Architecting and Exploiting Asymmetry to Accelerate Bottlenecks in the Cloud

Onur Mutlu
onur@cmu.edu
November 29, 2012
ISTC Retreat, Pittsburgh, PA

SAFARI Carnegie Mellon

Warning

- This is an asymmetric talk
- Component 1: A case for asymmetry *everywhere*
- Component 2: A (relatively) deep dive into a specific mechanism to exploit asymmetry in cores
- Asymmetry = heterogeneity
  - A way to enable specialization/customization

The Cloud Setting

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

Shield the Programmer from Shared Resources

- Writing even sequential software is hard enough
  - Optimizing code for a complex shared-resource parallel system will be a nightmare for most programmers

- Programmer should not worry about (hardware) resource management
  - What should be executed where with what resources

- Future cloud computer architectures should be designed to
  - Minimize programmer effort to optimize (parallel) programs
  - Maximize runtime system's effectiveness in automatic shared resource management

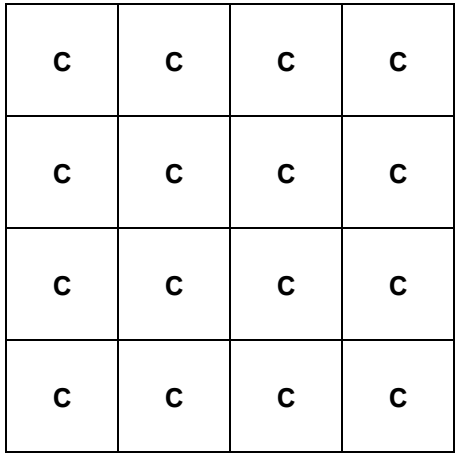
Shared Resource Management: Goals

- Future many-core systems should manage power and performance automatically across threads/applications

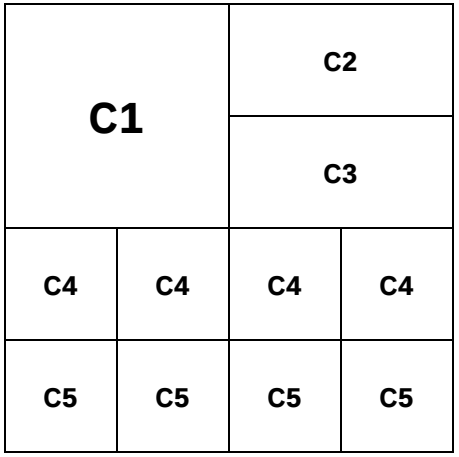
- Minimize energy/power consumption
- While satisfying performance/SLA requirements
  - Provide predictability and Quality of Service
- Minimize programmer effort
  - In creating optimized parallel programs

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

Asymmetry Enables Customization



Symmetric

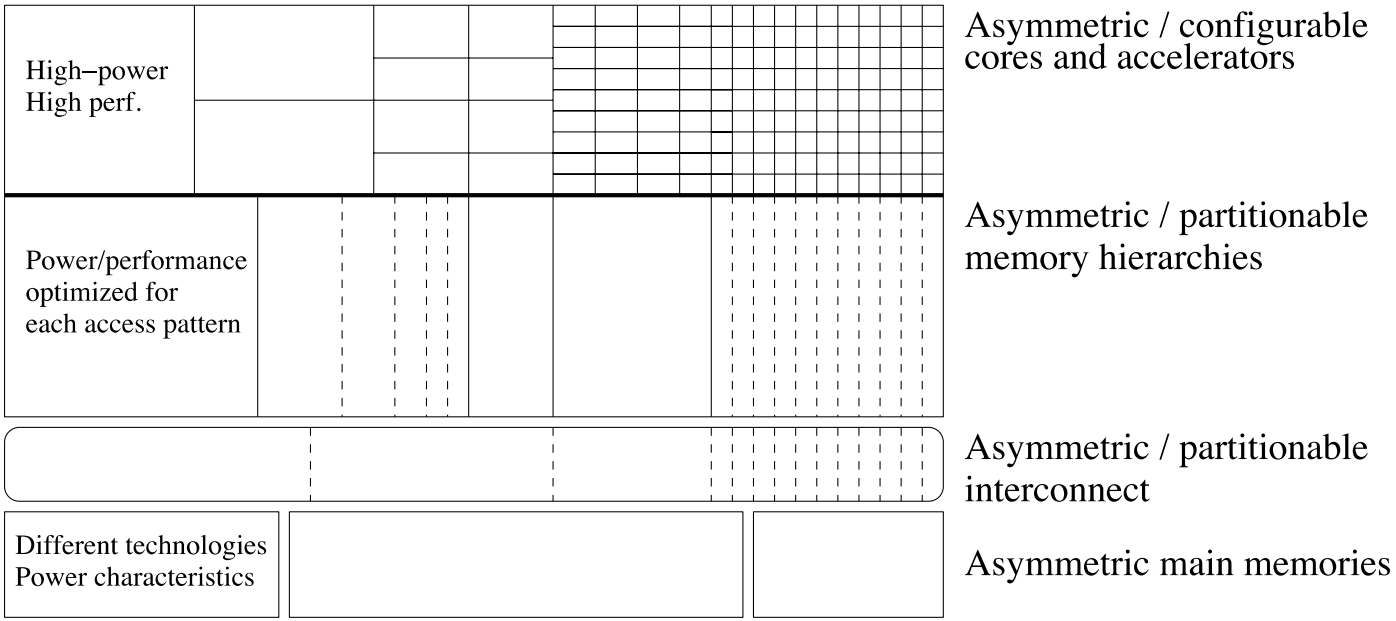


Asymmetric

- 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.)

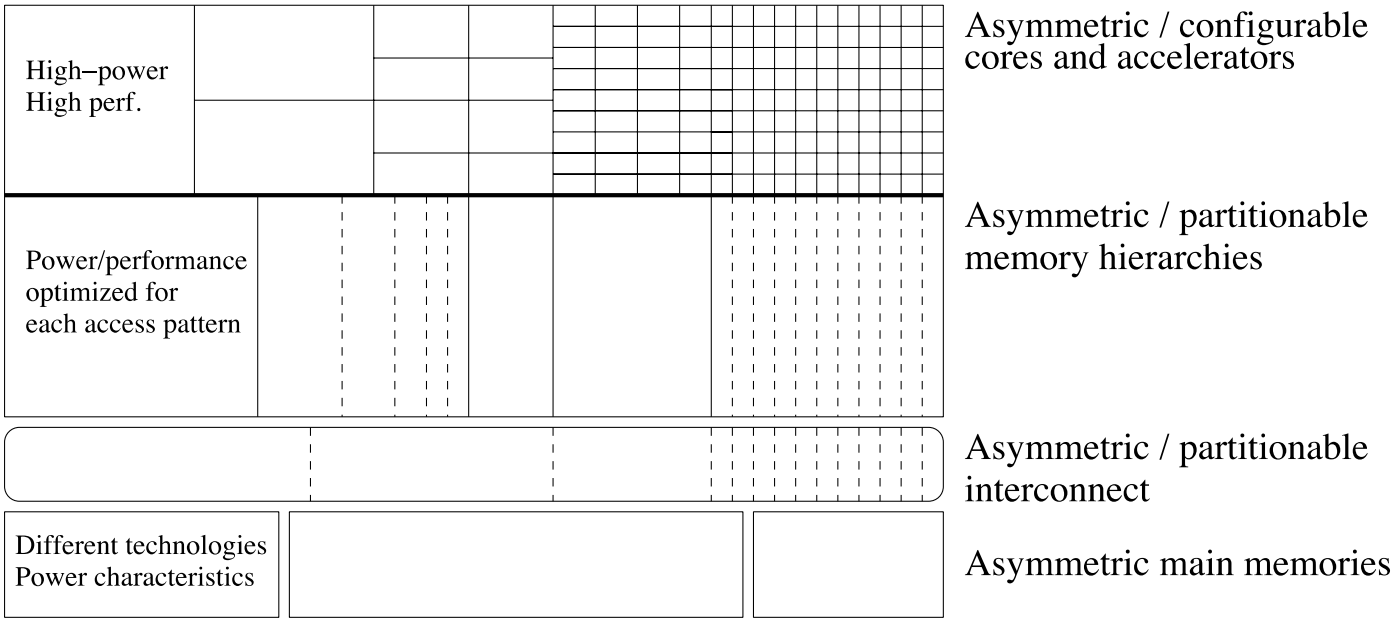
Thought Experiment: Asymmetry Everywhere

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



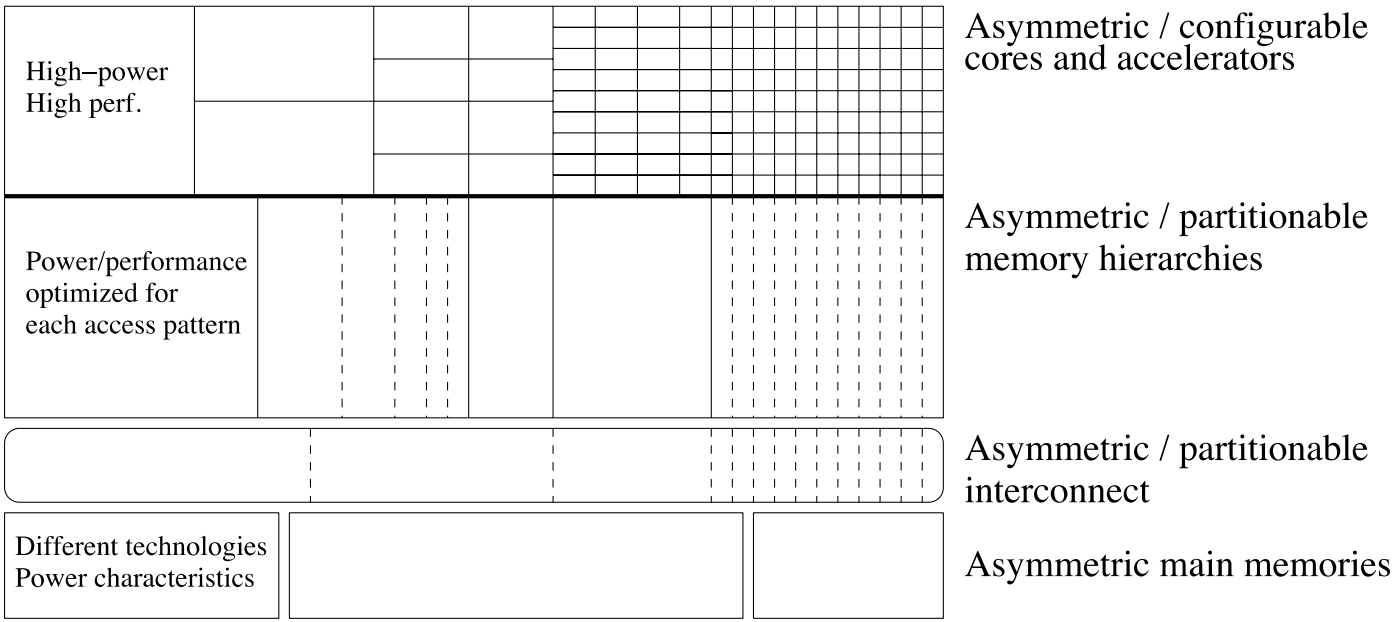
Thought Experiment: Asymmetry Everywhere

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



Thought Experiment: Asymmetry Everywhere

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



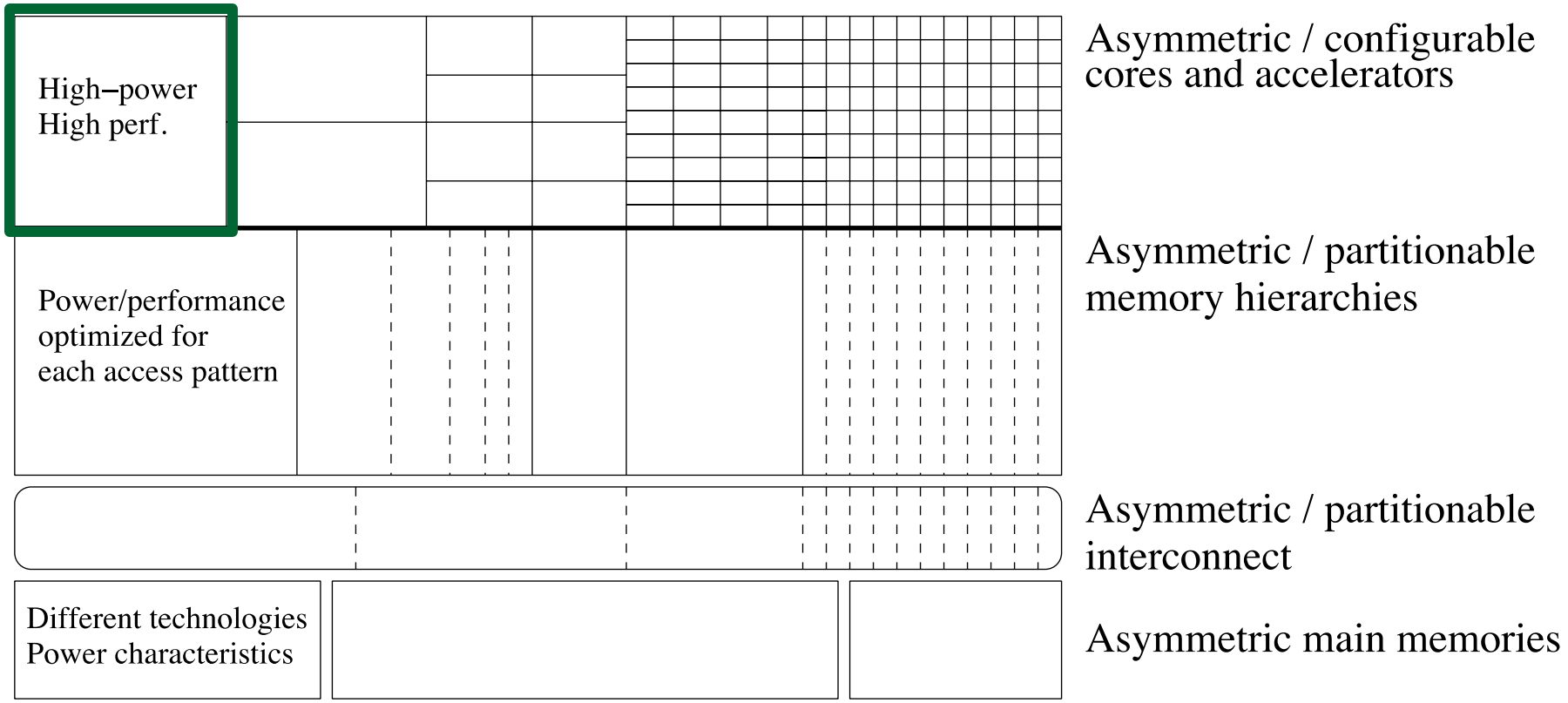
Many Research and Design Questions

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

Talk Outline

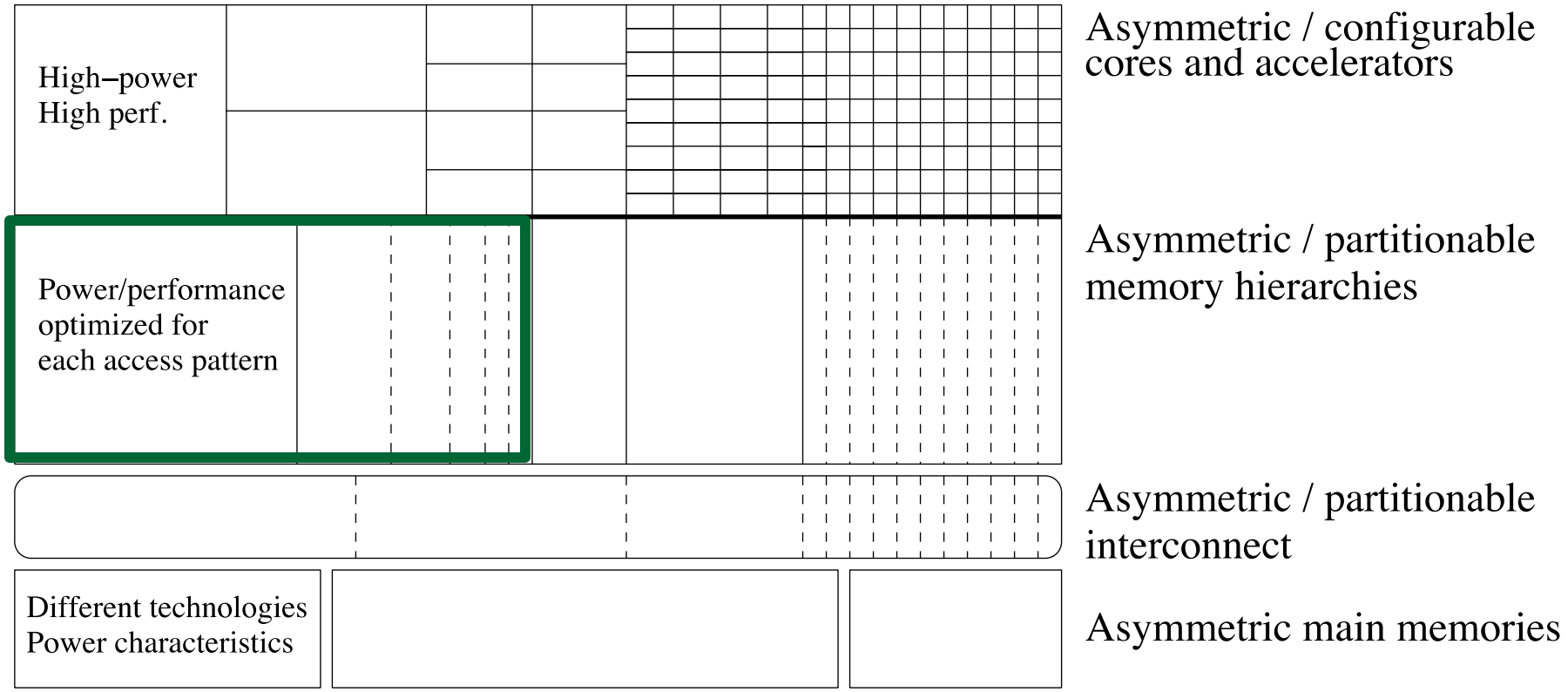
- Problem and Motivation
- How Do We Get There: Examples
- Bottleneck Identification and Scheduling (BIS)
- Handling Resource Contention Bottlenecks
- Conclusions

Exploiting Asymmetry: Simple Examples



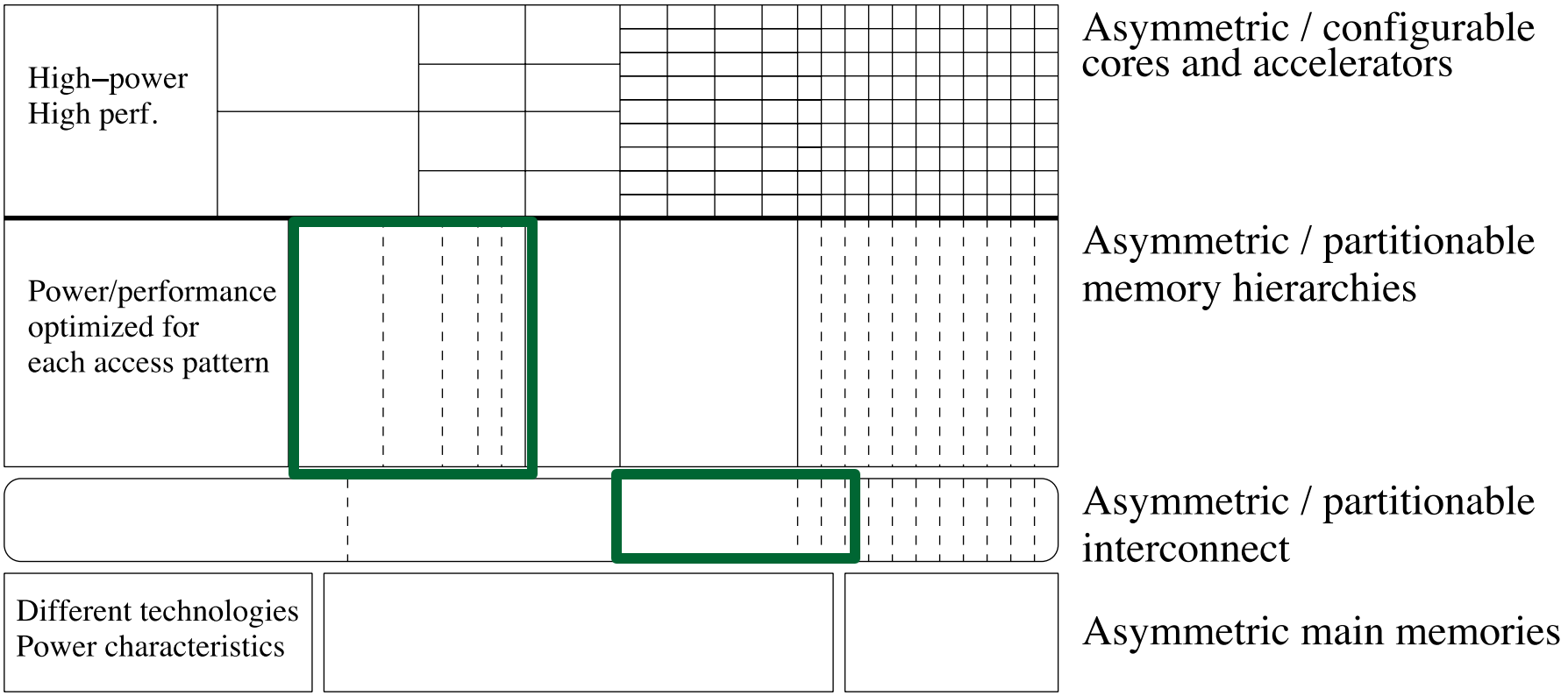
- Execute critical/serial sections on high-power, high-performance cores/resources [Suleman+ ASPLOS'09, ISCA'10, Top Picks'10'11, Joao+ ASPLOS'12]
  - Programmer can write less optimized, but more likely correct programs

Exploiting Asymmetry: Simple Examples



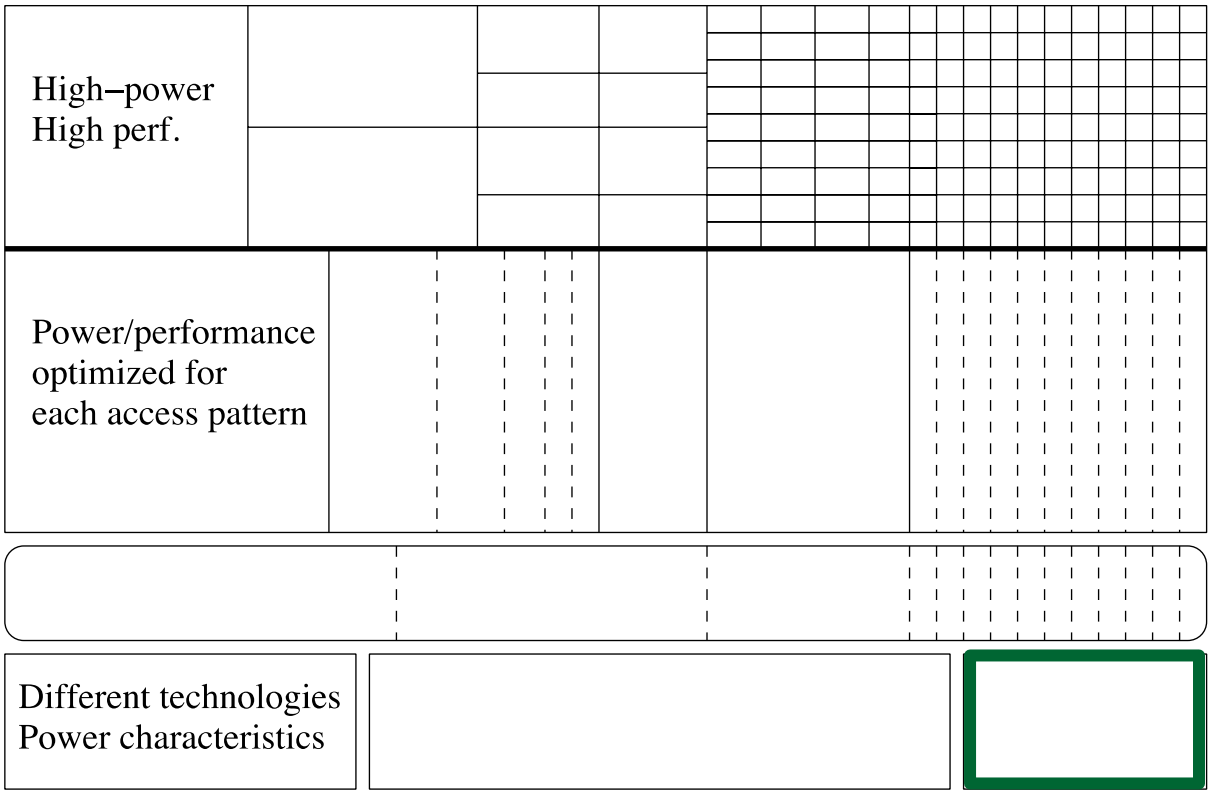
- Execute streaming "memory phases" on streaming-optimized cores and memory hierarchies
  - More efficient and higher performance than general purpose hierarchy

Exploiting Asymmetry: Simple Examples



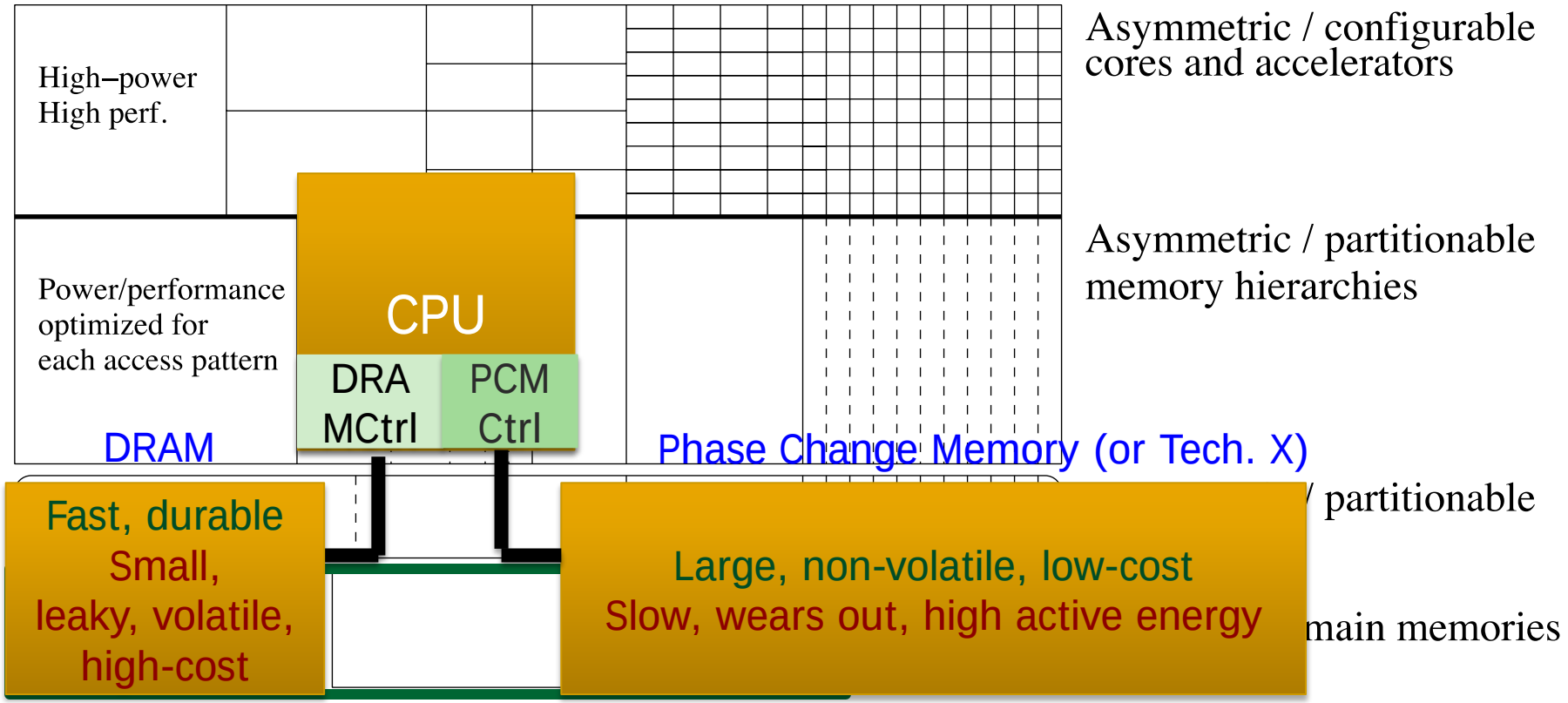
- Partition memory controller and on-chip network bandwidth asymmetrically among threads [Kim+ HPCA 2010, MICRO 2010, Top Picks 2011] [Nychis+ HotNets 2010] [Das+ MICRO 2009, ISCA 2010, Top Picks 2011]
  - Higher performance and energy-efficiency than symmetric/free-for-all

Exploiting Asymmetry: Simple Examples



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

Exploiting Asymmetry: Simple Examples



- Build main memory with different technologies with different characteristics (energy, latency, wear, bandwidth) [Meza+ IEEE CAL'12]
  - Map pages/applications to the best-fit memory resource
  - Higher performance and energy-efficiency than single-level memory

Talk Outline

- Problem and Motivation
- How Do We Get There: Examples
- Bottleneck Identification and Scheduling (BIS)
- Handling Resource Contention Bottlenecks
- Conclusions

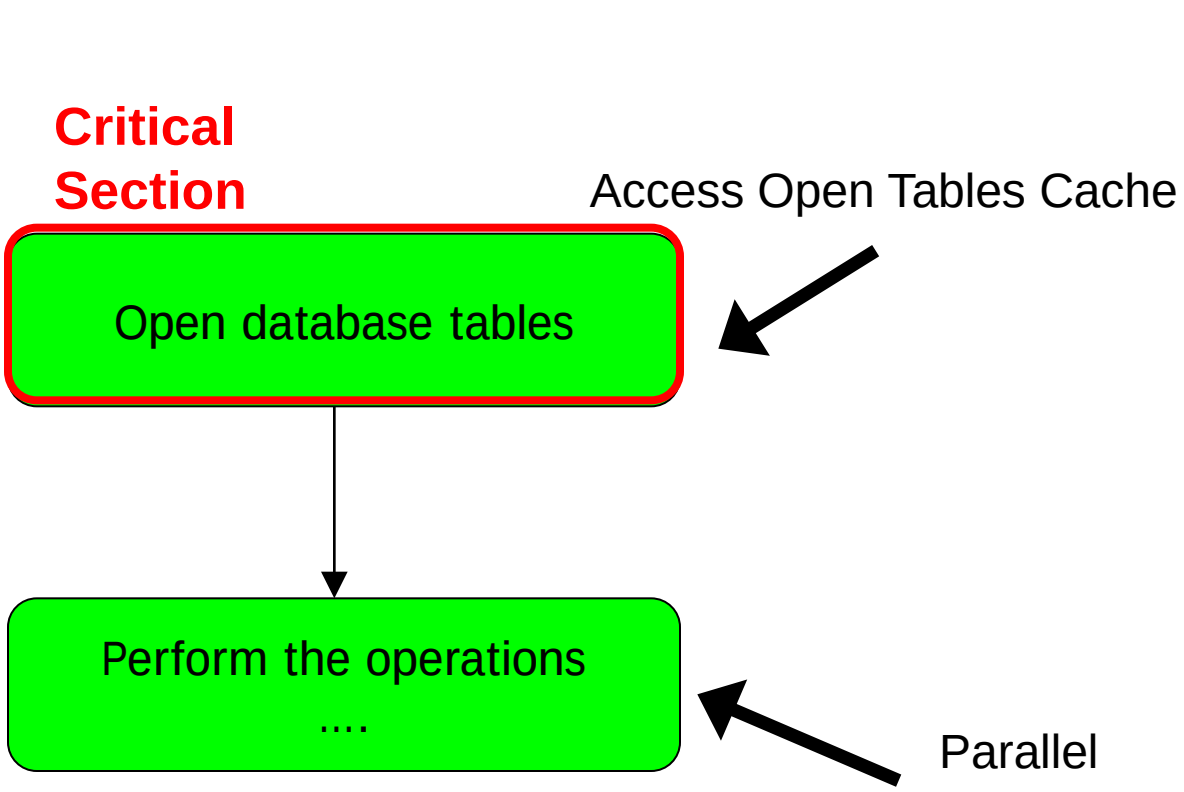
The Problem: Serialized Code Sections

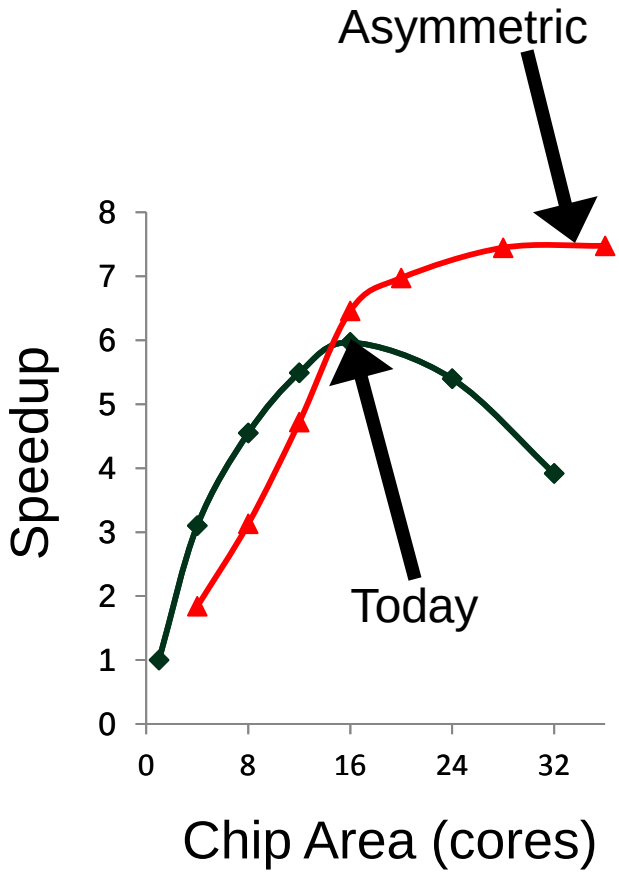
- Many parallel programs cannot be parallelized completely

- 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





BIS Summary

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

Bottlenecks in Multithreaded Applications

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

Examples:

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

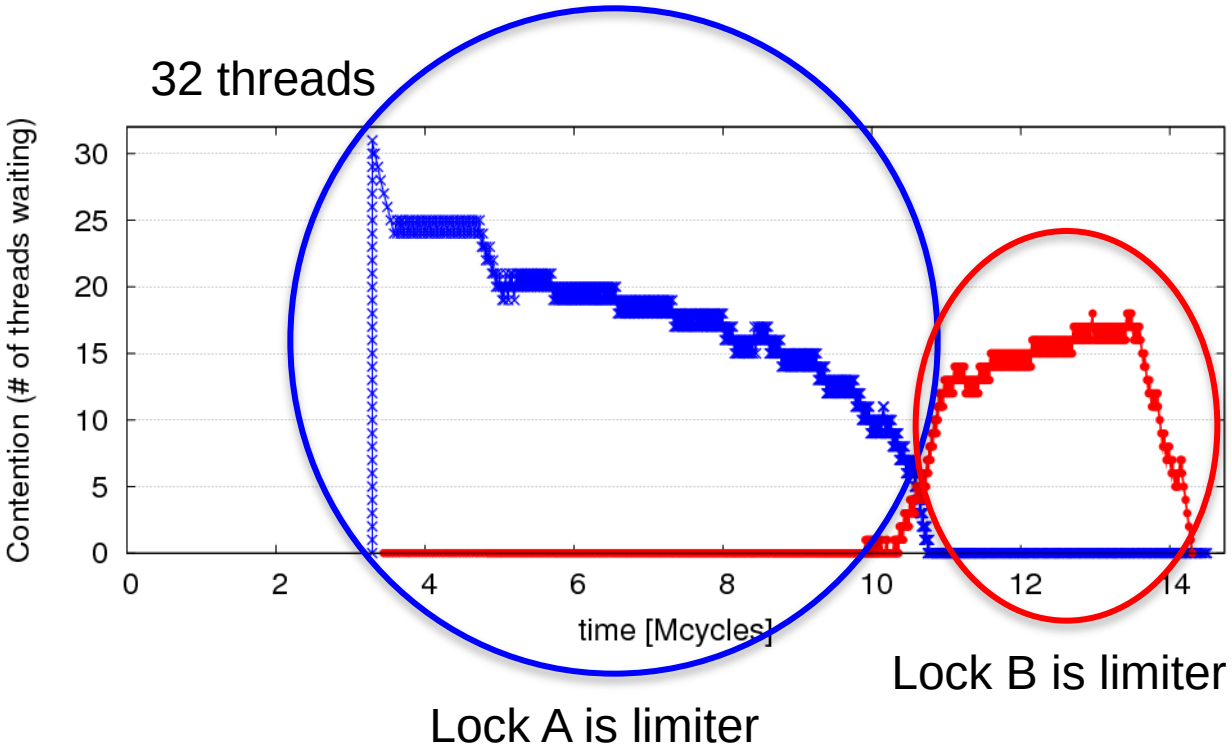
Observation: Limiting Bottlenecks Change Over Time

A=full linked list; B=empty linked list

repeat

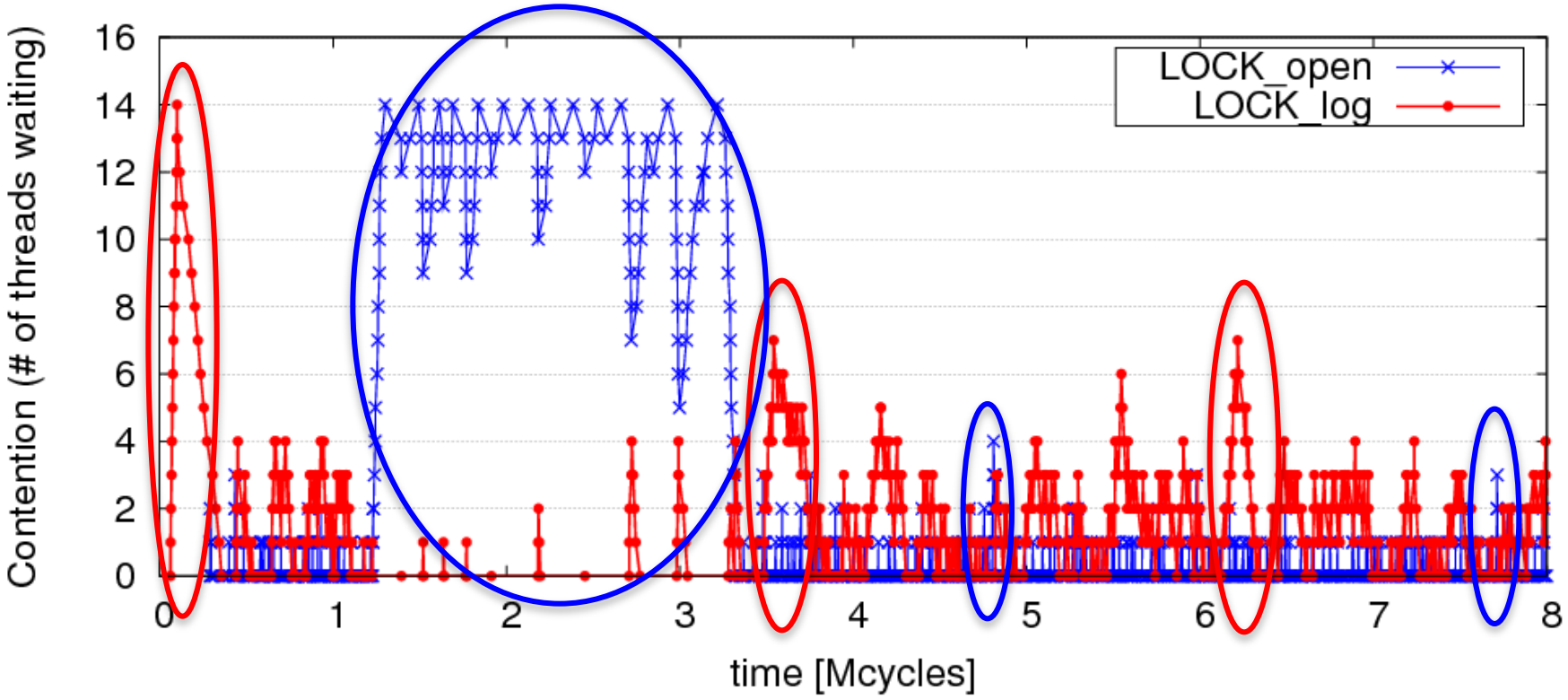
```
Lock A
        Traverse list A
        Remove X from A
Unlock A
Compute on X
Lock B
        Traverse list B
        Insert X into B
Unlock B
```

until A is empty



Limiting Bottlenecks Do Change on Real Applications

MySQL running Sysbench queries, 16 threads



Previous Work on Bottleneck Acceleration

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

No previous work

→ can accelerate all types of bottlenecks or

→ adapts to fine-grain changes in the *importance* of bottlenecks

Our goal:

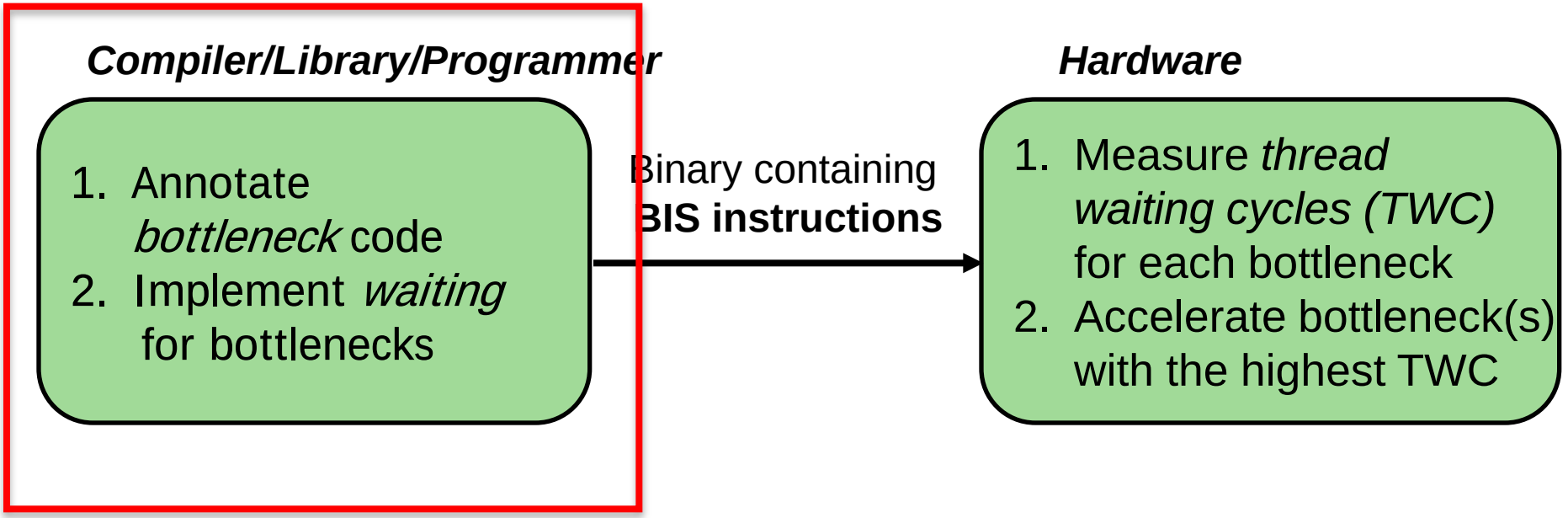
general mechanism to identify and accelerate performance-limiting bottlenecks of any type

Bottleneck Identification and Scheduling (BIS)

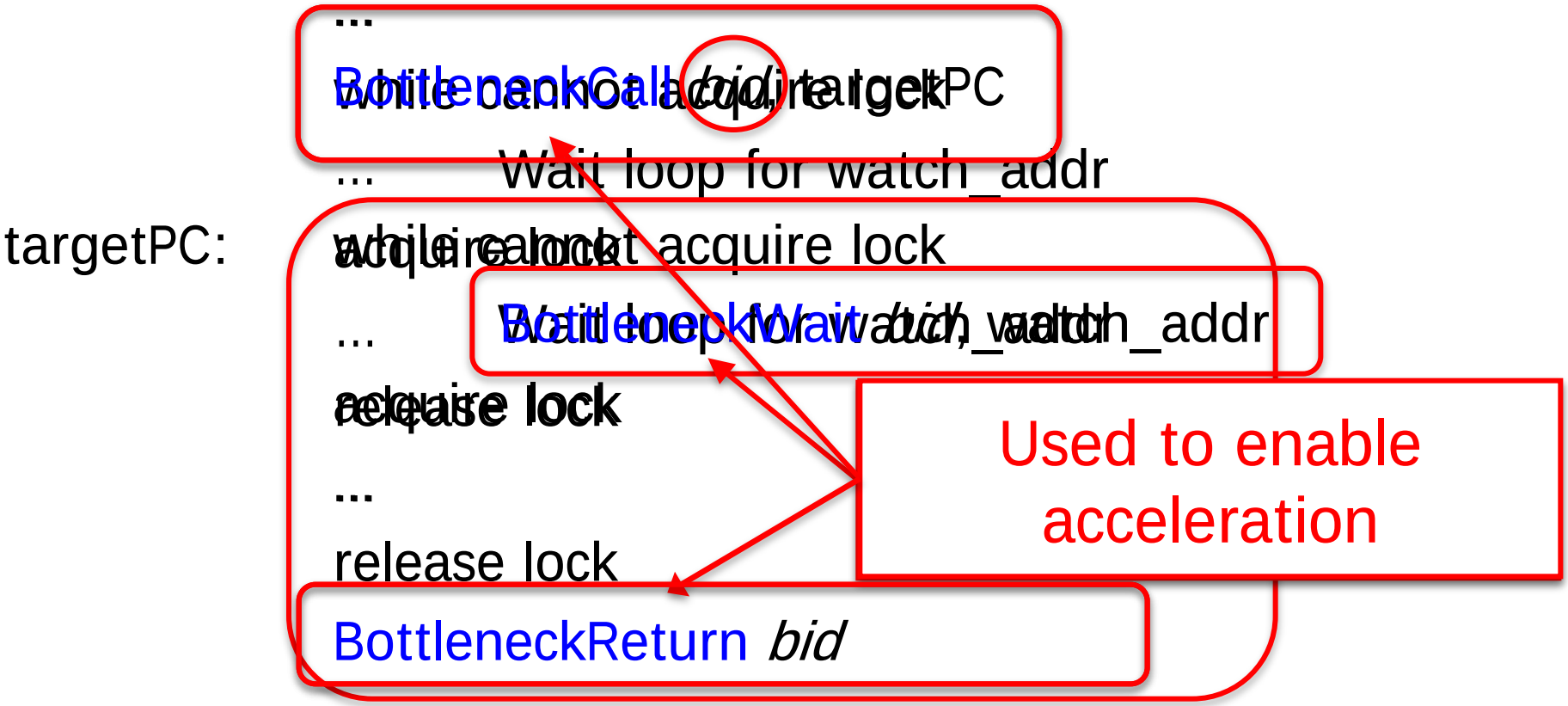
- Key insight:
  - Thread waiting reduces parallelism and is likely to reduce performance
  - Code causing the most thread waiting → likely critical path

- Key idea:
  - Dynamically identify bottlenecks that cause the most thread waiting
  - Accelerate them (using powerful cores in an ACMP)

Bottleneck Identification and Scheduling (BIS)



Critical Sections: Code Modifications



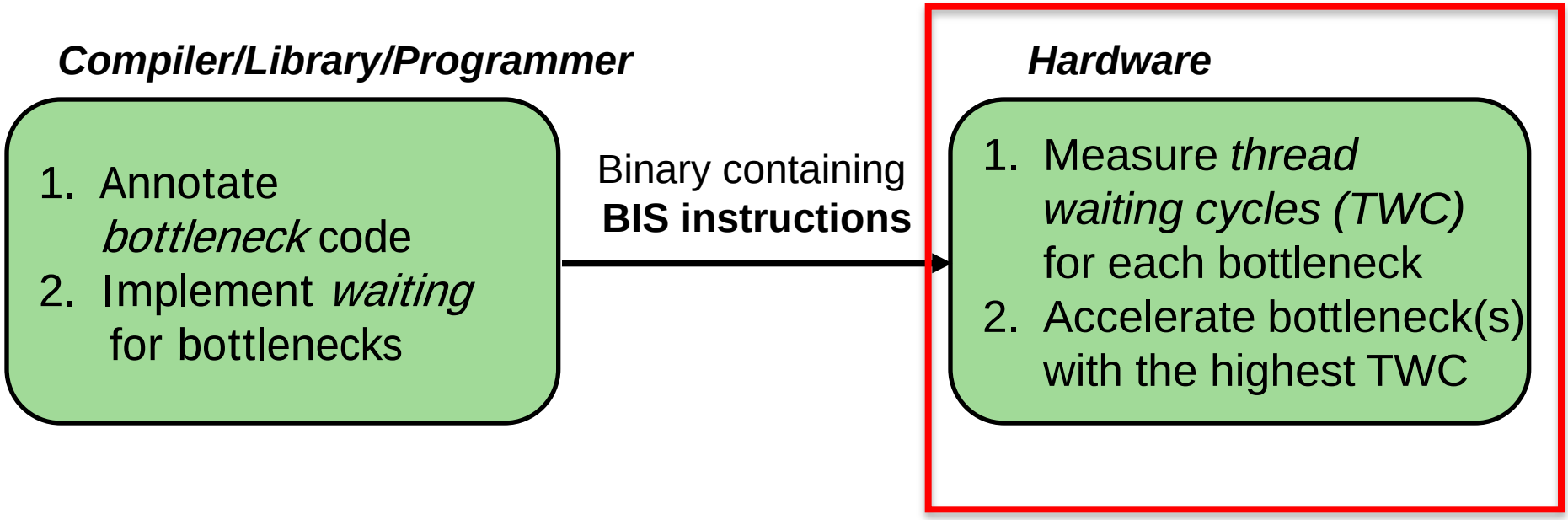
Barriers: Code Modifications

```
                ...
                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

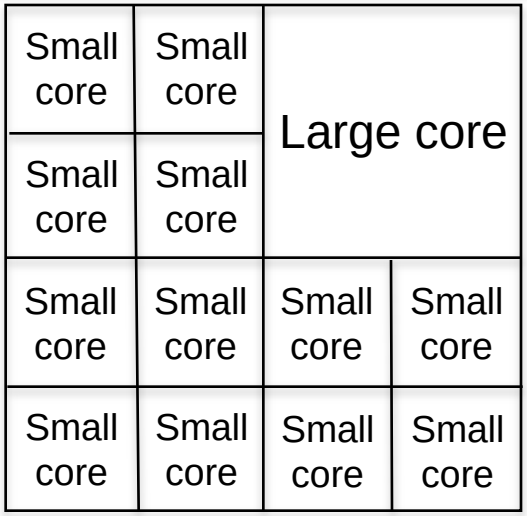
```
                BottleneckCall bid, targetPC
                ...
targetPC:       while not done
                        while empty queue
                                BottleneckWait prev_bid
                        dequeue work
                        do the work ...
                        while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

Bottleneck Identification and Scheduling (BIS)

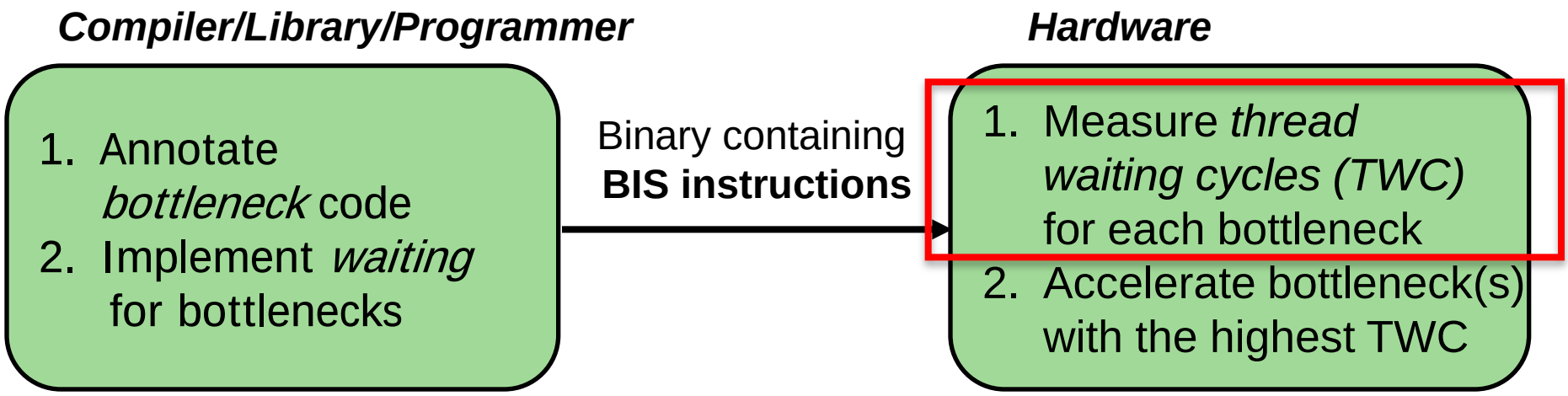


BIS: Hardware Overview

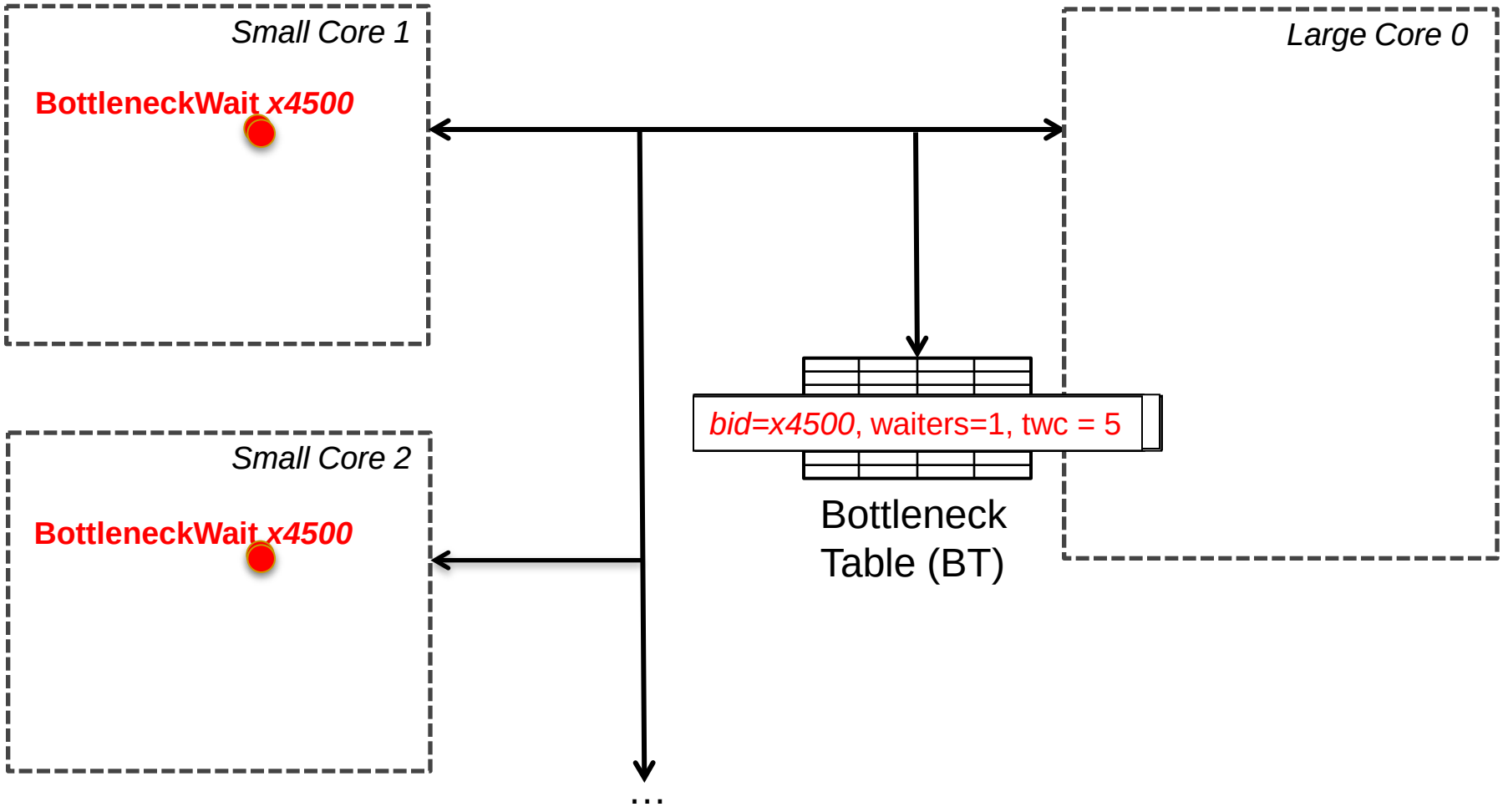
- Performance-limiting bottleneck identification and acceleration are independent tasks
- Acceleration can be accomplished in multiple ways
  - Increasing core frequency/voltage
  - Prioritization in shared resources [Ebrahimi+, MICRO'11]
  - Migration to faster cores in an Asymmetric CMP



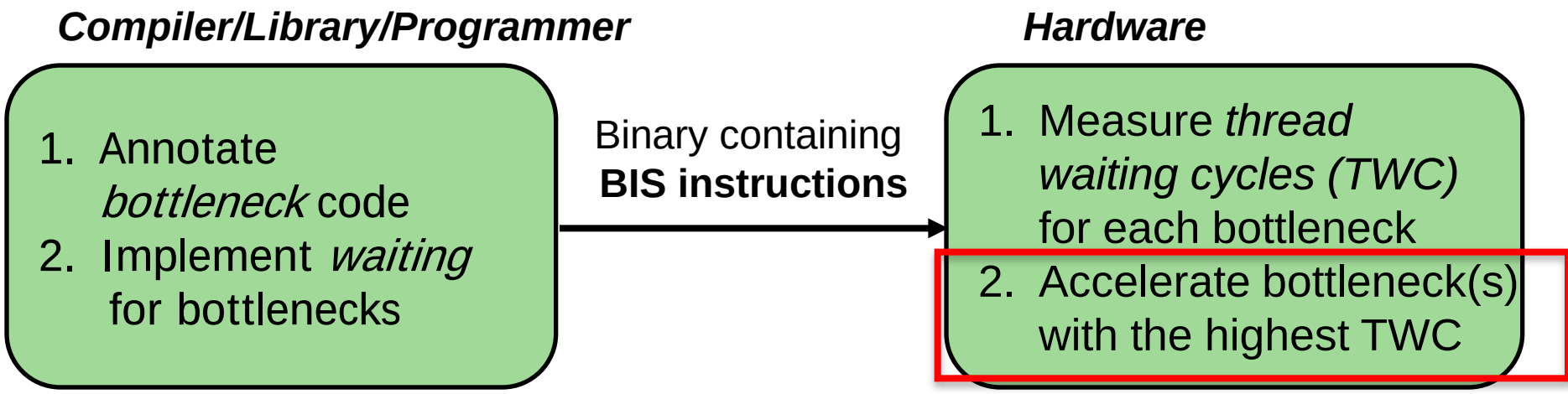
Bottleneck Identification and Scheduling (BIS)



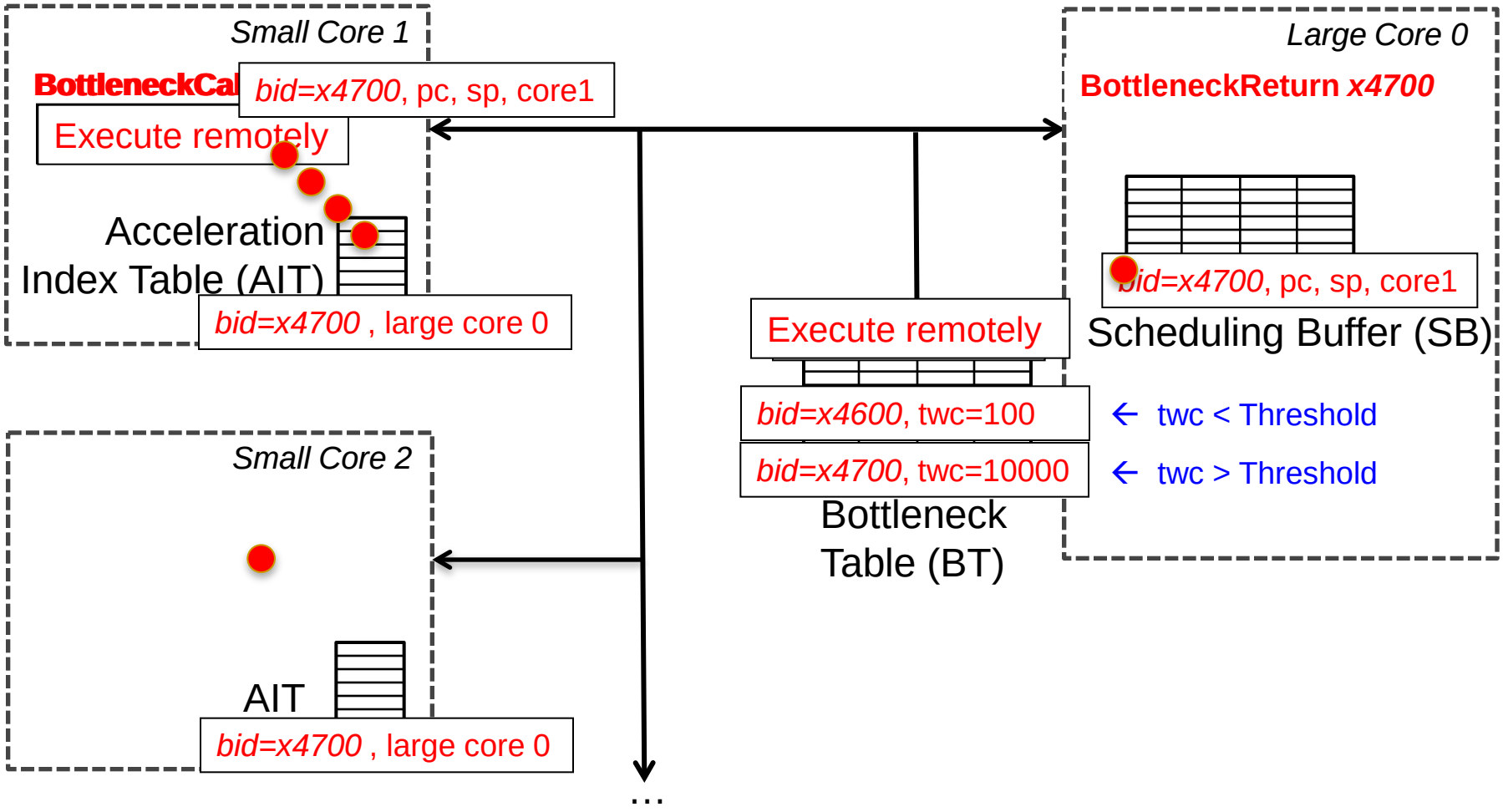
Determining Thread Waiting Cycles for Each Bottleneck



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

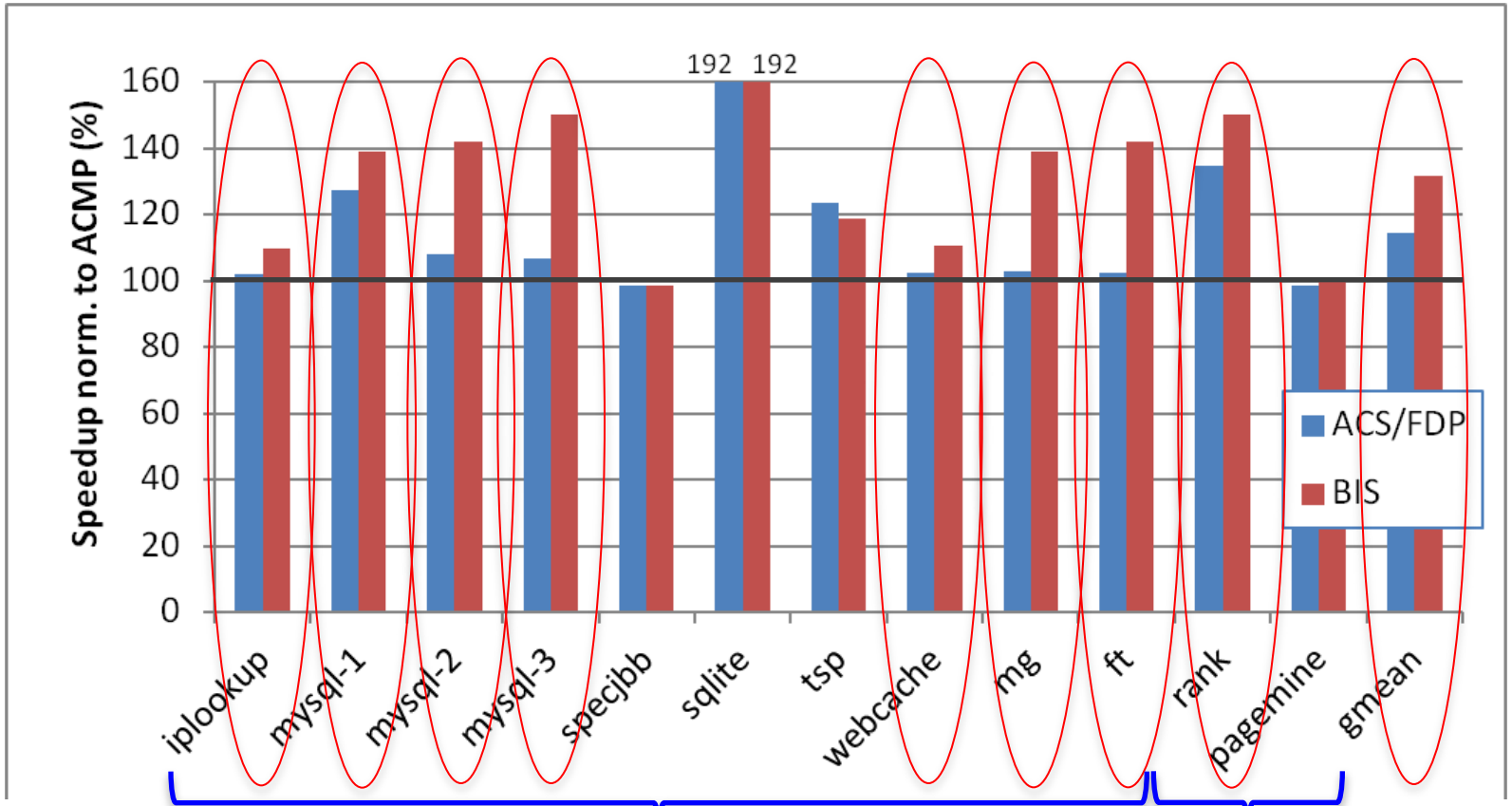
- Workloads: 8 critical section intensive, 2 barrier intensive and 2 pipeline-parallel applications
  - Data mining kernels, scientific, database, web, networking, specjbb
- Cycle-level multi-core x86 simulator
  - 8 to 64 small-core-equivalent area, 0 to 3 large cores, SMT
  - 1 large core is area-equivalent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

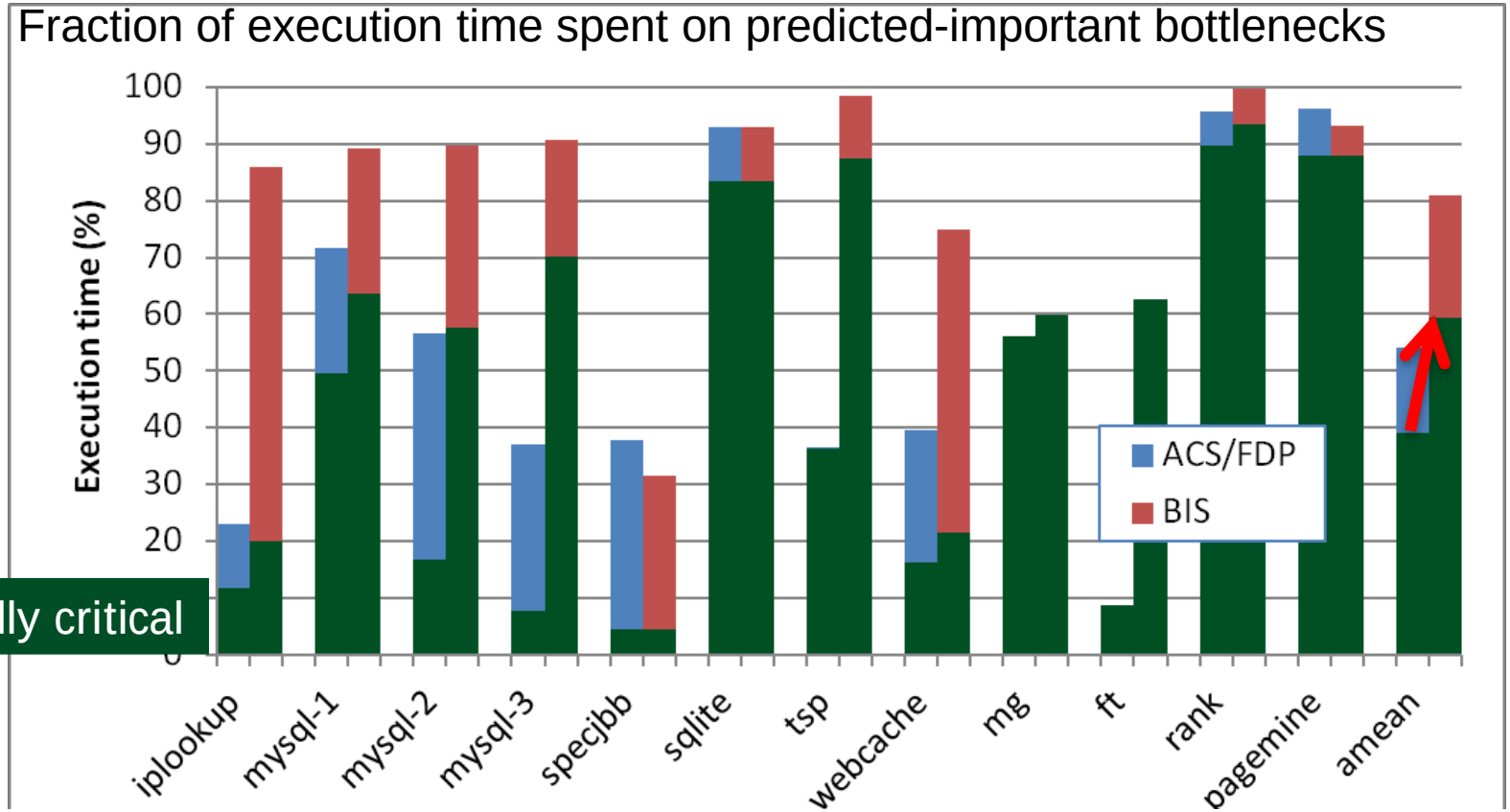
BIS Performance Improvement

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

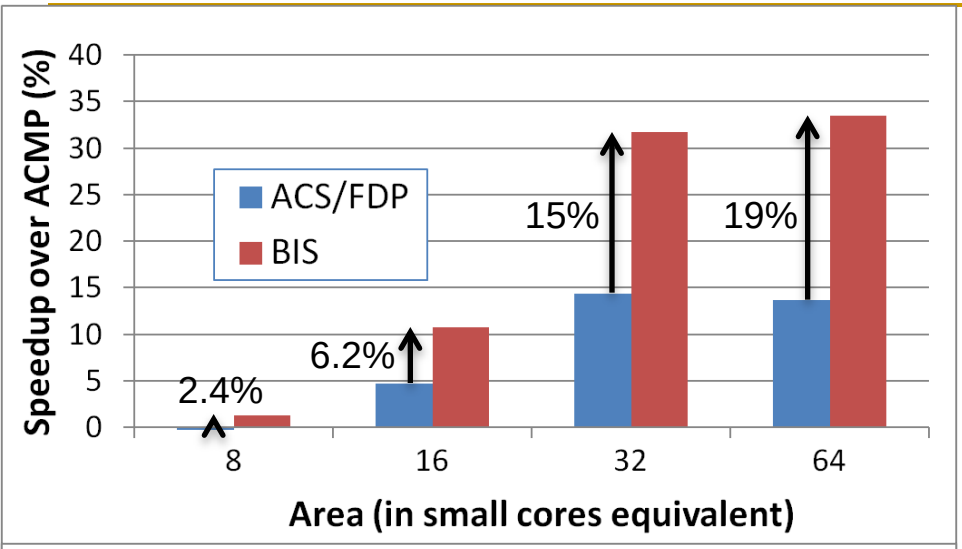
Why Does BIS Work?

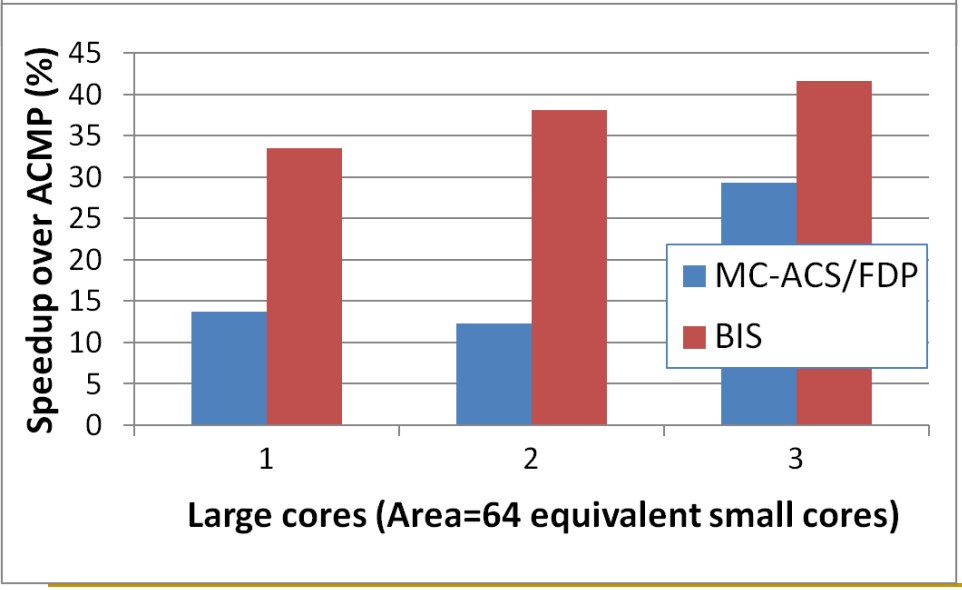


Actually critical

- 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

Talk Outline

- Problem and Motivation
- How Do We Get There: Examples
- Bottleneck Identification and Scheduling (BIS)
- Handling Resource Contention Bottlenecks
- Conclusions

Thread Serialization

- Three fundamental causes

1. Synchronization
2. Load imbalance
3. Resource contention

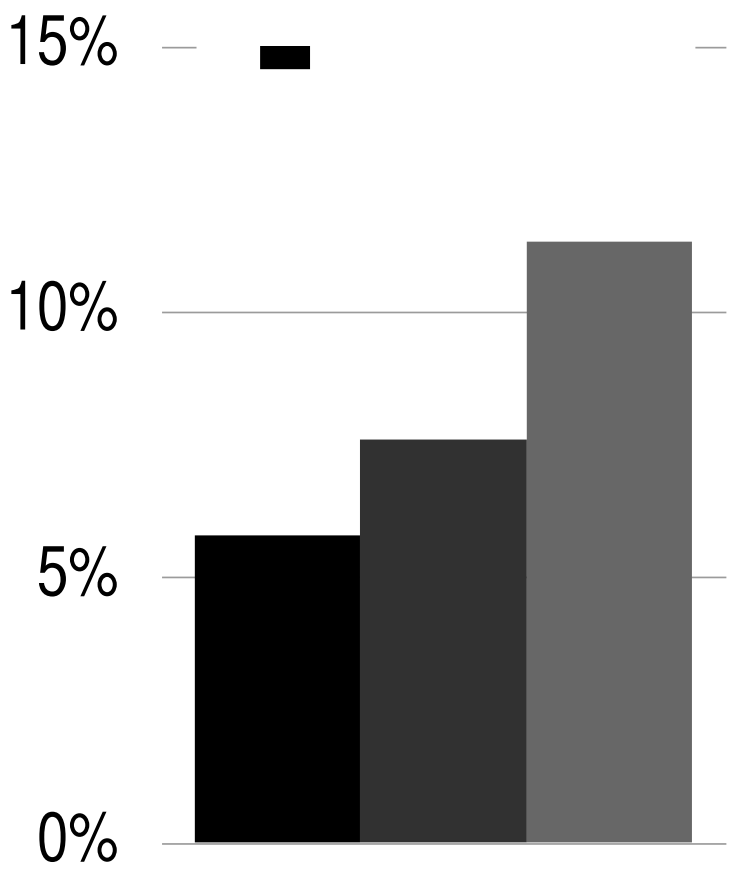
Memory Contention as a Bottleneck

- Problem:
  - Contended memory regions cause serialization of threads
  - Threads accessing such regions can form the critical path
  - Data-intensive workloads (MapReduce, GraphLab, Graph500) can be sped up by 1.5 to 4X by ideally removing contention
- Idea:
  - Identify contended regions dynamically
  - Prioritize caching the data from threads which are slowed down the most by such regions in faster DRAM/eDRAM
- Benefits:
  - Reduces contention, serialization, critical path

Evaluation

- Workloads: MapReduce, GraphLab, Graph500
- Cycle-level x86 platform simulator
  - **CPU**: 8 out-of-order cores, 32KB private L1, 512KB shared L2
  - **Hybrid Memory**: DDR3 1066 MT/s, 32MB DRAM, 8GB PCM
- Mechanisms
  - Baseline: DRAM as a conventional cache to PCM
  - CacheMiss: Prioritize caching data from threads with highest cache miss latency
  - Region: Cache data from most contended memory regions
  - ACTS: Prioritize caching data from threads most slowed down due to memory region contention

Caching Results



Talk Outline

- Problem and Motivation
- How Do We Get There: Examples
- Bottleneck Identification and Scheduling (BIS)
- Handling Resource Contention Bottlenecks
- Conclusions

Summary

- Cloud applications and phases have varying requirements
- Cloud computers evaluated on multiple metrics/constraints: energy, performance, reliability, fairness, ...
- One-size-fits-all design cannot satisfy all requirements and metrics: cannot get the best of all worlds
- Asymmetry in design enables tradeoffs: can approximate the best of all worlds
  - Asymmetry in core types → BIS → Good parallel performance + Good serialized performance
  - Asymmetry in main memory → Critical Group Caching → Good non-contended performance + Good contended performance
- Simple asymmetric designs can be effective and low-cost

Thank You

Onur Mutlu

onur@cmu.edu

http://www.ece.cmu.edu/~omutlu

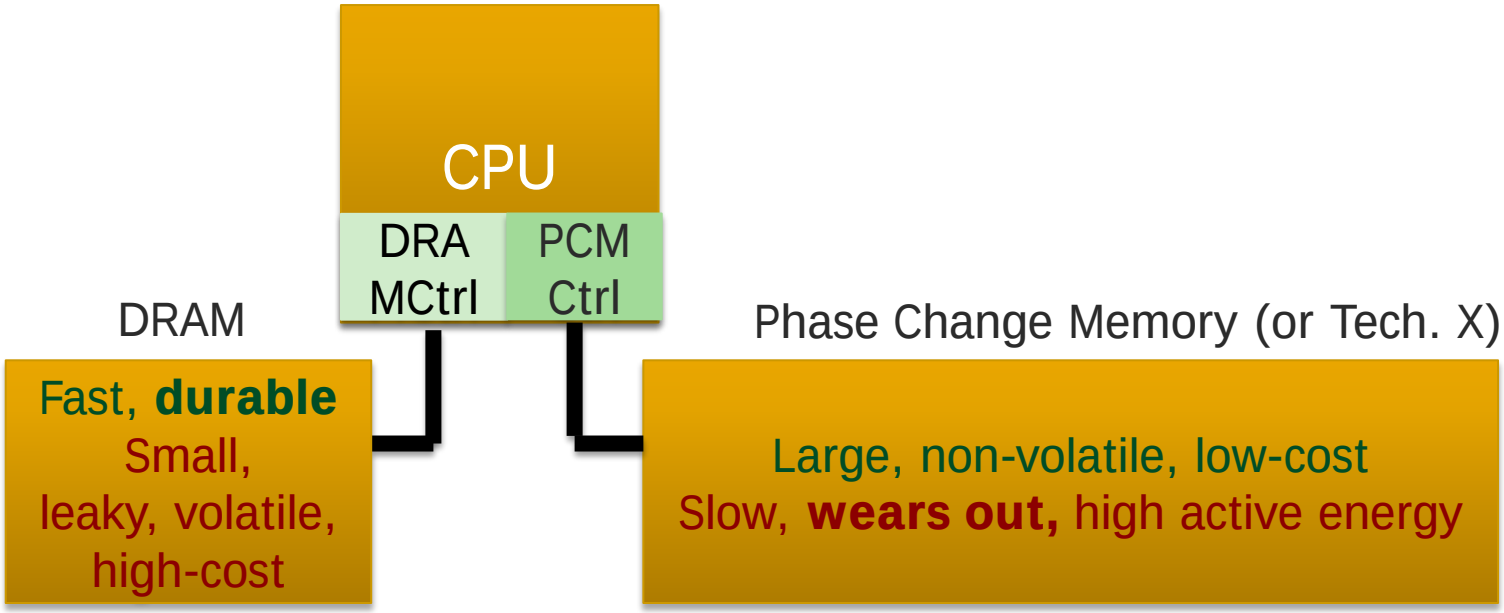
Architecting and Exploiting Asymmetry to Accelerate Bottlenecks in the Cloud

Onur Mutlu
onur@cmu.edu
November 29, 2012
ISTC Retreat, Pittsburgh, PA

SAFARI Carnegie Mellon

Asymmetric Main Memory

Heterogeneous Memory Systems



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

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

One Option: DRAM as a Cache for PCM

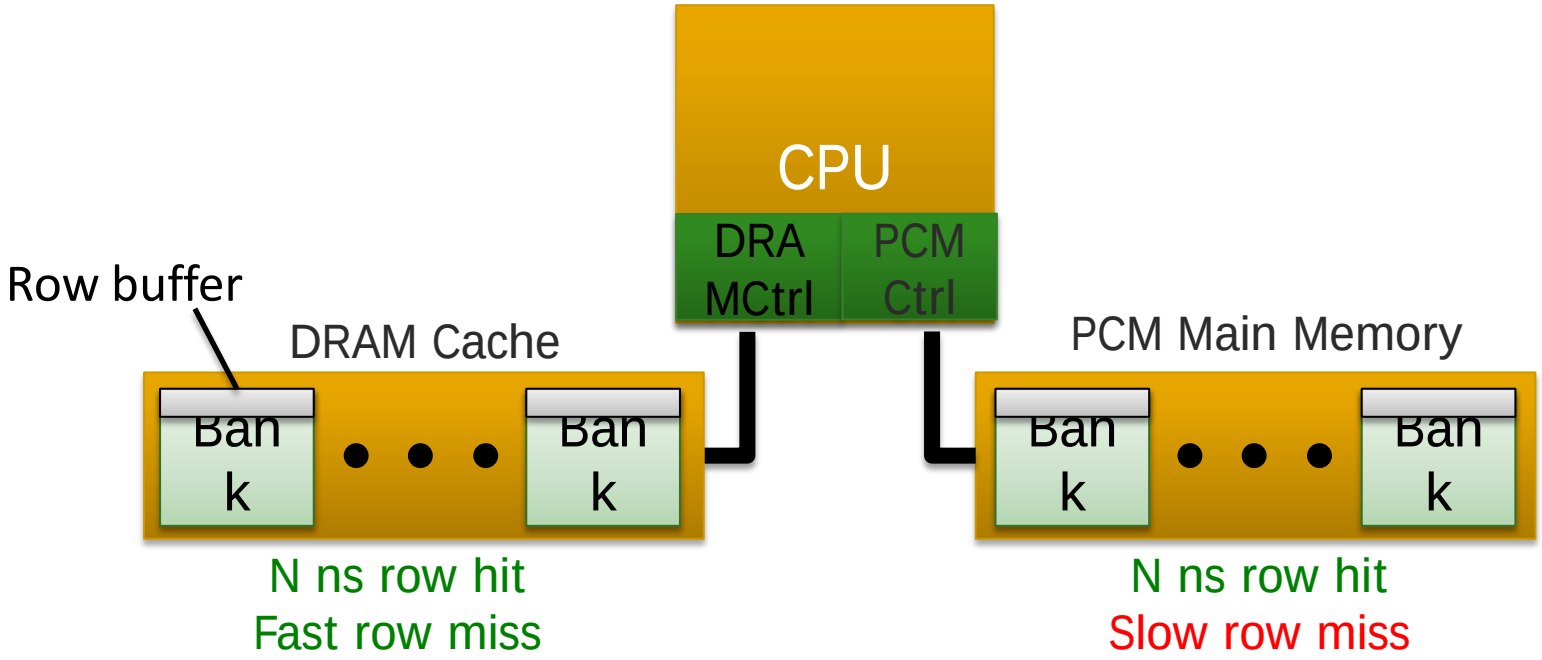
- PCM is main memory; DRAM caches memory rows/blocks
  - Benefits: Reduced latency on DRAM cache hit; write filtering
- Memory controller hardware manages the DRAM cache
  - Benefit: Eliminates system software overhead

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

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

DRAM vs. PCM: An Observation

- Row buffers are the same in DRAM and PCM
- Row buffer hit latency **same** in DRAM and PCM
- Row buffer miss latency **small** in DRAM, **large** in PCM



- Accessing the row buffer in PCM is fast
- What incurs high latency is the PCM array access → avoid this

Row-Locality-Aware Data Placement

- Idea: Cache in DRAM only those rows that
  - Frequently cause row buffer conflicts → because row-conflict latency is smaller in DRAM
  - Are reused many times → to reduce cache pollution and bandwidth waste

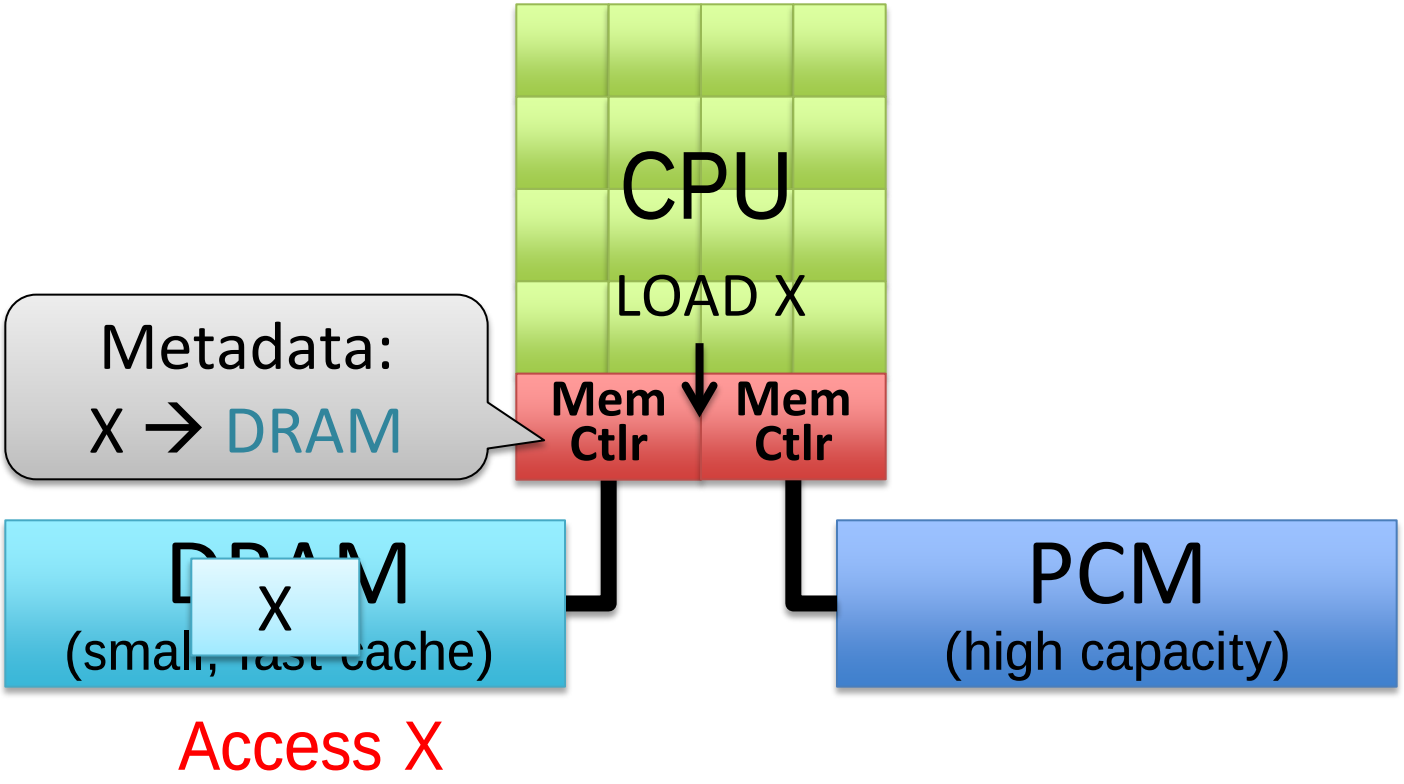
- Simplified rule of thumb:
  - Streaming accesses: Better to place in PCM
  - Other accesses (with some reuse): Better to place in DRAM

- Bridges half of the performance gap between all-DRAM and all-PCM memory on memory-intensive workloads

- Yoon et al. **"Row Buffer Locality Aware Caching Policies for Hybrid Memories,"** ICCD 2012.

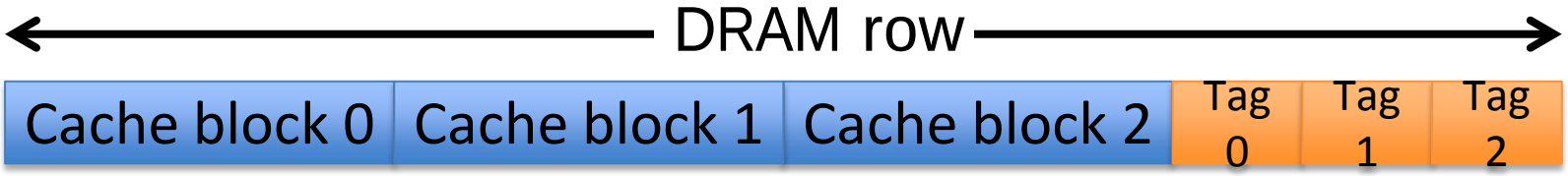
The Problem with Large DRAM Caches

- A large DRAM cache requires a large metadata (tag + block-based information) store
- How do we design an efficient DRAM cache?



Idea 1: Tags in Memory

- Store tags in the same row as data in DRAM
  - Store metadata in same row as their data
  - Data and metadata can be accessed together



- Benefit: No on-chip tag storage overhead
- Downsides:
  - Cache hit determined only after a DRAM access
  - Cache hit requires two DRAM accesses

Idea 2: Cache Tags in SRAM

- Recall Idea 1: Store all metadata in DRAM
  - To reduce metadata storage overhead

- Idea 2: Cache in on-chip SRAM frequently-accessed metadata
  - Cache only a small amount to keep SRAM size small

Idea 3: Dynamic Data Transfer Granularity

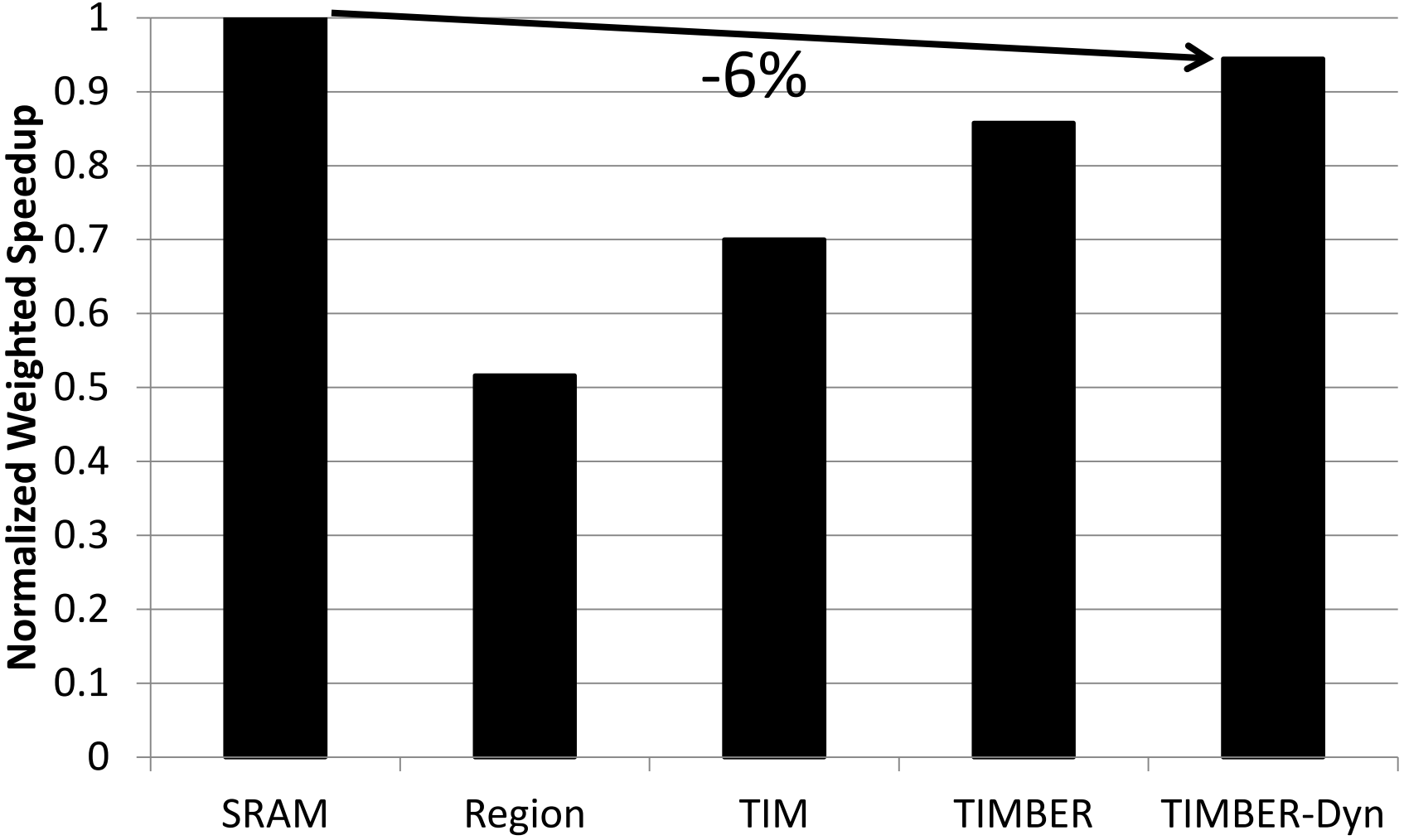
- Some applications benefit from caching more data
  - They have good spatial locality
- Others do not
  - Large granularity wastes bandwidth and reduces cache utilization

- Idea 3: Simple dynamic caching granularity policy
  - Cost-benefit analysis to determine best DRAM cache block size
  - Group main memory into sets of rows
  - Some row sets follow a fixed caching granularity
  - The rest of main memory follows the best granularity
    - Cost–benefit analysis: access latency versus number of cachings
    - Performed every quantum

Methodology

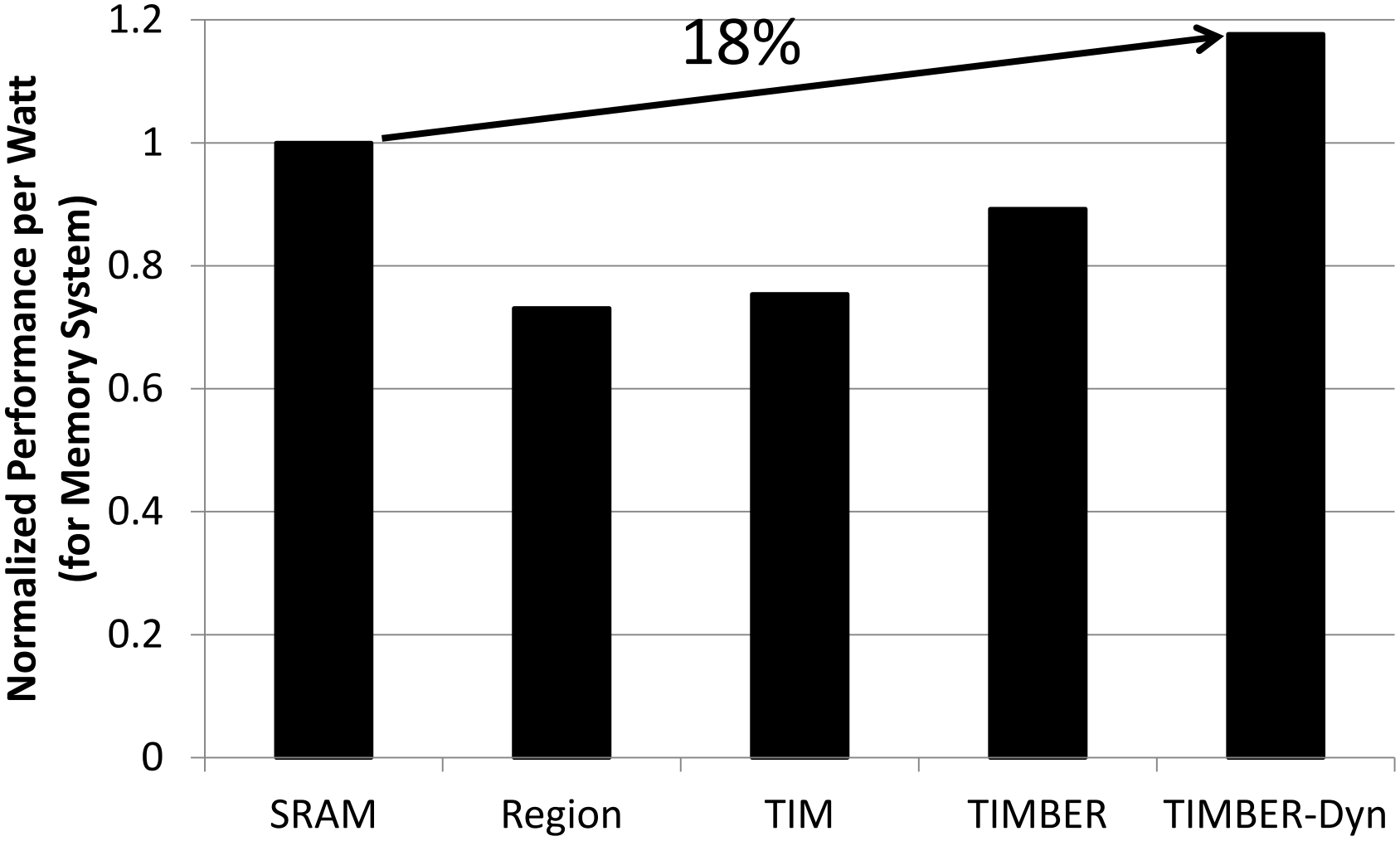
- System: 8 out-of-order cores at 4 GHz
- Memory: 512 MB direct-mapped DRAM, 8 GB PCM
  - 128B caching granularity
  - DRAM row hit (miss): 200 cycles (400 cycles)
  - PCM row hit (clean / dirty miss): 200 cycles (640 / 1840 cycles)
- Evaluated metadata storage techniques
  - All SRAM system (8MB of SRAM)
  - Region metadata storage
  - TIM metadata storage (same row as data)
  - TIMBER, 64-entry direct-mapped (8KB of SRAM)

TIMBER Performance



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

TIMBER Energy Efficiency

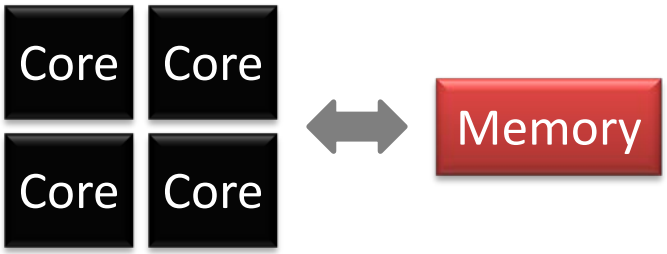


Meza, Chang, Yoon, Mutlu, Ranganathan, "Enabling Efficient and Scalable Hybrid Memories," IEEE Comp. Arch. Letters, 2012.

Asymmetric Memory Controllers

Motivation

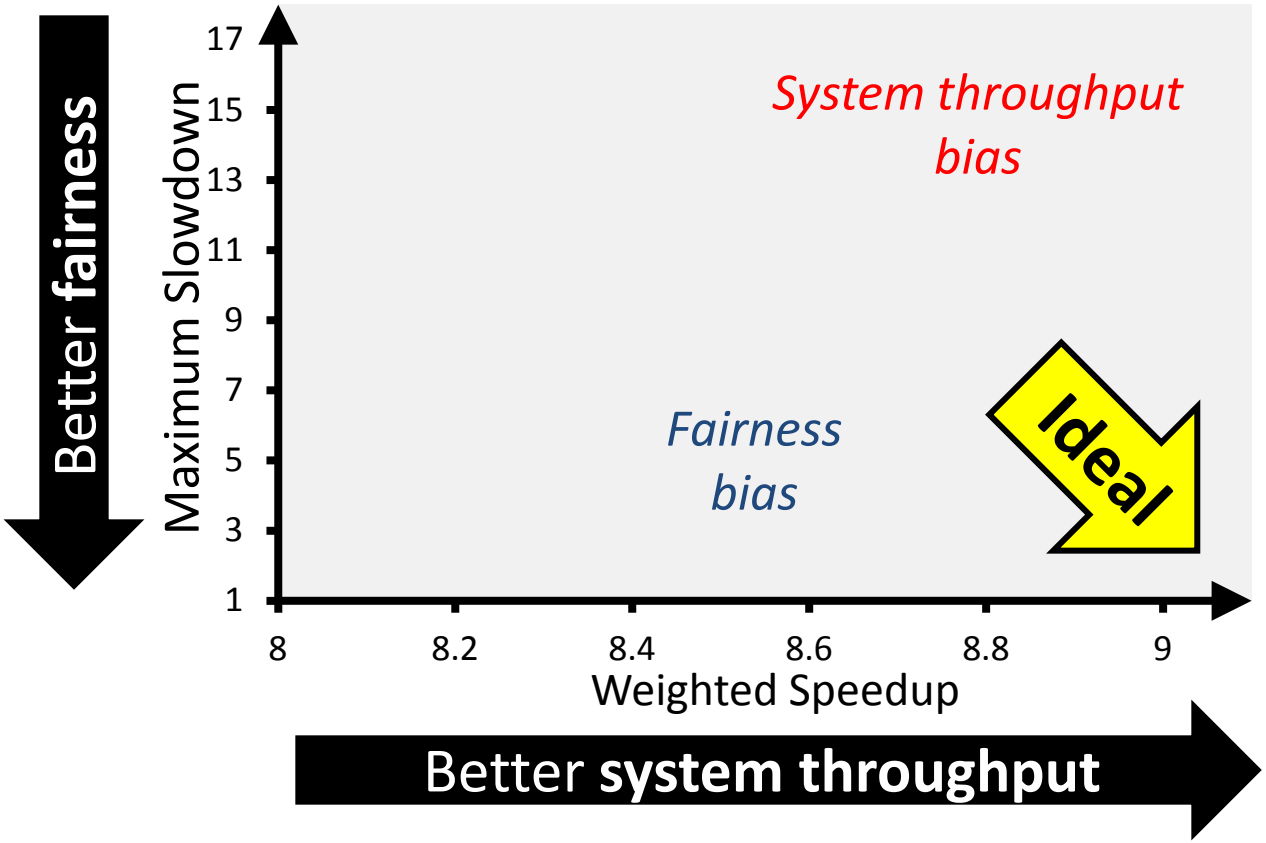
- Memory is a shared resource



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

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

Previous Scheduling Algorithms are Biased



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

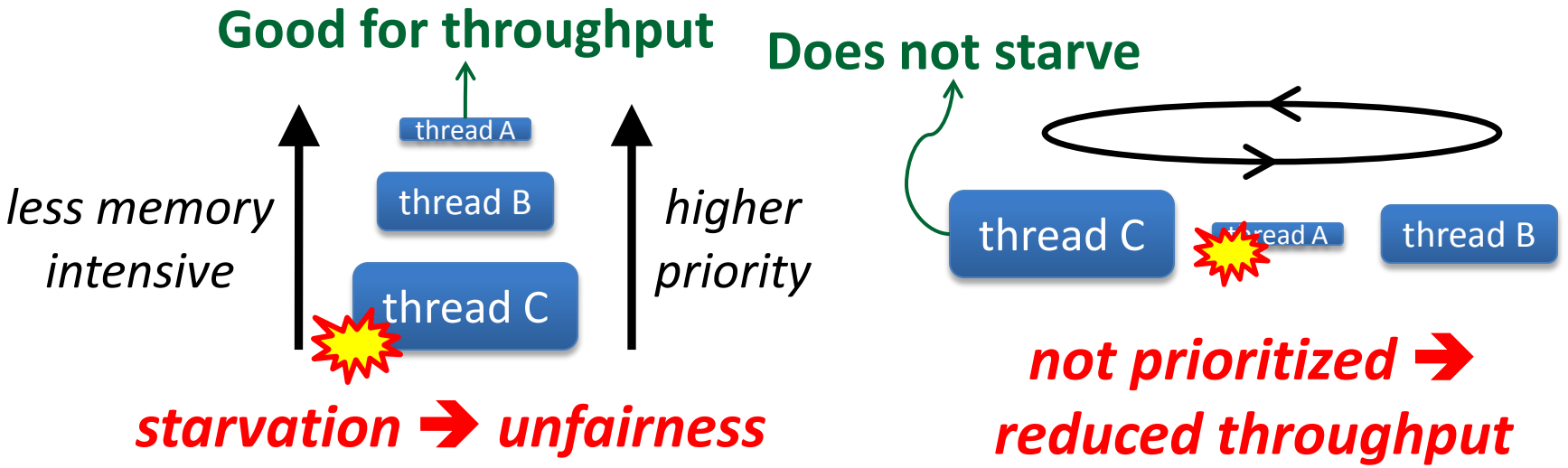
Why do Previous Algorithms Fail?

Throughput biased *approach*

Prioritize less memory-intensive threads

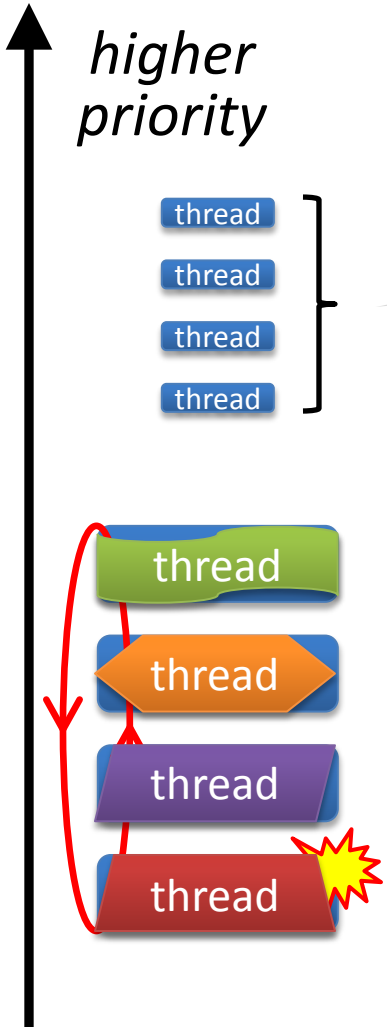
Fairness biased *approach*

Take turns accessing memory



Single policy for all threads is insufficient

Insight: Achieving Best of Both Worlds



For Throughput

Prioritize memory-non-intensive threads

For Fairness

Unfairness caused by memory-intensive being prioritized over each other

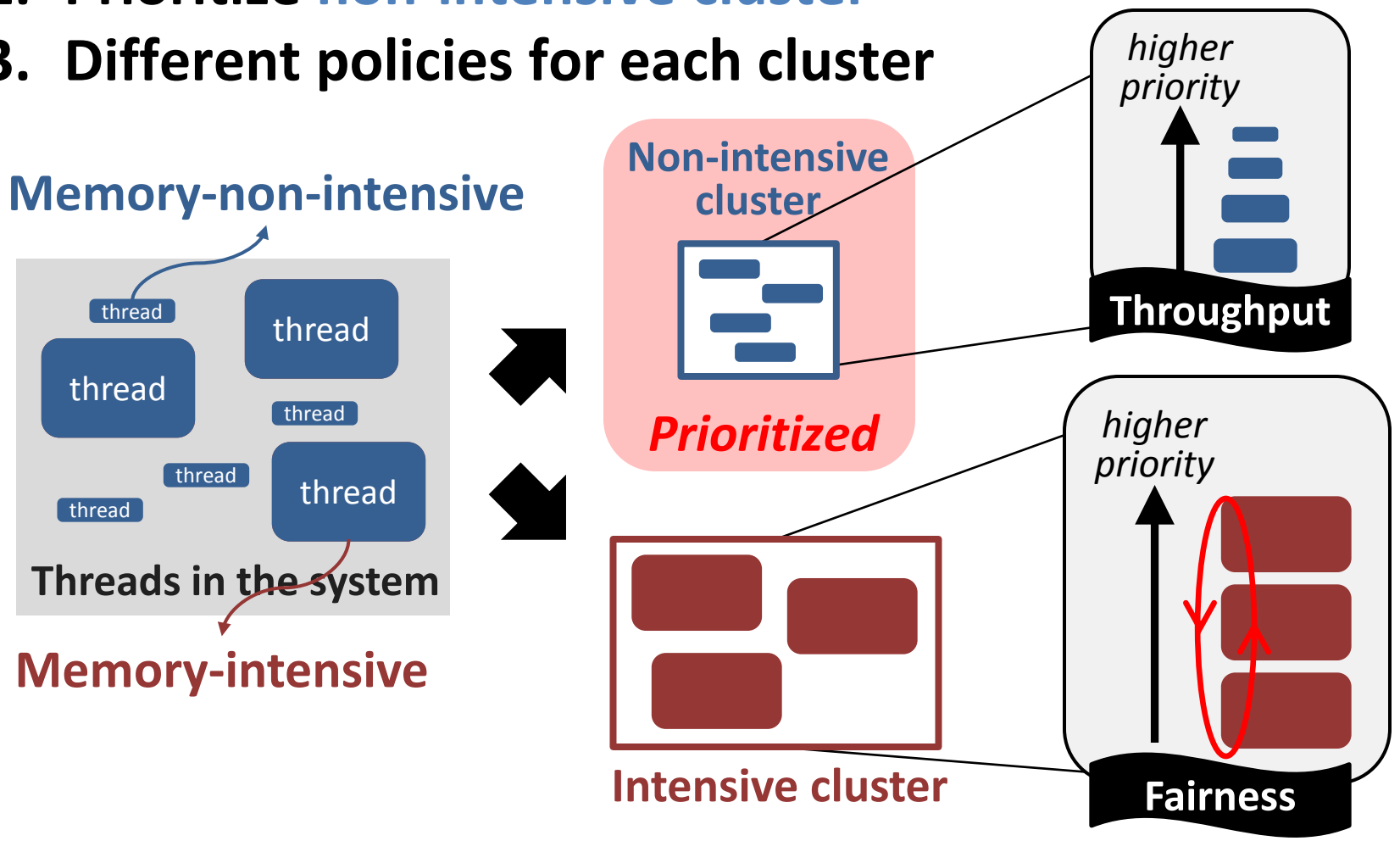
- Shuffle threads

Memory-intensive threads have different vulnerability to interference

- Shuffle asymmetrically

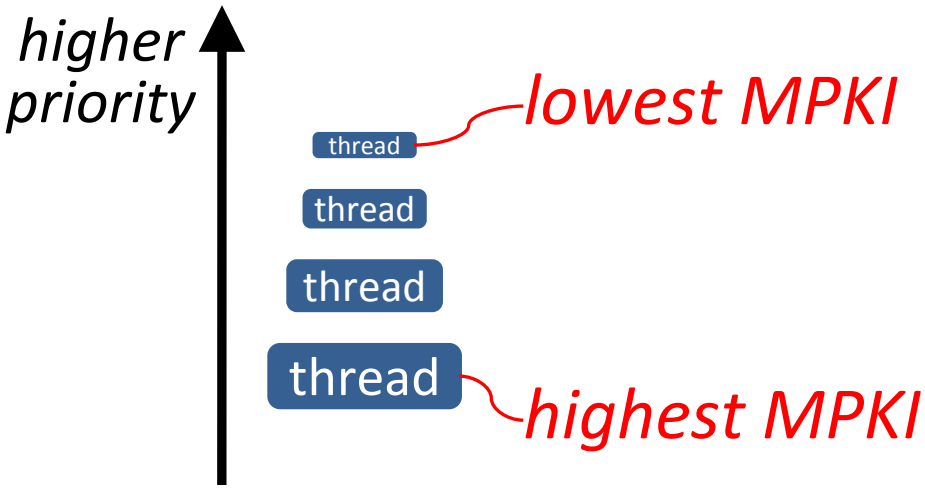
Overview: Thread Cluster Memory Scheduling

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



Non-Intensive Cluster

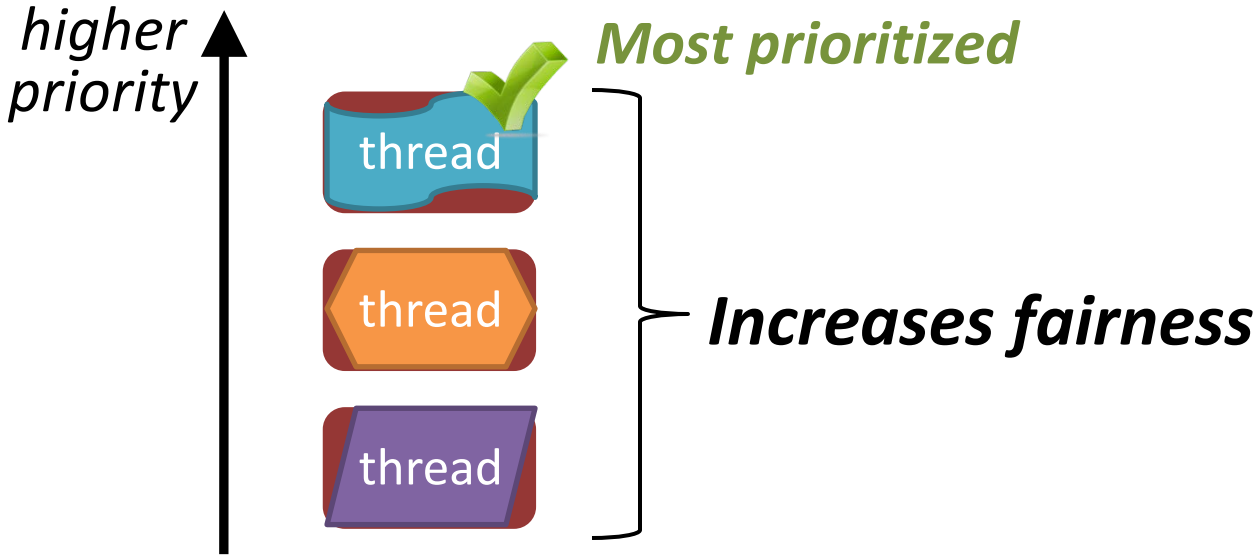
Prioritize threads according to MPKI



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

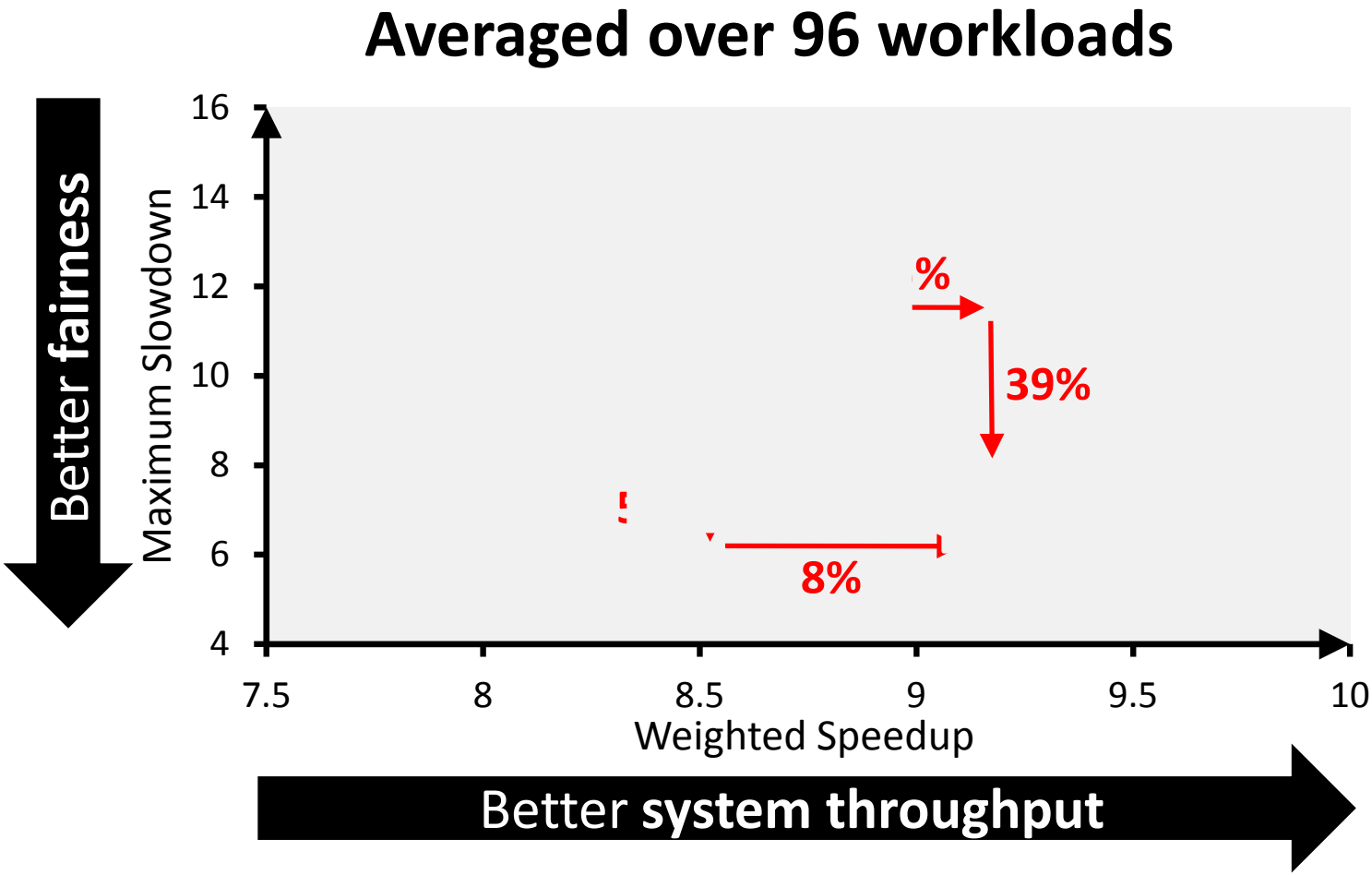
Intensive Cluster

Periodically shuffle the priority of threads



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

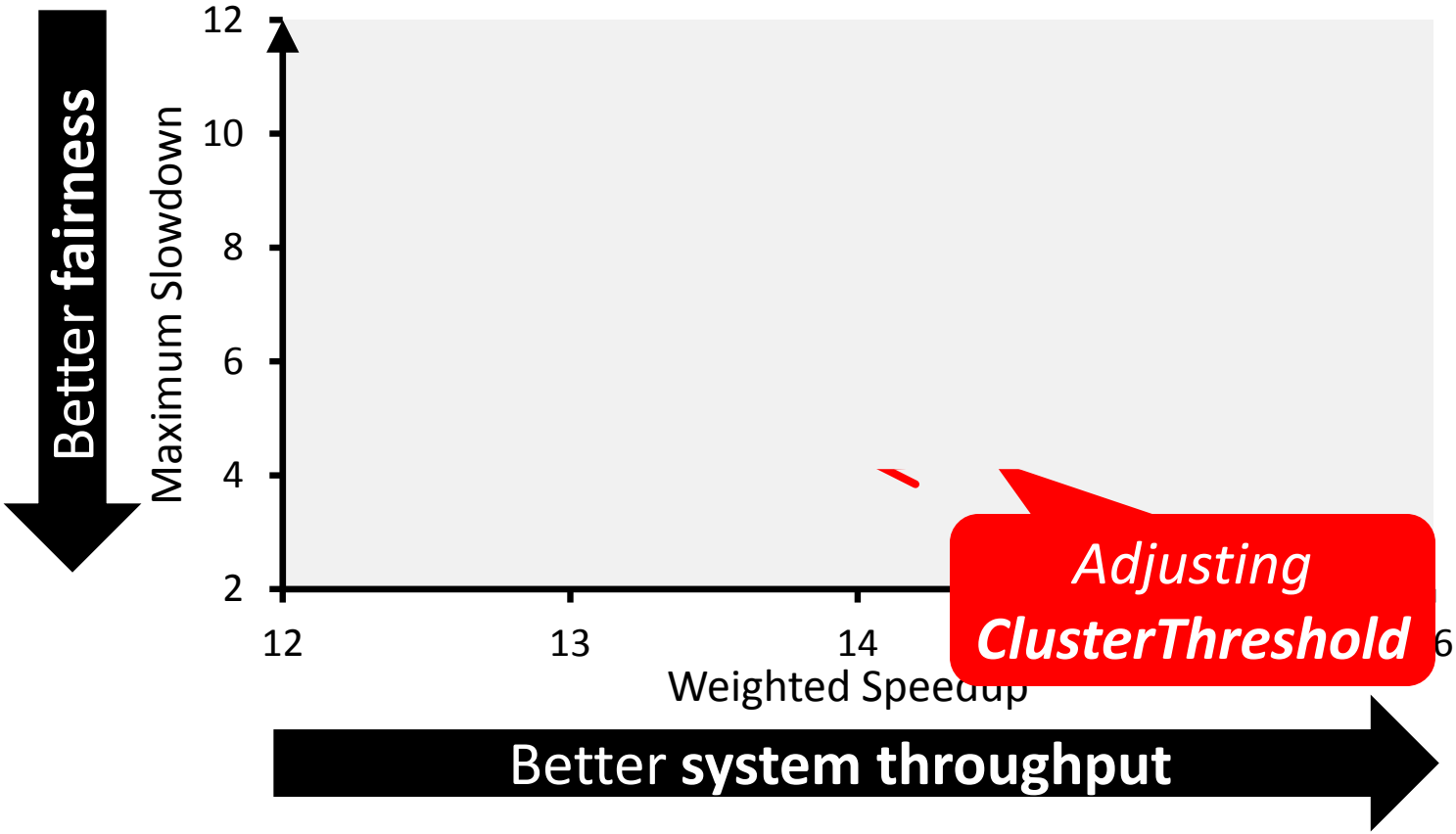
Results: Fairness vs. Throughput



TCM provides best fairness and system throughput

Results: Fairness-Throughput Tradeoff

When configuration parameter is varied…



TCM allows robust fairness-throughput tradeoff

TCM Summary

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

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

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

More Details on TCM

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

Memory Control in CPU-GPU Systems

- **Observation:** Heterogeneous CPU-GPU systems require memory schedulers with large request buffers
- **Problem:** Existing monolithic application-aware memory scheduler designs are hard to scale to large request buffer sizes
- **Solution:** Staged Memory Scheduling (SMS)

  decomposes the memory controller into three simple stages:

  1) Batch formation: maintains row buffer locality

  2) Batch scheduler: reduces interference between applications

  3) DRAM command scheduler: issues requests to DRAM

- Compared to state-of-the-art memory schedulers:
  - SMS is significantly simpler and more scalable
  - SMS provides higher performance and fairness

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

Asymmetric Memory QoS in a Parallel Application

- Threads in a multithreaded application are inter-dependent
- Some threads can be on the critical path of execution due to synchronization; some threads are not
- How do we schedule requests of inter-dependent threads to maximize multithreaded application performance?

- Idea: Estimate limiter threads likely to be on the critical path and prioritize their requests; shuffle priorities of non-limiter threads to reduce memory interference among them [Ebrahimi+, MICRO'11]

- Hardware/software cooperative limiter thread estimation:
  - Thread executing the most contended critical section
  - Thread that is falling behind the most in a *parallel for* loop

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