Multicore and Parallelism and Synchronization I

CS 3410, Spring 2014
Computer Science
Cornell University

See P&H Chapter: 4.10, 1.7, 1.8, 5.10, 6.4, 2.11
Next few weeks

• Week 12 (Apr 22): Lab4 release and Proj3 due Fri
  – Note Lab 4 is now IN CLASS
• Week 13 (Apr 29): Proj4 release, **Lab4 due Tue**, Prelim2
• Week 14 (May 6): Proj3 tournament Mon, Proj4 design doc due

Final Project for class

• Week 15 (May 13): Proj4 due Wed
Dynamic Multiple Issue

Diagram:
- Instruction fetch and decode unit
- Reservation station
- Integer
  - Functional units
- Reservation station
- Integer
- Out-of-order execute
- Reservation station
- Floating point
- In-order issue
- Reservation station
- Load-store
- In-order commit
- Commit unit
Limits of Static Scheduling

Compiler scheduling for dual-issue MIPS...

```assembly
lw  $t0, 0($s1)  # load A
addi $t0, $t0, +1 # increment A
sw  $t0, 0($s1)  # store A
lw  $t1, 0($s2)  # load B
addi $t1, $t1, +1 # increment B
sw  $t1, 0($s2)  # store B
```

ALU/branch slot

- nop
- nop
- addi $t0, $t0, +1
- nop
- nop
- addi $t1, $t1, +1
- nop

Load/store slot

<table>
<thead>
<tr>
<th>cycle</th>
<th>ALU/branch slot</th>
<th>Load/store slot</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>nop</td>
<td>lw  $t0, 0($s1)</td>
</tr>
<tr>
<td>2</td>
<td>nop</td>
<td>nop</td>
</tr>
<tr>
<td>3</td>
<td>nop</td>
<td>nop</td>
</tr>
<tr>
<td>4</td>
<td>nop</td>
<td>sw  $t0, 0($s1)</td>
</tr>
<tr>
<td>5</td>
<td>addi $t1, $t1, +1</td>
<td>lw  $t1, 0($s2)</td>
</tr>
<tr>
<td>6</td>
<td>nop</td>
<td>nop</td>
</tr>
<tr>
<td>7</td>
<td>nop</td>
<td>nop</td>
</tr>
<tr>
<td>8</td>
<td>nop</td>
<td>sw  $t1, 0($s2)</td>
</tr>
</tbody>
</table>
Does Multiple Issue Work?

Q: Does multiple issue / ILP work?
A: Kind of... but not as much as we’d like

Limiting factors?

• Programs dependencies
• Hard to detect dependencies → be conservative
  – e.g. Pointer Aliasing: A[0] += 1; B[0] *= 2;
• Hard to expose parallelism
  – Can only issue a few instructions ahead of PC
• Structural limits
  – Memory delays and limited bandwidth
• Hard to keep pipelines full
Today

Many ways to improve performance
Multicore
  Performance in multicore
  Synchronization

Next 2 lectures: synchronization and GPUs
How to improve performance?

We have looked at

- Pipelining

- To speed up:
  - Deeper pipelining
  - Make the clock run faster
  - Parallelism
    - Not a luxury, a necessity
Moore’s law

• A law about transistors
• Smaller means more transistors per die
• And smaller means faster too

But: need to worry about power too...
Power Wall

Power = \text{capacitance} \times \text{voltage}^2 \times \text{frequency}

\approx \text{capacitance} \times \text{voltage}^3

Reducing voltage helps (a lot)
Better cooling helps

The power wall

- We can’t reduce voltage further - leakage
- We can’t remove more heat
Power Limits

- Surface of Sun
- Rocket Nozzle
- Nuclear Reactor
- Hot Plate

Watts/cm² vs. Process Technology:

- 32nm
- 180nm

Xeon
Why Multicore?

- Single-Core Overclocked +20%
  - Performance: 1.2x
  - Power: 1.7x

- Single-Core
  - Performance: 1.0x
  - Power: 1.0x

- Dual-Core Underclocked -20%
  - Performance: 1.6x
  - Power: 1.02x

\[ (1.2)^3 \times 2 \]
Inside the Processor

AMD Barcelona Quad-Core: 4 processor cores
Inside the Processor

Intel Nehalem Hex-Core
Hardware multithreading

Hardware multithreading
  • Increase utilization with low overhead

Switch between hardware threads for stalls
What is a thread?

Process includes multiple threads, code, data and OS state

![Diagram showing single-threaded and multithreaded processes](image)
Hardware multithreading

Fine grained vs. Coarse grained hardware multithreading

Simultaneous multithreading (SMT)

Hyperthreads (Intel simultaneous multithreading)
  • Need to hide latency
Fine grained vs. Coarse grained hardware multithreading

Fine grained multithreading
  Switch on each cycle
  Pros: Can hide very short stalls
  Cons: Slows down every thread

Coarse grained multithreading
  Switch only on quite long stalls
  Pros: removes need for very fast switches
  Cons: flush pipeline, short stalls not handled
Simultaneous multithreading

SMT

- Leverages multi-issue pipeline with dynamic instruction scheduling and ILP
- Exploits functional unit parallelism better than single threads
- **Always running multiple instructions from multiple threads**
  - No cost of context switching
  - Uses dynamic scheduling and register renaming through reservation stations
- Can use all functional units very efficiently
Hyperthreading

Multi-Core vs. Multi-Issue vs. HT

<table>
<thead>
<tr>
<th>Programs:</th>
<th>Num. Pipelines:</th>
<th>Pipeline Width:</th>
</tr>
</thead>
<tbody>
<tr>
<td>$N$</td>
<td>$N$</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>$N$</td>
</tr>
</tbody>
</table>

Hyperthreads

- HT = MultiIssue + extra PCs and registers – dependency logic
- HT = MultiCore – redundant functional units + hazard avoidance

Hyperthreads (Intel)

- Illusion of multiple cores on a single core
- Easy to keep HT pipelines full + share functional units
Example: All of the above

- 8 multiprocessors
- 4 core per multiprocessor
- 2 HT per core
- Dynamic multi-issue: 4 issue
- Pipeline depth: 16

Note: each processor may have multiple processing cores, so this is an example of a multiprocessor multicore hyperthreaded system
Parallel Programming

Q: So lets just all use multicore from now on!
A: Software must be written as parallel program

Multicore difficulties

• Partitioning work, balancing load
• Coordination & synchronization
• Communication overhead
• How do you write parallel programs?
  – ... without knowing exact underlying architecture?
Work Partitioning

Partition work so all cores have something to do

T/8
Load Balancing

Need to partition so all cores are actually working
Amdahl’s Law

If tasks have a serial part and a parallel part...

Example:

step 1: divide input data into $n$ pieces
step 2: do work on each piece
step 3: combine all results

Recall: Amdahl’s Law

As number of cores increases ...

• time to execute parallel part? goes to zero
• time to execute serial part? Remains the same
• Serial part eventually dominates
Pitfall: Amdahl’s Law

Execution time after improvement =

\[
\frac{\text{affected execution time}}{\text{amount of improvement}} + \text{execution time unaffected}
\]

\[
T_{\text{improved}} = \frac{T_{\text{affected}}}{\text{improvement factor}} + T_{\text{unaffected}}
\]
Pitfall: Amdahl’s Law

Improving an aspect of a computer and expecting a proportional improvement in overall performance

\[ T_{\text{improved}} = \frac{T_{\text{affected}}}{\text{improvement factor}} + T_{\text{unaffected}} \]

Example: multiply accounts for 80s out of 100s
• How much improvement do we need in the multiply performance to get 5× overall improvement?

\[ 20 = \frac{80}{n} + 20 \quad \text{– Can’t be done!} \]
**Scaling Example**

Workload: sum of 10 scalars, and $10 \times 10$ matrix sum

- Speed up from 10 to 100 processors?

Single processor: \( \text{Time} = (10 + 100) \times t_{\text{add}} \)

10 processors

- \( \text{Time} = \frac{100}{10} \times t_{\text{add}} + 10 \times t_{\text{add}} = 20 \times t_{\text{add}} \)
- \( \text{Speedup} = \frac{110}{20} = 5.5 \)

100 processors

- \( \text{Time} = \frac{100}{100} \times t_{\text{add}} + 10 \times t_{\text{add}} = 11 \times t_{\text{add}} \)
- \( \text{Speedup} = \frac{110}{11} = 10 \)

Assumes load can be balanced across processors
Scaling Example

What if matrix size is $100 \times 100$?

Single processor: Time $= (10 + 10000) \times t_{\text{add}}$

10 processors
- Time $= 10 \times t_{\text{add}} + 10000/10 \times t_{\text{add}} = 1010 \times t_{\text{add}}$
- Speedup $= 10010/1010 = 9.9$

100 processors
- Time $= 10 \times t_{\text{add}} + 10000/100 \times t_{\text{add}} = 110 \times t_{\text{add}}$
- Speedup $= 10010/110 = 91$

Assuming load balanced
Scaling

Strong scaling vs. weak scaling

Strong scaling: scales with same problem size

Weak scaling: scales with increased problem size
Parallelism is a necessity

Necessity, not luxury
Power wall

Not easy to get performance out of

Many solutions
Pipelining
Multi-issue
Hyperthreading
Multicore
Parallel Programming

Q: So let's just all use multicore from now on!
A: Software must be written as parallel program

Multicore difficulties

- Partitioning work
- Coordination & synchronization
- Communications overhead
- Balancing load over cores
- How do you write parallel programs?
  - ... without knowing exact underlying architecture?
Synchronization

P&H Chapter 2.11 and 5.10
Parallelism and Synchronization

How do I take advantage of *parallelism*?

How do I write *(correct)* parallel programs?

What primitives do I need to implement correct parallel programs?
Topics

Understand Cache Coherency

Synchronizing parallel programs
• Atomic Instructions
• HW support for synchronization

How to write parallel programs
• Threads and processes
• Critical sections, race conditions, and mutexes
Parallelism and Synchronization

Cache Coherency Problem: What happens when two or more processors cache *shared* data?
Parallelism and Synchronization

Cache Coherency Problem: What happens when two or more processors cache *shared* data?

i.e. the view of memory held by two different processors is through their individual caches

As a result, processors can see different *(incoherent)* values to the *same* memory location
Parallelism and Synchronization

Each processor core has its own L1 cache
Parallelism and Synchronization

Each processor core has its own L1 cache
Shared Memory Multiprocessors

Shared Memory Multiprocessor (SMP)

- Typical (today): 2 – 8 cores each
- HW provides *single physical address* space for all processors
- Assume uniform memory access (UMA) (ignore NUMA)
Shared Memory Multiprocessors

Shared Memory Multiprocessor (SMP)

- Typical (today): 2 – 8 cores each
- HW provides *single physical address* space for all processors
- Assume uniform memory access (ignore NUMA)
Thread A (on Core0)
for(int i = 0, i < 5; i++) {
    x = x + 1;
}

Thread B (on Core1)
for(int j = 0; j < 5; j++) {
    x = x + 1;
}

What will the value of x be after both loops finish?

Start: x = 0
Thread A (on Core0)
for(int i = 0, i < 5; i++) {
    x = x + 1;
}

Thread B (on Core1)
for(int j = 0; j < 5; j++) {
    x = x + 1;
}
Cache Coherency Problem

Thread A (on Core0)
for(int i = 0, i < 5; i++) {
    LW $t0, addr(x)
    ADDIU $t0, $t0, 1
    SW $t0, addr(x)
}

Thread B (on Core1)
for(int j = 0; j < 5; j++) {
    LW $t1, addr(x)
    ADDIU $t1, $t1, 1
    SW $t1, addr(x)
}
Thread A (on Core0)
for(int i = 0; i < 5; i++) {
    \textcolor{blue}{x = x + 1;}
}

Thread B (on Core1)
for(int j = 0; j < 5; j++) {
    \textcolor{blue}{x = x + 1;}
}

What can the value of \( x \) be after both loops finish?

a) 6
b) 8
c) 10
d) All of the above
e) None of the above
# Cache Coherence Problem

Suppose two CPU cores share a physical address space

- Write-through caches

<table>
<thead>
<tr>
<th>Time</th>
<th>Event</th>
<th>CPU A’s cache</th>
<th>CPU B’s cache</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>CPU A reads X</td>
<td>0</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>CPU B reads X</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>CPU A writes 1 to X</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

---

**Diagram:**
- Core0
  - Cache
- Core1
  - Cache
- CoreN
  - Cache
- Interconnect
- Memory
- I/O
Two issues

Coherence
  What values can be returned by a read

Consistency
  When a written value will be returned by a read
Coherence Defined

Informal: **Reads** return most recently **written** value

Formal: For concurrent processes $P_1$ and $P_2$

- $P$ writes $X$ before $P$ reads $X$ (with no intervening writes)  
  $\Rightarrow$ read returns written value

- $P_1$ writes $X$ before $P_2$ reads $X$  
  $\Rightarrow$ read returns written value

- $P_1$ writes $X$ and $P_2$ writes $X$  
  $\Rightarrow$ all processors see writes in the same order
  - all see the same final value for $X$
  - Aka write serialization
Coherence Defined

Formal: For concurrent processes $P_1$ and $P_2$

- $P$ writes $X$ before $P$ reads $X$ (with no intervening writes)
  $\Rightarrow$ read returns written value
  - (preserve program order)

- $P_1$ writes $X$ before $P_2$ reads $X$
  $\Rightarrow$ read returns written value
  - (coherent memory view, can’t read old value forever)

- $P_1$ writes $X$ and $P_2$ writes $X$
  $\Rightarrow$ all processors see writes in the same order
  - all see the same final value for $X$
  - Aka write serialization
  - (else $X$ can see $P_2$’s write before $P_1$ and $Y$ can see the opposite; their final understanding of state is wrong)
Cache Coherence Protocols

Operations performed by caches in multiprocessors to ensure coherence and support shared memory

- **Migration** of data to local caches
  - Reduces bandwidth for shared memory (performance)
- **Replication** of read-shared data
  - Reduces contention for access (performance)

Snooping protocols

- Each cache monitors bus reads/writes (correctness)
Snooping for Hardware Cache Coherence

- All caches monitor bus and all other caches

Write invalidate protocol

- Bus read: respond if you have dirty data
- Bus write: update/invalidate your copy of data
Invalidating Snooping Protocols

Cache gets **exclusive access** to a block when it is to be written

- Broadcasts an invalidate message on the bus
- Subsequent read is another cache miss
  - Owning cache supplies updated value

<table>
<thead>
<tr>
<th>Time Step</th>
<th>CPU activity</th>
<th>Bus activity</th>
<th>CPU A's cache</th>
<th>CPU B's cache</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>CPU A reads X</td>
<td>Cache miss for X</td>
<td>0</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>CPU B reads X</td>
<td>Cache miss for X</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>CPU A writes 1 to X</td>
<td>Invalidate for X</td>
<td>1</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>CPU B read X</td>
<td>Cache miss for X</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>
# Invalidating Snooping Protocols

Cache gets **exclusive access** to a block when it is to be written

- Broadcasts an invalidate message on the bus
- Subsequent read is another cache miss
  - Owning cache supplies updated value

<table>
<thead>
<tr>
<th>Time Step</th>
<th>CPU activity</th>
<th>Bus activity</th>
<th>CPU A’s cache</th>
<th>CPU B’s cache</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>CPU A reads X</td>
<td>Cache miss for X</td>
<td>0</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>CPU B reads X</td>
<td>Cache miss for X</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>CPU A writes 1 to X</td>
<td>Invalidate for X</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>CPU B read X</td>
<td>Cache miss for X</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Writing

Write-back policies for bandwidth
Write-invalidate coherence policy
  - First invalidate all other copies of data
  - Then write it in cache line
  - Anybody else can read it

Works with one writer, multiple readers

In reality: many coherence protocols
  - Snooping doesn’t scale
  - Directory-based protocols
    - Caches and memory record sharing status of blocks in a directory
Summary of cache coherence

Informally, Cache Coherency requires that reads return most recently written value

Cache coherence hard problem

Snooping protocols are one approach