Improving the Accuracy of Dynamic Branch Prediction Using Branch Correlation

Shien–Tai Pan
Advanced Workstation Division
IBM Corporation, 9440
11400 Burnet Road
Austin, Texas 78758–3493

Kimming So
Advanced Workstation Division
IBM Corporation, 2808
11400 Burnet Road
Austin, Texas 78758–3493
sok@watson.ibm.com

Joseph T. Rahmeh
Electrical and Computer
Engineering Department
University of Texas at Austin
Austin, Texas 78712
rahmeh@cerc.austin.edu

Abstract

Long branch delay is a well–known problem in today's high performance superscalar and superpipeline processor designs. A common technique used to alleviate this problem is to predict the direction of branches during the instruction fetch. Counter–based branch prediction, in particular, has been reported as an effective scheme for predicting the direction of branches. However, its accuracy is generally limited by branches whose future behavior is also dependent upon the history of other branches. To enhance branch prediction accuracy with a minimum increase in hardware cost, we propose a correlation–based scheme and show how the prediction accuracy can be improved by incorporating information, not only from the history of a specific branch, but also from the history of other branches. Specifically, we use the information provided by a proper subhistory of a branch to predict the outcome of that branch. The proper subhistory is selected based on the outcomes of the most recently executed M branches. The new scheme is evaluated using traces collected from running the SPEC benchmark suite on an IBM RISC System/6000 workstation. The results show that, as compared with the 2–bit counter–based prediction scheme, the correlation–based branch prediction achieves up to 11% additional accuracy at the extra hardware cost of one shift register. The results also show that the accuracy of the new scheme surpasses that of the counter–based branch prediction at saturation.

1. Introduction

Recent advances in RISC architectures and VLSI technologies allow computer designers to exploit more instruction–level parallelism with deeper pipelines and more concurrent functional units [1, 2]. As sophisticated processors are built to exploit the available instruction–level parallelism, more attention needs to be paid to the disruption of pipeline flow as a result of branch instruction execution [8]. Pipeline disruption reduces the effective instruction throughput by introducing extra delays in the pipeline. Since branches constitute a large portion of all the executed instructions, the efficiency of handling branches is important. Our primary interest is in reducing the branch penalty incurred in executing conditional branches. All branches mentioned below, unless otherwise stated, are conditional branches.

Almost all the branch cost reduction techniques reported in the literature require the use of some mechanism for predicting the outcome of branches. Other than the profiling technique [3, 5], all prediction schemes require hardware assistance. Hardware–assisted branch predictions typically fall into two categories: *static* and *dynamic*. Overview of these schemes can be found in [4, 7]. Generally, dynamic prediction gives better results than static prediction, but at the cost of increased hardware complexity. A less–complex yet reasonably effective scheme is the ***N–bit counter scheme*** [3, 4, 7]. In this scheme, the prediction of the outcome of a branch is based on the output of a finite–state machine whose state is recorded in an N–bit up/down counter. The counter is incremented or decremented according to whether the branch is taken or not. We refer to this scheme as the ***counter–based*** branch prediction. Later we will show its operation in more detail.

A common limitation with most of the dynamic branch prediction schemes is that the prediction is based on "*self–history*". Specifically, the prediction is based exclusively on the past history of the branch under consideration, completely ignoring the information provided by the executions of other branches. Self–history prediction schemes generally work well for scientific/engineering applications where program execution is dominated by inner–loops. However, in many integer workloads, control–flows are complex and very often the outcome of a branch is affected by the outcomes of recently executed branches. In other words, the branches are **correlated**. Because of the correlation, the history of a branch, considered by itself, is very chaotic and that reduces the accuracy of self–history

ASPLOS V - 10/92/MA,USA
© 1992 ACM 0-89791-535-6/92/0010/0076...$1.50

prediction schemes. A prior study shows that *branch correlation* does take place in programs and that its source can be traced back to common high-level language constructs [6]. The Appendix summarizes some of our observations of the source code-level branch correlation that appear in the SPEC integer benchmarks.

Contrary to the self-history based approach, the *"two-level adaptive training branch prediction"* reported recently uses the global branch history pattern associated with each branch address for predicting the outcome of the branch [10]. The same global history pattern results in the same prediction, regardless of which branch address the history pattern is associated with. Although this approach is reported to produce a fairly high prediction accuracy [10], its hardware implementation seems quite complicated.

To our knowledge, very little work has been done in addressing the issue of branch correlation in branch prediction. In this paper, we study the effect of branch correlation in branch prediction and propose a **correlation-based** prediction scheme which also produces high prediction accuracy. The proposed branch prediction scheme is simple to implement and its implementation is very similar to that of the counter-based branch prediction.

The new scheme is evaluated using traces collected from running the SPEC benchmark suite [9] on an IBM RISC System/6000 workstation. The results show that, as compared with the 2-bit counter-based prediction scheme, the correlation-based branch prediction achieves up to 11% additional accuracy at the extra hardware cost of one shift register. The results also show that the accuracy of the new scheme surpasses that of the counter-based branch prediction at saturation.

The remainder of this paper is organized as follows: In section 2, the correlation-based branch prediction scheme is introduced with an example. A brief description of the counter-based branch prediction is also given. In section 3, simulation results evaluating the new scheme are presented. In section 4, we give the main conclusions.

2. Dynamic Branch Prediction

In this section, we will describe the N-bit counter scheme and introduce a new prediction scheme based on branch correlation. An example will be given to explain the difference between these two schemes. Finally, the implementation of the new scheme will be discussed.

2.1 Counter-Based Branch Prediction

The basic idea for the counter-based branch prediction is to use an N-bit up/down counter [3, 4, 7] for prediction. In the ideal case, an N-bit counter (with some initial value) is assigned to each *static branch* (branches with distinct addresses). When a branch is about to be executed, the counter value C, associated with that branch, is used for prediction. If C is greater than or equal to a predetermined threshold value L, the branch is predicted taken, otherwise it is predicted not taken. A typical value for L is 2^{N-1}. The counter value C is updated whenever that branch is resolved. If the branch is taken, C is incremented by one, otherwise it is decremented by one. If C is 2^N-1, it remains at that value as long as the branch is taken. If C is 0, it remains at zero as long as the branch is not taken.

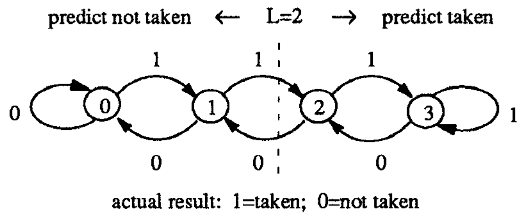

predict not taken ← L=2 → predict taken
1
1
1
0
0
1
2
3
1
0
0
0
actual result: 1=taken; 0=not taken

Fig. 1 FSM for the 2-bit Counter Scheme

The operation of the N-bit counter scheme corresponds to a finite-state machine (FSM) with 2^N states. Fig. 1 shows the FSM with N=2 and L=2. Smith [7] reported that a counter of 2 bits is usually as good or better than other strategies and a larger counter size does not necessarily give better results.

2.2 Correlation-Based Branch Prediction

Most studies of dynamic branch prediction focus on the history of the branch under consideration [4, 7]. With hardware-assisted branch prediction, only the most recent history of a branch is used to predict the outcome of that branch. These branch prediction schemes work well for scientific/engineering workloads where program execution is dominated by inner-loops. However, they do not work as well for integer workloads where the outcome of a branch is affected by the outcomes of recently executed branches. When one branch depends on another, in the sense that its outcome depends on the outcome of the other branch, we say that the branches are **correlated**.

As an illustration of branch correlation, consider the code fragment shown in Fig. 2:

```
if(aa==2)                    /* b1 */
        aa = 0;
if(bb==2)                    /* b2 */
        bb = 0;
if(aa != bb) {               /* b3 */
        .....
}
```

Fig. 2 A Code Fragment from SPEC Benchmark *eqntott*

This code fragment (other than the comments) appears in a frequently executed block of the SPEC integer benchmark *eqntott*. There are three *if*-statements in this code fragment. Assume that the *if*-statements are converted by a compiler to three branch instructions b_1, b_2, and b_3, and the action determined by each *if*-statement is the branch "fall-through path", meaning that the branch "taken" path is the path for which the condition is not true. Since the outcome of b_3 depends on the values of *aa* and *bb*, it is obvious that b_3 is **correlated** with b_1 and b_2.

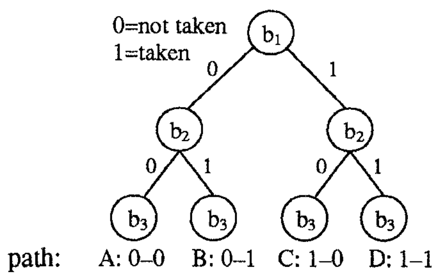


Fig. 3 Branch Tree for the Code Fragment Given in Fig. 2

Although the presence of branch correlation may cause the behavior of a branch to appear more random, it may shed some light on the condition upon which the branch decision is based. Consider again the same example given in Fig. 2. After the executions of b_1 and b_2, the condition that b_3 is dependent upon is already partially known. Fig. 3 shows the part of the branch tree before the execution of b_3 given that b_1 and b_2 have been executed.

There are four possible paths reaching b_3 through the executions of b_1 and b_2. For example, if b_1 is taken and b_2 is not taken, then b_3 is reached via the 1–0 path (path C in Fig. 3). Fig. 4 shows the information available at b_3, given that b_1 and b_2 have been executed. It is clear that if b_3 is reached via the 0–0 path, the outcome of b_3 can be determined prior to its execution. But this situation cannot be exploited by the conventional self–history based prediction schemes. This example suggests that the outcome of a branch can be more readily determined if the path leading to it is known. By splitting the branch history of b_3 into four **subhistories** according to the paths leading to b_3, one may reduce the randomness of the apparent behavior of b_3 and thus make a better prediction.

path leading to b_3:	A: 0–0	B: 0–1	C: 1–0	D: 1–1
	$\begin{cases} aa=0 \\ bb=0 \end{cases}$	$\begin{cases} aa=0 \\ bb\neq 2 \end{cases}$	$\begin{cases} aa\neq 2 \\ bb=0 \end{cases}$	$\begin{cases} aa\neq 2 \\ bb\neq 2 \end{cases}$

Fig. 4 Information About *aa, bb* Available at b_3 After b_1 and b_2 Have Been Executed

Let's further examine the example with data that are arbitrarily chosen only to reflect the branch correlation. Suppose that we run the code fragment given in Fig. 2 on a machine which implements the 2–bit counter scheme shown in Fig. 1 with initial state set to 0. Table 1 shows the predicted outcomes of b_3 and the state transitions. The first two columns show the initial values of *aa* and *bb* before the execution of b_1. Columns *aa'* and *bb'* in the table show the new values of *aa* and *bb* after b_1 and b_2 are executed. Column "*path*" indicates the path from which b_3 is reached. Column "*curr state*" shows the current state of the FSM. Column "*pred*" shows the predicted outcome of b_3. The actual outcome is given in column "*act*". Column "*c/w*" indicates the correct (c) or wrong (w) prediction. The state is updated according to the current state and the actual outcome. The updated state is shown in the column under "*next state*".

Table 1 State Transitions and Branch Predictions for b_3 Using 2–bit Counter–Based Prediction Scheme

aa	*bb*	*aa'*	*bb'*	*path*	*curr state*	*pred*	*act*	*c/w*	*next state*
0	2	0	0	C	0	N	T	w	1
2	2	0	0	A	1	N	T	w	2
2	1	0	1	B	2	T	N	w	1
2	0	0	0	B	1	N	T	w	2
2	2	0	0	A	2	T	T	c	3
1	0	1	0	D	3	T	N	w	2
1	0	1	0	D	2	T	N	w	1
2	0	0	0	B	1	N	T	w	2
0	1	0	1	D	2	T	N	w	1
1	1	1	1	D	1	N	T	w	2
1	2	1	0	C	2	T	N	w	1
1	2	1	0	C	1	N	N	c	0
2	2	0	0	A	0	N	T	w	1
2	0	0	0	B	1	N	T	w	2
0	1	0	1	D	2	T	N	w	1
2	2	0	0	A	1	N	T	w	2
0	2	0	0	C	2	T	T	c	3
0	1	0	1	D	3	T	N	w	2
1	0	1	0	D	2	T	N	w	1
2	2	0	0	A	1	N	T	w	2

N=not taken, T=taken, c=correct pred., w=wrong pred.

A careful inspection of the table reveals that the apparently random branch history of b_3 (column "*act*") is actually formed by interweaving four less random branch subhistories, each of which is associated with a branch path leading to b_3 (compare columns "*path*" and "*act*"). After splitting the branch history of b_3 according to the four branch paths shown in Fig. 5, one can obtain the four branch subhistories of b_3.

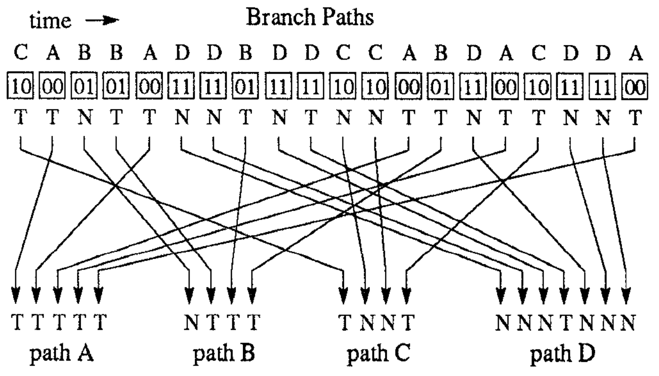


Fig. 5 Subhistories Obtained by Splitting the History of b_3 According to the Branch Paths

It is evident from Fig. 5 that the outcomes of b_3 are less random within each subhistory. Hence better predictions are expected if we independently implement a 2–bit counter for each subhistory. In fact, only 3 out of the 20 executions of b_3 are correctly predicted if only one 2–bit counter (with initial state equal to 0) is used. However, if four 2–bit counters are used (all initialized to 0), with one for each subhistory, 10 additional correct predictions can be obtained. Note that the state transition and the state update of the FSM associated with each counter are local to each branch path. This is shown in Fig.

6. Notice that we are not suggesting to use four counters for "each" branch. We are merely showing that taking the path leading to a branch into consideration leads to a better prediction. Later we will show an implementation scheme that exploits the "correlation" between branches without increasing the overall number of counters used to track the history of branches.

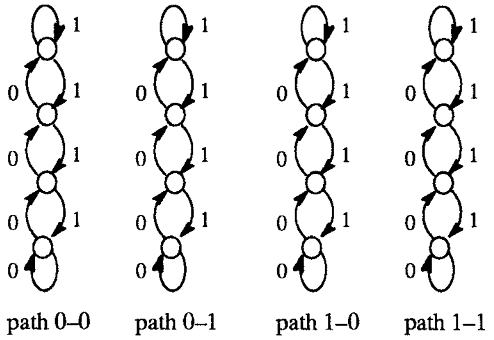


Fig. 6 FSMs using Four 2–bit Counters

Fig. 6 suggests that in order to select the proper 2–bit counter assigned to each subhistory for prediction, one needs to memorize the branch path leading to b_3. This can be achieved by using a 2–bit shift register which records the outcomes of the two most recently executed branches. The shift register is then used to select the appropriate counter. The use of a shift register for tracking and selectively relating the correlated information to proper branch subhistory is the main idea of the proposed **correlation–based** branch prediction. Basically, the proposed scheme uses the branch path information to split the history of a branch into several subhistories and selectively use the proper subhistory information for predicting the outcome of the branch.

Generally, an **M–step** correlation–based branch prediction uses the outcomes of the last M branches (including unconditional branches) seen by the machine to split the history of a branch into 2^M subhistories. The prediction is then done independently within each subhistory using any (or the best) history–based branch prediction algorithm. A good candidate for prediction within each subhistory is the N–bit counter–based branch prediction mentioned earlier. In this case, an **M–bit shift register** is required to store the outcomes of the last M branch executions (0 for not taken, 1 for taken). This shift register is able to identify a total of 2^M subhistories of a branch. Within each subhistory, the prediction is done using an N–bit counter associated with it. There are a total of 2^M FSM's associated with each branch. Everytime the outcome of a branch is to be predicted, the M–bit shift register is used to select the proper FSM, resulting in a set of N prediction bits. Once the FSM is selected, the prediction and the state update are done according to the N–bit counter–based prediction algorithm.

In the following, we will refer to this scheme as the **(M,N) correlation–based** branch prediction scheme or simply the **(M,N) correlation scheme**, meaning that an M–bit shift register is used to select an N–bit counter for prediction. The **number of correlation steps** is defined as the number of bits in the shift register. When the prediction scheme used within each subhistory is understandable without any ambiguity, we will simply refer to it as an **M–step correlation scheme.**

2.3 Implementation

When the N–bit counter scheme or the (M,N) correlation scheme is implemented by itself, a table is required to store the prediction information. We refer to this table as the *"branch prediction table"* or briefly, BPT. Fig. 7 (a) shows the logical organization of a 1K–entry BPT for the 2–bit counter scheme, with each entry containing 2 prediction bits. Fig. 7 (b) shows the logical organization of a 1K–entry BPT for the (2,2) correlation scheme, with each entry containing $2 \times 2^2 = 8$ prediction bits.

Notice the difference in physical size of the two tables, even though the number of logical entries is identical. In general, if a 2^I–entry table is used for (M,N) correlation scheme, a total of $N \times 2^{I+M}$ bits is required for the table, with each entry containing 2^M sets of N prediction bits. The table is generally accessed using the low–order I bits of the branch address. However, depending on the implementation, the table may be accessed using the address of the instruction immediately prior to the branch under consideration [11]. Once the entry is determined, the M–bit shift register which stores the outcomes of the last M branches is used to select the proper set of the N bits from the entry. These N bits are used for predicting the outcomes of all branches whose addresses are mapped into the same entry.

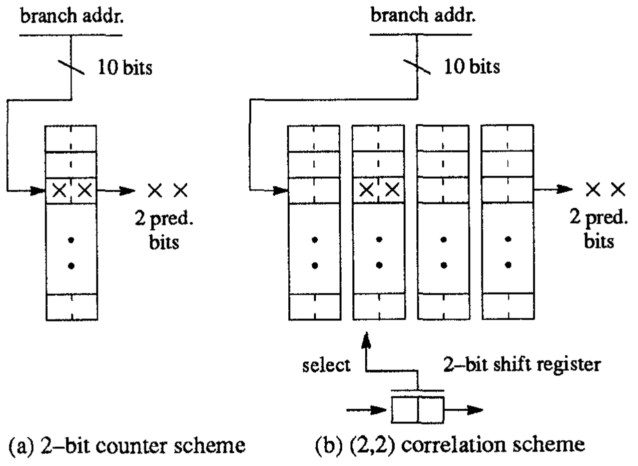


Fig. 7 Logical Organization of a 1K–Entry Table

A design tradeoff in implementing the dynamic branch prediction usually involves in choosing the physical size of the BPT for a desired prediction accuracy. It is interesting to note from Fig. 7 that if the BPT size is to be changed, two "logical directions" can be considered. The table size can be increased/decreased either along the vertical direction as shown in Fig. 8 (a) or along the horizontal direction as shown in Fig. 8 (b). Fig. 8 (a) is typical for implementing the counter–based scheme whereas Fig. 8 (b) is typical for correlation schemes. We will refer to the directions shown in Fig. 8 (a) and (b) as the **entry–dimension** and the **correlation–dimension**, respectively. Of course, any combination of the two is possible.

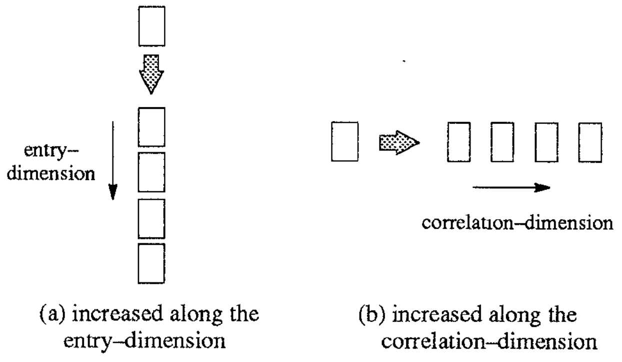

entry–dimension
correlation–dimension
(a) increased along the entry–dimension
(b) increased along the correlation–dimension

Fig. 8 Increasing the Size of a BPT

While the logical organization and the behavior of the tables for the counter and the correlation schemes are different, the physical implementations are quite similar. Fig. 9 shows the implementation using a 1KB–BPT. When this table is used for the 2–bit counter scheme, 12 bits are required for a table lookup (Fig. 9 (a)). As mentioned earlier, these 12 bits are usually obtained from the branch address. However, if the same table is used for correlation schemes, some of the bits for table lookup are obtained from the shift–register. For example, if the (8,2) correlation scheme is implemented as shown in Fig. 8 (b), the bits for table lookup consist of 8 bits from the shift register and 4 bits from the branch address. It is important to note that as a correlation scheme is implemented instead of the original 2–bit counter scheme using the same size of table, the only extra hardware cost incurred by the correlation scheme is the shift register (Fig. 9 (b)).

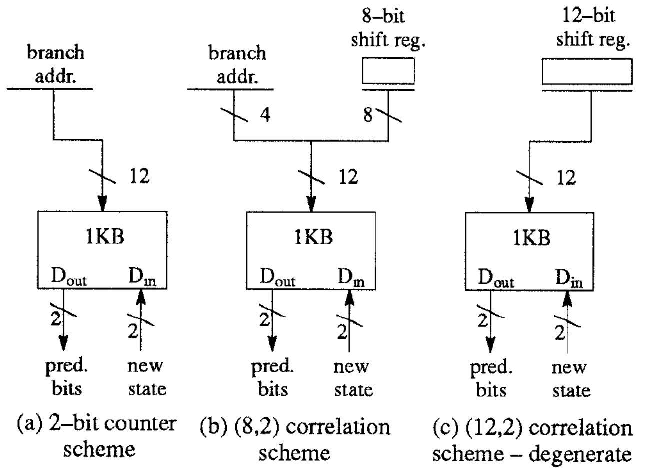


Fig. 9 Physical Implementation Using a 1KB–BPT

Fig. 9 (b) also shows an interesting case: as the table size is fixed, the larger the shift register used, the fewer branch address bits are required. In other words, as the table size is fixed, increasing the size of the shift register is equivalent to "squashing" the BPT along the entry–dimension. An interesting extreme case occurs when the table degenerates to a single–entry table. In this case, the bits for table lookup are obtained entirely from the shift register. Fig. 9 (c) shows the degenerate case for a 1KB–BPT. This case is equivalent to implementing the (12,2) correlation scheme using a single–entry table shown in Fig. 10. Similarly, Fig. 9 (a) can be thought as the other extreme case when the table in Fig. 9 (b) is "squashed" along the correlation–dimension. The advantage of considering the degenerate case is that its table lookup depends only on the shift register, completely independent of the branch address. Because of this unique characteristic, a resolved branch always predicts the outcome of the next branch. The degenerate case of the correlation scheme is interesting, not only because of its simple implementation, but also because the predicted outcome of a branch can be known way before the execution of that branch.

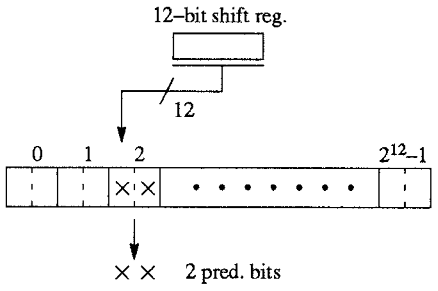


Fig. 10 Degenerate Case for a 1KB–BPT

3. Trace–Driven Simulations & Results

Trace–driven simulations are used to examine the (M,2) correlation schemes for BPT's with entries ranging from 1 to 32K. Due to the limitation of the program size and simulation time, only $M \leq 10$ are evaluated for non–degenerate cases and $M \leq 15$ for degenerate cases. Note that the scheme (0,2) corresponds to the original 2–bit counter scheme. The programs used for the experiment are from the SPEC benchmark suite. The traces are collected using a trace program and commercially available C and FORTRAN compilers for the IBM RISC System/6000 system. Table 2 summarizes the trace lengths and branch statistics for the benchmarks used in this study. The **accuracy**, defined as the *percentage of correct predictions*, will be used as the metric for measuring the efficiency of branch prediction.

For SPEC floating–point benchmarks *nasa7*, *matrix300*, and *tomcatv*, no difference is found between correlation–based schemes and the 2–bit counter scheme. All predict with more than 99% accuracy. These results are not surprising for loop–intensive scientific/ engineering applications where programming structures are dominated by simple loops. Because of this, only the results of the other 7 SPEC benchmarks, namely, *doduc, spice, fpppp, gcc, espresso, eqntott,* and *li*, are presented. For convenience, we will use the shorthand "7 SPEC benchmarks" or "7 benchmarks" to mean these 7 benchmarks, "floating–point benchmarks" to mean the benchmarks *doduc, spice,* and *fpppp,* and "integer benchmarks" to mean the benchmarks *gcc, espresso, eqntott,* and *li.*

Table 2 Branch Statistics for SPEC Benchmarks

	Inst.	b_u	b_c	p	q	s
spice	50M	.093	.125	.538	.196	41.3
doduc	50M	.020	.094	.630	.551	137.2
nasa7	50M	~ 0	.166	.994	.993	0.6
matrix300	50M	.001	.198	.993	.993	1.7
fpppp	50M	.005	.016	.575	.450	197.2
tomcatv	50M	~ 0	.059	.993	.993	72.6
gcc	50M	.041	.189	.635	.556	800.3
espresso	50M	.071	.193	.538	.369	46.7
li	50M	.062	.165	.601	.450	39.7
eqntott	50M	.021	.305	.445	.406	2.8

b_u:	frequency of unconditional branches
b_c:	frequency of conditional branches
p:	probability that a branch is taken
q:	probability that a conditional branch is taken
s:	static conditional branches per 1million executed conditional branches

3.1 Accuracy for Fixed Table Size

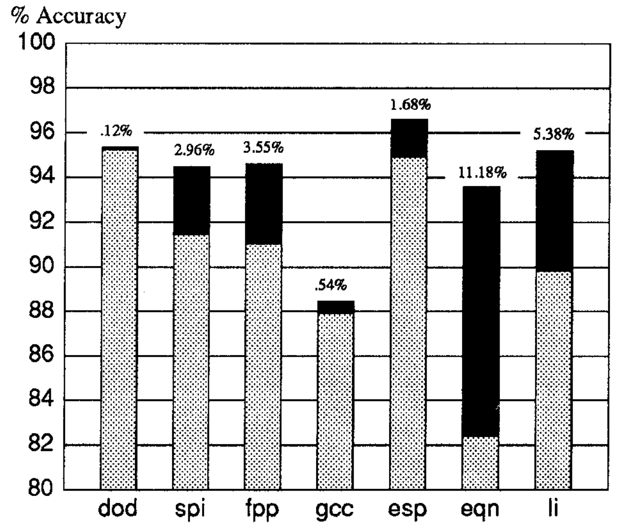
We first compare the correlation–based scheme with the 2–bit counter scheme using the same 1KB–BPT. Notice that the number of table entries for the two schemes are different (see Fig. 7). A 1KB–table has 4K entries when the 2–bit counter scheme is implemented, whereas the same table has only 16 entries when the (8, 2) correlation schemes is implemented.

Fig. 11 shows the results for a 1KB–BPT. The figure compares the accuracy obtained by implementing the 2–bit counter scheme and the additional accuracy gained by implementing the (8, 2) correlation scheme. Since the 2–bit counter scheme has already provided very high accuracies for *doduc* and *espresso* (about 95%), there is very little chance for correlation schemes to gain more accuracy. The benchmark *gcc* shows very little improvement in accuracy. This is because that a 1KB–table is not large enough to contain most of the frequently executed branches in *gcc.*

The remaining benchmarks show considerable improvements in accuracy. The two biggest gains in accuracy are obtained by *eqntott* and *li.* Since branches in *eqntott* are highly correlated, the 2-bit counter scheme cannot provide high accuracy (only about 83%). More than 11% of additional accuracy can be attained by the correlation scheme.

The second highest improvement in accuracy is achieved by *li* (more than 5%). It is known that *li* is a "pointer–chasing" oriented program where a compiler may generate *load, compare,* and *branch* instructions in sequence over and over again. The branch correlation exists wherever the data loaded for determining the branch direction is affected by the directions of prior branches. As reported in [2], a *compare–branch* pair of instructions in the IBM RISC System/6000 machine causes a 3–cycle bubble in the pipeline. The correlation–based scheme proposed here is particularly useful to reduce such delay.

Although we have only shown the results for the 8–step correlation scheme, it is observed from the simulation that, as the number of table entries is fixed, the accuracy increases as the number of correlation steps increases. This observation is true for all the 7 benchmarks.



▨ accuracy for the 2–bit counter scheme

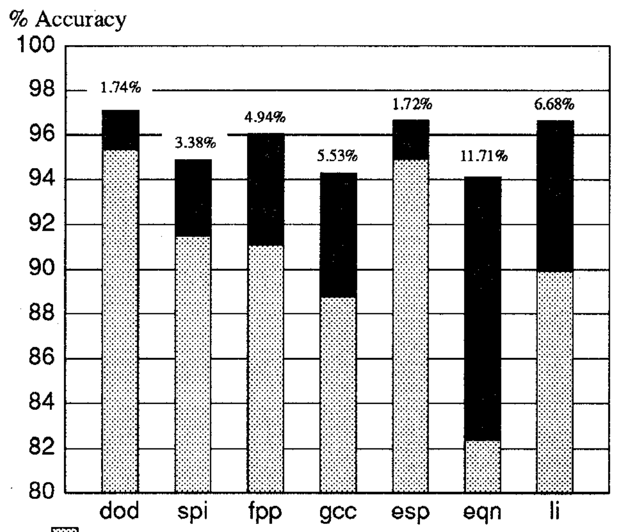
■ additional accuracy gained by implementing the (8,2) correlation scheme with the same table

Fig. 11 Accuracies for an 1KB–BPT: (0,2) v.s. (8,2)

3.2 Accuracy at the Limiting Case

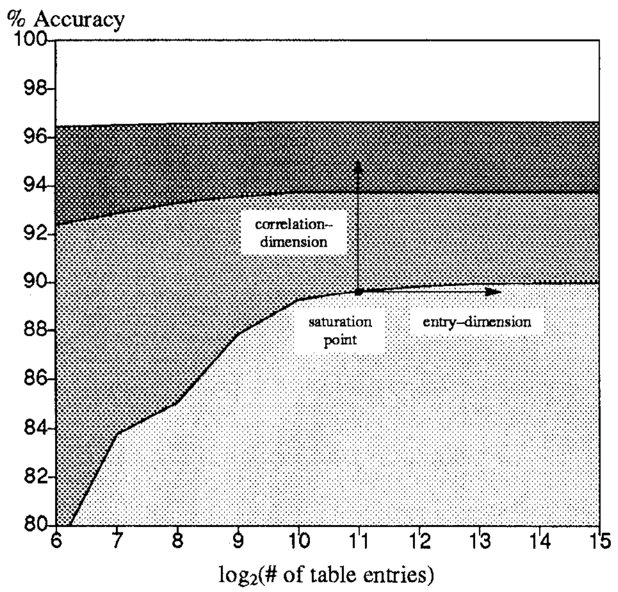
It is observed that the accuracy provided by the 2–bit counter scheme asymptotically approaches certain limit as the BPT size increases. Fig. 12 shows the limit at which the 2–bit counter scheme saturates. When the table is large enough to contain most of the frequently executed branches, the prediction capability of the 2–bit counter scheme reaches its inherent limits. As we mentioned earlier, one of the limitations of the 2–bit counter scheme is that it is self–history based. Since the correlation scheme provides better prediction by incorporating the information from other branches, it can surpass the limit at which the 2–bit counter scheme saturates.

As an illustration, consider the accuracy curves for *li* shown in Fig. 13. It is clear that the accuracy provided by the 2–bit counter scheme saturates at a table of 2K entries. Increasing the table size along the entry–dimension as shown in the figure makes very little improvement in accuracy. However, if the BPT size is increased along the correlation dimension (see Fig. 8 (b)), more accuracy can be gained. Fig. 12 shows the additional accuracy achievable by the correlation scheme for the 7 benchmarks.



▨ limiting case accuracy of the 2–bit counter scheme
■ additional accuracy achievable by the correlation scheme

Fig. 12 Limiting Case Accuracy



▨ 2–bit counter scheme ▨ (10,2) correlation scheme
▨ (5,2) correlation scheme

Fig. 13 Prediction Accuracy for *li*

3.3 Accuracy at the Degenerate Case

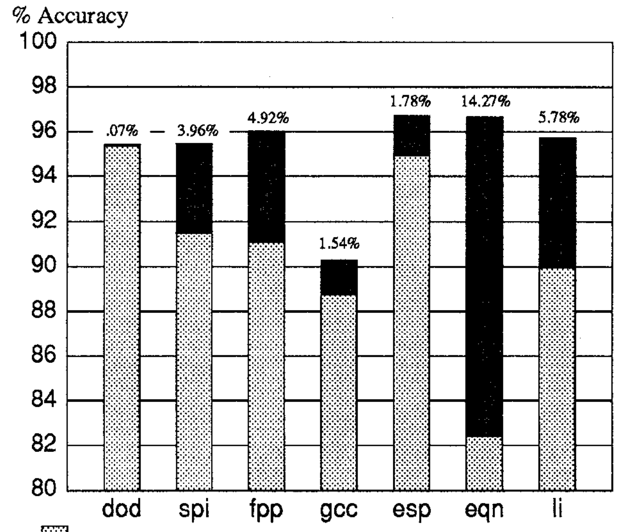
The degenerate correlation scheme provides an interesting case for a practical implementation, since its table lookup doesn't depend on the branch address. Because of this unique characteristic, the table lookup for the next branch can be done as soon as the current branch is resolved. This is attractive to timing–critical implementations of the branch prediction.

The only disadvantage with the degenerate case is that the table must be very large in order to outperform the 2–bit counter scheme. This is due to the fact that enormous amount of address conflicts are introduced with an one–entry table (Fig. 10). However, the effect of address conflict is attenuated when the table size is large. It is observed from the simulation that a larger correlation step is required before the degenerate case has a noticeable improvement over the 2–bit counter scheme. Table 3 summarizes the observation.

It is also observed that when the table size is large, the degenerate case sometimes performs better than the non–degenerate case. Fig. 14 shows the results of implementing the degenerate (15,2) scheme using an 8KB–table.

Table 3 # of Correlation Steps Required Before Degenerate Case has Noticeable Improvement Over the 2–Bit Counter Scheme

doduc	*spice*	*fpppp*	*gcc*	*espresso*	*eqntott*	*li*
15	6	10	14	8	5	11



▨ accuracy for the 2–bit counter scheme
■ additional accuracy gained by implementing the degenerate (15,2) correlation scheme

Fig. 14 Accuracy for an 8KB–BPT: (0,2) v.s. Degenerate (15,2)

4. Conclusions

In this paper, we have proposed a novel dynamic branch prediction scheme which uses the proper subhistory information of a branch to predict the outcome of that branch. The key idea is to relate the subhistory which is being selected to the most recently executed branches via a shift register. The new scheme is evaluated using traces collected from running the SPEC benchmark suite on an IBM RISC System/6000 machine. It is shown that the proposed new scheme gives considerably higher accuracy than that of the 2–bit counter prediction scheme at the extra hardware cost of one shift register. We have observed from the simulation that for the same BPT of size 1KB or above, the (M,2) correlation scheme generally provides the best improvement in accuracy over the 2–bit counter scheme for $5 \leq M \leq 8$. We want to emphasize that as more instruction–

level parallelism is exploited by today's superscalar and superpipelined processors, few percent increase in branch prediction accuracy is significant in improving the overall processor performance.

We have demonstrated that the new scheme is simple and easy to implement. It provides a new dimension as a design alternative for increasing the BPT size, i.e., the *correlation–dimension*. We have also shown that the accuracy of the correlation scheme surpasses that of the 2–bit counter scheme at saturation.

Acknowledgements

We would like to thank Ju–ho Tang of IBM T. J. Watson Research Center for providing the tracing tool, Chin–Cheng Kau, Ed Silha, Wade Shaw of IBM/Austin, and Kate Stewart of IBM/Toronto for reviewing our earlier drafts, and the management of IBM Advanced Workstation Division/Systems Architecture & Performance for their support of this research.

Reference

[1] A. Bashteen, I. Lui, J. Mullan, "A Superpipeline Approach to the MIPS Architecture," *Proceedings of the IEEE Compcon'91, February 1991, pp. 8–12.*

[2] G. F. Grohoski, "Machine Organization of the IBM RISC System/6000 Processor," *IBM J. of Research and Development, Vol. 34, No. 1, January 1990, pp. 37–58.*

[3] W. M. Hwu, T. M. Conte, P. P. Chang, "Comparing Software and Hardware Schemes for Reducing the Cost of Branches," *Proceedings of the 16th Annual International Symposium on Computer Architecture, May, 1989, pp. 224–233.*

[4] J. K. F. Lee, A. J. Smith, "Branch Prediction Strategies and Branch Target Buffer Design," *IEEE Computer, 17, 1, January, 1984, pp. 6–22.*

[5] S. McFarling, J. Hennessy, "Reducing the Cost of Branches," *Proceedings of the 13th Annual International Symposium on Computer Architecture, June, 1986, pp. 396–403.*

[6] S. T. Pan, K. So, J. T. Rahmeh, "Correlation–Based Branch Prediction," *Technical Report, UT–CERC–TR–JTR91–01, University of Texas at Austin, August, 1991.*

[7] J. E. Smith, "A Study of Branch Prediction Strategies," *Proceedings of the 8th Annual International Symposium on Computer Architecture, June, 1981, pp. 135–147.*

[8] D. W. Wall, "Limits of Instruction–Level Parallelism," *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems, April, 1991, pp176–188.*

[9] Workstation Performance, The SPEC Benchmark Suit Release 1.0, *System Performance Evaluation Cooperative, June, 1990.*

[10] T. Y. Yeh, Y. N. Patt, "Two–Level Adaptive Training Branch Prediction," *Proceedings of the 24th Annual International Symposium on Microarchitecture, November, 1991, pp. 51–61.*

[11] T. Yoshida, T. Shimizu, S. Mizugaki, J. Hinata, "The Gmicro/100 32–Bit Microprocessor," *IEEE Micro, August, 1991, pp. 20–23 & 62–72.*

Appendix

Examples of Source Code–Level Branch Correlation from the SPEC Integer Benchmarks:

benchmark: eqntott   file name: pterm_ops.c

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

benchmark: eqntott   file name: pterm_ops.c

```
while (low <= high) {
    i = (high + low) / 2;
    if (H (i) < hsh)
        low = i + 1;
    else if (i > 0 && H (i–1) >= hsh)
        high = i – 1;
    else if (H (i) == hsh)
        break;
    else   return (NIL_PTERM);
}
```

benchmark: eqntott   file name: ucbqsort.c

```
j = (j == jj ? i : jj);
if ((*qcmp)(j, tmp) < 0)
    j = tmp;
```

benchmark: li   file name: xllist.c

```
while (*adstr && consp(list))
 list = (*adstr++ == 'a' ? car(list) : cdr(list));
```

benchmark: li   file name: xlread.c

```
while ((ch = xlpeek(fptr)) != EOF) {
  if (islower(ch)) ch = toupper(ch);
  if (!isdigit(ch) && !(ch >= 'A' && ch <= 'F'))
    break;
    ........
}
```

benchmark: li file name: xlmath.c

```
if (imode)
  switch (fcn) {
  case '<':    icmp = (icmp < 0); break;
  case 'L':    icmp = (icmp <= 0); break;
  case '=':    icmp = (icmp == 0); break;
  case '#':    icmp = (icmp != 0); break;
  case 'G':    icmp = (icmp >= 0); break;
  case '>':    icmp = (icmp > 0); break;
  }
else
  switch (fcn) {
  case '<':    icmp = (fcmp < 0.0); break;
  case 'L':    icmp = (fcmp <= 0.0); break;
  case '=':    icmp = (fcmp == 0.0); break;
  case '#':    icmp = (fcmp != 0.0); break;
  case 'G':    icmp = (fcmp >= 0.0); break;
  case '>':    icmp = (fcmp > 0.0); break;
  }
return (icmp ? true : NIL);
```

benchmark: li file name: xlcont.c

```
rbreak = FALSE;
while (xleval(test) == NIL) {
  if (tagblock(arg,&rval)) {
    rbreak = TRUE;
    break;
  }
}
if (!rbreak)
  ........
```

benchmark: espresso file name: compl.c

```
for(pl = *L1, pr = *R1; (pl != NULL) &&
    (pr != NULL); )
switch (d1_order(L1, R1)) {
   case 1:
     pr = *(++R1); break;
   case -1:
     pl = *(++L1); break;
   case 0:
     RESET(pr, ACTIVE);
     INLINEset_or(pl, pl, pr);
     pr = *(++R1);
}
```

benchmark: gcc file name: reload.c

```
if (in != 0)
  class = PREFERRED_RELOAD_CLASS (in, class);
if (class == NO_REGS)
  ........
```

benchmark: gcc file name: cse.c

```
if (elt != 0 && elt->related_value != 0)
  relt = elt;
else if (elt == 0 && GET_CODE (x) == CONST)
  {
   rtx subexp = get_related_value (x);
   if (subexp != 0)
    relt = lookup (subexp,
        safe_hash (subexp, GET_MODE (subexp)) %
        NBUCKETS,
        GET_MODE (subexp));
  }
if (relt == 0)
  return 0;
```

benchmark: gcc file name: flow.c

```
for (j = XVECLEN (x, i) - 1; j >= 0; j--)
  {
   ........
 if (value == 0)
     value = tem;
     ........
  }
```

benchmark: gcc file name: flow.c

```
while (INSN_DELETED_P (first))
  first = NEXT_INSN (first);
while (prev != first)
  {
   prev = PREV_INSN (prev);
   PUT_CODE (prev, NOTE);
   NOTE_LINE_NUMBER (prev) = NOTE_INSN_DELETED;
   NOTE_SOURCE_FILE (prev) = 0;
  }
```

benchmark: gcc file name: cse.c

```
if (tem != 0)
     y0 = tem;
if (y0 == 0)
     return 0;
```

benchmark: gcc file name: cse.c

```
switch (i)
  {
  case 0:
   const_arg0 = const_arg;
   break;
  case 1:
   const_arg1 = const_arg;
   break;
  case 2:
   const_arg2 = const_arg;
   break;
  }
........
switch (code)
  {
  ........
  case EQ:
   if (const_arg0 && const_arg0 == XEXP (x, 0)
  && (! (const_arg1 && const_arg1 == XEXP (x, 1))
    || (GET_CODE (const_arg0) == CONST_INT
       && GET_CODE (const_arg1) != CONST_INT)))
  ........
```