Material from Section 4.3

This set under construction.

Outline

• Branch Prediction Overview

• One-Level Predictor

• Two-Level Correlating Predictor

• Other topics to be added.

• Sample Problems
Branch and Target Prediction

Motivation

Branches occur frequently in code.

At best, one cycle of branch delay; more with dependencies.

Therefore, impact on CPI is large.

Techniques

- **Branch Prediction**
  Predict outcome of branch. (Taken or not taken.)

- **Branch Target Prediction**
  Predict branch or other CTI’s target address.

- **Branch Folding**
  Replace branch or other CTI with target instruction.
Branch Prediction

Methods Covered

- One-level predictor

- \((m, n)\) two-level correlating predictor or \((m, n)\) predictor for short.
Branch Prediction Idea

Idea: Predict based on assumption that patterns hold.

Example:

```assembly
LOOP:
lw    r1, 0(r2)   ! Read random number, either 0 or 1.
addi  r2, r2, #4
slt   r6, r2, r7
beqz  r1, SKIP
addi  r3, r3, #1
SKIP:
bneq  r6, LOOP   ! Loop executes 100 iterations.
nop
```

Second branch, `bneq`, taken 99 out of 100 executions.

Pattern for `bneq`: T T T ... NT T T T

First branch shows no pattern.
Prediction Accuracy

SPEC89 benchmarks on IBM POWER (predecessor to PowerPC).

FIGURE 4.14 Prediction accuracy of a 4096-entry two-bit prediction buffer for the SPEC89 benchmarks.
FIGURE 4.15  Prediction accuracy of a 4096-entry two-bit prediction buffer versus an infinite buffer for the SPEC89 benchmarks.
Branch Prediction Terminology

Outcome [of a branch instruction execution].
The outcome of the execution of a branch instruction.

$T$
A taken branch.

$NT$ or $N$
A branch that was not taken.

Prediction [made by branch prediction hardware].
The predicted outcome of a branch.

Misprediction
An incorrectly predicted outcome.

Prediction Accuracy [of a branch prediction scheme].
The number of correct predictions divided by the number of predictions.
**Speculative Execution**

The execution of instructions following a predicted branch.

**Misprediction Recovery**

Undoing the effect of speculatively executed instructions . . .

. . . and re-starting instruction fetch at the correct address.
One-Level Branch Predictor

Idea: maintain a *branch history* for each branch instruction.

Branch histories stored in a *branch history table*.

Branch history can be an arbitrary finite state machine or a *counter*.

Branch outcome causes a change in branch history.

Branch prediction based on state of branch history.
Branch History Counter

If a counter used, branch history incremented when branch taken... 
... and decremented when branch not taken.

Symbol $n$ denotes number of bits for branch history.

To save space and for performance reasons ... 
... branch history limited to a few bits, usually $n = 2$.

Branch history updated using a *saturating counter*.

A saturating counter is an arithmetic unit that can add or subtract one ... 
... in which $x + 1 \rightarrow x + 1$ for $x \in [0, 2^n - 2]$ ... 
... $x - 1 \rightarrow x - 1$ for $x \in [1, 2^n - 1]$ ... 
... $(2^n - 1) + 1 \rightarrow 2^n - 1$ ... 
... and $0 - 1 \rightarrow 0$.

For an $n$-bit counter, predict taken if counter $> 2^{n-1}$. 
One-Level Branch Predictor Hardware

Illustrated for Chapter-3 DLX implementation . . .
. . . even though prediction not very useful.

Branch Prediction Steps

* Predict.
  Read branch history, available in ID.

* Determine Branch Outcome
  Execute predicted branch in usual way.

* Recover (If necessary.)
  Undo effect of speculatively executing instructions, start fetching from correct path.

* Update Branch History
Branch History Table

Stores branch histories,

Implemented using a memory device.

Address (called index) is hash of branch address (PC).

For $2^m$-entry BHT, hash is $m$ lowest bits of branch PC skipping alignment.

\[
\begin{array}{ccc|c}
\text{BHT Addr} & \text{Align.} \\
\hline
\text{Branch address:} & & & \\
31 & m+2 & m+1 & 2 & 1 & 0 \\
\end{array}
\]

Data input and output of BHT is branch history.
Outcomes for individual branches, categorized by pattern, sorted by frequency.

Branches running \TeX text formatter compiled for SPARC (Solaris).

<table>
<thead>
<tr>
<th>% Patterns</th>
<th>Patterns</th>
<th># Branches</th>
<th>gshre</th>
<th>local</th>
<th>corr</th>
<th>Local History</th>
</tr>
</thead>
<tbody>
<tr>
<td>0: fe7f</td>
<td>0.0004</td>
<td>1397</td>
<td>0.912</td>
<td>0.916</td>
<td>0.896</td>
<td>TTTTTTTNNTTTTTTTT 0</td>
</tr>
<tr>
<td>1: ff3f</td>
<td>0.0004</td>
<td>1323</td>
<td>0.924</td>
<td>0.909</td>
<td>0.900</td>
<td>TTTTTTNNTTTTTTTTTT 0</td>
</tr>
<tr>
<td>2: fcff</td>
<td>0.0004</td>
<td>1317</td>
<td>0.949</td>
<td>0.939</td>
<td>0.948</td>
<td>TTTTTTTTTNNTTTTTTTT 0</td>
</tr>
<tr>
<td>3: ff9f</td>
<td>0.0003</td>
<td>1245</td>
<td>0.910</td>
<td>0.905</td>
<td>0.898</td>
<td>TTTTTNTTTTTTTTTTTTTT 0</td>
</tr>
<tr>
<td>4: f9ff</td>
<td>0.0003</td>
<td>1235</td>
<td>0.955</td>
<td>0.950</td>
<td>0.955</td>
<td>TTTTTTTTTTTNNTTTTTTTT 0</td>
</tr>
<tr>
<td>5: ffef</td>
<td>0.0003</td>
<td>1188</td>
<td>0.926</td>
<td>0.921</td>
<td>0.923</td>
<td>TTTTNNTTTTTTTTTTTTTT 0</td>
</tr>
<tr>
<td>6: 60</td>
<td>0.0003</td>
<td>1163</td>
<td>0.873</td>
<td>0.829</td>
<td>0.854</td>
<td>NNNNNTNNTNNNNNNNNN 0</td>
</tr>
<tr>
<td>7: 180</td>
<td>0.0003</td>
<td>1159</td>
<td>0.955</td>
<td>0.914</td>
<td>0.926</td>
<td>NNNNNNNTNNTNNNNNNNN 0</td>
</tr>
<tr>
<td>8: 300</td>
<td>0.0003</td>
<td>1158</td>
<td>0.949</td>
<td>0.926</td>
<td>0.934</td>
<td>NNNNNNNNTNNTNNNNNNN 0</td>
</tr>
<tr>
<td>9: c0</td>
<td>0.0003</td>
<td>1155</td>
<td>0.944</td>
<td>0.917</td>
<td>0.926</td>
<td>NNNNNNNTNNTNNNNNNNN 0</td>
</tr>
</tbody>
</table>
Short Loop, pat 124, br 137681, 0.8908 0.9055 0.7441 (0.03700)

% Patterns # Branches gshre local corr Local History
0: 5555 0.0040 14753 0.987 0.981 0.912 TNTNTNTNTNTNTNTN 1
1: aaaa 0.0040 14730 0.859 0.978 0.461 NTNTNTNTNTNTNTNT 1
2: 9249 0.0022 8062 0.997 0.992 0.988 TNNTNNTNTNTNTNTN 1
3: 4924 0.0022 8055 0.997 0.998 0.998 NNTNNTNNTNTNTNTN 1
4: 2492 0.0022 8047 0.993 0.991 0.009 NTTNTNTNTNTNTNTN 1
5: db6d 0.0013 4864 0.713 0.915 0.065 TNTNNTNTNTNTNTNTT 1
6: b6db 0.0013 4713 0.862 0.903 0.926 TTTTTTTTTTTTTTTT 1
7: 6db6 0.0012 4640 0.991 0.978 0.970 TTTNTNTNTNTNTNTTN 1
8: bbbb 0.0008 3061 0.896 0.936 0.949 TTTTTTTTTTTTTTTT 1

Long Loop?, pat 32, br 185795, 0.9170 0.9052 0.9096 (0.04993)

0: fffe 0.0025 9204 0.902 0.930 0.913 NTTTTTTTTTTTTTTT 2
1: 8000 0.0025 9198 0.654 0.700 0.705 NNNNNNNNNNNNNNT 2
2: 7fff 0.0022 8052 0.890 0.817 0.818 TTTTTTTTTTTTTTN 2
3: ffbf 0.0018 6800 0.933 0.908 0.920 TTTTTTNTTTTTTTTN 2
4: feff 0.0018 6782 0.946 0.938 0.942 TTTTTTTNTTTTTTTT 2
5: ff7f 0.0018 6778 0.949 0.946 0.950 TTTTTTTTTTTTTTT 2
6: fdff 0.0018 6738 0.947 0.941 0.946 TTTTTTTTTTTTTTTT 2
7: 1 0.0018 6690 0.955 0.945 0.942 TNNNNNNNNNNNNNN 2
8: fffd 0.0018 6667 0.968 0.966 0.967 TTTTTTTTTTTTTTTT 2
### Phase Change, pat 26, br 48190, 0.8453 0.9040 0.8470 (0.01295)

<table>
<thead>
<tr>
<th>% Patterns</th>
<th># Branches</th>
<th>gshre</th>
<th>local</th>
<th>corr</th>
<th>Local History</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>c000</td>
<td>0.0012</td>
<td>4554</td>
<td>0.653</td>
<td>0.777 0.680</td>
</tr>
<tr>
<td>1</td>
<td>e000</td>
<td>0.0009</td>
<td>3420</td>
<td>0.714</td>
<td>0.859 0.758</td>
</tr>
<tr>
<td>2</td>
<td>f000</td>
<td>0.0008</td>
<td>2942</td>
<td>0.756</td>
<td>0.888 0.788</td>
</tr>
<tr>
<td>3</td>
<td>ffc00</td>
<td>0.0008</td>
<td>2878</td>
<td>0.908</td>
<td>0.960 0.959</td>
</tr>
<tr>
<td>4</td>
<td>f800</td>
<td>0.0007</td>
<td>2642</td>
<td>0.786</td>
<td>0.917 0.827</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>0.0007</td>
<td>2572</td>
<td>0.968</td>
<td>0.952 0.951</td>
</tr>
<tr>
<td>6</td>
<td>fc00</td>
<td>0.0007</td>
<td>2435</td>
<td>0.815</td>
<td>0.933 0.854</td>
</tr>
<tr>
<td>7</td>
<td>fe00</td>
<td>0.0006</td>
<td>2225</td>
<td>0.836</td>
<td>0.936 0.876</td>
</tr>
<tr>
<td>8</td>
<td>ff00</td>
<td>0.0006</td>
<td>2140</td>
<td>0.856</td>
<td>0.947 0.931</td>
</tr>
<tr>
<td>9</td>
<td>ff80</td>
<td>0.0006</td>
<td>2061</td>
<td>0.854</td>
<td>0.941 0.934</td>
</tr>
</tbody>
</table>

### One Way, pat 2, br 2617433, 0.9917 0.9934 0.9897 (0.70337)

<table>
<thead>
<tr>
<th>% Patterns</th>
<th># Branches</th>
<th>gshre</th>
<th>local</th>
<th>corr</th>
<th>Local History</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>ffff</td>
<td>0.5151</td>
<td>1916950</td>
<td>0.993</td>
<td>0.996 0.993</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0.1882</td>
<td>700483</td>
<td>0.988</td>
<td>0.986 0.982</td>
</tr>
</tbody>
</table>
Steps in BHT Use (refers to example.)

Register \( r1 \) not available until cycle ten\(^1\).

When branch in ID, read BHT and make prediction. (Cycle 1)

(Optional) Backup (checkpoint) register map (if present).

Execute branch in usual way and check prediction. (Cycle 10.)

If prediction correct, update BHT when branch commits (Cycle 11.).

If prediction wrong, start recovery process (does not occur here).

\[ ! \text{Predict not taken, not taken.} \]

Cycle: \[ 0 \quad 1 \quad 2 \quad 3 \quad 10 \quad 11 \quad 12 \quad 13 \]

\[ \text{bneq } r1, \text{TARGET} \quad \text{IF } \text{ID} \quad 0:RS \quad 0:RS \quad \ldots \quad 0:B \quad 0:WC \]

\[ \text{xor } r2, r3, r4 \quad \text{IF } \text{ID} \quad 5:EX \quad 5:WB \quad 5:C \]

\[ 6:C \]

\[ \ldots \]

\[ \text{TARGET:} \]

\[ \text{and } r5, r6, r7 \]

\[ \]

\(^1\) Perhaps due to a cache miss, or maybe it depended on a long-latency floating-point operation, the reason is not important
BHT use when branch taken, correctly predicted.

Register r1 not available until cycle 10.

When branch in ID, compute target, read BHT and make prediction. (Cycle 1).

Execute branch in usual way and check prediction. (Cycle 10.)

Commit branch after div. (Cycle 23).

! Predict taken, taken.

<table>
<thead>
<tr>
<th>Cycle:</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>10</th>
<th>11</th>
<th>...</th>
<th>21</th>
<th>22</th>
<th>23</th>
</tr>
</thead>
<tbody>
<tr>
<td>div f0,f2, f4</td>
<td>ID</td>
<td>DIV</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>DIV WC</td>
</tr>
<tr>
<td>bneq r1, TARGET</td>
<td>IF</td>
<td>ID</td>
<td>0:RS</td>
<td>0:RS</td>
<td>...</td>
<td>0:B</td>
<td>0:WB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>xor r2, r3, r4</td>
<td>IFx</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

... 

TARGET:

and r5, r6, r7 | IF... |    |    |    |    |    |     |    |    |    | C
BHT use when branch taken, incorrectly predicted, register map *not* backed up.

Register \( r1 \) not available until cycle 10.

When branch in ID, compute target, read BHT and make prediction. (Cycle 1).

Cycle 10: oops, misprediction. Because register map not backed up, recovery must wait until commit.

Cycle 23: Start recovery: Squash instructions in reorder buffer, start fetching correct path.
BHT use when branch taken, incorrectly predicted, register map backed up.

Register \( r1 \) not available until cycle 10.

When branch in ID, backup (checkpoint) register map, compute target, read BHT and make prediction. (Cycle 1).

Cycle 10: ooops, misprediction. Squash reorder buffer past branch, switch to backed up register map, start fetching correct path.

Cycle 23: Branch commits.

! Predict not taken, taken. Register map backed up.

<table>
<thead>
<tr>
<th>Cycle:</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>10</th>
<th>11</th>
<th>...</th>
<th>21</th>
<th>22</th>
<th>23</th>
</tr>
</thead>
<tbody>
<tr>
<td>div</td>
<td>f0, f2, f4</td>
<td>ID</td>
<td>DIV</td>
<td></td>
<td></td>
<td></td>
<td>DIV</td>
<td>WC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>bneq</td>
<td>r1, TARGET</td>
<td>IF</td>
<td>ID</td>
<td>0:RS</td>
<td>0:RS</td>
<td>...</td>
<td>0:B</td>
<td>0:WB</td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>xor</td>
<td>r2, r3, r4</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

... 
TARGET:
and r5, r6, r7

IF ....
(m, n) Two-Level Correlating Predictor

Idea: Base branch decision on ...
   ... the identity of the branch instruction (as in the one-level scheme) ...
   ... and the most recent branch outcomes (global history).

*Global History*

The outcome of the most recent branches.

In an (m, n) predictor, interested in m most-recent branches.
Global History Example

! Loop always iterates 4 times.
! Branch below never taken.
    bneq r2, SKIP   N
    add f0, f0, f2
SKIP:
    addi r1, r0, #4
LOOP:
    multd f0, f0, f2
    subi r1, r1, #1
    bneq r1, LOOP   T T T T N ... T T T T N ...
! Cycle    10  20  30  40  50  110  120  130  140  150
!
! Global History (m=4), X: depends on earlier branches.
! 10  XXXN  Human would predict taken.
! 20  XXNT  Human would predict taken.
! 30  XNTT  Human would predict taken.
! 40  NTTT  Human would predict not taken.
! 50  TTTN
Two methods of generating address for BHT:

**gselect**: Concatenate global history with branch address.

**gshare**: Exclusive-or global history with branch address.

gselect is easier to understand, but gshare uses BHT more efficiently.
Global History and Dynamic Scheduling

Global history must be accurate.

Why that’s a problem:

! First branch: Predict not taken, taken. Register map backed up.
Cycle: 0 1 2 3 10 11 12 13 ... 21 22 23
  div f0, f2, f4  ID DIV
  bneq r1, TARGET IF ID 0:RS 0:RS ... 0:B 0:WB C
  beqz r2, SKIP IF ID 1:B ...
  xor r2, r3, r4 IF ID EX ...

TARGET:
  and r5, r6, r7  IF ID EX ...
  beqz r4, LINE1 IF ID ...
Cycle: 0 1 2 3 10 11 12 13 ... 21 22 23

Cycle 2: beqz should see global history with bneq not taken.

Global history includes assumption that bneq not taken.
! First branch: Predict not taken, taken. Register map backed up.

Cycle:  0  1  2  3  10  11  12  13 ... 21  22  23
   div f0, f2, f4  ID  DIV  ID DIV  WC
  bneq r1, TARGET  IF  ID  0:RS  0:RS ... 0:B  0:WB  C
  beqz r2, SKIP   IF  ID  1:B  ...
   xor r2, r3, r4  IF  ID  EX  ...

TARGET:
   and r5, r6, r7   IF  ID  EX  ...
  beqz r4, LINE1   IF  ID ...
Cycle:  0  1  2  3  10  11  12  13 ... 21  22  23

Cycle 3: Now global history includes assumption that \texttt{bneq} and first \texttt{beqz} not taken.

Cycle 11: Ooops, \texttt{bneq} misprediction discovered.

Global history has two incorrect assumptions ...

... unless they’re fixed prediction for second \texttt{beqz} won’t be accurate.

Cycle 12: \texttt{beqz} should see global history with \texttt{bneq} taken.
Global History and Dynamic Execution

Global History in Two-Level Predictor with Dynamic Execution

Global history backed up \textit{(checkpointed)} at each branch.

\textit{Predicted} outcome shifted into global history.

If misprediction discovered, global history restored from backup \ldots
\ldots just as the register map can be.
Target Prediction
Predicting the outcome and target of a branch.

Branch Target Buffer
A table indexed by branch address holding a predicted target address.

Target Prediction

Put BTB in IF stage.

Use PC to read an entry from BTB.

If valid entry found, replace PC with predicted target.

With target correctly predicted, zero branch delay.
Target Prediction Example

Static scheduled system (for clarity).

<table>
<thead>
<tr>
<th>Cycle:</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
</tr>
</thead>
<tbody>
<tr>
<td>bneq</td>
<td>r1</td>
<td>TARGET</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
</tr>
<tr>
<td>xor</td>
<td>r2</td>
<td>r3</td>
<td>r4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TARGET:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>and</td>
<td>r5</td>
<td>r6</td>
<td>r7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Cycle 0

BTB lookup and prediction. Predict taken.

Target from BTB will be clocked into PC.
Target Prediction Example, continued.

Static scheduled system (for clarity).

<table>
<thead>
<tr>
<th>Cycle:</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
</tr>
</thead>
<tbody>
<tr>
<td>bneq r1, TARGET</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>xor r2, r3, r4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

TARGET:

<table>
<thead>
<tr>
<th>and r5, r6, r7</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
<th>IF</th>
<th>X</th>
</tr>
</thead>
</table>

Cycle 1

Start fetching predicted target.

Execute branch instruction (in ID).

Check predicted outcome and predicted target.

Correct predictions, continue execution.
Target Prediction Example, continued.

<table>
<thead>
<tr>
<th>Cycle:</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
</tr>
</thead>
<tbody>
<tr>
<td>bneq r1, TARGET</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>xor r2, r3, r4</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
</tbody>
</table>

TARGET:
and r5, r6, r7 | IF | ID | EX | MEM | WB | IF | X |

Cycle 10

BTB lookup and prediction. Predict taken.

Target from BTB will be clocked into PC.

Cycle 11

Start fetching predicted target.

Execute branch instruction (in ID).

Ooops, incorrect outcome prediction …
… replace target with nop …
… and clock correct target into PC.
Target Prediction for Register-Indirect CTI

What BTB predicts for branch instructions:

That instruction will be a CTI.

If CTI is a branch, that branch is taken.

CTI target.

For branches and non-indirect jumps (j, jal)

... predicting target is easy, since target always same.

\[ \text{bneq } r1, \text{ LOOP} \quad \text{! Target always PC + 4 + 4 * LOOP} \]
\[ \text{j } \text{LINEJ} \quad \text{! Target always PC + 4 + 4 * LINEJ} \]

For register-indirect jumps (jr, jalr)

... prediction depends on predictable behavior.

\[ \text{jr } r1 \quad \text{! Target is in r1. Can be different each time.} \]
\[ \text{jalr } r1 \quad \text{! Target is in r1. Can be different each time.} \]
Target Prediction

Two Methods

Keep a stack of (what appear to be) return addresses. Used for procedure return instructions.

Predict last target. Used for all other instructions.

Predict Last Target

Used for everything except return instructions.

Last time instruction executed target address stored in BTB.

If entry found and predicted taken (for a branch), last target address used.

Effectiveness:

Perfect for non-indirect jumps and branches (if taken).

Reasonably effective on indirect branches.
Predict Return Address

Used for return instruction. (An instruction used for a procedure return, which may not have the mnemonic `return`).

Hardware keeps a stack of return addresses.

BTB stores whether instruction is a return.

When a call instruction encountered push return address on stack.

When BTB identifies instruction as a return target address is popped off stack.

Effectiveness depends on whether call and return instructions can be identified.
Consider code for a C `switch` statement:

```c
! Possible code for a switch statement.
! switch( r2 ) { case 0: foo(); break; case 1: bar(); break; ... }
! Set r1 to base of switch address table.
  lhi  r1, #0x1234
  ori  r1, r1, #0x5670
! Multiply switch index by stride of table (4 bytes per address).
  slli r3, r2, #2
! Get address of case code address.
  add  r1, r1, r3
! Get case code address.
  lw   r4, 0(r1)
! Jump to case code.
  jr   r4
```

If `r2` rarely changes, `jr` predictable.
Possible BTB Contents

Target address.

History information (instead of in BHT).

Tag, to detect collisions.