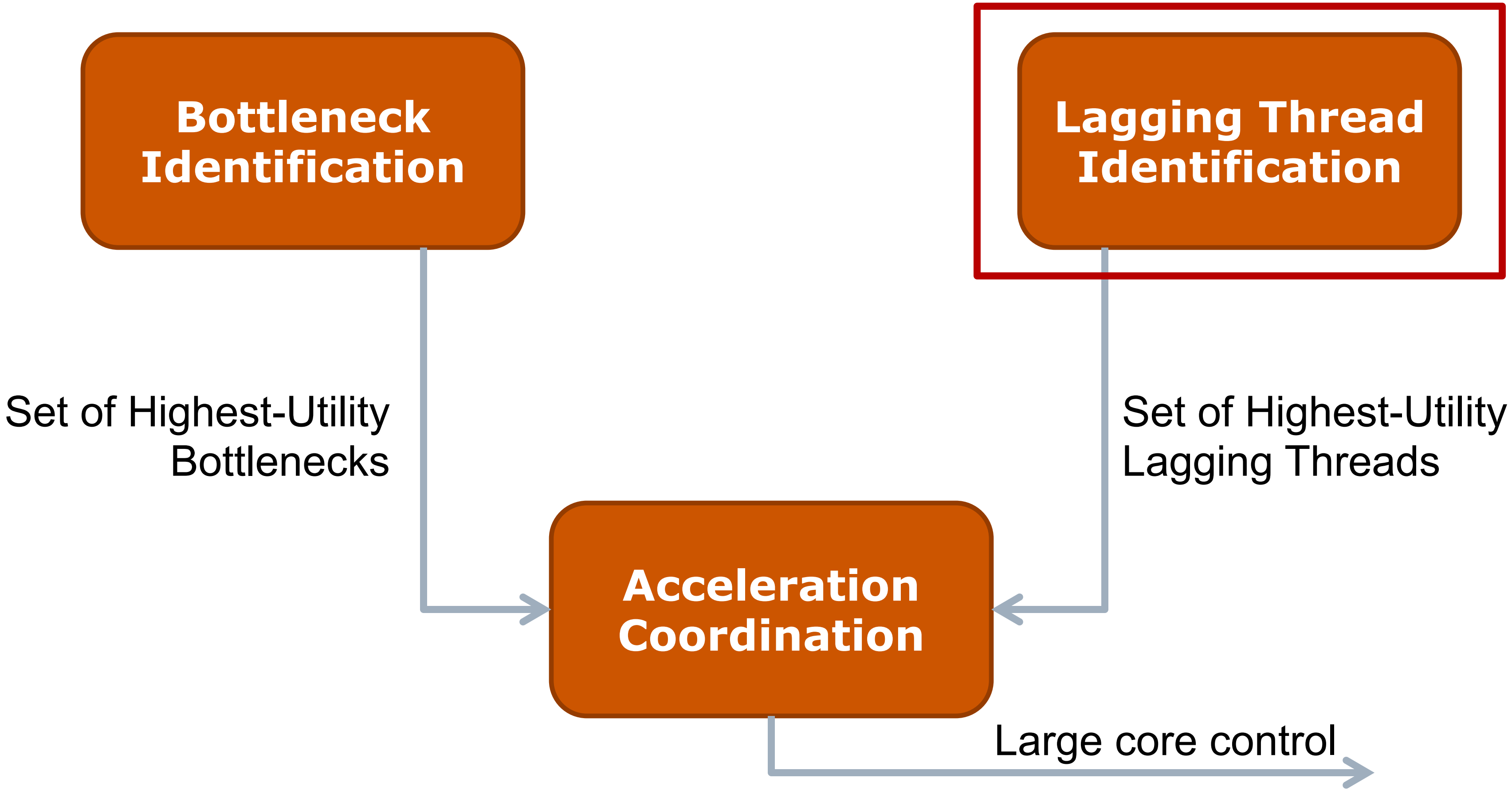
$$U_c = \frac{\Delta T}{T} = \left(\frac{\Delta t}{t}\right) \times \left(\frac{t}{T}\right) \times \left(\frac{\Delta T}{\Delta t}\right)$$

Global Criticality of Segment

Local Speedup of Segment

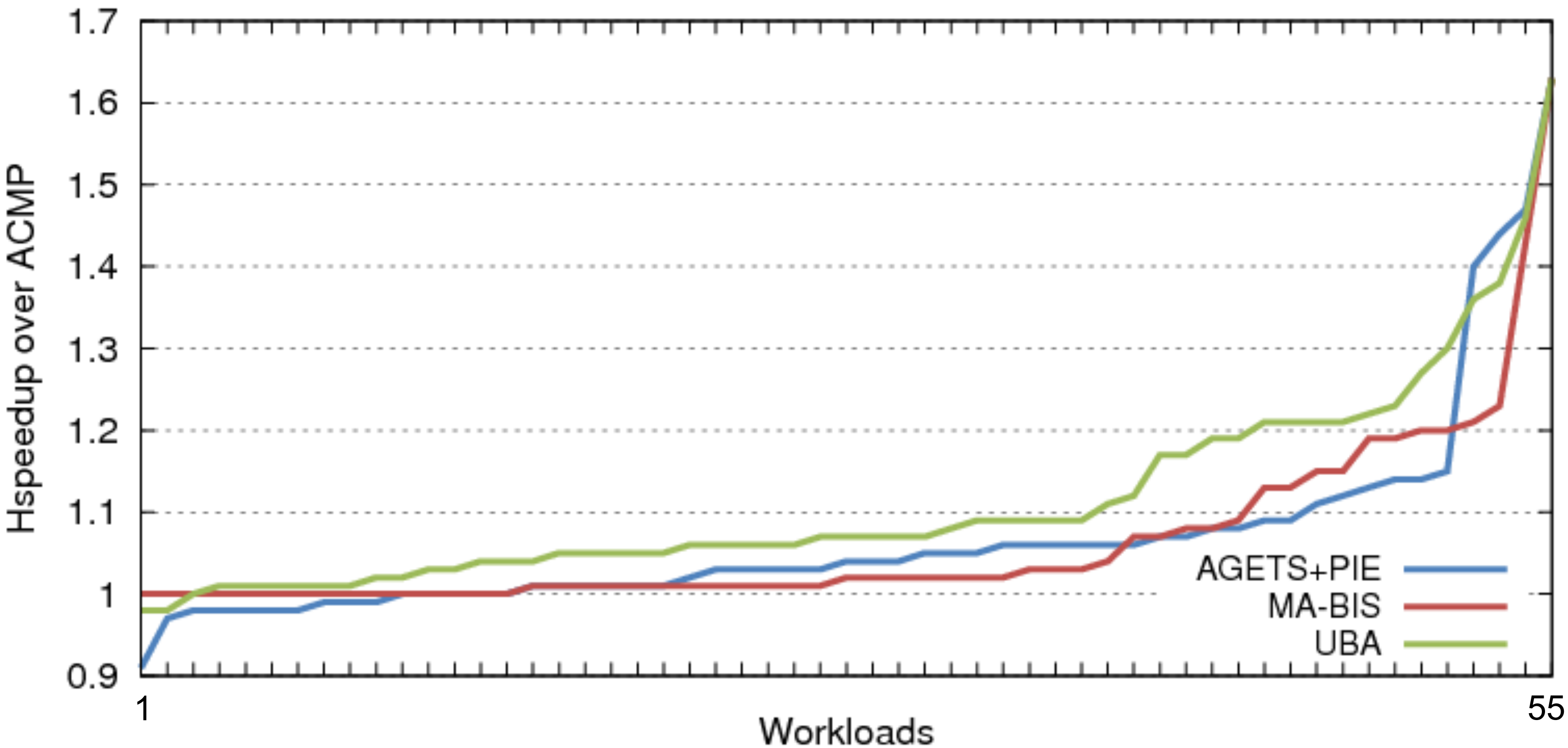
Fraction of Exec Time Spent on Segment

Utility-Based Acceleration (UBA)



UBA Results

2-application workloads, 60 small cores, 1 large core



UBA outperforms BIS and another alternative approach by ~8%.

More on Utility-Based Acceleration

- 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)

Utility-Based Acceleration of Multithreaded Applications on Asymmetric CMPs

José A. Joao [†] M. Aater Suleman [‡†] Onur Mutlu [§] Yale N. Patt [†]

[†] ECE Department
The University of Texas at Austin
Austin, TX, USA
{joao, patt}@ece.utexas.edu

[‡] Flux7 Consulting
Austin, TX, USA
suleman@hps.utexas.edu

[§] Computer Architecture Laboratory
Carnegie Mellon University
Pittsburgh, PA, USA
onur@cmu.edu

Better Bottleneck Acceleration

Can We Do Better?

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)

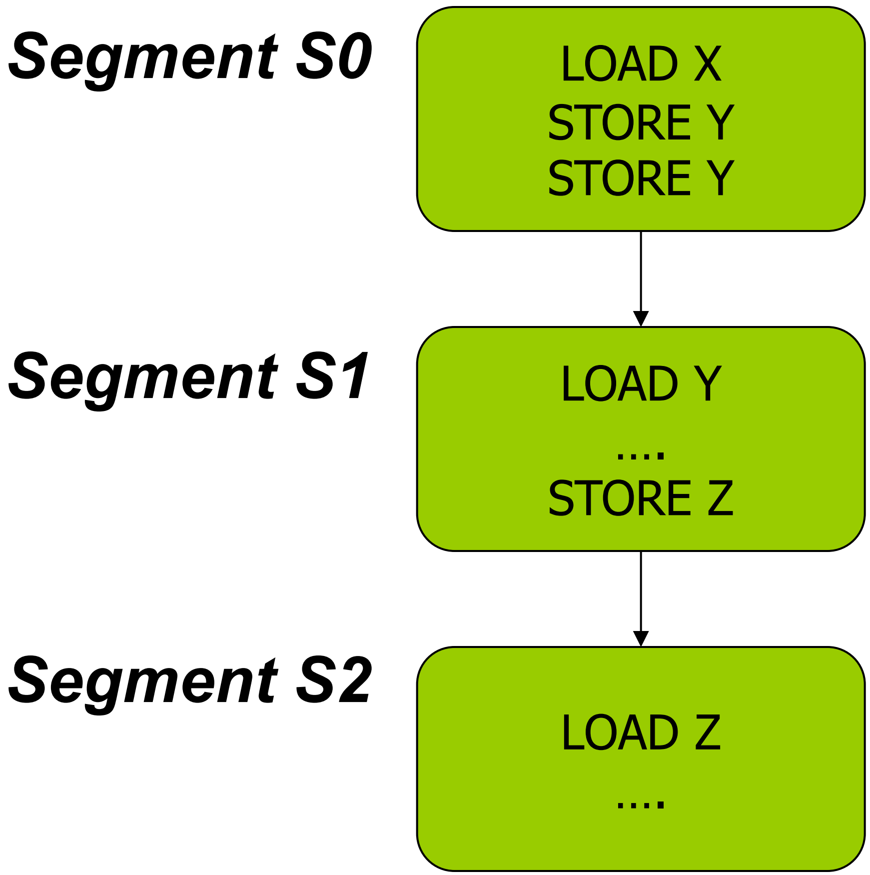
LOAD X
STORE Y
STORE Y

LOAD Y
....
STORE Z

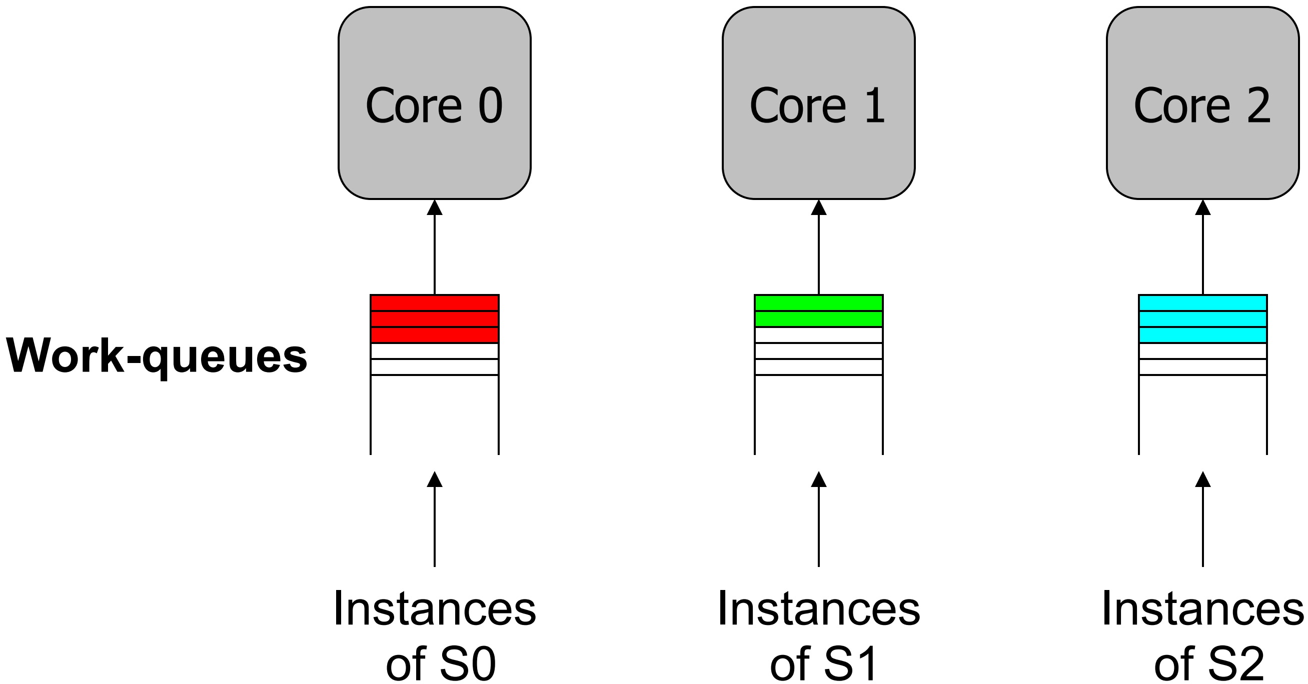
LOAD Z
....

Staged Execution Model (III)

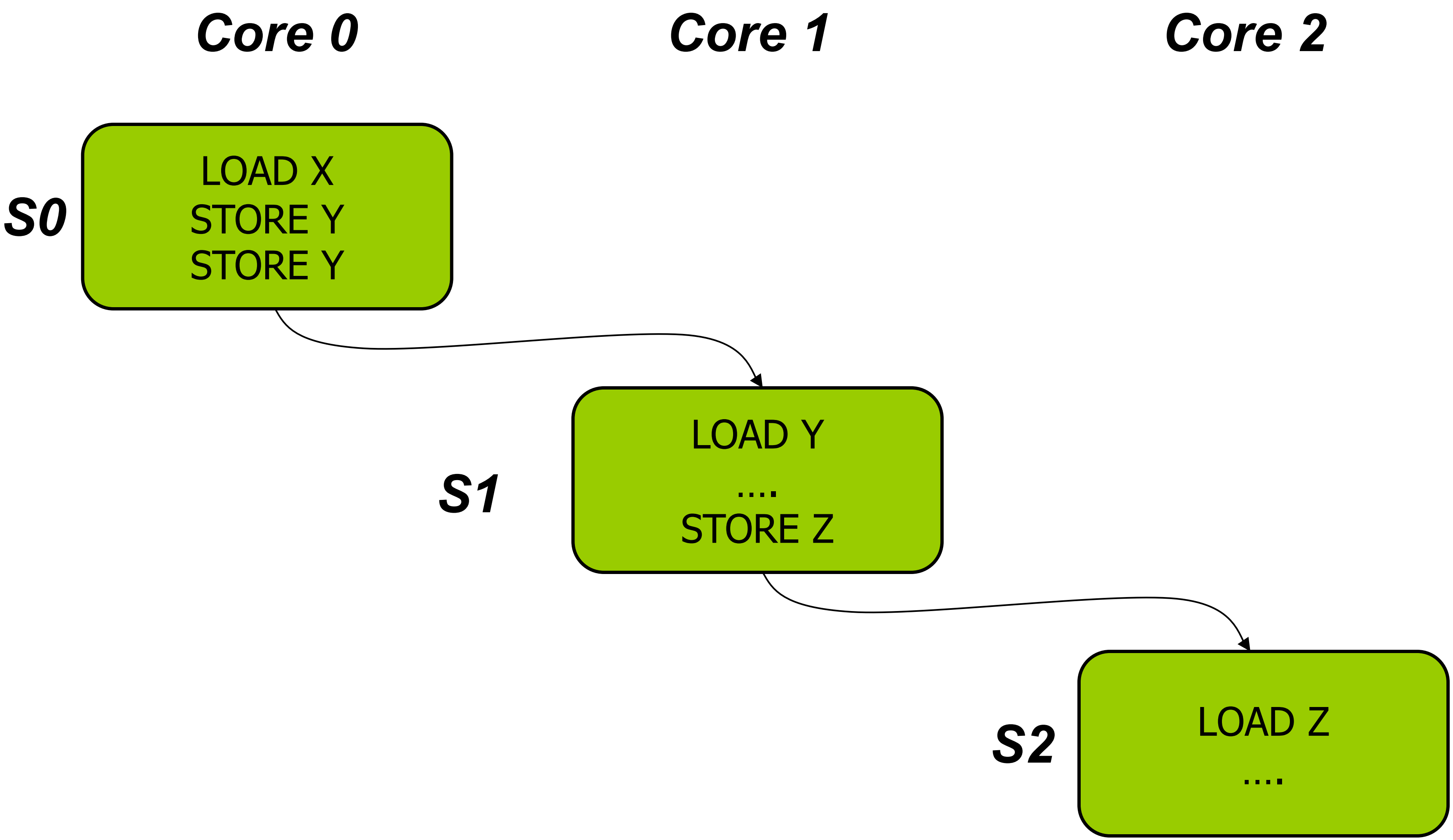
Split code into segments



Staged Execution Model (IV)



Staged Execution Model: Segment Spawning

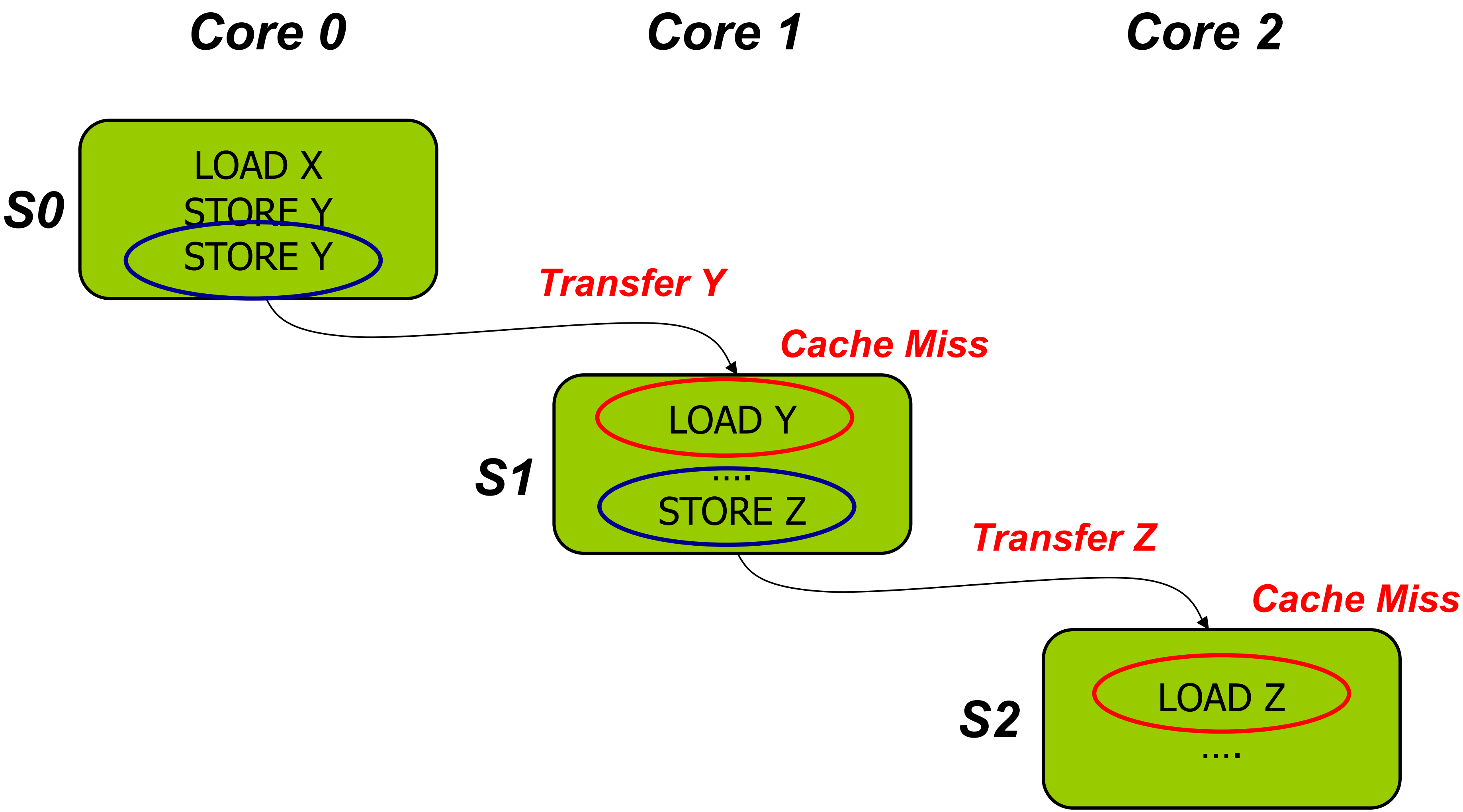


Staged Execution Model: Two Examples

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

- Producer-Consumer Pipeline Parallelism
  - Idea: Split a loop iteration into multiple "pipeline stages" where one stage consumes data produced by the previous 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



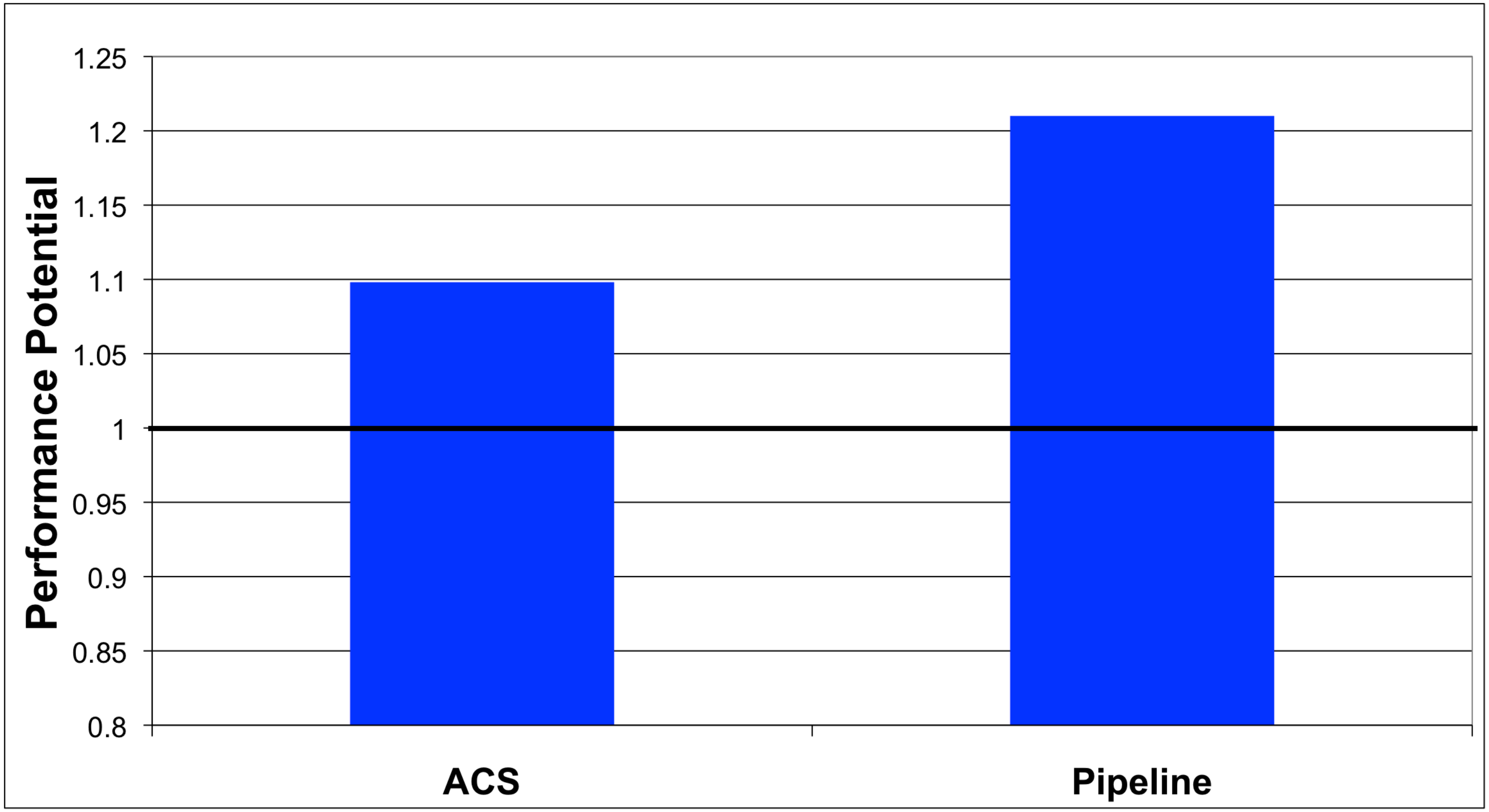
Problem: Locality of Inter-segment Data

- 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)

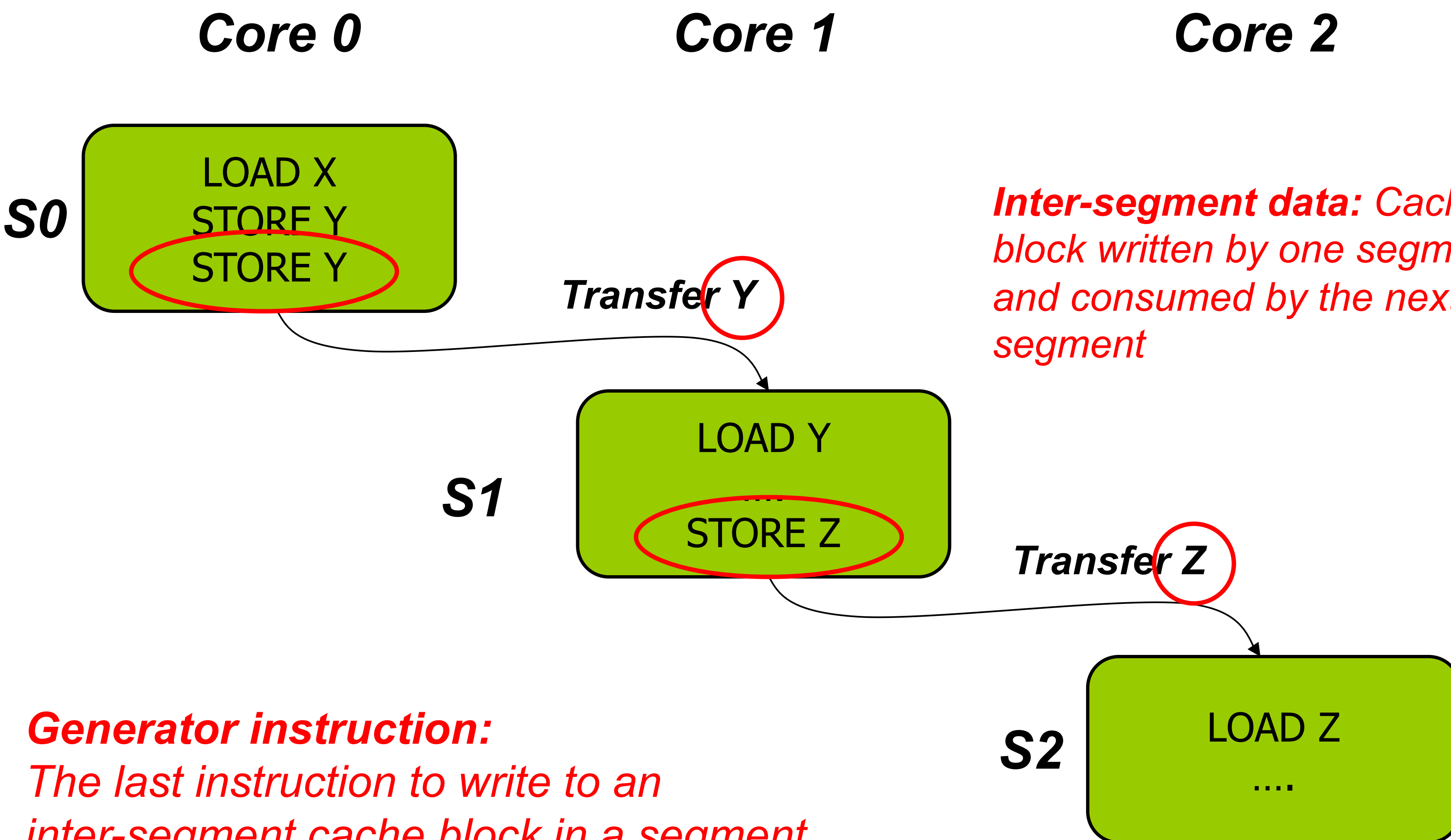
- Producer-Consumer Pipeline Parallelism
  - Idea: Split a loop iteration into multiple "pipeline stages" → each stage runs on a different core
  - Problem: A stage incurs a cache miss when it touches data produced by the previous stage

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

What if We Eliminated All Inter-segment Misses?



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

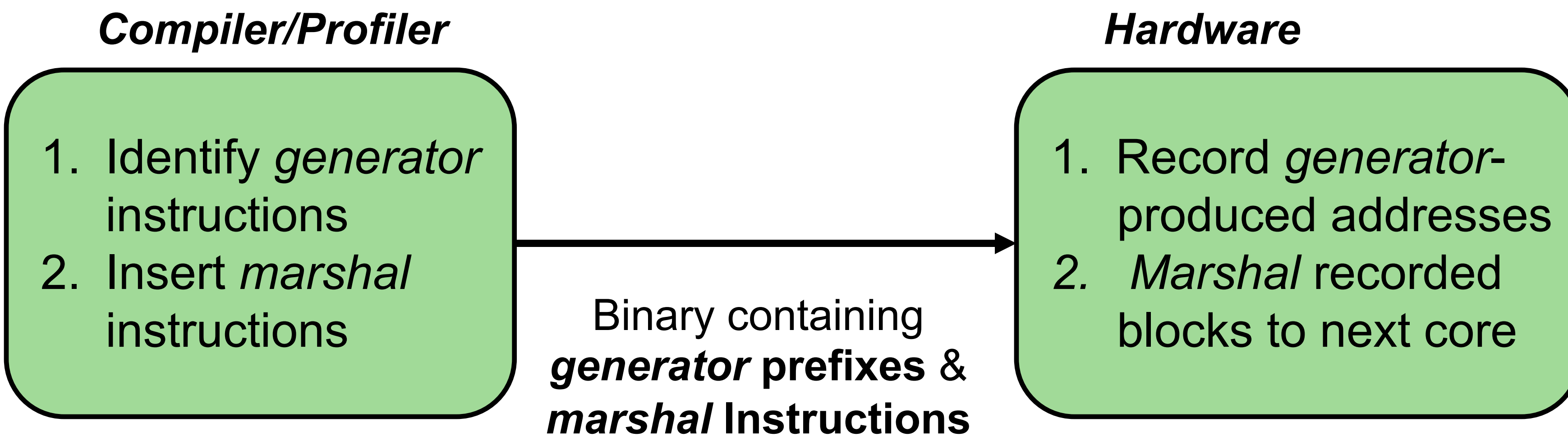
Key Observation and Idea

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

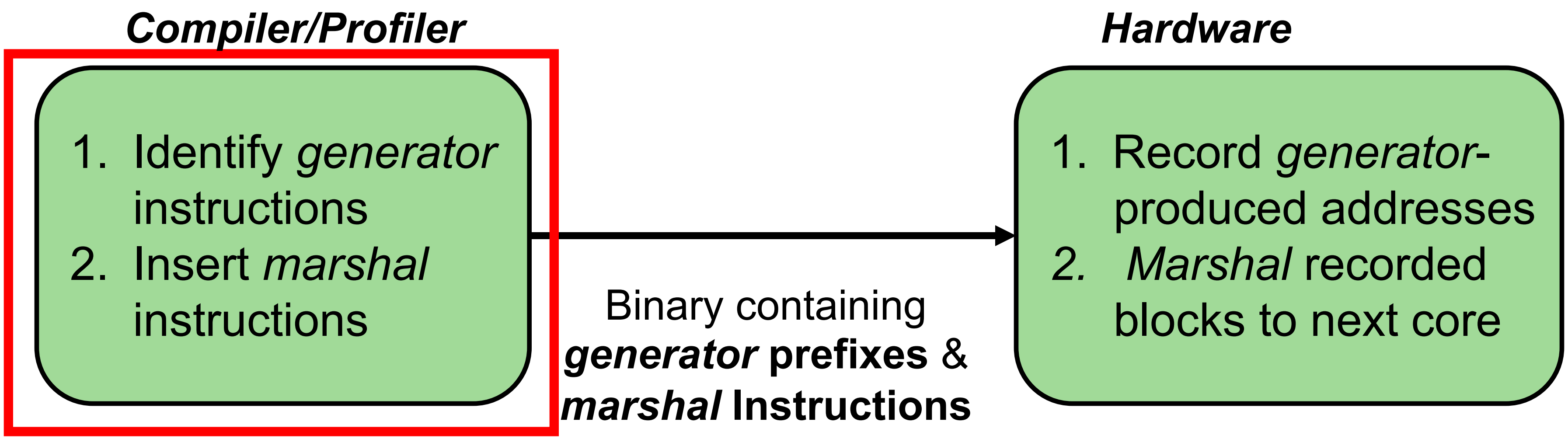
- Idea:
  - Identify the generator instructions
  - Record cache blocks produced by generator instructions
  - Proactively send such cache blocks to the next segment's core before initiating the next segment

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

Data Marshaling



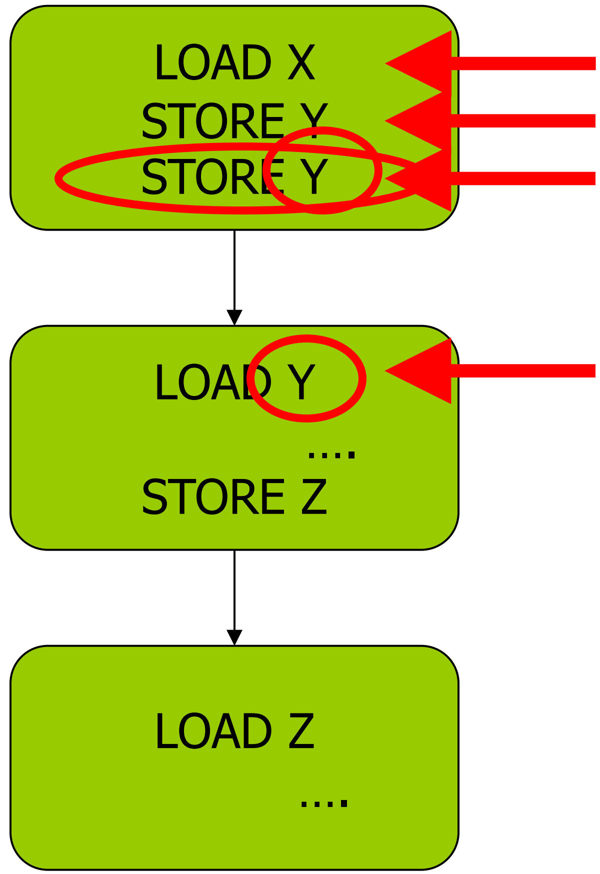
Data Marshaling



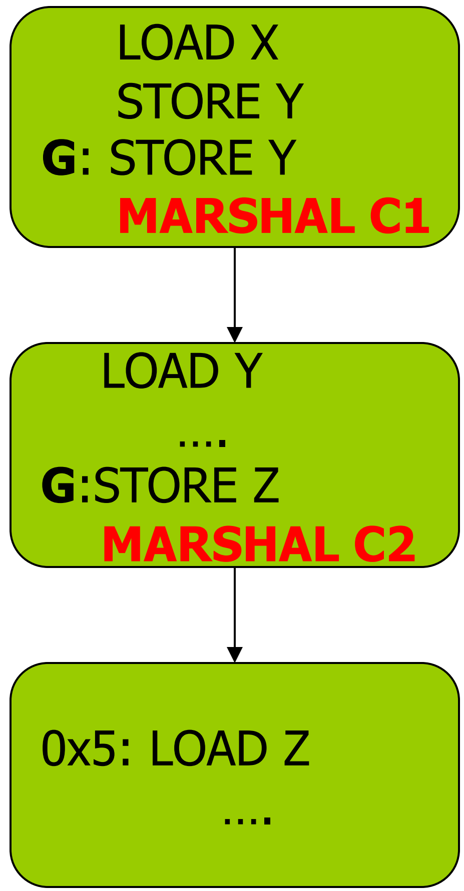
Profiling Algorithm

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

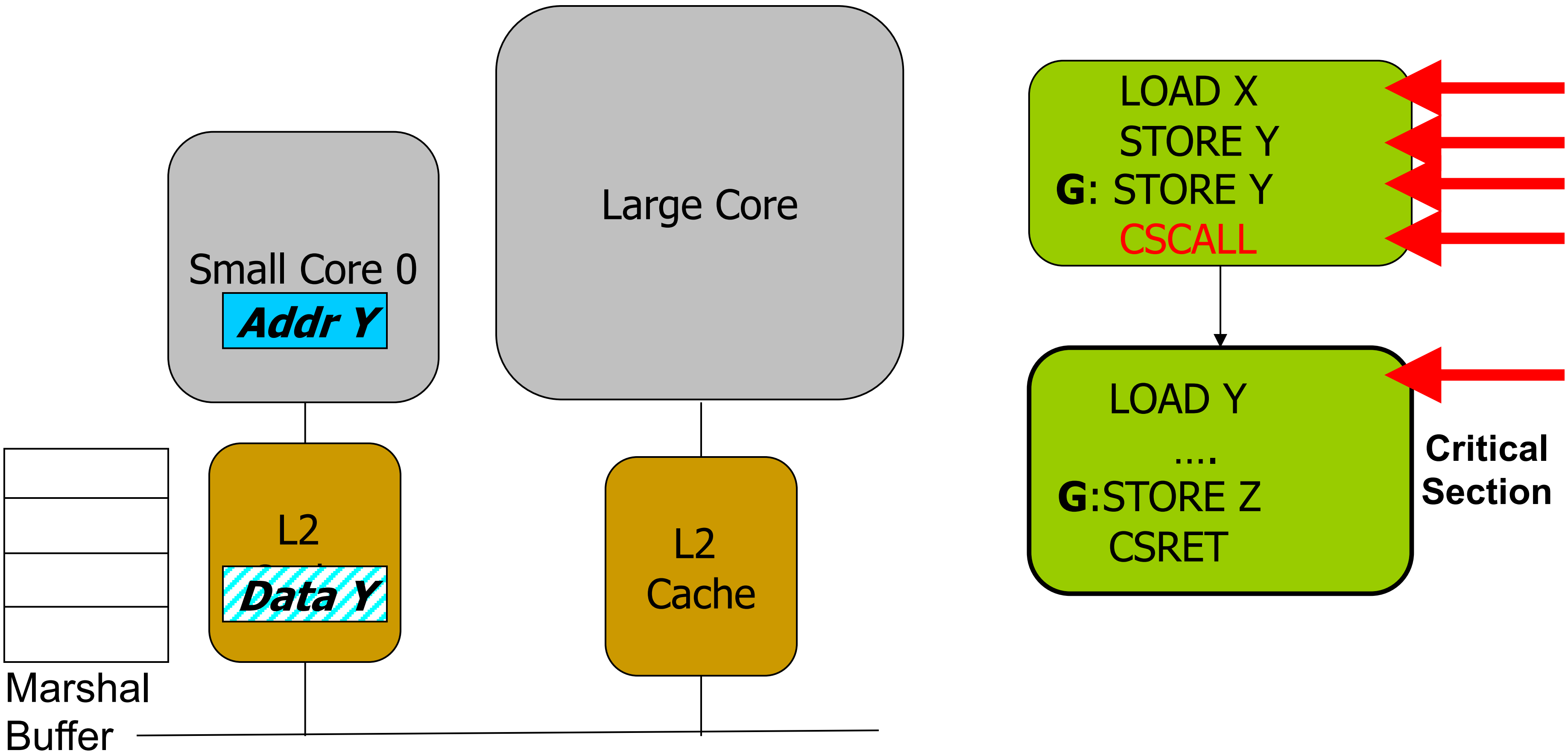
- Hardware
  - Marshal Buffer
    - Stores physical addresses of cache blocks to be marshaled
    - 16 entries enough for almost all workloads → 96 bytes per core
  - Ability to execute generator prefixes and marshal instructions
  - Ability to push data to another cache

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



Cache Hit!

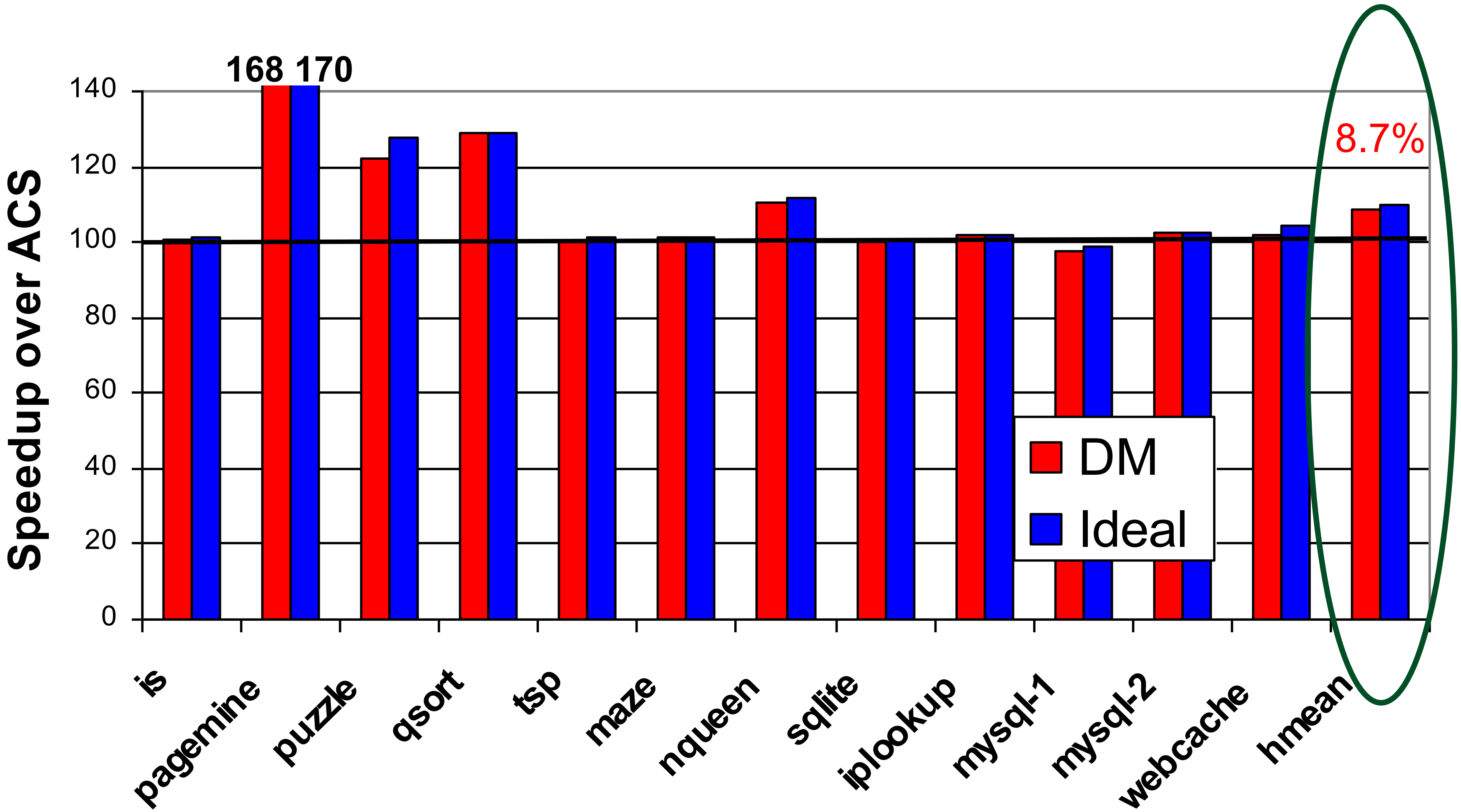
Accelerated Critical Sections: Methodology

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

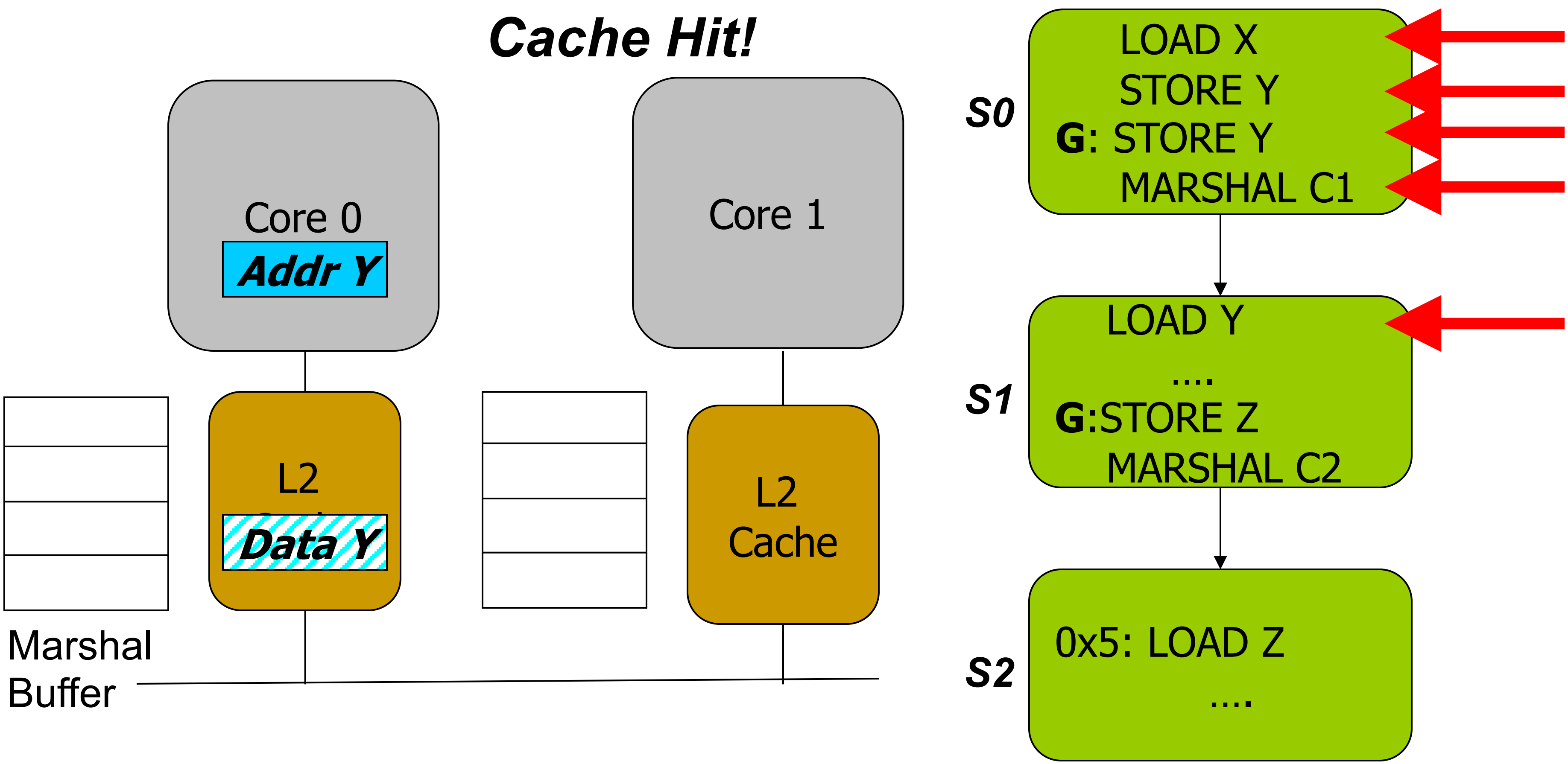
- Multi-core x86 simulator
  - 1 large and 28 small cores
  - Aggressive stream prefetcher employed at each core

- 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

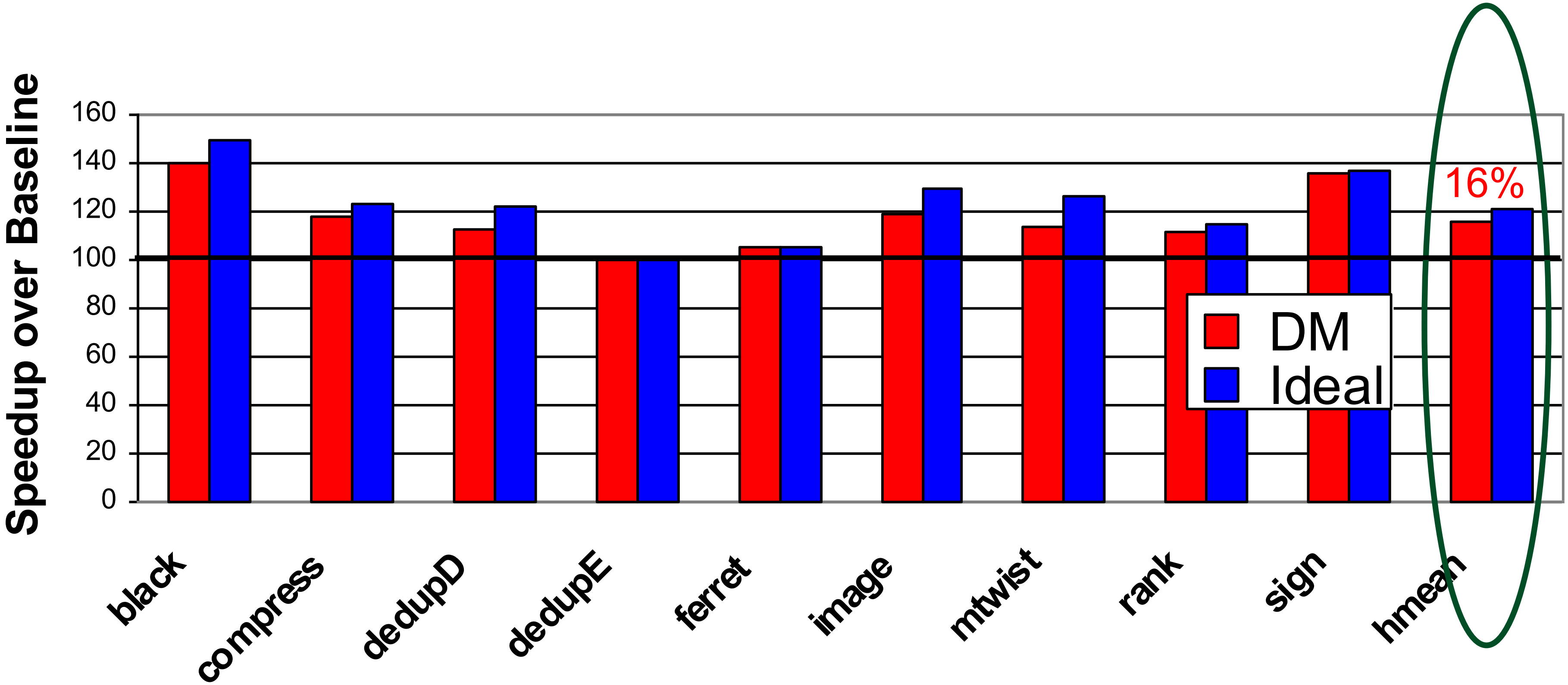


Pipeline Parallelism: Methodology

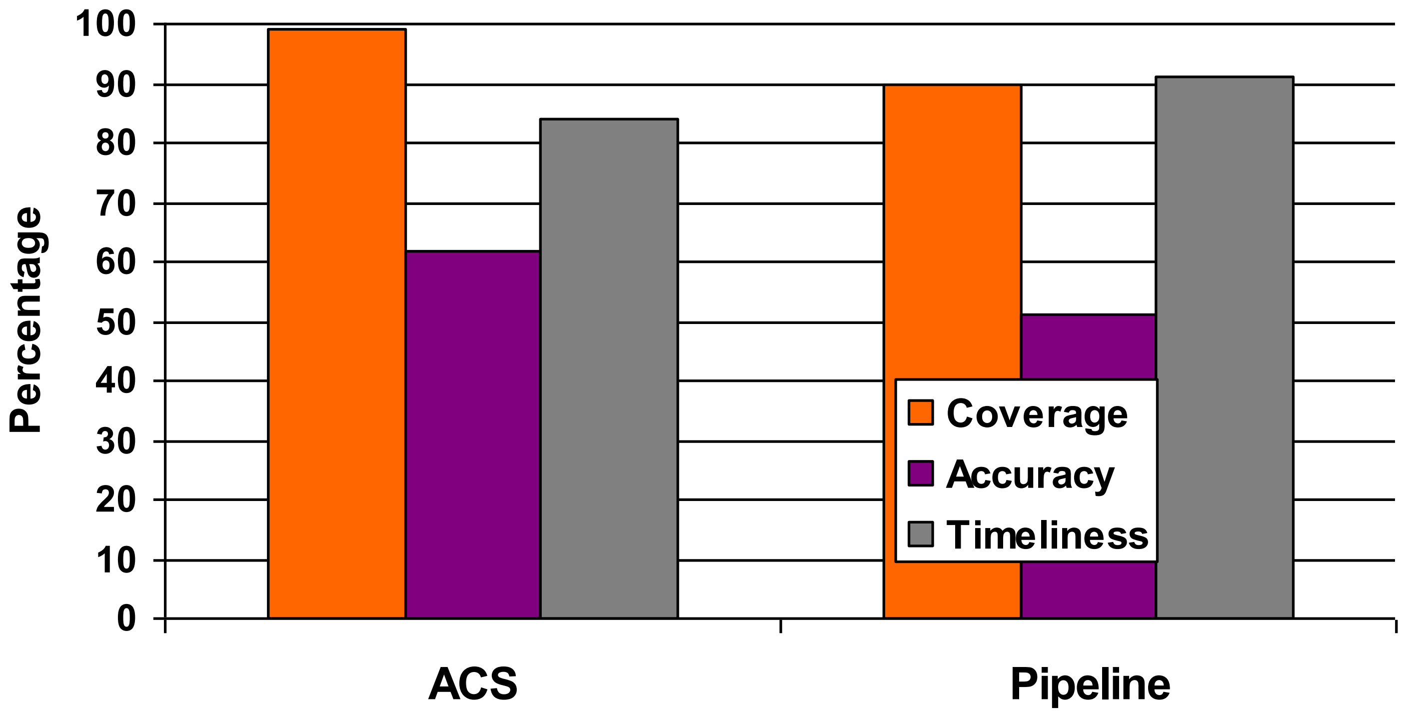
- Workloads: 9 applications with pipeline parallelism
  - Financial, compression, multimedia, encoding/decoding
  - Different training and simulation input sets

- 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

More on Bottleneck Identification & Scheduling

- 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)

Data Marshaling for Multi-core Architectures

M. Aater Suleman† Onur Mutlu§ José A. Joao† Khubaib† Yale N. Patt†

†The University of Texas at Austin
{suleman, joao, khubaib, patt}@hps.utexas.edu

§Carnegie Mellon University
onur@cmu.edu

Other Uses of Asymmetry

Use of Asymmetry for Energy Efficiency

- Kumar et al., "Single-ISA Heterogeneous Multi-Core Architectures: The Potential for Processor Power Reduction," MICRO 2003.

- Idea:
  - Implement multiple types of cores on chip
  - Monitor characteristics of the running thread (e.g., sample energy/perf on each core periodically)
  - Dynamically pick the core that provides the best energy/performance tradeoff for a given phase
    - "Best core" → Depends on optimization metric

Use of Asymmetry for Energy Efficiency

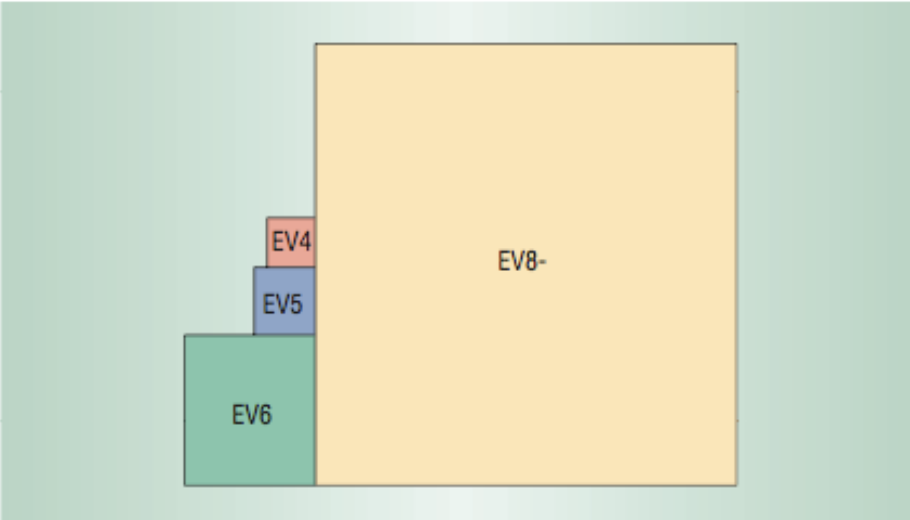
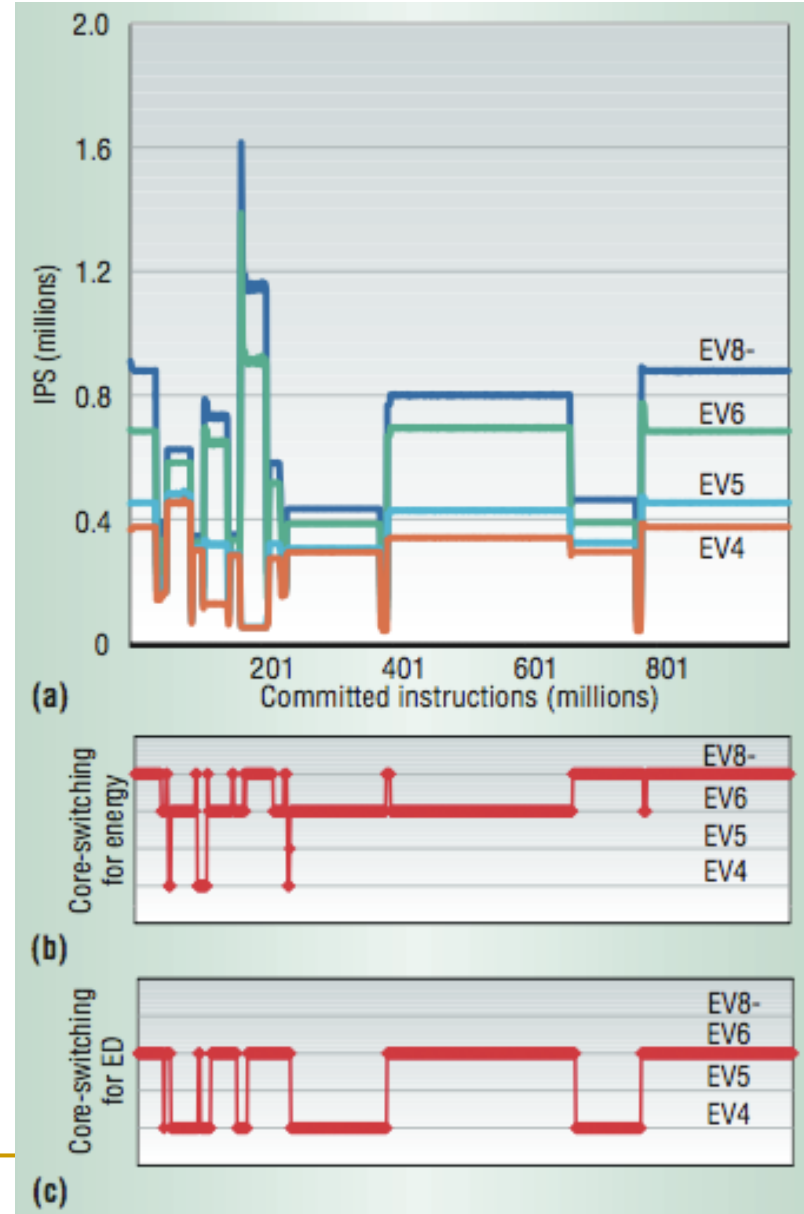


Figure 1. Relative sizes of the Alpha cores scaled to 0.10 µm. EV8 is 80 times bigger but provides only two to three times more single-threaded performance.

Table 1. Power and relative performance of Alpha cores scaled to 0.10 µm. Performance is expressed normalized to EV4 performance.

Core	Peak power (Watts)	Average power (Watts)	Performance (norm. IPC)
EV4	4.97	3.73	1.00
EV5	9.83	6.88	1.30
EV6	17.8	10.68	1.87
EV8	92.88	46.44	2.14



Use of Asymmetry for Energy Efficiency

- Advantages
  + More flexibility in energy-performance tradeoff
  + Can execute computation to the core that is best suited for it (in terms of energy)

- Disadvantages/issues
  - Incorrect predictions/sampling → wrong core → reduced performance or increased energy
  - Overhead of core switching
  - Disadvantages of asymmetric CMP (e.g., design multiple cores)
  - Need phase monitoring and matching algorithms
    - What characteristics should be monitored?
    - Once characteristics known, how do you pick the core?

Asymmetric vs. Symmetric Cores

- Advantages of Asymmetric
  + Can provide better performance when thread parallelism is limited
  + Can be more energy efficient
    + Schedule computation to the core type that can best execute it

- Disadvantages
  - Need to design more than one type of core. Always?
  - Scheduling becomes more complicated
    - What computation should be scheduled on the large core?
    - Who should decide? HW vs. SW?
  - Managing locality and load balancing can become difficult if threads move between cores (transparently to software)
  - Cores have different demands from shared resources

How to Achieve Asymmetry

- Static
  - Type and power of cores fixed at design time
  - Two approaches to design “faster cores”:
    - High frequency
    - Build a more complex, powerful core with entirely different uarch
  - Is static asymmetry natural? (chip-wide variations in frequency)

- Dynamic
  - Type and power of cores change dynamically
  - Two approaches to dynamically create “faster cores”:
    - Boost frequency dynamically (limited power budget)
    - Combine small cores to enable a more complex, powerful core
    - Is there a third, fourth, fifth approach?

Computer Architecture
Lecture 17:
Bottleneck Acceleration

Prof. Onur Mutlu
ETH Zürich
Fall 2020
20 November 2020

We Did Not Cover The Rest of the Slides. They Are For Your Benefit.

Asymmetry via Frequency Boosting

Asymmetry via Boosting of Frequency

- Static
  - Due to process variations, cores might have different frequency
  - Simply hardwire/design cores to have different frequencies
- Dynamic
  - Annavaram et al., "Mitigating Amdahl's Law Through EPI Throttling," ISCA 2005.
  - Dynamic voltage and frequency scaling

EPI Throttling

- Goal: Minimize execution time of parallel programs while keeping power within a fixed budget

- For best scalar and throughput performance, vary energy expended per instruction (EPI) based on available parallelism
  - P = EPI •IPS
  - P = fixed power budget
  - EPI = energy per instruction
  - IPS = aggregate instructions retired per second

- Idea: For a fixed power budget
  - Run sequential phases on high-EPI processor
  - Run parallel phases on multiple low-EPI processors

EPI Throttling via DVFS

- DVFS: Dynamic voltage frequency scaling

- In phases of low thread parallelism
  - Run a few cores at high supply voltage and high frequency

- In phases of high thread parallelism
  - Run many cores at low supply voltage and low frequency

Possible EPI Throttling Techniques

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

Method	EPI Range	Time to Alter EPI	Throttle Action
Voltage/frequency scaling	1:2 to 1:4	100us (ramp Vcc)	Lower voltage and frequency
Asymmetric cores	1:4 to 1:6	10us (migrate 256KB L2 cache)	Migrate threads from large cores to small cores
Variable-size core	1:1 to 1:2	1us (fill 32KB L1 cache)	Reduce capacity of processor resources
Speculation control	1:1 to 1:1.4	10ns (pipeline latency)	Reduce amount of speculation

Boosting Frequency of a Small Core vs. Large Core

- Frequency boosting implemented on Intel Nehalem, IBM POWER7

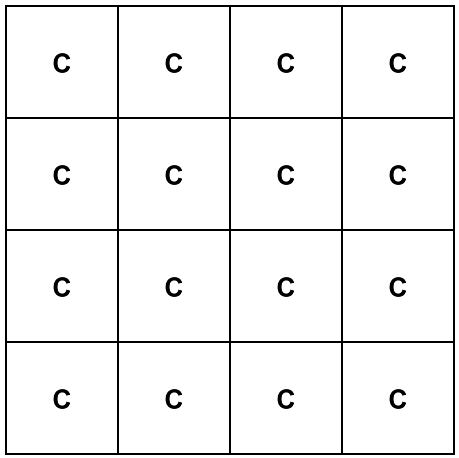
- Advantages of Boosting Frequency
  + Very simple to implement; no need to design a new core
  + Parallel throughput does not degrade when TLP is high
  + Preserves locality of boosted thread

- Disadvantages
  - Does not improve performance if thread is memory bound
  - Does not reduce Cycles per Instruction (remember the performance equation?)
  - Changing frequency/voltage can take longer than switching to a large core

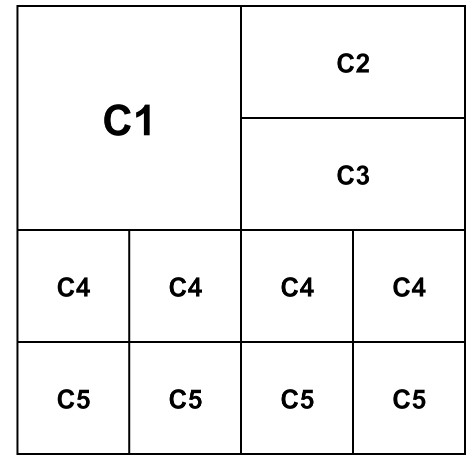
A Case for Asymmetry Everywhere

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

Asymmetry Enables Customization



Symmetric

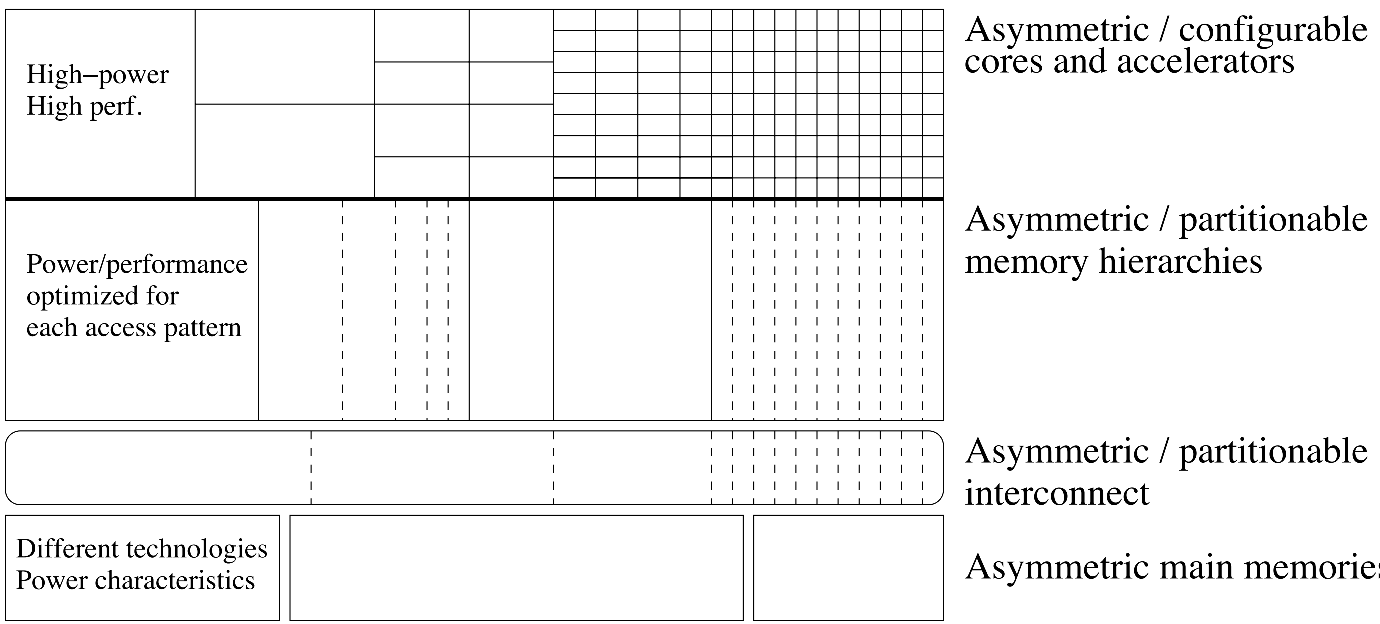


Asymmetric

- 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.)

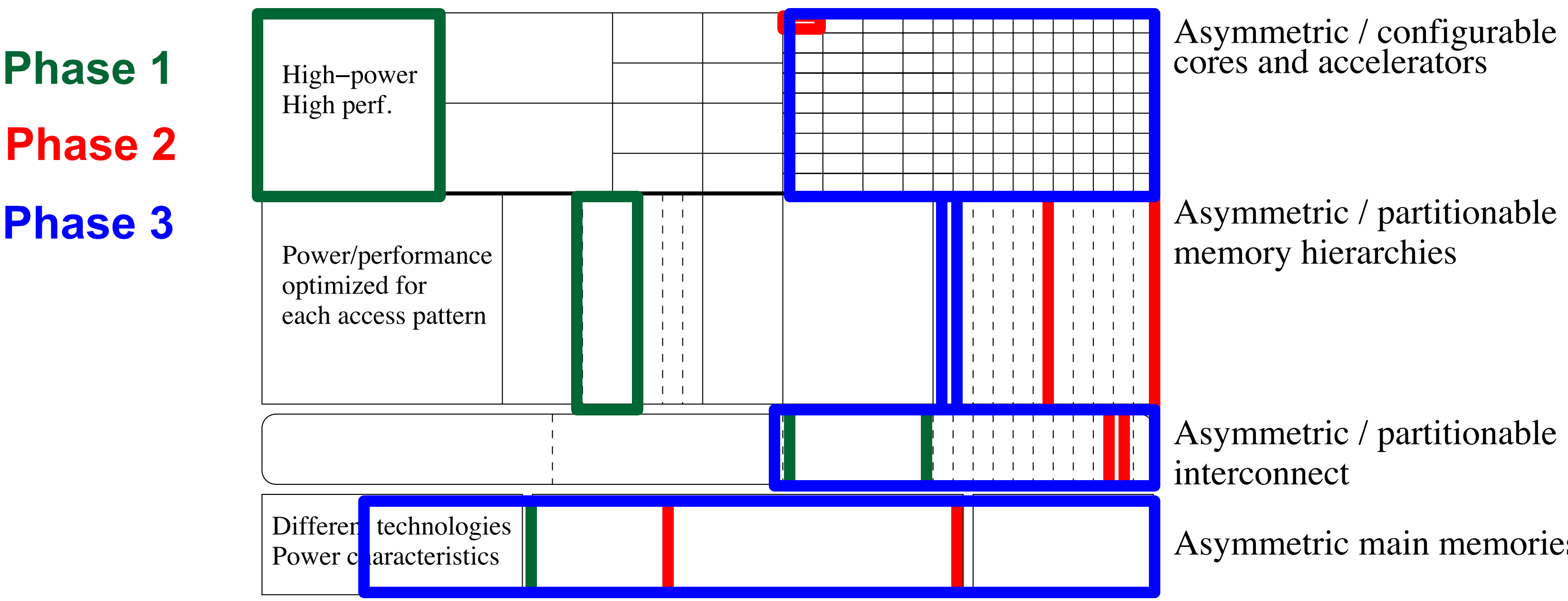
Thought Experiment: Asymmetry Everywhere

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



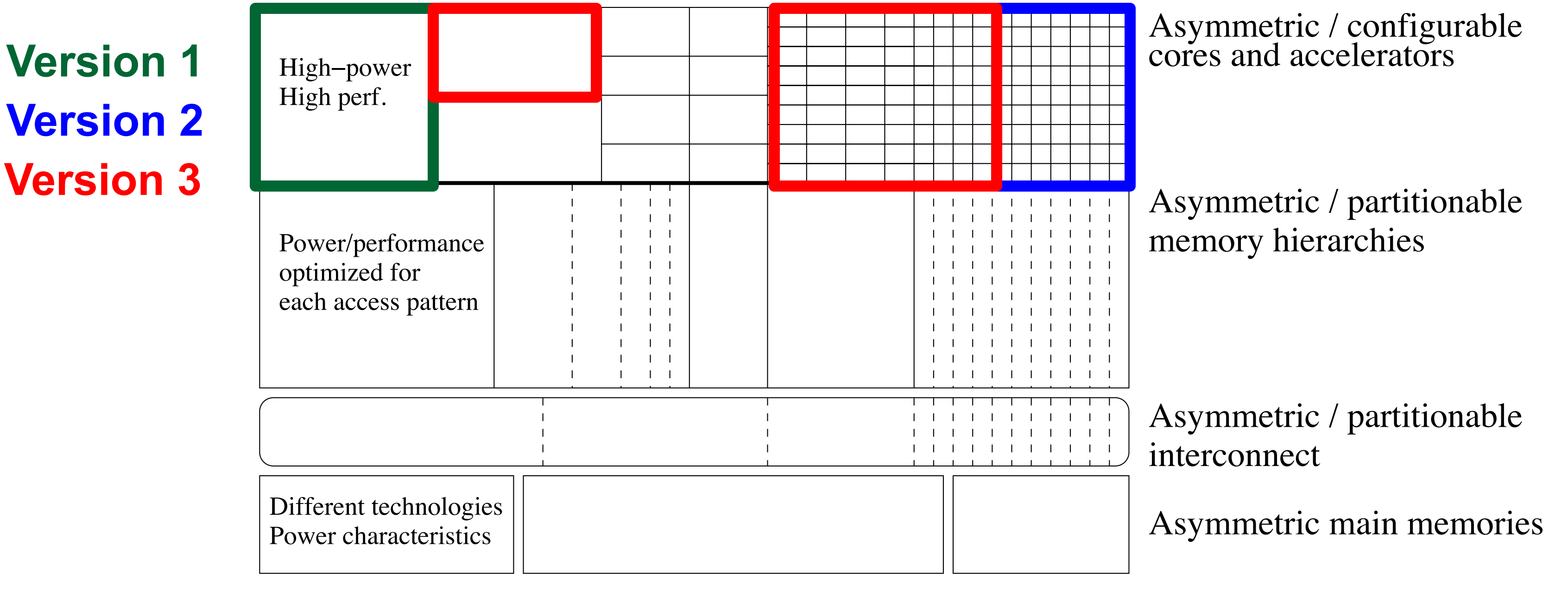
Thought Experiment: Asymmetry Everywhere

- 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

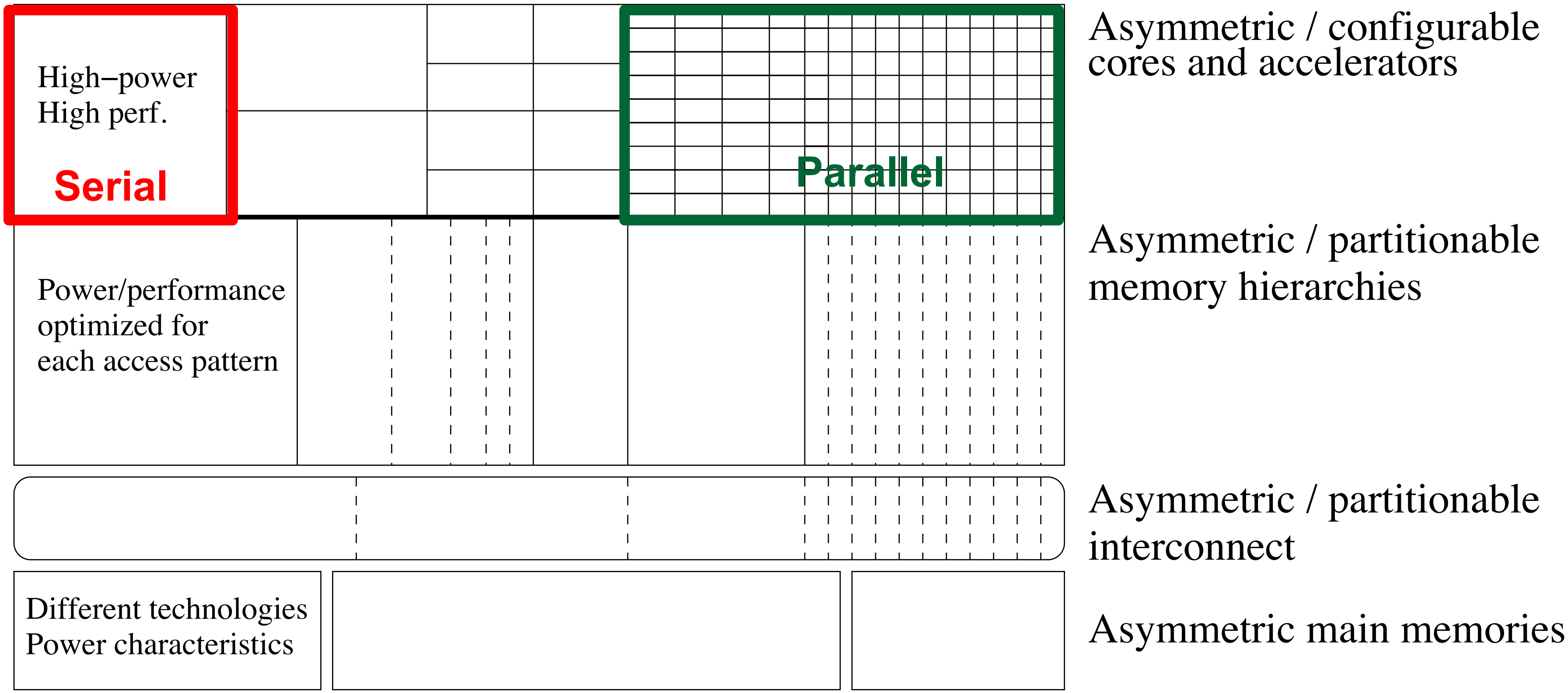
- How to design asymmetric components?
  - Fixed, partitionable, reconfigurable components?
  - What types of asymmetry? Access patterns, technologies?

- 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?

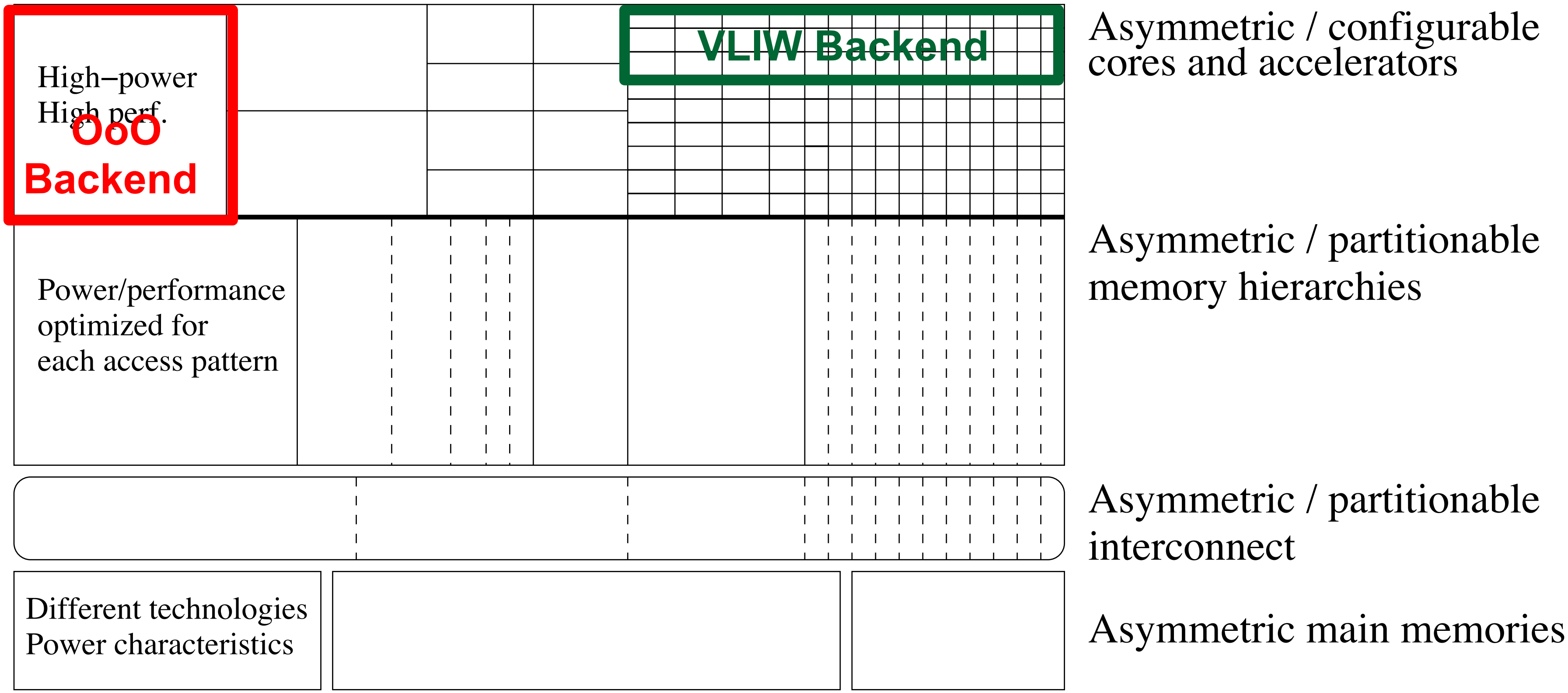
- How to design the runtime to automatically manage resources?
  - Track task behavior, pick "best-fit" components for the entire workload

Exploiting Asymmetry: Simple Examples



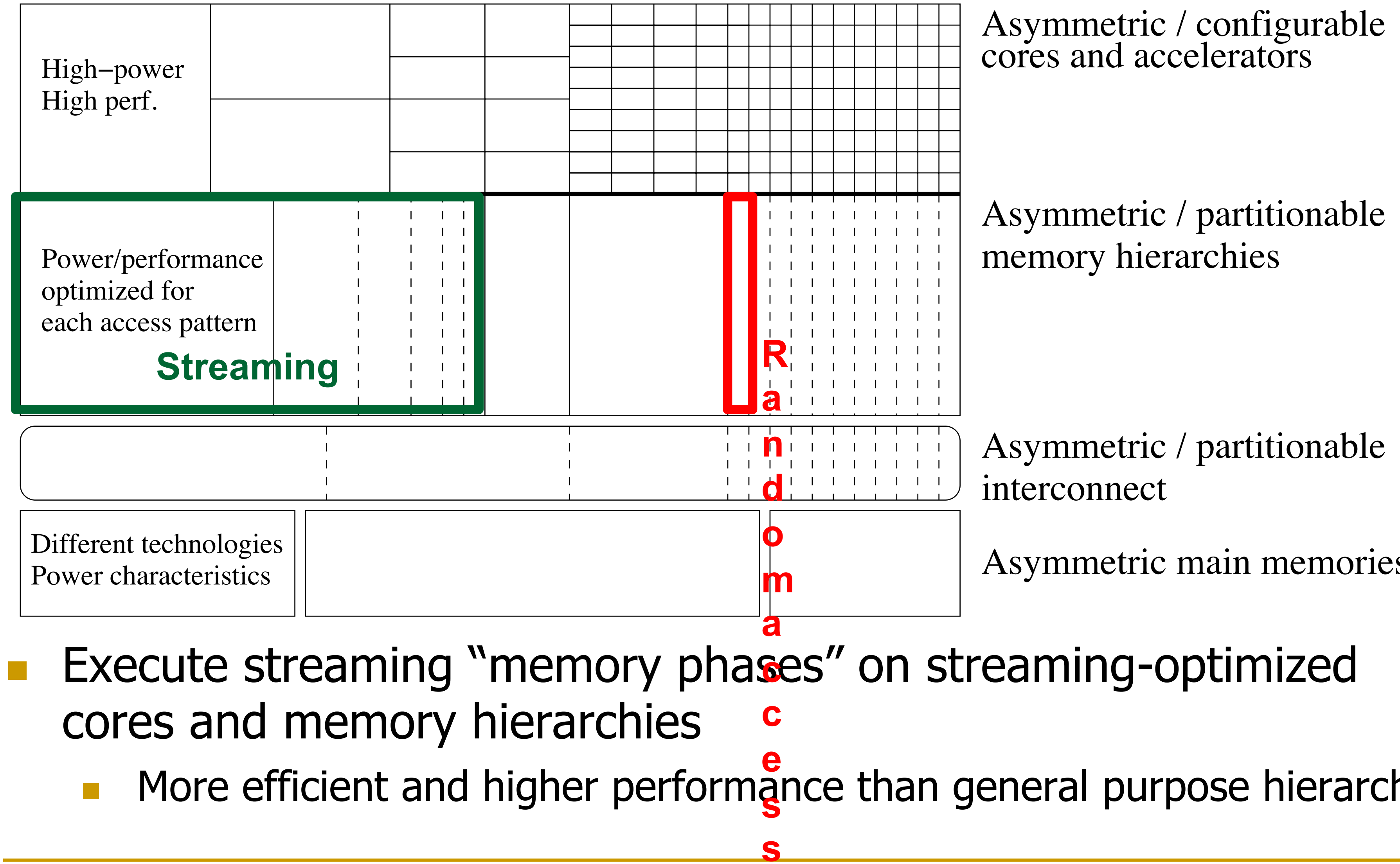
- Execute critical/serial sections on high-power, high-performance cores/resources [Suleman+ ASPLOS'09, ISCA'10, Top Picks'10'11, Joao+ ASPLOS'12,ISCA'13]
  - Programmer can write less optimized, but more likely correct programs

Exploiting Asymmetry: Simple Examples



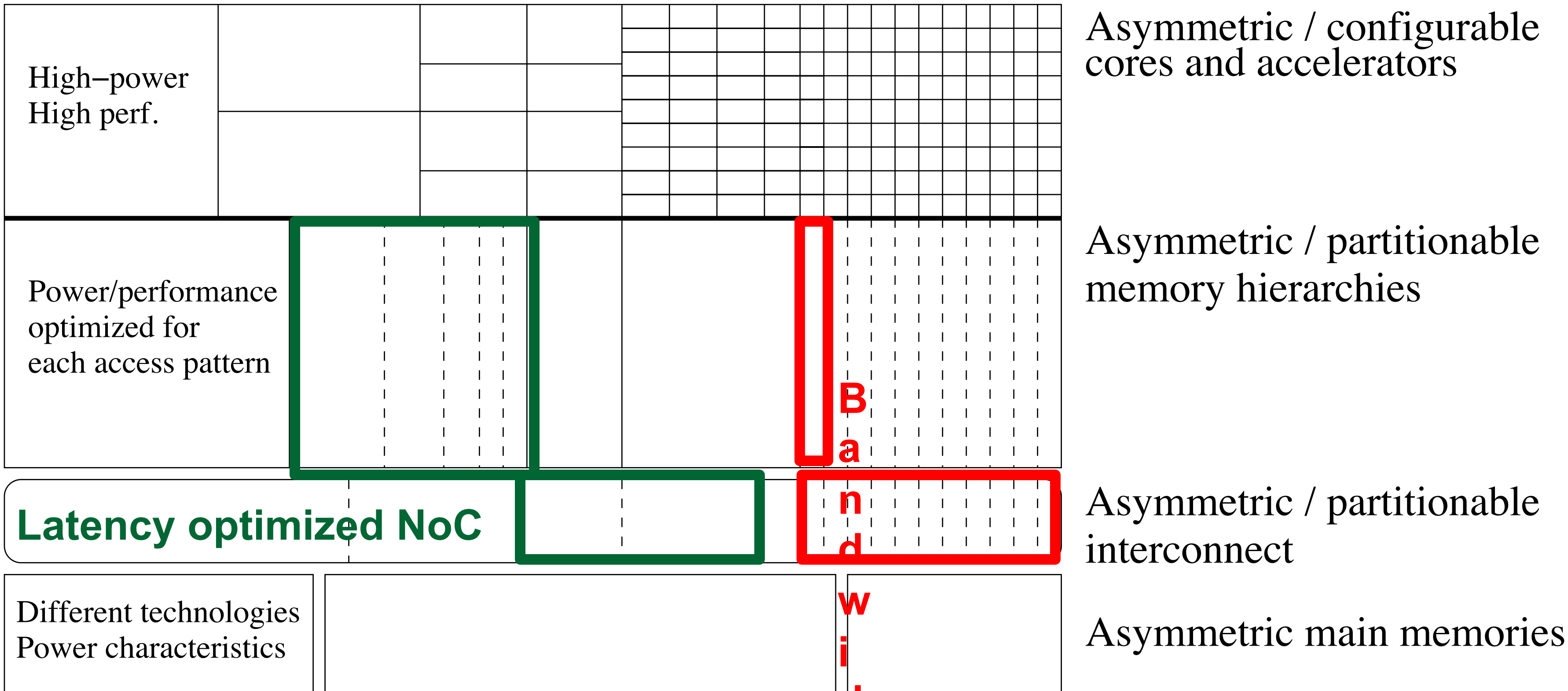
- Execute each code block on the most efficient execution backend for that block **[Fallin+ ICCD'14]**
  - Enables a much more efficient and still high performance core design

Exploiting Asymmetry: Simple Examples



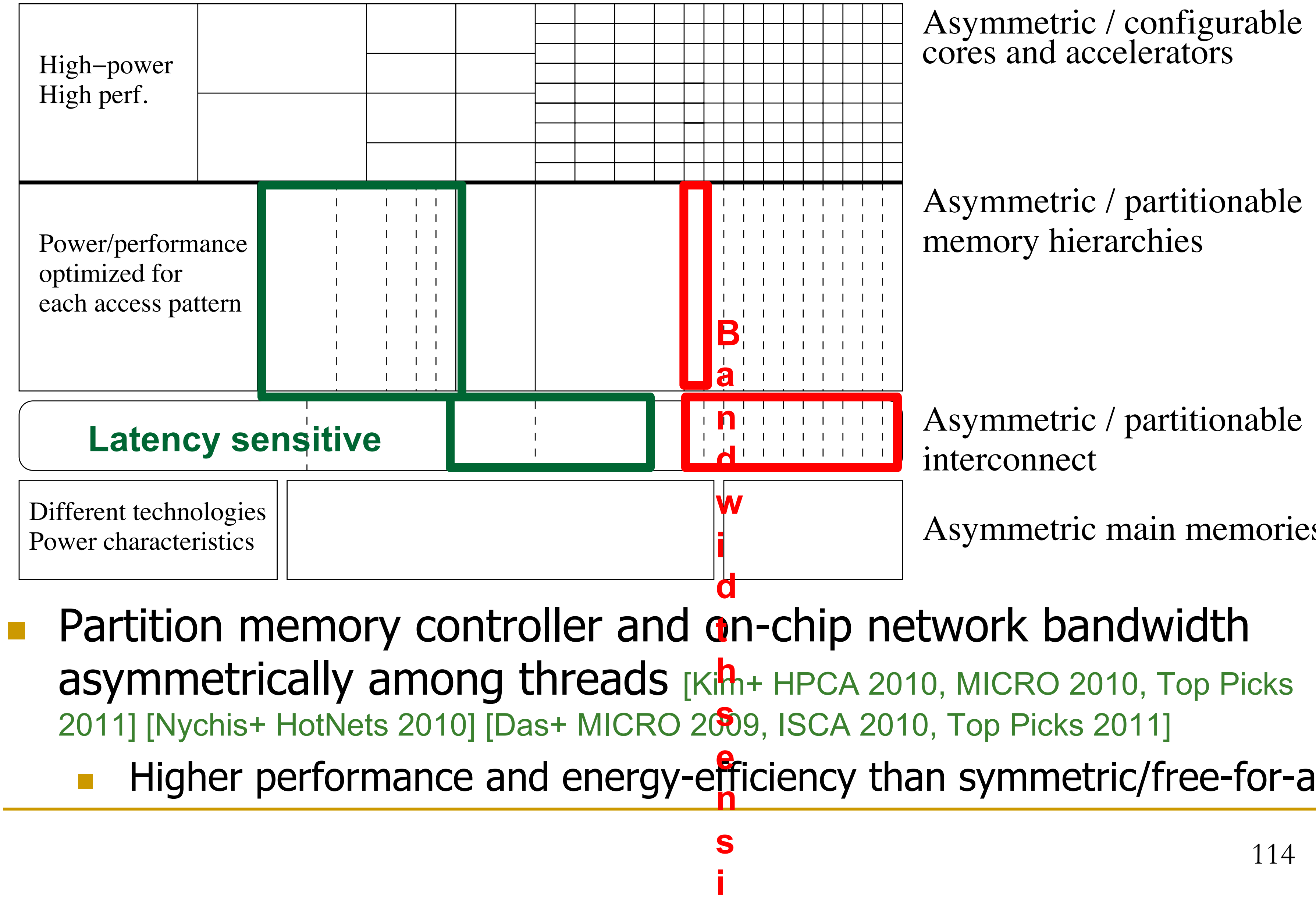
- Execute streaming "memory phases" on streaming-optimized cores and memory hierarchies
  - More efficient and higher performance than general purpose hierarchy

Exploiting Asymmetry: Simple Examples



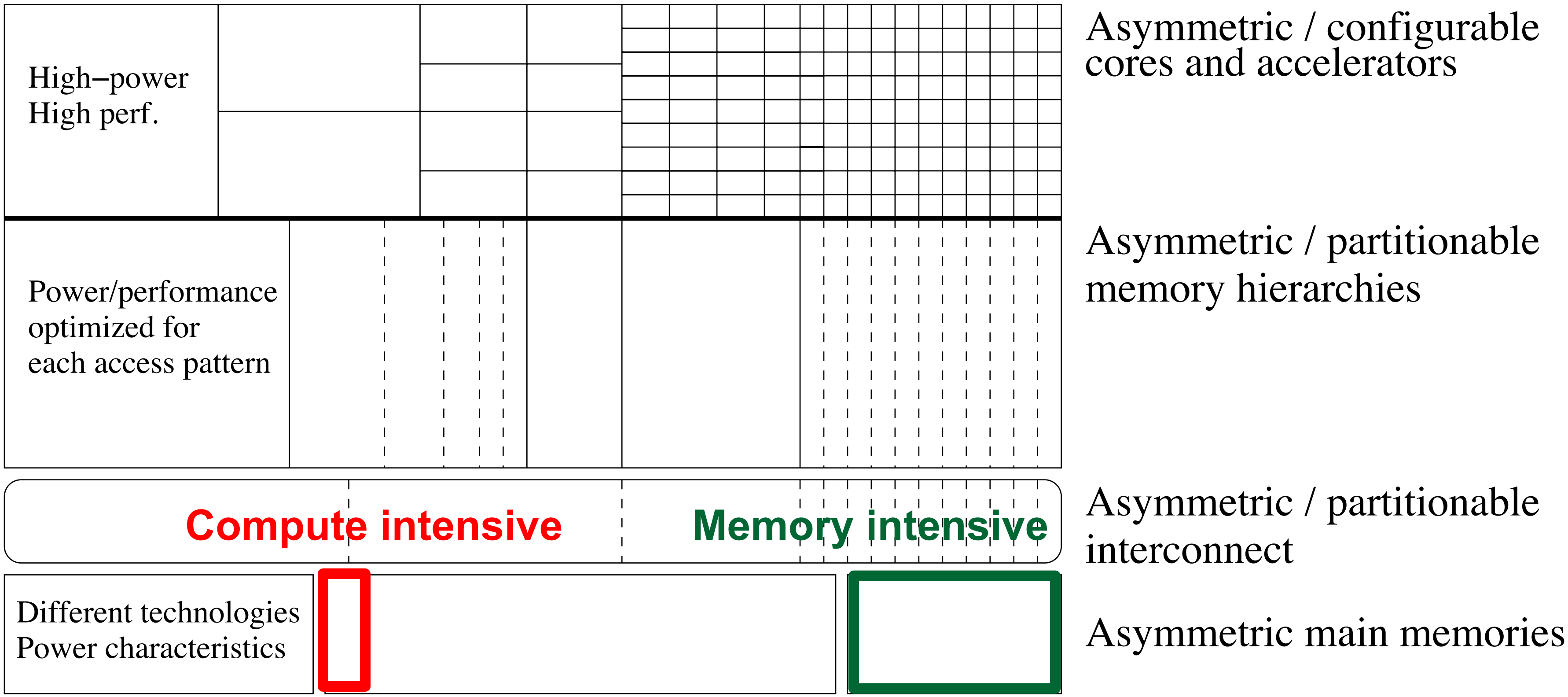
- Execute bandwidth-sensitive threads on a bandwidth-optimized network, latency-sensitive ones on a latency-optimized network **[Das+ DAC'13]**
  - Higher performance and energy-efficiency than a single network

Exploiting Asymmetry: Simple Examples



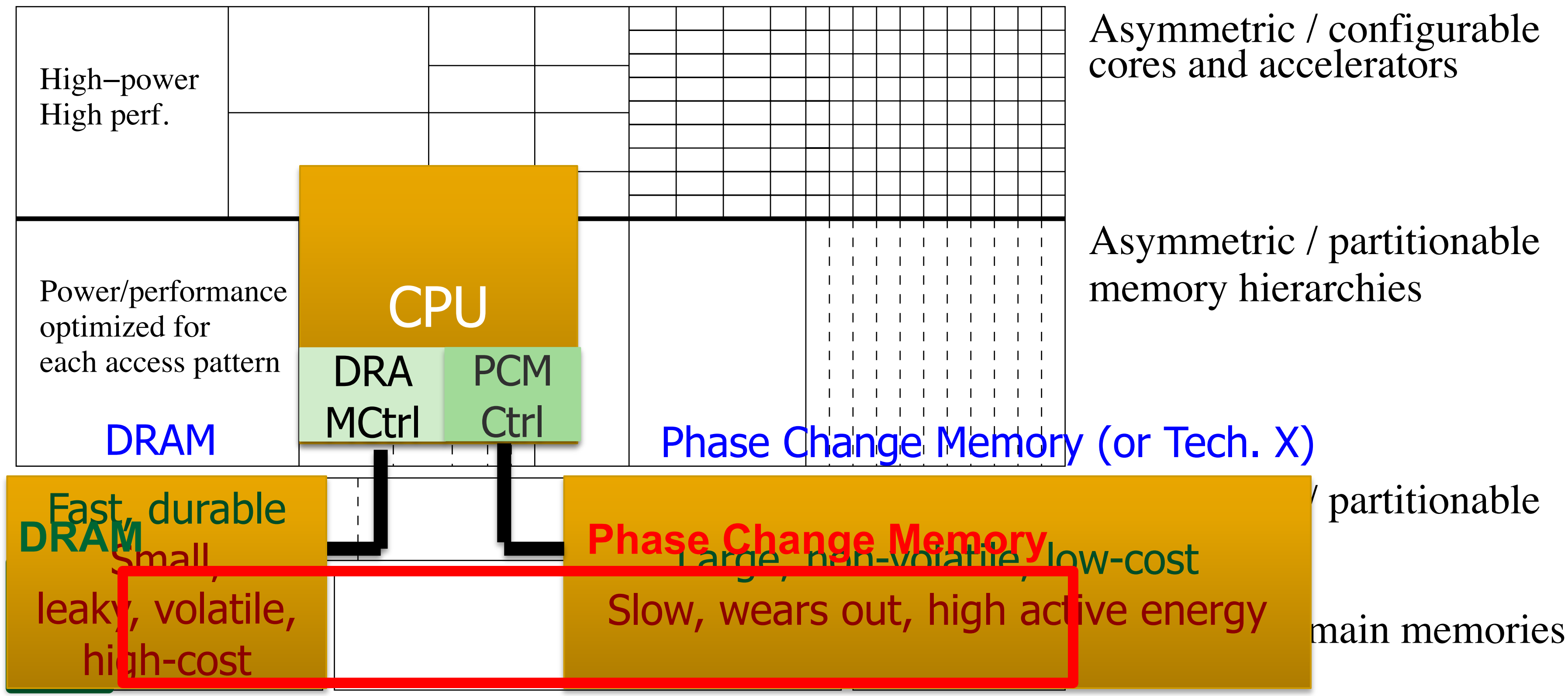
- Partition memory controller and on-chip network bandwidth asymmetrically among threads [Kim+ HPCA 2010, MICRO 2010, Top Picks 2011] [Nychis+ HotNets 2010] [Das+ MICRO 2009, ISCA 2010, Top Picks 2011]
  - Higher performance and energy-efficiency than symmetric/free-for-all

Exploiting Asymmetry: Simple Examples



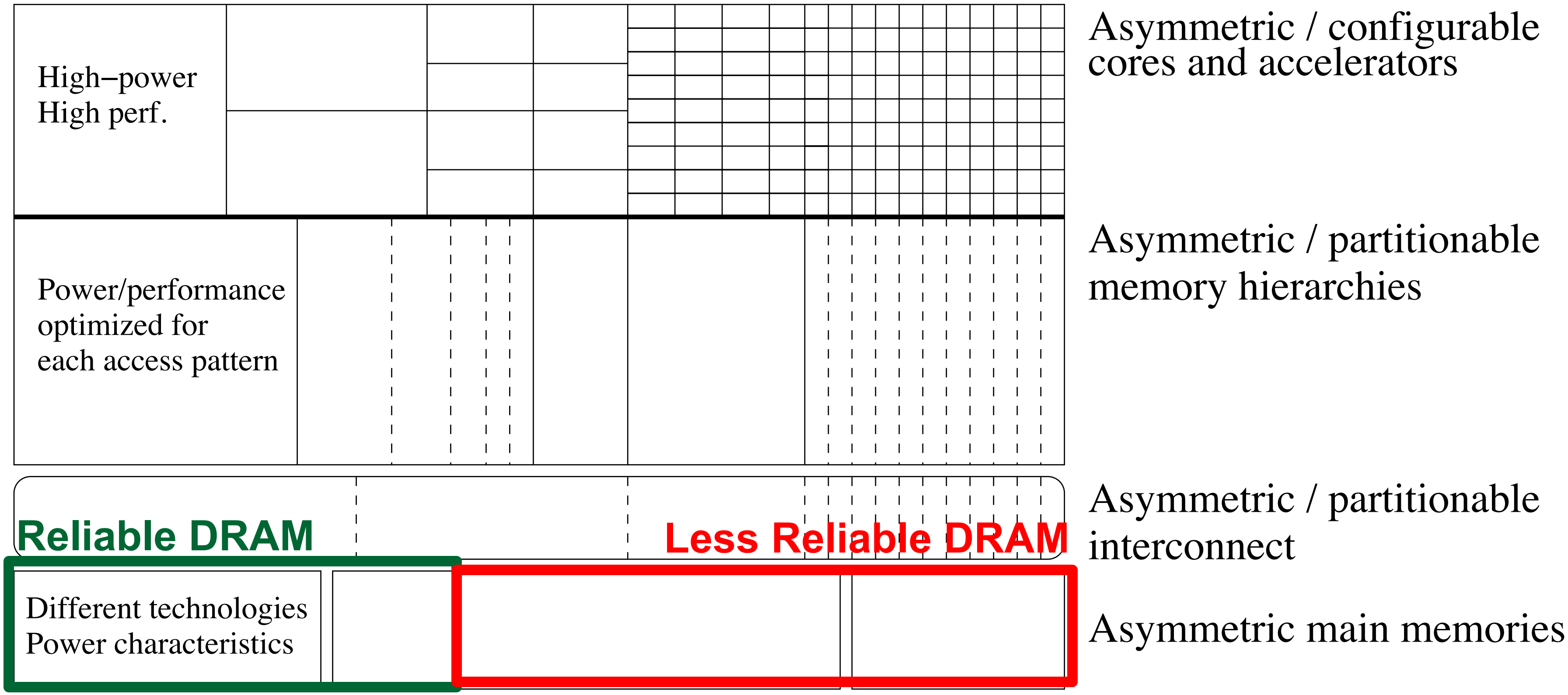
- Have multiple different memory scheduling policies apply them to different sets of threads based on thread behavior **[Kim+ MICRO 2010, Top Picks 2011] [Ausavarungnirun+ ISCA 2012]**
  - Higher performance and fairness than a homogeneous policy

Exploiting Asymmetry: Simple Examples



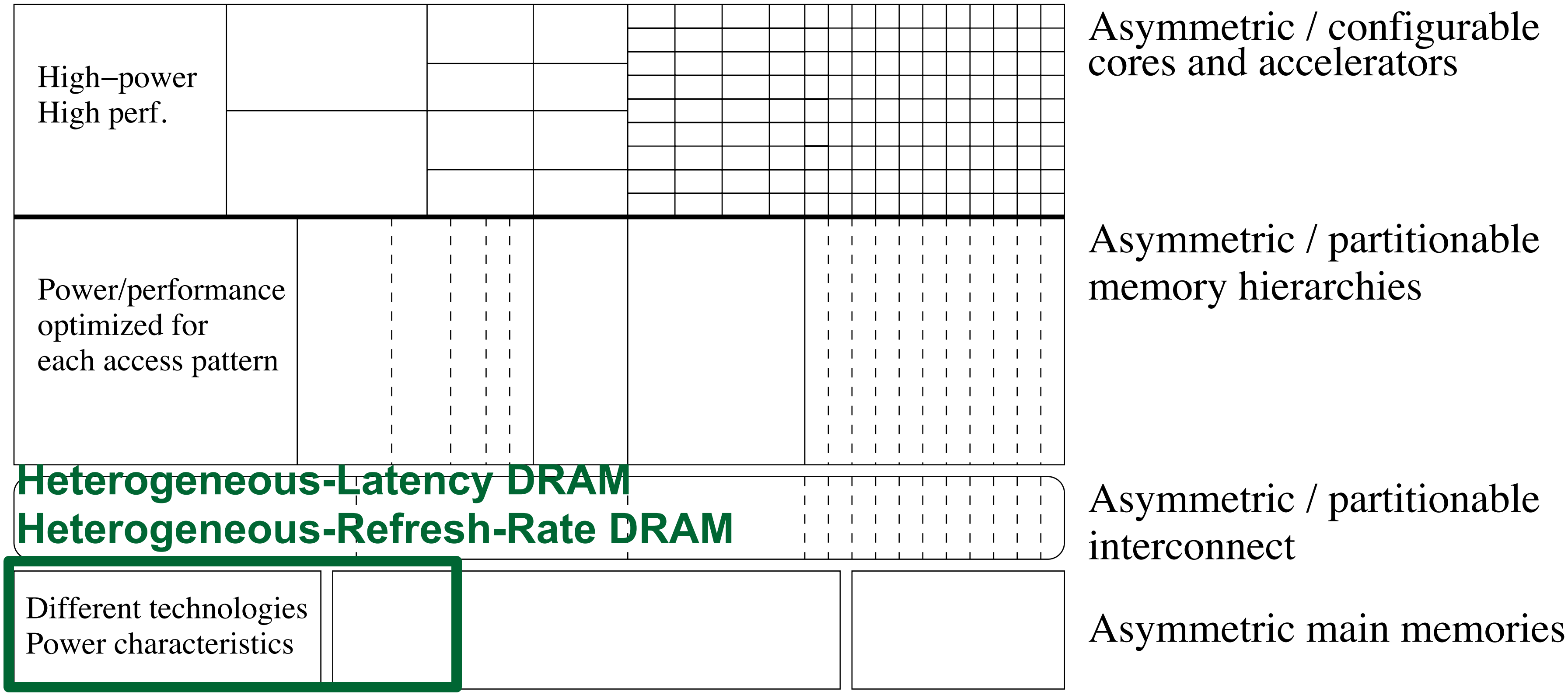
- Build main memory with different technologies with different characteristics (e.g., latency, bandwidth, cost, energy, reliability) **[Meza+ IEEE CAL'12, Yoon+ ICCD'12, Luo+ DSN'14]**
  - Higher performance and energy-efficiency than homogeneous memory

Exploiting Asymmetry: Simple Examples



- Build main memory with different technologies with different characteristics (e.g., latency, bandwidth, cost, energy, reliability) **[Meza+ IEEE CAL'12, Yoon+ ICCD'12, Luo+ DSN'14]**
  - Lower-cost than homogeneous-reliability memory at same availability

Exploiting Asymmetry: Simple Examples



- Design each memory chip to be heterogeneous to achieve low latency and low energy at reasonably low cost **[Lee+ HPCA'13, Liu+ ISCA'12]**
  - Higher performance and energy-efficiency than single-level memory