Combining Branch Predictors

Scott McFarling

June 1993

Western Research Laboratory 250 University Avenue Palo Alto, California 94301 USA

Abstract

One of the key factors determining computer performance is the degree to which the implementation can take advantage of instruction-level parallelism. Perhaps the most critical limit to this parallelism is the presence of conditional branches that determine which instructions need to be executed next. To increase parallelism, several authors have suggested ways of predicting the direction of conditional branches with hardware that uses the history of previous branches. The different proposed predictors take advantage of different observed patterns in branch behavior. This paper presents a method of combining the advantages of these different types of predictors. The new method uses a history mechanism to keep track of which predictor is most accurate for each branch so that the most accurate predictor can be used. In addition, this paper describes a method of increasing the usefulness of branch history by hashing it together with the branch address. Together, these new techniques are shown to outperform previously known approaches both in terms of maximum prediction accuracy and the prediction accuracy for a given predictor size. Specifically, prediction accuracy reaches 98.1% correct versus 97.1% correct for the most accurate previously known approach. Also, this new approach is typically at least a factor of two smaller than other schemes for a given prediction accuracy. Finally, this new approach allows predictors with a single level of history array access to outperform schemes with multiple levels of history for all but the largest predictor sizes.

1 Introduction

In the search for ever higher levels of performance, recent machine designs have made use of increasing degrees of instruction level parallelism. For example, both superscalar and superpipelining techniques are becoming increasingly popular. With both these techniques, branch instructions are increasingly important in determining overall machine performance. This trend is likely to continue as the use of superscalar and superpipelining increases especially if speculative execution becomes popular.

Moreover, some of the compiler assisted techniques for minimizing branch cost in early RISC designs are becoming less appropriate. In particular, delayed branches are decreasingly effective as the number of delay slots to fill increases. Also, multiple implementations of an architecture with different superscalar or superpipelining choices make the use of delay slots problematic[Sit93]. Together, these trends increase the importance of hardware methods of reducing branch cost.

The branch performance problem can be divided into two subproblems. First, a prediction of the branch direction is needed. Second, for taken branches, the instructions from the branch target must be available for execution with minimal delay. One way to provide the target instructions quickly is to use a Branch Target Buffer, which is a special instruction cache designed to store the target instructions. This paper focuses on predicting branch directions. The alternatives available for providing target instructions will not be discussed. The reader is referred to Lee and Smith[LS84] for more information.

Hardware branch prediction strategies have been studied extensively. The most well known technique, referred to here as *bimodal branch prediction*, makes a prediction based on the direction the branch went the last few times it was executed. More recent work has shown that significantly more accurate predictions can be made by utilizing more branch history. One method, considers the history of each branch independently and takes advantage of repetitive patterns. Since the histories are independent, we will refer to it as *local branch prediction*. Another technique uses the combined history of all recent branches in making a prediction. This technique will be referred to as *global* branch prediction. Each of these different branch prediction strategies have distinct advantages. The bimodal technique works well when each branch is strongly biased in a particular direction. The local technique works well for branches with simple repetitive patterns. The global technique works particularly well when the direction taken by sequentially executed branches is highly correlated.

This paper introduces a new technique that allows the distinct advantages of different branch predictors to be combined. The technique uses multiple branch predictors and selects the one which is performing best for each branch. This approach is shown to provide more accurate predictions than any one predictor alone. This paper also shows a method of increasing the utility of branch history by hashing it together with the branch address.

The organization of this paper is as follows. First, Section 2 discusses previous work in branch prediction. Later sections describe in detail the prediction methods found useful in combination and will evaluate them quantitatively to provide a basis for evaluating the new techniques. Sections 3, 4, and 5 review the bimodal, local, and global predictors. Section 6 discusses predictors indexed by both global history and branch address information. Section 7 discusses hashing global history and branch address information before indexing the predictor. Section 8 describes the technique for combining multiple predictors. Section 9 gives some concluding remarks. Section 10 gives some suggestions for future work. Finally, Appendix A presents some additional comparisons to variations of the local prediction method.

2 Related Work

Branch performance issues have been studied extensively. J. E. Smith[Smi81] presented several hardware schemes for predicting branch directions including the bimodal scheme that will be described in Section 3. Lee and A. J. Smith[LS84] evaluated several branch prediction schemes. In addition, they showed how branch target buffers can be used to reduce the pipeline delays normally encountered when branches are taken. McFarling and Hennessy[MH86] compared various hardware and software approaches to reducing branch cost including using profile information. Hwu, Conte, and Chang[HCC89] performed a similar study for a wider range of pipeline lengths. Fisher and Freudenberger[FF92] studied the stability of profile information across separate runs of a program. Both the local and global branch prediction schemes were described by Yeh and Patt[YP92, YP93]. Pan, So, and Rahmeh[PSR92] described how both global history and branch address information can be used in one predictor. Ball and Larus[BL93] describe several techniques for guessing the most common branches directions at compile time using static information. Several studies[Wal91, JW89, LW93] have looked at the implications of branches on available instruction level parallelism. These studies show that branch prediction errors are a critical factor determining the amount of local parallelism that can be exploited.

3 Bimodal Branch Prediction

The behavior of typical branches is far from random. Most branches are either usually taken or usually not taken. Bimodal branch prediction takes advantage of this bimodal distribution of branch behavior and attempts to distinguish usually taken from usually not-taken branches. There are a number of ways this can be done. Perhaps the simplest approach is shown in Figure 1. The figure shows a table of counters indexed by the low order address bits in the program counter. Each counter is two bits long. For each taken branch, the appropriate counter is incremented. Likewise for each not-taken branch, the appropriate counter is decremented. In addition, the counter is saturating. In other words, the counter is not decremented past zero, nor is it incremented past three. The most significant bit determines the prediction. Repeatedly taken branches will be predicted to be taken, and

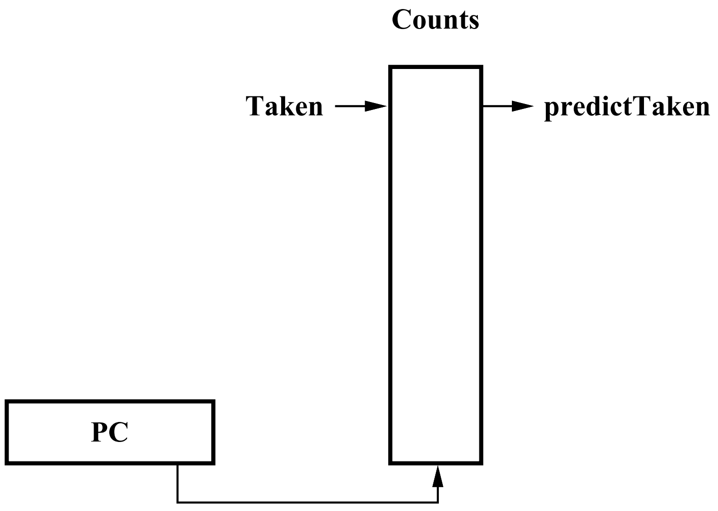

Figure 1: Bimodal Predictor Structure

repeatedly not-taken branches will be predicted to be not-taken. By using a 2-bit counter, the predictor can tolerate a branch going an unusual direction one time and keep predicting the usual branch direction.

For large counter tables, each branch will map to a unique counter. For smaller tables, multiple branches may share the same counter, resulting in degraded prediction accuracy. One alternate implementation is to store a tag with each counter and use a set-associative lookup to match counters with branches. For a fixed number of counters, a set-associative table has better performance. However, once the size of tags is accounted for, a simple array of counters often has better performance for a given predictor size. This would not be the case if the tags were already needed to support a branch target buffer.

To compare various branch prediction strategies, we will use the SPEC'89 benchmarks [SPE90] shown in Figure 2. These benchmarks include a mix of symbolic and numeric applications. However, to limit simulation time, only the first 10 million instructions from each benchmark was simulated. Execution traces were obtained on a DECstation 5000 using the pixie tracing facility[Kil86, Smi91]. Finally, all counters are initially set as if all previous branches were taken.

Figure 3 shows the average conditional branch prediction accuracy of bimodal prediction. The number plotted is the average accuracy across the SPEC'89 benchmarks with each benchmark simulated for 10 million instructions. The accuracy increases with predictor size since fewer branches share counters as the number of counters increases. However, prediction accuracy saturates at 93.5% correct once each branch maps to a unique counter. A set-associative predictor would saturate at the same accuracy.

benchmark	description
doduc	Monte Carlo simulation
eqntott	conversion from equation to truth table
espress	minimization of boolean functions
fpppp	quantum chemistry calculations
gcc	GNU C compiler
li	lisp interpreter
mat300	matrix multiplication
nasa7	NASA Ames FORTRAN Kernels
spice	circuit simulation
tomcatv	vectorized mesh generation

Figure 2: SPEC Benchmarks Used for Evaluation

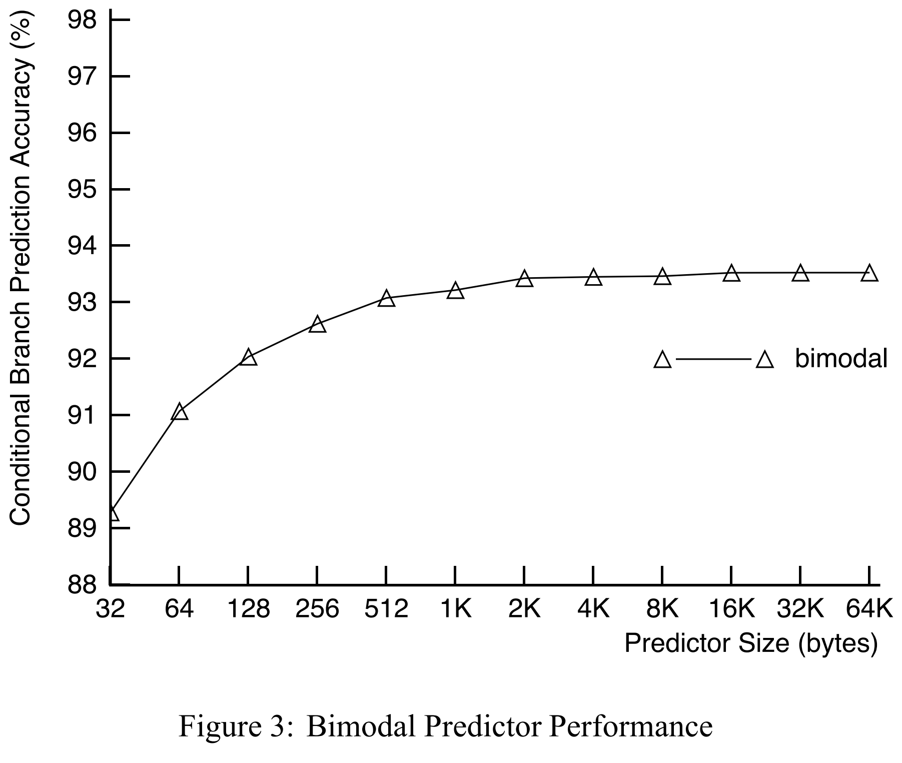

Figure 3: Bimodal Predictor Performance

4 Local Branch Prediction

One way to improve on bimodal prediction is to recognize that many branches execute repetitive patterns. Perhaps the most common example of this behavior is loop control branches. Consider the following example:

for (i=1; i<=4; i++) { }

If the loop test is done at the end of the body, the corresponding branch will execute the pattern $(1110)^n$, where 1 and 0 represent taken and not taken respectively, and n is the number of times the loop is executed. Clearly, if we knew the direction this branch had gone on the previous three executions, then we could always be able to predict the next branch direction.

A branch prediction method close to one developed by Yeh and Patt[YP92] that can take advantage of this type of pattern is shown in Figure 4. The figure shows a branch predictor that uses two tables. The first table records the history of recent branches. Many different organizations are possible for this table. In this paper, we will assume that it is simply an array indexed by the low-order bits of the branch address. Yeh and Patt assumed a set-associative branch history table. As with bimodal prediction, a simple array avoids the need to store tags but does suffer from degraded performance when multiple branches map to the same table entry, especially with smaller table sizes.

Each history table entry records the direction taken by the most recent n branches whose addresses map to this entry, where n is the length of the entry in bits. The second table is an array of 2-bit counters identical to those used for bimodal branch prediction. However, here they are indexed by the branch history stored in the first table. In this paper, this approach is referred to as *local* branch prediction because the history used is local to the current branch. In Yeh and Patt's nomenclature this method is referred to as a *per-address* scheme.

Consider again the simple loop example above. Let's assume that this is the only branch in the program. In this case, there will be a history table entry that stores the history of this branch only and the counter table will reflect solely the behavior of this branch. With 3 bits of history and 2^3 counters, the local branch predictor will be able determine the current iteration and always make the correct prediction after some initial settling of the counter values. If there are more branches in the program, a local predictor can suffer from two kinds of contention. First, the branch history may reflect a mix of histories of all the branches that map to each history entry. Second, since there is only one counter array for all branches, there may be conflict between patterns. For example, if there is another branch that typically executes the pattern $(0110)^n$ instead of $(1110)^n$, there will be contention when the branch history is (110). However, with 4 bits of history and 2^4 counters, this contention can be avoided. Note however, that if the first pattern is executed a large number of times followed by a large number of executions of the second pattern, then only 3 bits of history are needed since the counters dynamically adjust to the more recent patterns.

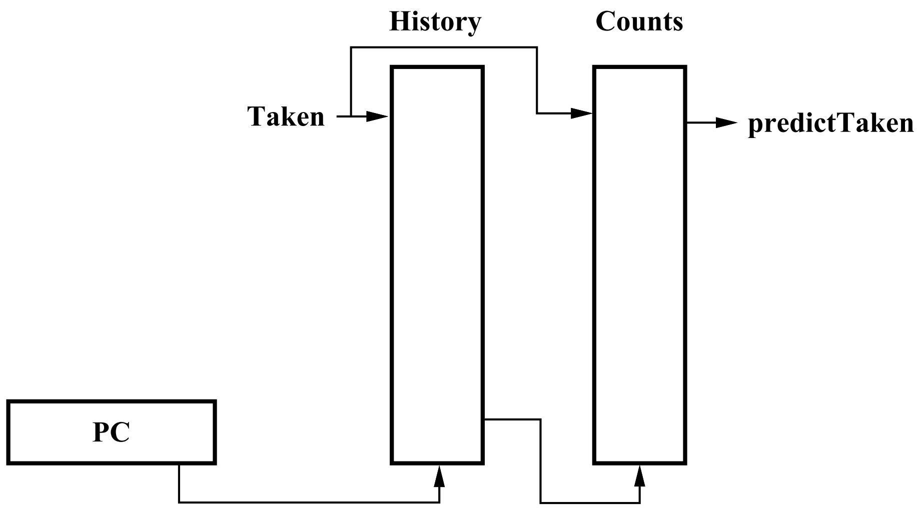

Figure 4: Local History Predictor Structure

Figure 5 shows the performance of local branch prediction as a function of the predictor size. For simplicity, we assume that the number of history and count array entries are the same. See Appendix A for a discussion of some alternatives. For very small predictors, the local scheme is actually worse than the bimodal scheme. If there is excessive contention for history entries, then storing this history is of no value. However, above about 128 bytes, the local predictor has significantly better performance. For large predictors, the accuracy approaches 97.1% correct, with less than half as many misspredictions as the bimodal scheme.

5 Global Branch Prediction

In the local branch prediction scheme, the only patterns considered are those of the current branch. Another scheme proposed by Yeh and Patt[YP92] is able to take advantage of other recent branches to make a prediction. One implementation of such an approach is shown in Figure 6. A single shift register GR, records the direction taken by the most recent n conditional branches. Since the branch history is global to all branches, this strategy is called global branch prediction in this paper.

Global branch prediction is able to take advantage of two types of patterns. First, the direction take by the current branch may depend strongly on other recent branches. Consider the example below:

```
if (x<1) . . .
if (x>1) . . .
```

Using global history prediction, we are able to base the prediction for the second if on the direction taken by the first if. If (x<1), we know that the second if will not be

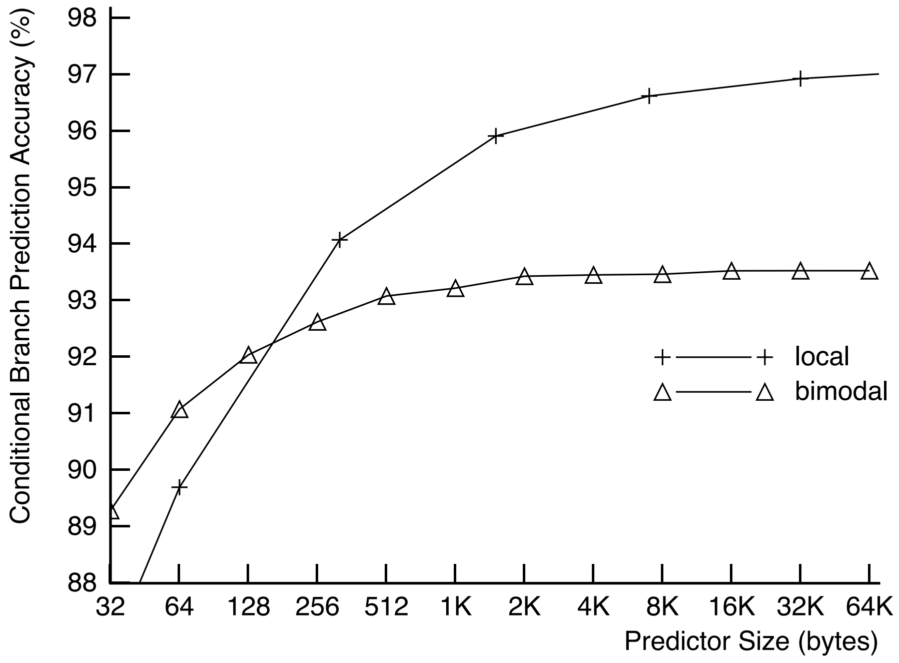

Figure 5: Local History Predictor Performance

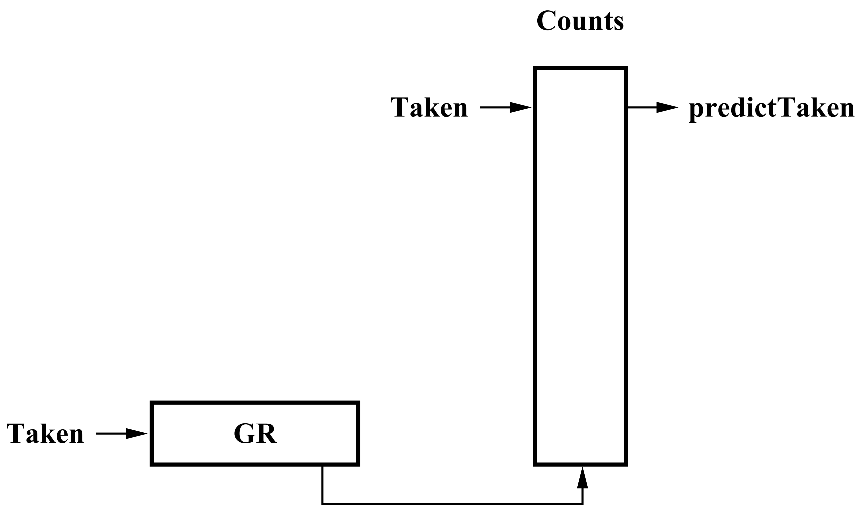

Figure 6: Global History Predictor Structure

taken. If ($x \geq 1$) then we don't know conclusively which way this branch will be taken, but the probability may well be skewed one direction or the other. If so, we should be able to make a better prediction than if we had no information about the value of x. Pan, So, and Rahmeh[PSR92] showed several examples of neighboring branches in the SPEC benchmarks with conditions that are correlated in this way.

A second way that global branch prediction can be effective is by duplicating the behavior of local branch prediction. This can occur when the global history includes all the local history needed to make an accurate prediction. Consider the example:

```
for (i=0; i<100; i++)
    for (j=0; j<3; j++)
```

After the initial startup time, the conditional branches have the following behavior, assuming GR is shifted to the left:

test	value	GR	result
j<3	j=1	1101	taken
j<3	j=2	1011	taken
j<3	j=3	0111	not taken
i<100		1110	usually taken

Here the global history is able to both distinguish which of the two branches is being executed and what the current value of j is. Thus, the prediction accuracy here would be as good as that of local prediction.

Figure 7 compares the performance of the global prediction with local and bimodal branch prediction. The plot shows that the global scheme is significantly less effective than the local scheme for a fixed size predictor. It is only better than the bimodal scheme above 1KB.

We can understand this behavior intuitively by looking at the information content of the counter table index. For small predictors, the bimodal scheme is relatively good. Here, the branch address bits used in the bimodal scheme efficiently distinguish different branches. As the number of counters doubles, roughly half as many branches will share the same counter. Informally, we can say that the information content of the address bits is high. For large counter tables, this is no longer true. As more counters are added, eventually each frequent branch will map to a unique counter. Thus, the information content in each additional address bit declines to zero for increasingly large counter tables.

The information content of the global history register begins relatively small, but it continues to grow for larger sizes. To understand why, consider the history one might expect when a particular branch is executed. Since over 90% of the time each branch goes the same direction, the sequence of previous branches and the directions taken by these branches will tend to be highly repetitive for any one branch, but perhaps very different for other branches. This behavior allows a global predictor to identify different branches. However as Figure 7 suggests, that the global history is less efficient at this than the branch

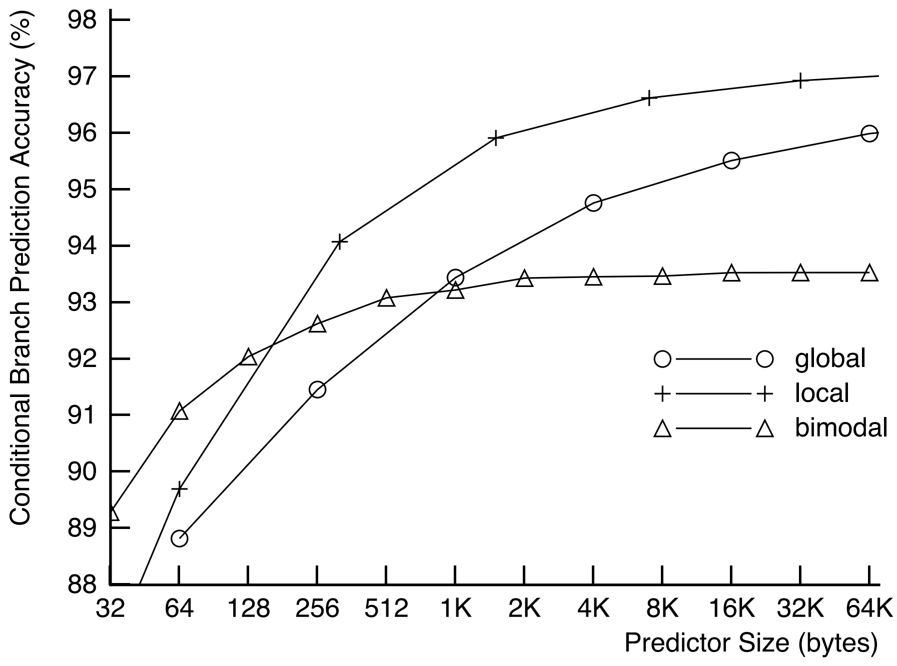

Figure 7: Global History Predictor Performance

address. On the other hand, the global history register can capture more information than just identifying which branch is current, and thus for sufficiently large predictors it does better than bimodal prediction.

6 Global Predictor with Index Selection

As discussed in the previous section, global history information is less efficient at identifying the current branch than simply using the branch address. This suggests that a more efficient prediction might be made using both the branch address and the global history. Such a scheme was proposed by Pan, So, and Rahmeh[PSR92]. Their approach is shown in Figure 8. Here the counter table is indexed with a concatenation of global history and branch address bits.

The performance of global prediction with selected address bits (gselect) is shown in Figure 9. With the bit selection approach, there is a tradeoff between using more history bits or more address bits. For a predictor table with 2^K counters, we could use anywhere from 1 to (K-1) address bits. Rather than show all these possibilities, Figure 9 only shows the performance of the predictor of the given size with with the best accuracy across the benchmarks (gselect-best).

As we would expect, gselect-best performs better than either bimodal or global prediction since both are essentially degenerate cases. For small sizes, gselect-best parallels the performance of bimodal prediction. However, once there are enough address bits to identify most branches, more global history bits are used, resulting in significantly better

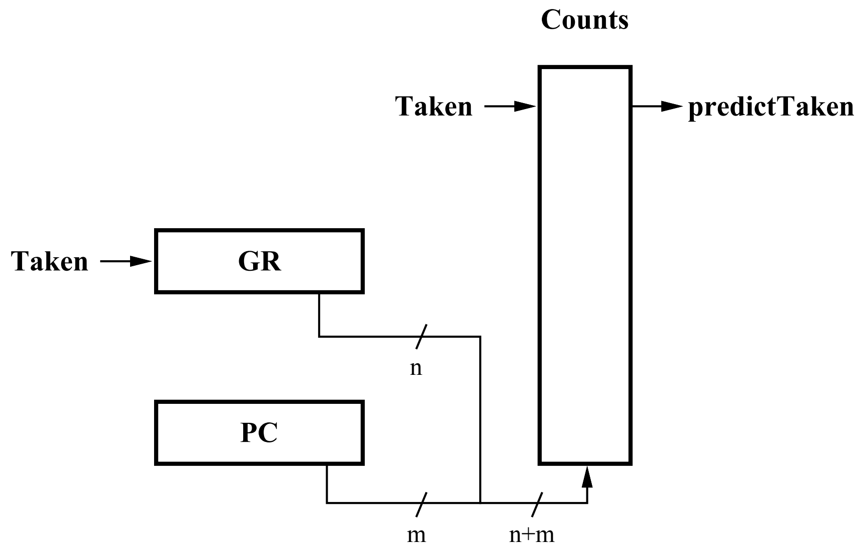

Figure 8: Global History Predictor with Index Selection

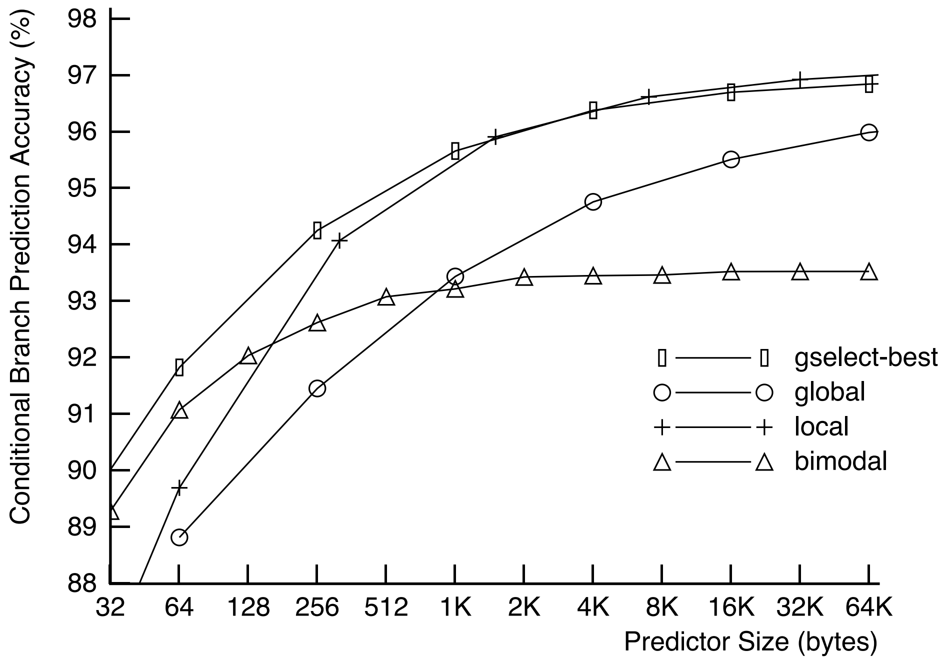

Figure 9: Global History with Index Selection Performance

prediction results than the bimodal scheme. The gselect-best method also significantly outperforms simple global prediction for most predictor sizes because the branch address bits more efficiently identify the branch.

For predictor sizes less than 1KB, gselect-best also outperforms local prediction. The global schemes have the advantage that the storage space required for global history is negligible. Moreover, even for larger predictors, the accuracies are close. This is especially interesting since gselect requires only a single array access whereas local prediction requires two array accesses in sequence. This suggests that a gselect predictor should have less delay and be easier to pipeline than a local predictor.

7 Global History with Index Sharing

In the discussion of global prediction, we described how global history information weakly identifies the current branch. This suggests that there is a lot of redundancy in the counter index used by gselect. If there are enough address bits to identify the branch, we can expect the frequent global history combinations to be rather sparse. We can take advantage of this effect by hashing the branch address and the global history together. In particular, we can expect the exclusive OR of the branch address with the global history to have more information than either component alone. Moreover, since more address bits and global history bits are in use, there is reason to expect better predictions than gselect. Consider the following simple example where there are only two branches and each branch has only two common global histories:

Branch Address	Global History	gselect 4/4	gshare 8/8
00000000	00000001	00000001	00000001
00000000	00000000	00000000	00000000
11111111	00000000	11110000	11111111
11111111	10000000	11110000	01111111

Strategy gselect 4/4 concatenates the low order 4 bits of both the branch address and the global history. We will call the strategy of exclusive ORing branch address and global history *gshare*. Strategy gshare 8/8 uses the bit-wise exclusive OR of all 8 bits of both the branch address and the global history. Comparing gshare 8/8 and gselect 4/4 shows that only gshare is able to separate all four cases. The gselect predictor can't take advantage of the distinguishing history in the upper four bits.

As with gselect, we can choose to use fewer global history bits than branch address bits. In this case, the global history bits are exclusive ORed with the higher order address bits. Typically, the higher order address bits will be more sparse than the lower order bits.

Figure 10 shows the gshare predictor structure. Figure 11 compares the performance of gshare with gselect. Figure 11 only shows the gshare predictor among the various history length choices that has the best performance across the benchmarks (gshare-best).

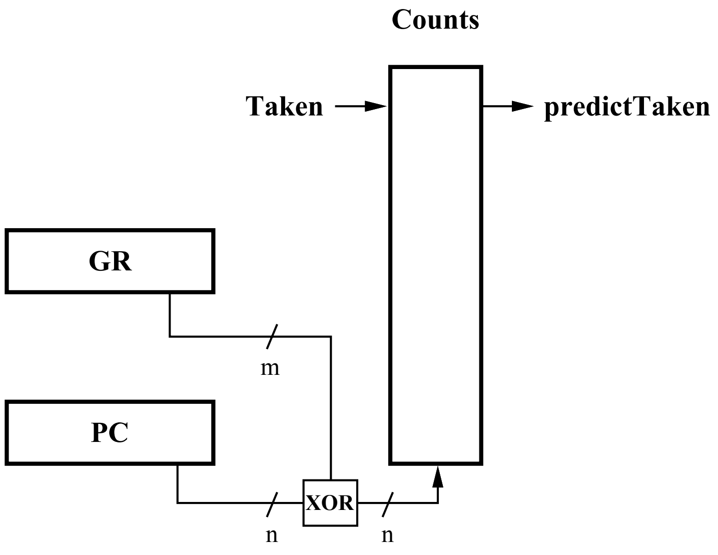

Figure 10: Global History Predictor with Index Sharing

For predictor sizes of 256 bytes and over, gshare-best outperforms gselect-best by a small margin. For smaller predictors, gshare underperforms gselect because there is already too much contention for counters between different branches and adding global information just makes it worse.

8 Combining Branch Predictors

The different branch prediction schemes we have presented have different advantages. A natural question is whether the different advantages can be combined in a new branch prediction method with better prediction accuracy. One such method is shown in Figure 12. This combined predictor contains two predictors P1 and P2 that could be one of the predictors discussed in the previous sections or indeed any kind of branch prediction method. In addition, the combined predictor contains an additional counter array which serves to select the best predictor to use. As before, we will use 2-bit up/down saturating counters. Each counter keeps track of which predictor is more accurate for the branches that share that counter. Specifically, using the notation P1c and P2c to denote whether predictors P1 and P2 are correct respectively, the counter is incremented or decremented by P1c-P2c as shown below:

P1c	P2c	P1c-P2c	
0	0	0	(no change)
0	1	-1	(decrement counter)
1	0	1	(increment counter)
1	1	0	(no change)

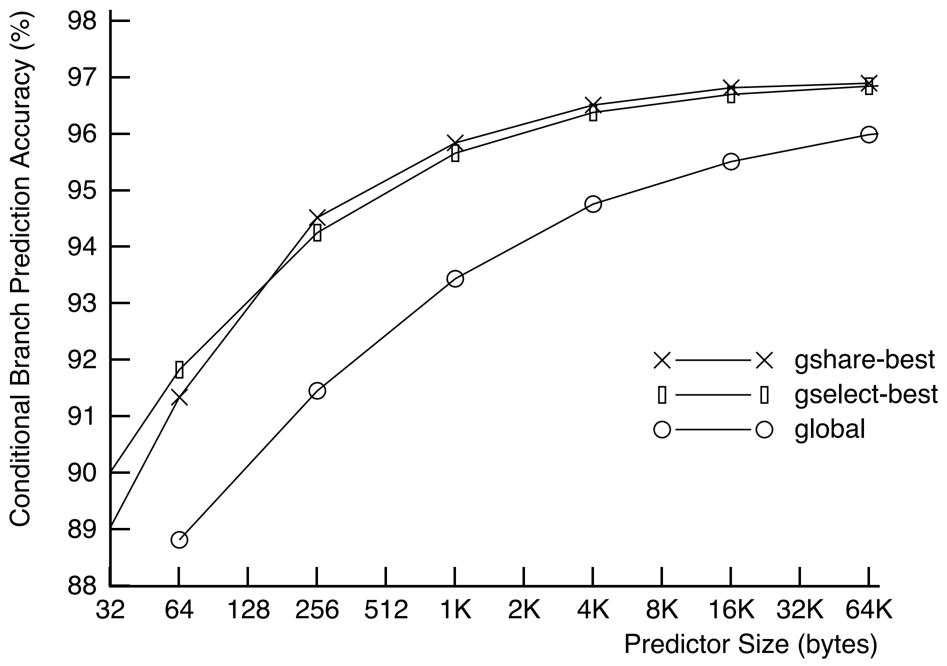

Figure 11: Global History with Index Sharing Performance

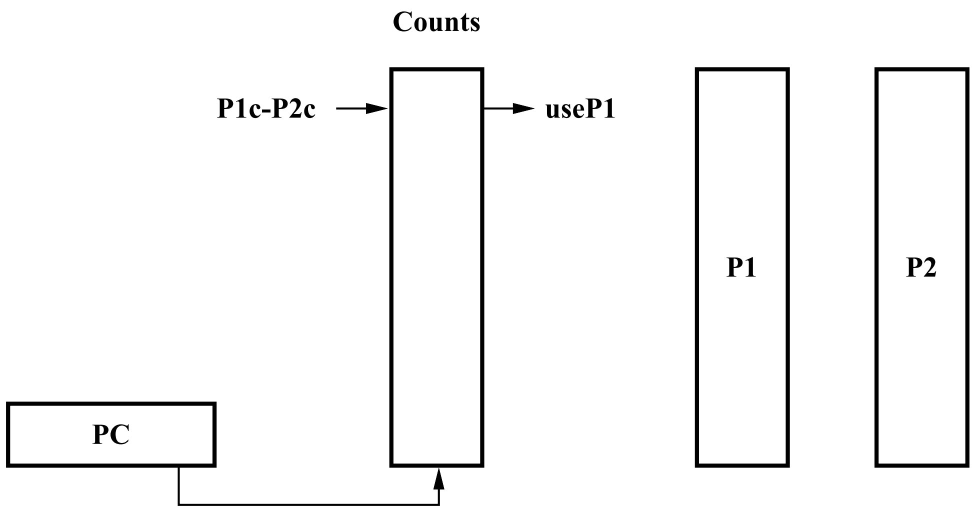

Figure 12: Combined Predictor Structure

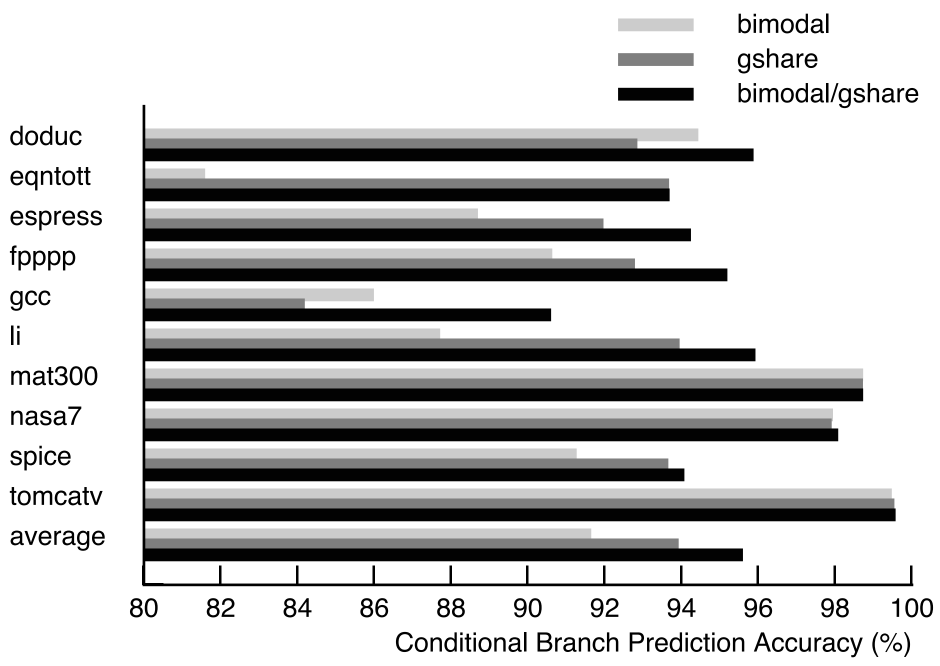

Figure 13: Combined Predictor Performance by Benchmark

One combination of branch predictors that is useful is bimodal/gshare. In this combination, global information can be used if it is worthwhile, otherwise the usual branch direction as predicted by the bimodal scheme can be used. Here, we assume gshare uses the same number of history and address bits. This assumption maximizes the amount of global information. Diluting the branch address information is less of a concern because the bimodal prediction can always be used. Similarly, gshare performs significantly better here than gselect since it uses more global information. Figure 13 shows how the bimodal/gshare combination works on the SPEC'89 benchmarks. Here, all the benchmarks were run to completion. Also, each predictor array has 1K counters. Thus, the combined predictor is actually 3 times as large. As the graph shows, the combined predictor always does better than either predictor alone. For example, with eqntott, gshare is much more effective than bimodal and bimodal/gshare matches the performance of gshare. Figure 14 shows how often each predictor was used in the bimodal/gshare combined predictor on these same runs. For these sizes, the bimodal predictor is typically used more often. However, for eqntott, the gshare predictor is more often used. Again the choice of predictors is made branch by branch. In any one benchmark, many branches may use the bimodal prediction while other branches use gshare. Figure 15 shows how using bimodal/gshare effects the number of instructions between misspredicted branches. The combination increases this measure significantly for some of the benchmarks, especially some of the less predictable benchmarks like gcc.

Figure 16 shows the combined predictor accuracy for a range of predictor sizes. As earlier, only the average accuracy across the SPEC'89 benchmarks run for 10M instructions is shown. In this chart, we choose to display a bimodal/gshare predictor where the

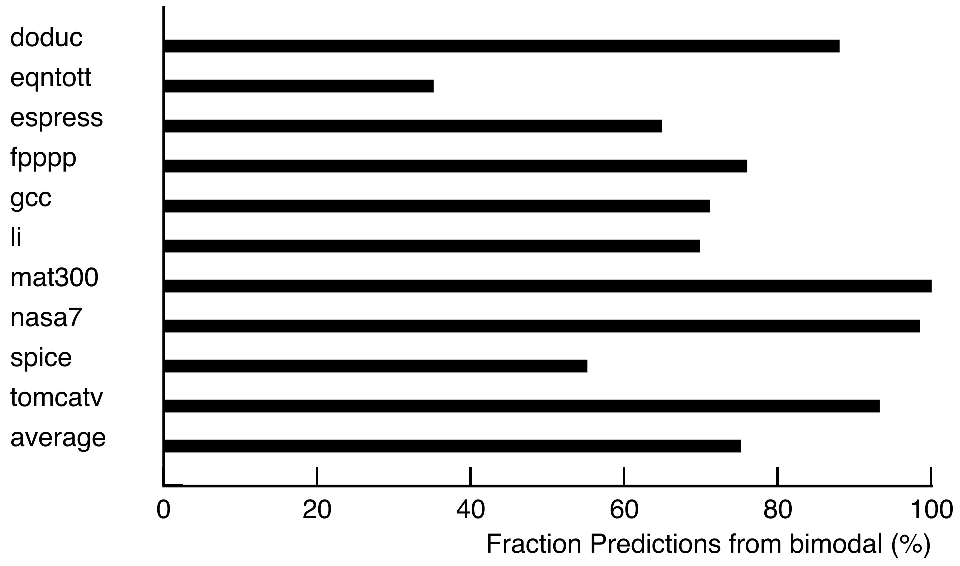

Figure 14: bimodal/gshare Predictor Performance by Benchmark

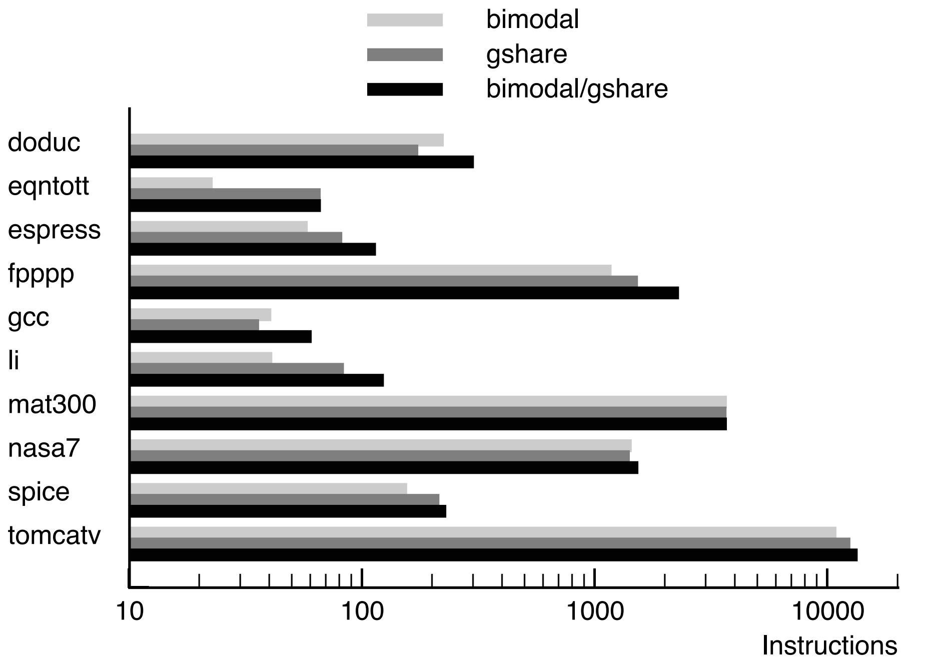

Figure 15: Instructions between Misspredicted Branches

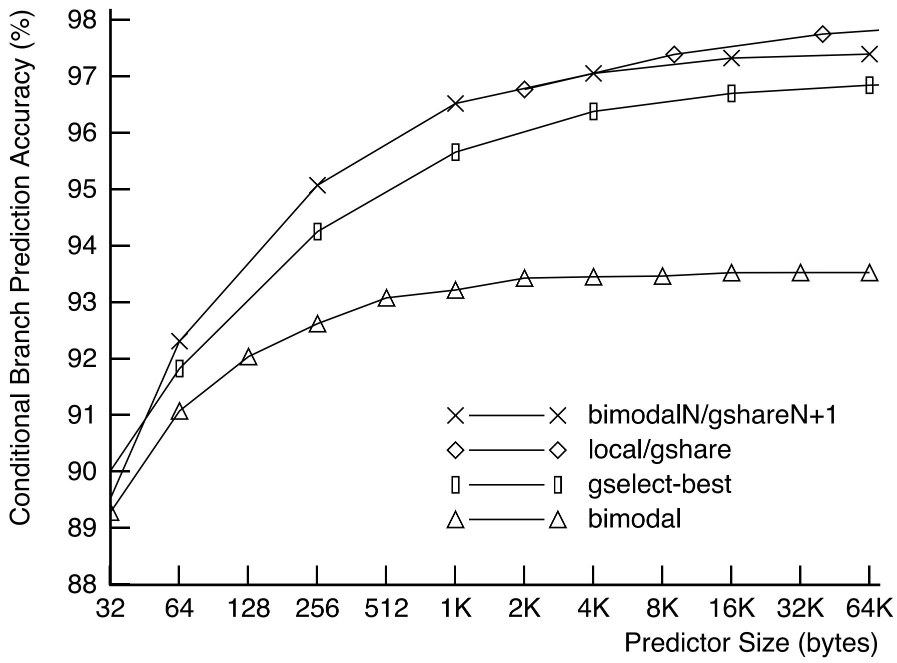

Figure 16: Combined Predictor Performance by Size

gshare array contains twice as many counters (bimodalN/gshareN+1). This allows a more direct comparison to gselect-best since the total predictor size is an integral number of bytes. Predictor bimodalN/gshareN+1 also has slightly better performance since the predictor selection array cost is amortized over more predictor counters. The performance of bimodalN/gshareN+1 is significantly better than gselect-best. The 1KB combined predictor has nearly the same performance as a 16KB gselect-best predictor.

Figure 16 also shows the performance of a combined local/gshare predictor where it outperforms bimodal/gshare. For this plot, all the local/gshare arrays have the same number of entries. For sizes of 2KB and larger, the local/gshare predictor has better accuracy than bimodalN/gshareN+1. For large arrays this accuracy approaches 98.1% correct. This result is as we would expect since large local predictors subsume the information available to a bimodal predictor.

9 Conclusions

In this paper, we have presented two new methods for improving branch prediction performance. First, we showed that using the bit-wise exclusive OR of the global branch history and the branch address to access predictor counters results in better performance for a given counter array size. We also showed that the advantages of multiple branch predictors can be combined by providing multiple predictors and by keeping track of which predictor is more accurate for the current branch. These methods permit construction of predictors that are more accurate for a given size than previously known methods. Also, combined predictors using local and global branch information reach a prediction accuracy of 98.1% as compared to 97.1% for the previously most accurate known scheme. The approaches presented here should be increasingly useful as machine designers attempt to take advantage of instruction level parallelism and miss-predicted branches become a critical performance bottleneck.

10 Suggestions for Future Work

There are a number of ways this study could be extended to possibly find more accurate and less costly branch predictors. First, there are a large number of parameters such as sizes, associativities, and pipeline costs that were not fully explored here. Careful exploration of this space might yield better predictors. Second, other sources of information such as whether the branch target is forward or backward might be usefully added to increase accuracy. Third, the typically sparse branch history might be compressed to reduce the number of counters needed. Finally, a compiler with profile support might be able to reduce or eliminate the need for branch predictors as described here. For example, previous work has shown that using profile information to set a likely-taken bit in the branch results in accuracy close to that of the bimodal scheme. Thus, for code optimized in this way, the bimodal predictor in the bimodal/gshare scheme might be unnecessary. More elaborate optimization might also eliminate the need for the gshare predictor as well. This might be done with either more careful inspection of branch conditions or more elaborate profiling of typical branch patterns. For example, branches with correlated conditions might be detected with semantic analysis or by more elaborate profiling that could detect branch correlation dynamically. This information might then be used to duplicate or restructure the branches so that a simpler branch prediction method could take advantage of the correlation. Furthermore, branch patterns caused by loops might be exploited by careful unrolling that takes advantage of the typical iteration count detected either semantically or with more elaborate profiling.

A Appendix

The local branch prediction scheme has a number of variations that were not discussed in the Section 4. In this appendix, we will discuss two variations and show that the combined predictor has better performance than these alternatives. First, similarly to the gselect scheme, it is possible to index the counter array with both the branch address and the local history. Again, there are a large number of possibilities. Figure 17 shows the family of performance curves where the number of history bits used to index the counter array is held constant. For example, local-2h implies that there are 2 history bits used to index the counter array and the remaining index bits come from the branch address. We keep the assumption that the number of history array and counter array entries are the same. As

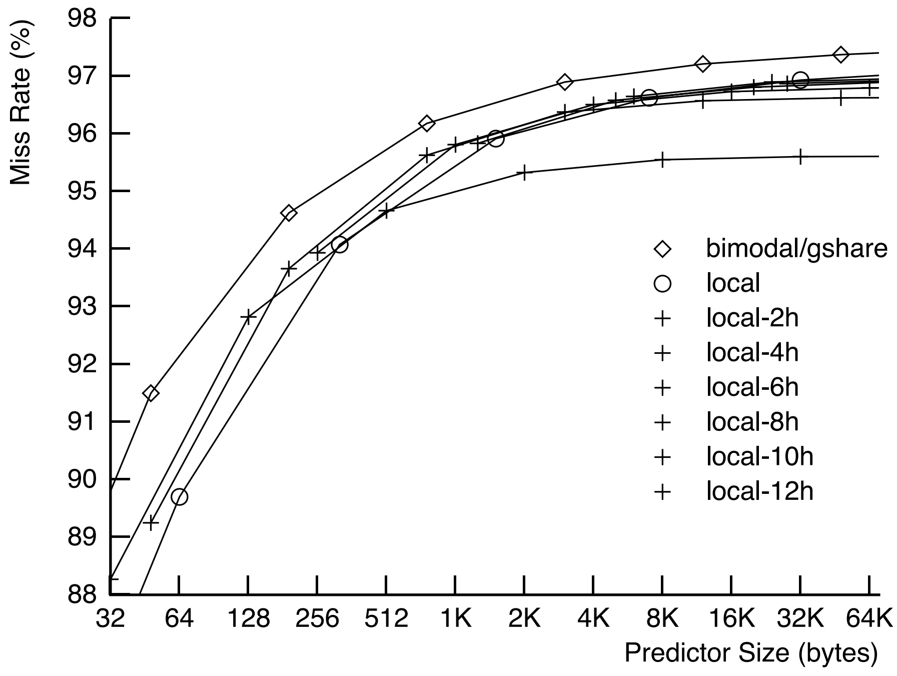

Figure 17: Local Predictor Performance with Address Bits

the figure shows, reducing the number of history bits can improve performance. This is mainly due to the reduction in the history array size itself. The figure also shows that the bimodal/gshare predictor performance is still significantly better than the different local predictor variations. In addition, the bimodal/gshare predictor only requires a single level of array access.

Another variation in the local scheme is to change the number of history entries. Figure 18 shows the resulting performance. The notation local-64HR signifies that there are 64 history table entries. As the figure shows, using the same number of history table entries as counters is usually a good choice.

References

[BL93] T. Ball and J. R. Larus. Branch prediction for free. In *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, Albuquerque, NM, 1993.

[FF92] J. A. Fisher and S. M. Freudenberger. Predicting conditional branch directions from previous runs of a program. In *Proceedings of ASPLOS V*, pages 85–95, Boston, MA, October 1992.

[HCC89] W. W. Hwu, T. M. Conte, and P. P. Chang. Comparing software and hardware schemes for reducing the cost of branches. In *Proc. 16th Int. Sym. on Computer Architecture*, pages 224–233, May 1989.

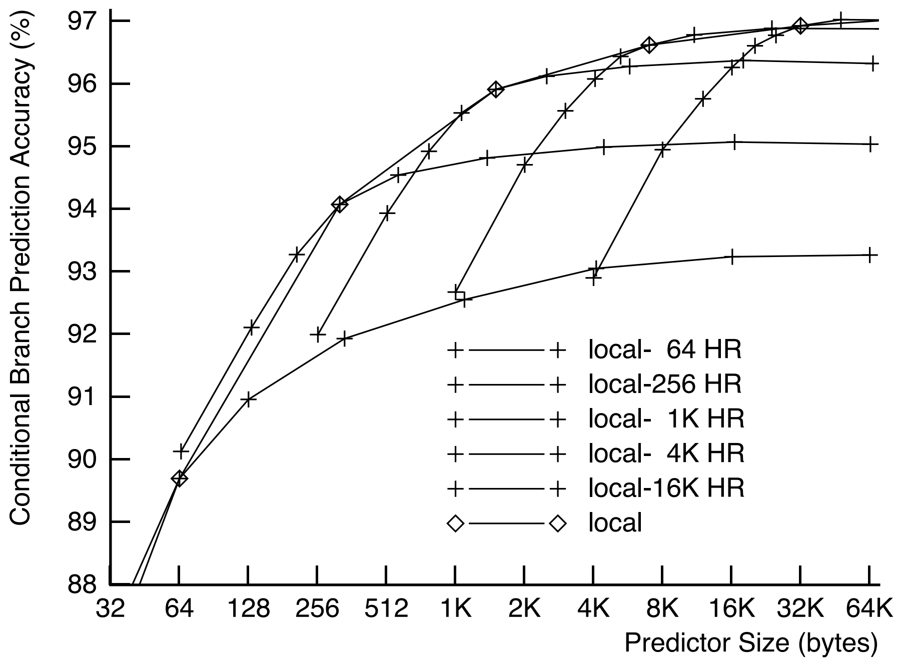

Figure 18: Local Predictor Performance with Varying Number of History Registers

[JW89] N. P. Jouppi and D. W. Wall. Available instruction-level parallelism for superscalar and superpipelined machines. In *Proceedings of ASPLOS III*, pages 272–282, Boston, MA, April 1989.

[Kil86] E. A. Killian. In *RISCompiler and C Programmer's Guide*. MIPS Computer Systems, 930 Arques Ave., Sunnyvale, CA 94086, 1986.

[LS84] J.K.L. Lee and A.J. Smith. Branch prediction strategies and branch target buffer design. *Computer*, 17(1), January 1984.

[LW93] M. S. Lam and R. P. Wilson. Limits of control flow on parallelism. In *Proc. 20th Int. Sym. on Computer Architecture*, May 1993.

[MH86] S. McFarling and J. Hennessy. Reducing the cost of branches. In *Proc. 13th Int. Sym. on Computer Architecture*, pages 396–403, June 1986.

[PSR92] S. T. Pan, K. So, and J. T. Rahmeh. Improving the accuracy of dynamic branch prediction using branch correlation. In *Proceedings of ASPLOS V*, pages 76–84, Boston, MA, October 1992.

[Sit93] R. L. Sites. Alpha AXP architecture. *Communications of the ACM*, 36(2):33–44, Feb. 1993.

[Smi81] J. E Smith. A study of branch prediction strategies. In *Proc. 8th Int. Sym. on Computer Architecture*, pages 135–148, May 1981.

[Smi91] M. D. Smith. Tracing with pixie. Technical Report CSL-TR-91-497, Computer Systems Laboratory, Stanford University, November 1991.

[SPE90] SPEC. *The SPEC Benchmark Report*. Waterside Associates, Fremont, CA, January 1990.

[Wal91] D. W. Wall. Limits of instruction-level parallelism. In *Proceedings of ASPLOS IV*, pages 176–188, Santa Clara, CA, April 1991.

[YP92] T. Y. Yeh and Y. N. Patt. Alternative implementations of two-level adaptive branch prediction. In *Proc. 19th Int. Sym. on Computer Architecture*, pages 124–134, May 1992.

[YP93] T. Y. Yeh and Y. N. Patt. A comparison of dynamic branch predictors that use two levels of branch history. In *Proc. 20th Int. Sym. on Computer Architecture*, pages 257–266, May 1993.