2001

An Efficient Indirect Branch Predictor

Yul Chu
The University of Texas Rio Grande Valley

M. R. Ito

Recommended Citation

Chu, Yul, and Mabo Robert Ito. "An efficient indirect branch predictor." In European Conference on Parallel Processing, pp. 394-402. Springer, Berlin, Heidelberg, 2001.

An Efficient Indirect Branch Predictor

Yul Chu[1] and M. R. Ito[2]

[1] Electrical and Computer Engineering Department, Mississippi State University,
Box 9571, Mississippi State, MS 39762, USA
chu@ece.msstate.edu
[2] Electrical and Computer Engineering Department, University of British Columbia,
2356 Main Mall, Vancouver, BC V6T1Z4, Canada
mito@ece.ubc.ca

Abstract. In this paper, we present a new hybrid branch predictor called the GoStay2, which can effectively reduce indirect misprediction rates. The GoStay2 has two different mechanisms compared to other 2-stage hybrid predictors that use a Branch Target Buffer (BTB) as the first stage predictor: Firstly, to reduce conflict misses in the first stage, a new effective 2-way cache scheme is used instead of a 4-way set-associative. Secondly, to reduce mispredictions caused by an inefficient predict and update rule, a new selection mechanism and update rule are proposed. We have developed a simulation program by using Shade and Spixtools, provided by SUN Microsystems, on an Ultra SPARC/10 processor. Our results show that the GoStay2 improves indirect misprediction rates of a 64-entry to 4K-entry BTB (with a 512- or 1K-entry PHT) by 14.9% to 21.53% compared to the leaky filter.

1 Introduction

Speculatively executed instructions used in a branch prediction can degrade system performance since they must be discarded when a branch is mispredicted. Thus, more accurate branch predictors are required for reducing the impact on overall system performance. Single-target direct branches can be predicted with reported hit-ratios of up to 97% [1]. By contrast, indirect branches with multi-targets are harder to predict accurately. The sources of indirect branches are switch statements, virtual function calls, or indirect function calls [2][3].

Conventional branch predictors predict branch direction and generate the target address associated with that direction. In conventional branch schemes, BTB-based prediction schemes are the only predictor for indirect branch prediction in conventional branch schemes since an indirect branch needs a full target address instead of just the direction (taken or not-taken). Chang et al. [4] showed that the small proportion of indirect branches (2 to 3%) for SPECint95 benchmarks can be a critical factor in degrading system performance.

There are two types of indirect branch predictors classified according to the number of component predictors: A single-scheme predictor that has only one predictor and a hybrid predictor that combines two or more single-scheme predictors. The Branch Target Buffer (BTB) represents typical single-scheme predictors. The

BTB stores both the branch address and target address. If a current branch is found in the BTB, it is predicted as 'taken' with the target address. If there is a misprediction or a first-miss, the branch and target addresses are updated after the execution. In this paper, we considered hybrid branch predictors consisting of two single-scheme predictors only. Moreover, a BTB is used for the first stage predictor.

Chang et al. [4] proposed a predictor by using the Target Cache to improve the accuracy of indirect branch predictions. The Target Cache is similar to the Pattern History Table (PHT) of a 2-level branch predictor except that the Target Cache records the branch target while the PHT holds branch directions such as taken/not taken. This predictor XORs a pattern- or path-based history bits with the branch address to index the prediction. The Target Cache can reduce the misprediction rates of indirect branches significantly. For example, a 512-entry Target Cache achieved a misprediction rate of 30.4% and 30.9% for gcc and perl, while a 1K–entry 4-way set-associative achieves misprediction rates of 60% and 70.4% [4].

Driesen and Hölzle [3] introduced two variants of the Cascaded Predictor, which has two stages and two different update rules (a leaky or strict filter); a BTB for the first stage and a gshare-like two-level predictor as the second stage. The small-sized BTB works as a filter and the second stage predictor stores indirect branches that need branch history-based prediction. The second stage uses an indexing function similar to the Target Cache such as a path-based branch history XORing with a low-order branch address to index the prediction table.

In this paper, we present a 2-stage hybrid predictor called the GoStay2, which employs a new cache scheme for the first stage, a new selection mechanism, and update rule by using a 2-bit flag for both stages. We show that the GoStay2 outperforms other 2-stage hybrid predictors such as the Cascaded predictor [3] and Target Cache [4] by improving the accuracy of indirect branch predictions.

This paper is organized as follows: section 2 presents the new branch architecture with the two mechanisms for reducing indirect mispredictions; section 3 describes simulation methodology and benchmark programs; section 4 presents our simulation results; and section 5 provides our conclusions.

2 GoStay2 Branch Predictor

Among the several indirect branch predictors in section 1, the leaky filter of the Cascaded predictor offers the most effective misprediction rate for indirect branches [2][3]. However, the leaky filter has some problems that degrade system performance:

- Conflict misses – If a table (BTB) has a small number of entries (say, less than 512 entries), conflict misses might degrade the misprediction rate considerably;
- Inefficient predict rules – If a branch address is found at both stages, the second stage has priority for prediction. If the first stage has a correct target address and the second stage has an incorrect target address, then the assumed priority of the second stage always causes a misprediction; and
- Inefficient update rules – If a predicted target address is wrong, then the resolved target address of the branch address is updated in both stages. This also causes a misprediction if the replaced target address is needed for a following branch.

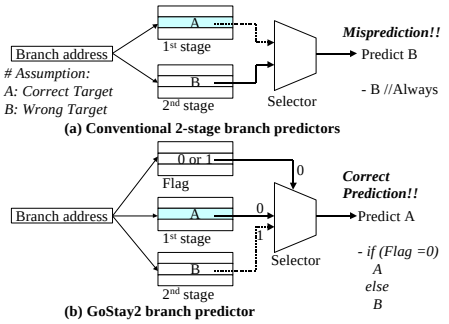


Fig.1. The basic operation of conventional 2-stage and GoStay2 branch predictors.

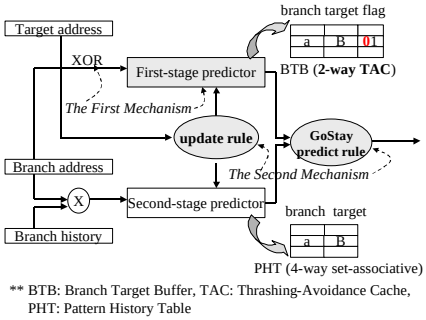


Fig. 2. The overview of a GoStay predictor.

Figure 1(a) shows that in a conventional 2-stage branch predictor, if the first stage has a correct target address (A) but the second stage has a wrong one (B), then the prediction (B) leads to misprediction since the second stage always takes priority of prediction.

Figure 1(b) shows the basic operation of the GoStay2 predictor, which can reduce mispredictions effectively. In the GoStay2 predictor, the prediction will be made according to the second flag in the first stage. In Figure 1(b), since the second flag is '0', the prediction (A) is made with the target address in the first stage (A), which leads to correct prediction. The flag is updated to '0' or '1' according to the update rule (refer to section 2.2).

Figure 2 shows the overview of a GoStay2 predictor, which has two different mechanisms compared to other 2-stage hybrid branch predictors. *'GoStay2' implies GoStay predict and update rules, as well as a 2-bit flag in the first stage.* The first bit of the flag is for the replacement policy of the first stage predictor [5], and the second bit is for the GoStay predict and update rule.

2.1 The 2-way TAC scheme for the first stage – The First Mechanism

For the first mechanism, we use a 2-way TAC (Thrashing-Avoidance Cache) developed by Chu and Ito [5] for the first stage to reduce conflict misses. The 2-way TAC scheme employs 2 banks and XOR indexing function [5][6].

Figure 3 shows the main function of the 2-way TAC, which is to place a group of instructions into a bank according to the BSL (Bank Selection Logic) and the BoPLRU (Bank originated Pseudo LRU) replacement policy. Figure 4 shows pseudo code for the basic operation in Figure 3.

The function of the Bank Selection Logic (BSL) is to select a bank initially on a cache miss according to call instructions. The BSL employs a 1-bit counter, which is toggled (0 or 1) whenever a fetched instruction proves to be a call instruction. In Figure 3 and 4, the value of the 1-bit counter represents a selected bank for each instruction. An alternate bank is selected for every procedural call. A group of instructions terminated by a procedure call can be placed into the same bank through the BSL (Bank Selection Logic) and XOR mapping functions (indexing to each

bank). The goal of the BSL is to help each bank place instructions in groups according to the occurrence of procedure call instructions.

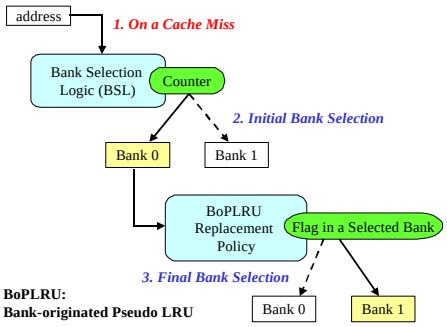


Fig.3. The basic operation of a 2-way TAC scheme.

```
// Bank Selection Logic (BSL)
// initial bank selection according to the counter
If the value of the 1-bit counter = 0
        initial bank = bank 0.
else
        initial bank = bank 1.

// Bank originated Pseudo LRU (BoPLRU)
// final bank selection according to the flag
If the flag of the initial bank = 0
        Replace data of the other bank (say, final bank).
        Set the flag of the initial bank to 1.
If the flag of the initial bank = 1
        Replace data of the initial bank (say, final bank).
        Set the flag of the initial bank to 0.
```

Fig.4. Pseudo code for the BSL and BoPLRU policy.

In Figure 3 and 4, after the BSL selects a bank on a cache miss, the BoPLRU will determine the final bank for updating a line as a correction mechanism by checking the flag for the selected cache line and set the flag of the initial bank.

The first mechanism helps to improve indirect misprediction rates by reducing conflict misses in a small-sized, say less than 512 entries, first stage predictor table.

2.2 The GoStay predict and update rule – The Second Mechanism

The second mechanism helps to reduce indirect misprediction rates by storing two different target addresses in the two stages and selecting one correct target address from two stages according to a designated flag, which represents a short history of indirect branch predictions.

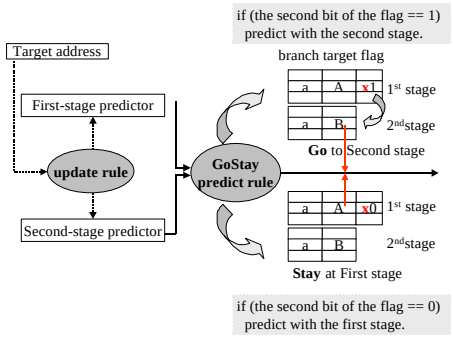


Fig.5. The GoStay predict rule of the second mechanism

Prediction with	Results	Update	Flag (2nd bit)
None	None	Both stages	1
First stage	Correct	None	0
First stage	Incorrect	Both stages	1
Second stage	Correct	None	1
Second stage	Incorrect	Second stage	1

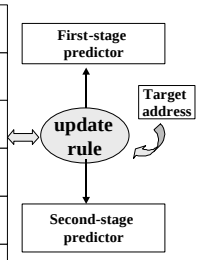


Fig.6. The update rule of the GoStay2 predictor.

In Figure 5, if both stages have the same matched branch address, the prediction will be determined according to the GoStay predict rule: if the second bit of the flag

in the first stage is ‘1’, the prediction will be done with the target address of the second stage (Go). Otherwise, the prediction will be done with the target address of the first stage (Stay). The goal of the GoStay predict rule is to reduce mispredictions caused by wrong target addresses of the second stage (e.g. as in the leaky filter).

Figure 6 shows the update rule after the branch instruction is resolved. There are three cases for updating both stage predictors:

1) If a prediction with a target address in the first stage is correct, only the second bit of the flag is set to ‘0’; and
2) If a prediction with a target address in the second stage is correct, there is no update; and
3) Otherwise, the target addresses of both stages are updated and the second bit of the flag is set to ‘1’.

3 Experimental Environments

An overview of our simulation methodology is described in the following ways: firstly, SPEC95INT and C++ programs were compiled by using a gcc compiler and secondly, the GoS-Sim (branch prediction simulator) ran each executable benchmark with its input data. GoS-Sim was developed by using the Shade and SpixTools. Shade and SpixTools are tracing and profiling tools developed by Sun Microsystems. Shade executes all the program instructions and passes them onto the branch prediction simulator. GoS-Sim not only simulates most indirect branch predictors such as the BTB-based Target Cache and Cascaded Predictor, but it also runs several XOR mapping functions and replacement policies such as the LRU (Least Recently Used) and the Pseudo LRU, etc. The simulator for the proposed predictor was added into the GoS-Sim. Finally, outputs such as misprediction rates, the number of control transfer and procedural call/return instructions, etc. were collected.

Table 1. Benchmark program characteristics

Program	Type	Dynamic instructions	Control Flow Instructions					
			Total		Cond. branches		Indirect branches	
			num.	%	num.	%	num.	%
xlisp	C	189,185K	43,643K	100	30,288K	69.40	4,076K	9.34
ixx	C++	31,830K	7,258K	100	4,731K	65.19	538K	7.42
perl	C	630,281K	130,746K	100	88,162K	67.43	7,656K	5.97
gcc	C	250,495K	53,190K	100	43,711K	82.18	3,177K	5.97
eqn	C++	58,401K	12,080K	100	9,033K	74.78	547K	4.53
m88ksim	C	851K	196K	100	171K	87.02	4K	2.27
go	C	584,163K	82,253K	100	69,163K	84.09	548K	0.67
deltablue	C++	42,149K	9,997K	100	5,122K	51.24	554K	5.54

Table 1 shows the percentages of conditional branches and indirect branches. Five of the SPECint95 programs were used for our simulation –xlisp, perl, gcc, m88ksim, and go. These are the same programs used in [3][7]. The next suite of programs is written in C++ and has been used for investigating the behavior between C and C++ [8][9]. These programs are ixx, eqn, and deltablue. For ‘go’, since the impact of

indirect branch prediction is very low, it will be excluded from all averages in section 4.1. For ‘deltablue’, it also excludes all averages like ‘go’ because of the small-sized (less than 500 lines) program.

4 Experimental Results

In this section, we determined the most effective branch predictor among the BTB (Branch Target Buffer, 4-way set-associative), TC (Target Cache), SF (Strict Filter), and LF (Leaky filter). We implemented hybrid predictors (the TC, SF, and LF) with: the first stage as a BTB; the second stage as a 4-way set-associative table with 512-entry; and the 9-bit (512-entry) pattern-based history register. The main differences in update rules for them are: 1) In case of TC, after resolving an indirect branch, the TC (second stage) can only be updated with its target address; 2) For the SF, only branches into the second stage predictor are allowed if the first predictor mispredicts; 3) For the LF, new second stage entries on first-misses are allowed in addition to the update rule of the SF.

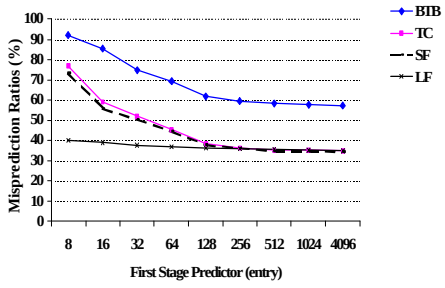


Fig.7. Misprediction rates for C programs.

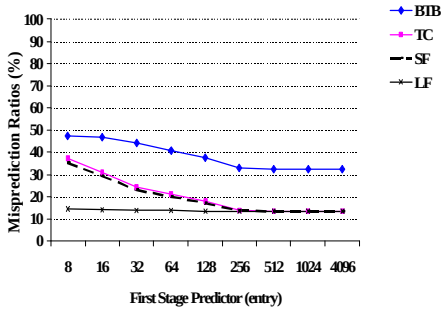


Fig.8. Misprediction rates for C++ programs.

From the Figure 7 and 8, we determined the LF as the most effective indirect predictor if the first stage has a table with less than 512 entries.

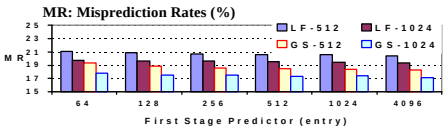
4.1 Misprediction rates between Leaky Filter (LF) and GoStay2 (GoS)

In this section, we compare the indirect misprediction rates between the LF and GoS predictors. In Figure 9(a), the GoS has lower misprediction rates than the LF for most sizes of the BTB (from 64 entries to 4K entries) and the second-stage predictor (512 entries and 1K entries) for all programs. The number of the LF-512 or LF-1024 means the entry size of the second-stage predictor in LF.

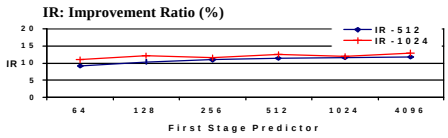
Figure 9 also showed the improvement ratio (IR) between the LF and GoS according to the sizes of the second-stage. We define the IR as: if a misprediction rate of LF-nnn = a and a misprediction rate of GoS-nnn = b, then a/b = 1 + n/100. It shows that ‘a’ has n% more misprediction rates than ‘b’. Where, nnn = 512 or 1024.

So, if n = IR, then IR-nnn = ((a-b)/b)*100 ---------- (1)

In Figure 9(b), in the case of the IR-512, the IR is increased from 14.9% (64-entry of BTB) to 19.35% (4096-entry of BTB). For the IR-1024, the IR is increased by 17.41% to 21.53%.

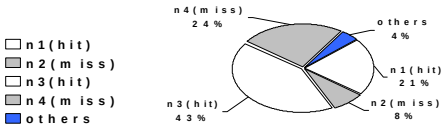


(a) MR for C and C++ programs (harmonic mean).

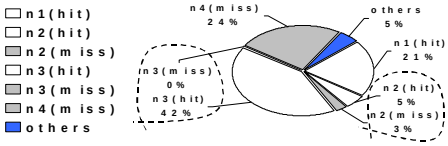


(b) IR (GS over LF) for C and C++ programs (harmonic mean).

Fig.9. Misprediction rates and Improvements ratios (MR and IR).



(a) 128-entry filter of LF with 1024-entry PHT (perl, C)



(b) 128-entry filter of GS with 1024-entry PHT (perl, C)

Fig.10. Analysis of the prediction rates according to the cases.

Most mispredictions of the indirect branches occur when two stages have a simultaneous prediction. There are four cases when both stages have a prediction:

- *addr_both_target_both (n1)*: Both stages have the same target and the target is correct. → Correct prediction in both the LF and GoS;
- *addr_both_target_BTB (n2)*: Both stages have a different target. The first stage has a correct one but the second stage has a wrong one. For the LF, this case leads to a misprediction. But for the GoS, if the second bit of the flag in the first predictor is 0, then this case results in correct prediction. The GoS can reduce misprediction rates considerably by using this predict rule.
- *addr_both_target_PHT (n3)*: Both stages have a different target. The first stage has a wrong one but the second stage has a correct one. In the LF, this case leads to a correct prediction. Meanwhile, for the GoS, if the flag bit is 0, it leads to a misprediction. However, the possibility for this case is very rare, as little as 1%. Otherwise, it is a correct prediction.
- *addr_both_target_none (n4)*: Both stages have a target, but neither target is the correct one. This case always leads to a misprediction in both the LF and GoS.

Figure 10 shows prediction rates according to the cases from the **n1** to **n4** between the LF and GoS with the 128-entry filter (BTB) and the 1024-entry PHT for the 'perl' benchmark program (C program). In Figure 10, 'others' means the prediction rates caused by the cases when one or none of the two stages has a prediction, which can lead to a hit or miss. However, since these rates are small compared to other cases, we ignore them for this section.

The important features provided by Figure 10 are: Figure 10(a) shows that 96% of the total predictions for LF occur within case n1 to n4. Among them, even if there is a correct target in the n2, the predictions caused by the n2 always lead to mispredictions because of the inefficient predict rule. The prediction rate caused by the n2 is 8%, which all lead to misprediction. Figure 10(b) shows that a prediction rate of 95% occurs for GoS in the cases of n1 to n4. However, the differences between the LF and GoS are the hit and miss rates caused by the case of the n2 and n3. In the GoS, more

than half of the predictions (5% out of 8%) lead to a hit (n2 hit) instead of a miss (n2 miss). As a result, the misprediction rates can be improved by using the GoStay predict and update rule. Also, even if the predictions of the n3 leads to a hit in LF, part of the predictions for the n3 can lead to mispredictions in GoS. However, since the misprediction rate caused by this case is very small (0% in Figure 6(b)), we can disregard the misprediction rates caused by the n3.

5 Conclusion

For indirect two-stage hybrid branch predictors, the leaky filter was found to be the most effective one. However, the accuracy of this predictor is affected by two factors: The conflict misses for small-sized tables (say, less than 512 entries) considerably degrade the misprediction rate. The other factor is inefficient predict and update rules.

In order to resolve these problems, we have presented new branch architecture, the GoStay2 predictor, which has two mechanisms that are different from the other hybrid branch predictors. The first mechanism is a new cache scheme, TAC, employed as the first stage to reduce conflict misses. The second mechanism is the GoStay predict and update rule to reduce the frequency of wrong predictions caused by inefficient predict and update rules. By using these mechanisms, the GoStay2 reduces indirect misprediction rate of a 64-entry to 4K-entry BTB (with a 512- or 1K-entry PHT) by 14.9% to 21.53% compared to the Cascaded predictor (with leaky filter).

References

1. Tse-Yu Yeh and Yale N. Patt: A comparison of dynamic branch predictors that use two levels of branch history, ISCA, pages 257-266, 1993.
2. John Kalamatianos and David R. Kaeli: Predicting Indirect Branches via Data Compression, IEEE, MICRO 31, 1998.
3. Karel Driesen and Urs Hölzle: The Cascaded Predictor: Economical and Adaptive Branch Target Prediction, IEEE Micro 31, 1998.
4. Po-Yung Chang, Eric Hao, and Yale N. Patt: Target Prediction for Indirect Jumps, Proceedings of the 24th ISCA, Denver, June 1997.
5. Yul Chu and M. R. Ito: The 2-way Thrashing-Avoidance Cache (TAC): An Efficient Instruction Cache Scheme for Object-Oriented Languages, Proceedings of the 17th IEEE ICCD, Austin, Texas, September 2000.
6. F. Bodin, A. Seznec: Skewed-associativity enhances performance predictability, Proc. of the 22nd ISCA, Santa-Margharita, June 1995.
7. R. Radhakrishnan and L. John: Execution Characteristics of Object-oriented Programs on the UltraSPARC-II, Proc. of the 5th Int. Conf. on High Performance Computing, Dec. 1998.
8. B. Calder, D. Grunwald, and B. Zorn: Quantifying Behavioral Differences Between C and C++ Programs, *Journal of Programming languages*, Vol. 2, No. 4, pp. 313-351, 1994.
9. Urs Hölzle and David Ungar: Do object-oriented languages need special hardware support? Technical Report TRCS 94-21, Department of Computer Science, University of California, Santa Barbara, November 1994.
10. R. F. Cmelik and D. Keppel: Shade: A Fast Instruction-Set Simulator for Execution Profiling, Sun Microsystems Laboratories, Technical Report SMLITR-93-12, 1993.