

# Computer Architecture

## Lecture 19b:

# Heterogeneous Multi-Core Systems

Prof. Onur Mutlu

ETH Zürich

Fall 2018

28 November 2018

# Computer Architecture Research

---

- If you want to do research in any of the covered topics or any topic in Comp Arch, HW/SW Interaction & related areas
  - We have many projects and a great environment to perform top-notch research, bachelor's/master's/semester projects
  - So, talk with me (email, in-person, WhatsApp, etc.)
- Many research topics and projects
  - Memory (DRAM, NVM, Flash, software/hardware issues)
  - Processing in Memory
  - Hardware Security
  - New Computing Paradigms
  - Machine Learning for System Design
  - Genomics, System Design for Bioinformatics and Health
  - ...

# Today and Tomorrow

---

- Heterogeneous Multi-Core Systems
- Bottleneck Acceleration

# Some Readings

---

- Suleman et al., “Accelerating Critical Section Execution with Asymmetric Multi-Core Architectures,” ASPLOS 2009.
- Joao et al., “Bottleneck Identification and Scheduling in Multithreaded Applications,” ASPLOS 2012.
- 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.
- Grochowski et al., “Best of Both Latency and Throughput,” ICCD 2004.



# Heterogeneity (Asymmetry)

# Heterogeneity (Asymmetry) → Specialization

---

- Heterogeneity and asymmetry have the same meaning
  - Contrast with homogeneity and symmetry
- Heterogeneity is a very general system design concept (and *life* concept, as well)
- Idea: Instead of having multiple instances of a “resource” to be the same (i.e., homogeneous or symmetric), design some instances to be different (i.e., heterogeneous or asymmetric)
- Different instances can be optimized to be more efficient in executing different types of workloads or satisfying different requirements/goals
  - Heterogeneity enables specialization/customization

# Why Asymmetry in Design? (I)

---

- Different workloads executing in a system can have different behavior
  - Different applications can have different behavior
  - Different execution phases of an application can have different behavior
  - The same application executing at different times can have different behavior (due to input set changes and dynamic events)
  - E.g., locality, predictability of branches, instruction-level parallelism, data dependencies, serial fraction in a parallel program, bottlenecks in parallel portion of a program, interference characteristics, ...
- Systems are designed to satisfy different metrics at the same time
  - There is almost never a single goal in design, depending on design point
  - E.g., Performance, energy efficiency, fairness, predictability, reliability, availability, cost, memory capacity, latency, bandwidth, ...

# Why Asymmetry in Design? (II)

---

- Problem: Symmetric design is one-size-fits-all
- It tries to fit a single-size design to all workloads and metrics
- It is very difficult to come up with a single design
  - that satisfies all workloads even for a single metric
  - that satisfies all design metrics at the same time
- This holds true for different system components, or resources
  - Cores, caches, memory, controllers, interconnect, disks, servers, ...
  - Algorithms, policies, ...

# Asymmetry Enables Customization

---

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

Symmetric

C1		C2	
		C3	
C4	C4	C4	C4
C5	C5	C5	C5

Asymmetric

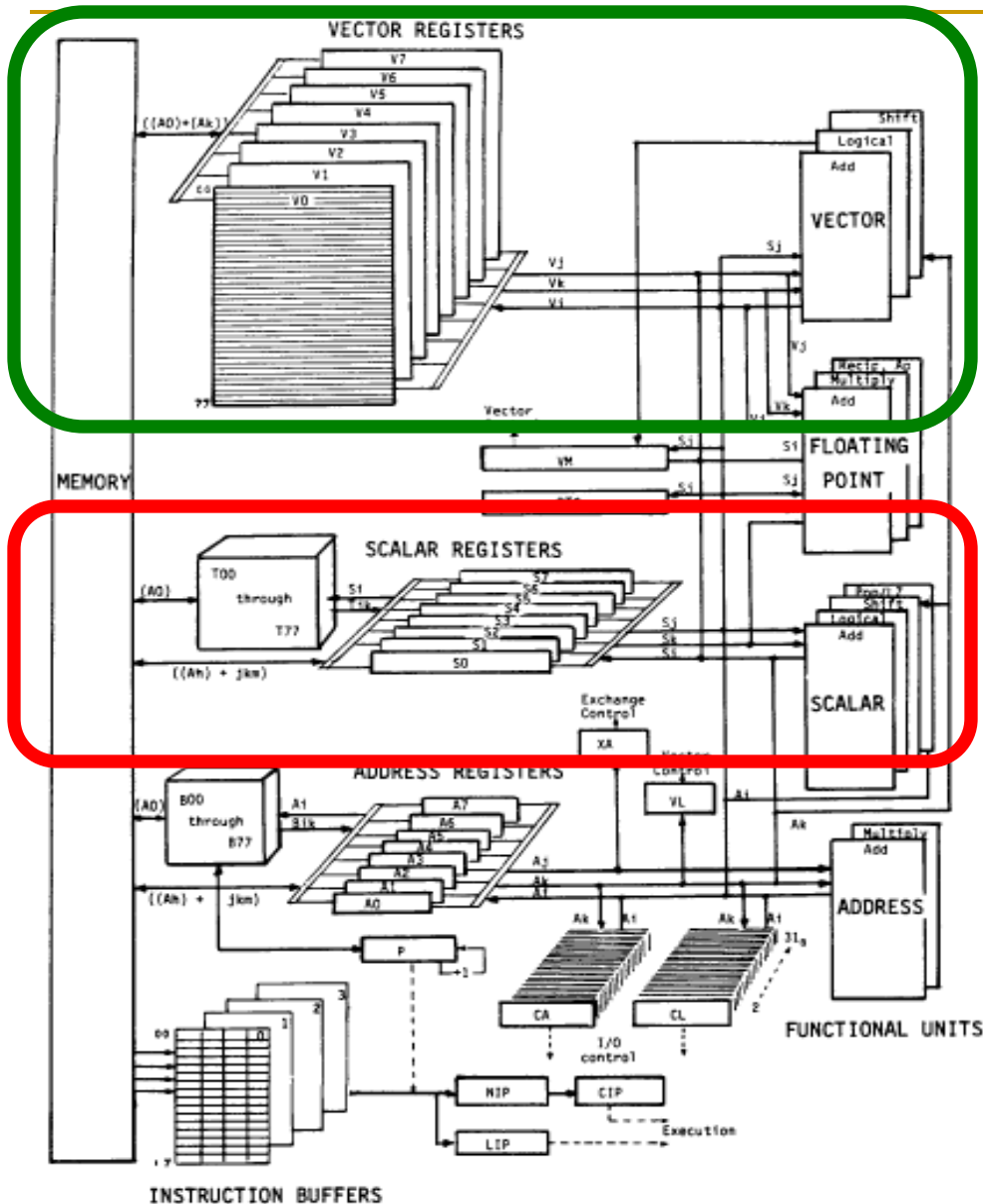
- Symmetric: One size fits all
  - Energy and performance suboptimal for different “workload” behaviors
- Asymmetric: Enables customization and adaptation
  - Processing requirements vary across workloads (applications and phases)
  - Execute code on best-fit resources (minimal energy, adequate perf.)

# We Have Already Seen Examples (Before)

---

- CRAY-1 design: scalar + vector pipelines
  - Modern processors: scalar instructions + SIMD extensions
  - Decoupled Access Execute: access + execute processors
  - Thread Cluster Memory Scheduling: different memory scheduling policies for different thread clusters
  - RAIDR: Heterogeneous refresh rates in DRAM
  - Heterogeneous-Latency DRAM (Tiered Latency DRAM)
  - Hybrid memory systems
    - DRAM + Phase Change Memory
    - Fast, Costly DRAM + Slow, Cheap DRAM
    - Reliable, Costly DRAM + Unreliable, Cheap DRAM
  - Heterogeneous cache replacement policies
-

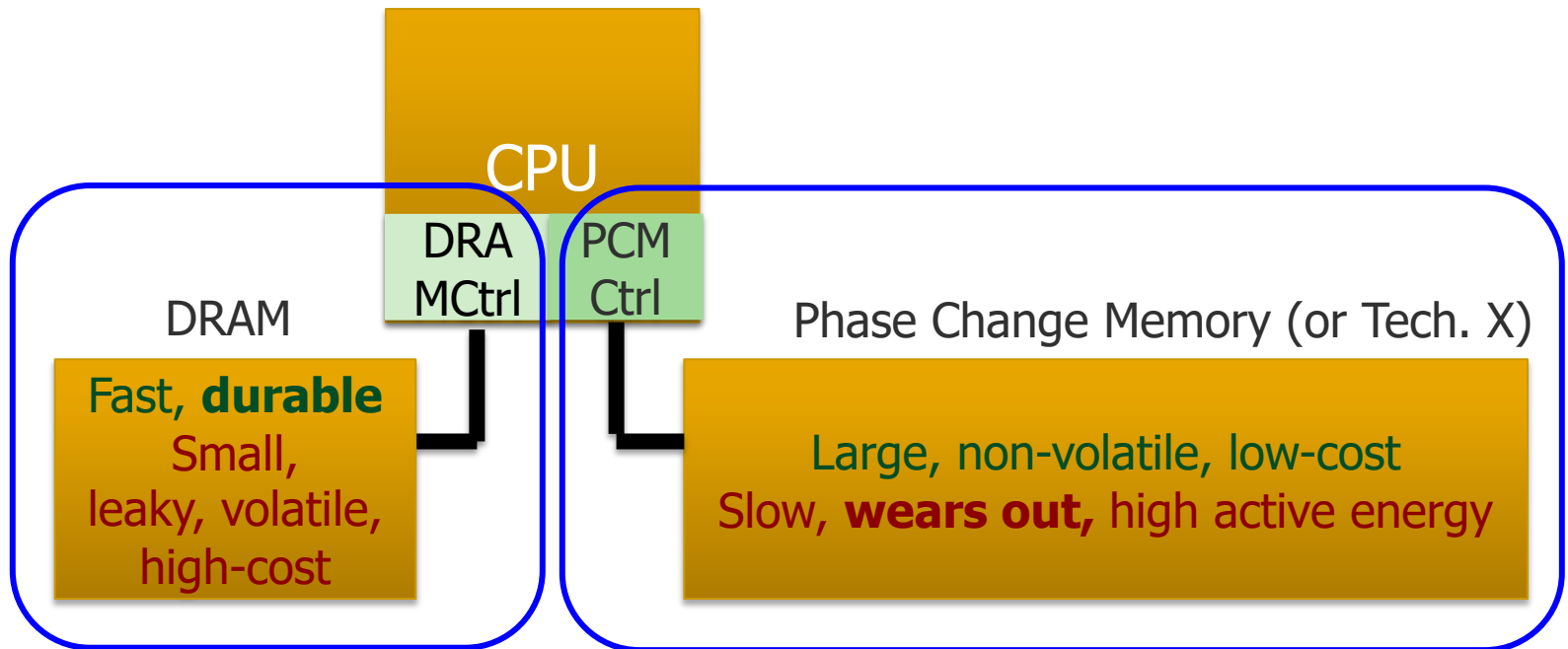
# An Example Asymmetric Design: CRAY-1



- 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

# Remember: Hybrid Memory Systems

---



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

Meza+, "Enabling Efficient and Scalable Hybrid Memories," IEEE Comp. Arch. Letters, 2012.  
Yoon, Meza et al., "Row Buffer Locality Aware Caching Policies for Hybrid Memories," ICCD 2012 Best Paper Award.

---

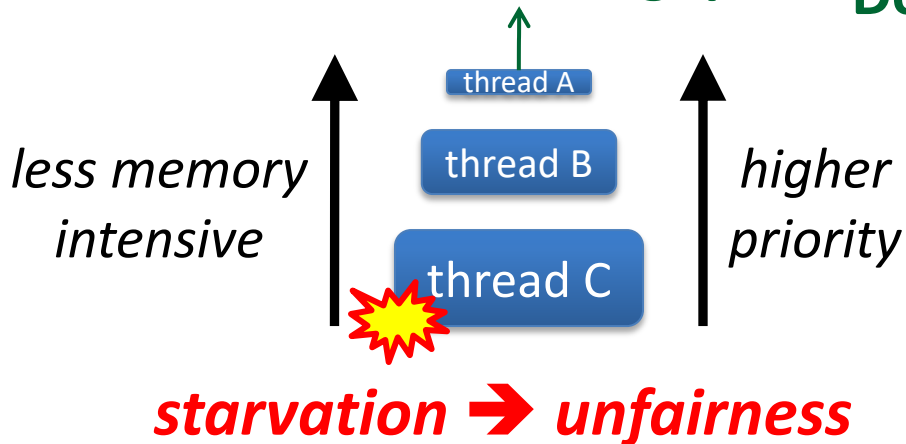


# Remember: Throughput vs. Fairness

## *Throughput biased approach*

Prioritize less memory-intensive threads

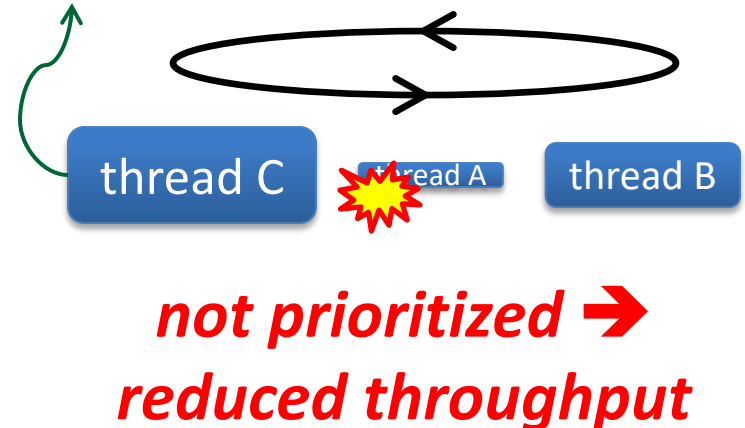
**Good for throughput**



## *Fairness biased approach*

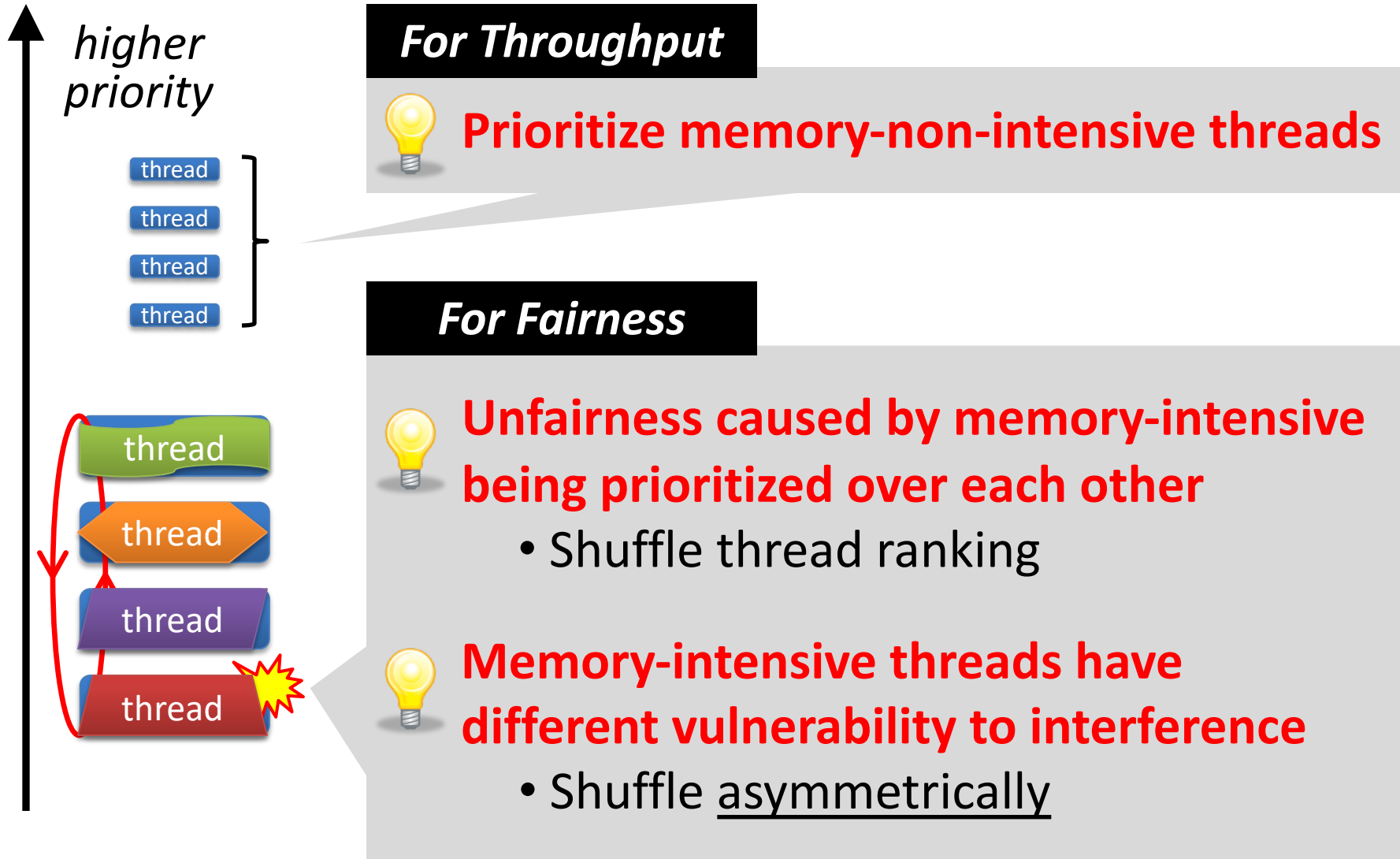
Take turns accessing memory

**Does not starve**



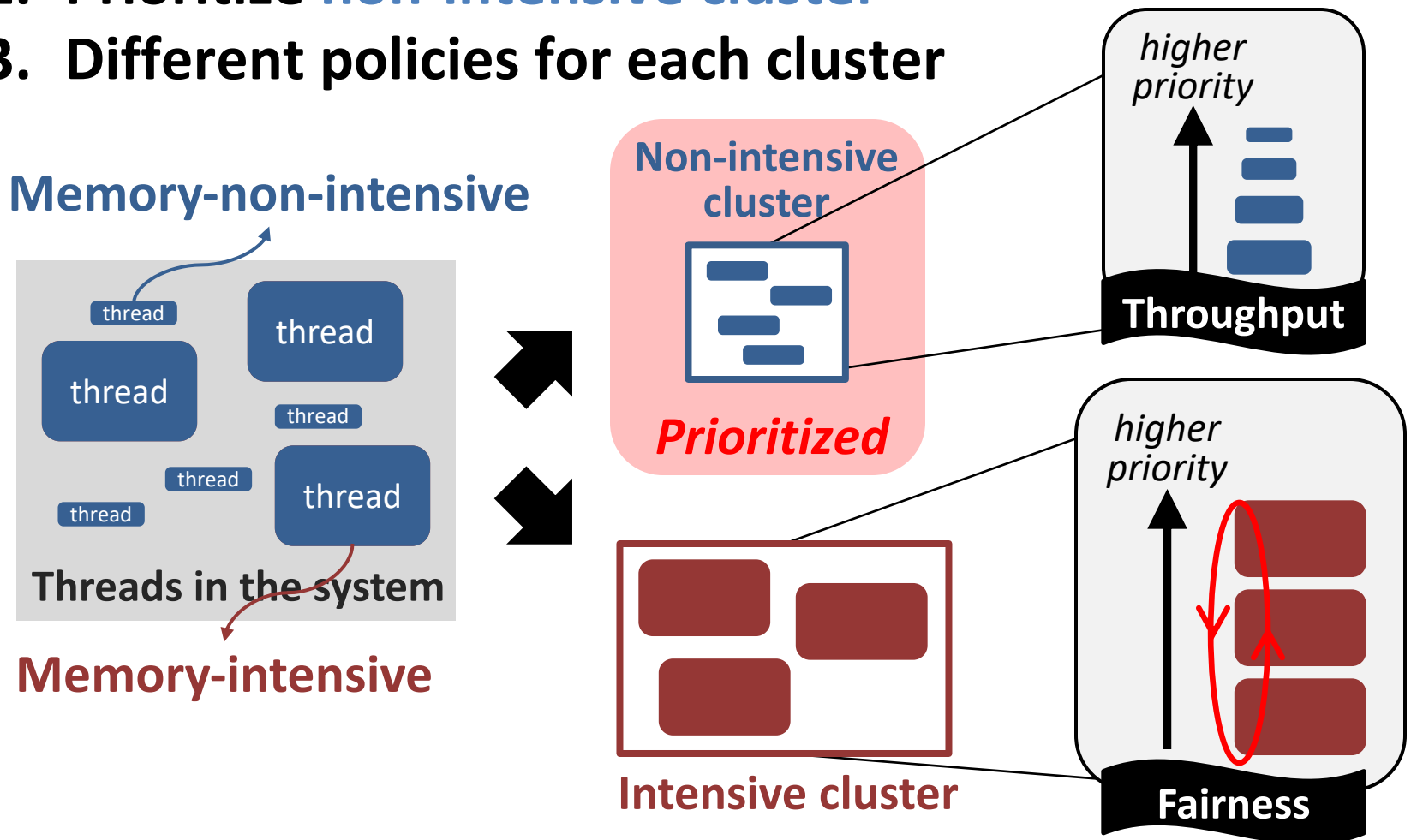
**Single policy for all threads is insufficient**

# Remember: Achieving the Best of Both Worlds



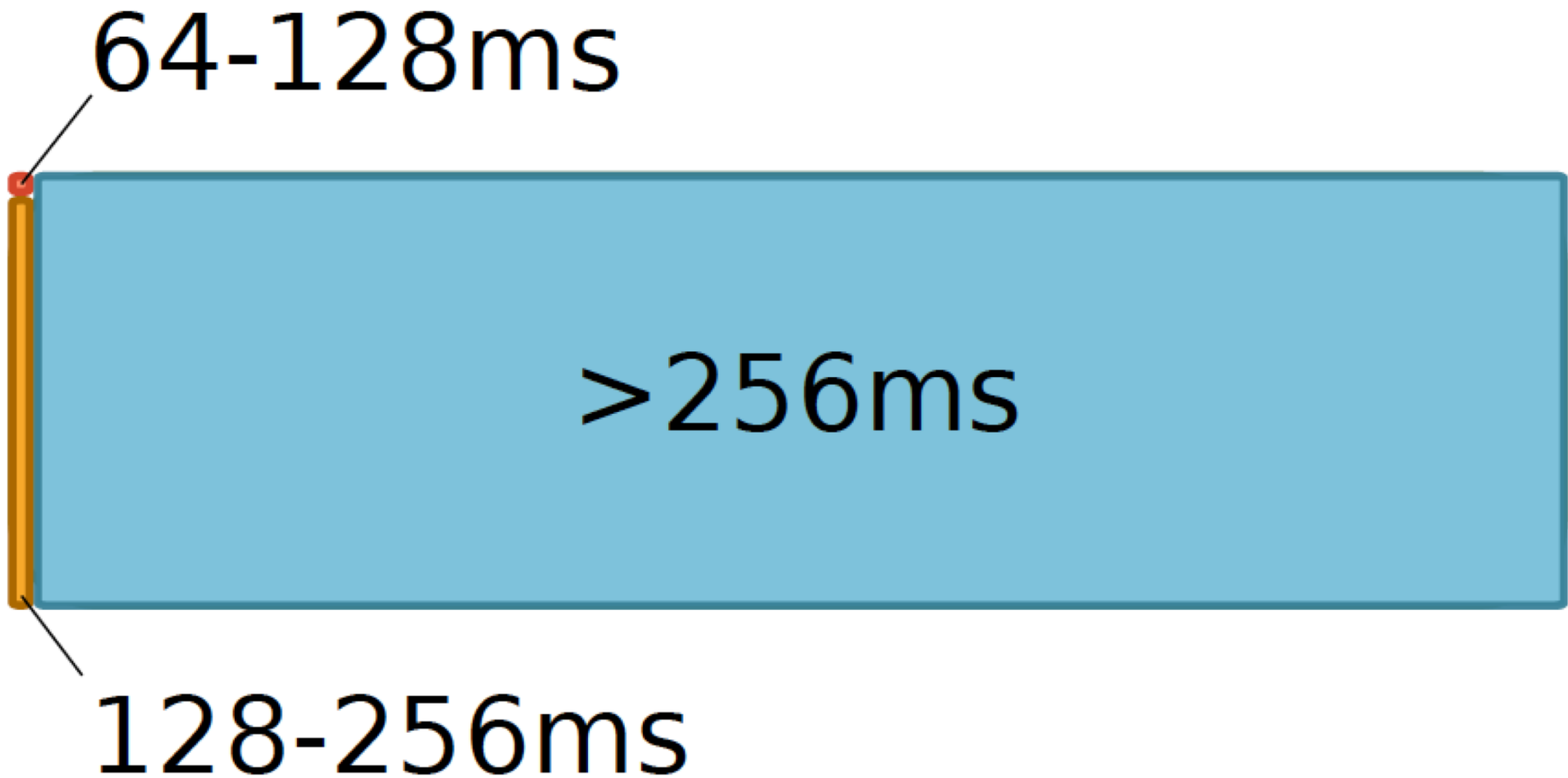
# Thread Cluster Memory Scheduling [Kim+ MICRO'10]

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



# Remember: Heterogeneous Retention Times in DRAM

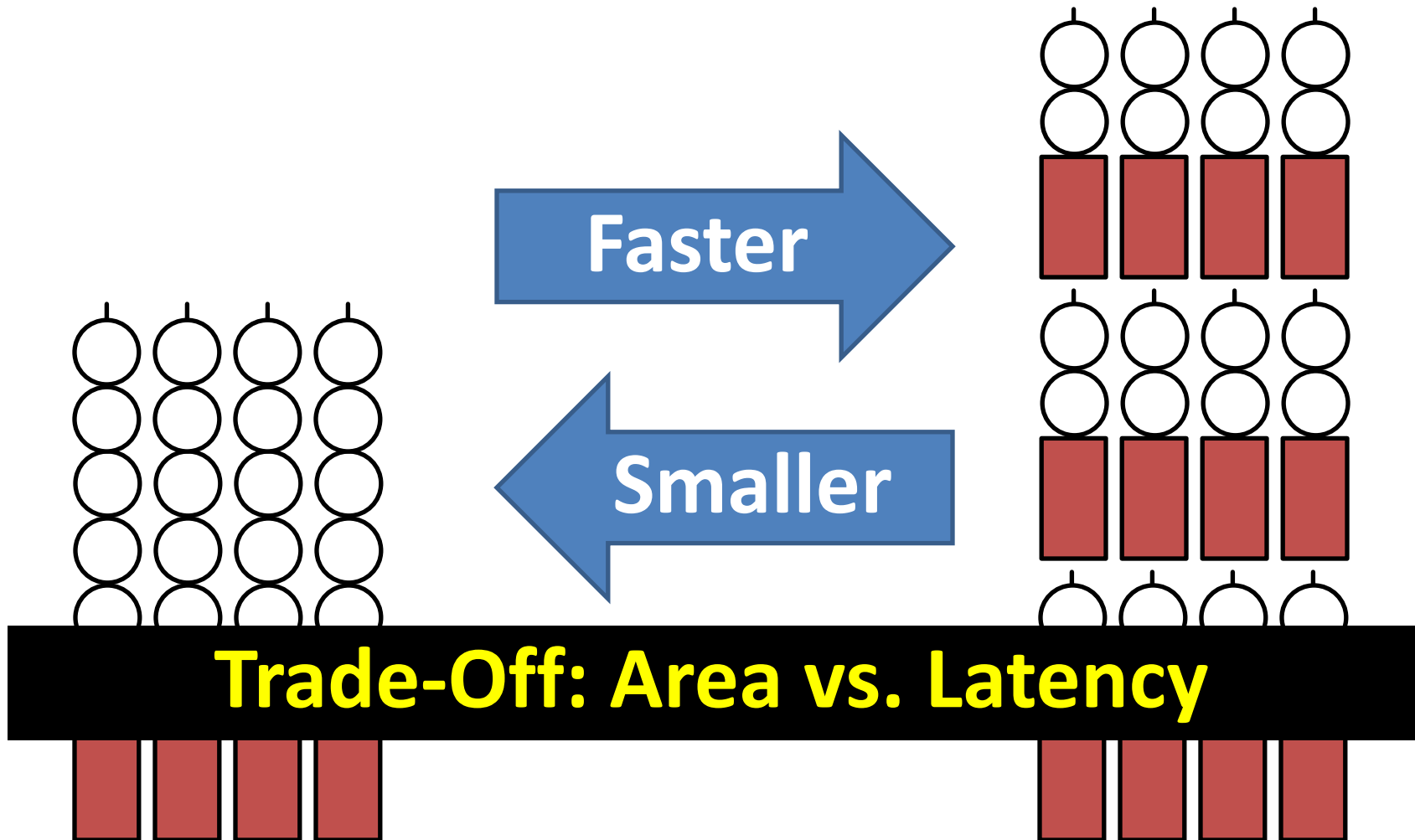
---



# Trade-Off: Area (Die Size) vs. Latency

Long Bitline

Short Bitline

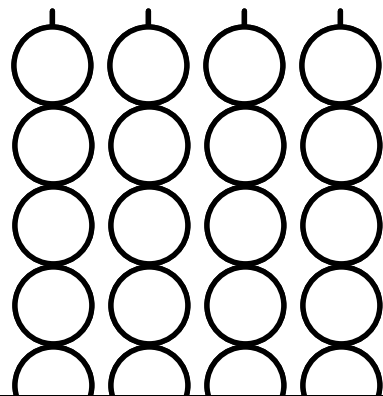


# Approximating the Best of Both Worlds

**Long Bitline**

*Small Area*

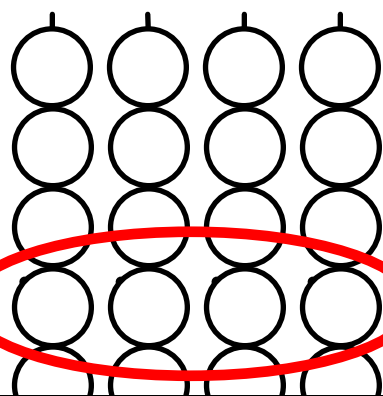
~~High Latency~~



*Need Isolation*

**Our Proposal**

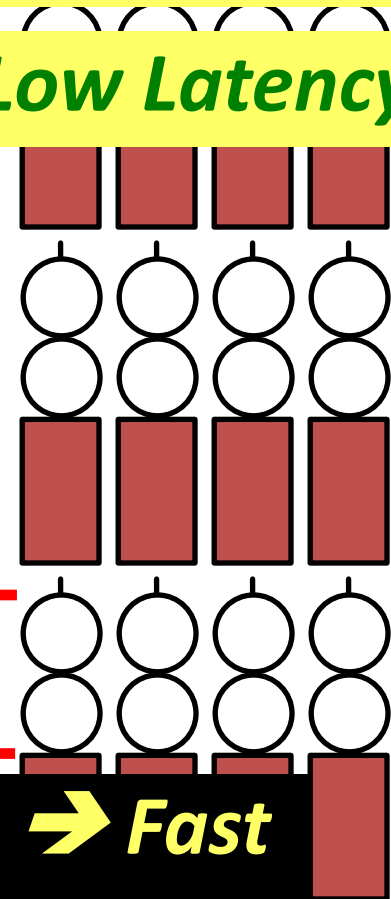
*Add Isolation Transistors*



**Short Bitline**

~~Large Area~~

*Low Latency*



*Fast*

# Approximating the Best of Both Worlds

**Long Bitline Tiered-Latency DRAM**   **Short Bitline**

*Small Area*

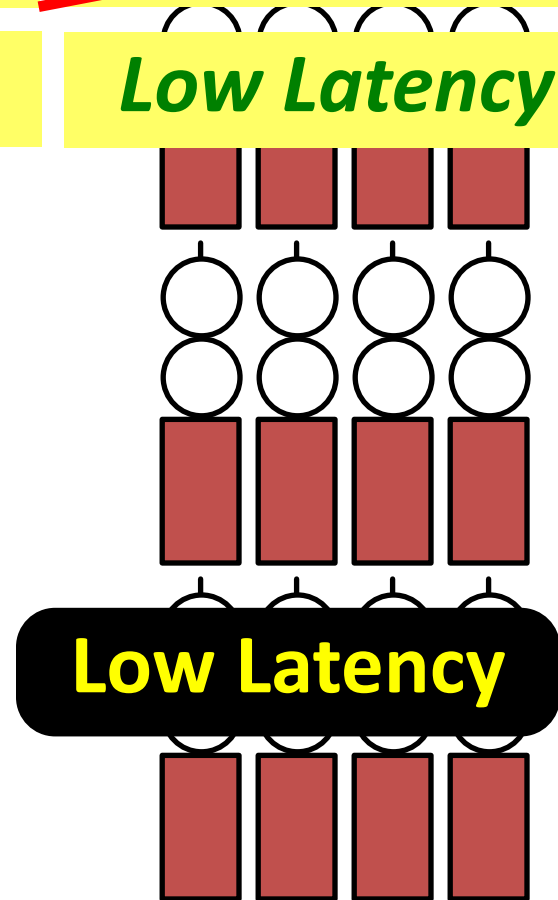
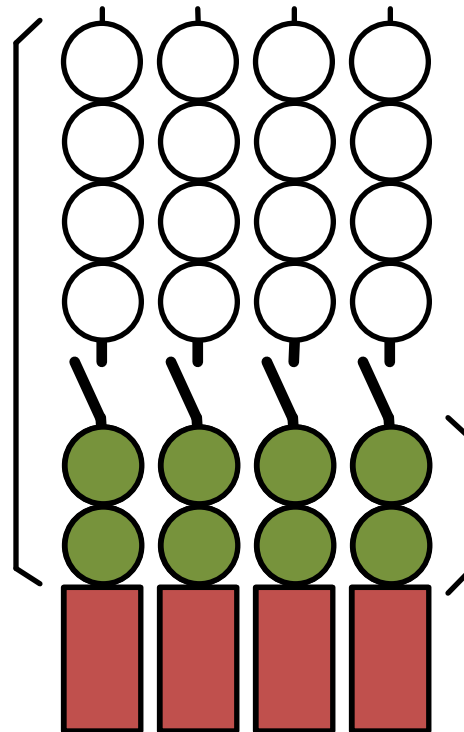
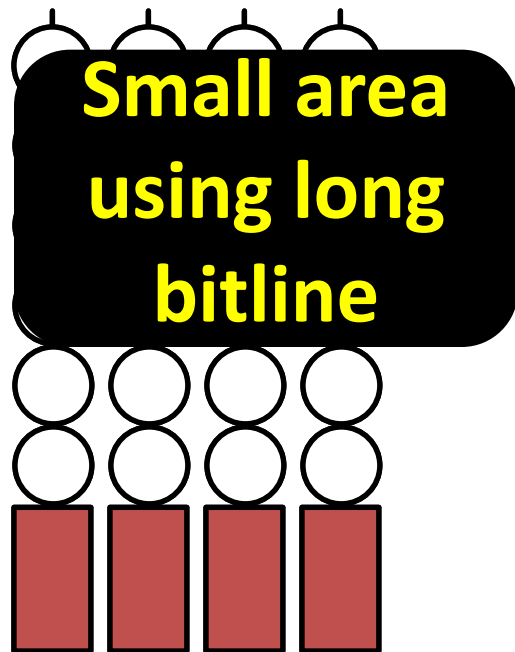
*Small Area*

~~*Large Area*~~

~~*High Latency*~~

*Low Latency*

*Low Latency*



# Heterogeneous Interconnects (in Tiler)

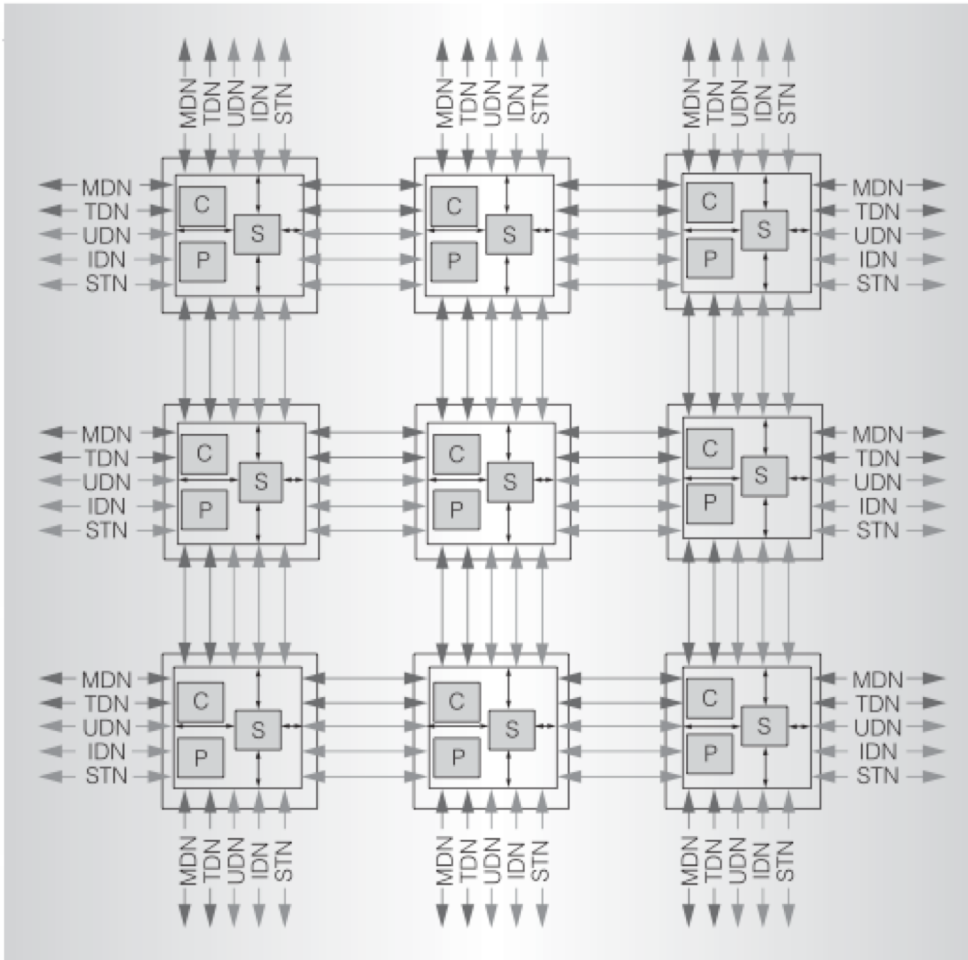


Figure 3. A  $3 \times 3$  array of tiles connected by networks. (MDN: memory dynamic network; TDN: tile dynamic network; UDN: user dynamic network; IDN: I/O dynamic network; STN: static network.)

- 2D Mesh
- Five networks
- Four packet switched
  - Dimension order routing, wormhole flow control
  - TDN: Cache request packets
  - MDN: Response packets
  - IDN: I/O packets
  - UDN: Core to core messaging
- One circuit switched
  - STN: Low-latency, high-bandwidth static network
  - Streaming data



# Aside: Examples from Life

---

- Heterogeneity is abundant in life
  - both in nature and human-made components
- Humans are heterogeneous
- Cells are heterogeneous → specialized for different tasks
- Organs are heterogeneous
- Cars are heterogeneous
- Buildings are heterogeneous
- Rooms are heterogeneous
- ...

# General-Purpose vs. Special-Purpose

---

- Asymmetry is a way of enabling specialization
- It bridges the gap between purely general purpose and purely special purpose
  - Purely general purpose: Single design for every workload or metric
  - Purely special purpose: Single design per workload or metric
  - Asymmetric: Multiple sub-designs optimized for sets of workloads/metrics and glued together
- The goal of a good asymmetric design is to get the best of both general purpose and special purpose

# Asymmetry Advantages and Disadvantages

---

- Advantages over Symmetric Design
  - + Can enable optimization of multiple metrics
  - + Can enable better adaptation to workload behavior
  - + Can provide special-purpose benefits with general-purpose usability/flexibility
  
- Disadvantages over Symmetric Design
  - Higher overhead and more complexity in design, verification
  - Higher overhead in management: scheduling onto asymmetric components
  - Overhead in switching between multiple components can lead to degradation

# Yet Another Example

---

- Modern processors integrate general purpose cores and GPUs
  - CPU-GPU systems
  - Heterogeneity in execution models

# Three Key Problems in Future Systems

---

## ■ Memory system

- ❑ Applications are increasingly data intensive
- ❑ 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

**Asymmetric Designs  
Can Help Solve These Problems**

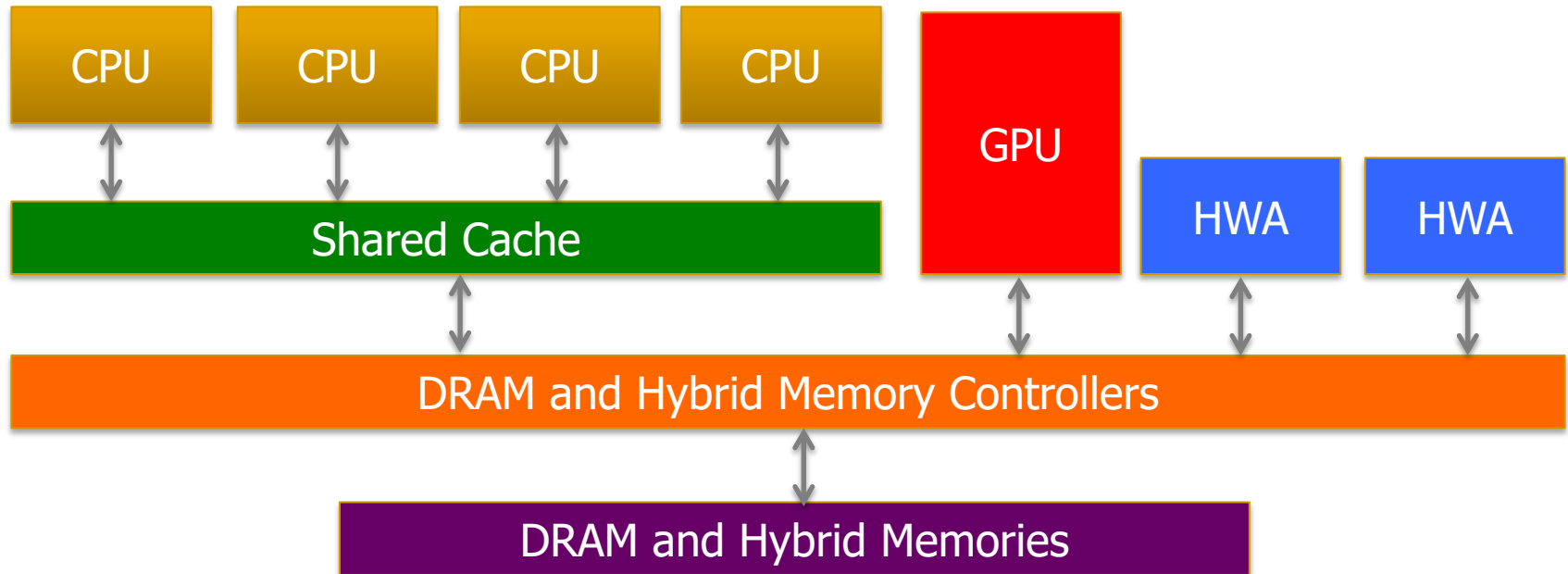
# Commercial Asymmetric Design Examples

---

- Integrated CPU-GPU systems (e.g., Intel SandyBridge)
- CPU + Hardware Accelerators (e.g., your cell phone)
- ARM big.LITTLE processor
- IBM Cell processor

# Increasing Asymmetry in Modern Systems

---



- Heterogeneous agents: CPUs, GPUs, and HWAs
- Heterogeneous memories: Fast vs. Slow DRAM
- Heterogeneous interconnects: Control, Data, Synchronization

# Computer Architecture

## Lecture 19b:

# Heterogeneous Multi-Core Systems

Prof. Onur Mutlu

ETH Zürich

Fall 2018

28 November 2018

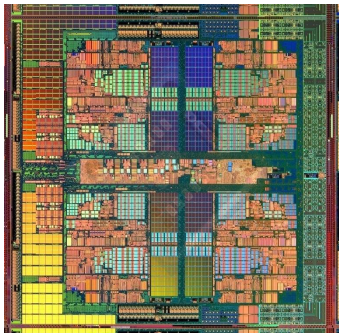


We did not cover the following slides in lecture.  
These are for your preparation for the next lecture.

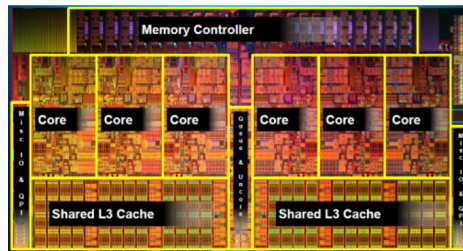
# Multi-Core Design: An Asymmetric Perspective

# Many Cores on Chip

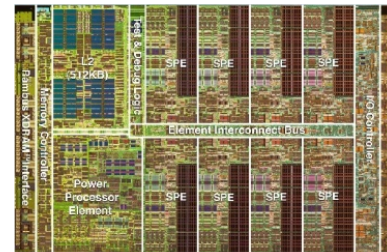
- Simpler and lower power than a single large core
- Large scale parallelism on chip



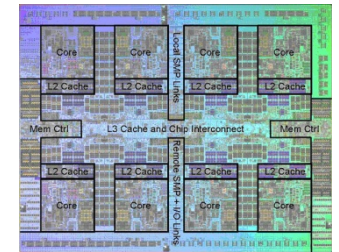
AMD Barcelona  
4 cores



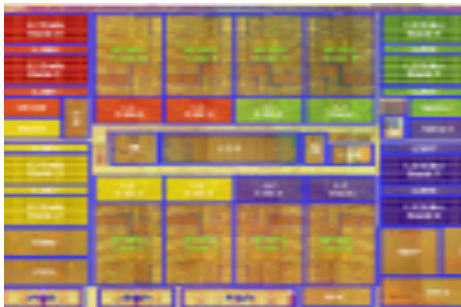
Intel Core i7  
8 cores



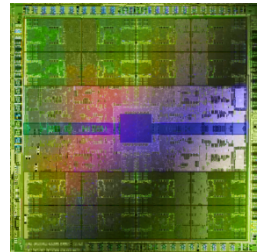
IBM Cell BE  
8+1 cores



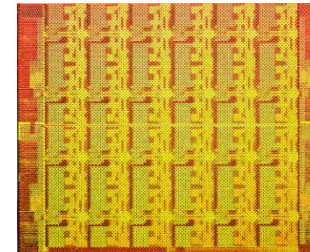
IBM POWER7  
8 cores



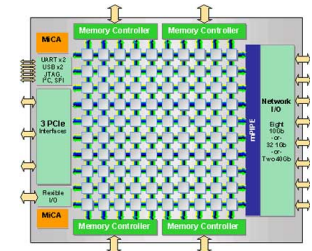
Sun Niagara II  
8 cores



Nvidia Fermi  
448 "cores"



Intel SCC  
48 cores, networked



Tiler TILE Gx  
100 cores, networked

# With Many Cores on Chip

---

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

# Caveats of Parallelism

---

## ■ Amdahl's Law

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

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

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

# The Problem: Serialized Code Sections

---

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

# Example from MySQL

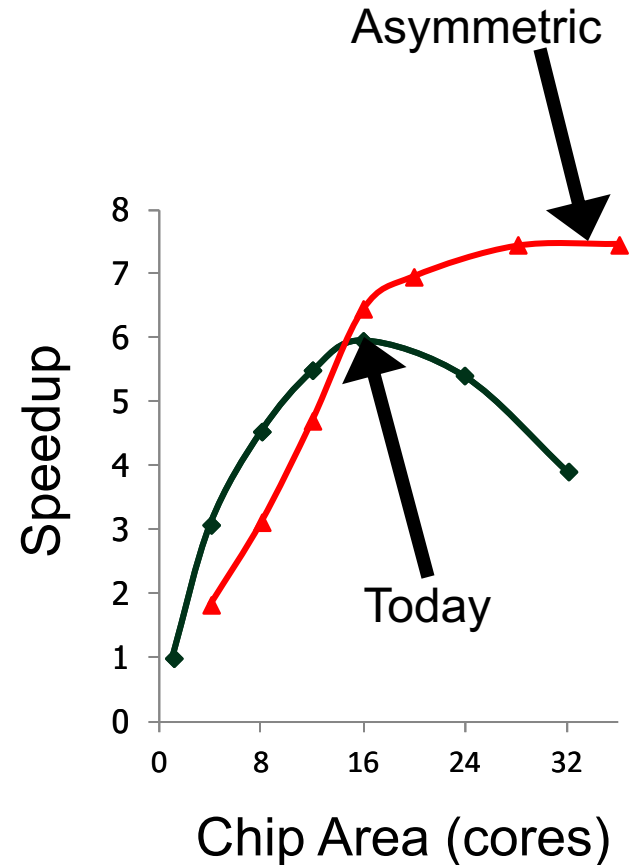
**Critical  
Section**

Access Open Tables Cache

Open database tables

Perform the operations  
....

Parallel



# Demands in Different Code Sections

---

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



# “Large” vs. “Small” Cores

---

## Large Core

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

## Small Core

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

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

# Large vs. Small Cores

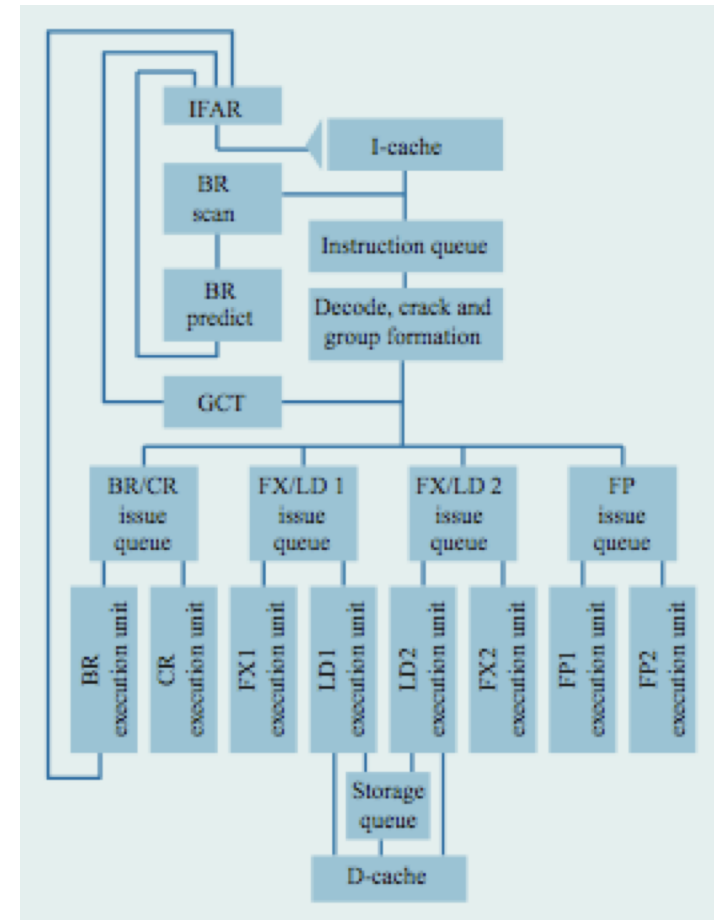
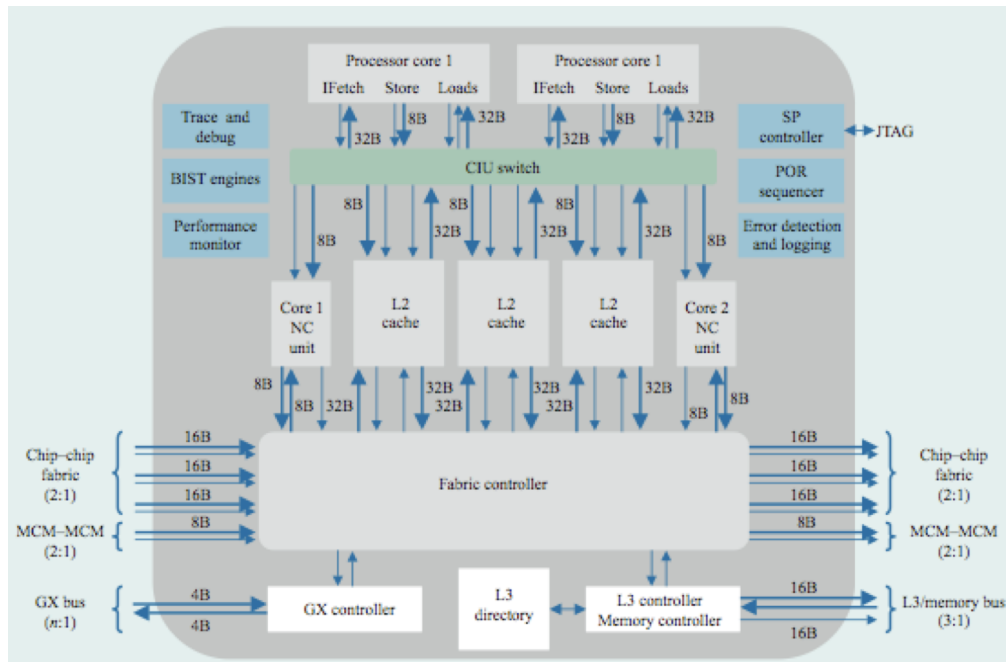
---

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

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

# Meet Large: IBM POWER4

- Tendler et al., “POWER4 system microarchitecture,” IBM J R&D, 2002.
- A symmetric multi-core chip...
- Two powerful cores



# IBM POWER4

---

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

# IBM POWER5

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

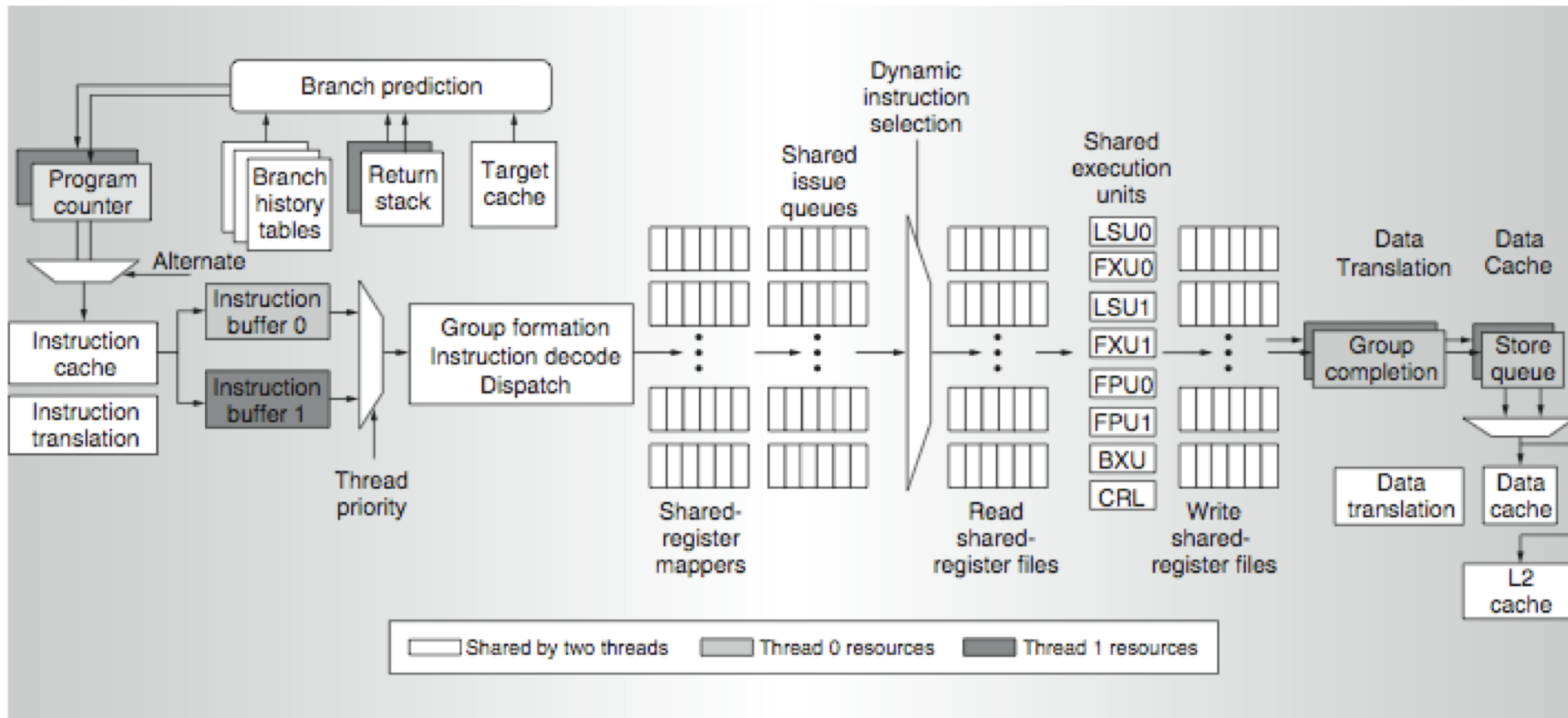
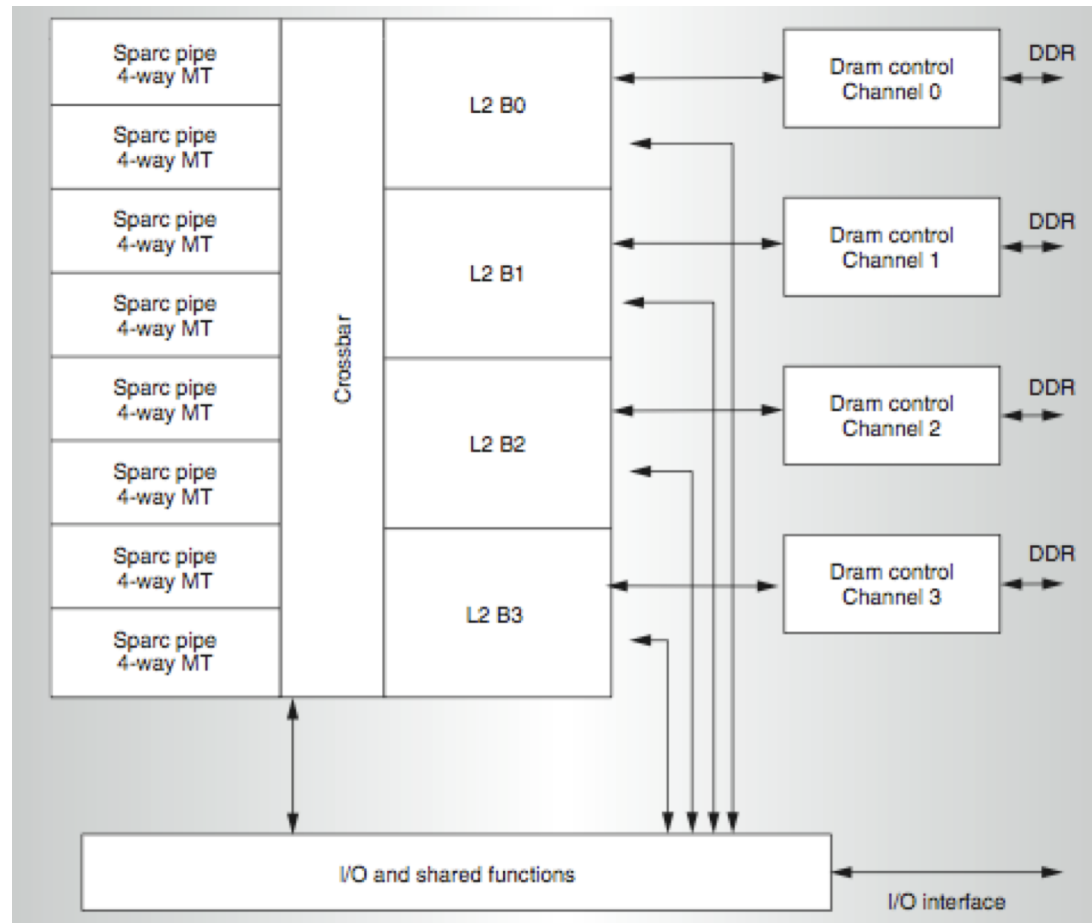


Figure 4. Power5 instruction data flow (BXU = branch execution unit and CRL = condition register logical execution unit).

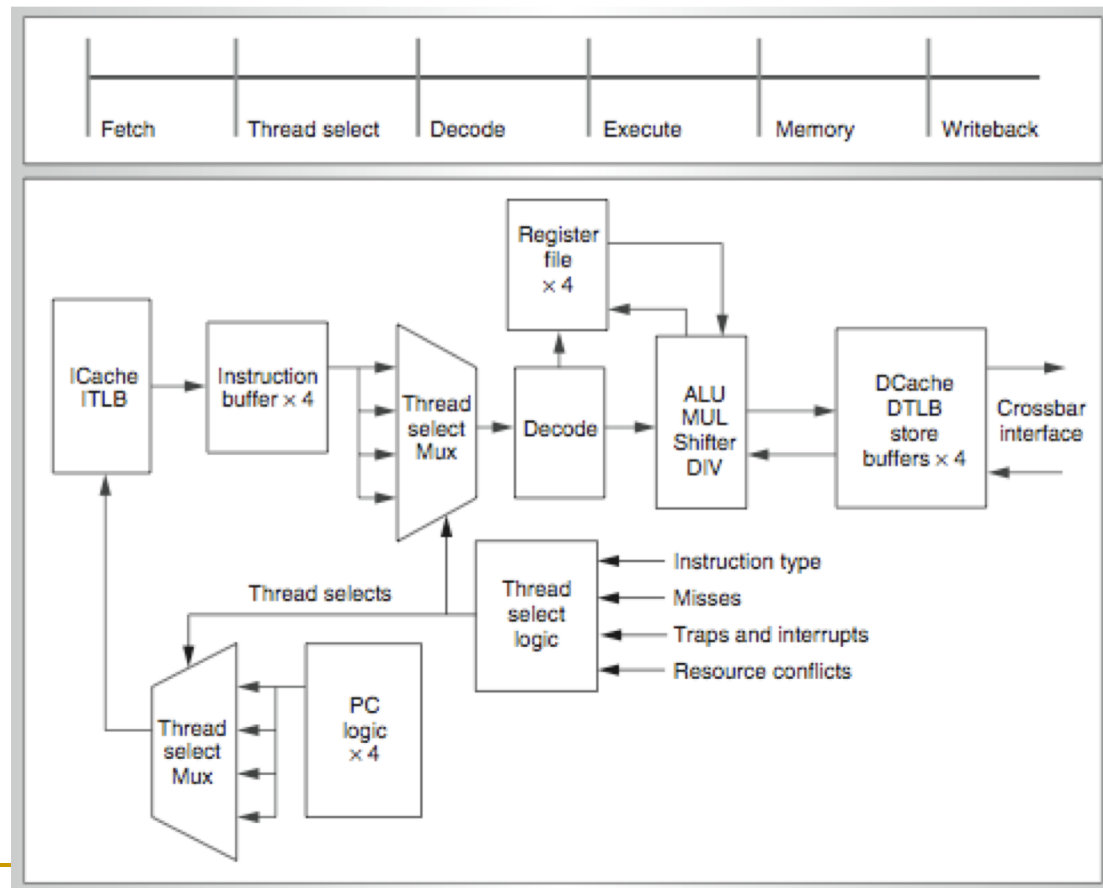
# Meet Small: Sun Niagara (UltraSPARC T1)

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



# Niagara Core

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



# Remember the Demands

---

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



# Performance vs. Parallelism

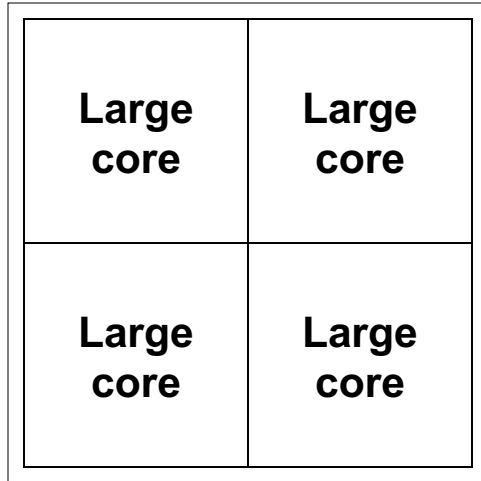
---

## *Assumptions:*

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

# Tile-Large Approach

---



“Tile-Large”

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

# Tile-Small Approach

---

Small core	<b>Small core</b>	<b>Small core</b>	<b>Small core</b>
<b>Small core</b>	<b>Small core</b>	<b>Small core</b>	<b>Small core</b>
<b>Small core</b>	<b>Small core</b>	<b>Small core</b>	<b>Small core</b>
<b>Small core</b>	<b>Small core</b>	<b>Small core</b>	<b>Small core</b>

“Tile-Small”

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

# Can we get the best of both worlds?

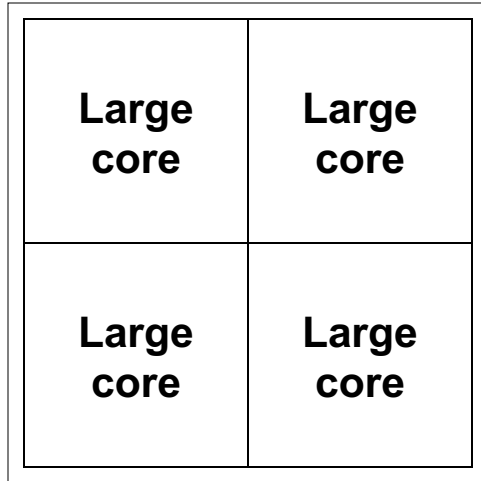
---

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

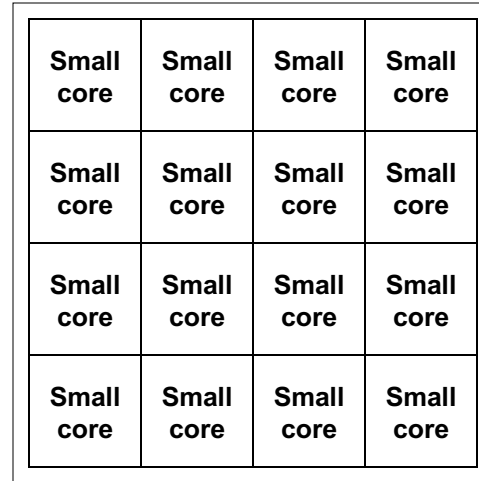
# Asymmetric Multi-Core

# Asymmetric Chip Multiprocessor (ACMP)

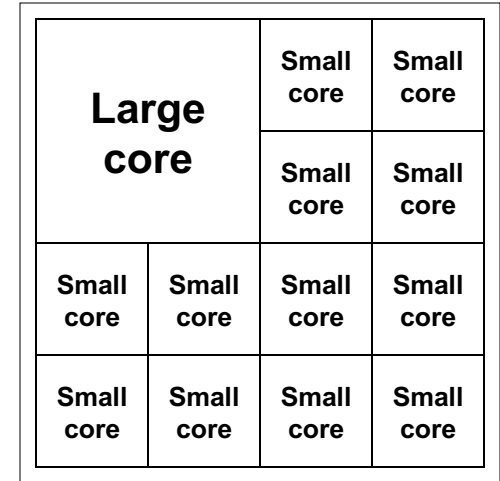
---



“Tile-Large”



“Tile-Small”



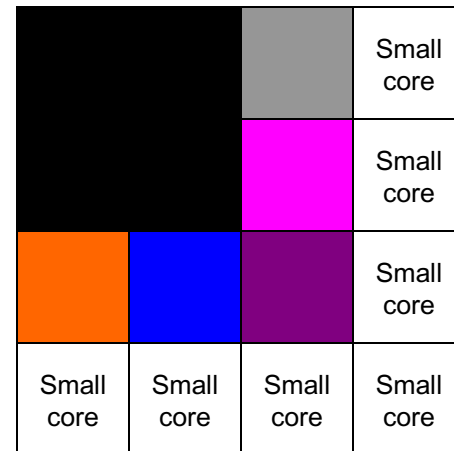
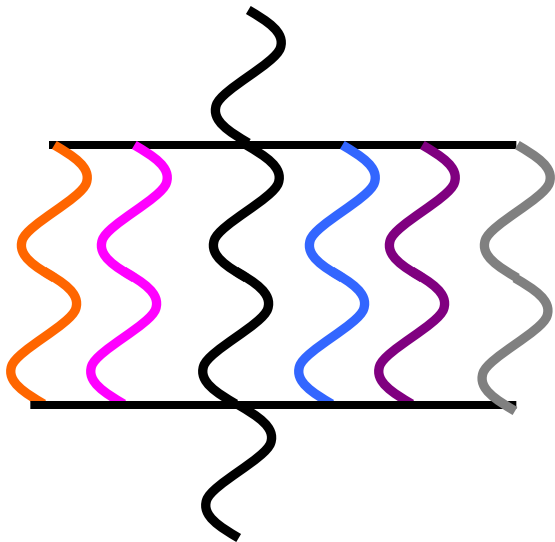
ACMP

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

# Accelerating Serial Bottlenecks

---

Single thread  $\rightarrow$  Large core



ACMP Approach

# Performance vs. Parallelism

---

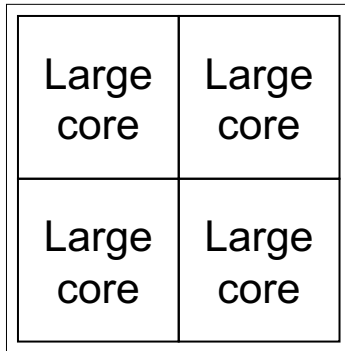
## *Assumptions:*

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

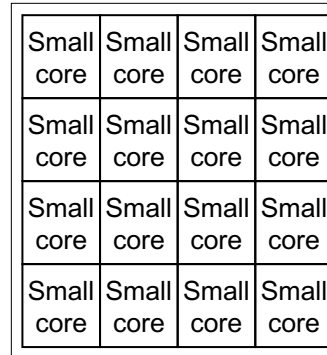


# ACMP Performance vs. Parallelism

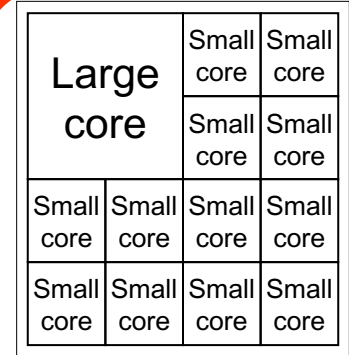
Area-budget = 16 small cores



“Tile-Large”



“Tile-Small”



ACMP

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

# Amdahl's Law Modified

---

- Simplified Amdahl's Law for an Asymmetric Multiprocessor
- Assumptions:
  - Serial portion executed on the large core
  - Parallel portion executed on both small cores and large cores
  - $f$ : Parallelizable fraction of a program
  - $L$ : Number of large processors
  - $S$ : Number of small processors
  - $X$ : Speedup of a large processor over a small one

$$\text{Speedup} = \frac{1}{\frac{1-f}{X} + \frac{f}{S + X*L}}$$

# Caveats of Parallelism, Revisited

---

## ■ Amdahl's Law

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

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

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

# Accelerating Parallel Bottlenecks

---

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

# Accelerated Critical Sections

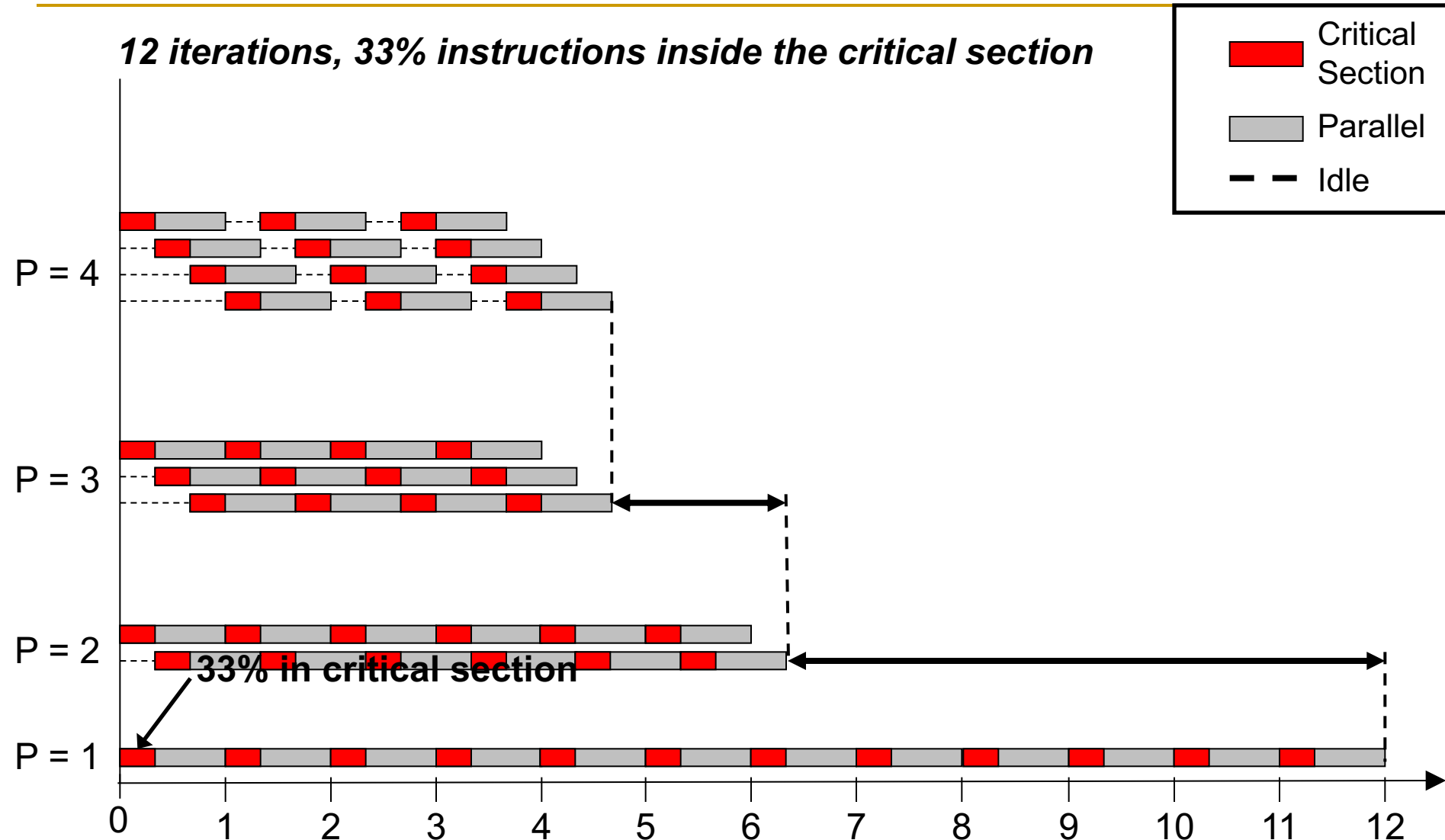
M. Aater Suleman, Onur Mutlu, Moinuddin K. Qureshi, and Yale N. Patt,

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

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

# Contention for Critical Sections

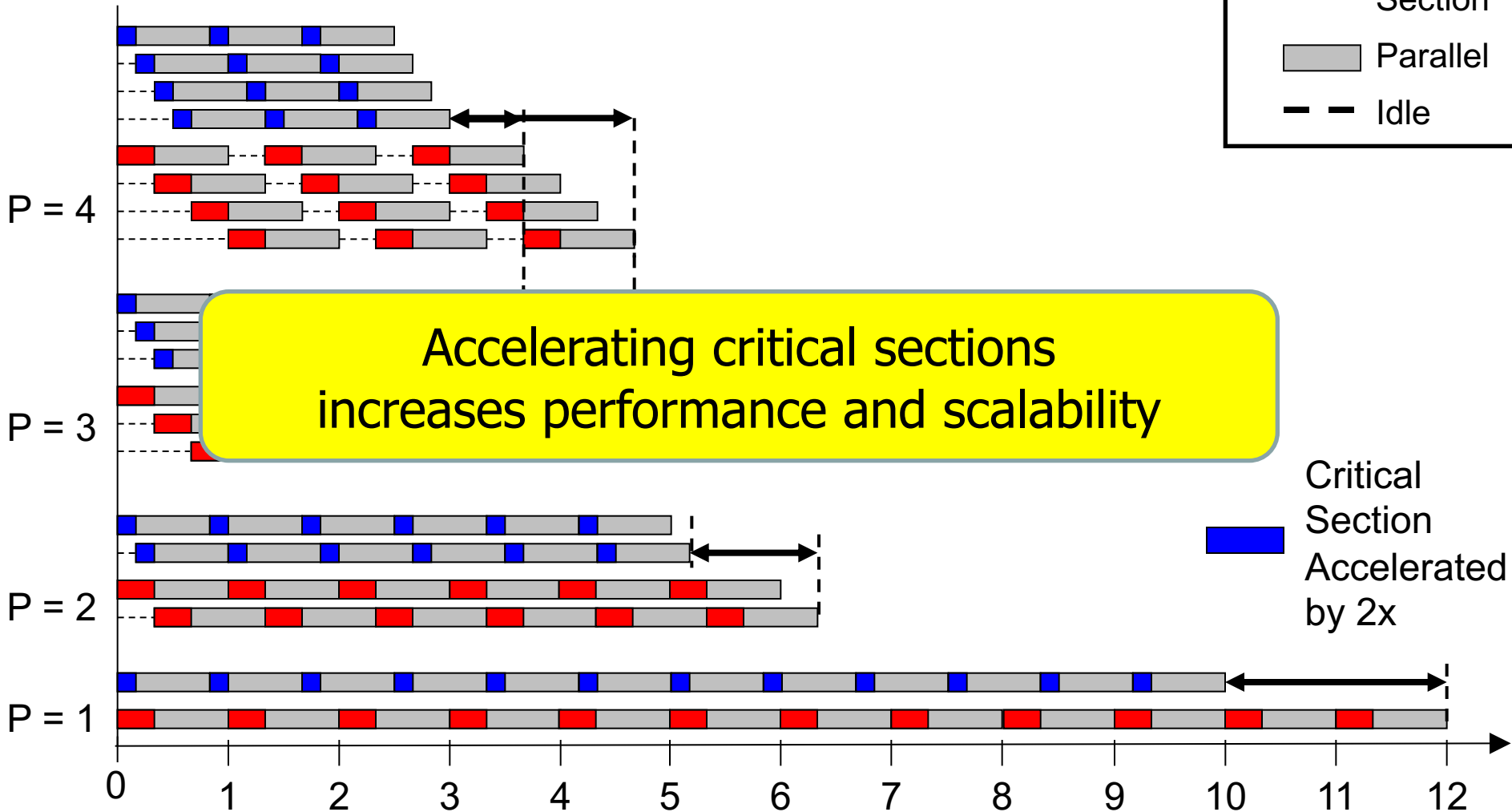
*12 iterations, 33% instructions inside the critical section*



# Contention for Critical Sections

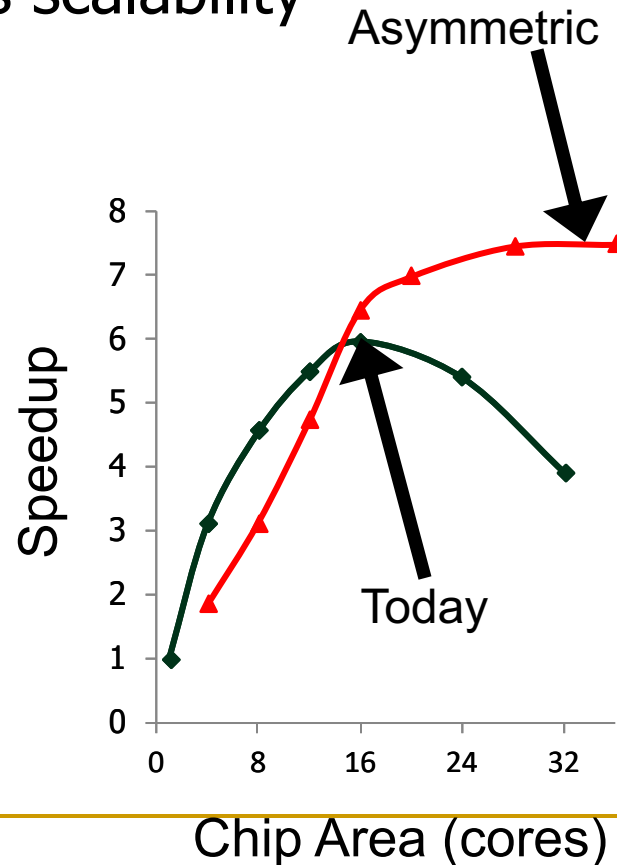
*12 iterations, 33% instructions inside the critical section*

Critical Section  
Parallel  
Idle



# 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

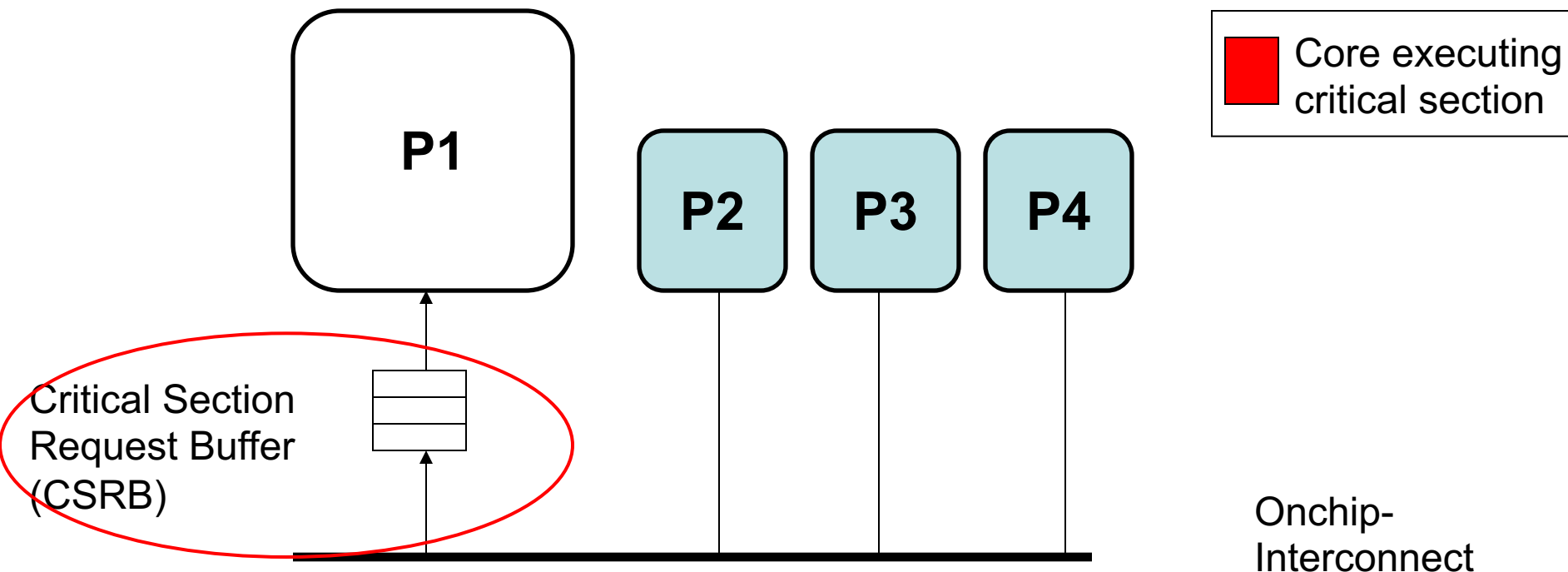
---

- 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

...

...

...

...

...

...

...

...

POP result

print result

## Large Core

CSCALL Request

Send X, TPC,  
STACK\_PTR, CORE\_ID

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

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

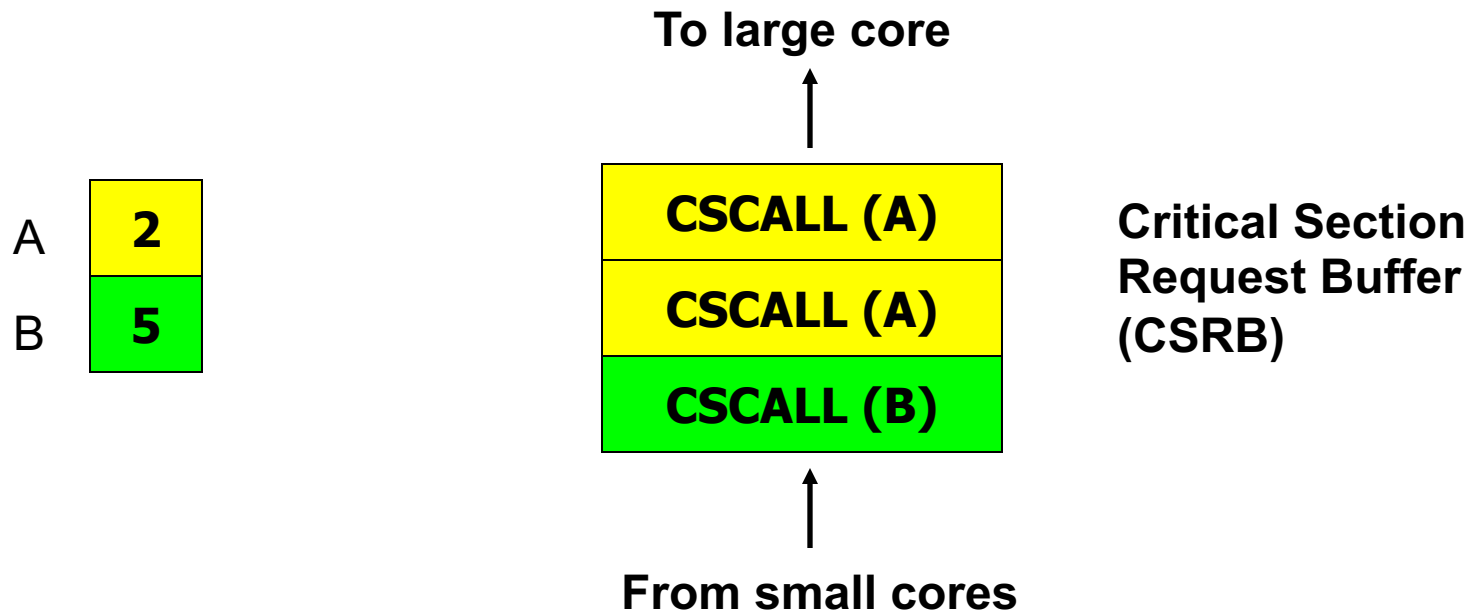
CSDONE Response

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

# False Serialization

---

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



# ACS Performance Tradeoffs

---

## ■ Pluses

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

## ■ Minuses

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

# ACS Performance Tradeoffs

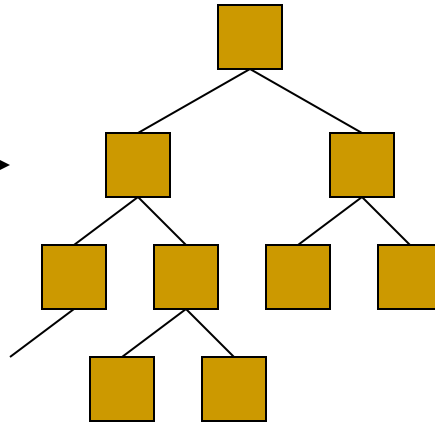
---

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

# Cache Misses for Private Data

**PriorityHeap.insert(NewSubProblems)**

**Private Data:**  
**NewSubProblems**



**Shared Data:**  
**The priority heap**

**Puzzle Benchmark**



# ACS Performance Tradeoffs

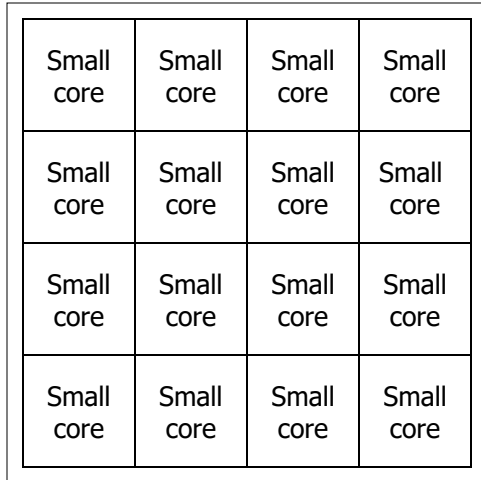
---

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

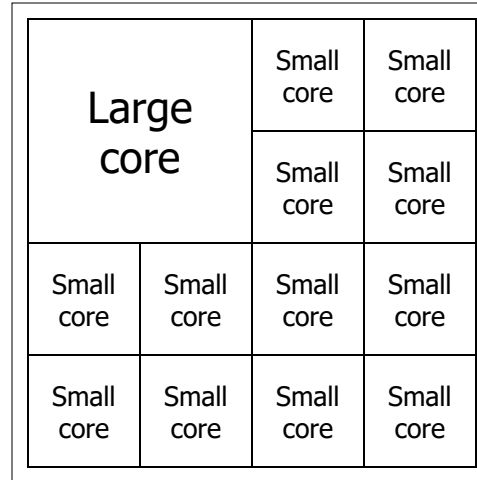
**This problem can be solved**

# ACS Comparison Points

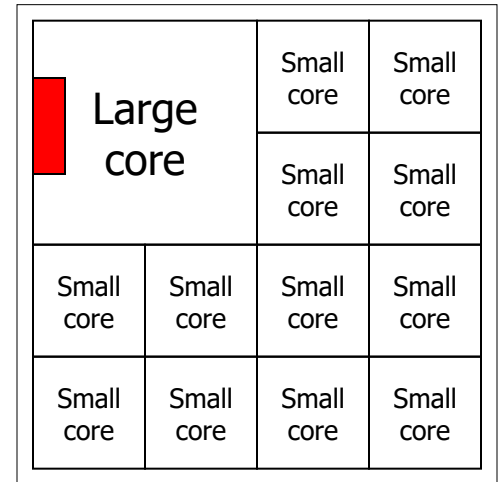
---



SCMP



ACMP



ACS

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

# Accelerated Critical Sections: Methodology

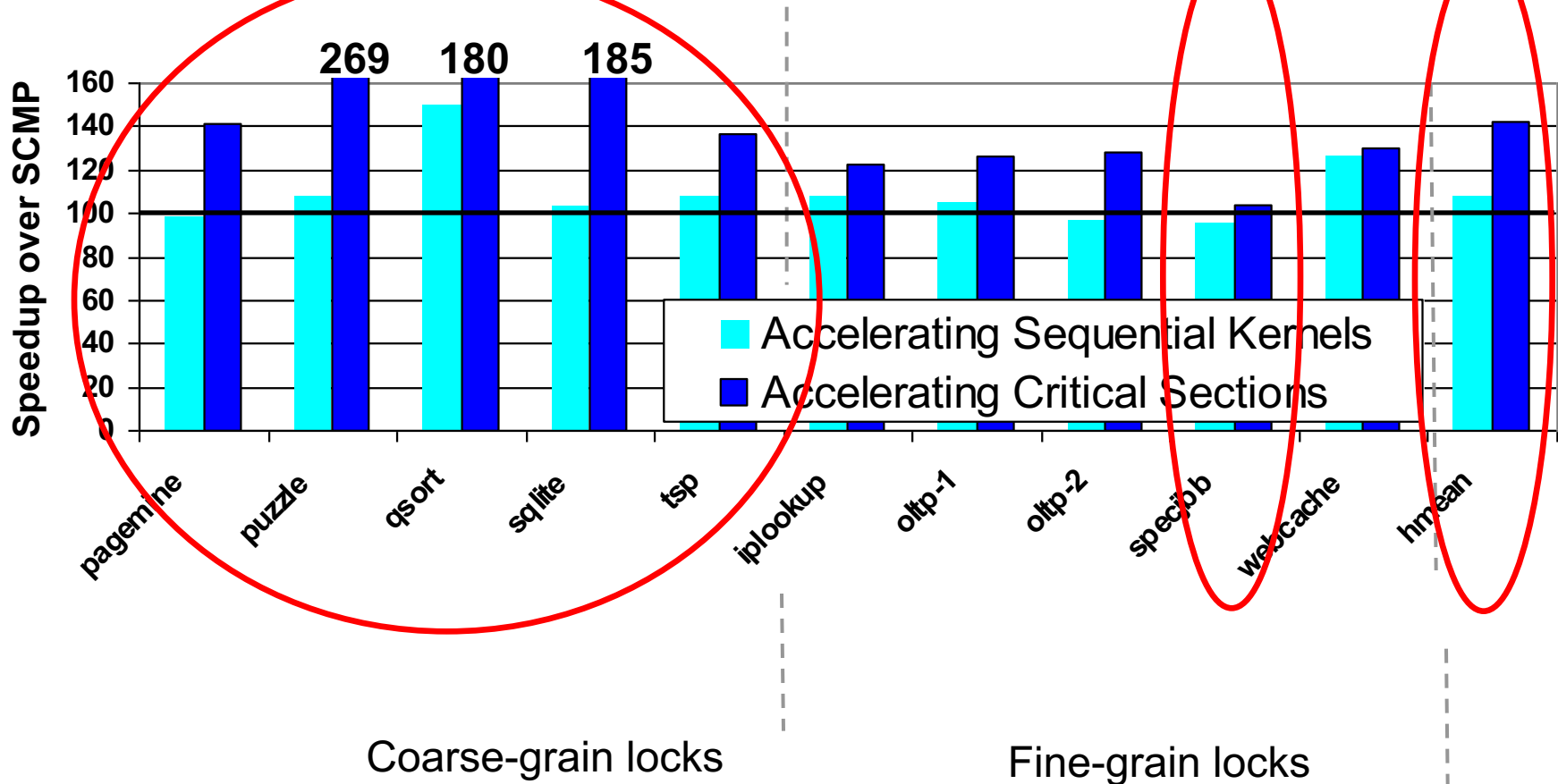
---

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

# ACS Performance

**Chip Area = 32 small cores**  
SCMP = 32 small cores  
ACMP = 1 large and 28 small cores

Equal-area comparison  
Number of threads = *Best threads*

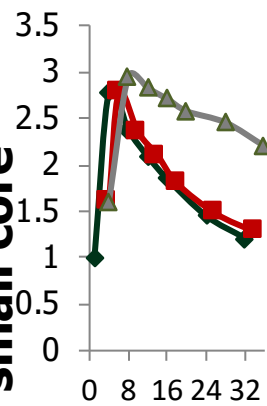


# Equal-Area Comparisons

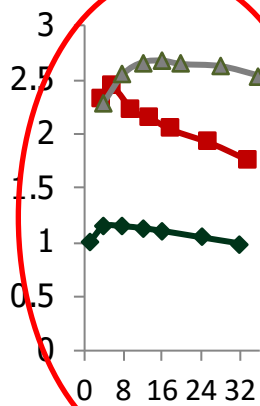
----- **SCMP**  
----- **ACMP**  
----- **ACS**

Number of threads = No. of cores

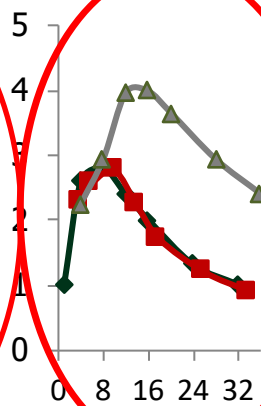
Speedup over a small core



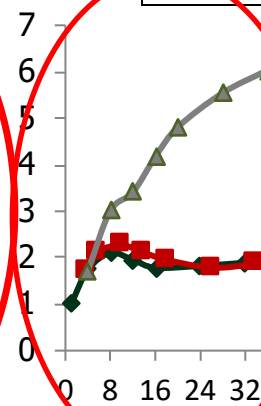
(a) ep



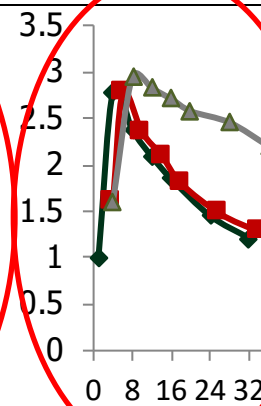
(b) is



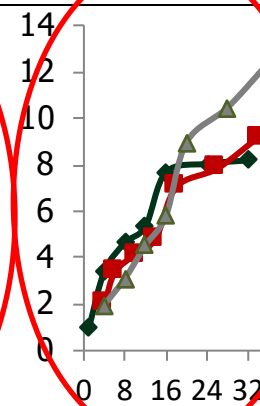
(c) pagemine



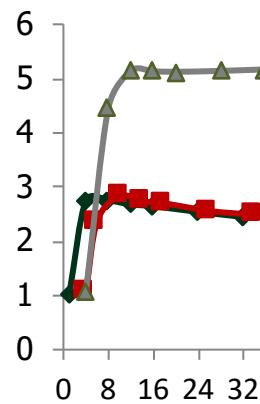
(d) puzzle



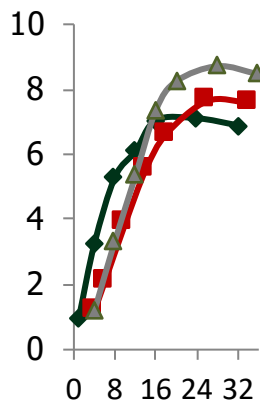
(e) qsort



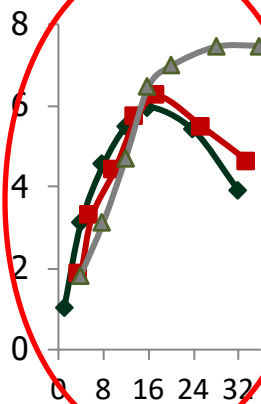
(f) tsp



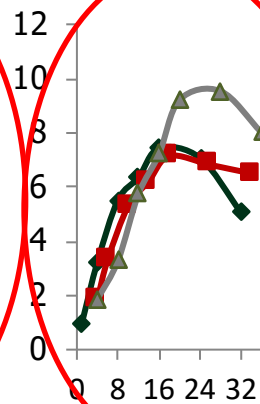
(g) sqlite



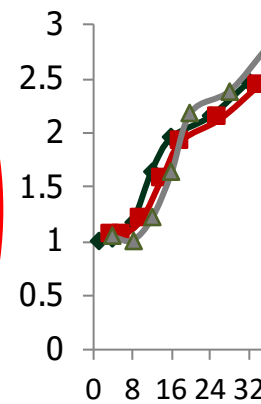
(h) iplookup



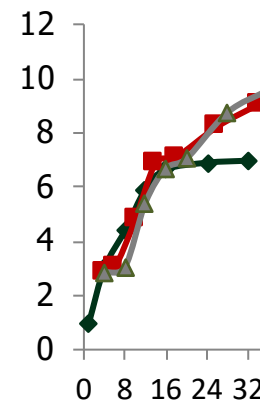
(i) oltp-1



(j) oltp-2



(k) specjbb



(l) webcache

Chip Area (small cores)

# ACS Summary

---

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

# 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

# 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

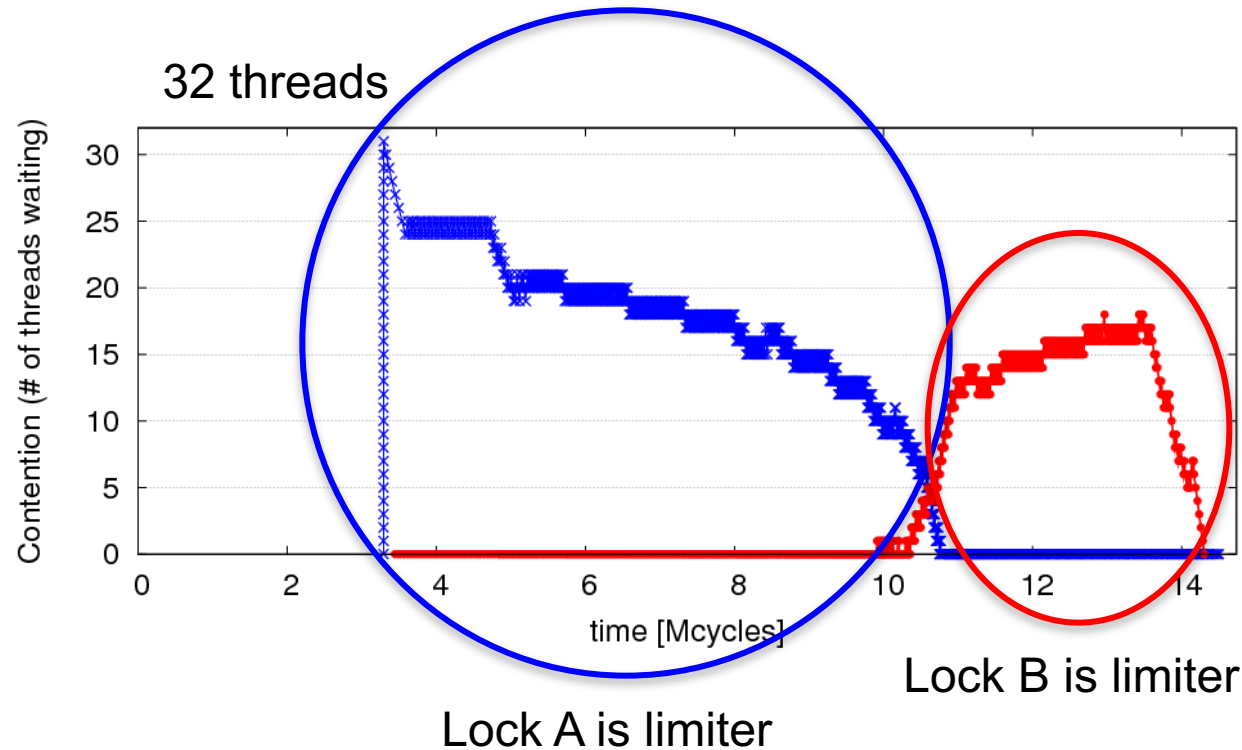
Lock B

Traverse list B

Insert X into B

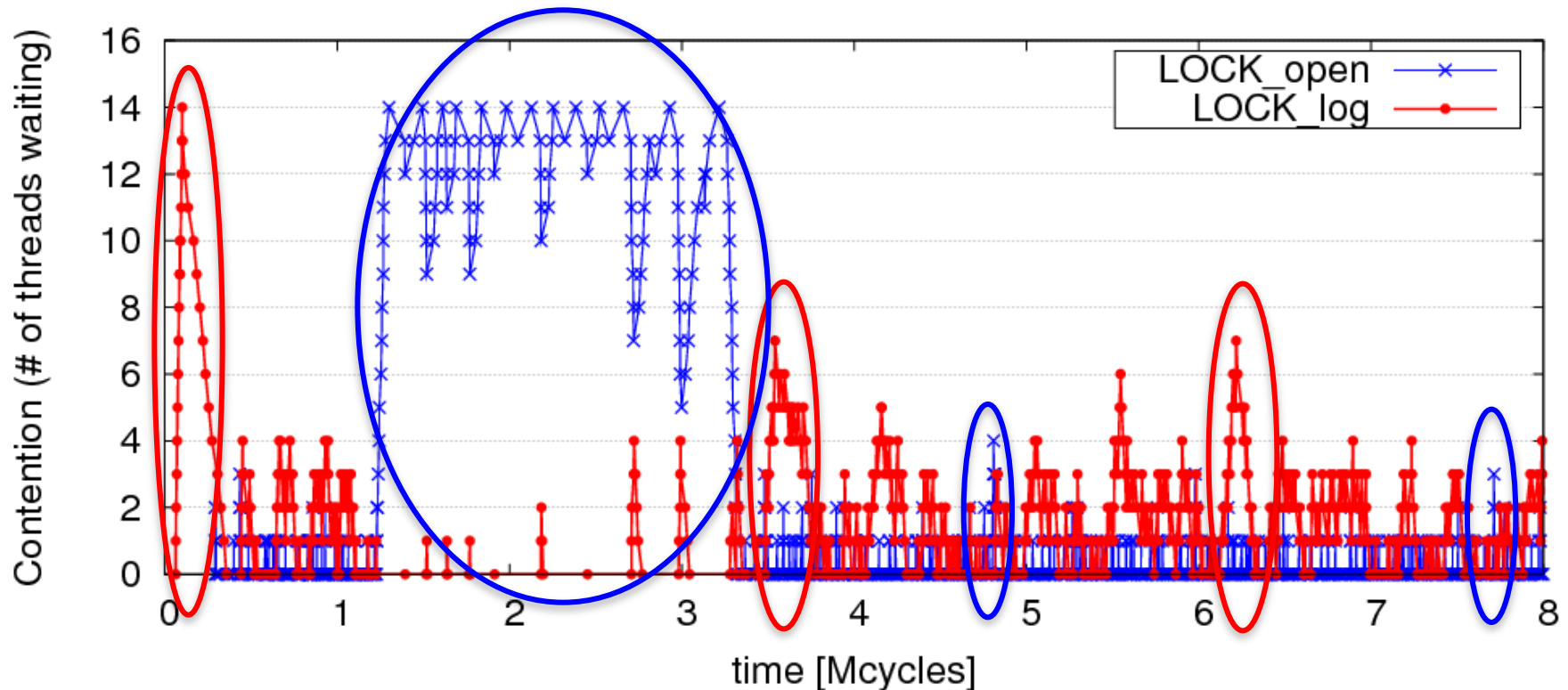
Unlock B

until A is empty



# 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)

---

## Compiler/Library/Programmer

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

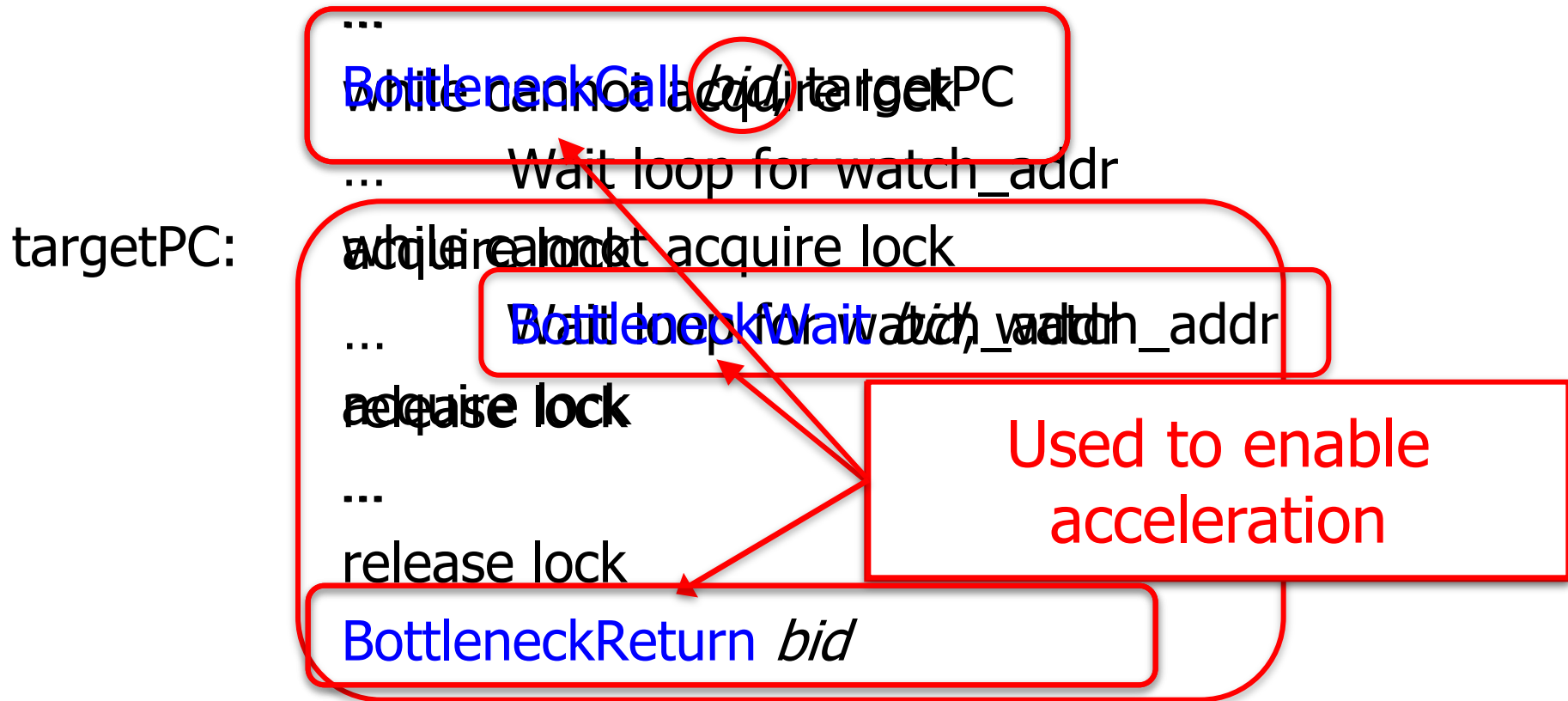
Binary containing  
**BIS instructions**

## Hardware

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

# Critical Sections: Code Modifications

---



# Barriers: Code Modifications

---

...

**BottleneckCall** *bid*, targetPC

enter barrier

while not all threads in barrier

**BottleneckWait** *bid*, watch\_addr

exit barrier

...

targetPC:

code running for the barrier

...

**BottleneckReturn** *bid*

# Pipeline Stages: Code Modifications

---

**BottleneckCall** *bid*, targetPC

...

targetPC:

while not done

while empty queue

**BottleneckWait** prev\_bid

dequeue work

do the work ...

while full queue

**BottleneckWait** next\_bid

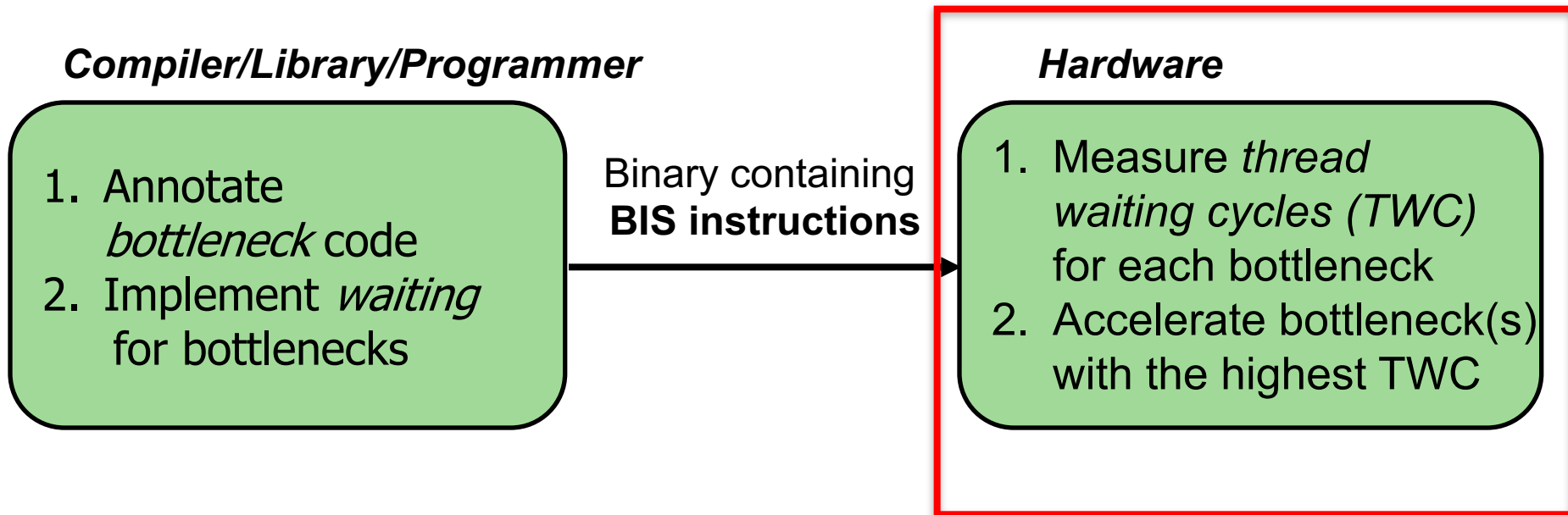
enqueue next work

**BottleneckReturn** *bid*



# Bottleneck Identification and Scheduling (BIS)

---



# BIS: Hardware Overview

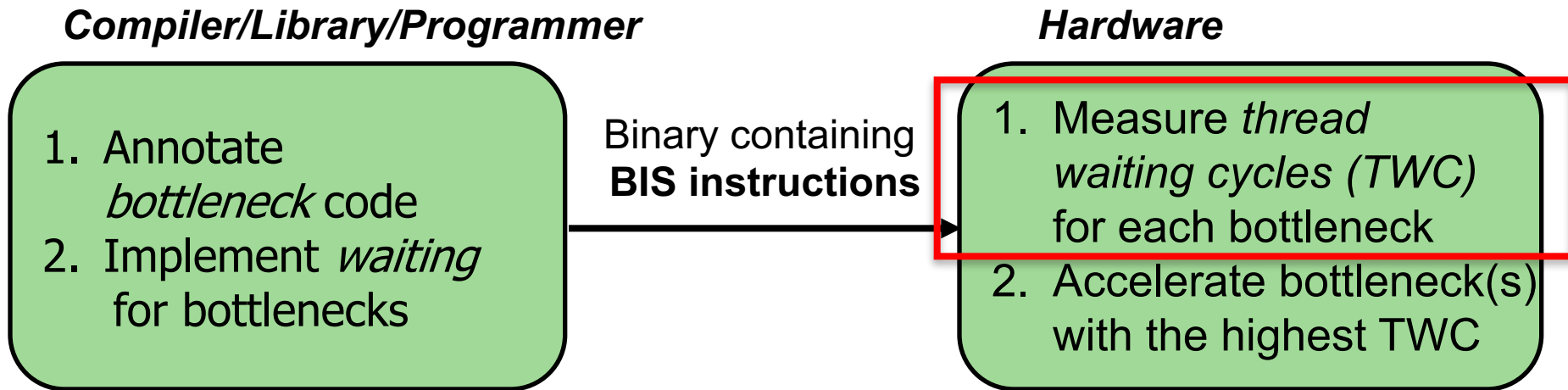
---

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

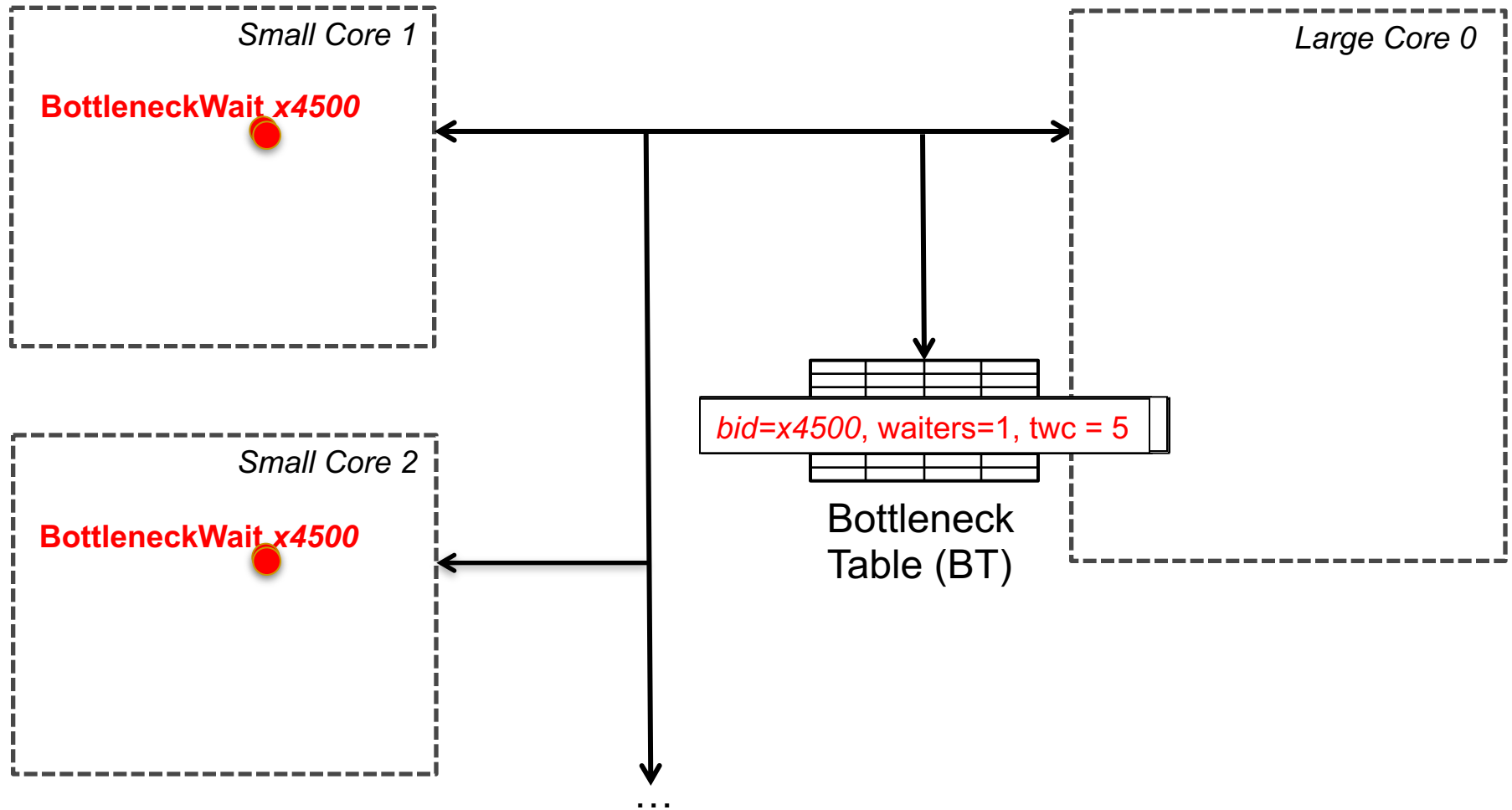
Small core	Small core	Large core	
Small core	Small core		
Small core	Small core	Small core	Small core
Small core	Small core	Small core	Small core

# Bottleneck Identification and Scheduling (BIS)

---

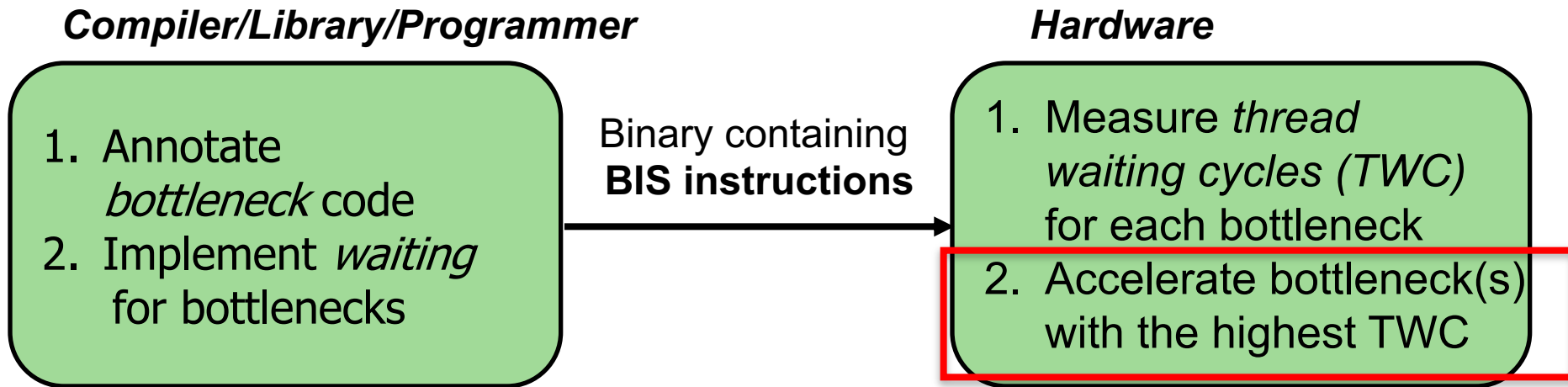


# Determining Thread Waiting Cycles for Each Bottleneck

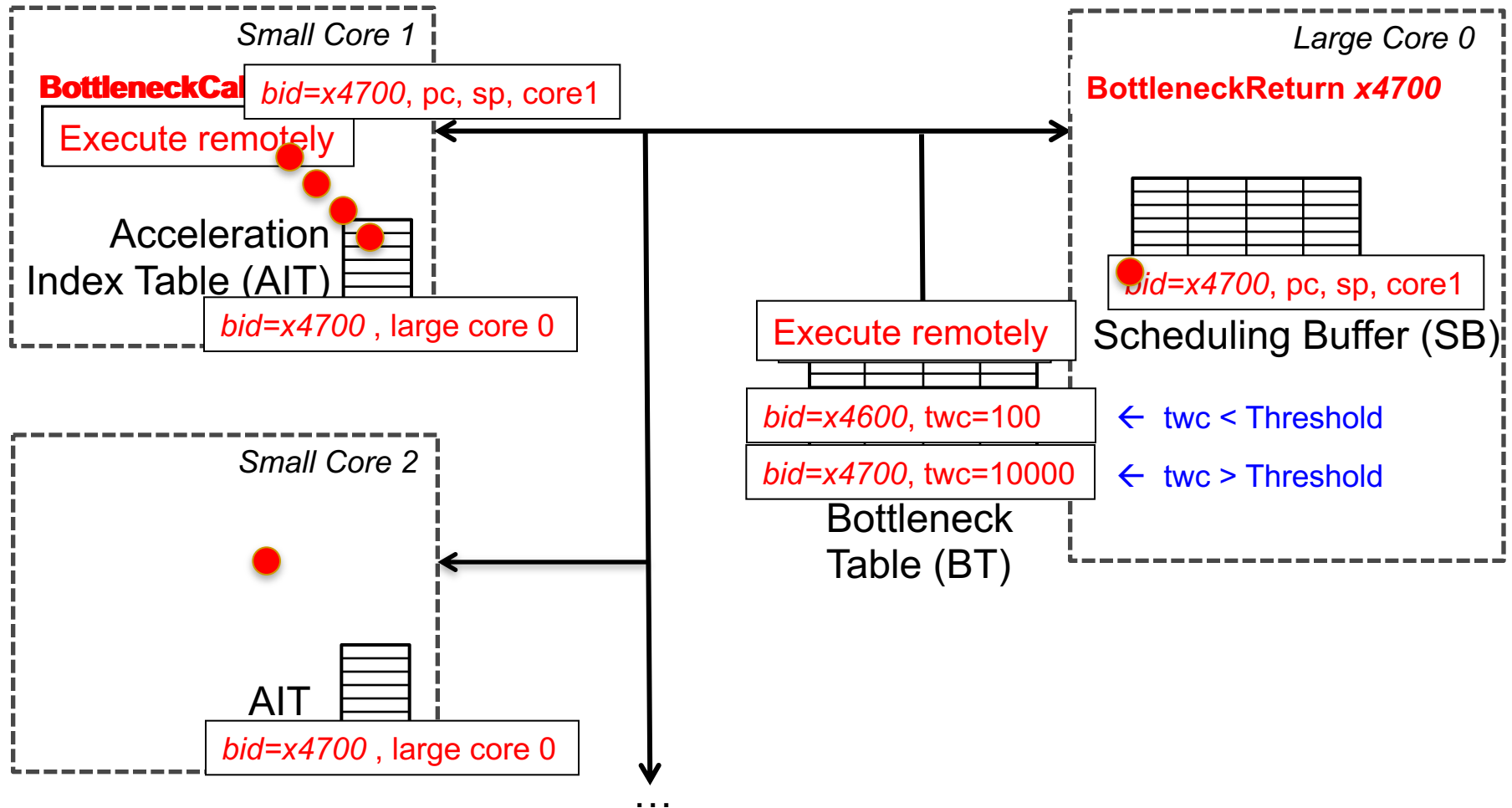


# Bottleneck Identification and Scheduling (BIS)

---



# Bottleneck Acceleration



# BIS Mechanisms

---

- Basic mechanisms for BIS:
  - Determining Thread Waiting Cycles ✓
  - Accelerating Bottlenecks ✓
  
- Mechanisms to improve performance and generality of BIS:
  - Dealing with false serialization
  - Preemptive acceleration
  - Support for multiple large cores

# Hardware Cost

---

- Main structures:

- Bottleneck Table (BT): global 32-entry associative cache, minimum-Thread-Waiting-Cycle replacement
- Scheduling Buffers (SB): one table per large core, as many entries as small cores
- Acceleration Index Tables (AIT): one 32-entry table per small core

- Off the critical path

- Total storage cost for 56-small-cores, 2-large-cores < 19 KB

---



# BIS Performance Trade-offs

---

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

# Evaluation Methodology

---

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

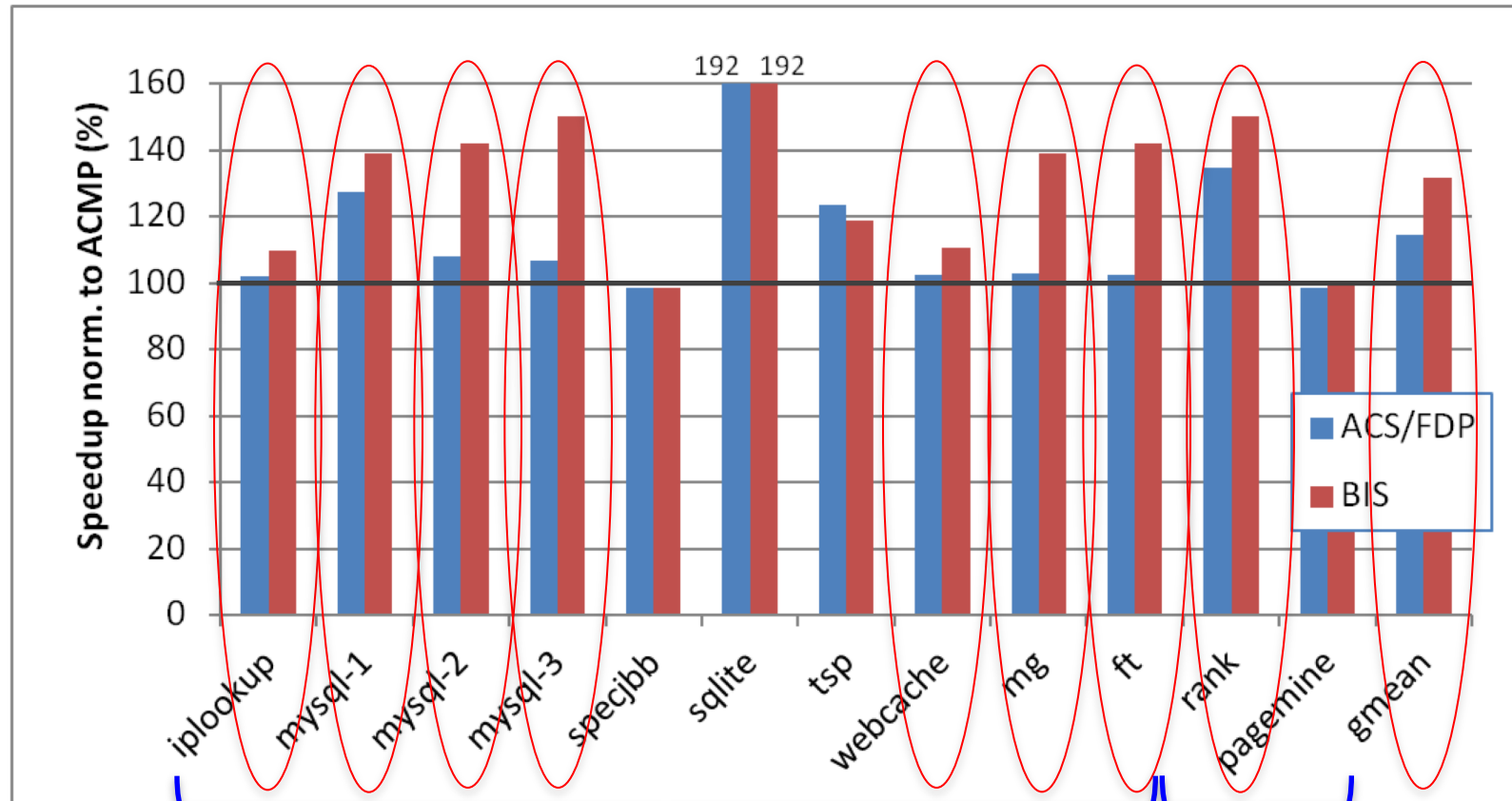
# BIS Comparison Points (Area-Equivalent)

---

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

# BIS Performance Improvement

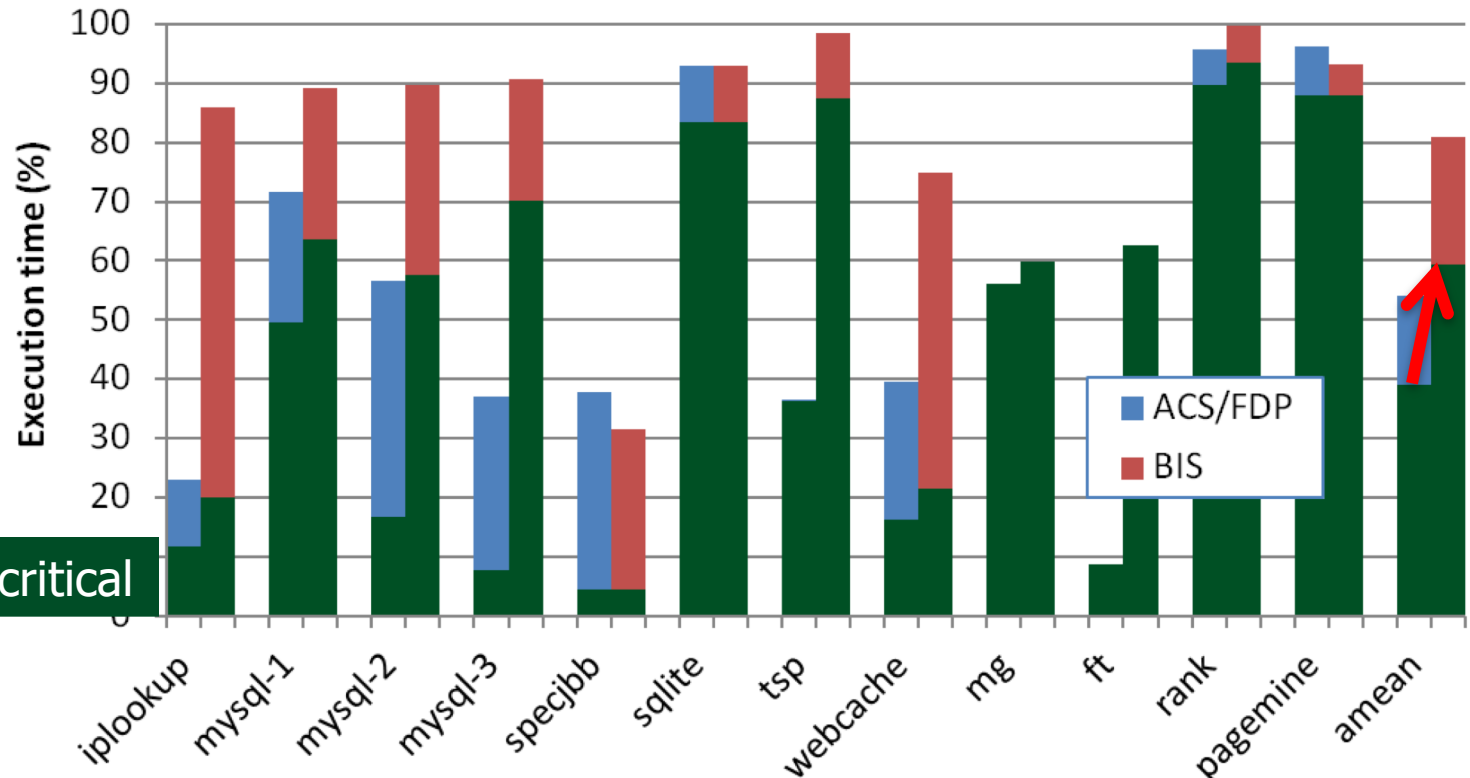
Optimal number of threads, 28 small cores, 1 large core



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

# Why Does BIS Work?

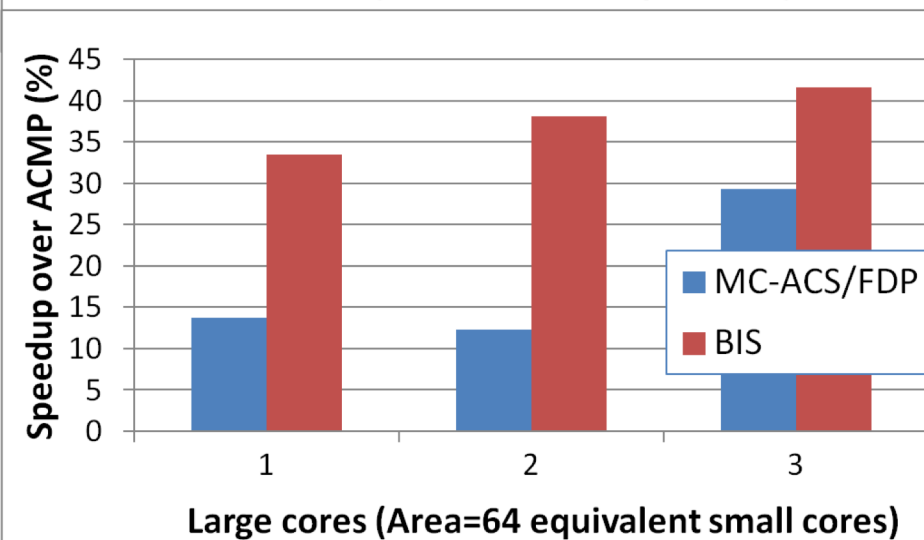
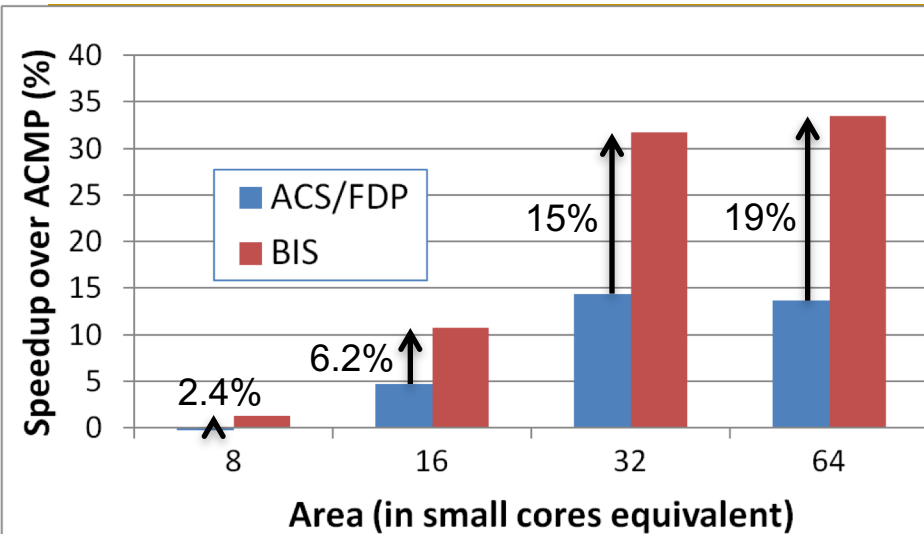
Fraction of execution time spent on predicted-important bottlenecks



Actually critical

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

# BIS Scaling Results



Performance increases with:

## 1) More small cores

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

## 2) More large cores

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

# BIS Summary

---

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

# 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



# Handling Private Data Locality: Data Marshaling

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

# Staged Execution Model (I)

---

- Goal: speed up a program by dividing it up into pieces
  - Idea
    - Split program code into *segments*
    - Run each segment on the core best-suited to run it
    - Each core assigned a work-queue, storing segments to be run
  - Benefits
    - Accelerates segments/critical-paths using specialized/heterogeneous cores
    - Exploits inter-segment parallelism
    - Improves locality of within-segment data
  - Examples
    - Accelerated critical sections, Bottleneck identification and scheduling
    - Producer-consumer pipeline parallelism
    - Task parallelism (Cilk, Intel TBB, Apple Grand Central Dispatch)
    - Special-purpose cores and functional units
-

# Staged Execution Model (II)

---



# Staged Execution Model (III)

---

*Split code into segments*

**Segment S0**

LOAD X  
STORE Y  
STORE Y

**Segment S1**

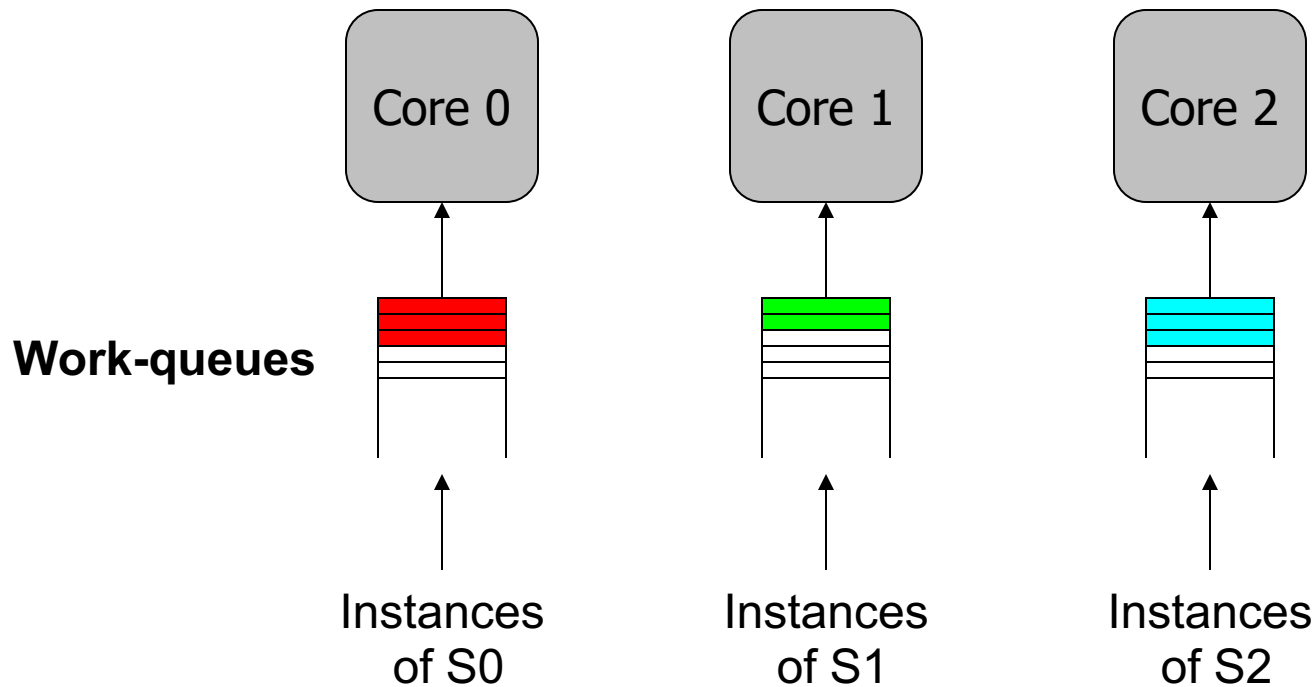
LOAD Y  
....  
STORE Z

**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 previous stage → each stage runs on a different core
    - Segment N: Stage N
  - Benefit: Stage-level parallelism, better locality → faster execution

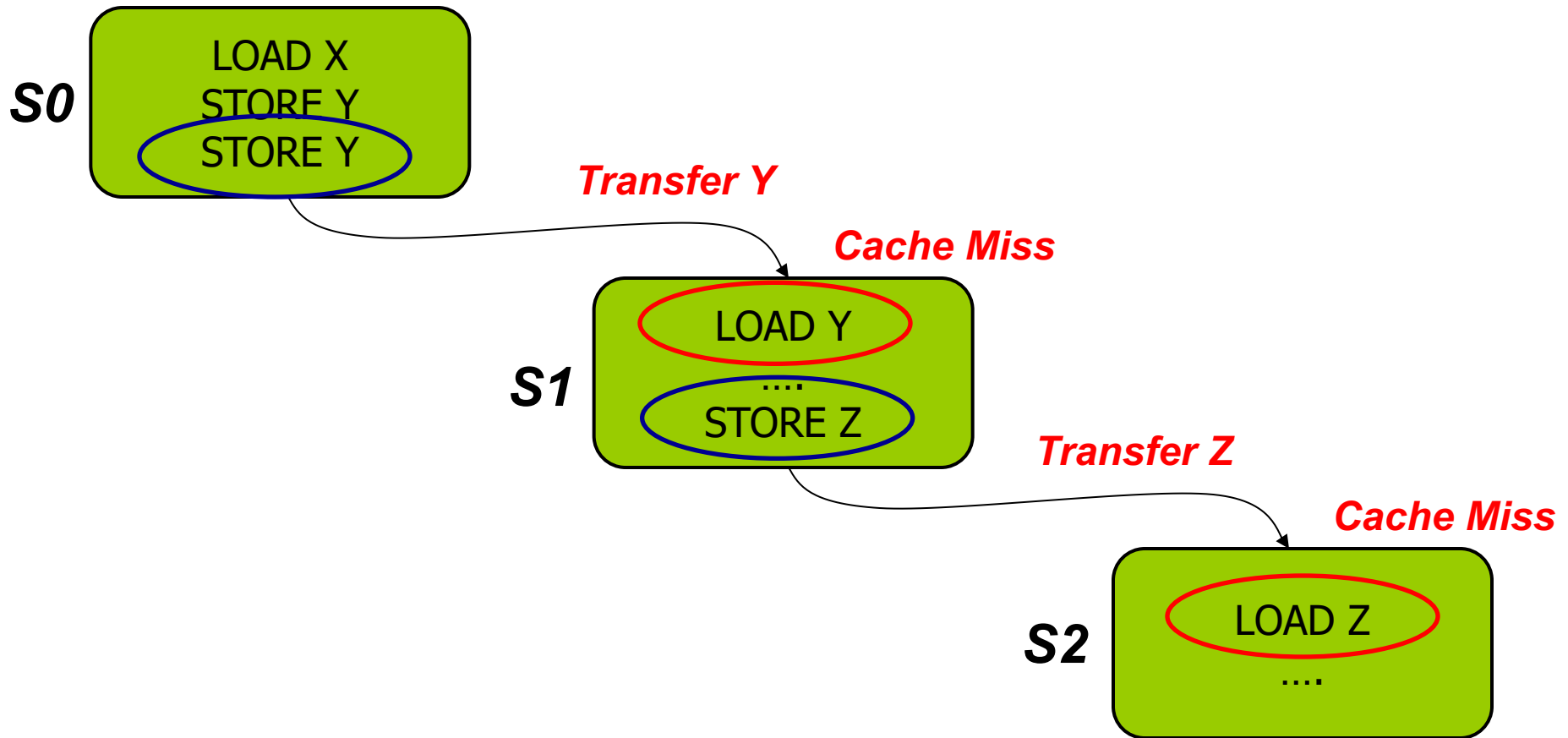
# Problem: Locality of Inter-segment Data

---

**Core 0**

**Core 1**

**Core 2**





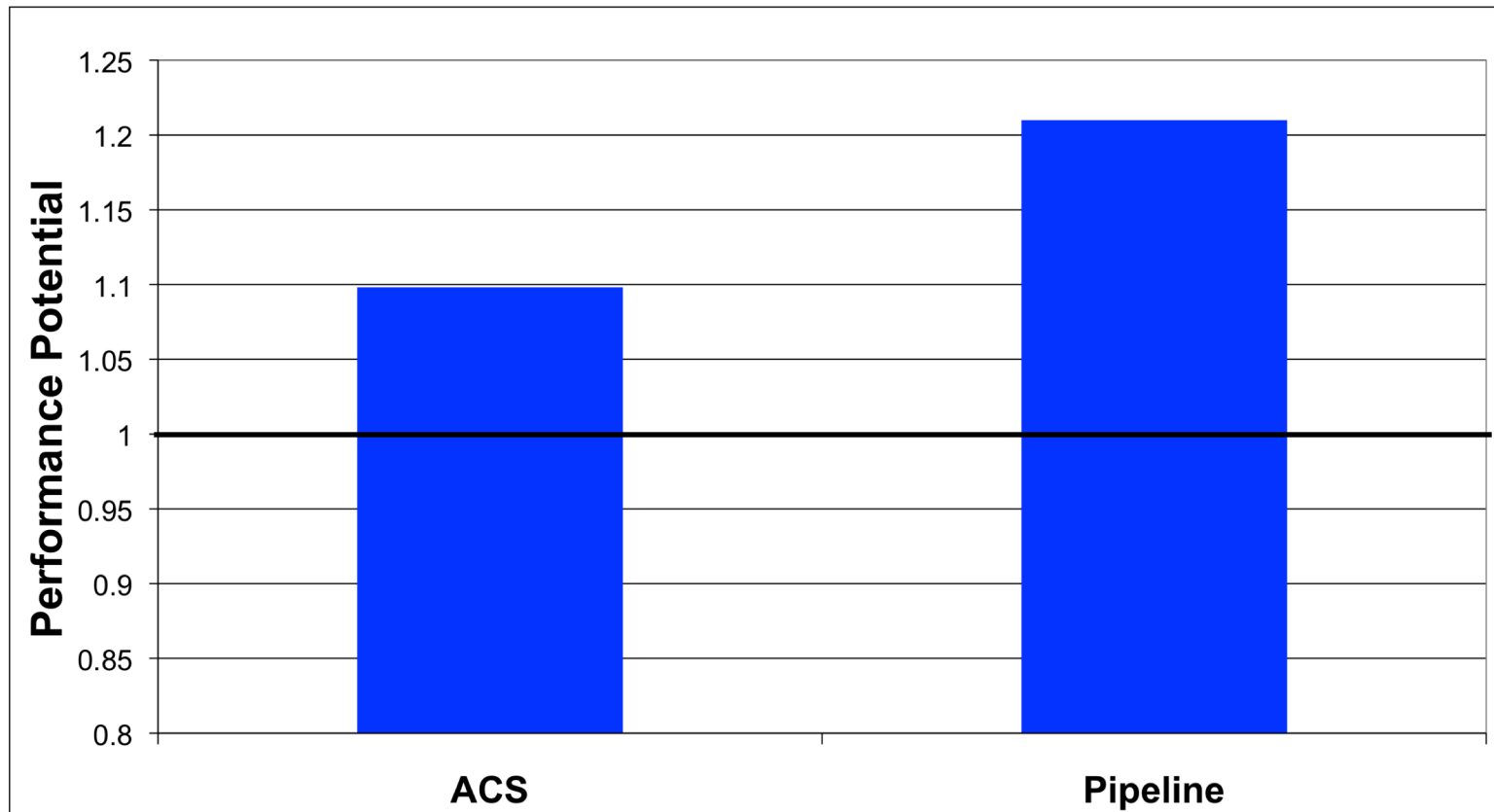
# Problem: Locality of Inter-segment Data

---

- Accelerated Critical Sections [Suleman et al., ASPLOS 2010]
  - Idea: Ship critical sections to a large core in an ACMP
  - Problem: Critical section incurs a cache miss when it touches data produced in the non-critical section (i.e., thread private data)
- Producer-Consumer Pipeline Parallelism
  - Idea: Split a loop iteration into multiple “pipeline stages” → each stage runs on a different core
  - Problem: A stage incurs a cache miss when it touches data produced by the previous stage
- Performance of Staged Execution limited by inter-segment cache misses

# What if We Eliminated All Inter-segment Misses?

---

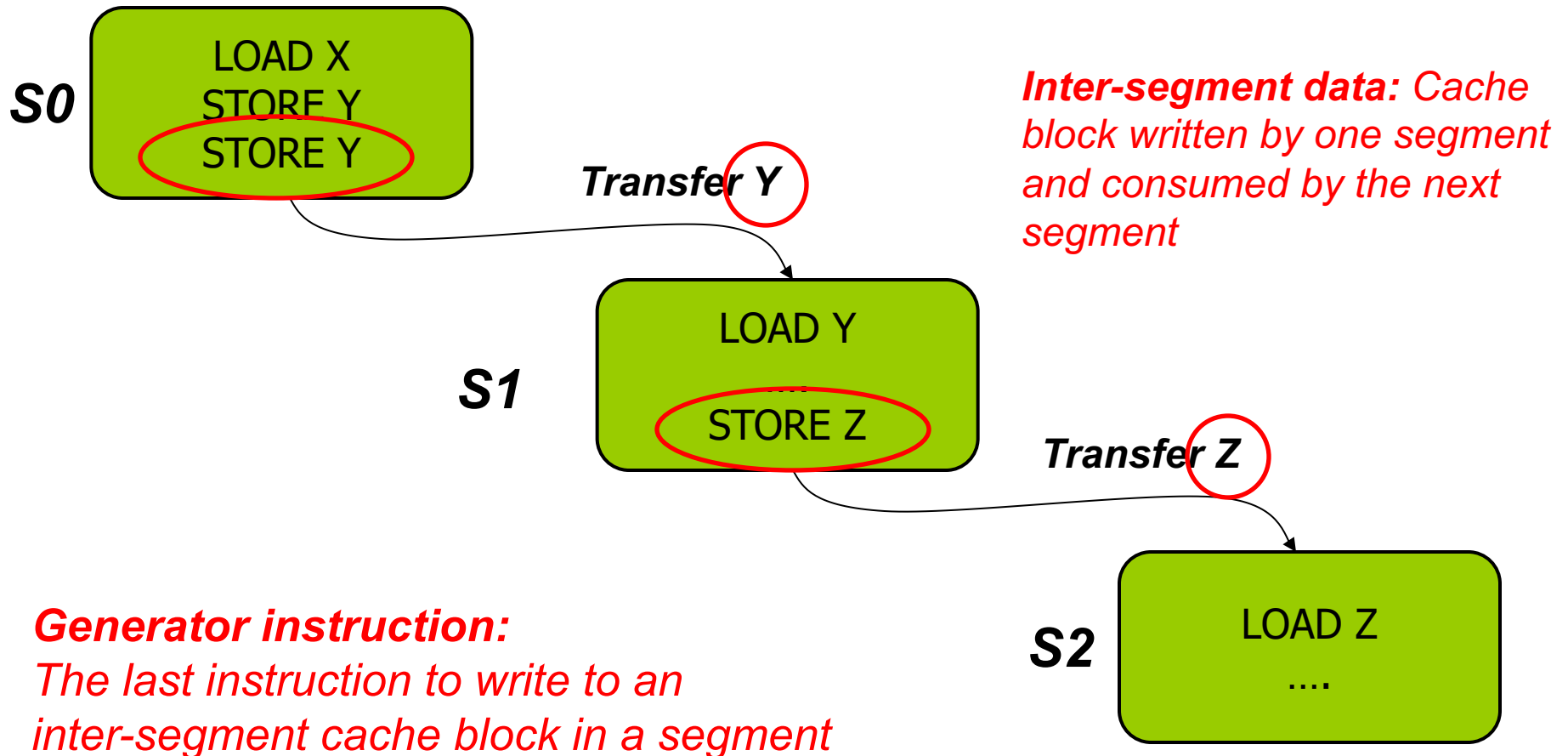


# Terminology

**Core 0**

**Core 1**

**Core 2**



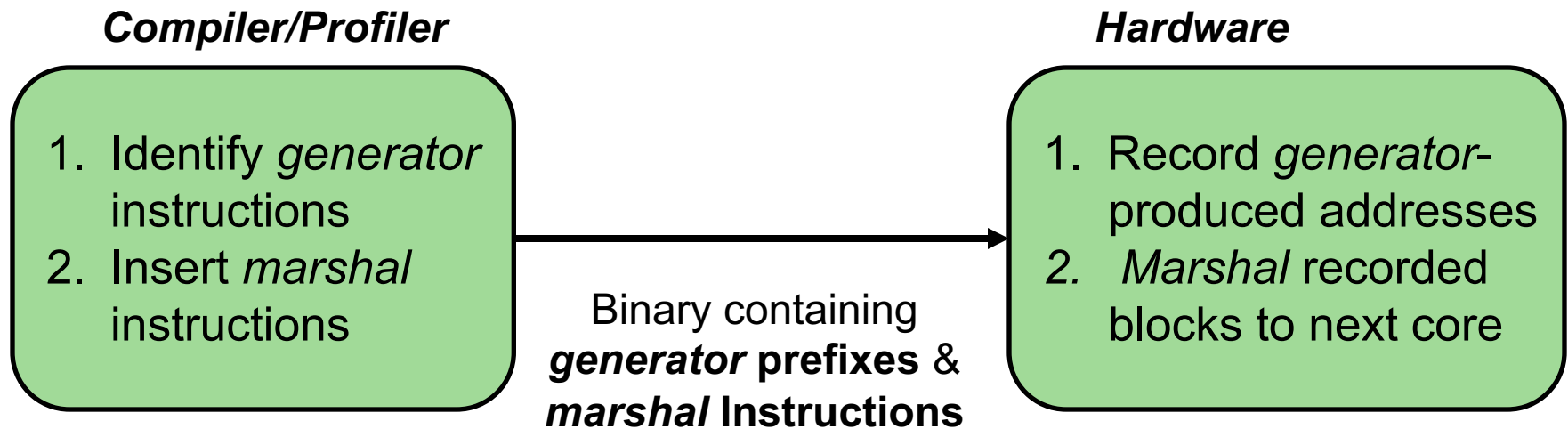
# Key Observation and Idea

---

- Observation: Set of generator instructions is stable over execution time and across input sets
  
- Idea:
  - Identify the generator instructions
  - Record cache blocks produced by generator instructions
  - Proactively send such cache blocks to the next segment's core before initiating the next segment
  
- Suleman et al., “Data Marshaling for Multi-Core Architectures,” ISCA 2010, IEEE Micro Top Picks 2011.

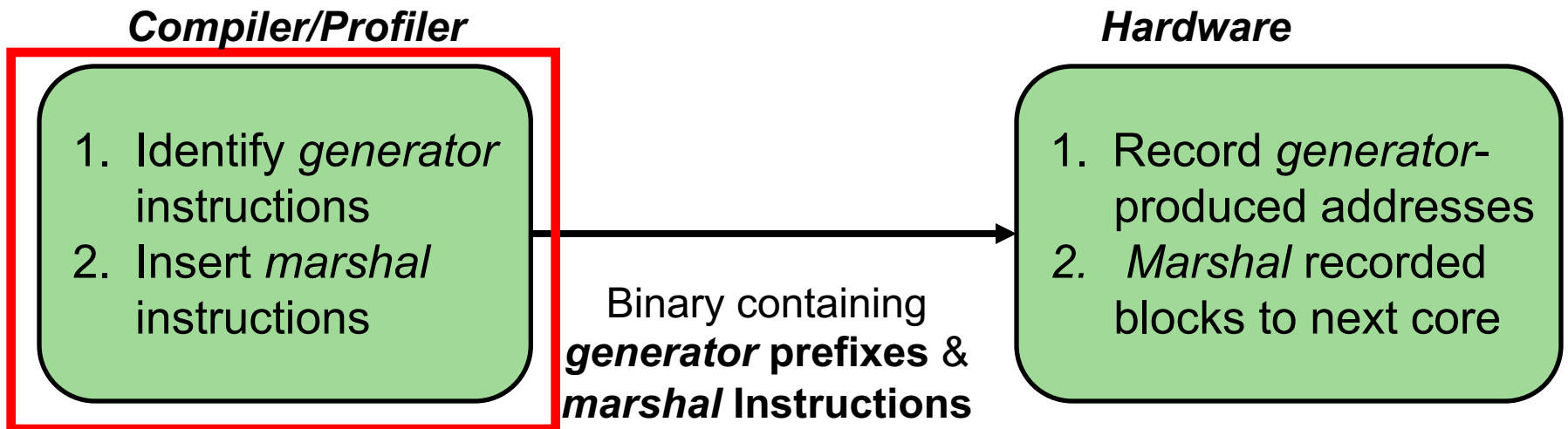
# Data Marshaling

---



# Data Marshaling

---

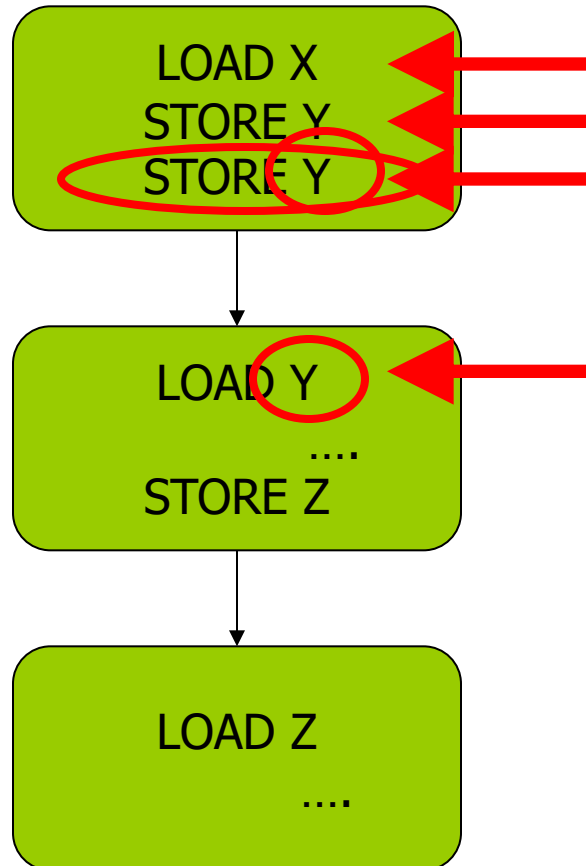


# Profiling Algorithm

---

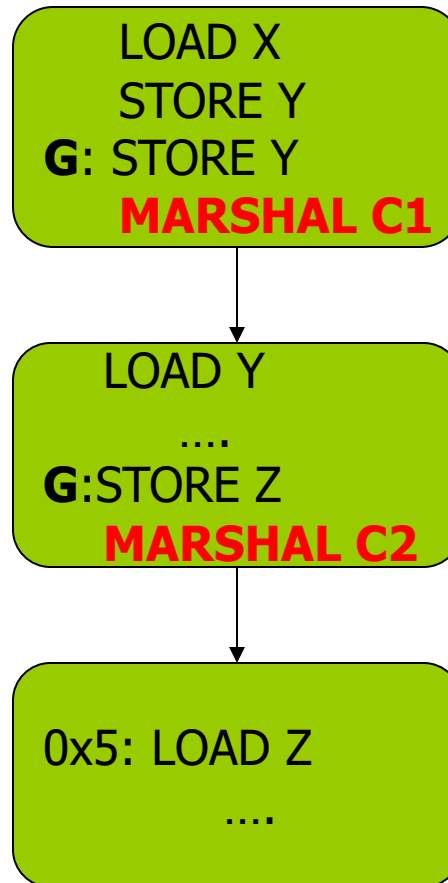
*Inter-segment data*

*Mark as Generator  
Instruction*



# Marshal Instructions

---



*When to send (Marshal)*

*Where to send (C1)*



# DM Support/Cost

---

- Profiler/Compiler: Generators, marshal instructions
- ISA: Generator prefix, marshal instructions
- Library/Hardware: Bind next segment ID to a physical core
  
- Hardware
  - Marshal Buffer
    - Stores physical addresses of cache blocks to be marshaled
    - 16 entries enough for almost all workloads → 96 bytes per core
  - Ability to execute generator prefixes and marshal instructions
  - Ability to push data to another cache

# DM: Advantages, Disadvantages

---

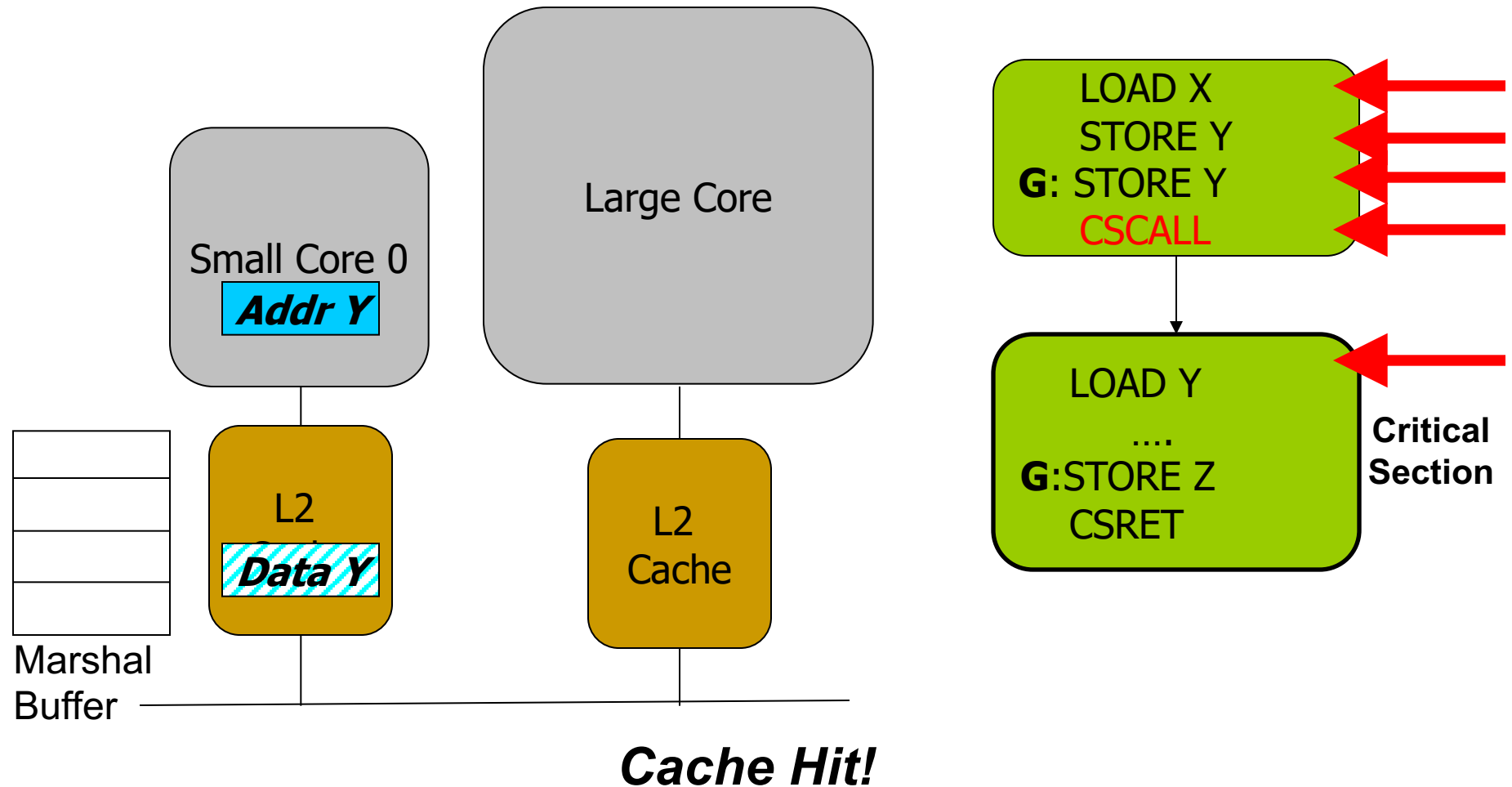
## ■ Advantages

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

## ■ Disadvantages

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

# Accelerated Critical Sections with DM

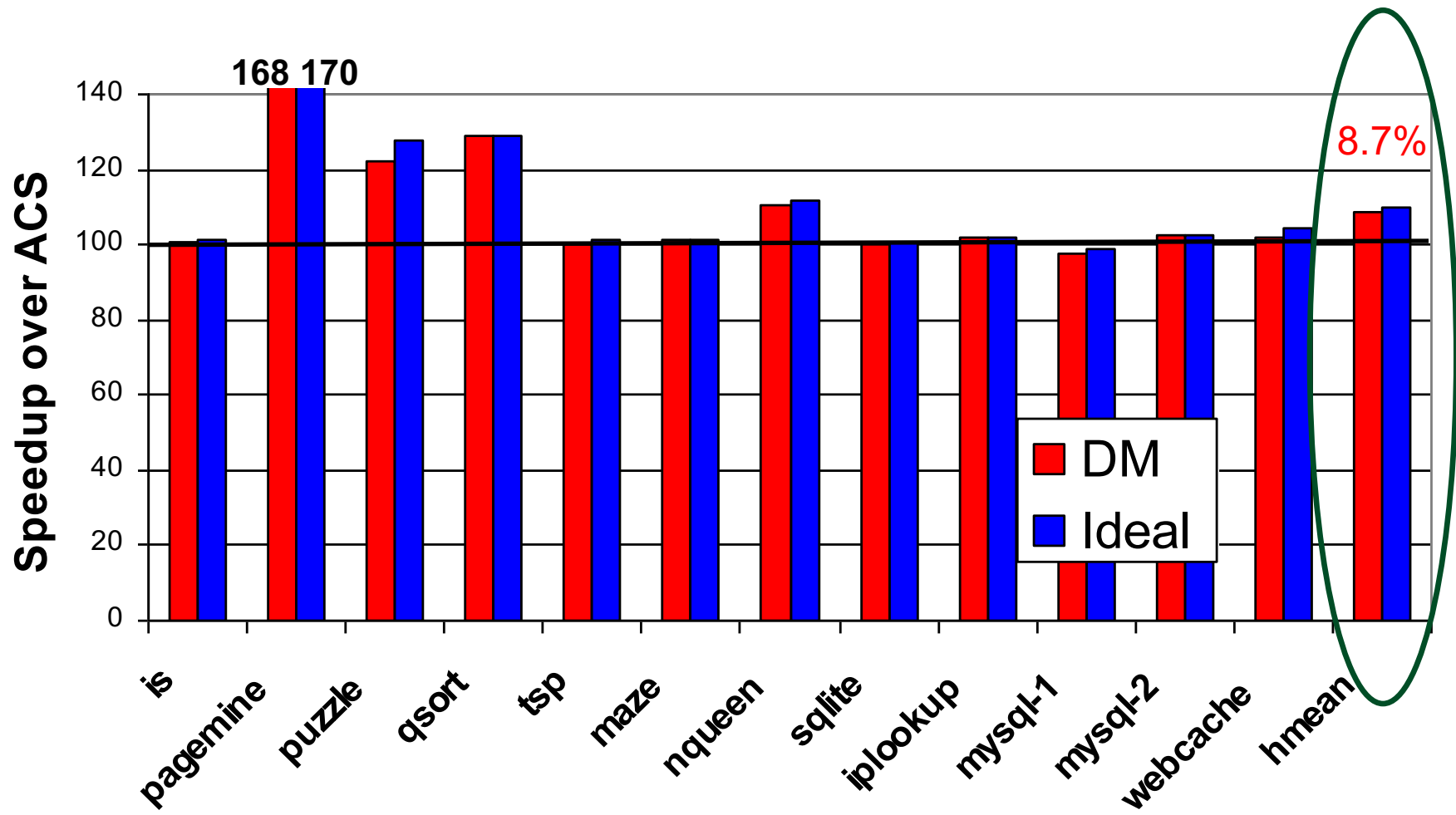


# Accelerated Critical Sections: Methodology

---

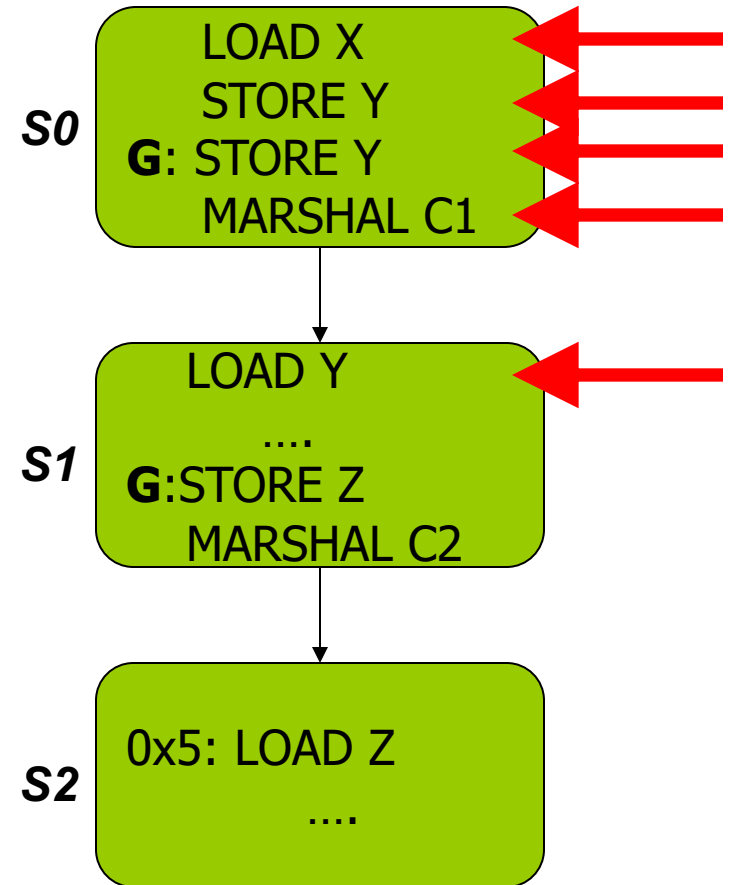
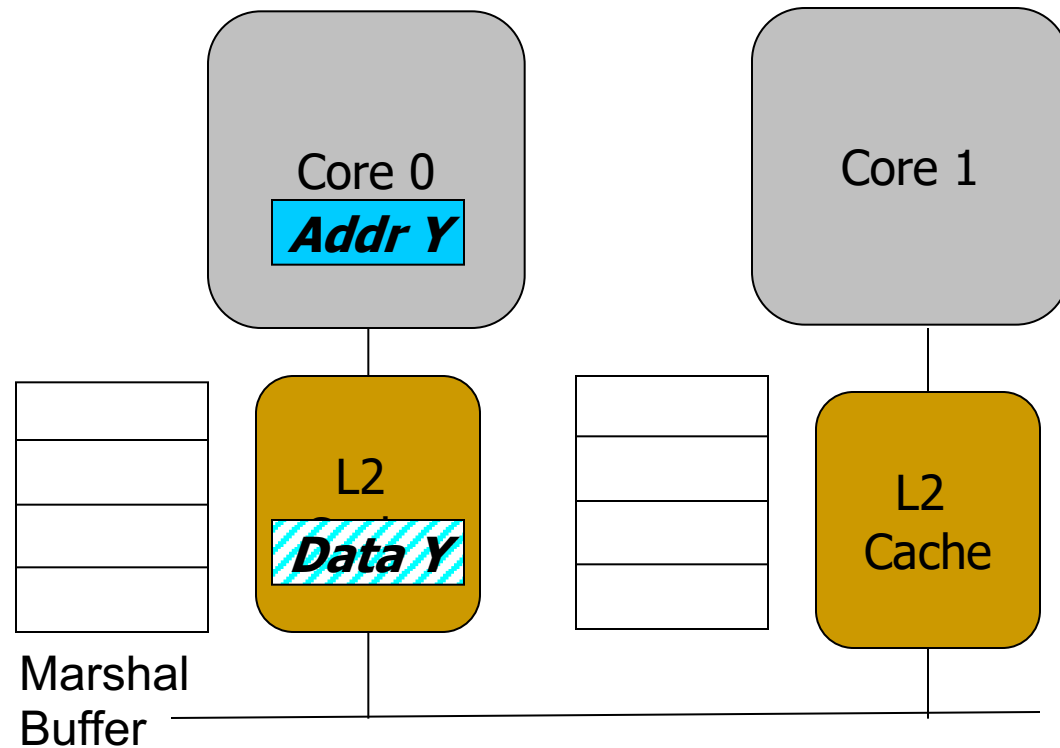
- Workloads: 12 critical section intensive applications
  - Data mining kernels, sorting, database, web, networking
  - Different training and simulation input sets
- Multi-core x86 simulator
  - 1 large and 28 small cores
  - Aggressive stream prefetcher employed at each core
- Details:
  - Large core: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

# DM on Accelerated Critical Sections: Results



# Pipeline Parallelism

**Cache Hit!**

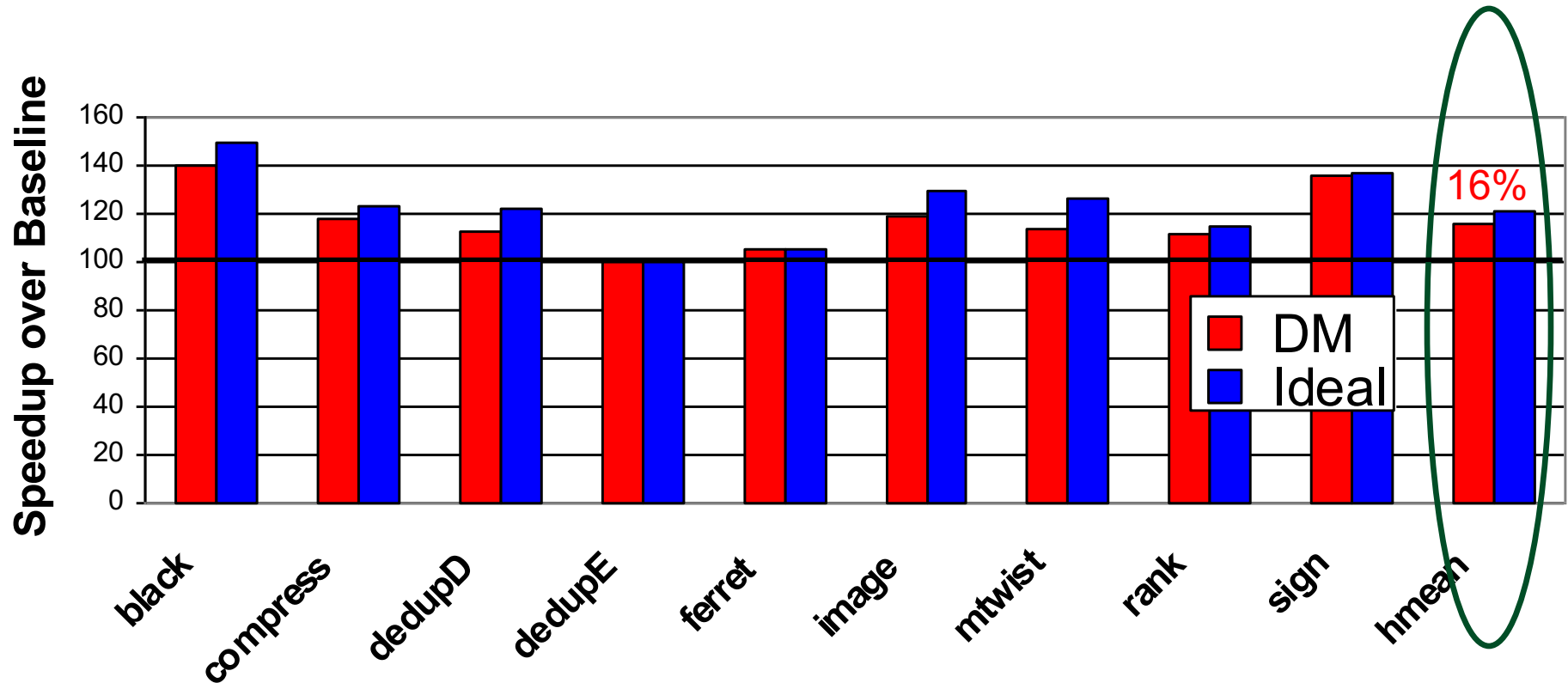


# Pipeline Parallelism: Methodology

---

- Workloads: 9 applications with pipeline parallelism
  - Financial, compression, multimedia, encoding/decoding
  - Different training and simulation input sets
- Multi-core x86 simulator
  - 32-core CMP: 2GHz, in-order, 2-wide, 5-stage
  - Aggressive stream prefetcher employed at each core
  - Private 32 KB L1, private 256KB L2, 8MB shared L3
  - On-chip interconnect: Bi-directional ring, 5-cycle hop latency

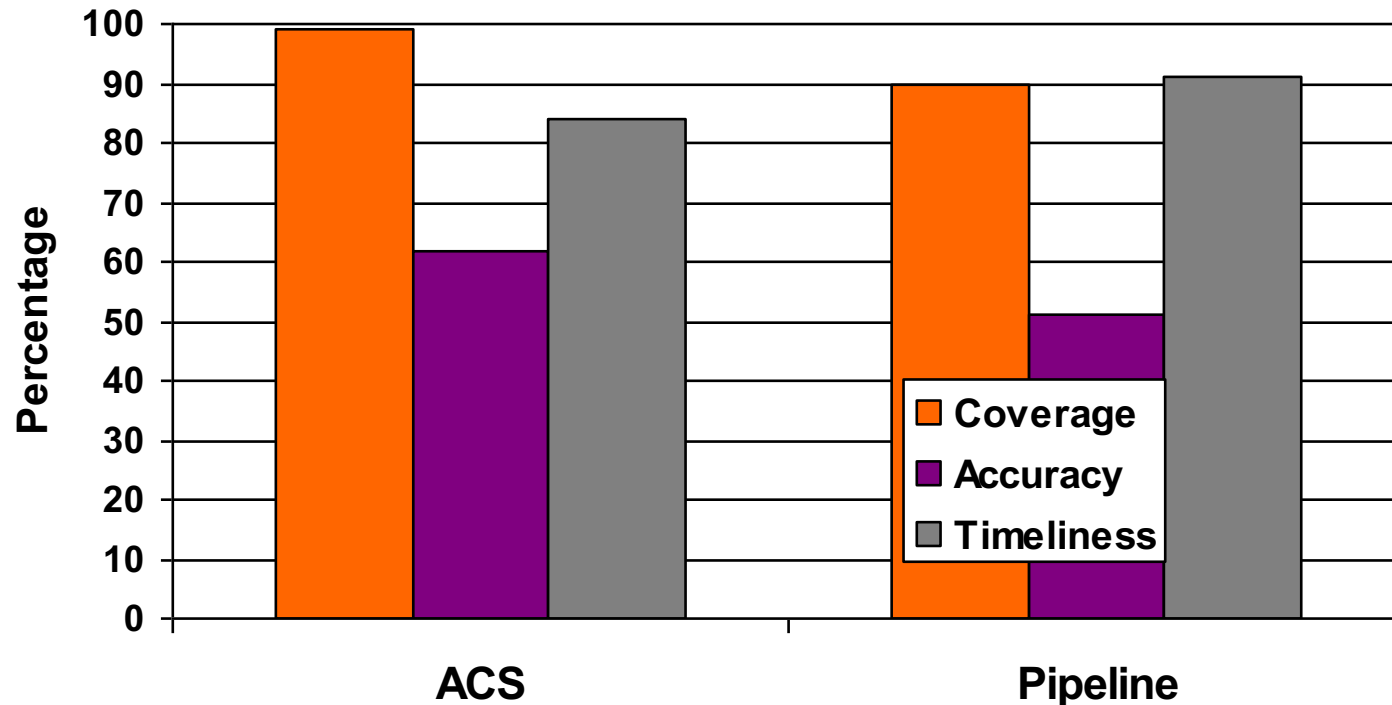
# DM on Pipeline Parallelism: Results





# DM Coverage, Accuracy, Timeliness

---



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

# Scaling Results

---

- DM performance improvement increases with
  - More cores
  - Higher interconnect latency
  - Larger private L2 caches
- Why? Inter-segment data misses become a larger bottleneck
  - More cores → More communication
  - Higher latency → Longer stalls due to communication
  - Larger L2 cache → Communication misses remain

# Other Applications of Data Marshaling

---

- Can be applied to other Staged Execution models
  - Task parallelism models
    - Cilk, Intel TBB, Apple Grand Central Dispatch
  - Special-purpose remote functional units
  - Computation spreading [Chakraborty et al., ASPLOS' 06]
  - Thread motion/migration [e.g., Rangan et al., ISCA' 09]
- Can be an enabler for more aggressive SE models
  - Lowers the cost of data migration
    - an important overhead in remote execution of code segments
  - Remote execution of finer-grained tasks can become more feasible → finer-grained parallelization in multi-cores

# Data Marshaling Summary

---

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

# 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<sup>†</sup>   Onur Mutlu<sup>§</sup>   José A. Joao<sup>†</sup>   Khubaib<sup>†</sup>   Yale N. Patt<sup>†</sup>

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

<sup>§</sup>Carnegie Mellon University  
onur@cmu.edu

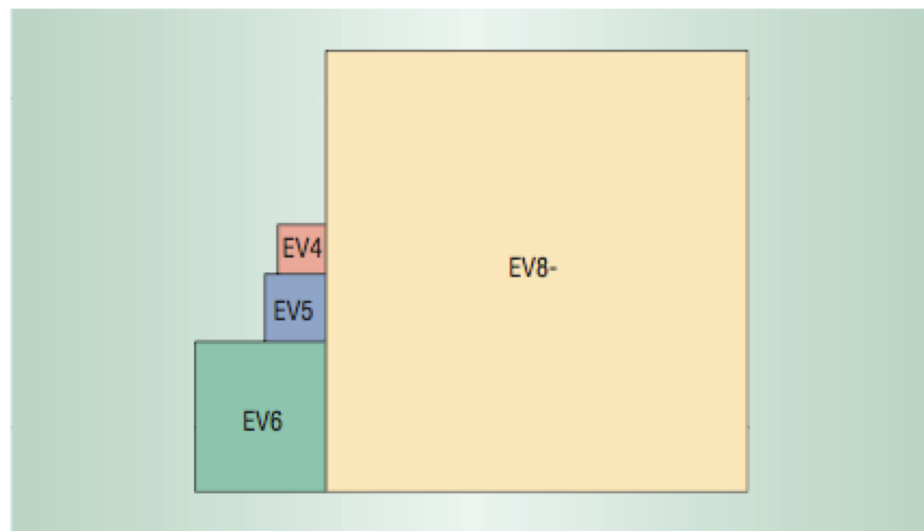
# Other Uses of Asymmetry

# 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

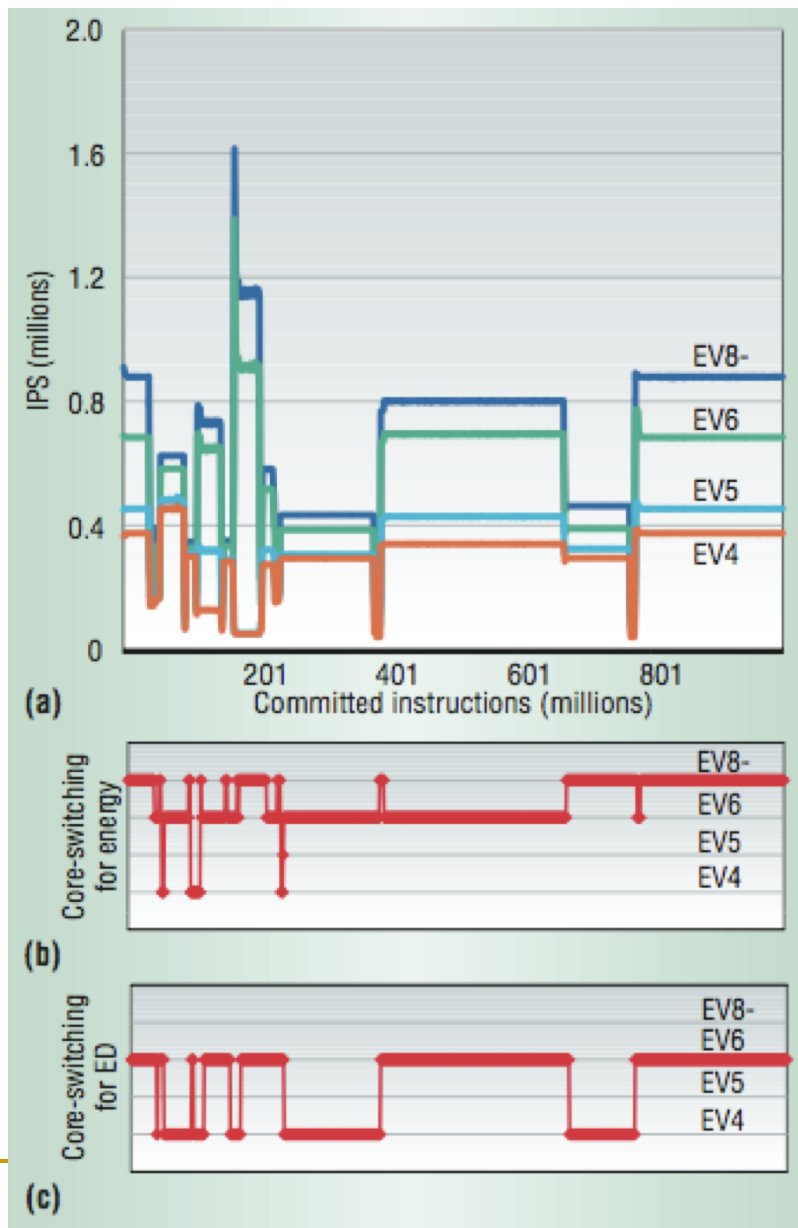
# Use of Asymmetry for Energy Efficiency



**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 19b:

# Heterogeneous Multi-Core Systems

Prof. Onur Mutlu

ETH Zürich

Fall 2018

28 November 2018

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

# 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
  - $P = \text{EPI} \bullet \text{IPS}$
  - $P = \text{fixed power budget}$
  - $\text{EPI} = \text{energy per instruction}$
  - $\text{IPS} = \text{aggregate instructions retired per second}$
- 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

---

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

Symmetric

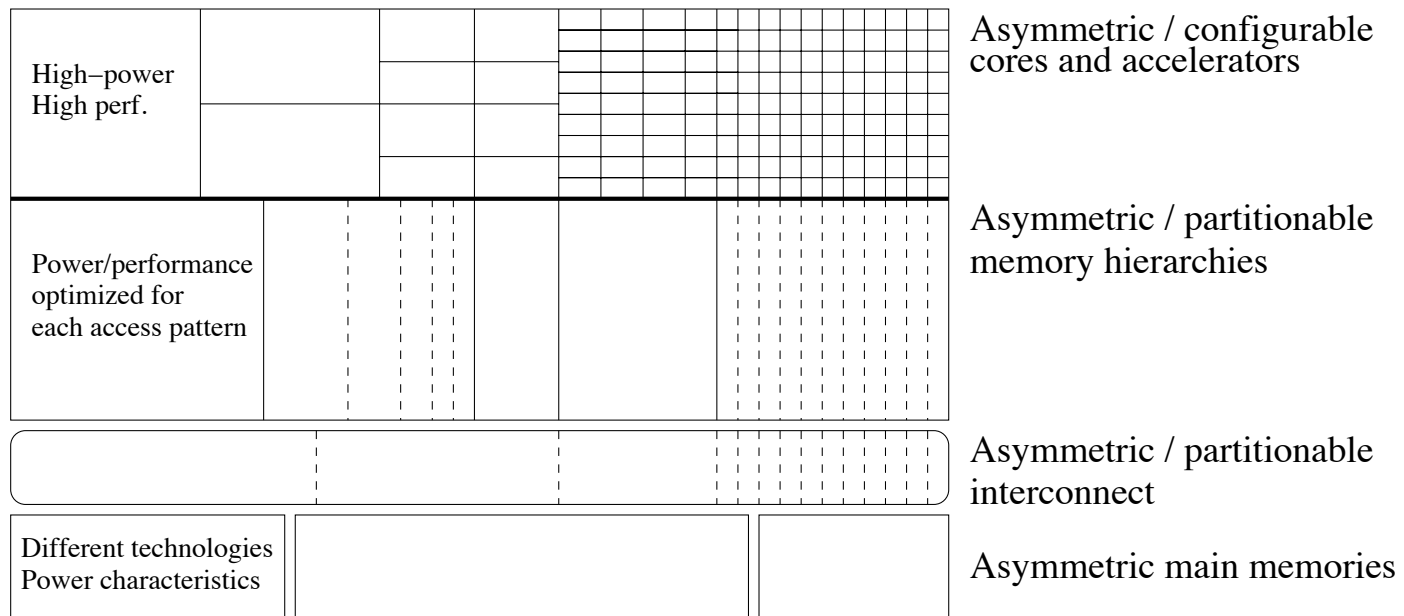
C1		C2	
		C3	
C4	C4	C4	C4
C5	C5	C5	C5

Asymmetric

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

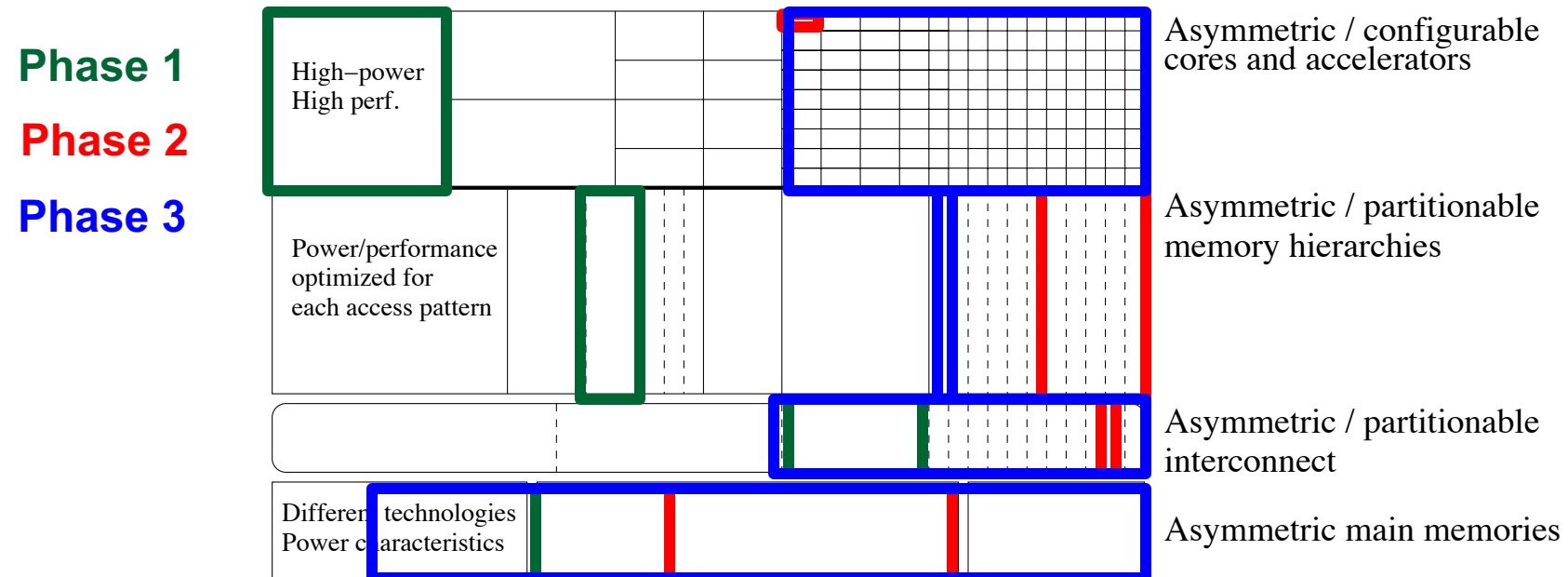
# Thought Experiment: Asymmetry Everywhere

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



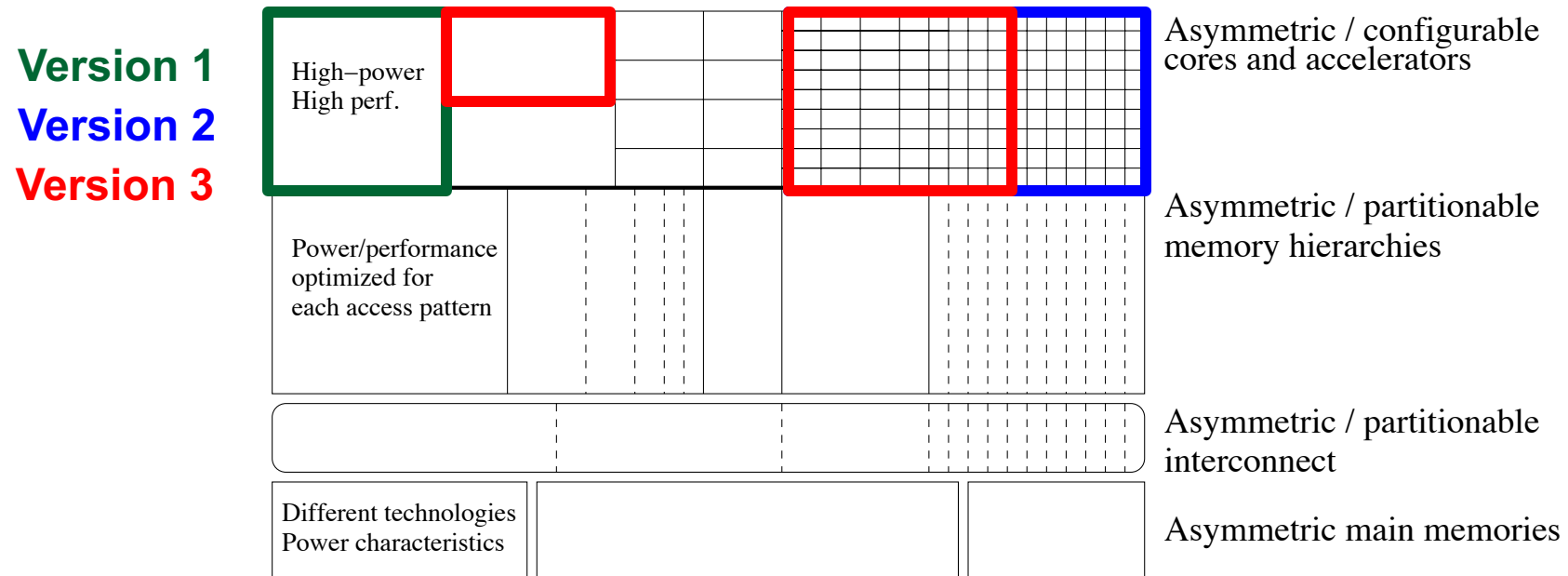
# Thought Experiment: Asymmetry Everywhere

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



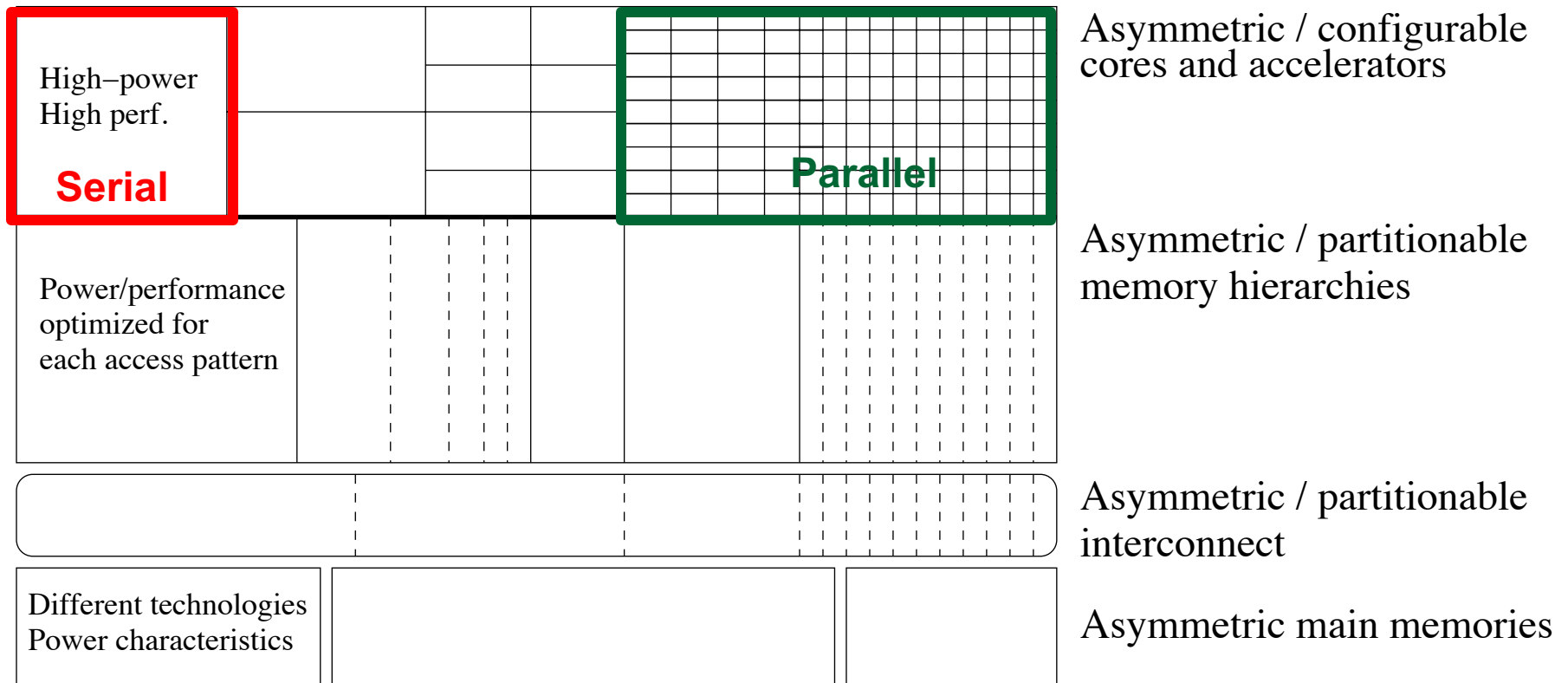
# Many Research and Design Questions

---

- 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

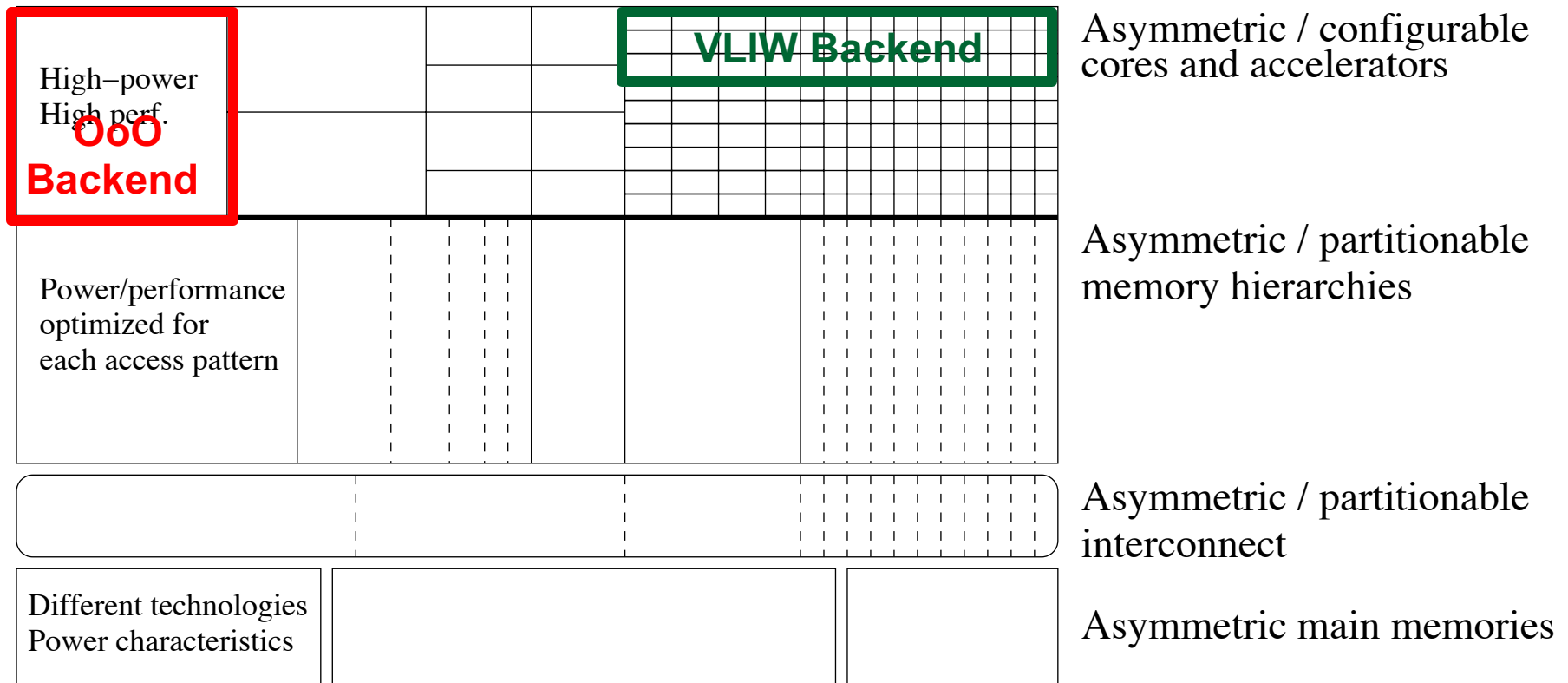


# Exploiting Asymmetry: Simple Examples



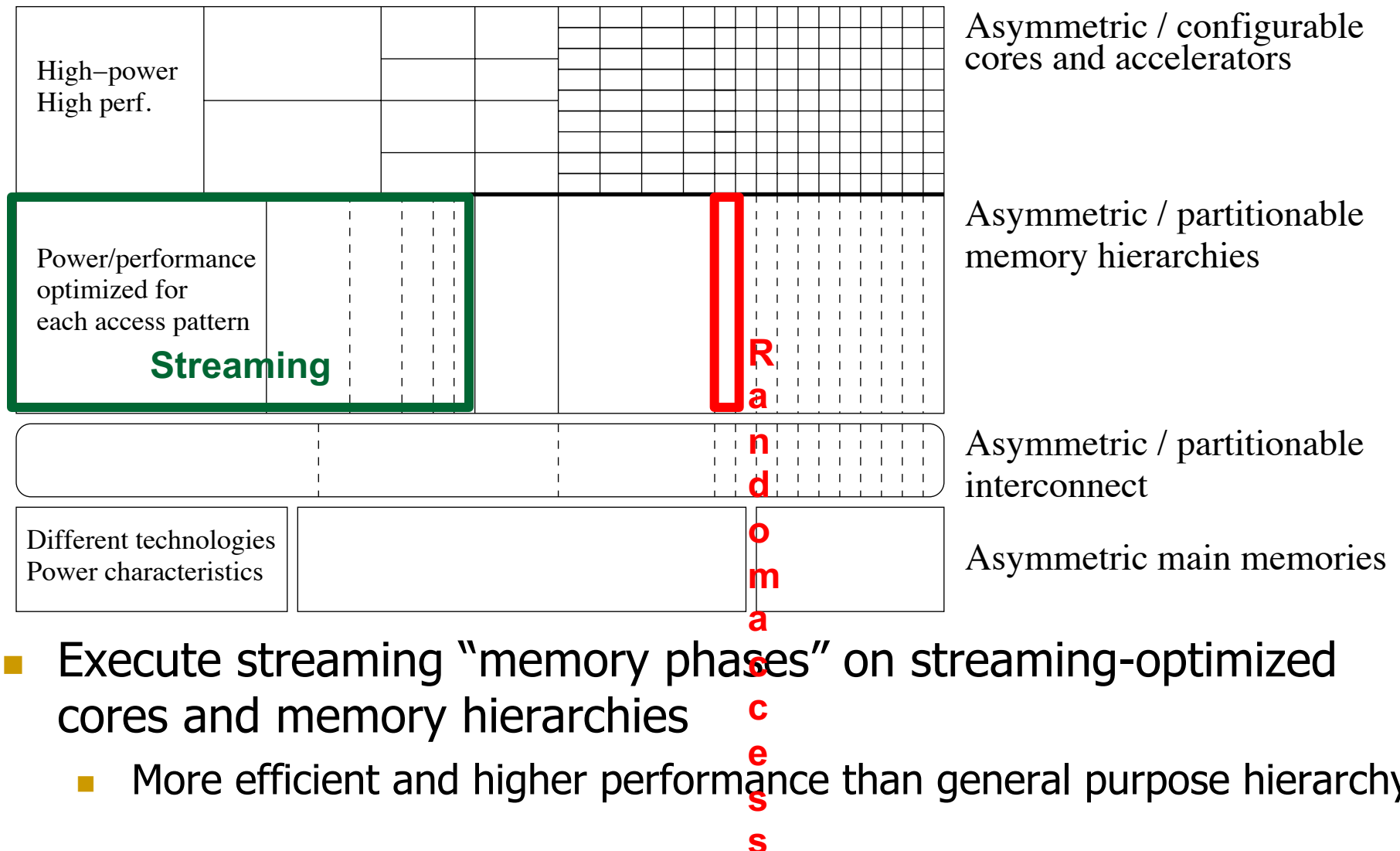
- Execute critical/serial sections on high-power, high-performance cores/resources [Suleman+ ASPLOS'09, ISCA'10, Top Picks'10'11, Joao+ ASPLOS'12, ISCA'13]
  - Programmer can write less optimized, but more likely correct programs

# Exploiting Asymmetry: Simple Examples



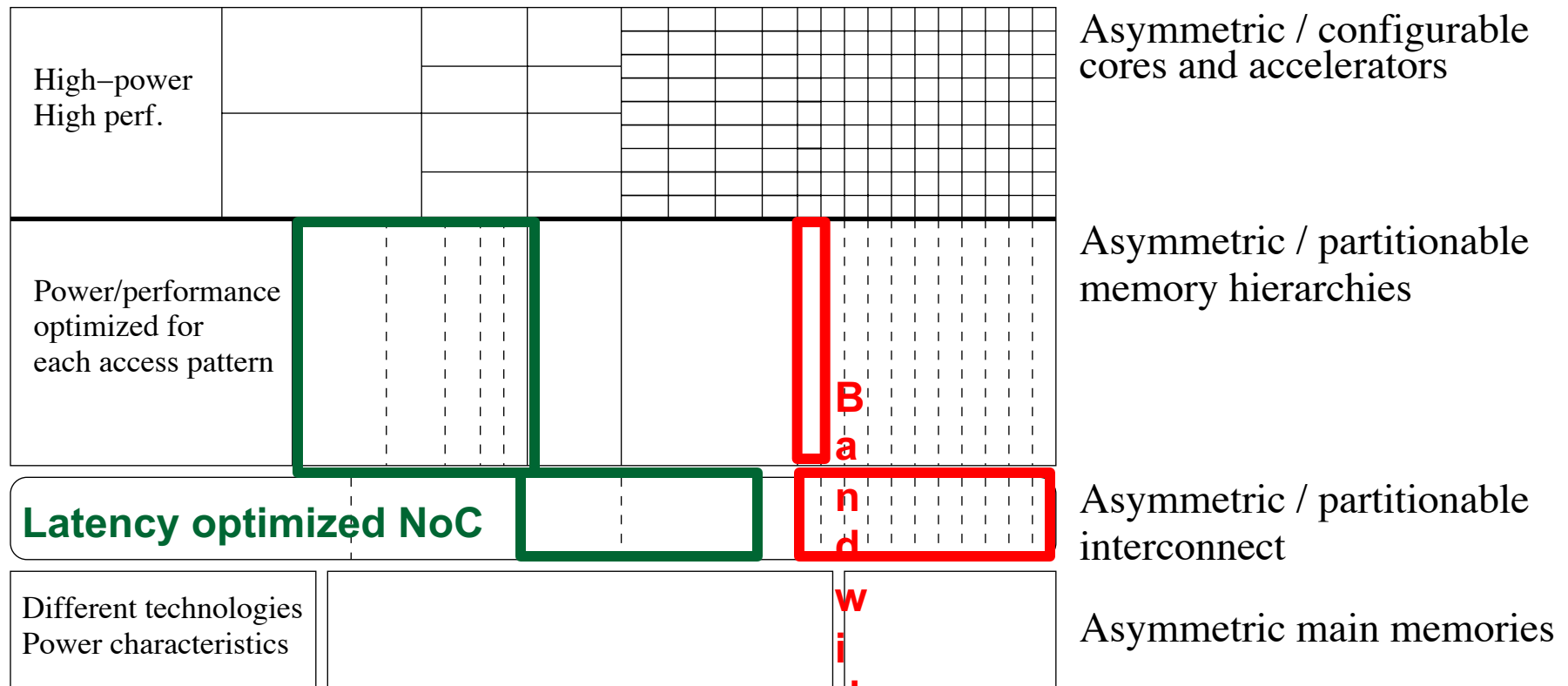
- Execute each code block on the most efficient execution backend for that block [Fallin+ ICCD'14]
  - Enables a much more efficient and still high performance core design

# Exploiting Asymmetry: Simple Examples



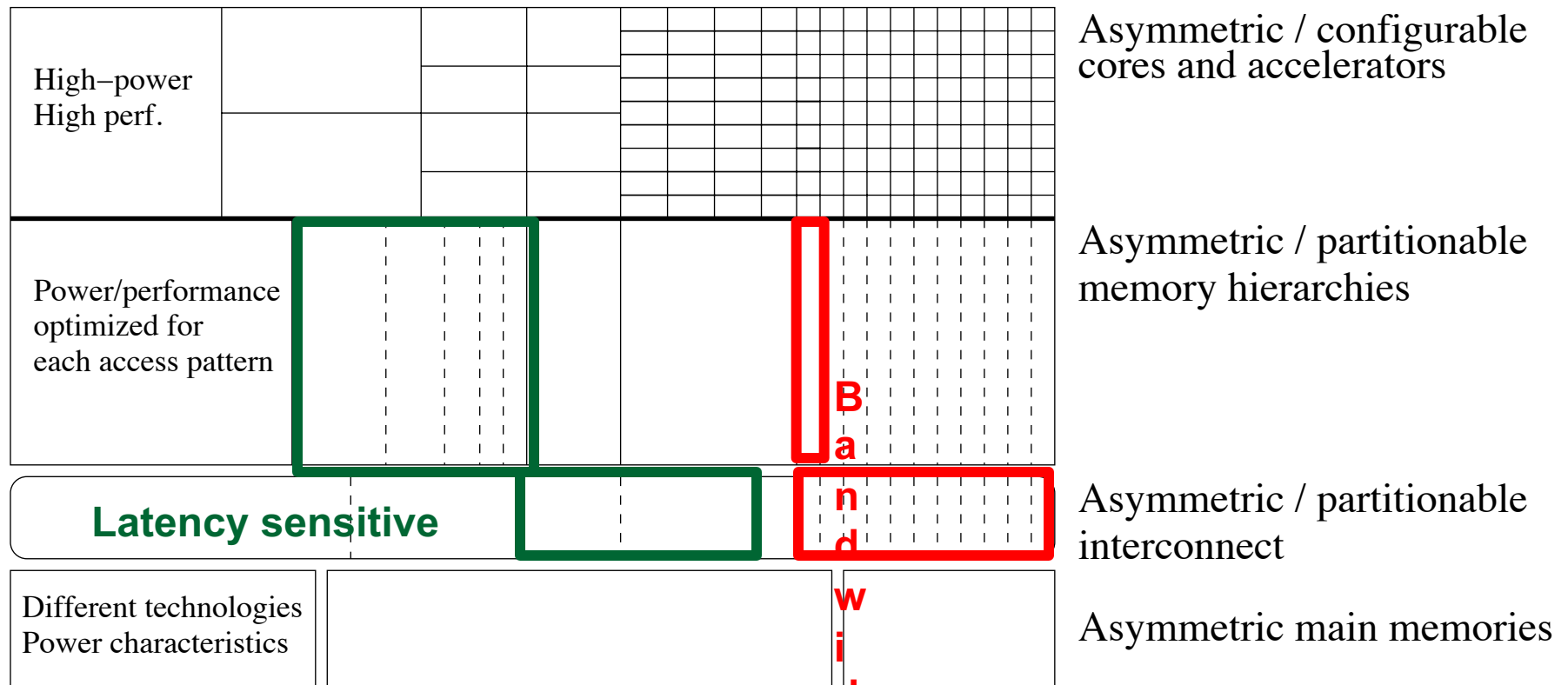
- Execute streaming "memory phases" on streaming-optimized cores and memory hierarchies
  - More efficient and higher performance than general purpose hierarchy

# Exploiting Asymmetry: Simple Examples



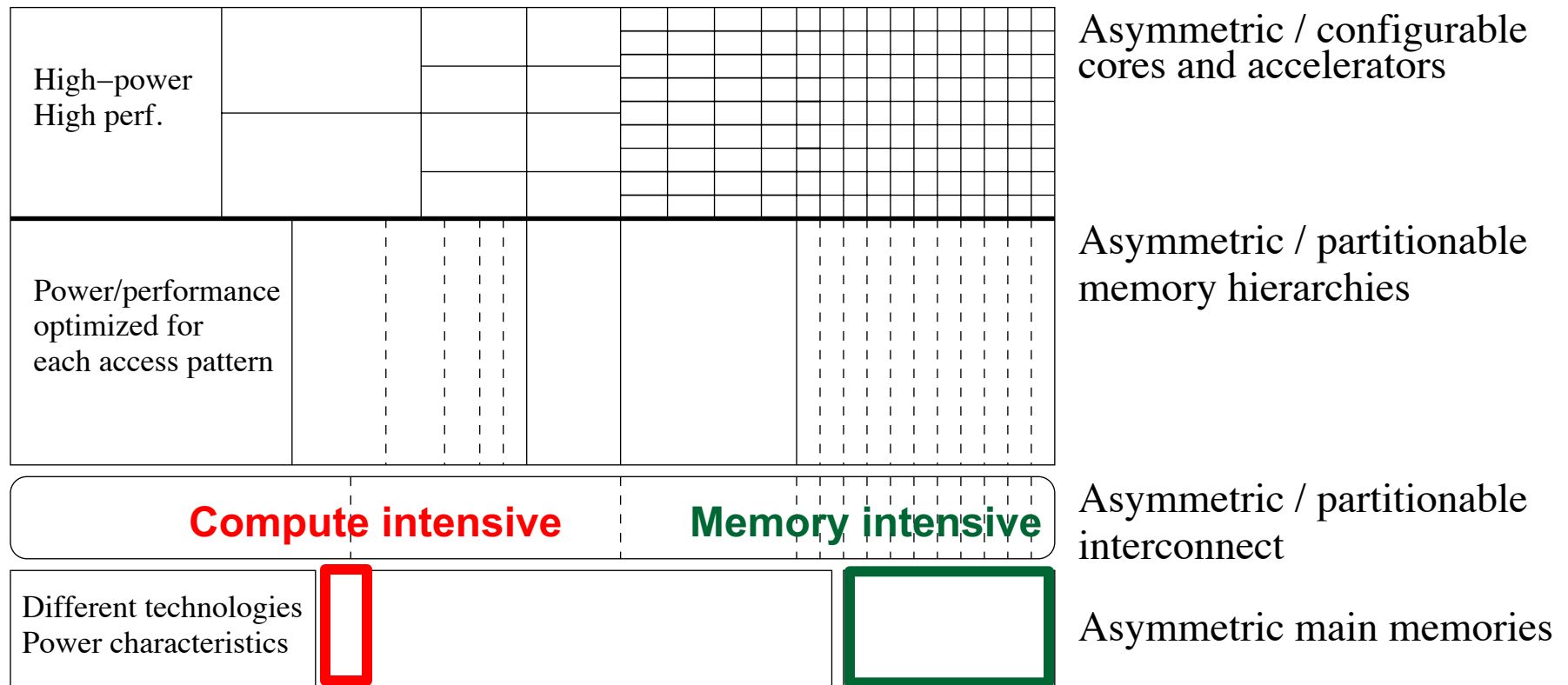
- Execute bandwidth-sensitive threads on a bandwidth-optimized network, latency-sensitive ones on a latency-optimized network [Das+ DAC'13]
  - Higher performance and energy-efficiency than a single network

# Exploiting Asymmetry: Simple Examples



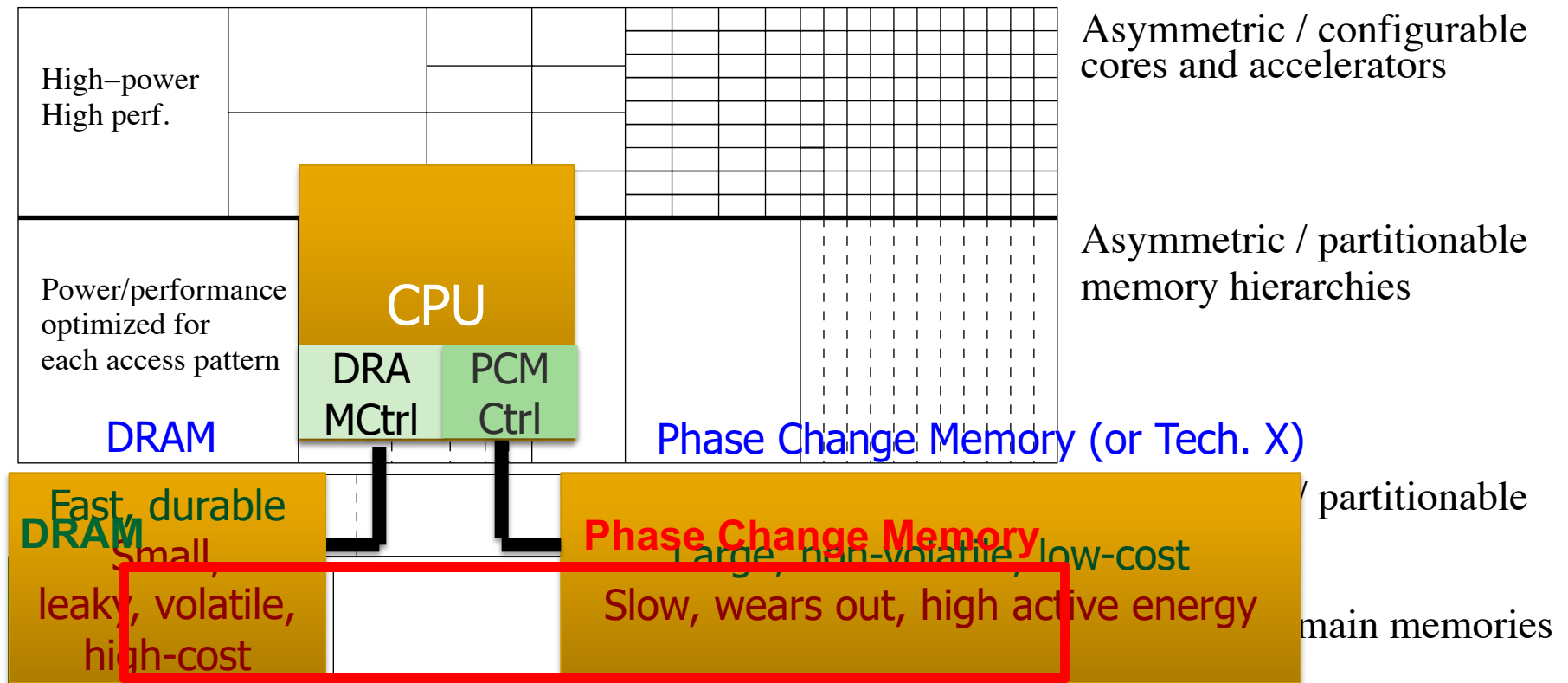
- Partition memory controller and on-chip network bandwidth asymmetrically among threads [Kim+ HPCA 2010, MICRO 2010, Top Picks 2011] [Nychis+ HotNets 2010] [Das+ MICRO 2009, ISCA 2010, Top Picks 2011]
  - Higher performance and energy-efficiency than symmetric/free-for-all

# Exploiting Asymmetry: Simple Examples



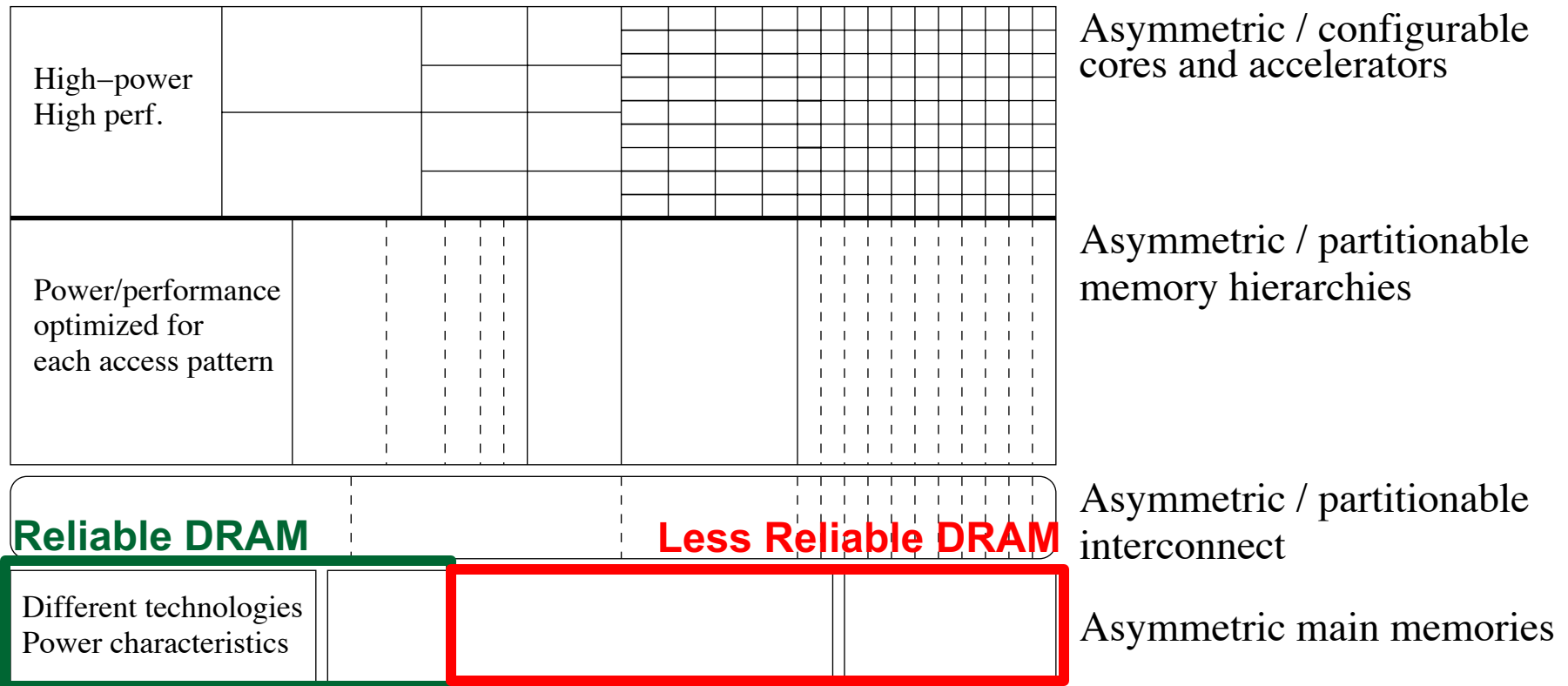
- Have multiple different memory scheduling policies apply them to different sets of threads based on thread behavior [Kim+ MICRO 2010, Top Picks 2011] [Ausavarungnirun+ ISCA 2012]
  - Higher performance and fairness than a homogeneous policy

# 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

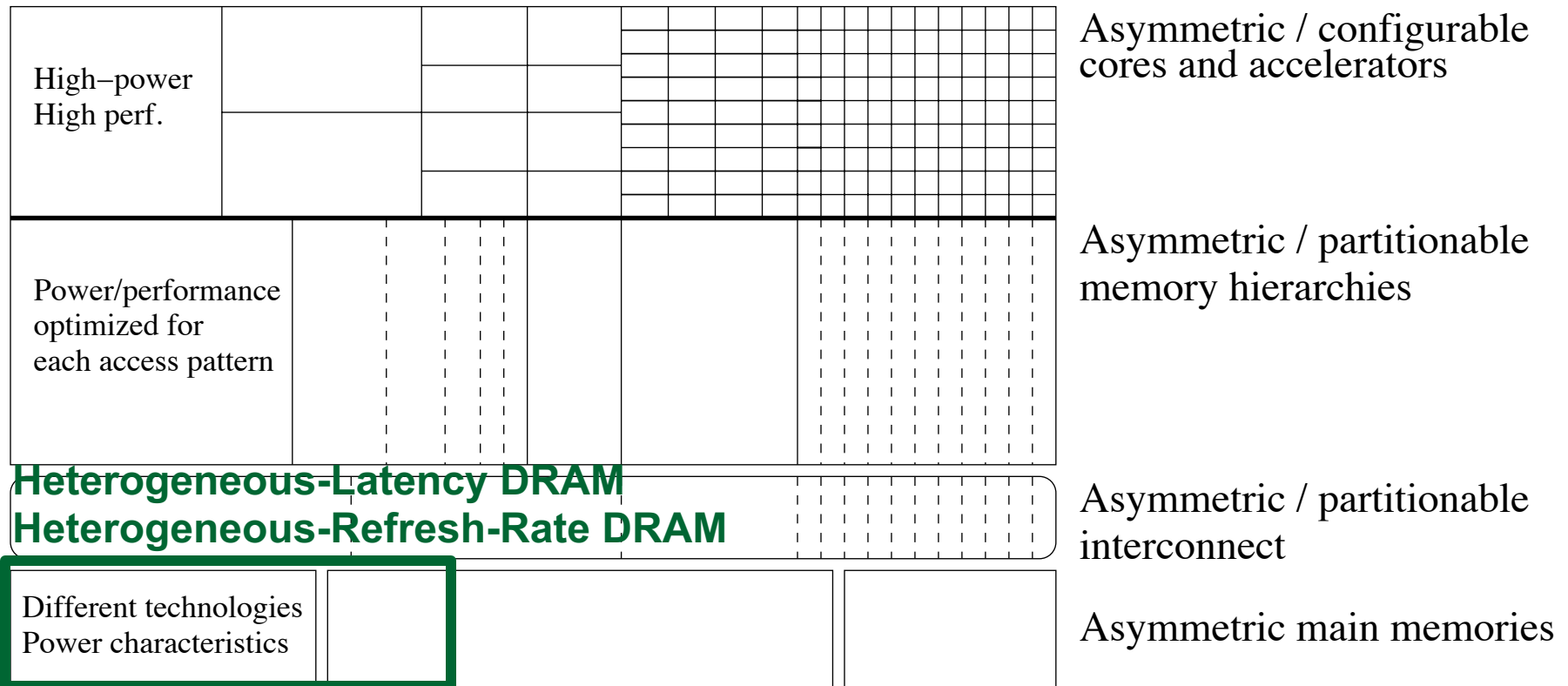
# Exploiting Asymmetry: Simple Examples



- 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