Modeling stalls

Let us develop a theory for how much the processor is slowed by branches.

\[
\text{Speedup} = \frac{\text{Avg. execution time unpipelined}}{\text{Avg. execution time pipelined}} = \frac{\text{CPI}_{\text{unpipe}} \times \text{CT}_{\text{unpipe}}}{\text{CPI}_{\text{pipe}} \times \text{CT}_{\text{pipe}}}
\]

\(\text{CPI}_{\text{unpipe}}\) is just \(n\), since in the absence of pipelining, the instruction takes \(n\) cycles.

\[\text{CPI}_{\text{pipe}} = \text{CPI}_{\text{no-stall}} + \text{stall cycles per instruction}\]

\[
\text{Speedup} = \frac{\text{CPI}_{\text{unpipe}} \times \text{CT}_{\text{unpipe}}}{\text{CPI}_{\text{pipe}} \times \text{CT}_{\text{pipe}}}
\]

\[
= \frac{\text{CPI}_{\text{unpipe}} (= n)}{1 + \text{stall cycles per instruction}}
\]

(This assumes that the two CTs are equal.)

Now let’s assume the two CPIs are equal (but the CTs are not).

\[
\text{Speedup} = \frac{\text{CPI}_{\text{unpipe}} \times \text{CT}_{\text{unpipe}}}{\text{CPI}_{\text{pipe}} \times \text{CT}_{\text{pipe}}}
\]

\[
= \frac{1}{1 + \text{stall cycles per instruction}} \times \frac{\text{CT}_{\text{unpipe}}}{\text{CT}_{\text{pipe}}}
\]

\[
= \frac{1}{1 + \text{stall cycles per instruction}} \times \frac{n \cdot \text{CT}_{\text{pipe}}}{\text{CT}_{\text{pipe}}}
\]

\[
= \frac{n}{1 + \text{stall cycles per instruction}}
\]
Let’s take an example.

Assume $n = 5$ (e.g., pipeline from last lecture).
20% of instructions are branches
60% of branches are taken

Penalties:

- Taken branches: 1 stall cycle
- Not-taken branches: 0 stall cycles

How many stall cycles per instruction are there on average?

$$\text{Stall cycles/instr.} = \frac{20\% \times 1 + 80\% \times 0}{100\%} = 20\%$$

$$\text{Speedup} = \frac{1}{20\%} = 5$$

**Control Hazards**

Suppose that we have the following sequence of instructions, where instruction A is `BLTZ R1,G`, and the branch target is not known until the MEM stage:

<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>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instr. A</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instr. B</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instr. C</td>
<td>IF</td>
<td>ID</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instr. D</td>
<td>IF</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instr. G</td>
<td></td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

What is the branch penalty here?

**Methods of handling control hazards**

However, there are various ways of dealing with control hazards. Let’s first assume we make no prediction as to whether the branch is going to be taken, but wait until we know.

One of the simplest is to *cancel the fetch and stall* waiting for the target instruction to become available:
What is the branch penalty?

Let us now suppose that the branch is taken.

Another strategy is to stall but not re-fetch the instruction after the branch.

What is the branch penalty?

Let us now suppose that the branch is taken.

A third strategy is to always predict the branch will not be taken.
Branch not taken

<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>A: <code>BLTZ R4, X</code></td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

What is the branch penalty?

Suppose the branch is taken:

Branch taken

<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>A: <code>BLTZ R4, X</code></td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>IF</td>
<td>ID</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>IF</td>
<td>ID</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>IF</td>
<td>ID</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>IF</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

What is the branch penalty?

In the above examples, we know *when* we are going to branch when we are in the ID stage (i.e., we know we have a branch instruction).

We don’t know *whether* or *where* we are going to branch until the MEM stage.

Suppose we have the pipeline we considered in the last lecture, where we knew all three (*when, whether, where*) in the ID stage. Then our branch penalty is 1 cycle:

<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>A: <code>BLTZ R4, X</code></td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>IF</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Issues in eliminating stalls**

*When:* Detect that an undecoded instruction is a branch.

*Where:* Predict where the branch will go (if taken).
**Whether:** (For conditional branches) Predict if it will branch or not, before execution.

Optimal: Try to determine all three in IF stage. It won’t work perfectly (*prediction*), but we can try our best.

**Issues with When and Where**

What does the IF stage know? Only the _________ of the instruction (the PC).

How can this information help us decide whether the branch might be taken?

So, we …

- keep a buffer (cache) of last known branch targets around
- Buffer is written to by WB stage.
- Traditional name for this is a *branch-target buffer* (BTB)

How do we index into the BTB? We could make the BTB fully associative, but …

Instead, let’s index it by the least-significant bits of the instruction address (not counting bits 1:0, of course).
So, if we have a 64-entry BTB, which address bits are used to index it?

**Predicting Where with returns**

**Problem:** A lot of branches are returns from procedures

Holding the last target address is a poor predictor.

**Solution:** Keep a hardware “stack” of return addresses.

- Push return address when a “call” is executed.
- Pop buffer on returns to get prediction.
  - Bottom of stack is filled with old value on a pop.
- Need approximately 4–8 entries for integer code.

**Issues with Whether**

Predicting conditional branches (and sometimes unconditional branches if needed before decode).

Two approaches:

- Hardware to supply prediction
- Software
  - Heuristics
  - Profiling

**Hardware branch prediction**

[H&P §3.4] Let’s add a bit to each BTB entry that says whether the branch was recently taken or not.

- Set prediction field = 1 if branch was *taken*, 0 if branch was not *taken*. 

• At IF, check “branch prediction buffer”:
  ♦ if prediction field = 1 then predict taken
  ♦ else predict not-taken

Of course, the bit may predict incorrectly. It may even have been put there by another instruction at an address that maps to the same BTB entry.

Problem: Some branches don’t do what they did last time!

Exercise: Consider a doubly nested loop

```c
for (i = 0; i < 5; i++)
  for (j = 0; j < 5; j++) {  … }
```

Supposing the branches at the end of these loops don’t conflict in the buffer, how often will they be predicted correctly? (Of course, the first time a branch is encountered, it will automatically be predicted “not taken.”)

In general, for branches used to form loops, a 1-bit predictor will mispredict at twice the rate that the branch is not taken.

We would like to have a predictor that at least matches the frequency at which the branches are taken.

In a 2-bit scheme, a prediction must miss twice before it is changed.

One complication is that a 2-bit counter updates the prediction bits more frequently than a one-bit counter. Why?

Smith’s n-bit counter predictor
Here is one way of using a two-bit counter (Smith’s $n$-bit counter predictor):

![Diagram of two-bit counter](image)

Unfortunately, it can do quite badly on certain types of branches:

<table>
<thead>
<tr>
<th>Prev. state</th>
<th>01 10 11 11 11 11 11 11 10 01 00 00 00 00 00 00 00 01</th>
</tr>
</thead>
<tbody>
<tr>
<td>New state</td>
<td>10 11 11 11 11 11 11 10 11 11 10 10 00 00 00 00 00 01</td>
</tr>
</tbody>
</table>

6 mispredictions out of 19 branch executions.

<table>
<thead>
<tr>
<th>Prev. state</th>
<th>01 10 01 10 01 10 01 10 01 10 01 10 01 10 01 10 01 10 01 10 01 10 01 10 01 10 01 10</th>
</tr>
</thead>
<tbody>
<tr>
<td>New state</td>
<td>10 01 10 01 10 01 10 01 10 01 10 01 10 01 10 01 10 01 10 01</td>
</tr>
</tbody>
</table>

19 mispredictions out of 19: the infamous “toggle branch.”
Another two-bit predictor

Another 2-bit predictor is shown on p. 198 of CAQA:

How would this predictor do on the “toggle branch”?

<table>
<thead>
<tr>
<th>Prev. state</th>
<th>T</th>
<th>N</th>
<th>T</th>
<th>N</th>
<th>T</th>
<th>N</th>
<th>T</th>
<th>N</th>
<th>T</th>
<th>N</th>
<th>T</th>
<th>N</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

New state

What kind of accuracy can we expect using a 2-bit predictor? Here are the results from SPEC89 benchmarks, with a branch-prediction buffer with 4096 entries.
Notice that the misprediction rate for the integer benchmarks (gcc, espresso, eqntott, and li) is substantially higher (11% avg.) than for the floating-point programs (4% avg.).

Unfortunately, these programs also have a higher branch frequency.

We can attack this problem in two ways:

- By increasing the size of the buffer
- By improving the accuracy of the prediction scheme.