Instruction Scheduling

Last time
– Register allocation

Today
– Instruction scheduling
  – The problem: Pipelined computer architecture
  – A solution: List scheduling

Background: Pipelining Basics

Idea
– Begin executing an instruction before completing the previous one

Without Pipelining

With Pipelining
**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 $\Rightarrow$ ID/RF $\Rightarrow$ EX $\Rightarrow$ MEM $\Rightarrow$ 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), 20 or 31 (Pentium 4)
- Issue 2 (Pentium), 4 (UltraSPARC) or more (dead Compaq EV8) instructions per cycle
- Dynamically schedule instructions (from limited instruction window) or statically schedule (e.g., IA-64)
- Hyperthreading or simultaneous multi-threading
- Speculate: Outcome of branches, Value of loads
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
\[
\begin{align*}
\text{add} & \quad \text{\$r1},\text{\$r2},\text{\$r3} & \quad \text{// \$r1 is the destination} \\
\text{mul} & \quad \text{\$r4},\text{\$r1},\text{\$r1} & \quad \text{// \$r4 is the destination}
\end{align*}
\]

Pipeline picture

![Pipeline diagram]
Stalls (Structural Hazards)

Code

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

Pipeline Picture

```
IF  ID  EX  EX  MM  WB
```

Stalls (Control Hazards)

Code

```plaintext
bz $r1, label  // if $r1==0, branch to label
add $r2,$r3,$r4
```

Pipeline Picture

```
IF  ID  EX  MM  WB
```

Instructions: time

```
IF  ID  EX  EX  MM  WB
```
**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

---

**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)

**Possible Simplifications**
- Do scheduling after instruction selection and register allocation
- Only consider data hazards
**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)

```
x = 5;
print (x);
```

**Anti-dependences**
– Read of variable x followed by a write (WAR)

```
print (x);
x = 5;
```

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

```
x = 6;
x = 5;
```

**Register Renaming**

**Idea**
– Reduce false data dependences by reducing register reuse
– Give the instruction scheduler greater freedom

**Example**

```
add $r1, $r2, 1  add $r1, $r2, 1
st $r1, [$fp+52]  st $r1, [$fp+52]
mul $r1, $r3, 2   mul $r11, $r3, 2
st $r1, [$fp+40]  st $r11, [$fp+40]
```

```
add $r1, $r2, 1
mul $r11, $r3, 2
st $r1, [$fp+52]
st $r11, [$fp+40]
```
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)

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

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

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

**Consider these questions**
- Does an instruction interlock with any immediate successors in the dependence graph? IOW is the delay greater than 1?
- 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 \( \{ \text{Using heuristics—in order} \} \)

Schedule \( s \)

Candidates \( \leftarrow \) Candidates \( \setminus \) \( s \)

Candidates \( \leftarrow \) Candidates \( \cup \) “exposed” nodes

\( \{ \text{Add to Candidates those nodes whose predecessors have all been scheduled} \} \)
Scheduling Example

Dependence Graph

Scheduled Code

Candidates

Hazards in new schedule

Scheduling Example (cont)

Original code

Hazards in original schedule

Hazards in new schedule
**Complexity**

**Quadratic in the number of instructions**
- Building dependence graph is $O(n^2)$
- May need to inspect each instruction at each scheduling step: $O(n)$
- In practice: closer to linear

**Example 10.6 in book**

**Stalls**
- LD takes two clocks but
  - ST to same can directly follow
  - any LD can directly follow

**Flow dependences**
- i1 to i2, i3 to i4, i4 to i5, i5 to i6
- i2 to i3?

**Anti dependences**
- i4 to i5
- i1 to i7, i3 to i7

**Output dependences**
- i3 to i4, i4 to i5
- i2 to i7
Improving Instruction Scheduling

Techniques

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

Deal with data hazards

Deal with control hazards

Register Renaming

Idea

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

Example

```
add $r1, $r2, 1       add $r1, $r2, 1
st $r1, [$fp+52]      st $r1, [$fp+52]
mul $r1, $r3, 2       mul $r11, $r3, 2
st $r1, [$fp+40]      st $r11, [$fp+40]
```

```
add $r1, $r2, 1
mul $r11, $r3, 2
st $r1, [$fp+52]
st $r11, [$fp+40]
```
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

Scheduling Loads (cont)

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
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:
\begin{align*}
\text{ldf} & \quad f0, [r1] \\
\text{fadds} & \quad f2, f0, f1 \\
\text{stf} & \quad [r1], f2 \\
\text{sub} & \quad r1, 4, r1 \\
\text{cmp} & \quad r1, 0 \\
\text{bg} & \quad L \\
\text{nop} & \\
\end{align*}

Cycles per iteration: 9

Loop Unrolling Example

Sample loop
L:
\begin{align*}
\text{ldf} & \quad f0, [r1] \\
\text{fadds} & \quad f2, f0, f1 \\
\text{ldf} & \quad f10, [r1-4] \\
\text{fadds} & \quad f12, f10, f1 \\
\text{stf} & \quad [r1], f2 \\
\text{stf} & \quad [r1-4], f12 \\
\text{sub} & \quad r1, 8, r1 \\
\text{cmp} & \quad r1, 0 \\
\text{bg} & \quad L \\
\text{nop} & \\
\end{align*}

Cycles per iteration: 12/2 = 6
(1.5 speedup!)

The larger window lets us hide some of the latency of the \texttt{fadds} instruction
**Loop Unrolling Summary**

**Benefit**
- Loop unrolling allows us to schedule code across iteration boundaries, providing more scheduling freedom

**Issues**
- How much unrolling should we do?
  - Try various unrolling factors and see which provides the best schedule?
  - Unroll as much as possible within a code expansion budget?
- An alternative: **Software pipelining**

**Software Pipelining**

**Basic idea**
- **Software pipelining** is a systematic approach to scheduling across iteration boundaries without doing loop unrolling
- Try to move the long latency instructions to previous iterations of the loop
- Use independent instructions to hide their latency

- 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
Visualizing Software Pipelining

Software Pipelining versus Loop Unrolling
SW Pipelining (Step 1: Construct DAG and Assign Registers)

```c
int A[100], B[100], C[100];
for (i=0; i<100; i++) {
}
```

Example assumes infinite functional units and single-cycle latency.

SW Pipelining (Step 2: “Unroll”, Satisfy Latencies, Find Pattern)

This pattern does not work!!
SW Pipelining (Step 3: Satisfy register constraints)

```
int A[100], B[100], C[100];
for (i=0; i<100; i++) {
}
```

Example assumes pg. 739 machine.

SW Pipelining (Step 1: Construct DAG and Assign Registers)

```
int A[100], B[100], C[100];
for (i=0; i<100; i++) {
}
```
This pattern does not work!!
SW Pipelining and Loop Unrolling Summary

Unrolling removes branching overhead and helps tolerate data dependence latency

SW pipelining maintains max parallelism in steady state through continuous tolerance of data dependence latency

Both work best with loops that are parallel, getting ILP by taking instructions from different iterations

Software Pipelining

Complications
- What if there is control flow within the loop?
  - Use control-flow profiles to identify most frequent path through the loop
  - Optimize for the most frequent path
- How do we identify the most frequent path?
  - Profiling
Concepts

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

Suggested Exercises
– for the simplifying register allocators [Chaitin and Briggs], can you prove that neither of the algorithms end up in an infinite loop where they are spilling the same temporary over and over again?
– exercise 10.2.1 and 10.2.3
– for exercise 10.3.2, use list scheduling algorithm covered in class, but try with prioritized order suggested in book and heuristics discussed in class
– by hand, come up with a schedule for the example on slide 19 that has no stalls

Next Time

Lecture
– Data-flow analysis theory, read chapter 9 through 9.3

Suggested Exercises
– for the simplifying register allocators [Chaitin and Briggs], can you prove that neither of the algorithms end up in an infinite loop where they are spilling the same temporary over and over again?
– exercise 10.2.1 and 10.2.3
– for exercise 10.3.2, use list scheduling algorithm covered in class, but try with prioritized order suggested in book and heuristics discussed in class
– by hand, come up with a schedule for the example on slide ?? that has no stalls