Multi-Core Architectures and Shared Resource Management

Prof. Onur Mutlu
http://www.ece.cmu.edu/~omutlu
onur@cmu.edu
Bogazici University
June 6, 2013

SAFARI Carnegie Mellon

Agenda for Today

- Course logistics, info, requirements
  - Who am I?
  - What will you learn?
  - How can you get the best out of these lectures?
- Outline of lectures this week and the next
- Some readings for next time
- Deep dive into the course material

Two Mini Courses

- Multi-core Architectures and Shared Resource Management: Fundamentals and Recent Research
  - June 6, 7, 10 (1-4pm)
- Memory Systems in the Multi-Core Era
  - June 13, 14, 17 (1-4pm)

What These Mini Lecture Series is About

- Multi-core Architectures and Shared Resource Management: Fundamentals and Recent Research
- Memory Systems in the Multi-Core Era

- A very "hot" portion of computer architecture research and practice
- A very large design space
- Many opportunities for innovation and groundbreaking research

- We will focus on major aspects of multi-core design
  - Fundamentals
  - Tradeoffs (advantages and disadvantages)
  - Cutting edge research

What These Mini Lecture Series is About

- Goal 1:
  - Build a strong understanding of the fundamentals of the multi-core architectures and the tradeoffs made in their design.
  - Examine how cores and shared resources can be designed.
  - The focus will be on *fundamentals*, *tradeoffs in parallel architecture design*, and *cutting-edge research*.

- Goal 2:
  - Build an understanding of the state-of-the-art research problems in multi-core architectures.
  - Get familiar with some important research papers.
  - You will be expected to read, critique, and discuss research papers.

Course Info: Who Am I?

- Instructor: Prof. Onur Mutlu
  - Carnegie Mellon University ECE/CS
  - PhD from UT-Austin, worked at Microsoft Research, Intel, AMD
  - http://www.ece.cmu.edu/~omutlu
  - onur@cmu.edu (Best way to reach me)
  - http://users.ece.cmu.edu/~omutlu/projects.htm

Interested in developing efficient, high-performance, and scalable (multi-core, memory) systems; solving difficult architectural problems at low cost & complexity

  - Interconnects
  - Hardware/software interaction and co-design (PL, OS, Architecture)
  - Predictable and QoS-aware systems
  - Hardware fault tolerance and security
  - Algorithms and architectures for genome analysis
  - …

A Bit More About My Group and CMU

Pittsburgh, Pennsylvania, USA

- "America's Most Livable City" multiple times; #1 in 2010 by Forbes
- Rated in the top 10 "Smartest Places to Live" for its low cost of living.
- Ranked #1 in the nation and #26 in the world for "Global Livability"
- Rated as one of the top 10 "World's Cleanest Cities"
- Top ten in "green" buildings in the nation, including world's 1st and largest green convention center and Carnegie Mellon's own LEED-certified residence hall, 1st in USA.

Sources: Forbes, Places Rated Almanac, Kiplinger's Personal Finance Magazine, The Economist, MSN Encarta

Carnegie Mellon

Research

- $320+ million per year in sponsored research

Award Highlights

- 17 Nobel Prize Laureates
- 10 Turing Award Winners
- 36 National Academy of Engineering Members
- 10 National Academy of Sciences Members
- 9 American Academy of Arts & Sciences Members
- 12 Fulbright Scholars
- 96 Emmy Award Winners
- 20 Tony Award Winners
- 6 Academy Award (Oscar) Winners

Carnegie Mellon




10,402 undergraduate and graduate students
1,426 faculty members
8:1 student to faculty ratio
72,496 alumni
50 U.S. alumni chapters
20 international alumni chapters
10 degree programs in 12 countries

A Bit More About My Group and CMU

- http://www.ece.cmu.edu/~safari/
- http://www.ece.cmu.edu/~safari/pubs.html

Home | News | People | Projects | Publications | Talks | Technical Reports | Positions | Courses | Conferences

What is SAFARI?

SAFARI is the research group of Professor Onur Mutlu in the Computer Architecture Lab (CALCM) at Carnegie Mellon University. We investigate *safe*, *fair*, *robust* and *intelligent* computer architecture, finding novel ways to provide a substrate with all of these properties for next-generation multicore and manycore systems.

My Students @ SAFARI

- http://www.ece.cmu.edu/~safari/people.html

Who Should Attend This Course?

- You should be motivated to learn about and possibly do research in computer architecture
- Must know some Computer Architecture basics
  - However, ask if you do not know a concept I talk about
- Be willing and ready to
  - Ask questions
  - Think hard
  - Read papers
  - Focus on tradeoffs
  - Discover on your own

What Will I Assume?

- Familiarity with basic computer architecture
- However, you should ask questions

How Can You Make the Best out of These Lectures?

- **Ask and answer questions**
- **Take notes**
- **Participate in discussion**
- **Read discussed papers**
- **Explore on your own**

Homework 0

- Due tonight at midnight Istanbul time

- Send me (onur@cmu.edu) an email with
  - Your name
  - Your picture
  - An interesting fact about something personal to you
  - Why are you interested in these lectures?
  - What do you expect to learn?
  - Anything else you would like to share or ask

What Will You Learn?

- Tentative, Aggressive Schedule
  - Lecture 1: Why multi-core? Basics, alternatives, tradeoffs
    Symmetric versus asymmetric multi-core systems
  - Lecture 2: Shared cache design for multi-cores
    (if time permits) Interconnect design for multi-cores
  - Lecture 3: Data parallelism and GPUs (if time permits)
    (if time permits) Prefetcher design and management

- But, do not believe all of this tentative schedule
  - Why?

- Systems that perform best are usually dynamically scheduled
  - Static vs. Dynamic Scheduling
  - Why do you *really* need dynamic scheduling?

Static versus Dynamic Scheduling

- Static: Done at compile time or parallel task creation time
  - Schedule does not change based on runtime information
- Dynamic: Done at run time (e.g., after tasks are created)
  - Schedule changes based on runtime information
- Example: Parallel Task Assignment

Parallel Task Assignment: Tradeoffs

- Problem: N tasks, P processors, N>P. Do we assign tasks to processors statically (fixed) or dynamically (adaptive)?

- Static assignment
  + Simpler: No movement of tasks.
  - Inefficient: Underutilizes resources when load is not balanced

    When can load not be balanced?

- Dynamic assignment
  + Efficient: Better utilizes processors when load is not balanced
  - More complex: Need to move tasks to balance processor load
  - Higher overhead: Task movement takes time, can disrupt locality

Parallel Task Assignment: Example

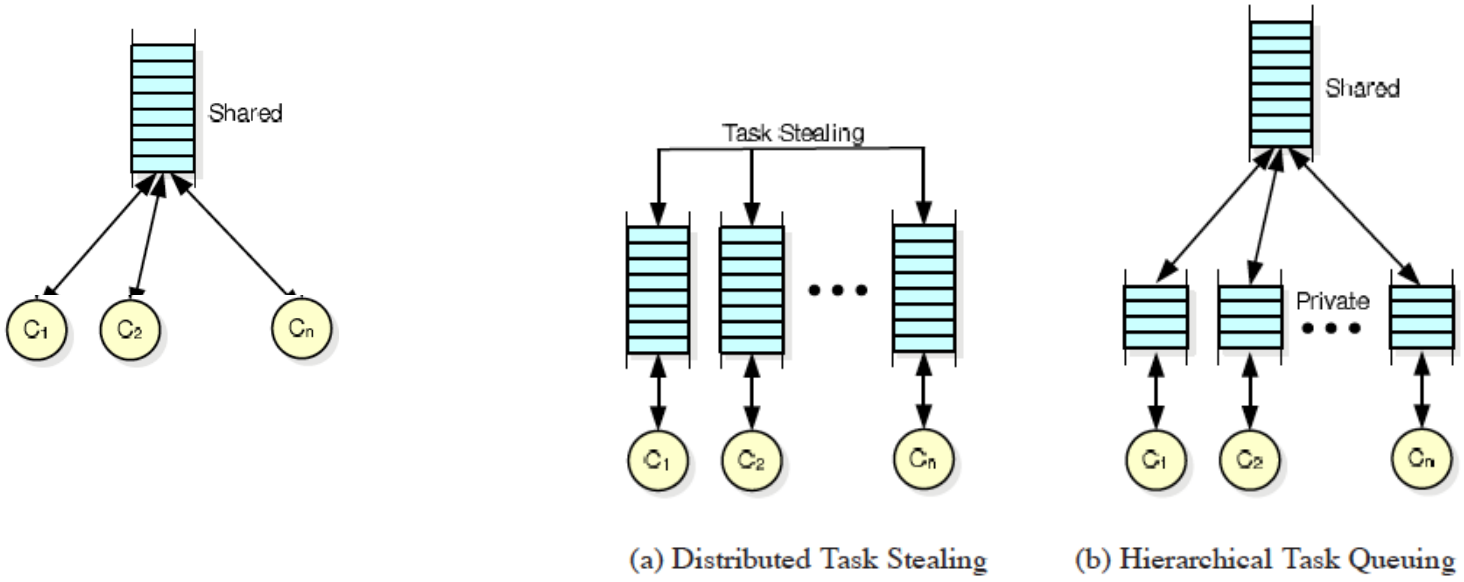
- Compute histogram of a large set of values
- Parallelization:
  - Divide the values across T tasks
  - Each task computes a local histogram for its value set
  - Local histograms merged with global histograms in the end

```
GetPageHistogram(Page *P)

    For each thread: {

        /* Parallel part of the function */
        UpdateLocalHistogram(Fraction of Page)

        /* Serial part of the function */
        Critical Section:
            Add local histogram to global histogram

        Barrier
    }

    Return global histogram
```

Parallel Task Assignment: Example (II)

- How to schedule tasks updating local histograms?
  - Static: Assign equal number of tasks to each processor
  - Dynamic: Assign tasks to a processor that is available
  - When does static work as well as dynamic?

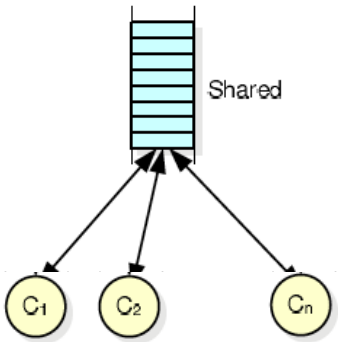
- Implementation of Dynamic Assignment with Task Queues

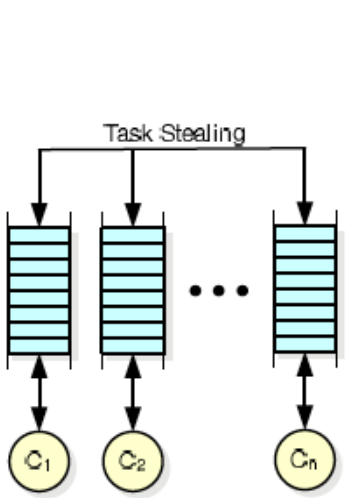


(a) Distributed Task Stealing (b) Hierarchical Task Queuing

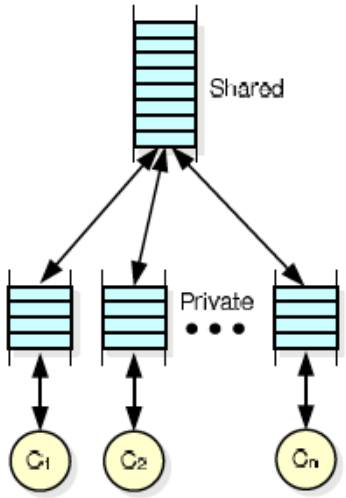
Software Task Queues

- What are the advantages and disadvantages of each?
  - Centralized
  - Distributed
  - Hierarchical





(a) Distributed Task Stealing



(b) Hierarchical Task Queuing

Task Stealing

- Idea: When a processor's task queue is empty it steals a task from another processor's task queue
  - Whom to steal from? (Randomized stealing works well)
  - How many tasks to steal?

+ Dynamic balancing of computation load

- Additional communication/synchronization overhead between processors
- Need to stop stealing if no tasks to steal

Parallel Task Assignment: Tradeoffs

- Who does the assignment? Hardware versus software?

- Software
  - + Better scope
  - - More time overhead
  - - Slow to adapt to dynamic events (e.g., a processor becoming idle)

- Hardware
  - + Low time overhead
  - + Can adjust to dynamic events faster
  - - Requires hardware changes (area and possibly energy overhead)

How Can the Hardware Help?

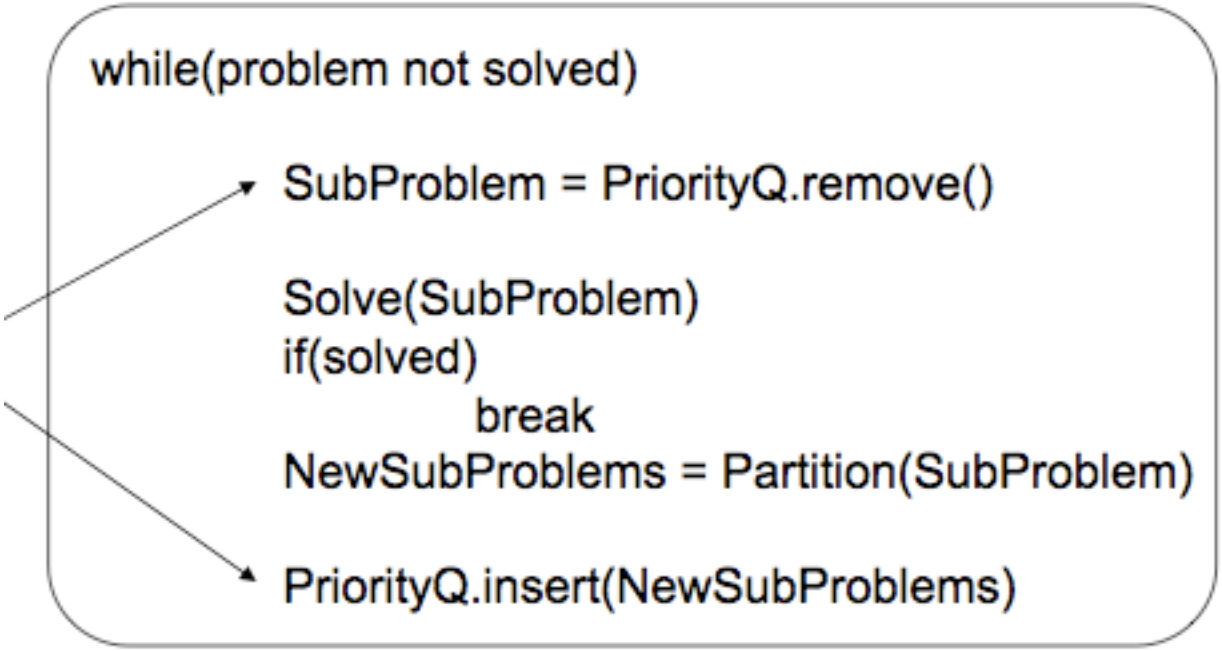
- Managing task queues in software has overhead
  - Especially high when task sizes are small

- An idea: Hardware Task Queues
  - Each processor has a dedicated task queue
  - Software fills the task queues (on demand)
  - Hardware manages movement of tasks from queue to queue
  - There can be a global task queue as well → hierarchical tasking in hardware

  - Kumar et al., "Carbon: Architectural Support for Fine-Grained Parallelism on Chip Multiprocessors," ISCA 2007.
    - Optional reading

Dynamic Task Generation

- Does static task assignment work in this case?
- Problem: Searching the exit of a maze



Why Do We Really Want Dynamic Scheduling?

- Uncertainty in dynamic events

- E.g., Out-of-order execution (dynamic instruction scheduling)
  - Really necessary if you do not know the latency of an instruction
  - Compiler cannot reorder instructions with unknown latencies

What Will You Learn in Mini Course 1?

- Multi-core Architectures and Shared Resource Management: Fundamentals and Recent Research
  - June 6, 7, 10 (1-4pm)

- Lecture 1: Why multi-core? Basics, alternatives, tradeoffs
  Symmetric versus asymmetric multi-core systems
- Lecture 2: Shared cache design for multi-cores
  (if time permits) Interconnect design for multi-cores
- Lecture 3: Data parallelism and GPUs (if time permits)
  (if time permits) Prefetcher design and management

What Will You Learn in Mini Course 2?

- Memory Systems in the Multi-Core Era
  - June 13, 14, 17 (1-4pm)

- Lecture 1: Main memory basics, DRAM scaling
- Lecture 2: Emerging memory technologies and hybrid memories
- Lecture 3: Main memory interference and QoS

Readings for Lecture Today (Lecture 1.1)

- Required
  - Mutlu et al., "Runahead Execution: An Alternative to Very Large Instruction Windows for Out-of-order Processors," HPCA 2003, IEEE Micro 2003.
  - Suleman et al., "Accelerating Critical Section Execution with Asymmetric Multi-Core Architectures," ASPLOS 2009, IEEE Micro 2010.
  - Suleman et al., "Data Marshaling for Multi-Core Architectures," ISCA 2010, IEEE Micro 2011.
  - Joao et al., "Bottleneck Identification and Scheduling for Multithreaded Applications," ASPLOS 2012.
  - Joao et al., "Utility-Based Acceleration of Multithreaded Applications on Asymmetric CMPs," ISCA 2013.
- Recommended
  - Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," AFIPS 1967.
  - Olukotun et al., "The Case for a Single-Chip Multiprocessor," ASPLOS 1996.
  - Mutlu et al., "Techniques for Efficient Processing in Runahead Execution Engines," ISCA 2005, IEEE Micro 2006.

Videos for Lecture Today (Lecture 1.1)

- Runahead Execution
  - http://www.youtube.com/watch?v=z8YpjqXQJIA&list=PL5PHm2jkkXmidJOd59REog9jDnPDTG6IJ&index=28

- Multiprocessors
  - Basics: http://www.youtube.com/watch?v=7ozCK_Mgxfk&list=PL5PHm2jkkXmidJOd59REog9jDnPDTG6IJ&index=31
  - Correctness and Coherence: http://www.youtube.com/watch?v=U-VZKMgItDM&list=PL5PHm2jkkXmidJOd59REog9jDnPDTG6IJ&index=32
  - Heterogeneous Multi-Core: http://www.youtube.com/watch?v=r6r2NJxj3kI&list=PL5PHm2jkkXmidJOd59REog9jDnPDTG6IJ&index=34

Readings for Lecture June 7 (Lecture 1.2)

- Required
  - Qureshi et al., "A Case for MLP-Aware Cache Replacement," ISCA 2005.
  - Seshadri et al., "The Evicted-Address Filter: A Unified Mechanism to Address both Cache Pollution and Thrashing," PACT 2012.
  - Pekhimenko et al., "Base-Delta-Immediate Compression: Practical Data Compression for On-Chip Caches," PACT 2012.
  - Pekhimenko et al., "Linearly Compressed Pages: A Main Memory Compression Framework with Low Complexity and Low Latency," SAFARI Technical Report 2013.

- Recommended
  - Qureshi et al., "Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches," MICRO 2006.

Videos for Lecture 1.2

- Cache basics:
  - http://www.youtube.com/watch?v=TpMdBrM1hVc&list=PL5PHm2jkkXmidJOd59REog9jDnPDTG6IJ&index=23

- Advanced caches:
  - http://www.youtube.com/watch?v=TboaFbjTd-E&list=PL5PHm2jkkXmidJOd59REog9jDnPDTG6IJ&index=24

Readings for Lecture June 10 (Lecture 1.3)

- Required
  - Moscibroda and Mutlu, "A Case for Bufferless Routing in On-Chip Networks," ISCA 2009.
  - Fallin et al., "CHIPPER: A Low-Complexity Bufferless Deflection Router," HPCA 2011.
  - Fallin et al., "MinBD: Minimally-Buffered Deflection Routing for Energy-Efficient Interconnect," NOCS 2012.
  - Das et al., "Application-Aware Prioritization Mechanisms for On-Chip Networks," MICRO 2009.
  - Das et al., "Aergia: Exploiting Packet Latency Slack in On-Chip Networks," ISCA 2010, IEEE Micro 2011.

- Recommended
  - Grot et al. "Preemptive Virtual Clock: A Flexible, Efficient, and Cost-effective QOS Scheme for Networks-on-Chip," MICRO 2009.
  - Grot et al., "Kilo-NOC: A Heterogeneous Network-on-Chip Architecture for Scalability and Service Guarantees," ISCA 2011, IEEE Micro 2012.

Videos for Lecture 1.3

- Interconnects
  - http://www.youtube.com/watch?v=6xEpbFVgnf8&list=PL5PHm2jkkXmidJOd59REog9jDnPDTG6IJ&index=33

- GPUs and SIMD processing
  - Vector/array processing basics: http://www.youtube.com/watch?v=f-XL4BNRoBA&list=PL5PHm2jkkXmidJOd59REog9jDnPDTG6IJ&index=15
  - GPUs versus other execution models: http://www.youtube.com/watch?v=vr5hbSkb1Eg&list=PL5PHm2jkkXmidJOd59REog9jDnPDTG6IJ&index=20
  - GPUs in more detail: http://www.youtube.com/watch?v=vr5hbSkb1Eg&list=PL5PHm2jkkXmidJOd59REog9jDnPDTG6IJ&index=20

Online Lectures and More Information

- Online Computer Architecture Lectures
  - http://www.youtube.com/playlist?list=PL5PHm2jkkXmidJOd59REog9jDnPDTG6IJ
- Online Computer Architecture Courses
  - Intro: http://www.ece.cmu.edu/~ece447/s13/doku.php
  - Advanced: http://www.ece.cmu.edu/~ece740/f11/doku.php
  - Advanced: http://www.ece.cmu.edu/~ece742/doku.php
- Recent Research Papers
  - http://users.ece.cmu.edu/~omutlu/projects.htm
  - http://scholar.google.com/citations?user=7XyGUGkAAAAJ&hl=en

Parallel Computer Architecture Basics

What is a Parallel Computer?

- Definition of a "parallel computer" not really precise
- "A 'parallel computer' is a "collection of processing elements that communicate and cooperate to solve large problems fast"
  - Almasi and Gottlieb, "Highly Parallel Computing," 1989
- Is a superscalar processor a parallel computer?

- A processor that gives the illusion of executing a sequential ISA on a single thread at a time is a sequential machine
- Almost anything else is a parallel machine

- Examples of parallel machines:
  - Multiple program counters (PCs)
  - Multiple data being operated on simultaneously
  - Some combination

Flynn's Taxonomy of Computers

- Mike Flynn, "Very High-Speed Computing Systems," Proc. of IEEE, 1966

- SISD: Single instruction operates on single data element
- SIMD: Single instruction operates on multiple data elements
  - Array processor
  - Vector processor
- MISD: Multiple instructions operate on single data element
  - Closest form: systolic array processor, streaming processor
- MIMD: Multiple instructions operate on multiple data elements (multiple instruction streams)
  - Multiprocessor
  - Multithreaded processor

Why Parallel Computers?

- Parallelism: Doing multiple things at a time
- Things: instructions, operations, tasks

- Main Goal
  - Improve performance (Execution time or task throughput)
    - Execution time of a program governed by Amdahl's Law

- Other Goals
  - Reduce power consumption
    - (4N units at freq F/4) consume less power than (N units at freq F)
    - Why?
  - Improve cost efficiency and scalability, reduce complexity
    - Harder to design a single unit that performs as well as N simpler units
  - Improve dependability: Redundant execution in space

Types of Parallelism and How to Exploit Them

- Instruction Level Parallelism
  - Different instructions within a stream can be executed in parallel
  - Pipelining, out-of-order execution, speculative execution, VLIW
  - Dataflow

- Data Parallelism
  - Different pieces of data can be operated on in parallel
  - SIMD: Vector processing, array processing
  - Systolic arrays, streaming processors

- Task Level Parallelism
  - Different "tasks/threads" can be executed in parallel
  - Multithreading
  - Multiprocessing (multi-core)

Task-Level Parallelism: Creating Tasks

- Partition a single problem into multiple related tasks (threads)
  - Explicitly: Parallel programming
    - Easy when tasks are natural in the problem
      - Web/database queries
    - Difficult when natural task boundaries are unclear
  - Transparently/implicitly: Thread level speculation
    - Partition a single thread speculatively
- Run many independent tasks (processes) together
  - Easy when there are many processes
    - Batch simulations, different users, cloud computing workloads
  - Does not improve the performance of a single task

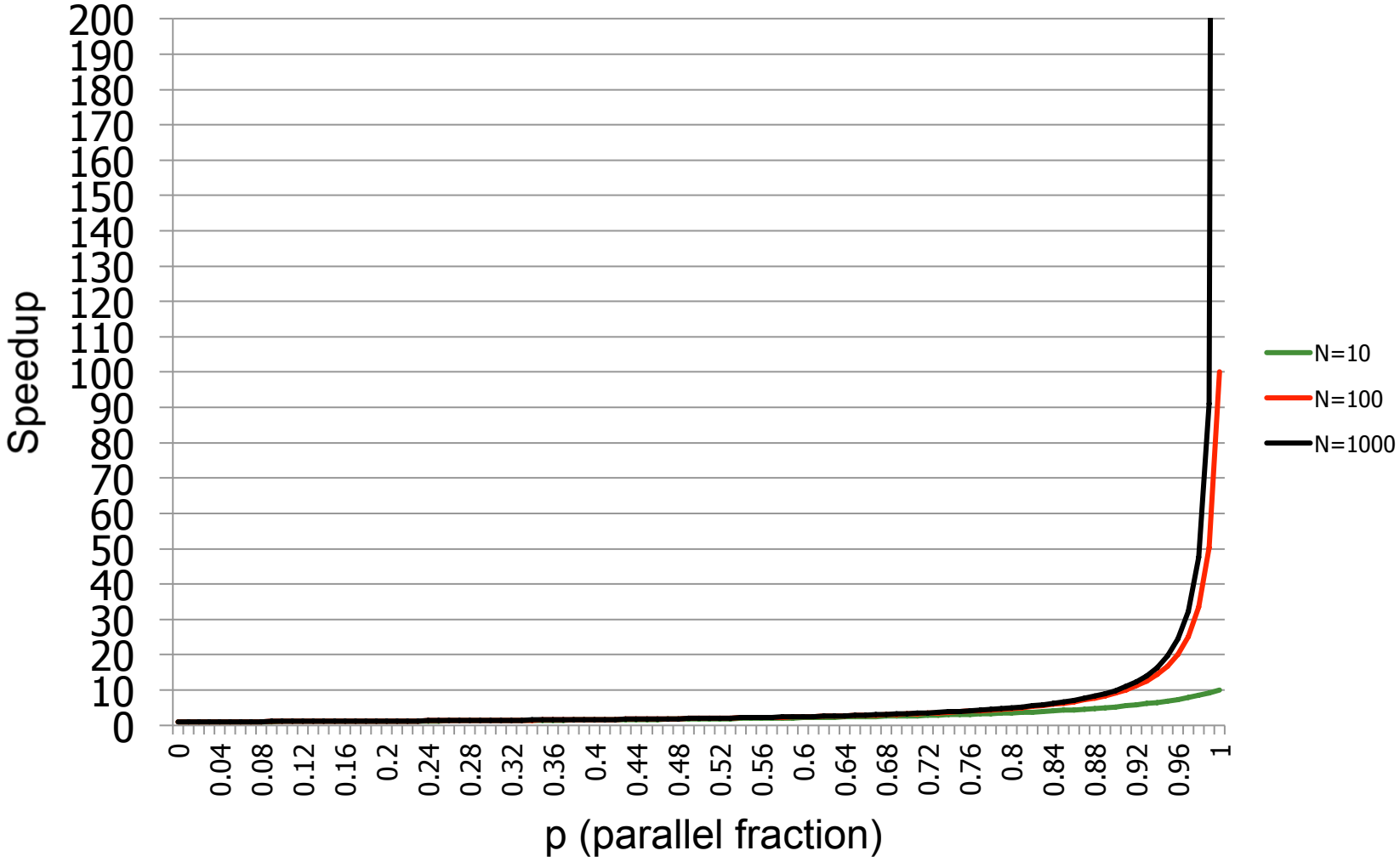
Caveats of Parallelism

- Amdahl's Law
  - p: Parallelizable fraction of a program
  - N: Number of processors

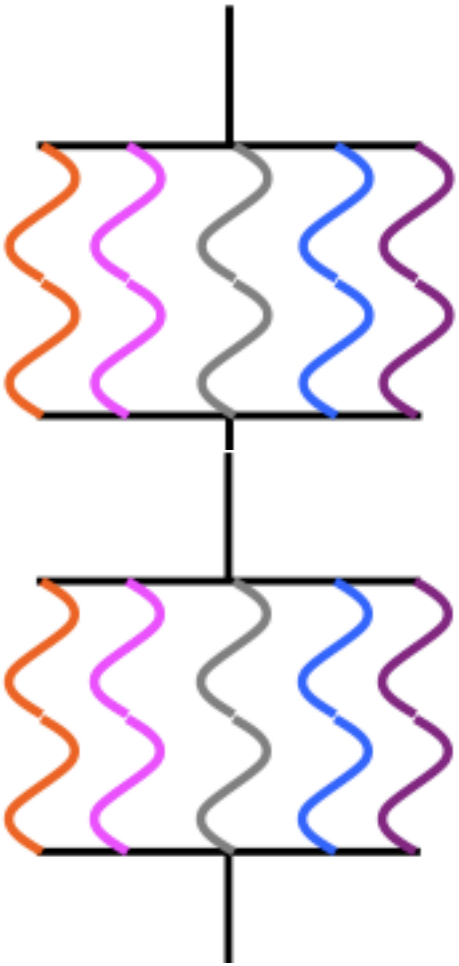
$$\text{Speedup} = \frac{1}{(1 - p) + \frac{p}{N}}$$

  - 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)

Sequential Bottleneck



Why the Sequential Bottleneck?

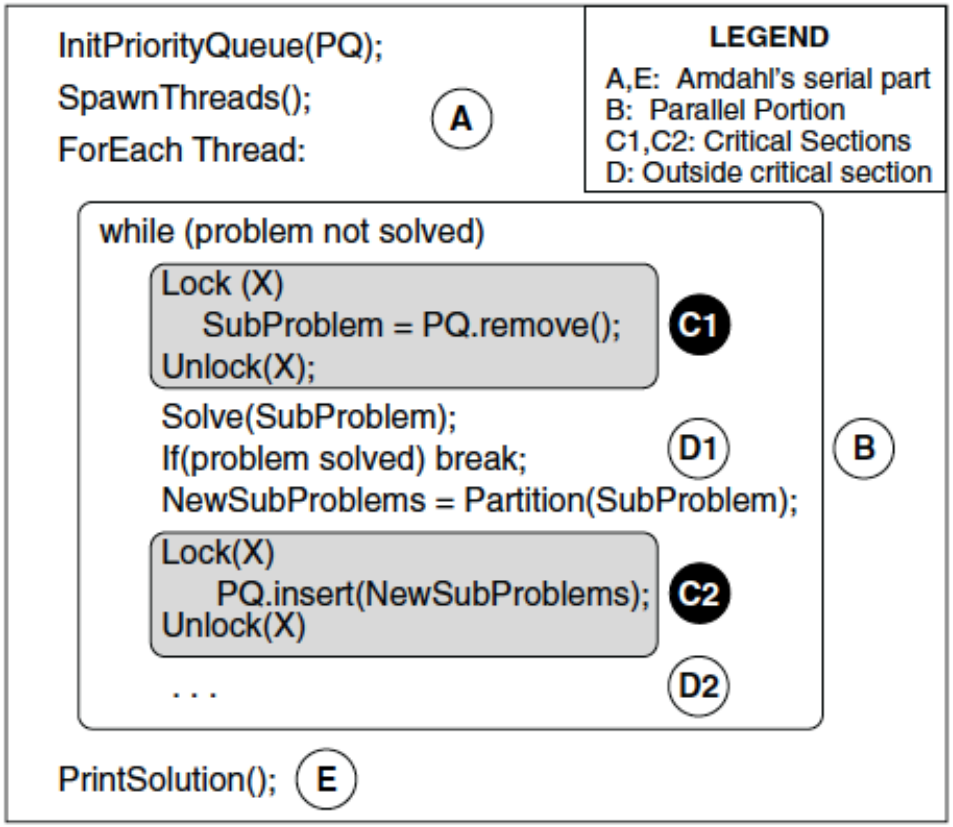


- Parallel machines have the sequential bottleneck
- Main cause: Non-parallelizable operations on data (e.g. non-parallelizable loops)

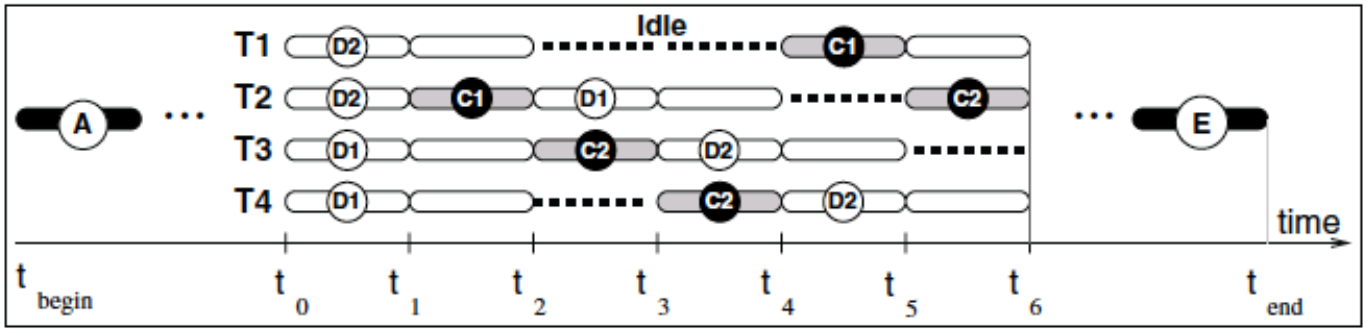
```
for ( i = 0 ; i < N; i++)
    A[i] = (A[i] + A[i-1]) / 2
```

- Single thread prepares data and spawns parallel tasks (usually sequential)

Another Example of Sequential Bottleneck



(a)



Bottlenecks in Parallel Portion

- Synchronization: Operations manipulating shared data cannot be parallelized
  - Locks, mutual exclusion, barrier synchronization
  - Communication: Tasks may need values from each other

  - Causes thread serialization when shared data is contended

- Load Imbalance: Parallel tasks may have different lengths
  - Due to imperfect parallelization or microarchitectural effects

  - Reduces speedup in parallel portion

- Resource Contention: Parallel tasks can share hardware resources, delaying each other
  - Replicating all resources (e.g., memory) expensive

  - Additional latency not present when each task runs alone

Difficulty in Parallel Programming

- Little difficulty if parallelism is natural
  - "Embarrassingly parallel" applications
  - Multimedia, physical simulation, graphics
  - Large web servers, databases?

- Difficulty is in
  - Getting parallel programs to work correctly
  - Optimizing performance in the presence of bottlenecks

- Much of **parallel computer architecture** is about
  - Designing machines that overcome the sequential and parallel bottlenecks to achieve higher performance and efficiency
  - Making programmer's job easier in writing correct and high-performance parallel programs

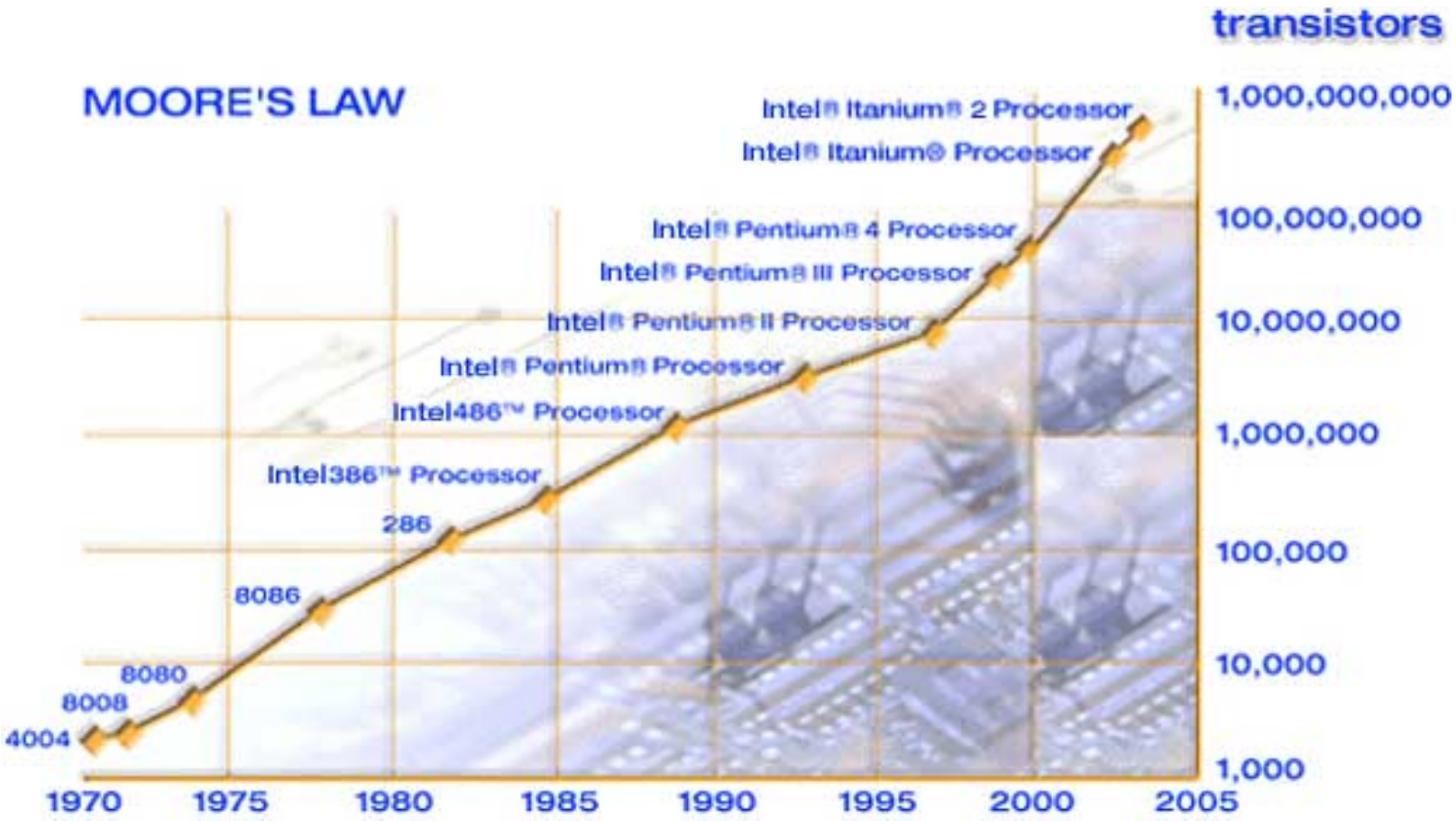
Multiprocessor Types

- Loosely coupled multiprocessors
  - No shared global memory address space
  - Multicomputer network
    - Network-based multiprocessors
  - Usually programmed via message passing
    - Explicit calls (send, receive) for communication

- Tightly coupled multiprocessors
  - Shared global memory address space
  - Traditional multiprocessing: symmetric multiprocessing (SMP)
    - Existing multi-core processors, multithreaded processors
  - Programming model similar to uniprocessors (i.e., multitasking uniprocessor) except
    - Operations on shared data require synchronization

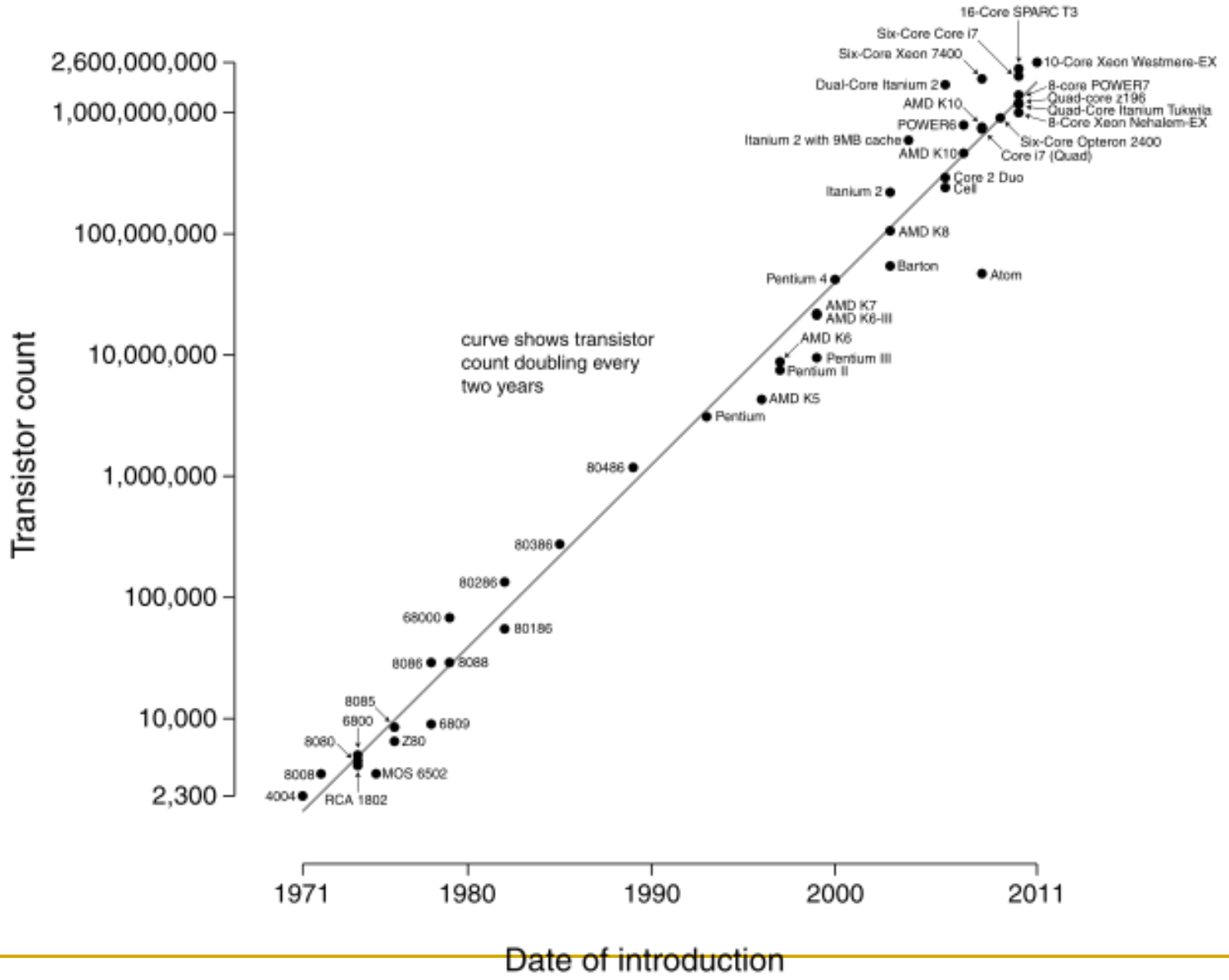
Multi-Core Processors

Moore's Law



Moore, "Cramming more components onto integrated circuits," Electronics Magazine, 1968.

Microprocessor Transistor Counts 1971-2011 & Moore's Law



Multi-Core

- Idea: Put multiple processors on the same die.

- Technology scaling (Moore's Law) enables more transistors to be placed on the same die area

- What else could you do with the die area you dedicate to multiple processors?
  - Have a bigger, more powerful core
  - Have larger caches in the memory hierarchy
  - Simultaneous multithreading
  - Integrate platform components on chip (e.g., network interface, memory controllers)

Why Multi-Core?

- Alternative: Bigger, more powerful single core
  - Larger superscalar issue width, larger instruction window, more execution units, large trace caches, large branch predictors, etc

+ Improves single-thread performance transparently to programmer, compiler

- Very difficult to design (Scalable algorithms for improving single-thread performance elusive)

- Power hungry – many out-of-order execution structures consume significant power/area when scaled. Why?

- Diminishing returns on performance

- Does not significantly help memory-bound application performance (Scalable algorithms for this elusive)

Large Superscalar vs. Multi-Core

- Olukotun et al., "The Case for a Single-Chip Multiprocessor," ASPLOS 1996.

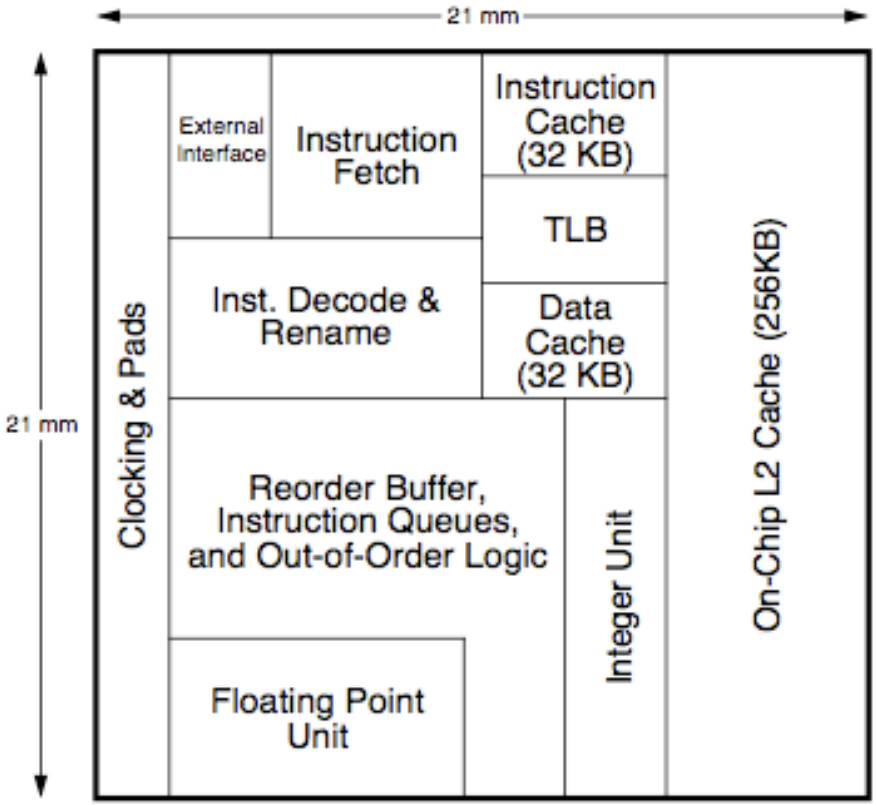


Figure 2. Floorplan for the six-issue dynamic superscalar microprocessor.

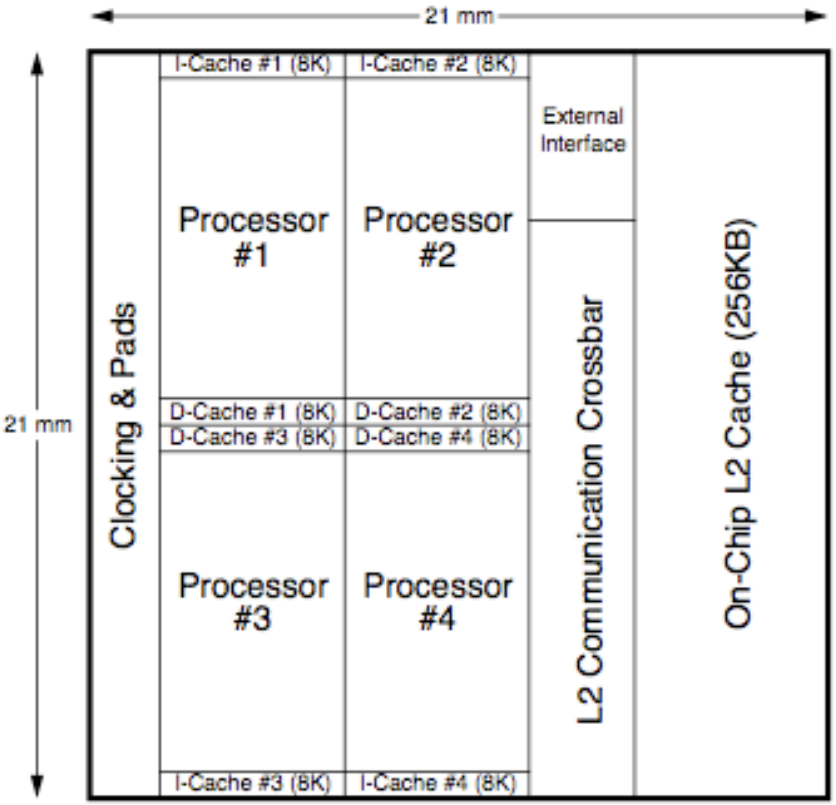


Figure 3. Floorplan for the four-way single-chip multiprocessor.

Multi-Core vs. Large Superscalar

- Multi-core advantages
  - + Simpler cores → more power efficient, lower complexity, easier to design and replicate, higher frequency (shorter wires, smaller structures)
  - + Higher system throughput on multiprogrammed workloads → reduced context switches
  - + Higher system throughput in parallel applications

- Multi-core disadvantages
  - - Requires parallel tasks/threads to improve performance (parallel programming)
  - - Resource sharing can reduce single-thread performance
  - - Shared hardware resources need to be managed
  - - Number of pins limits data supply for increased demand

Large Superscalar vs. Multi-Core

- Olukotun et al., "The Case for a Single-Chip Multiprocessor," ASPLOS 1996.

- Technology push
  - Instruction issue queue size limits the cycle time of the superscalar, OoO processor → diminishing performance
    - Quadratic increase in complexity with issue width
  - Large, multi-ported register files to support large instruction windows and issue widths → reduced frequency or longer RF access, diminishing performance
- Application pull
  - Integer applications: little parallelism?
  - FP applications: abundant loop-level parallelism
  - Others (transaction proc., multiprogramming): CMP better fit

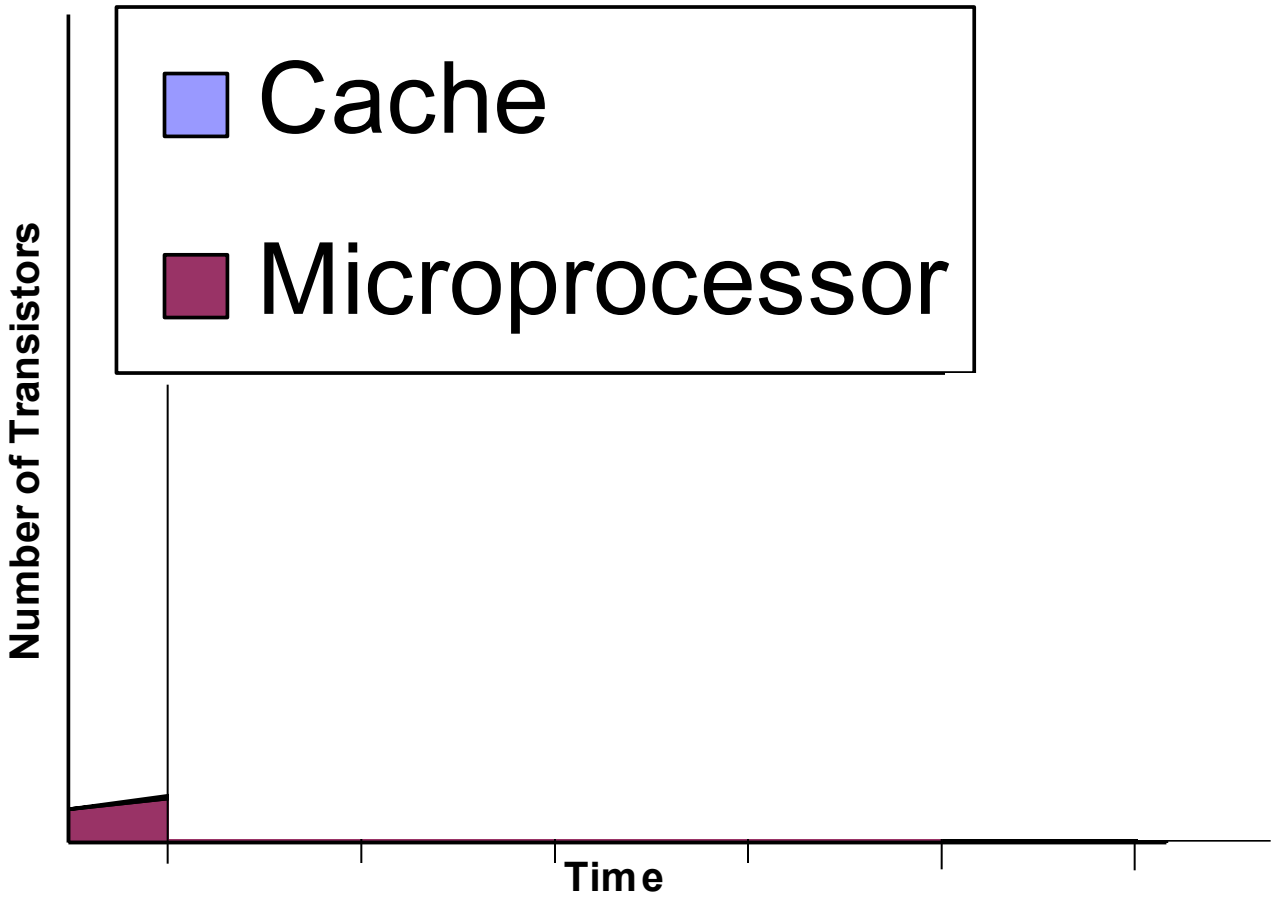
Why Multi-Core?

- Alternative: Bigger caches

  + Improves single-thread performance transparently to programmer, compiler
  + Simple to design

  - Diminishing single-thread performance returns from cache size. Why?
  - Multiple levels complicate memory hierarchy

Cache vs. Core



Why Multi-Core?

- Alternative: (Simultaneous) Multithreading

+ Exploits thread-level parallelism (just like multi-core)
+ Good single-thread performance with SMT
+ No need to have an entire core for another thread
+ Parallel performance aided by tight sharing of caches

- Scalability is limited: need bigger register files, larger issue width (and associated costs) to have many threads → complex with many threads
- Parallel performance limited by shared fetch bandwidth
- Extensive resource sharing at the pipeline and memory system reduces both single-thread and parallel application performance

Why Multi-Core?

- Alternative: Integrate platform components on chip instead

  + Speeds up many system functions (e.g., network interface cards, Ethernet controller, memory controller, I/O controller)

  - Not all applications benefit (e.g., CPU intensive code sections)

Why Multi-Core?

- Alternative: More scalable superscalar, out-of-order engines
  - Clustered superscalar processors (with multithreading)

+ Simpler to design than superscalar, more scalable than simultaneous multithreading (less resource sharing)
+ Can improve both single-thread and parallel application performance

- Diminishing performance returns on single thread: Clustering reduces IPC performance compared to monolithic superscalar. Why?
- Parallel performance limited by shared fetch bandwidth
- Difficult to design

Clustered Superscalar+OoO Processors

- Clustering (e.g., Alpha 21264 integer units)
  - Divide the scheduling window (and register file) into multiple clusters
  - Instructions steered into clusters (e.g. based on dependence)
  - Clusters schedule instructions out-of-order, within cluster scheduling can be in-order
  - Inter-cluster communication happens via register files (no full bypass)

+ Smaller scheduling windows, simpler wakeup algorithms

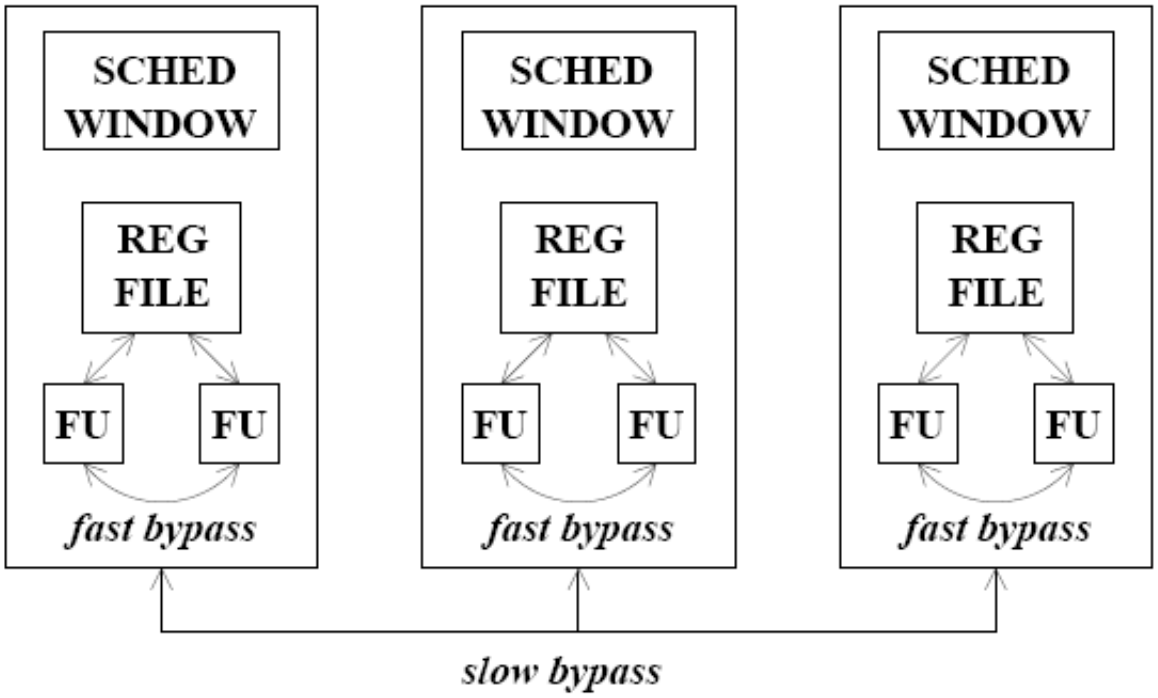
+ Smaller ports into register files

+ Faster within-cluster bypass

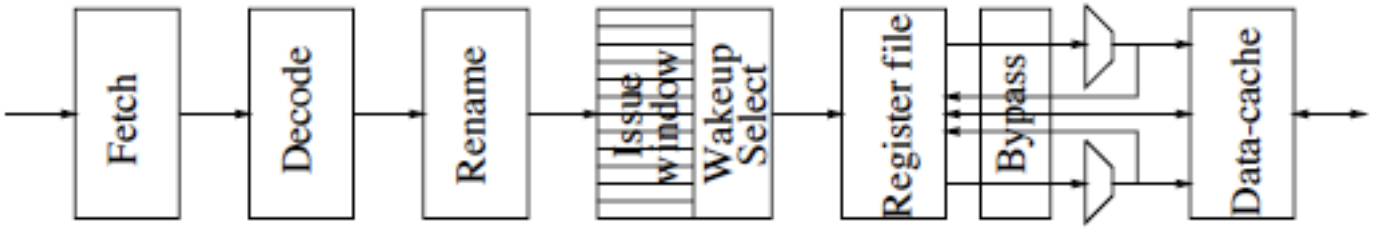
-- Extra delay when instructions require across-cluster communication

Clustering (I)

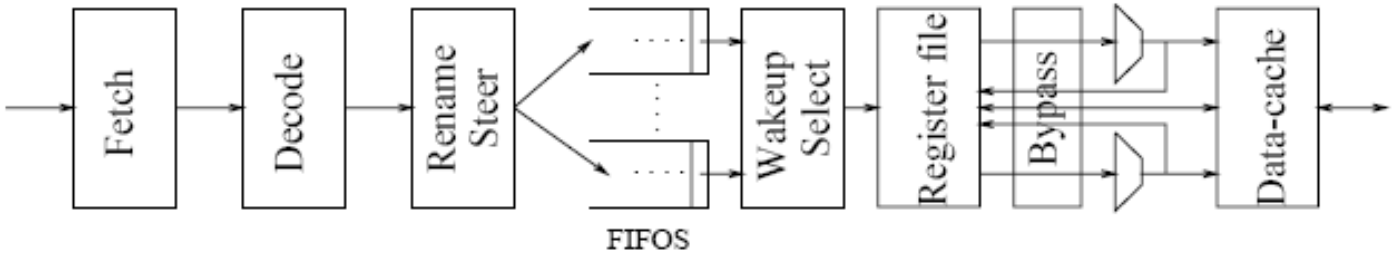
- Scheduling within each cluster can be out of order



Clustering (II)

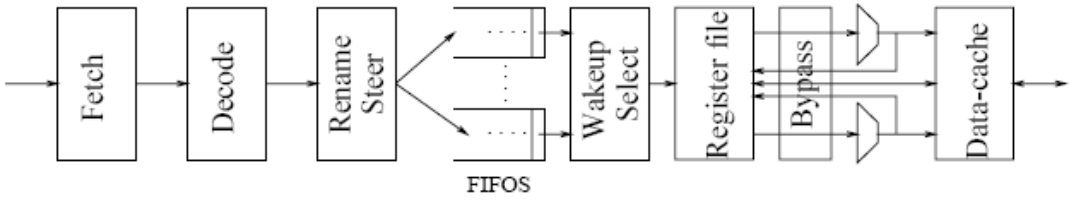


FETCH	DECODE	RENAME	WAKEUP SELECT	REG READ	EXECUTE BYPASS	DCACHE ACCESS	REG WRITE COMMIT

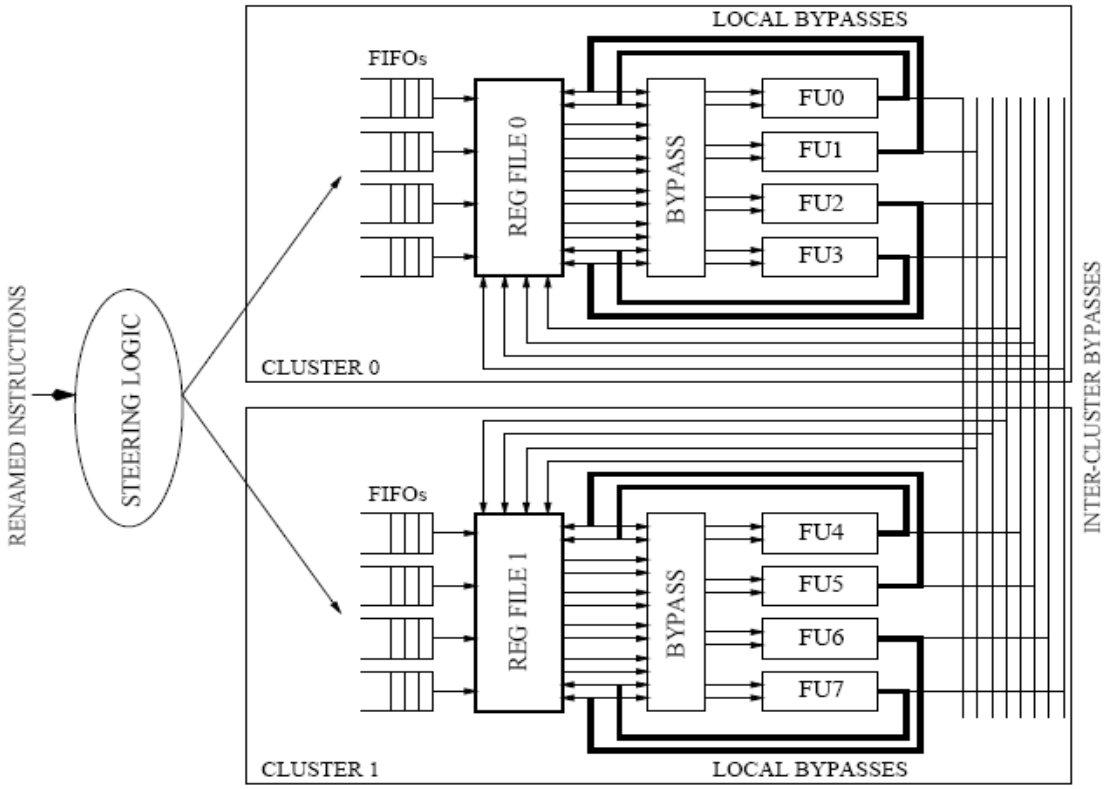


FETCH	DECODE	RENAME STEER	WAKEUP SELECT	REG READ	EXECUTE BYPASS	DCACHE ACCESS	REG WRITE COMMIT

Clustering (III)



FETCH	DECODE	RENAME STEER	WAKEUP SELECT	REG READ	EXECUTE BYPASS	DCACHE ACCESS	REG WRITE COMMIT



Each scheduler is a FIFO
+ Simpler
+ Can have N FIFOs
(OoO w.r.t. each other)
+ Reduces scheduling complexity
-- More dispatch stalls

Inter-cluster bypass: Results produced by an FU in Cluster 0 is not individually forwarded to each FU in another cluster.

- Palacharla et al., "Complexity Effective Superscalar Processors," ISCA 1997.

Why Multi-Core?

- Alternative: Traditional symmetric multiprocessors

  + Smaller die size (for the same processing core)
  + More memory bandwidth (no pin bottleneck)
  + Fewer shared resources → less contention between threads

  - Long latencies between cores (need to go off chip) → shared data accesses limit performance → parallel application scalability is limited
  - Worse resource efficiency due to less sharing → worse power/ energy efficiency

Why Multi-Core?

- Other alternatives?
  - Dataflow?
  - Vector processors (SIMD)?
  - Integrating DRAM on chip?
  - Reconfigurable logic? (general purpose?)

Review: Multi-Core Alternatives

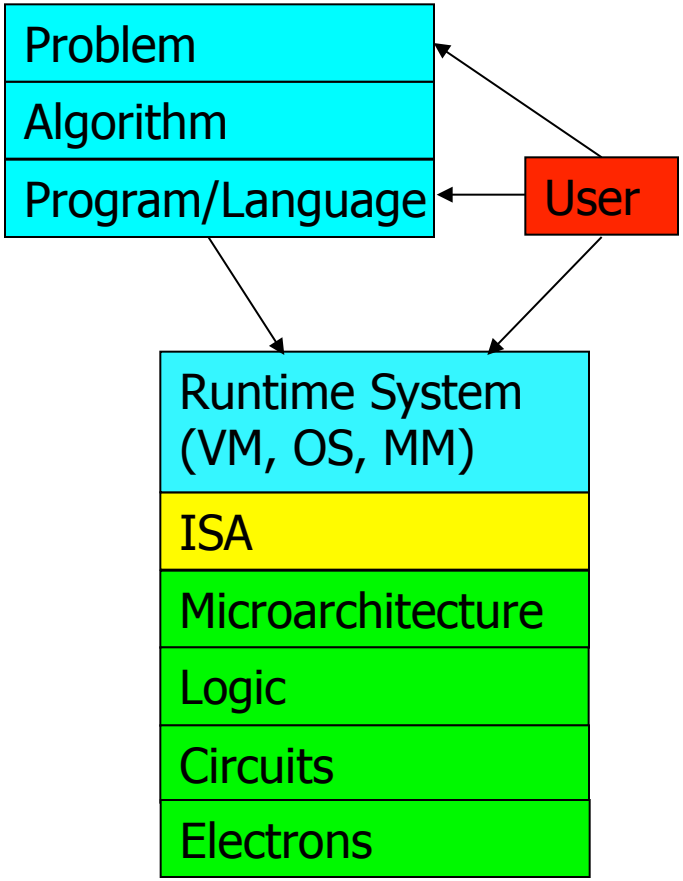
- Bigger, more powerful single core
- Bigger caches
- (Simultaneous) multithreading
- Integrate platform components on chip instead
- More scalable superscalar, out-of-order engines
- Traditional symmetric multiprocessors
- Dataflow?
- Vector processors (SIMD)?
- Integrating DRAM on chip?
- Reconfigurable logic? (general purpose?)
- Other alternatives?
- Your solution?

Computer Architecture Today (I)

- Today is a very exciting time to study computer architecture
- Industry is in a large paradigm shift (to multi-core and beyond) – many different potential system designs possible
- Many difficult problems *motivating* and *caused by* the shift
  - Power/energy constraints
  - Complexity of design → multi-core?
  - Difficulties in technology scaling → new technologies?
  - Memory wall/gap
  - Reliability wall/issues
  - Programmability wall/problem
- No clear, definitive answers to these problems

Computer Architecture Today (II)

- These problems affect all parts of the computing stack – if we do not change the way we design systems



- No clear, definitive answers to these problems

Computer Architecture Today (III)

- You can revolutionize the way computers are built, if you understand both the hardware and the software (and change each accordingly)

- You can invent new paradigms for computation, communication, and storage

- Recommended book: Kuhn, "The Structure of Scientific Revolutions" (1962)
  - Pre-paradigm science: no clear consensus in the field
  - Normal science: dominant theory used to explain things (business as usual); exceptions considered anomalies
  - Revolutionary science: underlying assumptions re-examined

… but, first …

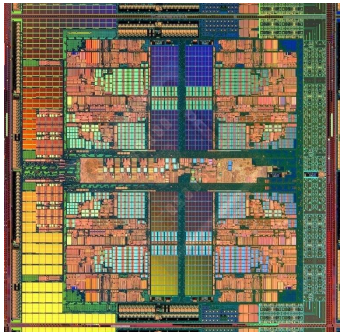
- Let's understand the fundamentals…

- You can change the world only if you understand it well enough…
  - Especially the past and present dominant paradigms
  - And, their advantages and shortcomings -- tradeoffs

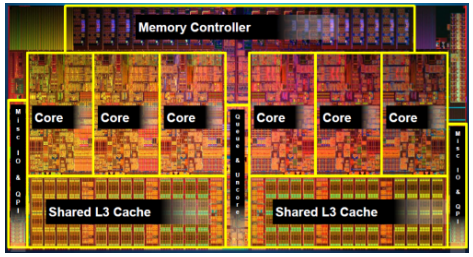
Multi-Core Design

Many Cores on Chip

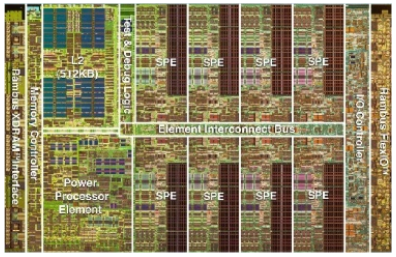
- Simpler and lower power than a single large core
- Large scale parallelism on chip



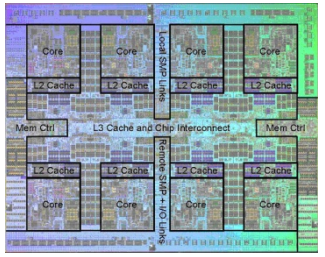
AMD Barcelona
4 cores



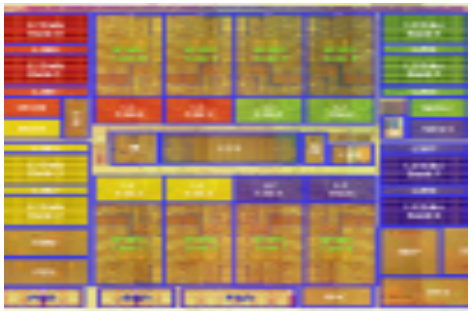
Intel Core i7
8 cores



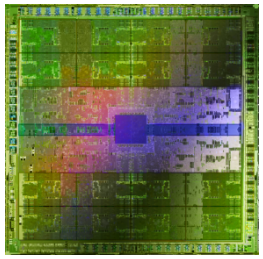
IBM Cell BE
8+1 cores



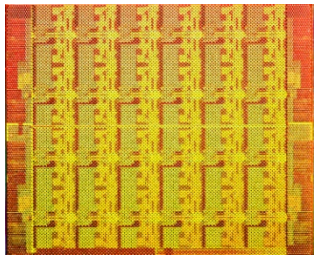
IBM POWER7
8 cores



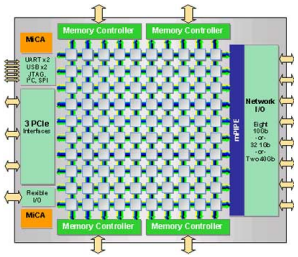
Sun Niagara II
8 cores



Nvidia Fermi
448 “cores”



Intel SCC
48 cores, networked



Tilera 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)

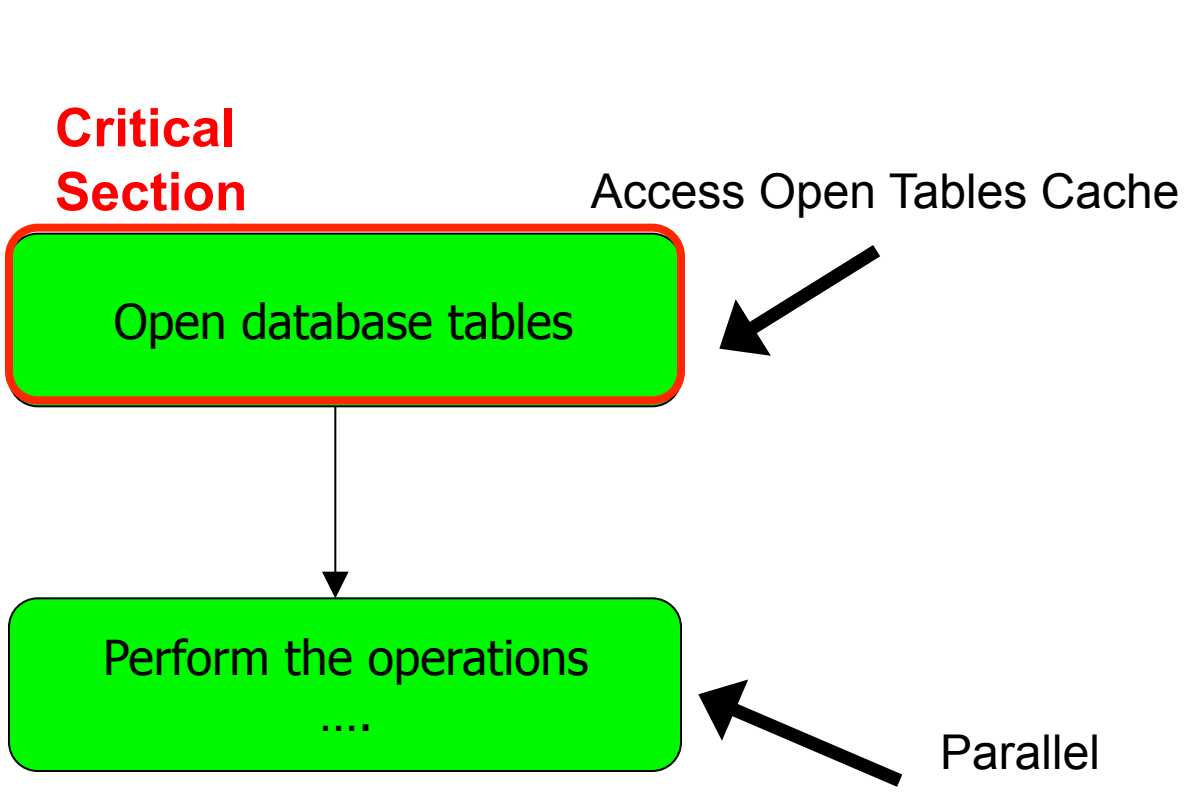
The Problem: Serialized Code Sections

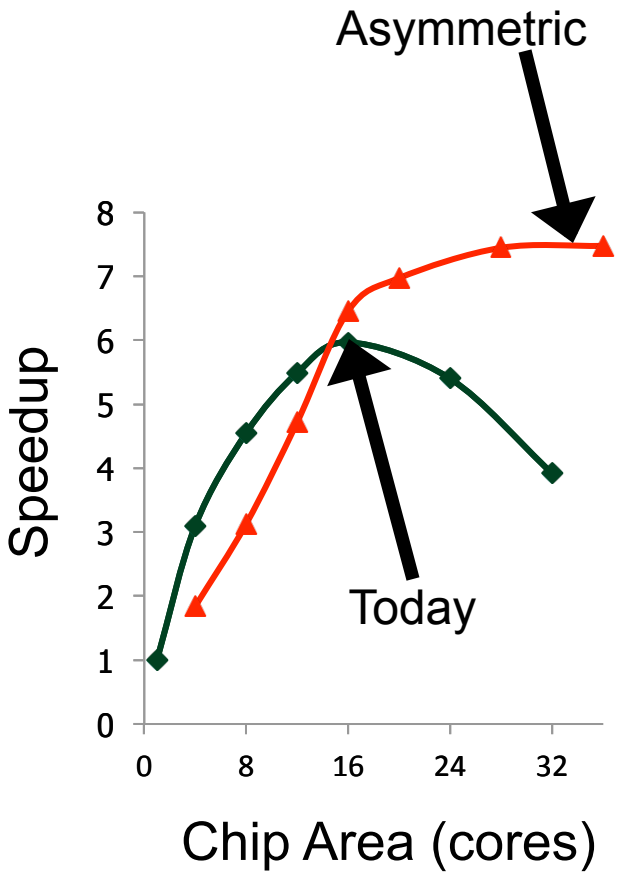
- Many parallel programs cannot be parallelized completely

- 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

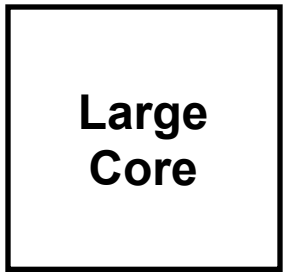




Demands in Different Code Sections

- 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



- *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*

- *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)

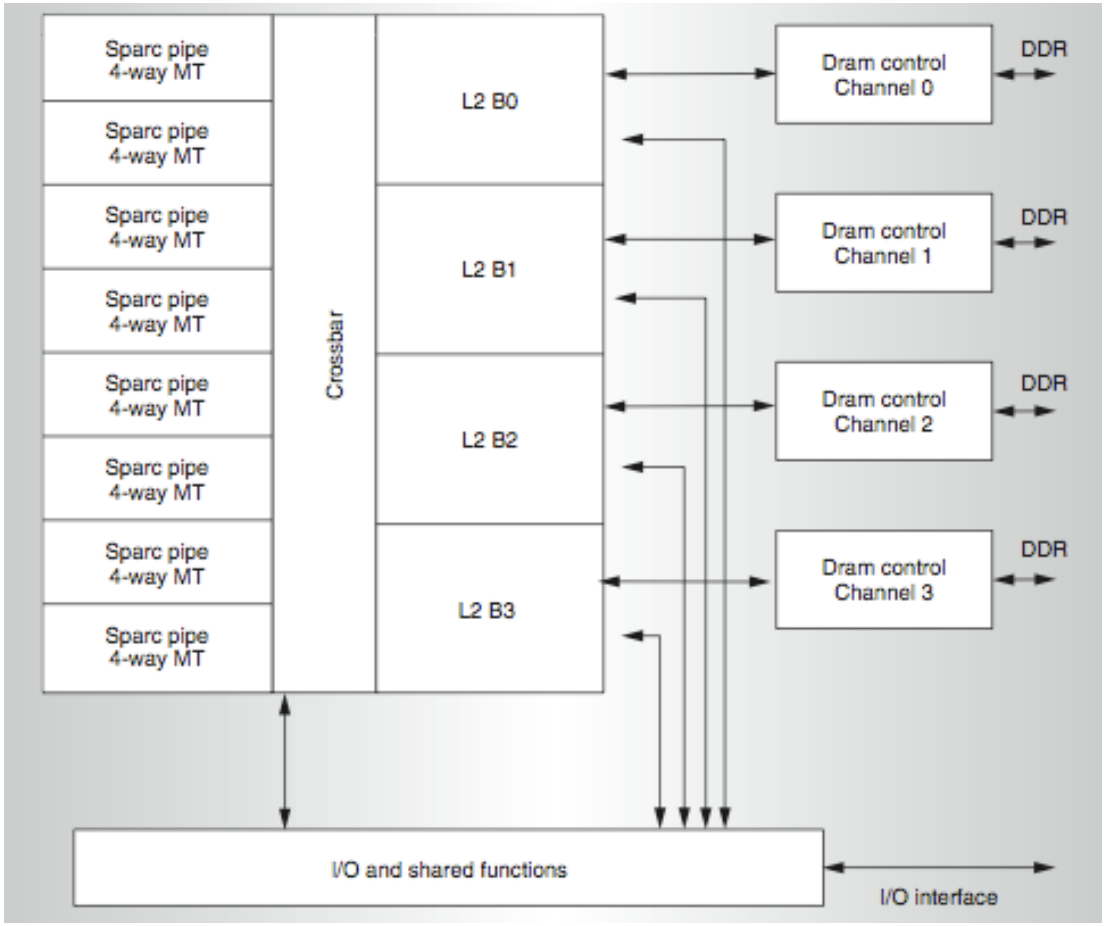
Large vs. Small Cores

- 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

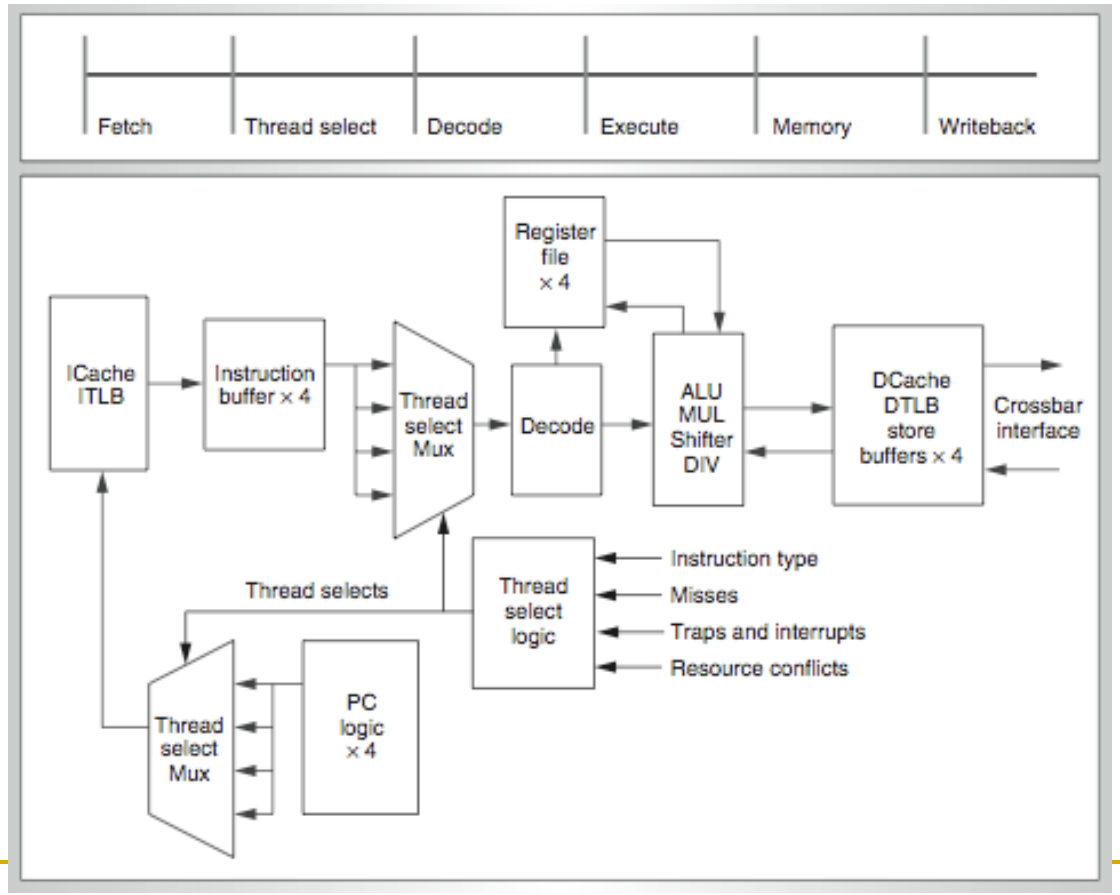
Meet Small: Sun Niagara (UltraSPARC T1)

- 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

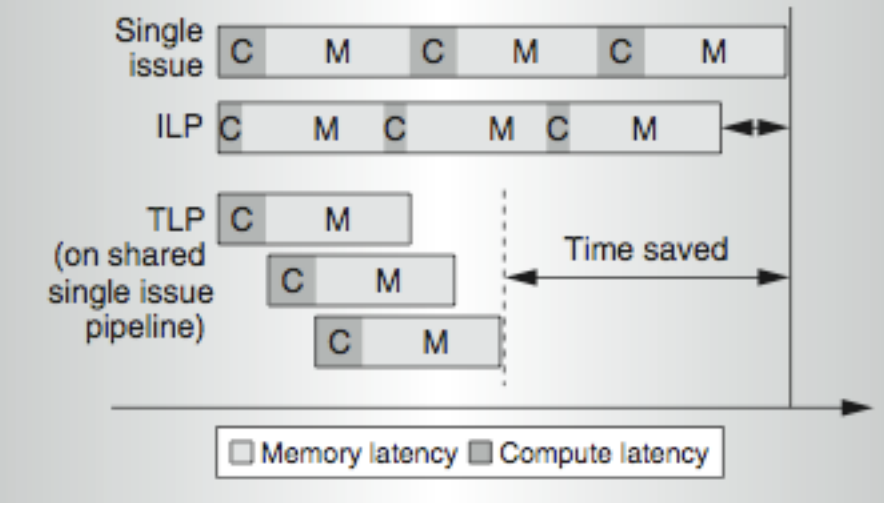


Niagara Design Point

- Designed for commercial applications

Table 1. Commercial server applications.

Benchmark	Application category	Instruction-level parallelism	Thread-level parallelism	Working set	Data sharing
Web99	Web server	Low	High	Large	Low
JBB	Java application server	Low	High	Large	Medium
TPC-C	Transaction processing	Low	High	Large	High
SAP-2T	Enterprise resource planning	Medium	High	Medium	Medium
SAP-3T	Enterprise resource planning	Low	High	Large	High
TPC-H	Decision support system	High	High	Large	Medium

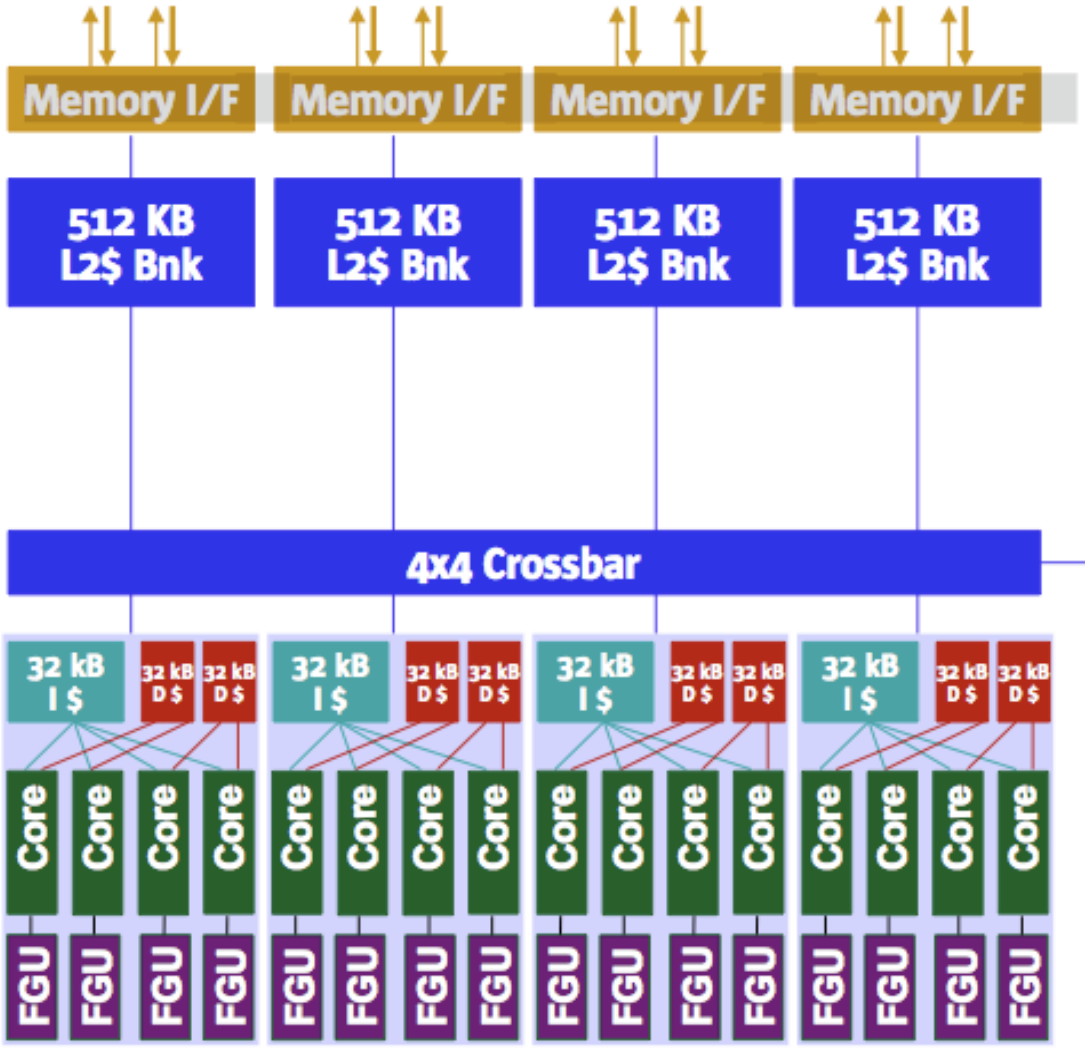


Meet Small, but Larger: Sun ROCK

- Chaudhry et al., "Rock: A High-Performance Sparc CMT Processor," IEEE Micro, 2009.
- Chaudhry et al., "Simultaneous Speculative Threading: A Novel Pipeline Architecture Implemented in Sun's ROCK Processor," ISCA 2009

- Goals:
  - Maximize throughput when threads are available
  - Boost single-thread performance when threads are not available and on cache misses
- Ideas:
  - Runahead on a cache miss → ahead thread executes miss-independent instructions, behind thread executes dependent instructions
  - Branch prediction (gshare)

Sun ROCK



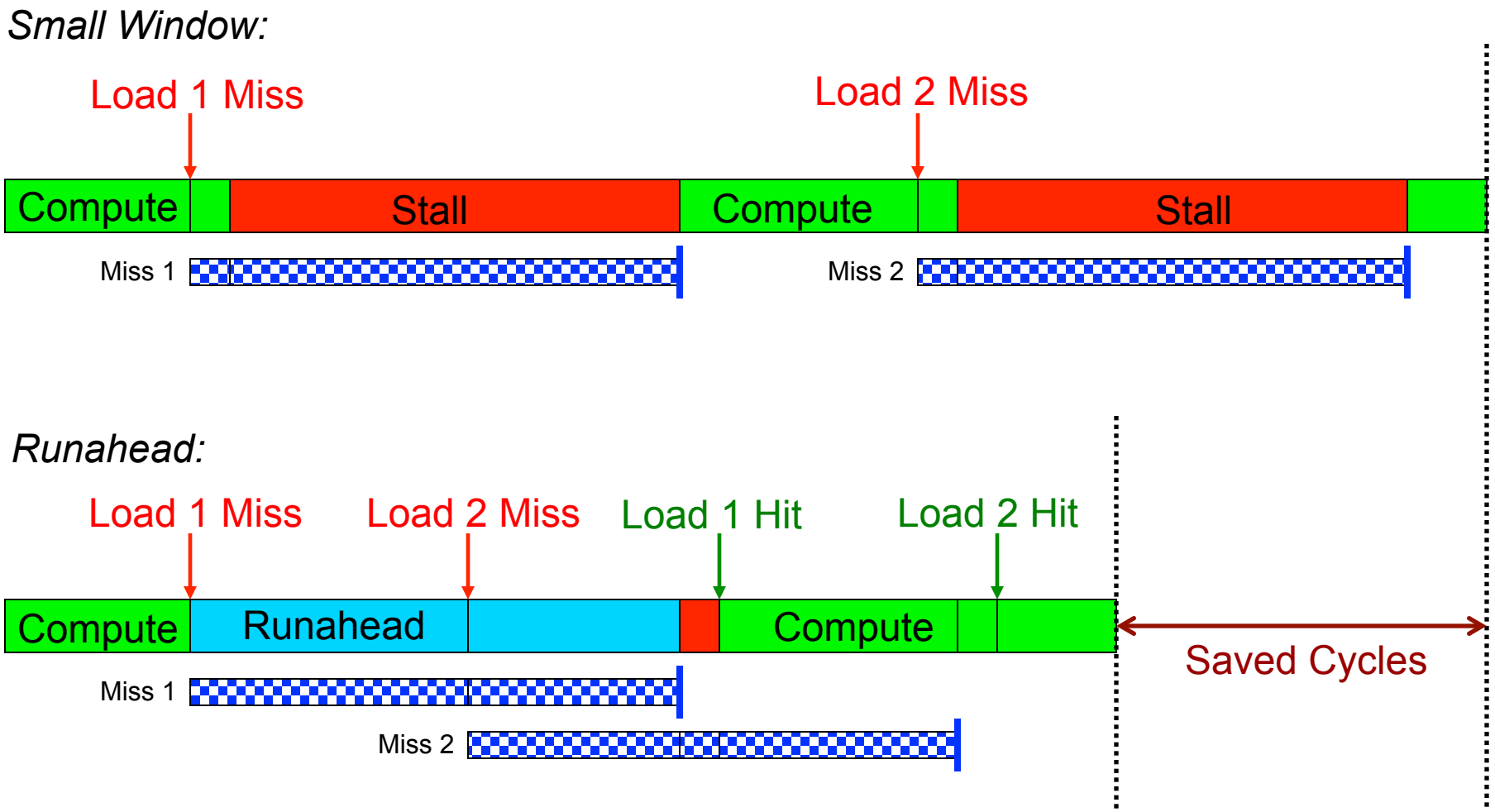
- 16 cores, 2 threads per core (fewer threads than Niagara 2)
- 4 cores share a 32KB instruction cache
- 2 cores share a 32KB data cache
- 2MB L2 cache (smaller than Niagara 2)

Runahead Execution (I)

- A simple pre-execution method for prefetching purposes
- Mutlu et al., "Runahead Execution: An Alternative to Very Large Instruction Windows for Out-of-order Processors," HPCA 2003, IEEE Micro 2003.

- When the oldest instruction is a long-latency cache miss:
  - Checkpoint architectural state and enter runahead mode
- In runahead mode:
  - Speculatively pre-execute instructions
  - The purpose of pre-execution is to generate prefetches
  - L2-miss dependent instructions are marked INV and dropped
- Runahead mode ends when the original miss returns
  - Checkpoint is restored and normal execution resumes

Runahead Execution (II)

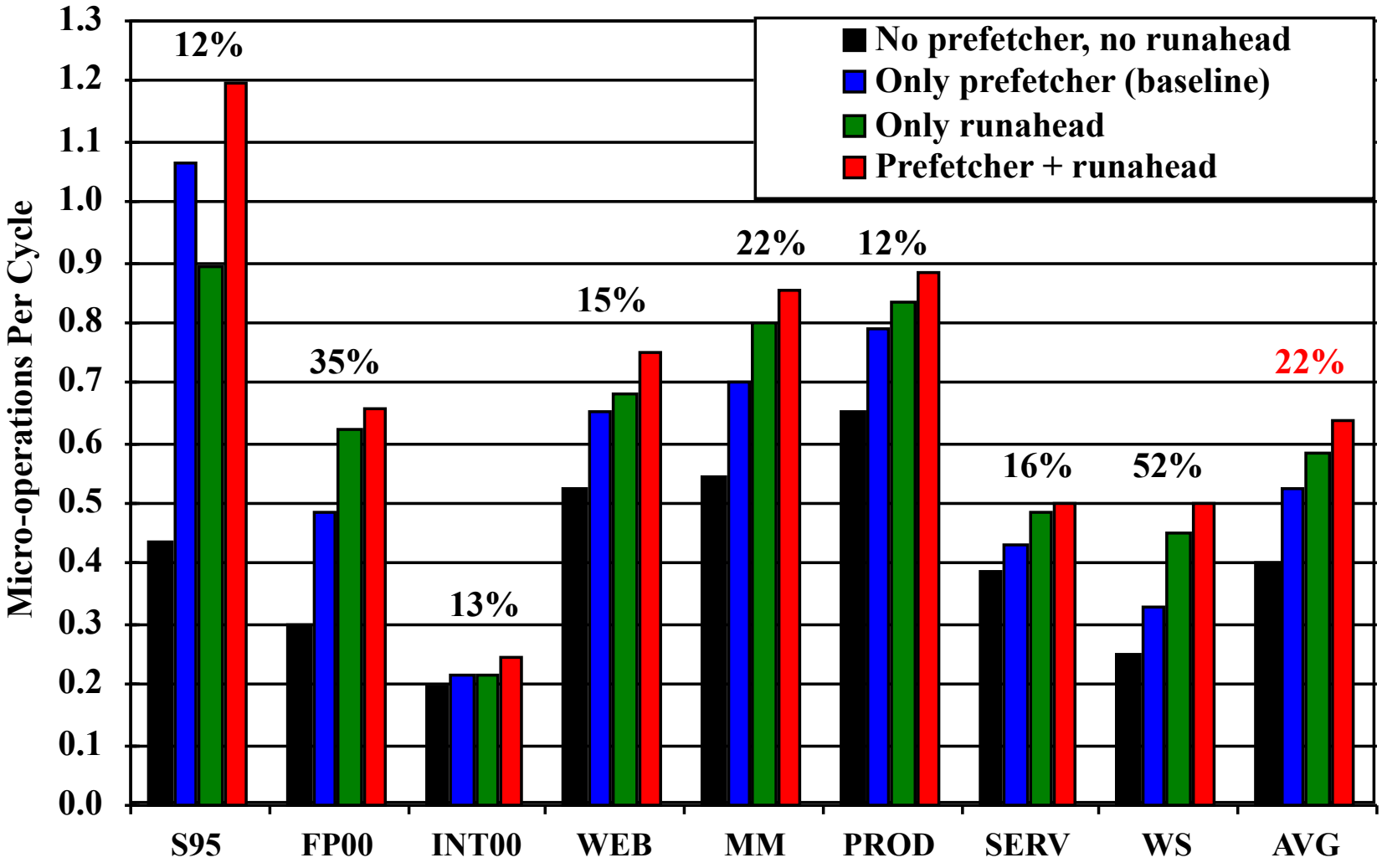


Runahead Execution (III)

- Advantages
  + Very accurate prefetches for data/instructions (all cache levels)
    + Follows the program path
  + Simple to implement, most of the hardware is already built in
- Disadvantages
  -- Extra executed instructions
- Limitations
  -- Limited by branch prediction accuracy
  -- Cannot prefetch dependent cache misses. Solution?
  -- Effectiveness limited by available Memory Level Parallelism

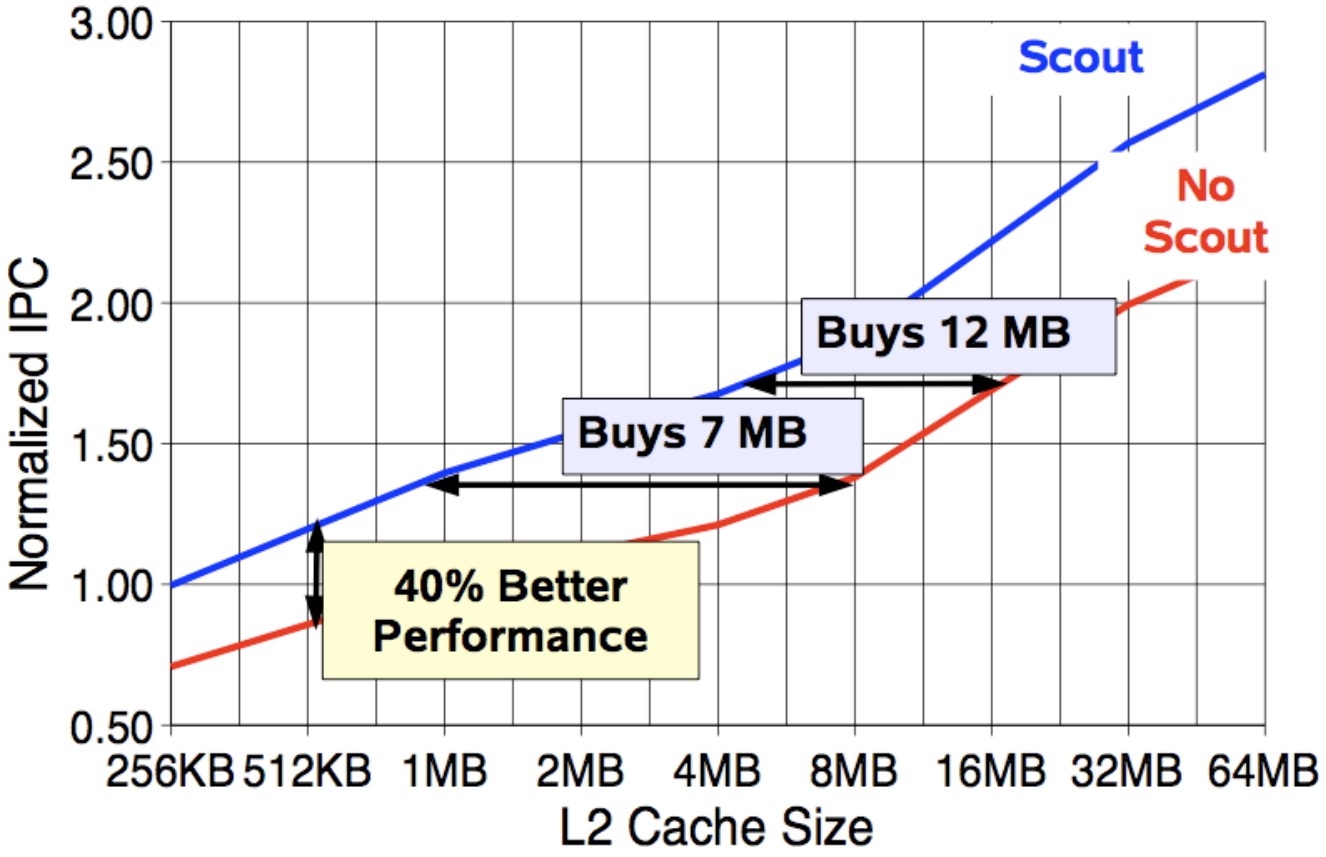
- Mutlu et al., "Efficient Runahead Execution: Power-Efficient Memory Latency Tolerance," IEEE Micro Jan/Feb 2006.
- Implemented in IBM POWER6, Sun ROCK

Performance of Runahead Execution



More Powerful Cores in Sun ROCK

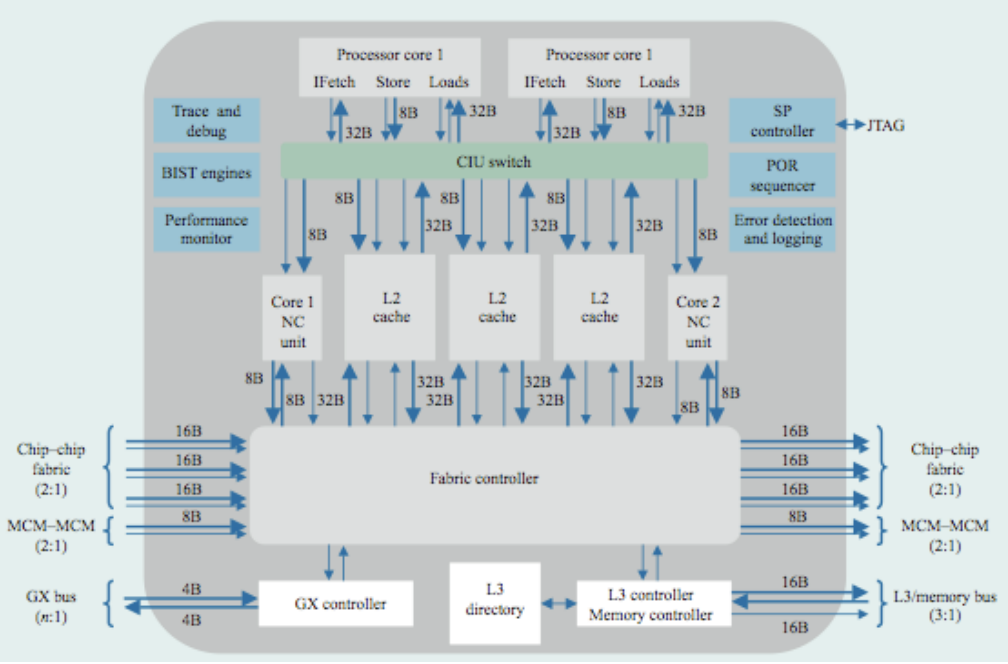
- Chaudhry talk, Aug 2008.

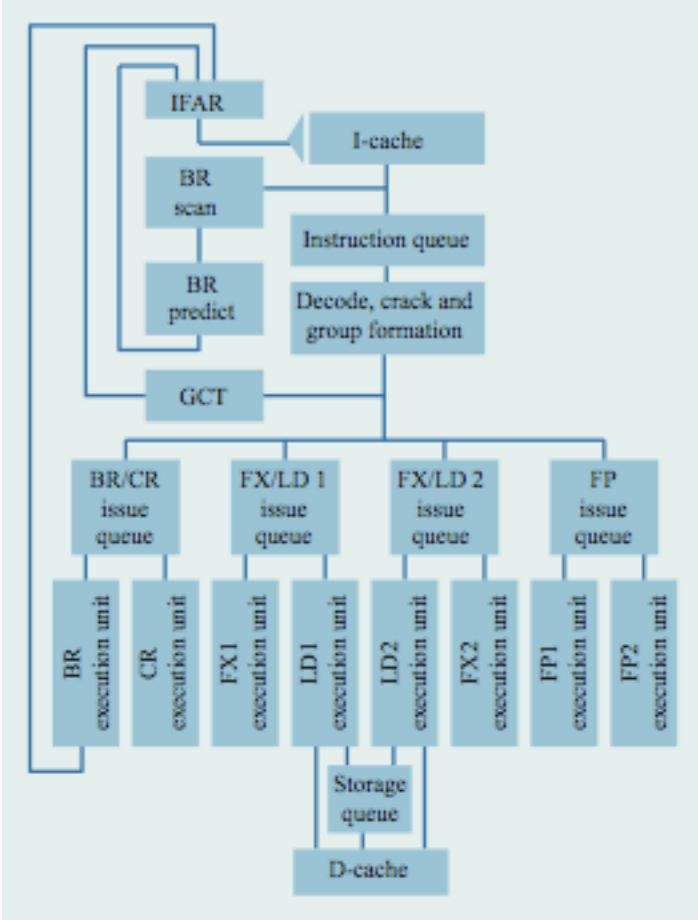


Meet Large: IBM POWER4

- Tendler et al., "POWER4 system microarchitecture," IBM J R&D, 2002.

- Another symmetric multi-core chip…
- But, fewer and more powerful cores





IBM POWER4

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

IBM POWER5

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

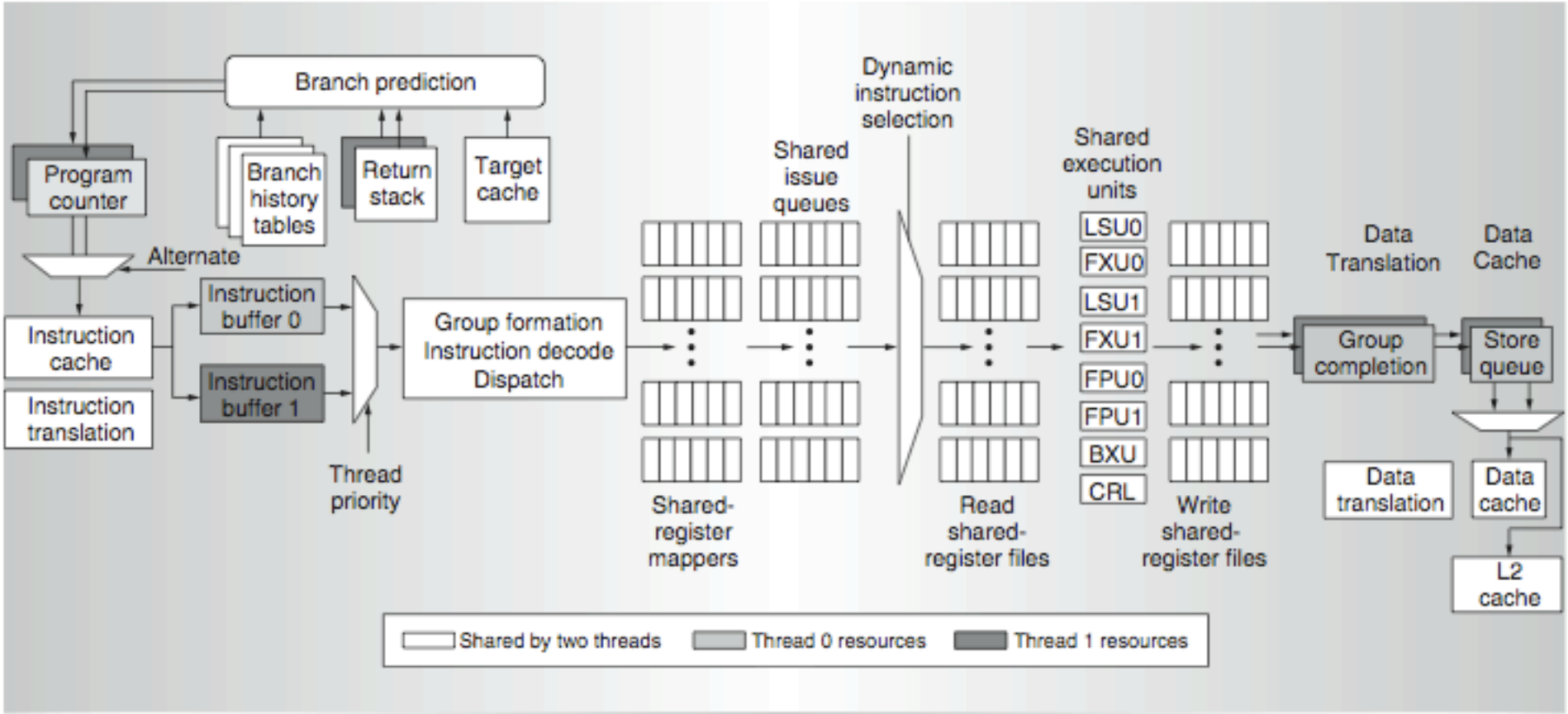
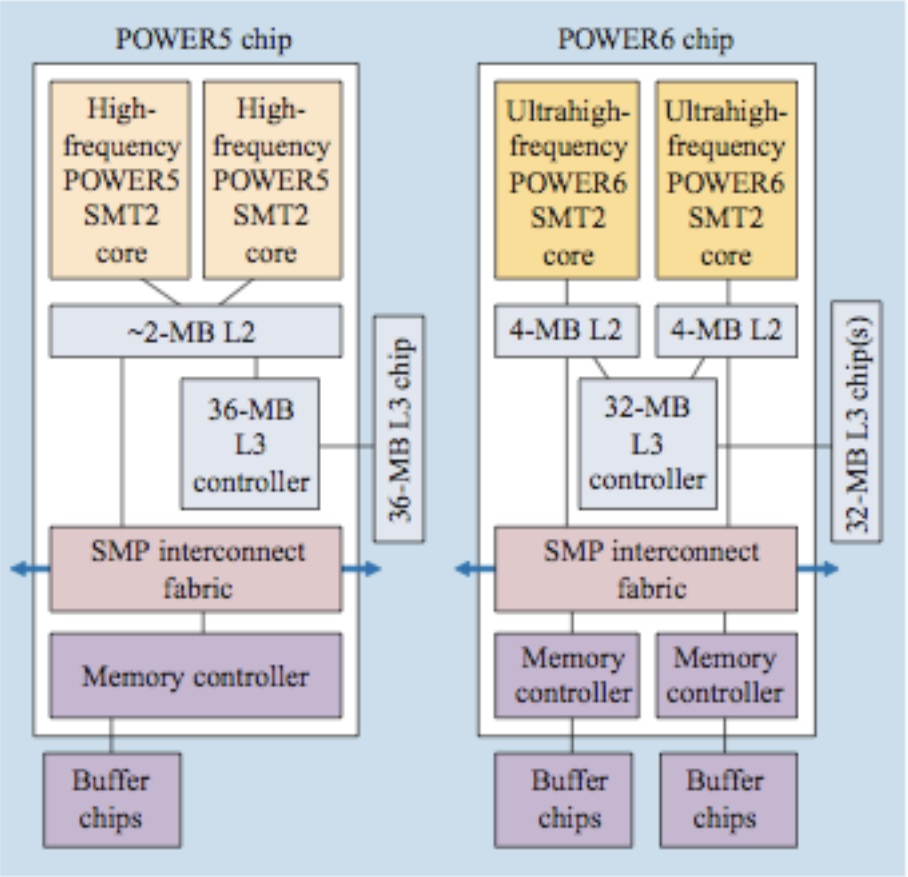


Figure 4. Power5 instruction data flow (BXU = branch execution unit and CRL = condition register logical execution unit).

Meet Large, but Smaller: IBM POWER6

- Le et al., "IBM POWER6 microarchitecture," IBM J R&D, 2007.

- 2 cores, in order, high frequency (4.7 GHz)
- 8 wide fetch
- Simultaneous multithreading in each core
- Runahead execution in each core
  - Similar to Sun ROCK



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?

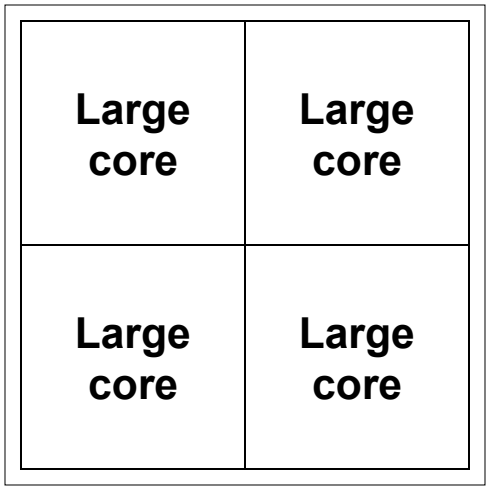
Performance vs. Parallelism

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



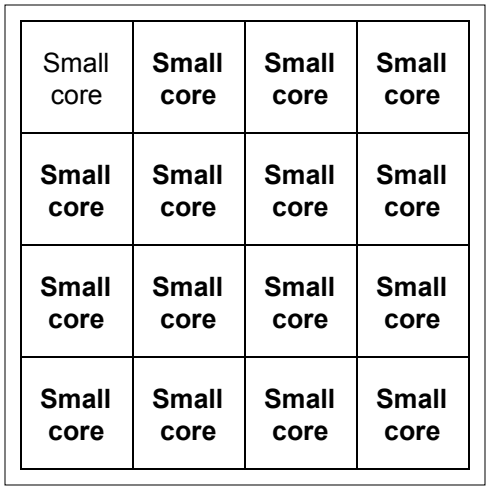
"Tile-Large"

- 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



"Tile-Small"

- Tile many small cores
- Sun Niagara, Intel Larrabee, Tilera 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

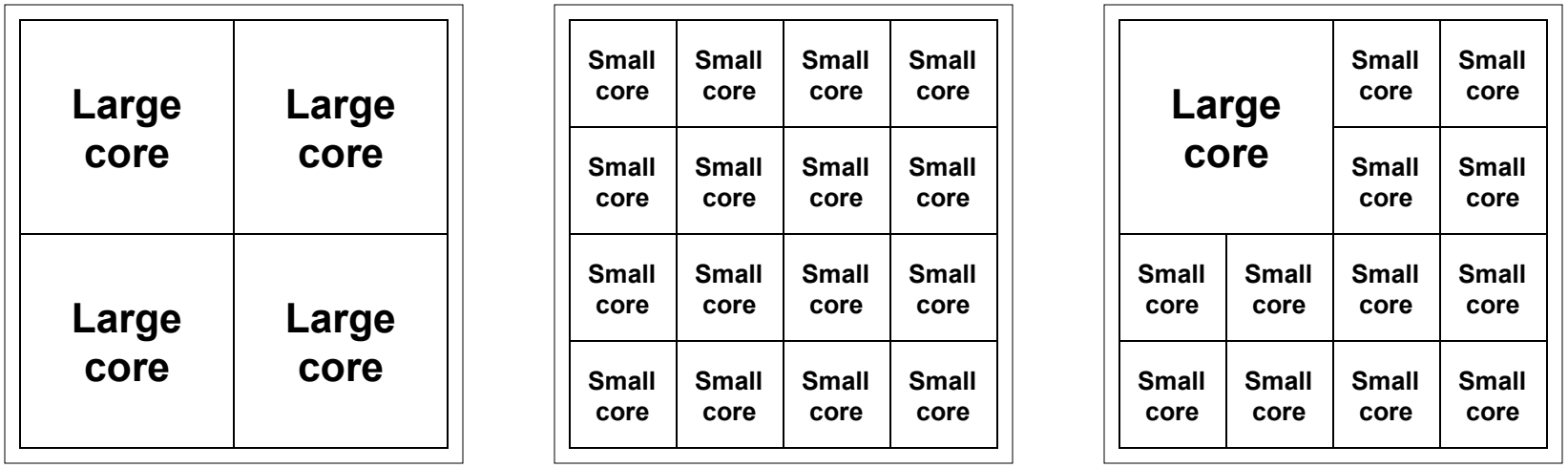
  + High throughput on the parallel part (16 units)

  - 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)



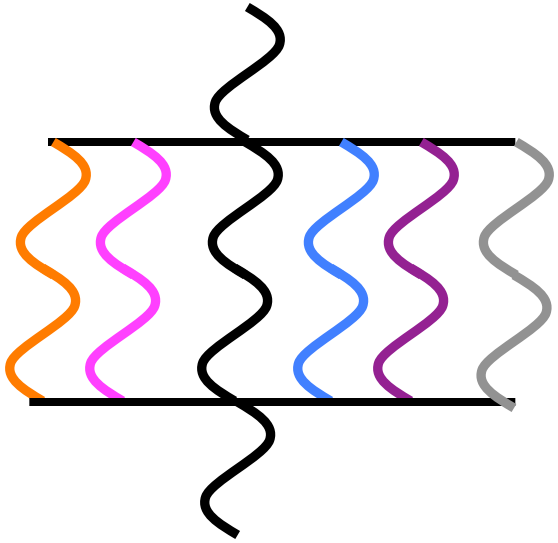
- Provide one large core and many small cores

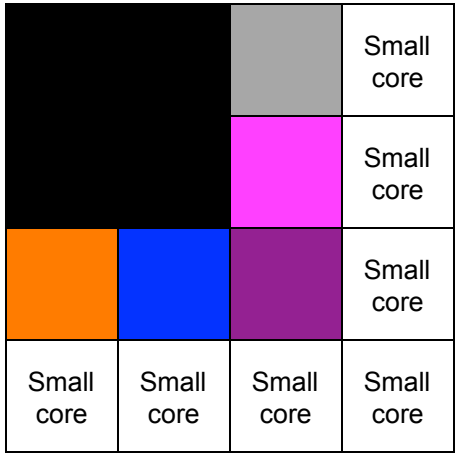
+ Accelerate serial part using the large core (2 units)

+ Execute parallel part on small cores and large core for high throughput (12+2 units)

Accelerating Serial Bottlenecks

Single thread → Large core





ACMP Approach

Performance vs. Parallelism

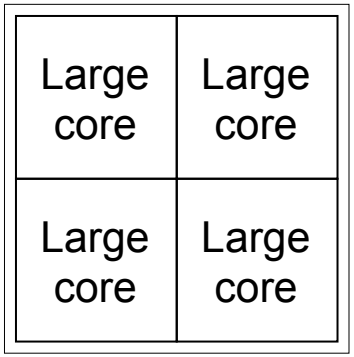
Assumptions:

1. *Small cores takes an area budget of 1 and has performance of 1*

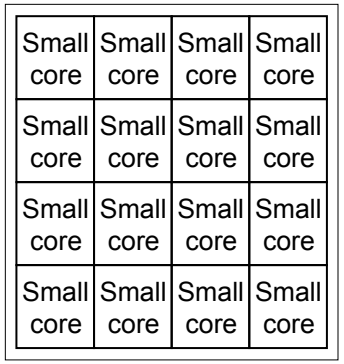
2. *Large core takes an area budget of 4 and has performance of 2*

ACMP Performance vs. Parallelism

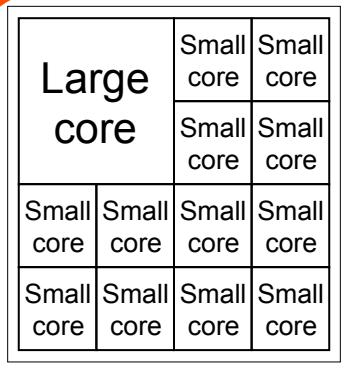
Area-budget = 16 small cores



"Tile-Large"



"Tile-Small"



ACMP

	"Tile-Large"	"Tile-Small"	ACMP
Large Cores	4	0	1
Small Cores	0	16	12
Serial Performance	2	1	2
Parallel Throughput	2 x 4 = 8	1 x 16 = 16	1x2 + 1x12 = 14

106

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

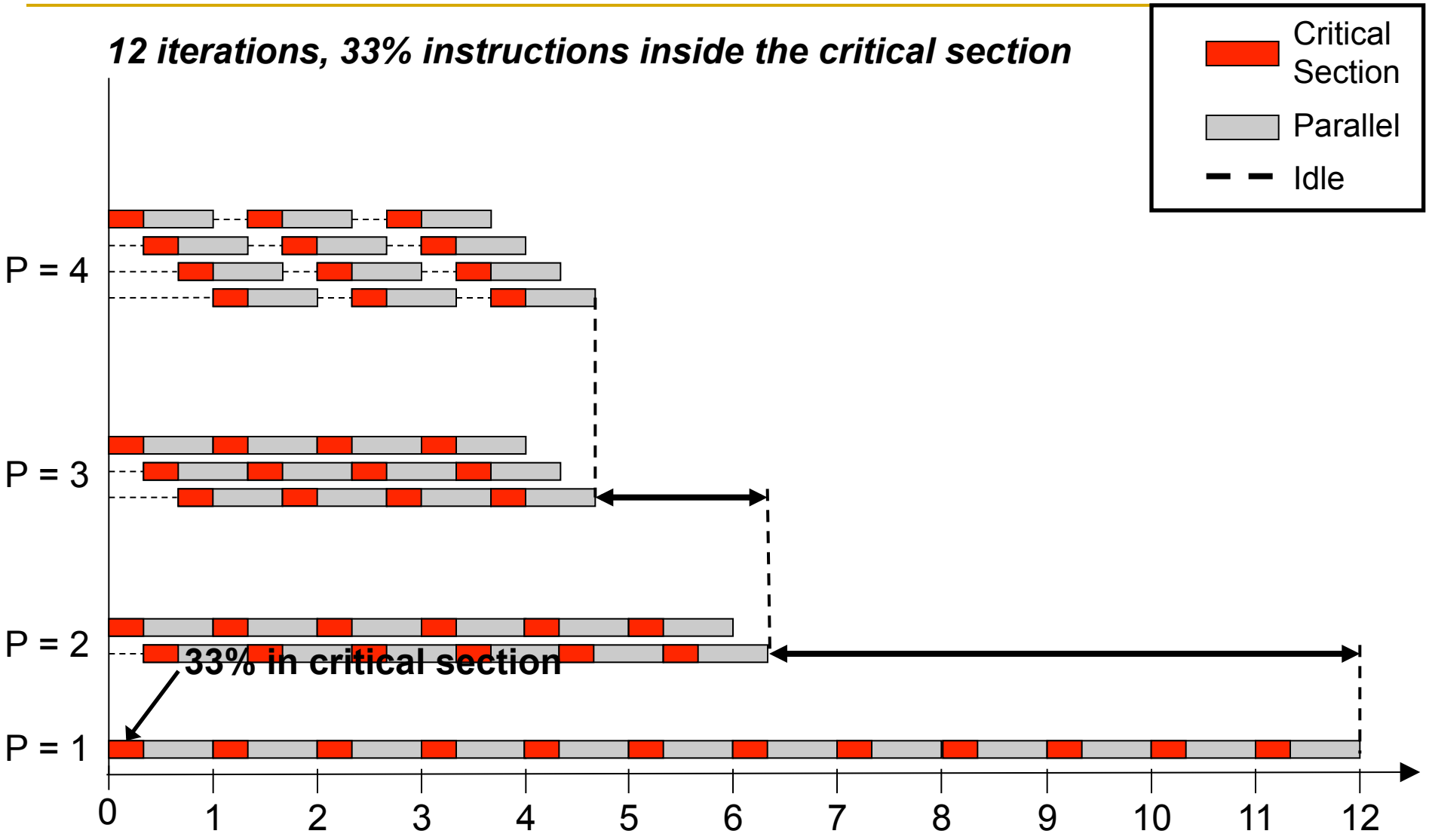
- Examples:
  - Critical sections that are contended
  - Parallel stages that take longer than others to execute

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

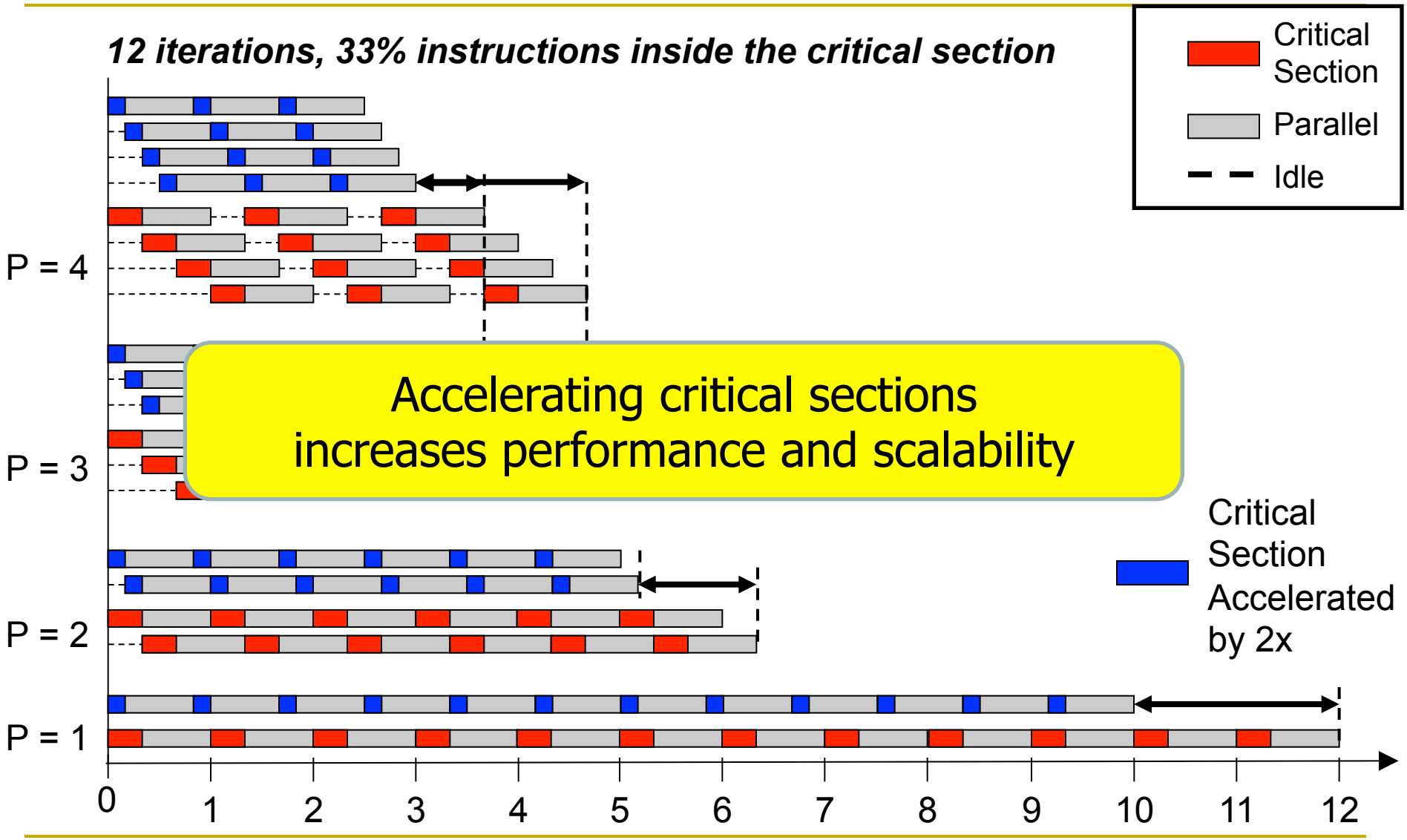
Accelerated Critical Sections

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***)

Contention for Critical Sections

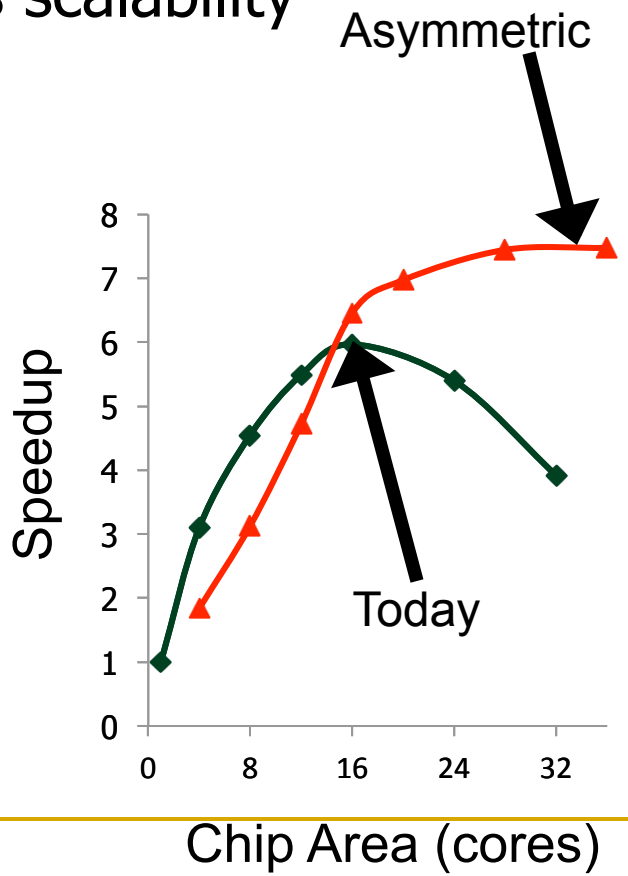


Contention for Critical Sections



Impact of Critical Sections on Scalability

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



MySQL (oltp-1)

A Case for Asymmetry

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

An Example: Accelerated Critical Sections

- Idea: HW/SW ships critical sections to a large, powerful core in an asymmetric multi-core architecture

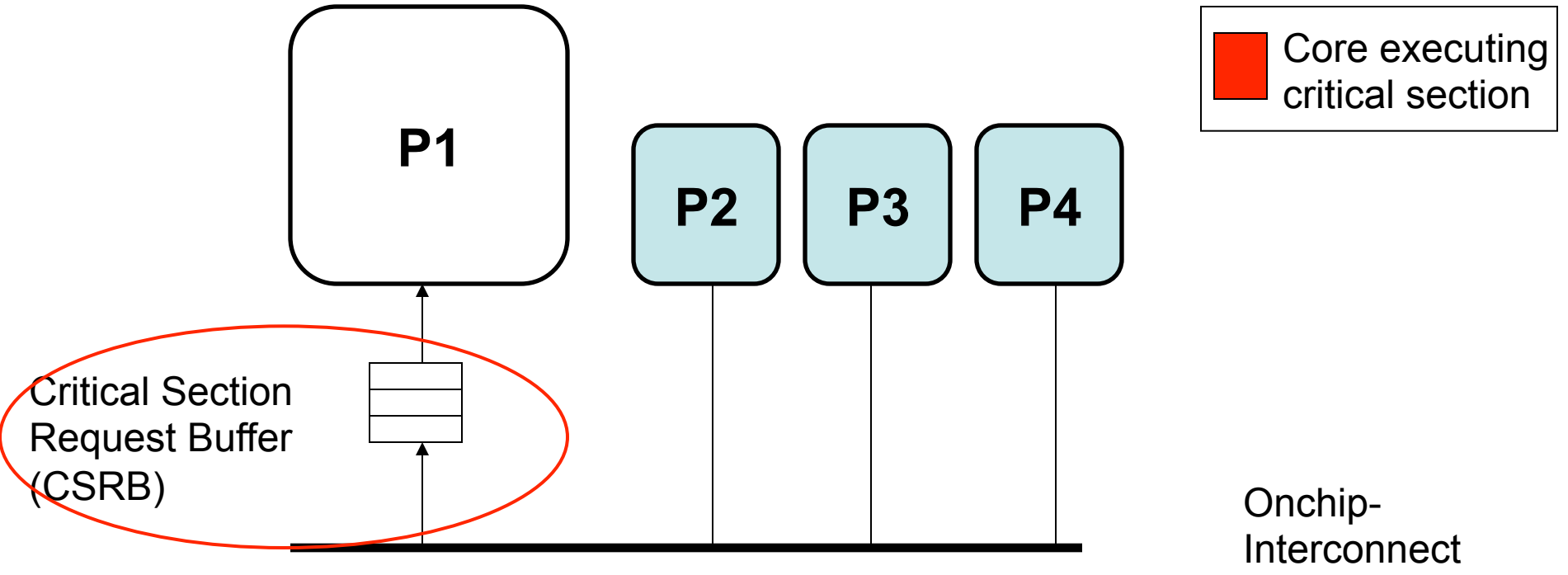
- Benefit:
  - Reduces serialization due to contended locks
  - Reduces the performance impact of hard-to-parallelize sections
  - Programmer does not need to (heavily) optimize parallel code → fewer bugs, improved productivity

- Suleman et al., "Accelerating Critical Section Execution with Asymmetric Multi-Core Architectures," ASPLOS 2009, IEEE Micro Top Picks 2010.
- Suleman et al., "Data Marshaling for Multi-Core Architectures," ISCA 2010, IEEE Micro Top Picks 2011.

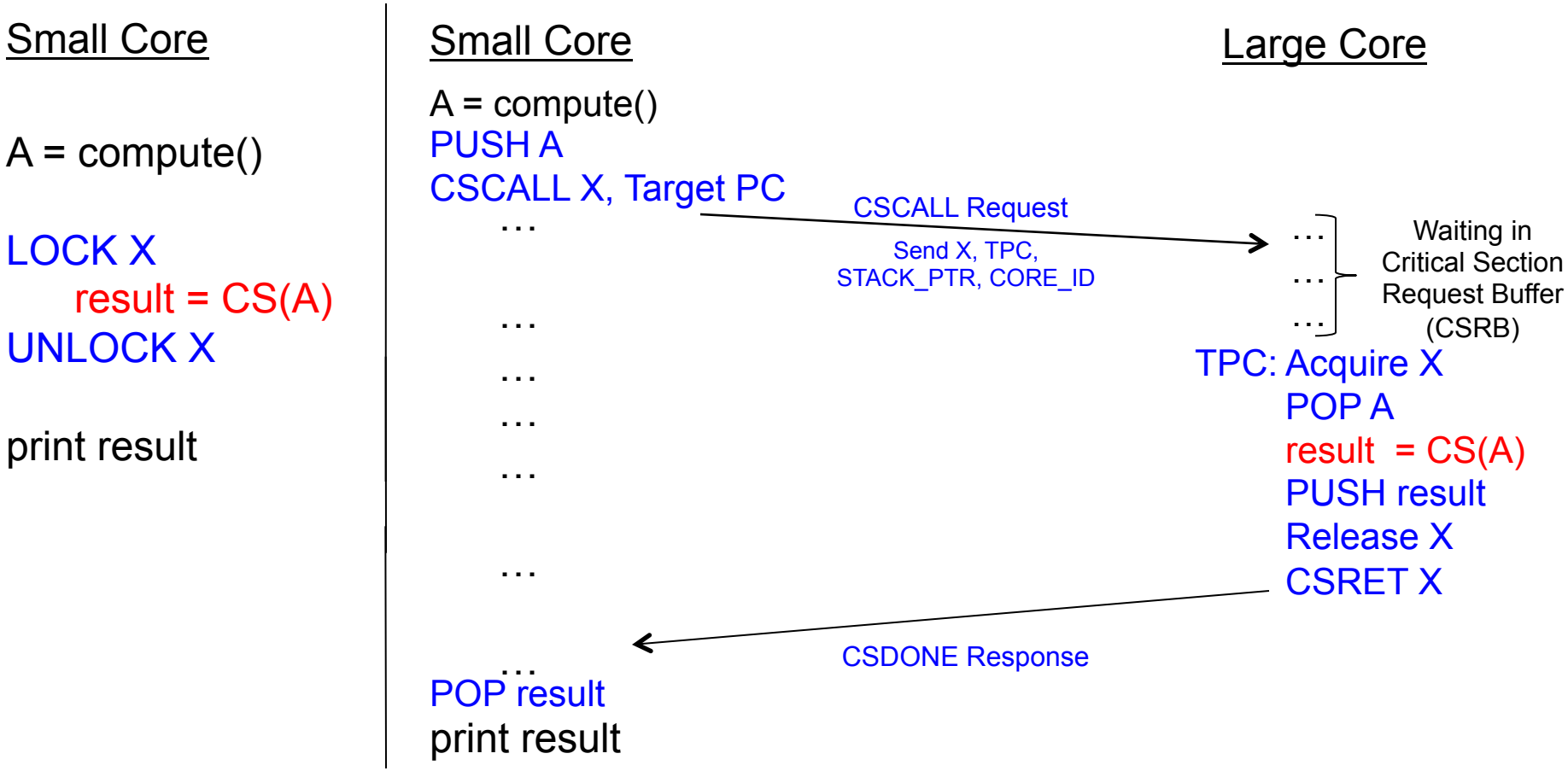
Accelerated Critical Sections

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

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



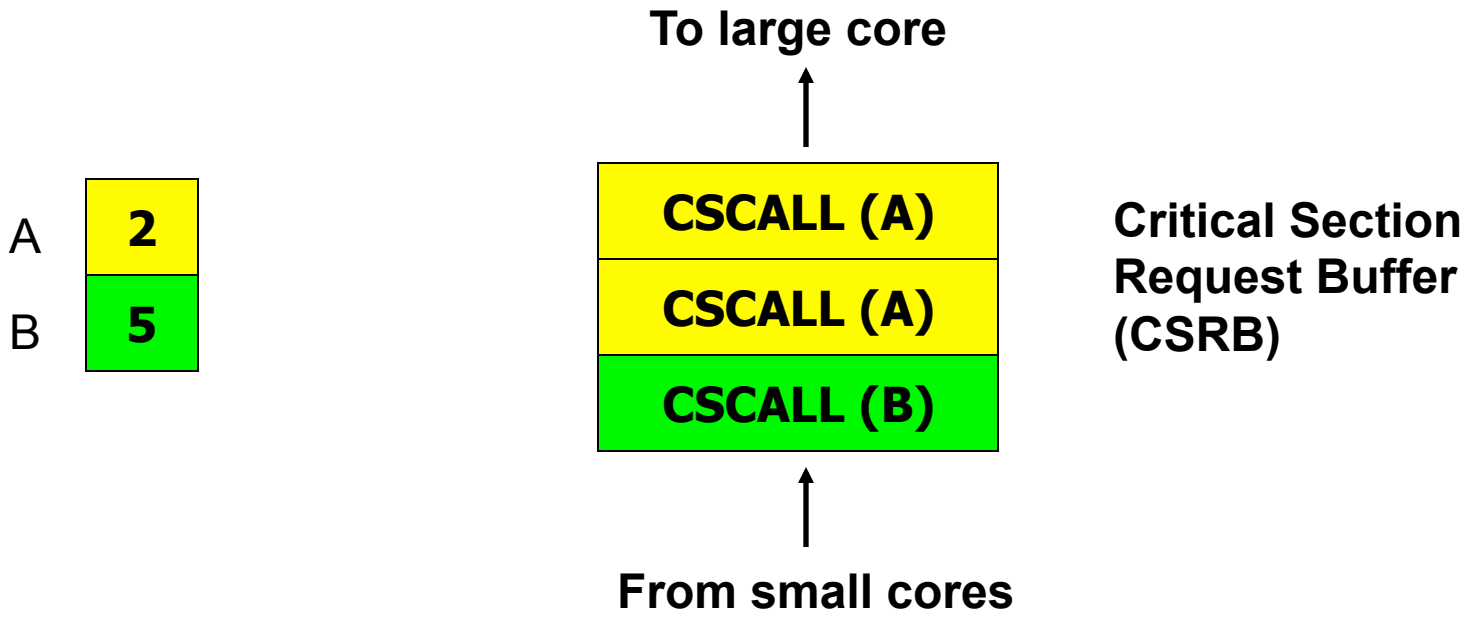
Accelerated Critical Sections (ACS)



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

False Serialization

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



ACS Performance Tradeoffs

- Pluses

  + Faster critical section execution

  + Shared locks stay in one place: better lock locality

  + Shared data stays in large core's (large) caches: better shared data locality, less ping-ponging

- Minuses

  - Large core dedicated for critical sections: reduced parallel throughput

  - CSCALL and CSDONE control transfer overhead

  - Thread-private data needs to be transferred to large core: worse private data locality

ACS Performance Tradeoffs

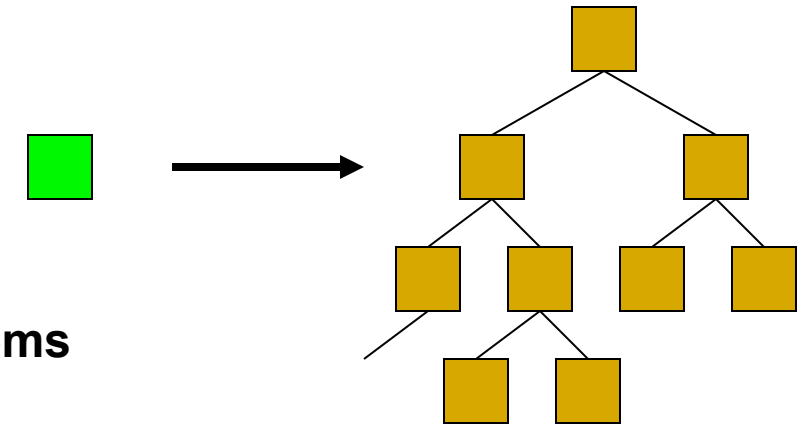
- ***Fewer parallel threads vs. accelerated critical sections***
  - Accelerating critical sections offsets loss in throughput
  - As the number of cores (threads) on chip increase:
    - Fractional loss in parallel performance decreases
    - Increased contention for critical sections makes acceleration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

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

- ***More cache misses for private data vs. fewer misses for shared data***

Cache Misses for Private Data

PriorityHeap.insert(NewSubProblems)



Private Data:
NewSubProblems

Shared Data:
The priority heap

Puzzle Benchmark

ACS Performance Tradeoffs

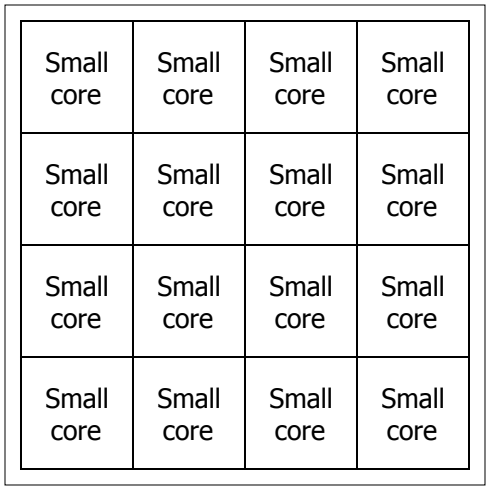
- ***Fewer parallel threads vs. accelerated critical sections***
  - Accelerating critical sections offsets loss in throughput
  - As the number of cores (threads) on chip increase:
    - Fractional loss in parallel performance decreases
    - Increased contention for critical sections makes acceleration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

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

- ***More cache misses for private data vs. fewer misses for shared data***
  - Cache misses reduce if shared data > private data

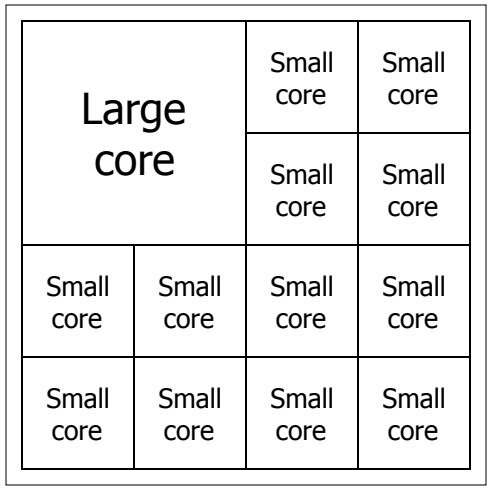
This problem can be solved

ACS Comparison Points



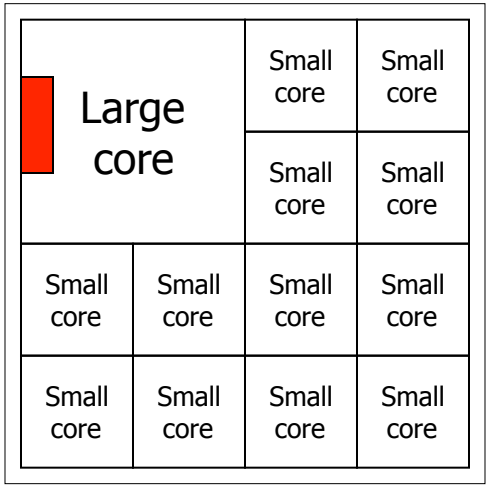
SCMP

- Conventional locking



ACMP

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



ACS

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

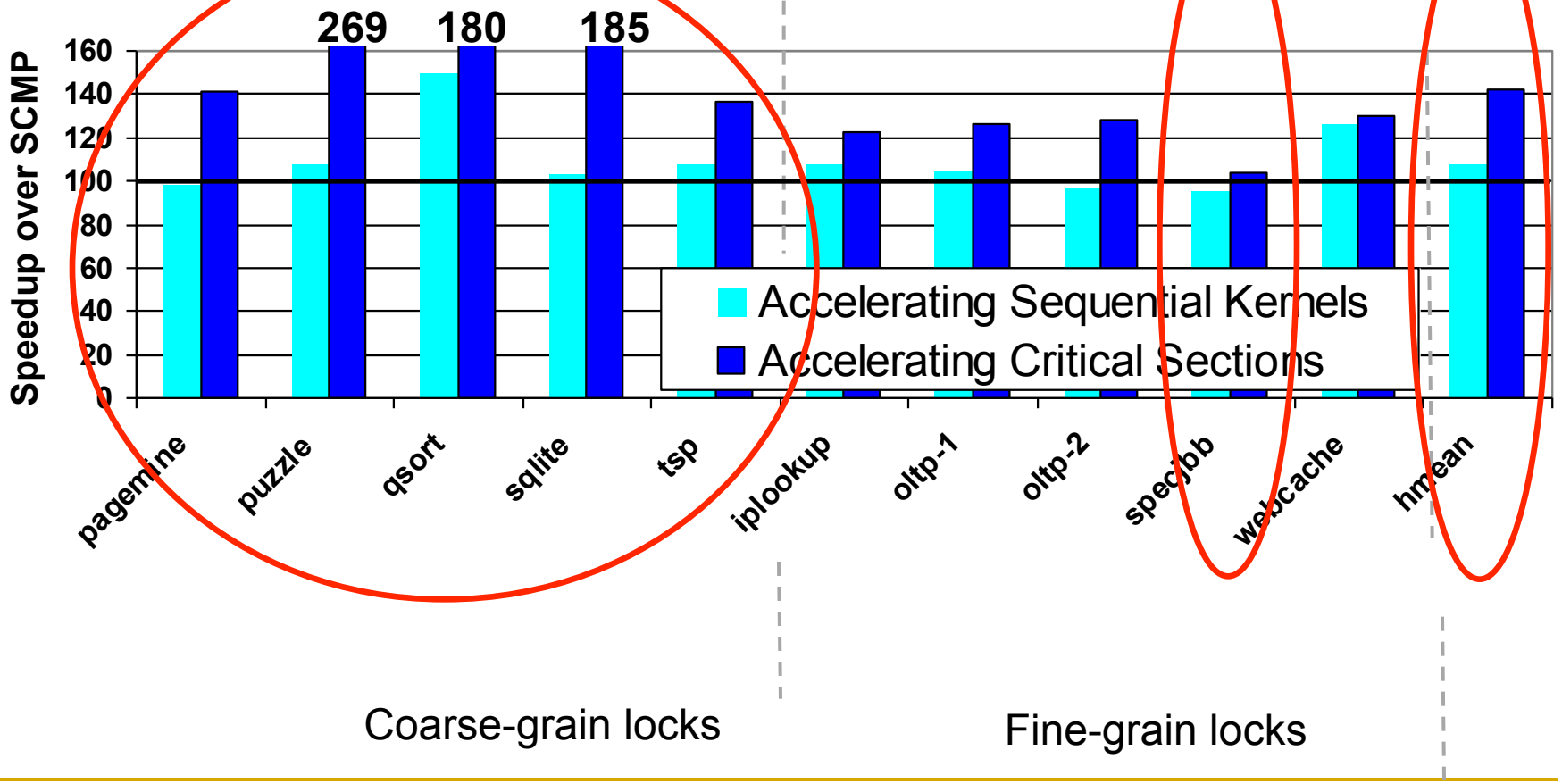
Accelerated Critical Sections: Methodology

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

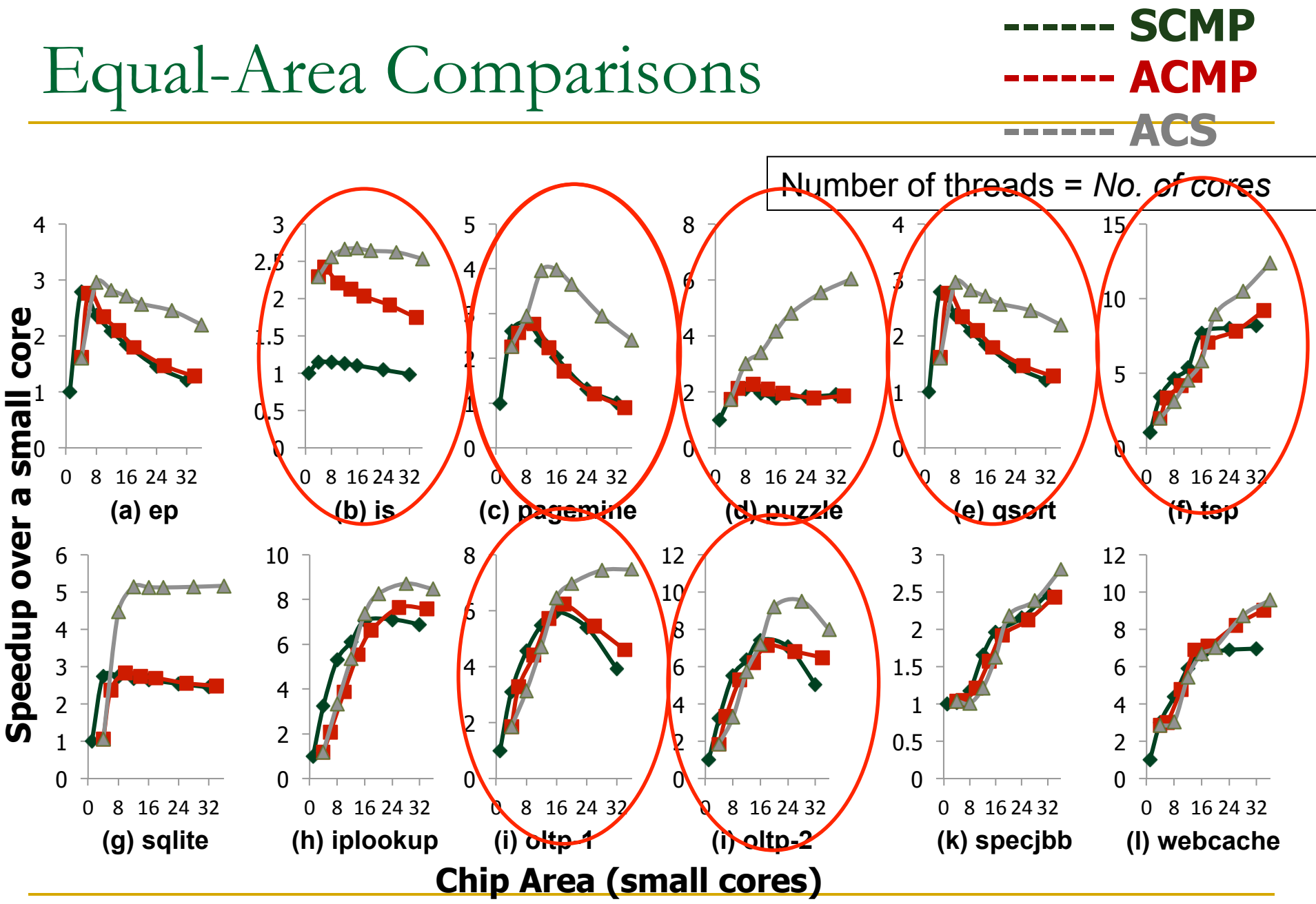
ACS Performance

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

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



Equal-Area Comparisons



ACS Summary

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

Bottleneck Identification and Scheduling

Jose A. Joao, M. Aater Suleman, Onur Mutlu, and Yale N. Patt,
"Bottleneck Identification and Scheduling in Multithreaded Applications"
Proceedings of the
17th International Conference on Architectural Support for Programming Languages and Operating Systems (***ASPLOS***), London, UK, March 2012.

BIS Summary

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

Bottlenecks in Multithreaded Applications

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

Examples:

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

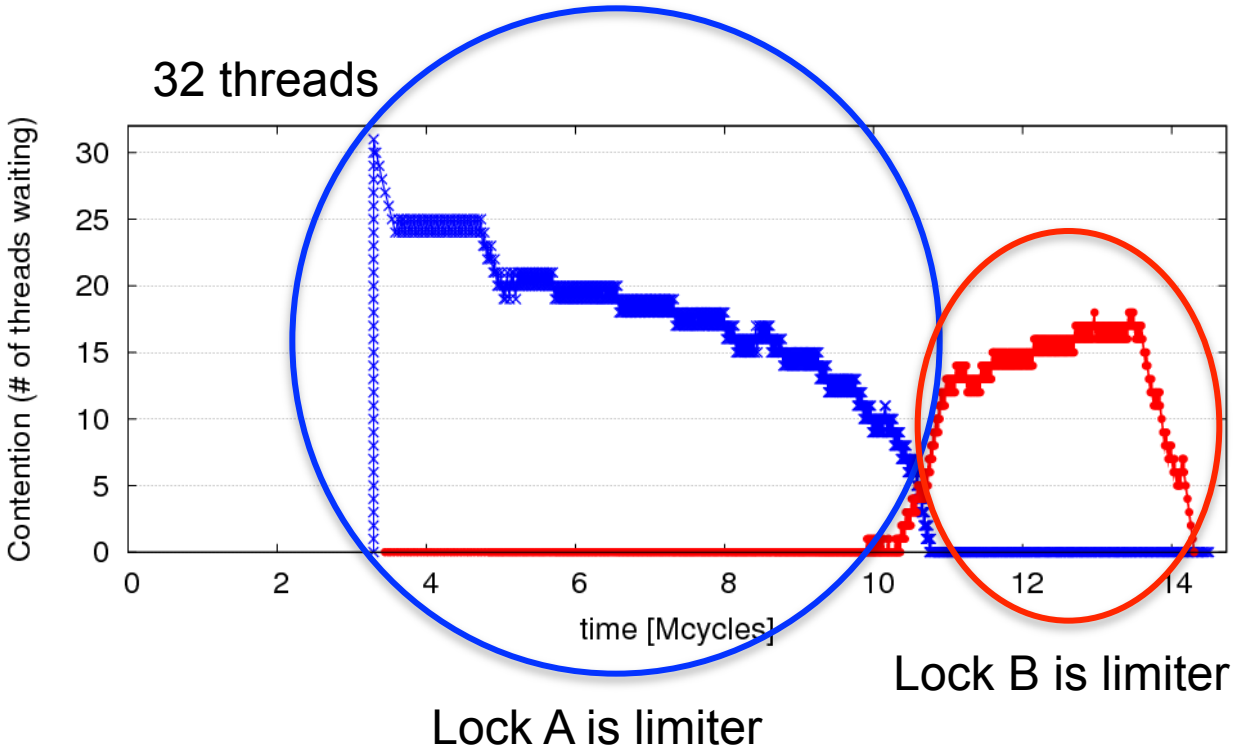
Observation: Limiting Bottlenecks Change Over Time

A=full linked list; B=empty linked list

repeat

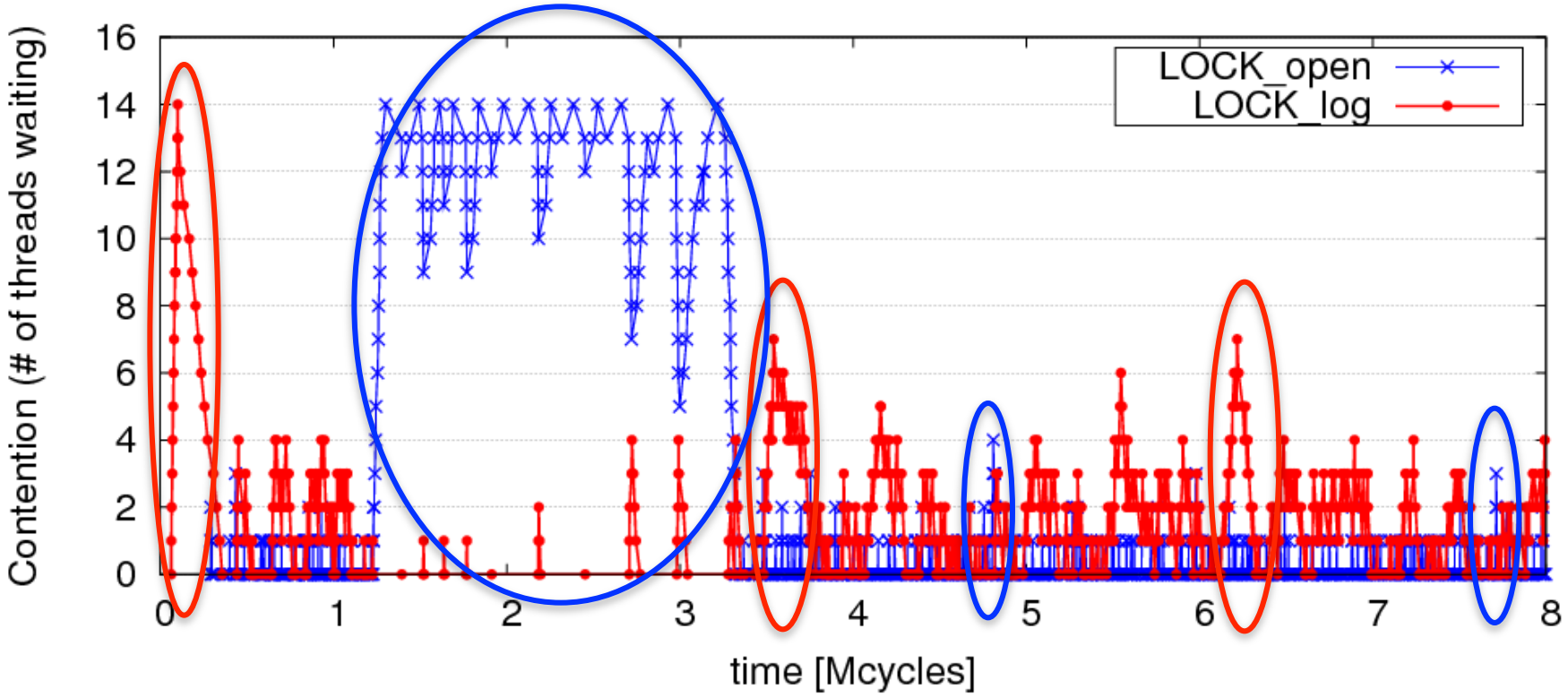
```
Lock A
        Traverse list A
        Remove X from A
Unlock A
Compute on X
Lock B
        Traverse list B
        Insert X into B
Unlock B
```

until A is empty



Limiting Bottlenecks Do Change on Real Applications

MySQL running Sysbench queries, 16 threads



Previous Work on Bottleneck Acceleration

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

No previous work

→ can accelerate all types of bottlenecks or

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

Our goal:

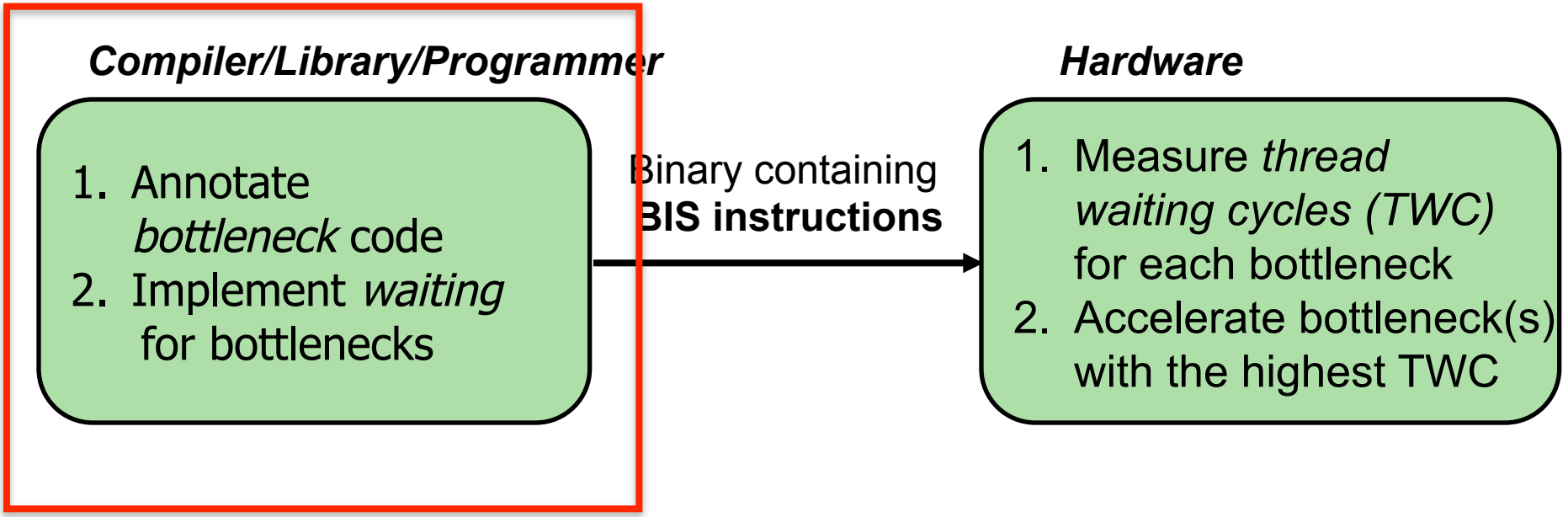
general mechanism to identify and accelerate performance-limiting bottlenecks of any type

Bottleneck Identification and Scheduling (BIS)

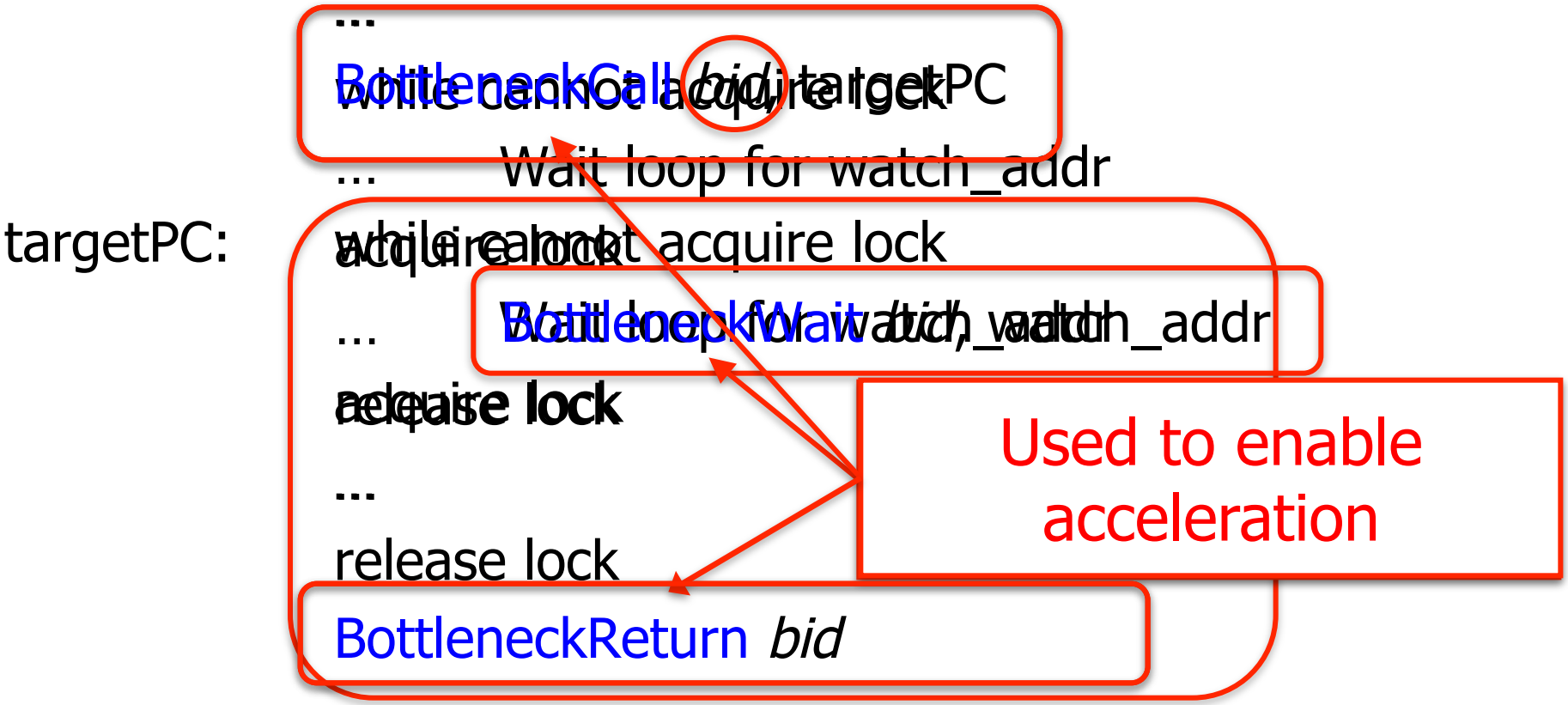
- Key insight:
  - Thread waiting reduces parallelism and is likely to reduce performance
  - Code causing the most thread waiting → likely critical path

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

Bottleneck Identification and Scheduling (BIS)



Critical Sections: Code Modifications



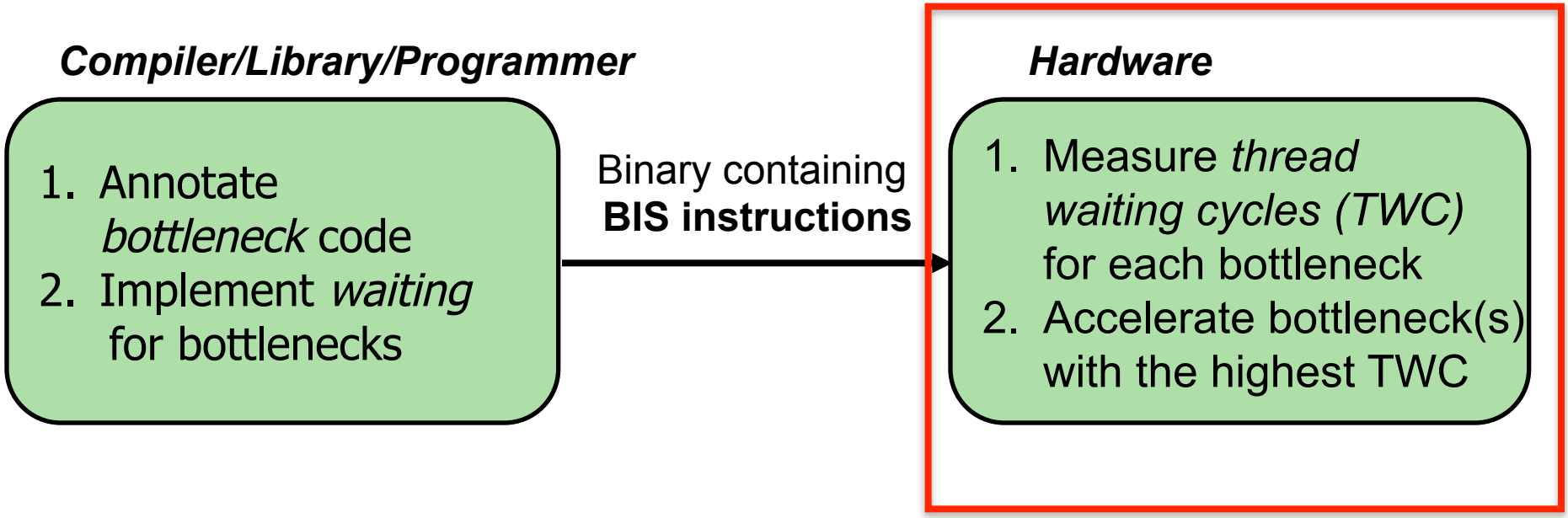
Barriers: Code Modifications

```
        ...
        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

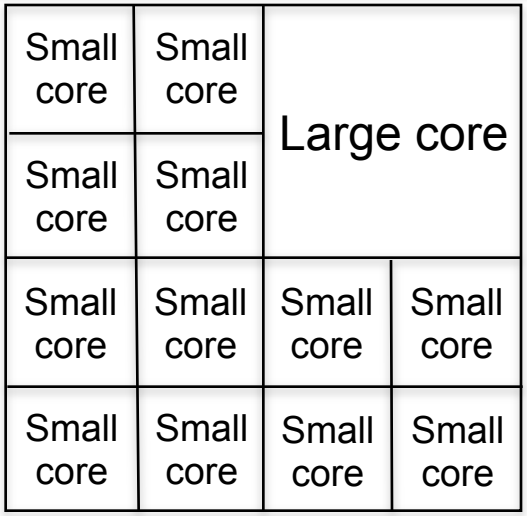
```
                BottleneckCall bid, targetPC
                ...
targetPC:       while not done
                        while empty queue
                                BottleneckWait prev_bid
                        dequeue work
                        do the work …
                        while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

Bottleneck Identification and Scheduling (BIS)

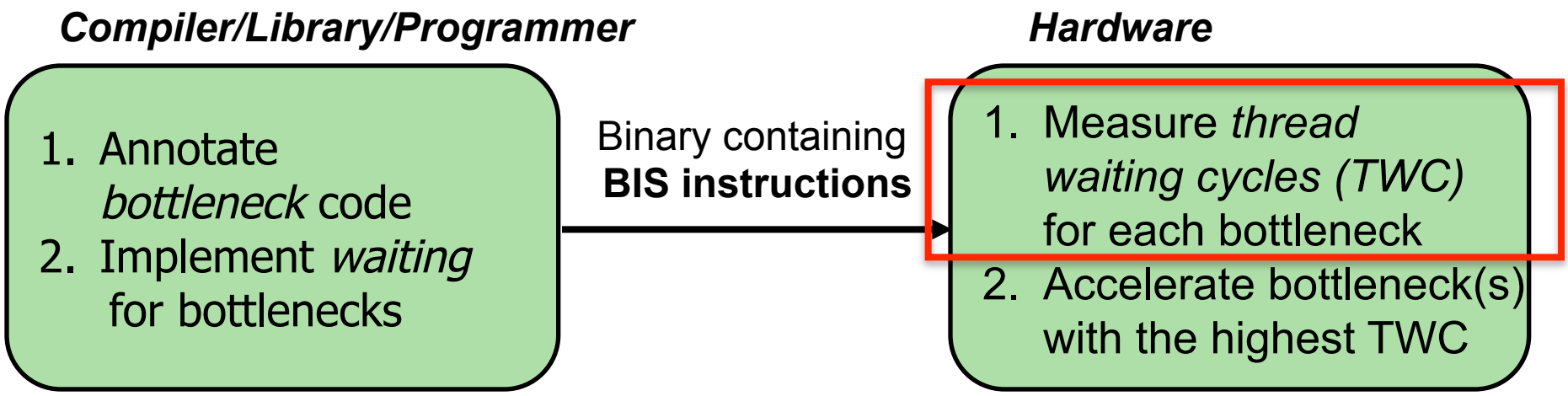


BIS: Hardware Overview

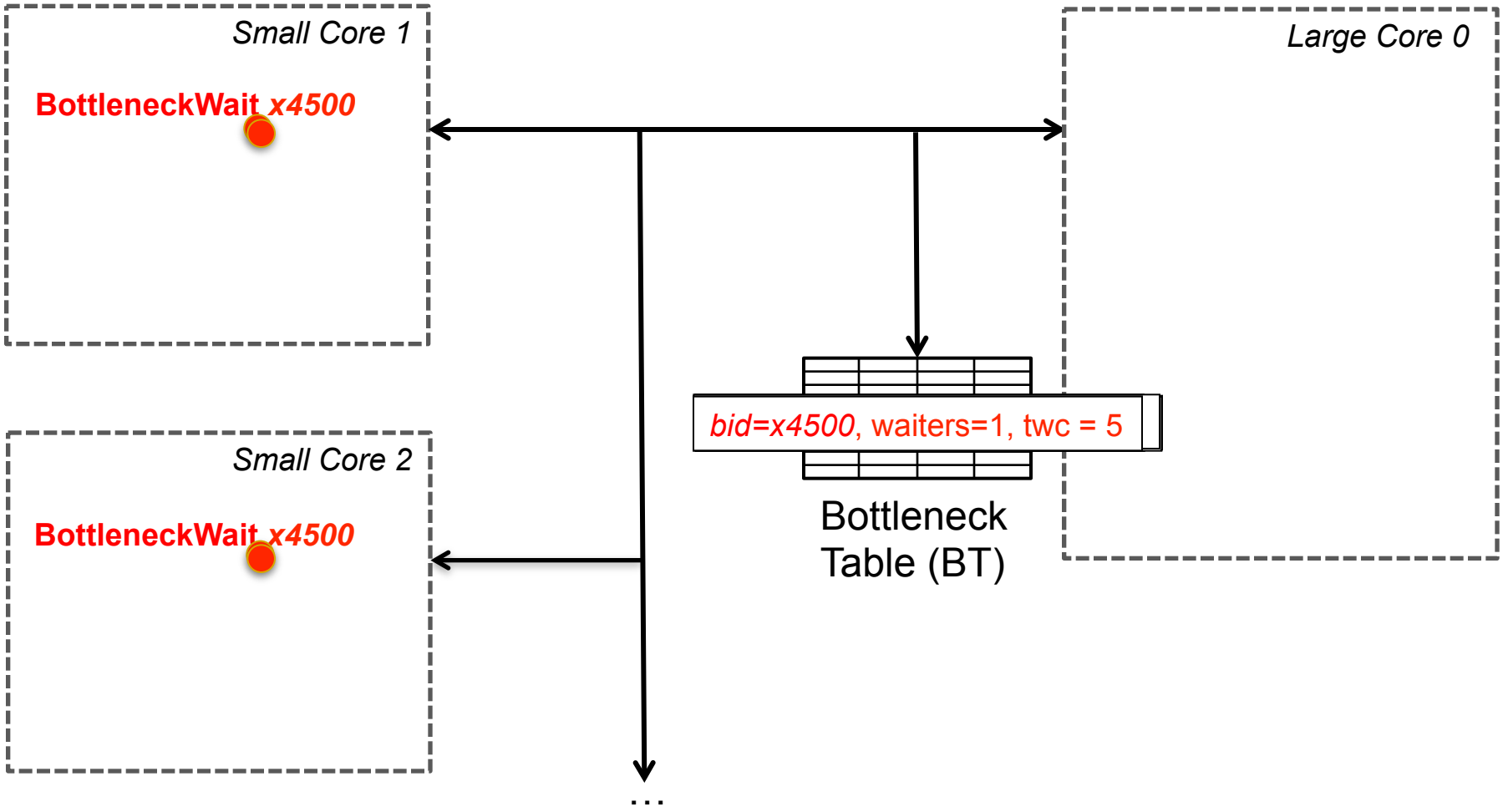
- Performance-limiting bottleneck identification and acceleration are independent tasks
- Acceleration can be accomplished in multiple ways
  - Increasing core frequency/voltage
  - Prioritization in shared resources [Ebrahimi+, MICRO'11]
  - Migration to faster cores in an Asymmetric CMP



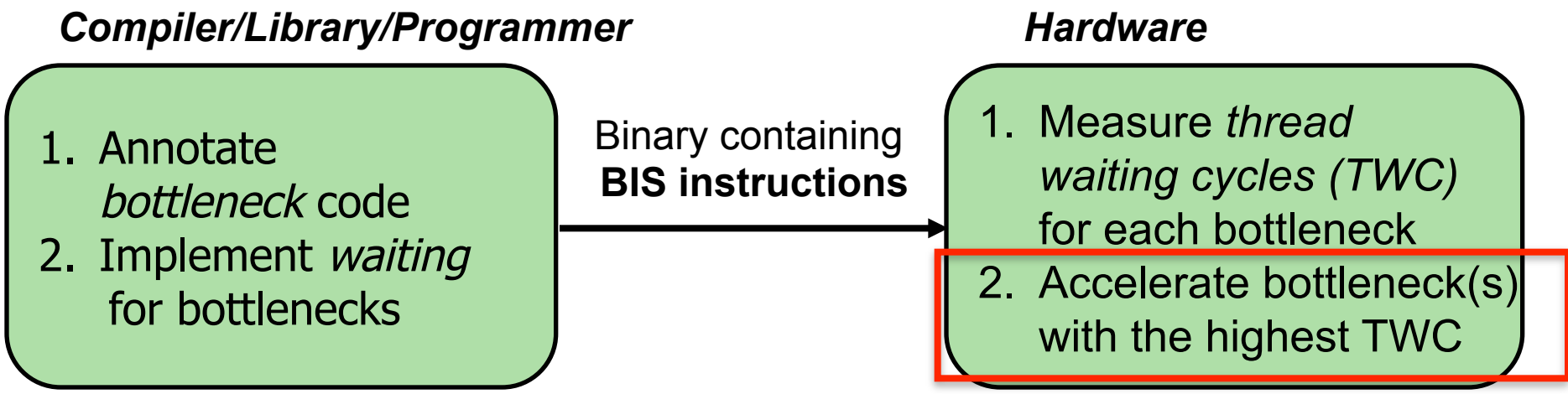
Bottleneck Identification and Scheduling (BIS)



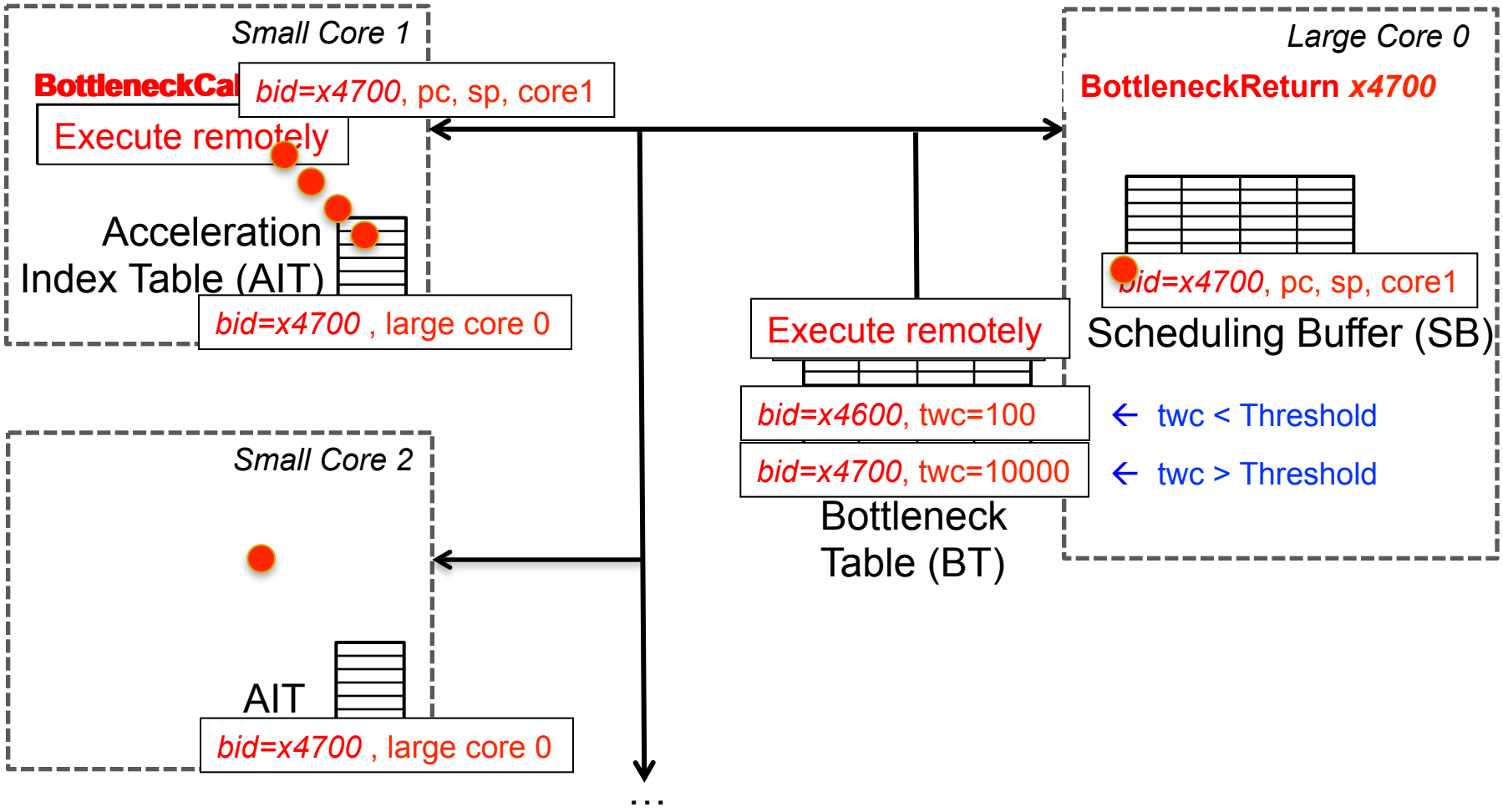
Determining Thread Waiting Cycles for Each Bottleneck



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)

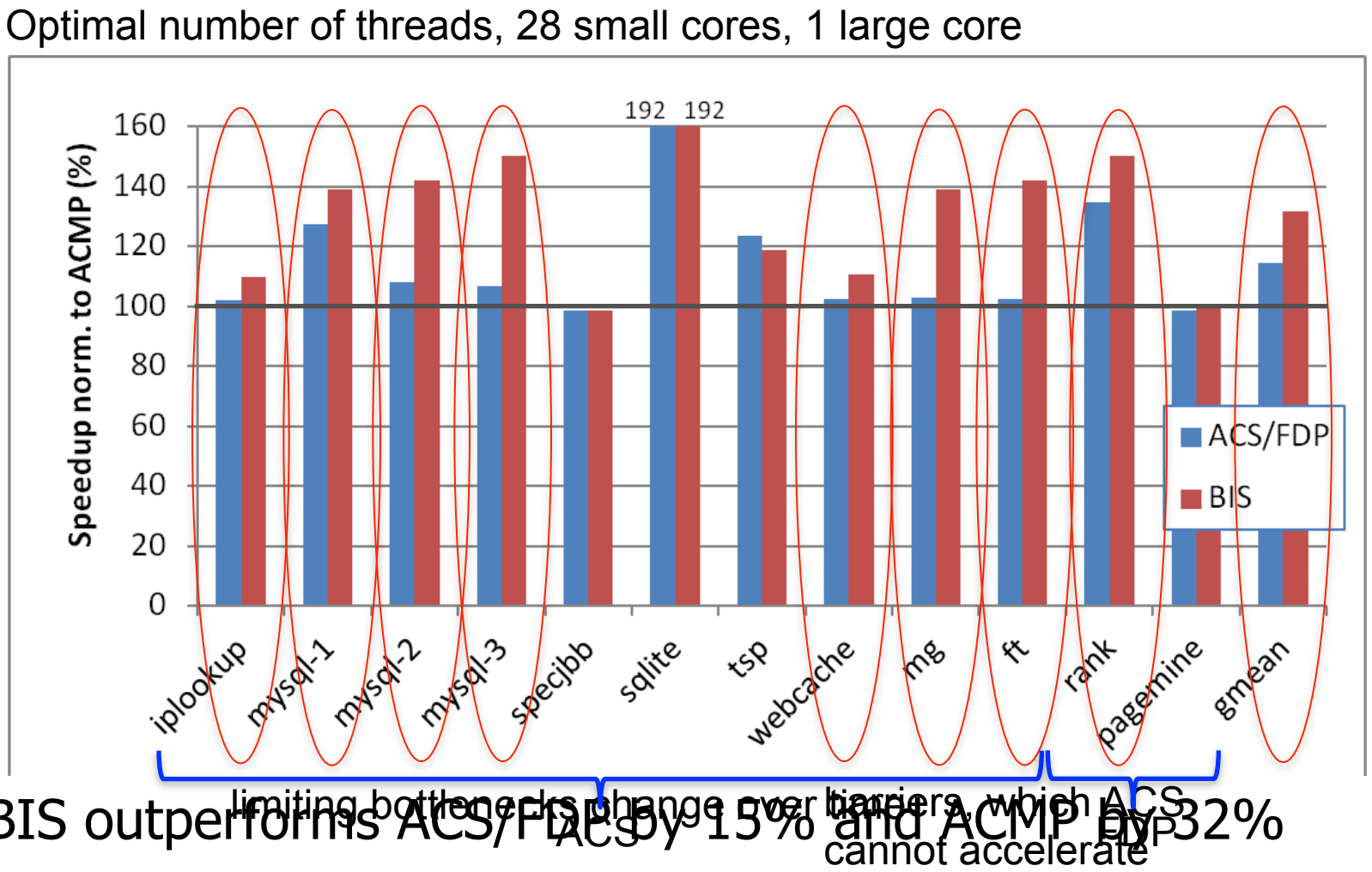
Evaluation Methodology

- 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)

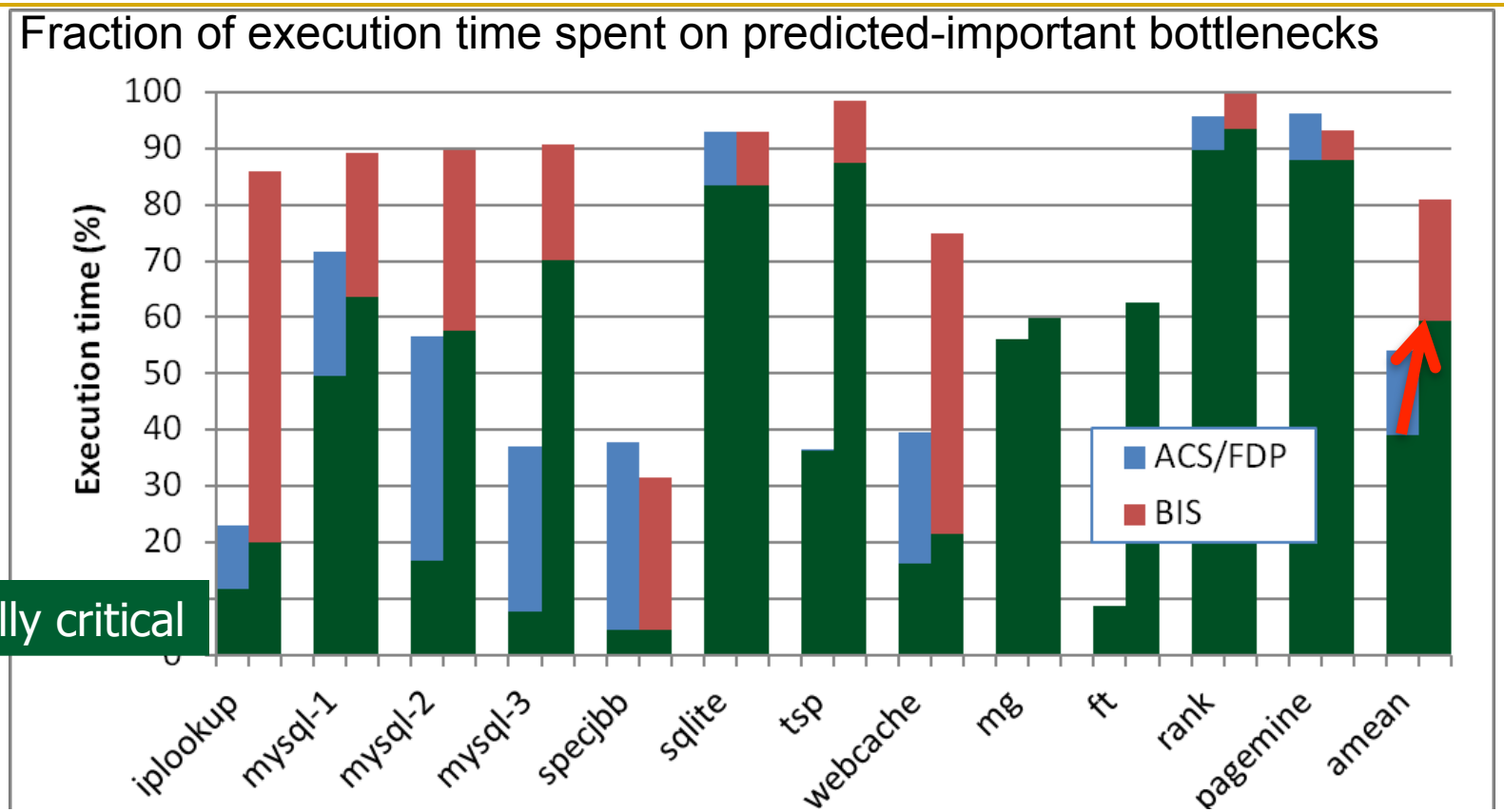
- SCMP (Symmetric CMP)
  - All small cores
- ACMP (Asymmetric CMP)
  - Accelerates only Amdahl's serial portions
  - Our baseline
- ACS (Accelerated Critical Sections)
  - Accelerates only critical sections and Amdahl's serial portions
  - Applicable to multithreaded workloads (iplookup, mysql, specjbb,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

BIS Performance Improvement



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

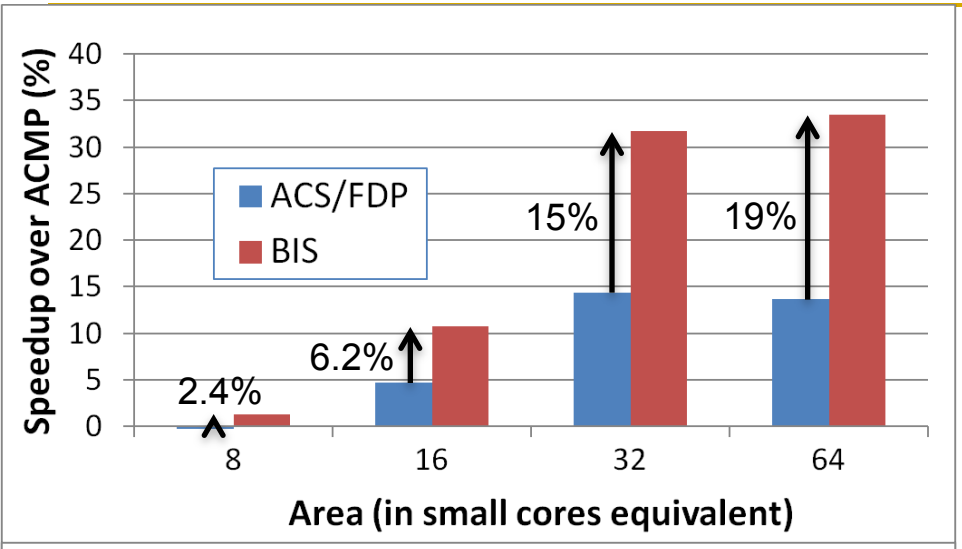
Why Does BIS Work?

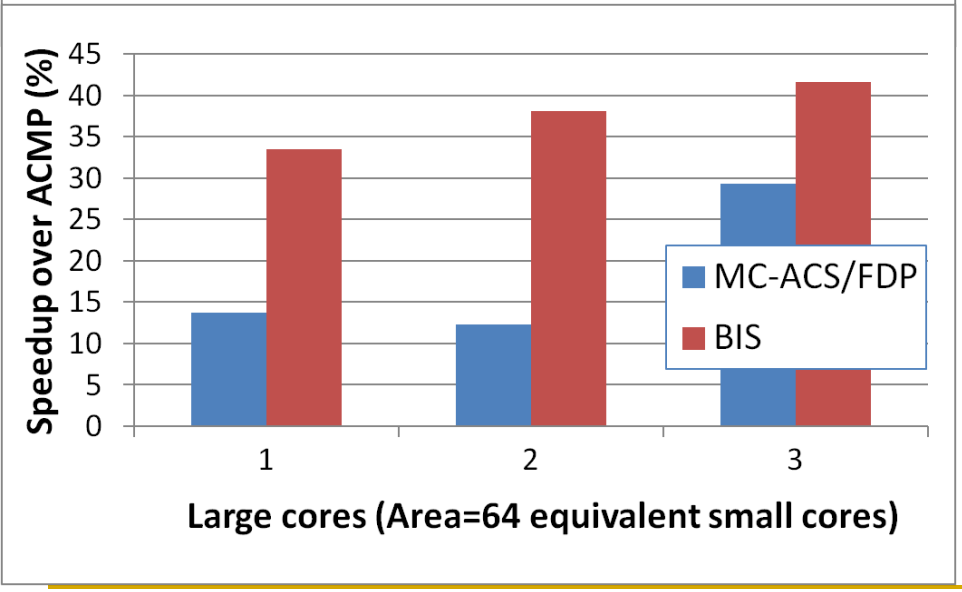


Actually critical

- 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

Handling Private Data Locality: Data Marshaling

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

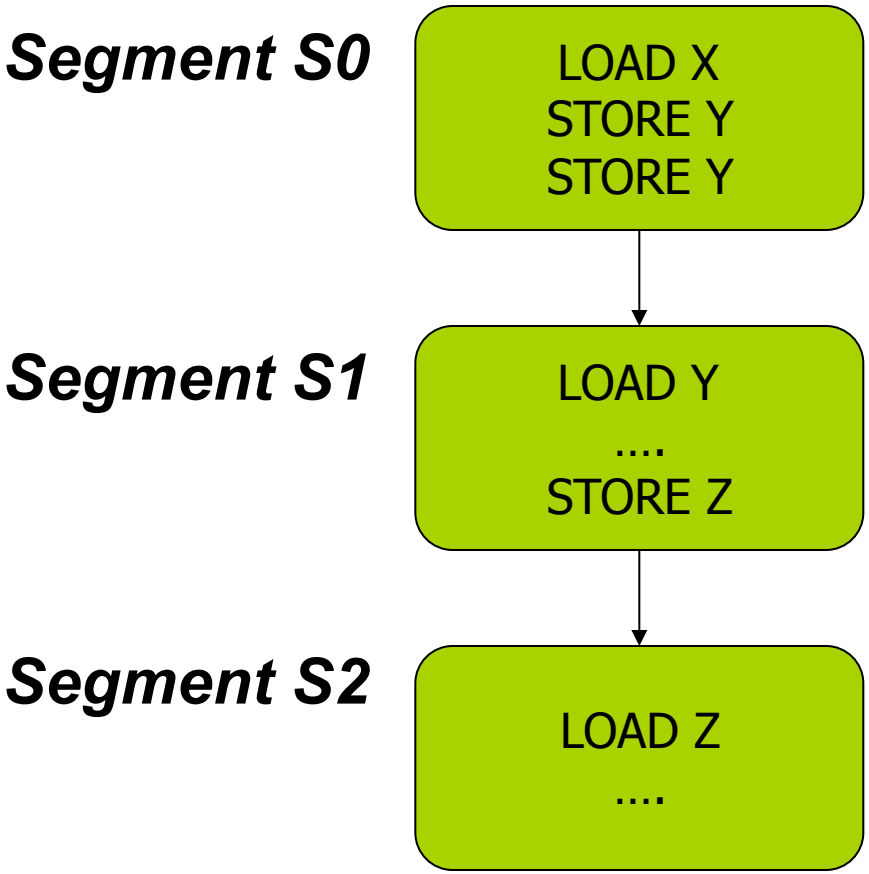
Staged Execution Model (I)

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

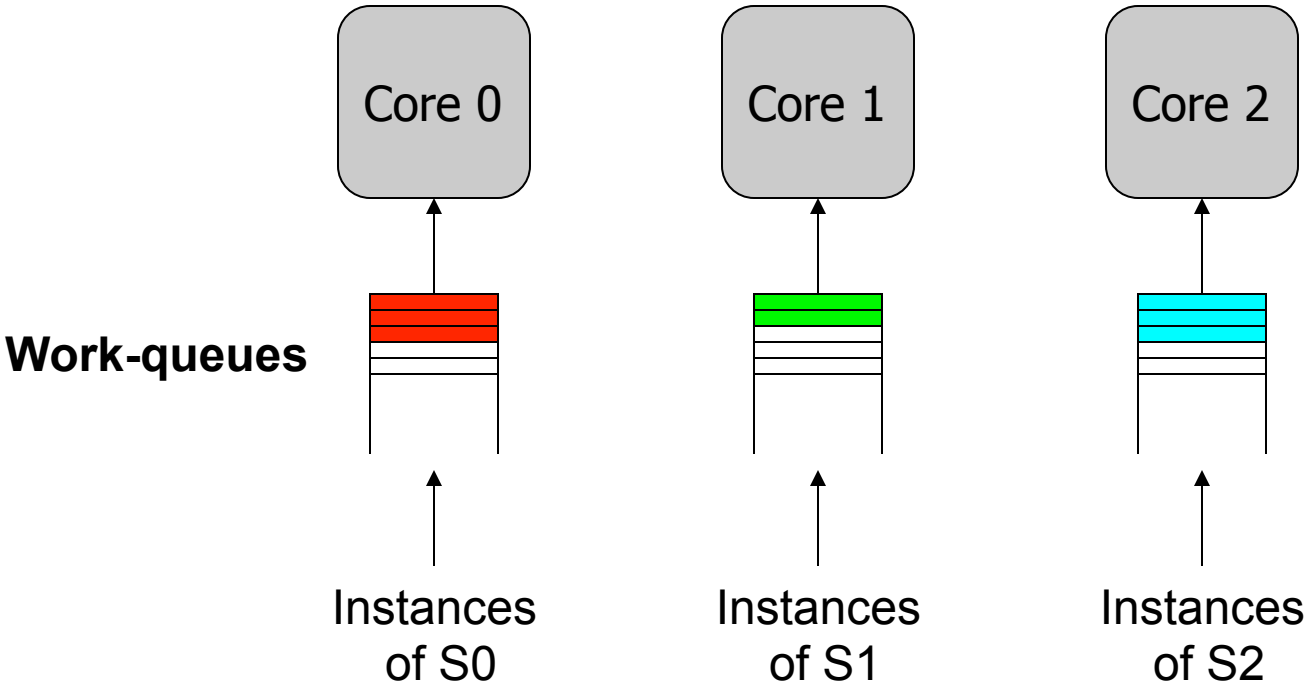
Staged Execution Model (II)

Staged Execution Model (III)

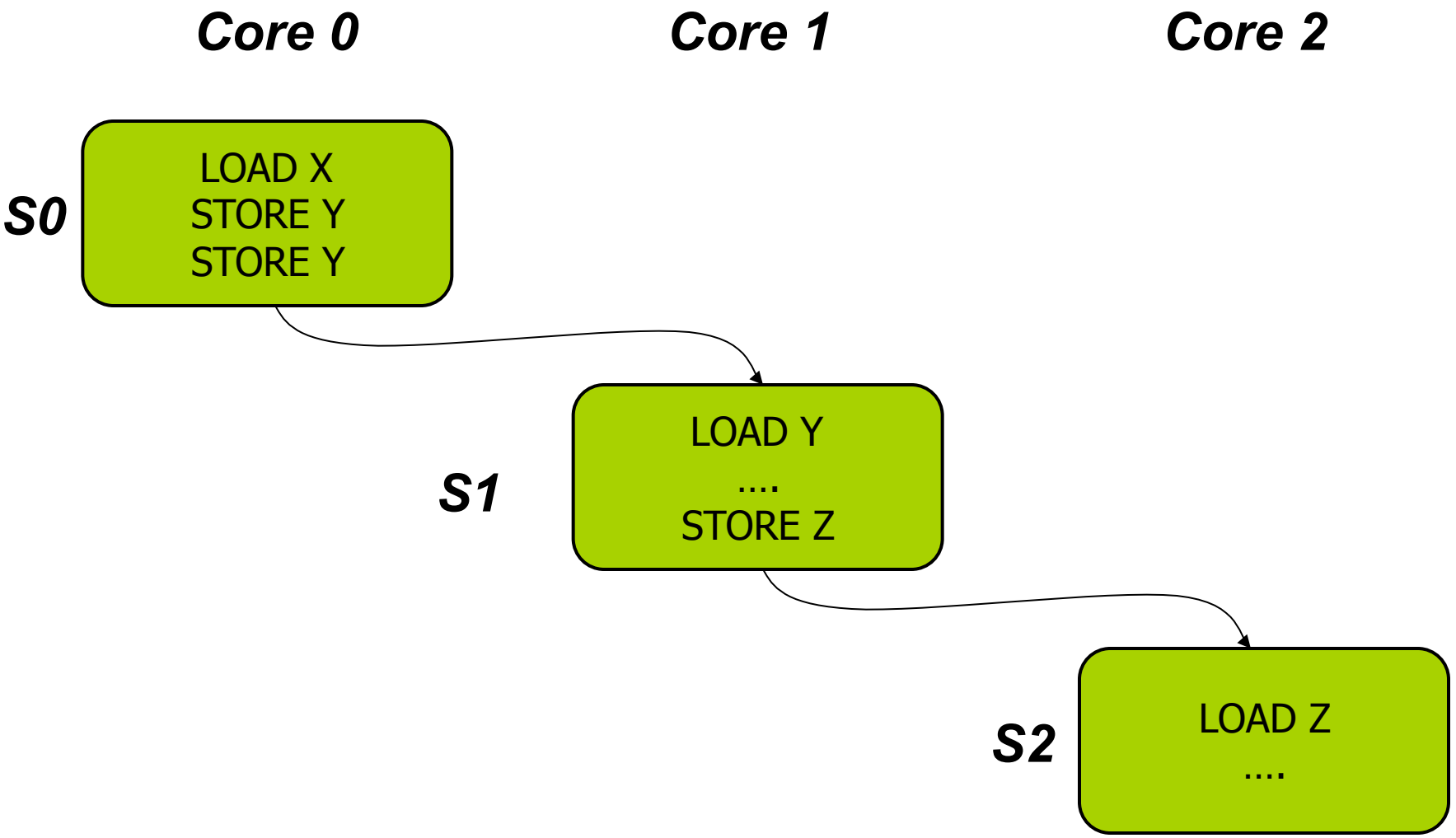
Split code into segments



Staged Execution Model (IV)



Staged Execution Model: Segment Spawning

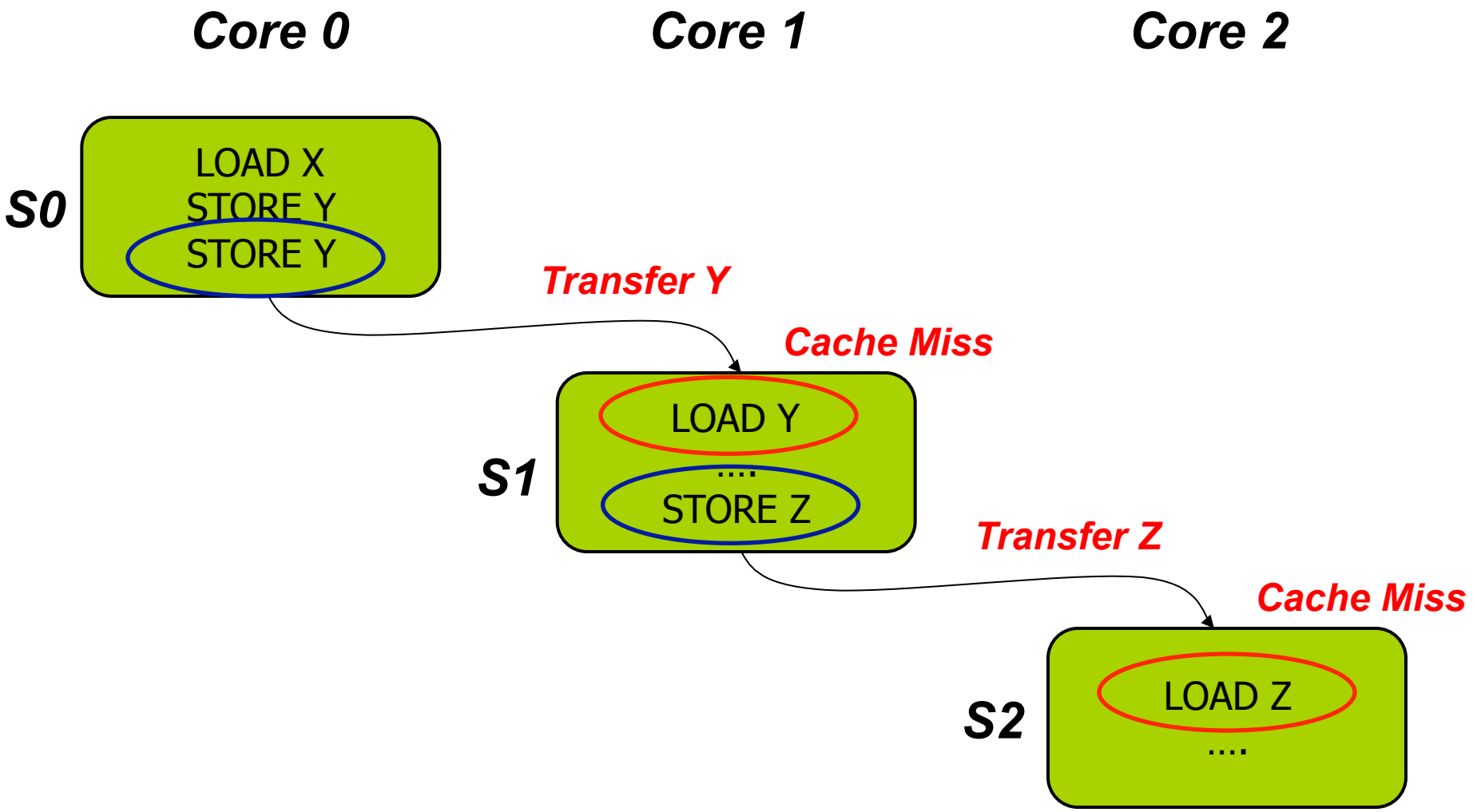


Staged Execution Model: Two Examples

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

- Producer-Consumer Pipeline Parallelism
  - Idea: Split a loop iteration into multiple "pipeline stages" where one stage consumes data produced by the next stage → each stage runs on a different core
    - Segment N: Stage N
  - Benefit: Stage-level parallelism, better locality → faster execution

Problem: Locality of Inter-segment Data



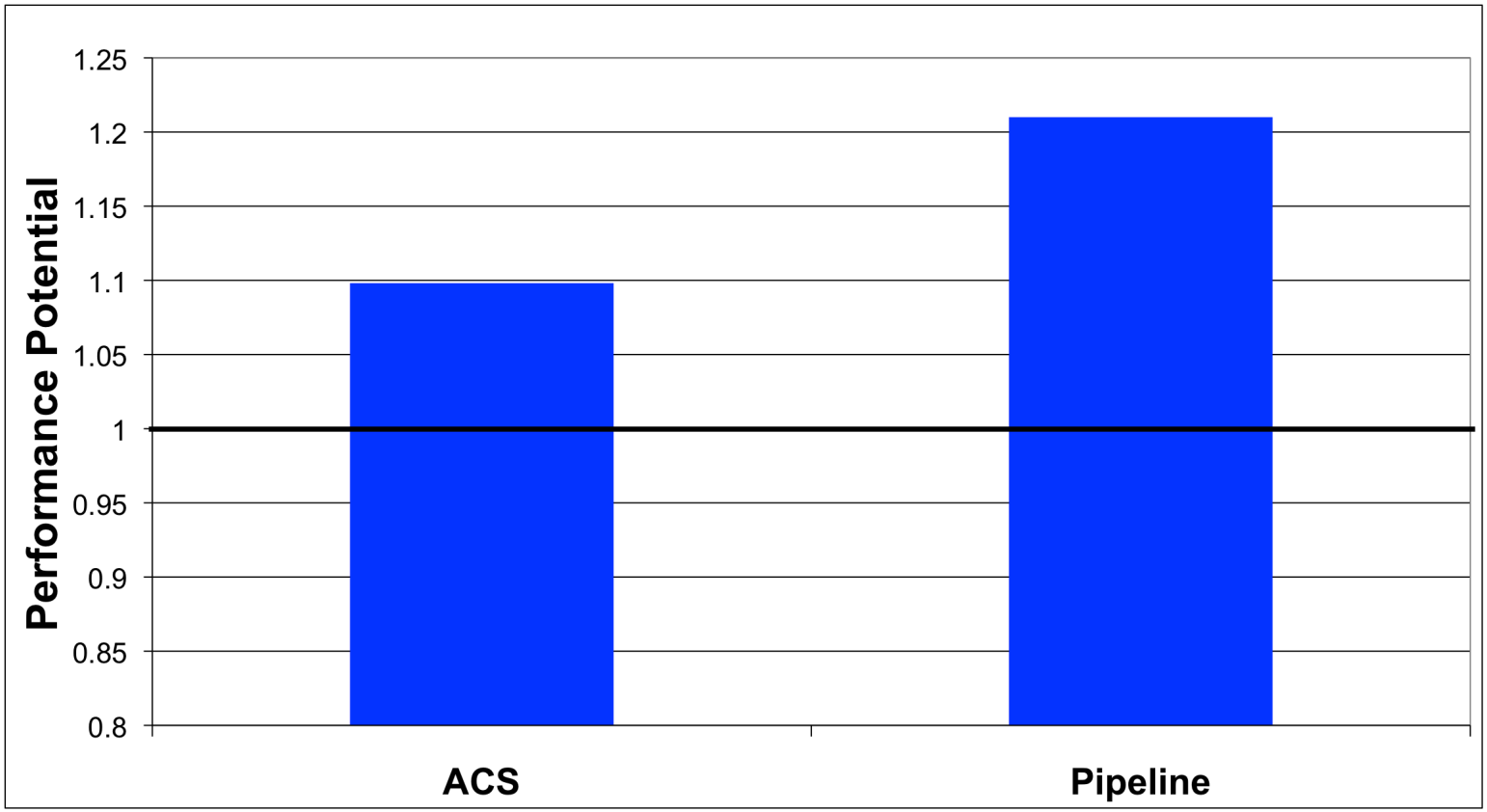
Problem: Locality of Inter-segment Data

- Accelerated Critical Sections [Suleman et al., ASPLOS 2010]
  - Idea: Ship critical sections to a large core in an ACMP
  - Problem: Critical section incurs a cache miss when it touches data produced in the non-critical section (i.e., thread private data)

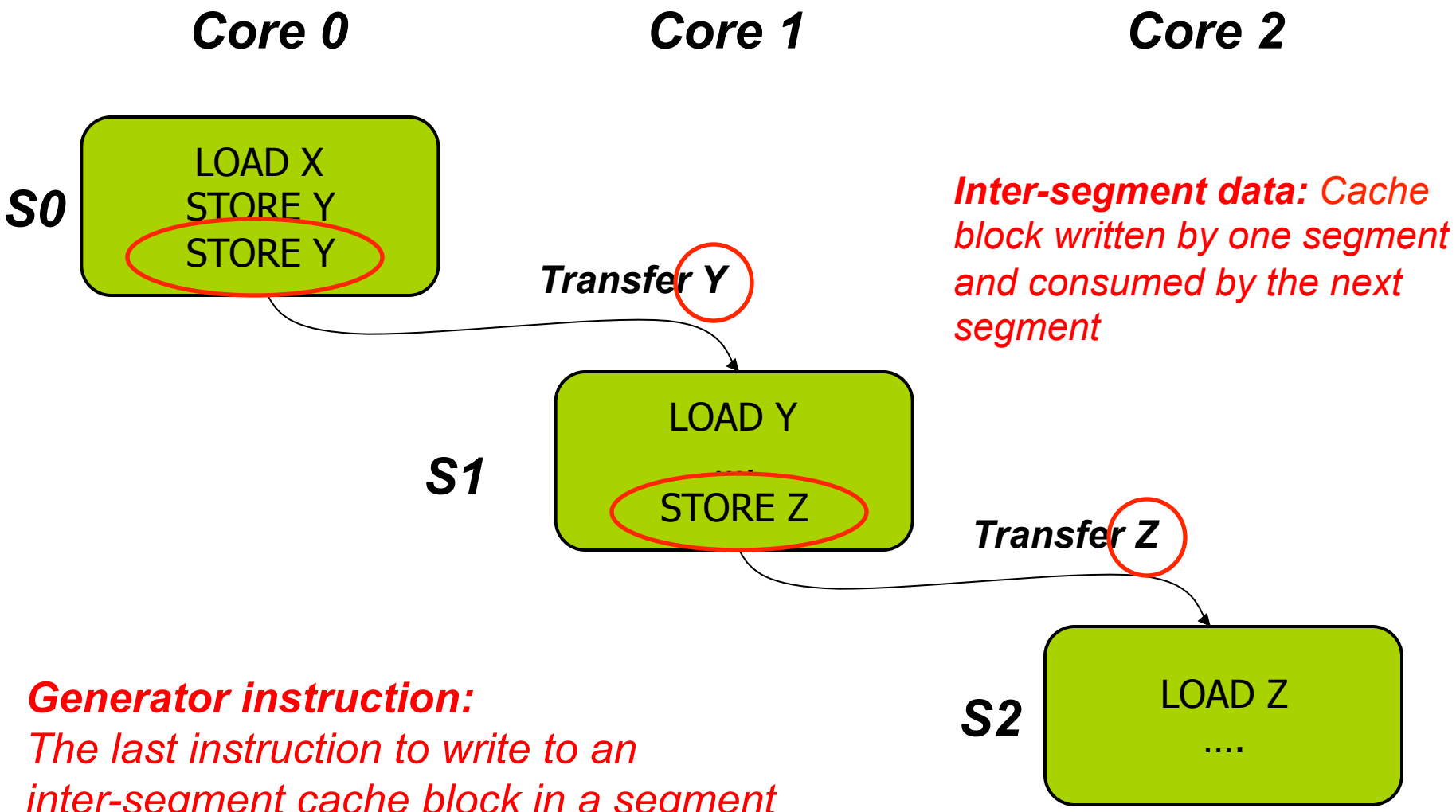
- Producer-Consumer Pipeline Parallelism
  - Idea: Split a loop iteration into multiple "pipeline stages" → each stage runs on a different core
  - Problem: A stage incurs a cache miss when it touches data produced by the previous stage

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

What if We Eliminated All Inter-segment Misses?



Terminology



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

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

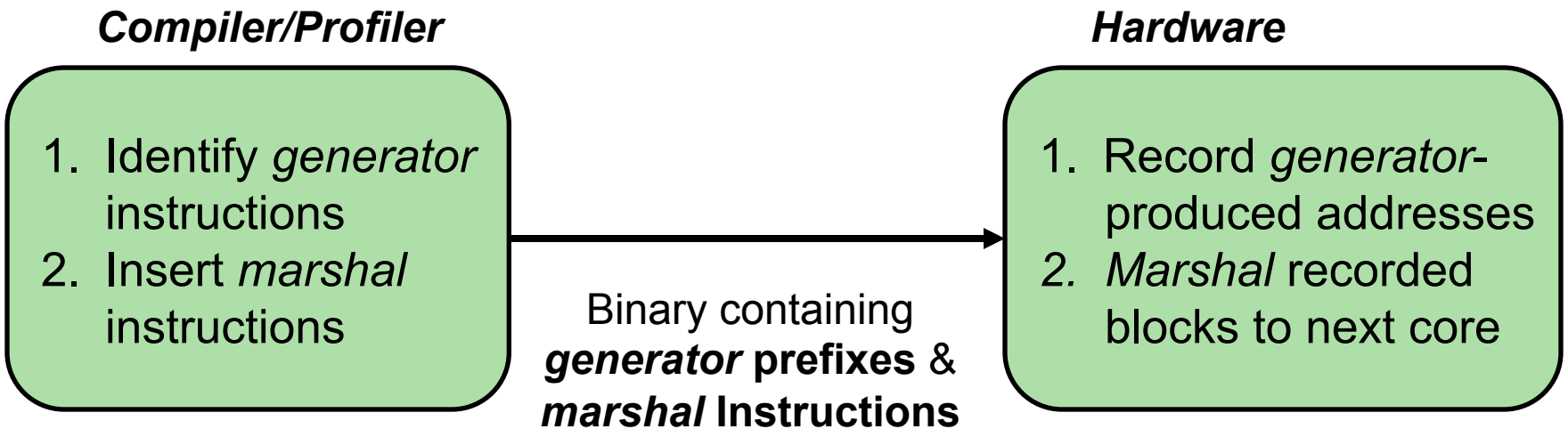
Key Observation and Idea

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

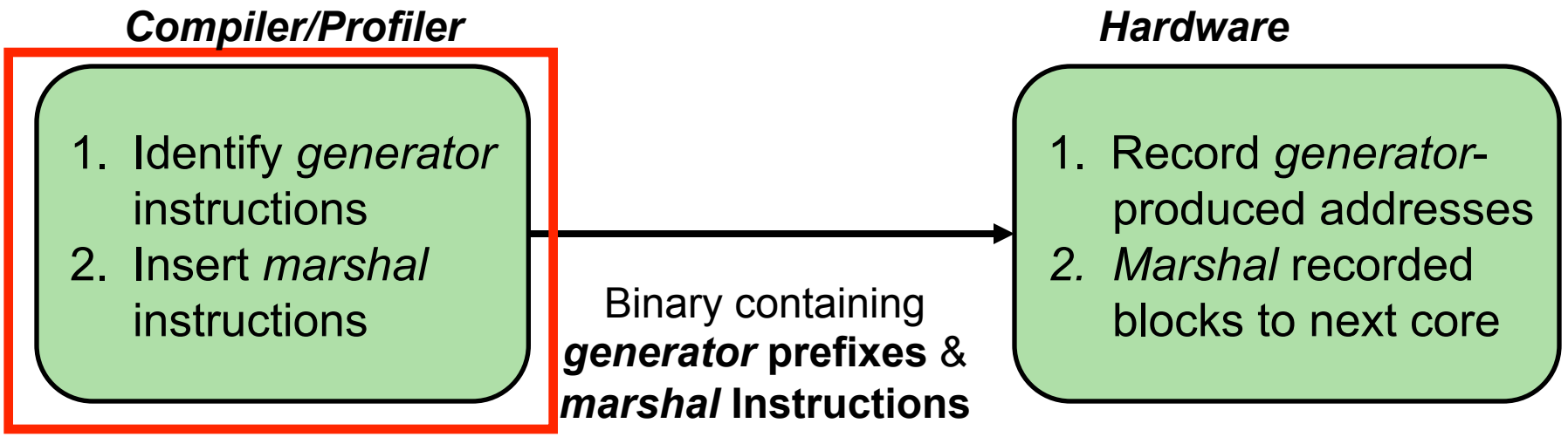
- Idea:
  - Identify the generator instructions
  - Record cache blocks produced by generator instructions
  - Proactively send such cache blocks to the next segment's core before initiating the next segment

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

Data Marshaling



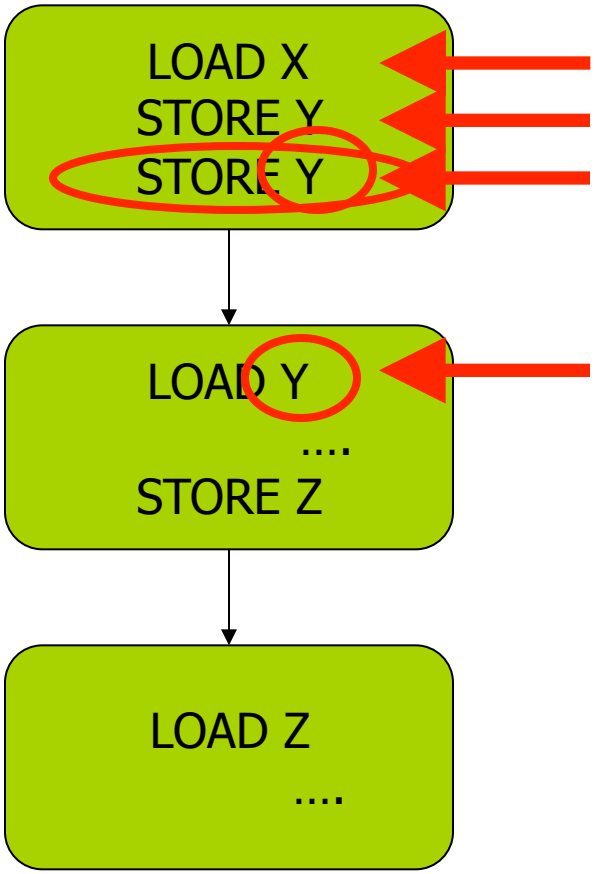
Data Marshaling



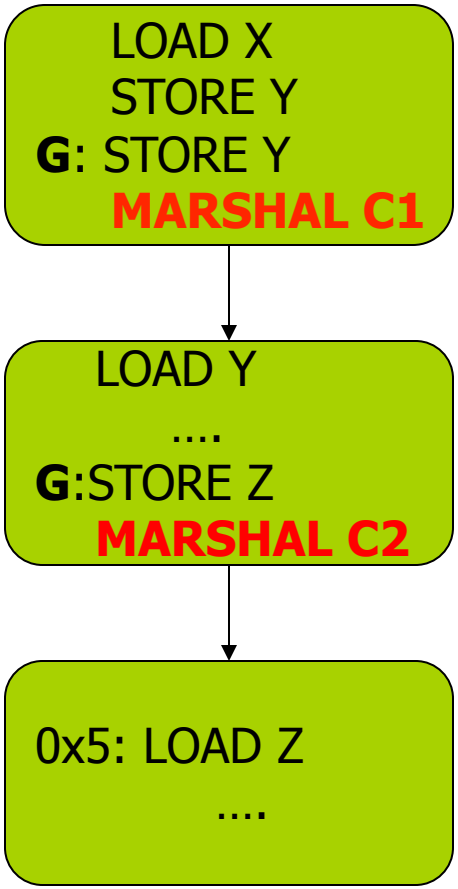
Profiling Algorithm

Inter-segment data

Mark as Generator Instruction



Marshal Instructions



When to send (Marshal)

Where to send (C1)

DM Support/Cost

- Profiler/Compiler: Generators, marshal instructions
- ISA: Generator prefix, marshal instructions
- Library/Hardware: Bind next segment ID to a physical core

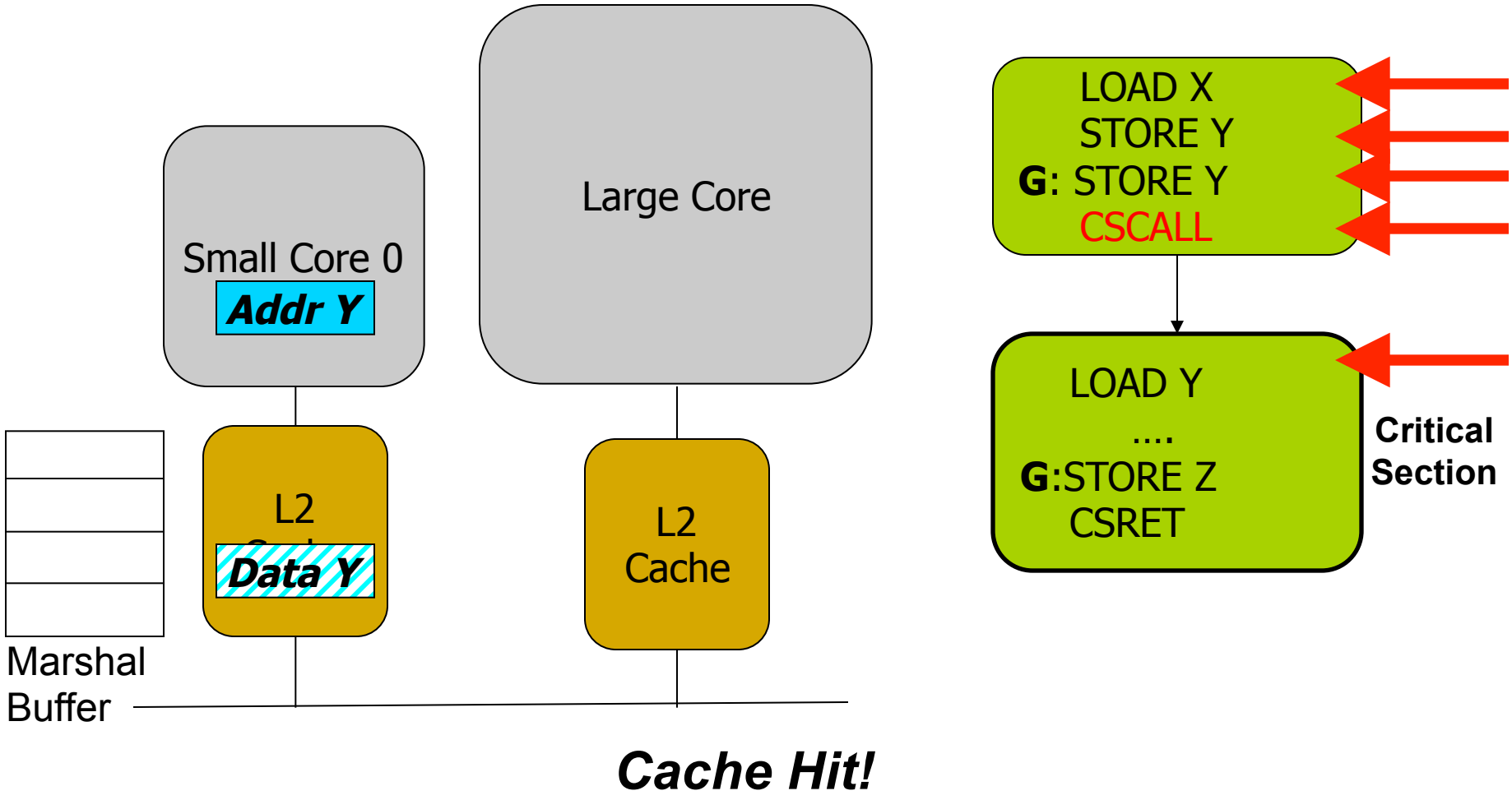
- Hardware
  - Marshal Buffer
    - Stores physical addresses of cache blocks to be marshaled
    - 16 entries enough for almost all workloads → 96 bytes per core
  - Ability to execute generator prefixes and marshal instructions
  - Ability to push data to another cache

DM: Advantages, Disadvantages

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

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

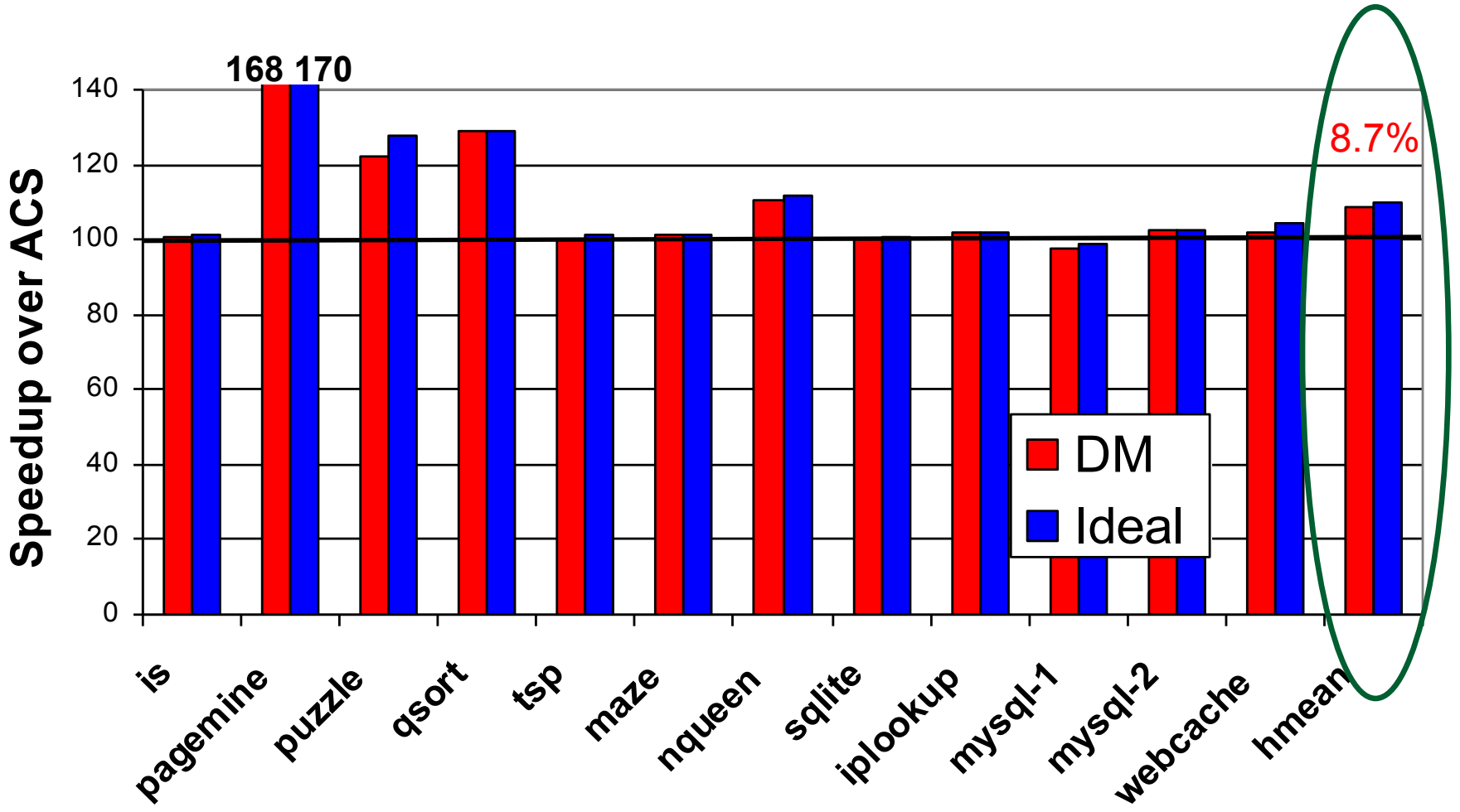
Accelerated Critical Sections with DM



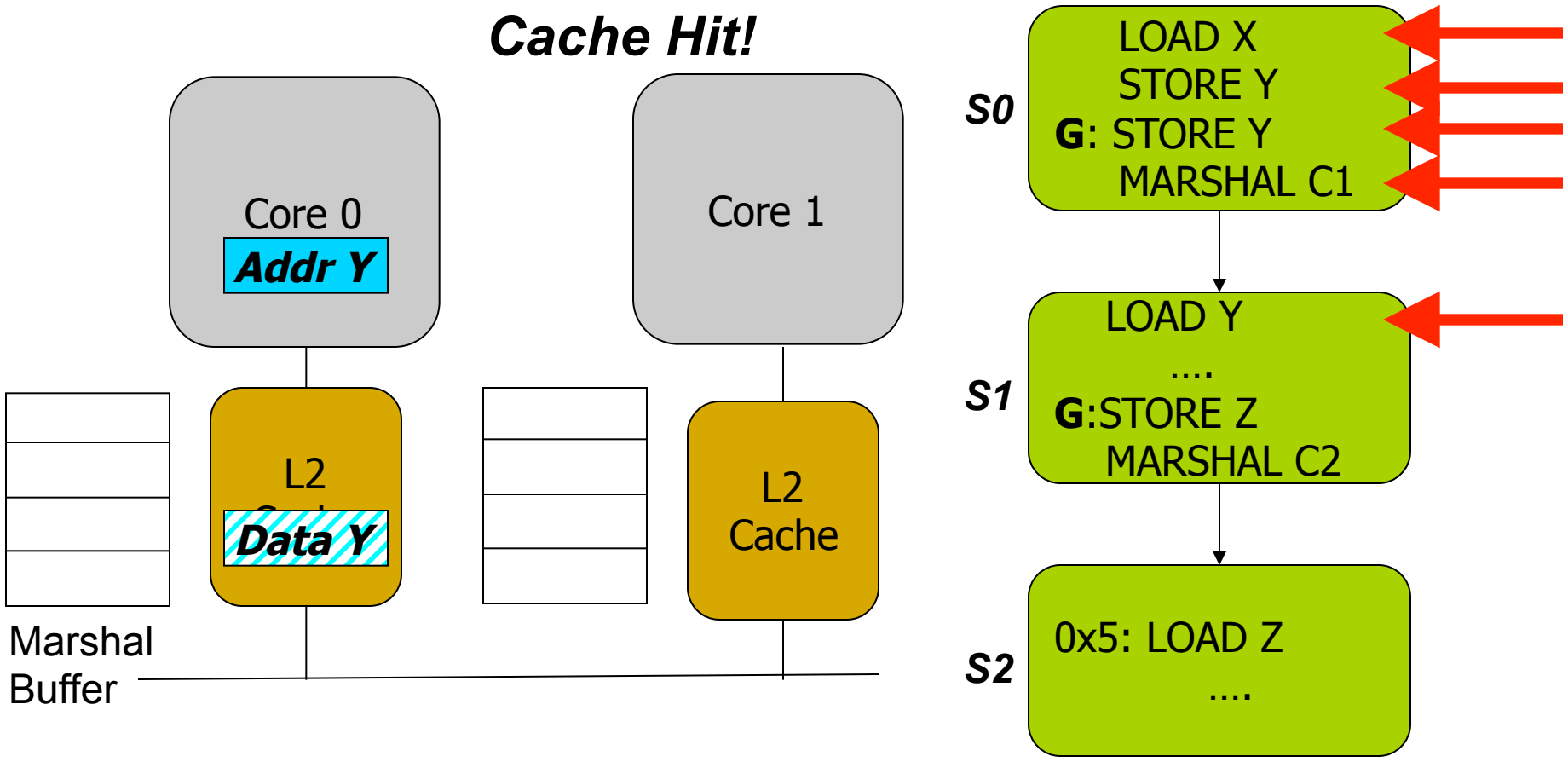
Accelerated Critical Sections: Methodology

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

DM on Accelerated Critical Sections: Results



Pipeline Parallelism

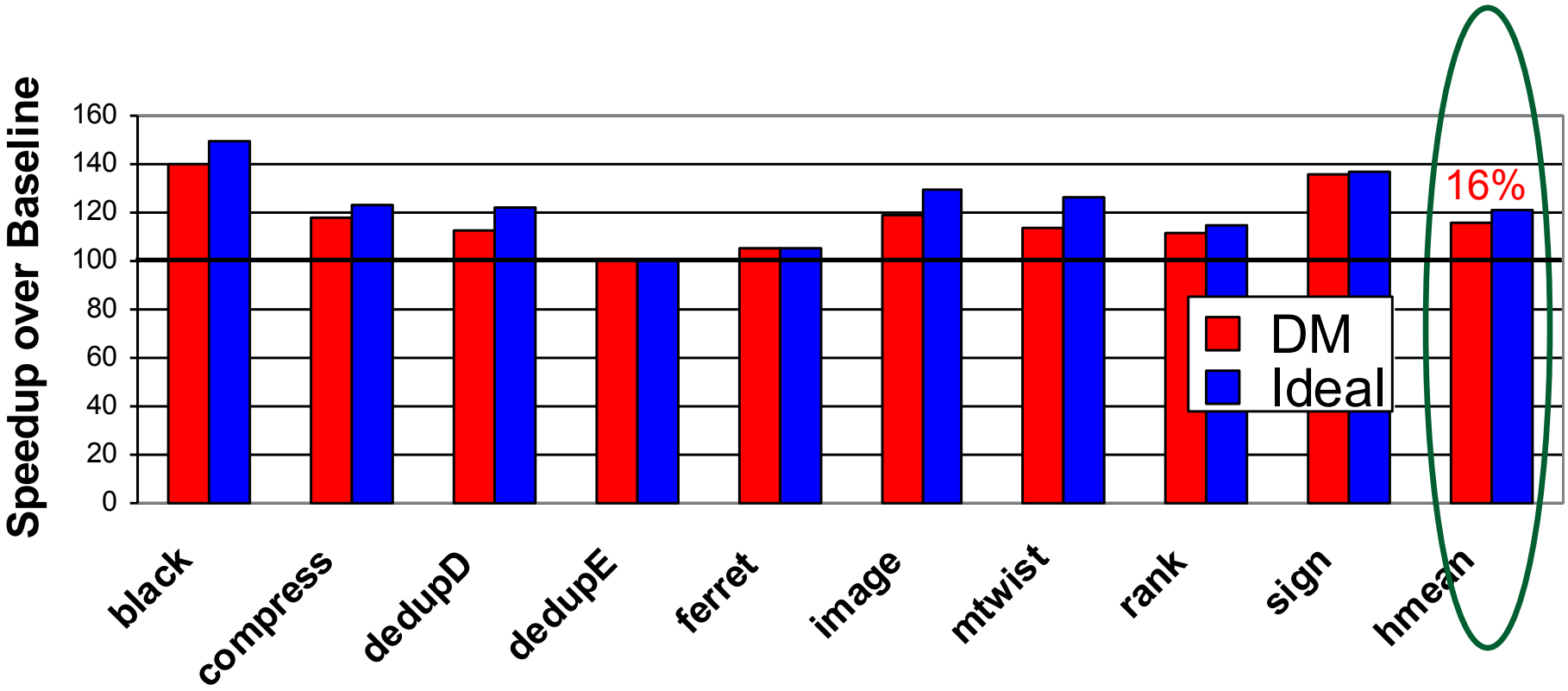


Pipeline Parallelism: Methodology

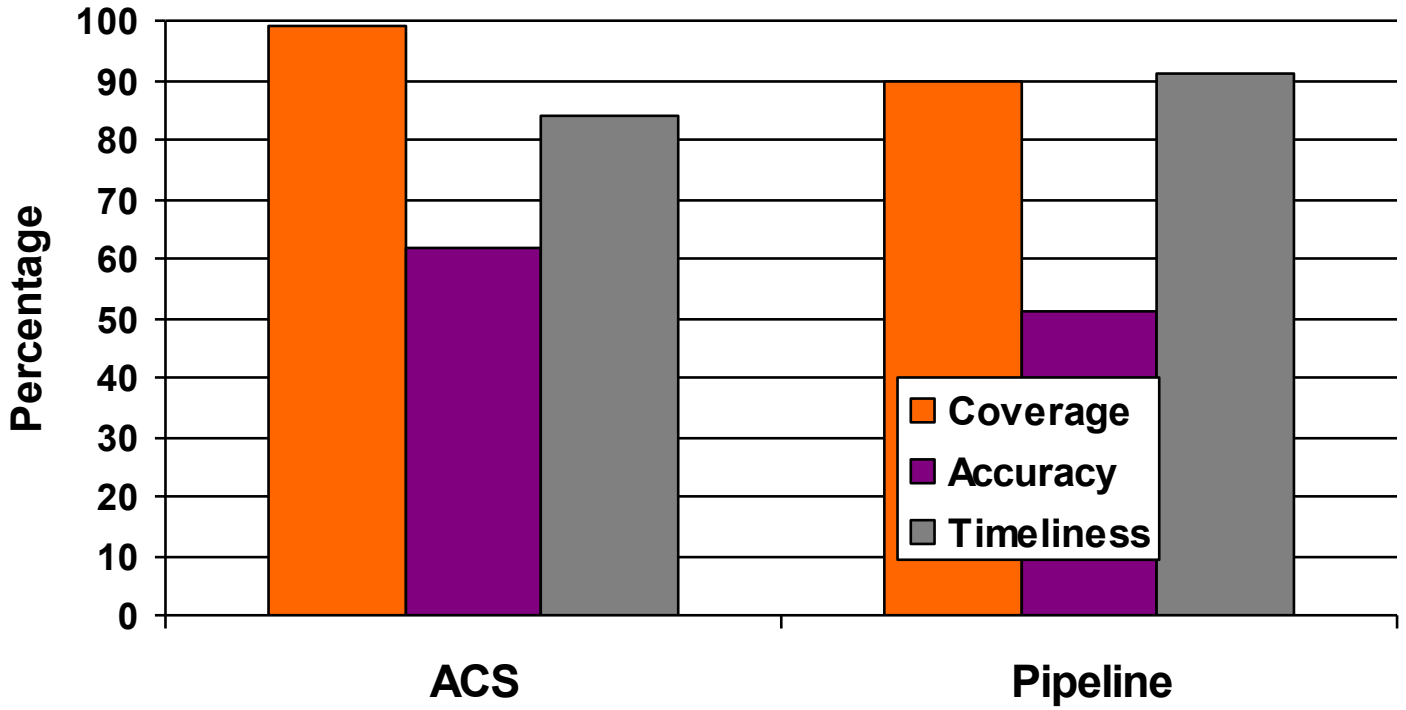
- Workloads: 9 applications with pipeline parallelism
  - Financial, compression, multimedia, encoding/decoding
  - Different training and simulation input sets

- Multi-core x86 simulator
  - 32-core CMP: 2GHz, in-order, 2-wide, 5-stage
  - Aggressive stream prefetcher employed at each core
  - Private 32 KB L1, private 256KB L2, 8MB shared L3
  - On-chip interconnect: Bi-directional ring, 5-cycle hop latency

DM on Pipeline Parallelism: Results



DM Coverage, Accuracy, Timeliness



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

Scaling Results

- DM performance improvement increases with
  - More cores
  - Higher interconnect latency
  - Larger private L2 caches

- Why? Inter-segment data misses become a larger bottleneck
  - More cores → More communication
  - Higher latency → Longer stalls due to communication
  - Larger L2 cache → Communication misses remain

Other Applications of Data Marshaling

- Can be applied to other Staged Execution models
  - Task parallelism models
    - Cilk, Intel TBB, Apple Grand Central Dispatch
  - Special-purpose remote functional units
  - Computation spreading [Chakraborty et al., ASPLOS' 06]
  - Thread motion/migration [e.g., Rangan et al., ISCA' 09]

- Can be an enabler for more aggressive SE models
  - Lowers the cost of data migration
    - an important overhead in remote execution of code segments
  - Remote execution of finer-grained tasks can become more feasible → finer-grained parallelization in multi-cores

Data Marshaling Summary

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

A Case for Asymmetry Everywhere

Onur Mutlu,
"Asymmetry Everywhere (with Automatic Resource Management)"
CRA Workshop on Advancing Computer Architecture Research: Popular Parallel Programming, San Diego, CA, February 2010.
Position paper

The Setting

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

Shield the Programmer from Shared Resources

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

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

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

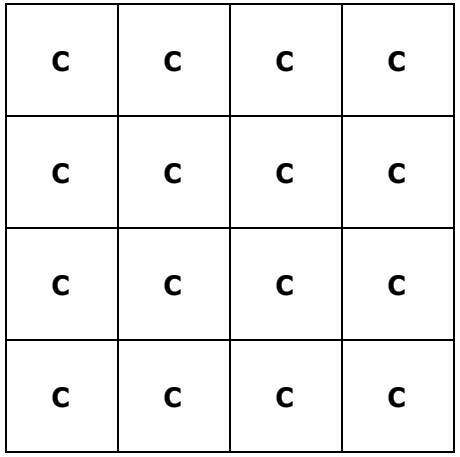
Shared Resource Management: Goals

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

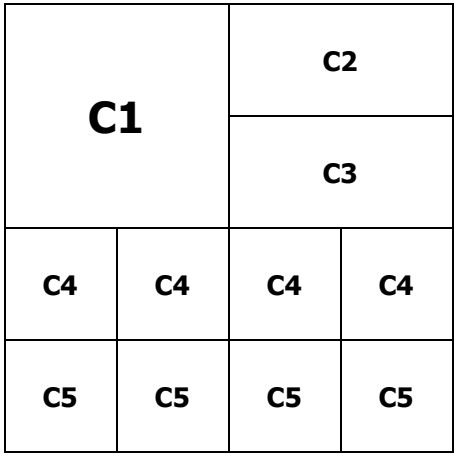
- Minimize energy/power consumption
- While satisfying performance/SLA requirements
  - Provide predictability and Quality of Service
- Minimize programmer effort
  - In creating optimized parallel programs

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

Asymmetry Enables Customization



Symmetric

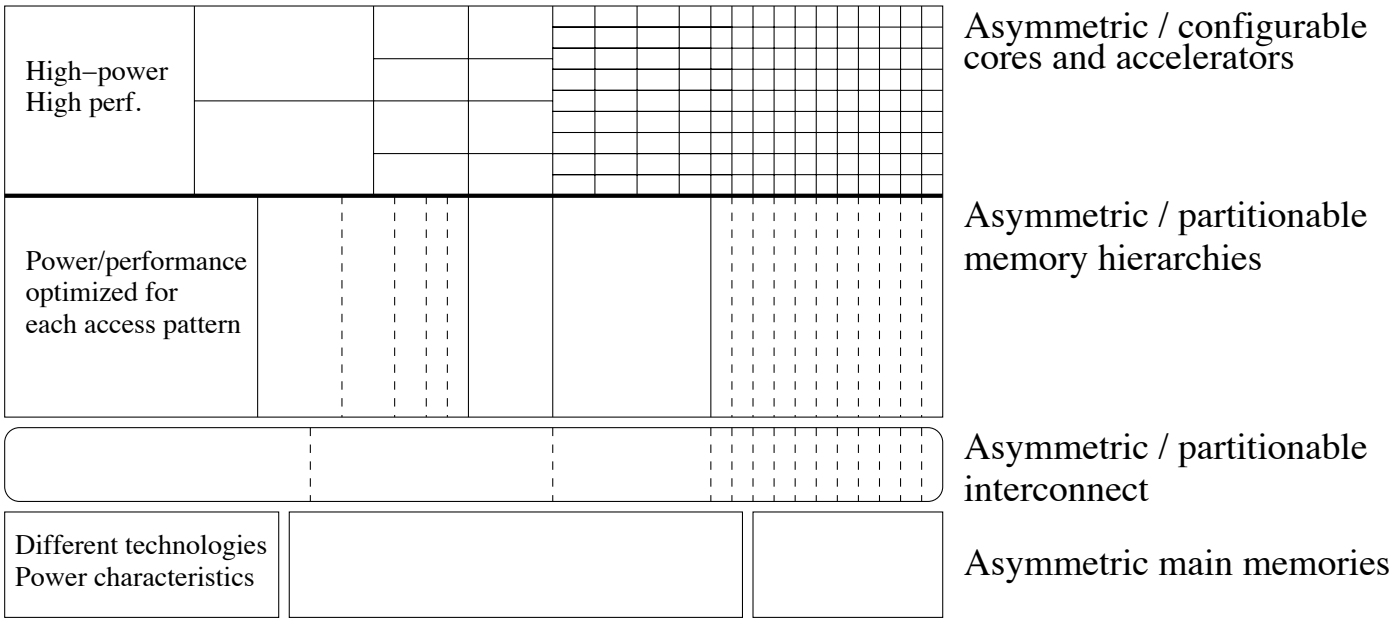


Asymmetric

- 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.)

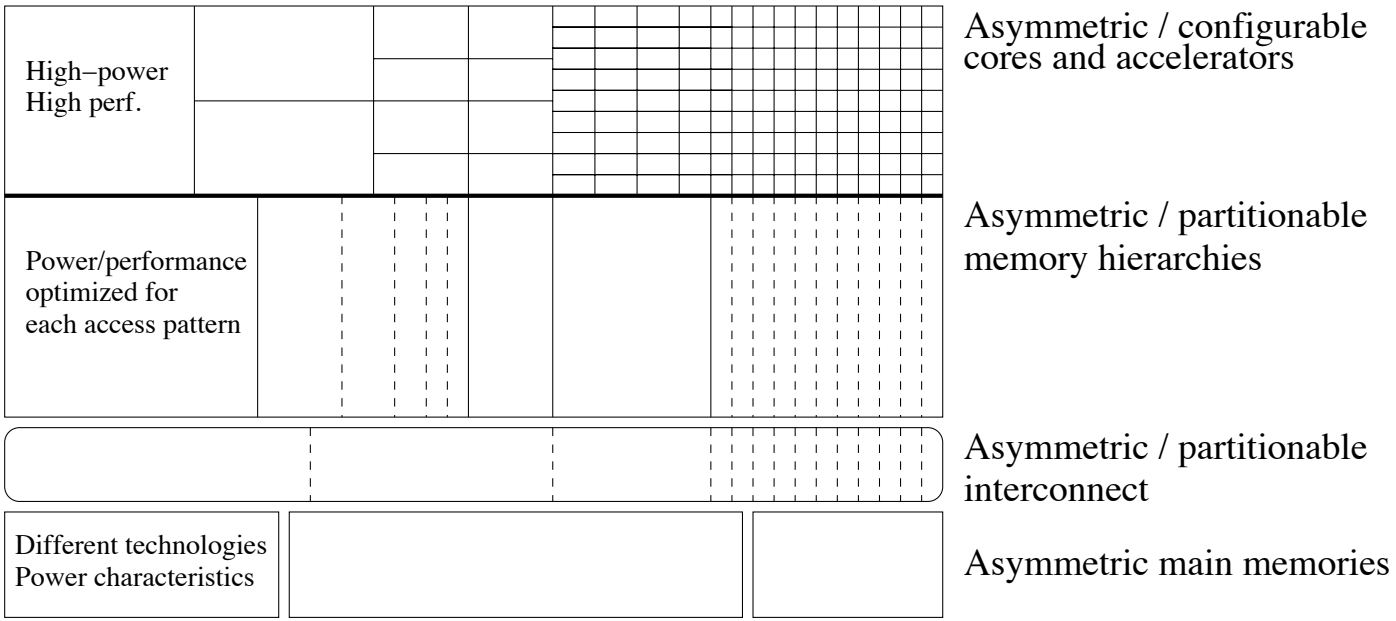
Thought Experiment: Asymmetry Everywhere

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



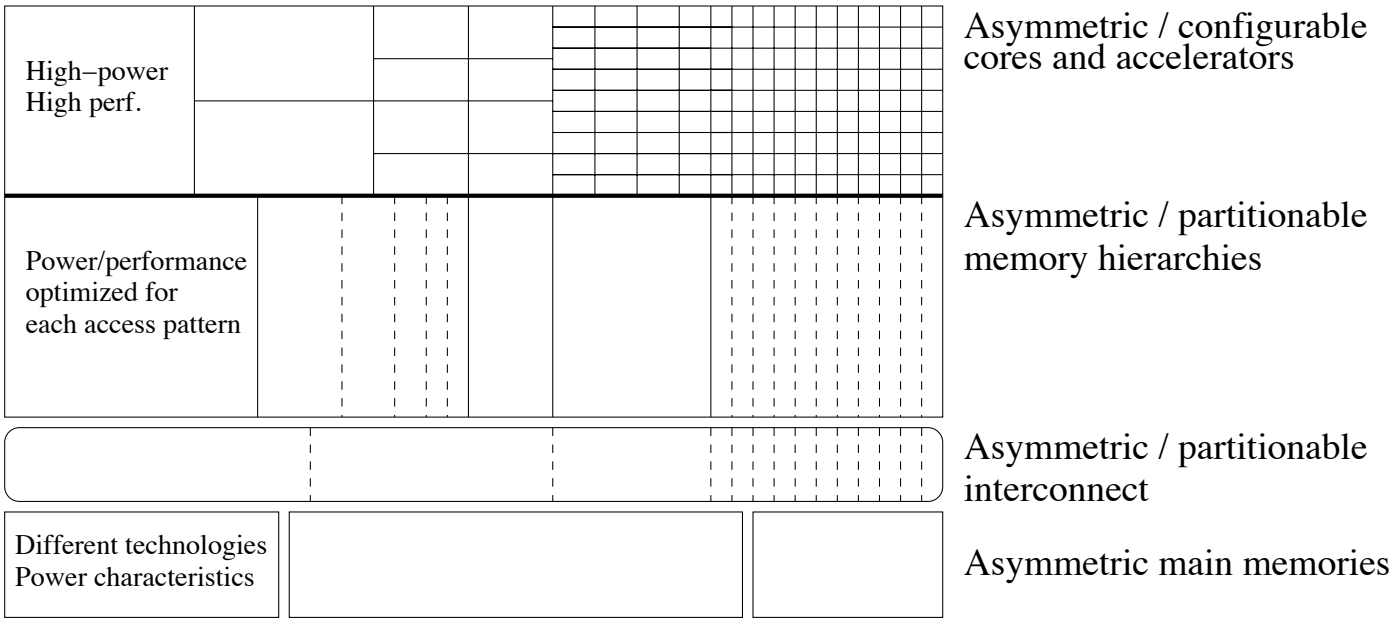
Thought Experiment: Asymmetry Everywhere

- 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

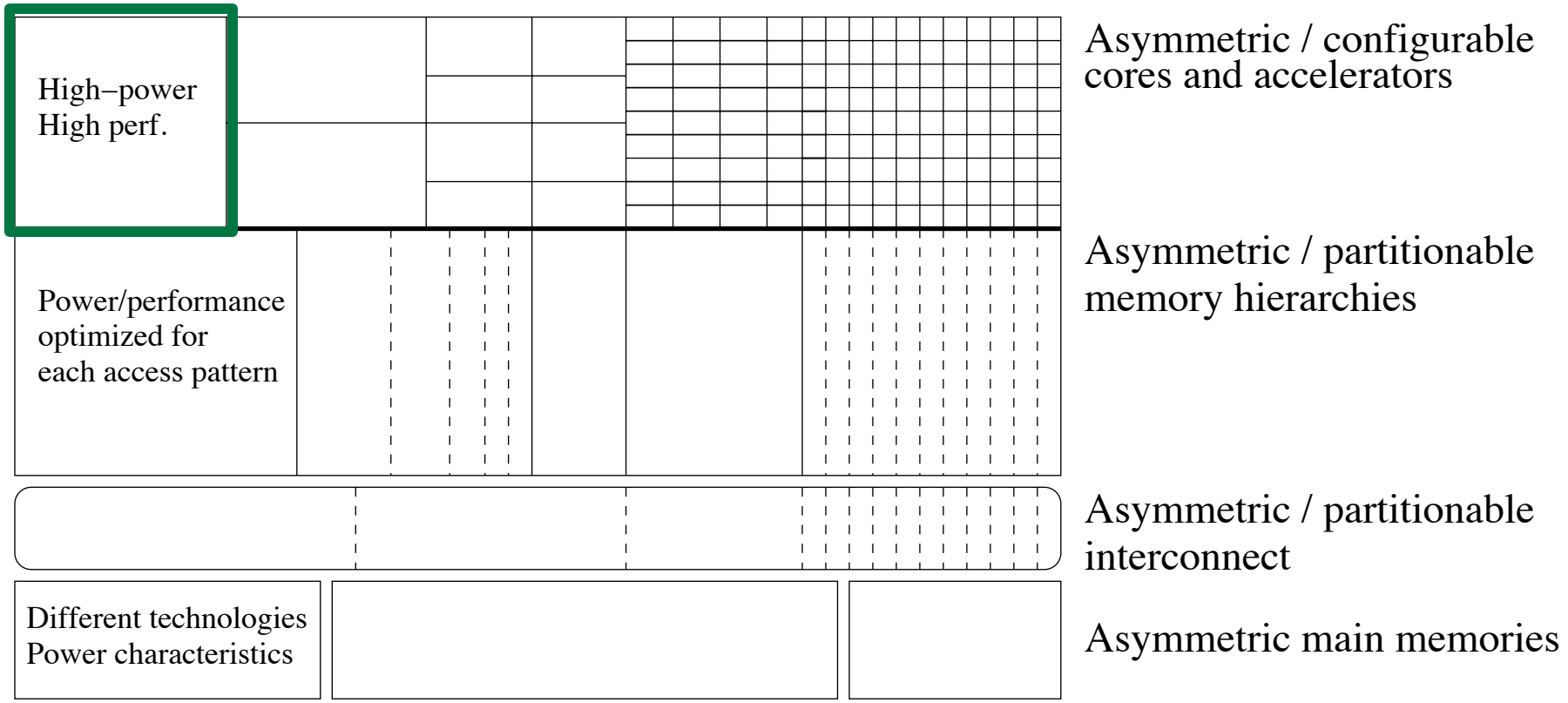
- Morph software components to match asymmetric HW components
  - Multiple versions for different resource characteristics



Many Research and Design Questions

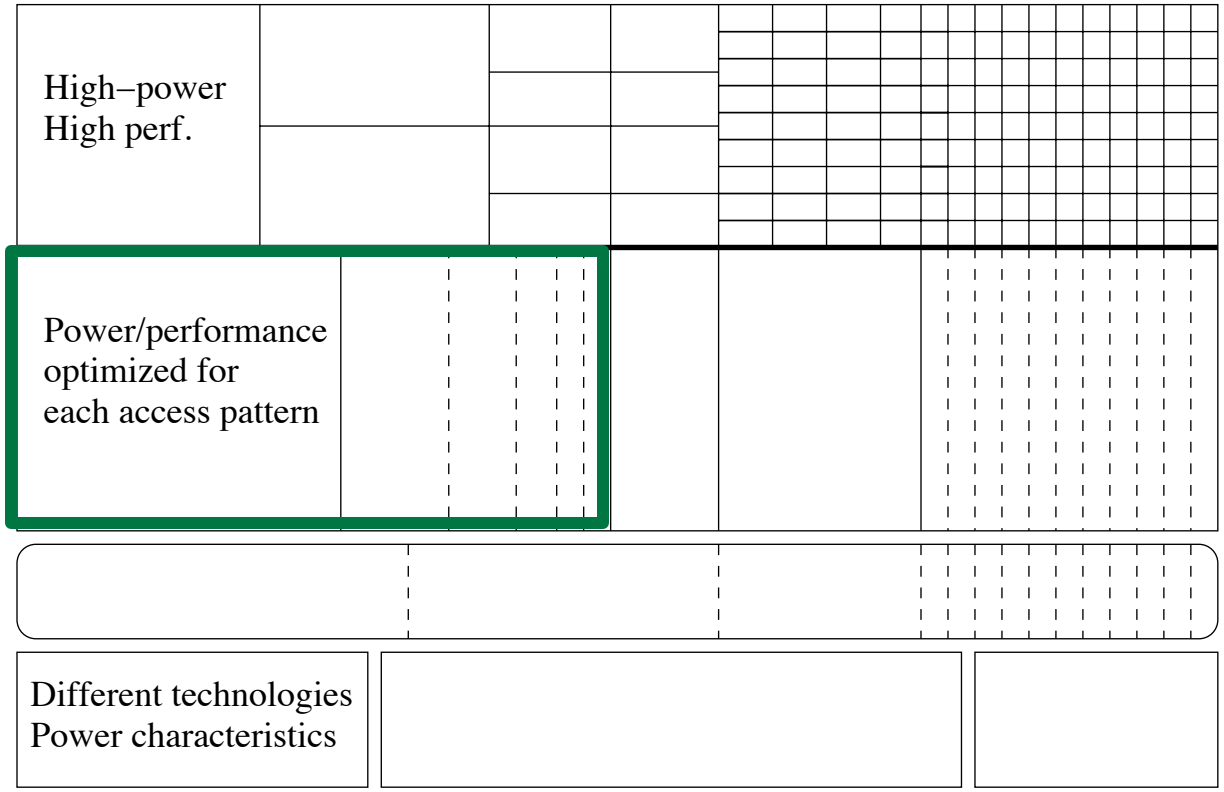
- How to design asymmetric components?
  - Fixed, partitionable, reconfigurable components?
  - What types of asymmetry? Access patterns, technologies?
- What monitoring to perform cooperatively in HW/SW?
  - Automatically discover phase/task requirements
- How to design feedback/control loop between 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

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



Asymmetric / configurable cores and accelerators

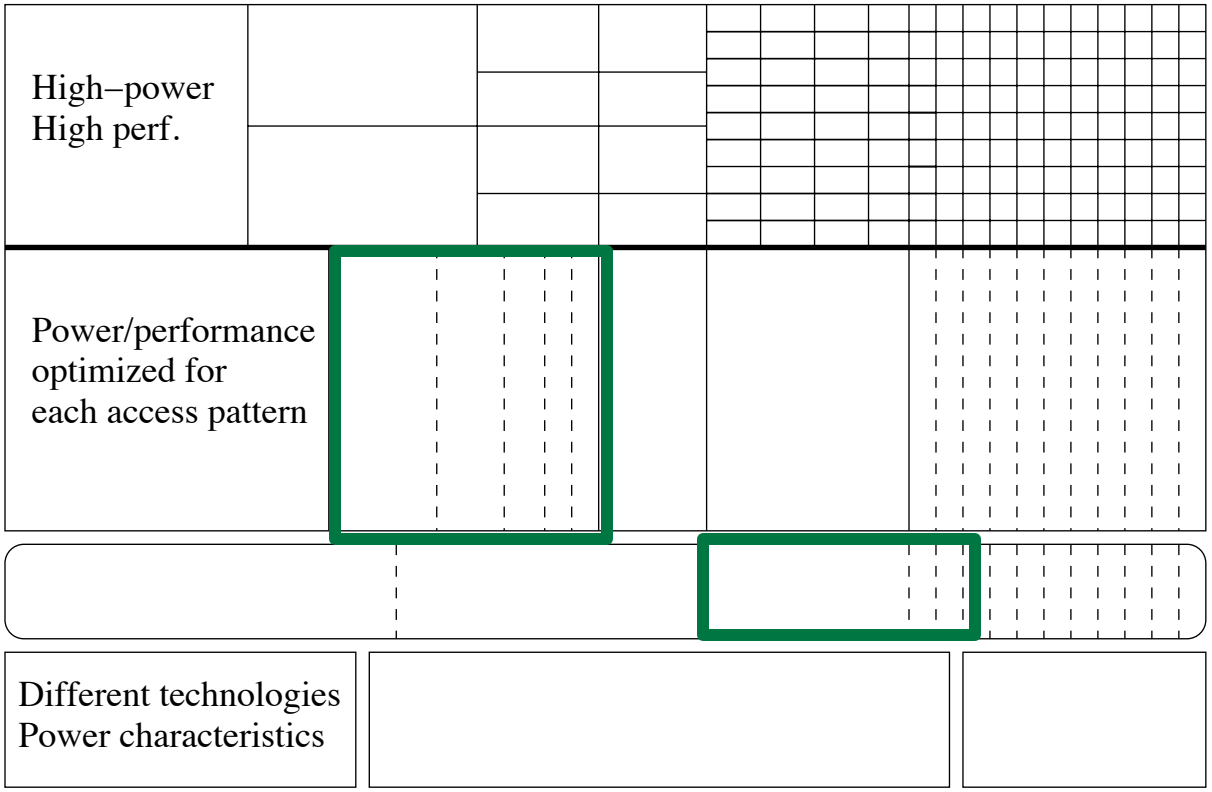
Asymmetric / partitionable memory hierarchies

Asymmetric / partitionable interconnect

Asymmetric main memories

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

Exploiting Asymmetry: Simple Examples



Asymmetric / configurable cores and accelerators

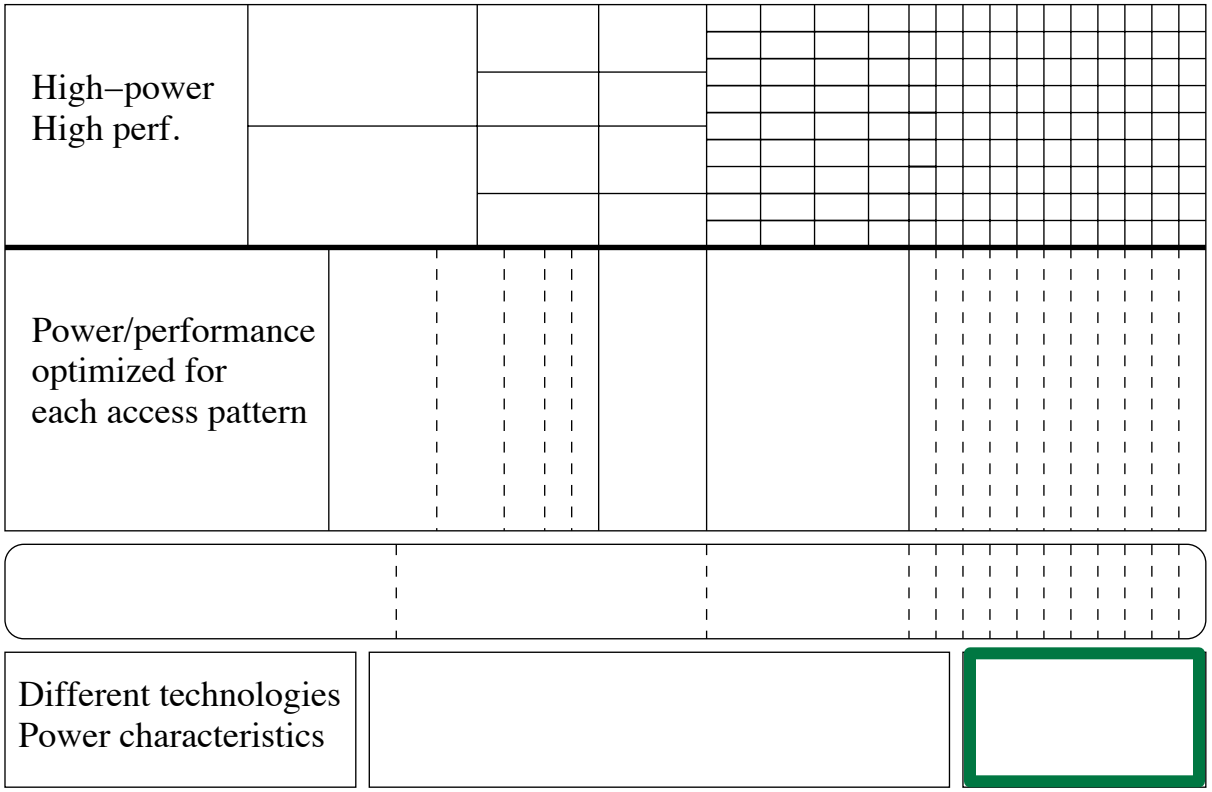
Asymmetric / partitionable memory hierarchies

Asymmetric / partitionable interconnect

Asymmetric main memories

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

Exploiting Asymmetry: Simple Examples



Asymmetric / configurable cores and accelerators

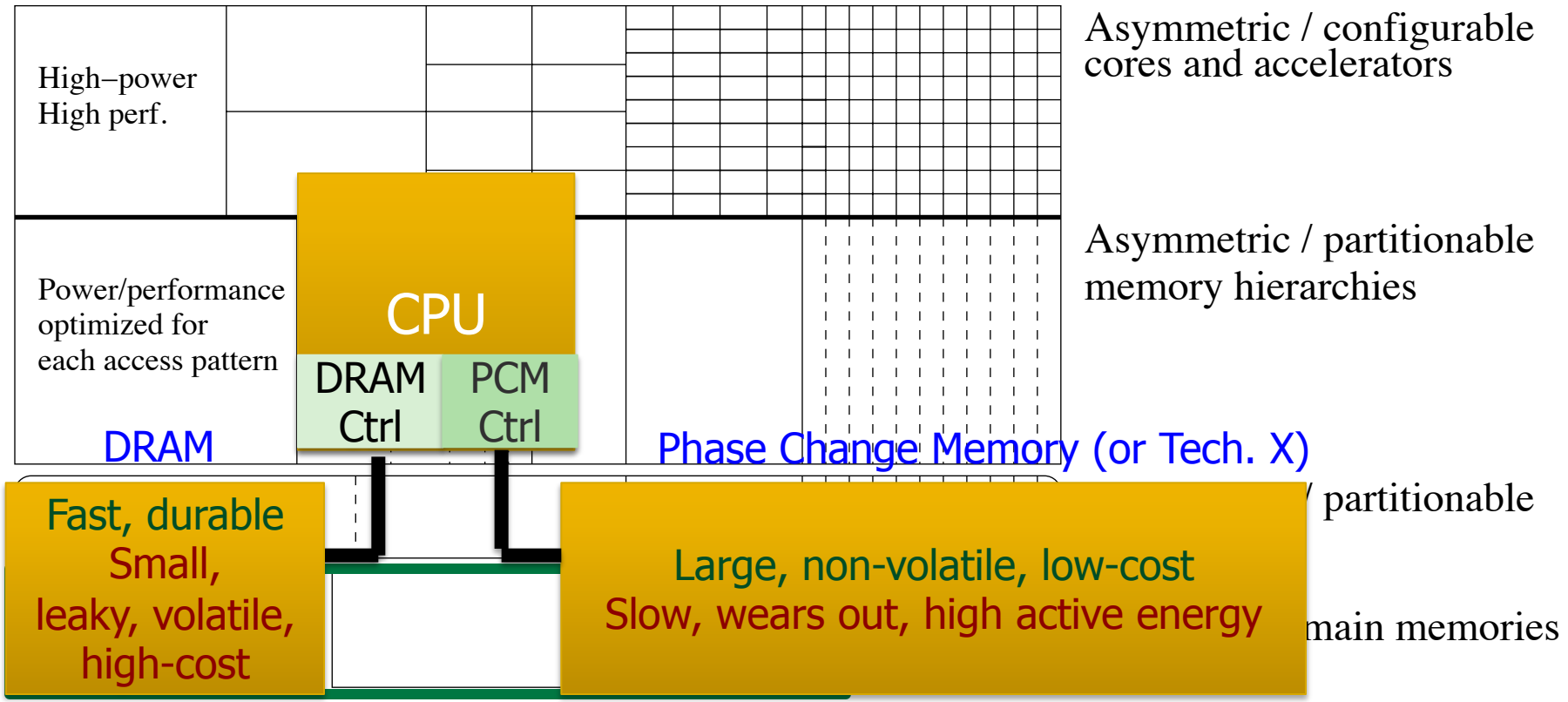
Asymmetric / partitionable memory hierarchies

Asymmetric / partitionable interconnect

Asymmetric main memories

- 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