Reduction of Control Hazards (Branch) Stalls with Dynamic Branch Prediction

- So far we have dealt with control hazards in instruction pipelines by:

1. – Assuming that the branch is not taken (i.e stall when branch is taken).
2. – Reducing the branch penalty by resolving the branch early in the pipeline
   - Branch penalty if branch is taken = stage resolved - 1

In IF ? 1-1 = 0?

3. – Branch delay slot and canceling branch delay slot. (ISA support needed)
4. – Compiler-based static branch prediction encoded in branch instructions
   - Prediction is based on program profile or branch direction
   - ISA support needed.

ISA Support Needed

How to further reduce the impact of branches on pipeline processor performance ?

- **Dynamic Branch Prediction:**

Why? Better branch prediction accuracy than static prediction and thus fewer branch stalls

Why?

By CPU

– Hardware-based schemes that utilize run-time behavior of branches to make dynamic predictions: + *No ISA support needed* ←

How?

- Information about outcomes of previous occurrences of branches are used to dynamically predict the outcome of the current branch.

- **Branch Target Buffer (BTB):** *(Goal: zero stall taken branches)* ←

– A hardware mechanism that aims at reducing the stall cycles resulting from correctly predicted taken branches to zero cycles (No stalls).

4th Edition: Static and Dynamic Prediction in ch. 2.3, BTB in Ch. 2.9
(3rd Edition: Static Pred. in Ch. 4.2 Dynamic Pred. in Ch. 3.4, BTB in Ch. 3.5)

ISA Support Needed

Static Conditional Branch Prediction

- Branch prediction schemes can be classified into static (at compilation time) and dynamic (at runtime) schemes.

Branch Encoding	X

X = Static Prediction bit
X= 0 Not Taken
X = 1 Taken

- Static methods are carried out by the compiler. They are static because the prediction is already known before the program is executed.
- Static Branch prediction is encoded in branch instructions using one prediction (or branch direction hint) bit = 0 = Not Taken, = 1 = Taken

Must be supported by ISA, Ex: HP PA-RISC, PowerPC, UltraSPARC

How?

Two basic methods to statically predict branches at compile time:

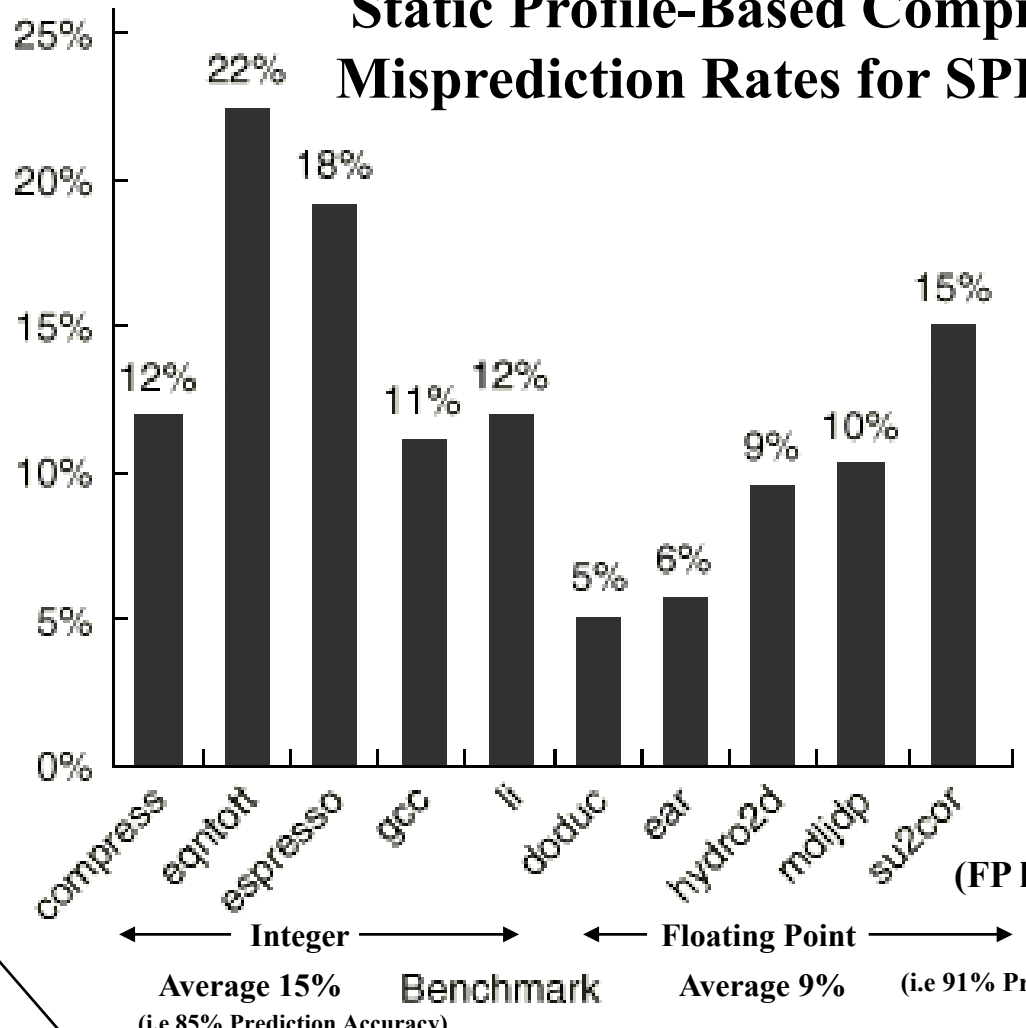
1 – Use the direction of a branch to base the prediction on. Predict backward branches (branches which decrease the PC) to be taken (e.g. branches used in loops) and forward branches (branches which increase the PC) not to be taken.

2 – Profiling can also be used to predict the outcome of a branch.

- A number runs of the program are used to collect program behavior information (i.e. if a given branch is likely to be taken or not)
- This information is included in the opcode of the branch (one bit branch direction hint) as the static prediction.

Static prediction was previously discussed in lecture 2
4th edition: in Chapter 2.3, 3rd Edition: In Chapter 4.2

Static Profile-Based Compiler Branch Misprediction Rates for SPEC92



→ **More Loops In FP Code**

(FP has more loops)

Thus lower misprediction rate In FP code

Misprediction rate for a profile-based predictor varies widely but is generally better for the FP programs, which have an average misprediction rate of 9% with a standard deviation of 4%, than for the integer programs, which have an average misprediction rate of 15% with a standard deviation of 5%.

(repeated here from lecture2)

Dynamic Conditional Branch Prediction

No ISA Support Needed ←

- Dynamic branch prediction schemes are different from static mechanisms because they utilize hardware-based mechanisms that use the run-time behavior of branches to make more accurate predictions than possible using static prediction.

How?

Why?

- Usually information about outcomes of previous occurrences of branches (branching history) is used to dynamically predict the outcome of the current branch. The two main types of dynamic branch prediction are:

1 – One-level or Bimodal: Usually implemented as a Pattern History Table (PHT), a table of usually two-bit saturating counters (predictors) which is indexed by a portion of the branch address (low bits of address). (First proposed mid 1980s)

  - Also called *non-correlating* dynamic branch predictors. ←

2 – Two-Level Adaptive Branch Prediction. (First proposed early 1990s).

  - Also called *correlating* dynamic branch predictors. ←

\+ BTB

- To potentially reduce the stall cycles resulting from correctly predicted taken branches to zero cycles, a Branch Target Buffer (BTB) that includes the addresses of conditional branches that were taken along with their targets is added to the instruction fetch (IF) stage.

BTB discussed next →

4th Edition: Dynamic Prediction in Chapter 2.3, BTB in Chapter 2.9
(3rd Dynamic Prediction in Chapter 3.4, BTB in Chapter 3.5)

Branch Target Buffer (BTB)

Why?

- Effective branch prediction requires the target of the branch at an early pipeline stage. (resolve the branch early in the pipeline)

In IF ? 1-1 = 0?

  - One can use additional adders to calculate the target, as soon as the branch instruction is decoded. This would mean that one has to wait until the ID stage before the target of the branch can be fetched, taken branches would be fetched with a one-cycle penalty (this was done in the enhanced MIPS pipeline Fig A.24).

BTB Goal

- To avoid this problem and to potentially achieve zero stall cycles for taken branches, one can use a Branch Target Buffer (BTB).

How?

- A typical BTB is an associative memory where the addresses of taken branch instructions are stored together with their target addresses.
- The BTB is is accessed in Instruction Fetch (IF) cycle and provides answers to the following three questions while the current instruction is being fetched:

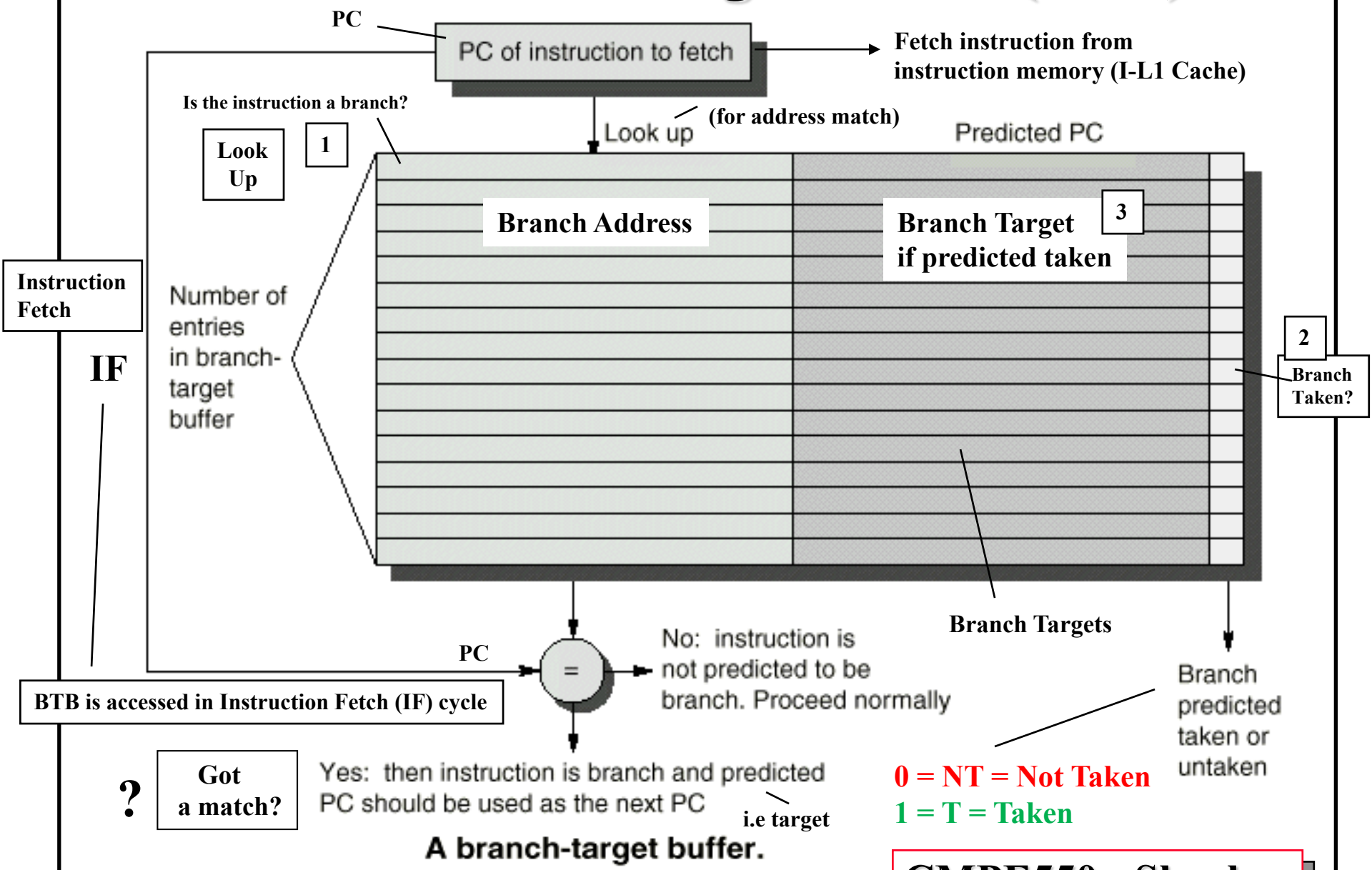
1 – Is the instruction a branch?
2 – If yes, is the branch predicted taken?
3 – If yes, what is the branch target?

- Instructions fetched from the target are stored in the BTB in case the branch is predicted-taken and thus found in BTB.
- After the branch has been resolved the BTB is updated. If a branch is encountered for the first time a new entry is created once it is resolved as taken.

Goal of BTB: Zero stalls for taken branches

4th Edition: BTB in Chapter 2.9 (pages 121-122)
(3rd BTB in Chapter 3.5)

Basic Branch Target Buffer (BTB)

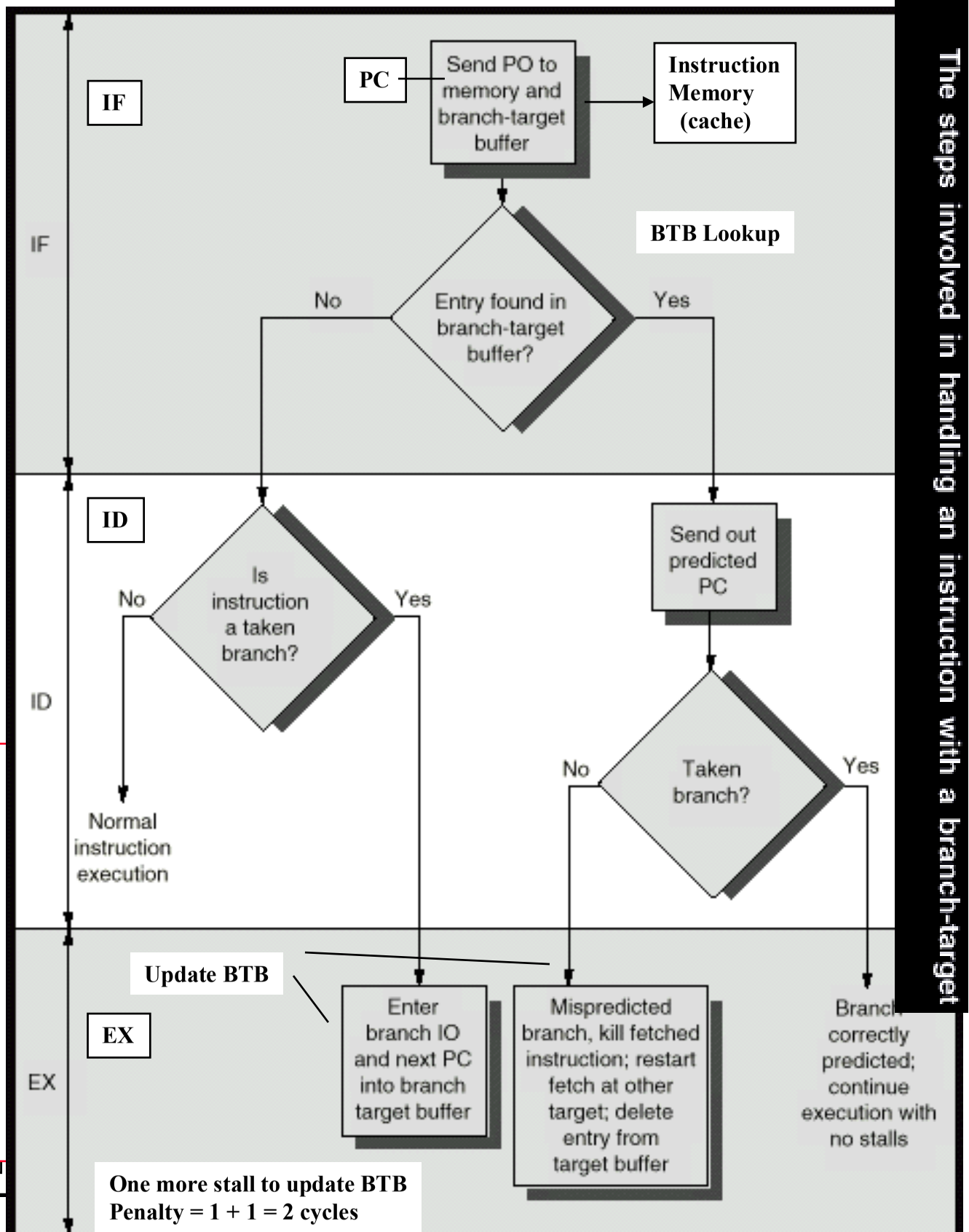


A branch-target buffer.

Goal of BTB: Zero stalls for taken branches

Here, branches are assumed to be resolved in ID

BTB Operation



buffer.

The steps involved in handling an instruction with a branch-target

Branch Penalty Cycles Using A Branch-Target Buffer (BTB)

➔ **Base Pipeline Taken Branch Penalty = 1 cycle (*i.e. branches resolved in ID*)**

i.e In BTB? Or Not A Branch

No **Not Taken** **Not Taken** **0**

Instruction in buffer	Prediction	Actual branch	Penalty cycles
Yes	Taken	Taken	0
Yes	Taken	Not taken	2 = 1 + 1
No		Taken	2 = 1 + 1

BTB Goal: Taken Branches with zero stalls

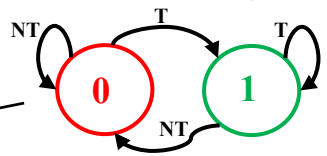
➔ **Assuming one more stall cycle to update BTB**
Penalty = 1 + 1 = 2 cycles

Penalties for all possible combinations of whether the branch is in the buffer and what it actually does, assuming we store only taken branches in the buffer.

Basic Dynamic Branch Prediction

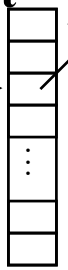
- **Simplest method: (One-Level or Non-Correlating)** ←
  - A branch prediction buffer or Pattern History Table (PHT) indexed by N low address bits of the branch instruction.
  - Each buffer location (or PHT entry or predictor) contains one bit indicating whether the branch was recently taken or not
    - e.g 0 = not taken , 1 =taken

Predictor = Saturating Counter



N Low Bits of Branch Address →

Saturating counter



PHT

PHT Entry: One Bit
0 = NT = Not Taken
1 = T = Taken
2^N entries or predictors

  - Always mispredicts in first and last loop iterations.

- **To improve prediction accuracy, two-bit prediction is used:**
  - A prediction must miss twice before it is changed. → (Smith Algorithm, 1985)

Why 2-bit Prediction?

- Thus, a branch involved in a loop will be mispredicted only once when encountered the next time as opposed to twice when one bit is used.

  - Two-bit prediction is a specific case of n-bit saturating counter incremented when the branch is taken and decremented when the branch is not taken.

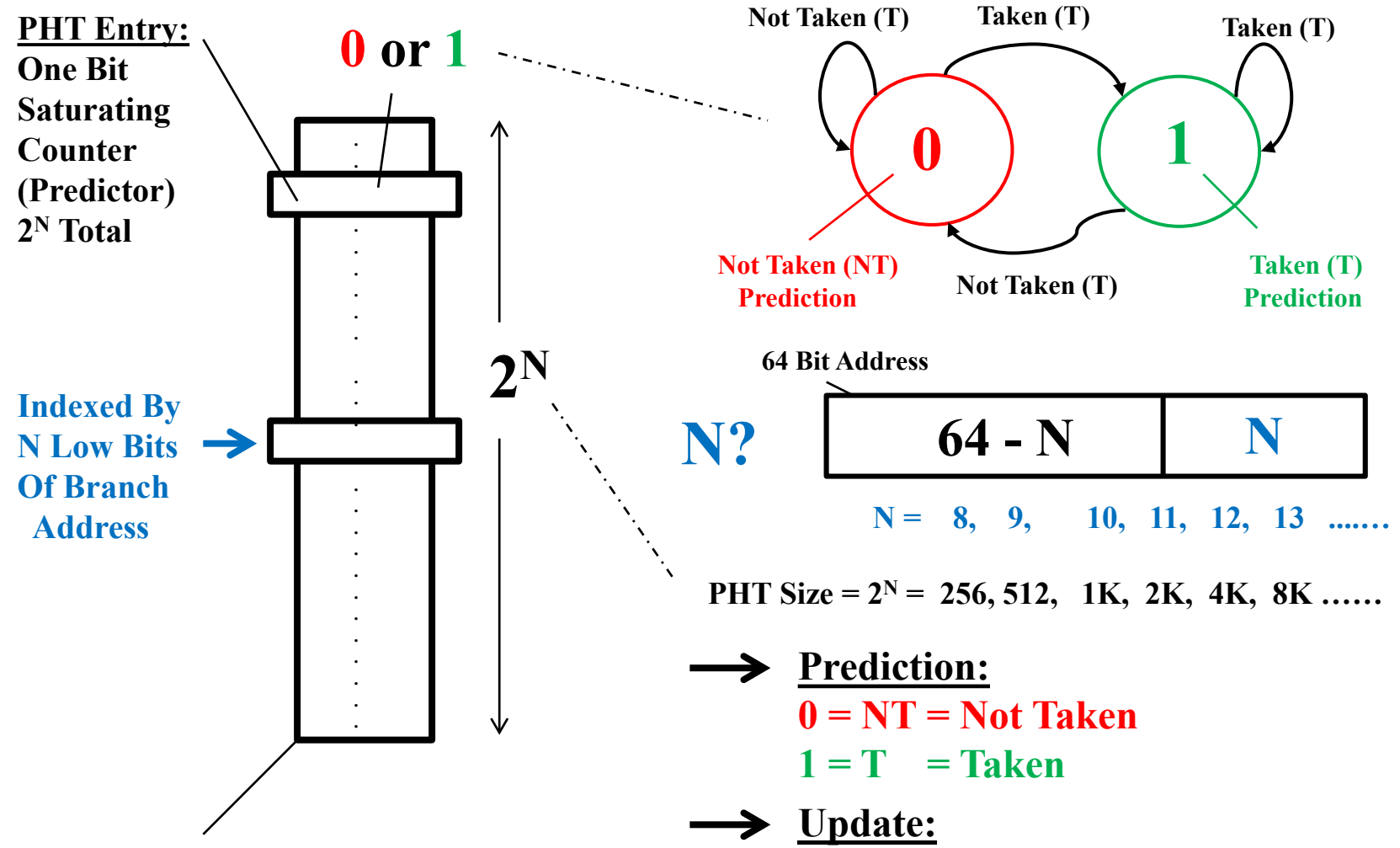
The counter (predictor) used is updated after the branch is resolved

  - Two-bit saturating counters (predictors) are usually always used based on observations that the performance of two-bit PHT prediction is comparable to that of n-bit predictors.

Smith Algorithm

4th Edition: In Chapter 2.3 (3rd Edition: In Chapter 3.4)

One Level (Bimodal) Dynamic Branch Prediction

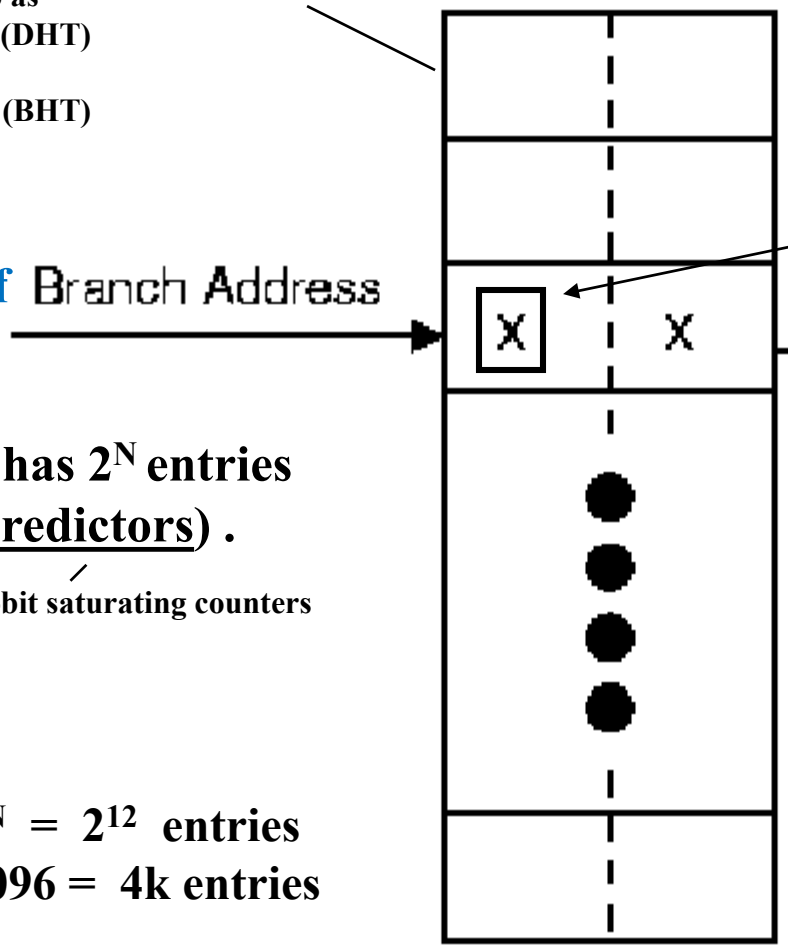


One-Level Bimodal Branch Predictors
Pattern History Table (PHT)

Most common one-level implementation

Sometimes referred to as
Decode History Table (DHT)
or
Branch History Table (BHT)

2-bit saturating counters (predictors)



High bit determines branch prediction

0 = NT = Not Taken
1 = T = Taken

Indexed by
N Low Bits of Branch Address

Prediction Bits

0	0	Not Taken (NT)
0	1	
1	0	Taken (T)
1	1	

Table (PHT) has 2^N entries
(also called predictors) .
2-bit saturating counters

Example:

For N =12
Table has $2^N = 2^{12}$ entries
$= 4096 = 4k$ entries

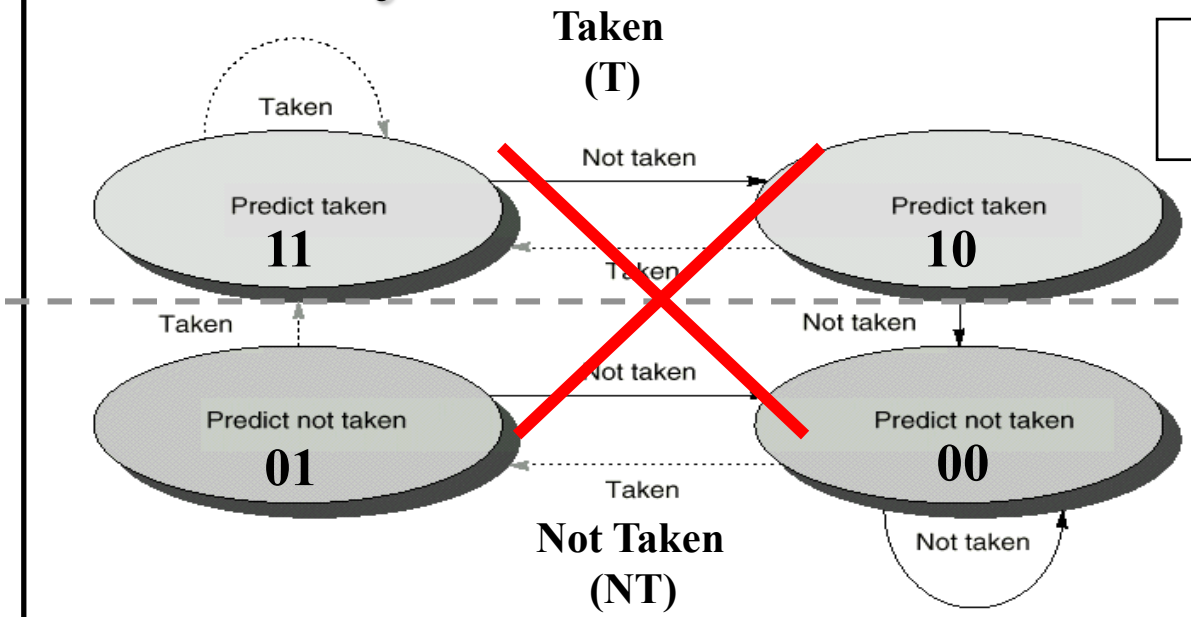
Number of bits needed = 2 x 4k = 8k bits

When to update

Update counter after branch is resolved:
-Increment counter used if branch is taken
- Decrement counter used if branch is not taken

What if different branches map to the same predictor (counter)?
This is called branch address aliasing and leads to interference with current branch prediction by other branches and may lower branch prediction accuracy for programs with aliasing.

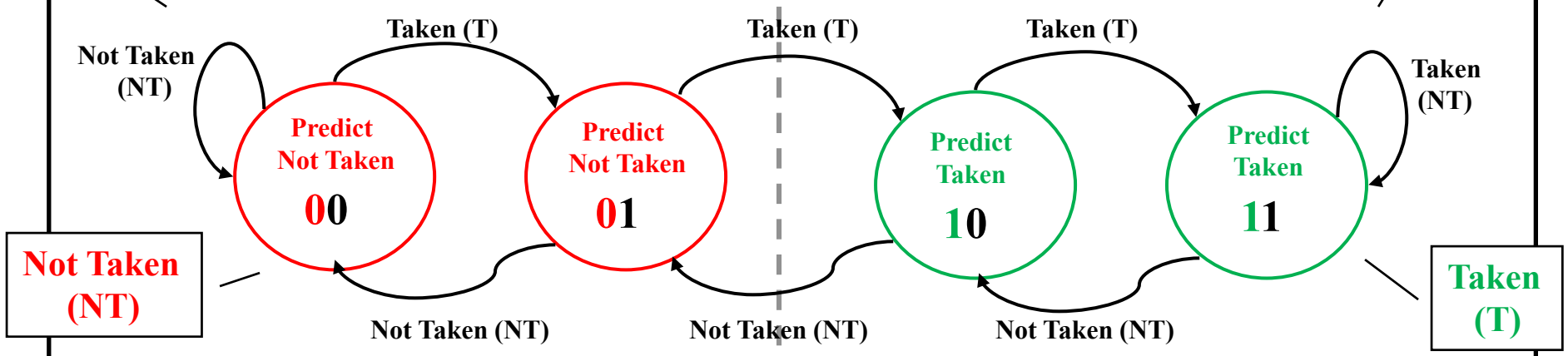
Basic Dynamic Two-Bit Branch Prediction:



Two-bit Predictor State Transition Diagram (in textbook)

0	0	Not Taken (NT)
0	1	
1	0	Taken (T)
1	1	

Or Two-bit saturating counter predictor state transition diagram (Smith Algorithm):



The two-bit predictor used is updated after the branch is resolved

Prediction Accuracy of A 4096-Entry Basic One-Level Dynamic Two-Bit Branch Predictor

N=12
$2^N = 4096$

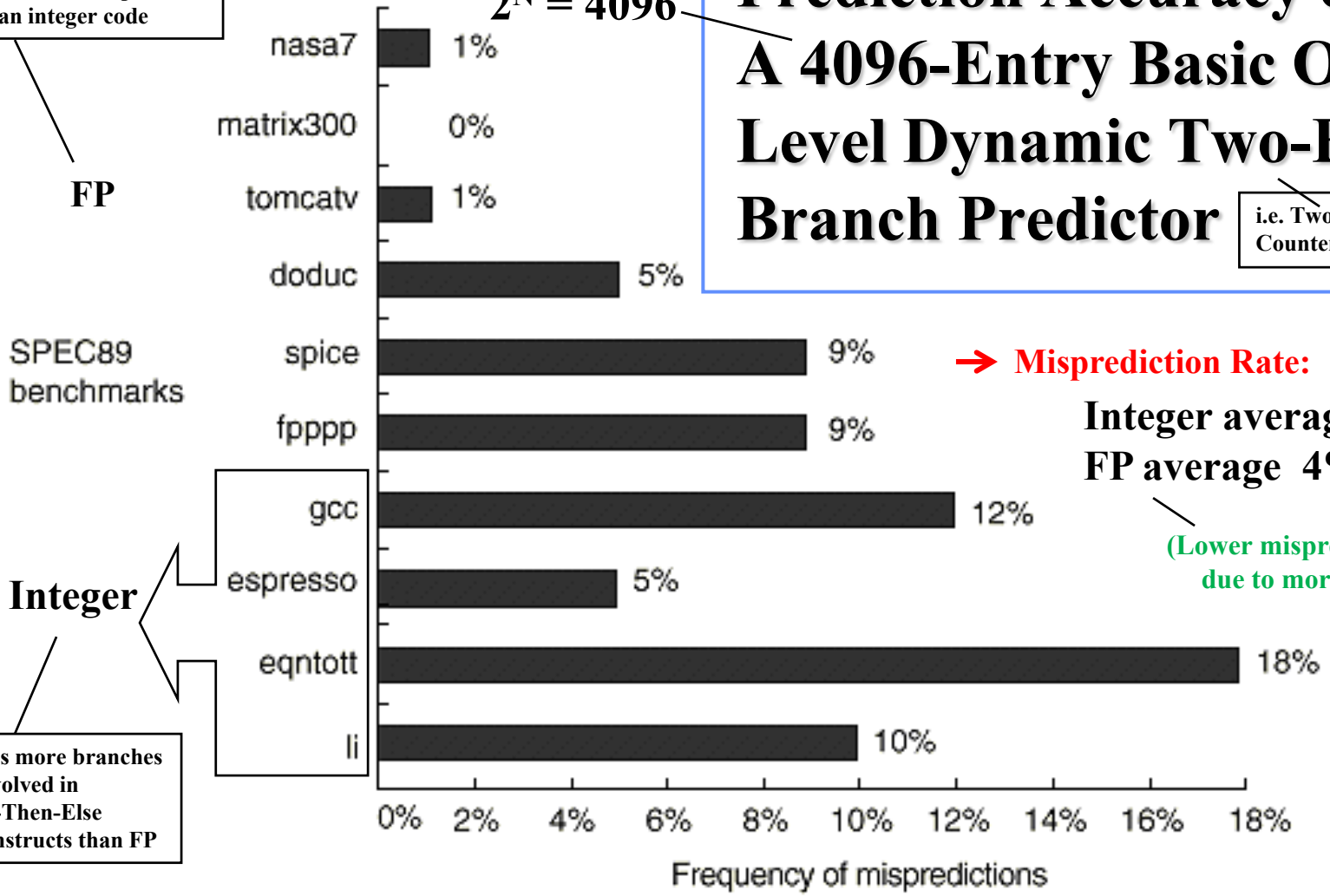
i.e. Two-bit Saturating Counters (Smith Algorithm)

FP has more branches involved in loops than integer code

FP

Integer

Has more branches involved in IF-Then-Else constructs than FP



Misprediction Rate:

Integer average 11%
FP average 4%

(Lower misprediction rate due to more loops)

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

From The Analysis of Static Branch Prediction :

MIPS Performance Using Canceling Delay Branches

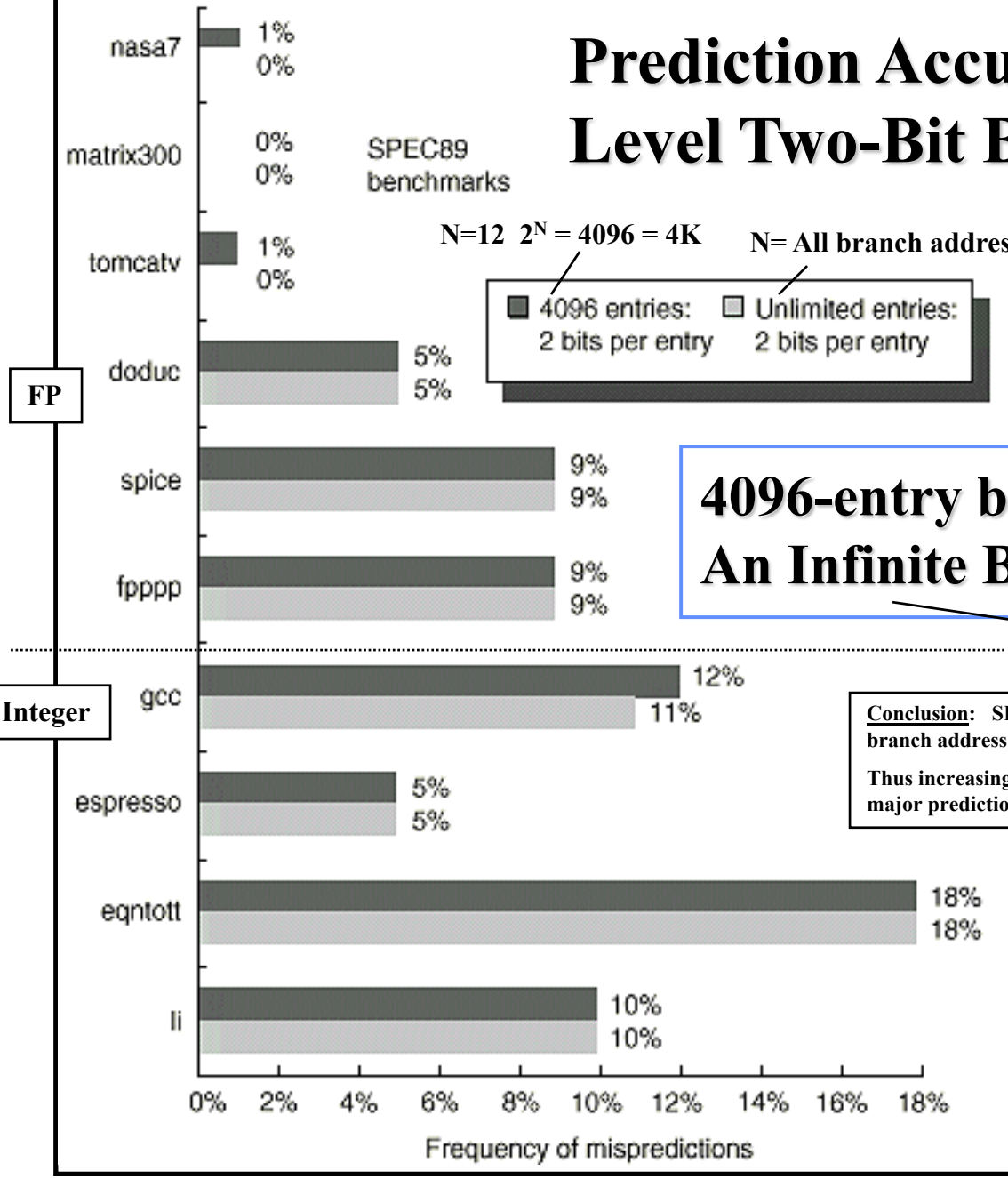
Benchmark	% conditional branches	% conditional branches with empty slots	% conditional branches that are cancelling	% cancelling branches that are cancelled	% branches with cancelled delay slots	Total % branches with empty or cancelled delay slot
compress	14%	18%	31%	43%	13%	31%
eqntott	24%	24%	50%	24%	12%	36%
espresso	15%	29%	19%	21%	4%	33%
gcc	15%	16%	33%	34%	11%	27%
li	15%	20%	55%	48%	26%	46%
Integer average	17%	21%	38%	34%	13%	35%
doduc	8%	33%	12%	62%	8%	41%
ear	10%	37%	36%	14%	5%	42%
hydro2d	12%	0%	69%	24%	16%	17%
mdljdp2	9%	0%	86%	10%	8%	8%
su2cor	3%	7%	17%	57%	10%	17%
FP average	8%	16%	44%	34%	9%	25%
Overall average	12%	18%	41%	34%	11%	30%

Delayed and cancelling delay branches for MIPS allow branch hazards to be hidden 70% of the time on average for these 10 SPEC benchmarks.

➔ **70% Static Branch Prediction Accuracy**

(repeated here from lecture2) ➔ **+ ISA Support Needed**

Prediction Accuracy of Basic One-Level Two-Bit Branch Predictors:



N=12 $2^N = 4096 = 4K$

N= All branch address bits

4096 entries: 2 bits per entry

Unlimited entries: 2 bits per entry

4096-entry buffer (PHT) Vs. An Infinite Buffer Under SPEC89

i.e. All branch address bit are used

Conclusion: SPEC89 programs do not have many branches that suffer from branch address aliasing (interference) when using a 4096-entry PHT.

Thus increasing PHT size (which usually lowers aliasing) did not result in major prediction accuracy improvement.

Correlating Branches

- **Recent branches are possibly correlated:** The behavior of recently executed branches affects outcome of current branch.

US — B3 in the example below

Occur in branches used to implement if-then-else constructs
Which are more common in integer than floating point code

Example:

Here aa = R1 bb = R2

B1	if (aa==3)		DSUBUI	R3, R1, #3	; R3 = R1 - 3
			BNEZ	R3, L1	; B1 (aa!=3)
	aa=0; (not taken)		DADD	R1, R0, R0	; aa==0
B2	if (bb==2)	L1:	DSUBUI	R3, R2, #2	; R3 = R2 - 2
			BNEZ	R3, L2	; B2 (bb!=2)
	bb=0; (not taken)		DADD	R2, R0, R0	; bb==0
B3	if (aa!==bb){	L2:	DSUBUI	R3, R1, R2	; R3=aa-bb
			BEQZ	R3, L3	; B3 (aa==bb)

B1 not taken R1 =0

B2 not taken R2 =0

B3 taken if aa=bb i.e. R1=R2=0

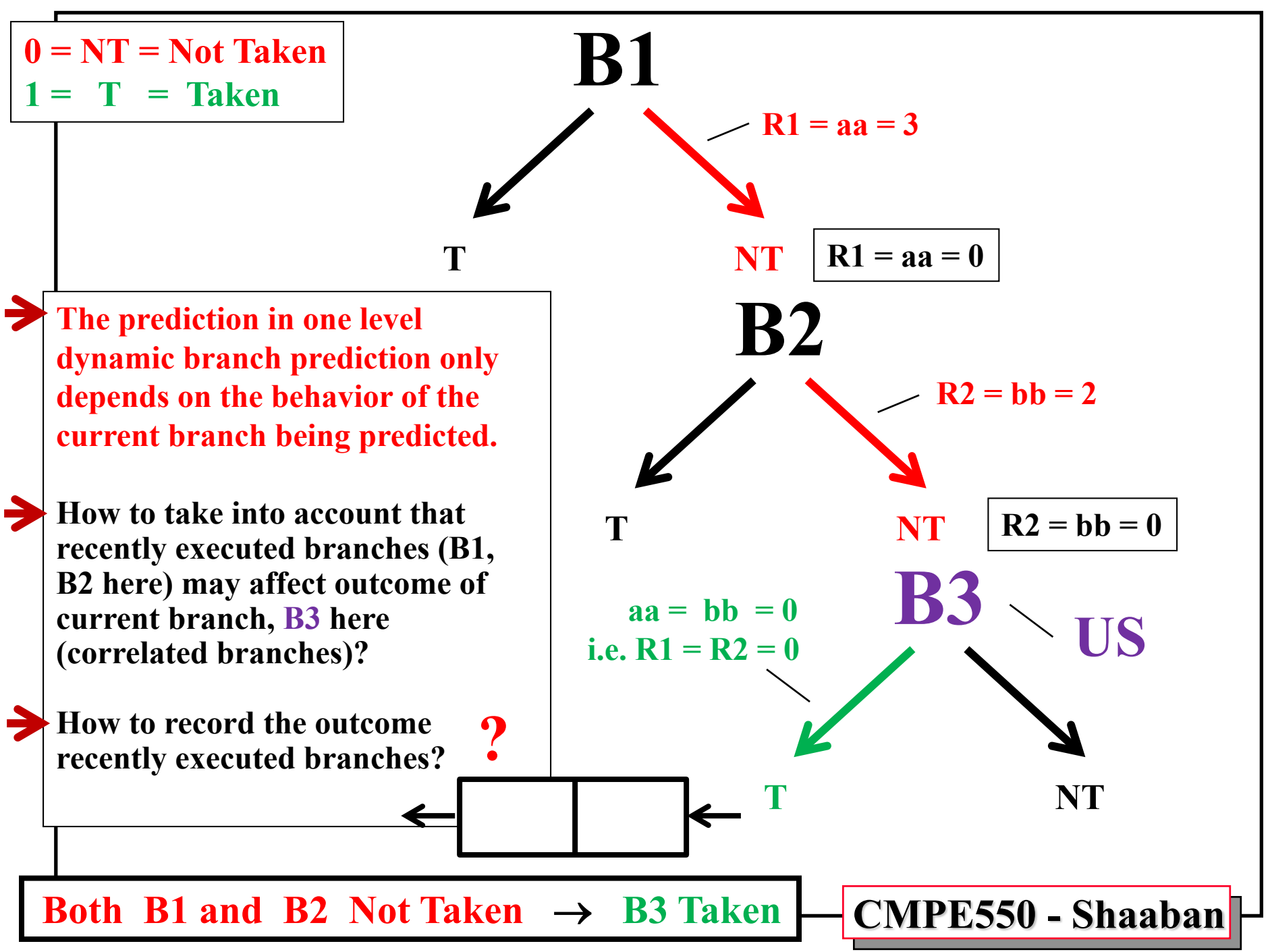
aa=3 bb=2

Branch B3 is correlated with branches B1, B2. If B1, B2 are both not taken, then B3 will be taken. Using only the behavior of one branch cannot detect this behavior.

B3 in this case

Both B1 and B2 Not Taken → B3 Taken

0 = NT = Not Taken
1 = T = Taken

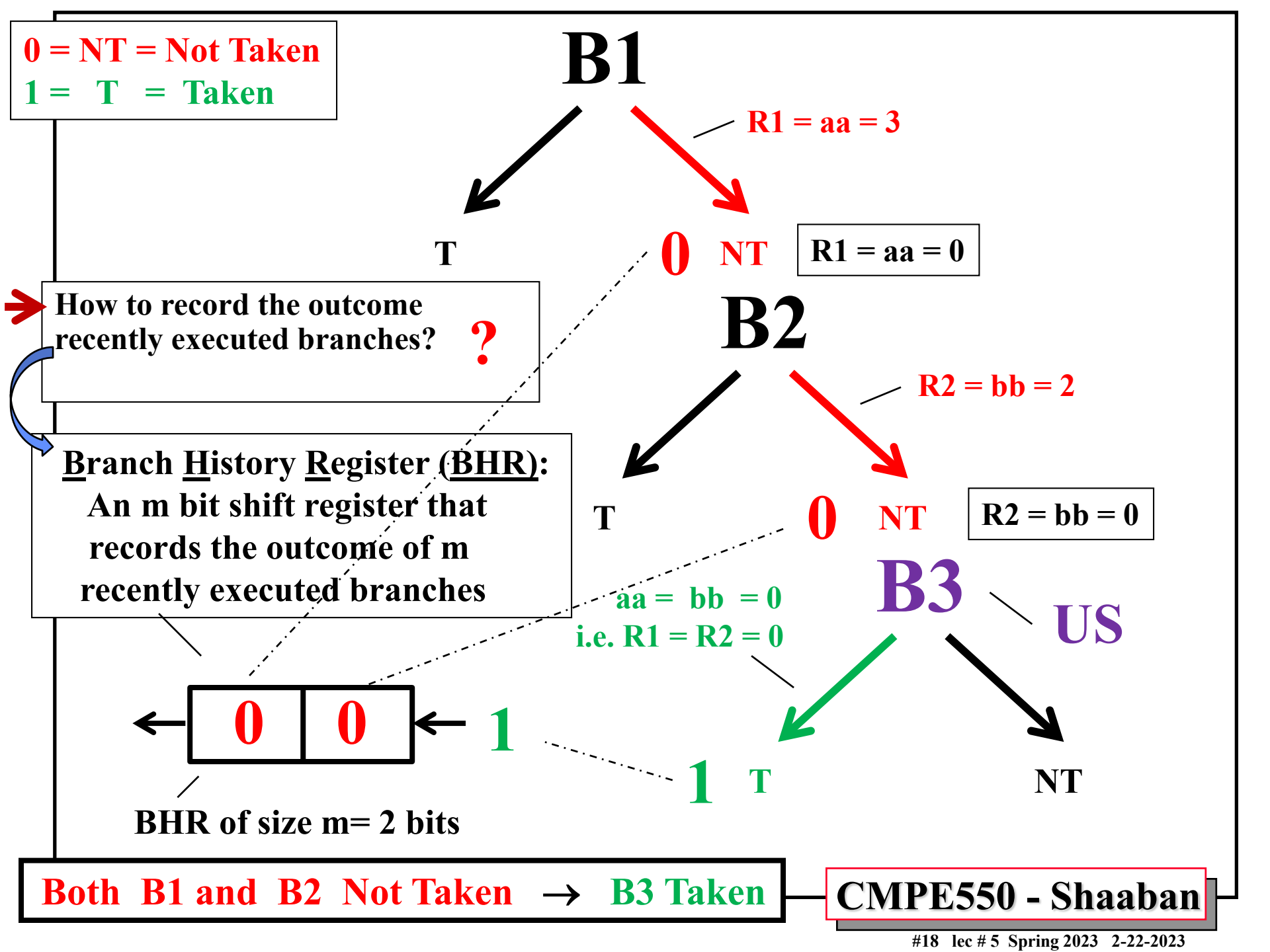


The prediction in one level dynamic branch prediction only depends on the behavior of the current branch being predicted.

How to take into account that recently executed branches (B1, B2 here) may affect outcome of current branch, B3 here (correlated branches)?

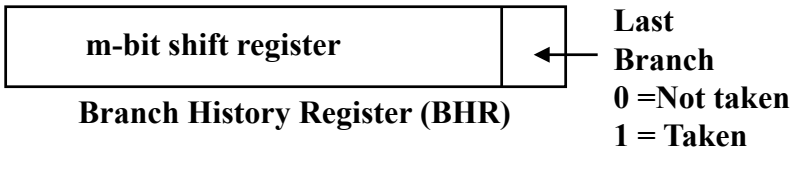
How to record the outcome recently executed branches?

Both B1 and B2 Not Taken → B3 Taken


0 = NT = Not Taken
1 = T = Taken
B1
R1 = aa = 3
T
0 NT
R1 = aa = 0
B2
How to record the outcome recently executed branches?
?
R2 = bb = 2
Branch History Register (BHR):
An m bit shift register that records the outcome of m recently executed branches
T
0 NT
R2 = bb = 0
B3
US
aa = bb = 0
i.e. R1 = R2 = 0
0 0
1
1 T
NT
BHR of size m= 2 bits
Both B1 and B2 Not Taken → B3 Taken

Correlating Two-Level Dynamic GAp Branch Predictors

- Improve branch prediction by looking not only at the history of the branch in question but also at that of other branches using two levels of branch history.
- Uses two levels of branch history:



1 – **First level (global):** BHR

One BHR

- Record the global pattern or history of the m most recently executed branches as taken or not taken. Usually an m-bit shift register.

2 – **Second level (per branch address):** 2^m Pattern History Tables (PHTs)

2^m PHTs

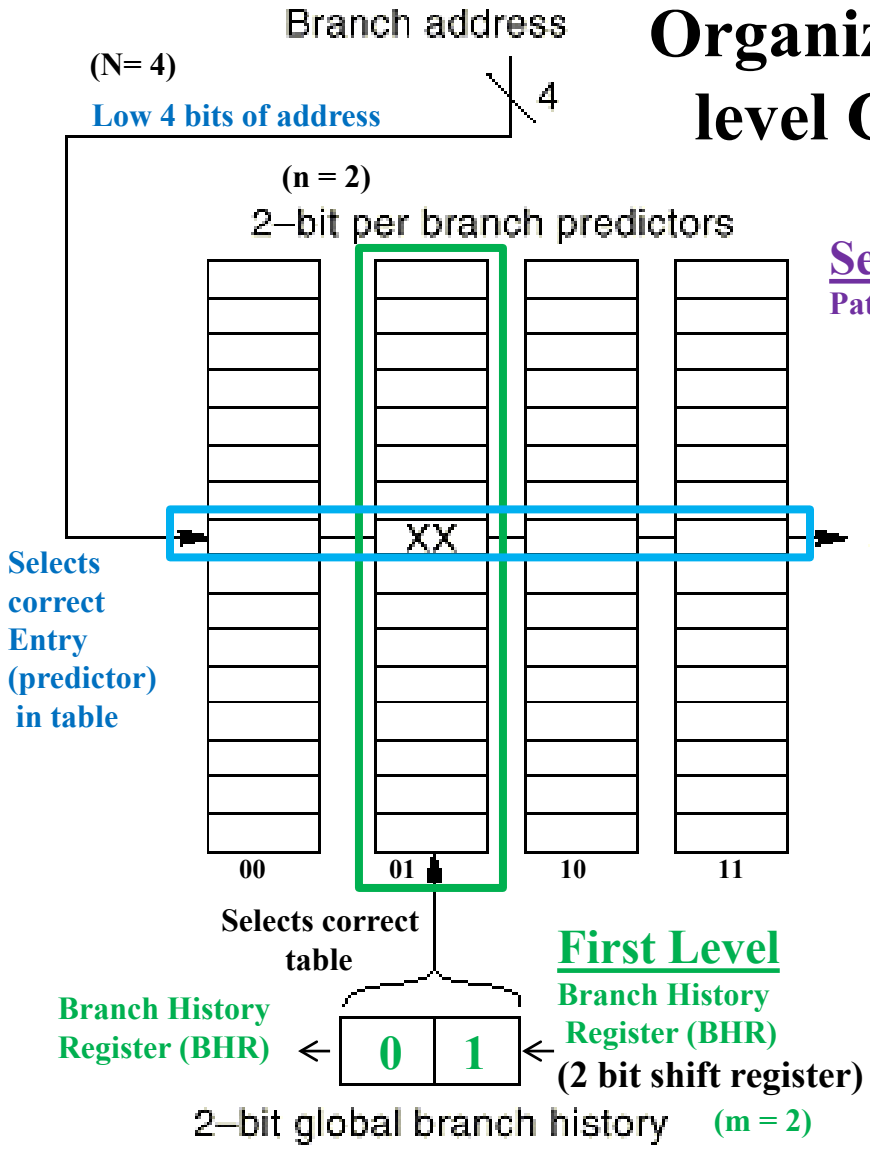
- 2^m prediction tables, each table entry has n bit saturating counter.
- The branch history pattern or register (BHR) from first level is used to select the proper branch prediction table in the second level.
- The low N bits of the branch address are used to select the correct prediction entry (predictor)within a the selected table, thus each of the 2^m tables has 2^N entries and each entry is 2 bits counter.
- Total number of bits needed for second level = $2^m \times n \times 2^N$ bits

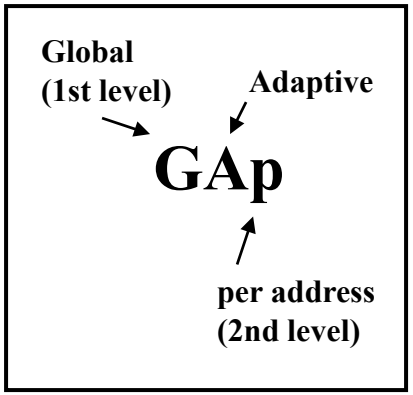
- In general, the notation: GAp (m,n) predictor means:
  - Record last m branches to select between 2^m history tables.
  - Each second level table uses n-bit counters (each table entry has n bits).
- Basic two-bit single-level Bimodal BHT is then a (0,2) predictor.

GAp (m,n)

4th Edition: In Chapter 2.3 (3rd Edition: In Chapter 3.4)

Organization of A Correlating Two-level GAp (2,2) Branch Predictor





m = # of branches tracked in first level = 2
Thus $2^m = 2^2 = 4$ tables in second level

N = # of low bits of branch address used = 4
Thus each table in 2nd level has 2N = 24 = 16 entries

n = # number of bits of 2nd level table entry = 2

Number of bits for 2nd level = 2^m x n x 2^N
= 4 x 2 x 16 = 128 bits

A (2,2) branch-prediction buffer uses a two-bit global history to choose from among four predictors for each branch address.

GAp (m,n) here m= 2 n =2 Thus Gap (2, 2)

Dynamic Branch Prediction: Example

b1 Not Taken (d = R1 = 0) → d = R1 = 1

```
if (d==0)
    d=1;
if (d==1)
```

Not Taken

d = R1 = 1 → b2 Not Taken

```
        b1   BNEZ    R1, L1      ; branch b1 (d!=0)
             DADDIU  R1, R0, #1  ;  d==0, so d=1
L1:          DADDIU  R3, R1, # -1
        b2   BNEZ    R3, L2      ;  branch  b2  (d!=1)
. . .
L2:
```

b1 Taken

b2 Taken

Possible execution sequences for a code fragment.

Initial value of d	d==0?	b1		Value of d before b2	d==1?	b2
0	Yes	Not taken	→	1	Yes	Not taken
1	No	Taken	→	1	Yes	Not taken
2	No	Taken	→	2	No	Taken

One Level

Behavior of a one-bit predictor initialized to not taken.

d=?	b1 prediction	b1 action	New b1 prediction	b2 prediction	b2 action	New b2 prediction
2	NT	T	T	NT	T	T
0	T	NT	NT	T	NT	NT
2	NT	T	T	NT	T	T
0	T	NT	NT	T	NT	NT

One Level with one-bit table entries (predictors) : NT = 0 = Not Taken
T = 1 = Taken

Dynamic Branch Prediction: Example (continued)

b1 Not Taken (d = R1 = 0) → d = R1 = 1

```
if (d==0)          b1   BNEZ    R1, L1     ; branch b1 (d!=0)
Not Taken – d=1;        DADDIU  R1, R0, #1  ;  d==0, so d=1
                  L1:   DADDIU  R3, R1, # -1
if (d==1)          b2   BNEZ    R3, L2     ;  branch  b2  (d!=1)
                        . . .
                  L2:
```

b1 Taken

b2 Taken

d = R1 = 1 → b2 Not Taken

Combinations and meaning of the taken/not taken prediction bits.

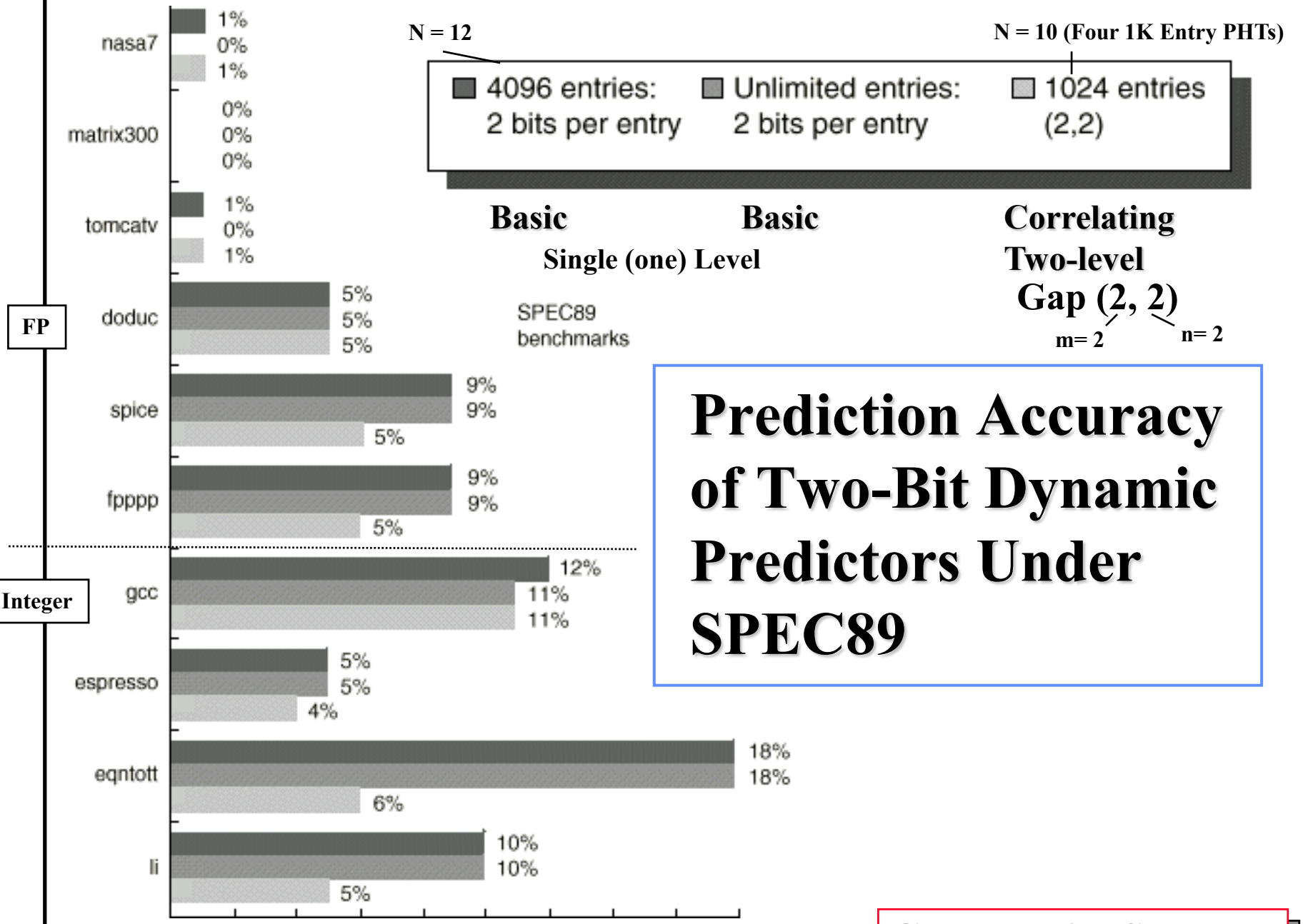
Initial value of d	d==0?	b1		Value of d before b2	d==1?	b2
0	Yes	Not taken	→	1	Yes	Not taken
1	No	Taken	→	1	Yes	Not taken
2	No	Taken	→	2	No	Taken

The action of the one-bit predictor with one bit of correlation, initialized to not taken/not taken.

d=?	b1 prediction	b1 action	New b1 prediction	b2 prediction	b2 action	New b2 prediction
2	**NT**/NT	T	T/NT	NT/**NT**	T	NT/T
0	T/**NT**	NT	T/NT	**NT**/T	NT	NT/T
2	**T**/NT	T	T/NT	NT/**T**	T	NT/T
0	T/**NT**	NT	T/NT	**NT**/T	NT	NT/T

Two level GAp(1,1)

m= 1 n= 1



Prediction Accuracy of Two-Bit Dynamic Predictors Under SPEC89

MCFarling's gshare Predictor

gshare = global history with index sharing

- McFarling noted (1993) that using global history information might be less efficient than simply using the address of the branch instruction, especially for small predictors.
- He suggests using both global history (BHR) and branch address by hashing them together. He proposed using the XOR of global branch history register (BHR) and branch address since he expects that this value has more information than either one of its components. The result is that this mechanism outperforms GAp scheme by a small margin.

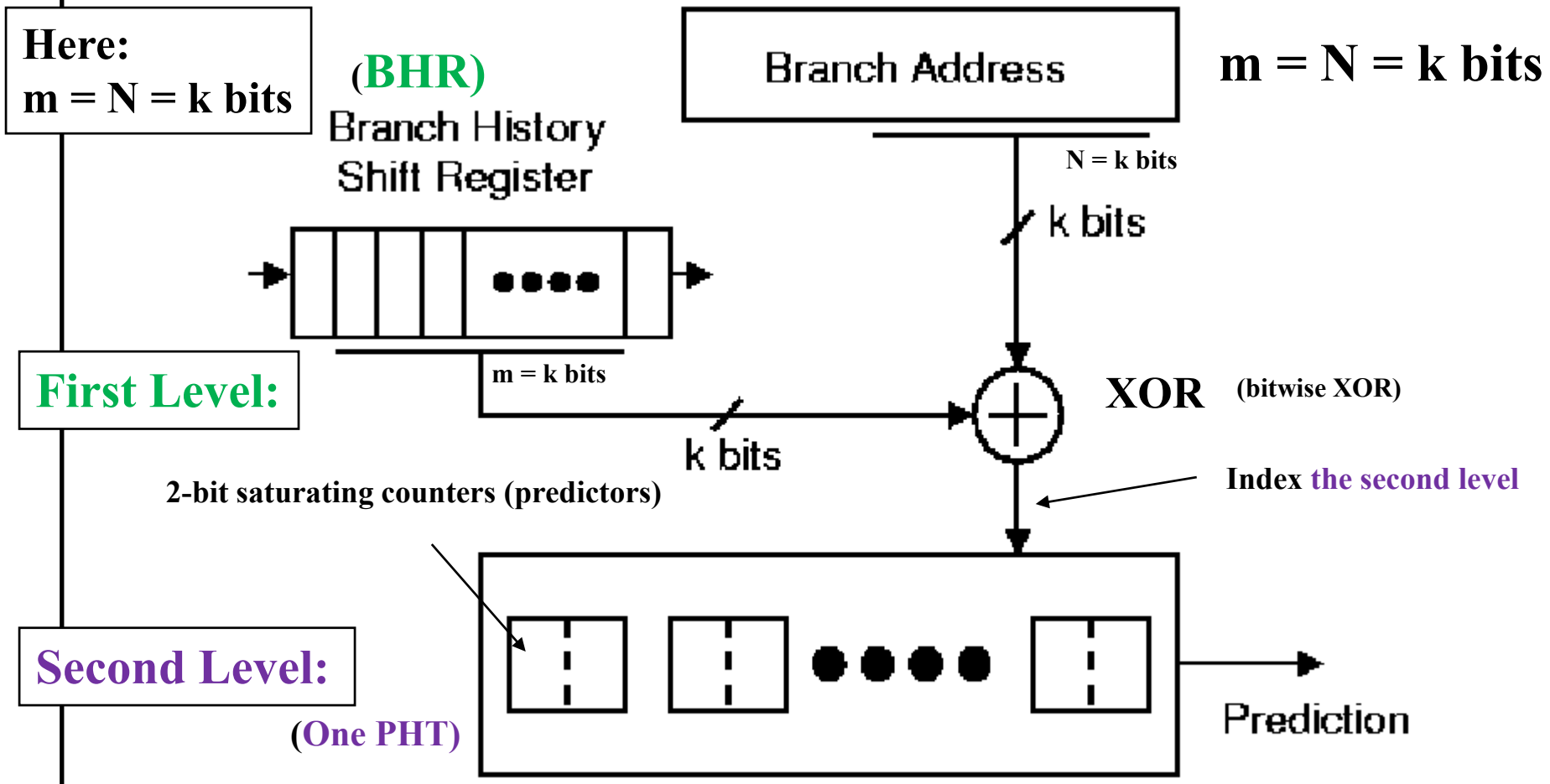
Bitwise XOR

- This mechanism uses less hardware than GAp, since both branch history (first level) and pattern history (second level) are kept globally.
- The hardware cost for k history bits is $k + 2 \times 2^k$ bits, neglecting costs for logic.

gshare is one of the most widely implemented two level dynamic branch prediction schemes

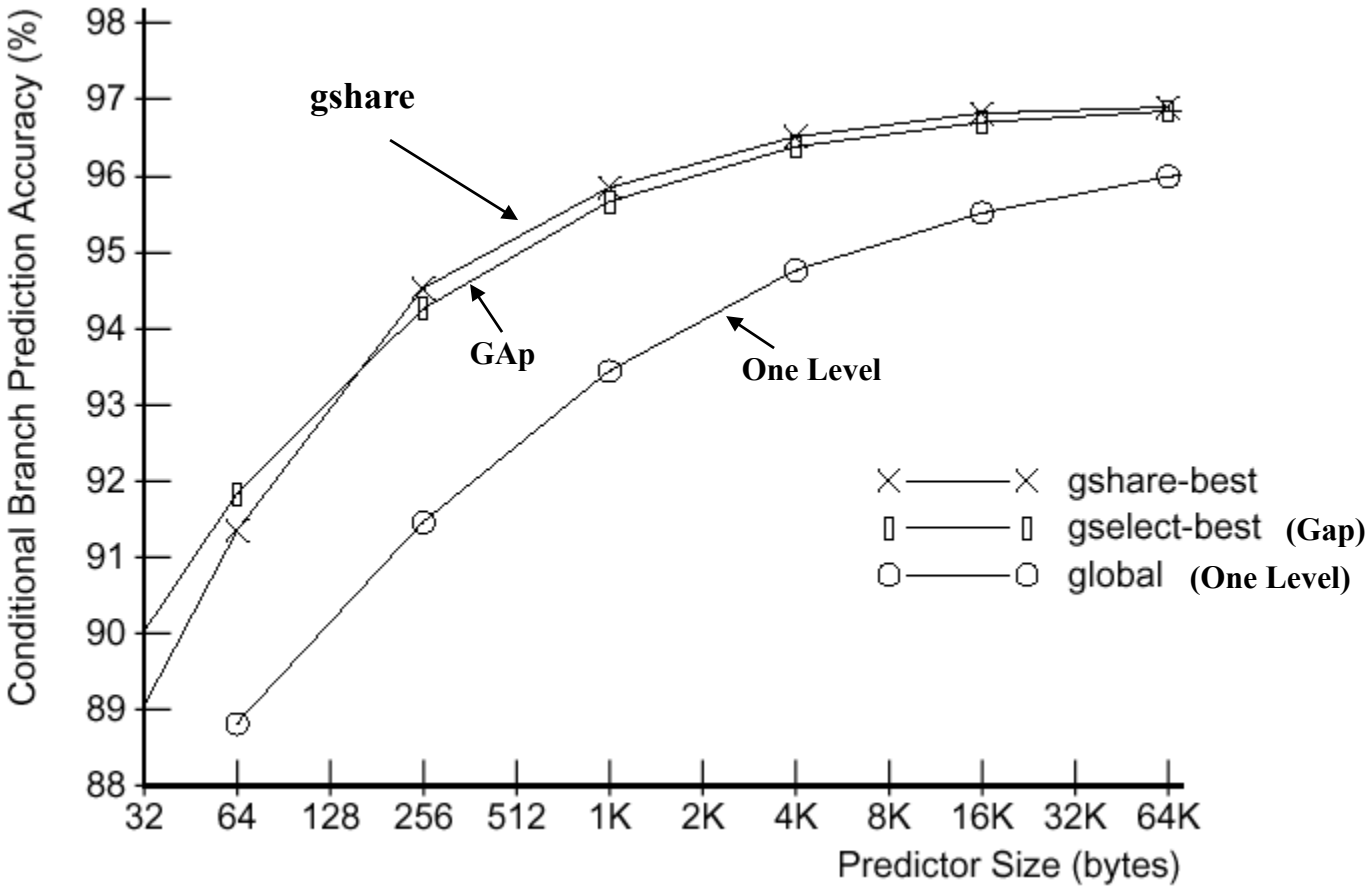
gshare Predictor

Branch and pattern history are kept globally. History and branch address are XORed and the result is used to index the pattern history table.



gshare = global history with index sharing

gshare Performance



Hybrid Dynamic Branch Predictors

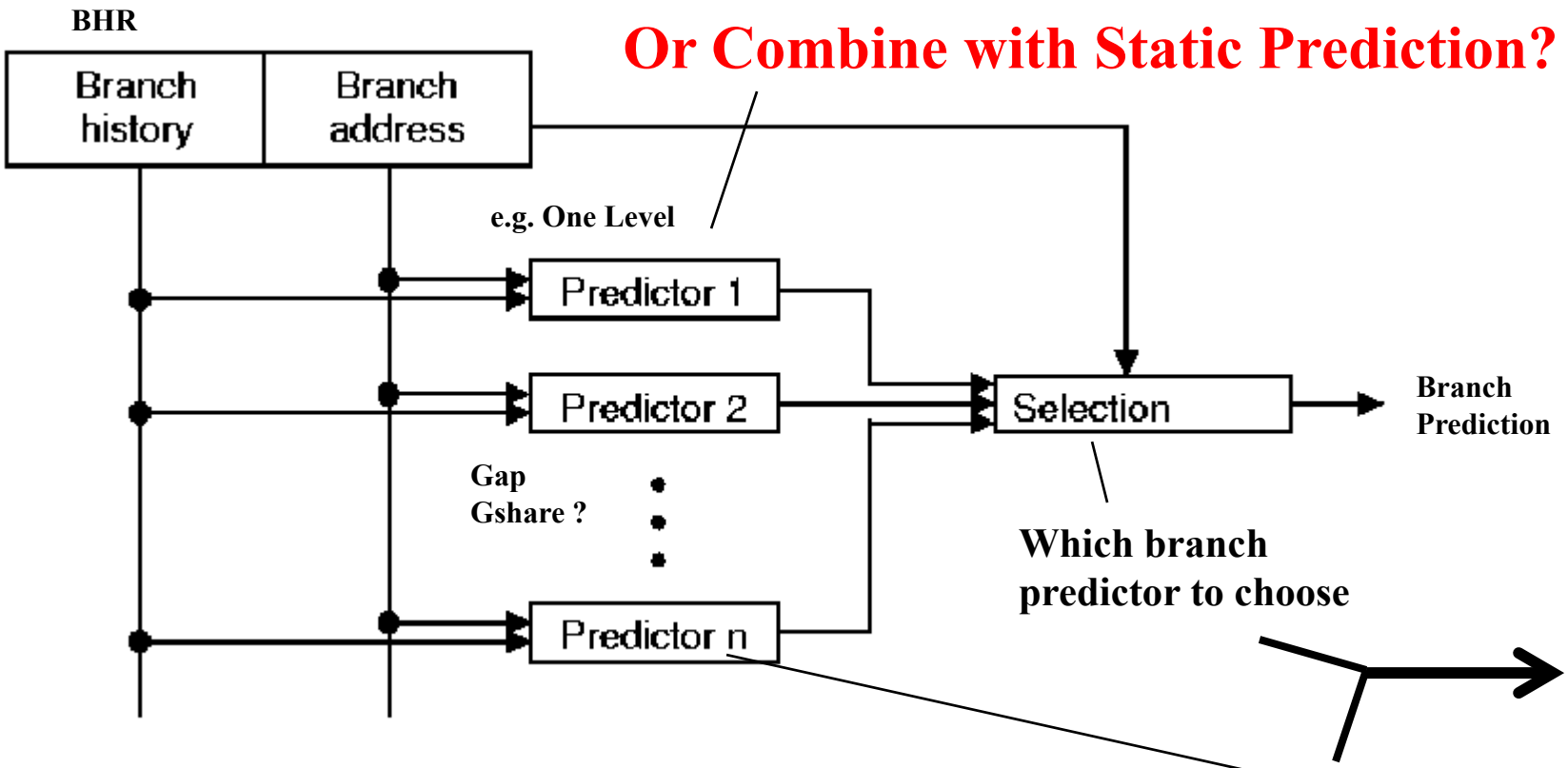
(Also known as tournament or combined predictors)

- Hybrid predictors are simply combinations of two (most common) or more branch prediction mechanisms.
- This approach takes into account that different mechanisms may perform best for different branch scenarios.
- McFarling presented (1993) a number of different combinations of two branch prediction mechanisms.
- He proposed to use an additional 2-bit counter selector array which serves to select the appropriate predictor for each branch.
- One predictor is chosen for the higher two counts, the second one for the lower two counts. The selector array counter used is updated as follows:

Predictor Selector Array Counter Update

1. If the first predictor is wrong and the second one is right the selector counter used counter is decremented,
2. If the first one is right and the second one is wrong, the selector counter used is incremented.
3. No changes are carried out to selector counter used if both predictors are correct or wrong.

A Generic Hybrid Predictor



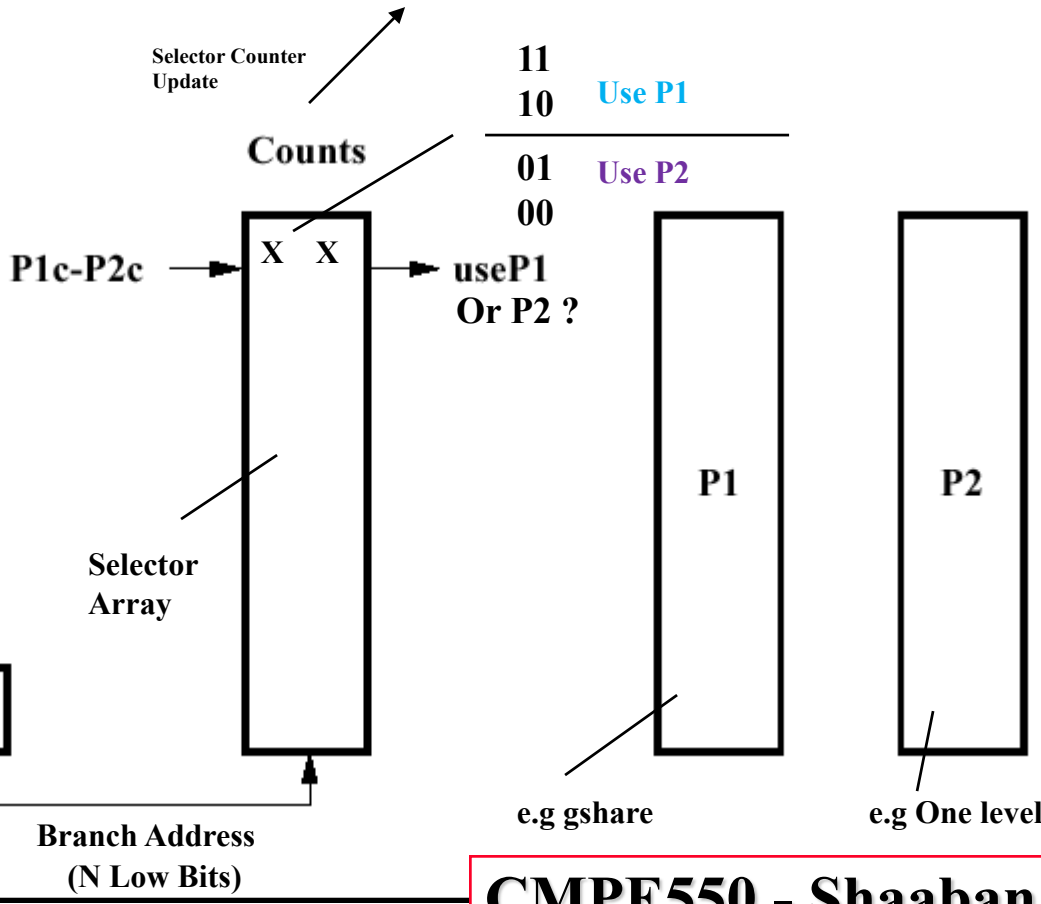
➔ **Usually only two predictors are used (i.e. n =2)**

e.g. As in Alpha, IBM POWER 4 - 9 …

MCFarling's Hybrid Predictor Structure

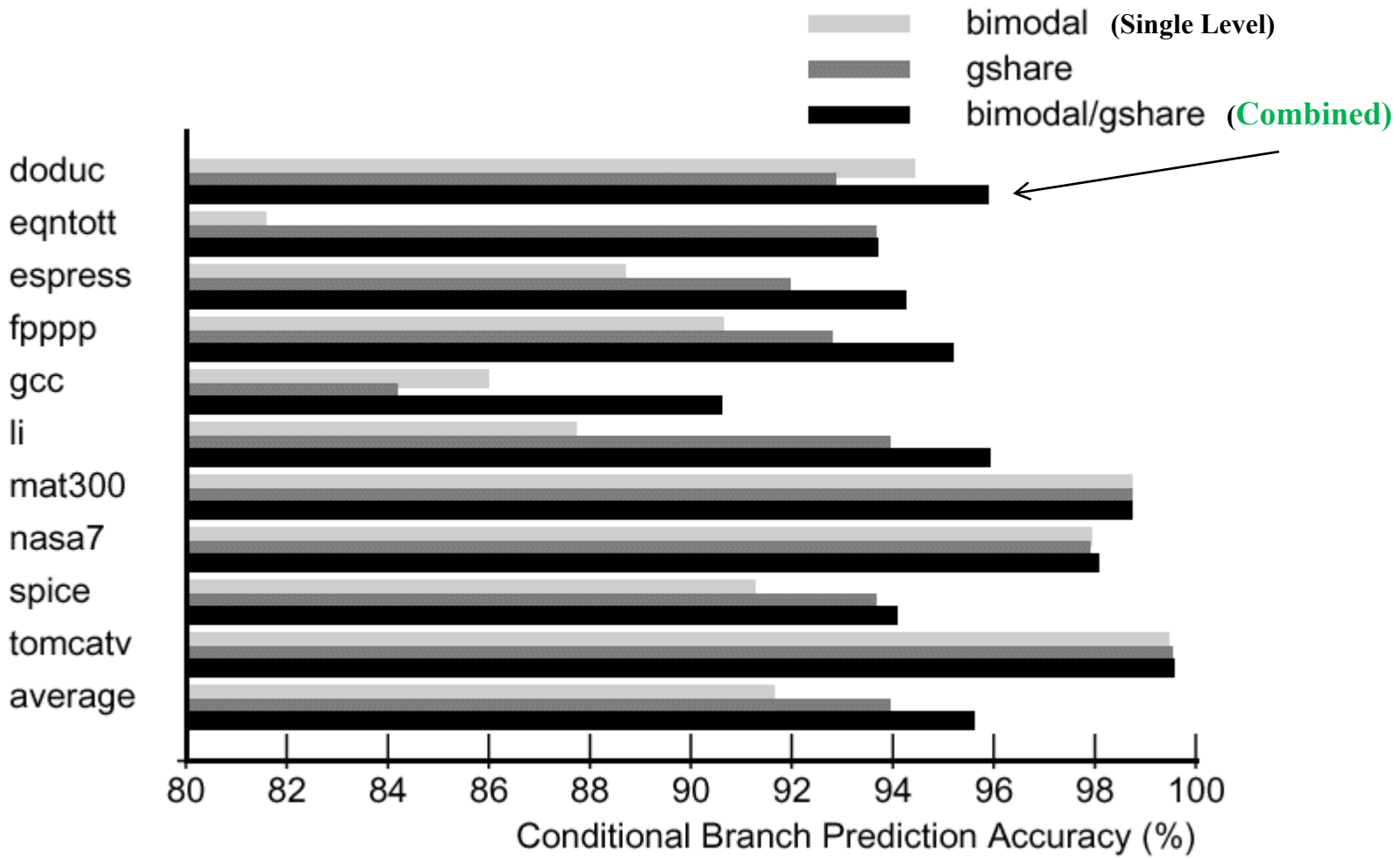
The hybrid predictor contains an additional counter array (selector array) with 2-bit up/down saturating counters. Which serves to select the best predictor to use. Each counter in the selector array keeps track of which predictor is more accurate for the branches that share that counter. Specifically, using the notation P1c and P2c to denote whether predictors P1 and P2 are correct respectively, the selector counter is incremented or decremented by P1c-P2c as shown.

	P1c	P2c	P1c-P2c	
Both wrong	0	0	0	(no change)
P2 correct	0	1	-1	(decrement counter)
P1 correct	1	0	1	(increment counter)
Both correct	1	1	0	(no change)



(Current example implementations: IBM POWER4, 5, 6)

MCFarling's Hybrid Predictor Performance by Benchmark



Processor Branch Prediction Examples

Processor	Released	Accuracy	Prediction Mechanism
Cyrix 6x86	early '96	ca. 85%	PHT associated with BTB
Cyrix 6x86MX	May '97	ca. 90%	PHT associated with BTB
AMD K5	mid '94	80%	PHT associated with I-cache
AMD K6	early '97	95%	2-level adaptive associated with BTIC and ALU
Intel Pentium	late '93	78%	PHT associated with BTB
Intel P6	mid '96	90%	2 level adaptive with BTB
PowerPC750	mid '97	90%	PHT associated with BTIC
MC68060	mid '94	90%	PHT associated with BTIC
DEC Alpha	early '97	95%	Hybrid 2-level adaptive associated with I-cache
HP PA8000	early '96	80%	PHT associated with BTB
SUN UltraSparc	mid '95	88%int 94%FP	PHT associated with I-cache

S+D

S+D

S+D

S+D : Uses both static (ISA supported) and dynamic branch prediction

PHT = One Level