

Dynamic Path-Based Branch Correlation

Ravi Nair

IBM Thomas J. Watson Research Center

P. O. Box 704, Yorktown Heights, NY 10598.

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. A stall of one cycle due to a disruption results in a potentially greater amount of work lost when the available parallelism in a processor is higher. In this paper, we study one of the more common forms of disruption in processor pipelines - that involving conditional branch instructions.

Conditional branch instructions, for example on the IBM PowerPC [1], 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, eg. the link register or the count register on the PowerPC.

In dynamically dispatched out-of-order superscalar pro-

cessors [2], 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 a register), 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 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 [3] 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 asso-

ciate 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, each indicating both the direction taken by the last instance of the branch as well as the prediction for the next. In practice, several researchers [3], [4] 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 [5] 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 [6], 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 [7], 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 [8], [9], 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 correlation schemes reported in the literature to date have used the pattern of directions taken by branches to define the context for the next prediction. This paper presents the results of a study which uses a representation of the sequence of basic-block addresses to define the context for the next prediction. The study uses trace-driven analysis of such a scheme using instruction

traces from common benchmark programs. The metrics used emphasize the effects of misprediction on the performance of the system, rather than simply the misprediction rate. The effect of periodic flushing of the state of the prediction table is also examined with a view to understanding the sensitivity of the predictors to context switches. A similar scheme for static prediction was proposed in [10] where the term *path-based correlation* was used to distinguish it from the *pattern-based correlation* schemes described in the last paragraph. The requirements for dynamic prediction are quite different. While static prediction is done in software, for example by the compiler, dynamic prediction is more severely constrained by hardware, both in terms of the space occupied and in the control complexity to access and update the prediction information.

2: Performance Modeling

2.1: Trace-driven analysis

The experiments described in this paper were performed using traces generated on IBM RS/6000 processors, which implement the Power architecture. The benchmark programs include all six programs in the SPECint92 suite as well as a C++ program, *idraw*. Based on the InterViews toolkit [11], *idraw* appears to be representative of object-oriented programming, and exhibits branching characteristics different from programs in the SPEC suite. Floating point benchmarks were not considered since most of them tend to have very predictable looping characteristics.

The traces include execution of shared library code, but do not include execution of operating system kernel code servicing the application. In order to keep trace size reasonable (50M-300M instructions) while capturing the behavior of the entire program, the input to each program was either a subset, or a modified version of the reference input provided with the benchmark.

Table 1: 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

2.2: Definitions

We define the *slack* of a conditional branch instruction as the number of instructions between it and the last

instruction that it depends on. We 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. Instead, 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). If B_i is the number of dynamic branches with slack i , and N is the total number of instructions in the application, the *misprediction penalty for the application* is defined as

$$\frac{\sum_{i < k} (k - i) B_i}{N}$$

Table 1 shows the miss rate and the misprediction penalty for a 2-bit predictor using a $3 - i$ model. 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. Unlike the miss rate, the misprediction penalty 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.

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

Table 2: Comparing branch prediction miss rate to misprediction penalty for a $3 - i$ model

Benchmark	Miss rate (%)	Misprediction penalty (per 100 instrs.)
compress	3.1	0.4
gcc	8.9	3.3
espresso	7.1	2.7
eqntott	8.7	3.1
sc	4.7	2.0
xlisp	10.3	3.3
idraw	6.2	1.6

average performance of 1 cycle per instruction, thus extending the available slack. Accurate analysis of the misprediction penalty requires a more sophisticated model than what was used in this paper. However we believe that the qualitative nature of the results is not affected by simply counting the number of intervening instructions to estimate the slack.

If there are p pipelines in a machine with condition code latency of κ , an upper bound on the number of intervening instructions required for no penalty is κp . The lower bound on the number of cycles required for executing N instructions in such a processor is N/p . Each conditional branch with a slack of i incurs a penalty of at most $\kappa - \lfloor i/p \rfloor$. The misprediction penalty per instruction is at most

$$\frac{\sum_{i < \kappa p} (\kappa - \lfloor i/p \rfloor) B_i}{N/p}$$

A reasonable approximation to this can be obtained by simply using the $k - i$ model with $k = \kappa p$. We henceforth refer to k , the product of the condition code latency and the pipeline depth, as the *minimum slack for no penalty*.

For a given trace, as k increases, the penalty incurred by each conditional branch with slack less than k increases. The total number of conditional branch instructions that incur a penalty also increases. However, while the penalty increases roughly linearly with k , the total branches incurring a penalty saturates to the traditionally studied miss rates as seen in the graph of Figure 1 which plots both measures for a 1K entry 2-bit predictor table. The curve for the misprediction penalty graphically emphasizes the importance of good branch prediction for future processors.

As mentioned earlier, the indexing scheme used to access the predictor table is usually quite simple -- the low

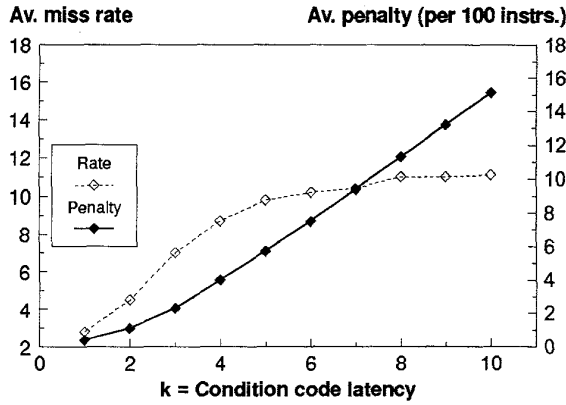


Figure 1: Effect of varying k for a 2K bit predictor table using 2-bit counters.

order b bits of the branch address are used for indexing into a 2^b -entry table (In the 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 training 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 [12], [13] and examining the sensitivity of the misprediction penalty to the flushing interval. Figure 2 shows the results obtained by varying the flush interval from 1000 instructions to 10 million instructions for a 1K entry 2-bit predictor table. We see that the penalties are very high at flush rates of once per 1000 instructions, but decreases with a slope that differs with applications. Applications like *gcc* and *idraw* with large working sets are seen to be more sensitive to the flush rate than others. Most studies of branch prediction schemes to date compare the efficacy of schemes only at infinite flush intervals where schemes requiring longer training generally perform better.

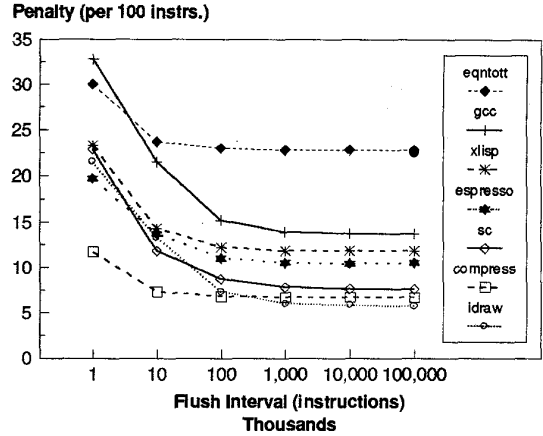


Figure 2: Variation of penalty with periodic flushing of predictor table

3: Dynamic Path-Based Branch Correlation

3.1: Organization

In order to understand the logical organization of a path-based correlation scheme, let us first look at one of several variations of the pattern-based correlation scheme. In Figure 3 we show the key components of a global-pattern branch correlation scheme. The pattern history register

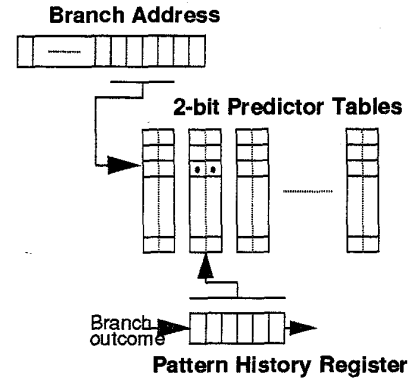


Figure 3: Logical organization of a pattern correlation scheme

(PHR) is essentially a g -bit shift register representing the directions taken by the immediately preceding g conditional branches. The value encoded by the PHR is used to

determine which of 2^8 predictor tables should be used for the next prediction. The chosen table is then indexed using b bits from the branch to be predicted.

This scheme can be adapted to implement a very simple path-based correlation scheme as shown in Figure 4. The

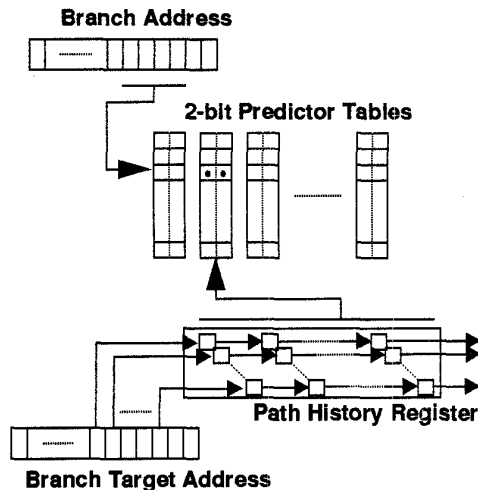


Figure 4: Logical organization of a path correlation scheme with equal bits per address

pattern history register is replaced by a g -bit path history register which encodes the addresses of the targets of the immediately preceding p conditional branches. The rest of the mechanism is identical to that of Figure 3.

Ideally, all significant bits of the target address should be used to ensure that each sequence of p addresses has a unique representation in the register. Hashing schemes such as those used in software are too expensive to implement in hardware. The scheme shown in Figure 4 is an easily realizable scheme which uses a subset of q bits from each of the target addresses, where $p \times q = g$.

We see immediately one major difference between the expected performance of path-based correlation as used in the static restructuring of code [10] and that of path-based correlation implemented in hardware. This is *path aliasing*, the inability to distinguish between two distinct paths leading to a branch. In addition, the hardware scheme could suffer from address aliasing resulting from the limited number of bits from the branch address used to index into the set of 2^8 predictor tables. The key question then is whether the improved performance of unaliased static path-based correlation [14] is also noticeable in hardware implementations of path-based correlation.

3.2: Characteristics

The basic observation behind both pattern-based and path-based correlation is that some branches can be more accurately predicted if the path taken by the program in arriving at this branch is known. Each branch could be associated with several predictors, rather than the single predictor used by static predictors or traditional 2-bit counter predictors. Path-based correlation attempts to overcome the limitations in performance of pattern-based correlation arising from *pattern aliasing* situations [14], where knowledge of the path leading to a branch results in higher predictability than knowledge of simply the pattern of branch outcomes along the path.

Consider the following code which has been observed at some subroutine call-return boundaries.

```
retval = B();
if (retval == 1) { /* a1 */
...
}
```

```
B: ....
    if (cond1) return 1;      /* b1 */
    ....
    if (cond2) return 0;      /* b2 */
    else return 1;
```

The direction of the last conditional branch before a return does not identify whether $(retval == 1)$, the condition $a1$, is true or not. It evaluates to true when the last branch is taken for the case of the return from $b1$, but when the last branch is not-taken for the case of the return from $b2$. If there is a pattern of taken/not-taken in the section immediately preceding the return from $b1$, which is distinguishable from that immediately preceding $b2$, a longer pattern history would enable better prediction of $a1$. However, in this example, a minimal information of the path, namely just the location of the basic block from which the return occurred, suffices to predict the direction of condition $a1$. There is evidence of decreasing percentage of conditional branches and increasing use of subroutine calls in modern code, especially object-oriented code [15], [16], suggesting the need to exploit path information more.

We must mention here that while it may be easier in hardware to use the branch addresses themselves as input to the path history register, we have found it important to use addresses of the target (the next executed instruction) instead. There are situations, for example in *eqntott*, where the target of a taken branch is precisely the branch which terminates the fall-through path of the same branch. In such cases the same sequence of conditional branch addresses

represents different sequences of basic blocks. This is also evident from the example above where the address of the branch corresponding to condition *b2* does not provide enough information to predict *a1*, whereas the target of the branch does. (In [14] 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 Path-Based Correlation

We will first compare the performance of various algorithms assuming no context switching during the run of the application. Later in Section 4.2 we will examine the performance of various schemes with context switching overhead.

4.1: Performance without context switching

We compared the performance of four schemes over the set of benchmarks described in Section 2. The first was a simple direct-mapped table with 2048 one-bit predictors. The second was a traditional predictor table having 1024 entries, each implementing the 2-bit up-down counter. The third was a pattern-based correlation scheme having 8 entries in each of 128 predictor tables ($b = 3, g = 7$), each entry again being a 2-bit up-down counter. The fourth was a path based correlation scheme having 16 entries (2-bit counters) in each of 64 predictor tables selected by choosing 2 low order bits from each of 3 previous target addresses ($b = 4, p = 3, q = 2, pq = g = 6$).

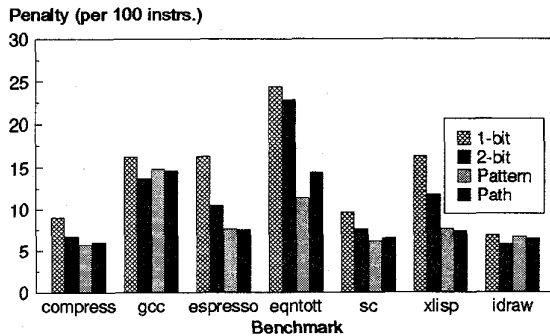


Figure 5: Relative performance of prediction schemes on various benchmarks

Figure 5 shows that both correlation schemes are significantly better than a 1-bit prediction scheme, and better in 5 out of 7 benchmarks than the traditional 2-bit scheme. The improvement is significant in the case of *xliisp* and *eqntott*. As explained in [14] 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 it appears that the collision in the predictor tables due to the larger number of substreams offsets the gains due to correlation information. Correlation will show improvement for these benchmarks only when the prediction hardware is increased beyond the currently feasible regions of 1K-4K bits.

Another disappointing observation from Figure 5 is that path-based correlation does not show any significant improvement over pattern-based correlation when it does show any improvement at all. This appears to be contradictory to the expectation generated by the static experiments of [14]. Our explanation for this is as follows. 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. Thus, the chosen simple encoding for the path information leads to an underutilization of the available substream representation for each branch. Despite this, path-based correlation is better than pattern-based correlation for 4 out of the 7 benchmarks, indicating that with a better hashing scheme, it is likely to consistently outperform the pattern correlation scheme.

A better hashing scheme would be able to record either more history items in the same size path register, or would encode more address bits for the same number of history items. We have not experimented with alternative hashing schemes, but an indication of its potential is evident from Figure 6. In this experiment we fixed the number of entries

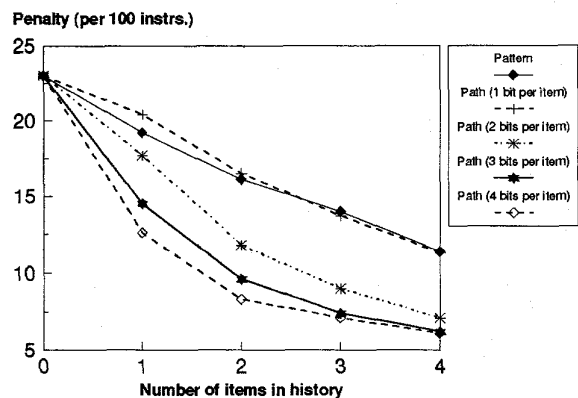


Figure 6: Comparison of correlation schemes for various numbers of history items

in each prediction table at 16 and varied the number of bits used for representing each history item in the path. Each point represents the average penalty over all benchmarks. The graphs 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 item increases. Moreover, the steeper nature of the curves for path correlation suggests that a small number of history items can exploit most of the benefit of this scheme.

In the graph of Figure 6, if we draw lines corresponding to equal pq products, where p is the number of history items and q is the number of bits per item, we notice that these lines are roughly horizontal. This explains why in the encoding of the implementation of Figure 4, the performance of path correlation is roughly the same as that of pattern correlation. It seems possible therefore, for path correlation to be more cost effective if we used fewer than pq bits to encode the same information, for example through an intelligent hashing scheme.

4.2: Performance with context switching

In Section 2 we suggested that periodic flushing of the state of the predictor tables may be a reasonable way to simulate the effect of context switching. For various schemes with the same hardware cost we flushed the contents at various intervals ranging from once every 1000 instructions to once every 100 million instructions. The hardware parameters were identical to those for Figure 5. The average over all benchmarks is depicted in Figure 7 for each scheme at various flush rates. It is interesting to observe how well the simple 1-bit predictor, which has very little learning overhead, performs at very high flush rates even though it does not perform asymptotically as well as the other predictors.

We also observe from Figure 7 that path-based correlation has an edge over pattern-based correlation even at flush intervals exceeding 10,000 instructions. Figure 8 verifies that this improvement is obtained across the board for almost all the benchmarks. The conclusion is that the fewer number of history items in the path correlation scheme for the same size of history register results in a learning overhead for this scheme that is lower than that of the pattern correlation scheme.

It is hard to determine which flush interval should be used as a basis for comparison. A larger flush interval (small flush rate) may be more typical of single user scientific workstations rather than of commercial environments. In [Perleberg and Smith 1993] it is observed that context switching intervals as short as 3000 instructions have been

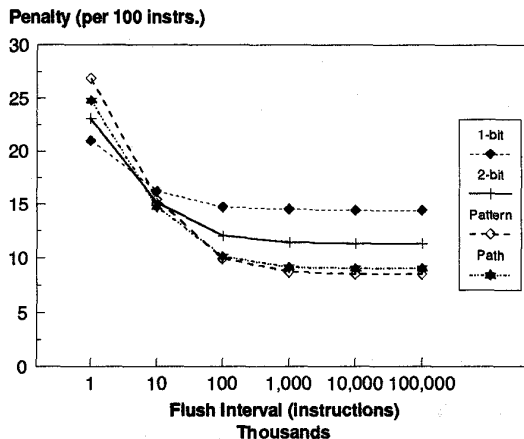


Figure 7: Trading off asymptotic accuracy with "learning" time

observed in mainframes. With the increasing adoption of microkernel technology, one expects this to be true also 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.

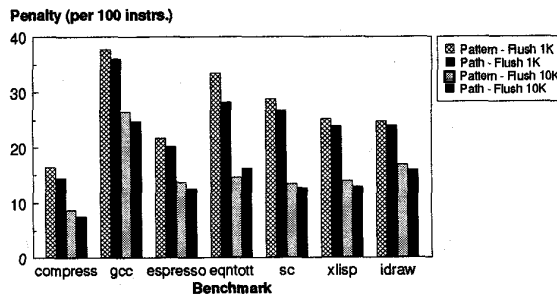


Figure 8: Relative performance of correlation schemes at high flush rates

4.3: Future Performance

We suggested in Section 4.1 that the current generation of processors is unlikely to devote much more area than the equivalent of about 4K bits for prediction hardware. However, as the number of functional units increases, and as the real estate available on a chip increases, it is reasonable to expect much larger predictor tables. We examined the performance of correlation schemes for a design point of approximately 32K bits. At this level of hardware, there

are several possible ways of organizing the hardware. For example, in the case of pattern correlation, it may be preferable to have more entries per predictor table, rather than increase the number of predictor tables. On the other hand, in path correlation, more hardware provides the opportunity to use more bits to represent the path, either by increasing the number of history items or by using more bits per history item. We experimented with both schemes with 16, 32 and 64 entries in each predictor table ($b = 4, 5, 6$), and, in the case of path correlation, various combinations of history items, p , and item sizes, q . In Table 3 below we show the best result obtained for each scheme and the parameter combination that produced this result. For example the (6,1) entry corresponding to *compress*

Table 3: Performance comparison for 64K bit predictor sizes (no context switching)

Benchmark	Pattern correlation		Path correlation	
	Penalty (per 100)	Best b	Penalty (per 100)	Best b, q
compress	5.5	4	5.6	6,1
gcc	8.7	6	9.3	6,1
espresso	5.0	6	5.3	6,1
eqntott	9.6	4	8.6	4,1
sc	3.9	4	4.0	4,1
xlisp	3.6	4	3.2	4,1
idraw	3.2	5	3.5	6,1

refers to 64 entries in each of 256 tables, the path history register having 8 bits, with 1 bit from each of 8 history items. A (6,3) entry would mean that 3 bits are used from each of two most recent history items, and 2 bits from the third to make up the required 8 bits.

It is interesting to observe that both schemes are fairly close in performance with pattern correlation appearing to have a slight edge. What is even more interesting is that the value of q which provided the best results in the case of path correlation is 1, indicating that in the tradeoff between history size and item size, history size wins.

The results are dramatically different when context switching is considered, for example using a flush rate of 1 per 10,000 as shown in Table 4. We notice now that path correlation not only performs better in all cases, but is consistently better by about 10% across the entire suite. Moreover, the bias towards smaller number of history items is evident by the consistent value of 3 for the best q .

The decision on which scheme is better for future processors therefore depends largely on the operating environment of the processor. Path correlation is suggested for commercial environments, pattern correlation for scientific workstations.

Table 4: Performance comparison for 64K bit predictor sizes (no context switching)

Benchmark	Pattern correlation		Path correlation	
	Penalty (per 100)	Best b	Penalty (per 100)	Best b, q
compress	9.5	6	7.5	6,3
gcc	27.4	6	24.6	6,3
espresso	13.9	6	12.5	6,3
eqntott	13.9	6	14.0	4,3
sc	13.7	6	12.2	6,3
xlisp	13.1	5	11.6	4,3
idraw	17.6	6	16.1	6,3

5: Conclusions

High performance superscalar microprocessors achieve their performance either by using short pipe stages and high frequency clocks, eg. DEC Alpha [17], or by having a large number of functional units that can process instructions in parallel, eg. PowerPC 620 [18]. Some processors, eg. Intel P6 [19], 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 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. 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 of 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 "learned" information across context switches. Algo-

rithms which perform well asymptotically tend to require more hardware, and tend to depend on high reuse of "learned" information. Thus, in situations where the working set of applications is large, or when a larger number of instructions has 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 correlation schemes which have more sophisticated learning algorithms.

3. We introduced a simple hardware scheme which used path information for branch correlation, rather than 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. The performance of path and pattern correlation schemes are equivalent for equivalent hardware.
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 very simple scheme described here. There are obviously many likely candidates for hardware implementation of sequence hashing. It is important to select one which performs consistently well both across a range of benchmarks as well as at different context switch intervals.

We end this paper by noting that it is important to supplement good branch prediction algorithms with both hardware efforts to reduce k (make condition code latency small) and software efforts to increase i (schedule instructions for higher slack) to keep the branch misprediction penalty, $k - i$, in future processors as low as possible.

5: References

- [1] The PowerPC Architecture: A Specification for a New Family of RISC Processors, Ed. C. May et al, Morgan Kaufmann Publishers, Inc., San Francisco, CA, 1994.
- [2] M. Johnson, "Superscalar Microprocessor Design," Prentice Hall, Englewood Cliffs, NJ, 1991.
- [3] J. Smith, "A Study of Branch Prediction Strategies," *Proc. 8th Annual Intl. Symp. on Computer Architecture*, June 1981.
- [4] J. Lee and A. Smith, "Branch Prediction Strategies and Branch Target Buffer Design," *Computer*, vol. 17, No. 1, Jan. 1984.
- [5] R. Nair, "Optimal 2-Bit Branch Predictors," *IEEE Transactions on Computers*, vol. 44, no. 5, pp. 698-702, May 1995.
- [6] PowerPC 604 Risc Microprocessor User's Manual, IBM Microelectronics, Essex Junction, VT, 1994.
- [7] 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.
- [8] T. Yeh and Y. Patt, "Two-Level Adaptive Training Branch Prediction," *Proc. 24th Annual ACM/IEEE Intl. Symp. and Workshop on Microarchitecture*, Nov. 1991.
- [9] 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.
- [10] 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.
- [11] 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.
- [12] 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.
- [13] 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.
- [14] C. Young, N. Gloy, and M. Smith, "A Comparative Analysis of Schemes for Correlated Branch Prediction," *Proc. 22nd Annual Intl. Symp. on Computer Architecture*, June 1995.
- [15] B. Calder, D. Grunwald, and B. Zorn, "Quantifying Behavioral Differences between C and C++ Programs," submitted to *J. Programming Languages and Design*, February 1994.
- [16] R. Nair, "Performance Evaluation of C++ Applications on the IBM RS/6000," available from author.
- [17] P. Rubinfield, "An Overview of the Alpha AXP 21164 Microarchitecture," *Proc. Hot Chips VI Symposium*, Oct. 1994.
- [18] D. Levitan, T. Thomas, and P. Tu, "The PowerPC 620 Microprocessor: A High Performance Superscalar RISC Microprocessor," *Proc. Spring COMPCON*, 1995.
- [19] R. Colwell and R. Steck, "A 0.6um BiCMOS Processor with Dynamic Execution," *Proc. ISSCC*, 1995.