This Set

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 Prediction

Methods Covered

- One-level predictor, a.k.a. *bimodal*.

Two-Level Predictors

- Global History, a.k.a. *GAg*.
- Local History, a.k.a. *PAG*.

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 Idea

Idea: Predict using past behavior.

Example:

```
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**

SPEC9 benchmarks on IBM POWER (predecessor to PowerPC).

![Graph showing frequency of mispredictions for SPEC9 benchmarks](image1)

**FIGURE 4.14** Prediction accuracy of a 4096-entry two-bit prediction buffer for the SPEC9 benchmarks.

**FIGURE 4.15** Prediction accuracy of a 4096-entry two-bit prediction buffer versus an infinite buffer for the SPEC9 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 is 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 History:
Information about past behavior of the branch.

Branch histories stored in a branch history table.

Often, branch history is sort of number of times branch taken... minus number of times not taken.

Other types of history possible.
Branch history read to make a prediction.
Branch history updated when branch outcome known.

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 → x + 1 for x ∈ [0, 2^n - 2]...
... x - 1 → x - 1 for x ∈ [1, 2^n - 1]...
... (2^n - 1) + 1 → 2^n...
... and 0 - 1 → 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
1: Predict.
   Read branch history, available in ID.

2: Determine Branch Outcome
   Execute predicted branch in usual way.

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

4: 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.

<table>
<thead>
<tr>
<th>BHT Addr</th>
<th>Align.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Branch address:</td>
<td>0</td>
</tr>
<tr>
<td>31</td>
<td>m+2</td>
</tr>
</tbody>
</table>

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).

Arbitrary, pat 60288, br732164, 0.7743 0.7170 0.7199 (0.19675)

% Patterns # Branches gshre local corr Local History
0: fe7f 0.0004 1397 0.912 0.916 0.896 TTTTTTTNNTTTTTTT 0
1: ff3f 0.0004 1323 0.924 0.909 0.900 TTTTTTNNTTTTTTTT 0
2: fcff 0.0003 1317 0.949 0.939 0.948 TTTTTTTTNNTTTTTT 0
3: ff9f 0.0003 1245 0.910 0.905 0.898 TTTTTNNTTTTTTTTT 0
4: f9ff 0.0003 1235 0.955 0.950 0.955 TTTTTTTTTNNTTTTT 0
5: ffcf 0.0003 1188 0.926 0.921 0.923 TTTTNNTTTTTTTTTT 0
6: 60 0.0003 1163 0.873 0.829 0.854 NNNNNTTNNNNNNNNN 0
7: 180 0.0003 1159 0.955 0.942 0.926 NNNNNNTTNNTNNTNTN 0
8: f000 0.0003 1155 0.944 0.917 0.926 NNNNNNTTNNNNNNNNN 0
9: c0 0.0003 1155 0.944 0.917 0.926 NNNNNNTTNNNNNNNNN 0

Short Loop, pat 124, br137681, 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 11
1: aaaa 0.0040 14730 0.859 0.978 0.461 NTNTNTNTNTNTNTNT 12
2: 9249 0.0022 8062 0.997 0.992 0.988 TNTNTNTNTNTNTNTN 13
3: 4924 0.0022 8055 0.997 0.998 0.998 TNTNTNTNTNTNTNTN 14
4: 2492 0.0022 8047 0.993 0.991 0.009 TNTNTNTNTNTNTNTN 15
5: db6d 0.0013 4864 0.713 0.915 0.065 TNTNTNTNTNTNTNTN 16
6: b6db 0.0013 4713 0.862 0.903 0.926 TNTNTNTNTNTNTNTN 17
7: 6db6 0.0012 4640 0.991 0.978 0.970 TNTNTNTNTNTNTNTN 18
8: bbbb 0.0008 3061 0.896 0.936 0.949 TNTNTNTNTNTNTNTN 19

Long Loop?, pat 32, br185795, 0.9170 0.9052 0.9096 (0.04993)

% Patterns # Branches gshre local corr Local History
0: fffe 0.0025 9204 0.902 0.930 0.913 NTTTTTTTTTTTTTTT 21
1: 8000 0.0025 9198 0.654 0.700 0.705 NNNNNNNNNNNNNNNT 22
2: 7fff 0.0022 8052 0.890 0.817 0.818 TNTNTNTNTNTNTNTN 23
3: fffe 0.0018 6782 0.946 0.942 0.942 TTTTTTTTTTTTTTTT 24
4: ff7f 0.0018 6778 0.949 0.946 0.950 TTTTTTTTTTTTTTTT 25
5: fdff 0.0018 6738 0.947 0.941 0.946 TTTTTTTTTTTTTTTT 26
6: ffd 0.0018 6690 0.955 0.945 0.942 TNTNTNTNTNTNTNTN 27
7: fffd 0.0018 6667 0.968 0.966 0.967 TNTTTTTTTTTTTTTT 28

Phase Change, pat 26, br48190, 0.8453 0.9040 0.8470 (0.01295)

% Patterns # Branches gshre local corr Local History
0: c000 0.0012 4554 0.653 0.777 0.680 NNNNNNNNNNNNNNTT 31
1: e000 0.0009 3420 0.714 0.859 0.758 NNNNNNNNNNNNNNNN 32
2: f000 0.0008 2942 0.756 0.888 0.788 NNNNNNNNNNNNNNNN 33
3: ffcc 0.0008 2878 0.908 0.960 0.959 NTTTTTTTTTTTTTTT 34
4: ff00 0.0007 2642 0.786 0.917 0.827 NNNNNNNNNNNNNNNN 35
5: fc00 0.0007 2572 0.968 0.952 0.951 NTTTTTTTTTTTTTTT 36
6: fc00 0.0007 2435 0.815 0.933 0.854 NNNNNNNNNNNNNNNN 37
7: fe00 0.0006 2225 0.836 0.936 0.876 NNNNNNNNNNNNNNNN 38
8: ff00 0.0006 2140 0.856 0.947 0.931 NNNNNNNNNNNNNNNN 39
9: ff80 0.0006 2061 0.854 0.941 0.934 NNNNNNNNNNNNNNNN 40

One Way, pat 2, br2617433, 0.9917 0.9934 0.9897 (0.70337)

% Patterns # Branches gshre local corr Local History
0: ffff 0.5151 1916950 0.993 0.996 0.993 TTTTTTTTTTTTTTTT 4
1: 0 0.1882 700483 0.988 0.986 0.993 TTTTTTTTTTTTTTTT 4

Two-Level Correlating Predictors

Idea: Base branch decision on ...

... the address of the branch instruction (as in the one-level scheme) ...

... and the most recent branch outcomes.

History:
The outcome (taken or not taken) of the most recent branches. Usually stored as a bit vector with 1 indicating taken.

Pattern History Table (PHT):
Memory for 2-bit counters, indexed (addressed) by some combination of history and the branch instruction address.
Some Types of Two-Level Predictors

**Global**, a.k.a. **GAg**.

History is global (same for all branches), stored in a *global history register (GHR)*. PHT indexed using history only.

**gshare**

History is global (same for all branches), stored in a *global history register (GHR)*. PHT indexed using history exclusive-ored with branch address.

**gselect**

History is global (same for all branches), stored in a *global history register (GHR)*. PHT indexed using history concatenated with branch address.

**Local**, a.k.a. **PAg**.

History is local, BHT stores history for each branch. PHT indexed using history only.

---

Global History Example

```
! Loop always iterates 4 times.
! Branch below never taken.
bneq r2, SKIP N N
addi r1, r0, #4
loop:
muli f0, f0, f2
subi r1, r1, #1
bneq r1, LOOP T T T N ...
```

---

Steps in BHT Use on a Dynamically Scheduled Proc.

Register `r1` not available until cycle ten.

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

Cycle 1: (Optional) Backup (checkpoint) register map (if present).

Cycle 10: Execute branch in usual way and check prediction.

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

Cycle 11: If pred. wrong, start recovery process (does not occur here).

---

1. Predict not taken, not taken.

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.

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

Cycle 10: Execute branch in usual way.

Cycle 11: Check outcome. Correctly predicted.

Cycle 23: Commit branch after \( \text{div} \).

\[ \text{BHT use when branch taken, incorrectly predicted, register map not backed up.} \]

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

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

Cycle 10: Compute branch condition.

Cycle 11: Misprediction “discovered.” Because register map not backed up, recovery must wait until commit.

Cycle 23: Start recovery: Squash instructions in reorder buffer, start fetching correct path.

\[ \text{BHT use when branch taken, incorrectly predicted, register map backed up.} \]

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

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

Cycle 10: Compute branch condition.

Cycle 11: Misprediction discovered. Squash reorder buffer past branch, switch to backed up register map, start fetching correct path.

Cycle 23: Branch commits.

\[ \text{Global History and Dynamic Scheduling} \]

Global history must be accurate.

Why that’s a problem:

\[ \text{Global history includes assumption that } \text{bneq} \text{ not taken.} \]
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 Q  DIV  
- bneq r1, TARGET IF ID Q  ...  B  WB  
- beqz r2, SKIP IF ID Q  B  ...
- xor r2, r3, r4 IF ID Q  EX...

TARGET:
- and r5, r6, r7 IF ID Q  ...
- 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 bneq and first beqz not taken.

Cycle 11: Ooops, bneq misprediction discovered.

Global history has two incorrect assumptions ... unless they’re fixed prediction for second beqz won’t be accurate.

Cycle 12: beqz should see global history with bneq taken.

Target Prediction and Folding

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.

Global History and Dynamic Execution

Global History in Two-Level Predictor with Dynamic Execution

Global history backed up (checkpointed) at each branch.

Predicted outcome shifted into global history.

If misprediction discovered, global history restored from backup ... just as the register map can be.

Target Prediction Example

Static scheduled system (for clarity).

Target:
- and r5, r6, r7 IF ID EX MEM WB IF X

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>bneq r1, TARGET</th>
<th>and r5, r6, r7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>IF ID EX MEM WB</td>
<td>IF X</td>
</tr>
<tr>
<td>1</td>
<td>IF ID EX</td>
<td>IF X</td>
</tr>
<tr>
<td>2</td>
<td>IF ID EX</td>
<td>IF X</td>
</tr>
<tr>
<td>3</td>
<td>IF ID EX</td>
<td>IF X</td>
</tr>
<tr>
<td>4</td>
<td>IF ID EX</td>
<td>IF X</td>
</tr>
<tr>
<td>10</td>
<td>IF ID EX MEM WB</td>
<td>IF X</td>
</tr>
<tr>
<td>11</td>
<td>IF ID EX MEM WB</td>
<td>IF X</td>
</tr>
<tr>
<td>12</td>
<td>IF ID EX MEM WB</td>
<td>IF X</td>
</tr>
<tr>
<td>13</td>
<td>IF ID EX MEM WB</td>
<td>IF X</td>
</tr>
<tr>
<td>14</td>
<td>IF ID EX MEM WB</td>
<td>IF X</td>
</tr>
</tbody>
</table>

TARGET:

Cycle 1

Start fetching predicted target.

Execute branch instruction (in ID).

Check predicted outcome and predicted target.

Correct predictions, continue execution.

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.

$bneq$ $r1$, LOOP  ! Target always PC + 4 + 4 * LOOP

$j$ LINEJ  ! Target always PC + 4 + 4 * LINEJ

For register-indirect jumps ($jr$, $jalr$)

... prediction depends on predictable behavior.

$jr$ $r1$  ! Target is in r1. Can be different each time.

$jalr$ $r1$  ! Target is in r1. Can be different each time.

Behavior of Register-Indirect Jumps

Predictability depends on how jumps used.

Major Uses

- Procedure Passed as Parameter
  
  For example, function passed to the C library’s $qsort$.
  
  These rarely change so target is predictable.

- Case Statements
  
  These change, and so prediction more difficult.
Indirect Jump Target Prediction

Separate techniques used for procedure returns and other indirect jumps.

Return Address Prediction

Keep a stack of (what appear to be) return addresses.

Other Indirect Jumps Prediction

Predict last target.

Use global branch history to index BTB.

Predict Return Address

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

Operation

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

Works fairly well.

Can be confused when returns skipped (as with long jumps).

Costly to implement precisely with dynamic scheduling.

Predict Last Target

Can be 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.

Use Global History

Can be used for everything except return instructions.

Much more effective on than last target.

Target Prediction Example

Consider code for 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.
li 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 (replaces BHT).
- Tag, to detect collisions.