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

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)
- Issue 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
```
add $r1,$r2,$r3
mul $r4,$r1,$r1
```

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

Pipeline Picture
### 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)
  
  ```
  x = 5;
  print (x);
  ```

#### Output dependences
- Write to variable x followed by another write to x (WAW)
  
  ```
  x = 5;
  ```

### Register Renaming

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

#### Example

```plaintext
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]
```

```plaintext
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 addi</td>
<td>$r2,1,$r1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2 addi</td>
<td>$sp,12,$sp</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3 st</td>
<td>a, $r0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4 ld</td>
<td>$r3,-4($sp)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5 ld</td>
<td>$r4,-8($sp)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6 addi</td>
<td>$sp,8,$sp</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7 st</td>
<td>0 ($sp), $r2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8 ld</td>
<td>$r5,a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9 addi</td>
<td>$r4,1,$r4</td>
<td></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 ← set of all roots (nodes with no in-edges) in G
while Candidates ≠ ∅

Select instruction s from Candidates
{Using heuristics—in order}
Schedule s
Candidates ← Candidates − s
Candidates ← Candidates ∪ “exposed” nodes
{Add to Candidates those nodes whose predecessors have all been scheduled}

Scheduling Example

Dependence Graph

Scheduled Code

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

Hazards in new schedule

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

Hazards in original schedule

(3,4), (5,6), (7,8), (8,9)

Hazards in new schedule

(8,1)

Scheduling Example (cont)

Original code

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 original schedule

(3,4), (5,6), (7,8), (8,9)

Hazards in new schedule

(8,1)
### 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

### 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**
- More instruction scheduling
- loop unrolling
- software pipelining
Improving Instruction Scheduling

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

Deal with data hazards
Deal with control hazards