International Journal of Parallel Programming, Vol. 24, No. 2, 1996

Branch Classification: A New Mechanism for Improving Branch Predictor Performance

Po-Yung Chang,[1] Eric Hao,[1] Tse-Yu Yeh,[2] and Yale Patt[1]

There is wide agreement that one of the most significant impediments to the performance of current and future pipelined superscalar processors is the presence of conditional branches in the instruction stream. Speculative execution is one solution to the branch problem, but speculative work is discarded if a branch is mispredicted. For it to be effective, speculative execution requires a very accurate branch predictor; 95% accuracy is not good enough. This paper proposes branch classification, a methodology for building more accurate branch predictors. Branch classification allows an individual branch instruction to be associated with the branch predictor best suited to predict its direction. Using this approach, a hybrid branch predictor can be constructed such that each component branch predictor predicts those branches for which it is best suited. To demonstrate the usefulness of branch classification, an example classification scheme is given and a new hybrid predictor is built based on this scheme which achieves a higher prediction accuracy than any branch predictor previously reported in the literature.

KEY WORDS: Branch classification; branch prediction; speculative execution; processor performance; superscalar.

1. INTRODUCTION

Branches can significantly reduce the performance of pipelined processors if they interrupt the steady supply of instructions to the instruction pipeline.[(1)] A branch predictor reduces the number of pipeline stalls by predicting the direction of the branch and fetching the instructions from

[1] Department of Electrical Engineering and Computer Science, The University of Michigan, Ann Arbor, Michigan 48109-2122.
[2] Intel Corporation, Santa Clara, California 95051.

0885-7458/96/0400-0133$09.50/0

that path. Because all speculative work beyond a branch must be thrown away if that branch is mispredicted, a very accurate branch prediction algorithm is important to a high-performance microprocessor.

If we ignore stalls such as cache misses and bus conflicts, the branch penalty is defined as $C \times ((1-p) \times r \times ipc)$, where C denotes the number of cycles wasted due to a branch misprediction, p denotes the prediction accuracy, r denotes the ratio of the number of branches over the number of total instructions, and ipc denotes the average number of instructions that are executed per cycle. For $C=5$ and $r \times ipc = 0.9$, a branch penalty of less than 10% requires a prediction accuracy p of greater than 97.7%. Other stalls such as cache misses and bus conflicts can further increase the branch penalty. For example, if the branch predicates do not reside in the cache, fetching these values from memory will result in longer branch resolution times and, thus, more cycles wasted when the branch is mispredicted.

To improve branch prediction accuracy, McFarling[2] proposed a hybrid branch prediction scheme that combines two single-scheme predictors into one branch predictor. For each branch, the hardware dynamically selects which one of the two component predictors to use to make the prediction. In this paper, we introduce branch classification as a methodology for designing hybrid branch predictors. Branch classification partitions a program's branches into sets or *branch classes.* A good classification scheme partitions branches possessing similar dynamic behavior into the same branch class; thus, once we understand the dynamic behavior of a class of branches, we can optimize for this class. For example, the compiler can try to eliminate hard-to-predict branches or the hardware can special case the handling of these branches (e.g., execute both paths of the branch). In addition, branch classification can be used to maximize the prediction accuracy obtained from a given hardware budget. Prediction accuracy is increased by associating each branch class with the most suitable predictor for that class. For example, we could use a simple predictor for predictable branches and dedicate more resources to handle branches that are more difficult to predict. In this paper, we introduce one example of branch classification and demonstrate how this branch classification model can be used to analyze the performance of individual branch predictors in order to determine effective combinations of those predictors. In addition, we propose a new hybrid branch predictor based on this branch classification model. This new predictor outperforms previously proposed predictors, showing branch classification's usefulness in designing more accurate branch predictors.

This paper is organized as follows: Section 2 presents previous work. Section 3 describes our simulation methodology. Section 4 describes a

branch classification model, proposes several new hybrid branch prediction schemes based on this model, and presents simulation results. Section 5 provides some concluding remarks.

2. PREVIOUS WORK

To improve prediction accuracy, various branch prediction strategies have been studied. These prediction schemes can be divided into two groups, static and dynamic predictors.

Static branch prediction algorithms use information gathered before program execution, such as branch opcodes or profiles, to predict branch direction. The simplest branch predictor is one that predicts that all conditional branches are always taken (as in Stanford MIPS-X),[3] or always not-taken (as in Motorola MC88000).[4] Predicting all branches to be taken achieves about 65% accuracy whereas predicting not-taken achieves about 35%.[5–7] The profile-guided branch predictor bases its prediction on the direction the branch most frequently takes.[8] This is determined by profiling the program on a training input set. To pass the prediction information to the hardware, modifications to the ISA are required; an additional bit to indicate the branch direction is added to the branch opcode.

Dynamic branch prediction algorithms use hardware to record branch execution history at run-time, and predict future branch directions based on that history. Many dynamic prediction methods have been studied.[7, 9–12] Smith[7] proposed a branch predictor which uses a table of two-bit saturating up-down counters to keep track of branch history. Each branch is mapped via its address to a counter which provides the prediction. If the most-significant bit of the associated counter is set, a branch is predicted to be taken; if clear, the branch is predicted to be not-taken. These counters are incremented/decremented based on branch outcomes. When a branch is taken, the 2-bit value of the associated counter is incremented by one; otherwise, the value is decremented by one. Branch prediction accuracies of about 85%–90% have been reported for simple history bit and counter-based schemes.[7]

By keeping more history information, a higher level of branch prediction accuracy, about 90%–95%, can be attained. Yeh and Patt[11] proposed dynamic branch predictors which use two levels of history to make branch predictions (see Fig. 1).

The Two-Level Branch Predictor uses k-bit shift register(s), called branch history register(s), to keep the first-level branch history. The shift register represents the branch results of the most recent k branches. If a branch is taken, a "1" is shifted into the branch history register (BHR); if a branch is not taken, a "0" is shifted into the BHR. There are several ways to keep this branch history—Per-address, Per-set, and Global. In the

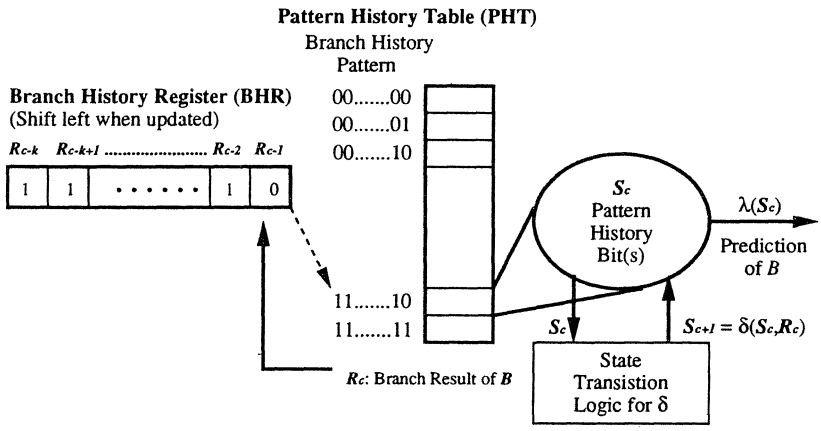

Fig. 1. Structure of Two-level Branch Predictor.

Per-address scheme, each branch history register records the results of the last k occurrences of one static instance of a branch instruction. In the Per-set scheme, each branch history register records the last k outcomes of a given set of branches. In the Global scheme, a global branch history register records the outcomes of the actual last k branches encountered in the dynamic instruction stream.

The second-level history is kept using array(s) of 2-bit saturating up-down counters, called Pattern History Table(s). Each counter keeps track of the more-likely direction of a branch when a particular pattern of first level history is encountered. The selection of the appropriate counter for a branch is based on both the lower bits of its instruction address and the history of recently executed branches. The lower-bits of its instruction address are used to select the appropriate Pattern History Table (PHT) and the content of the branch history register is used to select the appropriate entry within that PHT. The Two-level Branch Predictor can be further classified by the association of PHTs with branches. There can be one global PHT for all branches, one PHT for each set of branches, or one PHT for each static branch in the program.

To further improve prediction accuracy, McFarling[2] proposed a new technique, hybrid branch prediction, that combines two branch predictors. His technique uses 2-bit up-down counters to keep track of which predictor is currently more accurate for each branch; the hybrid predictor uses the more accurate predictor for making its prediction.

McFarling looked at the performance of specific hybrid predictor configurations. In this paper, we present branch classification as a methodology for designing hybrid branch predictors. Branch classification allows us

to examine branch prediction schemes in order to determine which combinations of component predictors make up an effective hybrid branch predictor. Branch classification also allows the construction of hybrid branch predictors by associating each branch with the most suitable component predictor. Since each component predictor only makes predictions for a particular class of branches, the component predictors can be specialized.

3. SIMULATION METHODOLOGY

3.1. Simulation

To collect the dynamic branch behavior, we replace the source code that calculates the branch conditions with calls to the branch simulator. For each of these calls, the branch predictor simulator generates the branch prediction and updates the state of the simulated prediction hardware. Using this method, we can compare the performance of various branch predictors on a per-branch basis. The behavior of the program is not changed because these function calls return the actual branch conditions; the program always executes down the correct path. This approach, however, is not accurate enough to fine-tune a real design because we simulate the branches at the source code level. The behavior of some of these branches at the machine code level may be different than at the source code level due to compiler optimizations. A more accurate approach would use instrumented executable files to measure per-branch prediction accuracy.

3.2. Benchmarks

The results presented in this paper are for the six integer programs from the SPECint92 suite: **espresso, xlisp, eqntott, compress, sc,** and **gcc**

Table I. Training and Testing Data Sets of Benchmarks

Benchmark	Training Data	Testing Data
0.08.espresso	cps	bca
0.22.li	deriv.cl	nine queens
0.23.eqntott	bool.eq.	int_pri_3.eqn
0.26.compress	gcc source	in
0.72.sc	loada2	loada1
0.85.gcc	jump.i	stmt.i

(see Table I). With the exception of **eqntott** and **xlisp**, the training and testing data sets used for these benchmarks were the ones distributed with the SPECint92 suite. **Eqntott** and **xlisp** did not use standard data sets for their training runs because the SPECint92 suite provided only one data set for each of these two benchmarks. For **xlisp**, the training data set, deriv.cl, is a Common Lisp version of a symbolic derivative benchmark written by Vaughan Pratt. For **eqntott**, the training data set, bool.eq, is a set of 27 Boolean equations with 37 different variables.

3.3. Cost Models

In our experiment, three different single-scheme branch predictors are simulated: profile-guided (PG), 2-bit up-down counter (2bC), and the Two-Level Branch Predictor. Three different implementations of the Two-Level Branch Predictor are studied. They are the Per-address Two-Level Branch Predictor using a set of pattern history tables (PAs), the Global Two-Level Branch Predictor using a set of pattern history tables (GAs), and a modified GAg scheme (gshare)[2] that exclusive-ORs the global history with the branch address to select the appropriate pattern history table entry.

For our experiments, we compare predictors of the same implementation cost. The hardware costs of the single-scheme predictors examined are estimated using the following equations[13]:

$$\mathrm{GAs}(k,p) = k + (p \times 2^k \times 2) \qquad \text{(bits)}$$

$$\mathrm{PAs}(k,p) = (b \times k) + (p \times 2^k \times 2) \qquad \text{(bits)}$$

$$\mathrm{gshare}(k,p) = k + (p \times 2^k \times 2) \qquad \text{(bits)}$$

$$\mathrm{2bC}(b) = b \times 2 \qquad \text{(bits)}$$

$$\mathrm{PG} = 0 \qquad \text{(bits)}$$

where k is the history register length, p is the number of pattern history tables (PHTs), b is the number of entries in the branch history table.

4. EXPERIMENTS

4.1. The Advantages of Branch Classification

To demonstrate the benefit of branch classification, we introduce one model of branch classification which partitions branches statically. Branches are classified based on their taken rates which are collected during the profile run as shown in Table II. One possible way to pass the

Table II. Static Classes

Classes	Descriptions
SC1	0% < =pr(br) < =5%
SC2	5% < pr(br) < =10%
SC3	10% < pr(br) < =50%
SC4	50% < pr(br) < =90%
SC5	90% < pr(br) < =95%
SC6	95% < pr(br) < =100%

profile information to the hardware is to encode the branch classification information in the branch instruction format. Because this partitioning of branches is done statically, we will refer to these branch classes as *static classes.* In the following section, we will refer to SC1, SC2, SC5, and SC6 branches as mostly-one-direction branches and the SC3 and SC4 branches as mixed-direction branches. Because branches in different classes have different dynamic behavior, the optimal branch prediction scheme may be different for each of these classes. In this section, we report and analyze the performance of branch prediction schemes on each static class to show the performance benefits of branch classification.

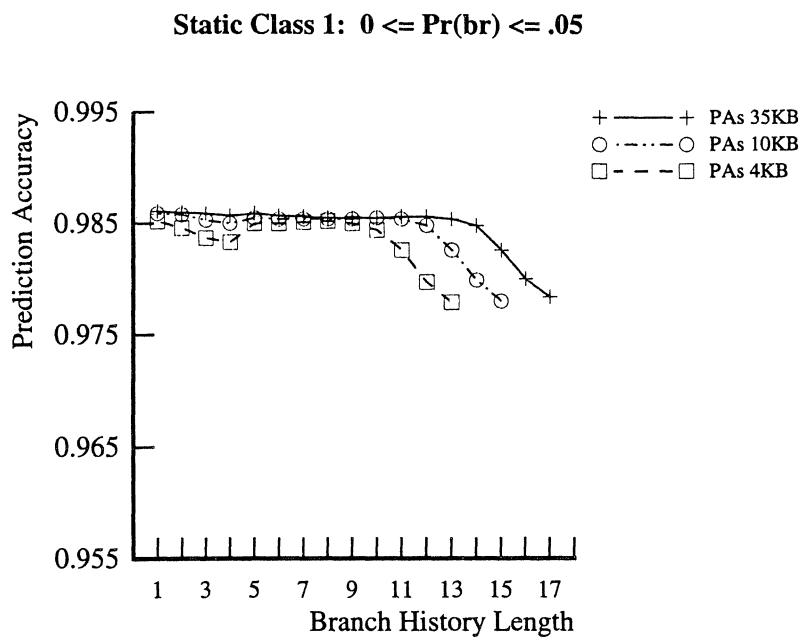

Fig. 2. Prediction accuracy of PAs on the mostly not-taken branches.

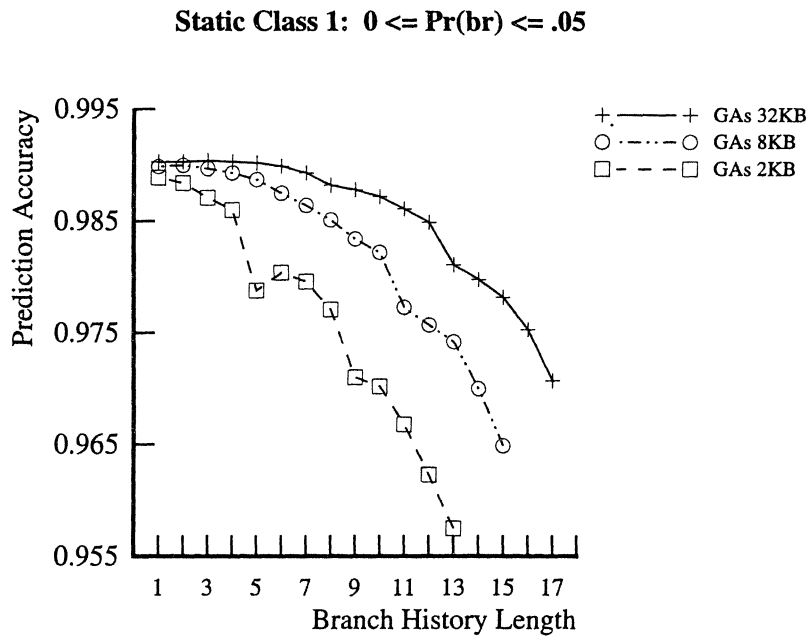

Fig. 3. Prediction accuracy of GAs on the mostly not-taken branches.

- *Analysis of the Mostly One-direction Branches* (*SC*1, *SC*2, *SC*5, *SC*6)

  Figures 2–4 show the prediction accuracy of SC1 branches using the PAs, the GAs, and the gshare schemes respectively. Each curve shows the prediction accuracy for a constant cost of implementation. Since the Two-level Branch Predictor uses the contents of the BHR to select the appropriate PHT entry (as shown in Fig. 1), the length of the branch history register indicates the size of each PHT; for example, the k-bit BHR implies PHTs of 2^k entries. Thus, for a given hardware cost, an implementation of the Two-level Branch Predictor with a longer BHR contains larger, but fewer, PHTs. Likewise, a shorter BHR means smaller, but more, PHTs. Therefore, for each curve in Figs. 2–4, as the branch history length decreases by one, the number of pattern history tables doubles.

  As shown in these figures, branch prediction schemes with short history registers are most effective in predicting the mostly-not-taken branches (SC1). Because we are looking at implementations of a fixed cost, there is a trade-off between having a longer branch history and having more PHTs. One advantage of having

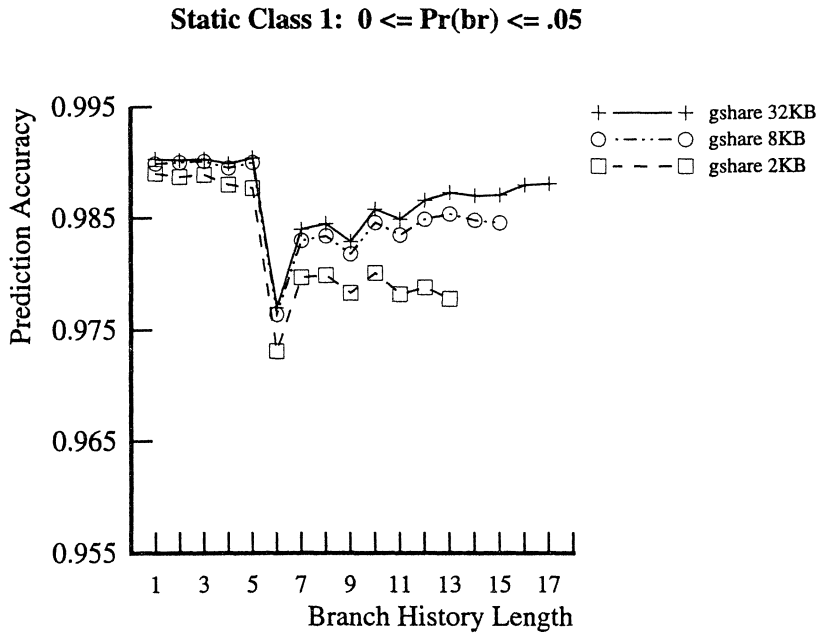

Fig. 4. Prediction accuracy of gshare on the mostly not-taken branches.

a shorter branch history is a faster predictor warm-up time because fewer PHT entries are accessed with a shorter BHR. A shorter history also reduces the amount of PHT interference because we have more PHTs resulting in fewer branches being mapped to the same PHT. Since branches in SC1 are mostly not-taken, prediction accuracy remains high even if the taken occurrences of the branch are mispredicted. Although a longer branch history register can keep more history of correlated branches and possibly capture these odd occurrences, the benefit of having a faster predictor warm-up time and less PHT interference outweighs the benefit of eliminating these mispredictions. Similar results are shown for the SC2, SC5, and SC6 branches because these classes also have the same dynamic characteristic of heavily favoring one direction over the other.

- *Analysis of the Mixed-direction Branches* (*SC*3,*SC*4)
  Figures 5–7 show the prediction accuracy of SC3 branches using the PAs, the GAs, and the gshare schemes respectively. Unlike the mostly-one-direction branches, the mixed-direction branches are most effectively predicted by prediction schemes with long branch history registers. Because these branches have dynamic taken rates

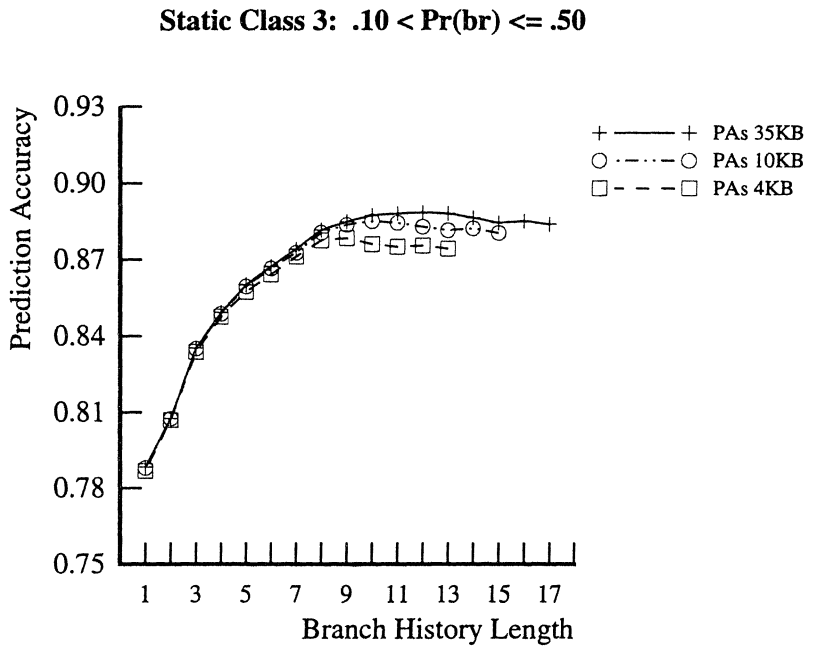

Fig. 5. Prediction accuracy of PAs on the mixed-direction branches.

between 10% and 50%, the predictor sees more execution patterns due to the mixing of taken and non-taken directions in the branch history. By having a longer branch history, we can distinguish more execution states. In addition, Hammerstrom and Davidson[14] have shown that there is a significant amount of correlation between instruction addresses by measuring the entropy in instruction address traces. With a longer branch history, the histories of correlated branches are more likely to remain in the branch history register, making the branch prediction scheme more effective in predicting the mixed-direction branches. The SC4 branches with taken rates between 50% and 90% showed similar results for the same reasons.

The importance of accurately predicting a class of branches depends on how frequently these branches are executed in the dynamic instruction stream. Figure 8 shows the dynamic weight of each static class, where the dynamic weight of a branch class is defined as

$$\frac{\textit{Number of dynamic branches belonging to that branch class}}{\textit{Total number of dynamic branches}}$$

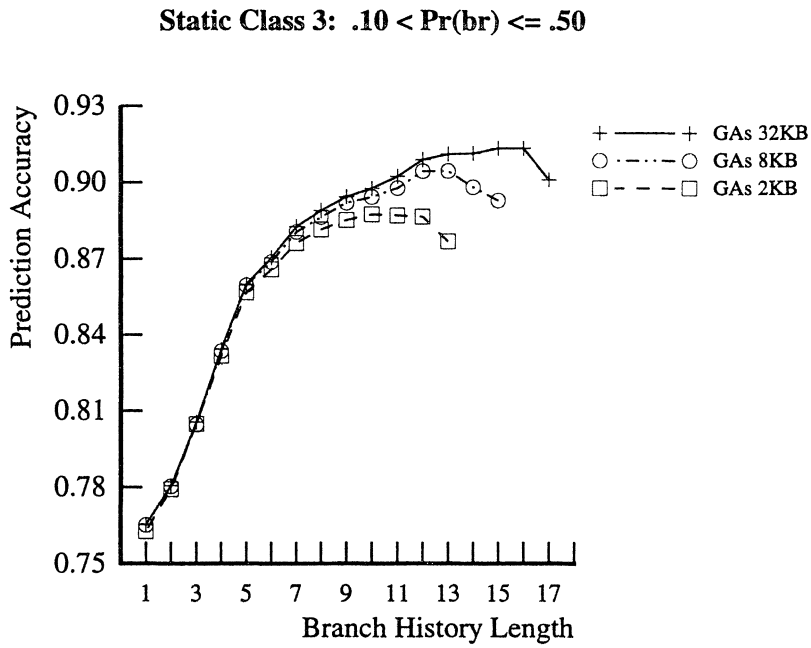

Fig. 6. Prediction accuracy of GAs on the mixed-direction branches.

Approximately 50% of all dynamic branches are mostly-one-direction branches and 50% are mixed-direction branches Thus, the performance of a predictor is dependent on its prediction accuracy for both types of branches.

We have shown that the optimal predictor configuration of the Two-level Branch Predictor for the mostly-one-direction branches is different from that of the mixed-direction branches. Thus, a single-scheme predictor cannot be configured optimally for both types of branches. Figures 9–11 show the average prediction accuracies over all branches using the GAs, PAs, and gshare schemes with the branch history length ranging from 1 to 18. Each curve in the graphs indicates the performance of a branch predictor at a fixed hardware cost. Our results show that the best predictor configurations for predicting all branches have branch history lengths that are neither optimal for the mostly-one-direction branches nor for the mixed-direction branches. For example, Fig. 10 shows that the best 32K-byte GAs configuration for predicting all branches uses 13 bits of branch history while Fig. 6 shows that for the mixed-direction branches only, the best branch history length is 16 and Fig. 3 shows that for the mostly-one-

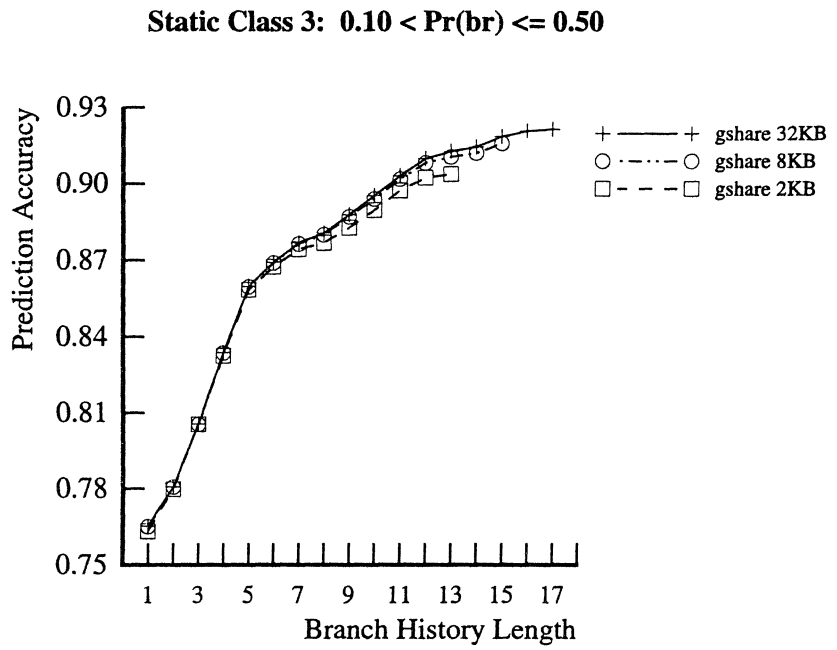

Fig. 7. Prediction accuracy of gshare on the mixed-direction branches.

direction branches only, the best branch history length is 3. Thus, hybrid branch predictors which use two different branch history lengths will outperform single-scheme predictors which use only one branch history length.

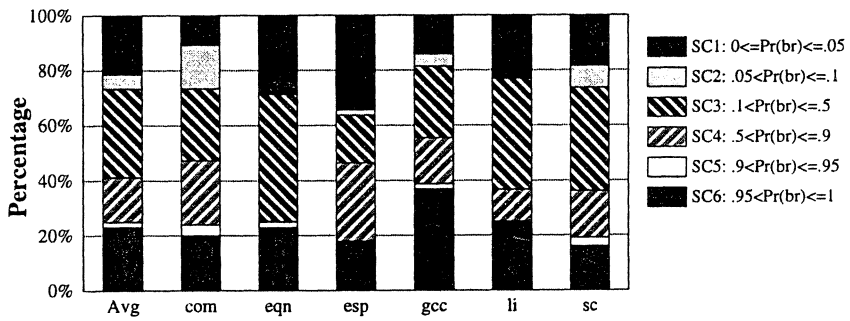

Fig. 8. Percentage of dynamic branches in each static class.

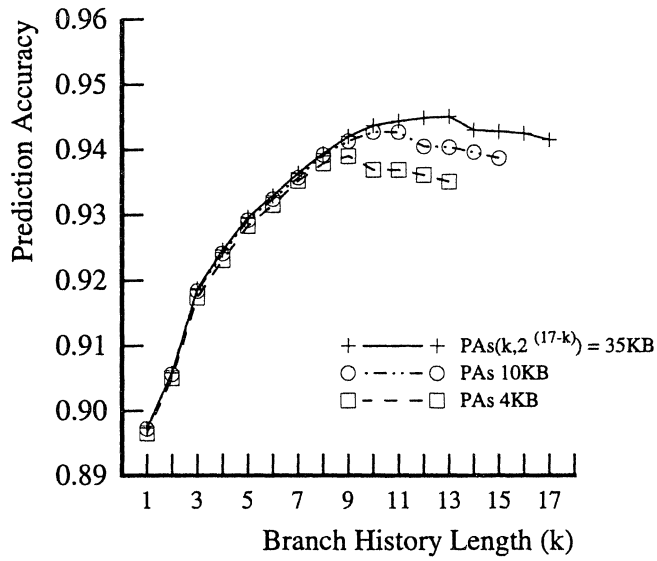

Fig. 9. Per-address history schemes with varying branch history lengths.

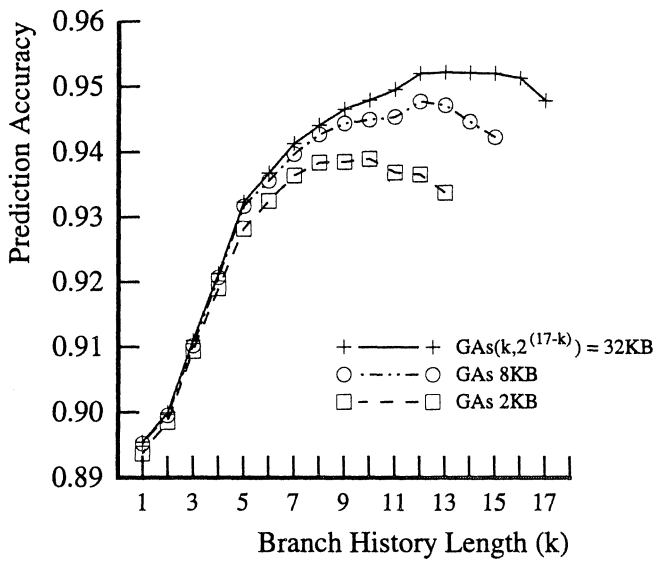

Fig. 10. Global history schemes with varying branch history lengths.

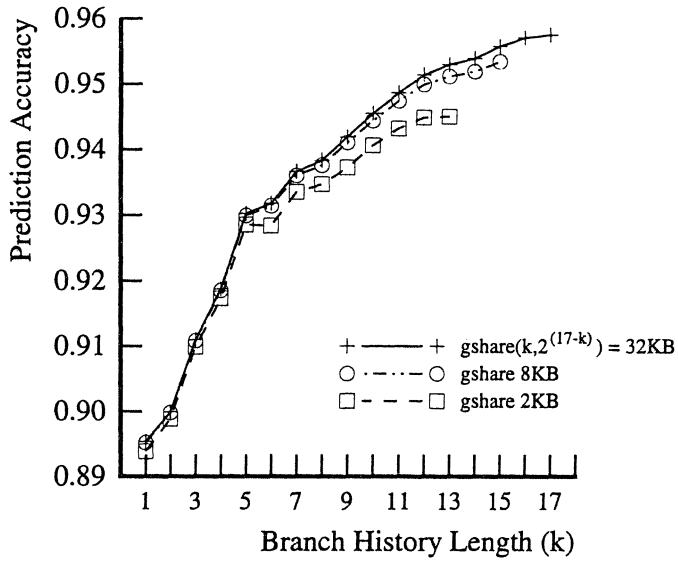

Fig. 11. Gshare with varying branch history lengths.

4.2. Combining the Advantages of Different Predictors

In this section, we first propose two hybrid branch predictors which statically select a single-scheme component predictor based on the aforementioned static classes. We then compare the performance of these hybrid predictors with McFarling's hybrid predictors, which dynamically determine the more accurate component predictor for each branch. Finally, we present another hybrid branch predictor design based on static classes that uses both compile-time and run-time information to select the most suitable component predictor for each branch.

4.2.1. Static Branch Predictor Selection

- *Gshare with a Static Predictor*
  Prediction accuracy can be increased by associating each branch class with the most suitable predictor for that class. To maximize the prediction accuracy obtained from a given hardware budget, we could use a simple and low-cost predictor for predictable branches and dedicate more resources to handle branches that are more difficult to predict.

If static predictors can accurately predict the mostly-one-direction branches, we can then use static predictors for these mostly-one-direction branches and dedicate all our hardware resources to predict the mixed-direction branches. In our experiment, we examine a hybrid predictor (PG + gshare) that uses the profile-guided predictor to statically predict SC1 and SC6 branches and the gshare predictor to dynamically predict the other branches. If a branch is not executed during the training run, then the gshare predictor is designated for predicting this branch during the testing run.

Figure 12 shows that PG + gshare outperforms the gshare, GAs, and PAs single-scheme predictors. To show how PG + gshare achieves a higher accuracy, Figs. 13 and 14 show the prediction accuracies of PG + gshare and gshare for the SC1 and SC3 branches respectively. For the SC1 branches, the profile-guided component in PG + gshare accurately predicts these branches because these branches are mostly not-taken. It achieves a significant performance advantage over the gshare predictor by itself because the gshare predictor suffers from PHT conflicts between the mixed-direction branches and the mostly-one-direction branches. Because the gshare component in PG + gshare predicts only the mixed-direction branches, it does not suffer from this conflict. The SC6 branches show a similar result.

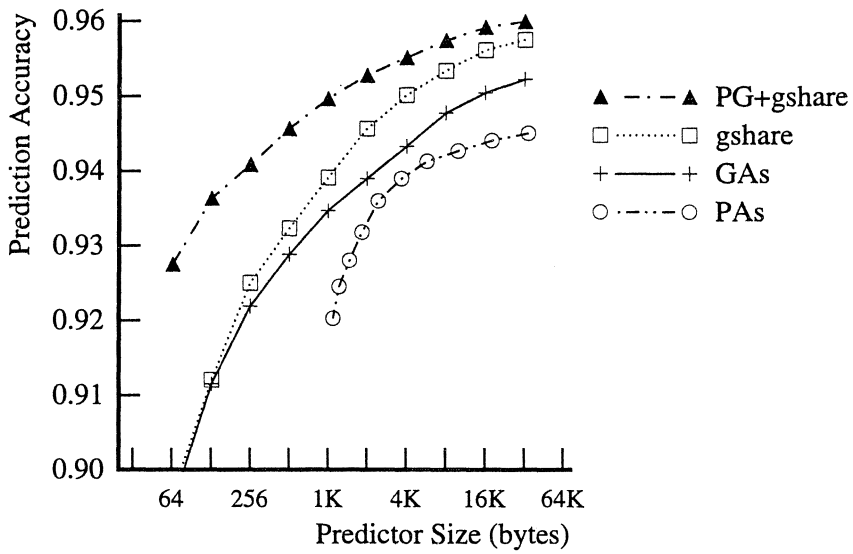

Fig. 12. Performance of the PG + gshare Hybrid Branch Predictor.

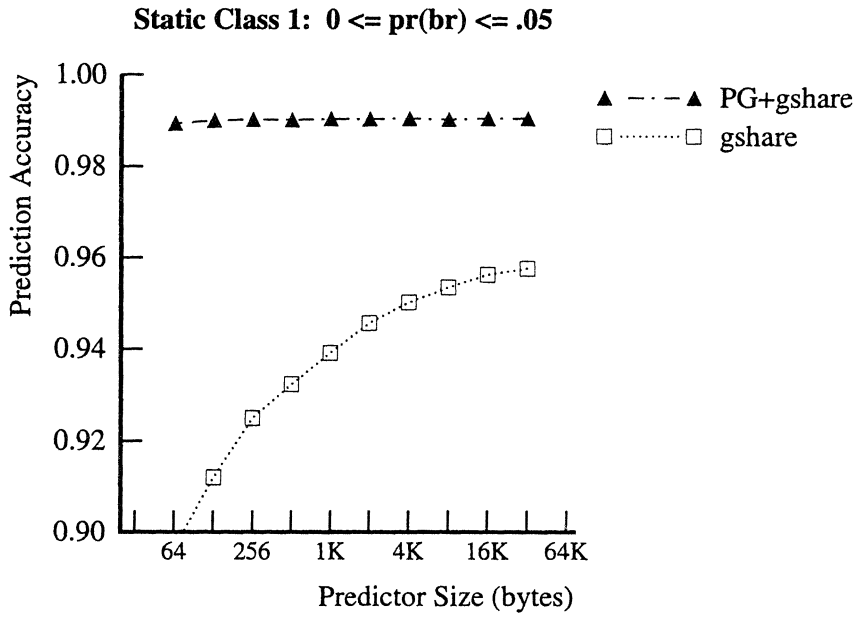

Fig. 13. Performance of PG + gshare and gshare on mostly not-taken branches.

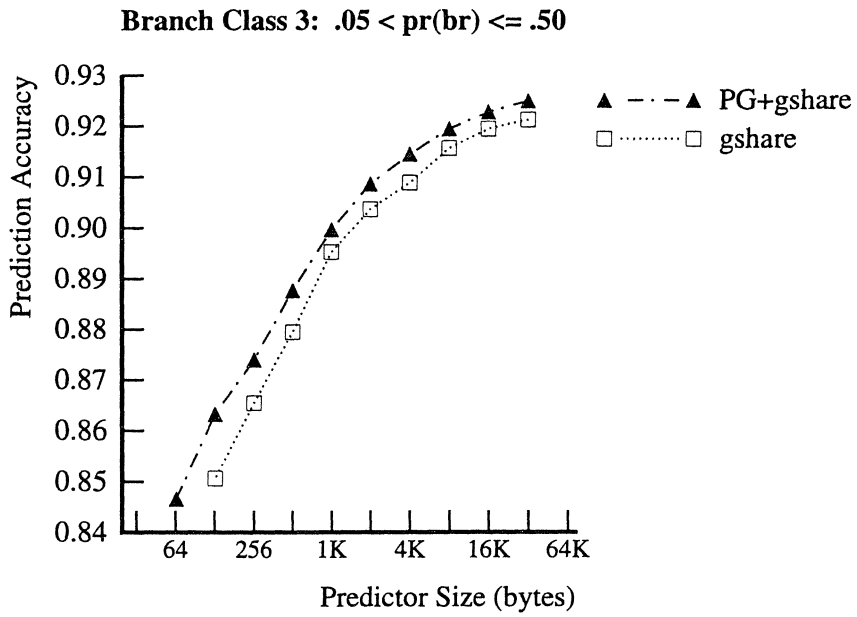

Fig. 14. Performance of PG + gshare and gshare on mixed-direction branches.

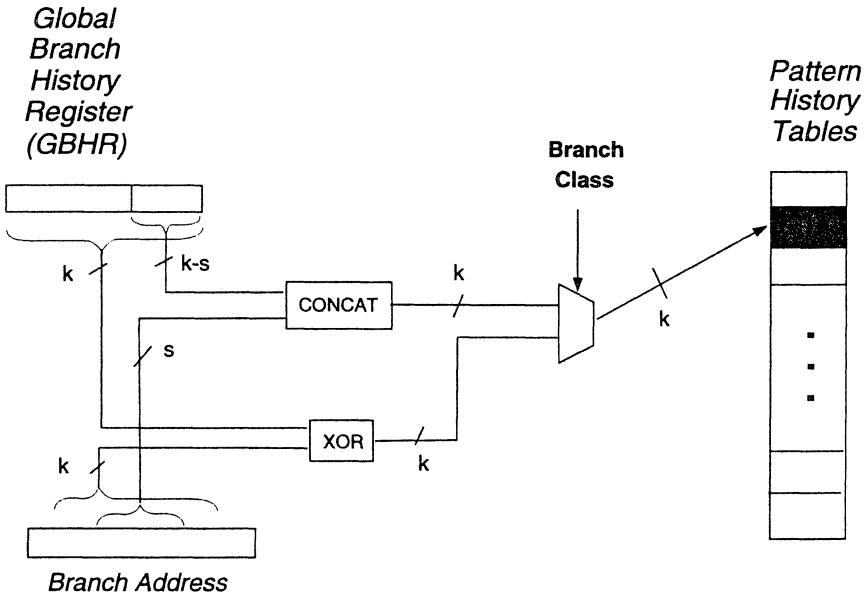

Fig. 15. Structure of the GAs + gshare Branch Predictor.

For the SC3 branches, PG + gshare achieves a slightly higher prediction accuracy than gshare. This is again because the gshare component in PG + gshare predicts only the mixed-direction branches, and so it does not suffer from the PHT contention between the mostly-one-direction branches and the mixed-direction branches found in the gshare single-scheme predictor.

- *GAs with Multiple Branch History Lengths*
  We have shown in Section 4.1 that the single-scheme predictors can not optimize prediction accuracy on both the mixed-direction and the mostly-one-direction branches. To increase prediction accuracy on both types of branches, we propose a new branch prediction scheme (GAs + gshare), which combines single-scheme predictors with different branch history lengths. It uses a short branch history for the mostly-one-direction branches and a long branch history for the mixed-direction branches. Figure 15 shows the structure of the GAs + gshare predictor. GAs + gshare uses the GAs predictor to handle the mostly-one-direction branches. For the mixed-direction branches, GAs + gshare uses the gshare predictor. The two prediction schemes share the same set of PHTs to save hardware cost. Because we are using fewer history bits for the mostly-one-direction

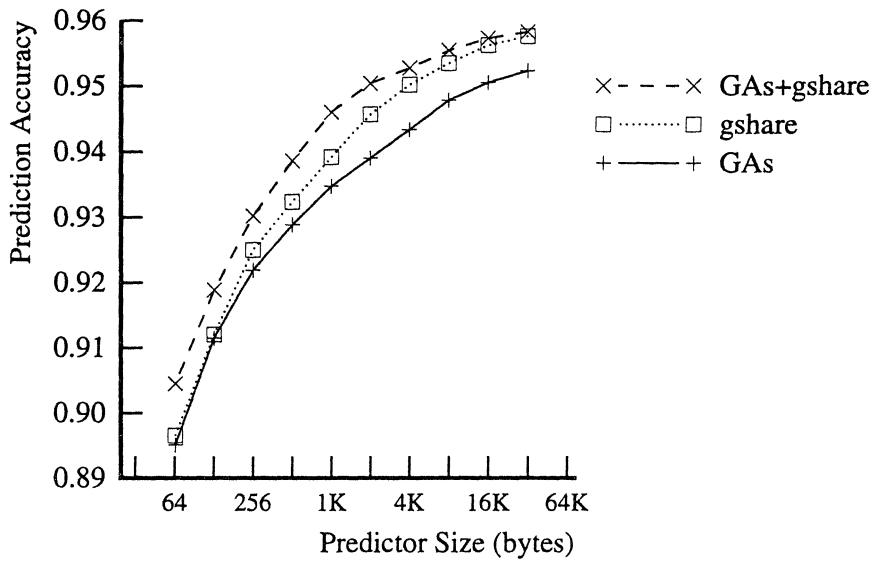

Fig. 16. Performance of the GAs + gshare Branch Predictor.

branches, there may not be enough history bits to identify each branch. To better identify each branch, the gshare scheme[2] exclusive-ORs the global history with the branch address to select the appropriate PHT entry. This is done to hash the frequent global history patterns to different PHT entries. Because the data set used to gather profile information is different from the one used in the actual testing run, some branches may only be executed during the testing run and, thus, have no profiled information. In this case, these branches are predicted with the gshare scheme.

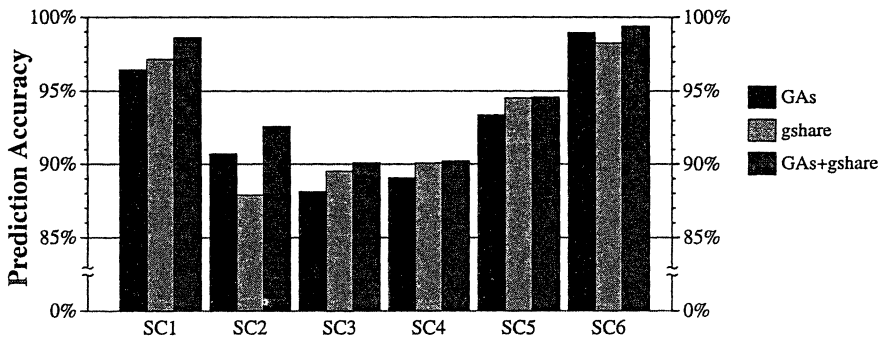

Fig. 17. Performance of 1K predictors on each static class.

Figure 16 compares the performance of GAs + gshare to the performance of the single-scheme predictors, GAs and gshare. This figure shows the best prediction accuracy of these predictors at each hardware cost. For all predictor sizes, GAs + gshare outperforms both GAs and gshare. Figure 17 shows the performance of these three predictors on each static class; each of these predictors has an implementation cost of 1K bytes. For the mixed-direction branches, GAs + gshare and gshare outperform GAs. For the same implementation cost, gshare uses longer branch history than GAs because the best GAs configuration for predicting all branches uses more than one PHT (as shown in Fig. 10). Thus, the gshare scheme can distinguish more execution states and is less likely to suffer from PHT conflicts. For the mostly-one-direction branches, GAs outperforms gshare on the SC2 and the SC6 branches. A shorter branch history in the GAs scheme results in fewer PHT conflicts. On the other hand, gshare outperforms GAs on the SC1 and the SC5 branches. Gshare, which XORs the branch history with the branch address to index into the PHT, more effectively uses the PHT than GAs for these branches. The most significant performance difference between GAs + gshare and gshare is on the SC1, SC2, and SC6 branches. By using a short history to predict the mostly-one-direction branches, GAs + gshare is able to achieve prediction accuracies that are 0.015, 0.047 and 0.010 higher than those of gshare on the SC1, SC2, and SC6 branches respectively. For the mixed-direction branches, the slight improvement in prediction accuracy is due to the fact that the mostly-one-direction branches are hashed into fewer PHT entries and result in fewer conflicts between the pattern histories of the mostly-one-direction branches and that of the mixed-direction branches.

4.2.2. Dynamic Branch Predictor Selection

Up to this point, we have statically selected the optimal predictor for each branch. Another method of combining branch predictors is to select the more suitable predictor dynamically. McFarling[2] proposed the use of 2-bit saturating up-down counters (2bC) to keep track of which predictor is doing better. The state of the counter is updated based on the algorithm of Table III.

Table III. Counter Update Rules

Predictor 1	Predictor 2	Update to Counter
Correct prediction	Correct prediction	No change
Correct prediction	Incorrect prediction	Increment
Incorrect prediction	Correct prediction	Decrement
Incorrect prediction	Incorrect prediction	No change

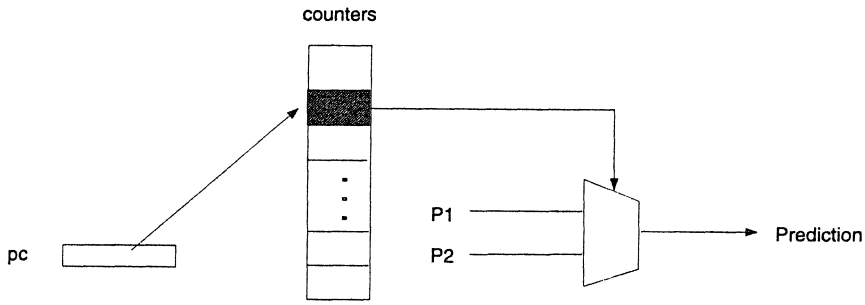

Fig. 18. Structure of a Hybrid Predictor (P1/P2) with Dynamic Predictor Selection.

In our study, we associate a counter with each entry in the fully associative branch address cache (BAC). When a branch is fetched, the corresponding counter found in the BAC is then used to determine which predictor to use, as shown in Fig. 18. Instead of associating counters with the BAC, McFarling used an array of counters for predictor selection, mapping each branch to a counter via the branch address. Associating counters with BAC (instead of using an array of counters) results in a slightly better prediction accuracy on the SPECint92 benchmarks.

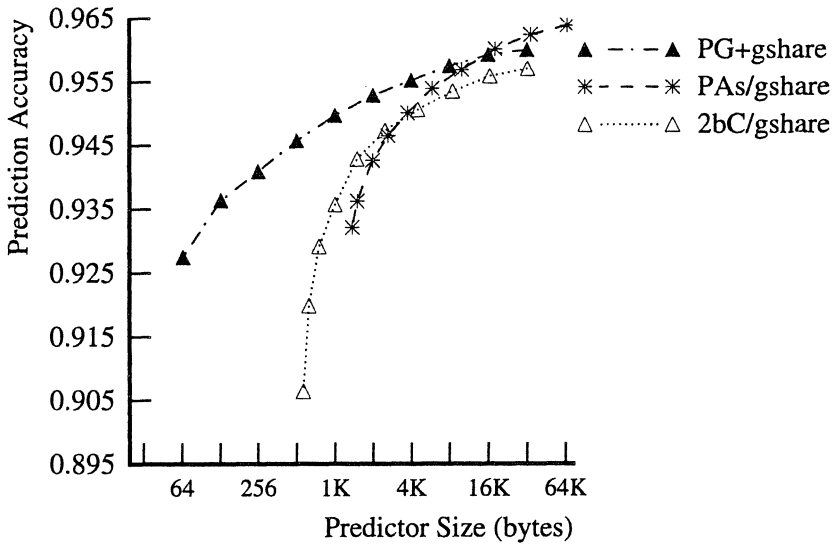

Fig. 19. Prediction accuracy of Hybrid Branch Predictors.

We simulated two different combinations of predictors: the 2-bit up-down counter predictor with gshare (2bC/gshare) and PAs with gshare (PAs/gshare). (Note: we use "+" to indicate static predictor selection and "/" for dynamic predictor selection.) Figure 19 compares the performance of these hybrid predictors with the static predictor selection scheme PG + gshare. The hardware costs for 2bC/gshare and PAs/gshare are estimated using the following equations:

$$\text{2bC/gshare}(k, p, a) = (a \times 2) + (a \times 2) + k + (p \times 2^k \times 2)$$

$$\text{PAs/gshare}(k, p, a) = (a \times 2) + (b \times k) + (p \times 2^k \times 2) + k + (p \times 2^k \times 2)$$

where a is the number of entries in the branch address cache, k is the history register length, p is the number of PHTs, and b is the number of entries in the branch history table. That is, the cost of the hybrid predictor is determined by summing the cost of the single-scheme predictors and the counters used to select the optimal predictor. As in McFarling's study, we only considered combining PAs and gshare configurations with the same implementation cost. Future studies will determine the best configuration for PAs/gshare.

For predictors smaller than 16K bytes, the PG + gshare scheme outperforms the PAs/gshare scheme. With PG + gshare and PAs/gshare of a similar size, the gshare in the PG + gshare scheme is approximately twice the size of either PAs or gshare in the PAs/gshare scheme. With an implementation cost below 16K bytes, the larger gshare was able to outperform the combination of the two smaller predictors. On the other hand, for predictors larger than 16K bytes, PAs/gshare is able to outperform PG + gshare. Because the benefits of a larger predictor diminishes as the size of the predictor increases, a combined predictor outperforms a larger single-scheme predictor.

PG + gshare outperforms the 2bC/gshare scheme at all predictor sizes. In this study, the 2bC/gshare predictor scheme uses a 1K-entry BAC and 1K 2-bit counters. Figure 20 shows how often gshare was used to make predictions in the 2bC/gshare scheme. The size of 2bC/gshare is increased by only increasing the gshare portion; the 2bC portion remains fixed at 256 bytes. With increasing 2bC/gshare size, the prediction accuracy of gshare increases and, thus, more branches are predicted using the gshare scheme. Figure 20 shows that the majority of predictions on the mostly not-taken branches are made by gshare. Since gshare, at a low implementation cost, is not as accurate as the profile-guided predictor, PG + gshare outperforms 2bC/gshare on the mostly-one-direction branches. Figure 20 also shows

that 40% of predictions on SC4 branches are made by the 2bC scheme. While 2bC can predict mostly-one-direction branches well, it is outperformed by gshare on the mixed-direction branches. Thus, PG + gshare also outperforms 2bC/gshare on the mixed-direction branches.

4.2.3. Static plus Dynamic Predictor Selection

In this section, we propose a new hybrid branch predictor that exploits the advantages of using both run-time and compile-time information to assist branch prediction. The PG + PAs/gshare scheme (see Fig. 21) uses the profile-guided predictor for the mostly-one-direction branches and the PAs/gshare scheme for the mixed-direction branches. Figure 22 shows the performance of PG + PAs/gshare. For predictors smaller than 4K bytes, the PG + gshare scheme provides the best performance of the schemes measured. On the other hand, for predictors larger than 4K bytes, the PG + PAs/gshare scheme outperforms all other predictors. For example, with a fixed implementation cost of 32K bytes, PG + PAs/gshare is able to achieve prediction accuracy of 96.4% on the SPEC integer

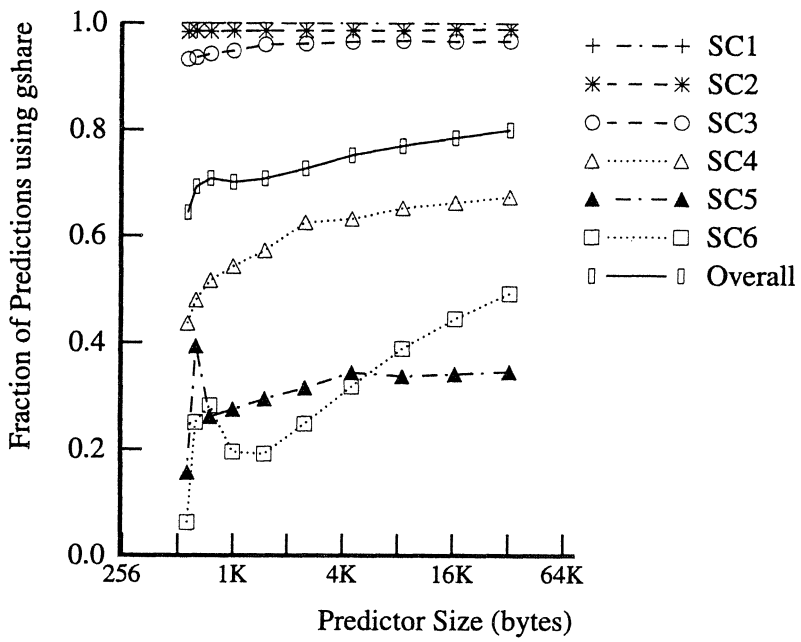

Fig. 20. Fraction of gshare usage in the 2bC/gshare scheme.

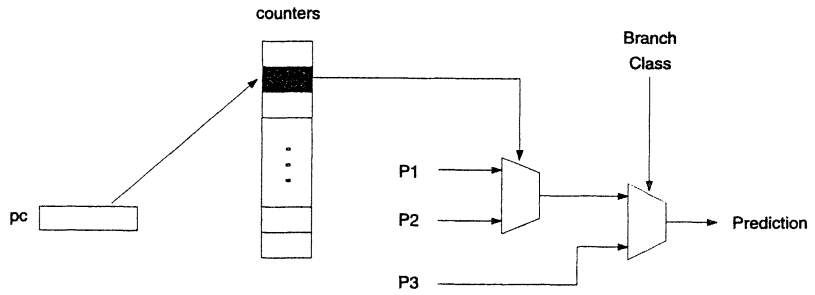

Fig. 21. Structure of a hybrid branch predictor (P3 + P1/P2) with static and dynamic selection.

benchmarks, as compared to 95.7% for gshare and 95.2% for GAs. For the SPEC92 benchmark gcc, which contains a particularly large number of branches, PG + PAs/gshare achieves prediction accuracy of 96.91%, as compared to 96.47% for the best previously proposed predictor (PAs/gshare).[2] These results for PG + PAs/gshare and PAs/gshare indicate that branch classification can be helpful in designing more accurate branch predictors.

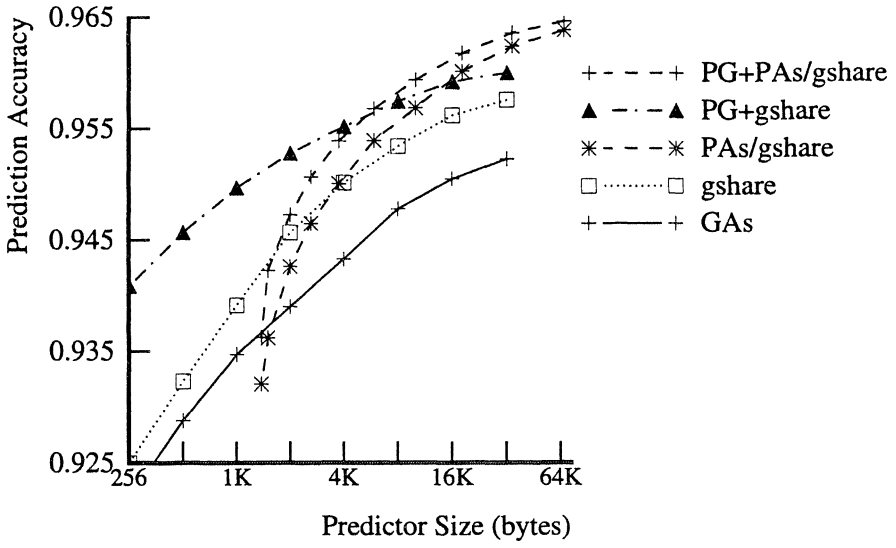

Fig. 22. Performance of hybrid branch predictor with both dynamic and static selection.

5. CONCLUSIONS

We have introduced branch classification as a means for analyzing single-scheme predictors in order to design better hybrid branch predictors. Comparing the prediction accuracies of different branch prediction schemes for different branch classes can be used to determine effective combinations of component predictors for a hybrid branch predictor. It also can be used to select the predictor which is most suitable for each branch. Thus, both compile time information and run time information can be used to enable the hybrid branch predictor to achieve higher accuracy.

To demonstrate the benefits of branch classification, we proposed a branch classification model that groups branches into classes based on their dynamic taken rates, during profiling. With this model, we showed that the mostly-one-direction branches are most accurately predicted by simple predictors and that the mixed-direction branches are most accurately predicted by complex predictors.

We then proposed two hybrid branch predictors, PG + gshare and GAs + gshare, that exploited this observation. The PG + gshare predictor used a simple profile-guided predictor for the mostly-one-direction branches and a gshare predictor for the mixed-direction branches. By adding the profile-guided component, the gshare component in the PG + gshare predictor could devote all its resources to predicting mixed-direction branches, resulting in a higher overall prediction accuracy. With an implementation cost of 32K bytes, PG + gshare achieved a 96.0% prediction accuracy, as compared to 95.7% for gshare by itself, reducing the miss rate by 7.5%. The GAs + gshare predictor used a GAs predictor with a short branch history for the mostly-one-direction branches and a gshare predictor with a long history for the mixed-direction branches. This hybrid predictor outperformed all other single-scheme predictors that consisted of one history length.

Our branch classification model was also used to construct a hybrid predictor that used McFarling's dynamically-selected hybrid predictor[2] as a component. This predictor, PG + PAs/gshare, was able to exploit both compile-time and run-time information in choosing a component predictor for each branch, improving the prediction accuracy of the resulting hybrid predictor. With a fixed implementation cost of 32K bytes, our proposed combination of a profile-guided predictor, PAs, and gshare achieved a prediction accuracy of 96.91% on gcc, a branch intensive benchmark, as compared to 96.47% for the best previously known predictor, reducing the miss rate by 12.5%. Using the branch penalty equation described in the introduction, for a processor with $C = 5$ and $r \times ipc = 0.9$, this improvement in prediction accuracy reduces branch penalty from 15.9% to 13.9%.

This paper examined only one model of branch classification. Further improvements in branch prediction accuracy can be achieved by considering other models. In addition, we must examine parameters such as the size/configuration of the component predictors and the different combinations of single-scheme predictors. We submit that there is still much headroom that can be attacked by exploring more of this design space.

6. ACKNOWLEDGMENTS

This paper is one result of our ongoing research in high performance computer implementation at the University of Michigan. The support of our industrial partners: Intel, NCR Corporation in its various incarnations, Motorola, Hewlett-Packard, and Scientific and Engineering Software is greatly appreciated. In addition, we wish to gratefully acknowledge the other members of our HPS research group for the stimulating environment they provide, in particular, Carlos Fuentes for his comments and suggestions on this work. We would also like to thank the reviewers for their helpful suggestions.

REFERENCES

1. P. M. Kogge, *The Architecture of Pipelined Computers*, McGraw-Hill, pp. 237–243, (1981).
2. S. McFarling, Combining branch predictors, WRL Technical Note TN-36, Digital Equipment Corporation, June 1993.
3. P. Chow and M. Horowitz, Architecture tradeoffs in the design of MIPS-X, *Proc. of the 14th Ann. Int. Symp. on Computer Architecture*, pp. 300–308 (June 1987).
4. C. Melear, The design of the 88000 RISC family, *IEEE MICRO*, pp. 26–38 (April 1989).
5. J. Emer and D. Clark, A characterization of processor performance in the VAX-11/780, *Proc. of the 11th Ann. Symp. on Computer Architecture*, pp. 301–310 (June 1984).
6. J. F. K. Lee and A. J. Smith, Branch prediction strategies and branch target buffer design, *IEEE Computer*, pp. 6–22 (January 1984).
7. J. E. Smith, A study of branch prediction strategies, *Proc. of the eighth Int. Symp. on Computer Architecture*, pp. 135–148 (June 1981).
8. J. A. Fisher and S. M. Freudenberger, Predicting conditional branch directions from previous runs of a program, *Proc. of the fifth Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, pp. 85–95 (1992).
9. J. A. DeRosa and H. M. Levy, An evaluation of branch architectures, *Proc. of the 14th Int. Symp. on Computer Architecture*, pp. 10–16 (June 1987).
10. S. McFarling and J. L. Hennessey, Reducing the cost of branches, *Proc. of the 13th Int. Symp. on Computer Architecture*, pp. 396–404 (June 1986).
11. T.-Y. Yeh and Y. N. Patt, Alternative implementations of two-level adaptive branch prediction, *Proc. of the 19th Ann. Int. Symp. on Computer Architecture*, pp. 124–135 (May 1992).
12. T.-Y. Yeh and Y. N. Patt, Two-level adaptive branch prediction, *Proc. of the 24th ACM/IEEE Int. Symp. on Microarchitecture*, pp. 51–61 (November 1991).

13. T.-Y. Yeh and Y. N. Patt, A comparison of dynamic branch predictors that use two levels of branch history, *Proc. of the 20th Ann. Int. Symp. on Computer Architecture*, pp. 257–266 (May 1993).
14. D. W. Hammerstrom and E. S. Davidson, Information content of CPU memory referencing behavior, *Proc. of the 15th Ann. Workshop on Microprogramming*, pp. 85–95 (October 1982).