Brief Contributions

Improving the Accuracy of History-Based Branch Prediction

David R. Kaeli, *Member, IEEE*, and Philip G. Emma, *Fellow, IEEE*

Abstract—In this paper, we present mechanisms that improve the accuracy and performance of history-based branch prediction. By studying the characteristics of the decision structures present in high-level languages, two mechanisms are proposed that reduce the number of wrong predictions made by a branch target buffer (BTB). Execution-driven modeling is used to evaluate the improvement in branch prediction accuracy, as well as the reduction in overall program execution.

Index Terms—History-based branch prediction, BTB, high-level language, execution-driven simulation, call/return stack, case block table.

1 Introduction

THIS paper describes mechanisms that increase the accuracy of history-based branch prediction. Past research on history-based branch prediction has focused on improving the accuracy of the prediction rate by optimizing the prediction algorithm or by varying the amount of recorded history [1], [2], [3], [4], [5], [6], [7], [8], [9]. Instead, the focus here is on the high-level language decision structures responsible for wrong branch predictions. Two mechanisms are presented that reduce the number of wrong branch predictions caused by these structures.

2 High-Level Language Branch Constructs

Every computer program contains some type of branching construct. High-level languages (e.g., Fortran, C, C++, Pascal) offer a rich syntax that enables the programmer to implement these constructs. Each of these primitives exhibits a unique execution behavior when compiled [16].

We have identified two branch constructs that pose problems to history-based branch prediction. These two branch classes are *case switch blocks* and *procedure returns*. We will inspect these two constructs from a programming perspective and illustrate why they can produce incorrect branch predictions.

2.1 Call/Return Branch Constructs

One key problem that has been overlooked in past work is the correctness of the branch target address that is stored in the BTB [5]. A BTB must be able to predict the direction of the branch (taken or not taken), and must also produce the correct target address. This allows instruction fetching to be redirected to the appropriate target address. The return from subroutine produces many wrong target address predictions [10].

To illustrate why incorrect predictions are produced, we inspect assembly code generated from a C program (see Fig. 1). The program invokes the PRINT subroutine at instruction addresses 100 and 130. Executing the program, at address 100 an unconditional branch (a call) is made to subroutine PRINT at address 500. An entry is made in the BTB for address 100 with a target of 500. On the return from PRINT an entry is made in the BTB with an branch address of 600 and a target of 110. From instruction 130, another call is is made to subroutine PRINT at address 500. An entry is made in the BTB for address 130 with a target 500. This time on the return from PRINT, a valid entry in the BTB is found (at address 600) and a prediction is made that instruction fetching should be redirected to address 110. Of course, this is not correct. The correct return point is to address 140.

Instruction Address	Instruction	Label	Instruction Address	Instruction
100	call @PRINT	PRINT:	500	.
110	.		510	.
120	.		.	.
130	call @PRINT		.	.
140	.		.	.
.	.		.	.
.	.		600	return

Fig. 1. An example of a call/return.

Later we will describe a mechanism that reduces the number of wrong branch target predictions due to subroutine returns.

2.2 Case Switch Construct

Frequently changing branch history accounts for a large number of the wrong BTB predictions [13]. One high-level language branching construct that exhibits this kind of erratic behavior is the case switch found in C and C++. This construct implements a multiway branch based on the value of a single variable. The value of this variable determines which of several paths is followed through the block. Case switches are commonly found in parsing code (e.g., in compilers). Fig. 2 shows a source code example of a case switch block.

```
switch (c)
        {
          case ‘x’:
                d = ‘=‘;   /* subcase = */
                break;
          case ‘y’:
                d = ‘?’;   /* subcase ? */
                break;
          case ‘z’:
                d = ‘/’;   /* subcase / */
                break;
          default:
                break;
        }
```

Fig. 2. HLL example of a case switch.

An assembly code version of the case switch construct is shown in Fig. 3. The switch variable **c** is loaded into register R1. Then a series of compares (subcases) are performed. If a match is found in any of the subcases, the variable **d** is changed accordingly. If no match is found, **d** remains unchanged.

• *D.R. Kaeli is with the Electrical and Computer Engineering Department, Northeastern University, Boston, MA 02115. E-mail: kaeli@ece.neu.edu.*
• *P.G. Emma is with the IBM T.J. Watson Research Center, Yorktown Height, NY 10598. E-mail: pemma@watson.ibm.com.*

Manuscript received 26 Jan. 1993; revised 23 Sept. 1996.
For information on obtaining reprints of this article, please send e-mail to: transcom@computer.org, and reference IEEECS Log Number C96296.

```
Label          Instruction   Instruction
               Address

               100           MOVE R1,(C)
               110           CMP R1,’x’
               120           BNE 150
SUBCASE X:     130           MOVE (D),’=‘
               140           B 220
               150           CMP R1,’y’
               160           BNE 190
SUBCASE Y:     170           MOVE (D),’?’
               180           B 220
               190           CMP R1,’z’
               200           BNE 220
SUBCASE Z:     210           MOVE (D),’/’
               220           .
               230           .
```

Fig. 3. Assembly code example of a case switch.

As the value of variable **c** changes, it is clear that the behavior of the case switch construct will also change. Further, the path length necessary to resolve the case switch construct can be quite long (though a jump table can also be used in some cases). In general, many conditional branches will be executed to satisfy one pass through the block.

Later we describe a new mechanism that detects these blocks. The mechanism eliminates a number of the wrong predictions made by the BTB. More importantly, this new mechanism can reduce the overall path length of a program by almost 10%.

2.3 Call/Return Stack Pair

To keep the BTB from making an incorrect branch target prediction when a subroutine is called from a number of different program locations, a set of stacks is introduced that detects and corrects the situation. A single stack could also be used if explicit return instructions are present in the architecture or it can be insured that calls and returns will always be paired. Stacks maintain temporal order and they are used here to pair a return instruction with the corresponding calling point. By using a pair of stacks (a call stack and a return stack), this branching behavior can be identified before a wrong target address prediction is made [11]. In addition to the two stacks, each BTB entry must be augmented with a *subroutine return bit* (SR) that is used to indicate that an entry is special.

The programming example in Fig. 1 is again used to describe our design. The call / return stack pair functions as follows:

1) When the call at address 100 is executed, an entry is created in the BTB with a branch address of 100 and a branch target address of 500. The target address of the call (500) is pushed onto the call stack, and the return address 110 (the address that is sequential to the call instruction) is pushed onto the return stack.
2) When the return is executed at 600 with target address 110, an entry in the BTB is made with a branch address of 600 and a branch target address of 110. In parallel, the return stack is accessed to see if it has an entry matching 110. In this example, there exists a match, so an inference is made that the branch at address 600 is a subroutine return (remember that we have not decoded this instruction yet so even if an explicit return instruction is provided by the architecture, we still need to infer its presence). The corresponding entry in the call stack (500) designates the calling address of the subroutine. The corresponding BTB entry is modified as follows: i) the branch target address is changed to 500, and ii) the subroutine return bit for this entry is turned on to indicate that the branch is a subroutine return, and that the target address field contains a subroutine calling address.
3) When the call at address 130 is executed, the address (130) and the target address (500) are sent to the BTB. The target address of the call (500), is pushed onto the call stack, and the return address (140) is pushed onto the return stack.
4) When instruction address 600 is subsequently fetched, the BTB finds an entry for address 600 (the return) with the subroutine return bit turned on. Whenever an entry is found in the BTB with the subroutine return bit turned on, a lookup is performed in the call stack using the subroutine calling address found in the branch target field in the BTB (500). In this example, the call stack does find a match, so the corresponding entry in the return stack is the subroutine return point. This entry is read out and used as the branch target address. The return stack provides the correct return address (140) instead of the historical return address (110). While two sequential accesses are required using this method, the performance degradation of using an incorrect target address greatly outweights the time for the extra table access.

Since a stack preserves the order in which subroutine calls are made, the entry closest to the top of the stack is used if more than one match is found in the call stack. If a subroutine contains multiple return points, and if the return point changes between two successive calls, a new BTB entry is created. The correct target address will then be provided on subsequent subroutine returns.

2.4 A Case Block Table

The case switch HLL branch construct generates frequent mispredictions in a BTB. This occurs because the case switch construct exhibits a changing branch behavior (e.g., changing from taken to not-taken, and then back to taken).

A case switch HLL construct is henceforth referred to as a *case block*. The case block is used to implement a multiway branch, based on the value of a variable. A new mechanism called the *case block table* (CBT) implements the multi-way branch directly, and eliminates the execution of the block. (This is called *case block folding* [12].)

Current branch prediction mechanisms perform poorly within the body of case blocks [13]. A case block table attempts to relieve the branch prediction mechanism from making incorrect predictions within case blocks. The CBT records past history for each case block by associating the eventual branch target with each particular value of the case block variable. For each case block, there will be one unique CBT-entry per unique operand value. A CBT-entry is made up of the case block starting address, the associated operand value, and the resulting target address.

The CBT is accessed when the beginning of a case block is recognized. The table is searched for a match on the pair [case block starting address, case block operand value]. When an appropriate match is found, an unconditional branch is made to the corresponding target address. Thus, the CBT causes the entire case block decision structure (typically containing many conditional branches) to be bypassed (folded). When using this mechanism, the wrong predictions typically encountered by the BTB are eliminated. The entire case block is resolved in one instruction execution time. A CBT prediction prevents the BTB from attempting to redirect the instruction fetching.

To allow the table to identify where a case block begins and ends, two new instructions must be added to the CPU instruction set: 1) BEGINCASE, and 2) ENDCASE.

The BEGINCASE has the semantics:

BEGINCASE (VARADDR),

where VARADDR specifies the name of the variable that is used to determine the outcome of the case block. The instruction does nothing functionally; it only serves to identify the beginning of the case block and the case variable. This instruction causes the block to be folded in the event of a CBT match.

The ENDCASE has the semantics:

ENDCASE (TARGADDR),

where TARGADDR specifies the final target address of the case block (and thus branches to the end of the block). The ENDCASE does nothing functionally except to denote the exit point of the case block.

To demonstrate how a case block table operates, the program example provided in Fig. 3 must be modified slightly. In Fig. 4, a recompiled version of the sample program is shown which uses the BEGINCASE and ENDCASE instructions.

```
Label          Instruction    Instruction
               Address

               100            BEGINCASE (C)
               110            MOVE R1,(C)
               120            CMP R1,’x’
               130            BNE 160
SUBCASE X:     140            MOVE (D),’=‘
               150            B 230
               160            CMP R1,’y’
               170            BNE 200
SUBCASE Y:     180            MOVE (D),’?’
               190            B 230
               200            CMP R1,’z’
               210            BNE 230
SUBCASE Z:     220            MOVE (D),’/’
               230            ENDCASE (240)
               240            .
```

Fig. 4. Modified assembly code of case switch example.

For this example, let the variable **c** take on the values “y,” “x,” and “y,” for three successive iterations of the case block. On the first iteration, **c** = “y.” An entry is made in the case block table. The entry contains:

1) the address of the BEGINCASE (100),
2) the value of the case variable (“y”), and
3) the address of the subcase (180).

No entries for any of the taken branches within the case block are made in the BTB. This should prevent the BTB from making a number of incorrect predictions (these branches typically exhibit an unpredictable behavior). It should also be noted that subcases contained in case blocks typically contain few branches. The BTB is enabled once the ENDCASE is executed.

For **c** = “x,” a similar entry is made in the CBT. For **c** = “y” (on the third iteration through the case block), a match is found in the CBT as soon as the BEGINCASE instruction is executed (BEGINCASE = 100 and case variable = “y”). An unconditional branch is immediately taken to the target address found in the CBT (180) (this branch is predicted by the CBT if history for this value of the case variable is found).

While a CBT redirection is delayed until the value of the case variable is resolved, the redirection will be to the correct subcase address. A first level of optimization to the CBT scheme would be to have the compiler move the setting of the case variable as early as possible or provide data forwarding hardware to feed the result directly to the instruction prefetching logic. A second level of optimization could be to detect a pattern of values that a case variable takes on (e.g., detecting the occurrence of word patterns in a parser) and then provide prediction of the next value of the case variable.

Case blocks are commonly found in parsing code. The above code is typical of case switches found throughout the code of the gcc compiler. This example demonstrates two of the merits of the CBT. First, since a BTB performs poorly while attempting to predict branches contained in a case block, BTB prediction and updating has been disabled. Second, after the history of a particular value of the case variable has been recorded in the CBT, a direct branch can be made to the appropriate subcase on subsequent executions when using the same value. The result is that many instructions need not be executed (especially the costly conditional branches encountered to resolve the instance of the case variable).

While a jump table could be used (a jump table provides a table of offsets for each of the subcases and is indexed by the case variable), the CBT would still be useful for predicting the branch target contained in the jump table, very much in the same way as described in the above example. But in this case, the reduction in the execution path would not be as significant.

3 SIMULATION RESULTS

We studied traces of SPEC benchmarks [15] as run through a very simple (1-bit counter) 4,096-entry BTB model. In our first model we add a pair of call/return stacks. Stack depths of five and 10 were simulated. Fig. 5 lists the results.

By adding a five-entry pair of stacks, the percentage of wrong predictions is reduced by up to 4.6% (an improvement of 31.5%). For many of the benchmarks, a five-entry stack pair combined with a 128-entry BTB predicted more branches correctly than a stackless-BTB with 4,096 entries [13] (e.g., li simulated with a 128-entry BTB and a pair of five-entry stacks showed a 33% improvement over a stackless 4,096-entry BTB).

Doubling the stack depth to 10 entries only shows a difference in the li and gcc benchmarks. Increasing the depth past 10 did not show any further improvement for the traces used. Another encouraging result is that the performance of the four benchmarks that were previously generating good results were not adversely affected by the introduction of the stacks.

benchmark	stackless	five-entry	10-entry
nasa	0.0	0.0	0.0
tomcatv	0.4	0.0	0.0
matrix	0.0	0.0	0.0
spice	6.2	4.9	4.9
eqntott	7.1	7.0	7.0
fpppp	9.3	8.0	8.0
doduc	14.8	13.4	13.4
li	14.6	10.0	8.9
gcc	14.5	12.5	11.2
espresso	23.1	20.2	20.2

Fig. 5. Percent wrong predictions (includes all predictions).

Next we combine a case block table with the same 4,096-entry BTB. The CBT modeled is infinite in size (we are mainly interested in showing the potential of the CBT here versus suggesting any particular organization).

From the applications studied in the SPEC benchmark suite, the three that produced the largest percentage of wrong predictions were selected for study using this model (gcc, li, and espresso). These three also contain the largest number of case blocks.

Fig. 6 shows that while the number of wrong branch predictions decreased slightly, the overall number of instructions executed was reduced by as much as 9%. The wrong predictions results are computed as the change in the total number of wrong branch predictions when adding a CBT. Similarly, the path-length results are computed as the reduction in the total number of dynamic instructions that are executed when adding a Case Block Table.

By folding the entire case block into a single instruction (as is done by this mechanism), large increases in application performance should be realized. Note that the overall performance gain should be amplified since the code that has been eliminated contains a large percentage of branch instructions.

benchmark	**wrong branch predictions**	**path length reduction**
li	4.5	9.0
gcc	3.0	3.5
espresso	2.0	1.9

Fig. 6. Percent decrease over a CBT-less design.

4 CONCLUSIONS

This paper has described two methods of improving the accuracy of history-based branch prediction mechanisms:

1) a call / return stack, and
2) a case block table.

Performance improvements were presented which showed that a five-entry pair of call / return stacks can reduce the number of wrong branch predictions by 31.5% and that a case block table can reduce the number of wrong branch predictions by as much as 4.5%, while reducing the number of dynamic instructions executed in a program by as much as 9%.

This paper addressed the problem of reducing the number of wrong predictions made by history-based branch prediction mechanisms. By inspecting the underlying HLL branching constructs, mechanisms can be devised that further reduce the number of wrong predictions.

The inability of history-based branch prediction mechanisms to reliably predict the if / then / else branch construct warrants further research. Most if / then / else clauses exhibit highly erratic behavior. One approach is to identify the sequence of taken / not taken branches leading up to a branch and use this in the prediction algorithm [4], [8]. While this technique has shown promise, many branches are still incorrectly predicted. A simpler approach may be to disable all branch prediction when this class of construct is encountered, much in the same way a case block table overrides any branch target buffer predictions. Other approaches, such as multilevel tables [17] or a DDBT [14], should also be explored.

ACKNOWLEDGMENTS

The authors would like to thank Juho Tang for providing the tracing tool used in this work, Josh Knight and Tom Puzak for their insightful discussions on branch prediction, Yale Patt for his helpful editorial comments.

REFERENCES

[1] C. Young, N. Gloy, and M.D. Smith, “A Comparative Analysis of Schemes for Correlated Branch Prediction,” *Proc. 22nd Int’l Symp. Computer Architecture*, pp. 276-286, Santa Margherita, Italy, June 1995.

[2] B. Calder and D. Grunwald, “Fast and Accurate Instruction Fetch and Branch Prediction,” *Proc. 21st Int’l Symp. Computer Architecture*, pp. 2-11, Chicago, Apr. 1994.

[3] T.Y. Yeh and Y. Patt, “A Comparison of Dynamic Branch Predictors the Use Two Levels of Branch History,” *Proc. 20th Int’l Symp. Computer Architecture*, pp. 257-266, San Diego, May 1993.

[4] S.T. Pan, K. So, and J.T. Rahmeh, “Improving the Accuracy of Dynamic Branch Prediction Using Branch Correlation,” *Proc. Fifth Int’l Conf. Architectural Support for Programming Languages and Operating Systems*, pp. 76-84, Oct. 1992.

[5] C.H. Perleberg and A.J. Smith, “Branch Target Buffer Design and Optimization,” *IEEE Trans. Computers*, vol. 42, no. 4, pp. 396-412, Apr. 1993.

[6] D.J. Lilja, “Reducing the Branch Penalty in Pipelined Processors,” *Computer*, pp. 47-55, July 1988.

[7] J.E. Smith, “Branch Predictor Using Random Access Memory,” U.S. Patent 4,370,711, Jan. 25, 1983.

[8] J.E. Smith, “A Study of Branch Prediction Strategies,” *Proc. Eighth Symp. Computer Architecture*, pp. 135-148, Minneapolis, May 1981.

[9] C.O. Stjernfeldt, “A Unified Comparison of Branch Prediction Strategies,” MS thesis, Dept. of Electrical and Computer Eng., Northeastern Univ., Jan. 1993.

[10] D.R. Kaeli and P.G. Emma, “Branch History Table Prediction of Moving Target Branches Due to Subroutine Returns,” *Proc. 18th Int’l Symp. Computer Architecture*, pp. 34-42, Toronto, May 1991.

[11] C.F. Webb, “Subroutine Call / Return Stack,” *IBM Technical Disclosure Bulletin*, vol. 30, no. 11, Apr. 1988.

[12] D.R. Kaeli and P.G. Emma, “Case Block Table for Folding Multiway Branches,” U.S. Patent No. 5,333,283, U.S. Patent Office, July 26, 1994.

[13] D.R. Kaeli, “Branch History Table Optimization,” PhD dissertation, Dept. of Electrical Eng., Rutgers Univ., New Brunswick, N.J., Oct. 1992.

[14] P.G. Emma, J.H. Pomerene, G.S. Rao, R.N. Rechtschaffen, H.E. Sachar, and F.J. Sparacio, “Branch Prediction Mechanism in Which a Branch History Table is Updated using an Operand Sensitive Branch Table,” U.S. Patent 4,763,245, Aug. 9, 1988.

[15] “SPEC Newsletter,” Systems Performance Cooperative, Spring 1990.

[16] A.V. Sampogna, “Architectural Implications of C and C++ Programming Models,” MS thesis, Dept. of Electrical and Computer Eng., Northeastern Univ., Boston, Aug. 1995.

[17] T.-Y. Yeh and Y.N. Patt, “Two-Level Adaptive Branch Prediction,” *Proc. 24th Ann. ACM/IEEE Int’l Symp. and Workshop Microarchitecture*, pp. 51-61, Nov. 1991.