Multicore and Parallelism

Prof. Hakim Weatherspoon
CS 3410, Spring 2015
Computer Science
Cornell University

P & H Chapter 4.10, 1.7, 1.8, 5.10, 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?
Big Picture: Multicore and Parallelism
Big Picture: Multicore and Parallelism
Why do I need *four* computing cores on my phone?!
Big Picture: Multicore and Parallelism

Why do I need *eight* computing cores on my phone?!
Big Picture: Multicore and Parallelism

Why do I need sixteen computing cores on my phone?!
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
• Multiply can be parallelized
• How much improvement do we need in the multiply performance to get 5× overall improvement?
  (a) 2x   (b) 10x  (c) 100x  (d) 1000x  (e) not possible
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

• Multiply can be parallized
• 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: Time = $(10 + 100) \times t_{add}$

10 processors
  • Time = $100/10 \times t_{add} + 10 \times t_{add} = 20 \times t_{add}$
  • Speedup = $110/20 = 5.5$

100 processors
  • Time = $100/100 \times t_{add} + 10 \times t_{add} = 11 \times t_{add}$
  • Speedup = $110/11 = 10$

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
Takeaway

Unfortunately, we cannot obtain unlimited scaling (speedup) by adding unlimited parallel resources, eventual performance is dominated by a component needing to be executed sequentially. Amdahl's Law is a caution about this diminishing return.
Announcements

• HW2 Review Sessions!
  • Saturday, April 18\textsuperscript{th}, Hollister B14@7pm
  • Tuesday, April 21\textsuperscript{st}, Hollister B14@7pm
• Lab3 is due yesterday!
  • Wednesday, April 15\textsuperscript{th}
  • \textit{In theory, theory and practice are the same. In practice, they are not.} –Albert Einstein
• PA3 started this week!
  • The Lord of the Cache!
  • Due Friday, April 24th
Announcements

• HW2-P5 (Pre-Lab4) due next week!
  • Monday, April 20th
  • Don’t forget to submit on CMS!
  • Designed for you to look over the code and understand virtual memory before coming to Lab 4.
Announcements

• Prelim 2 is on April 30\textsuperscript{th} at 7 PM at Statler Hall!

• If you have a conflict e-mail me:

\texttt{deniz@cs.cornell.edu}
Announcements

Next three weeks

• Week 12 (Apr 21): Lab4 due in-class, Proj3 due Fri, HW2 due Sat
• Week 13 (Apr 28): Proj4 release, Prelim2
• Week 14 (May 5): Proj3 tournament Mon, Proj4 design doc due

Final Project for class

• Week 15 (May 12): Proj4 due Wed
Goals for Today

How to improve System Performance?

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

Next 2 lectures:

• Cache coherency, synchronization, Concurrency, and programming
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
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?

A: Multiple issue pipeline
   - Start multiple instructions per clock cycle in duplicate stages

![Diagram showing ALU/Br and LW/SW stages](image_url)
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

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>Loop:</td>
<td>nop</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>lw $t0, 0($s1)</td>
<td></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>

\[
\text{IPC} = \frac{5 \text{ instructions}}{4 \text{ cycles}} = 1.25
\]

\[
\text{CPI} = \frac{4 \text{ cycles}}{5 \text{ instructions}} = 0.8
\]
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
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 $t0, -8($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
Loop: nop
nop
addu $t0, $t0, $s2
addu $t1, $t1, $s2
addi $s1, $s1, +8
bne $s1, $s3, Loop

Load/store slot
cycle

1 lw $t0, 0($s1) 1
2 lw $t1, 4($s1) 2
3 nop 3
4 sw $t0, 0($s1) 4
5 sw $t1, 4($s1) 5
6 nop 6

6 cycles
8 instructions

\[
\text{CPI} = \frac{6 \text{ cycles}}{8 \text{ instructions}} = 0.75
\]
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 $t0, -8($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

Loop:

```
nop
addi $s1, $s1, +8
addu $t0, $t0, $s2
addu $t1, $t1, $s2
bne $s1, $s3, Loop
```

Load/store slot

```
lw $t0, 0($s1) 1
lw $t1, 4($s1) 2
nop 3
sw $t0, -8($s1) 4
sw $t1, -4($s1) 5
```

8 cycles

8 instructions

\[
\text{CPI} = \frac{5 \text{ cycles}}{8 \text{ instructions}} = 0.625
\]
Limits of Static Scheduling
Compiler scheduling for dual-issue MIPS...

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

<table>
<thead>
<tr>
<th>ALU/branch slot</th>
<th>Load/store slot</th>
<th>cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>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...

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
nop
addi $t1, $t1, +1
nop
Load/store slot     cycle
lw $t0, 0($s1) 1
nop 2
sw $t0, 0($s1) 4
lw $t1, 0($s2) 5
nop 6
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
```

<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></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 (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

Diagram:
- Instruction fetch and decode unit
- Reservation station
- Integer
- Reservation station
- Integer
- ... (ellipses)
- Reservation station
- Floating point
- Reservation station
- Load-store
- Commit unit

Flow:
- In-order issue
- Out-of-order execute
- In-order commit
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
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>Core i5 Nehal</td>
<td>2010</td>
<td>3300MHz</td>
<td>14</td>
<td>4</td>
<td>Yes</td>
<td>1</td>
<td>87W</td>
</tr>
<tr>
<td>Core i5 Ivy Br</td>
<td>2012</td>
<td>3400MHz</td>
<td>14</td>
<td>4</td>
<td>Yes</td>
<td>8</td>
<td>77W</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?
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

1.5 µ 1 µ 0.7 µ 0.5 µ 0.35 µ 0.25 µ 0.18 µ 0.13 µ 0.1 µ 0.07 µ

Xeon
180nm
32nm
Power Wall

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

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

Lower Frequency

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%
  - Performance: 1.2x
  - Power: 1.7x

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

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

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

Intel Nehalem Hex-Core

- 4-wide pipeline
Multicore is a necessity. How can we get better performance?
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
Hardware multithreading

Fine grained vs. Coarse grained hardware multithreading

Simultaneous multithreading (SMT)

Hyperthreads (Intel simultaneous multithreading)
  • Need to hide latency
Hardware multithreading

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>Multi-Core</th>
<th>Multi-Issue</th>
<th>HT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Num. Pipelines:</td>
<td>N</td>
<td>1</td>
<td>N</td>
</tr>
<tr>
<td>Pipeline Width:</td>
<td>N</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<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 die (aka 8 sockets)
4 core per socket
2 HT per core

Note: a socket is a processor, where each processor may have multiple processing cores, so this is an example of a multiprocessor multicore hyperthreaded system
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?
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
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 HW
• Balancing load over cores
• How do you write parallel programs?
  – ... without knowing exact underlying architecture?