Review Set 9

- Due today (October 19)

- Recommended:
  - Hennessy and Patterson, Appendix C.2 and C.3
Readings for Today

- Accelerated Critical Sections and Data Marshaling

    - Shorter version of ASPLOS 2009 paper. Read the ASPLOS 2009 paper for details.

    - Shorter version of ISCA 2010 paper. Read the ISCA 2010 paper for details.
Announcements

- Midterm I next Monday
  - October 24

- Exam Review
  - Likely this Friday during class time (October 21)

- Extra Office Hours
  - During the weekend – check with the TAs

- Milestone II
  - Is postponed. Stay tuned.
Last Lecture

- Dual-core execution
- Memory disambiguation
Today

- Research issues in out-of-order execution or latency tolerance
- Accelerated critical sections
Open Research Issues in OOO Execution (I)

- Performance with simplicity and energy-efficiency
- How to build scalable and energy-efficient instruction windows
  - To tolerate very long memory latencies and to expose more memory level parallelism
  - Problems:
    - How to scale or avoid scaling register files, store buffers
    - How to supply useful instructions into a large window in the presence of branches
- How to approximate the benefits of a large window
  - MLP benefits vs. ILP benefits
  - Can the compiler pack more misses (MLP) into a smaller window?
- How to approximate the benefits of OOO with in-order + enhancements
Open Research Issues in OOO Execution (II)

- OOO in the presence of multi-core

  - More problems: Memory system contention becomes a lot more significant with multi-core
    - OOO execution can overcome extra latencies due to contention
    - How to preserve the benefits (e.g. MLP) of OOO in a multi-core system?

  - More opportunity: Can we utilize multiple cores to perform more scalable OOO execution?
    - Improve single-thread performance using multiple cores

- Asymmetric multi-cores (ACMP): What should different cores look like in a multi-core system?
  - OOO essential to execute serial code portions
### Open Research Issues in OOO Execution (III)

- **Out-of-order execution in the presence of multi-core**
- **Powerful execution engines are needed to execute**
  - Single-threaded applications
  - Serial sections of multithreaded applications (remember Amdahl’ s law)
  - Where single thread performance matters (e.g., transactions, game logic)
  - Accelerate multithreaded applications (e.g., critical sections)

<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-Large” Approach**
- **“Niagara” Approach**
- **ACMP Approach**

#### Niagara-like Core
- Niagara-like core
- Niagara-like core
- Niagara-like core
- Niagara-like core

#### Large core
- Niagara-like core
- Niagara-like core
- Niagara-like core
- Niagara-like core

#### “Tile-Large” Approach
- Niagara-like core
- Niagara-like core
- Niagara-like core
- Niagara-like core

#### “Niagara” Approach
- Niagara-like core
- Niagara-like core
- Niagara-like core
- Niagara-like core

#### ACMP Approach
- Niagara-like core
- Niagara-like core
- Niagara-like core
- Niagara-like 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
A Case for Asymmetry

- Execution time of sequential kernels, critical sections, and limiter stages must be short
- It is difficult for programmer to shorten these serial bottlenecks
  - Insufficient domain-specific knowledge
  - Variation in hardware platforms
  - Limited resources
- Goal: a mechanism to shorten serial bottlenecks without requiring programmer effort
- Solution: Ship serial code sections to a large, powerful core in an asymmetric multi-core processor
“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)
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-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 Chip Multiprocessor (ACMP)

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

Single thread $\rightarrow$ Large core

ACMP Approach
Performance vs. Parallelism

Assumptions:

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

2. Large core takes an area budget of 4 and has performance of 2
### ACMP Performance vs. Parallelism

Area-budget = 16 small cores

<table>
<thead>
<tr>
<th></th>
<th>“Tile-Large”</th>
<th>“Tile-Small”</th>
<th>ACMP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Large Cores</td>
<td>4</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Small Cores</td>
<td>0</td>
<td>16</td>
<td>12</td>
</tr>
<tr>
<td>Serial Performance</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Parallel Throughput</td>
<td>(2 \times 4 = 8)</td>
<td>(1 \times 16 = 16)</td>
<td>(1 \times 2 + 1 \times 12 = 14)</td>
</tr>
</tbody>
</table>
An Example: Accelerated Critical Sections

- **Problem:** Synchronization and parallelization is difficult for programmers → Critical sections are a performance bottleneck
- **Idea:** HW/SW ships critical sections to a large, powerful core in Asymmetric MC

- **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

Contention for Critical Sections

**Accelerating critical sections not only helps the thread executing the critical sections, but also the waiting threads.**

Critical Sections execute 2x faster
Impact of Critical Sections on Scalability

- Contention for critical sections increases with the number of threads and limits scalability

```plaintext
LOCK_open -> Acquire()
foreach (table locked by thread)
  table.lock -> release()
  table.file -> release()
if (table.temporary)
  table.close()
LOCK_open -> Release()
```
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

Onchip-Interconnect

Core executing critical section

Critical Section Request Buffer (CSRB)
Accelerated Critical Sections (ACS)

**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**

- **Speedup over SCMP**
  - pagetine
  - puzzle
  - qsort
  - sqlite
  - tsp
  - iplookup
  - oltp-1
  - oltp-2
  - specjbb
  - webcache
  - hmean

- Coarse-grain locks
- Fine-grain locks
ACS Performance Tradeoffs

- **Fewer 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 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
ACS Comparison Points

<table>
<thead>
<tr>
<th>SCMP</th>
<th>ACMP</th>
<th>ACS</th>
</tr>
</thead>
<tbody>
<tr>
<td>All small cores</td>
<td>One large core (area-equal 4 small cores)</td>
<td>ACMP with a CSRB</td>
</tr>
<tr>
<td>Conventional locking</td>
<td>Conventional locking</td>
<td>Accelerates Critical Sections</td>
</tr>
</tbody>
</table>

**ACS**

- ACMP with a CSRB
- Accelerates Critical Sections
Equal-Area Comparisons

Number of threads = No. of cores

Chip Area (small cores)

Speedup over a small core

(a) ep  (b) is  (c) pagemine  (d) puzzle  (e) qsort  (f) tsp

(g) sqlite  (h) iplookup  (i) oltp-1  (i) oltp-2  (k) specjbb  (l) webcache

--- SCMP  --- ACMP  ------ ACS
How Can We Do Better?

- Transfer of private data to the large core limits performance of ACS
- Can we identify/predict which data will need to be transferred to the large core and ship it there while shipping the critical section?

Data Marshaling Summary

- **Staged execution (SE):** Break a program into segments; run each segment on the “best suited” core
  - new performance improvement and power savings opportunities
  - accelerators, pipeline parallelism, task parallelism, customized cores, ...

- **Problem:** SE performance limited by inter-segment data transfers
  - A segment incurs a cache miss for data it needs from a previous segment

- **Data marshaling:** detect inter-segment data and send it to the next segment’s core before needed
  - Profiler: Identify and mark “generator” instructions; insert “marshal” hints
  - Hardware: Buffer “generated” data and “marshal” it to next segment

- Achieves almost all benefit of ideally eliminating inter-segment cache misses on two SE models, with low hardware overhead
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 [Suleman et al., ASPLOS 2010]
  - 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
....
Staged Execution Model (III)

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)
Staged Execution Model: Segment Spawning

Core 0
S0
LOAD X
STORE Y
STORE Y

Core 1
S1
LOAD Y
....
STORE Z

Core 2
S2
LOAD Z
....
Staged Execution Model: Two Examples

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

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

Core 0

S0

LOAD X
STORE Y
STORE Y

Core 1

S0

LOAD Y
...
STORE Z

Transfer Y

Cache Miss

Core 2

S2

LOAD Z
...

Transfer Z

Cache Miss
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

1. Identify generator instructions
2. Insert marshal instructions

Binary containing generator prefixes & marshal Instructions

1. Record generator-produced addresses
2. Marshal recorded blocks to next core
Data Marshaling for ACS

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!

Marshal Buffer

Addr Y

Data Y
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
DM on Accelerated Critical Sections: Results

The graph shows the speedup over ACS for various applications, with the DM and Ideal scenarios compared. The applications include 'is', 'pagemine', 'puzzle', 'qsort', 'tsp', 'maze', 'nqueen', 'sqlite', 'iplookup', 'mysql-1', 'mysql-2', 'webcache', and 'hmean'.

The highest speedup is observed for 'puzzle', with a speedup of 168 for DM and 170 for Ideal, resulting in an 8.7% improvement.
Pipeline Parallelism

Cache Hit!

Core 0
Addr Y

L2
Data Y

Core 1

L2 Cache

LOAD X
STORE Y
MARSHAL C1

LOAD Y
...
MARSHAL C2

G: STORE Y

G: STORE Z

0x5: LOAD Z

Cache Hit!
DM on Pipeline Parallelism: Results

The bar chart shows the speedup over the baseline for various applications using DM and the ideal scenario. The chart indicates that the speedup ranges from approximately 110% to 150% for different applications. The peak speedup is observed for the 'hmean' application, which shows a 16% improvement over the baseline when using DM compared to the ideal scenario.
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