Multicore and Parallelism

CS 3410, Spring 2014

Computer Science

Cornell University

See P&H Chapter: 4.10, 1.7, 1.8, 5.10, 6
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
Instruction-Level Parallelism (ILP)

Pipelining: execute multiple instructions in parallel

Q: How to get more instruction level parallelism?
A: Deeper pipeline
   - E.g. 250MHz 1-stage; 500Mhz 2-stage; 1GHz 4-stage; 4GHz 16-stage

Pipeline depth limited by...
   - max clock speed
   - min unit of work (less work per stage ⇒ shorter clock cycle)
   - dependencies, hazards / forwarding logic
Pipelining: execute multiple instructions in parallel
Q: How to get more instruction level parallelism?
A: Multiple issue pipeline
   – Start multiple instructions per clock cycle in duplicate stages
Multiple issue pipeline

Static multiple issue
aka Very Long Instruction Word
Decisions made by compiler

Dynamic multiple issue
Decisions made on the fly

Cost: More execute hardware
Reading/writing register files: more ports
Static Multiple Issue

a.k.a. Very Long Instruction Word (VLIW)

Compiler groups instructions to be issued together
  • Packages them into “issue slots”

Q: How does HW detect and resolve hazards?
A: It doesn’t
  ➔ Simple HW, assumes compiler avoids hazards

Example: Static Dual-Issue 32-bit MIPS
  • Instructions come in pairs (64-bit aligned)
    – One ALU/branch instruction (or nop)
    – One load/store instruction (or nop)
MIPS with Static Dual Issue

Two-issue packets

• One ALU/branch instruction
• One load/store instruction
• 64-bit aligned
  – ALU/branch, then load/store
  – Pad an unused instruction with nop
• Delay slot: 2 instructions (1 cycle)

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction type</th>
<th>Pipeline Stages</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>ALU/branch</td>
<td>IF ID EX MEM WB</td>
</tr>
<tr>
<td>n + 4</td>
<td>Load/store</td>
<td>IF ID EX MEM WB</td>
</tr>
<tr>
<td>n + 8</td>
<td>ALU/branch</td>
<td>IF ID EX MEM WB</td>
</tr>
<tr>
<td>n + 12</td>
<td>Load/store</td>
<td>IF ID EX MEM WB</td>
</tr>
<tr>
<td>n + 16</td>
<td>ALU/branch</td>
<td>IF ID EX MEM WB</td>
</tr>
<tr>
<td>n + 20</td>
<td>Load/store</td>
<td>IF ID EX MEM WB</td>
</tr>
</tbody>
</table>
Scheduling Example

Schedule this for dual-issue MIPS

Loop:

lw $t0, 0($s1)      # $t0=array element
addu $t0, $t0, $s2    # add scalar in $s2
sw $t0, 0($s1)      # store result
addi $s1, $s1,–4      # decrement pointer
bne $s1, $zero, Loop # branch $s1!=0

<table>
<thead>
<tr>
<th>ALU/branch</th>
<th>Load/store</th>
<th>cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loop:</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
</tr>
</tbody>
</table>
Speculation

Reorder instructions
To fill the issue slot with useful work
Complicated: exceptions may occur
Optimizations to make it work

Move instructions to fill in nops
Need to track hazards and dependencies

Loop unrolling
Scheduling Example

Schedule this for dual-issue MIPS

Loop:  
- `lw   $t0, 0($s1)`  # $t0=array element
- `addu $t0, $t0, $s2`  # add scalar in $s2
- `sw   $t0, 0($s1)`  # store result
- `addi $s1, $s1,–4`  # decrement pointer
- `bne  $s1, $zero, Loop`  # branch $s1!=0
Scheduling Example

Compiler scheduling for dual-issue MIPS...

Loop:  
\begin{align*}
&\text{lw } \$t0, 0($s1) \quad \# \$t0 = A[i] \\
&\text{lw } \$t1, 4($s1) \quad \# \$t1 = A[i+1] \\
&\text{addu } \$t0, \$t0, \$s2 \quad \# \text{add } \$s2 \\
&\text{addu } \$t1, \$t1, \$s2 \quad \# \text{add } \$s2 \\
&\text{sw } \$t0, 0($s1) \quad \# \text{store } A[i] \\
&\text{sw } \$t1, 4($s1) \quad \# \text{store } A[i+1] \\
&\text{addi } \$s1, \$s1, +8 \quad \# \text{increment pointer} \\
&\text{bne } \$s1, \$s3, \text{Loop} \quad \# \text{continue if } \$s1!=\text{end} \\
\end{align*}

ALU/branch slot \hspace{2cm} \text{Load/store slot} \hspace{2cm} \text{cycle}

8 cycles \hspace{2cm} 6 cycles
Scheduling Example

Compiler scheduling for dual-issue MIPS...

Loop:
```
  lw  $t0, 0($s1)  # $t0 = A[i]
  lw  $t1, 4($s1)  # $t1 = A[i+1]
  addu $t0, $t0, $s2  # add $s2
  addu $t1, $t1, $s2  # add $s2
  sw  $t0, 0($s1)  # store A[i]
  sw  $t1, 4($s1)  # store A[i+1]
  addi $s1, $s1, +8  # increment pointer
  bne  $s1, $s3, Loop  # continue if $s1!=end
```

ALU/branch slot  Load/store slot  cycle

8 cycles  5 cycles
Limits of Static Scheduling

Compiler scheduling for dual-issue MIPS...

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

<table>
<thead>
<tr>
<th>ALU/branch slot</th>
<th>Load/store slot</th>
<th>cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>nop</td>
<td>lw  $t0, 0($s1)</td>
<td>1</td>
</tr>
<tr>
<td>nop</td>
<td>nop</td>
<td>2</td>
</tr>
<tr>
<td>addi $t0, $t0, +1</td>
<td>nop</td>
<td>3</td>
</tr>
<tr>
<td>nop</td>
<td>sw  $t0, 0($s1)</td>
<td>4</td>
</tr>
<tr>
<td>nop</td>
<td>lw  $t0, 0($s2)</td>
<td>5</td>
</tr>
<tr>
<td>nop</td>
<td>nop</td>
<td>6</td>
</tr>
<tr>
<td>addi $t0, $t0, +1</td>
<td>nop</td>
<td>7</td>
</tr>
<tr>
<td>nop</td>
<td>sw  $t0, 0($s2)</td>
<td>8</td>
</tr>
</tbody>
</table>
Limits of Static Scheduling

Compiler scheduling for dual-issue MIPS...

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw $t0, 0($s1)</td>
<td># load A</td>
<td>1</td>
</tr>
<tr>
<td>addi $t0, $t0, +1</td>
<td># increment A</td>
<td>2</td>
</tr>
<tr>
<td>sw $t0, 0($s1)</td>
<td># store A</td>
<td>3</td>
</tr>
<tr>
<td>lw $t1, 0($s2)</td>
<td># load B</td>
<td>4</td>
</tr>
<tr>
<td>addi $t1, $t1, +1</td>
<td># increment B</td>
<td>5</td>
</tr>
<tr>
<td>sw $t1, 0($s2)</td>
<td># store B</td>
<td>6</td>
</tr>
</tbody>
</table>

ALU/branch slot
- nop
- nop
- addi $t0, $t0, +1
- nop
- nop
- addi $t1, $t1, +1
- nop

Load/store slot
- lw $t0, 0($s1) | 1 |
- nop | 2 |
- nop | 3 |
- sw $t0, 0($s1) | 4 |
- lw $t1, 0($s2) | 5 |
- nop | 6 |
- nop | 7 |
- sw $t1, 0($s2) | 8 |
Limits of Static Scheduling

Compiler scheduling for dual-issue MIPS...

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
Load/store slot:

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

Problem: What if $s1 and $s2 are equal (aliasing)? Won’t work
Dynamic Multiple Issue

a.k.a. SuperScalar Processor

CPU examines instruction stream and chooses multiple instructions to issue each cycle

- Compiler can help by reordering instructions....
- ... but CPU is responsible for resolving hazards
Dynamic Multiple Issue

a.k.a. SuperScalar Processor

Speculation/Out-of-order Execution

- Execute instructions as early as possible
- Aggressive register renaming
- Guess results of branches, loads, etc.
- Roll back if guesses were wrong
- Don’t commit results until all previous insts. are retired
Dynamic Multiple Issue

Instruction fetch and decode unit

Reservation station
Reservation station
... Reservation station
Reservation station

Functional units
Integer
Integer
... Floating point
Load-store

Commit unit

In-order issue

Out-of-order execute

In-order commit
Why dynamic scheduling?

To handle unpredictable stalls
Like cache misses

Hides details of pipeline from applications
Abstraction
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
Dual-core Itanium 2

Itanium 2

Pentium

P4

K8

K10

Atom

Curve shows ‘Moore’s Law’: transistor count doubling every two years.
### Power Efficiency

**Q:** Does multiple issue / ILP cost much?

**A:** Yes. Dynamic issue & speculation requires power.

![Power Efficiency Table](chart)

→ Multiple simpler cores may be better?
Why Multicore?

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 = capacitance * voltage² * frequency
approx. capacitance * voltage³

Reducing voltage helps (a lot)
Better cooling helps

The power wall
- We can’t reduce voltage further - leakage
- We can’t remove more heat
Why Multicore?

Performance

Power

Single-Core

Overclocked +20%

1.2x

1.7x

1.0x

1.0x

Single-Core

1.6x

1.6x

1.02x

Dual-Core

Underclocked -20%
Inside the Processor

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

Intel Nehalem Hex-Core
Hyperthreading

Hyperthreads (Intel)
  • Illusion of multiple cores on a single core

Switch between hardware threads for stalls
  Fine grained and coarse grained
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>$1$</td>
<td>$N$</td>
</tr>
<tr>
<td>$N$</td>
<td>$1$</td>
<td>$1$</td>
</tr>
<tr>
<td>$1$</td>
<td>$N$</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
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**
Amdahl’s Law
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?

\[ 20 = \frac{80}{n} + 20 \]

– 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: Time

10 processors
  - Time =
  - Speedup =

100 processors
  - Time =
  - Speedup =

Assumes load can be balanced across processors
Scaling Example

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

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

10 processors
- Time =
- Speedup =

100 processors
- Time =
- Speedup =

Assuming load balanced
Scaling

Strong scaling vs. weak scaling

Strong scaling: scales with same problem size

Weak scaling: scales with increased problem size
Parallel Programming

Q: So lets 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 multiple processors; *parallelism*?

How do I write *(correct)* parallel programs, *cache coherency and synchronization*?

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 (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

<table>
<thead>
<tr>
<th>Thread A (on Core0)</th>
<th>Thread B (on Core1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>for(int i = 0, i &lt; 5; i++) {</td>
<td>for(int j = 0; j &lt; 5; j++) {</td>
</tr>
<tr>
<td>LW $t0, addr(x)</td>
<td>LW $t0, addr(x)</td>
</tr>
<tr>
<td>ADDIU $t0, $t0, 1</td>
<td>ADDIU $t0, $t0, 1</td>
</tr>
<tr>
<td>SW $t0, addr(x)</td>
<td>SW $t0, addr(x)</td>
</tr>
<tr>
<td>}</td>
<td>}</td>
</tr>
</tbody>
</table>
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?

a) 6
b) 8
c) 10
d) All of the above
e) None of the above