**Performance of tournament predictors**

In the last lecture, we saw the design of the “tournament” predictor used by the Alpha 21264.

The Alpha’s predictor is very successful.

- On the SPECfp 95 benchmarks, there is less than 1 misprediction per 1000 completed instructions.
- On the SPECint 95 benchmarks, there are about 11.5 mispredictions per 1000 completed instructions.

The textbook (p. 208) shows a graph of mispredictions vs. predictor size for the SPEC89 benchmarks. The “correlating predictor” here is Gshare.

If we examine individual benchmarks, we can see that the local predictors work pretty well for floating-point benchmarks (the first three at the top of the graph), while global information is more important for integer benchmarks.
Branch-target buffers

[H&P §3.5] A branch-prediction buffer is accessed during the ID cycle, so that at the end of ID we know—

- the branch-target address (computed during ID),
- the fall-through address (computed during IF), and
- the prediction.

If the prediction is not the next sequential instruction, we have a one-cycle delay:

What is the penalty for correctly predicting a taken branch?

We would like to do better than this … no penalty for correct predictions.
In order to do this, we must know—

in the IF stage. How can we achieve this?

Let’s store the branch-target address along with the branch address in the buffer!

This is called a *branch-target buffer* (BTB).

Each entry in this buffer has the form

\[(\text{addr. where branch is found}, \text{branch-target addr.}[\text{, prediction bits}])\]

In this case, we may not even need to store non-taken branches in the buffer. Why?

With a BTB,

- The PC of the instruction being fetched is matched against a set of instruction addresses in the “look-up” column.
- If there’s a match, then the PC is set to the corresponding value in the “predicted PC” column.

(What does this organization remind you of?)

Since the branch target is known during the IF stage, fetching can immediately proceed from the target address, and the branch penalty is 0 cycles.
Let's make a table of penalties. Let's assume we're not storing non-taken branches in the buffer. (Refer to the diagram on the next page.)

<table>
<thead>
<tr>
<th>Instruction in buffer?</th>
<th>Prediction</th>
<th>Outcome</th>
<th>Penalty cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td>Taken</td>
<td>Taken</td>
<td>0</td>
</tr>
<tr>
<td>Yes</td>
<td>Taken</td>
<td>Not taken</td>
<td>2</td>
</tr>
<tr>
<td>No</td>
<td>—</td>
<td>Taken</td>
<td>2</td>
</tr>
<tr>
<td>No</td>
<td>—</td>
<td>Not taken</td>
<td>0</td>
</tr>
</tbody>
</table>

*Exercise:* Assume that—

- 80% of branch instructions are found in the BTB.
- 95% prediction accuracy for these instructions.
- 1/4 of instructions are branches.
- 60% of branches are taken.
- The pipeline has 5 stages.
- 1 cycle per instruction, except for branches.

What is the CPI of this machine?
Based on notes from Drs. Tom Conte & Eric Rotenberg of NCSU
Figures from CAQA used with permission of Morgan Kaufmann Publishers. © 2003 Elsevier Science (USA)
Unlike a branch-prediction buffer, a BTB “needs” to store a complete branch address, not just certain bits. Why?

Optimization: Store target instructions, instead of target addresses.

How does this help?

- Suppose the BTB is so large that it takes more than one cycle to access it …

- It allows branch folding—elimination of unconditional branches, and even conditional branches if WHEN is known in advance.

**Delay slots**

So far, we have looked at reducing branch penalty by changing the hardware. It is also possible to pursue a software solution.

**Solution**: Redefine branches so they do not take effect until after the next instruction. This is called a delayed branch.

After a delayed branch, a no-op (a do-nothing instruction) is needed (“delayed branch” column below).

But the no-op can frequently be removed if the code is rearranged (“optimized delayed branch” column below).

<table>
<thead>
<tr>
<th>Address</th>
<th>Normal Branch</th>
<th>Delayed Branch</th>
<th>Optimized Delayed Branch</th>
</tr>
</thead>
<tbody>
<tr>
<td>200</td>
<td>LD R1, X</td>
<td>LD 3, X</td>
<td>LD 3, X</td>
</tr>
<tr>
<td>204</td>
<td>DADD R2, R2, R1</td>
<td>DADD R2, R2, R1</td>
<td>BEQZ R10, 220</td>
</tr>
<tr>
<td>208</td>
<td>BEQZ R10, 220</td>
<td>BEQZ R10, 224</td>
<td>DADD R2, R2, R1</td>
</tr>
<tr>
<td>212</td>
<td>DADD R3, R4, R5</td>
<td>NOP</td>
<td>DADD R3, R4, R5</td>
</tr>
<tr>
<td>216</td>
<td>DSUB R2, R6, R3</td>
<td>DADD R3, R4, R5</td>
<td>DSUB R2, R6, R3</td>
</tr>
<tr>
<td>220</td>
<td>SD R2, Z</td>
<td>DSUB R2, R6, R3</td>
<td>SD R2, Z</td>
</tr>
<tr>
<td>224</td>
<td>SD R2, Z</td>
<td>SD R2, Z</td>
<td></td>
</tr>
</tbody>
</table>
This technique allows us to wait until the end of the ID stage to decide whether to branch, and still not incur a pipeline delay:

<table>
<thead>
<tr>
<th>Clock #</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instr. A: BEQZ R10, X</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instr. B: DADD R2, R2, R1</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instr. X</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

We can generalize this technique to add \( n \) delay slots to cover \( n \) holes.

ISA is changed to mean “\( n \) instructions after any branch are always executed.”

What are some problems with this approach, in the long run?

The compiler typically can fill—

- 1 slot 75% of time
- 2 slots about 25% of time
- > 2 slots almost never

**Filling delay slots**

The compiler has several options in how to fill the delay slot.

1. Filling the slot from above the branch.

   - *Disadvantage*: Need a “safe” instruction from above the branch. What does “safe” mean?
   
   - *Advantage*: Delay slot instruction can always execute regardless of branch outcome (don’t ever need to quash it).
2. If you can’t fill slot(s) from above the branch, use instructions from either—

- target of branch (if frequently taken), or
- fall-through of branch (if frequently not taken).

*Example*: Fill from target (branch is frequently taken).

```
GOOD SCENARIO

ADD R4, R2, R3
BEQZ R1, X
NOP
BEQZ R1, X
ADD R4, R2, R3

BAD SCENARIO

ADD R1, R2, R3
BEQZ R1, X
NOP
BEQZ R1, X
ADD R1, R2, R3

BEQZ R1, Y
SUB R4, R2, R3

X: SUB R4, R2, R3
Y: ...

BEQZ R1, X
ADD R4, R2, R3

X: SUB R4, R2, R3
Y: ...
```

*Disadvantage*: Only works if delay slot instruction is *safe* to execute when

*Instruction enhancements*

1. Canceling branches: Provide two types of delayed branches.
   - Normal delayed branch
     - Delay slot instruction is always executed
• Canceling delayed branch
  – Slot filled from target (assumes a taken branch); instruction in delay slot is quashed if branch is not taken.
  – Slot filled from fall-through (assumes a not-taken branch); instruction in delay slot is quashed if branch is taken.

Having both types greatly enhances compiler’s ability to fill most slots.

• Use a normal branch when a safe instruction is available to fill slot (from above, target, or fall-through).
• Use a canceling branch when only unsafe candidates are available to fill slot (from target or fall-through).

2. Change encoding to add a likely bit.

<table>
<thead>
<tr>
<th>6</th>
<th>1</th>
<th>5</th>
<th>5</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>Opcode</td>
<td>rs1</td>
<td>rd</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

likely bit

• If compiler thinks branch is frequently taken,
  – Compiler sets likely bit ← 1.
  – Compiler fills delay slot from target.
  – Hardware knows to quash delay slot(s) if branch is not taken.

• If compiler thinks branch is frequently not taken,
  – Compiler sets likely bit ← 0.
  – Compiler fills delay slot from fall-through.
  – Hardware knows to quash delay slot(s) if branch taken.

If the likely bit is set capriciously, most delay slot instructions must be quashed.
Methods for setting likely bit

- The likely bit is essentially a static branch prediction.
  - Compiler makes a prediction that is fixed for that branch.
  - Likely bit = 1 means predict taken.
  - Likely bit = 0 means predict not taken.

- Static branch prediction methods (compiler branch prediction)
  - Heuristics
  - Profiling

Heuristics

- Heuristic #1: Branches are not taken
  - The majority of conditional branches are not taken.
  - True about 60% of the time.

- Heuristic #2: Backward branches are taken, forward branches are not taken (BTFNT).
  - Theme: Most backward branches are loops.
  - Since branches are PC relative, sign bit of offset = the prediction.
  - Jim Smith reports 70% accuracy for this scheme for scientific workloads.

- Heuristic #3: Ball/Larus style predictions—a set of rules to predict branches in special situations.
  - Detect loops and use BTFNT
  - Since error values returned by library functions are negative: ...and since errors are rare, Predict BLTZ, BLEZ, etc. not taken; Predict BGTZ, BGEZ, etc., taken.
– If a call is in the body of an if …then, predict the “then” branch as not taken.

Since most calls in if …then guard special-case code.

*Problem:* Works great for SPECint92 (from which it was designed).

My code might not work like that!

**Profiling**

Three steps:

1. Run the program (the “profiled run”).
2. Record the average preferred direction for each branch (taken or not taken).
3. Recompile to set the likely bits.

What if the program takes inputs (e.g., sort)?

– Collect a representative set of inputs somehow

*Problems*:

One prediction for

The profiled run is slow!

*Solution:* Use hardware to collect predictions.

Runs at normal speed: users don’t realize it’s profiling.