2011

Reducing complexity of processor front ends with static analysis and selective preloading

Santhosh Verma
Louisiana State University and Agricultural and Mechanical College, santosh3@gmail.com

REDUCING COMPLEXITY OF PROCESSOR FRONT ENDS WITH STATIC ANALYSIS AND SELECTIVE PRELOADING

A Dissertation

Submitted to the Graduate Faculty of the
Louisiana State University and
Agricultural and Mechanical College
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

in

The Department of Electrical and Computer Engineering

by
Santhosh Verma
B.S.E.E, Louisiana State University, 2003
M.S.E.E, Louisiana State University, 2008
December 2011

Acknowledgments

I would like to thank my advisor, Dr. David Koppelman, for always being available and providing insightful advice regarding not just this dissertation, but all aspects of my studies. He also provided me with a great deal of freedom in choosing a dissertation topic and in the direction of the topic. I would also like to thank Dr. Ramanujam for his support and help over several years, and for helping me diversify my experience into the compiler optimization area. I also want to thank Dr. Lu Peng for providing me with support and encouragement and useful advice on the direction of research I pursued in computer architecture. Further, I want to thank Dr. Jerry Trahan, Dr. Brygg Ullmer and Dr. James Van Scotter for being extremely accommodative in terms of being on the committee and dealing with scheduling issues. I also want to thank my family and friends for their support during this time.

Table of Contents

List of Tables

List of Figures

Abstract

General purpose processors were once designed with the major goal of maximizing performance. As power consumption has grown, with the advent of multi-core processors and the rising importance of embedded and mobile devices, the importance of designing efficient and low cost architectures has increased. This dissertation focuses on reducing the complexity of the front end of the processor, mainly branch predictors. Branch predictors have also been designed with a focus on improving prediction accuracy so that performance is maximized. To accomplish this, the predictors proposed in the literature and used in real systems have become increasingly complex and larger, a trend that is inconsistent with the anticipated trend of simpler and more numerous cores in future processors. Much of the increased complexity in many recently proposed predictors is used to select a part of history most correlated to a branch. This makes them costly, if not impossible to implement practically. We suggest that the complex decisions do not have to be made in hardware at prediction or run time and can be moved offline. High accuracy can be achieved by making complex prediction decisions in a one-time profile run instead of using complex hardware. We apply these techniques to Spotlight, our own low cost, low complexity branch predictor. A static analysis step determines, for each branch, the history segment yielding the highest accuracy. This information is placed in unused instruction space. Spotlight achieves higher accuracy than other implementation-simple predictors such as Gshare and YAGS and matches or outperforms the two complex neural predictors that we compare it to. To ensure timely access, we evaluate using a hardware table (called a BIT) to store profile bits after they are extracted from instructions, and the accuracy of using this table. The drawback of a BIT is its size. We introduce a novel technique, preloading that places data for an instruction in prior blocks on the path to the instruction. By doing so, it is able to significantly reduce the size of the

BIT needed for good performance. We discuss other applications of preloading on the front end other than branch predictors.

Chapter 1
Introduction

Modern processors were traditionally designed with a focus on maximizing performance (minimizing overall execution time) on a single core. However, as the power consumed by these processors has grown and multiple cores are placed on a single chip, the importance of designing individual cores with low cost (in terms of area) and low complexity has increased. Even though the number of transistors per chip is expected to continue to grow exponentially, since processors with hundreds of cores are expected in the future, it is important that the cost of each core (in terms of the hardware occupied) be kept as low as possible. Low complexity hardware is important because the total power consumption that is acceptable per processor has essentially peaked and generally speaking, more complex solutions dissipate more power and energy.

The front end of the processor is the part responsible for fetching instructions into the core. It comprises of the instruction cache, the branch predictor, the jump predictor and either a branch target buffer or a basic block table. This dissertation is mainly focused on the branch predictor, although the techniques we develop could easily be used for all other parts of the front end.

Branch predictors are an important component in the design of modern processors. Branches are instructions whose outcomes determine which of two possible paths a program can take and therefore, which instruction should be fetched next. Since the instruction fetch unit needs the branch outcome in the same cycle that a branch is fetched and computing the actual outcome takes many clock cycles, branch predictors are used to make an early prediction and avoid program stalls. However, a mispredicted branch results in the incorrect fetch of many instructions along the wrong program path. This results in reduced performance because clock cycles that could otherwise be spent fetching correct path instructions are

wasted. Further, the energy spent in the fetch, decode and execution of these wrong path instructions is also wastted [23]. Therefore, highly accurate branch predictors are important in achieving both high instruction throughput and low energy consumption.

While branch predictors can improve execution time and reduce wasted energy by minimizing the number of wrong path instructions that are processed, the predictor itself can be costly and complex. Early branch predictors were simple in terms of their hardware complexity. For example, the bimodal predictor consists of a single table of two bit saturating counters. A branch uses its program counter (PC) bits to access an individual entry and makes a prediction based on a single bit of the counter's value. The Gshare predictor uses a similar table of saturating counters but indexes the table using a combination of PC and branch history bits, called the global branch outcome history. Other predictors such as YAGS[4] deliver higher accuracy at slightly greater complexity but are still relatively simple. However, in order to continue the trend towards higher accuracy, the branch predictors proposed in literature and those used in real systems started becoming larger and more complex over time. A new class of predictors that have been proposed in recent years provide extremely high accuracy but require complex hardware and are unable to deliver single cycle predictions without using techniques that may require even further complexity. For example, neural based predictors such as the Hashed Perceptron [34] and the Path-based Neural [11] require complex computations and many table lookups that result in a latency of several cycles per prediction if clock frequencies are to be preserved. These approaches requiring complex hardware are counter to the widely anticipated trend in future multi-cores of simpler and more numerous cores and the need for low-cost hardware. Further, a single cycle latency is difficult to achieve without using either a faster first-level primary predictor whose prediction may be reversed later by the main branch predictor or other complicated schemes like ahead pipelining [12, 24, 28, 34].

These complex predictors can be seen as doing two things: discovering correlations between a branch outcome and some kind of history, and using such correlations to make a

prediction. Since these correlations are determined primarily by program structure (as opposed to program input), it might be possible to discover such correlations in advance and then deliver this information to a much leaner predictor, one that would not need hardware to discover correlations.

Profile-based branch predictors are hybrid static/dynamic schemes that use some information determined before program execution to aid in the prediction process. A well designed profile-based branch predictor could have the potential to attain high accuracy with simple hardware by performing the more variable, data dependent parts of the decision making process in the flexible hardware while offloading the less variant, data independent parts of the process to software that runs before program execution. The performance of the neural predictors suggest that complex computations are difficult to avoid if higher prediction accuracy is to be achieved. While these computations may not be avoidable, they may not have to be done at prediction time or even at run time. Instead, the complexity can be moved from hardware to software using profiling. Typically, profiling involves performing an initial training run on a representative training input in order to collect some general information (such as the percentage of times that a given branch is taken) about the program and using this information to assist in the decision making process during the actual execution of the program. In branch predictors, profiling has the potential to reduce the complexity of the on-chip hardware without sacrificing accuracy by moving some of the complexity of the branch prediction process to an offline, software step.

One important question is how to place the profiled data needed by the hardware. The simplest approach would be ISA (Instruction Set Architecture) based, where the format of branch instructions is extended so that they have a designated field for profiled data. A less costly approach that can avoid extending the instruction is to modify the format of existing branch instructions so that free space due to underused or unused bits (such as some bits in the branch displacement field) are used to store profile bits. Custom carry-only instructions can be added to the ISA, but in order avoid increasing the code size, the preferred method

would be to use special versions of existing instructions. Free space can be used, not just within a single branch instruction but other instructions within the control block of the branch instruction. This is more complicated but increases the amount of profiled data bits that may be stored.

The other major challenge is the placement of the data obtained during profiling in a manner that allows it to be accessed easily and quickly enough to make an on-time prediction. The most cost effective and obvious way of placing profiled data would be within the instruction of the branch being predicted. However, waiting for an instruction within the block being predicted to be decoded to extract the data is not feasible, since by then it is too late for the front end of the processor to access the data (a branch needs to be predicted in the same cycle that it is fetched). For profiled data to be accessed in time, the information has to be placed within instructions in *prior branches* or *control blocks* (a control block is a block of instructions which contain the branch instruction) in the instruction stream. This can provide enough of a head start so that the data is ready to be used by the time the branch being predicted is fetched. But in which previous control block (or blocks) do we place data for the current branch, and what happens if more than one branch wants to use a given block? We refer to this key problem of where profiled information should be placed in an optimal manner as the placement problem.

Our investigation started with a profile-based predictor called Spotlight [38]. A low-cost, single-cycle version was developed and characterized, in particular in a search for low cost configurations. The main focus of the investigation was to develop methods of delivering profile information to a predictor, including Spotlight, shortly before the need for this information, without requiring large tables and without increasing instruction fetch bandwidth.

Chapter 2
Background and Related Work

Branch instructions control the flow of programs by determining which instructions should be fetched after the branch to continue program execution. In order to prevent stalling the fetch of instructions, the fetch unit needs the outcome of a branch in the same cycle that the branch is fetched. Since it is not possible to compute the actual branch outcome in the same cycle that it is fetched, branch predictors are used to predict the branch outcome. After a branch is fetched, it needs to be decoded and executed before its actual outcome is determined and this process can take many cycles. An incorrect prediction can be corrected after the branch executes, but this results in the time between the fetch and execution of the branch being wasted fetching, decoding and executing other instructions which will later have to be squashed. The result is not only a waste of clock cycles (and hence a loss of performance) but also a waste of energy due to the processing of unnecessary instructions. In addition to predicting the direction of the branch, the target of the branch (which instruction is fetched next) is also predicted in processors. The focus in this dissertation is on branch direction prediction.

Branch prediction is a widely studied topic, but we will concentrate on certain aspects of earlier work. We look at how predictors became increasingly more complex as higher accuracy was sought as a means to drive performance improvement. We also look at how the more complex predictors use complex pipelining schemes in order to achieve on-time prediction without sacrificing clock frequency. We also focus on the fact that much of the complexity of these predictors is directed towards finding the part of history that is most correlated to a branch. We look at how previous profile-based predictors use profiling to do some parts of the prediction that would normally be done in hardware.

2.1 Early, Low Complexity Branch Predictors

The Bimodal predictor is the simplest, hardware based branch predictor both in terms of concept and hardware complexity. A branch instruction accesses an entry in a Branch History Table (BHT) using its program counter (PC) bits. Each table entry is a two-bit saturating counter (range of possible values are ≥ 0 and ≤ 3) that is incremented on a taken branch and decremented on a not-taken branch. A prediction is made based on the entry's value, a value ≥ 2 results in a taken prediction while a lower value results in a not-taken prediction. The bimodal predictor can be seen as a majority predictor, where a branch's predicted outcome is based on whether its recent history indicates that it is mostly taken or mostly not-taken. . The Local predictor uses two table lookups. The first table is indexed using the branch PC bits of the branch being predicted and contains the most recent branch outcomes. The value in this first table is used to index a Pattern History Table (PHT) of two-bit saturating counters. The PHT is similar to the BHT but stores a history of local branch patterns. The local predictor's name is derived from the fact that it predicts a branch's outcome based on the recent local history of that branch. The Gshare predictor has almost the same hardware complexity as the Bimodal predictor, but is conceptually more sophisticated. It uses a PHT which is indexed using the most recent set of global branch outcomes (as opposed to the per instruction local history of Bimodal and Local) stored in a global history register (GHR). The GHR is exclusive OR'd with branch PC bits to reduce conflicts in the PHT table. Gshare exploits the idea that a branch's outcome is correlated to other branches preceding it in the instruction stream and delivers higher accuracy compared to Bimodal, despite having similar hardware complexity. The drawback with Gshare is the amount of table conflicts (called aliasing) in the PHT that occur when multiple branches access the same PHT entry. The Bi-Mode branch predictor [16] converts destructive aliasing with neutral aliasing by dividing the PHT into three equal parts. Two direction PHTs (called taken and not taken) are indexed using branch PC and global history bits, with the goal being to use one PHT for branches which are mostly taken and the other for branches which are mostly not-taken.

A choice PHT indexed with just the branch PC bits is used to decide which of the two direction PHTs to use. Since only the direction PHT selected by the choice PHT is updated, destructive aliasing is reduced. The YAGS branch predictor [4] mitigates the aliasing problem by using a bimodal component combined with two choice PHTs that operate similarly to the PHT in Gshare. One choice PHT is used for branches that are biased taken and the other for biased not-taken branches. The bimodal component is first accessed to determine the bias of a branch, and this bias determines which of the two choice PHTs is accessed. A branch that is biased taken will check the not-taken choice PHT, Instead of providing an actual prediction, the choice PHTs indicate whether a branch disagrees with its bias. Since branches will agree most often with their bias, the amount of conflict in the choice PHTs is significantly lower compared to the PHT in Gshare.

2.2 Predictors with Greater Complexity

While the previously mentioned predictors understandably increase in complexity and size as accuracy increases, they are still relatively simple and practical relatively low-cost implementations of these predictors is possible. However, more recently proposed predictors, while delivering improvement in accuracy, have become so complex that practical implementations are difficult. Even if implemented, these predictors are extremely costly (in terms of hardware budgets and energy consumed) and cannot deliver on-time single cycle predictions without using even more complex hardware mechanisms. Seznec [25, 27] analyzes the O-GEHL predictor. This predictor can explore very long history lengths up to 128 bits but needs to compute the sum of M items to make prediction. Recent work using perceptrons improve predictor accuracy by using longer global histories without exponentially increasing hardware size. The idea of neural branch prediction was originally introduced by Vintan and Iridon [39]. Jiménez *et al.* [10] propose a dynamic Perceptron predictor that exploits long branch histories by increasing hardware table size linearly with the history length. A hash function of the branch address is used to select a single perceptron from a table and an

output is computed by using the dot product of the perceptron and global branch history. However, the global perceptron has to determine the hash of the branch address and the dot product of the perceptron and global branch history, a process that can take up to 4 cycles even in an efficient circuit level implementation of the predictor [14]. To counter this problem, Jiménez [11] proposes the Path-based Neural Predictor which computes a running sum along the path leading up to each branch. This makes it possible for the latency to be reduced from four to two cycles. However, the Path-based neural predictor is extremely complex in its use of anywhere between 14 and 34 weight tables to make a prediction. Tarjan *et al.* [34] improve on the linear scaling of previous perceptron predictors by assigning multiple branches to a single weight in their Hashed Perceptron scheme. This predictor is able to outperform the path based neural predictor, hence demonstrating that the one-to-one mapping between weights and branch history bits that is used in other Perceptron predictors is not necessary. The neural approaches to branch prediction obtain high accuracy but are clearly very complex (especially when compared to simple predictors like Gshare) and difficult to implement with a single cycle latency because of the need for relatively complex calculations before arriving at a prediction. The Alpha EV6 [26] solves the latency issue by using a simple quick predictor which is backed up by a larger and more accurate predictor.

Another approach to solving the latency problem is to use ahead pipelining [12, 24, 28, 34], where the computation required to make a prediction in a particular cycle is started several cycles earlier using the information available in the earlier cycle. However, these schemes do not mitigate the complexity of the hardware itself and in fact add to the complexity. There can also be a significant drop in accuracy when the "ideal" one-cycle latency versions of these predictors are compared to the pipelined versions. Furthermore, Tarjan et. al [34] point out that pipelined versions of branch predictors will need to checkpoint intermediate predictor data to account for the recovery process after a branch misprediction. The amount of data that needs to be checkpointed is dependent both on the complexity of the predictor and on

the number of pipelined stages that the predictor uses to make a prediction. In [34], Tarjan et. al note that Perceptron-based predictors need to checkpoint a significant amount of data.

2.3 Improving Prediction Accuracy by Determining Most Useful History Segment

2.3.1 Dynamically (at Run Time)

In addition to O-GEHL and the Neural predictors, many other predictors also seek to improve accuracy by determining which part of branch history is useful. Juan *et al.* [8] propose Dynamic History Length fitting, where the length of the GHR used is varied dynamically until one that works best for the code in execution is found. Ever et. al [3] explore global histories up to a size of 32, were up to three of the most important branches over this history length are selected using an oracle mechanism for use in a hypothetical predictor. Their results indicate that considering longer global histories up to 20 bits can improve performance. Thomas *et al.* [36] use run-time data flow information to identify correlated branches over a long global history. An Affector Register File (ARF) contains dataflow information for every architectural register. Each entry in the ARF is a bit vector, where a bit represents whether a branch in the global history affected the architectural register. For each branch, an Affector Branch Bitmap is created by combining the affector registers of the branch's source operands and this bitmap is combined with the GHR to form the predictor index. Their technique uses 64 bits of GHR which are hashed down to the required number of index bits by using a fold and XOR hash.

2.3.2 Using Profiling

The O-GEHL, Neural-based, Variable Length Path Predictor and the ARF predictor spend a lot of hardware resources to determine which part of prior branch history is useful. Preliminary work has shown that this cost can be vastly reduced by using profiling (instead of hardware) to make this decision and by designing an efficient mechanism to place profiled data within the instruction stream. Tarlescu *et al.* propose the Elastic History Buffer (EHB) [35] predictor where the length of history used to predict a branch can vary and is determined

using profiled data. Stark *et al.* [33] propose Variable Length Path Branch Prediction which use profiling to select one of N hash functions for indexing a prediction table, where a hash function $k \in \{1, 2, \ldots, N\}$ uses the target address of the k most recent branches to produce the hash. Similar to this proposal, they also propose storing profiled information in branch instructions, caching this data at runtime in a table and organizing this table so that entries store data for downstream branches. However, they only suggest such an approach and do not simulate or analyze the accuracy, costs and challenges that such a scheme poses. Further, computing several hash functions and selecting amongst them involves complex hardware because each of these hash functions requires several rotations and XOR's to be performed on different target addresses in each cycle.

2.4 Other Profile Based Predictors

In addition to the predictors mentioned above, profiling has been used by others to aid in branch prediction. McFarling and Hennessy [19] investigated two static based branch prediction schemes, the first of which predicts all branches in a program the same way and the second one which predicts individual static branches in a program the same way, with the predictions determined by profiling. They reported accuracies greater than 60% for the former technique and accuracies greater than 80% for the latter technique. Chang *et al.* [2] use profiling to classify branches based on their dynamic taken rates. Profiled data is used to predict highly biased branches statically. This allows them to optimize the hardware predictor by reducing PHT conflicts between highly biased and mixed direction branches. Sprangle *et al.* [32] propose the Agree predictor and investigate a version of the predictor that uses profiled data to set a bias bit in the branch instruction. PHT conflicts which are harmful are reduced by updating PHT entries based on whether a branch agrees or disagrees with its hint bit, instead of taken and not taken rates. Path profiling techniques collect information on the paths taken during the execution of a program. Jiménez *et al.* [9] use path profiling to assign paths to PHT entries so that similarly biased paths are mapped to

the same PHT entry. Each branch has an inversion bit which may be set during profiling if two paths with different biases map to the same PHT entry. During the actual run of the program, a branch's outcome is inverted before it is recorded in the global history register if its inversion bit is set. Young *et al.* [43] propose static correlated branch prediction which uses path profiling to increase the accuracy of static branch prediction by duplicating program blocks and encoding information about the path taken to reach a particular block. Most of these previously proposed profile-based predictors ideally assume that profiled information can be obtained directly from the branch instruction in time for a prediction. This is clearly an impractical assumption which we discard in this work.

Chapter 3
Profile Based Branch Prediction

The previous chapter has an overview of the history of profile based branch predictors. This chapter takes a detailed look at some of these predictors, including our proposed Spotlight Branch Predictor [38] and discusses why these predictors can be useful.

3.1 Benefits of Profile Based Predictors

At first glance, it seems obvious that improvements in branch prediction accuracy cannot be achieved without increasing the complexity and size of the prediction hardware. This is the case when one considers the evolution of branch predictors discussed in the background chapter. Predictors such as the Neural-based predictors and O-GEHL are very accurate but so complex that practical implementations of those predictors are extremely costly if not impossible.

Another requirement for a processor with a branch predictor is a mechanism for misprediction recovery. When a misprediction is detected, the processor has to squash all subsequent instructions and restore the state of the front end back to what it was just after the mispredicted branch. The amount of state that has to be saved and the complexity of the recovery is dependent on the complexity of the predictor itself. This is another reason for increased complexity in neural predictors compared to simpler predictors.

We observed that these predictors use a significant amount of hardware resources trying to identify which parts of history are most correlated and useful in predicting a branch. For example, the Perceptron predictors assign weights to different Sections of the global history and these weights are trained to determine the most useful parts of this history. Not only do you need multiple tables of multiple-bit entries to store these weights, the complexity of these schemes is high because of the need to add several non-positive integers. If those table

values could be pre-computed and then loaded when needed, then a complex part of the predictor could be eliminated.

We suggest that while it may not be possible to get much improvement in accuracy without increasing complexity, some of the complexity can be moved away from the hardware to an offline step done in software before program execution. Higher prediction accuracies can be achieved with relatively simple hardware if this can be done successfully. This is the motivation behind the Spotlight predictor, one of the main contributions of this dissertation.

Many profile-based predictors such as the Agree Predictor have been proposed. These predictors use profiled data to replace some of the decision making that is normally done in hardware, and so achieve improvements compared to the non-profile based predictors of comparable hardware complexity. The Spotlight predictor is a robust profile-based predictor which uses aggressive profiling to make complex decisions away from the hardware. Specifically, profiling is used to precisely select the Section of history that is most correlated to the predicted branch. We show that this approach can reduce predictor complexity, since the complex predictors perform this aspect of prediction in hardware and also show that high accuracy can be achieved despite the simpler hardware. In the next two Sections we describe the Agree and Elastic History Buffer predictors and show how the Spotlight predictor is a natural next step in the evolution of profile-based predictors.

3.2 Static Branch Predictors

Static branch predictor schemes predict branches without using any hardware. One option is to predict individual static branches the same way (always taken or always not taken) for all dynamic instances of the branch during program execution. This technique can deliver reasonably high accuracy, greater than 80%, as reported by McFarling and Hennessy [19]. Young *et al.* [43] propose a static scheme that improves accuracy by using compiler based techniques such as path profiling, duplicating program blocks and encoding path information about the path taken to a branch. In such a scheme, a static branch may be predicted either

taken or not taken depending on the encoded information. As discussed in Section 2.4, Chang et. Al [2] use static prediction but only for branches that are highly biased. This enables them to make better use of the hardware scheme, Gshare, by reducing PHT conflicts. Some ISA's such as the Sparc V9 [41] have hint bits associated with branch instructions that could be used in a static prediction based branch predictor. While a static scheme seems attractive, at best an accuracy of 90% can be attained.

3.3 The Agree Predictor

The basic Agree predictor [32] is shown in Figure 3.1 and has been used in real systems such as the HP-PA 2700 [17, 37]. In terms of hardware complexity, the predictor is almost identical to a Gshare predictor. The only addition is a single XOR gate that XOR's the outcome from the rest of the prediction scheme with a predicted biasing bit (shown in the oval). Sprangle et. al [32] simulate two versions of the Agree predictor, one version in which the biasing bit is determined by the direction a branch takes during its first occurance, and another version in which it is selected using profiling by looking at whether a static branch is mostly taken or not-taken. The base predictor for Agree is Gshare, which is described in Section 2.1. While Gshare increments or decrements the two-bit saturating counters in the PHT based on whether a branch is taken or not taken, the Agree scheme modifies the counters based on whether a branch agrees or disagrees with its own bias. Consider two branches that both map to the same PHT entry, with one of these branches being mostly taken and the other mostly not-taken. In a traditional Gshare, the two branches may fight for the entry. That will be the case if the two *colliding branches* have approximately the same frequency and access the entry alternatively. When this occurs, the mostly taken branch pushes the counter to the maximum while the mostly not-taken branch pushes it towards the minimum. The result is that neither of these colliding branches are predicted as accurately as they would be if they each had a unique entry. The Agree scheme mitigates this problem because the PHT entries indicate whether a branch agrees or disagrees with its bias, instead of indicating

the absolute outcome of the branch. As such, both the mostly taken and mostly not taken branch are expected to, by definition, agree with their bias most of the time. While there will still be conflicts when the actual outcome of one branch does not agree with its bias, the number of conflicts are greatly reduced. As a result, the Agree scheme can improve on the accuracy of the base Gshare predictor by using profiling and a small increase in the hardware itself, particularly at smaller PHT sizes (since there are more collisions at smaller sizes).

The Agree predictor is simpler than other predictors such as YAGS [4] and Bi-Mode [16], both of which reduce destructive aliasing in the PHT. Both of these predictors reduce destructive aliasing by dividing the PHT into multiple parts. One table is similar to Bimodal in that it is indexed using just the branch PC and essentially determines the biasing bit for these predictors. Two additional tables, direction PHTs, are indexed using a combination of branch PC and global history bits. One direction PHT is a *taken* PHT and the other is a *not-taken* PHT. The relative simplicity of the Agree predictor is due to its use of profiled data (or easy to determine information such as the direction a branch takes during its first occurance). The advantage of these other predictors is that they can deal with situations where the bias of a significant number of branches flips back and forth. In contrast, a flip from one direction to another would reduce prediction accuracy for a while (compared to YAGS) if only one PHT is used, as in the Agree predictor.

3.4 The Elastic History Buffer Predictor

The Elastic History Buffer (EHB) [35] belongs to the class of predictors that chooses, on a per-branch basis, the part of the outcome history to use, effectively taking correlation into account. The basic idea is that the amount of correlated history needed to predict a branch accurately varies on a per branch basis. Some branches require longer histories to be predicted accurately (for example, consider a branch inside a loop that correlates to branches outside). Other branches require a much shorter history, and using longer histories will only hurt accuracy. This is because more history often means that more PHT entries are used, all

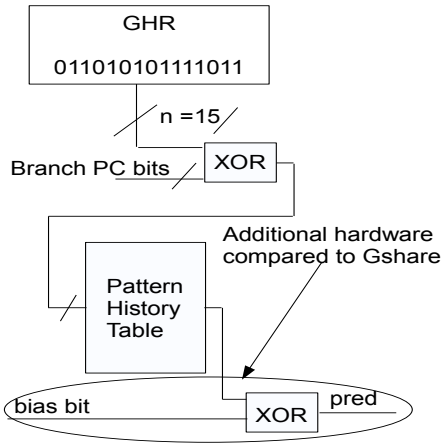

FIGURE 3.1: The Agree Predictor

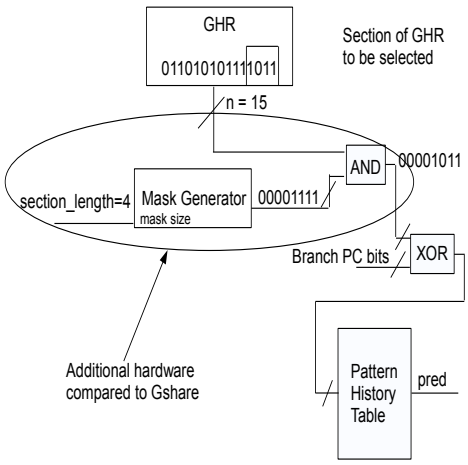

FIGURE 3.2: The Elastic History Buffer Predictor

of which must adjust to the branch. Further, because of the greater number of PHT entries used, there is a greater chance of collision with other branches. The combined phenomenon due to these two factors is called *warmup*.

The EHB determines a custom GHR history length for each static branch in the program. This is determined during profile runs by testing a range of lengths for each static branch and selecting the one that works best (is the most accurate). The hardware for this scheme is shown in Figure 3.2, with Section length representing the history length. Once again, the complexity does not increase much compared to the base predictor (the addition to the base predictor is shown in the oval). The only addition is a mask function that selects the lowest N bits of the GHR, where N is the history length mentioned above.

The EHB is another scheme that significantly improves on the accuracy of the baseline predictor without a major increase in complexity. It does so by removing the complexity of determining the most related correlated global history using profiling, instead of the hardware. The dynamic history length fitting scheme of Juan *et al.* [8] is similar, but determines this correlation in hardware. This is done dynamically by computing the number of mispredictions over program intervals and dynamically adjusting the length at the end of each interval based on the current number of mispredictions and the minimum encountered until

that point. This requires a misprediction table, with a size equal to the number of bits used in indexing the PHT. Two pointers are needed, one pointing to the entry in the table representing the segment that has the minimum misprediction count and another pointing to the entry representing the segment with the current misprediction count. Two counters are also needed, one of which counts the number of mispredictions for the current interval and another that counts the total number of predictions in the current interval (to determine when an interval starts and ends).

3.5 The Spotlight Branch Predictor

The basic motivation for our Spotlight predictor [38] is that the segment of the GHR that best correlates with the predicted branch can range widely both in terms of its distance from the predicted branch (its starting point) and the length of the segment from this starting point. The ability to vary the starting point makes Spotlight a logical improvement over EHB. Spotlight focuses on finding the best correlated region that gives the highly accurate predictors such as the neural predictors their accuracy. Figure 3.3 shows the basic structure of the Spotlight predictor along with an example prediction. We refer to this version of Spotlight as *Spotlight-Direct*. The predictor uses an n bit global history register

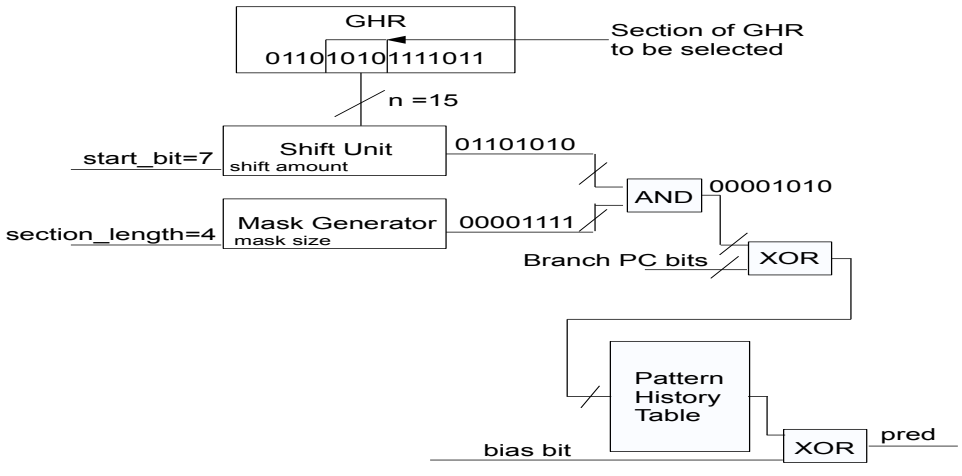

FIGURE 3.3: Spotlight Direct

as its first level of history. For each prediction that needs to be made, a start_bit and a

Section_length is provided to the predictor. The start_bit represents the starting point of the GHR Section that will be used for a prediction, and the Section_length represents the length of this Section. The start_bit and Section_length are used to perform a combination of shift and mask logical operations on the global history so that a particular Section of the GHR is selected (spotlighted). The example in the figure shows how a GHR Section starting at the 7th bit with a length of 4 bits is selected. The selected segment of the GHR is XOR'd with the branch PC and this result is used to index a pattern history table (PHT) of two-bit saturating counters. The start_bit and the Shift operator are what distinguishes Spotlight from the EHB. Hence, the Spotlight predictor can use GHR bits starting at any point between 0 and $n-1$ with any length between 0 and the log_2 size of the PHT.

The cost of determining the start_bit and Section_length dynamically in a pure hardware scheme would be extremely high and might suffer from long warmup times. Spotlight avoids these issues by determining this information using profiling. The methods used for profiling, analysis and discovery will be covered in the next chapter.

Instead of using the PHT counters in the traditional Gshare approach, we use the Agree scheme [32]. A bias is stored with each branch and the PHT counter for that branch indicates whether its outcome is expected to agree or disagree with its bias. Since Spotlight already uses profiled information to make predictions, the cost of including the Agree scheme is minimal while providing some performance benefit to the predictor.

When comparing Spotlight to Agree and EHB shown previously, we can clearly see Spotlight as the next step in the evolution of profile based predictors. Spotlight combines both the benefits of Agree and EHB but provides a further improvement in terms of its ability to select a GHR segment of any length starting not just at the most recent branch, but at a point earlier in the code. As such, it is capable of looking further into history than the EHB. In terms of hardware, it is slightly more complex than the EHB due to the additional shift operation, but the overall complexity is still simple compared to more complex schemes such

as Neural predictors. Because of its simplicity, Spotlight is also able to perform well at very small PHT sizes.

3.6 Summary

This chapter describes the potential benefits of profile based predictors, particularly in the context of the trend in computer architecture of simpler, more numerous cores. The profile based predictors here are simple compared to the baseline predictor (Gshare) that does not use profiling. These predictors have the potential to significantly outperform the baseline predictor with minimal additional hardware by moving the complexity offline. We will see that a significant advantage of profile based predictors is that they perform particularly well at smaller hardware budgets. This has the potential to make them suitable for future processors, where a large number of cores is expected and as such, the hardware cost of each core needs to be kept low. Another advantage of profile-based predictors is their flexibility. By making significant decisions in software, it is possible to implement various profile based predictors on the same hardware. For example, the Spotlight hardware shown in Figure 3.3 can be used as an Agree predictor by setting the start_bit to 0 and Section_length to log_2 of the PHT size. It can be used to implement the EHB by setting the start_bit to 0. The choice of which predictor works best can be determined by the compiler based on an understanding of the program. Spotlight can be seen as a superset of EHB and Agree, so an architecture with Spotlight can be used to implement any of these three predictors based on what the compiler thinks is appropriate. Further, improved prediction techniques can be implemented and deployed with just a software update.

Chapter 4
Instruction Profiling in the Real World, Profiling, Analysis and Annotation (PAA)

As discussed in the previous chapter, the various profile based branch predictors need typical branch behavior data, information that is typically collected using profiling. Spotlight uses profile-collected data to determine a start bit, a Section length and a biasing bit, collectively called a configuration, for each branch; it also needs to choose a program-wide default predictor. This will be the fall back configuration for a branch which does not have custom profiled data available (Section 4.2 describes how it is obtained). The EHB requires a section length, and the original Agree predictor is the simplest, requiring a single biasing bit per branch. The *profiling, analysis, and annotation (PAA)* steps cover the process of extracting this type of information for a program and preparing an executable based on it. The simulator we use models a general purpose processor, and the benchmarks we use are typically executed on this type of processor. However, due to the current importance of embedded systems, we describe how PAA would be implemented for both a general purpose and embedded system. The PAA process starts with a program's executable and possibly a set of training data (training data can be omitted using dynamic PAA, in which live data is used). The process finishes with an annotated binary.

In the *profile step*, a *Control Transfer Instruction* (CTI) trace is extracted for some sampling of the program's execution. CTI's refer to branch, jump, call and return instructions. The trace consists of the PC and outcome (taken or not taken) of each dynamic branch. In the *analysis step* the trace is analyzed to determine the configuration for each branch and to choose a default predictor. Finally, in the *annotation step* the configuration data is used to prepare an annotated version of the original program executable. The annotated version includes *carry instructions* which carry and deliver branch configurations to the BIT. If a carry instruction were inserted for each branch there would be a significant increase in code

size, since branches can make up more than one out of every ten instructions. Instead the configuration data, which amounts to just 10 or 11 bits for the Spotlight configurations described so far, will be carried by variants of existing instructions which perform their original function but also carry the data. When such an instruction is decoded the configuration data will be loaded into the BIT, for use the next time the branch is encountered. Each of these steps is elaborated on in the Sections that follow.

PAA can be performed using commonplace code development tools and techniques since it is a form of feedback-directed optimization; that will be called *Build PAA*. Two *Dynamic PAA* systems will be described. Both, by sampling execution, impose a light load on the program. *Development Dynamic PAA* is for use within a development environment, allow the tracing of programs for which compiler profiling techniques would interfere too much with execution for a meaningful run. The end product of development dynamic PAA is an annotated binary for placement on the targeted system. The more interesting scheme, though feasible and of benefit for fewer systems, is *Live Dynamic PAA* in which PAA is performed continuously in the target system itself.

4.1 Profiling and Sampling

Profiling can be used to collect all kinds of information about a running program. For brevity, the term profiling used here will specifically refer to the process of extracting a CTI trace from a running program. For build-PAA, profiling will be done in a specially instrumented build prepared for this purpose. That build will be prepared using compiler options similar to those used for other sorts of profiling that have been standard features of compilers for some time (used for feedback-directed compiler optimization and as a programmer's aid). The build will have code at each basic block appending CTI outcomes to a trace file, though efficiently buffering this data to reduce overhead. The build would be run on training data, followed by a second build in which analysis of the trace will determine where to use carry instructions and what they should carry.

Such a trace collection scheme is not suitable for live PAA, and might even be too cumbersome or too slow for collecting data on a development system. The dynamic PAA schemes rely on sampling of a running program to reduce impact on its execution. *Sampling* is the repeated collection of trace data during at least one execution of a program. The duration of a sample will be kept short to minimize impact and avoid the need to immediately write the trace buffer. The time between samples is made long enough to avoid disturbing the program, and the number of samples, which might be collected over several runs, is made large enough to be significant.

Trace collection can be scheduled by setting a timer interrupt for some appropriate time. When the interrupt occurs trace collection would start by continuing execution with a functional simulator that recorded the trace, or special hardware would be set to collecting the trace information into a hardware buffer for later harvesting. At the end of the sample the timer would be set for the next sample. Even the best functional simulators are several times slower than native code, and so any system with real-time constraints would need hardware support. Each sample would append more CTIs to the trace collected for a program, eventually there would be enough for profile analysis. By limiting trace size and relying on hardware support this sampling can be made as unobtrusive as necessary. Tracing techniques, of which there are many, have been employed for decades and a detailed discussion is beyond the scope of this paper.

4.2 Analysis of Profiled Information

For the Analysis step, the trace data collected in the Profiling step is analyzed by predictor specific code to generate carry data for the Annotation step. This Section describes how this step is carried out for the Spotlight predictor.

Consider a Spotlight system using a GHR of length n and a PHT of size $P = 2^p$ entries, with $n > p$. The number of distinct configurations for each branch, denoted C, is $1 + (n - p)p + \sum_{i=0}^{p}(p - i)$, where the first term is the bimodal (zero GHR bits) configuration, the

second term considers all GHR segments of lengths $[1, p]$ starting at positions $[0, n-p]$, and the last term considers segments near the end of the GHR (for which there are fewer lengths to explore because longer ones would exceed the GHR length). Simplifying, we get $C = 1 + np + \frac{p}{2}(1-p)$ or a size complexity of $O(np)$.

Let B denote the number of static branches in a program. The goal of profile analysis is to choose the configuration for each one of these that minimizes the number of branch mispredictions. Because of PHT collisions the choice of a configuration for one branch can affect the accuracy of another, so an exhaustive search would have to consider the branch prediction accuracy for an entire program execution on each of C^B configurations, much much too large. With the assumption that PHT collisions do not occur, one can choose a configuration for each branch separately, selecting the one that minimizes the number of mispredictions. That reduces the search space to $CB = O(npC)$. The time for such a profile analysis would be $O(npT)$, where T is the time needed to analyze the performance of one predictor over the other, which is feasible. If a PHT were provided for each branch the amount of space needed would be $O(Bnp2^p)$, which is large. The approach we use here for profile analysis reduces the amount of space and roughly accounts for PHT collisions as described below.

Since the predictor can select any part and any length of the GHR and the GHR can be larger than the PHT address, our profiling algorithm has to explore each of the C distinct configurations. While the reader may be concerned about the exhaustive nature of this method, profiling is only a one-time step that would take several hours per program and can be done in an unobtrusive way when the system is not busy. Heuristic methods may be possible, such as testing coarser regions in an initial phase and finer regions in a second phase. We did not see the need for such methods because the speed of profiling was not an issue, the longest profiling simulation for the training inputs took about 10 hours with most benchmarks finishing much faster.

During profiling, a single PHT is instantiated for each combination of start bit, *sb*, and section length, *sl*. The log size of an individual PHT is determined by the function $min[min\_pc\_bits, sl]$. The minimum number of PC bits, *min_pc_bits*, is used for the PHTs to reduce conflicts between branches when the GHR segment for that PHT is very short. The PHTs are indexed using the XOR of a particular GHR segment and the PC of the branch being predicted. In this way, there is one PHT for each GHR segment and this PHT tracks the performance of that segment throughout the profiling benchmark. When a branch instruction is executed during the profile run, every PHT provides a prediction for that branch. The prediction of each PHT is compared to the actual outcome of the branch and prediction accuracy data is collected for the branch over all PHTs (and hence over all possible GHR segments).

At the end of the profiling run, we have the prediction accuracy over all of the C configurations for each static branch that was executed during profiling. Since each PHT represents a specific combination of *sb* and *sl*, we can select the *sb* and *sl* for a branch based on the most accurate PHT for that branch during the profile step. The biasing bit required for the Agree mechanism is straightforward to obtain, since it simply indicates whether a branch is taken or not-taken most of the time. The default configuration is also selected for a benchmark in this step. This is done based on whether the Gshare configuration or the Bimodal configuration performs better for branches which do not have a BIT entry.

4.3 Storing Configuration Information in Carry Instructions

Branches' configurations will be carried by special implementation dependent instructions that duplicate the functionality of existing instructions except that part of immediate fields and unused fields will be used to hold configurations. The obvious place to put the data is in the block with the branch, but as we will see in later chapters, there are reasons to put it upstream in previous blocks leading to the branch. The predictor hardware will extract

the configurations from such instructions and write them into the BIT. There are many such candidate instructions, consider for example immediate ADDs. An ISA might leave 16 or more bits for an immediate but in typical code the fraction of immediate ADDs that use numbers representable in four or fewer bits is large. If the ISA has a free opcode in the format used for an ADD, that could be used for an implementation-dependent version with a four-bit immediate field and with the remainder of the "old" field used to carry configuration information. One would define enough of these implementation-dependent instructions to carry the configuration data needed by Spotlight.

Configuration data for a branch would be carried by instructions either in the branch's basic block or in previous blocks leading up to the branch. If there was not enough space in the block a fallback configuration would be used for the branch. If the configuration information is sufficiently small, a single instruction will have enough space to store it. Fortunately, Spotlight (as described at this point) enjoys this advantage. This eliminates the need for complex hardware to combine configuration information spread over several instructions, and only one or two new instructions are needed.

An analysis of the ability of an ISA to support carry instructions was performed for SPARC V9 [41]. Branch instructions have a 19- or 22-bit displacement field (the 19-bit displacement is in instructions that can access one of several condition code registers). The format used for branch instructions encodes five types of branches, a non-branch, and two unused instructions. Those unused instructions can be borrowed for branch instructions that have, say, a 12-bit displacement and 10 bits of Spotlight storage. Two implementation-dependent instructions are defined, one for integer branches, and one for floating-point branches. The integer branch replaces both SPARC Bicc (branch integer condition code) and BPcc (branch integer condition code with prediction). The floating-point branch replaces SPARC FBfcc and FBPfcc. Other RISC ISAs are likely to have similar carrying capabilities.

This approach is the simplest to implement, but configuration data cannot be stored for branches that need a displacement larger than 12 bits. These will use the original branch

instruction and the default configuration. In Section 6.7, we show that such branches are rare and do not impact overall prediction accuracy even in benchmarks where they do occur.

4.4 Build PAA and Dynamic PAA

As explained earlier PAA can be performed at build time or dynamically.

Build PAA is less convenient for general purpose processors. This is because the PAA system needs to know the exact implementation, so that it knows the characteristics of the Spotlight (or other) branch predictor and so that it knows what kind of carry instructions are available. This condition exists for embedded systems, where code is compiled just for that system. Since the exact implementation is known anyway, build PAA is not a problem for embedded systems. In contrast, when developing for general purpose processors, the exact implementation is not known. One might compile for an ISA variation that 99.9% of the customers have, so that they can all run the code, but optimize for a newer implementation that only 50% have, so that those customers who need it will see the best performance. Because customers will have different implementations Build PAA cannot be used unless one shipped multiple builds, one for each implementation. That is feasible for smaller codes, like math or graphics libraries, but would be too large for a major application.

Development Dynamic PAA is more suitable for general purpose processors and less suitable for embedded systems since fewer of them would be able to host the tracing code. This is because embedded systems have less memory, might not have disk storage, and because even occasional tracing might interfere with real-time requirements (these are requirements that must be completed by a deadline). Despite these issues, there are several reasons to perform PAA dynamically. In dynamic PAA traces are collected using a lightly intrusive tracing system, and analysis is performed by a program independent of the compiler. Development dynamic PAA is used by developers after the build, but before the system is deployed. We expect that this will be the most common type of PAA, since it is more flexible than build-PAA and because live dynamic PAA is only appropriate for certain larger systems.

Live Dynamic PAA would be suitable for general purpose processors but is the least suitable for embedded systems. This is because it might change the behavior of running code (for example, a change in branch prediction accuracy would change how fast the code executes), which could compromise real-time requirements. Live dynamic PAA is a post-deployment technique where steps would be performed periodically within the system itself, providing annotated binaries that are adapted to the data to which the system is actually exposed. This is appropriate only for systems with re-writable storage and that are large enough to host the analysis code. This form of Live Dynamic PAA can tune execution for the case where, say, input data varies based on where a system is installed (the annotated binary would adapt and remain unchanged within a particular system), or the case where input data changes slowly over time (perhaps over hours), slowly enough so that the PAA could produce an annotated binary for the input data. Because of the way the binary is annotated, it would be possible to replace the text (machine instruction) portion of an executing image with one that has been annotated differently, so long as individual instructions change atomically.

Spotlight can adapt even if data varies more quickly. Live dynamic PAA would identify phases of execution, and PAA runs would be performed on distinct phases, producing phase-specific binaries. When a phase used to prepare a binary re-occurs, the binary would be used, avoiding the significant delay needed for the PAA process. It is not necessary that a phase be particularly long. As described above the dynamic profiling system samples execution, and samples can be collected over many different occurrences of a phase, until enough data is collected.

Researchers have investigated many techniques for detecting phases of execution [15, 21, 30, 31]. For Spotlight, phase boundaries need to be based on changes in input data character that affect branch bias and correlation. At one extreme, hardware can provide a hash of branch outcomes and addresses, for periodic collection by a monitoring system. At the other extreme an application-specific method can be developed. Application programmers would insert new API calls that would identify to the OS phase related information.

Chapter 5
Modifying Profile Based Predictors For Low Latency

The previous chapter suggests that profiled data can be stored within program instructions called carry instructions. Jiménez and Lin [9] point out that prior research either assumes or suggests that the profiled data for a given branch instruction should be stored in free space available within that instruction [2, 17, 32, 33, 37]. However, as they further suggest, it is not possible to decode instructions in the same cycle as they are fetched in high clock frequency modern processors. Since the processor must predict a branch in the same clock cycle that it is fetched, a single cycle prediction cannot be made if we rely on obtaining information from the decoded branch instruction as is assumed in chapter 3 for all of the profile-based predictors. It is still possible to use such a predictor in a real implementation that involves an overriding prediction scheme as is done in the Alpha EV6 [26] processor. In this scheme, a simple primary branch predictor (such as Bimodal) will be used to supply an initial prediction that may then be overridden later by a more accurate predictor like Spotlight. However, such a scheme increases complexity and sacrifices accuracy. In this chapter, we discuss how Spotlight and other profile-based predictors such as EHB and Agree can be augmented to obtain lower latency predictions, including single cycle predictions.

5.1 Reducing the Latency of Profile Based Predictors with a Hardware Table for Profiled Data

In a modern processor, multiple cycles can elapse between when an instruction is fetched and when the processor starts decoding the instruction. Therefore, a branch predictor cannot rely on extracting data from a decoded branch instruction if it is to deliver a single cycle prediction for that branch. If we are to obtain profiled information in time, a profile based branch predictor will have to access the data from a hardware structure instead of the decoded branch. This concept is shown for Spotlight in Figure 5.1. When a branch first occurs in a

program (such as instruction 0_x1000 in the figure), profiled information will not be available for it and it will use the default configuration. At the same time, the branch will write its information into a hardware structure called the Branch Information Table (BIT) for future use. When the same branch eventually re-executes, the profiled data will be available from the table and can be supplied to the rest of the predictor after being read from the table. We refer to the version of the predictor shown in Figure 5.1 as *BIT-Direct*.

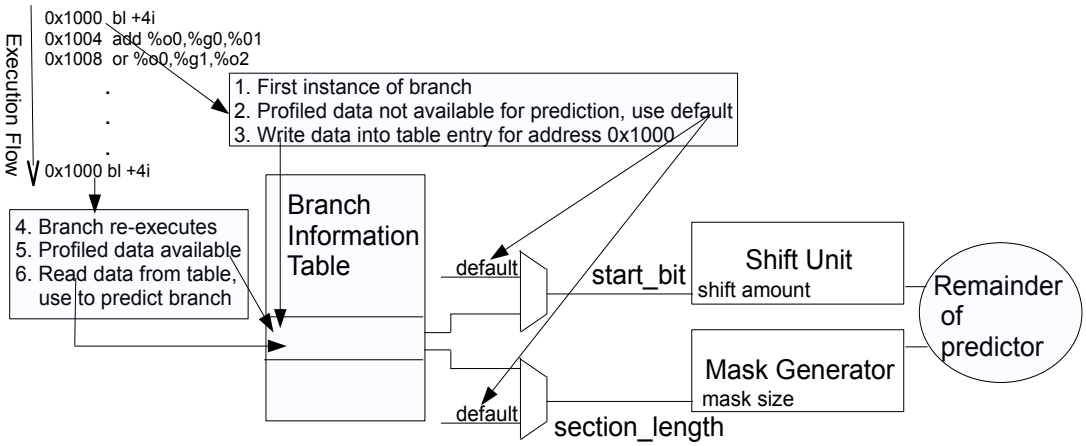

FIGURE 5.1: Spotlight BIT-Direct: Using a hardware structure to store profiled data

5.2 Attaining Single Cycle Latency by Storing Data Upstream

While using a BIT as shown in Figure 5.1 will provide a low latency implementation of Spotlight, there is no guarantee that the latency of the predictor will be just one cycle. This is because the predictor has to perform a table lookup in the the BIT, followed by some combinational logic before accessing the PHT to make a prediction. It is unlikely that all of this work could be performed in a single cycle. To achieve single cycle latency, we suggest pipelining the predictor. The concept of how pipelining is achieved is shown in Figure 5.2. The left side of the figure shows the program flow in terms of basic blocks. A basic block is a block of instructions where every instruction is guaranteed to execute each time the block executes (this means that a block contains a maximum of one branch). Consider a branch

that needs to be predicted in cycle c. The figure shows a block that is fetched in cycle $c-1$. In this cycle, there are two possibilities for what block (and for our purposes, the branch within that block) will be fetched in cycle c, depending on whether the block from cycle $c-1$ contains a branch that is taken or not-taken. If the prediction process starts in cycle $c-1$, we need to account for both possibilities since we do not know which block will eventually be fetched in cycle c. As such, the BIT entry for cycle $c-1$ stores data for both possible branches.

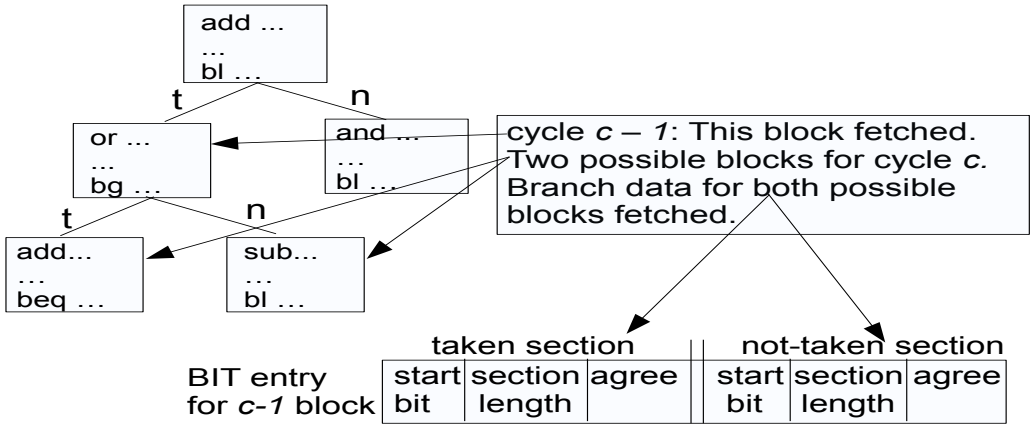

FIGURE 5.2: Spotlight BIT-Upstream: BIT use for single cycle prediction

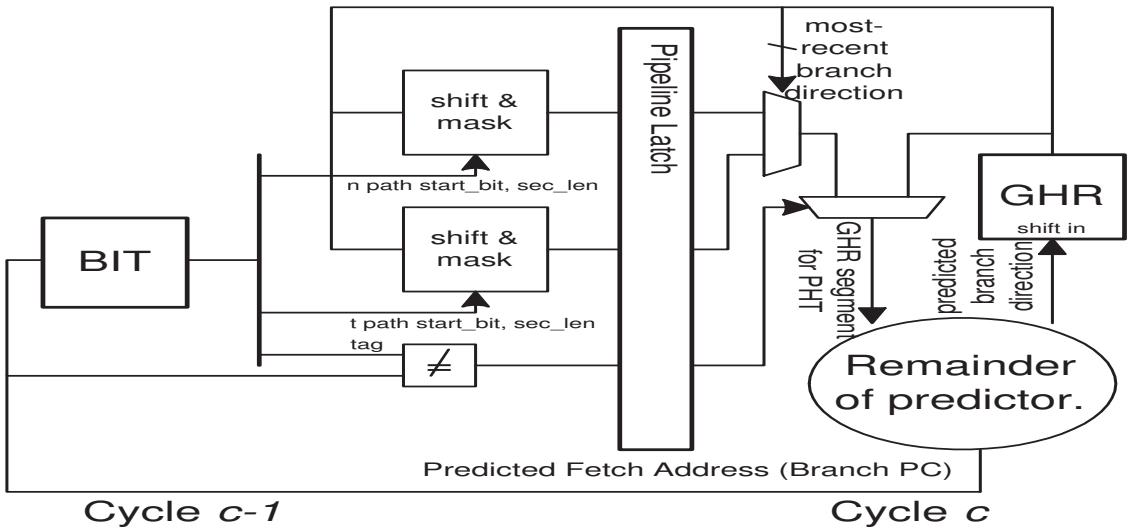

FIGURE 5.3: Spotlight BIT-Upstream: Hardware for single cycle prediction

The hardware to achieve this is showed in Figure 5.3 [38]. Profiled information is first obtained from a decoded instruction and then stored in the Branch Information Table (BIT). As described above, a branch in the BIT will not store profiled information for its own use but will instead store information about the next branch in the control flow of the program. A branch prediction is derived as follows. A branch that needs to be predicted in cycle c will use data retrieved from the BIT in cycle $c-1$ using the lower bits of the PC of the branch being predicted in cycle $c-1$. The information for both branches in the BIT entry is read and shift and mask operations are performed as described in Section 3.5 (with minor modifications) to two separate copies of the GHR. As can be seen in Figure 5.3, one GHR represents the taken path following the branch in cycle $c-1$ and the other represents the not-taken path. In order to reduce aliasing, each BIT entry is associated with a tag. If there is a tag miss, the default GHR is selected instead of the custom GHR for both paths. The default GHR has a starting_bit value of 0 and a Section_length that is either 0 (if the default predictor is bimodal) or $\log_2$ of the PHT size (if the default predictor is Gshare). The prediction for cycle $c-1$ is available at the beginning of cycle c and can hence be used to select between the two copies of the GHR available in this cycle. One of these copies is selected and used to index the PHT. If PHT access in cycle c is on the critical path, the PHT can be indexed earlier using both copies of the GHR and one of the two predictions obtained can be selected after the PHT access.

This scheme is described as *BIT-Upstream (SingleTag)* [38] because only one tag, which is based on the PC of the branch in cycle $c-1$, is stored in the BIT. An alternative would be to perform tag checking in cycle c using the PC of the branch in that cycle. This is likely to increase prediction accuracy but also increases the workload in cycle c, hence making it more difficult to achieve single cycle access. We call this second scheme *BIT-Upstream (DoubleTag)* [38] since the BIT would need to store two tags per entry, one for the taken path and one for the not-taken path from that entry.

Chapter 6
Evaluating Performance of Spotlight Schemes

6.1 Experimental Framework

6.1.1 Simulator Configuration

For all experiments in this dissertation, we use an extensively modified version of the RSIM [22] simulator. The simulator does detailed timing simulation of a dynamically scheduled superscalar processor and memory system. It implements a subset of the SPARC V9 ISA [41]. The system configuration details are shown in Table 6.1.

6.1.2 Benchmarks

The results reported in this paper are obtained using 11 benchmarks from the SPEC 2000 integer benchmark suite. These benchmarks are *gzip, swim, vpr, gcc, mcf, crafty, parser, perlbmk, gap, bzip2* and *twolf*. The benchmark *eon* is not simulated because it is not available for our simulator. These benchmarks are chosen because they reflect common programs run on general purpose processors and suffer from branch mispredictions.

6.1.3 Benchmark Sampling

We use the full training inputs provided by SPEC to perform profiling. Full training inputs can be used because profiling is done in functional simulation mode and is hence very fast. The data collection runs are done using SPEC reference inputs in the detailed timing simulation mode of RSIM. We use a tool based on the Simpoint multiple simulation point methodology [30] to generate a set of weighted samples that are selected over the entire duration of each benchmark. Each sample has 100 million instructions and a 10 million instruction warm up period is used prior to each sample.

6.2 Set of Experiments

We compare the accuracies of different versions of the Spotlight predictor. Specifically, we compare Spotlight-Direct, the version of the predictor described in chapter 3 and illustrated

TABLE 6.1: Simulator Configuration

Issue Width	6-way
ROB Size	256
Minimum Branch Mispred. Penalty	18 cycles
Branch Pred. Table Update Method	At Commit
L1 Cache	64 KiB data, 4-way set associative, 2 cycles latency
L2 Cache	4 MB data, 8-way set associative, 16 cycles latency
Main Memory Latency	200 cycles

in Figure 3.3 and the *BIT* versions of Spotlight described in chapter 5. For the Spotlight-BIT predictors, we measure the impact of the BIT size on accuracy. We compare the accuracy of Spotlight to other branch predictors such as Gshare, the profile-based EHB, as well as two highly accurate predictors with simple design, the YAGS predictor and a Gshare-Bimodal Hybrid predictor. We also evaluate the Hashed Perceptron and the Path-based Neural, two very high accuracy but implementation-complex predictors which compute the sum of several non-positive numbers that are looked up from several tables. The Hashed Perceptron does 5 table lookups and computes their sum to make a prediction, while the Path-based neural can do between 14 and 34 table lookups per prediction. Since Spotlight requires storing profiled data in instructions, we evaluate the amount of free opcode bits that are available in branch instructions of real programs to store this data. Finally, we analyze the performance of Spotlight to get an insight into why the predictor works well.

The experiments are done over a wide range of total predictor storage sizes. For the Spotlight-BIT configurations, the size of the BIT is included in the total size of the predictor. Each entry stores a 4 or 6 bit tag as well as the start bit and section length. For BIT-Upstream, an entry stores this information for two branches. We use a maximum GHR history length of 32 during the profiling step, i.e., Spotlight can select subsets of a 32 bit GHR. For the relevant competing predictors, we explore their design space to find the best configuration at a given size. Table 6.2 shows the main configuration settings used for the simulated predictors. The settings for Spotlight Direct and EHB are not shown because they

TABLE 6.2: Predictor Configurations

Size	Spotlight BIT		Hybrid		Hashed Perceptron			Path Neural	
	BIT Lg	PHT Lg	PHT Lg	Bmdl Lg	Wt.T Lg	Num.T's	Hist. Ln	Wt.T Lg	Hist. Ln
2 kB	10	12	12	11	10	5	28	7	20
6kB	10	14	14	12	11	5	44	8	26
10kB	10	15	15	12	12	5	48	9	28
40kB	11	17	17	14	14	5	56	10	34

consist of just one table, the PHT. As such, the Lg size of the PHT is straightforward to determine for a given target size.

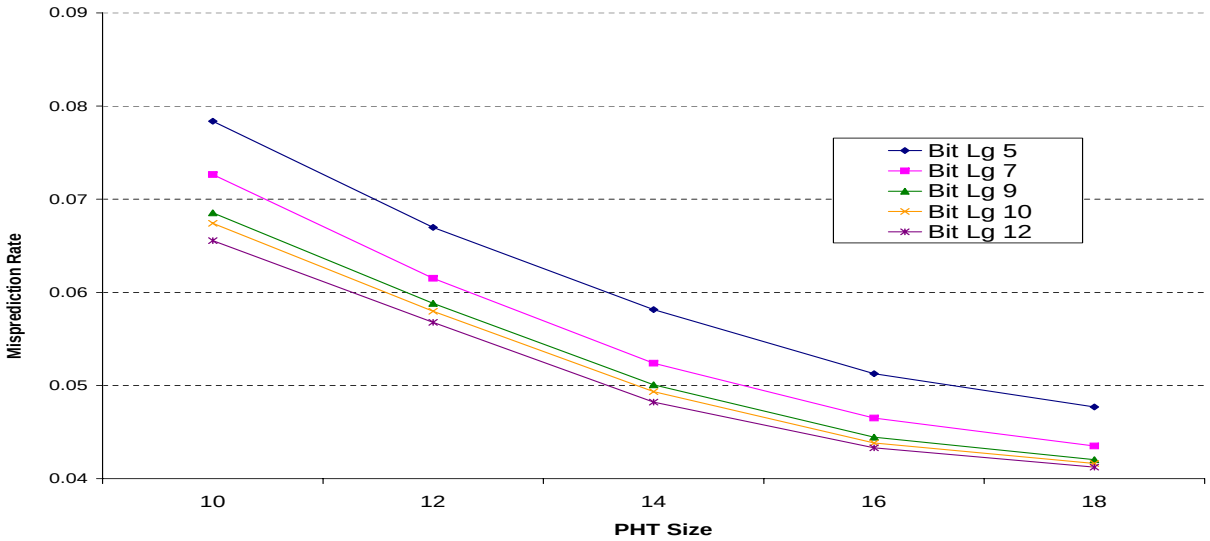

FIGURE 6.1: Average misprediction for different BIT Sizes

6.3 Impact of BIT Size on Spotlight's Accuracy

To show the impact of BIT size on the Spotlight predictor, Figures 6.1 - 6.4 show the impact of varying the size of the BIT on misprediction rates for the Spotlight BIT-Direct scheme.

Figure 6.1 shows that as the size of the BIT increases, the average misprediction rate consistently decreases as expected. This is the case across all PHT sizes, although the improvement is smaller with larger PHT sizes. The difference between the smallest and largest BIT size is significant, suggesting that larger BIT's are needed to achieve the best possible performance. Figures 6.2 - 6.4 show that the impact varies significantly across benchmarks. For five benchmarks (gzip, vpr, mcf, gap, bzip2), the impact of increasing BIT is minimal on misprediction rate. However, benchmarks like perlbmk, parser and crafty are more signif-

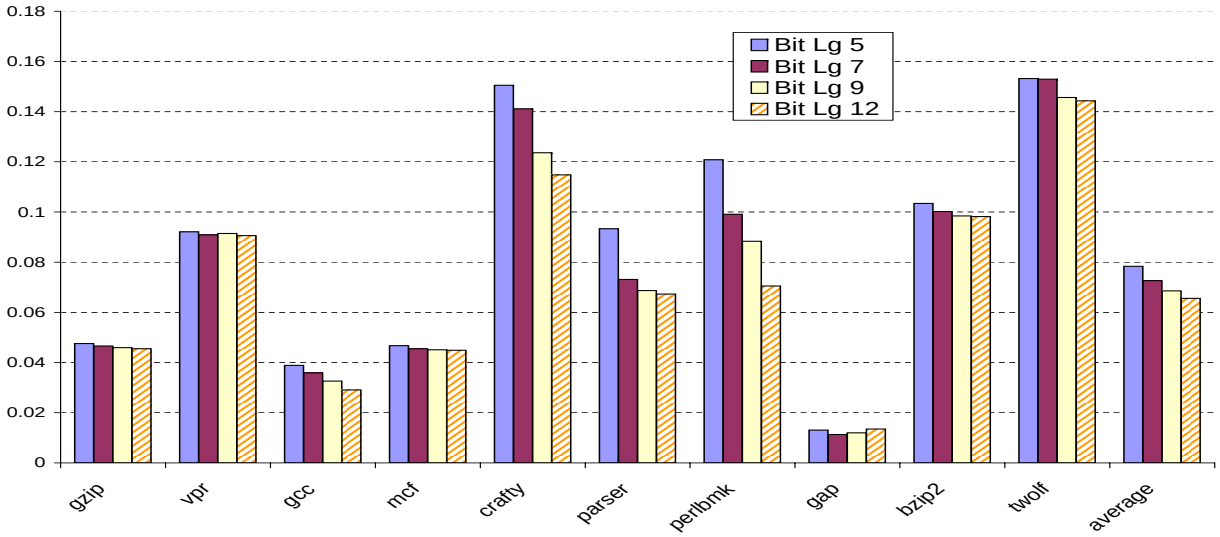

FIGURE 6.2: Per benchmark misprediction for PHT Size 10

icantly impacted by increasing the size of the BIT. For perlbmk, every increase in BIT size results in a proportional decrease in misprediction rate. Generally, programs with large kernels will be impacted more significantly while the impact on programs with smaller kernels is likely to be minimal. For the larger PHT sizes (Figures 6.3 - 6.4), the impact of increasing BIT size is not significantly different from that for the smallest size.

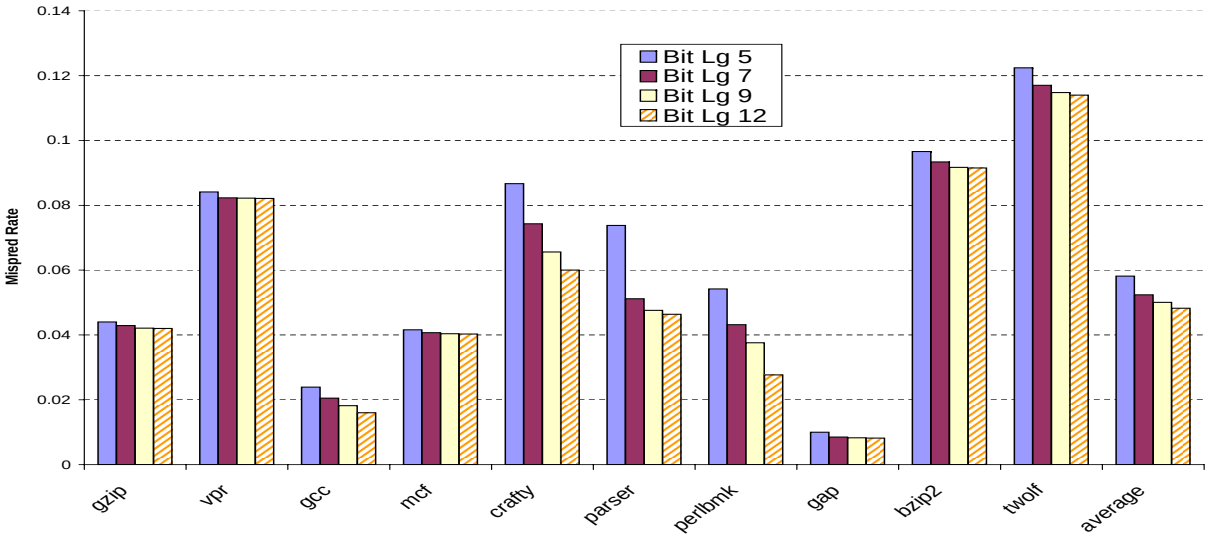

FIGURE 6.3: Per benchmark misprediction for PHT Size 14

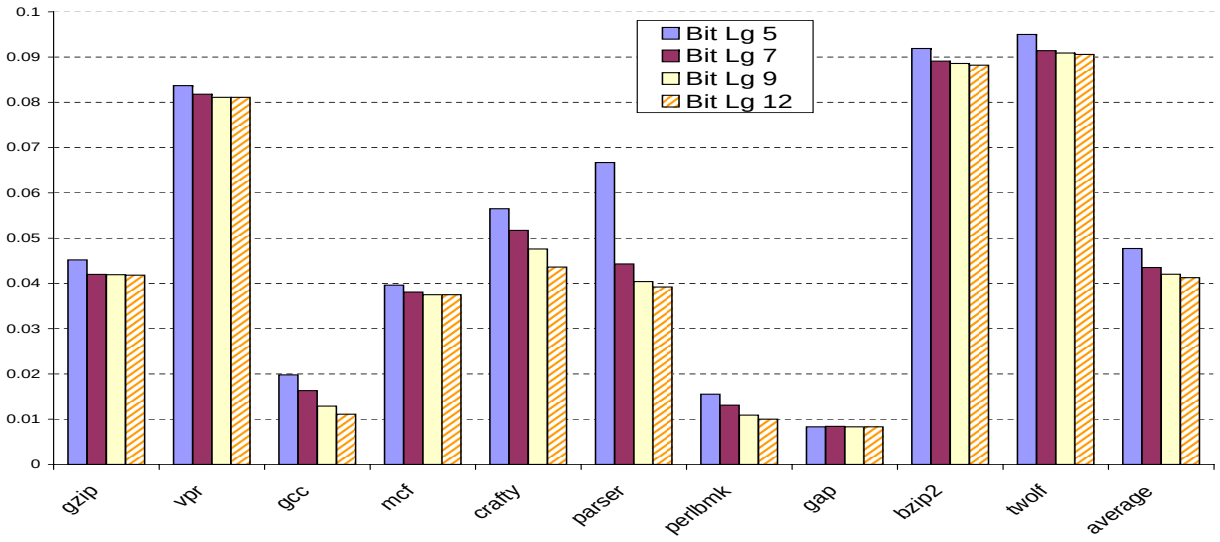

FIGURE 6.4: Per benchmark misprediction for PHT Size 18

6.4 Accuracy of Various Spotlight Predictors

This Section compares the accuracy of Spotlight-Direct (Figure 3.3) with the BIT based configurations described in chapter 5. The Spotlight-Direct predictor assumes that profiled data is always available in a timely manner after being decoded from a branch instruction while the BIT based predictors store profiled data in a hardware structure called the BIT. Since Spotlight uses the Agree scheme for its PHT update, we also evaluate a version of Spotlight called Spotlight-Direct (No Agree) to evaluate the contribution of the Agree scheme to Spotlight's performance benefits. The PHT in this predictor is updated using the traditional Gshare method. The misprediction rates for these predictors are shown in Figure 6.5. For this figure (and Figures 6.6 - 6.9), we simulate all PHT sizes in the range Lg $10 - 18$. The BIT configuration plots are shifted to the right because the size of the BIT is included. Spotlight-Direct is smallest since it does not use a BIT, and BIT-Direct is smaller than BIT-Upstream because it stores data for only one branch per entry. To compare the misprediction rate of two predictors, we have to compare the corresponding points. We can see from Figure 6.5 that the two Spotlight BIT-Upstream configurations are almost identical in accuracy to Spotlight BIT-Direct for most sizes. This shows that pipelining the predictor resulted in no loss in accuracy on average. We also see that the performance difference be-

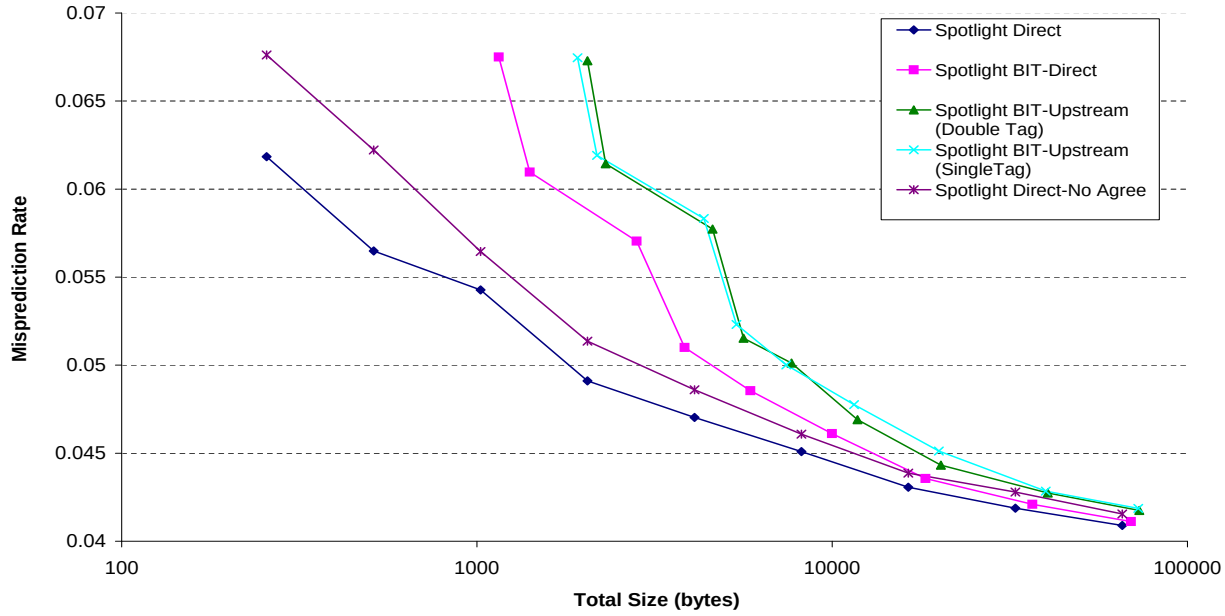

FIGURE 6.5: Various Spotlight Configurations

tween the SingleTag and DoubleTag schemes is minor. The performance drop compared to Spotlight-Direct is larger but becomes less significant starting at a PHT Size Lg of 12. As the PHT size increases, the BIT becomes an increasingly smaller component (see Table 6.2 for specific BIT sizes) of the overall size and the points in the BIT configuration plots start approaching the Spotlight Direct plot. These results indicate that single cycle prediction can be achieved without sacrificing much accuracy, but the cost of the BIT is significant at smaller PHT sizes. As one might expect, the gap between Spotlight-Direct and Spotlight-Direct (No Agree) declines as the PHT size increases. The difference is more significant at smaller sizes but becomes minor starting at a size of 1 kB. This indicates that while Spotlight benefits due to Agree, the benefit becomes smaller with increasing PHT size and is not responsible for most of Spotlight's performance benefits. e

6.5 Misprediction Rates

Figures 6.6 - 6.9 show the average and per benchmark misprediction rates for the various predictors over the 11 SPEC integer benchmarks.

From Figure 6.6, it can be seen that Spotlight-Direct outperforms Gshare, EHB and Hybrid for all sizes simulated. This predictor achieves an average reduction in misprediction rate of

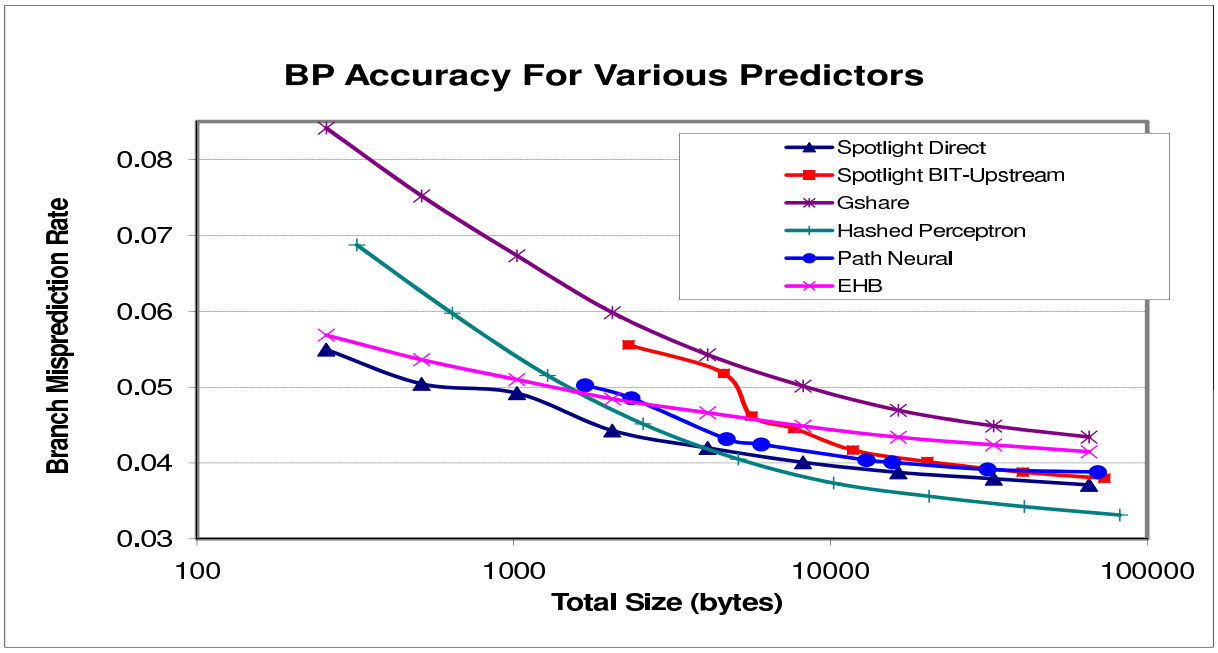

FIGURE 6.6: Average misprediction rates for various branch predictors

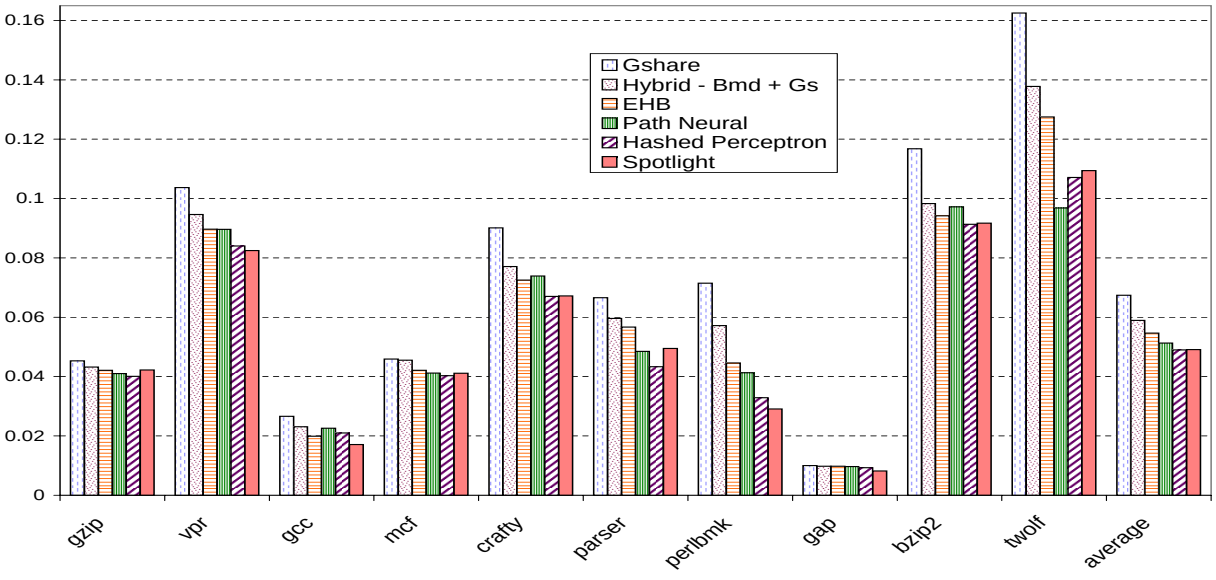

FIGURE 6.7: Per benchmark misprediction rates for size 2KB

about 19% over Gshare, 9% over EHB and 13% over Hybrid at a size of 8 kB, with smaller budgets providing more significant improvements for Spotlight. We also simulated the YAGS branch predictor, but the data is not shown here because its performance is very close to the Hybrid predictor. Even with the size of the BIT factored in, the Spotlight BIT-Upstream predictor is able to outperform Gshare at all sizes and the Gshare-Bimodal Hybrid branch predictors at a size of 5KB. It starts outperforming EHB Direct at a size of 11 kB, even though comparing the two predictors is unreasonable since the version of EHB simulated here is similar to Spotlight Direct in that it ideally assumes that profiled data is always available at no cost in a single cycle. Spotlight-Direct outperforms the Path-based Neural predictor, which is remarkable considering the far greater complexity of that predictor (it uses between 14 and 34 tables). It stops outperforming the Hashed Perceptron predictor starting at a size of 10 kB but like the Path-based Neural, this predictor is more complex in its design and is also one of the most accurate predictors that have been proposed in the literature. The fact that Spotlight-Direct is very competitive with the Hashed Perceptron at all sizes suggests that it could be used in a system with an overriding prediction mechanism similar to the Alpha EV6 [26]. In such a system, a first-level predictor (such as a small bimodal predictor) would need to make a primary prediction that may be overridden by a Spotlight predictor which waits for profiled information from the decoded branch instruction. We must note that the version of the Hashed Perceptron and Path-based Neural that we simulated are ideal versions that assume that all computations can be done in the same cycle (we assume that all table lookups and the sum of non-positive numbers are done in that cycle).

Per benchmark data is shown in Figures 6.7 - 6.9 at approximate sizes of 2 kB, 8 kB and 32 kB. Spotlight Direct works well at sizes smaller than 2 kB, but since the minimum size for the Path Based Neural is almost 2 kB, this is the lowest size shown. We can see that the performance benefit provided by Spotlight occurs over a broad range of benchmarks. It outperforms Gshare, Hybrid and EHB for almost all benchmarks and sizes and outperforms

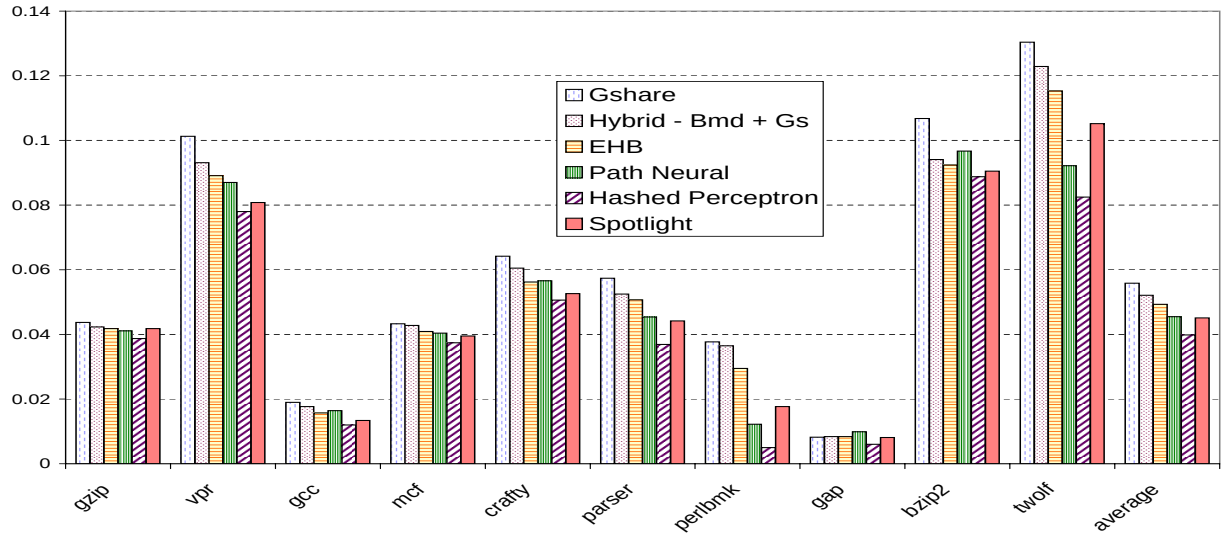

FIGURE 6.8: Per benchmark misprediction rate for size 8KB

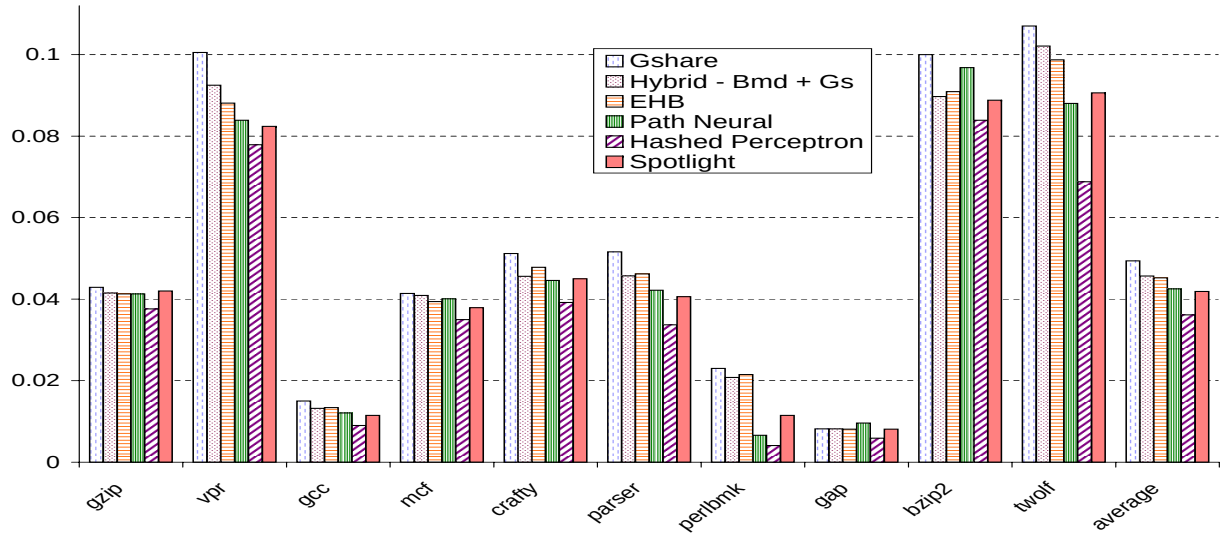

FIGURE 6.9: Per benchmark misprediction rate for size 32KB

the Path Based Neural predictor in most benchmarks for all sizes. As observed in Figure 6.8 and 6.9, the Hashed Perceptron outperforms Spotlight at sizes of 8 kB and 32 kB. The figures however slightly exaggerate its improvement over Spotlight because the sizes used here are the closest to our target sizes and not exact. For example, the closest configuration to 32 kB for Hashed Perceptron is actually 40 kB. Similarly, some of the Path Neural configurations are slightly below the target sizes.

6.6 Practicality of the Various Spotlight Predictors

As mentioned above, the Spotlight Direct predictor can be used in an overriding prediction scheme where a primary predictor provides an initial prediction. Because this scheme needs to extract data from a decoded instruction, it will take a few cycles to provide a prediction. A branch that is mispredicted by the primary predictor but is correctly predicted by Spotlight will hence incur a small misprediction penalty. The Spotlight BIT-Direct predictor does not have to wait for the decoded instruction, but since it requires two table lookups and some combinational logic, it will take more than one cycle to deliver a prediction. It could also be used in an overriding prediction scheme, but the latency of the predictor will be 2 cycles, much lower than that for Spotlight Direct. The Spotlight BIT-Upstream predictor is capable of providing single cycle predictions and hence does not need to be in a system with a primary predictor. However, the size of the BIT needed to maximize accuracy is a significant component of the overall predictor size at smaller PHT sizes (this is also true to a lesser extent for Spotlight BIT-Direct). As such, this scheme is more likely to be chosen by designers if the overall hardware budget for branch prediction is greater than 10 kB. For embedded systems, a large BIT is clearly not a practical option because of size concerns. These processors could use the BIT predictors with a smaller BIT (perhaps with 32 or 64 entries) by trading some accuracy for size and practicality. As we saw in Figures 6.2 - 6.4, the amount of accuracy lost will depend on the benchmark. Benchmarks with larger kernels will be more significantly impacted while the impact of benchmarks with smaller kernels is

likely to be minimal. We must note that even in benchmarks that are more significantly impacted by BIT size (such as perlbmk), the Spotlight configurations at the smallest BIT size are still more accurate than the corresponding Gshare configuration.

6.7 Evaluating Limited Branch Carry Space

As discussed in chapter 4, one way to store profiled information is in modified branch instructions using a technique called opcode borrowing. We suggest that the simplest way to handle branches whose displacement is too large is to prevent them from using a profiled configuration (they will use the default configuration). While this technique is restrictive and is not the only option, we show here that it does not impact misprediction rates.

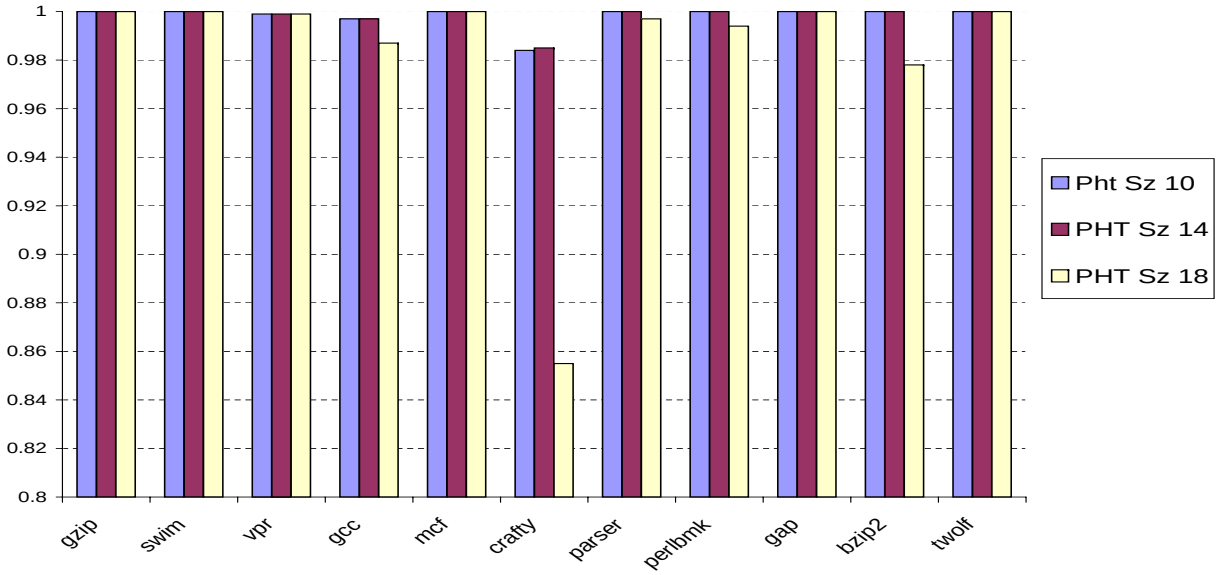

FIGURE 6.10: Fraction of branches with sufficiently small offsets

Recall that Spotlight needs a start bit, section length and a biasing bit. The start bit representation for the Spotlight predictor evaluated in this chapter requires 5 bits ($\log_2$ of 32) and the biasing bit is represented by a single bit. The section length might either require 4 or 5 bits for representation, depending on the PHT Size of the target system (the number of bits will be $\log_2$ of the section length). Therefore, the total number of bits required for representing a Spotlight configuration will either 10 or 11 bits. Figure 6.10 shows the ratio of dynamic branches that have adequately small displacements and can use the modified

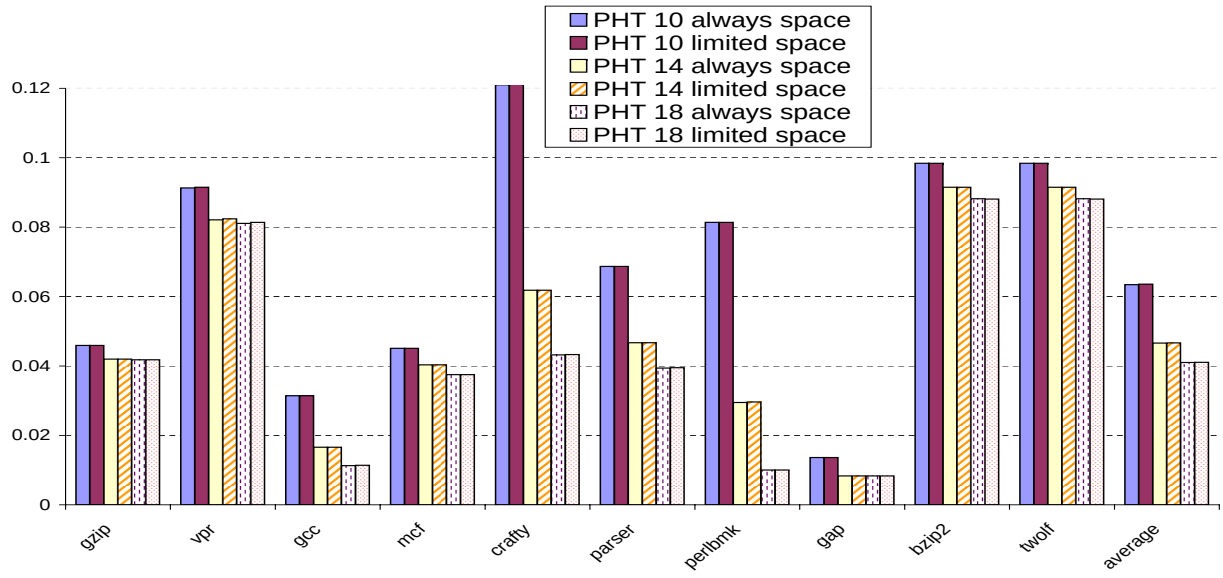

FIGURE 6.11: Impact of limited carry space on misprediction rates

branch instructions. We can see that for most benchmarks, almost all branches have small enough displacements (indicated by a ratio of 1). In benchmarks like perlbmk and bzip2 and crafty, there is a drop off for the largest PHT size of 18. This is most likely because a PHT size of 18 requires an extra bit of storage compared to the other sizes because of the need to encode any number between 0 and 18. The lowest ratio is for crafty for PHT Size Lg 14, where 85.5% of branches can use the smaller displacement instruction. Figure 6.11 shows the performance of Spotlight BIT-Direct considering this limitation. In the limited space configuration mentioned in this figure, branches that have displacements that are too large will use the original branch instruction and the default predictor configuration. The results demonstrate that limiting the amount of space available has virtually no impact on the misprediction rate for any of the benchmarks. While it is difficult to see the exact numbers in the figure, there is no increase in misprediction rate for most benchmarks and sizes. In cases where there is an impact, it is no greater than 0.03% in almost all of those cases. The only exception is crafty at PHT Size Lg 10, which sees a still minor drop of 0.14%.

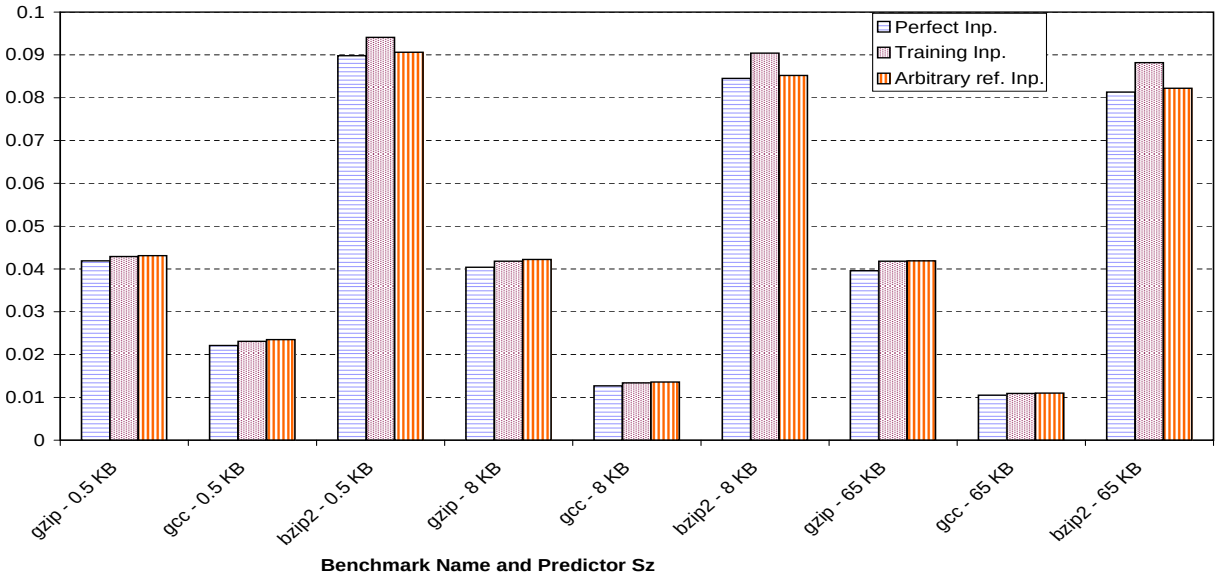

FIGURE 6.12: Impact of training input data on Spotlight

6.8 Impact of Training Input Data on Spotlight's Accuracy

As mentioned above, we saw in Figures 6.7 - 6.9 that Spotlight does well over a broad range of benchmarks. This result may seem surprising at first given that Spotlight relies on profiled information and either an inaccurate poorly designed training input or varying input data could impact its accuracy. It is beyond the scope of this paper to show that profiling is a robust and accurate technique. However, we note that profiling is used successfully by nearly all general-purpose compilers, such as gcc, and the evaluation and generation of accurate profiles has been widely studied in literature. For example, Wall [40] notes in early work that real profiles generated from different runs are very effective and are often nearly as good as perfect profiles. Hsu *et al.* [43] claim that profile data from training inputs can be used to reliably predict branch directions. We expect Spotlight to be even less sensitive to training data since it relies on using a set of correlated branches and does not depend on the actual branch outcomes or input data to make predictions.

To evaluate how sensitive Spotlight is to training data, we simulate Spotlight using three different sets of training inputs for a few benchmarks. Each benchmark is evaluated using a perfect input, the training input and an arbitrary reference input. For the arbitrary reference

input, an arbitrarily selected input is used for training and a different input is used for the reference run (for example, we train with gcc-ref-integrate and evaluate with gcc-ref-expr). For the perfect input case, the same input that is used for training is also used for the reference run. The results are shown in Figure 6.12 for three different predictor sizes. We would expect the perfect input to be most accurate, followed by the training input and the arbitrary-ref input. The accuracy of Spotlight does not vary significantly for gcc and gzip across the three training inputs (the biggest difference in accuracy is about 0.15% for gcc). The results for bzip2 are interesting. The training input under-performs even the arbitrary reference input while the arbitrary reference input is almost as accurate as the perfect input. This would seem to indicate that the training input for bzip2 is poorly designed. Fang *et al.* [5] observe that the SPEC 2000 bzip2 training input does not touch a lot of the code that reference inputs touch and this may be a factor. However, despite this limitation, Spotlight does very well for bzip2 compared to other branch predictors (see Figures 6.7 - 6.9). This result suggests that Spotlight can be an effective branch predictor even for benchmarks that do not have very accurate training data.

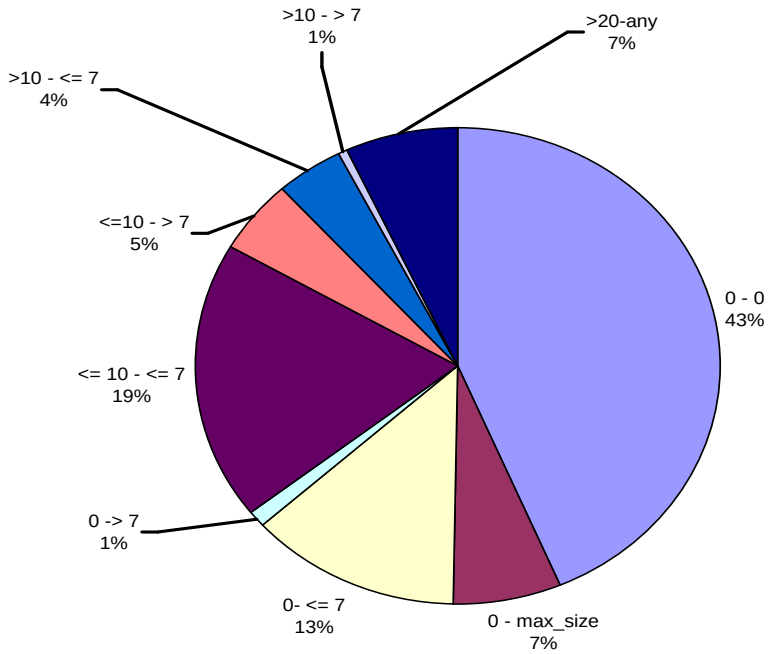

FIGURE 6.13: Spotlight configurations used for PHT Sz 10

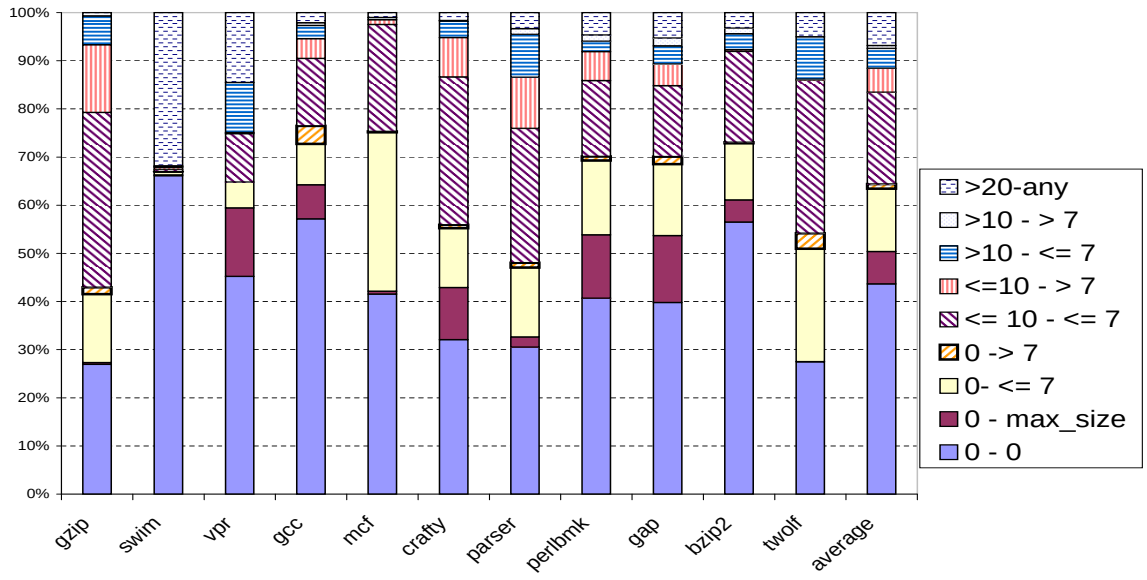

FIGURE 6.14: Configs used per benchmark for PHT Sz 10

6.9 Explaining the Performance of Spotlight

We can gain an understanding of why Spotlight does well by looking at Figures 6.13 - 6.16. The data shows how frequently various Sections of the GHR are used for the various benchmarks for PHT sizes of Lg 10 and Lg 14. The pie charts demonstrate the average use for each configuration.

In the key listed on the right hand side, the first value describes the starting bit of GHR segments and the second value describes the section length. For example, the segments that comprise the ≤ 10 - ≤ 7 bar are those GHR configurations that have starting bit less than 10 and section length greater than 7. The segment $0 - 0$ is one that uses no GHR bits and is essentially a bimodal configuration (the difference between the two is PHT collisions in Spotlight). The segment $0 - max\_size$ is the configuration that would be used for all predictions in a Gshare predictor and represents the default configuration. We define short segments as those with section length ≤ 7.

The pie charts show a lot of diversity in the Spotlight configuration used. The most common configuration used is the $0 - 0$ bimodal configuration and this configuration along with the default GHR configuration of $0 - max\_size$ is used in about 50% of all dynamic branch predictions for both PHT sizes. These results suggest that Spotlight partly works

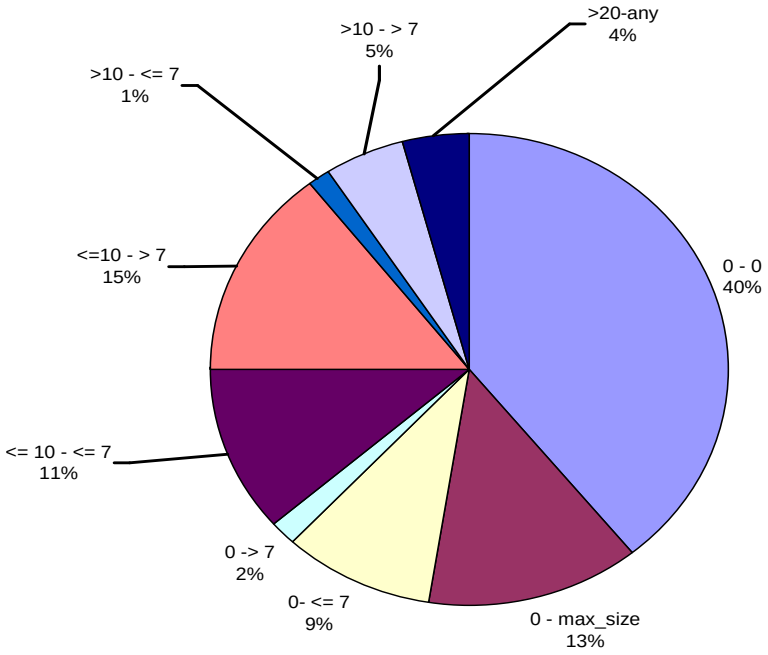

FIGURE 6.15: Spotlight configurations used for PHT Sz 14

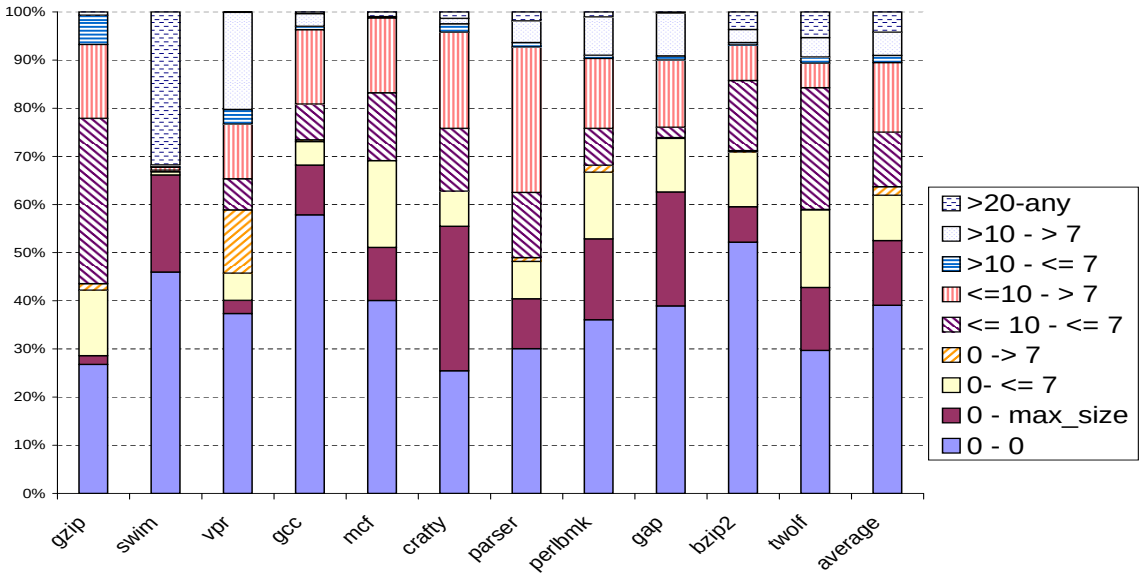

FIGURE 6.16: Configs used per benchmark for PHT Sz 14

like the Gshare-Bimodal Hybrid predictor even though unlike the actual Hybrid predictor, it does not have separate bimodal and chooser tables. We also see that the remaining 50% comprises a very diverse set of GHR segments. They include significant amounts of short segments (section length ≤ 7) starting at 0, long segments starting at 0, short segments starting at other values and so on. Short GHR segments are very common, for a PHT size of 10 they represent 36% of all dynamic prediction and for a PHT size of 14, they represent 21% of all dynamic predictions. Segments that start further from the current branch (non-zero starting bit) add up to approximately 37% of all dynamic predictions.

The per benchmark data in Figures 6.14 and 6.16 demonstrate a diverse use of GHR segments. For example, in Figure 6.16, gzip uses a significant ratio of ≤ 10 - ≤ 7 segments while this segment is rarely used in gap. Swim uses a high percentage of > 20 segments in both figures, while most other benchmarks use them rarely. After the bimodal configuration, the ≤ 10 - ≤ 7 configuration is the most commonly used.

Chapter 7

Preloading - A Technique for Delivering Profiled Data to Hardware Without Using Large Tables

Consider the situation where, in some pre-processing step, some *statically chosen* data has been determined for each static instruction in some subset of program instructions, an instruction in the subset is called a *benefiting instruction*. This data is needed by the front end at a time in which the address of the benefiting instruction is available, and so it might be retrieved from some PC-indexed *Benefiting Instruction Table* (BIT). Suppose further that this custom data can be *carried* by program instructions, not necessarily the benefiting instruction, and *preloaded* into the table soon after the carry instruction is decoded. A *placement* is an assignment of statically chosen data to carry instructions. The *placement problem*, the subject of this chapter, is finding a placement that realizes high BIT hit ratio, but more importantly, high performance, using a very small BIT.

The preloading technique described in general above can be specifically applied to the Spotlight branch predictor. Let us define the ideal case as the situation where data for the benefiting instructions, BIs (in Spotlight's case, the benefiting instructions are the branch instructions that need non-default GHR segments), is always available in a timely manner to the hardware. Although we only use profiled data in this dissertation, the data delivered to the hardware may be derived by other means. Hence the term *statically chosen* will be used instead of *profiled* for generality purposes. As mentioned previously, statically chosen data cannot be extracted directly from the BI if it is to become available on time for the BI's use at the front end of the processor. In general, a structure such as a Benefiting Instruction Table (BIT, like the Branch Information Table described in chapter 5) can be used to store the statically chosen data (after extracting it from either the BI or an instruction in the prior control block) during the first instance of a BI, and this table (as opposed to the instruction itself) can be used to obtain statically chosen data for subsequent executions of

the BI. A specific application of this BIT based technique was described in chapter 5 and evaluated in chapter 6 for the Spotlight branch predictor. The results in chapter 6 show that there are two potential drawbacks to this approach. Firstly, since the BIT is required to store data for a large number of BIs, the number of entries needed and hence the overall size of the BIT can be very high. This is contradictory to our goal of designing low-cost predictors. Secondly, even a BIT of reasonable size will suffer a loss in accuracy compared to the ideal case because of BIT misses during the first instance of the BI and during instances when the entry of a BI is overwritten by other BIs (i.e., aliasing occurs). This issue may be an even bigger factor in real world applications than in the SPECcpu benchmarks shown in the previous Section. This is because SPECcpu programs have been criticized for having very small kernels compared to real world applications, and the data for these smaller kernels can fit easily within smaller predictor tables.

We will show that both of these problems can be mitigated by taking a different approach. In the previous chapters, we assume that when a BIT entry is written, the entry will rarely be overwritten. This makes it necessary to maintain a large BIT. But what if we maintain a BIT with the expectation that a written entry will be used within a short period of time by the BI? Clearly, this could drastically reduce the size of the table needed. This can be achieved by a techniques that we call *preloading*. The basic idea is to place the statically chosen data (which we will refer to as preload data) for a BI several control blocks back in the program control flow. When this previous block or instruction is reached, the process of extracting the preload data from the instruction and placing it in the BIT begins. We define the location that carries the data for a BI as a preload site, PLS. A PLS can either be a specific instruction in a previous block, or data may be spread over multiple instructions in that block. If the data is placed sufficiently early in the code, it will be preloaded into the BIT by the time the BI being predicted is fetched. However, since there can be many paths to a branch, the data for an individual BI will typically have to be preloaded into multiple PLSs if it is to be available for most dynamic predictions of the BI. Therefore,

it is obviously impossible to preload for all or even most typical instances of the BI. The minimum distance from a BI where its preload data can be placed in a PLS in order to ensure that it is available in the BIT on time is defined as the minimum preload distance, pls_min. Depending on how the front end groups instructions, the distance may either be measured in terms of instructions or basic blocks (we use blocks in this dissertation). We will demonstrate that this problem of distributing the data for the BI's amongst the limited number of PLS's in the most optimal manner, the *Placement Problem*, is a global optimization problem. The next Section illustrates the placement problem and preloading for a branch predictor. This is followed by a formulation of the placement problem and an explanation of what is required to solve the problem.

7.1 Illustrating The Placement Problem for a Branch Predictor

In Figure 7.1, each node represents a control block which might contain a branch instruction and the arrows represent the program data flow. Block A1 can take one of two possible paths either to block B1 or B2. Block B1 may lead to either block C1 or C2 and so on. Consider the case where we want to predict a branch in D1. The preload data for D1 may be placed in a block that is on the path to this block (C1, B1 or A1). If the data is placed sufficiently ahead (the closest it can be placed is the minimum preload distance, pls_min), it would be possible to extract the data from the block or instruction and place it in the BIT in enough time for the branch in D1 to access it. With this approach, a branch only needs to access a BIT entry which was written a few cycles earlier, not necessarily one that was written during a previous instance of the branch or several hundred cycles earlier. Hence it would be possible to use a very small BIT and keep the overall size of the branch predictor small. Despite using a smaller BIT, this approach has the potential to deliver higher accuracy compared to the previously mentioned approach. This is because the loss in accuracy that occurs with the

previous approach during the first instance of a branch or due to aliasing in the BIT is no longer a major factor (some aliasing could still occur in the BIT).

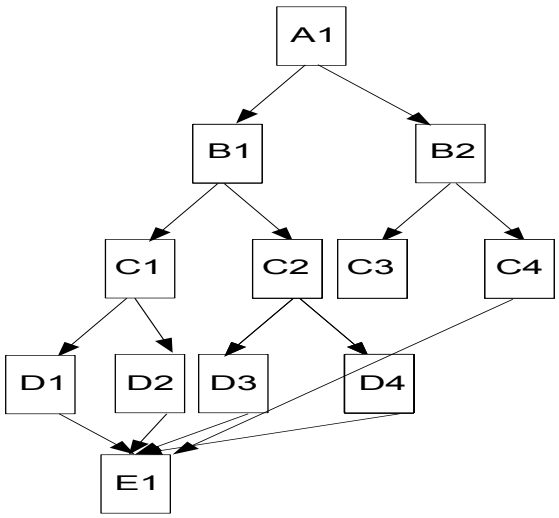

FIGURE 7.1: Control flow diagram indicating different paths to a branch

The challenge with this approach has to do with the placement of this preload data amongst the PLS's. Specifically, in which previous block or blocks do we place the preload data for a given branch (a BI as described previously). The issue is complicated by the fact that there may be many possible paths to a given branch. This implies that the data for a branch may have to be placed in more than one PLS. More importantly, it also implies that two or more branches may be competing for one available spot in a PLS, and the branches not chosen for that spot may suffer a loss in accuracy due to the unavailability of preload data. On the other hand, there will be many branches that are not BIs (because they do not benefit at all from preload data), and a subset of these BIs will only get a small benefit by using preload data. Therefore, the total number of BI's that need to be placed to achieve close to ideal performance will be fewer than the number of PLSs in the program.

Consider Figure 7.1 again. Let us assume that the minimum distance that preload data can be placed at, pls_min, is 2, i.e., the data must be placed at least two control blocks ahead in order to be extracted in time. Assuming that the figure represents the entire program, data for any branches in the C blocks may only be placed in block A1. If there is a branch in each

of the four C blocks, those blocks will all compete for the single spot at A1. Assuming that all these blocks benefit from using preload data, 3 out of 4 C blocks will suffer a performance loss compared to the ideal case. Similarly, the four D blocks compete with each other for the slot in B1 but may also compete with the four C blocks for the slot in A1. On the other hand, block E1 can be reached by taking multiple paths through the program. Since it can be reached from either C1, C2 or B2, it may be necessary to place its data in each of these three blocks.

Define the *preload horizon* for a dynamic execution of a branch to be a sequence of blocks from distance XXX to distance pls_min. A set of preload sites is said to *fully cover* a branch if at least one is in the preload horizon for every dynamic execution of the branch. In summary, there are three issues and cases to consider for a BI branch.

1. There exists a PLS or set of PLSs, any of which fully covers the branch. This is the most efficient case in terms of the amount of data that needs to be preloaded, since placing the data in any of these PLS('s) ensures that the branch will always have preload data available.

2. None of the PLS's fully cover the branch, i.e, the minimum set of preload sites needed to fully cover a branch has more than one element. This necessitates placing the data in multiple PLSs, and is hence wasteful in terms of the amount of data that needs to be preloaded to get good performance.

3. In general, while there are usually many PLSs where a branch's data can be placed, many of these PLSs will see competition for their placement slot from multiple branches.

The number of placement sites needed to cover a branch can vary widely. Suppose data is to be placed d blocks back, and assume all blocks end with branches. In the worst case 2^d placement sites are needed to cover the branch. However, because of code reconvergence, the average number is much smaller.

Hence, the efficient placement of preload data is a significant challenge if low-cost profile-based predictors are to be built. For each branch, we must decide at which preload sites its data needs to be placed at and how this can be done in an intelligent manner so that overall prediction accuracy is close to the ideal case. This requires choosing among multiple branches that are competing for a single slot in a PLS and determining at how many PLSs we should place data when there are many unique paths to a branch, as described in case *2* above.

7.2 Defining the Placement Problem

In addition to the terms defined earlier, we define some other key terms.

A *static instruction*, s, is a single instruction in a program, with each static instruction having its own address (program counter). An execution of a program can have multiple dynamic instances of the same static instruction.

A *static program*, S, is a set of instructions (for brevity, instruction will be used for a static instruction), the instructions will be denoted s_j, $0 \leq j < N$.

An *execution*, e, is a sequence of *static instructions*, $(\epsilon_0, \epsilon_1, \ldots, \epsilon_{T-1})$, $\epsilon_j \in S$ for $0 \leq j < T$.

A *path*, e, is a subsequence of E. Call path e an *entry path* if it is a prefix of E. Path e *leads to* instruction i if the last element of e is i.

Let B denote the set of possible preload sites or PLS's in a program, and let I denote the set of statically chosen BI's in the program for which preloading is required.

An *assignment* is a two-tuple (b, i) indicating that placement site $b \in B$ carries preload data for an instruction $i \in I$. Assignment (b, ϕ) indicates that placement site b is not occupied.

A *placement*, denoted $\mathcal{P}$, is a mapping $B \rightarrow I \cup \{\phi\}$. For notational convenience it will be represented by the set of elements $P = \{(b, \mathcal{P}(b)) | b \in I\}$. That is $(b, i) \in P$ indicates that placement site b carries data for instruction i and $(b, \phi) \in P$ indicates that placement site b does not carry data.

A real-valued performance measure will be associated with each placement. For our purposes, it is chosen to be a good estimate of branch predictor performance as described in the following equation.

$$\text{Perf(P)} = \sum_{i \in I} \sum_{e \in E(i)} \begin{cases} \text{ProbCorrPred}_{\text{hit}}(e, i), & if \text{ e/i;} \\ \text{ProbCorrPred}_{\text{miss}}(e, i), & otherwise, \end{cases}$$

where:

e is a path to a BI, $E(i)$ is the set of all paths to a BI i,

e/i indicates that the data for a BI i was found in the path e and not overwritten before the BIT was read,

$\text{ProbCorrPred}_{\text{hit}}(e, i)$ for a branch predictor, the probability of a correct prediction when a BI i is on path e and is able to access its preload data,

$\text{ProbCorrPred}_{\text{miss}}(e, i)$ for a branch predictor, tthe probability of a correct prediction when a BI i is not able to access its preload data because the data is not on the path e for two possible reasons.

i. All PLS's on the path are taken by other BI's and therefore unavailable for i.

ii. A PLS is available for i but the BIT entry needed by the PLS chosen for i is occupied by the PLS for another BI.

For preloading applications that do not involve branch predictors, the probability of a correct prediction will be replaced by another probability.

The equation above indicates that for each dynamic instance of a BI, two possible performances are possible. One is the performance expected if preload data for the BI is found on the path to the instruction (and is hence available for a prediction), and the other is the performance expected when preload data is not available for the instruction. Note that benefiting instructions, BIs, will be chosen such that it is expected that the probability (and hence the performance achieved) when preload data is used ($\text{ProbCorrPred}_{\text{hit}}$) will be greater than the probability when preload data is not used.

7.3 Finding a Good Placement

Before explaining why the placement problem is a global optimization problem, we discuss the basic approach to solving the placement problem in this Section. The solution involves finding a set of placements $P_0, P_1, \ldots, P_j$. The initial placement P_0 can either be empty or be initalized to some configuration. At each iteration j, the task is to find a placement P_j starting with placement P_{j-1} .

Let P_0 be an arbitrary placement, possibly empty, $P = \{(b, \phi) | b \in B\}$, or possibly in an *autocarry* state, $P = \{(b, b) | b \in I\}$ (assuming $I \subseteq B$).

1. At iteration j, choose some $b \in B$ and $i_2 \in I$, and let placement candidate $Q = P_{j-1} - (b, i_1) + (b, i_2)$, where $(b, i_1) \in P_{j-1}$. That is, replace (b, i_1) with (b, i_2).

2. Evaluate the performance of Q and compare to $Perf(P_{j-1})$.

3. Based on the difference either $P_j = P_{j-1}$ or $P_j = Q$.

4. Proceed to the next iteration, $j + 1$.

The above is intended to provide a general understanding of the placement problem, but there some challenges to evaluating Perf() in an implementation of the algorithm.

1. An approach that will consider all solutions will evaluate Perf() I^B times. It is clearly impractical to do this. For example, even for a small program with 5 PLS's and 5 BI's, Perf() will have to be evaluated 3125 times.

2. The straightforward approach for each computation of Perf() will re-evaluate all placement pairs (assignments). This is because a new assignment (or set of assignments) could displace any of the existing assignments.

3. Because of the limited size of the BIT (or similar structure) used to hold placement data, it is possible that a new placement can undermine an existing placement because the two BI's share the same BIT entry. This can occur even though the two BIs are assigned to different PLSs. While the equation above accounts for this, determining cases where this occurs is not trivial.

A practical evaluation scheme therefore needs to overcome the following challenges:
i. An approach that will limit the number of times Perf() is evaluated.
ii. An approach provide a quick way of computing or estimating Perf().
ii. Say that the evaluation described above has to compare $Q1 = (b2, i4)$ with $Q2 = (b2, i5)$. Lets assume that $Q1$ will be chosen over $Q2$. This will be due to the expectation that $Q1$

will benefit more from a placement at $b2$ than $Q2$. However, it is still possible that $Q2$ is the better choice, perhaps because the BI in $Q1$ can easily be placed at other PLSs than $b2$. We can say that $Q1$ is a local optimal solution but not a global optimal solution in this case (local vs. global optimal will be discussed in detail in the next Section).
iii. When an assignment is made, the scheme will have to consider the issue described in 3. That is, it will have to consider how making an assignment may result in a BIT conflict, even if we have two assignments that are made to two different PLS's.

Hence, we need to find a way of evaluating cost that does not involve reevaluating all placement pairs for each new placement. We also need to do the placement in a manner that approaches a global optimal solution, where a placement is done considering not just the benefit of placing a branch at a location but the harm that the placement causes other branches which lose out on the location. One approach to limiting the number of branches that we need to evaluate for a placement site is to consider only branches which are within a maximum path distance pls_max from a control block. This would eliminate the need to explore all possible branches for a given site.

7.4 Simulated Annealing - A Metaheuristic for the Global Optimization roblem

The task of finding an optimal placement of the PLS's for all the BI's in a program is a global optimization problem. We explain why and define the *Placement problem* in the next two Sections. First, we explain what a global optimization problem is and provide background on simulated annealing, a generic metaheuristic for solving the global optimization problem.

7.4.1 Global Optimization

Global optimization usually involves optimizing a given function f of many variables, with the optimization typically being a minimization of the function. Functions representing real-life problems often have several local minima and maxima, but a single global minimum and

maximum. While finding a local optimum can be achieved with simple heuristics, finding the global maximum is difficult and can be a NP-Complete problem. We will illustrate these concepts with the commonly known Traveling Salesman problem and describe a simulated annealing based solution.

7.4.2 The Traveling Salesman Problem

This problem involves a salesman who has to travel through a given set of cities in the most cost effective manner. It can be represented by a graph with the nodes representing cities and weighted, non-negative edges that represent the cost of traveling between two cities. The global optimum solution is one that finds the least costly way of covering all nodes without touching any node more than once, and returns to the source node. It can be found using backtracking but this direct solution is exhaustive (complexity of $O(n!)$ iterations).

7.4.3 Applying Simulated Annealing to the Traveling Salesman Problem

Simulated annealing is a stochastic method, i.e., it uses probabilistic methods in the algorithm. The solution arrived at by simulated annealing may or may not be globally optimal but is likely to be better than a local optimal solution. The name is derived from the physical annealing process, where heating a material causes atoms to be displaced from their initial positions, allowing them to move around randomly. The heating is followed by a slow cooling process that gives the atoms the opportunity to move to more stable, lower energy configurations than they were in initially. The goal is to create a smoother piece of metal by increasing crystal sizes and reducing the defects in the material. As such, the temperature T of the system is initially at a high level and gradually decreases.

To solve the Traveling Salesman problem, the algorithm starts at an initial high temperature and an initial energy level and state. A sequence of moves (node traversals) are made at the temperature and the system moves to a new state. The new state may either be accepted or rejected based on the following probability function. The temperature is lowered

by a fixed amount for each node traversal or based on the energy not reaching a new low for some number of iterations. The equation below is also called the Metropolis Criterion [18].

$$\text{Prob}(\text{E}- > \text{E}', \text{T}) = e^{-(E'-E)/kT}$$

where:

k - A constant,

E - The energy of the current state,

E' - The energy of the new state,

T - The temperature of the new state.

In this analogy, the energy level at a state is simply the total travel cost mentioned in the previous Section. Depending on the solution, the algorithm stops until either a specified temperature is reached or a specified performance criterion is met. There are three important characteristics of the probability function.

1. The greater the reduction in energy between the old and new state, the more likely it is that a new state is accepted.
2. The probability of a move from a lower to a higher energy state is non-zero.
3. The probability of accepting a higher energy state solution decreases as the temperature decreases.

These characteristics demonstrate how the algorithm differs from greedy solutions and is capable of finding global optimal solutions. While the system is generally likely to accept "better" (local optimal) solutions because of 1, it avoids getting stuck at these solutions because of 2. As the number of steps in the algorithm increases (the temperature decreases) as specified by 3, the less likely it is that a higher energy state is accepted. As such, the

algorithm will initially accept worse states almost at random but this becomes increasingly less likely with decreases in temperature, i.e., it starts approaching a greedy solution.

7.5 Modeling the Placement Problem as a Simulated Annealing Problem

From the Traveling Salesman (TS) problem, we see that simulated annealing needs a state (solution) space, an energy function for determining the quality of a solution, a method for determining the set of alternative (neighbor) paths that can replace the current solution, a probability function for determining if a new state should be accepted and a range of decreasing temperature values called an annealing schedule. The table below provides a summary and the equivalent criteria that we use to model the placement problem.

TABLE 7.1: Modeling Traveling Salesman vs. Modeling Placement Problem for Simulated Annealing

	Traveling Salesman	Preload Placement
Solution Space	All paths through graph	All possible placements over all branches, $\|I\|^{\|B\|}$
Energy Func.	Total weight of edges on path	Benefit (mispreds avoided) vs. default,
Alternate solns.	Paths starting at adjacent nodes	Other PLS's where the BI can be placed.
Prob. Func.	Based on difference in energy	Based on difference in energy

The solution space for the TS problem is all possible graph paths that cover all the nodes in the graph. For the Placement problem, it is any combination of placements for the static program branches (it does not have to cover all branches or even all placement sites), i.e., I^B. The energy function for TS is the combined weight of the graph edges through a path covering all nodes. For the Placement problem, it is the unrealized potential that is left over due to branches that have not been placed for this state. For TS, the set of alternative solutions (neighbor paths) for a given node are all the alternative paths other than the current one that can be taken from this node. For the Placement problem, the set of alternative solutions is any other PLS where a given BI may be placed. Since this can be very large, we limit the neighbor set to branches that are a certain distance from the placement site. The fundamental idea behind the probability function does not vary as it is generally based on the difference

in performance (energy) between two states. Similarly, the annealing schedule is a discrete set of declining temperatures and as such, the concept does not change and its not listed in the table above.

Chapter 8
Implementing Preloading for a Processor Front End

The previous chapter explained the preloading technique, the Placement problem that needs to be solved to achieve preloading and gave a general overview of the simulated annealing solution that will be used to solve the Placement problem. This chapter provides a detailed explanation of how preloading would be implemented in real system and how it is accomplished in a processor simulator. The simulated annealing solution is explained in detail, including practical issues that arise.

8.1 Steps in Implementing Preloading

This Section will provide a step by step explanation of how preloading is implemented in our simulator and how it would be implemented in a real system. Preloading will be accomplished within the PAA system described in chapter 4. As described in that chapter, the profile step generates a CTI trace for some sampling of the program's execution. The analysis step analyzes the trace to determine configurations for each branch (or other BI's as described in the previous chapter). The annotation step uses the configuration data to prepare an annotated version of the original program executable. In our simulator, we do not create an annotated binary and decode a new instruction, but we do check if an instruction would have been annotated, and we only mark an instruction as annotated if there is enough carry space available. The profile step for preloading will determine the PC and outcome of each dynamic branch as described previously, but will also need to determine path information for each branch. The major additions that need to be made to implement preloading occur in the analysis step. The profile step and the additions to the analysis step will be described in the following subsections.

8.1.1 Profile Step - Obtaining Trace / Path Data

As described in the beginning of the previous chapter, there can be many paths to a BI and therefore, the data for an individual BI may have to be placed in multiple PLSs to cover most typical instances of the BI. To accomplish this, the profile step not only has to store the PC and outcome of the target BI but also information about the different paths taken to the BI. It does this by recording the distinct paths, or *traces*, that reach each BI starting from some number of blocks before the BI. The concept of traces is illustrated in the Figure 8.1, which shows (in dashed lines format) two traces for the same BI, E1. Trace 1 goes through blocks *(E1, D2, C1, B1, A1)* and Trace 2 goes through *(E1, D4, C4, B2, A1)*. Note that these two traces have two blocks in common. For our simulator, traces are stored as an array of blocks. The length of a trace recorded and stored is limited to a certain *trace length*, *tr_len*, otherwise the number of distinct traces will be extremely high.

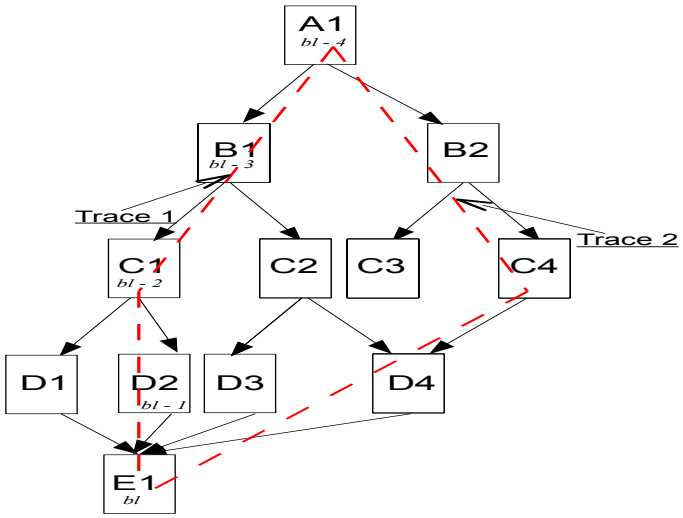

FIGURE 8.1: Control flow diagram indicating different paths to a branch

The main concern with storing traces is whether the amount of data collected for the traces in a program will be prohibitively large. For example, if we have a program with 1000 static branches and the trace length stored is 18 blocks, we can have a maximum of $2^{17} * 1000$ or $131,072,000$ traces. However for the benchmarks we have analyzed, the number of distinct traces per branch is much smaller than this maximum, and so the size of the trace data

TABLE 8.1: Per Block Trace Data

Data Stored	Explanation
Start PC	The PC of the first instruction in the block
CTI PC	The PC of branch or other CTI in the block
CTI Type	Branch, Jump, Call or Return
Branch Outcome	If CTI is branch, its outcome

collected is maneagable. If trace storage size is a problem for other benchmarks, multiple traces will have common element(s), which can further reduce the overall storage required (if a tree representation is used as described below).

Another issue is how the code determines whether a discovered trace is new or an instance of a previously discovered trace. This distinction is made by computing a hash of the start PC and branch outcomes of all the blocks in the trace and doing a lookup for the hash into a sorted table of existing hashes.

A better overall approach would be to store all the traces for a particular branch in a tree representation instead of representing each trace as an array of blocks. This would not only make it easy to distinguish between new and existing traces, but reduce the amount of storage required. We took the latter approach in our simulator because it was easier to implement and time and space were not constraints. A real system would likely take the former approach.

Call a trace *placed* if the BIT data for the BI at the end of the trace is placed in a PLS within that trace, at a distance greater than or equal to the minimum preload distance, pls_min. Consider Figure 8.1 again. The block with the target BI for Trace 1 is labeled bl. The second to last block, $bl-1$, would be where data is stored in the Upsteam configuration described in chapter 5 and shown in Figures 5.2 - 5.3. The information recorded and stored in the trace for each of these blocks is shown in Table 8.1. There is some additional information about the trace as a whole which is also needs to be recorded, such as the number of occurences of the trace (a use count) and the number of times the BI at the end of the trace is taken.

8.1.2 Analysis Step - Obtaining Per Branch Preload Data

This step is identical to the Analysis of Profiled Information Section described in chapter 4. The data collected in the Profile Step is analyzed by predictor specific code to generate per branch configuration (start bit, section length and biasing bit for Spotlight). See Section 4.2 in chapter 4 for details on how analysis is performed.

8.1.3 Analysis Step - Setting Up Traces for Preloading

This step involves computing and setting up the traces collected so that the Placement Problem can be solved and preloading can be accomplished. It can be divided into two parts, determining data that is common for all the traces that lead to the same BI and determining data that is unique to each individual trace. These two parts are described here for Spotlight.

To begin with, the analysis determines whether the instruction in the block at the end of a set of traces contains a BI. For Spotlight, a BI exists if there is a branch in this block and if a non-default configuration has been computed and stored (as described above) for this block. If this is the case, the block containing the BI is called a *Benefiting Block*, BB. For each BB, the predicted benefit for the BI within the BB is computed. This is the expected benefit in accuracy that the BI would have compared to the accuracy if the default configuration is used. For example, if a BI was predicted with 95% accuracy using the default configuration and with 98% accuracy using a special configuration, the predicted benefit for the BI is 3%. Since Spotlight also uses the Agree mechanism, the overall expected benefit is computed by adding the expected benefit from Agree to the expected benefit from the non-default configuration. For example, a BI which is biased 95% in one direction (either taken or not taken) will have a high agree benefit while a BI that is only 70% biased will have a lower benefit.

After this per BI data is determined, computations need to be made for the traces that lead to these BIs. The benefits described above are weighted for each trace based on the

frequency of the trace. If a trace has an overall benefit of x and occurs n times, its expected benefit is xn if the trace is placed, since the total number mispredictions that we would expect to eliminate will be xn. Other per trace information includes whether a trace has been placed or not and a hash of the starting block addresses of all the blocks in the trace.

As described in chapter 5 and illustrated in Figure 5.3, a single cycle latency version of Spotlight and other profile based branch predictors will most likely need to store the data for a branch that is fetched in cycle c in the BIT entry corresponding to cycle $c-1$. We refer to this block corresponding to the BIT entry as the *BIT Block*. For example, consider Trace 1 in Figure 8.1. The preload data for branch E1 in Figure 8.1 may be placed in the BIT entry corresponding to the D2 block (D2 is the BIT Block). Similarly, each trace will have a fixed BIT Block in which the preload data for its BI can be placed. The BIT entry and index into the BIT for the BIT Block of each trace is also computed and stored. The BIT index is the lowest $\log_2 BITSize$ bits of the instruction PC at the beginning of the BIT Block. Lets assume that the PC of the first instruction in block D2 is $0x124F$ and the $\log_2$ size of the BIT is 8. For Trace 1, the corresponding BIT index is $0x4F$ (the lowest four bits of the first instruction in block D2). Note that the least significant bits of the PC are ignored if instructions are aligned, the example used here assumes non aligned instructions for simplicity. If a tag field is used in the BIT, the tag bits will also need to be stored. These bits are the next lowest bits after the index (if four tag bits are used, $0x2$ are the tag bits).

Once the traces have been initialized, we are ready to begin the simulated annealing process and place data for BIs into PLSs.

8.1.4 Analysis Step - Implementing Simulated Annealing

The simulated annealing process starts with a set of traces, and ends with the preload data for these traces being placed in the available PLSs. Since there are a limited number of PLSs for all the traces that need to be placed, the goal of the process is to selectively place the most important traces (those that provide the most benefit) so that the maximum

performance can be achieved. The step by step process by which this is achieved is described here.

1. After the initial set up described above, all the unplaced traces are sorted and ranked in order of the expected overall benefit. The idea is to place the most important traces first, and continue placing traces in decreased order of importance. An initial *unrealized score* potential is also computed, the equivalent of the energy value described in the previous chapter (see Table 7.1). This is the sum of the potential benefit (in terms of reduced mispredictions) that would be achieved if all branches in the program are placed. Consistent with simulated annealing concepts, the goal of the process is to minimize this unrealized score.

2. The unplaced trace with the highest score is selected. For this discussion, the BI of the trace will be called the *placement candidate*. A BB is defined as a basic block that contains a BI. The goal of this step is to select one of the PLSs leading to the block containing the placement candidate for further consideration. Starting with this highest ranked trace, for each unplaced trace, the PLSs starting with the closest PLS to the placement candidate is considered by performing an estimate of the placement score that will be achieved by placing at that PLS. From the example used in the previous Section, PLS block C1 will be the first site evaluated when considering Trace 1 (assuming a minimum preload distance, pls_min of 1), followed by B1 and A1. Note that block $D2$ is not a potential PLS during simulated annealing because of the Upstream mechanism described above.

Fully evaluating the benefit of a placement site is computationally expensive, so the evaluation is done in two passes, the quick estimate for each placement site in a trace (described here), and then a more thorough estimate for the best placement site found. The estimate is the change in unrealized score potential that would be achieved by placing the placement candidate in the PLS. If the BI for another trace is currently placed in the PLS, this BI is a potential *victim*, and the score must be adjusted accordingly. Specifically, we would expect that the estimated change in unrealized score will be negative (good) for the placement candidate because it now has an additional PLS and the estimate will be positive (bad) for

the victim because it loses a PLS. Returning to our example, lets say that PLS C1 is being evaluated for the placement candidate in E1. Further, assume that PLS C1 is currently assigned to hold data for BB F2 (not shown in Figure 8.1). The BI in F2 is a potential victim if its data must be displaced from C1 to make room for the placement candidate in E1. Therefore,

$Overall\ Change\ in\ Score = Change\ in\ Score\ (C1, E1)$

$+ Change\ in\ Score\ (C1, F2)$

where:

(A,B) indicates that PLS A will place data for BI B.

As described in the previous chapter, simulated annealing is a stochastic method. It is necessary to adjust this reduction in score by a random factor (called an annealing factor), so that the algorithm avoids getting stuck at locally optimal solutions (see Section 7.4.3). The relative score for a particular PLS is either increased or decreased based on the annealing factor. This would make it possible to select a PLS with a lower estimated benefit over one with a higher estimated benefit. Further, we also adjust the score by a distance factor. This makes it more likely that the algorithm will select a PLS closer to the BB compared to one further away. After the score is computed for all the potential PLSs (for example, C1, B1, and A1) up to a maximum distance, the PLS with the highest expected reduction in score is selected for the next step.

Recall that a trace has a fixed BIT entry that its preload data will be written to. Figure 8.2 explains what happens during the current step for the BB E1 which is at the end of the trace through *(A1,B1,C1,D1,E1)* (shown by a dashed line). For this Figure, assume a BIT with 2^8 entries. The number of index bits needed to access this BIT will be 8. The index of the BIT entry which will hold preload data for E1 is based on D1, whose starting PC address is $0x325A$. The BIT index used by E1 will be the lower 8 bits of this address, $0x5A$. The current step for E1 starts by evaluating C1, followed by B1 and A1. The figure shows

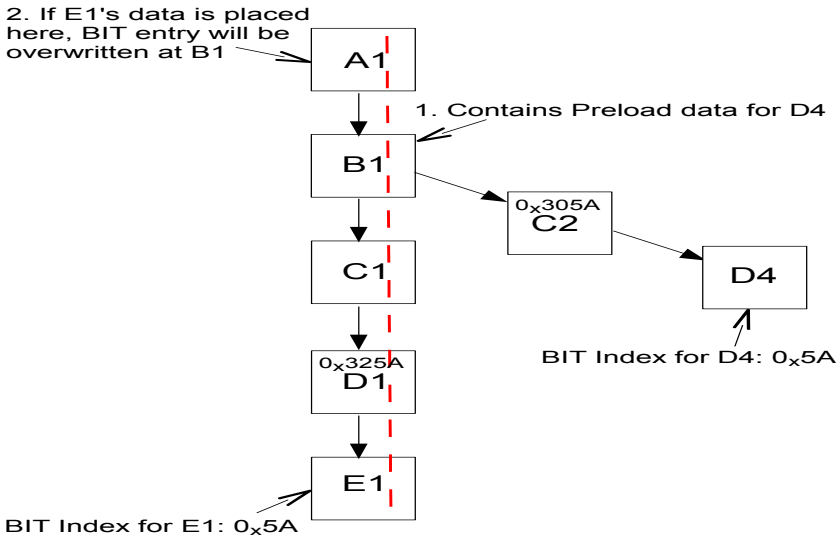

FIGURE 8.2: Control flow diagram showing BIT overlap between E1 and D4

another trace, *A1,B1,C2,D4* that reaches D4 and whose preload data was previously placed in PLS B1. The BIT entry index that this second trace needs to write, $0x5A$, will be the same one that the first trace needs to write. Lets assume that the current step selects A1 as the potential PLS for the first trace. If A1 is selected, the BIT entry for the first trace will *always* be overwritten by the data for the second trace when block B1 is fetched. Clearly, there is no point in considering A1 or any previous blocks before A1 when evaluating PLSs. Hence, this step will always check for this condition. If a potential PLS currently writes the same BIT entry (for another branch) that the trace being evaluated in this step writes to, no previous blocks before that PLS will be considered.

The algorithm checks the BIT entry that each of its potential PLS blocks may currently be writing (for another BI). If the BIT entry that a block writes is the same one that the current BI will need to write, no further blocks in the trace need to be considered (if data is placed in those further blocks, it will just be overwritten by this block). This is another advantage of preloading, as obvious BIT conflicts such as this can be determined and eliminated at compile time because of the path information that is available.

3. After a potential PLS is selected for a given placement candidate BB, this step does a more complete calculation to determine if a placement should be made at the selected PLS,

say $PLS1$. If there is another BI that is currently placed in $PLS1$, it is the victim BI. Firstly, as is done above, the amount by which the placement candidate BI will benefit by being placed at $PLS1$ is computed. Next, the algorithm goes through a list of all the other possible BIs that can be reached from $PLS1$ and checks to see if any of these have already been placed in another PLS. If they have been placed elsewhere, and the BIT index of the placement is the same as the placement candidate or victim, there is a possibility that one or more of these other BIs will now be displaced. The benefit to the placement candidate is therefore adjusted by the negative effect that might occur if any of these other BIs are now displaced. If the improvement from placing the placement candidate is still adequate, $PLS1$ is selected as a PLS for this BI and its data is preloaded into it. After the placement is completed, the new scores for all affected traces are computed. Any trace that has been displaced by the placement is added to the list of unplaced traces.

4. The algorithm returns to Step 1 for another round. Based on the simulated annealing algorithm, if the reduction in unrealized score is large enough, the annealing factors are adjusted to reduce the likelihood of a worse PLS being selected over a better PLS. Recall that simulated annealing starts with a high likelihood of accepting worse solutions and this likelihood gradually decreases as the algorithm proceeds.

The algorithm stops after a certain number of temperature reductions or a certain number of rounds (one round is Steps 1 through 4 described above).

8.2 Carry Space Needed for Preloading

In chapter 4, we discuss how the configuration data for branches will be carried in special carry instructions that duplicate existing instructions, with part of the immediate fields and unused fields carrying the configuration data. See Section 4.3 for more details. The basic idea does not change when preloading is introduced. However, the amount of data that needs to be stored in carry space increases with preloading. When preloading is not used, the BIT entry for a BI, as well as the tag bits, are determined at run time when the BI first occurs

so that the entry is used for subsequent occurences of the BI (see Figure 5.1). The data is not available during the first occurrence of the BI. Since the goal in preloading is to place data for a BI upstream and have it available in the BIT by the time the BI is fetched for each occurrence of the BI (including the very first occurrence), the BIT index will also have to be carried in carry instructions in addition to the predictor configuration. If a tag field is used in the BIT, the tag bits will also have to be carried.

In Section 6.7, we looked at branch instructions and discovered that they typically have adequately short displacements to store the 10 to 11 bit configuration for Spotlight without adversely affecting performance. Because of the additional bits that need to be carried for preloading, it is no longer sufficient to use a single branch based carry instruction. Additional space is needed to store the BIT index and tag bits. Other than branches, there are several potential instructions such as ADD, SUBCC and LDW which also have immediate fields that can be used to store carry bits. We evaluated some of these instructions for SPARC V9 to determine a set of special carry instructions in addition to the special branch instruction proposed previously. We then looked at how much total space will be available in typical basic blocks if this set of carry instructions is used. The results are demonstrated in the next chapter.

8.3 Initializing Preloading

Simulated annealing optimizes the placement of BIs in PLSs after starting at an initial state. This initial state can be empty (none of the BIs are placed), but this is not a requirement of the algorithm. In fact, the algorithm can be initialized to any configuration before being optimized. The disadvantage of an empty initial setting is that it does not consider the configuration of Spotlight that was discussed in previous chapters. In this configuration, the data for a BI is stored within the BI itself and loaded into the BIT on the first occurrence of the BI. Since a BI carries its own data, this version of preload data placement is called

Auto Preload. Another disadvantage of an empty initial setting is that the algorithm will take longer to converge on a good solution.

In addition to considering an inital empty state, we also consider initializing the annealing process with the Auto Preload configuration, where every BI is its own PLS and carries its own preload data. The possible advantage of doing this is that for some BIs, the best placement for it might be Auto Preloading its data. Specifically, consider a BI which is reached by a large number of paths (traces) and each of these paths are taken with roughly the same frequency. The best configuration for this type of BI might be Auto Preload, since the alternative is to place the BI in a large number of PLSs. Since PLSs are limited, doing so can adversely affect the performance of other BIs.

Therefore, we evaluate both the empty and Auto Preload initial configuration. The results are shown in the next chapter.

8.4 A Special Case - The Agree Predictor

The preloading technique discussed so far has been described generally with specific references to Spotlight. Preloading can be used for any front end scheme that uses statically chosen data with little or no changes to what has been described. This section discusses a special case which is particularly suited for preloading with some changes. The Agree predictor is described in chapter 3 and is illustrated in Figure 3.1.

The special feature of Agree is that each branch requires only one bit of preload data, called a biasing bit. This provides an opportunity to improve on the preloading technique described so far. Recall that pls_min indicates the minimum distance (in basic blocks) from a BI that its data can be placed at in a PLS. Lets assume that this value, d, is set to 2. Further, assume that all CTIs within this distance are branches. If we start at a given PLS, there are only two possible BIs that can be reached from the PLS at a distance, $d = 1$ (because the PLS may either be taken or not taken). For $d = 2$, there are four possible BIs that can be reached from the PLS. Hence, storing the biasing bit of ALL possible BIs at

$d = 2$ from the PLS will only need 4 bits. In general, a total of 2^d bits are needed to store data for all possible branches at a distance of d from a PLS. The hardware structure that stores the biases for these branches is called a *bias vector*.

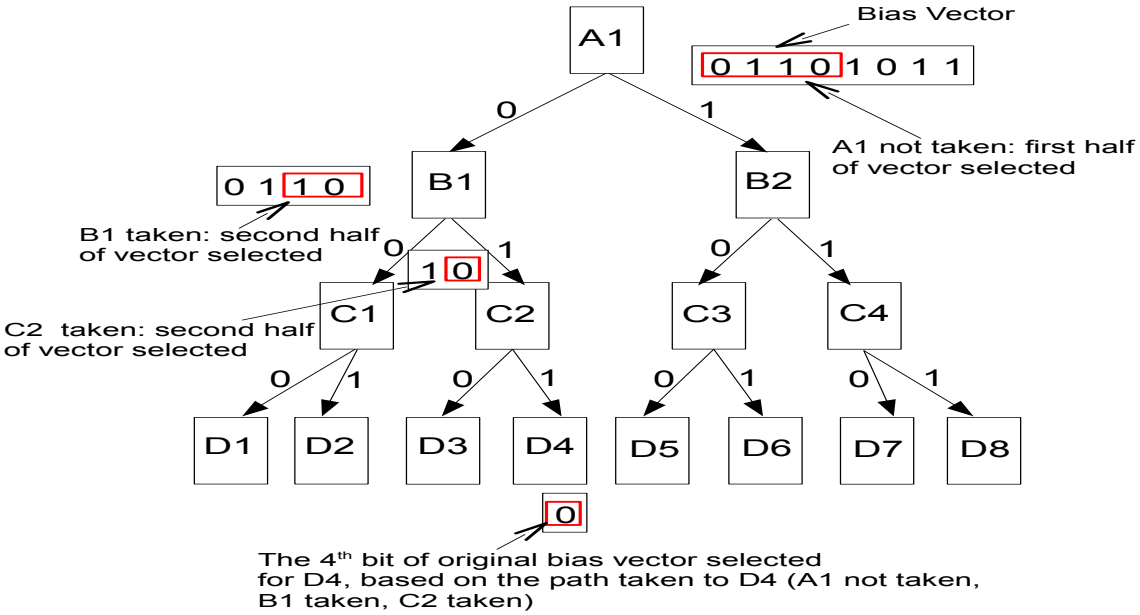

FIGURE 8.3: Control flow diagram showing BIT overlap between E1 and D4

This concept is illustrated in Figure 8.3, which shows an example where $d = 3$ and all CTIs are branches. The left arrows marked with a 0 show the not taken path from a branch and the right arrows marked with 1 show the taken path. Starting at block A1, there are a total of 8 possible blocks, D1 through D8, that can be reached at a distance of 3. If A1 is not taken, the bias vector can be narrowed down to its first half and B1 is reached. If B1 is taken, the bias vector from A1 can be narrowed down to its second half and C2 is reached. Finally, if C2 is taken, the bias vector from B1 can be narrowed down to its second half. The bit that remains is the bias of branch D4. Hence, the bias for D4 is found in the fourth bit (from the left) of the original bias vector at A1. In general, the bias for a branch DX can be found in the Xth (from the left) bit of the original bias vector at A1. If all CTI's are branches, the bias for any branch at a distance d from a PLS can be found and easily extracted starting with a 2^d bit bias vector placed at the PLS.

Therefore, we consider a version of preloading that always preloads the bias vector at a fixed distance. This is feasible because the number of bits required to cover all possible branches at a short distance is small and easily manageable. We call this version of Agree the *Agree Preplace* predictor.

The obvious problem with the above technique is that not all CTI's in a program are branches. When indirect jumps are encountered, they can reach several target branches. This is dealt with by adding the taken count of all the target branches from a PLS and using the sum to determine a single bias.

Preloading actually provides an opportunity to improve on the original Agree predictor since it has the ability to consider path information. In the original Agree predictor, a branch has a fixed bias. This can be improved upon by having multiple biases for a branch, based on which path is taken to that branch. Instead of a per branch bias, the bias of the trace from a PLS to a branch is placed in the PLS. If one path to a branch results in the branch being mostly taken, and another path results in it being mostly not taken, the branch will use a different bias for each case. This can provide better performance than storing a single bias for the branch for both paths to it.

Chapter 9
Evaluating Preloading

9.1 Experimental Framework

The simulator configuration, benchmarks and sampling method used in this chapter are identical to that described in chapter 6. Traces for benchmarks were collected and stored in one simulation step. The length of each trace stored was 20 basic blocks. The trace blocks have a single exit, but can have multiple entrances.

9.2 Set of Experiments

We evaluate preloading for Spotlight by comparing the accuracy of versions that use preloading to non-preloading versions of these predictors. We also evaluate the accuracy of the Agree Preplace predictor. We break down the performance of preloaded Spotlight by looking at different measures that contribute to performance. We evaluate the effect that different initialization methods have on preloading and look at data such as BIT hit ratios and compute synthetic prediction ratios (an estimate of the expected prediction rate) to help understand the performance of preloaded Spotlight. We also evaluate the availability of carry space in existing program instructions.

9.3 Comparing Preload to Non-Preloaded Spotlight

Figures 9.1 - 9.4 show the misprediction rates for Spotlight for Preload vs. non-Preload configurations for four PHT sizes. The ideal BIT hit rate (which assumes that profiled data is always available) is also shown, as it represents the best case performance that can be expected to be achieved. All configurations use a relatively small BIT with 2^6 entries. From Figure 9.1, we see that preloading is able to close the performance gap between non Preload and Ideal by approximately half for gcc, crafty, parser, perlbmk, gap, bzip2 and twolf. This is also the case on average. The other benchmarks do not benefit much from preloading, with

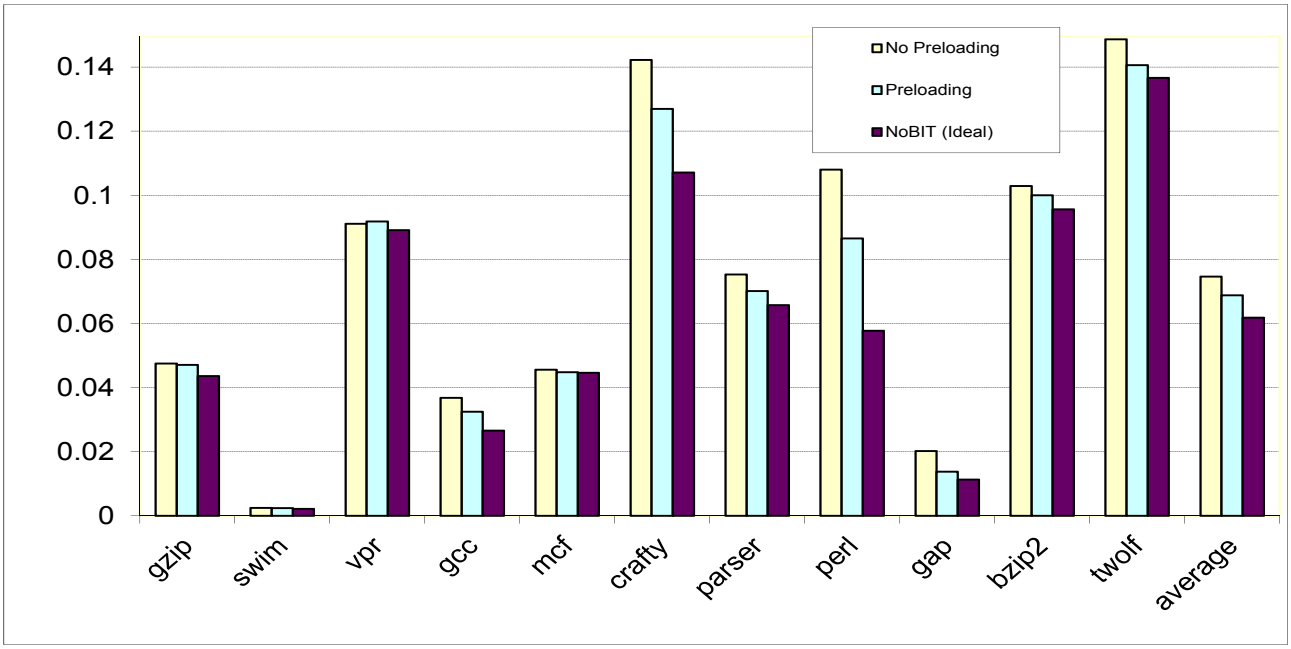

FIGURE 9.1: Misprediction Rates for Preload vs. Non Preload (PHT Lg Size 10)

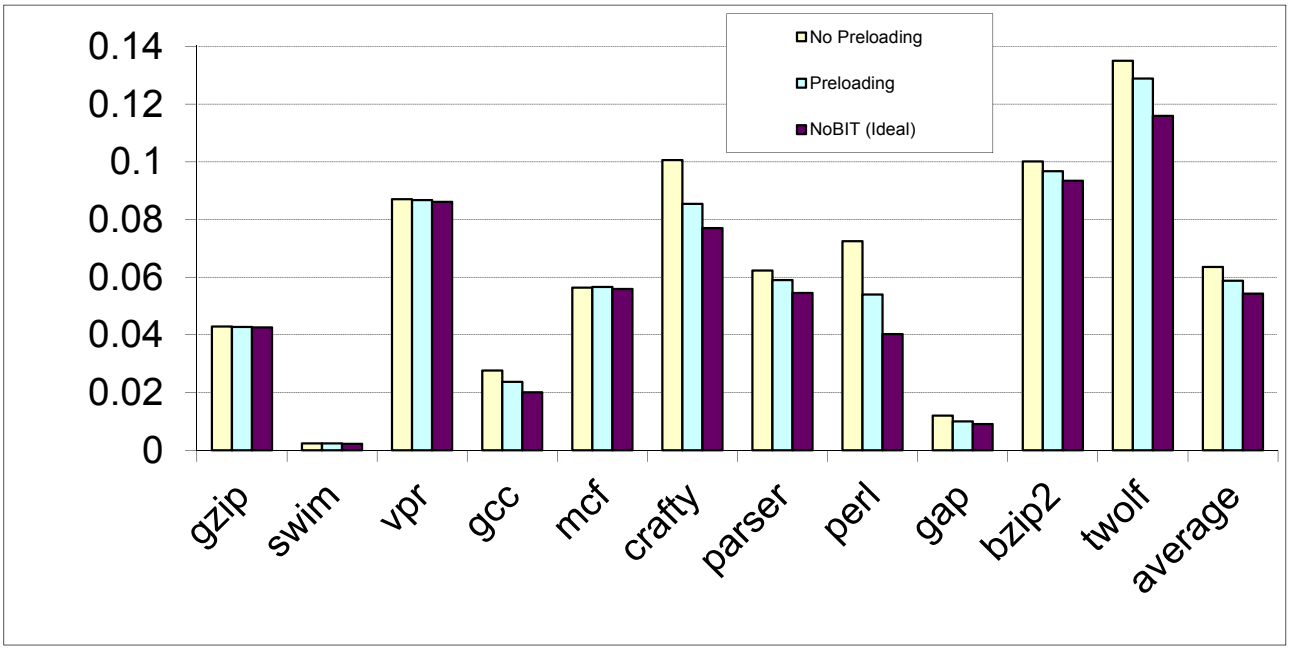

FIGURE 9.2: Misprediction Rates for Preload vs. Non Preload (PHT Lg Size 12)

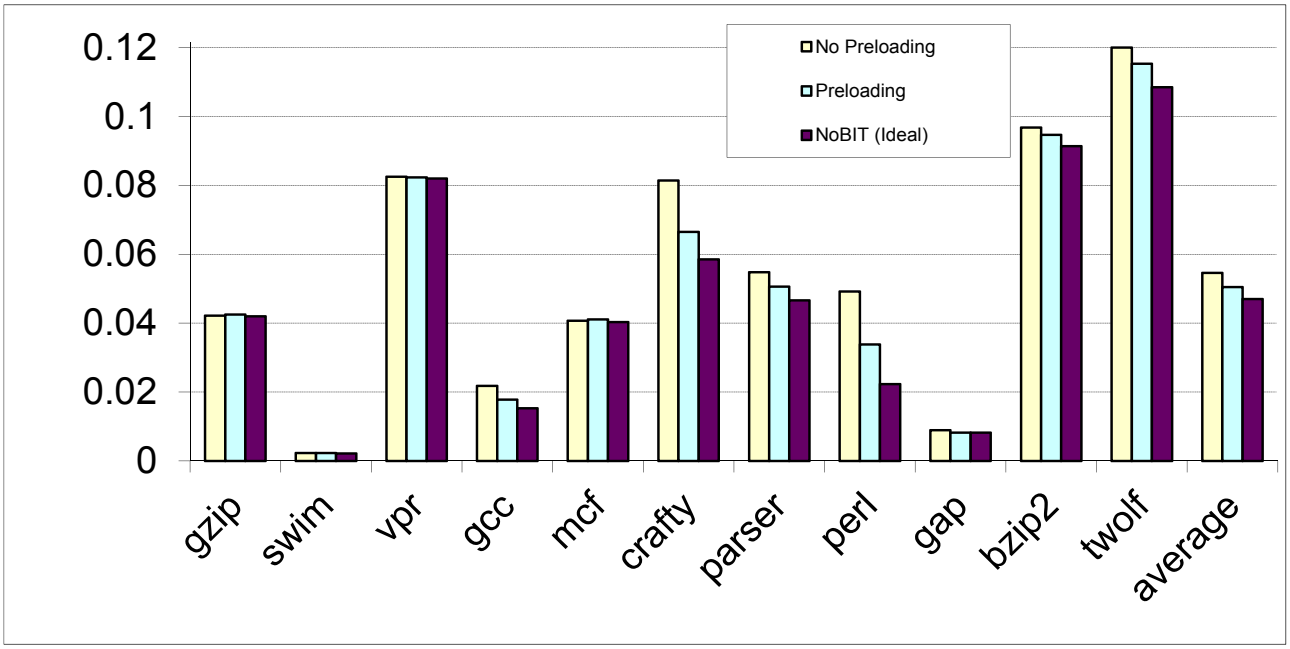

FIGURE 9.3: Misprediction Rates for Preload vs. Non Preload (PHT Lg Size 14)

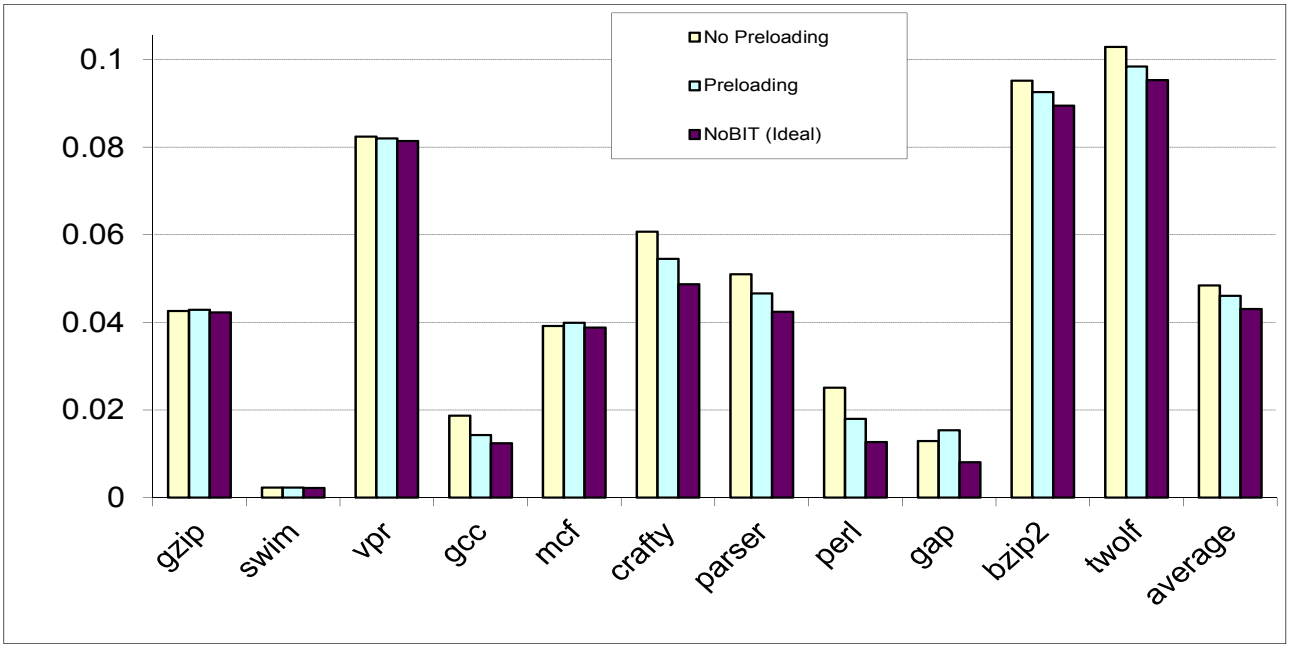

FIGURE 9.4: Misprediction Rates for Preload vs. Non Preload (PHT Lg Size 16)

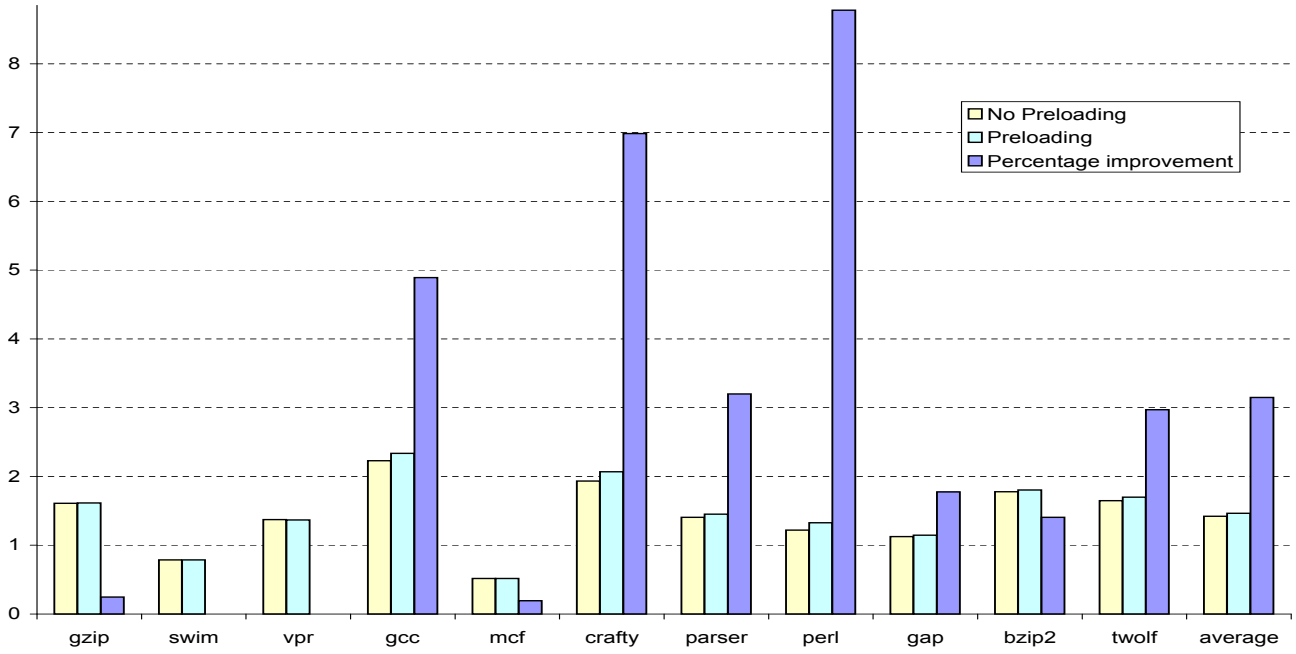

FIGURE 9.5: IPC Improvement for Spotlight with Preload vs. Non Preload (PHT Lg Sz 10)

vpr suffering having a slightly higher misprediction. The benefits of preloading continue to hold as the PHT Size increases, as shown in the figures for PHT lg size 12 and 14. The amount of actual improvement declines, but this is expected as overall misprediction rates decrease. Further, as the PHT size increases, the number of branches that need special configurations diminishes. Due to fewer conflicts in the PHT at larger PHT sizes, more branches can use the default configuration without suffering a loss in performance.

A very important point is that since preloading is a technique where the complexity is moved away from hardware to software, the software developer can make a choice about whether it should be used or not. For example, if the developer believes that preloading will not be beneficial, they can choose to not perform the analysis and annotation steps associated with preloading (see chapter 4). Instead, the developer can choose to either not use profiling at all, or use profiling to implement the non-Preloaded version of Spotlight. The developer's decision could be based on the developer's feel for how reliable the training data is. That is, for some benchmarks there might not be a good set of training data so far as Spotlight and preloading is concerned. This is in contrast to a pure hardware scheme on a general purpose

processor which have to be used for a diverse range of programs, and therefore, the cost of the scheme has to be weighed against the benefits that are expected over all programs. The analysis code can also be improved so that preloading is only considered if it is expected to help based on some criteria. This would automate the decision and make the developer's job easier. However, this may not be straightforward because the performance on the training input compared to actual (reference) input may differ.

The improvement in terms of IPC is shown in Figure 9.5 for PHT size 10. The improvement is significant for the benchmarks which benefit most from preloading such as gcc, crafty, parser, perlbmk and twolf. For example, the misprediction rate for perlbmk improved by 2.15%, which resulted in an IPC improvement of about 9/

The following subsections will examine the factors leading to the performance seen here.

9.3.1 Performance and Frequency of Most Mispredicted Branches

The main goal of preloading is to optimize the use of space in a small BIT. Preloading will not be beneficial if the most frequently mispredicted branches can easily fit in the BIT, i.e, the BIT hit ratio is high even when the BIT is small. In the previous Figures, we saw that gzip, swim and mcf do not benefit from preloading, with vpr suffering a slight performance loss. These are also the benchmarks that were not affected much by the BIT size when preloading was not used, as shown in chapter 6, Figures 6.2 - 6.4. On the other hand, the former benchmarks were all sensitive to BIT size and saw an improvement in performance with every increase in the BIT size.

Figure 9.6 clarifies this relationship. It shows what percentage of dynamic mispredictions occur due to the top 30 most mispredicted branches. It is clear immediately that gzip, swim, mcf and vpr do not need a large BIT, since the 30 most frequently mispredicted branches make up greater than 90% of mispredictions in these benchmarks. The number is very close to100% for swim and vpr. On the other hand, they make up less than 20% of mispredictions in gcc, and about 50% or less in crafty, parser and perlbmk. They make up

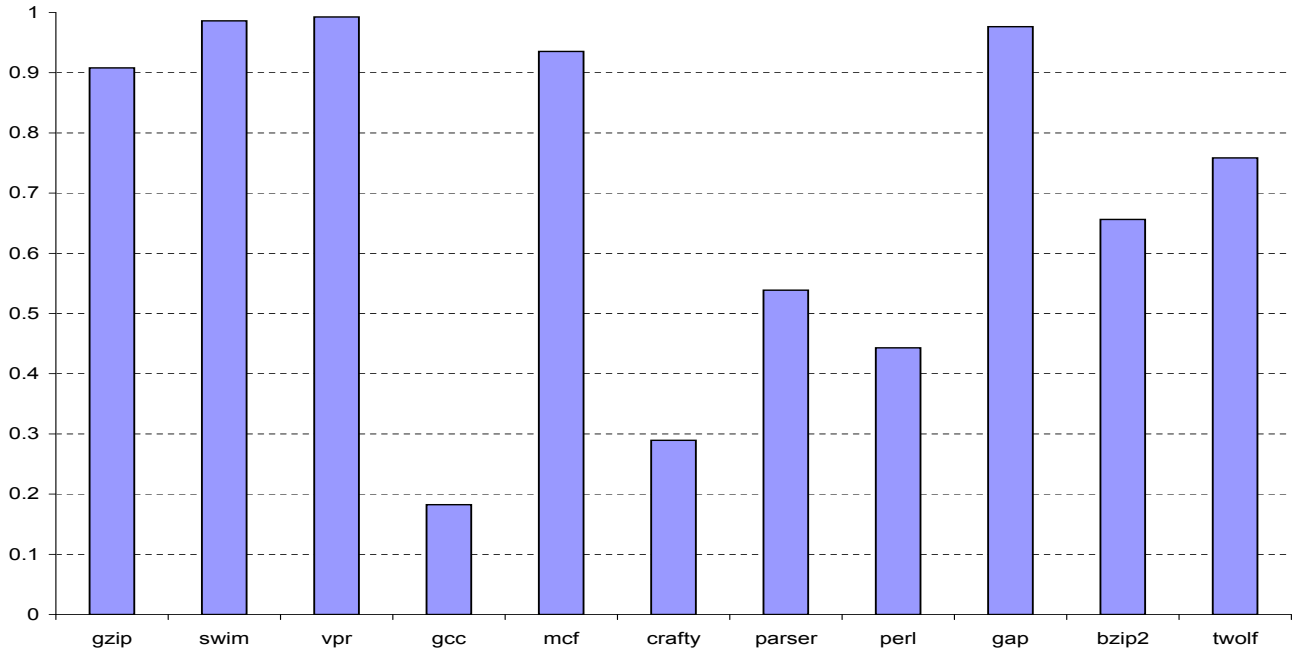

FIGURE 9.6: Percentage of Mispredictions Due to Top 30 Branches

less than 80% in bzip2 and twolf, the two other benchmarks which consistently benefit from preloading. The higher percentages for these two benchmarks corresponds with relatively lower benefits from preloading, i.e., while these two benchmarks benefit from preloading, the benefit is not as significant as the other benchmarks. The 30 most mispredicted branches are 97.6% of mispredictions in gap. While gap benefits from preloading at the smallest PHT size (Figure 9.1), the benefits diminish at higher PHT sizes. A possible reason is that gap has a few branches which benefit from a special configuration, but the branches are constantly evicted from the BIT, perhaps due to conflict misses.

9.3.2 Performance and BIT Miss Rates

The more branches that can find their special configurations in the BIT, the higher the prediction rate is likely to be. We examine the correlation between performance and BIT miss rates in this Section. The BIT miss rate is the percentage of branches that have a special predictor configuration but do not find it in the BIT. Figures 9.7 - 9.8 show the BIT miss rates for a small (lg size 10) and large PHT (lg size 16) respectively for preload and

non-preload configurations. The number on top of each bar shows the absolute improvement in misprediction rate for preloading over non-preloading.

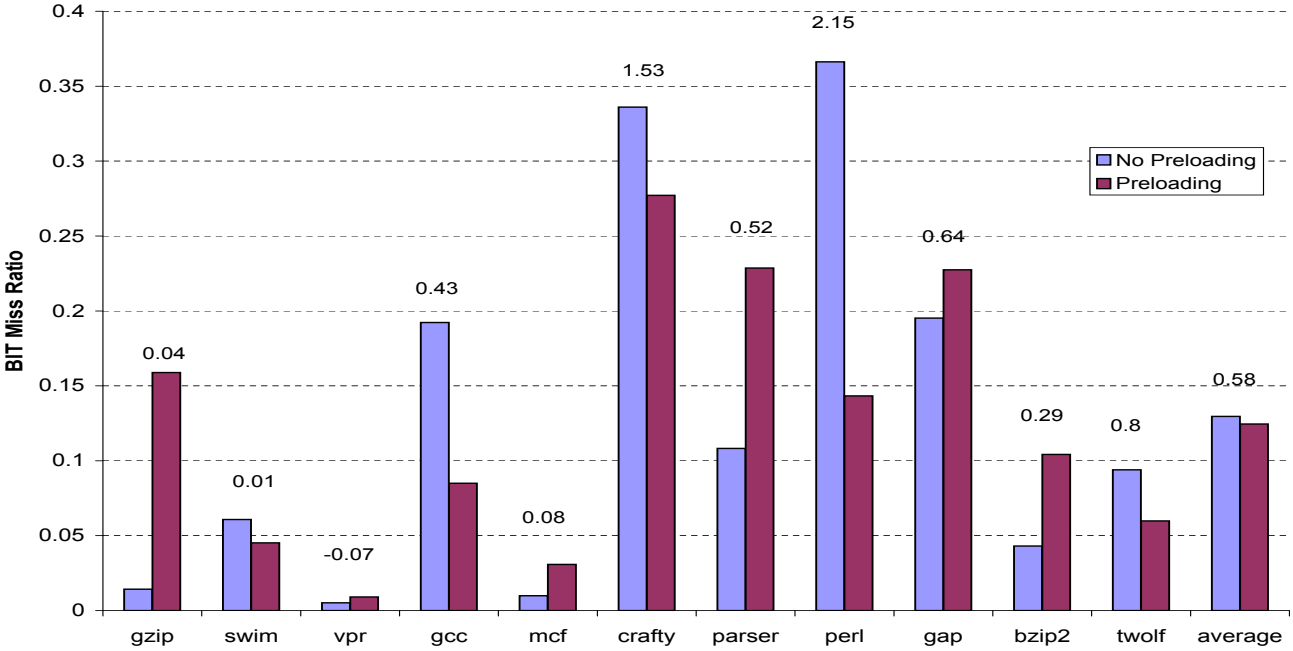

FIGURE 9.7: BIT Miss Rates for PHT Sz 10

From both Figures, we can see that gcc, crafty, perlbmk and twolf benefit from preloading by achieving a signficantly lower BIT Miss rate. This is not the case for parser, gap and bzip2 at PHT size lg 10 and for parser and bzip2 at PHT size lg 16. For the other benchmarks, preloading has a higher miss rate but this does not correlate with significantly higher misprediction rates. Hence, we can conclude that while a significantly lower BIT miss rate results in better performance, it is not a necessary condition.

These results are not surprising because the goal of preloading is to make efficient use of the BIT space. As illustrated in Figure 8.2 and discussed in Section 8.1.4, preloading uses techniques to avoid obvious BIT misses by taking advantage of the path information available. Further, preloading places data based on both how frequently a trace executes and how much a branch is expected to benefit by using the special configuration over the default configuration. Therefore, if a branch benefits little from a special configuration, preloading may not place its data in a PLS or set of PLSs. In general, this technique will minimize

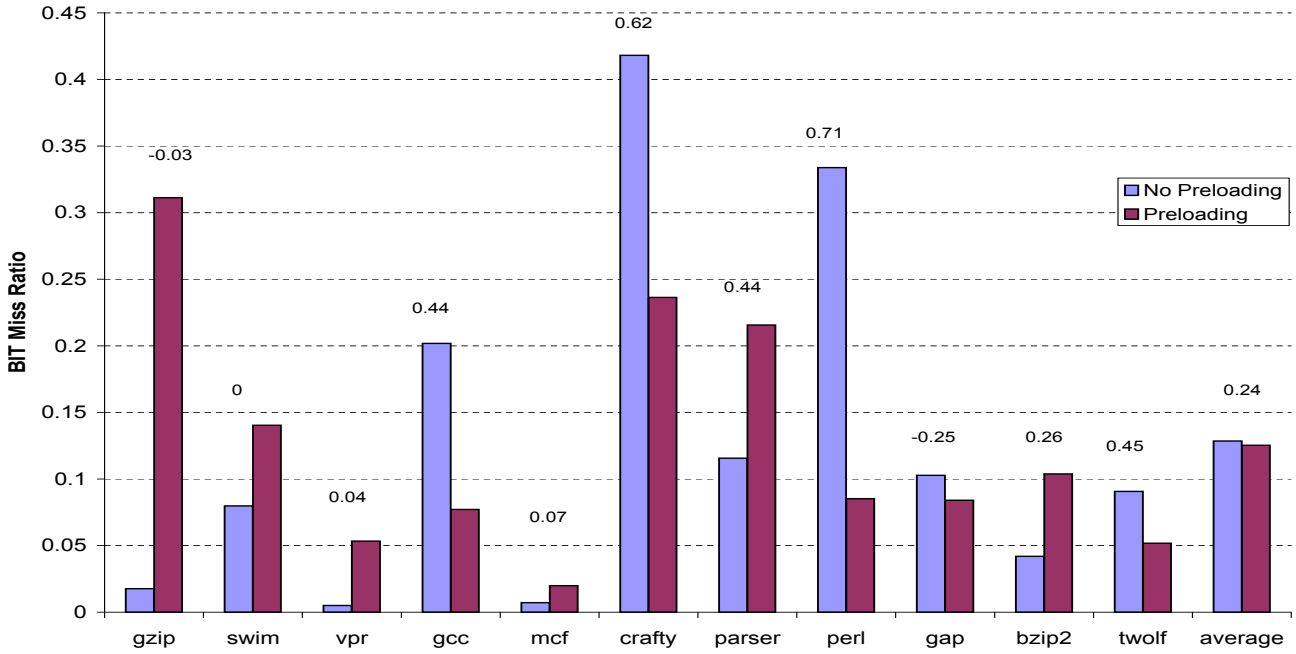

FIGURE 9.8: BIT Miss Rates for PHT Sz 16

the performance impact of BIT misses, by trying to ensure that the misses are for branches which do not benefit much from preload data. By ensuring that more important branches stay in the BIT at the expense of less important branches, it is even possible to get better performance with higher miss rates, as is the case with parser, gap and bzip2 for Figure 9.7.

9.3.3 Performance and Synthetic Misprediction Ratios

During the profiling analysis step, we compute a synthetic ratio which estimates the expected misprediction ratio. This is computed as follows. As described in Section 4.2, the accuracy of all possible Spotlight segments including the default and selected special configurations are computed for each branch. Therefore, if a trace is placed, the accuracy of the special configuration for that trace is multiplied by the frequency of the trace. If a trace is not placed, the accuracy of the default configuration is multiplied by the frequency. This should give us an estimate of the misprediction ratio expected during the actual execution (the reference run) of the program. Figure 9.9 shows the ratio of the synthetic to actual misprediction ratios. To avoid differences due to the program input, we use the same data to do both the training and reference runs. A ratio of 1 indicates that the actual rate is exactly what is

expected. A number greater than 1 indicates that the synthetic misprediction ratio is higher than actual.

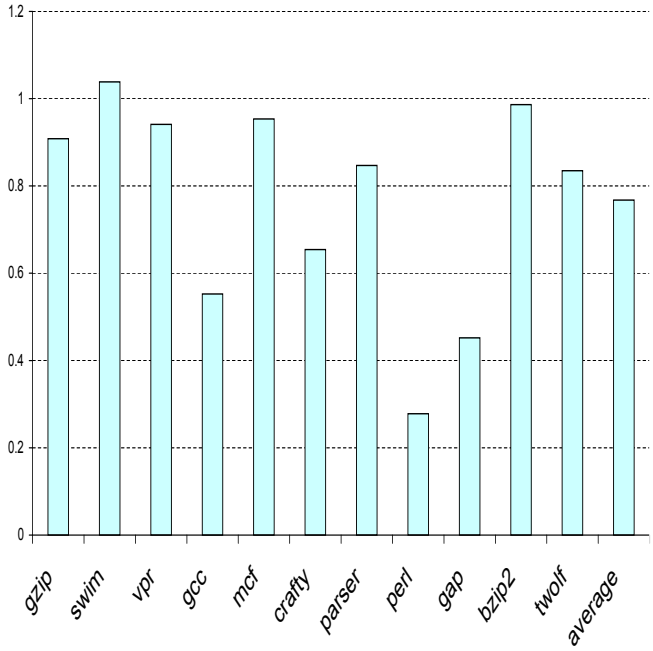

FIGURE 9.9: Ratio of Synthetic Misprediction to Actual Misprediction Ratio

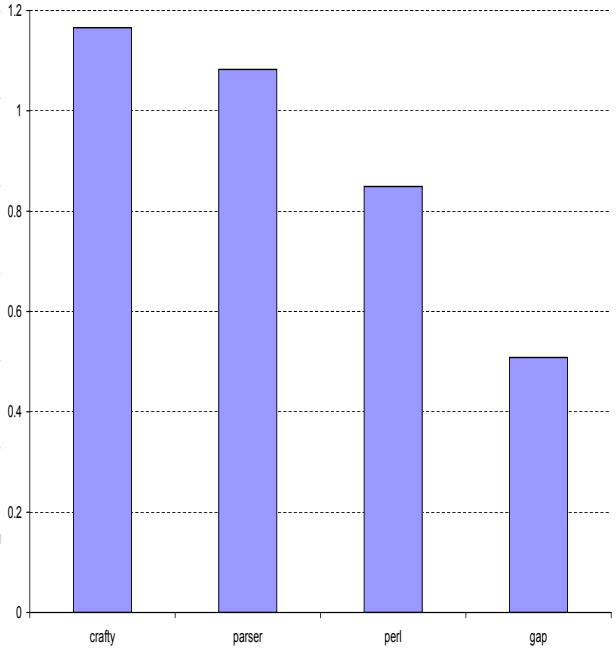

FIGURE 9.10: Ratio of Synthetic to Actual with Larger PHT

We see that while some benchmarks have a ratio very close to 1, many benchmarks have a much lower rate, indicating that the actual misprediction rate is higher than expected. The benchmarks gcc, crafty, parser, perlbmk and gap show the biggest difference. This is the case even though we use the same exact inputs for training and reference runs. We considered that another possibility for the difference is due to a higher number of PHT conflicts in the reference run. To evaluate if this is the case, we increased the size of the PHT used during the reference runs to avoid conflicts. The maximum number of GHR bits used to index the PHT did not change, but additional PC bits were added to the PHT index so that branches which mapped to the same PHT entry before would map to different PHT entries with the change. Figure 9.10 shows the results for 4 of the benchmarks with the biggest difference in absolute misprediction ratios from Figure 9.9.

We see that adding the extra PHT bits reduces the difference between expected and actual. The most significant difference is for perlbmk, which improves from a ratio of 28% to 85%.

The three other benchmarks evaluated also show a significant improvement. The benchmarks crafty and parser move from a ratio of < 1 to a ratio > 1 . The reason for this is that during profiling, we use unique PHT's to evaluate the accuracy of each configuration. During the reference run, there is only one PHT used for all branches. As such, the profile run may not be able to properly account for PHT conflicts. The difference is still significant for gap, but this is somewhat misleading since the overall accuracy for gap is very high and a small difference in accuracy results in a big change in the ratio. Of course, by increasing the PHT size during the reference run, we introduced an additional difference in conditions by having a smaller PHT during training runs. Further, the training runs are done on a functional simulator while the reference runs are done on a timing simulator. These and possibly other factors account for the difference in misprediction ratios that still exist.

9.3.4 Performance and Preload Initialization

We discussed two options for initializing the simulated annealing, preloading algorithm in Section 8.3. One option was to initialize the placements to empty, and the other was Auto Preload, where each branch carries its own data initially. These two initalizations are evaluated in Figure 9.11.

The difference between the two methods is not significant. The benchmarks gcc, mcf, perlbmk, gap and bzip2 do better with the auto preload initialization. The benchmarks vpr, crafty and twolf do better with the empty initialization and the other benchmarks are unchanged. The benchmark which benefits the most from auto preloading is perlbmk which improves by a relatively significant misprediction ratio of 0.41. The results here indicate that the initialization technique has some impact on misprediction rates, although it is not significant. Which choice is better depends on the benchmark.

9.4 Evaluating Carry Space

Section 6.7 demonstrated that branch instructions have adequately small displacements for replacement branch carry instructions with 10 or 11 bits of carry space to be effective. But

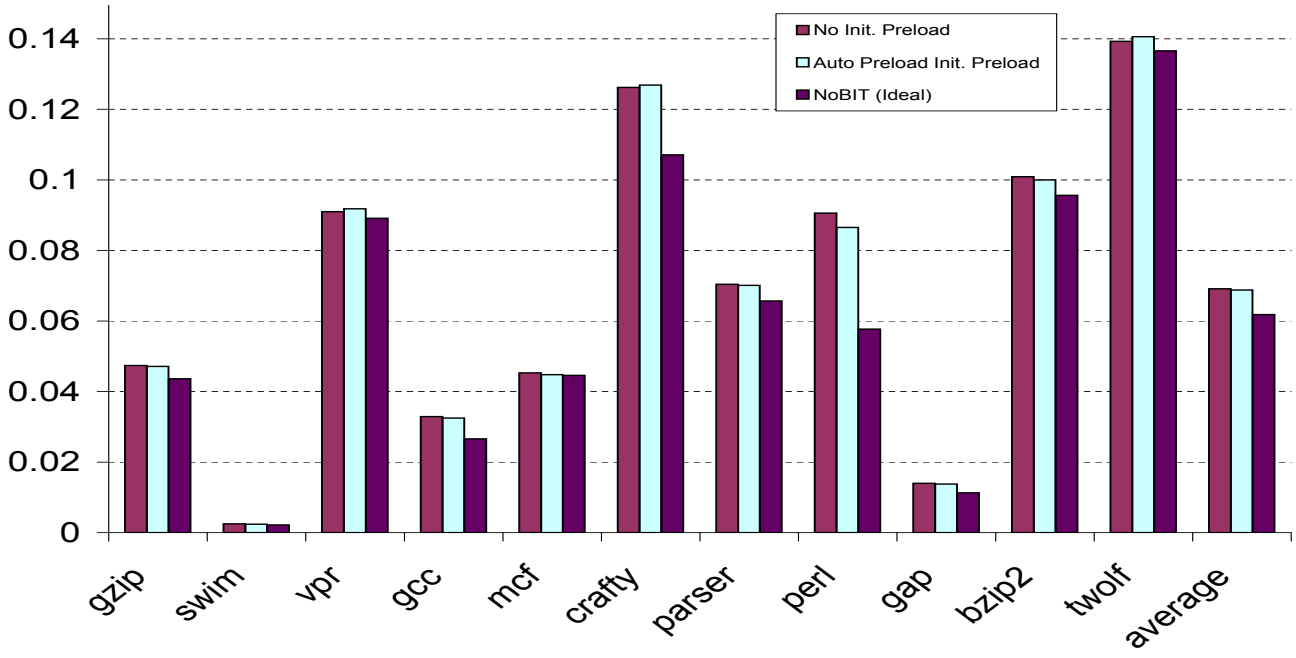

FIGURE 9.11: Preload with Empty vs. Auto Preload Initalization

preloading requires more data, a BIT index and tag, straining the capacity of a single branch to carry the data. For Spotlight, we need 10 bits to represent the configuration. If a BIT of 2^6 entries is used, the BIT index will require an additional 6 bits and tag field will require 2 to 6 bits. The total number of carry bits needed for Spotlight will be 18 to 22 bits. Other profile based predictors such as EHB and Agree need fewer bits, but unless the configuration itself is very small, the space available in branch instructions is not adequate by itself to carry data for preloading. We need to find additional instructions which can also carry preload bits, with the hardware being responsible for combining bits from several instructions in a block. Data on the amount of available carry space would also be useful for those designing other preloading. mechanisms.

9.4.1 Evaluating Per Instruction Carry Space

As explained in Section 8.2, there are many SPARC V9 instructions such as Load, Store, Add, Shift and Sub which are potential candidates for storing carry bits. The opcode map for Sparc V9 [41] indicates that there are five free opcodes available for this type of instruction. Based on a study of the instruction frequency and the amount of bits available, we selected ADD,

SUBCC, LDUW, STW and SLL as the instructions for which special annotated versions will be created.

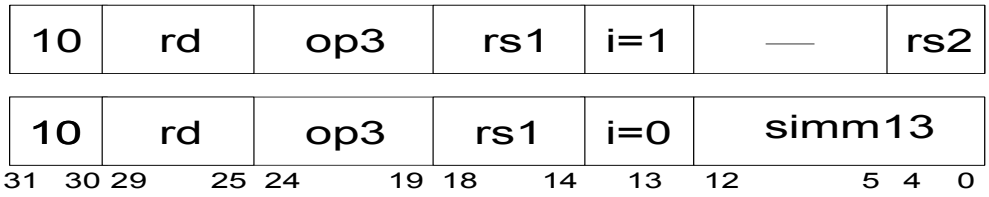

FIGURE 9.12: SPARC Instruction format for ADD, SUBCC, LDUW and STW

Figure 9.12 shows the format for four of these instructions. The first format is for the non-immediate (two register fields) version of these instructions, and the second format is for the immediate version. The non-immediate version always has 8 free bits, which is available for holding carry data. The immediate version has a 13 bit immediate field, and as we did for branches, we evaluate the bit size of immediate values in our benchmark to determine how many of these bits can be used for carry bits in annotated versions of the instruction. The format for SLL is shown in Figure 9.13. Both the non-immediate versions always have 7 free bits for these instructions. The size of the immediate field (shcnt32 in Figure 9.12) is limited to 5 bits and we also evaluate how many of these bits can be used for carry bits.

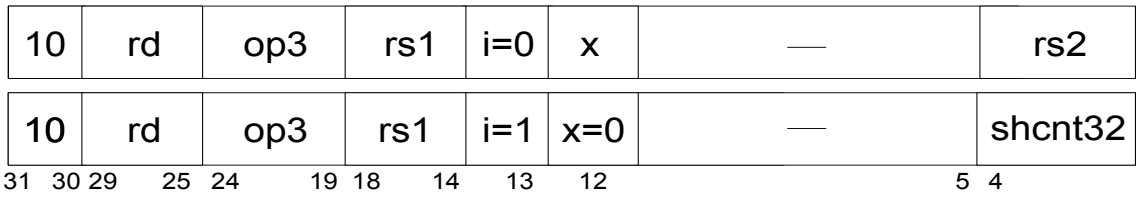

FIGURE 9.13: SPARC Instruction format for SLL

Figures 9.14 - 9.18 show the percentage of immediate instructions which can give up a certain number of bits of carry space for each of these instructions. The data is shown for a few selected benchmarks and on average over all benchmarks. The number in brackets in the Figure labels indicate what percentage of dynamic instructions this instruction represents on average. For example, ADD instructions make up 11.1% of all program instructions on average while SUBCC makes up 10.3%.

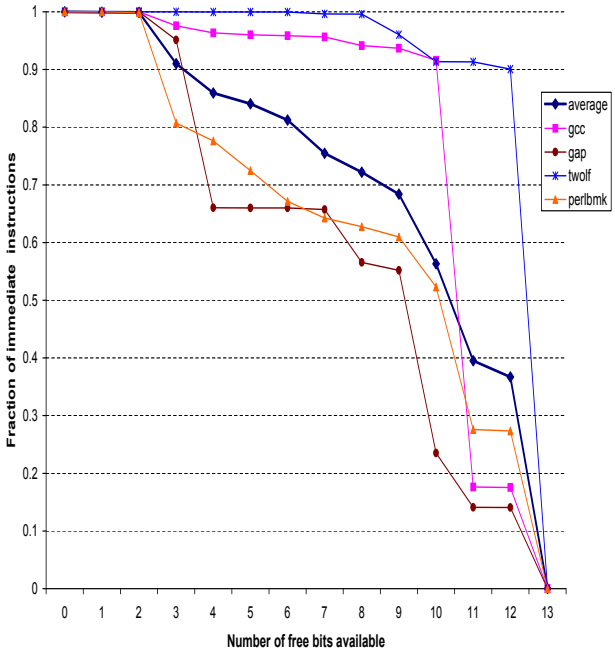

FIGURE 9.14: Free Space Available in Immediate ADD (11.1%)

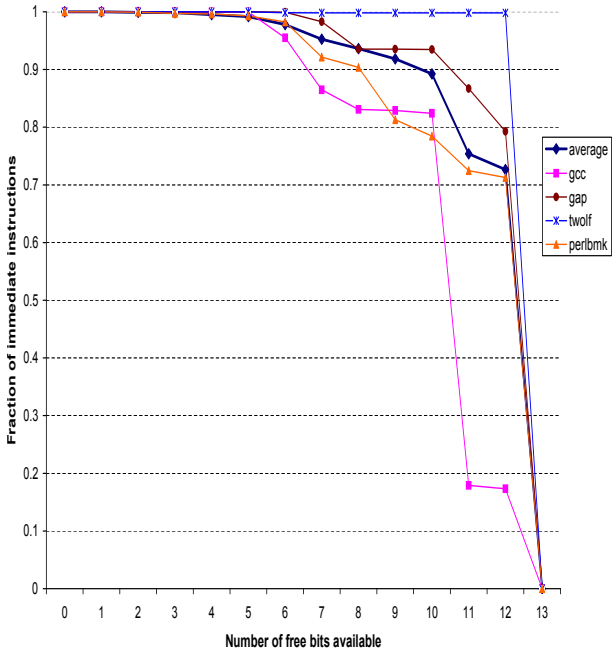

FIGURE 9.15: Free Space Available in Immediate SUBCC (10.3%)

We see that ADD immediate instructions use most of their immediate fields on average, i.e, they have large absolute immediate values. Almost 100% of instructions can give up 2 bits for carry, but the number drops to about 90% when we go to 3 bits and the percentage continues to decline steadily after that. However, this is not the case in all benchmarks. We see that more than 90% of ADDs only need an one or two bit immediate field in gcc and twolf and can give up 11 bits for carry. On the other hand, gap and perlbmk see a drop off starting at 3 bits.

The immediate fields in SUBCC are very small and on average, approximately 90% of instructions can give up 10 bits for carry. The variation across benchmarks is not extreme, the four benchmarks shown each have 80% of SUBCC immediate instructions that can give up 10 bits. For twolf, almost all SUBCC immediate instructions can give up 12 bits, indicating that in most cases, the instruction has an immediate value of 0.

The immediate field in the LDUW instruction can be reduced by more than three bits for more than 90% of instructions and by five bits for more than 80% of instructions on average. For this instruction, the benchmark twolf has the biggest immediate fields and the

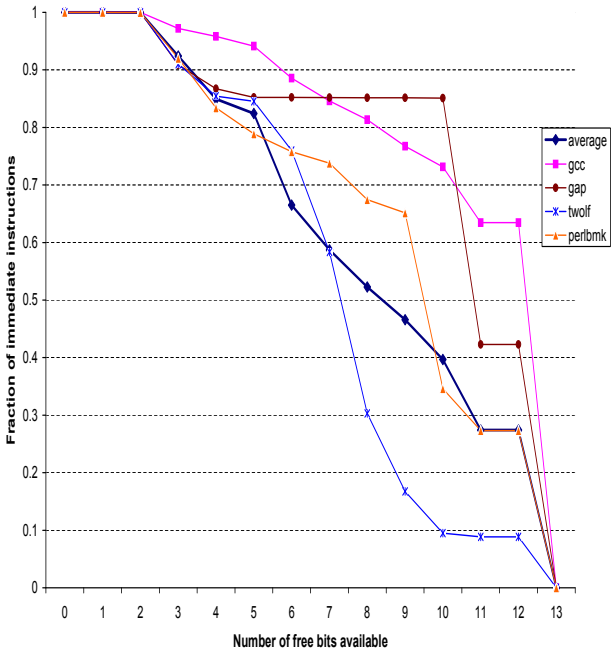

FIGURE 9.16: Free Space Available in Immediate LDUW (17.6%)

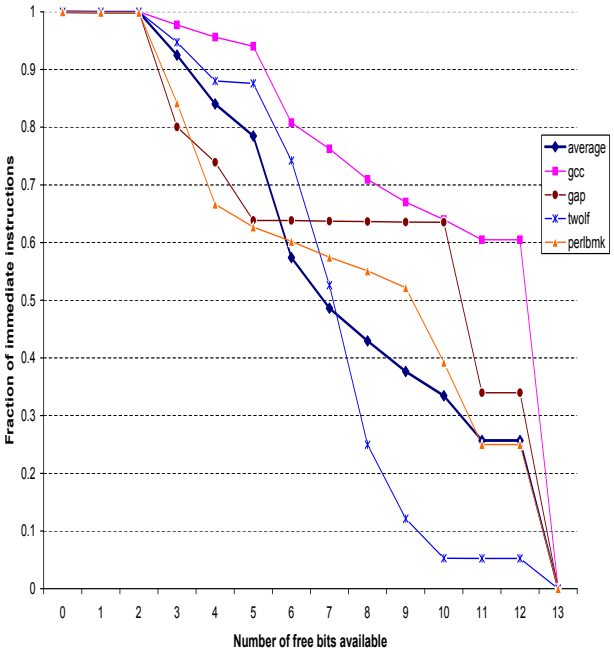

FIGURE 9.17: Free Space Available in Immediate STW (3.8%)

percentage of instructions drops significantly to less than 70% starting at 6 bits. In fact, all benchmarks seem to have the biggest drop starting at 6 bits. The perlbmk benchmark also has similarly large immediate fields. The STW instruction is similar to LDUW on average but the benchmark data differs. The benchmark twolf has smaller immediate fields for STW compared to LDUW while the benchmark gap is the opposite, with bigger immediate fields for STW compared to LDUW.

The SLL instruction is unique in that both immediate and non-immediate versions always have 7 free bits, with the instruction using just a 5 bit field for immediate instructions. This is illustrated in Figure 9.18, where 100% of immediate instructions have at least 7 bits free. An additional two bits can be taken from the immediate field for about 90% of instructions on average. This is the case for all of the individual benchmarks shown. There is not much variation in the per benchmark data for the benchmark shown. Some benchmarks which are not shown, such as gzip and mcf, see a bigger drop when we reach 9 bits, but these are benchmarks which generally do not benefit much from preload data.

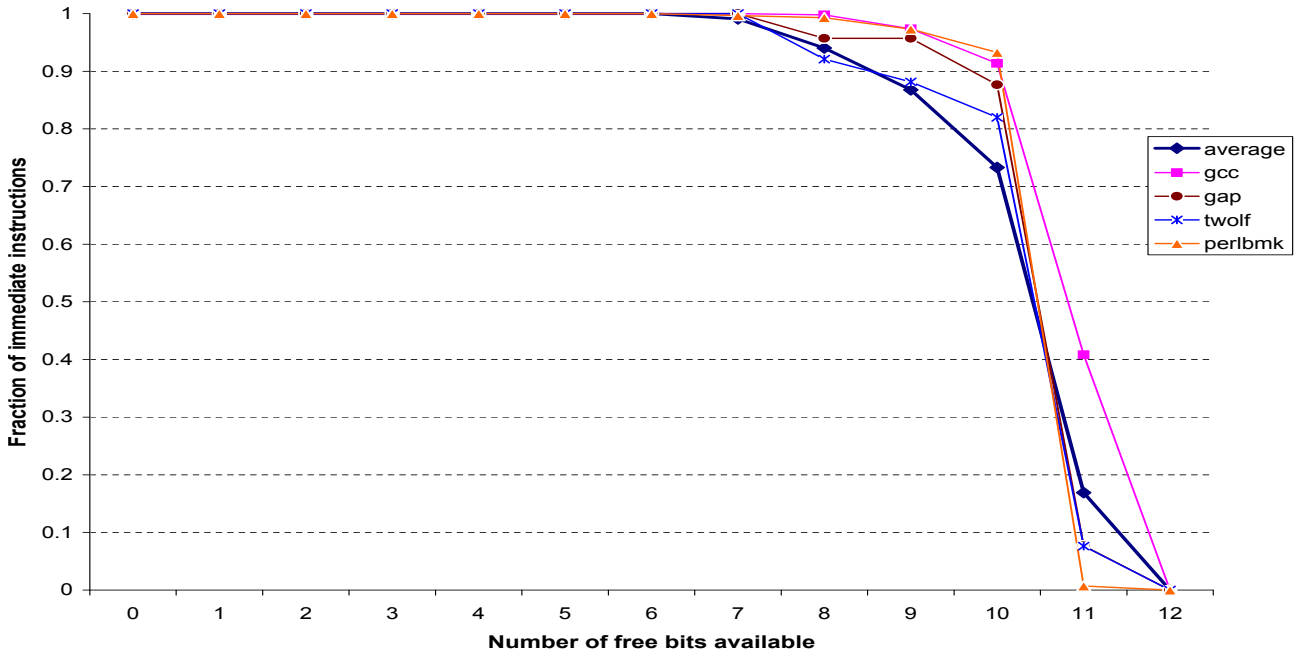

FIGURE 9.18: Free Space Available in Immediate SLL (6.1%)

TABLE 9.1: Carry Space Assigned by Instruction

Instruction	Carry Space Provided
BRANCH	10
ADD	4
SUBCC	10
LDUW	4
STW	4
SLL	9

9.4.2 Evaluating per Block Carry Space

In the decode stage, the hardware will be responsible for identifying the instructions discussed in the previous subsection (or another set of suitable instructions), extracting the carry bits and combining them to get the special configuration, bit index and tag bits. This would be done on a per block basis. This subsection evaluates the total number of carry bits that are available on a per block basis. To do this, we need to decide how many bits each of the instructions mentioned previously will give up. Based on the results in the previous subsection and Section 6.7 (for branches), Table 9.1 shows the number of bits that we select for each of these instructions.

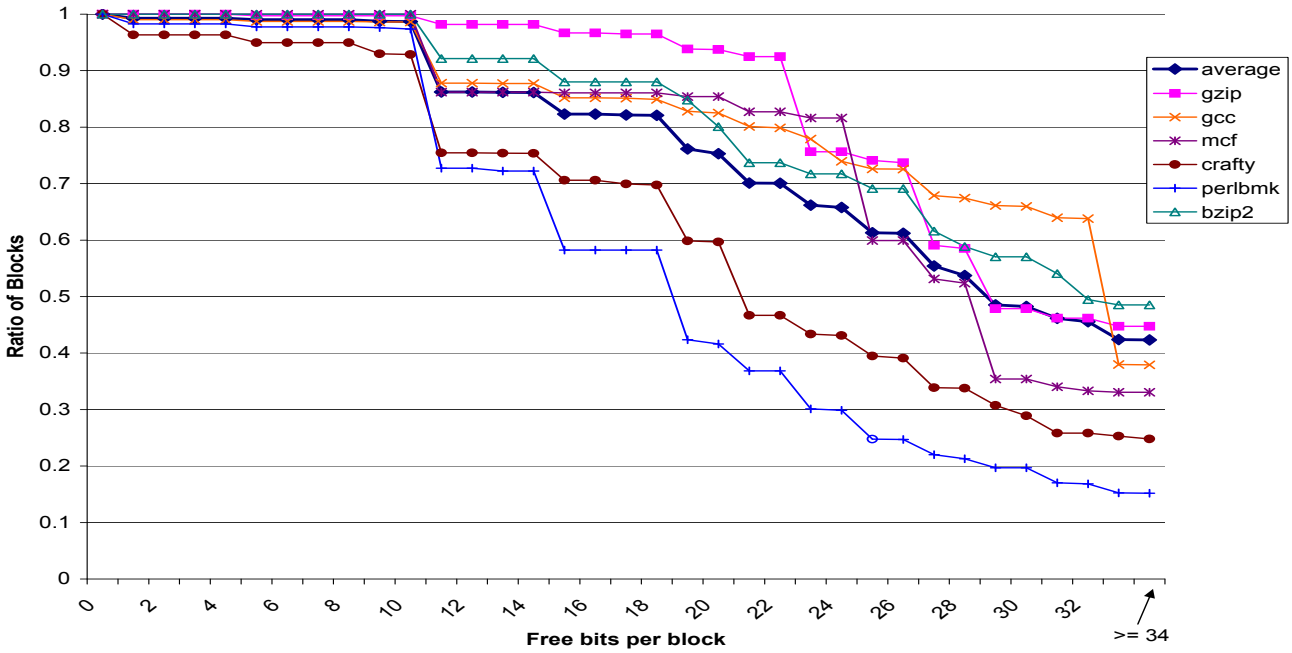

FIGURE 9.19: Carry Space Bit Availability in Blocks

The overall results are shown in Figure 9.19. We see that both on average and for all benchmarks shown, 10 bits can be found in a basic block for almost 100% of instructions. The average percentage of basic blocks that can hold 10 bits of carry space is is 98.7%. However, the number drops to 86.2% for 11 bits. We mentioned at the beginning of this Section that Spotlight needs between 18 and 22 bits of carry space to store a special configuration. The average number of blocks with 18 bits of space is 82% and the number of blocks with 22 bits of carry space is 70%. The per benchmark data (shown for a set of selected benchmarks) shows that for benchmarks like gzip, gcc, mcf and bzip2, the percentage of blocks which can give up these higher number of bits is greater than 80%. On the other hand, benchmarks such as crafty and perlbmk have a limited number of blocks which can give up these higher number of bits. The most negative result is for perlbmk, where less than 60% of blocks can give up 18 bits. The result show that the availability of space is not necessarily correlated to whether a benchmark is preload sensitive. The benchmarks gcc and bzip2 have a large amount of free space per block even though they are preload sensitive. This is also the case for twolf (not shown). On the other hand, benchmarks such as crafty, parser and perlbmk are

preload sensitive but have limited carry space availability per block. This is probably because not enough of the frequently used instructions in these benchmarks have carry versions.

9.4.3 Summary of Carry Space Availability

The results shown above are mixed and inconclusive. While some benchmarks have a large amount of blocks with high carry space, other Preload sensitive benchmarks have a limited amount of blocks with high carry space. There are some important points to be made.

Firstly, the amount of carry space that is calculated per block is dependent on the set of instructions selected and the amount of carry bits selected for each of these instructions. The per block results can vary significantly based on these choices. The parameter space is relatively large and our study above was far from exhaustive. For example, for ADD instructions, the benchmarks gcc and twolf have very small displacements while perlbmk and gap have larger displacements. Should we select 7 bits of carry space for ADD (see Figure 9.14) so that gcc and twolf can benefit, and so that when perlbmk and gap do find enough space in these instructions, they can get 7 or 8 bits of carry space? Or should we assign 3 bits of carry space to ADD, so that perlbmk and twolf have a higher probability of finding at least some carry space bits? There is no obvious answer to this question.

Secondly, it is not clear what percentage of blocks need to have enough carry space for preloading to be effective. If preload space is not available in some blocks but there are very few branches near that block which need carry space data, this is unlikely to affect performance. On the other hand, if preload space is not available in a part of the code that benefits significantly from preloading, this would affect accuracy in a negative way.

Finally, the amount of carry space available will be affected by the choice of instructions. The instructions we selected were based on how frequently they occur, and how much free space they may be able to give up. It is possible that different instructions will give better results. All of these questions merit further investigation.

9.4.4 Reducing the Size of Carried Data

Since Spotlight needs to preload a lot of carry data, it would be beneficial to reduce the amount of data required. This subsection discusses some ways this could be done.

One option is to develop an additional compact representation of the special configuration. A single bit could indicate that the preload data representation is compact. The goal would be to use the compact representation for some of the most common configurations. We saw in Figure 6.13 and Figure 6.15 that special configurations such as 0 GHR bits and very short GHR segments are common. If these are represented compactly, many branches would need fewer carry space bits, increasing the probability that they can find a block with enough carry space. Further, if some branches use special configurations that vary in accuracy only slightly from a compact configuration, those branches could use the compact representation instead.

The number of tag bits used also impacts performance. The tag bit representation can be reduced by evaluating whether a branch has some bits in common with the preload site. This is likely, because the tag bits are the next higher order bits after the BIT index. Generally, if a branch is within a reasonable distance from its preload site, there is a good possibility that the branch and the site have the higher order bits of their PC's in common. Therefore, an additional option is to reduce the number of tag bits that need to be carried for some branches. Huffman coding could also be used, but this would complicate the hardware.

9.5 Evaluating Agree Preplace Predictor

In Section 8.4, we introduced the Agree Preplace predictor which takes advantage of the fact that Agree only needs one bit of data per branch. This makes it possible to cover all possible branches that could occur d blocks away. We also introduced an improvement to the basic Agree predictor which selects the biasing bit based on the path taken to a branch instead of using a single bias for all branch instances. The advantage of both of these schemes is that they do not require a large number of configuration bits like Spotlight. It

also does not use a BIT and hence does not need to store BIT index or tag bits in the carry instruction. We evaluate the accuracy of both of these predictors by comparing them to the Ideal Agree predictor. This is similar to the NoBIT (Ideal) version of Spotlight used in previous Sections in that it assumes that profiled data (the biasing bit) is always available without the possibility of a BIT miss.

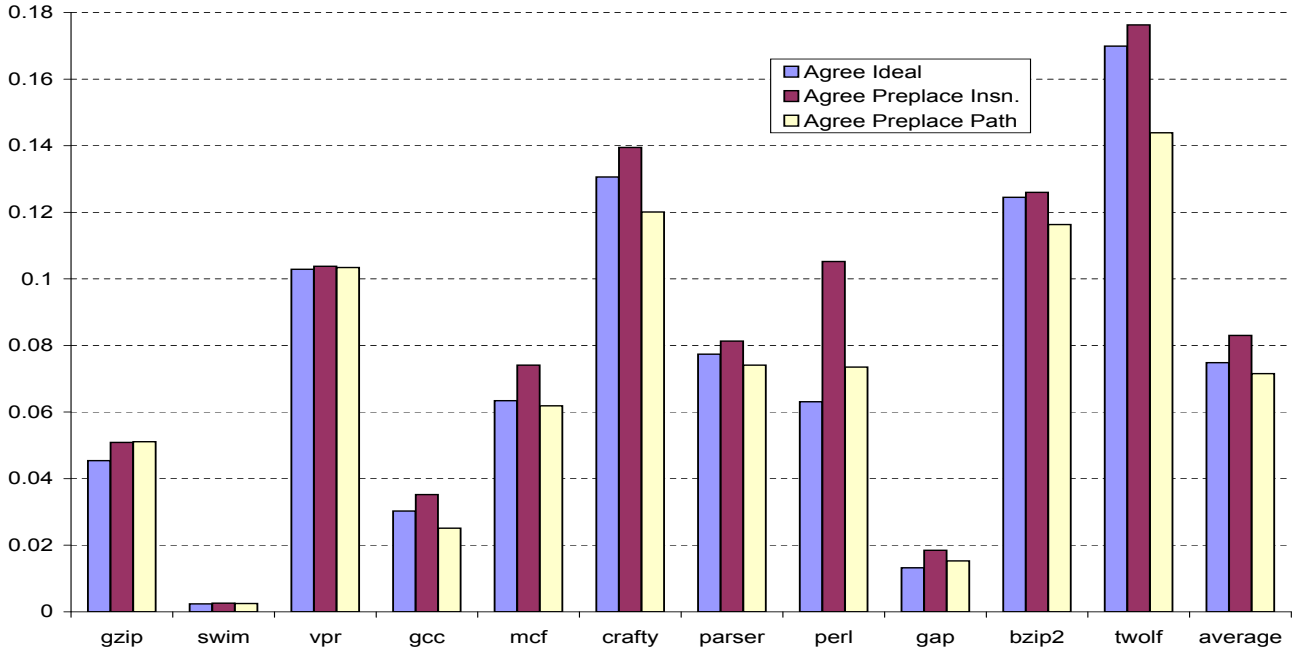

FIGURE 9.20: Accuracy of Agree Preplace for PHT Lg Sz 10

Figures 9.20 - 9.23 show the results for these three schemes for four PHT sizes. The *Agree Preplace Instruction* predictor uses one bias for all instances of a branch, while the *Agree Preplace Path* predictor selects the bias based on the path taken to a branch from d blocks back. From Figure 9.20, we see that the Preplace Instruction predictor is fairly close to Ideal in benchmarks such as swim, vpr, parser, bzip2 and twolf. In gap, gzip and gcc, the difference is significant in percentage terms but not in absolute terms (0.53, 0.57 and 0.62 respectively). The difference is more significant for mcf, crafty and perlbmk. Twolf does better with Preplace Instruction compared to Ideal. The data chart above is wrong, but I can't access the ECE systems to change it. The largest difference is in the benchmark perlbmk, which is not surprising since it has a high number of indirect jumps, something

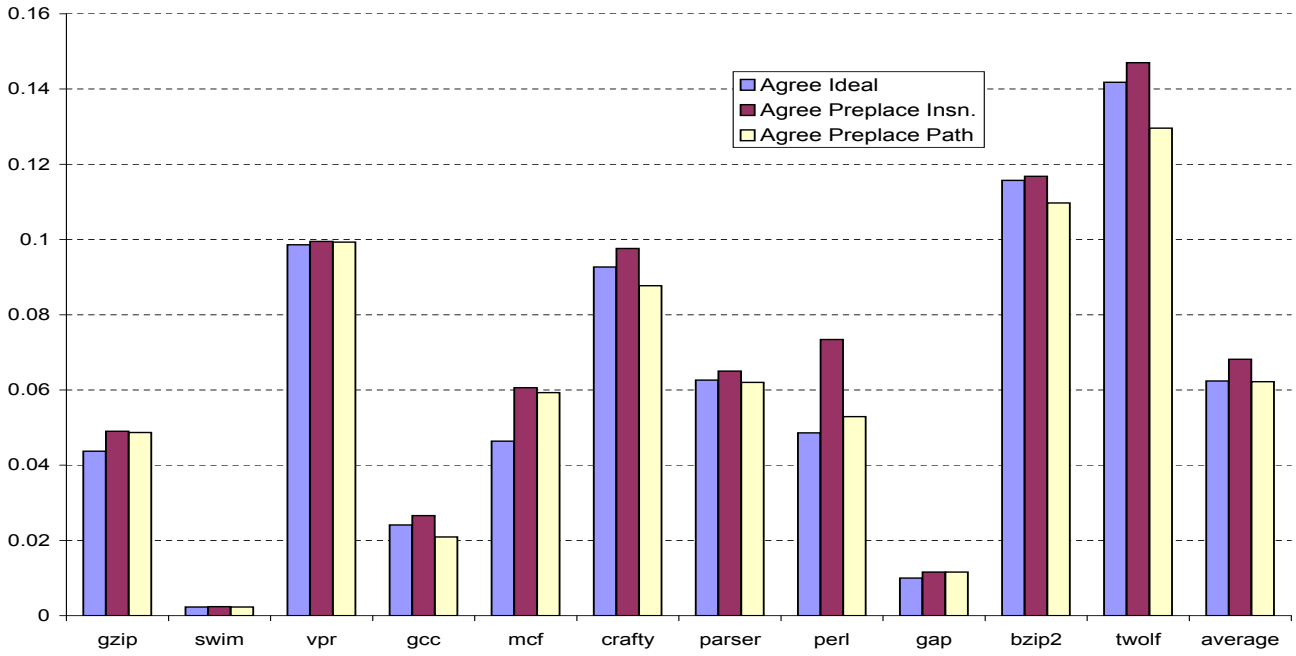

FIGURE 9.21: Accuracy of Agree Preplace for PHT Lg Size 12

that Agree Preplace is not designed to handle. As the size of the PHT increases as shown in Figure 9.21, the difference between the two schemes declines significantly for the benchmarks with the biggest difference. The exceptions are mcf and perlbmk, which still see a big drop between Ideal and Agree Preplace Instruction. In fact, as we increase the PHT size as shown in Figure 9.22 and Figure 9.23, the difference continues to decline for all benchmarks other than these two.

By selecting a bias based on the path to a branch, the Agree Preplace Path predictor has the potential to outperform the Ideal predictor, and the results indicate that this is exactly what happens. Even at the smallest PHT size, where Agree Preplace Instruction predictor fails to match the Ideal predictor, the Preplace Path predictor is able to outperform Ideal in all benchmarks other than benchmarks gzip, vpr, perlbmk and gap. The difference for vpr is almost trivial (0.05%) while the difference for gap is not significant (0.21%). While there is a difference for perlbmk of about 1.04%, it is much smaller than the difference between Ideal and Preplace Instruction. The only benchmark (from the set which saw a big drop between Ideal and Path Instruction) which does not benefit from Preplace Path

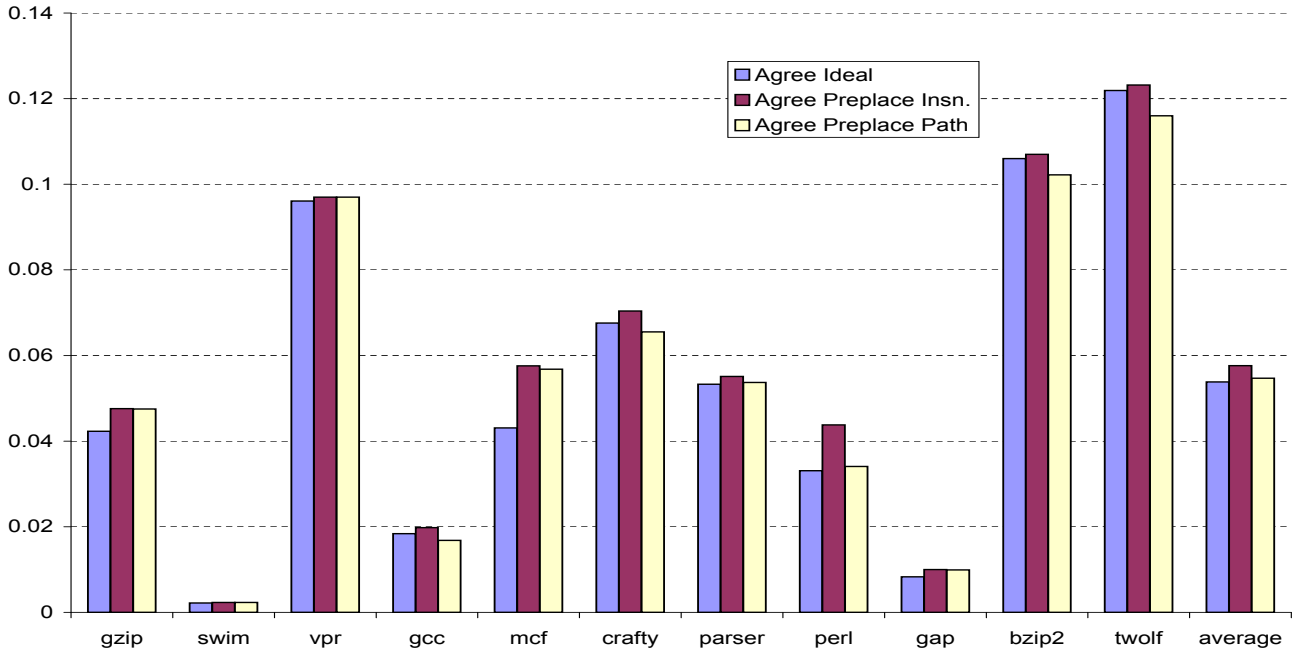

FIGURE 9.22: Accuracy of Agree Preplace for PHT Lg Size 14

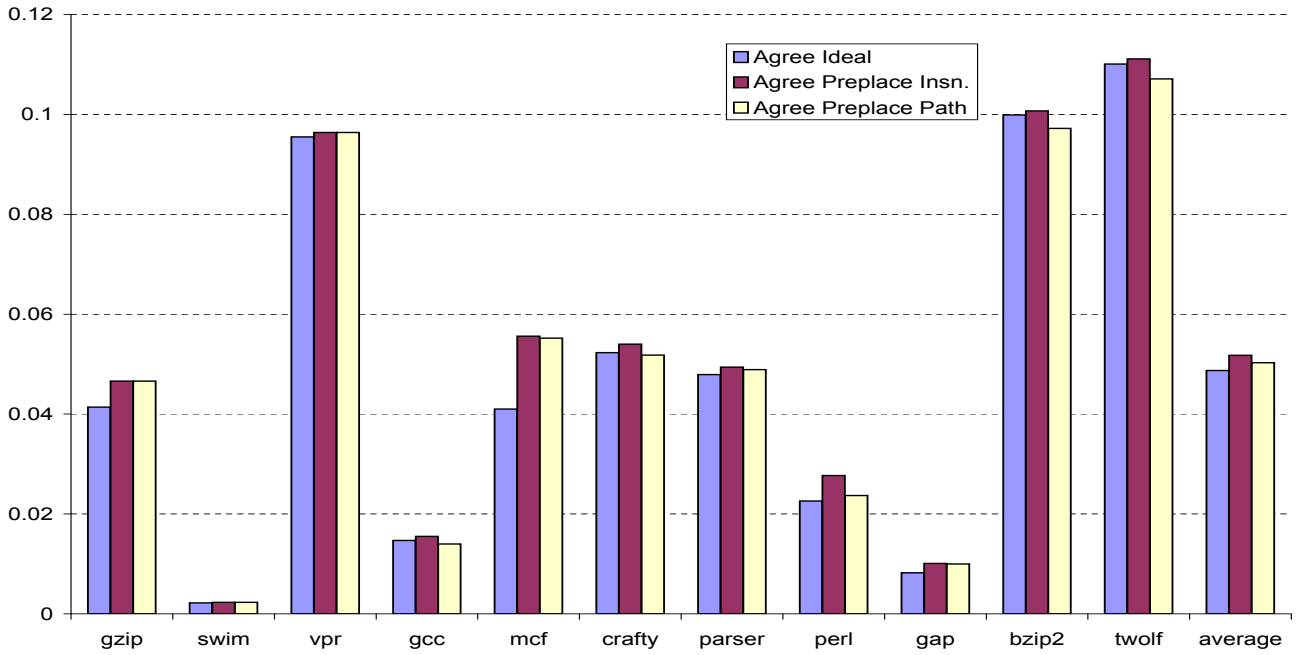

FIGURE 9.23: Accuracy of Agree Preplace for PHT Lg Size 16

is gzip, which sees a trivial decline compared to Preplace Instruction. The benchmarks gcc, mcf, crafty, parser, bzip2 and twolf all get better performance than Ideal, delivering on the potential of the Preplace Path predictor to improve over ideal. As the PHT size increases, the main observation is that both the gap between Preplace Path and Ideal declines, both for benchmarks where Preplace Path is better and for benchmarks where Ideal is better. The exception is mcf, which has better performance at PHT Size 10 but has worse performance at higher PHT sizes.

Chapter 10

Reducing Complexity of Non-Branch Predictor Front End Schemes with Preloading

This chapter discusses some of the other potential uses of preloading on the processor front end, other than the branch prediction schemes described so far. Preloading is a general technique that can deliver some type of statically chosen data to the front end of the processor by extracting it from instructions leading up to a block. The data for a block cannot be stored within the block itself because the front end of the processor is the first few pipeline stages before instruction decode. As such, on the front end, the only way to deliver statically chosen data in a low cost manner to a block is to Preload it in prior blocks leading up to the block.

10.1 Using Preloading to Potentially Improve Performance and Reduce Cost of the Basic Block Table

Figure 10.1 shows the part of the front end of a processor relevant to the basic block table. The block table is used to compute the address of the next block that needs to be fetched from the Instruction Cache. The table delivers the length of the next block, the CTI type and the address of a target. There are two possibilities based on whether a branch is not taken, a branch is taken or the block ends with a non-branch CTI. If a branch is not taken, the next block is computed as the sum of the start address of the current block and the length of the block provided by the block table. If a branch is taken, or if the branch is not a CTI, the target field is used as the start PC of the next block that needs to be fetched.

The same type of benefit from preloading branches are also available for the block predictor. When a block is seen for the first time, its data will not be available in the block table. Data will also not be available if a block's data is evicted due to another block. This can be avoided with preloading by placing the data for a block on the path to the block. By the time the block is reached, its data will have been preloaded, preventing the miss in the

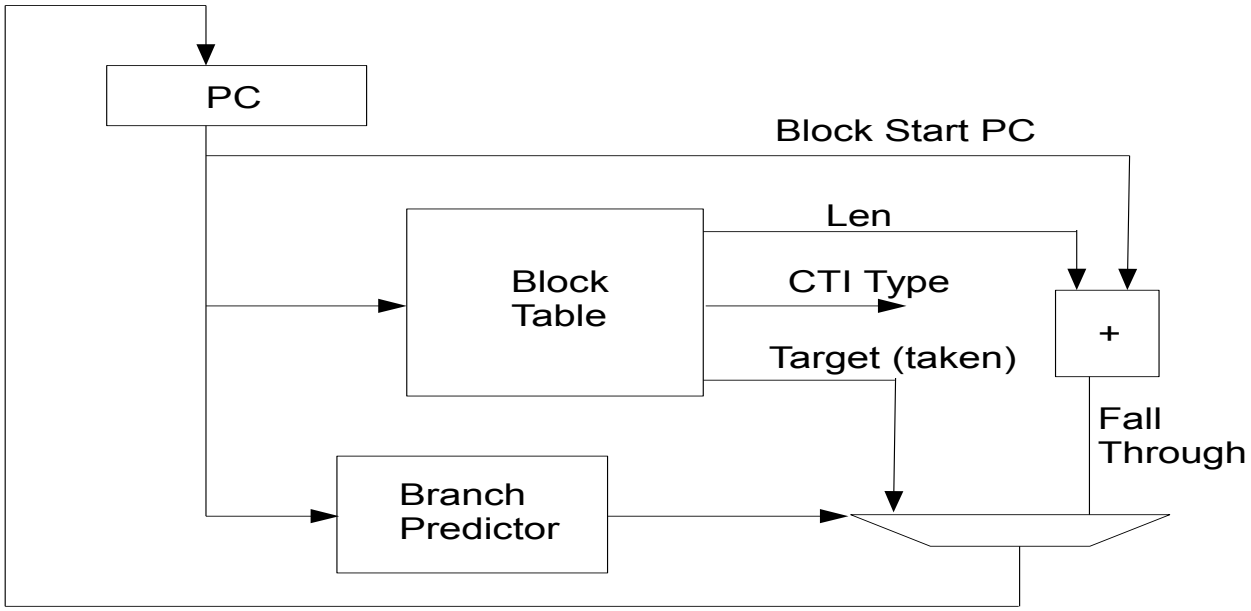

FIGURE 10.1: Front End with Block Table

block table. As is the case with the BIT discussed in previous chapters, preloading can also help reduce the size of the block table because the data will typically be used shortly after preloading it into the table. See chapter 7 for a complete discussion. This will provide the same benefit as instruction prefetch since it will be able to generate instruction addresses (though the usual prediction mechanism) for instructions that have never been cached and which are at targets of CTIs, not just fall-throughs, as sequential prefetch can find.

The number of carry bits needed for preloading the block data is manageable, although clearly it would be harder to manage if it is combined with a Preload based branch predictor (the carry space will have to be split between the two). The length of a block should not require more than 4 or 5 bits, because blocks are unlikely to have more than $2^4 = 16$ or $2^5 = 32$ instructions. The CTI type may be one or two bits. The target will match the size of the physical address of the system and has the potential to be very large. However, this can be handled by limiting the type of blocks we Preload and storing the offset to the target instead of the address of the target itself.

10.2 Using Preloading for Other Parts of the Front End

Preloading could be used for an indirect jump predictor. A preload analysis could determine the multiple targets of a jump and the jump-table indices used for each of them (generated by xoring the PC with a GHR). These values could then be pre-loaded into the jump table. In some ways that would provide the same benefit as the Agree predictor, by yielding a correct prediction for a CTI that has not yet been executed.

Chapter 11
Conclusions and Future Work

11.1 Contributions

With the increased importance of reduced power consumption and embedded systems, and the emergence of multi-core processors, it has become necessary to develop architectural techniques that can improve performance, but do without increasing costs, complexity and power consumption. This is counter to the trend in branch prediction research, which has been towards increasingly complex schemes which are both hard to implement and likely to consume a significant amount of chip area and a significant amount of energy. Yet, real processors continue to improve single core performance by improving the performance of the front end. This dissertation makes the following contributions aimed at reducing the complexity of the processor front end without reducing performance.

- **Spotlight - A profile based branch predictor.** We introduce a novel profile based branch predictor which gets high accuracy despite a simple hardware design. It does this by moving the complexity of the prediction process offline to a profiling step. The part of the history most correlated to a branch is determined by software instead of hardware. Because it is a profile based scheme, it is also able to combine the benefits of other predictors such as EHB and Agree and improve on their performance.

- **The Profiling, Analysis and Annotation (PAA) system.** We explain how statically chosen data will be collected and how analysis would be performed in a real system so that it is non-intrusive to the user. We discuss how annotated binaries can be generated that use special versions of existing instructions to store and carry stati-

cally chosen profiled (or other) data.

- **Evaluation of free space available in benchmarks for carry instructions.** We evaluate how much free space can be extracted from existing branch and other instructions, so that special versions can be created that can store carry bits obtained from profiling.

- **Techniques for delivering statically chosen data to hardware in a single cycle.** We discuss how a Benefiting / Branch Instruction Table (BIT), combined with an upstream placement hardware scheme can be used to deliver statically chosen data to the hardware in a single cycle, and evaluate the cost and accuracy of such a scheme.

- **Preloading - A low cost technique for delivering statically chosen data.** We introduce a novel technique, preloading, which places statically chosen data in basic blocks that lead to a branch or other benefiting instruction. By doing so, the size of the BIT used to store the data can be minimized without reducing performance. We demonstrate a special version of preloading for the Agree predictor (or other front end schemes which require one or two bits of data), which can be effective without using a BIT or additional hardware.

- **Other uses for preloading on the front end** We briefly discuss how preloading can potentially be used to reduce the complexity and improve the accuracy of other front end schemes such as basic block predictors and indirect jump predictors.

11.2 Future Work

- **Improving the accuracy of the baseline schemes with Preload data.** Fundamentally, preloading uses path information to place statically chosen data. The path information itself can be used to improve the accuracy of the baseline scheme. This was demonstrated one of the special versions of the Agree Predictor, Agree Preplace Path. The same could be done for Spotlight or other profile based predictors. The currently proposed scheme use the same configuration for all instances of a branch, regardless of the path taken.

- **Reducing the amount of carry bits needed.** The basic preloading scheme requires storing the BIT index and tag bits in carry instructions, increasing the amount of data that needs to be carried. There are some potential techniques which can reduce this data, such as a compact representation for the most common configurations and reducing the number of tag bits by finding common bits between the PC of the placement site and the benefiting instruction.

- **Evaluating preloading for other front end schemes.** We explained how preloading can be used for basic block predictors and suggested other uses for preloading on the front end. These schemes need to be implemented to evaluate their benefits in terms of cost and performance improvement.

- **Evaluating the energy benefits due to preloading.** Preloading improves performance by moving the complexity offline. We do not measure energy usage in this dissertation, but the energy dissipation of Preload based schemes such as Spotlight need to be compared to more complex branch predictor schemes such as the Hashed Perceptron.

Bibliography

[1] 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.

[2] Po-Yung Chang, Eric Hao, Tse-Yu Yeh, Yale N. Patt. Branch Classification: a new mechanism for improving branch predictor performance. *In Proceedings of the 27th Annual ACM/IEEE Int'l Symposium on Microarchitecture*, pages 22-31, 1994.

[3] Marius Evers, Sanjay J. Patel, Robert S. Chappell, Yale 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*, pages 52-61, June 1998.

[4] 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.

[5] Changpeng Fang, Steve Carr, Soner Onder, Zhenlin Wang. Feedback-directed memory disambiguation through store distance analysis. *In Proceedings of the 20th annual international conference on Supercomputing*, June 28-July 01, 2006.

[6] Wei Chung Hsu , Howard Chen , Pen Chung Yew , Dong-Yuan Chen. On the Predictability of Program Behavior Using Different Input Data Sets. *In Proceedings of the Sixth Annual Workshop on Interaction between Compilers and Computer Architectures*, p.45, February 03-03, 2002

[7] Intel Itanium Architecture Software Developer's Manual. *Volume 3: Instruction Set Reference, revision 2.1*, October 2002.

[8] T. Juan, S. Sanjeevan, J. Navarro. Dynamic History-Length Fitting: A Third Level of Adaptivity for Branch Prediction. *In Proceedings of the 25th Annual International Symposium on Computer Architecture*, 1998.

[9] Daniel A. Jiménez, Calvin Lin. Branch Path Re-Aliaising. *In Proceedings of the 4th Workshop on Feedback Directed and Dynamic Optimization (FDDO-4)*, December, 2001 (co-located with MICRO 34).

[10] Daniel A. Jiménez, Calvin Lin. Dynamic Branch Prediction with Perceptrons. *In Proceedings of the 7th Int'l Symposium on High Performance Computer Architecture*, pages 197–206, January 2001.

[11] Daniel A. Jiménez. Improved Latency and Accuracy for Neural Branch Prediction. *In Proceedings of the ACM Transactions on Computer Systems*, pages 197-218, May 2005.

[12] Daniel Jiménez. Fast path-based neural branch prediction. *In Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 243-252, December 2003.

[13] 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.

[14] Daniel A. Jiménez, Calvin Lin. Neural methods for Dynamic Branch Prediction. *ACM Transactions on Computer Systems*, 20(4):369-397, November 2002.

[15] J. Lau , S. Schoemackers , B. Calder. Structures for phase classification. *In Proceedings of the 2004 IEEE International Symposium on Performance Analysis of Systems and Software*, p.57-67, March 2004.

[16] 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.

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

[18] N. Metropolis , A. Rosenbluth, M. Rosenbluth, A. Teller and E. Teller (1953). Equation of state calculations by fast computing machines. The Journal of Chemical Physics, 21 (6), 1087-1092.

[19] Scott McFarling and John Hennessy. Reducing the cost of branches. *In Proceedings of the 13th Annual International Symposium on Computer Architecture*, May 1986.

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

[21] E. Perelman , G. Hamerly , B. Calder. Picking Statistically Valid and Early Simulation Points. *In Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques*, p.244, 2003

[22] Vijay S. Pai, Parthasarathy Ranganathan, and Sarita V. Adve. RSIM Reference Manual Version 1.0. Rice University Dept. of Electrical and Computer Engineering, Technical Report 9705, August 1997.

[23] John T. Strasser. Evaluation of branch prediction on processor energy consumption. *Masters Thesis, University of Illinois at Urbana-Champaign*, 2002.

[24] A. Seznec and A. Fraboulet. Effective ahead pipelining of the instruction address generator. *In Proceedings of the 30th Annual International Symposium on Computer Architecture*, pages 241-252, June 2003.

[25] Andrè Seznec. Analysis of the O-GEometric History Length Branch Predictor. *In Proceedings of the 32nd Annual International Symposium on Computer Architecture, pages 394-405*, June 2005.

[26] Andrè Seznec, Stephen Felix, Venkata Krishnan, Yiannakis Sazeides. Design Tradeoffs for the Alpha EV8 Conditional Branch Predictor. *In Proceedings of the 29th Annual International Symposium on Computer Architecture*, pages 296–306, Anchorage, AK, May 2002.

[27] A. Seznec. The O-gehl Branch Predictor. *In the first JILP Championship Branch Prediction Competition (CBP-1)*, 2004.

[28] A. Seznec. Revisiting the perceptron predictor. Technical Report PI-1620, IRISA Report, May 2004.

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

[30] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. *In Tenth International Comference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, October 2002.

[31] T. Sherwood , S. Sair , B. Calder. Phase tracking and prediction. *In Proceedings of the 30th Annual International Symposium on Computer Architecture*, June 2003.

[32] Eric Sprangle, Robert S. Chappell, Mitch Alsup, Yale N. Patt. The Agree Predictor: A mechanism for reducing branch history interference. *In Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 284-291, 1997.

[33] Jared Stark, Marius Evers, Yale 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.

[34] David Tarjan, Kevin Skadron. Merging Path and Gshare Indexing in Perceptron Branch Prediction. *In Proceedings of the ACM Transactions on Architecture and Code Optimization*, pages 280-300, September 2005.

[35] Maria-Dana Tarlescu, Kevin B. Theobald, Guang R. Gao. Elastic History Buffer: A Low-Cost Method to Improve Branch Prediction Accuracy. *In Proceedings of the 1997 International Conference on Computer Design*, October 1997.

[36] Renju Thomas, Manoj Franklin, Chris Wilkerson, Jared Stark. Improving Branch Prediction by Dynamic Dataflow-based Identification of Correlated Branches from a Large Global History. *In Proceedings of the 30th Annual International Symposium on Computer Architecture*, pages 314-323, June 2003.

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

[38] Spotlight - A Low Complexity Highly Accurate Profile-Based Branch Predictor. *In Proceedings of the International Performance, Computing and Communications Conference*, December 2009.

[39] L. Vintan and M. Iridon. Towards a High Performance Neural Branch Predictor. *In Proceedings of the 9th International Joint Conference on Neural Networks*, pages 868-873, July 1999.

[40] David W. Wall. Predicting Program Behavior Using Real or Estimated Profiles. *WRL Technical Note TN-18*, December 1990.

[41] David L. Weaver and Tom Germond. The SPARC Architecture Manual, Version 9. *Sparc International and PTR Prentice Hall, Englewood Cliffs, NJ*, 1994.

[42] Joe Wetzel, Ed Silha, Cathy May and Brad Frey. PowerPC User Instruction Set Architecture. *Book 1, version 2.01*, September 2003.

[43] Cliff Young, Michael D. Smith. Static Correlated Branch Prediction. *ACM Transactions on Programming Languages and Systems*, vol. 21, pp. 111-159, 1999.

Vita

Santhosh Verma was born in Karnataka, India, in 1981. He graduated with a Bachelor of Science in Electrical Engineering degree (dual majoring in computer engineering and electrical engineering) in August 2003. After graduation, he decided to pursue his passion for computer architecture by joining the graduate program in electrical engineering under Dr. David Koppelman. He joined the doctoral program in August 2004 and continued to work on research in computer architecture. During his advanced studies, he has also worked as a Research Assistant in computer architecture under Dr. Lu Peng and as a Research Assistant in compiler optimizations under Dr. J. (Ram) Ramanujam. He has also served as a Teaching Assistant for several courses in the Electrical and Computer Engineering Department. He also received an Master of Science in Electrical Engineering in December 2008.