Techniques for Improving Efficiency and Accuracy of Contemporary Dynamic Branch Predictors

A dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Science and Engineering

by

Ekkasit Tiamkaew

Dr. Angkul Kongmunvattana/Dissertation Advisor

April, 2005
ABSTRACT

Efficient branch predictions can significantly increase processor throughput by allowing an instruction stream to flow continuously without waiting for branch outcomes. Three major paradigms in branch prediction schemes, per-address two-level, global-history, and neural-learning branch predictors, are discussed in this dissertation. I have selected three most efficient branch predictors, one from each paradigm, and proposed robust mechanisms to help mitigate their possible shortcomings.

While per-address two-level branch predictors experience low branch interference, their implementation costs are too expensive. I propose a scalable per-address (SPA) predictor that leverages value locality in the history of branch outcomes to reduce the size of pattern history table (PHT) by about 50%, while maintaining high prediction accuracy. Moreover, alternative designs of SPA predictor are proposed to reduce internal conflicts within the PHT and improve its prediction accuracy. Experimental results reveal that a SPA predictor with 4-way set-associative PHT using 7 tag bits yields the highest prediction accuracy.

I propose and evaluate two alternate designs of YAGS (Yet Another Global Scheme) predictor, one of the most efficient global-history branch prediction schemes, to improve its prediction accuracy. The first method combines two direction PHTs to increases the overall space utilization. The second uses 3-bit counters in the choice PHT to keep branch biases more stable. Experimental results show that the merging benefits small branch predictors the most, making it suitable for embedded microprocessors. Meanwhile, the 3-bit-counter approach has the most impact on large branch predictors, which makes it suitable for superscalar microprocessors.
Finally, I propose to improve prediction accuracy of the O-GEHL (Optimized GEometric History Length) predictor, a neural-learning branch predictor that won in an international branch prediction contest. First method uses an adaptive adjustment of branch history lengths to increase space utilization in the first predictor table, consequently improving prediction accuracy in most cases, particularly in floating-point and integer benchmarks. Moreover, adding an extra predictor table using space saved from the sharing of hysteresis bits increases the accuracy of large branch predictors, implying that increasing the number of tables is more likely to benefit the predictor than increasing the number of table entries.
# Table of Contents

Abstract

List of Tables

List of Figures

1. Introduction
   1.1 Contributions .......................................................... 3
   1.2 Dissertation Organization .............................................. 5

2. Background Materials and Related Work ........................................ 6
   2.1 Bimodal Branch Predictors .............................................. 8
   2.2 Branch Target Buffer .................................................... 10
   2.3 Two-Level Branch Predictors ............................................ 12
      2.3.1 Traditional Two-Level Branch Predictors ...................... 12
      2.3.2 Gshare Predictor .................................................. 16
      2.3.3 Agree Predictor .................................................... 17
      2.3.4 Bi-Mode Predictor ................................................. 18
      2.3.5 YAGS Predictor .................................................... 20
   2.4 Hybrid Branch Predictors ............................................... 22
   2.5 Neural-Learning Branch Predictors .................................... 24
      2.5.1 Perceptron Predictor .............................................. 25
      2.5.2 O-GEHL Predictor .................................................. 26

3. Scalable Per-Address Branch Predictors (SPA) .................................. 29
### LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>SimpleScalar’s benchmark programs.</td>
<td>36</td>
</tr>
<tr>
<td>5.1</td>
<td>Space utilization in each O-GEHL predictor table.</td>
<td>57</td>
</tr>
<tr>
<td>5.2</td>
<td>The number of T0 entries used in the execution of each benchmark.</td>
<td>58</td>
</tr>
<tr>
<td>5.3</td>
<td>The effects on prediction accuracy when more tables are added into</td>
<td>61</td>
</tr>
<tr>
<td></td>
<td>the O-GEHL predictor.</td>
<td></td>
</tr>
<tr>
<td>5.4</td>
<td>Improvement in the prediction accuracy of each benchmark group</td>
<td>65</td>
</tr>
<tr>
<td></td>
<td>when dynamic L(n) adjustment is incorporated.</td>
<td></td>
</tr>
<tr>
<td>5.5</td>
<td>Improvement in the prediction accuracy across all hardware budgets</td>
<td>69</td>
</tr>
<tr>
<td></td>
<td>when T8 is added.</td>
<td></td>
</tr>
<tr>
<td>5.6</td>
<td>Percentages of the improved prediction accuracy for each</td>
<td>70</td>
</tr>
<tr>
<td></td>
<td>benchmark category, with 9 predictor tables in the O-GEHL</td>
<td></td>
</tr>
<tr>
<td></td>
<td>predictors with budgets larger than 64K bits.</td>
<td></td>
</tr>
</tbody>
</table>
LIST OF FIGURES

| Figure 2.1 | State diagrams of the 1-bit and 2-bit counters | 9 |
| Figure 2.2 | Branch target buffer. | 11 |
| Figure 2.3 | GAg branch predictor. | 14 |
| Figure 2.4 | GAp branch predictor. | 14 |
| Figure 2.5 | PAg branch predictor. | 14 |
| Figure 2.6 | PAp branch predictor. | 14 |
| Figure 2.7 | Gshare branch predictor. | 17 |
| Figure 2.8 | Agree branch predictor. | 18 |
| Figure 2.9 | Bi-mode branch predictor. | 19 |
| Figure 2.10 | YAGS branch predictor | 21 |
| Figure 2.11 | Tournament branch predictor. | 23 |
| Figure 2.12 | Perceptron branch predictor. | 26 |
| Figure 2.13 | O-GEHL branch predictor | 27 |
| Figure 3.1 | Aliasing percentages in PAp, PAg, and GAp. | 29 |
| Figure 3.2 | PAp and SPA predictors. | 31 |
| Figure 3.3 | Grouping of branch history patterns in SPA predictor. | 32 |
| Figure 3.4 | Diagram of SPA indexing operation. | 33 |
| Figure 3.5 | Percentages of pattern aliasing in SPA predictor. | 37 |
| Figure 3.6 | Degree of pattern aliasing in SPA predictor. | 38 |
| Figure 3.7 | Prediction accuracy of PAp and SPA branch predictors | 39 |
| Figure 3.8 | Prediction accuracy of SPA branch predictor with direct-mapped |
PHT and various tag-bit lengths.............................. 41

Figure 3.9 Prediction accuracy of SPA branch predictor with direct-mapped, 2-way, and 4-way set-associative PHTs ....................... 42

Figure 4.1 Unification of the two direction PHTs in YAGS predictor ........ 47

Figure 4.2 Prediction accuracy of YAGS with direction and exception PHTs across all hardware budgets ......................... 50

Figure 4.3 Prediction accuracy of 8K-bit YAGS with direction and exception PHTs under each benchmark ......................... 50

Figure 4.4 Prediction accuracy of YAGS with 2-bit and 3-bit counters across all hardware budgets................................. 52

Figure 4.5 Prediction accuracy of 1M-bit YAGS with 2-bit and 3-bit counters under each benchmark ......................... 53

Figure 5.1 Improvement in prediction accuracy when L(0) = 3 and L(1) = 5...... 60

Figure 5.2 Misprediction rates when the dynamic adjustment of L(n) is incorporated into the O-GEHL predictor......................... 63

Figure 5.3 Misprediction rates when 9 predictor tables with shared hysteresis bits are used in the O-GEHL predictor......................... 68
CHAPTER 1

INTRODUCTION

With an ability to fetch and issue multiple instructions per cycle becoming common in most modern microprocessors, a pool of independent instructions, a so-called instruction window, is inevitably emerging as another essential factor in boosting processor performance. A larger instruction window means that more independent instructions will be available for being selected and issued into functional units. Even though an instruction window size of 32 is common in current microprocessor design, there has been a study by Henry et al. [20], which allows as many as 128 instructions to be simultaneously in flight. Moreover, a high-end microprocessor, such as IBM PowerPC 970, can support up to 200 instructions in pipelines at the same time [42]. Unfortunately, these impressive numbers do not represent the realistic processor throughput, but rather the extent of attempts by micro-architects to search for instructions that are independent of one another and thus can be overlapped during the execution. This potential overlap, which is called Instruction Level Parallelism (ILP), is one of the key elements in improving computer performance because its presence allows for a more continuous flow of instructions through pipeline stages. Despite various techniques being proposed to increase ILP, microprocessors’ pipelines are still incapable of operating at full efficiency since the delays introduced by branch instructions can easily disrupt instruction flows and consequently create gaps between instructions in the pipelines. These gaps are also known as “pipeline bubbles”.
While branch instructions can be classified as either conditional or unconditional, only unconditional branches pose no threats to the continuity of program execution flow as their outcomes are always the same (i.e. taken). On the other hand, conditional branches present two possible paths to which the subsequent program execution could proceed, depending on branch outcomes. Therefore, this dissertation will simply use the term ‘branch’ to refer to the conditional one.

When a branch is not taken, the program continues its execution down the next instruction in a sequential fashion. Meanwhile, the program execution flow jumps to the instruction pointed to by the branch target address when a current branch is taken. Branches are usually from ‘if-then-else’ statements and ‘for’ as well as ‘while’ loops. The conditions of ‘if’ statements and ‘while’ or ‘for’ loops are tested prior to the decision on which instructions are to be executed next. Unfortunately, when branch outcomes are not known by the time the branches finish their execution, the program does not know for sure along which path should it continue its execution. Until then, subsequent instructions cannot be fetched, issued, or executed. Since in the SPEC200 integer benchmarks [15] a branch instruction can be found after every 7 to 8 regular instructions on average [32], branch instructions should arrive in the pipeline at every 2 cycles in a four-issue processor on average. At this rate, the branch outcome cannot be resolved in time and the pipeline efficiency can dramatically decline.

An effective way to solve this problem is to predict the outcome of branches in a timely manner using information gathered from past executions (e.g. past behaviors of a branch). The prediction represents a path along which subsequent instructions are speculatively fetched, issued, and executed. However, before the branch outcome is
resolved, the results produced by these speculative instructions must be retained within the pipeline and cannot overwrite memory or registers. If the prediction was correct, these values are written back, effectively increasing processor throughput. Otherwise, the pipeline is flushed and restarted, causing further delays.

### 1.1 Contributions

Since branch mispredictions are widely recognized as one of the major impediments to exploiting ILP, this dissertation proposes efficient mechanisms to improve the performance of existing contemporary branch prediction designs.

One of the traditional branch prediction schemes is a per-address two-level branch predictor, which contains two tables for keeping branch history and patterns of each branch instruction. With this method, different branches are unlikely to interfere with one another, and prediction accuracy is likely to increase, if the size of each predictor table is sufficiently large. I have proposed a scalable per-address (SPA) predictor that is capable of leveraging the existence of value locality in the history of branch outcomes to reduce the size of one of the predictor tables, the pattern history table (PHT), by about 50%. Experimental results on eight SPEC2000 benchmark programs reveal that SPA prediction accuracy is only slightly than a branch predictor with a full-size PHT [22]. Therefore, various alternative designs of SPA predictors have been evaluated to reduce interference in PHT as well as to improve prediction accuracy. Experimental results show that a SPA predictor with 4-way set-associative PHT using 7-tag bits yields the highest prediction accuracy [21].
There also exists another type of branch prediction scheme usually called “global branch predictors”, which uses tables to keep collective behaviors of all branches. These global branch predictors usually employ complicated mechanisms to compensate for their relatively small die space. YAGS (Yet Another Global Scheme) predictor is one of the most accurate global branch prediction schemes [9]. It requires two PHTs, called direction PHTs, to store exception instances of each branch outcome direction (i.e. taken or not taken), and one PHT, called choice PHT, to store branch biases. I propose two mechanisms to improve the prediction accuracy of YAGS predictor. The first proposed method combines two direction PHTs together to increases space utilization of the predictor tables. The second one proposes to use 3-bit counters in the choice PHT to inject higher stability to bias bits. Experimental results show that the merging technique benefits small branch predictors (8K – 64K bits) while the 3-bit-counter approach benefits large branch predictors (128K-1Mbits) [44].

Without their highly complex mechanism and long prediction latency problem, branch predictors using neural networks to learn and predict branch behaviors can probably be considered the best branch prediction scheme in term of prediction accuracy. It is therefore worthwhile to discuss about this particular type of branch predictors as a separate study case. I perform an extensive analysis on the O-GEHL (Optimized GEometric History Length) branch predictor [35], which, with its neural-learning mechanism, has officially proven to provide highest prediction accuracy under a distributed set of benchmarks in an international branch prediction contest, CBP-1 [3], and propose two alternate designs to further improve its prediction accuracy. The first scheme dynamically adjusts branch history lengths used in the predictor’s indexing
function, effectively increasing space utilization in one of the seven predictor tables and overall prediction accuracy except for branch predictors with budget of 8K-16K bits. Another scheme adds an additional table into the predictor without requiring extra space by means of sharing certain bits in table entries. This scheme is effective only in the predictor with budget of 128K-1M bits, suggesting that increasing the number of predictor tables is more likely to benefit large predictors than increasing the number of table entries [43].

1.2 Dissertation Organization

This dissertation is organized as follows. In chapter 2, basic background and related work on branch prediction techniques are described along with an in-depth explanation of two-level branch predictors, both traditional and modern, as well as hybrid branch predictors and neural learning branch predictors. Their limitations are also formally defined. Chapter 3 provides details about SPA predictor and its impact on branch prediction accuracy, while chapter 4 describes my proposed schemes to improve the accuracy of YAGS predictor, along with experimental results. Chapter 5 shows an in-depth analysis of O-GEHL predictor and discusses about proposed architectures to enhance the predictor’s efficiency, including some experimental results. Finally, conclusion and recommended future work are explained in chapter 6.
CHAPTER 2
BACKGROUND MATERIALS AND RELATED WORK

Stalling the execution for the branch results could prove unproductive and eventually put an upper-bound limitation on the overall performance. In order to ameliorate such problem, branch outcomes must be predicted before the branch conditions are determined, allowing the processor to speculatively fetch the next instructions in the predicted path. With these mechanisms, the throughput of the processor can be significantly increased if the predictions are sufficiently accurate. The performance of the processors, particularly those with high issue width, is susceptible even to a small change in the misprediction rate. For instance, a misprediction rate of 10% reduces performance by 55% in a 4-issue processor, causing a sharp decrease in the IPC (instructions per cycle) from 4 to 1.8 when assuming that a branch arrives every four instructions [5].

Branch predictions can be carried out either statically or dynamically. Static branch predictions usually involve examining branch behaviors from previous executions of the observed programs while carefully choosing an input set that truly reflects the actual data in real applications. Profile information is then collected to modify the branch prediction mechanisms and/or the compiler. Since most branches have been found to often exhibit a highly biased behavior, either toward taken or not taken, using branch statistics to statically predict the branch outcomes has achieved a considerable success.
Some of the static branch prediction schemes in early days use a compiler for presetting a ‘hint’ bit in each branch instruction to indicate the direction most likely to be taken by the corresponding branch [25]. Some use a technique called ‘Backward Taken Forward Not Taken’ (BTFNT), meaning that the backward branches are predicted taken while the forward ones not taken [25]. This is because a backward branch normally represents a ‘while’ or ‘for’ loop that always rolls back to re-execute the same set of instructions until a certain condition is met. By always predicting taken for backward branches, the predicted control flow is forced to go back and forth, resembling a genuine loop behavior. The prediction results are therefore guaranteed to be incorrect only once throughout the loop execution. Meanwhile, forward branches can be caused by ‘if’ instructions whose conditions usually indicate rare incidents during the program execution, and thus are always predicted not taken. Code optimizations have also been proposed for a compiler to adjust the instruction sets to favor the BTFNT scheme, increasing prediction accuracy even more.

Dynamic branch predictors use hardware-based mechanisms to gather branch statistics during program execution while at the same time adapting their internal states to match the branch patterns. Since these dynamic approaches are capable of continuously readjusting themselves for a program with any input data set, they can achieve higher prediction accuracy with a wider range of programs and data inputs, and higher scalability as well, than the static schemes. Most dynamic branch prediction schemes utilize a high degree of branch correlations, which exist between a current branch, branches in the past, and others within the vicinity, to predict branch outcomes in the decode stage. Some examples of early dynamic branch prediction schemes are [28], [39],
[45], and [47]. From this point onward, only the mechanisms of dynamic branch predictors will be discussed as they are the main subject of this dissertation.

### 2.1 Bimodal Branch Predictors

Early dynamic branch predictors simply keep the gathered branch statistics in a single table format. Such table is normally called a Branch History Table (BHT) and is indexed by the lower portions of the branch address. Each entry contains a single bit that indicates the most recent outcome of the corresponding branch. Based on the assumption that a branch is likely to repeat its last behavior, these bits are directly used as the prediction outcomes. The shortcoming of this one-bit scheme is that if a branch represents a loop, it is likely to be predicted incorrectly twice rather than once, one when the loop is exited and another when a loop is reentered. This can seriously affect the branch prediction accuracy and therefore performance, especially in loop-intensive programs such as most compression algorithms.

To correct such anomalies, a bimodal branch prediction scheme that uses a table of 2-bit entries has been proposed [39] with all the entries being implemented as saturating counters, meaning that their values cannot be increased (or decreased) beyond the maximum value, 3 (or minimum value, 0). Each counter is incremented by 1 when its indexing branch is taken and decremented by 1 otherwise. With the prediction outcome being derived from the most significant bit of each counter, only two successive mispredictions can change the prediction state, eliminating the misprediction problem associated with the loop instructions. Even though counters can be implemented using a different number of bits, it has been shown that branch prediction schemes with 2-bit
saturating counters deliver the best performance [30]. This has also been confirmed by studies from Chen et al. using a Markov model [6]. The state diagrams of both 1-bit and 2-bit counter schemes are shown in the figure below.

![State diagrams of the 1-bit and 2-bit counter schemes](image)

**Figure 2.1 State diagrams of the 1-bit and 2-bit counter schemes**

A two-bit counter table has also been implemented in other dynamic branch prediction schemes. It is used throughout this dissertation as well, so from this point onward the word ‘counter’ will simply refer to the two-bit counter with the state diagram shown in figure 2.1, unless indicated otherwise.

Apparently there are a few problems associated with storing branch patterns in a table. A large portion of die area needs to be allocated for the storage if all the branches are to be kept there. An indexing function must also be able to avoid the collisions of different branch addresses in the same table entries. Unfortunately, the cost of branch predictors is not unlimited and the ideal indexing mechanism is almost impossible to find. The need for swift predictions of the branch outcomes to reduce pipeline bubbles also
rules out the possibility of implementing highly complicated indexing functions. As a result, branch predictors are usually small and their indexing mechanisms are rather simple. This has led to incidents where useful information in some table entries is accessed and/or updated by multiple branch instructions. This phenomenon is commonly known as interference or aliasing.

Many studies have analyzed the effects of the aliasing in branch prediction table and identified it as a major source of branch mispredictions [4], [23], [27], [29], [34], [40], [41], [48]. However, not all aliasing has negative effects on branch predictors since there is a so-called ‘constructive aliasing’, which helps improve the branch prediction accuracy. This is mainly due to the positive correlations between branches. In the other word, the branches that cause interference exhibit similar behaviors. Nonetheless, it has been shown that the ‘destructive aliasing’ is consistently a dominant factor in branch prediction performance even while there are considerable occurrences of both types of aliasing [4], [40], [41], [48]. Attempting to reduce the number of occurrences of aliasing and its effects is therefore a main focus in many well-known branch prediction schemes [9], [23], [37], [40].

2.2 Branch Target Buffer

When a branch is predicted taken, a target address must be obtained before the next instruction fetch continues. This is achieved through a means of storing the branch target addresses in a cache, which is called branch target cache or branch target buffer (BTB). A branch address is compared to a set of instruction addresses in the first column of the BTB (or the tag field). If matched, the second column of the matching entry is used
to provide the target address. This process could be carried through at the same time the branch predictors are functioning, so that the target address will be available immediately after the prediction result is generated.

![Diagram of Branch Target Buffer](image)

**Figure 2.2 Branch Target Buffer**

Figure 2.2 shows the architecture of the BTB assuming that its entries are fully associative, which is exactly like the designs proposed in one of the first BTB papers [25]. Various alternative designs are possible as well and have been investigated in great detail by Perleberg and Smith [33]. They also examined several features, including the presence or absence of the tag field.

A major contribution of BTB is to provide an existing branch predictor within the same microprocessor with a destination address before the next fetch stage, in order to avoid possible pipeline bubbles. However, it is sometimes used in other studies as the only means to provide branch prediction outcomes (i.e. predicted not taken when a tag
miss is present, taken otherwise). This approach has not been adopted in any modern microprocessors because its prediction accuracy is rather limited.

2.3 Two-Level Branch Predictors

Since a third of branch outcomes exhibit non-repeating patterns that are highly unpredictable [12], simple dynamic branch prediction techniques, such as two-bit predictor, are inefficient. Several branch predictors that contain two tables for keeping both branch history and patterns have been proposed for higher branch prediction accuracy. These schemes use branch history in one table as an index to one of the entries in another table, hence the name “two-level branch predictor”. Therefore, a bimodal predictor in the previous section can be viewed as a one-level branch predictor.

2.3.1 Traditional Two-level Branch Predictors

Two-level branch prediction techniques can be generally categorized as global or per-address schemes depending on the number of entries in the branch prediction tables. The global scheme means that a table contains only one entry for all branches to share. On the contrary, the per-address scheme, sometimes called the local scheme, allows for multiple entries so that each of them can be occupied by an individual branch assuming the table is large enough. This thus explains the name ‘per-address’.

In every configuration, the first-level table is responsible for keeping the history of branch outcomes while the second-level one keeping the likely results of every possible pattern of the branch history, normally represented by a set of two-bit counters. For example, in a global scheme, if the first-level table contains n bits, the second-level
one will have a single row of $2^n$ counters. The first table is usually called BHR (Branch History Register) or BHT (Branch History Table) when it consists of a single or multiple entries, respectively. Meanwhile, the second-level table is called PHT (Pattern History Table).

Yeh and Patt applied the ideas of global and per-address schemes even further as the tables in both levels can exhibit different schemes. In their papers, several combinations of two-level branch prediction techniques have been extensively studied [45], [46] and subsequently named GAg, GAp, PAg, and PAp depending on the schemes applied to the prediction tables in both levels.

GAg (Global two-level Adaptive branch prediction using Global pattern history table) is the simplest version among them as both of its prediction tables (BHR and PHT) have only one entry (In PHT, it means there is only one set or one row of counters). Branch prediction is achieved by using the branch history in BHR as an index to pick a two-bit counter in the PHT entry. GAp (Global two-level Adaptive branch prediction using Per-address pattern history table), or a correlated branch predictor as named by Pan et al. [31], has a single-entry BHR but its PHT contains multiple sets of counters. The branch address (or the current program counter – PC) is used to select a row in the PHT and the value in BHR is later used as an index to one of the counters in the selected row.

PAg (Per-address two-level Adaptive branch prediction using Global pattern history table), on the other hand, has a BHT with multiple entries while having a single row of counters in the PHT. The branch prediction result is obtained using the value of the PC-indexed BHT’s entry to point to the counter in the PHT. Finally, PAp (Per-address two-level Adaptive branch prediction using Per-address pattern history table)
uses per-address scheme in both BHT and PHT, and therefore requires a very high implementation cost. In every design, the most significant bit of the counter is used as the prediction result.
With all branches sharing a single entry in each table, the aliasing rate in GAg is undoubtedly the highest among all configurations. However, it does not necessarily mean that GAg exhibit the lowest branch prediction accuracy because the aliasing is not always harmful. Studies by Sechrest et al. [34] reported that for the large instances of their selected benchmarks, approximately a fifth of the aliasing that occurred was caused by the same branch pattern, taken. The authors explained such behavior can be attributed to repeated executions of a tight loop and is identical in several branch instructions. The aliasing caused by this incident is thus non-destructive.

However, to allow for a high degree of branch correlations the branch history kept in the BHR must be sufficiently large. A study by Yeh and Patt has reported that, to achieve 97% prediction accuracy, GAg needs the BHR with the length of at least 18 bits while only 12 and 6 bits are sufficient for PAg and PAp, respectively [45]. This requirement is even more critical in wide-issue superscalar microprocessors. In a four-issue processor with 12 pipeline stages, longer branch history can reduce the misprediction rate from 10% to 5% [27], [45], [46], [47]. Unfortunately, longer BHR has a direct impact on the PHT size as the number of the corresponding counters must be exponentially increased. This condition inevitably limits the practical size of the GAg predictor since its performance is highly dependent on the BHR length.

Per-address branch prediction schemes attempt to get rid of the aliasing as it is one of the major factors that influences prediction accuracy of the popular two-level branch predictors (i.e., most of the modern microprocessors implemented a combination of two-level predictors). PAp, with its multiple BHT and PHT entries, apparently exhibits the least aliasing rate among them and, in the simulations performed by Yeh and Patt,
bears the highest performance when using the same BHR length [45]. Nevertheless, it has never been adopted in commercial processors due to its large circuit area requirement.

Even though the exponential increase of the PHT size when the BHR becomes larger unfortunately puts a restriction on its size, GAg is still an attractive choice for microprocessor designers due to its relatively low implementation cost compared to the other three schemes. GAp has also received a lot of attentions since it similarly utilizes the branch correlations in the global branch history and thus has similar advantages over the local scheme. As a result, more complicated mechanisms have been proposed as an addition to the global-history schemes (GAg and GAp) for higher prediction accuracy. These new techniques are discussed in the subsequent 4 subsections.

2.3.2 Gshare Predictor

With an observation that, in traditional two-level branch predictors, the usage of the PHT entries is not uniformly distributed, leading to unnecessarily excessive aliasing, a new index mechanism has been proposed to address this problem [27]. The idea is to exclusive-or the global branch history with the branch address before using it as an index to the PHT.

It has been claimed that this so-called ‘gshare’ scheme represented an efficient way for a more evenly distributed utilization of the PHT entries and led to superior prediction accuracy. Further studies by Sechrest et al., however, have demonstrated that only a little improvement is gained from using the gshare scheme alone [34].
2.3.3 Agree Predictor

Based on the idea that most branches are highly biased either into a taken or not taken direction, the agree predictor introduces a ‘bias’ bit to each of the BTB entries, hoping that a branch will exhibit its biased behavior during the first time it is entered into the BTB [40]. The corresponding counter in the PHT, now interpreted as ‘agree’ or ‘disagree’ instead of ‘taken’ or ‘not-taken’, provides the final prediction, which is the bias direction if it agrees or the opposite if it does not. The update mechanism for the PHT is very similar to that of the conventional two-level branch predictors, the counter being incremented if the branch outcome complies with the bias bit, decremented otherwise. When a new branch is introduced into an already occupied BTB entry, the tag address is updated and the bias bit reset.

The agree predictor can effectively reduce the branch misprediction rate based on the observation that the first instances of most branches normally reflect their actual biased behaviors. Since branches are likely to have their actual biases stored in the BTB,
most of the PHT entries should contain the ‘agree’ value and multiple branches sharing the same PHT entry should also update the counter toward the same direction (i.e. agreeing with the bias). An aliasing in this entry will not result in a misprediction and thus can be said to have converted into a neutral, or even constructive, one. However, the first instance of a branch sometimes contradicts with its bias and pollutes the PHT by turning its entry to disagree. There is also an aliasing caused by branch instances that do not comply with the bias. Furthermore, a prediction is not available if a branch is not found in the BTB.

![Diagram of Agree predictor](image)

**Figure 2.8 Agree predictor**

### 2.3.4 Bi-Mode Predictor

The bi-mode predictor [23] works in a different manner from the agree predictor as it uses, instead of the BTB, an array of counters, namely a choice PHT, to store the biases. With this approach, if the first instance of a branch does not comply with its bias,
the choice PHT’s counter with an erroneous bias value will be corrected only after a couple mispredictions, making it rather immune to the pollution problem present in the agree predictor. The bi-mode predictor uses these biases to categorize the branches into either ‘taken’ or ‘not-taken’ and store them into the respective direction PHTs, which are indexed by the exclusive-or between the branch address and BHR. As a consequence, the ‘taken’ direction PHT will provide the predictions only for branches that are biased to be taken while, similarly, the ‘not-taken’ direction PHT is responsible for the ‘not-taken’ ones. Since each direction PHT comprises only branches with the same bias, the aliasing that takes place is likely to be non-destructive, effectively reducing misprediction rate.

The update mechanism of the bi-mode predictor is a little more complicated than the agree predictor. Normally, the choice PHT always gets updated unless it contradicts the branch outcome and the chosen direction PHT provides the correct prediction. For the direction PHTs, only the one chosen by the choice PHT is updated.

![Bi-mode predictor diagram](image)

**Figure 2.9 Bi-mode predictor**
Even though the bi-mode predictor seems to have less problems than the agree predictor, there still is a hardware budget issue as its prediction table space must be, as in the proposed design, equally split for 3 PHTs. Furthermore, the aliasing caused by the instances that do not comply with the bias and those that do has not been properly handled.

2.3.5 YAGS Predictor

Eden and Mudge have proposed a new branch prediction scheme known as YAGS branch predictor [9] that uses a filter mechanism to help reduce the information stored in the direction PHTs. It keeps the biases, both taken and not-taken, in the choice PHT while the direction PHTs are reserved only for the instances that do not comply with the biases in the choice PHT. This mechanism significantly reduces the size of the die space required by the direction PHTs, allowing them to be smaller than the choice PHT. Additionally, since the biases and exception instances of branches are now separately stored, the amount of conflicts in all PHTs has also been reduced. A few tag bits, reportedly around 6-8 bits, have been added to each entry in the direction PHTs to identify the exception instances and help reduce the aliasing among multiple branches.

When a branch instruction is issued, the choice PHT is accessed using the branch address as an index. If the branch is predicted taken, the ‘not taken’ direction PHT is accessed, using as an index the exclusive-or output between the branch address and BHR, and its tag is compared to the branch address. In the case of a match, the prediction result will be given by the ‘not taken’ PHT. Otherwise, the choice PHT will provide the prediction. The same process is repeated when the branch is predicted not taken, except
that the ‘taken’ PHT will be used instead. In most circumstances where branches behave according to their biases, the choice PHT can sufficiently provide the predictions with high precision. The direction PHTs, on the other hand, are responsible to provide predictions for those special cases that contradict the biases.

![Diagram](image)

**Figure 2.10 YAGS predictor (2bc stands for 2-bit counters)**

When the prediction from the choice PHT is incorrect, the corresponding direction PHT is updated and the branch address is inserted into the tag area. If the choice PHT predicts correctly, the choice PHT will be updated only if the tag in the corresponding entry matches the branch address. The choice PHT is always updated.

Among other two-level branch predictors, YAGS predictor obviously is the one with a very high potential because of a couple reasons. First, the space requirement for its direction PHTs is considerably weakened, leaving more room for other improvements.
Second, since the direction PHTs are now responsible for keeping only the exception cases, the aliasing effects caused by instances that comply with the bias and those that do not have been ameliorated, which is something the previous two predictors cannot achieve.

However, the fact that both direction PHTs are of the equal size can hinder the branch predictor performance. If branches are highly biased into a single direction as stated in the assumption of the agree predictor, they must be likely to inhabit in one direction PHT more often than the other, and therefore should also be larger. With its current design, YAGS predictor is rather prone to the space utilization problem since one of the direction PHTs tends to be less crowded than the other. The over-utilized direction PHT is also more susceptible to the destructive aliasing. Consequently, a more efficient mechanism is needed to optimize the performance. This can be achieved by extensively exploring the design space, which is unfortunately impractical because individual benchmarks with particular data input sets work well in different configurations.

YAGS predictor also uses 2-bit counters in all 3 PHTs. That is sensible for the direction PHTs since there are studies that have confirmed the superiority of the 2-bit counter scheme. Nonetheless, the choice PHT deals with branch biases, which in their nature are very different from conventional dynamic branch directions. Further investigations and experiments in different approaches may prove useful.

2.4 Hybrid Branch Predictors

The idea of hybrid branch predictors has been proposed and studied [27], [37] since it is believed that a branch predictor performs well with a certain set of branches
while another predictor works better with different branches. A selection scheme, usually implemented in another prediction table called “choice predictor”, is then necessary for dynamically choosing a more accurate predictor for the current branch. During the execution, the choice predictor increments its counter if a certain predictor is correct, decrements otherwise. It has been shown that a predictor that combines 16 different schemes, using both global and local history information, achieves 1% improvement in branch prediction accuracy over the best one among the 16 schemes.

One of the hybrid branch predictors that have actually been implemented in a commercial microprocessor is the tournament predictor in Alpha 21264 [20]. This predictor consists of GAg and PAg predictors with another GAg as a choice predictor. Two GAg predictors use the same branch history as an index while the PAg has a 1024-entry 10-bit BHT and an array of 1024 3-bit counters.

![Figure 2.11 Tournament predictor](image-url)
Despite the claim that hybrid branch predictors achieve high prediction accuracy by effectively selecting a more accurate branch predictor for each branch, a study from Eden showed that a selection scheme in hybrid predictors actually is a different way of reducing the aliasing [10]. The author further pointed out a futility of several attempts to mix together different branch prediction schemes in search for superior performance.

In a bid to reduce the aliasing, the best performer is the one that can do the most within a fixed hardware budget. This is the main reason for the shortcoming of most traditional two-level and some hybrid branch predictors since the exponential growth of their PHT size indirectly serves as a barricade of their performance. With the number of counters in the PHT being limited, branch predictors cannot have a BHR long enough to allow an efficient exploitation of the correlations in global branch history. As a result, their performance is severely restricted.

The conclusion from the study by Eden also said that the aliasing reduction can improve the prediction accuracy only to a certain extent and that a radical change in research direction regarding branch prediction schemes is thus required [10].

### 2.5 Neural Learning Branch Predictors

Considering prediction accuracy alone, perhaps the best branch predictors are those using neural networks to learn and predict branch behaviors. High mechanism complexity and long prediction latency unfortunately make this kind of branch predictors impractical. However, one can argue that future microprocessor technology is likely to provide sufficiently large die space for highly complex architectures, and allow time-consuming operations to be performed without significant delays. It is therefore
worthwhile to discuss about this particular type of branch predictors as a separate study case.

### 2.5.1 Perceptron Predictor

A perceptron is a simple form of neural networks, designed to keep a series of neuron weights. Since a perceptron requires a rather small die space when compared to other variations of neural networks, it is effectively utilized by the perceptron predictor [19] for the calculation of prediction outcome.

The perceptron predictor comprises a table of perceptrons, each containing \((n+1)\) neural weights: \(w_0, w_1, w_2, \ldots, w_n\). When a branch instruction arrives, its address is used to index the perceptron table. Weights are read and finally used, along with global branch history, to compute the prediction outcome. While each neuron weight is represented as a signed number using a multiple-bit counter, all the bits in global branch history are represented as \(x_1, x_2, x_3, \ldots, x_n\), where each \(x_i\) is either 1 with a taken branch or -1 with a not taken one. The following equations show how to calculate the prediction.

\[
y = w_0 + \sum_{i=1}^{n} x_i w_i \quad \ldots(1)
\]
\[
\text{Prediction} = (y > 0) \quad \ldots(2)
\]

The training procedure begins when branch outcome is known, as shown in figure 2.12. Its algorithm is as follows. If \(y\) does not exceed the pre-identified threshold, each neuron weight is increased by 1 when the branch outcome matches the corresponding history bit, and decreased by 1 otherwise.
Figure 2.13 shows the mechanism of O-GEHL predictor, which is similar to the perceptron predictor with some exceptions. Instead of having all weights read from the same table entry like the perceptron predictor, weights from different table entries are selected, using different forms of indexing parameters. There are totally 8 predictor tables, T0-T7, each of which contains multiple entries of a multiple-bit signed counter. A counter from each table is read and then added together to generate prediction outcome, which is the sign of the final sum.

Indexing function of the O-GEHL predictor allows an efficient exploitation of very long branch history, while at the same time maintains the use for some short ones. Only the first table, T0, is always indexed by a branch address alone. Various combinations of branch address and global branch history are used to index the remaining 7 tables. Let L(i) be the length of global branch history used in the indexing function of
the ith table. L(0) and L(1) are initially set to 0 and 3, respectively. The values of other L(i) can be computed using the geometric series equation: L(i) = α\(i-1\) * L(1), where a value of α is \(\{L(M-1)/L(1)\}^{1/(M-2)}\) and M is the number of predictor tables.

Despite having physical space just enough to accommodate for 8 physical predictor tables, the O-GEHL predictor has set M to 11 and L(10) to 200 in order to exploit really large branch history. All of its 8 tables are initially programmed to use L(0) to L(7) as table indices. However, three selected predictor tables can adaptively change their branch history lengths when an aliasing ratio in T7 drops to a pre-specified value. These tables, which are T2, T4, and T6, can switch from using L(2), L(4), and L(6) to L(8), L(9), and L(10), respectively. They can also switch backward if the aliasing ratio reaches a pre-specified threshold. The rationale behind this dynamic history adjustment is that if T7, which is indexed by a long branch history, experiences a low aliasing rate, longer branch history should be used. Meanwhile, with a high aliasing rate on T7, short branch history should be preferred. All of these threshold values are empirically selected.

![Figure 2.13 O-GEHL predictor](image-url)
In the 64K-bit O-GEHL predictor, the number of entries in each predictor table is 2K, with the exception of T1, which has only 1K entries. Each entry in T0 and T1 is represented by a 5-bit counter, while an entry in the remaining tables uses a 4-bit counter. These parameters have been empirically selected and therefore proven to optimize the performance of the O-GEHL predictor.

The training method in the O-GEHL predictor is very similar to that in the perceptron predictor. Each counter is increased by 1 if the prediction is correct, decreased by 1 otherwise.
CHAPTER 3

SCALABLE PER-ADDRESS BRANCH PREDICTOR (SPA)

Interference from sharing a single entry of branch history table (BHT) or pattern history table (PHT) among multiple branch addresses is one of the major factors that influences prediction accuracy of popular two-level branch predictors (i.e., most of the modern microprocessors implemented a combination of two-level predictors). According to the previous chapter, a per-address two-level predictor, known as PAp, is the least sensitive to interference since it has multiple BHT and PHT entries. Figure 3.1 below shows the percentages of interference in PAp, GAp, and PAg predictors under eight SPEC2000 benchmark programs. Each of these predictors has a budget of 8KB [22].

![Figure 3.1 Aliasing percentages in PAp, PAg, and GAp](image)

The figure indicates that, among other per-address predictors, PAp is the one most insensitive to the aliasing. However, instead of adopting the PAp organization, most of the proposals in branch prediction schemes still choose to work on other variations of
two-level branch predictors and attempt to reduce high interference among different branches [4, 23, 24, and 29]. This is mainly because of PAp’s excessive space requirement. An obvious solution to this problem is to reduce the die size required by the PAp’s PHT while trying to minimize the effects to its prediction accuracy.

In this chapter, I propose a scalable per-address two-level branch predictor called SPA that decreases the cost of PAp implementation by almost 50%. It exploits value locality of branch outcomes kept in each entry of BHT under PAp configuration. Specifically, SPA groups all branch history outcomes that are very unlikely to happen at the same time into multiple pairs and makes them share the same set of 2-bit counters in PHT, effectively folding PHT by half. It is apparent that SPA predictor reserves die space in a much higher scale than some die area saving methods in [7] and [13], which save only a small number of tag bits.

While both SPA and PAp are equally immune to interference since the number of BHT and PHT entries in SPA remains the same as PAp, the reduction of PHT size in SPA may lead to internal conflicts within each PHT entry. However, lower branch prediction accuracy caused by these conflicts within PHT is well compensated by this implementation cost saving.

3.1 Mechanisms in SPA Predictor

The structures of SPA predictor, along with PAp predictor, are shown below in figure 3.2 for comparison.
Figure 3.2 PAp (left) and SPA (right) predictors

In the above figure, address of branch instruction, \( b \), is used for indexing BHT, which keeps the outcomes of previous \( n \) branch instructions in each of its entries. For PAp, there is the same number of rows in PHT as that in BHT since both of them are indexed by the same branch address \( b \). Each row of PHT contains \( 2^n \) two-bit counters indexed by history of branch outcomes kept in BHT. Each of these two-bit counters maintains prediction status of a particular branch address. Since most branch instructions cannot possibly exhibit all possible patterns, a large number of two-bit counters in PHT of PAp predictor will not be used.

By carefully grouping the patterns of branch outcomes in BHT together, I can assign a set of two-bit counters to a pair of branch history patterns that are unlikely to happen at the same time, cutting the die space for PHT by almost 50% (i.e., from \( 2^n \) rows
in PAp to $2^{(n-1)}$ rows in SPA). Since most conditional branch instructions tend to be either frequently taken or rarely taken [1], I further assume that branch instructions with high percentages of being taken are highly unlikely to become the rarely taken ones, and vice versa, during the execution time. Based on this assumption, only half of PHT is sufficient to keep the counters for all likely patterns (either mostly taken or mostly not taken) of the corresponding branch addresses. Therefore, I fold each column of PHT in half, making two distinct sets of branch history patterns to share the same set of counters in the corresponding PHT column.

![Figure 3.3 Grouping of branch history patterns in SPA predictor](image)

The example in figure 3.3 is given to illustrate the grouping technique used in SPA predictor. Assuming that BHT keeps the history of the past 4 branch instructions (i.e. $n = 4$), for normal PAp predictor, each column in the PHT consists of $2^4 = 16$ two-bit counters for all possible branch history patterns, which are 0000, 0001, 0010, …, 1110,
and 1111. For SPA predictor, the number of counters in each column will be reduced to 8. From figure 3.3, one group of branch history patterns contains branches that are frequently not-taken (0000, 0001, 0010, 0011, 0100, 0101, 0110, and 1000), whereas another group contains branches that are frequently taken (1111, 1110, 1101, 1100, 1011, 1010, 1001, and 0111). If history pattern of a branch instruction belongs to one of these sets, it is unlikely to change its pattern to another set. There is also a chance that branch instructions may exhibit both patterns alternately during execution, resulting in each pattern interfering with one another in the smaller PHT of SPA predictor. This new form of internal interference called “pattern aliasing” is caused by the sharing of two-bit counters by two different branch history patterns in SPA. However, the high degree of value locality in branch history pattern makes this type of aliasing relatively rare during execution time.

---

**Figure 3.4 Diagram of SPA indexing operations**
While SPA reduces the size of PHT, it may increases complexity of the branch prediction stage in instruction execution pipeline. To overcome this problem, I only need to add an exclusive-OR operation into PHT access. The most significant bit of selected BHR is duplicated and exclusive-ORed with the rest of BHR bits. The result is then used for accessing PHT. An example in Figure 3.4 assumes that each entry of BHT has 4 bits and that the value of selected BHR is 1110. The most significant bit of this BHR is copied into 111 and exclusive-OR with the rest of BHR bits, 110, resulted in an index 001 for accessing PHT. The simplicity of this indexing and PHT access guarantees a short branch prediction time.

3.2 Exploring Design Space of SPA Predictor

While the SPA predictor reduces the cost of implementing per-address branch predictor significantly, it suffers from internal interference, which causes a slight drop in prediction accuracy of the predictor. To alleviate this problem, I have also proposed a replacement of direct-mapped PHT in the original SPA predictor with set-associative PHT using various lengths of tag bits. Direct-mapped PHT is simple to implement, offers high-speed access time, and requires zero or a few tag bits, but suffers from low utilization, whereas set-associative PHT is more complex to implement, requires longer access time, and uses more tag bits, but often yields higher utilization. Set-associative also reduces the chance of destructive aliasing, resulting in higher branch prediction accuracy. This is a tradeoff between prediction accuracy and implementation cost as higher degree of set-associativity translates to a larger number of tag bits needed in the PHT.
However, Fagin [13] discovered that, for a 512-entry branch target buffer (BTB), each entry needs to keep only 2 tag bits to obtain 99.9% of the accuracy obtained by keeping a full tag (9 bits). Chen, Lee, Postiff, and Mudge [7] also experimented with direct-mapped tagless branch prediction tables without flushing mechanism. They found that the tagless implementation has many advantages over the tagged one, including faster access time, lower power consumption, and simpler implementation. Based on these studies, I have evaluated the performance of SPA predictors when using PHT with various partial tag-bit lengths, in an attempt to limit the cost of implementing set-associative PHT within a reasonable margin while still maintaining or even improving the prediction accuracy of set-associative SPA predictors. To minimize an increase in PHT access time or delay, the maximum degree of associativity in PHT is limited to 4-way.

### 3.3 Methodology

I performed all execution-driven simulations using a modified version of ‘sim-bpred’ in the SimpleScalar Toolset [2]. Table 3.1 lists five integer and three floating-point programs from SPEC2000 benchmark suite that I used in my experiments along with their inputs and parameters as well as number of instruction executed.

In all simulations, both BHT and PHT in the SPA predictor are configured to contain 1024 entries (capable of simultaneously supporting 1024 branch addresses). Since the BHT can keep history of the past 10 branch outcomes, the PHT must contain 210/2 = 512 two-bit counters in each branch address. The implementation cost of this SPA configuration is, thus, only 1024x10 + 512x1024x2 = 1034K bits while the cost of
PAp with the same configuration is 2058K bits. Hence, SPA predictor saves the implementation cost by 1024K bits or 49.76% in this configuration.

<table>
<thead>
<tr>
<th>Program</th>
<th>Input</th>
<th>Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>ammp</td>
<td>ref</td>
<td>2 billions</td>
</tr>
<tr>
<td>art</td>
<td>ref</td>
<td>2 billions</td>
</tr>
<tr>
<td>wupwise</td>
<td>ref</td>
<td>2 billions</td>
</tr>
<tr>
<td>gcc</td>
<td>expr</td>
<td>9 billions</td>
</tr>
<tr>
<td>parser</td>
<td>ref</td>
<td>2 billions</td>
</tr>
<tr>
<td>twolf</td>
<td>ref</td>
<td>2 billions</td>
</tr>
<tr>
<td>vortex</td>
<td>bendian2</td>
<td>2 billions</td>
</tr>
<tr>
<td>Vpr</td>
<td>net, arch</td>
<td>2 billions</td>
</tr>
</tbody>
</table>

Table 3.1 Benchmark programs

3.4 Experimental Results

I measured the percentages of pattern aliasing in PHT of SPA predictor to verify that branch instructions with high percentages of being taken are highly unlikely to become the rarely taken ones, and vice versa, during the execution time. The results are shown in figure 3.5.

The Figure shows the percentages of pattern aliasing in all two-bit counters of PHT used during the execution. With the exception of gcc, almost all of the benchmark programs exhibit pattern aliasing between 13% and 28%. These values obviously support my assumption about redundancies in PHT.
Since the effects of pattern aliasing on benchmark programs are varied, I adopt the term “degree of pattern aliasing” as an indicator on how much it affects a particular benchmark program. In particular, branch instructions that exhibit one set of history patterns as much as another have the degree of pattern aliasing of 100%. Branch instructions that have their history patterns belonging to only one set have 0% degree of pattern aliasing. If a branch instruction appears to have its history pattern occurring $n$ times in one set and $m$ times in another set, its degree of pattern aliasing is calculated by the following equation:

\[ \text{Degree of pattern aliasing} = \left[\frac{\min(n,m)}{\max(n,m)}\right] \times 100\% \]

Figure 3.6 depicts the results from measuring degree of pattern aliasing under each benchmark. With none of the benchmarks having the degree of pattern aliasing more
than 50%, it shows that there is always one dominating pattern throughout the execution time. Consequently, it demonstrated that the pattern aliasing has very little negative effect on the prediction accuracy of SPA predictor.

![Figure 3.6 Degree of pattern aliasing in SPA predictor](image)

While there are pattern aliasing in all benchmarks, it is very unlikely that branch behaviors will change frequently during the execution time. On the other hand, it is highly possible that these two sets of branch history patterns occur in different period of execution time as branch instruction switches from one branch behavior to another. When this particular phenomenon occurs, the existing pattern aliasing is considered “soft pattern aliasing” as it has less negative effects on the branch prediction accuracy.

For example, consider a branch that exhibits a pattern of being taken three times and not taken once. The possible history patterns of this behavior are 1110, 1101, 1011, and 0111. Each of these history patterns point to a different two-bit counter in the corresponding PHT column. When the branch changes its behavior to taken twice and not
taken twice, its history patterns changes to 1100, 1001, and 0011. All of these history patterns points to different counters from the previous patterns. Although the real pattern aliasing does not occur in this case, the performance still degrades due to the time it takes for the counters to re-train during the change of branch history patterns. This change in branch behavior should also have similar negative effects on other two-level branch mechanisms. Thus, in many cases, SPA predictor is expected to give the same performance as PAp, despite having used only 50% of PHT die space.

3.4.1 Direct-Mapped SPA Predictor

I compare branch prediction accuracy of the proposed SPA predictor with that of the original PAp predictor. The results are shown in figure 3.7.

![Figure 3.7 Prediction accuracy of PAp and SPA branch predictors](image)

Figure 3.7 Prediction accuracy of PAp and SPA branch predictors

From Figure 3.7, two benchmark programs, parser and twolf, suffer the highest loss in branch prediction accuracy (i.e., up to 1.39%). However, the average loss of branch prediction accuracy of all benchmarks is relatively low (i.e., 0.73%). In particular,
branch prediction accuracy of the floating-point benchmarks suffers an average loss of accuracy by about 0.23%, whereas the integer benchmarks suffer 1.03% loss on average. This is because the integer benchmarks contain many if-then-else instructions that resulted in branch instructions with high possibility of changing their behaviors during the execution time, leading to high pattern aliasing rate. This problem also occurred in floating-point benchmarks, but with much less intensity. For example, wupwise does not experience any loss in branch prediction accuracy at all, and SPA prediction accuracy in art is very close to PAp predictor.

### 3.4.2 Associative SPA Predictor

I performed an evaluation of a SPA predictor with direct-mapped PHT using various lengths of tag bits. The results are depicted in Figure 3.8, which shows the differences in branch prediction accuracy of SPA predictor with direct-mapped PHT using 3 to 11 tag bits in comparison to SPA predictor with no-tag direct-mapped PHT. Results from three programs (i.e., art, parser, and wupwise) were omitted because prediction accuracy remains constant across all PHT configurations.

From Figure 3.8, it is apparent that adding tag bits to direct-mapped PHT has very little positive impact on SPA prediction accuracy (i.e., less than 0.5% improvement), and in some cases, significantly reduces its prediction accuracy. Specifically, a small increase in SPA prediction accuracy of three applications (i.e., ammp, gcc, and twolf) can be attributed to a reduction of destructive conflict aliasing, whereas a substantial decrease in SPA prediction accuracy of vortex and vpr programs can be attributed to a reduction of constructive conflict aliasing in PHT. Since adding tag bits to direct-mapped PHT hardly
increases SPA prediction accuracy, I decided to use the original implementation (i.e., SPA predictor with no-tag direct-mapped PHT) as a baseline for comparison with SPA predictor that adopted set-associative PHT.

![Figure 3.8 Prediction accuracy of SPA with direct-mapped PHT using various tag-bit lengths](image)

Figure 3.8 Prediction accuracy of SPA with direct-mapped PHT using various tag-bit lengths

Figure 3.9 (on the next page) shows branch prediction accuracy of a SPA branch predictor with different PHT configurations, ranging from no-tag direct-mapped PHT to 4-way set-associative PHT with full-tag. Noted that although there is horizontal straight line that represents prediction accuracy of SPA with no-tag direct-mapped PHT in every figure, going across multiple tag-bit lengths, these results are actually representing prediction accuracy of SPA predictor with no tag-bit only.

The results in figure 3.9 demonstrate that a SPA predictor with 4-way set-associative PHT and sufficient tag bits yields the best branch prediction accuracy in every benchmark program. For both vortex and twolf, prediction accuracy of SPA predictor with 4-way set-associative PHT is even higher than the full-scale, per-address two-level
branch predictor with direct-mapped PHT. On average, an increase in branch prediction accuracy is 0.07% and 0.25% for SPA predictor with 2-way and 4-way set-associative PHT, respectively. This average increase in prediction accuracy compensates at least one third of the 0.73% average loss of prediction accuracy when using the original no-tag, direct-mapped PHT implementation. Moreover, the results tell us that set-associative implementations do not need to use full address tag to obtain higher branch prediction accuracy. In fact, only 5 bits and 7 bits are sufficient for 2-way and 4-way implementations, respectively. Hence, SPA predictor with 4-way set-associative PHT and 7-tag bit is the best candidate of all SPA implementations considered and these 7 tag bits only increase the circuit area by 7K bits from the original SPA implementation.
Figure 3.9 Prediction accuracy of SPA predictor with direct-mapped, 2-way, and 4-way set-associative PHTs.
3.5 Summary

I have proposed and evaluated a new scalable per-address two-level branch predictor called SPA predictor. It exploits value locality in the history of branch outcomes to decrease the die space of pattern history table (PHT) by almost 50%. Experimental results demonstrate that SPA prediction accuracy is comparable to PAp predictor, which uses a full-size PHT, making SPA predictor a practical and attractive implementation of per-address two-level branch prediction scheme.

However, SPA predictor suffers from internal conflict within the pattern history table (PHT), called pattern aliasing or interference. Therefore, I have proposed and evaluated several designs for a SPA branch predictor, ranging from direct-mapped PHT with no tag to set-associative PHT with various lengths of tag bits. Experimental results demonstrated that SPA predictor with set-associative PHT yields higher branch prediction accuracy than SPA predictor with direct-mapped PHT. I also found that the best implementation of SPA predictor is using 4-way set-associative PHT and 7-tag bit.
CHAPTER 4

DIRECTIONLESS YAGS BRANCH PREDICTOR (DLYAGS)

Among global-history branch prediction schemes that attempt to reduce the aliasing effects on prediction accuracy [4, 23, 31, 34, 29, and 40], YAGS (Yet Another Global Scheme) branch predictor [9] is one of the most effective as it combines the strengths of several previously proposed schemes. It uses a choice PHT to capture the biases and reserves two direction PHTs for the instances that do not comply with the biases in the choice PHT. This mechanism allows both direction PHTs to be much smaller than previously proposed in the bi-mode predictor [23]. Additionally, since the biases and exception instances of branches are separately stored, the amount of conflicts in all PHTs has also been reduced. Tag bits in each entry of the direction PHTs also help identify the exception instances and decrease the aliasing among multiple branches.

In this chapter, I propose and evaluate two alternate designs of YAGS branch predictor, which are: 1) unifying the two direction PHTs, and 2) using 3-bit counters in the choice PHT.

4.1 Mechanisms Directionless YAGS Branch Predictor (DLYAGS)

In previous chapter, I have already indicated two possible shortcomings of YAGS predictor, which are: 1) the equivalent size of both direction PHTs can cause space utilization problem since branches are likely to inhabit in one direction PHT more often than the other, and 2) the use of 2-bit counters in all PHTs may not be a good idea since
exception instances in the direction PHTs and branch biases in the choice PHT are likely
to exhibit very different behaviors. Two proposed approaches are discussed in the next
two subsections as a possible means to resolve these problems. Since one of the two
approaches is to combine two direction PHTs into one, this enhanced branch predictor is
given the name of Directionless YAGS (DLYAGS) predictor.

4.1.1 Unification of Direction PHTs

The motivation behind this proposed method is the observation that most
branches are biased in only one direction during a particular time span (i.e. their
behaviors would consistently remain in one direction within a certain time period and
then probably change into another direction). That could lead to a situation when one
direction PHT is used more frequently than another. Consequently, the under-utilized die
space in one direction PHT would limit the table size of the other, causing higher aliasing
rate in the highly-used PHT and accordingly impairing the branch prediction
performance.

I propose to combine the two direction PHTs in YAGS into a single PHT, which I
call an “exception PHT”. After the merge, the exception PHT now does not suffer from
the under-utilization problem since both types of exception instances (taken and not-
taken) are now residing in the same table. It is obvious that by doing this, the advantages
of less destructive aliasing gained from separating the branch types are not existent
anymore, making the unifying scheme look rather counter-intuitive. However,
experiments have shown that the negative aliasing effect is compensated by the benefits
gained from a higher space utilization of the exception PHT. The implementation of the modified version of YAGS predictor is shown in the diagram below.

![Figure 4.1 Unification of the two direction PHTs in YAGS predictor](image)

**Figure 4.1 Unification of the two direction PHTs in YAGS predictor**

### 4.1.2 Implementation of 3-bit Counters in Choice Predictor

With a PHT with 2-bit counters, only two consecutive mispredictions can cause a change of the prediction result. This strategy has been proven efficient in a regular PHT, which attempts to capture the dynamic behavior of branch outcomes. Branch biases, however, are less prone to transition and should be made more difficult to change the state. Therefore, I propose to use 3-bit counters in the choice PHT in regard to the above rationale. This strategy of using 3-bit counters has already been implemented in one of the PHTs of the tournament branch predictor in Alpha 21264 [20]. Even though the author has not indicated any reason as to why it improves the prediction accuracy, I
strongly believe that the more stable prediction state in 3-bit counters is the major cause of the increase in prediction accuracy.

4.2 Methodology

In order to conduct my experiments, I use a branch predictor framework and input traces from the 1st Championship Branch Prediction competition (CBP) [3], which is sponsored by Intel MRL and IEEE TC-uARCH. This branch prediction contest provides participants with a common evaluation framework, written in C++, to implement and evaluate their branch prediction algorithms.

In the experiments, I performed a series of simulations using the provided framework with each of the benchmark programs distributed by the competition committee. Benchmarks are classified into 4 categories: FP (floating point), INT (integer), MM (multimedia), and SERV (server). All the simulations have been run until completion. The performance metric used throughout this chapter is the output from the framework, which is the number of mispredictions per 1000 instructions. The size of all simulated branch predictors includes the tags of the direction PHTs (or the exception PHT in my scheme).

4.3 Experimental Results

In an attempt to increase the number of entries in both PHTs, I follow the example set in [7] by partially storing branch addresses bits in the tag entries. Keeping only 6 bits in each tag is found to yield the optimal results. Various proportions of the choice and
exception PHTs have also been experimented. The best configuration is the one with the total number of entries in the choice PHT approximately twice that in the direction PHTs (or the exception PHT). This is because the instances where branches comply with their biases take place much more frequently than the ones where they do not and hence require a larger die area to keep them. The number of bits in the BHR has been calculated and instantiated to accommodate the number of entries in the direction PHTs (or the exception PHT). In the rest of the chapter, I use this configuration for the simulations of all experimented branch predictors.

4.3.1 Unification of Direction PHTs

Figure 4.2 (on the next page) depicts the results from measuring average prediction accuracy of all the benchmarks simulated across all the hardware budgets ranging from 8K to 1Mbits. The proposed mechanism has negative effects on branch prediction accuracy for the hardware budgets of 128K, 256K, and 512Kbits. The 1-Mbit branch predictor also observes a very little prediction accuracy improvement, only 0.45%. This is mainly because the higher space utilization caused by the merging of the two direction PHTs is unnecessary for large branch predictors and thus cannot provide sufficient benefits to compensate for the higher aliasing rate. On the contrary, I observe a higher prediction accuracy growth in smaller branch predictors, particularly an 8-Kbit YAGS with the exception PHT, which experiences the highest improvement of branch prediction accuracy, 4.41%. The average increase in prediction accuracy of the proposed branch predictors with the hardware budgets of 8K-64Kbits is 2.00%. This can be
attributed to the fact that the exception PHT uses the die space more efficiently than the direction PHTs, especially in the branch predictors with limited budgets.

Figure 4.2 Prediction accuracy of YAGS branch predictors with direction PHT and exception PHT across all hardware budgets

Figure 4.3 Prediction accuracy of 8-Kbit YAGS branch predictors with direction PHT and exception PHT under each benchmark.
These results show that YAGS with the exception PHT is perfectly suitable for a small branch predictor in pipelined embedded processors, such as XScale, ARM, and Nios-II, or microcontroller.

Figure 4.3 presents the prediction accuracy of the branch predictors with 8-Kbit budget for each benchmark. Most of the prediction accuracy gains come from the set of SERV benchmarks, averaging 9.24%. The biggest prediction accuracy deficit, -16.77%, is in one of the MM benchmarks, supposedly due to the higher aliasing rate. However, it is negated by the prediction accuracy increases from other benchmarks, especially from the SERV-4 and SERV-5 which yield an impressive improvement of 12.43% and 13.99%, respectively.

### 4.3.2 Implementation of 3-bit Counters in Choice Predictor

The use of 3-bit counters enlarges the size of the choice PHT, leaving less die space for the direction PHTs. The resulting higher aliasing rate could prove to be a significant performance block in small branch predictors. Figure 4.4 (on the next page) has confirmed this hypothesis as 16K-64Kbit YAGS with 3-bit counters all encounter prediction accuracy losses and the 8Kbit one shows a miniscule improvement of 0.59%. On the contrary, I observe a higher prediction accuracy growth in larger branch predictors. The average prediction accuracy gain of the 3-bit scheme with the hardware budgets of 128K-1Mbits is 1.10%. The 1-Mbit configuration experiences the highest improvement of branch prediction accuracy, 1.43%. This is because, despite the extra die space taken by all the 3-bit counters in the choice PHT, the remaining die capacity dedicated to the branch predictor is still big enough to reduce the negative aliasing
effects. Meanwhile, the increase in the prediction accuracy can be attributed to the higher stability in the choice PHT with 3-bit counters.

![Prediction accuracy of YAGS branch predictors with 2-bit and 3-bit counters across all hardware budgets](image)

**Figure 4.4** Prediction accuracy of YAGS branch predictors with 2-bit and 3-bit counters across all hardware budgets

Based on the results shown here, the scheme to deploy 3-bit counters in the choice PHT has demonstrated that it can help boost the prediction accuracy of a large YAGS branch predictor, making it well suited to be deployed in current and future superscalar microprocessors.

Figure 4.5 depicts the prediction accuracy of the branch predictors with 1-Mbit budget for each benchmark. Even though the percentage of the prediction accuracy gain is relatively low (the average increase in prediction accuracy for the SERV benchmarks is only 6.6%, comparing to the 9.24% of the previous scheme), the 3-bit scheme has benefited the branch predictor performance particularly in the MM-3 benchmark which gains 6.34% improvement in prediction accuracy, comparing to the deficit of -16.77% in
the previous scheme. This is largely due to two main reasons: 1) the lower aliasing rate caused by the bigger size of the branch predictor and 2) the more stable nature of the counters in choice PHT caused by the use of 3-bit counters. The proposed scheme suffers a prediction accuracy loss in the INT benchmarks, primarily owing to the dynamic behaviors of branches in those benchmarks. It, however, benefits from high prediction accuracy gain in the remaining benchmarks, resulting in a positive outcome for the overall improvement percentage.

![Figure 4.5 Prediction accuracy of 1-Mbit YAGS branch predictors with 2-bit and 3-bit counters under each benchmark](image)

**Figure 4.5 Prediction accuracy of 1-Mbit YAGS branch predictors with 2-bit and 3-bit counters under each benchmark**

### 4.3.3 Combined Mechanism

While the combination of the two mechanisms provides some prediction accuracy boosts to the branch predictor, its benefits are inferior to those when the two schemes are adopted separately. Hence, I choose to omit the discussion of its experimental results.
4.4 Summary

Even though YAGS predictor is one of the most efficient branch prediction schemes that use global branch history, there still are two possible shortcomings of its design. The first is a space utilization problem since both direction PHTs are of equal size and branches are more likely to inhabit more often in one PHT than another. The second is an inappropriate use of 2-bit counters for all kinds of branch instances despite their likely different natures.

I have proposed and evaluated two new mechanisms to help improve prediction accuracy of YAGS predictor. The first method of merging two direction PHTs increases the space utilization in the second-level PHT while the second one uses 3-bit counters to make the biases in the choice PHT more difficult to change states. Experimental results show that keeping only 6 tag bits for each entry in the combined direction PHT yields the optimal results. The total number of entries in the choice PHT should also be set to approximately twice that in the direction PHTs (or the exception PHT). This is because the instances where branches comply with their biases take place much more frequently than the ones where they do not and hence require a larger die area to keep them. Finally, the results demonstrate that, while both proposed techniques enhance the branch prediction accuracy of YAGS, the former is more suitable for a small branch predictor in embedded microprocessors and the latter for a large one in superscalar microprocessors.
CHAPTER 5

ENHANCING THE ACCURACY OF O-GEHL PREDICTOR

While so much efforts in branch prediction have focused on reducing the aliasing rate or at least limiting its impact [4, 9, 23, 27, 29, and 40], this strategy can only improve the prediction accuracy so far since branch behaviors in some benchmarks are unpredictable beyond human understanding. The next logical step commonly perceived by many researchers is to leave the task to an intelligent autonomous learning system. Jimenez has proposed to use the perceptron, a simple version of neural networks, in a branch predictor to help in the process of learning branch behaviors and finally predicting their outcomes [17, 19]. As a result, the perceptron predictor achieved increased accuracy over other traditional branch predictors, and has received increased attentions recently. However, the perceptron predictor is still considered impractical due to its highly complex mechanisms and long prediction latency.

In the first Championship Branch Prediction competition (CBP-1) [3] where complexity is not an issue, the top three proposed algorithms are primarily based on the perceptron predictor. With a limited storage budget of 64K bits and a distributed common evaluation framework for fair comparisons, the top three branch predictors are at least 4.34% more accurate than the fourth placed branch predictor that does not use the perceptron. The first-placed branch predictor, the O-GEHL (Optimized GEometric History Length) predictor [35], has modified the perceptron predictor to exploit various lengths of global branch history. It also contains a dynamic mechanism that can
adaptively adjust the history lengths used in 3 of its 8 predictor tables, allowing for the use of even longer branch history when necessary.

In this chapter, I perform an extensive analysis on the O-GEHL predictor, mainly searching for its characteristics that can be exploited to further increase its efficiency. Based on the analysis results, I subsequently propose two alternate designs of the O-GEHL branch predictor to improve its accuracy, which are: 1) increasing the space utilization of its first predictor table, T0, by dynamically adjusting branch history lengths used in the indexing function, and 2) adding an additional predictor table without requiring extra space by means of sharing hysteresis bits.

5.1 Methodology

In order to conduct the experiments, I use a branch predictor framework and input traces from the 1st Championship Branch Prediction competition (CBP) [1], which is sponsored by Intel MRL and IEEE TC-uARCH. This branch prediction contest provides participants with a common evaluation framework, written in C++, and a fixed hardware budget, which is 64K bits, to implement and evaluate their branch prediction algorithms.

In the experiments, I performed a series of simulations using the provided framework with each of the benchmark programs distributed by the competition committee. Benchmarks are classified into 4 categories: FP (floating point), INT (integer), MM (multimedia), and SERV (server). All the simulations have been run until completion. The performance metric used throughout this chapter is the output from the framework, which is the number of mispredictions per 1000 instructions.
Because the O-GEHL predictor’s original size is 64K bits, enlarging or reducing the predictor requires some modifications to the predictor configurations. I have decided to simply double, or halves, the number of entries in each predictor table when increasing, or decreasing, the predictor size.

### 5.2 Analysis of the O-GEHL predictor

#### 5.2.1 Space Utilization of the Predictor Tables

After careful and thorough analysis of the predictor to promote a better understanding of its characteristics and to probably discover its shortcomings, I have found quite an interesting statistics regarding the space utilization of each predictor table. As shown in table 5.1, while the allocated spaces of most predictor tables are used more than 74%, even more than 92% in three of them, the usage percentage of T0 is surprisingly low, just around 52%. This apparently introduces an opportunity for enhancing the predictor’s performance.

<table>
<thead>
<tr>
<th></th>
<th>T 0</th>
<th>T 1</th>
<th>T 2</th>
<th>T 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Utilization (%)</td>
<td>52.66</td>
<td>74.07</td>
<td>86.90</td>
<td>75.39</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>T 4</th>
<th>T 5</th>
<th>T 6</th>
<th>T 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Utilization (%)</td>
<td>92.12</td>
<td>86.88</td>
<td>93.56</td>
<td>92.75</td>
</tr>
</tbody>
</table>

**Table 5.1 Space utilization in each predictor table**
Despite T0’s low utilization, I cannot simply reduce its size without inflicting considerable damage to the prediction accuracy. This is because each benchmark exhibits a highly different branch distribution pattern across T0. In an attempt to seek for an insight into branch behaviors and eventually find a way to make a better use of T0 space, I have conducted further studies of the way T0 entries are occupied. The results shown in table 5.2 are the number of entries that has been used for each benchmark during the entire execution. They are grouped into 4 categories, which are FP, INT, MM, and SERV, as previously mentioned. The average value of each category is also calculated and shown in the table.

<table>
<thead>
<tr>
<th>Category</th>
<th>FP 1</th>
<th>FP 2</th>
<th>FP 3</th>
<th>FP 4</th>
<th>FP 5</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>INT</td>
<td>400</td>
<td>405</td>
<td>682</td>
<td>489</td>
<td>231</td>
<td>441.4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Category</th>
<th>MM 1</th>
<th>MM 2</th>
<th>MM 3</th>
<th>MM 4</th>
<th>MM 5</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>SERV</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2043.2</td>
</tr>
</tbody>
</table>

Table 5.2 the number of T0 entries used in the execution of each benchmark

From the results, all SERV benchmarks have their branches spread throughout T0, making full use of the allocated table space. This is a level of success other benchmarks unfortunately fail to match. Specifically, several FP and INT benchmarks
(even one from MM) have occupied only 20% of the total T0 capacity. Not only do such drastic variations in the table usages undoubtedly call for a more efficient way to distribute branches across the tables, but also indicate how difficult it would be to find one.

However, higher space utilization in other predictor tables suggests that the simplest and most intuitive scheme to improve T0 usage percentage probably is to use global branch history as another parameter for the indexing function of T0. In the other word, L(0) of the new scheme is no longer zero.

I have performed various experiments with different values of L(0). One of the best configurations is to set L(0) to 3 and L(1) to 5 while other L(n) are recalculated using to the same equation given in the section 2.1. Figure 5.1 shows the effects this configuration has on the prediction accuracy for each benchmark set. FP and INT benchmarks benefit from larger L(0), gaining 5.22% and 1.99% of prediction accuracy, respectively. Unfortunately, the overall prediction accuracy is decreased because MM and SERV benchmarks suffer 7.24% and 10.24% loss in prediction accuracy respectively. This scenario gets worse when other configurations with even larger L(0) have been used in the experiments. The results reveal that while an indexing function with larger L(0) improves the prediction accuracy of the benchmarks that exhibited low utilization percentage, specifically FP and INT, it degrades that of the others.

Apparently, short global branch history still plays a significant role in maintaining high branch prediction accuracy in the O-GEHL predictor, especially for both MM and SERV benchmarks. This makes the task of improving the predictor’s efficiency and accuracy much more complicated since simply increasing the length of global branch
history alone, as done in traditional perceptron predictor [19], is not going to accomplish
the job anymore.

Figure 5.1 Improvement in prediction accuracy when \( L(0) = 3 \) and \( L(1) = 5 \)

5.2.2 The Effects of Three Additional Predictor Tables, T8-T10

Due to a space limitation in the O-GEHL predictor, table T8, T9, and T10 are
arranged to collocate with T2, T4, and T6, respectively. These extra tables are physically
assigned their own spaces in my experiments for a study of their impact on the prediction
accuracy. As shown in table 5.3, approximately 1% improvement on prediction accuracy
is observed each time an extra table is added. The third table (T10) however is an
exception as it causes a slight drop in the prediction accuracy. These results show that
only 9 or 10 tables are sufficient to obtain higher predictor efficiency.

I have also performed other experiments where the traditional O-GEHL predictor
is added with a rather small table (1-2K bits). Various versions of global branch history
with different lengths are used to index to the extra table. Even with four-way
associativity being implemented on the table to increase its space utilization, the consequent prediction accuracy is not improved. This result, along with the finding from table 2 that most of MM and all SERV benchmarks use up almost all the table entries, reveals that an extra table requires a large number of entries, preferably 2K entries, to have a positive impact on the overall prediction accuracy.

<table>
<thead>
<tr>
<th></th>
<th>O-GEHL (64K bits)</th>
<th>9 tables (72K bits)</th>
<th>10 tables (80K bits)</th>
<th>11 tables (88K bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Misprediction</td>
<td>2.82</td>
<td>2.79</td>
<td>2.75</td>
<td>2.76</td>
</tr>
</tbody>
</table>

Table 5.3 The effects on prediction accuracy when more tables are added into the predictor

5.3 Proposed Approaches and Experimental Results

5.3.1 Increasing Space Utilization in T0

One interesting fact about T0 is that it is under utilized when used with particular benchmarks but almost fully utilized with the others. Therefore, static modifications to table size or indexing function cannot be made without sacrificing the prediction accuracy in some benchmarks. An appropriate approach is to find a means to dynamically adjust the length of global branch history used in the indexing function of each predictor table, which is $L(n)$ where $0 \leq n \leq 10$, with regard to what kind of benchmark is being used at the time.
Distinguishing between benchmarks that need a small $L(n)$ and those that need a larger one is a challenging problem. In CBP-1, Gao and Zhou, whose proposed branch predictor won a second place, have approached a similar problem with an adaptive approach to modify the perceptron predictor to suit each benchmark [14]. A similar idea is used in this proposed scheme to distinguish individual benchmarks for an appropriate adjustment of the indexing function’s parameters.

I have decided to use the number of conditional branches as a deciding factor in this process since in most cases there are more of them in FP and INT than in MM and SERV benchmarks. In my approach, each $L(n)$ is initially given a default value as done in the O-GEHL predictor. After a certain time period, $t$, has passed, the number of conditional branches, $c$, is then compared to a pre-specified value, $v$. If $c$ is less than $v$, it is likely that the currently running benchmark program is in either MM or SERV category and these $L(n)$ values are not changed. Otherwise, $L(0)$ is set to 3, $L(1)$ to 5, and $L(10)$ to 200, which is the same value, while all other $L(n)$ values will be re-calculated using the given geometric series equation. These values have been proven to have biggest positive impact on the prediction accuracy for FP and INT benchmarks, as previously shown in figure 5.1. Once this process is done, regular prediction mechanism of the O-GEHL predictor can go on without any other interruption. As a result, an overhead cost is kept very small and each benchmark is likely to run in the predictor configuration that suits it the most.

Not only should $t$ be small enough to allow for a timely adjustment of the $L(n)$ values, but should also be sufficiently large for an efficient classification of benchmark programs. Therefore, a 7-bit counter responsible for tracking the number of all branch
instructions, both conditional and unconditional ones, is chosen to represent the amount of time that has passed. The counter overflow signals that the time $t$ is reached, and triggers the $L(n)$ adjustment process. The use of this counter allows the time $t$ to be extremely short, compared to the whole execution time, while the value of $v$ can be set to, based on the information I have gathered on previous runs of all benchmark programs, approximately 84% of the counter’s maximum value possible. With this scheme, another 7-bit counter is required to track the number of conditional branches, and a comparator is needed for the comparison process.

I have added this mechanism into the O-GEHL predictor and run the experiments with various budget sizes, ranging from 8K-1M bits. The results are shown in figure 5.2.

![Figure 5.2 Misprediction rates when the dynamic adjustment of $L(n)$ is incorporated into the predictor](image)

The prediction accuracy is improved in every case except for 8K- and 16K-bit predictors. This is not surprising since $T_0$ in these budgets is 4-8 times smaller than usual
and the task of increasing its space utilization becomes almost impossible to accomplish. However, an average decrease of 0.025 in misprediction rate can be observed in larger predictors, which is equal to 0.96% improvement in prediction accuracy. In fact, the most impressive improvement occurs in the 128K-bit predictor, which exhibits almost 2% gain in prediction accuracy. 512K- and 1M-bit predictors are also enjoying more than 1% increase in prediction accuracy. This is all because higher space utilization can be achieved when T0 is larger.

The fact that the O-GEHL predictor is specifically optimized for a 64K-bit budget suggests that the predictors with larger budget may be able to experience even higher rise in prediction accuracy with different combination of L(n) or different space allocation method for each predictor table. This requires extensive examinations of the predictor with an extremely large set of parameters, and therefore is outside the scope of this chapter. However, one can easily begin the study by concentrating on T0 space utilization since the analysis and experimental results have already confirmed that it certainly is an important factor in deciding the prediction accuracy.

It is also inevitable that further performance improvement of the highly optimized 64K-bit O-GEHL predictor will be highly difficult to achieve. Nevertheless, my proposed scheme still has reduced the misprediction rate by 0.02, which is roughly a 0.71% improvement. Comparing with the difference in prediction accuracy between the first-placed O-GEHL predictor and the second-placed Gao’s proposed predictor [14], which is only 0.003 apart (a tiny 11% increase), the prediction accuracy incurred by my proposed mechanism over the O-GEHL predictor is impressively almost 7 times better.
<table>
<thead>
<tr>
<th>Size (bits)</th>
<th>FP</th>
<th>INT</th>
<th>MM</th>
<th>SERV</th>
</tr>
</thead>
<tbody>
<tr>
<td>8K</td>
<td>7.42</td>
<td>-6.55</td>
<td>-1.03</td>
<td>-1.27</td>
</tr>
<tr>
<td>16K</td>
<td>15.30</td>
<td>2.63</td>
<td>-0.38</td>
<td>-3.07</td>
</tr>
<tr>
<td>32K</td>
<td>18.58</td>
<td>5.63</td>
<td>0.02</td>
<td>-4.95</td>
</tr>
<tr>
<td>64K</td>
<td>13.24</td>
<td>2.06</td>
<td>0.03</td>
<td>-1.13</td>
</tr>
<tr>
<td>128K</td>
<td>27.6</td>
<td>0.22</td>
<td>0.69</td>
<td>-0.04</td>
</tr>
<tr>
<td>256K</td>
<td>14.65</td>
<td>0.83</td>
<td>-0.22</td>
<td>-1.28</td>
</tr>
<tr>
<td>512K</td>
<td>13.17</td>
<td>1.30</td>
<td>0.37</td>
<td>-1.25</td>
</tr>
<tr>
<td>1M</td>
<td>20.75</td>
<td>0.79</td>
<td>0.27</td>
<td>-2.28</td>
</tr>
<tr>
<td>Average</td>
<td>16.34</td>
<td>0.86</td>
<td>-0.03</td>
<td>-1.91</td>
</tr>
</tbody>
</table>

**Table 5.4 Improvement from dynamic L(n) adjustment in each benchmark group**

To elaborate the contributions made by my approach, I have performed an analysis to evaluate its impact on the prediction accuracy of each benchmark category, as shown in table 5.4. The decrease in prediction accuracy is displayed in a negative number.

FP benchmarks are undoubtedly the biggest beneficiary with an average of 16.34% increase in prediction accuracy. The lowest improvement, 7.42%, lies in the predictor with 8K-bit budget while the highest one, 27.6%, is enjoyed by the 128K-bit predictor. Meanwhile, almost all INT benchmarks are experiencing various levels of improvement in prediction accuracy, particularly with the highest rise of 5.63% belonging to the 32K-bit predictor. The average increase in prediction accuracy of all INT benchmarks is disappointingly only 0.86%. This is mainly because of a 6.55% deficit in the 8K-bit predictor, which, among the predictors of all sizes, is the only one that suffers a prediction accuracy loss. These improvements are the direct consequences from heightening T0 space utilization, which is however not the case for SERV benchmarks that unfortunately have not benefited from the proposed scheme at all. Insignificant
changes around or less than 1% in prediction accuracy can also be observed in all MM benchmarks.

The decrease of prediction accuracy in MM and SERV benchmarks takes place because the process of distinguishing between each benchmark category is not efficient enough. As a result, a few MM and SERV benchmark programs sometimes have to be run with large L(n), causing more mispredictions. To perfectly classifying the benchmarks, either a highly complicated mechanism must be used or a thorough profiling must be conducted. The former is very likely going to slow down the prediction process while requiring more die space. On the other hand, despite being faster and requiring no extra space, the latter is highly dependent on the profiled benchmarks and cannot adapt well to a new type of benchmarks.

The results also reveal that the proposed scheme works particularly well with most FP and INT benchmarks and is therefore most suitable for microprocessors that run scientific and computation-intensive applications. For computers whose jobs involve running applications with multimedia or server workloads, a traditional O-GEHL predictor is more appropriate. Nonetheless, because its overall misprediction rate is still lower than the O-GEHL predictor’s, my proposed approach is considered a better alternative in microprocessors running general applications with unknown workloads.

5.3.2 Adding an Extra Predictor Table without Requiring More Space

In the O-GEHL predictor, prediction accuracy increases when extra predictor tables with different lengths of global history are added. This idea of multiple global history lengths used in the O-GEHL predictor was initially introduced in [27], and then
refined by Evers et al. [11]. Since then, it has appeared in several branch prediction schemes and been proven to be highly effective in boosting the prediction accuracy. However, the analysis results in section 5.2 have demonstrated that only up to 2 extra predictor tables are actually required for the accuracy improvement of O-GEHL predictor.

My proposed approach is to add an extra table into the predictor without ever increasing the allocated table space. With the introduction of hysteresis bits being shared between 2 table entries, the number of entries in each predictor table does not even need to be reduced to make room for an extra table.

In a 2-bit counter, the most significant bit is a direction bit, which provides the prediction result, while the least significant bit is a hysteresis bit, which prevents the direction bit from immediately changing after just a single misprediction. Seznec et al. have proposed an approach of sharing a single hysteresis bit between 2 adjacent table entries [36] to save up more die space of the branch predictor in an EV8 microprocessor and then use it in a more fruitful fashion. This strategy has been proven to work efficiently by Loh, who shows that hysteresis bits are strongly biased and that it is unnecessary to waste an entire bit for hysteresis in a single table entry [26]. His experimental results also illustrate that, by having 2 entries in gshare predictor share a hysteresis bit, the entropy per prediction has increased only slightly, implying that its prediction accuracy is unlikely to be significantly affected.

I have adopted the hysteresis sharing policy to save the predictor space and create room for an extra predictor table. Since higher degree of aliasing is an inevitable consequence to the hysteresis-bit sharing scheme and is likely to cause more damages
with small predictor tables, T1 is exempted from being implemented the mechanism. As a result, the overall die space saved from using this approach in the 7 remaining predictor tables, each of which contains 2K entries, is exactly 7K bits. The area of this size is precisely what another predictor table, T8 in this case, actually needs as it will contain 2K entries of 3-bit counters and 1K entries of a hysteresis bit.

The misprediction rates, before and after the sharing of hysteresis bits, across various predictor budgets are shown in figure 5.3. Listing of the improved percentages in prediction accuracy for each predictor is also presented in table 5.5. The results get worse with the predictor of size 8K-64K bits while some improvement can be seen in the predictors larger than 64K bits. This is simply because small branch predictors tend to already have high aliasing rate. Sharing hysteresis bits is likely to multiply increase the interference to the extent that it cannot be compensated by the benefits gained from having an extra predictor table anymore.

---

**Figure 5.3** Misprediction rates when 9 predictor tables with shared hysteresis bits are used in the predictor
Table 5.5 Improvement percentages in prediction accuracy across all predictor budgets when T8 is added

<table>
<thead>
<tr>
<th>Budget</th>
<th>Improv (%)</th>
<th>Budget</th>
<th>Improv (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>8K</td>
<td>-7.75</td>
<td>128K</td>
<td>2.54</td>
</tr>
<tr>
<td>16K</td>
<td>-5.73</td>
<td>256K</td>
<td>0.95</td>
</tr>
<tr>
<td>32K</td>
<td>-3.50</td>
<td>512K</td>
<td>1.08</td>
</tr>
<tr>
<td>64K</td>
<td>-2.38</td>
<td>1M</td>
<td>1.05</td>
</tr>
<tr>
<td>Average</td>
<td>-4.84</td>
<td>Average</td>
<td>1.40</td>
</tr>
</tbody>
</table>

Table 5.5 shows that the average decrease in prediction accuracy of the predictors sized 8K-64K bits is 4.84%, indicating that the strategy does not work well in small branch predictors. Meanwhile, the average increase in prediction accuracy of the predictors with hardware budget between 128K and 1M bits is 1.40%, which is better than 1.17% improvement obtained from the equal-sized predictors with dynamic-L(n) scheme. The highest improvement percentage of prediction accuracy from the hysteresis-sharing scheme is 2.54% in 128K-bit predictor and the lowest is 0.95% in 256K-bit predictor, while in the dynamic L(n) scheme the highest is only 1.96% in 128K-bit predictor and the lowest only 0.57% in 256K-bit predictor. These results reveal to us a very interesting discovery: Even though the hysteresis-sharing scheme incurs, on average, higher misprediction rate when all hardware budgets are considered, it outperforms the dynamic-L(n) scheme in large branch predictors. Consequently, it can be concluded that a larger number of tables definitely are an essential factor in enhancing the O-GEHL prediction accuracy when hardware budget is not limited.

To further elaborate the contributions made by the proposed approach, I have performed an analysis to evaluate its impact on the accuracy of large predictors (128K-
1M bits) with each benchmark category, as shown in table 6. Even though the hysteresis-sharing scheme has had a reasonable success with all FP benchmarks, with an impressive average improvement of 16.39% in prediction accuracy, it is still slightly lagging behind the dynamic-L(n) scheme that exhibits an average of 19.04% increase in prediction accuracy over the same range of hardware budget. However, on average for predictors of size 128K-1M bits, it manages to improve 0.63% and 2.82% of prediction accuracy in MM and SERV benchmarks, respectively, while the dynamic-L(n) scheme can only achieve 0.28% improvement in MM benchmark and even suffer 1.21% deficit in SERV benchmark. These results confirm the hysteresis-sharing scheme as an attractive alternative in improving the prediction accuracy in large predictors, with a wide range of applications.

<table>
<thead>
<tr>
<th></th>
<th>FP</th>
<th>INT</th>
<th>MM</th>
<th>SERV</th>
</tr>
</thead>
<tbody>
<tr>
<td>128K</td>
<td>24.88</td>
<td>-2.94</td>
<td>0.47</td>
<td>8.64</td>
</tr>
<tr>
<td>256K</td>
<td>10.08</td>
<td>-0.43</td>
<td>-0.18</td>
<td>3.98</td>
</tr>
<tr>
<td>512K</td>
<td>14.55</td>
<td>0.77</td>
<td>-1.92</td>
<td>0.20</td>
</tr>
<tr>
<td>1M</td>
<td>16.05</td>
<td>1.00</td>
<td>4.14</td>
<td>-1.55</td>
</tr>
<tr>
<td>Average</td>
<td>16.39</td>
<td>-0.40</td>
<td>0.63</td>
<td>2.82</td>
</tr>
</tbody>
</table>

**Table 5.6** Percentages of the improved prediction accuracy for each benchmark category, in the predictors with budget larger than 64K bits

**5.3.3 Combined Mechanism**

In term of prediction accuracy, the combination of the two mechanisms is only slightly better than the dynamic-L(n) scheme and is still inferior to the hysteresis-sharing scheme when the predictor size is larger than 64K bits. For the predictors of smaller size,
it performs worse than each individual scheme. Hence, I choose to omit the discussion of its experimental results.

5.4 Summary

I have performed an analysis on the O-GEHL predictor and, based on the results, proposed and evaluated two new mechanisms to help improve its prediction accuracy. Since T0 is underutilized, my first proposed scheme is to increase its space utilization by dynamically adjusting the lengths of global branch history to best suit the benchmark type. The second scheme proposes to add an extra predictor table without incurring any hardware budget, using the space saved from sharing a single hysteresis bit between 2 table entries.

The first scheme increases the prediction accuracy in almost every hardware budget but 8K and 16K bits, which are too small for the approach to be effective. In a budget of 64K bits, a regular O-GEHL predictor size, the proposed approach improves the prediction accuracy by a much larger margin than what the O-GEHL predictor does over the second-placed predictor in the CBP competition. It also works particularly well with most FP and INT benchmarks and is thus suitable for microprocessors that usually run scientific and computation-intensive applications.

Even though the second scheme causes the prediction accuracy to drop when a hardware budget is 8K-64K bits, it outperforms the first scheme when a hardware budget is 128K-1M bits. A larger number of tables are hence an essential factor in improving the O-GEHL prediction accuracy when hardware resources are not limited. Moreover, for branch predictors of size 128K-1M bits, it is even capable of increasing the prediction
accuracy in MM and SERV benchmarks, which is a task the first scheme fails to accomplish. This indicates that when there is an abundant hardware resource, adding extra predictor tables is more likely to improve the prediction accuracy than increasing the size of the existing predictor tables. Further analysis of the experimental results reveals that even though the second scheme performs worse than the first one on average, particularly in FP and INT benchmarks, it enjoys more success in some MM and all SERV benchmarks.

For branch predictors sized 32K and 64K bits, the first scheme using dynamic L(n) is the best option, while the O-GEHL predictor with regular configurations is still the best when hardware resources are very limited, 8K-16K bits in this case.
CHAPTER 6

CONCLUSION AND FUTURE WORK

I have discussed three major paradigms in branch prediction schemes, which are per-address two-level, global-history, and neural-learning branch predictors. Each paradigm has its own strengths, weaknesses, and limitations. This dissertation has carefully selected three most efficient branch predictors, one from each paradigm, and proposed more robust mechanisms to help mitigate the imminent problems in these branch predictors. Each mechanism is thoroughly evaluated and experimental results are presented. Finally, the conclusions for all proposed mechanisms are shown as follows.

6.1 Conclusions

While per-address two-level branch predictors typically experience low interference or aliasing rate, their implementation costs in term of die spaces are often too expensive for practical use. I propose a scalable per-address (SPA) predictor that requires only about half of the original die space while maintaining high branch prediction accuracy. Specifically, SPA predictor leverages the existence of value locality in the history of branch outcomes to reduce the size of pattern history table (PHT) by about 50%. Experimental results on eight SPEC2000 benchmark programs reveal that SPA prediction accuracy is only slightly lower than a branch predictor with a full-size PHT by 0.23% to 1.39%. Thus, the proposed SPA predictor makes the implementation of per-address two-level branch predictors more practical and attractive.
In other low cost branch prediction schemes [8], [37], relatively high branch prediction accuracy is achieved by the reduction of both capacity and conflict aliasing. SPA predictor, on the other hand, hardly suffers from conflict aliasing due to its per-address mechanism, and mainly concentrates on exploiting the value locality within each entry of the PHT, making it a cost-effective candidate for being used in combination with other branch prediction schemes that use a per-address PHT.

Unfortunately, SPA branch predictor still suffers from internal conflict within the pattern history table (PHT), called pattern aliasing or interference. Therefore, I further propose and evaluate alternative designs of SPA predictor to reduce aliasing or interference rate in its PHT as well as to improve its prediction accuracy. Several PHT designs using set-associative PHT with various lengths of tag bits are evaluated against a direct-mapped PHT used in the original SPA branch predictor design. Experimental results on eight SPEC2000 benchmark programs demonstrated that SPA predictor with set-associative PHT yields higher branch prediction accuracy than SPA predictor with direct-mapped PHT. They also reveal that a SPA predictor with 4-way set-associative PHT using 7-tag bit yields the highest prediction accuracy.

For a global-history branch prediction scheme, I propose and evaluate two alternate designs of YAGS (Yet Another Global Scheme) predictor, one of the most efficient branch predictors in this paradigm, to help improve its prediction accuracy. The first proposed method combines two direction PHTs together to increases the space utilization in the second-level PHT. The second one proposes to use 3-bit counters in the choice PHT in order to inject a higher stability to the biases. Experimental results show that the merging technique benefits most for a small branch predictor (8K-64Kbits) with
the 2.00% average gain in branch prediction accuracy. The biggest payoff goes to the 8-Kbit YAGS with the 4.41% prediction accuracy improvement. Meanwhile, the 3-bit-counter approach benefits most for a large branch predictor (128K-1Mbits) with the average prediction accuracy gain of 1.10%. The 1-Mbit configuration experiences the highest improvement of branch prediction accuracy, which is 1.43%. These results demonstrate that, while both proposed techniques enhance the branch prediction accuracy of YAGS, the former is more suitable for a small branch predictor in embedded microprocessors and the latter for a large one in superscalar microprocessors.

For a neural-learning branch prediction scheme, the O-GEHL branch predictor, with its perceptron-like mechanism and an ability to exploit branch history of various lengths, has officially proven to provide highest prediction accuracy under a distributed set of benchmarks against other branch prediction schemes with the same storage budget of 64K bits in an international branch prediction contest, CBP-1. I propose and evaluate two alternate designs of the predictor to further improve its prediction accuracy. The first proposed method dynamically adjusts the lengths of branch history used in its indexing function regarding to the type of a benchmark currently in execution, in order to increase space utilization of the first predictor table. The second one proposes to add an extra table into the predictor using the space saved from the sharing of hysteresis bits. Experimental results across the hardware budgets of 8K-1M bits show that the increase in space utilization improves the prediction accuracy in most cases, except for the 8K and 16K bits. Particularly, the O-GEHL predictor of regular size, 64K bits, when equipped with this mechanism, enjoys 0.71% rise in prediction accuracy, which is impressively 7 times the difference in prediction accuracy between the original O-GEHL
predictor and the second-placed branch predictor. Floating point and integer benchmarks benefit the most from the scheme, suggesting that it is suitable for microprocessors that mostly run scientific or computation-intensive applications. Meanwhile, adding an extra predictor table increases the accuracy only in the predictors with 128K-1M bits, implying that increasing the number of tables is more likely to benefit the predictor than increasing the number of table entries. Unlike the first scheme, the second one does not decrease the average prediction accuracy in multimedia and server benchmarks, making it an attractive choice for a wider range of applications and workloads.

6.2 Future Directions

For SPA predictor, a new kind of interference, which is called pattern aliasing, has been introduced and therefore requires more studies on its characteristics and impact on prediction accuracy. With a better understanding on the pattern aliasing, future research can more efficiently reduce the amount of this interference and boost the performance of SPA predictor. Moreover, further experiments with an extremely large die space (more than 1M bits) being allocated for branch predictors should be conducted to truly reflect a future trend of microprocessor technology. I strongly believe that SPA predictor should be able to take most advantage from the increased space.

One possible means to improve the prediction accuracy of DLYAGS predictor is probably to adopt the strategy of exploiting various lengths of global branch history, just like in the O-GEHL predictor. Having more number of global branch history instead of one, this new version of O-GEHL predictor must also have more number of tables, each of them is indexed by branch history of different lengths. There must be a mechanism to
decide from which table an exception instance for the particular branch bias must be read. Consequently, another complex mechanism must be carefully designed to allow for efficient entrances and replacements of exception instances in each predictor table. This requires a large amount of work and is definitely a perfect subject for future research.

In chapter 5, I proposed to increase space utilization in the table T0 of O-GEHL predictor. It is the task that I have accomplished only to a certain level because the proposed scheme still has a negative impact on multimedia and server benchmarks. Apparently, the mechanism used to distinguish each benchmark type is not sufficiently effective. I therefore suggest that one of the possible future research directions involving O-GEHL predictor is to find a more efficient way to classify the benchmarks to prevent the predictor from using inappropriate global branch history lengths. Furthermore, a more systematic and automatic approach to decide the configuration of each predictor table is certainly an interesting subject for future investigations.
References


