18-447 Lecture 9: Control Hazard and Resolution

James C. Hoe
Department of ECE
Carnegie Mellon University

Housekeeping

- Your goal today
  - “simple” control flow resolution in in-order pipelines
  - there is more fun to come on this

- Notices
  - Lab 2, status check next week, due wk of 2/27
  - HW2, due next Wed
  - Midterm 2/27 in class; covers Lectures 1~10
  - practice midterm-1 from S’16

- Readings
  - P&H Ch 4
Format of the Midterm

- **Coverage**
  - lectures (L1~L10), HWs, projects, assigned readings (textbooks and papers)
- **Types of questions**
  - freebies: can you remember the materials
  - probing: did you understand the materials
  - applied: can you apply the materials in original thoughts
- **100 minutes, 100 points**
  - if a question is worth 5 points, don’t spend 20 minutes on it
  - skip questions you can’t do and come back to them later
  - closed-book, one 2-sided 8½x11-in² hand-written cribsheet
  - no calculators, no cell phone call, no electronics
    - *** use pencil or black/blue ink only
    - *** be on time, 12:30 sharp!!!

Instruction Dependence

- **Data Dependence**
  - true dependence ⇒ Read after Write (RAW)
    - instruction must wait for all required input operands
  - anti-Dependence ⇒ Write after Read (WAR)
    - younger write must not clobber a still-pending older read
  - output dependence ⇒ Write after Write (WAW)
    - older write must not clobber an already-finished younger write

- **Control Dependence (or Procedural Dependence)**
  - all instructions are dependent by control flow
  - every instruction use and set the PC

  Control dependence is data dependence on the PC!!
PC Data Hazard Analysis

<table>
<thead>
<tr>
<th>R/I-Type</th>
<th>LW</th>
<th>SW</th>
<th>Bxx</th>
<th>Jal</th>
<th>Jalr</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>use</td>
<td>use</td>
<td>use</td>
<td>use</td>
<td>use</td>
</tr>
<tr>
<td>ID</td>
<td>produce</td>
<td>produce</td>
<td>produce</td>
<td>produce</td>
<td>produce</td>
</tr>
<tr>
<td>EX</td>
<td>produce</td>
<td>produce</td>
<td>produce</td>
<td>produce</td>
<td>produce</td>
</tr>
<tr>
<td>MEM</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- PC dependence distance is 1
- PC hazard distance is at least 1
- Does that mean we must stall after every inst. fetch?
  - IF stage can’t know which PC to fetch next at least until the current PC is fetched and instruction decoded

Resolve Control Hazard by Stalling

We are talking about non-control-flow instructions
There is only 1 way to beat “true” dependence . . . .

Control Speculation for Dummies

- Don’t wait for data-dependence on PC to resolve, just guess nextPC = PC+4 to keep fetching every cycle
  
  Is this a good guess?
  
  What do you lose if you guessed incorrectly?

- Only ~20% of the instruction mix is control flow
  - ~50% of “forward” control flow (i.e., if-then-else) is taken
  - ~90% of “backward” control flow (i.e., end-of-loop) is taken

  Over all, typically ~70% taken and ~30% not taken
  [Lee and Smith, 1984]

- Expect “nextPC = PC+4” ~86% of the time, but what about the remaining 14%?
Control Speculation: PC+4

- branch target (Inst₃) is fetched
- all instructions fetched since inst₃ (so called “wrong-path” instructions) must be flushed

Pipeline Flush on Misprediction

- ALU instructions killed
- all instructions fetched since inst₃ must be flushed
Pipeline Flush on Misprediction

<table>
<thead>
<tr>
<th>t0</th>
<th>t1</th>
<th>t2</th>
<th>t3</th>
<th>t4</th>
<th>t5</th>
<th>t6</th>
<th>t7</th>
<th>t8</th>
<th>t9</th>
<th>t10</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>h</td>
<td>i</td>
<td>j</td>
<td>k</td>
<td>l</td>
<td>m</td>
<td>n</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ID</td>
<td>h</td>
<td>i</td>
<td>bub</td>
<td>k</td>
<td>l</td>
<td>m</td>
<td>n</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EX</td>
<td>h</td>
<td>bub</td>
<td>bub</td>
<td>k</td>
<td>l</td>
<td>m</td>
<td>n</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MEM</td>
<td>h</td>
<td>bub</td>
<td>bub</td>
<td>k</td>
<td>l</td>
<td>m</td>
<td>n</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WB</td>
<td>h</td>
<td>bub</td>
<td>bub</td>
<td>k</td>
<td>l</td>
<td>m</td>
<td>n</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

branch resolved

Performance Impact

- correct guess \(\Rightarrow\) no penalty most of the time!!
- incorrect guess \(\Rightarrow\) 2 bubbles
- Assume
  - no data hazards
  - 20% control flow instructions
  - 70% of control flow instructions are taken
  - IPC = \(\frac{1}{1 + (0.20 \times 0.7) \times 2}\) = \(\frac{1}{1 + 0.14 \times 2}\) = \(\frac{1}{1.28}\) = 0.78

percentage of penalty for wrong guesses a wrong guess
Can we reduce either of the two penalty terms?
Reducing Mispredict Penalty

MIPS R2000 ISA Control Flow Design

- Simple address calculation based on instruction only
  - branch PC-offset: 16-bit full-addition + 14-bit half-addition
  - jump PC-offset: concatenation only

- Simple branch condition based on RF
  - one register relative (>, <, =) to 0
  - equality between 2 registers
    No addition/subtraction necessary!

Explicit ISA design choices to make possible branch resolution in ID of a 5-stage pipeline
Branch Resolved in ID

\[ IPC = \frac{1}{1 + (0.2 \times 0.7) \times 1} = 0.88 \]

Branch Delay Slots

- Bxx r- L1
- PC+4
- L1 if taken else PC+8

- PC+4 is already in the pipeline
  - throwing PC+4 away cost 1 bubble
  - letting PC+4 finish won’t hurt performance

- R2000 branches have 1 instruction architectural latency
  - instruction immediately after a branch is always executed
    (no pipeline flush logic)
  - branch target takes effect on the 2nd instruction
  - if delay slot can always do useful work, effective IPC=1
    ~80% of delay slots can be filled automatically by compilers

\[ IPC = \frac{1}{1 + (0.2 \times 0.2) \times 1} = 0.96 \]
Filling Delay Slots by Static Reordering Transformation

reordering data independent (RAW, WAW, WAR) instructions within basic block does not change program

within same basic block

a new instruction added to not-taken path??

Safe?

MIPS R2000 Interlock Free Pipeline

<table>
<thead>
<tr>
<th>R/I-Type</th>
<th>LW</th>
<th>SW</th>
<th>Bxx</th>
<th>Jal</th>
<th>Jalr</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>use</td>
<td>use</td>
<td>use</td>
<td>use</td>
<td>use</td>
</tr>
<tr>
<td>ID</td>
<td>produce</td>
<td>produce</td>
<td>produce</td>
<td>produce</td>
<td>produce</td>
</tr>
<tr>
<td>EX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MEM</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- PC hazard distance is 1 (for both control-flow and non-control-flow instructions)
- Control delay slot makes PC dependence distance 2 (true for ALU instructions as well, i.e., nextnextPC = nextPC+4)
Strangeness in Semantics

Where should you end up from \_s (assuming previous instruction executed was ADD)?

\[
\begin{align*}
\_s: & \quad j \text{ L1} \\
& \quad j \text{ L2} \\
& \quad j \text{ L3} \\
\text{L1:} & \quad j \text{ L4} \\
\text{L2:} & \quad j \text{ L5} \\
\text{L3:} & \quad \text{foo} \\
\text{L4:} & \quad \text{bar} \\
\text{L5:} & \quad \text{baz}
\end{align*}
\]

MIPS had to ban control-flow instructions from control delay slots (but not because of this)

Wait just a second . . . .

<table>
<thead>
<tr>
<th>R/I-Type</th>
<th>LW</th>
<th>SW</th>
<th>Bxx</th>
<th>J</th>
<th>Jr</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ID</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EX</td>
<td>use</td>
<td>use</td>
<td>use</td>
<td>(cheat)</td>
<td>(cheat)</td>
</tr>
<tr>
<td>MEM</td>
<td>produce</td>
<td>(use)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Last lecture (without control flow)
  - with forwarding, RF hazard distance is 0 except for RAW dependence on LW where it is 1
  - load delay slot semantics ensures a dependent instruction to be at least distance 2
- But now we are using in ID . . .
Forwarding Paths (v1)

Assumes RF forwards internally

Forwarding Paths (v2)
Making a Better Guess
(for when it is not MIPS or 5-stage)

- For non-control-flow instructions
  - can’t do better than guessing nextPC=PC+4
  - still tricky since must guess before you know it is control-flow or non-control-flow
- For control-flow instructions
  - why not always guess in the taken direction since 70% correct
  - must guess nextPC before the branch instruction is fetched
    (and branch target is encoded in the instruction)

⇒ Only solid information is the current fetch PC !!!
⇒ Fortunately,
  - PC-offset branch/jump target is static
  - we are allowed to be wrong some of the time

Branch Target Buffer (magic version)

- BTB
  - a giant table indexed by PC
  - returns the “guess” for nextPC
- When encountering a PC for the first time, after decoding, record in BTB . . .
  - PC + 4 if ALU/LD/ST
  - PC+offset if Branch or Jump
  - ?? if JR
- Effectively guessing branches are always taken (and where to)
  IPC = 1 / [ 1 + (0.20*0.3) * 2 ]
  = 0.89
BTB (Reality)

- “Hash” PC into a $2^N$ entry table
- What happens when two branches hash to the same entry?

The Locality Principle

- One’s recent past is a very good predictor of one’s near future.
  - **Temporal Locality**: If you just did something, it is very likely that you will do the same thing again soon
    - since you are here today, there is a good chance you will be here again and again regularly
    - inverse is also true
  - **Spatial Locality**: If you just did something, it is very likely you will do something similar/related
    - every time I find you in this room, you are probably sitting in the same seat (or nearby)
    - you are probably sitting near the same people

Programs are even more predictable than people