RC 20537 (90751) 6/26/96
Computer Science/Mathematics 19 pages

IBM Research Report

The Hint Bit for Conditional Branch Instructions

Ravi Nair

IBM Research Division
T. J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

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

The Hint Bit for Conditional Branch Instructions

Ravi Nair
IBM Thomas J. Watson Research Center
P.O.Box 218, Yorktown Heights, NY 10598
nair@watson.ibm.com

June 17, 1996

Abstract

Misprediction of conditional branches is a major cause for reduced performance in high-performance processors. Recently there have been several branch prediction techniques proposed to improve the prediction accuracy of conditional branches. Most of these techniques are very expensive in their hardware requirements and incur a fairly significant amount of learning overhead. In this report we investigate the possibility of using a hint bit in conditional branch instructions to direct all branches to one of two dynamic predictors in an implementation. We identify two classes of predictors represented by this hint bit, and direct most of the branches to the simpler of the two predictors to ensure that the more sophisticated predictor is used mostly for the cases where its use is beneficial. We examine an algorithm to identify these classes and present results to demonstrate the effectiveness of the technique under various size constraints and context switching situations.

1. Introduction

The performance of any processor is dependent on the availability of a continuous supply of instructions for dispatch to the functional units. This supply becomes even more critical in the case of modern superscalar processors with large numbers of functional units, where a disruption in the instruction fetch pipeline results in a potentially greater amount of work lost. Cache misses and conditional branch mispredictions form the most common causes for such disruptions. We address only the problem of conditional branch mispredictions in this paper.

The dispatcher in a superscalar processor dynamically determines resource dependences between instructions to make available a set of instructions for parallel execution. The larger the window of instructions available for analysis by the dispatcher, the larger is the set of instructions that can be identified for parallel execution in a given cycle. This window of instructions can include instructions beyond conditional branches only if the condition tested by the branch has been resolved already. Rather than restrict the size of the window by closing it at conditional branches, most high-performance processors attempt to predict the direction in which the condition will be resolved and take measures to back up appropriately when a misprediction occurs.

There are basically two types of branch prediction, *static* and *dynamic*. We refer the reader to [Lee and Smith 1984] for a general discussion of early branch prediction schemes. Most of the work in *static branch prediction* is done by the compiler, which provides information in the instruction stream as to whether a specific branch is likely to be taken or not taken. In *dynamic branch prediction*, hardware structures in the processor determine the likelihood of a specific branch being taken or not taken. Static branch prediction depends on intelligently guessing the nature of the workload, for example through profiling of program execution on training data, while dynamic prediction depends on gathering of run-time information in hardware. Thus while static branch prediction adds no hardware cost to a processor, dynamic prediction requires hardware structures whose cost usually increases with the accuracy of prediction desired. Also, the static predictor can suffer in performance when the workload assumed for training does not reflect the actual workload. The accuracy of dynamic prediction can be better than that of static because the prediction can adjust to the local dynamic context, and because the workload assumed for training the static predictor may not be representative of the actual workload. On the other hand, when the working set of a workload is large, or when there is frequent switching of contexts, the accuracy of dynamic prediction can degrade considerably [Nair 1995].

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* or *not-taken* 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.

Static prediction may be supported by the provision of a *hint bit* associated with each conditional branch instruction. The hardware separates conditional branch instructions into different classes. For example, on the PowerPC, conditional branches follow the *backward-taken forward-not-taken* scheme, where a branch is expected to be taken if the branch target address is lower than the branch address, and not taken otherwise. The hint bit is available to mark instances when these class rules are likely to be violated. The compiler uses these rules along with an appropriate setting of the hint bit to arrive at an efficient layout of the executable. Other instruction set architectures have similar provisions which allow the compiler to suggest a likely violation of some assumed hardware behavior.

Note that the PowerPC hint bit only suggests whether the hardware behavior implicit in the instruction should be changed or not. It does not directly indicate whether the branch is likely to be taken, likely to be not taken, or whether the branch has unknown behavior.

The two bits necessary to represent these three cases is a luxury for most instruction sets.

High performance implementations generally incorporate dynamic branch prediction schemes and ignore the static hint bit. One can envision situations however where the static hint bit could provide useful information even in the presence of dynamic branch prediction. One such situation is during the warmup stage of sophisticated pattern and path correlation predictors.

2. A New Definition for the Hint Bit

The static hint bit, as defined in the previous section, is most useful for low-end implementations, which have neither the chip area nor the power budget to incorporate sophisticated dynamic prediction structures. However, with shrinking lithography, lower voltages, and newer power conservation techniques, it is possible that even low-end implementations will incorporate some form of dynamic branch prediction in future. It therefore becomes useful to investigate whether the hint bit in conditional branch instructions could be used to improve the performance of high-end implementations which, because of their higher instruction-level parallelism, are more critically dependent for their performance on good branch prediction.

Our new proposal for the hint bit is based on the following observations:

1. There are several branch instructions in a typical program whose behavior can be predicted by simple predictors which base their predictions only on the recent behavior of the branch itself. Included in these branches are *one-directional* branches [Chang et al, 1994], which are either taken a very large percentage of the time, or not taken a very large percentage of the time.
2. There are a number of branch instructions in common application programs that can be predicted well only in the presence of information about the path leading to the branch. Such applications benefit from the use of a correlation predictors.
3. Correlation-based predictors have large learning overhead, i.e., they could mispredict during the period when pattern behaviors are being established for a branch. Thus these predictors may not do well if the learnt patterns are lost either due to a context switch, or due to address aliasing resulting from predictor size limitations.

Proposed here is a statically-guided hybrid prediction scheme, in which the hint bit of the previous section is redefined to distinguish between two classes of conditional branches, *critical* and *non-critical*. The definition of critical branches is examined in more detail in section 3. For now, it suffices to view a critical branch as a conditional branch which could potentially be predicted with significantly greater accuracy when a sophisticated predictor, like a correlation predictor, is used instead of a simple predictor like a static predictor, a one-bit predictor, or a 2-bit predictor.

The principal idea is to have two dynamic predictors, a simple predictor to predict as many branches as possible, and a complex predictor to tackle a small number of branches which benefit from path information for better prediction. All conditional branches will be considered non-critical by default. Thus, if profiling information is not available for a program, then all branches in the program should be considered non-critical. Similarly, if the compiler is unable to determine the possible performance improvement for a branch due to a complex predictor, that branch should be predicted using a simple predictor. This prevents the possible pollution of the tables of the complex predictor by such branches. There could also be cases where the compiler intentionally treats the critical branches in a section of code as non-critical, for example when the performance of that section of the code is not critical to the overall performance of the application.

The motivation for the new definition of the hint bit is essentially the same as that for the old definition -- to improve the performance of conditional branches with simple hardware. In the new definition we are attempting to reduce the cost of dynamic branch prediction rather than eliminating it. We are also attempting to make the prediction more robust by directing most branches to a simple predictor which does not have a large learning overhead and is less sensitive to size limitations.

3. Determination of critical branches

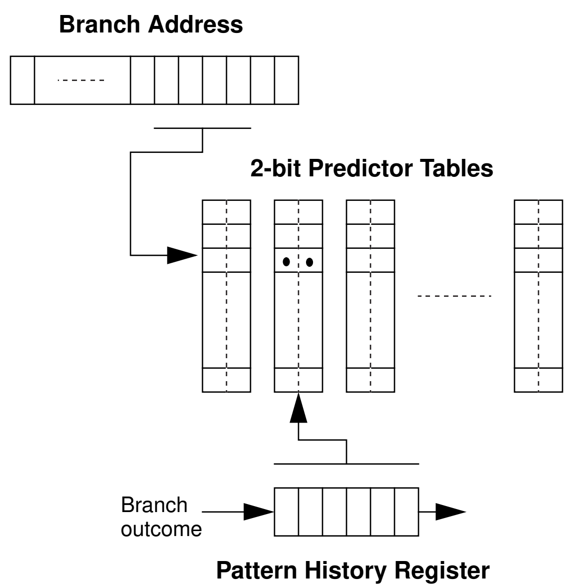

FIGURE 1 **Schematic diagram of a pattern correlation branch predictor**

In Figure 1 we show the schematics for the specific pattern correlation branch predictor [Pan et al 1992] referred to in the rest of the paper. For more details on the different types of pattern correlation predictors, the reader is referred to [Yeh and Patt 1993]. One-bit and 2-bit predictors have been discussed in [Smith 1981] and [Lee and Smith 1984].The cost of each type of predictor is summarized in Table 1. For the 1-bit and 2-bit predictors, e refers to the number of 1-bit or 2-bit entries addressed using bits from the branch address. For the pattern predictor, a p-bit history contained in the pattern history register indexes into 2^p tables each having e entries addressed through bits from the branch address.

TABLE 1 **Cost of each type of predictor**

Predictor	Cost
Static	0
1-bit	e
2-bit	$2e$
Pattern	$2e \times 2^p$

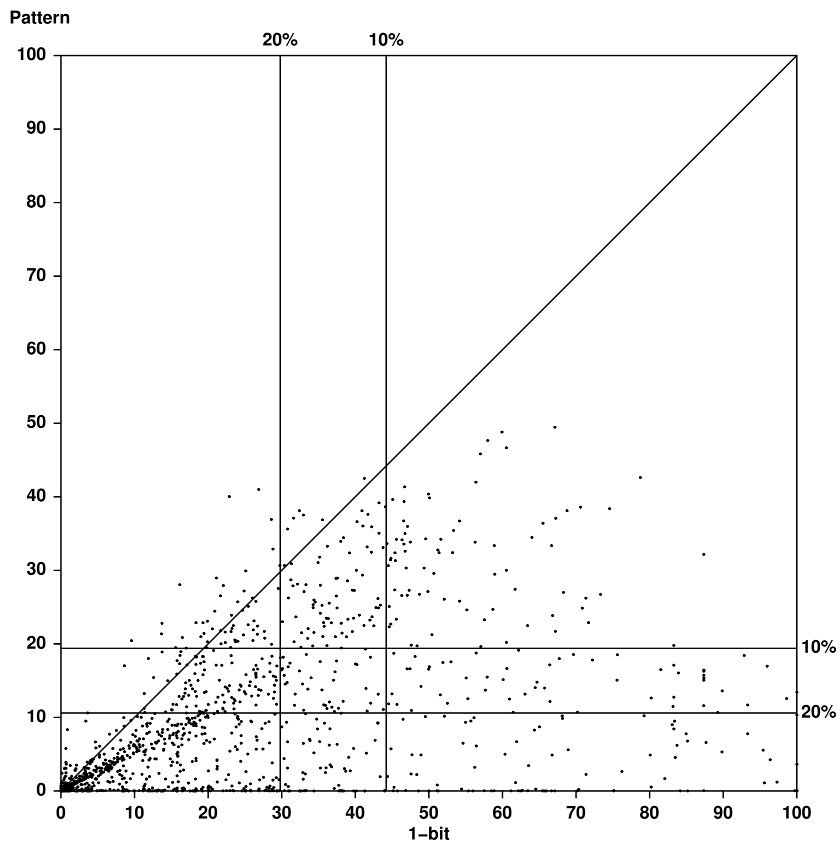

FIGURE 2 **Comparison of predictability of branches with 1-bit predictor vs. pattern predictor**

Figure 2 shows a scatter plot for all conditional branches in *gcc*, one of the programs in the SPEC95 benchmark suite. The x-axis represents the percentage predictability for each branch when a 1-bit predictor with an infinite number of entries is used. On the y-

axis is the percentage predictability when a pattern correlation predictor with an infinite number of entries and a pattern history register of length 8 is used. The infinite number of entries ensures that there is no address collision, and therefore that the behavior of each branch is independently captured. The diagonal line in the plot is the line on which all points would fall if both predictors predicted at an identical rate for all branches.

The bias of the points below the diagonal indicates that many more branches are better predicted when a pattern correlation scheme is used than when a 1-bit predictor is used. Since the trace-driven analysis of this program did not include any context switches, and since the size of the trace was about 1 billion instructions, the learning overhead is amortized over several executions of each branch, and the depicted result is expected. What was not expected was the large number of branches which do not predict significantly better with a correlation predictor, or even perform worse. The branches on the top right quadrant of the plot are instrinsically unpredictable -- they perform badly with either predictor.

Also shown on the plot are lines labelled 10% and 20%. The branches above the horizontal 10% line represent 10% of all conditional branches in the program that were the hardest to predict using a pattern predictor. Similarly the vertical 10% line demarcates 10% of hardest branches for the 1-bit predictor. The intersection of these two lines well below the diagonal is an indication of the better performance of the pattern predictor for the hard cases.

Contrast this plot with Figure 3 which plots the predictability of the 1-bit predictor against an infinite-entry 2-bit counter predictor. We see an intrinsic similarity between the 1-bit and 2-bit predictors. There is a clustering of points along a line through the origin lower than the diagonal. The 1-bit and 2-bit predictors are members of a class of predictors which use only the history of outcomes of the branch itself to drive the next prediction. The pattern predictor utilizes much more information for its prediction, but can be very bad in its prediction during its relatively long learning phase as evidenced by the greater number of points on the top left of Figure 2 compared to Figure 3. It may also be observed that the intersection of the 10% lines in Figure 3 is not as far from the diagonal as was the case for the pattern predictor.

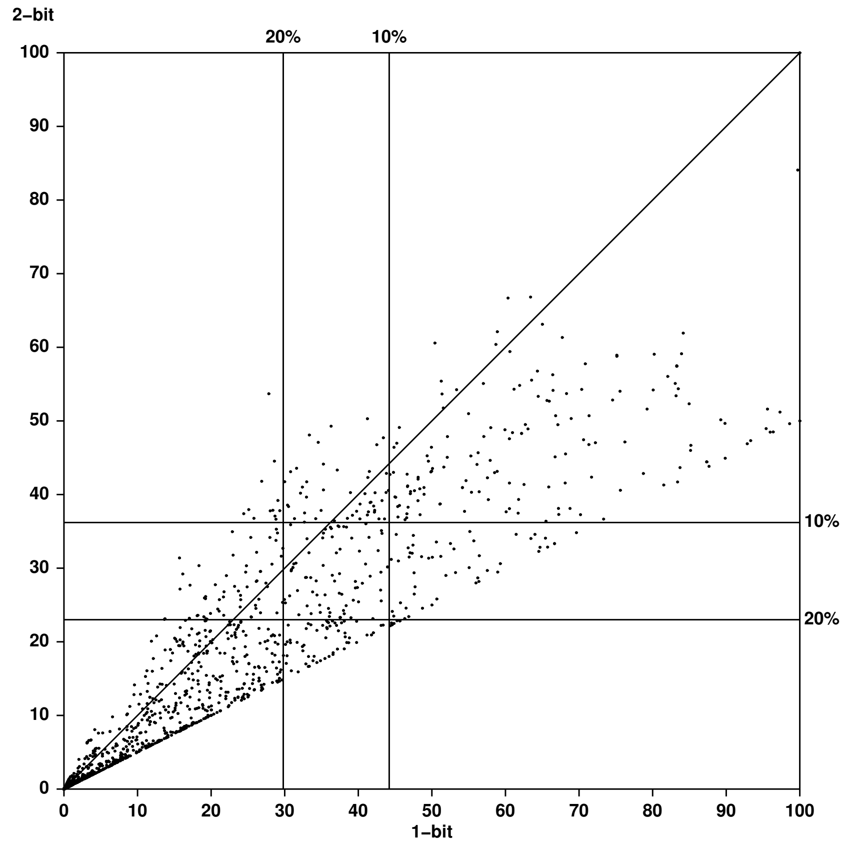

FIGURE 3 **Comparison of predictability of branches with 1-bit predictor vs. 2-bit predictor**

The above observations suggest that the criticality of a branch, i.e. the potential performance improvement of a branch by the use of a more complex predictor depends on the combination of predictors being used. Thus a branch that is critical when a 1-bit/pattern correlation hybrid scheme is used, may not be critical when a 1-bit/2-bit hybrid scheme is used. It is more likely however that the same branch is critical when technology evolves to allow another simple-complex combination, eg. a 2-bit/path-correlation hybrid scheme, to be implemented.

Rather than provide a formal definition for critical branches we propose below an algorithm to identify the most critical branches in a program.

Algorithm A:
To determine *n* critical branches in a program based on two branch predictors, *S* and *C*.

1. Choose a representative input for the program and generate an execution trace for the program on the chosen input.

2. Use the trace to drive a branch prediction simulator for predictor *S*, gathering for each conditional branch, *b*, the total number of references, r_b, and the number of misses, $m_S(b)$.
3. Generate $m_C(b)$, the misses for the other predictor *C*.
4. For each branch determine the difference $d(b) = m_C(b) - m_S(b)$. We will refer to this difference as the *miss differential* for branch *b*.
5. Order the branches by decreasing value of *d*.
6. The top *n* branches from this list forms the desired critical branch list.

Note that the ordering of the branches is done based, not on the miss rate, but on the potential improvement in performance of a branch when the complex predictor is used instead of the simple predictor. Thus branches which could have been predicted significantly more accurately by the complex predictor may be omitted because they do not occur often.

Let us define the miss differential, *d(B)*, of a set of branches, *B*, as the sum of the miss differentials of each branch in the set, i.e.

$$d(B)= \sum_{b \in B} d(b)$$

The miss differential of the critical set will depend on the size of the critical set. Its variation is shown Figure 4, for *gcc* using the training input *amptjp.i*,with the simple predictor *S* being a 1-bit predictor with an infinite number entries, and *C* being a pattern correlation predictor with infinite entries and an 8-bit global pattern history. The graph shows that the potential improvement in the miss rate increases rapidly initially, flattens out and then dips back slightly down when all branches are deemed critical. The dip back down is due to those branches which were not executed frequently enough to establish a history for the pattern correlation predictor. The interesting observation from this graph is that most of the potential benefit can be obtained by picking a moderately small percentage of the branches as candidates for the complex predictor.

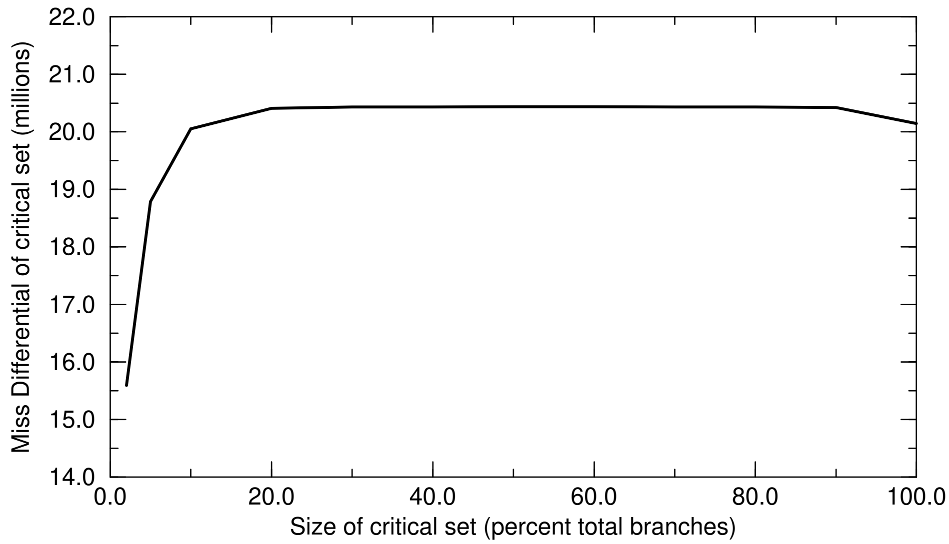

FIGURE 4 **Variation of miss differential with size of critical set (Total branches = 207 M)**

For any given number n, the list of branches determined by Algorithm A will differ with the nature of the predictors, S and C, as well as with their sizes. Ideally one would choose these based on the target implementation. However, commercial code is designed to be portable across a range of implementations of an instruction set architecture. Hence it is highly desirable that the critical list determined by the algorithm be robust across a range of implementations. A suitable training predictor would be one which produces a list of critical branches that covers a large percentage of critical branches for other hybrid combinations, especially those that are likely to be implemented. We now examine the extent to which a list produced by one choice of predictor combination overlaps with that produced for another combination.

Table 2 shows a matrix with a row r corresponding to a predictor combination $(S, C)_r$, and a column c corresponds to a predictor combination $(S, C)_c$. An entry in a given row r, and column c determines the percentage of critical branches of the predictor $(S, C)_c$ that are also contained in $(S, C)_r$, assuming the critical branch set was 10% of all conditional branches. We see from this table that a fairly large number of branches

TABLE 2 **Percentage overlap of critical set between pairs of hybrid predictors**

	1-bit/2-bit	**static/pattern**	**1-bit/pattern**	**2-bit/pattern**
1-bit/2-bit		58	82	63
static/pattern	58		74	85
1-bit/pattern	82	74		79
2-bit/pattern	63	85	79	

obtained in one critical list is also present in another critical list. Let us examine now whether the branches that overlap are indeed the useful ones.

The *coverage* of a predictor combination $(S, C)_c$ for another combination $(S, C)_r$ is defined as the ratio of the miss differential of two sets

$$\frac{d(B_c \cup B_r)}{d(B_r)}$$

where B_c and B_r refer to the critical lists of the two predictors. Table 3 shows a matrix similar to that in Table 2 where each entry refers to the percentage coverage, rather than the percentage of critical branches.

TABLE 3 **Percentage coverage of miss differential between pairs of hybrid predictors**

	1-bit/2-bit	static/pattern	1-bit/pattern	2-bit/pattern
1-bit/2-bit		69	90	76
static/pattern	80		91	98
1-bit/pattern	99	90		98
2-bit/pattern	83	92	93	

We observe that the percentages in Table 3 are consistently higher than the percentages in Table 1, indicating that even when a lower fraction of the branches in the critical list is covered, a larger fraction of the branches that can benefit from a complex predictor is getting covered. For example, even though the 1-bit/pattern combination captures only 79% of the critical list of the 2-bit/pattern combination, it is able to capture 98% of the miss differential of the latter critical list. It is interesting to note that even when the combination contained two simple predictors, a 1-bit and a 2-bit predictor, 99% of the miss differential could have been obtained from the critical list of a 1-bit/pattern-correlation predictor.

4. Experimental results

Choice of benchmark

It has been a recent trend to evaluate architectural proposals by performing trace-driven simulations of all programs in the SPEC benchmark suite, quoting some form of mean of the performance on all benchmarks. We feel that the SPEC benchmark suite as a whole is not representative of many of the real programs encountered. On the other hand, the SPEC suite is convenient to get access to, to instrument, and to experiment with, whereas commercial code, especially operating system intensive code is less convenient to instrument and gather traces for. We have observed though that one program in the SPEC suite, *gcc*, displays many of the characteristics of several pieces of commercial code. Moreover, like newer object-oriented code, *gcc* has a very large footprint, and stresses the instruction cache and branch prediction mechanisms of processors considerably. So we decided to concentrate our analysis on just this benchmark.

Previous work [Fisher and Freudenberger 1992] has pointed out that satisfactory information about the branch behavior of a program to a given set of input data can be obtained by examining its behavior on a different set of input data. The SPEC95 benchmark suite specifies an input called *amptjp.i* as training input for the *gcc* benchmark when the compiler wishes to include profile-driven optimizations to improve the performance of compiled code. We used the same input stream for our training runs for branch prediction analysis. For our final run we used two programs, *cexp.i*, which executes about 56 million instructions, and *cccp.i* which executes over a billion instructions. Early during our experiments we discovered that the qualitative nature of the results we obtained with *cexp.i* was quite close to that of *cccp.i*, and therefore we explored the design issues to a far greater extent using the faster-running *cexp.i* trace stream. The trace had 56,237,751 instructions, including 9,588,627 conditional branches**.**

Shown in Table 4 below are the characteristics of the set of profiles gathered. We have also included a nomenclature for describing various critical lists used in the experiments. Thus *op2* refers to the list of critical branches as determined by the algorithm described in the previous section, when this list is restricted to be 2% of all conditional branches in the program. Similarly the critical list in *op10* contains 10% of all conditional branches.

TABLE 4 **Characteristics of profiling runs**

Program	*gcc*
Training input	*amptjp.i*
Simple predictor	1-bit with infinite entries
Complex predictor	Pattern predictor with 8-bit global history and infinite entries
Percentage instructions marked critical	*n* for critical list op<*n*>, *n*=2,5,10,20,30,...,90,100

Variation of performance with profile

Using each of the 12 profiles described in Table 1 above, we performed a trace driven simulation of *gcc* executing *cexp.i*, where branches in the critical list were directed to a complex predictor, with all other branches being directed to the simple predictor. Figure 5 below shows the variation in performance with chosen profile for three sizes of the hybrid predictor, 12K bits, 96K bits, and infinite bits. A third of the bits in each case was devoted to the simple predictor. The complex predictor used 2-bit counters, with a global pattern history roughly half the total bits available for indexing, i.e. $2^p \cong e$. Thus the hybrid predictor in the smallest case (12K bits) had a 1-bit predictor with $e = 4096$, and a pattern predictor with $e = 64$ and $p = 6$.

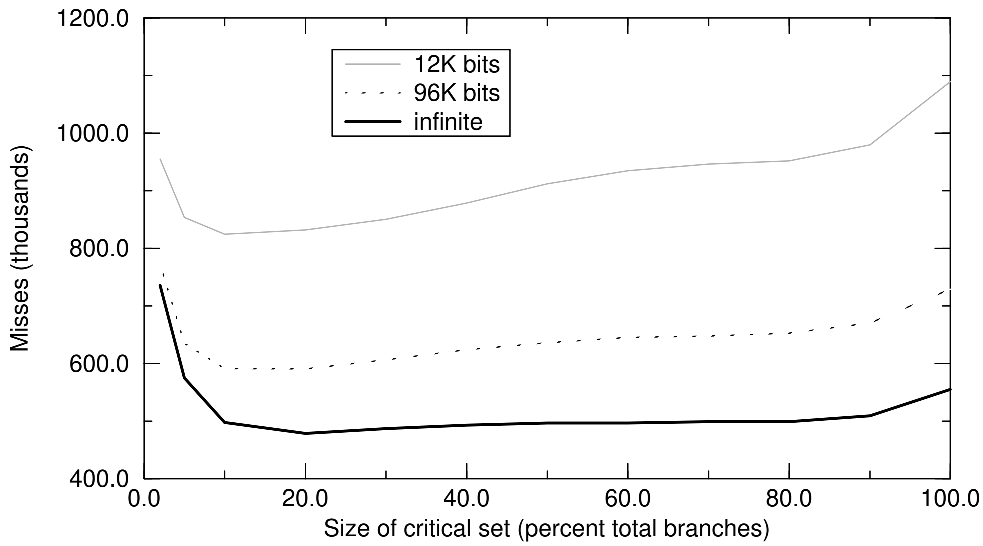

FIGURE 5 **Variation of number of misses with number of branches in critical set**

It is interesting to note that in all cases the miss rate reaches a minimum when only 10-20% of the branches is directed to the complex predictor. In fact, as the predictor size gets smaller, it becomes more advantageous to direct a smaller percentage of the branches to the complex predictor. This is because address aliasing in the complex predictor causes degradation in performance as the size of the predictor gets smaller.

Variation of performance with size

In order to observe the effect of predictor size on performance, we studied the miss rate for *cexp.i* on a number of predictors ranging in size from 3K bits to 3M bits. As before, for each size, a third of the bits was devoted to the simple 1-bit predictor. The complex predictor used 2-bit counters, with $2^p \cong e$. Figure 6 shows the variation in the number of misses as a function of the size for a simple 1-bit predictor, a pattern predictor, and a hybrid predictor with the *op10* critical list.

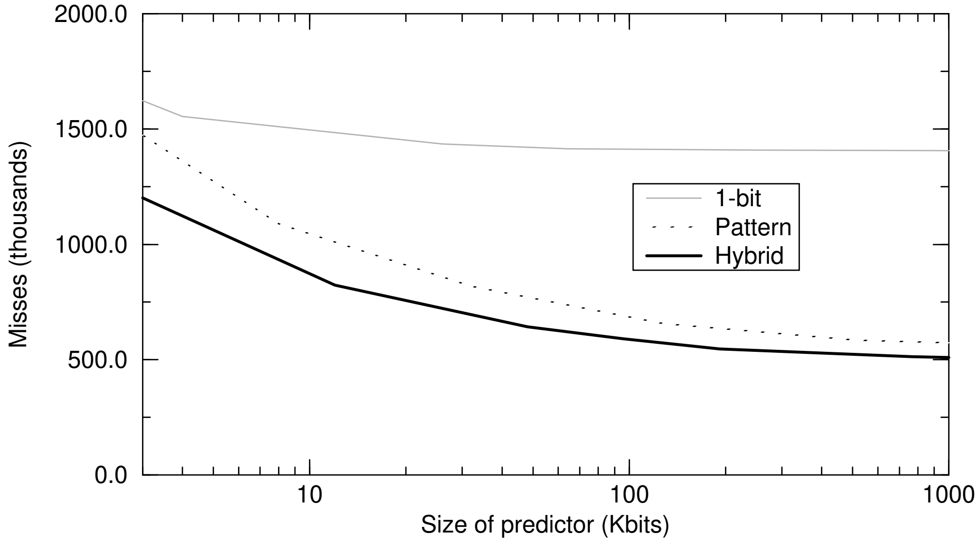

FIGURE 6 **Variation of number of misses with size of predictor**

The superiority of the hybrid predictor is evident at all sizes. In fact, for moderate to large predictor sizes, the hybrid predictor performs as well as a 4 times larger pattern predictor. Moreover, the hybrid predictor asymptotically attains a level of performance unapproachable by even an infinitely large pattern predictor. These results are clear demonstration of the value of the hint-driven hybrid prediction scheme. We emphasize again that these results are obtained when only a small fraction of the branches, 10% of all conditional branches, is predicted using a correlation predictor, with the remaining 90% of the branches being predicted through a very simple 1-bit prediction scheme.

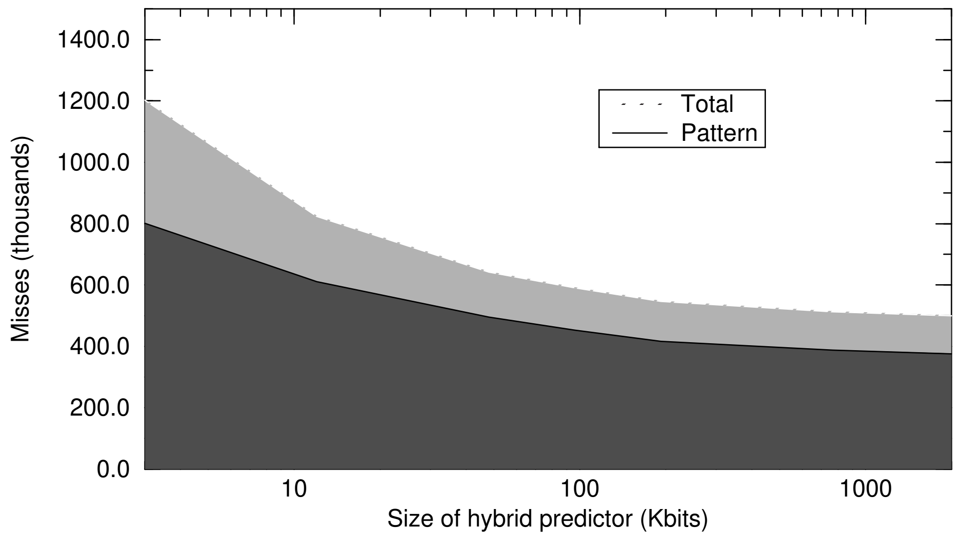

FIGURE 7 **Fraction of total misses due to each predictor in a hybrid predictor**

The graph of Figure 7 shows the breakdown of the misses. The upper part under the total miss curve shows the contribution due to the 1-bit predictor, and the lower part shows the contribution due to the pattern predictor. We can see that even though only 10% of the branches is predicted by the pattern predictor, and even though two-thirds of the total hardware is devoted to the pattern predictor, the pattern predictor still accounts for the majority of the misses. Table 5 shows the actual number of misses in each section of the hybrid predictor as a function of the section size.

TABLE 5 **Composition of misses for a hybrid 1-bit/pattern predictor**

Size (in Kbits)			Number of misses		
1-bit	pattern	total	1-bit	pattern	total
1	2	3	401,233	800,978	1,202,211
4	8	12	212,740	611,264	824,004
16	32	48	146,512	496,126	642,638
32	64	96	136,674	453,416	591,090
64	128	192	129,925	416,772	546,697
256	512	768	125,266	387,669	512,935
1024	2048	3072	124,238	375,725	499,963
∞	∞	∞	124,238	373,400	497,638

Using the table above, and with some interpolation between the values shown, we can calculate the miss rate for various combinations of sizes for the 1-bit/pattern hybrid predictor. The results are represented in the graph of Figure 8. Each curve shows the variation in the number of misses with size. The different curves correspond to different fractions (0, 1/3, 1/2, 2/3 and 1) of hardware devoted to the 1-bit predictor. All three hybrid combinations perform better than a pure pattern predictor and considerably better than a 1-bit predictor for a fixed size of predictor hardware. Given this data, our bias would be to devote less than half the hardware in an implementation to the simple predictor. (Yet well over half of the branches will be directed to this simple predictor.)

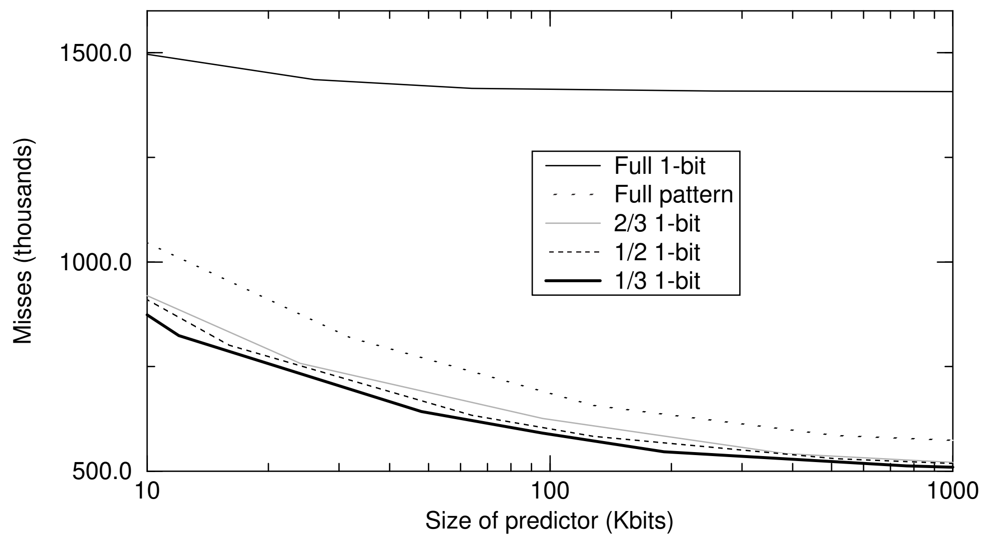

FIGURE 8 **Variation of number of misses with fraction of hardware devoted to 1-bit predictor**

Effect of flushing predictor history

We examined the sensitivity of prediction performance to frequent flushing of predictor information. For these experiments, we chose a flushing interval, *f*, and at the end of *f* instructions, reset the state of the predictors back to the initial value. This is an attempt to examine the sensitivity of a predictor in the presence of a context switch, which could overwrite the information in the predictor tables with new information for the new context. In the case of predictors which depend on a learning run before being able to predict accurately, it has been shown that there is a significant degradation in performance when the frequency of flushing is high [Nair 1995].

Figure 9 shows the variation of miss rate with flushing interval for the 1-bit predictor, the pattern predictor and the hybrid predictor. Again, the critical set composed of 10% of the branches from a training run using *amptjp.i*, with the final run being done on *cexp.i*. The sizes of predictors used for this experiment were not exactly the same -- 64 Kbits for the one-bit and pattern predictors and 48K for the hybrid predictor. We can see that the hybrid predictor, even though smaller, manages to outperform the pattern predictor for all flush rates, and also outperform the 1-bit predictor for most flush rates. Even at a very high flush rate (1 in 1000 instructions), the hybrid predictor comes much closer to the 1-bit predictor than the pattern predictor. The results demonstrate the robustness of the hybrid predictor in context switching situations.

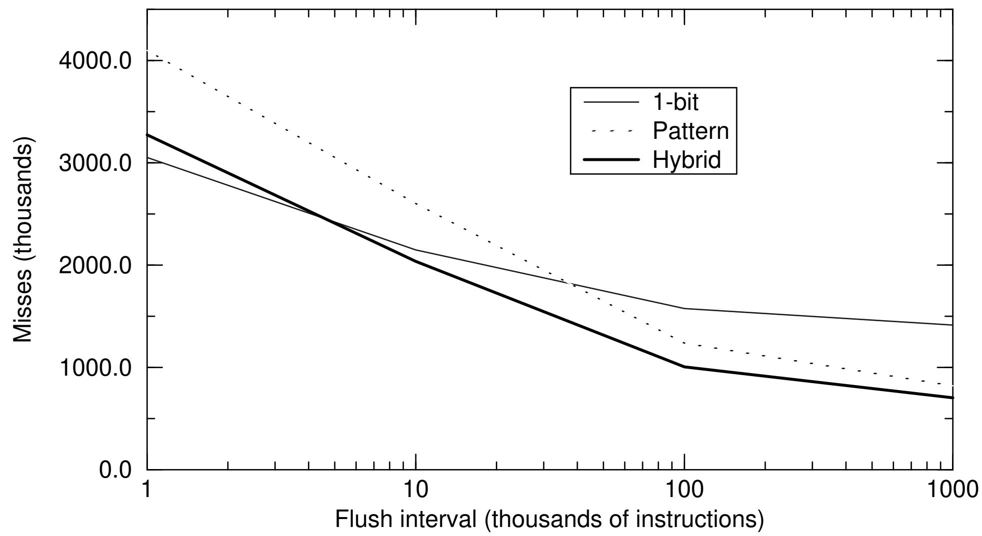

FIGURE 9 **Sensitivity of predictors to flush rate**

5. Prior Work

It was suggested in [McFarling 1993] that a combination of predictors could be useful in improving branch prediction performance. He noted that different prediction schemes have different advantages and postulated that combining them could result in more accurate prediction. He suggested a scheme which dynamically determined the appropriate predictor for each branch. This work was expanded upon in [Chang et al 1994], where it was suggested that static branch classification be used to augment such dynamic hybrid schemes. The static classification that was employed in this work was simply based on the one-directional nature of branches, i.e., the probability (as measured over the entire training run) of a specific branch going in the same direction. In a later work [Chang et al 1995] abandoned the static classification and investigated fairly sophisticated schemes to dynamically determine the correct predictor to choose among two or more implemented predictors.

The hybrid predictor guided by the hint bit as described in this paper is simpler than most of the schemes described in the above papers because the choice of predictor is guided statically. Yet, every branch is predicted using a dynamic predictor, so that the dynamic variability of each branch is adequately captured. The statically directed schemes of [Chang et al 1994], on the other hand, usually directed a somewhat small subset of the branches to a static predictor. Such static prediction needs an additional, and usually expensive, bit in the conditional branch instruction format. Moreover, it cannot capture patterns like 0000...00011111110000...000, where small runs of 1's are interspersed in large runs of 0's or vice versa -- a pattern which is easily handled by a 1-bit predictor.

There is an obvious saving in hardware in the scheme we have described compared to the dynamic hybrid schemes mentioned above, because the hardware to choose between the predictors is eliminated. There is also a not-so-obvious saving in hardware because the dynamic hybrid schemes have to be simulating the prediction of *all* branches using *both* predictors all the time. This could result in a situation where a predictor is polluted by branches which may never be chosen for prediction, resulting in a reduction in efficiency of that predictor.

The distinguishing features of our prediction scheme can therefore be summarized as follows:

1. The hint bit is used to drive two dynamic predictors, rather than a static predictor and a dynamic predictor.
2. Classes of branches are distinguished based on the differential in performance between simple and complex predictors, rather than on its performance on a single predictor.
3. Branch performance is weighted by the expected frequency of the branch. Thus the actual number of misses is more important than the miss rate.
4. Only a small fraction of the branches is directed towards the complex predictor. This is in contrast to the philosophy of preventing pollution of the complex predictor by siphoning off the "easy" branches.

6. Conclusions

We have introduced a new mechanism for providing static hints in executables to improve the performance of dynamic branch prediction. The mechanism consists of providing a single bit in conditional branch instructions, and two dynamic branch predictors in the processor. The hint bit in the conditional branch instruction determines which of the two predictors should be used to predict the branch.

We proposed that one of the dynamic predictors be a simple predictor which bases its decision only on the recent history of the branch itself and the other a complex predictor which could use information on the path leading to the branch in addition. We also suggested an algorithm to determine how the hint bit should be set.

We showed that it is better to direct a very small percentage of the branches to the complex predictor and the rest to a simple predictor. This is particularly beneficial because a smaller set of branches can utilize complex predictors better than a larger set of branches.

Our results indicate that the proposed hybrid scheme has about as many misses as a pattern correlation predictor about 4 times its size. It also performs better in situations where the predictor can be overwritten with foreign information at frequent intervals.

This scheme postulates that when enough information is not available about a branch, it is better to use simple means to predict it because of two reasons:

1. A complex predictor may not be able to do much better than a simple predictor for the branch.

2. The branch could interfere with the effective prediction of other branches which have a better chance of utilizing the more sophisticated predictor.

This is in contrast to previous schemes which argued exactly the opposite, viz. that unless the branch is known to be easy it should be predicted by a complex predictor.

The compiler may intentionally choose to mark all branches in a program, eg. a device driver, as non-critical, because the potential destruction of history belonging to the application that calls the device driver, may suppress the benefit to be gained from including the device driver in the complex predictor. The programmer could also provide information to the compiler indicating that the duration of the program is too short, or that its performance is not critical enough, to include its branches for consideration in the complex predictor. Note that we are still allowing a dynamic predictor, although a simple one, to work on the program.

As processors get more complex, the complexity is not likely to be exploited equally by all programs, or even uniformly across a single program. Dynamic exploitation of this complexity is therefore desirable, but often expensive. The hint-based dynamic prediction scheme described in this paper is an example of a technique, where the compiler, or even the programmer, using means not available to the hardware, can exploit the complexity in a more cost-effective manner. We expect hint schemes for other architectural features in future to exhibit a similar property: When a sophisticated piece of hardware is available in an implementation to improve the performance of an identifiable type of code, it is advantageous to identify such code and limit the use of the hardware structure, rather than expose it to all types of code.

7. References

[Chang et al 1992] P.-Y. Chang, E. Hao, T.-Y. Yeh, and Y. Patt, "Branch Classification: A New Mechanism for Improving Branch Predictor Performance," *Proc. 27th Annual ACM/IEEE Intl. Symp. on Microarchitecture*, Nov. 1994.

[Chang et al 1995] P.-Y. Chang, E. Hao, T.-Y. Yeh, and Y. Patt, "Alternative Implementations for Hybrid Branch Predictors," *Proc. 28th Annual ACM/IEEE Intl. Symp. on Microarchitecture*, Nov. 1995.

[Fisher and Freudenberger 1992] J. A. Fisher and S. M. Freudenberger, "Predicting Conditional Branch Directions from Previous Runs of a Program," *Proc. 5th Annual Intl. Conf. on Architectural Support for Prog. Lang. and Operating Systems*, Oct. 1992.

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

[McFarling 1993] S. McFarling, "Combining Branch Predictors," WRL Technical Note TN-36, Digital Equipment Corporation, June 1993.

[Nair 1995] "Dynamic Path-Based Branch Correlation," *Proc. 28th Annual ACM/IEEE Intl. Symp. on Microarchitecture*, Nov. 1995.

[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.

[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.

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

[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.

Copies may be requested from:

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