To appear in Proceedings of the 31st International Symposium on Microarchitecture, Nov. 30 - Dec. 2, 1998

Analyzing the Working Set Characteristics of Branch Execution

Sangwook P. Kim* and Gary S. Tyson

Advanced Computer Architecture Laboratory
Department of Electrical Engineering and Computer Science
The University of Michigan
Ann Arbor, Michigan 48109-2122
{swkpeter, tyson}@eecs.umich.edu

Abstract

To achieve highly accurate branch prediction, it is necessary not only to allocate more resources to branch prediction hardware but also to improve the understanding of branch execution characteristics. In this paper, we present a new profile-based conditional branch analysis technique called ***branch working set analysis*** *to provide additional information about control flow behavior of general purpose applications. This analysis evaluates the dynamic behavior of branch execution by partitioning either individual branches or pre-classified branch groups into sets based on temporal locality and ordering information. We refer to these sets as the working sets of branches.*

To demonstrate the usefulness of this form of analysis, we examine the efficiency of current allocation techniques for branch history table (BHT) space and propose a new solution to this allocation process that improves the performance of these tables. In our approach the mapping between branch instructions and BHT entries is specified during compilation to reduce table contention -- leading to more relevant histories and improved predictor performance. As a result, even for programs with a large number of static branches, only 100 to 200 history entries are needed to approximate the performance of larger 1024-entry BHT. Furthermore, when the technique is applied to a predictor with 1024-entry BHT, its prediction accuracy is improved by 16% -- comparable with the performance of a BHT of infinite capacity.

*Currently employed with Intel Corporation, Santa Clara, CA 95052

1. Introduction

It is well-known that achieving high performance in a wide issue and deeply pipelined processor demands a highly accurate branch prediction mechanism. Branch prediction research has developed increasingly sophisticated analysis methods to understand the underlying execution behavior of branch instructions. This analysis has lead to the development of a string of new, powerful predictors in both academic and industrial research and development initiatives. Since the introduction of 2-level branch prediction [2, 3], branch prediction research has focused on improving predictor performance by using more sophisticated analysis techniques to identify the characteristics of mispredictions. It has been shown [4, 6, 10] that certain branches are better predicted with history information derived from different branch instructions (*inter-correlation*) while others have better prediction accuracy when branch history is limited to that branch instruction (*intra-correlation*). Designs of hybrid branch predictors [6, 14] seek to improve predictor accuracy by incorporating multiple predictors with a selection mechanism to pick the best predictor (either inter-correlated or intra-correlated). Recent work has explored the effect of interfering branches in prediction tables [11] and the feasibility of classifying branches into groups with similar behavior [9]. This work has been shown to improve the prediction accuracy of 2-level branch predictors [15, 18]. In addition, program or profile-based static branch predictions [5, 8, 12, 13, 16, 25] and branch alignment techniques [7, 17] have exploited compiler supports to improve the performance of branch prediction -- both static and dynamic.

In this paper, we propose a new profile-based conditional branch analysis technique called *branch working set analysis* which provides additional information about the behavior of branch execution and how that behavior effects the underlying dynamic branch prediction microarchitecture. This analysis is based on the notion that the execution of conditional branches in a general purpose program contain temporal locality and ordering characteristics just as the execution of other operations and data references. N. Gloy et al [19] proposed a procedure reordering method that utilizes the temporal ordering information related to function level execution interleaving, and showed that their method can significantly reduce I-cache conflict misses. We adapt their techniques to improve the efficiency of the resources used to perform branch prediction with the goal of improved predictor performance.

Our analysis of branch interaction evaluates the interleaving of conditional branch execution obtained from profiling several runs of a program. The branch instructions in a program can then be partitioned into sets such that branches whose execution interleaves above certain threshold are placed in a same set. We refer to these sets as the working sets of branches; branch instructions in the same working set compete for predictor resources potentially increasing branch table interference. The method to determine the working sets of branches is similar to live variable data flow analysis and a graph coloring based register allocation technique [21], only instead of dealing with variables and register specifiers, the analysis is performed on conditional branch instructions.

To demonstrate the potential use of the working set analysis related to conditional branch execution, we examine an alternative branch prediction technique called *branch allocation,* which enables a compiler to manage an underlying hardware branch prediction table by specifying the mapping between the table entries and conditional branch instructions. In this paper, we apply branch allocation to improve the efficiency of the BHT of a 2-level predictor. By assigning each static conditional branch to a BHT entry which does not match that of other branch instruction in the same working set, we can minimize table interference and improve prediction accuracy. The mapping method used in the allocation technique closely follows a graph coloring based register allocation technique that enables a compiler to efficiently manage the processor register file.

To illustrate performance benefits of the branch allocation on BHT, a local history based 2-level branch predictor, PAg [3] is chosen, and performance comparison of SPECint95 and several common UNIX applications [20] are presented.

This paper is organized into six sections. Section 2 presents some related works in branch prediction. Section 3 describes the simulation environment used for the branch working set analysis and the branch allocation. Section 4 presents the branch working set analysis methodology and its results. Section 5 introduces and analyzes the branch allocation technique and its performance results. Finally, section 6 concludes the paper.

2. Related Work

Recent research in branch prediction has focused on understanding execution behavior of branch instructions to further enhance already accurate dynamic prediction mechanisms. In this section, we examine research relating to branch working set analysis and conflict avoidance in the BHT.

N. Gloy et al [19] introduced an algorithm to minimize instruction cache conflicts by reordering procedures based on the specifics of an I-cache configuration and the temporal ordering of procedure invocation. In branch working set analysis, we examine the temporal ordering of branch instructions to improve branch prediction accuracy.

Branch working set analysis partitions branches or pre-classified branch groups into sets based on their amount of execution interleaving. Branches are placed in the same working set if their execution is interleaved above certain threshold value. The algorithm used to calculate the degree to which branches are interleaved will be presented later in this paper. If branch instructions exhibit the same execution behavior (e.g. high biased to be taken) it may be beneficial to gather the branches into groups which share resources. As an example, two branches that almost always proceed down the taken path can share the same history in the BHT without a reduction in predictor accuracy -- their histories would be the same anyway. By grouping the resource (branch history) of these branch instructions contention for remaining BHT entries can be reduced. P.-Y. Chang et al [9] introduced a mechanism called *branch classification* to enhance branch prediction accuracy by classifying branches into groups of highly biased (towards taken and towards not taken) or unbiased branches and used this information to reduce the conflict between branches with different classifications. In the branch working set analysis, to improve the performance of the branch allocation, we reduce the working set size of branches in a program by incorporating the taken frequency based branch classification into the analysis, treating all highly biased branches (e.g. not taken) as a single branch group sharing predictor resources (e.g. BHT history).

B. Calder et al [7] proposed improved basic block reordering algorithms called *branch alignments* which incorporate static or dynamic branch prediction architectures, and perform a link-time basic block reordering to reduce mispredict and misfetch penalty cycles.

Since the introduction of 2-level branch prediction [2, 3], dynamic branch prediction research has concentrated on analyzing and enhancing the 2-level schemes [4, 6, 9, 10, 11, 14, 15, 18, 24]. In general, a 2-level predictor uses a part of instruction fetch address in PC as an index value to its first level history table such as BHT or its second level history table in conjunction with branch direction outcomes from the first level table. The PC based index hashing generally uses a modulo of the low order instruction fetch address bits to indicate the table index. This will lead to conflicts among branches that share the same low order bits. Talcott et al [11] analyzed the effect of branch prediction table interference and showed its negative influence on prediction accuracy. P.-Y. Chang et al [15] and E. Sprangle [18] suggested hardware solutions to the negative interference problems in the second level table of 2-level prediction scheme by filtering biased branches [15] and by neutralizing negatively interfering branches [18]. For the purpose of the branch allocation, we attempt to reduce the contention in BHT by statically assigning branches to entries in BHT based on the working set analysis information.

3. Simulation Environment

All simulations in this study are conducted using the SimpleScalar tool set [22] provided by T.M. Austin and D. Burger from the University of Wisconsin. For the branch working set analysis the SimpleScalar toolset is used to generate necessary profile information on conditional branch execution. Then, we have designed analysis tools -- adapted from live variable data flow analysis used in graph coloring based register allocation algorithms -- to process the data on conditional branches in order to summarize the temporal locality and ordering information. Finally, the last step of analysis parses the temporal ordering information and generates the working set information of conditional branches.

In branch allocation we use the analysis of branch interleaving to assign each branch instruction to a BHT entry. For the performance comparison between a typical local history based 2-level predictor and a predictor with branch allocation technique, we have modified a branch predictor in the base SimpleScalar toolset (sim-bpred) to incorporate the static BHT index assignments generated by the allocation routine.

Six out of eight SPECint95 benchmarks and several common UNIX applications [20] are used to perform the branch working analysis and to demonstrate the performance of the branch allocation technique. All benchmarks are either run to completion or run for the first 500 million instructions. Note that to maintain reasonable time and space for the experiments, we have reduced the number of static conditional branches from each benchmark based on the frequency of occurrences. Table 1 shows the benchmarks, the input sets used and the percentage of dynamic branches analyzed.

Benchmarks	Input set	Total dynamic branches	Dynamic branches analyzed	**Percentage of the dynamic branches analyzed**
compress	compress_small.in	17084797	17082310	**99.99%**
gcc	jump.i	31060483	29116913	**93.74%**
ijpeg	vigo.ppm	47371202	47370048	**99.99%**
li	li_ref.out	117061934	117050321	**99.99%**
m88ksim	ctl.big	113363291	113350675	**99.99%**
perl	scrabbl.in	7764844	7752111	**99.84%**
chess	sim.in	23448420	23426413	**99.91%**
gs	sigmetrics94.ps	40286429	40224125	**99.85%**
pgp	IJPP97.ps	48456134	48437347	**99.96%**
plot	surface2.dem	55500887	55476752	**99.96%**
python	yarn.tests.py	47134930	47105870	**99.94%**
simplescalar (ss)	test-fmath	18961136	18940120	**99.89%**
tex	output-PACT96.tex	27516428	27488833	**99.90%**

Table 1: Benchmarks, Input sets and Percentage of branches analyzed

4. Branch Working Set Analysis

We propose a new profile-based conditional branch analysis technique called *branch working set analysis*. Branch working set analysis uses profile information, built from one or more runs of an application to determine the execution behavior of conditional branches. It utilizes a summary of temporal locality and ordering information collected during conditional branch profiling to recognize execution interleaving of conditional branches. The following subsections present the detailed methodology used in branch working set analysis and its primary results.

4.1 Methodology

The first step of the analysis is to identify the amount of execution interleaving for each conditional branch. This step is analogous to the live variable data flow analysis typically used for a graph coloring based register allocation [21]. The difference is that the working set analysis is profile-based and the live variable analysis is a program-based static technique.

To identify the execution interleaving of conditional branches, during the profile run of a program, we mark each branch with a time stamp when the branch is executed. we use a count of the number of instructions executed prior to that dynamic branch instance as a domesticate. Figure 1 shows an example of this process for an application that executes one branch every 5 instructions.

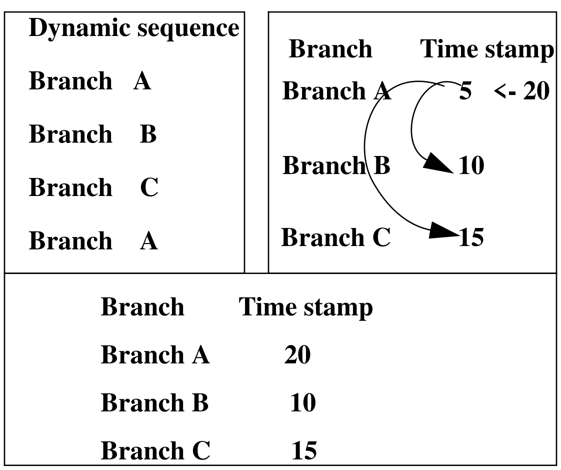

Figure 1: Time stamp analysis example of branch execution behavior

Referring to Figure 1, branch A is time stamped **5** indicating that there have been 5 instructions executed prior to branch A. The profile run continues and executes branches B and C, and each receives time stamp values **10** and **15** respectively. After branch C, we encounter another dynamic instance of branch A. We then check if there is any branch with time stamp greater than the branch A's time stamp of 5; both B and C have time stamps that are greater than the branch A's time stamp. At this point, we have identified the execution interleaving of branches A with B, and C. The second step of the analysis records each instance of branch execution interleaving between branches A and B, and branches A and C. This will be used to calculate an overall interleave count during program execution. After the second step, branch A's time stamp is updated with a new time stamp (**20**) and processing continues until the end of profile run is reached.

The second step of the analysis summarizes the temporal locality and ordering information by constructing a conflict graph based on the branch execution interleaving information from the first step analysis. This is analogous to constructing a variable conflict or interference graph in a graph coloring based register allocation. A variable conflict graph is composed of nodes which represent variable names and edges which represent a conflict, meaning, if there is an edge between two nodes, the two variables are simultaneously live. In Figure 2, the variable conflict graph reveals the fact that the variables a, b, and c are simultaneously live.

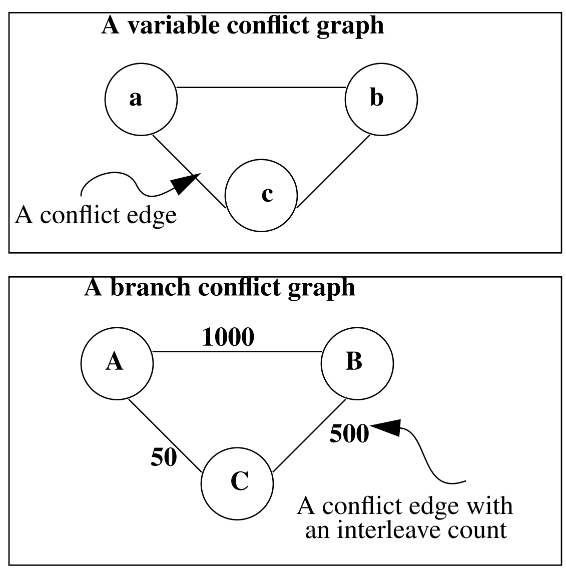

Figure 2: Variable conflict graph and branch conflict graph examples

A conflict graph for summarizing the temporal locality and interleaving related to conditional branch execution is similar to that of a variable conflict graph. As shown in Figure 2, each node in the branch conflict graph represents a conditional branch instruction, and an edge between two nodes represent that the execution of the two branches is interleaved at some point in the profile run of a program. Execution interleaving for a pair of branches can occur multiple times during the profile run, hence, to capture the multiple occurrences of branch interleaving, we add an interleave counter along with each edge in the branch conflict graph. For instance, in Figure 2, the execution of branch A and B have been interleaved for 1000 times.

The final step of the analysis partitions the conditional branch instructions into working sets based on the branch conflict information from the first two analysis steps. Our definition of a working set is a set of conditional branch instructions which form a completely interconnected subgraph in the branch conflict graph. Note that many other definitions of a working set are possible and undoubtedly some will prove better at categorizing branches, but for the simplicity of the study, a complete subgraph definition is used.

4.2 Analysis and Results

Before presenting the results of the analysis, there is one refinement step applied to a branch conflict graph. Each edge in the conflict graph identifies the number of times two branch instructions have been interleaved during execution. The interleave count for two branches can be very high if they are in the same loop, but for others, the count can be trivially small and can be eliminated from consideration to reduce processing time.

To eliminate small and incidental conflicts, a threshold value is given and any edge with a smaller count than the threshold is eliminated. This can effectively reduce the size of the conflict graph while having little impact on the overall working set information. We have chosen a threshold value of 100; the majority of edge values for branches in a working set are several orders of magnitude higher than 100, but 100 is high enough to eliminate a significant number of edges for programs with a large number of working sets. Other threshold values such as 500 or 1000 show no significant difference on the results. Table 2 shows the primary results of the analysis. For each benchmark shown in Table 2, the total number of working sets, the static average number of branches in a working set, and its dynamic average number -- weighted by branch execution count -- are presented.

Benchmarks	Total static # of working sets	**Average static working set size**	**Average dynamic working set size**
compress	224	**41**	**25**
gcc	51888	**365**	**336**
ijpeg	246	**27**	**36**
li	2792	**178**	**154**
m88ksim	1203	**144**	**150**
perl	1079	**51**	**51**
chess	23936	**250**	**244**
pgp	775	**45**	**39**
plot	5370	**143**	**185**
python	25216	**347**	**318**
ss	19368	**287**	**246**

Table 2: The Sizes of Branch Working Sets

The primary result in Table 2 shows that the number of conditional branches that are in the same working set is relatively small. For example, *compress* shows an average of only 25 branch instructions while even *gcc* which contains more than 16,000 static conditional branches shows an average working set size of only 336. This indicates that a branch predictor only needs to allocate its cache-like history table space for a relatively small set of conditional branches at one time following the temporal ordering information.

We believe that analyzing the working set behavior of branch execution can be used to improve predictor performance by getting more efficient utilization from predictor resources. In addition, although not presented in this paper, it is clear that the analysis can be applied not only to individual branches but also to pre-classified branch groups to be incorporated with other branch analysis techniques.

5. Branch Allocation

To demonstrate the use of the working set analysis we now propose a mechanism called *branch allocation* enabling compiler control of the hardware prediction table by statically assigning each conditional branch to a prediction table entry. In this paper, we apply the branch allocation technique to improve the utilization of the BHT. This is achieved by improving the mapping function between conditional branches and BHT in a local history based 2-level branch predictor. As discussed earlier, in a 2-level predictor, the BHT index is generally determined by hashing the low order bits of fetch address in PC. This leads to contention among branches that share the same low order bits. The branch allocation technique enables a compiler to directly manage the mapping of branch instruction to table entries. The compiler can then use the profile based branch working set analysis to minimize the number of conflicts, or when classification is used to minimize the interference caused by conflict between branches in different groups.

In this study, we augment the format of branch instructions to include an index specifier for BHT entries. However, the allocation of index bits in any existing branch ISA would be difficult, and without changing the ISA, branches in library routines will not be affected by the allocation technique. If augmenting an ISA is not an option, the working set information used in the allocation technique can be incorporated into a branch alignment transformation [7] for any ISA without change although it may not be as effective as our scheme. In addition, for a hardware predictor to utilize a static mapping index to generate a prediction at the fetch cycle of a branch instruction, the index value may be needed with the preceding branch instruction, or hardware support to cache the index values may be required[1].

5.1 Methodology and Analysis

The branch allocation is performed in three steps. The first two steps are exactly the same as the first two steps in the branch working analysis summarizing the temporal locality and ordering information related to conditional branch execution by constructing a branch conflict graph as shown in Figure 2. The final step of the allocation technique processes the branch conflict graph and specifies an index to BHT entry for each conditional branch in much the same manner as a graph coloring based register allocator specifies a register for each variable. The difference is that in register allocation, register spilling requires modifications of code to eliminate simultaneously live values, while in branch allocation, it need not eliminate conflicts, it only attempts to minimize them. In other words, if it is determined that a working set has too many member branch instructions for a one to one mapping into the BHT table, multiple branches within the same working set are mapped to the same BHT entry location. The allocation routine chooses the branches with the fewest conflicts among the working set branches to map to the same location in order to minimize contention.

In Table 3 below, we show the BHT size necessary to allow branch allocation to reduce the table conflicts to below that of a 1024-entry conventional BHT with PC indexing scheme.

Benchmark	BHT size required for branch allocation
chess	320
compress	208
gcc	544
gs	740
li	270
m88ksim	166
perl_a	288
perl_b	288
pgp	188
plot	224
python	570
ss_a	336
ss_b	360
tex	680

Table 3: BHT size required for the branch allocation technique

The results from Table 3 show that for most of the benchmarks, the BHT size requirements are in the

1. The parameters of a cache of indices would have to be carefully managed to avoid the original problem of contention, only this time in the cache instead of the BHT.

200 to 400 range, indicating that the branch allocation based indexing scheme has a potential to significantly reduce the conflicts or conversely a smaller BHT is possible with proper allocation of table entries. Even for *gcc* benchmark which contains more than 16,000 static conditional branches, approximately half of the conventional BHT size is required when the branch allocation is used.

5.2 Enhancement and Analysis

The branch allocation performance heavily relies on the effectiveness of the working set information in a branch conflict graph. Two branches conflict each other if the amount of execution interleaving is above a threshold value. However, if the two branches have similar execution behavior, the contention between branches may not degrade predictor performance. In other words, the conflicts between the two branches with similar execution characteristics do not contain significant negative effects. P.-Y. Chang et al [9] introduced *branch classification* to enhance branch prediction accuracy by partitioning branches into classes with similar execution characteristics. To improve the effectiveness of the working set information in a branch conflict graph, we utilize branch classification to further eliminate conflict edges for branches exhibiting the same characteristics (remember that these characteristics include highly biased taken and not taken branch execution). By eliminating these conflict edges from the graph, branch allocation can be improved and table size reduced.

When refining the working set information in a branch conflict graph using the branch classification, we identify those branches that are highly biased towards one direction; either greater than 99% taken or less than 1% taken. If two conflicting branches are in the same highly biased class, we ignore the conflict even if it is above a threshold value. If a target ISA allows, these highly biased conditional branches can be statically predicted reducing the requirements of a hardware predictor. If not, two history entries from BHT can be set aside such that highly biased towards *taken* and *not taken* branches can be mapped to these two entries separated from others.

Table 4 shows the BHT size requirements for the branch allocator to reduce the conflicts below that of a 1024-entry conventional BHT when the allocator is incorporated with branch classification. In these results, we see that for all benchmarks the BHT size does not need to exceed 160 entries with the majority of applications needing no more than 50 entries in the BHT. *gs*, which had the largest table requirements in Table 3, now requires only 80 entries in the table.

We also have examined differences in profiling by using two different input data sets on perl and SimpleScalar benchmarks. Interestingly, in SimpleScalar benchmark runs, *ss_a* and *ss_b*, there are significant difference in the table size requirements. This is the result of different areas of the program being exercised depending on the input data set used during profiling. Clearly, any profile based analysis technique including the branch allocation will not be effective when input data for actual run of a program exercises different segments of the code from those executed in the profile run. However, in the case of branch allocation, the branch conflict graphs of several profiles from different input data can be merged until the resulting graph indicates that most part of the program has been exercised. This cumulative profile approach will not necessarily lead to significantly larger table requirements because it is likely that the composed conflict graph will have more total number of working sets, but the size of each working set will not be different significantly.

Benchmark	BHT size required for branch allocation with the branch classification
chess	160
compress	40
gcc	150
gs	80
li	48
m88ksim	40
perl_a	32
perl_b	32
pgp	118
plot	40
python	48
ss_a	160
ss_b	85
tex	80

Table 4: BHT size required for the branch allocation with the branch classification

5.3 Effects on Prediction Accuracy

To illustrate the potential performance benefits of the branch allocation technique, we have compared the misprediction rates of the two branch allocation techniques proposed in section 5.1 and 5.2 against a local history based 2-level branch predictor. For the comparison, PAg scheme [2, 3] with 1024-entry BHT as the first level table and 4096-entry pattern history table (PHT) as the second level table is chosen to be the baseline 2-level predictor. The comparison is performed on several benchmarks from SPECint95 and common UNIX applications [20]. In addition, we have included the misprediction rates of an interference free PAg with 2 million-entry BHT and 4096-entry PHT to see how well each of these schemes approximates an interference free BHT performance.

Figure 3 shows the misprediction rate comparison of the branch allocation technique without the benefits from the branch classification. We have varied the BHT size for the branch allocation based predictor; 16-entry, 128-entry, and 1024-entry, and have compared their performance against PAg with 1024-entry and with an interference free 2 million-entry BHT. For all predictors, a 4096-entry PHT is used. The result shows that on average, a 1024-entry BHT using branch allocation not only outperforms the baseline PAg with 1024-entry BHT but also shows no significant difference as compared to the interference free PAg scheme -- with the exception of *gcc* which contains more than 16,000 static conditional branches pressuring the hardware prediction tables to the limit.

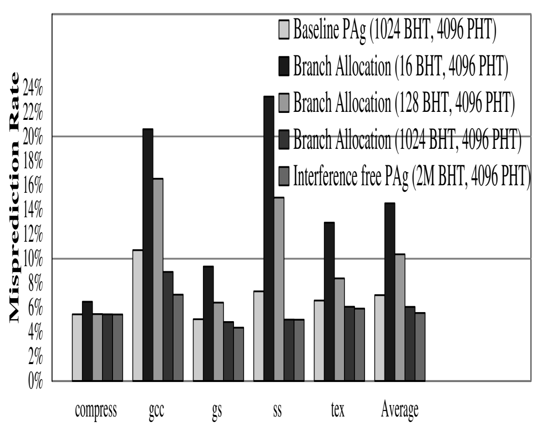

Figure 3: The Branch Allocation performance results without the branch classification

Next, we perform the misprediction comparison for the branch allocation technique with the benefits from the branch classification. As the results in Figure 4 show, branch allocation with only 128-entry BHT outperforms the PAg predictor with 1024-entry BHT with an except of gcc benchmark. When the branch allocation is applied to a 1024-entry BHT predictor, its prediction accuracy improves by 16% showing the efficient utilization of the same size table space. ‘This indicates that if BHT entries are carefully allocated, table sizes greater than 1024-entry are unnecessary to approximate the results of a BHT without conflict (an infinite table).

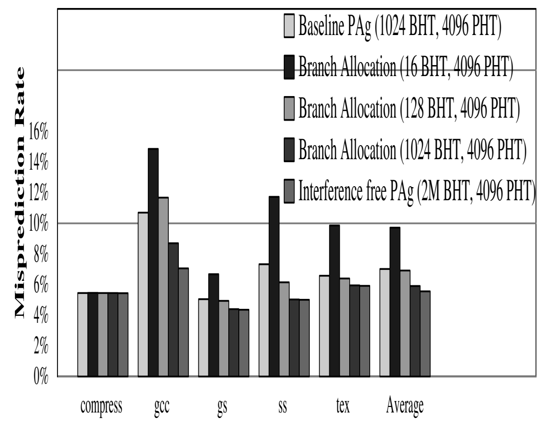

Figure 4: The Branch Allocation performance results with the branch classification

6. Conclusions and Future work

In this paper we have introduced a new profile-based analysis method called *branch working set analysis*. The analysis evaluates the temporal ordering information by capturing interleaved execution of conditional branch instructions obtained from profiling and partitions the branches into working sets. The analysis result reveals that the size of each working set in a general purpose program is relatively small indicating that, at any one time, a branch predictor only needs to allocate its cache-like history table space for a relatively small set of conditional branches while maintaining high prediction accuracy.

To demonstrate the effectiveness of using working sets to manage branch prediction resources, we have examined an alternate compiler controlled technique called *branch allocation.* Branch allocation statically assigns each conditional branch to the first level history table in a 2-level branch prediction scheme efficiently utilizing BHT table space. Result indicate that with branch allocation BHT size can be reduced 60% to 80% without a reduction in performance. By incorporating t *branch classification* in the branch allocation scheme BHT table size can be reduced by as much as 97% without performance degradation. Branch prediction accuracy is shown to improve with branch classification even when fewer hardware resources are available to perform the prediction. When the classification information is incorporated to the allocation technique, 128-entry BHT outperforms the conventional 1024-entry BHT scheme. In addition, the branch allocation based 1024-entry BHT predictor outperforms a conventional BHT predictor of the same size by 16%. Furthermore, with the branch allocation technique managing the history table assignments, 1024 entry tables can achieve the performance of PAg predictor with a near conflict free large first level history table.

These results indicate that a feasible alternative to continually expanding the size of prediction tables is to develop better hashing algorithms by analyzing and understanding execution characteristics of conditional branches.

We believe that branch working set analysis provides a new approach to studying branch behavior. This research can be extended in many ways. This work is not limited to individual static conditional branches. Branches can be pre-classified based on intra or inter-correlations and similar history patterns, and the working set analysis can be applied to these pre-classified branch groups. Hence, by incorporating correlation information into the working set analysis and by modifying the branch allocation technique accordingly, we can further improve the prediction accuracy.

Additionally, by identifying branch working sets, we open new research areas in branch prediction.

- Are the clustered branch mispredictions found in recent work on dynamic prediction caused by changes in working set?
- Do clustered cache misses not associated with large vector strides also correlated to the cluster mispredictions?

By developing new analysis methods we hope to develop a deeper understanding of branch execution leading to new and more sophisticated prediction capabilities.

Acknowledgments

The authors would like to thank Sanjay J. Patel and Jared Stark for their support on simulation environment and the anonymous reviewers for providing helpful comments. This research has been supported by the National Science Foundation grant MIP 9734023 and through the Intel Technology for Education 2000 grant.

References

[1] J.E. Smith, “A study of branch prediction strategies,” *In Proceedings of the 8th International Symposium on Computer Architecture, pp. 135-148, May 1981.*

[2] T.-Y. Yeh and Y.N. Patt, “Two-level Adaptive Training Branch Prediction”, *in Proceedings of the 24th International Symposium on Microarchitecture, 1991.*

[3] T.-Y. Yeh and Y.N. Patt, “Alternative Implementations of Two-level Adaptive Branch Prediction”, *in Proceedings of the 19th International Symposium on Computer Architecture, 1992.*

[4] S.-T. Pan, K. So, and J. T. Rahmeh. “Improving the accuracy of dynamic branch prediction using branch correlation”, *in Proceeding of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems, pages 76-84, Boston, Mass., October 1992.*

[5] T. Ball and J.R. Larus, “Branch prediction for free”, *in Proceedings of SIGPLAN Conference on Programming Language Design and Implementation, June 1993.*

[6] S. McFarling, “Combining branch predictors”, *TN 36, DEC-WRL, June 1993.*

[7] Brad Calder and Dirk Grunwald, “Reducing Branch Costs via Branch Alignment”, *in Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems, Oct. 1994.*

[8] C. Young and M.D. smith, “Improving the Accuracy of Static Branch Prediction Using Branch Correlation”, *in Proceedings of ASPLOS VI, Oct. 1994.*

[9] P.-Y. Chang, E. Hao, T.-Y. Yeh, and Y.N. Patt, “Branch Classification: a New Mechanism for Improving Branch Predictor Performance”, *in Proceedings of the 27th International Symposium on Microarchitecture, 1994.*

[10] C. Young, N. Gloy, and M.D. Smith, “A Comparative Analysis of Schemes for Correlated Branch Prediction”, *in Proceedings of the 22nd Annual International Symposium*

on Computer Architecture, 1995.

[11] A. R. Talcott, M. Nemirovsky, and R.C. Wood, "The Influence of Branch Prediction Table Interference on Branch Prediction Scheme Performance", *in Proceedings of International Conference on Parallel Architectures and Compilation Techniques, 1995.*

[12] Brad Calder, Dirk Grunwald, Donald Lindsay, James Martin, Michael Mozer, and Benjamin Zorn, "Corpus-based Static Branch Prediction", *in Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation, June 1995.*

[13] J.R.C. Patterson, "Accurate Static Branch Prediction by Value Range Propagation", *in Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation, June 1995.*

[14] P.-Y. Chang, E. Hao, and Y.N. Patt, "Alternative Implementations of Hybrid Branch Predictors", *in Proceedings of the 28th International Symposium on Microarchitecture, 1995.*

[15] P.-Y. Chang, M. Evers, and Y.N. Patt, "Improving branch prediction accuracy by reducing pattern history table interference", *in Proceedings of the 1996 ACM/IEEE Conference on Parallel Architecture and Compilation Techniques, 1996.*

[16] S. A. Mahlke and B. Natarajan, "Compiler Synthesized Dynamic Branch Prediction", *in Proceedings of the 29th International Symposium on Microarchitecture, 1996.*

[17] C. Young, D. S. Johnson, D. R. Karger, and M.D. Smith, "Near-optimal Intraprocedural Branch Alignment", *in Proceedings of PLDI, June 1997.*

[18] Eric Sprangle, Robert S. Chappell, Mitch Alsup, and Yale N. Patt, "The Agree Predictor: A Mechanism for Reducing Negative Branch History Interference," *in Proceedings of the 24th International Symposium on Computer Architecture, Denver, June 1997.*

[19] N. Gloy, T. Blackwell, M.D. Smith, and B. Calder, "Procedure Placement Using Temporal Ordering Information", *in Proceedings of the 30th International Symposium on Microarchitecture, Dec. 1997.*

[20] J. Stark, P. Racunas, and Y. N. Patt, "Reducing the performance impact of instruction cache misses by writing instructions into the reservation stations out-of-order", *in Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture, 1997.*

[21] Steven S. Muchnick, "Advanced Compiler Design & Implementation", 1997.

[22] T.M. Austin and D. Burger, "The SimpleScalar Tool Set, Version 2.0", 1997.

[23] T. Ball, P. Mataga, and M. Sagiv, "Edge Profiling versus Path Profiling: The Showdown", *in proceeding of the 25th ACM Symposium on Principles of Programming Languages, Jan. 1998.*

[24] Marius Evers, Sanjay J. Patel, Robert S. Chappell, Yale N. Patt, "Analysis of Correlation and Predictability: What Makes Two-Level Branch Predictors Work," *in Proceedings of the 25th International Symposium on Computer Architecture, Barcelona, June 1998.*

[25] B. Deitrich, B. Cheng, and W. Hwu, "Improving Static Branch Prediction in a Compiler", *in Proceedings of International Conference on Parallel Architectures and Compilation Techniques, 1998.*

[26] J. Kalamatianos and D. Kaeli, "Temporal-based Procedure Reordering for Improved Instruction Cache Performance", *in Proceedings of the 4th International Symposium on High Performance Computer Architecture, 1998.*