Instruction Scheduling

Last week
– 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  // \$r1 is the destination
mul \$r4,\$r1,\$r1  // \$r6 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
bz \$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)

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

---

## 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>
<th>Dependence graph</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 addi</td>
<td>$r2,1,$r1</td>
<td>2 addi</td>
<td>$sp,12,$sp</td>
<td></td>
</tr>
<tr>
<td>3 st</td>
<td>a, $r0</td>
<td>4 ld</td>
<td>$r3,-4($sp)</td>
<td>5 ld</td>
</tr>
<tr>
<td>6 addi</td>
<td>$sp,8,$sp</td>
<td>7 st</td>
<td>($sp),$r2</td>
<td>8 ld</td>
</tr>
<tr>
<td>9 addi</td>
<td>$r4,1,$r4</td>
<td>10</td>
<td></td>
<td>11</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?
– 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 x from Candidates {Using heuristics— in order}
    Schedule x
    Candidates ← Candidates − x
    Candidates ← Candidates U “exposed” nodes {Add to Candidates those nodes whose predecessors have all been scheduled}
### Scheduling Example

#### Dependence Graph

<table>
<thead>
<tr>
<th>Instruction</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>addi</code></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>addi</code></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>addi</code></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>addi</code></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>st</code></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Candidates

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

#### Scheduling Code

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

#### Hazards in new schedule

- (8,1)

#### Hazards in original schedule

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

### 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

### Concepts

**Instruction scheduling**

- Reorder instructions to efficiently use machine resources
- List scheduling
Next Time

Lecture
- More instruction scheduling
  - loop unrolling
  - software pipelining