

# **Branch Prediction Techniques**

E. Sanchez

**Politecnico di Torino  
Dipartimento di Automatica e Informatica**

# INTRODUCTION

**Branches can potentially impact in a very serious way the pipeline performance.**

**It is possible to reduce the performance losses by predicting how branches will behave.**

# Branch prediction schemes

They aim at correctly forecasting branches, thus reducing the chance that control dependences cause stalls.

They can be categorized in two groups:

- ***static techniques***: they are handled by the compiler resorting to a preliminary analysis of the code
- ***dynamic techniques***: they are implemented by the hardware based on the behavior of the code.

# Static branch prediction

It can be useful when combined with other static techniques, such as

- Enabling delayed branches
- Rescheduling to avoid data hazards.

# Static branch prediction

The compiler may predict branch behavior in different alternative ways:

- Always predicting branches as taken
- Predicting branch behavior depending on branch direction
- Predicting on the basis of profile information coming from earlier runs.

# Predicting branches as taken

On the SPEC92 programs this gives:

- 34% average misprediction rate
- highly variable rate (from 9% to 59%).

Other techniques may behave better in the average, but still with very high variations from program to program.

# **Predicting branch behavior depending on branch direction**

**This technique is based on the observation that**

- Forward branches are more often untaken**
- Backward branches are more often taken.**

**This behavior is mainly due to loop constructs.**

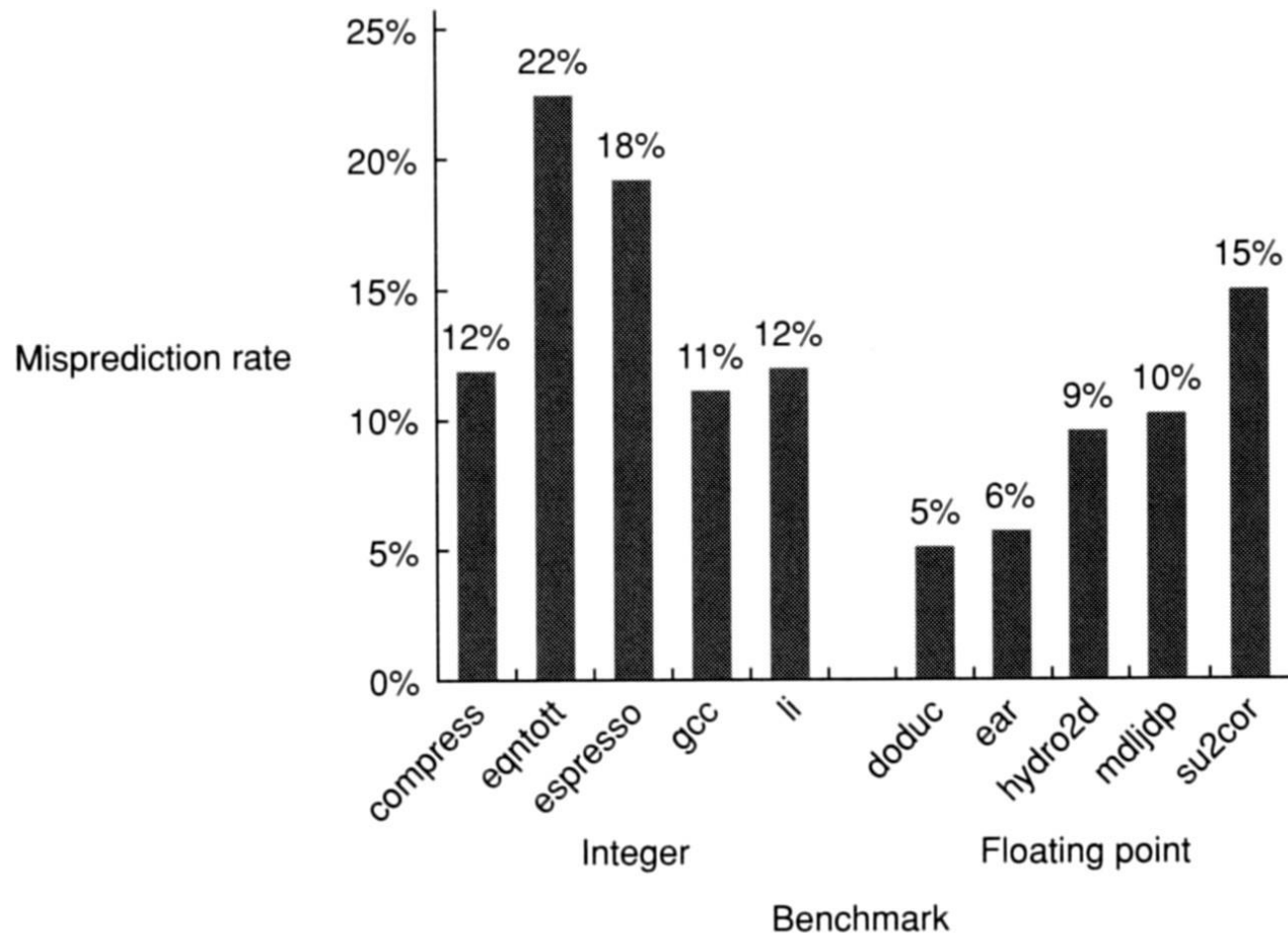
# **Predicting branches based on profile information**

**This technique is based on**

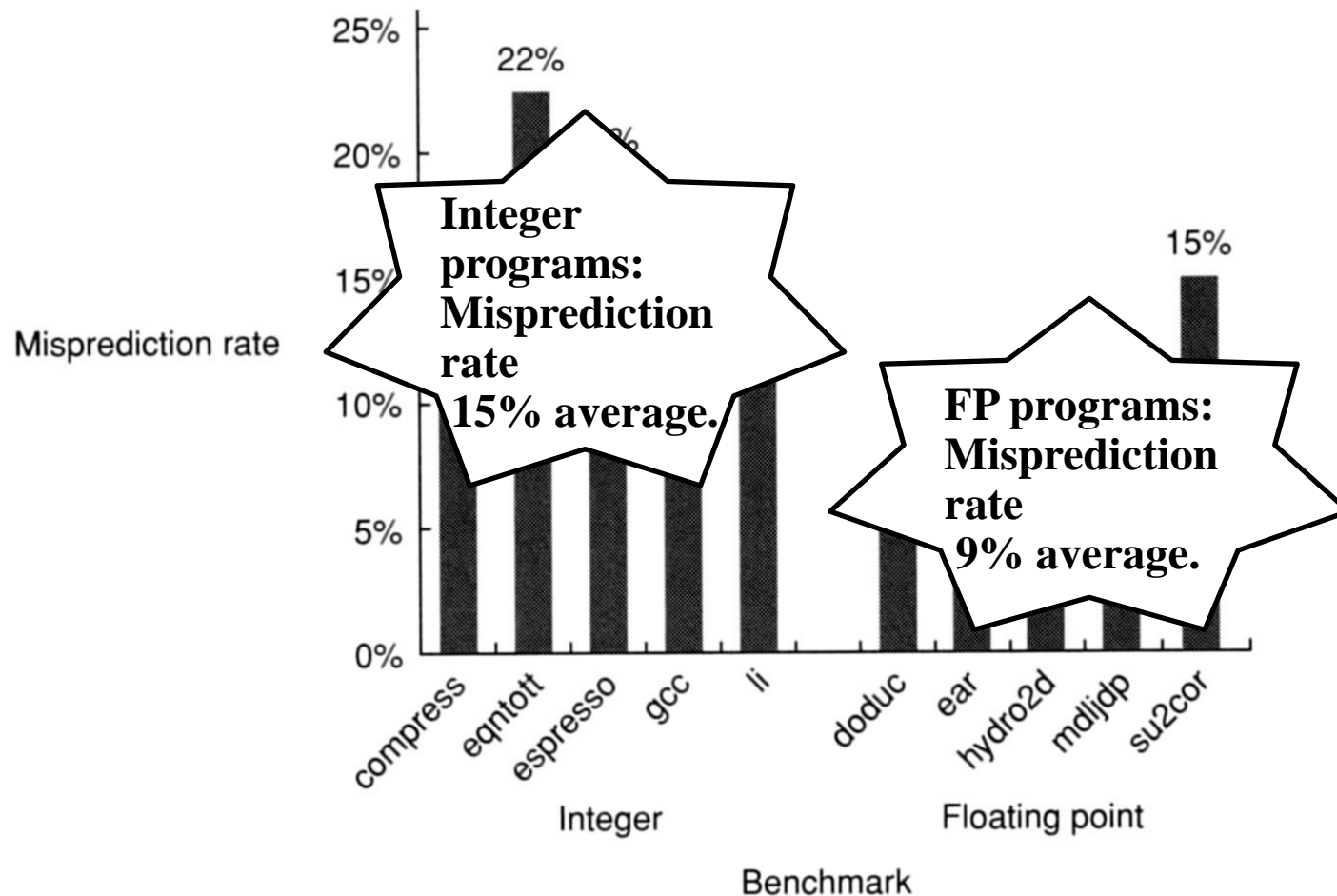
- **Identifying a typical sequence of input stimula for the program**
- **Running the program a limited number of times using these stimula**
- **Gathering statistics on the behavior of branches**
- **Using these statistics as predictions.**



# Profile-based prediction effectiveness



# Profile-based prediction effectiveness



# Dynamic branch prediction

Dynamic schemas are based on hardware, and use the branch instruction address to activate the different prediction mechanisms.

Dynamic branch prediction can be based on different techniques:

- Branch history table
  - One- and two-bit prediction schemes
- Two-level prediction schemes (correlating predictors)
- Branch-target buffer
- Others.

# BRANCH HISTORY TABLE

It is the simplest method for dynamic branch prediction.

The *Branch History Table* (BHT) is a small memory:

- indexed by the lower portion of the address of the branch instruction
- containing for each entry one or more bits recording whether the branch has been taken or not the last time it has been executed.

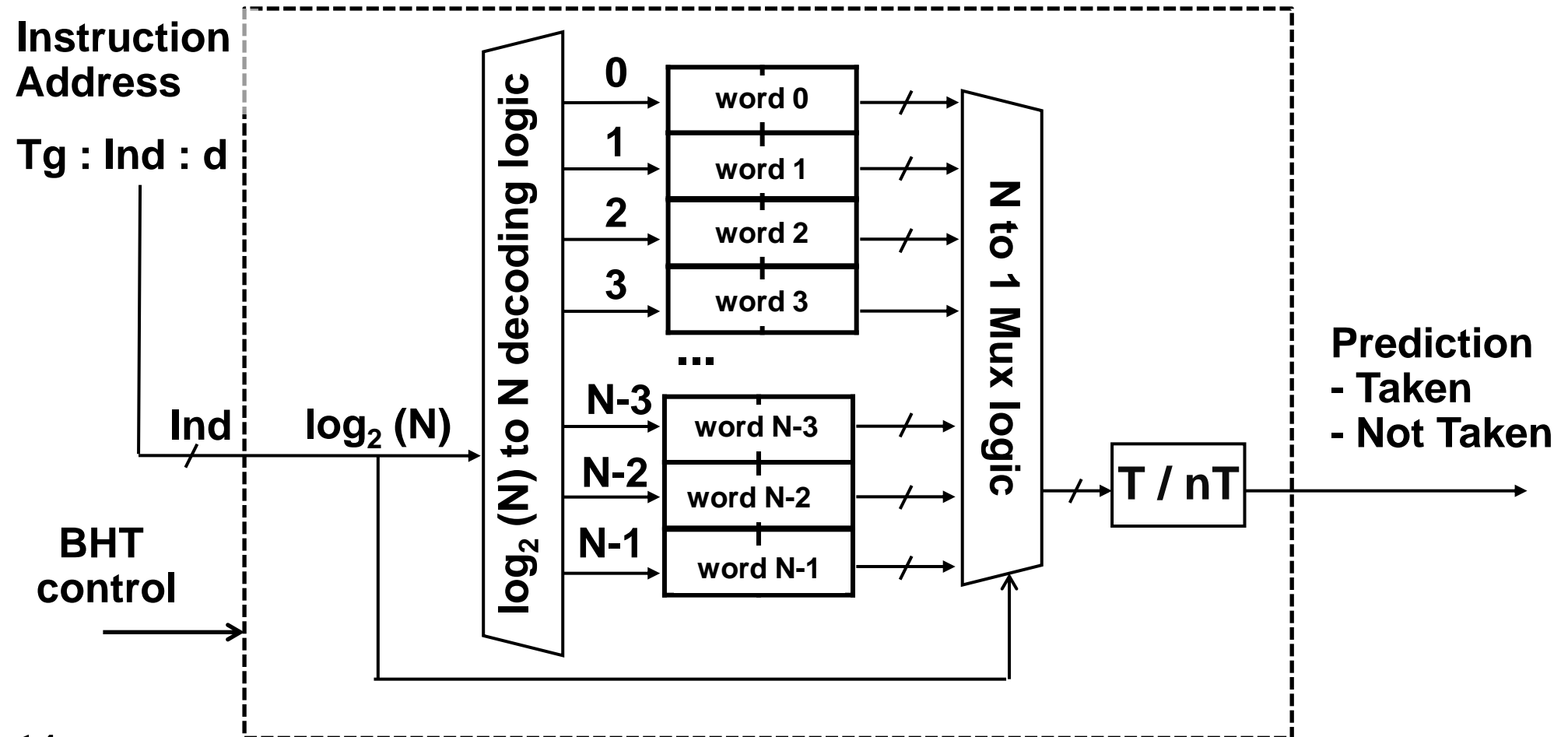
# Algorithm

**Each time a branch is decoded**

- **An access is made in the BHT using the lowest portion of its address as an index**
- **The prediction stored in the table is used, and the new PC is computed according to the prediction.**

**When the branch result is known, the BHT is possibly updated.**

# BHT implementation



# Effectiveness

**The effectiveness of the method depends on:**

- **the chance that the BHT entry relates to the branch of interest**
- **the accuracy of the prediction.**

**In the MIPS processor, the evaluation of the branch condition is performed while branch instructions are identified. Therefore, the described technique does not give any advantage.**

# Example

**Consider a loop branch which is taken nine times in a row, then not taken once. Assume that the entry for the branch is not shared with other branches.**

**The steady-state prediction behavior using a 1-bit BHT will mispredict on the first and last loop iterations.**

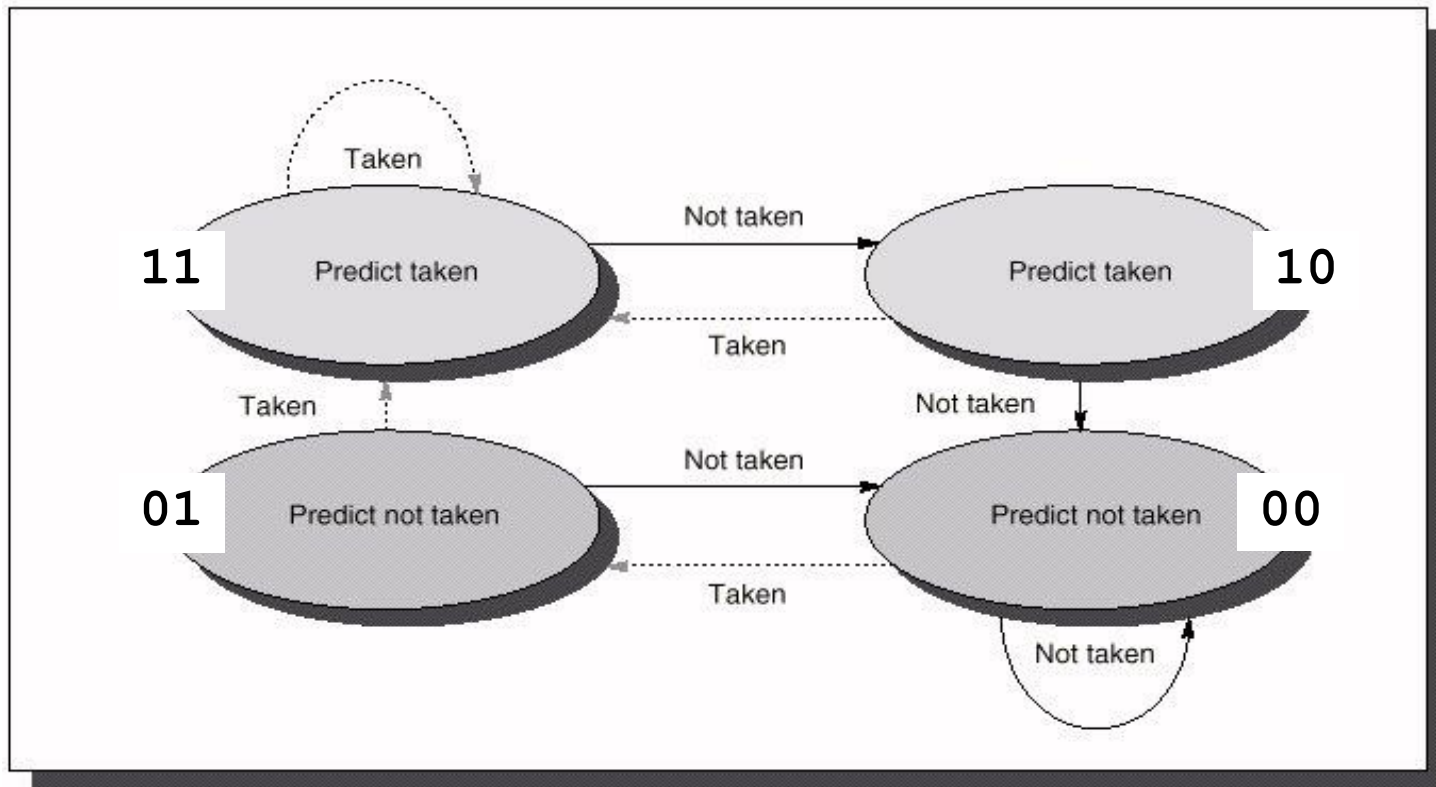
**The prediction accuracy is thus 80%, lower than simply assuming that the branch is taken (90%).**



# Two-bit Prediction Schemes

They provide higher prediction capabilities.

For every branch, two bits are maintained, and the prediction is changed only after missing twice.



# **n-bit Prediction Scheme**

**It is the general case of the previous one.**

**It is based on an n-bit saturating counter associated to every branch instruction.**

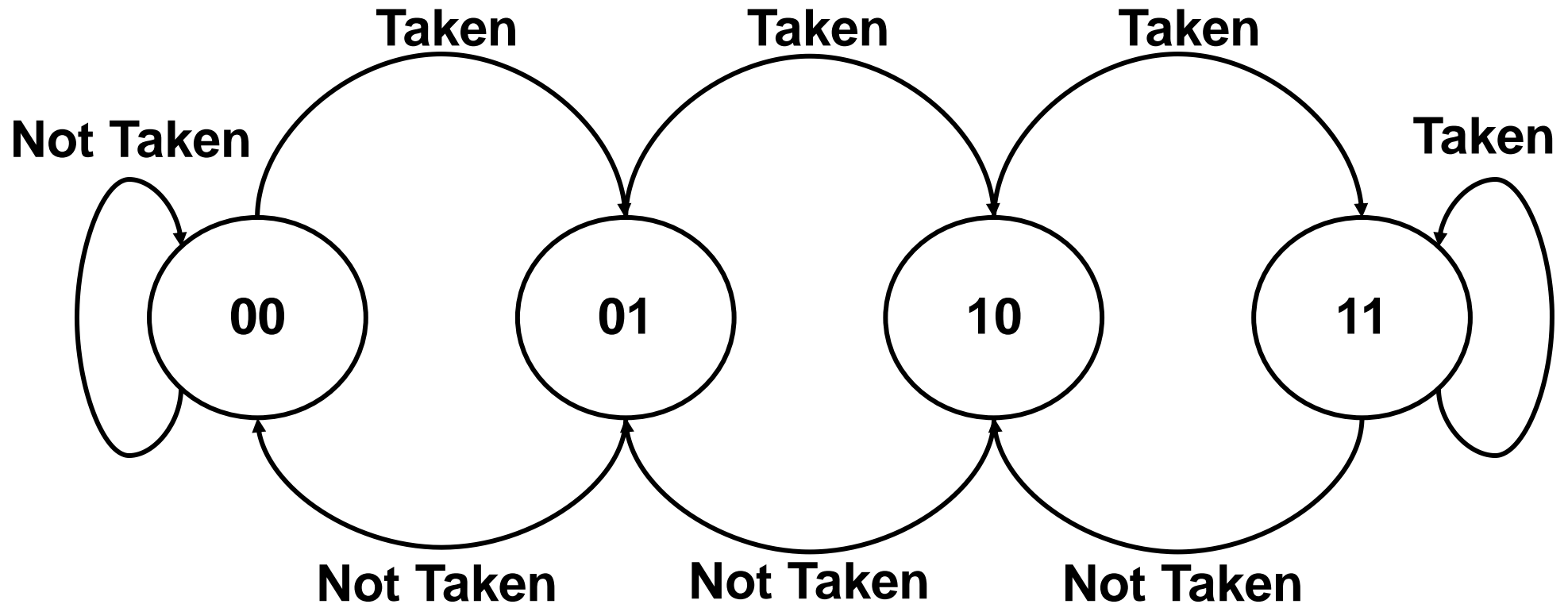
**The counter is incremented each time the branch is taken, decremented each time it is not.**

**When the counter value is greater than one half of its maximum value, the branch is predicted as taken, when it is lower it is predicted as not taken.**

**Experiments have shown that there is little advantage in using  $n > 2$ .**

**This technique is also called bimodal branch prediction.**

# 2-bit saturating counter

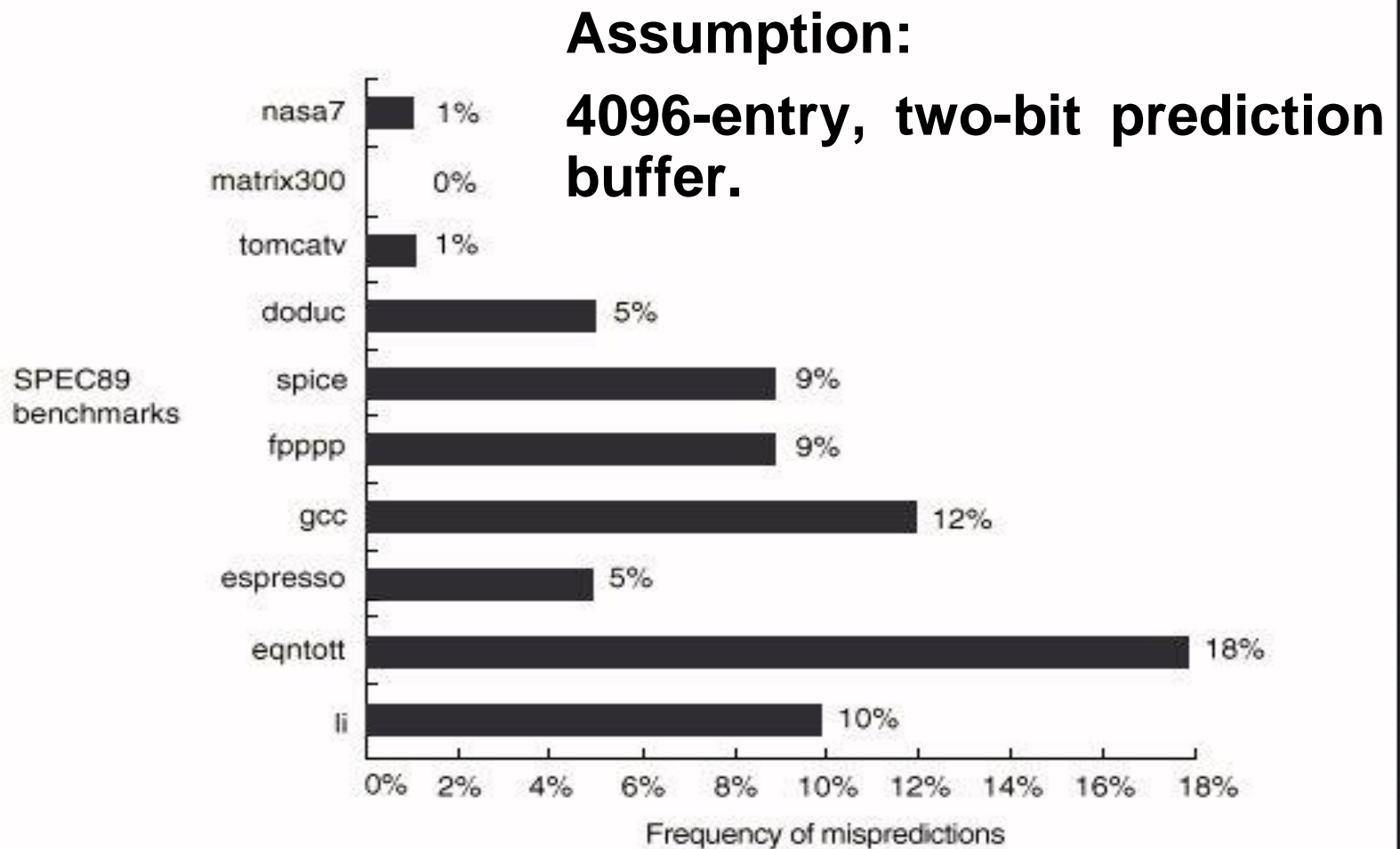


# Performance impact of branches

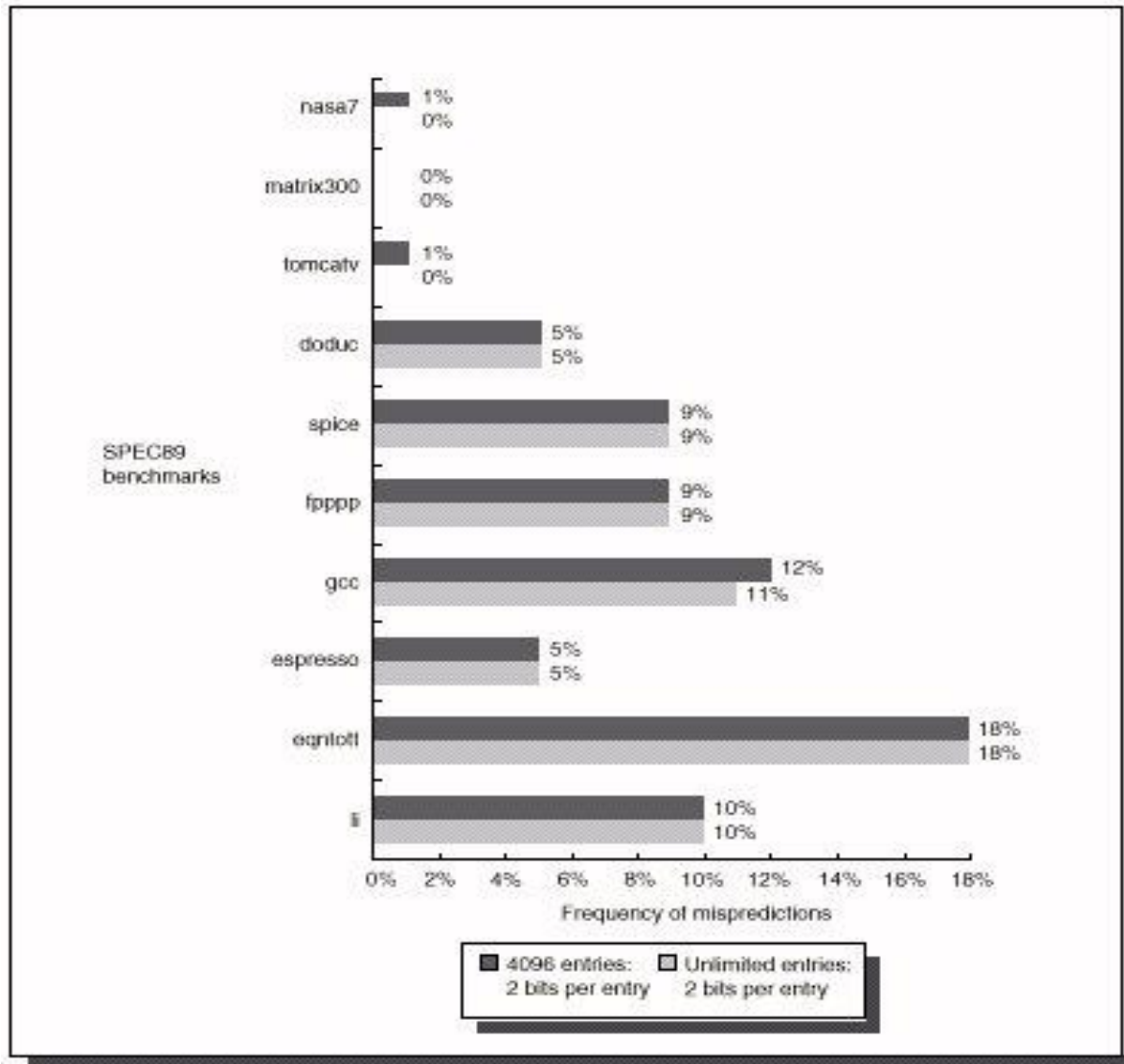
It depends on

- prediction accuracy
- branch cost (penalty for misprediction)
- branch frequency (lower for FP programs).

# Prediction Accuracy for SPEC89



# Dependence on Buffer Size



# CORRELATING PREDICTORS

This approach (also called *two-level predictors*) is based on the dependencies between the results of last branches.

## Example

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

The behavior of the last branch is strongly dependent on the result of the previous ones.

# **( $m,n$ ) predictors**

**They use the behavior of the last  $m$  branches to choose from  $2^m$  branch predictors, each of which is a  $n$ -bit predictor.**

**The hardware required for implementing this scheme is very simple:**

- The history of the most recent  $m$  branches is recorded in an  $m$ -bit shift register, where each bit records whether the branch was taken or not.**
- The branch-prediction buffer is indexed using a concatenation of the low-order bits from the branch address with the  $m$  low-order history bits.**



# **(1,1) predictor**

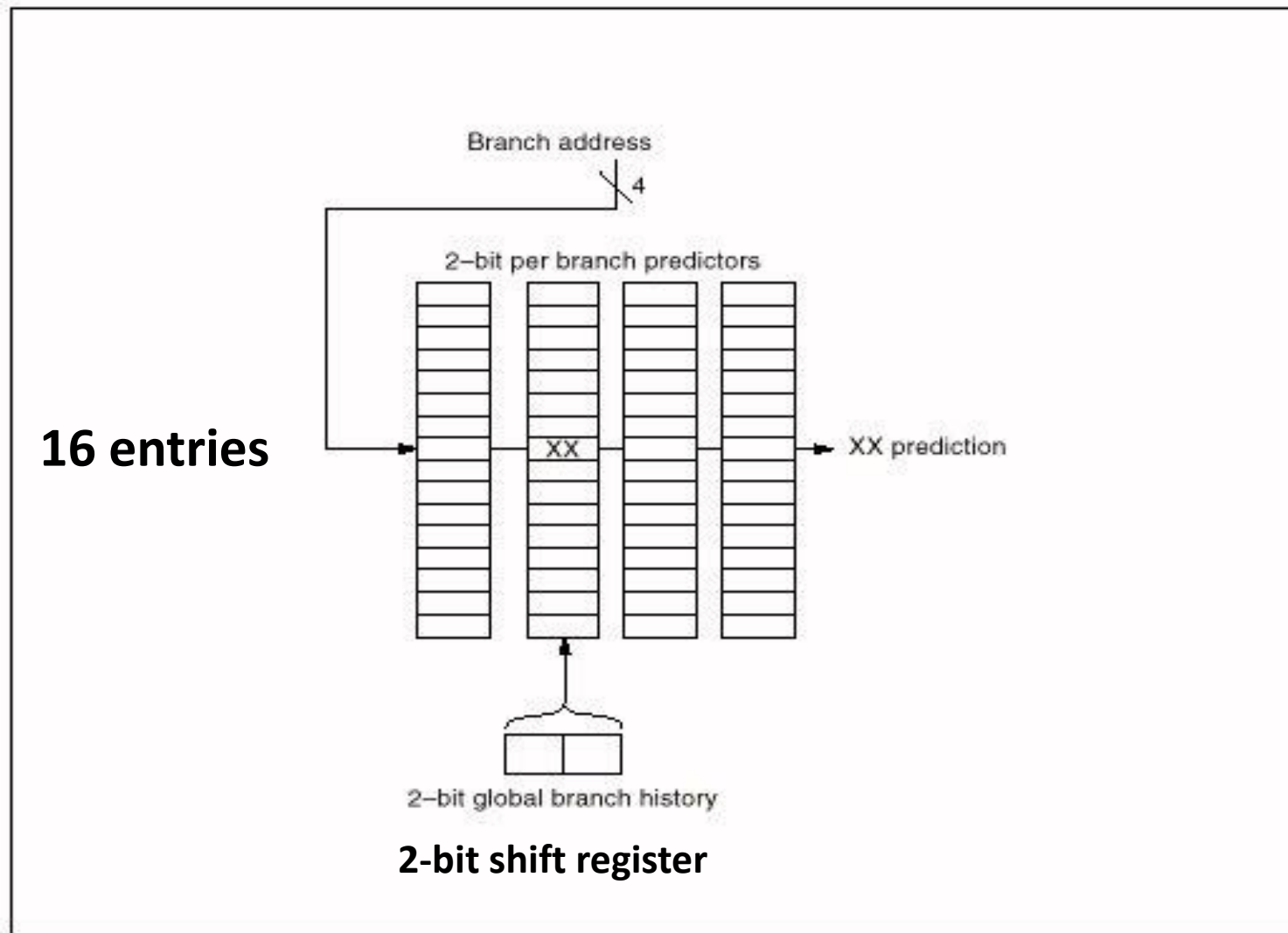
**In this case:  $m = 1$ ,  $n = 1$ .**

**Each branch is associated with  $2^m$  (two) predictors of  $n$  (one) bits:**

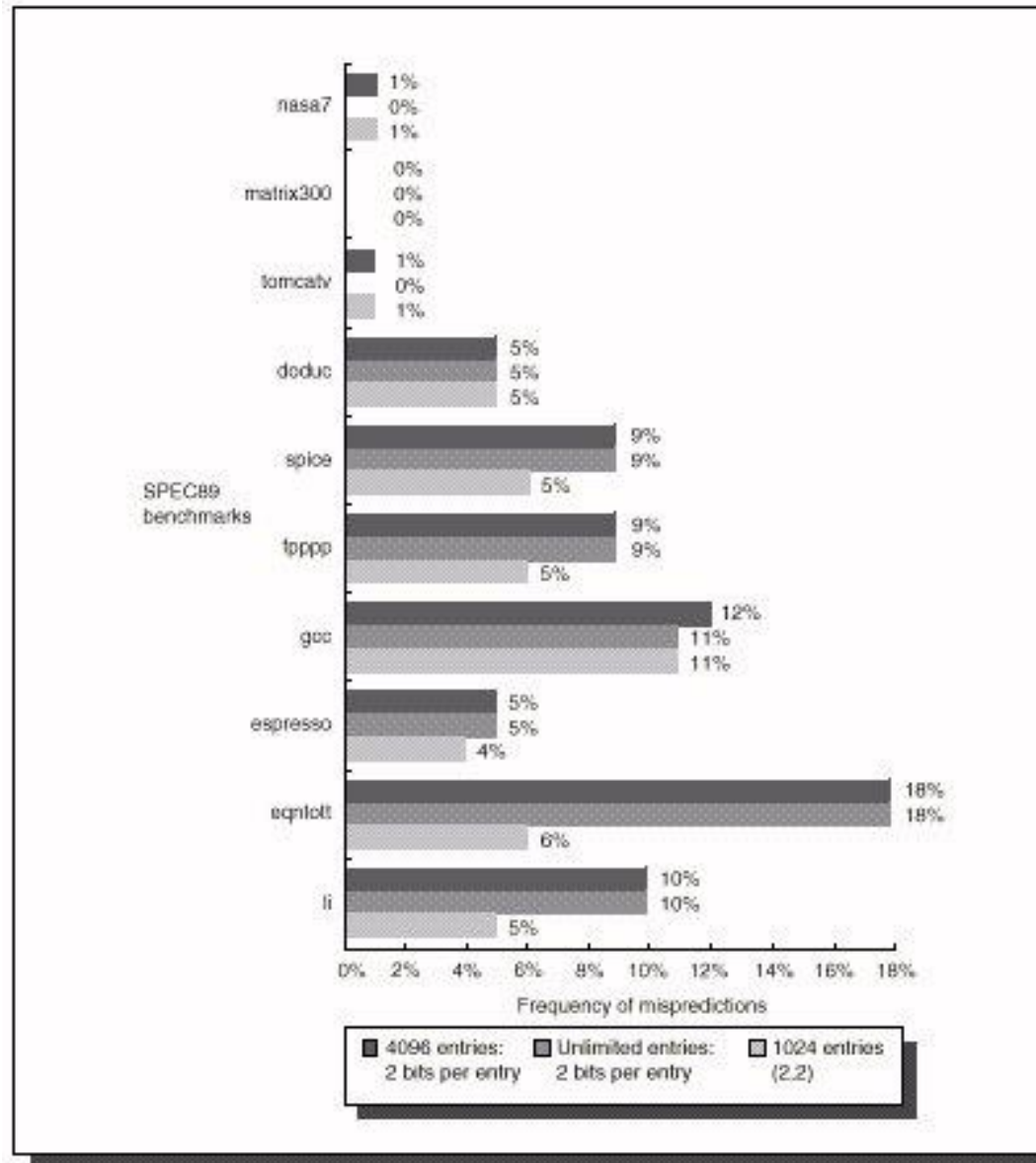
- **one reporting the prediction in the case the previous branch was taken**
- **one with the prediction in the case the previous branch was not taken.**

**Warning: the two branches the prediction is based on can be different.**

# (2,2) predictor implementation



# Performance comparison



# Experimental Evidence

Prediction accuracy does not grow significantly by increasing

- the buffer size
- the number of bits per predictor.

# BRANCH-TARGET BUFFER

Reducing the negative effects of control dependencies requires knowing as soon as possible

- whether the branch has to be taken or not
- the new value of the PC (if the branch is assumed to be taken).

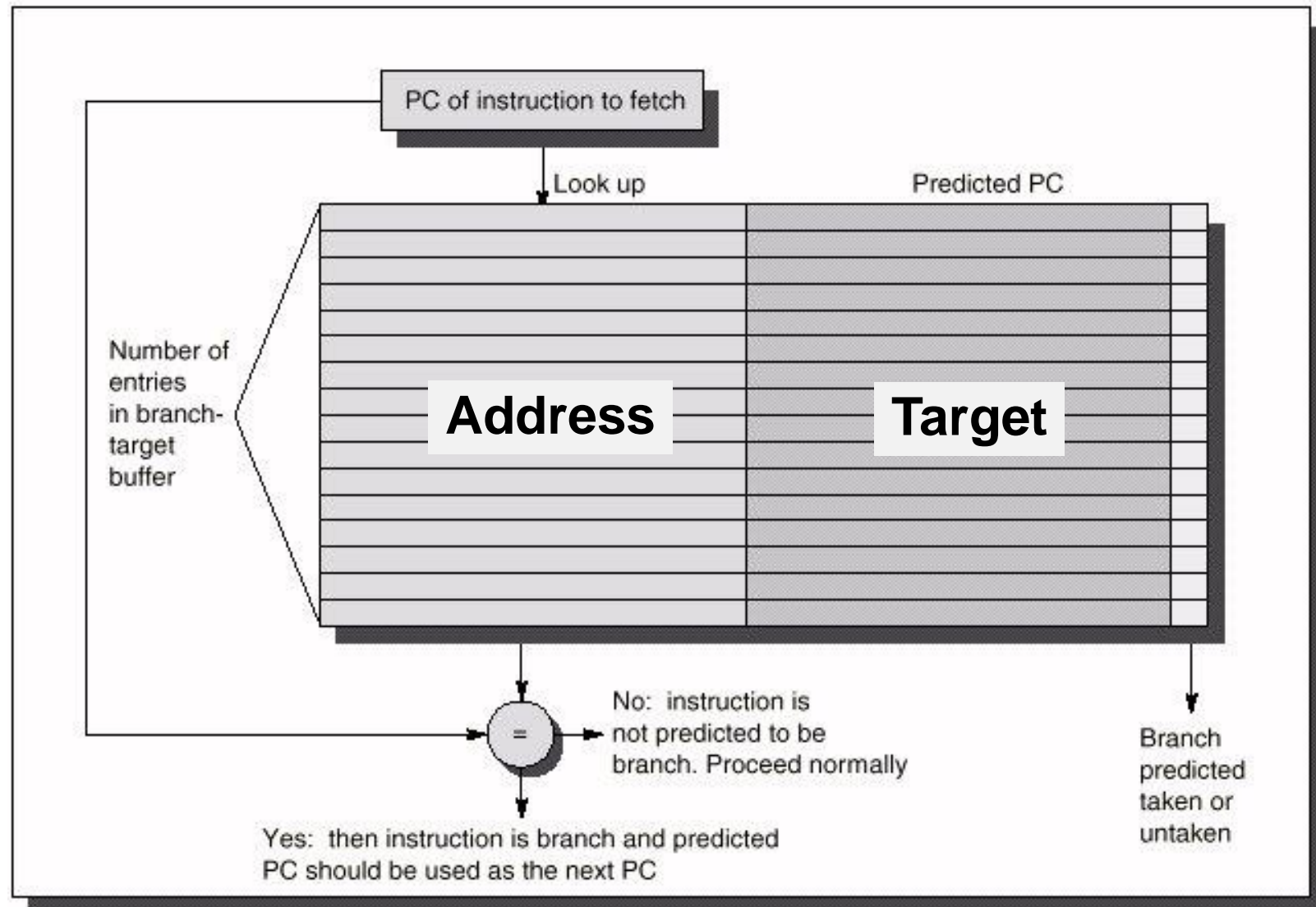
The later issue is faced by introducing a *branch-target buffer* (or *cache*).

Each entry of the branch-target buffer contains

- the *address* of the considered branch
- the *target* value to be loaded in the PC.

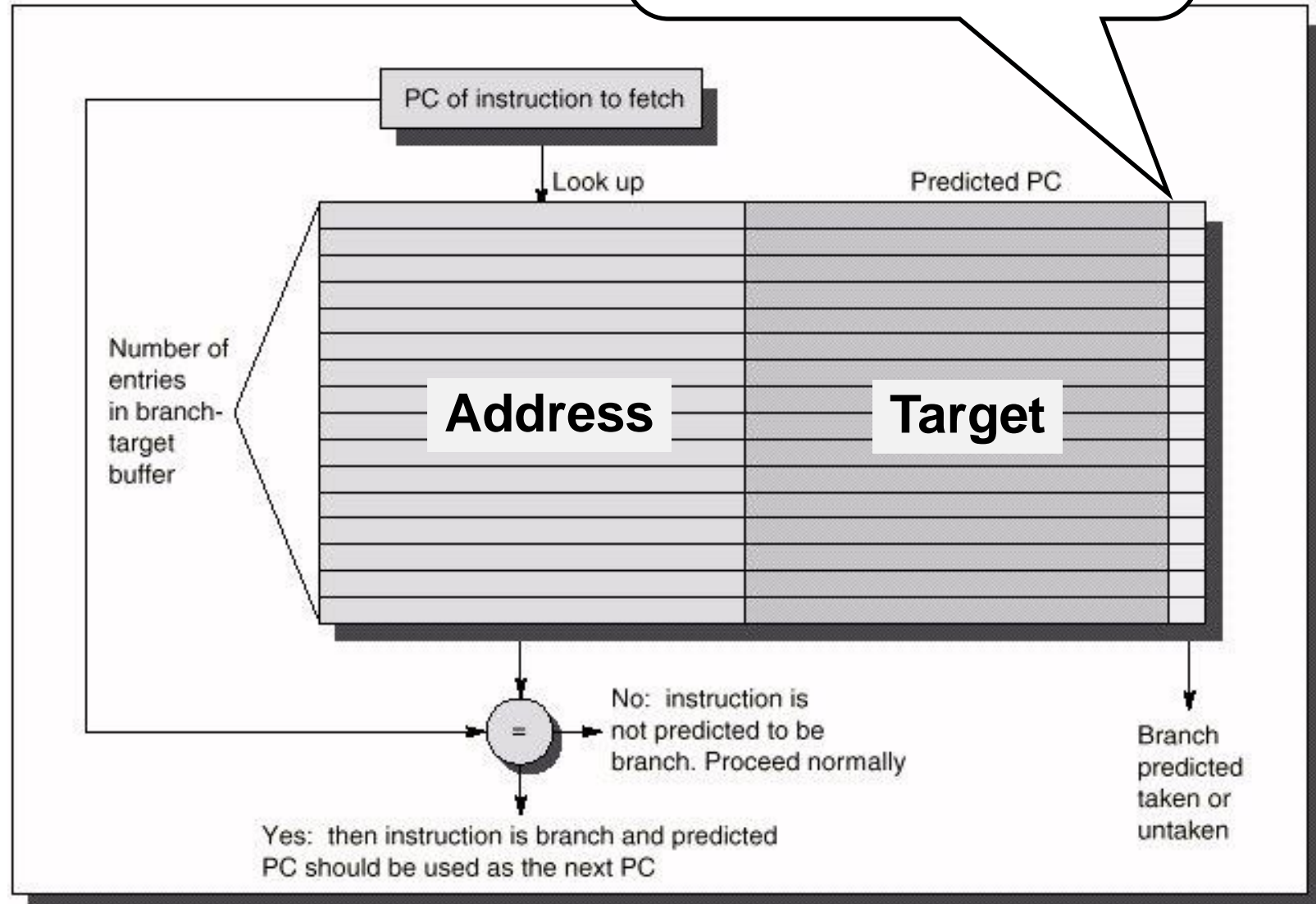
Using the branch-target buffer, the PC is loaded with the new value at the end of the IF stage, i.e., even before the branch instruction is decoded.

# Branch-target Buffer: Architecture

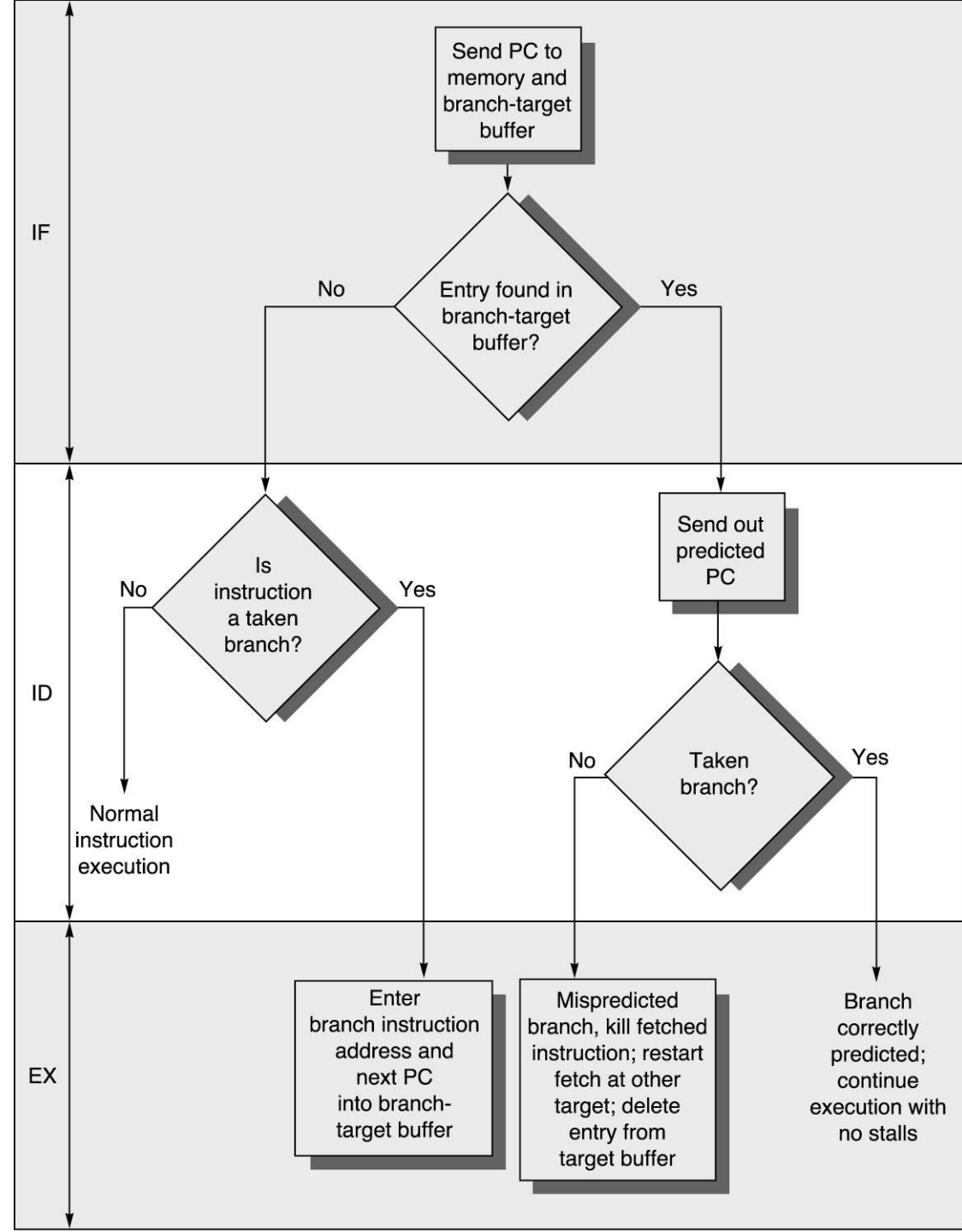


# Branch-target Buffer Architecture

This field is not strictly necessary.



# Branch-target Buffer: Behavior





# **Branch-target buffer: Advanced Issues**

**If a two-bit prediction strategy is adopted, it is possible to combine a branch-target buffer with a branch prediction buffer (i.e., branch history table).**

# Branch-target buffer: Performance Effects

Let assume the following penalty parameters

Instruction in buffer	Prediction	Actual branch	Penalty c.c.
Yes	taken	taken	0
Yes	taken	not taken	2
No		taken	2
No		not taken	0

Let also assume that

- the prediction accuracy is 90%
- the hit rate in the buffer is 90%
- taken branches are 60%.

Which is the total branch penalty?

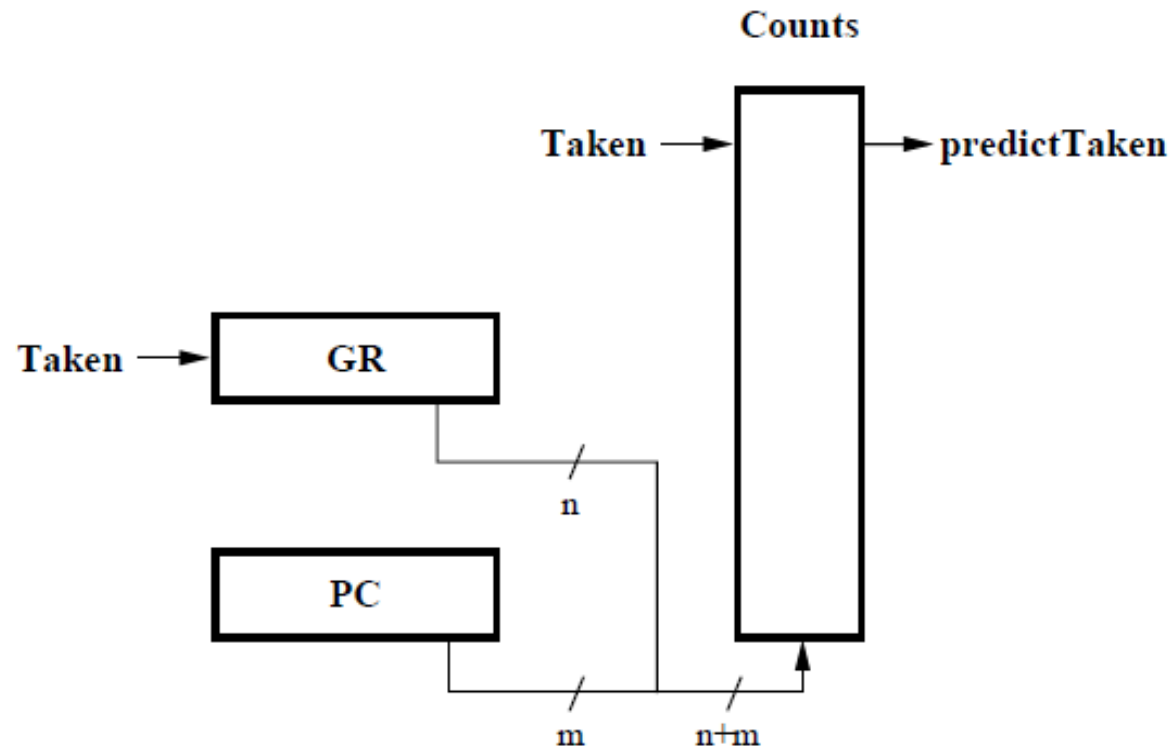
# Solution

$$\begin{aligned}\text{Branch Penalty} &= \text{Hit branches \& Not Taken} + \text{Miss branches \& Taken} \\ &= (\text{percent buffer hit rate} \times \text{percent incorrect predictions} \times 2) + \\ &\quad ((1 - \text{percent buffer hit rate}) \times \text{taken branches} \times 2) \\ &= (90\% \times 10\% \times 2) + (10\% \times 60\% \times 2) \\ &= 0.18 + 0.12 = 0.30 \text{ c.c.}\end{aligned}$$

This figure should be compared with the 0.50 clock cycles per branch penalty existing with delayed branches.

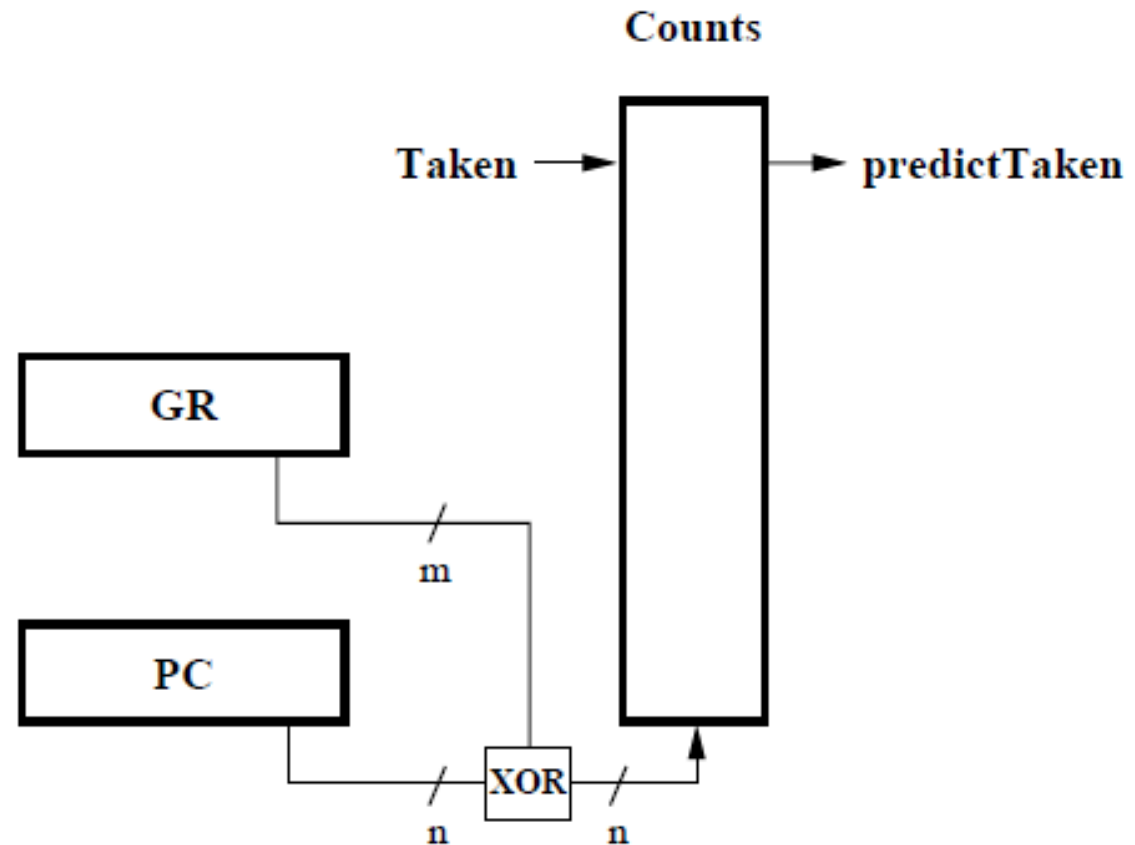
# Global predictor with index selection (*gselect*)

A saturated-counter prediction table is accessed by concatenating the branches global history (GR) and the branch address (PC).

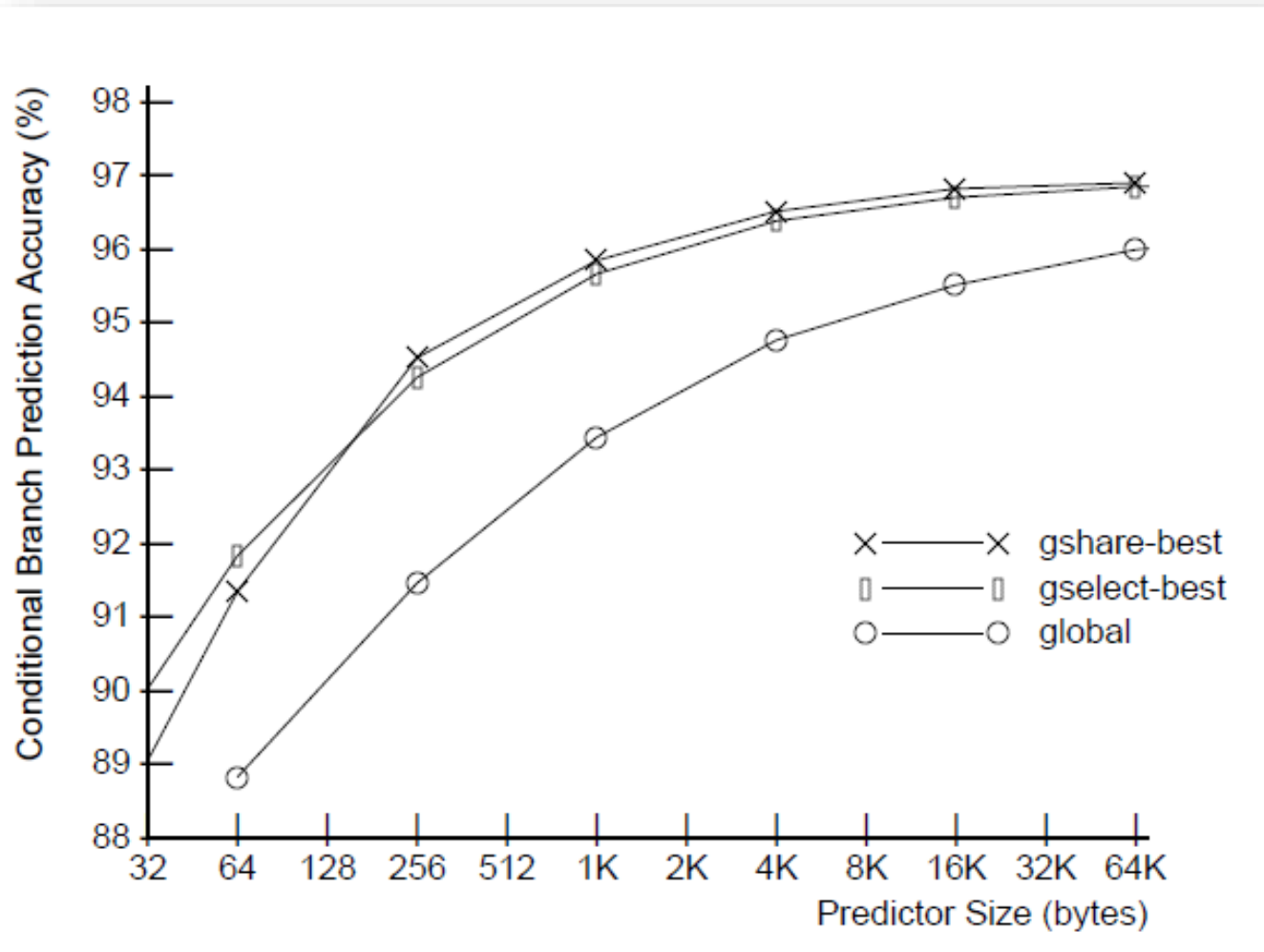


# Global history with index sharing (*gshare*)

A saturated-counter prediction table is accessed by XORing the branches global history (GR) and the branch address (PC).



# Performance comparison



# Performance comparison

