Classification-Directed Branch Predictor Design

by

Po-Yung Chang

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
1997

Doctoral Committee:
Professor Yale N. Patt, Chair
Professor Edward S. Davidson
Professor Ronald J. Lomax
Professor Trevor N. Mudge
Tse-Yu Yeh, CPU Micro-Architect and Senior Design Engineer, Intel

To my family for their love and support.

ACKNOWLEDGEMENTS

I would like to thank my advisor, Professor Yale Patt, for his guidance. My stay at the University of Michigan working for Professor Patt has been very enjoyable as well as educational.

I would like to thank Professor Edward Davidson, Professor Trevor Mudge, Professor Ronald Lomax, and Tse-Yu Yeh for taking time out of their busy schedule to serve on my doctoral committee and for providing many valuable suggestions on my work.

The support of our industrial partner, Intel, is greatly appreciated.

I am glad to have worked with the other members of the HPS research group: Michael Butler, Tse-Yu Yeh, Robert Hou, Greg Ganger, Bruce Worthington, Debbie Marr, Carlos Fuentes, Eric Hao, Chris Eberly, Jared Stark, Eric Sprangle, Lea-Hwang Lee, Tse-Hao Hsing, Sanjay Patel, Mark Evers, Dan Friendly, Paul Racunas, Rob Chapell, and Peter Kim. Michael Butler and Tse-Yu Yeh have provided the group with useful simulation tools. Eric Hao, Marius Evers, Tse-Yu Yeh and I have collaborated on various projects. Eric Hao, Jared Stark, Carlos Fuentes, Mark Evers, Sanjay Patel, Dan Friendly, Eric Sprangle, Greg Ganger, Bruce Worthington, and Robert Hou have carefully proofread and made suggestions on various papers. I have also enjoyed the many activities that we do outside of work, including xtank, xpilot, tennis, basketball, and racquetball.

Finally, I would like to thank my family for their love and support. My parents have always provided the best environment for me to pursue my goals. My brothers have always been there to make sure that I enjoy life inside and outside of school.

TABLE OF CONTENTS

LIST OF TABLES

Table

LIST OF FIGURES

LIST OF APPENDICES

APPENDIX

ABSTRACT

Classification-Directed Branch Predictor Design

by
Po-Yung Chang

Chair: Yale N. Patt

Pipeline stalls due to branches represent one of the most significant impediments to realizing the performance potential of deeply pipelined superscalar processors. Two well-known mechanisms have been proposed to reduce the branch penalty, speculative execution in conjunction with branch prediction and predicated execution.

This dissertation proposes branch classification, coupled with improvements in conditional branch prediction, indirect branch prediction, and predicted execution, to reduce the branch execution penalty.

Branch classification allows an individual branch instruction to be associated with the branch predictor best suited to predict its direction. Using this approach, a hybrid branch predictor is constructed which achieves a higher prediction accuracy than any branch predictor previously reported in the literature.

This dissertation also proposes a new prediction mechanism for predicting indirect jump targets. For the perl and gcc benchmarks, this mechanism reduces the indirect jump misprediction rate by 93.4% and 63.3% and the overall execution time on an 8-wide issue out-of-order execution machine by 14% and 5%.

Finally, this dissertation proposes a new method for combining the performance benefits of predicated execution and speculative execution. This approach significantly reduces the branch execution penalty suffered by wide-issue processors.

CHAPTER 1

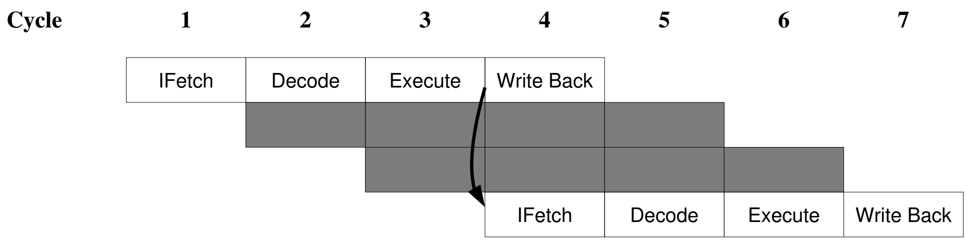
INTRODUCTION

1.1 The Branch Problem

Branches are instructions that change the flow of control and cause breaks in the normal sequence of execution. Whenever a branch is encountered, the instruction fetch has to stall until the target and the direction of the branch become ready. Thus, branches can significantly reduce the performance of pipelined processors by interrupting the steady supply of instructions to the instruction pipeline.

To illustrate the branch problem in pipelined processors, Figure 1.1 shows the pipeline of a single-issue processor with four pipeline stages – fetch, decode, execute and write back. If a branch is not resolved until the end of its execution stage, two clock cycles are wasted for every branch encountered in the instruction stream. Assuming that the processor only stalls due to branches and the average basic block size is four instructions, the processor stalls for two cycles out of every six cycles. Thus, one third of the execution bandwidth is wasted.

With improvements in process technology and increases in transistor budgets, today's processors are being built with wider issue rates and deeper pipelines in order



Shaded blocks indicate pipeline bubbles caused by the branch.

Figure 1.1: Pipeline bubble: a single-issue processor with four pipeline stages

to exploit larger amounts of instruction-level parallelism (as shown in Table 1.1). The amount of bandwidth wasted due to a branch is significantly greater for a wide-issue deeply-pipelined processor than for a single-issue processor, as shown in Figures 1.1 and 1.2. With a four wide-issue processor, an entire basic block can be fetched and issued in one cycle. If a branch takes three cycles to be resolved after it is fetched, the processor stalls for three out of every four cycles, wasting 75 percent of its execution bandwidth. Therefore, branch execution is a critical issue in processor design.

Processor	issue-width	integer pipeline depth
Intel Pentium Pro	3 (6†)	12
HP/PA8000	4	7
PowerPC 620	4	5
Alpha 21164	4	7
IBM Power2	6	5
MIPS R10000	4	5

Table 1.1: Processor issue-width and pipeline depth

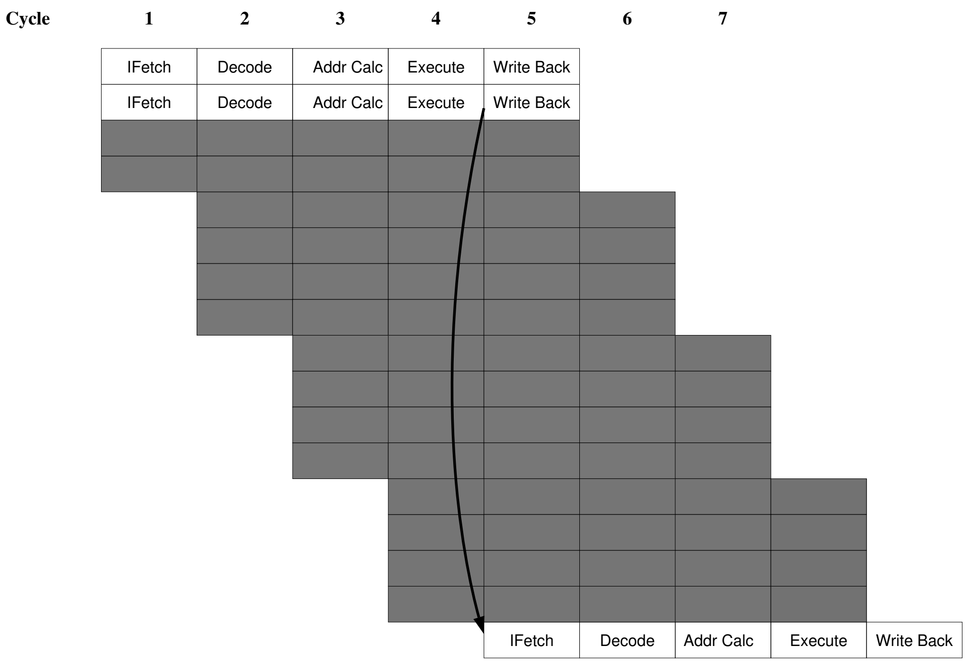


Figure 1.2: Pipeline bubble: a 4-wide issue processor with five pipeline stages

†The three x86 instructions can be translated into six Micro-ops; the micro-ops use a load/store model.

Two well-known mechanisms have been proposed to reduce the branch penalty, speculative execution in conjunction with branch prediction and predicated execution.

Speculative execution [27, 23] is a microarchitectural mechanism that solves the branch problem by guessing the target of a branch. After making a prediction for a branch in the dynamic instruction stream, the processor speculatively executes the instructions from the predicted target. The branch prediction is confirmed when the branch instruction is executed. If the prediction is correct, then the processor suffers no performance penalty for this dynamic instance of the branch. If the prediction is incorrect, the processor, after removing from its state the effects of the speculatively executed instructions, must return to the point of the branch prediction and begin executing instructions from the correct path. In this case, the full branch execution penalty is suffered. Thus, an excellent branch predictor is vital to deliver the potential performance of a wide-issue deeply-pipelined microarchitecture. This dissertation proposes several mechanisms for increasing the accuracy of branch predictors.

Predicated execution [14, 29, 9, 19, 28, 36, 18, 12] is an architectural mechanism that addresses the branch problem by providing the compiler with a set of predicated instructions that can be used to eliminate static branches in the program. The branches can be eliminated by replacing their control-dependent instructions with flow-dependent predicated instructions [2]. A predicated instruction contains an extra source operand known as the predicate operand. The predicated instruction is conditionally executed based on the value of this operand. If the predicate evaluates to *true*, the predicated instruction is executed like a normal instruction. If the predicate evaluates to *false*, the instruction is not executed. Given these semantics, a compiler can replace the branch and the set of instructions that represent an *if-then-else* statement by predicating the *then* clause with the branch condition as the predicate and predicating the *else* clause with the complement of the branch condition as the predicate. By eliminating the branch, predicated execution ensures that the processor never suffers any branch execution penalties due to that branch. While both speculative and predicated execution have been shown to significantly reduce the performance penalty due to branches, both mechanisms have disadvantages as well as advantages. A processor using speculative execution suffers the full branch execution penalty whenever it makes a branch misprediction. A processor using predicated execution may see a drop in performance because predicated execution changes control dependencies into flow dependencies, lengthening the dependency chains in the program. In addition, using predicated execution wastes issue bandwidth because the predicated instructions from both branch paths must always be issued. This dissertation proposes a method for combining the performance benefits of speculative and predicated execution to reduce the branch execution penalty for wide-issue, dynamically scheduled machines.

1.2 Thesis Problem: Can branch classification reduce the branch execution penalty?

Thesis Statement: Branch classification, coupled with improvements in conditional branch prediction, indirect branch prediction, and predicated execution, can reduce the branch execution penalty.

Since branches have different dynamic behavior, the most suitable mechanism to handle one branch can be very different from that for another branch. Performance of processors can be improved if we can identify and invoke the most suitable mechanism for each branch.

This dissertation proposes branch classification as a means for combining the advantages of different branch handling mechanisms. Branch classification partitions the branches of a program into sets or branch classes. A good classification scheme partitions branches possessing similar dynamic behavior into the same branch class; thus, once the dynamic behavior of a class of branches is understood, the most suitable mechanism for this class can be identified. For example, the compiler can try to eliminate the hard-to-predict branches, leaving only the easy-to-predict branches to be handled by speculative execution.

In addition, this dissertation proposes branch classification as a means for improving the performance of each individual branch handling mechanism. Various branch handling mechanisms are considered for improvement.

- Hybrid branch predictors have been proposed as a way to achieve higher prediction accuracies [20, 7]. They combine multiple prediction schemes into a single predictor. A selection mechanism is used to decide, for each branch, which single-scheme predictor to use. For this attempt to result in significant increases in prediction accuracy, the hybrid predictor must combine an appropriate set of single-scheme predictors and use an effective predictor selection mechanism.

- Two-level branch predictors have been shown to achieve high prediction accuracy, yet they still suffer a significant number of mispredictions. Recent studies [34, 41] have shown that a number of these mispredictions are due to interference in the pattern history tables. The performance of two-level branch predictors can be improved if the amount of pattern history table interference can be reduced.

- Many existing branch prediction schemes are capable of accurately predicting the direction of conditional branches. However, these schemes are ineffective in predicting the targets of indirect jumps, achieving on average, a prediction accuracy rate of 52% for the SPECint95 benchmarks. Accurate predictions on indirect jumps are essential for high performance processors.

- Speculative execution can completely eliminate the penalty associated with a particular branch, but requires accurate branch prediction to be effective. Predicated execution does not require accurate branch prediction to eliminate the branch penalty, but is not applicable to all branches and usually increases the latencies within the program. Speculative execution and predicated execution must be carefully combined to minimize the effect of each approach's disadvantages.

1.3 Contributions

This dissertation makes five major contributions:

1. **Branch Classification**

   This dissertation demonstrates the benefits of branch classification. For example, branch classification allows an individual branch instruction to be associated with the branch predictor best suited to predict its direction. Using this approach, a hybrid branch predictor has been built which achieves a higher prediction accuracy than any branch predictor previously reported in the literature. With a fixed implementation cost of 32K bytes, this hybrid branch predictor achieves a prediction accuracy of 96.91% on gcc, a branch intensive benchmark, as compared to 96.47%for the best previously known predictor, reducing the miss rate by 12.5%.

2. **Hybrid Branch Predictors**

   This dissertation has evaluated the performance of various implementations of the hybrid branch predictor. Different combinations of single-scheme predictors and different selection mechanisms were considered. In addition, a new selection mechanism, the 2-level branch predictor selector, is proposed. By using more run-time information, it performs better than previously proposed selection mechanisms. For example, for a 16KByte Gshare/PAs hybrid branch predictor, we eliminated 5.5% of the mispredictions for the SPECint92 gcc benchmark.

3. **PHT Interference**

   This dissertation introduces a method for reducing the amount of pattern history table interference in two-level branch predictors by dynamically identifying some easily predictable branches and inhibiting the pattern history table update for these branches. We show that inhibiting the update in this manner reduces the amount of destructive interference in the global history variation of the two-level branch predictor, resulting in significantly improved branch prediction accuracy. For example, for a 2KByte gshare predictor, we eliminate 38% of the mispredictions for the SPECint95 gcc benchmark.

4. **Target Predictions**

   This dissertation proposes a new prediction mechanism for predicting indirect jump targets. This mechanism improves on the prediction accuracy achieved by BTB-based schemes. For the perl and gcc benchmarks, this mechanism reduces the indirect jump misprediction rate by 93.4% and 63.3% and the overall execution time by 14% and 5%.

5. **Predicated Execution**

   This dissertation proposes a new method for combining the performance benefit of both speculative execution and predicated execution to reduce the branch execution penalty for wide-issue, dynamically scheduled machines that use the two-level branch predictor. This approach significantly reduces the branch execution penalty suffered by wide-issue processors, reducing the execution times of the compress and eqntott benchmarks by 23% and 20% respectively.

1.4 Organization of This Dissertation

This dissertation is organized in nine chapters. Chapter 2 presents related work. Chapter 3 describes the simulation methodology, the benchmarks, and the machine model used to evaluate the proposed designs. Chapter 4 presents the branch classification concept. One method of branch classification is then proposed to demonstrate the performance benefit of branch classification. Chapter 5 examines hybrid branch predictors which combine the advantage of single-scheme predictors to achieve higher prediction accuracy. Chapter 6 introduces a method for reducing the pattern history table interference to improve the prediction accuracy of two-level branch predictors. Chapter 7 proposes target caches for accurately predicting indirect branches. Chapter 8 examines the performance benefit of using both speculative and predicated execution to reduce the branch execution penalty for wide-issue, dynamically scheduled machines that use the two-level branch predictor. Chapter 9 presents the conclusions for this study and suggests future directions for solving the branch problem.

CHAPTER 2

Related Work

2.1 Conditional Branch Predictors

Conditional branches conditionally redirect the instruction stream to their targets. A branch predictor must accurately predict the directions of these branches in order for a high-performance microprocessor to approach its performance potential. To improve prediction accuracy, various branch prediction strategies have been studied. These prediction schemes can be divided into two groups, static and dynamic predictors.

Static branch prediction schemes use information gathered before program execution, such as branch opcodes or profiles, to predict branch direction. The simplest of these predicts that all conditional branches are always taken as in Stanford MIPS-X [8], or always not-taken as in Motorola MC88000 [22]. Predicting all branches to be taken achieves about 66% accuracy whereas predicting not-taken achieves about 34% for the SPEC95 integer benchmarks. With additional hint bits in the branch opcodes, PowerPC 604 [10] allows the compiler to pass prediction information to the hardware. The program-based branch predictor [3] bases its prediction on the program structure; this predictor classifies branches into loop branches and non-loop branches using natural loop analysis [1] of the control flow graph. Loops are predicted to iterate rather than exit. Non-loop branches are predicted using a number of heuristics – opcode, loop, call, return, guard, store, and pointer comparisons. The opcode heuristic assumes that negative integers are used to denote error values; thus the heuristic predicts that bltz (branch-less-than-zero) and blez (branch-less-than-or-equal-to-zero) are not taken and that bgtz and bgez are taken. This heuristic also predicts floating point comparisons that check if two floating point numbers are equal to be false. The loop, call, return, guard, and store heuristics predict branches based on the properties of the basic block successors of these branches. The loop heuristic handles branches which choose between executing or avoiding a loop. This heuristic predicts that loops are executed rather than avoided because while and for loops are generated as an if-then branch around a do-until loop. The call heuristic handles branches that decide between executing or avoiding function calls. This heuristic predicts that function calls are avoided because many conditional calls are handling exceptional situations. The return heuristic handles branches which choose between executing or avoiding function returns. This heuristic predicts that function returns are avoided because many returns from procedures handle cases which occur infrequently. The guard heuristic handles branches where a branch on a value guards a later use of that value. For example, a null pointer test usually guards the use of the same pointer; if the pointer is not null, then the pointer value can be used. This heuristic predicts that the guarded code will be executed since the guards are usually used to catch exceptional conditions, e.g. null pointers. The store heuristic avoids successor blocks with store instructions. The pointer comparisons heuristic handles branches which compare a pointer to null or compare two pointers. Both comparison to a null pointer and equality comparison between two pointers are predicted to be false. The program-based branch predictor combines the aforementioned heuristics for predicting branches. If none of the heuristics applies to a branch, then a random prediction is used. Although the heuristics are simple and require little program analysis, this program-based predictor is on average a factor of two worse than the profile-guided predictor. The profile-guided branch predictor bases its prediction on the direction the branch most frequently takes, which is determined by profiling the program on a training input data set [11]. The profile-guided branch predictor achieves the highest prediction accuracy among the static predictors, correctly predicting about 87% of the branches.

Dynamic branch prediction algorithms use information gathered at run-time to predict branch directions. Smith [32] proposed a branch prediction scheme which uses a table of 2-bit saturating up-down counters to keep track of the direction a branch is more likely to take. Each branch is mapped via its address to a counter. The branch is predicted taken if the the most significant bit of the associated counter is set; otherwise, it is predicted not-taken. These counters are updated based on the branch outcomes. When a branch is taken, the 2-bit value of the associated counter is incremented by one; otherwise, the value is decremented by one. An 8KByte 2-bit counter scheme achieves a prediction accuracy of of 87%.

By keeping more history information, a higher level of branch prediction accuracy can be attained [38]. Yeh and Patt proposed the two-level branch predictor which uses two levels of history to make branch predictions (see Figure 2.1). The first level records the outcomes of the most recently executed branches and the second level keeps track of the more likely direction of a branch when a particular pattern is encountered in the first level history. The two-level branch predictor uses one or more k-bit shift registers, called branch history registers, to record the branch outcomes of the most recent k branches. In the Per-address scheme, each branch history register records the results of the last k occurrences of one static instance of a branch instruction. In the Per-set scheme, each branch history register records the last k outcomes of a particular set of branches. In the Global scheme, a global branch history register records the outcomes of the last k branches encountered in the dynamic instruction stream.

The two-level branch predictor uses one or more arrays of 2-bit saturating up-down counters, called Pattern History Tables, to keep track of the more-likely direction for branches. The lower bits of the branch address is used to select the appropriate

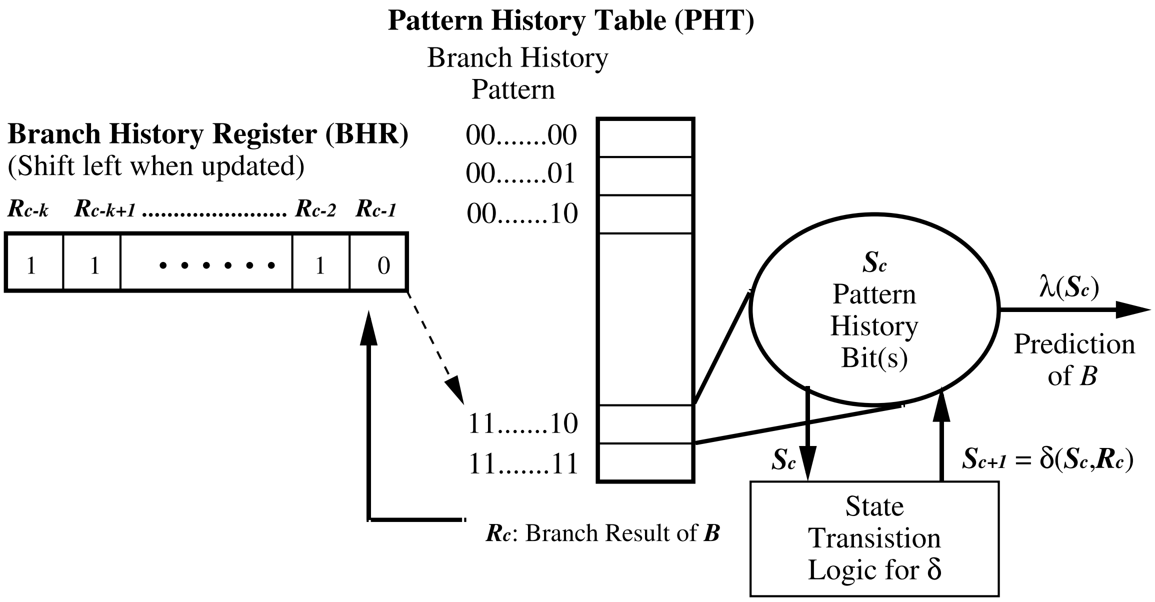


Figure 2.1: Structure of a two-level branch predictor

Pattern History Table(PHT) and the content of the branch history register (BHR) selects the appropriate 2-bit counter to use within that PHT. The two-level Branch Predictor can be further classified by the association of PHTs with branches. There can be one global PHT for all branches, one PHT for each set of branches, or one PHT for each static branch in the program. The two-level branch predictor using 13 global history bits achieves a prediction accuracy of approximately 91.8%. Using more history bits, higher prediction accuracy can be achieved; however, the implementation cost of the predictor also increases with more history bits. Using 15 global history bits, the two-level branch predictor achieves a prediction accuracy of 93.3%.

Several variations of the two-level branch predictor have been proposed [39]. McFarling [20] introduced gshare, a variation of the global-history two-level branch predictor which XORs the global branch history with the branch address to index into the PHT. Since the same global history patterns can occur for different branches during program execution, the global history pattern can be less efficient at identifying the current branch than the branch address itself. The gshare scheme tries to better identify the machine execution states by using both the branch address and the branch history. A gshare using 13 history bits achieves a predication accuracy of approximately 92%. Lee and Smith [17] proposed a scheme where the value of each Pattern History Table entry is determined statically, using profile information; this scheme is referred to as the PSg scheme by Yeh and Patt [40]. Sechrest et al. [31] introduced another method, PSg (algo), of statically determining the values in the PHT. The PSg (algo) works on the premise that for branches with a recurring pattern, the next outcome of the branch is likely an extension of this pattern. For example, if the branch history of a branch is `10101010`, a recurring pattern of `10` is detected and the outcome of this branch is predicted to be `1` (taken). Thus, the `10101010` PHT entry is set to `1`. The PSg (algo) also considers that some history patterns represent a transition between two cyclic patterns. To capture these patterns, the two oldest bits in the history pattern are ignored and a prediction is made if a recurring pattern is detected in the remaining pattern. For example, no recurring pattern is detected for the `11101010` pattern. However, if we ignored the two oldest history bits, the recurring pattern of `10` is detected for the `101010` pattern and the `11101010` PHT entry is set to `1`. This process is repeated, i.e. ignoring two oldest remaining bits during each iteration. For the remaining PHT entries, the entry is set to `1` if `1`'s outnumbers the `0`'s in the history pattern (the oldest bit is ignored if the number of bits in the history pattern is even). Otherwise, the entry is set to `0`. Since the contents of PHT are determined statically, the PSg scheme trades the benefits of having the ability to adapt for the benefits of having no PHT warm-up time. With a branch history length of 4, the performance of PSg(algo) is comparable to that of PAs. However, with a branch history length of 10, PAs outperforms PSg(algo). PAs using 13 history bits achieves a prediction accuracy of 92.5%.

2.2 Interference in Two-level Branch Predictors

The two-level branch predictors have been shown to achieve high prediction accuracy, yet they still suffer a significant number of mispredictions. Recent studies [34, 41] have shown that a number of these mispredictions are due to interference in the pattern history tables. Pattern history table interference occurs when a conditional branch references a PHT entry that was last referenced by another conditional branch. Talcott et al. [34] classified the interference as positive if the counter in the PHT entry correctly predicts the branch outcome. Otherwise, they classified the interference as negative. They showed that destructive interference causes branch prediction schemes to operate much below their potential performance level. Young et al. [41] classified interference by comparing the outcome of the PHT counter's prediction to the outcome of the prediction made by a predictor that have not PHT interference. This predictor uses an infinite number of PHTs and has no PHT interference because each static branch in the program has its own PHT. They classify interference as constructive if the counter correctly predicts the branch outcome and a predictor with an infinite number of PHTs mispredicts the outcome. They classified interference as destructive if the counter mispredicts while the predictor with an infinite number of PHTs predicts correctly. Otherwise, they classified the interference as neutral. Young et al. also showed that although branch interference can have a constructive, destructive, or neutral effect, destructive interference occurs much more frequently than constructive interference. Using 13 branch history bits, the gshare predictor achieves a 92% prediction accuracy whereas the gshare predictor with no PHT interference achieves a 95% prediction accuracy, leaving significant room for improvement.

2.3 Hybrid Branch Predictors

To further improve prediction accuracy, hybrid branch predictors have been proposed [20, 6]. A hybrid branch predictor is composed of two or more single-scheme predictors and a mechanism to select among these predictors. A hybrid branch predictor can exploit the different strengths of its single-scheme component predictors, enabling it to achieve a prediction accuracy greater than that achieved by any of its components alone. McFarling [20] proposed a selection mechanism that combines two branch predictors using an array of 2-bit up-down counters to keep track of which predictor is currently more accurate for each branch; each branch is mapped to a counter via its address (see Figure 2.2). The counter is incremented based on the rule shown in Table 2.1. The most significant bit of the counter determines which one of the two predictors to use. A hybrid branch predictor that combines PAs(13) and gshare(13) using 1K-entry 2bit predictor selection counters achieves a prediction accuracy of 93.9%

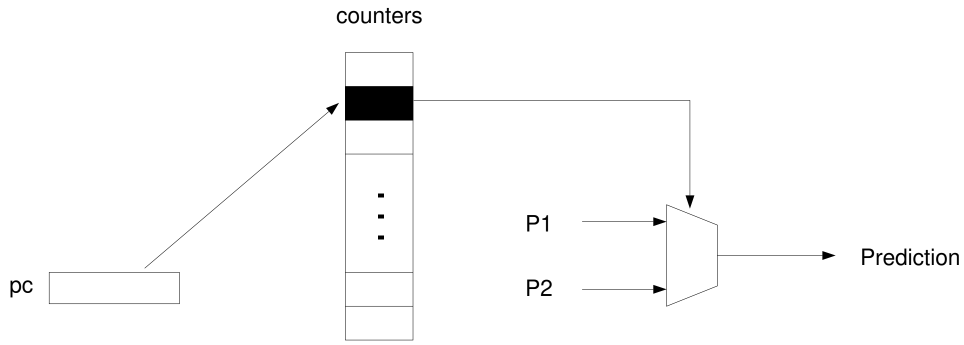


Figure 2.2: Structure of a McFarling's hybrid branch predictor

Predictor 1	Predictor 2	Update to Counter
Correct Prediction	Correct Prediction	No Change
Correct Prediction	Incorrect Prediction	Increment
Incorrect Prediction	Correct Prediction	Decrement
Incorrect Prediction	Incorrect Prediction	No Change

Table 2.1: Counter update rules

Chang and Banerjee[6] proposed the AVG predictor which can accurately predict loop branches. The AVG predictor keeps track of the average number of iterations executed for each loop. A branch is then predicted to exit the loop on the ith occurrence of that branch, where i is the average number of iterations associated with this loop. With hybrid branch predictors, this predictor can be used to handle loop branches.

2.4 Indirect Branch Predictors

An indirect branch has a dynamically specified target which may point to any number of locations in the program. In the past, branch prediction research has focused on accurately predicting direct branches [32, 21, 40, 20, 7, 24]; a direct branch has a statically specified target which points to a single location in the program. To predict such branches, the prediction mechanism predicts the branch direction and then generates the target associated with that direction. To keep track of the target addresses, a branch target buffer (BTB) is used. The BTB stores the last target seen for each of the two possible branch directions. However, BTB-based prediction schemes perform poorly for indirect branches. Because the target of an indirect branch can change with every dynamic instance of that branch, always using the target of the previous instance as the predicted target leads to poor prediction accuracy.

To address the problem of target prediction for indirect jumps in C++ programs, Calder and Grunwald proposed a new strategy, the 2-bit strategy, for updating BTB target addresses [5]. The typical strategy is to update the BTB on every indirect jump misprediction, Calder and Grunwald's 2-bit strategy does not update a BTB entry's target address until two consecutive predictions with that target address are incorrect. This strategy was shown to achieve a higher target prediction accuracy than that achieved by the typical strategy.

Kaeli et al. proposed a hardware mechanism, the case block table (CBT), to speed up the execution of SWITCH/CASE statements [16]. Figure 2.3 gives an example of a

```
High-Level
Construct              Assembly Code
----------             -------------

switch(v) {               r1 <- compare v, 1
 case 1:                  beq r1, L1
                           ; beq == branch if equal
   char = 'a';            r1 <- compare v, 2
   break;                 beq r1, L2
 case 2:                  r1 <- compare v, 3
   char = 'b';            beq r1, L3
   break;                 goto L4
 case 3:              L1: char <- 'a'
   char = 'c';            goto L5
   break;             L2: char <- 'b'
 default:                 goto L5
   char = 'd';        L3: char <- 'c'
}                         goto L5
                        L4: char <- 'd'
                        L5:
```

Figure 2.3: An Example of a SWITCH/CASE Construct

SWITCH/CASE statement and the corresponding assembly code for that statement. The assembly code consists of a series of conditional branches that determines which case of the SWITCH/CASE statement is to be executed. Because a single variable, the case block variable, specifies the case to be executed, this series of conditional branches can be avoided if the instruction stream could be directly redirected to the appropriate case. The CBT enables this redirection by recording, for each value of the case block variable, the corresponding case address. This mapping of case block variable values to case addresses is dynamically created. When a SWITCH/CASE statement is encountered, the value of the case block variable is used to search the CBT for the next fetch address. In effect, the CBT is dynamically generating a jump table to replace the SWITCH/CASE statements.

The study done by Kaeli et al. showed that an oracle CBT, that is, a CBT which always selects the correct case to execute, can reduce significantly the number of conditional branches in the instruction stream. However, the CBT's usefulness is limited by two factors. First, modern day compilers are capable of directly generating jump tables for SWITCH/CASE statements at compile-time, eliminating the need for the CBT to generate the tables dynamically. Second, for processors with out-of-order execution, the value of the case block variable is often not yet known when the instruction corresponding to the SWITCH/CASE statement is fetched. As a result, the CBT cannot redirect the instruction stream to the appropriate case until that value is computed.

2.5 Predicated Execution

Predicated execution was first proposed in the form of vector masks for vector machines such as the CRAY [30]. Many current commercial computer architectures (e.g. the Intel Pentium Pro, the DEC Alpha, the SPARC V9, and the HP PA-RISC) include some form of predicated execution.

Hsu and Davidson [14] studied the use of predicated instructions to better schedule decision trees on scalar processors. Instructions on the critical path of a decision tree are scheduled as early as possible to minimize the execution time through the decision tree. They proposed the decision tree scheduling (DTS) technique, which is based on an extension of list scheduling, to take advantage of guarded store and jump instructions, which relax the scheduling constraints due to control-flow dependencies. The DTS compilation technique with the guarded store and jump instructions are shown to be effective for both vector code and scalar code. However, instead of decision trees, directed acyclic graphs are commonly used to represent the control structure of a program.

Speculative and predicated execution have advantages, as well as disadvantages. Recent researchers have studied the effectiveness of combining speculative and predicated execution. Pnevmatikatos and Sohi [28] studied the performance benefit of using predicated execution in conjunction with the two-level branch predictor. They eliminated as many branches as possible through predicated execution. They proposed two models of predication, full guarding and restricted guarding. In the full guarding model, all instructions can be predicated. In the restricted guarding model, only ALU operations can be predicated. The full guarding model increased the average effective block size (the number of useful instructions in a basic block) by 25% and the average dynamic window size (the number of instructions issued between branch mispredictions) by 52%. However, the full guarding model also increased the total number of instructions that were executed; 33% of these instructions do not represent useful computation, wasting a significant amount of issue bandwidth. Using the restricted guarding model, where fewer instructions are predicated, a smaller percentage (8%) of all instructions executed do not represent useful work. However, the restricted guarding model is less effective than the full guarding model in increasing the effective block sizes and the dynamic window sizes. Instead of predicating as many branches as possible, the compiler needs to carefully select the branches to eliminate.

Tyson [36] studied the performance benefit of predicating all short forward branches. This aggressive approach of predicating branches of distance less than or equal to 12 reduced the number of instruction slots lost due to branches by as much as 50% for the SPEC92 benchmarks. However, not all short forward branches can be predicated. For example, a loop exit branch can not be eliminated. The performance benefit of predicating all short forward branches depends on the predictability of the short forward branches that can be eliminated.

Mahlke et al. [19, 18] used predicated execution to eliminate the branches that were difficult to predict for a static branch predictor and reported the effect on branch mispredictions and the performance of a statically scheduled machine. However, an aggressive branch predictor may be able to predict branches that were hard to predict for a simple predictor, resulting in a different set of branches that should be eliminated for the best processor performance.

CHAPTER 3

Simulation Methodology

3.1 Simulation Environment

3.1.1 Branch Prediction

Trace-Driven

Trace-driven simulations are used for the studies on branch prediction. Figure 3.1 shows the trace-driven simulation methodology. A Motorola MC88110 instruction level simulator (Archsim) reads in the object code and input data; it then simulates execution and produces an instruction trace. The instruction traces are then processed by either the restricted data flow (RDF) simulator [4] or the instruction fetch mechanism (IFM) simulator [37].

The RDF simulator simulates the execution of instructions on a given machine model, which is described in a configuration file read by the RDF simulator. The IFM simulator only simulates the actions involved in predicting instruction fetch address. Thus, the IFM simulator has a shorter simulation time than that of the RDF simulator; however, it only collects statistics associated with instruction fetch (e.g. the prediction accuracy of the branch predictor).

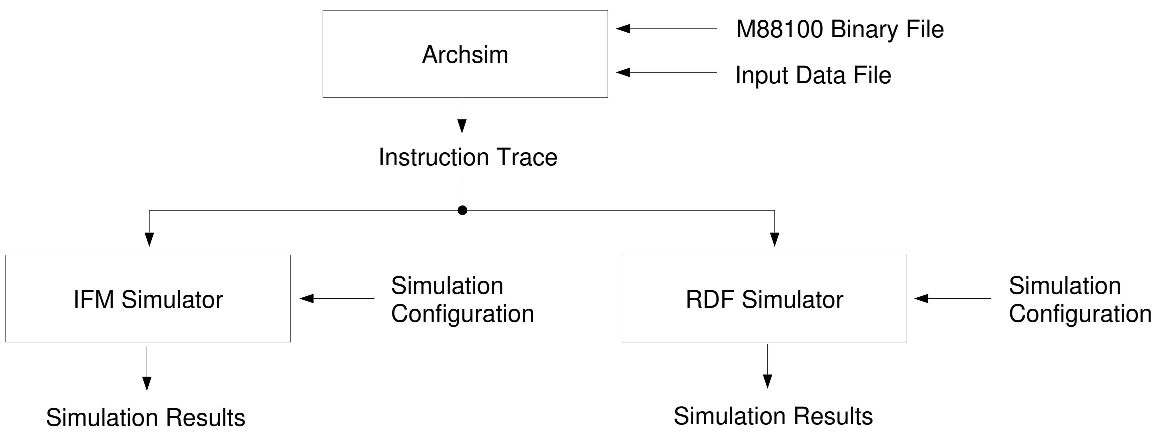


Figure 3.1: Simulation methodology

In this study, the RDF simulator is used to determine the execution time of the benchmarks on a given machine model. The IFM simulator is used to collect the hit rate of branch target buffer accesses and the prediction accuracy of branch predictors.

Source Code Annotation

Branch prediction simulators are embedded in the instrumented programs. To collect the dynamic branch behavior, the source code that calculates the branch conditions are replaced with calls to the branch simulator. For each of these calls, the branch predictor simulator generates the branch prediction and updates the state of the simulated prediction hardware. Using this method, the performance of various branch predictors can be compared on a per-branch basis. The behavior of the program is not changed because these function calls return the actual branch conditions; the program always executes down the correct path. This approach has a shorter simulation time than that of a trace-driven simulator but is not accurate enough to fine-tune a real design because we simulate the branches at the source code level. The behavior of some of these branches at the machine code level may be different than at the source code level due to compiler optimizations.

3.1.2 Predicated Execution

The predicated execution experiment takes two steps – generating the predicated object file and simulating the object file with the trace-driven simulator.

Generating the Predicated Object Files

The code compilation process consists of the following steps. First, we profile the benchmarks to identify the hard-to-predict branches. The assembly code for the source programs are generated using the GCC V2.4.3 compiler. We then hand-modify the assembly programs, predicating the hard-to-predict branches that can be eliminated. Finally, predicated object files are generated using the GNU assembler.

Simulation Process

The simulation process consists of the following steps (see Figure 3.2). First, we insert special instructions, `SPEC_OPs`, in the assembly program to indicate which

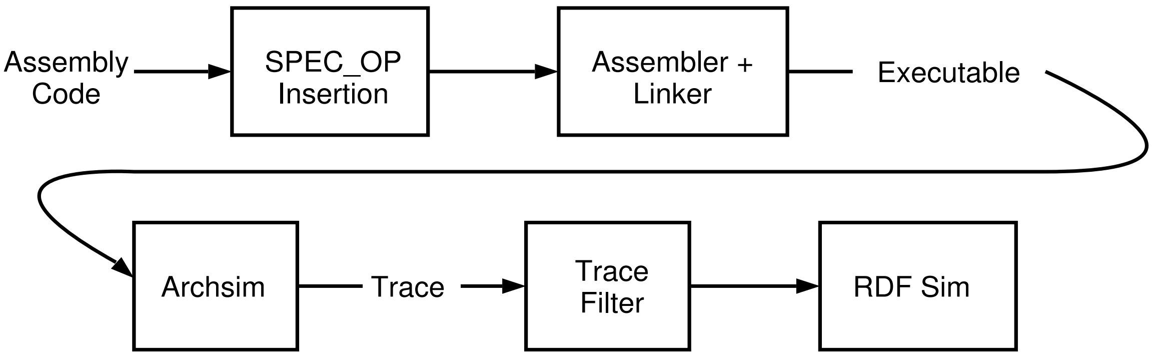


Figure 3.2: Process flow for simulating predicated instructions

branches are to be predicated. Figure 3.3 shows a code segment before and after a branch has been marked for predication. This modification to the program does not change its behavior because these special instructions are NOPs and do not affect the machine state. A Motorola MC88100 instruction level simulator, ArchSim, then reads in the object code and simulates execution, producing an instruction trace. To replace the selected branches with predicated instructions, a trace-filter module reads the trace generated by ArchSim and scans for occurrences of the `SPEC_OP` instruction. If a `SPEC_OP` instruction is detected, the filter module does the following:

- If the branch following the `SPEC_OP` instruction is not taken, we replace the instructions in the fall-through path with predicated instructions (see Figure 3.4).
- If the branch is taken, the instructions in the fall-through path will not be in the instruction trace. To determine what predicated instructions to insert into the new instruction stream, the filter module reads in a table which contains the predicated instructions associated with each branch (see Figure 3.5).

The new instruction trace is then processed by the trace-driven simulator, TraceSim, to produce the execution statistics.

Before	After
⋮	⋮
`beq r0, 0, L1`	`SPEC_OP`
`op1`	`beq r0, 0, L1`
`op2`	`op1`
`L1: ···`	`op2`
	`L1: ···`

Figure 3.3: Marking branches in the assembly code for predication with the special instruction `SPEC_OP`

Before	After
⋮	⋮
`SPEC_OP`	`op1 if r0`
`beq r0, 0, L1`	`op2 if r0`
`op1`	`L1: ···`
`op2`	
`L1: ···`	

Figure 3.4: Replacing a not taken branch in the instruction trace with the appropriate set of predicated instructions

Before	After

`⋮`
`SPEC_OP`
`beq r0, 0, L1`
`L1: ...`
	`⋮`
	`op1 if r0`
	`op2 if r0`
	`L1: ...`

Figure 3.5: Replacing a taken branch in the instruction trace with the appropriate set of predicated instructions

3.2 Benchmarks

3.2.1 Benchmarks

The results presented in this dissertation are for the six integer programs from the SPECint92 suite and the 8 integer programs from the SPECint95 suite. Table 3.1

SPECint92		
Benchmark	Abbr.	Description
008.espresso	esp	A tool for generating and optimizing Programmable Logic Arrays
022.li	li	A LISP interpreter
023.eqntott	eqn	A translator for converting a logical representation of a boolean equation to a truth table
026.compress	com	reduce size of an input file by using Lempel-Ziv coding
072.sc	sc	a spreadsheet calculator
085.gcc	gcc	GNU C compiler version 1.3.5

SPECint95		
Benchmark	Abbr.	Description
099.go	go	go-playing computer program which plays the game of go against itself
124.m88ksim	m88k	Motorola 88100 microprocessor simulator
126.gcc	gcc	GNU C compiler version 2.5.3
128.compress	com	reduce size of an input file by using Lempel-Ziv coding
130.li	li	A LISP interpreter
132.ijpeg	ijpeg	Image compression/decompression on in-memory images based on the JPEG facilities.
134.perl	perl	A Perl language interpreter
147.vortex	vortex	a single-user object-oriented database transaction benchmark

Table 3.1: Description of benchmarks

gives short descriptions of the benchmarks. Table 3.2 lists the input data set that was used, the number of instructions executed, and the number of branches executed for each benchmark.

SPECint92					
Benchmark	Input	#Dynamic Instructions	#Dynamic BR	#Dynamic Cond BR	#Static Cond BR
008.espresso	bca	288,728,123	68,130,561	64,273,636	1,245
022.li	nine queens	100,000,000	23,004,735	14,777,111	501
023.eqntott	int_pri_3.eqn	100,000,000	21,658,145	19,343,202	395
026.compress	in	86,445,440	14,099,619	11,178,108	177
072.sc	loada1	143,176,340	36,330,310	30,484,812	830
085.gcc	stmt.i	107,163,102	22,131,439	17,252,756	7,396

SPECint95					
Benchmark	Input	#Dynamic Instructions	#Dynamic BR	#Dynamic Cond BR	#Static Cond BR
099.go	2stone9.in	161,696,039	23,378,150	18,462,968	5,219
124.m88ksim	dcrand.train.big	131,732,141	23,840,021	17,529,089	1,033
126.gcc	jump.i	172,329,018	35,979,748	27,410,921	16,185
128.compress	test.in	125,162,687	17,460,753	10,661,859	310
130.li	train.lsp	192,569,022	40,909,525	27,374,670	537
132.ijpeg	specmun.ppm	231,543,794	23,449,572	20,507,891	1,178
134.perl	scrabbl.pl	106,140,746	16,727,047	10,606,041	1,761
147.vortex	vortex.in	236,081,621	44,635,060	34,211,423	6,436

Table 3.2: Branch counts of benchmarks

Branch Characteristics

Branches in the program can be categorized into four different types – conditional branch, immediate branch, indirect branch, and return. Figure 3.6 shows the distribution of the different types of branches in the dynamic instruction traces. Approximately 73% of branches in SPECint95 are conditional branches. Since conditional branches are executed frequently, processors must effectively handle these branches in order to achieve their potential performance.

Figure 3.7 shows the distribution of the static conditional branches with different dynamic execution frequencies. For the SPEC integer benchmarks, approximately 30 percent of the static branches execute more than 1000 times. Figure 3.8 shows the weighted distribution the static branches with different dynamic execution frequencies, where the branches are weighted by the number of their dynamic occurrences. This figure shows that the 30% of the static branches accounts for approximately

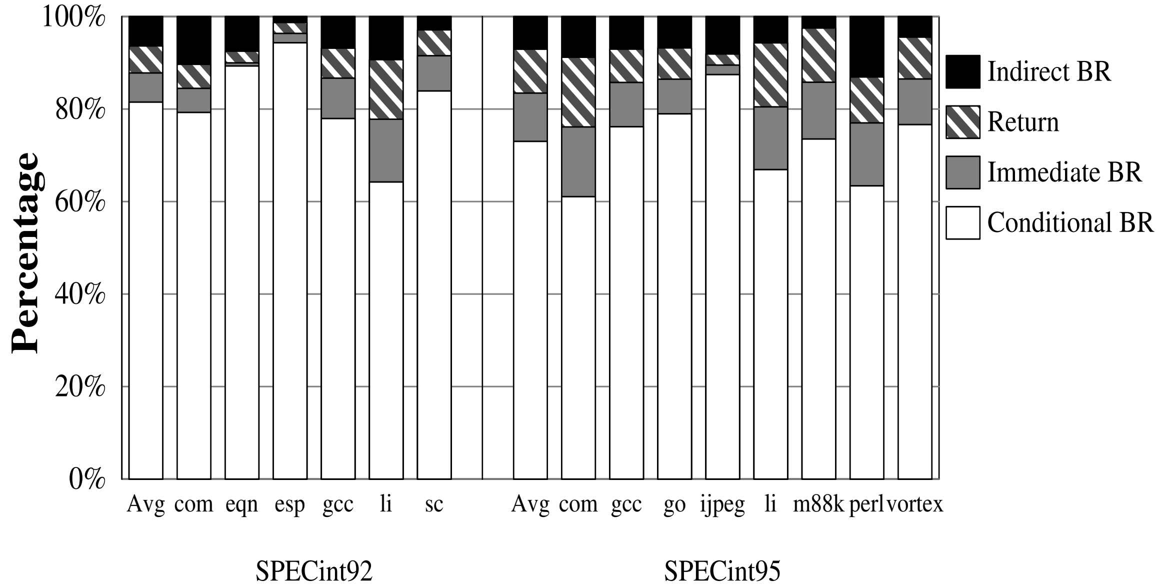


Figure 3.6: Distribution of dynamic branch instructions

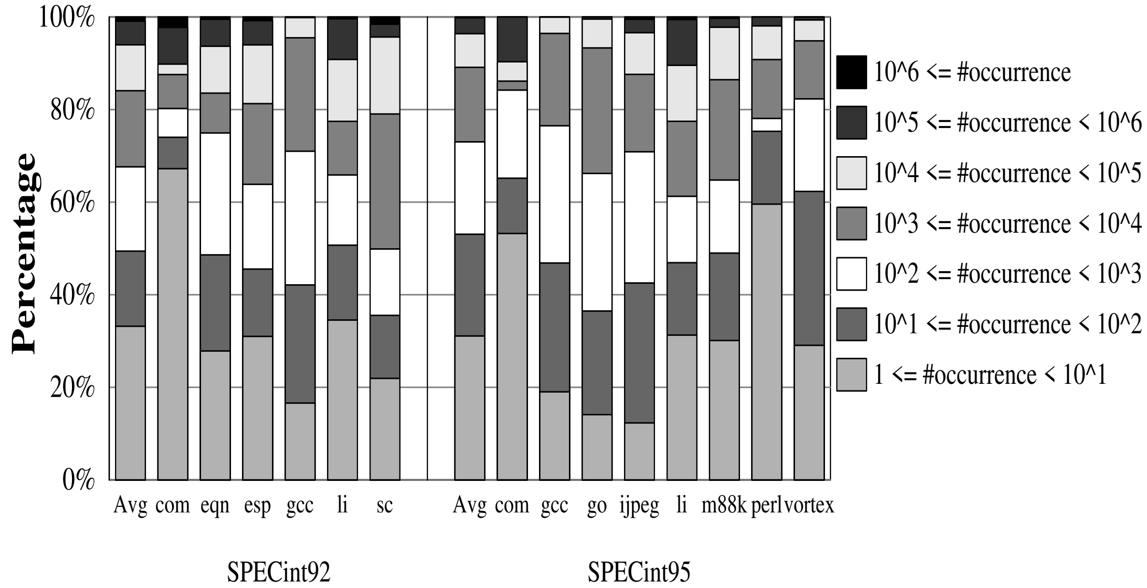


Figure 3.7: Distribution of the static branches with different dynamic execution frequencies

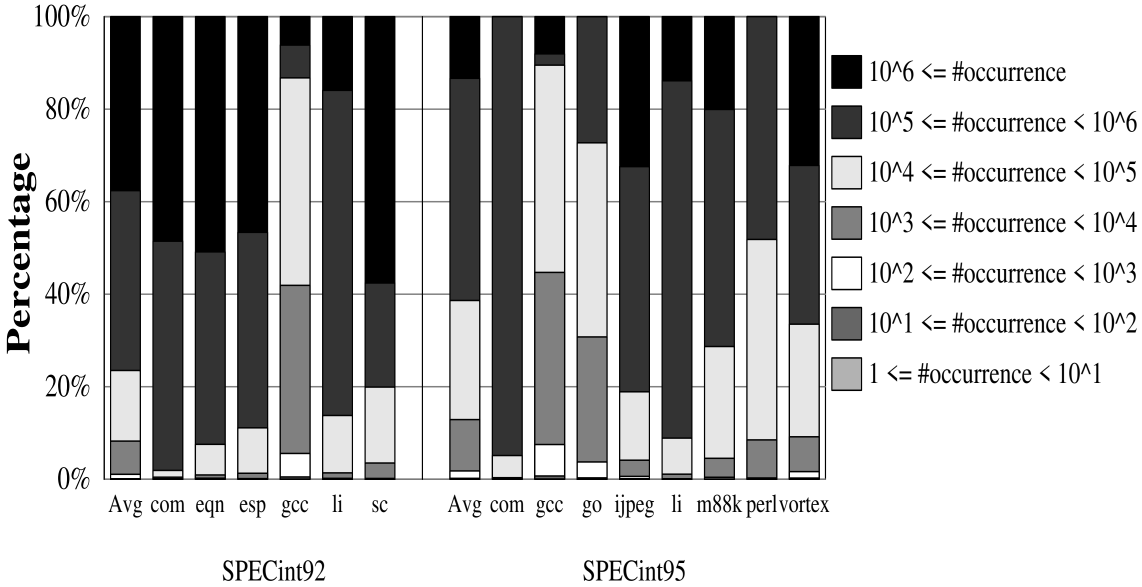


Figure 3.8: Weighted distribution of the static branches with different dynamic execution frequencies

90% of the dynamic branches. Thus, branch handling mechanisms must effectively handle these conditional branches in order for the processor to achieve its potential performance.

3.3 Machine Model

The machine model simulated is the HPS microarchitecture [26] [27]. Figure 3.9 shows the block diagram of the HPS architecture. Execution in HPS flows as follows: Each cycle, multiple instructions are issued, and using the information in the register files, the instructions are merged into node tables, much like the Tomasulo algorithm merges operations into the reservation stations of the IBM 360/91 [35]. Associated with each instruction (node) are the source operands for that instruction (or identifiers for obtaining the operands), and destination information. Each node is stored in its proper node table independent of and decoupled from all other nodes currently awaiting dependencies in the datapath until all its operands are available, at which point the node is eligible for scheduling. Each cycle, the oldest firable node of each node table is scheduled, i.e., it is shipped to a pipelined functional unit for execution. Each cycle, functional units complete execution of nodes and distribute the results to nodes waiting for these results, which then may become firable. Checkpointing [15] is used to maintain precise exceptions. Checkpoints are established for each branch; thus, once a branch misprediction is determined, instructions from the correct path are fetched in the next cycle.

The HPS processor simulated in this dissertation supports 8 wide issue with a

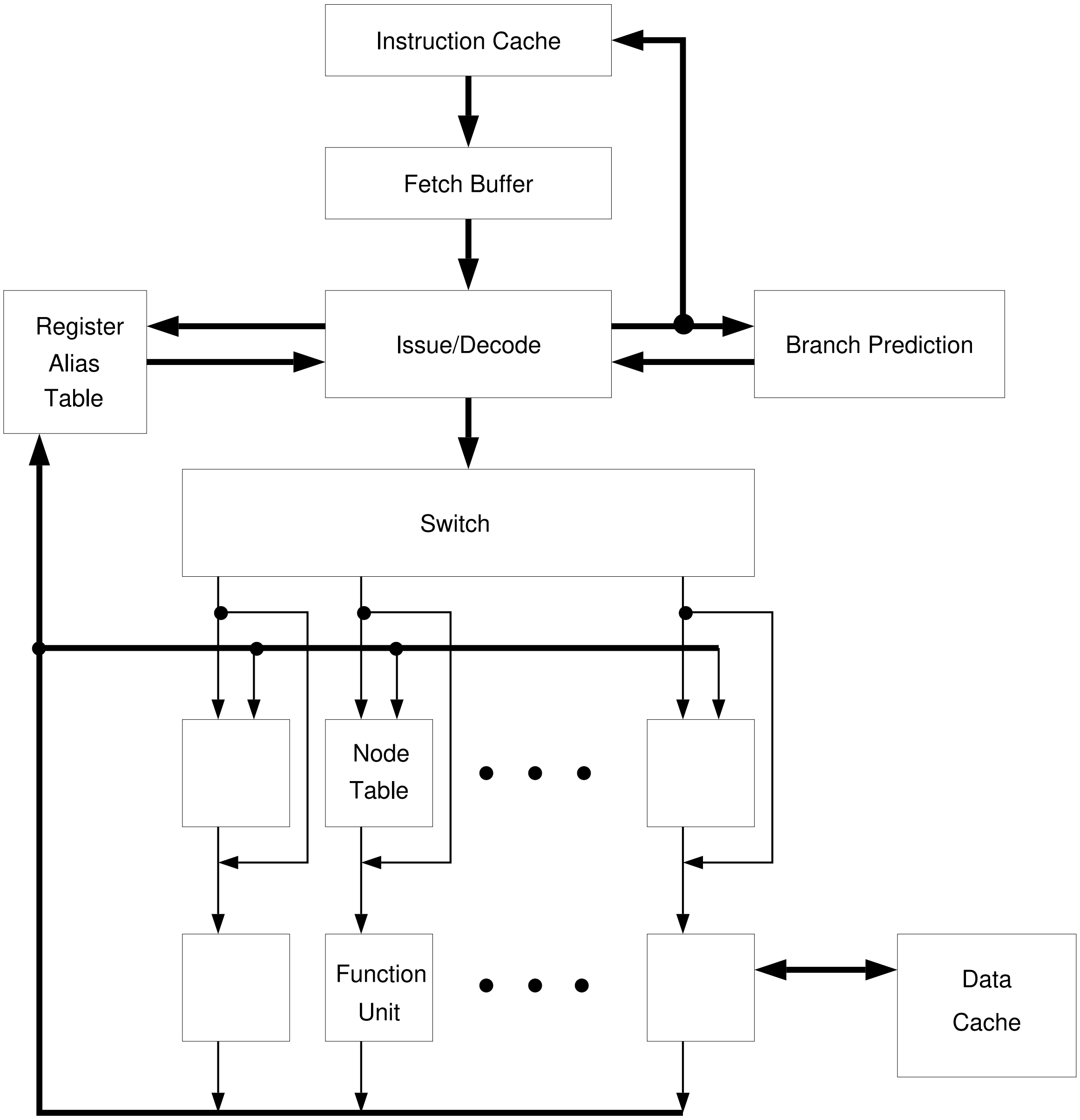


Figure 3.9: HPS block diagram

perfect instruction cache and a 16KB data cache. Latency for fetching data from memory is 10 cycles. Table 3.3 shows the instruction classes and their simulated execution latencies, along with a description of the instructions that belong to that class. In the processor simulated, each functional unit can execute instructions from any of the instruction classes. The maximum number of instructions that can exist in the machine at one time is 128. An instruction is considered in the machine from the time it is issued until it is retired.

Instruction Class	Exec. Lat.	Description
Integer	1	INT add, sub and logic OPs
FP Add	3	FP add, sub, and convert
FP/INT Mul	3	FP mul and INT mul
FP/INT Div	8	FP div and INT div
Load	2	Memory loads
Store	-	Memory stores
Bit Field	1	Shift, and bit testing
Branch	1	Control instructions

Table 3.3: Instruction classes and latencies

CHAPTER 4

Branch Classification

Branch classification partitions a program's branches into sets or *branch classes*. A good classification scheme partitions branches possessing similar dynamic behavior into the same branch class; thus, once we understand the dynamic behavior of a class of branches, we can optimize the handling of this class.

4.1 The Advantages of Branch Classification

To demonstrate the usefulness of branch classification, we propose one model of branch classification and show how this model can be used to improve branch predictors. This branch classification model partitions branches based on their taken rates, which are collected during the profile run, as shown in Table 4.1. These branch classes are referred to as *static classes* because the partitioning of branches is done statically. We will refer to SC1, SC2, SC5, and SC6 branches as mostly-one-direction branches and the SC3 and SC4 branches as mixed-direction branches.

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

Table 4.1: Static classes

4.1.1 Analysis of Branch Predictors

The importance of accurately predicting a class of branches depends on how frequently these branches are executed in the dynamic instruction stream. Figure 4.1 shows the dynamic weight of each static class, where the dynamic weight of a branch class is defined as

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

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

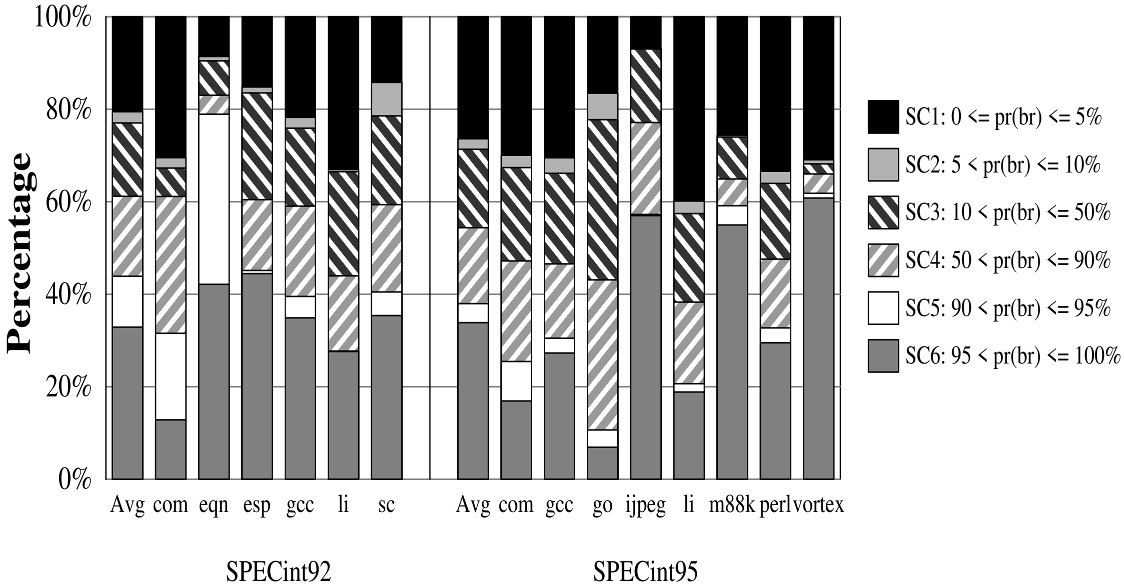


Figure 4.1: Percentage of dynamic branches in each static class

Analysis of the Mostly One-direction Branches (SC1,SC2,SC5,SC6)

Figure 4.2, shows the misprediction rate of the SC1 branches using the GAs predictor. Each curve shows the misprediction rate for a constant cost of implementation. The hardware costs of two-level branch predictors[1] are estimated using the following equations [40]:

$$\begin{aligned} \text{GAs}(k,p) &= \quad k \quad + (p \times 2^k \times 2) \qquad (bits) \\ \text{PAs}(k,p) &= (b \times k) + (p \times 2^k \times 2) \qquad (bits) \\ \text{gshare}(k,p) &= \quad k \quad + (p \times 2^k \times 2) \qquad (bits) \end{aligned}$$

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

[1] The performance of the gshare and PAs predictors are also examined. Their results are similar to those of GAs (see Appendix A.2 and A.3).

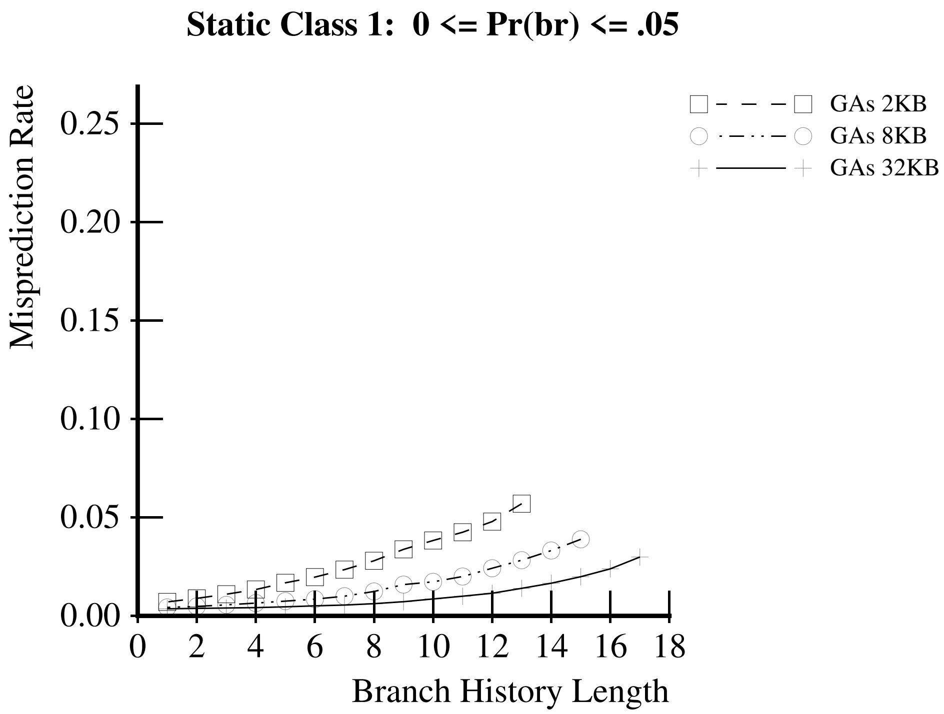


Figure 4.2: Misprediction rate of GAs on SC1 branches

Since the two-level branch predictor uses the contents of the BHR to select the appropriate PHT entry (as shown in Figure 2.1), the length of the branch history register indicates the size of each PHT; for example, the k-bit BHR implies PHTs of 2^k entries. Thus, for a given hardware cost, an implementation of the two-level branch predictor with a longer BHR contains larger, but fewer, PHTs. Likewise, a shorter BHR means smaller, but more, PHTs. Therefore, for each curve in Figure 4.2, as the branch history length decreases by one, the number of pattern history tables doubles.

As shown in Figure 4.2, branch prediction schemes with short history registers are most effective in predicting the mostly-not-taken branches (SC1). For implementations of a fixed cost, there is a trade-off between having a longer branch history and having more PHTs. One advantage of having a shorter branch history is a faster predictor warm-up time because fewer PHT entries are accessed with a shorter BHR. A shorter history also reduces the amount of PHT interference because we have more PHTs resulting in fewer branches being mapped to the same PHT. Since branches in SC1 are mostly not-taken, prediction accuracy remains high even if the taken occurrences of the branch are mispredicted. Although a longer branch history register can keep more history of correlated branches and possibly capture these odd occurrences, the benefit of having a faster predictor warm-up time and less PHT interference outweighs the benefit of eliminating these mispredictions. Similar results are shown for the SC2, SC5, and SC6 branches because these classes also have the same dynamic characteristic of heavily favoring one direction over the other (see Appendix A.1).

Analysis of the Mixed-direction Branches (SC3,SC4)

Figure 4.3 shows the misprediction rate of SC3 branches using the GAs predictor. Unlike the mostly-one-direction branches, the mixed-direction branches are most effectively predicted by prediction schemes with long branch history registers. Because these branches have dynamic taken rates between 10% and 50%, the predictor sees more execution patterns due to the mixing of taken and non-taken directions in the branch history. By having a longer branch history, we can distinguish more execution states. With a longer branch history, the histories of correlated branches are more likely to remain in the branch history register, making the branch prediction scheme more effective in predicting the mixed-direction branches. Figure 4.3 also shows that the performance of GAs decreases when its branch history register is too long. For example, the performance of 2KByte GAs predictors starts to decrease when the branch history length becomes greater than 11. As the branch history length increases, the amount of PHT interference increases because we have fewer PHTs. When the branch history register is too long, the performance lost due to PHT interference outweighs the performance gain by using a longer branch history register. The SC4 branches with taken rates between 50% and 90% have similar results to the SC3 branches (see Appendix A.1).

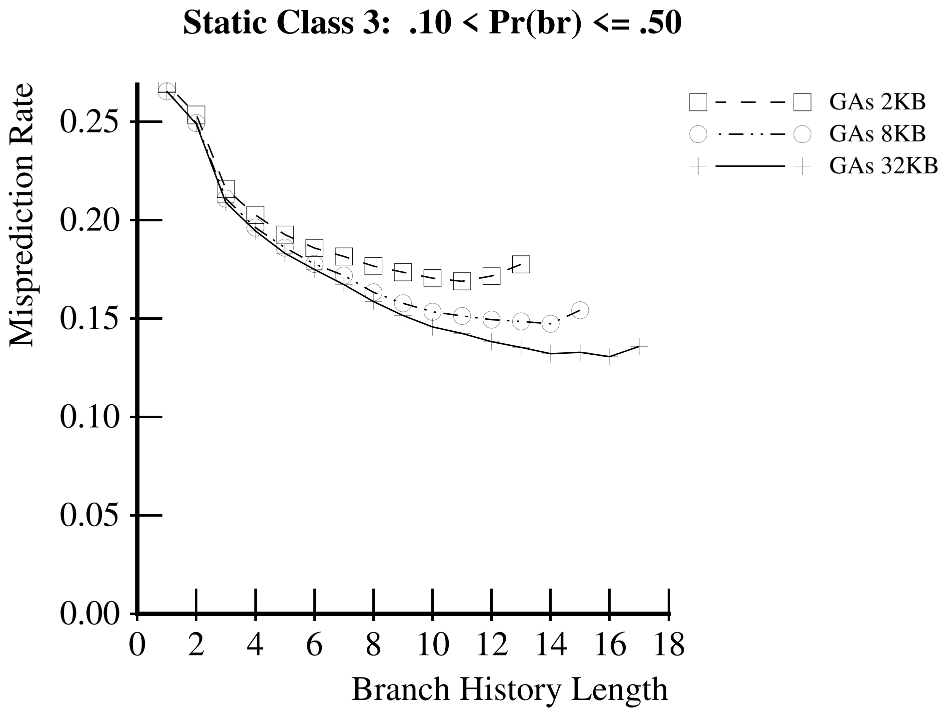


Figure 4.3: Misprediction rate of GAs on SC3 branches

Performance of Branch Predictors

We have shown that the optimal predictor configuration of the two-level branch predictor for the mostly-one-direction branches is different from that of the mixed-direction branches. Thus, a single-scheme predictor cannot be configured optimally for both types of branches. Figure 4.4 shows the average misprediction over all branches using the GAs predictor with the branch history length ranging from 1 to 17. Each curve in the graphs indicates the performance of a branch predictor at a fixed hardware cost. Our results show that the best predictor configurations for predicting all branches have branch history lengths that are neither optimal for the mostly-one-direction branches nor for the mixed-direction branches. For example, Figure 4.4 shows that the best 32K-byte GAs configuration for predicting all branches uses 13 bits of branch history while Figure 4.3 shows that for the mixed-direction branches only, the best branch history length is 16 and Figure 4.2 shows that for the mostly-one-direction branches only, the best branch history length is 1. Thus, a branch predictor which uses two different branch history lengths will outperform single-scheme predictors which use only one branch history.

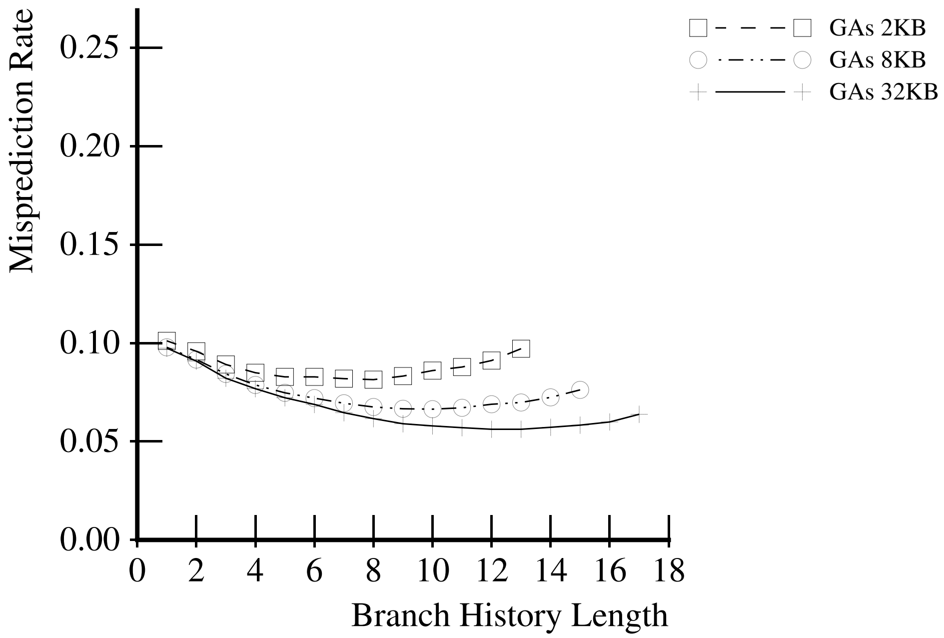


Figure 4.4: Performance of GAs with different branch history length

4.1.2 Improving Branch Predictor Performance

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

Since static predictors can accurately predict the mostly-one-direction branches [7], we use static predictors for these mostly-one-direction branches and dedicate our hardware for predicting the mixed-direction branches. For the mixed-direction branches, we use an aggressive hybrid branch predictor, PAs/gshare [20]. The resulting predictor, PG+PAs/gshare, uses the profile-guided predictor for the mostly-one-direction branches and the PAs/gshare scheme for the mixed-direction branches[2]. Figure 4.5 shows the structure of the PG+PAs/gshare predictor.

Figure 4.6 shows the performance of PG+PAs/gshare on the SPECint92 benchmarks. For predictors larger than 4K bytes, PG+PAs/gshare outperforms all other

[2]we use the "+" to indicate static predictor selection and "/" for dynamic predictor selection.

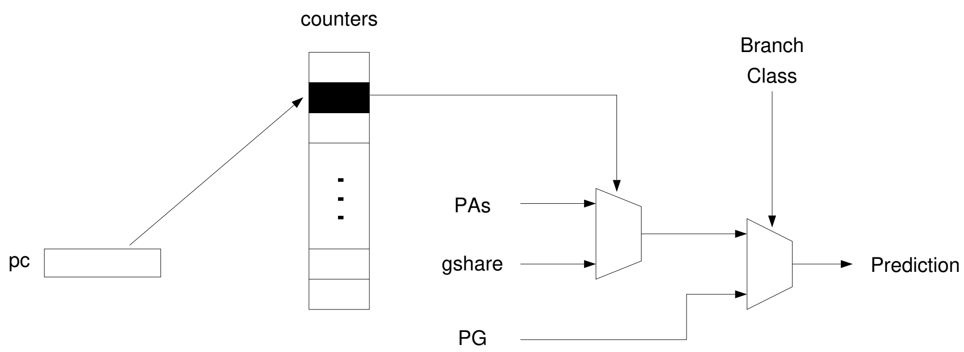


Figure 4.5: Structure of the PG+PAs/gshare Hybrid Branch Predictor

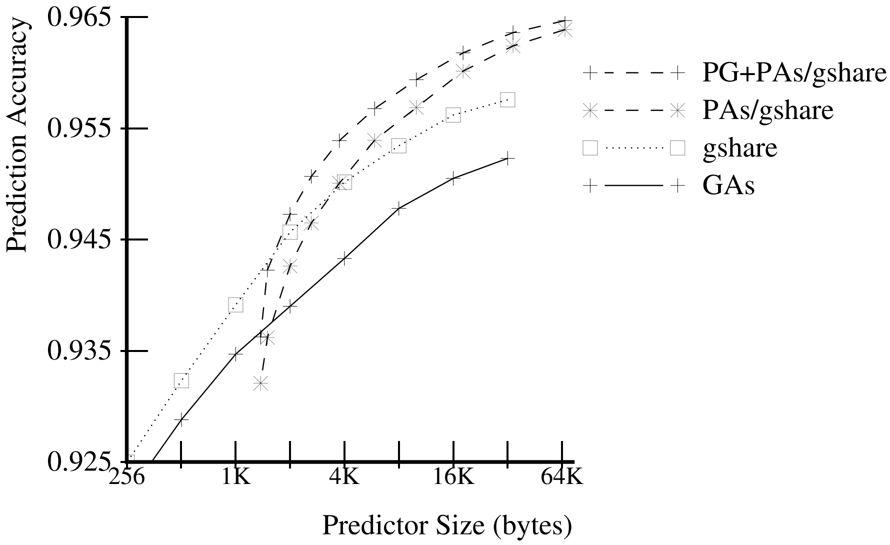


Figure 4.6: Performance of the PG+PAs/gshare Hybrid Branch Predictor

predictors. For example, with a fixed implementation cost of 32K bytes, PG+PAs/gshare is able to achieve prediction accuracy of 96.4% on the SPEC integer benchmarks, as compared to 95.7% for gshare and 95.2% for GAs. Using PAs/gshare for predicting only the mixed-direction branches, we remove the pattern history table interference between the mostly-one-direction branches and the mixed-direction branches. This reduction in the pattern history table interference improves the PAs/gshare's accuracy in predicting the mixed-direction branches (Chapter 6 will examine the effect of pattern history table interference on two-level branch predictors in greater detail). For the SPEC92 benchmark gcc, which contains many branches, PG+PAs/gshare achieves prediction accuracy of 96.91%, as compared to 96.47% for the best previously known predictor (PAs/gshare) [20]. PG+PAs/gshare outperforming PAs/gshare indicates that branch classification can be helpful in designing more accurate branch predictors.

4.2 Summary

Branches in a program can be categorized into different classes. Since branches in the different classes can have different behaviors, the most suitable mechanism to handle each branch can be different. We introduced branch classification as a means for identifying and applying the most suitable mechanism for each branch.

To demonstrate the benefits of branch classification, a branch classification model that groups branches into classes based on their dynamic taken rates was introduced. With this model, we showed that the mostly-one-direction branches are most accurately predicted by simple predictors and that the mixed-direction branches are most accurately predicted by complex predictors. Thus, single-scheme predictors can not be configured optimally for both types of branches. Using branch classification, a new hybrid branch predictor is built which achieves a higher prediction accuracy than any branch predictor previously reported in the literature. With a fixed implementation cost of 32K bytes, the new hybrid branch predictor achieved a prediction accuracy of 96.91% on gcc, a branch intensive benchmark, as compared to 96.47% for the best previously known predictor, reducing the miss rate by 12.5%.

CHAPTER 5

Hybrid Branch Predictors

To improve prediction accuracy, several hybrid branch predictors have recently been proposed [20, 7, 6]. They combine multiple prediction schemes into a single predictor and use a selection mechanism to decide for each branch, which single-scheme predictor to use (see Figure 5.1). An effective hybrid branch predictor can exploit the different strengths of its single-scheme predictor components, enabling it to achieve a prediction accuracy greater than that which could be achieved by any of its components alone.

The performance of a hybrid branch predictor depends on its component predictors and its selection mechanism. In this chapter, we evaluate the performance of various hybrid branch predictors. In addition, we propose the 2-level branch predictor selection mechanism, which uses more run-time information to improve the performance of the predictor selection mechanism.

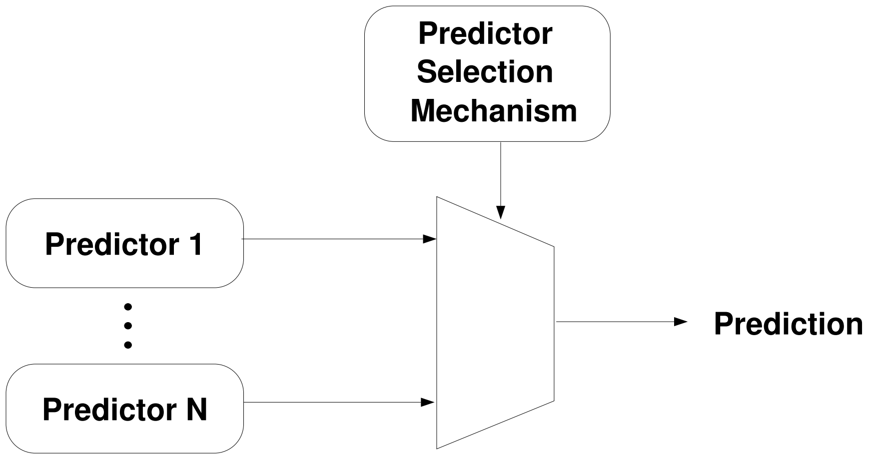


Figure 5.1: Structure of a hybrid branch predictor

5.1 Component Predictors

An effective hybrid branch predictor combines the different strengths of its single-scheme predictor components to achieve a greater prediction accuracy. Combining two predictors that both achieve high prediction accuracies on the same subset of a program's branches will not yield a hybrid predictor with a significant increase in performance. In this section, the performance of various combinations of single-scheme predictors are compared in order to identify the most effective combinations.

Only hybrid branch predictor configurations consisting of two single-scheme predictors were considered. To identify the best configuration among them, every configuration in the design space must be considered. Because this would have been completely unmanageable, we opted for a process that assumed an idealized selection mechanism. This reduced the number of simulations that we had to consider to the number of single-scheme predictors. Our idealized selection mechanism operated as follows: Using foreknowledge of each single-scheme predictor's performance, we mapped each static branch to the single-scheme predictor (of the pair being considered) that would achieve the higher prediction accuracy for that branch over the entire run of the benchmark. With this assumption, the performance of a given configuration can be quickly calculated by examining the simulation results of its single-scheme predictor components as follow:

Each single-scheme predictor component of the hybrid predictor was first simulated by itself, using the instruction fetch mechanism simulator described in Section 3.1.1. The simulation recorded the number of correct and incorrect predictions for each static branch in the benchmark. To measure the performance of the hybrid predictor, the simulation results for the two single-scheme components were analyzed. For each static branch, the prediction results for the single-scheme component that achieved the higher prediction accuracy were used to represent the hybrid predictor's performance for that branch, duplicating the behavior of the idealized selection mechanism.

Note that while this selection mechanism achieves optimal performance for a static selector, it may not achieve optimal performance for a dynamic selector. A dynamic selector can choose a different predictor during different periods of a program's execution based on the particular dynamic information available at that time. This allows it to potentially achieve higher prediction accuracies for branches that under different circumstances are more accurately predicted by different predictors. However, the consistent ordering of predictor class combinations shown in our results (see Section 5.1.2) supports the possibility that the best predictor class combination for the ideal static selector is also best for dynamic selectors.

5.1.1 Single-Scheme Predictors

For a given hardware cost, the hybrid predictor configuration specifies the classes of the single-scheme predictors used and the amount of hardware devoted to each scheme. The single-scheme predictors examined were divided into four classes:

1. static - the profile-guided branch predictor [11].
2. 2bC(n) - the two bit counter predictor [32]. It consists of an array of n two bit counters.
3. PAs(k,s) - the per-address variation of the Two-Level Adaptive Branch Predictor [39] consisting of 1K k-bit branch history registers and s pattern history tables.
4. gshare(k) - a modified version of the the two-level branch predictor [20] consisting of a single k-bit global branch history and a single pattern history table.

The static predictor was considered because it is a compile-time predictor which has no hardware cost. The 2bC predictor was considered because it is still used by many of the current generation of commercial microprocessors. The PAs and gshare predictors were considered because they are variations of the highest performing single-scheme predictor, the two-level branch predictor [38, 25, 39, 40, 20].

For each predictor type, a range of predictor sizes was considered allowing us to vary the amount of hardware devoted to each scheme. The 2bC array size was varied from 2^{10} to 2^{20} entries. The branch history registers for the gshare and PAs schemes were varied from 10 to 20 bits. The number of pattern history tables for the PAs scheme was varied from 1 to 4. The hardware cost for each predictor type was estimated by the equations in Table 5.1. Because different sizes were considered for each predictor class, it is possible for a given combination of predictor classes to have multiple representatives at a given level of hardware cost.

Predictor	Cost (Bits)
static	0
2bC(n)	$2n$
PAs(k,s)	$2^{10}k + 2^{k+1}s$
gshare(k)	$k + 2^{k+1}$

Table 5.1: Hardware costs for the four classes of single-scheme predictors.

5.1.2 Experimental Results

Every possible combination of single-scheme predictors from the set considered was simulated. Figure 5.2 lists for six levels of hardware cost (8KB–256KB) the misprediction rates of the best representative for each combination of single-scheme predictor classes. The best single-scheme predictor is included as well. The misprediction rates are the average of the rates achieved for the six SPECint92 benchmarks. The corresponding hardware cost level for each combination was determined by rounding its exact cost up to the next closest level.

At every level of hardware cost (with a minor exception at the 8KB level), the ordering of the predictor class combinations was the same with the gshare/PAs combination always achieving the lowest misprediction rate. This misprediction rate was

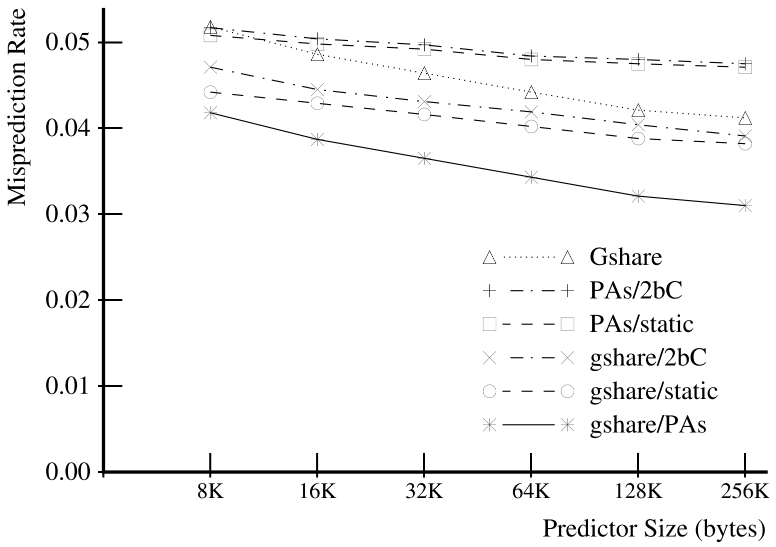


Figure 5.2: Misprediction rates of the best representative for each possible combination of single-scheme predictor classes.

on average 13% lower than that of its closest competitor, gshare/static. The best single scheme predictor at all levels of cost was gshare. Although it outperformed the PAs/static and PAs/2bC hybrid combinations at cost levels above 8KB, it was outperformed by all the gshare hybrid combinations. The gshare/PAs combination was able to achieve the best performance because it was the only combination that effectively exploited both inter-branch and intra-branch correlation. The gshare component was able to accurately predict branches whose outcomes are dependent on the outcomes of other static branches. The PAs component was able to accurately predict branches whose outcomes are dependent on previous outcomes of the same static branch. Working together, they enabled the hybrid predictor to accurately predict a larger set of branches than what could be accurately predicted by a predictor that contained only one of the components.

When comparing the predictor combinations on a per-benchmark basis, the gshare/PAs combination was still always the best with the exception of the gcc benchmark. For gcc, the gshare/static and gshare/2bC achieved lower misprediction rates than gshare/PAs. Gcc's results differed from the other benchmarks because it contains a large number of static branches in its working set. This large set can cause interference in the pattern history tables of the gshare and PAs predictors, reducing their ability to make accurate predictions [41, 34]. In addition, the large number of branches can incur a significant training cost. Both the gshare and PAs predictors must train themselves on the first few instances of the branch before they can begin to accurately predict it. Because both the static and 2bC schemes suffer little or no performance penalties due to interference or training, pairing either one with the gshare predictor produces a more effective hybrid predictor than gshare/PAs. Despite its weaknesses, gshare is still used as one of the components because there are still a significant number of branches in gcc for which gshare is the most effective predictor.

Table 5.2 lists the exact configurations for the best hybrid predictor (i.e. gshare/PAs combination) at each cost level. For every one of these combinations, half of the hardware was devoted to the gshare component and half was devoted to the PAs component. This result is due to the gshare component's cost and the hardware cost levels considered both always being a power of two. Thus, the configuration which maximizes the amount of hardware devoted to the gshare component while still leaving space for a PAs component is one that divides the hardware evenly. Table 5.2 also lists the exact configurations for the best hybrid predictors when each predictor's cost is rounded to the closest level instead of always upwards. In this case, the best combinations are the same as before except that the size of the gshare component is doubled so that the gshare component occupies two-thirds of the hardware budget. As in the upward rounding case, these configurations are the ones that maximize the gshare component size while still affording space for a PAs component. Table 5.3 lists the best hybrid predictors for the gcc benchmark at each cost level.

Cost	Rounding Model	
(KB)	Upwards	Nearest
8	gsh(14)/PAs(10,4)	gsh(15)/PAs(10,4)
16	gsh(15)/PAs(12,4)	gsh(16)/PAs(12,4)
32	gsh(16)/PAs(13,4)	gsh(17)/PAs(13,4)
64	gsh(17)/PAs(16,1)	gsh(18)/PAs(16,1)
128	gsh(18)/PAs(15,4)	gsh(19)/PAs(15,4)
256	gsh(19)/PAs(16,4)	gsh(20)/PAs(16,4)

Table 5.2: Configurations for the best hybrid combination where *upwards* rounds cost to the next highest level and *nearest* rounds cost to the closest level.

Cost	Rounding Model	
(KB)	Upwards	Nearest
8	gsh(14)/static	gsh(15)/static
16	gsh(15)/static	gsh(16)/static
32	gsh(16)/static	gsh(17)/static
64	gsh(17)/static	gsh(18)/static
128	gsh(18)/static	gsh(19)/static
256	gsh(19)/2bC(18)	gsh(20)/2bC(18)

Table 5.3: Configurations for the best hybrid combination for gcc.

5.2 Selection Mechanisms

The performance of a hybrid branch predictor also depends on its predictor selection mechanism. In the previous section, we have determined the optimal configurations of hybrid branch predictors with an idealized static selection mechanism. In this section, we will use these same configurations for evaluating the performance of our real selection mechanisms. We propose a new technique, the 2-level branch predictor selection mechanism, which uses more run-time information to improve the performance of the predictor selection mechanism.

5.2.1 2-level Branch Predictor Selection Algorithm

It is now well-known that the two-level branch predictor improves prediction accuracy over previously known single-level branch predictors [38]. The concepts embodied in the two-level branch predictor can also be applied to the hybrid branch predictor selection mechanism. Figure 5.3 shows the structure of the 2-level branch predictor selection mechanism. A Branch History Register (BHR) holds the branch outcomes of the last m branches encountered, where m is the length of the BHR. This first level of branch history represents the state of branch execution when a branch is encountered. No extra hardware is required to maintain the first level of history if one of the component predictors already contains this information. That is, if the component predictor maintains a BHR, then the 2-level BPS mechanism does not need to maintain another copy of the BHR; instead, it just uses the component predictor's BHR. The Branch Predictor Selection Table (BPST) records which predictor was most frequently correct for the times this branch occurred with the associated branch history. This second level of history keeps track of the more accurate predictor for branches at different branch execution states.

When a branch is fetched, its instruction address and the current branch history is used to hash into the BPST. The associated counter is then used to select the appropriate prediction. By using the branch history to distinguish more execution states, a 2-level predictor selection scheme can more accurately select the appropriate predictions.

Since the BPST and the PHT can be accessed in parallel, the time required for a hybrid predictor to make a prediction is *max(BPST_time, PHT_time) + mux_time*

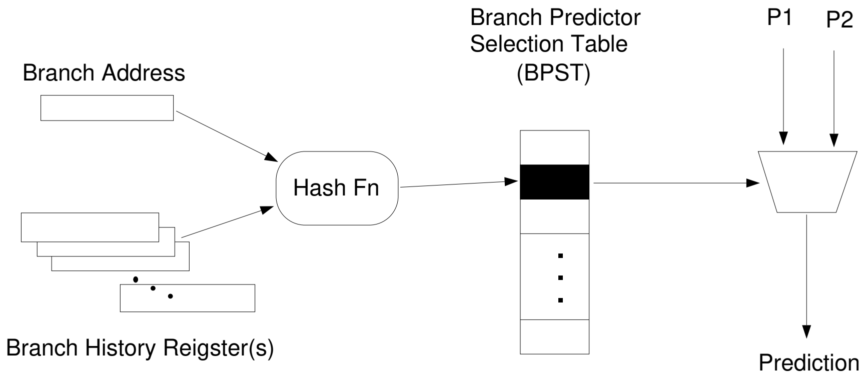


Figure 5.3: Structure of 2-level Predictor Selection Mechanism

where *BPST_time, PHT_time, and mux_time* are the access time of BPST, PHT, and mux respectively. Since the PHT is often much larger than the BPST, *PHT_time* is usually greater than *BPST_time*. Thus, the time for making a prediction is *PHT_time + mux_time*, which is almost equivalent to the access time of single-scheme predictors.

Several variations of the 2-level predictor selection mechanism can be implemented. They are different in the manner in which the first level of branch history is kept and in the hashing function used to index into the BPST. For example, the first-level branch history can be Global, Per-set, or Per-address. In the Global history scheme, the first-level branch history is composed of the last m branches encountered in the dynamic execution stream. In the Per-address scheme, the first-level branch history is composed of the last m occurrences of the same branch instruction. In the Per-set scheme, the first-level branch history is composed of the last m occurrences of branches in the same set. For the hashing function, we can, for example, XOR or concatenate the branch history with the instruction address.

5.2.2 Performance of the 2-level Branch Predictor Selection Mechanism

Four variations of the 2-level predictor selection mechanism were studied. Two types of first-level branch history were examined: (1) Global and (2) Per-address. Two hashing functions were studied: (1) the branch history is XORed with the instruction address and (2) the branch history is concatenated with the instruction address. Table 5.4 summarizes these different configurations.

For each one of these schemes, various branch history lengths can be used for indexing into the BPST. Rather than showing all these possibilities, Figure 5.4 only

Hash Function \ History Type	Global	Per-address
XOR	gXOR	pXOR
Concatenate	gCONC	pCONC

Table 5.4: Summary of 2-level predictor selection

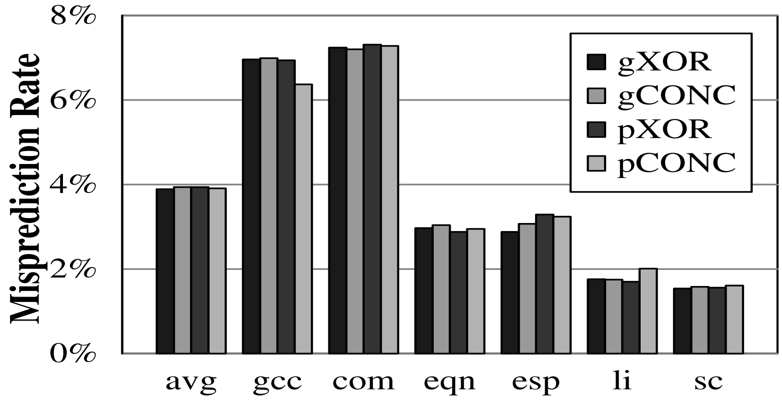


Figure 5.4: Performance of Various 2-level BPS Mechanisms

shows the performance of the predictors with the highest accuracy across the benchmarks. The pCONC selection mechanism uses three bits of branch history, while all other methods use ten bits. The configuration of the hybrid branch predictor considered is gshare(16)/PAs(12,4), the best hybrid combination at implementation cost of 16Kbytes when predictors' costs are rounded to the closest level. The BPST is a 1K-entry table of two bit counters. The misprediction rate in the graph indicates the rate at which the chosen predictor makes incorrect prediction.

With the exception of gcc, all four hashing schemes have similar performance. Young et al. [41] and Talcott et al. [34] have shown how branch prediction table interference can affect the performance of branch prediction schemes. Similarly, interference in the BPS table can affect the performance of the branch predictor selection mechanism. One advantage of using the branch history is that it reduces BPST interference. This can be accomplished by utilizing more of the BPST, reducing the number of branch instances that hash to a given BPST entry. For example, the espresso benchmark, with the 2-bit counter selection mechanism, utilizes 21.7% of the BPST. With 2-level selection mechanism such as gXOR, 89.2% of the BPST is utilized. This reduction in BPST interference results in gXOR's higher prediction accuracy. For benchmarks where BPST interference is low, all four selection schemes have similar performance.

However, when there is already too much contention for the counters between different branches, as in gcc, the pCONC scheme outperforms the other three schemes. Conceptually, the pCONC scheme partitions the BPST into several sections. The branch address bits are used to select the appropriate section and the branch history bits are used to choose the appropriate counter within each section. The pCONC scheme with a small number of branch history bits results in lower misprediction rates; a shorter BHR partitions the BPST into a larger number of sections. With more sections, fewer branches are mapped to the same section, reducing the amount of BPST interference.

Figure 5.5 compares the performance of the gXOR selection mechanism with that

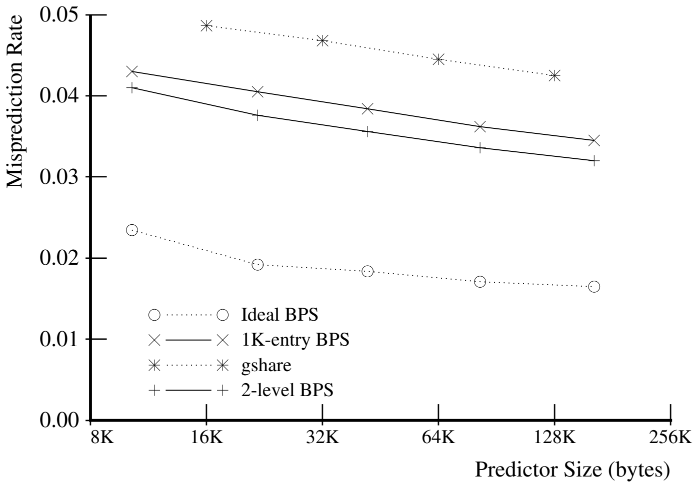


Figure 5.5: gXOR vs 2-bit counter BPS

of the 2-bit counter branch predictor selection mechanism at various implementation costs. For illustration purposes, this figure also shows the performance of hybrid branch predictors with an ideal dynamic predictor selector and that of the best single-scheme predictor, gshare. The ideal dynamic predictor selection mechanism differs from the ideal static selection mechanism described in Section 5.1. The ideal dynamic selector will at runtime always choose the component predictor that yields the correct prediction, if one exists, whereas the ideal static selector chooses the more suitable predictor for each branch at compile time. Although the gXOR mechanism outperforms the 2-bit counter predictor selection mechanism, it still operates far below its potential performance.

5.3 Summary

By combining multiple single-scheme branch predictors, hybrid branch predictors attempt to exploit the different strengths of different predictors. For this attempt to result in significant increases in prediction accuracy, the hybrid predictor must combine an appropriate set of single-scheme predictors and use an effective predictor selection mechanism. This dissertation compared various hybrid predictor implementations to determine which single-scheme predictor combinations and branch selection mechanisms were most effective.

The hybrid predictor configurations considered in the study consisted of two single-scheme predictor components. Each component came from one of four classes: static, 2bC, PAs, and gshare. For an idealized static selection mechanism, the single-scheme predictor combination that achieved the lowest average misprediction rate for the SPECint92 benchmarks was the gshare/PAs combination. Its average misprediction rate was 13% lower than that of the next most effective combination. It was able to achieve this low misprediction rate because it was able to effectively exploit inter-branch correlation with the gshare component and intra-branch correlation with the PAs component. When pattern history table contention is a factor as in benchmarks with large numbers of static branches (e.g. gcc), the best combination was gshare/static, because the static component has no tables and does not suffer from this problem. For a fixed level of hardware cost, the gshare/PAs configurations that devoted the majority of the hardware to the gshare component achieved the lowest misprediction rates.

To further improve the prediction accuracy of hybrid predictors, we introduced a new branch selection mechanism, the 2-level Branch Predictor Selector. It has four variations: gXOR, pXOR, gCONC, and pCONC. These variations use both branch history and the branch address to improve the accuracy of predictor selection. Our experiments showed that 2-level BPS outperforms 2-bit counter BPS and pCONC is more effective than pXOR, gCONC, and gXOR when significant BPST interference exists and the BPST is fully utilized. For the SPECint92 benchmarks, using the gXOR selection mechanism instead of the 2-bit counter BPS mechanism reduces the misprediction rate from 4.06% to 3.76%. For the gcc benchmark where the BPST is fully utilized, the pCONC BPS mechanism consistently outperforms the 2-bit counter

BPS mechanism because it reduces BPST interference, decreasing the misprediction rate by 5.45%.

CHAPTER 6

Interference

Chapter 4 showed that a significant number of branches in the dynamic execution stream tend to be mostly taken or mostly not-taken. Using static prediction on the strongly biased branches and dynamic prediction on the remaining branches resulted in lower contention in the pattern history table and thus more accurate predictions. In this chapter, we propose a new branch predictor that dynamically classifies branches based on their history patterns. In order to reduce contention, this predictor does not update the pattern history table for easily predictable branches. A simple predictor is used to handle these branches. To determine an effective classification of branches, we first examine the frequency of regular recurring history patterns in branches because these branches are more likely to be easily predictable by simple predictors. Based on this information, we propose an implementation of the new predictor which dynamically detects easily predictable branches. It then predicts the easily predictable branches with a simple predictor while using a two-level predictor for the other branches.

6.1 Interference Characteristics

Researchers have shown that interference in the pattern history tables can significantly degrade the performance of two-level branch predictors. The amount of PHT interference in different two-level branch predictors varies because of their different hashing schemes. Tables 6.1 and 6.2 show the amount of interference for the PAs predictor and the gshare predictor. The tables also list the amount of constructive and destructive interference for each of the two schemes.

Tables 6.1 and 6.2 show that the total amount of interference for the PAs is significantly greater than that for gshare. However, the amount of constructive interference is approximately equal to that of destructive interference for PAs, while the amount of destructive interference is significantly greater than that of constructive interference for gshare. This is because the PAs scheme indexes into the PHT using the past branch history of the same static branch. Thus, two static branches that hash into the same PHT have had the same past branch behavior and they are likely to have similar future behavior. Furthermore, most of the interference under PAs is due to strongly

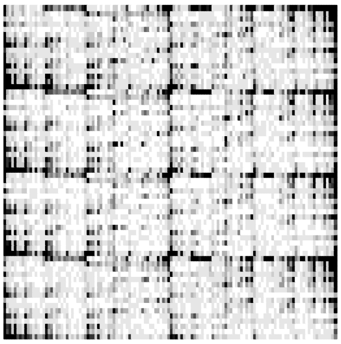
predictor size	benchmark	Amount of PHT interference		
		total	constr.	destr.
~4KB	com	2,911,783	57,071	74,018
	gcc	16,636,241	594,399	869,330
	go	12,239,623	712,916	1,741,964
	ijpeg	3,693,315	122,780	306,263
	li	13,826,569	163,305	445,815
	perl	6,897,469	36,936	170,029
~6KB	com	2,834,676	48,847	82,963
	gcc	16,531,875	633,864	837,829
	go	12,041,878	715,970	1,647,649
	ijpeg	3,645,691	114,422	276,325
	li	13,547,008	133,017	393,282
	perl	6,718,881	31,064	142,706
~10KB	com	2,737,118	25,566	73,916
	gcc	16,439,145	674,925	816,309
	go	11,835,232	722,125	1,559,105
	ijpeg	3,577,811	106,014	242,672
	li	13,278,814	110,923	347,900
	perl	6,565,341	27,753	123,358

Table 6.1: PHT interference for PAs

predictor size	benchmark	Amount of PHT interference		
		total	const.	destr.
~2KB	com	120,570	2,663	32,010
	gcc	4,284,082	129,942	1,532,657
	go	8,267,089	355,738	2,880,388
	ijpeg	979,009	36,173	264,897
	li	510,333	9,213	70,661
	perl	717,860	4,447	227,771
~4KB	com	102,521	2,137	26,947
	gcc	3,326,864	119,743	1,211,608
	go	7,088,468	316,082	2,318,796
	ijpeg	813,960	29,785	214,714
	li	367,563	7,641	47,725
	perl	461,174	2,364	143,655
~8KB	com	70,316	1,820	9,822
	gcc	2,658,880	112,123	960,944
	go	5,709,259	279,467	1,770,366
	ijpeg	625,951	21,815	155,043
	li	302,362	6,890	35,688
	perl	191,099	636	68,697

Table 6.2: PHT interference for gshare

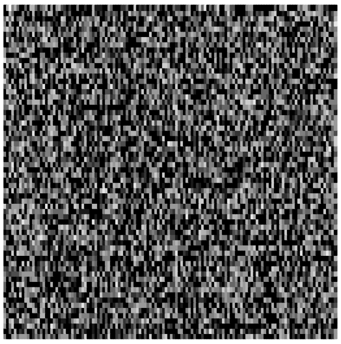
biased branches (e.g. always taken branches) because a large number of branches in the instruction stream are strongly biased. These branches tend to access the same PHT entries under the PAs scheme; thus, a significant amount of interference occurs at only a few PHT entries (as shown in Figure 6.1). Since the interference is due to strongly biased branches, most of the interference under PAs is neutral.



This figure shows the amount of interference in each entry of a 8192 entry PHT for the PAs scheme when running gcc. The black squares represent counters with 606 cases of interference or more (606 is the average amount of interference per entry). The white squares represent counters with 60 cases of interference or fewer.

Figure 6.1: Interference in PAs' PHTs

On the other hand, there is a significantly greater amount of destructive interference than constructive interference for the gshare scheme. Branches whose past behavior are either all 0's or all 1's may use various entries in the PHT because the gshare scheme selects the PHT entry by XORing the global branch history with the branch address. Due to this hashing scheme, branches that share the same PHT entries may not have similar past behavior, resulting in significant amount of destructive interference. Unlike the PAs scheme, the interference for gshare is distributed across all PHT entries as shown in Figure 6.2.



This figure shows the amount of interference in each entry of a 8192 entry PHT for the gshare scheme when running gcc. The black squares represent counters with 325 cases of interference or more (325 is the average amount of interference per entry). The white squares represent counters with 32 cases of interference or fewer.

Figure 6.2: Interference in gshare's PHTs

6.2 History Pattern Characteristics

Dynamic predictors use past branch history to identify recurring execution patterns in order to make accurate predictions.

In order to capture the behavior of branches with complex recurring patterns, a large amount of branch history information may have to be examined. For this reason, a sophisticated predictor, such as the two-level branch predictor, which examines more branch history information, is able to detect more branch execution patterns and thus outperforms simpler predictors.

On the other hand, even a less sophisticated predictor should be able to capture the dynamic behavior of branches with simple recurring patterns. Thus, we propose to reduce the amount of PHT interference by using a simple predictor for predicting these easily predictable branches and inhibiting the pattern history table update for these branches. This technique eliminates the interference between predictable branches and harder-to-predict branches.

In this section, we will try to determine a set of branches that can be easily identified and accurately predicted by simple predictors and whose removal can significantly reduce the negative impact of PHT interference.

6.2.1 Frequency of Recurring Patterns

A branch may be easily predictable if it follows a regular repeating pattern of execution. A branch is said to have a repeating pattern of length k if the branch's outcome on the ith time, d_i, is identical to the outcome on the $(i+k)$th time, d_{i+k}. Furthermore, $< d_i, d_{i+1}, ..., d_{i+k} >$ must be the shortest repeating pattern; i.e. there can not be any repeating pattern within $< d_i, ...d_{i+k} >$. For example, a branch that is always taken or always not-taken will have a repeating pattern of length 1. A branch that is taken every other time has a repeating pattern of length 2.

To measure how often repeating execution patterns occur, we record the history of each branch in the branch target buffer(BTB). In this experiment, we use a 16-bit branch history register to hold the outcomes of the 16 most recent occurrences of the branch. A pattern is detected only if it repeats for the entire length of the history register. In addition, the pattern must repeat at least once in the history register; i.e., a repeating pattern in the 16-bit branch history register can be of at most length 8.

Figure 6.3 shows what fraction of the time patterns of different lengths were detected when using a 1024 and a 8192 entry BTB respectively. The 8192 entry BTB was used to approximate an infinite size BTB[1]. With the 8192 entry BTB, a repeating pattern of length one was detected for approximately 55 percent of the branches in the instruction stream, indicating that over half of the PHT accesses can be avoided if we inhibit updates to the PHT for these branches. With a smaller 1024 entry BTB, fewer repeating patterns were detected because of more BTB misses.

[1]For the SPEC95 integer benchmarks with the exception of gcc, only an insignificant number of BTB misses occurred when the 8192 entry BTB was used.

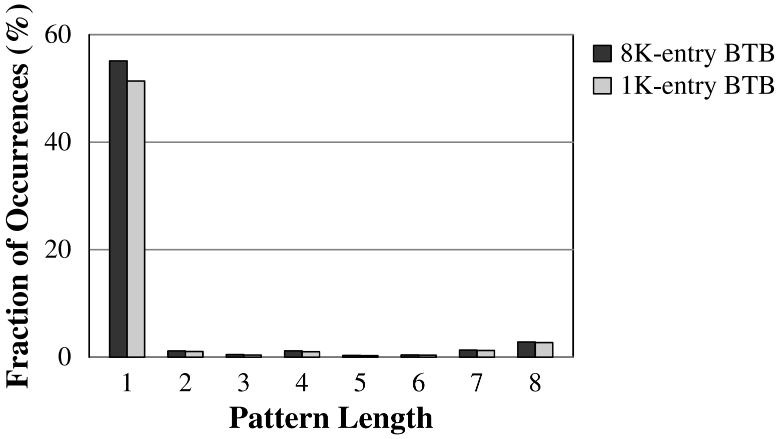


Figure 6.3: Frequency of pattern

That is, when an entry is displaced from the BTB, the associated branch history is lost. This branch must occur 16 more times before its history register is filled with enough information for detecting repeating patterns. However, even with the smaller 1K entry BTB, which we will use for the remaining experiments in the dissertation, a repeating pattern length of one was still detected over 50 percent of the time.

6.2.2 Accuracy of Static PHT

Inhibiting PHT updates for certain branches can improve performance by reducing the negative effects of PHT interference. However, predicting these branches using simpler predictors may result in lower accuracy. Thus, to achieve higher performance, the disadvantage of having less accurate predictions for these branches must not outweigh the benefits of reducing the PHT interference.

For branches that have a recurring pattern, the next outcome of the branch is an extension of this pattern. Since the PSg(algo) [31] works on this premise, we expect it to perform well for branches with repeating patterns and apply it to these branches. Figure 6.4 compares the performance of PSg(algo) to that of gshare. Branches with

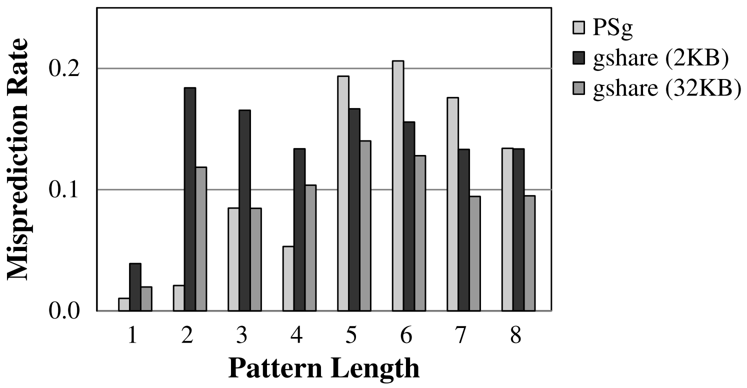


Figure 6.4: Performance of PSg and gshare on branches with repeating patterns

short repeating patterns were accurately predicted with PSg(algo), indicating it could be advantageous to predict them with a simple predictor and inhibit the PHT update.

6.2.3 Characteristics of PHT Interference

PHT interference can have positive, negative, or no effect on the performance of two-level branch predictors. The performance of two-level branch predictors can be improved if inhibiting the PHT updates for easily-predictable branches reduces the negative impact of interference.

Tables 6.3 and 6.4 show the amount of PHT interference for the PAs predictor and the gshare predictor due to branches with repeating history patterns. That is, interference is included in this table when the current branch and/or the branch that last referenced the same PHT entry have a repeating history pattern. These tables also list the amount of constructive and destructive interference. Inhibiting PHT updates can improve predictor performance if the reduction in the amount of destructive interference is greater than that of constructive interference.

As shown in Section 6.1, the amount of PHT interference in different two-level branch predictors varies because of their different hashing schemes. Thus the amount of interference due to easily predictable branches also varies among different two-

		Amount of PHT interference		
predictor size	benchmark	total	constr.	destr.
~4KB	com	2,587,226	31,690	24,778
	gcc	8,409,808	56,219	63,506
	go	2,116,261	47,094	82,675
	ijpeg	2,163,734	10,636	47,464
	li	11,511,671	23,695	39,371
	perl	5,879,042	15,518	42,836
~6KB	com	2,526,494	26,499	36,493
	gcc	8,202,007	56,397	58,241
	go	2,004,582	44,309	71,173
	ijpeg	2,083,745	8,572	34,940
	li	11,373,134	21,329	35,328
	perl	5,782,630	11,566	31,109
~10KB	com	2,445,425	4,021	31,190
	gcc	8,005,498	55,984	61,485
	go	1,900,640	42,399	64,098
	ijpeg	2,022,661	7,003	30,600
	li	11,246,232	19,978	30,566
	perl	5,702,299	8,813	20,396

Table 6.3: PHT interference due to branches with repeating history patterns for PAs

predictor size	benchmark	Amount of PHT interference total	const.	destr.
~2KB	com	110,076	1,831	27,701
	gcc	2,626,637	69,008	920,628
	go	2,070,520	73,492	724,863
	ijpeg	727,585	21,968	199,668
	li	474,728	8,243	64,362
	perl	636,471	2,442	207,310
~4KB	com	94,649	1,601	23,385
	gcc	1,990,704	64,344	720,700
	go	1,806,605	67,153	600,030
	ijpeg	600,631	17,929	160,500
	li	348,563	7,026	43,522
	perl	412,114	1,890	133,905
~8KB	com	68,845	1,643	9,351
	gcc	1,577,203	61,458	573,345
	go	1,491,355	61,412	469,789
	ijpeg	451,869	12,710	112,633
	li	285,947	6,299	31,583
	perl	179,498	426	66,054

Table 6.4: PHT interference due to branches with repeating history patterns for gshare

level branch predictors. Similar to the distribution of the overall interference, the total amount of interference due to easily predictable branches for the PAs is significantly greater than that for gshare, and the amount of constructive interference is approximately equal to that of destructive interference for PAs, while the amount of destructive interference is significantly greater than that of constructive interference for gshare. Since over 50% of branches have repeating history patterns of length 1, these branches tend to access the same PHT entries under the PAs scheme; thus, a significant amount of interference occurs due to these branches. And since these branches have similar dynamic behaviors, most of the interference under PAs is neutral. Thus, removing this sort of interference has little effect on the performance of the PAs predictor.

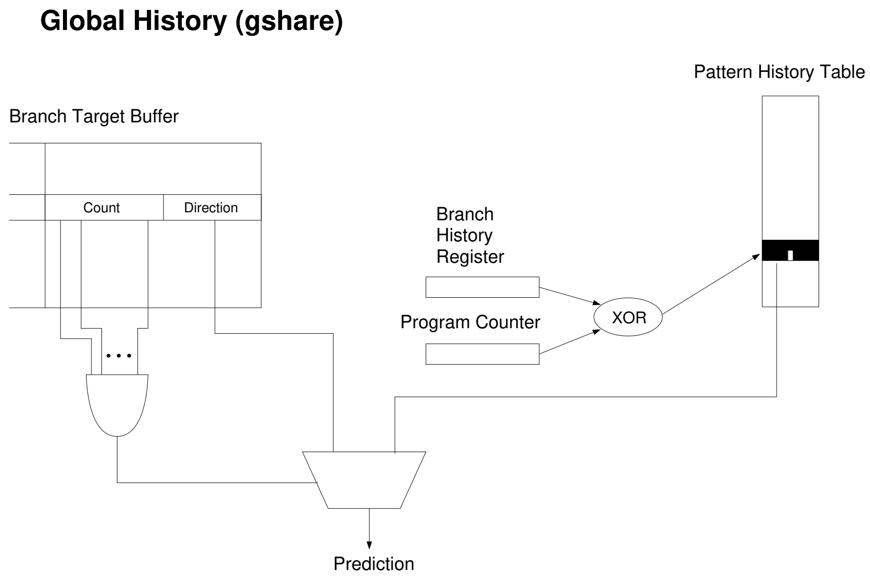
For the gshare scheme, there is a significantly greater amount of destructive interference than constructive interference due to easily predictable branches. Having significantly more destructive interference than constructive interference indicates that inhibiting the PHT updates for easily-predictable branches can reduce the negative effects of PHT interference for the gshare variation of the two-level branch predictor. In the following studies, we will examine the performance impact of the filtering mechanism on the gshare scheme.

6.3 Predictor Model

As shown in Section 6.2, branches with a repeating history pattern of length one occur frequently in the dynamic instruction stream and these branches can be accurately predicted by a PSg predictor. In addition, Section 6.2 showed that a significant amount of destructive interference may be removed if the PHTs are not updated for these branches. Therefore, to improve predictor performance, we add a filtering mechanism which records the dynamic history of each branch and dynamically separates always taken and always not-taken branches from the other branches; the appropriate operations can then be applied to each branch based on its classification.

To be able to dynamically identify branches with repeating patterns of length one, we add a direction bit and a counter to each BTB entry. Figure 6.5 shows the structure of the gshare scheme with the filtering mechanism; the concept extends easily to other global variations of the two-level predictor. Figure 6.6 shows the flow chart of the workings of the filtering algorithm. Essentially, a branch's counter records the number of consecutive occurrences in which the branch goes in the same direction. The direction bit records that direction. If the counter is not at its maximum value, the prediction for the branch is made by the default predictor. When the counter reaches its maximum, the direction bit supplies the prediction for the branch and the pattern history table is not updated.

The filtering mechanism is implemented in slightly different ways for per-address and global predictors. For the per-address predictor, we already maintain the branch history of each branch for selection of the PHT entry, so the counters mentioned above are not needed. The branch history can be used to determine whether the outcomes of a branch are either always taken or always not-taken (as shown in Figure 6.7).



Although this figure only shows the gshare implementation, a similar structure can be implemented for all global history variations of the two-level branch predictor.

Figure 6.5: Structure of the filtering mechanism with gshare

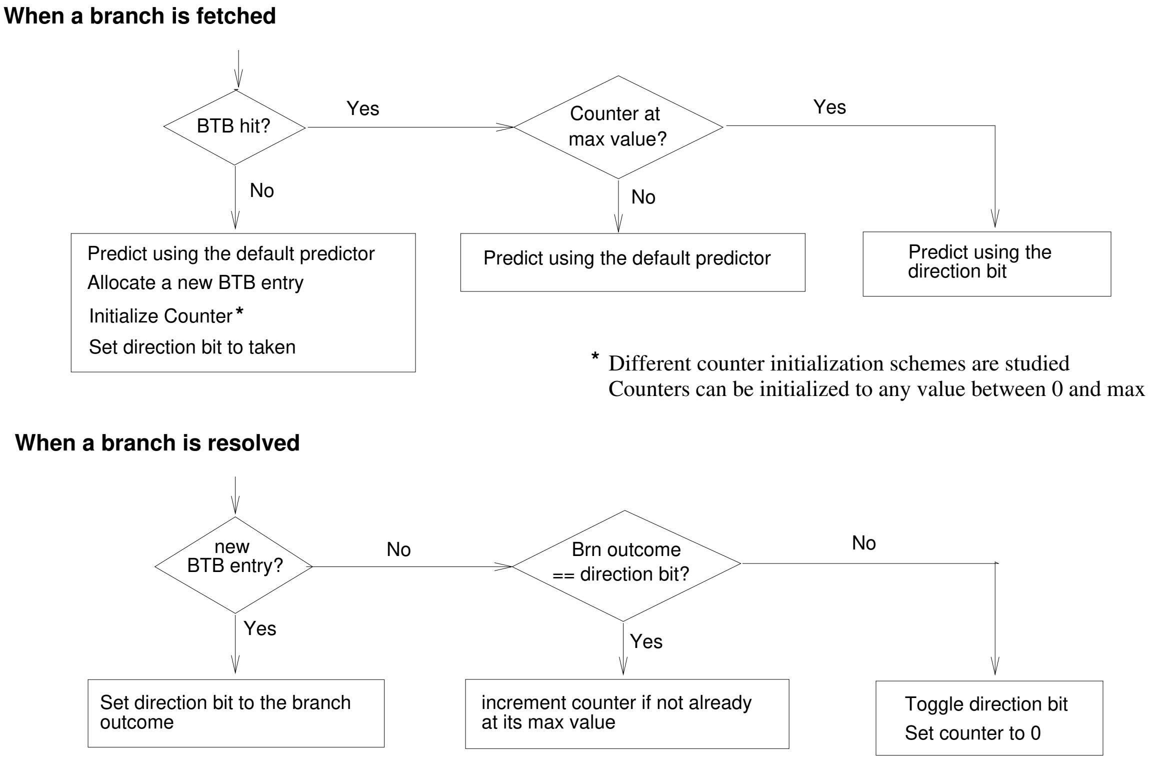


Figure 6.6: Flow chart of the filtering algorithm

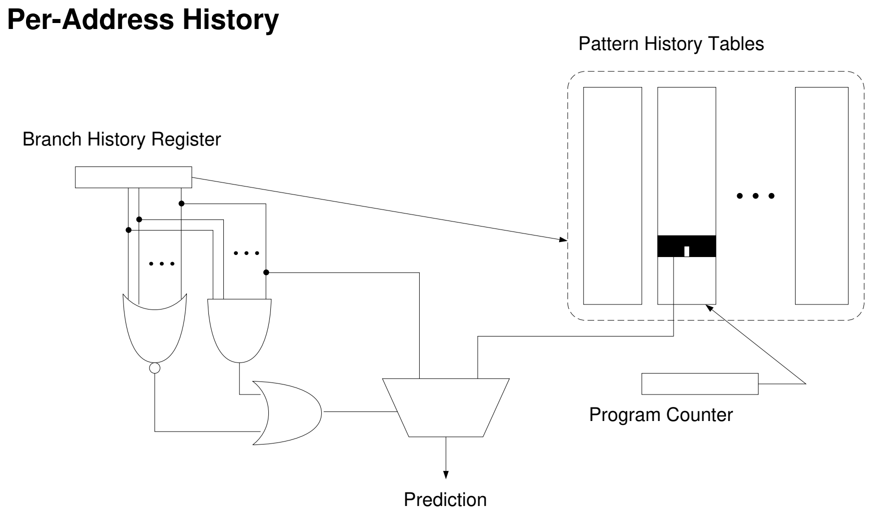


Figure 6.7: Structure of the filtering mechanism with PAs

6.4 Experimental Results

6.4.1 Filtering Mechanism Configurations

The performance of the filtering mechanism depends on the size of counters used to detect recurring patterns and on the initial value assigned to these counters.

Performance vs Counter Initialization Value

When a new BTB entry is allocated for a branch, the counter for detecting repeating history patterns needs to be initialized. The performance of the filtering mechanism depends on the initial value assigned to these counters.

Figure 6.8 shows the performance of several configurations of the filtering mechanism when the counters are initialized to their minimum value or to their maximum value. The branch target buffer used in our experiments has 1K entries and is 4-way set associative. The maximum counter value in this experiment is 15. As shown in Figure 6.8, the filtering mechanism can significantly reduce a greater number of mispredictions of gshare when its counters are initialized to 15 instead of 0. Table 6.5 shows the performance of these different configurations on the individual benchmarks. Each value in this table indicates the prediction accuracy achieved by the gshare predictor with the filtering mechanism. For the gcc and go benchmarks, the performance of the filtering mechanism improves when the counters are initialized to a larger value. For example, for the gcc benchmark, by initializing the counters to 15 instead of 0, the filtering mechanism reduces the misprediction rate of a 2KByte gshare by 38.0% instead of 24.5%. For the go benchmark, by initializing the counters to 15 instead of 0, the filtering mechanism reduces the misprediction rate by 23.2% instead of 8.1%.

Since the filtering mechanism is trying to detect branches whose recent outcomes are always taken or always not-taken, assigning a larger initial value to a counter has the advantage that the counter will reach its maximum value in a shorter period

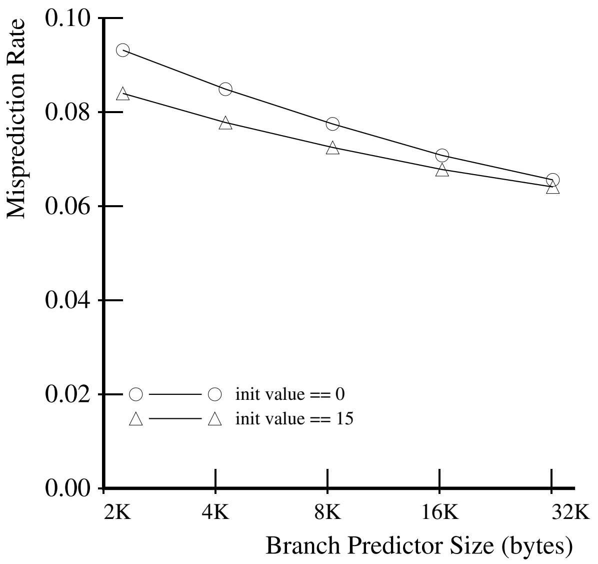


Figure 6.8: Performance vs. counter initialization value

benchmark	init value	Predictor Size 2KB	4KB	8KB	16KB	32KB
com	0	91.68%	91.84%	92.02%	92.24%	92.32%
	13	91.68%	91.84%	92.02%	92.24%	92.32%
	14	91.68%	91.84%	92.02%	92.24%	92.32%
	15	91.68%	91.85%	92.02%	92.25%	92.32%
gcc	0	91.27%	92.10%	92.94%	93.77%	94.56%
	13	92.34%	93.04%	93.75%	94.42%	95.04%
	14	92.55%	93.21%	93.88%	94.52%	95.09%
	15	92.83%	93.42%	94.02%	94.58%	95.08%
go	0	76.59%	79.78%	82.77%	85.25%	87.20%
	13	79.08%	81.72%	84.17%	86.17%	87.75%
	14	79.68%	82.14%	84.42%	86.27%	87.73%
	15	80.45%	82.64%	84.63%	86.26%	87.54%
ijpeg	0	92.12%	92.47%	92.65%	92.80%	92.95%
	13	92.17%	92.50%	92.68%	92.82%	92.96%
	14	92.19%	92.51%	92.68%	92.82%	92.96%
	15	92.21%	92.52%	92.69%	92.82%	92.96%
li	0	94.93%	95.08%	95.19%	95.36%	95.48%
	13	94.93%	95.08%	95.20%	95.36%	95.49%
	14	94.94%	95.08%	95.20%	95.36%	95.49%
	15	94.94%	95.08%	95.20%	95.36%	95.49%
perl	0	97.47%	97.82%	97.96%	98.08%	98.14%
	13	97.48%	97.82%	97.96%	98.09%	98.15%
	14	97.49%	97.83%	97.97%	98.09%	98.15%
	15	97.49%	97.84%	97.96%	98.08%	98.13%

Table 6.5: Performance vs. counter initialization value

of time; the shorter warm-up time results in fewer PHT updates and thus less PHT interference. Table 6.6 shows the percentage of predictions made by the BTB direction bit for different classes of branches. Branches in a program are partitioned into several classes based on their dynamic taken rates as shown in Table 6.7. Since strongly biased branches can be accurately predicted by a simple predictor, the performance of the filtering mechanism can be improved if more of these branches are predicted with the BTB direction bit to reduce PHT interference. For gcc and go, when the counters are initialized to 0, the filtering mechanism fails to detect repeating patterns for a significant number of strongly biased branches; e.g. only 71.61% and 21.22% of the always not-taken branches in gcc and go were filtered. Initializing the counters to their maximum allows the filtering mechanism to filter significantly more of the strongly biased branches; e.g. 93.06% and 79.22% of the always not-taken branches in gcc and go are now filtered.

For the other benchmarks, the performance of the filtering mechanism is not significantly affected by the counter initial value because there are few static branches exercised in these programs. With a 1K-entry 4-way set-associative BTB, BTB misses

bench.	init value	Branch Class BC1	BC2	BC3	BC4	BC5	BC6	BC7	BC8
com	0	99.91%	99.67%	42.89%	0.74%	1.08%	0.00%	98.78%	99.93%
	13	99.97%	99.68%	42.90%	0.74%	1.08%	0.01%	98.80%	99.98%
	14	99.98%	99.68%	42.90%	0.74%	1.09%	0.01%	98.80%	99.98%
	15	99.99%	99.68%	42.90%	0.74%	1.09%	0.01%	98.80%	99.99%
gcc	0	71.61%	71.21%	39.11%	10.60%	16.63%	32.26%	77.75%	71.26%
	13	86.38%	80.48%	47.89%	14.14%	21.31%	40.70%	83.29%	86.84%
	14	89.12%	81.90%	49.57%	15.40%	22.62%	42.51%	84.07%	89.58%
	15	93.06%	83.82%	52.11%	17.84%	24.91%	45.00%	85.11%	93.51%
go	0	21.22%	37.81%	19.42%	7.10%	8.41%	37.05%	59.00%	41.52%
	13	58.26%	68.47%	43.04%	14.46%	13.19%	50.71%	74.43%	70.76%
	14	67.08%	74.26%	49.09%	17.50%	15.42%	52.89%	77.06%	77.17%
	15	79.22%	81.66%	57.55%	23.23%	19.01%	55.54%	80.27%	85.75%
ijpeg	0	96.27%	93.10%	20.50%	45.67%	23.51%	29.55%	82.49%	94.08%
	13	98.60%	93.50%	33.14%	45.77%	23.60%	37.14%	82.63%	97.81%
	14	98.94%	93.54%	35.02%	45.80%	23.65%	37.79%	82.64%	98.33%
	15	99.36%	93.57%	37.24%	45.88%	23.71%	38.50%	82.66%	98.96%
li	0	99.96%	91.15%	61.73%	12.49%	19.65%	27.83%	88.07%	99.96%
	13	99.99%	91.16%	61.73%	12.50%	19.66%	27.88%	88.08%	99.99%
	14	99.99%	91.16%	61.73%	12.50%	19.66%	27.89%	88.08%	99.99%
	15	100.00%	91.16%	61.73%	12.50%	19.66%	27.91%	88.08%	100.00%
perl	0	96.95%	85.61%	37.39%	11.51%	2.05%	33.20%	84.99%	96.67%
	13	98.94%	85.66%	37.46%	11.73%	2.24%	33.24%	85.08%	99.16%
	14	99.21%	85.67%	37.47%	11.88%	2.34%	33.24%	85.09%	99.41%
	15	99.59%	85.82%	37.49%	12.11%	2.71%	33.25%	85.11%	99.67%

Table 6.6: Initialization value vs. fraction of direction bit usage for each branch class

Classes	Descriptions
BC1	pr(br) = 0%
BC2	0% < pr(br) <= 5%
BC3	5% < pr(br) <= 10%
BC4	10% < pr(br) <= 50%
BC5	50% < pr(br) <= 90%
BC6	90% < pr(br) <= 95%
BC7	95% < pr(br) < 100%
BC8	pr(br) = 100%

Table 6.7: Branch classes

occur infrequently; thus, the counters rarely need to be reinitialized.

In the following sections, we will examine filtering mechanisms where their counters are initialized to their maximum value.

Performance vs Counter Size

The performance of the filtering mechanism also depends on the size of the counters used for identifying repeating patterns.

Table 6.8 shows the performance of several configurations of the filtering mechanism where the size of the counters is varied. Table 6.8 shows that for the gcc and go benchmarks, the performance of the filtering mechanism continues to improve slightly as the size of the counters becomes smaller. This is because there is a significant amount of destructive PHT interference in gcc and go. Basing its decision

		Predictor Size				
benchmark	max count	2KB	4KB	8KB	16KB	32KB
com	4	90.55%	90.68%	90.82%	90.97%	91.02%
	8	91.01%	91.17%	91.34%	91.57%	91.64%
	12	91.23%	91.40%	91.57%	91.80%	91.87%
	16	91.68%	91.85%	92.02%	92.25%	92.32%
	20	91.68%	91.85%	92.02%	92.25%	92.32%
	24	91.68%	91.85%	92.03%	92.25%	92.32%
gcc	4	92.66%	93.14%	93.63%	94.07%	94.43%
	8	92.92%	93.47%	94.02%	94.54%	94.99%
	12	92.87%	93.44%	94.03%	94.57%	95.05%
	16	92.83%	93.42%	94.02%	94.58%	95.08%
	20	92.78%	93.39%	94.00%	94.57%	95.08%
	24	92.75%	93.36%	93.98%	94.56%	95.08%
go	4	81.54%	83.27%	84.80%	86.02%	86.96%
	8	80.96%	83.02%	84.88%	86.39%	87.56%
	12	80.65%	82.80%	84.75%	86.34%	87.59%
	16	80.45%	82.64%	84.63%	86.26%	87.54%
	20	80.31%	82.53%	84.54%	86.20%	87.50%
	24	80.22%	82.45%	84.48%	86.15%	87.47%
ijpeg	4	91.91%	92.05%	92.15%	92.24%	92.33%
	8	92.11%	92.41%	92.56%	92.67%	92.80%
	12	92.22%	92.53%	92.69%	92.82%	92.96%
	16	92.21%	92.52%	92.69%	92.82%	92.96%
	20	92.20%	92.52%	92.68%	92.82%	92.96%
	24	92.21%	92.53%	92.70%	92.84%	92.98%
li	4	94.18%	94.28%	94.34%	94.45%	94.48%
	8	94.81%	94.96%	95.06%	95.20%	95.31%
	12	94.92%	95.07%	95.19%	95.33%	95.45%
	16	94.94%	95.08%	95.20%	95.36%	95.49%
	20	94.97%	95.12%	95.23%	95.39%	95.52%
	24	94.99%	95.14%	95.25%	95.42%	95.55%
perl	4	96.20%	96.38%	96.49%	96.62%	96.67%
	8	97.14%	97.40%	97.57%	97.69%	97.72%
	12	97.34%	97.67%	97.80%	97.92%	97.97%
	16	97.49%	97.84%	97.96%	98.08%	98.13%
	20	97.58%	97.93%	98.05%	98.17%	98.22%
	24	97.59%	97.94%	98.07%	98.19%	98.23%

Table 6.8: Performance vs. counter size

on smaller maximum counts, the filtering mechanism will detect regularly repeating patterns for more branches. For example, with a loop that always executes for 10 iterations, the history pattern of the loop-ending branch would be a repeating pattern of 1111111110. Detecting patterns using the maximum count of 4, the filtering mechanism will use the BTB direction bit to predict this branch 6 out of 10 occurrences. Detecting patterns using the maximum count of 8, the filtering mechanism will use the BTB direction bit only 20% of the time. As shown in Table 6.9, changing the

bench.	max count	Branch Class							
		BC1	BC2	BC3	BC4	BC5	BC6	BC7	BC8
com	4	99.99%	99.92%	75.85%	18.17%	15.83%	68.56%	99.67%	99.99%
	8	99.99%	99.84%	61.26%	2.48%	2.53%	37.13%	99.38%	99.99%
	12	99.99%	99.76%	51.03%	1.24%	1.52%	7.64%	99.09%	99.99%
	16	99.99%	99.68%	42.90%	0.74%	1.09%	0.01%	98.80%	99.99%
	20	99.99%	99.60%	36.43%	0.49%	0.84%	0.00%	98.52%	99.99%
	24	99.99%	99.52%	31.37%	0.34%	0.69%	0.00%	98.24%	99.99%
gcc	4	93.06%	92.40%	77.03%	37.54%	48.08%	75.46%	94.32%	93.51%
	8	93.06%	88.95%	65.59%	24.99%	34.05%	61.63%	90.92%	93.51%
	12	93.06%	86.17%	57.86%	20.35%	28.25%	52.12%	87.90%	93.51%
	16	93.06%	83.82%	52.11%	17.84%	24.91%	45.00%	85.11%	93.51%
	20	93.06%	81.75%	47.75%	16.33%	22.92%	40.26%	82.52%	93.51%
	24	93.06%	79.91%	44.38%	15.34%	21.50%	36.61%	80.06%	93.51%
go	4	79.22%	85.09%	71.25%	40.81%	39.90%	77.63%	89.41%	85.75%
	8	79.22%	83.55%	64.23%	28.98%	26.70%	66.97%	85.34%	85.75%
	12	79.22%	82.50%	60.31%	25.19%	21.63%	60.21%	82.51%	85.75%
	16	79.22%	81.66%	57.55%	23.23%	19.01%	55.54%	80.27%	85.75%
	20	79.22%	80.93%	55.74%	22.11%	17.45%	52.08%	78.44%	85.75%
	24	79.22%	80.28%	54.44%	21.41%	16.47%	49.46%	76.92%	85.75%
ijpeg	4	99.36%	98.24%	73.22%	67.03%	49.76%	82.87%	95.38%	98.96%
	8	99.36%	96.60%	53.79%	56.02%	32.16%	66.97%	91.06%	98.96%
	12	99.36%	95.02%	43.80%	50.51%	26.94%	52.16%	86.82%	98.96%
	16	99.36%	93.57%	37.24%	45.88%	23.71%	38.50%	82.66%	98.96%
	20	99.36%	92.14%	30.70%	41.66%	21.11%	36.47%	78.70%	98.96%
	24	99.36%	90.94%	24.65%	37.60%	18.85%	34.75%	74.75%	98.96%
li	4	100.00%	97.48%	82.77%	30.94%	37.64%	78.26%	95.26%	100.00%
	8	100.00%	95.23%	72.47%	18.94%	26.68%	58.72%	92.28%	100.00%
	12	100.00%	93.11%	66.35%	14.88%	22.27%	41.14%	90.05%	100.00%
	16	100.00%	91.16%	61.73%	12.50%	19.66%	27.91%	88.08%	100.00%
	20	100.00%	89.29%	57.99%	10.74%	17.86%	21.43%	86.27%	100.00%
	24	100.00%	87.64%	55.17%	9.41%	16.42%	16.37%	84.55%	100.00%
perl	4	99.59%	95.59%	78.72%	40.44%	37.65%	73.89%	96.06%	99.67%
	8	99.59%	91.93%	61.34%	23.94%	10.26%	57.81%	92.33%	99.67%
	12	99.59%	88.58%	48.32%	16.86%	4.59%	43.50%	88.70%	99.67%
	16	99.59%	85.82%	37.49%	12.11%	2.71%	33.25%	85.11%	99.67%
	20	99.59%	83.54%	27.91%	9.47%	1.79%	26.28%	81.54%	99.67%
	24	99.59%	81.70%	21.84%	7.77%	1.40%	20.27%	78.42%	99.67%

Table 6.9: Counter size vs. fraction of direction bit usage for each branch class

counter size causes the filtering mechanism to filter a different number of unbiased branches. With the counters initialized to their maximum, the filtering mechanism already detects repeating patterns for most of the strongly biased branches. Since the behavior of unbiased branches is harder to predict, the dynamic predictor outperforms the BTB direction bits for these branches. Thus, the performance of the filtering mechanism improves with smaller counters because there is a significant amount of PHT interference. The benefits of reducing the PHT interference is greater than the disadvantages of using a less accurate branch predictor.

For counters with a maximum count of 4 (i.e. maximum value of 3), too many unbiased branches are prematurely predicted using the simple predictor instead of the more accurate dynamic predictor. The benefits of reducing the PHT interference no longer outweigh the disadvantages of having less accurate predictions for these branches. Similarly with a large predictor of 64Kbyte where a smaller amount of PHT interference occurs, larger counters achieve higher prediction accuracy.

For the other benchmarks, the performance of the predictor improves as the size of the counters increases. Since there is only a small amount of PHT interference for these benchmarks, the benefit of choosing the more accurate predictor outweighs the benefit of reducing PHT interference.

For the following experiments, we will use counters with a maximum count of 16 for the filtering mechanism. This configuration resulted in the best average predictor performance for the benchmarks.

6.4.2 Predictor Performance

Branches with a repeating pattern length of one occur frequently as shown in Section 6.2.1, so filtering away these branches from the PHT can significantly reduce the amount of interference. Table 6.10 shows the amount of PHT interference after filtering out these branches. The reduction in the amount of PHT interference after filtering depends on the particular benchmark, from a maximum of 93.1% for the li benchmark to a minimum of 39.0% for the go benchmark (cf. Table 6.2).

Figure 6.9 shows the performance of gshare with and without the filtering mechanism. Figure 6.10 shows that for the gcc benchmark, where we saw a large amount of destructive interference, the filtering mechanism is able to significantly improve the performance of gshare. As shown in section 6.4.1, the performance of gshare on gcc can still be improved by using smaller counters for the filtering mechanism.

At lower implementation costs, where a larger amount of PHT interference occurs, our scheme is able to remove more interference and thus achieve higher performance. For larger predictors where the effect of PHT interference is less significant, the results of the two predictors are closer.

Figure 6.11 shows the performance of the 2 KByte gshare schemes on the individual benchmarks. The filtering mechanism is able to improve the performance of these benchmarks. For benchmarks that have a large number of static branches in the dynamic instruction stream, such as gcc, a large amount of PHT interference occurs. The filtering mechanism was able to remove a significant amount of the destructive interference (see Table 6.2), reducing the number of mispredictions by 30% to 38%

predictor size	benchmarks	Amount of PHT interference		
		total	constr.	destr.
~2KB	com	15,816	1,106	6,225
	gcc	1,683,558	73,978	627,319
	go	5,043,994	256,506	1,661,338
	ijpeg	389,086	20,072	95,720
	li	46,126	1,132	9,273
	perl	115,785	4,258	33,305
~4KB	com	12,260	616	5,255
	gcc	1,400,030	67,328	515,074
	go	4,200,579	220,357	1,317,971
	ijpeg	311,146	15,835	74,897
	li	25,515	676	5,965
	perl	48,185	1,435	13,838
~8KB	com	2,133	333	576
	gcc	1,135,745	59,605	404,875
	go	3,326,209	188,097	998,730
	ijpeg	248,734	12,196	57,267
	li	22,124	688	5,844
	perl	9,991	141	4,622

Table 6.10: PHT interference for gshare when PHT is not updated for branches whose per-address branch histories are either always taken or always not-taken

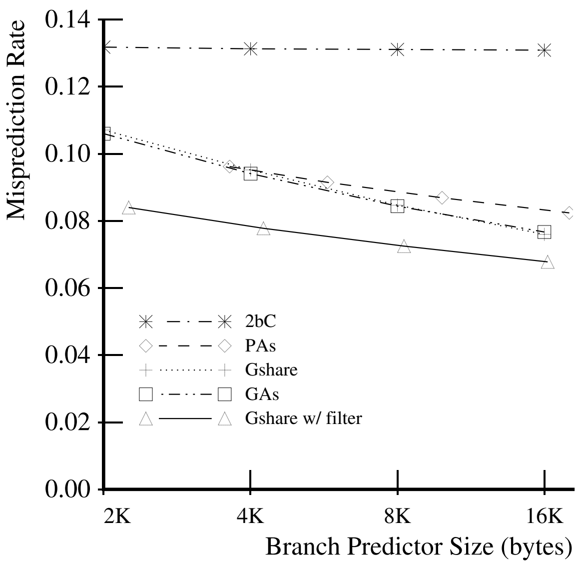


Figure 6.9: Performance impact of the filtering mechanism on gshare on SPECint95

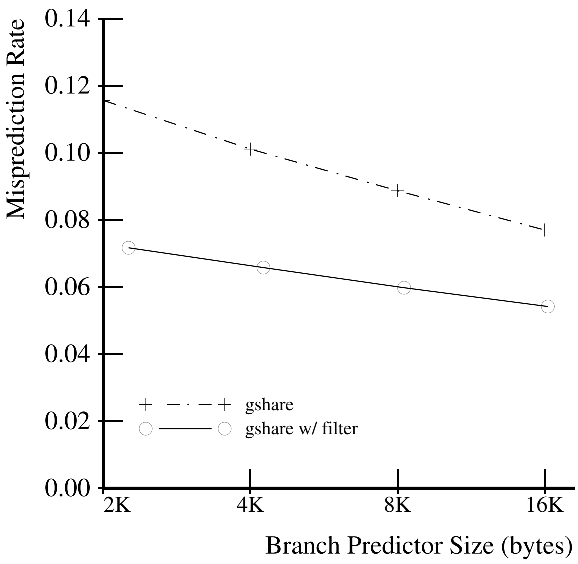


Figure 6.10: Performance impact of the filtering mechanism on gshare (gcc)

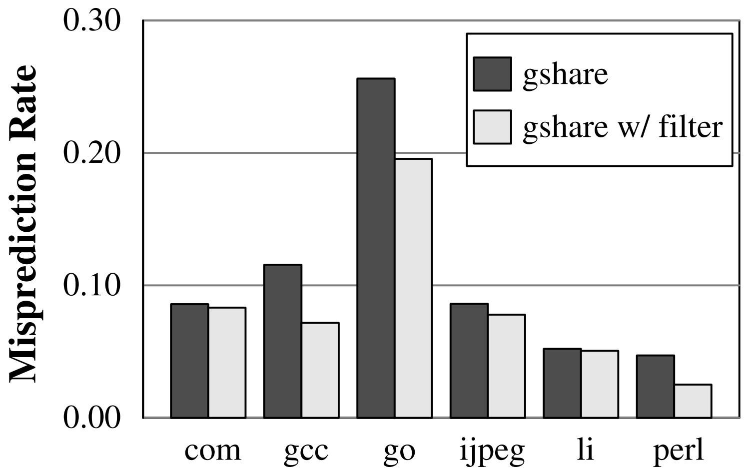


Figure 6.11: Performance impact of the filtering mechanism on a 2 KByte gshare

for the gcc benchmark when using gshare.

Figure 6.12 shows the performance of the gshare/pshare hybrid branch predictor with and without the filtering mechanism. This hybrid branch predictor combines the gshare and pshare predictors; pshare is a variation of the two-level branch predictor where the per-address branch history is XORed with the branch address for selecting the appropriate PHT entry. Since gshare/pshare uses gshare for one of its

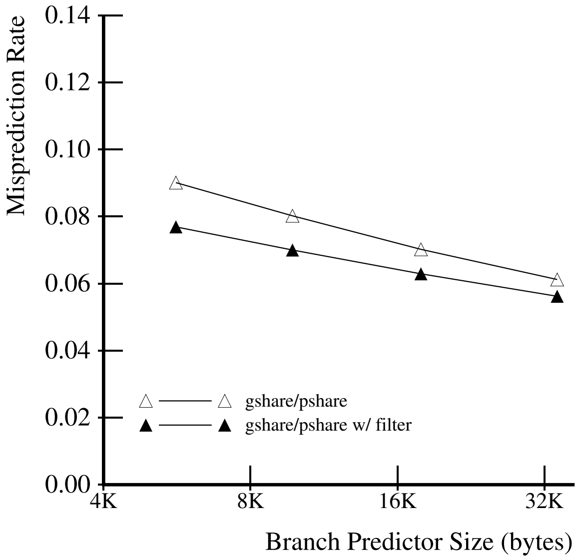


Figure 6.12: Performance impact of the filtering mechanism on gshare/pshare on SPECint95

component predictors, the filtering mechanism is also able to significantly improve the performance of this hybrid branch predictor.

Figure 6.13 shows that for the gcc benchmark, where a large amount of destructive interference exists, the filtering mechanism is able to significantly improve the

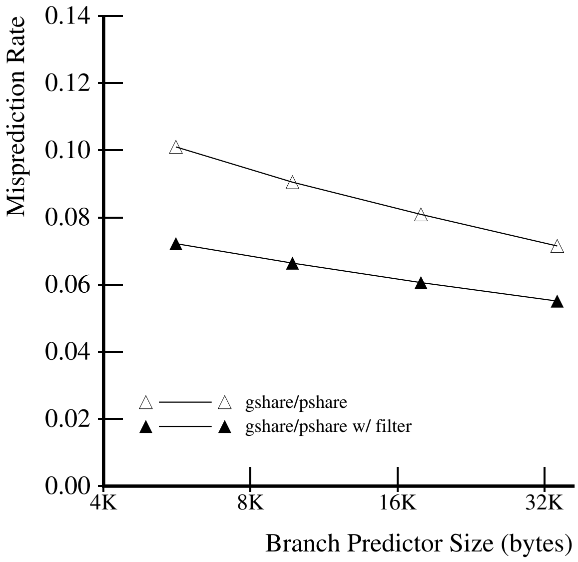


Figure 6.13: Performance impact of the filtering mechanism on gshare/pshare (gcc)

performance of gshare/pshare, reducing the number of mispredictions by 23% to 29%.

6.5 Summary

In this chapter, we have introduced a method of reducing pattern history table interference by dynamically classifying branches as strongly biased or mixed-directional. Our experiments showed that about half of the branches in the dynamic instruction stream are strongly biased, and can therefore be handled with a simple predictor.

Inhibiting the update to the pattern history table for the strongly biased branches, we were able to get a considerable decrease in the amount of destructive PHT interference with a resulting improvement in the accuracy of the gshare two-level branch predictor. For six of the SPECint95 benchmarks, we achieved an average 21.6% reduction of mispredictions for a 2 KByte gshare predictor. The filtering mechanism also significantly improved the performance of the gshare/pshare hybrid branch predictor which uses gshare for one of its component predictor. For a 6KByte gshare/pshare, the number of mispredictions was reduced by 14.7%. However, for larger predictors where the effect of PHT interference is less significant, the benefit of the filtering mechanism is also less significant. For a 32 KByte gshare predictor, the number of mispredictions was reduced by 7%.

For the gcc benchmark which executes a large number of static branches, the filtering mechanism was able to remove a significant amount of the destructive interference. We therefore reduce the number of mispredictions by 30% to 38% for the gshare predictor. For gshare/pshare, the number of mispredictions was reduced by 23% to 29%.

CHAPTER 7

Indirect Branches

7.1 Characteristics

In the past, branch prediction research has focused on accurately predicting conditional and unconditional direct branches [32, 21, 40, 20, 7, 24]. To predict such branches, the prediction mechanism predicts the branch direction (for unconditional branches, this part is trivial) and then generates the target associated with that direction. To generate target addresses, a branch target buffer (BTB) is used. The BTB stores the last target seen for each of the two possible branch directions. However, BTB-based prediction schemes perform poorly for indirect branches. Because the target of an indirect branch can change with every dynamic instance of that branch, always using the target of the previous instance as the predicted target will lead to poor prediction accuracy. Figures 7.1 through 7.5 show the number of different dynamic targets seen for each indirect branch in the SPECint95 benchmarks.

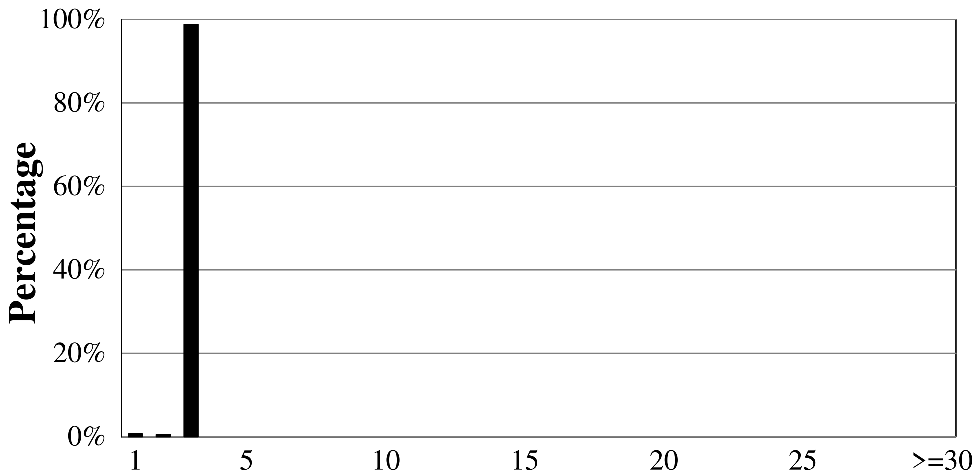


Figure 7.1: Number of Targets per Indirect Jump (compress)

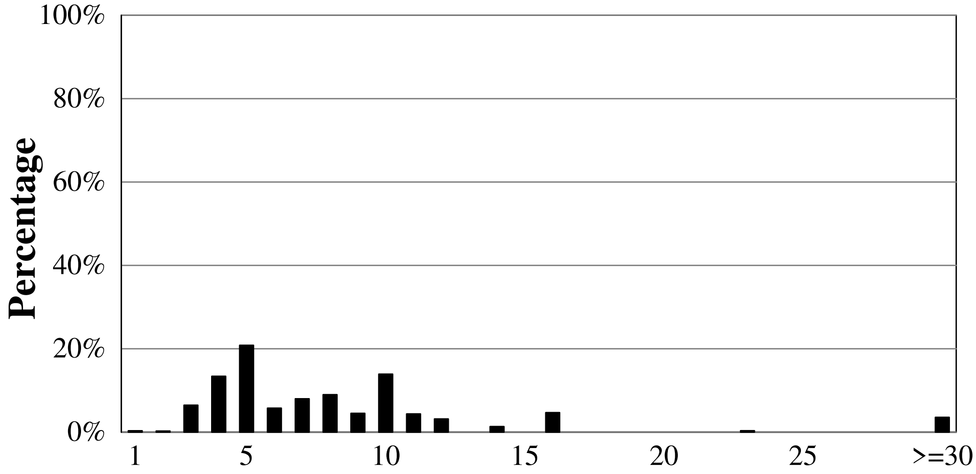


Figure 7.2: Number of Targets per Indirect Jump (gcc)

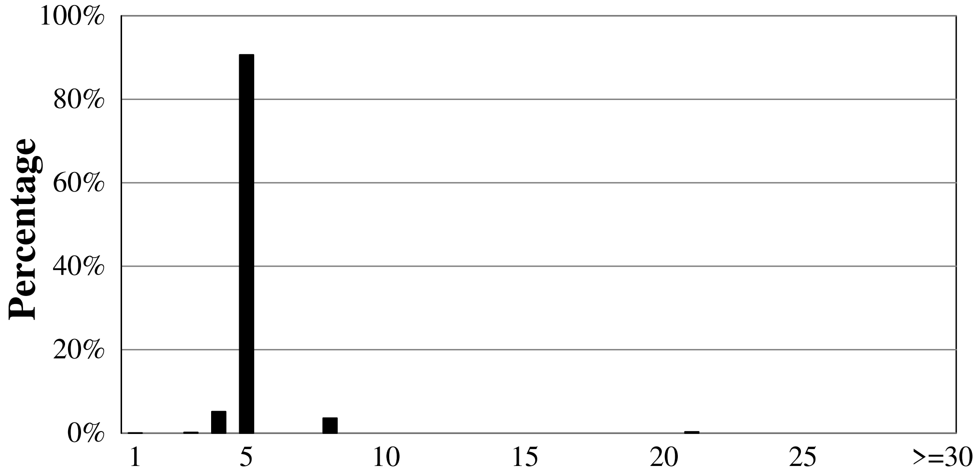


Figure 7.3: Number of Targets per Indirect Jump (go)

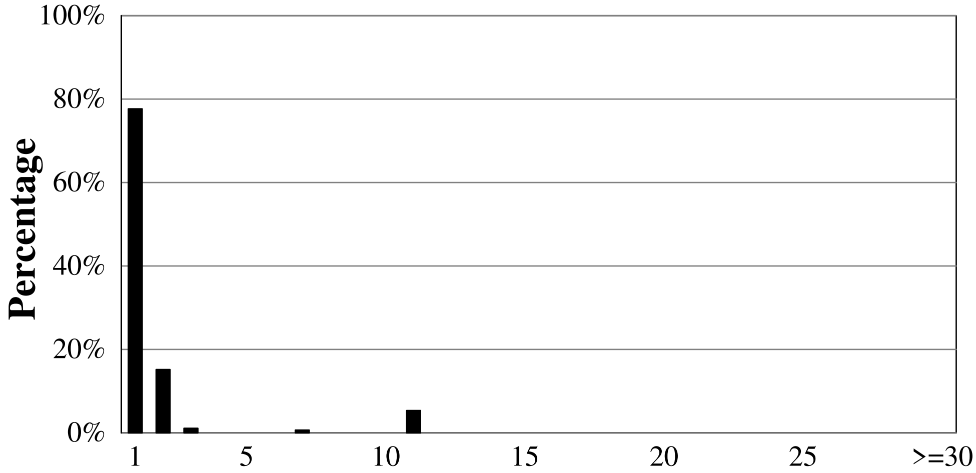


Figure 7.4: Number of Targets per Indirect Jump (ijpeg)

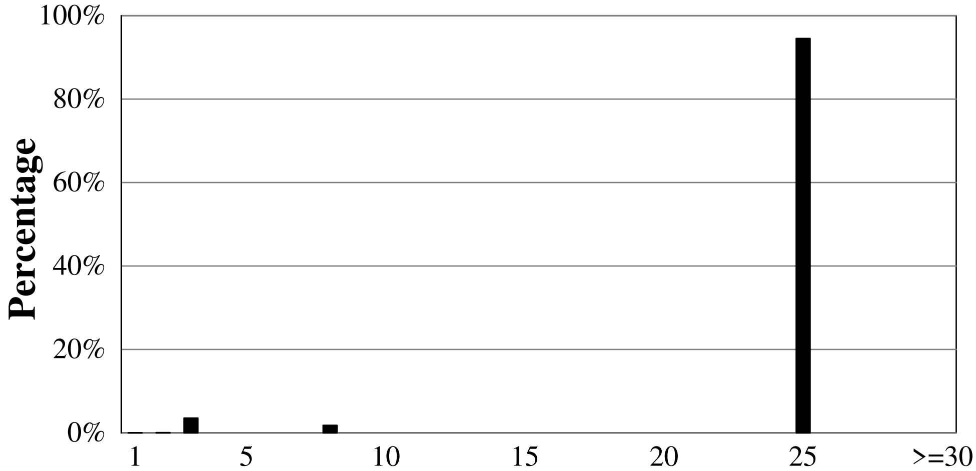


Figure 7.5: Number of Targets per Indirect Jump (li)

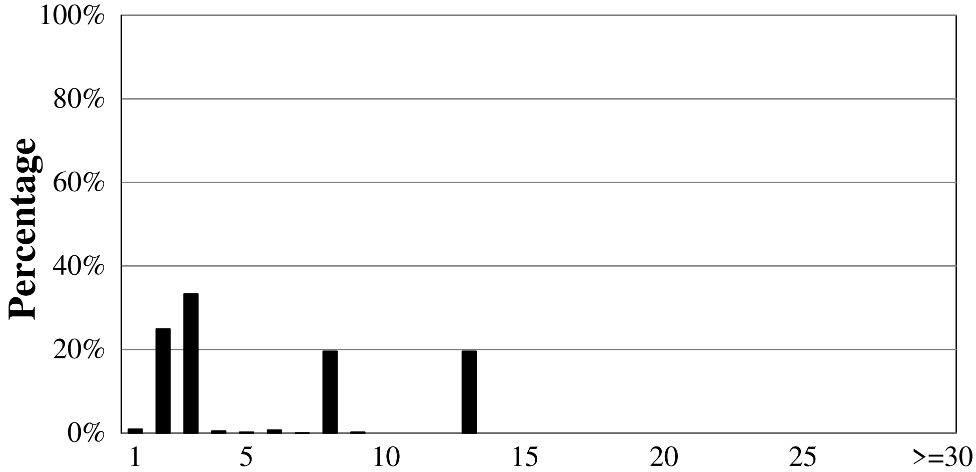


Figure 7.6: Number of Targets per Indirect Jump (m88ksim)

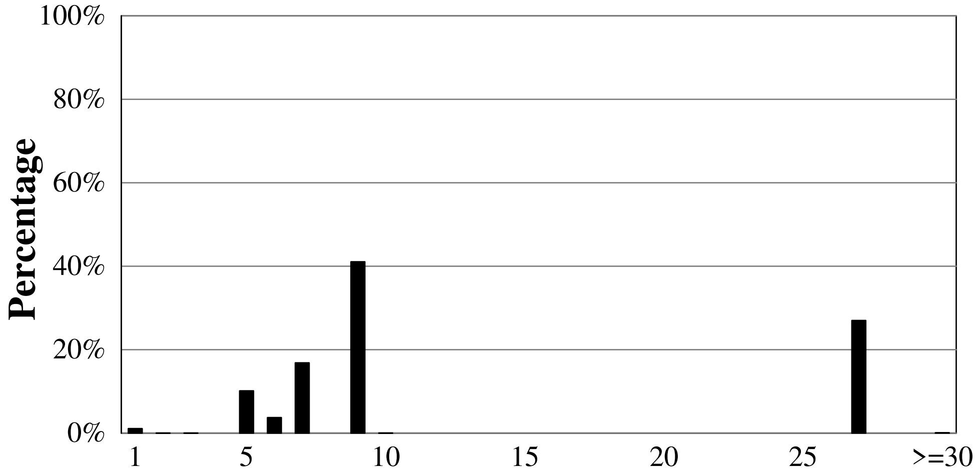


Figure 7.7: Number of Targets per Indirect Jump (perl)

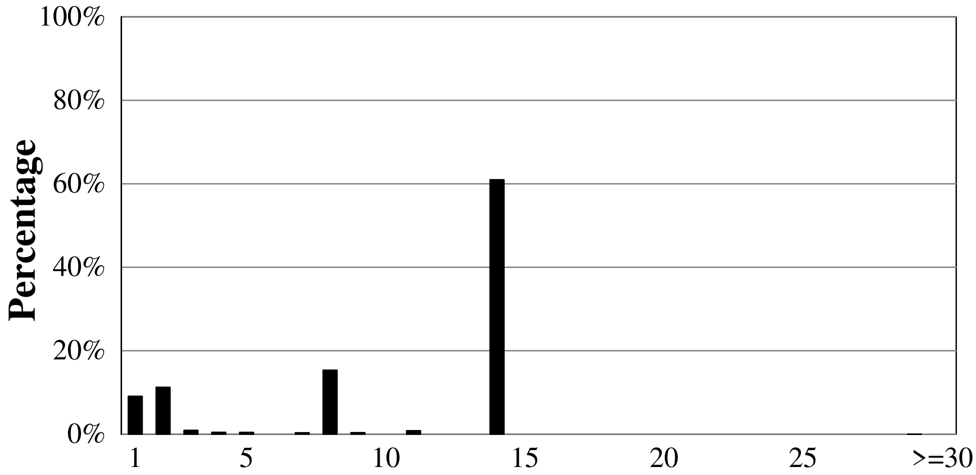


Figure 7.8: Number of Targets per Indirect Jump (vortex)

Table 7.1 lists the Indirect branch target misprediction rate achieved by a 1K-entry 4-way set-associative BTB for the SPECint95 benchmarks. For gcc and perl, the two benchmarks with significant numbers of indirect branches, the misprediction rates were 66.0% and 76.4%. Table 7.1 also shows that the indirect branch target misprediction rates for the SPECint95 benchmarks are significantly higher than the conditional branch misprediction rates achieved by a 2KByte gshare predictor.

Benchmark	Input	#Instr's.	#Branches	Cond. Br Mispred. Rate	#Indirect Jumps	Ind. Jump Mispred. Rate
compress	test.in	125,162,687	17,460,753	8.58%	590	61.7%
gcc	jump.i	172,328,834	35,979,748	11.56%	939,417	66.0%
go	2stone9.in	125,637,006	23,378,150	25.61%	173,719	37.6%
ijpeg	specmun.ppm	206,802,135	23,449,572	8.61%	103,876	14.3%
li	train.lsp	192,569,022	40,909,525	5.21%	114,789	80.7%
m88ksim	dcrand.train.big	131,732,141	23,840,021	2.01%	186,285	37.3%
perl	scrabbl.pl	106,140,733	16,727,047	4.71%	588,136	76.4%
vortex	vortex.in	236,081,621	44,635,060	2.63%	243,706	11.3%

Table 7.1: Misprediction counts for indirect jumps in the SPECint95 benchmarks

7.2 Target Cache

The target cache improves on the prediction accuracy achieved by BTB-based schemes for indirect jumps by choosing its prediction from (usually) all the targets of the indirect jump that have already been encountered rather than just the target that was most recently encountered. When fetching an indirect jump, the target cache is accessed with the fetch address and other pieces of the machine state to produce the predicted target address. As the program executes, the target cache records the target for each indirect jump target encountered.

7.2.1 Accessing the Target Cache

In our study, we consider using branch history along with the branch address to index into the target cache. Two types of branch history information are used to decide which target of the indirect jump will be predicted – pattern history and path history.

- **Branch History**

  It is now well known that the two-level branch predictor improves prediction accuracy over previous single-level branch predictors [38]. The two-level predictors attain high prediction accuracies by using pattern history to distinguish different dynamic occurrences of a conditional branch. To predict indirect jumps, the target cache is indexed using branch address and global pattern history. No extra hardware is required to maintain the branch history for the target cache if the branch prediction mechanism already contains this information. The target cache can use the branch predictor's BHR.

- **Path History**

  Previous research [42, 24] has shown that path history can also provide useful correlation information to improve branch prediction accuracy. Path history consists of the target addresses of branches that lead to the current branch. This information is also useful in predicting indirect branch targets.

  In this study, two different types of path history can be associated with each indirect jump – global or per-address. In the per-address scheme, one path history register is associated with each distinct static indirect branch. Each n-bit path history register records the last k target addresses for the associated indirect jump. That is, when an indirect branch is resolved, n/k bits from its target address are shifted into the path history register.

  In the global scheme, one path history register is used for all indirect branches. Because the history register has a fixed length, it can record a limited number of branches in the past history. Thus, the history register may be better utilized by only recording a particular type of branch instruction. For example, if a sequence of conditional branches is sufficient to distinguish the different paths of each indirect jump, then there is no need to include other types of branches in the path history. Four variations of the global scheme were considered – `control`, `branch`, `call/ret`, `ind jmp`. The `control` scheme records the target address of all instructions that can redirect the instruction stream. The `branch` scheme only records the targets of conditional branches. The `call/ret` scheme records only the targets of procedure calls and returns. The `ind jmp` scheme records only the targets of indirect jumps.

7.2.2 Target Cache Structure

In addition to varying the type of information used to access the target cache, we also studied tagged and tagless target caches.

- **Tagless Target Cache**

  Figure 7.9 shows the structure of a tagless cache. The target cache is similar to the PHT of the two-level branch predictor; the only difference is that a target cache's storage structure records branch targets while a two-level branch predictor's pattern history table records branch directions.

  The target cache works as follows: during instruction fetch, the BTB and the target cache are examined concurrently. If the BTB detects an indirect branch, then the target address found in the target cache is used to redirect the instruction stream. When the indirect branch is resolved, the target cache entry is updated with its target address.

  Several variations of the tagless target cache can be implemented. They differ in the ways that branch address and history information are hashed into the target cache. For example, the branch address can be XORed with the history information for selecting the appropriate entry. Section 7.3 will describe the different hashing schemes considered.

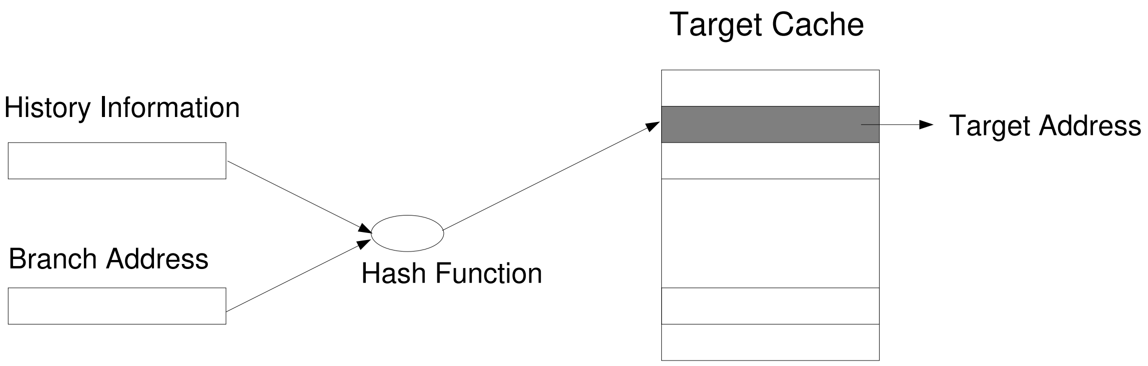


Figure 7.9: Structure of a Tagless Target Cache

- **Tagged Target Cache**

  Like the PHTs, interference occurs in the target cache when a branch uses an entry that was last accessed by another branch. Interference is particularly detrimental to the target cache because each entry in the target cache stores the branch target address. Since the targets of two different indirect branches are usually different, interference will most likely cause a misprediction.

  To avoid predicting targets of indirect jumps based on the outcomes of other branches, we propose the tagged target cache where a tag is added to each target cache entry (see Figure 7.10). The branch address and/or the branch history are used for tag matching. When an indirect branch is fetched, its instruction address and the associated branch history are used to select the appropriate target cache entry. If an entry is found, the instruction stream is redirected to the associated target address.

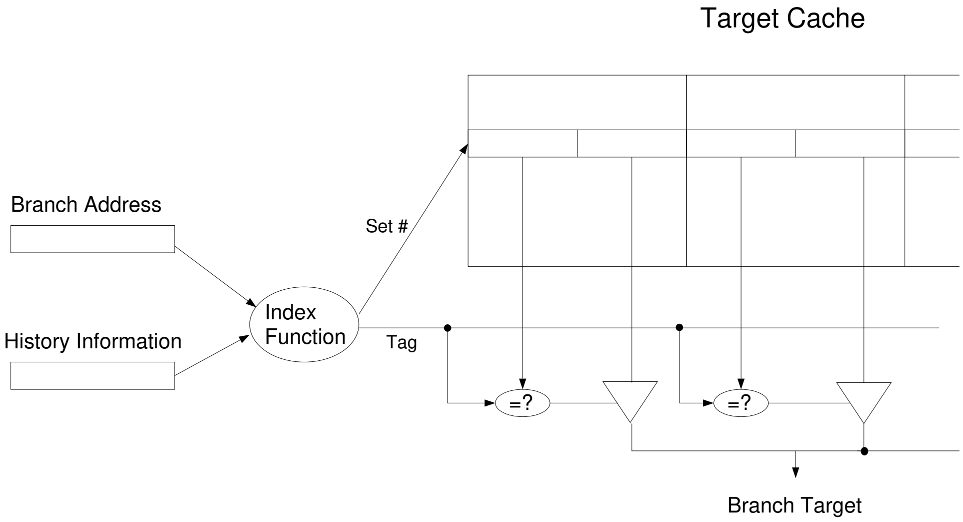


Figure 7.10: Structure of a Tagged Target Cache

7.3 Performance

7.3.1 Tagless Target Cache

This section examines the performance of tagless target caches. The size of every target cache considered in this section is 512 entries. Since the BTB has 256 sets and is 4-way set-associative, the target cache increases the predictor hardware budget by 10 percent. The cost of the predictor is estimated using the following equations:

$$\begin{aligned} \text{BTB}^{*} &= 77 \times 2048 \;\; \texttt{bits} \\ \text{target cache}(n) &= 32 \times n \;\; \texttt{bits} \\ \text{predictor budget} &= \text{BTB} + \text{target cache}(n) \;\; \texttt{bits} \end{aligned}$$

where n is the number of target cache entries.

Hashing Function

With the tagless schemes, branch history information and address bits are hashed together to select the appropriate target cache entry. An effective hashing scheme must distribute the cache indexes as widely as possible to avoid interference between different branches.

[*]each BTB entry consists of 1 valid bit, 2 least-recently-used bits, 23 tag bits, 32 target address bits, 2 branch type bits, 4 fall-thru address bits, and 13 branch history bits.

Table 7.2 shows the performance benefit of tagless target caches using different history information for indexing into the storage structure. The GAg(9) scheme uses 9 bits of branch pattern history to select the appropriate target cache entry. In the GAs schemes, the target cache is conceptually partitioned into several tables. The address bits are used to select the appropriate table and the history bits are used to select the entry within the table. The GAs(8,1) scheme uses 8 history bits and 1 address bits while the GAs(7,2) scheme uses 7 history bits and 2 address bit. For the perl benchmark, GAg(9) slightly outperforms GAs(8,1), showing that branch pattern history provides marginally more useful information than branch address. This is because the perl benchmark executes only 22 static indirect jumps. On the other hand, GAs(8,1) is competitive with GAg(9) for the gcc benchmark, a benchmark which executes a large number of static indirect jumps. In the gshare scheme, the branch address is XORed with the branch history to form the target cache index. Like the previous two-level branch predictors, the gshare scheme outperforms the GAs scheme because it effectively utilizes more of the entries in the target cache. In the following sections, we will use the gshare scheme for tagless target caches.

Perl		
config	Misprediction Rate	Reduction in Exec. Time
gshare(9)	30.9%	8.92%
GAg(9)	31.2%	8.87%
GAs(8,1)	33.4%	8.36%
GAs(7,2)	48.1%	5.17%
Gcc		
config	Misprediction Rate	Reduction in Exec. Time
gshare(9)	30.4%	4.27%
GAg(9)	35.8%	3.61%
GAs(8,1)	35.7%	3.62%
GAs(7,2)	36.7%	3.48%

Table 7.2: Performance of Pattern History Tagless Target Caches

Branch Path History: Tradeoffs

Path history consists of the target addresses of branches that lead to the current branch. Ideally, each path leading to a branch should have a unique representation in the path history register. However, since only a few bits from each target are recorded in the path history register, different targets may have the same representation in the path history. When this occurs, the path history may not be able to distinguish between different paths leading to a branch. Thus, the performance of a path-based target cache depends on the address bits from each target used to form the path history. Table 7.3 shows the performance of target caches using different address bits from each target in the path history. The lower significant bits provide more information than the higher significant bits. In the following experiments, the lower significant bits from each target are recorded in the path history register; the 2 least significant bits from each address are ignored because instructions are aligned on word boundaries.

Perl					
addr	Reduction in Execution Time				
bit	Per-addr	Global			
no.		branch	control	ind jmp	call/ret
2	7.64%	7.41%	4.70%	12.36%	8.45%
3	10.09%	6.85%	3.41%	11.00%	9.75%
4	5.52%	5.97%	4.88%	10.50%	9.97%
5	7.34%	9.13%	4.31%	9.24%	10.82%
7	6.17%	6.89%	4.22%	11.34%	11.16%
10	6.19%	6.60%	3.94%	12.82%	10.42%
15	1.87%	2.98%	2.30%	10.97%	9.92%
20	0.00%	0.00%	0.00%	0.00%	0.00%
Gcc					
addr	Reduction in Execution Time				
bit	Per-addr	Global			
no.		branch	control	ind jmp	call/ret
2	2.57%	3.82%	3.75%	2.68%	2.52%
3	2.46%	3.71%	3.81%	2.63%	2.24%
4	2.67%	3.88%	3.63%	2.69%	2.80%
5	2.02%	3.75%	3.82%	2.37%	2.63%
7	2.60%	3.57%	3.76%	2.72%	2.98%
10	1.34%	3.24%	3.14%	1.83%	2.60%
15	0.01%	1.05%	0.78%	1.06%	1.56%
20	-0.02%	0.26%	0.14%	0.24%	0.78%

Table 7.3: Path History: Address Bit Selection

Because the length of the history register is fixed, there is also a tradeoff between identifying more branches in the past history and better identifying each branch in the past history. Increasing the number of bits recorded per address results in fewer branch targets being recorded in the history register. Table 7.4 shows the performance of target caches using different numbers of bits from each target in the path history. For example, the first row in the table shows the performance of target caches which record one bit from each target address in the path history register. The second row shows the performance of target caches which record two bits from each target address in the path history register. Nine bits of path history are then used to select the appropriate target cache entry. In general, with nine history bits,

Perl					
bits per addr	Reduction in Execution Time				
	Per-addr	Global			
		branch	control	ind jmp	call/ret
1	7.64%	7.41%	4.70%	12.36%	8.45%
2	9.17%	3.12%	0.88%	11.81%	8.40%
3	7.19%	0.30%	0.74%	10.99%	6.24%
Gcc					
bits per addr	Reduction in Execution Time				
	Per-addr	Global			
		branch	control	ind jmp	call/ret
1	2.57%	3.82%	3.75%	2.68%	2.52%
2	2.96%	2.60%	2.21%	3.35%	2.66%
3	3.57%	2.02%	1.66%	3.83%	2.63%

Table 7.4: Path History: Address Bits per Branch

the performance benefit of the target cache decreases as the number of address bits recorded per target increases. This is especially true for the `Control` and `Branch` schemes. With the `Control` scheme, we record the target of all instructions that can redirect the instruction stream, which may include branches that provide little or no useful information, e.g. unconditional branches. Each of these uncorrelated branches takes up history bits, possibly displacing useful history. As a result, the performance benefit of the `Control` scheme drops even more significantly when the number of bits to be recorded for each target increases from 1 to 2; for perl, the reduction in execution time dropped from 4.7% to 0.9%.

Pattern History vs. Path History

Table 7.2 and Table 7.4 show that using pattern history results in better performance for gcc and using global path history results in better performance for perl. Using the `Indirect Jmp` scheme, the execution time for perl was reduced by 12.34%, as compared to 8.92% for the best pattern history scheme. The branch path history was able to perform extremely well for the perl benchmark because it is an interpreter. The main loop of the interpreter parses the the perl script to be executed. This parser consists of a set of indirect jumps whose targets are decided by the tokens (i.e. components) which make up the current line of the perl script. The perl script used for our simulations contains a loop that executes for many iterations. As a result, when the interpreter executes this loop, the interpreter will process the same sequence of tokens for many iterations. By capturing the path history in this situation, the target cache is able to accurately predict the targets of the indirect jumps which process these tokens.

7.3.2 Tagged Target Caches

This section examines the performance of tagged target caches. While the degree of associativity was varied, the total size of each target cache simulated was kept constant at 256 entries. The tagged target caches have half the number of entries as that of tagless target caches to compensate for the hardware used to store tags.

Indexing Function

Since there can be several targets for each indirect branch and each target may be reached with a different history, a large number of target cache entries may be needed to store all the possible combinations. Thus, the indexing scheme into a target cache must be carefully selected to avoid unnecessary trashing of useful information.

Three different indexing schemes were studied – `Address`, `History Concatenate`, and `History Xor`. The `Address` scheme uses the lower address bits for set selection. The higher address bits and the global branch pattern history are XORED to form the tag. The `History Concatenate` scheme uses the lower bits of the history register for set selection. The higher bits of the history register are concatenated with the address bits to form the tag. The `History Xor` scheme XORs the branch address with the branch history; it uses the lower bits from the result of the XOR for set selection and the higher bits for tag comparison. Table 7.5 shows the performance of the different indexing schemes. Global pattern history is used in these experiments. The `Address` selection scheme results in a significant number of conflict misses in target caches with a low degree of set-associativity because all targets of an indirect jump are mapped to the same set. Since there are several targets for each indirect jump for gcc and perl as shown in Section 7.1, a high degree of set-associativity is required to avoid trashing of useful information due to conflict misses. The `History Concatenate` and `History Xor` schemes suffer a much smaller number of conflict misses because they

Perl				Gcc			
	Reduction in Exec Time				Reduction in Exec Time		
	Addr	History			Addr	History	
set-assoc.		Conc	Xor	set-assoc.		Conc	Xor
1	0.00%	8.31%	7.42%	1	0.00%	3.51%	3.56%
2	0.21%	8.30%	7.24%	2	0.30%	3.85%	3.92%
4	2.76%	8.71%	7.76%	4	1.27%	4.01%	4.13%
8	6.82%	9.11%	7.73%	8	2.54%	4.19%	4.24%
16	8.67%	9.14%	8.17%	16	3.48%	4.30%	4.28%
32	8.67%	9.14%	8.20%	32	4.22%	4.32%	4.30%
64	9.10%	9.14%	8.10%	64	4.16%	4.38%	4.31%
128	9.06%	9.14%	8.10%	128	4.26%	4.52%	4.31%
256	8.10%	9.14%	8.10%	256	4.31%	4.59%	4.31%

Table 7.5: Performance of Tagged Target Cache using 9 pattern history bits

can map the targets of an indirect jump into any set in the target cache, removing the need for a high degree of associativity in the target cache. In the following sections, we will use the `History Xor` scheme for tagged target caches.

Pattern History vs. Path History

Table 7.6 shows the performance of tagged target caches that use branch path histories. The path history schemes reported in this section record one bit from each target address into the 9-bit path history register. This design choice resulted in the best performance for most of the path history schemes. As in the tagless schemes, using pattern history results in better performance for gcc and using global path history results in better performance for perl.

Perl					
	Reduction in Execution Time				
set assoc	Per-addr	Global			
		branch	control	ind jmp	call/ret
1	7.13%	6.20%	2.98%	10.71%	7.59%
2	8.20%	6.34%	4.15%	11.62%	7.91%
4	9.15%	7.21%	4.30%	12.29%	7.91%
8	9.46%	7.21%	4.70%	13.38%	8.79%
16	9.69%	7.21%	4.70%	13.78%	8.69%
32	9.77%	7.21%	4.70%	13.97%	8.83%
64	9.77%	7.21%	4.70%	14.14%	8.78%
128	9.77%	7.21%	4.70%	14.15%	8.98%
256	9.77%	7.21%	4.70%	14.15%	8.66%
Gcc					
	Reduction in Execution Time				
set assoc	Per-Addr	Global			
		branch	control	ind jmp	call/ret
1	2.45%	3.28%	2.88%	2.15%	2.07%
2	2.72%	3.35%	3.28%	2.46%	2.25%
4	2.90%	3.59%	3.58%	2.65%	2.42%
8	3.01%	3.73%	3.55%	2.76%	2.56%
16	3.05%	3.76%	3.58%	2.86%	2.58%
32	3.11%	3.76%	3.59%	2.90%	2.63%
64	3.12%	3.77%	3.59%	2.92%	2.64%
128	3.14%	3.79%	3.57%	2.92%	2.65%
256	3.14%	3.79%	3.57%	2.94%	2.65%

Table 7.6: Performance of Tagged Target Caches using 9 path history bits

Branch History Length

For tagged target caches, the number of branch history bits used is not limited to the size of the target cache because additional history bits can be stored in the tag fields. Using more history bits may enable the target cache to better identify each occurrence of an indirect jump in the instruction stream.

Table 7.7 compares the performance of 256-entry tagged target caches using different numbers of global pattern history bits. For caches with a high degree of set-associativity, using more history bits results in a significant performance improvement. For example, when using 16 history bits, an 8-way tagged target cache reduces the execution time of perl and gcc by 10.27% and 4.31% respectively, as compared to 7.73% and 4.28% when using 9 history bits. Greater improvements are seen for caches with higher set associativity. For a 16-way tagged target cache, using 16 history bits results in 12.66% and 4.74% reduction in execution time for perl and gcc, while using 9 history bits results in 8.17% and 4.28% reduction in execution time.

For target caches with a small degree of set-associativity, using more history bits degrades performance. Using more history bits enables the target cache to better identify each occurrence of an indirect jump in the instruction stream; however, more entries are required to record the different occurrences of each indirect jump. The additional pressure on the target cache results in a significant number of conflict misses. The performance loss due to conflict misses outweighs the performance gain due to better identification of each indirect jump.

Perl			Gcc		
	Reduction in Exec Time			Reduction in Exec Time	
set-assoc.	9 bits	16 bits	set-assoc.	9 bits	16 bits
1	7.42%	7.07%	1	3.56%	2.35%
2	7.24%	8.39%	2	3.92%	2.99%
4	7.76%	10.02%	4	4.13%	3.62%
8	7.73%	10.27%	8	4.24%	4.31%
16	8.17%	12.66%	16	4.28%	4.74%
32	8.20%	12.71%	32	4.30%	4.87%
64	8.10%	12.83%	64	4.31%	5.11%
128	8.10%	12.84%	128	4.31%	5.19%
256	8.10%	12.85%	256	4.31%	5.24%

Table 7.7: Tagged Target Cache: 9 vs 16 pattern history bits

7.3.3 Tagless Target Cache vs Tagged Target Cache

A tagged target cache requires tags in its storage structure for tag comparisons while a tagless target cache does not. Thus, for a given implementation cost, a tagless target cache can have more entries than a tagged target cache. On the other hand, interference in the storage structure can degrade the performance of a tagless target cache.

Figures 7.11 and 7.12 compare the performance of a 512-entry tagless target cache to that of several 256-entry target caches, with various degrees of set-associativity. The tagless target cache outperforms tagged target caches with a small degree of set-associativity. On the other hand, a tagged target cache with 8 or more entries per set outperforms the tagless target cache.

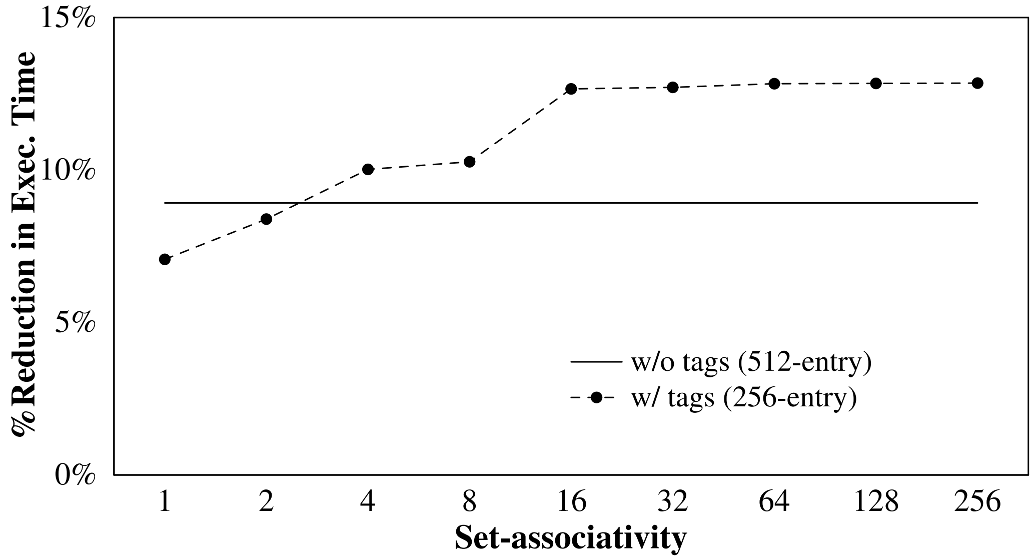


Figure 7.11: Tagged vs. tagless target cache (perl)

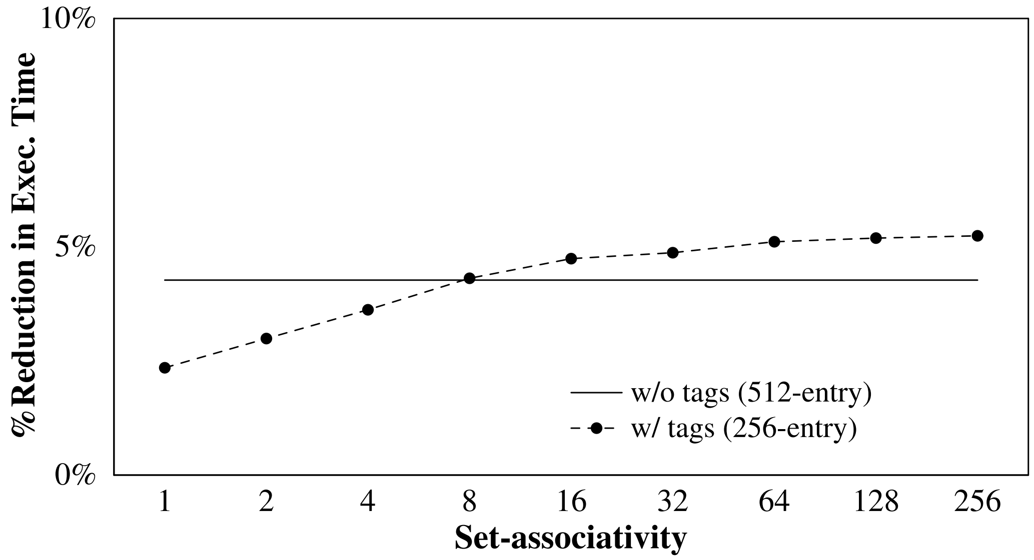


Figure 7.12: Tagged vs. tagless target cache (gcc)

7.4 Summary

Current BTB-based branch prediction schemes are not effective in predicting the targets of indirect jumps. These schemes have a 48% misprediction rate for the indirect jumps in the SPEC95 integer benchmarks. To address this problem, this dissertation proposes the target cache, a prediction mechanism that significantly improves the accuracy of predicting indirect jump targets. The target cache uses concepts embodied in the two-level branch predictor.

By using branch history to distinguish different dynamic occurrences of indirect branches, the target cache was able to reduce the total execution time of perl and gcc by 8.92% and 4.27% respectively.

Like the PHTs of the two-level branch predictors, interference can significantly degrade the performance of a target cache. To avoid predicting targets of indirect jumps based on the outcomes of uncorrelated branches, tags are added to the target cache. However, with the tagged target cache, useful information can be displaced if different branches are mapped to the same set. Thus, set-associative caches may be required to avoid mispredictions due to conflict misses.

Our experiments showed that a tagless target cache outperforms a tagged target cache with a small degree of set-associativity. On the other hand, a tagged target cache with 8 or more entries per set outperforms a tagless target cache. For example, a 512-entry tagless target cache can reduce the execution time of perl and gcc by 8.92% and 4.27% respectively, as compared to 7.42% and 3.56% for a direct mapped 256-entry tagged target cache. A 16-way set-associative tagged target cache can reduce the execution of gcc and perl by 12.66% and 4.74% respectively.

CHAPTER 8

Predicated Execution

This chapter examines the performance benefit of using both speculative and predicated execution. To minimize the effect of each approach's disadvantages, speculative execution is used to handle the branches that are accurately predicted by the branch predictor and predicated execution is used to eliminate the remaining hard-to-predict branches. Profiling is used to determine which branches are inaccurately predicted by the branch predictor. We show that for most of the SPECint92 benchmarks, profiling is an effective means for determining which branches are difficult to predict for the two-level branch predictor. Furthermore, we show that the addition of predicated execution can provide a significant performance increase.

8.1 Identifying Hard-to-Predict Branches

Since predicated execution is a big win when handling conditional branches that are poorly predictable, we examine the performance benefits of using predicated execution to eliminate the hard-to-predict branches while using speculative execution to handle the branches that are accurately predicted by the branch predictor.

Previous researchers have used branch taken rates to determine which branches to eliminate. Although branch taken-rate can effectively identify the hard-to-predict branches for static branch predictors, branch taken-rate alone are probably not sufficient to identify the hard-to-predict branches for dynamic branch predictors. That is, dynamic predictors may be able to accurately predict those branches whose taken-rates are not mostly-taken or mostly-not-taken. Using taken-rate alone, we may mistakenly classify these branches as hard-to-predict. Figure 8.1 shows a branch with a dynamic taken-rate of 60%. This branch is taken for the first 6 times and then not-taken for the next 4 times. Using taken-rate as the metric, this branch will be classified as hard-to-predict. However, as shown in the lower part of the table, a simple dynamic predictor like the last time taken predictor can capture the change in this branch's behavior and predict this branch with 90% accuracy. Therefore, profiling the performance of the dynamic branch predictor may better identify the hard-to-predict branches.

To determine which branches to consider for elimination, each benchmark was profiled with a training data set. The profiler modeled the processor's branch predictor

- Static predictors: record branch direction

Branch:	T	T	T	T	T	T	N	N	N	N
Profile:	T: 6, N: 4									

- Dynamic predictors: record correct branch predictions

Branch:	T	T	T	T	T	T	N	N	N	N
Predictions:	T	T	T	T	T	T	T	N	N	N
Profile:	C: 9, I: 1									

C = correct prediction, I = incorrect prediction

Figure 8.1: Taken rate vs. predictor accuracy

and recorded the number of mispredictions for each static branch. Branches whose misprediction counts exceeded a given threshold were considered hard to predict and marked for elimination.

The branch predictor simulated by the profiler in our experiments was the Gshare/PAg predictor [20, 7]. This hybrid branch predictor combines the gshare(13) and PAs(11,2) predictors.

Compiled simulations as described in Section 3.1.1 are used to report the performance of the branch predictor for each static branch in each of the six SPECint92 benchmarks. Table 8.1 lists the input data sets used to profile each of the benchmarks.

	Profiling Inputs		
Benchmark	Input 1	Input 2	Input 3
008.espresso	cps	bca	ti
022.li	9 queens	hanoi[a]	roots[b]
023.eqntott	int1.eqn[c]	int2.eqn[d]	fx2fp.eqn[e]
026.compress	in	gcc src[f]	trace[g]
072.sc	loada1	loada2	loada3
085.gcc	rttv.i[h]	stmt.i	gcc.i

[a] Tower of Hanoi

[b] Newton's method for approximating square roots

[c] Abbreviated version of the SPECint reference input set int_pri_3.eqn. It consists of 15 boolean equations with 39 different variables.

[d] Abbreviated version of the SPECint reference input set int_pri_3.eqn. It consists of 27 boolean equations with 49 different variables. The majority of the equations differ from those used in int1.eqn.

[e] Fixed point to floating point encoder.

[f] Concatenated gcc source files (~1MB).

[g] Motorola MC88110 instruction trace of compress (~1MB).

[h] rttv.i is the concatenation regclass.i, toplev.i, tree.i, and varasm.i.

Table 8.1: Input data sets used to profile the SPECint92 benchmarks

The SPECint92 reference input sets were used whenever possible, but because only one input set was provided for eqntott, compress, and li, their profiles were based on inputs that were not from the SPECint92 suite.

For each profile, the branches were listed from worst to best where the branch with the largest number of mispredictions was considered the worst one. Branches that appeared in the list before the cumulative number reached 75% of the total number of mispredictions were considered to be the hard-to-predict branches for that profile. The profiles show that for all the benchmarks, with the exception of gcc, there were very few hard-to-predict branches. Tables 8.2 through 8.4 list for each benchmark the percentage of total mispredictions that were covered by the set of branches that are profiled as hard-to-predict. Each table uses a different input file to generate the profile. With the exception of sc, an average of 73% of the total mispredictions for each of the benchmarks were covered by the set of branches that were profiled as hard-to-predict. This result shows that the set of hard-to-predict branches for a given benchmark is consistent over all the input data sets profiled, indicating that

Benchmark	*Input 1*	Input 2	Input 3
008.espresso	*75.25*	70.55	67.28
022.li	*75.03*	68.32	50.36
023.eqntott	*83.26*	78.36	85.12
026.compress	*76.57*	83.29	86.38
072.sc	*89.66*	55.25	6.43
085.gcc	*75.01*	76.68	74.54

Table 8.2: Percentage of mispredictions covered by branches specified as hard-to-predict by the *Input 1* data set

Benchmark	Input 1	*Input 2*	Input 3
008.espresso	35.77	*75.05*	30.57
022.li	79.52	*75.62*	59.54
023.eqntott	83.26	*78.36*	85.12
026.compress	76.57	*83.29*	86.38
072.sc	53.34	*76.34*	5.59
085.gcc	71.74	*75.02*	71.21

Table 8.3: Percentage of mispredictions covered by branches specified as hard-to-predict by the *Input 2* data set

Benchmark	Input 1	Input 2	*Input 3*
008.espresso	70.46	67.02	*75.18*
022.li	71.19	74.01	*75.20*
023.eqntott	83.26	78.36	*85.12*
026.compress	76.57	83.29	*86.38*
072.sc	32.94	36.70	*76.56*
085.gcc	73.89	75.38	*75.01*

Table 8.4: Percentage of mispredictions covered by branches specified as hard-to-predict by the *Input 3* data set

the majority of the hard-to-predict branches can be detected by profiling. Sc, the spreadsheet program, did not fare as well because each of the input data sets focused on a different subset of the spreadsheet commands. Our experiment also yielded an anomalous result for espresso. The profiles generated from the *Input 1* and *Input 3* data sets covered the majority of the hard-to-predict branches for the other two data sets, but the *Input 2* profile did not. This is because *Input 1* and *Input 3*'s profiles had to include a large number of static branches to reach the 75% misprediction threshold while *Input 2*'s profile required a smaller number of static branches. In addition, this smaller set was subsumed by the larger sets. As a result, *Input 2*'s profile was unable to provide good coverage of the other input data sets even though their profiles were able to provide good coverage of its hard-to-predict branches.

8.2 Predication Model

Our predication model assumes that each predicated instruction has three or four source operands: one or two source operands used for calculating the value generated by the instruction, one predicate operand, and one implicit source operand specified by the destination register [4, 28]. The predicate operand is an ordinary register. The predicated instruction can interpret its value by its least significant bit or by the complement of its least significant bit. Although predicate registers are usually set by compare instructions, they can be set by any instruction. If the predicate evaluates to *true*, the predicated instruction executes like a regular instruction, the destination register is set to the value calculated from the source operands. If the predicate evaluates to *false*, the predicated instruction writes the old value of the destination register (the implicit operand) back into the destination register. This is done instead of suppressing the execution of the instruction because the machine simulated uses dynamic register renaming [35, 26]. For register renaming to function correctly, every issued instruction must eventually produce a value. This requirement has the drawback that it forces every predicated instruction to be part of a dependency chain regardless of the value of its predicate. Figure 8.2 illustrates this drawback with a modified code fragment from the eqntott benchmark. Although there is no true dependence between the two predicated move instructions in Figure 8.2, the second move instruction has a data dependence on the first move instruction due to the implicit destination operand (r5) [1].

```
/* C code */
if (aa < bb)
   res = -1;
else
   res = 1;

;; Assembly code with predicated instructions
cmp_lt  r4, r2, r3
movi    r5, -1,     if r4
movi    r5,  1,     ifnot r4
```

Figure 8.2: Elimination of an *if-then-else* branch with predicated instructions.

8.3 Experimental Results

Experiments were run to measure the performance benefit of adding speculative execution for the four SPECint92 benchmarks, eqntott, compress, sc, and gcc. Three different variations for each benchmark were simulated:

- *np* – baseline version of the benchmark in which none of the branches were eliminated.
- *sp* – software-based predication version in which branches were eliminated by the GCC compiler through the use of logical and bit-manipulation operations [13]. Table 8.5 shows an example of software-based predication.
- *hp* – ISA-based predication version in which branches were eliminated by predicated instructions.

Before	After

`if (a == 2)`
`   a = 0;`
	`cc = (a != 2);`
	`cc` = sign-extend the least significant bit of `cc`;
	`a = a & cc;`

Table 8.5: Example of software-based predication

[1]Sprangle and Patt [33] have proposed a static register tagging scheme that avoids this drawback by eliminating the need to execute predicated instructions when their predicates are false. Our simulations, however, do not take advantage of this scheme.

The branches that were considered for elimination by predicated execution were chosen based on the profiles of the *Input 1* data set for each benchmark (see Section 8.1). Tables 8.6 through 8.9 list up to the ten worst branches in descending order for each benchmark simulated. The branch which accounts for the largest number of branch mispredictions is considered to be the worst. A check in the *hp* or the *sp* column indicates that the branch is removed in the *hp* or *sp* version of the program

File	Line No.	% misprediction			*hp*	*sp*
		input1	input2	input3		
compress.c	790	29.06	35.50	41.37	√	
compress.c	799	27.31	19.16	20.07	√	√
compress.c	802	20.20	28.63	24.94	√	
	total :	76.57	83.29	86.38		

Table 8.6: Hard-to-predict branches (compress)

File	Line No.	% misprediction			*hp*	*sp*
		input1	input2	input3		
pterm_ops.c	45	58.52	52.40	59.71	√	√
pterm_ops.c	47	24.74	25.96	25.41	√	√
	total :	83.26	78.36	85.12		

Table 8.7: Hard-to-predict branches (eqntott)

File	Line No.	% misprediction			*hp*	*sp*
		input1	input2	input3		
tree.c	408	2.31	2.13	1.84	√	
c-parse.tab.c	1016	2.02	2.15	1.56		
c-parse.tab.c	3185	1.57	1.62	1.11	√	
reload.c	1190	1.52	1.25	1.72		
c-parse.tab.c	3185	1.45	1.54	0.99	√	
c-parse.tab.c	1056	1.41	1.53	1.13	√	
c-parse.tab.c	2272	1.29	1.25	1.09	√	
tree.c	522	1.23	1.40	0.82	√	
c-parse.tab.c	1056	1.15	1.16	0.92	√	
reload.c	1190	1.14	1.02	1.27		
	total :	15.09	15.05	12.45		

Table 8.8: Hard-to-predict branches (gcc)

File	Line No.	% misprediction input1	input2	input3	*hp*	*sp*
interp.c	264	36.32	0.00	0.85	√	
interp.c	1002	32.11	27.55	5.18	√	
interp.c	1001	21.22	27.70	0.40		
	total :	89.66	55.25	6.03		

Table 8.9: Hard-to-predict branches (sc)

respectively. Since not all branches can be predicated (e.g. loop branches can not be predicated), one of the hard-to-predict branches in the *sc* benchmark was not eliminated. In addition, because *gcc* had a large number of hard-to-predict branches and the branch elimination for each benchmark was done by hand, only a small fraction of the hard-to-predict branches was eliminated for the *gcc* benchmark. For the sc and gcc benchmarks, the compiler did not eliminate any branches through software predication. The *sp* and *np* versions were identical for those benchmarks. Experimental runs were done using the *Input 1*, *Input 2*, and *Input 3* data sets for each benchmark. Although the *Input 1* results were unrealistically optimistic because they were based on the use of the same data set for both profiling and execution, they were included to provide a baseline to which the *Input 2* and *Input 3* results could be compared.

Figures 8.3 through 8.6 show the misprediction rate for each benchmark. The ISA-predicated versions of the compress and eqntott benchmarks showed large reductions in misprediction rates across all three input data sets. The relative drops in the

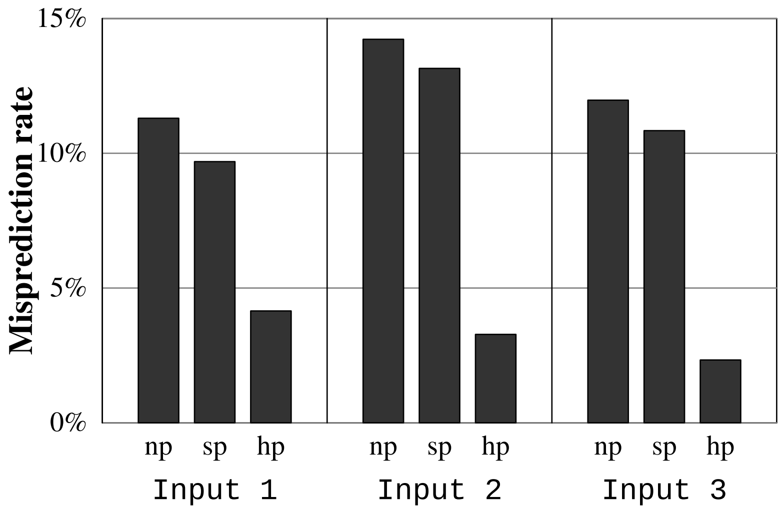


Figure 8.3: Predicated execution's effect on the misprediction rate (compress)

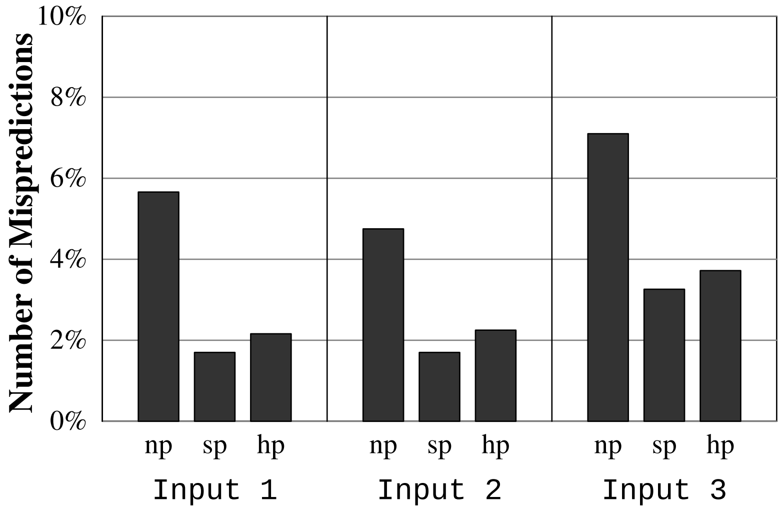


Figure 8.4: Predicated execution's effect on the misprediction rate (eqntott)

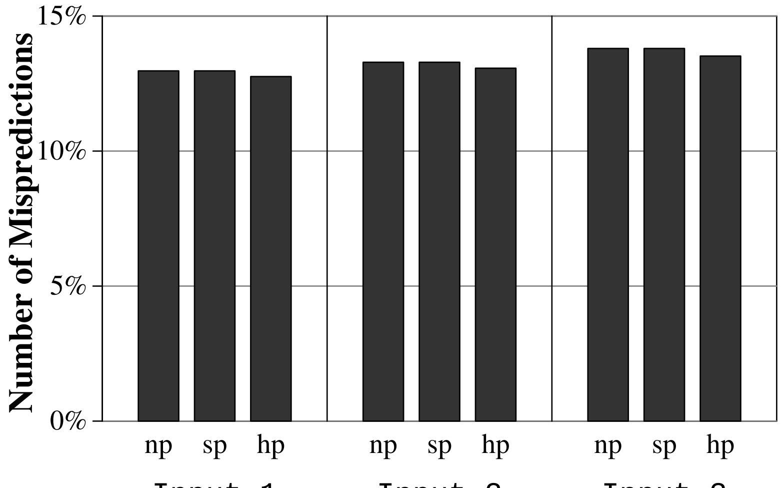


Figure 8.5: Predicated execution's effect on the misprediction rate (gcc)

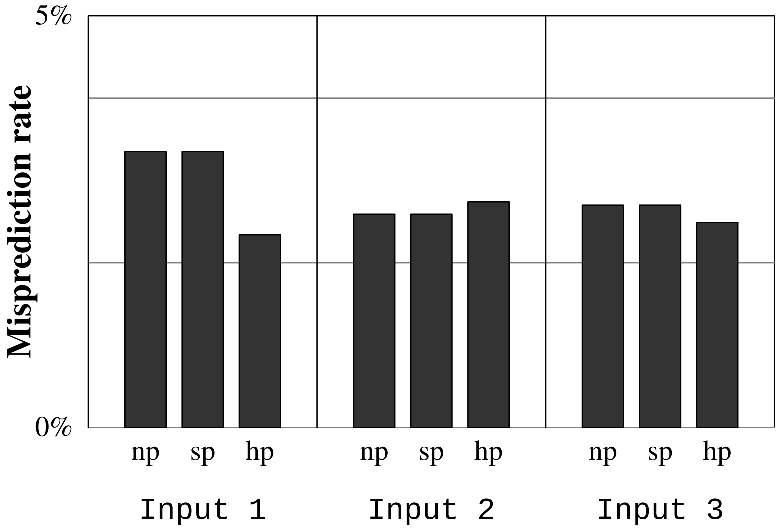


Figure 8.6: Predicated execution's effect on the misprediction rate (sc)

misprediction rates for the second and third data sets were consistent with the first data set, indicating that profiling was locating a significant number of the hard-to-predict branches. In addition, the misprediction rate for compress's *hp* version was significantly lower than the misprediction rate for its *sp* version because ISA-based predication was able to eliminate more of the hard-to-predict branches than software-based predication. The eqntott benchmark showed little difference between its *hp* and *sp* versions because software-based predication was able to eliminate all the hard-to-predict branches eliminated by ISA-based predication.

The ISA-predicated versions of the gcc benchmark showed small reductions in mispredictions across all three input data sets. Because gcc had a large number of hard-to-predict branches and the branch elimination for each benchmark was done by hand, only a small fraction of the hard-to-predict branches were eliminated for the gcc executable used in this study. Only the top ten hard-to-predict branches were considered for elimination. These branches accounted for 7.8% of the total mispredictions in the profile run. By considering only these branches, an average of 3.5% of the total mispredictions for the three input sets were eliminated. Assuming these numbers are representative for the rest of gcc's hard-to-predict branches, predicated execution can be projected to eliminate 40% of gcc's mispredictions.

The ISA-predicated versions of the sc benchmark showed a large reduction in the misprediction rate for the first input data set, but little reduction in the misprediction rate for the second and third input sets. Given that the first input data set was used to profile the benchmark, this result indicates that while predicated execution can be effective in eliminating the hard-to-predict branches in the sc benchmark, profiling was not effective in locating them.

We have shown that predicated execution can be effective in removing hard-to-predict branches. Although removing the hard-to-predict branches eliminates the branch penalties due to these branches, predicated execution may still degrade the performance of a processor by increasing the dependency chains in the program and wasting issue bandwidth. Thus, we show the performance impact of predicated execution on the HPS microarchitecture. Figures 8.7 through 8.10 show the total number of cycles needed to execute each of the benchmarks. This number was broken down into two components, the total number of cycles spent on the correct path of the program (i.e. doing the real work) and the total number of cycles that were spent on incorrect paths of the program, waiting for mispredicted branches to be resolved.

The execution times of the *hp* version of the compress benchmark was 23% faster than that of the *np* version. This speedup was solely due to the reduction in cycles wasted due to branch misprediction. The *hp* version was actually spending additional time executing along the correct path. This effect was due to the predicated instructions increasing the latency of the program because of their extra flow dependencies. Despite its smaller number of mispredictions, the *sp* version was slower than the *np* version by an average of 9.7%. The slowdown was caused by the software predication. Elimination of an inner-loop branch in the *sp* version required a large number of logical operations that greatly increased the critical path within the loop. This resulted in an increase in execution time that could not be offset by the number of cycles saved due to the elimination of the branch.

The *hp* version of the eqntott benchmark outperformed the *np* version by 20%. For the *hp* version, this increase in performance was almost entirely due to the difference in branch execution penalty. Unlike compress' *sp* version, eqntott's *sp* version showed a significant performance increase over the *np* version (24%) because the sequences of logical operations required to eliminate the hard-to-predict branches were extremely short.

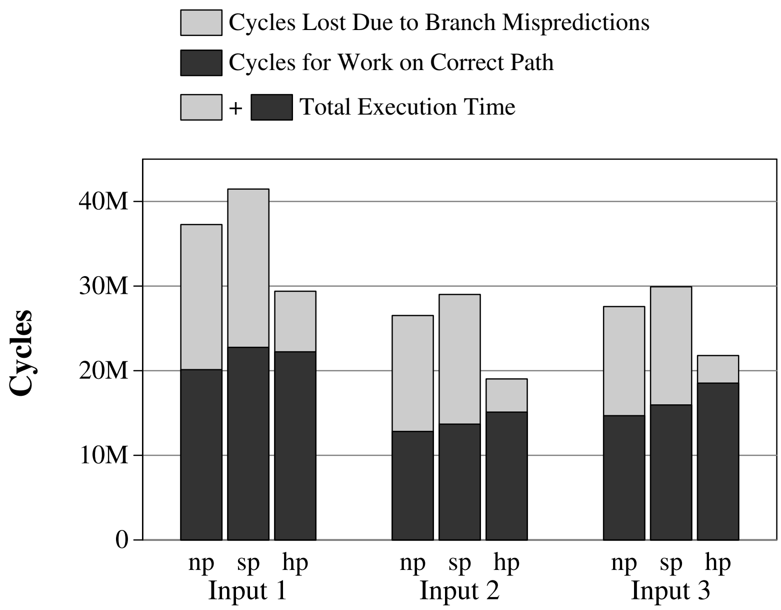


Figure 8.7: Predicated execution's effect on execution time (compress)

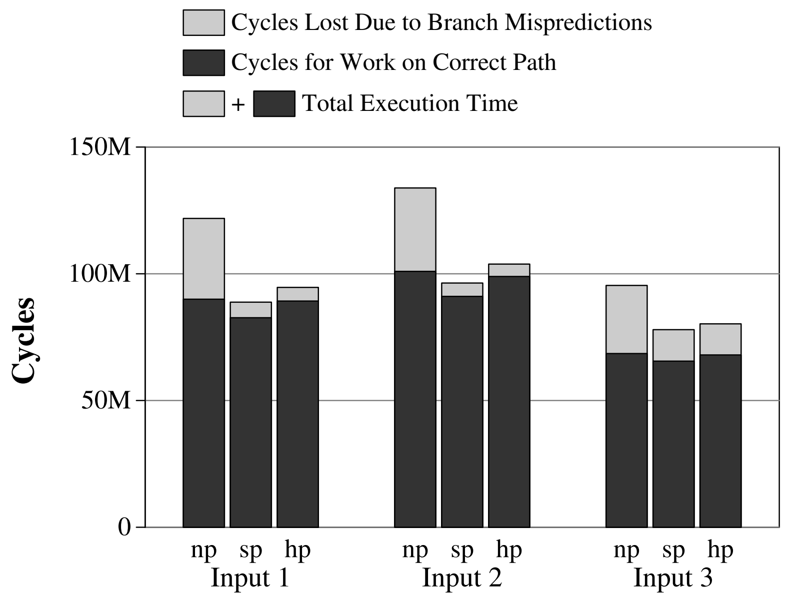


Figure 8.8: Predicated execution's effect on execution time (eqntott)

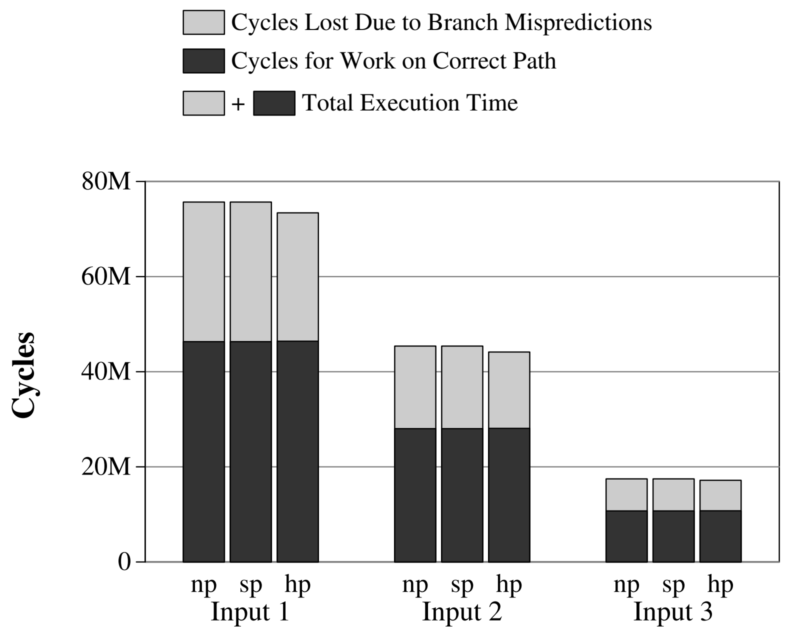


Figure 8.9: Predicated execution's effect on execution time (gcc)

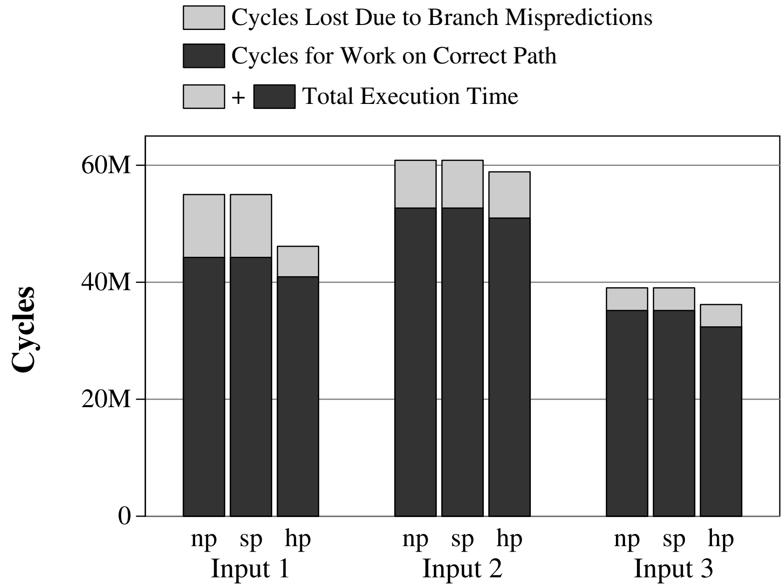


Figure 8.10: Predicated execution's effect on execution time (sc)

The *hp* predicated version of the gcc benchmark outperformed the *np* version by a very small margin, 2.5%. As mentioned above, the difference in performance was small because this study attempted to eliminate only a small fraction of the hard-to-predict branches.

The *hp* predicated versions of the sc benchmark showed small performance increases over the *np* version for the second and third input sets. The sc benchmark did not achieve the same level of improvement as the other benchmarks because the profile generated by its first input set did not locate many of the hard-to-predict branches for its second and third input sets.

8.4 Summary

This study examined the performance benefit of combining speculative and predicated execution to reduce branch execution penalty. Speculative execution eliminates the branch execution penalty for branches that are predicted correctly. Predicated execution can eliminate a branch's execution penalty regardless of the processor's ability to predict it correctly. However, it incurs a small performance overhead and is not applicable to all branches. To achieve a better combination of these two mechanisms, this study used predicated execution to handle the hard-to-predict branches and speculative execution to handle the remaining branches. Profiling was used to determine the hard-to-predict branches for each benchmark. The performance benefit of this approach was measured for a wide-issue dynamically scheduled processor.

The results show that profiling is an effective mechanism for detecting hard-to-predict branches. For a given benchmark, one input data set was chosen to generate the profile. The effectiveness of the profile was then measured for all the input data sets for the benchmark. With the exception of sc, the set of branches denoted by profiling as hard-to-predict accounted for an average of 73% of the total mispredictions for each of the SPECint92 benchmarks.

By using ISA-based predication to eliminate only the branches from the profile-generated set of hard-to-predict branches, significant performance improvements were achieved for compress and eqntott (23% and 20%). Profiling was effective in locating the hard-to-predict branches for both these benchmarks and predicated execution was effective in eliminating those branches. Predicated execution provided only a small performance benefit for gcc (2.5%). However this result was considered promising because the limitations of this study forced us to only consider a very small subset of gcc's hard-to-predict branches for elimination. Based on this result, we project that predicated execution can eliminate up to 40% of gcc's mispredictions. The predicated version for sc showed little improvement in performance. Because profiling was not effective in finding the hard-to-predict branches for sc, predicated execution was not able provide much performance benefit.

Software-based predication can also significantly improve performance. The eqntott benchmark showed little difference between its *hp* and *sp* versions because software-based predication was able to efficiently eliminate all the hard-to-predict branches eliminated by ISA-based predication. However, software-based prediction can also degrade performance when not carefully applied. The sp-version of the *compress* benchmark was slower than the baseline model because software-based predication greatly increased the critical path of a program after eliminating a hard-to-predict branch with a large number of logical operations. This increase in execution time was not offset by the number of cycles saved due to the elimination of the branch.

CHAPTER 9

Concluding Remarks and Future Directions

9.1 Conclusions

This dissertation proposes branch classification as a means for improving branch predictors. The benefit of branch classification was then demonstrated by improving the accuracy of branch predictions:

- Branch classification allows an individual branch instruction to be associated with the branch predictor best suited to predict its direction. Using this approach, a hybrid branch predictor was constructed which exploits both compile-time and run-time information in choosing a component predictor for each branch; profile-guided prediction is used for biased branches and the PAs/gshare hybrid branch predictor for mixed-direction branches. With a fixed implementation cost of 32K bytes, the new hybrid branch predictor achieved a prediction accuracy of 96.91% on gcc as compared to 96.47% for the best previously known predictor, reducing the miss rate by 12.5%.

- Two-level branch predictors have been shown to achieve high prediction accuracy. This dissertation introduced a new method for improving the performance of two-level branch predictors. The performance of two-level branch predictors is degraded due to a high degree of pattern history table interference. For reducing the pattern history table interference of two-level branch predictors, branches are dynamically classified as strongly biased or mixed-directional. Experiments showed that about half of the branches in the dynamic instruction stream are strongly biased, and can therefore be handled with a simple predictor.

  Inhibiting the update to the pattern history table for the strongly biased branches can eliminate a considerable amount of destructive PHT interference with a resulting improvement in the accuracy of the gshare two-level branch predictor. For six of the SPECint95 benchmarks, we achieved an average 21.6% reduction of mispredictions for a 2 KByte gshare predictor. The filtering mechanism also significantly improved the performance of the gshare/pshare hybrid branch predictor which uses gshare for one of its component predictors. For a 6KByte gshare/pshare, the number of mispredictions was reduced by 14.7%.

For the gcc benchmark which executes a large number of static branches, the filtering mechanism was able to remove a significant amount of the destructive interference, reducing the number of mispredictions by 30% to 38% for the gshare predictor. For the gshare/pshare hybrid branch predictor, the number of mispredictions was reduced by 23% to 29%.

- This dissertation showed that the current BTB-based branch prediction schemes are not effective in predicting the targets of indirect jumps. These schemes have a 48% misprediction rate for the indirect jumps in the SPEC95 integer benchmarks. To address this problem, this dissertation proposes the target cache. The target cache uses the concepts embodied in the two-level branch predictor. By using branch history to distinguish different dynamic occurrences of indirect branches, the target cache was able to reduce the total execution time of perl and gcc by 14% and 5%.

- This dissertation also proposed a method for combining speculative and predicated execution to improve performance. Speculative execution eliminates the branch execution penalty for branches that are predicted correctly. Predicated execution can eliminate a branch's execution penalty regardless of the processor's ability to predict it correctly. However, it incurs a small performance overhead and is not applicable to all branches. To achieve a better combination of these two mechanisms, this study used predicated execution to handle the hard-to-predict branches and speculative execution to handle the remaining branches. Profiling was used to determine the hard-to-predict branches for each benchmark. The performance benefit of this approach was measured for a wide-issue dynamically scheduled processor. By using ISA-based predication to eliminate only the branches from the profile-generated set of hard-to-predict branches, additional performance improvements of 23% and 20% were obtained for compress and eqntott. Profiling was effective in locating the hard-to-predict branches for both these benchmarks and predicated execution was effective in eliminating those branches. Predicated execution provided only a small performance benefit for gcc (2.5%). However this result was considered promising because the limitations of this study forced us to consider only a very small subset of gcc's hard-to-predict branches for elimination. The predicated version for sc showed little improvement in performance. Because profiling was not effective in finding the hard-to-predict branches for sc, predicated execution was not able provide much performance benefit.

9.2 Future Directions

This thesis examined one model of branch classification. Further improvements in branch handling mechanisms may be achieved by considering other branch classification models, e.g. partitioning branches based on the original boolean expressions in the source programs.

By combining multiple single-scheme branch predictors, hybrid branch predictors attempt to exploit the different strengths of different predictors. For this attempt to result in significant increases in prediction accuracy, the hybrid predictor must combine an appropriate set of single-scheme predictors and use an effective predictor selection mechanism. In determining the best single-scheme predictor combinations, this study assumed a static model of branch selection. Because of this assumption, this study did not fully exploit the capabilities of the hybrid predictor. Further work needs to be done to determine if these configurations are optimal for dynamic selection mechanisms as well.

For a given hardware cost, the hybrid predictor configuration specifies the classes of the single-scheme predictors used and the amount of hardware devoted to each scheme. This study considered only four types of single-scheme predictors – profile-guided predictor, 2bC, PAs, and gshare. In addition, for each predictor type, a limited range of predictor sizes was considered. Considering additional types and sizes of single-scheme predictors may result in better hybrid branch predictor designs.

Although the 2-level BPS mechanism introduced in this dissertation provides a performance increase over the 2-bit counter BPS mechanism, more effective BPS mechanisms are still required for the hybrid branch predictor to achieve its full potential performance. For one 16KB hybrid predictor, for example, the 2-level BPS achieves a 4.1% misprediction rate whereas an ideal selector would achieve a 2.1% misprediction rate, leaving significant room for improvement. In addition, this study considered hybrid branch predictors which dynamically combine only two single-scheme predictors. Hybrid branch predictors that combine more than two component predictors may achieve higher prediction accuracy.

For the target cache study, the input data for the experiments were the SPEC95 integer benchmarks where only a small fraction of instructions are indirect branches; e.g. 0.5% in gcc and 0.6% in perl. It will be interesting to study the benefits of target caches for object oriented programs where more indirect branches may be executed. For example, more indirect branches can lead to more interference in the target cache. The optimal configuration and/or design may be different when more indirect branches are executed.

We have shown that adding predicated execution to a machine whose ISA supports speculative execution can lead to significant increases in performance. However, more research can still be done to further increase the performance benefit provided by predicated execution. In particular, predicated execution's performance benefit would be significantly increased by compiler optimizations that transform the code so that more of the hard-to-predict branches could be eliminated. A compiler can also be built to automate branch elimination through predication and to evaluate the performance benefits of these optimizations as well.

APPENDICES

APPENDIX A

Branch Classification

A.1 GAs

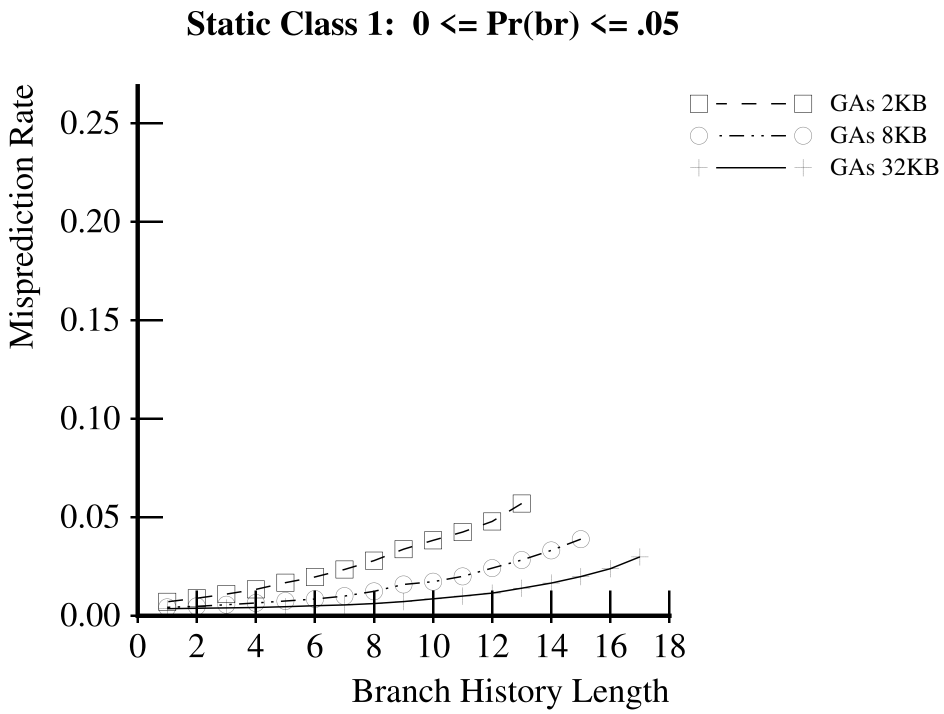


Figure A.1: Misprediction rate of GAs on SC1 branches

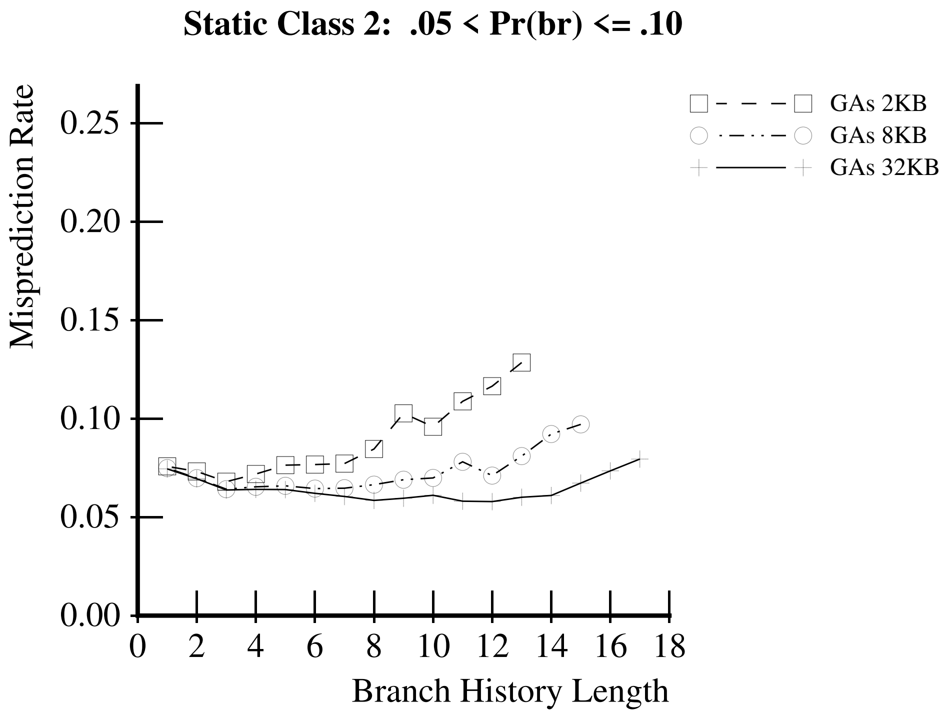


Figure A.2: Misprediction rate of GAs on SC2 branches

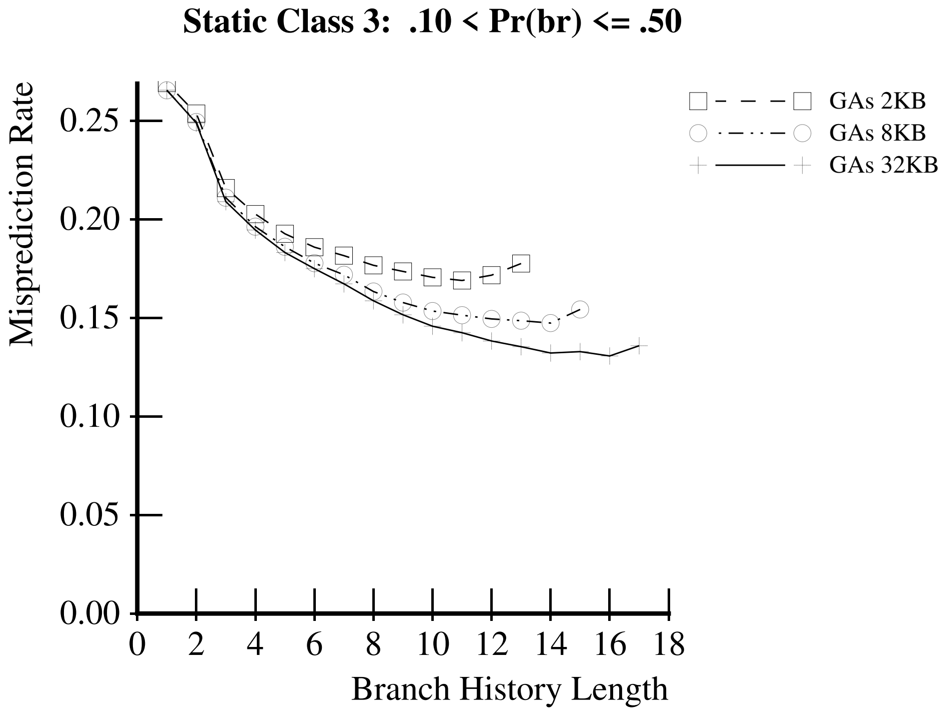


Figure A.3: Misprediction rate of GAs on SC3 branches

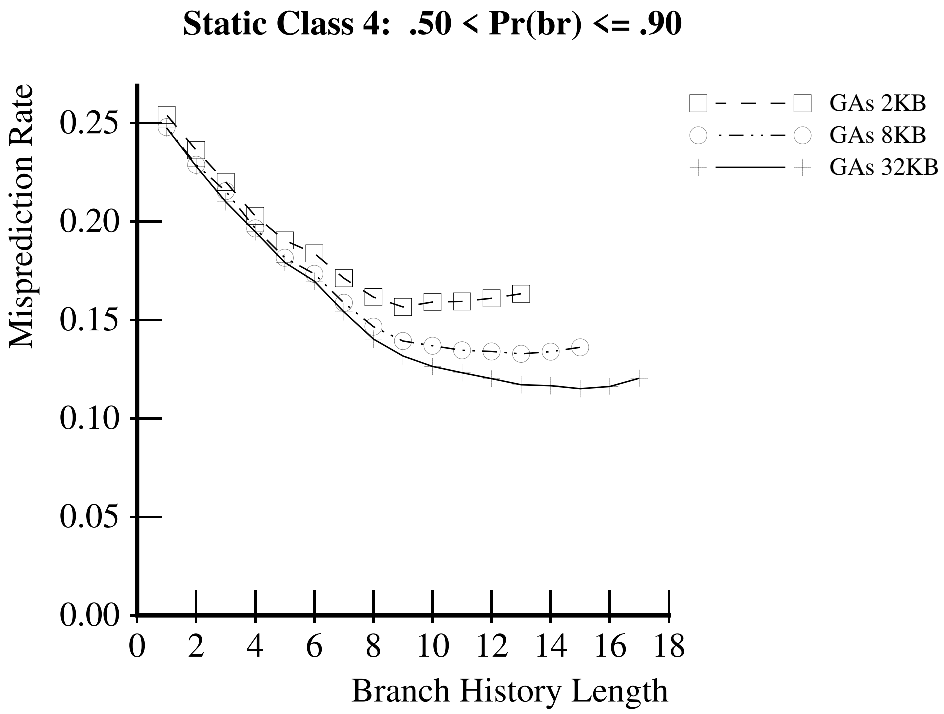


Figure A.4: Misprediction rate of GAs on SC4 branches

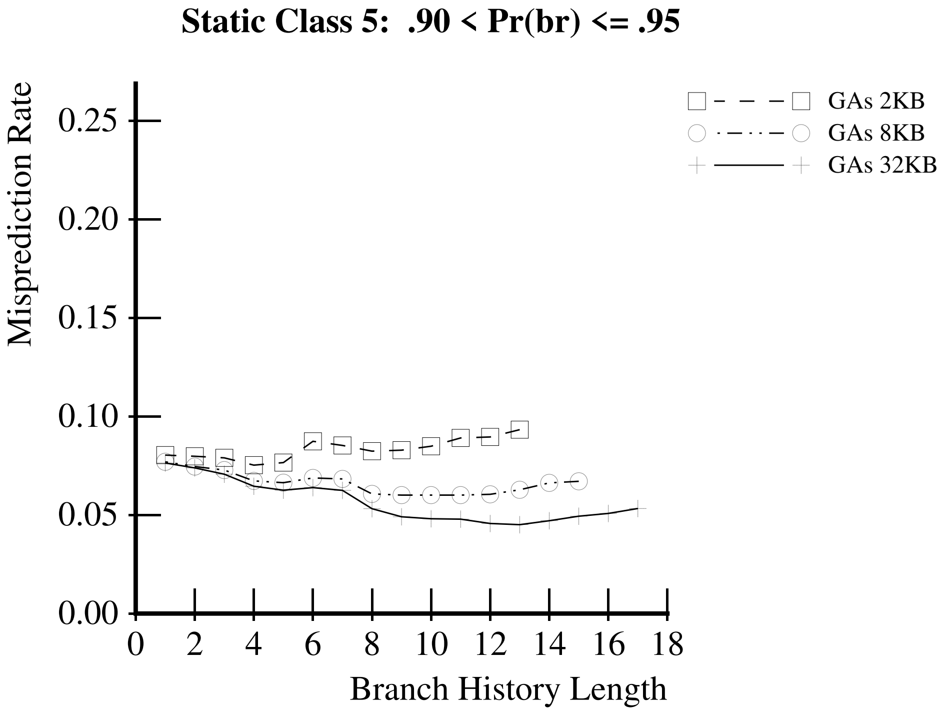


Figure A.5: Misprediction rate of GAs on SC5 branches

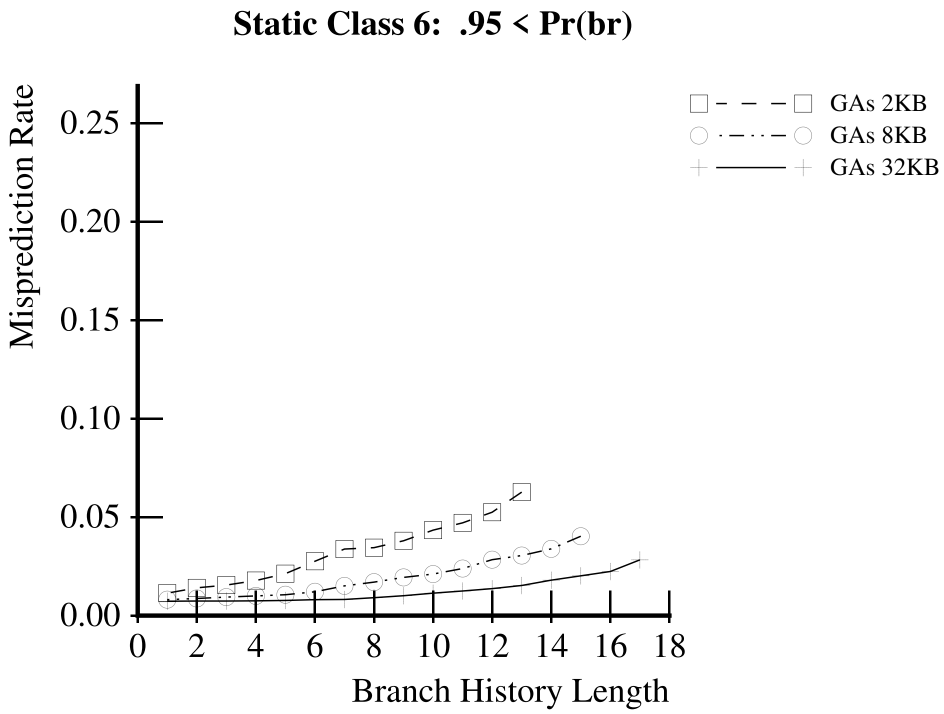


Figure A.6: Misprediction rate of GAs on SC6 branches

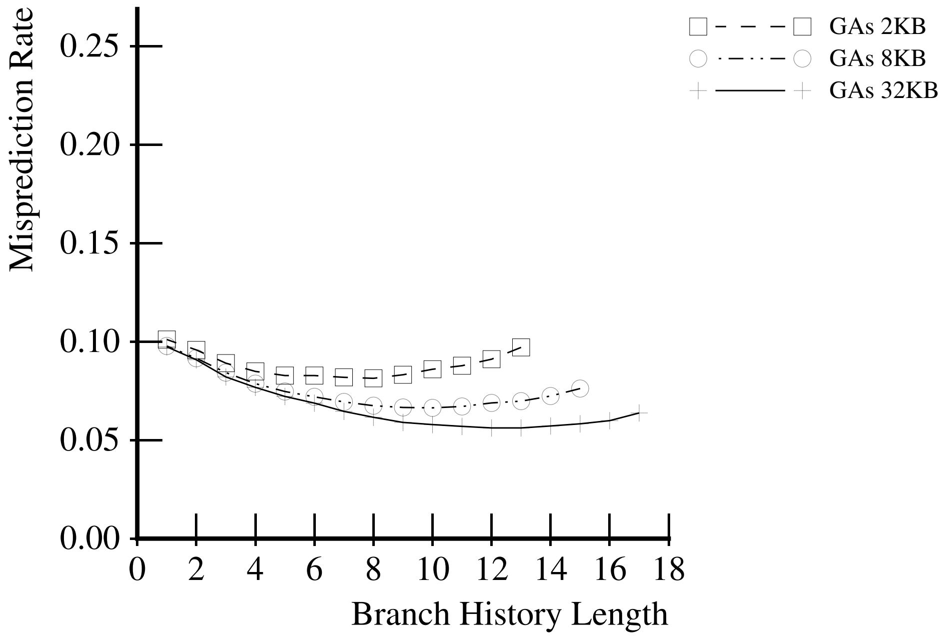


Figure A.7: Performance of GAs with different branch history length

A.2 gshare

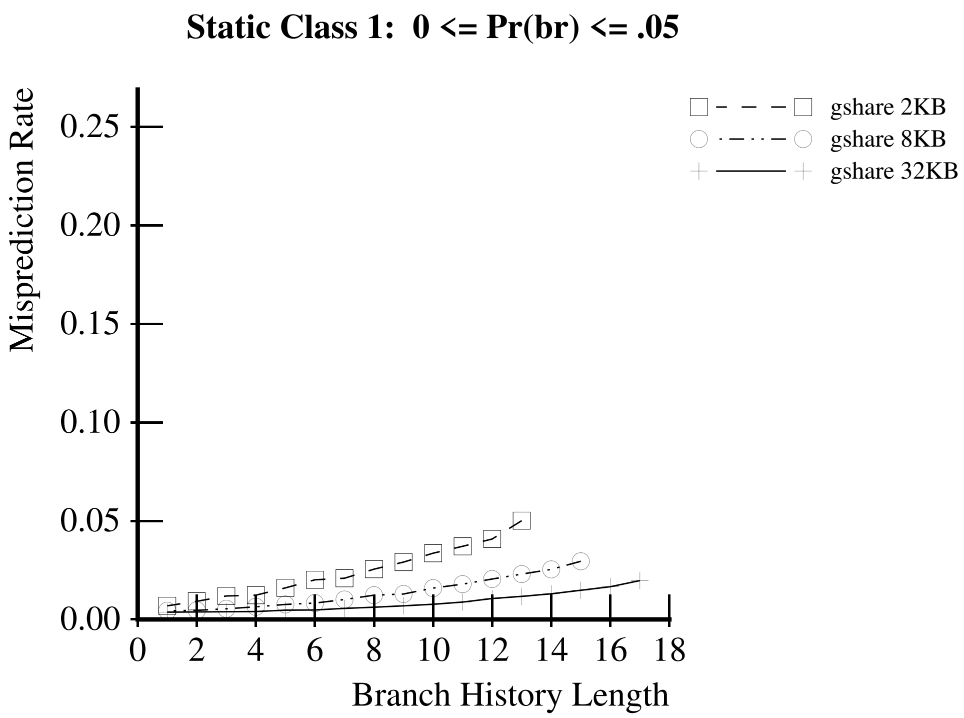


Figure A.8: Misprediction rate of gshare on SC1 branches

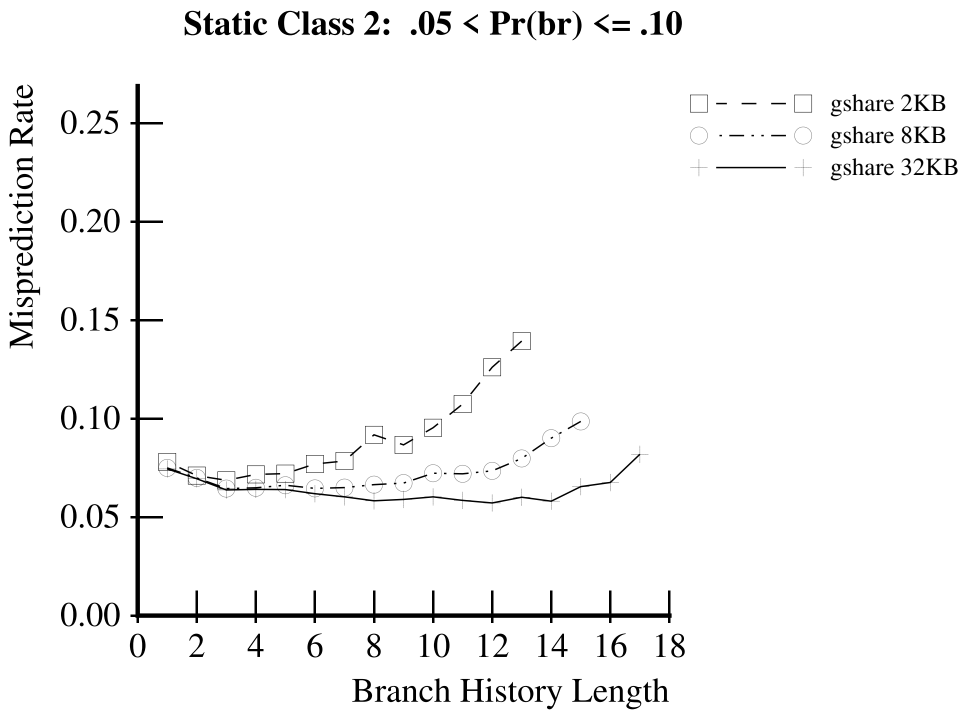


Figure A.9: Misprediction rate of gshare on SC2 branches

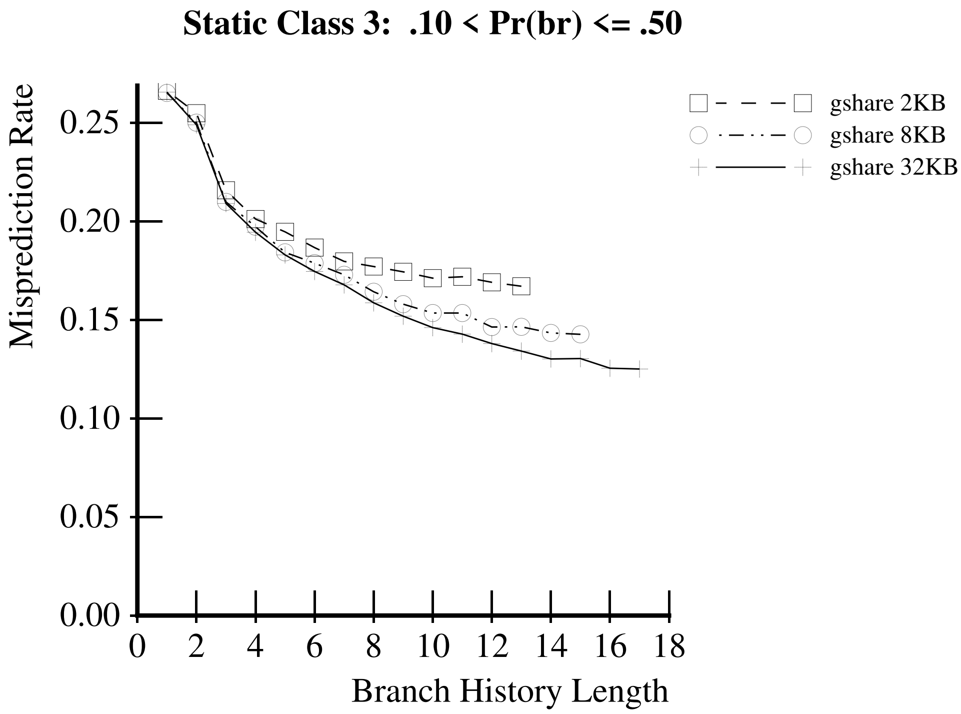


Figure A.10: Misprediction rate of gshare on SC3 branches

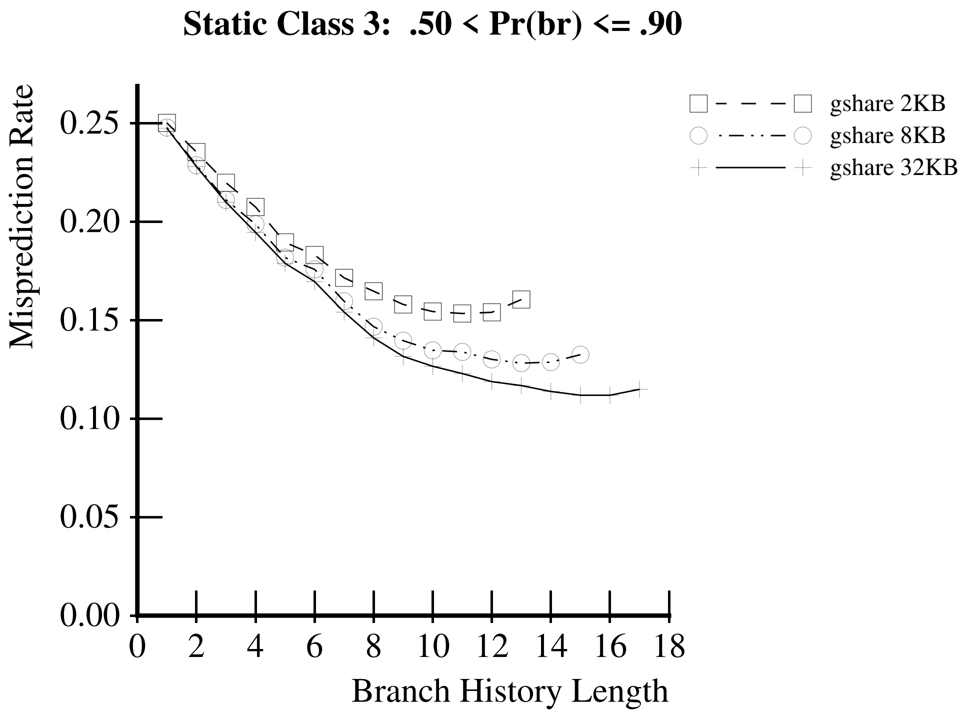


Figure A.11: Misprediction rate of gshare on SC4 branches

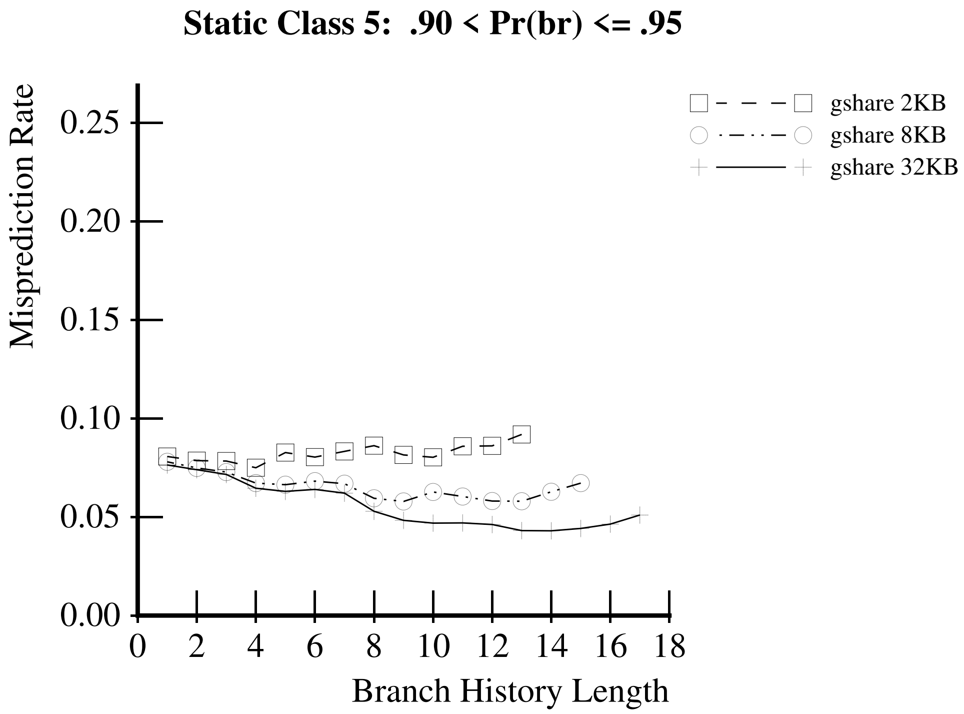


Figure A.12: Misprediction rate of gshare on SC5 branches

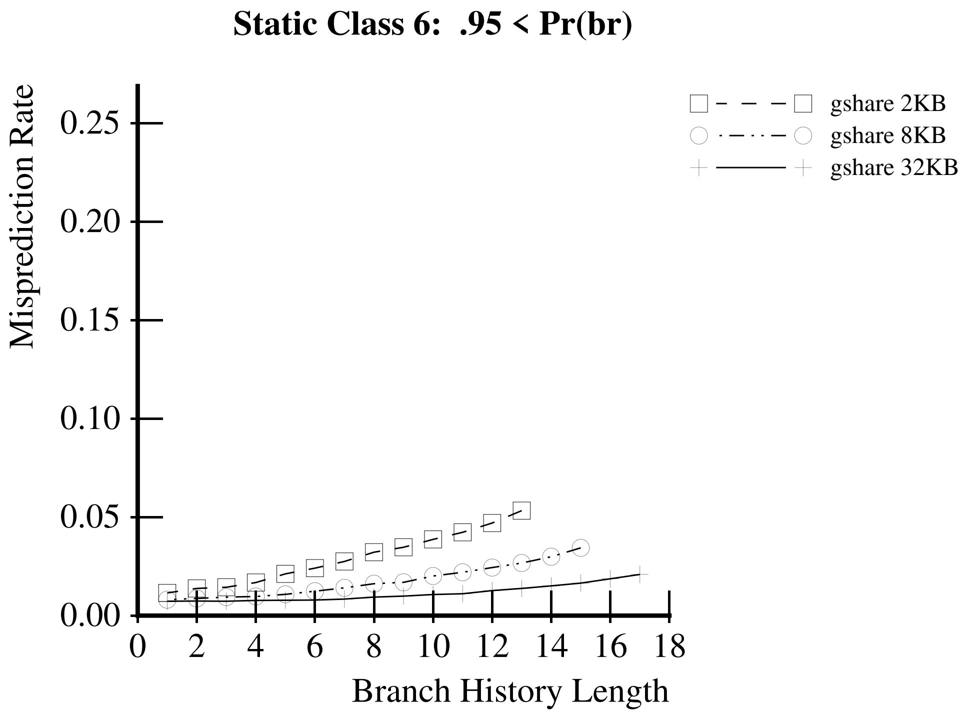


Figure A.13: Misprediction rate of gshare on SC6 branches

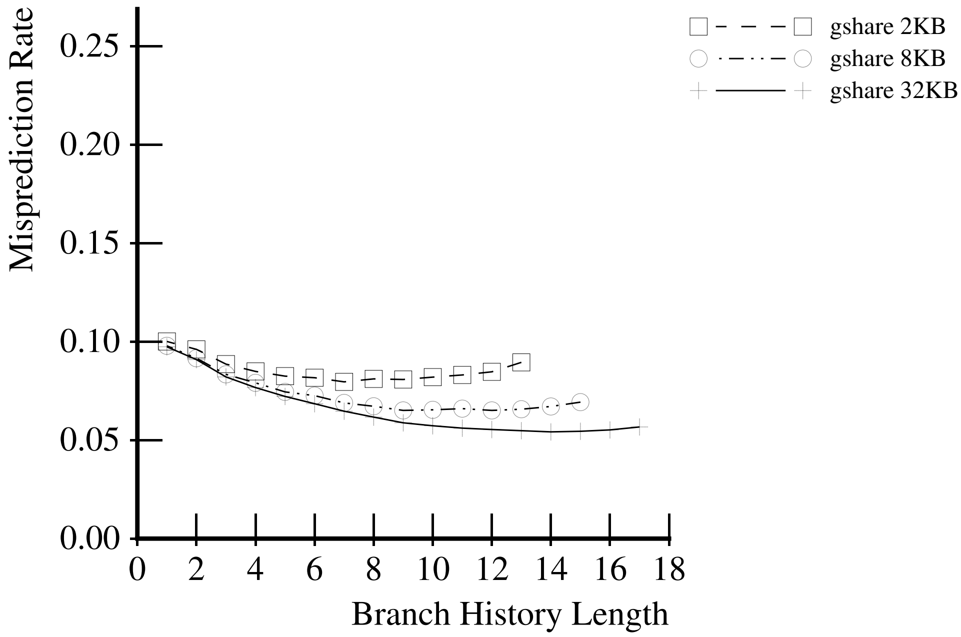


Figure A.14: Performance of gshare with different branch history length

A.3 PAs

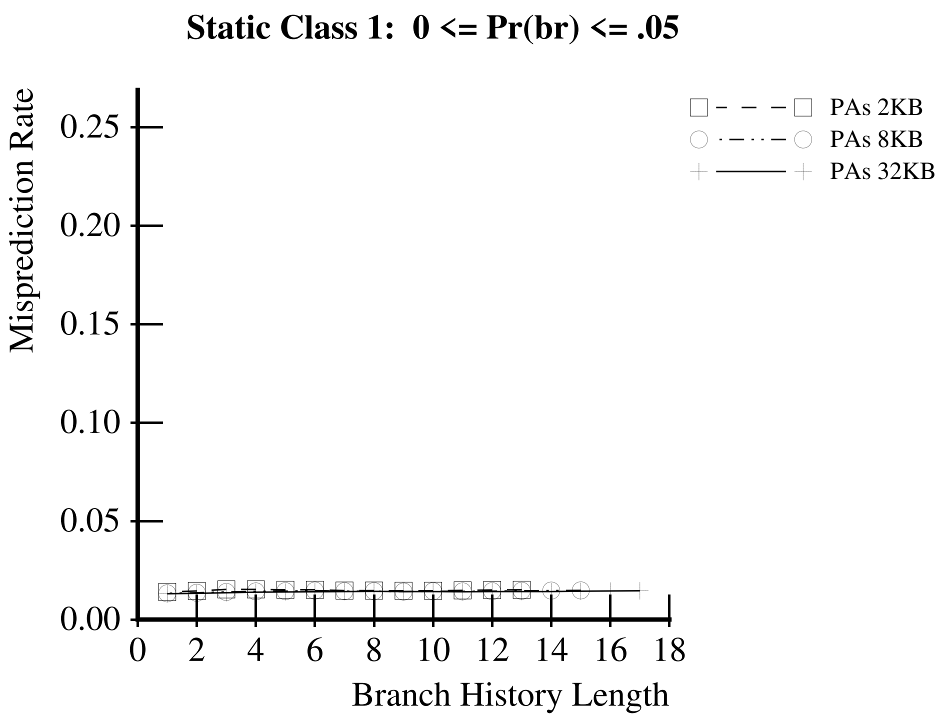


Figure A.15: Misprediction rate of PAs on SC1 branches

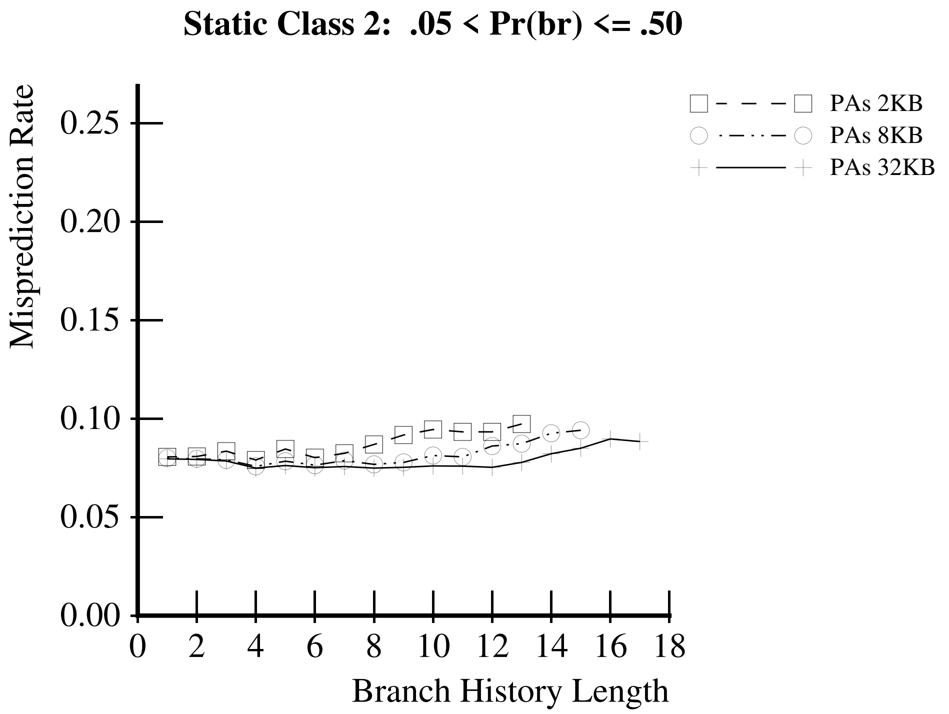


Figure A.16: Misprediction rate of PAs on SC2 branches

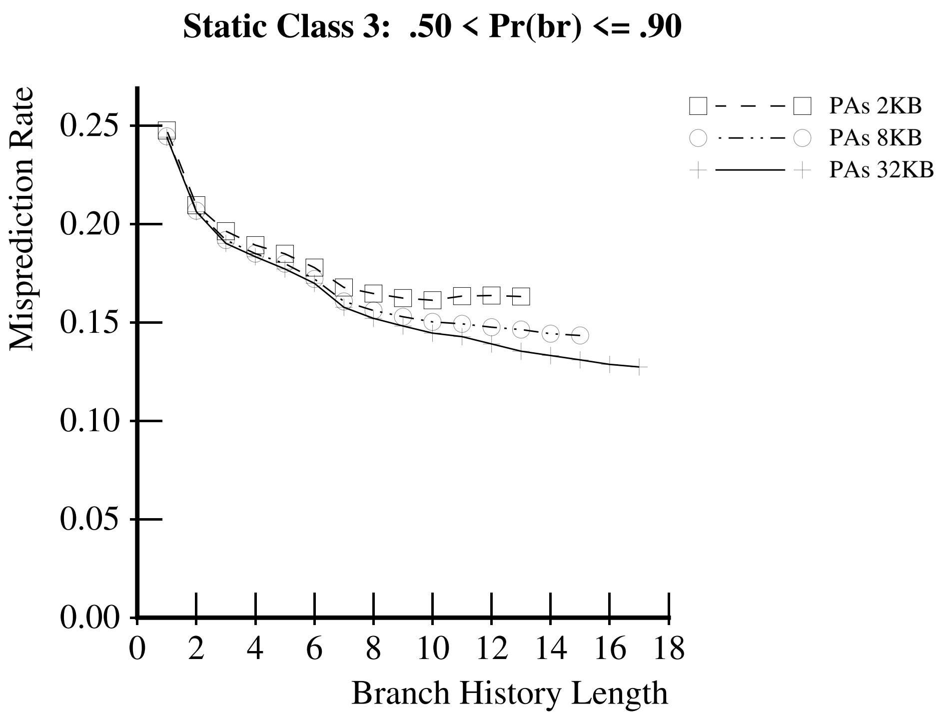


Figure A.17: Misprediction rate of PAs on SC3 branches

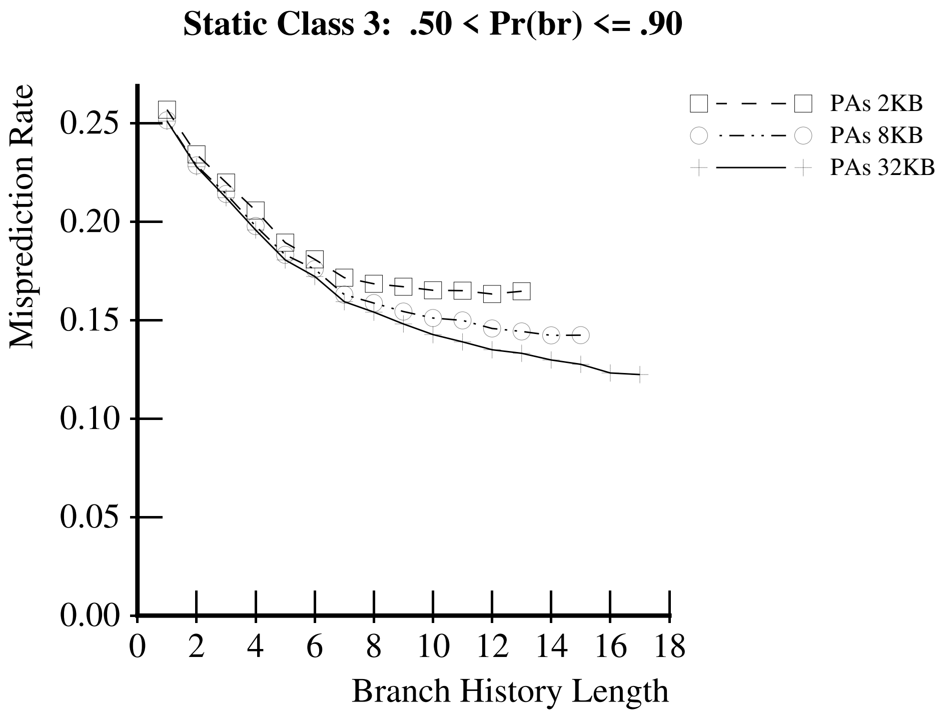


Figure A.18: Misprediction rate of PAs on SC4 branches

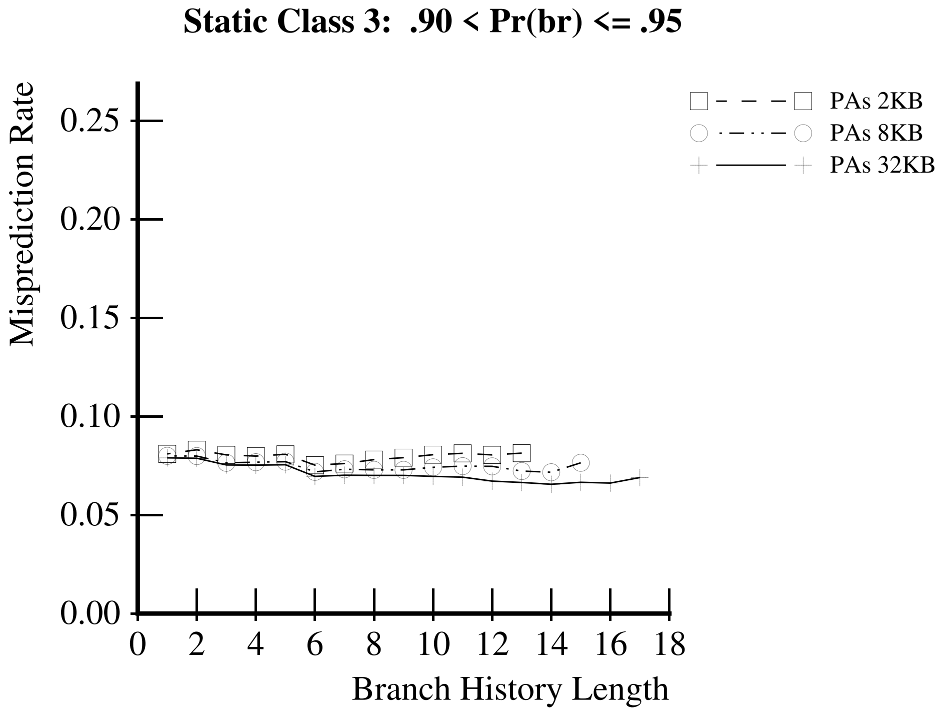


Figure A.19: Misprediction rate of PAs on SC5 branches

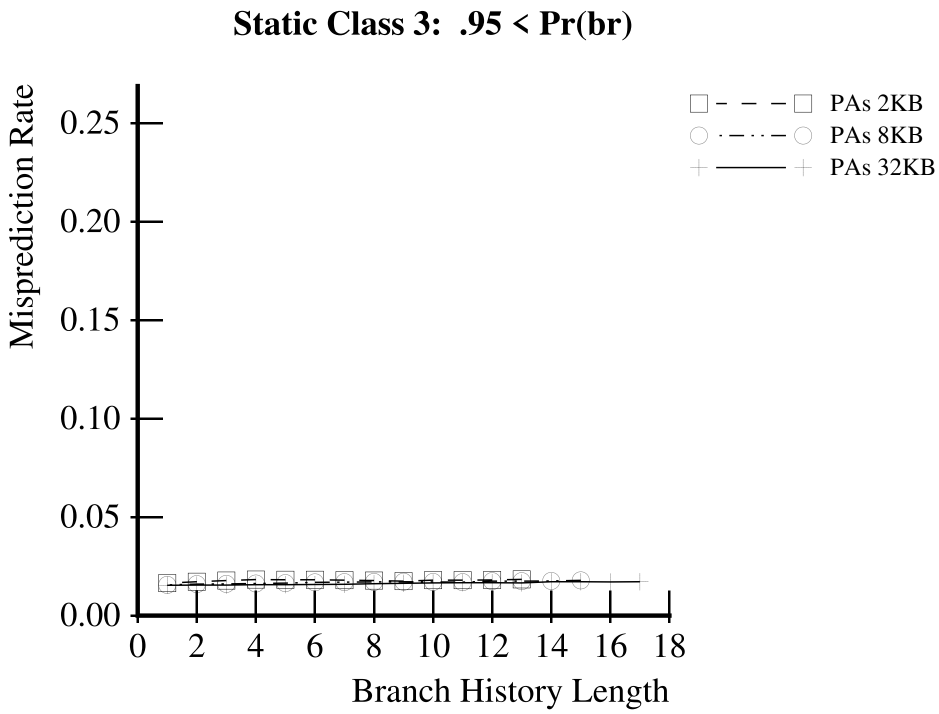


Figure A.20: Misprediction rate of PAs on SC6 branches

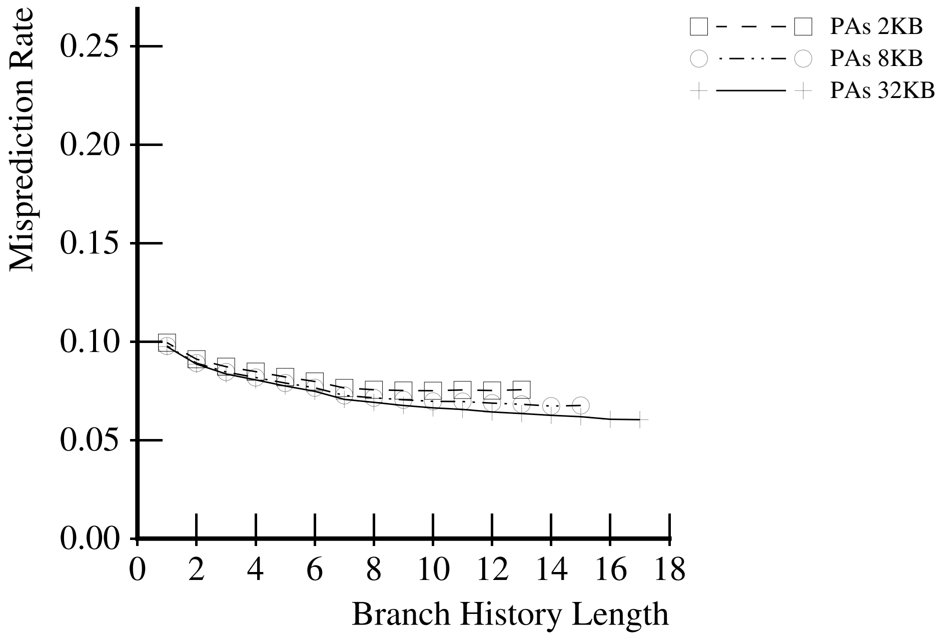


Figure A.21: Performance of PAs with different branch history length

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers Principles, Techniques, and Tools*, Addison Wesley Publishing Company, 1986.

[2] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren, "Conversion of control dependence to data dependence," in *10th Annual ACM Symposium on Principles of Programming Languages*, pp. 177–189, 1983.

[3] T. Ball and J. R. Larus, "Branch prediction for free," Technical Report 1137, Computer Sciences Department, University of Wisconsin - Madison, February 1993.

[4] M. G. Butler, *Aggressive Execution Engines for Surpassing Single Basic Block Execution*, PhD thesis, University of Michigan, 1993.

[5] B. Calder and D. Grunwald, "Reducing indirect function call overhead in c++ programs," in *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, 1994.

[6] P. Chang and U. Banerjee, "Profile-guided multi-heuristic branch prediction," in *Proceedings of the International Conference on Parallel Processing*, 1995.

[7] P.-Y. Chang, E. Hao, T.-Y. Yeh, and Y. N. Patt, "Branch classification: A new mechanism for improving branch predictor performance," *International Journal of Parallel Programming*, vol. 24, no. 2, pp. 133–158, 1996. Previous version published in Proceedings of the 27th ACM/IEEE International Symposium on Microarchitecture, Nov. 1994.

[8] P. Chow and M. Horowitz, "Architecture tradeoffs in the design of mips-x," in *Proceedings of the 14th Annual International Symposium on Computer Architecture*, 1987.

[9] J. C. Dehnert, P. Y. T. Hsu, and J. P. Bratt, "Overlapped loop support in the Cydra 5," in *Proceedings of the 3th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 26–38, 1989.

[10] E. C. M. et al, editor, *The PowerPC Architecture: A Specification for a New Family of RISC Processors*, Morgan Kaufmann Publishers, Inc., 1994.

[11] J. A. Fisher and S. M. Freudenberger, "Predicting conditional branch directions from previous runs of a program," in *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 85–95, 1992.

[12] M. L. Golden, *Reducing the Penalty of Branch and Load Hazards in Pipelined Microprocessors*, PhD thesis, University of Michigan, 1995.

[13] T. Granlund and R. Kenner, "Eliminating branches using a superoptimizer and the GNU C compiler," in *Proceedings of the ACM SIGPLAN'92 Conference on Programming Language Design and Implementation*, pp. 341–352, 1992.

[14] P. Hsu and E. Davidson, "Highly concurrent scalar processing," in *Proceedings of the 13th Annual International Symposium on Computer Architecture*, 1986.

[15] W. W. Hwu and Y. N. Patt, "Checkpoint repair for out-of-order execution machines," in *Proceedings of the 14th Annual International Symposium on Computer Architecture*, pp. 18–26, 1987.

[16] D. R. Kaeli and P. G. Emma, "Improving the accuracy of history-based branch prediction," Submitted to IEEE Transactions on Computers, 1994.

[17] J. K. F. Lee and A. J. Smith, "Branch prediction strategies and branch target buffer design," *IEEE Computer*, pp. 6–22, January 1984.

[18] S. A. Mahlke, R. E. Hank, R. A. Bringmann, J. C. Gyllenhaal, D. M. Gallagher, and W. W. Hwu, "Characterizing the impact of predicated execution on branch prediction," in *Proceedings of the 27th Annual ACM/IEEE International Symposium on Microarchitecture*, pp. 217–227, 1994.

[19] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann, "Effective compiler support for predicated execution using the hyperblock," in *Proceedings of the 25th Annual ACM/IEEE International Symposium on Microarchitecture*, pp. 45–54, 1992.

[20] S. McFarling, "Combining branch predictors," Technical Report TN-36, Digital Western Research Laboratory, June 1993.

[21] S. McFarling and J. Hennessy, "Reducing the cost of branches," in *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pp. 396–403, 1986.

[22] C. Melear, "The design of the 88000 risc family," in *IEEE Micro*, pp. 26–38, 1989.

[23] S. Melvin and Y. N. Patt, "Exploiting fine-grained parallelism through a combination of hardware and software techniques," in *Proceedings of the 18th Annual International Symposium on Computer Architecture*, pp. 287–297, 1991.

[24] R. Nair, "Dynamic path-based branch correlation," in *Proceedings of the 28th Annual ACM/IEEE International Symposium on Microarchitecture*, pp. 15–23, 1995.

[25] S.-T. Pan, K. So, and J. T. Rahmeh, "Improving the accuracy of dynamic branch prediction using branch correlation," in *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 76–84, 1992.

[26] Y. Patt, W. Hwu, and M. Shebanow, "HPS, a new microarchitecture: Rationale and introduction," in *Proceedings of the 18th Annual ACM/IEEE International Symposium on Microarchitecture*, pp. 103–107, 1985.

[27] Y. N. Patt, S. W. Melvin, W. Hwu, and M. C. Shebanow, "Critical issues regarding HPS, a high performance microarchitecture," in *Proceedings of the 18th Annual ACM/IEEE International Symposium on Microarchitecture*, pp. 109–116, 1985.

[28] D. N. Pnevmatikatos and G. S. Sohi, "Guarded execution and dynamic branch prediction in dynamic ILP processors," in *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pp. 120–129, 1994.

[29] B. R. Rau, D. W. L. Yen, W. Yen, and R. A. Towle, "The Cydra 5 departmental supercomputer," *IEEE Computer*, vol. 22, pp. 12–35, January 1989.

[30] R. M. Russell, "The CRAY-1 computer system," *Communications of the ACM*, vol. 21, no. 1, pp. 63–72, January 1978.

[31] S. Sechrest, C.-C. Lee, and T. Mudge, "The role of adaptivity in two-level adaptive branch prediction," in *Proceedings of the 28th Annual ACM/IEEE International Symposium on Microarchitecture*, 1995.

[32] J. E. Smith, "A study of branch prediction strategies," in *Proceedings of the 8th Annual International Symposium on Computer Architecture*, pp. 135–148, 1981.

[33] E. Sprangle and Y. Patt, "Facilitating superscalar processing via a combined static/dynamic register renaming scheme," in *Proceedings of the 27th Annual ACM/IEEE International Symposium on Microarchitecture*, pp. 143–147, 1994.

[34] A. R. Talcott, M. Nemirovsky, and R. C. Wood, "The influence of branch prediction table interference on branch prediction scheme performance," in *International Conference on Parallel Architectures and Compilation Techniques*, 1995.

[35] R. M. Tomasulo, "An efficient algorithm for exploiting multiple arithmetic units," *IBM Journal of Research and Development*, vol. 11, pp. 25–33, January 1967.

[36] G. S. Tyson, "The effects of predication on branch prediction," in *Proceedings of the 27th Annual ACM/IEEE International Symposium on Microarchitecture*, pp. 196–206, 1994.

[37] T.-Y. Yeh, *Two-level adaptive branch prediction and instruction fetch mechanisms for high performance superscalar processors*, PhD thesis, University of Michigan, 1993.

[38] T.-Y. Yeh and Y. N. Patt, "Two-level adaptive branch prediction," in *Proceedings of the 24th Annual ACM/IEEE International Symposium on Microarchitecture*, pp. 51–61, 1991.

[39] T.-Y. Yeh and Y. N. Patt, "Alternative implementations of two-level adaptive branch prediction," in *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pp. 124–134, 1992.

[40] T.-Y. Yeh and Y. N. Patt, "A comparison of dynamic branch predictors that use two levels of branch history," in *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pp. 257–266, 1993.

[41] C. Young, N. Gloy, and M. D. Smith, "A comparative analysis of schemes for correlated branch prediction," in *Proceedings of the 22st Annual International Symposium on Computer Architecture*, pp. 276–286, 1995.

[42] C. Young and M. D. Smith, "Improving the accuracy of static branch prediction using branch correlation," in *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 232–241, 1994.