Yesterday and Today

- Heterogeneous Multi-Core Systems
- Bottleneck Acceleration
Some Readings


Recall: 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)
Recall: 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
Contestion for Critical Sections

12 iterations, 33% instructions inside the critical section

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

Critical Section
Parallel
Idle

33% in critical section
Contestion for Critical Sections

12 iterations, 33% instructions inside the critical section

Accelerating critical sections increases performance and scalability
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:**
- Y-axis: Speedup
- X-axis: Chip Area (cores)
- Two curves labeled "Today" and "Asymmetric" for MySQL (oltp-1)
A Case for Asymmetry

- Execution time of sequential kernels, critical sections, and limiter stages must be short

- It is difficult for the programmer to shorten these serialized sections
  - Insufficient domain-specific knowledge
  - Variation in hardware platforms
  - Limited resources
  - Performance-debugging tradeoff

- Goal: A mechanism to shorten serial bottlenecks without requiring programmer effort

- Idea: Accelerate serialized code sections by shipping them to powerful cores in an asymmetric multi-core (ACMP)
An Example: Accelerated Critical Sections

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

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

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

![Diagram showing ACS operation]
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-pong

- **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**
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

<table>
<thead>
<tr>
<th>SCMP</th>
<th>ACMP</th>
<th>ACS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conventional locking</td>
<td>Conventional locking</td>
<td>Large core executes Amdahl’s serial part and critical sections</td>
</tr>
<tr>
<td>Large core executes Amdahl’s serial part</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The tables and diagrams illustrate the comparison between different core execution strategies. Each cell in the tables represents a scenario where "Small core" or "Large core" is executed, highlighting the differences in locking mechanisms and execution patterns between SCMP, ACMP, and ACS.
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
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**
More on Accelerated Critical Sections


Accelerating Critical Section Execution with Asymmetric Multi-Core Architectures

M. Aater Suleman
University of Texas at Austin
suleman@hps.utexas.edu

Onur Mutlu
Carnegie Mellon University
onur@cmu.edu

Moinuddin K. Qureshi
IBM Research
mkquresh@us.ibm.com

Yale N. Patt
University of Texas at Austin
patt@ece.utexas.edu
Can We Accelerate All Types of Synchronization Bottlenecks?
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
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

Binary containing BIS instructions

**Compiler/Library/Programmer**

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

**Hardware**
Critical Sections: Code Modifications

```
while cannot acquire lock
  Wait loop for watch_addr
  acquire lock
  release lock
BottleneckCall bid, targetPC
BottleneckWait for watch_addr
  release lock
BottleneckReturn bid
```
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
...
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)

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
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

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

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. Implement *waiting* for bottlenecks

Binary containing BIS instructions

Compiler/Library/Programmer

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

Hardware

Bottleneck Identification and Scheduling (BIS)
Bottleneck Acceleration

Small Core 1

BottleneckCall $bid=x4700$, $pc$, $sp$, core1

Execute remotely

Acceleration Index Table (AIT)

bid=x4700, large core 0

Small Core 2

AIT

bid=x4700, large core 0

Large Core 0

BottleneckCall $bid=x4700$

BottleneckReturn $x4700$

Scheduling Buffer (SB)

bid=x4700, $pc$, $sp$, core1

 Execute remotely

Bottleneck Table (BT)

Bid=x4600, twc=100

Bid=x4700, twc=10000

$twc < \text{Threshold}$

$twc > \text{Threshold}$

$\leftarrow$
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?

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

Performance increases with:

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

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

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

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

- BIS improves application performance and scalability:
  - Performance benefits increase with more cores

- Provides comprehensive fine-grained bottleneck acceleration with no programmer effort
More on Bottleneck Identification & Scheduling

- Jose A. Joao, M. Aater Suleman, Onur Mutlu, and Yale N. Patt, "Bottleneck Identification and Scheduling in Multithreaded Applications" 
Improving on BIS?

Can We Make Better Acceleration Decisions?
Utility-Based Acceleration of Multithreaded Applications

Jose A. Joao, M. Aater Suleman, Onur Mutlu, and Yale N. Patt,
"Utility-Based Acceleration of Multithreaded Applications on Asymmetric CMPs"
Proceedings of the 40th International Symposium on Computer Architecture (ISCA), Tel-Aviv, Israel, June 2013. Slides (ppt) Slides (pdf)
Bottlenecks

Accelerating Critical Sections (ACS), Suleman et al., ASPLOS’ 09

Bottleneck Identification and Scheduling (BIS), Joao et al., ASPLOS’ 12
Lagging Threads

T2: Lagging thread

Barrier 1  t1  Barrier 2

Lagging thread = potential future bottleneck
Execution time reduction
Two Problems

1) Do we accelerate bottlenecks or lagging threads?

2) Multiple applications: which application do we accelerate?

Acceleration decisions need to consider both:
- the criticality of code segments
- how much speedup they get for bottlenecks and lagging threads from any running application
Utility-Based Acceleration (UBA)

- **Goal:** identify performance-limiting bottlenecks or lagging threads from any running application and accelerate them on large cores of an ACMP.

- **Key insight:** A New Utility of Acceleration metric that combines speedup and criticality of each code segment.

- Utility of accelerating code segment $c$ of length $t$ on an application of length $T$:

  $$U_c = \frac{\Delta T}{T} = \left(\frac{\Delta t}{t}\right) \times \left(\frac{t}{T}\right) \times \left(\frac{\Delta T}{\Delta t}\right)$$

  - **Global Criticality of Segment**
  - **Local Speedup of Segment**
  - **Fraction of Exec Time Spent on Segment**
Utility-Based Acceleration (UBA)

- Bottleneck Identification
- Lagging Thread Identification
  - Set of Highest-Utility Bottlenecks
  - Set of Highest-Utility Lagging Threads
- Acceleration Coordination
- Large core control
UBA Results

2-application workloads, 60 small cores, 1 large core

UBA outperforms BIS and another alternative approach by \(~8\%\).
More on Utility-Based Acceleration

- Jose A. Joao, M. Aater Suleman, Onur Mutlu, and Yale N. Patt, "Utility-Based Acceleration of Multithreaded Applications on Asymmetric CMPs"
  Proceeding of the 40th International Symposium on Computer Architecture (ISCA), Tel-Aviv, Israel, June 2013. Slides (ppt) Slides (pdf)

Utility-Based Acceleration of Multithreaded Applications on Asymmetric CMPs

José A. Joao †  M. Aater Suleman ††  Onur Mutlu §  Yale N. Patt †

† ECE Department  †† Flux7 Consulting  § Computer Architecture Laboratory
The University of Texas at Austin  Austin, TX, USA  Carnegie Mellon University
{joao, patt}@ece.utexas.edu  suleman@hps.utexas.edu  Pittsburgh, PA, USA
59 onur@cmu.edu
Can We Do Better?
Handling Private Data Locality: Data Marshaling

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)

LOAD X
STORE Y
STORE Y

LOAD Y
....
STORE Z

LOAD Z
....
Split code into segments

**Segment S0**
- LOAD X
- STORE Y
- STORE Y

**Segment S1**
- LOAD Y
- ....
- STORE Z

**Segment S2**
- LOAD Z
- ....
Staged Execution Model (IV)

Core 0

Instances of S0

Core 1

Instances of S1

Core 2

Instances of S2
Staged Execution Model: Segment Spawning

Core 0

LOAD X
STORE Y
STORE Y

Core 1

LOAD Y
....
STORE Z

Core 2

LOAD Z
....

S0

S1

S2
Staged Execution Model: Two Examples

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

- **Producer-Consumer Pipeline Parallelism**
  - Idea: Split a loop iteration into multiple “pipeline stages” where one stage consumes data produced by the previous stage → each stage runs on a different core
    - Segment N: Stage N
  - Benefit: Stage-level parallelism, better locality → faster execution
Problem: Locality of Inter-segment Data

LOAD X
STORE Y
STORE Y

Transfer Y

Core 0

Core 1

Core 2

LOAD Y
...
STORE Z
...
LOAD Z
...

Transfer Z

Cache Miss

Cache Miss

Cache Miss

S0

S1

S2
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

Data Marshaling

**Compiler/Profiler**
1. Identify *generator* instructions
2. Insert *marshal* instructions

**Hardware**
1. Record *generator*-produced addresses
2. Marshal recorded blocks to next core

Binary containing *generator prefixes* & *marshal Instructions*
Data Marshaling

**Compiler/Profiler**

1. Identify *generator* instructions
2. Insert *marshal* instructions

Binary containing *generator prefixes & marshal Instructions*

**Hardware**

1. Record *generator*-produced addresses
2. *Marshal* recorded blocks to next core
Profiling Algorithm

Inter-segment data

Mark as Generator Instruction

LOAD X
STORE Y
STORE Y

LOAD Y
....
STORE Z

LOAD Z
....
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

Small Core 0

Large Core

L2 Cache

LOAD X
STORE Y
G: STORE Y
CSCALL

LOAD Y
....
G:STORE Z
CSRET

Critical Section

Cache Hit!

L2 Cache

Addr Y

Data Y

Marshal Buffer
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

Speedup over ACS

- puzzle: 168, ideal: 170
- is: 100, ideal: 100
- maze: 110, ideal: 110
- nqueen: 120, ideal: 120
- sqlite: 130, ideal: 130
- iplookup: 140, ideal: 140
- mysql-1: 150, ideal: 150
- mysql-2: 160, ideal: 160
- webcache: 170, ideal: 170
- hmean: 8.7%
Pipeline Parallelism

Cache Hit!

Core 0

Addr Y

L2 Data Y

Core 1

L2 Cache

LOAD X

STORE Y

MARSHAL C1

G: STORE Y

MARSHAL C2

0x5: LOAD Z

L2 Cache

L2

Addr Y

Data Y

Marshal Buffer
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

The chart shows the speedup over the baseline for various applications, such as black, compress, dedupD, dedupE, ferret, image, mtwist, rank, sign, and hmean. The x-axis represents the applications, and the y-axis represents the speedup over the baseline. The red bars indicate the DM speedup, and the blue bars indicate the ideal speedup. The chart highlights an average speedup of 16%.
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 $\rightarrow$ More communication
  - Higher latency $\rightarrow$ Longer stalls due to communication
  - Larger L2 cache $\rightarrow$ Communication misses remain
Other Applications of Data Marshaling

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

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

- **Inter-segment data transfers between cores** limit the benefit of promising Staged Execution (SE) models

- Data Marshaling is a hardware/software cooperative solution:
  - detect inter-segment data generator instructions and push their data to next segment’s core
    - Significantly reduces cache misses for inter-segment data
    - Low cost, high-coverage, timely for arbitrary address sequences
    - Achieves most of the potential of eliminating such misses

- Applicable to several existing Staged Execution models
  - Accelerated Critical Sections: 9% performance benefit
  - Pipeline Parallelism: 16% performance benefit

- Can enable new models→ very fine-grained remote execution
More on Bottleneck Identification & Scheduling

- M. Aater Suleman, Onur Mutlu, Jose A. Joao, Khubaib, and Yale N. Patt,
  "Data Marshaling for Multi-core Architectures"

Data Marshaling for Multi-core Architectures

M. Aater Suleman†  Onur Mutlu§  José A. Joao†  Khubaib†  Yale N. Patt†

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

§Carnegie Mellon University
onur@cmu.edu
Other Uses of Asymmetry
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 $\rightarrow$ wrong core $\rightarrow$ reduced performance or increased energy
  - Overhead of core switching
  - Disadvantages of asymmetric CMP (e.g., design multiple cores)
  - Need phase monitoring and matching algorithms
    - What characteristics should be monitored?
    - Once characteristics known, how do you pick the core?
Asymmetric vs. Symmetric Cores

- **Advantages of Asymmetric**
  + Can provide better performance when thread parallelism is limited
  + Can be more energy efficient
    + Schedule computation to the core type that can best execute it

- **Disadvantages**
  - Need to design more than one type of core. Always?
  - Scheduling becomes more complicated
    - What computation should be scheduled on the large core?
    - Who should decide? HW vs. SW?
  - Managing locality and load balancing can become difficult if threads move between cores (transparently to software)
  - Cores have different demands from shared resources
How to Achieve Asymmetry

- **Static**
  - Type and power of cores fixed at design time
  - Two approaches to design “faster cores”:
    - High frequency
    - Build a more complex, powerful core with entirely different uarch
  - Is static asymmetry natural? (chip-wide variations in frequency)

- **Dynamic**
  - Type and power of cores change dynamically
  - Two approaches to dynamically create “faster cores”:
    - Boost frequency dynamically (limited power budget)
    - Combine small cores to enable a more complex, powerful core
    - Is there a third, fourth, fifth approach?
Asymmetry via Frequency Boosting
Asymmetry via Boosting of Frequency

- **Static**
  - Due to process variations, cores might have different frequency
  - Simply hardwire/design cores to have different frequencies

- **Dynamic**
  - Dynamic voltage and frequency scaling
EPI Throttling

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

- For best scalar and throughput performance, vary energy expended per instruction (EPI) based on available parallelism
  - \[ P = EPI \cdot IPS \]
  - \( P = \) 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
EPI Throttling via DVFS

- **DVFS**: Dynamic voltage frequency scaling

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

- In phases of high thread parallelism
  - Run many cores at low supply voltage and low frequency
Possible EPI Throttling Techniques


<table>
<thead>
<tr>
<th>Method</th>
<th>EPI Range</th>
<th>Time to Alter EPI</th>
<th>Throttle Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Voltage/frequency scaling</td>
<td>1:2 to 1:4</td>
<td>100us (ramp Vcc)</td>
<td>Lower voltage and frequency</td>
</tr>
<tr>
<td>Asymmetric cores</td>
<td>1:4 to 1:6</td>
<td>10us (migrate 256KB L2 cache)</td>
<td>Migrate threads from large cores to small cores</td>
</tr>
<tr>
<td>Variable-size core</td>
<td>1:1 to 1:2</td>
<td>1us (fill 32KB L1 cache)</td>
<td>Reduce capacity of processor resources</td>
</tr>
<tr>
<td>Speculation control</td>
<td>1:1 to 1:1.4</td>
<td>10ns (pipeline latency)</td>
<td>Reduce amount of speculation</td>
</tr>
</tbody>
</table>
Boosting Frequency of a Small Core vs. Large Core

- Frequency boosting implemented on Intel Nehalem, IBM POWER7

- Advantages of Boosting Frequency
  + Very simple to implement; no need to design a new core
  + Parallel throughput does not degrade when TLP is high
  + Preserves locality of boosted thread

- Disadvantages
  - Does not improve performance if thread is memory bound
  - Does not reduce Cycles per Instruction (remember the performance equation?)
  - Changing frequency/voltage can take longer than switching to a large core
A Case for Asymmetry Everywhere

Onur Mutlu,
"Asymmetry Everywhere (with Automatic Resource Management)"
Position paper
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

<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

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

<table>
<thead>
<tr>
<th>Version 1</th>
<th>Version 2</th>
<th>Version 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>High-power</td>
<td>Asymmetric / configurable cores and accelerators</td>
<td></td>
</tr>
<tr>
<td>High perf.</td>
<td>Asymmetric / partitionable memory hierarchies</td>
<td></td>
</tr>
<tr>
<td>Power/performance optimized for each access pattern</td>
<td>Asymmetric / partitionable interconnect</td>
<td></td>
</tr>
<tr>
<td>Different technologies</td>
<td>Power characteristics</td>
<td>Asymmetric main memories</td>
</tr>
</tbody>
</table>
Many Research and Design Questions

- How to design asymmetric components?
  - Fixed, partitionable, reconfigurable components?
  - What types of asymmetry? Access patterns, technologies?

- What monitoring to perform cooperatively in HW/SW?
  - Automatically discover phase/task requirements

- How to design feedback/control loop between components and runtime system software?

- How to design the runtime to automatically manage resources?
  - Track task behavior, pick “best-fit” components for the entire workload
Exploiting Asymmetry: Simple Examples

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

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

<table>
<thead>
<tr>
<th>Backend</th>
<th>VLIW Backend</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>

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

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

- Execute streaming “memory phases” on streaming-optimized cores and memory hierarchies
  - More efficient and higher performance than general purpose hierarchy
## Exploiting Asymmetry: Simple Examples

<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><strong>Latency optimized NoC</strong></td>
<td>Asymmetric / partitionable interconnect</td>
</tr>
<tr>
<td>Different technologies Power characteristics</td>
<td>Asymmetric main memories</td>
</tr>
</tbody>
</table>

- 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

<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></td>
<td>Asymmetric / partitionable interconnect</td>
</tr>
<tr>
<td></td>
<td>Asymmetric main memories</td>
</tr>
</tbody>
</table>

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

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

- Compute intensive
- Memory intensive

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 (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

| High–power | Power/performance optimized for each access pattern |
| High perf. | DRAM | CPU |
| Asymmetric / configurable cores and accelerators |
| Asymmetric / partitionable memory hierarchies |

- DRAM
  - Fast, durable
  - Small, leaky, volatile, high-cost

- Phase Change Memory (or Tech. X)
  - Large, non-volatile, low-cost
  - Slow, wears out, high active energy

- PCM Ctrl
- DRA MCtrl

- DRA
- PCM Ctrl
Exploiting Asymmetry: Simple Examples

| High–power | Asymmetric / configurable cores and accelerators |
| High perf. | |
| Power/performance optimized for each access pattern | Asymmetric / partitionable memory hierarchies |

- **Reliable DRAM**
- **Less Reliable DRAM**

- 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

| High-power | Asymmetric / configurable cores and accelerators |
| High perf. | |
| Power/performance optimized for each access pattern | Asymmetric / partitionable memory hierarchies |

**Heterogeneous-Latency DRAM**

**Heterogeneous-Refresh-Rate DRAM**

**Different technologies**

**Power characteristics**

- 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