OPTIMIZING HIGH PERFORMANCE DYNAMIC BRANCH PREDICTORS

by

Chih-Chieh Lee

A dissertation submitted in partial fulfillment of
the requirement for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
1998

To my parents, Yi-Chin and Dai-Li Kung.

ACKNOWLEDGEMENTS

First and foremost, I would like to sincerely thank my advisor, Professor Trevor Mudge, for his precious guidance, insight, and encouragement throughout the course of my research. His extensive proofreading also improved this dissertation significantly.

Second, I would like to thank Dr. Stuart Sechrest for his insight during my research study. Discussing with him has always been a source of valuable suggestions. I would also like to thank the other members of my dissertation committee, Professor Richard Brown, Professor Yale Patt and Professor Gary Tyson, for their helpful comments.

I am particularly indebted to my parents for their precious love and advice. I could always draw my strength from them to overcome hardships. I would also like to thank my brothers, Yu-Chin and Chun-Kai, and sister-in-law, Phoebe, for their warm encouragement.

I also wish to thank I-Cheng Chen, a research partner and a best friend of mine, with whom I have been studying and working for years. His inspiration and intelligence have helped me to break many research bottlenecks. I cherish the opportunity to have his collaboration and friendship.

I would like to express my gratitude to my senior colleagues, Richard Uhlig and David Nagle, who helped me establish the early stage of my research work. I would also like to thank my colleagues and friends in the PUMA research group and the Advanced Computer Architecture Lab. of Michigan. Working with such bright minds has made my doctoral study in Michigan enjoyable and memorable.

TABLE OF CONTENTS

LIST OF TABLES

Table

LIST OF FIGURES

CHAPTER 1

Introduction

The importance of accurate branch prediction to future processors has been widely noted. The correct prediction of conditional branch outcomes can help avoid pipeline bubbles and the attendant loss in performance. As more instruction level parallelism is exploited in high-performance microprocessors, the requirement of very accurate branch prediction has taken on a central importance.

This dissertation addresses the branch prediction problem by providing a study of high-performance dynamic branch prediction schemes. In particular, two classes of high-performance dynamic branch predictors are studied. They are distinguished by the type of information about previous branch outcomes that they employ: global outcome history and per-address outcome history. The purpose of this dissertation is to identify the design decisions that can improve performance for these two kinds of dynamic branch predictors, given that they will operate in a realistic computing environment. The contributions of this dissertation are as follows,

1. A general performance picture is provided of the branch prediction schemes that exploit global history information. It can be used to select the optimal design for these dynamic branch predictors (Chapter 4). The advantages and disadvantages of using global history information are also fully studied. Based

on the study, two design criteria for global history predictors are proposed. They are used to highlight a performance bottleneck common among global history schemes.

2. A new predictor, the bi-mode scheme, is proposed (Chapter 5). The bi-mode scheme meets both the design criteria referred to above, allowing it to provide significant performance improvement over existing schemes at modest cost. A detailed comparison between the bi-mode scheme and several variations of the global history schemes is also presented.

3. Branch prediction schemes that exploit per-address outcome history information are examined (Chapter 6). A general performance picture is provided to select the optimal design for these dynamic branch predictors. The pros and cons of these predictors and the role of adaptivity are studied. Based on this study, a simple but competitive per-address history scheme that uses algorithm-derived static-trained predictors is presented. This static-trained predictor is a cost-effective alternative to the dynamic branch predictors.

In all the studies mentioned above, a diverse set of fifteen popular benchmarks are used to represent a wide range of programs. These are fairly large programs, and they allow us to rectify some earlier misunderstandings about dynamic branch prediction that was based on small programs. They also illustrate the impact of growing program sizes on the dynamic branch predictor design [52].

We conclude this chapter with an introduction to the branch problem and a brief review of current high-performance dynamic branch predictors.

1.1 The branch problem

The sequential programming paradigm has been the predominant model of program execution. It requires that each instruction execute one at a time in sequential order. This paradigm places too stringent a constraint on computation because many instructions are actually independent of each other. There has been extensive research into architectures that relax this constraint, because such relaxation allows concurrent execution of independent instructions and hence faster compute times. The limits to this parallelism are set by the obvious requirement that their execution results must agree with the results of the sequential version.

In the sequential programming paradigm, branch instructions play an important role. The purpose of the branch is to redirect the instruction stream based on the current state of the computation — it is this feature that distinguishes computers from mere calculators and gives them their "universal computing" capability.

Unfortunately, branches can become a severe performance problem for the machine designer when instruction level parallelism (ILP) is also being employed to speed execution. This is because instructions following a branch may be started before the branch is resolved, and may have to be discarded depending on the branch outcome. This period before the branch is resolved may cause significant performance loss if no useful work is carried out.

In very simple machines, where only one instruction is fetched and executed at a time, branches are not a problem because the machine examines branches and makes any necessary instruction stream change within the time allotted to the instruction. However, the performance of such a machine is inherently slow because only one of the main CPU components is busy at any time. To improve performance, techniques were invented, such as pipelining, to keep every unit in the microprocessor busy. Ideally, in a single pipeline machine, during each cycle there is always one instruction being fetched, one instruction being executed and one instruction being completed. However, pipelining, like other types of ILP, can result in wasted work during branch resolution. These pipeline bubbles are a source of inefficiency.

As VLSI technology advances, more transistors can be integrated onto the chip, allowing microarchitectural designs with higher levels of ILP. The trend is thus to increase branch resolution time, resulting in the potential for even greater losses from branches. A key ingredient to realizing the full benefit of higher levels of VLSI integration and speed is being able to handle branches successfully. Consequently, there have been quite a few proposals, many of which will be reviewed in Chapter 2. Among these proposals, the most widely used are those that employ a dynamic branch prediction to select the path for execution before a branch is resolved. They are popular because dynamic branch prediction requires little, if any, modification to the instruction set, which is critical if binary compatibility is a concern. Furthermore, dynamic branch prediction achieves better performance than most other approaches [28]. However, there is still room for improvement as we will show.

Dynamic branch prediction is not the only way to predict branch outcomes, but it has out-performed most other prediction schemes, such as static prediction, e.g., compiler hints. Dynamic branch prediction can be better because it captures branch behavior at run time. In the early 1980's, a set of two-bit saturating counters, each assigned to a branch instruction, was shown to achieve good prediction accuracy for many programs [29, 46].

Recently, as processors are more aggressively exploiting ILP, interest in the design of branch prediction hardware has intensified. Many of the latest designs have sought to exploit information from multiple distinct branches for the prediction of each new branch instance. One of these techniques that has shown great promise is *two-level dynamic branch prediction.*

1.2 Two-level dynamic branch predictors

Branch outcomes are not merely random results; the behavior of a branch depends heavily on its own past behavior or the behavior of its neighboring branches. Exploiting this correlation between branches can improve prediction accuracy. There has been research work proposed to utilize this correlation [29, 46], with, perhaps, the most significant development being the invention of the two-level dynamic branch predictors. The earliest reference to two-level schemes appears to be a DEC patent first filed in 1990 [4]. Its second-level table was fixed. This idea was taken much further in the work of Yeh and Patt [56, 57], where adaptivity was added to the second level and a per-address version was proposed.

Theoretically, the two-level schemes approximate a set of Markov predictors [15]. These have been used successfully in the field of data compression, where accurate prediction of future characters in the data stream is an integral part of the compression algorithms. A Markov predictor estimates the likelihood of a particular character occurring by counting the frequency of each character that follows various character patterns. It then predicts the next character that follows a particular pattern as the most frequent character that has been observed to follow that pattern [2]. Predicting branch outcomes is actually a simplified version of predicting characters, where there are only two characters, "taken" and "not-taken." Therefore, the techniques developed in Markov predictors, or other prediction methods for that matter, are applicable to branch prediction.

The implementation of dynamic branch predictors has stringent restrictions with respect to on-chip area and access time. In microprocessors, the branch prediction unit needs to be located close to the CPU because it is accessed and updated very frequently. Clearly it must be on the same chip as the CPU. In the past, the cost of the on-chip area was an obstacle. This obstacle is disappearing as more and more transistors are integrated on to chips; however, the predictor access time is still a major limitation. In a high performance microprocessor, branch prediction must be completed in just one or two CPU cycles so that instruction fetching will not be stalled. Consequently, algorithms with the complexity of Markov predictors are generally not practical for branch prediction.

The two-level branch predictor, however, provides a simplified but effective form of the Markov predictor. The two-level predictor records the branch outcome history in registers, counts the number of taken branches for each possible history pattern with a simple two-bit saturating counter, and then makes prediction based on the state of the counter.

Figure 1.1 presents a model of a generic two-level predictor. It consists of two major data structures, one at each level. The first-level table contains shift registers to record the branch outcome history. This table is referred to as the branch history table (BHT) and the shift registers as branch history registers (BHRs) in Yeh and Patt [57]. The second-level consists of a number of finite state machines, typically two-bit saturating counters; each of the counters is used to count the number of taken branches for a

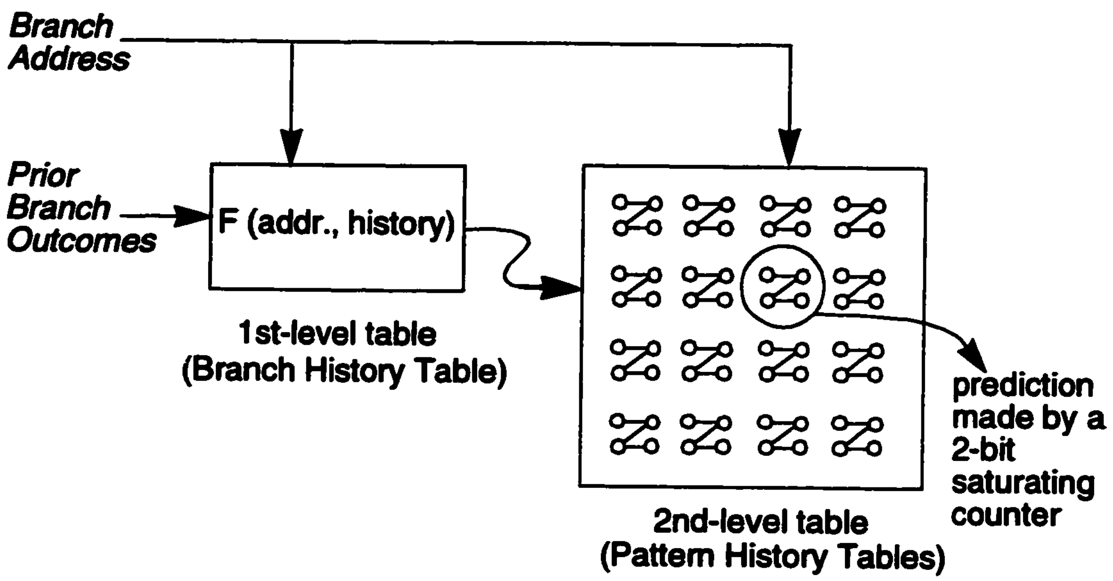

Branch Address
Prior Branch Outcomes
F (addr., history)
1st-level table (Branch History Table)
2nd-level table (Pattern History Tables)
prediction made by a 2-bit saturating counter

Figure 1.1: A model for a generic two-level dynamic branch predictor

particular history pattern. It is also possible for the second-level table to have more than one column. For multiple-column configurations, each column is referred to as a pattern history table (PHT).

As shown in the figure, to make a prediction, a state machine is first selected by column and row indices. The column index is usually the branch address from a program counter (PC), and the row index is formed in the first-level table, which implements a function of the branch address and branch outcome history. As mentioned above, this first-level table contains BHRs to record the branch outcome history. When a branch is resolved, the branch outcome will be shifted into a BHR. If the there is only one BHR, all branch outcomes will be shifted in this BHR, and a *global history* is obtained. On the other hand, if there are enough BHRs in the table, separate BHRs can be kept for each static branch and a *per-address history*, or *self-history*, is obtained.

When a state machine is selected, the state of this machine determines the prediction, while the actual branch outcome determines the next state for the selected state machine. If two-bit saturating counters are employed, the value of the selected counter will be increased by one when the outcome is taken and the value will be decreased by one when the outcome is not-taken. Since each two-bit counter can have only four values, the counter will not be increased if its value is already a maximum or will not be decreased if it is a minimum. This saturation is an important simplification to the Markov predictor, which predictions based on saturating counters emphasize the most recent outcomes rather than examining the entire history of the pattern.

If there is no limitation on area and access time, it is possible to build a two-level predictor with a very large table of counters. Each distinct conditional branch would be given a separate column in the table, with the rows representing conditions under which that branch's outcome is likely to vary. For example, separate counters may be kept for the various combinations of outcomes of past executions of the branch, or the outcomes of correlated branches. It would be the responsibility of the BHT to record this history and from it to determine the appropriate row for each branch instance. The cost of the BHT could vary considerably, depending upon the selection scheme that we chose to implement.

In practice, resource constraints make it necessary to determine a cost-effective combination of row-selection strategy and predictor table organization. Many possibilities have been suggested. Yeh and Patt [59] introduced a taxonomy for such two-level schemes, coding them with three letters. In the context of the model shown in Figure 1.1, the first letter states whether the row selection is based upon history kept globally (G), kept for a set of addresses (S), or kept for individual addresses (P). The second letter indicates whether the predictor table contains an adaptive state machine, such as a two-bit saturating counter (A), or a fixed prediction (S). The third letter indicates whether the table has a single (global) column used for all addresses (g), a set of columns indexed by bits extracted from the address (s), or a separate column for every address (p). Strictly speaking, schemes with coding letters, P or p, are not possible to implement in the real world, because there are usually many more static branches in a program than the maximum number of columns which can be realistically implemented in the BHT. Nevertheless, by convention we will ignore this subtle difference, and still use P and p to refer to per-address schemes, even though overlaps between branches do exist. Typically, only the low order bits of the address are used, so there is a possibility of several addresses mapping to one BHR even in "per-address" schemes. We will refer to this as aliasing later on.

In this dissertation, the schemes that exploit global and per-address history information are studied. The global and per-address history schemes represent two extreme ends of the configuration spectrum of two-level schemes, and are ideal points to contrast and compare. The per-set history scheme falls in the middle of the spectrum, and previous study has shown that the per-set history scheme is no better than the other two-level schemes [59].

1.3 Impact of software development on the branch predictors

The size of programs has continually increased, as observed by Uhlig *et al.* [52]. In their study, they showed that code bloat can degrade the performance of instruction cache significantly. While this is not a concern for small programs, such as SPEC benchmarks, larger programs may also have more static branches to exercise, causing degradation of the performance of the branch predictor. In the past, there have been quite a few ideas and designs for branch prediction schemes using correlation. Unfortunately, they primarily used small benchmarks, such as SPEC89 and SPEC92 for performance evaluation. Because most of these benchmarks have tiny numbers of static branches to exercise, these studies were unable to indicate the appropriate design for a more realistic and larger workload space.

This dissertation includes more realistic programs to evaluate the performance for the dynamic branch predictor. We conduct our experiments for dynamic branch predictors under a wide range of benchmarks. Our results explains the sensitivity of predictors to variations in resources allocated, design, and workload. We evaluate the predictors using programs from SPEC CINT95, the latest version of SPEC integer benchmarks, IBS-Ultrix, a set of widely used system workload traces, and *sql95*, a database system workload trace distributed by Digital Equipment Corp. By exhaustively searching the design space and including more benchmarks, our results can present a "global picture" of the design space and describe the trade-offs of using branch outcome history for the branch predictors.

1.4 Dissertation organization

The dissertation is organized as follows. Chapter 2 discusses previous research work. Chapter 3 describes the methodology and discusses intrinsic differences between benchmarks used in this work and earlier studies. Chapter 4 examines the global history branch prediction schemes and analyzes the effects of correlation. Chapter 5 proposes a new scheme, the bi-mode scheme, that improves upon the global history schemes, comparing it favorably with other more recent variants of global history schemes. Chapter 6 presents the study for the per-address history schemes. Chapter 7 concludes with a summary of our work and proposes some future directions.

CHAPTER 2

Related Work

The branch problem has been known for a long time. As early as the 1970's, researchers identified the severe performance degradation that could occur with branches. In particular, Flynn estimated performance degradation due to branches for single-instruction-stream machines [20]. Research work by Tjaden *et al.* and Riseman *et al.* showed that if there is no branch problem, it is possible to execute fifty instructions per cycle in scientific codes, but if instruction fetch stops due to branches, as few as two instructions may be that can be executed at a time [51, 42]. More recently, Butler *et al.* have examined the instruction level parallelism (ILP) for the SPEC89 benchmarks, and showed that there exists potentially high ILP in the benchmarks, but branches can be a serious bottleneck to achieving that potential [5]. As machines more aggressively exploit ILP, the branch problem grows, consequently, it is receiving more attention. In this chapter, some popular and interesting approaches to solving the problem are discussed. Since branch prediction dominates the methods of handling branches, it will be the focus of this chapter.

2.1 Handling branch problems

Currently, the most popular method to reduce performance loss due to branches is speculation with branch prediction. Branch prediction refers to the technique of predicting which one of the alternative execution paths following a branch will be executed before the branch is resolved. This speculation allows machines to continue fetching instructions along the predicted path, instead of waiting for branch results. The branch prediction, however, needs to be accurate, or all the speculated instructions after the mispredicted branch must be discarded.

For most of current instruction set architectures (ISAs), there are two kinds of branches, conditional and unconditional. A conditional branch usually has two paths to take, either the fall-through path or the target path. The path taken depends on the branch outcome, not-taken or taken, respectively. Because the target address is usually specified in the branch instruction and can be computed early, the major problem with conditional branches is to determine the outcome of the condition they test. Conditional branches occur frequently, usually accounting for about 15% of total dynamic instructions, so most branch prediction methods are used to predict outcomes for conditional branches. Unconditional branches always redirect the instruction stream, i.e., their outcomes are always taken, so machines need to jump to a target whenever such a branch is executed. Predicting their outcomes is not an issue, but predicting their targets can be problematic if the targets are not explicit in the instructions.

Unconditional branches fall into two groups: direct and indirect branches. A direct unconditional branch specifies its target in the branch instruction and therefore has only one target to jump to. Direct unconditional branches can be effectively predicted by its last-time target, or easily resolved in an early pipeline stage. An indirect unconditional branch specifies its target in a register which may not be accessed until a later pipeline stage, such as the execution stage. Therefore, indirect unconditional branches need their targets predicting. This can be difficult because their targets can be modified during execution. Fortunately, indirect unconditional branches appear quite infrequently (less than 1% in the SPEC CINT95 benchmarks [12]), and thus are not the primary concern compared to the conditional branches. A branch target buffer (BTB) [29] combined with a return-address stack [27] is a popular way to predict targets for unconditional branches. The BTB predicts targets for unconditional branches by their last-time targets and the return-address stack predicts for return-from-subroutine branches by their caller's addresses pushed on the stack. Chang *et al.* has recently proposed a target prediction method based on the history of branches to improve the prediction accuracy [12].

Branches construct control flows for programs; in other words, the execution of the instructions following a branch is determined by the branch outcome. However, analyses on control dependency have shown that not all instructions after a branch need to wait for the branch execution result [28, 53]. These studies examine some possible ways of identifying the *true* control dependency so that the unnecessary stalls due to unresolved branches can be removed. This is similar to the distinction between the true data dependency (read-after-write) and false dependencies (write-after-write and write-after-read) in data dependency analysis. In that area, researchers have realized that the false dependencies can and should be removed in order to speed up the computation, while true dependencies pose inherent performance limitations.

However, it is also shown by Lam *et al.* [28] that performance gained by eliminating false control dependency alone is small. This is because, without other supporting techniques, such as branch prediction, the parallelism is primarily limited by the constraint that branches must be executed in order. Since conditional branches occur so frequently in most of the programs, executing one branch at a time is a serious bottleneck. The same study has suggested that if minimizing control dependency is combined with other techniques, such as branch prediction, it can enhance the performance significantly.

When microprocessors aggressively speculate along the instruction stream, with many branches being predicted but yet unresolved, the probability of wrong speculation rises. For example, assume the averaged prediction accuracy of a predictor is 80%, then the accuracy for the speculated path is roughly $(80\%)^n$, where n is the number of outstanding branches in a machine. If n is 5, the likelihood that the speculated path is the correct one falls to 33% by the time the sixth branch is encountered. Therefore, rather than continuing to fetch instructions along the very deep speculative execution path, researchers have proposed multiple-path instruction execution, or dual-path execution after a conditional branch, which will fetch along both paths after a branch. Because the compute engine normally cannot resolve branches more quickly than it can fetch new instructions, it is possible that dual-path fetching can overload the machine and memory bandwidth. Therefore, some simplified methods have been proposed; these include disjoint eager execution and limited dual path execution [54, 31].

The disjoint eager execution assigns a cumulative prediction accuracy for each path after an unresolved branch, sorts paths of all unresolved branches by their cumulative accuracies, and then fetches the path that has the highest cumulative accuracy. For example, if the averaged prediction accuracy of a branch predictor is 80%, after 8 branches along the speculative path have been predicted but not yet resolved, the cumulative prediction accuracy for the 9th branch can be no more than 16.7%. This cumulative prediction accuracy is lower than the one for the un-taken path of the first unresolved branch, which is 20%. In this case, the disjoint eager execution scheme will fetch the path that has 20% prediction accuracy, instead of speculating on the 9th branch.

Theoretically, this disjoint eager execution can deliver the optimal performance, but practically it has several problems. First, it needs a fast way to determine the cumulative prediction accuracy, usually one computation per CPU cycle. The example above requires multiplication of prediction accuracy, which is prohibitive in high speed microprocessors. Second, the prediction accuracy is not equal for all the branches; some branches are easy to predict, while others are not. For those branches that are easy to predict, there is no reason to fetch the un-predicted paths, because the branch predictor has strong confidence on its prediction. On the other hand, for those hard-to-predict branches, both paths need to be fetched because of the higher chance of misprediction. Therefore, a feasible scheme, limited dual path execution, has been proposed, which examines the confidence of prediction accuracy, and fetches both paths for those branches that have low confidence on prediction. Nevertheless, this scheme may waste instruction fetch bandwidth if it encounters many highly confident branches, because the extra fetch unit for the other path is idle.

There have been numerous approaches proposed to solve the branch problem through code transformation by compilers. These proposals usually achieve goals by eliminating branches, or enlarging basic blocks. This kind of techniques may provide significant improvement, but they also require some degrees of modification to the ISAs, which can complicate the matter of backward code compatibility. A typical example of such techniques is guarded instructions, or predicated instructions [16, 25, 41, 40]. This technique is designed to eliminate branches by conditionally executing or committing instructions dynamically. Instructions to be guarded are associated with a boolean valued expression, or a guard expression. Guarded instructions will always be fetched when the instruction pointer reaches them. However, if the guard expression evaluates to false, the results of guarded instructions will not be committed; in other words, the guarded instructions are converted to useless instructions dynamically when the guarding condition is not true. By conditionally executing (or committing) instructions based on conditional values, compilers can eliminate a branch by combining instructions from alternative paths after the branch. With this branch elimination, compilers can form a larger basic block to enhance its code scheduling capability. However, guarding can adversely impact on instruction fetch bandwidth because some instructions are eventually discarded after being fetched. Moreover, if a branch is easy to predict, the guarded instructions on the wrong-path may waste the computing resources most of time. Recent studies have examined combining guarded instructions and branch prediction to achieve better performance [50, 32, 13]. A key idea in these studies is to use guards to eliminate branches that are difficult to predict, while predicting other branches with dynamic branch predictors.

Loop unrolling is another technique to eliminate branches through code transformations. Loop unrolling may combine several successive iterations of a loop to form a larger loop, thus reducing the number of dynamic branches. Like the guarded instructions, loop unrolling can form larger basic blocks to enhance compilers' scheduling capability. This technique is useful for the scientific codes where there are many loops with a fixed (compile-time) number of iterations, but the advantages of loop unrolling are limited for integer programs because integer programs usually have more if-then-else constructs and few loops with fixed iteration counts.

In addition to branch prediction and elimination, the branch problem can be alleviated by reducing the penalty of resuming correct execution due to branch misprediction. Bondi *et al.* proposed a misprediction recovery cache (MRC) to cache the instructions on the mispredicted path that have been fetched and decoded due to a wrong speculation [3]. If the same branch is mispredicted again and the mispredicted path is stored in the MRC, this MRC may help resuming the correct execution more quickly because it can supply the decoded instructions on the correct path to the execution engine.

2.2 Predicting for branch outcomes

Predicting for branch outcomes can be done statically or dynamically. A static branch prediction is usually made at compiling time. The compiler can employ some heuristics or profiling to determine the most likely direction that a static branch will take, and then use the most likely direction as the prediction for the branch and encode it into the branch instruction. When the branch is fetched at run time, machines will predict the branch outcome based on the compiler's hint.

Static prediction has proved to be quite effective [19, 1], it has been frequently employed in assisting code scheduling and optimization for compilers, such as trace scheduling [18]. However, its prediction accuracy is lower than the dynamic approaches [46, 29, 56, 55]. One major reason is that dynamic approaches can adapt to the run-time behavior of branches due to different phases of program execution, while static branch prediction has to fix a prediction value for each static branch that is time invariant. To moderate this problem, Young *et al.* have proposed a static prediction method that exploits branch correlation [60]. They examined the branches behaviors for various history patterns at compile time. If a branch behaves differently under different history patterns, the basic block containing the branch will be replicated in the program and each copy of the branch will be assigned a different prediction value based on the history pattern. This technique can improve the static branch prediction, but is still inferior to the dynamic approaches. Besides, a shortcoming of this technique is that it may increase the program size significantly.

Dynamic branch prediction schemes make prediction based on the information collected at run time. An early scheme is the two-bit saturating counter proposed by Smith [46], which, in an ideal situation, assigns a unique counter to each static branch to record the recent behavior of that branch. The use of a two-bit saturating counter means that the prediction is based only on recent branch behaviors. Nevertheless, performance is quite good and the two-bit scheme provides an adequate amount of damping against false predictions, best typified by loop exits. Moreover, Nair has presented empirical results to show that the two-bit saturating counter is the best among finite state machines of two bits [36].

Smith's two-bit saturating counter scheme classifies branch outcome information only according to the addresses of static branches, which can limit the prediction accuracy because much of the correlation information between successive executions of a branch is not captured. To account for this auto-correlation, Yeh and Patt proposed their first two-level dynamic prediction scheme, known as the per-address history scheme in 1991 [56]. The per-address history scheme associates a two-bit counter to each possible per-address outcome history pattern instead of each static branch site in order to exploit auto-correlation information. This pushed prediction accuracy above 90% for some SPEC89 benchmarks.

Shortly after the per-address proposal, another form of two-level dynamic prediction schemes was proposed [38, 57]. Although, an earlier proposal by DEC employees was first filed in 1990 [4]. It was designed to exploit inter-branch correlation, i.e. history from branches in the vicinity of the branch. These correlated prediction schemes rely on the global history of branch behavior rather than the self (per-address) history, and thus are referred to as global history schemes. These global history schemes are simple and can achieve similar performance to the per-address two-level schemes. However, these global history schemes show some performance limitation due to interference in the second-level table. Some variations of the global history scheme have also been suggested, such as gshare and path-based correlation schemes [34, 37]. In particular, the gshare scheme xors global history bits with branch address bits to randomize global history patterns in order to evenly distribute counter usage and hence to lessen the interference for the second-level table that can occur with the standard correlated prediction scheme. The path-based scheme employs partial address bits of previous dynamic branches to represent the execution path leading up to a particular branch and uses this, instead of the global history outcomes, to index into the second-level table. The path-based scheme attempts to distinguish the static branches that produce the same global history pattern, in a manner suggested by the static correlated prediction work by Young *et al.* [60]. Both the gshare and path-based schemes have been shown to perform slightly better than the standard global history schemes for SPEC benchmarks

[45].

All of the studies mentioned above have a common shortcoming: they only used small benchmarks, such as SPEC89 and SPEC92, when they evaluated predictor performance. Because of the sizes of the benchmarks, meaningful conclusions about the effects of large footprint programs cannot be evaluated. In particular, appropriate resource allocation for prediction schemes in large footprint programs cannot be determined. One of our points is that the large footprint programs that we use for benchmarks are more representative of actual workloads that present-day systems experience.

To examine global history prediction schemes, Young, *et al.* propose a framework that categorizes branch prediction schemes by the way in which they partition dynamic branches and by the kind of predictor that they use. With their framework, they also introduced several concepts, such as aliasing (or interference) and biasing, that are important for discussing performance analysis. Their work focuses on distinguishing their proposed static correlation scheme from dynamic prediction schemes. However, their results are deduced from SPEC92, thus suffering from the shortcomings inherent in small benchmarks. In this dissertation work, we will extend their ideas of aliasing and bias to examine dynamic branch predictors.

The work by Talcott *et al.* also studied the effect of aliasing in the second-level table for the global history schemes [49]. They found that interference in the second-level table degrades performance. However, they only examined schemes with 1,024 counters in the second-level table, which is not sufficient to reveal performance trends for the global history schemes they studied. Most of their conclusions only apply to small benchmarks, although they did note that the sole large footprint program in their benchmarks, *gcc*, shows characteristics distinctive from the rest.

As it became clear that global history schemes suffer from interference, several schemes have been proposed recently to overcome the interference problem, including filtering schemes, skewed predictors, and agree predictors [11, 35, 48]. The filtering scheme makes use of self-history information to identify easy-to-predict branches dynamically. For these branches, the filter scheme does not update the counters of its global history scheme in order to reduce interference. The skewed predictors reduce interference by having multiple second-level tables, each using a different hashing function for the index into the table, and by partially updating these second-level tables according to the prediction results. The agree predictor changes the usage of counters in the second-level table. In an agree predictor, each static branch is assigned a bias bit, indicating the most likely direction the static branch normally takes. The bias bit can be dynamically determined or set at compile time. The counters in the second-level table are then used to see if they agree with the bias bit when a branch is predicted. Predictions can be based on the result. These three schemes will be examined and compared with our proposed bi-mode scheme in Chapter 5.

Yeh and Patt [59] found that the per-address history scheme predicts better for floating-point scientific programs than the global history scheme. Branches for loops constitute most of the dynamic branch execution in this class of programs and the per-address history (self-history) can quickly capture the loop behavior. On the hand, global history schemes are better for integer programs. In integer programs, there are many if-then-else constructs, which are better predicted by exploiting the correlation of a branch with its neighboring branches. To achieve the overall best prediction accuracy, researchers have proposed to combine these two kinds of branch predictors into hybrid predictors [34, 10]. A processor has a dynamic selector to keep tracks of the performance of these component branch predictors for each branch at run time. The selector "learns" to select the best component predictor for each branch.

Another technique to improve branch prediction is the branch classification of Chang *et al.* [9]. At compile time, a compiler can use profiling to classify each static branch into different groups according to the probability of being taken. For each group, it assigns a different predictor to make predictions. Static classification (compiler's selection) offers an opportunity to examine program semantics, but it is usually not as accurate as dynamic classification (dynamic selectors in the hybrid scheme), because branches may behave differently in various phases of program execution. In the same study, Chang *et al.* have also examined combining the static classification and dynamic selector to achieve better predication accuracy. In this combination, the compiler first identifies a group of static branches that can be better predicted with fixed values. These branches are usually strongly biased branches. Then, a hybrid scheme using a dynamic selector is used to predict for the remaining branches, which are less strongly biased and typically harder to predict.

Hybrid schemes have been reported to achieve a better prediction accuracy than single two-level dynamic predictors, but they are built upon the two-level dynamic predictors. Therefore, improving single two-level predictors remains important for improving branch prediction techniques.

CHAPTER 3

Experimental Methodology and Benchmarks Descriptions

This chapter describes the experimental methodology employed in this dissertation. The benchmarks used in the dissertation are also presented, and statistical profiles are developed to illustrate their differences.

3.1 Experimental Methodology

Trace-driven simulation is employed to conduct experiments in this dissertation. The simulator assumes one branch is fetched and resolved before another branch is fetched. Our simulator has the advantage of being simple and flexible enough to allow the examination of a wide range of dynamic branch predictor organizations; furthermore, the results are independent of underlying machine design parameters, such as branch resolution time. However, this flexibility comes at the expense of some accuracy. For example, in real machines a branch will not be resolved immediately after it is fetched, requiring the branch predictors to be updated speculatively, which may degrade the prediction accuracy slightly from that reported here. The issues of updating branch predictors speculatively has been examined in [23, 26].

The traces used in the experiments were collected from the benchmarks described below. The traces contain among other things, instructions, their addresses, opcodes, address space identifiers, and in the case of branches, their outcomes.

3.2 Benchmarks

One major purpose of this dissertation is to examine the performance of dynamic branch predictors under programs that are representative of general-purposed computing environments. Integer programs and operating system functions are two of the most frequently executed programs in such a computing environment. Scientific programs, that makes heavy use of floating-point operations, are frequently used in some high-performance computing environments, but they are much less representative of present-day general purposed workloads. Moreover, loop branches constitute most of the branches in the floating-point programs, which are much easier to predict than the branches found in integer programs because of their more regular behavior [57].

In this dissertation work a total of fifteen integer program traces are used; six of them are from SPEC CINT95, eight from the Instruction Benchmark Suite (IBS), and one from the DEC PatchWrk suite, *sql95*. Table 3.1 lists characteristics of the benchmarks.

SPEC CINT95 [47] programs were compiled on an Alpha 21064-based workstation with the OSF/1 C compiler using the -O optimization flag and were traced while executing the input files listed in Table 3.2. We employed DEC's ATOM instrumentation tool [17] to capture all user-level instructions for the CINT95 traces, including shared libraries.

The IBS and PatchWrk benchmarks spend more of their execution time in the operating system. The IBS-Ultrix benchmarks are a set of applications running under Ultrix 3.1. The traces were collected through hardware monitoring of a MIPS R2000-based workstation by Uhlig *et al.* [52]. These traces include both instructions executed from the user applications and the operating system, as well as instructions executed by

Benchmarks		Dynamic conditional branches	Portion of dynamic conditional branches from OS	Static conditional branches	Static branches constituting X% of total dynamic conditional branches	
					X=90	X=99
SPEC CINT95	compress	10,114,353	—	482	25	38
	xlisp	25,008,567	—	636	50	125
	perl	39,714,684	—	1,974	156	311
	vortex	27,792,020	—	6,599	354	1,452
	go	17,873,772	—	5,112	1,021	2,443
	gcc	26,520,618	—	16,035	3,244	7,890
IBS-Ultrix	nroff	22,574,884	6.53%	5,249	228	966
	groff	11,901,481	11.37%	6,333	459	1,658
	sdet	5,514,439	98.46%	5,310	508	1,920
	mpeg_play	9,566,290	28.51%	5,598	532	1,899
	video_play	5,759,231	68.35%	4,606	757	1,736
	verilog	6,212,381	13.57%	4,636	850	2,387
	gs	16,308,247	10.33%	12,852	1,160	3,665
	real_gcc	14,309,867	9.99%	17,361	3,214	8,698
	sql95	2,599,046	17.82%	8,748	1,653	3,830

Table 3.1: Benchmarks characteristics

auxiliary processes such as the X-server.

The *sql95* trace is distributed in the PatchWrk suite by DEC [39]. The distributed version of *sql95* is a single-CPU trace of the Microsoft SQL server running on an Alpha under NT 3.5, while executing the TPC-B benchmark. *Sql95* is interesting to us because, in addition to its being representative of database application programs, it exhibits a higher percentage of static branches exercised than any benchmarks in SPEC CINT95 and IBS-Ultrix.

Benchmarks	Input data set
compress	reduced version of *bigtest.in* (reference data), reduced to 30,000 elements (instead of 14,000,000)
xlisp	*train.lsp* (training data)
perl	reduced version of *scrabbl.in* (reference data), reduced to the first 5 items (instead of 7)
vortex	reduced version of training data, *vortex.train*; the iteration counts and data were reduced to the first 10 items (instead of 250)
go	*2stone9.in* (training data), the game_level was reduced to 19 (instead of 50)
gcc	*jump.i* (one of the reference data sets)

Table 3.2: The input data set for SPEC CINT95

The input data set to the SPEC CINT95 benchmarks was a reduced set; each benchmark was run to completion.

Considered statically, the SPEC CINT95 benchmarks seem like a reasonable set of programs with which to study branch prediction. All contain a fairly large number of conditional branch instructions, with five containing over a thousand. However, when the dynamic frequency of these branches is examined, potential problems with the benchmarks can be seen. In three out of the six benchmarks, a small number of distinct branches contribute the overwhelming majority of the branch instances. The small number of branches is related to the small instruction cache footprints for these programs. This characteristic was noted for SPEC CINT92, the previous version of SPEC CIN95, and rendered them unsuitable for evaluating the instruction caches for high performance microprocessors [52].

The IBS-Ultrix benchmarks and PatchWrk *sql95* all exercise substantially more static branches than the SPEC CINT95 benchmarks. The IBS-Ultrix benchmarks include a run of the GNU C compiler, *real_gcc*, which, though run on different inputs than the SPEC CINT95 *gcc* benchmark, is quite comparable to it. The remaining programs of IBS-

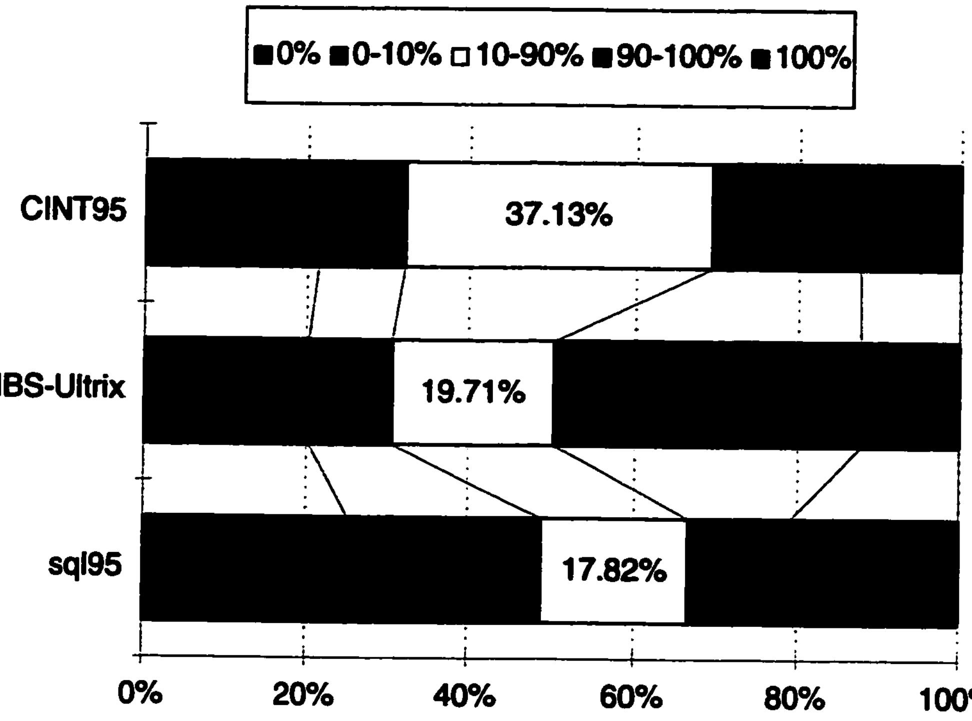


Figure 3.1: Classification of dynamic branches by probability taken

For each benchmark we first calculated the proportion of the time that each static branch was taken. We then determined the proportion of the corresponding dynamic branches that fell into each of five ranges. Five ranges are specified: never-taken (0%), between 0% and 10%, between 10% and 90%, between 90% and 100%, and always-taken (100%). This figure shows the average size of the ranges for the programs within each benchmark suite.

Ultrix are somewhat smaller than either version of *gcc*. It will be shown in this dissertation that the inclusion of operating system references does not strongly affect the predictability of branches in these programs, aside from the consequences of trying to predict a greater number of branches. The operating system branch behavior actually falls within the range covered by the IBS and *sql95* application programs.

A higher number of branches does not necessarily lead directly to greater difficulty for prediction as far as the predictability is concerned. In Figure 3.1 we classify dynamic branches by their probability of being taken for three benchmarks sets, according to the following formula [9],

Though the IBS and *sql95* benchmarks have substantially more static branches

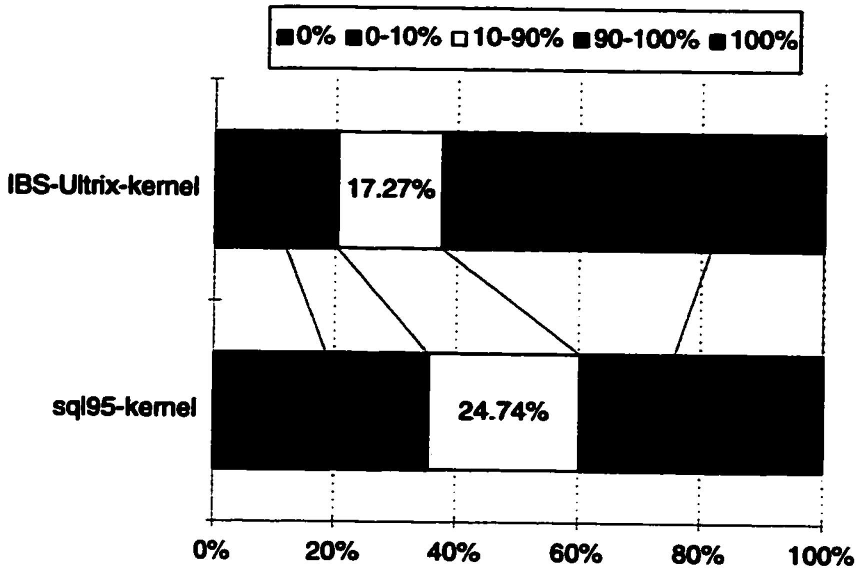


Figure 3.2: Classification of dynamic branches by probability taken

For each IBS-Ultrix benchmarks and *sql95*, we first calculated the proportion of the time that each static branch of the kernel (OS) was taken. We then determined the proportion of the total dynamic branches of the kernel that were to static branches that fell into each of five ranges. Five ranges are specified: never-taken (0%), between 0% and 10%, between 10% and 90%, between 90% and 100%, and always-taken (100%). This figure shows the average size of the ranges for the kernel part of the programs within two of the benchmark suites.

$$\text{normalized dynamic count of a branch class} = \frac{\text{dynamic count of the branch class}}{\text{total dynamic branches}},$$

where a branch class is a group of static branches whose probability of being taken falls in a specific range. Five ranges are specified: never-taken, between 0% and 10%, between 10% and 90%, between 90% and 100%, and always-taken.

than CINT95, most of these additional branches are highly biased compared to CINT95. They are either almost always or almost never taken, and mainly consist of loops, or error and bounds checks. Moreover, this highly biased feature is true for both applications and the operating system. In Figure 3.2, we collect the statistics of Figure 3.1 for IBS and *sql95* kernel branches only, and found that the kernel branches exhibit similar characteristics. Other studies have noted the frequency of highly biased branches is strongly related to the capability of achieving high prediction accuracy [9, 19, 61].

To summarize, the benchmarks that we collected to represent more realistic computing environments. For example, the IBS benchmarks and *sql95* have on average many more static branches than the SPEC CINT95 benchmarks. These extra branches are not harder to predict by nature, but they may pose more problems for predictor design than SPEC CINT95. The problems arise from the severe interference at the predictors, which will be fully discussed in the remaining of this dissertation.

3.3 Performance metrics

In the study that follows we use *misprediction rate* for conditional branches as the performance metric. Changes in misprediction rate do not translate directly into changes in performance, i.e., execution time. The execution time is also determined by the performance penalty due to each misprediction. The performance penalty, however, depends upon the size of instruction window allowed within the computing engine, the depth of pipelines, and the availability of the branch target instructions. Sometimes, the branch penalty can be hidden because processors may also stall and wait for imperfect memory systems. All of the factors mentioned above are machine dependent; different microarchitectural designs have different values. We restrict ourselves in this study to the misprediction rate in order to examine wider ranges of design spaces and to reveal the machine-independent global performance trends for branch predictors. A number of studies have made careful assessment of the link between changes in the misprediction rate and changes in performance [6, 7, 19, 33, 58].

CHAPTER 4

Global History Schemes

This chapter is concerned with the optimal design of two-level dynamic branch predictors that use global history information. A model for such a predictor is first presented. A study that shows how to search the optimal design for such schemes then follows. This chapter also provides a general picture of global history schemes that shows their performance trends.

The major design issue for the global history scheme is how the second-level table, the table containing two-bit counters, should be organized to achieve optimal performance. Specifically, an answer is needed to the trade-off between the number of branch address bits, and the number of branch outcome history bits, that are required for an optimal predictor.

However, the answer to this design question is highly dependent on the applications to be predicted: the different footprint sizes (the number of static branches) of applications can shift the optimal design point. To help with this, the set of benchmarks, discussed in Chapter 3, is used in an attempt to provide a more complete answer.

As will be seen from the general performance pictures, there is a key design problem existing in the global history schemes. It is the destructive interference in the second-level table that is particularly prevalent when global history is heavily employed. A detailed analysis on the dynamic behavior of predictors is conducted to explain why destructive interference exists in this class of predictors and how it degrades the performance. Finally, based on the analysis, two criteria will be proposed for designing a good global history two-level branch predictor.

4.1 The global history scheme model

In this section, a thorough performance evaluation and optimal design configuration search for the global history two-level predictor is presented.

As described in Chapter 1, there are three variations of the global history scheme, GAg, GAs and GAp. The GAg scheme actually is a special case of the GAs scheme when the second-level table has only one column. GAp, the scheme allocating one column for each static branch, is an ideal scheme which cannot be implemented due to hardware resource restriction. Accordingly, we focus our study the GAs scheme (including the GAg) in this section.

A GAs model is shown in Figure 4.1. With a fixed number, 2^n, of two-bit counters in the second-level table, a GAs scheme may have a variety of possible second-level table configurations. One extreme is to merge together all the rows (all possible history patterns for a given branch) to a single row and simply use an n-bit branch address to select a counter from the row. This is exactly the form of the two-bit counter scheme proposed by Smith [46], which we will refer to as the "address-indexed predictor." At the other extreme, one can reduce the table to a single column, and the selection of a counter is then solely determined by the n-bit output of the first-level table, which is simply the outcomes of the most recent n dynamic branches executed. Yeh and Patt encode this single-column scheme as a GAg scheme, and we will follow this convention in this chapter. In addition

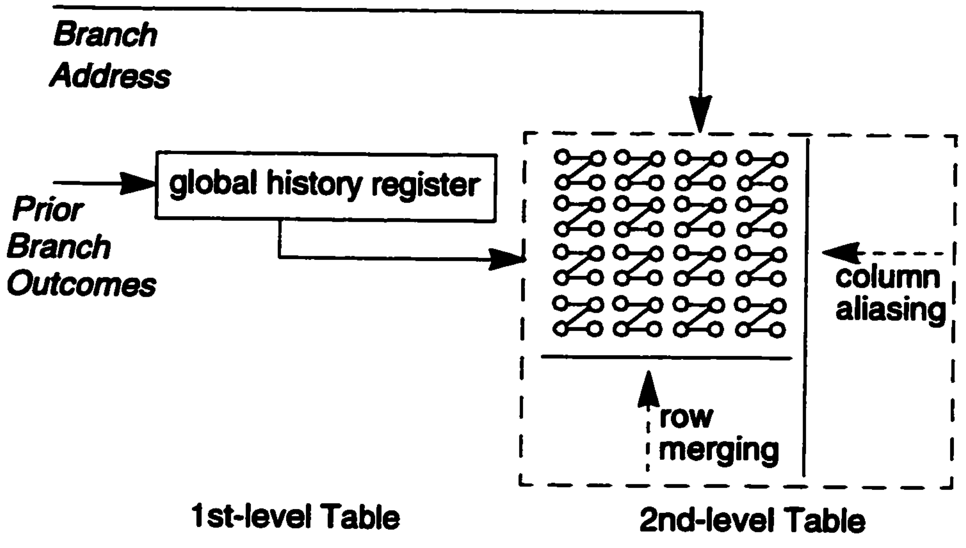


Figure 4.1: The model for the global history two-level dynamic branch predictor

to the two extreme configurations for the second-level table, there are many other configurations each of which has a pair-wise combination of r-bit global history and c-bit branch address that form an n-bit index into the table (n = r+c, r and c are non-negative integers).

4.1.1 Performance comparison between the address-indexed and GAg schemes

Figure 4.2 compares the misprediction rates for all of our benchmarks using address-indexed and GAg predictors for various sizes of the second-level table. The rate is plotted as a function of the number of address or history bits used to index the counters. The size of the second-level table ranges from 16 (2^4) to 32,768 (2^{15}) counters.

For the address-indexed scheme, the optimal performance is achieved when each static branch of a program is assigned a unique counter for prediction. Therefore, we can find that the performance of the address-indexed scheme for the small SPEC CINT95 benchmarks, such as *compress* and *xlisp*, is almost saturated; specifically, there is no performance improvement after the table contains 512 (2^9) counters or more (using 9 or more address bits) for these benchmarks. Since only a tiny numbers of branches are exercised, all but the smallest of tables assign a separate counter to each static branch, and no additional improvement can be found by increasing the table size (by increasing the address bits). For the large benchmarks, such as *real-gcc* and *sql95*, the address-indexed scheme can still gain some degree of improvement even for the largest tables. This is simply because there are still some counters shared by static branches in the largest predictors and with more counters the sharing can be continuously reduced.

Although it is not as good in most cases, the GAg scheme, using the global history to select from a column of counters, shows continuous improvement as the table size is increased (more history bits are used). This suggests that employing more global history bits, or exploiting more correlation between branches, can consistently provide some benefit over the range of the sizes examined.

However, as McFarling pointed out [34], global histories are usually weaker than branch addresses at identifying branches, and this is especially true for short histories. In other words, it is more common to find that different static branches generate the same short global history patterns than have the same partial branch addresses. For short histories, the aliasing of different branches to the same history pattern can be so severe that the benefit of correlation is overwhelmed by the interference. In this case, simply using the branch address to select counter performs better.

Therefore, as shown in Figure 4.2, for most of the benchmarks except the small SPEC benchmarks (like *compress* and *xlisp*), the address-indexed scheme performs better

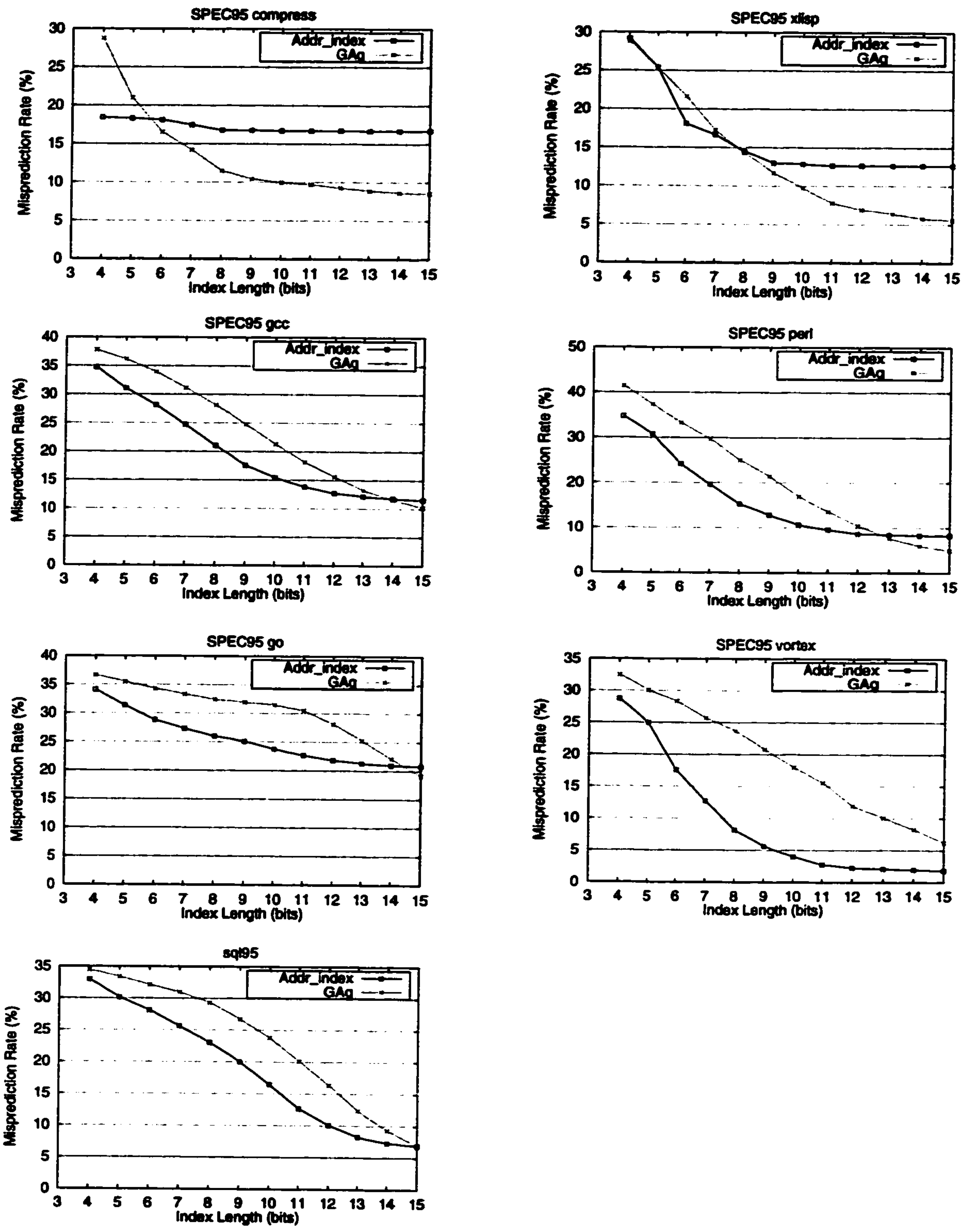


Figure 4.2: Misprediction rates for each individual benchmark using address-indexed and GAg schemes (cont. on next page)

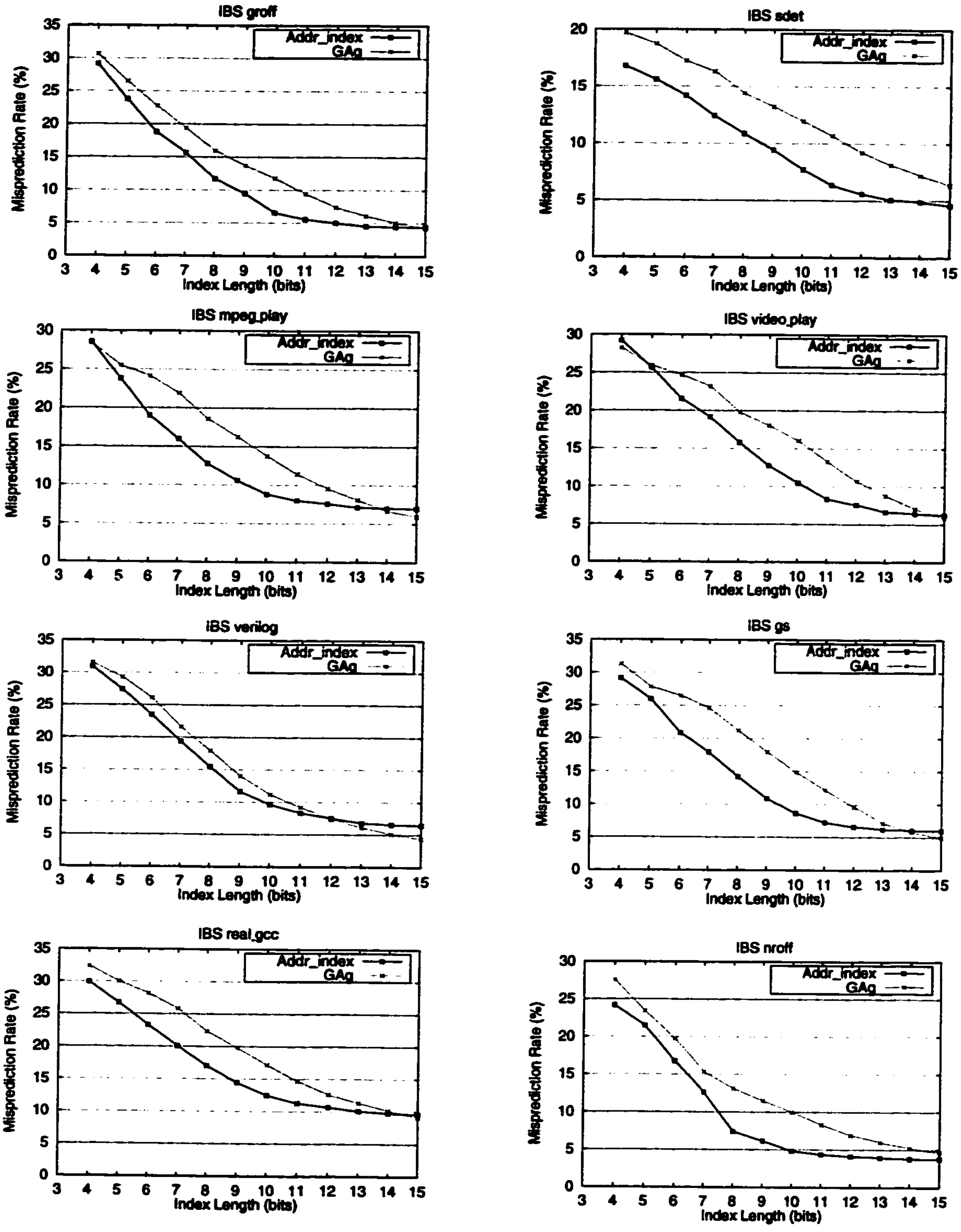


Figure 4.2 (continued): Misprediction rates for each individual benchmark using address-indexed and GAg schemes

than the GAg scheme. For small benchmarks where there are few static branches to exercise, there are enough global history patterns generated, so that, even with short histories, aliasing is not a concern and the performance of GAg scheme is good. In contrast, when many static branches are exercised, the aliasing of a short history pattern is so high that the resulting performance is bad. If a branch prediction study uses only these small benchmarks, the results will favor the GAg scheme rather than the address-indexed scheme. Unfortunately, in earlier SPEC benchmark suites, SPEC89 and SPEC92, most of benchmarks have the same footprint size as *compress* and *xlisp* [45].

To summarize, in this section we showed and compared the performance for the two extreme configurations of the global history scheme, the address-indexed and the GAg schemes. With relatively few static branches in the small SPEC CINT95 benchmarks, the GAg has little interference for short histories, and thus it can quickly outperform the address-indexed scheme as the table size is increased. However, for large programs such as *go* and *gcc* of the CINT95, all the IBS benchmarks, and *sql95*, the GAg scheme suffers from the severe interference because of the large numbers of static branches and hence GAg performs worse most of time. However, the GAg performance can be improved continuously, though for *go* and *sql95* the improvement is not significant until after the second-level table contains more than 1,024 counters. This suggests there may be a role for global history bits, and as we will see in the next subsection, there is.

We now turn to a more complete performance evaluation for the entire design space for global history schemes.

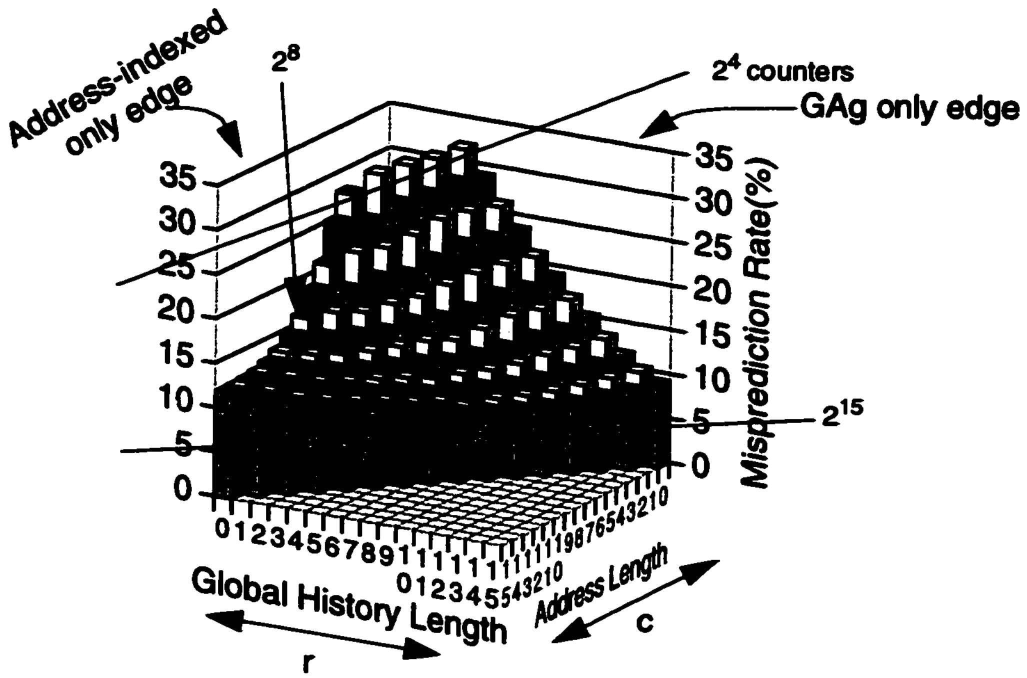
4.1.2 Optimal configurations for the GAs schemes

In this section, an analysis is presented that exhaustively searches the design space of the GAs schemes to show performance trends and to determine optimal configurations. A variation of the GAs scheme, referred as gshare, has been proposed as an improvement by McFarling [34]. It will be included in the next section because of its increasing popularity in research work [7, 10, 21, 61].

The address-indexed and GAg schemes are the two extreme ends of the spectrum of predictor table configurations, as explained earlier. GAs schemes generalize the GAg scheme by allowing the address to be used to select one from a set of columns, while using global history selects one of the rows. One can arrange 2^n state machines in n+1 configurations of 2^c columns and 2^r rows, where $c+r = n$. At the extremes, the configuration with 2^n rows is identical to the GAg scheme just described, while the configuration with 2^n columns is identical to the address-indexed scheme.

We can view the performance of all the GAs configurations as forming a surface that interpolates between the performance curves for the address-indexed and GAg schemes. Figure 4.3 and Figure 4.4 present the surfaces for the averaged misprediction rates of CINT95, IBS-Ultrix, and *sql95*. Each gray or white tier represents a line of having a constant number of two-bit counters, ranging from 16 (2^4) for the rearmost to 32,768 (2^{15}) for the frontmost. Each tier ranges from the address-index configuration on the left to the GAg configuration on the right. Within each tier, we have marked in black the bar that represents the optimal configuration for that size of predictors.

The shapes of GAs performance surface for these three benchmark suites are similar: when the second-level table is small, the misprediction rates are strikingly high



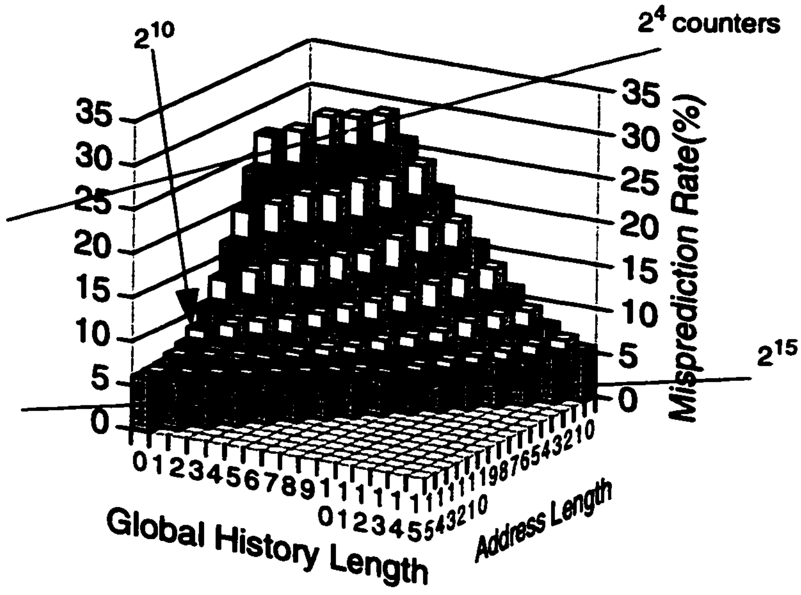


Figure 4.3: Averaged misprediction rates of GAs — SPEC CINT95 (top) and IBS-Ultrix (bottom)

In this figure, the averaged misprediction rates of the GAs scheme for the SPEC CINT95 and IBS-Ultrix benchmarks are presented. The second-level table of the GAs scheme ranges from 2^4 counters (the tier in back) to 2^{15} counters (the tier in front). All possible combinations of address and history bits for each size of the table are examined. Each white and grey tier represents a fixed-size table. The optimal configuration within each tier is marked in black. We show that the configuration that does not use history bits (the address-indexed 2-bit counter scheme) is the best for small budgets, while the GAs scheme using some address bits becomes the best for large budgets. GAg, a scheme without using address bits, is always suboptimal. The IBS benchmarks are on average larger than the SPEC, and for these large benchmarks the address-indexed scheme is still the best even for a table as large as 2^{10} counters.

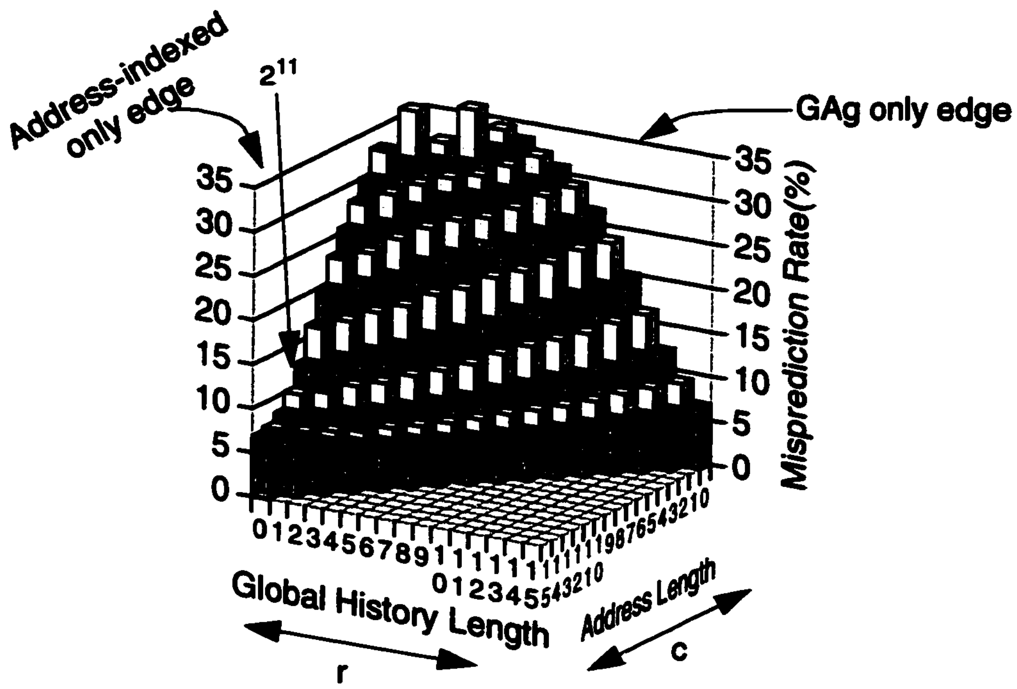


Figure 4.4: Misprediction rates of GAs — sql95

This figure shows the misprediction rate for the *sql95* benchmark. It can be seen that most of the time the address-indexed 2-bit counter scheme is the best configuration.

and the address-index scheme is always the best predictor. As the table becomes larger, the best configurations, marked in black, move to the middle range of tiers, which represent multiple-column, multiple-row configurations. One noticeable difference between these three benchmark suites is that black bars of CINT95 move to the middle range more quickly than the other two.

The primary determinant for these surface shapes and the optimal points is the numbers of distinct branches exercised and the resulting interference. The more distinct branches exercised, the higher chances to have interference. *Interference* occurs when branch instances accessing a particular counter are from distinct static branches. If two static branches tend to have opposite outcomes, destructive interference occurs which can degrade the performance. This will be clear as each benchmark is examined individually.

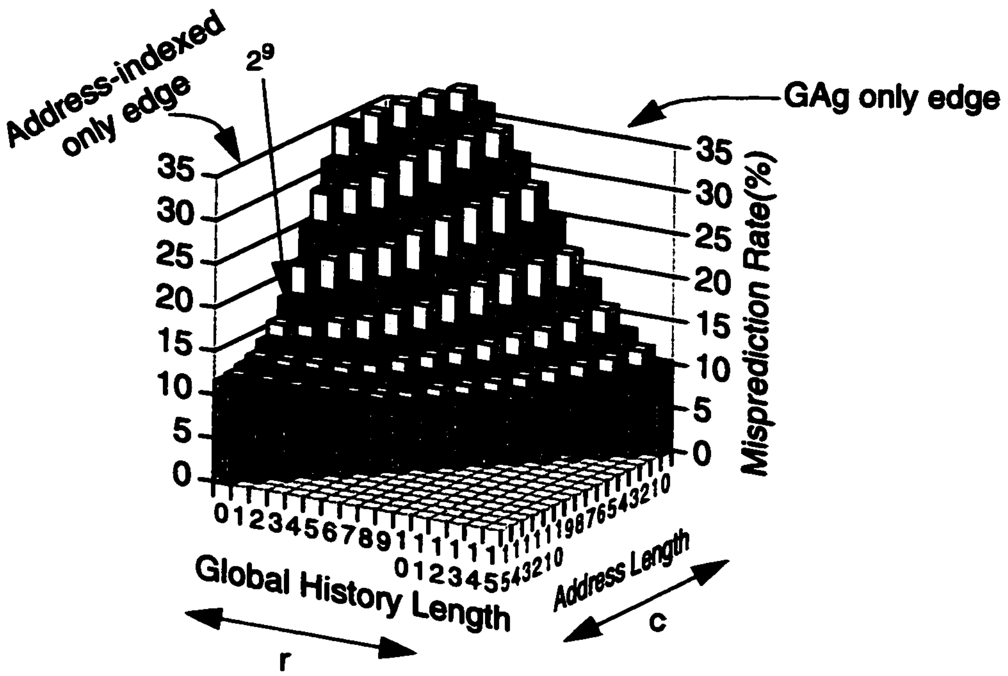


Figure 4.5: Misprediction rates of GAs — CINT95 *gcc*

The misprediction rates of GAs for the *gcc* benchmark of CINT95 is shown in Figure 4.5. The *gcc* result is representative of most of the benchmarks, except a few small benchmarks such as *compress* and *xlisp*. When the second-level table has fewer than 1,024 counters, the address-indexed scheme is again the best. This is because interference occurs even in moderate size tables and negatively dominates the performance results. Doubling the number of columns in the table, at the expense of halving the number of rows, will help significantly reducing interference, since the address bit is more useful at distinguishing between branches than is the global history. Besides, because a high proportion of the branches are strongly biased, the penalty for any additional interference is so severe that it far outweighs any possible benefit from exploring global history patterns within a branch. Thus, the best performance available for small- to moderate-size tables comes from the simple address-indexed scheme. For larger tables, interference will

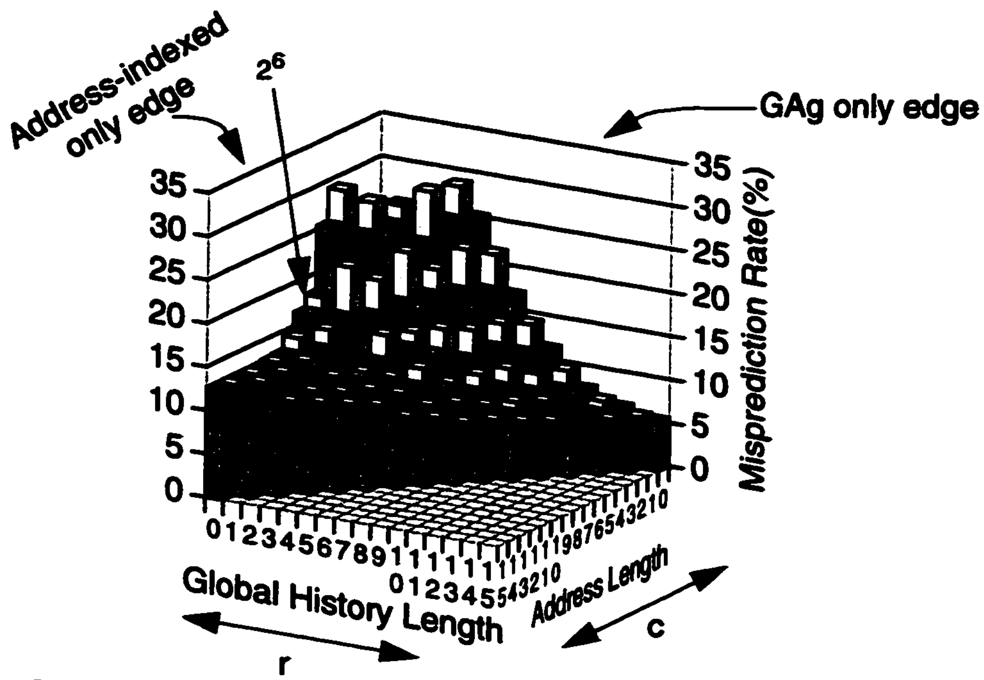


Figure 4.6: Misprediction rates of GAs — CINT95 *xlisp*

be low as long as sufficient address bits are used. For these tables, dividing a column into additional rows can pay off up to a point.

On the other hand, the results from small benchmarks fail to reveal the complete performance characteristics discussed above. Figure 4.6 shows the performance surface for the small CINT95 benchmark, *xlisp*. The number of branches in *xlisp* is so small that for moderate-size tables it is possible to devote several counters to each branch. Under these circumstances, very little interference occurs, provided even a few address bits are used. We have also found that other small CINT95 benchmarks as well as most of the CINT92 benchmarks, such as *compress*, *eqntott* (CINT92) and *espresso* (CINT92), exhibit the same characteristics as *xlisp*, i.e., preferring more history bits to address bits in order to achieve a better performance.

The tilt towards small programs of the SPEC CINT92 benchmark suite, and of the

SPEC89 integer benchmarks before it, has resulted in widespread misunderstandings about the performance of global history schemes. The relatively small numbers of the active branches tend to overstate the benefits of associating multiple state machines with individual branches, thus overwhelming the advantage of global history bits.

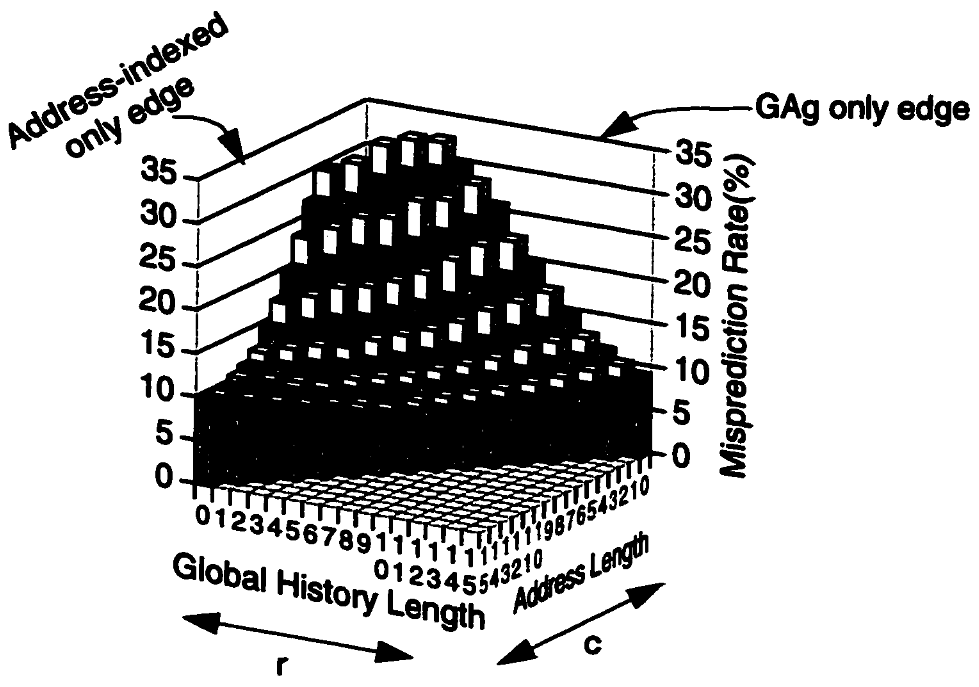
4.1.2.1 Operating systems effects on branches

More than half of the benchmarks used in this dissertation work contain branches from operating systems (OS): all the IBS benchmarks include branches from *Ultrix*, and *sql95* contains branches from *Window NT*. We are interested in the effects of OS branches on the performance of predictors.

Branches from operating systems are not less predictable than those from application codes. As we saw in Chapter 3, most operating system branches are easy to predict because operating system branches are more highly biased. They are either taken or not-taken most of time, since many of them are for loops that move or copy data, or for error checking, which is only very rarely invoked.

However, adding operating system branches means adding in more static branches to the branch predictor which potentially can increase interference. This is the reason why all the IBS benchmarks require some branch address bits as part of the index into the second-level table to achieve good prediction accuracy, since the branch addresses are the best means of reducing interference.

The performance surface of a predictor is strongly determined by the number of distinct branches involved, rather than the sources of branches, be they applications or operating systems. Figure 4.7 presents the misprediction rates of GAs schemes for two



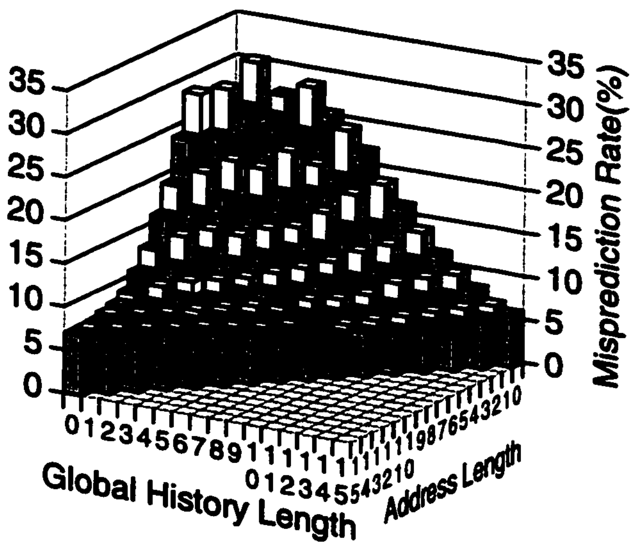


Figure 4.7: Misprediction rates of GAs — *real_gcc* (top) and *mpeg_play* (bottom)

This figure shows that including OS branches in the branch execution stream will not change the surface we observed for the SPEC benchmarks. OS branches are not hard to predict, but may increase numbers of static branches, thus worsening the interference problem for branch predictors.

IBS benchmarks, *real_gcc* and *mpeg_play*. The surfaces of these two benchmarks are similar to that of *gcc* (see Figure 4.5); all characterize the surface of large footprint programs. Of particular note is *real_gcc*. It compiles an application program which is very similar to the *gcc* of SPEC CINT95, except that *real_gcc* also includes branches from the *Ultrix* system calls. By comparing performance surfaces for *real_gcc* and *gcc* benchmarks (see Figure 4.5), we can see that including operating system branches does not change the characteristics observed earlier. Therefore, in a real computing environment where an operating system is involved, controlling interference is still the key issue to the design of global history predictors.

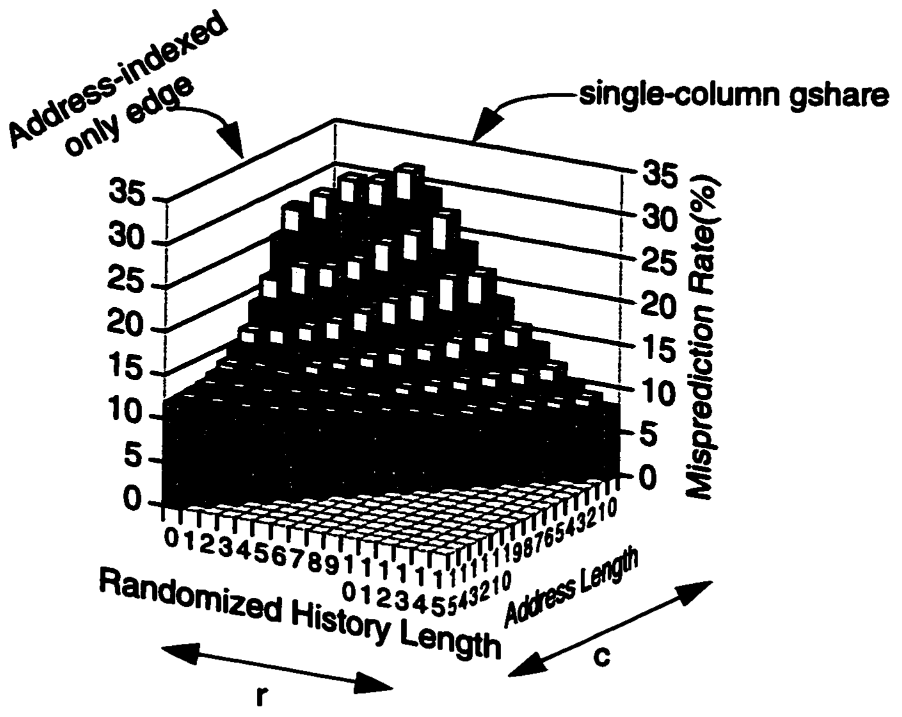
4.1.3 The gshare scheme

McFarling [34] proposed a variant to the GAs scheme, referred to as gshare, in which the global history is xor-ed with bits from the branch address. The idea is to combine the information from the global history and the address bits more effectively. McFarling reasoned that in sufficiently large tables this xor-ing can reduce interference between the global history patterns while retaining the advantages of using long global history to discover branch correlation. By xor-ing the global history pattern with branch addresses, the gshare scheme can produce new distinct indexing values for the counters, each associated with a static branch. As in the GAs, additional address bits can be used to select one of several columns. Hence, for a fixed table size there is a range of gshare configurations. McFarling compared the best performance for a given size predictor table of any GAs configuration with the best performance for any gshare configuration and found a slight advantage for gshare schemes on the SPEC CINT92 benchmarks. We should note in passing that many subsequent studies of gshare have been limited to configurations with a single column (one PHT).

For the gshare in our experiments, we formed an index of n bits (n = r+c, as before, r is the number of row index bits and c the column index) to the second-level table by using the lowest-order c address bits as the column index and xor-ing the next low-order r address bits with the most recent r global history bits as the row index. Figure 4.8 and Figure 4.9 show the experimental results of such gshare schemes for the three benchmark sets. The resulting surfaces are almost identical to those of the GAs scheme, except that as the table increases the lowest bars, marked in black, tend more to move to the right side, favoring one or two more history bits than the GAs. Note that the leftmost configurations within each tier are for the address-indexed schemes and are thus exactly the same as the leftmost configurations in Figure 4.3 and Figure 4.4.

Figure 4.10 and Figure 4.11 show the difference in averaged prediction rates between the GAs and gshare schemes with identically configured second-level tables. The benchmarks are again the CINT95 suite, the IBS-Ultrix suite and *sql95*. Positive numbers indicate superior prediction by gshare. We see that the areas of superior performance of gshare are clustered on the right side of the graph, where the tables have more rows than columns. For these configurations, xor-ing can be effective because there is a better chance to randomize the row index.

In Figure 4.12, a detailed examination for three representative benchmarks is shown, including *xlisp*, *gcc*, and *sql95*. It shows the comparison between gshare and GAs schemes for three kinds of the second-level table, including 1-, 2-, and 8-column (PHT) tables. For each of them, a range of history lengths are examined (in other words, a range



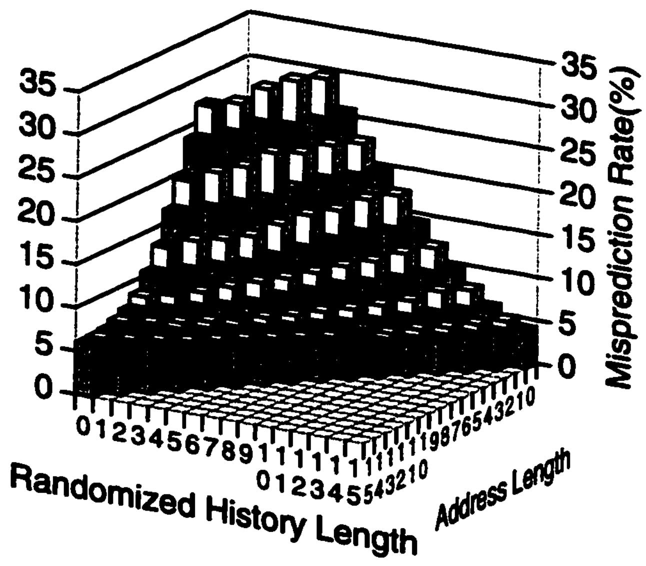


Figure 4.8: Averaged misprediction rates of gshare—SPEC CINT95 (top) and IBS-Ultrix (bottom)

In this figure, the averaged prediction rates of the gshare scheme for the SPEC CINT95 and IBS-Ultrix benchmarks are presented. The second-level table of the gshare scheme ranges from 2^4 counters (the tier in back) to 2^{15} counters (the tier in front). All possible pair-wised combinations of address and history bits for each size of the table are included. Each white and grey tier represents such possible pair-wised combinations for a fixed-size table. The optimal configuration within each tier is marked in black. It can be seen that even though the index of the second-level table is randomized in an attempt to reduce interference, the optimal configurations are still the same as we observe for the GAs scheme: the address-indexed scheme is the best for small budgets, while the GAs-like scheme using some address bits becomes the best for large budgets.

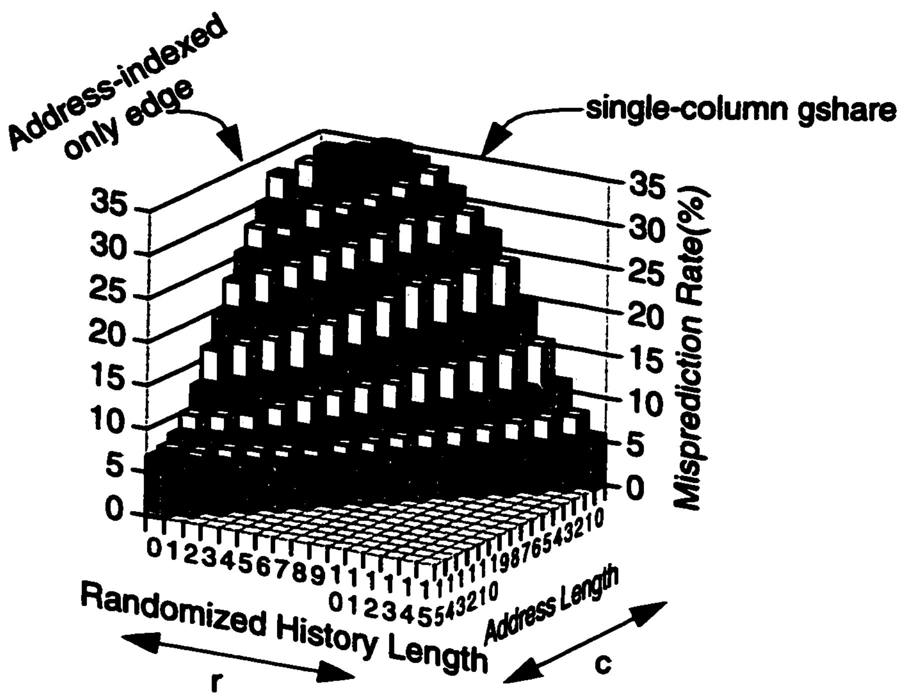
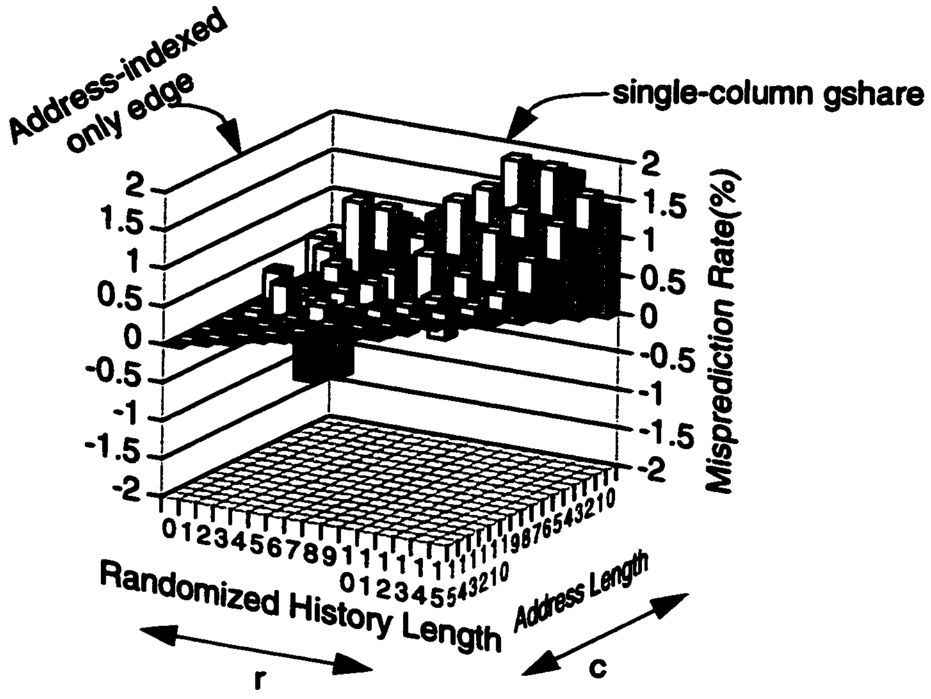


Figure 4.9: Misprediction rates of gshare — sql95

of column sizes are examined). It can be seen that, the randomizing effect in gshare can be positive only when predictors have few but long columns. Long columns means many history bits are used, and the "many" is relative to the numbers of static branches involved. *xlisp* has the smallest number of static branches to exercise, so gshare can outperform the GAs for most of the configurations. However, for the large benchmarks, *gcc* and *sql95*, gshare is better only when significantly long history is used.

Nevertheless, these configurations are, in fact, suboptimal for both the GAs and gshare schemes, so improving their performance is not meaningful. In the area towards the center of the global performance pictures, where GAs schemes achieve their best performance, GAs and gshare differ little in performance. In other word, randomizing global history patterns is less effective at reducing interference when branch address bits are also directly employed.



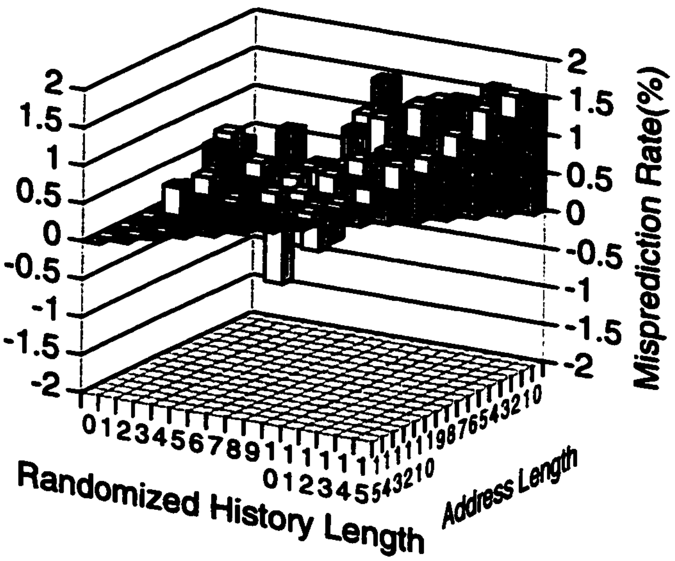


Figure 4.10: Difference in averaged misprediction rates between gshare and GAs — SPEC CINT95 (top) and IBS-Ultrix (bottom)

In this figure, the difference in averaged prediction rates is measured for the SPEC CINT95 and IBS-Ultrix benchmarks between the gshare and GAs schemes with identically configured second-level tables. The positive Z-axis value indicates superior prediction by gshare. For both schemes, the second-level table ranges from 2^4 counters (the tier in back) to 2^{15} counters (the tier in front). The optimal configuration within a tier for the gshare scheme is marked in black. It can be seen that gshare schemes are superior for those configurations heavily using history bits. Those configurations are suboptimal. For the optimal configurations, the difference between gshare and GAs schemes is marginal (less than 0.5%).

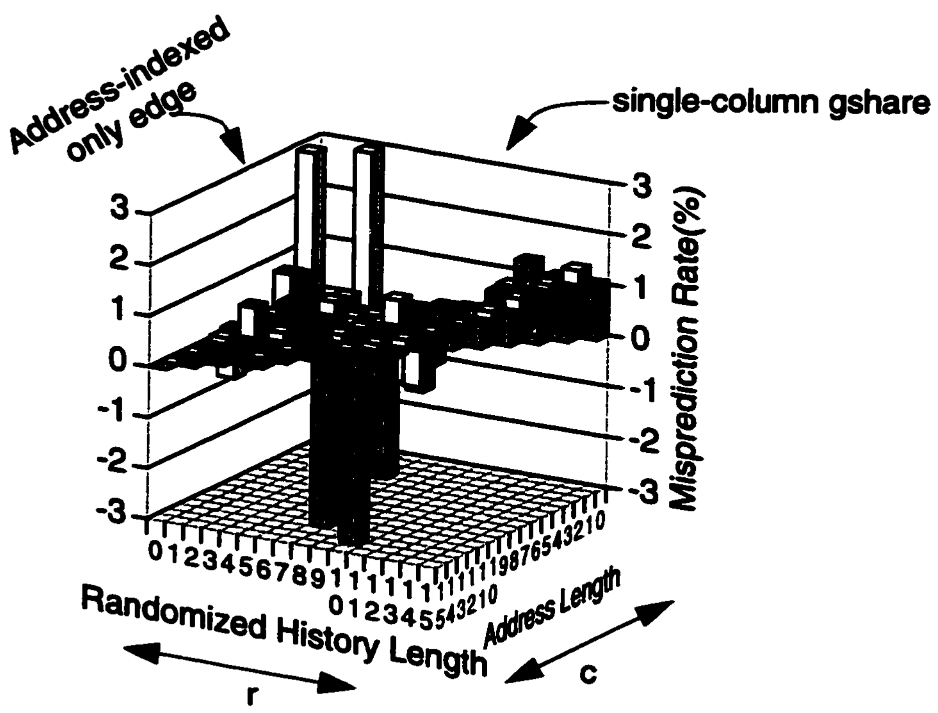


Figure 4.11: Difference in misprediction rates between gshare and GAs — sql95

In summary, for the global history schemes, the branch address provides the most effective way to discriminate between branches and to reduce interference. Adding partial branch addresses can successfully discriminate these branches for the same global history patterns and thus improve the performance significantly. Only after enough address bits are used to reduce interference, can we start to exploit global history patterns to gain more benefit.

The gshare scheme attempts to reduce interference by randomizing global history patterns, or the row index. However, this scheme offers limited benefits, because randomization can only "blindly" separate aliased branches. This process may reduce the destructive interference by chance. Therefore, gshare works better only when long columns (long PHTs) are used. With longer columns the chance that aliases are separated

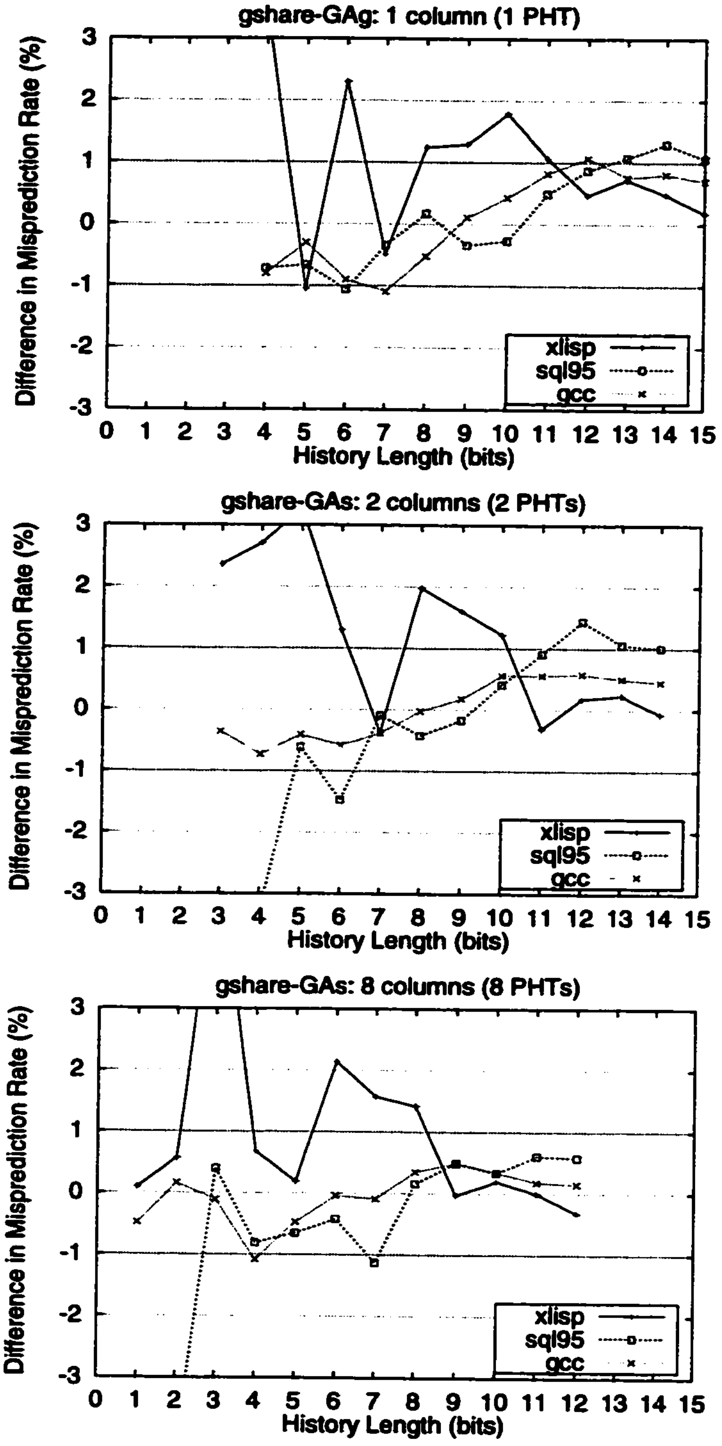


Figure 4.12: Difference in misprediction rates between gshare and GAs for *xlisp*, *sql95*, and *gcc*.

These three plots show that gshare is superior to the GAs scheme only when the benchmark is small or when there are few but large PHTs. There are three benchmarks examined, *xlisp*, representing a small benchmark, and sql95 and gcc, representing large benchmarks. Three second-level tables are examined, 1 PHT, 2 PHTs, and 8 PHTs. In each plot, the X-axis values represent the numbers of history bits; longer history means large PHTs. The positive Y-axis values indicate gshare is better.

is larger. Meanwhile, harmless interference may also be reduced, which is not necessary. In the next chapter, we will show a better variant of the GAs scheme, the bi-mode scheme, which improves performance by reducing interference more intelligently.

To improve performance for the global history scheme, it is necessary to understand the pros and cons of using global history bits in more depth. A good design should preserve the merits of using global history bits and avoid the problems. In the next section we will present a detailed study that identifies the strong points and weak points of using global history bits by analyzing the dynamic branch streams that arrive at each counter in the second-level table.

4.2 The effect of correlation

Many branches have a tendency to be either taken or not-taken most of time. Common examples are branches for error checking and loops. These kinds of branches are usually described as being strongly biased in one direction. As might be expected, strongly biased branches are much easier to predict than weakly biased branches in dynamic branch predictors, and this was confirmed by Chang *et al.* [9]. In the same study, they also measured the distribution of branch biases for SPEC CINT92. Their measurement showed that on average about 50% of total dynamic branches correspond to static branches that are biased in either the taken or not-taken direction for more than 90% of the time.

In this section, we extend the bias measurement to the dynamic branch outcome streams in predictors that exploit correlation. First, we will show that, if correlation is employed static branches show stronger biases than if correlation is not used. Second, we will show that, in the two-level schemes, a predictor that employs more global history bits has the potential to achieve better prediction accuracy, but it suffers from destructive interference between oppositely biased streams. This becomes the major performance limitation.

4.2.1 Decomposition of the branch execution stream

We first examine the bias of each static branch when different degrees of correlation are exploited. We consider the collection of dynamic instances of a static branch associated with a global history pattern as a distinct branch outcome substream. For example, when a 4-bit global history is used, there can be 16 substreams from a static branch because there are 16 ($=2^4$) possible global history patterns.

We then classify each of these substreams to a bias class by the probability of a branch outcome being taken in the substream. There are five bias classes corresponds to five ranges of probability: 0%, (0%, 10%], (10%, 90%)[1], [90%, 100%), and 100%. For the following discussion, we define the 0% (never-taken) and (0%-10%] classes as strongly-not-taken (SNT), the 100% (always-taken) and (90%-100%] classes as strongly-taken (ST), and the (10%-90%) class as weakly-biased (WB).

An entire outcome stream of a static branch *i* can be decomposed into multiple substreams; each of them, represented as s_{ij}, is an outcome sequence of the static branch, *i*, associated with a particular global history pattern, *j*. Let t_{ij} of these outcomes be taken. The term, $|s_{ij}|$, denotes the dynamic count of the instances of the static branch *i* when the associated global history pattern is *j*. We define $|t_{ij}|$ in a similar way. Substreams of a static

1. (10%, 90%) is the range from 10% to 90%, not inclusive. (0%, 10%] is the range from 0% to 10%, not including 0%.

branch can then be classified to one of the five bias classes based on the value of $\frac{|t_{ij}|}{|s_{ij}|}$. This is the bias of a static branch in the presence of correlation.

We consider three different global history lengths, 0, 4, and 15 bits, in this experiment. Therefore, in the case of 4-bit global history, for example, *i* runs from 0 up to the total number of static branches exercised, and *j* can be any of the 16 patterns.

Table 4.1 illustrates the bias classification for substreams from a static branch *b* for

global history pattern, *j*:	dynamic count of the static branch *b* under the history pattern *j*, $\lvert s_{bj}\rvert$	count of taken outcomes of the static branch *b* under the history pattern *j*, $\lvert t_{bj}\rvert$	bias class
0010_2	40	21	(10%-90%)
1010_2	80	2	(0%-10%]
1111_2	80	79	[90%-100%)

Table 4.1: An example of the bias classification for a branch *b* = *0x030*.
In this example, we assume the branch *b* has no dynamic instances for all 4-bit global history patterns except the three shown in the table.

a 4-bit global history. Supposed the branch address is 0x030, and this static branch has only produced three global history patterns (j = 0010_2, 1010_2, or 1111_2) during the program execution, i.e., there are three substreams from the static branch. These three substreams fall into different bias classes with respect to each pattern *j* due to a different likelihood of being taken ($\frac{|t_{ij}|}{|s_{ij}|}$). The dynamic count of the [90%-100%) class, in this example, is 80, the (0%-10%] class is 80, and the (10%-90%) class is 40.

We then calculate the normalized count for each bias class in a program by normalizing the sum of the dynamic counts of all static branches in the same bias class with the total dynamic branch count of the program.

When no correlation is used, i.e., 0-bit global history, each static branch has only

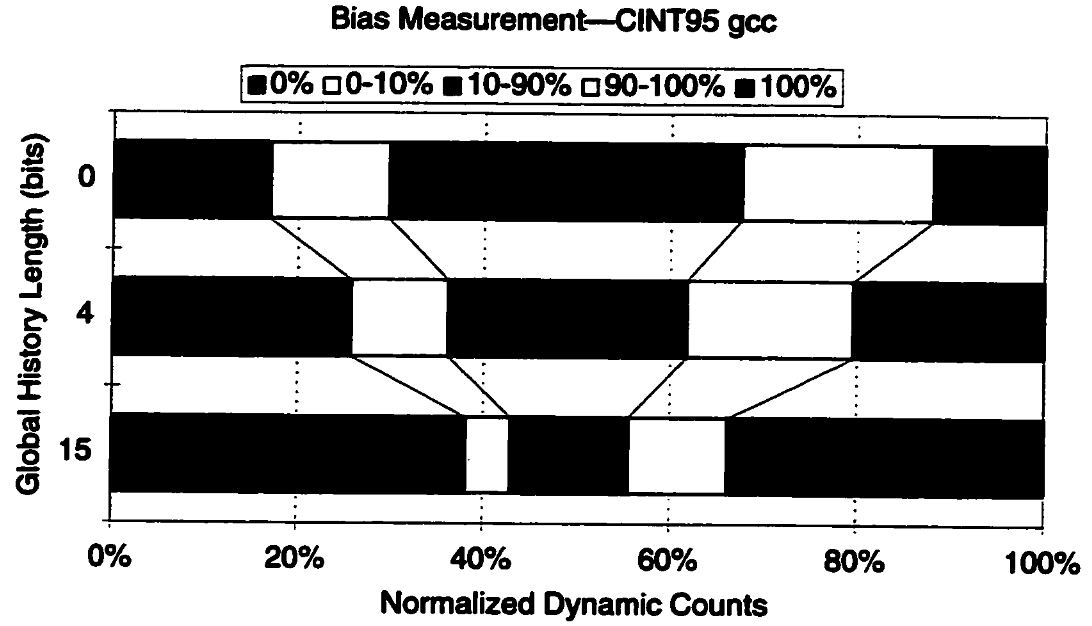


Figure 4.13: Bias measurement when different numbers of global history bits are used — SPEC CINT95 *gcc*.

This graph shows that dynamic branch streams become more strongly biased when more global history bits are used. We first measure the bias (probability of being taken) for the outcome stream of each (address, history) pair. According to its bias, each (address, history) pair is classified into one of the five bias classes (0%, 0-10%, 10-90%, 90-10%, and 100%). The dynamic count of each bias class is accumulated and normalized to the total count of dynamic branches in the program. The stack bars from the top to the bottom represent different global history lengths, 0, 4, and 15 bits, respectively. It can be seen that, when more global history bits are used, the WB region (10-90%) becomes smaller, suggesting a better chance for very accurate branch prediction.

one substream, which is simply the original outcome sequence generated by the static branch. In this special case, each static branch belongs to one bias class exclusively. The result of this measurement is, in fact, the one shown in Chapter 3, Figure 3.1, which corresponds to the definition used by Chang *et al.* [9].

Figure 4.13 presents the experimental results for the CINT95 *gcc* benchmark. The Y axis lists three different degrees of correlation and the X axis represents the normalized dynamic counts of the bias classes in the program; the 0% and (0%-10%] bars represent the SNT class, the (10%-90%) bar represents the WB class, and the [90%-100%) and 100% bars represent the ST class.

It can be seen that, when no global history is used, each of the SNT, ST, and WB classes contributes about 33% of total dynamic branches individually. However, as correlation is used, the normalized dynamic counts of the bias classes change. The more correlation is used, the higher the biases of the substreams become. When a 4-bit global history is used, 25% of total dynamic branches fall into the WB (10%-90%) class. As the history length is increased to 15 bits, only 15% of them fall into the WB class. Employing correlation significantly increases the bias of the substreams.

In general, branches have a tendency to be, either mostly taken or mostly not-taken. However, under some special conditions, branches may behave differently from their normal biased directions. The exiting condition for looping branches is a good example. Fortunately, whenever one of the special conditions occurs, the branch behaves consistently, though differently from the direction which it usually takes. Our study suggests that history information is effective at identifying these special conditions, and whenever the special conditions are identified, the outcomes are easy to predict. Consequently, increasing history bits can potentially improve prediction accuracy.

4.2.2 Bias measurement for global-history schemes

From the discussion above, it can be seen that correlation makes branches more biased. This is the reason that two-level dynamic branch predictors may achieve higher prediction accuracy than the traditional two-bit counter scheme proposed by Smith [46]. However, the performance of the two-level schemes is not always superior. In this subsection, we will identify the performance limitation.

The index for the second-level table divides the dynamic branch stream into substreams that are directed to a saturating two-bit counter. Ideally, the index scheme should generate highly biased substreams so that the value of the saturating counter selected by the index can stay at one of the saturated values most of time. However, if the indexing method mixes oppositely biased substreams together, then destructive interference can arise and the assigned counter will perform badly as a predictor, because it will oscillate between the two saturated values. In this subsection, we extend the experiments in the previous subsection by examining the bias of branch outcome streams for practical branch predictors. We will compare using branch addresses with using global history to separate out oppositely biased substreams, and show how destructive interference can degrade the performance of two-level schemes that use global history.

To contrast the benefits of address versus global history bits, we consider two alternative two-level gshare style predictors. Both have the same size second-level tables but differ in that one employs more history bits, representing history-indexed schemes, while the other represents address-index schemes. The first scheme xors 8 bits of branch address with 8 bits of global history to form the index into the second-level table ("history-indexed"). The second scheme xors 8 bits of branch address with only 2 bits of global history as the index ("address-indexed").

We are interested in the stream of branch outcomes, s_{ij}, from a particular static branch, i, to a particular prediction counter, j (note that the second index, j, now denotes a counter rather than a global history pattern). This stream belongs to one of the three previously defined bias classes, i.e., exactly one of the following is true: $s_{ij} \in$ ST, $s_{ij} \in$ SNT, or $s_{ij} \in$ WB. A good indexing method will create these streams so that the following two conditions hold:

1. The number of streams that are in the WB class are kept small.
2. Most of the streams incident on a particular prediction counter, $j = c$, belong to only the ST class, or alternatively, only the SNT class, i.e., $s_{ic} \in$ ST for most i, or $s_{ic} \in$ SNT for most i. A counter should <u>not</u> see an even mix of streams from both classes or its prediction ability will be reduced.

Condition 2 actually states that one of the two strongly biased class should dominate the other strongly biased class at a counter. When this domination occurs, the counter will be biased at one saturated value with little destructive interference. We will refer to the more frequent strongly-biased class at a counter as the *dominant* class, and the other less frequent strongly-biased class as the *non-dominant* class.

To be more precise, we consider streams weighted by their lengths. If $|s_{ij}|$ is the number of outcomes in the stream s_{ij}, we define the normalized count that a branch, i=b, contributes to a particular prediction counter, $j = c$, to be:

$$N_{bc} = \frac{|s_{bc}|}{\displaystyle\sum_{\text{over all static branches } i} |s_{ic}|}$$

Thus the two conditions become:

1. ($\sum_i N_{ic}$| for those i such that $s_{ic} \in$ WB) << ($\sum_i N_{ic}$| for those i such that $s_{ic} \notin$ WB)
2. ($\sum_i N_{ic}$ | for those i such that $s_{ic} \in$ ST) should differ greatly from ($\sum_i N_{ic}$ | for those i such that $s_{ic} \in$ SNT). In an ideal situation, one of the sums should be 0.

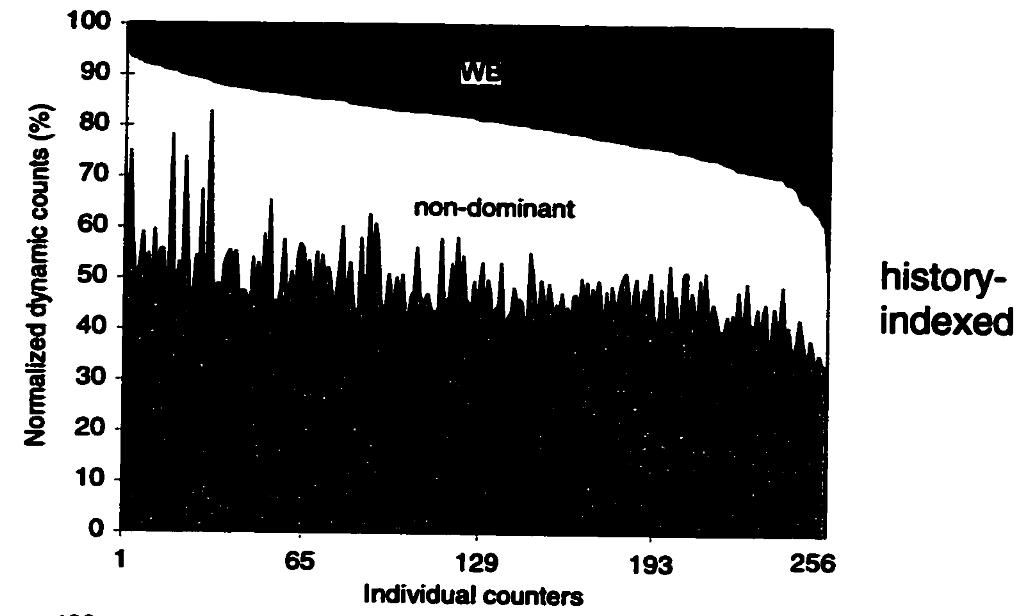
Table 4.2 illustrates the normalized count resulting from three streams incident on the same counter c. In this example, there is a total of four static branches ($i = 4$) whose addresses are 0x001, 0x005, 0x100 and 0x150, respectively, that used the two-bit counter *c* for prediction during the program execution (they may also use other counters too). These four streams fall into different bias classes with respect to *c*. The normalized count of ST class at the counter *c* is 24%, the SNT class is 60% (40%+20%), and the WB class is 16%. Because the SNT class is more frequent than the ST class, the SNT class is the dominant class in the counter *c*, while the ST is the non-dominant class. In fact, Table 4.2

branch address, i	dynamic count when using counter *c*, $\lvert s_{ic} \rvert$	count of taken outcomes when using counter *c*	bias class	normalized count from *i* to *c*, N_{ic}
0x 001	12	11	ST	12/50 = 24%
0x 005	20	1	SNT	20/50 = 40%
0x 100	8	3	WB	8/50 = 16%
0x 150	10	1	SNT	10/50 = 20%

Table 4.2: An example of calculating the normalized count for a counter *c*

shows an undesirable situation because the indexing method has done a poor job of separating the bias classes and the SNT class is not overwhelmingly dominant.

Figure 4.14 illustrates the bias classes for all of the prediction counters for the *gcc* benchmark. We have performed the same experiments for other SPEC benchmarks, but for brevity we show only the *gcc* results because they are the most representative as we have seen before. The X axis lists all the counters in the second-level table, and the Y axis represents the normalized counts of the three bias classes in each counter. The counters listed in the X axis are sorted according to the normalized dynamic frequency of WB class. It can be seen that the area size of WB region of the history-indexed scheme is smaller than that of the address-indexed one. This suggests that the scheme employing



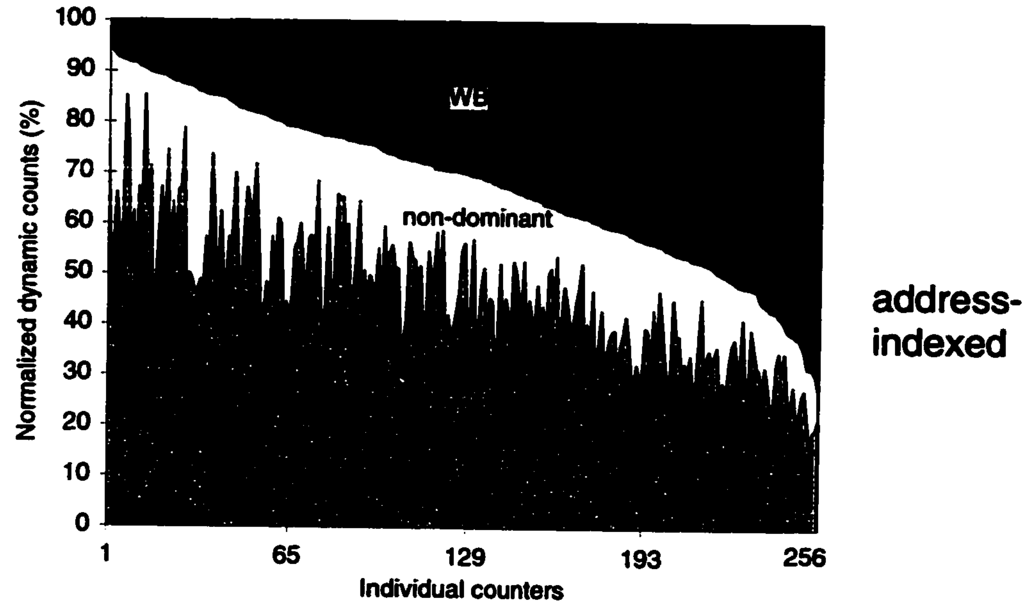


Figure 4.14: Bias breakdown for the gshare scheme for the SPEC CINT95 *gcc*

In this experiment, the branch outcome stream arriving at a counter is first separated into substreams according to the branch address. Each substreams is classified into one of the three bias (SNT, ST and WB) classes based on its probability of being taken. The total dynamic count of each bias class is accumulated. A *normalized dynamic count* of a bias class is then obtained by normalizing the total count of the bias class to the total number of dynamic branches predicted by the counter. Between SNT and ST, the class with a higher normalized count is referred to as the *dominant*. The other is considered *non-dominant*. In a good predictor, each counter should see dominant substreams most of the time.

To contrast the address bits and the global history bits, the top plot uses a gshare scheme xoring 8-bit address with 8-bit global history to represent an "history-indexed" scheme, while the bottom is a gshare scheme xors 8-bit address with 2-bit global history, representing a "address-indexed" scheme. It can be seen that, the address-indexed scheme suffers from the large WB class, while the history-indexed scheme suffers from the interference between the dominant and non-dominant streams.

more branch history can generate more highly biased substreams for predictors. If there is no harmful aliasing problem in the history-index scheme, i.e., each counter only needs to deal with substreams of one bias class, the prediction accuracy will be very high [49, 60].

However, in the usual situation where the harmful aliasing does exist, the performance of the history based scheme can degrade. As shown in the same figure (Figure 4.14), the non-dominant class of the history-indexed scheme is larger than the one in the address-indexed scheme. In other words, although the history-indexed selects the greater number of highly biased substreams, it does not separate the taken and not-taken ones as well as the address-indexed scheme.

To summarize the analysis above, an ideal dynamic branch predictor should generate as few weakly biased substreams as possible; in other words, the area of the weakly biased region should be as small as possible. At the same time, the resulting substreams merged at each counter should be as unidirectional as possible; in other words, the dominant area in Figure 4.14 should be large. Unfortunately, neither the address-indexed scheme nor the history-indexed scheme can achieve both of these two design goals simultaneously.

4.2.3 Lengths of dynamic sequence of the three bias classes

In Figure 4.14 we have seen that the dominant and non-dominant classes occupy roughly equal execution time in the global-history indexed scheme, which suggests a high degree of interference. In this subsection, we confirm the interference problem by measuring the length of a consecutive sequence of branch outcomes from a bias class. The results will show that in the global-history indexed scheme, though there are more branches from the SNT and ST classes, they are divided into smaller dynamic sequences, compared to the address-indexed scheme.

First, we define a *dynamic sequence* as a consecutive sequence of branch outcomes arriving at a counter in program execution order that belong to a bias class, either the SNT, ST, or WB class. For this discussion, we convert the SNT and ST sequences to the dominant and non-dominant sequences based on their dynamic counts at each counter. A dynamic sequence will terminate when a new sequence from a different bias class follows or the program execution stops. Then we measure lengths for all dynamic sequences and collect cumulative dynamic branch counts contributed by sequences, see Figure 4.15. The cumulative count is used instead of a histogram of sequence lengths because total numbers of dynamic branches contributed by sequences also need to be considered.

Two observations can be made from the plots in Figure 4.15. First, the steeper slope of the history curve in the top graph indicates that the history-indexed scheme has shorter dominant sequences than the address-indexed scheme. The slope represents the incremental amount of dynamic branches contributed by sequences whose lengths are equal to the X value. Second, the position of the history curve above the address curve in the middle graph indicates that the history-indexed scheme has significantly more non-dominant branches than the address-indexed scheme. These two observations provide further support for the case that the history-indexed scheme suffers most from the interference between oppositely biased branch streams.

Finally, from the lower plot for the WB sequences, we can see that address-indexed scheme has more, and longer, WB sequences than the history-indexed scheme. WB sequences should be as few and as short as possible in a good predictor, because they

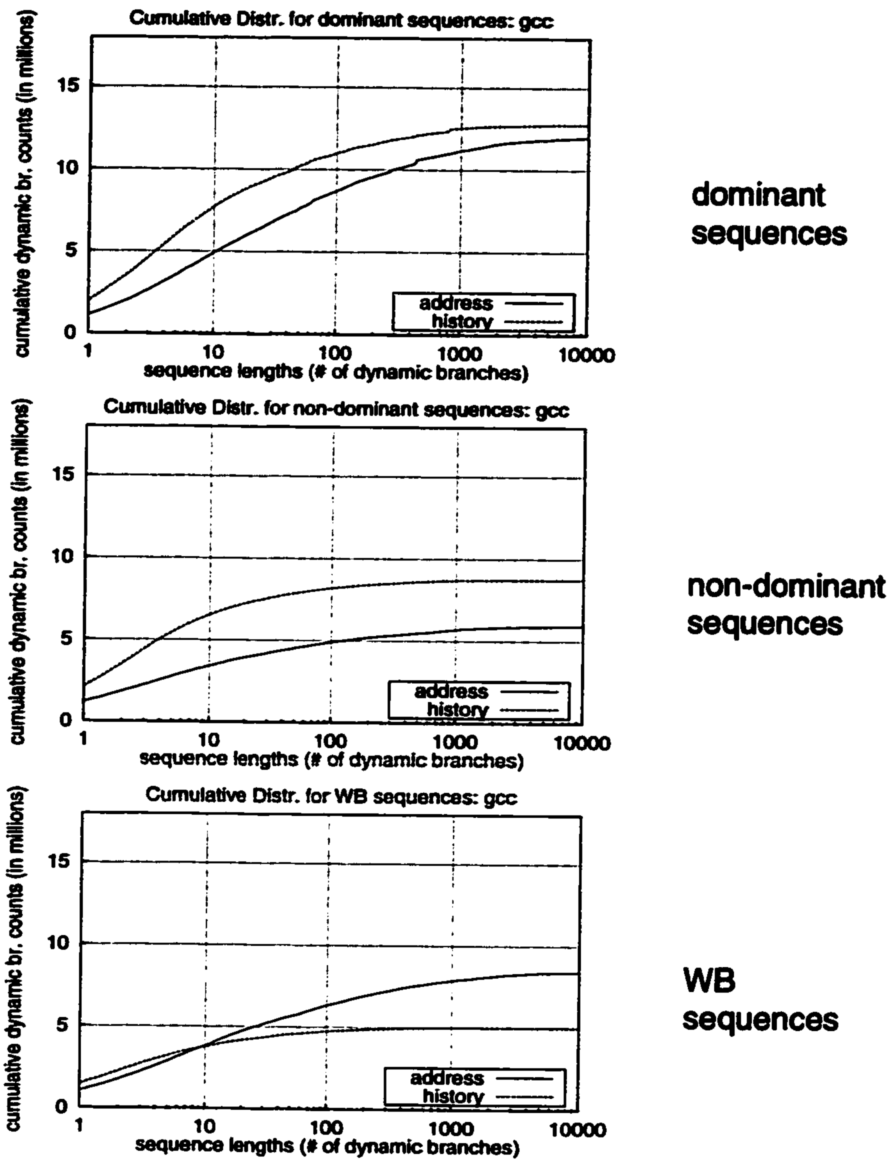


Figure 4.15: Cumulative distribution of sequence lengths for the three bias classes — SPEC CINT95 *gcc*

In this experiment, the lengths of dynamic sequence of three bias classes, dominant, non-dominant and WB, are measured for *gcc*. Two schemes, an address-indexed and a history-indexed, are examined. A dynamic sequence is a consecutive sequence of branch outcomes arriving at a counter in program order that belong to a bias class. A dynamic sequence will not terminate until it is interrupted by another new sequence of different bias class or the program execution stops. The plot on the top is cumulative dynamic branch counts for the dominant, the middle for the non-dominant, and the bottom for the WB. Each point in a curve of a plot represents the total dynamic branches (Y-axis value) contributed by all the sequences that are equal to or smaller than the length specified by the X-axis value. Sequences which are longer than 10,000 dynamic branches are grouped together due to space restriction.
It can be seen that the history-indexed scheme has significantly more non-dominant branches (see the points of the non-dominant curves at X=10,000) and it also has shorter dominant sequences (see the slops of the curves). Therefore, the history-indexed scheme suffers most from the interference between oppositely biased streams. On the other hand, the address-indexed scheme suffers most from the large amount of WB sequences (see the highest point of the WB curve).

are usually hard to predict.

4.3 Summary

This chapter provides a thorough investigation into the design of global history based two-level dynamic branch predictors. A comprehensive performance evaluation is first conducted with a broad range of benchmarks, SPEC CINT95, IBS, and a database benchmark, *sql95*. This is followed by a study into the effects of correlation on the branch prediction.

The global performance picture that is developed from the extensive simulations of a variety of configurations shows that the accurate prediction of large programs depends primarily upon the deployment of sufficient resources to keep track of information of a large number of branches. Performance results from small footprint benchmarks usually lead to misleading conclusions.

The global performance picture from our evaluation work can be used by architects to select the optimal design of global history schemes. On the other hand, the study also shows the shortcomings of such schemes, which can instruct researchers how to further improve the prediction accuracy. Correlation's strength is that it can sort the branch outcome streams so that multiple highly biased substreams are generated which are easy to design predictors for. However, in global history schemes, the resulting highly biased streams are poorly distributed, resulting in the severe interference and attendant performance degradation. For good global history schemes, controlling interference is a key to improving prediction accuracy while taking advantage of correlation. A scheme proposed to reduce interference by randomizing global history patterns, gshare, has been studied, and it shows that this randomization is hardly better than using branch address bits alone. In the next chapter we will examine ways of more effectively reducing interference for the global history schemes.

As we have seen, it is important to recognize the position of branch prediction schemes discussed within the larger space of possibilities, lest resources be misapplied. In their widely-used textbook, Hennessy and Patterson [24] make the statement, regarding GAs, that "the attraction of this type of correlating branch predictor is that it can yield higher prediction rates than the two-bit scheme and requires only a trivial amount of hardware." We have shown the degree to which this attractive element is limited for large programs. Past branch prediction results require a more careful interpretation; in particular, it needs to be recognized that the benefits of correlation can be easily drowned by destructive interference.

CHAPTER 5

Reducing Interference in Global History Schemes: The Bi-Mode Scheme and Other Designs

As we have seen, there are two design criteria for an optimal dynamic branch predictor: the predictor should generate as many highly biased branch streams as possible, and the predictor should separate those highly biased branch streams to minimize the collision of oppositely biased streams at a counter.

However, as we have also seen, the conventional two-level global history schemes cannot achieve these two design criteria simultaneously. For example, if a scheme use more branch address bits than history bits, referred to as an address-indexed scheme in the previous chapter, it tends not to produce the highly biased streams needed for better prediction; on the other hand, the history-indexed schemes that use a large number of history bits, suffer from interference between oppositely biased streams.

In this chapter, we propose a scheme, the bi-mode scheme, which solves this problem and, as a consequence, improves prediction accuracy. This chapter is devoted to a detailed discussion of bi-mode dynamic branch predictors.

5.1 The bi-mode scheme

To reduce interference in global history indexed schemes, we propose a new scheme, the bi-mode branch predictor. This scheme, shown in Figure 5.1, splits the

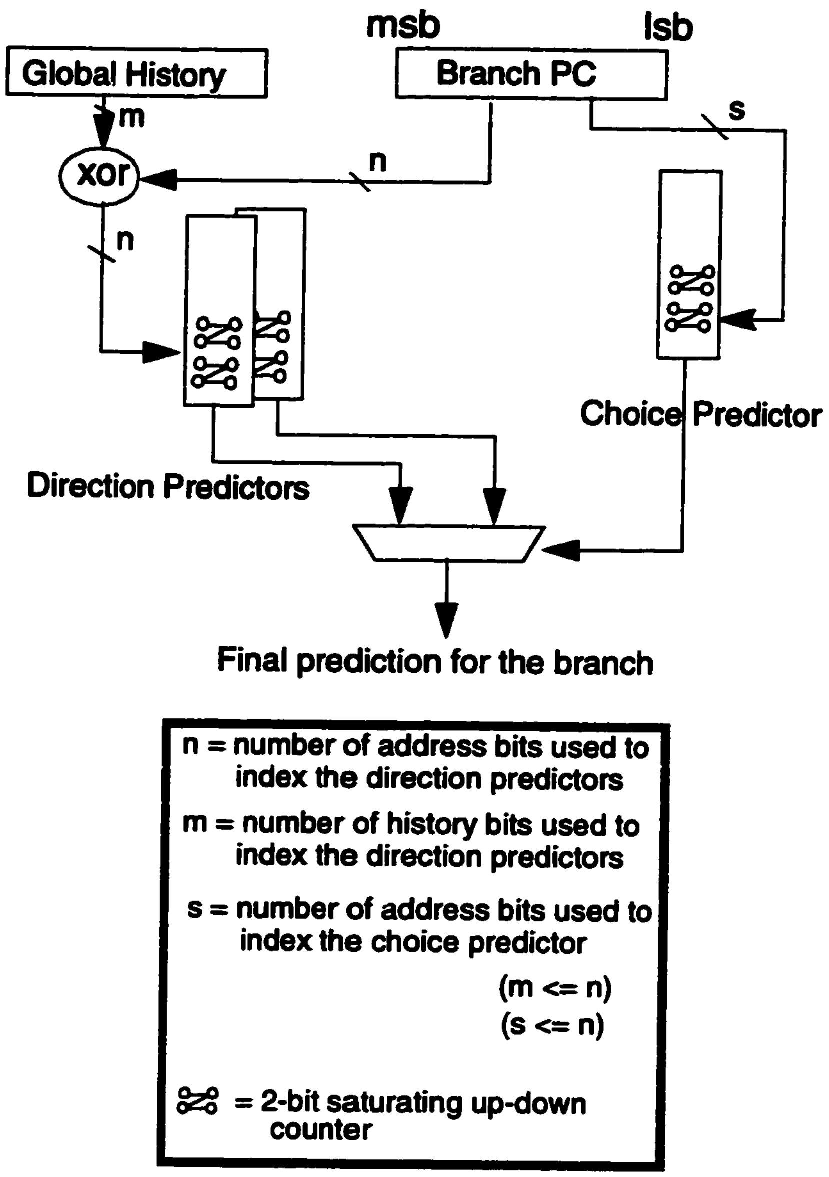

msb
lsb
Global History
Branch PC
m
xor
n
n
s
Choice Predictor
Direction Predictors
Final prediction for the branch
n = number of address bits used to index the direction predictors
m = number of history bits used to index the direction predictors
s = number of address bits used to index the choice predictor
(m <= n)
(s <= n)
= 2-bit saturating up-down counter

Figure 5.1: Proposed branch prediction scheme diagram

This diagram illustrates the bi-mode scheme. The second-level table is divided into two halves, which are referred to as direction predictors. In addition to the direction predictors, a choice predictor is also added. To predict a branch, two counters, each from a direction predictor, are accessed. The choice predictor then determines which of the two counters should be used for the final prediction. In this illustration, the choice predictor is indexed by the branch address, while the direction predictors are accessed by xor of the global history and branch address. As a variant of the bi-mode scheme, the choice predictor can also be indexed by xor of the global history and branch address, which will be discussed later.

second-level table of a gshare into two halves. Given a history pattern, two counters, one from each half, are selected. We refer to these as the direction predictors. Another two-bit counter table, indexed by the branch addresses only, is used to provide a final selection for these two counters. We refer to this as the choice predictor. The final prediction is determined by the state of the counter selected from the direction predictors and, equally importantly, only the selected counter will be updated with the branch outcome; the status of the un-selected one, will not be altered. The choice predictor is always updated with the branch outcome.

As we will show, our proposed scheme can perform better than other existing global history based branch predictors, because, although the behavior of global history patterns are still kept in the second level table, they are dynamically classified before being stored to further reduce distinctive interference. The global history patterns are classified by a preliminary prediction from the choice predictor which is simply a conventional two-bit counter scheme, and, as such, typically can provide 80% or better prediction accuracy with relatively modest cost. Thus, the bi-mode scheme divides branches into two groups according to the per-address bias of the choice predictor, and then uses the global history patterns to identify the special conditions for each of two groups separately. The two groups of direction predictor counters correspond to the strongly-taken and strongly-not-taken cases. The effect of the choice predictor is to separate the destructive interference streams while keeping the harmless interference streams together.

5.1.1 Experiment Results

In this section, we demonstrate that our proposed bi-mode branch predictor delivers higher prediction accuracy than the conventional two-level predictors by reducing the interference significantly. One of the best two-level branch predictors, gshare, has been selected to represent the two-level predictors in our comparison. (Refer to Chapter 4 for the detailed analysis of gshare.) To evaluate the improvement, we have conducted trace-driven simulations. As input for the simulation, we again use the IBS benchmarks and the SPEC CINT95 benchmark suite, described in Chapter 3.

According to the model in Chapter 4, in gshare the lowest order address bits of a branch are used as the column index into the second-level table, and the global history is xor-ed with the next low-order address bits to form the row index. This row index is then used to select a 2-bit saturating up-down counter from a column of the second level table, or a pattern history table (PHT). Depending on the sign bit of the selected 2-bit counter, the branch is either predicted as taken or not taken.

To make a fair comparison with the gshare predictor, the best configuration of

2nd-level table sizes: (2^n counters)		n=11	12	13	14	15	16
# of global history bits employed	SPEC CINT95	3	6	6	9	10	13
	IBS-Ultrix	2	4	8	9	10	11

Table 5.1: The gshare configurations that yield the best average prediction accuracy (gshare.best).

gshare must be determined and used. This point is often overlooked and the single-column (1 PHT) gshare configuration is used for comparisons. However, this single-column gshare configuration is not the optimal configuration as was shown in Chapter 4. To find the best configuration, we exhaustively simulated all pair-wise combinations of history

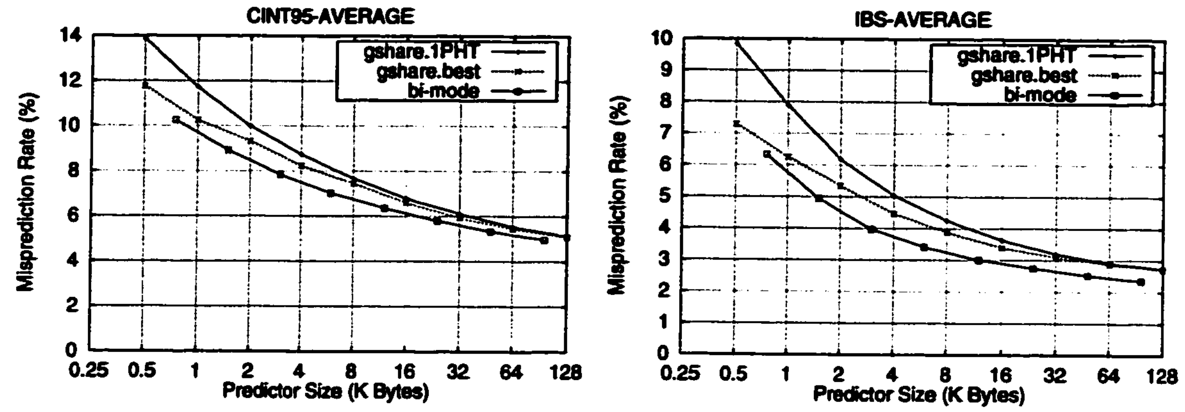


Figure 5.2: Averaged misprediction rates — SPEC CINT95 (left) and IBS-Ultrix (right)

length and address length. In general, the best combination has multiple columns, as listed in Table 5.1. Since the best configuration is different for each benchmark, we present results using the configuration that yields the best accuracy for the average of all the benchmarks studied.

Figure 5.2 shows the misprediction rates for the best gshare and bi-mode predictors. We label the best configurations of gshare as *gshare.best*. For comparison with other published results, we also include the misprediction rates for the single-column (1 PHT) gshare configuration, which is labeled *gshare.1PHT*. In Figure 5.2, the vertical axis represents the branch misprediction rate, and the horizontal axis for the size of predictors. A lower curve indicates that the scheme has better performance for the same cost. Cost is measured by counting the number of bytes used in the 2-bit counters. Note that the bi-mode predictors naturally have a cost that is 1.5 times that of the next smaller gshare scheme. This reflects the additional cost of the choice predictors.

Figure 5.2 shows that the bi-mode predictors outperforms gshare predictors for all sizes of predictors measured. This is indicated by lower curves. In addition, the bi-mode predictors are more cost effective, because, for predictors larger than 4K bytes, they need only be half the size of gshare predictors to achieve the same misprediction rate.

Bi-mode predictors also outperform gshare on most of the individual benchmark examined, see Figure 5.3 and Figure 5.4. Moreover, the single-column gshare scheme (*gshare.1PHT*) is worse than the multiple-column gshare scheme for all benchmarks except the *compress* and *xlisp*, where it outperforms even the bi-mode scheme. These two benchmarks, with the fewest static branches, have no interference problems and thus can enjoy the benefits from correlation in branch histories. The results of these two small benchmarks correspond to the findings shown in Chapter 4. The case of the *go* benchmark, where the bi-mode method is worse than the multiple-column, will be discussed in more detail later in this chapter.

5.1.2 Bias measurement for the bi-mode scheme

In this subsection, the bias measurement for the bi-mode prediction scheme is made by following the method developed in Section 4.2.2 of Chapter 4 (see Figure 4.14). The configuration under examination has a 128-counter choice predictor indexed by the branch address and two banks of 128 counters in the second-level table, each of which is indexed by 7 bits of branch address xor-ed with 7 bits of global history.

Figure 5.5 presents the measurement results. As shown in the figure, the weakly biased class region in the bi-mode scheme is kept as small as the one in the history-indexed scheme, indicating that the advantage of employing history information is preserved. On the other hand, the bi-mode scheme has a larger area for the dominant mode than the history-indexed scheme implying that the destructive interference has been reduced. This illustrates why our proposed prediction scheme performs better than the

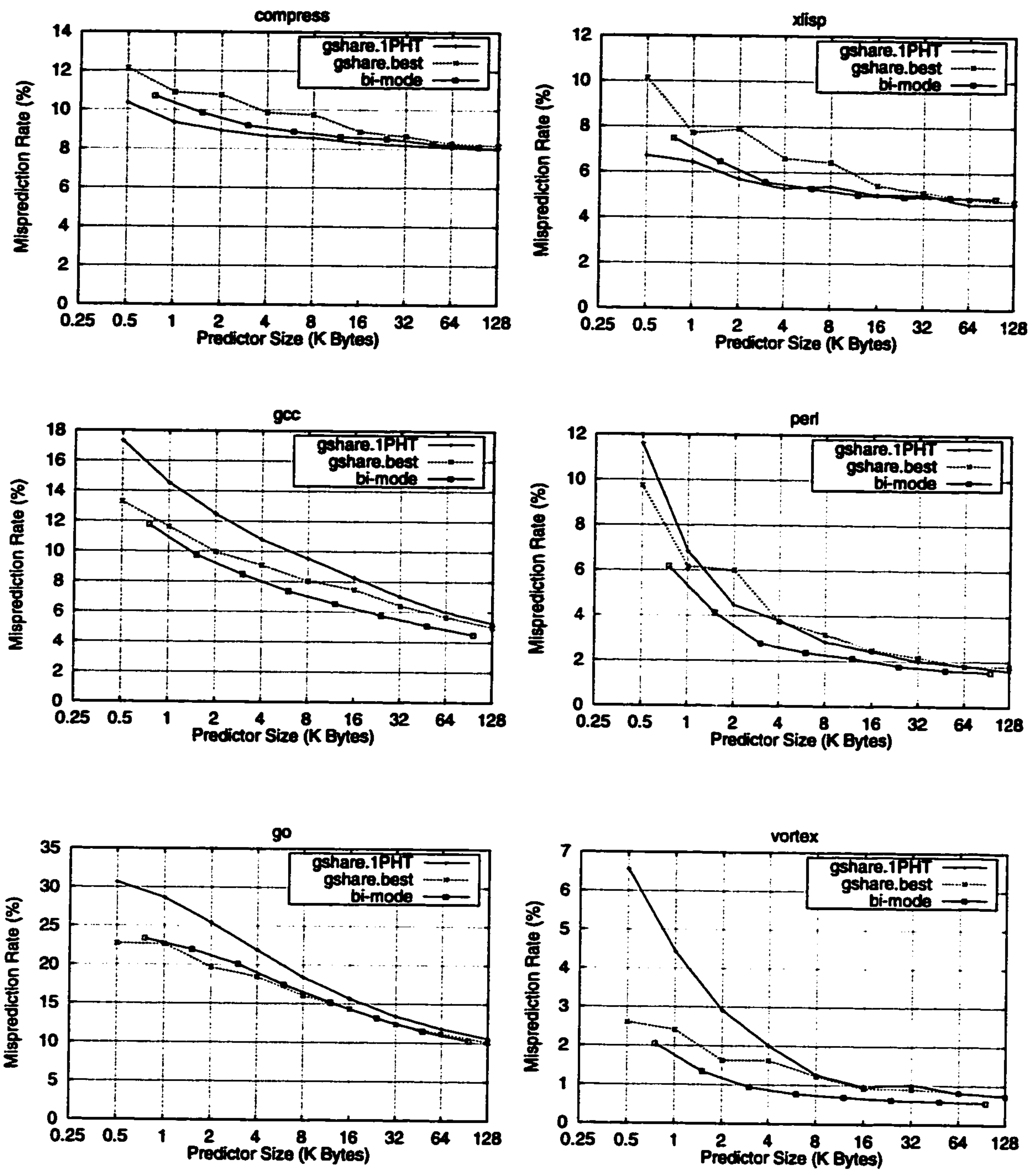


Figure 5.3: Misprediction rates for SPEC CINT95

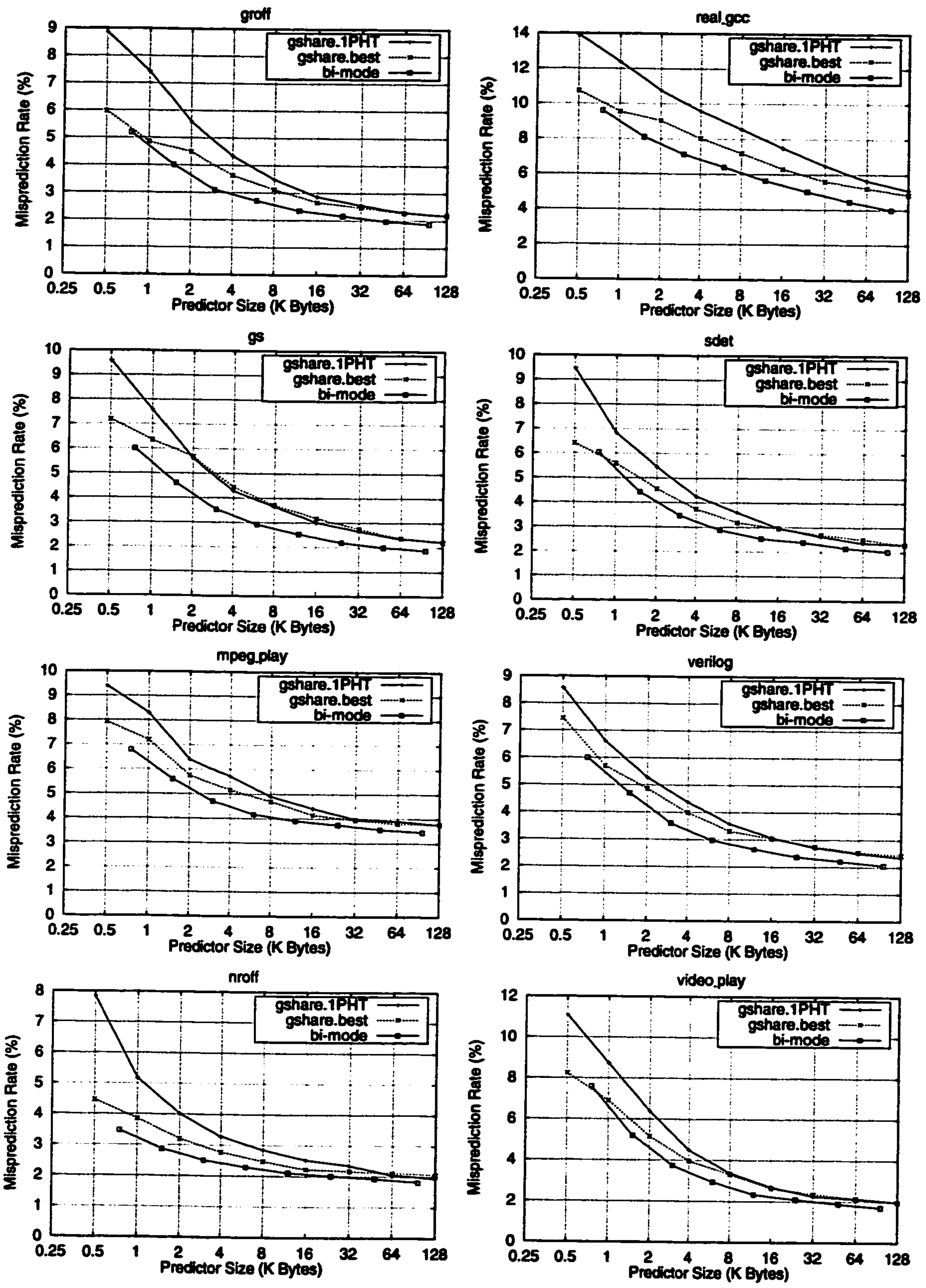


Figure 5.4: Misprediction rates for IBS-Ultrix

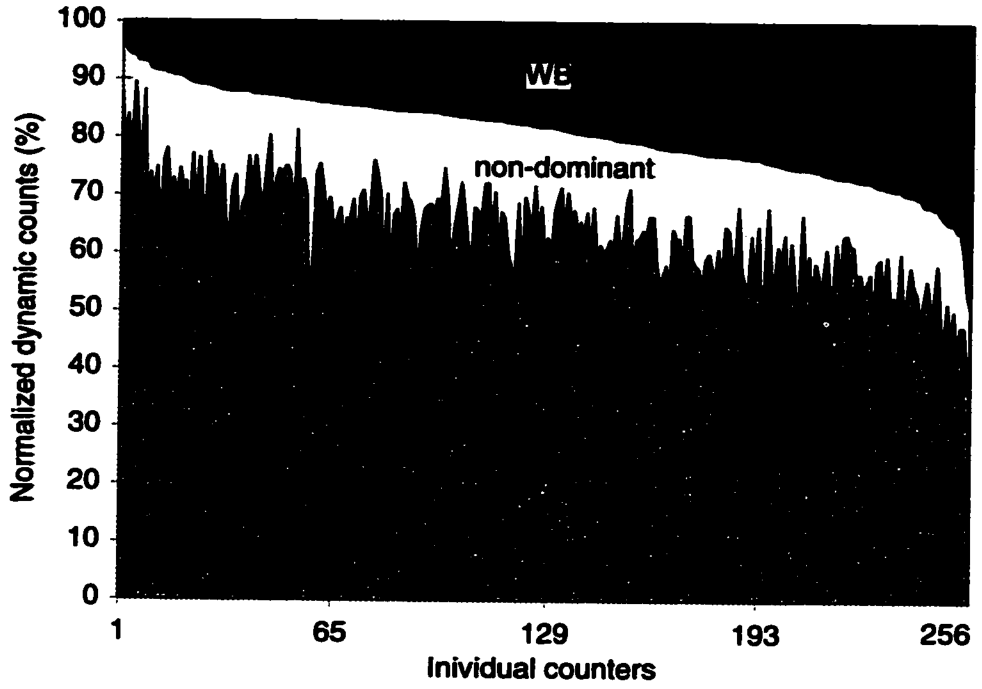


Figure 5.5: Bias breakdown for the bi-mode scheme for the SPEC CINT95 *gcc*

The experiment shown in Figure 4.14 in Chapter 4 is repeated for the bi-mode scheme. In this experiment, the branch outcome stream arriving at a counter is first separated into substreams according to the branch address. Each substreams is classified into one of the three bias (SNT, ST and WB) classes based on its probability of being taken. The total dynamic count of each bias class is accumulated. A *normalized dynamic count* of a bias class is then obtained by normalizing the total count of the bias class to the total number of dynamic branches predicted by the counter. Between SNT and ST, the class with a higher normalized count is referred to as the *dominant*. The other is considered as *non-dominant*. In a good predictor, each counter should see dominant substreams most of the time. In this experiment, the choice predictor of the bi-mode scheme has a 128-entry counter table, and the direction predictor is a gshare scheme that xors 7 address bits with 7 global history bits.
Comparing with Figure 4.14 we can find that in the bi-mode scheme, the dominant mode is more pronounced in most of the counters, suggesting that counters have much less destructive interference. This illustrates why the bi-mode scheme can deliver higher prediction accuracy than conventional global history schemes.

conventional two-level schemes as shown above.

5.1.3 Lengths of dynamic sequence of the three bias classes

In Section 4.2.3, we measured lengths for dynamic sequences of the three bias classes for the history-indexed and address-indexed schemes. (A *dynamic sequence* is a consecutive sequence of branch outcomes arriving at a counter in program execution order that belong to one of the bias classes, SNT, ST, or WB, defined in Section 4.2.3.) From that result, we found that the global history scheme has more but shorter dominant sequences than the address-indexed scheme. The shorter sequences are due to a higher degrees of interference. It is important for the dominant sequences to be *long* to achieve high prediction accuracy. On the other hand, we found that the address-indexed scheme has more and longer WB sequences. This is the key weakness of the address-indexed scheme, since WB sequences are hard to predict.

In Figure 5.6, we compare sequence lengths of the bi-mode scheme and the two schemes. From the plot for the dominant class, we can see that the bi-mode scheme has the highest number of dominant sequences and the lowest number of non-dominant and WB sequences. The dominant sequences in the bi-mode scheme are also on average longer than the history-indexed scheme (see the slopes of the curves). If we compare with Figure 4.15, keeping in mind that bi-mode predictors have about 50% more bytes, it can be seen that interference is further reduced by the bi-mode scheme.

5.1.4 Breakdown of misprediction for the bi-mode schemes

Continuing the analysis, we conducted another experiment to measure the misprediction contributed by the three biased classes for the gshare and bi-mode schemes. Again, for the gshare scheme, the configurations using fewer global history bits and more global history bits are both included for comparison.

Figure 5.7 presents the measurement results for the SPEC CINT95 *gcc* benchmark. Three different sizes are studied for the branch predictors: 256, 1024, and 32,768 counters in the second level table. For each configuration, the misprediction is broken down to

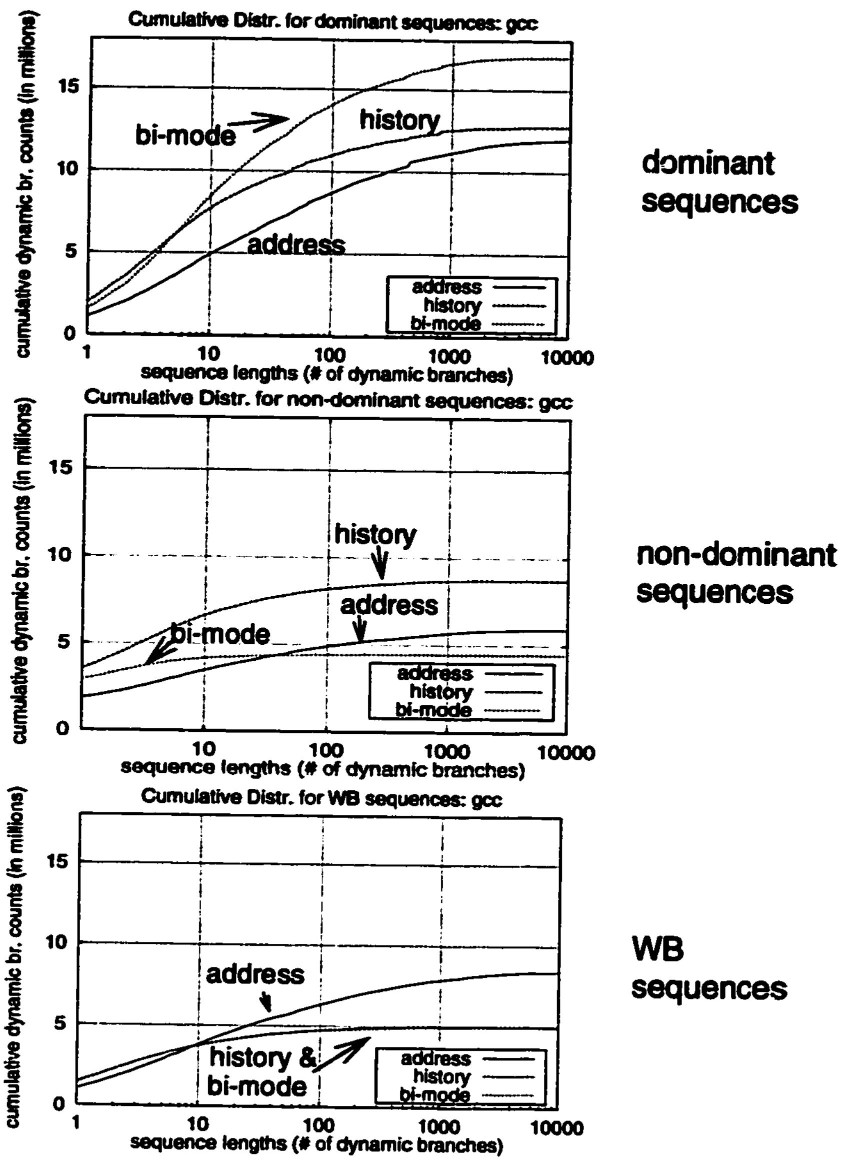


Figure 5.6: Cumulative distribution of sequence lengths for the three bias classes — SPEC CINT95 *gcc*

In this experiment, the lengths of dynamic sequence of three bias classes, dominant, non-dominant and WB (see Figure 5.5), are measured for *gcc*. Three schemes are examined, an address-indexed, a history-indexed and a bi-mode scheme. A *dynamic sequence* is a consecutive sequence of branch outcomes arriving at a counter that belong to a bias class. A dynamic sequence will not terminate until it is interrupted by another new sequence of different bias class or the program execution stops. The plot on the top is cumulative dynamic branch counts for the dominant class, the middle is for the non-dominant, and the bottom is for the WB. Each point in a curve of a plot represents the total dynamic branches (Y-axis value) contributed by all the sequences that are equal to or smaller than the length specified by the X-axis value. Sequences which are longer than 10,000 dynamic branches are grouped together due to space restriction.

It can be seen that the bi-mode has the highest number of dynamic branches from the dominant class, and they are in longer sequences too (sleeper slope). The history-indexed scheme is most impacted by non-dominant sequences while the address-indexed scheme suffers most from the WB sequences.

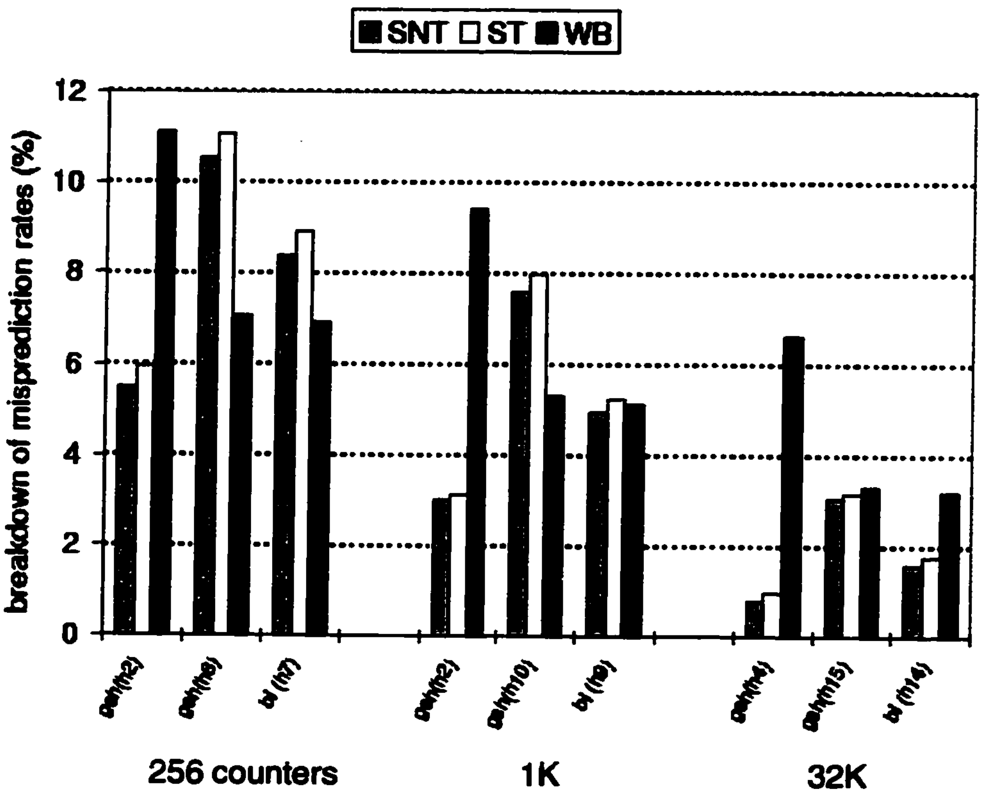


Figure 5.7: Misprediction rates contributed by three bias classes for *gcc*

In this figure, misprediction rates are broken down according to the three bias classes for three schemes. The three schemes include an "address-indexed" scheme, a "history indexed" scheme and a bi-mode scheme. For each of the schemes, three different sizes of the second-level table are examined: 256, 1,024, and 32K counters. The notation, gsh (h*m*), represents a gshare scheme that xors *m* global history bits with branch addresses to form the index. Thus, for two gshare schemes of the same size, the one with fewer history bits represents the "address-indexed" scheme while the one with more history bits is the "history-indexed" scheme. The bi-mode scheme always uses same numbers of counters in the second-level table as the "history-indexed" scheme; the size of its choice predictor is half of the second-level table. This experiment attempts to provide qualitative rather than quantitative comparison between these three schemes because the size of the bi-mode scheme is always 1.5 times of the next smaller gshare scheme. As shown in the figure, most of the error in the "address-indexed" scheme is due to WB streams, while error of the "history-indexed" scheme is mostly due to the destructive interference between ST and SNT streams. The bi-mode scheme can reduce destructive interference between ST and SNT streams while keeping smaller amounts of error due to WB streams.

three categories according to the bias classes. The sum of misprediction from three classes is the misprediction rate for the corresponding scheme. For the gshare predictors of the same size, the one using fewer global history bits always has the least error from the strongly-biased classes, but it suffers from poor prediction for the weakly-biased substream. The bi-mode scheme maintains a reduced error for the weakly biased class, while successfully reducing the error from strongly-biased classes for global-history based scheme. This experimental result further supports the observation made in the previous subsection.

5.1.5 *go* benchmark

In Section 5.1.1, we noted that the bi-mode scheme was not the best for the *go* benchmark. In this section, we provide further analysis.

The *go* benchmark is intrinsically hard to predict because about half of its dynamic branches are WB branches. Figure 5.8 illustrates the normalized dynamic counts of three bias classes. (Compare the measurement results for the *gcc* benchmark in Figure 4.13.) From the figure, we can see that the WB region dominates in the *go* benchmark. Figure 5.9 shows the misprediction contributed by three bias classes for the *go* benchmark. It is clear that for all the schemes and configurations the misprediction for the WB class dominates—destructive aliasing is not the major concern. Therefore, there is not much room for the bi-mode scheme to improve performance because its strength is eliminating destructive interference rather than improving prediction for weakly biased substreams. As observed in the previous chapter, the normalized dynamic counts of the weakly biased class is mainly determined by the number of global history bits used. From Figure 5.9, we see that the error due to the WB class is reduced as more global history bits are applied. Accordingly, the right approach to improving the prediction accuracy for the *go* benchmark is to incorporate more global history information so that more strongly biased substreams can be generated.

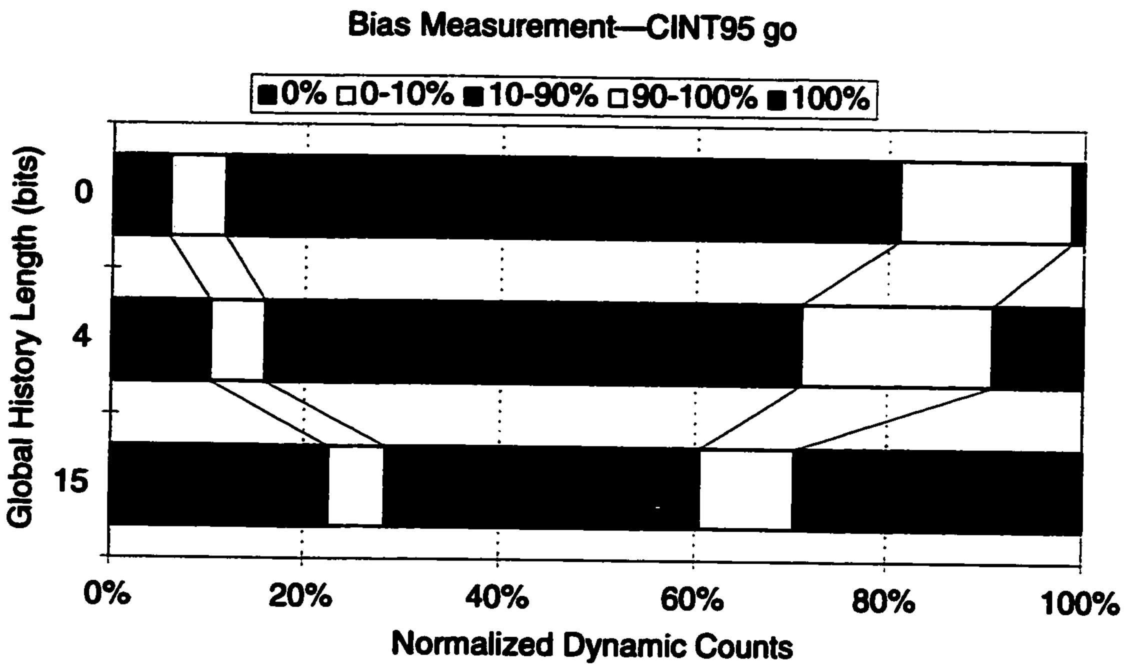


Figure 5.8: Normalized dynamic counts of three bias classes for the *go* benchmark

This graph shows that branches in the *go* benchmark are intrinsically hard to predict unless an significant number of global history bits are used. We first measure the bias (probability of being taken) for the outcome stream of each (address, history) pair. According to its bias, each (address, history) pair is classified into one of the five bias classes (0%, 0-10%, 10-90%, 90-10%, and 100%). The dynamic count of each bias class is accumulated and normalized to the total count of dynamic branches in the program. The stack bars from the top to the bottom represent different global history lengths, 0, 4, and 15 bits, respectively. When global history bits are fewer than 4, more than 50% of the dynamic branch streams are weakly biased, illustrating the difficulty of branch prediction for *go*.

5.2 Other techniques of controlling interference

The interference problem in the global history schemes has been the subject of recent research, and there are several proposals, in addition to the bi-mode scheme, to address the problem. In this subsection, three proposals are reviewed briefly and a quantitative comparison is made between them and the bi-mode scheme.

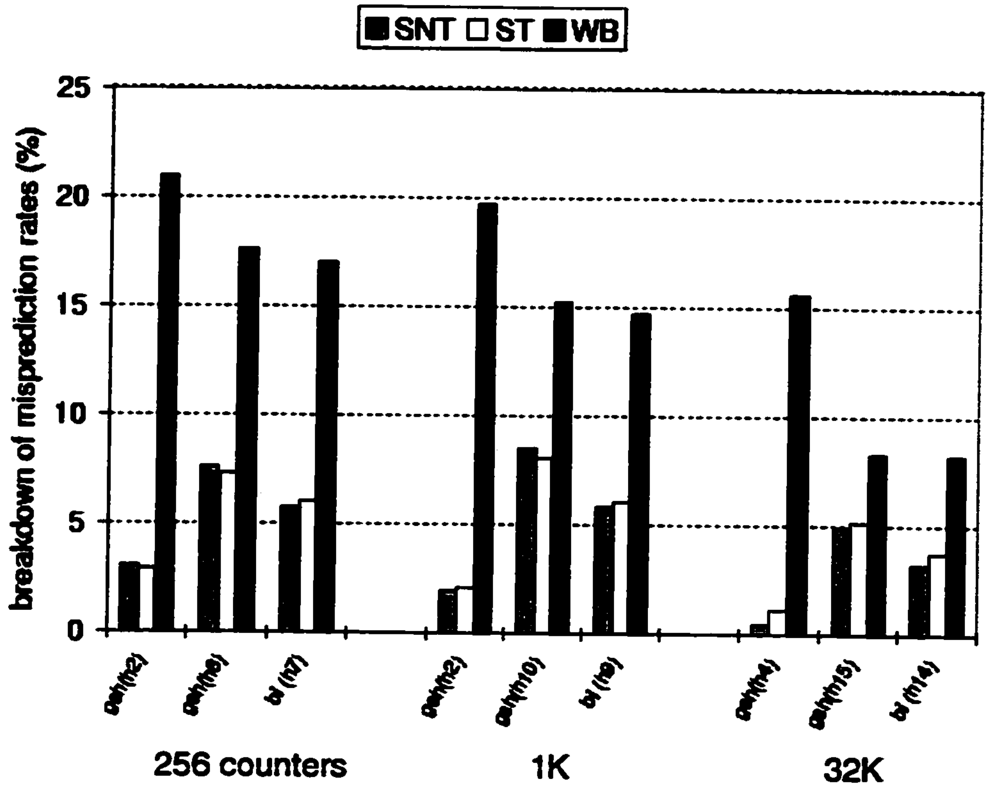


Figure 5.9: Misprediction rates contributed by three bias classes for *go*.

In this figure, misprediction rates are broken down according to the three bias classes for three schemes. These three schemes include an "address-indexed" scheme, a "history indexed" scheme and a bi-mode scheme. For each of the schemes, three different sizes of the second-level table are examined, including 256, 1,024, and 32K counters. The notation, gsh (h*m*) represents a gshare scheme that xors *m* global history bits with branch addresses to form the index. Thus, for two gshare schemes of the same size, one with fewer history bits represents the "address-indexed" scheme while the other one with more history bits is for the "history-indexed" scheme. The bi-mode scheme always uses same numbers of counters in the second-level table as the "history-indexed" scheme and its choice predictor is half of the second-level table size. This experiment attempts to provide qualitative rather than quantitative comparison between these three schemes because the size of the bi-mode scheme is always 1.5 times of the next smaller gshare scheme.
As shown in the figure, the misprediction due to the WB class dominates in all schemes. To improve prediction accuracy for *go*, the use of more global history bits is required, because history can reduce WB streams.

5.2.1 Agree Predictor

The agree predictor reduces interference for the global history predictors by changing the way of using two-bit counters in the second-level table. It has been reported by Sprangle *et al.* [48]. Earlier, Hewlett Packard had implemented a similar branch

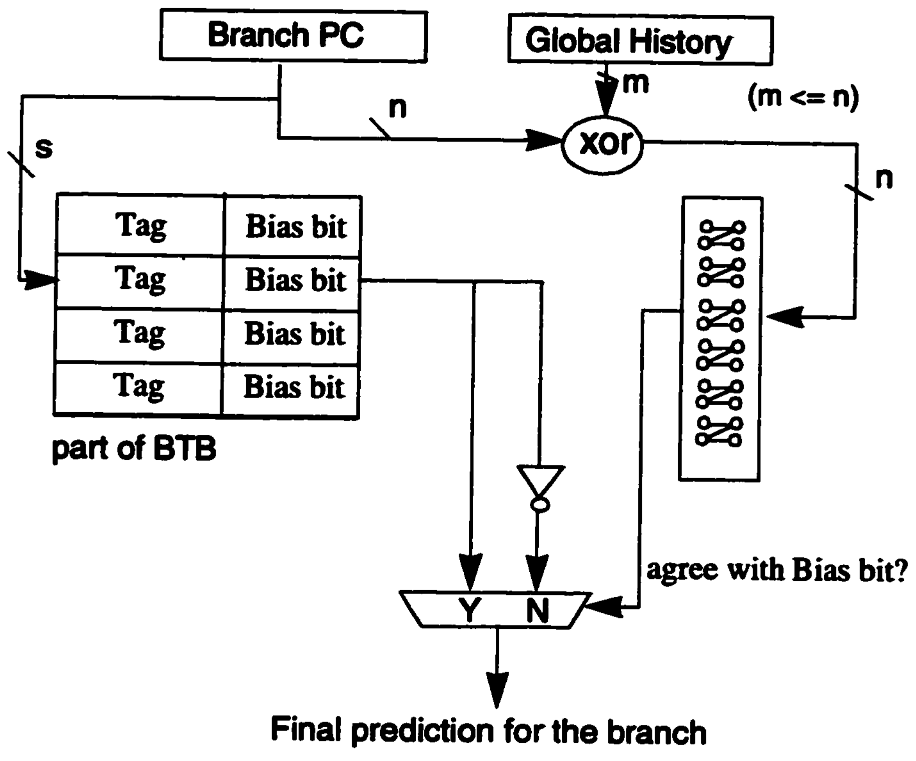


Figure 5.10: Diagram for the agree predictor

predictor in one of its recent microprocessors, the HP8500 [30]. Both claim to reduce the interference significantly.

As discussed in previous subsections, a two-bit counter of the second-level table is used to predict whether a dynamic branch instance is take or not. When the branch outcome is resolved, the counter value is updated with the true outcome. For example, the counter can be decreased by one if the outcome is not-taken and be increased by one if it is taken. The authors of the agree predictors realized that if two oppositely biased branches are aliased to the same counter, the counter value will bounce back and forth between two saturated values. Therefore, they proposed the agree predictor, as shown in Figure 5.10, in which each static branch is assigned a bias bit, possibly stored in the branch target buffer or in the cache, and, rather than predicting the direction for a branch, the two-bit counters predict if the branch will go in the direction indicated by the bias bit. In other words, the counter in the second-level table has now to *agree* that the bias bit should be used for predicting the outcome of the current dynamic branch instance. If two strongly but oppositely biased branches alias to the same counter they will not cause destructive interference because, if the bias bit is set correctly, the aliased counter should agree that both the bias bits associated with the branches are correct.

However, good predictions now depends on setting the bias bits correctly. Ideally, the bias bit should indicate the direction that the corresponding branch takes most of time during program execution. To obtain the optimal value for this bias bit requires pre-running the program. Sprangle *et al.* have examined a technique in which the bias bit can be estimated by the first outcome of each static branch. Their results show that an agree predictor using the first outcome as the bias bit estimator can deliver prediction accuracy that is only 1% less than that with an optimal bias bit. This first outcome as the bias bit estimator will be used for the agree predictor examined in this dissertation.

5.2.2 Skewed branch predictor

Another scheme to reduce interference, referred to as the skewed branch predictor, has been proposed by Michaud *et al.* [35]. Michaud *et al.* have found that the interference in the second-level table is similar to the conflict miss in a cache system. To reduce conflicts for the predictor, they proposed a reorganization of the second-level table. Specifically, the original second-level table is divided into three sub-tables, all three of which record counts for the histories and make prediction for every branches, as shown in Figure 5.11. The final prediction of the predictor is then based on the majority vote among

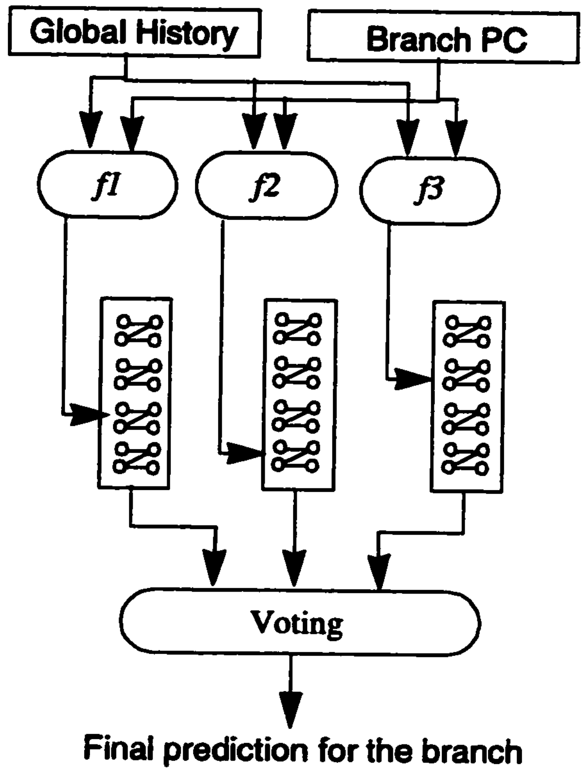


Figure 5.11: Diagram for the skewed predictor

the three sub-tables. The key ingredient in this predictor is that the three sub-tables use three different indexing functions so that the chance for two branches to collide in all three sub-tables at the same time is very small. A potential drawback is that using three sub-tables to record history for each branch may reduce the effective capacity of the second-level table in a skewed predictor. To compensate for this capacity loss, Michaud *et al.* adopted a partial update policy that will not update the incorrect sub-table when a prediction is found to be correct. In other words, when a prediction is finally known to be correct (so at least two of the three sub-tables voted correctly) and there is a sub-table which made an incorrect vote, the incorrect sub-table will not be updated with the branch outcome. Michaud *et al.* realized that when a final prediction is correct, the sub-table which made an incorrect vote may actually be used for recording history for another branch. Therefore, this sub-table should remain unchanged so that it can keep the history information for the other branch. They showed that this partial update policy can compensate for the capacity loss and thus improve the prediction accuracy significantly.

The skewed predictor has been shown to be an improvement over the conventional global history schemes. However, the three indexing functions need to be carefully selected to achieve the goal of reducing interference. Michaud *et al.* have found a scheme that is not costly to implement, but one that may still lengthen the clock cycle because of the address decoder for the sub-tables. The indexing functions used for the skewed predictor examined in this dissertation will be the same as proposed by Michaud, which are described next.

Supposed there is a $2n$-bit index vector, V, which is formed by concatenating the address (c bits) and history bits (r bits, $c+r = 2n$). This V is decomposed into two n-bit vectors, $V1$ and $V2$. $V1$ is the low-order half of V, while $V2$ is the upper half, i.e., both V1 and V2 are n-bit vectors and one of them can contain address and history bits if c and r are not equal to n. V is then equivalent to the concatenated vector $(V2, V1)$. Let $(y_n, y_{n-1}, \ldots, y_1)$ be the bit representation of an n-bit vector. A hash function, H, is defined as follows,

$$H: (y_n, y_{n-1}, \ldots, y_1) \rightarrow ((y_n \text{ xor } y_1), y_n, y_{n-1}, \ldots, y_3, y_2),$$

and its inverse, H^{-1}, is defined as:

$$H^{-1}: (y_n, y_{n-1}, \ldots, y_1) \rightarrow (y_{n-1}, y_{n-2}, \ldots, y_2, y_1, (y_n \text{ xor } y_{n-1}))$$

Three indexing functions are then defined as follows, using bit-wise xors as mappings of $2n$-bit vectors to n-bit vectors,

$$f1: (V2, V1) \rightarrow H(V1) \text{ xor } H^{-1}(V2) \text{ xor } V2$$

$$f2: (V2, V1) \rightarrow H(V1) \text{ xor } H^{-1}(V2) \text{ xor } V1$$

$f3: (V2, V1) \rightarrow H^{-1}(V1) \text{ xor } H(V2) \text{ xor } V2$

In these three indexing functions, some index bits require three xor operations. This increases the access time to the second-level table.

Though three different hashing functions are important to reduce chances of interference, the partial updating of the sub-tables is key to the performance of the skewed predictor. When the branch outcome is resolved and the predictor is correct, only the sub-tables that make correct predictions are updated. In contrast, when the prediction is not correct, all sub-tables are updated. This selective update policy is crucial to the prediction accuracy of the skewed predictor, but it may complicate the logic design.

5.2.3 Filtering

Chang *et al.* have also proposed a scheme to reduce interference for the two-level scheme [11]. Their idea is based on the observation that most branches are predicted accurately by their last outcome. They proposed a scheme, shown in Figure 5.12, that adds an n-bit counter for each static branch to record the number of times the same outcome is repeated. If a branch has 2^n repeating outcomes that are the same, it is simply predicted by its last outcome without consulting the global history scheme; otherwise, it is predicted by the global history scheme. In effect, this scheme uses n-bit counters to separate and filter out easy-to-predict branches (branches of repeating outcome patterns of length 1) from the rest, so that the interference in the global history scheme can be significantly reduced. Chang *et al.* showed that, usually 3-bit or 4-bit counters can deliver good prediction accuracy.

One requirement for this design is that it needs to identify the ownership of the n-

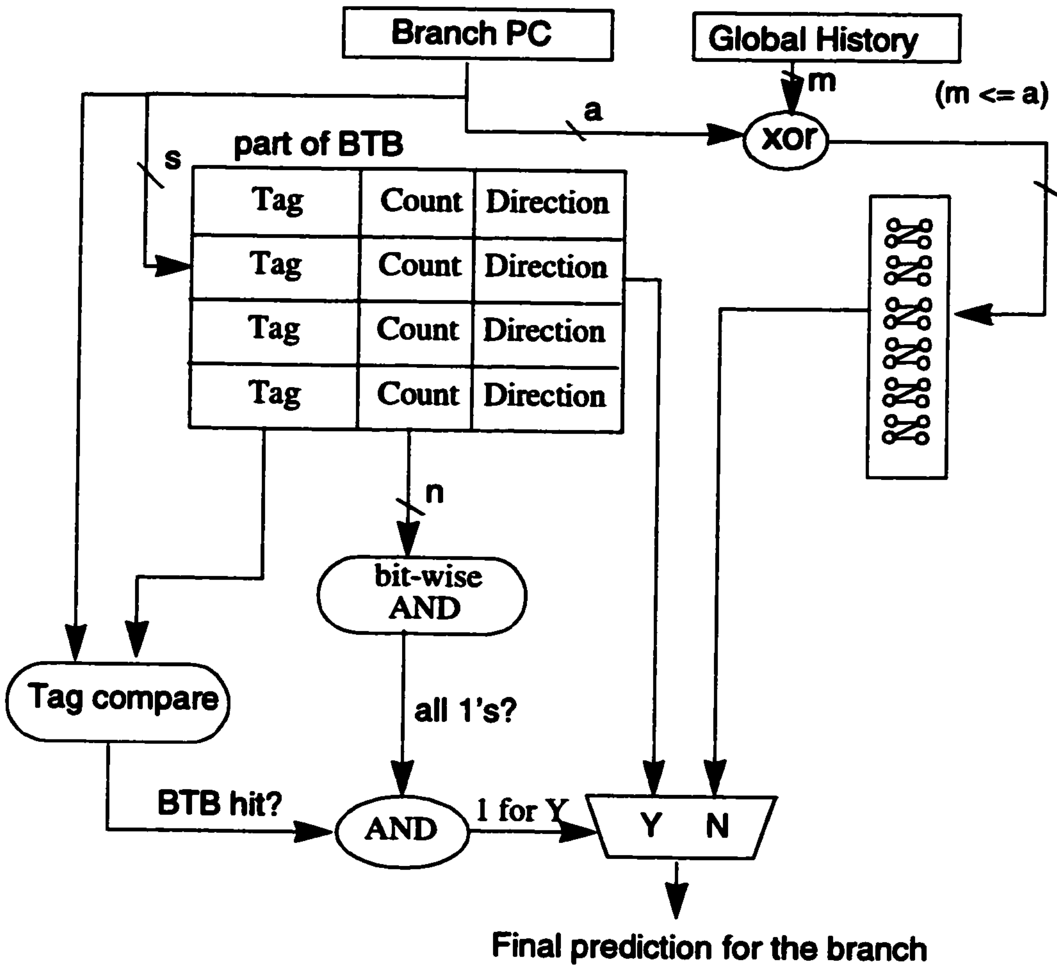
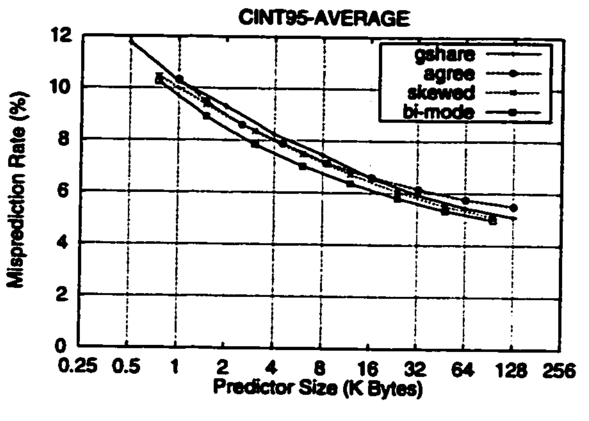


Figure 5.12: Diagram for the filtering scheme

bit counters, because each counter is counting the repeating times of the same outcome for a particular branch, and thus it should not be polluted by other branches. As a result, the scheme requires hardware for storing the branch address, or the scheme has to be implemented in the branch target buffer so it can use the branch address tags for this purpose. This limits the flexibility of the scheme. In the next subsection, we will present the experimental results for the filtering scheme and examine the effect of pollution.



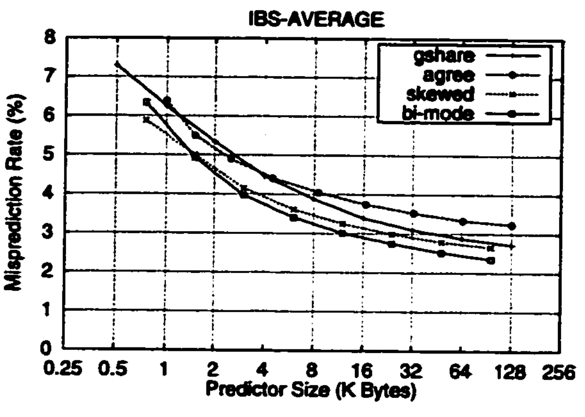


Figure 5.13: Comparison between gshare, agree, skewed, and bi-mode schemes.

In this comparison, the best results for four prediction schemes are shown. The agree predictor has a fixed direct-mapped branch target buffer of 4K entries to record the bias bits; the bias bit is the first outcome of the branch. For the bi-mode scheme, the size of the choice predictor is always half of the second-level table; the total cost is 1.5 times that of the next smaller gshare scheme.

5.2.4 Comparison

In this section, we provide cost-effective performance comparisons for all the schemes discussed in this chapter.

Figure 5.13 compares averaged misprediction rates for SPEC CINT95 and IBS suites among the gshare, agree, skewed, and bi-mode schemes. Figure 5.14 details the performance comparison for individual benchmarks. As can be seen from the figures, the bi-mode scheme performs the best for most of the hardware budgets and benchmarks. However, the second best scheme, the skewed predictor, can outperform the bi-mode scheme when the total budget is small. In such cases, the bi-mode scheme may suffer from relatively high interference in the choice predictor. Finally, for many IBS benchmarks, the skewed predictor performs very close to the bi-mode scheme.

The agree predictor does not perform as well as expected; it even performs worse

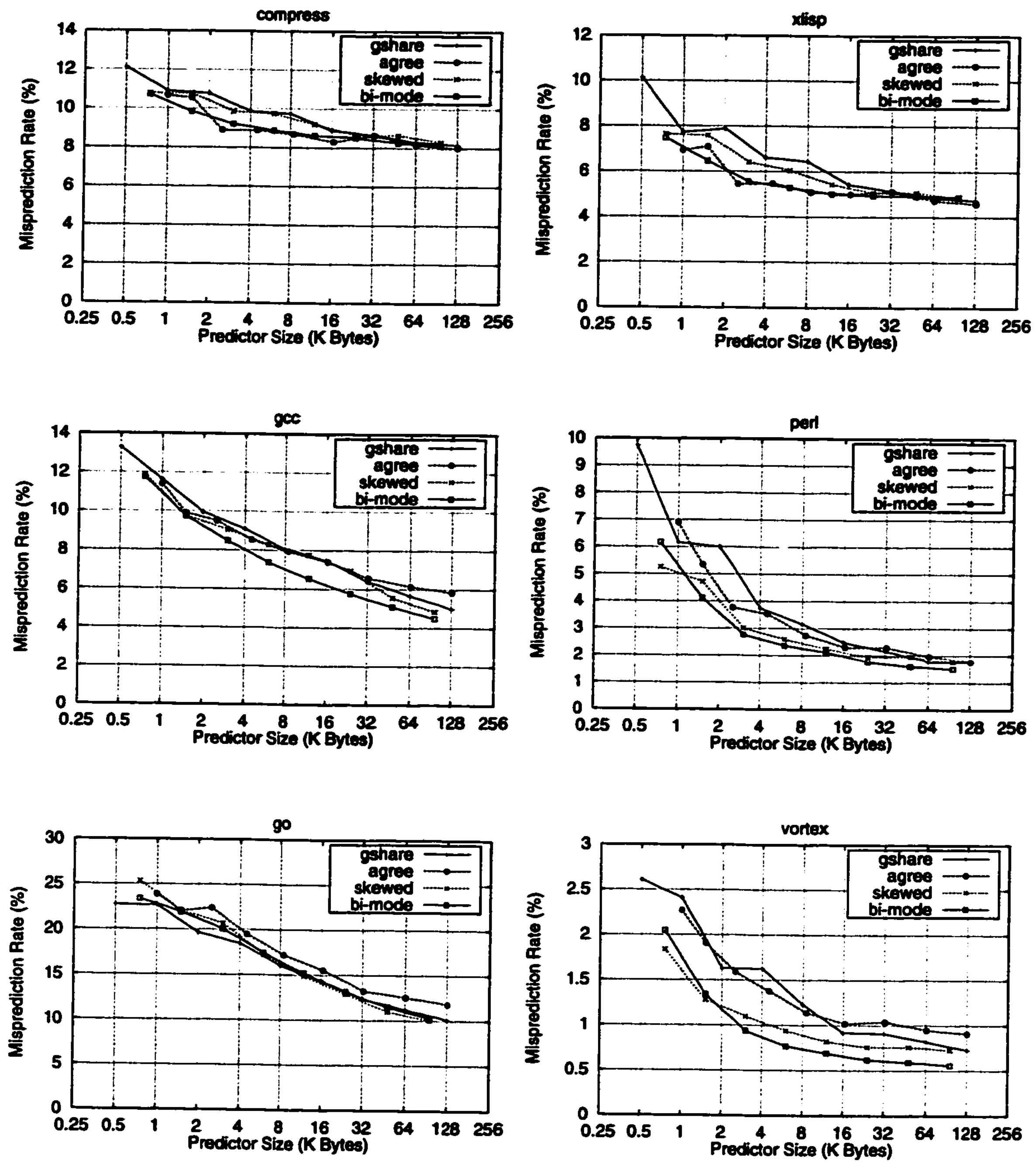


Figure 5.14: Comparison between gshare, agree, skewed, and bi-mode schemes — SPEC CINT95

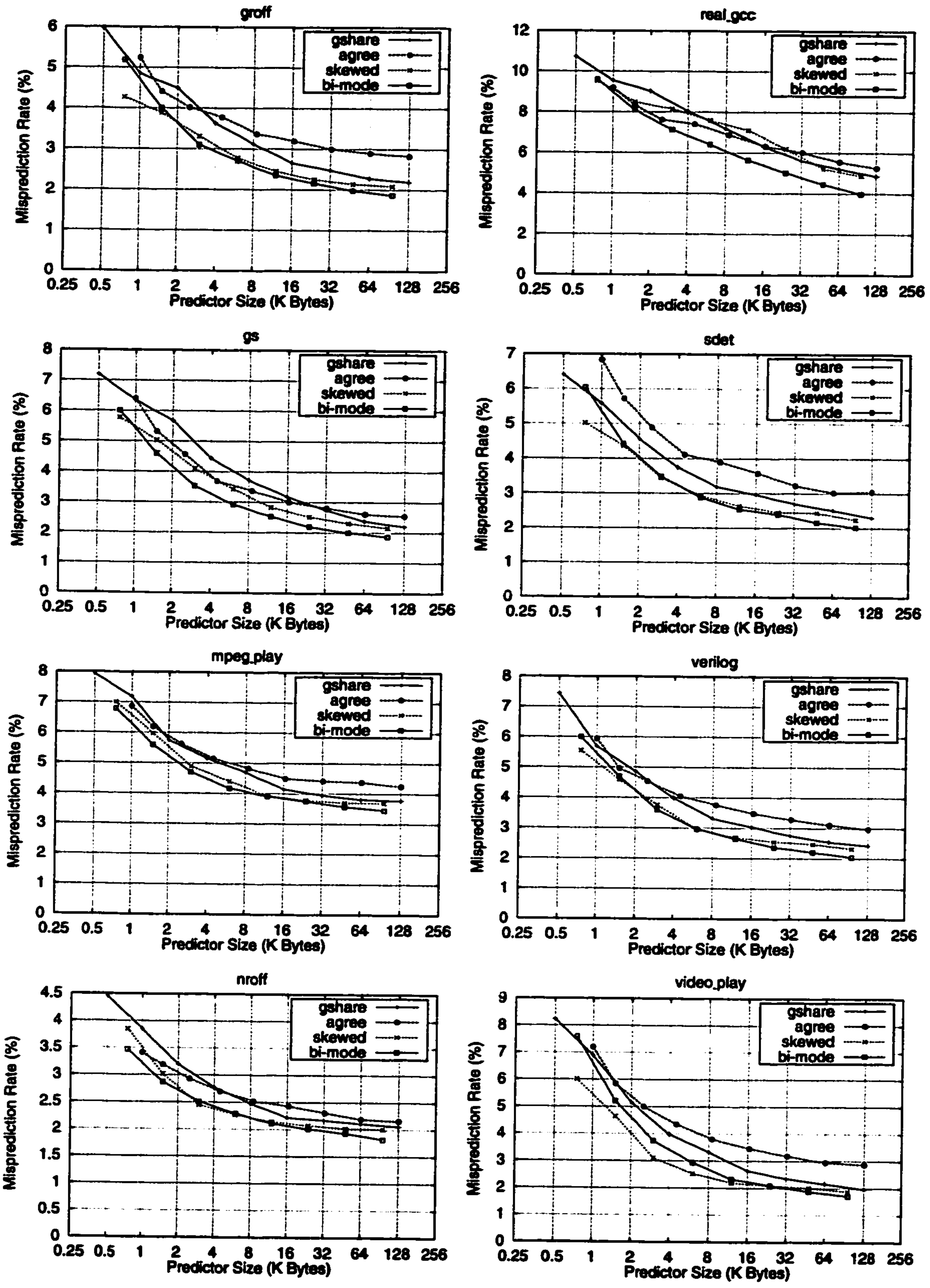
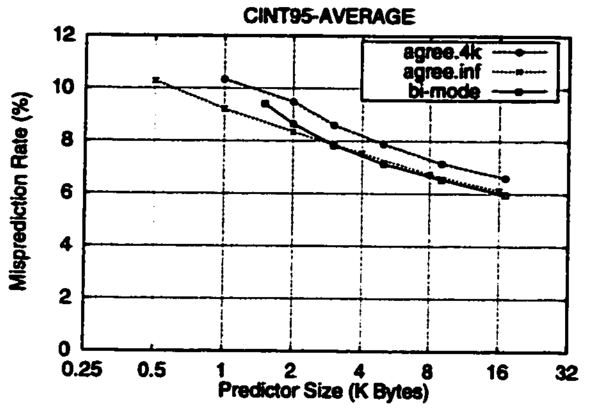


Figure 5.14 (continued): Comparison between gshare, agree, skewed, and bi-mode schemes — IBS-Ultrix



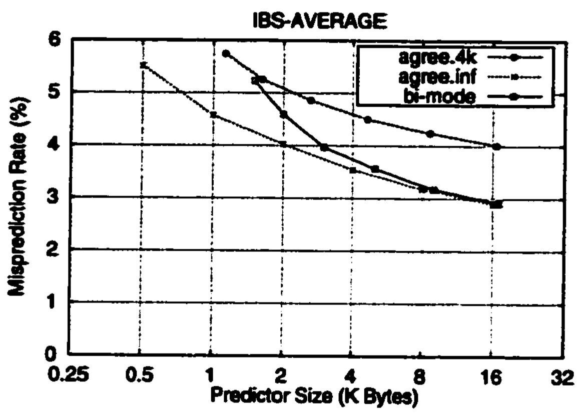
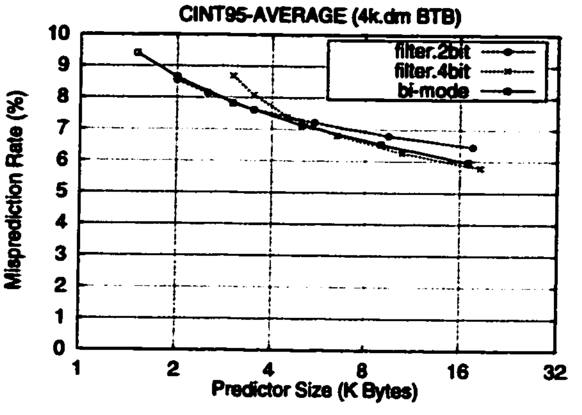


Figure 5.15: Comparison between the agree predictor with a perfect branch target buffer and the bi-mode scheme.

In this comparison, the averaged misprediction rates of three prediction schemes for SPEC CINT95 and IBS are presented. Two agree predictors are examined; one has 4K-entry, direct-mapped branch target buffer (BTB), labeled as *agree.4k*, and the other has unlimited-entry BTB (i.e., this BTB has no conflict misses), labeled as *agree.inf*. Both the agree predictors use the first outcome of each static branch as the bias bit. (As shown in [48], using the first outcome as the bias bit is near optimal.) The bi-mode predictor shown in this comparison has a choice predictor with a fixed size of 4K entries. Both the limited sized agree predictor and the bi-mode predictor are plotted to reflect the total number of bytes used, including costs for the bias bits and the choice predictor, respectively. As for the agree predictor with unlimited-entry BTB, only the second-level table is counted. Therefore, the result for the *agree.inf* is optimistic. As an aside, we note that *agree.inf* can be used to approximate a scheme that uses the compiler to encode the bias bit in branch instructions.
The graph shows *agree.inf* is better than the bi-mode scheme for small budgets. However, we will see that the bi-mode scheme can be improved for the small budgets by employing a partial update policy.

than the best of gshare schemes. The performance of the agree predictor is actually sensitive to the size of the buffer storing the bias bits. Figure 5.15 examines an agree predictor with a perfect branch target buffer. From the figure, we observe a performance gap of about 1% between the agree predictor with a limited BTB and one with an unlimited BTB. However, even the agree predictor with unlimited resources can only perform as well as the bi-mode scheme with a 4K-entry choice predictor.

Figure 5.16 compares averaged misprediction rates for SPEC CINT95 and IBS suites between the filtering and bi-mode schemes. Chang *et al.* have suggested using 3- or 4-bit counters to count pattern repetition. In our comparison, two filtering schemes, one



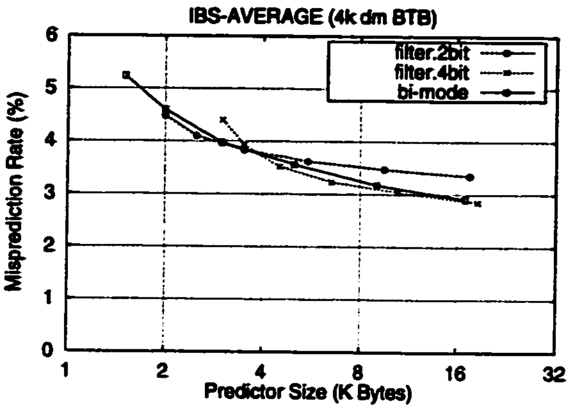
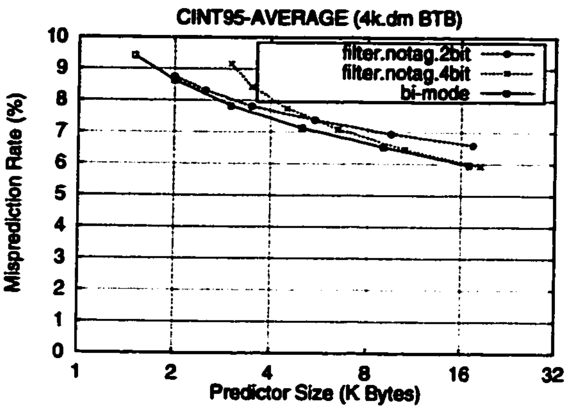


Figure 5.16: Comparison between filtering and bi-mode schemes.

In this comparison, the averaged misprediction rates for SPEC CINT95 and IBS suites are presented. Three schemes are compared: two filtering schemes and a bi-mode scheme. Both filtering schemes have same size (4K-entry) direct-mapped BTBs. One filtering scheme uses 2-bit counters as the filter, labeled as *filter.2bit*, while the other uses 4-bit counters, labeled as *filter.4bit*. If there is a BTB miss, the gshare scheme is used for prediction. The update policy follows Chang's proposal; see [11]. The bi-mode scheme in this comparison has a fixed sized choice predictor of 4K entries. As shown in the graphs, the bi-mode scheme consistently performs as well as the best performance of the filtering schemes, and the bi-mode scheme does not require branch address tags.

with a 2-bit counter and the other with a 4-bit counter are shown to illustrate the performance trend. It can be seen from the figures that the filtering scheme and the bi-mode scheme are close in performance. The filtering scheme proves to be an effective way to filter out the strongly-biased branches for global history predictors. However, to determine the best value of the n for the n-bit counter is difficult because different benchmarks require different values to achieve the best performance. This is not an issue with the bi-mode scheme. Moreover, identifying the ownerships of the filtering counter is crucial to the performance. Figure 5.17 shows that if there are no tags available to identify the ownership of the counters, the performance of the filtering scheme can be degraded.

Michaud *et al.* have also proposed an enhanced skewed predictor, in which the *f1* hashing function is replaced with partial branch address bits [35]. This enhanced scheme



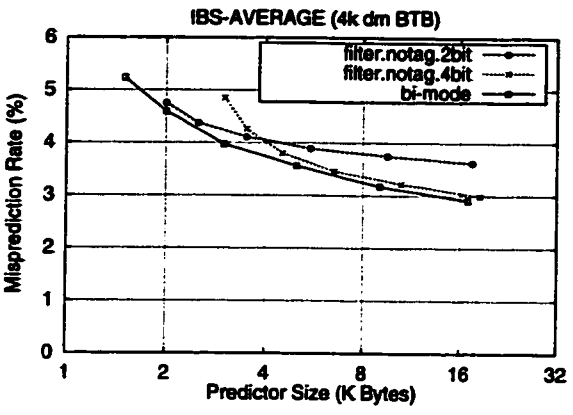


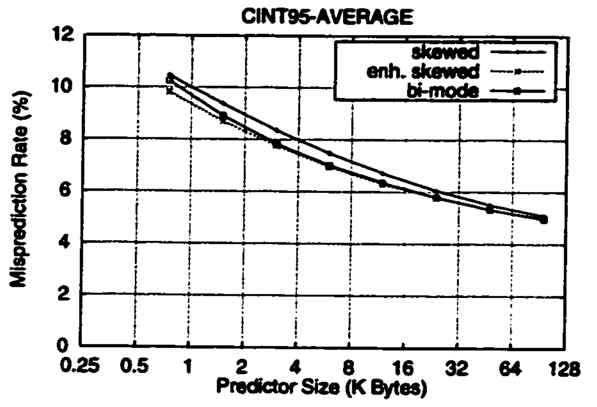
Figure 5.17: The filtering schemes with no address tags

The schemes examined in this figure are identical to those in Figure 5.16 except that filtering schemes have no address tags. Without address tags, the filtering scheme cannot identify the ownerships for its filtering counters. The counters can be polluted and performance is degraded.

can further increase the prediction accuracy by an average of 0.5% for the skewed predictor, see Figure 5.18. For the moment, the enhanced skewed predictor is the most effective scheme for small budgets. Later we will show an improved bi-mode scheme which performs as well as the enhanced skewed predictor for this range of budgets.

5.3 Further experiments

The bi-mode scheme opens a large unexplored area. In this section, several design options for the bi-mode scheme are examined. First, the effectiveness of the choice predictor size is examined. We will show that a 4K-entry choice predictor is a reasonable design point for the bi-mode scheme. Second, we generalized the bi-mode scheme to a multi-mode scheme, which, instead of 2 bias groups, divides branches into four groups: including strongly-taken, strongly-not-taken, weakly-taken, and weakly-not-taken groups. The idea behind this scheme is to further reduce the interference between the weakly and



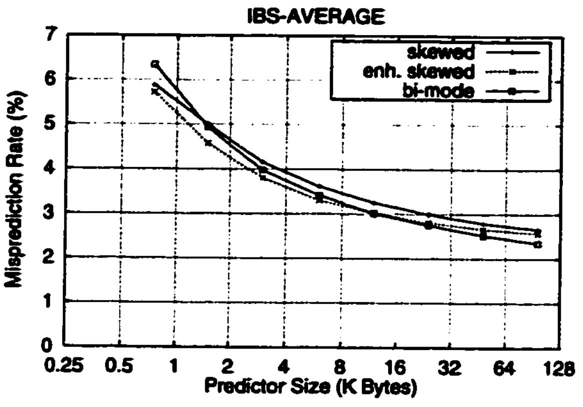


Figure 5.18: Comparison between skewed, enhanced skewed, and bi-mode schemes.

This figure compares the performance of three prediction schemes: skewed, enhanced skewed and bi-mode. The enhanced skewed predictors replaces one of the hash-indexing functions in the skewed with the branch address bits. For the bi-mode scheme, the size of the choice predictor is always half of the second-level table. As shown in the figure, the enhanced skewed predictor can outperform the bi-mode scheme for small budgets. However, the bi-mode scheme can be enhanced for this range of budgets, see Figure 5.26.

strongly biased streams. Another variation we have explored adds a second choice predictor to the bi-mode scheme. This second choice predictor is indexed by part of the global history which is older than the one for the second-level table (the direction predictor). The idea behind this new scheme is to employ more global history bits without incurring much cost. As has been shown before, more global history can potentially improve prediction accuracy as long as the interference is under control.

We also study partial updates for the choice predictor. We will show that partial updates help to improve the bi-mode scheme. Finally, we generalize the bi-mode scheme by using a gshare-like index system for the choice predictor.

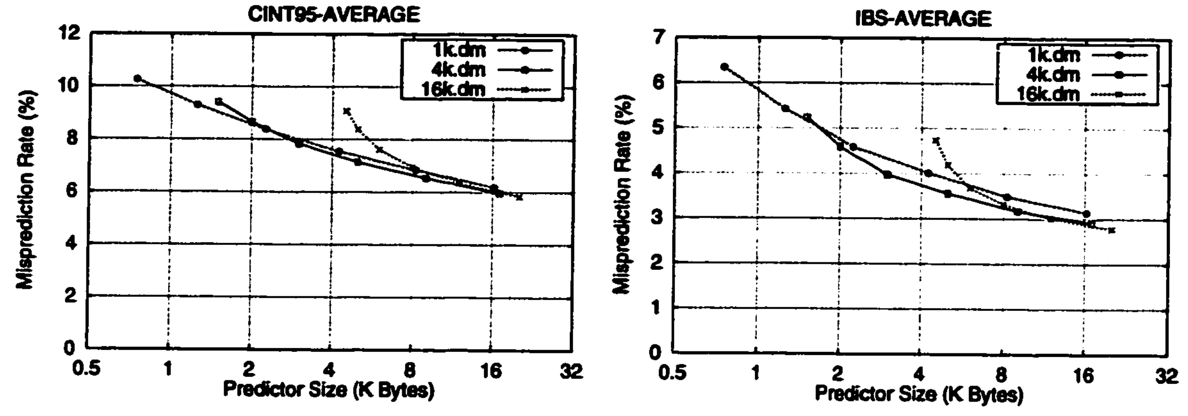


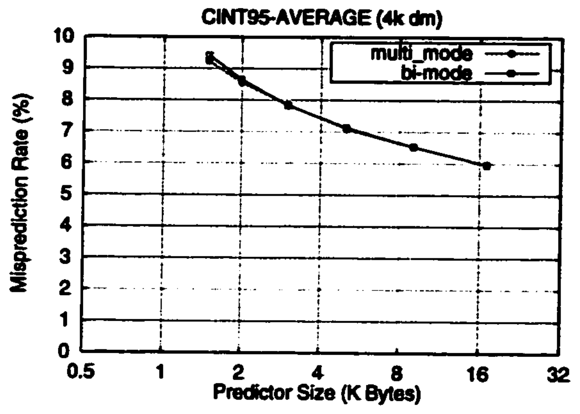
Figure 5.19: Average misprediction rates for bi-mode schemes with different sizes of the choice predictor

In this experiment, three different sizes of the choice predictor are examined for the bi-mode scheme, including 1k-entry, 4K-entry and 16k-entry. The 4K-entry choice predictor delivers the most cost-effective performance.

5.3.1 Effectiveness of the choice predictor size

The discriminating capability of the choice predictor is crucial to the performance of bi-mode schemes. Therefore, the size of the choice predictor cannot be too small, otherwise the interference in the choice predictor becomes noticeable, as was seen in Figure 5.13, where the bi-mode scheme performance is no longer the best when the total budget is small. However, the choice predictor does not need to be very large, as long as it can cover the working set. This is because the choice predictor consists of two-bit counters which can adapt to different branch behaviors quickly (two instances of a branch can bring the counter back on the right track), unlike the filtering schemes where a miss in the BTB will reset the counter, causing it to start counting all over again.

Figure 5.19 shows the results of experiments with three different sizes of the choice predictor. The choice predictor with 4K entries delivers the best performance under the budget constraints shown. To further increase the choice predictor size is not cost-effective. If we refer to Table 3.1 in Chapter 3, where the basic statistics of the



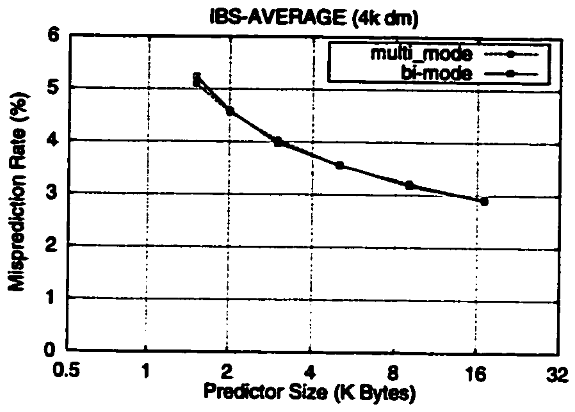


Figure 5.20: Averaged misprediction rates for multi-mode schemes

In these two figures, a multi-mode scheme and a bi-mode scheme are examined. Both schemes have the same size (4K) of the choice predictor. In the multi-mode scheme, the second-level table is divided into four quarters and each quarter is selected by one of the four states of the choice predictor. As shown in the graphs, the multi-mode offer only marginal benefit for small budgets.

benchmarks are listed, we see that the largest benchmark, *real_gcc*, has more than 17,000 static branches, but only about 3,000 of these constitute 90% of the dynamic branches. A 4K choice predictor matches this working set and predictors larger than 4K may only offer marginal benefits.

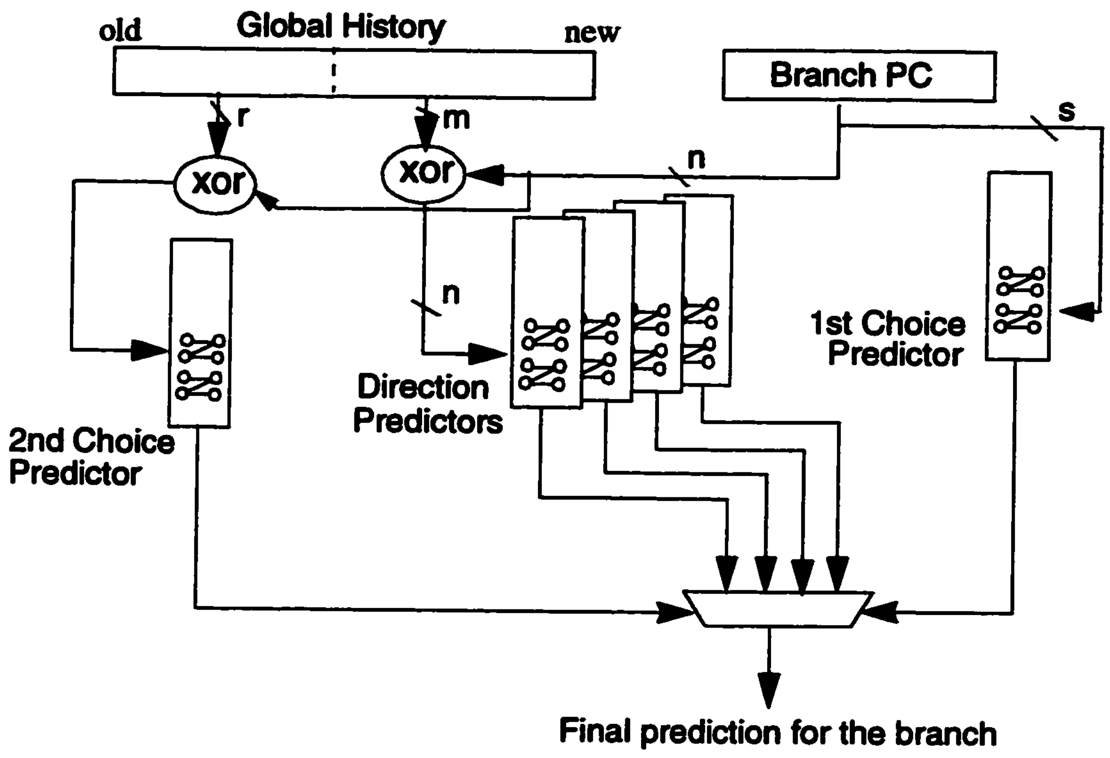
5.3.2 Multi-mode branch predictors

The multi-mode branch predictor we examined divides the second-level table into four equal quarters, and each indexed by the same history bits. To determine which of the four counters, from each quarter of the table, should be used, the multi-mode predictor uses the choice predictor, which is indexed by the branch address. Since there are four possible values for a counter in the choice predictor, each value selects one quarter of the second-level table in this experiment.

Figure 5.20 compares the performance between the multi-mode scheme and the bi-mode scheme. These two schemes perform identically except that when the budget is small the multi-mode scheme provides marginal benefit. When the budget is small, the choice predictor is degraded due to interference. When there is interference, the counters in the choice predictor will bounce back and forth between the taken and not-taken states. By further classifying them into strongly and weakly, taken and not-taken states, the performance loss can be slightly reduced.

5.3.3 Using more history bits

As we have seen, the bi-mode scheme can be improved if the strongly biased branch streams can be separated. However, reducing the number of weakly-biased streams is more difficult, because, as shown in the previous chapter, the number of weakly-biased streams incident on a predictor is primarily determined by the number of history bits used. To reduce the number of weakly biased streams, more history bits must be used. However, this leads to a doubling of the number of counters in the direction predictors with each additional history bit. In this section, we propose a new variant on the bi-mode method that improves prediction accuracies by using more history bits, but that does not lead to an overwhelming growth in the number of direction counters. The new scheme, called history+, adds another choice predictor, which is similar to the original choice predictor but it is indexed by older global history bits, as shown in Figure 5.21. For example, if the second-level table, the direction predictor, is selected by the most recent m global history bits, then the newly added choice predictor use the r global history bits that are the next older group. When a branch outcome is resolved, the second choice predictor will be updated with the branch outcome. The history+ scheme requires $4\text{x}2^{m}$ direction counters rather than $2\text{x}2^{(m+r)}$, as would be the case if we simply extended the number of history bits



n = number of address bits used to index the direction predictors

m = number of history bits used to index the direction predictors

s = number of address bits used to index the 1st choice predictor

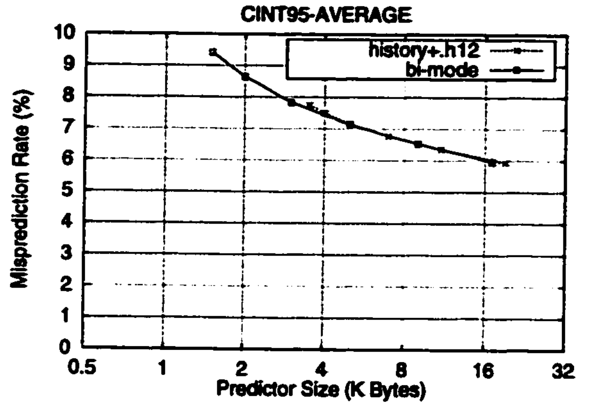
r = number of old history bits used to index the 2nd choice predictor

(m <= n) (s <= n)

= 2-bit saturating up-down counter

Figure 5.21: Diagram for the history+ scheme

In this new bi-mode scheme, a second choice predictor is added to exploit more global history information. As shown in the figure, the second choice predictor is indexed by *r* global history bits which are older than the *m* bits used for the direction predictors. In this figure, the second choice predictor is implemented as a gshare. It can also be implemented as a bi-mode scheme if less interference is desired.



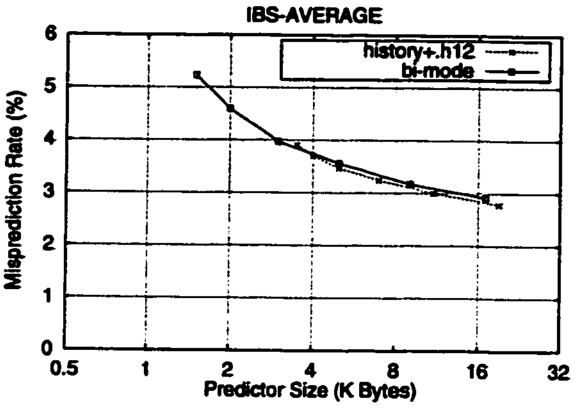


Figure 5.22: Average misprediction rates for the history+ scheme

In this comparison, a bi-mode scheme with a second choice predictor indexed by older global history bits, called history+, is compared with the original bi-mode scheme. The second choice predictor is indexed by 12 global history bits which are immediately older than the global history bits used for the direction predictors, i.e., r equals 12 in the model shown in Figure 5.21. As shown in the graphs, the history+ scheme can only offer marginal benefit on average. However, if individual benchmarks are examined, as we will show next, some benchmarks are improved significantly.

to $m+r$. This scheme can improve performance because the r extra history pattern can help distinguishing some situations that the m-bit history pattern cannot.

Figure 5.22 presents averaged misprediction rates for the SPEC CINT95 and IBS suites using the history+ scheme. It seems that, on average, the history+ method offers little advantage. However, if each individual benchmark is examined, as shown in Figure 5.23, the history+ method can improve performance significantly for some of benchmarks, such as *perl*, *vortex*, *nroff*, and *verilog*. The proposed history+ method reduces the variance of misprediction. This looks promising, but clearly further study is needed before this can be exploited fully. We leave it for future work.

5.3.4 Partial update

In the original bi-mode scheme, the choice predictor is always updated with branch outcomes. We refer to this policy as full update. This full update policy may

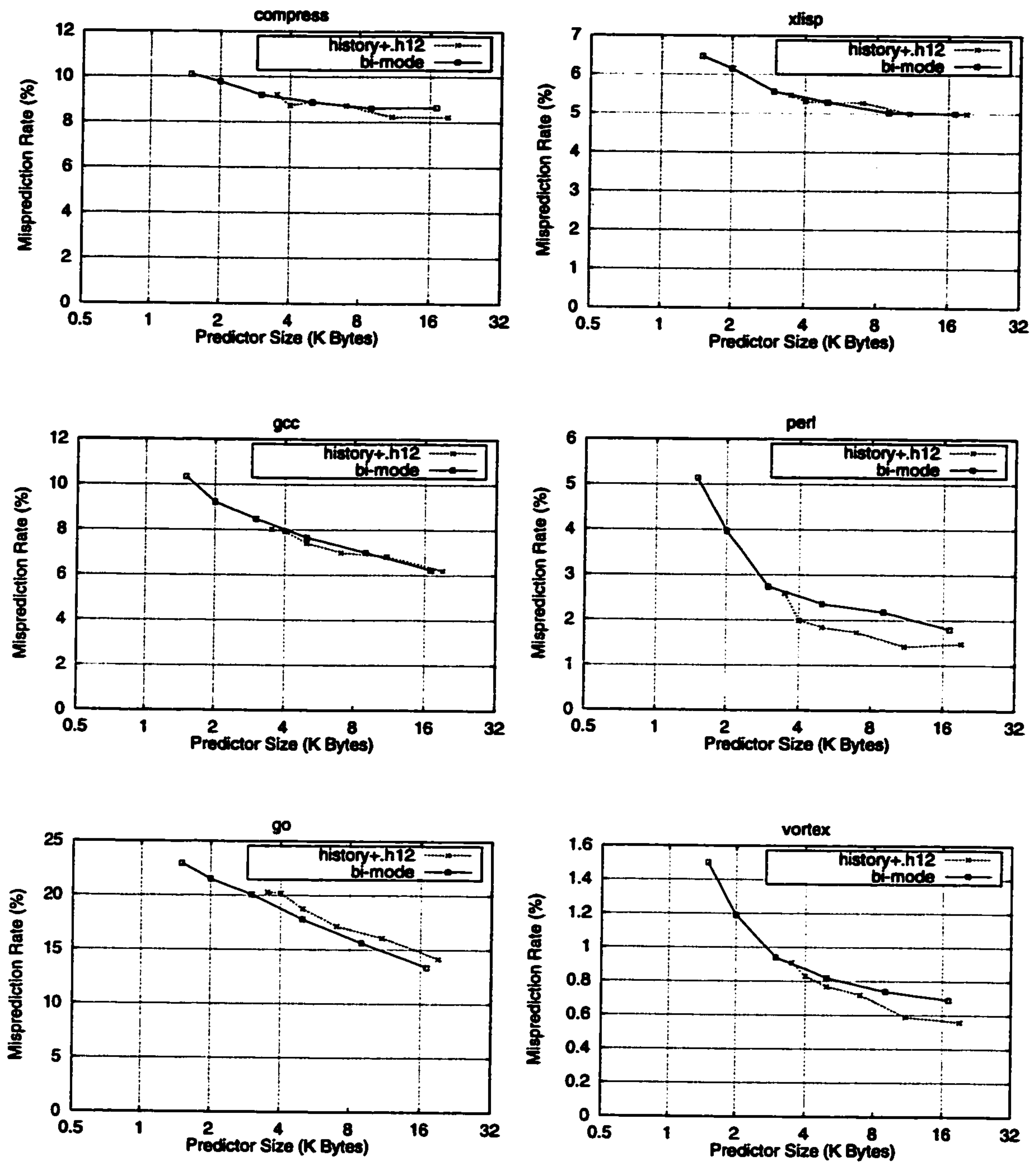


Figure 5.23: Misprediction rates for the history+ scheme — SPEC CINT95

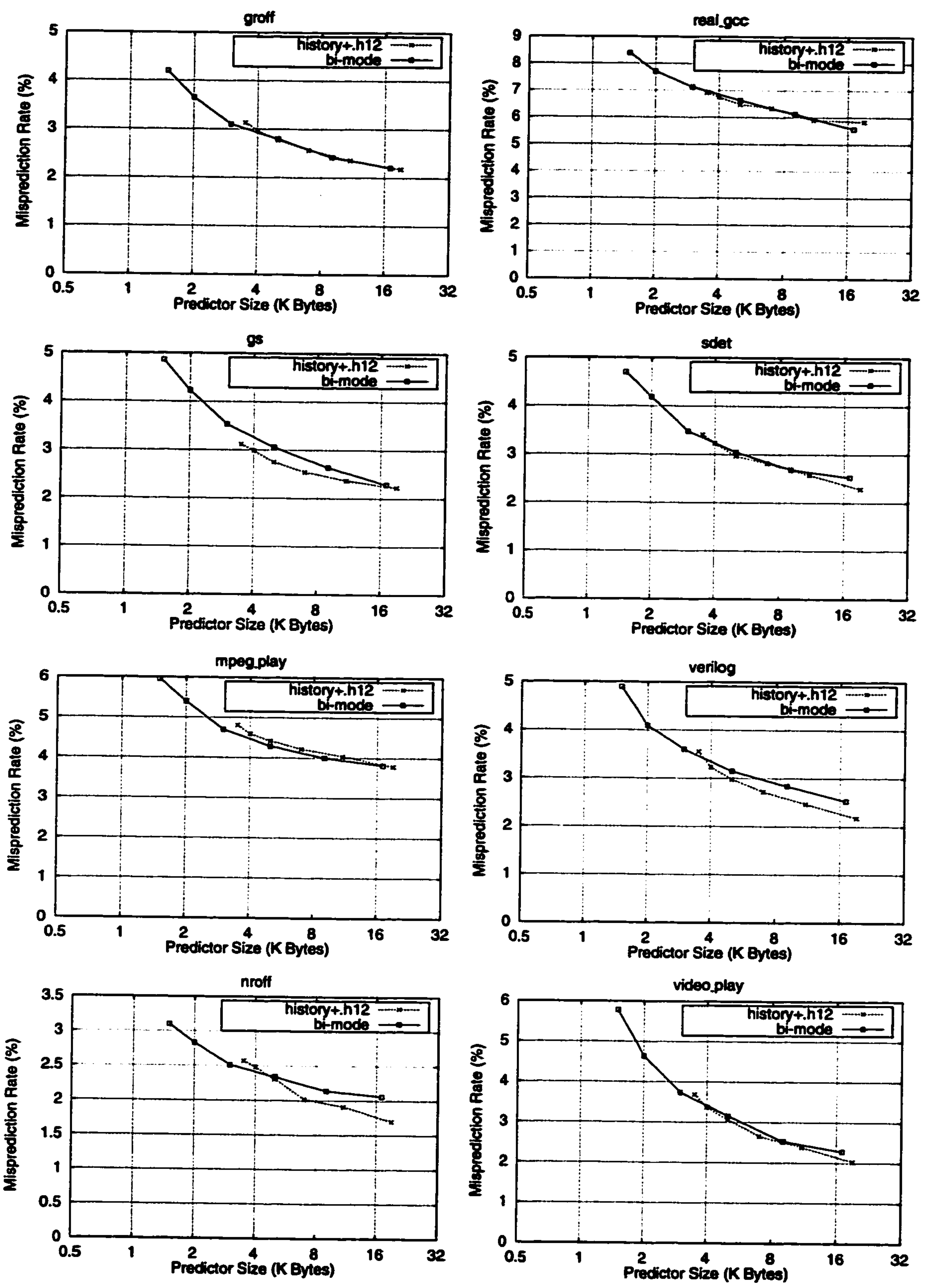
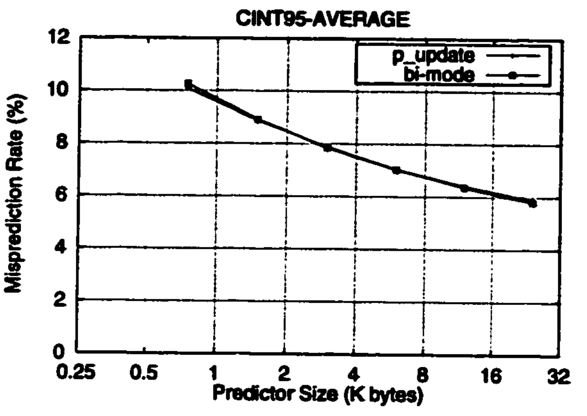


Figure 5.23 (continued): Misprediction rates for the history+ scheme — IBS-Ultrix

potentially increase the number of counters from the direction predictor used by each static branch. Normally, the choice predictor sticks to the biased direction of a static branch and chooses counters from the corresponding bank of the direction predictor to make the final predictions. However, as a result of interference caused by other branches or exceptional cases of the static branch (e.g., exits from a looping branch), the choice predictor may change its choice for the next instance of the static branch. In other words, some counters in the other bank may be chosen and altered. Ideally, these counters should not be changed because it has the effect of sharing more counters among the static branches. In small predictors where sharing counters is heavier, restricting the counter usage by each static branch may help improve performance.

In this subsection we examine an alternative update policy, which is referred to as partial update. The choice predictor will be updated except when the final prediction is correct but the choice is different from the final prediction. The idea of this partial update is to keep the choice predictor stuck on the normal bias of the static branches and be less influenced by noise or exceptional cases. In this way, the adaptivity of the choice predictor is sacrificed but each static branch may use fewer counters than it does in the original scheme. The restriction of counter usage may provide a net performance gain because interference caused by sharing is reduced. Figure 5.24 presents the experimental results. It can be seen that the partial update slightly improves performance for the bi-mode scheme when the predictor sizes are small. This is more pronounced for larger workloads, such as IBS benchmarks, which have a greater degree of interference. However, this update policy has less impact for large predictors where sharing is not significant.



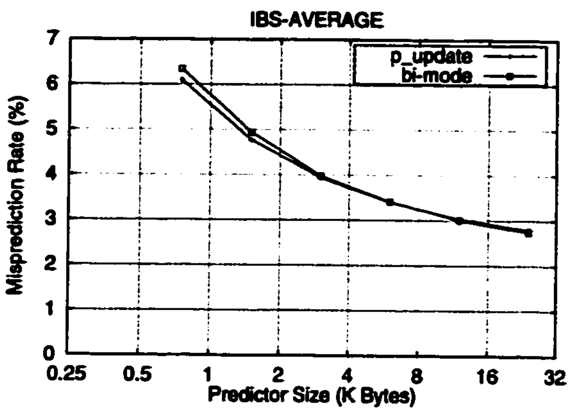


Figure 5.24: Comparison between full update and partial update for the choice predictor

This figure compares a bi-mode scheme with partial update (p_update) and the original bi-mode scheme. The partial update is to not update the choice predictor if the choice is opposite to the outcome but the selected direction predictor can make good prediction.

5.3.5 gshare-like indexed choice predictors

The choice predictor need not be indexed by the branch address. As long as an index can separate out oppositely biased branch outcome streams for the second-level table, i.e., the direction predictors, it is a good index to use for the choice predictor. We can expect that, to separate oppositely biased branch streams for the direction predictors, the index employed in the choice predictor should be "orthogonal" to the index for the direction predictors. For example, if the direction predictors make heavy use of history bits as an index, the choice predictor should use few history bits, and vice versa. The idea is to avoid interference simultaneously in both choice and direction predictors for a branch.

In this subsection, we examine a bi-mode scheme whose choice predictor is indexed by the xor of equal numbers of branch address bits and global history bits (10 to 15 history bits in our experiments, depending on the predictor sizes). To be "orthogonal,"

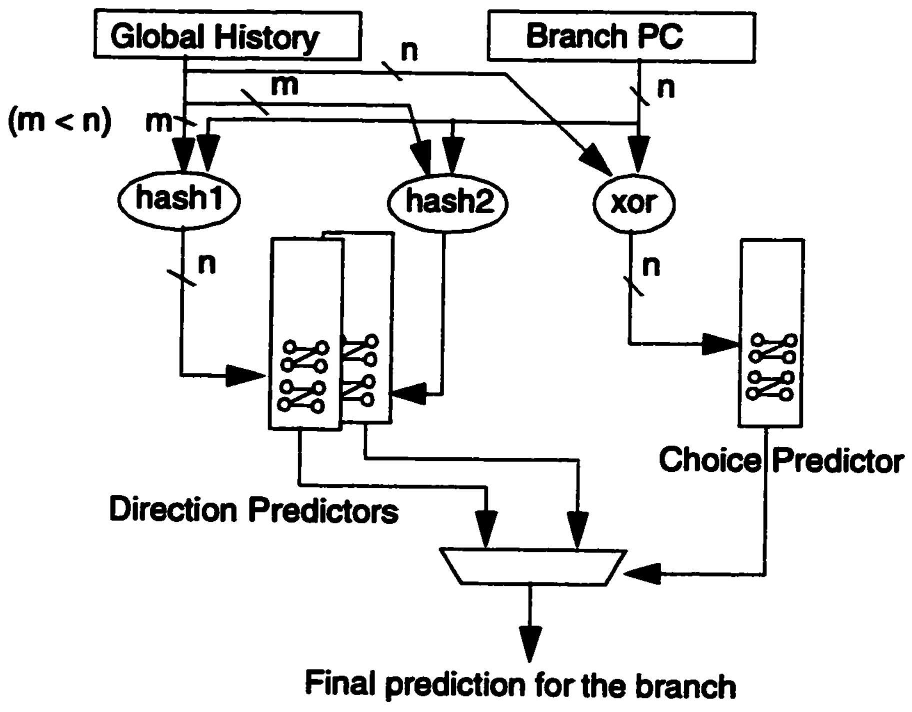
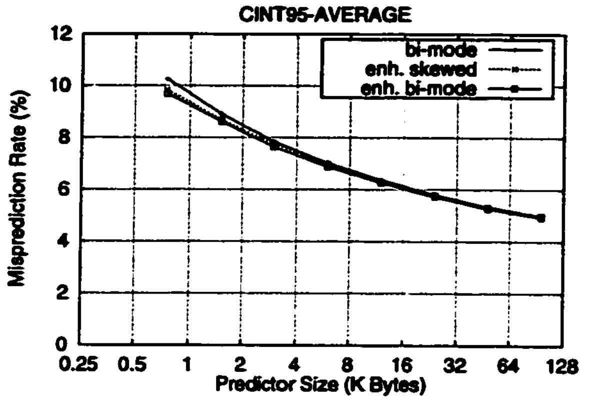


Figure 5.25: Diagram for the enhanced bi-mode predictor

In this enhanced bi-mode predictor, both the choice and direction predictors use an xor of history and address bits to form the indices. To achieve good performance, the direction predictors should use fewer history bits than the choice predictor. Two options can also be employed to optimize performance: the choice predictor can employ a partial update policy and the two direction predictors can use different indices (for example, using different hash functions, as shown in the diagram).

predictors but with the addition of a few global history bits (1 to 5 history bits in our experiments). To optimize the predictor, the partial update policy is also employed. Figure 5.25 illustrates the enhanced bi-mode predictor.

Figure 5.26 shows the averaged misprediction rates of the enhanced bi-mode, the normal bi-mode, and the enhanced skew schemes for SPEC CINT95 and IBS-Ultrix. Figure 5.27 provides a detailed examination for individual benchmarks. As can been seen from the figures, the enhanced bi-mode scheme performs very closely to the enhanced skewed predictors; both are effective schemes for smaller budgets. However, each scheme has its own advantages and disadvantages. The skewed predictor treats all three banks



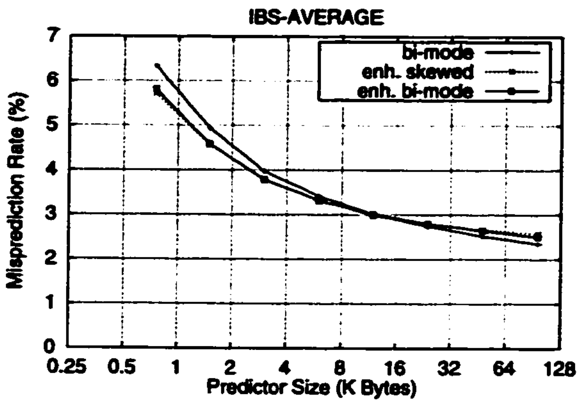


Figure 5.26: Comparison between bi-mode, enhanced skewed, and enhanced bi-mode schemes

In this figure, the enhanced bi-mode scheme is compared with the original bi-mode and the enhanced skewed schemes. It can be seen that the enhanced bi-mode scheme performs on average almost identically to the enhanced skewed predictor.

equally; when the address-indexed bank does not do a good job, the predictor will rely on the other two banks to make the final prediction. In contrast, the bi-mode scheme always relies on its choice predictor to select a bank for the final prediction. The choice predictor may not be able to make good selections all the time, thus limiting the overall prediction accuracy.

The bi-mode scheme, on the other hand, has shorter training periods for new branches, compared to the skewed predictor. In the bi-mode scheme, one bank of the second-level table is mostly biased in the not-taken direction, while the other bank is biased in the take direction. Since most branches in realistic programs are strongly biased, the bank selected by the choice predictor has typically been trained by some other similarly biased branches; it is enough for just the choice predictor to capture the bias of a new branch for the bi-mode predictor to start to make accurate predictions. In contrast, the skewed predictor requires at least two of the banks to be trained for new branches. This can be a problem for programs that have numbers of branches that have many global

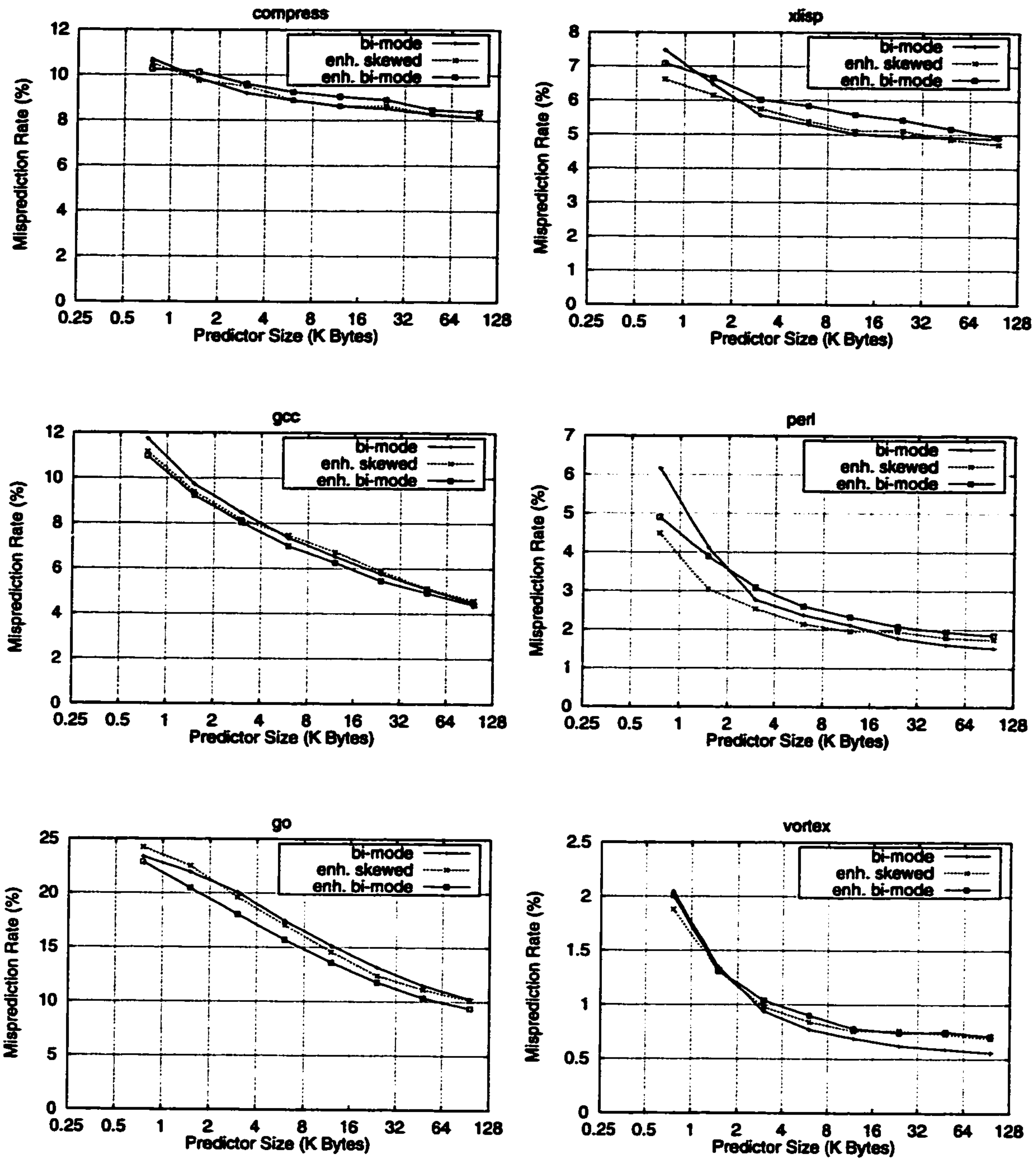
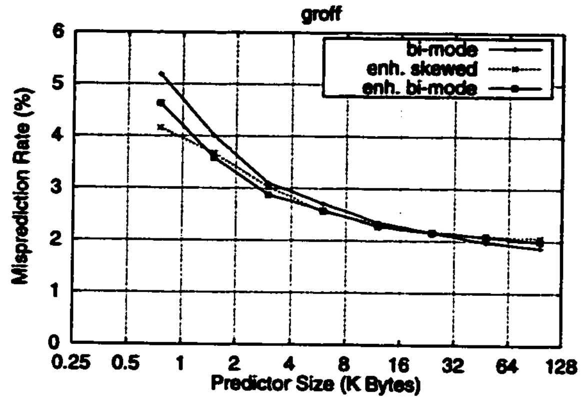
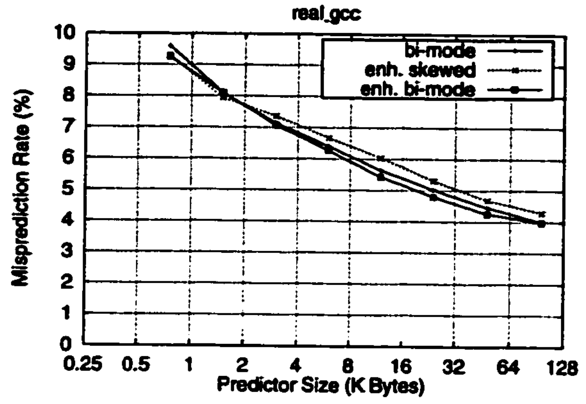
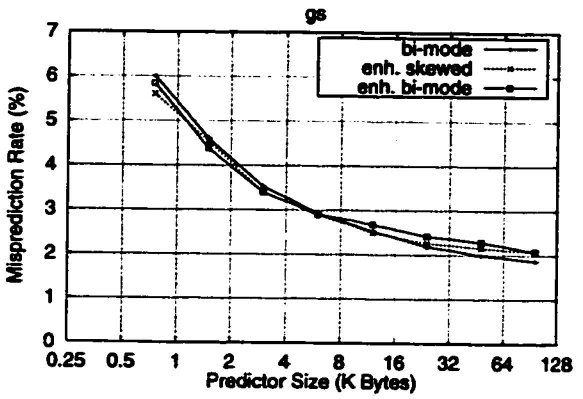
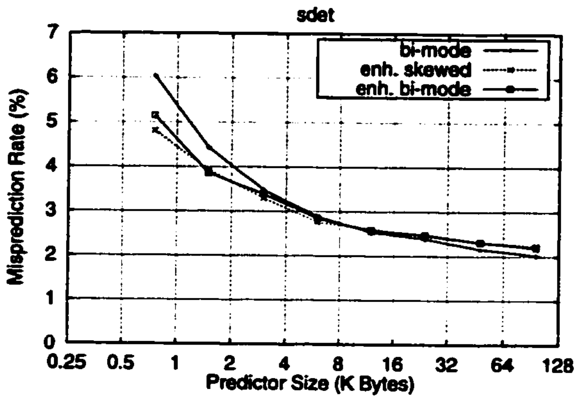


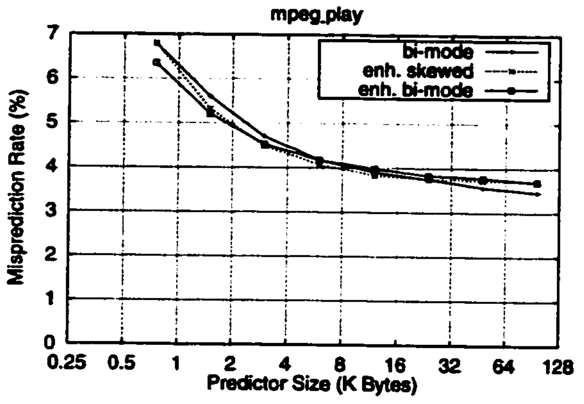
Figure 5.27: Comparison between filtering, enhanced skewed, and enhanced bi-mode schemes — SPEC CINT95

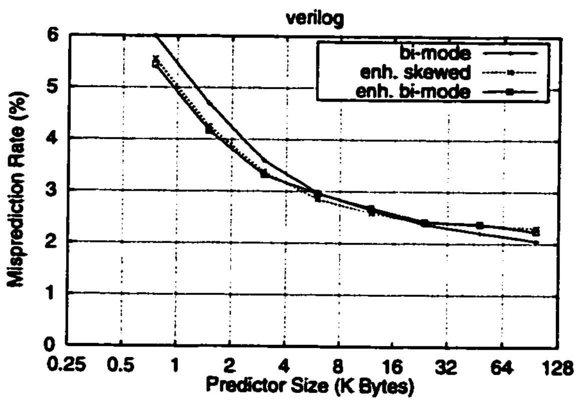


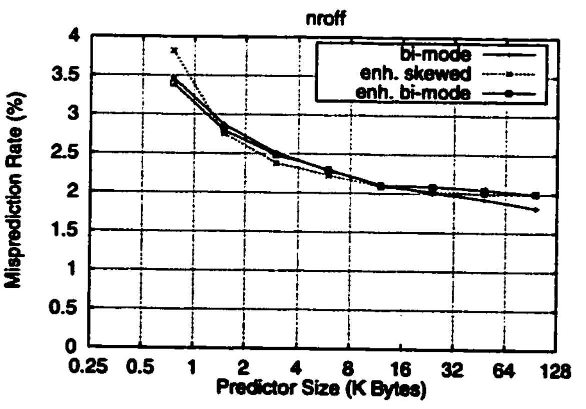












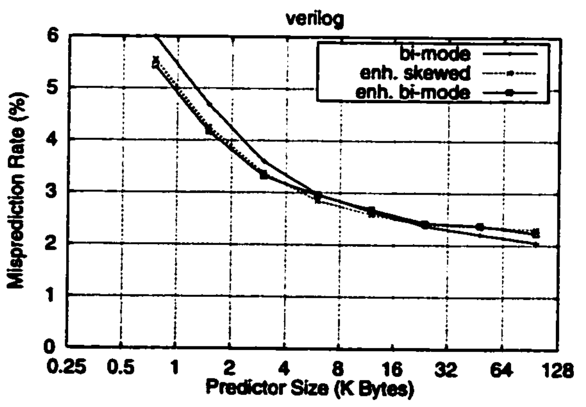


Figure 5.27 (continued): Comparison between filtering, enhanced skewed, and enhanced bi-mode schemes — IBS-Ultrix.

history patterns, such as *real_gcc*, *gcc* and the *go* benchmarks. Chang has also observed that *gcc* and *go* benchmarks have a large percentage of strongly biased branches that are better predicted with simple two-bit counters in the filtering scheme [13]. In Figure 5.27 we can see that for these three benchmarks, the enhanced bi-mode predictor is better than the enhanced skewed predictor. For large sizes, the original bi-mode predictor is still the best predictor.

5.4 Summary

In this chapter, a new global-history based branch prediction scheme, the bi-mode predictor, was proposed. It sought to improve prediction by eliminating the destructive interference in dynamic branch predictors. Its success relies on dynamically determining the taken or not-taken direction with an accurate but simple choice predictor. This classification can help remove much of the destructive aliasing incident on the two-bit counter tables while keeping the harmless aliasing together.

A detailed analysis of the bi-mode scheme's method for indexing into the two-bit counter tables was also presented, using the analysis technique developed in Chapter 4. It was shown that the bi-mode scheme can preserve the benefits gained from using global history bits as the index into the second-level table, and, by adding a moderate sized choice predictor to dynamically discriminate branches, the destructive interference caused by history bits can be significantly reduced. The benefits of using branch addresses and global history cannot be preserved in current two-level schemes simultaneously, but they can in the bi-mode scheme.

Other recently proposed techniques to reduce interference are also examined, including the agree predictor, skewed predictor, and filtering schemes. A cost-effective comparison among these schemes and the bi-mode scheme is performed, in which cost is measured by the amount of storage in bytes that each requires. The result shows that the bi-mode scheme is better overall because of its simple hardware design and adaptivity.

Five variations of the bi-mode scheme were also examined. First, several different sizes for the choice predictor are compared, and it is shown that a choice predictor with 4K entries is the best configuration for the benchmarks examined. Second, a multi-mode scheme is evaluated, and the results show that the multi-mode offers no improvement. Third, a variant of the bi-mode predictor, the history+ scheme, that uses extra older global history bits was also proposed. On average, it does not deliver significantly better results than the original bi-mode scheme, but it does improve performance significantly for some of the benchmarks, suggesting a possible direction to further increase prediction accuracy. Fourth, partial update for the choice predictor was studied and it is shown to improve performance for the small sized bi-mode scheme. Lastly, an enhanced bi-mode scheme is proposed and evaluated. This enhanced bi-mode scheme employs a gshare-like index for the choice predictor and proves to be a very accurate dynamic branch predictor when the budget is small.

CHAPTER 6
Per-address History Schemes

In previous chapters, the dynamic branch predictors that use global history information have been investigated. In this chapter, branch predictors that use per-address history are examined. The pros and cons of per-address schemes are studied. The benefit of exploiting auto-correlation of a branch with its past outcomes, and the performance bottlenecks of the per-address scheme are identified. We will see that per-address schemes preform very well for highly biased branches, but poorly for weakly biased branches. Furthermore, interference in the second-level table of per-address schemes is found not to be a significant issue, unlike the first-level table where it can be detrimental. We conclude that the design of the per-address history scheme should focus on reducing the interference for the first-level table.

Because interference in the second-level table is not a concern, sharing counters is possible. We will show that the adaptive counter itself is not critical to the performance and can be replaced with a static 1-bit scheme that uses algorithm-derived (not by profiling), static-trained values without compromising the performance too much. This new static scheme employes a set of fixed prediction values for all programs, so its prediction values can be implemented with simple fixed logic rather than memory cells or n-bit counters. This simplified design provides advantages of smaller on-chip area and shorter access time to the second-level table (accessing simple logic is usually faster than accessing memory cells or n-bit counters). We will show that the static scheme compares quite well with the adaptive scheme.

Finally, a cost-effective comparison including the cost for the first-level table is examined for the per-address scheme. In this comparison, we assume the first-level table is direct-mapped instead of associative as was done in [59], because we think the access time to the branch predictor becomes a serious concern in most of current high-performance microprocessors. This comparison will identify the optimal configurations for the per-address history schemes, and also compares the optimal configurations against the global history schemes. We will show that the per-address scheme with a direct-mapped first-level table performs worse than the global history scheme for the integer benchmarks examined. This is because the direct-mapped first-level table of the per-address scheme requires larger area to mitigate its interference problem. We begin by reviewing the PAs model.

6.1 The per-address history scheme model

The per-address history scheme differs from the global history scheme in having more than one history register in the first-level table. With multiple history registers, it is possible to record outcome history for each static branch, and these histories can then be used as the index into the second-level table. In such schemes, the per-address history of a static branch is the key ingredient, and the correlation with its neighboring branches is ignored. Whether the per-address history is more effective than global history depends on the type of workload, and we will investigate this point for our workloads later in this chapter.

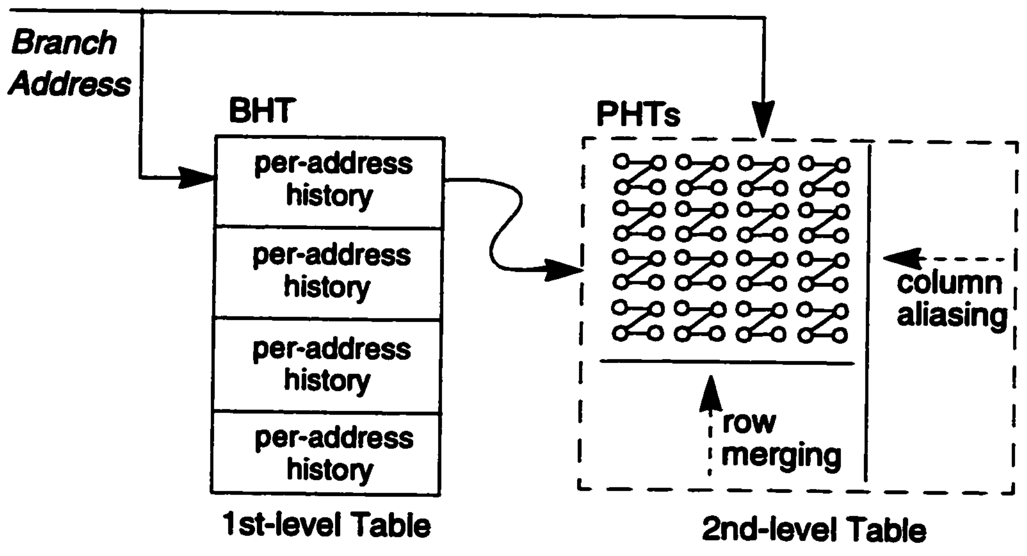


Figure 6.1: The model for the per-address history two-level dynamic branch predictor

A model for the per-address history dynamic branch predictor is depicted in Figure 6.1. The second-level table still consists of two-bit saturating counters, but the first-level table contains more than one branch history register, in contrast to the model for the global history scheme shown in Figure 4.1. As we noted earlier, this first-level table is referred to as the branch history table (BHT). The idea is that the BHT has one register for each static branch so that true per-address branch outcome history can be recorded. However, with typically thousands of static branches in a program, it is impossible to implement a per-address scheme without having more than one branch sharing a history register. Interference then arises because of this restricted first-level table, and this interference can degrade the performance.

The second-level table, similar to the GAs scheme, can contain multiple columns of counters, or pattern history tables (PHTs). A per-address history scheme with a multiple-column second-level table is generally referred to as a PAs scheme. If the second-level table contains only single column, it becomes a PAg scheme. Branches that have same per-address history patterns may be aliased to counters, causing interference in the second-level table. However, we will show that the interference in the second-level table is mostly not destructive and does not degrade the performance significantly.

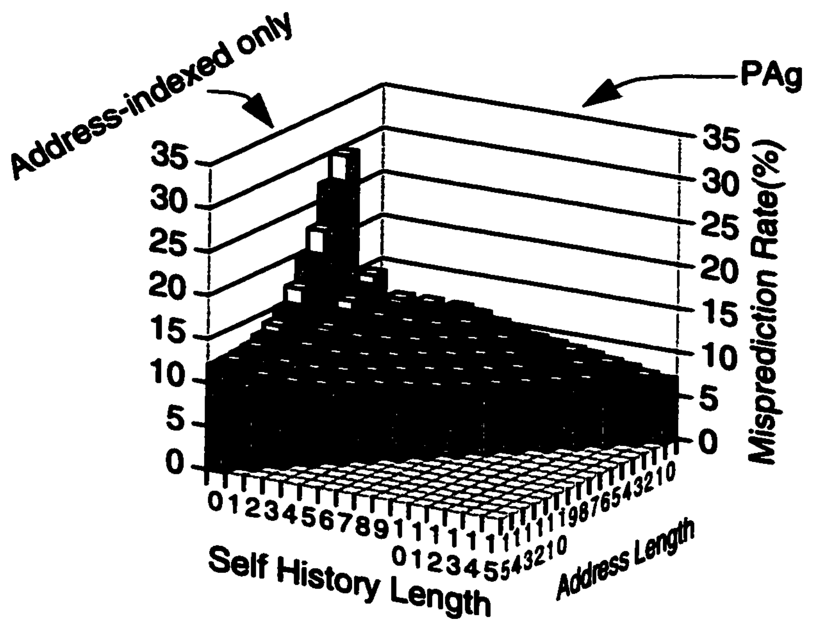
The following section will provide our detailed performance evaluation for the various per-address history schemes.

6.2 Performance issues

6.2.1 Interference in the second-level table

In order to not be affected by the interference problem in the first-level table, we start by assuming the size of the first-level table is unlimited; the BHT can assign one history register for every static branch without causing aliasing in the table. Figure 6.2 shows the averaged misprediction rates of such a PAs scheme, labeled PAs(inf), for the SPEC CINT95 and IBS-Ultrix suites. Figure 6.3 is for the *sql95* benchmark. Again, the optimal configurations are marked in black for each size of the second-level table.

The first noticeable thing is that the surface of PAs(inf) is radically different from that of the global history schemes shown in previous chapters. The surface of PAs(inf) is more like a plateau, suggesting that the performance of PAs(inf) is much less sensitive to the size and configuration of the second-level table than the global history schemes. In fact, the same figure shows that if the second-level table is expanded a thousand times, the misprediction rate of the optimal configurations decreases by less than 5%. Therefore, allocating expensive chip area to the second-level table may not be cost-effective.



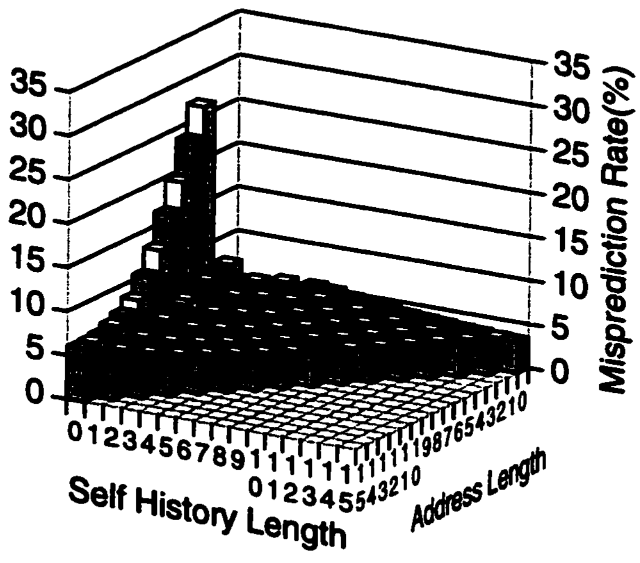


Figure 6.2: Averaged misprediction rates of PAs with an unlimited-size branch history table — SPEC CINT95 (top) and IBS-Ultrix (bottom)

In this figure, the averaged misprediction rates of the PAs scheme for the SPEC CINT95 and IBS-Ultrix benchmarks are presented. The second-level table of the PAs scheme ranges from 2^4 counters (the tier in back) to 2^{15} counters (the tier in front). All possible pair-wised combinations of columns and rows for each size of the table are included. Each white and grey tier represents a fixed-size table. The optimal configuration within each tier is marked in black. We can see that PAg, the configuration that does not use address bits, performs best in most cases. Furthermore, increasing the second-level table size 1,000 times only provides a small benefit (less than 5%), suggesting that PAs has no severe interference problem in its second-level table.

Second, from this figure we can see that the single-column configuration of the second-level table tends to perform the best—the PAg scheme. (Bear in mind that this comparison only considers combinations of columns and rows for a fixed size second-level table, but ignores the first-level table cost.) It suggests that per-address history bits are more effective than the address bits for achieving good prediction accuracy, assuming the history bits are not much more expensive than the address bits. Later in this chapter we will discuss the issue of cost-effectiveness about using per-address history bits.

In the PAg scheme, branch addresses are used to index into the first-level table, or the BHT, to record histories for each static branch. Thus, the performance of such a predictor is determined by whether different branches having the same history patterns behave alike. If they do not behave alike, destructive interference will arise. Since the single column table (no address information used in the second-level) is not able to discriminate between branches that have the same per-address history patterns, the good prediction accuracy of the PAg scheme suggests that branches with same history patterns will most likely have similar outcomes. Therefore, we conclude that interference in the second-level table is minor compared to the global history schemes of Chapter 4.

However, when the second-level table becomes very large in Figure 6.2 and Figure 6.3, the optimal configurations (see black bars in the figures) move to the middle of the tiers, suggesting that the multiple-column per-address schemes are the best. This occurs because some of the benchmarks do not generate very long histories, and therefore the benefit from an additional history bit becomes less than that from an additional address bit. However, the difference between the optimal and the PAg configurations is negligible—less than 0.2% in prediction accuracy. The benefit from discriminating

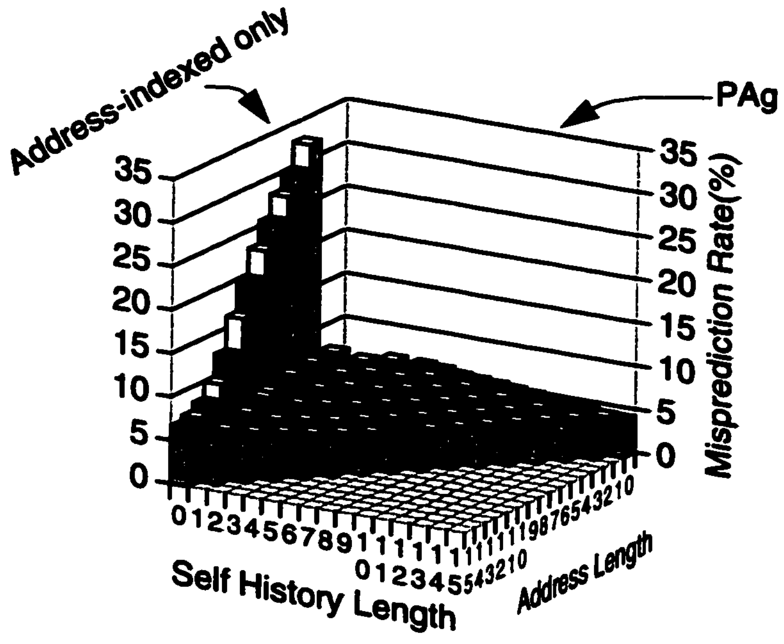


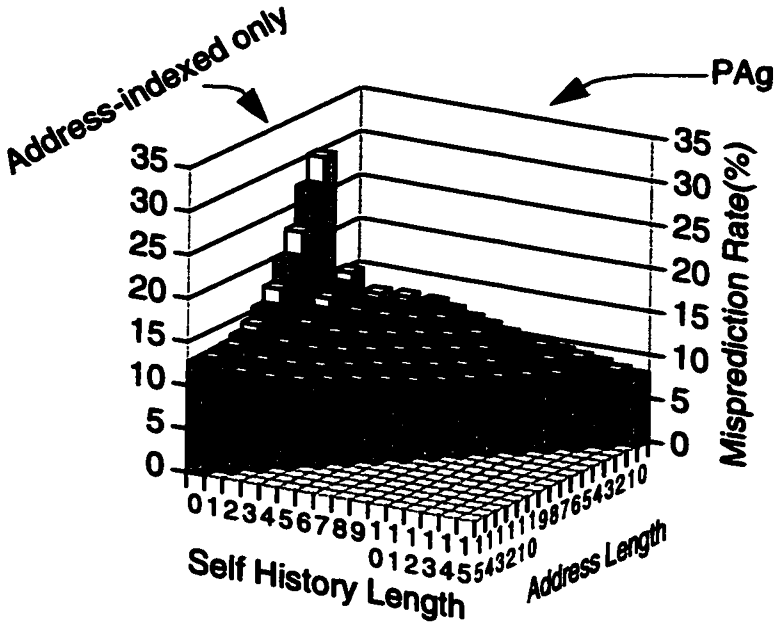
Figure 6.3: Misprediction rates of PAs with an unlimited-size branch history table — *sql95*

between branches in the second-level table is tiny in per-address schemes.

6.2.2 Interference in the first-level table

We have shown that interference in the second-level table of the per-address scheme does not affect the performance significantly, assuming an unrealistically large first-level table. To reflect a real situation, Figure 6.4 and Figure 6.5 present the misprediction rates of the per-address schemes with a limited first-level table (4K-entry, direct-mapped) for the SPEC CINT95, IBS-Ultrix and *sql95* benchmarks.

As can be seen in the figures, although the first-level table becomes smaller, the surfaces are still similar to the ones of the unlimited first-level table, i.e, the plateau-like surfaces. The plateau, however, is elevated to a higher level as the first-level table



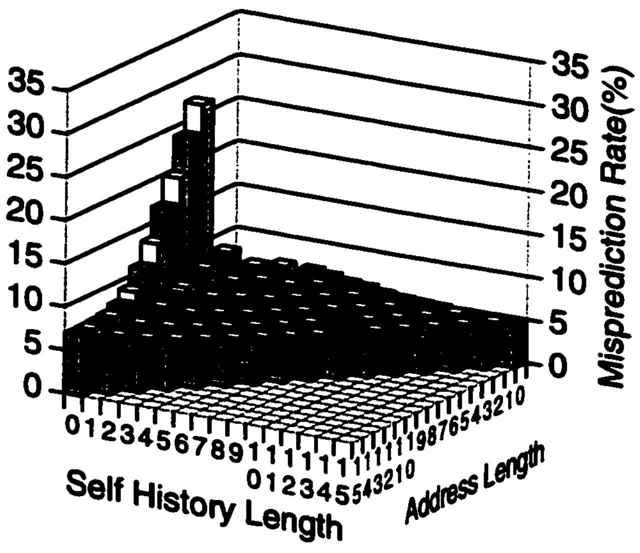


Figure 6.4: Averaged misprediction rates of PAs with a 4K-entry, direct-mapped BHT — SPEC CINT95 (top) and IBS-Ultrix (bottom)

In this figure, the averaged prediction rates of the PAs scheme for the SPEC CINT95 and IBS-Ultrix benchmarks are presented. The second-level table of the PAs scheme ranges from 2^4 counters (tier in back) to 2^{15} counters (tier in front). All possible pair-wised combinations of address bits and history bits for each size of the table are included. Each white and grey tier represents a fixed-size table. The optimal configuration within each tier is marked in black. Comparing with Figure 6.2 shows that the performance of the PAs scheme is slightly restricted by the limited size of the first-level table because of the interference in the table.

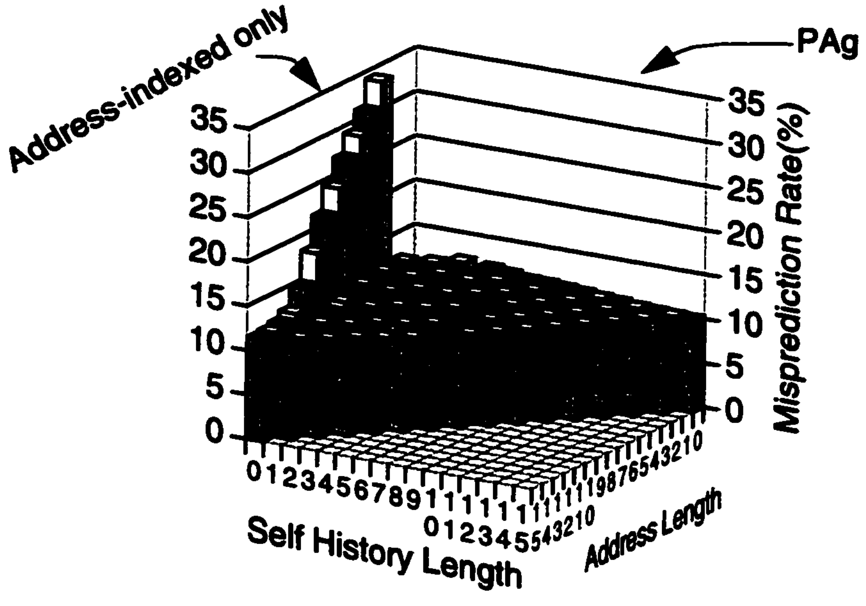


Figure 6.5: Misprediction rates of PAs with a 4K-entry, direct-mapped BHT — *sql95*

This figure shows misprediction rates of PAs with a 4K-entry, direct-mapped BHT for the *sql95* benchmark. By comparing with Figure 6.3, we can easily find that the performance surface of the PAs scheme does not change for *sql95* when the first-level table becomes smaller. However, the entire surface elevates to a higher position due to the interference in the smaller table.

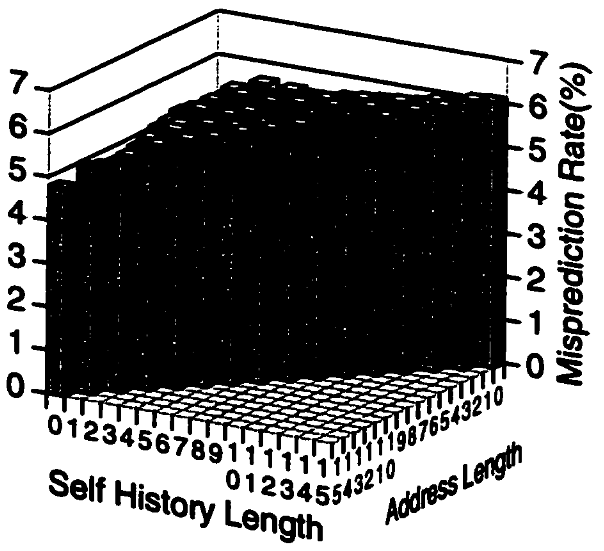


Figure 6.6: Difference in misprediction rates for *sql95* between PAs schemes with unlimited-size and 4K-entry direct-mapped BHTs

This figure shows that when the first-level table of the PAs scheme is restricted (from unlimited-size to 4K-entry, direct-mapped), the performance is significantly degraded for the *sql95* benchmark.

becomes smaller; specifically, there are about 1% and 2% differences in misprediction rate between the unlimited and 4K-entry BHTs for the SPEC CINT95 and IBS-Ultrix benchmarks, respectively. For *sql95*, the difference is even larger, accounting for above 5%, as Figure 6.6 shows. As the first-level table size is restricted, conflicts arise between branches, and therefore accurate per-address history cannot be maintained. Inaccurate history will degrade the prediction accuracy. This degradation is observed in almost all configurations and sizes of the second-level tables examined. The degree of degradation also corresponds to the footprint size of benchmarks. *sql95*, for example, shows larger degradation than the other two benchmark suites because it has more static branches. As another example, Figure 6.7 shows the difference in misprediction rates for the *gcc* benchmark between two BHT sizes: one is unlimited and the other is a 4K-entry, direct-mapped. Figure 6.8 shows the difference for the *xlisp* benchmark—it is much less. *gcc* represents a large benchmark while *xlisp* is for the small benchmark. By comparing these two figures, we can see the impact of different footprint sizes on the design of the first-level table.

6.2.3 The operating system effects

The operating system (OS) effects on the performance of the per-address history scheme is similar to the findings for the global history scheme as discussed in Chapter 4: including OS branches will not make branches less predictable, but OS code may add more static branches, thus making worse the interference problem in the first-level table. Figure 6.9 and Figure 6.10 show the difference in misprediction rates for the *real_gcc* and *mpeg_play* benchmarks, respectively. Both benchmarks have shorter dynamic traces than

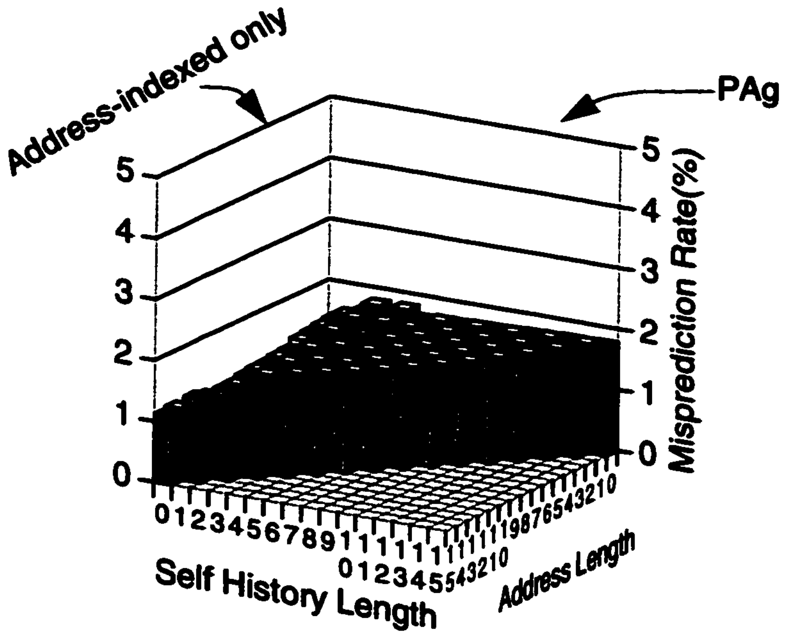


Figure 6.7: Difference in misprediction rates for *gcc* between PAs schemes with unlimited-size and 4K-entry direct-mapped BHTs

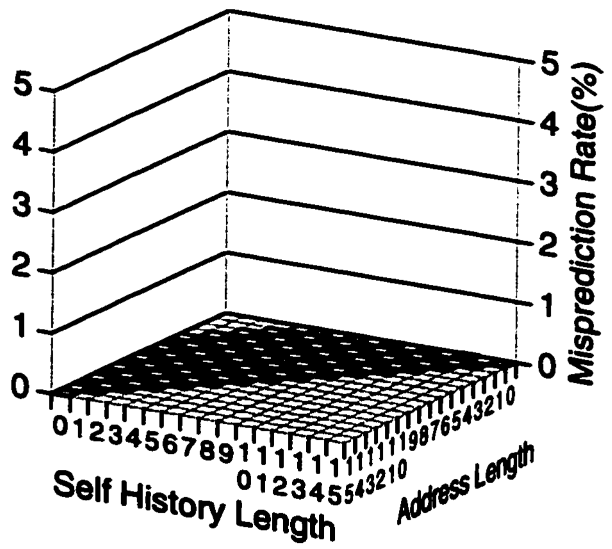


Figure 6.8: Difference in misprediction rates for *xlisp* between PAs schemes with unlimited-size and 4K-entry direct-mapped BHTs

Compared to the results for the *gcc* benchmark shown in Figure 6.7, the performance difference between PAs schemes of two BHT sizes for the *xlisp* benchmark is insignificant. This is because, in *xlisp*, there are only 125 static branches used in 99% of the dynamic branches.

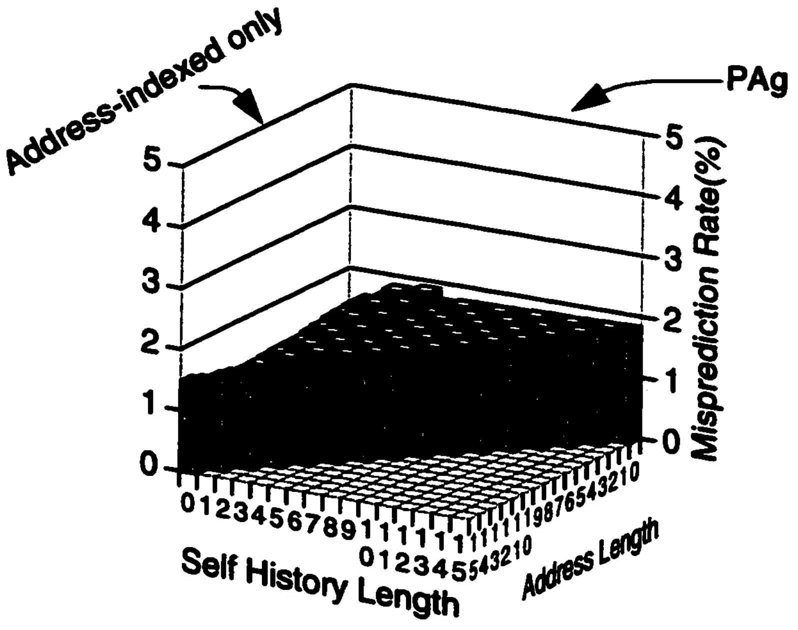


Figure 6.9: Difference in misprediction rates for *real_gcc* between PAs schemes with unlimited-size and 4K-entry direct-mapped BHTs

SPEC CINT95, but they have relatively more static branches. Therefore, the performance of the per-address scheme for these two benchmarks is strongly affected by the sizes of the first-level table.

6.2.4 Predicting for strongly biased branches

We now examine where the most performance is gained from for per-address schemes. We categorize each dynamic branch into one of sixteen groups according to the most recent 4-bit per-address self history pattern of the branch, and then measure the misprediction rate for each class. Figure 6.11 shows the results for two PAg schemes that have different history lengths: one has a 4-bit history, and the other has a 10-bit history. For the 10-bit history, we coalesce the 10-bit patterns that share the same four most recent bits into one class for ease of presentation. If we assume the most significant bit is the

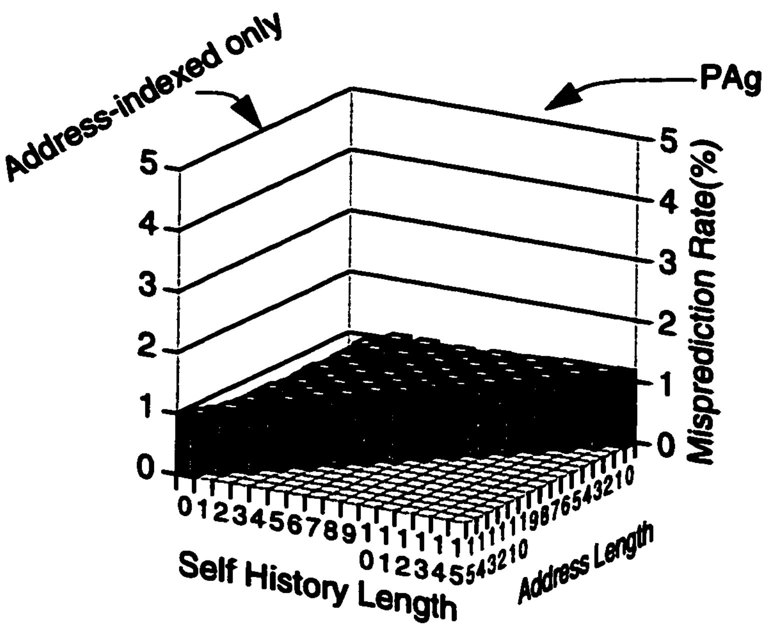
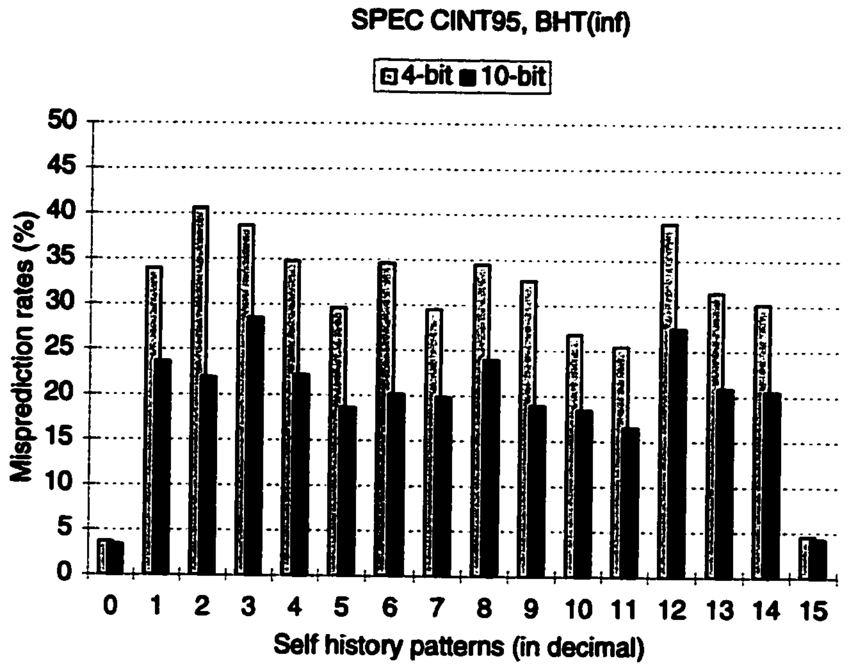


Figure 6.10: Difference in misprediction rates for *mpeg_play* between PAs schemes with unlimited-size and 4K-entry direct-mapped BHTs

Figure 6.9 and Figure 6.10 show that adding OS branches for prediction may cause an interference problem to the first-level table design. The degree of the interference depends on the footprint size of the program. The *real_gcc* benchmark is larger than *mpeg_play*, and therefore *real_gcc* suffers more when the table becomes smaller.

most recent branch outcome, 0000_2 includes all 0000xxxxxx patterns (x means either 1 or 0). In this comparison, we assume an unlimited-size first-level table to avoid interference. We can see that per-address schemes predict accurately for 0000_2 and 1111_2 patterns, but poorly for other mixed patterns. This is not surprising since the 0000_2 and 1111_2 patterns are mostly produced by the strongly-biased branches and strongly-biased branches are easy to be predicted [9]. On the other hand, the mixed patterns may be generated by some branches which are simply not "predictable" by the dynamic approach.

Fortunately, the mixed patterns occur much less frequently. Most of the dynamic branches belong to the 0000_2 and 1111_2 patterns, as shown in Figure 6.12. In this figure, normalized dynamic counts for three history pattern groups, 0000_2, 1111_2 and other mixed



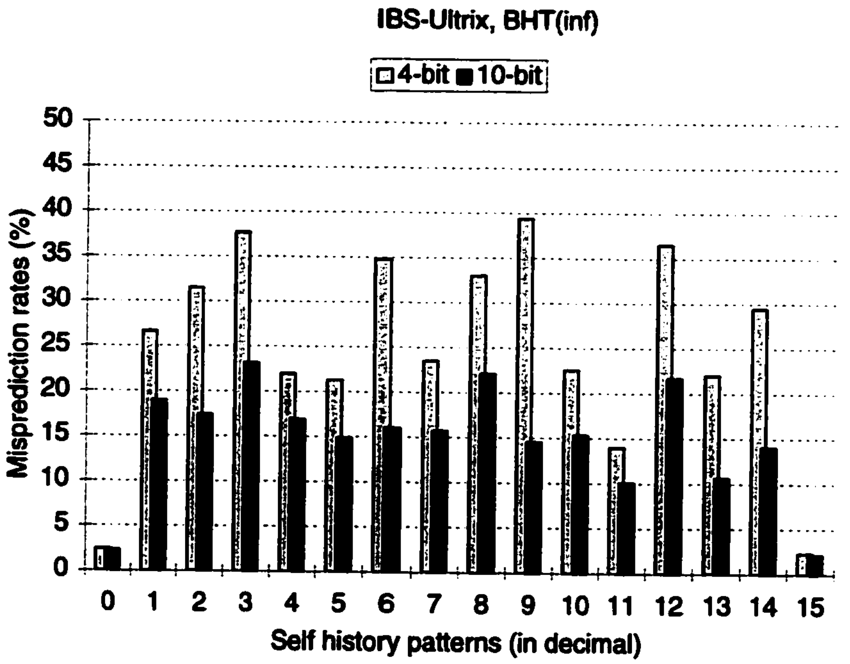
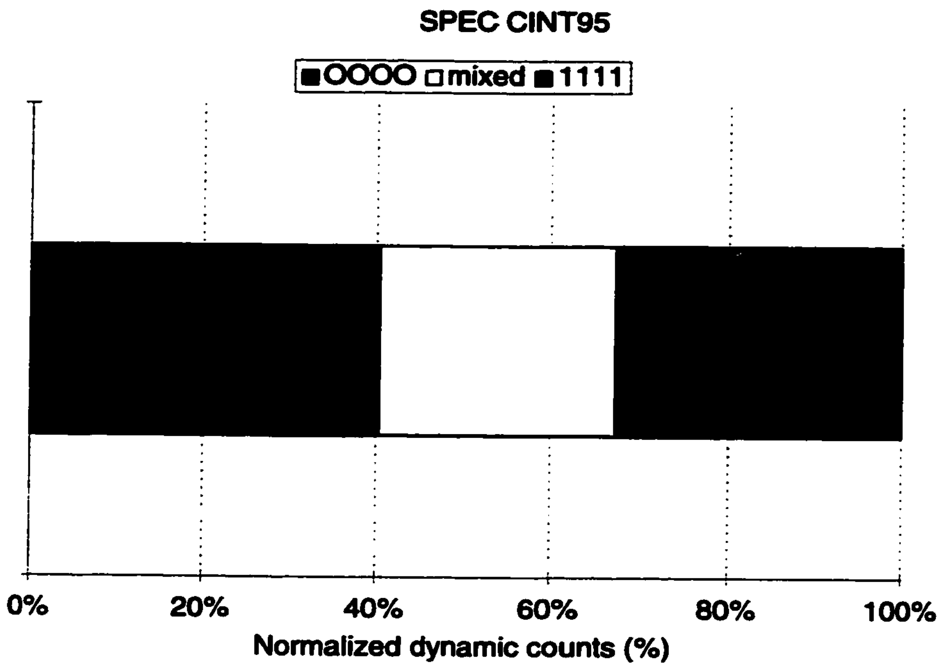


Figure 6.11: Averaged misprediction rates for each per-address history pattern — SPEC CINT95 (top) and IBS-Ultrix (bottom)

In this experiment, each dynamic branch instance is categorized into 16 groups according to the most recent 4-bit per-address history pattern of the static branch. Then the averaged misprediction rates for each pattern is measured for the two benchmark suites. The X axis lists all possible 16 4-bit history patterns in decimal and the Y is for the misprediction rate. The size of the first-level tables, or the BHT, is unlimited.



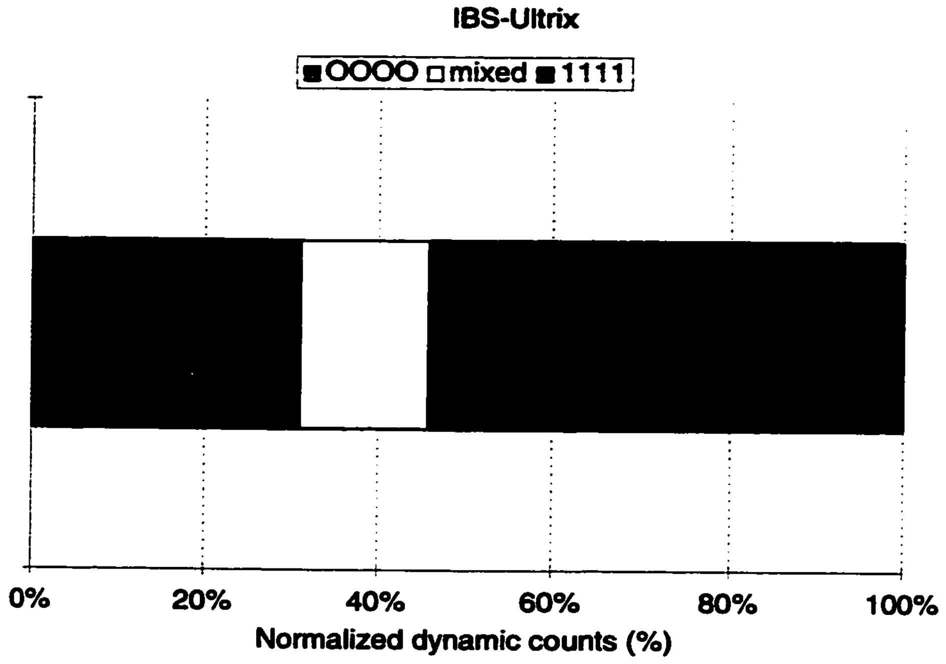


Figure 6.12: Normalized dynamic counts for three per-address history pattern groups — SPEC CINT95 (top) and IBS-Ultrix (bottom)

In this figure, each dynamic branch instance is categorized into 16 groups according to the most recent 4-bit per-address history of the static branch. Then, except for 0000_2 and 1111_2 , all other patterns are grouped in the mixed group. The normalized dynamic counts for each patterns is obtained by normalizing the dynamic counts with total numbers of dynamic branches of each of the two benchmark suites. The size of the first-level tables, or the BHTs, is unlimited. We see that for the SPEC CINT95 suite, the 0000_2 and 1111_2 patterns account for more than 75% of total dynamic branches. In the IBS-Ultrix suite, these two patterns include even more branches.

patterns, are presented. We can find that 0000_2 and 1111_2 have contributed above 75% of total dynamic branches. Because of the prevailingly high dynamic counts of strongly-biased (easy-to-predict) patterns, the per-address scheme can sustain its high prediction accuracy.

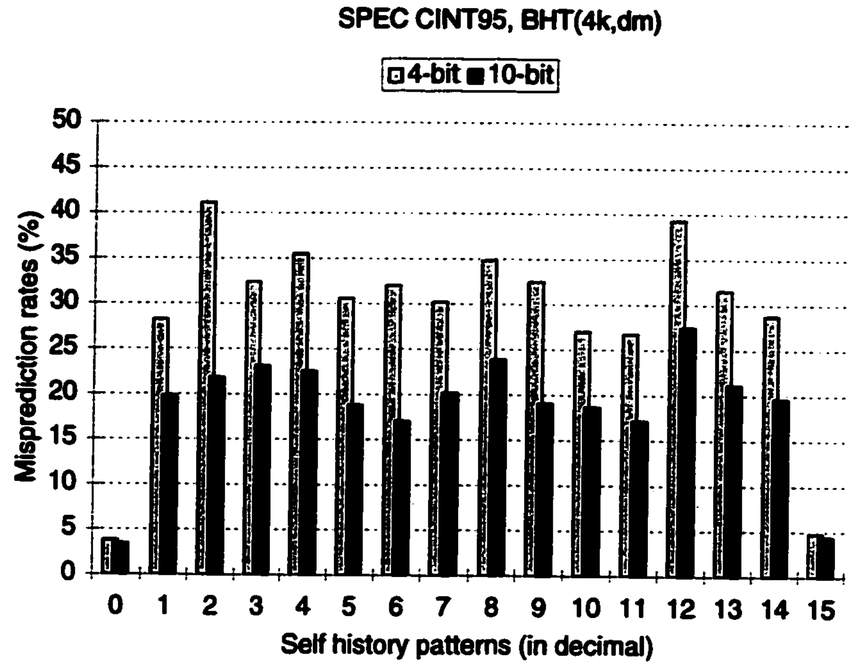
Recall that the measurement in Chapter 3 has shown the IBS-Ultrix benchmarks have on average more strongly biased static branches than the SPEC CINT95 benchmarks; the statistics in Figure 6.12 agrees with this measurement by showing that the IBS-Ultrix benchmarks generate a larger fraction of dynamic branches that belong to the 0000_2 and 1111_2 patterns. Therefore, if the first-level table does not have an interference problem, the per-address scheme can perform very well for the IBS-Ultrix benchmarks.

As the first-level table size is restricted, the misprediction rates for each history pattern may vary, but it is still very accurate for the 0000_2 and 1111_2 patterns, as shown in Figure 6.13.

6.2.5 Adaptivity

The per-address scheme gains most of the benefit by accurately predicting for the strongly-biased static branches. For the weakly-biased branches, or the mixed per-address history patterns, it predicts poorly.

For the 0000_2 and 1111_2 patterns, it is very likely to repeat the last outcome the next time. Simply fixing the prediction value at 0 and 1 for these two patterns can already achieve very good prediction accuracy. Actually, we will see that the fixed prediction method performs better than the adaptive one for these two strongly-biased patterns.



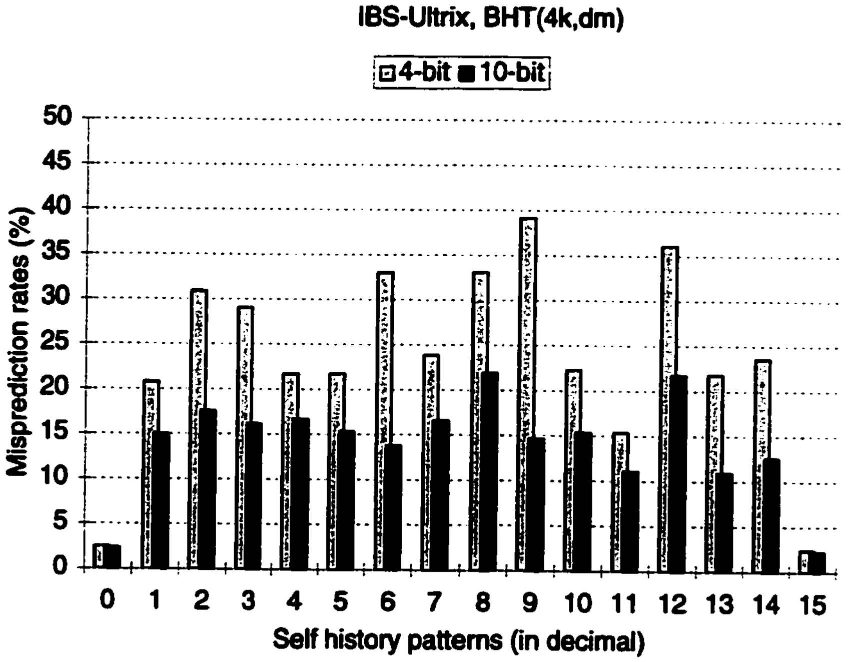


Figure 6.13: averaged misprediction rates for each per-address history pattern — SPEC CINT95 (top) and IBS-Ultrix (bottom)

In this experiment, each dynamic branch instance is categorized into 16 groups according to the most recent 4-bit per-address history of the static branch. Then the averaged misprediction rates for each patterns is obtained for the two benchmark suites. The x axis lists all possible 16 4-bit per-address history patterns in the decimal. Also, the first-level tables, or the BHTs, is 4K-entry, direct-mapped. For a miss in the BHT, the corresponding history register is raset to 1100_2 for the 4-bit scheme and to 1100001111_2 for the 10-bit scheme (see later).

Therefore, for these two patterns, adaptivity is not necessary.

For the mixed patterns, a predictor that can discriminate between branches and assign a separate adaptive counter to each of them should be able to achieve higher prediction accuracy. This is because when branches behave differently they need separate counters to capture their distinct behaviors. However, if the mixed patterns occur infrequently and if predicting them with fixed values may already provide fair overall prediction accuracy (and require less hardware), sacrificing adaptivity may be a better choice as far as the cost-effectiveness is concerned.

With this in mind, we hypothesize that a per-address history scheme using a static-trained fixed-value second-level table may deliver competitive performance to an adaptive scheme. In the next section, we will justify the hypothesis by examining the static-trained scheme, PSg, and identify the conditions when adaptivity is needed.

6.3 PSg — the per-address history scheme using a static-trained second-level table

The statically trained scheme was first proposed in [4] and has been studied by Lee and Smith [29]. It was also further examined and termed PSg by Yeh and Patt [56, 57]. The PSg scheme in these two studies is similar in structure to the per-address history adaptive scheme, but its prediction for a per-address history pattern is pre-determined by profiling. This PSg scheme requires the second-level table to be loaded with a trained data set every time a new program starts to run, thus incurring a certain amount of overhead. Early studies showed that the static scheme is worse than the adaptive one. As a result, researchers concluded PSg is not competitive with the adaptive schemes [56].

However, the comparative study between PAg and PSg mentioned above only examined a small benchmark set, SPEC89, which might be biased to the importance of adaptivity.

The notion that adaptivity may not be that important can be better understood as follows. The first-level table of both the PAg and PSg schemes divides the branch outcome stream into substreams based upon the recent history of the individual branches. Instances of separate branches that happen to have the same history will be combined in a substream. This mixing of branches in a substream will not occur if only a few branches are active. A single branch may have a strong characteristic behavior, for example, a branch may be taken every third execution. For this particular branch, the 0010_2 per-address history pattern indicates that the branch is going to be taken the next time. For most branches, however, this pattern is more likely to indicate that the branch will be not-taken. If all the instances with the 0010_2 pattern arise from the branch with the three-cycle periodical behavior, an adaptive counter will quickly begin predicting 1, thus having better performance.

Nevertheless, for large footprint programs where there are significant numbers of branches, instances of the 3-cycle periodical branch can hardly dominate, so the counter will continue to predict 0, resulting in misprediction rates similar to that of a PSg scheme. If a burst of instances from this branch do cause a change in prediction in an adaptive scheme, the subsequent instances for other branches are likely to be mispredicted and the overall result will not be much better than the static method. Therefore, the statically trained scheme should be re-examined with large footprint programs.

In the following, we will re-examine the performance of the PSg scheme using our benchmarks. Also, we are interested in a PSg scheme with a second-level table which is built up by some heuristics, instead of by profiling [44].

6.3.1 The algorithm-derived static table

The algorithm employed for the static table is based on simple heuristics. We construct a table for n-bit history patterns by first identifying all of the patterns that can be produced using cycles of at most $n/2$ bits. For each of these patterns, we predict the next branch will continue the pattern. For example, the eight-bit pattern 11101110_2 is interpreted as two four-bit pattern 1110_2, and thus the next outcome is determined to be 0—the start of a 1110 sequence. If the eight-bit pattern is 11011011_2, then we consider it to represent a repeated three-bit pattern 110_2.

If this is not successful, we look for shorter patterns. We then ignore the oldest two bits and again check if there is any repeating patterns[1]. This is because we project that some of patterns may represent a transition between two modes of cyclic behavior. For example, for the pattern 10101011_2, we will ignore the oldest two bits, 11, and treat the pattern as 101010xx (x for "don't care"). Then we can find a repeating pattern of 10, and predict 0 as the next outcome. This process will continues, substituting two additional don't care bits with each iteration.

If a pattern cannot be processed by the above two procedures, then the pattern must differ in their two most significant bits, for example, 10111110_2. (This is because the previous step has already checked repeating patterns for the length of 2.) For this kind of patterns, we choose the majority voting from all bits in the pattern. If 0 dominates, 0 is used as the prediction; otherwise, 1 is used. In the example above, the prediction will be 1.

1. If n is an odd number, we first ignore the oldest bit and check the repeating patterns. (Then, we ignore the next two oldest bits as we do for the case of n being even.)

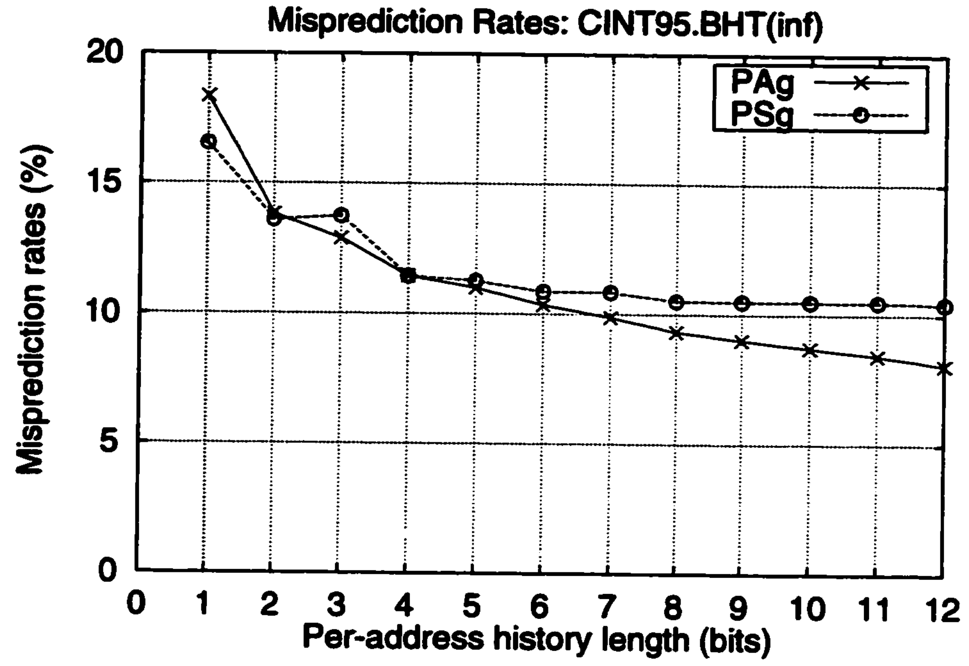
If the number of bits is even, then the oldest bit is ignored to get the majority voting result. For example, for 100110_2 the prediction is 1.

With this algorithm-derived second-level table, the PSg does not require profiling and reloading for every program. We refer to it as PSg(algo). The algorithm employed in this study is by no means optimal, and there is an open area to refine the algorithms based on different heuristics. However, our algorithms provides quite good performance results, as we will see.

In addition to avoiding reloading the second-level table, another major advantage of this PSg(algo) over the original PSg scheme is that the PSg(algo) does not require large memory cells for the second-level table. (The original PSg requires an array of memory cells to load a different set of prediction values for a new program.) The fixed prediction values of the PSg(algo) can be implemented as fixed logic, thus saving hardware cost significantly.

6.3.2 Performance evaluation

In this subsection, the performance comparison between the PSg(algo) and PAg is performed. Figure 6.14 presents average misprediction rates for SPEC CINT95 and IBS-Ultrix benchmarks. In this figure, an unlimited-size BHT is assumed to avoid interference in the first-level table. As can be seen from the figure, when the history length is short, the PSg scheme performs very closely to the PAg scheme. When the history length is long, there is a gap of 2-3% prediction accuracy between the PSg and PAg schemes for the SPEC CINT95 benchmarks. This is because the likelihood of the instances of branches with dissimilar behaviors mixing in the stream of a pattern diminishes. Therefore, PAg



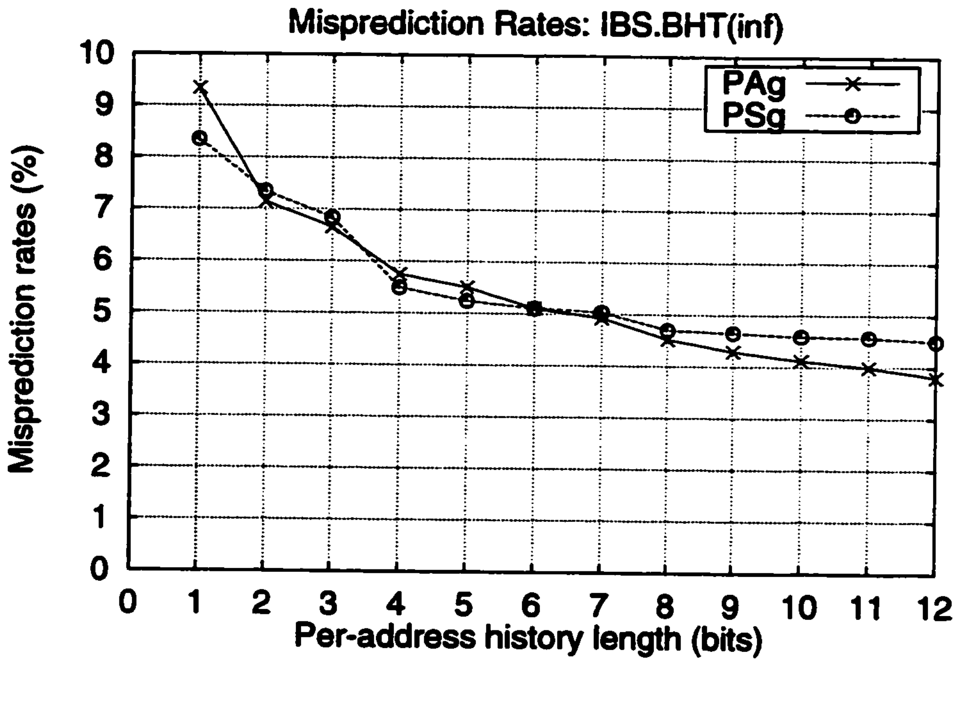


Figure 6.14: Averaged misprediction rates for the PAg and PSg(algo) schemes with unlimited-size BHTs — SPEC CINT95 (top) and IBS-Ultrix (bottom)

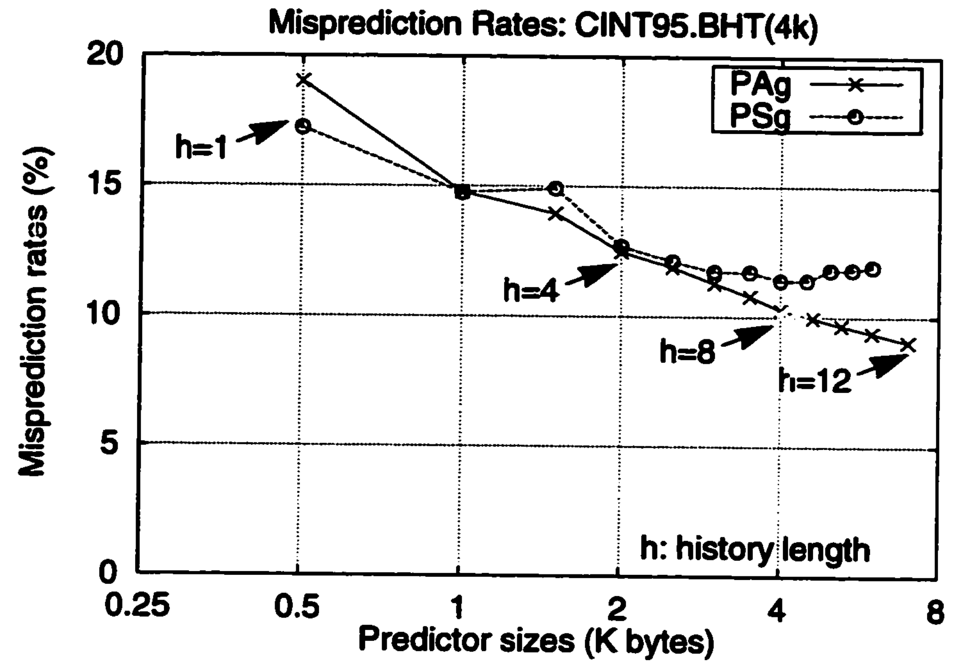
can continue to improve but PSg(algo) cannot.

For the IBS benchmarks, the PSg scheme keeps its performance close to the PAg scheme up to the point when the history length is 8. For short histories, PSg can even perform better than PAg, and for histories longer than 8 bits, the difference between these two schemes is only 0.6%. When the stream of a pattern is mixed by many static branches, the performance can be determined by the majority behavior of these branches; in this case, the algorithm-derived static table becomes effective.

To evaluate the performance of the PSg(algo) in a real situation, Figure 6.15 presents the averaged misprediction rates for the PSg and PAs schemes with a 4K-entry, direct-mapped BHT. In this experiment, the history registers in the first-level table are shared among branches—the per-address history is mixed and, thus, polluted.

As shown in the figure, when the history length is shorter than 8 bits, the PSg can still perform well, but when the history length is longer than 8 bits, PSg performs badly. Actually its performance is degraded after 8-bit history. This is because as history registers are shared, they cannot provide accurate per-address history information of branches which is essential to the algorithm used in the PSg(algo) scheme. We will provide detailed explanation for this degradation shortly. On the other hand, PAg can still sustain a good performance by being adaptive.

Since PSg(algo) suffers from inaccurate per-address history due to limited size BHTs, we proposed a modified PSg scheme which can always extract correct per-address history from the polluted history table. This proposed PSg scheme will be presented later.



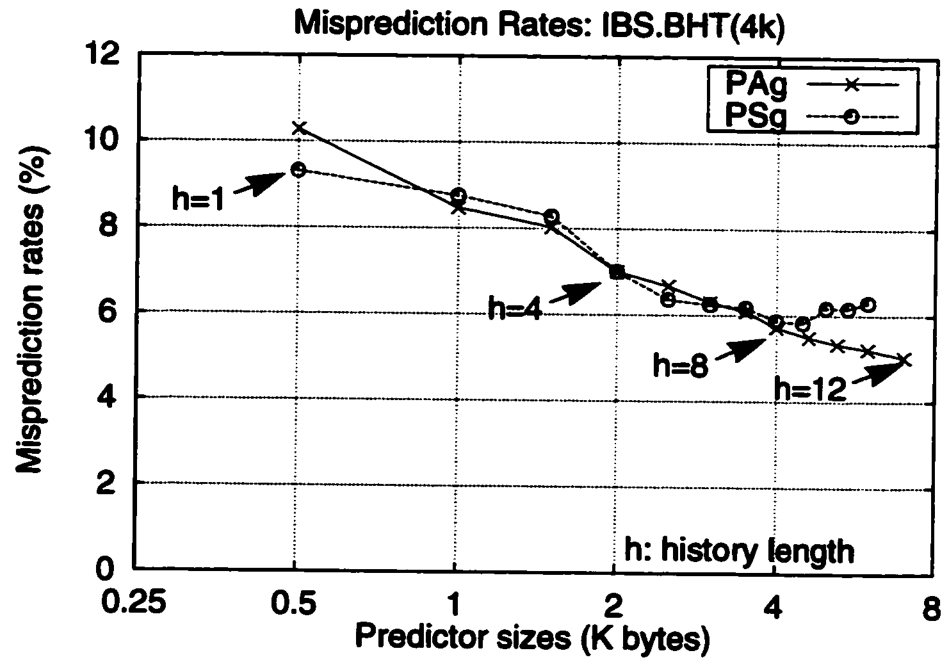
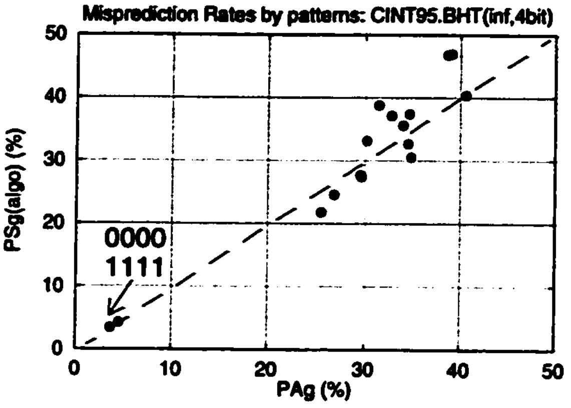


Figure 6.15: Cost-effective comparison between PAg and PSg(algo) — SPEC CINT95 (top) and IBS-Ultrix (bottom)

In this figure, a cost-effective comparison is performed. The averaged misprediction rates versus total numbers of bits used in predictors are plotted for the two benchmarks suites. Both PAg and PSg(algo) have 4K-entry, direct-mapped BHTs. For each scheme, the per-address history length ranges from 1 to 12 bits. Branches missing in the BHT are assumed taken. As shown in the figure, when the BHT is restricted, the performance of the PSg(algo) becomes worse, especially for longer history. A restricted BHT cannot record accurate per-address history all the time, and the accurate history is crucial to the performance of the PSg(algo).



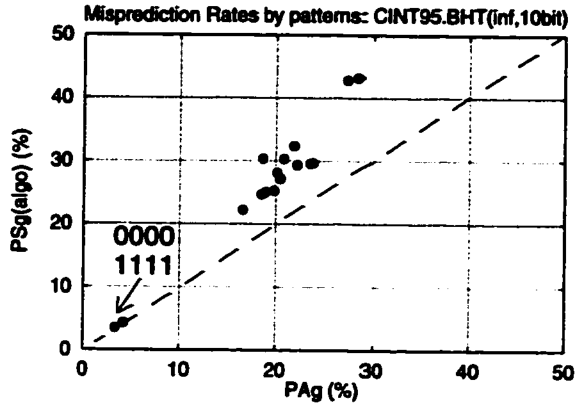


Figure 6.16: Averaged misprediction rates of each history pattern, compared between PAg and PSg(algo), for SPEC CINT95.

The scatter plot on the left shows the misprediction rates for each of the sixteen possible per-address history patterns for a 4-bit history. The x and y coordinates of a point are determined by averaging, respectively, a pattern's PAg and PSg(algo) misprediction rates across all of the SPEC CINT95 benchmarks. Thus, points lying close to the diagonal line represent patterns for which the predictive powers of the two schemes are nearly equivalent. Points lying above this line represent patterns for which the PAg scheme is superior, and points below the line for the patterns for which PSg(algo) is superior. In this comparison, both schemes have unlimited-size branch history tables.

The scatter plot on the right shows the misprediction rates for a 10-bit history. We coalesce the 10-bit patterns sharing the same four most recent bits into single points. For example, 0000 represents the misprediction rates for all 0000xxxxxx patterns.

As can be seen in the figure, PSg(algo) does as well as the PAg scheme for the 0000 and 1111 patterns. For the mixed patterns, PSg(algo) is still as good as PAg for shorter history, but it becomes worse than PAg for longer history. For longer history, fewer static branches contribute to the branch stream of a history pattern, and therefore adapting to unique behaviors of the few branches becomes important.

6.3.3 Comparison of the performance of PSg and PAs schemes for each pattern

In this subsection, the misprediction rates and the prediction error for each per-address history pattern are presented and compared between the PSg(algo) and PAs schemes.

Figure 6.16 and Figure 6.17 compare the misprediction rates of the two schemes for each per-address history pattern. Both schemes have an unlimited-size BHT. In these scatter plots each point represents a comparison of misprediction rates for a pattern between the two schemes, and there are sixteen points in each plot. There is a diagonal line; above the line, PAg predicts better than PSg(algo) for the corresponding pattern, and

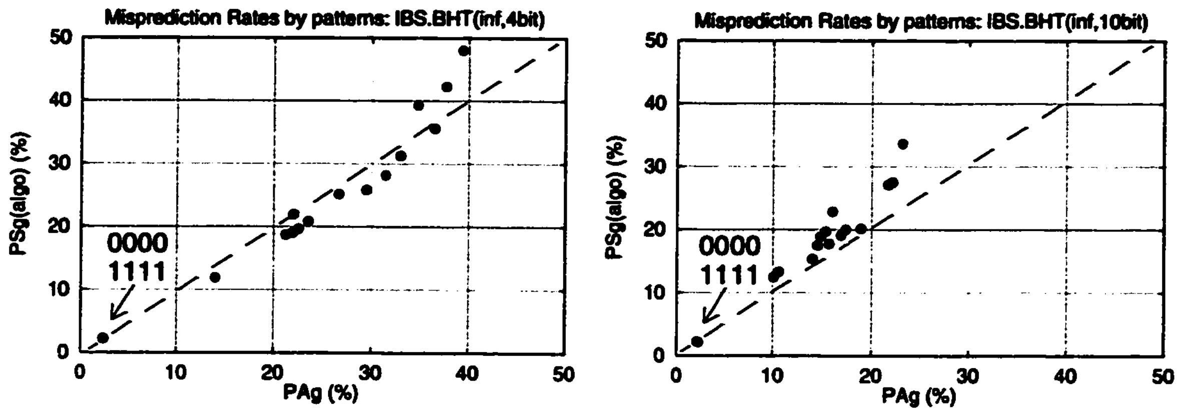
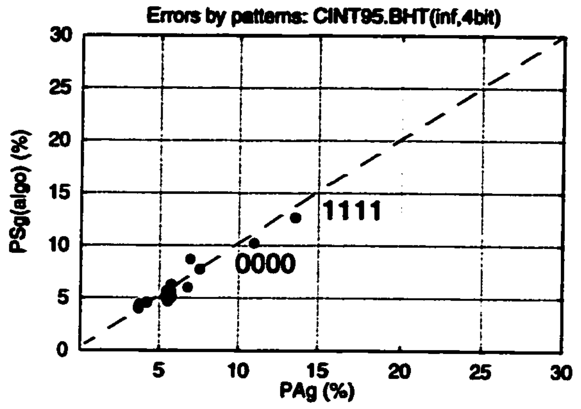


Figure 6.17: Averaged misprediction rates of each pattern, compared between PAg and PSg(algo), for IBS-Ultrix.

The scatter plot on the left shows the misprediction rates for each of the sixteen possible patterns for a 4-bit history. The x and y coordinates of a point are determined by averaging, respectively, a pattern's PAg and PSg(algo) misprediction rates across all of the IBS-Ultrix benchmarks. THus, points lying close to the diagonal line represent patterns for which the predictive powers of the two schemes are nearly equivalent. Points lying above this line represent patterns for which the PAg scheme is superior, and points below the line for the patterns for which PSg(algo) is superior. In this comparison, both schemes have unlimited-size branch history tables.
The scatter plot on the right shows the misprediction rates for a 10-bit history. We coalesce the 10-bit patterns sharing the same four most recent bits into single points. For example, 0000 represents the misprediction rates for all 0000xxxxxx patterns.
As can be seen in the figure, PSg(algo) does as well as the PAg scheme for the 0000 and 1111 patterns. For the mixed patterns, PSg(algo) can be better than PAg for shorter history in the IBS benchmarks. However, it becomes worse than PAg for longer history. For shorter history, more static branches contribute to the branch stream of a history pattern. In this case, adaptivity cannot provide benefits because different biased branches may cause interference for the counter. Fixing prediction value, on the hand, may deliver a better overall prediction accuracy.

vice versa. For both benchmark suites, we can see that there are two groups of the points in each plot; one group of which has lower misprediction rates than the other. The group having lower misprediction rates contains only two patterns, 0000_2 and 1111_2, while the other group contains the remaining fourteen mixed patterns. This result implies that both PAg and PSg(algo) predict very well for all-1's and all-0's patterns, but badly for the other patterns. For the all-1's and all-0's patterns, there is no obvious performance difference between PSg(algo) and PAg, for either of the history lengths and for either of the benchmark suites. (Actually, we will see that PSg(algo) is slightly better than PAg for IBS.) As long as the most recent four instances of a branch were all taken (or not-taken),



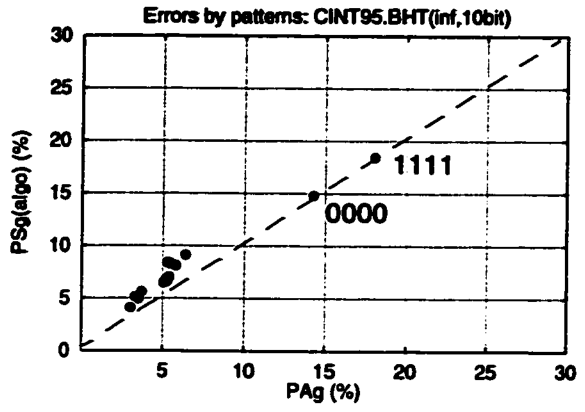


Figure 6.18: Error distribution for the per-address schemes with unlimited-size BHTs for SPEC CINT95 (4-bit history on the left and 10-bit on the right).

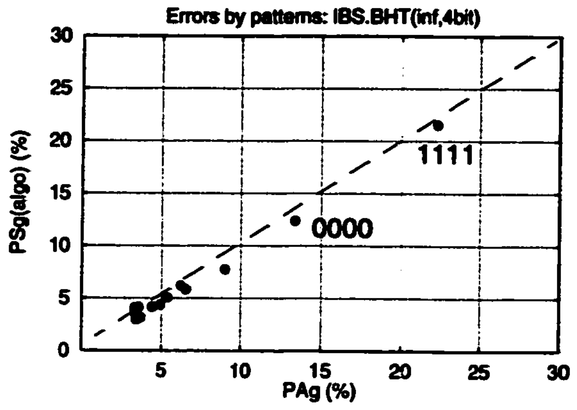
The scatter plot on the left shows the relative number of error due to each of the 16 possible 4-bit per-address history patterns. The error is normalized for each benchmark, with the total number of error made by the PAg scheme on a given benchmark set to 100%. The normalized numbers were then averaged across the benchmarks. The scatter plot on the right shows the relative number of error for 10-bit history. As before, we coalesce the patterns with the same four most recent bits. For example, with 10-bit history, error due to the patterns 0000xxxxxx is represented by the point, 0000. In this comparison, an unlimited-size BHT is assumed for both schemes.

As shown in the plots, most of the misprediction error is from the 0000 and 1111 groups. Therefore, predicting accurately for these two patterns is more critical to the performance than for other mixed patterns.

predicting the next outcome for the branch with the repeating outcome is always a good strategy.

On a closer examination of the results for the SPEC CINT95 benchmarks, we find that for the mixed patterns, PSg(algo) performs as well as or even better than PAg for the 4-bit history, but worse than PAg for the 10-bit history. Adaptivity is important to predict the outcome for mixed patterns when longer histories are employed. The same trend can be observed in the IBS benchmarks, but the difference between two schemes becomes less. When a large number of static branches are involved, the situation that a few branches with unique behavior dominate the stream of a pattern becomes unlikely.

Although PSg(algo) performs worse for 14 out of 16 per-address history patterns, this does not imply that the overall performance of PSg(algo) is significantly worse,



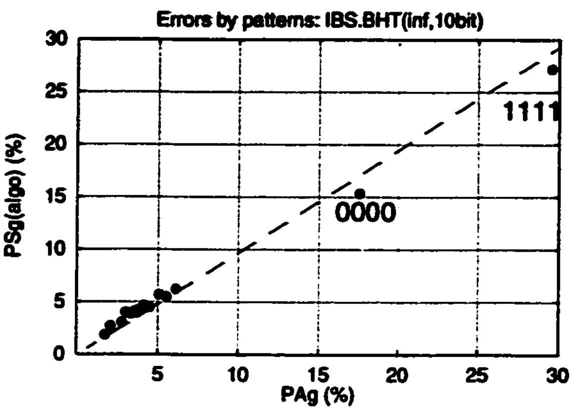


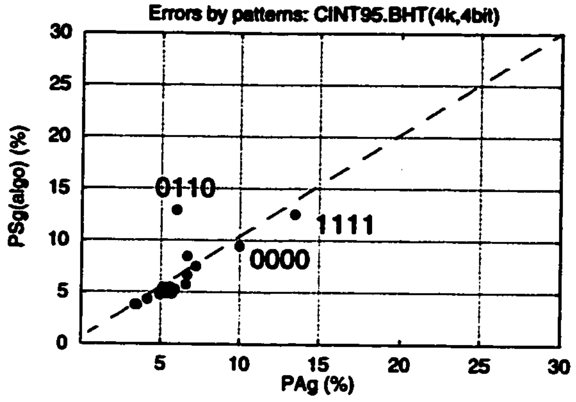
Figure 6.19: Error distribution for the per-address schemes with unlimited-size BHTs for IBS-Ultrix (4-bit history on the left and 10-bit on the right).

The scatter plot on the left shows the relative number of error due to each of the 16 possible 4-bit per-address history patterns. The error is normalized for each benchmark, with the total number of error made by the PAg scheme on a given benchmark set to 100%. The normalized numbers were then averaged across the benchmarks. The scatter plot on the right shows the relative number of error for 10-bit history. As before, we coalesce the patterns with the same four most recent bits. For example, with 10-bit history, error due to the patterns 0000xxxxxx is represented by the point, 0000. In this comparison, an unlimited-size BHT is assumed for all the schemes.

As shown in the plots, PSg(algo) performs better than PAg for the two most frequent patterns, 0000 and 1111 in the IBS benchmarks. That is the reason why PSg(algo) is competitive to the PAg for the IBS benchmarks.

because the 14 patterns only account for small portion of the dynamic branch stream. Figure 6.18 and Figure 6.19 compare the percentages of error due to misprediction for each history pattern between the two schemes. As shown in the figures, the all-1's and all-0's patterns account for most of the misprediction error. These two patterns have much more dynamic branches to predict than the other 14 mixed patterns. This is especially true for the IBS benchmarks. Improving the prediction accuracy for these two patterns is more critical than for the other mixed patterns. PSg(algo) is designed to predict extremely well for these two patterns, and gains accordingly.

Figure 6.20 and Figure 6.21 further examine percentages of error for the patterns when a 4K-entry, direct-mapped BHT is used. In these two figures, the history patterns are obtained from the contents of the branch history registers, and the contents may include



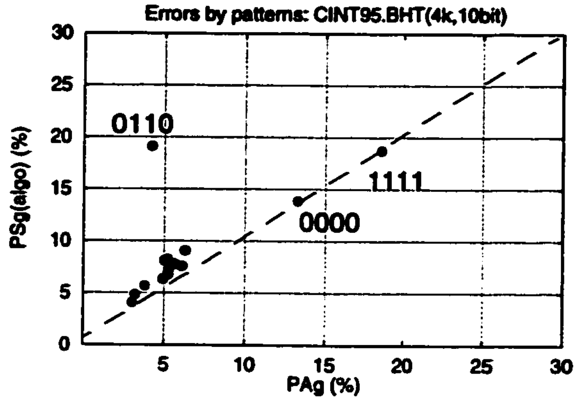


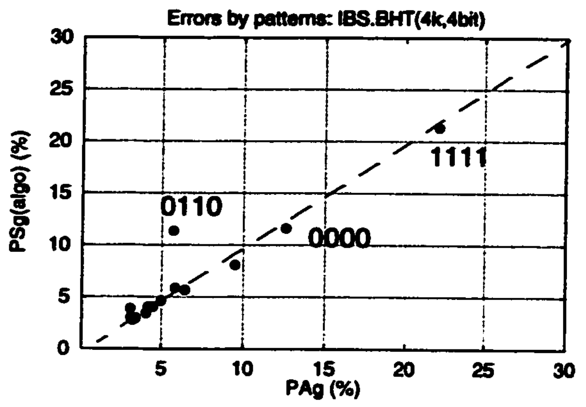
Figure 6.20: Error distribution for the per-address schemes with limited-size BHTs for SPEC CINT95 (4-bit history on the left and 10-bit on the right).

The scatter plot on the left shows the relative number of error due to each of the 16 possible 4-bit per-address history patterns. The error is normalized for each benchmark, with the total number of error made by the PAg scheme on a given benchmark set to 100%. The normalized numbers were then averaged across the benchmarks. The scatter plot on the right shows the relative number of error for 10-bit history. As before, we coalesce the patterns with the same four most recent bits. For example, with 10-bit history, error due to the patterns 0000xxxxxx is represented by the point, 0000. In this comparison, a 4K-entry, direct-mapped BHT is assumed for all the schemes. PSg(algo) performs badly for the 0110 pattern because of the conflicts in the BHT. Please see the text for detailed explanation.

outcomes from more than one static branch because the registers are shared.

As shown in the figures, the observation obtained from the unlimited-size BHT can still hold for the limited-size BHT, except the point for the pattern, 0110_2. In our experiments, the branch history register was reset to 1100_2[1] or 1100011111_2 for the 4-bit and 10-bit schemes, respectively, when another static branch starts to use the branch history register. This condition can only be detected if the branch history register have address tags. The content of the register will then become 0110_2 or 0110001111_2 if the branch outcome turns out to be not-taken (110_2 and 110001111_2 are from the reset values.). In fact, the only relevant history is the first 0. Based on this, the next branch should be predicted not-taken. However, PSg(algo) will predict the next outcome as taken,

1. An exhaustive study showed that any pattern, except all-1's and all-0's, was suitable for a reset value [14].



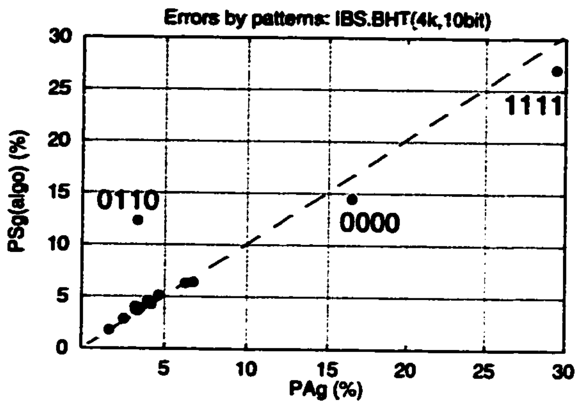


Figure 6.21: Error distribution for the per-address schemes with limited-size BHTs for IBS-Ultrix (4-bit history on the left and 10-bit on the right).

The scatter plot on the left shows the relative number of error due to each of the 16 possible 4-bit per-address history patterns. The error is normalized for each benchmark, with the total number of error made by the PAg scheme on a given benchmark set to 100%. The normalized numbers were then averaged across the benchmarks. The scatter plot on the right shows the relative number of error for 10-bit history. As before, we coalesce the patterns with the same four most recent bits. With 10-bit history, error due to the patterns 0000xxxxxx is represented by the point, 0000. In this comparison, a 4K-entry, direct-mapped BHT is assumed for all schemes. As shown in the plots, PSg(algo) performs better than PAg for the two most frequent patterns, 0000 and 1111 in the IBS benchmarks. That is the reason why PSg(algo) is competitive to the PAg for the IBS. Also, PSg(algo) performs badly for the 0110 pattern because of the conflicts in the BHT. Please see the text for a detailed explanation.

thus the anomaly with the 0110_2 point. We need a method that recognizes that only part of the branch history is relevant. Next, we will propose a new PSg(algo) scheme that can extract the correct per-address history from a polluted history register and thus improve the performance.

6.3.4 PSg using correct history

A "correct" history means we need to be able to specify three possibilities for a branch history entry: taken (1), not-taken (0), or not relevant (x). In the previous example the sequence is 0xxx (4bits). Thus the simple shift register will not work. Instead, for branch histories of length n we propose a finite state machine with n+1 states. The extra state allows us to encode not only the sequences of length n but also those of length n-1, n-

2, ... 1, i.e., sequences where some of the entries are irrelevant. Figure 6.22 shows a 4-bit encoding for a 3-entry branch history register—the transition logic follows directly. With such a modified BHT, the PSg(algo) scheme always has the correct per-address history to make a prediction.

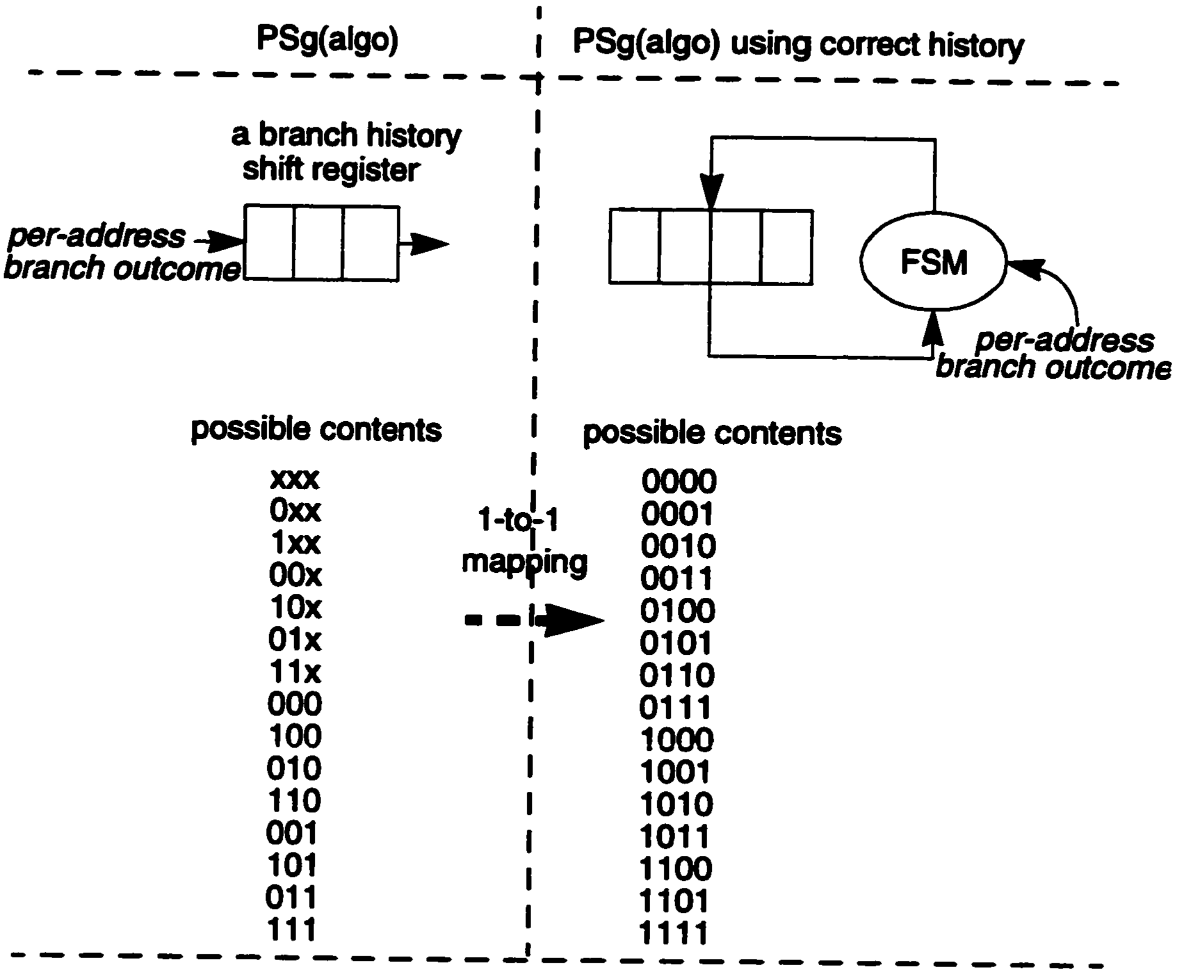
Figure 6.23 compares the performance for such a PSg scheme with the original PSg(algo) and PAg schemes. For fair comparison, the total cost is considered for each scheme: the BHT is included for the two PSg schemes, and the tables of both levels for the PAg scheme.

As can be seen, using correct per-address history can improve performance for the PSg scheme. The new scheme can even perform better than the PAg scheme for the IBS benchmarks, where a significant number of branches are involved. It shows that for smaller BHTs where more interference exists, the scheme that can extract correct per-address history will be more effective.

To summarize, under realistic workloads such as the IBS benchmarks, PSg(algo) is a competitive alternative to the PAg scheme. PSg(algo) performs quite close to PAg, and PSg(algo) can be implemented with faster circuitry because it does not need on-chip memory for the counter table. A modified BHT that can extract correct history for the PSg scheme was also proposed to resolve the interference problem in the BHT. This modified BHT is effective and can further slightly improve the original PSg scheme.

6.4 Cost-effective analysis

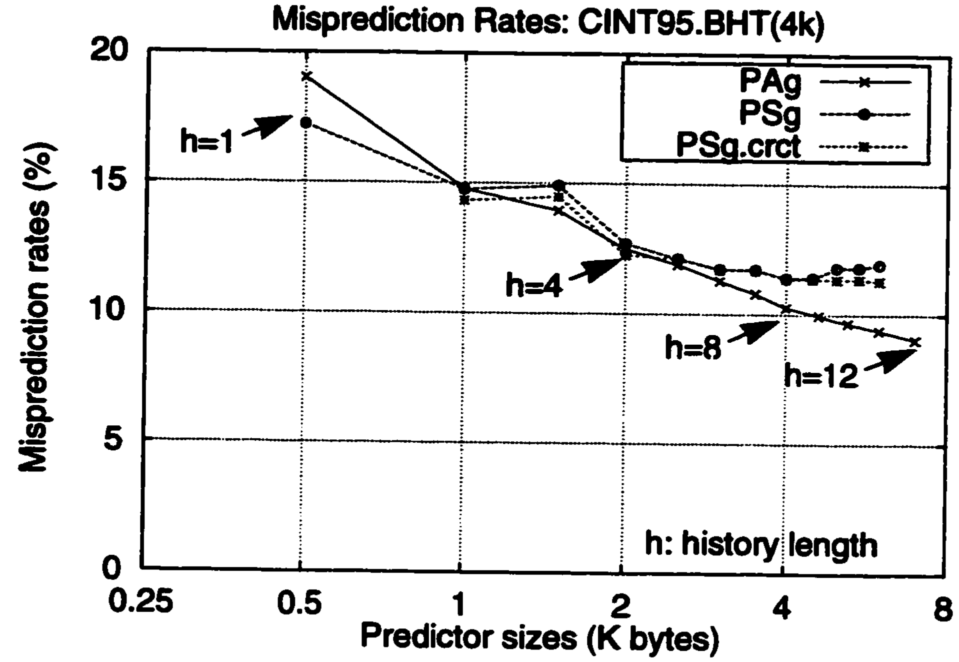
A cost-effective performance analysis by Yeh and Patt [59] has shown that per-address history schemes are more effective for the floating-point benchmarks of SPEC89



x: outcomes from other static branches.
a 4-bit state can encode a 3-bit correct per-address history.

Figure 6.22: An example for the PSg scheme using correct 3-bit per-address history.

In this example, on the left is the original PSg(algo), which employs 3-bit per-address history shift registers. Because of sharing the history register, it is possible the register contains outcome history of the current static branch as well as of other static branches. An enhanced PSg, shown on the right, uses an extra bit in the register to capture this transition. This new PSg can always supply a correct per-address history for prediction.



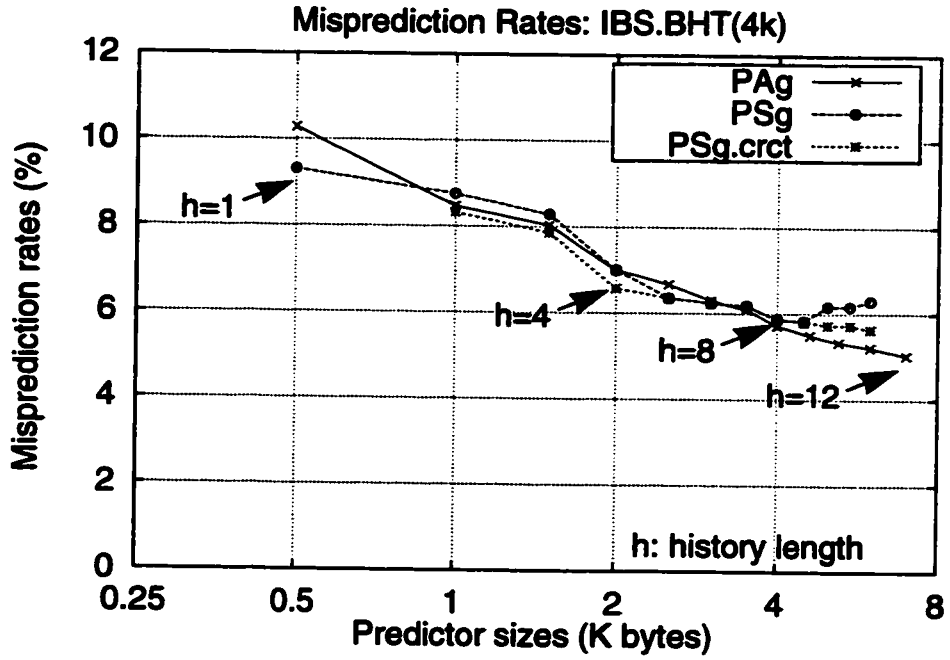


Figure 6.23: Performance evaluation for the PSg(algo) using correct per-address history — SPEC CINT95 (top) and IBS-Ultrix (bottom)

In this figure, the averaged misprediction rates for the PSg(algo) using correct per-address history versus total numbers of bits used in the predictor are plotted for the two benchmarks suites. For comparison, both PAg and the original PSg(algo) are also included. All three schemes have 4K-entry, direct-mapped branch history tables. For each scheme, per-address history lengths ranging from 1 to 12 are examined. Missing branches in the BHTs are always assume taken. As shown in the figure, the PSg using the corrected history can improve on the original PSg(algo). For the IBS benchmarks, it is better than the PAg scheme when a short history is used. For a longer history, however, adaptivity becomes useful and the PAg scheme performs better.

suite, while global history schemes tend to perform better for integer programs of SPEC89 when large hardware budgets are available. They explained that per-address history is good at predicting loop branches which occur frequently in floating-point benchmarks, while global history is effective at predicting if-then-else branches that appear more frequently in integer benchmarks.

However, using per-address history information is not free; it can be very expensive sometimes. As we found before, interference in the first-level table of the per-address history scheme is significant, so it is usually desirable to have more history registers. Per-address history becomes expensive when a large BHT is employed, because the cost of a per-address history bit is equivalent to the number of entries in the branch history table. In this section, we present the cost-effective analysis results to complete our study.

6.4.1 Cost-effective comparison for the per-address history schemes

In this experiment, the cost includes the second-level table as well as the BHT. The cost calculation is described as follows,

$$\text{Cost}_{\text{in bits}} = \text{history\_length} \times \text{\#\_of\_entry\_in\_BHT} + 2^{\text{history\_length}} \times \text{\#\_of\_PHTs} \times 2$$

The storage for the address tags is ignored because the BHT is assumed to share tags with the branch target buffer.

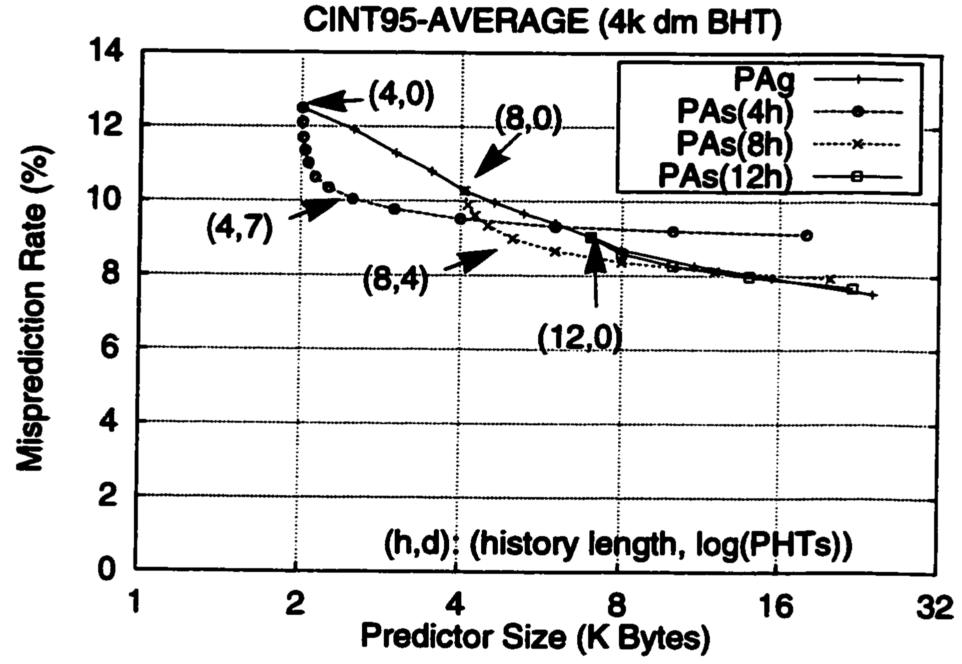
In this experiment, all schemes have a 4K-entry, directed mapped BHT. This is different from the work by Yeh and Patt [59], where a 4-way set-associative BHT was used. We assume that, as the CPU cycle time becomes very short, the associative table is too expensive to implement.

Figure 6.24 provides the cost-effective comparison for the per-address history schemes for both SPEC CINT95 and IBS-Ultrix suites. From the results we can see that when the budget is large, the PAg scheme is among the best schemes. However, as the budget is small, the PAs schemes that have multiple columns in the second-level table, or PHTs, are the best. It would seem that the per-address history bit is more useful than the address bit at making accurate predictions, but the per-address history bit is too expensive to be used in small systems. The extra cost incurred by per-address history bits offsets the benefit.

6.4.2 Cost-effective comparison between the per-address schemes and global history schemes

To set our studies in a broader context, we compare the per-address history scheme with the global history scheme.

Figure 6.25 presents the comparison for the SPEC CINT95 and IBS-Ultrix benchmarks. The per-address history schemes examined in this experiment are the same as the ones in Figure 6.24, except that the PAs(12h), the PAs scheme using 12-bit per-address history, is excluded, because it is not effective. A gshare scheme is used for comparison. As shown in the figure, the per-address history scheme is less accurate than the global history scheme for the integer benchmarks for the range of budgets examined. This result is similar to the findings by Yeh and Patt [59], except that this result shows that even for small budgets the per-address scheme is not able to outperform the global history scheme.



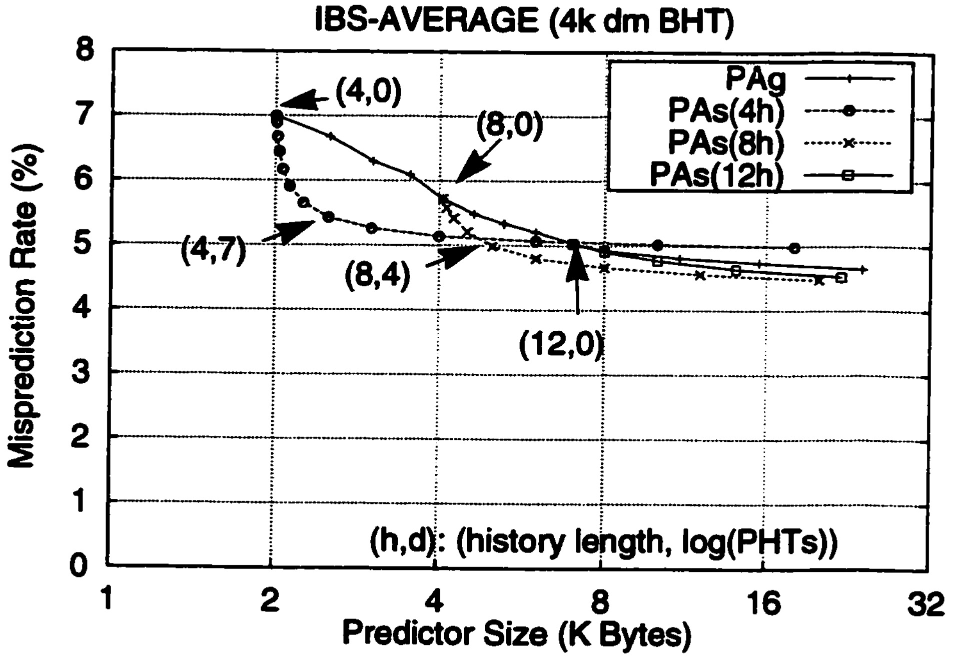
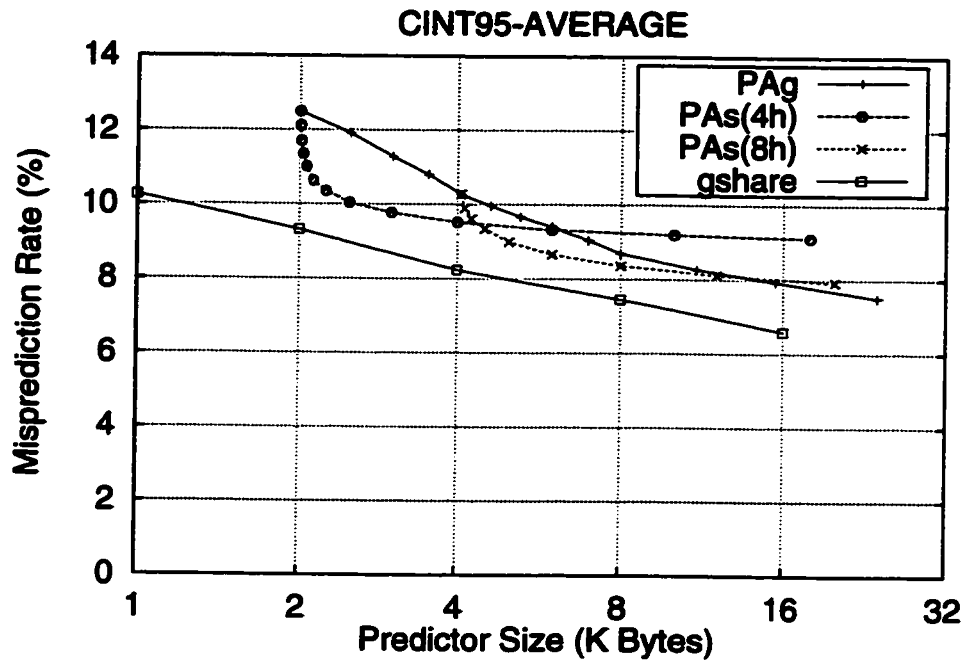


Figure 6.24: Cost-effective comparison for PAs schemes, including BHT cost — SPEC CINT95 (top) and IBS-Ultrix (bottom)

In this figure, four variations of the PAs schemes are examined. All schemes have a 4K-entry, direct-mapped branch history table. The PAg scheme has always one pattern history table (PHT). The PAs(4h) represent the schemes that use 4-bit per-address history but have various numbers of PHTs. Similarly, PAs(8h) has 8-bit per-address history and PAs(12h) has 12-bit per-address history.

As shown in the figure, for large budgets, PAg is among the best schemes. However, for small budgets, because the history bit is more expensive than the address bit, the PAs with multiple PHTs is the most cost-effective scheme.



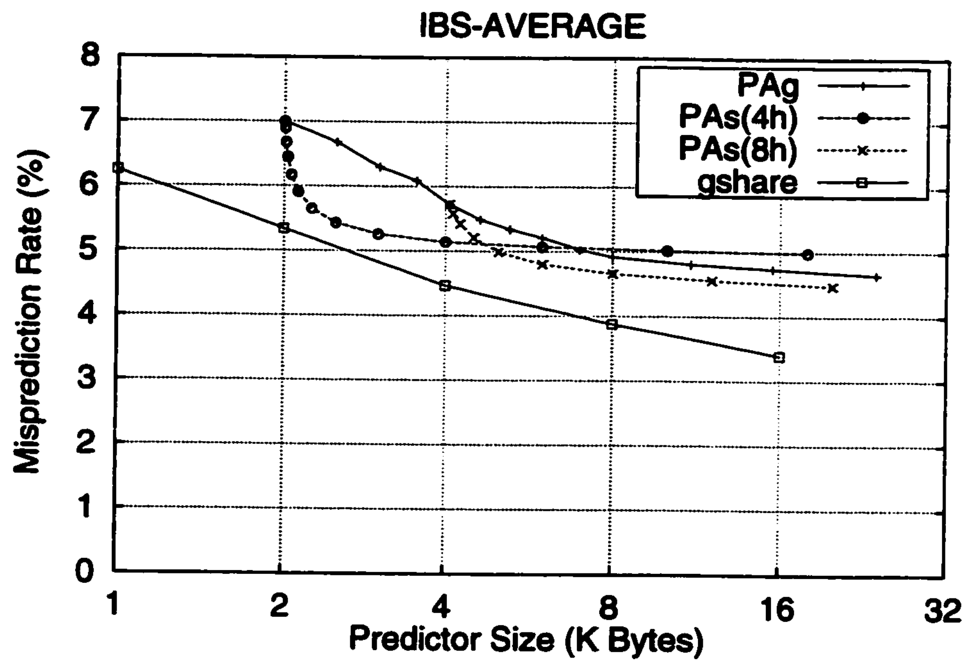


Figure 6.25: Cost-effective comparison for gshare and the PAs scheme, including BHT cost — SPEC CINT95 (top) and IBS-Ultrix (bottom)

In this figure, three per-address history schemes are compared against the best of the gshare schemes. All of per-address schemes have a 4K-entry, direct-mapped branch history table. The PAg scheme has one pattern history table (PHT). The PAs(4h) represents the schemes that use 4-bit per-address history but have various numbers of PHTs. Similarly, PAs(8h) has 8-bit per-address history.

As shown in the figure, gshare consistently outperforms the per-address scheme for the budgets examined, suggesting that global history is more effective for predicting integer benchmarks.

6.5 Summary

In this section, we have presented our performance studies for dynamic branch prediction schemes that employ per-address history information. We began by reviewing the per-address two-level dynamic branch predictors. We have shown and analyzed the overall performance trends for the per-address schemes, and also examined the impacts of large footprint programs on the schemes.

We have shown that, the dominant factor for the performance of per-address schemes is the interference in the first-level table, or the BHT. Increasing the size of BHT is the most straightforward way to reduce the interference. In contrast, the second-level table does not have a severe interference problem.

We then examined closely the prediction accuracy of the per-address scheme for each per-address history pattern. We have shown that the per-address scheme performs very well for the 0000_2 and 1111_2 patterns, the strongly-biased patterns, but poorly for the other mixed patterns. Fortunately, in the per-address scheme, the strongly-biased patterns, occur more than 80% of the time, and therefore the per-address scheme still delivers accurate prediction.

An inexpensive PSg scheme employing an algorithm-derived static-trained second-level table was then proposed as an alternative to the PAg scheme. This PSg scheme performs well because it predicts accurately for the large number of strongly-biased branches that occur in realistic workloads. It is an attractive alternative because it performs closely to the PAg scheme but requires less hardware, less on-chip area, and results in a smaller access time.

However, the PSg scheme heavily relies on the correct per-address history information to make good prediction. To ensure the PSg scheme can always obtain the correct per-address history, a modified BHT was proposed which encodes correct histories shorter than *n* bits to *n*-bit values. This modified BHT has been shown effective and can improve the PSg schemes slightly.

Finally, the cost-effective analysis for the per-address scheme was examined by including the cost of the first-level table. From the analysis, we found that the PAg scheme is more cost-effective when large hardware budgets are available, and the PAs scheme should be used instead for smaller budgets. This is a consequence of address bits being cheaper than the per-address history bits. We also showed that the global history scheme performs better than the per-address history scheme for integer programs over the range of budgets examined. The cost-effectiveness of per-address schemes is limited because of its requirement for a large BHT.

CHAPTER 7

Conclusions

This dissertation is concerned with the optimal designs of high performance dynamic branch predictors that exploit branch outcome history. Branch outcomes are usually the result of random activities; most of time they are correlated with the past behavior of neighboring branches. By keeping track of the history of branch outcomes it is possible to anticipate with a high degree of certainty which direction branches will take. However, exploiting branch outcome history is not free, but requires significant hardware resources to memorize complicated branch states related to history patterns. It is thus important to understand the advantages and disadvantages of employing history when designing a branch predictor, or unexpected performance may result. It is especially important to evaluate predictors by using benchmarks that reflect the environments the predictors are designed for. In this dissertation, a diverse set of programs were employed in our studies on predictor performance. As a result we showed that many benchmarks popularly used in previous branch prediction studies were inadequate and lead to misleading conclusions.

This dissertation explored and examined various prediction schemes using global and per-address history. For the global history schemes, this dissertation first provided a general performance pictures to illustrate the performance trends. It also indicated design points for the predictors under different sizes of programs. For small programs, using long histories is the best strategy, while for large programs, such as systems code, shorter histories with address indexed multi-column second level tables performed better.

Global history was shown to have the potential to achieve very accurate prediction accuracy because it factored out large numbers of highly biased dynamic branch streams which are easily predicted. However, the conventional two-level schemes that rely on global history often cause the intermingling of highly but oppositely biased streams. This becomes the major source of the interference in predictors. The issue of interference was thoroughly investigated. We showed that employing branch address bits can help reduce interference significantly but at the same time their use reduces the bias in the resulting branch stream outcomes. We thus showed that the requirement of generating highly biased streams that do not contain a significant fraction of oppositely biased substreams is impossible with the conventional global history scheme. Some schemes that use some address bits such as the GAs schemes, can, however, obtain limited improvements.

This dissertation then proposed a new global history scheme, the bi-mode scheme, to address this problem. By adding a two-bit counter table indexed by the branch address in the first level, the bi-mode scheme can successfully separate highly but oppositely biased dynamic branch streams for the second-level table, thus realizing the advantages of global history without causing interference The bi-mode scheme was shown to outperform the global history scheme and to be more cost-effective. The bi-mode scheme was then compared with other recently proposed schemes that also attempt to reduce interference. These including the gshare scheme, the skewed predictor, the agree predictor, and the filtering scheme. Our experimental results show the bi-mode scheme to be the best for larger predictors (more bytes of memory) and to be better than most for small predictors.

Furthermore it is simple and fast—it does not lengthen critical paths. A variant of the bi-mode predictor, the history+ scheme, that uses extra older global history bits was also proposed and evaluated. This variant can exploit more history information and is to achieve the highest prediction accuracy for some of the benchmarks examined. In particular, it increased the accuracy of prediction on the worst case benchmarks. The history+ scheme is a candidate for future research work.

This dissertation also examined per-address history schemes. General performance pictures are provided to illustrate the trends and bottlenecks for this class of predictors. The results show that there is no severe interference problem in the second table, but interference in the first level table is a major design concern. The first-level table must be able to record per-address history for branches separately. We showed that, provided interference in the first-level table is controlled, the performance gain for per-address history schemes is mostly a result of accurate prediction for strongly taken and strongly not taken branches.

A cost-effective per-address scheme that uses a fixed algorithm-derived second-level table, PSg(algo), was then proposed and evaluated. Because this PSg(algo) predicts very accurately for the strongly taken and strongly not taken branches its performance was close to the dynamic per-address history schemes. However, the PSg(algo) was even more sensitive to interference in the first-level table than the dynamic schemes, because the algorithm employed in PSg(algo) assumes that correct per-address history is recorded in the first-level table. To resolve this problem, we modified the first level table so that it can always provide correct per-address history regardless of interference. This modification was shown to yield modest improvements to the PSg(algo). Compared to PAg predictors, the PSg(algo) schemes were most cost-effective for smaller sized predictors

Finally, we also compared the global history and per-address history schemes. Comparing same-sized systems, we found that, for the benchmarks examined, global history schemes are superior. The per-address history scheme is worse because increasing per-address history bits is a much more expensive way to improve prediction than by increasing global history bits.

Accurate branches prediction is crucial for wide issue processors if they are to perform up to their potential. Dynamic branch prediction schemes provide the most straightforward and effective solution, but require very accurate dynamic branch predictors. This dissertation has contributed a detailed examination and fuller understanding about the mechanisms underlying dynamic branch predictors. It also proposed ways to further improve prediction accuracy. However, the branch problem is not yet totally resolved. For example, making multiple branch predictions in a single clock cycle becomes an important issue as the technology allows more than one basic block to be fetched at a time. A recent study on caches that attempts to increase fetch bandwidth by exploiting the locality of dynamic instruction streams has attracted much attention [43]. This study has also suggested that very accurate multiple branch prediction per cycle is important. However, there have been no significant research reported on this topic, and the optimal design is still unknown.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] Ball, T. and Larus, J. Branch Prediction For Free. *Proc. of ACM SIGPLAN*, June 1993, pp. 300-313.

[2] Bell, T.C., Cleary, J.G., and Witten, I.H. *Text compression*, Englewood Cliffs, NJ: Prentice-Hall, 1990.

[3] Bondi, J., Nanda, A., and Dutta, S. Integrating a Misprediction Recovery Cache into a Superscalar Pipeline. *Proc. of the 29th Ann. Int. Symp. on Microarchitecture*, Dec. 1996, pp. 14-23.

[4] Brown, J., Persels, S., and Meyer, J. *Branch Prediction Unit for High-Performance Processor*, US patent 5,394,529, Feb. 28, 1995. [First filed in 1990.]

[5] Butler, M., Yeh, T-Y., Patt, Y., Alsup, M., Scales, H., and Shebanow, M. Single Instruction Stream Parallelism Is Greater than Two. *Proc. of the 18st Ann. Int. Symp. on Computer Architecture*, May 1991, pp. 276-286.

[6] Calder, B., and Grunwald, D. Fast & Accurate Instruction Fetch and Branch Prediction. *Proc. of the 21st Ann. Int. Symp. on Computer Architecture*, Apr. 1994, pp. 2-11.

[7] Calder, B., Grunwald, D., and Emer, J. A System Level Perspective on Branch Architecture Performance. *Proc. of the 28th Ann. Int. Symp. on Microarchitecture*, Nov. 1995, pp. 199-206.

[8] Chang P-P., Mahlke, S., Chen, W., Warter, N., and Hwu, W-M. IMPACT: An Architectural Framework for Multiple-Instruction-Issue Processors. *Proc. of the 18th Ann. Int. Symp. on Computer Architecture*, May 1991, pp. 266-275.

[9] Chang, P-Y., Hao, E., Yeh, T., and Patt, Y. Branch Classification: a New Mechanism for Improving Branch Predictor Performance. *Proc. of the 27th Ann. Int. Symp. on Microarchitecture*, Nov. 1994, pp. 22-31.

[10] Chang, P-Y., Hao, E., and Patt, Y. Alternative Implementations of Hybrid Branch Predictors. *Proc. of the 28th Ann. Int. Symp. on Microarchitecture*, Nov. 1995, pp. 252-257.

[11] Chang, P-Y., Evers, M., and Patt, Y. Improving Branch Prediction Accuracy by Reducing Pattern History Table Interference. *Proc. of the 4th Int. Conference on Parallel Architectures and Compilation Techniques*, 1996, pp. 48-57.

[12] Chang, P-Y., Hao, E., and Patt, Y. Target Prediction for Indirect Jumps. *Proc. of the 24th Ann. Int. Symp. on Computer Architecture*, May 1997, pp. 274-291.

[13] Chang, P-Y. Classification-Directed Branch Predictor Design. *Ph.D Dissertation*, The University of Michigan, Ann Arbor, 1997.

[14] Chen, I-C. Enhancing the Instruction Fetching Mechanism Using Data Compression. *Ph.D Dissertation*, The University of Michigan, Ann Arbor, 1997.

[15] Chen, I-C., Coffey, J. and Mudge, T. Analysis of branch prediction via data compression. *Proc. of the 7th Int. Conference on Architectural Support for Programming Languages and Operating Systems,* Oct. 1996, pp. 128-137.

[16] Dijkstra, E.W. Guarded Commands, Nondeterminacy, and Formal Derivation of Programs. *Communication of ACM,* vol. 18, Aug. 1975, pp. 453-457.

[17] Eustace, A. and Srivastava, A. ATOM: A flexible interface for building high performance program analysis tools. *Proc. of the Winter 1995 USENIX Technical Conference on UNIX and Advanced Computing Systems,* Jan. 1995.

[18] Fisher, J.A. Trace Scheduling: A Technique for Global Microcode Compaction. *IEEE Transactions on Computers,* Vol. C-30, No. 6, June 1981, pp. 478-490.

[19] Fisher, J.A., and Freudenberger, S.M. Predicting Conditional Branch Directions From Previous Runs of a Program. *Proc. 5th Int. Conf. on Architectural Support for Programming Languages and Operating Systems,* Oct. 1992, pp. 85-95.

[20] Flynn, M. Some Computer Organizations and Their Effectiveness. *IEEE Transactions on Computers,* Vol. c-21, No. 9, Sep. 1972, pp. 948-960.

[21] Gloy, N., Young, C., Chen, B., and Smith, M. An Analysis of Dynamic Branch Prediction Schemes on System Workloads. *Proc. of the 23rd Ann. Int. Symp. on Computer Architecture,* May 1996, pp. 12-21.

[22] Gwennap, L. Intel's P6 Uses Decoupled Superscalar Design. *Microprocessor Report,* vol. 9, No. 2, Feb. 16, 1995.

[23] Hao, E., Chang, P-Y., and Patt, Y. The Effect of Speculatively Updating Branch History on Branch Prediction Accuracy, Revisited. *Proc. of the 27th Ann. Int. Symp. on Microarchitecture,* Nov. 1994, pp. 228-232.

[24] Hennessy, J. L. and Patterson, D. A. *Computer architecture: a quantitative approach, 2nd ed.,* San Francisco, CA: Morgan Kaufmann Publishers Inc. 1996.

[25] Hsu, P. and Davidson, E. Highly Concurrent Scalar Processing. *Proc. of the 13th Ann. Int. Symp. on Computer Architecture,* June 1986, pp. 386-395.

[26] Jourdan, S., Hsing, T-H., Stark, J., Patt, Y. The Effects of Mispredicted-Path Execution on Branch Prediction Structures. *Proc. of the 4th Int. Conference on Parallel Architectures and Compilation Techniques,* 1996, pp. 58-67.

[27] Kaeli, D. and Emma, P. G. Branch history table prediction of moving target branches due to subroutine returns. *Proc. of the 18th Ann. Int. Symp. on Computer Architecture,* May 1991, pp. 34-41.

[28] Lam, M. and Wilson, R. Limits of Control Flow on Parallelism. *Proc. of the 19th Ann. Int. Symp. on Computer Architecture,* May 1992, pp. 46-57.

[29] Lee, J.K.F. and Smith, A.J. Branch Prediction Strategies and Branch Target Buffer Design. *IEEE Computer*, 21(7), Jan. 1984, pp. 6-22.

[30] Lesartre, G. and Hunt, D. PA-8500: The Continuing Evolution of the PA-8000 Family., Hewlett Packard Co., *technical white paper*, http://www.hp.com/computing/framed/technology/micropro/pa-8500/docs/8500.html, also presented in Compcon 1997.

[31] Lick, K.L. Hybrid Branch Prediction Using Limited Dual Path Execution. *MS. Thesis*, University of California, Riverside, Dec. 1996.

[32] Mahlke, S., Hand, R., Bringmann, R., Gyllenhaal, J., Gallagher, D., Hwu, W-M. Characterizing the Impact of Predicated Execution on Branch Prediction. *Proc. of the 27th Ann. Int. Symp. on Microarchitecture*, Nov. 1994, pp. 217-227.

[33] McFarling, S, and Hennessy, J. Reducing the Cost of Branches. *Proc. of the 13th Ann. Int. Symp. on Computer Architecture*, June 1986, pp. 396-403.

[34] McFarling, S. Combining Branch Predictors. *WRL Technical Note TN-36*, June 1993.

[35] Michaud, P., Seznec, A., and Uhlig, R. Trading Conflict and Capacity Aliasing in Conditional Branch Predictors. *Proc. of the 24th Ann. Int. Symp. on Computer Architecture*, May 1997, pp. 292-303.

[36] Nair, R. Optimal 2-Bit Branch Predictors. *IEEE Transactions on Computers, Vol. 44, No. 5, 1995*, pp. *698-702*.

[37] Nair, R. Dynamic Path-Based Branch Correlation. *Proc. of the 28th Ann. Int. Symp. on Microarchitecture*, Nov. 1995, pp. 15-23.

[38] Pan, S.T., So, K., and Rahmeh, J.T. Improving the Accuracy of Dynamic Branch Predication Using Branch Correlation. *Proc. 5th Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, Oct. 1992, pp. 76-84.

[39] Perl, S. and Sites, R. Studies of Windows NT performance using dynamic execution traces. *Proc. of the USENIX 2nd Symp. on Operating Systems Design and Implementation*, Oct. 1996.

[40] Pnevmatikatos, D.N. and Sohi, G.S. Guarded Execution and Branch Prediction in Dynamic ILP Processors. *Proc. of the 21st Ann. Int. Symp. on Computer Architecture*, Apr. 1994, pp. 120-129.

[41] Rau, B.R., Yen, D., Yen, W., and Towle, R. The Cydra 5 Departmental Supercomputer—Design Philosophies, Decisions, and Trade-offs. *IEEE Computer*, Jan. 1989, pp. 12-35.

[42] Riseman, E., Foster, C. The Inhibition of Potential Parallelism by Conditional Jumps. *IEEE Transactions on Computers*, Vol. c-21, No. 12. Dec. 1972, pp. 1405-1411.

[43] Rotenberg, E., Bennett, S. and Smith, J. Trace Cache: a Low Latency Approach to High Bandwidth Instruction Fetching. *Proc. of the 29th Ann. Int. Symp. on Microarchitecture,* Dec. 1996, pp. 24-34.

[44] Sechrest, S, Lee, C-C, and Mudge, T. The Role of Adaptivity in Two-Level Adaptive Branch Prediction. *Proc. of the 28th Ann. Int. Symp. on Microarchitecture,* Nov. 1995, pp. 264-270.

[45] Sechrest, S, Lee, C-C, and Mudge, T. Correlation and Aliasing in Dynamic Branch Predictors: Full Simulation Results. *Tech. Report CSE-TR-283-96,* Univ. of Michigan, Ann Arbor, MI, Feb. 1996.

[46] Smith, J.E. A Study of Branch Prediction Strategies. *Proc. of the 8th Ann. Int. Symp. on Computer Architecture,* May 1981, pp. 135-148.

[47] SPEC CPU'95, *Technical Manual,* August 1995.

[48] Sprangle, E., Chappell R., Alsup, M., and Patt, Y. The Agree Predictor: A Mechanism for Reducing Negative Branch History Interference. *Proc. of the 24th Ann. Int. Symp. on Computer Architecture,* May 1997, pp. 284-291.

[49] Talcott, A.R., Nemirovsky, M., and Wood, R.C. The Influence of Branch Prediction Table Interference on Branch Prediction Scheme Performance. *Proc. of the 3rd Int. Conference on Parallel Architectures and Compilation Techniques,* June 1995, pp. 89-98.

[50] Tyson, G. The Effects of Predicated Execution on Branch Prediction. *Proc. of the 27th Ann. Int. Symp. on Microarchitecture,* Nov. 1994, pp. 196-206.

[51] Tjaden, G. and Flynn, M. Detection and parallel execution of independent instructions. *IEEE Transactions on Computers,* Vol. c-19, No. 10, Oct. 1970, pp. 889-895.

[52] Uhlig, R., Nagle, D, Mudge, T., Sechrest, S., and Emer, J. Instruction Fetching: Coping with Code Bloat. *Proc. of the 22th Ann. Int. Symp. on Computer Architecture,* Italy, June 1995, pp. 345-356.

[53] Uht, A. A Theory of Reduced and Minimal Procedural Dependencies. *IEEE Transactions on Computers,* Vol. 40, No. 6, June 1991, pp. 681-692.

[54] Uht, A. and Sindagi, V. Disjoint Eager Execution: An Optimal Form of Speculative Execution. *Proc. of the 28th Ann. Int. Symp. on Microarchitecture,* Dec. 1995, pp. 313-325.

[55] Uht, A., Sindagi, V., and Somanathan, S. Branch Effect Reduction Techniques. *IEEE Computer,* May 1997, pp. 71-81.

[56] Yeh, T-Y. and Patt, Y. Two-Level Adaptive Training Branch Prediction. *Proc. of the 24th Ann. Int. Symp. on Microarchitecture,* Nov. 1991, pp. 51-61.

[57] Yeh, T-Y. and Patt, Y. Alternative Implementations of two-level adaptive branch predictions. *Proc. of the 19th Ann. Int. Symp. on Computer Architecture,* May 1992, pp. 124-134.

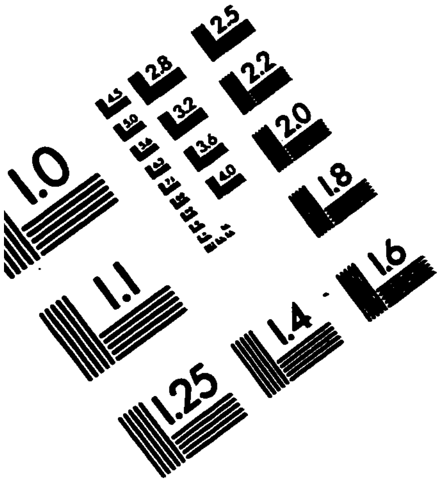
[58] Yeh, T-Y. and Patt, Y. A Comprehensive Instruction Fetch Mechanism for a Processor Supporting Speculative Execution. *Proc. of the 25th Ann. Int. Symp. on Microarchitecture,* Dec. 1992, pp. 129-139.

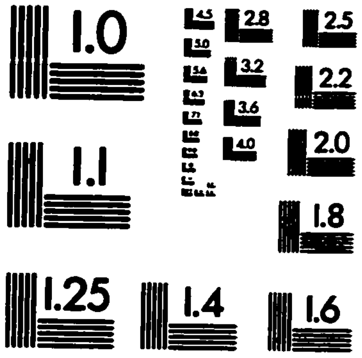
[59] Yeh, T-Y. and Patt, Y. A Comparison of Dynamic Branch Predictors that use Two Levels of Branch History. *Proc. of the 20th Ann. Int. Symp. on Computer Architecture,* May 1993, pp. 257-266.

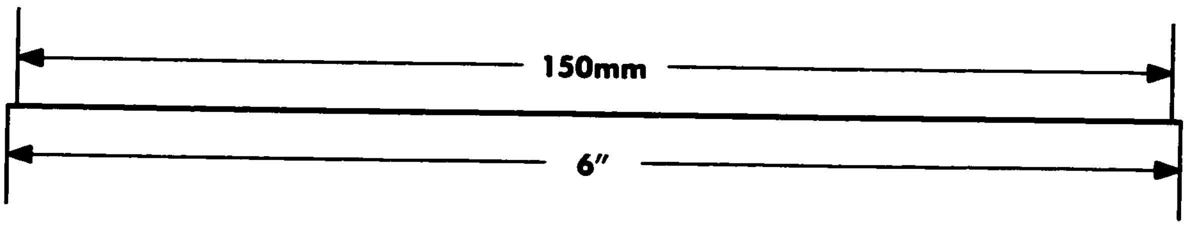
[60] Young, C. and Smith, M. Improving the accuracy of static branch prediction using branch correlation" *Proc. of the 6th Int. Conference on Architectural Support for Programming Languages and Operating Systems,* Oct. 1994, pp. 232-241.

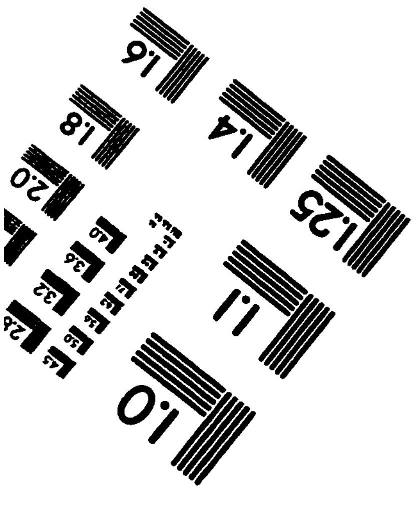
[61] Young, C., Gloy, N. and Smith, M. A comparative analysis of schemes for correlated branch prediction. *Proc. of the 22nd Ann. Int. Symp. on Computer Architecture,* June 1995, pp. 276-286

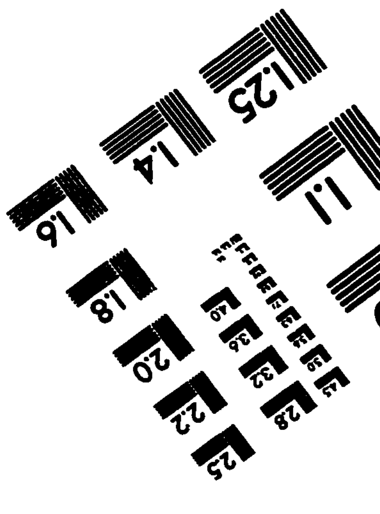
IMAGE EVALUATION
TEST TARGET (QA-3)











APPLIED IMAGE . Inc
1653 East Main Street
Rochester, NY 14609 USA
Phone: 716/482-0300
Fax: 716/288-5989