Instruction Scheduling

Last time
– Register allocation

Today
– Instruction scheduling
  – The problem: Pipelined computer architecture
  – A solution: List scheduling
  – Improvements on this solution
Background: Pipelining Basics

Idea

- Begin executing an instruction before completing the previous one
## Idealized Instruction Data-Path

Instructions go through several stages of execution

<table>
<thead>
<tr>
<th>Stage 1</th>
<th>Stage 2</th>
<th>Stage 3</th>
<th>Stage 4</th>
<th>Stage 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction</td>
<td>Instruction</td>
<td>Execute</td>
<td>Memory Access</td>
<td>Register Write-back</td>
</tr>
<tr>
<td>Fetch</td>
<td>Decode &amp; Register Fetch</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| IF              | ID/RF                             | EX          | MEM                      | WB                       |

| time            |                                   |             |                          |                          |

| instructions    |                                   |             |                          |                          |

```
IF ID EX MM WB
IF ID EX MM WB
IF ID EX MM WB
IF ID EX MM WB
IF ID EX MM WB
IF ID EX MM WB
IF ID EX MM WB
IF ID EX MM WB
```
Pipelining Details

Observations
– Individual instructions are no faster (but throughput is higher)
– Potential speedup determined by number of stages (more or less)
– Filling and draining pipe limits speedup
– Rate through pipe is limited by slowest stage
– Less work per stage implies faster clock

Modern Processors
– Long pipelines: 5 (Pentium), 14 (Pentium Pro), 22 (Pentium 4), 31 (Prescott), 14 (Core i7), 8 ARM 11
– Issue width: 2 (Pentium), 4 (UltraSPARC) or more (dead Compaq EV8)
– Dynamically schedule instructions (from limited instruction window) or statically schedule (e.g., IA-64)
– Speculate
  – Outcome of branches
  – Value of loads (research)
What Limits Performance?

**Data hazards**
- Instruction depends on result of prior instruction that is still in the pipe

**Structural hazards**
- Hardware cannot support certain instruction sequences because of limited hardware resources

**Control hazards**
- Control flow depends on the result of branch instruction that is still in the pipe

**An obvious solution**
- Stall (insert bubbles into pipeline)
Stalls (Data Hazards)

Code

\[
\text{add } \$r1, \$r2, \$r3 \quad // \$r1 \text{ is the destination} \\
\text{mul } \$r4, \$r1, \$r1 \quad // \$r4 \text{ is the destination}
\]

Pipeline picture
Stalls (Structural Hazards)

Code

```
mul $r1,$r2,$r3  // Suppose multiplies take two cycles
mul $r4,$r5,$r6
```

Pipeline Picture
Stalls (Control Hazards)

Code

\[
\text{bz } \$r1, \text{ label} \quad // \text{ if } \$r1==0, \text{ branch to label} \\
\text{add } \$r2,\$r3,\$r4
\]

Pipeline Picture

```
  time
  \[
  | IF | ID | EX | MM | WB |
  \[
  | IF | ID | EX | MM | WB |
  \[
  | IF | ID | EX | MM | WB |
  \]
```

April 19, 2015
Hardware Solutions

Data hazards
- Data forwarding (doesn’t completely solve problem)
- Runtime speculation (doesn’t always work)

Structural hazards
- Hardware replication (expensive)
- More pipelining (doesn’t always work)

Control hazards
- Runtime speculation (branch prediction)

Dynamic scheduling
- Can address all of these issues
- Very successful
Context: The MIPS R2000

MIPS Computer Systems
- “First” commercial RISC processor (R2000 in 1984)
- Began trend of requiring nontrivial instruction scheduling by the compiler

What does MIPS mean?
- Microprocessor without Interlocked Pipeline Stages
Instruction Scheduling for Pipelined Architectures

Goal
- An efficient algorithm for reordering instructions to minimize pipeline stalls

Constraints
- Data dependences (for correctness)
- Hazards (can only have performance implications)

Simplifications
- Do scheduling after instruction selection and register allocation
- Only consider data hazards
Recall Data Dependences

Data dependence
- A data dependence is an ordering constraint on 2 statements
- When reordering statements, all data dependences must be observed to preserve program correctness

True (or flow) dependences
- Write to variable x followed by a read of x (read after write or RAW)
- Read of variable x followed by a write (WAR)

Anti-dependences
- Write to variable x followed by another write to x (WAW)

Output dependences
- Write to variable x followed by another write to x (WAW)

false dependences

\[
\begin{align*}
x &= 5; \\
p\text{rint}(x); \\
x &= 5; \\
p\text{rint}(x); \\
x &= 6; \\
x &= 5;
\end{align*}
\]
List Scheduling [Gibbons & Muchnick ’86]

Scope
– Basic blocks

Assumptions
– Pipeline interlocks are provided (i.e., algorithm need not introduce no-ops)
– Pointers can refer to any memory address (i.e., no alias analysis)
– Hazards take a single cycle (stall); here let’s assume there are two...
  – Load immediately followed by ALU op produces interlock
  – Store immediately followed by load produces interlock

Main data structure: dependence DAG
– Nodes represent instructions
– Edges \((s_1,s_2)\) represent dependences between instructions
  – Instruction \(s_1\) must execute before \(s_2\)
– Sometimes called data dependence graph or data-flow graph
Dependence Graph Example

Sample code  dst src src

1  addi  $r2,1,$r1
2  addi  $sp,12,$sp
3  st  a, $r0
4  ld  $r3,-4($sp)
5  ld  $r4,-8($sp)
6  addi  $sp,8,$sp
7  st  0($sp),$r2
8  ld  $r5,a
9  addi  $r4,1,$r4

Hazards in current schedule
– (3,4), (5,6), (7,8), (8,9)

Any topological sort is okay, but we want best one
Scheduling Heuristics

Goal

– Avoid stalls

What are some good heuristics?

– Does an instruction interlock with any immediate successors in the dependence graph?
– How many immediate successors does an instruction have?
– Is an instruction on the critical path?
Scheduling Heuristics (cont)

Idea: schedule an instruction earlier when...

- It does not interlock with the previously scheduled instruction (avoid stalls)
- It interlocks with its successors in the dependence graph (may enable successors to be scheduled without stall)
- It has many successors in the graph (may enable successors to be scheduled with greater flexibility)
- It is on the critical path (the goal is to minimize time, after all)
**Scheduling Algorithm**

Build dependence graph $G$

Candidates $\leftarrow$ set of all roots (nodes with no in-edges) in $G$

**while** Candidates $\neq \emptyset$

Select instruction $s$ from Candidates  \hspace{1cm} \{Using heuristics—in order\}

Schedule $s$

Candidates $\leftarrow$ Candidates $-$ $s$

Candidates $\leftarrow$ Candidates $\cup$ “exposed” nodes

\hspace{1cm} \{Add to Candidates those nodes whose predecessors have all been scheduled\}
## Scheduling Example

### Dependence Graph

- **Node 1**: `addi $r2,1,$r1
- **Node 2**: `addi $sp,12,$sp
- **Node 3**: `st 0($sp),$r2
- **Node 4**: `ld $r4,-8($sp)
- **Node 5**: `ld $r3,-4($sp)
- **Node 6**: `addi $sp,8,$sp
- **Node 7**: `st
- **Node 8**: `ld $r5,a
- **Node 9**: `addi $r4,1,$r4

### Scheduled Code

```assembly
3    st    a, $r0
2    addi  $sp,12,$sp
5    ld    $r4,-8($sp)
4    ld    $r3,-4($sp)
8    ld    $r5,a
1    addi  $r2,1,$r1
6    addi  $sp,8,$sp
7    st    0($sp),$r2
9    addi  $r4,1,$r4
```

### Candidates

1. `addi $r2,1,$r1
2. `addi $sp,12,$sp

### Hazards in new schedule

- `-(8,1)`
Scheduling Example (cont)

Original code

<table>
<thead>
<tr>
<th>Original code</th>
<th>Hazards in original schedule</th>
</tr>
</thead>
<tbody>
<tr>
<td>1  addi $r2,1,$r1</td>
<td>– (3,4), (5,6), (7,8), (8,9)</td>
</tr>
<tr>
<td>2  addi $sp,12,$sp</td>
<td></td>
</tr>
<tr>
<td>3  st a, $r0</td>
<td></td>
</tr>
<tr>
<td>4  ld $r3,-4($sp)</td>
<td></td>
</tr>
<tr>
<td>5  ld $r4,-8($sp)</td>
<td></td>
</tr>
<tr>
<td>6  addi $sp,8,$sp</td>
<td></td>
</tr>
<tr>
<td>7  st 0($sp),$r2</td>
<td></td>
</tr>
<tr>
<td>8  ld $r5,a</td>
<td></td>
</tr>
<tr>
<td>9  addi $r4,1,$r4</td>
<td></td>
</tr>
</tbody>
</table>

Hazards in new schedule

<table>
<thead>
<tr>
<th>Hazards in new schedule</th>
</tr>
</thead>
<tbody>
<tr>
<td>– (8,1)</td>
</tr>
</tbody>
</table>
Complexity

**Quadratic in the number of instructions**

- Building dependence graph is $O(n^2)$
- May need to inspect each instruction at each scheduling step: $O(n^2)$
- In practice: closer to linear
Improving Instruction Scheduling

**Techniques**
- Scheduling loads
- Register renaming
- Loop unrolling
- Software pipelining
- Predication and speculation

\{ Deal with data hazards  \\
\{ Deal with control hazards  \\

Scheduling Loads

Reality
- Loads can take many cycles (slow caches, cache misses)
- Many cycles may be wasted

Most modern architectures provide non-blocking (delayed) loads
- Loads never stall
- Instead, the use of a register stalls if the value is not yet available
- Scheduler should try to place loads well before the use of target register
Hiding latency

- Place independent instructions behind loads

- How many instructions should we insert?
  - Depends on latency
  - Difference between cache miss and cache hits are growing
  - If we underestimate latency: Stall waiting for the load
  - If we overestimate latency: Hold register longer than necessary
    Wasted parallelism
Balanced Scheduling [Kerns and Eggers’92]

Idea
– Impossible to know the latencies statically
– Instead of estimating latency, balance the ILP (instruction-level parallelism) across all loads
– Schedule for characteristics of the code instead of for characteristics of the machine

Balancing load
– Compute load level parallelism

\[ LLP = 1 + \frac{\text{# independent instructions}}{\text{# of loads that can use this parallelism}} \]
Balanced Scheduling Example

Example

LLP for L0 = 1 + 4/2 = 3
LLP for L1 = 1 + 2/1 = 3

<table>
<thead>
<tr>
<th></th>
<th>list scheduling</th>
<th>balanced scheduling</th>
</tr>
</thead>
<tbody>
<tr>
<td>w=5</td>
<td>w=1</td>
<td></td>
</tr>
<tr>
<td>L0</td>
<td>L0</td>
<td>L0</td>
</tr>
<tr>
<td>X0</td>
<td>L1</td>
<td>X0</td>
</tr>
<tr>
<td>X1</td>
<td>X0</td>
<td>X1</td>
</tr>
<tr>
<td>X2</td>
<td>X1</td>
<td></td>
</tr>
<tr>
<td>X3</td>
<td>X2</td>
<td></td>
</tr>
<tr>
<td>X4</td>
<td>X3</td>
<td></td>
</tr>
</tbody>
</table>

Pessimistic Optimistic
Register Renaming

Idea

- Reduce false data dependences by reducing register reuse
- Give the instruction scheduler greater freedom

Example

\[
\begin{align*}
\text{add} & \quad \$r1, \; \$r2, \; 1 \\
\text{st} & \quad \$r1, \; [\$fp+52] \\
\text{mul} & \quad \$r1, \; \$r3, \; 2 \\
\text{st} & \quad \$r1, \; [\$fp+40] \\
\text{add} & \quad \$r1, \; \$r2, \; 1 \\
\text{st} & \quad \$r1, \; [\$fp+52] \\
\text{mul} & \quad \$r11, \; \$r3, \; 2 \\
\text{st} & \quad \$r11, \; [\$fp+40]
\end{align*}
\]
Loop Unrolling

Idea
– Replicate body of loop and iterate fewer times
– Reduces loop overhead (test and branch)
– Creates larger loop body ⇒ more scheduling freedom

Example

L: ldf [r1], f0
   fadds f0, f1, f2
   stf f2, [r1]
   sub r1, 4, r1
   cmp r1, 0
   bg L
   nop

Cycles per iteration: 12
Loop Unrolling Example

Sample loop

L: ldf [r1], f0
    fadds f0, f1, f2
    ldf [r1-4], f10
    fadds f10, f1, f12
    stf f2, [r1]
    stf f12, [r1-4]
    sub r1, 8, r1
    cmp r1, 0
    bg L
    nop

Cycles per iteration: 14/2 = 7
(71% speedup!)

The larger window lets us hide the latency of the fadds instruction
Phase Ordering Problem

Register allocation
- Tries to reuse registers
- Artificially constrains instruction schedule

Just schedule instructions first?
- Scheduling can dramatically increase register pressure

Classic phase ordering problem
- Tradeoff between memory and parallelism

Approaches
- Consider allocation & scheduling together
- Run allocation & scheduling multiple times
  (schedule, allocate, schedule)
Concepts

Instruction scheduling
– Reorder instructions to efficiently use machine resources
– List scheduling

Improving instruction scheduling
– Balanced scheduling
  – Consider characteristics of the program
– Register renaming
– Loop unrolling

Phase ordering problem
Next Time

Lecture

– More instruction scheduling
### Scheduling Example

#### Dependence Graph

![Dependence Graph](image)

#### Scheduled Code

```
3  st   a, $r0
2  addi $sp,12,$sp
4  ld   $r3,-4($sp)
5  ld   $r4,-8($sp)
8  ld   $r5,a
1  addi $r2,1,$r1
6  addi $sp,8,$sp
7  st   0($sp),$r2
9  addi $r4,1,$r4
```

#### Candidates

```
1  addi $r2,1,$r1
2  addi $sp,12,$sp
3  st   0($sp),$r2
4  ld   $r3,-4($sp)
5  ld   $r4,-8($sp)
8  ld   $r5,a
9  addi $r4,1,$r4
```

#### Hazards in New Schedule

\(-(8,1)\)
Scheduling Example

Dependence Graph

Scheduled Code

3  st  a, $r0
2  addi $sp,12,$sp
4  ld  $r3,-4($sp)
5  ld  $r4,-8($sp)
8  ld  $r5,a
6  addi $sp,8,$sp
1  addi $r2,1,$r1
7  st  0($sp),$r2
9  addi $r4,1,$r42

Hazards in New Schedule

−(8,1)

Candidates

1  addi $r2,1,$r1
2  addi $sp,12,$sp
3  st  0($sp),$r2
8  ld  $r5,a
9  addi $r4,1,$r4
Scheduling Example

Dependence Graph

Scheduled Code

3  st  a, $r0
2  addi  $sp,12,$sp
4  ld  $r3,-4($sp)
5  ld  $r4,-8($sp)
1  addi  $r2,1,$r1
6  addi  $sp,8,$sp
8  ld  $r5,a
7  st  0($sp),$r2
9  addi  $r4,1,$r42

Candidates

1  addi  $r2,1,$r1
2  addi  $sp,12,$sp

Hazards in New Schedule

−(8,1)
Scheduling Example

Dependence Graph

Scheduled Code
3  st   a, $r0
2  addi $sp,12,$sp
4  ld   $r3,-4($sp)
5  ld   $r4,-8($sp)
6  addi $sp,8,$sp
1  addi $r2,1,$r1
7  st   0($sp),$r2
8  ld   $r5,a
9  addi $r4,1,$r4

Hazards in New Schedule
−(5,6), (7,8)

Candidates
1  addi $r2,1,$r1
2  addi $sp,12,$sp
3  st   a, $r0
Software Pipelining

Basic Idea

– Ideally, we could completely unroll loops and have complete freedom in scheduling across iteration boundaries
– Software pipelining is a systematic approach to scheduling across iteration boundaries without doing loop unrolling

– Use control-flow profiles to identify most frequent path through a loop
– If the most frequent path has hazards, try to move some of the long latency instructions to previous iterations of the loop

– Three parts of a software pipeline
  – **Kernel**: Steady state execution of the pipeline
  – **Prologue**: Code to fill the pipeline
  – **Epilogue**: Code to empty the pipeline
Software Pipelining Example

Sample loop (reprise)

L: ldf [r1], f0
  fadds f0, f1, f2
  stf f2, [r1]
  sub r1, 4, r1
  cmp r1, 0
  bg L
  nop

Cycles per iteration: 12
Software Pipelining Example (cont)

```
ldf [r1], f0
fadds f0, f1, f2
stf f2, [r1]
sub r1, 4, r1
```

```
ldf [r1-8], f0
fadds f0, f1, f2
stf f2, [r1]
sub r1, 4, r1
```
Software Pipelining Example (cont)

Sample loop

```
lfd [r1], f0
fadds f0, f1, f2
lfd [r1-4], f0
L: stf f2, [r1]
fadds f0, f1, f2
lfd [r1-8], f0
cmp r1, 8
bg L
sub r1, 4, r1
stf f2, [r1]
sub r1, 4, r1
fadds f0, f1, f2
stf f2, [r1]
```

Cycles per iteration: 7 (71% speedup!)