Required Readings

➢ Required Reading Assignment:
  • Suleman et al., “Accelerating Critical Section Execution with Asymmetric Multi-Core Architectures,” ASPLOS 2009.

➢ Recommended References:
  • Joao et al., “Bottleneck Identification and Scheduling in Multithreaded Applications,” ASPLOS 2012.
Heterogeneity (Asymmetry)
Heterogeneity (Asymmetry) → Specialization

- Heterogeneity and asymmetry have the same meaning
  - Contrast with homogeneity and symmetry

- Heterogeneity is a very general system design concept (and life concept, as well)

- Idea: Instead of having multiple instances of the same “resource” to be the same (i.e., homogeneous or symmetric), design some instances to be different (i.e., heterogeneous or asymmetric)

- Different instances can be optimized to be more efficient in executing different types of workloads or satisfying different requirements/goals
  - Heterogeneity enables specialization/customization
Why Asymmetry in Design? (I)

- Different workloads executing in a system can have different behavior
  - Different applications can have different behavior
  - Different execution phases of an application can have different behavior
  - The same application executing at different times can have different behavior (due to input set changes and dynamic events)
  - E.g., locality, predictability of branches, instruction-level parallelism, data dependencies, serial fraction, bottlenecks in parallel portion, interference characteristics, ...

- Systems are designed to satisfy different metrics at the same time
  - There is almost never a single goal in design, depending on design point
  - E.g., Performance, energy efficiency, fairness, predictability, reliability, availability, cost, memory capacity, latency, bandwidth, ...
Why Asymmetry in Design? (II)

- Problem: *Symmetric design is one-size-fits-all*
- It tries to fit a single-size design to all workloads and metrics

- It is very difficult to come up with a single design
  - that satisfies all workloads even for a single metric
  - that satisfies all design metrics at the same time

- This holds true for different system components, or resources
  - Cores, caches, memory, controllers, interconnect, disks, servers, ...
  - Algorithms, policies, ...
Asymmetry Enables Customization

- **Symmetric: One size fits all**
  - Energy and performance suboptimal for different “workload” behaviors

- **Asymmetric: Enables customization and adaptation**
  - Processing requirements vary across workloads (applications and phases)
  - Execute code on best-fit resources (minimal energy, adequate perf.)

---

**Symmetric**

\[
\begin{array}{cccc}
  c & c & c & c \\
  c & c & c & c \\
  c & c & c & c \\
  c & c & c & c \\
\end{array}
\]

**Asymmetric**

\[
\begin{array}{c|c}
  C1 & C2 \\
  \hline
  C4 & C4 \\
  C4 & C4 \\
  C5 & C5 \\
  C5 & C5 \\
\end{array}
\]
We Have Already Seen Examples (Before)

- CRAY-1 design: scalar + vector pipelines
- Modern processors: scalar instructions + SIMD extensions
- Decoupled Access Execute: access + execute processors

- Thread Cluster Memory Scheduling: different memory scheduling policies for different thread clusters
- RAIDR: Heterogeneous refresh rates in DRAM
- Heterogeneous-Latency DRAM (Tiered Latency DRAM)
- Hybrid memory systems
  - DRAM + Phase Change Memory
  - Fast, Costly DRAM + Slow, Cheap DRAM
  - Reliable, Costly DRAM + Unreliable, Cheap DRAM
An Example Asymmetric Design: CRAY-1

- CRAY-1

- Scalar and vector modes
- 8 64-element vector registers
- 64 bits per element
- 16 memory banks
- 8 64-bit scalar registers
- 8 24-bit address registers
Remember: Hybrid Memory Systems

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


Yoon, Meza et al., “Row Buffer Locality Aware Caching Policies for Hybrid Memories,” ICCD 2012 Best Paper Award.
Remember: Throughput vs. Fairness

**Throughput biased approach**
Prioritize less memory-intensive threads

**Fairness biased approach**
Take turns accessing memory

Good for throughput

- less memory intensive
- thread A
- thread B
- thread C
- higher priority

Does not starve

- thread C
- thread A
- thread B

starvation $\rightarrow$ unfairness

not prioritized $\rightarrow$ reduced throughput

Single policy for all threads is insufficient

Remember: Achieving the Best of Both Worlds

**For Throughput**

- Prioritize memory-non-intensive threads

**For Fairness**

- Unfairness caused by memory-intensive being prioritized over each other
  - Shuffle thread ranking
- Memory-intensive threads have different vulnerability to interference
  - Shuffle asymmetrically

Thread Cluster Memory Scheduling [Kim+ MICRO’10]

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

Remember: Heterogeneous Retention Times in DRAM

64-128ms

>256ms

128-256ms

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

Long Bitline

Short Bitline

Faster

Smaller

Approximating the Best of Both Worlds

Long Bitline
Small Area
High Latency

Our Proposal

Short Bitline
Large Area
Low Latency

Need Isolation
Add Isolation Transistors
Short Bitline ➔ Fast

Approximating the Best of Both Worlds

Long Bitline Tiered-Latency DRAM Short Bitline

Small Area Small Area Large Area

High Latency Low Latency Low Latency

Small area using long bitline

Heterogeneous Interconnect in Tilera

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

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

Aside: Examples from Life

- Heterogeneity is abundant in life
  - both in nature and human-made components

- Humans are heterogeneous
- Cells are heterogeneous → specialized for different tasks
- Organs are heterogeneous
- Cars are heterogeneous
- Buildings are heterogeneous
- Rooms are heterogeneous
- ...
General-Purpose vs. Special-Purpose

- Asymmetry is a way of enabling specialization

- It bridges the gap between purely general purpose and purely special purpose
  - Purely general purpose: Single design for every workload or metric
  - Purely special purpose: Single design per workload or metric
  - Asymmetric: Multiple sub-designs optimized for sets of workloads/metrics and glued together

- The goal of a good asymmetric design is to get the best of both general purpose and special purpose
Asymmetry Advantages and Disadvantages

Advantages over Symmetric Design
+ Can enable optimization of multiple metrics
+ Can enable better adaptation to workload behavior
+ Can provide special-purpose benefits with general-purpose usability/flexibility

Disadvantages over Symmetric Design
- Higher overhead and more complexity in design, verification
- Higher overhead in management: scheduling onto asymmetric components
- Overhead in switching between multiple components can lead to degradation
Modern processors integrate general purpose cores and GPUs

- CPU-GPU systems
- Heterogeneity in execution models
Three Key Problems in Future Systems

- Memory system
  - Applications are increasingly data intensive
  - Data storage and movement limits performance & efficiency

- Efficiency (performance and energy) \(\rightarrow\) scalability
  - Enables scalable systems \(\rightarrow\) new applications
  - Enables better user experience \(\rightarrow\) new usage models

- Predictability and robustness

Asymmetric Designs Can Help Solve These Problems
Commercial Asymmetric Design Examples

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

- Heterogeneous agents: CPUs, GPUs, and HWAs
- Heterogeneous memories: Fast vs. Slow DRAM
- Heterogeneous interconnects: Control, Data, Synchronization
Multi-Core Design: An Asymmetric Perspective
Many Cores on Chip

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

- AMD Barcelona
  4 cores

- Intel Core i7
  8 cores

- IBM Cell BE
  8+1 cores

- Sun Niagara II
  8 cores

- Nvidia Fermi
  448 “cores”

- IBM POWER7
  8 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

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


- **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

Open database tables

Perform the operations

Critical Section

Access Open Tables Cache

Parallel

Asymmetric

Today

Chip Area (cores)

Speedup
Demands in Different Code Sections

- What we want:
  - In a serialized code section → one powerful "large" core
  - In a parallel code section → many wimpy "small" cores

- These two conflict with each other:
  - If you have a single powerful core, you cannot have many cores
  - A small core is much more energy and area efficient than a large core
“Large” vs. “Small” Cores

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

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

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


<table>
<thead>
<tr>
<th></th>
<th>Large core</th>
<th>Small core</th>
</tr>
</thead>
<tbody>
<tr>
<td>Microarchitecture</td>
<td>Out-of-order, 128-256 entry ROB</td>
<td>In-order</td>
</tr>
<tr>
<td>Width</td>
<td>3-4</td>
<td>1</td>
</tr>
<tr>
<td>Pipeline depth</td>
<td>20-30</td>
<td>5</td>
</tr>
<tr>
<td>Normalized performance</td>
<td>5-8x</td>
<td>1x</td>
</tr>
<tr>
<td>Normalized power</td>
<td>20-50x</td>
<td>1x</td>
</tr>
<tr>
<td>Normalized energy/instruction</td>
<td>4-6x</td>
<td>1x</td>
</tr>
</tbody>
</table>
Meet Large: IBM POWER4


- A symmetric multi-core chip...

- Two powerful cores
IBM POWER4

- 2 cores, out-of-order execution
- 100-entry instruction window in each core
- 8-wide instruction fetch, issue, execute
- Large, local+global hybrid branch predictor
- 1.5MB, 8-way L2 cache
- Aggressive stream based prefetching
IBM POWER5


Figure 4. Power5 instruction data flow (BXU = branch execution unit and CRL = condition register logical execution unit).
Meet Small: Sun Niagara (UltraSPARC T1)

Niagara Core

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

- What we want:
  - In a serialized code section → one powerful “large” core
  - In a parallel code section → many wimpy “small” cores

- These two conflict with each other:
  - If you have a single powerful core, you cannot have many cores
  - A small core is much more energy and area efficient than a large core

- Can we get the best of both worlds?
Performance vs. Parallelism

Assumptions:

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

2. Large core takes an area budget of 4 and has performance of 2
## Tile-Large Approach

- **Tile 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-Large”**

<table>
<thead>
<tr>
<th>Large core</th>
<th>Large core</th>
</tr>
</thead>
<tbody>
<tr>
<td>Large core</td>
<td>Large core</td>
</tr>
</tbody>
</table>
Tile-Small Approach

- 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)

<table>
<thead>
<tr>
<th>Large core</th>
<th>Large core</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Large core</td>
<td>Large core</td>
</tr>
</tbody>
</table>

- “Tile-Large”

<table>
<thead>
<tr>
<th>Small core</th>
<th>Small core</th>
<th>Small core</th>
<th>Small core</th>
</tr>
</thead>
<tbody>
<tr>
<td>Small core</td>
<td>Small core</td>
<td>Small core</td>
<td>Small core</td>
</tr>
<tr>
<td>Small core</td>
<td>Small core</td>
<td>Small core</td>
<td>Small core</td>
</tr>
<tr>
<td>Small core</td>
<td>Small core</td>
<td>Small core</td>
<td>Small core</td>
</tr>
</tbody>
</table>

- “Tile-Small”

<table>
<thead>
<tr>
<th>Large core</th>
<th>Small core</th>
<th>Small core</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- ACMP

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

Single thread $\rightarrow$ Large core

ACMP Approach
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

<table>
<thead>
<tr>
<th></th>
<th>Large Core</th>
<th>Large Core</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Large Core</td>
<td>Large Core</td>
</tr>
<tr>
<td><strong>“Tile-Large”</strong></td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td><strong>“Tile-Small”</strong></td>
<td>0</td>
<td>16</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Large Core</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ACMP</strong></td>
<td>1</td>
</tr>
<tr>
<td><strong>Serial Performance</strong></td>
<td>2</td>
</tr>
<tr>
<td><strong>Parallel Throughput</strong></td>
<td>2 x 4 = 8</td>
</tr>
<tr>
<td><strong>Area-budget</strong></td>
<td>16 small cores</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Large Core</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Small Cores</strong></td>
<td>0</td>
</tr>
<tr>
<td><strong>ACMP</strong></td>
<td>12</td>
</tr>
<tr>
<td><strong>Serial Performance</strong></td>
<td>2</td>
</tr>
<tr>
<td><strong>Parallel Throughput</strong></td>
<td>1 x 16 = 16</td>
</tr>
<tr>
<td><strong>Area-budget</strong></td>
<td>16 small cores</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Large Core</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Small Cores</strong></td>
<td>16</td>
</tr>
<tr>
<td><strong>ACMP</strong></td>
<td>14</td>
</tr>
<tr>
<td><strong>Serial Performance</strong></td>
<td>2</td>
</tr>
<tr>
<td><strong>Parallel Throughput</strong></td>
<td>1x2 + 1x12 = 14</td>
</tr>
</tbody>
</table>
Amdahl’s Law Modified

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

\[
\text{Speedup} = \frac{1}{1 - f \cdot \frac{1}{X} + \frac{f}{S + X \cdot L}}
\]
Caveats of Parallelism, Revisited

- **Amdahl’s Law**
  - $f$: Parallelizable fraction of a program
  - $N$: Number of processors

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


- **Maximum speedup limited by serial portion:** Serial bottleneck

- **Parallel portion is usually not perfectly parallel**
  - **Synchronization** overhead (e.g., updates to shared data)
  - **Load imbalance** overhead (imperfect parallelization)
  - **Resource sharing** overhead (contention among $N$ processors)
Accelerating Parallel Bottlenecks

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

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

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

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

12 iterations, 33% instructions inside the critical section

- P = 4
- P = 3
- P = 2
- P = 1

Critical Section
Parallel
Idle

33% in critical section
Contestion for Critical Sections

12 iterations, 33% instructions inside the critical section

- Critical Section
- Parallel
- Idle

Accelerating critical sections increases performance and scalability

Critical Section Accelerated by 2x
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

Accelerated Critical Sections

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)

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

Coarse-grain locks
Fine-grain locks

Deviation analysis for Speedup over SCMP
Accelerating Sequential Kernels
Accelerating Critical Sections

Speedup over SCMP

Accelerating Sequential Kernels
Accelerating Critical Sections
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*
Bottleneck Identification and Scheduling

Jose A. Joao, M. Aater Suleman, Onur Mutlu, and Yale N. Patt,
"Bottleneck Identification and Scheduling in Multithreaded Applications"
Bottlenecks in Multithreaded Applications

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

Examples:

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

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

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

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

A=full linked list; B=empty linked list
repeat
  Lock A
  Traverse list A
  Remove X from A
  Unlock A
  Compute on X
  Lock B
  Traverse list B
  Insert X into B
  Unlock B
until A is empty

32 threads

Lock A is limiter

Lock B is limiter
Limiting Bottlenecks Do Change on Real Applications

MySQL running Sysbench queries, 16 threads
Bottleneck Identification and Scheduling (BIS)

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

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

1. Annotate *bottleneck* code
2. Implement *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
Critical Sections: Code Modifications

```plaintext
...  
BottleneckCall bid, targetPC  
...  
while cannot acquire lock  
...  
Wait loop for watch_addr  
...  
while cannot acquire lock  
...  
release lock  
...  
release lock  

BottleneckWait bid, watch_addr  

BottleneckReturn bid
```

Used to enable acceleration
Barriers: Code Modifications

...  
\textbf{BottleneckCall} \textit{bid}, targetPC  
enter barrier  
while not all threads in barrier  
\textbf{BottleneckWait} \textit{bid}, watch_addr  
extit{exit barrier}  
...  

\textbf{targetPC:} \textit{code running for the barrier}  
...  
...  
\textbf{BottleneckReturn} \textit{bid}
Pipeline Stages: Code Modifications

BottleneckCall \( bid, \text{targetPC} \)

... targetPC:

while not done

while empty queue

BottleneckWait \( \text{prev\_bid} \)
dequeue work
do the work ...

while full queue

BottleneckWait \( \text{next\_bid} \)
enqueue next work

BottleneckReturn \( bid \)
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
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

Table:

<table>
<thead>
<tr>
<th>Small core</th>
<th>Small core</th>
<th>Large core</th>
</tr>
</thead>
<tbody>
<tr>
<td>Small core</td>
<td>Small core</td>
<td></td>
</tr>
<tr>
<td>Small core</td>
<td>Small core</td>
<td>Small core</td>
</tr>
<tr>
<td>Small core</td>
<td>Small core</td>
<td>Small core</td>
</tr>
<tr>
<td>Small core</td>
<td>Small core</td>
<td>Small core</td>
</tr>
</tbody>
</table>
Bottleneck Identification and Scheduling (BIS)

Compiler/Library/Programmer

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

Binary containing BIS instructions

Hardware

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

Small Core 1
BottleneckWait x4500

Large Core 0

Small Core 2
BottleneckWait x4500

Bottleneck Table (BT)

...
Bottleneck Identification and Scheduling (BIS)

1. Annotate bottleneck code
2. Implement 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
Bottleneck Acceleration

**BottleneckCall**

- **Small Core 1**
  - bid=x4700, pc, sp, core1
  - Execute remotely

- **BottleneckTable (BT)**
  - bid=x4700, large core 0

- **BottleneckIndex Table (AIT)**
  - bid=x4600, twc=100
  - bid=x4700, twc=10000

**Large Core 0**

- **BottleneckReturn x4700**
  - bid=x4700, pc, sp, core1

**Scheduling Buffer (SB)**

- twc < Threshold
- twc > Threshold

**Small Core 2**

- 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
Hardware Cost

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

- Off the critical path

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

- **Faster bottleneck execution** vs. **fewer parallel threads**
  - Acceleration offsets loss of parallel throughput with large core counts

- **Better shared data locality** vs. **worse private data locality**
  - Shared data stays on large core (good)
  - Private data migrates to large core (bad, but latency hidden with Data Marshaling [Suleman+, ISCA’ 10])

- **Benefit of acceleration** vs. **migration latency**
  - Migration latency usually hidden by waiting (good)
  - Unless bottleneck not contended (bad, but likely not on critical path)
Evaluation Methodology

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

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

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

- **SCMP (Symmetric CMP)**
  - All small cores

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

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

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

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

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

Limiting bottlenecks change over time; barriers, which ACS cannot accelerate
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)

**Fraction of execution time spent on predicted-important bottlenecks**

- **Actually critical**
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
If Time Permits …
A Case for
Asymmetry Everywhere

Onur Mutlu,
"Asymmetry Everywhere (with Automatic Resource Management)"
Asymmetry Enables Customization

- **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

| High-power | Power/performance optimized for each access pattern | Different technologies Power characteristics |
| High perf. | | |

Asymmetric / configurable cores and accelerators
Asymmetric / partitionable memory hierarchies
Asymmetric / partitionable interconnect
Asymmetric main memories
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

Phase 1
- High-power
  - High perf.

Phase 2
- Power/performance optimized for each access pattern

Phase 3
- Different technologies
  - Power characteristics

Asymmetric / configurable cores and accelerators
Asymmetric / partitionable memory hierarchies
Asymmetric / partitionable interconnect
Asymmetric main memories
Thought Experiment: Asymmetry Everywhere

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

![Diagram showing three versions of software components]

- **Version 1**
  - High-power
  - High perf.
- **Version 2**
  - Power/performance optimized for each access pattern
- **Version 3**
  - Different technologies
  - Power characteristics

- Asymmetric / configurable cores and accelerators
- Asymmetric / partitionable memory hierarchies
- Asymmetric / partitionable interconnect
- Asymmetric main memories
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

<table>
<thead>
<tr>
<th>High-power</th>
<th>Asymmetric / configurable cores and accelerators</th>
</tr>
</thead>
<tbody>
<tr>
<td>High perf.</td>
<td>Asymmetric / partitionable memory hierarchies</td>
</tr>
<tr>
<td><strong>Serial</strong></td>
<td>Asymmetric / partitionable interconnect</td>
</tr>
<tr>
<td></td>
<td>Asymmetric main memories</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Power/performance optimized for each access pattern</th>
<th>VLIW Backend</th>
</tr>
</thead>
<tbody>
<tr>
<td>High-power</td>
<td>High perf.</td>
</tr>
<tr>
<td>Backend</td>
<td></td>
</tr>
<tr>
<td>Different technologies</td>
<td></td>
</tr>
<tr>
<td>Power characteristics</td>
<td></td>
</tr>
</tbody>
</table>

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

Asymmetric / configurable cores and accelerators
Asymmetric / partitionable memory hierarchies
Asymmetric / partitionable interconnect
Asymmetric main memories
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

| High–power  |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |�

- Execute bandwidth-sensitive threads on a bandwidth-optimized network, latency-sensitive ones on a latency-optimized network [Das+ DAC’13]
- Higher performance and energy-efficiency than a single network
Exploiting Asymmetry: Simple Examples

- Partition memory controller and on-chip network bandwidth asymmetrically among threads
  
  Higher performance and energy-efficiency than symmetric/free-for-all

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

<table>
<thead>
<tr>
<th>High-power</th>
<th>High perf.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Power/performance optimized for each access pattern</td>
<td></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

<table>
<thead>
<tr>
<th>Compute intensive</th>
<th>Memory intensive</th>
</tr>
</thead>
<tbody>
<tr>
<td>Asymmetric / configurable cores and accelerators</td>
<td></td>
</tr>
<tr>
<td>Asymmetric / partitionable memory hierarchies</td>
<td></td>
</tr>
<tr>
<td>Asymmetric / partitionable interconnect</td>
<td></td>
</tr>
<tr>
<td>Asymmetric main memories</td>
<td></td>
</tr>
</tbody>
</table>
Exploiting Asymmetry: Simple Examples

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

<table>
<thead>
<tr>
<th>High-power</th>
<th>Power/performance optimized for each access pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>High perf.</td>
<td></td>
</tr>
</tbody>
</table>

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

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

| High–power | Power/performance optimized for each access pattern | Asymmetric / configurable cores and accelerators |
| High perf. | | |
| | | |
| | | Asymmetric / partitionable memory hierarchies |
| | | |
| Heterogeneous-Latency DRAM | | |
| Heterogeneous-Refresh-Rate DRAM | | |
| Different technologies | | |
| Power characteristics | | |
| | Asymmetric / partitionable interconnect |
| | Asymmetric main memories |