Parallelism and Multicore

Anne Bracy
CS 3410
Computer Science
Cornell University

The slides are the product of many rounds of teaching CS 3410 by Professors Weatherspoon, Bala, Bracy, McKee, and Sirer.

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
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

```
   ALU/Br
```

```
   LW/SW
```
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></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>

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

\[
\frac{4 \text{ cycles}}{4 \text{ cycles}} = \text{CPI} = 0.8
\]
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
Load/store slot
cycle
Loop: nop

```
delay slot
lw   $t0, 0($s1) 1
lw   $t1, 4($s1) 2
```

```
addu $t0, $t0, $s2 3
addu $t1, $t1, $s2 4
```

```
addi $s1, $s1, +8 5
bne  $s1, $s3, Loop 6
```

6 cycles
8 cycles

6 cycles = CPI = 0.75

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

Load/store slot
cycle
lw $t0, 0($s1) 1
lw $t1, 4($s1) 2
nop 3
sw $t0, -8($s1) 4
sw $t1, -4($s1) 5

\[
\frac{5 \text{ cycles}}{8 \text{ instructions}} = \text{CPI} = 0.625
\]

8 cycles
5 cycles
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

Instruction fetch and decode unit

Reservation station
Reservation station
... Reservation station
Reservation station

Integer
Integer
...
Floating point
Load-store

Functional units

Commit unit

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
### 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>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?
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

- 180nm
- 32nm

Processors:
- Pentium III® processor
- Pentium II® processor
- Pentium Pro® processor
- Pentium® processor
- i386
- i486

Xeon
Power Wall

Power = capacitance * voltage$^2$ * frequency

In practice: Power ~ 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**
  - Power: Underclocked -20%
  - Performance: 1.0x
  - Power: 1.02x

- **Dual-Core**
  - Power: Underclocked -20%
  - Performance: 1.0x
  - Power: 1.02x

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

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

Intel Nehalem Hex-Core

4-wide pipeline
Parallel Programming

Q: So let's just all use multicore from now on!
A: Software must be written as a 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 (not covered)
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?