RC 20100 (88908) 6/14/95
Computer Science/Mathematics **16 pages**

IBM Research Report

Dynamic Path-Based Branch Correlation

Ravi Nair

IBM Research Division
T. J. Watson Research Center
Yorktown Heights, NY 10598

Research Division
Almaden . T.J. Watson . Tokyo . Zurich

Copies may be requested from:

IBM Thomas J. Watson Research Center
Publications Office, 16-220
Post Office Box 218
Yorktown Heights, NY 10598

Dynamic Path-Based Branch Correlation

Ravi Nair
IBM T. J. Watson Research Center
P. O. Box 704
Yorktown Heights, NY 10598

nair@watson.ibm.com

Abstract: **Misprediction of conditional branches is a major cause for reduced performance in processor implementations with large numbers of functional units. We present a hardware scheme which records the path leading to a conditional branch in order to predict the outcome of the branch instruction more accurately. The proposed scheme is analyzed using instruction traces from integer benchmark programs. The results indicate that knowledge of path information leads to better prediction than knowledge of simply the previous branch outcomes for a given number of history items. The results further show that even for equivalent hardware cost, path-based correlation often outperforms pattern-based correlation, especially when history information is periodically destroyed, for example, due to context switches.**

1. Introduction

With the number of instructions executed in parallel increasing in newer generations of microprocessors, there is increasing emphasis on ensuring that useful work is available for functional units at every cycle. This implies that disruptions to the smooth execution of instructions in processor pipelines must be kept as low as possible -- a stall of one cycle due to a disruption results in a potentially greater amount of work lost when the degree of parallelism in a processor is higher. Disruptions to the flow of instructions generally occur due to limitations in the number of resources, latency in operations, or dependences between instructions. In this paper, we study one of the more common forms of disruption involving conditional branch instructions.

Conditional branch instructions, for example on the IBM PowerPC [PowerPC 1994], test some condition to determine whether the instruction to be executed next is the immediately following one (the fall-through instruction) or some other instruction (the taken instruction). The target instruction for a taken branch may be encoded either in the branch instruction itself or may be contained in some other resource. On the PowerPC, this is one of two special purpose registers -- the link register or the count register. The tested condition may be either a bit in the condition register, the zero/non-zero value of the count register, or a combination of the two. A branch instruction that tests the count register has a resource dependence on the previous execution of the same instruction because this instruction decrements the count register as a side effect.

In dynamically dispatched out-of-order superscalar processors [Johnson 88], the number of parallel instructions executed per cycle is maximized by examining a large window of instructions spanning more than one basic block. Thus branch instructions which terminate basic blocks are typically preprocessed to anticipate basic blocks that are to be executed next. When branches are unconditional, and the address of the next basic block is encoded as a relative value from the address of the branch instruction, it is relatively straightforward to ensure that the analyzed window includes instructions from the target stream. However, when a conditional branch is encountered (or when the target is encoded in one of the special purpose registers), the window can include instructions from the target stream only if the tested condition has already been resolved (or if the register has been loaded with the target address). Rather than wait for the condition to be resolved, most modern processors attempt to predict the direction in which the condition will be resolved (or the target address in the register) and take measures to back up appropriately when a misprediction occurs.

The compiler assists in reducing the penalty due to mispredictions by attempting to separate the setting of the condition as much as possible from the dependent branch that tests the condition. The PowerPC architecture also allows the compiler to set a bit in the branch instruction if the instruction is expected to behave opposite to the conventional backward-taken-forward-not-taken assumption. In most cases these hint bits are obtained by analyzing performance on sample data which may not be representative of the actual data on which the program is eventually executed. Moreover, even when such static hints are provided, the dynamic behavior of programs may be different from that predicted statically, either because the compiler does not have enough knowledge about specific regions of behavior of the program, or because a simple bit does not adequately capture the behavior of a given branch.

Dynamic branch prediction [Smith 81] attempts to overcome these limitations by attempting to learn the behavior of branches at run time. Until recently, the usual method of predicting the direction of a branch at run time was to associate a finite-state machine as a predictor for each branch or set of branches. The state of the predictor associated with a branch determines the prediction for that branch, while the actual direction of the branch determines the transition between states. For example, in a 1-bit predictor, there are two states, one which is reached because the last occurrence of the branch was taken and another because it was not taken (fall-through). The first state is also used to predict that the next occurrence of the same branch will be taken, while the second is used to predict a fall-through. In practice, several researchers [Smith 81, Lee and Smith 84] have determined that it is more cost-effective to have a 2-bit finite-state machine as a predictor. Such a predictor tends to smooth out local variations in the behavior of a single branch, while not having too long a learning time. Studies [Nair 1995] have shown that one of the best 2-bit predictors across a wide range of applications is the saturating up-down counter, which requires two mispredictions before reversing its prediction.

It is quite common for implementations to limit the size of the table containing the 2-bit predictors, allowing multiple branches to share the same predictor. For example, a predefined set of b bits in the address of the branch could be used to index into a table of 2^b predictor entries. The PowerPC 604 [PowerPC604 1994], for example, has a 512-entry table indexed by bits 21-29 of the 32-bit branch address.

Recently, more sophisticated predictors have been proposed. These are based on the idea that the direction of some conditional branches is better predicted using multiple predictors, the choice of predictor being determined by the history of the immediately preceding branches. In the *global pattern* scheme [Pan et al 1992], also sometimes referred to as the *IBM correlation* scheme, for example, there are several predictor tables, each of which can be indexed by b bits of a branch address. The global pattern of branch outcomes is used to determine which among these predictor tables is to be used for the next branch. The observation was that as b is increased or as the length of the pattern history is increased, the prediction accuracy asymptotically reaches higher levels than what is possible without such a pattern history. The global pattern scheme is a specific example of a *two-level adaptive* scheme [Yeh and Patt 1991, Yeh and Patt 1993], where the predictor to be used for a given branch is not unique, but is determined both by the identity of the branch itself and by a pattern history. The pattern history may be the pattern of outcomes of either that branch or of all conditional branches recently executed.

All dynamic branch prediction schemes reported in the literature to date have used the pattern of directions taken by branches to define the context for the next prediction. In this paper we present the results of a study which uses a representation of the sequence of basic-block addresses to define the context for the next prediction. In [Young and Smith 1994], a similar scheme was proposed for static prediction. Such schemes are referred to as *path-based correlation* schemes as opposed to the *pattern-based correlation* schemes described in the last paragraph. We analyze hardware path-based schemes using instruction traces from common benchmark programs. The metrics we use emphasize the effects of misprediction on the performance of the system, rather than simply the misprediction rate. We also examine the effect of periodic flushing of the state of the prediction table with a view to understanding the sensitivity of the predictors to context switches.

Section 2 describes our modeling technique and the metrics used for our evaluations. In Section 3 we describe our path-based correlation technique. Section 4 discusses the performance of the path-based correlation technique in comparison with the traditional single-level 2-bit predictors as well as with the global pattern-based correlation scheme. Finally Section 5 presents some conclusions from our work.

2. Performance Modeling

In order to evaluate the relative performance of branch prediction strategies, we performed some experiments using instruction traces from execution of programs on the RS/6000. Floating point benchmarks tend to have well-predictable loops and hence were not considered. All benchmark programs in SPECint92 were studied. In addition, a C++ program, *idraw*, was also studied, because this program, based on the InterViews toolkit [Linton et al, 1989], appears representative of object-oriented programming, and exhibits branching characteristics different from programs in the SPEC suite.

Traces were gathered using *xtrace*, a tool which instruments RS/6000 executables to enable the collection of accurate instruction and data reference traces during program execution. Except for *idraw*, analysis was performed on-the-fly to avoid having to save large amounts of trace information. When available, the input data set for which a trace was gathered was that provided with the benchmark. In a couple of cases (*gcc* and *eqntott*), smaller input sets were created very similar to the reference input sets to keep the size of the traces manageable while ensuring that the results were representative. For *idraw*, which is interactive in nature, a range of editing and browsing operations were performed and the trace saved to ensure consistency between runs. The size of trace gathered for each benchmark is shown in Table 1. It should be noted that these traces include execution of shared library code, but do not include execution of operating system kernel code servicing the application.

TABLE 1 **Sizes of traces gathered for various benchmarks**

	Benchmark	Instructions traced
SPECint92	compress	82,534,946
	gcc	112,353,148
	espresso	133,900,249
	eqntott	97,373,971
	sc	267,147,251
	xlisp	51,574,757
C++	idraw	64,417,207

The trace analysis program examines each instruction in turn and analyzes it to determine the resources it *uses* to generate its results. Each result essentially *defines* some resource. A resource is typically a register, like a general purpose integer register, or a floating point register. The condition register is broken up into 8 resources corresponding to fields that may be independently set or examined by instructions. For each conditional branch instruction, we define the *slack* of the instruction as the number of instructions between it and the last instruction that it depends on. We could also define the *condition code latency* for a processor as the minimum number of instructions that must intervene the setting of the condition code and a dependent branch in order that there be no stalls in the processor due to the conditional branch. In a simple pipelined machine this is the latency of the pipeline from the start of the “instruction fetch” stage to the start of the “instruction execute” stage. For the RS/6000, for example, the condition code latency is 3, implying that if there are fewer than three, say i, instructions intervening the condition and the branch, there would have to be a stall for $3 - i$ cycles before execution resumes along the resolved path. Alternatively, one could say that if execution always proceeded along a predicted branch path, there would be a penalty of $3 - i$ cycles to refill the pipeline, if the prediction turned out to be wrong.

Most branch prediction studies reported in the literature compare performances of various schemes by comparing the success rate in predicting branches. However the preceding discussion indicates that prediction miss rate does not translate linearly to the penalty due to misprediction. To evaluate different branch prediction schemes in this study we compare *misprediction penalties*, where the misprediction penalty for a single branch is defined as $k - i$, k being the implementation-dependent condition code latency, and i being the slack for the branch instruction. $k - i$ is meaningful only when it is positive. When it is negative, the slack for the instruction is greater than the condition code latency, implying that there is no penalty in waiting until the condition is resolved. The misprediction penalty for the entire application is defined as the sum of misprediction penalties for all branches divided by the total number of instructions executed by the application.

If all instructions in an instruction trace were capable of executing in a single cycle, the misprediction penalty per 100 instructions would represent the percentage degradation in performance due to misprediction. However, this number should be regarded as an upper bound for performance degradation in a scalar processor, because instruction latencies and cache misses make it hard to achieve an average performance of 1 cycle per instruction. The graph of Figure 1 shows miss rate and penalty for our benchmarks for a $3 - i$ model using a traditional prediction table. We note that while there is some correlation between the miss rate and penalty, a comparison of the performance of *sc* and *idraw* indicates that a higher miss rate in branch prediction for an application does not necessarily mean a higher performance degradation. A lot depends both on the actual fraction of total instructions that are conditional branches, as well as on the ability of the compiler to schedule branches appropriately.

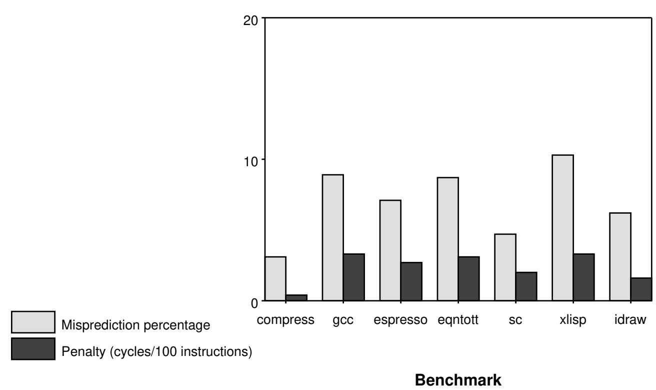

FIGURE 1 **Relating branch prediction miss rate to misprediction penalty for a *3-i* model**

The number of instructions that must intervene a condition setting and the dependent branch to ensure zero misprediction penalty depends on the actual instructions that intervene. If the condition code latency for a processor is 3, it suffices to have 3 intervening instructions for a single dispatch machine, 6 for a machine that can dispatch and execute 2 instructions each cycle and $3p$ instructions for a superscalar machine with p functional units. In such a superscalar machine, there would be, at most, a 3-cycle penalty for a conditional branch with a slack less than p, a 2-cycle penalty if the slack were less than $2p$, and a 1 cycle penalty if the slack were less than $3p$. If B_i represents the number of dynamic branches with a slack of i, and N is the total number of instructions, the misprediction penalty per instruction will now be given by

$$\frac{\sum_{i<3p} B_i}{N/3}$$

A reasonable approximation to this can be obtained by simply using the $k-i$ model with $k = 3p$. By plotting the misprediction penalties for various values of k, one could characterize the penalty for machine implementations with different number of functional units and different pipeline depths. We will henceforth refer to k, the product of these two quantities, as the *minimum slack for no penalty.*

Figure 2 shows the effect of increasing k in the $k-i$ model. As k increases, the number of conditional branches that need to be predicted increases because fewer conditions get resolved in time. This leads to an increase in the miss rate, which saturates as almost all conditional branches become candidates for prediction. The performance penalty however keeps increasing. This is because, as k increases, each mispredicted branch contributes a proportionally greater amount to the total penalty, leading to the asymptotically linear part of the penalty curve for high k. What is important to observe, in light of the discussion of the previous paragraph about multiple functional unit machines, is that even at fairly low levels of parallelism, longer pipeline latencies, and hence larger penalties due to misprediction, lead to alarmingly high performance degradation. Schemes which predict better would therefore be useful in such machine organizations.

As mentioned earlier, the indexing scheme used to access the predictor table is usually quite simple -- the low order b bits of the branch address are used for indexing into a 2^b-entry table (In th PowerPC, the instructions are word-aligned and hence the last two bits are not included in b). This implies that the working set of an application should be reasonably small to avoid contention in the table. Contention also causes problems when the processor switches between multiple processes running on the system. Branch prediction for a given application could therefore be less effective at rapid context switching rates, either because the application has a large working set which needs to be initiated in the table after each context switch or because the branch prediction scheme itself has a long learning overhead.

A good idea of the robustness of branch prediction to context switches may be obtained by simply clearing the prediction table at periodic intervals [Nair 1992, Perleberg and Smith 1993] and examining the sensitivity of the misprediction penalty to the flushing interval as shown in Figure 3. The figure shows that the penalties are very high at flush rates of once per 1000 instructions, but decreases with a slope that differs with applications. Applications with larger working sets like *gcc* and *idraw* are seen to be more sensitive to the flush rate than others. In comparison, *compress*, which has a very small working set, shows a rather flat characteristic for its penalty. .

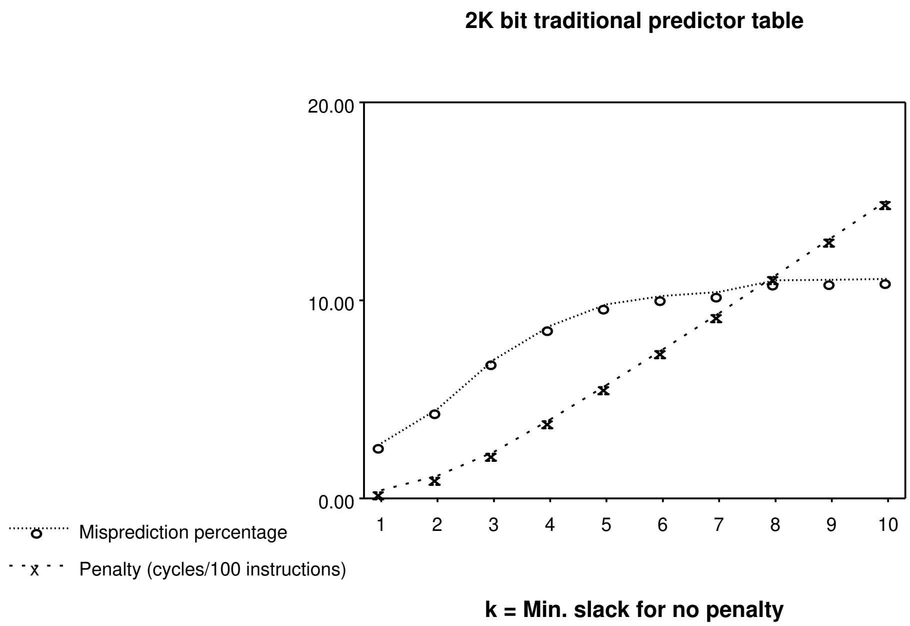

FIGURE 2 **Effect of varying *k* in *k-i* model for a 2K bit predictor table using 2-bit counters.** The results shown are arithmetic averages of the results for all benchmarks

We also observed that simpler predictors perform better at higher flush rates because they tend to have smaller "learning" or "warm-up" times. For example, Figure 4 shows that for 2K bits in the prediction table, a 1-bit predictor does better at flush rates of once per 1000 instructions than a 2-bit predictor which in turn does better than a global pattern-based correlation technique. The order reverses when context switches become infrequent.

3. Dynamic Path-Based Branch Correlation

The key components of a global-pattern based branch correlation scheme is a pattern history register and an array of predictor tables as shown in Figure 5. The pattern history register (PHR) is essentially a g-bit shift register representing the direction taken by the immediately preceding g conditional branches. The value encoded by the PHR is used to determine which of 2^g predictor tables should be used for the next prediction. The chosen table is then indexed using b bits from the branch that is to be predicted.

The basic observation behind pattern-based correlation is that some branches can be more accurately predicted if the pattern of directions taken by the immediately preceding branches is also known. This pattern is, in a sense, an indication of the path taken by the program in arriving at this branch. The path information can therefore be used to choose different predictors for each branch.

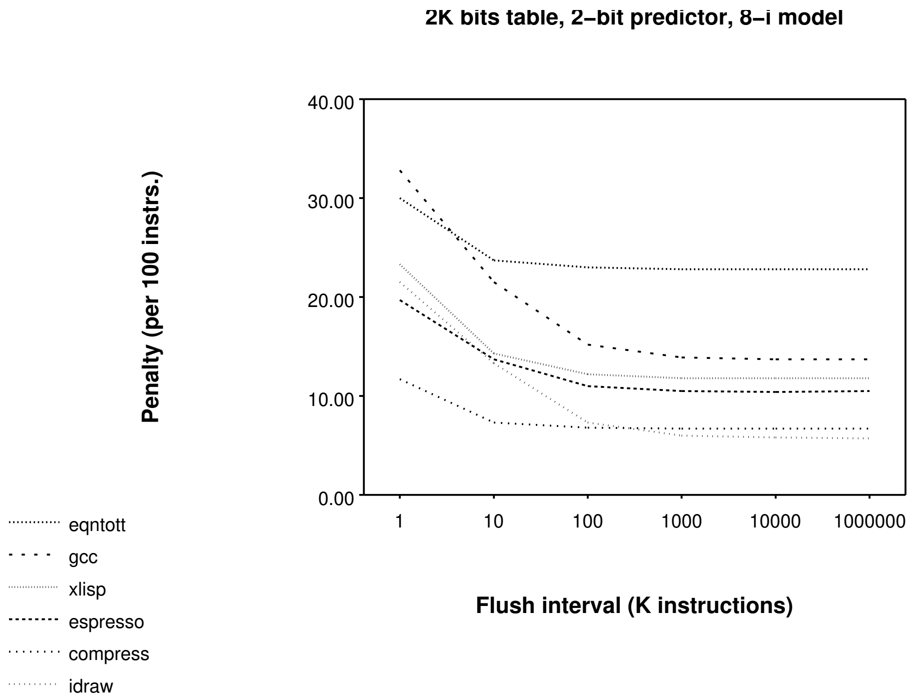

FIGURE 3 **Variation of penalty with periodic flushing of predictor table**

From the observed performance of global pattern-based branch prediction one expects that better performance would result if the actual path information were used instead of the pattern of the branch outcomes. It is easy to construct *pattern aliasing* situations [Young et al 1995], where paths leading to a branch are predictable whereas the pattern of branch outcomes for these paths are not.

Figure 6 shows a modification to the scheme of Figure 5 where the pattern history register is replaced by a path history register. Ideally, this register should contain a distinct value for each distinct path leading to the branch which needs to be predicted. To limit the length of this register (and thereby the number of prediction tables), we keep the history of only the last few globally encountered conditional branches. Further, only a few bits are used in the representation of each of these history items.

The path encoding scheme that we experimented with involved concatenation of bits from the addresses of immediately preceding basic blocks to make up *g* bits of the path history register. If g_1 bits are used to represent the last encountered basic block, g_2 bits for the one before that, and so on, it is reasonable to restrict $g_1, g_2, \ldots, g_p$ to the relation $g_1 \geq g_2 \geq \ldots \geq g_p$, both because it simplifies hardware, and because a basic block is likely to have a smaller effect on a branch further away from it than one closer to it. When $g_1 = g_2 = \ldots = g_p = q$, the path history register becomes a simple shift register which is shifted right by *q* bits when *q* bits of the next basic block address are shifted in, as shown in Figure 6.

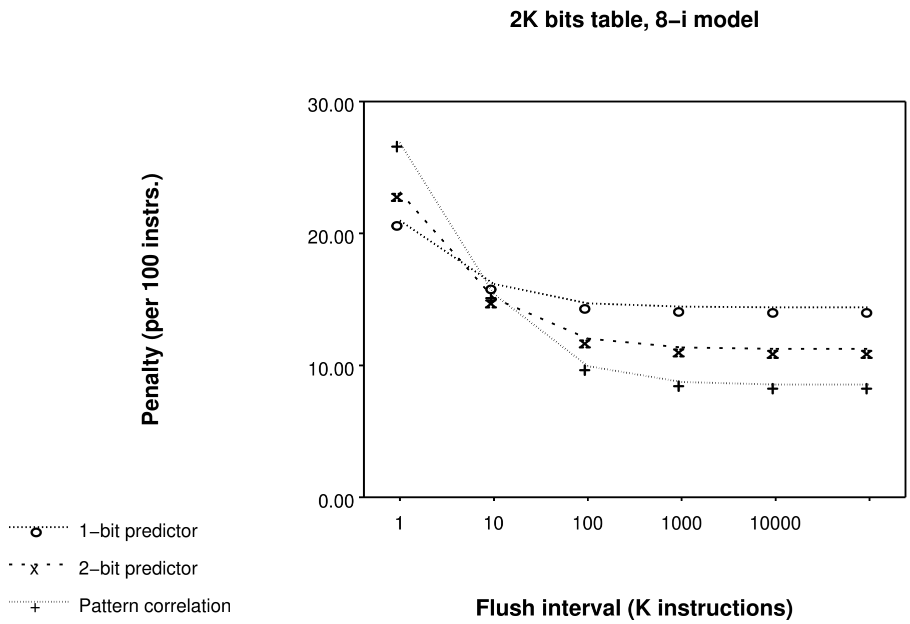

FIGURE 4 **Trading off asymptotic accuracy with "learning" time**

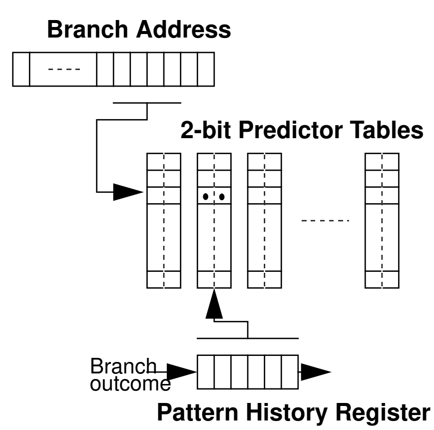

FIGURE 5 **Logical Organization of a Pattern Correlation Scheme**

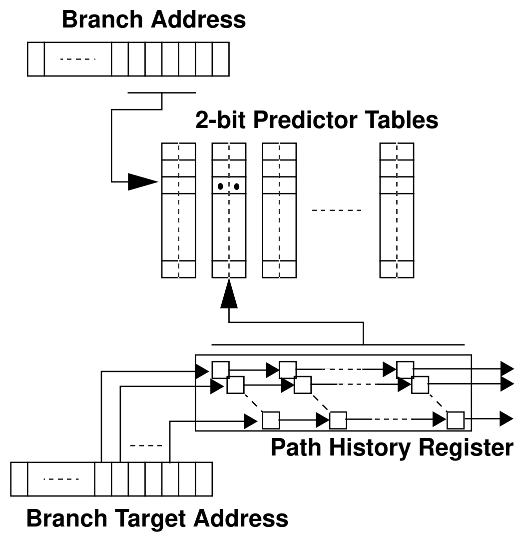

FIGURE 6 **Logical Organization of a Path Correlation Scheme with Equal Bits per Address**

Restrictions used in representing the path, both in the number of basic blocks in the path and in the bits used to represent the basic block, could result in *path aliasing*, the inability to distinguish between two distinct paths. In static restructuring of code [Young and Smith 1994] the compiler could maintain data structures for paths with many more bits representing each path. Path aliasing is therefore not as much of a problem in static as in dynamic path-based correlation. The key question then is whether the improved performance of unaliased static path-based correlation [Young et al 1995] is also noticeable in hardware implementations of path-based correlation.

We must mention here that the path sequence is defined by addresses of basic blocks rather than addresses of conditional branches in the path. There are situations, for example in *eqntott*, when the target of a taken branch is precisely the branch which terminates the fall-though path of the same branch. In such cases the same sequence of conditional branch addresses represents different sequences of basic blocks. In order to get a more accurate representation of the sequence of basic blocks, therefore, we used the address of the target (next executed instruction) of each branch, rather than the address of the branch itself. (In [Young et al 1995] the tuple consisting of the address of the branch and the direction of the outcome of the branch was used to solve this problem.)

4. Performance of Dynamic Path-Based Correlation

Figure 7 shows that both correlation schemes are better than the traditional 2-bit scheme in 5 out of 7 benchmarks. As explained in [Young et al 1995] correlation schemes essentially divide the streams of instructions leading to each branch into substreams based on the path history. *gcc* and *idraw* have larger working sets than the other benchmarks, and

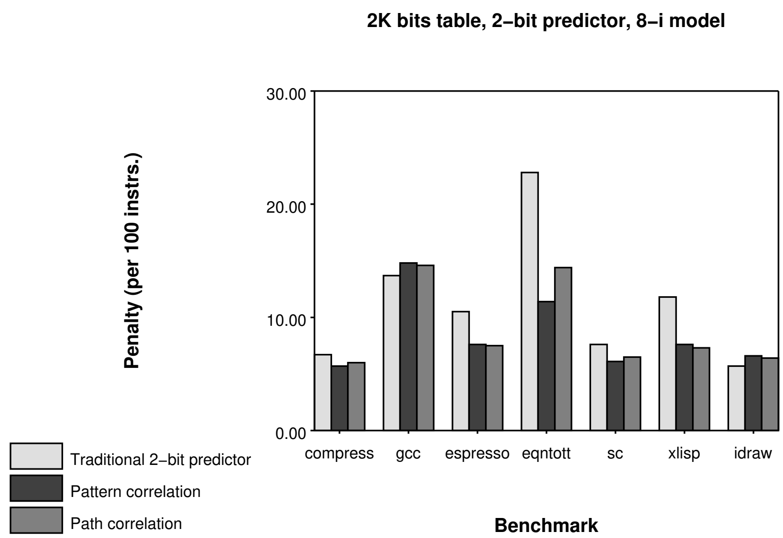

FIGURE 7 **Relative performance of path-based correlation on various benchmarks**

it appears that for the chosen hardware point, the collision in the predictor tables due to the larger number of substreams offsets the gains due to correlation information.

For a fixed amount of hardware in the prediction tables, path-based correlation uses a smaller history than pattern-based correlation because the same number of bits represents fewer basic blocks in the path history register than branch outcomes in the pattern history register. Despite this, Figure 7 shows, that path-based correlation is better than pattern-based correlation for 4 out the 7 benchmarks.

The fewer number of history items in the path correlation scheme for the same size of history register suggests that the learning overhead of this scheme should be lower than that of the pattern correlation scheme. One expects then that path correlation should do better than pattern correlation at higher flush rates. Figure 8 verifies this for flush rates of once per 1000 instructions and once per 10000 instructions. In [Perleberg and Smith 1993] it is observed that context switching intervals as short as 3000 instructions have been observed in mainframes. With the increasing adoption of microkernel technology, one expects this to be true even in commercial workstation environments. Similar behavior should also be expected in object-oriented frameworks and in interactive environments that make heavy use of dynamic linked libraries.

Previous studies on pattern correlation have suggested that the asymptotic performance (i.e. when a lot of hardware can be devoted to branch prediction) of pattern correlation is higher than that of traditional simple schemes. The graphs in Figure 9 show that the performance of path-based correlation is virtually identical to that of pattern-based correlation when only one bit is devoted for each address in the path, and that path-based correlation outperforms pattern-based correlation as the number of bits devoted to each

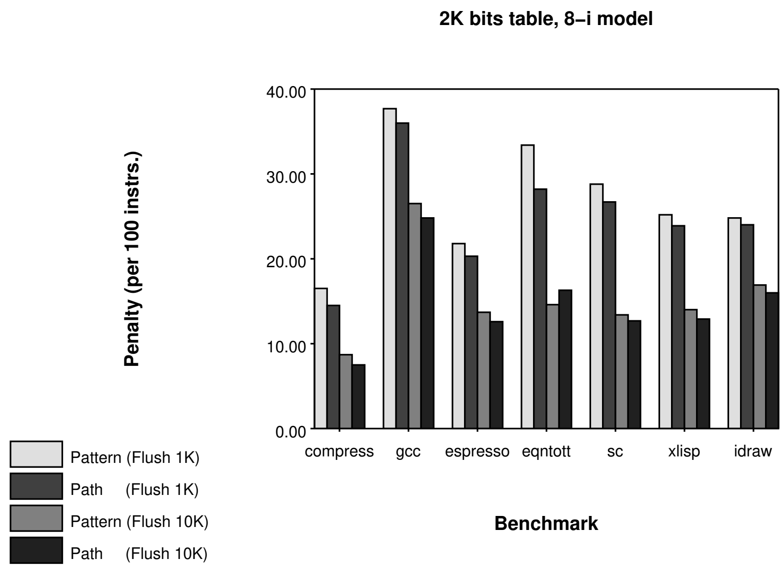

FIGURE 8 **Relative performance of correlation schemes at high flush rates**

item increases. Computing resources prevented us from verifying this for more than 4 items per history.

Throughout this paper when comparing schemes, we have used 2K bits as the allocated hardware. This appears to be the amount that current processors are willing to devote to branch prediction. Figure 10 shows the performance of the same benchmarks when 8K bits are available for branch prediction hardware. If one believes that learning overhead due to frequent context switches is going to play a big part in future commercial systems, there is benefit to be gained from using dynamic path-based branch prediction in future processor implementations.

5. Conclusions

High performance superscalar microprocessors achieve their performance either by using short pipe stages and high frequency clocks, eg. DEC Alpha [Rubinfield 1994], or by having a large number of functional units that can process instructions in parallel, eg. PowerPC 620 [Levitan et al 1995]. Some processors, eg. Intel P6 [Colwell and Steck 1995], attempt both. Both solutions rely heavily on the ability to keep fetching instructions that are most likely to be executed. Thus branch prediction plays an important role in all high-performance superscalar processors.

In this paper we have reported results from an extensive number of experiments performed using trace-driven simulation to study branch prediction algorithms. We described our metrics for evaluating the performance of branch prediction schemes and

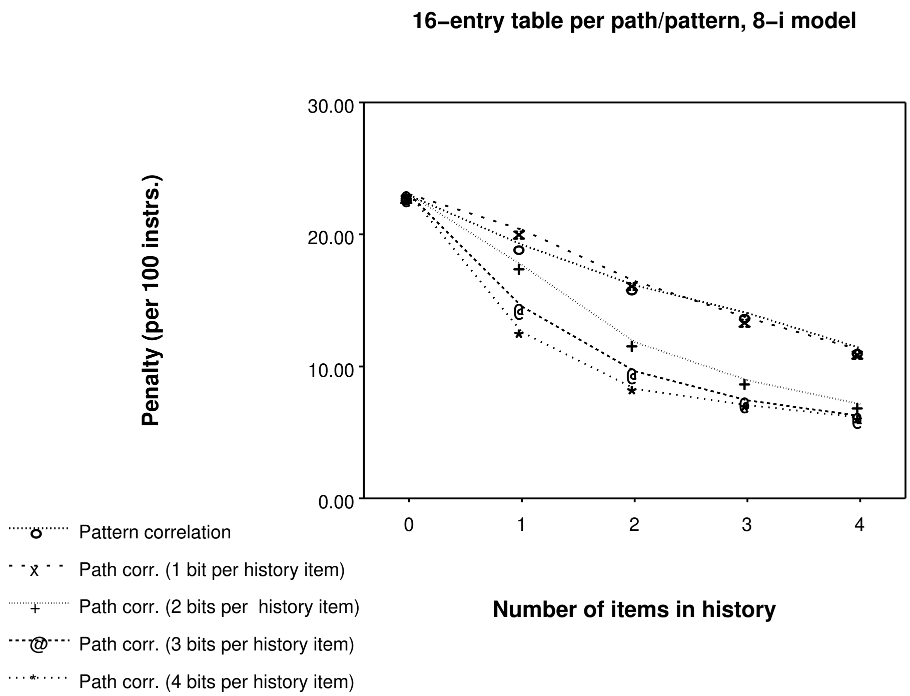

FIGURE 9 **Comparison of correlation schemes for various numbers of history items**

also proposed a new dynamic path-based correlation scheme. Below is a summary of our findings.

1. An important measure of how a branch prediction scheme performs is the extent to which the performance of an application is degraded when the scheme mispredicts. The $k - i$ model provides a better indication of the performance implications compared to the more commonly used metric, the branch prediction miss rate. We suggest that with appropriate interpretation of k, this model may be used even for processors with multiple functional units. We must emphasize though that an accurate prediction of the performance of a scheme must use more sophisticated (and hence more computationally expensive) cycle-by-cycle simulation models.
2. It is necessary to distinguish the asymptotic performance of branch prediction schemes from their actual performance in real situations. The real performance for a a given scheme for a given application depends on the amount of hardware that is devoted to the branch prediction hardware and on the ability to retain “learnt” information across context switches. Algorithms which perform well asymptotically tend to require more hardware, and tend to depend on high reuse of “learnt” information. Thus, in situations where the working set of applications is large, or when a larger number of instructions have to be executed before predictions become accurate, these algorithms may suffer due to aliasing problems or due to learning overhead. This was illustrated in this study by using a range of predictors, ranging from a simple 1-bit predictor which remembers only the last outcome of a branch, to sophisti-

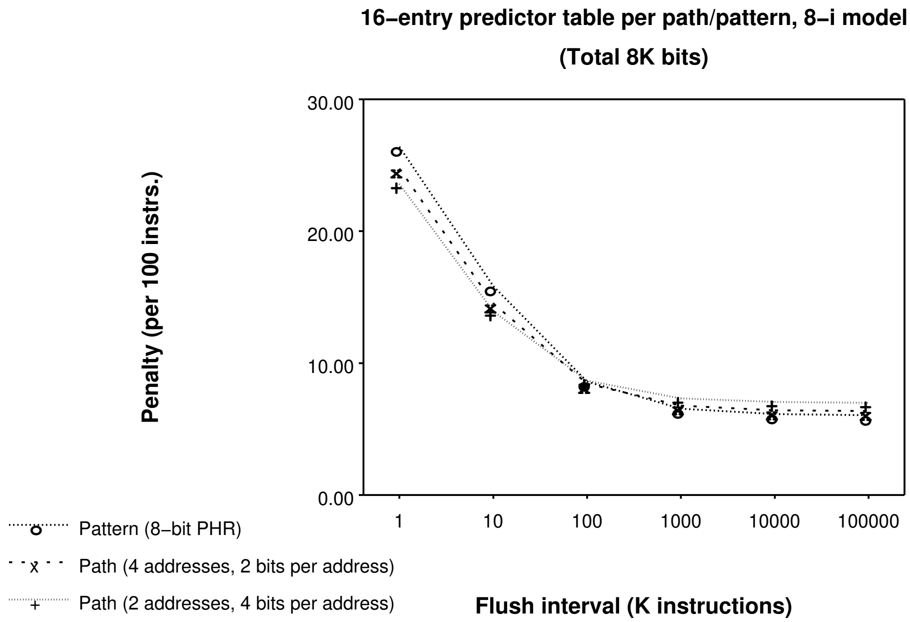

FIGURE 10 **Comparison of correlation schemes at various flush rates**

cated pattern-based correlation schemes which have more sophisticated learning algorithms.

3. We introduced a simple hardware scheme for using path information for branch correlation, rather than using the pattern of outcomes of previous branches. We showed that, for a given number of items, path history provides a better basis of prediction compared to pattern history. However, some of this advantage is offset by the fact that more bits are needed to represent path history compared to that needed to represent pattern history. Thus, while the performance of path and pattern correlation schemes are equivalent for equivalent hardware, significant benefit from the additional information provided by path correlation is realizable only at a higher hardware cost.
4. For fixed-size history tables, the choice of branch prediction scheme depends on the expected frequency at which the learnt information is destroyed either due to a context switch or due to address aliasing (usually as a result of a large working set). Schemes which rely on less history are preferable where this frequency is high. It may be possible to devise schemes which adapt appropriately to the situation, perhaps at the cost of some amount of sophisticated control. For a non-adaptive scheme, path-based correlation with fewer history items provides a good compromise between fast learning and top performance.

A good area for further study would be to devise better schemes to encode path information compared to the simple concatenation of bits from addresses along the path. Note that the obvious technique of XORing addresses along the path is not suitable, as it does not encode the *sequence* information which characterizes a path.

The choice of branch prediction algorithms can indeed influence to some extent the penalty due to unknown control flow information. However, in the absence of hardware to execute both legs of a branch concurrently, greater benefit appears likely from reducing k or by increasing i in the $k - i$ model. Thus, keeping condition code latency low (reducing k) and scheduling instructions for higher slack (increasing i) become even more important at high degrees of parallelism.

6. References

[Colwell and Steck 1995] R. Colwell and R. Steck, "A 0.6um BiCMOS Processor with Dynamic Execution," *Proc. ISSCC*, 1995.

[Johnson 1991] M. Johnson, " Superscalar Microprocessor Design," Prentice Hall, Englewood Cliffs, NJ, 1991.

[Lee and Smith 1984] J. Lee and A. Smith, "Branch Prediction Strategies and Branch Target Buffer Design," *Computer*, vol. 17, No. 1, Jan. 1984.

[Levitan et al 1995] D. Levitan, T. Thomas, and P. Tu, "The PowerPC 620 Microprocessor: A High Performance Superscalar RISC Microprocessor," *Proc. Spring COMPCON*, 1995.

[Linton 1989] M. A. Linton, J. M. Vlissides, and P. R. Calder, "Composing User Interfaces with Interviews," *Computer*, vol. 22, no. 2, pp. 8-22, Feb. 1989.

[Nair 1995] R. Nair, "Optimal 2-Bit Branch Predictors," *IEEE Transactions on Computers*, vol. 44, no. 5, pp. 698-702, May 1995.

[Nair 1992] R. Nair, "Branch Behavior on the IBM RISC/System 6000, " Research Report RC 17859, IBM T. J. Watson Research Center, Yorktown Heights, NY, April 1992.

[Pan et al 1992] S. Pan, K. So, and J. Rahmeh, "Improving the Accuracy of Dynamic Branch Prediction using Branch Correlation," *Proc. 5th Annual Intl. Conf. on Architectural Support for Prog. Lang. and Operating Systems*, Oct. 1992.

[Perleberg and Smith 1993] C.H. Perleberg and A. J. Smith, " Branch Target Buffer Design and Optimization," *IEEE Transactions on Computers*, vol 42, no. 4, pp. 396-412, April 1993.

[PowerPC 1994] The PowerPC Architecture: A Specification for a New Family of RISC Processors, Ed. C. May et al, Morgan Kaufmann Publishers, Inc., San Franciso, CA, 1994.

[PowerPC601 1994] PowerPC 604 Risc Microprocessor User's Manual, IBM Microelectronics, Essex Junction, VT, 1994.

[Rubenfield 1994] P. Rubinfield, "An Overview of the Alpha AXP 21164 Microarchitecture," *Proc. Hot Chips VI Symposium*, Oct. 1994.

[Smith 1981] J. Smith, "A Study of Branch Prediction Strategies," *Proc. 8th Annual Intl. Symp. on Computer Architecture*, June 1981.

[Yeh and Patt 1991] T. Yeh and Y. Patt, " Two-Level Adaptive Training Branch Predicttion," *Proc. 24th Annual ACM/IEEE Intl. Symp. and Workshop on Microarchitecture*, Nov. 1991.

[Yeh and Patt 1993] T. Yeh and Y. Patt, "A Comparison of Dynamic Branch Predictors that use Two Levels of Branch History," *Proc. 20th Annual Intl. Symp. on Computer Architecture*, May 1993.

[Young and Smith 1994] C. Young and M. Smith, "Improving the Accuracy of Static Branch Prediction Using Branch Correlation," *Proc. 6th Annual Intl. Conf. on Architectural Support for Prog. Lang. and Operating Systems*, Oct. 1994.

[Young et al 1995] C. Young, N. Gloy, and M. Smith, "A Comparative Analysis of Schemes for Correlated Branch Prediction," P*roc. 22nd Annual Intl. Symp. on Computer Architecture*, June 1995.