Reminder: Review Assignments

- Due: Friday, September 21, 11:59pm.


Other Recommended Papers


Last Lecture

- An Early History of Multi-Core
- Homogeneous Multi-Core Evolution
- From Symmetry to Asymmetry
Today

- More on Asymmetric Multi-Core
- And, Asymmetry in General
Asymmetric Multi-Core
Review: 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
Review: 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 all cores for high throughput (14 units)
Review: EPI Throttling

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

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

- **Idea:** For a fixed power budget:
  - Run sequential phases on high-EPI processor
  - Run parallel phases on multiple low-EPI processors
Review: EPI Throttling via DVFS

- DVFS: Dynamic voltage frequency scaling

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

- In phases of high thread parallelism
  - Run many cores at low supply voltage and low frequency
EPI Throttling (Annavaram et al., ISCA’05)

- **Static AMP**
  - Duty cycles set once prior to program run
  - Parallel phases run on 3P/1.25GHz
  - Sequential phases run on 1P/2GHz
  - Affinity guarantees sequential on 1P and parallel on 3
  - Benchmarks that rapidly transition between sequential and parallel phases

- **Dynamic AMP**
  - Duty cycle changes during program run
  - Parallel phases run on all or a subset of four processors
  - Sequential phases of execution on 1P/2GHz
  - Benchmarks with long sequential and parallel phases
EPI Throttling (Annavaram et al., ISCA ’05)

- Evaluation on Base SMP: 4-way 2GHz Xeon, 2MB L3, 4GB Memory

- Hand-modified programs
  - OMP threads set to 3 for static AMP
  - Calls to set affinity in each thread for static AMP
  - Calls to change duty cycle and to set affinity in dynamic AMP

<table>
<thead>
<tr>
<th>AMP Configuration</th>
<th>Programs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Static AMP: 1P/2GHz or 3P/1.25GHz</td>
<td>wupwise, swim, mgrid, equake, fma3d, art, ammp, BLAST, HMMER</td>
</tr>
<tr>
<td>Dynamic AMP: 1P/2GHz to 4P/1GHz</td>
<td>applu, apsi, FFTW, TPC-H</td>
</tr>
</tbody>
</table>
EPI Throttling (Annavaram et al., ISCA’05)

- Frequency boosting AMP improves performance compared to 4-way SMP for many applications
EPI Throttling

- Why does Frequency Boosting (FB) AMP not always improve performance?

- Loss of throughput in static AMP (only 3 processors in parallel portion)
  - Is this really the best way of using FB-AMP?

- Rapid transitions between serial and parallel phases
  - Data/thread migration and throttling overhead

- Boosting frequency does not help memory-bound phases
Review So Far

- **Symmetric Multicore**
  - Evolution of Sun’s and IBM’s Multicore systems and design choices
  - Niagara, Niagara 2, ROCK
  - IBM POWERx

- **Asymmetric multicore**
  - Motivation
  - Functional vs. Performance Asymmetry
  - Static vs. Dynamic Asymmetry
  - EPI Throttling
Design Tradeoffs in ACMP (I)

- **Hardware Design Effort vs. Programmer Effort**
  - ACMP requires more design effort
  + Performance becomes less dependent on length of the serial part
  + Can reduce programmer effort: Serial portions are not as bad for performance with ACMP

- **Migration Overhead vs. Accelerated Serial Bottleneck**
  + Performance gain from faster execution of serial portion
  - Performance loss when architectural state is migrated/switched in when the master changes
    - Can be alleviated with multithreading and hidden by long serial portion
  - Serial portion incurs cache misses when it needs data generated by the parallel portion
  - Parallel portion incurs cache misses when it needs data generated by the serial portion
Design Tradeoffs in ACMP (II)

- Fewer threads vs. accelerated serial bottleneck
  - Performance gain from accelerated serial portion
  - Performance loss due to unavailability of L threads in parallel portion

- This need not be the case → Large core can implement Multithreading to improve parallel throughput
- As the number of cores (threads) on chip increases, fractional loss in parallel performance decreases
Uses of Asymmetry

- So far:
  - Improvement in serial performance (sequential bottleneck)

- What else can we do with asymmetry?
  - Energy reduction?
  - Energy/performance tradeoff?
  - Improvement in parallel portion?
Use of Asymmetry for Energy Efficiency


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

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

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

<table>
<thead>
<tr>
<th>Core</th>
<th>Peak power (Watts)</th>
<th>Average power (Watts)</th>
<th>Performance (norm. IPC)</th>
</tr>
</thead>
<tbody>
<tr>
<td>EV4</td>
<td>4.97</td>
<td>3.73</td>
<td>1.00</td>
</tr>
<tr>
<td>EV5</td>
<td>9.83</td>
<td>6.88</td>
<td>1.30</td>
</tr>
<tr>
<td>EV6</td>
<td>17.8</td>
<td>10.68</td>
<td>1.87</td>
</tr>
<tr>
<td>EV8</td>
<td>92.88</td>
<td>46.44</td>
<td>2.14</td>
</tr>
</tbody>
</table>
Use of Asymmetry for Energy Efficiency

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

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

- Mutual Exclusion:
  - Threads are not allowed to update shared data concurrently

- Accesses to shared data are encapsulated inside critical sections

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

- **Idea**: Ship critical sections to a large core

Use of ACMP to Improve Parallel Portion Performance


Asymmetry Everywhere
The Setting

- **Hardware resources are shared among many threads/apps in a many-core system**
  - Cores, caches, interconnects, memory, disks, power, lifetime, ...

- Management of these resources is a very difficult task
  - When optimizing parallel/multiprogrammed workloads
  - Threads interact unpredictably/unfairly in shared resources

- **Power/energy consumption** is arguably the most valuable shared resource
  - Main limiter to efficiency and performance
Shield the Programmer from Shared Resources

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

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

- Future computer architectures should be designed to
  - Minimize programmer effort to optimize (parallel) programs
  - Maximize runtime system’s effectiveness in automatic shared resource management
Shared Resource Management: Goals

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

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

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

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
</tr>
<tr>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
</tr>
<tr>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
</tr>
<tr>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
</tr>
</tbody>
</table>

Symmetric

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>C1</td>
<td>C2</td>
</tr>
<tr>
<td>C3</td>
<td></td>
</tr>
<tr>
<td>C4</td>
<td>C4</td>
</tr>
<tr>
<td>C5</td>
<td>C5</td>
</tr>
</tbody>
</table>

Asymmetric

**Symmetric: One size fits all**
- Energy and performance suboptimal for different phase behaviors

**Asymmetric: Enables tradeoffs and customization**
- Processing requirements vary across applications and phases
- Execute code on best-fit resources (minimal energy, adequate perf.)
**Thought Experiment: Asymmetry Everywhere**

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

<table>
<thead>
<tr>
<th>High-power High perf.</th>
<th>Asymmetric / configurable cores and accelerators</th>
</tr>
</thead>
<tbody>
<tr>
<td>Power/performance optimized for each access pattern</td>
<td>Asymmetric / partitionable memory hierarchies</td>
</tr>
<tr>
<td>Different technologies Power characteristics</td>
<td>Asymmetric / partitionable interconnect</td>
</tr>
<tr>
<td></td>
<td>Asymmetric main memories</td>
</tr>
</tbody>
</table>
Thought Experiment: Asymmetry Everywhere

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

| High-power |
| High perf. |

| Power/performance optimized for each access pattern |
| Asymmetric / configurable cores and accelerators |
| Asymmetric / partitionable memory hierarchies |

| Different technologies |
| Power characteristics |
| Asymmetric / partitionable interconnect |
| Asymmetric main memories |
Thought Experiment: Asymmetry Everywhere

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

<table>
<thead>
<tr>
<th>High-powerperf. High perf.</th>
<th>Asymmetric / configurable cores and accelerators</th>
</tr>
</thead>
<tbody>
<tr>
<td>Power/performanceoptimized for each access pattern</td>
<td>Asymmetric / partitionable memory hierarchies</td>
</tr>
<tr>
<td>Different technologiesPower characteristics</td>
<td>Asymmetric / partitionable interconnect</td>
</tr>
<tr>
<td></td>
<td>Asymmetric main memories</td>
</tr>
</tbody>
</table>
Thought Experiment: Asymmetry Everywhere

- Design each **hardware** resource with asymmetric, (re-)configurable, partitionable components

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

- **Morph software components** to match asymmetric HW components
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?
  - To characterize a phase and match it to best-fit components
  - 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
Summary of the Thought Experiment

- Need to minimize energy while satisfying performance requirements
  - While also minimizing programmer effort

- **Asymmetry** key to energy/performance/reliability tradeoffs

- Design systems with many asymmetric/partitionable components
  - Many types of cores, memories, interconnects, ...
  - Partitionable/configurable components, customized accelerators on chip

- Provide all-automatic resource management
  - Impose structure: HW and SW cooperatively map phases to components
  - Dynamically stitch together the system that best fits the running tasks

- Programmer does not need to worry about complex resource sharing
Outline

- How Do We Get There: Examples

- Accelerated Critical Sections (ACS)
- Bottleneck Identification and Scheduling (BIS)
- Staged Execution and Data Marshaling

- Asymmetry in Memory
  - Thread Cluster Memory Scheduling
  - Heterogeneous DRAM+NVM Main Memory
### 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

<table>
<thead>
<tr>
<th>Power/performance optimized for each access pattern</th>
<th>Asymmetric / configurable cores and accelerators</th>
</tr>
</thead>
<tbody>
<tr>
<td>High-power High perf.</td>
<td>Asymmetric / partitionable memory hierarchies</td>
</tr>
<tr>
<td>Different technologies</td>
<td>Asymmetric / partitionable interconnect</td>
</tr>
<tr>
<td>Power characteristics</td>
<td>Asymmetric main memories</td>
</tr>
</tbody>
</table>

- **High–power High perf.**

- **Power/performance optimized for each access pattern**

- **Different technologies**

- **Power characteristics**
Exploiting Asymmetry: Simple Examples

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

- Asymmetric / configurable cores and accelerators
- Asymmetric / partitionable memory hierarchies
- Asymmetric / partitionable interconnect
- Asymmetric main memories
Exploiting Asymmetry: Simple Examples

  - Higher performance and energy-efficiency than symmetric/free-for-all
Exploiting Asymmetry: Simple Examples

<table>
<thead>
<tr>
<th>High-power</th>
<th>Asymmetric / configurable cores and accelerators</th>
</tr>
</thead>
<tbody>
<tr>
<td>High perf.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Power/performance optimized for each access pattern</th>
<th>Asymmetric / partitionable memory hierarchies</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Different technologies Power characteristics</th>
<th>Asymmetric / partitionable interconnect</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Asymmetric main memories</td>
</tr>
</tbody>
</table>

- 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
Outline

- How Do We Get There: Examples
  - Accelerated Critical Sections (ACS)
  - Bottleneck Identification and Scheduling (BIS)
  - Staged Execution and Data Marshaling

- Asymmetry in Memory
  - Thread Cluster Memory Scheduling
  - Heterogeneous DRAM+NVM Main Memory
Serialized Code Sections in Parallel Applications

- Multithreaded applications:
  - Programs split into threads

- Threads execute concurrently on multiple cores

- Many programs cannot be parallelized completely

- Serialized code sections:
  - Reduce performance
  - Limit scalability
  - Waste energy
Causes of Serialized Code Sections

- Sequential portions (Amdahl’s “serial part”)
- Critical sections
- Barriers
- Limiter stages in pipelined programs
Bottlenecks in Multithreaded Applications

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

Examples:

- **Amdahl’s serial portions**
  - Only one thread exists → on the critical path

- **Critical sections**
  - Ensure mutual exclusion → likely to be on the critical path if contended

- **Barriers**
  - Ensure all threads reach a point before continuing → the latest thread arriving is on the critical path

- **Pipeline stages**
  - Different stages of a loop iteration may execute on different threads, slowest stage makes other stages wait → on the critical path
Critical Sections

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

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

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

- Critical Section
- Access Open Tables Cache
- Open database tables
- Perform the operations
- Parallel
Contestion for Critical Sections

12 iterations, 33% instructions inside the critical section

P = 1

P = 2

P = 3

P = 4

33% in critical section

Critical Section
Parallel
Idle
Contestation for Critical Sections

12 iterations, 33% instructions inside the critical section

Accelerating critical sections increases performance and scalability.

Critical Section
Parallel
Idle

Critical Section Accelerated by 2x

SAFARI
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.

![Graph showing speedup vs. chip area (cores)](image-url)
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)
- Provide one large core and many small cores
- Execute parallel part on small cores for high throughput
- Accelerate serialized sections using the large core
Conventional ACMP

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

EnterCS()
PriorityQ.insert(…)
LeaveCS()
Accelerated Critical Sections (ACS)

- Accelerate Amdahl’s serial part \textbf{and critical sections} using the large core
Accelerated Critical Sections (ACS)

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

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

- ISA extensions
  - CSCALL  LOCK_ADDR, TARGET_PC
  - CSRET  LOCK_ADDR

- Compiler/Library inserts CSCALL/CSRET

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

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

Small Core

A = compute()

LOCK X
result = CS(A)
UNLOCK X

print result

Small Core

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

... ...

... ...

... ...

... ...

... ...

Large Core

... ...

Waiting in Critical Section Request Buffer (CSRB)

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

CSDONE Response

Safari
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

We will get back to this
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
Chip Area = 32 small cores
SCMP = 32 small cores
ACMP = 1 large and 28 small cores

Equal-area comparison
Number of threads = Best threads

Coarse-grain locks
Fine-grain locks
Equal-Area Comparisons

Number of threads = No. of cores

Chip Area (small cores)
ACS Summary

- Critical sections reduce performance and limit scalability

- Accelerate critical sections by executing them on a powerful core

- ACS reduces average execution time by:
  - 34% compared to an equal-area SCMP
  - 23% compared to an equal-area ACMP

- ACS improves scalability of 7 of the 12 workloads

- Generalizing the idea: Accelerate all bottlenecks ("critical paths") by executing them on a powerful core
Outline

- How Do We Get There: Examples
- Accelerated Critical Sections (ACS)
- Bottleneck Identification and Scheduling (BIS)
- Staged Execution and Data Marshaling
- Asymmetry in Memory
  - Thread Cluster Memory Scheduling
  - Heterogeneous DRAM+NVM Main Memory
**BIS Summary**

- **Problem**: Performance and scalability of multithreaded applications are limited by serializing bottlenecks
  - different types: critical sections, barriers, slow pipeline stages
  - importance (criticality) of a bottleneck can change over time

- **Our Goal**: Dynamically identify the most important bottlenecks and accelerate them
  - How to identify the most critical bottlenecks
  - How to efficiently accelerate them

- **Solution**: Bottleneck Identification and Scheduling (BIS)
  - Software: annotate bottlenecks (BottleneckCall, BottleneckReturn) and implement waiting for bottlenecks with a special instruction (BottleneckWait)
  - Hardware: identify bottlenecks that cause the most thread waiting and accelerate those bottlenecks on large cores of an asymmetric multi-core system

- Improves multithreaded application performance and scalability, outperforms previous work, and performance improves with more cores
Bottlenecks in Multithreaded Applications

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

Examples:

- **Amdahl’s serial portions**
  - Only one thread exists → on the critical path

- **Critical sections**
  - Ensure mutual exclusion → likely to be on the critical path if contended

- **Barriers**
  - Ensure all threads reach a point before continuing → the latest thread arriving is on the critical path

- **Pipeline stages**
  - Different stages of a loop iteration may execute on different threads, slowest stage makes other stages wait → on the critical path
Observation: Limiting Bottlenecks Change Over Time

A = full linked list; B = empty linked list

repeat

Lock A
- Traverse list A
- Remove X from A
Unlock A

Compute on X

Lock B
- Traverse list B
- Insert X into B
Unlock B

until A is empty

32 threads

Contestion (# of threads waiting)

Lock A is limiter

Lock B is limiter
Limiting Bottlenecks Do Change on Real Applications

MySQL running Sysbench queries, 16 threads

Contention (# of threads waiting)

time [Mcycles]
Previous Work on Bottleneck Acceleration

- Asymmetric CMP (ACMP) proposals [Annavaram+, ISCA’05] [Morad+, Comp. Arch. Letters’06] [Suleman+, Tech. Report’07]
  - Accelerate only the Amdahl’s bottleneck

- Accelerated Critical Sections (ACS) [Suleman+, ASPLOS’09]
  - Accelerate only critical sections
  - Does not take into account importance of critical sections

- Feedback-Directed Pipelining (FDP) [Suleman+, PACT’10 and PhD thesis’11]
  - Accelerate only stages with lowest throughput
  - Slow to adapt to phase changes (software based library)

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

Our goal: general mechanism to identify performance-limiting bottlenecks of any type and accelerate them on an ACMP
Bottleneck Identification and Scheduling (BIS)

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

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

Compiler/Library/Programmer

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

Hardware

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

Binary containing BIS instructions
Critical Sections: Code Modifications

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

BottleneckCall bid, targetPC

BottleneckWait bid, watch_addr

targetPC:

BottleneckReturn bid

Used to keep track of waiting cycles
Used to enable acceleration
Barriers: Code Modifications

... **BottleneckCall** *bid*, targetPC
enter barrier
while not all threads in barrier
    **BottleneckWait** *bid*, watch_addr
exit barrier
...

targetPC: code running for the barrier
...
    **BottleneckReturn** *bid*
Pipeline Stages: Code Modifications

**BottleneckCall** `bid`, `targetPC`

...  

**targetPC:**  
while not done  
  while empty queue  
    **BottleneckWait** `prev_bid`  
  dequeue work  
  do the work ...  
  while full queue  
    **BottleneckWait** `next_bid`  
  enqueue next work  
**BottleneckReturn** `bid`
1. Annotate *bottleneck* code
2. Implements *waiting* for bottlenecks

Binary containing **BIS instructions**

**Compiler/Library/Programmer**

**Hardware**

1. Measure *thread waiting cycles (TWC)* for each bottleneck
2. Accelerate bottleneck(s) with the highest TWC
We did not cover the following slides in lecture. These are for your preparation for the next lecture.
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

![Diagram showing cores and core configurations](image)
Bottleneck Identification and Scheduling (BIS)

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

**Compiler/Library/Programmer**

- Binary containing **BIS instructions**

**Hardware**

1. Measure *thread waiting cycles* (*TWC*) for each bottleneck
2. Accelerate bottleneck(s) with the highest TWC
Determining Thread Waiting Cycles for Each Bottleneck

Small Core 1

BottleneckWait x4500

Small Core 2

BottleneckWait x4500

Large Core 0

Bottleneck Table (BT)

Bid=x4500, waiters=1, twc = 5
Bottleneck Identification and Scheduling (BIS)

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

Binary containing BIS instructions

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

- **Small Core 1**
  - **BottleneckCall** \(bid=x4700, pc, sp, core1\)
  - Execute remotely

- **Large Core 0**
  - **BottleneckReturn** \(x4700\)

- **Acceleration Index Table (AIT)**
  - \(bid=x4700,\) large core 0

- **Scheduling Buffer (SB)**
  - \(bid=x4700, pc, sp, core1\)
  - \(\text{twc} < \text{Threshold}\)
  - \(\text{twc} > \text{Threshold}\)

- **Bottleneck Table (BT)**
  - \(bid=x4600,\) \(\text{twc}=100\)
  - \(bid=x4700,\) \(\text{twc}=10000\)

- **Small Core 2**
  - AIT
  - \(bid=x4700,\) large core 0
BIS Mechanisms

- Basic mechanisms for BIS:
  - Determining Thread Waiting Cycles ✓
  - Accelerating Bottlenecks ✓

- Mechanisms to improve performance and generality of BIS:
  - Dealing with false serialization
  - Preemptive acceleration
  - Support for multiple large cores
False Serialization and Starvation

**Observation:** Bottlenecks are picked from Scheduling Buffer in Thread Waiting Cycles order

**Problem:** An independent bottleneck that is ready to execute has to wait for another bottleneck that has higher thread waiting cycles → **False serialization**

**Starvation:** Extreme false serialization

**Solution:** Large core detects when a bottleneck is ready to execute in the Scheduling Buffer but it cannot → sends the bottleneck back to the small core
Preemptive Acceleration

- **Observation:** A bottleneck executing on a small core can become the bottleneck with the highest thread waiting cycles.

- **Problem:** This bottleneck should really be accelerated (i.e., executed on the large core).

- **Solution:** The Bottleneck Table detects the situation and sends a preemption signal to the small core. Small core:
  - saves register state on stack, ships the bottleneck to the large core.

- **Main acceleration mechanism for barriers and pipeline stages.**
Support for Multiple Large Cores

- **Objective:** to accelerate independent bottlenecks

- Each large core has its own Scheduling Buffer (shared by all of its SMT threads)

- Bottleneck Table assigns each bottleneck to a fixed large core context to
  - preserve cache locality
  - avoid busy waiting

- Preemptive acceleration extended to send multiple instances of a bottleneck to different large core contexts
Hardware Cost

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

- **Off the critical path**

- **Total storage cost for 56-small-cores, 2-large-cores < 19 KB**
BIS Performance Trade-offs

- Bottleneck identification:
  - Small cost: BottleneckWait instruction and Bottleneck Table

- Bottleneck acceleration on an ACMP (execution migration):
  - 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 to not be on critical path)
Methodology

- **Workloads:** 8 critical section intensive, 2 barrier intensive and 2 pipeline-parallel applications
  - Data mining kernels, scientific, database, web, networking, specjbb

- **Cycle-level multi-core x86 simulator**
  - 8 to 64 small-core-equivalent area, 0 to 3 large cores, SMT
  - 1 large core is area-equivalent to 4 small cores

- **Details:**
  - Large core: 4GHz, out-of-order, 128-entry ROB, 4-wide, 12-stage
  - Small core: 4GHz, in-order, 2-wide, 5-stage
  - Private 32KB L1, private 256KB L2, shared 8MB L3
  - On-chip interconnect: Bi-directional ring, 2-cycle hop latency
BIS Comparison Points (Area-Equivalent)

- **SCMP (Symmetric CMP)**
  - All small cores
  - Results in the paper

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

- **ACS (Accelerated Critical Sections)**
  - Accelerates only critical sections and Amdahl’s serial portions
  - Applicable to multithreaded workloads (iplookup, mysql, specjbb, sqlite, tsp, webcache, mg, ft)

- **FDP (Feedback-Directed Pipelining)**
  - Accelerates only slowest pipeline stages
  - Applicable to pipeline-parallel workloads (rank, pagemine)
BIS Performance Improvement

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

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

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

Performance increases with:

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

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

- Serializing bottlenecks of different types limit performance of multithreaded applications: Importance changes over time

- BIS is a hardware/software cooperative solution:
  - Dynamically identifies bottlenecks that cause the most thread waiting and accelerates them on large cores of an ACMP
  - Applicable to critical sections, barriers, pipeline stages

- BIS improves application performance and scalability:
  - 15% speedup over ACS/FDP
  - Can accelerate multiple independent critical bottlenecks
  - Performance benefits increase with more cores

- Provides comprehensive fine-grained bottleneck acceleration for future ACMPs with little or no programmer effort