Parallelism, Multicore, and Synchronization

Hakim Weatherspoon
CS 3410
Computer Science
Cornell University

[Weatherspoon, Bala, Bracy, McKee, and Sirer, Roth, Martin]
Announcements

• P4-Buffer Overflow is due today
  • Due Tuesday, April 16\textsuperscript{th}

• C practice assignment
  • Due Friday, April 19\textsuperscript{th}

• P5-Cache project
  • Due Friday, April 26\textsuperscript{th}

• Prelim2
  • Thursday, May 2\textsuperscript{nd}, 7:30pm
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

• Multiply can be parallelized
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

Assumes load can be balanced across processors
Takeaway

Unfortunately, we cannot not 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
Performance Improvement 101

seconds
program = instructions
program x cycles
instruction x seconds
cycle

2 Classic Goals of Architects:

\(\downarrow\text{Clock period (\uparrow\text{Clock frequency})}\)

\(\downarrow\text{Cycles per Instruction (\uparrow\text{IPC})}\)
Clock frequencies have stalled

**Darling** of performance improvement for decades

Why is this no longer the strategy?

**Hitting Limits:**

- Pipeline depth
- Clock frequency
- Moore’s Law & Technology Scaling
- Power
Improving IPC via ILP

Exploiting Intra-instruction parallelism: Pipelining (decode A while fetching B)

Exploiting Instruction Level Parallelism (ILP): Multiple issue pipeline (2-wide, 4-wide, etc.)
  • Statically detected by compiler (VLIW)
  • Dynamically detected by HW

Dynamically Scheduled (OoO)
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”

How does HW detect and resolve hazards?
It doesn’t. 😊 Compiler must avoid hazards

Example: Static Dual-Issue 32-bit RISC-V
- Instructions come in pairs (64-bit aligned)
  - One ALU/branch instruction (or nop)
  - One load/store instruction (or nop)
RISC-V 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, s1, 0  # $t0=array element
       add t0, t0, s2  # add scalar in $s2
       sw  t0, s1, 0  # 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>
Techniques and Limits of Static Scheduling

Goal: larger instruction windows (to play with)
• Predication
• Loop unrolling
• Function in-lining
• Basic block modifications (superblocks, etc.)

Roadblocks
• Memory dependences (aliasing)
• Control dependences
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 RISC-V...

Loop:

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

### ALU/branch slot

<table>
<thead>
<tr>
<th>Loop:</th>
<th>ALU/branch slot</th>
<th>Load/store slot</th>
<th>cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>nop</td>
<td></td>
<td>lw t0, s1, 0</td>
<td>1</td>
</tr>
<tr>
<td>nop</td>
<td></td>
<td>lw t1, s1, 4</td>
<td>2</td>
</tr>
<tr>
<td>add</td>
<td>t0, t0, s2</td>
<td>nop</td>
<td>3</td>
</tr>
<tr>
<td>add</td>
<td>t1, t1, s2</td>
<td>sw t0, s1, 0</td>
<td>4</td>
</tr>
<tr>
<td>addi</td>
<td>s1, s1, +8</td>
<td>sw t1, s1, 4</td>
<td>5</td>
</tr>
<tr>
<td>bne</td>
<td>s1, s3, Loop</td>
<td>nop</td>
<td>6</td>
</tr>
</tbody>
</table>
Limits of Static Scheduling

Compiler scheduling for dual-issue RISC-V...

lw   t0, s1, 0     # load A
addi t0, t0, +1    # increment A
sw   t0, s1, 0     # store A
lw   t0, s2, 0     # load B
addi t0, t0, +1    # increment B
sw   t0, s2, 0     # 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, s1, 0</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, s1, 0</td>
<td>4</td>
</tr>
<tr>
<td>nop</td>
<td>lw t0, s2, 0</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, s2, 0</td>
<td>8</td>
</tr>
</tbody>
</table>
Improving IPC via ILP

Exploiting Intra-instruction parallelism:
• Pipelining (decode A while fetching B)

Exploiting Instruction Level Parallelism (ILP):
Multiple issue pipeline (2-wide, 4-wide, etc.)
• Statically detected by compiler (VLIW)
• Dynamically detected by HW
Dynamically Scheduled (OoO)
Dynamic Multiple Issue

aka **SuperScalar Processor** (c.f. Intel)

- CPU chooses multiple instructions to issue each cycle
- Compiler can help, by reordering instructions….
- … but CPU resolves hazards

Even better: **Speculation/Out-of-order Execution**

- Execute instructions as early as possible
- Aggressive register renaming (indirection to the rescue!)
- Guess results of branches, loads, etc.
- Roll back if guesses were wrong
- Don’t commit results until all previous insns committed
Dynamic Multiple Issue

![Diagram of dynamic multiple issue](image-url)
Effectiveness of OoO Superscalar

It was awesome, but then it stopped improving

Limiting factors?

• Programs dependencies
• Memory dependence detection $\rightarrow$ be conservative
  - e.g. Pointer Aliasing: A[0] += 1; B[0] *= 2;
• Hard to expose parallelism
  - Still limited by the fetch stream of the static program
• Structural limits
  - Memory delays and limited bandwidth
• Hard to keep pipelines full, especially with branches
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>
</tbody>
</table>

Those simpler cores did something very right.
Moore's Law:

- 4004
- 8008
- 8080
- 8088
- 286
- 386
- 486
- Pentium
- P4
- K8
- Itanium 2
- Dual-core Itanium 2
- K10
- Atom

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 Wall

Power = capacitance * voltage^2 * frequency

In practice: Power ~ 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?

<table>
<thead>
<tr>
<th></th>
<th>Performance</th>
<th>Power</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single-Core</td>
<td>1.0x</td>
<td>1.0x</td>
</tr>
<tr>
<td>Single-Core Overclocked +20%</td>
<td>1.2x</td>
<td>1.7x</td>
</tr>
<tr>
<td>Dual-Core Underclocked -20%</td>
<td>1.0x</td>
<td>1.02x</td>
</tr>
</tbody>
</table>
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>

Those simpler cores did something very right.
Inside the Processor

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

Intel Nehalem Hex-Core

4-wide pipeline
Exploiting Thread-Level parallelism

Hardware multithreading to improve utilization:

- Multiplexing multiple threads on single CPU
- Sacrifices latency for throughput
- Single thread cannot fully utilize CPU? *Try more!*

- Three types:
  - Course-grain (has preferred thread)
  - Fine-grain (round robin between threads)
  - Simultaneous (hyperthreading)
What is a thread?

Process: multiple threads, code, data and OS state
Threads: share code, data, files, **not** regs or stack

<table>
<thead>
<tr>
<th>code</th>
<th>data</th>
<th>files</th>
</tr>
</thead>
<tbody>
<tr>
<td>registers</td>
<td>stack</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>code</th>
<th>data</th>
<th>files</th>
</tr>
</thead>
<tbody>
<tr>
<td>registers</td>
<td>registers</td>
<td>registers</td>
</tr>
<tr>
<td>stack</td>
<td>stack</td>
<td>stack</td>
</tr>
</tbody>
</table>

thread

single-threaded process

multithreaded process
Standard Multithreading Picture

Time evolution of issue slots

- Color = thread, white = no instruction

---

<table>
<thead>
<tr>
<th>Time</th>
<th>4-wide Superscalar</th>
<th>CGMT</th>
<th>FGMT</th>
<th>SMT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Switch to thread B on thread A L2 miss</td>
<td>Switch threads every cycle</td>
<td>Insns from multiple threads coexist</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Hyperthreading

Multi-Core vs. Multi-Issue vs. HT

<table>
<thead>
<tr>
<th>Programs:</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Num. Pipelines:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pipeline Width:</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Hyperthreads

- HT = Multilssue + 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
- How do you write parallel programs?
  ... without knowing exact underlying architecture?
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
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
- How do you write parallel programs? ... without knowing exact underlying architecture?