Where We Are in Lecture Schedule

- The memory hierarchy
- Caches, caches, more caches
- Virtualizing the memory hierarchy: Virtual Memory
- Main memory: DRAM
- Main memory control, scheduling
- Memory latency tolerance techniques
- Non-volatile memory

- Multiprocessors
- Coherence and consistency
- In-memory computation and predictable performance
- Multi-core issues (e.g., heterogeneous multi-core)
- Interconnection networks
First, Some Administrative Things
Midterm II and Midterm II Review

- **Midterm II is this Friday** (April 24, 2015)
  - 12:30-2:30pm, CIC Panther Hollow Room (4th floor)
  - Please arrive 5 minutes early and sit with 1-seat separation
  - Same rules as Midterm I except you get to have 2 cheat sheets
  - Covers all topics we have examined so far, with more focus on Lectures 17-32 (Memory Hierarchy and Multiprocessors)

- **Midterm II Review is Wednesday** (April 22)
  - Come prepared with questions on concepts and lectures
  - Detailed homework and exam questions and solutions → study on your own and ask TAs during office hours
Suggestions for Midterm II

- Solve past midterms (and finals) on your own...
  - And, check your solutions vs. the online solutions
  - Questions will be similar in spirit


- Do Homework 7 and go over past homeworks.
- Study and internalize the lecture material well.
  - Buzzwords can help you. Ditto for slides and videos.
- Understand how to solve all homework & exam questions.
- Study hard.
- Also read: [https://piazza.com/class/i3540xiz8ku40a?cid=335](https://piazza.com/class/i3540xiz8ku40a?cid=335)
Lab 8: Multi-Core Cache Coherence

- Due May 3; Last submission accepted on May 10, 11:59pm
- Cycle-level modeling of the MESI cache coherence protocol

**Invalid**
- Other cache has write-miss (invalidate)

**Shared**
- Cache miss (1 requester)
- Write (upgrade and inval. others)
- Other cache has read-miss (downgrade)

**Modified**
- Other cache has write-miss (invalidate)
- Cache miss (1 requester)

**Exclusive**
- Write (mark dirty)
- Other cache has read-miss (downgrade)
Reminder of 447 policy:

- Absolutely no form of collaboration allowed
- No discussions, no code sharing, no code reviews with fellow students, no brainstorming, ...

All labs and all portions of each lab has to be your own work

- Just focus on doing the lab yourself, alone
We Have Another Course for Collaboration

- 740 is the next course in sequence
- Tentative Time: Lect. MW 7:30-9:20pm, (Rect. T 7:30pm)
- Content:
  - Lectures: More advanced, with a different perspective
  - Recitations: Delving deeper into papers, advanced topics
  - Readings: Many fundamental and research readings; will do many reviews
  - **Project**: More open ended research project. Proposal → milestones → final poster and presentation
    - Done in groups of 1-3
    - Focus of the course is the project (and papers)
  - Exams: lighter and fewer
  - Homeworks: None
A Note on Testing Your Own Code

- We provide the reference simulator to aid you
- Do not expect it to be given, and do not rely on it much

- In real life, there are no reference simulators

- The architect designs the reference simulator
- The architect verifies it
- The architect tests it
- The architect fixes it
- The architect makes sure there are no bugs
- The architect ensures the simulator matches the specification
Lab 6 Grade Distribution

- Average: 97.3
- Median: 100
- Std Dev: 9.40
Lab 6 Extra Credit Recognitions

- Stay tuned...
Lab 4-5 Special Recognition

- Limited out-of-order execution
  - Terence An
Where We Are in Lecture Schedule

- The memory hierarchy
- Caches, caches, more caches
- Virtualizing the memory hierarchy: Virtual Memory
- Main memory: DRAM
- Main memory control, scheduling
- Memory latency tolerance techniques
- Non-volatile memory

- Multiprocessors
- Coherence and consistency
- In-memory computation and predictable performance
- Multi-core issues (e.g., heterogeneous multi-core)
- Interconnection networks
Today

- Heterogeneity (asymmetry) in system design
- Evolution of multi-core systems
- Handling serial and parallel bottlenecks better
- Heterogeneous multi-core systems
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.)
We Have Already Seen Examples Before (in 447)

- 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 rate
- 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

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

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

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

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

CPU

MCtrl

DRAM Ctrl

PCM Ctrl

Hardware/software manage data allocation and movement to achieve the best of multiple technologies
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
  - Higher priority

- **Does not starve**
  - Not prioritized
  - Reduced throughput

_starvation ➔ unfairness*

---

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
Remember: Heterogeneous Retention Times in DRAM

64-128ms

>256ms

128-256ms
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
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
Multi-Core Design
Many Cores on Chip

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

**AMD Barcelona**
- 4 cores

**Intel Core i7**
- 8 cores

**IBM Cell BE**
- 8+1 cores

**IBM POWER7**
- 8 cores

**Sun Niagara II**
- 8 cores

**Nvidia Fermi**
- 448 “cores”

**Intel SCC**
- 48 cores, networked

**Tilera TILE Gx**
- 100 cores, networked
With Many Cores on Chip

- **What we want:**
  - N times the performance with N times the cores when we parallelize an application on N cores

- **What we get:**
  - Amdahl’s Law (serial bottleneck)
  - Bottlenecks in the parallel portion
Caveats of Parallelism

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

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


- **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)
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

Critical Section
- Open database tables
- Perform the operations

Access Open Tables Cache

Parallel Access

Chip Area (cores)

Speedup

Asymmetric

Today
Demands in Different Code Sections

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

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

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

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

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


<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 $\rightarrow$ one powerful “large” core
  - In a parallel code section $\rightarrow$ 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-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)

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)

<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”

<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>
</tbody>
</table>

“Tile-Small”

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

ACMP
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>Large Cores</th>
<th>Small Cores</th>
<th>Serial Performance</th>
<th>Parallel Throughput</th>
</tr>
</thead>
<tbody>
<tr>
<td>Large core</td>
<td>Large core</td>
<td>4</td>
<td>2 x 4 = 8</td>
</tr>
<tr>
<td>Large core</td>
<td>Large core</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Small Cores</td>
<td></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Small Cores</td>
<td></td>
<td>16</td>
<td>1 x 16 = 16</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>“Tile-Large”</th>
<th>“Tile-Small”</th>
<th>ACMP</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>16</td>
<td>12</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

1 x 2 + 1 x 12 = 14
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

\[
\text{Speedup} = \frac{1}{\frac{1 - f}{X} + \frac{f}{S + X \cdot L}}
\]
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 = 1
P = 2
P = 3
P = 4

33% in critical section
Contention for Critical Sections

12 iterations, 33% instructions inside the critical section

Accelerating critical sections increases performance and scalability

Critical Section
Parallel
Idle

Critical Section Accelerated by 2x
Impact of Critical Sections on Scalability

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

![Graph showing impact on scalability](image)

**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

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

P1

P2
P3
P4

Critical Section Request Buffer (CSRB)

Onchip-Interconnect

Core executing critical section
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

Speedup over SCMP

Accelerating Sequential Kernels
Accelerating Critical Sections

Coarse-grain locks
Fine-grain locks

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

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

Number of threads = No. of cores

Speedup over a small core

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