Prediction of Backward Branches by Pattern Detection

Muhammad Aurangzeb, Muhammad Ahmad Ghazali, Farooq Ahmed, Fakhir Shaheen
National University of Computer and Emerging Sciences, Lahore, Pakistan.
[aurangzeb, ahmad.ghazali, mscs00-1054, mscs115]@nu.edu.pk

Abstract

A number of branch prediction schemes have been proposed in literature. In order to improve the prediction probability, all of them address the problem of accurate branch prediction in various ways. In this paper we have proposed a predictor which can accurately predict a branch which is once registered with it.

1. Introduction

Accurate branch prediction is essential for yielding better performance in a pipelined processor. However, the previously known techniques for branch prediction have been less accurate than desired. The problem seems to be inherent in the basic approach – the prediction. Prediction is a guess – a speculation – that cannot be guaranteed to be correct. No matter, how sophisticated the technique may be, there is always the possibility that the prediction may turn out to be false, leading to undesirable stalls in the pipeline. In order to give more accurate branch prediction, Vintan et al. [1] have recently suggested an alternative to this scheme in the form of pre-computed branches. Heil et al. [2] proposed another approach for branch prediction that was based on the idea of data value prediction. They used the differences between the branch operands to improve the branch prediction accuracy.

Based upon the two aforementioned techniques, we have proposed a predictor named Backward Branch Predictor (BBP) that is capable of accurately predicting a branch instruction that is once registered with it.

Rest of this paper is organized as follows. Section 2 presents the background and briefly describes the two techniques mentioned above. Section 3 presents the design of BBP, while section 4 concludes the paper.

2. Background

In this section, we are going to review the two branch prediction mechanisms, namely: Pre-computed branches [1] and Branch Difference Predictor [2].

2.1. Pre-computed Branches

The idea of pre-computed branches is to actually determine the outcome of the branch instead of relying on the history information for the prediction. The branch outcome is calculated and cached for further use as soon as all the operands of the branches are known. When branch instruction is actually encountered, it is searched in the cache and if a hit occurs, the pre-computed outcome of the branch is used to fetch the branch target. Since, the actual branch outcome was pre-computed, it is guaranteed that the next instruction fetch as the branch target instruction will be the correct one, ruling out pipeline stalls.

The pre-computed branch algorithm works using additional hardware in form of a Prediction Table (PT) and an extended register file named Register Unit (RU). Both PT and RU comprise of several fields, explained in Table 2.1.1 and 2.1.2 respectively.

Every branch instruction is initially predicted as NOT TAKEN. Once the branch is executed for the first time, a corresponding entry is made into the PT and the fields in the PT are updated. For every entry in PT, two fields PC1 and PC2 are maintained that keep track of the last two instructions that produced the branch operands. Next time, when a register is updated, which happens to be one of the source operands of a branch instruction, the corresponding branch instruction is re-computed and the outcome is stored in the PRED field in PT. In order to determine whether an instruction changes the value of one of the branch operands or not, a search is made into PT on the Program Counter (PC) of instruction.

Table 2.1.1 Details of PT data structure

Field Name	Field Description
TAG	High order bits of branch instruction's PC
PC1, PC2	PCs of the instructions producing branch's operands
OPC	Branch instruction's opcode
OP1, OP2	Register names of the branch operands
PRED	Branch outcome (Taken or Not-Taken)
LRU	Least recently used field

Table 2.1.2 Details of RU data structure

Field Name	Field Description
Value	Data value stored in register
LDPC	The most recent instruction label (PC) that wrote in the register
RC	Reference counter indicating number of branch instructions stored in PT that have this particular register as one of the operands

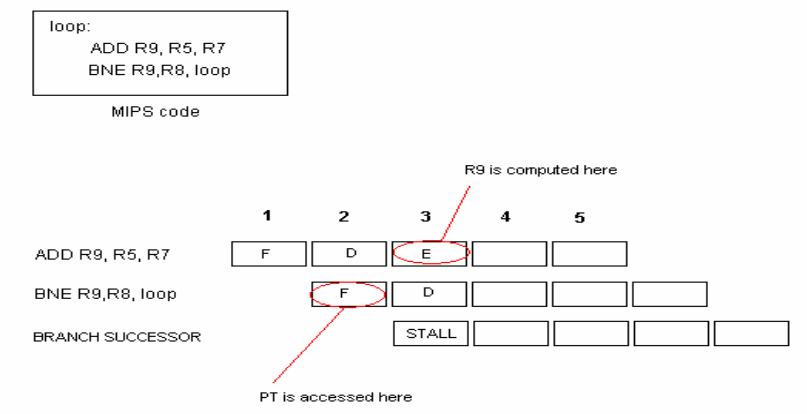

Figure 2.1.1 Figure illustrating RAW hazard problem with pre-computation algorithm for the MIPS code shown in box. R9 is computed at the end of cycle 3, where as PT needs to be accessed at the end of cycle 2 to fetch the branch successor in cycle 3

If this search results in a match with PC1 or PC2 then the instruction updates one of the branch operands and hence the corresponding branch is re-computed. In order to avoid searching of PT after execution of every instruction, a reference counter (RC) is maintained in RU for every register. RC indicates the number of entries in the PT, of which this particular register is one of the operands. Thus, PT is searched only for non-branch instructions updating registers having RC > 0.

The above approach helps in accurately determining the outcome of the branch. However, even the idea of pre-computation does not promise 100 % accuracy. The pre-computed branch algorithm can have miss-predictions, which are basically due to the following three reasons:

1. Every branch is initially predicted as not-taken. This initial prediction can be wrong.
2. There can be capacity misses in the prediction table.
3. The idea of pre-computation may not work if the branch instruction and the corresponding operand producer are too close in the program follow to trigger a Read after Write (RAW) hazard.

Last of the above three reasons is of serious concern here, because there are many instances when such kind of situations occur. The problem is illustrated in Figure 2.1.1 for a classic five-stage pipeline. Value of R9 is computed at the end of cycle 3, whereas PT table needs to be accessed in cycle 2 to fetch the correct branch successor in cycle 3. Even with some data forwarding, stall of 1 cycle will occur. If no data forwarding is used, then upto 3-cycle stall may occur.

One of the ways to avoid this problem is to use a conventional predictor for such cases instead of pre-computation scheme. For all the other cases Pre-computation scheme can be used giving accurate prediction except for this one. Another way to resolve this problem is to use some sort of instruction scheduling. However, no published work exists so far that has shown a feasible solution to this problem in this scheme.

2.2. Branch Difference Predictor (BDP)

Data value predictors are based on the premise that for many branches the previous branch outcomes and the current PC are insufficient. Therefore, these predictors use data values to improve branch prediction accuracy. Heil et al. [2] proposed one such kind of predictor called Branch Difference Predictor (BDP). Because a large number of branches depend either on the sign of difference between the two register values used in the branch instruction or whether the difference is zero. Therefore, instead of storing the data values of the branches, BDP stores the differences between the two values.

There are three main components of BDP: Backing predictor, Rare Event Predictor (REP) and Value History Table (VHT). VHT stores the differences between the values of branches. This table is indexed by the branch PC and provides the difference history of the two sources used in the branch instruction. However, the number of differences to be stored can still be very large. So, BDP stores these differences only for rare branches and normal branches are predicted by a traditional predictor (called Backing predictor). The Rare Event Predictor (REP), which is a tagged cache-like structure, is used to predict these rare branches. The values in REP are only added when the backing predictor mispredicts a branch. Otherwise, if the backing predictor predicts a branch correctly then REP is not updated. The overall structure of BDP is shown below in Figure 2.2.1.

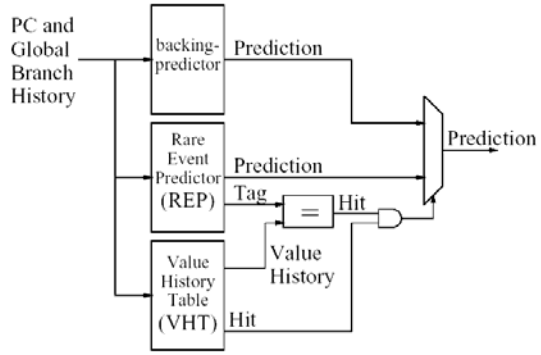

Figure 2.2.1 Structure of BDP [2]

REP is accessed in parallel with VHT using the branch PC and Global Branch History (GBH). VHT is used as a selector to select between the predictions made by the backing predictor and REP. The value history is compared with the Tag resulting from the REP. If the two values are equal then the prediction from REP is used, otherwise prediction of the backing predictor is used. The process of accessing VHT and REP is summarized below in Figure 2.2.2.

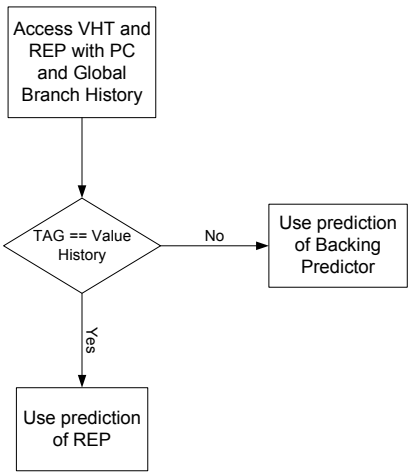

Figure 2.2.2 Accessing VHT and REP

An entry is placed in the REP only when it correctly predicts a branch which was mispredicted by the backing predictor. The counters of the new entry are placed in weakly taken or weakly not taken state depending on whether the branch was taken or not. This method of updating REP is shown below in Figure 2.2.3.

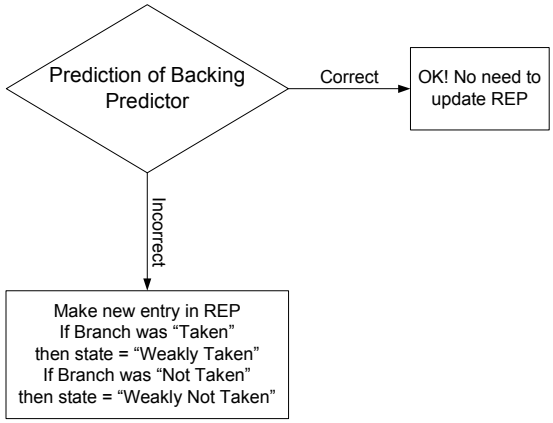

Figure 2.2.3 Updating REP

3. Backward Branch Predictor (BBP)

In this section we propose the Backward Branch Predictor (BBP). Which would work in superposition to the conventional predictor or any other predictor proposed in [1], [2] and [3].

3.1. Basic Design Philosophy

The location of BBP in overall predictor design is shown in Figure 3.1.1. The proposed BBP works in parallel with the two level conventional branch predictor and any other special predictor like Rare Event Predictor (REP) [2]. Whenever a branch is encountered for the first time, by examining current PC

and the branch target address it is determined that whether the branch is backward. The subject branch is predicted by using the mechanism as explained in [1], [2] and [3]. However, if it is classified as a backward branch it is enlisted in BBP; details are explained in the next section.

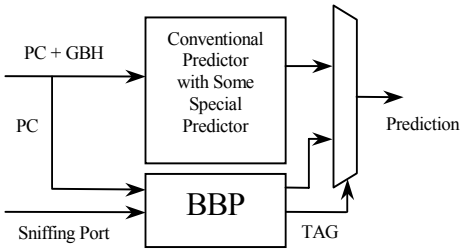

Figure 3.1.1 Location of BBP in overall layout of branch prediction module

If the subject branch is actually taken then BBP determines the patterns in the variations of the two operands in the subject branch instruction through the sniffing port of BBP as the processor continues its execution of the block of instructions between the target address and the subject branch instruction. This time also the branch is predicted like it is predicted for the first time. These patterns are stored in BBP and the instruction is registered into the BBP. Now, whenever a registered branch instruction is fetched it is predicted by BBP. If the prediction of BBP remains correct its registration level is improved till a maximum level and if due to one reason or the other, at any stage, the prediction goes wrong its status is reverted back to enlisted status. Ultimately when the backward branch is correctly predicted as not taken the branch is vetted. Through out the execution of the task the operands of the branch enlisted/registered/vetted are monitored and the moment the control comes on a branch registered/vetted its prediction is made by BBP. By this stage it may be evident that the design philosophy of BBP is to get utmost prediction accuracy for loops but the same idea may be further extended.

3.2. Detailed Layout

The detailed layout of BBP is shown in Figure 3.2.1. Once a branch is encountered for the first time and it is determined to be a back ward branch it is enlisted in the Pattern Table within BBP. The detailed lay out of the Pattern Table is shown in Figure 3.2.2. The said branch is enlisted in the Pattern Table directly against its PC or hashed[*] in it.

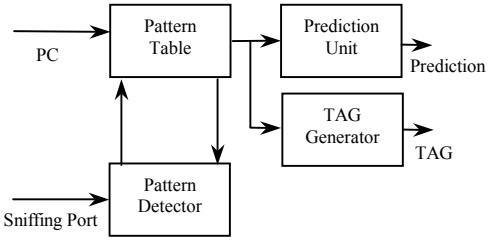

Figure 3.2.1 Detailed layout of BBP

Status (3 bits)	PC (≤32 bits)	Operand 1 (5 bits)	Value 1 (32 bits)	Operand 2 (5 bits)	Value 2 (32 bits)	Pattern 1	Pattern 2

Figure 3.2.2 Layout of Pattern Table

Significance of the Status bits is shown in Figure 3.2.3. As shown, initially the status of an entry is marked as Invalid. When a backward branch is detected its status is changed to Enlisted. Operands and their values at the stage of Instruction Decode[**] (ID) stage of the branch instruction are stored against the PC. Now as the control comes back while executing the code BBP detects the pattern which the two operands are following by sniffing the code under execution through its Sniffing Port. When the control comes to an enlisted instruction for the second time its status is converted to registered and pattern detected by the Pattern Detector is stored in the table against the PC.

Status (3 bits)	Status of the Corresponding Branch Instruction
000	*Invalid (Initial Status)*
001	*Enlisted*
010 to 110	*Registered (with 010 as least level and 110 maximum level)*
111	*Vetted*

Figure 3.2.3 Significance of Status bits

This time the prediction is made by BBP and Tag is set by the Tag Generator. The tag will be set if the entry's status is either Registered or Vetted. The

[*] Here we are not concerned with bit optimization.

[**] The basic MIPS processor pipeline with forwarding may be considered.

prediction is selected by the multiplexer circuitry shown in Fig 3.1.1. If the prediction works well its registration level is improved*. Finally if the branch is correctly predicted as Not Taken (NT), its status is changed to Vetted**. Now during the execution of the code whenever control comes to such a branch instruction BBP is used to predict it.

The BBP as proposed above works in quite a deterministic fashion and capable of accurately predicting backward branches registered with it. To further strengthen our claim we have quoted a few examples.

3.3. Working Examples of BBP

Following are a few examples to elaborate the above-mentioned BBP.

Example 1
Consider the following high-level code:

```
void Loop1(){
        long var1 = 100, var2 = 20;
        for(long x = 0; x != var1; x++){
                var2 = var2 * x;
        }
}
```

Following is its equivalent MIPS64 assembly language:

```
        LD      R1,     #100    ; var1
        LD      R2,     #20     ; var2
        LD      R3,     #0      ; x
MAIN_LOOP:
        DMUL    R2, R2, R3
        ADD.D   R3,     #1
        BNE     R1, R3, MAIN_LOOP
```

When the branch instruction is encountered for the first time the following entry is made in the Prediction Table of BBP. It should be noted that Value-2 contains the value of the Operand-2 available at the ID stage of the branch instruction.

Status	PC	Operand 1	Value 1	Operand 2	Value 2	Pattern 1	Pattern 2
001	PC	R1	100	R3	0	-	-

When the control comes to the branch instruction for the second time the following would be the entry in the table. Again the values are those available at ID stage of the branch instruction. Further the *Pattern Detector* has detected that the Operand-1 remains fixed during the execution of the code and the Operand-2 is incremented by 1. This time the tag is set and the branch was correctly predicted as taken by BBP.

Status	PC	Operand 1	Value 1	Operand 2	Value 2	Pattern 1	Pattern 2
010	PC	R1	100	R3	1	Fixed	+1

The execution of the code goes on and every time the branch instruction is executed it is correctly predicted by BBP. Following would be the entry for the branch instruction when the control comes to it for the last time.

Status	PC	Operand 1	Value 1	Operand 2	Value 2	Pattern 1	Pattern 2
110	PC	R1	100	R3	99	Fixed	+1

BBP will correctly predict it as not taken and the corresponding entry in the table would be as following.

Status	PC	Operand 1	Value 1	Operand 2	Value 2	Pattern 1	Pattern 2
111	PC	R1	100	R3	100	Fixed	+1

The *Sniffing Port* will keep track of the two operands and whenever the control comes on the same branch for the next time it is directly predicted by BBP.

* Though we have proposed this scheme for simple loops yet the idea may also be extended to complex situations by inducing artificial intelligence in the pattern detector. And in such situations in case prediction goes wrong, it is suggested that the status may be reverted back to *Enlisted*.

** (Above note is applicable)

Example 2
Consider the following high-level code.

```
void Loop2(){
        long var1 = 0, var2 = 50;
        do{     var1++;
                var2--;
        }while(var1 != var2);
}
```

Following is its equivalent MIPS64 assembly language:

```
        LD      R1,     #0      ; var1
        LD      R2,     #50     ; var2
MAIN_LOOP:
        ADD.D   R1,     #1
        SUB.D   R2,     #-1
        BNE     R1, R2, MAIN_LOOP
```

When the branch instruction is encountered for the first time the following entry is made in the Prediction Table of BBP.

Status	PC	Operand 1	Value 1	Operand 2	Value 2	Pattern 1	Pattern 2
001	PC	R1	1	R2	50	-	-

It should be noted that Value-2 contains the value of the Operand-2 available at the ID stage of the branch instruction.

When the control comes to the branch instruction for the second time the following would be the entry in the table. Again the values are those available at ID stage of the branch instruction. Further the *Pattern Detector* has detected that the Operand-1 incremented by 1 during the execution of the code and the Operand-2 is decremented by 1. This time the tag is set and the branch was correctly predicted as taken by BBP.

Status	PC	Operand 1	Value 1	Operand 2	Value 2	Pattern 1	Pattern 2
010	PC	R1	2	R2	49	+1	-1

The execution of the code goes on and every time the branch instruction is executed it is correctly predicted by BBP. Following would be the entry for the branch instruction when the control comes to it for the last time.

Status	PC	Operand 1	Value 1	Operand 2	Value 2	Pattern 1	Pattern 2
110	PC	R1	25	R2	24	+1	-1

BBP will correctly predict it as not taken and the corresponding entry in the table would be as following.

Status	PC	Operand 1	Value 1	Operand 2	Value 2	Pattern 1	Pattern 2
111	PC	R1	25	R2	25	+1	-1

The *Sniffing Port* will keep the track of the two operands and whenever the control comes on the same branch for the next time it is directly predicted by BBP.

4. Conclusion

Apart from conventional branch predictors there are many other predictors for some special events and for multiple branches. The one proposed in this paper is for backward branches. The proposed predictor can work along with any of the above predictors and may be used to accurately predict the backward branches. Performance of the proposed detector is evident from the motivating examples mentioned in section 3 yet some simulation may also be performed to further analyze its utilization. The proposed predictor may be improved to predict any kind of branches by inducing artificial intelligence in the pattern detector unit.

5. References

[1] L. Vintan, M. Sbera, I. Mihu, and A. Florea, "An alternative to branch prediction: pre-computed branches", *ACM SIGARCH Computer Architecture News*, Volume 31, Issue 3, June 2003, Pages: 20-29.

[2] Timothy H. Heil, Zak Smith, and J. E. Smith, "Improving branch predictors by correlating on data values", *Proceedings of the 32nd annual ACM/IEEE international symposium on Microarchitecture*, November 16-18, 1999, Haifa, Israel, Pages: 28-37.

[3] R. Rakvic, B. Black, and J.P. Shen, "Completion time multiple branch prediction for enhancing trace cache performance", *Proceedings of the 27th annual international symposium on Computer Architecture*, Vancouver, Canada, 2000, Pages: 47-58.