1 Branch Prediction Overview

2 Software-Based Branch Prediction
   2.1. Branch Delay Slots
   2.2. Static Software Prediction
   2.3. Predication

3 Hardware-Based Branch Prediction
   3.1. Fixed Prediction
   3.2. One-Level Branch History Table
   3.3. Two-Level BHT For Temporal Correlation
   3.4. Two-Level BHT For Spatial Correlation
   3.5. Generalized Two-Level BHTs
   3.6. Other Predictors
# 1. Branch Prediction Overview

<table>
<thead>
<tr>
<th>branch</th>
<th>opA</th>
<th>opB</th>
<th>opC</th>
<th>opD</th>
<th>opE</th>
<th>opF</th>
<th>opG</th>
<th>opH</th>
</tr>
</thead>
</table>

Essential in modern processors to avoid mitigate branch delay latencies.

2 key types of prediction

1. Predict branch outcome
2. Predict branch | jump target address

Pipe Pipeline

\[ F \rightarrow D \rightarrow I \times M \rightarrow W \]

- Know branch outcome
- Know target for JR, JALR
- Know target for branches, J, JAL
2. Software-Based Branch Prediction

2.2. Branch Delay Slots

SW BPRED: Branch Delay Slots

<table>
<thead>
<tr>
<th>BR</th>
<th>OpA</th>
<th>OpB</th>
<th>Targ</th>
<th>F</th>
<th>D</th>
<th>X</th>
<th>M</th>
<th>W</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>F</td>
<td>D</td>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

No delay slots must squash fail through instr if taken branch.

Delay slots are useful work in slots. instr in delay slot always executed regardless of branch cond.

Exposes too much about microarch, less popular now

SW BPRED: STATIC SW PREDICTION

Extend ISA so that compiler can tell microarchitecture if branch is likely to be taken or not

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>F D X M W</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td></td>
<td></td>
<td>F</td>
<td>F</td>
</tr>
</tbody>
</table>

As soon as branch is decoded, use hint to request control flow

Only need to squash one instruction now

Taken/Not taken "hint" from compiler

What if hint is wrong?

<table>
<thead>
<tr>
<th>BR. +</th>
<th>OpA</th>
<th>Targ</th>
<th>Op D</th>
</tr>
</thead>
<tbody>
<tr>
<td>F D X M W</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
</tbody>
</table>

Check hint in X stage if wrong squash target + request control flow

Notice we do NOT need to re-fetch opA if we want one cycle to squash it!
Static sw prediction can work because we can exploit static structure in the program. For example, most backwards branches are taken so use hint to specify taken for these branches.

**Sw Based: Predication**

Not really “prediction”. Idea is to turn control flow into data flow completely eliminating control hazard.

\[
\text{MOVN rd, rs, r+} \quad \text{if (} R[r+] = 0 \text{) } R[rd] \leftarrow R[rs]
\]
\[
\text{MOVZ rd, rs, r+} \quad \text{if (} R[r+] = 0 \text{) } R[rd] \leftarrow R[rs]
\]

**Pseudo Code**

<table>
<thead>
<tr>
<th>if (a &lt; b)</th>
<th>if (a &lt; b)</th>
<th>if (a &lt; b)</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = a</td>
<td>slt r1, r2, r3</td>
<td>slt r1, r2, r3</td>
</tr>
<tr>
<td>else</td>
<td>beqz r1, r2, r1</td>
<td>move r4, r2</td>
</tr>
<tr>
<td>x = b</td>
<td></td>
<td>l2:</td>
</tr>
<tr>
<td></td>
<td>c1: move r4, r3</td>
<td>c2:</td>
</tr>
</tbody>
</table>

What if /then/ else has many instructions? What if one side of the branch has many more instructions than the other side?

**Full Predication**

- Almost all instructions can be executed under predicated
- Instruction becomes NOP if predicate is false

\[
\text{if (a < b)} \quad \text{slt r1, r2, r3} \quad \text{Special predication flag}
\]
\[
\text{op A} \quad \text{(r1): op A}
\]
\[
\text{op B} \quad \text{(r1): op B}
\]
\[
\text{else} \quad \text{(r1): op C}
\]
\[
\text{op D} \quad \text{(r1): op D}
\]
3. Hardware-Based Branch Prediction

3.1. Fixed Prediction

\[ HW \text{ Branch: Fixed Prediction } \]

1. **Always predict NOT-TAKEN**
   - What we have been assuming
   - Simple to implement
   - Know fall through PC in F
   - Poor accuracy, especially on very important
     Backwards branches in loops.

2. **Always predict TAKEN**
   - Difficult to implement because don't know target until?
   - Could still cause Branch Delay Latency in Data Path
   - Poor accuracy, especially on IF, then, else

3. **Backward Branch TAKEN, Forward Branch NOT TAKEN**
   - Similar to (2) in terms of Target
   - Better accuracy

\[
\text{Loop:} \\
\text{lw } r1, 0(r2) \\
\text{lw } r3, 0(r4) \\
\text{slt } r5, r1, r3 \\
\text{beq } r5, L1 \\
\text{move } r6, r1 \\
\text{J } L2 \\
\text{L1:} \\
\text{move } r6, r3 \\
\text{sw } r6, 0(r7) \\
\text{addw } r2, r2, 4 \\
\text{addw } r4, r4, 4 \\
\text{addw } r7, r1, 4 \\
\text{addw } r8, r8, -1 \\
\text{J } r8, \text{loop} \\
\text{on avg 90% taken}
\]

\[
\text{Forward Branch} \quad \text{on avg 50% taken}
\]
3. Hardware-Based Branch Prediction

3.2. One-Level Branch History Table

**HW Branch: Exploiting Temporal Correlation**

Exploit structure in program: the way a branch resolves may be a good indicator of the way it will resolve the next time it is executed (temporal correlation).

1-bit Saturating Counter

![Branch History Table Diagram]

<table>
<thead>
<tr>
<th>History</th>
<th>Prediction</th>
<th>Actual</th>
<th>Mispredict?</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>NT</td>
<td>T</td>
<td>Y</td>
</tr>
<tr>
<td>2</td>
<td>T</td>
<td>T</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>T</td>
<td>T</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>T</td>
<td>NT</td>
<td>Y</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>1</td>
<td>NT</td>
<td>T</td>
<td>Y</td>
</tr>
<tr>
<td>2</td>
<td>T</td>
<td>T</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>T</td>
<td>NT</td>
<td>Y</td>
</tr>
</tbody>
</table>

Always 2 mispredicts. Loops are very common exploiting temporal correlation works very well.

*Assume loop is executed multiple times*
3. Hardware-Based Branch Prediction

3.2. One-Level Branch History Table

<table>
<thead>
<tr>
<th>History</th>
<th>Predicted</th>
<th>Actual</th>
<th>Mispredicted?</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>NT</td>
<td>NT</td>
<td>T</td>
<td>Y</td>
<td>Strong NT</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>NT</td>
<td>Weak NT</td>
<td>Weak T</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>Strong T</td>
</tr>
</tbody>
</table>

What if start state is strong taken?

Only 1 mispredict per loop.
3. Hardware-Based Branch Prediction

3.2. One-Level Branch History Table

Other 2-bit FSM Branch Predictors

Jump directly to strong from weak

Biased towards predicting branches as taken

See Fig 5.8 in Sved + Lipasti for other alternatives
3. Hardware-Based Branch Prediction

3.2. One-Level Branch History Table

Implementing FSMs with Branch History Table

FSM state for specific PC

Two PC's can "alias" to the same entry in BHT. Similar to a cache conflict, except we can't really tell when a conflict happens.

Separate PC as a tag is too expensive

\[ \text{Aliasing} = \uparrow \text{mispredictions} \]

\[ \downarrow \text{Aliasing in larger but on can also \# associativity (still only use a few bits as "tag") in associative structure.} \]

Most BHTs and direct mapped because with only 2-bit per entry cheap to use large but

* 4k 2-bit entry but

* 80-90% prediction accuracy

How do we continue to improve prediction accuracy? Examine more complicated temporal correlations

More Complicated Temporal Correlation

Oftentimes, a branch exhibits more complicated patterns than just "always taken" or "always not taken." Could develop a more complicated FSM, but these patterns vary per branch.

void convolve ( int BS [ ], int BS [ ], int size ) 3
  for ( int i = 2; i < size-2; i++)
    for ( int j = 0; j < 5; j++)
      BS [ i - (2-j) ] = A [ i ] * COEFF [ j ]

Can we predict that every 5th dynamic instance of the backwards loop branch will be not taken?
3. Hardware-Based Branch Prediction  

3.3. Two-Level BHT For Temporal Correlation

---

**2-Level BHT to Exploit Temporal Correlation**

![Diagram of 2-Level BHT](image)

*PC* \[ \rightarrow \]

BHT \[ \downarrow \]

PHT \[ \downarrow \]

FSM \[ \downarrow \]

THAT?

**BusLT = Branch History Shift Register Table**

**PHT = Pattern History Table**

Each entry is a standard 6-bit FSM Prediction

---

When a Branch is taken or not taken we shift in either a one (taken) or zero (not taken) into the least significant bit of the corresponding **BusLT**.

The **BusLT** captures the temporal pattern for that Branch.

We use **BusLT** to index into the **PHT**. A **PHT** has an entry per Branch, but a **PHT** has an entry per Branch pattern.

The **PHT** says for a given pattern over the past n executions of a branch, should I take or not take the next execution of this Branch?

So in previous example patterns for inner loop backwards branch will be

```
0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 ...
```

With 2-level **PHT** and for $n \geq 4$ we can learn this pattern and perfectly predict this branch.

**Problem:** Allowing in the **BusLT**

**Solution:** Make **BusLT** larger or increase associativity
3. Hardware-Based Branch Prediction

3.3. Two-Level BHT For Temporal Correlation

Problem: Multiple branches with same history might need different predictions. In other words, aliasing in the BHT can reduce accuracy.

Solution: Add multiple BHTs, use bits from PC to choose which BHT to use.

Diagram:

Isomorphic—two different ways of drawing the same two-level structure.

PC

 FSM output logic

 1? 1??
HW Branch: Exploiting Spatial Correlation

The way one branch is resolved may be a good indicator of the way a later (different) branch will resolve.

\[
\begin{align*}
\text{if } (x < 7) & \quad \text{branch } \phi \\
\text{y++} & \\
\text{if } (x < 5) & \quad \text{branch } 1 \\
\text{z++} & \\
\end{align*}
\]

If branch \( \phi \) is taken (i.e., \( x \geq 7 \))
then branch 1 is always taken (i.e., \( x \) must be \( \leq 5 \))

So whether branch \( \phi \) is taken or not taken can be used to predict if we should take branch 1.

For above example, \( \text{BUSH} \) will capture history - so we will know even first branch is taken, and that value(s) of the BUSH will point to an entry in the PUT that predicts taken.

Only one \( \text{BUSH} \), so history of all branches merged into
A single history shift register.

\text{PUT-based 6-bit FSM predictor
As before, multiple paths can help avoid aliasing in ANT.
Generalized Two-Level BHTs

Combined approach to exploit from complex temporal correlation and spatial correlation.

Difference from discussion on complex temporal correlation is that we purposely choose a smaller \( m \) to cause aliasing in the BHT, since this aliasing allows us to capture spatial correlation.

(Choose higher order \( m \) bits.)

<table>
<thead>
<tr>
<th>( m = 0 )</th>
<th>( 0 &lt; k &lt; 30 )</th>
<th>( k = 30 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( M = M )</td>
<td>( M = M )</td>
<td>( M = M )</td>
</tr>
<tr>
<td>( G_{Ag} )</td>
<td>( G_{As} )</td>
<td>( G_{Ap} )</td>
</tr>
<tr>
<td>( 0 &lt; m &lt; 30 )</td>
<td>( P_{Ag} )</td>
<td>( P_{As} )</td>
</tr>
<tr>
<td>( m = 30 )</td>
<td>( S_{Ag} )</td>
<td>( S_{As} )</td>
</tr>
</tbody>
</table>

97% Accuracy
3. Hardware-Based Branch Prediction

3.5. Generalized Two-Level BHTs

Diagram:

- g-select
- Bus
- Output
- PC
- Output

Text:

Instead of concatenating all bits from PC with BSM, keeps avoid aliasing in the output more effectively.
TOURNAMENT PREDICTORS

Different predictors are better at predicting different types of branches:
- One-level 2-bit saturating counter - loops
- Two-levelAware - irregular code

Branch prediction selection table - predicts which branch predictor we should use
3. Hardware-Based Branch Prediction

3.6. Other Predictors

**HW BR pred: Predicting Target Address**

Even with best possible prediction of branch outcome, still need to wait for target address to be determined.

**Branch Target Buffer**

![Diagram of Branch Target Buffer]

- PC
- Predicted Target
- FSM
- New PC = predicted target PC if hit and predicted taken

Can put BTB in fetch stage:
- Predicting if PC points to a branch
- Predicting target of branch
- Predicting if branch is taken

Sometimes 6-0, if hit then assume predict taken
3. Hardware-Based Branch Prediction

3.6. Other Predictors

- Combine TB/TB and BTB

TB/TB is much more expensive than BTB, but TB/TB earlier in pipeline can accelerate JR.

- Combine TB/TB with BTB with few entries with BTB for many entries.

- Update BTB for JR

- Return Address Stack Predictor

TB/TB only works for JR function call returns if always call function from same place (not realistic)

Stack Predictor:
- Push target address on stack for JAL/JALR
- Pop off target address for JR to predict target

Move stack predictor into fetch and predict which PC's are JR.

Use tournament prediction to choose between TB/TB and Stack Predictor.