Multicore and Parallel Processing

Hakim Weatherspoon
CS 3410, Spring 2013
Computer Science
Cornell University

P & H Chapter 4.10-11, 7.1-6
It took a lot of work, but this latest Linux patch enables support for machines with 4,096 CPUs, up from the old limit of 1,024.

Do you have support for smooth full-screen Flash video yet?

No, but who uses that?
Pitfall: Amdahl’s Law

Execution time after improvement =

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

+ 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
Scaling Example

Workload: sum of 10 scalars, and 10 × 10 matrix sum
  • Speed up from 10 to 100 processors?

Single processor: Time = (10 + 100) × t_{add}

10 processors

100 processors
Scaling Example

What if matrix size is 100 \times 100?

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

10 processors

100 processors
Goals for Today

How to improve System Performance?

• Instruction Level Parallelism (ILP)
• Multicore
  – Increase clock frequency vs multicore
• Beware of Amdahls Law

Next time:

• Concurrency, programming, and synchronization
Problem Statement

Q: How to improve system performance?
→ Increase CPU clock rate?
   → But I/O speeds are limited
      Disk, Memory, Networks, etc.

Recall: Amdahl’s Law

Solution: Parallelism
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 (less work per stage ⇒ shorter clock cycle)
− min unit of work
− dependencies, hazards / forwarding logic
Instruction-Level Parallelism (ILP)

Pipelining: execute multiple instructions in parallel

Q: How to get more instruction level parallelism?
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?
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

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

Compiler scheduling for dual-issue MIPS...

### Loop:

<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 $t1, 4($s1)</td>
<td>2</td>
</tr>
<tr>
<td>addu $t0, $t0, $s2</td>
<td>nop</td>
<td>3</td>
</tr>
<tr>
<td>addu $t1, $t1, $s2</td>
<td>sw $t0, 0($s1)</td>
<td>4</td>
</tr>
<tr>
<td>addi $s1, $s1, +8</td>
<td>sw $t1, 4($s1)</td>
<td>5</td>
</tr>
<tr>
<td>bne $s1, $s3, TOP</td>
<td>nop</td>
<td>6</td>
</tr>
</tbody>
</table>
## Scheduling Example

### Compiler scheduling for dual-issue MIPS...

**Loop:**

<table>
<thead>
<tr>
<th>ALU/branch slot</th>
<th>Load/store slot</th>
<th>cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loop: nop</td>
<td>lw $t0, 0($s1)</td>
<td>1</td>
</tr>
<tr>
<td>addi $s1, $s1, +8</td>
<td>lw $t1, 4($s1)</td>
<td>2</td>
</tr>
<tr>
<td>addu $t0, $t0, $s2</td>
<td>nop</td>
<td>3</td>
</tr>
<tr>
<td>addu $t1, $t1, $s2</td>
<td>sw $t0, -8($s1)</td>
<td>4</td>
</tr>
<tr>
<td>bne $s1, $s3, Loop</td>
<td>sw $t1, -4($s1)</td>
<td>5</td>
</tr>
</tbody>
</table>
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...

\[
\begin{align*}
\text{lw} & \quad \text{\$t0, 0(\$s1)} & \# \text{ load A} \\
\text{addi} & \quad \text{\$t0, \$t0, +1} & \# \text{ increment A} \\
\text{sw} & \quad \text{\$t0, 0(\$s1)} & \# \text{ store A} \\
\text{lw} & \quad \text{\$t1, 0(\$s2)} & \# \text{ load B} \\
\text{addi} & \quad \text{\$t1, \$t1, +1} & \# \text{ increment B} \\
\text{sw} & \quad \text{\$t0, 0(\$s2)} & \# \text{ store B}
\end{align*}
\]

<table>
<thead>
<tr>
<th>ALU/branch slot</th>
<th>Load/store slot</th>
<th>cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{nop}</td>
<td>\text{lw} $\text{$t0, 0($s1)}$</td>
<td>1</td>
</tr>
<tr>
<td>\text{nop}</td>
<td>\text{lw} $\text{$t1, 0($s2)}$</td>
<td>2</td>
</tr>
<tr>
<td>\text{addi} \text{$t0, $t0, +1}</td>
<td>\text{nop}</td>
<td>3</td>
</tr>
<tr>
<td>\text{addi} \text{$t1, $t1, +1}</td>
<td>\text{sw} $\text{$t0, 0($s1)}$</td>
<td>4</td>
</tr>
<tr>
<td>\text{nop}</td>
<td>\text{sw} $\text{$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 (c.f. Intel)

• 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

Even better: 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
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
Power Efficiency

Q: Does multiple issue / ILP cost much?
Moore's Law

Pentium

386

8088

8086

286

486

Dual-core Itanium 2

Itanium 2

P4

K8

K10

Atom

Dual-core Itanium 2

Itanium 2

P4

K8

K10

Atom

Curve shows ‘Moore’s Law’: transistor count doubling every two years

Transistor count

2,000,000,000

1,000,000,000

100,000,000

10,000,000

1,000,000

2,300

Why Multicore?

Moore’s law

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

But: Power consumption growing too...
Power Wall

Power = capacitance * voltage\(^2\) * frequency

In practice: Power \(\sim\) voltage\(^3\)

Reducing voltage helps (a lot)
... so does reducing clock speed
Better cooling helps

The power wall

- We can’t reduce voltage further
- We can’t remove more heat
Why Multicore?

Performance
Power

Single-Core
Overclocked +20%

1.2x
1.7x

Single-Core

1.0x
1.0x

Dual-Core
Underclocked -20%

1.6x
1.02x
Inside the Processor

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

Intel Nehalem Hex-Core
**Hyperthreading**

<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>
## Hyperthreading

<table>
<thead>
<tr>
<th></th>
<th>Multi-Core vs. Multi-Issue</th>
<th>vs. HT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Programs:</td>
<td>(N)</td>
<td>(N)</td>
</tr>
<tr>
<td>Num. Pipelines:</td>
<td>(N)</td>
<td>(1)</td>
</tr>
<tr>
<td>Pipeline Width:</td>
<td>(1)</td>
<td>(N)</td>
</tr>
</tbody>
</table>

### Hyperthreads

- \(HT = \text{MultiIssue} + \) extra PCs and registers – dependency logic
- \(HT = \text{MultiCore} – \) 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
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?
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
Parallel Programming

Q: So let's just all use multicore from now on!

Multicore difficulties

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

Lab3 is *due today*, Thursday, April 11\(^{th}\)

Project3 available now, due Monday, April 22\(^{nd}\)
  - Design Doc *due next week*, Monday, April 15\(^{th}\)
  - Schedule a Design Doc review Mtg now, by *tomorrow* Friday, April 12\(^{th}\)
  - *See me after class if looking for new partner*
  - **Competition/Games night Friday, April 26\(^{th}\), 5-7pm. Location: B17 Upson**

Homework4 is available now, *due next week*, Wednesday, April 17\(^{th}\)
  - Work alone
  - Question1 on Virtual Memory is pre-lab question for in-class Lab4
  - HW Help Session *Thurs (Apr 11) and Mon (Apr 15)*, 6-7:30pm in B17 Upson

Prelim3 is in *two weeks*, Thursday, April 25\(^{th}\)
  - Time and Location: 7:30pm in Phillips 101 and Upson B17
  - Old prelims are online in CMS
Administrivia

Next four weeks

• Week 11 (Apr 8): Lab3 due and Project3/HW4 handout
• Week 12 (Apr 15): Project3 design doc due and HW4 due
• Week 13 (Apr 22): Project3 due and Prelim3
• Week 14 (Apr 29): Project4 handout

Final Project for class

• Week 15 (May 6): Project4 design doc due
• Week 16 (May 13): Project4 due