Computer Architecture

Dynamic Hardware Branch Prediction & Branch-Target Buffers

College of Computer Science
Chongqing University

Review: Reducing CPI (or Increasing IPC)

$$CPI = CPI_{ideal} + stalls_{structural} + stalls_{dataHazard} + stalls_{control}$$

technique	reduces
forwarding/ bypassing	potential data-hazard stalls
delayed branches	control-hazard stalls
basic dynamic scheduling (scoreboarding)	data-hazard stalls from true dependencies
dynamic scheduling with register renaming	data-hazard, anti-dep. & output dep. stalls
dynamic branch prediction	control stalls
issuing multiple instruction per clock cycle	ideal CPI
speculation	data-hazard and control-hazard stalls
dynamic memory disambiguation	data-hazard stalls with memory
loop unrolling	control hazard stalls
basic compiler pipeline scheduling	data-hazard stalls
compiler dependency analysis	ideal CPI, data-hazard stalls
software pipelining & trace scheduling	ideal CPI, data-hazard stalls
compiler speculation	ideal CPI, data-hazard stalls

Reducing Branch Penalty

Branch penalty : wasted cycles due to pipeline flushing on mis-predicted branches

Reduce branch penalty:

1. Predict branch/jump instructions AND branch direction (taken or not taken)
2. Predict branch/jump target address (for taken branches)
3. Speculatively execute instructions along the predicted path

Branch Prediction Strategies

- Static
  - Decided before runtime
  - Examples:
    - Always-Not Taken
    - Always-Taken
    - Backwards Taken, Forward Not Taken (BTFNT？)
    - Profile-driven prediction
- Dynamic
  - Prediction decisions may change during the execution of the program

Example: Branch Penalty (in CPI units) on the MIPS R4000 Processor

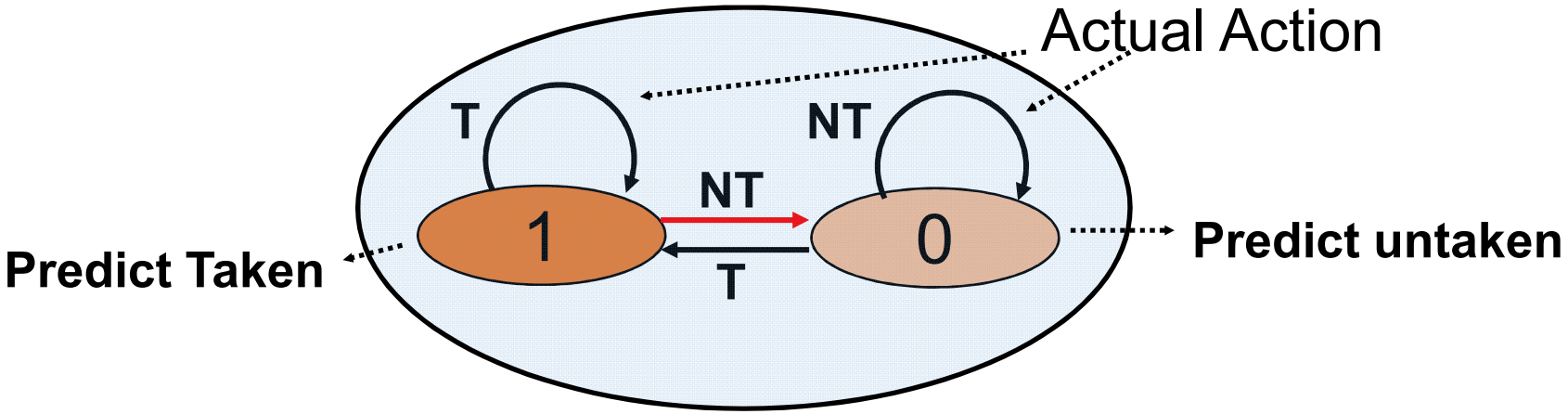
Branch scheme	Penalty for jumps	Penalty for untaken branches	Penalty for taken branches
Flush Pipeline	2	3	3
Predict Taken	2	3	2
Predict Untaken	2	0	3

MIPS R4000 takes 3 cycles to compute the target address (i.e. minimum 2 CPI of *additional* penalty) and 4 cycles to know the branch outcome

Strategy	Jumps	Untaken branches	Taken branches	Combined
Frequency (est.)	4%	6%	10%	20%
Flush Pipeline	0.08	0.18	0.30	0.56
Predict Taken	0.08	0.18	0.20	0.46
Predict Untaken	0.08	0.00	0.30	0.38

A Simple Scheme: 1-bit Predictor for a Single Branch

1-bit prediction



Basic Branch Prediction: Branch-History Table (or Branch-Prediction Buffer)

- Implemented as a small memory indexed by a portion (usually some low-significant bits) of the address of the branch instruction.
  - So the size of this table is: the number of entries * the number of bits per entry.
  - If unfortunately, the address portions of two branch instructions are identical, they share one entry. Hence, the more entries, the less such conflicts.

e.g., branch instructions at the following addresses share the BHT entry

- 00F2BC 0101 1100
- 010A5D 1001 1100

Index	Taken?
0000	1
0001	0
0010	1
0011	0
0100	1
0101	0
0110	1
0111	1
1000	1
1001	0
1010	1
1011	1
1100	0
1101	1
1110	1
1111	1

1-bit Predictor: weakness on nested loops

- Consider a nested loop :

```
for (…){
   for (i=0; i<9; i++)
      a[i] = a[i] * 2.0;
}
```

  - Mispredict twice, once on entry, once on exit, every time when the inner loop is executed.

Example: Accuracy of 1-Bit Prediction Scheme in the Presence of "Loop Branching"

```
loop: L.D F0, 0(R1)
      MUL.D F4, F0, F2
      S.D F4, 0(R1)
      DADDIU R1, R1, #-8
      BNEZ R1 loop
```

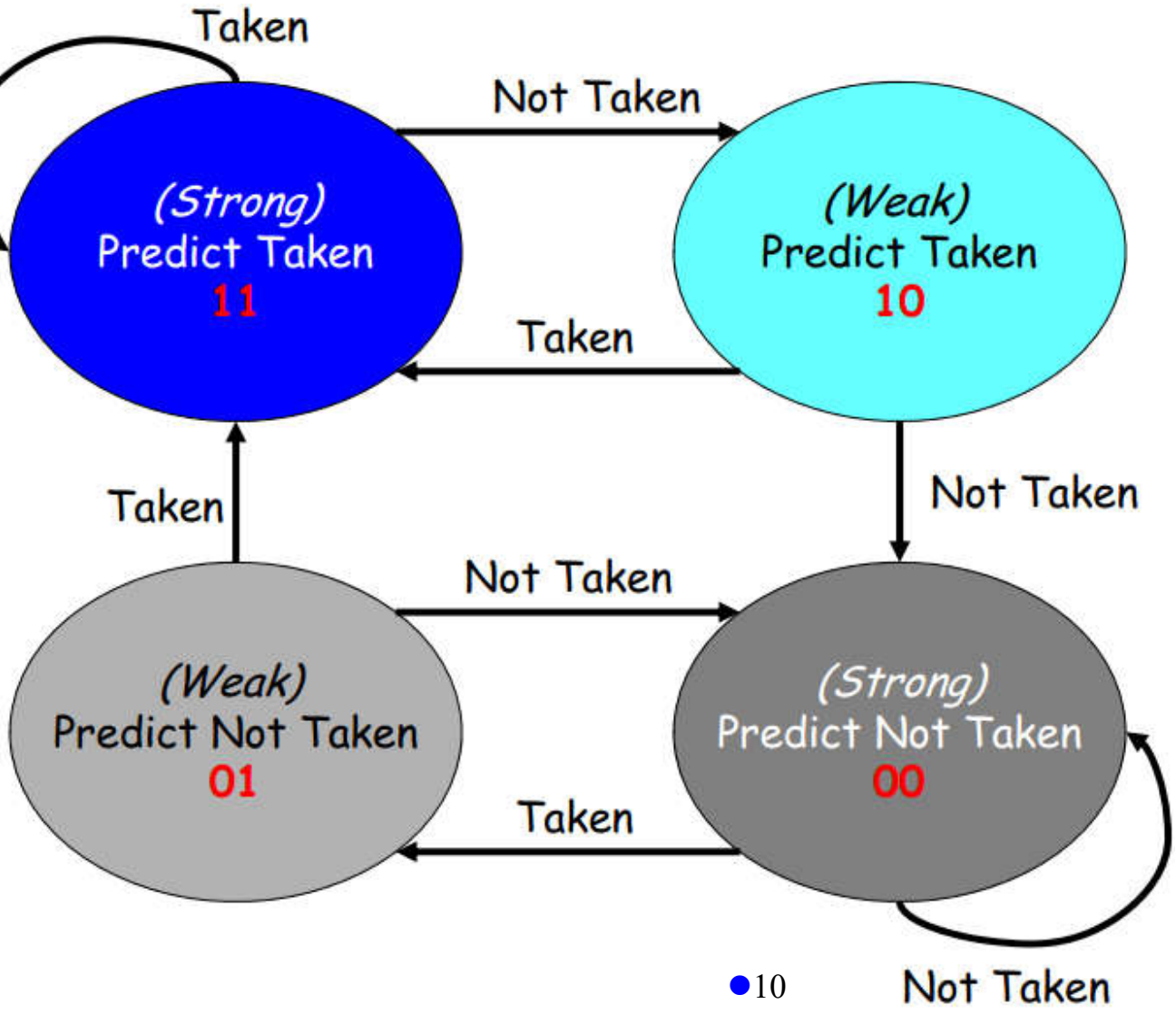
- Assumptions
  - R1 is initialized to #80

Iteration	0	1	2	3	4	5	6	7	8	9	10	0	1	2	3	4	5	6	7	8	9	10
Predicted behavior	NT	T	T	T	T	T	T	T	T	T	T	NT	T	T	T	T	T	T	T	T	T	T
Actual behavior	T	T	T	T	T	T	T	T	T	T	NT	T	T	T	T	T	T	T	T	T	T	NT

- Branch taken 90% of the times, but branch prediction accuracy is only 80%
- A 1-bit predictor for "loop branches" mispredicts at twice the rate that the branch is not taken

2-Bit Prediction Scheme (a.k.a. Bimodal Predictor)

- Adding *hysteresis* to the prediction scheme
  - a prediction must be missed twice before it is changed
- Implementation
  - a *(2-bit) saturated counter* that is incremented on a taken branch and decremented on an untaken branch



Example: 2-Bit Prediction Scheme in Action with "Loop Branching"

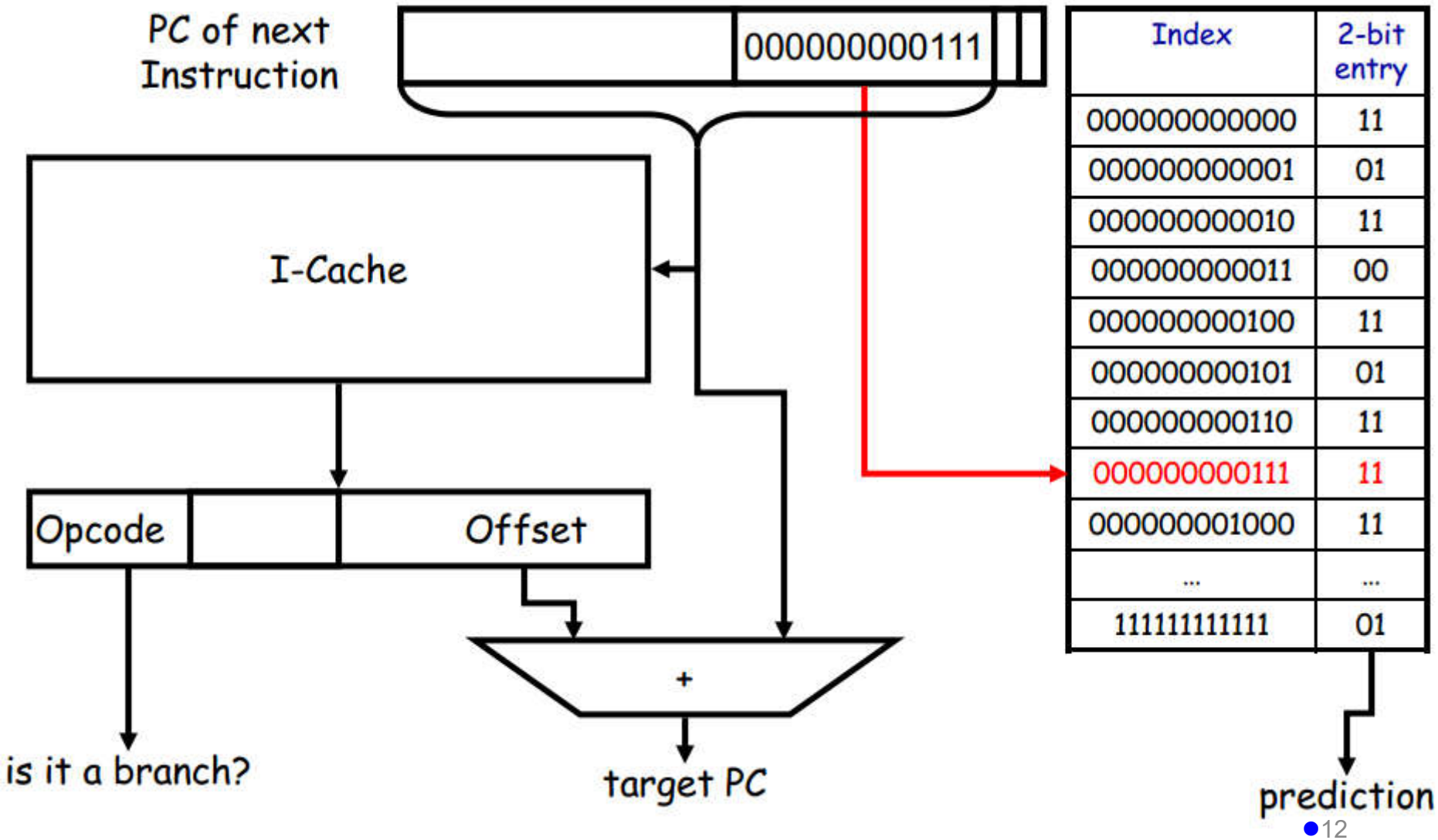
```
loop: L.D F0, 0(R1)
      MUL.D F4, F0, F2
      S.D F4, 0(R1)
      DADDIU R1, R1, #-8
      BNEZ R1 loop
```

- Assumptions
  - R1 is initialized to #80

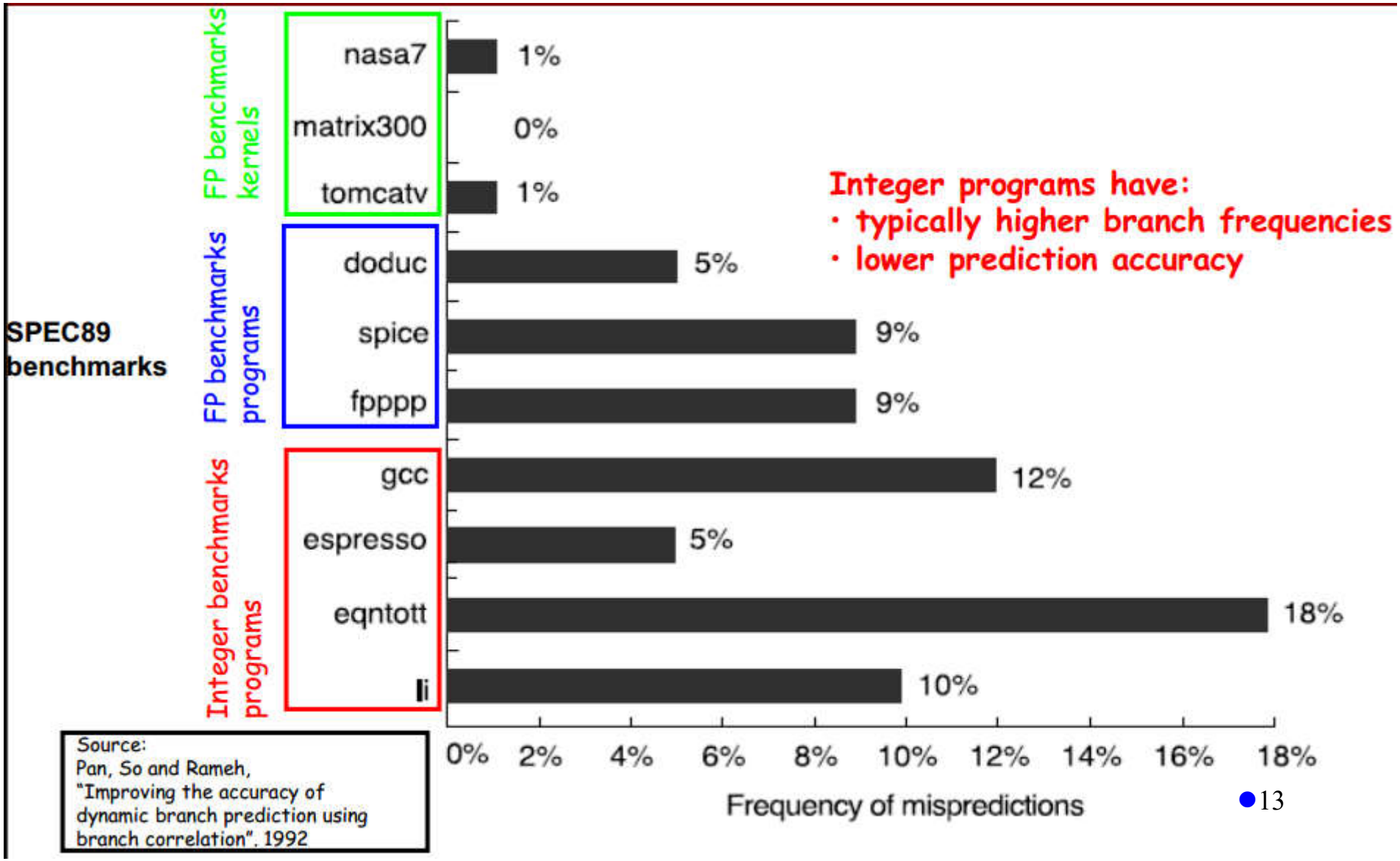
Iterationss & steps	0	1	2	3	4	5	6	7	8	9	10	0	1	2	3	4	5	6	7	8	9	10
Predicted behavior	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T
Actual behavior	T	T	T	T	T	T	T	T	T	T	NT	T	T	T	T	T	T	T	T	T	T	NT

- Branch taken 90% of the times, and branch prediction accuracy is now 90%
- The 2-bit predictor mispredicts at the 10th step of the 1st iteration, but *doesn't change its mind*. It just moves to "weak-predict-taken" for the 1st step of the following iteration

Branch History Table for 4K-Entry 2-Bit Predictor: 4K Entry * 2 bits/Entry = 8K bits



Measure: Prediction Accuracy of 4K-entry 2-bit-prediction buffer with SPEC89

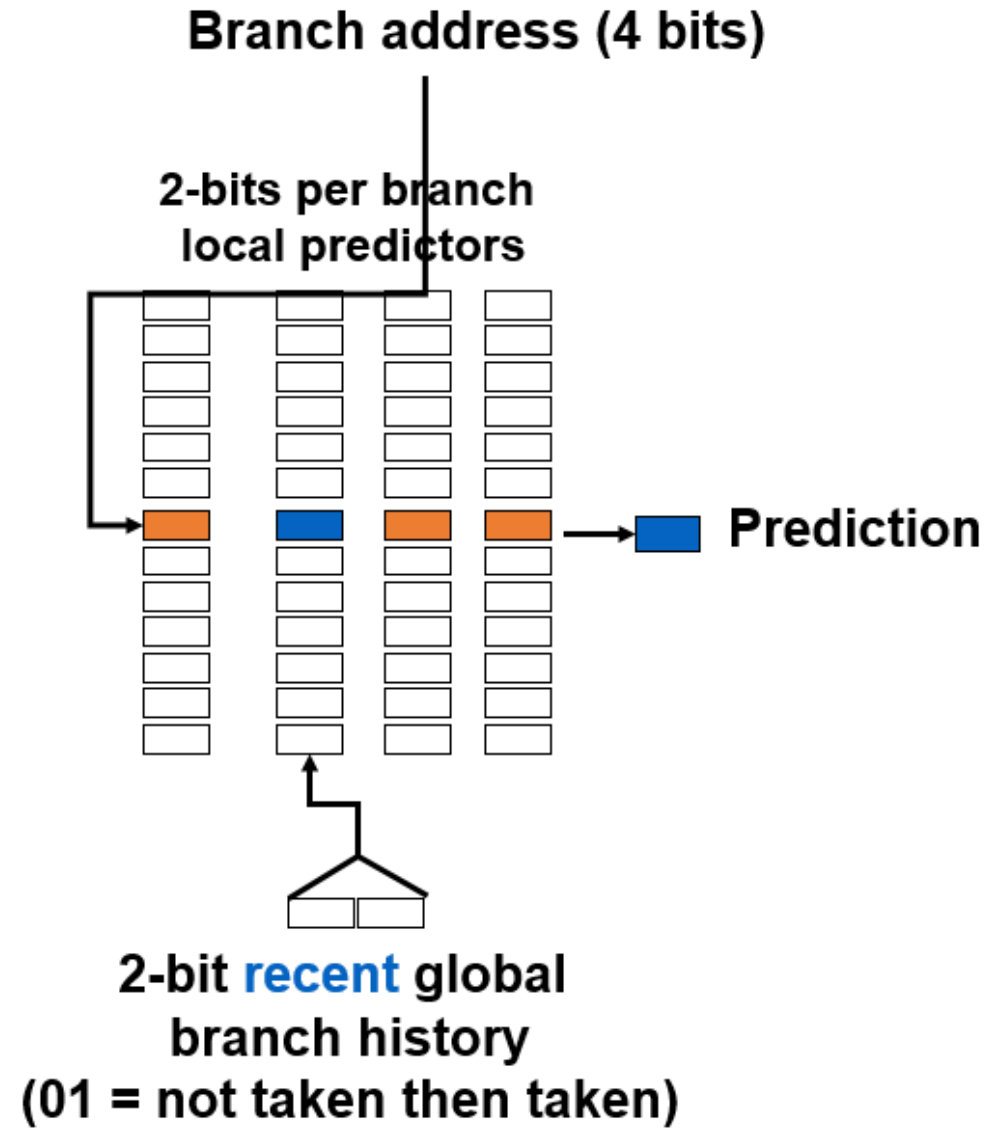


Correlating Branch Predictors

```
if (aa==2)
        aa=0;
if (bb==2)
        bb=0;
if (aa!=bb) {
```

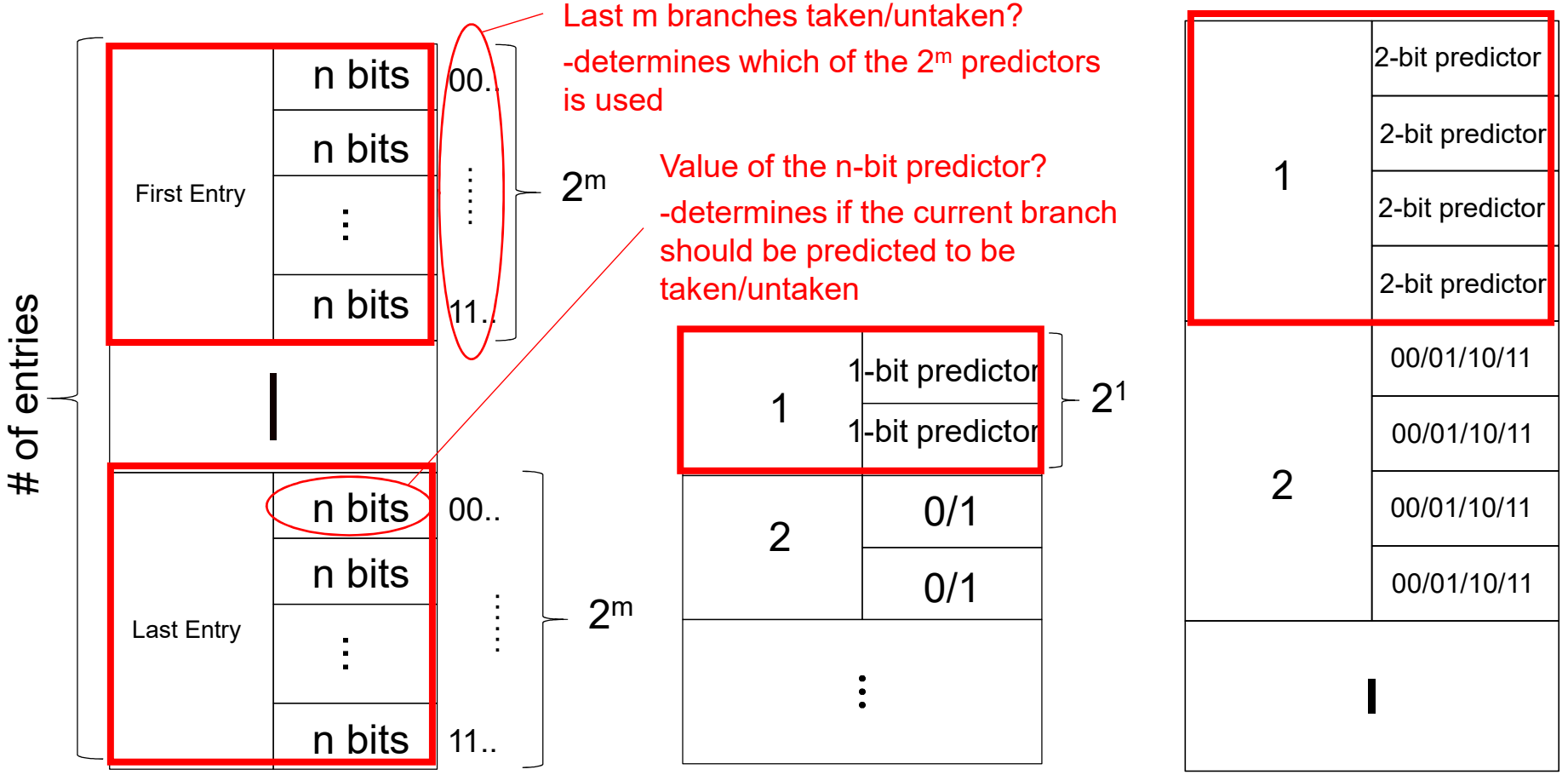
- The limitation of the basic 2-bit predictor:
  - The 2-bit predictor schemes use only the recent behavior of **a single branch** to predict the future behavior of that branch. Increasing to 3-bit or more does not help much!
  - How about looking at the recent behavior of other branches?
  - Look at the code in the top-right corner: if both of the first two branches are not taken (aa=0 && bb=0), the 3rd is always taken (as always aa=bb).
- Correlating predictor / 2-level predictor:
  - Adds information of the most recent branches to decide how to predict a given branch.
  - An (m,n) 2-level predictor uses the behavior of the last m branches to choose from 2^m branch predictors, each of which is an n-bit predictor for a single branch.
  - The size of BHT = # of Entries * # of bits / Entry = # of Entries * 2^m * n

硬件实现非常容易



(2,2) predictor: 2-bit global, 2-bit local

Correlating Branch Predictors



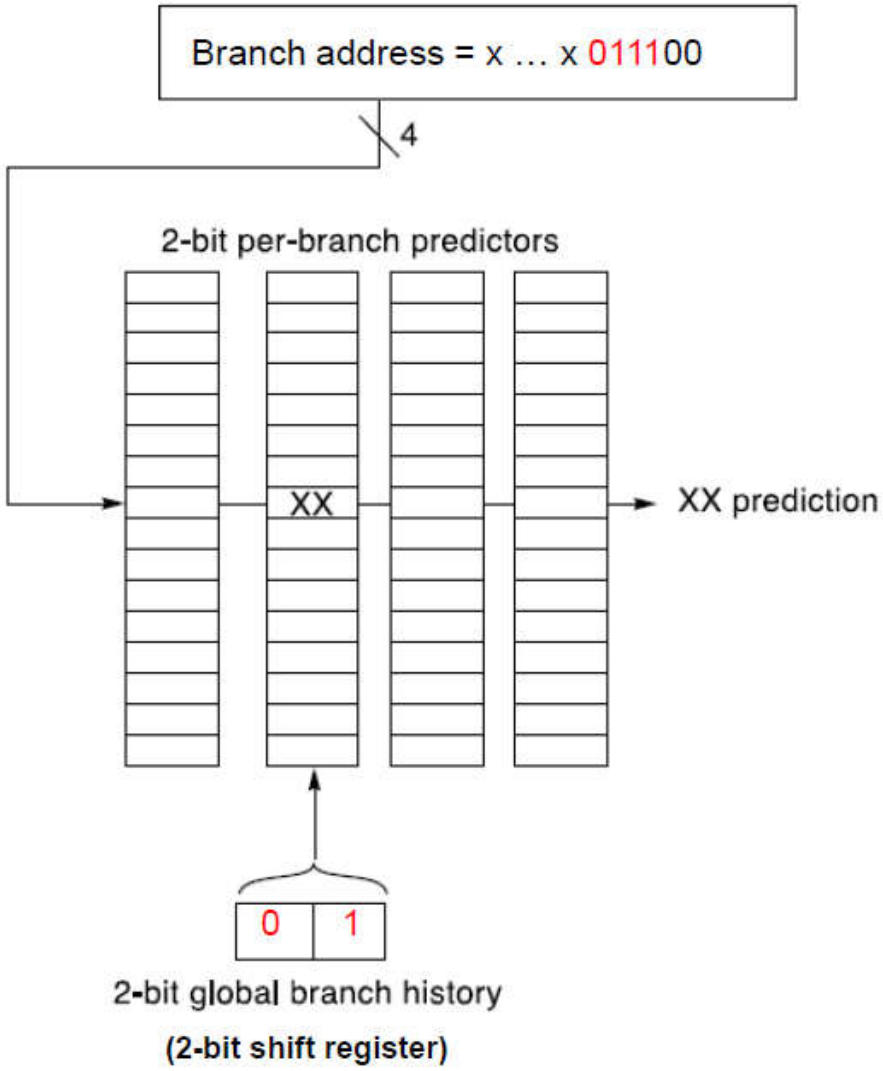
(m,n) correlating predictor BHT

(1,1) correlating predictor BHT

(2,2) correlating predictor BHT

Example:
A 64-Entry (2,2) Branch Prediction Buffer

- Global history of the most recent m=2 branches is recorded in an 2-bit shift register where each bit records whether the branch was taken or not taken
- A concatenation of the low-order bits of the branch instruction address and the 2-bit global history is used to index the buffer and get the (n=2)-bit predictor
- Concept can be generalized to (m,n) predictor



Adding "Global" Information to Branch Prediction: Correlating (or Two-Level) Branch Predictors

```
if (a == 2)
  a = 0;
…
if (b == 2)
  b = 0;
…
if (a != b) {
  …
```

```
      DSUBUI R3, R1, #2
b1 →  BNEZ R3, L1
      DADD R1, R0, R0
L1:   DSUBUI R3, R2, #2
b2 →  BNEZ R3, L2
      DADD R2, R0, R0
L2:   DSUBU R3, R1, R2
b3 →  BEQZ R3, L3
```

- Prediction accuracy can be improved by accounting for branch spatial correlations and looking at the behavior of other branches
  - e.g. in the example above, if the first two branches (b1 and b2) are not taken then the third (b3) will be taken

A Simple Example: Set-Up

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

```
b1 → BNEZ R1, L1
     DADDIU R1, R0, #1
L1:  DSUBUI R3, R1, #-1
b2 → BNEZ R3, L2
     …
L2:  …
```

init d	d==0?	b1	d before b2	d==1?	b2
0	yes	not taken	1	yes	not taken
1	no	taken	1	yes	not taken
x ≠{0,1}	no	taken	x ≠ {0,1}	no	taken

Correlation: if b1 is not taken then d is set to 1 and b2 is also not taken

A Simple Example: 1-Bit Predictor

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

Assumption: d alternates between 2 and 0

```
b1 ──→ BNEZ R1, L1
       DADDIU R1, R0, #1
L1:    DSUBUI R3, R1, #-1
b2 ──→ BNEZ R3, L2
       …
L2:    …
```

d=?	b1 pred.	b1 action	b1 new pred.	b2 pred.	b2 action	b2 new pred.
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

A 1-bit predictor that is initialized to not-taken mispredicts all branches

A Simple Example: 1-Bit Predictor with 1-Bit Correlation {i.e., a (1,1) Predictor}

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

```
b1 → BNEZ R1, L1
     DADDIU R1, R0, #1
L1:  DSUBUI R3, R1, #-1
b2 → BNEZ R3, L2
     …
L2:  …
```

Assumption: d alternates between 2 and 0

X/Y: use X if last branch was not taken, use Y if last branch was taken

d=?	b1 pred.	b1 action	b1 new pred.	b2 pred.	b2 action	b2 new pred.
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

A (1,1) predictor initialized to NT/NT mispredicts only at the first iter.

A Simple Example of (1,1) Predictor: Simulation - Cycle 0

```
     BNEZ R1, L1
     DADDIU R1, R0, #1
L1:  DSUBUI R3, R1, #-1
     BNEZ R3, L2
     ...
L2:  ...
```

b1

b2

Assumption: d alternates between 2 and 0

Initial Status

b[redacted]	NT 0	b1's last branch untaken
	NT 1	b1's last branch taken
b[redacted]	NT 0	b2's last branch untaken
	NT 1	b2's last branch taken

d=?	b1 pred.	b1 action	b1 new pred.	b2 pred.	b2 action	b2 new pred.
2	NT/NT	T		NT/NT	T	
0		NT			NT	
2		T			T	
0		NT			NT	

A (1,1) predictor initialized to NT/NT mispredicts only at the first iter.

A Simple Example of (1,1) Predictor: Simulation - Cycle 1

```
      BNEZ R1, L1
      DADDIU R1, R0, #1
L1:   DSUBUI R3, R1, #-1
      BNEZ R3, L2
      …
L2:   …
```

Assumption: d alternates between 2 and 0
X/Y: use X if last branch was not taken,
use Y if last branch was taken

Initial Status

b1	NT 0	b1's last branch untaken
	NT 1	b1's last branch taken
b2	NT 0	b2's last branch untaken
	NT 1	b2's last branch taken

d=?	b1 pred.	b1 action	b1 new pred.	b2 pred.	b2 action	b2 new pred.
2	NT/NT	T	T/NT	NT/**NT**	T	
0	T/NT	NT			NT	
2		T			T	
0		NT			NT	

A (1,1) predictor initialized to NT/NT mispredicts only at the first iter.

A Simple Example of (1,1) Predictor: Simulation - Cycle 2

```
     BNEZ R1, L1
     DADDIU R1, R0, #1
L1:  DSUBUI R3, R1, #-1
     BNEZ R3, L2
     …
L2:  …
```

Assumption: d alternates between 2 and 0
X/Y: use X if last branch was not taken,
use Y if last branch was taken

Initial Status

b[redacted]	NT	0	b1's last branch untaken
	NT	1	b1's last branch taken
b[redacted]	NT	0	b2's last branch untaken
	NT	1	b2's last branch taken

d=?	b1 pred.	b1 action	b1 new pred.	b2 pred.	b2 action	b2 new pred.
2	NT/NT	T	T/NT	NT/NT	T	NT/T
0	T/NT	NT		NT/T	NT	
2		T			T	
0		NT			NT	

A (1,1) predictor initialized to NT/NT mispredicts only at the first iter.

A Simple Example of (1,1) Predictor: Simulation - Cycle 3

```
     BNEZ R1, L1
     DADDIU R1, R0, #1
L1:  DSUBUI R3, R1, #-1
     BNEZ R3, L2
     …
L2:  …
```

Assumption: d alternates between 2 and 0
X/Y: use X if last branch was not taken,
use Y if last branch was taken

Initial Status

b[redacted]	NT 0	b1's last branch untaken
	NT 1	b1's last branch taken
b[redacted]	NT 0	b2's last branch untaken
	NT 1	b2's last branch taken

d=?	b1 pred.	b1 action	b1 new pred.	b2 pred.	b2 action	b2 new pred.
2	NT/NT	T	T/NT	NT/NT	T	NT/T
0	T/NT	NT	T/NT	NT/T	NT	
2	T/NT	T			T	
0		NT			NT	

A (1,1) predictor initialized to NT/NT mispredicts only at the first iter.

A Simple Example of (1,1) Predictor: Simulation - Cycle 4

```
     BNEZ R1, L1
     DADDIU R1, R0, #1
L1:  DSUBUI R3, R1, #-1
     BNEZ R3, L2
     …
L2:  …
```

Assumption: d alternates between 2 and 0
X/Y: use X if last branch was not taken,
use Y if last branch was taken

Initial Status

b1	NT 0	b1's last branch untaken
	NT 1	b1's last branch taken
b2	NT 0	b2's last branch untaken
	NT 1	b2's last branch taken

d=?	b1 pred.	b1 action	b1 new pred.	b2 pred.	b2 action	b2 new pred.
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		NT/T	T	
0		NT			NT	

A (1,1) predictor initialized to NT/NT mispredicts only at the first iter.

A Simple Example of (1,1) Predictor: Simulation - Cycle 5

```
      BNEZ R1, L1
      DADDIU R1, R0, #1
L1:   DSUBUI R3, R1, #-1
      BNEZ R3, L2
      …
L2:   …
```

Assumption: d alternates between 2 and 0
X/Y: use X if last branch was not taken,
use Y if last branch was taken

Initial Status

b1	NT 0	b1's last branch untaken
	NT 1	b1's last branch taken
b2	NT 0	b2's last branch untaken
	NT 1	b2's last branch taken

d=?	b1 pred.	b1 action	b1 new pred.	b2 pred.	b2 action	b2 new pred.
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	
0	T/NT	NT			NT	

A (1,1) predictor initialized to NT/NT mispredicts only at the first iter.

A Simple Example of (1,1) Predictor: Simulation - Cycle 6

```
     BNEZ R1, L1
     DADDIU R1, R0, #1
L1:  DSUBUI R3, R1, #-1
     BNEZ R3, L2
     …
L2:  …
```

Assumption: d alternates between 2 and 0
X/Y: use X if last branch was not taken,
use Y if last branch was taken

Initial Status

b1	NT 0	b1's last branch untaken
	NT 1	b1's last branch taken
b2	NT 0	b2's last branch untaken
	NT 1	b2's last branch taken

d=?	b1 pred.	b1 action	b1 new pred.	b2 pred.	b2 action	b2 new pred.
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		NT/T	NT	

A (1,1) predictor initialized to NT/NT mispredicts only at the first iter.

A Simple Example of (1,1) Predictor: Simulation - Cycle 7

```
     BNEZ R1, L1
     DADDIU R1, R0, #1
L1:  DSUBUI R3, R1, #-1
     BNEZ R3, L2
     …
L2:  …
```

Assumption: d alternates between 2 and 0
X/Y: use X if last branch was not taken,
use Y if last branch was taken

Initial Status

b[redacted]	NT 0	b1's last branch untaken
	NT 1	b1's last branch taken
b[redacted]	NT 0	b2's last branch untaken
	NT 1	b2's last branch taken

d=?	b1 pred.	b1 action	b1 new pred.	b2 pred.	b2 action	b2 new pred.
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	

A (1,1) predictor initialized to NT/NT mispredicts only at the first iter.

A Simple Example of (1,1) Predictor: Simulation - Cycle 8

```
     BNEZ R1, L1
     DADDIU R1, R0, #1
L1:  DSUBUI R3, R1, #-1
     BNEZ R3, L2
     …
L2:  …
```

Assumption: d alternates between 2 and 0
X/Y: use X if last branch was not taken,
use Y if last branch was taken

Initial Status

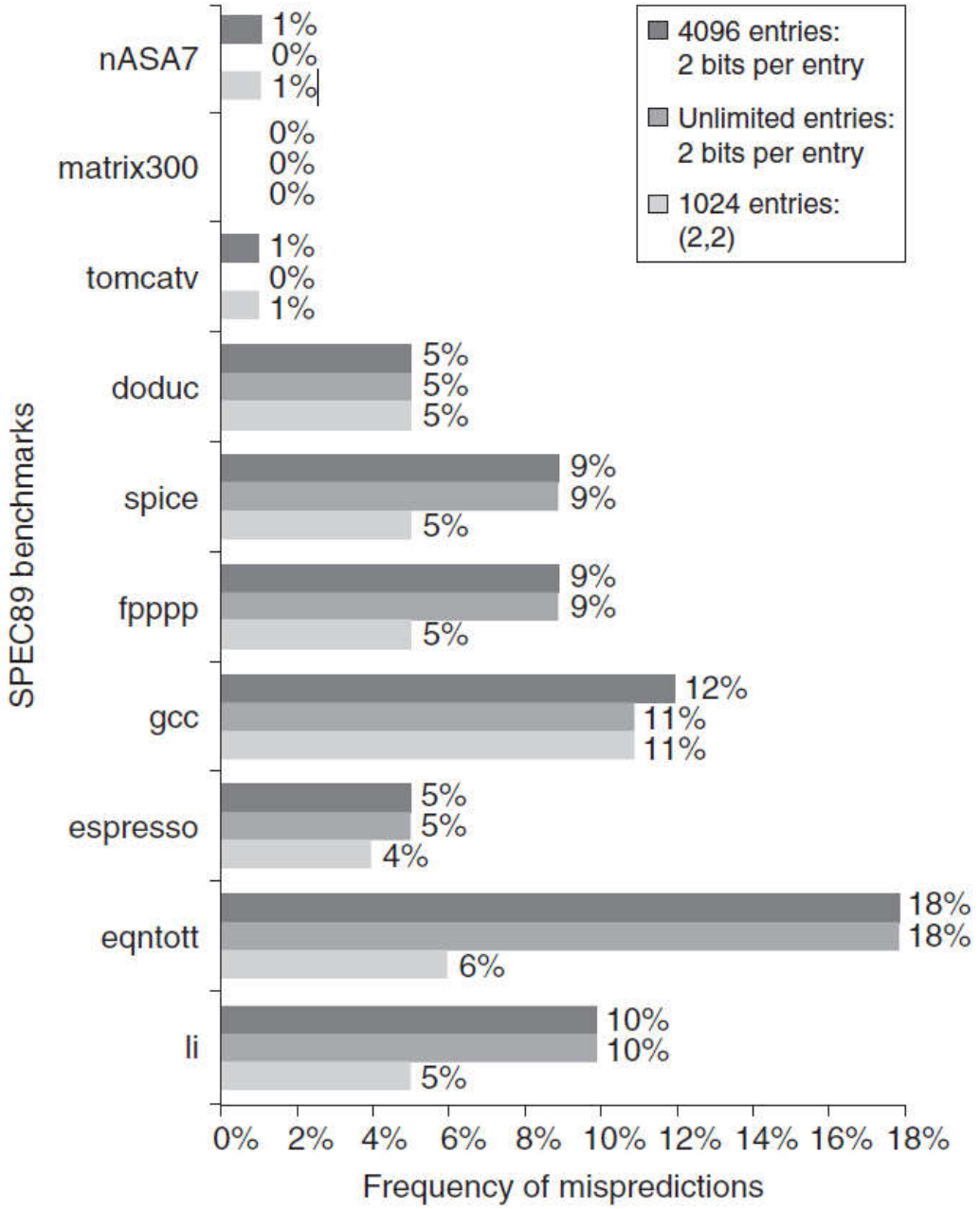
b1	NT 0	b1's last branch untaken
	NT 1	b1's last branch taken
b2	NT 0	b2's last branch untaken
	NT 1	b2's last branch taken

d=?	b1 pred.	b1 action	b1 new pred.	b2 pred.	b2 action	b2 new pred.
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

A (1,1) predictor initialized to NT/NT mispredicts only at the first iter.

Compare 2-bit vs Correlating

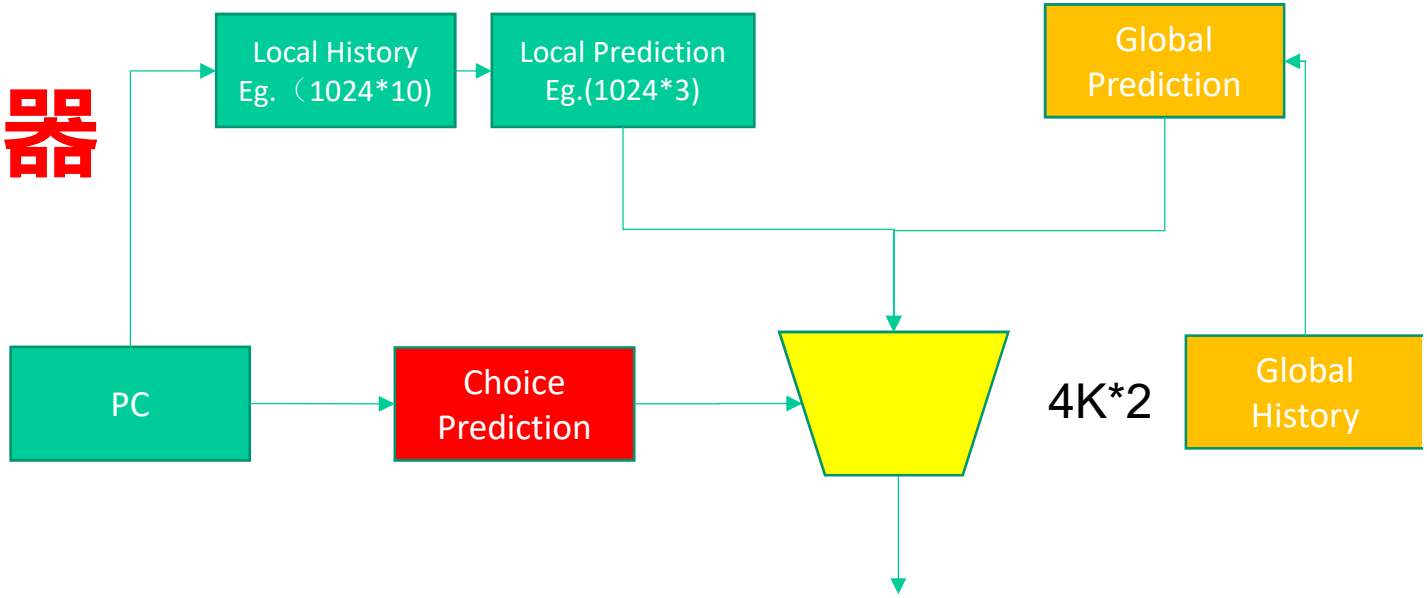
- Three are in comparison:
  - 2-bit Predictor, (0,2)
  - (2,2) Correlating
  - 2-bit Predictor with unlimited entries.
- The first two have the same total number of bits in their BHT.



Tournament Predictors: Adaptively Combining Local and Global Predictors

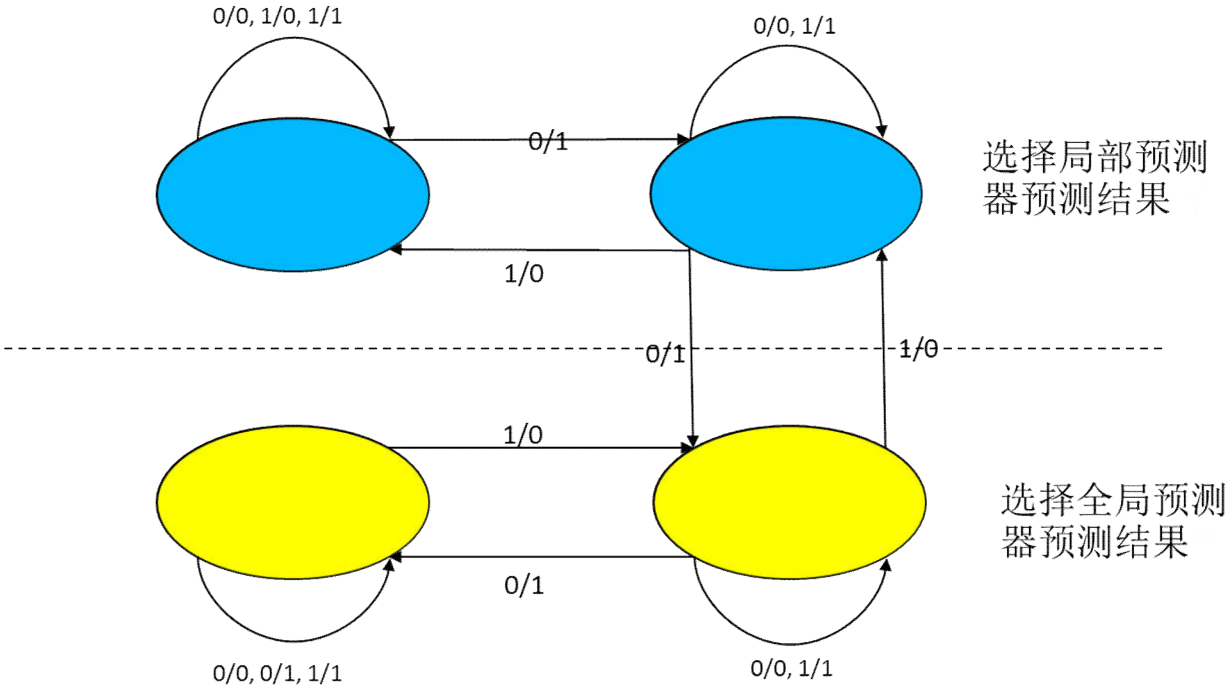
- 2-bit predictor uses only local information
- Correlating adds global information, so better.
- *Tournament predictors* take this insight to the next level, by using multiple predictors. Even better!
  - one based on global information
  - one based on local information
  - combine them with a selector.
- The selector is a 2-bit saturating counter, one per branch. Selector chooses among the two predictors based on their recent performance

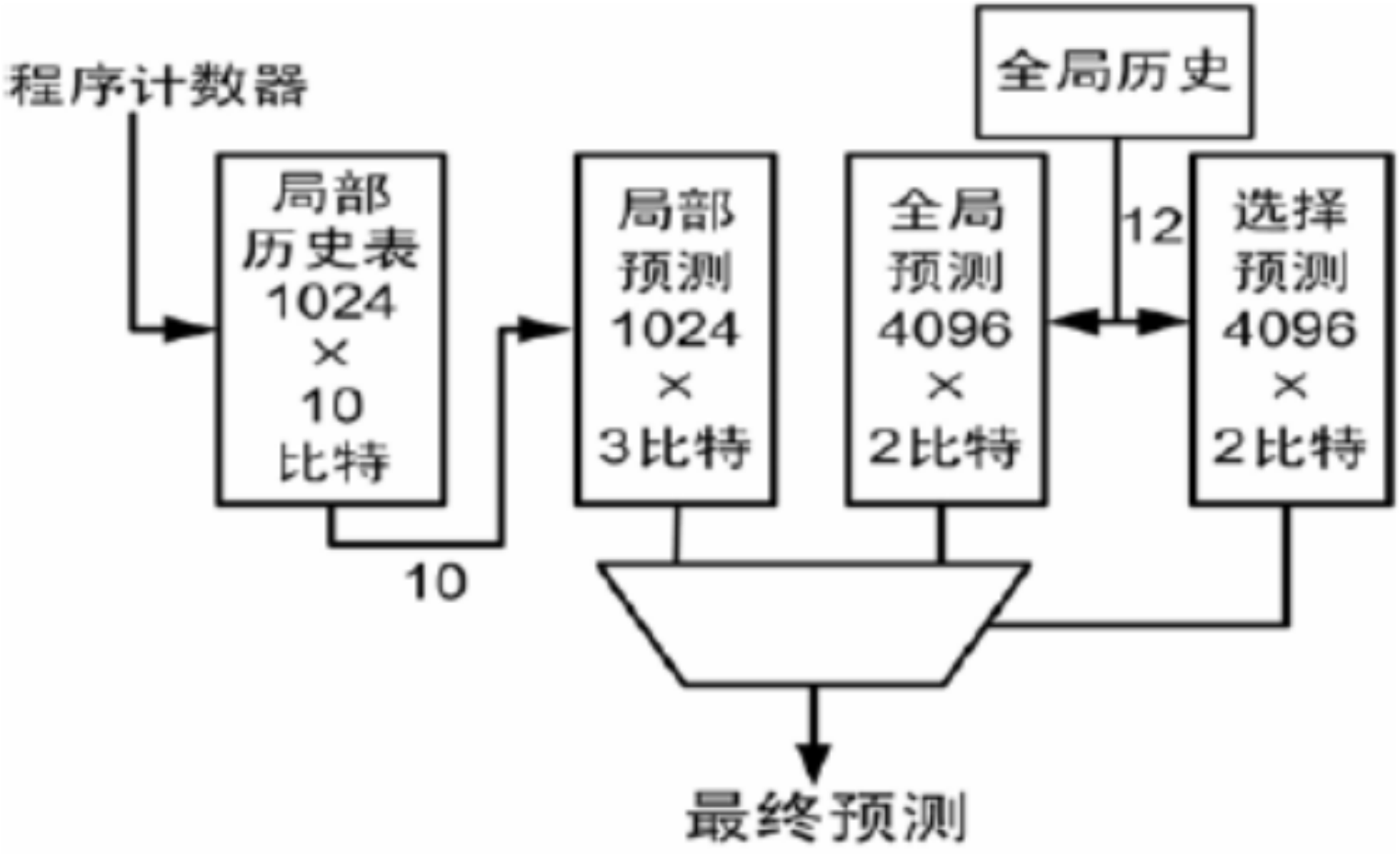
竞赛预测器



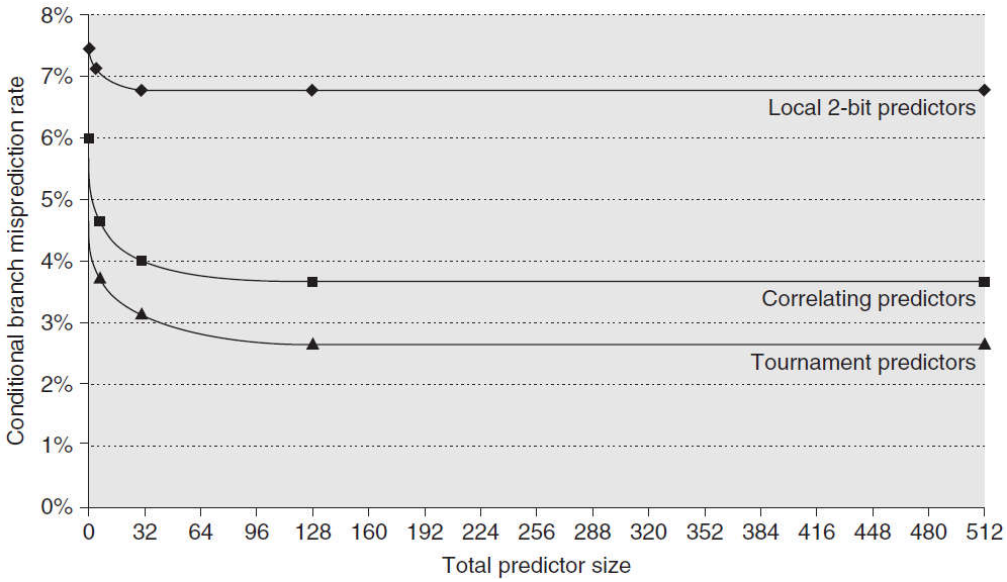
- **全局预测器**
  - 使用最近n（n=12）次分支跳转情况来索引，即全局预测器入口数：2^n每个入口是一个标准的2位预测器
- **局部预测器：**设计为两层。
  - 一个局部历史记录,使用指令地址的低m位进行索引，每个入口k位，分别对应这个入口最近的k次分支，即最近k次分支的 跳转情况
  - 从局部历史记录选择出的入口对一个2^K的入口表进行索引，这些入口由2位计数器构成，以提供本地预测。
- **选择器：**
  - 使用分支局部地址的低m位（或者全局分枝跳转情况来构造选择器）分支局部地址索引，每个索引得到一个两位计数器，用来选择使用局部预测器还是使用全局预测器的预测结果。
  - 在设计时默认使用局部预测器，当两个预测器都正确或都不正确时，不改变计数器；当全局预测器正确而局部预测器预测错误时，计数器加1，否则减1。

选择器状态转移图


0/0, 1/0, 1/1
0/0, 1/1
0/1
1/0
0/1
1/0
1/0
0/1
0/0, 0/1, 1/1
0/0, 1/1
选择局部预测器预测结果
选择全局预测器预测结果


程序计数器
局部
历史表
1024
×
10
比特
10
局部
预测
1024
×
3比特
全局
预测
4096
×
2比特
全局历史
12
选择
预测
4096
×
2比特
最终预测

Comparisons



- The prediction capability of the local predictor does not improve beyond a certain size.
- The correlating predictor shows a significant improvement.
- The tournament predictor generates slightly better performance.

The Intel Core i7 Branch Predictor

- In addition to a local and a global predictor, it adds a loop exit predictor.
- The loop exit predictor tries to predict the Exact number of loop iterations.
- A selector then chooses from these three predictors based on their recent performance.
- A separate unit predicts target address for indirect branches.
- A stack used to predict return address is also used.

	处理器	分支预测器
静态分支预测	Intel 8086	无分支预测
	Intel 486	总是跳转
	Sun superSparc	总是不跳转
	HP 7x00	BTFN
	早期的 PowerPC	Profile
动态分支预测	Alpha21064、AMD K5	1 位分支预测
	PowerPC604 、 MIPS R10000	2 位分支预测
	Pentium Pro、Pentium II	2 级分支预测
	Alpha 21264	组合分支预测

表 1 一些典型商品处理器的分支预测机制

Example

A snapshot of the taken/not-taken behavior of a branch is:

... T T T T T T T T **N N T T N N T N N T**

If the branch predictor used is a 2-bit saturating counter, how many of the last ten branches are predicted correctly?

Answer:

According to the 2-bit branch predictor, the prediction for the last ten branches are:

ST, WT, SN, WN, ST, WT, SN, WN, SN, SN (S for strong, W for weak)

N N T T N N T N N T

According to the 2-bit saturating counter:

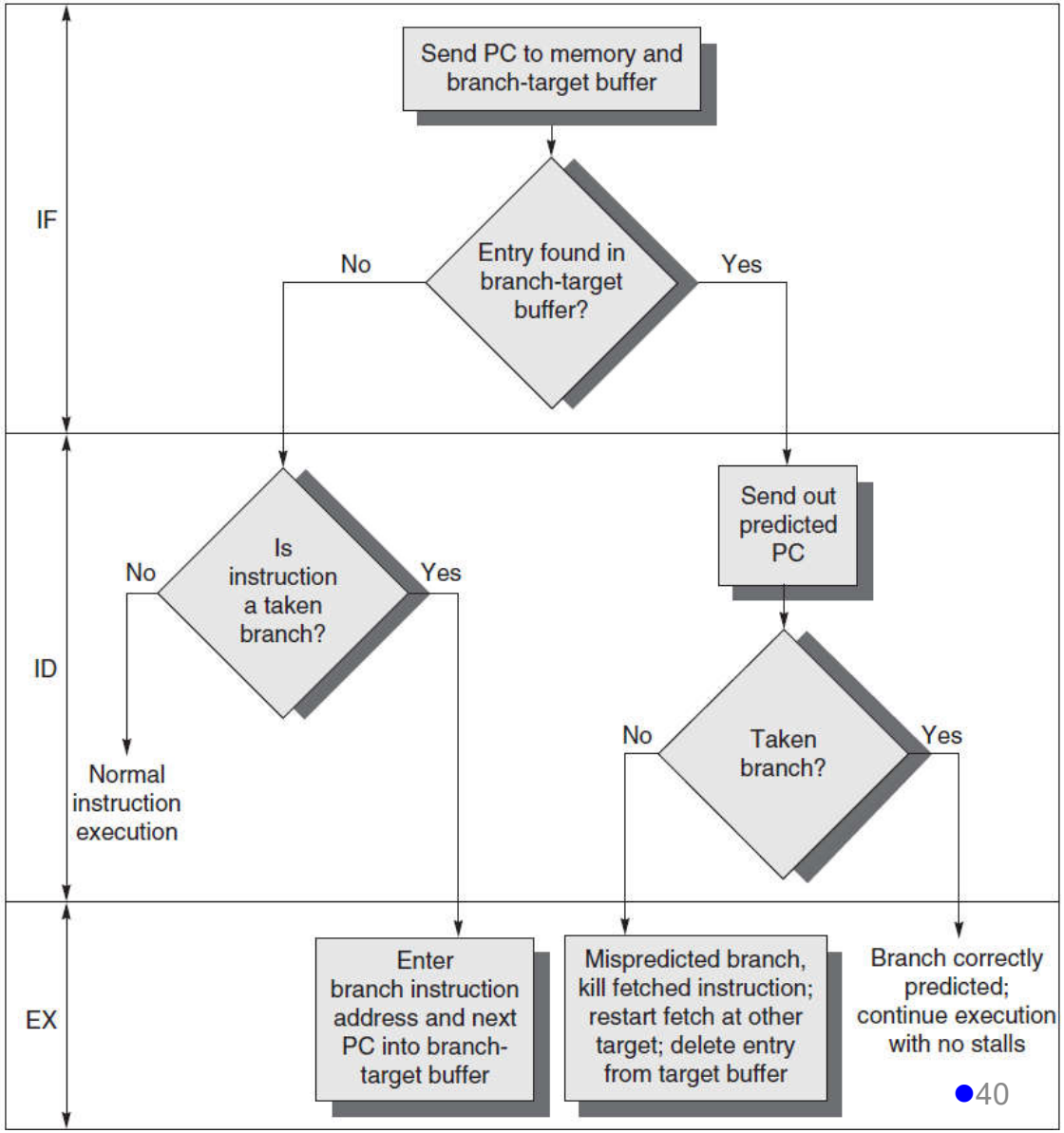
ST, WT, WN, WT, ST, WT, WN, WT, WN, SN

N N T T N N T N N T

No matter which one you use, the answer is 2 branches are predicted correctly.

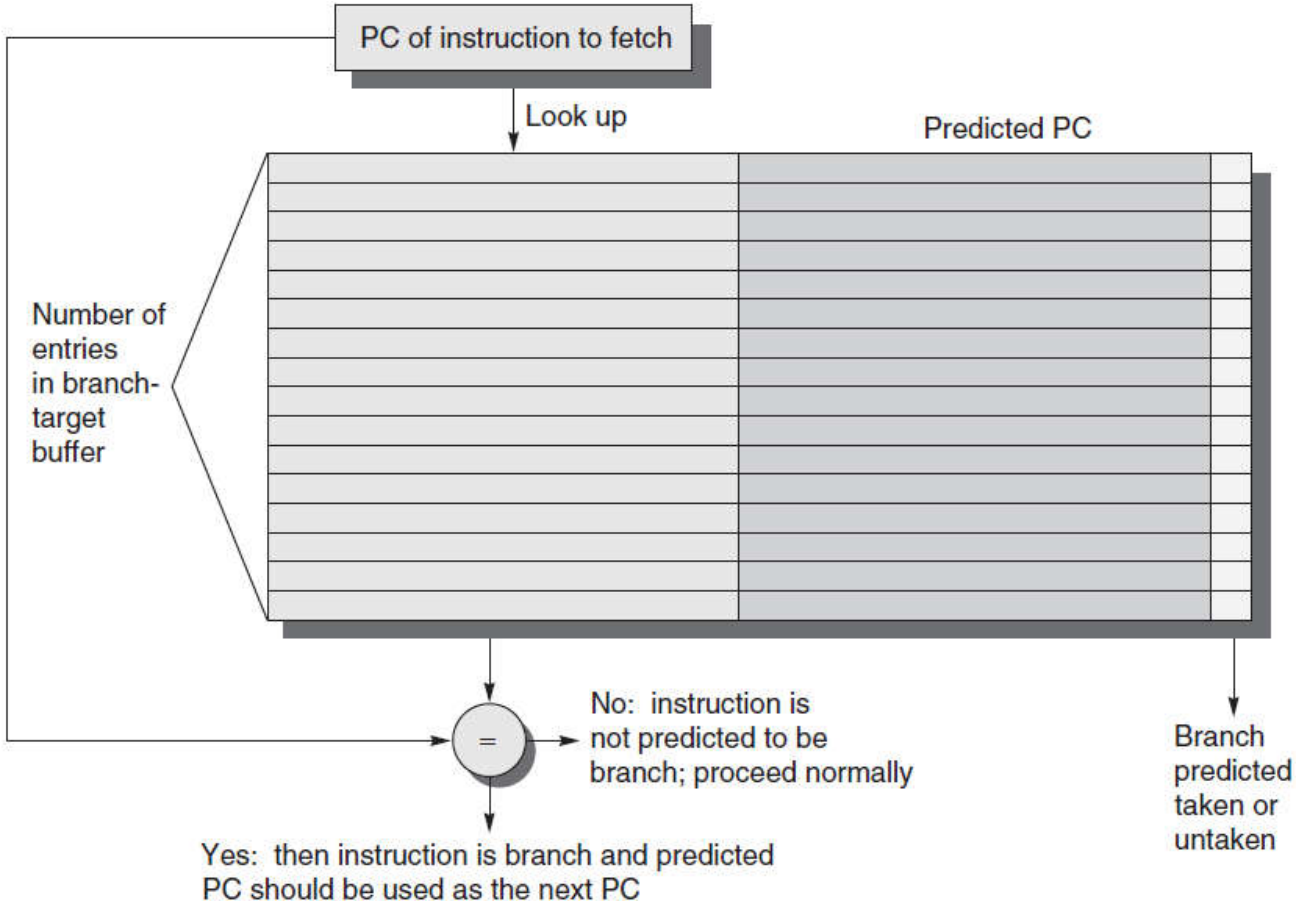
Predicting the Instruction Target Address: Branch-Target Buffer (or Branch-Target Cache)

- **Goal**: learn the **predicted instruction address at the** end of the IF stage
  - one cycle earlier
    - at best, branch-prediction buffers know the next predicted instruction at the end of ID
- **No branch delay**
  - if entry is found in buffer and prediction is correct
  - if entry not found and branch is not taken
- **Two cycle penalty**
  - if prediction is incorrect
  - if entry is not found and branch is taken



Predicting the Instruction Target Address: Branch-Target Buffer (or Branch-Target Cache)

- Only necessary to store the predicted taken branches since an untaken branch follows the same strategy as a non-branch instruction (to fetch the fall-through instruction)
- But using a 2-bit predictor requires to store information also for untaken branches



Branch Target Buffer: Penalty Table

Instruction in buffer	Prediction	Actual Branch	Penalty Clock Cycles
yes	taken	taken	0
yes	taken	not taken	2
no		taken	2
no		not taken	0

- Assuming
  - –85% prediction accuracy, 90% buffer hit rate, 60% branches taken

- P(branch not in buffer, but taken) = 0.1 x 0.6 = 0.06
- P(branch in buffer but not taken) = 0.9 x 0.15 = 0.135
- Total branch penalty = (0.135 + 0.06) x 2 = 0.39

  – the penalty for delayed-branch was about 0.5 clock cycles
  – penalty is lower with better predictors (and smaller branch delays)

Example

Assume a machine that has a branch-target buffer with 8 entries. A branch in this machine has a penalty of 2 clock cycles if the branch is taken and the target instruction is not in the branch-target buffer, or if the branch is predicted as taken, the instruction is in the branch-target buffer, but the branch is actually not taken. In all other situations the branch penalty is zero. What is the total branch penalty in this machine, measured in clock cycles, if

- the branch prediction accuracy is 90%;
- 80% of the time the target instruction is in the buffer (80% hit rate in the buffer);
- 60% of the branches are actually taken.

Answer：

Probability of branch taken but not found in the buffer:

Percentage of taken branches * buffer miss rate = 60% * 20% = 0.12

Probability of branch found in buffer but predicted wrong:

Buffer hit rate * prediction miss rate = 80% * 10% = 0.08

Average branch penalty = (0.12 + 0.08) * 2 = 0.4 clock cycles

In-class Quiz #3