Multicore and Parallel Processing

Hakim Weatherspoon
CS 3410, Spring 2012
Computer Science
Cornell University

P & H Chapter 4.10-11, 7.1-6
Administrivia

FlameWar Games Night Next Friday, April 27\textsuperscript{th}

• 5pm in Upson B17
• Please come, eat, drink and have fun

No Lab4 or Lab Section \textit{next} week!
Administrivia

PA3: FlameWar is due next Monday, April 23\textsuperscript{rd}

• The goal is to have fun with it
• Recitations today will talk about it

HW6 Due next Tuesday, April 24\textsuperscript{th}

Prelim3 next Thursday, April 26\textsuperscript{th}
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 =

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

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

100 processors

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

100 processors

Assuming load balanced
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 \(\Rightarrow\) 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?

A: Multiple issue pipeline

– Start multiple instructions per clock cycle in duplicate stages
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

<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><strong>Loop:</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>nop</td>
<td>lw $t0, 0($s1)</td>
<td>1</td>
</tr>
<tr>
<td>addi $s1, $s1,–4</td>
<td>nop</td>
<td>2</td>
</tr>
<tr>
<td>addu $t0, $t0, $s2</td>
<td>nop</td>
<td>3</td>
</tr>
<tr>
<td>bne $s1, $zero, Loop</td>
<td>sw $t0, 4($s1)</td>
<td>4</td>
</tr>
</tbody>
</table>

- IPC = 5/4 = 1.25 (c.f. peak IPC = 2)
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, TOP  # continue if $s1!=end

<table>
<thead>
<tr>
<th>ALU/branch slot</th>
<th>Load/store slot</th>
<th>cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loop:</td>
<td></td>
<td></td>
</tr>
<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: 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, TOP  # continue if $s1!=end

<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...

```asm
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>
Dynamic Multiple Issue

da.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
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?

A: Yes.

→ Dynamic issue and speculation requires power

<table>
<thead>
<tr>
<th>CPU</th>
<th>Year</th>
<th>Clock Rate</th>
<th>Pipeline Stages</th>
<th>Issue width</th>
<th>Out-of-order/Speculation</th>
<th>Cores</th>
<th>Power</th>
</tr>
</thead>
<tbody>
<tr>
<td>i486</td>
<td>1989</td>
<td>25MHz</td>
<td>5</td>
<td>1</td>
<td>No</td>
<td>1</td>
<td>5W</td>
</tr>
<tr>
<td>Pentium</td>
<td>1993</td>
<td>66MHz</td>
<td>5</td>
<td>2</td>
<td>No</td>
<td>1</td>
<td>10W</td>
</tr>
<tr>
<td>Pentium Pro</td>
<td>1997</td>
<td>200MHz</td>
<td>10</td>
<td>3</td>
<td>Yes</td>
<td>1</td>
<td>29W</td>
</tr>
<tr>
<td>P4 Willamette</td>
<td>2001</td>
<td>2000MHz</td>
<td>22</td>
<td>3</td>
<td>Yes</td>
<td>1</td>
<td>75W</td>
</tr>
<tr>
<td>UltraSparc III</td>
<td>2003</td>
<td>1950MHz</td>
<td>14</td>
<td>4</td>
<td>No</td>
<td>1</td>
<td>90W</td>
</tr>
<tr>
<td>P4 Prescott</td>
<td>2004</td>
<td>3600MHz</td>
<td>31</td>
<td>3</td>
<td>Yes</td>
<td>1</td>
<td>103W</td>
</tr>
<tr>
<td>Core</td>
<td>2006</td>
<td>2930MHz</td>
<td>14</td>
<td>4</td>
<td>Yes</td>
<td>2</td>
<td>75W</td>
</tr>
<tr>
<td>UltraSparc T1</td>
<td>2005</td>
<td>1200MHz</td>
<td>6</td>
<td>1</td>
<td>No</td>
<td>8</td>
<td>70W</td>
</tr>
</tbody>
</table>

→ Multiple simpler cores may be better?
Curve shows 'Moore’s Law': transistor count doubling every two years.
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 Limits

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

Watts/cm²

10000
1000
100
10
1
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?

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

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

- **Dual-Core**
  - Underclocked -20%
  - Power: 1.02x
  - Performance: 1.6x
Inside the Processor

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

Intel Nehalem Hex-Core
Hyperthreading
Multi-Core vs. Multi-Issue vs. HT

Programs:
Num. Pipelines:
Pipeline Width:

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

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:

1. divide input data into $n$ pieces
2. do work on each piece
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 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?