Delay-Sensitive Branch Predictors for Future Technologies

DRAFT

Daniel A. Jiménez, B.S., M.S.

March 22, 2002

Note: The compact format for this draft version of the dissertation is motivated by the need to save paper and have a document that can be stapled and carried easily in a stack of papers. A version using the official UT dissertation format is available to committee members upon request.

Acknowledgments In my research, I have received assistance from many people.

Calvin Lin, an assistant professor in Computer Sciences, is my supervising professor. He has been deeply involved in guiding my research and is a co-author on all of my recent publications. Many of the ideas presented in this dissertation were formed in discussions with Calvin, and many of the successful ideas I came up with on my own were ideas I almost gave up on when Calvin said "let's take another look at this." Calvin has made it his responsibility to make sure that I, as well as all of his other students, have had the financial support we need to accomplish our goals; he is a staunch advocate for each of us, and although he leads a busy life, it seems he always has time to meet with us and discuss whatever is on our minds; we are all thankful for him.

The members of my dissertation committee are: Doug Burger, Steve Keckler, Calvin Lin, Hugh Maynard, Kathryn McKinley and Yale Patt. Each of them has made invaluable contributions to this dissertation through discussions, feedback on my writings, and the proposal process.

Steve Keckler, an assistant professor in Computer Sciences, has contributed several distinctive insights to my research through many fruitful discussions as well as actual work on my ideas. It was his idea to study how technology scaling will affect branch predictors, providing the foundation upon which this dissertation is built. Steve is a co-author on our paper on hierarchical branch predictors [32]. It was his suggestion to use a Wallace-tree to implement the perceptron predictor.

Heather Hanson, a graduate student in Electrical & Computer Engineering, has helped me understand the tools used for studying CMOS circuits. She is a co-author on our paper on the Boolean formula predictor. She helped design the circuit for that predictor and she set up the scripts used to measure its power and delay. She helped me understand the HSPICE tool so that I could use it for studying the perceptron predictor.

Vikas Agarwal, a graduate student in Electrical & Computer Engineering, modified the CACTI 2.0 cache simulator for technology scaling for his research. He provided me with his enhanced versions of CACTI and, through a detailed discussion, helped Calvin and me understand the results his predictor yields.

Rajagopalan Desikan, a graduate student in Electrical & Computer Engineering, has helped me understand various aspects of the Alpha 21264 microarchitecture, including the operation of the branch predictor.

Samuel Z. Guyer, a doctoral candidate in Computer Sciences, and Ibrahim Hur, a doctoral candidate in Electrical & Computer Engineering, provided useful suggestions and feedback on drafts of papers.

Many other fellow students have contributed to my research by providing useful feedback at practice talks. They are: Kartik K. Agaram, Vikas Agarwal, Emery D. Berger, Mary D. Brown, Brendon D. Cahoon, Rich J. Cardone, Rajagopalan Desikan, Samuel Z. Guyer, Heather L. Hanson, Ibrahim Hur, Maria E. Jump, Ramgopal R. Mettu, Hrishikesh S Murukkathampoondi, Ramadass Nagarajan, Erik H. Reeber, Juan C. Rubio, Karthikeyan Sankaralingam, Alison N. Smith, Teck Bok Tok, Francis Tseng and Phoebe K. Weidmann.

During the course of my research, I have submitted several papers to peer-reviewed conferences. The anonymous reviewers have provided valuable insights, pointers to literature, and criticisms that I have used to make my research stronger.

Last, but not least, my wife, Stella L. Jiménez, has endured the several years of my graduate career with more cheer than I could have expected. She is my best friend and the source of my strength. She has supported our household financially through my years of voluntarily reduced earnings.

Abstract

Accurate branch prediction is an essential component of a modern, deeply pipelined microprocessors. Because the branch predictor is on the critical path for fetching instructions, it must deliver a prediction in a single cycle. However, as feature sizes shrink and clock rates increase, access delay will significantly decrease the size and accuracy of branch predictors that can be accessed in a single cycle. Thus, there is a tradeoff between branch prediction accuracy and latency. Deeper pipelines improve overall performance by allowing more aggressive clock rates, but some performance is lost due to increased branch misprediction penalties. Ironically, with shorter clock periods, the branch predictor has less time to make a prediction and might have to be scaled back to make it faster, which decreases accuracy and reduces the advantage of higher clock rates.

We propose several methods for breaking the tradeoff between accuracy and latency in branch predictors. Our methods fall into two broad categories: hierarchical predictors using purely hardware implementations, and cooperative predictors that off-load some prediction work to the compiler. We describe hierarchical organizations that extend traditional predictors. We then describe a highly accurate branch predictor based on a neural learning technique. Using a hierarchical organization, this complex multi-cycle predictor can be used as a component of a fast delay-sensitive predictor. We introduce a novel cooperative branch predictor that off-loads most of the prediction work to the compiler with profiling. The compiler communicates profiled information to the microprocessor using extensions to the instruction set. This *Boolean formula predictor* has a small and fast hardware implementation, and will work in less than one cycle in even the smallest technologies with the most aggressive projected clock rates. Finally, we present another cooperative technique, *branch path re-aliasing*, that moves complexity off of the critical path for making a prediction and into the compiler; this technique increases accuracy by reducing destructive aliasing during the less critical update stage.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Modern microprocessors achieve good performance by executing many instructions in parallel. This *instruction-level parallelism* (ILP) can be limited by various bottlenecks, so microprocessors often perform speculative work to reduce the impact of these bottlenecks. One particularly important type of speculation is *dynamic branch prediction*. When a conditional branch instruction is fetched, it may take several cycles for the branch condition to be determined, and until then, it is not clear which path should be followed. A branch predictor uses statistical information to predict which direction the branch will take and to and speculatively fetch and execute instructions along the predicted path. This technique yields a tremendous increase in performance by keeping the pipeline full even in the presence of control hazards. Since about one in seven executed instructions is a branch, branch prediction is essential for modern pipelines that may have tens or even hundreds of instructions simultaneously in flight. If a prediction is incorrect, i.e. there is a *misprediction*, a considerable number of cycles is wasted executing the wrong instructions and restoring the processor state such that the correct path can be executed. Thus, branch predictors must be highly accurate to avoid mispredictions.

Current techniques can achieve correct branch prediction rates of 95% [41], i.e., *misprediction rates* of 5%, but the high cost of recovering from mispredictions [12] remains one of the largest impediments to performance on current and future processors. Because of the large penalty of a branch misprediction, small improvements in accuracy can have a large impact on performance. As pipelines become deeper to support higher clock rates, the penalty for a mispredicted branch will increase. For instance, the Pentium 4 microprocessor has a 20-stage pipeline, with a branch misprediction penalty of approximately 20 cycles [26]. For a simulated processor with a 20-stage pipeline, increasing the branch misprediction rate from 4% to 7% decreases performance by 11% in terms of instructions executed per cycle (IPC). Thus, we are motivated to find more accurate branch predictors for future technologies.

1.1 The Problem: Delay in Branch Predictors

It takes work to accurately predict branches. The amount of time available to do this work has a large impact on the accuracy of the branch predictor. Branch predictor access delay is a crucial component in determining the performance of the processor, since it has an impact on both clock rate and instruction throughput. This delay is affected by trends in technology. In this section, we explain the source of branch predictor delay and the consequences of ignoring delay.

Modern branch predictors are based on the two-level adaptive branch prediction technique introduced by Yeh and Patt [62]. This scheme uses a table of counters to find correlations between previous branch outcomes to make a prediction. For the branch predictor to be accurate, this table should have hundreds or thousands of entries, causing it to resemble a small cache memory.

In the past three decades, performance improvements in microprocessors have been driven in large part by improvements in *process technology*, i.e., the process with which microprocessors are fabricated on wafers of silicon. As process technology improves, the sizes and delays of the transistors and wires on a microprocessor decrease, allowing computer architects to squeeze more functionality onto the chip, and run the chip at a higher clock frequency. Recent studies, however, have shown that as feature sizes have been shrinking in current and future process technology, increasingly aggressive clock rates and larger wire delays will lead to multi-cycle access to large on-chip structures [1] such as caches and branch predictors.

Until now, the huge body of branch prediction research has focused on only two dimensions of the problem—area and accuracy—and has found that larger hardware budgets yield higher accuracy for two reasons: They allow the use of longer history lengths, and they reduce *aliasing,* which occurs when two unrelated branches destructively share the same hardware branch prediction resources. Indeed, much of the recent work has focused on methods for reducing aliasing [52, 41, 38, 57, 18]. With growing chip capacities, the focus of the research community on area and accuracy has led to large elaborate predictors, some of which require 16K to 64K byte structures [20], and to complex prediction schemes that add levels of logic to combat destructive aliasing [18, 38].

Since dynamic branch predictors use large tables to find correlations and make predictions, future branch predictors will need to consider a third dimension: delay. Figure 1.1 illustrates the problem of ignoring delay. Using an idealized delay of one cycle to access the pattern history table (PHT), the *gshare* predictor [41] sees improved *instructions per cycle* (IPC)—due to improved prediction accuracy—as the size of the PHT is increased. By contrast, with an aggressive clock frequency (1.9 GHz) and a realistic delay model for today's 180 nanometer technology, the curve drops off at 512 bytes where the PHT requires two cycles to access, and drops again at 64KB where delay becomes three cycles. This problem will be exacerbated by the smaller process technologies of the future, as shown by the curve for 50nm technology and a projected 6.9 GHz clock rate. In this technology, wire delay causes the table access times to slip above one cycle even earlier.

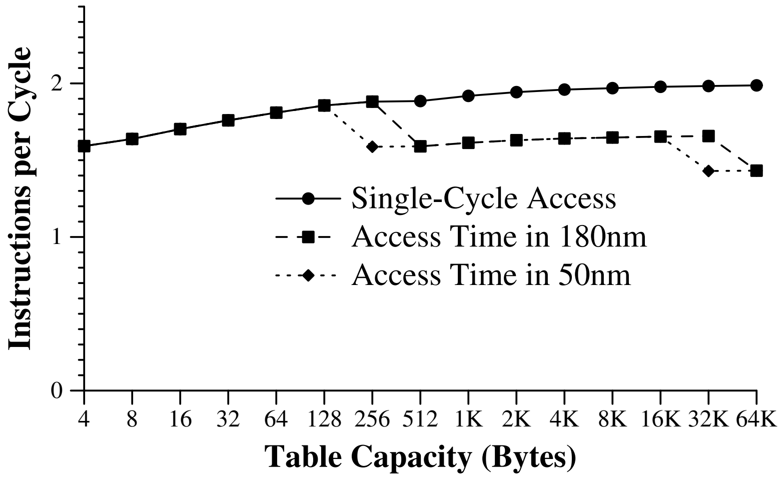


Figure 1.1: Instruction Throughput versus Capacity for the *gshare* predictor.

As an example of a real-world instance of this problem, the branch predictor for the AMD Athlon microprocessor represents a step backward when compared to its predecessor, the K6. While the K6 has a highly accurate 8K-entry GAs predictor, the Athlon uses a less accurate 2K-entry GAs predictor [16]. This change reduces the delay and real estate costs of the branch predictor and could be one reason why the Athlon is able to achieve an aggressive clock rate of 1.4 GHz. Nevertheless, the Athlon has decreased performance in terms of IPC because of the less accurate branch predictor.

Higher clock rates also increase the need for higher branch prediction accuracy. As pipelines become deeper to create less work per cycle, the penalty of a misprediction increases. For example, the Pentium 4 has a 20 stage misprediction pipeline [26]. Table 1.1 shows the clock rates and pipeline depths of several current microprocessors.

In a nutshell, the problem is this: Using smaller branch prediction tables results in lower IPC because of lower accuracy. Naively using larger tables in future technologies results in even lower IPC because of aggressive clock scaling trends and increased relative wire delay. The question that this dissertation addresses is: how can we get both high accuracy and low latency?

Microprocessor	Integer Pipeline Depth	Clock Frequency (MHz)
PowerPC 7400	4	733
HP PA-8700	7	800
Alpha 21264	7	833
AMD Athlon	9	1400
Intel Pentium 4	20	1760

Table 1.1: Pipeline depth vs. clock rate for various processors.

1.2 Dimensions of the Problem

As microarchitecture designs evolve and process technology improves, several dimensions of the branch predictor delay problem are emerging. In this section, we explore these dimensions, and ask important questions for the future of branch predictors.

1.2.1 Extending Traditional Predictors in the Near Future

In the near future, i.e., the next few years, we would like to be able to continue using the traditional branch predictors that have provided such good performance in the past. Table-based branch predictors have been researched heavily. Industrial and academic researchers have very good ideas about how to extract a great deal of accuracy using variations of two-level adaptive predictors; we survey several of these efforts in Chapter 2. However, the impressive performance of these predictors comes at the cost of high access delay. As pipelines deepen to support more and more aggressive clock rates in the near future, the viability of these schemes is threatened by the delay they impose. We can't simply throw away these schemes without having something to replace them with. Thus, we are highly motivated to find ways around the delay problem, so that we can extend the utility of these predictors into the future and continue building traditional cores with deeper pipelines and higher clock rates.

1.2.2 Increasing Accuracy in the Face of Delay

Simply sustaining traditional branch predictors is not sufficient, especially since mispredictions are becoming more costly. How can we make the branch predictor more accurate if it has less time? As we have noted, a large amount of research has been done to improve the accuracy of table-based branch predictors. However, by no means do we believe that this research effort is finished. We believe that there are many more ideas yet to be discovered. Indeed, we introduce one such technique of our own in Chapter 5. How can highly accurate predictors with high access delays be used in processors with very short clock periods? Similarly, are there ways to use table-based predictors that result in high accuracy but are more economical with their time?

1.2.3 Addressing Technology Scaling

As the limits of CMOS process technology scaling are approached in the next decade, wire delay and power will become dominant forces shaping processor design. Because of wire delay, the time it takes to access a reasonably large branch predictor may be a significant fraction of the time it takes an instruction to traverse the pipeline. Thus, traditional table-based branch predictors may become infeasible or even useless in this new setting, and we will be forced to look for something new. Are there ways to accurately predict branches without tables or other expensive components?

1.3 Our Solutions

Our solution to the problem of delay in branch predictors is to divide the prediction work into parts with different delays. During one part of the prediction, a fast branch predictor operates in a single cycle. During another part of the prediction, either off-line through profiling, or on-line through hardware, more time is spent working on making the branch predictor more accurate. We explore two main techniques for dividing the work:

Hierarchical Predictors We propose hierarchical organizations for branch predictors. We describe three branch predictor organizations, each with the common goal of combining a fast predictor with a slower but more accurate predictor to achieve accurate prediction in a single cycle. We apply these ideas in two contexts:

- We demonstrate how these techniques can be applied to conventional predictors whose delay comes from table access time. Thus, we show how traditional predictors can be extended into the next several years of clock scaling and technology improvements.
- We explore the space of more complex predictors that would otherwise be infeasible because of delay: we describe a novel branch predictor based on a neural learning technique. This *perceptron predictor* has unique properties that allow it to yield high accuracy. Using the techniques described above, this complicated multi-cycle predictor can be used as a component of a fast delay-sensitive predictor. Thus, we show that ever more complex and accurate predictors are still feasible, even in the face of the branch predictor delay problem.

Cooperative Predictors Another way to tolerate delay is to off-load some of the prediction work to the compiler, with profiling. In this way, the compiler and hardware cooperate to produce the prediction. Prediction work takes place in two stages: First, an off-line profiling algorithm analyzes the program's behavior on a training input. The compiler communicates profiled information to the microprocessor using extensions to the instruction set architecture (ISA), indicating how to perform the branch prediction with high accuracy. Second, the instruction set communicates the hints to the branch predictor in the running program such that prediction is quick. We describe two novel techniques:

- *Branch path re-aliasing*, a technique that moves complexity off of the critical path for making a prediction and into the compiler. This technique increases accuracy by reducing destructive aliasing during the less critical update stage. This technique allows us to reduce branch predictor delay by shrinking a branch predictor from one generation to the next without sacrificing accuracy. This technique is specific to one particular family of branch predictors.
- A branch predictor in which a profiling phase finds a function used to perform branch prediction for each branch. Each function is encoded as a compact Boolean formula in the branch instruction. The PHT is eliminated, so it is no longer a source of delay. This *Boolean formula predictor* has a small and fast hardware implementation and will work in less than one cycle even in the small technologies and aggressive clock rates for which conventional table-based predictors are infeasible.

1.3.1 Scope and Limitations of the Research

In this dissertation, we focus mainly on the effects of technology scaling on the branch direction predictor. Thus, we mainly study the branch direction predictor in isolation, assuming for the sake of simplicity that other microarchitectural structures are not affected by technology scaling. We only briefly consider other related aspects of microarchitecture, such as branch target buffer delay, instruction cache delay, and branch predictor power, and we do not propose complete solutions to these problems. This methodology allows us to make stronger statements about the future of branch predictors themselves without relying on predictions of other components; however, without taking into account these other components, it is more difficult to interpret our simulated performance numbers. It is important to note that other problems may form more important bottlenecks, such as the increasing disparity between DRAM and CPU speeds, and that our IPC results may be optimistic by assuming that these problems will not get any worse.

1.4 Thesis Statement

The central thesis of this dissertation is this:

Despite the effects of aggressive clock scaling, wire delay, and complex organizations, future branch direction predictors can have improved accuracy while still providing a prediction in a single cycle.

1.5 Contributions

This dissertation makes the following contributions:

1. We show that delay in the predictor significantly erodes performance, so future branch prediction work must consider delay in their designs. We show that increasing delay to improve accuracy is never a good tradeoff. We show that structures with multi-cycle access times can be exploited by hierarchical organizations for branch predictors.

2. We introduce the perceptron predictor, the first dynamic predictor to successfully use neural networks, and we show that it is more accurate than existing dynamic global branch predictors. For example, for a 4K byte hardware budget, our global predictor achieves a misprediction rates on the SPEC 2000 integer benchmarks of 1.94%, compared with 4.13% for a *gshare* predictor of the same size and 2.66% for the McFarling-style hybrid predictor of the Alpha 21264. A version of our predictor that uses both global and per-branch information improves the misprediction rate to 1.71%, an improvement of 36% over the hybrid predictor. By comparing our predictor against a multi-component hybrid predictor, we provide evidence that the perceptron predictor is the most accurate fully dynamic branch predictor known. We suggest how, using hierarchical organizations, the perceptron predictor can be implemented and used in modern CPUs. We show that the perceptron predictor can improve IPC by 15.8% over *gshare* and 5.7% over a hybrid predictor.

3. We present branch path re-aliasing, a technique in which the compiler reduces destructive aliasing by setting a hint bit in the ISA, thereby allowing some dynamic predictors to use smaller tables more effectively. We describe an algorithm for using path profiles to set these hint bits. We present experimental evidence that branch path re-aliasing allows small branch predictors to achieve greater accuracy than other, slower predictors. Our simulations show that a 2048-entry GAg predictor enhanced with branch path re-aliasing has a misprediction rate of 6.5%, 21% lower than the misprediction rate of 8.2% for the same sized, but more complicated, *gshare* predictor, and equivalent to the misprediction rate of a *gshare* predictor with twice the size. We also show that our technique improves accuracy even for the *agree* predictor, which was designed to convert destructive aliasing into constructive aliasing, and we show that our technique can improve the accuracy of complex predictors, such as the hybrid predictor of the Alpha 21264.

4. We present a new method for branch prediction based on Boolean formulas that breaks the trade-off between delay and accuracy. For instance, in one cycle, our predictor can deliver a prediction with the accuracy of a 8K-entry *gshare* predictor in a technology where only a 512-entry *gshare* predictor can be accessed in one cycle. We describe the hardware implementation of our predictor, showing that it has one third of the delay and consumes 1% of the power of a conventional branch predictor. We describe a profiling algorithm for training our predictor.

Chapter 2

Background

To familiarize the reader with the ideas of branch prediction, as well as help place our work in its proper context, we now provide some background into branch prediction. We review the history of branch prediction, explain the basic mechanism, describe characteristics of branch predictors, and review some implemented branch predictors. We also provide background into the technology scaling issues addressed by our work.

2.1 History of Branch Prediction

Branch prediction has a long history in high performance computing. In first survey of branch prediction strategies, Smith describes mechanisms already in place in mainframe computers at the end of the 1970's [56]. Most of these mechanisms are simple, and are based on static characteristics of the program. For example, some IBM System 360/370 models predict whether a branch will be taken based on the branch instruction opcode, since certain opcodes are used for loop back edges and other are used for `IF/THEN/ELSE` statements [56]. Another simple static mechanism is to predict that a backward branch will be taken, while a forward branch will not, observing that backwards loop back edges are frequently traversed more than once. Smith also proposes simple dynamic branch predictors. The basic idea is to use a hash of the branch address as an index into a table of counters that are incremented when the branch is taken, decremented otherwise. When branch prediction is required, the high bit of the corresponding counter is used as the prediction; 1 means *predict taken*, and 0 means *predict not taken*. These historical prediction mechanisms were moderately accurate, but as branch misprediction penalties started to increase, more accurate techniques became necessary.

An important breakthrough came in 1991 when Yeh and Patt observed that the outcome of a given branch is often highly correlated with the outcomes of other recent branches [62]. This history of branch outcomes forms a pattern that can be used to provide a dynamic context for prediction. Most modern branch predictors are based on this pattern history. In the scheme of Yeh and Patt, every time a branch outcome becomes known, a single bit (0 for *not taken*, 1 for *taken*) is shifted into a pattern history register. A pattern history table (PHT) of two-bit saturating counters is indexed by a combination of branch address and history register. The high bit of the counter is taken as the prediction. Once the branch outcome is known, the counter is decremented if the branch is *not taken,* or incremented otherwise, and the pattern history is updated. Recent branch prediction work focuses on refining this scheme of Yeh and Patt. Several predictors have been proposed to deal with the problem of destructive aliasing, which occurs when two unrelated branches contend for the same PHT resources, resulting in decreased accuracy [38, 57, 18]. Hybrid predictors that combine two branch predictors to improve accuracy have been proposed [41, 20] and implemented [35]. The 1990's produced a great deal of research in improving branch predictor accuracy and studying characteristics of branch prediction, and we expect this research to continue.

2.2 Characteristics of Branch Prediction

Several factors influence the design of branch predictors:

Branches are biased. Branches, which can only have two outcomes, *taken* and *not taken,* are highly biased. For instance, a branch that transfers control from the end of a loop back to the beginning will usually be *taken*, since loops usually iterate many times before finishing. Figure 2.1 shows the bias of dynamic branches in the SPEC 2000 integer benchmark suite. The x axis gives the *bias* of a branch, i.e., the percentage of time a branch is *taken*, and the y axis shows the number of branches with a given bias in the SPEC 2000 integer benchmarks. Of all branches, 53% are *taken* at least 98% or at most 2% of the time. The graph on the right excludes these branches, again showing clear biases and a number of branches taken exactly half the time.

Only frequent branches matter. For a conditional branch to have a significant impact on the performance of a program, it must be executed many millions of times. It doesn't matter if a low-frequency branch is incorrectly predicted, because its overall impact on the program's speed is low. Most branches are executed very few times, so it makes sense to concentrate our effort on those few branches that are executed frequently.

Branch predictors must be fast. Branch predictors must meet strict physical constraints. They must operate in one CPU cycle and be small enough to fit on a chip. Most of the hardware devoted to branch predictors is memory for large tables, so the *hardware budget* of a predictor, i.e., the cost of the predictor as a component of the chip, is appropriately measured in kilobytes. A typical predictor occupies 4K bytes of SRAM [35]. However, as we will see, the amount of area reachable in one cycle will decrease in future technologies.

Aliasing is a problem. One problem with dynamic branch prediction schemes is *aliasing*, where the limited resources cause two unrelated branches to use the same prediction resources, resulting in poor performance. Many techniques have been proposed to reduce the impact of aliasing [41, 38, 57, 18]. As the amount of resources reachable in one cycle decreases, this problem will become more difficult to solve.

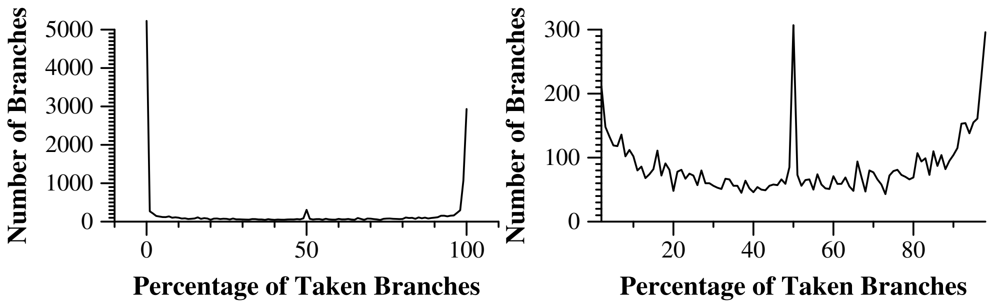


Figure 2.1: Bias in branches.

2.3 Branch Prediction Mechanisms

This section provides background in several branch prediction mechanisms, paying particular attention to branch predictors that directly relate to the research presented in this dissertation.

2.3.1 GAg and GAs Predictors

Our work on branch path re-aliasing focuses on improving the accuracy of GAg branch predictors. Although more accurate predictors exist, GAg and its close relative, GAs, are both used as components of implemented branch predictors in modern machines, as we will see in Section 2.3.6. Yeh and Patt taxonomize two-level branch predictors using a three-letter naming scheme [63]. The first letter represents how the first level branch history is kept. G means a single global history register is used. The second letter denotes the prediction mechanism: A means that a two-bit saturating counter is used. The third letter indicates how the second level table is indexed; g means a single column of counters is used for all addresses while s means that bits extracted from the branch address are used to select a set of counters, and the set is indexed by the history register. Thus, a GAs predictor selects a set of counters from a pattern history table (PHT) using bits from the branch address, and chooses a particular counter from that set using bits from the global history. A GAg predictor uses only the global history to index the PHT.

2.3.2 *gshare* predictor

One popular variant of the GAs predictor is *gshare* [41]. A *gshare* predictor combines the branch history and branch address by exclusive-ORing them together. The exclusive-OR operation has the effect of evenly distributing accesses to the PHT, which would otherwise be unequally distributed due to the non-uniform nature of branch histories. In this way, *gshare* increases accuracy by reducing the probability that two different branches will interfere with each other in the PHT.

2.3.3 *Agree* predictors

A branch direction predictor can be enhanced using the *agree* mechanism [57]. Rather than correlating with branch outcome, the PHT entries in an *agree* predictor keep track of whether a branch outcome will agree with a bias bit set in the branch instruction. The *agree* mechanism turns destructive interference into constructive interference, increasing accuracy. However, since the branch instruction opcode must be read and combined with the PHT prediction, the instruction cache is on the critical path for branch prediction. Note that branch biases can be learned and stored in the branch target buffer rather than branch instruction.

2.3.4 Bi-Mode predictor

Several other branch predictor organizations have been proposed to reduce destructive aliasing in the PHT. We choose the Bi-Mode predictor [38] as a representative of these predictors. The Bi-Mode predictor uses three history tables: two PHTs and a choice table that is used to indicate which PHT to use for a particular combination of branch address and history. Details of this predictor allow it to reduce aliasing by separating into different tables histories that would destructively alias one another, at the cost of increased complexity in the organization.

2.3.5 Static Branch Prediction

A purely static branch predictor always predicts the same outcome for a particular static branch. The prediction can be derived from the structure of the branch itself, e.g., the “backwards taken/forwards not taken” approach of the Alpha AXP-21064, or encoded into the branch instruction itself as a bias bit, as in the IA-64 instruction set. The compiler, through profiling or static heuristics [6, 11], can provide hints to the microarchitecture about the likely direction of the branch. Given enough state, dynamic branch predictors are more accurate than static branch predictors, since dynamic predictors take into account changing conditions at run-time.

2.3.6 Branch Predictors in Current CPUs

Current microprocessors use two-level branch predictors. The following are three notable examples:

- The AMD K6 and K7 (Athlon) processors use GAs predictors [16].
- The HP-PA 8700 uses a 2048-entry GAs with the *agree* mechanism [39, 59].
- The Alpha 21264 core uses a hybrid predictor composed of two two-level predictors [35]: a 4K-entry GAg is indexed by a 12-bit global branch history while a 1K-entry PHT of 3-bit saturating counters is indexed by one of 1024 local 10-bit branch histories. The final branch prediction is chosen by indexing a third predictor that keeps track of the relative accuracies of the two predictors for a particular global history. The Alpha predictor is very accurate; indeed, it is the most accurate of implemented branch predictors that we have observed. However, its implementation complexity comes with a cost. The Alpha branch predictor overrides a less accurate instruction cache line predictor, introducing a single-cycle bubble into the pipeline whenever the two disagree [35].

2.4 Technology Scaling

Branch predictors, like other microarchitecture structures, are affected by two technology scaling trends. Microprocessor designers continue to aggressively increase the clock rates, outstripping the speed improvements achieved by transistors that have smaller gate lengths in each successive technology [1]. Furthermore, at smaller feature sizes, wire delay grows in significance relative to transistor speeds and can affect the latency of the fetch engine and the branch predictor. Faster clocks exacerbate the tradeoff between capacity and delay in microprocessor components. As these trends continue, the chip area reachable in a single cycle will decrease. This means that large banks of SRAM, such as caches and branch prediction tables, will have to either decrease in size or increase in delay. Branch prediction tables are particularly hard hit by clock scaling because they require more address lines than similarly sized caches because caches have wide lines, while branch predictors have narrow two-bit entries. These extra address lines cause significant decoder delay.

To account for accelerating clock rates, we use a technology independent metric, the *fanout-of-four* (FO4) delay metric, to measure clock period [27]. One FO4 delay is the time for an inverter to drive 4 copies of itself. Reasonable models show that under typical conditions, the FO4 delay, measured in picoseconds, is equal to $360 \times L_{drawn}$, where L_{drawn} is the minimum gate length for a technology, measured in microns. The number of FO4 delays in a clock period is an indicator of the number of levels of logic in a pipeline stage. An example of an aggressive clock rate is f_8, which corresponds to a clock period of 8 FO4 delays. The current trends in pipeline depths and clock rates suggest that a clock rate near f_8 may be used in real microprocessors in the near future.

Chapter 3

Methodology

In this chapter, we explain the general methodology we use to obtain our experimental results. Later sections will go into more detail where appropriate. There are three main types of results that we gather: branch misprediction rates, instructions per cycle (IPC), and circuit timings. We gather these statistics in the context of a out-of-order core simulator based on the SimpleScalar/Alpha simulator [10].

3.1 Simulated Microprocessor

We use the SimpleScalar/Alpha out-of-order simulator, configured with microarchitectural parameters similar to those of the Alpha 21264 [35]. We choose this microarchitecture because it is recognized as a leading-edge high performance microprocessor. Since we are focusing solely on the branch predictor, we keep the other structure sizes constant at values shown in Table 3.1. This means that, as we scale feature sizes and clock rates, we assume that the number of cycles to access other structures will not change. Although this is an optimistic assumption, it allows us to isolate the effect of the branch predictor. We assume a baseline microarchitecture with a five-stage pipeline and issue width of four; however, we investigate multiple pipeline depths simulated by changing the misprediction penalty. Note that an issue width of four is conservative; as issue width increases, branch prediction becomes even more important since more work is wasted on a misprediction.

By changing the number of cycles for the branch misprediction penalty and for accessing the branch predictor, we simulate the effect of increasing the clock rate and the depth of the pipeline.

	Capacity (bits)	# entries	Bits/entry	Ports
BTB	48K	512	96	1
Reorder buffer	8K	64	128	8
Issue window	800/320	20	56	8
Integer RF	5K	80	64	10
FP RF	5.6K	72	80	10
L1 I-Cache	512K	1K	512	1
L1 D-Cache	512K	1K	512	2
L2 Cache	16M	16K	1024	2
I-TLB	14K	128	112	1
D-TLB	14K	128	112	2

Table 3.1: Parameters used for the simulations, similar to the Alpha 21264.

Benchmark	Description
`164.gzip`	LZ77 compression
`175.vpr`	Place and route for FPGAs
`176.gcc`	C compiler for Motorola 88100
`181.mcf`	Minimum cost network flow solver
`186.crafty`	Chess playing program
`197.parser`	Natural language processing
`252.eon`	Ray-tracing program
`253.perlbmk`	Perl
`254.gap`	Computational group theory
`255.vortex`	Database
`256.bzip2`	Block-sorting compression
`300.twolf`	Place and route

Table 3.2: SPEC 2000 integer benchmark suite.

3.2 Benchmarks

We simulate the 12 programs in the SPEC CPU 2000 suite of integer benchmarks. The programs are compiled on an Alpha 21264 workstation using the Compaq C compiler V6.3-025 and g++ compiler version 2.9-gnupro-99r1 The optimization levels are chosen from the `peak` settings for the SPEC-supplied configuration files. Table 3.2 shows the name of each benchmark, along with a short description.

3.3 Branch Misprediction Rates

We use the following methodology when reporting branch misprediction rates.

3.3.1 Simulated Branch Predictors

Most of the branch predictors studied are simulated in a C++ framework that is patched into the SimpleScalar/Alpha branch prediction system. The framework can also function in a stand-alone trace-driven simulator. The framework currently supports the following branch predictors:

1. Two-level adaptive branch prediction [62]. This category includes predictors such as *gshare* and other predictors that index a pattern history table (PHT) using a combination of branch address and global (e.g. as in *gshare*) or per-branch history information. The parameters to a two-level predictor are history length, size of the PHT, number of per-branch histories to keep, number of bits per counter, and whether or not to exclusive-OR the branch address with the branch history (as in *gshare*).

2. Hybrid branch prediction [20]. Any two simulated branch predictors can be combined into a hybrid predictor. A table of two-bit saturating counters indexed by global history and/or branch address is used to keep track of which predictor performs best for which branch, and the prediction from the better predictor is returned. The hybrid predictor of the Alpha 21264 is simulated using this mechanism. The parameters are the size of the chooser table and the sort of information that should be used to index it.

3. The *bi-mode* predictor [38]. The parameters are the size of each pattern history table and the global history length.

4. The Agree predictor [57]. A method call allows the user to read a table of bias bits into any branch predictor. The table is then used to implement the *agree* mechanism in that predictor.

5. Perceptron prediction [33]. The parameters are the number of perceptrons, the number of per-branch history bits, the number of global history bits, the number of per-branch histories to keep, and the threshold value θ.

We used this framework to write many trace-driven simulators, testing different areas of our research. The Boolean Formula predictor is simulated alongside this framework, using separate data structures.

3.3.2 Tuning Branch Predictors

It has been observed that branch predictor accuracy is sensitive to history length [41]. We tune each predictor for history length using traces gathered from the each of the 12 benchmarks and the `train` inputs. The traces record the address and outcome (i.e., taken or not taken) of up to 300 million branches for each benchmark. We exhaustively test every possible history length at each hardware budget for each predictor, keeping the history length that yields the lowest harmonic mean misprediction rate. For the *agree* mechanism, we set bias bits in the branch instructions using branch biases learned from the `train` inputs.

3.3.3 Testing Branch Predictors

For each benchmark, we gather traces giving the branch address and outcome for 300 million branches for both `train` and `ref` inputs. Each benchmark executes over one billion instructions before the simulation ends. In our simulations, we skip the first 50 million branches before beginning to record branch prediction accuracy; we have observed that the benchmarks exhibit highly predictable initialization behavior before this time, and then settle into a steady-state.

3.4 Instructions per Cycle

For generating instructions per cycle (IPC) results, we use a modified version of the `sim-outorder` simulator from SimpleScalar/Alpha that uses our branch prediction framework. We simulate each benchmark, measuring the number of cycles and instructions executed, and add in a number of cycles equivalent to the various delays or penalties associated with the particular experiments. We then divide the number of retired instructions by the number of cycles used. Note that this methodology fails to capture some of the wrong-path effects such as cache pollution that would actually be observed in a real latency-sensitive branch predictor; however, these effects are small. For experiments that require fewer than 1000 executions of the performance simulator, we run each benchmark for one billion instructions. For experiments that require more than 1000 executions, we run each benchmark for 500 million instructions. For instance, tuning a predictor to search a large design space may require many thousands of executions, but getting results using an already tuned set of configurations may require only dozens or hundreds of executions.

3.5 Circuit Timings

Several of our experiments require analysis to determine the delay of circuit components such as pattern history tables and computational elements related to the perceptron and Boolean formula branch predictors.

3.5.1 HSPICE

We simulate combinational logic circuits using the HSPICE simulator. The HSPICE simulations use transistor models tailored to fabrication processes to simulate the circuit behavior for several technology generations, from current generations with minimum feature (transistor and wires) sizes of 180nm down to future generations that will have minimum feature sizes of 35nm.

3.5.2 CACTI

To estimate pattern history table access times for a range of current and future integrated circuit generations, we use circuit simulations and a modified version [2] of the CACTI 2.0 tool for simulating cache delay. This modified version of CACTI is more accurate than the original in several ways. First, while the original version of CACTI 2.0 [49] uses a simplistic linear scaling for delay estimates, the modified simulator uses separate wire models to account for the physical layout of wire interconnects: thin local interconnect, taller and wider wires for longer distances, and the widest and tallest metal traces for global interconnect. Second, wire resistance is based on copper rather than aluminum material properties. Third, all capacitance values are derived from three-dimensional electric field equations. Fourth, bit-lines are placed in the middle layer metal, where resistance is lower. Finally, bit-addressing is allowed instead of byte-addressing. Our versions of HSPICE and CACTI both use the same parameters for technology scaling.

3.6 Computing Facilities

We run our simulations on a network of approximately 200 Pentium III computers using the Condor system for coordinating the execution of many jobs [8].

Chapter 4

Hierarchical Organizations

In this chapter, we examine three organizational approaches for dealing with delay in future process technologies: (1) a two level caching scheme, (2) an *overriding* scheme that allows a first prediction to be overturned by a more accurate second prediction, and (3) a *cascading* lookahead scheme that exploits cycles between branches to do prediction work. We show that delay in the predictor significantly erodes performance, so future branch prediction work must consider delay in their designs. We show that increasing delay to improve accuracy is never a good tradeoff. We show that there are approaches to branch prediction that can effectively use large structures with multi-cycle access times, and give experimental results showing that IPC can be sustained using these techniques.

4.1 Motivation

Larger branch prediction structures lead to larger access delays. Aggressively increasing clock rates (as the market-place demands) increases the structure access time as measured in clock cycles.

Our studies show that it is never worth increasing the delay of a branch predictor for the sake of improved accuracy [32]. For example, Figure 1.1 shows that as we increase the capacity of the tables in *gshare*, we increase delay and decrease IPC. This effect can be explained with the following equation which roughly approximates the cost C of executing a branch instruction:

$$C = d + (r \times p)$$

where d is the delay of branch predictor, r is the misprediction rate, and p is the misprediction penalty. While the delay d may not always be on the critical path of the pipeline, increasing d will reduce the instruction fetch bandwidth to the execution cores. Because misprediction rates tend to be below 10%, changes in delay have a larger impact than small changes in r for practical values of d (i.e., for forseeable pipeline depths).

4.2 Branch Frequency

A program's control behavior is based not only on the predictability of its branches, but also on the branch frequency. If branch prediction is required on every clock cycle, any delay in branch prediction will substantially slow the instruction fetch rate. However, if branches are widely spaced, then branch prediction latency will have less impact on performance. While the common wisdom is that branches occur on average every fourth or fifth instruction, we find that, in our framework (i.e., a real-world optimizing compiler on the Alpha), branches actually occur one every nine instructions, on average. The actual dynamic distribution of inter-branch latencies is more instructive. We use SimpleScalar/Alpha [10] to measure the average branch frequency the twelve SPEC 2000 integer benchmarks on a 4-way out-of-order machine configuration. Figure 4.1 is a histogram of average inter-branch latencies, measured in cycles between prediction requests, for the SPEC 2000 integer benchmarks. Over 67% of the prediction requests occur more than one cycle after the previous request. The unused cycles provide additional time to predict future branches. For wider issue machines, there is less additional time to make a prediction.

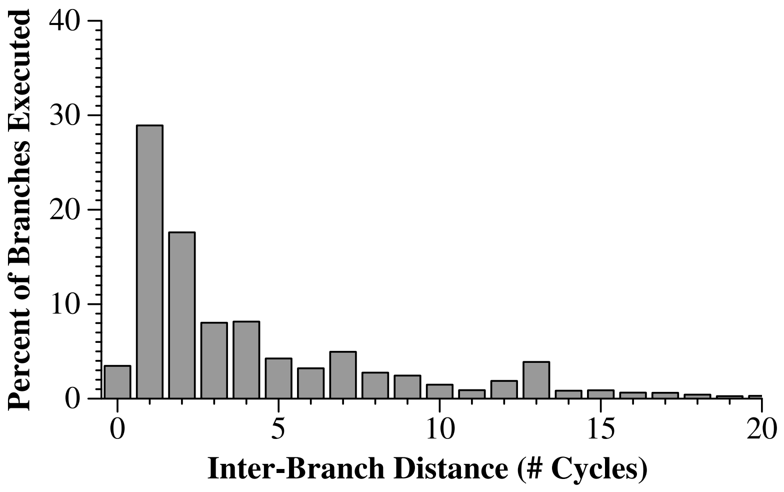


Figure 4.1: Inter-branch latencies

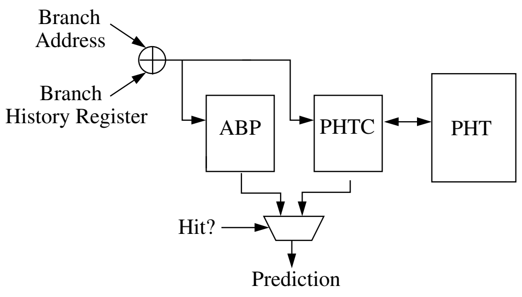


Figure 4.2: Caching branch predictor

4.3 Hierarchical Organizations for Latency Sensitive Branch Predictors

In this section, we describe three ways to configure branch predictors to increase accuracy in the face of increasing latency. These techniques are appropriate when standard techniques for branch prediction might exceed one cycle, and these are general techniques that can be applied to most prediction algorithms.

To achieve high prediction accuracy, the branch predictor may require larger tables. The goal of the microarchitect is to achieve accuracy approaching that of a large table, with the latency of a small table. We examine three methods for achieving this goal.

4.3.1 Caching Prediction Tables

The first strategy to combat the long latency of large branch prediction tables is to build a small cache of branch prediction table entries. This strategy allows us to realize the benefits of reduced aliasing and increased history length without the added latency of the large table, since the cache will have an access time of one cycle. For instance, a 128K-entry PHT accessible in two cycles can be cached using a 1K-entry PHT accessible in one cycle. Figure 4.2 shows the organization of the *gshare* predictor augmented by a cache. The branch history and branch address are hashed using the XOR gate, and the resulting address is sent to both the pattern history table cache (PHTC) and the pattern history table (PHT). The PHT consists of 2-bit saturating counters, with the number of counters equal to the number of combinations of addresses produced by the hash function. The PHTC caches a subset of those counters in a smaller table that can be accessed more quickly than the PHT. If the correct counter is found in the PHTC, then the prediction can be made immediately. If a miss in the PHTC occurs, then the PHT must

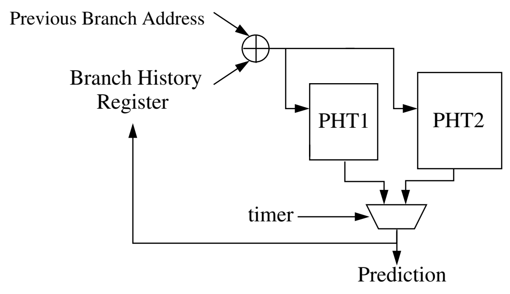


Figure 4.3: Cascading branch predictor

be consulted to find the correct saturating counter. Like traditional caches, an entry in the PHTC is replaced with the correct counter from the PHT. When the branch direction is determined during a later stage of the execution pipeline, the counters in the PHT and PHTC are updated to reflect the correct or incorrect prediction of that branch.

If a PHTC miss occurs, the wait for the correct prediction from the PHT will delay instruction fetch and will degrade overall performance. Two alternatives can be used to prevent this additional delay. First, the prediction produced by the PHTC, albeit for the wrong branch, can be used. Alternatively, we suggest building a small auxiliary branch predictor (ABP) that can be accessed at the same time as the PHTC. If the PHTC misses, then the result from the ABP is used.

4.3.2 Cascading Lookahead Branch Prediction

Lookahead branch prediction has been proposed as a mechanism to increase fetch bandwidth by generating addresses for future branches [64, 54] (see Chapter 8 for more related work). The same technique can be applied to reduce the impact of longer latency branch predictors. If the branch predictor is not needed on every cycle, then natural spacing between branches can be used to perform a prediction for the next branch that is likely to arrive. Thus, if branches are spaced so that the predictor is accessed only every other cycle, the predictor can have a two cycle latency without introducing additional delay. Figure 4.1 shows us that the predictor is usually needed at most once every other cycle.

The *gshare* predictor can be adapted to look one branch ahead. While *gshare* uses the branch history register and branch address to compute the PHT address, a lookahead predictor uses the predicted history and the address of the most recently fetched branch. Since the prediction can complete before the next branch arrives at the predictor, prediction is instantaneous. However, if the prediction requires multiple cycles (due to a large table) and the next branch arrives before the prediction is complete, the instruction fetch engine stalls.

Cascading lookahead branch prediction implements a series of tables of ascending size and latency. Figure 4.3 shows a two-level cascading predictor. Like a lookahead predictor, the next prediction is based on the last prediction and the last branch address. Prediction begins simultaneously on both levels of the cascading predictor. If the latency to the next branch to be predicted is large, then the prediction from the second level table is selected. If the next branch arrives before the second level table can complete its access, then the prediction from the first level table is used.

Thus, the combination of a small first level table and a larger second level table can provide better aggregate accuracy with the minimum latency. However, the utility of the larger table depends on its access time and the inter-branch latency. If branches occur extremely frequently, the second level of the cascade will not be used. The cascading design can be trivially extended to more than two levels. Furthermore, hybrid predictors of varying latencies can be incorporated into the cascading strategy. In our description above, the logic that selects the prediction to use is based only on the arrival time of the next branch. More complicated selectors could trade off latency versus accuracy by predicting which of many predictions is best for the subsequent branch.

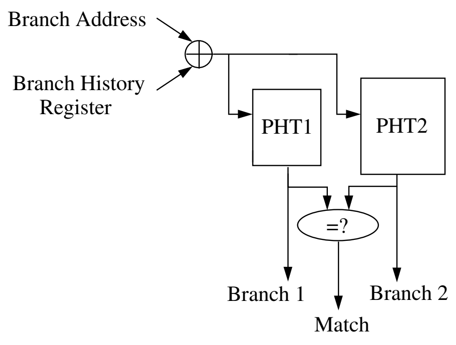


Figure 4.4: Overriding branch predictor

Gate (nm)	8FO4 Clk f_8 (GHz)
180	1.9
130	2.7
100	3.5
70	5.0
50	7.0
35	10.0

Table 4.1: Projected clock rates using 8 FO4 Clock scaling.

4.3.3 Overriding Branch Predictor

An overriding branch predictor (Figure 4.4) provides two predictions. The first prediction comes from a fast PHT (PHT1), and the second prediction comes from a slower, but more accurate PHT (PHT2). When branch prediction is requested, the first prediction is used and acted upon while the second prediction is still being made. If the second prediction differs from the first prediction, the actions taken based on the first prediction are squashed and instructions are fetched using the second prediction; thus, the second predictor overrides the first predictor. For the overriding scheme, we assume that the penalty of restarting an overridden fetch is equal to the delay of PHT2. A similar technique is used in the Alpha 21264, in which the branch predictor, whose results become known only in the second stage of the pipeline, can override the less accurate instruction cache line predictor [35] with a single-cycle penalty. We assume the predictor is pipelined such that no branch needs to wait for the completion of a PHT2 lookup for a previous branch.

4.4 Technology Scaling

Table 4.1 lists the technologies that we consider and the clock rates that result from aggressive (f_8) scaling. We base our estimates of branch predictor delay on the access time of the memory-oriented structures such as the pattern history table (PHT). To model PHT delay, we use the methodology described by Agarwal, et al. [1], which augments the CACTI cache delay modeling tool [49] with scaled technology parameters. We convert the access time produced by the augmented CACTI model into cycles, according to the f_8 clock scaling strategy. As shown in Figure 4.5, only small tables of 1024 entries can be accessed in a single cycle, and at 35nm, only 512 entries can be accessed in one cycle. Accepting a 2 or 3 cycle delay increases the capacity to 16K and 64K entries, respectively.

4.5 Results and Analysis

In this section we evaluate the three latency sensitive branch predictors and compare them to *gshare* across a spectrum of process technologies. In addition, we evaluate a fourth predictor that combines the cascading and overriding

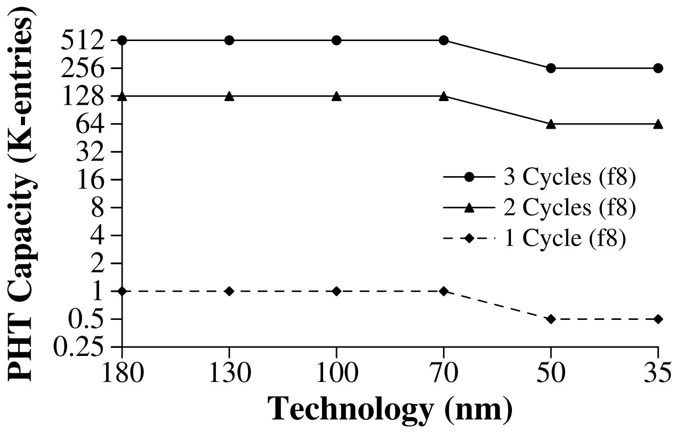


Figure 4.5: Pattern History Table capacity and access latencies.

predictors. This predictor uses a cascading predictor that continues predicting after the branch has been encountered, overriding the first prediction if the second prediction is different.

We used the methodology outlined in Chapter 3 to evaluate the different prediction strategies described above using delay estimates at seven process technologies ranging from 180 nm to 35 nm, representing technologies from today to the predicted smallest feature sizes for which conventional CMOS will be feasible. Each `sim-outorder` simulation runs for 500 million instructions. In the simulations, the global pattern history register is updated speculatively and backed up on a mispredict, while updates to the PHTs are done when the updating branch commits.

We report two types of results. First, we give results using an aggressive eight FO4 (f_8) clock rate, an aggressive clock rate for future technologies [1] that emphasizes the scaling difficulties of branch predictor structures. Next, we give results for a fixed process technology with a clock rate varying from 5 FO4 to 16 FO4, to show the effect of aggressive clock rates independent of process technology scaling. This set of clock rates allows us to explore a wide range of processor design philosophies, from sophisticated wide-issue low clock rate processors to deeply pipelined, high clock rate processors.

4.5.1 Process Technology Scaling

For each process technology, we configure the simulator with the largest branch prediction structures (predictor tables, cache, etc.) reachable at the given number of cycles allocated to branch prediction. The structure sizes are obtained using the modified version of CACTI described in Chapter 3. For each benchmark we measure IPC, aggregate branch predictor accuracy, and other statistics related to the branch prediction schemes. Aggregate branch prediction performance is computed as the arithmetic mean over the benchmarks. Note that the capacity of each structure is set by its access time, rather than any chip area limitation. With smaller feature sizes, this assumption is reasonable, as the amount of effective chip area is far larger than is reachable in the number of cycles we consider.

Predictor Configuration For each predictor, we consider several configurations of structure capacity and latency in search of the best configuration at each technology generation. In the caching predictor, the two structures are the PHTC and the PHT, while in the overriding and cascading predictor the two structures are the PHT1 and PHT2. As the secondary structure access times increase, the resulting IPC is slightly worse for the overriding predictor and slightly better for the cascading predictor. The size of the secondary structure for the caching predictor makes little difference in performance. The rest of our results are reported using the best configurations found for each prediction technique.

In the caching predictor, we varied the latency of the PHT from 2 to 4 cycles, keeping the PHTC at a 1-cycle access time. Note that increasing the latency of each table also increases its capacity.

Technology (nm)	PHT1 Delay	PHT1 Entries	PHT2 Delay	PHT2 Entries
180	1	1K	2	128K
130	1	1K	2	128K
100	1	1K	2	128K
70	1	1K	2	128K
50	1	512	2	64K
35	1	512	2	64K

Table 4.2: The best configurations of the PHT1 and PHT2 for the cascading and overriding predictors.

For the cascading and overriding predictors, we keep access to the primary PHT at one cycle while varying access to the secondary PHT from from 2 to 4 cycles. Increasing the second level (PHT2) latency reduces IPC slightly for the overriding predictor, but increases IPC slightly for the cascading predictor.

Initially, while tuning the caching predictor, we noticed that the PHTC has an unusually small number of entries compared with the other structures. Unlike a normal cache that has large cache lines, our caching predictor requires many times more tag bits than data bits. The extra wire length involved in accessing the tag bits severely restricts the number of cache entries, limiting the effectiveness of this scheme. Other prediction components in which the size of the basic prediction element is large with respect to the number of tag bits, such as the perceptron predictor, may be more amenable to a caching scheme. For this study, we chose a 2-way set-associative cache with a line size equal to the square root of the number of cached prediction table entries. Thus, we trade tag bits for locality. We can access a larger structure, but, due to the absence of spatial locality in branch prediction table access patterns, we must settle for high miss rates in the cache.

The best configurations for the cascading predictor at the f_8 clock rate are shown in Table 4.2. The best configurations for the overriding predictor are identical to those of the cascading predictor, since the two predictors have much the same architecture and differ only in their policy of when and whether to use the second-level PHT. Indeed, the stream of updates to the PHT1 and PHT2 structures should be the same in both overriding and cascading predictors; the only difference is that the overriding predictor always uses the PHT2 prediction, while the cascading predictor only uses the PHT2 prediction when it has enough time.

Accuracy and Performance: Figure 4.6 shows the accuracies of the best configurations of the various predictors at the f_8 clock rates. As shown in the graph, accuracy tends to decrease with feature sizes, because the prediction table capacities decrease. The accuracy of the overriding predictor increases slightly from 100 to 70nm, since the best configuration for 70nm technology allows the PHT2 to take three cycles, while the best configuration in 100nm allows only two cycles. The combination of the cascading and overriding predictors achieves the highest accuracy because it always uses a larger second-level predictor, either because it agrees with or overrides the first-level predictor. The cascading predictor by itself performs worse because it sometimes uses the less accurate first-level predictor when there are not enough cycles to use the second-level predictor. Thus this predictor faces the challenge of branch misprediction as well as branch target misprediction. Finally, caching performs less well, not even exceeding the accuracy of a single level *gshare* predictor due to the fact that pattern history table accesses exhibit very little locality.

Of course, accuracy is not necessarily indicative of performance, particularly when prediction time is a variable. Figure 4.7 show the instruction throughput (IPC) for each of the configuration described above. The predictors follow parallel trajectories with performance reflecting the overall accuracy of the predictor. Clearly, the combination of cascading and overriding predictors, with it higher accuracy, is best for every process technology at the aggressive f_8 clock rate.

4.5.2 Clock Rate Scaling

We have seen how wire delay will affect branch predictor delay at the fixed f_8 clock rate in future technologies. Now, we illustrate the problem along a different dimension. We look at a fixed technology, 130 nm feature size, and vary the clock rate from 1.3 GHz to 3.6 GHz. This technology is especially relevant since it is in the process of

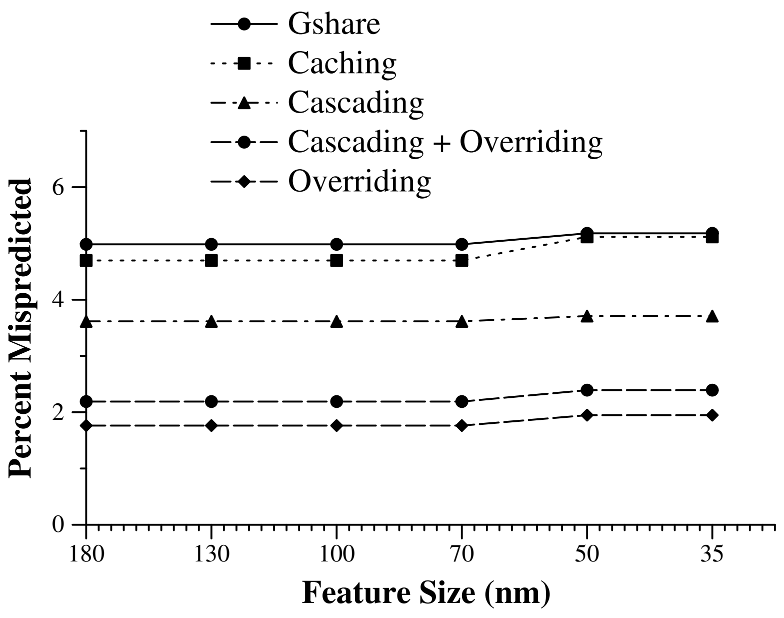


Figure 4.6: Misprediction Rate vs. Technology for the prediction strategies at f_8.

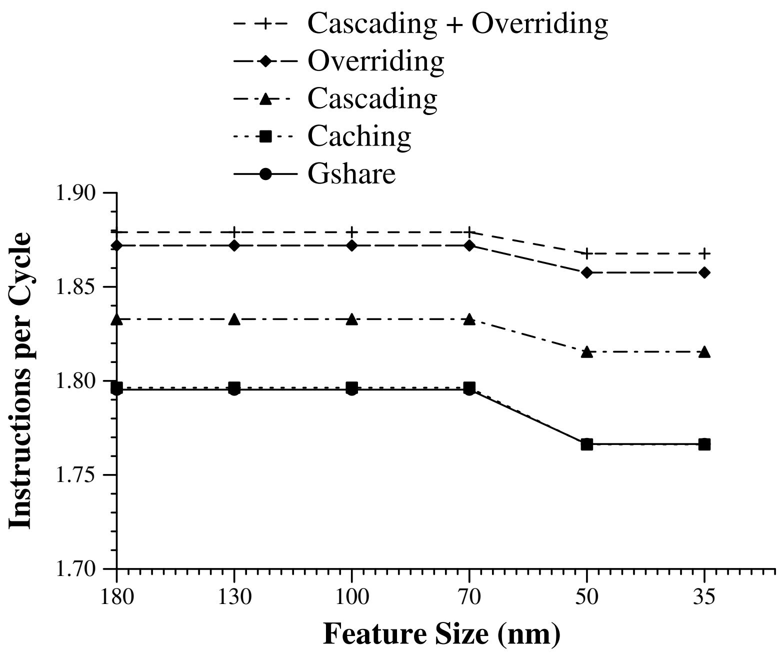


Figure 4.7: IPC vs. Technology for the prediction strategies at f_8.

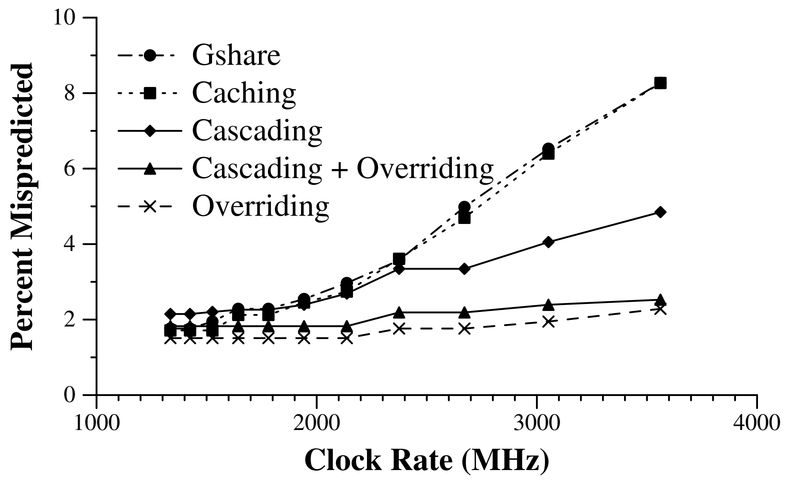


Figure 4.8: Misprediction Rate vs. Clock Rate in 130 nm Technology.

being adopted by manufacturers as we write this dissertation. This range of clock rates is equivalent to clock periods from f_{16} down to f_5. As the clock rate increases, the size of the largest PHT accessible in a single cycle decreases.

Figure 4.8 shows the misprediction rates of the various predictors as the clock rate is increased. The misprediction rate for *gshare* increases dramatically as the clock rate is increased. At 1.3 GHz (f_{16}), the misprediction rate of a 32KB *gshare* is 1.76%. The rate increases to 2.3% at for a 4KB *gshare* at 1.8GHz, a clock rate equivalent to f_{11}. At 3.5 GHz or f_5, the misprediction rate is a distressing 8.26%, because only a very small 4-byte *gshare* can be built at this aggressive clock rate. It's important to note that at this point, the CACTI models of circuit behavior are unrealistic, since we would not use a cache-like structure to address 32 bits of SRAM.

The misprediction rate of the overriding predictor remains the lowest of all the techniques. At 1.3 GHz, the misprediction rate is 1.83%. At 3.6 GHz, the misprediction rate is still low at 2.52%, an improvement of 70% over *gshare*.

Figure 4.9 shows the IPCs yielded by the predictors as the clock rate is increased. As expected, all of the IPCs go down as clock rate increases. Nevertheless, using hierarchical organizations, we can reduce the percentage by which IPCs decrease. Using *gshare*, going from 16 FO4s to 5 FO4s results in a reduction in IPC due to increase branch misprediction of 16%, from 1.90 to 1.61. By combining overriding and cascading, IPC is reduced by only 2.6%, down to 1.85%.

4.6 Summary

In this chapter we have examined a number of hierarchical branch predictor organizations and evaluated them in the context of aggressive clock rates and future process technologies. The predictor that caches a pattern history table (PHT) for *gshare* performs no better than *gshare* by itself. The tags needed to implement a caching scheme requires more bits than the cache itself, and limits both cache capacity and utility. The cascading lookahead predictor that uses the time in between branches to make predictions performs reasonably well at aggressive clock rates. An overriding predictor that allows a slow predictor to cancel the prediction of a faster, but less accurate predictor performs even better than the cascading approach. We achieve the best performance by combining the cascading and overriding approaches.

To continue supplying a sufficient number of instructions to the execution core while continuing to use large table-based branch predictors, future microarchitectures must move branch prediction latency off of the critical path. The schemes we present, particularly the combination of cascading and overriding predictors, can be augmented by using something other than *gshare* as the primary or secondary predictor. We believe that the secondary predictor is the ideal place for a more complex and longer latency predictor, as it can be kept off of the critical path. We explore

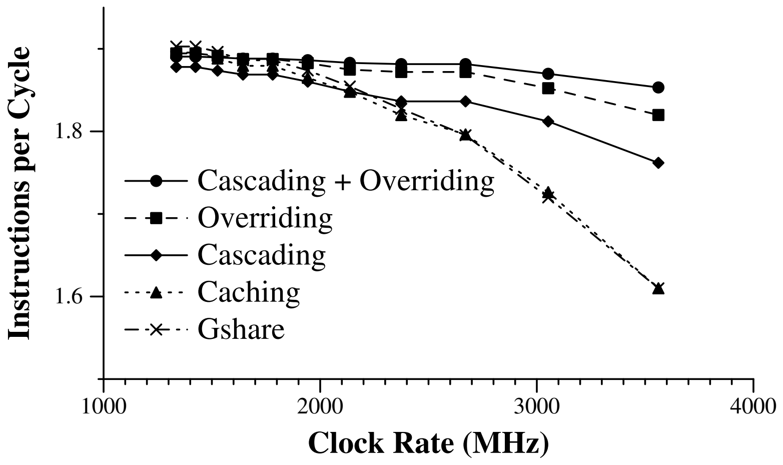


Figure 4.9: Clock Rate vs. IPC in 130 nm Technology.

hierarchical organizations for one such predictor in the next chapter.

Chapter 5

Perceptron Predictor

We have seen that hierarchical predictor organizations allow us to tolerate some latency in the branch predictor, and still deliver a prediction in a single cycle. Rather than simply extend existing predictors to use larger tables for increased accuracy, we explore the use of computationally complex branch predictors that have previously been infeasible because of delay. We propose a new predictor, the *perceptron predictor*, based on neural learning. This predictor provides a case study for hierarchical delay-sensitive predictors, since the neural method used takes multiple cycles to provide a prediction. Our work builds on the observation that all existing two-level techniques use tables of saturating counters. Neural networks are another prediction mechanism capable of providing good predictions. It is interesting to ask whether we can improve accuracy by replacing these counters with neural networks. Since most neural networks would be prohibitively expensive to implement as branch predictors, we explore the use of perceptrons, one of the simplest possible neural networks. Perceptrons are easy to understand, simple to implement, and have several attractive properties that differentiate them from more complex neural networks, such as a space-efficient representation and a relatively quick method for computing the prediction.

We propose a two-level scheme that uses fast perceptrons instead of two-bit counters [33]. Ideally, each static branch is allocated its own perceptron to predict its outcome. Traditional two-level adaptive schemes use a pattern history table (PHT) of two-bit saturating counters, indexed by a global history shift register that stores the outcomes of previous branches. This structure limits the length of the history register to the logarithm of the number of counters. Our scheme not only uses a more sophisticated prediction mechanism, but it can consider much longer histories than saturating counters.

We give results showing that our predictor outperforms other predictors at moderate and large hardware budget, providing evidence that the perceptron predictor is the most accurate fully dynamic branch predictor known. We explain why and when our predictor performs well. We show that the neural network we have chosen works well a class of *linearly separable branches*, a term we introduce. We also show that programs tend to have many linearly separable branches, and that linearly inseparable branches are predicted just as well by the perceptron predictor as by other predictors.

This chapter describes our technique for doing branch prediction using neural learning. We motivate the idea, describe neural methods for branch prediction, discuss implementation of the predictor, and give results showing how a hierarchical organization can enable our complex predictor to provide a prediction in a single cycle.

5.1 Neural Methods for Dynamic Branch Prediction

Artificial neural networks learn to compute a function using example inputs and outputs. Neural networks have been used for a variety of applications, including pattern recognition, classification [23], and image understanding [36, 30]. In this section, we explain how neural methods might be applied to dynamic branch prediction. We discuss the general idea, then explain why we chose the perceptron in particular for branch prediction.

5.1.1 Prediction with Neural Methods

Suppose a set S is partitioned into n classes, and we are faced with the problem of determining, for an arbitrary element $s \in S$, what class s is in. The elements of S have certain features which correlate with their classifications. An artificial neural network can learn correlations between these features and the classification. An artificial neural network is a collection of neurons, some of which receive input and some of which produce output, that are connected by links. Each link has a weight associated with it that determines the strength of the connection [23]. For a classification problem such as deciding to which of n classes an input s belongs, there are n output neurons. In the special case where there are only two classes, there is only one output neuron. Each neuron computes its output from the sum of its input using an *activation function*. During a training phase, the weights are adjusted using a training algorithm. The algorithm uses a set of training data, which are ordered pairs of inputs and corresponding outputs. The neural network learns correlations between the inputs and outputs, and generalize this learning to other inputs. To predict which class a new input s is in, we supply s to the input units of the trained neural network, propagate the values through the network, and examine the n output neurons. We classify s according to the neuron with the strongest output. In the special case where $n = 2$, there is only one output neuron; in this case, we classify s according to whether the output value exceeds a certain threshold, typically 0 or $\frac{1}{2}$.

5.1.2 Neural Learning for Dynamic Branch Prediction

For dynamic branch prediction, the inputs to a neural learning method are the binary outcomes of recently executed branches, and the output is a prediction of whether or not a branch will be taken. Each time a branch is executed and the true outcome becomes known, the history that led to this outcome can be used to train the neural method on-line to produce a more accurate result in the future.

5.1.3 Choosing a Neural Method

There are many types of neural networks. Most of them are inappropriate for branch prediction because they require much longer than several machine cycles to operate. Thus, for our discussion, we limit ourselves to neural network architectures that could feasibly be made to operate at the high speeds required for branch prediction. We consider four methods: multi-layer perceptrons with back-propagation, the ADALINE neuron [60], Hebb learning [23], and the Block perceptron [9]. In preliminary work, we measured the misprediction rates yielded by each method on the SPEC95 benchmarks. Hebb learning, ADALINE neurons and Block perceptrons are simple neural learning methods, in which a single neuron is used for computation and is trained with a simple algorithm. Hebb learning yields poor branch prediction accuracy. While ADALINE and the perceptron yield similar prediction accuracy, the ADALINE neuron requires twice as much space to represent the weights with sufficient accuracy. Back-propagation is infeasible because of its implementation complexity, since there is no way to implement back-propagation in hardware such that a prediction can be produced in just a few cycles. Moreover, in our preliminary experiments we find that the perceptron learns faster and yields more accurate prediction than back-propagation. For instance, on the SPEC95 benchmark *126.gcc*, perceptrons achieve a 2.44% misprediction rate, compared with 3.33% for back-propagation [34].

One benefit of perceptrons is that by examining their *weights*, i.e., the correlations that they learn, it is easy to understand the decisions that they make. By contrast, a criticism of many neural networks is that it is difficult or impossible to determine exactly how the neural network is making its decision. Techniques have been proposed to extract rules from neural networks [53], but these rules are not always accurate. Perceptrons do not suffer from this opaqueness; the perceptron's decision-making process is easy to understand as the result of a simple mathematical formula.

5.2 Branch Prediction with Perceptrons

This section provides the background needed to understand our predictor. We describe perceptrons, explain how they can be used in branch prediction, and discuss their strengths and weaknesses. Our method is essentially a two-level predictor, replacing the pattern history table with a table of perceptrons.

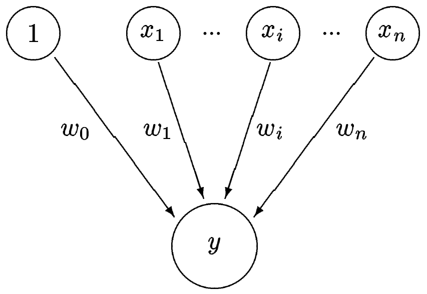


Figure 5.1: Perceptron Model.

5.2.1 How Perceptrons Work

The perceptron was introduced in 1962 [50] as a way to study brain function. We consider the simplest of many types of perceptrons [9], a *single-layer perceptron* consisting of one artificial *neuron* connecting several *input units* by weighted edges to one *output unit.* A perceptron learns a target Boolean function $t(x_1, ..., x_n)$ of n inputs. In our case, the x_i are the bits of a global branch history shift register, and the target function predicts whether a particular branch will be taken. Intuitively, a perceptron keeps track of positive and negative correlations between branch outcomes in the global history and the branch being predicted.

Figure 5.1 shows a graphical model of a perceptron. A perceptron is represented by a vector whose elements are the weights. For our purposes, the weights are signed integers. The output is the dot product of the weights vector, $w_{0..n}$, and the input vector, $x_{1..n}$ (x_0 is always set to 1, providing a "bias" input). The output y of a perceptron is computed as

$$y = w_0 + \sum_{i=1}^{n} x_i w_i.$$

The inputs to our perceptrons are *bipolar*, i.e., each x_i is either -1, meaning *not taken* or 1, meaning *taken.* A negative output is interpreted as *predict not taken.* A non-negative output is interpreted as *predict taken.*

5.2.2 Training Perceptrons

Once the perceptron output y has been computed, the following algorithm is used to train the perceptron. Let t be -1 if the branch was not taken, or 1 if it was taken, and let θ be the *threshold*, a parameter to the training algorithm used to decide when enough training has been done.

```
if $\text{sign}(y_{out}) \neq t$ or $|y_{out}| \leq \theta$ then
        for $i := 0$ to $n$ do
                $w_i := w_i + t x_i$
        end for
end if
```

Since t and x_i are always either -1 or 1, this algorithm increments the i^{th} weight when the branch outcome agrees with x_i, and decrements the weight when it disagrees. Intuitively, when there is mostly agreement, i.e., positive correlation, the weight becomes large. When there is mostly disagreement, i.e., negative correlation, the weight becomes negative with large magnitude. In both cases, the weight has a large influence on the prediction. When there is weak correlation, the weight remains close to 0 and contributes little to the output of the perceptron.

5.2.3 Linear Separability

A limitation of perceptrons is that they are only capable of learning *linearly separable* functions [23]. Imagine the set of all possible inputs to a perceptron as an n-dimensional space. The solution to the equation

$$w_0 + \sum_{i=1}^{n} x_i w_i = 0$$

is a hyperplane (e.g. a line, if $n = 2$) dividing the space into the set of inputs for which the perceptron will respond *false* and the set for which the perceptron will respond *true* [23]. A Boolean function over variables $x_{1..n}$ is *linearly separable* if and only if there exist values for $w_{0..n}$ such that all of the *true* instances can be separated from all of the *false* instances by that hyperplane. Since the output of a perceptron is decided by the above equation, only linearly separable functions can be learned perfectly by perceptrons. For instance, a perceptron can learn the logical AND of two inputs, but not the exclusive-OR, since there is no line separating *true* instances of the exclusive-OR function from *false* ones on the Boolean plane. Figure 5.2 graphs the bipolar AND and XOR functions. A solid line separates the *true* instance of AND from the *false* instances, but the dotted line is unable to separated *true* instance of XOR from the *false* instances.

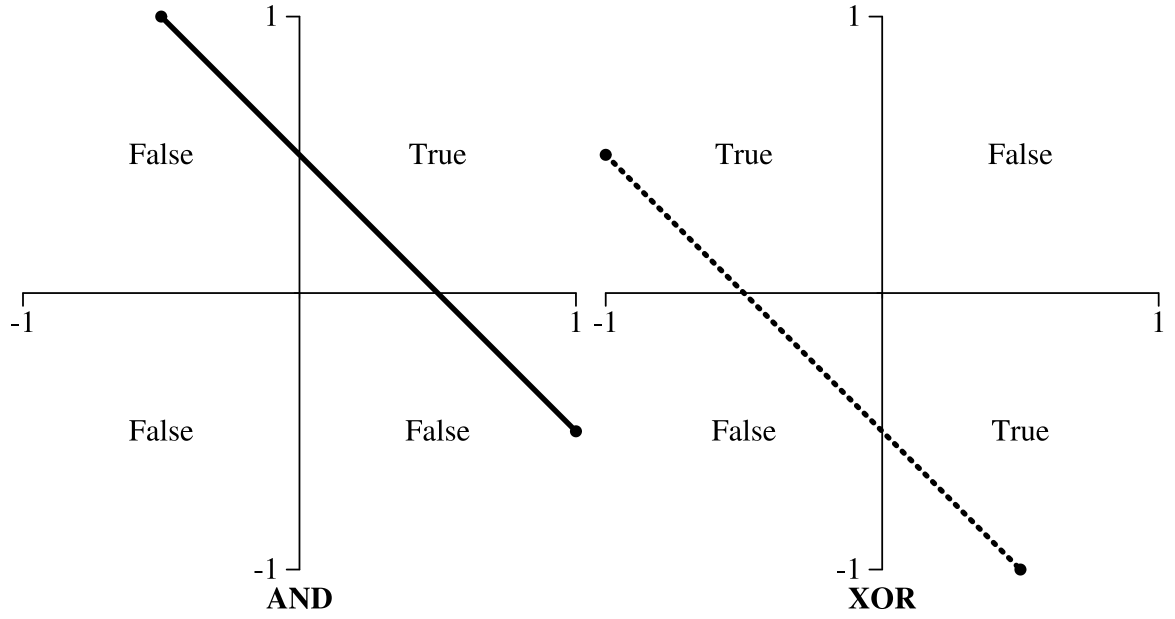


Figure 5.2: Bipolar AND and XOR functions.

As we will show later, many of the functions describing the behavior of branches in programs are linearly separable. Also, since we allow the perceptron to learn over time, it can adapt to the non-linearity introduced by phase transitions in program behavior. A perceptron can still give good predictions when learning a linearly inseparable function, but it will not achieve 100% accuracy. By contrast, two-level PHT schemes like *gshare* can learn any Boolean function if given enough training time.

5.2.4 Branch Prediction with Perceptrons

We can use a perceptron to learn correlations between particular branch outcomes in the global history and the behavior of the current branch. These correlations are represented by the weights. The larger the weight, the stronger the correlation, and the more that particular branch in the global history contributes to the prediction of the current branch. The input to the bias weight is always 1, so instead of learning a correlation with a previous branch outcome, the bias weight, w_0, learns the bias of the branch, independent of the history.

The processor keeps a table of N perceptrons in fast SRAM, similar to the table of two-bit counters in other branch prediction schemes. The number of perceptrons, N, is dictated by the hardware budget and number of weights, which itself is determined by the amount of branch history we keep. Special circuitry computes the value of y and performs the training. We discuss this circuitry in Section 5.3. When the processor encounters a branch in the fetch stage, the following steps are conceptually taken:

1. The branch address is hashed to produce an index $i \in 0..N-1$ into the table of perceptrons.
2. The i^{th} perceptron is fetched from the table into a vector register, $P_{0..n}$, of weights.
3. The value of y is computed as the dot product of P and the global history register.
4. The branch is predicted not taken when y is negative, or taken otherwise.
5. Once the actual outcome of the branch becomes known, the training algorithm uses this outcome and the output y to update the weights in P.
6. P is written back to the i^{th} entry in the table.

It may appear that prediction is slow because many computations and SRAM transactions take place in steps 1 through 5. However, Section 5.3 shows that a number of arithmetic and microarchitectural tricks enable a prediction in a single cycle. It is important to note that training occurs continously, on-line.

5.3 Implementation

This section describes details of the implementation of the perceptron predictor. We explore the design space for perceptron predictors and discuss details of the circuit-level implementation.

5.3.1 Design Space

Given a fixed hardware budget, three parameters need to be tuned to achieve the best performance: the history length, the number of bits used to represent the weights, and the threshold.

History length. Long history lengths can yield more accurate predictions [21] but also reduce the number of table entries, thereby increasing aliasing. In our experiments, the best history lengths ranged from 4 to 50, depending on the hardware budget. The perceptron predictor can use more than one kind of history. We have used both purely global history as well as a combination of global and per-branch history.

Representation of weights. The weights for the perceptron predictor are signed integers. Although many neural networks have floating-point weights, we found that integers are sufficient for our perceptrons, and they simplify the design. We find that using 8 bit weights provides the best trade-off between accuracy and hardware budget.

Threshold. The threshold is a parameter to the perceptron training algorithm that determines whether the predictor needs more training. If the magnitude of the output of the perceptron is below the threshold, or if the prediction is incorrect, then the training algorithm adjusts the perceptron weights; otherwise, the perceptron is judged to have been trained enough.

5.3.2 Circuit-Level Implementation

Here, we discuss general techniques that will allow us to implement a quick perceptron predictor, then give more detailed results of a transistor-level simulation.

Computing the Perceptron Output. Computing the output of the perceptron is on the critical path for making a branch prediction. Thus, the circuit that evaluates the perceptron should be as fast as possible. Several properties of the problem allow us to make a fast prediction. Since -1 and 1 are the only possible input values to the perceptron, multiplication is not needed to compute the dot product. Instead, we simply add when the input bit is 1 and subtract (add the two's-complement) when the input bit is -1. In practice, we have found that adding the one's-complement, which is a good estimate for the two's-complement, works just as well and lets us avoid the delay of a small carry-propagate adder in favor of a set of inverters to perform the negation. This computation is similar to that performed by multiplication circuits, which must find the sum of partial products that are each a function of an integer and a single bit. Furthermore, only the sign bit of the result is needed to make a prediction, so the other bits of the output can be computed more slowly without having to wait for a prediction. In this chapter, we report only results that simulate this complementation idea.

Training. The training algorithm of Section 5.2.2 can be implemented efficiently in hardware. Since there are no dependences between loop iterations, all iterations can execute in parallel. Since in our case both x_i and t can only be -1 or 1, the loop body can be restated as "increment w_i by 1 if $t = x_i$, and decrement otherwise," a quick arithmetic operation since the w_i are 8-bit numbers:

```
for each bit in parallel
    if $t = x_i$ then
        $w_i := w_i + 1$
    else
        $w_i := w_i - 1$
    end if
```

Circuit-Level Simulation. Using a custom logic design program and the HSPICE and CACTI 2.0 simulators we designed and simulated a hardware implementation of the elements of the critical path for the perceptron predictor for several table sizes and history lengths. We used CACTI, a cache modeling tool, to estimate the amount of time taken to read the table of perceptrons, and we used HSPICE to measure the latency of our perceptron output circuit.

The perceptron output circuit accepts input signals from the weights array and from the history register. As weights are read, they are bitwise exclusive-ORed with the corresponding bits of the history register. If the i^{th} history bit is set, then this operation has the effect of taking the one's-complement of the i^{th} weight; otherwise, the weight is passed unchanged. After the weights are processed, their sum is found using a Wallace-tree of 3-to-2 carry-save adders [15], which reduces the problem of finding the sum of n numbers to the problem of finding the sum of 2 numbers. The final two numbers are summed with a carry-lookahead adder. The Wallace-tree has depth $O(\log n)$, and the carry-lookahead adder has depth $O(\log n)$, so the computation is relatively quick. The sign of the sum is inverted and taken as the prediction.

Table 5.1 shows the delay of the perceptron predictor for several hardware budgets and history lengths, simulated with HSPICE and CACTI for 180nm process technology. We obtain these delay estimates by selecting inputs designed to elicit the worst-case gate delay. We measure the time it takes for one of the input signals to cross half of V_{DD} until the time the perceptron predictor yields a steady, usable signal. For a 4KB hardware budget and history length of 24, the total time taken for a perceptron prediction is 2.4 nanoseconds. This delay works out to slightly less than 2 clock cycles for a CPU with a clock rate of 833 MHz, the clock rate of the fastest 180 nm Alpha 21264 processor as of this writing. The Alpha 21264 branch predictor itself takes 2 clock cycles to deliver a prediction, so our predictor is within the bounds of existing technology. Note that a perceptron predictor with a history of 23 instead of 24 takes only 2.2 nanoseconds; it is about 10% faster because a predictor with 24 weights (23 for history plus 1 for bias) can be organized more efficiently than predictor with 25 weights, since decreasing the number of weights to 24 decreases the depth of the Wallace-tree by one.

5.4 Results and Analysis

We use simulations of the SPEC 2000 integer benchmarks to compare the perceptron predictor against two well-known techniques from the literature. We give results showing how an overriding version of the perceptron predictor outperforms a hybrid predictor. Finally, we present analysis to explain why the perceptron predictor performs well.

History Length	Table Size (bytes)	Table Delay (ps)	Perceptron Delay (ps)	Total Delay (ps)
4	128	386	811	1197
7	256	411	808	1219
9	512	432	725	1157
13	1K	468	1090	1558
17	2K	504	1170	1674
23	4K	571	1700	2271
24	4K	571	1860	2431

Table 5.1: Perceptron Predictor Delay.

5.4.1 Methodology

Here we describe our experimental methodology. We discuss the other predictors simulated, the benchmarks used, the tuning of the predictors, and other issues.

Predictors simulated. We compare our new predictor against *gshare* [41], and *bi-mode* [38], and a McFarling-style combination *gshare* and PAg hybrid predictor similar to that of the Alpha 21264, with all tables scaled exponentially for increasing hardware budgets. For the perceptron predictor, we simulate both a purely global predictor, as well as a predictor that uses both global and local history. This global/local predictor takes some input to the perceptron from the global history register, and other input from a set of per-branch histories; all other details of the perceptron implementation remain the same. For the global/local perceptron predictor, the extra state used by the table of local histories was constrained to be within 35% of the hardware budget for the rest of the predictor, reflecting the design of the Alpha 21264 hybrid predictor. For *gshare* and the perceptron predictors, we also simulate the *agree* mechanism [57], which predicts whether a branch outcome will agree with a bias bit set in the branch instruction. The *agree* mechanism turns destructive aliasing into constructive aliasing, increasing accuracy at small hardware budgets.

Our methodology differs from our previous work on the perceptron predictor [33] in which used traces from x86 executables of SPEC2000 and only explored global versions of the perceptron predictor. Using the Alpha instruction set, we find that the improvement yielded by the perceptron predictor over other predictors is higher than with the x86 instruction set. We believe that this is because the Alpha's RISC instruction set requires more dynamic branches to accomplish the same work, thus longer histories will be required. The perceptron predictor can make use of longer histories than other predictors.

Tuning the predictors. We tune each predictor for history length using traces gathered from the each of the 12 benchmarks and the `train` inputs. We exhaustively test every possible history length at each hardware budget for each predictor, keeping the history length yielding the lowest harmonic mean misprediction rate. For the global/local perceptron predictor, we exhaustively test each pair of history lengths such that the sum of global and local history length is at most 50. For the *agree* mechanism, we set bias bits in the branch instructions using branch biases learned from the `train` inputs.

For the global perceptron predictor, we find, for each history length, the best value of the threshold by using an intelligent search of the space of values, pruning areas of the space that give poor performance. We re-use the same thresholds for the global/local and *agree* perceptron predictors.

Table 5.2 shows the results of the history length tuning. We find an interesting relationship between history length and threshold: the best threshold θ for a given history length h is always *exactly* $\theta = \lfloor 1.93h + 14 \rfloor$. This is because adding another weight to a perceptron increases its average output by some constant, so the threshold must be increased by a constant, yielding a linear relationship between history length and threshold. Through experimentation, we determine that using 8 bits for the perceptron weights yields the best results.

Estimating area costs. Our hardware budgets do not include the cost of the logic required to do the computation. By examining die photos of hardware multipliers, we estimate that at the longest history lengths, this cost is approximately the same as that of 1K of SRAM. Using the parameters tuned for the 4K hardware budget, we estimate that the extra hardware will consume about the same logic as 256 bytes of SRAM. Thus, the cost for the computation hardware is small compared to the size of the table.

5.4.2 Impact of History Length on Accuracy

One of the strengths of the perceptron predictor is its ability to consider much longer history lengths than traditional two-level schemes, which helps because highly correlated branches can occur at a large distance from each other [21]. Any global branch prediction technique that uses a fixed amount of history information will have an optimal history length for a given set of benchmarks. As we can see from Table 5.2, the perceptron predictor works best with much longer histories than the other two predictors. For example, with a 4K byte hardware budget, *gshare* works best with a history length of 14, the maximum possible length for *gshare*. At the same hardware budget, the global perceptron predictor works best with a history length of 24.

hardware	*gshare*		global perceptron		global/local perceptron	
budget in bytes	history length	number of entries	history length	number of entries	global/local history	number of entries
128	2	512	4	25	8/2	11
256	1	1K	7	32	10/2	19
512	11	2K	9	51	23/2	19
1K	12	4K	13	73	25/5	33
2K	13	8K	17	113	31/5	55
4K	14	16K	24	163	34/10	91
8K	15	32K	28	282	34/10	182
16K	16	64K	47	348	36/11	341

Table 5.2: Best History Lengths for *gshare* and Perceptron.

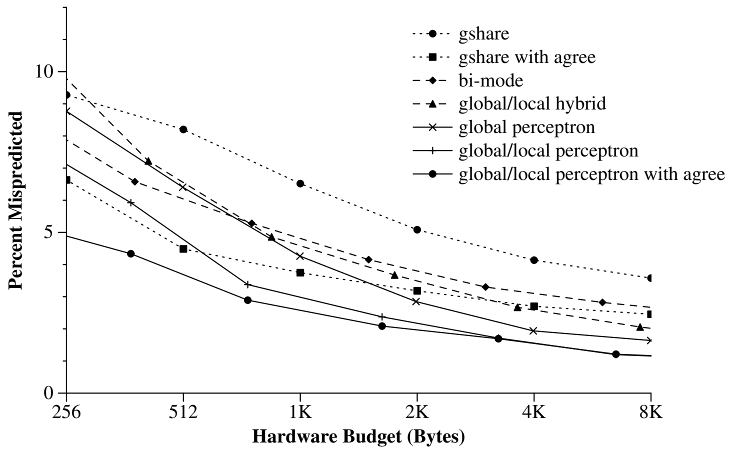
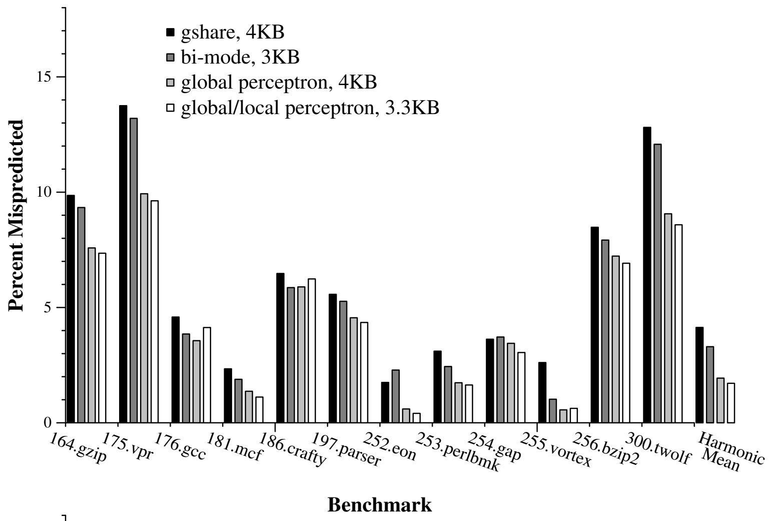


Figure 5.3: Hardware Budget vs. Misprediction Rate on SPEC 2000.

5.4.3 Misprediction Rates

Figure 5.3 shows the harmonic mean of misprediction rates achieved with increasing hardware budgets on the SPEC 2000 benchmarks. At a 4K byte hardware budget, the global perceptron predictor has a misprediction rate of 1.94%, an improvement of 53% over *gshare* at 4.13% and 31% over a 6K byte *bi-mode* at 2.82%. When both global and local history information is used, the perceptron predictor still has superior accuracy. A global/local hybrid predictor with the same configuration as the Alpha 21264 predictor using 3712 bytes has a misprediction rate of 2.67%. A global/local perceptron predictor with 3315 bytes of state has a misprediction rate of 1.71%, representing a 36% decrease in misprediction rate over the Alpha hybrid. The *agree* mechanism improves accuracy, especially at small hardware budgets. With a small budget of only 750 bytes, the global/local perceptron predictor achieves a misprediction rate of 2.89%, which is less than the misprediction rate of a *gshare* predictor with 11 times the hardware budget, and less than the misprediction rate of a *gshare/agree* predictor with a 2K byte budget. Figure 5.4 show the misprediction rates of two PHT-based methods and two perceptron predictors on the SPEC 2000 benchmarks for hardware budgets of 4K and 16K bytes.



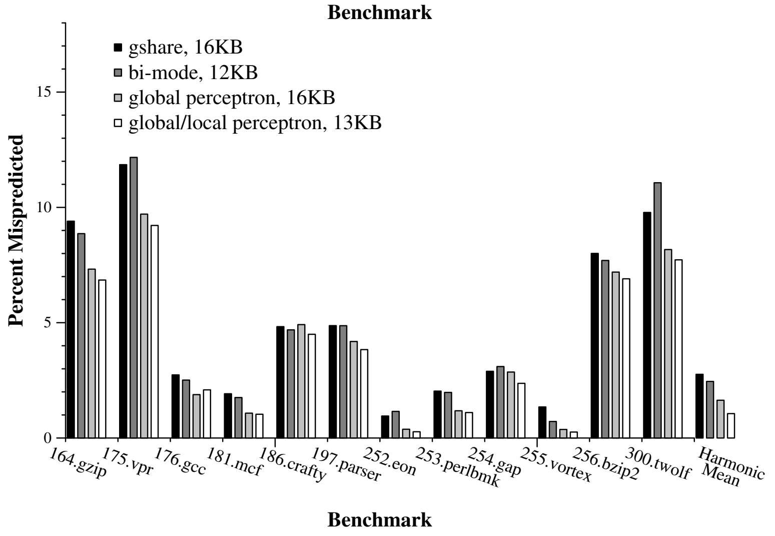


Figure 5.4: Misprediction Rates for Individual Benchmarks at 4KB and 16KB Budgets

.

Large Hardware Budgets

As Moore's Law continues to provide more and more transistors in the same area, it makes sense to explore much larger hardware budgets for branch predictors. Evers' thesis [22] explores the design space for multi-component hybrid predictors using large hardware budgets, from 18 KB to 368 KB. To our knowledge, the multi-component predictors presented in Evers' thesis are the most accurate fully dynamic branch predictors known in previous work. This predictor uses a McFarling-style chooser to choose between two other McFarling-style hybrid predictors. The first hybrid component joins a *gshare* with a short history to a *gshare* with a long history. The other hybrid component consists of a PAs hybridized with a *loop predictor*, which is capable of recognizing regular looping behavior even for loops with long trip counts.

We simulate Evers' multi-component predictors using the same configuration parameters given in his thesis. At the same set of hardware budgets, we simulate a global/local version of the perceptron predictor. The tuning of this large perceptron predictor is not as exhaustive as for the smaller hardware budgets, due to the huge design space. We tune for the best global history length on the SPEC `train` inputs, and then for the best fraction of global versus local history at a single hardware budget, extrapolating this fraction to the entire set of hardware budgets. As with our previous global/local perceptron experiments, we allocate 35% of the hardware budgets to storing the table of local histories. The configuration of the perceptron predictor is given in Table 5.3.

Size (KB)	Global History	Local History	Number of Perceptrons	Number of Local Histories
18	38	14	280	2,048
30	40	14	428	4,096
53	50	18	519	8,192
98	54	19	1093	8,192
188	64	23	1652	16,384
368	66	24	3060	32,768

Table 5.3: Configurations for Large Budget Perceptron Predictors.

Figure 5.5 shows the harmonic mean misprediction rates of Evers' multi-component predictor and the global/local perceptron predictor on the SPEC 2000 integer benchmarks. The perceptron predictor outperforms the multi-component predictor at every hardware budget, with the misprediction rates getting closer to one another as the hardware budget is increased. Both predictors are capable of reaching amazingly low misprediction rates at the 368 KB hardware budget, with the perceptron at 0.85% and the multi-component predictor at 0.93%.

We claim that our results are evidence that the perceptron predictor is the most accurate fully dynamic branch predictor known. We must point out that we have not exhaustively tuned either the multi-component or the perceptron predictors because of the huge computational challenge. Nevertheless, there is a clear separation between the misprediction rates of the multi-component and perceptron predictors, and between the perceptron and all other predictors we have examined at lower hardware budgets; thus, we are confident that our claim can be verified by independent researchers.

5.4.4 Delay Sensitive Perceptron Predictor

As we have seen, the perceptron predictor has a substantial delay associated with it. Here, we present the results of using one technique to mitigate this delay. We simulate an overriding perceptron predictor, and compare our results to the overriding hybrid branch predictor used by the Alpha 21264. Currently, the fastest Alpha processor in 180 nm technology is clocked at a rate of 833 MHz. At this clock rate, both the perceptron predictor and Alpha hybrid predictor deliver a prediction in two clock cycles.

Moderate Clock Rate Simulations

Using SimpleScalar/Alpha, we simulate a two-level overriding predictor at 833 MHz. The first level is a 256-entry Smith predictor [56], i.e., a simple one-level table of two-bit saturating counters indexed by branch address. This predictor roughly simulates the line predictor of the overriding Alpha predictor. Our Smith predictor achieves a harmonic mean accuracy of 85.2%, which is the same accuracy quoted for the Alpha line predictor [35]. For the second level predictor, we simulate both the perceptron predictor and the Alpha hybrid predictor. The perceptron predictor consists of 133 perceptrons, each with 24 weights. Although the 25 weight perceptron predictor was the best choice at this hardware budget in our simulations, the 24 weight version has much the same accuracy but is 10% faster. We have observed that the ideal ratio of per-branch history bits to total history bits is roughly 20%,

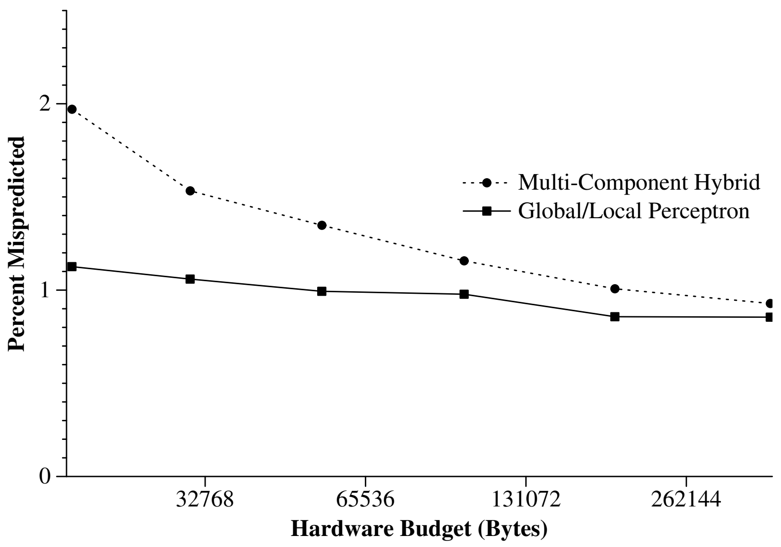


Figure 5.5: Hardware Budget vs. Misprediction Rate for Large Predictors.

so we use 19 bits of global history and 4 bits of per-branch history from a table of 1024 histories. The total state required for this predictor is 3704 bytes, approximately the same as the Alpha hybrid predictor, which uses 3712 bytes. Both the Alpha hybrid predictor and the perceptron predictor incur a single-cycle penalty when they override the Smith predictor. We also simulate a 2048-entry non-overriding *gshare* predictor for reference. This *gshare* uses less state since it operates in a single cycle; note that this is the amount of state allocated to the branch predictor in the HP-PA/RISC 8500 [39], which uses a clock rate similar to that of the Alpha. We again simulate the 12 SPEC int 2000 benchmarks, this time allowing each benchmark to execute 2 billion instructions. We simulate the 7-cycle misprediction penalty of the Alpha 21264.

When a branch is encountered, there are four possibilities with the overriding predictor:

- The first and second level predictions agree and are correct. In this case, there is no penalty.
- The first and second level predictions disagree, and the second one is correct. In this case, the second predictor overrides the first, with a small penalty.
- The first and second level predictions disagree, and the second one is incorrect. In this case, there is a penalty equal to the overriding penalty from the previous case as well as the penalty of a full misprediction. Fortunately, the second predictor is more accurate that the first, so this case is less frequent.
- The first and second level predictor agree and are both incorrect. In this case, there is no overriding, but the prediction is wrong, so a full misprediction penalty is incurred.

Figure 5.6 shows the instructions per cycle (IPC) for each of the predictors. The figure shows the IPCs yielded by *gshare*, an Alpha-like hybrid, and global/local perceptron predictor given a 7-cycle misprediction penalty. The hybrid and perceptron predictors have a 2-cycle latency, and are used as overriding predictors with a small Smith predictor. Even though there is a penalty when the overriding Alpha and perceptron predictors override the Smith predictor, their increased accuracies more than compensate for this effect, achieving higher IPCs than a single-cycle *gshare*. The perceptron predictor yields a harmonic mean IPC of 1.65, which is higher than the overriding predictor at 1.59, which itself is higher than *gshare* at 1.53.

Higher Clock Rates

The current trend in microarchitecture is to deeply pipeline microprocessors, sacrificing some IPC for the ability to use much higher clock rates. For instance, the Intel Pentium 4 uses a 20-stage integer pipeline at a clock rate of 1.76

GHz, as of this writing. In this situation, one might expect the perceptron predictor to yield poor performance, since it requires so much time to make a prediction relative to the short clock period. Nevertheless, we will show that the perceptron predictor can improve performance even more than in the previous case.

At a 1.76 GHz clock rate, the perceptron predictor described above would take four clock cycles: one to read the table of perceptrons and three to propagate signals to compute the perceptron output. Pipelining the perceptron predictor will allow us to get one prediction each cycle, so that branches that come close together do not have to wait until the predictor is finished predicting the previous branch. The Wallace-tree for this perceptron has 7 levels. With a small cost in latch delay, we can pipeline the Wallace-tree in four stages: one to read the perceptron from the table, another for the first three levels of the tree, another for the second three levels, and a fourth for the final level and the carry-lookahead adder at the root of the tree. The new perceptron predictor operates as follows:

1. When a branch is encountered, it is immediately predicted with a small Smith predictor. Execution continues along the predicted path.
2. Simultaneously, the local history table and perceptron tables are accessed using the branch address as an index.
3. The circuit that computes the perceptron output takes its input from the global and local history registers and the perceptron weights.
4. Four cycles after the initial prediction, the perceptron prediction is available. If it differs from the initial prediction, instructions executed since that prediction are squashed and execution continues along the other path.
5. When the branch executes, the corresponding perceptron is quickly trained and stored back to the table of perceptrons.

Figure 5.7 shows the result of simulating predictors in a microarchitecture with characteristics of the Pentium 4. The misprediction penalty is 20, to simulate the long pipeline of the Pentium 4. The Alpha overriding hybrid predictor is conservatively scaled to take 3 clock cycles, while the overriding perceptron predictor takes 4 clock cycles. The 2048-entry *gshare* predictor is unmodified. Even though the perceptron predictor takes longer to make a prediction, it still yields the highest IPC in all benchmarks because of its superior accuracy. The perceptron predictor yields an IPC of 1.48, which is 5.7% higher than that of the hybrid predictor at 1.40, and 15.8% higher than the baseline IPC of 1.28 yielded by *gshare*.

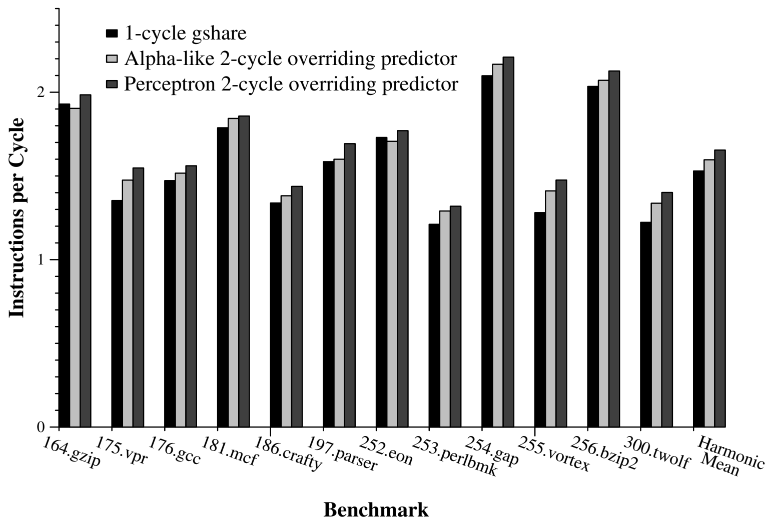


Figure 5.6: IPC for overriding perceptron and hybrid predictors.

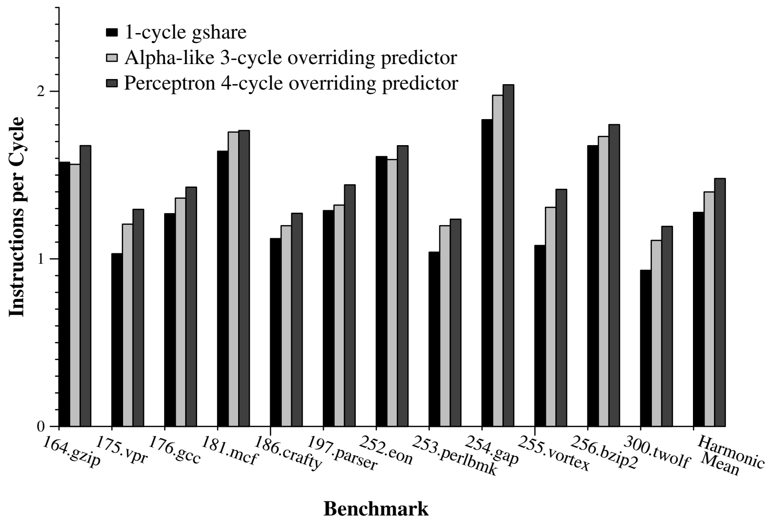


Figure 5.7: IPC for overriding perceptron and hybrid predictors with long pipelines.

5.4.5 Training Times

To compare the training speeds of the perceptron predictor with PHT methods, we examine the first 100 times each branch in each of the SPEC 2000 benchmarks is executed (for those branches executing at least 100 times). Figure 5.8 shows the average accuracy of each of the 100 predictions for each of the static branches. The x axis is the number of times a branch has been executed. The y-axis is the average, over all branches in the program, of 1 if the branch was mispredicted, 0 otherwise. The average is weighted by the relative frequencies of each branch. Over time, this statistic tracks how quickly each predictor learns. The perceptron predictor achieves greater accuracy earlier than the other two methods.

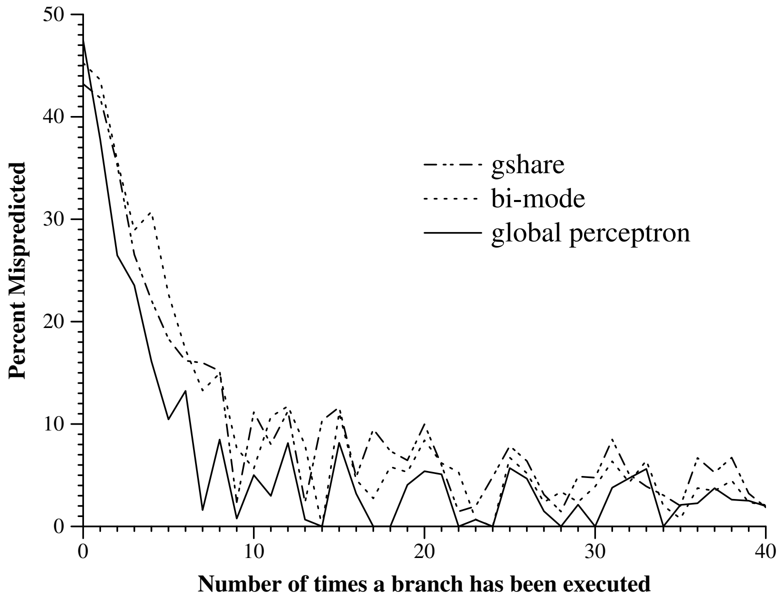


Figure 5.8: Average Training Times for SPEC 2000 benchmarks.

The perceptron method learns more quickly the *gshare* or *bi-mode*. For the perceptron predictor, training time is independent of history length. For techniques such as *gshare* that index a table of counters, training time depends on the amount of history considered; a longer history may lead to a larger working set of two-bit counters that must be initialized when the predictor is first learning the branch. This effect has a negative impact on prediction rates, and at a certain point, longer histories begin to hurt performance for these schemes [42]. As we will see in the next section, the perceptron prediction does not have this weakness, as it always does better with a longer history length.

5.4.6 Why Does it Do Well?

We hypothesize that the main advantage of the perceptron predictor is its ability to make use of longer history lengths. Schemes like *gshare* that use the history register as an index into a table require space exponential in the history length, while the perceptron predictor requires space linear in the history length.

To provide experimental support for our hypothesis, we simulate *gshare* and the perceptron predictor at a 64K hardware budget, where the perceptron predictor normally outperforms *gshare*. However, by only allowing the perceptron predictor to use as many history bits as *gshare* (18 bits), we find that *gshare* performs better, with a misprediction rate of 1.86% compared with 1.96% for the perceptron predictor. The inferior performance of this crippled predictor has two likely causes: there is more destructive aliasing with perceptrons because they are larger, and thus fewer, than *gshare*'s two-bit counters, and perceptrons are capable of learning only linearly separable functions of their input, while *gshare* can potentially learn any Boolean function.

Figure 5.9 shows the result of simulating *gshare* and the perceptron predictor with varying history lengths on the SPEC 2000 benchmarks. Here, we use a 4M byte hardware budget to allow *gshare* to consider longer history lengths than usual. As we allow each predictor to consider longer histories, each becomes more accurate until *gshare* becomes worse and then runs out of bits at a history length of 23 (since *gshare* requires resources exponential in the number of history bits), while the perceptron predictor continues to improve. With this unrealistically huge hardware budget, *gshare* performs best with a history length of 23, where it achieves a misprediction rate of 1.55%. The perceptron predictor is best at a history length of 66, where it achieves a misprediction rate of 1.09%.

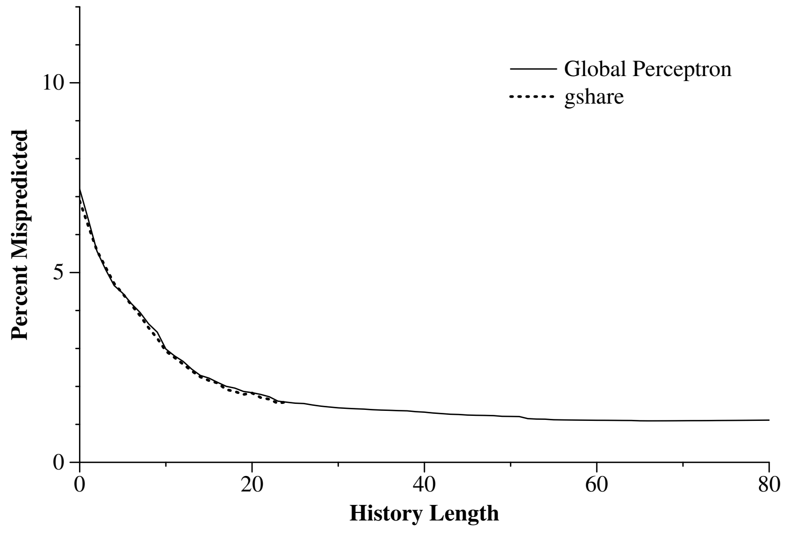


Figure 5.9: History Length vs. Performance.

5.4.7 When Does It Do Well?

The perceptron predictor does well when the predicted branch exhibits *linearly separable behavior.* To define this term, let h_n be the most recent n bits of global branch history. For a static branch B, there exists a Boolean function $f_B(h_n)$ that best predicts B's behavior. It is this function, f_B, that all branch predictors strive to learn. If f_B is not linearly separable then *gshare* may predict B better than the perceptron predictor, and we say that such branches are *linearly inseparable*. We compute $f_B(h_{14})$ for each static branch B for each benchmark and test for linear separability of the function.

Figure 5.10 shows the misprediction rates for each benchmark for a 4KB budget, as well as the percentage of dynamically executed branches that is linearly inseparable. For each benchmark, the bar on the left shows the misprediction rate of *gshare*, while the bar on the right shows the misprediction rate of a global perceptron predictor. Each bar also shows, using shading, the portion of mispredictions due to linearly inseparable branches and linearly separable branches. We observe two interesting features of this chart. First, most mispredicted branches are linearly inseparable, thus linear inseparability correlates highly with unpredictability in general. Second, the perceptron predictor outperforms *gshare* in all cases except for that of `186.crafty`, the benchmark with the lowest fraction of linearly separable branches.

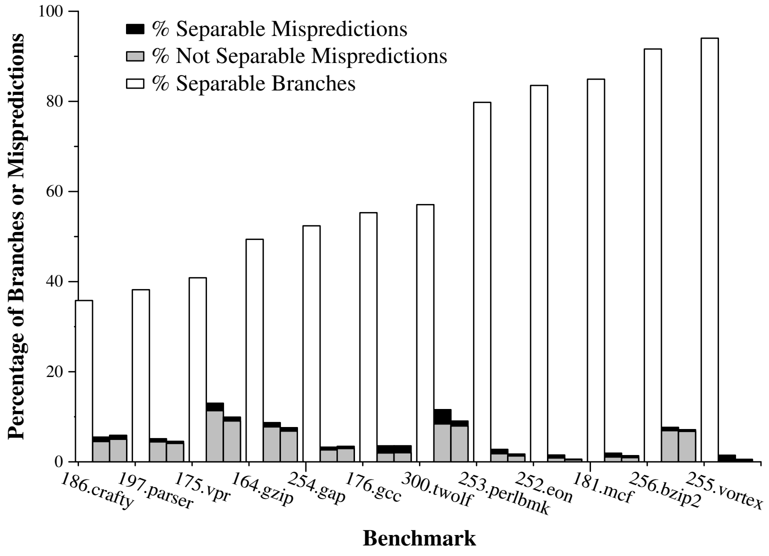


Figure 5.10: Linear Separability vs. Accuracy at a 4KB budget.

Some branches require longer histories than others for accurate prediction, and the perceptron predictor often has an advantage for these branches. Figure 5.11 shows the relationship between this advantage and the required history length, with one curve for linearly separable branches and one for inseparable branches. The y axis represents the advantage of our predictor, computed by subtracting the misprediction rate of the perceptron predictor from that of *gshare*. We sorted all static branches according to their "best" history length, which is represented on the x axis. Each data point represents the average misprediction rate of static branches (without regard to execution frequency) that have a given best history length. We use the perceptron predictor in our methodology for finding these best lengths: Using a perceptron trained for each branch, we find the most distant of the three weights with the greatest magnitude. This methodology is motivated by the work of Evers *et al.*, who show that most branches can be predicted by looking at three previous branches [21]. As the best history length increases, the advantage of the perceptron predictor generally increases as well. We also see that our predictor is more accurate for linearly separable branches. For linearly inseparable branches, our predictor performs generally better when the branches require long histories, while *gshare* sometimes performs better when branches require short histories.

Linearly inseparable branches requiring longer histories, as well as all linearly separable branches, are always predicted better with the perceptron predictor. Linearly inseparable branches requiring fewer bits of history are predicted better by gshare. Thus, the longer the history required, the better the performance of the perceptron predictor, even on the linearly inseparable branches.

We found this history length by finding the most distant of the three weights with the greatest magnitude in a perceptron trained for each branch, an application of the perceptron predictor for analyzing branch behavior.

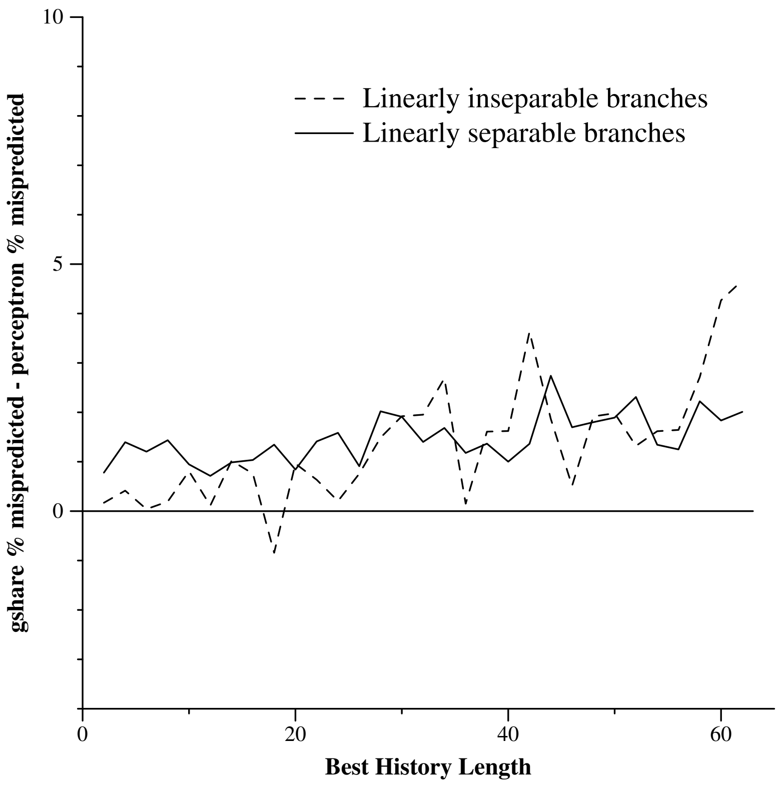


Figure 5.11: Classifying the Advantage of the Perceptron Predictor.

5.4.8 Additional Advantages of the Perceptron Predictor

This subsection describes two additional benefits of using perceptrons to perform branch prediction.

Branch prediction with perceptrons has other advantages over previous methods. A perceptron output can give a confidence in the prediction. The weight vector can be used to find correlations between branches, so this method can be used in simulation to analyze the behavior of a program.

Assigning confidence to decisions. Our predictor can provide a confidence-level in its predictions that can be useful in guiding hardware speculation. The output, y, of the perceptron predictor is not a Boolean value, but a number that we interpret as *taken* if $y \geq 0$. The value of y provides important information about the branch since the distance of y from 0 is proportional to the *certainty* that the branch will be taken [30]. This confidence can be used, for example, to allow a microarchitecture to speculatively execute both branch paths when confidence is low, and to execute only the predicted path when confidence is high. Some branch prediction schemes explicitly compute a confidence in their predictions [29], but in our predictor this information comes for free. We have observed experimentally that the probability that a branch will be taken can be accurately estimated as a linear function of the output of the perceptron predictor. Figure 5.12 shows an emprical measurement of the sample probability that a branch is taken as a function of the perceptron output for the SPEC int 2000 benchmarks.

Analyzing branch behavior with perceptrons. Perceptrons can be used to analyze correlations among branches. The perceptron predictor assigns each bit in the branch history a weight. When a particular bit is strongly correlated with a particular branch outcome, the magnitude of the weight is higher than when there is less or no correlation. Thus, the perceptron predictor learns to recognize the bits in the history of a particular branch that are important for prediction, and it learns to ignore the unimportant bits. This property of the perceptron predictor can be used with profiling to provide feedback for other branch prediction schemes. For example, our methodology in Section 5.4.7 could be used with a profiler to provide path length information to the variable length path predictor [58].

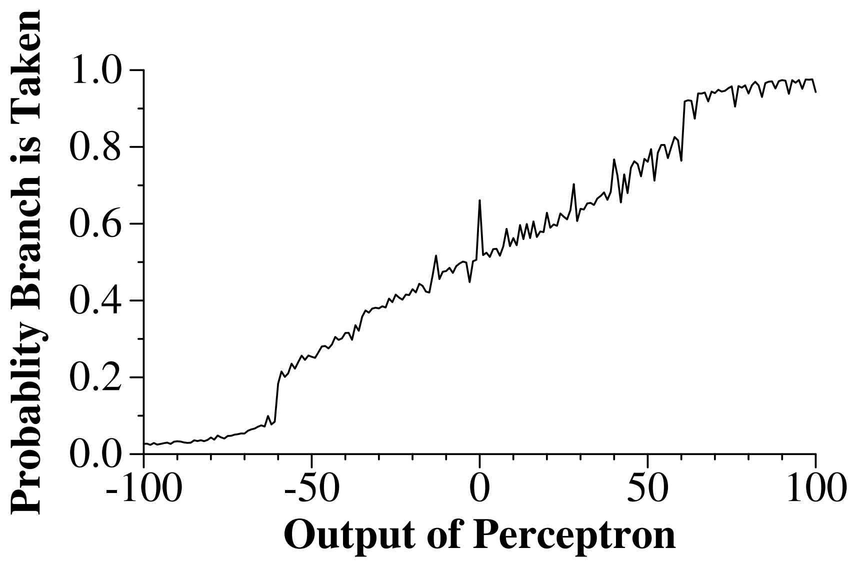


Figure 5.12: Probability Branch is Taken as a Function of Perceptron Output

5.5 Summary

In this chapter we have introduced a new branch predictor that uses neural networks—the perceptron in particular—as the basic prediction mechanism. Perceptrons are attractive because they can use long history lengths without requiring exponential resources. A potential weakness of perceptrons is their increased computational complexity when compared with two-bit counters, but we have shown how a perceptron predictor can be implemented efficiently by using delay-hiding hierarchical organizations. Another weakness of perceptrons is their inability to learn linearly inseparable functions. Nevertheless, the perceptron predictor performs well, achieving a lower misprediction rate, at all hardware budgets, than well-known global predictors on the SPEC 2000 integer benchmarks. Branches exhibiting linearly inseparability are hard to predict in general, not just hard for perceptrons.

In the Introduction, the reader may have momentarily worried that the era of proposals for increasingly complex branch predictors is over. The reader can rest assured that, with hierarchical organizations, researchers are free to explore more expensive and exotic solutions to the problem of increasing branch predictor accuracy. Nevertheless, we must ask ourselves whether this approach can be sustained indefinitely, and whether there are simpler ideas that address branch predictor delay and accuracy without increasing the complexity that the microarchitect has to deal with. In the next two chapters, we explore alternative ideas that address delay by reducing complexity, rather than increasing it.

Chapter 6

Cooperative Prediction with Branch Path Re-Aliasing

We have seen how to mitigate the problem of branch predictor delay by using more complex hardware. However, it seems intuitive that we would rather use *less* hardware, because branch predictor delay is due in large part to the propagation delay of signals through complex circuitry and long wires. In this chapter and the next chapter, we explore the idea of shifting some of the work of making a prediction to the compiler, so that the compiler and processor cooperate to make the prediction. By reducing the complexity of the hardware, these cooperative predictors have reduced access delay.

Traditional predictors such as *gshare* [41], *bi-mode* [38], YAGS [18], and hybrid predictors [20] reduce destructive aliasing in the PHT by introducing more levels of logic onto the critical path for making a prediction; with aggressive clock rates, these branch predictors will become less feasible.

In this chapter, we propose *branch path re-aliasing*, a branch prediction technique which enlists the compiler's help in moving important functionality off of the critical path to making a prediction, providing a quick prediction in a single cycle while moving other prediction work to the less critical predictor update stage. In particular, our scheme gives the compiler the task of decreasing destructive aliasing and increasing constructive aliasing, so that the branch predictor hardware can be simplified. While other approaches have used the compiler to provide hints which decrease aliasing, this scheme is unique in that the hint bits are kept off the critical path for prediction. Branch path re-aliasing is limited in scope to branch predictors that use GAg, i.e., a simple PHT indexed only by the global history, as the prediction mechanism.

In our scheme, the compiler uses path profiling information to provide hints to branch instructions so that paths with different outcomes will have histories that map to different locations in the branch predictor's tables. For our purposes, a path through the program is a sequence of conditional branch executions up to a certain length; path profiling is a technique that keeps a count of the number of times each path through the program is executed. A small, simple predictor is used to make a branch prediction, after which the branch history is updated so that destructive aliasing is decreased. Our scheme places a *branch inversion bit* in each branch instruction to indicate whether the branch outcome should be inverted before it is recorded in the global history register. Even in CPUs with multi-cycle instruction caches, our scheme can deliver a prediction in parallel with the instruction cache access, and only needs to read the hint bit to update the branch predictor.

Our simulations show that a 2048-entry GAg predictor enhanced with branch path re-aliasing has a misprediction rate of 6.5%, 21% lower than the misprediction rate of 8.2% for the same sized, but more complicated, *gshare* predictor, and equivalent to the misprediction rate of a *gshare* predictor with twice the size. We also show that our predictor can improve accuracy for other PHT-based predictors.

In this chapter we introduce the concept of branch path re-aliasing. We discuss the motivation behind this idea, discuss the problem of aliasing in branch predictors, describe our optimization and algorithm, and provide results showing how our scheme improves accuracy.

6.1 Branch Path Re-Aliasing

In this section, we describe the problem of history aliasing, which is common to many two-level branch predictors. We then describe a technique that increases accuracy by decreasing aliasing.

6.1.1 Path and Outcome Histories

Branch path re-aliasing gives the compiler explicit control over how paths through the program are mapped to PHT entries. Branch outcomes are highly correlated both with path and pattern histories [43, 65, 58]. Pattern histories are easier to use than path histories since they require recording only a single bit for each branch. However, pattern histories are highly susceptible to aliasing, both between different static branches and within the same branch. That is, several different paths correlated with different branch behaviors may all induce the same pattern history, leading to destructive aliasing. Our optimization *re-aliases* pattern histories to better reflect path histories, improving accuracy by decreasing destructive aliasing.

6.1.2 History Aliasing in a Global Predictor

Several types of aliasing have been identified in branch predictors [42]. Our focus is on *conflict aliasing*. Consider a GAg predictor, which consists of a PHT indexed by a global history register. Two different paths in the program may coincidentally lead to the same global history, even though the code being executed is unrelated. In this case, the same PHT entry will be used for both branches, but the prediction may not correlate highly with the outcome of either. Thus, the branch predictor will have poor accuracy for these branches.

6.1.3 Our Solution: Branch Path Re-Aliasing

Our approach to solving the history aliasing problem is to insert a hint bit into each instruction that tells the branch history update mechanism whether or not to invert the branch outcome before recording it in the history register. We choose the hint bits, which we call *inversion bits*, such that paths leading to branches with opposite outcomes will have different histories. Essentially, by changing the way paths alias one another in the PHT, we reduce destructive aliasing.

We introduce our idea by modifying the simplest possible two-level branch predictor: the GAg. A global history register is used to index a PHT of two-bit saturating counters, from which the prediction is directly read. Once the prediction is read and made available to the fetch engine, the critical time to make a prediction is over, so the predictor is no slower than a normal GAg. The branch prediction is then used to speculatively update the global history register, which is backed up and corrected after a misprediction. With branch path re-aliasing, the difference comes in how the history register is updated. Each branch instruction encodes an inversion bit. If this bit is set, then the branch outcome is inverted before it is recorded in the global history register. In short, the value recorded in the history register is the exclusive-OR of the inversion bit and the branch outcome.

At first glance, it might seem that this technique could be implemented by simply changing branch senses and reordering code; however, this transformation would be at odds with techniques such as branch alignment [13] that seek to minimize the number of taken branches to increase fetch bandwidth. Branch alignment can increase performance, even though it may decrease prediction accuracy [47]. Our technique can nicely complement branch alignment by decreasing the destructive aliasing introduced by alignment.

Path Profiles

Path profiling collects information on the the execution paths of a program [7]. Branch path re-aliasing uses path profiles to determine which branches should have their inversion bits set. For a history length of N, i.e., a GAg with an N-bit history, each path profile stores the following information for a path p:

1. The addresses of the last N branches encountered.
2. The outcomes (*taken* or *not taken*) of the last N branches encountered.
3. $freq(p)$, the frequency with which this path was executed.

4. *ntaken*(p), the number of times this path led to a taken branch.

Algorithm

Once the path profiles have been collected, we use a two-phase algorithm to set inversion bits. In the first phase, the algorithm tries to map paths to PHT entries by setting the inversion bits of certain branches, causing constructive aliasing between paths that agree on branch outcome and choosing different PHT entries for paths with different outcomes. Each path is examined in decreasing order of execution frequency. For each path, we choose a set of inversion bits that either map the path to PHT to which similarly-biased paths are mapped, or to an unused PHT entry. As inversion bits are set, they become fixed for paths that are examined later; thus, this greedy local algorithm is augmented with a second phase that considers the global situation. In the second phase, a hill-climbing heuristic sets the inversion bits of each branch sense one at a time, keeping the set of inversion bits that maximizes a fitness function based on the estimated amount of constructive and destructive interference. The details the two phases are as follows:

1. The first phase of the algorithm maps paths to PHT entries by inverting or not inverting branches along the path. The algorithm considers each path profile in descending order of frequency. For each profile p, the algorithm looks for an entry i in the PHT to which similarly biased paths are mapped, or to which no paths are mapped at all. If one is found, then path p is mapped to PHT entry i; otherwise, the inversion bits of the path p are left the same.

2. The second phase considers each static branch, choosing the inversion hint bit for that branch that maximizes a fitness function over all branches. Let P_i be the set of paths all mapped to PHT entry i, and let n be the history length, so that there are 2^n counters in the PHT. Let a Boolean *taken*$_i$ be the aggregate bias (i.e. true for *taken* or false for *not taken*) of all the paths mapped to PHT entry i, i.e., *taken*$_i$ is true if and only if:

$$\sum_{p \in P_i} \mathit{ntaken}(p) \geq \frac{1}{2} \sum_{p \in P_i} \mathit{freq}(p)$$

   In other words, *taken*$_i$ is true if and only if all the paths mapped to PHT entry i lead to taken branches at least half the time. For a path p, let a Boolean *bias*$_p$ be true if and only if $\mathit{ntaken}(p) \geq \mathit{freq}(p)/2$, i.e., *bias*$_p$ is the bias of an individual path. Then the value of the fitness function is:

$$\sum_{0 \leq i \leq 2^h} \sum_{p \in P_i} \begin{cases} \mathit{freq}(p) & \text{if } taken_i = bias_p \\ -\mathit{freq}(p) & \text{otherwise} \end{cases}$$

   Each path is mapped to a particular PHT entry. Intuitively, the fitness function is the sum, over all paths, of the frequencies of paths mapped to PHT entries with the same bias, minus the frequencies of paths mapped to PHT entries with different bias. The higher the fitness function, the more constructive and less destructive interference there is.

6.1.4 Implementing Inversion Bits

An important consideration for branch path re-aliasing is the representation of the inversion bits. Each branch instruction encodes an inversion bit, which is reasonable since several existing ISAs already dedicate one or two bits in each branch instruction to managing branch prediction. For example, the HP/PA-RISC architecture allows each branch to encode a bias bit [39], which is used either for static or *agree* branch prediction. The Pentium 4 microprocessor extends the IA-32 instruction set to include branch hints [28]. The IA-64 architecture encodes several hint bits in branch instructions [25]. These extra bits in the ISA could be re-used to represent inversion bits. Old binaries would still run with reduced performance, and newer ones could be optimized to use the inversion bits for branch path re-aliasing.

6.2 Results and Analysis

In this section, we give the results of branch path re-aliasing on the SPEC 2000 integer benchmarks, measuring the decrease in misprediction rates on several branch predictors. We show that our optimization also helps more complex *agree* and hybrid predictors. Finally, we measure the decrease in aliasing responsible for the improved accuracy.

6.2.1 Predictor Simulation Methodology

We use the `train` inputs for collecting the path profiles, and we use the `ref` inputs to evaluate the accuracy of the predictors. We use traces to gather our path profiles. This method is costly, but there are techniques in the literature that would make this task much more efficient, for example, the efficient algorithm of Young [66], which gathers bounded-length paths with both forward and backward edges, or the forward-path profiling of Ball and Larus [7]. We consider path profiles with history lengths of 8 to 15.

We use branch path re-aliasing to decrease the misprediction rates of three dynamic branch predictors: GAg, an *agree* predictor, and a hybrid predictor. We compare our improved predictors with several other predictors. We first tune each predictor for optimal history length using the traces collected with the `train` inputs.

6.2.2 Algorithm Implementation

We measure misprediction rates using a trace-driven simulation program. For our simulations, we use a 733MHz Pentium III that reads compressed traces from an NFS server. On this machine, the branch path re-aliasing algorithm takes from 5 to 30 minutes, depending on the history length, number of paths in the program, and compression ratios of the traces. We did not pay particular attention to the efficiency of the program, using C++ and STL for rapid development. If profiling performance becomes a problem in a production version of branch path re-aliasing, Young's more complex path profiling algorithm could be used to provide greater efficiency.

6.2.3 How *not* to do Branch Path Re-Aliasing

Before we go on to our main results, we will explore two obvious but unwise ideas that may come to mind when implementing branch path re-aliasing. We describe them here, and explain why, although they may seem like good ideas at first glance, they are not.

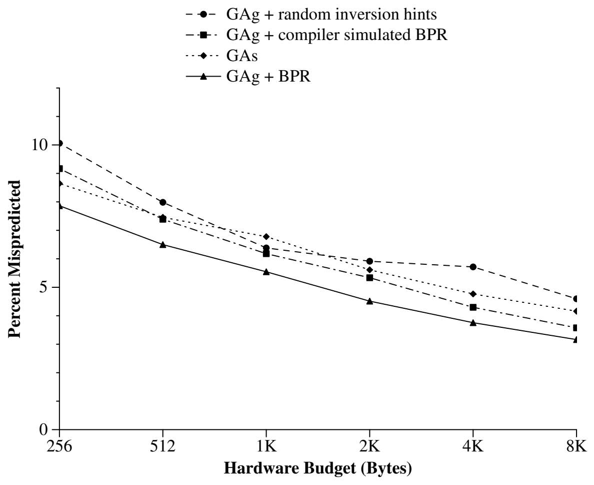


Figure 6.1: Misprediction rates for alternate implementations of branch path re-aliasing, along with GAs.

Random Inversion Bits

One idea is to set the inversion bits randomly. Branch histories come from a very non-uniform distribution. The *gshare* predictor uses the branch address as a way to more randomly distribute accesses among the PHT. Perhaps we could use random inversion bits to achieve the same effect. If this works, then maybe the inversion bits would not be needed in the ISA at all, but could be derived from, say, the bits in the branch address. Figure 6.1 shows the results of simulating GAg with pseudo-random inversion bits over a range of hardware budget, along with GAs and GAg with branch path re-aliasing. At almost every hardware budget, but especially at the small ones with which we are most concerned, this technique yields the poorest misprediction rates. Without using some intelligence in setting the inversion bits, this technique fails.

Simulating Inversion Bits with Branch Senses

Another way to implement branch path re-aliasing without inversion bits in the ISA is to use branch senses to simulate inversion bits. For instance, if we would normally set the inversion bit for a "branch if zero" branch, we would instead change the branch to a "branch if not zero" branch and re-order the basic blocks in the code to maintain the correct program semantics. Thus, it would seem that our optimization could be used with an unmodified GAg predictor in existing hardware, such as the GAg component of the Alpha 21264.

Figure 6.1 shows the misprediction rates resulting from simulating this idea. We modified the branch path re-aliasing algorithm to take into account the fact that the actual predictions were changing, not just the contents of the history register, so that the fitness function would provide meaningful inversion hints for changing branch senses. Again, this technique performs more poorly than GAs. Our regular branch path re-aliasing technique only changes the distribution of the bits in the global history register, to distribute accesses to the PHT such that aliasing is avoided. Changing branch senses also changes the distribution of the values in the PHT itself, introducing many more degrees of freedom to the problem and fundamentally increasing the complexity of finding an optimal solution.

There is a much more compelling reason not to do branch path re-aliasing by changing branch senses. On many microarchitectures, taken branches incur a penalty, since instruction fetch usually cannot be accomplished across a taken branch, and instruction cache misses are more likely when there are non-sequential accesses. Changing branch senses to increase predictor accuracy is at odds with code-reordering optimizations such as branch alignment [13] that try to minimize the number of taken branches. Even though code-reordering sometimes results in slightly reduced predictor accuracy [47], performance increases overall because there are fewer taken branches. Indeed, our hardware version of branch path re-aliasing might help regain some of the lost accuracy and complement code-reordering transformations.

Our conclusion from this subsection is that the best approach to implementing branch path re-aliasing is to use inversion bits and extra hardware in the microprocessor, and extra intelligence in the compiler.

6.2.4 Simple Two-Level Predictors

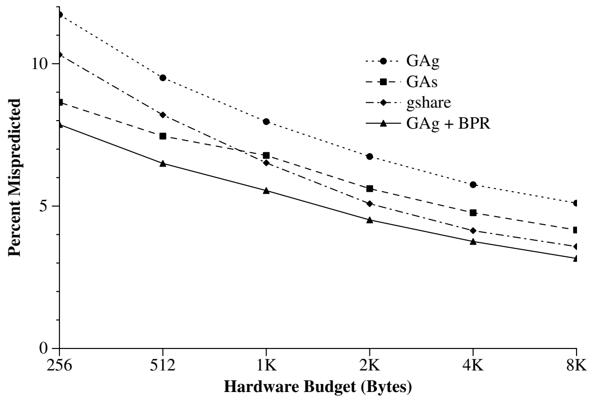


Figure 6.2: Branch misprediction rates on the SPEC 2000 integer benchmarks.

Figure 6.2 compares our basic scheme, GAg with branch path re-aliasing, against three simple two-level predictors: GAg, GAs, and *gshare*. The graph shows misprediction rates for hardware budgets ranging from 256 to 8K bytes. At all hardware budgets, our basic scheme achieves the lowest misprediction rate. The graph does not show, of course, that our scheme allows faster clocking by removing work from the critical path. For a branch predictor with 2K-entries, the same hardware budget used in the AMD Athlon, branch path re-aliasing reduces the misprediction of GAg by 32%, from 9.5% down to 6.5%. The misprediction rates for a 2K-entry GAs and *gshare* are 7.5% and 8.2%, respectively; for 2K-entries, our basic predictor sees misprediction rates that are lower than GAs and *gshare* by 13% and 21%, respectively.

To see how these numbers might be used to design future predictors, suppose the microarchitects of a CPU core that uses a 4K-entry GAs predictor decided it was necessary to shrink the branch predictor to 2K entries to allow for more aggressive clocking. Our simulations show that the misprediction rate would increase by 12%, from 6.7% to 7.5%. Instead, the microarchitects could replace the 4K-entry GAs with a 2K-entry GAg and provide inversion bits. Branch path re-aliasing could achieve a misprediction rate of 6.5%, decreasing the misprediction rate of the larger predictor by 3%.

6.2.5 More Complex Predictors

We have argued that high-latency, complex predictors will become less feasible as clock rates increase and pipelines get longer. Nevertheless, some CPU designs will continue to keep shorter pipelines and less aggressive clock rates. Even with more complex predictors, branch path re-aliasing offers higher accuracy.

Agree Predictors

The *agree* predictor achieves increased accuracy by turning the destructive aliasing of a normal PHT predictor into constructive aliasing. Rather than predicting the outcome of a branch, the PHT is used to predict whether the outcome will agree with a bias bit. Still, there is a different kind of destructive aliasing to which *agree* predictors are susceptible. Instead of paths that lead to taken and not taken branches colliding in the PHT, we may have paths that lead to agreement and disagreement with the bias bit aliasing each other. We modify the branch path re-aliasing algorithm to reduce aliasing in a GAg-based *agree* predictor that uses bias bits set in each branch instruction. Instead of keeping track of the taken/not taken bias of a particular path, the new algorithm keeps track of the agree/disagree bias of a path. That is, for each PHT entry, the algorithm determines whether each path leading to that entry usually agrees or disagrees with the corresponding bias bit.

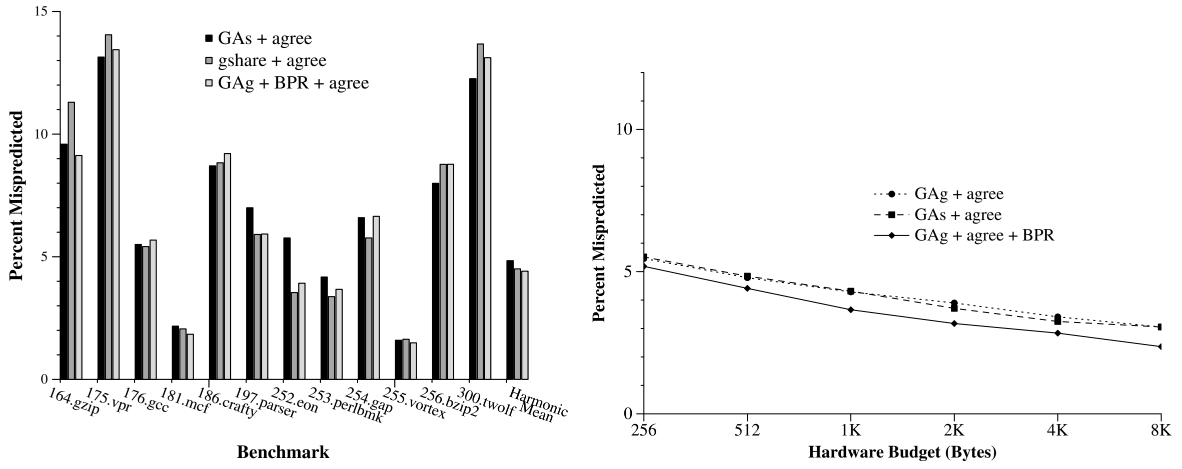


Figure 6.3: Branch misprediction rates on each SPEC 2000 integer benchmarks for *agree* predictors.

Figure 6.3 shows harmonic means of misprediction rates for several hardware budgets, as well as the misprediction rates on each SPEC integer benchmark means for 2K-entry GAs, *gshare*, and GAg predictors with branch path re-aliasing, each using the *agree* mechanism. These predictors use the same size table as the *agree* predictor of recent HP-PA/RISC cores such as the 8700 [39, 59]. Branch path re-aliasing achieves the lowest harmonic mean misprediction rate of 4.4%, compared with 4.8% for GAs with *agree* and 4.5% for *gshare* with *agree*.

Although reading the bias bit is still on the critical path for making a prediction with branch path re-aliasing, reading address bits and propagating values through exclusive-OR gates is not.

Hybrid Predictors

One of the components of the Alpha 21264 hybrid branch predictor is a 4K-entry GAg predictor. The choice predictor, which predicts whether the global or per-branch component will be more accurate, is also a 4K-entry table of 2-bit counters indexed by the global branch history. We modified the branch path re-aliasing program to measure the bias of a particular branch to be predicted better by a global or per-branch predictor by tracking the misprediction rates of both prediction components. We modified the fitness function to take into account both taken/not taken and global/per-branch biases. This way, aliasing is reduced both in the global PHT as well as in the choice table.

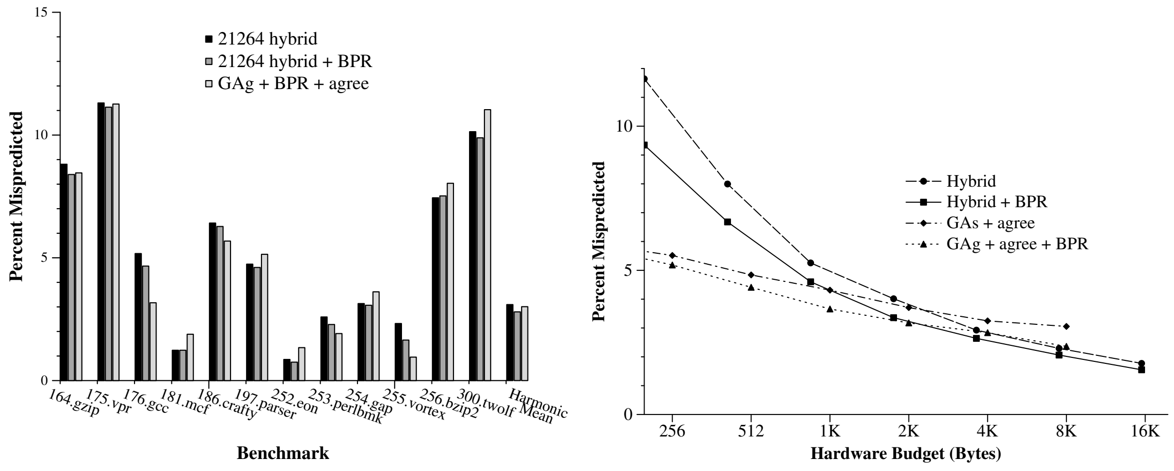


Figure 6.4: Branch misprediction rates on each SPEC 2000 integer benchmarks for hybrid and *agree* predictors.

We simulate the unmodified Alpha 21264 hybrid predictor, as well as a version of the Alpha predictor augmented with branch path re-aliasing. We allow the global and chooser PHTs to range in size from 256 to 32K entries, scaling the per-branch table of histories and PHT with 1/4 the entries as the global PHT, yielding a sequence of Alpha-like predictors at increasing hardware budgets. Figure 6.4 shows a plot of the harmonic means of misprediction rates as a function of hardware budget for the hybrid predictors as well as two *agree* predictors, one with branch path re-aliasing. Figure 6.4 also shows a bar graph for the 4K-entry global PHT versions of the hybrid predictors, using the same configuration as the Alpha 21264 predictor. The bargraph shows a 16K-entry *agree* predictor using branch path re-aliasing. This *agree* predictor uses about the same hardware budget (4096 bytes) available to the Alpha 21264 (3712 bytes). Using branch path re-aliasing with the hybrid predictor reduces the harmonic mean of the misprediction rate by 10%, from 3.1% to 2.8%. Using GAg with branch path re-aliasing and the *agree* mechanism, the misprediction rate is 3.0%, slightly better than the original hybrid predictor and with reduced complexity.

6.2.6 Aliasing Rates

The purpose of branch path re-aliasing is to reduce destructive aliasing in the PHT for a GAg predictor. In our experiments, we model a "de-aliased" predictor, i.e., a predictor where different paths cannot alias the same PHT entries. We use this predictor to measure three kinds of aliasing [42]:

- Destructive aliasing occurs when PHT aliasing leads to a misprediction in GAg where the de-aliased predictor has no misprediction.

- Constructive aliasing occurs when PHT aliasing leads to a correct prediction where the de-aliased predictor mispredicts.
- Harmless aliasing occurs when aliasing in the PHT has no effect on whether or not a prediction is correct.

Note that these cases are mutually exclusive and account for all aliasing in the PHT. Figure 6.5 shows these different types of aliasing rates in a 2K-entry GAg predictor for the SPEC 2000 integer benchmarks, before and after applying branch path re-aliasing. The harmonic mean of the destructive aliasing rate is reduced by 21%, from 6.1% before re-aliasing to 4.8% after re-aliasing. Constructive aliasing is also reduced slightly, from 0.41% to 0.31%. Total aliasing is reduced by 48%, from 18.3% to 9.5%.

On `181.mcf`, re-aliasing reduces destructive aliasing by 30%, from 16.6% down to 11.5%, explaining the 64% decrease in the misprediction rate, from 7.9% for GAg down to 2.8% for GAg with branch path re-aliasing.

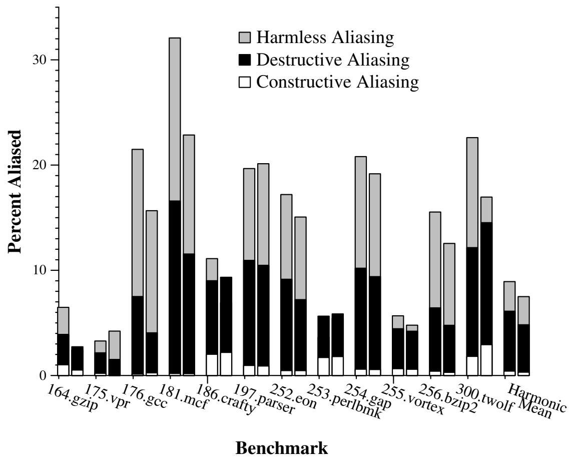


Figure 6.5: Branch aliasing rates on the SPEC 2000 integer benchmarks.

6.3 Limitations of Branch Path Re-Aliasing

In its current form, branch path re-aliasing works only for GAg predictors and predictors that use GAg as a component. It may have limited applicability to GAs predictors, but it cannot be used for per-branch predictors (e.g. PAg and PAs) or other kinds of global predictors like the perceptron predictor. It is also unable to improve the accuracy of certain predictors like *gshare* that use a dynamic technique to reduce destructive aliasing. Branch path re-aliasing is limited to GAg because it reduces aliasing by explicitly controlling how different paths map to PHT entries. Other predictors choose PHT entries (or perceptrons, as the case may be) based on some combination of branch address and global history. Branch path re-aliasing is frustrated in these cases, since the algorithm has no knowledge of branch addresses. In the case of the perceptron predictor, the majority of destructive aliasing that occurs is between unrelated static branches. Since the perceptron predictor uses only the branch address and not the global history to select perceptrons, there is no way that branch path re-aliasing could reduce this kind of destructive aliasing.

6.4 Summary

We have seen how, by moving complexity off the critical path to making a prediction and into the compiler, we can reduce the size of a GAs predictor and replace it with an enhanced GAg predictor with much the same accuracy. Branch path re-aliasing works by enlisting the help of the compiler, through profiling, to control aliasing explicitly. We have also seen that branch path re-aliasing can be applied to other predictors that use GAg as a component, such as *agree* predictors or hybrid predictors. Nevertheless, branch path re-aliasing only buys us some time. As we have seen with the example of the AMD K6 and Athlon, branch path re-aliasing will allow us to move to the next generation of processors and retain much the same accuracy. But we will still have the same fundamental problem for future generations: tables are slow. In the next chapter, we propose a more radical solution that will work in any CMOS technology generation.

Chapter 7

Cooperative Prediction with Boolean Formulas

Up until this point in the dissertation, we have not eliminated the fundamental source of the problem: the slow access time to tables. Rather, we have shifted the burden in various ways. In this chapter, we propose a longer-term solution to the problem: eliminate the tables altogether, and replace them with something much faster. Due to its unique implementation, the delay of this predictor remains low relative to the most aggressive clock rates and smallest future process technologies.

Existing architectures such as IA-64 allow hint bits in a branch instruction to specify whether to use the dynamic branch predictor or a static prediction, thus filtering the accesses to the dynamic predictor and reducing aliasing (i.e., contention for branch prediction resources). If the static predictions are chosen well, we can obtain better branch prediction accuracy, even with a smaller dynamic branch predictor.

We extend this idea to consider history-based predictors encoded in the branch instruction. In our scheme, a branch instruction encodes a Boolean function, learned through profiling, whose input is the branch history and whose output is a prediction [31]. The key to our solution is a concise encoding of Boolean functions—based on *monotone read-once Boolean formulas*—that is well-suited for branch prediction. Whereas an arbitrary Boolean function in N variables requires $O(2^N)$ bits to encode, monotone read-once Boolean formulas only require N bits. Figure 7.1 shows such a formula as a logic diagram. Because of our unique encoding and implementation, our Boolean formula predictor can deliver a branch prediction in a single cycle even at aggressive clock rates for which accurate PHT-based predictors are infeasible. The primary contribution of this chapter is a new branch prediction scheme that encodes into branch instructions a predictor in the form of a Boolean formula. Our method is particularly attractive in light of trends in technology scaling and wire delays. Secondary contributions include the following: (1) We describe the hardware implementation of our predictor and analyze it in terms of delay and power; (2) we describe a profiling algorithm for training our predictor; (3) we describe hybrid versions of our predictor that combine our technique with dynamic predictors; and (4) we evaluate the accuracy of our method using the SPEC 2000 integer benchmarks.

7.1 Branch Prediction with Boolean Formulas

In this section, we describe the main ideas behind predicting branches with Boolean formulas.

7.1.1 Boolean Formulas as Branch Predictors

History-based branch prediction can be viewed as the problem of learning the Boolean function of the branch history that gives the best prediction. Let $\mathbf{h}$ be a Boolean N-vector containing the outcomes of the last N branches executed. For now, we can think of this branch history as being either global or per-branch. For a static branch B, there exists a Boolean function $f_B(\mathbf{h})$ that best predicts whether B will be taken given the history $\mathbf{h}$. The goal of dynamic branch predictors is to learn this function as quickly as possible to provide accurate prediction.

One approach to branch prediction is to learn $f_B(\mathbf{h})$ for each branch in a profiling run, then somehow encode each $f_B(\mathbf{h})$ in the branch instruction and have the hardware use the dynamic history to compute the function and provide a branch prediction. Statically chosen bias bits, such as those available on HP-PA/RISC and IA-64, encode constant Boolean functions, which require no history information.

If the behavior of branches is stable across different program inputs, then we would expect branch prediction using these functions to perform very well, even better than dynamic branch predictors, which have the disadvantages of destructive aliasing. In practice, input-dependent behavior, such as loop trip counts that vary from run to run, limits the accuracy of a Boolean formula predictor. But as we will see, these functions still provide highly accurate predictions.

One problem with this approach is that of representing a Boolean function within a branch instruction. For instance, with a moderate history length of 10, there are $2^{2^{10}}$ different Boolean functions. Branch instructions would need to have over 1000 bits to allow all of these functions to be encoded. Therefore, we consider an extremely compact, but sufficiently expressive, encoding of Boolean formulas.

7.1.2 Read-Once Monotone Boolean Formulas

We now describe a subset of Boolean formulas that can be compactly represented. The basic idea is to restrict the Boolean formulas such that each variable appears in the formula only once, and the only operations allowed are AND and OR.

Let $\mathbf{x}, \mathbf{y} \in \{0,1\}^N$, i.e., $\mathbf{x}$ and $\mathbf{y}$ are N-bit vectors of Boolean values. We say that $\mathbf{x} \leq \mathbf{y}$ if, for all i, $\mathbf{x}_i \leq \mathbf{y}_i$. Consider a Boolean function $f\{0,1\}^N \mapsto \{0,1\}$, i.e., a function f mapping a vector of N bits to a single bit. We say that f is *monotone* if $\mathbf{x} \leq \mathbf{y}$ implies $f(\mathbf{x}) \leq f(\mathbf{y})$ [37]. A *monotone Boolean formula* is a Boolean formula that uses only AND ($\wedge$) and OR ($\vee$), without NOT, as connectives. The functions induced by these formulas are monotone [37], hence the name.

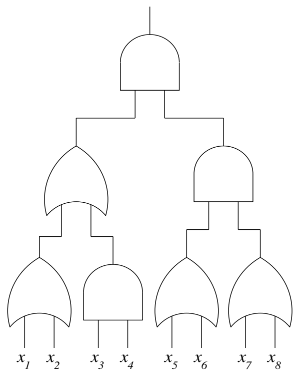


Figure 7.1: Tree representation of the formula $((x_1 \vee x_2) \vee (x_3 \wedge x_4)) \wedge ((x_5 \vee x_6) \wedge (x_7 \vee x_8))$.

In a *read-once formula* each variable appears exactly once in the formula. Read-once formulas are also known as μ-formulas or Boolean trees [3]. Read-once monotone Boolean formulas have a concise description as a tree whose internal nodes are ANDs and ORs and whose leaves are the Boolean variables. As an example, Figure 7.1 shows the tree representation of the formula $((x_1 \vee x_2) \vee (x_3 \wedge x_4)) \wedge ((x_5 \vee x_6) \wedge (x_7 \vee x_8))$ as a logic diagram.

7.1.3 Using Monotone Read-Once Formulas for Branch Prediction

A read-once monotone Boolean formula of N variables can be encoded as a bit vector of size $N-1$, each bit representing a connective in the Boolean tree, with 0 for AND and 1 for OR. Thus, each branch instruction encodes a read-once monotone Boolean formula using $N-1$ bits. We also store another bit that, if set to 1, causes the value of the function to be inverted, so that we can also represent the complements of monotone read-once formulas. No two different bit patterns represent the same Boolean function, so this encoding is quite efficient. For a history length of N, the formula encoding in the branch instruction takes N bits. Monotone Boolean formulas are incapable of representing Boolean constants, so we allow the formula whose connectives are all ANDs to compute 0 (i.e., *false*). By choosing to invert the output, this formula can also produce 1 (i.e., *true*). These two values are necessary, since they allow us to represent "always predict taken" and "always predict not taken," which are the most common Boolean functions for branch prediction.

For branch prediction, we keep a branch history shift register into which the Boolean outcomes (i.e., 1 for *taken* and 0 for *not taken*) of branches are shifted. We keep a global history, using the same shift register for all branches. When a branch instruction is fetched, the Boolean formula is sent, along with the contents of the history register, to a circuit that decodes the formula and computes the prediction.

We use a profiling phase to decide which formulas to encode in each branch instruction. The profiling algorithm uses statistics about the behavior of each static branch to choose the best monotone read-once formula for that branch.

The following formula is an example of a monotone read-once Boolean formula used for branch prediction with a history length of 8:

$$(x_0 \vee x_1) \wedge x_2 \wedge x_3 \wedge (x_4 \vee x_5 \vee x_6 \vee x_7),$$

This formula corresponds to a branch prediction policy of "predict taken if either of the last two branches were taken *and* the third and fourth most recent branches were both taken, *and* any of the other branches in the history were taken."

7.1.4 Profiling Algorithm

We now describe our algorithm for determining which formulas best predict each static branch. Using a trace of each branch address and outcome, the algorithm simulates the dynamic contents of the history register. For each static branch, the algorithm keeps a list of the different histories that lead up to that branch, along with the number of times each history leads to the branch being taken or not taken. After the algorithm has examined every dynamic branch, it checks the list for each static branch B and exhaustively tests every monotone Boolean formula and its complement to see which one would have yielded the fewest mispredictions given all the histories that led up to B. This best formula is then encoded into the branch instruction.

For branches that are executed fewer than 500 times in the profiled program, we simply use the constant formula (0 or 1) that best predicts that branch, rather than considering all 2^N formulas. We are investigating ways to speed up the algorithm with a more intelligent search. Section 7.2.5 gives timing results for the profiling algorithm and argues that the cost is reasonable for history lengths up to 16.

We have found that we can find formulas with a value for N of up to 18 in a reasonable amount of time, i.e., up to a few hours per benchmark. With $N \leq 10$, we can find the formulas in a few minutes, during which a large portion of the time is devoted simply to reading the profiled traces from disk.

7.1.5 Hardware Implementation

A hardware implementation of a Boolean formula branch predictor is simple. Each Boolean connective (i.e., AND or OR) in the formula is represented by a circuit with three inputs: two data inputs, corresponding to the variables or outputs of other gates, and one control input that specifies whether the Boolean connective should compute AND or OR. Coincidentally, this function is equivalent to the carry-out computed by a full adder. Figure 7.2 shows a logic diagram for this four-NAND circuit. With a history length of N, our predictor is built from $N - 1$ connectives and a single XOR gate at the output that acts as an inverter when its input is 1. Figure 7.3 shows a circuit implementation of the predictor for $N = 8$. For clarity, the extra logic to produce 0 when all the connectives are ANDs is not shown, since this logic requires relatively few gates and is not on the critical timing path.

We simulate a straightforward static CMOS implementation of the Boolean formula predictor with the HSPICE circuit simulator. First, we create a sub-circuit composed of four NAND gates as shown in Figure 7.2. Then, we instantiate $2 \log_2 N$ of these subcircuits and add an XOR, which is a sub-circuit consisting of two inverters and two NAND gates. The connections between the subcircuits are shown in Figure 7.3. Finally, we add capacitance between the gates to model local interconnect.

Note that although the concept of a read-once monotone Boolean formula is somewhat similar to the actual implementation as a circuit, to avoid confusion, the two should be thought of separately as function vs. implementation. In particular, the circuit is optimized for static CMOS technology with NAND gates and is not a read-once circuit.

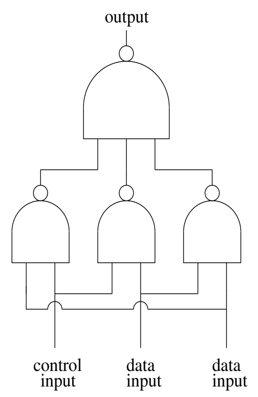


Figure 7.2: Boolean connective subcircuit.

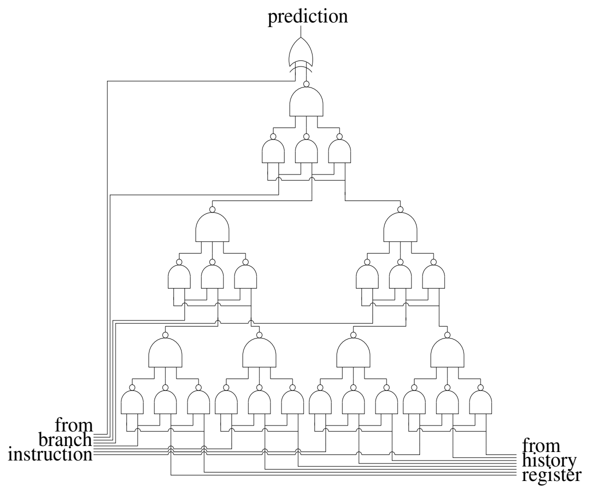


Figure 7.3: Boolean formula branch predictor circuit for a history length of 8.

Minimum Feature Size (nm)	Access Time (picoseconds)		
	4K-entry *gshare*	Formula, $N = 8$	Formula, $N = 16$
180	551	211	260
130	402	168	208
100	321	112	138
70	228	85	103
50	167	50	59

Table 7.1: Access times for a 4K-entry *gshare* predictor vs. two versions of the Boolean formula predictor.

7.1.6 Delay

The depth of the formula evaluation circuit with N inputs is $2 \log_2 N$ plus the final XOR gate. For instance, for $N = 16$, the critical delay path passes through eight NAND gates and one XOR gate. In contrast, the *gshare* predictor looks up values from a table by reading from an SRAM array. We use the methodology given in Chapter 3 for obtaining the access times for the pattern history tables.

We estimate the access time of the Boolean formula predictor by simulating the combinational circuit and measuring the delay from the branch instruction and history register inputs to the output of the XOR gate. The delay measurements are the time from the midpoint of the input signal switching to the midpoint of the output signal switching. We calculated the lookup time for a *gshare* predictor using our modified CACTI tool. Table 7.1 shows the access times for a 4K-entry *gshare* predictor and two sizes of the Boolean formula predictor, $N = 8$ and $N = 16$, for a range of fabrication technologies. We chose the 4K-entry predictor because, as we will see in Section 7.2, the $N = 8$ version of the Boolean formula predictor only slightly exceeds the accuracy of a 4K-entry *gshare*. Thus, our delay comparisons show that we can achieve higher accuracy with lower latency.

As fabrication technology improves, transistors can be made smaller and faster, resulting in higher clock frequencies and faster combinational circuits. As Table 7.1 shows, access times for each structure improve as the minimum feature size decreases.

The Boolean formula predictor is consistently faster than the 4K-entry *gshare* predictor, allowing more time for communication and computation within a clock cycle. At the projected clock rate of 7 Ghz for 50 nm technology, the clock period would be 144 picoseconds. A traditional table-lookup predictor such as *gshare* would require more than a single cycle—167 picoseconds in this case—for the prediction. In the same technology, the Boolean formula predictor would provide a prediction in 59 picoseconds, leaving over half of the cycle to prepare for and act upon the prediction.

One concern with our predictor is that the contents of the branch opcode are on the critical path to making a prediction; the Boolean formula must be read before it can be evaluated. However, this delay is common to any branch predictor that uses bias bits or any other type of information from the branch instruction, such as the agree predictor used on the HP-PA/RISC or the static/dynamic and bias bits provided by IA-64. One solution is to provide pre-decode bits in the instruction cache that provide the opcode information quickly.

7.1.7 Power

Power consumption has recently become a primary concern in microprocessor design. In this section, we contrast the power consumption of traditional branch predictors with that of the Boolean formula predictor.

The Boolean formula predictor is a combinational circuit that uses less dynamic power than an SRAM-based predictor. This small predictor has smaller gate and interconnect capacitance than an SRAM structure, which has decoding logic, a memory array, sensing logic, and output logic.

Table 7.2 shows the Boolean formula predictor's dynamic power consumption for $N = 8$ and $N = 16$, as measured with the HSPICE simulator. This table also shows the power of a 4K-entry *gshare* predictor, measure using the modified CACTI 2.0. The $N = 8$ results show that the Boolean formula predictor consumes between 0.4% to 2.9% of the power of a *gshare* predictor with comparable accuracy.

With lower transistor threshold voltages in emerging technologies, static power—due to leakage current through

Minimum Feature Size (nm)	Power (milliwatts)		
	4K-entry *gshare*	Formula, $N = 8$	Formula, $N = 16$
180	51.4	0.61	1.28
130	31.0	0.28	0.58
100	27.4	0.11	0.24
70	12.9	0.06	0.12
50	8.40	0.06	0.13

Table 7.2: Dynamic power consumption for two versions of the Boolean formula predictor and a 4K-entry *gshare*.

transistors—is becoming a sizable percentage of the total power consumed [61]. With fewer transistors in the circuit to leak current, the Boolean predictor circuit will also have less static power than an SRAM structure. Furthermore, the Boolean circuit implementation is amenable to a low static power design technique that takes advantage of the stacked transistors within gates to bias transistors into a low-leakage mode [61].

7.1.8 Impact of Encoding

Since each branch instruction encodes a Boolean formula, we must find an efficient way to encode the formula in the instruction without having a negative impact on performance. Some instructions sets already provide extra bits for communicating hints to the microarchitecture. For instance, the Alpha AXP ISA provides 14 bits in each indirect branch instruction for profiling information [55]. In their work on variable length path branch prediction, Stark *et al.* [58] use extra bits such as these to communicate to the microarchitecture information on hash functions for a branch predictor.

We propose changing the ISA so that branch instructions encode the formulas. For example, each branch instruction on the Alpha is 32 bits long: six bits indicate the op code of the instruction, five more bits indicate the register to test, and 21 bits are for the branch offset. For a Boolean-formula based branch predictor requiring N bits in a branch instruction, we propose to reallocate N of the offset bits to the formula. Some long branches will need to be split into a branch followed by a jump to the target, increasing the number of instructions executed.

Figure 7.4 shows the harmonic mean over the SPEC 2000 integer benchmarks of the percentage of extra instructions executed on the Alpha when offset bits are reallocated to Boolean formula predictors. We obtained these figures by assuming that every branch with an offset larger than would fit given the restricted number of offset bits would incur an additional jump instruction, and adding the number of such dynamically executed branches to the total number of instructions executed.

We measure the harmonic mean over the SPEC 2000 integer benchmarks of the percentage of extra instructions executed on the Alpha when offset bits are reallocated to Boolean formula predictors. With formulas of up to 9 bits, the number of extra instructions is negligible. With 12-bit formulas, only 0.2% more instructions are executed. With 14-bit formulas, 1.0% more instructions are executed. As history length increases beyond 16 bits, this encoding technique becomes less feasible. For longer histories, we have developed a more sophisticated technique that exploits the fact that most of the functions are constant. With this technique, only those branches for which the Boolean formula is more accurate than bias bits use Boolean formulas. The rest use simple bias bits, keeping the rest of the branch instruction opcode for storing the branch offset.

7.2 Results and Analysis

In this section, we give the results of simulating our branch predictor on the SPEC 2000 integer benchmarks, and we compare our results against both static (i.e., bias bits) and dynamic branch prediction. We also give results for a predictor that combines Boolean formulas with dynamic prediction, and we compare this to similar work that combines static and dynamic prediction.

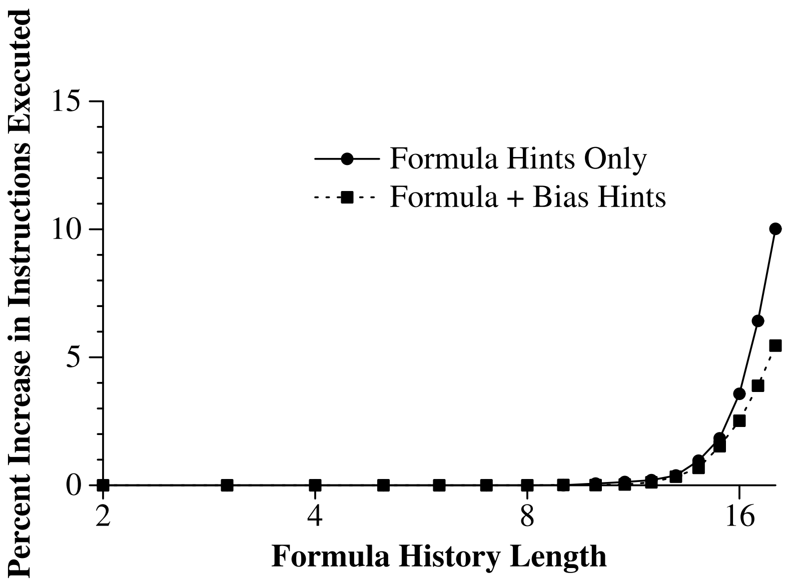


Figure 7.4: Impact of Formula Encoding on Performance.

7.2.1 Methodology

We use the 12 SPEC 2000 integer benchmarks running under SimpleScalar/Alpha [10] to collect traces. For each benchmark, we gather traces giving the branch address and outcome for up to 300 million branches. We use the `train` inputs for the profiling runs, and we use the `ref` inputs to evaluate the accuracy of the various predictors. To better capture the steady-state performance of the branch predictors, our evaluation runs skip the first 50 million branches, as several of the benchmarks have an initialization period (lasting fewer than 50 million branches), during which branch prediction accuracy is unusually high. Each benchmark executes at least 300 million branches and over one billion instructions on the `test` inputs before the simulation ends.

7.2.2 Predictors Simulated

We simulate monotone read-once Boolean formula predictors for $2 \leq N \leq 18$. We use only global history information, i.e., we do not use path or per-branch information. We also simulate the *gshare* [41], *bi-mode* [38] and agree [57] branch predictors, three well-known global dynamic branch predictors from the literature. The *gshare* and *bi-mode* predictors use only dynamic history information. The agree predictor combines static and dynamic information by predicting whether a branch will agree with a bias bit.

History length has been observed to have a significant impact on predictor accuracy [41], so for each predictor and each hardware budget, we try all possible history lengths on the `train` inputs and keep the one with the lowest average misprediction accuracy.

To give a lower-bound on misprediction rates for any Boolean-formula based predictor, we also measure the results of using *arbitrary* Boolean formulas. That is, we measure the results of using the best possible static Boolean function for a given branch, over a training set, regardless of the cost of implementing the function. To find the best arbitrary Boolean formula for a particular static branch, we measure the number of taken versus not-taken branches for each history leading up to that branch in the training set, then assign to each history the prediction yielding the most correctly predicted dynamic branches. Out of all the possible histories leading to a branch, only a small fraction will actually be observed; all other histories are assigned the bias bit for that branch. The arbitrary predictor is represented by the profiling algorithm as a set of rows in a truth table where the inputs are the histories and the output is the prediction. Note that this arbitrary formula predictor is actually implementable for history lengths of up to four, since the truth table for a Boolean function in four variables can be encoded in only 16 bits.

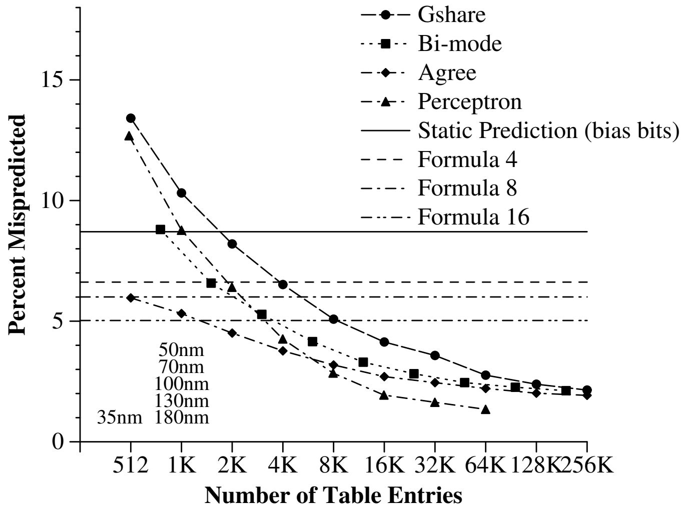


Figure 7.5: Accuracy of dynamic branch predictors vs. static prediction and the Boolean formula predictor.

7.2.3 Misprediction Rates

Figure 7.5 shows misprediction rates for the monotone read-once Boolean formula predictor at history lengths of 4, 8 and 16, compared with *gshare*, agree and *bi-mode* predictors at hardware budgets from 512 to 256K entries. Labels above the 512 and 1K-entry hardware budgets show the process technologies for which the corresponding budget is reachable in one cycle at an aggressive clock rate. Also shown is the misprediction rate for the global perceptron predictor from Chapter 5 in the same range of hardware budgets, to provide a comparison of the accuracies of the perceptron and formula predictors. Note that the perceptron predictor cannot work in a single cycle at any hardware budget without delay-hiding techniques, so the labels above the x-axis apply only to the PHT-based predictors.

At today's 180 nm and 130 nm technologies, for which branch predictors with only about 1K to 2K table entries state are available at more aggressive clock speeds, a 4-bit Boolean formula predictor with a misprediction rate of 6.6% roughly matches the accuracy of the *bi-mode* predictor. With a history length of 16, the Boolean formula predictor has a misprediction rate of 5.02%, an improvement of 24% over the 1.5K-entry *bi-mode* predictor, and roughly matching the misprediction rate of the perceptron predictor at the equivalent of a 1K-entry budget.

To put these figures another way, a 4-bit Boolean formula predictor achieves roughly the same predictive power as a 4K-entry *gshare* predictor. A 16-bit Boolean formula predictor is about as accurate as an 8K-entry gshare predictor, a 3K-entry *bi-mode* predictor, or a 2K-entry agree predictor.

Figure 7.6 shows, for history lengths ranging from 2 to 18, misprediction rates for the monotone read-once Boolean formula predictor, as well as for the predictor that uses arbitrary formulas. For reference, it also shows the misprediction rates with history lengths from 0 to 18 for pure static prediction with bias bits, as well as for dynamic prediction with a 1K entry *gshare*, a 1K entry agree predictor, and a 1.5K entry *bi-mode* predictor; these table sizes represent the predictors accessible in a single cycle in 50 through 130 nm technology with aggressive clock rates. As history length increases, the misprediction rate of the Boolean formula predictor decreases and remains close to the performance of the arbitrary formula predictor. As a lower bound on dynamic branch predictor misprediction rate, Figure 7.6 also shows the misprediction rate of a perceptron predictor with arbitrarily many perceptrons and an increasing history length, i.e., a predictor with no conflict aliasing. Clearly, this ideal perceptron predictor is more accurate than even arbitrary Boolean formulas, so the idea of encoding any static function in the branch instruction has its own limitations.

Figure 7.7 shows misprediction rates on each benchmark with the same limited-budget two-level predictors as well as Boolean formula predictors with history lengths of 8 and 16. The Boolean formula predictor usually has a misprediction rate lower than that of the dynamic predictors. However, in a few cases, such as `256.bzip2`, the formula predictor's misprediction rate is high, most likely due to input-dependent program behavior that cannot be learned by profiling.

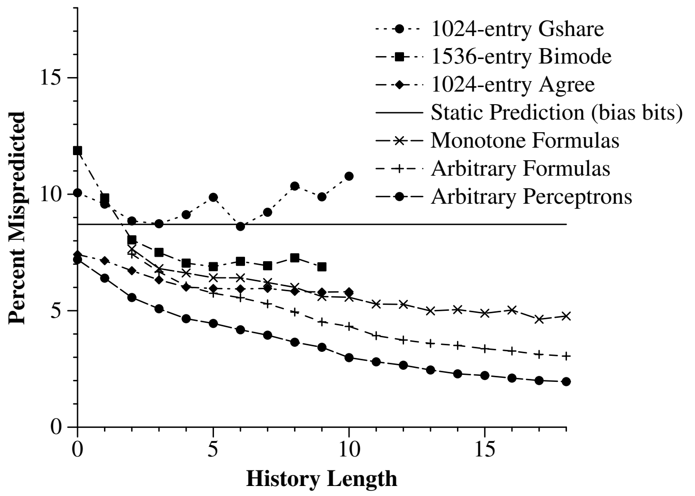


Figure 7.6: Misprediction rate for the Boolean formula predictor as a function of history length.

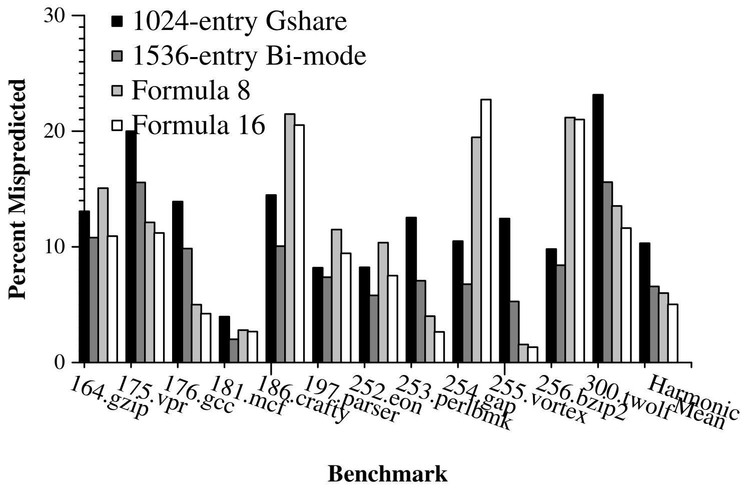


Figure 7.7: Accuracy of the predictors on each benchmark.

Figure 7.8 shows the misprediction rates of predictors using the agree mechanism combined with our formula predictor. An agree predictor predicts whether a branch outcome will agree with a bias bit, turning destructive aliasing into constructive aliasing. Our combined agree/formula predictors use a PHT to predict whether the branch outcome will agree with the output of a Boolean formula, rather than a bias bit. With a 1K-entry PHT, the agree predictor with bias bits yields a misprediction rate of 5.3%. The 8-bit version of our agree/formula predictor decreases this rate to 4.4%, an improvement of 17%. The 16-bit version of our predictor has a misprediction rate of 3.9%, an improvement of 25%.

For reference, we compare our predictor with the Alpha 21264 hybrid branch predictor, which is the most accurate existing predictor for which implementation details are readily available [35]. This predictor uses a 4K-entry global history predictor and a 1K-entry per-branch history predictor combined with a 4K-entry chooser, consuming roughly 4KB of state. The Alpha 21264 predictor achieves a misprediction rate of 2.93% on the traces we gathered. At the same hardware budget, the agree predictor, when enhanced with the 16-bit version of our Boolean formula predictor, achieves a misprediction rate of 2.55%. Even at half the hardware budget of the Alpha 21264 predictor, an 8K-entry version of our agree/formula hybrid achieves a misprediction rate of 2.86%, narrowly better than the Alpha hybrid. Using our aggressive clock modeling, the largest hybrid agree/formula predictor available in a single cycle will achieve a misprediction rate of 3.97%, which is 35% higher than that of the Alpha predictor. However, an important point of our research is that complex predictors such as the Alpha's are infeasible at higher clock rates. Even today's Alpha must employ an overriding mechanism [35], in which branch predictions that do not agree with the less sophisticated cache line predictor introduce a single-cycle bubble into the pipeline, reducing the performance advantage of the more accurate hybrid predictor.

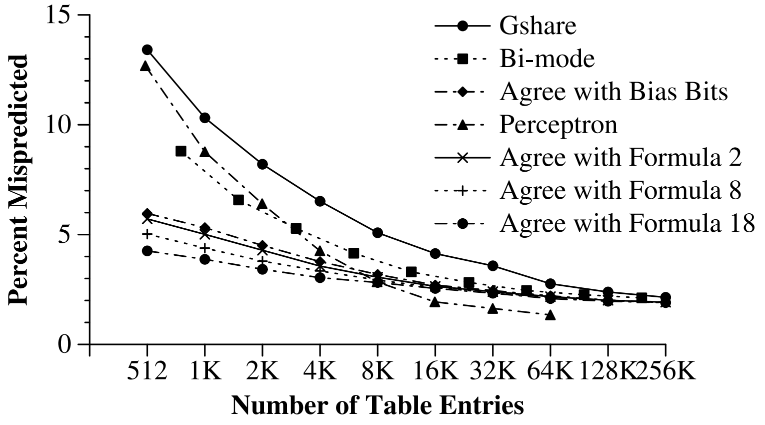


Figure 7.8: Accuracies of Boolean formula predictors using the agree mechanism. Misprediction rates are harmonic means over SPEC 2000.

7.2.4 Distribution of Formulas

An analysis of the distribution of Boolean formulas chosen by the profiling algorithm shows that most of the Boolean formulas chosen are the two constant functions, 0 and 1. This dependence on constant formulas decreases as history length increases. For instance, with a history length of 4, 78% of static branches in the SPEC 2000 integer benchmarks are best predicted with a constant formula, as opposed to only 49% for a history length of 16. As history length increases, the predictive power of the Boolean formula predictor increases, and the constant functions representing "predict taken always" and "predict not taken always" give way to more intelligent choices. Figure 7.9 shows, for history lengths from 2 to 18, the percentage of dynamic and static branches for which constant formulas are chosen.

Table 7.3 shows the dynamic frequencies for each formula with a history length of four, along with the misprediction rate for each formula using a 4-bit Boolean formula predictor and for bias bits. For brevity, we omit similar tables for the other history lengths.

7.2.5 Profiling Cost

The cost of determining the best Boolean formula for each branch is an important component of the cost of our branch predictor. Here, we quantify this cost.

The majority of the time of the profiling algorithm is spent evaluating Boolean formulas for the set of histories leading up to a branch. The time to evaluate a formula for a given input (i.e. history) is roughly constant. With a history length of up to 12, we have found that generating a lookup table to hold the function values is most efficient; for these history lengths, the cost of evaluating a formula is the cost of a single memory access. Beyond a history length of 12, we use a C++ function call to evaluate formulas. With efficient coding and formula representation, using a history length of 16, a single formula evaluation takes an average of 270 ns on a 733MHz Pentium III.

Figure 7.10 shows the amount of time taken for profiling, as a function of history length. The graph shows the arithmetic mean of the time, in seconds, that our algorithm spent for each benchmark. These times were collected

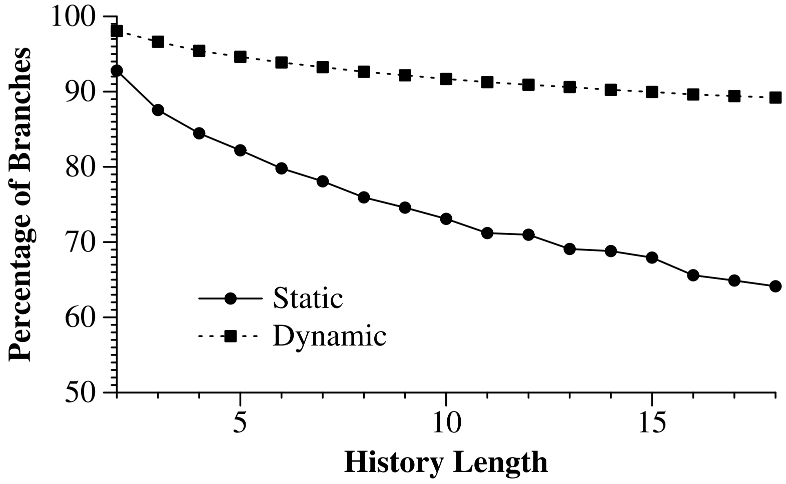


Figure 7.9: Percentage of branches using constant formulas at various history lengths.

Formula	% Dyn. Freq.	% Mispredicted Formula	% Mispredicted Bias
1	40.84	9.4	9.4
0	37.14	10.0	10.0
$(x_0 \vee x_1) \wedge (x_2 \vee x_3)$	3.15	21.8	36.3
$\neg((x_0 \vee x_1) \wedge (x_2 \vee x_3))$	2.36	24.6	36.6
$(x_0 \vee x_1) \vee (x_2 \wedge x_3)$	2.06	21.5	29.3
$\neg((x_0 \vee x_1) \vee (x_2 \wedge x_3))$	1.73	14.4	24.5
$\neg((x_0 \wedge x_1) \wedge (x_2 \vee x_3))$	1.64	20.1	26.8
$(x_0 \wedge x_1) \wedge (x_2 \vee x_3)$	1.60	15.8	22.0
$\neg((x_0 \wedge x_1) \vee (x_2 \vee x_3))$	1.54	16.3	23.7
$(x_0 \wedge x_1) \vee (x_2 \vee x_3)$	1.49	14.1	18.6
$(x_0 \vee x_1) \wedge (x_2 \wedge x_3)$	1.30	26.4	34.9
$(x_0 \wedge x_1) \vee (x_2 \wedge x_3)$	1.23	20.3	38.7
$\neg((x_0 \wedge x_1) \vee (x_2 \wedge x_3))$	1.16	35.6	42.1
$\neg((x_0 \vee x_1) \wedge (x_2 \wedge x_3))$	1.09	26.2	36.1
$\neg((x_0 \wedge x_1) \wedge (x_2 \wedge x_3))$	0.99	21.6	18.5
$(x_0 \wedge x_1) \wedge (x_2 \wedge x_3)$	0.66	5.3	10.3

Table 7.3: Distribution of Boolean formulas for a history length of 4.

by running our program on our network of 733MHz Pentium III computers.

Our current implementation takes time exponential in the history length. However, for the small history lengths that we consider in this study, the time is not unreasonable. For instance, with a history length of 16, the profiling algorithm takes about 12 minutes on a 733MHz Pentium III. For a history length of 10, the program takes about 2 minutes. For history lengths less than about 12, the time for the program is dominated by activities unrelated to finding the best Boolean function. For instance, much time is spent simply reading the large trace file from the disk and performing other tasks that any typical feedback-directed optimization would require. Our algorithm is also easy to parallelize. The time-consuming part of the algorithm—during which the best Boolean formula is decided for each static branch—is embarrassingly parallel, as the various static branches can be partitioned among many processors. Thus, we feel that our profiling algorithm would be appropriate in a framework in which other optimizations are also being explored by simulation.

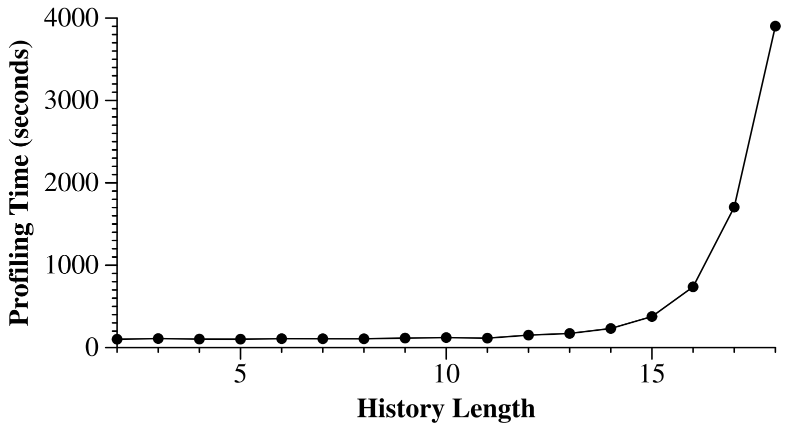


Figure 7.10: Average, over all benchmarks, of the amount of time spent profiling as a function of history length.

7.3 Summary

We have introduced and evaluated a new branch prediction scheme that borrows from complexity theory the concept of a read-once monotone Boolean formula. These Boolean formulas provide a compact encoding of a class of functions that is expressive enough to perform branch prediction yet concise enough to be encoded in branch instructions. By off-loading most of the prediction work to the compiler, our Boolean formula predictor is small, fast and consumes little power. While our scheme provides a competitive alternative to existing dynamic branch predictors, the real benefit of our scheme lies in the future, as our scheme is significantly less sensitive to the impending technology scaling issues caused by increased wire delays. Our predictor can also form a valuable component of an agree or hybrid predictor, decreasing misprediction rates by providing better estimates of branch outcomes than bias bits.

Chapter 8

Related Work

Our work builds on the contributions of many other researchers and engineers. To place our work in the context of other research, we now review some of the recent related work.

8.1 Hint Bits in Branch Predictors

Our Boolean formula predictor and branch path re-aliasing scheme are two of many prediction ideas that provide hints through the ISA to the branch instruction. One highly successful technique is *branch classification* [14], in which a branch instruction specifies which predictor is best for that branch. Many branches are predicted well with a static prediction; these branches can be "filtered" out of the stream of branches that are allowed to update the PHT, thus reducing aliasing. A version of the *agree* predictor predicts whether a branch outcome will agree with a bias bit set in the branch instruction [57]. August *et al.* propose placing hint bits in each branch instruction that tell a dynamic predictor what kind of state to examine to make a prediction [5]. The variable length path branch predictor [58] encodes profiling information in branch instructions; this information guides a dynamic predictor, telling it what history length to use and what hash function of past branch addresses to use to form an index into a table of counters.

Unfortunately, for most of these techniques to work, the branch instruction has to have been at least partially decoded before the branch prediction can be made. These techniques will not be feasible in aggressively clocked CPUs with multi-cycle instruction cache latencies, since the predictor is in series with the instruction cache. Our Boolean formula predictor suffers from the same problem. However, our branch path re-aliasing predictor is different; it uses a hint bit in the branch instruction, but the hint is not needed until the branch predictor is updated.

8.2 Combining Static and Dynamic Branch Prediction

Branch prediction accuracy can be increased by combining static with dynamic branch prediction. Some of the branches can be predicted with a static bias bit, while others with less biased behavior can use the dynamic predictor. Since the easily predictable branches are filtered out, aliasing in the dynamic predictor is alleviated and accuracy is improved. This technique, along with a methodology for choosing the bias bits, was introduced by Chang *et al.* as branch classification. Patil and Emer study the technique, measuring its utility in reducing destructive aliasing and refining the heuristics used to decide which branches should be predicted statically [46].

8.3 Branch Prediction and Machine Learning

Our perceptron predictor borrows from neural learning techniques, which are a subset of the field of machine learning. Our Boolean formula predictor is also similar in some respects to other branch predictors based on decision trees. In this section, we review other work related to branch prediction and machine learning.

8.3.1 Neural Networks

Neural networks have been used for doing branch prediction before, but in a quite different context. Neural networks have been used to perform static branch prediction [11]. The likely branch direction of a static branch is predicted at compile-time by supplying program features, such as control-flow and opcode information, as input to a trained neural network. This approach achieves an 80% correct prediction rate, compared to 75% for static heuristics [6, 11]. Our work proposes putting a neural structure inside the microprocessor itself, so that it can learn on-line from the branch history.

8.3.2 Genetic Algorithms

Machine learning also includes genetic algorithms. Emer and Gloy use genetic algorithms to "evolve" branch predictors [19], but it is important to note the difference between their work and ours. Their work uses evolution to design more accurate predictors, but the end result is something similar to a highly tuned traditional predictor. We perform extra work in the microarchitecture, so the branch predictor can learn and adapt on-line.

8.3.3 Decision Trees

Decision trees are predictors learned through training, much like neural networks. The work of Calder *et al.* in static branch prediction with neural networks also explores the use of decision trees [11]. Lindsay explores the use of decision trees to encode statically-learned Boolean functions [40]. The decision trees are learned by profiling and are encoded in programmable logic arrays (PLAs). By contrast, our Boolean formula predictor encoding is represented only in the branch instruction, requiring little hardware in the CPU itself. Although Lindsay's thesis addresses latency issues, PLAs representing the behavior of large sets of branch instructions will have the same technology scaling issues in future technologies as large banks of SRAM. Similarly, Fern *et al.* [24] study the use of decision trees, grown dynamically, for branch prediction. The trees are kept in a large structure in the CPU and would have the same problems with delay as other predictors. Thus, our technique is distinctly well-suited to the issues of technology scaling.

8.4 Latency-Sensitive Branch Prediction

Although the problem of delay in branch predictors has not been studied in detail, there are quite a few studies in which related issues have appeared.

Lookahead branch prediction, including predicting multiple branches per cycle, has been suggested as a means for predicting branches that have not yet been presented to the predictor. One of the first lookahead branch predictors was proposed by Yeh *et al.* [64] as the Multiple Branch Two-Level Adaptive Branch Predictor. This predictor uses the result of the first branch prediction to speculatively update the history register for a second branch prediction. No branch addresses are required since only the global history register is used to access the pattern history tables. Seznec *et al.* improve on this idea by enhancing the branch target buffer (BTB) to enable the predictor to use the address of the current instruction block to perform prediction for the next instruction block [54]. This scheme enables the fetches to multiple blocks to be pipelined. Onder, Xu and Gupta propose a similar scheme in which predictions for an entire branch sequence are made all at once, and instruction fetch can continue unimpeded through the last branch [44].

Driesen and Hölze propose a "cascaded" predictor that dynamically filters easily predicted branches, relieving aliasing effects in the pattern history table (PHT) [17]. Our work borrows the idea of cascading, but uses it to alleviate delay. Similarly, Evers describes the use of two PHTs with different history lengths and different access times, where the slower one can override the other [22]. The Alpha 21264 branch predictor uses the idea of overriding: the branch predictor can override the less accurate instruction cache line predictor, with a penalty of a single cycle, as opposed to the seven-cycle branch misprediction penalty [35].

Some of our work builds on the fact that there is often more than one cycle between branches, even on a wide-issue processor, and that useful branch prediction work can be done in between branches. This fact was also noticed in work by Onder *et al.* in their paper on predicting branch
sequences [44].

Of course, the real goal in these strategies is to improve instruction fetch bandwidth and preferably take branch prediction off the critical path. Recent research has focused on trace caches as a mechanism to capture a long stream of sequential instructions that can be easily fetched at peak bandwidth [51, 45]. Branch prediction guides the trace selection in the instruction fetch engine, at times predicting multiple branches per cycle. A more radical approach is the Fetch Target Buffer (FTB) proposed by Reinman, et al. [48]. The FTB stores the addresses of predicted blocks of instructions and is designed as a two-level cache for fast access and accurate block prediction. Like our study, Reinman *et al.* consider technology constraints in the design of the FTB. Frameworks like the FTB can benefit by using our delay-sensitive branch prediction stategies as their branch prediction components.

Chapter 9

Conclusions

In this chapter, we review the contributions of this dissertation and discuss the relationships between the various proposed techniques.

9.1 Contributions

Recall the thesis statement from the introduction:

> Despite the effects of aggressive clock scaling, wire delay, and complex organizations, future branch direction predictors can have improved accuracy while still providing a prediction in a single cycle.

Until now, branch prediction design has focused on accuracy while ignoring delay. We have shown that as wire delays and clock rates increase, branch predictor designs that optimize for accuracy can have a negative impact on overall IPC (see Figures 4.7 and 4.6). Thus, future branch predictor efficacy depends on both *accuracy* and *delay*, and researchers should account for both when reporting branch prediction results. According to our scalable models for branch predictor access time, today's predictors will not be accessible in a single cycle in sub-100nm technologies with aggressive clocking. In deep sub-micron technologies that are latency rather than capacity-dominated, a branch predictor's area will become less important than its latency in the critical path.

In this section, we show how the various techniques described provide compelling evidence for our thesis.

9.1.1 Hierarchical Organizations

In this dissertation we have examined a number of alternative branch predictor architectures and evaluated them in the context of future process technologies. These hierarchical organizations are capable of extending traditional predictors such as *gshare* into future technologies, as well as enabling more complex predictors such as the perceptron predictor. A cascading lookahead predictor that uses the time in between branches to make predictions performs well. An overriding predictor that allows a slow predictor to cancel the prediction of a faster, but less accurate predictor performs the best.

We have introduced a new branch predictor that uses neural networks—the perceptron in particular—as the basic prediction mechanism. Perceptrons are attractive because they can use long history lengths without requiring exponential resources. A potential weakness of perceptrons is their increased computational complexity when compared with two-bit counters, but we have shown how a perceptron predictor can be implemented efficiently with respect to both area and delay using hierarchical organizations. The perceptron predictor performs well at all hardware budgets, achieving a lower misprediction rate than several well-known global predictors on the SPEC 2000 integer benchmarks; indeed, since the perceptron predictor outperforms Evers' multi-component predictor, we claim that the perceptron predictor is the most accurate fully dynamic branch predictor known. Despite its higher latency, the perceptron predictor yields higher IPCs than the other predictors because of our use of hierarchical organizations.

In our simulations, we have taken into account both clock rate and wire delay in future technologies, and shown that hierarchical organizations, both for traditional predictors and for the perceptron predictor, allow branch prediction in a single cycle while making use of slower structures to improve prediction accuracy. These techniques and experiments provide strong evidence for our thesis.

9.1.2 Cooperative Predictors

Cooperative branch predictors break the tradeoff between delay and accuracy by off-loading a significant amount of the prediction work to compile-time. We have evaluated two cooperative branch prediction schemes.

Branch path re-aliasing is a new branch prediction technique that improves accuracy for GAg predictors. By using path profiles to map paths leading to different outcomes to different PHT locations and paths with similar outcomes to the same PHT locations, re-aliasing decreases destructive aliasing. An advantage of branch path re-aliasing is its simplicity and low delay. The time to access the predictor is limited only by the time to access the PHT; there is no other logic on the critical path. Moreover, variations of our technique improves accuracy in complex predictor organizations such as *agree* and hybrid predictors.

The Boolean formula predictor eliminates tables and their associated delays altogether, using a compact encoding of Boolean functions to perform branch prediction. This predictor is feasible even with the most aggressive future clock rates and smallest process technologies.

A disadvantage of both the Boolean formula predictor and branch path re-aliasing is that they require profiling, whereas the perceptron and hierarchical predictors are not burdened by this constraint.

Our cooperative branch prediction approach fits in with the general trend towards moving more work out of the processor and into the compiler. By making prediction simpler without reducing accuracy, we can enjoy the benefits both of high IPC and high clock rates enabled by these single-cycle predictors. This work addresses part of the thesis statement relating to improved accuracy at higher clock rates, and provides an alternative to the hierarchical techniques.

9.2 Comparison of the Techniques

We have proposed several different techniques that address the problem of delay in branch predictors. In this section, we compare the techniques with one another, providing information useful to a microarchitect trying to decide which one to use.

To illustrate the comparison, Figure 9.1 shows the misprediction rates of many of the predictors introduced in this dissertation, as well as the misprediction rates of *gshare*, *agree* and hybrid predictors.

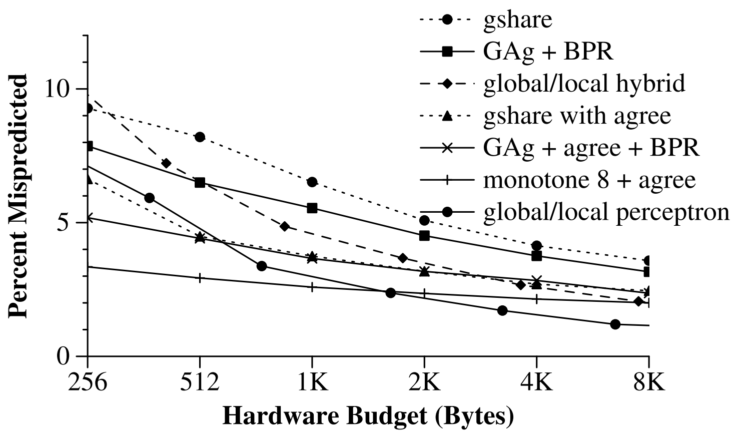


Figure 9.1: Misprediction Rates of Our Predictors

9.2.1 Advantages of Hierarchical Organizations

The main advantage of hierarchical organizations is that compiler and ISA support is not needed; predictors with hierarchical organizations are strictly a microarchitectural technique, with no directly observable consequences on the ISA or compiler. Conversely, cooperative predictors require changes to the ISA (although branch path re-aliasing may re-use existing ISA bits), and profiling. Figure 9.1 shows that, at hardware budgets over one kilobyte, the global/local perceptron predictor has the lowest misprediction rate of any of the predictors examined in this dissertation. We recommend this predictor for situations where unmodified legacy programs must be run as quickly as possible, with design complexity being only a secondary concern. Unfortunately, the perceptron predictor introduces complexity into the design of the processor, particularly in the layout of the irregularly-shaped Wallace-tree circuit. A less ambitious design could still use hierarchical organizations with a more traditional *gshare* or hybrid predictor, taking advantage of the accuracy of a large structure as well as the speed of a small structure.

9.2.2 Advantages of Cooperative Predictors

Recent trends in computer architecture point to a greater willingness to modify the ISA and rely on profiling or even dynamic compilation to achieve performance improvements [28, 4]. Our cooperative predictors are in line with this trend. Both of our cooperative predictors greatly simplify the hardware aspects of the branch predictor at the expense of profiling. The Boolean formula predictor uses about 1% of the area and power, and 33% of the delay of a *gshare* predictor with the same predictive power. Branch path re-aliasing allows us to use a smaller, faster table while maintaining the same accuracy previously achieves with a larger table.

Figure 9.1 shows that the best predictor at small hardware budgets is a Boolean formula predictor combined with a small *agree* predictor. This organization enables quick prediction and a low misprediction rate, with low implementation costs. At a cost of 1024 two-bit counters, this predictor achieves a misprediction rate of 2.59%, which is slightly better than a much larger *gshare* with 16,384 counters at 2.76%.

9.2.3 Recommendations

We make the following recommendations for delay-sensitive branch predictors, according to the particular goals of a microarchitect and the progress in process technology:

Hierarchical Organizations. We recommend the hierarchical organizations as a cheap way for a microarchitect to continue using traditional branch predictors with large budgets in a situation where high clock rates prevent single-cycle access. Hierarchical organizations are sensible in the near future, when binary compatibility with previous generations of microprocessors prevents invasive changes to the ISA.

Hierarchical Perceptron Predictor. We recommend a global/local hierarchical perceptron predictor when the microarchitect is unconstrained by chip area and power, and is simply concerned with the most accurate branch predictor possible. The perceptron predictor will make sense in the next several years, when increasingly deep pipelines for high-performance microprocessors will cause branch prediction accuracy to become a bottleneck for performance.

Branch Path Re-Aliasing. We recommend branch path re-aliasing in two cases. First, for processors with small branch prediction tables and branch bias bits in the ISA, the bias bits can be reused to implement branch path re-aliasing for a performance boost. Second, in future technologies when instruction cache latencies cause techniques such as bias bits and the Boolean formula predictor to become infeasible because they are on the critical path to making a prediction, branch path re-aliasing will still be feasible and allow fast and accurate predictions.

Boolean Formula Predictor. We recommend the Boolean formula predictor when performance as well as power and chip area are a concern, but ISA changes are not a problem. For instance, this predictor can be used in embedded or DSP systems, such as hand-held devices, that require high performance with minimum resources. For general purpose computing, the Boolean formula predictor will make more sense in about the next 10 years, when chip multiprocessors may become the dominant computing substrate. Since the area occupied by the branch predictor can be essentially eliminated by the Boolean formula predictor, the area savings can be devoted to other resources such as caches or more cores.

9.3 Final Thoughts

Branch predictor delay is a important barrier that future microarchitectures must overcome to achieve higher performance. Microarchitects cannot simply ignore the problem by settling for smaller, less accurate predictors or naively implementing multi-cycle predictors. We have shown that branch predictor delay can be overcome with a variety of techniques to enable processors of the future to maintain and even improve IPCs in the face of technology constraints. This work provides a point from which microarchitects of future processors can begin designing faster and more accurate branch predictors. Although there are other technological issues that will complicate future microarchitecture designs, we are confident that, if future microarchitects will use the ideas presented in this dissertation, the branch predictor will not be a performance bottleneck. Rather, branch prediction research can continue to provide improved accuracy and greater instruction fetch bandwidth needed for the challenges facing high-performance computing.

Bibliography

[1] V. Agarwal, M.S. Hrishikesh, S. W. Keckler, and D. Burger. Clock rate versus ipc: The end of the road for conventional microarchitectures. In *the 27th Annual International Symposium on Computer Architecture*, pages 248–259, May 2000.

[2] Vikas Agarwal, Stephen W. Keckler, and Doug Burger. Scaling of microarchitectural structures in future process technologies. Technical Report TR2000-02, Department of Computer Sciences, The University of Texas at Austin, Austin, TX, February 2000.

[3] Dana Angluin, Lisa Hellerstein, and Marek Karpinski. Learning read-once formulas with queries. *Journal of the Association for Computing Machinery*, 40(1):185–210, January 1993.

[4] Matthew Arnold, Stephen Fink, David Grove, Michael Hind, and Peter F. Sweeney. Adaptive optimization in the Jalapeño JVM. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'00)*, October 2000.

[5] David I. August, Daniel A. Connors, John C. Gyllenhaal, and Wen mei W. Hwu. Architectural support for compiler-synthesized dynamic branch prediction strategies: Rationale and initial results. In *Proceedings of the Third International Symposium on High-Performance Computer Architecture*, February 1997.

[6] T. Ball and J. Larus. Branch prediction for free. In *Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 300–313, June 1993.

[7] T. Ball and J. Larus. Efficient path profiling. In *Proceedings of the 29th International Symposium on Microarchitecture*, December 1996.

[8] Jim Basney, Miron Livny, and Todd Tannenbaum. High throughput computing with condor. *HPCU News*, 1(2), June 1997.

[9] H. D. Block. The perceptron: A model for brain functioning. *Reviews of Modern Physics*, 34:123–135, 1962.

[10] Doug Burger and Todd M. Austin. The simplescalar tool set version 2.0. Technical Report 1342, Computer Sciences Department, University of Wisconsin, June 1997.

[11] B. Calder, D. Grunwald, M. Jones, D. Lindsay, J. Martin, M. Mozer, and B. Zorn. Evidence-based static branch prediction using machine learning. *ACM Transactions on Programming Languages and Systems*, 19(1), 1997.

[12] Brad Calder and Dirk Grunwald. Fast and accurate instruction fetch and branch prediction. In *Proceedings of the 21st International Symposium on Computer Architecture*, pages 2–11, April 1994.

[13] Brad Calder and Dirk Grunwald. Reducing branch costs via branch alignment. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1994.

[14] P.-Y. Chang and U. Banerjee. Branch classification: a new mechanism for improving branch predictor performance. In *Proceedings of the 27th International Symposium on Microarchitecture*, November 1994.

[15] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. McGraw Hill, 1990.

[16] Keith Diefendorff. K7 challenges Intel. *Microprocessor Report*, 12(14), October 1998.

[17] Karel Driesen and Urs Hölze. The cascaded predictor: Economical and adaptive branch target prediction. In *Proceedings of the 31th International Symposium on Microarchitecture*, December 1998.

[18] A.N. Eden and T.N. Mudge. The YAGS branch prediction scheme. In *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture*, November 1998.

[19] J. Emer and N. Gloy. A language for describing predictors and its application to automatic synthesis. In *Proceedings of the 24th International Symposium on Computer Architecture*, June 1997.

[20] 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 International Symposium on Computer Architecture*, May 1996.

[21] M. Evers, S. J. Patel, R. S. Chappell, and Y. N. Patt. An analysis of correlation and predictability: What makes two-level branch predictors work. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, July 1998.

[22] Marius Evers. *Improving Branch Prediction by Understanding Branch Behavior*. PhD thesis, University of Michigan, Department of Computer Science and Engineering, 2000.

[23] L. Faucett. *Fundamentals of Neural Networks: Architectures, Algorithms and Applications*. Prentice-Hall, Englewood Cliffs, NJ, 1994.

[24] A. Fern, R. Givan, B. Falsafi, and T.N. Vijaykumar. Dynamic feature selection for hardware prediction. Technical Report TR-ECE 00-12, School of Electrical and Computer Engineering, Purdue University, 2000.

[25] Linley Gwennap. Ia-64: A parallel instruction set. *Microprocessor Report*, 13(7):6–11, May 1998.

[26] Glenn Hinton, Dave Sager, Mike Upton, Darrell Boggs, Doug Carmean, Alan Kyker, and Patrice Roussel. The microarchitecture of the Pentium 4 processor. *Intel Technology Journal Q1*, 2001.

[27] Mark Horowitz, Ron Ho, and Ken Mai. The future of wires. In *Semiconductor Research Corporation Workshop on Interconnects for Systems on a Chip*, May 1999.

[28] Intel Corporation. Intel Pentium 4 processor optimization. Technical Report Order Number: 248966, Intel Corporation, 2001.

[29] Erik Jacobsen, Eric Rotenberg, and James E. Smith. Assigning confidence to conditional branch predictions. In *Proceedings of the 29th Annual International Symposium on Microarchitecture*, December 1996.

[30] D. A. Jiménez and N. Walsh. Dynamically weighted ensemble neural networks for classification. In *Proceedings of the 1998 International Joint Conference on Neural Networks*, May 1998.

[31] Daniel A. Jiménez, Heather L. Hanson, and Calvin Lin. Boolean formula-based branch prediction for future technologies. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, September 2001.

[32] Daniel A. Jiménez, Stephen W. Keckler, and Calvin Lin. The impact of delay on the design of branch predictors. In *Proceedings of the 33th Annual International Symposium on Microarchitecture*, pages 67–76, December 2000.

[33] Daniel A. Jiménez and Calvin Lin. Dynamic branch prediction with perceptrons. In *Proceedings of the 7th International Symposium on High Performance Computer Architecture*, pages 197–206, January 2001.

[34] Daniel A. Jiménez and Calvin Lin. Perceptron learning for predicting the behavior of conditional branches. In *Proceedings of the International Joint Conference on Neural Networks (IJCNN-01)*, July 2001.

[35] Richard E. Kessler. The Alpha 21264 microprocessor. *IEEE Micro*, 19(2):24–36, March/April 1999.

[36] A. D. Kulkarni. *Artificial Neural Networks for Image Understanding*. Van Nostrand Reinhold, 1993.

[37] Eyal Kushilevitz and Noam Nisan. *Communication Complexity*. Cambridge University Press, 1997.

[38] C.-C. Lee, C.C. Chen, and T.N. Mudge. The bi-mode branch predictor. In *Proceedings of the 30th Annual International Symposium on Microarchitecture*, November 1997.

[39] Gregg Lesartre and Doug Hunt. PA-8500: The continuing evolution of the PA-8000 family. In *42nd IEEE International Computer Conference*, February 1997.

[40] Donald Lindsay. *Static Methods in Branch Prediction*. PhD thesis, University of Colorado, Department of Computer Science, 1998.

[41] Scott McFarling. Combining branch predictors. Technical Report TN-36m, Digital Western Research Laboratory, June 1993.

[42] P. Michaud, A. Seznec, and R. Uhlig. Trading conflict and capacity aliasing in conditional branch predictors. In *Proceedings of the 24th International Symposium on Computer Architecture*, June 1997.

[43] Ravi Nair. Dynamic path-based branch correlation. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, pages 15–23, December 1995.

[44] Soner Onder, Jun Xu, and Rajiv Gupta. Caching and predicting branch sequences for improved fetch effectiveness. In *International Conference on Parallel Architectures and Compilation Techniques*, October 1999.

[45] Sanjay J. Patel, Daniel H. Friendly, and Yale N. Patt. Critical issues regarding the trace cache fetch mechanism. Technical Report CSE-TR-335-97, Department of Electrical Engineering and Computer Science, The University of Michigan, May 1997.

[46] Harish Patil and Joel Emer. Combining static and dynamic branch prediction to reduce destructive aliasing. In *Proceedings of the Sixth International Symposium on High Performance Computer Architecture*, January 2000.

[47] Alex Ramirez, Josep L. Larriba-Pey, and Mateo Valero. The effect of code reordering on branch prediction. In *Proceedings of the 2000 International Conference on Parallel Architectures and Compilation Techniques*. IEEE Computer Society Press, October 15–19, 2000.

[48] Glenn Reinman, Todd Austin, and Brad Calder. A scalable front-end architecture for fast instruction delivery. In *Proceedings of the 26th International Symposium on Computer Architecture*, May 1999.

[49] Glenn Reinman and Norm Jouppi. Extensions to cacti, 1999. Unpublished document.

[50] F. Rosenblatt. *Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms*. Spartan, 1962.

[51] Eric Rotenberg, Steve Bennett, and James E. Smith. Trace cache: A low latency approach to high bandwidth instruction fetching. In *Proceedings of the 29th International Symposium on Microarchitecture*, December 1996.

[52] S. Sechrest, C.-C. Lee, and T.N. Mudge. Correlation and aliasing in dynamic branch predictors. In *Proceedings of the 23rd International Symposium on Computer Architecture*, May 1999.

[53] R. Setiono and H. Liu. Understanding neural networks via rule extraction. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence*, pages 480–485, 1995.

[54] André Seznec, Stéphan Jourdan, Pascal Sainrat, and Pierre Michaud. Multiple-block ahead branch predictors. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 116–127, October 1996.

[55] R. L. Sites. *Alpha Architecture Reference Manual*. Digital Press, Burlington, MA, 1992.

[56] James E. Smith. A study of branch prediction strategies. In *Proceedings of the 8th Annual International Symposium on Computer Architecture*, May 1981.

[57] 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 International Symposium on Computer Architecture*, June 1997.

[58] J. Stark, M. Evers, and Y. N. Patt. Variable length path branch prediction. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1998.

[59] Li C. Tsai. A 1GHz PA-RISC processor. In *Proceedings of the 2001 International Solid State Circuits Conference (ISSCC)*, February 2001.

[60] B. Widrow and M.E. Hoff Jr. Adaptive switching circuits. In *IRE WESCON Convention Record, part 4*, pages 96–104, 1960.

[61] Y. Ye, S. Borkar, and V. De. A new technique for standby leakage reduction in high performance circuits. In *Symposium on VLSI Circuits*, June 1998.

[62] T.-Y. Yeh and Yale N. Patt. Two-level adaptive branch prediction. In *Proceedings of the 24th ACM/IEEE Int'l Symposium on Microarchitecture*, November 1991.

[63] T.-Y. Yeh and Yale N. Patt. A comparison of dynamic branch predictors that use two levels of branch history. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, May 1993.

[64] Tse-Yu Yeh, Deborah T. Marr, and Yale N. Patt. Increasing the instruction fetch rate via multiple branch prediction and a branch address cache. In *Proceedings of the 7th ACM Conference on Supercomputing*, pages 67–76, July 1993.

[65] C. Young and M. D. Smith. Improving the accuracy of static branch prediction using branch correlation. In *Proceedings of ASPLOS VI*, pages 232–241, 1994.

[66] Reginald Clifford Young. *Path-based Compilation*. PhD thesis, Harvard University, Department of Computer Science, January 1998.

Chapter 10

Vita

Daniel Angel Jiménez was born in Fort Hood, Texas on September 1, 1969, the son of Dr. and Mrs. Angel R. Jiménez-Cerna. He attended Tom C. Clark High School in San Antonio, Texas, graduating in 1987. He received his B.S. degree in in Computer Science in 1992 and his M.S. degree in Computer Science in 1994, both from the University of Texas at San Antonio. He subsequently taught computer science classes at UTSA for two years. In 1996 he joined the research faculty of the Department of Rehabilitation Medicine at The University of Texas Health Science Center at San Antonio where he stayed for three years. He joined the graduate program in Computer Sciences at The University of Texas at Austin in September of 1995.