

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

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.

Methods Covered

- *One-level predictor*
- (m, n) *two-level correlating predictor* or (m, n) *predictor* for short.

Idea: Predict based on assumption that patterns hold.

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.

SPEC89 benchmarks on IBM POWER (predecessor to PowerPC).

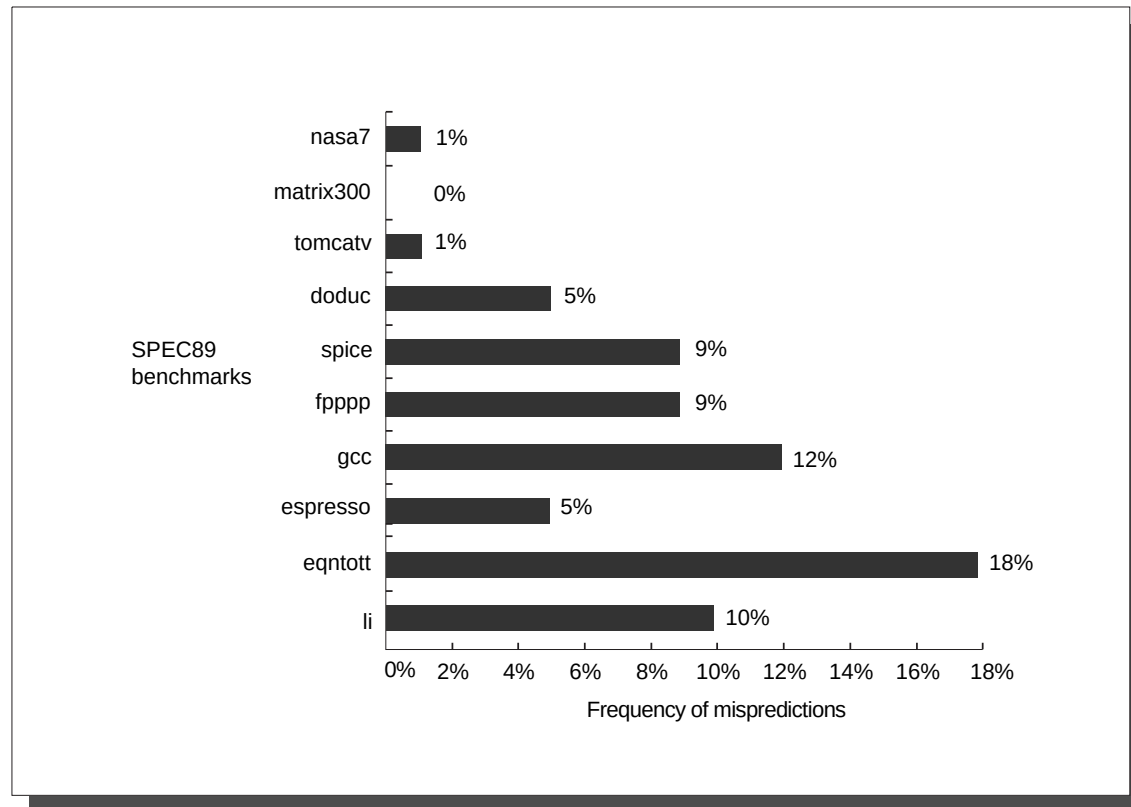


FIGURE 4.14 Prediction accuracy of a 4096-entry two-bit prediction buffer for the SPEC89 benchmarks.

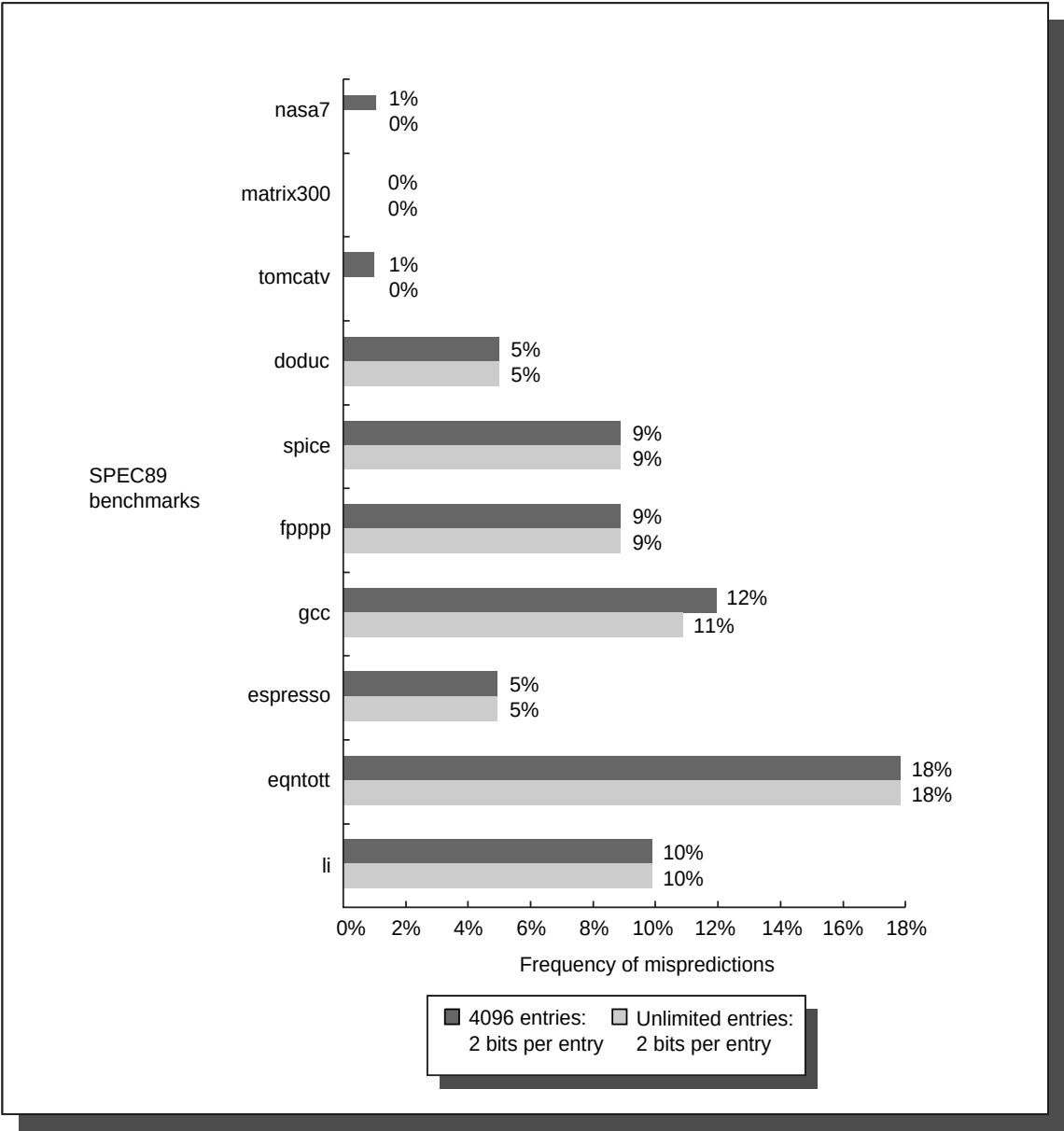


FIGURE 4.15 Prediction accuracy of a 4096-entry two-bit prediction buffer versus an infinite buffer for the SPEC89 benchmarks.

EE 4720 Lecture Transparency. Formatted 13:31, 2 December 1999 from lsli12.

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.

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 $> 2^{n-1}$.

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

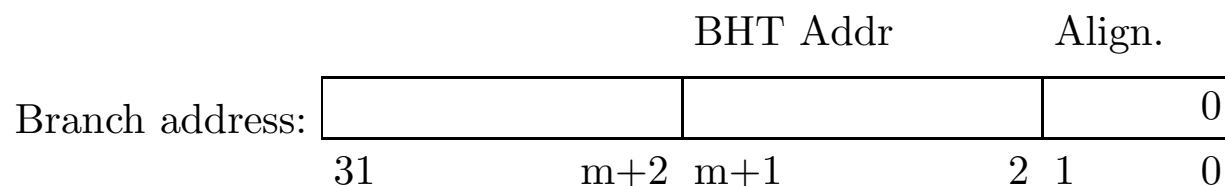
Branch History Table

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**.



Data input and output of BHT is branch history.

Sample Local History

Outcomes for individual branches, categorized by pattern, sorted by frequency.

Branches running T_EX 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 TTTTTTTNNTTTTTTTT 0
1: ff3f 0.0004 1323 0.924 0.909 0.900 TTTTTTNNTTTTTTTT 0
2: fcff 0.0004 1317 0.949 0.939 0.948 TTTTTTTTNNTTTTTTT 0
3: ff9f 0.0003 1245 0.910 0.905 0.898 TTTTNNNTTTTTTTTTT 0
4: f9ff 0.0003 1235 0.955 0.950 0.955 TTTTTTTTTTNNTTTTT 0
5: ffcf 0.0003 1188 0.926 0.921 0.923 TTTNNTTTTTTTTTTTT 0
6: 60 0.0003 1163 0.873 0.829 0.854 NNNNNTTNNNNNNNNNN 0
7: 180 0.0003 1159 0.955 0.914 0.926 NNNNNNNTTNNNNNNNN 0
8: 300 0.0003 1158 0.949 0.926 0.934 NNNNNNNNTTNNNNNNNN 0
9: c0 0.0003 1155 0.944 0.917 0.926 NNNNNNTTNNNNNNNNNN 0
```

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	TNNTNNTNNTNNTNNT 1
3:	4924	0.0022	8055	0.997	0.998	0.998	NNTNNTNNTNNTNNTN 1
4:	2492	0.0022	8047	0.993	0.991	0.009	NTNNTNNTNNTNNTNN 1
5:	db6d	0.0013	4864	0.713	0.915	0.065	TNTTNTTNTTNTTNTT 1
6:	b6db	0.0013	4713	0.862	0.903	0.926	TTNTTNTTNTTNTTNT 1
7:	6db6	0.0012	4640	0.991	0.978	0.970	NTTNTTNTTNTTNTTN 1
8:	bbbb	0.0008	3061	0.896	0.936	0.949	TTNTTTNTTTNTTTNT 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	NTTTTTTTTTTTTTTTT 2
1:	8000	0.0025	9198	0.654	0.700	0.705	NNNNNNNNNNNNNNNT 2
2:	7fff	0.0022	8052	0.890	0.817	0.818	TTTTTTTTTTTTTTTTN 2
3:	ffbf	0.0018	6800	0.933	0.908	0.920	TTTTTTNTTTTTTTTTT 2
4:	feff	0.0018	6782	0.946	0.938	0.942	TTTTTTTTNTTTTTTTT 2
5:	ff7f	0.0018	6778	0.949	0.946	0.950	TTTTTTTTNTTTTTTTTT 2
6:	fdf	0.0018	6738	0.947	0.941	0.946	TTTTTTTTNTTTTTTTT 2
7:	1	0.0018	6690	0.955	0.945	0.942	TNNNNNNNNNNNNNNN 2
8:	fffd	0.0018	6667	0.968	0.966	0.967	TNTTTTTTTTTTTTTTT 2

Phase Change, pat 26, br 48190, 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	NNNNNNNNNNNNNNNTT	3
1:	e000	0.0009	3420	0.714	0.859	0.758	NNNNNNNNNNNNNNNTT	3
2:	f000	0.0008	2942	0.756	0.888	0.788	NNNNNNNNNNNNNNNTT	3
3:	fffc	0.0008	2878	0.908	0.960	0.959	NNTTTTTTTTTTTTTTT	3
4:	f800	0.0007	2642	0.786	0.917	0.827	NNNNNNNNNNNNNTTTT	3
5:	3	0.0007	2572	0.968	0.952	0.951	TTNNNNNNNNNNNNNNN	3
6:	fc00	0.0007	2435	0.815	0.933	0.854	NNNNNNNNNNNTTTTTT	3
7:	fe00	0.0006	2225	0.836	0.936	0.876	NNNNNNNNNNNTTTTTT	3
8:	ff00	0.0006	2140	0.856	0.947	0.931	NNNNNNNNNTTTTTTTT	3
9:	ff80	0.0006	2061	0.854	0.941	0.934	NNNNNNNTTTTTTTTTT	3

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

0:	ffff	0.5151	1916950	0.993	0.996	0.993	TTTTTTTTTTTTTTTTT	4
1:	0	0.1882	700483	0.988	0.986	0.982	NNNNNNNNNNNNNNNNN	4

Register `r1` not available until cycle ten¹.

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).

`! Predict not taken, not taken.`

Cycle:	0	1	2	3		10	11	12	13
<code>bneq r1, TARGET</code>	IF	ID	0:RS	0:RS ...		0:B	0:WC		
<code>xor r2, r3, r4</code>		IF	ID	5:EX	5:WB			5:C	
									6:C

...

TARGET:

`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.

Cycle:	0	1	2	3		10	11	...	21	22	23
<code>div f0,f2, f4</code>	ID	DIV							DIV	WC	
<code>bneq r1, TARGET</code>	IF	ID	0:RS	0:RS	...	0:B	0:WB				C
<code>xor r2, r3, r4</code>		IFx									
...											
TARGET:											
<code>and r5, r6, r7</code>				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.

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

Cycle:	0	1	2	3		10	11	...	21	22	23
<code>div f0,f2, f4</code>	ID	DIV							DIV	WC	
<code>bneq r1, TARGET</code>	IF	ID	0:RS	0:RS	...	0:B	0:WB				C
<code>xor r2, r3, r4</code>		IF	ID	EX	...						
...											
TARGET:											
<code>and r5, r6, r7</code>										IF	

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: oops, 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.

Cycle:	0	1	2	3		10	11	...	21	22	23
<code>div f0,f2, f4</code>	ID	DIV							DIV	WC	
<code>bneq r1, TARGET</code>	IF	ID	0:RS	0:RS	...	0:B	0:WB				C
<code>xor r2, r3, r4</code>		IF	ID	EX							
...											
TARGET:											
<code>and r5, r6, r7</code>								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.

```

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

```

BHT Indexing for (m, n) Predictor

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 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									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 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									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 **bneq** and first **beqz** not taken.

Cycle 11: Oops, **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.

Global History in Two-Level Predictor with Dynamic Execution

Global history backed up (*checkpointed*) at each branch.

Predicted outcome shifted in to global history.

If misprediction discovered, global history restored from backup ...
... 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).

Cycle:	0	1	2	3	4		10	11	12	13	14
bneq r1, TARGET	IF	ID	EX	MEM	WB		IF	ID	EX	MEM	WB
xor r2, r3, r4									IF	ID	EX
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.

Static scheduled system (for clarity).

Cycle:	0	1	2	3	4		10	11	12	13	14
bneq r1, TARGET	IF	ID	EX	MEM	WB		IF	ID	EX	MEM	WB
xor r2, r3, r4									IF	ID	EX
TARGET:											
and r5, r6, r7		IF	ID	EX	MEM	WB		IF	X		

Cycle 1

Start fetching predicted target.

Execute branch instruction (in ID).

Check predicted outcome and predicted target.

Correct predictions, continue execution.

Cycle:	0	1	2	3	4		10	11	12	13	14
bneq r1, TARGET	IF	ID	EX	MEM	WB		IF	ID	EX	MEM	WB
xor r2, r3, r4									IF	ID	EX
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.

What is being predicted:

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.

For register-indirect jumps (`jr`, `jalr`) ...

... prediction depends on predictable behavior.

Two Methods

Predict last target.

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

For most CTIs ...

... use last target as prediction for next target.

Assuming a BTB hit ...

... last target works perfectly for non-indirect CTIs ...

... but depends on register value not changing for indirect CTIs.

If call and return instructions can be perfectly identified ...

... call stack will make perfect prediction for return instructions.

Consider code for C switch statement:

```
! 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.