Improving Branch Prediction by Understanding Branch Behavior

by

Marius Evers

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
2000

Doctoral Committee:
Professor Yale N. Patt, Chair
Professor Richard B. Brown
Professor Edward S. Davidson
Assistant Professor Steven Reinhardt
Tse-Yu Yeh, Consulting Engineer, SiByte

ABSTRACT

Improving Branch Prediction
by Understanding Branch Behavior

by
Marius Evers

Chair: Yale N. Patt

Accurate branch prediction can be seen as a mechanism for enabling design decisions. When short pipelines were the norm, accurate branch prediction was not as important. However, having accurate branch prediction enables technologies like wide-issue deeply pipelined superscalar processors. If branch predictors can be improved further, we can more successfully use more aggressive speculation techniques. Accurate branch prediction enables larger scheduling windows, out-of-order fetch, deeper pipelines etc. It is therefore likely that there will be a growing demand for more accurate predictors beyond today's prediction technology.

Previous studies have shown which branch predictors and configurations best predict the branches in a given set of benchmarks. Some studies have also investigated effects, such as pattern history table interference, that can be detrimental to the performance of these branch predictors. However, little research has been done on which characteristics of branch behavior make branches predictable.

This dissertation approaches the branch problem in a different way from previous studies. The focus is on understanding how branches behave and why they are predictable. Branches are classified based on the type of behavior, and the extent of each type of behavior is quantified. One important result is that two thirds of all branches are very predictable using a simple predictor because they follow repeating patterns. We also show how correlation between branches works, and what part of this correlation is important for branch prediction.

Based on this information about branch behavior, some shortcomings of current branch predictors are identified, new branch predictors are introduced, and potential areas for future improvement are identified. One of the new predictors, Dual History Length Gshare with Selective Update is more accurate than a Gshare predictor using Branch Filtering while having a simpler implementation. Another new predictor, the Multi Hybrid, suffers 10% fewer mispredictions than a state-of-the-art PAs/Gshare hybrid predictor at an implementation cost of 100 KB.

© Marius Evers 2000
All Rights Reserved

Til Moi

ACKNOWLEDGEMENTS

I would like to thank the many people that have helped and guided me throughout my research. Without these people, I could not have completed this work. I am grateful to all of you.

First, special thanks go to my advisor, Yale Patt, for convincing me to go to graduate school, and for the guidance and advice he has provided. I would also like to thank the rest of my committee; Richard Brown, Edward Davidson, Steven Reinhardt, and Tse-Yu Yeh; both for the valuable feedback they provided on my dissertation and for all that I have learned from them in and out of classes.

I would like to thank all the members of the HPS group for the stimulating environment they have provided. Much of this research has benefited greatly from the interaction with other members of the group. In particular I want to thank: Eric Hao for guiding me through the early stages of my research, Po-Yung Chang for collaborating on several projects, Sanjay Patel for always providing valuable insights and for showing up to my thesis defense when least expected, and Jared Stark for providing crucial support in the final stages of the dissertation process.

Finally, I would like to thank Intel Corporation and HAL Computer Systems for providing me with summer jobs that helped me gain valuable industry experience related to this dissertation.

TABLE OF CONTENTS

DEDICATION ii

ACKNOWLEDGEMENTS iii

LIST OF TABLES vii

LIST OF FIGURES ix

CHAPTERS

1 Introduction 1
  1.1 The Branch Problem 1
  1.2 Understanding Branch Behavior 2
  1.3 Thesis Statement 3
  1.4 Contributions of This Dissertation 3
  1.5 Organization of This Dissertation 4

2 Related Work 6
  2.1 Hardware Branch Prediction 6
  2.2 Assisting Branch Prediction Using Profiling 12
  2.3 Branch Behavior and Effects Seen in Branch Predictors 13

3 Simulation Methodology 16
  3.1 Simulation Environment 16
  3.2 The SPECint95 Benchmarks 17

4 Evaluation of Basic Branch Properties and Predictors 19
  4.1 Branch Properties 19
  4.2 Basic Branch Predictors 24
  4.3 Two-Level Adaptive Branch Predictors 25
    4.3.1 Branch History Register Length and Number of Pattern History Tables 25
    4.3.2 Effect of Branch History Table Size 27
    4.3.3 Comparison of Two-Level Branch Prediction Schemes 27
  4.4 PHT Interference and Interference Reduction Schemes 29
    4.4.1 Reducing Interference Via Branch Classification or Filtering 32
  4.5 Hybrid Branch Predictors 33
  4.6 Summary of Branch Predictor Performance 35

5 Self Correlation . . . 36
5.1 Classes of Patterns and Their Characteristics . . . 36
5.1.1 Classes of Patterns . . . 37
5.1.2 Classifying a Branch Instance Using Complete History . . . 38
5.1.3 Distribution of Branches by Pattern Class . . . 41
5.1.4 Characteristics of Biased Patterns . . . 44
5.1.5 Characteristics of Repeating Patterns . . . 47
5.1.6 Summary . . . 58
5.2 Pattern Usage in Per-Address Predictors . . . 58
5.2.1 Classifying a Branch Instance Using Limited History . . . 59
5.2.2 Distribution of Patterns and General Statistics . . . 60
5.2.3 Classifying Using Limited vs. Complete History . . . 62
5.2.4 Effect of Lengthening PAs History . . . 63
5.2.5 Role of Adaptivity in Per-Address Prediction . . . 67
5.2.6 Summary . . . 68
5.3 Methodology for Simulating Per-Address Predictors . . . 69
5.4 Improving Prediction using Per-Address History . . . 69
5.4.1 Mixed Length History . . . 71
5.4.2 Skewed Per-Address Prediction . . . 72
5.4.3 Other Approaches . . . 74
5.5 Improving Prediction by Filtering Repeating Patterns . . . 74
5.5.1 Implementation . . . 75
5.5.2 Results . . . 78
5.6 Summary . . . 80

6 Branch Correlation . . . 82
6.1 What is Branch Correlation . . . 82
6.2 Correlation in Two-Level Branch Predictors . . . 85
6.3 Correlation Using a Selective History . . . 86
6.4 Identifying Correlation in a Simulator . . . 86
6.4.1 General Case . . . 87
6.4.2 Complexities Introduced by Loops . . . 88
6.4.3 How the Best Set of Correlated Branches is Detected . . . 89
6.5 Understanding the Nature of Correlation . . . 89
6.5.1 How Many Correlated Branches are Needed . . . 90
6.5.2 Potential for Improving Gshare with Correlation . . . 94
6.5.3 Distance to Correlated Branches . . . 95
6.5.4 Correlation Across Subroutine Calls . . . 96
6.5.5 Correlation Across Loops . . . 99
6.5.6 Correlation with Subroutine Callpoints . . . 102
6.5.7 Dependence of Correlation on Input Set . . . 103
6.5.8 Comparison of Correlation Based Methods . . . 104
6.6 Improving Branch Correlation Based Prediction . . . 106
6.6.1 Dual History Length Gshare with Selective Update . . . 108
6.6.2 Dual History Length Gshare with Selective Update and Return History Stack . . . 112
6.7 Summary . . . 116

7 Interaction Between Branch Predictors 117
7.1 Fundamentals 118
7.1.1 Importance of Each Single Scheme Predictor 118
7.1.2 Stability Over Time 123
7.1.3 Stability Between Input Sets 124
7.2 Selection Mechanisms 127
7.2.1 Basic and Idealized Selection Mechanisms 127
7.2.2 Size and History in Predictor Selectors 130
7.3 Improving Prediction using a Multiple Component Hybrid Branch Predictor 134
7.3.1 Implementation 135
7.3.2 Results 137
7.4 Summary 140

8 Conclusions 142
8.1 Contributions 142
8.2 Future Directions 145

BIBLIOGRAPHY 147

LIST OF TABLES

Table

3.1 Description of benchmarks and their input sets . . . 17
3.2 Execution statistics for test data sets . . . 18
4.1 Configurations used for two-level adaptive branch predictors . . . 26
4.2 Misprediction rates of 64 KB predictors . . . 29
4.3 PHT interference per 100,000 accesses for Gshare . . . 30
4.4 PHT interference per 100,000 accesses for PAs . . . 31
4.5 Configurations used for Branch Classification and Branch Filtering . . . 32
5.1 Determining repeating pattern type . . . 40
5.2 Pattern breakdown for 16 K entry BTB . . . 42
5.3 Pattern breakdown for 2 K entry BTB . . . 43
5.4 Frequency and accuracy of patterns by region . . . 50
5.5 General statistics for PAs patterns . . . 60
5.6 Loop patterns vs. PAs patterns . . . 62
5.7 Effect of adding history bits by pattern type . . . 64
5.8 Frequency and utilization of PAs patterns by history length . . . 66
5.9 The role of adaptivity in a PAs predictor . . . 67
5.10 Thresholds for the loop filtering mechanism . . . 77
5.11 Misprediction rate of Gshare, PAs, and PAs/Gshare w/filter and loop filter 79
6.1 Selective history vs. Gshare by region . . . 93
6.2 Potential for improving Gshare and interference free Gshare with selective history . . . 94
6.3 Correlation across subroutine calls . . . 98
6.4 Correlation across loop bodies . . . 101
6.5 Callpoint vs. selective history by region . . . 103
6.6 Identifying most correlated branches using profiling . . . 103
6.7 Comparison of all correlation based methods . . . 106
6.8 Configurations for Dual History Length Gshare with Selective Update . . . 110
6.9 Configurations for Dual History Length Gshare with Selective Update and Return History Stack . . . 115
7.1 Difference between accuracy of best predictor and accuracy of second best predictor . . . 120
7.2 Difference between accuracy of best predictor and accuracy of second best predictor (interference free) . . . 122

7.3 Probability of best predictor for a branch changing over time 123
7.4 Increase in misprediction rate from selecting predictor using profiling 126
7.5 Misprediction rates for PAs/Gshare for 4 selection mechanisms 128
7.6 Misprediction rates for six-component hybrid for 4 selection mechanisms . . 129
7.7 Best history lengths for PAs/Gshare selectors 131
7.8 Best history lengths for six component hybrid selector 133
7.9 Best sizes and history lengths for PAs/Gshare selectors 133
7.10 Configurations for Multi-Hybrid 137

LIST OF FIGURES

Figure

2.1 Diagram of a GAs two-level predictor 8
2.2 Diagram of a hybrid branch predictor 10
4.1 Weighted distribution of branches by type 20
4.2 Distribution of static branches by execution frequency 21
4.3 Weighted distribution of static branches by execution frequency 21
4.4 Weighted distribution of static branches by taken rate 22
4.5 Change in taken rate between profiling and test runs 23
4.6 Accuracy of basic branch predictors 24
4.7 Effect of BHT size on misprediction rate of PAs 27
4.8 Comparison of two-level predictors 28
4.9 Improving Gshare using Branch Classification or Branch Filtering 33
4.10 Accuracy of hybrid branch predictors 34
5.1 Distribution of branches by pattern class. 16 K entry BTB 42
5.2 Distribution of branches by pattern class. 2 K entry BTB 43
5.3 Characteristics of transient biased patterns 45
5.4 Characteristics of stable biased patterns 46
5.5 Frequency of repeating patterns 48
5.6 Prediction accuracy of repeating patterns 49
5.7 Frequency, accuracy, and best predictor. For-type patterns 51
5.8 Improvement over and interference for Gshare, improvement over PAs. For-type patterns 53
5.9 Frequency, accuracy, and best predictor. While-type patterns and alternating patterns 55
5.10 Improvement over and interference for Gshare, improvement over PAs. While-type patterns and alternating patterns 57
5.11 Diagram of per-address mixed length history predictor 72
5.12 Loop filter performance on Gshare 80
6.1 Correlation example 1 83
6.2 Correlation example 2 83
6.3 Correlation example 3 84
6.4 Correlation example 4 85
6.5 Fields in record for each correlated branch 87
6.6 Selective history vs. Gshare 90

6.7 Weighted distribution of branches by best predictor 91
6.8 Misprediction rate of Selective History predictor vs. history length 95
6.9 Frequency of branches with correlation across subroutine calls 97
6.10 Example of correlation across subroutine calls in go 98
6.11 Frequency of branches with correlation across loop bodies 100
6.12 Frequency of branches correlated with the callpoint 102
6.13 Distribution of branches by best correlation based predictor 105
6.14 Dual History Length Gshare with Selective Update 109
6.15 Performance of Dual History Length Gshare with Selective Update 111
6.16 Selection Mechanism for a Dual History Length Gshare with Selective Update and Return History Stack 113
6.17 Performance of Dual History Length Gshare with Selective Update and Return History Stack 115
7.1 Distribution of branches by best predictor 119
7.2 Distribution of branches by best predictor (interference free) 122
7.3 Change in best predictor between profiling and test runs 125
7.4 PAs/Gshare selection mechanisms 130
7.5 Six component selection mechanisms 132
7.6 Multi-Hybrid 136
7.7 PAs/Gshare vs. Multi-Hybrid 138
7.8 PAs/Gshare vs. Multi-Hybrid by benchmark 139

CHAPTER 1

Introduction

1.1 The Branch Problem

Branch instructions are used to choose which path to follow through a program. Branches can be used to include a subroutine in several places in a program. They can be used to allow a loop body to be executed repeatedly, and they can be used to execute a piece of code only if some condition is met. Branches are integral to the function of most programs and appear frequently. It is estimated that every fifth[1] instruction that is executed is a branch.

Branches cause problems for processors for two reasons. Branches can change the flow through the program, so the next instruction is not always the instruction following sequentially after the branch. Branches can also be conditional, so it is not known until the branch is executed whether the next instruction is the next sequential or the instruction at the branch target.

In early processor designs, instructions were fetched and executed one at a time. By the time the fetch of a new instruction started, the target address and condition of a previous branch was already known. The processor always knew which instruction to fetch next. However, in pipelined processors, the execution of several instructions is overlapped. In a pipelined processor, the instruction following the branch needs to be fetched before the branch is executed. However, the next fetch address is not yet known. This problem is known as the branch problem.

If nothing is done about the branch problem, bubbles will be introduced into the pipeline

[1] In the benchmarks used in this dissertation, on average 19% of the instructions are branches.

after the branch. Since the target address and condition of the branch is not known until after the branch is executed, all pipeline stages before the execute stage will be filled with bubbles by the time the branch is ready to execute. If an instruction executes in the nth stage, there will be $n - 1$ bubbles per branch. Each of the bubbles represents the lost opportunity to execute an instruction.

In superscalar processors, the problem is more serious as each pipeline stage can hold several instructions. For a superscalar processor capable of executing k instructions per cycle, the number of bubbles is $(n-1) \times k$. Each bubble still represents the lost opportunity to execute an instruction. The number of cycles lost due to each branch is the same in the pipelined and superscalar processors, but the superscalar processor can do much more in that period of time. For example, consider a 4-way superscalar ($k = 4$) processor where branches are executed in the 6th pipeline stage ($n = 6$). If every fifth instruction is a branch instruction, there will be 20 bubbles for every 5 useful instructions executed. Due to the branch problem, only 20% of the execution bandwidth is used to execute instructions. The trend in processor design is towards wider issue and deeper pipelines, further aggravating the branch problem.

Branch prediction is one way of dealing with the branch problem. A branch predictor uses the current fetch address to predict whether a branch will be fetched this cycle, whether that branch will be taken or not, and what the target address of the branch is. The predictor uses this information to decide where to fetch from in the next cycle. When a branch predictor is used, the branch penalty is only seen if the branch is mispredicted. A highly accurate branch predictor is therefore a very important mechanism for reducing the branch penalty in a high performance microprocessor.

1.2 Understanding Branch Behavior

The ultimate goal of any work examining branches is to reduce the branch execution penalty. Branch prediction is one way of doing this. The goal of this dissertation is to examine branch behavior to identify in which ways branches are predictable, so that this information can be used to design better branch predictors.

A branch predictor works by guessing the next outcome of a branch. To do this correctly, the predictor must in some way understand how the branch is likely to behave, so it can

deduce what the likely next outcome is. However, the predictor will only be able to exploit the type of behavior that it has been designed to detect. This leaves it up to the branch predictor designer to understand the behavior of branches, and use this understanding to design better predictors.

Many studies investigate how to improve the mechanics of branch predictors by improving configurations, reducing interference or other means. What most of these studies have in common is that they try to more efficiently exploit the branch behavior we already know about. This is useful in itself, but to go one step further it is important to understand how branches behave. With an understanding of branch behavior, current predictors can be examined to see whether they capture this behavior. If current predictors do not capture a type of branch behavior, new predictors can be built to capture this behavior, and then the mechanics of the new predictors can be tuned.

1.3 Thesis Statement

As microprocessor pipelines get deeper and wider, the need for more accurate branch prediction grows. Understanding conditional branch behavior provides an important foundation for the design of better branch predictors. If you understand how a branch is likely to behave, you can design a better branch predictor for it.

In this dissertation correlation and branch execution patterns are examined to contribute to a better understanding of how branches behave and how they can be predicted. Branch behavior is classified and quantified, and it is shown that some of this behavior is not captured by existing predictors. Several new predictor designs are proposed to take advantage of the behavior that is seen. These predictors are more accurate than similar existing predictors.

1.4 Contributions of This Dissertation

- This dissertation contributes to a better understanding of how branches behave by classifying and quantifying how branches are predictable.

- Based on one type of branch behavior that is frequently seen, a new mechanism, the loop filter, that uses a specialized per-address history is introduced. It is shown that

this mechanism can improve the accuracy of existing predictors.

- Based on the observations about correlation in this dissertation, a new predictor, Dual History Length Gshare with Selective Update, is introduced. This predictor is shown to achieve lower misprediction rates than other comparable correlation based methods, while using simpler hardware than the closest competing predictors.
- Based on a study of how predictors interact, a hybrid branch predictor that uses more than two component predictors is introduced. This predictor is shown to have 7-11% fewer mispredictions than existing hybrid branch predictors for 54 to 188 KB predictors. Improvements are seen for all sizes larger than 18 KB.

1.5 Organization of This Dissertation

This dissertation is divided into 8 chapters. Chapter 2 describes related work. Chapter 3 describes the simulation environment and the benchmarks that are used. Chapter 4 reproduces previous studies on the simple properties of conditional branches, such as the taken rates and frequencies of branches. A selection of the branch prediction strategies proposed in previous studies are also evaluated to provide a baseline for the predictor improvements made in this dissertation.

The remaining part of the dissertation investigates branch behavior, the interaction between branches, and how this can be used to improve branch predictors. This is done in three chapters. Chapters 5 and 6 investigate two distinct classes of branch behavior, and Chapter 7 investigates the interaction between branch predictors. In each chapter, the behavior of branches (or interaction between the predictors exploiting different types of behavior) is examined first, followed by new predictors being proposed to take advantage of the information presented in that chapter.

Branch execution patterns are investigated in Chapter 5. These are the patterns that per-address two-level predictors and most simpler predictors partially exploit to make their predictions. These branch execution patterns are classified and quantified. Using this information a predictor is proposed that dynamically detects and predicts certain patterns with high accuracy, while using existing predictors for the remaining branches.

Correlation between branches is investigated in Chapter 6. Branch correlation is the

effect that global two-level predictors exploit to make their predictions. The nature of the correlation that makes branches predictable is investigated. It is shown how this correlation works, and what part of this correlation is important for branch prediction. Based on the results in this chapter, a new correlation based predictor is proposed and shown to achieve lower misprediction rates than other comparable correlation based methods.

Finally, Chapter 7 investigates ways to combine branch predictors to effectively predict branches whose behavior differs. The usefulness of different branch predictors as components in a hybrid branch predictor is examined, and a new hybrid branch predictor is proposed and shown to achieve lower misprediction rates than existing hybrid predictors.

Chapter 8 provides concluding remarks, and suggests future directions for branch prediction research.

CHAPTER 2

Related Work

2.1 Hardware Branch Prediction

Branch prediction consists of two parts. Predicting the target of a branch, and predicting the outcome, taken or not taken, of that branch. For immediate or PC-relative branches, the target address of the branch does not change between each time the branch is seen, so predicting the target address is merely a matter of caching it. The Branch Target Buffer (BTB) is a small cache that is accessed using the fetch address. An entry in the BTB stores a tag to identify whether there is a branch in the current fetch, and the target address of the branch. Optionally, the BTB may also store prediction information. Some of the design options for the BTB are evaluated in [19]. The BTB is accessed during the fetch cycle, and upon a hit provides the target address of the branch that is being fetched. If a branch misses in the BTB, there is a penalty equal to the time it takes to calculate the target address and the prediction for that branch must be made without using any prediction information that normally resides in the BTB. When a branch is retired, the BTB is updated with the new target information.

The simplest scheme for predicting the outcome of branches is to predict all branches to be either always not taken or always taken. These two schemes, although an improvement over using no prediction at all, are only 40-60% accurate and therefore not very effective. Technically, these schemes do not require any hardware to make a prediction. However, they are included in this section as they do not require compiler assistance. A variation of these schemes is to predict all backwards branches to be taken and forwards branches to be not taken. Most backwards branches are loops, and therefore taken the majority of the

time. This scheme works best if the compiler also orders the code such that the not taken path is the most likely for forward branches.

Branch prediction can be improved by adding dynamic state to the predictor. The simplest dynamic branch prediction scheme is to predict that a branch will have the same outcome as the previous time it was executed. This predictor, referred to as the Last-Time predictor [26], requires one bit per branch to store the outcome of the previous execution. To predict a branch, its dynamic prediction bit is examined. A "0" results in a not taken prediction, and a "1" results in a taken prediction. When the branch is retired, its bit is updated based on the outcome of the branch. The Last-Time scheme can be implemented by adding one bit for each instruction in the instruction cache or alternatively adding the bit to the BTB. The scheme can also be implemented using a separate table of these one-bit histories, where each branch is mapped to one of the bits using the least significant bits of the fetch address.[1]

The prediction accuracy can be improved by collecting more branch history using a 2-bit saturating counter [26]. One counter is used to collect the execution history for each branch. The counter is incremented by one whenever the branch is taken, and is decremented by one if the branch is not taken. The branch is predicted taken if the counter value is greater than or equal to 2, otherwise the branch is predicted not taken. We refer to this scheme as the Two-Bit Counter scheme. The 2-bit counters can be stored in the instruction cache, BTB, or in a separate table as in the Last-Time scheme. The 2-bit counter does not change its prediction based on a single outcome in the opposite direction. The Two-Bit Counter scheme therefore predicts loop branches, which occasionally have one outcome in the opposite direction, better than the Last-Time predictor.

More accurate predictions can be made by using two levels of history. Whereas the Last-Time and Two-Bit Counter schemes both try to predict the dominant direction of a branch, the Two-Level Adaptive [32, 33] predictor collects a first level history of execution outcomes, and then predicts the direction separately for each of the possible patterns in the first level history. The first level history is recorded in one or more k-bit shift registers, called branch history registers, which record the outcomes of the k most recent branches. The second level history is recorded in one or more tables of 2-bit saturating counters, called

[1] If instruction addresses are aligned on 4 byte boundaries, the two least significant bits will always be zero and are therefore not used for selection.

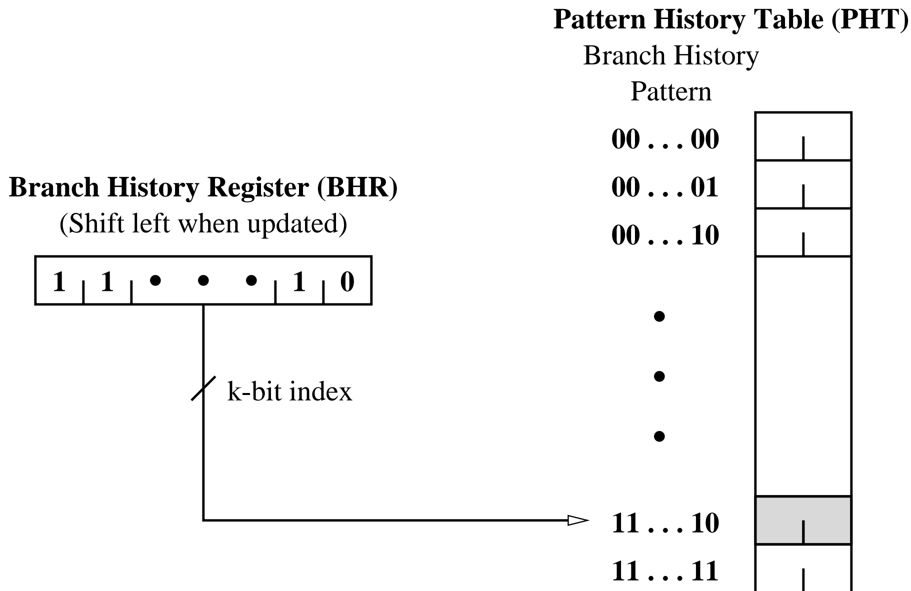


Figure 2.1: Diagram of a GAs two-level predictor

Pattern History Tables (PHTs). The branch history register is used to index into the PHT to select which 2-bit counter to use. If the configuration uses more than one PHT, one of the PHTs is chosen based on the least significant bits of the fetch address. Once the 2-bit counter is selected, the prediction is made using the same method as in the Two-Bit Counter scheme.

A global two-level predictor (GAs) uses one branch history register to collect the outcomes of all branches. Figure 2.1 shows how a GAs predictor with one PHT works. The contents of the branch history register are used to index into the PHT. This selects a 2-bit counter, shaded in the figure. The prediction for the current branch is made based on the value of this counter as explained earlier. The branch history register is updated by shifting left by one, with the prediction that was just made entered as the least significant bit. The 2-bit counter that was used to make the prediction is updated when the branch is retired using the same rules as for the Two-Bit Counter scheme. Since the outcomes in the branch history register represent the branches preceding the current branch, the global two-level predictor can use the correlation between the current branch and the other branches in the history to make its prediction. A global two-level predictor with only one PHT is also called a GAg predictor. A per-address two-level predictor (PAs) uses one branch history register per branch, stored in a Branch History Table (BHT), where each of the history registers collects the outcomes of one branch. The BHT is a tagged structure similar to the BTB,

and the histories can be added to the BTB instead of having a separate BHT if desired. It is common to include the cost of the histories, but not the cost of the BHT or BTB tags when calculating the implementation cost of a PAs predictor. To make a prediction, the branch history register corresponding to the current branch is selected and this history is used to make a prediction in the same way as in the GAs predictor. A PAs predictor with only one PHT is also referred to as a PAg predictor. Since the PAs predictor uses the previous execution pattern of the branch to make a prediction, it is able to correctly predict branches with complex, but recurring execution patterns. The two-level predictors represent a great improvement over previous schemes. Hower, they do suffer from interference between the branches that use the predictor, and they take longer time to train than simpler predictors.

Several variations of the two-level branch predictors have been proposed. The Gshare [20] predictor is identical to the GAs predictor except for the generation of the index into the PHT. In a Gshare predictor with a k-bit history, the index into the PHT is the history XORed with the least significant k bits of the fetch address that are not used to select which PHT to use. The XOR hashing function creates a more random usage pattern in the PHT. Different branches are less likely to conflict for the use of a particular entry in the PHT, with the effect that Gshare has a slightly higher prediction accuracy than GAs. Another global two-level predictor is the path-based history predictor [22]. The path-based predictor stores several bits, usually 2, from each branch target in the history register instead of using one bit for the direction of each branch. The path-based scheme more explicitly represents the path taken to get to a branch, but at the cost of allowing information from fewer past branches in the history register. In general, the path-based history predictor does not work as well as pattern history based two-level predictors.

Due to the many different ways a branch can behave, a single predictor will not be the best for all branches. This is the motivation for building hybrid branch predictors [20]. A hybrid branch predictor consists of two or more component predictors and a selection mechanism to choose which of the predictors to use for each branch. Figure 2.2 illustrates how a hybrid predictor generates a prediction. The selection mechanism [20] on the left uses a table of 2-bit saturating counters to keep track of which predictor is currently more accurate for each branch. Each branch is mapped to a counter using the least significant bits in its fetch address. If the counter value is larger than or equal to 2, the first predictor is used. Otherwise, the second predictor is used. The counter is updated when the branch

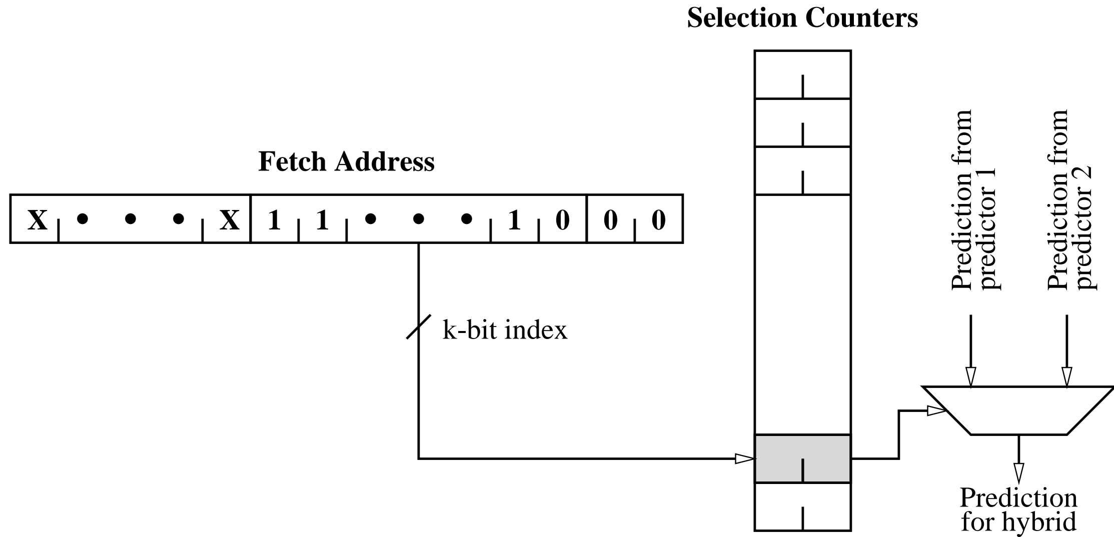


Figure 2.2: Diagram of a hybrid branch predictor

is retired. If only the first predictor was correct, the counter is incremented. If only the second predictor was correct, the counter is decremented. If both predictors made the same prediction, the counter value is not changed. In a way analogous to the way a two-level predictor works, the selection mechanism can also be organized using a history and pattern history tables for even better accuracy [4]. The component predictors used in the hybrid can be any predictors, but it is generally best to have the components complement each other. The PAs and Gshare predictors perform particularly well together in a hybrid branch predictor.

Several mechanisms have been proposed to improve global variations of the two-level predictors by reducing the amount of interference in the PHTs. Interference happens when two or more branches compete for the same entry in the PHT, and is destructive for prediction if the competing branches are likely to have opposite outcomes. Branch Filtering [3] uses a mechanism to dynamically identify strongly biased branches and predict these using a simple Last-Time predictor. The two-level predictor is only used for the less biased branches, reducing the interference between branches in the PHTs. The Agree predictor [27] makes a preliminary prediction for each branch the first time the branch is seen. Subsequently the two-level predictor is used to predict whether to agree or disagree with that prediction. Since the original prediction will be correct the majority of the time, the agree outcome will be dominant, thus creating less destructive interference. The Skewed [21] branch predictor uses three predictor banks, or PHTs, each indexed with slightly different hash functions.

The prediction of the Skewed branch predictor is the prediction made by two or more of the banks. The Bi-Mode [18] predictor uses a table of two-bit counters indexed by the branch address to make an initial prediction. This initial prediction is used to select which of two Gshare predictors, which were accessed in parallel with the table, to use for the prediction. Only the state of the selected Gshare predictor is updated when the branch resolves. These four predictors were compared in [17], and it was found that Branch Filtering outperforms the other mechanisms assuming that the fetch unit has a Branch Target Buffer (BTB) or other tagged structure in which the filter counters can be kept. If no tagged structure is available, the Bi-Mode predictor outperformed the other mechanisms including filtering using a direct mapped table of untagged counters. Common for all of these methods is that they substantially improve the accuracy of two-level predictors at the cost of a small increase in design complexity.

The correlation between branches that are separated by a subroutine call is often not captured by global two-level predictors. If the subroutine executes a large number of branches, the branch history from prior to the subroutine is lost. The Return History Stack [14] partially solves this problem by providing history information from before subroutine calls. When a call instruction is encountered, the global history is pushed on a stack without changing the current value of the history register. When a return instruction is encountered, the history from before the function call is popped off the stack and concatenated with the one or two most recent outcomes in the current history. However, this mechanism does not substantially improve prediction accuracy unless the two-level predictor is also needed for predicting return addresses.

When accessing a global two-level predictor, it is generally not known whether the PHT entry that is accessed was last written by the branch that is being predicted. This is an especially important problem for next trace or multiple branch prediction, as an entry belonging to a different branch is very unlikely to be correct. The Path-Based Next Trace predictor [14] combines a global two-level trace predictor with a trace predictor similar to a Last-Time predictor. Each entry in the PHT of the global two-level predictor has a tag to show which trace ID (which has a similar function as the fetch address) it belongs to. When a prediction is made, the current trace ID is compared to the tag. If the tag matches, the two-level predictor is used. Otherwise, the untagged Last-Time predictor, which is less likely to be subject to interference, is used. The YAGS [8] predictor uses a similar scheme

for predicting a single branch per cycle, but with update rules that inhibit the update to the two-level predictor if the simple predictor is correct. Both of these predictors have a table lookup, a tag match and one or two MUXes on the critical path of the prediction.

The optimal history length for a global two-level predictor of a given size is generally not the same for all benchmarks. The Dynamic History Length Fitting [15] predictor attempts to dynamically identify the best history length while the program is running. The program is divided into intervals of 16,000 or more branches. For each interval, one history length is used and the number of mispredictions is counted. At the end of the interval, the number of mispredictions is stored in a table and compared with previous misprediction counts for other history lengths. A new history length is then chosen based on which history length has the lowest misprediction count registered in the table, and the process is repeated. This idea is appealing, but adds substantial complexity to the predictor.

2.2 Assisting Branch Prediction Using Profiling

Several mechanisms have been proposed which use profiling to either do fully static branch prediction, or to improve dynamic branch predictors. In the simplest form of profiling for branch prediction, often referred to as Simple Profiling, the most frequent direction of each branch is determined during the profile run [12]. The most frequent direction is communicated to the branch prediction hardware using a bit in the branch opcode. The prediction made for the branch at run-time is the direction given by the prediction bit.

A more complex profile can be used to improve the accuracy of static branch prediction by code duplication [35]. Taken and not-taken counts are captured for each possible path leading up to a branch. If a branch has different behavior for different incoming paths, the compiler can duplicate the basic block containing the branch, and possibly some of the blocks leading up to it, such that one copy of the branch will be reached if the block is entered through a path that makes the branch likely taken and another copy is reached if the path makes the branch likely not-taken. This makes the branch behavior more biased, increasing the accuracy of static branch predictors and simple dynamic predictors such as the Two-Bit Counter scheme. An important benefit of this scheme is that the improved static branch prediction accuracy can be used to enhance superblock scheduling.

As mentioned earlier, branches can behave in many different ways. Branch Classifica-

tion [5] takes advantage of this by separating branches into different classes based on their behavior during a profile run. The class of a branch is passed to the dynamic predictor hardware through the branch opcode. During the run of a program, different predictors are used for branches belonging to different classes. One such proposed scheme is to divide branches into three classes: mostly taken branches, mostly not taken branches, and all other branches. A Simple Profiling predictor is used for the mostly taken and mostly not taken branches, while a dynamic hybrid predictor is used for the other branches. The Profile-Guided Multi-Heuristic branch predictor [6] uses a similar concept. The compiler identifies loop branches by code analysis or profiling. Each branch has one bit encoded in the opcode to show whether it is a loop branch or not. During the run of the program, loop branches are predicted using a complex loop predictor and other branches are predicted using a Gshare or similar predictor. The Statically Selected Hybrid branch predictor [13] also partitions branches based on which predictor is likely to be best for it. During the profile run, the best of the two component predictors for each branch is identified. This is done by simulating the component branch predictors during the profile run, but ways of alleviating the complexity of this are suggested. Each branch has a bit encoded in the opcode to show which predictor is best for it. During the run of the program, this bit determines which predictor to use for the branch.

The Variable Length Path branch predictor [29] takes profiling one step further. As was mentioned earlier, the optimal history length for a global two-level predictor is not the same for all branches. The Variable Length Path predictor determines a nearly optimal history length for each of the branches during the profile run, and communicates this to the prediction hardware using a few bits in the branch opcode. The prediction hardware predicts the branch using the suggested history length. This scheme also involves a novel method for generating path histories by XORing target addresses together. This predictor is much more accurate than other comparable predictors, but the implementation complexity is substantial.

2.3 Branch Behavior and Effects Seen in Branch Predictors

There have also been studies on the behavior of branches and branch predictors. Pan et al. [23] identified several cases of branches being correlated in the source code of the

SPECint89 benchmarks and used this as a motivation for why global two-level predictors work.

Talcott et al. [30] and Young et al. [34] studied and classified the effects of pattern history table interference, and showed that it negatively affects the performance of two-level branch predictors. These two papers used interference-free predictors to aid in the understanding of the potential of two-level predictors. An interference-free predictor has one PHT for each branch and is therefore prohibitively large, but does not suffer from the negative effects of PHT interference.

Young et al. [34] also showed the slight advantage of path histories over pattern histories for static branch prediction. They further showed that the history information from before a call is more useful for the prediction of a branch than the history information from within a called subroutine. Furthermore, they investigated the importance of adaptivity in the PHTs of global two-level predictors. They found that a statically determined PHT, when using the same profiling and testing set and having a separate PHT for each branch, sometimes outperforms a PHT using 2-bit counters. This was shown to indicate the potential benefits of statically exploiting correlation using their code restructuring scheme.

Sechrest et al. [25] studied the role of adaptivity in two-level branch predictors and determined that, for per-address predictors with short histories, having statically determined values in the PHT performed on par with the adaptive scheme using 2-bit counters. The PSg(algo) per-address two-level predictor, was introduced to show the potential of a static PHT. In this predictor, the prediction for each history pattern was statically defined based on an algorithm that detects repeating outcomes in the history pattern. However, this study compared only with the PAg predictor, and not with the more accurate PAs predictor.

Chen et al. [7] analyzed the performance of the PAg two-level predictor by comparing it to an optimal predictor in data compression, Prediction by Partial Matching (PPM). They asserted that PPM would also be optimal for branch prediction, and showed that a modified version of PPM, reduced to be more cost effective, outperformed PAg slightly on both the SPECint95 and the IBS benchmarks. Federovsky et al. [11] extended this by also examining a Context Tree Weighting algorithm. They indicated that PAg can be improved upon, but did not present an implementation. Neither of the studies compared their results with the slightly better PAs predictor.

Farcy et al. [10] identified and investigated 53 branches in the SPECint95 benchmarks

that were frequently mispredicted. Of these, the 60% that were inside loops were investigated and classified based on the way the branch condition was generated. 20% of the 53 branches were shown to be based on predictable functions, either predictable arithmetic functions or table traversals, that can be speculatively computed to generate the condition early. Another 36% of the 53 branches were based on list or tree traversals, for which the mechanism of speculatively generating the condition can not be used. The remaining 4% of the branches that were examined were based on non-linear arithmetic functions.

CHAPTER 3

Simulation Methodology

3.1 Simulation Environment

The experiments in this dissertation are conducted using a trace driven simulator for a load/store architecture using the instruction set of the Motorola MC 88000. There are two parts to the simulator: the trace generator, and the microarchitectural simulator. The trace generator generates a trace of all user mode instructions corresponding to the correct execution path. All system calls are emulated using the host machine. The microarchitectural simulator uses the trace to simulate a microprocessor using the HPS model of execution [24]. The microarchitectural simulator is capable of evaluating branch predictors, cache and memory organizations, and different processing cores.

The simulator can evaluate branch predictors in full pipeline or branch predictor only mode. In the full pipeline mode, a single branch predictor is connected to the execution core to provide full statistics of the program execution, such as IPC, branch resolution time, cache miss ratios etc. In branch predictor only mode, a large number of branch predictors can be simulated at the same time to evaluate their misprediction rates. The branch predictor only mode drastically reduces the required simulation time while still producing accurate results for the misprediction rates of the branch predictors.

The metric that is used for comparing predictors in this dissertation is branch misprediction rates. Although it is desirable to rate a predictor by the impact it has on the performance (such as IPC) of a processor, this can not be done without selecting a particular processor implementation. Misprediction rate was chosen as the metric as it is most commonly used in the research literature, and because using misprediction rate as the

Benchmark	Abbr.	Description	Training Set	Test Set
compress	com	Data compression program	prof.in*	test.in*
gcc	gcc	GNU C compiler version 2.5.3	stmt.i	jump.i
go	go	Computer program playing go	short.in*	2stone9.in*
ijpeg	ijp	Image compression program	vigo.ppm*	specmun.ppm*
m88ksim	m88k	Motorola 88100 simulator	dhry.test.big	dcrand.train.big
perl	perl	Perl interpreter	primes.pl*	scrabbl.pl*
vortex	vor	Object-oriented database	vortex.35M*	vortex.in*
xlisp	xli	XLISP interpreter	7queens.lsp*	train.lsp

*The input set is a modified version of one of the SPECint95 data sets.

Table 3.1: Description of benchmarks and their input sets

metric reduces the simulation requirements. The prediction accuracy has previously been shown to be strongly correlated to the performance of a processor [1]. As the misprediction rate is equal to $(1 - Prediction\ Accuracy)$, the misprediction rate is similarly correlated to processor performance.

3.2 The SPECint95 Benchmarks

The results presented in this dissertation are for the eight integer programs from the SPECint95 [28] suite. The SPECint95 benchmarks were chosen to make comparisons with other studies possible. These benchmarks cover a diverse set of general purpose applications with varying characteristics. Some, like gcc, vortex, and go, have large footprints and many static conditional branches. Others, like compress and xlisp, have smaller footprints and much fewer static conditional branches.

Table 3.1 gives a short description of each of the benchmarks and lists the training and test data sets used. The test data sets were used to generate all performance results reported in this thesis. The training data sets were used to generate the benchmark profiles for those experiments that required profiling. All the data sets are either the test, training, or reference data sets provided with the SPECint95 benchmarks or abbreviated versions of these. In most cases, abbreviated versions of the SPECint95 input sets had to be used as the SPECint95 input sets were constructed to finish in a reasonable time on existing hardware rather than simulators and therefore run for a long time. The benchmarks were always simulated until completion. Table 3.1 also gives abbreviated names for all of the

Benchmark	#Dynamic Instructions	#Dynamic Branches	#Dynamic Cond BR	#Static Cond BR
compress	103,015,025	17,069,558	10,661,855	310
gcc	154,450,036	34,060,956	25,903,086	14,755
go	125,636,236	22,047,565	17,924,928	4,891
ijpeg	206,802,135	24,147,330	20,441,307	1,179
m88ksim	120,720,559	23,212,206	16,719,523	991
perl	78,148,849	16,030,598	10,570,887	1,670
vortex	231,997,610	43,171,027	33,853,896	6,385
xlisp	187,724,756	44,758,353	26,422,064	512

Table 3.2: Execution statistics for test data sets

benchmarks for use in the figures in the rest of this dissertation.

Table 3.2 lists more detailed information about the execution of the test input set for each of the benchmarks. The total number of instructions executed during the program run is listed in the second column. The number of branches executed is listed in the third column. The number of conditional branches is listed in the fourth column, and the number of static conditional branches that were executed at least once during the run of the program is listed in the final column.

CHAPTER 4

Evaluation of Basic Branch Properties and Predictors

In this chapter, three basic branch properties are investigated to provide some fundamental knowledge about the behavior of branches. This will be the foundation for the more detailed investigation in the later chapters. This chapter also contains an evaluation of several of the branch prediction algorithms described in Chapter 2. The prediction algorithms that are evaluated here were chosen because they are well-known, illustrate some property well, or are particularly likely to be useful in high performance microprocessors. The experiments that are presented in this chapter mostly reproduce work done by prior researchers. This work is reproduced here to present the prior work in a single place and using the same benchmarks and input sets. Furthermore, the results given in this chapter are used as a baseline for comparisons with new mechanisms introduced in the later chapters of this dissertation.

4.1 Branch Properties

Three basic branch properties are investigated in this section: type, frequency and taken rates. The experiments are similar to those done in [2, 31], but are reproduced here as the benchmarks and input data sets differ slightly. These properties are easily measured and are useful for understanding branch behavior.

Branches can be divided into conditional and unconditional branches. Based on the source of the target address, unconditional branches can be further divided into immediate branches, indirect branches and returns. Immediate branches have the target address encoded in the branch instruction, indirect branches get their target address from a register,

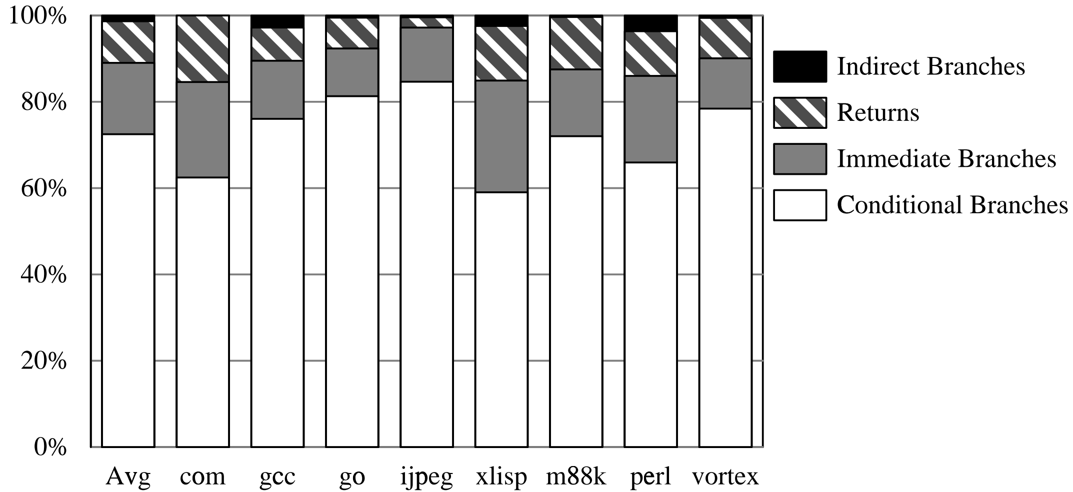


Figure 4.1: Weighted distribution of branches by type

and returns get their target address from a link register or a stack. Technically, indirect branches and returns could also be conditional. However, most ISAs do not allow this. Figure 4.1 shows the distribution of the branch types encountered when executing each of the SPECint95 benchmarks. The majority of the branches, 72% on average, are conditional. 17% are unconditional immediate, 10% are returns, and 1% are indirect. This highlights the importance of conditional branch prediction. Of the unconditional branches, immediate branches can be predicted accurately using a BTB, and returns can be predicted accurately using a Return Address Stack [16]. Target prediction for indirect branches is an important research topic for the performance of object-oriented programs, but is less important for the SPECint95 benchmarks.

Figure 4.2 shows the distribution of the execution frequencies of static conditional branches. Most static branches are only executed a few times during the run of a program. On average, 53% of all branches were executed 99 times or fewer. Only 11% of all branches were executed 10,000 times or more. Figure 4.3 shows a similar distribution, but this time with each branch weighted by its execution frequency. This shows the representation of each of the categories in the dynamic instruction stream. The 53% of the branches that were executed 99 times of fewer make up only 0.2% of the branches in the dynamic instruction stream. The 11% of the branches that were executed 10,000 times or more make up 87% of the branches in the dynamic instruction stream. This confirms the rule of thumb that 10% of the code is responsible for 90% of the execution of a program. This also

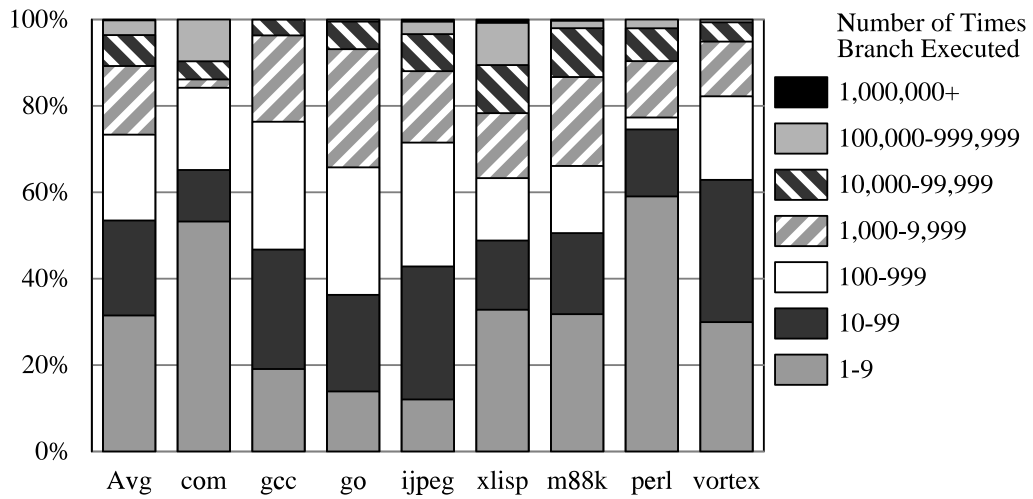


Figure 4.2: Distribution of static branches by execution frequency

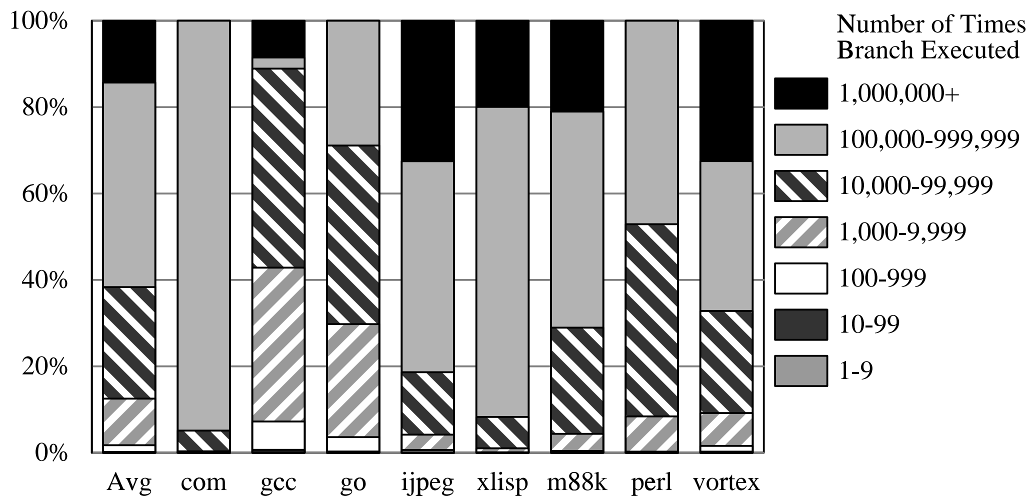


Figure 4.3: Weighted distribution of static branches by execution frequency

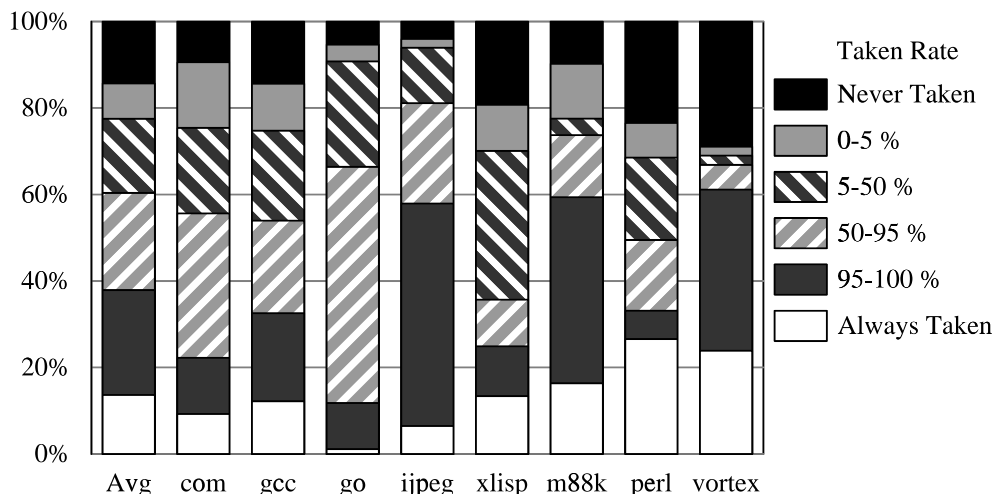


Figure 4.4: Weighted distribution of static branches by taken rate

emphasizes that frequently executed branches account for the vast majority of all branches that are executed. Even in the go and gcc benchmarks, which have a larger number of static branches, branches executed 99 times or fewer account for less than one percent of the branches in the dynamic instruction stream. On the other side of the spectrum, the vortex benchmark has one single branch which accounts for 32% of all branches executed. Due to the predominance of frequently executed branches in the instruction stream, it is very important to predict the frequent branches well.

Figure 4.4 shows the distribution of the taken rates of static branches with each branch weighted by its execution frequency. The taken rate of a branch is the fraction of the time that branch was taken during the complete run of the program. 28% of all branches in the dynamic instruction stream were instances of static branches that were either always taken or never taken. These should be easy to predict. Another 32% of the branches were less than 5% taken or more than 95% taken. Being able to predict the dominant direction will achieve at least 95% accuracy for these branches, so the Two-Bit Counter scheme does reasonably well on these. 40% of the branches have taken rates between 5% and 95%. It is for these branches that prediction is most challenging.

Knowing the predominant direction and taken rate of a branch at compile time can be useful for prediction. Three predictors in particular can take advantage of this information: a general profile based predictor [12], an agree predictor [27] using static agree bits, and a classifying branch predictor [5]. To determine the stability of the taken rates when the

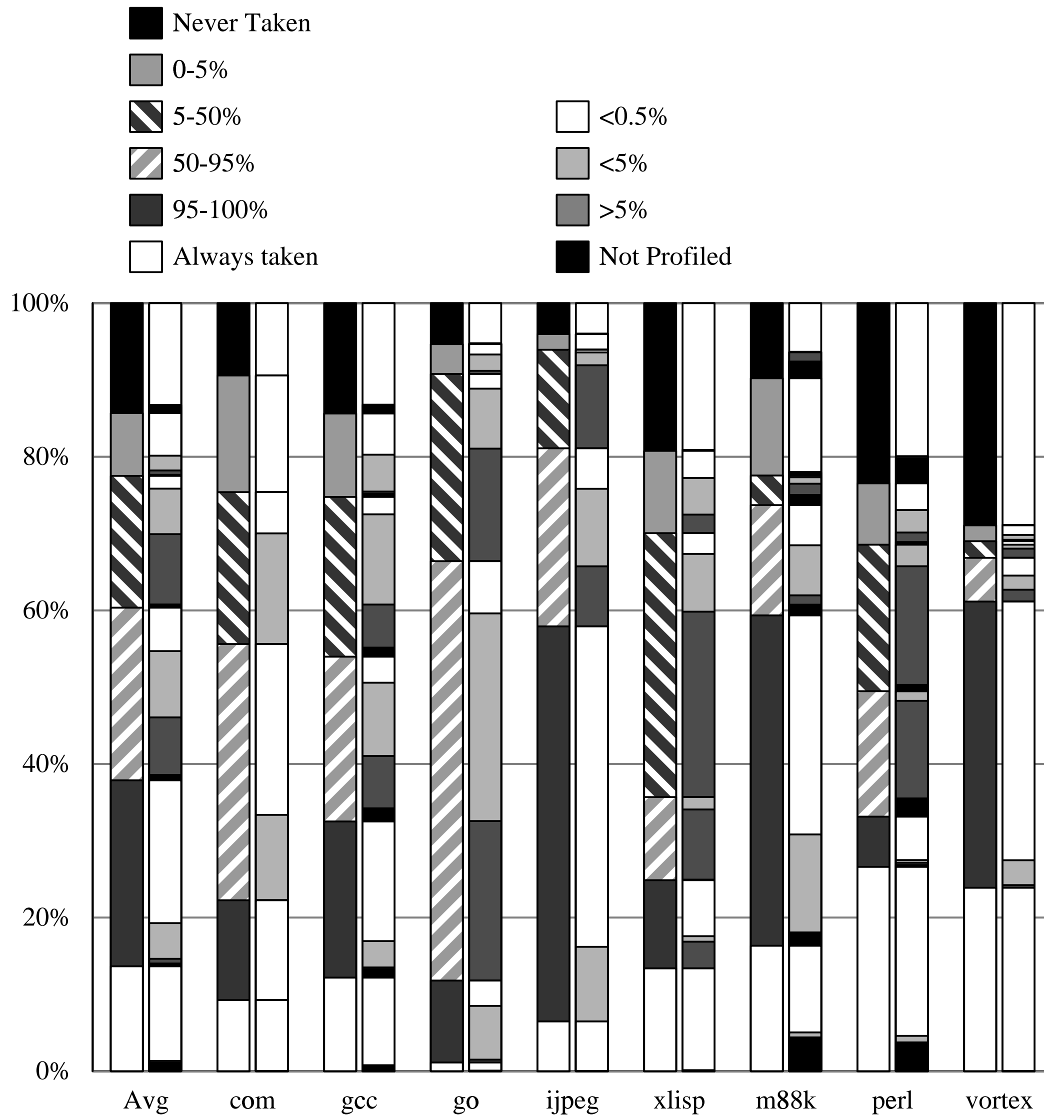


Figure 4.5: Change in taken rate between profiling and test runs

input set changes, we compared the taken rate of each branch using a profiling set to the taken rates found with the testing set.

Figure 4.5 shows the change in taken rates between input sets. Each benchmark is represented by a pair of bars. There are two legends on the top of the graph. The leftmost legend refers to the leftmost bar in each pair. The rightmost legend refers to the rightmost bar in each pair. The leftmost bars correspond to the taken rates of branches using the testing set. These bars are identical to those in Figure 4.4. The rightmost bar breaks down each of the categories from the leftmost bar showing whether the branches in that category were also represented in the profiling set, and if they were, how much the taken

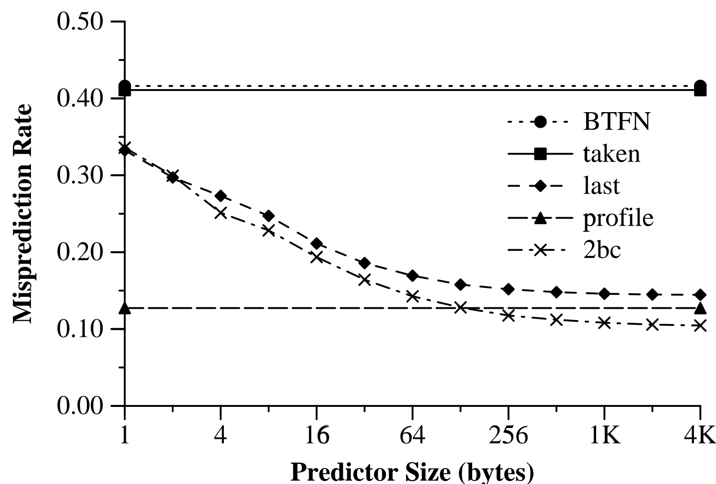


Figure 4.6: Accuracy of basic branch predictors

rate changed. The darker the color of a section of the rightmost bar, the more the taken rate of the branches represented changed. The black regions represent branches that were not included in the profile run. The conclusion from this experiment is that branches that were always or never taken in one input set are likely to remain that way when the input set changes. The less biased a branch is, the more likely its taken rate is to change.

4.2 Basic Branch Predictors

The branch prediction algorithms that are easiest to implement are those requiring no history, or only a single level of history. The algorithms evaluated here are (with the labels used on the graph in parenthesis): Always Taken (taken), Backward Taken and Forward Not-Taken (BTFN), Simple Profiling (profile), Last-Time (last), and Two-Bit Counter (2bc). The Last-Time and Two-Bit Counter predictors were both constructed using an untagged table.

Figure 4.6 shows the average misprediction rate for the SPECint95 benchmarks for each of these predictors. The vertical axis shows the misprediction rate, and the horizontal axis shows the size of the predictor. The size is calculated as the storage space used for dynamic history information, such as counters.

As the static predictors do not need storage for dynamic history, their performance is represented by horizontal lines. The Always Taken and Backward Taken, Forward Not-Taken predictors have high misprediction rates, 41.1% and 41.6% respectively.[1] These

[1]The accuracy of the BTFN predictor can be improved using compiler support to make not-taken the

predictors are cheap, require little hardware support, but still represent an improvement over using no prediction at all. However, they are not appropriate for modern microprocessors. The misprediction rate of the other static predictor, Simple Profiling, is 12.7%. While this is much better than the other static predictors, one out of every eight branches is still mispredicted, and the program must be profiled when it is compiled.

The dynamic predictors learn the behavior of the branches during the run of the program, and can therefore achieve lower misprediction rates without profiling. As the prediction tables of the dynamic predictors increase in size, there is less conflict between different branches using the table. This leads to lower misprediction rates as the size increases. Most of the performance can be achieved with a table of approximately 256 B, and there are only very marginal gains from increasing the size beyond 2 KB. At 2 KB, the misprediction rates of the Last-Time and Two-Bit Counter predictors are 14.5% and 10.5% respectively. As explained in Section 2.1 the added history of the Two-Bit Counter enables it to predict loops and other branches with occasional glitches in the execution pattern better than the Last-Time predictor.

4.3 Two-Level Adaptive Branch Predictors

Even the best of the simple prediction schemes, a table of two-bit saturating counters, mispredicted 10.5% of all branches. To achieve lower misprediction rates, two-level branch predictors can be used. Several variations of these are evaluated in this section. Several key design decisions relating to which configurations of the two-level predictors perform best are also evaluated.

4.3.1 Branch History Register Length and Number of Pattern History Tables

As explained in Section 2.1, two-level predictors use a history to index into one or more Pattern History Tables (PHTs). For each bit that is added to the history, the number of entries in each of the PHTs doubles. Increasing the length of the history usually improves the predictor's ability to capture correlation, and increasing the number of the PHTs reduces the contention between branches using the same entries in a PHT. Both lengthening the

most likely outcome of forward branches.

Size	Predictor			
	GAs	Gshare	Path	PAs
	(HL,PHTs)	(HL,PHTs)	(HL,PHTs)	(HL,PHTs)
1 KB	(6,64)	(6,64)	(7,32)	(3,128)
4 KB	(9,32)	(8,64)	(9,32)	(8,32)
16 KB	(12,16)	(12,16)	(11,32)	(16,1)
64 KB	(14,16)	(14,16)	(13,32)	(18,1)
256 KB	(16,16)	(20,1)	(15,32)	(20,1)

Table 4.1: Configurations used for two-level adaptive branch predictors. The configurations are represented as (HL,PHTs) where HL is the history length in bits and PHTs is the number of pattern history tables.

history and adding PHTs result in improved prediction accuracy, so when considering a fixed hardware budget, there is a tradeoff between having fewer larger PHTs or having more smaller PHTs. To determine which configurations work best, all possible configurations for each predictor were simulated. The configurations which yielded the lowest misprediction rate for each of the predictors are listed in Table 4.1. “Path” refers to the Global Path-Based History scheme where 2 bits from every branch target were shifted into the branch history register. For the PAs predictor, the cost of the branch histories in the Branch History Table (BHT) was included, but the tags were assumed to be shared with a different structure. If no configuration was exactly the desired size, the closest size was used. A 2 K entry 4-way set-associative BHT was used for PAs.

The GAs and Gshare global predictors have similar optimal configurations. As the predictors grow larger, there is less contention in the PHTs, so fewer PHTs are needed. The path-based predictor worked best with 32 PHTs for all sizes. The PAs predictor suffers less from sharing the PHTs, so moderately large predictors needed only few PHTs. The configurations given in the table perform best on average for the benchmarks, but there is some variation between the individual benchmarks. As a general rule, the optimal configuration for larger benchmarks has more PHTs whereas for smaller benchmarks a longer history is better.

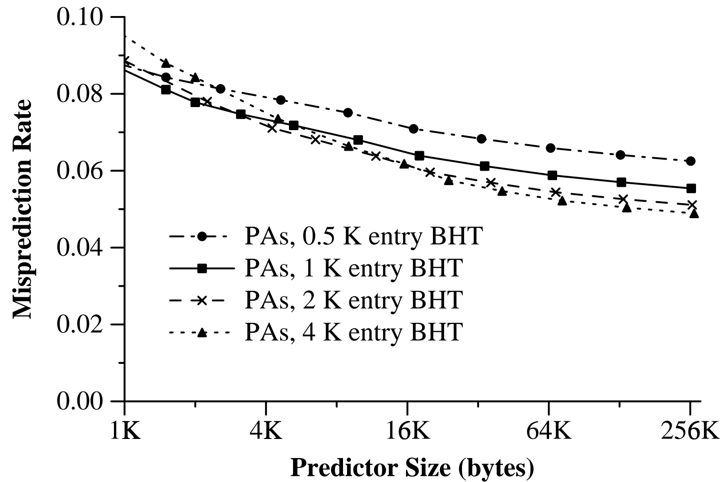


Figure 4.7: Effect of BHT size on misprediction rate of PAs

4.3.2 Effect of Branch History Table Size

The Branch History Table size has a strong effect on the performance of predictors using per-address history. If the branch history table is too small, branch histories will frequently be lost due to contention between different branches. If no branch history is found in the BHT, the PAs predictor cannot be used to predict that branch and a less accurate predictor, BTFN for the experiments in this dissertation, must be used. On the other hand, a large BHT uses storage that could otherwise be used for longer histories or more PHTs.

In Figure 4.7, the performance of PAs is evaluated for BHT sizes ranging from 0.5 K entries to 4 K entries, all four-way set associative. As is the case with caches, reducing the associativity has an effect similar to reducing the size. For sizes between 1 K and 4 KB, the PAs predictor using the 1 K entry BHT achieved the lowest misprediction rate. For 4-16 KB predictors, the 2 K entry BHT worked best, and for even larger predictors the 4 K entry BHT worked best. The 2 K entry BHT was close to being best for all sizes. We therefore use a 2 K entry BHT for further experiments in this chapter. If a large BHT can not be accessed within the cycle time, two-level structures similar to two-level caches may be considered.

4.3.3 Comparison of Two-Level Branch Prediction Schemes

Four two-level predictors are compared for sizes ranging from 1 KB to 256 KB in Figure 4.8. The Global Path-Based History scheme always works poorly compared to the other predictors. This is because several bits are devoted to each branch outcome, so information

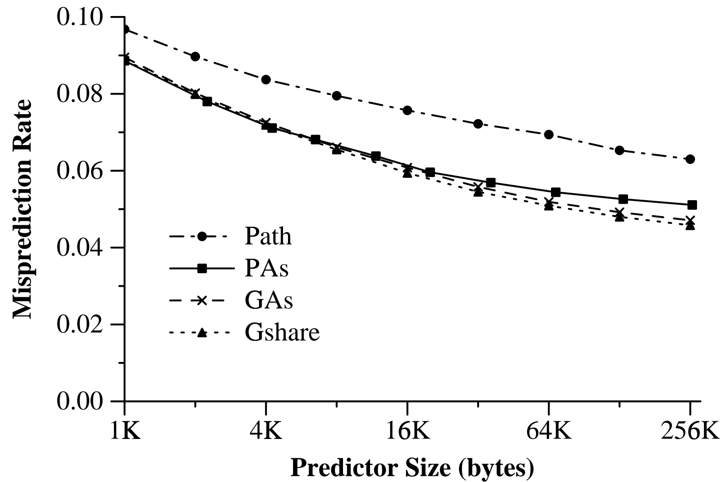


Figure 4.8: Comparison of two-level predictors

about fewer branches fits in the history register. The remaining three predictors, Gshare, GAs, and PAs, achieve nearly identical misprediction rates for predictors smaller than 8 KB. PAs uses a different type of information to make predictions than GAs and Gshare, but each type of information is equally useful on average. For larger predictors, PAs performs worse than Gshare and GAs. Due to the interference in the GAs and Gshare predictors, increasing the size has a more significant effect for these two. Gshare is marginally better than GAs at all sizes. GAs and Gshare exploit the same type of correlation for prediction, but Gshare works a little better due to its improved hashing function. For comparison, if the Two-Bit Counter predictor was included in the figure, its misprediction rate of 10.5% would be off the scale.

Even for 64 KB predictors, the accuracy of Gshare, GAs, and PAs is close. At this size, Gshare has a misprediction rate of 5.09%, GAs 5.19%, and PAs 5.44%. This indicates that Gshare is likely to be the best of these predictors at this size. However, even though Gshare is on average better than PAs, there is large variation between the benchmarks. Table 4.2 shows the misprediction rates of the Gshare and PAs predictors for the individual benchmarks. Gshare is consistently better than GAs and Path for all benchmarks, so these predictors are not listed in the table.

Benchmark	Gshare	PAs
compress	8.23%	5.38%
gcc	5.09%	7.76%
go	10.97%	17.43%
ijpeg	6.99%	4.23%
m88ksim	1.59%	1.31%
perl	2.18%	2.84%
vortex	0.86%	1.10%
xlisp	4.81%	3.50%

Table 4.2: Misprediction rates of 64 KB predictors

For the 64 KB size, Gshare is substantially better than PAs for go, gcc, and perl. PAs is substantially better than Gshare for compress, ijpeg, and xlisp. The difference is only small for two benchmarks, m88ksim and vortex. This is one reason why hybrid branch predictors, which use both a Gshare and a PAs predictor, do so well.

4.4 PHT Interference and Interference Reduction Schemes

PHT interference severely limits the prediction accuracy of two-level predictors, most notably the global history schemes. There is only a limited number of PHTs, so branches that have conflicting behavior often end up sharing the same PHT. This limits the performance of two-level predictors so much that it is important to be aware of this effect. The amount of PHT interference in the Gshare and PAs predictors is quantified in this section.

The definitions of interference used here are those given by Young et al. [34]. PHT interference occurs when a conditional branch references a PHT entry that was last referenced by another conditional branch. This interference is classified as constructive if the counter correctly predicts the branch outcome and a predictor with an infinite number of PHTs, thus having no PHT interference, mispredicts the outcome. The interference is destructive if the counter mispredicts while the predictor with an infinite number of PHTs predicts correctly. Otherwise, the interference is neutral.

	~4 KB			~16 KB			~64 KB		
benchm.	const	neutral	destr	const	neutral	destr	const	neutral	destr
com	19	686	252	14	513	81	12	433	57
gcc	235	7,498	4,637	143	4,525	2,730	70	2,287	1,276
go	1,270	23,521	12,402	718	14,152	6,543	337	6,940	2,845
ijp	139	2,749	1,038	79	1,785	569	51	1,242	358
m88k	10	967	213	2	517	62	0	359	25
perl	17	3,107	1,427	2	617	292	1	216	101
vor	15	1,986	801	3	757	231	0	317	78
xli	27	1,179	179	22	802	112	16	541	37

Table 4.3: PHT interference per 100,000 accesses for Gshare

Table 4.3 shows the amount of interference per 100,000 accesses for three different size Gshare predictors: 4, 16 and 64 KB. As we are studying interference, and interference is the effect that forces us to use multiple PHTs, configurations with only one PHT were used here. The table shows that the amount of interference decreases with the size of the predictor. The larger the PHT, the less of a chance for accidental interference between branches. Benchmarks, such as gcc and go, that frequently execute a large number of static branches suffer the most from interference. Depending on the predictor size, 12, 7 or 3 % of all branches in go are mispredicted because of destructive interference. In contrast, constructive interference is rare. For go, only 1% or less of the branches are correctly predicted because of constructive interference. Neutral interference is more frequent, but has no systematic effect on prediction accuracy. Interference is a very serious problem for two benchmarks, gcc and go, and is a fairly serious problem for two more. The remaining four benchmarks are less affected by interference.

If all interference (constructive, destructive, and neutral) were removed, the average misprediction rate of a 4 KB Gshare predictor would be 4.61% instead of 7.19%. For a 16 KB Gshare the misprediction rate of the interference free predictor is 4.42% instead of 5.94%, and for the 64 KB predictor the misprediction rate is 4.30% instead of 5.09%. In addition to demonstrating the impact of interference, this indicates, as also stated in [27], that the main reason for the improvement when increasing the size of global two-level predictors is reduction in the amount of destructive interference.

benchm.	~5 KB const	~5 KB neutral	~5 KB destr	~20 KB const	~20 KB neutral	~20 KB destr	~68 KB const	~68 KB neutral	~68 KB destr
com	490	48,063	2,080	300	45,882	1,210	230	45,678	960
gcc	1,273	66,062	2,471	1,064	64,036	2,023	881	62,142	1,659
go	3,079	60,701	7,555	2,449	56,554	5,934	1,922	52,473	4,719
ijp	854	50,565	2,609	437	48,872	1,395	299	47,704	922
m88k	68	62,024	570	66	60,268	596	47	59,024	529
perl	347	79,502	2,352	244	76,459	1,216	149	74,927	988
vor	115	63,044	367	85	62,453	261	69	62,133	219
xli	549	62,117	2,256	258	58,661	1,356	171	56,606	873

Table 4.4: PHT interference per 100,000 accesses for PAs

Table 4.4 shows the amount of interference per 100,000 accesses for three different size PAs predictors. As in the previous table, all configurations have a single PHT. The 5 KB predictor uses 13 history bits, the 20 KB predictor uses 16 history bits, and the 68 KB predictor uses 18 history bits. As with Gshare, the larger the predictor, the less constructive and destructive interference. However, the neutral interference in a PAs predictor is mostly due to branches that are always taken, always not-taken, or have short periodic patterns. These patterns are generally predicted the same way for all branches, and the interference they cause is therefore neutral. These patterns continue as the history length is increased, so the amount of neutral interference only drops slightly as the size increases. For a per-address two-level predictor, interference is a very serious problem only for go, but a fairly serious problem for about five more of the benchmarks.

If all interference (constructive, destructive, and neutral) were removed, the average misprediction rate of a 5 KB PAs predictor would be 5.48% instead of 7.11%. For a 20 KB PAs the misprediction rate of the interference free predictor is 5.10% instead of 5.96%, and for the 68 KB predictor the misprediction rate is 4.95% instead of 5.44%. As is the case with Gshare, the negative effects of interference diminish for larger predictor sizes. For PAs, it is also the case that the main reason for improvement when increasing the size is the reduction in the amount of destructive interference.

Size	Predictor			
	Branch Classification		Branch Filtering	
	Thresh(x)	(HL,PHTs)	(n,n_{init})	(HL,PHTs)
1 KB	90	(12,1)	-	-
4 KB	100	(10,16)	(16,15)	(13,1)
16 KB	100	(14,4)	(20,19)	(16,1)
64 KB	100	(18,1)	(24,22)	(18,1)
256 KB	100	(20,1)	(24,21)	(20,1)

Table 4.5: Configurations used for Branch Classification and Branch Filtering. HL is the history length and PHTs is the number of pattern history tables used.

4.4.1 Reducing Interference Via Branch Classification or Filtering

A number of mechanisms for reducing interference in two-level predictors were described in Chapter 2. The effectiveness of two of these, Branch Classification and Branch Filtering, is examined here. The version of Branch Classification used here uses a profile run to find the branches that were more than x% taken or x% not-taken. These branches are then predicted using their profiled direction rather than using the Gshare predictor. The threshold (x) values used in this experiment are listed in Table 4.5 along with the configurations of the Gshare predictors Branch Classification was applied to. A threshold of 100% indicates that only always taken or always not-taken branches were selected for profile-based prediction. Branch Filtering uses a set of counters in the BTB to identify branches that have been taken or not-taken more than n times in a row. These branches are predicted using a Last-Time predictor instead of the main predictor. When a branch misses in the BTB, the counter is initialized to n_{init}. The threshold and initialization values used for Branch Filtering are also listed in Table 4.5 along with the configurations of the Gshare predictors Branch Filtering was applied to. The BTB size was 2 K entries. Due to the cost of the counters for the filtering mechanism, a 1 KB configuration is not given.

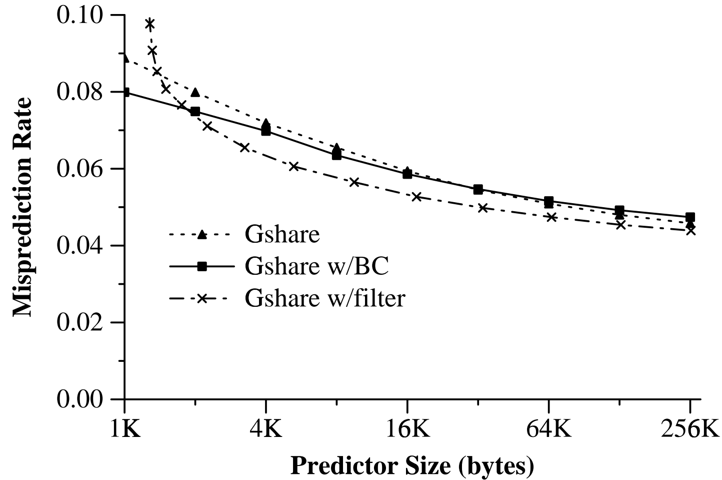


Figure 4.9: Improving Gshare using Branch Classification or Branch Filtering

In Figure 4.9, the misprediction rate of a normal Gshare predictor is compared to that of a Gshare predictor using Branch Classification (Gshare w/BC) and a Gshare predictor using Branch Filtering (Gshare w/filter). For predictors of 2 KB or less, Branch Classification performs best. For 2 KB or larger, Branch Filtering performs best. Interference reduction techniques are particularly effective for small predictors, but using Branch Filtering results in an improvement even for sizes of 64 KB or more. At 64 KB, a regular Gshare has a misprediction rate of 5.09%, whereas the misprediction rate of Gshare using Branch Filtering is only 4.74%. The improvements for Branch Classification and Branch Filtering are better for programs, such as gcc and go, with many static branches.

4.5 Hybrid Branch Predictors

The general framework for a hybrid branch predictor was explained in Section 2.1. In this section, two such hybrid branch predictors are evaluated. Both of them use Gshare as one component, with the other component being either PAs or the Two-Bit Counter predictor. These schemes are referred to as the PAs/Gshare and 2bc/Gshare predictors. The selection mechanism is a table of saturating counters, such as the one in [20]. However, the selection mechanism uses 3-bit saturating counters instead of 2-bit counters, as we found these to work better.

It is not trivial to come up with the best configurations for hybrid branch predictors. Each of the component predictors have their own parameters, and the size of each component and the selection mechanism are other parameters. For this experiment, the configurations

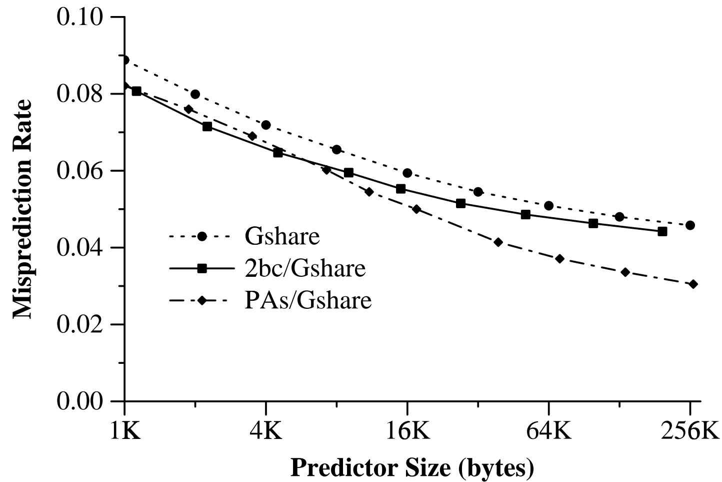


Figure 4.10: Accuracy of hybrid branch predictors

were based on the original paper [20] on hybrid predictors. For the PAs/Gshare predictor, the hardware is shared almost equally between the selection mechanism, PAs, and Gshare. For the 2bc/Gshare predictor, half of the hardware is allocated to the Gshare component, and approximately a quarter each to the Two-Bit Counter predictor and the selection mechanism. In both cases, selection mechanisms over 3 KB were not considered as this is unlikely to improve performance.[2] For each of the component predictors, the configurations with only a single PHT were used. This was used even though it may not be optimal for the smallest predictors.

Figure 4.10 shows the performance of the two hybrid predictors along with a Gshare predictor for comparison. For predictors smaller than 4 KB, the 2bc/Gshare hybrid is slightly better than the PAs/Gshare hybrid. Over 8 KB, PAs/Gshare is better, with a large performance advantage for sizes over 32 KB. At an approximate cost of 64 KB, PAs/Gshare has a misprediction rate of 3.7%, 2bc/Gshare has a misprediction rate of 4.8% and Gshare has a misprediction rate of 5.1%. All of these predictors can be improved using Branch Filtering or Branch Classification, with a larger improvement expected from Gshare. Only the selection mechanism described in the original hybrid branch prediction study was used here, although other mechanisms [4, 9] have been proposed more recently.

[2] As will be shown in Chapter 7, larger selection mechanisms only improve performance if a two-level selection mechanism is used.

4.6 Summary of Branch Predictor Performance

This chapter has provided a general evaluation of many of the important branch prediction schemes available today. This is not an exhaustive study of all available schemes, but a representative mix of the types of predictors that are available. Several of these schemes will be used in the other chapters of this dissertation either as baselines to compare against, or as components in hybrid branch predictors. A number of different strategies in branch prediction were evaluated: static predictors, simple dynamic predictors, two-level predictors, interference reduction schemes and hybrid branch predictors.

In summary, Simple Profiling can achieve a misprediction rate of 13% without dynamic branch prediction hardware. The Two-Bit Counter scheme is a simple dynamic predictor that has a misprediction rate of just over 10% at low complexity and cost. Two-level predictors require slightly more complex control logic, but provide lower misprediction rates at low cost, and approach 5% for 64 KB predictors. Interference reduction schemes are particularly good at improving global two-level schemes, with especially large improvements in the 2 to 32 KB range which is likely important for near term microprocessors. Hybrid branch predictors achieved even lower misprediction rates, under 4% for 64 KB predictors, and promise accuracy growth well beyond the sizes that were investigated here.

CHAPTER 5

Self Correlation

Many branches are predictable based on the history of their own past outcomes. These may be loop branches, branches that are strongly biased in one direction, or branches that in some other way follow predictable execution patterns. For these branches, the next outcome is correlated to the previous outcomes in their own execution histories. We call this type of correlation *self correlation*. This is different from *branch correlation* where correlation with other branches is also considered. Branch correlation will be discussed in Chapter 6. In this chapter, the per-branch execution patterns are classified, both in general terms, and in terms of the history seen by per-address branch predictors such as PAs. Through this classification it is shown that 67% of all branches are more than 99% predictable using self correlation. Based on this, a new prediction mechanism, loop filtering, is introduced. The loop filtering mechanism is used together with existing predictors to increase accuracy.

5.1 Classes of Patterns and Their Characteristics

A large number of branches follow repeating branch execution patterns. An example of a repeating branch execution pattern is **110110110**.[1] In this section, patterns are classified to show the extent and usefulness of repeating branch execution patterns. Two of the classes represent patterns similar to those generated by for and while loops. The other classes show the extent of other repeating patterns. The frequency and predictability of these patterns are shown.

[1] 0 represents a not taken branch outcome and 1 represents a taken branch outcome. The leftmost digit represents the oldest outcome, and the rightmost digit the youngest.

5.1.1 Classes of Patterns

We divide branch patterns into six classes. Each instance of a branch is put in the class corresponding to the pattern seen in its recent execution history. Within each of the classes, the branch is further categorized using the period of the pattern and the number of times the pattern has repeated.[2] The following six classes of patterns are considered:

- *Biased* pattern (Example **111111**): This pattern consists of a string of taken or a string of not taken branches. This type of pattern accounts for the majority of branches. The period of this pattern is always 1.

- *Alternating* repeating pattern (Example **101010**): This pattern consists of every other outcome being taken, and every other outcome being not taken. This type of pattern accounts for 1.9% of all branches. The period of this pattern is always 2.

- *For*-type repeating pattern (Example **110110110**): This pattern consists of n, where $n > 1$, consecutive taken outcomes followed by a single not taken outcome, with this sequence of $n + 1$ outcomes repeating. This type of pattern accounts for 17.5% of all branches. The period of this pattern is $n + 1$. This pattern is named for-type as it is consistent with the behavior of the loop ending branch of a for loop with a constant loop count. However, other branches exhibiting the same behavior may also fall into this category.

- *While*-type repeating pattern (Example **001001001**): This pattern consists of n, where $n > 1$, consecutive not taken outcomes followed by a single taken outcome, with this sequence of $n + 1$ outcomes repeating. This type of pattern accounts for 3.5% of all branches. The period of this pattern is $n + 1$. This pattern is named while-type as it is consistent with the behavior of the loop branch of a while loop with a constant loop count. However, more frequently the branches in this category will be if constructs that follow a repeating pattern of this type. Many current compilers convert while loops into for loops during compilation, which reduces the frequency of these patterns.

[2]The period of a pattern is the length of the substring that is repeating. For example, the period of the pattern **1101101101** is 3. The complete substring has been seen 3 times, so we say it has repeated 2 times.

- *Simple* repeating pattern (Example **1100011000**): This pattern consists of n consecutive not taken outcomes followed by m consecutive taken outcomes, where $n > 1$ and $m > 1$, with this sequence of $n + m$ outcomes repeating. This type of pattern accounts for 0.5% of all branches. The period of this pattern is $n + m$.

- *Complex* repeating pattern (Example **1101011010**): Any repeating n-bit pattern that does not fall into any of the above classes. This type of pattern accounts for 1.9% of all branches. The period of this pattern is n. Due to the complexity of detecting these patterns, only complex patterns with periods smaller than or equal to 30 were detected for the experiments in this dissertation.

These classes are further split into *stable* and *transient* patterns. When a branch is encountered during the run of a program, its past execution pattern is examined to determine the class and period of the pattern it is currently following. If the branch has always in the past followed exactly the same class of pattern with the same period, the pattern is said to be *stable*. If the branch has previously behaved differently, the pattern is said to be *transient*.

A substring must be repeated at least once at the end (the most recent part) of the branch history for the branch to be put in the class corresponding to that substring. For example, the pattern **1110111** is not a for-type pattern as it has not yet repeated. Instead it is considered a transient biased pattern because of the three consecutive taken outcomes. However, the pattern **111011101** is a for-type pattern.

5.1.2 Classifying a Branch Instance Using Complete History

This section explains how each instance of a branch is examined during simulation to determine the class and period of the current pattern along with how many times the pattern has repeated. In this section, the classification is not restricted by the length of history needed to detect a pattern.

For simulation purposes, the pattern followed by a particular branch is tracked using a set of counters and other history information kept in association with the branch target buffer (BTB). This is done to capture statistics on the frequency of occurrence of the pattern classes, and an implementation for use on a processor is not implied. Since the information is stored in the BTB, it may be lost due to contention and some branches may be classified

slightly differently depending on the BTB size. Two BTB configurations are simulated: A 16-way set associative 16 K entry BTB which suffers virtually no information loss, and a 4-way set associative 2 K entry BTB which better indicates which patterns can be detected at run-time in a processor.

The class, period and pattern-type of a branch is detected before it is executed using the history information in the BTB. Statistics based on the execution of the branch, such as frequency, taken rate and accuracies, are accounted for in the class the branch was determined to belong to before the execution. For instance, a branch which prior to execution was determined to follow a for-type pattern (prior history **1101101**), but currently goes in a direction (not-taken or **0**) that is inconsistent with that pattern type will still be accounted for in the class, for-type, that was determined prior to execution.

When a branch is inserted into the BTB, it is placed in the stable biased category. It is then moved over to the transient biased category the first time it goes in the opposite direction. It is not until a pattern has repeated at least once that it is a candidate for the repeating pattern classes. At that time, it is placed in the stable class if that pattern has repeated for the entire execution history (as kept in the BTB), otherwise it is placed in the transient class.

Biased patterns are detected by having a bit, $Direction$, showing the direction of the previous outcome of that branch, and a counter, $Count$, which counts how many consecutive times the branch has gone in that direction. $Count$ is incremented if the current outcome is the same as the outcome in $Direction$, otherwise $Count$ is cleared to zero. In addition, there is a counter, $Num_Changed$, for how many times the branch has changed direction. If the branch has never changed direction, the pattern is stable biased. If it has changed direction one or more times, the pattern is transient biased.

In addition to the fields used to detect biased patterns, a few more fields are needed to also capture alternating, while-type, for-type, and simple repeating patterns. Two fields, $Num_Last[NotTaken, Taken]$, hold the length of the previous string of not-takens and the length of the previous string of takens.[3] An additional counter, Num_Same, counts how many consecutive times a string of not-takens or takens was the same length as the previous such string. When a branch changes direction, $Count$ is compared to $Num_Last[Direction]$. If they are equal, Num_Same is incremented, otherwise Num_Same is cleared to zero. If

[3]Example: If the pattern seen so far is **111011**, $Num_Last[NotTaken]$ is 1 and $Num_Last[Taken]$ is 3.

NotTaken	Taken	Pattern Type
1	1	Alternating
1	>1	For-type
>1	1	While-type
>1	>1	Simple repeating

Table 5.1: Determining repeating pattern type based on values in Num_Last

Num_Same is two or higher, the pattern has repeated, and the class and period of the pattern can be found by examining $Num_Last[NotTaken]$ and $Num_Last[Taken]$ as shown in Table 5.1. If Num_Same is exactly two less than $Num_Changed$, the pattern is stable (the first two times the direction changes initialize the Num_Last counts, thereafter the pattern has repeated as the same). If Num_Same is less than two, no repeating pattern has been detected.

Complex patterns are harder to detect, and the method is only outlined here. All repeating patterns with a period of 30 or less are captured by this method. A history of the 30 most recent outcomes is kept, and one counter is associated with each of the history bits. If a specific history bit is the same as the current outcome, its associated counter is incremented. Otherwise, the counter is reset to zero. If the counter associated with the kth most recent history bit is larger than k a repeating pattern with period k has been detected. If patterns of several periods are detected, the one that has lasted longer (the larger value in the counter) is considered the true pattern. The number of times the pattern has repeated is the value in the counter divided by k.

Given these rules, a pattern may in some cases fit in several classes. In that case, the class which had the longer period is used, with alternating, for-type, while-type, and simple repeating taking precedence over complex repeating if the period is the same.

To clarify this, some examples will be given. These patterns all started after a BTB-miss.

- **111**: Stable biased, repeated 2 times.
- **1110**: Transient biased, repeated 0 times.
- **11101110**: Transient biased, repeated 0 times. Repeating patterns are detected based on full strings of ones or zeroes, so the direction needs to change before this pattern is detected.
- **111011101**: Stable for-type, repeated 1 time, period 4.

- **11111011101**: Transient biased, repeated 0 times. This is not identified as a repeating for-type pattern due to the extra leading ones, which make the first string of ones length 5.

- **11010110101**: Stable complex, repeated once, period 5.

5.1.3 Distribution of Branches by Pattern Class

The importance of each type of pattern is partly determined by its frequency of occurrence. Figure 5.1 shows the distribution of dynamic branch instances based on their pattern class. The figure is for a 16 K entry BTB, so the classification process was not distorted by contention in the BTB. The biased class, which represents 75% of all branches, was further broken down into its transient and stable categories as these have substantially different behavior. The for-type repeating patterns account for an additional 17.5% of all branches, and while-type patterns account for 3.5%. The alternating pattern class, which is similar to both for and while type patterns, accounts for 1.9% of the branches. Complex repeating patterns account for 1.9%, and simple repeating patterns account for the remaining 0.5%.

The same information is also given in Table 5.2, here with the transient and stable components separated out for all of the classes. One observation that can be made from the table is that about half of the biased, while-type, and for-type patterns are stable, whereas nearly all of the alternating, simple, and complex repeating patterns are transient. Transient patterns are more likely to change, so they are not as useful for prediction. From the perspective of individual benchmarks, go and ijpeg have fewer stable biased patterns than average, while perl and vortex have more stable biased patterns than average. Compress has 8 times more while-type patterns[4] than any other benchmark. Ijpeg and m88ksim have more for-type patterns than average.

The same experiment was also run with a 2 K entry BTB to give a better indication of which patterns can be detected at run-time. Figure 5.2 and Table 5.3 show the pattern distribution for this BTB size. The most significant change caused by reducing the BTB size is that there is a slight shift from transient to stable within each class. This is because a pattern is considered stable until it changes behavior. With a smaller BTB, a branch gets

[4]The transient while-type patterns are due to three related if-constructs that are dependent on the number of bits that are examined in a particular function. The stable while-type patterns are due to the two main while loops in the compress and decompress functions, and an if-statement closely related to the condition for the while loop in the compress function.

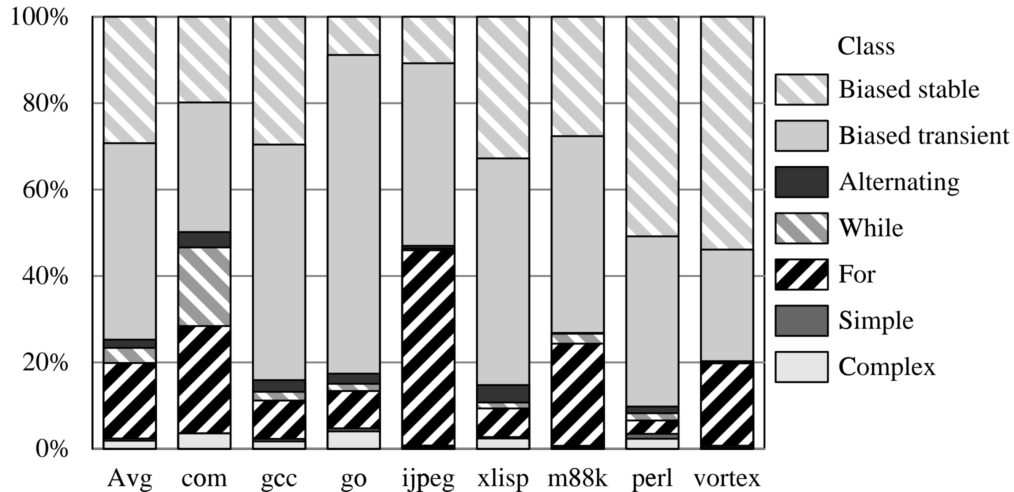


Figure 5.1: Distribution of branches by pattern class. 16 K entry BTB

	Transient Patterns					
	Biased	Alternating	While	For	Simple	Complex
compress	30.0%	3.6%	6.7%	14.7%	0.0%	3.6%
gcc	54.5%	1.9%	1.6%	6.3%	0.6%	1.7%
go	73.8%	2.0%	1.7%	6.2%	0.7%	4.0%
ijpeg	42.3%	0.6%	0.2%	6.5%	0.2%	0.3%
m88ksim	45.6%	0.1%	2.2%	6.0%	0.3%	0.3%
perl	39.4%	1.2%	1.7%	2.9%	1.1%	2.3%
vortex	25.8%	0.2%	0.2%	19.0%	0.5%	0.3%
xlisp	52.5%	4.0%	1.3%	0.6%	0.3%	2.4%
average	45.5%	1.7%	2.0%	7.8%	0.5%	1.9%
	Stable Patterns					
	Biased	Alternating	While	For	Simple	Complex
compress	19.8%	0.0%	11.6%	10.1%	0.0%	0.0%
gcc	29.6%	0.8%	0.4%	2.6%	0.0%	0.0%
go	8.8%	0.3%	0.0%	2.5%	0.0%	0.0%
ijpeg	10.8%	0.1%	0.2%	38.7%	0.2%	0.1%
m88ksim	27.6%	0.1%	0.1%	17.7%	0.0%	0.0%
perl	50.8%	0.4%	0.0%	0.1%	0.0%	0.0%
vortex	53.9%	0.0%	0.1%	0.1%	0.0%	0.0%
xlisp	32.8%	0.0%	0.0%	6.0%	0.0%	0.0%
average	29.3%	0.2%	1.5%	9.7%	0.0%	0.0%

Table 5.2: Pattern breakdown for 16 K entry BTB

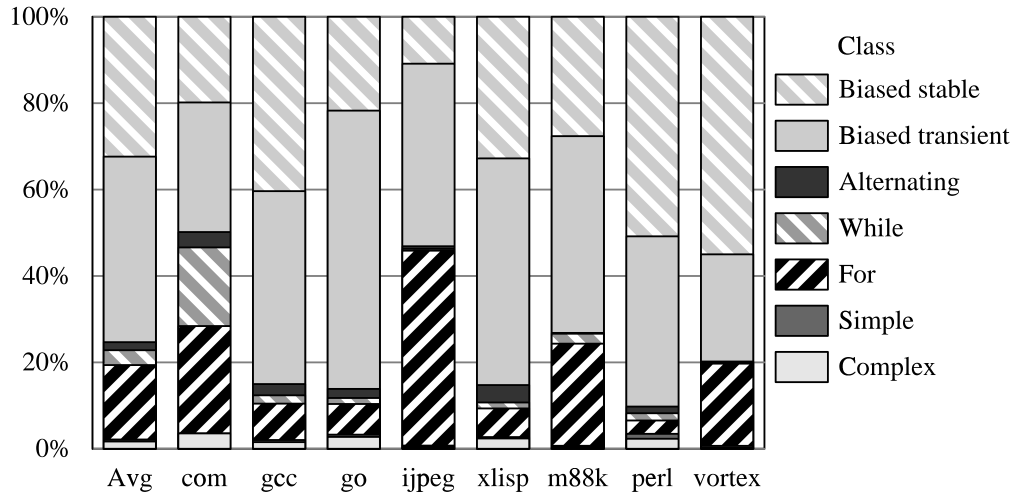


Figure 5.2: Distribution of branches by pattern class. 2 K entry BTB

	Transient Patterns					
	Biased	Alternating	While	For	Simple	Complex
compress	30.0%	3.6%	6.7%	14.7%	0.0%	3.6%
gcc	44.7%	1.6%	1.3%	5.6%	0.5%	1.5%
go	64.4%	1.6%	1.3%	4.4%	0.5%	2.6%
ijpeg	42.3%	0.6%	0.2%	6.5%	0.2%	0.3%
m88ksim	45.6%	0.1%	2.2%	6.0%	0.3%	0.3%
perl	39.4%	1.2%	1.7%	2.9%	1.1%	2.3%
vortex	24.8%	0.2%	0.2%	18.9%	0.4%	0.2%
xlisp	52.5%	4.0%	1.3%	0.6%	0.3%	2.4%
average	42.9%	1.6%	1.9%	7.4%	0.4%	1.6%
	Stable Patterns					
	Biased	Alternating	While	For	Simple	Complex
compress	19.8%	0.0%	11.6%	10.1%	0.0%	0.0%
gcc	40.4%	1.0%	0.7%	2.9%	0.0%	0.1%
go	21.7%	0.5%	0.1%	2.8%	0.0%	0.2%
ijpeg	10.9%	0.0%	0.2%	38.6%	0.2%	0.1%
m88ksim	27.6%	0.1%	0.1%	17.7%	0.0%	0.0%
perl	50.8%	0.4%	0.0%	0.1%	0.0%	0.0%
vortex	55.0%	0.0%	0.1%	0.2%	0.0%	0.0%
xlisp	32.8%	0.0%	0.0%	6.0%	0.0%	0.0%
average	32.4%	0.3%	1.6%	9.8%	0.0%	0.1%

Table 5.3: Pattern breakdown for 2 K entry BTB

a new chance at being considered stable each time it is re-entered into the BTB. The other difference with the smaller BTB is that fewer repeating patterns were detected, especially for benchmarks with a large footprint. This is because a pattern must repeat at least once before it is considered a repeating pattern. This may not happen if the branch is frequently discarded from the BTB and the period of the pattern is large.

With a 2 K entry BTB, 32% of all branches were classified as stable biased and an additional 10% of the branches were classified as stable for-type. These branches are almost 100% predictable assuming, of course, that the right predictor is used. For all the remaining experiments, a 2 K entry BTB will be used so that the results can more easily be applied to real predictor designs.

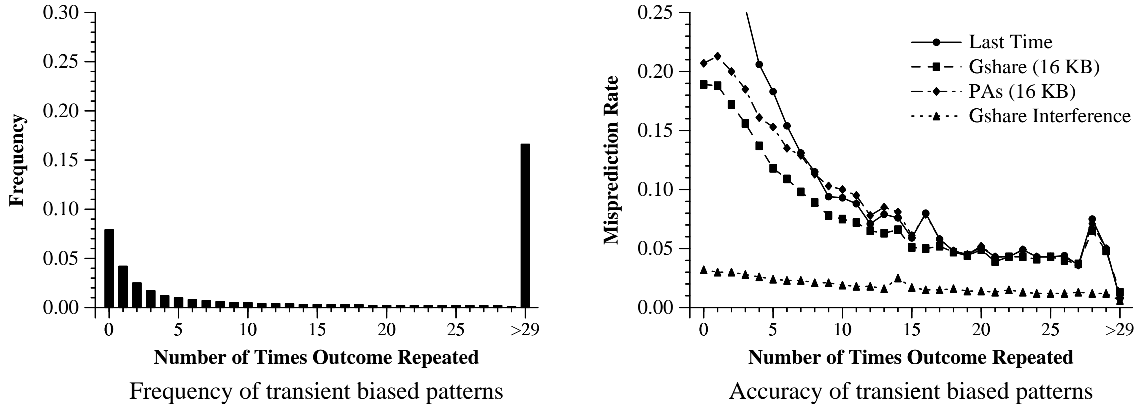
5.1.4 Characteristics of Biased Patterns

Biased patterns account for 75% of all branches. This section examines the characteristics of these patterns. A 2 K entry BTB is used for these experiments. However, the results for a 16 K entry BTB are almost identical. To understand the behavior of the branches in the biased class, the class is further categorized based on how many consecutive times the same outcome has been repeated.

Figure 5.3 shows the frequency and accuracy of transient biased branches depending on how many consecutive times the same outcome has been repeated. The category of branches for which the outcome has repeated zero times consists of branches that have gone in the current direction once, and so on. The leftmost graph in Figure 5.3 shows the frequency, as a fraction of all branches, of transient biased patterns. The rightmost graph shows the accuracy of three predictors for these patterns. The first predictor is the "Last-Time" predictor. The prediction of the Last-Time predictor is the direction the branch took the previous time. The other two predictors are 16 KB versions of the Gshare and PAs two-level adaptive predictors. These are shown to identify how the predictability of a branch varies with the number of consecutive times it has had the same outcome.

Figure 5.3 also shows the amount of PHT interference in Gshare caused by these transient biased branches. The magnitude of the destructive interference is shown as the "Gshare interference"[5] curve. If the branches in the transient biased category were predicted using

[5]The amount of constructive interference was subtracted from the amount of destructive interference to arrive at the number for "Gshare interference".



Frequency of transient biased patterns

Accuracy of transient biased patterns

Figure 5.3: Characteristics of transient biased patterns

a different predictor, this interference could be removed from the Gshare predictor.

Transient biased branches account for 42.9% of all branches. Of these, half have repeated the same outcome 10 times or fewer, and 39% have repeated the same outcome 30 times or more. As can be seen from the rightmost graph in Figure 5.3, transient biased patterns are generally not very predictable. Even after an outcome has been repeated 20 times, the misprediction rate is almost 5%, regardless of which predictor is used. The history of the PAs predictor is completely filled with either zeroes or ones after 15 repeated outcomes. After that, the accuracies of the PAs and Last-Time predictors track closely. Before that, the history mechanism allows PAs to perform better than the Last-Time predictor. The Gshare predictor achieves a much lower misprediction rate than the PAs and Last-Time predictor if the same outcome has been repeated only a few times. After the same outcome has been repeated approximately 20 times, the misprediction rates of the three predictors are very close. In the 30 times or more category, the misprediction rate for the PAs and Last-Time predictors is 0.8%, whereas the misprediction rate for Gshare is 1.3%.

The misprediction rates of the PAs and Last-Time predictors at 16 and 28 repeated outcomes are higher than the trend suggests. This is due to branches in the m88ksim benchmark.[6]

[6]The peak at 16 is due to three related if statements in the check_scoreboard and execute functions, and one branch checking the segment type in the checklmt function. The peak at 28 is due to a branch used to find the minimum of two values in the killtime function.

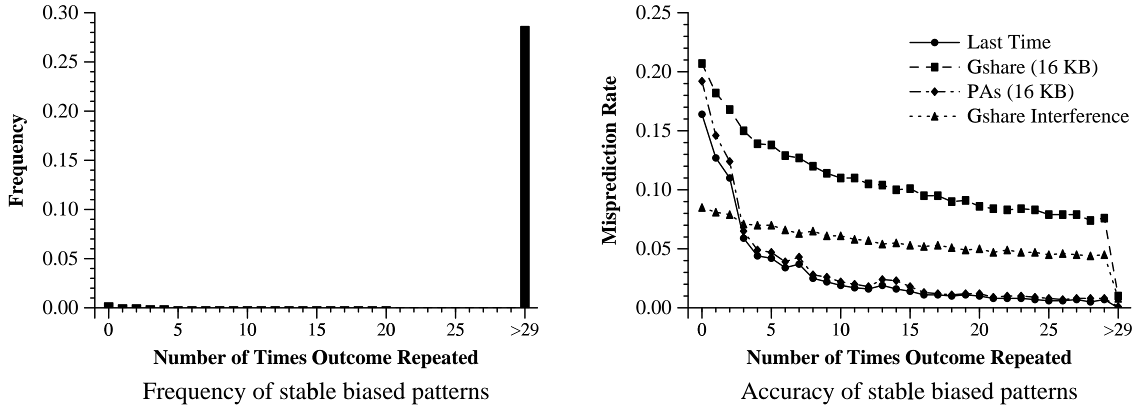


Figure 5.4: Characteristics of stable biased patterns

Figure 5.4 shows similar data for the stable biased patterns. It is clear that the characteristics of these patterns are very different. These patterns represent branches that have always taken the same direction, so they are very likely to continue taking the same direction. Out of the 32.4% of all branches that follow stable biased patterns, 88% have repeated the same outcome 30 or more times. The effect of the stability of these patterns is also seen in the accuracy of the Last-Time predictor. After the outcome has been repeated only 4 times, the misprediction rate drops below 5%, after it has been repeated 18 times, the misprediction rate drops to 1.0%, and for the 88% of the stable biased patterns where the outcome has repeated 30 or more times, the misprediction rate is 0.0%. Figure 5.4 also shows that for stable biased branches, the Last-Time predictor is always slightly better than PAs. This is because these branches are being seen for the first few times (or for the first times for a while after they were kicked out of the BTB), so the PAs predictor does not have the advantage of history information relating specifically to these branches. This lack of history information is even more devastating to the Gshare predictor, which does poorly for these branches. These branches also cause a lot of interference in the Gshare predictor. On average, 5-9% of the accesses these branches make in the Gshare predictor cause destructive interference, with the exception being when the outcome has been repeated 30 times or more. For patterns that have repeated the same outcome 30 times or more, the misprediction rate of the Last-Time predictor is 0.0%, PAs 0.1%, Gshare 1.0%, and the interference caused in Gshare is 0.8%.

Maybe the most important conclusion from this section is that 28.6% of all branches follow stable biased patterns with the same outcome repeated 30 times or more, and 16.6% of all branches follow transient biased patterns with the same outcome repeated 30 times or

more. For these branches, the Last-Time or PAs predictors are naturally enough superior to the Gshare predictor. However, for the 15.8% of branches that have shorter transient biased patterns, the Gshare predictor is better than Last-Time and PAs.

5.1.5 Characteristics of Repeating Patterns

The characteristics of the classes of repeating patterns[7] are shown here. First, the behavior averaged over all classes is shown. Then, more detailed characteristics are shown for the three most frequent classes: For-type, while-type, and alternating repeating patterns.

Average Characteristics

As explained earlier, each of the repeating pattern classes were further categorized based on the period of the pattern and the number of consecutive times that the pattern has repeated. So, a pattern of **1110111011101** is classified as a for-type pattern, with period 4 having repeated 2 times. For the average characteristics examined here, the behavior is averaged for all repeating patterns (the biased class excluded), but patterns are still categorized based on their period and the number of times the pattern has repeated.

When examining the characteristics of the repeating patterns, one type of graph will be used repeatedly. The first two such graphs are shown in Figure 5.5. These graphs show data in three dimensions. Common for all of these graphs is that the x-axis shows the number of times the pattern has repeated, and the y-axis shows the period of the pattern. The third dimension, generally the frequency or accuracy of that type of pattern is given by the color (or more accurately, the darkness) of the point. Categories which account for less than 0.002% of all branches are shown as the lightest shade of gray in all graphs. The top row shows all patterns with a period of more than 30, and the rightmost column shows all patterns that have repeated more than 14 times. All of these graphs come in pairs, with the left graph being for transient patterns and the right graph being for stable patterns.

Figure 5.5 shows the frequency of repeating patterns. The darker areas show the most frequent patterns, whereas the lighter areas show the least frequent patterns. The frequency, given as the fraction of all branches, represented by a shade is given in the key on the right. As an example of how to read the graph, the square representing transient patterns (left graph) repeated more than 14 times (>14 on x-axis) with a period of 8 (8 on y-axis) is

[7]Repeating patterns refers to all patterns, except those that fall in the biased class.

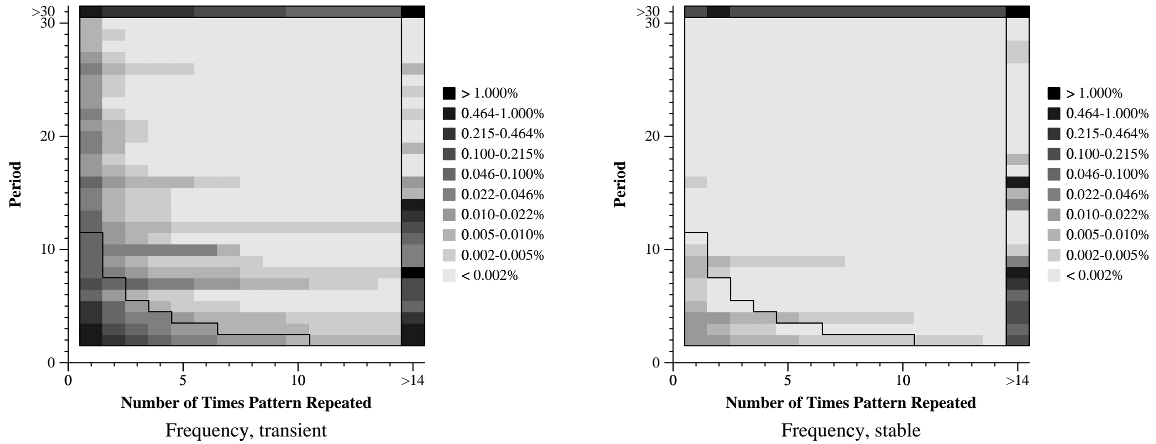


Figure 5.5: Frequency of repeating patterns

black. As can be seen from the key, these patterns represent over 1% of all branches.

All transient repeating patterns together account for 13.0% of all branches, while stable repeating patterns account for 11.7%. Of these, 8% of the transient patterns and 60% of the stable patterns are patterns with a period longer than 30 that have repeated more than 14 times. This is the black square in the top right of the graphs.

To simplify the discussion of the repeating patterns, we define two regions. One region covers all of the patterns that have repeated more than 14 times or have a period longer than 30. This region is delineated in Figure 5.5 and consists of the top row and the right column. The region will be referred to as the *high confidence* region, as these patterns are very likely to continue. The high confidence region covers most of the stable patterns. For transient branches, a vaguely triangular shape of darker shades can be seen to the bottom left of Figure 5.5. This is because a number of patterns repeat only a few times before changing period or class, more so for the shorter periods. Shorter patterns are more likely to accidentally repeat than longer patterns. For instance, a pattern of **101010** could happen by accident on a fairly random branch, whereas **111110111110111110** is much less likely to happen by accident. The darker triangular region will be referred to as the *low confidence* region, as it can not confidently be said that these patterns will continue to repeat. For purposes of this dissertation, the triangular region is chosen to start with patterns repeated once having period 11, and ending with patterns repeated 10 times having period 2. In between, all patterns for which $(((\#repeated + 1) \times period) <= 22)$ is true are considered to be in the region.[8] This region is also delineated in Figure 5.5. The patterns

[8]The pattern length 22 was chosen as this coincides closely with the region for which the probability of

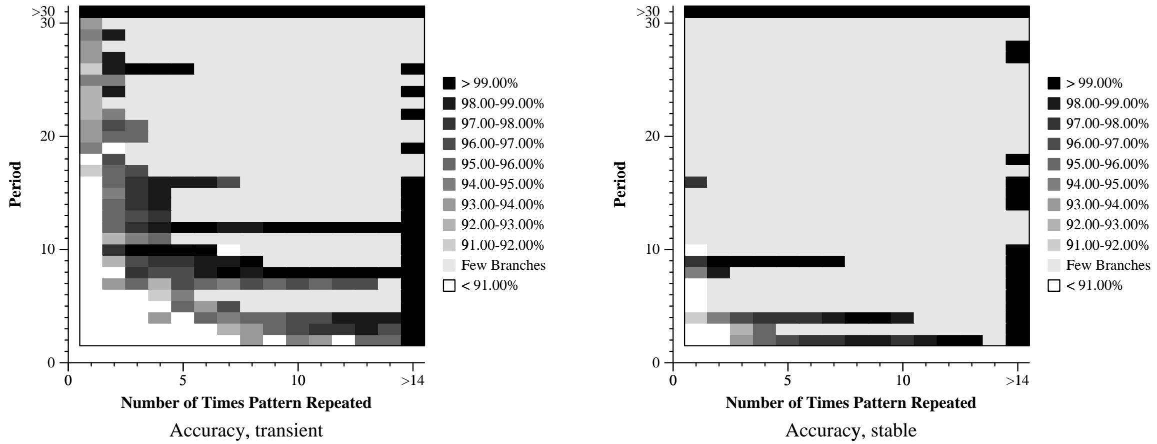


Figure 5.6: Prediction accuracy of repeating patterns

within the low confidence region account for a quarter of all transient repeating patterns.

Figure 5.6 shows the prediction accuracy of the repeating patterns. This is the accuracy of predicting that the pattern will continue repeating as before. The longer the period, and the more times the pattern has repeated, the higher the accuracy. Patterns in the high confidence region, the top row and left column, can be predicted with more than 99% accuracy. The branches in the low confidence region are generally less than 91% predictable. Stable repeating patterns are always more predictable than transient repeating patterns.

The average results given above provide a general view of the characteristics of repeating patterns. Stable patterns are more predictable than transient patterns, and long patterns are more predictable than short patterns. However, there is substantial variation between the different classes. Table 5.4 shows the frequency and accuracy of each class of repeating patterns. Furthermore, the table shows what fraction of the patterns in each class are in the low and high confidence regions, and the average prediction accuracies of these patterns. Perhaps the main conclusion that can be drawn from this table is that for-type patterns are consistently more predictable than other patterns, with while-type patterns coming in second. The high confidence region is very predictable for all classes, and accounts for 63% of the transient patterns and 97% of the stable patterns. The three most frequently encountered classes—for-type, while-type, and alternating repeating patterns—are examined separately and in more detail, and some of the numbers in Table 5.4 are discussed further then.

the pattern continuing is less than 91%, as will be shown shortly.

Class	Transient Patterns					
	Total		Low Confidence		High Confidence	
	Freq	Acc	Freq	Acc	Freq	Acc
All	13.0%	93.2%	25.5%	76.2%	62.8%	99.9%
Alternating	1.6%	81.6%	65.2%	72.2%	33.1%	99.5%
While-type	1.9%	93.2%	26.8%	76.3%	59.2%	99.9%
For-type	7.4%	96.7%	14.3%	79.6%	78.0%	99.9%
Simple	0.4%	90.4%	29.2%	71.3%	46.0%	99.4%
Complex	1.6%	89.1%	35.4%	78.4%	25.7%	99.9%
	Stable Patterns					
Class	Total		Low Confidence		High Confidence	
	Freq	Acc	Freq	Acc	Freq	Acc
All	11.7%	99.8%	1.4%	90.3%	97.3%	100.0%
Alternating	0.3%	97.3%	24.6%	89.3%	72.0%	99.9%
While-type	1.6%	99.9%	0.8%	85.8%	98.7%	100.0%
For-type	9.8%	99.9%	0.8%	93.2%	98.2%	100.0%
Simple	0.0%	97.8%	15.2%	85.7%	75.8%	100.0%
Complex	0.1%	94.8%	23.5%	85.4%	33.3%	100.0%

Table 5.4: Frequency and accuracy of patterns by region. Frequency in "Total" column is given as fraction of all branches. All other frequencies are fraction of branches in that class.

Characteristics of For-Type Patterns

The for-type patterns account for 70% of all remaining patterns when excluding the biased patterns, and therefore make up the second most important class. There are six sets of graphs showing the characteristics of for-type patterns. Each set contains one graph for transient patterns and one for stable patterns. These will be explained in turn. The first two sets in Figure 5.7 show the frequency and accuracy of for-type patterns, and are similar to Figure 5.5 and 5.6. Transient for-type patterns account for 7.4% of all branches and stable for-type patterns account for 9.8% of all branches. Of these, 10% of the transient and 65% of the stable patterns have a period longer than 30 and have repeated over 14 times (the top right point). This is slightly more than for the other classes of repeating patterns. The entire high confidence region is also larger, 78% of transient and 98% of stable patterns, than for the other classes. Furthermore, the low confidence region is smaller, 14% of transient and 1% of stable patterns. The for-type patterns in the low confidence region are also slightly more predictable than for other classes as shown in Table 5.4. Finally, the fraction of for-type patterns that are stable is 57%, as compared to 25% on average for other repeating patterns. This indicates that for-type patterns are more stable and more

predictable than other repeating patterns.

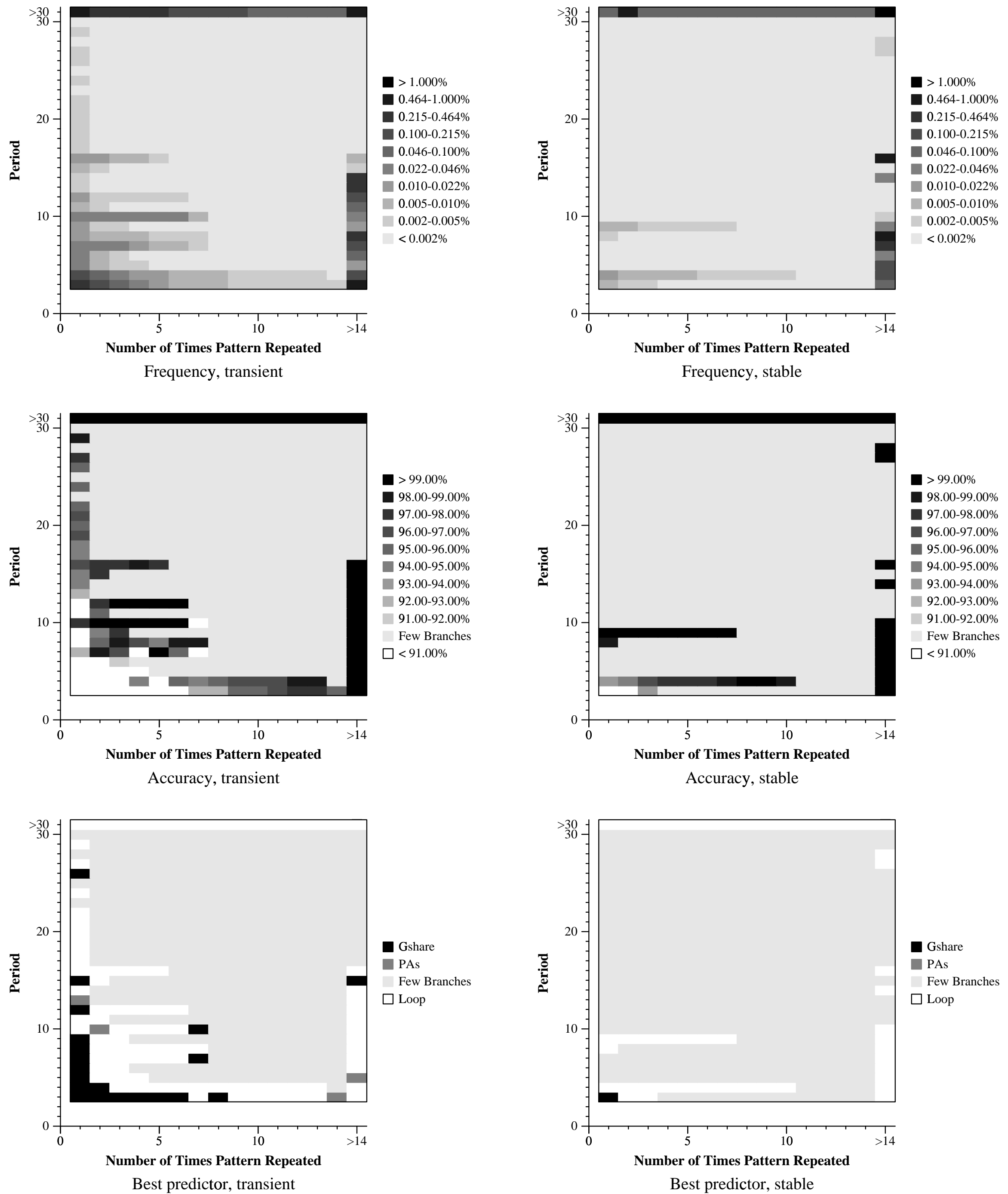


Figure 5.7: Frequency, accuracy, and best predictor. For-type patterns

The final set of graphs in Figure 5.7 shows which predictor is best suited for predicting the for-type patterns. 16 KB PAs and Gshare predictors are compared against a 4 KB loop predictor. The loop predictor[9] merely predicts that the pattern will continue, and its accuracy is the same as that given in the "Accuracy" graphs in Figure 5.7. For most except the shortest transient patterns, the loop predictor is better. For stable patterns, the loop predictor is better for all but one data point, patterns with a period of three having repeated once. Note also that for the shortest transient patterns, Gshare is best, not PAs. This is because for this type of patterns, the loop predictor is generally better than PAs (the additional adaptivity that PAs provides does not help), so if Gshare is better than the loop predictor, it is also better than PAs. However, this does not show the magnitude of the difference between the loop predictor and Gshare or PAs.

The top set of graphs in Figure 5.8 shows the improvement of the loop predictor over Gshare. A black square indicates that the prediction accuracy of the loop predictor is 1.6% higher than that of Gshare. Of special interest are the top row and the rightmost column as these are the most frequent patterns. For the transient patterns with a period longer than 30, the top row, the improvement over Gshare is 0.5%. For the rest of the transient patterns, the trend appears to be that the more times a pattern has repeated, the better the improvement. For the stable patterns with a period longer than 30 which have repeated more than 14 times (65% of the stable patterns), the improvement is just under 1.6%. The improvement is substantial for almost all of the stable patterns. The interference caused by these branches in the Gshare predictor is shown in the middle set of graphs in Figure 5.8. The branches with shorter periods cause more interference, and there is little difference between transient and stable branches.

[9] The implementation of the loop predictor will be given in Section 5.5.

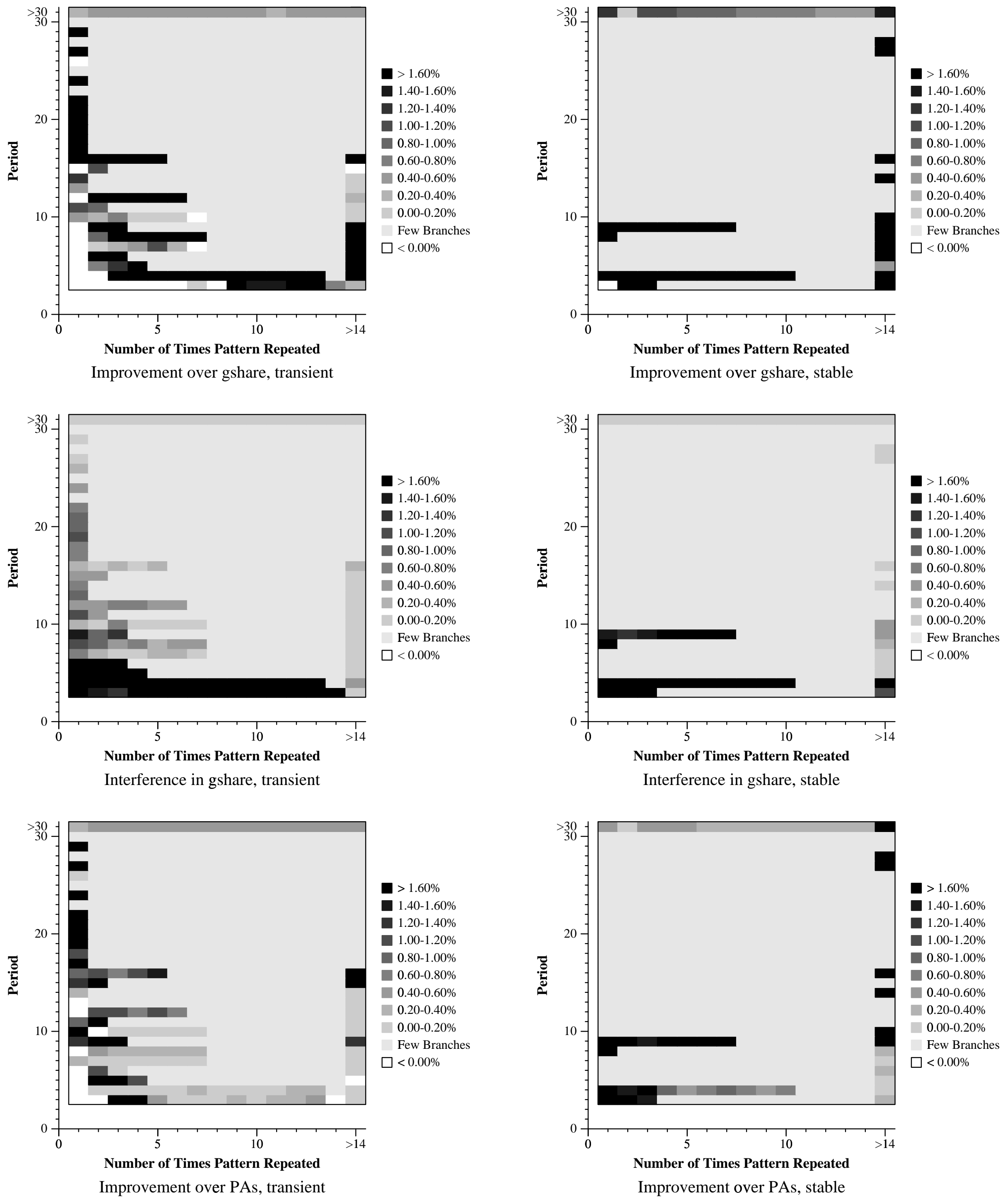


Figure 5.8: Improvement over and interference for Gshare, improvement over PAs. For-type patterns

The final set of graphs in Figure 5.8 shows the improvement of the loop predictor over PAs. The improvement over PAs is generally smaller than the improvement over Gshare. For PAs, the largest improvement is seen when the period of the pattern is longer than the history (the improvement is smaller for very large periods as PAs generally only mispredicts one outcome). There is also an improvement for those patterns that have repeated a few times, but don't fill the entire history (e.g. period 3, repeated 3 times). For stable patterns the loop predictor is substantially better for the crucial category of patterns that have repeated more than 14 times.

In summary, for-type patterns are frequent, accounting for 17.2% of all executed branches. Most of these are very predictable, with accuracies for the loop predictor close to 100%. The Gshare predictor is not very good at predicting these branches. PAs is good, but not as good as the loop predictor for patterns with long periods.

Characteristics of While-Type and Alternating Patterns

The while-type patterns account for 14% of all repeating patterns, and alternating patterns account for 8% of all repeating patterns. As alternating patterns are similar to while-type[10] patterns but always with a period of 2, they are treated together here. For the remainder of this section, while-type and alternating patterns will be referred to collectively as while-type patterns. The graphs are similar to those in the previous section. The frequency and accuracy of while-type patterns can be seen in the top two sets of graphs in Figure 5.9. Transient while-type, including alternating, patterns account for 3.5% of all branches, whereas stable while-type patterns account for 1.9% of all branches. Due to the lower frequency, while-type patterns are not as important as for-type patterns. The high confidence region covers 47% of the transient and 94% of the stable patterns. The low confidence region covers 44% of the transient and 5% of the stable while-type patterns. Except for the increased tendency of while-type patterns to be shorter, the accuracy of while-patterns is similar to that of for-type patterns.

[10] They are also similar to for-type patterns, but were included with the while-type patterns as this class is smaller.

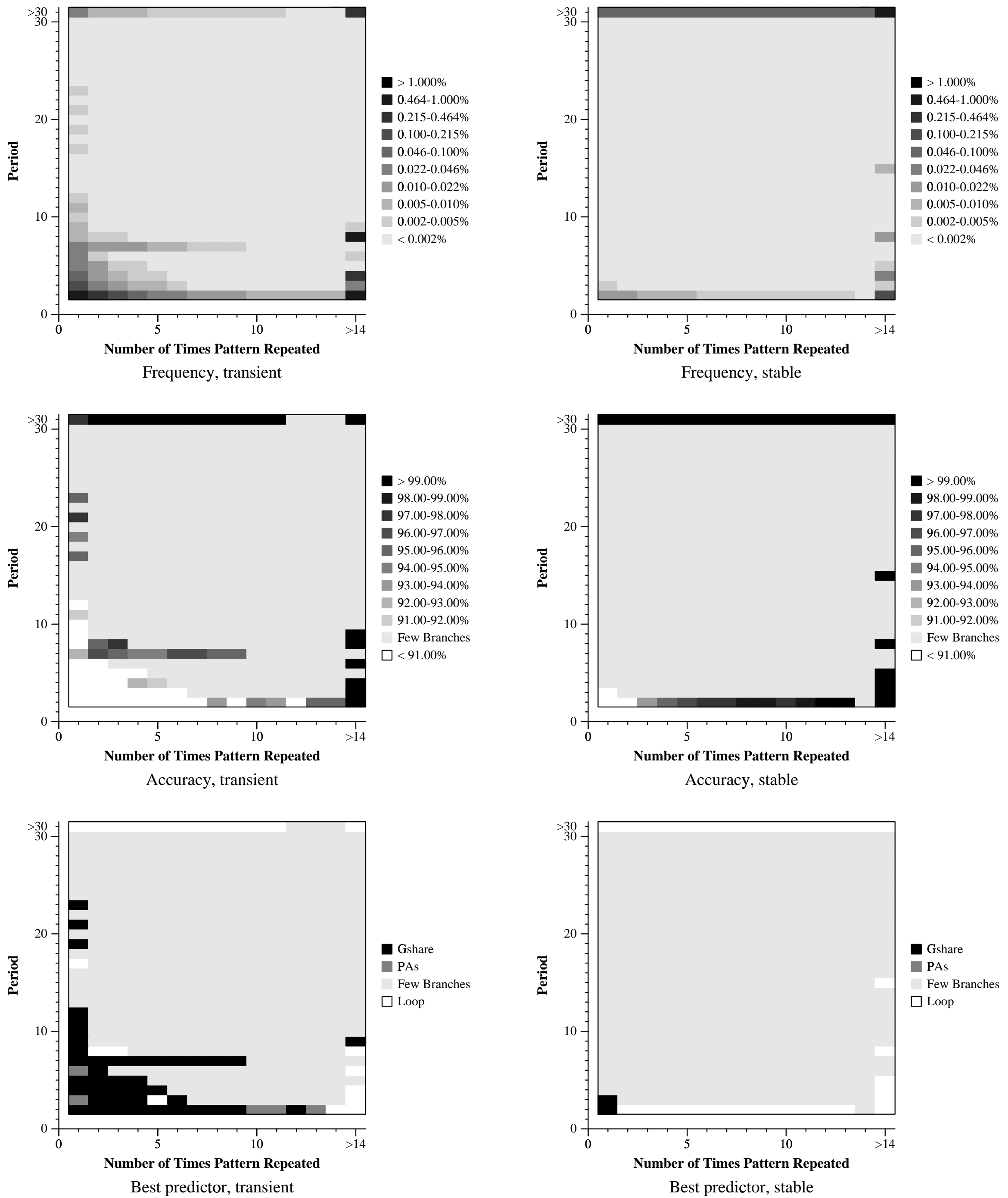


Figure 5.9: Frequency, accuracy, and best predictor. While-type patterns and alternating patterns

The final set of graphs in Figure 5.9 shows which predictor is better suited for predicting the while-type patterns. 16 KB PAs and Gshare predictors are compared against the 4 KB loop predictor. The loop predictor is best for transient patterns with long periods, and those that have repeated many times. However, the Gshare predictor is better than the loop predictor for a much larger range than was the case with for-type patterns. This is due to the Gshare predictor being more accurate for while-type patterns than it is for for-type patterns. Several of the dominant branches in this category, all from the compress benchmark, were identified and most of them were correlated with other branches preceding them. Most of the transient while-type patterns originated from if constructs. This is in contrast to the for-type patterns that mostly originate from loops. For stable patterns, the loop predictor is once again the best predictor.

The magnitude of the difference between the loop predictor and Gshare is shown in the top set of graphs in Figure 5.10. Of special interest are the top row and the rightmost column as these are frequent patterns. For most of these patterns, the loop predictor is substantially better than Gshare. For the most of the other transient patterns, Gshare is better. For the stable patterns, the improvement is more than 1.6% for most categories, except for the patterns with a period longer than 30. The majority of these have very long periods, in excess of 30,000. For these, the improvement is only around 0.1%. As with the for-type patterns, patterns with shorter periods tend to cause more interference. In addition, the transient patterns with a period longer than 30 cause a lot of interference in Gshare. The improvement over PAs, shown in the final set of graphs in Figure 5.10, is large for transient patterns with a period longer than 30. For other patterns, the improvement is smaller. As with for-type patterns, PAs is better for short patterns.

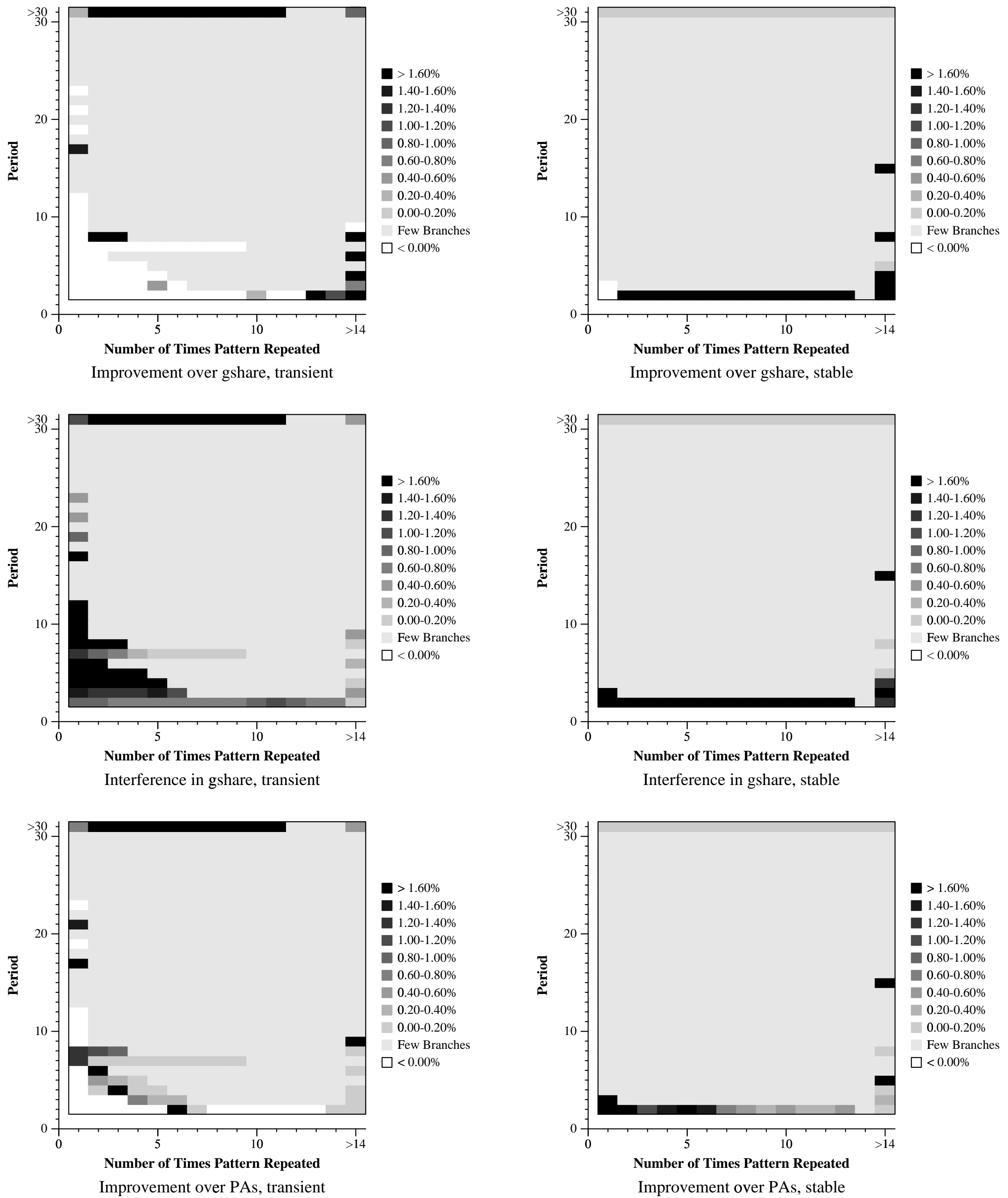


Figure 5.10: Improvement over and interference for Gshare, improvement over PAs. While-type patterns and alternating patterns

In summary, while-type and alternating patterns are less frequent than for-type patterns, together accounting for only 5.4% of all executed branches. Many of these patterns, 29%, are in the low confidence region. Furthermore, the Gshare predictor is fairly good at predicting these patterns, with the loop predictor having a substantial advantage only for a small subset. However, while-type patterns with long periods and/or that have repeated many times are, like similar for-type patterns, almost 100% predictable.

5.1.6 Summary

In this section, branch execution patters were classified and the frequency and predictability of each class was quantified. It was shown that a large number of easily identifiable branches are very predictable using a simple predictor. Stable biased patterns that have repeated the same outcome 30 times or more account for 29% of all branches and are 100.0% predictable. Transient biased patterns that have repeated the same outcome 30 times or more account for 17% of all branches and are 99.2% predictable. Repeating patterns are very frequent, and account for 25% of all branches. High confidence repeating patterns account for 20% of all branches, and are on average more than 99.9% predictable. Using this classification method, we can easily and dynamically identify a set of branches that are at least 99.9% predictable that accounts for 48% of all branches.[11] We can also identify an additional set of branches that are at least 99% predictable that accounts for 17% of all branches. If these are predicted using a special predictor, almost two thirds of all branches are taken care of and guaranteed to be predicted almost perfectly.

The largest class of patterns after the biased class is for-type patterns. These account for 17% of all branches, and are more predictable than other patterns. An additional 3.5% are while-type patterns.

5.2 Pattern Usage in Per-Address Predictors

In a per-address predictor, such as PAs, a per-address history pattern is used to select one of the entries in a pattern history table (PHT). Each distinct pattern maps to its own PHT-entry, which is then used to predict the branch. This entry can be a 2 or 3-bit

[11] The numbers given here use a 2 K entry BTB. A slightly larger number of predictable branches can be found using a larger BTB.

saturating counter, a static prediction derived from the history pattern, or any other state machine. In a manner similar to that used in Section 5.1, we can determine the class of the pattern used for selecting the PHT-entry. For instance, a counter selected using the pattern **110110110** is in the stable for-type category. Each counter uniquely represents one pattern.

In this section, patterns are classified in the way the PAs predictor sees them: With the limited history available in the history register. This differs from the method used in Section 5.1 in that it looks more closely at the way the PAs predictor works, including the imperfections caused by limited history. Each PHT-entry is classified based on the class of pattern that selects it. The frequency of use for PHT-entries in each class is presented along with the accuracy of the predictions.

This classification is used to provide insight into how a PAs predictor works, and to point out the skewed use of resources in that predictor. Specifically, it is shown that over 80% of all branches use only a small subset consisting of approximately 1% of the counters in the PHT.

5.2.1 Classifying a Branch Instance Using Limited History

Branches are classified using the same algorithm as in Section 5.1.2, although only the limited number of outcomes, 16 to 20 for these experiments, available in the per-address history are used. The same 6 classes, biased, alternating, while-type, for-type, simple repeating, and complex repeating, are used. Each can be transient or stable. In addition, since only limited history is examined and the period of many repeating patterns is long, patterns that could be parts of longer repeating patterns are separated out. These are called *potential* patterns. Only while-type, for-type and simple repeating patterns have a potential category. The history is long enough to guarantee that alternating patterns are always detected when present. All patterns are potentially part of a larger complex repeating pattern, so a potential complex pattern category does not make sense. Two examples of potential for-type patterns are **111101111** and **110111101**. The behavior is consistent with a for-type pattern, but the history is too short to know if the pattern has repeated.

In all of the experiments later in this section, only branches that have been seen 20 or more times after a BHT miss are counted. This leaves out 5% of the branches, but allows

Transient Patterns								
	General		Accuracy			Utilization		
Class	Freq	Counters	2-bc	3-bc	PSg	50%	90%	99%
Biased	15.7	60,432	81.2	81.7	65.6	5.7	52.3	90.0
Alternating	1.0	2,042	79.9	80.9	71.2	5.3	44.8	86.9
While	0.4	857	86.4	86.7	67.1	5.8	46.8	87.4
For	0.7	857	79.6	81.0	75.0	6.2	40.5	84.0
Simple	0.1	306	78.2	78.6	61.5	19.6	71.2	95.1
Complex	0.2	188	81.4	82.1	77.1	7.4	49.5	87.8
Potential Patterns								
	General		Accuracy			Utilization		
Class	Freq	Counters	2-bc	3-bc	PSg	50%	90%	99%
While	2.8	57	90.9	91.6	88.5	22.8	52.6	94.7
For	7.9	57	94.8	95.1	91.4	19.3	52.6	93.0
Simple	2.1	540	84.7	85.7	79.6	8.3	47.0	88.3
Stable Patterns								
	General		Accuracy			Utilization		
Class	Freq	Counters	2-bc	3-bc	PSg	50%	90%	99%
Biased	60.4	2	99.2	99.3	99.3	50.0	100.0	100.0
Alternating	0.8	2	98.1	98.3	98.3	50.0	100.0	100.0
While	0.6	20	96.3	96.5	96.2	20.0	60.0	95.0
For	2.0	20	97.4	97.6	97.6	20.0	70.0	100.0
Simple	0.1	48	89.6	90.6	87.9	22.9	66.7	95.8
Complex	0.4	108	86.1	86.9	82.0	13.9	43.5	87.0

Table 5.5: General statistics for PAs patterns

for more direct comparison between the experiments.

5.2.2 Distribution of Patterns and General Statistics

We here investigate the frequency and accuracy of predictions for each class of patterns, using a PAs predictor with a 16-bit history as a base. We also investigate how the PHT is utilized for these patterns. A 2 K entry BHT is used.

Table 5.5 has information about each class of patterns, given in three groups. The "General" group shows the frequency of each class, along with the number of counters in the PHT used by each class. For example, the transient biased class accounts for 15.7% of all branches, and uses 60,432 of the 65,536 counters in the PHT. The "Accuracy" group shows the accuracy of predicting these patterns using either 2-bit or 3-bit saturating counters in the PHT, or using a static prediction determined with the PSg(algo) algorithm [25]. Again examining the transient biased class, we see that these branches are predicted with 81.2% accuracy if the PHT consists of 2-bit saturating counters. The accuracy is slightly higher for 3-bit counters, and much lower for the statically determined predictions (PSg). Finally,

the "Utilization" group shows what percentage of the counters account for 50, 90, and 99% of the dynamic branches in this class. So, 50% of the branches in the transient biased class use only 5.7% of the counters belonging to this class. 90% of the branches use 52.3% of the counters, and so on.

First, let us examine the General statistics in the "Freq" and "Counters" columns. The "Freq" column shows the fraction (in percent) of all branches that are in that class. The "Counters" column shows how many of the counters in the PHT that are used (exclusively) by that class. Examining the numbers in these columns, we see that 60% of all branches are classified as stable biased. That is, the history is either all zeroes or all ones. This class accounts only for two of the counters in the PHT of a PAs predictor. 16% of all branches are in the transient biased class. However, this class uses over 60,000 counters, more than 90% of the PHT. Repeating patterns, both stable and transient are seldom recognized. However, 13% of all patterns are potential repeating patterns. About half of these are, as we will see in the following section, parts of longer actual repeating patterns but the history is to short to detect them. All in all, the transient classes account for only 18% of branches, but use almost 99% of all counters in the PHT.

The "Accuracy" columns show how well the branches in these classes are predicted using 2-bit or 3-bit saturating counters in the PHT, or using statically determined predictions in the PHT (PSg). Using 3-bit counters is always slightly better than using 2-bit counters. PSg works as well as 3-bit counters for stable classes, except for simple and complex repeating patterns. A PAs predictor using 2-bit counters in the PHT can be improved slightly if the PHT entries corresponding to stable biased, alternating, and for-type classes are tied to the predictions of the PSg(algo) predictor. However, PSg works poorly for transient and potential patterns as it cannot adapt to the behavior of these patterns. As was also found in Section 5.1, stable patterns are much more predictable than transient patterns.

The "Utilization" columns further show how most of the branches use only a small number of the PHT-entries. The 6% most frequent transient patterns (each of the patterns corresponding to one PHT-entry) account for 50% of the executed branches within the transient class. The 50% most frequent transient patterns account for 90% of the executed branches in the transient class. The PHT-entries in the transient classes are sparsely used to begin with (compared to the stable and potential pattern classes). The use of PHT-entries within the transient class is, as shown here, skewed towards a smaller number of dominant

Transient Patterns												
Class	Biased		Alternating		While		For		Simple		Complex	
	TR	ST	TR	ST	TR	ST	TR	ST	TR	ST	TR	ST
Biased	94.3										5.5	0.2
Alternating			94.1								5.9	
While					80.9						19.1	
For							91.6				8.4	
Simple									97.4		2.6	
Complex											100.0	
Potential Patterns												
Class	Biased		Alternating		While		For		Simple		Complex	
	TR	ST	TR	ST	TR	ST	TR	ST	TR	ST	TR	ST
While	73.4				24.7	1.3			0.1		0.4	
For	37.0						28.7	33.7	0.1		0.5	
Simple	88.8								10.4	0.6	0.2	
Stable Patterns												
Class	Biased		Alternating		While		For		Simple		Complex	
	TR	ST	TR	ST	TR	ST	TR	ST	TR	ST	TR	ST
Biased	32.9	48.2			0.5	2.5	5.1	10.7	0.1			
Alternating			75.4	24.4							0.2	
While	2.7				91.4	5.4					0.5	
For	1.1						67.7	30.7			0.5	
Simple	6.0								86.2	7.3	0.5	
Complex	4.1										91.6	4.3

Table 5.6: Loop patterns vs. PAs patterns

counters. The remaining counters are used only infrequently. The stable and potential pattern classes are more evenly used.

5.2.3 Classifying Using Limited vs. Complete History

We here show the difference between classification using limited per-address history and the classification from Section 5.1 using complete history information. Less history means that fewer patterns are classified as repeating patterns, with more patterns being classified as biased. For the experiment presented here, all patterns were first classified using the limited history, and then using the complete history.

Table 5.6 shows what complete patterns were seen for each class of limited patterns. The classes of limited patterns are shown in the leftmost column, and the classes of complete patterns are shown across the top. Each of the complete pattern classes are split into transient (TR) and stable (ST) classes as before.

Most transient and stable patterns were in the same class regardless of the classification scheme, although transient patterns often appear stable using a limited history. Some longer

complex patterns were misclassified as shorter transient patterns using limited history. This is one of the problems that a per-address predictor must contend with. Of the potential repeating patterns, many turned out to be transient biased patterns. Only 62% of the potential for-type and 26% of the potential while-type patterns were actually for-type or while-type patterns. The stable patterns were correctly classified most of the time. However, 19% of the stable biased patterns were actually parts of longer repeating patterns, mostly for-type. This means that the majority of for-type patterns are seen as stable biased patterns in a short history. A small number of stable repeating patterns were also misclassified due to the short history.

In addition to showing the differences between classifying a branch using a limited or complete history, this also shows the problem a per-address predictor has in determining what to predict. If we cannot tell the type of a pattern using the limited per-address history, the predictor is likely to have problems predicting the branch.

5.2.4 Effect of Lengthening PAs History

The obvious way of improving a per-address predictor is to increase the number of bits in the history. We here examine for which classes this helps and how this affects the utilization of the PHT.

Table 5.7 shows the effect on accuracy of adding history bits. The base is a 16-bit history, and all patterns are classified based on the last 16 history bits only, so a branch will appear in the same class for all history lengths. The prediction accuracy of each class is shown for a 16, 18, and 20-bit history. The history length is given by the number in parenthesis after the class name. The improvement in accuracy from increasing the history length for these patterns can be found by comparing the appropriate column (2-bc, 3-bc, or PSg) over these three rows. For instance, transient biased patterns are predicted with 81.2% accuracy using a 16-bit history if 2-bit counters are used in the PHT. If the history length is increased to 20 bits, these patterns are predicted with 85.4% accuracy.

The increase in accuracy from increasing the history length is large (3.7 to 6.1 percentage points) for all transient classes. The improvement is also substantial for stable simple and complex repeating patterns and potential repeating patterns. Other stable patterns show little improvement from increasing the history length. For the statically determined PHT predictions (PSg), there is little gain from increasing the history length.

Transient Patterns				
Class	Freq	2-bc	3-bc	PSg
Biased(16)	15.7	81.2	81.7	65.6
Biased(18)		83.8	84.0	65.8
Biased(20)		85.4	85.6	66.0
Alternating(16)	1.0	79.9	80.9	71.2
Alternating(18)		82.9	83.6	71.3
Alternating(20)		85.5	85.9	71.1
While(16)	0.4	86.4	86.7	67.1
While(18)		88.9	88.9	75.1
While(20)		90.3	90.2	75.1
For(16)	0.7	79.6	81.0	75.0
For(18)		81.4	82.6	76.3
For(20)		83.3	84.2	76.3
Simple(16)	0.1	78.2	78.6	61.5
Simple(18)		82.0	82.2	61.5
Simple(20)		84.3	84.4	61.5
Complex(16)	0.2	81.4	82.1	77.1
Complex(18)		83.7	84.4	77.1
Complex(20)		87.0	87.6	77.1
Potential Patterns				
Class	Freq	2-bc	3-bc	PSg
While(16)	2.8	90.9	91.6	88.5
While(18)		91.4	92.1	88.5
While(20)		91.9	92.6	88.9
For(16)	7.9	94.8	95.1	91.4
For(18)		95.0	95.3	91.7
For(20)		95.4	95.7	92.4
Simple(16)	2.1	84.7	85.7	79.6
Simple(18)		86.2	86.9	79.9
Simple(20)		87.4	88.0	79.8
Stable Patterns				
Class	Freq	2-bc	3-bc	PSg
Biased(16)	60.4	99.2	99.3	99.3
Biased(18)		99.2	99.3	99.3
Biased(20)		99.2	99.3	99.3
Alternating(16)	0.8	98.1	98.3	98.3
Alternating(18)		98.1	98.3	98.3
Alternating(20)		98.3	98.3	98.3
While(16)	0.6	96.3	96.5	96.2
While(18)		96.7	96.8	96.2
While(20)		96.9	97.1	96.2
For(16)	2.0	97.4	97.6	97.6
For(18)		97.6	97.7	97.6
For(20)		97.7	97.8	97.6
Simple(16)	0.1	89.6	90.6	87.9
Simple(18)		91.6	92.3	89.2
Simple(20)		92.6	92.9	89.2
Complex(16)	0.4	86.1	86.9	82.0
Complex(18)		87.7	88.3	82.0
Complex(20)		88.7	89.3	82.0

Table 5.7: Effect of adding history bits by pattern type

In Table 5.8, the PHT usage for the three history lengths is shown. As in the previous table, three rows representing three history lengths are shown for each class. However, unlike the previous table, branches are classified with the full history (16, 18, or 20 bits) available. As the history gets longer, more branches go in the transient biased class. For a 20-bit history, this class accounts for 17.2% of all branches compared to 15.7% for a 16-bit history. This is due to some of the potential repeating patterns now being classified as transient biased, and some of the stable biased patterns being classified as transient due to the longer history. There is also a small increase in the number of stable repeating patterns. These mostly come from the potential pattern classes.

Table 5.8 also shows the dramatic increase in the number of PHT-entries being used for the transient classes. For a 20-bit history, transient patterns account for 99.8% of the PHT-entries, but are only used to predict 20% of the patterns. As the history length increases, a smaller portion of the counters are used to frequently make predictions. Using a 20-bit history for the transient biased branches, 50% of the branches use only 1.5% of the PHT-entries. 90% of the branches use 24.0% of the entries. This shows that the PHT is being poorly utilized, especially for long histories.

Transient Patterns					
Class	Freq	Counters	50%	90%	99%
Biased(16)	15.7	60,432	5.7	52.3	90.0
Biased(18)	16.5	243,308	3.1	38.5	82.4
Biased(20)	17.2	975,392	1.5	24.0	67.4
Alternating(16)	1.0	2,042	5.3	44.8	86.9
Alternating(18)	1.0	8,168	2.7	33.4	78.1
Alternating(20)	1.0	32,662	1.3	20.4	63.4
While(16)	0.4	857	5.8	46.8	87.4
While(18)	0.4	3,482	4.4	35.9	80.3
While(20)	0.4	14,000	2.0	21.2	66.3
For(16)	0.7	857	6.2	40.5	84.0
For(18)	0.8	3,482	3.6	30.5	74.2
For(20)	0.9	14,000	1.9	19.6	59.9
Simple(16)	0.1	306	19.6	71.2	95.1
Simple(18)	0.1	1,348	11.3	57.1	90.4
Simple(20)	0.1	5,598	5.4	40.3	78.8
Complex(16)	0.2	188	7.4	49.5	87.8
Complex(18)	0.3	1,032	4.0	36.4	78.9
Complex(20)	0.3	4,844	2.0	23.6	66.0
Potential Patterns					
Class	Freq	Counters	50%	90%	99%
While(16)	2.8	57	22.8	52.6	94.7
While(18)	2.5	70	18.6	48.6	94.3
While(20)	2.2	84	15.5	48.8	94.0
For(16)	7.9	57	19.3	52.6	93.0
For(18)	7.5	70	17.1	52.9	92.9
For(20)	7.3	84	15.5	52.4	89.3
Simple(16)	2.1	540	8.3	47.0	88.3
Simple(18)	1.9	778	6.6	37.7	82.5
Simple(20)	1.8	1,076	5.3	31.3	76.8
Stable Patterns					
Class	Freq	Counters	50%	90%	99%
Biased(16)	60.4	2	50.0	100.0	100.0
Biased(18)	59.5	2	50.0	100.0	100.0
Biased(20)	58.8	2	50.0	100.0	100.0
Alternating(16)	0.8	2	50.0	100.0	100.0
Alternating(18)	0.8	2	50.0	100.0	100.0
Alternating(20)	0.8	2	50.0	100.0	100.0
While(16)	0.6	20	20.0	60.0	95.0
While(18)	0.8	26	19.2	53.8	92.3
While(20)	0.9	33	18.2	51.5	87.9
For(16)	2.0	20	20.0	70.0	100.0
For(18)	2.4	26	23.1	69.2	100.0
For(20)	2.7	33	24.2	66.7	93.9
Simple(16)	0.1	48	22.9	66.7	95.8
Simple(18)	0.1	78	16.7	65.4	93.6
Simple(20)	0.1	118	12.7	58.5	89.8
Complex(16)	0.4	108	13.9	43.5	87.0
Complex(18)	0.7	272	5.1	22.1	71.0
Complex(20)	0.7	648	2.3	18.1	65.3

Table 5.8: Frequency and utilization of PAs patterns by history length

Transient Patterns				
Class	Freq	3-bc	PSg	PSg-ideal
Biased	15.7	81.7	65.6	76.8
Alternating	1.0	80.9	71.2	76.7
While	0.4	86.7	67.1	82.2
For	0.7	81.0	75.0	78.0
Simple	0.1	78.6	61.5	71.7
Complex	0.2	82.1	77.1	80.0
Potential Patterns				
Class	Freq	3-bc	PSg	PSg-ideal
While	2.8	91.6	88.5	90.9
For	7.9	95.1	91.4	93.7
Simple	2.1	85.7	79.6	82.1
Stable Patterns				
Class	Freq	3-bc	PSg	PSg-ideal
Biased	60.4	99.3	99.3	99.3
Alternating	0.8	98.3	98.3	98.3
While	0.6	96.5	96.2	96.2
For	2.0	97.6	97.6	97.6
Simple	0.1	90.6	87.9	88.8
Complex	0.4	86.9	82.0	84.3

Table 5.9: The role of adaptivity in a PAs predictor

5.2.5 Role of Adaptivity in Per-Address Prediction

There are two major variations of two-level per-address predictors, adaptive and static. An adaptive predictor has a state mechanism, such as a saturating counter, for every entry in the PHT. In a static two-level predictor, the prediction for each pattern has been decided statically, either using an algorithm or using some form of profiling.

In Table 5.9, adaptive and non-adaptive (static) prediction is compared for each of the classes. One column shows the accuracy when using 3-bit saturating counters in the PHT. As the 3-bit counter was earlier shown to work better than the 2-bit counter, this will be used to represent adaptive two-level prediction. The next column shows the accuracy when the prediction for each pattern was chosen statically using the PSg(algo) [25] algorithm. The last column gives the accuracy of an ideal static two-level predictor. That is, for each PHT-entry, the prediction that worked best on average for all benchmarks was used. The ideal PSg is not attainable, but represents the best possible[12] static two-level predictor for these benchmarks.

For the stable biased, alternating, while-type, and for-type patterns the three predictors

[12] Assuming that the static predictions can not be changed between programs or on context switches.

perform almost equally well. The prediction for these patterns do not change over time, and two different branches having the same pattern are likely to have the same next outcome. In this case, a statically determined PHT is as good as an adaptive one.

For the stable simple and complex repeating patterns, the 3-bit counters in the PHT predict the branches more accurately than any possible static prediction. The 3-bit counter is about 2% better than the ideal PSg, and about 4% better than PSg(algo). For the potential repeating patterns, the situation is much the same. The PHT with 3-bit counters is a few percent more accurate than the ideal PSg, with PSg(algo) only a little further behind.

However, for the transient patterns, the situation is very different. In this case, the 3-bit counter achieves 6-20% higher accuracy than PSg(algo), and 2-7% higher than the ideal PSg. The transient patterns are inherently less stable than the stable patterns. Therefore, a pattern may be followed by different outcomes at different times. In addition, a pattern may always be followed by a not taken outcome for one branch, while always being followed by a taken outcome for another branch.

Adaptivity is important in two-level per-address predictors. This is mostly because of the transient patterns.

5.2.6 Summary

In this section, patterns were examined as they are seen by a PAs per-address predictor. It was shown that a substantial amount of for-type and while-type patterns with long periods are not properly recognized based on a limited history. Most other patterns are detected using a history length reasonable for a PAs predictor.

Furthermore, it was shown that most of the improvement from increasing the size of a PAs predictor comes from the increased accuracy on transient and potential patterns. Most of the PHT-entries, 98.7% for a 16-bit history, and 99.8% for a 20-bit history, are used by these transient patterns. However, these patterns only account for under 20% of all branches. Furthermore, a few of the most frequent transient patterns (each pattern corresponding to one PHT-entry) account for half of the transient pattern usage, and the proportion of the patterns used frequently drops as the history length increases.

The poor utilization of the PHT indicates that there is potential in either compressing or in some other way hashing a history so that the counters can be used more efficiently.

5.3 Methodology for Simulating Per-Address Predictors

One issue when simulating per-address two-level branch predictors is how to initialize the pattern history table. The normal methodology for all two-level branch predictors is to initialize all of the counters in the PHT to weakly taken. This is probably a reasonable starting state for global two-level predictors. However, for per-address two-level predictors, such as PAs, this is pessimistic and it over-estimates the training time of the predictor. The initial state should represent the likely state after running a different program. To better estimate a likely initial state, we initialize the PHT according to the predictions of a PSg(algo) predictor. The counters associated with patterns for which PSg(algo) predicts taken are initialized to weakly taken, the others to weakly not-taken. This gives a better estimate of what really happens when per-address predictors are used.

The algorithm used to determine the predictions for a PSg(algo) predictor is as follows. The history pattern corresponding to each PHT entry is examined to see whether it contains a repeating pattern. If it does, the prediction for that PHT entry is set to predict a continuation of that pattern. If no pattern repeats for the entire history length, consecutively smaller subsets of the history are considered. If no repeating history is found in a pattern of length 2, the prediction will be that of the most frequent outcome in the history.

5.4 Improving Prediction using Per-Address History

The most useful aspect of the information about self correlation and branch execution patterns that was studied in this chapter is how it relates to improving existing branch predictors. In this section, we look at how to improve two-level predictors, such as PAs, that use normal per-address histories. In Section 5.5, we look at other ways of exploiting branch execution patterns for prediction. In general, the most successful methods are those presented in Section 5.5.

We use two characteristics of the way branch execution patterns are predictable in a per-address two-level predictor. These two characteristics indicate how per-address based predictors can be improved, and were first examined in Section 5.2 (in particular in Section 5.2.4). To see how these characteristics can be used to improve prediction, we consider what happens when the history length in a per-address two-level predictor is increased.

The accuracy of a per-address two-level predictor, such as PAs, increases when the

history length is increased. This comes at the cost of extra storage required for the PHTs. Every time the history length is increased by one bit, the size of the PHTs is doubled. The way in which increasing the history length improves the prediction accuracy of a per-address predictor for each type of patterns was shown in Table 5.7. Also, the number of counters in the PHT used for each type of patterns and the utilization of these counters was shown in Table 5.8.

The first characteristic that we consider is how the accuracy of transient patterns increases as the history length increases. We see from Table 5.7 that the accuracy of transient patterns increases sharply when the history length is increased from 16 to 20 bits. This indicates that additional history is very useful for predicting transient patterns. However, when going from 16 to 20 bits of history, the number of PHT entries needed for the transient patterns increases from about 64,000 to over one million (see Table 5.8). This makes it very costly, in terms of predictor size, to improve the accuracy of these patterns by merely adding history bits. Table 5.8 also showed that the utilization of these counters is also very low. For a 20-bit history, about 25% of the counters were responsible for 90% of the transient patterns. So, merely adding history bits is not a very cost-effective way of increasing the accuracy of these patterns. Due to the skewed utilization of the counters in this category, it is likely that a better hashing scheme could take a longer history and hash it down to a shorter length, thus possibly increasing accuracy without increasing size.

The second characteristic that we consider is how the accuracy of potential patterns increases as the history length increases. This improvement is not as large as that for transient patterns, but is substantial. Additional history is therefore also useful for predicting potential patterns. The number of PHT entries needed for potential patterns only increases slightly from 654 at 16 bits to 1,244 at 20 bits. Because of the low number of PHT entries needed, adding history bits is a cost-effective way of increasing the accuracy for potential patterns.

From the perspective of transient patterns a better hashing scheme may improve prediction accuracy. From the perspective of potential patterns, simply increasing the length of the history appears to be a cost effective way to achieve improved prediction accuracies. These are conflicting goals, and indicate that it may be cost-effective to predict transient and potential patterns using different amounts of history.

Several hashing schemes or ways of selectively increasing the history length for some

patterns were examined. However, none of these schemes provided enough of an improvement to make them clearly worth the extra complexity. The two most interesting schemes are examined here to show the tradeoffs involved. Each of the schemes resulted in a small reduction in the misprediction rate compared to a normal PAs predictor.

5.4.1 Mixed Length History

As explained earlier, increasing the history length of a per-address two-level predictor is a cost-effective way of improving the accuracy of potential patterns, but not cost-effective for transient patterns. We therefore investigate a predictor for which the history length depends on the pattern. Patterns for which it is cost-effective to use a long history are predicted using a long history. Patterns for which it is cost-effective to use a shorter history are predicted using a shorter history.

A mixed length history allows some patterns to have longer histories than others, thus using more counters for these. A mixed length history uses a normal PHT, but is indexed using a special, and more expensive, decoder.

The easiest way to describe a mixed length history predictor is by example. Consider a predictor with 8 PHT entries, but a 4-bit history. In a normal predictor, a 4-bit history requires a 16-entry PHT. However, in the mixed length history predictor, some of the older history bits are ignored if the newer bits match certain patterns that were considered to be sufficient for prediction. A diagram of the mixed length history predictor is shown in Figure 5.11. As the figure shows, the pattern **1110** maps into the 4th PHT entry. As indicated by the **xx10** notation, all other patterns ending in **10** also map into that same entry. On the other hand, the patterns **0000** and **1000** are still distinguished even though they only differ in their oldest bit.

For our experiments, the choice of length for each pattern was made based on a combined profile of the entire SPECint95 suite running with separate profiling inputs. To further avoid specific advantages of designing the predictor expressly for these benchmarks, three additional benchmarks from the SPECint92 suite; espresso, eqntott, and sc; were included in the profile. The optimal mixed length history configuration for the profile was found. This configuration was then used when running the benchmarks with the regular data set. The misprediction rate of a 16 KB mixed length history predictor was 5.81% compared to 5.94% for a normal PAs. However, if the same profile and input set were used, the

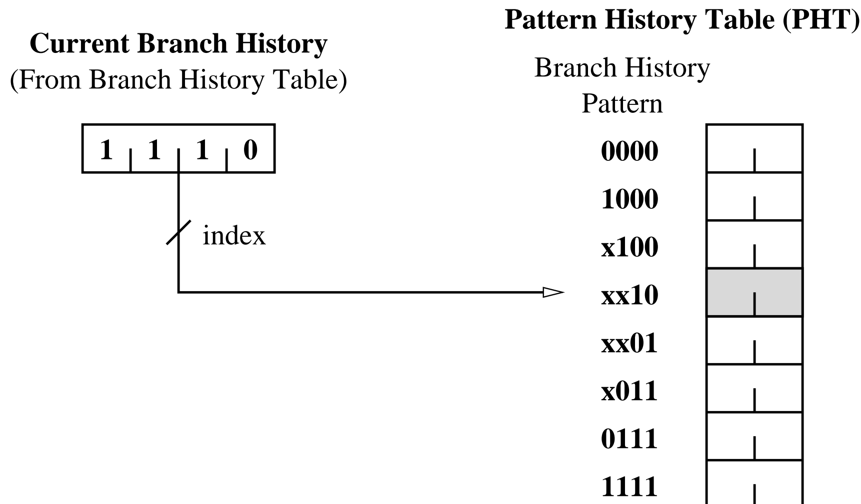


Figure 5.11: Diagram of per-address mixed length history predictor

misprediction rate dropped to 5.24%. This indicates that the pattern usage on a detailed level does not remain constant between programs and even for the same programs with different data sets.

The mixed length history predictor is able to exploit that some types of patterns (most notably the potential for, while, and simple repeating patterns) get more return for the extra counters used when adding history bits than other patterns (most notably transient patterns). However, this predictor is not able to exploit the large potential from increasing the history length of transient patterns, as this requires almost as many extra counters as just increasing the history length in a normal PAs predictor. Given the extra complexity of designing the decoder for this predictor, the small improvement probably does not warrant using this in an actual implementation.

5.4.2 Skewed Per-Address Prediction

We showed earlier that increasing the history length for transient patterns results in a large increase in prediction accuracy. Transient patterns use the majority of the counters in the PHT and the size of the PHT grows exponentially as the history length is increased. However, since the PHT is sparsely utilized for the transient patterns, it is likely that a longer history can be hashed down to a shorter history before the PHT is indexed without much loss of accuracy. We here suggest a scheme that hashes the history for all types of patterns: transient, potential, and stable. This is a slight variation on a previously proposed

scheme used for global two-level predictors.

Similar to the Gskewed and e-Gskewed predictors proposed by [21], one can construct a per-address skewed predictor. Extending the terminology used in [21], this would be called a Pskewed predictor. The concept is still the same: By storing redundant prediction information in three separate predictor banks, interference can be tolerated. Since interference is tolerated, more information can be hashed together to use in the indexing function. For the Gskewed predictor, $2 \times n$ bits of information (history or address) were hashed down into three separate indexes. The hashing functions were constructed such that two different (history,address) pairs never conflict in more than one bank.

A major difference between global and per-address prediction is that for global predictors, the history pattern only has meaning in the context of the branch it is used to predict. A per-address history has some intrinsic information about what the next prediction is likely to be. Consider the per-address pattern **11011101**. The next outcome is likely to be **1** regardless of which branch it is used to predict. Therefore, interference in per-address predictors is often neutral. However, with the hashing functions used in the Gskewed predictor, random patterns conflict with each other, and the interference is more likely to be destructive. In order to avoid this undesired effect, we only hash part of the index. The youngest history bits are used unchanged in the index to reduce the amount of negative interference in each of the banks. The hash functions used to generate the remaining part of the index are those presented in [21].

The best of the Gskewed predictors, e-Gskewed, uses two banks hashed with both history and address information and one bank which is hashed only with address, essentially the Two-Bit Counter predictor. However, we found that for the per-address version, using a PAs predictor as the third bank was the best option. This gives lower accuracy on BTB-misses, but higher accuracy otherwise. In a hybrid predictor, BTB-misses would likely be handled by a different component, so we use PAs as the third bank here.

For a 3×16 KB configuration, 13 history and 7 address bits are hashed together, using hash functions 1 and 2 from [21], to form 10 bits, which are concatenated with the 6 most recent history bits. The third bank was a PAs predictor with 16 history bits and a single PHT. The misprediction rate of this predictor, at a total size of 48 KB, was 5.22% compared to 5.39% for a 64 KB PAs predictor. For smaller sizes, only very marginal improvements were found.

For this Pskewed predictor, the hashing of the history resulted in an improvement for the sparsely used transient patterns. However, the hashing has a negative effect on the accuracy of the potential and stable patterns.

5.4.3 Other Approaches

The problem with the previous two approaches is that the first works well for potential patterns, while not taking advantage of the opportunities for transient patterns. The second works well for transient patterns, but is not as well suited to potential and stable patterns. Therefore, both approaches provided only small improvements over a normal PAs predictor given the extra complexity.

A number of other approaches were also tried, including several ways of hashing a larger number of history bits down to fit the index for the PHT. Even very complex hashing functions resulted in only marginal improvements. Profile based hashing functions were also tried, but when using different profiling and testing sets, the improvements were small given the complexity of the hashing schemes.

The problem with most of the hashing functions is how they perform for the transient patterns. Although much of the frequency of transient patterns is concentrated on a small number of patterns, it is hard to select a hashing function that consistently avoids collisions (interference) between patterns that should be predicted differently.

5.5 Improving Prediction by Filtering Repeating Patterns

In this section, another characteristic of self correlation is used to design a new predictor. In Section 5.1 and 5.2, we showed that repeating branch execution patterns are very frequent. We also showed that these patterns were often too long to be captured fully within a normal per-address history. They were often mispredicted by both PAs and Gshare predictors. However, they can be predicted very accurately by predicting a continuation of the repeating pattern. To improve the prediction accuracy for these frequent patterns, we introduce a new predictor, the loop predictor, that can predict alternating, for-type, and while-type repeating patterns. We also introduce a mechanism, loop filtering, which adds the functionality of a loop predictor to an existing predictor while also reducing the interference in that predictor.

One of our previous studies [3] looked at the Last-Time predictor, and showed that this predictor was very accurate for a subset of branches that are easily detected at runtime. We showed that because the Last-Time predictor was very accurate for these branches (essentially the biased patterns discussed earlier in this chapter), an additional benefit from this predictor could be created using a method we introduced and called Branch Filtering. With Branch Filtering, we detected the branches for which the Last-Time predictor is highly accurate. For these branches we used the Last-Time predictor, and disabled the main predictor. This resulted in increased prediction accuracy for the biased patterns that were predicted using the Last-Time predictor, and also resulted in reduced interference in the main predictor.

The loop predictor was designed explicitly to detect the alternating, for-type, and while-type repeating patterns found earlier in this chapter. Its accuracy on those patterns, and on long biased patterns is very high, and given the state contained in the loop predictor, it can easily be extended to detect whether the current pattern is one that it can predict accurately. The parallel to Branch Filtering is obvious. We therefore propose using the loop predictor we introduce in this section in a Loop Filtering mechanism much like a Last-Time predictor is used in a Branch Filtering mechanism. However, the Loop Filtering mechanism can filter out a larger subset of branches from the main predictor.

5.5.1 Implementation

A loop predictor is used to detect biased, for-type, while-type, and alternating patterns. In addition, a counter is added to that predictor to determine the stability of the pattern. As in a standard filtering mechanism, this is done by adding several fields to each BTB-entry. This new mechanism, the loop filtering mechanism, needs enough state to know what type of pattern the branch is following, and where in that pattern the branch currently is. This state also includes all the state needed to perform standard filtering, so the filtering mechanism is able to also filter out non-loop biased branches.

To implement the filtering mechanism, each BTB-entry contains the following fields. The number of bits for each field is given in parentheses:

- $Direction(1)$: 0 for while-type and 1 for for-type. This is the direction the branch takes for the non-exit part of the pattern. Initialized to the first outcome on a BTB-

miss.

- *Count*(7): How many consecutive times the non-exit outcome has been seen. (Number of times outcome of branch equal to *Direction*). Saturates at 126. Initialized to 1.

- *Last_Max* (7): Records the period (-1) of the last completed loop-pattern. Initialized to 127. If *Last_Max* is equal to 127, the pattern is stable biased.

- *Number_Repetitions*(4): How many times the pattern has repeated with the same period. Saturates at 15. Initialized to 0.

As for the history in a PAs predictor, the update to this mechanism should be done speculatively at predict time, and it must be repaired in the event of a misprediction. The filter is updated based on the outcome of the branch.

If the current outcome is the same as recorded in the *Direction* bit, add one to *Count*. No other action is taken.

If the current outcome is different from the outcome recorded in the *Direction* bit, we are at the exit of the pattern and the following rules apply:

- If *Count* is equal to *Last_Max*, increment *Number_Repetitions*. Otherwise *Number_Repetitions* is cleared.

- If *Count* is non-zero, *Last_Max* is set to the value in *Count* and *Count* is cleared.

- If *Count* is zero, this indicates that the pattern cannot be of the type indicated by *Direction*. *Last_Max* is cleared. *Count* is set to one. *Direction* is inverted.

The prediction of the loop filter is always given by the following formula (The C-like notation == is used to indicate a comparator):

Prediction = ((*Count* == *Last_Max*) AND NOT(*Count*==126)) XOR *Direction*

This equation means that if we are at the point where the pattern changed direction last time, we predict a change in direction. Otherwise, we predict that we will continue going in the current direction. Furthermore, if *Count* is saturated we will continue predicting the current direction. If calculating the prediction (one 7-bit comparator, one AND, and one XOR gate) is on the critical path, the prediction can be calculated at the time the loop filter is updated and stored in the BTB at the cost of one extra bit per BTB-entry.

Predictor	Loop		Stable Biased	Other
	Thres1 (R,P)	Thres2 R		
Gshare 4 KB	(2,4)	7	1	10
Gshare 16 KB	(2,4)	10	1	19
Gshare 64 KB	(2,8)	10	3	63
PAs 4 KB	(1,7)	5	1	7
PAs 16 KB	(1,7)	6	1	10
PAs 64 KB	(1,7)	7	1	10
PAs/Gshare 8 KB	(1,8)	15	1	65
PAs/Gshare 32 KB	(1,8)	15	1	65
PAs/Gshare 128 KB	(1,16)	15	5	70

Table 5.10: Thresholds for the loop filtering mechanism. The loop thresholds are represented as (R,P) where R is the number of repetitions and P is the period.

Although the loop filter will always generate a prediction, this prediction will only be used for branches for which the filter predictor is likely to be very accurate. For these branch instances, the filter predictor is used and the update to the conventional predictor is inhibited to reduce pattern history table interference. For our filtering mechanism, the decision on whether to use the filter to predict a branch depends on the previous behavior of that branch.

The decision of whether to engage the filter is made by examining the state for the branch in the BTB. Branches are divided into stable biased and other branches. For stable biased, the filter will be engaged if *Count* is larger than or equal to a threshold. This threshold depends on the predictor the loop filter is applied to, and is given in the "Stable Biased" column of Table 5.10. For other branches, the filter can be engaged in one of two ways. One is if *Count* is larger than or equal to a threshold and no direction change is predicted. This threshold is given in the "Other" column. Finally, the filter can be engaged for loop patterns for which we believe the loop predictor to be very accurate. These are identified if the period and number of times the pattern has repeated are larger than or equal to one of two sets of thresholds. The first set has one threshold for the period $(Last_Max + 1)$ and another for the number of repetitions. This threshold is given in the column "Thres1". The second threshold is for the number of repetitions only[13] and is given in the column

[13] The filter is engaged for all patterns that have repeated the same number or more times than the second

"Thres2".

As an example, for the 16 KB Gshare configuration, the filter will be engaged for all stable biased branches. For other branches if ($Count \geq 19$) and NOT($Count == Last_Max$). The filter will also be engaged if:

$((Number_Repetitions \geq 2) AND ((Last_Max + 1) \geq 4)) OR (Number_Repetitions \geq 10)$

The critical path of this is one magnitude comparator (although it can be simplified as we are comparing to a fixed value), one and gate, and one or gate. If this still remains on the critical path, the decision can be made when the counters are updated, and stored in the BTB at the cost of one bit per BTB-entry.

The threshold values in Table 5.10 are not guaranteed to be optimal. Due to the large number of potential threshold combinations, full simulation of all combinations was not possible. The thresholds were generated by a program that used data of the type presented in Figures 5.7 through 5.10. This program evaluated all possible combinations of thresholds to determine which would yield the best overall performance if all the prediction accuracy improvements and interference reductions were exactly as measured. However, due to the complex nature of branch interference, one can not accurately evaluate whether the destructive interference caused by a group of branches will disappear when those branches are removed from the predictor. The interference of other branches competing for the same pattern history table entries may be reclassified as destructive when a group of branches are removed. Although the thresholds given here can not be guaranteed to be optimal, different methods of choosing threshold values were tried and it is unlikely that great advances can be made.

It is not optimal to use the same thresholds for both PAs and Gshare in the PAs/Gshare hybrid. By using different thresholds for filtering predictions from these two predictors, prediction accuracy could be improved further. This improvement would come at a slight increase in complexity, and is not evaluated further.

5.5.2 Results

The improvement from adding loop filtering to a predictor is evaluated here. The base predictors that are compared against are Gshare and PAs predictors using 14, 16, or 18 bits to index the PHT, and PAs/Gshare hybrids using these predictors as components. The

threshold regardless of the period.

Gshare				PAs				PAs/Gshare			
size	norm	filter	loop	size	norm	filter	loop	size	norm	filter	loop
4	7.19	5.98	5.55	7.5	6.75	6.64	6.40	14	5.16	4.92	4.78
16	5.94	5.23	4.84	20	5.94	5.83	5.70	39	4.13	4.01	3.88
64	5.09	4.74	4.35	69	5.39	5.28	5.17	136	3.35	3.31	3.16

Table 5.11: Misprediction rate of Gshare, PAs, and PAs/Gshare. Unfiltered predictors, as well as predictors using a regular and a loop filter are shown. Sizes are in KB

configurations used for each of the single scheme predictors were the optimal ones identified in Table 4.1. For the PAs/Gshare hybrid, each of the components had a single PHT. Finally, the filter and loop filter were both added to predictors with only a single PHT as this gave the best results.

The performance of all the predictors, including versions using a normal filter or a loop filter are given in Table 5.11 for the three different sizes. Note that the PAs predictor is larger than a Gshare predictor with similar size PHTs. This is because there is an additional cost for the history information used by the PAs predictor. There are four columns for each of the predictors. The first column, "size", gives the size of the unfiltered configuration. There is an additional cost of 1-1.5 KB for the filtered configuration. For the loop filtered configuration, the additional cost is 5 KB. The second column, "norm", gives the misprediction rate of the unfiltered configuration. The third configuration, "filter", gives the misprediction rate of the predictor using a standard non-loop filter. The final column, "loop", gives the misprediction rate of the predictor using a loop filter.

As is shown in Table 5.11, the loop filter reduces the number of mispredictions from Gshare by 15-23% depending on the size, and improves by 7-8% over using a normal filter. The improvement over PAs is 4-5%, and 2-4% over the filtered PAs. More encouraging than the PAs results, the improvement over PAs/Gshare is 6-7% with a 3-5% improvement over a filtered PAs/Gshare. Although the improvement over PAs/Gshare, which can be considered to be a truly state of the art predictor, is only 6-7%, there is little additional complexity in adding the loop filter to that predictor.

The filtered and loop filtered configurations are slightly larger than the corresponding unfiltered predictions. In Figure 5.12, the misprediction rates of Gshare predictors with

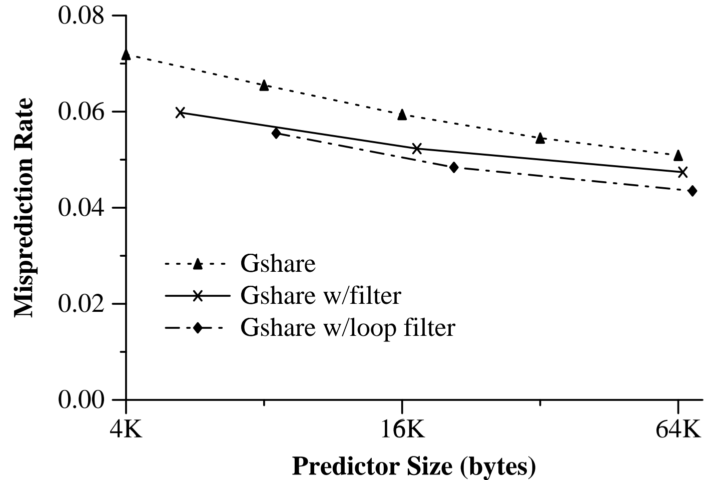


Figure 5.12: Loop filter performance on Gshare

and without the two filtering mechanisms are graphed according to size. The improvement in misprediction rate of the loop filter over the standard filter is almost constant for the three sizes. However, for small predictors, the additional hardware required to implement the loop filter could provide fairly substantial improvements to the base predictor instead. For larger predictors, the 5 KB used by the loop predictor are less important. It is therefore not until sizes of 16 KB or more that the loop filter is really attractive. For benchmarks, such as gcc and go, that have a large footprint, the improvement from using a loop filter is larger.

5.6 Summary

In this chapter, we examined the execution patterns of branches, and how the outcome of a branch is correlated to its previous outcomes.

One of the findings was the abundance of branches that can be confidently predicted using per-address information alone, and that these branches can easily be identified dynamically. Stable biased, and high confidence repeating patterns that account for 49% of all branches are over 99.9% predictable. In addition, transient biased branches that have repeated the same outcome 30 or more times account for another 17% of all branches and are over 99% predictable. The idea of using a separate predictor for these branches, so that other prediction resources can be used for hard-to-predict branches is appealing.

Per-address pattern use was also examined in terms of the limited history seen by a regular per-address predictor, such as PAs. It was seen that patterns that were transient

in the limited history accounted for less than 20% of all branches, but these patterns use 99% of the counters in a 16 KB PAs predictor and 99.9% of the counters in a 256 KB PAs predictor. Even within the transient patterns, 5% of the patterns account for half the branches. This means that most of the pattern history table of a per-address predictor is used very infrequently. However, in the studies in this chapter, only marginal improvements were seen from techniques to improve the usage of the pattern history table.

It was also shown that the improvement from increasing the history length of a PAs predictor was mostly due to the improved prediction of transient patterns. A smaller improvement was due to the improved prediction of repeating patterns too short to fit in the history. The transient patterns are also the patterns that require adaptive prediction (i.e. 2-bit counters) in the PHT. Some stable patterns are slightly better predicted using static values rather than 2-bit counters in the PHT, as in the PSg(algo) scheme. As a result, PAs can be improved slightly by selectively tying some of the PHT entries to static predictions.

Finally, an enhanced filtering mechanism, the loop filter, was introduced in this section. High confidence for-type, while-type, alternating and strongly biased patterns were predicted using the loop predictor, while other branches were predicted using the main predictor. Using this mechanism, the number of mispredictions suffered by the Gshare predictor was reduced by 15-23% depending on the size, with a 7-8% improvement over a standard filtering mechanism. The number of mispredictions suffered by a state of the art PAs/Gshare hybrid was reduced by 6-7%, with a 3-5% improvement over the standard filtering mechanism. The improvements over the standard filter increased for sizes over 16 KB.

The key contribution of this chapter is that it quantifies the type of execution patterns branches follow and gives a better understanding of how branches behave. For improving future predictors, we should explore not only the improvement that can be had by predicting repeating patterns with a separate predictor, but rather how we can construct other predictors differently knowing that two thirds of the branches have already been taken care of.

CHAPTER 6

Branch Correlation

In the previous chapter, branch behavior was examined in terms of a single branch at a time. Only past outcomes of the branch itself were used to determine how the branch behaved. In this chapter, the correlation between the outcomes of different branches is examined. If two branches are correlated, knowing the outcome of the first branch indicates which direction the second branch is likely to take. This correlation between branches is the effect that global two-level branch predictors, such as GAs and Gshare, use to predict branches.

First, branch correlation is described, and its usefulness for branch prediction is explained. The manner in which global two-level predictors use correlation to make a prediction is also shown. Then the nature of correlation between branches is explored through a sequence of experiments to give a better understanding of the correlation that is present in the benchmarks, and to what extent current predictors are taking advantage of this correlation. Finally, new branch correlation based prediction schemes are proposed and evaluated.

6.1 What is Branch Correlation

Two branches are said to be correlated if there is a reciprocal relation between their outcomes. That is, if knowing the outcome of one branch provides information about the likely outcome of the other branch, the two branches are correlated. The correlation may be strong, such that the outcome of the first branch can be confidently used to predict the second branch, or the correlation may be weak, such that the outcome of the first branch

only hints at what the second branch will do. Although strong correlation is more effective for prediction, weak correlation can also be useful for hard-to-predict branches.

Four examples of how branches can be correlated are examined here. The examples are written in pseudo-code for clarity, showing "if-statements" rather than the branches they translate into. In all of the examples, the final branch (branch X) is the one that is being predicted. This will be referred to as the *current* branch. The preceding branches whose outcomes are correlated with the current branch will be referred to as the *correlated* branches.

There are two reasons for the directions of two branches to be correlated. One is that the conditions of the two branches are based (fully or partly) on the same or related information. An example of this is shown in Figure 6.1. The other reason is that information affecting the outcome of the current branch is generated based on the outcome of the first branch. An example of this is shown in Figure 6.2. Since the outcome of the second branch is correlated to the direction of the first branch, we will refer to these two types of correlation as *direction* correlation.

```
branch Y: if (cond1)
...
branch X: if (cond1 AND cond2)
```

Figure 6.1: Correlation example 1

```
branch Y: if (cond1) a = 2;
...
branch X: if (a == 0)
```

Figure 6.2: Correlation example 2

```
branch Y: if (NOT(cond1)) ...
branch Z: else if (NOT(cond2)) ...
branch V: else if (cond3) ...
...
branch X: if (cond1 AND cond2)
```

Figure 6.3: Correlation example 3

Frequently, the outcome of the current branch is also correlated with the existence of certain branches in the recent history. That is, knowing whether a particular branch appeared in the recent history can help predict the current branch. This is illustrated by Figure 6.3. In this case, if branch V is in the recent history, we know the conditions for branch Y and Z were false, so cond1 and cond2 are both true. Note that the direction of branch V is not related to the direction of branch X, but from knowing that branch V was on the path to branch X we know that the condition of branch X will be true. If a branch is in the recent history leading up to the current branch, we will say that it was in the path to the current branch. We will refer to the correlation between a branch being in the path and the outcome of the current branch as *in-path* correlation. In example 3, the in-path correlation between branch V and X resulted from branch V only being reached through certain outcomes of branch Y and Z, each of which is direction correlated with branch X. However, in-path correlation can also work more directly. For example, the outcomes of branches in the beginning of a subroutine may be correlated with where the subroutine was called from. In-path correlation can also account for this.

```
branch Y: if (cond1)
...
branch Z: if (cond2)
...
branch X: if (cond1 AND cond2)
```

Figure 6.4: Correlation example 4

In many cases, the correlation between one pair of branches, such as the pairs shown in Figure 6.1 and 6.2, is not strong enough to confidently determine the direction of the second branch. Sometimes the correlation is strong only if the correlated branch is taken (or not taken) or the correlation may be weak altogether. In these cases, we need to look at the correlation between several branches and the current branch. In the example shown in Figure 6.4, branch X will be taken if both Y and Z are taken. Branch X will not be taken if either Y or Z is not taken. We do not know whether X will be taken if neither Y nor Z is in the path. Later in this section, we investigate how the predictability of branches increases as a function of the number of correlated branches that are used to determine the prediction.

6.2 Correlation in Two-Level Branch Predictors

Of the branches in the path leading up to the current branch, there will be some that are correlated and some that are not. Ideally, we want to build a history including the outcomes of the branches that are correlated, but excluding the outcomes of branches that are not correlated.

Global two-level branch predictors, such as the GAs predictor shown in Figure 2.1 in Chapter 2, are able to exploit correlation by basing the prediction on the outcomes of all the recently executed branches. The most recent branch outcomes are recorded in the first level of history, and the second level of history records the most likely outcome when a particular pattern in the first level history is encountered.

For each of the branches in the history, if that branch is taken, it will generate a different pattern than if that branch is not taken. These patterns will then use different entries in the second level history table allowing different predictions to be made depending on the outcome of that branch.

Not all of the outcomes in the branch history register contain information that is useful for prediction. That is, some of the branches that are recorded in the history are not correlated to the current branch. Different outcomes of these branches will still cause different patterns to be used, but with no beneficial effect on prediction accuracy. However, the added noise, resulting in more interference and longer training times, may have a negative effect.

Global two-level predictors work because of correlation. However, they exploit correlation on a pattern basis rather than based on the correlation with individual branches. Though fairly effective, this can be considered a brute force method.

6.3 Correlation Using a Selective History

To investigate how many of the entries in the history are really needed, we defined a hypothetical predictor. This predictor works in a manner similar to a global 2-level predictor, but only the outcomes of the 1, 2, or 3 most important correlated branches are included in the history. For this *selective* history, we used an oracle to choose the set of 1, 2, or 3 most important branches to include in the history for each branch. The approach used to find the most important branches for the history is explained in Section 6.4.3.

The outcome of each of the 1-3 correlated branches is recorded in the history as taken, not taken or not in the recent path of branches leading up to the current branch. The "not in path" outcome was required in this hypothetical predictor as we are looking at 1, 2, or 3 particular branches, and not all of these appear in the recent path all the time.

The history with 1 branch can have 3 possible patterns (taken, not taken, or not in path), the history with 2 branches can have 3^2 patterns, and the history with 3 branches can have 3^3 patterns. Predicting using these history patterns is then done identically to predicting in a global 2-level predictor. The pattern is used to select a counter in the second level table. The most significant bit of this counter provides the prediction, and the counter is updated with the outcome of the branch.

6.4 Identifying Correlation in a Simulator

Section 6.5 describes several experiments showing the nature of correlation using the Selective History predictor. This section explains how the correlation is detected in the

```
int correlated_branch_addr;  /* Address of correlated branch */
int times_curr_taken[3];     /* Times current branch taken for
                                each outcome of correlated branch */
int times_curr_not_taken[3]; /* Times current branch not taken for
                                each outcome of correlated branch */
char tbc[3];  /* A two bit counter for each outcome */
int tbc_corr; /* Number of times two bit counter correct */
```

Figure 6.5: Fields in record for each correlated branch

simulator. First the general case, assuming that there are no loops in the code, is discussed. Then, the effect of loops and thus having the same branch appear multiple times in the history is considered. Finally, the oracle used in the Selective History predictor is explained.

6.4.1 General Case

In the general case, we need to consider the correlation between the current branch and each of the branches that ever appear in the recent history leading to the current branch. For each pair consisting of the current branch and each of the possible correlated branches [1], the information specified in Figure 6.5 is collected. This record contains the number of times the current branch was taken or not taken for each outcome of the correlated branch, and is used to calculate the correlation coefficients between the current and correlated branch.[2] In addition, as a different measure of the correlation between the branches, three two-bit counters are used to predict the outcome of the current branch depending on the outcome of the correlated branch as in the Selective History predictor explained above. If these counters predict the branch better than a single two-bit counter, this also indicates that the two branches are correlated.

At the end of the run, we have two different measures of the correlation between each branch and each of its possible correlated branches. One measure is the correlation coefficients whose magnitudes show the actual amount of correlation. The other measure is the difference between the accuracy of the three two-bit counters and the accuracy of a

[1] For reasons of memory availability, only up to 384 correlated branches were considered for each branch. However, this limit was rarely reached.

[2] In fact, two correlation coefficients are calculated. One for the correlation between the directions of the branches and one for the correlation between whether the correlated branch was in the path and the direction of the current branch.

single two-bit counter for the branch. This second measure shows how much the correlation between the current and correlated branches improves the prediction accuracy.

6.4.2 Complexities Introduced by Loops

The general case just discussed used the assumption that the address of a branch is sufficient to distinguish branches for correlation purposes. However, in tight loops, several iterations can sometimes fit in the recent history of branches that we are examining, so we must be able to distinguish between multiple instances of the same branch.

There are two straightforward methods to distinguish between multiple instances of a previous branch. For each of these, the branch will be identified by its address along with a "tag" that represents a particular dynamic instance of that static branch. The correlation data is then collected as just described, with the exception that data is collected separately for branch instances with different tags.

One of these methods is to number the instances of a branch starting at the current branch. So, if branch A appears in the history 3 times, the most recent occurrence would be A_0, the second most recent would be A_1 and the oldest would be A_2. However, with this method there is no way to clearly identify branch A from a specific iteration of a loop if it does not appear in every iteration.

The other method is to number the instance of a branch by how many backward branches have occurred between it and the current branch. This enables us to clearly identify the instance of the branch from a certain number of iterations ago. However, with this method you can not easily identify branches that occurred before the current loop. A branch that occurred before the beginning of the loop will be tagged differently depending on how many iterations of the loop have passed.[3]

Each of these methods of identifying the instance of a branch have different limitations. Therefore, all branches were tagged using both methods. Branches tagged using the two different methods were considered to be distinct instances when investigating correlation with the current branch.

[3]This could be fixed by only counting the number of backward branches that branched past the branch in question, but due to subroutine calls, this is difficult to determine.

6.4.3 How the Best Set of Correlated Branches is Detected

For the Selective History predictor described in Section 6.3, an oracle is needed to choose the set of 1, 2, or 3 most important branches to include in the history for each branch. One way to do this is to try all possible sets of 3 (or 1 or 2) correlated branches and then select the best set after the fact. However, this approach requires a very large amount of memory. Initially, the maximum number of correlated branches considered for each current branch was set to 384. This solves the memory problem for the 1 branch selective history, but still leaves a maximum of 384 *choose* 3 $\approx$ 9 *million* combinations of 3 correlated branches for each static branch. Both the memory and processing time needed to evaluate all these possibilities make this option impractical.

To make the oracle work, a first pass through the program identifies the branches that have the strongest correlation with the current branch. For each static branch, the 30 candidates with the largest correlation coefficients were chosen. In the second pass, the 3 (or 2) branch selective history was simulated for each set of 3 (or 2) out of these 30 branches. At the end, the oracle chose the set of 3 correlated branches that gave the highest prediction accuracy. The Selective History predictor identifies the strength of the correlation with up to 3 branches, but does not show how the most correlated branches can be found.

6.5 Understanding the Nature of Correlation

In this section, the nature of correlation is studied in terms of the amount of correlation present in programs, how many correlated branches are needed to predict a branch, and where these correlated branches are. As the Selective History predictor directly exploits the correlation in a program, this predictor is used to measure the amount and nature of the correlation. We show that some of this correlation is not exploited by the Gshare predictor.

We investigate reintroducing information that has been replaced in the history due to subroutine calls or loop bodies, and show that there is significant additional correlation with this history information. We investigate correlation between branches and the location of previous subroutine calls. The dependence of correlation on the input set to the benchmark is also investigated. Finally, we quantify the number of branches that show the best correlation with each type of history information.

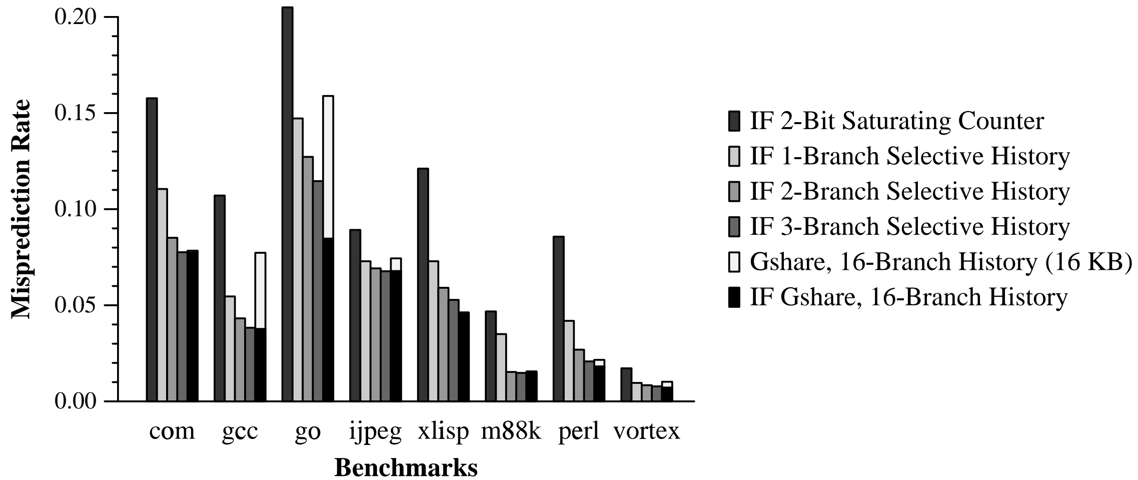


Figure 6.6: Misprediction rates of Two-Bit Counter, Selective History, and Gshare predictors

6.5.1 How Many Correlated Branches are Needed

The amount and type of information that is needed for prediction varies from branch to branch. In Chapter 5, we investigated predicting branches using self correlation, that is, correlation based on the branch's past outcomes. Other branches are predictable based on branch, or global, correlation, which is the correlation investigated here. A branch can be correlated with one or more branches in the recent history. However, some branches are not correlated with any previous branches, either because they are highly biased, or because they are just not predictable using correlation. In this section we identify how many previous branches a branch is correlated to.

Figure 6.6 shows the prediction accuracy for each benchmark using a selective history of only the 1, 2, or 3 most important branches. This is compared to the prediction accuracy of an interference free Gshare predictor using a 16 branch history, and an interference free table of 2-bit saturating counters. In the legend of Figure 6.6, "IF" refers to interference free. The accuracy of a regular Gshare predictor is shown for reference.[4] The significance of the bars is that the Two-Bit Counter predictor does not exploit any correlation. The Selective History predictor exploits only the correlation with 1-3 previous branches. The Gshare predictor can exploit the correlation with all of the most recent 16 branches, but must also deal with the noise introduced by uncorrelated branches.

[4]For compress, xlisp and m88ksim the accuracy of Gshare is almost identical to that of interference free Gshare, so the Gshare bar is not visible.

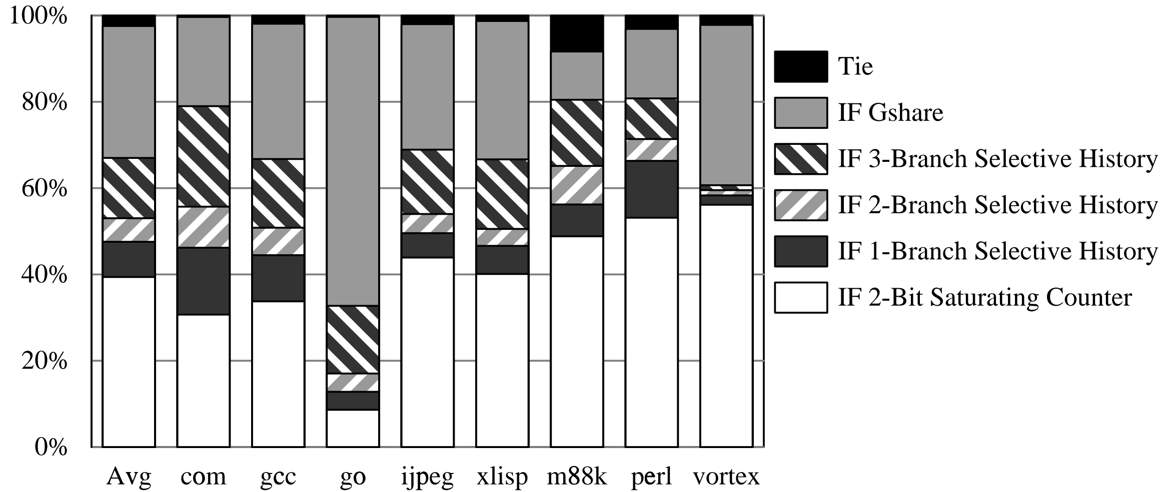


Figure 6.7: Weighted distribution of branches by best predictor

Most of the benchmarks in Figure 6.6 display the same general behavior. Even with a selective history containing only one branch, meaning that only correlation with that one branch could be exploited, the misprediction rate is much lower than for the Two-Bit Counter predictor. Only for the go benchmark, and to some extent xlisp, is the interference free Gshare predictor substantially better than the 3-branch Selective History predictor. For all but xlisp, the 3-branch Selective History predictor is better than Gshare. It is an important point that the 3-branch Selective History performs as well as an interference-free Gshare for most benchmarks. The interference-free Gshare predictor is using the outcomes of all of the 16 most recent branches to make its prediction. However, this does not result in much better prediction accuracy. Using all 16 outcomes when only a few are needed introduces undesired noise. This noise impacts a Gshare predictor in two ways. One is added interference (obviously not a factor in the interference free Gshare). The other is increased training time. Finally, whereas an interference free Gshare predictor requires a prohibitive amount of storage for the pattern history tables, the interference free 3-branch Selective History predictor requires a much smaller amount of storage.

The previous figure showed only the average misprediction rates for each predictor. It is revealing to also examine for how many branches each of the predictors performed best. Figure 6.7 shows the weighted distribution of branches according to which predictor has the lowest misprediction rate. If the Two-Bit Counter predictor and the Selective History predictor were equally accurate, the simpler predictor (Two-Bit Counter) was considered

the winner. For a tie between different size selective histories, the shorter history was considered the winner. A tie between interference free Gshare and any of the Selective History predictors was recorded in the tie category.

The Two-Bit Counter predictor is at least as good as the others for 39% of the dynamic branches. These are mostly strongly biased easily predictable branches, but also a small number of branches for which correlation is not useful. The Selective History predictor is best for 28% of the branches. For these branches, correlation with just a few branches is sufficient, and the Selective History predictor avoids noise from unneeded branches. There are significant regions each where 1, 2, and 3 correlated branches are needed. The interference free Gshare predictor is best for 31% of the branches. For these branches, the correlation with three branches is not enough, either because the correlation is complex, or because there are too many possible paths to the branch to get meaningful information out of only three branches. Ties account for the remaining 2%. Two benchmarks, go and vortex, behave differently from the others. Go has a smaller region of branches for which the Two-Bit Counter predictor is best, with a corresponding increase in the number of branches for which the interference free Gshare predictor is best. For vortex, the Selective History predictor is rarely successful, indicating that simple correlation of the kind exploited by this predictor is rare in vortex.

The number of branches for which each predictor is best still does not tell the full story. To put this information in perspective, one must also know how much more accurate each predictor is in the region where it is best. In Table 6.1 we compare the misprediction rate of the 3-branch selective history to the misprediction rates of the interference free Gshare for each of these regions. The top row identifies which predictor is best in that region. "1 br SH" means 1-branch selective history and so on. The second row identifies the predictor whose misprediction rates are listed in the column. There is one row for each of the benchmarks. As an example, in the "IF 3 br SH" region, for the go benchmark, the 3-branch selective history had a misprediction rate of 6.34% while the interference free Gshare predictor mispredicted 8.34% of the time. These are the misprediction rates of these predictors for the region shown as "IF 3-Branch Selective History" on the bar marked "go" in Figure 6.7. The figure and the table should be considered together, and this combination of figure and table will be used several places in this chapter. The purpose of this table is twofold. First, we show that even though the average misprediction rates of the Selective

	IF 2bc		IF 1 br SH		IF 2 br SH		IF 3 br SH		IF Gshare	
	3 br	gsh	3 br	gsh	3 br	gsh	3 br	gsh	3 br	gsh
	(%)	(%)	(%)	(%)	(%)	(%)	(%)	(%)	(%)	(%)
com	0.04	0.06	1.17	1.33	10.80	12.18	22.50	23.68	6.27	4.14
gcc	0.72	1.43	1.10	2.19	2.74	4.03	5.78	7.09	7.42	5.15
go	0.69	4.29	2.28	4.74	3.67	5.92	6.34	8.34	15.14	9.45
ijp	1.38	1.46	3.21	4.70	0.80	0.88	26.53	27.10	6.63	5.94
xli	0.12	0.14	5.53	5.56	8.46	8.53	11.82	12.43	8.19	5.49
m88k	0.87	0.89	3.38	3.77	0.96	1.43	1.48	1.71	2.20	1.16
per	0.27	0.34	0.35	0.63	2.81	4.64	5.25	7.19	7.44	3.70
vor	0.28	0.33	1.67	2.13	1.43	2.28	8.83	12.45	1.25	0.85

Table 6.1: Misprediction rates for 3-branch Selective History and interference free Gshare predictors for each region

History and interference free Gshare predictors are close on average, each is significantly better for those regions where it is better. Second, by comparing the misprediction rates of the two predictors in the regions where Gshare does worse, we can see the amount of correlation that is detected that Gshare does not exploit.

Referring to Table 6.1 we see that for many regions, particularly the 1-3 branch SH regions, the Selective History predictor is significantly better than interference free Gshare. This is especially true for gcc and go. In these benchmarks there are many static branches, with each being executed only a limited number of times. The warmup effects using a full 16-branch history are therefore especially important. However, especially for go, but also for perl and gcc, interference free Gshare is also much better than the Selective History in the region where interference free Gshare is best. This indicates that there is also correlation that is too complex to be detected using only three correlated branches.

To summarize the most important results of this section: a Selective History predictor using only 3 branches in its history can predict as accurately as an interference free Gshare predictor for most benchmarks. There is a large set of branches for which correlation with only a few prior branches was needed in order to predict better than an interference free Gshare predictor. For this set, the Selective History predictor is substantially better than interference free Gshare, indicating that the noise from unimportant branches in the history severely impairs the ability of Gshare to make accurate predictions.

	Gshare					IF Gshare				
	2bc	1 br	2 br	3 br	gsh	2bc	1 br	2 br	3 br	IF gsh
	(%)	(%)	(%)	(%)	(%)	(%)	(%)	(%)	(%)	(%)
com	7.74	7.53	7.35	7.33	7.84	7.72	7.52	7.34	7.32	7.75
gcc	5.03	4.14	3.79	3.60	7.73	3.43	3.27	3.19	3.12	3.77
go	12.89	11.65	10.92	10.33	15.89	8.01	7.86	7.74	7.65	8.47
ijp	6.96	6.84	6.74	6.68	7.44	6.71	6.65	6.59	6.57	6.78
xli	4.53	4.50	4.47	4.42	4.63	4.52	4.50	4.46	4.41	4.53
m88k	1.50	1.42	1.39	1.38	1.56	1.47	1.40	1.38	1.37	1.49
per	1.84	1.68	1.58	1.51	2.16	1.71	1.63	1.54	1.47	1.82
vor	0.74	0.70	0.68	0.66	1.02	0.69	0.66	0.65	0.63	0.72
AVG	5.15	4.81	4.62	4.49	6.03	4.28	4.19	4.11	4.07	4.42

Table 6.2: Potential for improving Gshare and interference free Gshare with selective history

6.5.2 Potential for Improving Gshare with Correlation

In order to examine the potential for improving Gshare and interference free Gshare by more directly exploiting correlation and reducing warm-up effects, we constructed a hypothetical predictor. For this hypothetical predictor, we combined Gshare with either a Two-Bit Counter predictor or a 1, 2, or 3-branch Selective History predictor. For each static branch, we used the best of the two components. This is not an upper bound for the performance of a predictor using both components, as the best component might change during the run of a program.

Table 6.2 shows the misprediction rates for each of these hypothetical predictors along with the misprediction rates of normal Gshare and interference free Gshare predictors. The top row shows whether Gshare or interference free Gshare is used in the hypothetical predictor. The second row shows which predictor Gshare or interference free Gshare is combined with. The columns marked "gsh" and "IF gsh" are the misprediction rates of the Gshare and interference free Gshare predictors respectively. When combined with the 3-branch Selective History predictor, the reduction in average misprediction rate is 26% over Gshare and 8% over interference free Gshare. As expected, the improvements are more pronounced for the gcc and go benchmarks, where the reductions are 53% and 35% over Gshare and 17% and 10% over interference free Gshare. This shows the extent to which Gshare sometimes fails to capture the available correlation.

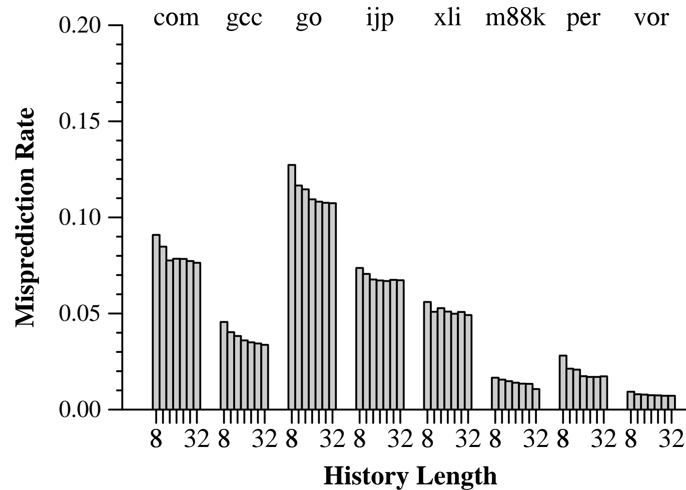


Figure 6.8: Misprediction rate of Selective History predictor vs. history length

6.5.3 Distance to Correlated Branches

We showed in Section 6.5.1 that for a large number of branches, only a small selective history is needed to make an accurate prediction. In that experiment, the 16 most recent branches were considered in forming the selective history. In this experiment, we examine how far back the important branches are by considering the n most recent branches, where n is varied from 8 to 32. We will refer to n as the history length. Clearly, the closer the most important branches are to the current branch, the easier it is to exploit the correlation in a predictor implementation. Figure 6.8 shows the misprediction rate using a selective history of 3 branches for history lengths going from 8 to 32 branches in intervals of 4. In general, history lengths of less than 12 were found to be limiting. There is a slow but steady growth from 12 up to a history length of 20, but little gain in looking farther back. This indicates that the most correlated branches are often close to the current branch.[5]

In a few cases, such as for xlisp at a history length of 12, increasing the history length actually reduced the amount of correlation found. This is due to the in path correlation. With in path correlation, a branch can be correlated with whether a certain prior branch appeared in the recent history. However, a branch that is not in a history of 12 branches, may be in a history of 16 branches, and thus a subtle advantage of in path correlation may go away. Whether the branch was in the recent history or not may depend on several

[5]This conclusion holds in the context of a 3-branch selective history. It is possible that having a larger selective history could increase the benefit of longer histories.

control flow decisions between the correlated branch and the current branch. Such decisions may be the number of iterations of a short loop or whether a subroutine was executed. In some cases, this may add useful information for the prediction, so this correlation may be lost when the history length is increased.

6.5.4 Correlation Across Subroutine Calls

Correlated branches are often close to the current branch in the source code of a program. These branches will frequently also show up as the most recent branches in the history. However, sometimes control flow structures such as subroutine calls and loops come between the current branch and the correlated branches. Often a subroutine will contain enough branches to completely fill the branch history and remove any correlated branches prior to the subroutine from the history.

In order to preserve the history from before a call, we use a Return History Stack[6] similar to that used in [14]. When a call instruction is encountered, the global history is pushed on a stack without changing the current value of the history register. When a return instruction is encountered, the history from before the function call is popped off the stack.

To evaluate the amount of correlation across subroutine calls, we compare three versions of the Selective History predictor. The first, "Normal", is the normal version described in Section 6.3. The second, "Sub Only", preserves its history across calls using a Return History Stack as just described. The third, "Sub + Normal", has both a normal history and the preserved history available, and can therefore detect correlation where one or more correlated branches are inside the subroutine, while others are before. Given the oracle used for selecting the most correlated branches in the Selective History predictor, the third version will always be at least as good as the two others, but it has the advantage of being able to use more history information.

[6] For these experiments, a 256-entry Return History Stack was used.

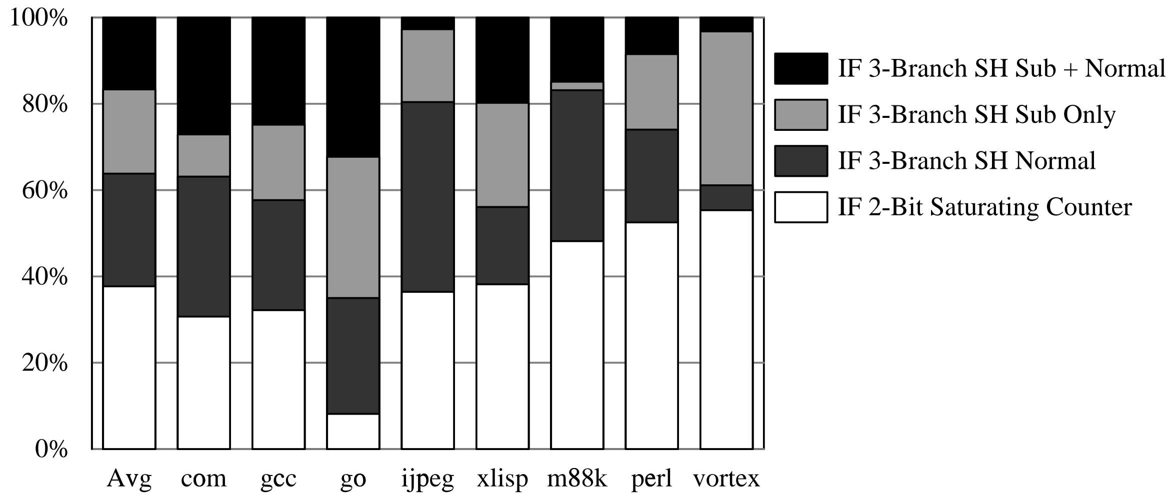


Figure 6.9: Frequency of branches with correlation across subroutine calls

Figure 6.9 shows the weighted distribution of branches according to which of these predictors has the lowest misprediction rate. The interference free Two-Bit Counter predictor is also included. On average, 36% of all branches showed some improvement from having history from before the subroutine call available. These are branches that are correlated with branches from before the subroutine call where the correlated branches have been flushed from the history by the branches in the subroutine. 19% of the branches needed *only* history from before the subroutine call, while 17% used correlation with branches both in and before the subroutine. For go, 65% of the branches benefit from history information from before the subroutine. This is because go has a large number of subroutine calls. Many of these calls are unrelated to the control flow decisions that are made locally. For ijpeg and m88k, fewer than 20% of the branches see benefit from this history information. If comparing only the "Normal" and "Sub Only" configurations, 26% of the branches are predicted most accurately using only the information from before the call.

There is a particularly large number of branches in the go benchmark where history from before the call is useful. One section of code where this is the case is shown in Figure 6.10. This piece of code is from the getefflibs function. Notice that the first part of the conditions for the first and third if-statements are identical conditions. Due to the nature of the "&&" operator in c, there is one branch corresponding to each of the three tests in each of the if-statements. The branch corresponding to the first test in the first if-statement therefore has a condition that is identical to the branch corresponding to the first test in the third

```
if((lnbn[s] == 0) && (lnbf[s][1-c] == 1) && !friendly_capture){
    /* 12-line body of if-statement removed here */
}
if(newlibs > 0)
    newlibs -= grlibs[g] - mrglist(grlbp[g],&tmplist);
if((lnbn[s] == 0) && (lnbf[s][1-c] == 0) && !friendly_capture)
    numeyes++;
```

Figure 6.10: Example of correlation across subroutine calls in go

if-statement. The correlation between these two branches is therefore very strong. However, if the body of the second if-statement is executed, the subroutine mrglist is called, and a normal history would lose information about the directions of previously executed branches. In this example, the "Normal" Selective History predictor mispredicted 5.63% of the time. The "Sub Only" Selective History predictor mispredicted 0.01% of the time (only to warm up the counters). In cases like this, it is clearly advantageous to be able to use the history from before the subroutine call.

In Table 6.3, we compare the misprediction rates of the three versions of the Selective History predictor for the regions where each of them is best. For the region where the "Normal" Selective History predictor is best, "Sub + Normal" achieves the same accuracy as it can also use the normal history information. This merely means that the history from before subroutines did not add useful information for prediction. For the region where "Sub Only" was best, there are substantial improvements over the normal history for all benchmarks but vortex. For the region where "Sub + Normal" was best, there were

	Normal		Sub Only		Sub + Normal	
	sub	norm	sub	norm	sub	norm
com	5.69%	5.69%	21.50%	23.68%	13.02%	13.22%
gcc	3.85%	3.85%	4.71%	6.15%	5.67%	6.54%
go	11.02%	11.02%	8.83%	10.40%	14.34%	15.72%
ijp	9.83%	9.83%	6.80%	7.91%	26.21%	26.96%
xli	9.02%	9.02%	6.99%	8.95%	5.72%	7.53%
m88k	1.96%	1.96%	4.01%	7.58%	1.28%	1.81%
per	1.81%	1.81%	3.72%	6.54%	2.84%	5.37%
vor	2.37%	2.37%	0.71%	0.92%	6.81%	7.67%

Table 6.3: Correlation across subroutine calls

substantial improvements for go, xlisp and perl, and smaller improvements for the other benchmarks. The improvements are smaller in the "Sub + Normal" region compared to the "Sub Only" regions as the normal predictor can detect some of the branches that are correlated.

Most benchmarks have a large fraction of branches, 40% on average, that are correlated most strongly with a set of branches including branches from before a subroutine call, where these correlated branches would normally have been replaced in the history by branches from the subroutine call. Since there is a substantial reduction in the number of mispredictions when using this history, a structure like the Return History Stack could improve a correlation based predictor. However, a good way of incorporating the return history with the normal history is needed, as there is a tradeoff between keeping history from inside the subroutine or from before the subroutine.

6.5.5 Correlation Across Loops

In a way similar to subroutines, loops can also flush useful correlated branches from the history. In this section, we investigate correlation with branches from before loops in a similar manner as was done with subroutines in Section 6.5.4. However, detecting the start of a loop is not as simple as detecting a subroutine call. The compiler knows where each loop starts and ends. However, it is not easy to get this information from the compiler. Instead, we captured this information in the simulator.

The region of code starting at the first instruction of a loop and ending at the last instruction of a loop is called a *loop body*. If needed for a predictor implementation, this information can be provided by the compiler. However, we detected the loop bodies through an initial run with the normal test set, using the following algorithm: A backward branch (except indirect branches, returns and subroutine calls) that does not straddle the boundary of an existing loop body will define a new one. The new loop body starts at the branch target and ends at the backward branch. If the backward branch does straddle the boundary of a loop body, this loop body will be extended so both the backward branch and the target of the backward branch are included in that loop body. This simple rule will properly identify almost all loops, including nested ones. The exception is loops where a branch exiting the loop goes backward farther than the beginning of the loop. In this case, some outside code will be erroneously considered to be part of the loop. However, by examining

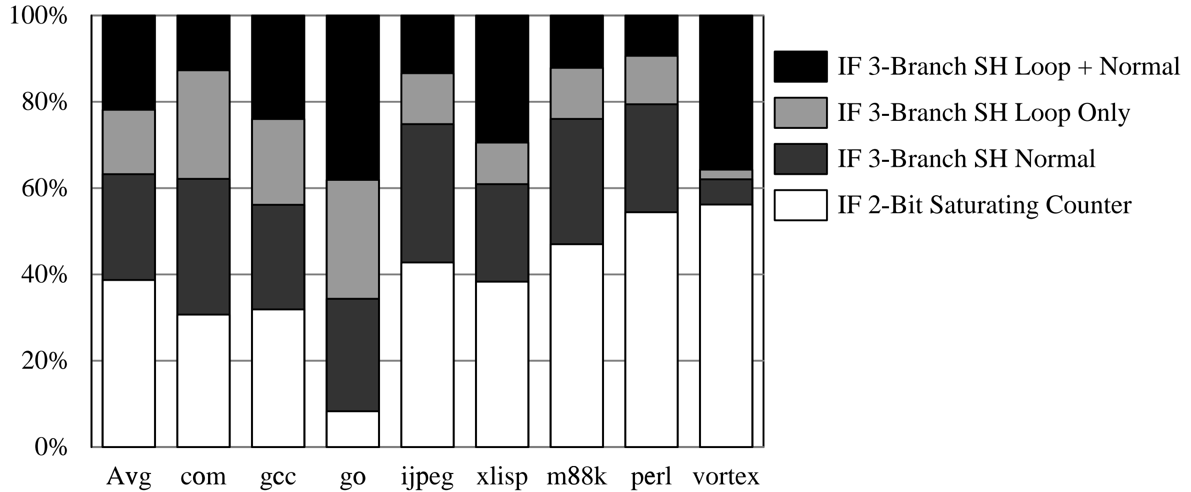


Figure 6.11: Frequency of branches with correlation across loop bodies

a large number of the loop bodies that were found, we saw that this case is very rare.

To preserve history across loops, a mechanism similar to the Return History Stack can be used. Every time a loop body is entered, the history is pushed onto the stack. When a loop body is exited, the history is popped back off the stack. A subroutine call from inside a loop is not considered to be exiting the loop as a later return is expected. A return is considered to exit any loop it is inside.

As in Section 6.5.4, we compare three versions of the Selective History predictor. The first, "Normal", is the normal version described earlier. The second, "Loop Only", preserves its history across calls using a Loop History Stack as just described. The third, "Loop + Normal", has both a normal history and the preserved history available, and can detect correlation where one or more correlated branches are inside the loop body, while others are before. Given the oracle used for selecting the most correlated branches in the Selective History predictor, the third version will always be at least as good as the two others, but it has the advantage of being able to use more history information.

Figure 6.11 shows the weighted distribution of branches according to which of these predictors has the lowest misprediction rate. The interference free Two-Bit Counter predictor is also included. On average, 37% of all branches showed some improvement from having history from before the loop body available. These are branches that are at least partially correlated with branches from before the loop body. 15% of the branches needed *only* history from before the loop body, while 22% used correlation with branches both in and

	Normal		LoopOnly		Loop + Normal	
	loop	norm	loop	norm	loop	norm
com	2.52%	2.52%	16.09%	17.09%	20.74%	20.89%
gcc	3.75%	3.75%	5.19%	6.41%	5.33%	6.20%
go	9.94%	9.94%	9.18%	10.63%	14.13%	15.47%
ijp	5.16%	5.16%	7.96%	10.07%	25.85%	26.13%
xli	12.98%	12.98%	0.79%	0.87%	6.34%	7.61%
m88k	2.21%	2.21%	1.70%	2.83%	1.03%	1.11%
per	2.18%	2.18%	5.01%	6.72%	3.89%	5.75%
vor	2.49%	2.49%	3.72%	4.38%	1.00%	1.09%

Table 6.4: Correlation across loop bodies

before the loop body. For go, 65% of the branches benefit from history information from before the loop body. Go has a large number of loops, and very often there is correlation across these loops. For ijpeg, m88k, and perl, fewer than 25% of the branches benefit from this history information. If comparing only the "Normal" and "Loop Only" configurations, 27% of the branches are predicted more accurately using only the information from before the loop. The similarities between Figure 6.11 and Figure 6.9 are striking. In fact, closer study reveals that to a large extent the same branches benefit from correlation across both loop bodies and subroutines.

In Table 6.4, we compare the misprediction rates of the three versions of the Selective History predictor for the regions where each is best. For the "Normal" region, "Loop + Normal" achieves identical accuracy. For the region where "Loop Only" was best, there are substantial improvements over the normal history for all benchmarks except xlisp and vortex. For the region where "Loop + Normal" was best, there were substantial improvements for go, xlisp and perl, and smaller improvements for the other benchmarks. In general, the improvements were smaller than in Section 6.5.4.

Most benchmarks have a large fraction of branches, 37% on average, that are correlated most strongly with branches from before a loop body, where these correlated branches would normally have been replaced in the history by branches from within the loop body. For those branches that benefit from using history from before the loop, there is a substantial reduction in the number of mispredictions. However, a cross-loop history is harder to generate than a cross-call history, and there is substantial overlap between the two methods. A cross-loop history may yield improvements in a future correlation based predictor, but those improvements are not as easy to get as those of a predictor using cross-call history.

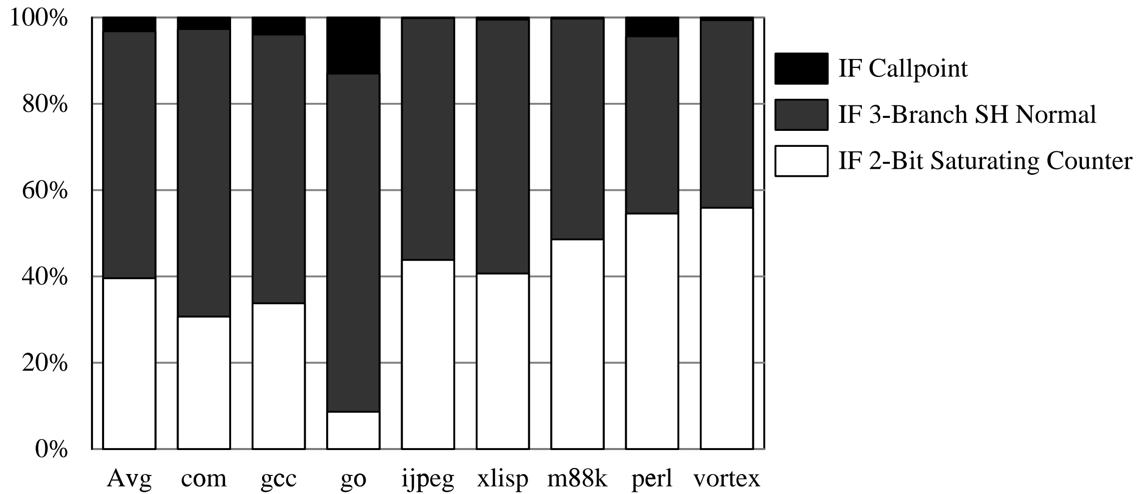


Figure 6.12: Frequency of branches correlated with the callpoint

6.5.6 Correlation with Subroutine Callpoints

Most subroutines are written such that they can be called from several different places in a program. Often, the input to the subroutine depends on where it was called from, and the input in turn affects the outcome of branches inside the subroutine. We call the place the subroutine was called from the *subroutine callpoint*, and in this section, we examine the correlation between the callpoint and the outcomes of branches inside the subroutine.

To detect the correlation between branches and the callpoint, we use a set of 2-bit saturating counters for each static branch. One counter is assigned to each possible callpoint for the subroutine the branch is in. To predict a branch, a counter is selected based on the callpoint, and that counter is used to make the prediction.

Figure 6.12 shows the weighted distribution of branches according to which of three predictors has the lowest misprediction rate. The three predictors are the normal 3-branch Selective History predictor, the Callpoint predictor just described, and an interference free Two-Bit Counter predictor. Only a few branches, 3% of all branches on average, are best predicted using the Callpoint predictor. For go, the number is 13%. Compress, gcc and perl have some callpoint correlation, while the Callpoint predictor is rarely the best for the other benchmarks.

Table 6.5 compares the misprediction rates of the Callpoint and Selective History predictor for the regions where each predictor is best. For go, gcc, and perl there are solid improvements for the small set of branches for which the Callpoint predictor is best. There

	Normal		Callpoint	
	callp	norm	callp	norm
com	23.64%	11.62%	0.00%	0.02%
gcc	14.17%	5.30%	4.70%	7.47%
go	20.51%	10.76%	17.29%	22.84%
ijp	14.90%	11.00%	12.21%	17.19%
xli	19.19%	8.80%	1.69%	11.39%
m88k	7.88%	2.05%	7.29%	13.05%
per	16.72%	3.90%	2.91%	4.41%
vor	3.19%	1.42%	4.12%	8.15%

Table 6.5: Callpoint vs. selective history by region

is little difference for compress. Callpoint correlation performs poorly for all benchmarks on the rest of the branches, so it can not be considered to be a general prediction scheme. Given the small set of branches for which it is the best predictor, it is unlikely that the cost of the Callpoint predictor can be justified as a specialized predictor.

6.5.7 Dependence of Correlation on Input Set

For the Selective History predictor, much depends on being able to find the few most important branches to record in the history. It is therefore important to determine whether the correlation stays constant between input sets. In the following experiment a profiling set is used to identify the best 3 branches for the selective branch history, and those 3 branches are used for the selective history when running the normal test set.

The first column of Table 6.6 shows the percentage of branches in the test set that were covered by the profiling set. For those branches that were covered, the misprediction rate of the Selective History predictor is shown both when using the oracle to select branches for the history and when using the profile to select branches. The profile coverage was less

	Coverage	Oracle	Profiled
com	100.00%	7.76%	8.20%
gcc	94.14%	3.76%	4.10%
go	99.96%	11.46%	12.73%
ijp	100.00%	6.77%	7.07%
xli	99.78%	5.28%	7.42%
m88k	88.88%	1.53%	1.73%
per	89.23%	1.98%	5.41%
vor	99.94%	0.78%	1.28%

Table 6.6: Identifying most correlated branches using profiling

than 90% for m88ksim and perl. For gcc, the coverage is 94% which is also low. For these benchmarks, the profiling and testing sets are so different that many of the branches in the testing set are not seen in the profiling set. This indicates that a better profiling set is needed.

For 5 of the benchmarks—compress, gcc, ijpeg, m88ksim and vortex—using the profiled sets of branches for the history works reasonably well. However, for go, xlisp and especially perl, the profile is a poor predictor of which branches will be the most correlated. For perl the situation is particularly bad. The input set, primes.pl, causes most branches to be heavily biased. When a branch is heavily biased, you can not determine which other branches it is correlated to, so selecting a set of 3 correlated branches becomes a matter of guessing. A better profiling set might remedy this. One way to get better profiling is to use more than one profiling set. Another way is to make sure that the profiling set actually represents likely behavior for the benchmark. At the very least, the profiling set should not represent trivial behavior for the benchmark, such as is the case with perl. Finally, the compiler may be able to identify some of the most correlated branches based on code analysis. Using better profiling sets is likely to improve our ability to identify the most correlated branches using profiling, but is unlikely to solve the problem completely.

Despite the problems with finding the set of correlated branches through profiling, the correlation detected earlier in this chapter is still there. However, if we want to use the Selective History predictor to exploit it, a better method of finding the correlated branches is needed.

6.5.8 Comparison of Correlation Based Methods

In Section 6.5.4 to 6.5.6, several forms of correlation were investigated and compared to basic correlation. To give a better view of the relative importance of each of these types of correlation, they are all compared here. Figure 6.13 shows the distribution of branches according to which correlation based prediction method is best. For the loop and subroutine based correlation, the "Loop + Normal" and the "Sub + Normal" variations are used. The "Tie" category refers to ties between interference free Gshare and any other method except the interference free Two-Bit Counter predictor. For other ties, the simpler class (lower on the figure) wins.

On average, the interference free Two-Bit Counter predictor is the best for 37% of all

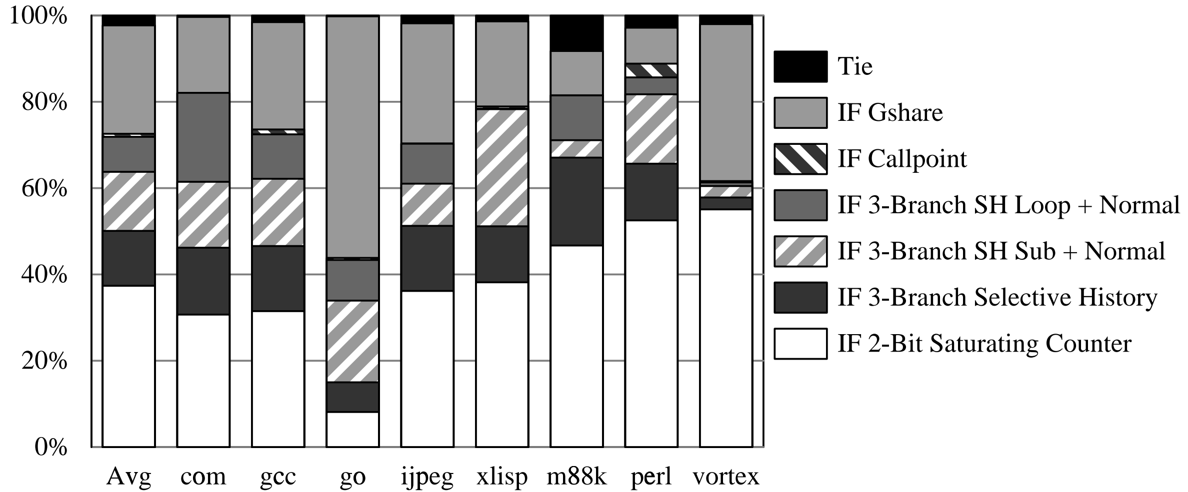


Figure 6.13: Distribution of branches by best predictor. All correlation methods considered

branches. As mentioned earlier, these are mostly strongly biased branches, but also some branches for which correlation is not a useful prediction method. 13% of the branches are best predicted with a Selective History predictor using simple correlation. For 14%, correlation across subroutine calls works better, while 8% are best predicted with correlation from before loop bodies. Callpoint correlation is a small factor with 1%. Interference free Gshare is best for 25% of all branches. These are the branches that benefit from the more complex correlation that can be detected by Gshare's 16-branch history.

The exceptions from the average are that the Two-Bit Counter predictor does poorly for go, with a corresponding increase in the region for which the interference free Gshare is best. Xlisp and go have more correlation across subroutine calls than the other benchmarks. Compress has more correlation across loops. Vortex has little correlation of the direct nature detected by variations of the Selective History predictor. Callpoint correlation is only significant for perl and gcc.

Table 6.7 compares the misprediction rate of the interference free Gshare predictor to that of the best of the correlation based methods for each region. The consistently best improvements over interference free Gshare are for the branches best predicted using correlation across subroutine calls and loop bodies. This is hardly surprising, as this is correlation that Gshare was not designed to detect.

Gshare is generally thought of as the primary correlation based branch predictor. How-

	3 br SH		Sub		Loop		Callpoint		IF Gshare	
	3 br	IF gsh	sub	IF gsh	loop	IF gsh	callp	IF gsh	3 br	IF gsh
	(%)	(%)	(%)	(%)	(%)	(%)	(%)	(%)	(%)	(%)
com	4.10	5.45	12.66	13.31	19.85	20.95	*	*	4.67	3.06
gcc	1.65	2.76	4.12	6.17	4.53	6.73	1.24	5.36	7.47	5.04
go	2.67	4.79	5.81	8.29	5.27	8.78	3.43	12.16	16.01	9.60
ijp	3.13	3.54	7.09	9.18	34.23	36.84	0.45	2.74	6.41	5.72
xli	12.36	12.79	5.74	7.84	0.47	1.28	1.70	10.63	7.12	3.32
m88k	1.72	2.03	2.36	3.86	1.51	2.32	*	*	2.01	0.97
per	1.37	2.44	2.30	5.68	3.30	5.54	0.01	0.02	8.31	2.86
vor	1.88	2.63	4.02	7.73	4.75	7.04	1.60	4.33	1.18	0.78

Table 6.7: Comparison of all correlation based methods. "*" means sample to small to be meaningful

ever, based on this study, we see that Gshare is frequently unable to exploit the available correlation. Two major reasons explain why Gshare does not exploit all of the correlation. First, Gshare often looks at too much history, so it suffers from noise caused by uncorrelated or unimportant branches. Second, Gshare can not see the history before subroutine calls and loop bodies if the subroutine called or loop body includes more branches than can fit in the history. To create a better correlation based predictor, these reasons should be addressed.

6.6 Improving Branch Correlation Based Prediction

In this section, we examine how we can use the characteristics of branch correlation that were shown earlier in this chapter to improve correlation based prediction. Three of the characteristics that were shown are immediately useful for predictor design.

First, we saw in Section 6.5.1 and 6.5.2 that the explicit correlation between branches is not always captured by current correlation based predictors, and that this explicit correlation was often between the current branch and only a few correlated branches in the past history. We do not yet have a good method of capturing explicit correlation, but in many cases it is likely that a short global history can capture this correlation better than a long history. This is because the shorter history is less susceptible to noise from uncorrelated branches.

Second, we saw in Section 6.5.1 and 6.5.2 that for a large subset of branches a long

correlation based history was substantially better than using a shorter history based on explicit correlation.

Third, we saw in Section 6.5.4 that branches were often correlated with other branches that were separated by subroutine calls. For these branches, using a Return History Stack to preserve history across subroutine calls resulted in increased accuracy. This was shown to be the case using explicit correlation in a selective history predictor in Section 6.5.4, but is also true for a Gshare predictor.

We focus the thrust to improve correlation based prediction on how to improve a Gshare predictor. The first improved predictor proposed here uses two Gshare components with different history lengths. Having a short history was suggested by the first characteristic above. Having a long history was suggested by the second characteristic above. To keep this predictor cost-effective, we allocate most of the storage to the Gshare component with the longer history length. A two-level selection mechanism is used to select whether to use the component with a short or a long history. Two-level selection mechanisms were proposed in [4], and are used here because they work better than address-only mechanisms. To simplify the implementation, the selection mechanism uses the same index as the Gshare component using a short history. Interference is always an important concern in global two-level predictors. To reduce the amount of interference in the predictor we propose here, the large predictor is only updated if the small predictor has recently mispredicted the pattern. This predictor is explained in detail in Section 6.6.1.

The second improved predictor we propose here is an extension of the first to also address the third characteristic above. A third Gshare component using a Return History Stack is added. Also, a new selection mechanism is proposed to dynamically choose between the three components of this predictor. The new selection mechanism was needed as standard selection mechanisms can only select between two component predictors at a time. This predictor is explained in detail in Section 6.6.2.

In the past, other mechanisms for improving correlation based predictors have been proposed. Most of these were based on reducing interference. The best basic correlation based two-level predictor is Gshare, so this predictor is used as a baseline. Four of the mechanisms that improve correlation based predictors were compared in [17], and it was found that Branch Filtering outperforms the other mechanisms assuming that the fetch unit has a Branch Target Buffer (BTB) or other tagged structure in which the filter counters can

be kept. A filtered Gshare predictor is also used for comparison in this section. Mechanisms such as Branch Filtering, Agree prediction, and Bi-Mode structures can also be applied to the predictors we introduce in this section. This would produce a small additional decrease in the misprediction rate. However, to maintain the implementation advantage of the new predictors, we decided not to apply any of those mechanisms.

The two new predictors that are introduced in this section are more accurate than Gshare while still relying only on branch correlation for prediction. The first of the predictors also improves over a filtered Gshare, while being simpler to implement. The second of the predictors improves over the first for sizes over 23 KB but has a slightly more complex implementation.

6.6.1 Dual History Length Gshare with Selective Update

The data in Section 6.5 suggested that Gshare frequently uses more history than is needed to make a prediction. To alleviate this problem, we designed a hybrid predictor using two Gshare components with different history lengths. One component, referred to as “small Gshare”, has a short history to capture correlation for which only the most recent branch outcomes are needed. The other, “large Gshare” has a longer history to capture more complex correlation. There is also a selection mechanism to select which of the two Gshare predictors to use. Finally, there is a table to keep track of how accurate the small Gshare predictor is for each pattern, so that the update to the large Gshare can be inhibited when the small predictor is very accurate. This reduces the amount of interference suffered by this predictor.

Figure 6.14 shows a diagram of the Dual History Length Gshare predictor. The large and small Gshare predictors function the same way as regular Gshare predictors. However, each uses a different number of history bits to generate the index. The index that is used for the small Gshare component is also used to index the hybrid selection counters and update counters. The hybrid selection mechanism therefore uses some bits of the global history in addition to the branch address to decide which predictor is better, as suggested in [4]. The hybrid selection counters are 3-bit counters, which are used to select between the two components in the same way as the counters used in McFarling’s selection mechanism (see Chapter 2). 3-bit counters work better than 2-bit counters. Their decision is more stable, so temporary changes in which predictor is best is less likely to change the selection

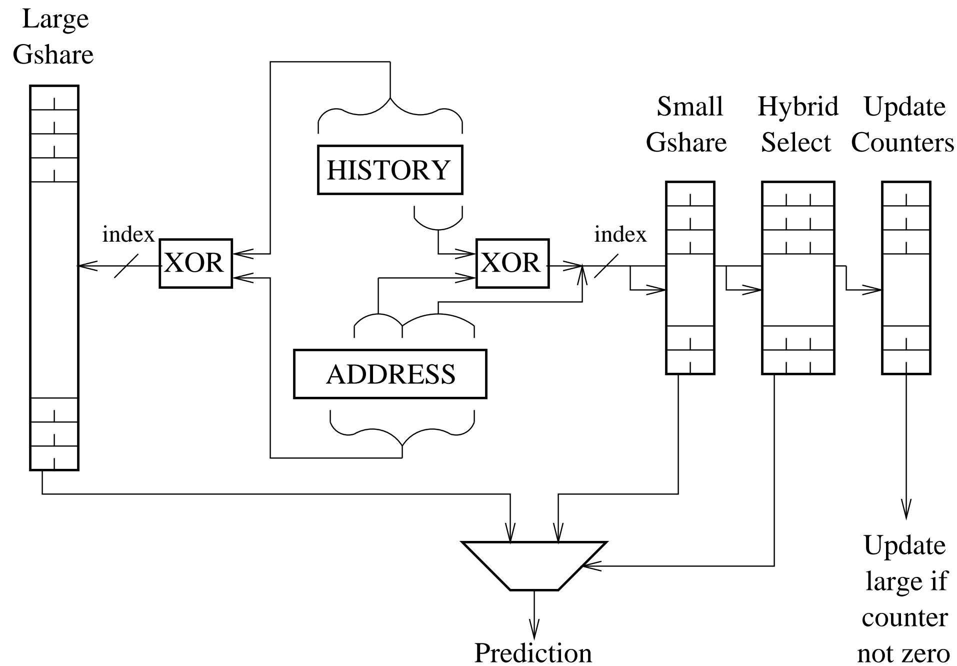


Figure 6.14: Dual History Length Gshare with Selective Update

prematurely. The short history Gshare index is also used to select a 2-bit counter from the Update Counter Table. This counter is used to decide whether to inhibit the update of the larger predictor. If the counter is zero, the update to the large Gshare is inhibited, otherwise the update is allowed. When the small Gshare mispredicts, the counter is set to 3. Every time thereafter, the counter is decremented if the large Gshare mispredicts. This way, the large Gshare component is only updated for patterns the small Gshare has recently mispredicted.

Using two Gshare components with different history lengths in a hybrid predictor has been suggested before. GAS.mhl [5] uses a single Gshare predictor, but has the ability to select between two index functions (each with a different history length) using a bit in the branch opcode. This bit was determined based on profiled taken rates. Heavily biased branches used the shorter history length. However, due to the imperfections of profiling, and the lack of ability to change history lengths throughout the run, this predictor improved only marginally over Gshare for sizes of 4 KB and over. The Variable Length Path (VLP) branch predictor we introduced in [29] can select between 32 index functions with different history lengths. The profiling step for this predictor simulates the predictor for each of the index functions, and decides the appropriate hash function for each branch through an iterative refinement procedure that goes through the profiling run seven times.

Size	Small Gshare	Large Gshare	Selector	Update Table
	(HL,PHTs)	(HL,PHTs)	(HL,PHTs)	(HL,PHTs)
6 KB	(1,1024)	(14,1)	(1,1024)	(1,1024)
12 KB	(2,1024)	(15,1)	(2,1024)	(2,1024)
23 KB	(3,1024)	(16,1)	(3,1024)	(3,1024)
46 KB	(3,2048)	(17,1)	(3,2048)	(3,2048)
92 KB	(4,2048)	(18,1)	(4,2048)	(4,2048)
184 KB	(4,4096)	(19,1)	(4,4096)	(4,4096)

Table 6.8: Configurations for Dual History Length Gshare with Selective Update

There are several disadvantages of the VLP predictor compared to the predictor described here. The VLP requires an extensive profiling step. Without profiling, the performance is similar to Gshare. The instruction set must be augmented to communicate which index function to use for each branch to the hardware. Finally, the hardware required for index generation and restoring state after pipeline flushes in the VLP is extensive. However, although more complex, the accuracy of the VLP is better than that of the Dual History Length Gshare with Selective Update. We believe there are times when the extra accuracy of the VLP predictor warrants this more complicated implementation. However, often the simpler dynamic predictor presented here will be preferable.

The idea of selectively updating a predictor has also been seen before in different variations. Statically selected hybrid predictors only update the component that is used for each branch. Branch Filtering [3], the Skewed branch predictor [21], the Bi-Mode predictor [18], the Path-Based Next Trace predictor [14], and the YAGS [8] predictor all use different forms of selective update. However, it is the combination of two Gshare components with different history lengths and the new method of selective update that is used in this predictor that makes it better than the other predictors. Furthermore, unlike Branch Filtering, no BTB is needed to make the conditional branch prediction, and unlike the Path-Based Next Trace predictor and YAGS, no tag matches are required to determine which component to use.

The configurations of the components used for the Dual History Length Gshare with Selective Update at various sizes are given in Table 6.8. In this table, "HL" refers to the global history length, and "PHTs" is the number of pattern history tables (which PHT to use is selected using the lower bits of the branch address). A large number of configurations were tested to find these configurations. However, due to the large design space of such a

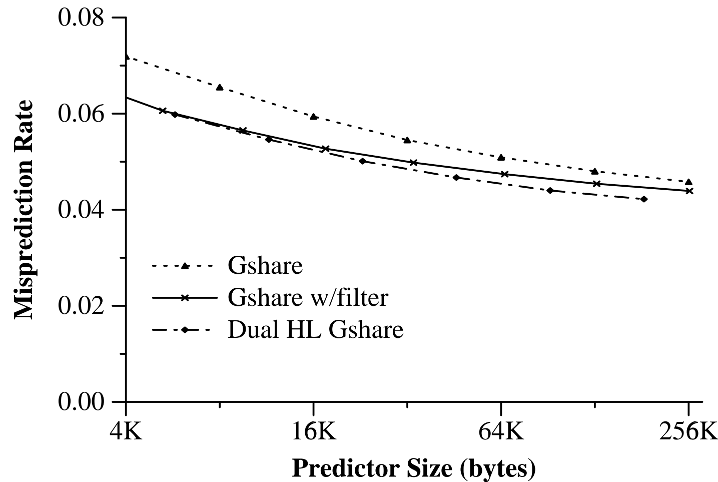


Figure 6.15: Performance of Dual History Length Gshare with Selective Update

predictor there is no guarantee that there are no better configurations. In particular, the configurations examined were limited to those where the small Gshare, the hybrid selector, and the Update Table, use the same index.

Figure 6.15 compares the misprediction rates of the Dual History Length (DHL) Gshare with Selective Update to the misprediction rates of Gshare and Gshare with Branch Filtering. DHL Gshare is always much better than the normal Gshare. The reduction in average misprediction rate varies between 13%[7] at 6 KB and 10% at 184 KB. Compared to a filtered Gshare, DHL Gshare has the same accuracy for a 6 KB predictor, and reduces mispredictions by 5% at 184 KB. The selective update policy is responsible for 1-4% of the improvement over Gshare depending on the size, whereas the Dual History Length configuration is responsible for the remaining part.

The Dual History Length Gshare with Selective Update performs much better than Gshare and better than even a filtered Gshare for moderate to large sizes. A 23 KB DHL Gshare predictor performs as well as a 32 KB filtered Gshare or a 64 KB regular Gshare predictor. This is an important improvement given that the DHL Gshare is simpler to implement than a filtered Gshare predictor. The critical path in the DHL Gshare predictor has only one MUX delay (data in to data out) added to the cycle time of a Gshare predictor. The select for the MUX will be set up early in the cycle as the hybrid selection table is

[7] As the sizes of the Gshare, Gshare with Branch Filtering and DHL Gshare predictors differ, the misprediction rates of the first two were extrapolated from nearby data points.

smaller than the large Gshare bank.

If desired, the small Gshare bank can also be used to make an initial prediction to fit in a shorter cycle time. For the few branches for which the prediction of the small Gshare is overturned by the large Gshare bank, a one cycle penalty would result to get the fetch mechanism back on track.

6.6.2 Dual History Length Gshare with Selective Update and Return History Stack

The data in Section 6.5 also suggested that correlation is often undetected because the correlated branches were flushed from the history due to subroutine calls located between the correlated branch and the current branch. In this section we add a component using a 64-entry Return History Stack to the DHL Gshare predictor. A Gshare using a Return History Stack only marginally outperforms a regular Gshare predictor (reduction in mispredictions is less than 1%[8]). However, we here investigate its usefulness as a component.

Conventional hybrid selection mechanisms only provide the ability to select between two component predictors. Therefore, we also introduce a selection mechanism capable of selecting between any number of component predictors.

To select between 3 component predictors, we use a table of counters as in a normal hybrid. A Gshare-style index created from global history bits and the branch address is used to select an entry in the table. However, each entry consists of 3 3-bit counters, rather than the single counter in the conventional scheme. The values in these counters represent the relative accuracies of the three component predictors. The higher the counter value, the more accurate the associated component has been recently. The first counter is associated with the small Gshare, the second is associated with the large Gshare and the third is associated with the Gshare using a Return History Stack to preserve its history across subroutine calls.

[8] For a 16 KB predictor, the best configuration conserves the 4 most recent bits from the subroutine and only uses the Return History Stack if the subroutine has more than 4 conditional branches in it. The predictor used as a component in this section does not conserve any of the most recent bits from the subroutine.

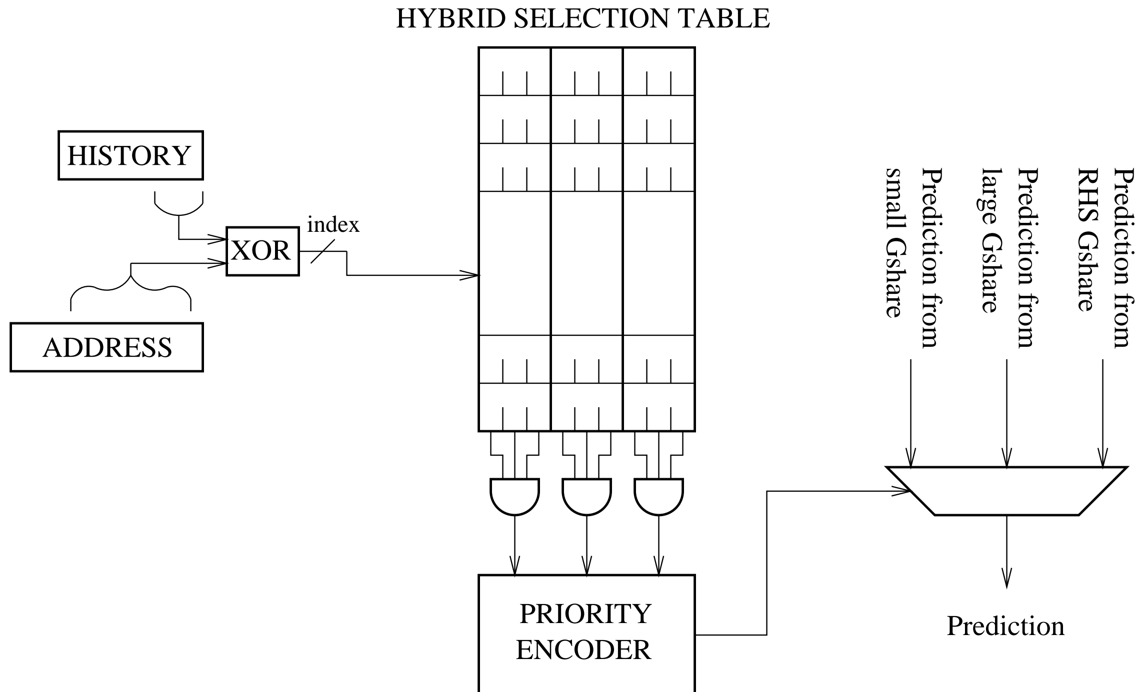


Figure 6.16: Selection Mechanism for a Dual History Length Gshare with Selective Update and Return History Stack

Figure 6.16 shows how the prediction is selected in the DHL predictor with a Return History Stack. The global history and branch address are used to form an index into the hybrid selection table. Each of the counters is associated with one of the predictors, and the predictor whose counter has the maximum value, 7, is chosen. The three counters that are selected are examined using and gates to determine whether the value is 7. If more than one counter is at the maximum, a priority encoder is used to decide which predictor to use. The small Gshare has the highest priority followed by the large Gshare and the Gshare using a Return History Stack. The update scheme guarantees that at least one of the counters is always at 7.

The counters are used and updated using the following algorithm:

- When the processor is reset, all counters are reset to 7
- The counters get updated when the branch is resolved. If one of the predictors that had the value 7 in its selection counter was correct, the selection counters for all the incorrect predictors are decremented. Otherwise, the selection counters for all the correct predictors are incremented.

This updating strategy guarantees that the value in at least one of the selection counters will be 7, simplifying our predictor selection mechanism. Our predictor selection mechanism also captures more information than saturating counters because it can better differentiate which of the component predictors are currently more accurate for each branch. For example, given the same initial counter values, the selection mechanism can differentiate a predictor that has been correct for the last 9 times from a predictor that has been correct for the last 8 times, while saturating counters can not.

If the selection mechanism is on the critical path, a variation can be implemented to reduce the time required to make a prediction. The priority encoding can be computed when the selection counters are updated and stored in the hybrid selection table. Thus, when the branch is fetched, the previously calculated priority encoding is used to directly select the appropriate prediction. The resulting selection mechanism requires only one extra mux delay (data in to data out) for choosing the appropriate prediction from the component predictors. Since the selection table is smaller than the large Gshare component predictor, it is possible that the priority encoding can be done at fetch time without affecting the cycle time.

As with the DHL Gshare predictor in Section 6.6.1, there is also an Update Counter Table. This table is used to determine whether to update the two larger predictors. A counter is selected using the same index as used for the small Gshare component. If the counter is zero, the update to the large Gshare and Gshare with RHS is inhibited, otherwise the update is allowed. When the small Gshare mispredicts, the counter is set to 3. Every time thereafter, the counter is decremented if the other component (large Gshare or Gshare with RHS) with the higher value in its selection counter mispredicts.

The configurations of the components used for the different size implementations of the Dual History Length (DHL) Gshare with Selective Update and Return History Stack (RHS) are given in Table 6.9. A large number of configurations were tested to find these configurations. However, due to the large design space of such a predictor there is no guarantee that there are no better configurations. In particular, in looking for the best configurations, we restricted the search to configurations where the small Gshare, the hybrid selector, and the Update Table had the same history length.

Size	Small Gshare	Large Gshare	Gshare w/RHS	Selector	Update Table
	(HL,PHTs)	(HL,PHTs)	(HL,PHTs)	(HL,PHTs)	(HL,PHTs)
7 KB	(1,1024)	(14,1)	(11,1)	(1,1024)	(1,1024)
14 KB	(2,1024)	(15,1)	(12,1)	(2,1024)	(2,1024)
28 KB	(3,1024)	(16,1)	(13,1)	(3,1024)	(3,1024)
56 KB	(4,1024)	(17,1)	(14,1)	(4,1024)	(4,1024)
112 KB	(4,2048)	(18,1)	(15,1)	(4,2048)	(4,2048)
224 KB	(4,4096)	(19,1)	(16,1)	(4,4096)	(4,4096)

Table 6.9: Configurations for Dual History Length Gshare with Selective Update and Return History Stack

Figure 6.17 compares the misprediction rates of the DHL Gshare with Selective Update and RHS to the misprediction rates of Gshare and Gshare with Branch Filtering. DHL Gshare with RHS is always much better than the normal Gshare. The reduction in mispredictions varies from 12% at 7 KB and 14% at 224 KB. Compared to a filtered Gshare, DHL Gshare with RHS increases mispredictions by 1% for a 7 KB predictor, and reduces mispredictions by 10% at 224 KB. The selective update policy is responsible for 1-3% of the improvement over Gshare depending on the size.

The Dual History Length Gshare without the Return History Stack was not shown in Figure 6.17 as the performance is too close to easily distinguish the two lines. The improvement over the Dual History Length Gshare without the Return History Stack starts at 23 KB and grows to a 4% reduction in mispredictions at 184 KB. This comes at an

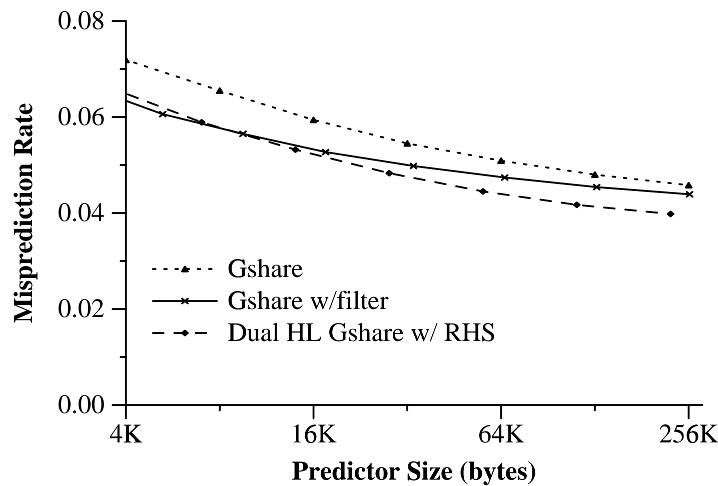


Figure 6.17: Performance of Dual History Length Gshare with Selective Update and Return History Stack

increase in complexity. Which of these predictors is best would have to be determined based on how well they fit into the processor implementation. However, both are simpler to implement than a filtered Gshare. One advantage of this predictor is that it can easily be extended to include other components that are not based on branch correlation.

6.7 Summary

In this chapter we have examined the nature of branch correlation and have shown that most branches do not need a large amount of history for prediction. We showed that for most branches, correlation with fewer than 3 previous branches was needed. We also showed that Gshare, using a pattern history of 16 branches, was often unable to capture the direct correlation with 1, 2, or 3 branches, indicating that Gshare often examines too much history. It was suggested that large improvements can be had if predictors more directly use correlation between branches to make predictions.

Furthermore, we investigated the effect of preserving history from before subroutines and loop bodies. We showed that about 20% of all branches can be predicted more accurately using a history that includes the outcomes from before subroutines and loop bodies and that the reduction in misprediction rate from using this information was often large.

Two versions of the Dual History Length Gshare predictor with Selective Update were introduced: one with and one without a mechanism to keep history from before subroutine calls. These predictors improved on Gshare by up to 14% and improved on a filtered Gshare by up to 10%. The complexity of these predictors is a little higher than that of Gshare. However, both predictors are less complex than a filtered Gshare predictor as there is no need for a BTB access or tag match on the prediction path.

The performance of the Dual History Length Gshare predictors is encouraging. However, comparing our results with the hypothetical predictors of Table 6.2, it is clear that there is room for further improvements. Still, the predictors introduced in this chapter represent a good improvement in correlation based prediction. This is especially true when considering the simpler implementation compared to a filtering mechanism.

CHAPTER 7

Interaction Between Branch Predictors

Most branch predictors are either single scheme predictors, such as Gshare or PAs, or hybrid predictors, such as PAs/Gshare or 2bc/Gshare. The hybrid predictors are most accurate, and are made up of two or more single scheme predictors and a selection mechanism to decide which of the single scheme, or component, predictors to use for each branch. To build better hybrid predictors, it is useful to understand how single scheme predictors interact and complement each other. Two branch predictors that complement each other well, such as PAs and Gshare, combine into a good hybrid predictor, whereas two predictors that are similar, such as GAs and Gshare, do not combine to make as good a hybrid predictor.

In this chapter, the interaction between six single scheme predictors is examined. First, we investigate the proportion of branches for which each of these is the best predictor, and the importance of using the best predictor for each category of branches. We investigate how the best predictor for each branch changes during the run of a program, and between runs using different input sets. It is shown that the best predictor changes frequently. Second, several real and idealized selection mechanisms are investigated. The effect of using history in the selector as suggested in [4] is shown. The size of the selector has largely been ignored by previous studies, so the effect of changing the size of the selector is also examined. Finally, a new hybrid predictor, which achieves a lower misprediction rate, at a given cost, than previously reported predictors, is introduced.

7.1 Fundamentals

In this section, we try to answer three fundamental questions for hybrid branch predictors: Which single scheme predictors are important candidates as components for a hybrid branch predictor? Does the best predictor for a given branch change throughout the run of a program? Does the best predictor for a given branch change between input sets? These three questions are important in terms of which components a hybrid predictor should use and how to best select between them.

The six potential component predictors studied here are: Two-Bit Counter, PAs, a Loop predictor, Short and Long History Gshare, and a Gshare predictor with a Return History Stack. The Two-Bit Counter predictor is included because it is one of the most basic dynamic branch predictors, and has a very short training time. The PAs and Loop predictors are included because the study in Chapter 5 showed that these two predictors captured different types of self correlation. The three variations of the Gshare predictor are included because the study in Chapter 6 indicated that these three predictors each captured different types of branch correlation. Each of the single scheme predictors except for the Loop predictor were examined at a size of 16 KB. The short history Gshare used 4 history bits and 4096 PHTs, the long history Gshare, the Gshare with Return History Stack, and the PAs[1] predictor used 16 history bits and 1 PHT. The Loop predictor was simulated at a cost of 4 KB.[2]

7.1.1 Importance of Each Single Scheme Predictor

The importance of each of the predictors we are considering for use in a hybrid predictor can be estimated using two factors: how many branches each predictor is best for, and how much better than the other predictors it is for these branches.

Figure 7.1 shows the distribution of branches by best predictor. For example, if the PAs predictor was best for a static branch, that branch, weighted by execution frequency, would be accounted for in the PAs category. In a tie including the Two-Bit Counter predictor, the Two-Bit Counter category was used. For all other ties, the Tie category was used. For a quarter of the branches, the Two-Bit Counter predictor is best. These are mostly

[1] A PAs predictor using only one PHT can also be referred to as a PAg predictor.

[2] As explained in Chapter 5, the loop counters are kept in the BTB and the size of the BTB is kept fixed at 2 K entries. Each BTB entry holds one direction bit, a 7-bit counter value, and a 7-bit value containing the period of the loop.

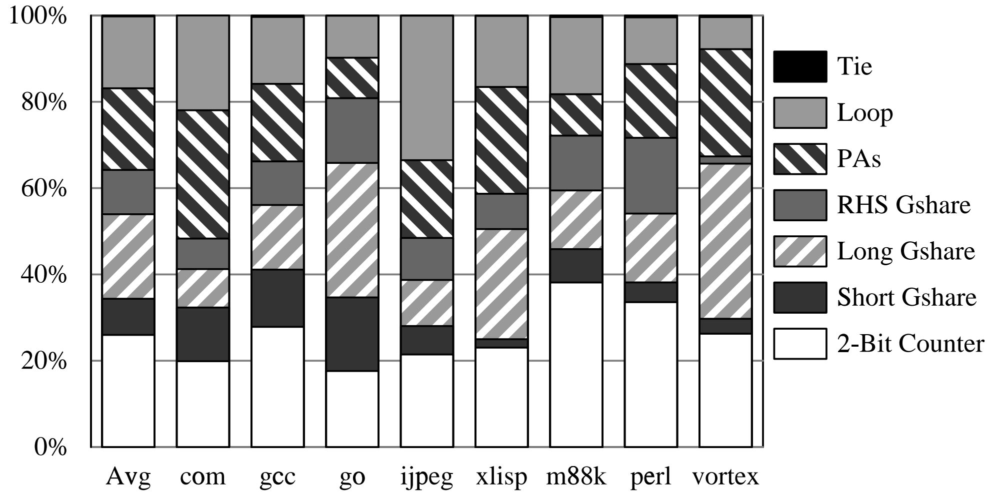


Figure 7.1: Distribution of branches by best predictor

highly biased branches, or branches that are executed only a few times so that the short training time of the Two-Bit Counter predictor is the most important factor. However, interference in the other predictors can also make a branch better predicted using less history information. Almost 20% of the branches are best predicted by each of the Gshare, PAs, and Loop predictors. These are branches for which correlation (branch correlation for Gshare and self correlation for PAs and Loop) is important. The final two branch correlation based predictors, short history Gshare and Gshare with a Return History Stack (RHS) each account for 10% of all branches. In all, 40% of all branches are best predicted using branch correlation (Gshare variations) and 35% are best predicted using self correlation (PAs and Loop). There is some variation between the benchmarks. For go, branch correlation based predictors are best for 60% of the branches while for compress and ijpeg, 50% of the branches are best predicted using self correlation.

The distribution that was shown in Figure 7.1 indicates that the Two-Bit Counter predictor, long history Gshare, PAs, and the Loop predictor are almost equally important. It is also possible that the short history Gshare and Gshare with RHS are important as they are each best for 10% of the branches. However, this does not show whether another predictor is almost as good for these branches. To further investigate the benefits of each predictor, we calculated the penalty for removing one of the six predictors while keeping the other five. The penalty is the prediction accuracy if the best of the six predictors for each branch is used minus the prediction accuracy if the best of the remaining five predictors for

	2bc	Short Gsh	Long Gsh	RHS Gsh	PAs	Loop
com	0.00%	0.00%	0.01%	0.00%	1.86%	0.01%
gcc	0.08%	0.27%	0.38%	0.20%	0.12%	0.09%
go	0.09%	0.63%	0.50%	0.27%	0.28%	0.07%
ijp	0.01%	0.00%	0.13%	0.02%	2.12%	0.39%
xli	0.00%	0.00%	0.32%	0.09%	0.93%	0.02%
m88k	0.00%	0.00%	0.07%	0.04%	0.31%	0.46%
per	0.00%	0.01%	0.24%	0.27%	0.16%	0.00%
vor	0.00%	0.01%	0.05%	0.04%	0.05%	0.01%

Table 7.1: Difference between accuracy of best predictor and accuracy of second best predictor

each branch is used.

Table 7.1 shows the penalty for each of the benchmarks if one of the predictors is removed. There is very little penalty for removing the Two-Bit Counter predictor (2bc in the table). Although this was the best predictor for 25% of the branches, the short history Gshare, PAs, and Loop predictors are seldom far behind. The only predictors the Two-Bit Counter predictor ever holds a large advantage over is the long history Gshare and Gshare with a RHS. The short history Gshare is mostly important for the two benchmarks, gcc and go, that execute a large number of static branches frequently. This is primarily because of the large amount of interference suffered by the Gshare predictors for these benchmarks, but also to a lesser extent because the warmup time is more important given the large number of static branches. When the short Gshare is best, the Two-Bit Counter predictor is usually second best. The penalty for removing short history Gshare if the Two-Bit Counter predictor has already been removed increases to about 0.5% for gcc and 1.0% for go. There is little reason to have both a Two-Bit Counter and a short history Gshare as components for the same hybrid branch predictor. The penalty for removing the long history Gshare is more consistent for all benchmarks, and generally larger than for the short history version. The penalty would be much larger, on par with the penalty for removing PAs, if Gshare with RHS was not also considered, as there is substantial overlap between these predictors. The penalty for removing Gshare with RHS is about the same as the penalty for removing short history Gshare. For the self correlation based predictors, there is a large penalty for removing PAs, especially for compress and ijpeg. The penalty for removing the Loop predictor is close to that of removing a short history Gshare or Gshare

with RHS.

These results indicate that PAs and a Gshare predictor with a long history, or closely related predictors should be included as components in a hybrid predictor. Furthermore, the Loop predictor is likely to be worthwhile as it comes at a low cost and there is a moderate penalty for not including it. The short history Gshare can work well at smaller implementation costs than the 16 KB version used here, and may be a useful addition, especially with large footprint benchmarks in mind. The Two-Bit Counter and Gshare with RHS show less promise.

Next we look at how interference affects which predictors are best. Figure 7.2 is similar to Figure 7.1 with the exception that the predictors used are interference free.[3] The main difference is that the long history Gshare increased its share from 30% to 42% on average, with corresponding reductions in the shares for the short history Gshare and Gshare with RHS. This indicates that one of the main reasons both short history Gshare and Gshare with RHS were frequently better in the previous experiment was interference. This was expected for the short history Gshare, but unexpected for the Gshare with RHS. There is some overlap between Gshare with and without a RHS, so these predictors are sometimes able to predict the same branches. However, interference will affect the two predictors differently, thus occasionally making one predictor better for a branch even though the other predictor has slightly better information for predicting that branch. Since long history Gshare is generally the more accurate of the two, this effect is more likely to work in favor of Gshare with RHS.

Table 7.2 shows the penalty of removing one of the interference free predictors. When comparing to Table 7.1, we see that the penalties for removing the Two-Bit Counter, short history Gshare, and Gshare with RHS predictors have almost disappeared. However, the short history Gshare still has a slight advantage over the long history version for gcc and go. We think this is entirely due to training time. The PAs and Loop predictors still have penalties for removal that are similar to those given earlier.

[3] A 16 K entry highly associative BTB was used to hold the counters for the loop predictors and the histories for the PAs predictors to remove interference also at this level.

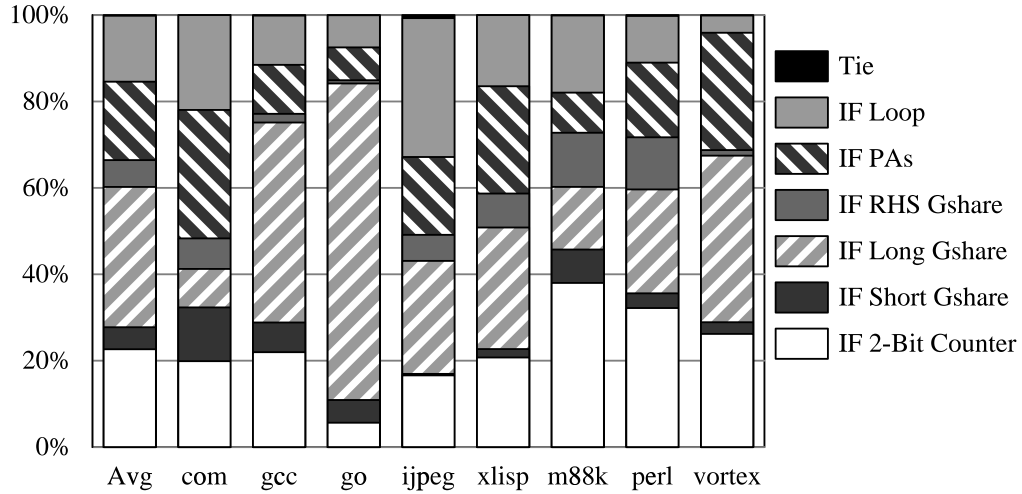


Figure 7.2: Distribution of branches by best predictor (interference free)

	2bc	Short Gsh	Long Gsh	RHS Gsh	PAs	Loop
com	0.00%	0.00%	0.01%	0.01%	1.85%	0.01%
gcc	0.01%	0.03%	1.32%	0.09%	0.07%	0.08%
go	0.01%	0.07%	3.78%	0.02%	0.17%	0.05%
ijp	0.01%	0.00%	0.24%	0.02%	2.07%	0.38%
xli	0.00%	0.00%	0.33%	0.08%	0.93%	0.02%
m88k	0.00%	0.00%	0.08%	0.04%	0.30%	0.46%
per	0.00%	0.00%	0.26%	0.18%	0.16%	0.00%
vor	0.00%	0.00%	0.08%	0.03%	0.05%	0.01%

Table 7.2: Difference between accuracy of best predictor and accuracy of second best predictor (interference free)

	Probability of Change
com	9.64%
gcc	17.13%
go	35.07%
ijp	14.02%
xli	10.91%
m88k	2.84%
per	3.92%
vor	3.12%

Table 7.3: Probability of best predictor for a branch changing over time

7.1.2 Stability Over Time

A program may change behavior several times throughout a program. For instance, a program can go through several stages, with different behavior in each. We here try to identify whether individual branches change their behavior in such a way that different predictors are best for predicting them at different times.

To identify how the behavior of a branch changes throughout a program, we split the lifetime of each static branch into regions of 100 branch executions, ignoring any end regions of less than 100 executions. For example, a branch that was executed 22,134 times is divided into 221 regions. For each two consecutive regions, we determined whether the best predictor for that branch changed. If the predictor that was best in the previous region was no longer best in the second region, we incremented a counter. A tie between the previous best predictor and a different predictor did not cause the counter to be incremented. The total number of changes divided by the total number of region pairs gives the frequency of change in best predictor for each branch.

Table 7.3 shows that the best predictor for a branch often changes throughout its lifetime. For the go benchmark, the best predictor changes between 35% of the regions. For four more benchmarks, the best predictor changes 9% or more of the time. Only three benchmarks have a probability of change of 2-4%. These are programs where usually only 1-2 of every 100 branches are mispredicted by either the Gshare or PAs predictors, so most of the results end in ties. For these benchmarks, it is mostly the case that *any* of the predictors would do, so there is no change during the lifetime of the branch. In conclusion, the branches that can be almost perfectly predicted generally do not change which predictor is best throughout the run of the program. For branches that are harder to predict, and can only be partially

predicted using any given type of information, the best predictor changes more frequently. The effect the changes in best predictor can have on prediction accuracy are examined in Section 7.2 which deals with selection mechanisms.

7.1.3 Stability Between Input Sets

Changing the input set for a program sometimes alters the way the program behaves. We here investigate whether the input set also alters which predictor is best for predicting a branch. This is important when considering profile based selection mechanisms for hybrid branch predictors.

For the experiment here, the best predictor for each branch was first determined using the profile set, and then using the regular testing set. We investigated how often the branches in the testing set were seen in the profile set, and whether they were best predicted using the same predictor as in the profile set. This information is presented in Figure 7.3. Each benchmark is represented by a pair of bars. There are two legends on the top of the graph. The leftmost legend refers to the leftmost bar in each pair. The rightmost legend refers to the rightmost bar in each pair. The leftmost bar corresponds to how often each predictor was most accurate for the testing set. The rightmost bar breaks down each of the categories from the leftmost bar showing whether the branches in that category were also represented in the profiling set, and if they were, whether the same predictor was best in the profiling set.

Those branches best predicted by the Two-Bit Counter predictor were the least volatile. Most of these are highly biased branches that, as shown in Section 4.1, remain biased regardless of input set. 88% of the branches best predicted using Two-Bit Counter for the testing set were also best predicted using that predictor for the profiling set. The branches best predicted using the PAs predictor were the next most likely to be best predicted using the same predictor for both data sets. Short history Gshare, the Loop predictor, and long history Gshare follow next. Gshare with a RHS is the most volatile, with only 57% of the branches best predicted using the same predictor for both data sets. Altogether, only 75% of the branches are best predicted by the same predictor for both data sets. Of the remaining branches, 4% were not found in the profiling set. This remains one of the problems for static selection mechanisms. Even if profiling sets are available, branch behavior does vary between input sets.

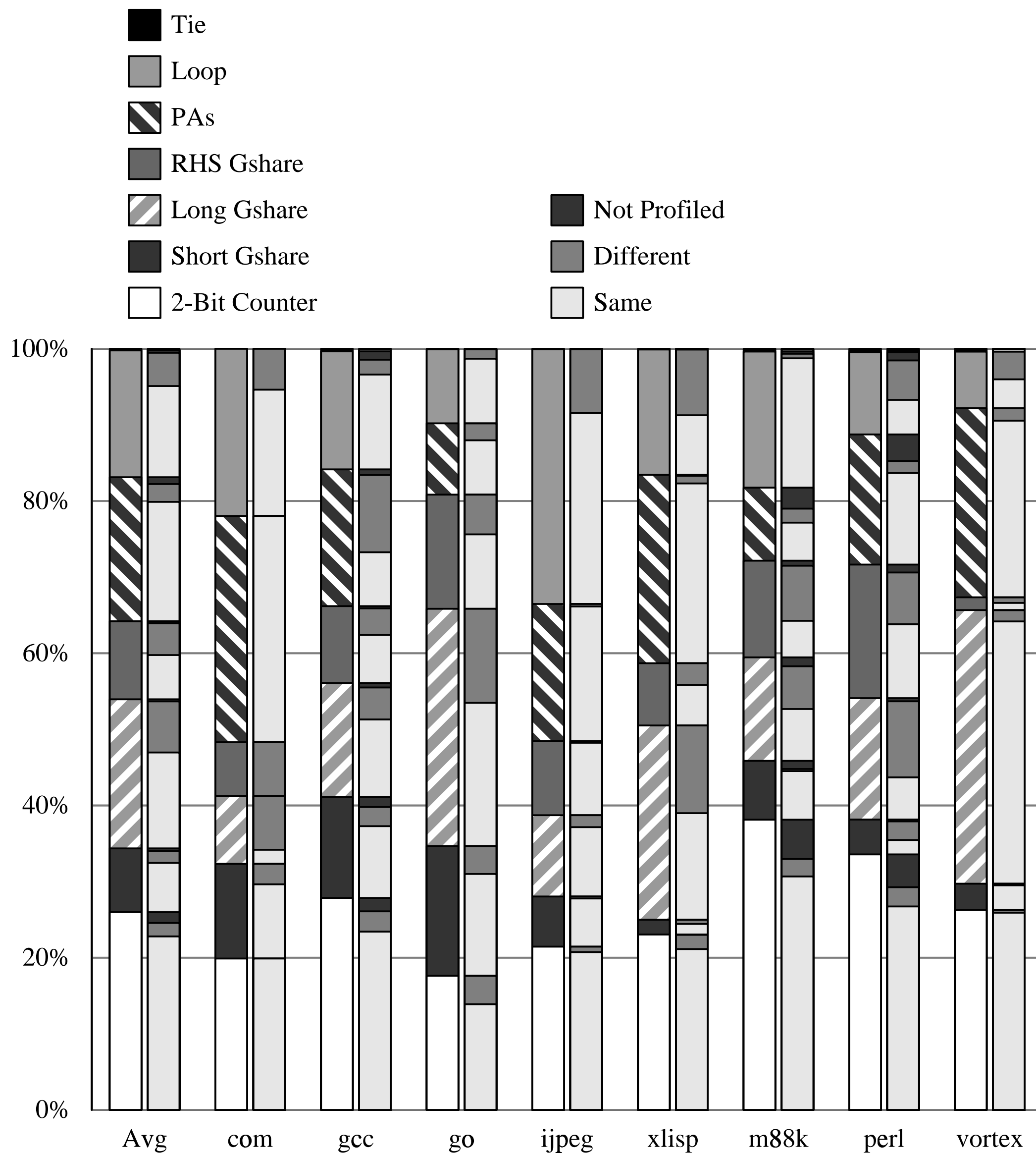


Figure 7.3: Change in best predictor between profiling and test runs

	Total	2bc	Short Gsh	Long Gsh	RHS Gsh	PAs	Loop
com	0.01%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
gcc	0.22%	0.02%	0.04%	0.07%	0.06%	0.02%	0.01%
go	0.69%	0.06%	0.11%	0.24%	0.22%	0.04%	0.03%
ijp	0.13%	0.01%	0.00%	0.03%	0.00%	0.00%	0.08%
xli	0.32%	0.00%	0.02%	0.20%	0.07%	0.01%	0.01%
m88k	0.07%	0.01%	0.01%	0.01%	0.00%	0.03%	0.01%
per	2.19%	0.07%	0.25%	0.82%	0.81%	0.22%	0.02%
vor	0.05%	0.00%	0.00%	0.03%	0.01%	0.01%	0.00%

Table 7.4: Increase in misprediction rate from selecting predictor using profiling

21% of all branches were best predicted by a different predictor in the two input sets. To further see the impact of this, we need to consider how much lower prediction accuracy these branches would have if we used the best predictor from the profiling set while running on the testing set. We therefore calculated the penalty from using the wrong predictor in a way similar to the calculation of the penalty from using the second best predictor in Section 7.1.1. For each of the regions where different predictors were best, the penalty is the fraction of all branches in the “Different” region multiplied by the difference in misprediction rate between the actual best predictor and the best predictor during the profiling run. The branches that were not in the profiling set were ignored, although these would generally add a further penalty if profiling is used for predictor selection.

Table 7.4 shows the penalty for each category and benchmark. In addition, the total penalty for each benchmark—the misprediction rate increase in percentage points from using the wrong predictor—is shown. The benchmarks that suffer the most from using the wrong predictor for some branches are perl and go. Most of the penalty overall was for branches that were best predicted using long history Gshare or Gshare with RHS.

Branches do not always remain best predicted using the same branch predictor when the data set changes. Most of the penalty from using the best predictor from the profiling set is for branches that are best predicted using a variation of the Gshare predictor. This is something that should be considered if using a profile based selection mechanism.

7.2 Selection Mechanisms

We here examine how the components in a hybrid predictor are selected. Real selection mechanisms are compared to idealized selection mechanisms to evaluate how well they work. We show the effect of using history in the selector as suggested in [4]. The effect of changing the size of the selector, which has mostly been ignored by previous studies, is also examined. Also, the best configurations for the selection mechanism for the PAs/Gshare hybrid are determined.

7.2.1 Basic and Idealized Selection Mechanisms

In this section, we try to identify how well selection mechanisms work. To study this, we examine several ways of selecting between 16 KB predictor components. Two idealized selection mechanisms are described, and compared to two implementable selection mechanisms.

The first selection mechanism is the McFarling selection mechanism, a table of counters indexed using the branch address. 3-bit counters were used instead of 2-bit counters as they work better. Their decision is more stable, so temporary changes in which predictor is best is less likely to change the selection prematurely. A 8 K entry selection mechanism, a typical size for this size hybrid, was used. The predictor using this selector is labeled "Dynamic 3-bit" in the following table. The second selection mechanism uses a profiling run to determine the best predictor for each branch. The best predictor for a branch during the profiling run is then used to predict the branch during the test run. If a branch was not seen during the profiling run, the Gshare predictor is used to predict it during the testing run. This predictor is labeled "Prof Static" in the following table.

The other two selection mechanisms are idealized. That is, they do not have an immediate counterpart that can be implemented. The first predictor is the same as the profile based static, although the same profile and test set is used. This is the best a static selector can do. This mechanism is labeled "Ideal Static" in the following table. The final mechanism splits the execution of each branch into regions of 100 branch executions as in Section 7.1.2. For each of these regions, the predictor which achieves the highest accuracy is used. This does not represent an actual mechanism, but estimates how well a selector would do if it could select the best predictor for every set of 100 executions of each branch. This

	Dynamic 3-bit	Prof Static	Ideal Static	Best per Region
com	4.61%	4.67%	4.67%	4.36%
gcc	5.17%	5.57%	5.21%	4.36%
go	12.88%	13.41%	12.88%	11.44%
ijp	4.66%	4.58%	4.50%	4.26%
xli	2.85%	3.04%	2.81%	2.69%
m88k	1.07%	1.21%	1.08%	1.02%
per	1.31%	2.20%	1.30%	1.22%
vor	0.63%	0.68%	0.63%	0.55%

Table 7.5: Misprediction rates of a PAs/Gshare hybrid predictor using two real and two idealized selection mechanisms

mechanism is labeled "Best per Region" in the following table. This selector was chosen as it, in the author's opinion, represents an achievable but challenging goal for the performance of a selection mechanism. We will see later that this is close to the performance we can achieve with large two-level selection mechanism.

It is hard to find a meaningful upper limit for how good a selection mechanism can be. An absolute upper limit is a mechanism that always selects the correct predictor if such a predictor exists. However, this gives an upper bound that is unreachable and unhelpful, as one predictor may just be accidentally correct. Consider the case of having the two predictors Always Taken and Always Not Taken. A perfect selection mechanism achieves a 0% misprediction rate. This shows that the Always Taken and Always Not Taken predictors are perfectly matched, that is, one of the predictors is always right. However, this does not say how well a predictor selecting between these is likely to do. In light of this, the "Best per Region" mechanism was created to be a more helpful measure. However, it is not an upper bound.

Table 7.5 shows the branch misprediction rates of PAs/Gshare using the four selection mechanisms described above. The dynamic selection mechanism is better, often much better, than the profiled static mechanism for 7 of 8 benchmarks. The dynamic selection mechanism performs almost identically to the ideal static mechanism. This highlights the problem with static predictor selection mechanisms. Even in the ideal case, they only do as well as a dynamic selector. When profiling is considered, selecting dynamically works much better than selecting statically.[4]

[4]This is from the perspective of how accurate the selection is. However, static selection has other benefits, such as being able to disable component predictors that are not being used.

	Dyn Six Comp	Prof Static	Ideal Static	Best per Region
com	4.75%	4.66%	4.65%	4.22%
gcc	4.18%	4.66%	4.11%	3.44%
go	11.13%	11.75%	11.05%	9.36%
ijp	4.24%	4.12%	3.99%	3.70%
xli	2.83%	3.02%	2.69%	2.46%
m88k	0.57%	0.72%	0.56%	0.52%
per	0.98%	3.43%	0.97%	0.88%
vor	0.56%	0.61%	0.55%	0.47%

Table 7.6: Misprediction rates of a six-component hybrid predictor using two real and two idealized selection mechanisms

As we take the idealized selection mechanisms one step further with the "Best per Region" mechanism, the misprediction rates drop to 3.73% on average compared to 4.13% for the ideal static and 4.14% for the dynamic selector. However, as we shall see in Section 7.2.2, a larger selector using dynamic history can bridge most of this gap at the cost of extra hardware.

We also consider the case where we have six predictors to choose from. These are the same six predictors that have been used throughout this chapter. The dynamic McFarling selector can only be used to select between two predictors, so we instead use the selector from Section 6.6.2. The dynamic selection mechanism has 8 K entries. This selector is labeled "Dyn Six Comp" in the following table. The profiled static and idealized selection mechanisms remain the same as before, but now have six components to choose from.

Table 7.6 shows the misprediction rates of six component hybrid branch predictors using each of the four selection mechanisms. The dynamic selection mechanism is still better than profiled selection for six of the benchmarks. However, the ideal static mechanism is now always a little better than the dynamic mechanism due to interference effects. When choosing between 6 components, interference in the selector is more likely to be destructive. If two branches share the same entry in a selection mechanism, they are 50%[5] likely to be best predicted using the same component in a 2-way hybrid, but only 17% likely to be best predicted using the same component in a 6-way hybrid.

Going from "Ideal Static" to the "Best per Region" mechanism, the misprediction rate drops to 3.13% compared to 3.57% for the ideal static and 3.65% for the dynamic selector.

[5]Assuming all components are equally likely to be the best predictor for a branch.

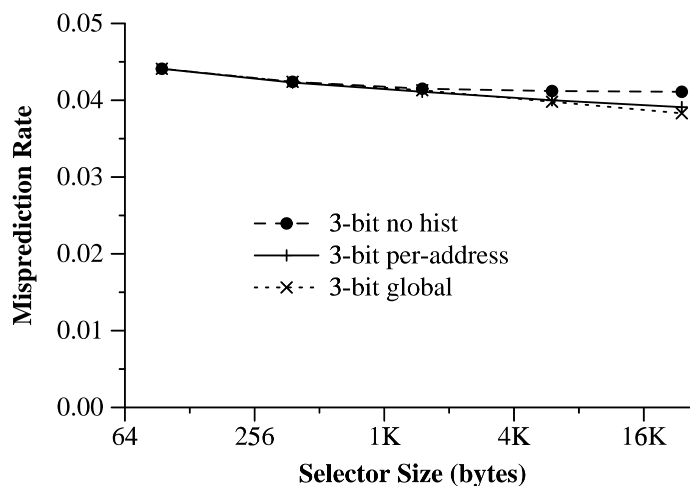


Figure 7.4: PAs/Gshare selection mechanisms

However, this gap can be covered for the six component hybrid by using dynamic history and a larger selector.

7.2.2 Size and History in Predictor Selectors

Two factors that affect the performance of a hybrid selection mechanism are studied here. One is the size of the table used for the selection mechanism, and the other is the use of history in indexing that table. Three schemes for indexing into the selection table are studied: using bits from the branch address, using a Gshare type index (gXOR in [4]), and using a Pshare type index (pXOR in [4]). The size of the table is varied for each of the selector types. We consider selection mechanisms for both a PAs/Gshare hybrid and a six component hybrid. Each of the components is 16 KB.

The PAs/Gshare hybrid is considered first. The index, either address-only, global, or per-address, is used to index into a table of 3-bit counters. The chosen 3-bit counter is then used for selection. 2-bit counters were also considered, but as they performed slightly worse than 3-bit counters at similar costs, the results using 3-bit counters are emphasized.

Figure 7.4 shows the misprediction rates of a PAs/Gshare hybrid using each of the three selection mechanisms for sizes between 96 B and 24 KB. The cost of the per-address histories for the per-address selector are not included in the size as they are already accounted for by the PAs predictor. The cost of the component predictors are also not included in this figure.

For selection mechanisms larger than 384 bytes the selection mechanisms using global

index size	Per-Address		Global	
	2-bit	3-bit	2-bit	3-bit
8	0	0	1	0-1
10	0-1	1	0	0
12	2	2-3	3	3,5
14	4	4	9,11,13	7,9
16	12-16	6-7,11-13	14-16	13-14

Table 7.7: Best history lengths for PAs/Gshare selectors

history work best. The mechanism using global history works better than the mechanism using per-address history for all sizes. Whereas the size of the selector does not matter past 1.5 KB for the selector using only the branch address, the selector using a global index still shows growth at 24 KB. With a 6 KB selector, a PAs/Gshare using a address-only selector has a misprediction rate of 4.12%, but when using a global-history selector the misprediction rate is only 3.98%. For a 24 KB selector, the misprediction rate using a address-only selector is almost unchanged at 4.11% while the misprediction rate of the one using a global selector drops to 3.83%.

Considering that a hybrid predictor can be improved either by increasing the size of the components or the selection mechanism, a 6 KB selector using global history turns out to be the optimal choice for this size PAs/Gshare hybrid.

Referring back to the "Best per Region" idealized selector in Table 7.5, we see that for the 24 KB selector, the misprediction rate is only 0.1 percentage points higher than using the "Best per Region" selector. If the size of the selection mechanism could be reduced, dynamic selectors could attain this limit.

The best number of history bits to use in selection mechanisms of varying sizes is given in Table 7.7. The size of the selector is given as the number of bits in the index. For 2-bit selectors, the size in bits is 2×2^{ibits} where $ibits$ is the number of bits in the index. For 3-bit selectors, the size is 3×2^{ibits}. As the selector gets larger, more history is used in the selector as there is less interference. For some sizes, several history lengths or ranges are given. In these cases, each of these performed equally well.

We now consider the effect of the size of the selection mechanism and the use of history information in the selection mechanism for a six component hybrid. The indexing schemes are the same as for the PAs/Gshare hybrid. However, the index is used to select a set of six counters using the selection mechanism from Section 6.6.2. The counters used in the

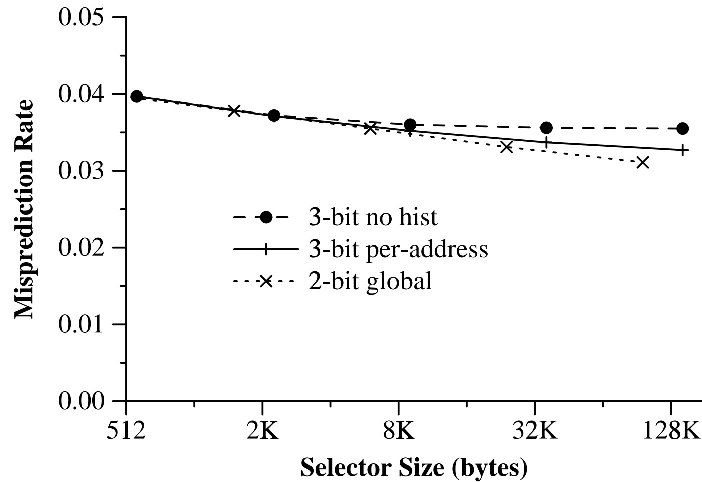


Figure 7.5: Six component selection mechanisms

selection mechanism were 3-bit counters for the address-only and per-address indexes, and 2-bit counters for the global index. The size of counter has only marginal effect for the six component hybrid predictor.

Figure 7.5 shows the misprediction rates of a six component hybrid using each of the three types of indexing for the selection mechanism. The sizes covered are 512 B to 192 KB. This represents the same number of entries as in the previous figure, but the cost of each entry is multiplied by 6. Once again, global history works better than per-address history. Size, in terms of entries, matters slightly more than for the two component hybrid. This is likely due to interference effects. Given that the cost per entry is 6 times higher for the six component hybrid, size is a very limiting factor for this type of selection mechanism.

If very large selectors are used, the misprediction rate can actually be better than using the "Best per Region" idealized selector. At 128 KB, the misprediction rate is 3.11% using a global history selector compared to 3.13% for the "Best per Region" selector. To make hybrid predictors with a large number of components more attractive, it is important to find a way to reduce the cost of the selection mechanism.

index size	Per-Address 2-bit	Per-Address 3-bit	Global 2-bit	Global 3-bit
8	0	0	0-1	1
10	1	1	1,3	3
12	2	2	5-6	4-6
14	4	4	9	9
16	6	5-6	13-14	13

Table 7.8: Best history lengths for six component hybrid selector

The best number of history bits to use in selection mechanisms of varying sizes is given in Table 7.8. The size of the selector is given as the number of bits in the index. For 2-bit selectors, the size in bits is $6 \times 2 \times 2^{ibits}$ where $ibits$ is the number of bits in the index. For 3-bit selectors, the size is $6 \times 3 \times 2^{ibits}$. As the selector gets larger, more history is used in the selector as the effects of interference subside.

Since PAs/Gshare is the leading hybrid branch predictor, we also determined the best selector size and configuration for each size hybrid. At any size, the predictor can be improved either by increasing the size of the PAs and/or Gshare components, or by increasing the size of the selection mechanism. We found by experimentation that it is best to increase the size of both the PAs and Gshare predictors simultaneously. For each size, the size of the selection mechanism was increased until the accuracy improvement per byte of enlarging the selection mechanism was smaller than the improvement per byte from enlarging the PAs and Gshare components. The optimal selection mechanism size and history length for each hybrid predictor size is given in Table 7.9.

One curious result in this table, is that the optimal history length for the selection mechanism remained almost constant at 9 even as the size of the selection mechanism increased. This is seemingly at odds with the conclusions in Table 7.7. However, the one point that is in common between the two tables (index size 14, 3-bit global selector in

Size of PAs/Gshare	Size of Selector	History Bits in Selector
15 KB	3 KB	10-11
26 KB	6 KB	9
42 KB	6 KB	9
80 KB	12 KB	9
157 KB	24 KB	9
309 KB	48 KB	8-11

Table 7.9: Best sizes and history lengths for PAs/Gshare selectors

Table 7.7; and 42 KB hybrid, 6 KB selector in Table 7.9), shows that the best history length for that size is 9. Unlike in Table 7.7, the size of the hybrid predictor changes with the size of the selector in Table 7.9, and this is probably the cause of this curious effect.

A possible explanation for why the history length of the selector remains the same as the size of the hybrid is increased involves two factors. First, small Gshare predictors suffer from a large amount of interference. As the predictor gets larger, the amount of interference is reduced. Second, by using global history in a selector, the selector can learn to avoid Gshare for the patterns that are likely to be affected by interference.

For small PAs/Gshare hybrids, the Gshare component experiences a large amount of interference. Aggressively using a large amount of history, 9 of 14 bits, in the selector allows us to avoid using Gshare for the patterns that suffer the most from interference. As the component predictors get larger, there is less interference in the Gshare predictor. It becomes less important to use global history in the selection mechanism to avoid interference, and more important to accurately select the best component for a given branch. Therefore, the tradeoff between using more history and reducing interference in the selection mechanism itself leads to the history length remaining almost constant while the selector size is increased.

7.3 Improving Prediction using a Multiple Component Hybrid Branch Predictor

In this section, we examine how we can use the information about predictor interaction and selection presented in this chapter to improve hybrid branch prediction. A few of the conclusions reached in this chapter are particularly relevant for how to build the best hybrid branch predictor.

First, we showed in Section 7.1.1 that it is important for a hybrid predictor to have both a PAs component, and a Gshare component using a long history. We also showed that a Loop component and a Gshare using a short history are likely to be cost effective additions to a hybrid branch predictor.

Second, we showed in Section 7.1.2 that the best predictor for a branch changes during the run of the program. In Section 7.1.3 we also showed that the best predictor for a branch changes between input sets. This indicated that it is probably best to use a dynamic

selection mechanism to select which component predictor to use for each branch.

Third, we showed in Section 7.2 that the two-level selection mechanisms proposed in [4] work better than other known selection mechanisms, and should therefore be used for hybrid branch predictors.

The principle of a hybrid predictor is that there are several component predictors, and each branch (or subset of a branch's lifetime) is predicted by the component that is best suited for that branch. Throughout this chapter, it has been evident that there are potential benefits of having more than two component predictors in a hybrid predictor. We therefore propose a Multiple Component Hybrid Branch Predictor, or Multi-Hybrid for short. The Multi-Hybrid presented here has four component predictors, and a dynamic two-level selection mechanism do decide which component to use for each branch.

The Multi-Hybrid presented here is a refined version of the predictor we introduced in [9] when we first used the name Multi-Hybrid. The main change is in changing the structure of the selection mechanism from the old type, which was similar to the one presented in Section 6.6.2, to an approach less susceptible to interference. A smaller change is the combination of component predictors used. The combination of component predictors selected through the analysis in this chapter works better than the combination used in the previous version.

7.3.1 Implementation

The Multi-Hybrid presented here uses two Gshare components, one with short and one with long history. Gshare with a Return History Stack was not included, as no configurations tested showed sufficient improvements from adding this component. The Multi-Hybrid also uses the PAs and Loop predictors as components.

The Multi-Hybrid is formed as two separate hybrid predictors, a Dual History Length (DHL) Gshare with selective update and a PAs/Loop predictor. The DHL Gshare was explained in Section 6.6.1 and was shown in Figure 6.14. The PAs/Loop hybrid is similar to a PAs/Gshare hybrid using a two-level selection mechanism, with the one exception that the Gshare predictor is replaced by the Loop predictor. Each of the two hybrid predictors, DHL Gshare and PAs/Loop, is considered a component in the Multi-Hybrid.

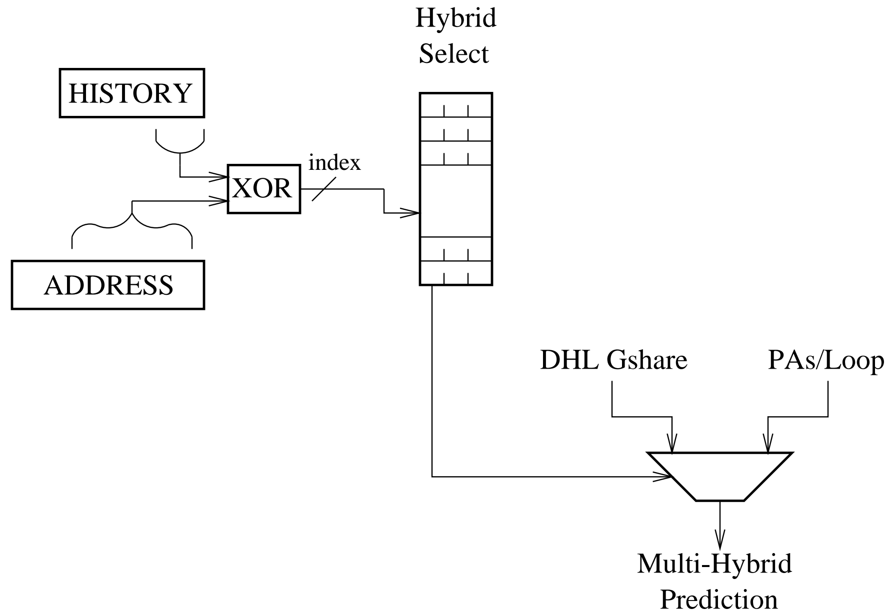


Figure 7.6: Multi-Hybrid

Figure 7.6 shows a diagram of the Multi-Hybrid predictor. Each cycle, all of the component predictors make a prediction, and the selection mechanism decides which prediction to use. The selection mechanism in the DHL Gshare hybrid determines whether a short or a long history is better for the current branch. The selection mechanism in the PAs/Loop hybrid determines whether a PAs or Loop predictor is better for the branch. Finally, the outer selector shown in Figure 7.6 determines whether branch correlation (DHL Gshare) or self correlation (PAs/Loop) is better for the branch. This selection is done using a two-level hybrid selector as shown. As in the DHL Gshare predictor, the small Gshare component can be used to generate a quick prediction if the entire predictor does not fit in the cycle time. If the full prediction of the Multi-Hybrid is different from that of the small Gshare component, the front-end of the processor is redirected later with a one or two cycle penalty.

Extensive experimentation was performed to determine how much hardware to devote to each of the component predictors and to the selection mechanisms. However, it is impossible to guarantee that the configurations given here are optimal. The original premise, supported by the studies earlier in this chapter, was that both the PAs and long history Gshare predictors were both very important in a hybrid predictor. These were therefore allocated a nearly equal amount of the hardware budget. The short history Gshare suffers from less interference, so it does not need to be as large. The size of the Loop predictor remains almost constant, varying only slightly as the size of the loop counter is increased. After setting

Size	Short HL Gsh + Update Tab	Long HL Gshare	DHL Gshare Selector	PAs	Loop Counter	Other Selectors
	(HL,PHTs)	(HL,PHTs)	(HL,PHTs)	(HL,PHTs)	Size	(HL,PHTs)
18 KB	(1,1024)	(14,1)	(1,1024)	(14,1)	7 bits	(4,128)
30 KB	(2,1024)	(15,1)	(2,1024)	(15,1)	7 bits	(7,32)
53 KB	(3,1024)	(16,1)	(3,1024)	(16,1)	7 bits	(8,32)
98 KB	(3,2048)	(17,1)	(3,2048)	(17,1)	9 bits	(9,32)
188 KB	(4,2048)	(18,1)	(4,2048)	(18,1)	9 bits	(12,8)
368 KB	(4,4096)	(19,1)	(4,4096)	(19,1)	9 bits	(13,8)

Table 7.10: Configurations for Multi-Hybrid

the initial sizes, the sizes and configurations of all component and the selection mechanisms were varied using an ad hoc hill-climbing algorithm. At each point, a large number of small changes to the sizes and configurations were considered. If a better configuration was found, it would be the basis for the next round and a number of new small changes were once again considered. As pointed out earlier, there is no guarantee that the resulting configuration is optimal, as there may be local maximums.

The best configurations that were found are listed in Table 7.10. For all selectors and components other than PAs, the history used is global. The first three columns show the configuration of the DHL Gshare part. These configurations are identical to those given in Table 6.8. The fourth column shows the configuration of the PAs predictor. A PAs with a single PHT was always used. The fifth column shows the size of the counter used by the Loop predictor. Increasing this size has a small effect on the misprediction rate, but only marginally changes the size of the Multi-Hybrid. For predictors of 98 KB and up, the advantage of a larger loop counter marginally offsets the extra cost. The final column shows the configuration of both the selector for the PAs/Loop hybrid and the overall selector for the Multi-Hybrid.[6]

7.3.2 Results

The misprediction rate of the Multi-Hybrid is compared to the misprediction rate of a PAs/Gshare hybrid using a two-level selection mechanism. PAs/Gshare is a very accurate hybrid branch predictor, and provides a high standard to compare to. The configurations used for the PAs/Gshare hybrid are the optimal configurations given earlier. Figure 7.7

[6]Due to a limitation of my simulation environment, only Multi-Hybrid configurations that had the same configuration for the PAs/Loop selector and the overall selector were considered.

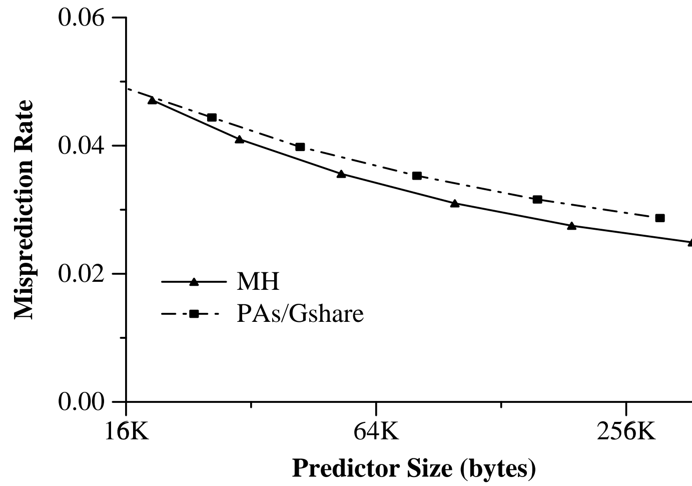


Figure 7.7: PAs/Gshare vs. Multi-Hybrid

shows that the Multi-Hybrid generally outperforms the PAs/Gshare predictor. Figure 7.8 compares the Multi-Hybrid to the PAs/Gshare predictor for each benchmark.

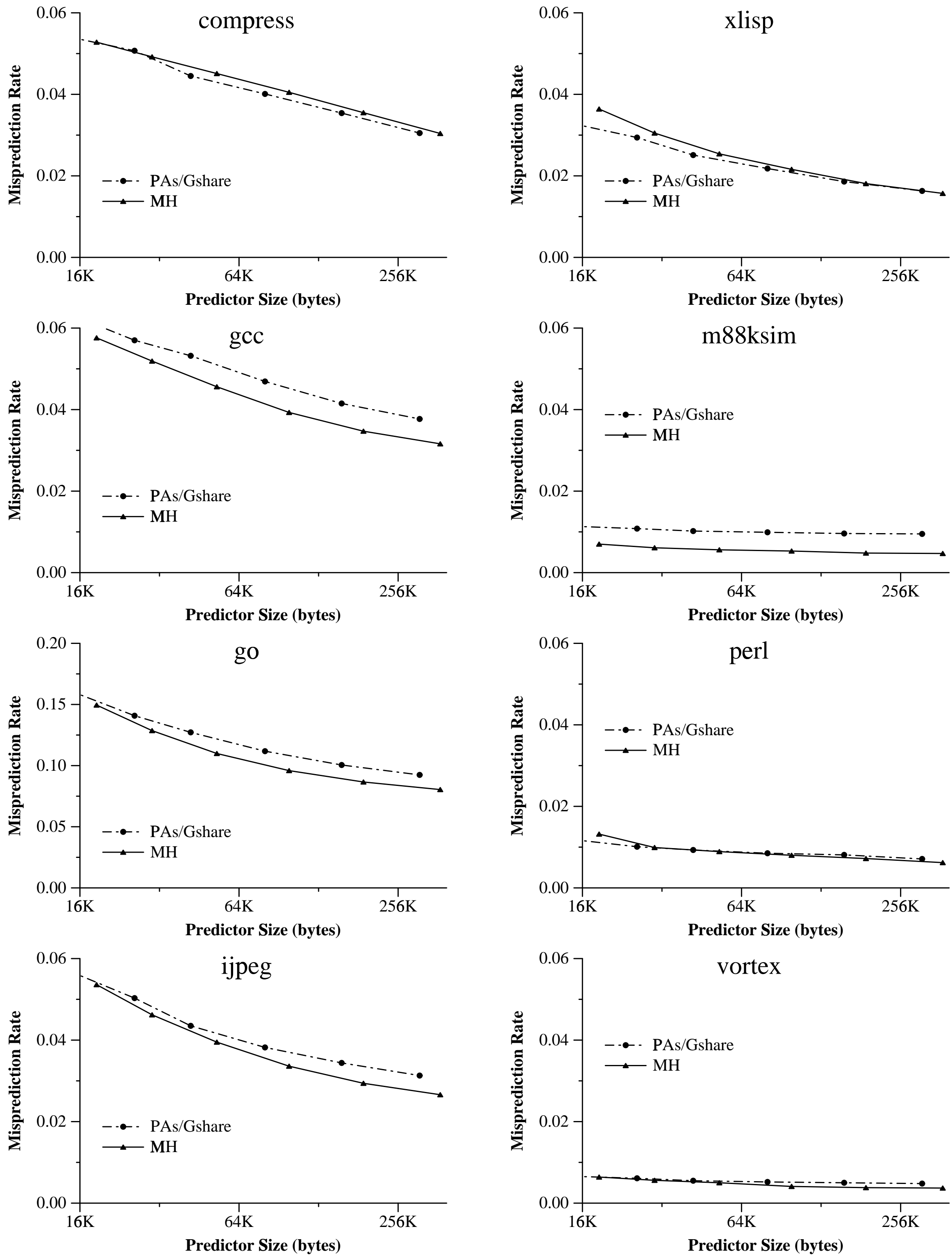


Figure 7.8: PAs/Gshare vs. Multi-Hybrid by benchmark

For the smallest size studied, 18 KB, the Multi-Hybrid is only marginally better than the PAs/Gshare predictor. However, as the predictor size increases, the advantage of the Multi-Hybrid over PAs/Gshare increases. At 54 KB, the Multi-Hybrid has a misprediction rate of 3.56% compared to 3.82% for PAs/Gshare. This means that the Multi-Hybrid has 7% fewer mispredictions. For a 188 KB predictor, the Multi-Hybrid has a misprediction rate of 2.75%, which is 11% fewer mispredictions than PAs/Gshare.

The Multi-Hybrid does include more components than the PAs/Gshare predictor, but the implementation complexity is not much higher since per-address history is already supported in PAs/Gshare.

7.4 Summary

In studying the importance of six single scheme predictors for use in hybrid branch prediction, we found that the Gshare predictor using a long history and the PAs predictor were clearly the most important. A Gshare predictor using a shorter history and a Loop predictor were found to be useful, especially given their lower cost. A Gshare predictor using a Return History Stack, was found to be useful for some branches, but no hybrid predictor implementation was found in which the cost of adding this component was justified. A table of two-bit counters is not needed if either a short history Gshare, a PAs, or a Loop predictor is available. In all, 40% of the branches were best predicted using branch correlation, 35% of the branches were best predicted using self correlation, and the remaining branches were equally well predicted using a simple Two-Bit Counter predictor.

Dynamic selection of components in a hybrid branch predictor was found to be inherently better than static selection, even better than ideal static selection. In addition, static selection suffers from problems if the profile used is not representative of the testing input. A two-level dynamic selection mechanism utilizing a global history, such as the one introduced in [4], was shown to provide a substantial improvement over mechanisms using address only. However, it was shown that the two-level mechanisms are far more sensitive to size than the address only mechanisms, especially when choosing between more than two component predictors. Based on this we anticipate that interference reduction in two-level selection mechanisms will be important in the future. Since the Multi-Hybrid relies more on selection mechanisms than normal hybrids, it is likely to benefit more from such techniques.

Finally, the Multi-Hybrid predictor combining both short and long history Gshare components with PAs and Loop predictors was introduced. The Multi-Hybrid was shown to be better than the PAs/Gshare predictor for all sizes investigated, but the improvement was most substantial for the largest predictors. On average, the Multi-Hybrid has 7 to 11% fewer mispredictions compared to PAs/Gshare for 54 to 188 KB predictors.

CHAPTER 8

Conclusions

8.1 Contributions

In this dissertation correlation and branch execution patterns were examined to contribute to a better understanding of how branches behave and how they can be predicted. We classified and quantified branch behavior, and showed that some of this behavior was not captured by existing predictors. We proposed several new predictor designs to take advantage of the behavior we saw. These predictors are more accurate than similar existing predictors.

By classifying branch behavior, we showed that:

- Two thirds of all branches follow highly predictable patterns.
- Transient patterns account for less than 20% of all branches, but use most of the resources of a per-address two-level predictor.
- Correlation with only a few branches is sufficient for branch correlation based prediction.
- For 20% of all branches, the most correlated branches have been flushed from the history due to loops or subroutines between the correlated branch and the current branch.
- A PAs and a Gshare component using a long history are the most important components to include in a hybrid branch predictor. A Loop predictor and a Gshare using a short history are likely to be cost effective components to include in a hybrid branch predictor.

- Dynamic selection mechanisms are inherently better than static selection mechanisms for hybrid branch predictors.

In Section 5.1, we examined branch execution patterns, and found that two thirds of all branches follow highly predictable patterns and that these branches can easily be identified dynamically. Stable biased, and other high confidence repeating patterns that account for 49% of all branches are over 99.9% predictable. Other branches that have repeated the same outcome 30 or more times account for another 17% of all branches and are over 99% predictable. This adds to the previously known categories of highly predictable branches. The idea of using a separate predictor for these branches, so that other prediction resources can be used for hard-to-predict branches is appealing.

In Section 5.2, we examined per-address pattern use in terms of the limited history seen by a regular per-address predictor, such as PAs. We found that patterns that were transient in the limited history accounted for less than 20% of all branches, but these patterns use 99% of the counters in a 16 KB predictor and 99.9% of the counters in a 256 KB predictor. Even within the transient patterns, the usage was skewed towards a smaller number of the counters. This means that most of the pattern history table of a per-address predictor is used very sparsely.

In Section 6.5, we examined the nature of branch correlation, that is, correlation between different branches. We showed that most branches do not need a large amount of history. For most branches, correlation with fewer than 3 previous branches was needed. The leading correlation based predictor, Gshare, using a pattern history of 16 branches, was often unable to capture the direct correlation with 1, 2, or 3 branches. We introduced a hypothetical predictor that can capture this correlation fully, and used this to develop an implementable predictor that captures most of the correlation.

In Section 6.5, we also showed that the most correlated branches are often removed form the history due to branches in subroutines or loops between the correlated branch and current branch. These branches are close together in the code, but farther apart in the dynamic instruction stream. About 20% of all branches can be better predicted using history that includes the outcomes from before subroutines and loop-bodies.

In Section 7.1, we examined the interaction between component predictors in hybrid predictors. We found that a Gshare predictor using a long history and a PAs predictor were the two most important. A Gshare predictor using a shorter history and a Loop predictor

were found to be useful, especially given their lower cost. A Gshare predictor using a Return History Stack, was found to be useful for some branches, but no hybrid predictor implementation was found in which the cost of adding this component was justified. A table of two-bit counters is not needed in a hybrid predictor if either a short history Gshare, a PAs, or a Loop predictor is available.

In Section 7.2, we found that dynamic selection mechanisms for hybrid branch predictors are inherently better than static selection mechanisms, even under ideal circumstances. In addition, static selection suffers from problems if the profile used is not totally representative of the testing input. A two-level dynamic selection mechanism utilizing a global history, such as the one introduced in [4], was shown to be a substantial improvement over mechanisms using address information only. However, it was shown that the two-level mechanisms are far more sensitive to size than the address only mechanisms, particularly when choosing between more than two-component predictors. Based on this size sensitivity we anticipate that interference reduction in two-level selection mechanisms will be a source of improvement in the future.

Some of the behavior that we found was not exploited by current predictors. To exploit this behavior, we introduced several new prediction mechanisms:

- The loop filtering mechanism dynamically identifies highly predictable biased and repeating patterns. For these highly predictable patterns, the loop predictor is used in place of the main predictor. The loop filtering mechanism, when applied to Gshare, reduces the number of mispredictions by 15-23%.

- The Dual History Length Gshare predictor with Selective Update uses two Gshare components with different history lengths. The two versions of this predictor reduce the number of mispredictions by up to 14% compared to Gshare and up to 10% compared to a filtered Gshare.

- The Multi-Hybrid predictor uses the four component predictors that were found to be most important in Chapter 7. The Multi-Hybrid reduces the number of mispredictions by 7 to 11% compared to a PAs/Gshare hybrid.

In Section 5.5, we introduced the loop filtering mechanism, an improvement over the branch filtering mechanism. High confidence for-type, while-type, alternating and strongly biased patterns are predicted using the loop predictor, while other branches are predicted

using the main predictor. Using this mechanism, the number of mispredictions suffered by the Gshare predictor is reduced by 15-23% depending on the size, with a 7-8% improvement over a standard filtering mechanism. The number of mispredictions suffered by a state of the art PAs/Gshare hybrid is reduced by 6-7%, with a 3-5% improvement over the standard filtering mechanism. The improvements over the standard filter increases for larger sizes.

In Section 6.6, we introduced two versions of Dual History Length Gshare predictor with Selective Update. One with and one without a mechanism to keep history from before subroutine calls. These predictors reduce the number of mispredictions by up to 14% compared to Gshare and up to 10% compared to a filtered Gshare. The complexity of these predictors is a little higher than that of Gshare. However, both predictors are less complex than a filtered Gshare predictor as there is no need for a BTB access or tag match on the prediction path.

In Section 7.3, we introduced the Multi-Hybrid predictor which combines both short and long history Gshare components with PAs and Loop predictors. The Multi-Hybrid is better than the PAs/Gshare predictor for all sizes investigated, but the improvement is most substantial for 54 KB or larger predictors. On average, the Multi-Hybrid mispredicts 7-11% fewer branches compared to PAs/Gshare for 54 to 188 KB predictors.

8.2 Future Directions

In this dissertation we showed several ways to improve branch prediction. However, the best predictor presented here mispredicts 2.5% of all branches even at high implementation costs. To improve predictors further, we should pay more attention to: interference in predictors and selection mechanisms, methods for identifying which predictor to use without having to update all predictors for each branch, and ways to more explicitly exploit correlation between branches.

Reducing interference makes a predictor more cost effective. If interference is brought down to a low level in two-level predictors, more history can be hashed to fit in an index. That is, a 20-bit history pattern could be compressed into 16 bits if there is little chance of conflict in the pattern history table. The best methods for both reducing the interference and compressing these patterns still need to be found. Reducing interference will also help make each component in a hybrid predictor smaller, so that more component predictors can

be included.

Reducing interference in the selection mechanisms for hybrid predictors is also important. Two-level selection mechanisms are as much affected by interference as two-level predictors are, yet there is no research on how to limit this interference. The problem is magnified for hybrid predictors with more than two components, as the cost of the selection mechanism is higher.

Identifying which component in a hybrid predictor to use without having to update all components is likely to be important in future hybrid predictors. If all components are updated for all branches, there is unnecessary duplication of information. However, only the components that are being updated can effectively predict the branch, so a balance must be found. As an example, consider the loop predictor we used in two of the predictors proposed in this dissertation. In the first, the loop filter, the loop predictor dynamically detects the patterns for which it should be used, and takes over the prediction for those. The main predictor is not updated for these patterns. In the Multi-Hybrid, the loop predictor is included as just another component. The best solution is likely somewhere in between. The loop predictor should by itself detect those patterns for which it is extremely accurate and filter those out from the rest of the predictors. However, for some of the less accurate patterns, the loop predictor should still be available as a normal component subject to normal selection. Similarly, other components should be able to detect when they are accurate enough to filter out a branch from the rest of the predictor.

A predictor that can directly use the correlation between branches could possibly achieve much better prediction accuracies than regular two-level predictors, and would at the same time be less susceptible to both warm-up time and interference. Such a predictor could be used as the only correlation based predictor, or as a supplement to Gshare or GAs. Developing such a predictor could improve prediction significantly.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] B. Calder, D. Grunwald, and J. Emer, "A system level perspective on branch architecture performance," in *Proceedings of the 28th Annual ACM/IEEE International Symposium on Microarchitecture*, pp. 199–206, 1995.

[2] P.-Y. Chang, *Classification-Directed Branch Predictor Design*, PhD thesis, University of Michigan, Ann Arbor, 1997.

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

[4] P.-Y. Chang, E. Hao, and Y. N. Patt, "Alternative implementations of hybrid branch predictors," in *Proceedings of the 28th Annual ACM/IEEE International Symposium on Microarchitecture*, pp. 252–263, 1995.

[5] P.-Y. Chang, E. Hao, T.-Y. Yeh, and Y. N. Patt, "Branch classification: A new mechanism for improving branch predictor performance," in *Proceedings of the 27th Annual ACM/IEEE International Symposium on Microarchitecture*, pp. 22–31, 1994.

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

[7] I.-C. K. Chen, J. T. Coffey, and T. N. Mudge, "Analysis of branch prediction via data compression," in *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, 1996.

[8] A. N. Eden and T. Mudge, "The YAGS branch prediction scheme," in *Proceedings of the 31th Annual ACM/IEEE International Symposium on Microarchitecture*, pp. 69–77, 1998.

[9] M. Evers, P.-Y. Chang, and Y. N. Patt, "Using hybrid branch predictors to improve branch prediction accuracy in the presence of context switches," in *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pp. 3–11, 1996.

[10] A. Farcy, O. Temam, R. Espasa, and T. Juan, "Dataflow analysis of branch mispredictions and its application to early resolution of branch outcomes," in *Proceedings of the 31th Annual ACM/IEEE International Symposium on Microarchitecture*, pp. 59–68, 1998.

[11] E. Federovsky, M. Feder, and S. Weiss, "Branch prediction based on universal data compression algorithms," in *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pp. 62–72, 1998.

[12] 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.

[13] D. Grunwald, D. Lindsay, and B. Zorn, "Static methods in hybrid branch prediction," in *Proceedings of the 1998 ACM/IEEE Conference on Parallel Architectures and Compilation Techniques*, 1998.

[14] Q. Jacobson, E. Rotenberg, and J. E. Smith, "Path-based next trace prediction," in *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, 1997.

[15] T. Juan, S. Sanjeevan, and J. J. Navarro, "Dynamic history-length fitting: A third level of adaptivity for branch prediction," in *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pp. 155–166, 1998.

[16] D. Kaeli and P. Emma, "Branch history table predictions of moving target branches due to subroutine returns," in *Proceedings of the 18th Annual International Symposium on Computer Architecture*, 1991.

[17] C.-C. Lee, *Optimizing High Performance Dynamic Branch Predictors*, PhD thesis, University of Michigan, Ann Arbor, 1997.

[18] C.-C. Lee, I.-C. K. Chen, and T. N. Mudge, "The bi-mode branch predictor," in *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, pp. 4–13, 1997.

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

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

[21] P. Michaud, A. Seznec, and R. Uhlig, "Trading conflict and capacity aliasing in conditional branch predictors," in *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pp. 292–303, 1997.

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

[23] 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.

[24] 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.

[25] 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.

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

[27] E. Sprangle, R. S. Chappell, M. Alsup, and Y. N. Patt, “The agree predictor: A mechanism for reducing negative branch history interference,” in *Proceedings of the 24th Annual International Symposium on Computer Architecture*, 1997.

[28] *Welcome to SPEC*, The Standard Performance Evaluation Corporation. http://www.specbench.org/.

[29] J. Stark, M. Evers, and Y. N. Patt, “Variable length path branch predcition,” in *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 170–179, 1998.

[30] A. R. Talcott, M. Nemirovsky, and R. C. Wood, “The influence of branch prediction table interference on branch prediction scheme performance,” in *Proceedings of the 1995 ACM/IEEE Conference on Parallel Architectures and Compilation Techniques*, 1995.

[31] T.-Y. Yeh, *Two-Level Adaptive Branch Prediction and Instruction Fetch Mechanisms for High Performance Superscalar Processors*, PhD thesis, University of Michigan, Ann Arbor, 1993.

[32] 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.

[33] 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.

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

[35] 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.