2014

BOOTSTRAP AGGREGATING BRANCH PREDICTORS

Ibraham Burak Karsli
University of Rhode Island, burak.ibrahim.karsli@gmail.com

BOOTSTRAP AGGREGATING

BRANCH PREDICTORS

BY

IBRAHIM BURAK KARSLI

A DISSERTATION SUBMITTED IN PARTIAL FULFILLMENT OF THE

REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

IN

ELECTRICAL, COMPUTER & BIOMEDICAL ENGINEERING

UNIVERSITY OF RHODE ISLAND

2014

MASTER OF SCIENCE THESIS

OF

IBRAHIM BURAK KARSLI

APPROVED:

Thesis Committee:

Major Professor Resit Sendag

Jien-Chung Lo

Lutz Hamel

Nasser H. Zawia

DEAN OF THE GRADUATE SCHOOL

UNIVERSITY OF RHODE ISLAND

2014

ABSTRACT

After over two decades of extensive research on branch prediction, branch mispredictions are still an important performance and power bottleneck for today's aggressive processors. All this research has introduced very sophisticated and accurate branch predictor designs, TAGE predictor being the current-state-of-art.

In this work, instead of directly improving on individual predictor's accuracy, I focus on an orthogonal statistical method called bootstrap aggregating, or bagging. Bagging is used to improve overall accuracy by using an ensemble of predictors, which are trained with slightly different data sets. Each predictor (can be same or different predictors) is trained using a resampled (with replacement) training set (bootstrapping). Then, the final prediction is simply provided by weighting or majority voting (aggregating). This work shows that applying bagging improves performance more than simply increasing the size of the predictor.

ACKNOWLEDGEMENT

First, special thanks go to my advisor, Resit Sendag, for convincing me to go to graduate school, and for guidance and advice he has provided. I would also like to thank the rest of my committee; Jien-Chung Lo and Lutz Hamel; both for the valuable feedback they provided on my thesis and for all that I have learned from them in and out of classes.

Also I would like to thank to Professor Oguz Ergin, as well as essentially teaching me everything I know about computer architecture, has been a good friend.

Finally, I would also like to thank my parents, and my sister. They were always supporting me and encouraging me with their best wishes.

TABLE OF CONTENTS

ABSTRACT ii

ACKNOWLEDGEMENT iii

TABLE OF CONTENTS iv

LIST OF TABLES v

LIST OF FIGURES vi

CHAPTER 1 1

INTRODUCTION 1

BRANCH PREDICTOR 3

STATIC BRANCH PREDICTOR 5

DYNAMIC BRANCH PREDICTOR 6

BRANCH MISPREDICTION RECOVERY 12

TWO LEVEL BRANCH PREDICTOR 14

GSHARE 20

CHAPTER 2 22

ADVANCED BRANCH PREDICTORS 22

TAGE 22

OH-SNAP 25

BOOTSTRAP AGGREGATING BRANCH PREDICTOR 29

CHAPTER 3 33

METHODOLOGY 33

CHAPTER 4 36

RESULTS 36

CHAPTER 5 45

CONCLUSION 45

BIBLIOGRAPHY 46

LIST OF TABLES

TABLE PAGE

Table 1. Characteristics of the CBP-4 traces. 35

Table 2. Base Simulation Results for Used Branch Predictors 37

Table 3. Effect of Increasing Size of TAGE Predictor 38

Table 4. Correlation between Number of Tables and Max History Length in TAGE Predictor 40

Table 5. Sub-predictor Configuration 43

Table 6. OH-SNAP Bagging Results 44

LIST OF FIGURES

FIGURE PAGE

Figure 1. Branch Target Speculation Using a Branch Target Buffer. 4

Figure 2. FSM Model for History-Based Branch Direction Predictors. 7

Figure 3. History-Based Branch Predictor 7

Figure 4. FSM Model for History-Based Branch Direction Predictors 10

Figure 5. Two Aspects of Branch Prediction 12

Figure 6. Two-level Adaptive Branch Prediction of Yeh and Patt. 16

Figure 7. Correlated Branch Predictor with Global BHSR and Shared PHTs. 18

Figure 8. Correlated Branch Predictor with Individual BHSRs and Shared PHTs.....19

Figure 9. The gshare Correlated Branch Predictor 21

Figure 10. A 5-component TAGE predictor 23

Figure 11. Perceptron Prediction and Training. 25

Figure 12. OH-SNAP Data Path. 27

Figure 13. Offline Bagging 30

Figure 14. Online Bagging 31

Figure 15. Comparison of Different Bagging Configurations 41

CHAPTER 1

INTRODUCTION

Accurate branch prediction can be seen as a mechanism for enabling design decisions. When short pipelines were the norm, accurate branch prediction was not as important. However, having accurate branch prediction enables technologies like wide-issue deeply pipelined superscalar processors. If branch predictors can be improved further, we can more successfully use more aggressive speculation techniques. Accurate branch prediction enables larger scheduling windows, out-of—order execution, deeper pipelines etc.

A pipelined machine achieves its maximum throughput when it is in the streaming mode. For the fetch stage, streaming stage implies the continuous fetching of instructions from sequential locations in the program memory. Whenever the control flow of the program deviates from the sequential path, potential disruption to the streaming mode can occur. For unconditional branches, subsequent instructions cannot be fetched until target address of the target address of the branch is determined. For conditional branches, the machine must wait for the resolution of the branch condition, and if the branch is to be taken, it must further wait until the target address is available. Branch instructions are executed by the branch functional unit. For a conditional branch, it is not until it exits the branch unit and when both the branch condition and the branch target address are known that the fetch stage can correctly fetch the next instruction.

This delay in processing conditional branches incurs a cycle penalty in fetching the next instruction, corresponding to the traversal of decode, dispatch, and execute stages by the conditional branch. The actual lost-opportunity cost of stalled cycles is not just empty instruction slots, but the number of empty instruction slots must be multiplied by the width of the machine.

Maximizing the volume of the instruction flow path is equivalent to maximizing the sustained instruction fetch bandwidth. To do this, the number of stall cycles in the fetch stage must be minimized. For an n-wide machine each stalled cycle is equal to fetching n no-op instructions. The primary aim of instruction flow technique is to minimize the number of such fetch stall cycles and/or to make use of these cycles to do potentially useful work. The current dominant approach to accomplishing this is via branch prediction.

TAGE [1] predictor has been widely accepted as the current state-of-the-art in branch prediction. In this work, instead of directly improving predictor's accuracy, I focus on an orthogonal statistical method called bootstrap aggregating, or bagging. Bagging is used to improve overall accuracy by using an ensemble of predictors, which are trained with slightly different data sets. Each predictor (can be same or different predictors) is trained using a resampled (with replacement) training set (bootstrapping). Then, the final prediction is simply provided by weighting or majority voting (aggregating). This work shows that applying bagging improves performance more than simply increasing the size of the predictor.

BRANCH PREDICTOR

Experimental studies have shown that the behavior of branch instructions is highly predictable. A key approach to minimizing branch penalty and maximizing instruction flow throughput is to speculate on both branch target address and branch condition of branch instruction. As a static branch instruction is repeatedly executed at run time, its dynamic behavior can be tracked. Based on its past behavior, its future behavior can be effectively predicted. Two fundamental components of branch prediction are branch target speculation and branch condition speculation. With any speculative technique, there must be mechanisms to validate the prediction and to safely recover from any misprediction.

Branch target speculation involves the use if a branch target buffer (BTB) to store previous branch target address. BTB is a small cache memory accessed during the instruction fetch stage using the instruction fetch address (PC). Each entry of BTB contains two fields: the branch instruction address (BIA) and the branch target address (BTA). When a static branch instruction is executed for the first time, an entry in the BTB is allocated for it. Its instruction address is stored in the BIA field, and its target is stored in the BTA field. Assuming the BTB is fully associative cache, BIA field is used for the associative access of the BTB. The BTB is accessed concurrently with the accessing of the I-cache. When the current PC matches the BIA of an entry in the BTB, a hit in the BTB results. This implies that the current instruction being fetched from the I-cache has been executed before and is a branch instruction. As shown in Figure 1, when a hit in the BTB occurs, the BTA field of the hit entry is accessed and

can be used as the next instruction fetch address if that particular branch instruction is predicted to be taken.

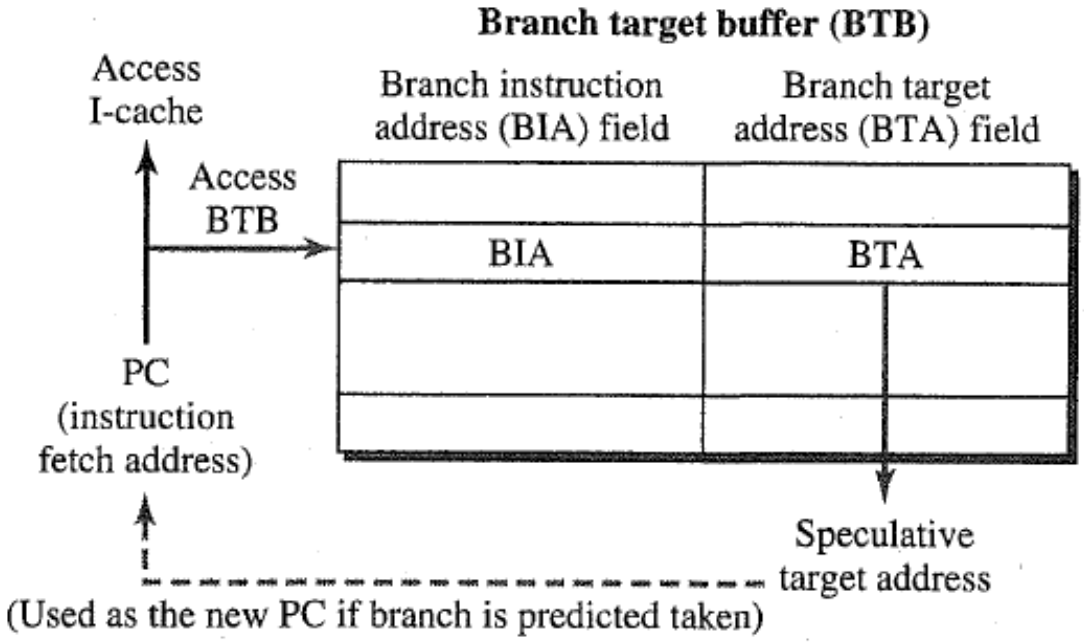

Figure 1: Branch Target Speculation Using a Branch Target Buffer

By accessing the BTB using the branch instruction address and retrieving branch target address from BTB all during the fetch stage, the speculative branch address will be ready to be used in the next machine cycle as the new instruction fetch address if the branch instruction is predicted to be taken. If the branch instruction is predicted to be taken and this prediction turn out to be correct, then branch instruction is effectively executed in the fetch stage, incurring no branch penalty. The nonspeculative execution of the branch instruction is still performed for the purpose of validating the speculative execution. The branch instruction is still fetched from the I-cache and executed. The resultant target address and branch condition are compared with speculative version. If they agree, then correct prediction was made; otherwise,

misprediction has occurred and recovery must be initiated. The result from nonspeculative execution is also used to update the content, i.e., the BTA field, of the BTB.

STATIC BRANCH PREDICTOR

There are a number ways to do branch condition speculation. The simplest form is to design the fetch hardware to be biased for not taken, i.e., to always predict not taken. When a branch instruction is encountered, prior to its resolution, the fetch stage continuous fetching down the fall-through path without stalling. This form of minimal branch prediction is easy to implement but not very effective. For example, many branches are used for loop closing instructions, which are mostly taken during execution except when exiting loops. Another form of prediction employs software support and can require ISA changes. For example, an extra bit can be allocated in branch instruction format that is set by the compiler. This bit is used as a hint to hardware to perform either predict not taken or predict taken depending on the value of this bit. The compiler can use branch instruction type and profiling information to determine the most appropriate value for this bit. This allows each static branch instruction to have its own specified prediction. However, this prediction is static in the sense that the same prediction is used for all dynamic executions of the branch. A more aggressive and dynamic form of prediction makes prediction based on the branch target address offset. This form of prediction first determines the relative offset between the address of the branch of the instruction and the address of the target instruction. A positive offset will trigger the hardware to predict not taken, whereas a

negative offset, most likely indicating a loop closing branch, will trigger the hardware to predict taken. The most common branch condition speculation technique employed in contemporary superscalar machines is based on history of previous branch executions.

DYNAMIC BRANCH PREDICTOR

History-based branch prediction makes a prediction of branch direction, whether taken (T) or not taken (N), based on previously observed branch directions. The assumption is that historical information on the direction that a static branch takes in previous executions can give helpful hints on the direction that is likely to taken in future executions. Design decisions for such type of branch prediction include how much history should be tracked and for each observed history pattern what prediction should be made. As shown in Figure 2, the specific algorithm for history-based branch direction prediction can be characterized by a finite state machine (FSM). The n state variable encode the directions taken by the last n executions of that branch. Hence each state represents a particular history in terms of a sequence of takens and not takens. The output logic generates a predictions made based on the outcome of the previous n executions of that branch. When a branch finally executed, the actual outcome is used as an input to the FSM to trigger a state transition. The next state logic is trivial; it simply involves chaining the state variables into a shift register, which records the branch directions of the previous n executions of that branch instructions.

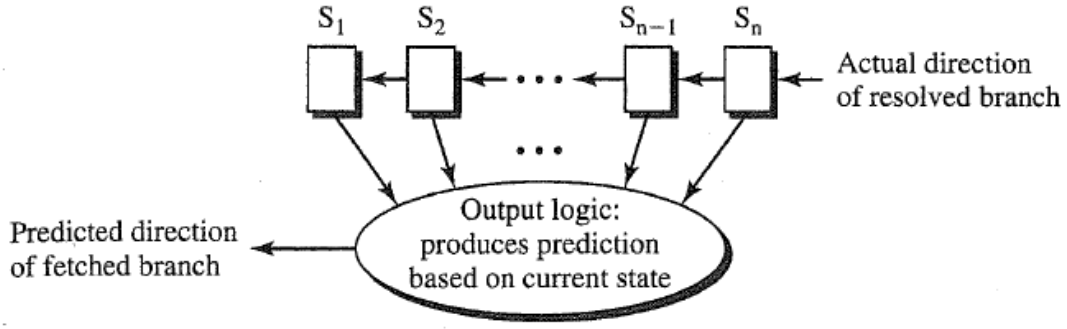

Figure 2: FSM Model for History-Based Branch Direction Predictors

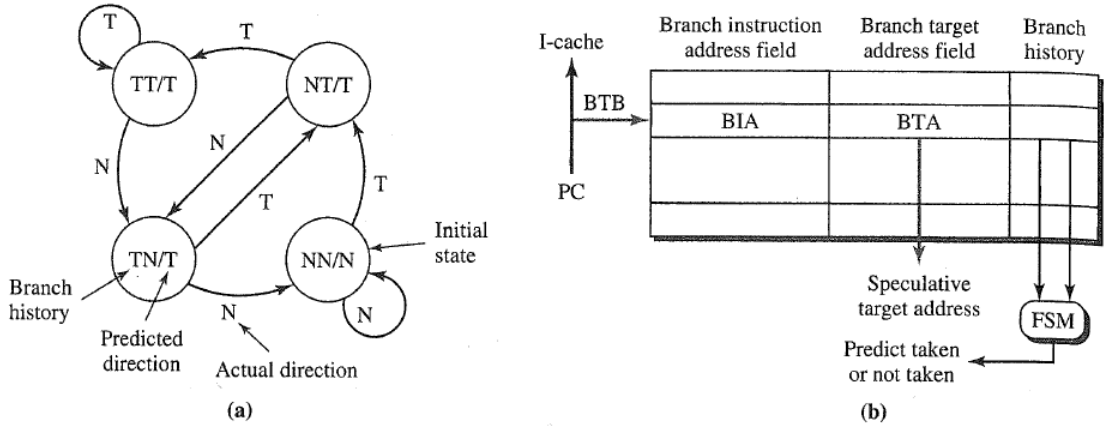

Figure 3: History-Based Branch Predictor: (a) A 2-bit Branch Predictor Algorithm; (b) Branch Target Buffer with an Additional Field for Storing Branch History Bits.

Figure 3 illustrates the FSM diagram of a typical 2-bit branch predictor that employs two history bits to track the outcome of two previous executions of the branch. The two history bits constitute the state variables of the FSM. The predictor can be in one of four states: NN, NT, TT, or TN, representing the directions taken in previous two executions of the branch. The NN state can be designated as the initial

state. An output value either T or N is associated with each of the four states representing the prediction that would be made when a predictor is in that state. When a branch is executed, the actual direction taken is used as an input to the FSM, and a state transition occurs to update the branch history which will be used to do the next prediction. The particular algorithm implemented in the predictor of Figure 3 is a biased toward predicting branches to be taken; note that three of the four states predict the branch to be taken. It anticipates either long runs of N's (in the NN state) or long runs of T's (in the TT state). As long as at least one of the two previous executions was a taken branch, it will predict the next execution to be taken. The prediction will only be switched to not taken when it has encountered two consecutive N's in a row. This represents one particular branch prediction algorithm; clearly there are many possible designs for such history-based predictors, and many designs there have been evaluated by researchers.

To support history-based branch direction predictors, the BTB can be augmented to include a history field for each of its entries. The width, in number of bits, of this field determined by number of history bits being tracked. When a PC address hits in the BTB, in addition to the speculative target address, the history bits are retrieved. These history bits are fed to the logic that implements the next state and output functions of the branch predictor FSM. The retrieved history bits, the output logic produces the 1-bit output that indicates the predicted direction. If the prediction is a taken branch, then this output is used as the new instruction fetch address in the next machine cycle. If the prediction turns the prediction turns out to be correct, then

effectively the branch instruction has been executed in the fetch stage without incurring any penalty or stalled cycle.

A classic experimental study on branch prediction was done by Lee and Smith [2]. In this study, 26 programs from six different types of workloads for three different machines were used. Averaged across all the benchmarks, 67.6% of the branches were taken while 32.4% were not taken. Branches tend to be taken more than not taken by a ratio of 2 to 1. With static branch prediction based on the op-code type, the prediction accuracy ranged from 55% to 80% for six workloads. Using only 1 bit of history, history-based dynamic branch prediction achieved prediction accuracies ranging from 79.7% to 96.5%. With 2 history bits, the accuracies for the six workloads ranged from 83.4% to 97.5%. Continued increase of the number of history bits brought additional incremental accuracy. However, beyond four history bits there is a very minimal increase in the prediction accuracy with the BTB hit rate, the resultant average prediction effectiveness was approximately 80%.

Another experimental study was done in 1992 at IBM by Ravi Nair using Systems Performance Evaluation Cooperative (SPEC) benchmarks [3]. This was a very comprehensive study of possible branch prediction algorithms. The goal for branch prediction is to overlap the execution of branches or accomplish branch folding; i.e., branches are folded out of the critical latency path of instruction execution. This study performed an exhaustive search for optimal 2-bit predictors. There are 2^{20} possible FSMs of 2-bit predictors. Nair determined that many of these machines are uninteresting and pruned the entire design space down to 5248 machines. Extensive simulations are performed to determine the optimal (achieves best

prediction accuracy) 2-bit predictor for each of the benchmarks. The list of SPEC benchmarks, their prediction accuracies, and the associated optimal predictors are shown in Figure 4.

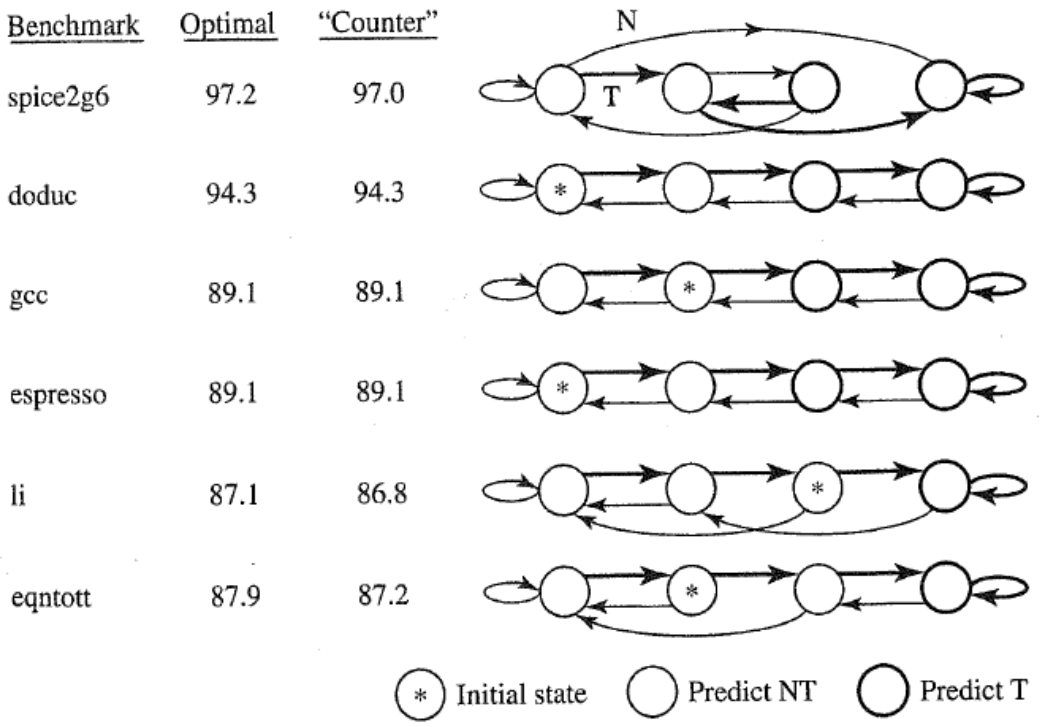

Figure 4: FSM Model for History-Based Branch Direction Predictors

In Figure 4, the states denoted with bold circles represent states which the branch is predicted taken; to nonbold circles represent states that predict not taken. Similarly the bold edges represent state transitions when the branches is actually taken; the nonbold edges represent transitions corresponding to the branch actually not taken. The state denoted with asterisk indicates the initial state. The prediction accuracies for optimal predictors of these six benchmarks range from 87.1% to 97.2%. Notice that optimal predictors for doduc, gcc, and espresso are identical (disregarding the different initial state of the gcc predictor) and exhibit the behavior of a 2-bit

up/down saturating counter. We can label the four states from left to right as 0, 1, 2, and 3, representing the four count values of a 2-bit counter. Whenever a branch is resolved taken, the count is incremented; and it is decremented otherwise. Two lower-count states predict a branch to be not taken, while other two higher-count states predict a branch taken. Figure 4 also provides the prediction accuracies for the six benchmarks. The prediction accuracies for the six benchmarks if the 2-bit saturating counter predictor is used for all six benchmarks. The prediction accuracies for spice2g6, li, and eqntott only decreased minimally from optimal values, indicating that the 2-bit saturating counter, originally invented by Jim Smith, has become a popular prediction algorithm in real and experimental designs.

The same study by Nair also investigated the effectiveness of counter-based predictors. With 1-bit counter as predictor, i.e., remembering the direction taken last time and predicting the same direction for the next time, the prediction accuracies range of 82.5% to 96.2%. As I shown in Figure 5, a 2-bit counter yields an accuracy range of 86.8% to 97.0%. If a 3-bit counter is used, the increase in accuracy is minimal; accuracies range from 88.3% to 97.0%. Based on this study, the 2-bit saturating counter appears to be a very good choice for a history-based predictor. Direct-mapped branch history tables are assumed in this study. While some programs, such as gcc, have more than 7000 conditional branches, for most programs, the branch penalty due to aliasing in finite-sized branch history tables levels out at about 1024 entries for table size.

BRANCH MISPREDICTION RECOVERY

Branch prediction is a speculative technique. Any speculative technique requires mechanisms for validating the speculation. Dynamic branch prediction can be viewed as consisting of two interacting engines. The leading engine performs validation in the later stages of the pipeline. In the case of misprediction the trailing engine also performs recovery. These two aspects of branch prediction are illustrated in Figure 5.

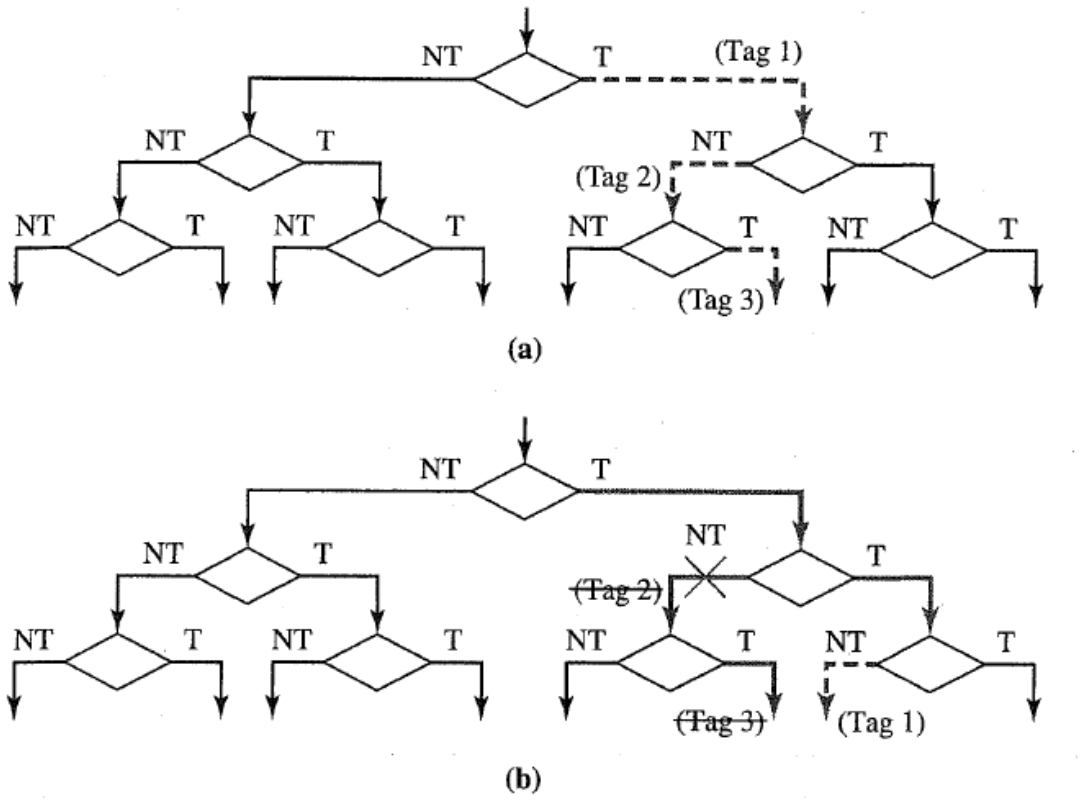

Figure 5: Two Aspects of Branch Prediction: (a) Branch Speculation; (b) Branch Validation and Recovery.

Branch speculation involves predicting the direction of a branch and then proceeding to fetch along the predicted path of control flow. While fetching from the predicted path, additional branch introductions may be encountered. Prediction of these additional branches can be similarly performed, potentially resulting in speculating past multiple conditional branches before the first speculated branch is resolved. Figure 5 illustrates speculating past three branches with the first and third branches being predicted taken and second one predicted not taken. When this occurs, instructions from three speculative basic blocks are now resident in the machine and must be appropriately identified. Instructions from each speculative basic block are given the same identifying tag. In the example of Figure 5, three distinct tags are used to identify the instructions from the three speculative basic blocks. A tagged instruction indicates that it is a speculative instruction, and the value of the tag identifies which basic block it belongs to. As a speculative instruction advances down the pipeline stages, tag is also carried along. When speculating, the instruction addresses of all the speculated branch instructions (or the next sequential instructions) must be buffered in the event that recovery is required.

Branch validation occurs when the branch is executed and the actual direction of a branch is resolved. The correctness of the earlier prediction can then be determined. If the prediction turns out to be correct, the speculation tag is deallocated and all instructions associated with that tag becomes nonspeculative and are allowed to complete. If a misprediction is detected, two actions are required; namely, the incorrect path must be terminated, and fetching from new correct path must be initiated. To initiate a new path, the PC must be updated with a new instruction fetch

address. If the incorrect prediction was a not-taken prediction, then PC is updated with the computed branch target address. If the incorrect prediction was a taken prediction, then the PC is updated with sequential (fall-through) instruction address, which is obtained from the previously buffered instruction address when the branch was predicted taken. Once the PC has been updated, fetching of the instructions resumes along the new path, and branch prediction begins anew. To terminate the incorrect path, speculation tags are used. All the tags that are associated with mispredicted branch are used to identify the instructions that must be eliminated. All such instructions that are still in decode and dispatch buffers as well as those in reservation station entries are invalidated. Reorder buffer entries occupied by these instructions are deallocated. Figure 5 illustrates this validation/recovery task when the second of the three predictions is incorrect. The first branch is correctly predicted, and therefore instructions with Tag 1 becomes nonspeculative and are allowed to complete. The second prediction is incorrect, and all the instructions with Tag 2 and Tag 3 must be invalidated and their buffer entries must be deallocated. After fetching down the correct path, branch prediction can begin once again, Tag 1 is used again to denote the instruction in the first speculative basic block. During branch validation, the associated BTB entry is also updated.

TWO LEVEL BRANCH PREDICTOR

The dynamic branch prediction schemes discussed thus far have a number of limitations. Prediction for a branch is made based on the limited history of only that particular static branch instructions. The actual prediction algorithm does not take into

account the dynamic context within which the branch is being executed. For example, it does not make use of any information on the particular flow path taken in arriving at that branch. Furthermore the same fixed algorithm is used to make the prediction regardless of dynamic context. It has been observed experimentally that the behavior of certain branches is strongly correlated with the behavior of other branches that precede them during execution. Consequently more accurate branch prediction can be achieved with algorithms that take into account the branch history of other correlated branches and that can adapt the prediction algorithm to the dynamic branching context.

In 1991, Yeh and Patt proposed a two-level adaptive branch prediction technique that can potentially achieve better than 95% prediction accuracy by having a highly flexible prediction algorithm that can adapt to changing dynamic contexts [4]. In previous schemes, a single branch history table is used and indexed by the branch address. As shown in Figure 6, in the two-level adaptive scheme, a set of history tables is used. These are identified as the pattern history table (PHT). Each branch address indexes to a set of relevant entries; one of these entries is then selected based on the dynamic branching context. The context is determined by a specific pattern of recently executed branches stored in a branch history shift register (BHSR). The content of the BHSR is used to index in to the PHT to select one of the relevant entries. The content of this entry is then used as the state for the prediction algorithm FSM to produce a prediction. When a branch is resolved, the branch result is used to update both the BHSR and selected entry in the PHT.

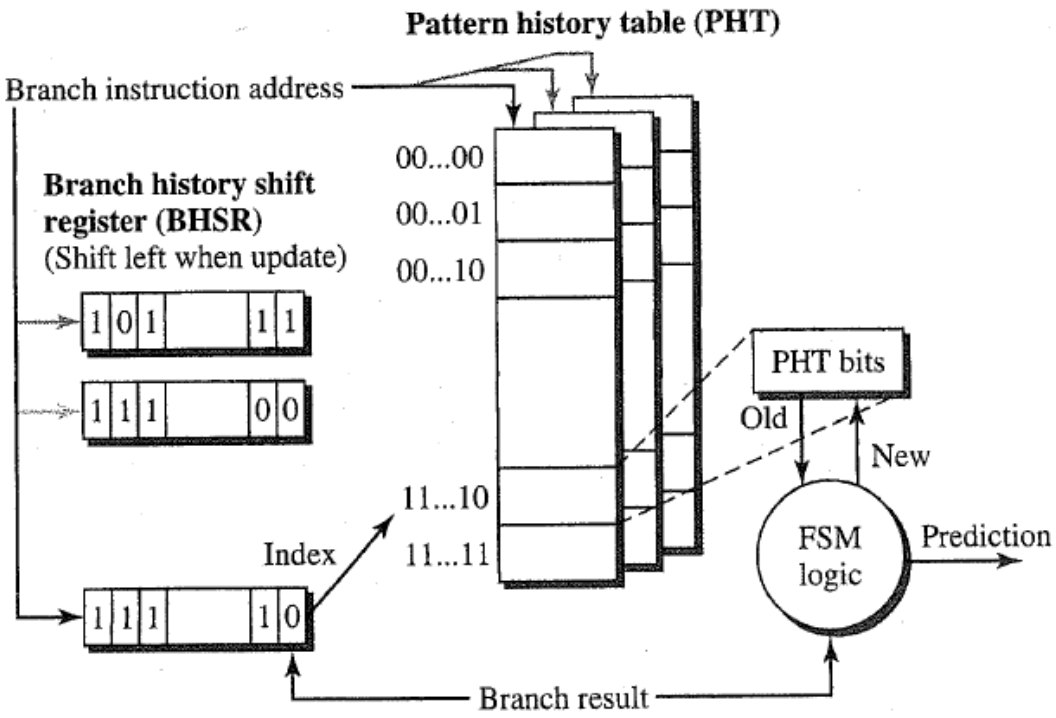

Figure 6: Two-level Adaptive Branch Prediction of Yeh and Patt

The two-level adaptive branch prediction technique actually specifies a framework within which many possible designs can be implemented. There are two options to implementing the BHSR: global (G) and individual (P). The global implementation employs a single BHSR of k bits that tracks the branch directions of the last k dynamic branch instructions in program executions. These can involve any number (1 to k) of static branch instructions. The individual implementation employs a set of k-bit BHSRs as illustrated in Figure 6, one of which is selected based on branch address. Essentially the global BHSR is shared by all static branches, whereas with individual BHSRs each BHSR is dedicated to each static branch or a subset of static branches if there is address aliasing when indexing into the set of BHSRs using the branch address. There are three options to implement the PHT global (g), individual

(p), or shared (s). Alternatively, individual PHTs can be used in which each PHT is dedicated to each static branch (p) or a small subset of static branches (s) if there is address aliasing when indexing into the set of PHTs using the branch address. A third dimension to this design space involves the implementation of the actual prediction algorithm. When a history-based FSM is used to implement the prediction algorithm, Yeh and Patt identified such schemes as adaptive (A).

All possible implementations of the two-level adaptive branch prediction can be classified based on these three dimensions of design parameters. A given implementation can then be denoted using a three-letter notation: e.g., Gas represents a design that employs a single global BHSR, an adaptive prediction algorithm, and a set of PHTs with each being shared by a number of static branches. Yeh and Patt presented three specific implementations that are able to achieve a prediction accuracy 97% for their given set of benchmarks:

- GAg: (1) BHSR of size 18 bits; (1) PHT of size 2^{18} x 2 bits.
- PAg: (512 x 4) BHSRs of size 12 bits; (1) PHT of size 2^{12} x 2 bits.
- PAs: (512 x 4) BHSRs of size 6 bits; (512) PHT of size 2^{6} x 2 bits.

All three implementations use an adaptive (A) predictor that is a 2-bit FSM. The first implementation employs a global BHSR (G) of 18 bits and a global PHT (g) with 218 entries indexed by the BHSR bits. The second implementation employs 512 sets (four-way set-associative) of 12-bit BHSRs (P) and a global PHT (g) with 212 entries. The third implementation also employs 512 set of four-way set-associative BHSRs (P), but each is only 6 bits wide. It also uses 512 PHTs (s), each having 26 entries indexed by the BHSR bits. Both the 512 sets of BHSRs and the 512 PHTs are indexed using 9 bits

of the branch address. Additional branch address bits are used for the set-associative access of the BHSRs. The 512 PHTs are direct-mapped, and there can be aliasing, i.e., multiple branch addresses sharing the same PHT. From experimental data, such aliasing had minimal impact on degrading the prediction accuracy. Achieving greater than 95% prediction accuracy by the two-level adaptive branch prediction schemes is quite impressive; the best traditional prediction techniques can only achieve about 90% prediction accuracy.

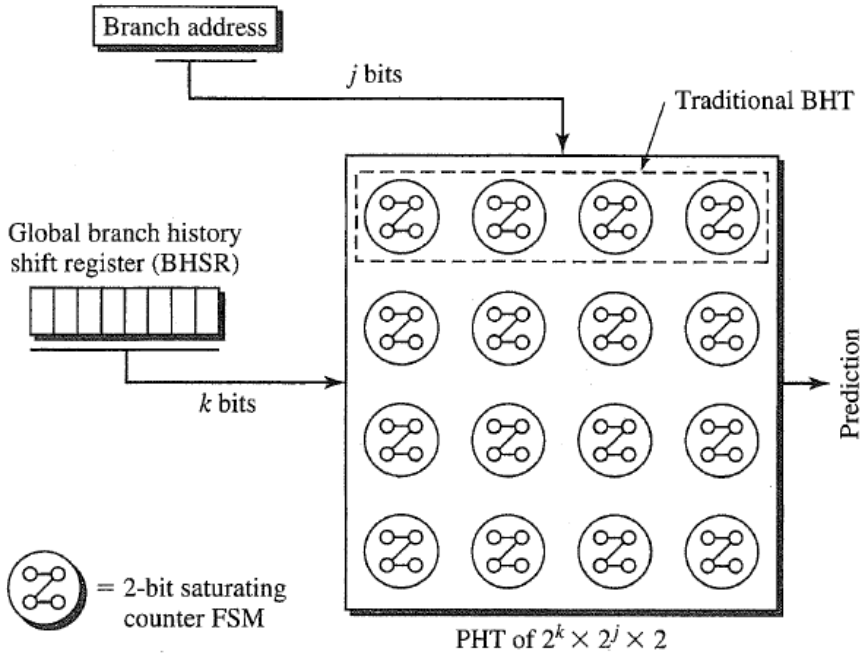

Figure 7: Correlated Branch Predictor with Global BHSR and Shared PHTs (GAs).

Following the original Yeh and Patt proposal, other studies by McFarling [5], Young and Smith [6], and Gloy et al. [7] have gained further insights into two-level adaptive, or more recently called correlated, branch predictors. Figure 7 illustrates a correlated branch prediction with a global BHSR (G) and a shared PHT (s). The 2-bit

saturating counter is used as the predictor FSM. The global BHSR tracks the directions of the last k dynamic branches and captures the dynamic control flow context. The PHT can be viewed as a single table containing a two-dimensional array, with 2^j columns and 2^k rows, of 2-bit predictors. If the branch address has n bits, a subset of j bits is used to index into the PHT to select one of the 2^j columns. Since j is less than n, some aliasing can occur where two different branch addresses can index into the same column of the PHT. Hence the designation of shared PHT. The k bits from the BHSR are used to select one of the 2^k entries in the selected column. The 2 history bits in the selected entry are used to make a history-based prediction. The traditional branch history table is equivalent to having only one row of the PHT that is indexed only by the j bits of the branch address, as illustrated in Figure by the dashed rectangular block of 2-bits predictors in the row of the PHT.

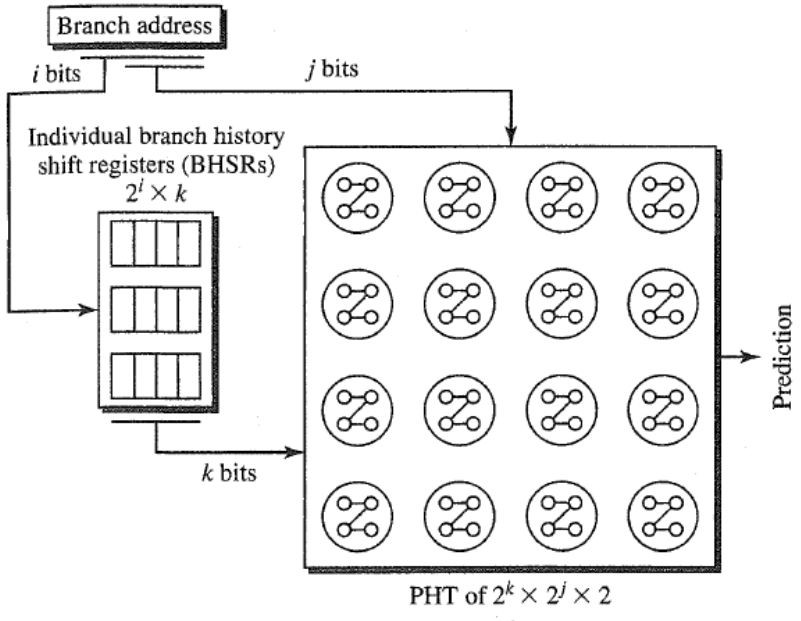

Figure 8: Correlated Branch Predictor with Individual BHSRs and Shared PHTs (PAs).

Figure 8 illustrates a correlated branch predictor with individual BHSRs (P) and the same shared PHT (s). Similar to the GAs scheme, the PAs scheme also uses j bits of the branch address to select one of the 2^j columns of the PHT. However, i bits of the branch address, which can overlap with the j bits used to access the PHT, are used to index into a set of BHSRs. Depending on the branch address, one of the 2^i BHSRs is selected. Hence, each BHSR is associated with one particular branch address, or a set of branch addresses if there is aliasing. Essentially, instead of using a single BHSR to provide the dynamic control flow context for all static branches, multiple BHSRs are used to provide distinct dynamic control flow contexts for different subsets of static branches. This adds flexibility in tracking and exploiting correlations between different branch instructions. Each BHSR tracks the directions of the last k dynamic branches belonging to the same subset of static branches. Both the GAs and the PAs schemes require a PHT of size 2^k x 2^j x 2 bits. The GAs scheme has only one k-bit BHSR whereas the PAs scheme requires 2^i k-bit BHSRs.

GSHARE

A fairly efficient correlated branch predictor called gshare was proposed by Scott McFarling [5]. In this scheme which is shown in Figure 9, j bits from the branch address are hashed (via bitwise XOR function) with k bits from global BHSR. The resultant max {k, j} bits are used to index into a PHT of size $2^{\max\{k, j\}}$ x 2 bits to select one of the 2max {k, j} 2-bit branch predictors. The gshare scheme requires only one k-

bit BHSR and a much smaller PHT, yet archives comparable prediction accuracy to other correlated branch predictors.

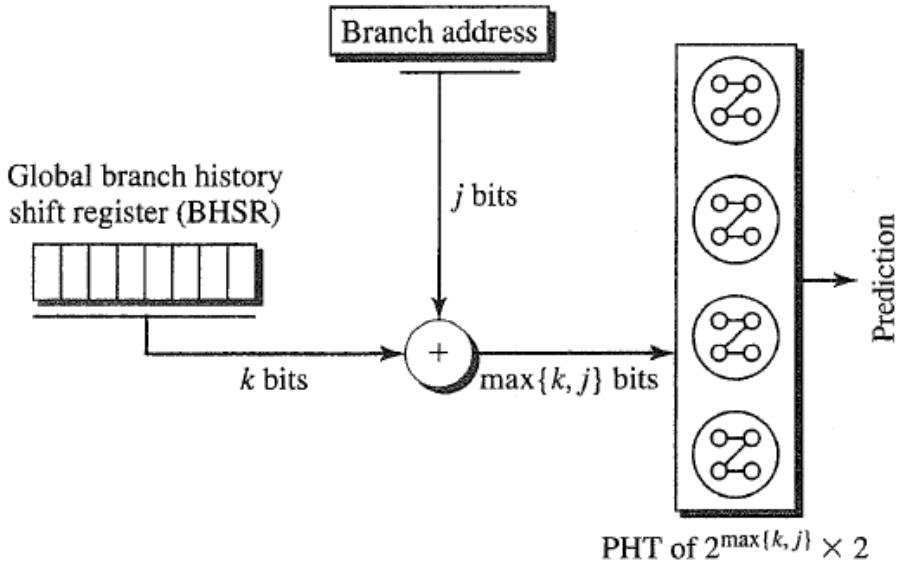

Figure 9: The gshare Correlated Branch Predictor

CHAPTER 2

ADVANCED BRANCH PREDICTORS

In this chapter I will explain bootstrap aggregating branch predictors. But before that I need to explain which predictors are used in bagging. These predictors are TAGE, and OH-SNAP, which are more complex and better performing than ones described in Chapter 1.

TAGE

TAGE stands for Tagged Geometric history length. It relies on a default tagless predictor backed with plurality of tagged predictor components indexed using different history lengths for index computation. These history lengths form a geometric series. The prediction is provided either by a tag match on a tagged predictor component or by the default predictor. In case of multiple hits, the prediction is provided by the tag matching table the longest history.

Geometric history length prediction was introduced with the O-GEHL predictor [8]. The predictor features M distinct predictor tables Ti, $0 \leq i \leq M$ indexed with hash functions of the branch address and the global branch history. Distinct history lengths are used for computing the index of the distinct tables. Table T0 is indexed using the branch address. The history lengths used for computing the indexing functions for tables Ti, $1 \leq i \leq M$ are of the form $L(i) = \alpha^{i-1} * L(1)$, i.e., the lengths L(i) form a geometric series. More precisely, as history lengths are integers, it uses $L(i) = (int)\ (\alpha^{i-1} * L(1)$.

Using a geometric series of history lengths allows to use very long history lengths for indexing some predictor tables, while still dedicating most of the storage space to predictor tables using short global history lengths. As an example on a 8-component predictor, using $\alpha = 2$ and $L(1) = 2$ leads to the following series {0, 2, 4, 8, 16, 32, 64, 128}.

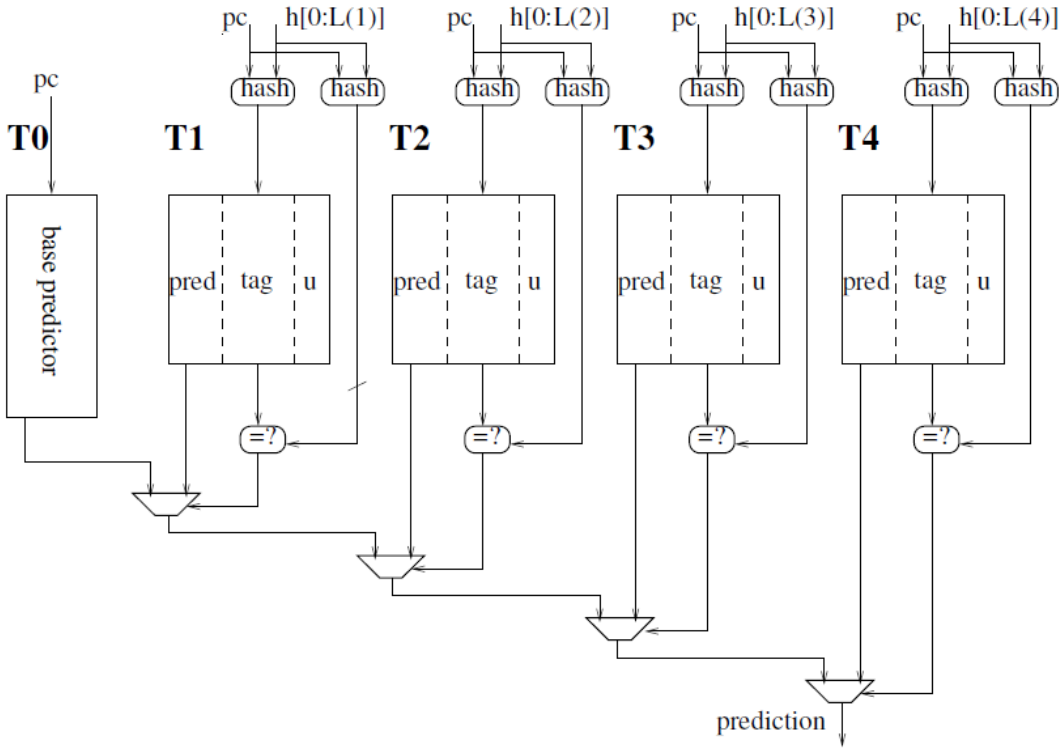

Figure 10: A 5-component TAGE predictor

Figure 10 illustrates a TAGE predictor. The TAGE predictor features a base predictor T0 in charge of providing a basic prediction and a set of tagged predictor components Ti. These tagged predictor components Ti, $1 \leq i \leq M$ are indexed using different history lengths that form a geometric series. The base predictor is a simple PC indexed 2-bit counter bimodal table. An entry in a tagged component consists in a

single counter ctr which sign provides the prediction, a tag and an unsigned useful counter u.

At prediction time, the base predictor and the tagged components are accessed simultaneously. The base predictor provides a default prediction. The tagged components provide a prediction only on a tag match. The prediction is provided by the hitting tagged predictor component that uses the longest history. In case of no matching tagged predictor component, the default prediction use.

The provider component is defined as the predictor component that ultimately provides the prediction. The alternate prediction as the prediction that would have occurred if there had been a miss on the provider component. That is, if there are tag hits on T2 and T4 and tag misses on T1 and T3, T4 is the provider component and T2 provides alternate prediction. If there is no hitting component then alternate prediction is the default prediction.

The useful counter u of the provider component is updated when alternate prediction altpred is different from the final prediction pred. u is incremented when the actual prediction pred is correct and decremented otherwise. Moreover, the useful u counter is also used as an age counter. Here useful counter is 2-bits. Periodically, the whole column of most significant bits of the u counters is reset to zero, then whole column of least significant bits are reset. On correct prediction, the prediction counter of the provider component is updated. On incorrect prediction, first, the provider component prediction counter is updated. As a second step, if provider component Ti is not the component using the longest history, it tries to allocate an entry on a predictor component Tk using a longer history than Ti(i.e., $i < k < M$).

Most proposals for neural branch predictors derive from the perceptron branch predictor [9]. A perceptron is a vector of h + 1 small integer weights, where h is the history length of the predictor. As shown in Figure 11, a table of n perceptrons is kept in a fast memory. A global history shift register of the h most recent branch outcomes is also kept. The shift register and table of perceptrons are analogous to the shift register and table of counters in traditional global two-level predictors, since both the indexed counter and the indexed perceptron are used to compute the prediction.

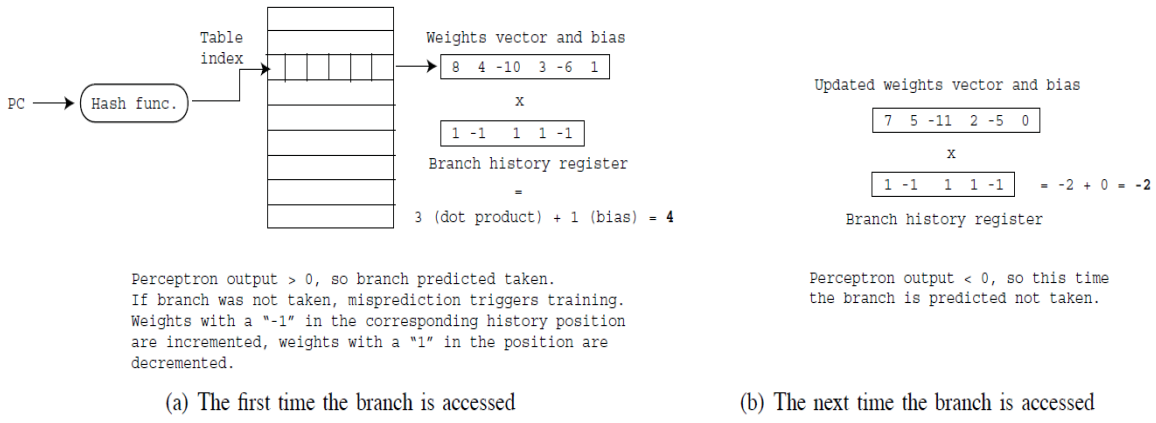

Figure 11: Perceptron Prediction and Training

To predict a branch, a perceptron is selected using a hash function of the branch PC. The output of the perceptron is computed as the dot product of the perceptron and the history shift register, with the not-taken values in the shift registers being interpreted as -1. Added to the dot product is an extra bias weight in the perceptron, which takes into account the tendency of a branch to be taken or not taken without regard for its correlation to other branches. If the dot product result is at least

0, then the branch is predicted taken; otherwise, it is predicted not taken. The magnitude of the weight indicates the strength of the positive or negative correlation. Branch history shift register is speculatively updated which is called ahead pipelining and rolled back on misprediction.

When the branch outcome becomes known, the perceptron that provided the prediction may be updated. The perceptron is trained on a misprediction or a when the magnitude of the perceptron output is below a specified threshold value. Upon training, both the bias weight and the h correlating weights are updated. The bias weight is incremented or decremented if the branch is taken or not taken, respectively. Each correlating weight in the perceptron is incremented if the predicted branch has the same outcome as the corresponding bit in the history register and decremented otherwise with saturating arithmetic. If there is no correlation between the predicted branch and a branch in the history register, the latter's corresponding weight will tend toward 0. If there is high positive or negative correlation, weight will have a large magnitude.

Figure 11 illustrates the concept of a perceptron producing a prediction and being trained. A hash function, based on the PC, accesses the weights table to obtain a perceptron weights vector, which is then multiplied by the branch history, and summed with bias weight to form perceptron output. In this example, the perceptron incorrectly predicts that the branch is taken. The microarchitecture adjusts the weights when it discovers the misprediction. With the adjusted weights, assuming that the history is the same the next time this branch is predicted, the perceptron output is negative, so the branch will be predicted not taken.

OH-SNAP achieves higher accuracies than neural algorithms. The higher accuracy result from accessing the weights using a function of the PC and the path, breaking the weights into a number of independently accessible tables, scaling the weights by the coefficient based on their location on branch history register, and taking the dot product of a modified global branch history vector and the scaled weights. Figure 12 shows a high level diagram of the prediction algorithm and data path.

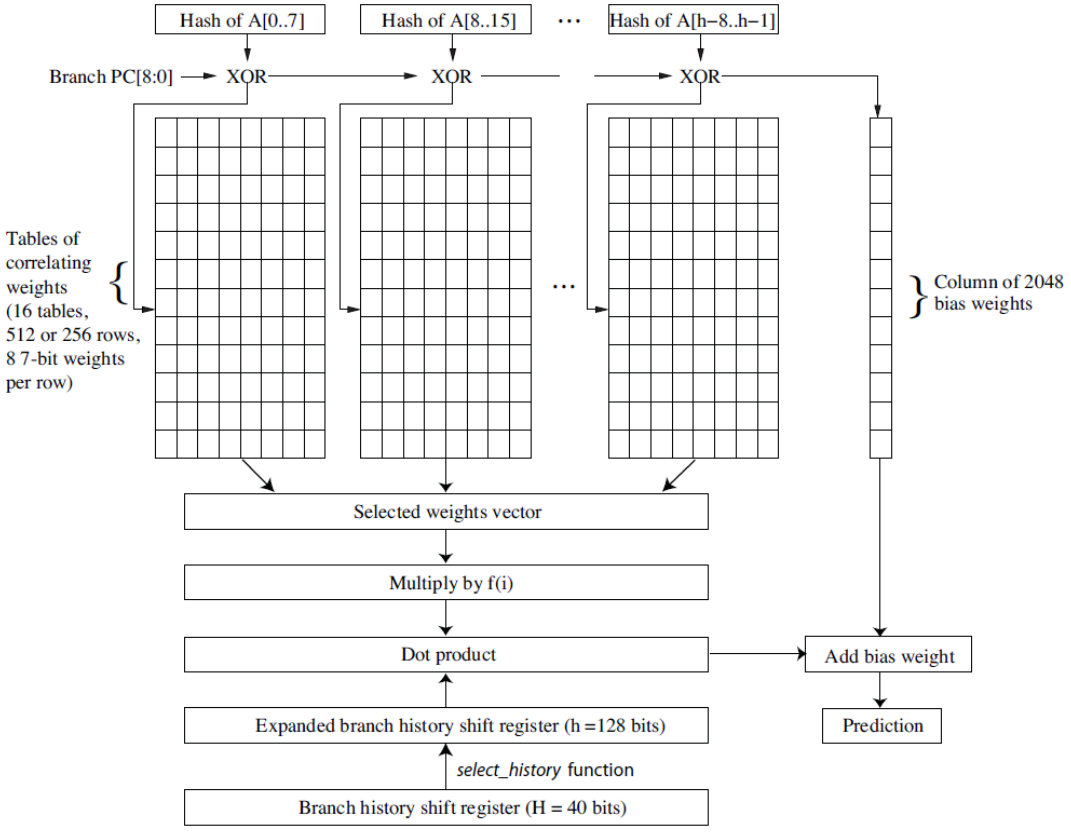

Figure 12: OH-SNAP Data Path

The two key parameters of the predictor are h, the length of the vector with which the dot product is computed, and r, the number of rows in each weight table. In this example, h = 128 and r = 512. Other inputs to the predictor are A, a vector of the low-order bit of each of the past h branch addresses, and H, the global branch history register. The example uses a history register H of 40 bits. The two components of the dot-product computation are the history vector and the weights vector. The history vector consists of h = 128 bit, which is expanded from the 40 bits of H. The use of redundant history can improve prediction accuracy [10], so this predictor replicates the 40 branch history bits to obtain the required 128.

The second component of the dot-product computation, the weights vector, is obtained by reading eight weights from each of 16 tables, as well as a single weight from a table of bias weights. First table, containing the weights for the most recent history bits, has the most entries because the most recent weights are most important. The bias weights table has 2048 entries. In this example, other tables each have 256 entries. The tables are portioned, rather than one large indexed row, because the separation reduces aliasing and achieves higher accuracy.

When the outcome of a branch becomes known, it is shifted into H. The lowest order bit of the branch's address is shifted into A. A high accuracy implementation must keep speculative versions of H and A that are restored on misprediction. If the prediction was incorrect, or if the magnitude of the predictor output was under a set threshold, then the predictor output was under a set threshold, then the predictor invokes it training algorithm. As in neural predictors, the weights responsible for the

output are incremented if the corresponding history outcome matches the current branch outcome, decremented otherwise.

BOOTSTRAP AGGREGATING BRANCH PREDICTOR

Bootstrap aggregating (a.k.a, bagging), introduced by Breiman [11] in 1996, is a meta-algorithm to improve the stability and accuracy of learning algorithms. It has been shown to be very effective in improving generalization performance compared to individual base models [12]. Basic idea behind is by combining many weak learners to produce a strong learner. Bagging is special case of having a hybrid predictor, where predictions from multiple predictors are aggregated using meta-predictors, adder-trees, voting, etc. Bagging works by resampling (with replacement, i.e., some samples may be used more than once) the original training set of size N to produce M bootstrap training sets of size N, each of which is used to train a base model. The predictions by each base model are then aggregated to reach the final prediction. The bagging method is shown in Figure 13. Each predictor's training set contains each of the original training samples K times, where $P(K = k) = \binom{N}{k}\left(\frac{1}{N}\right)^{k}\left(1-\frac{1}{N}\right)^{N-k}$.

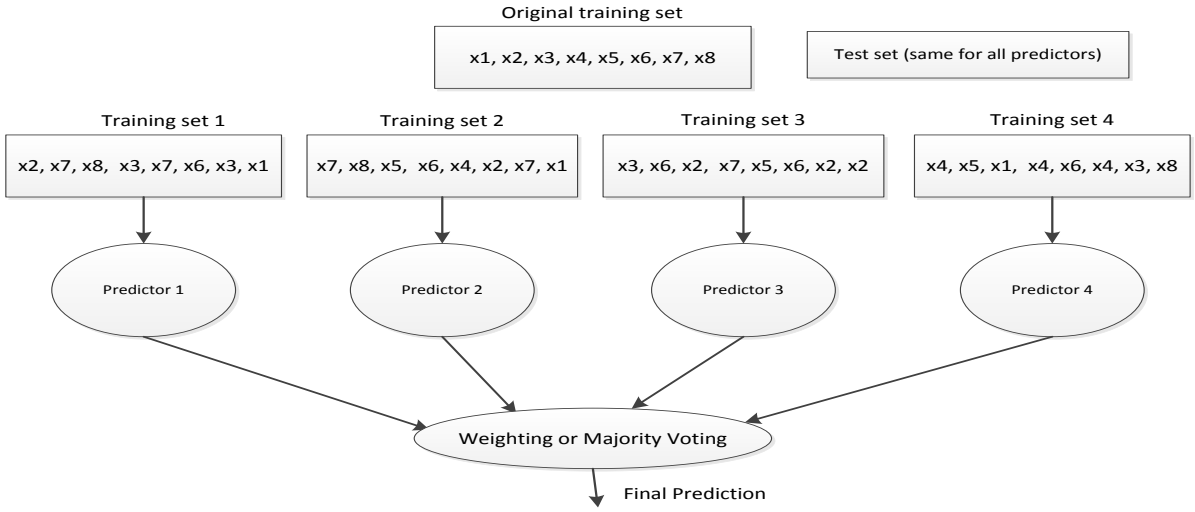

Figure 13: Offline Bagging

In this work, I applied bagging to branch prediction. Because original bagging method is offline – that is, all the training data set must already be available –, I need to develop an online version of bagging. Previous work by Oza and Russel [13] modeled sequential arrival of the data by a Poisson(1) distribution and proved the convergence of this method to offline bagging as $N \rightarrow \infty$. I first used their method in my implementation, which improved performance most of the time. However, I observed that multinomial distribution worked better and hence this method was used in later simulations. The situation is more complicated for branch prediction data because bootstrapping must be carried out in a way that suitably captures the dependence structures for the data. Oza and Russel's [13] method assumed that samples were independent of each other, and thus it does not produce good bootstrapping for branch prediction data. There are studies that developed methods for bootstrapping time series [14], which are better fit for branch prediction. Further research is needed to develop better online bootstrapping methods for branch

prediction or adopt methods from previous work on bootstrapping for time series data, which is left as future work.

In my bagging implementation, each predictor is updated on each sample k times in a row where k is a random number generated by multinomial distribution. I illustrate online bagging in Figure 14.

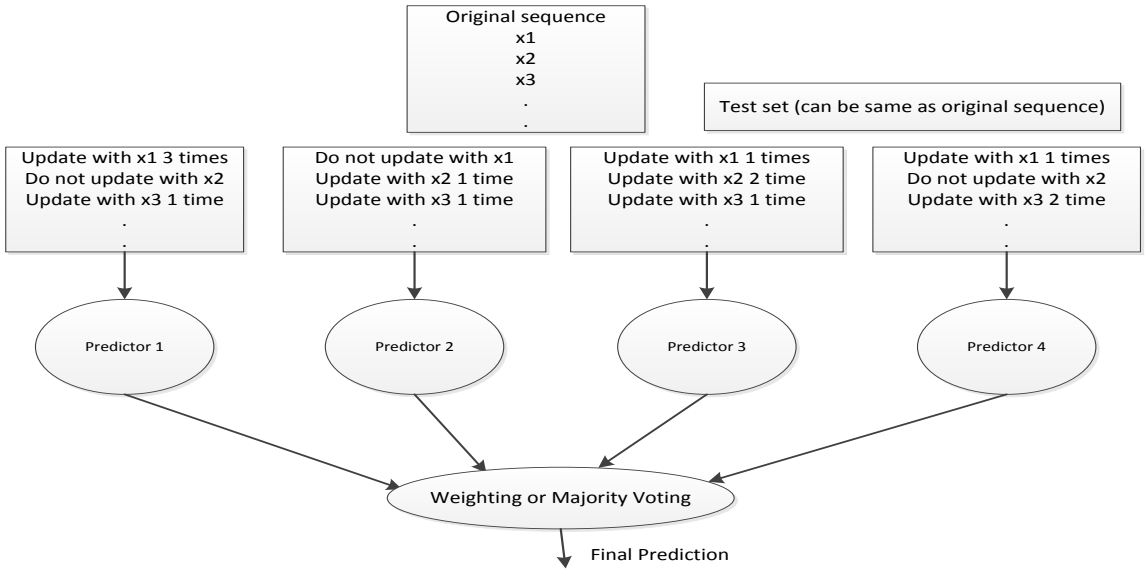

Figure 14: Online Bagging

In general, bagging can be applied to any predictor. Group of same predictors (e.g., a number of TAGE predictors) as well as different predictors may be used.

TAGE bagging (T-BAG) uses a number of TAGE predictors of approximately the same size as sub-predictors. Each sub-predictor provides prediction for the current branch independent of each other. Online bagging is performed by determining whether or not a sub-predictor is updated with the current branch's outcome. Note that this update may occur multiple times for the current branch based on a random number generated. The branch history, however, is always updated as usual.

For final prediction computation, each sub-predictor remembers the success of its last 16 predictions using a sliding window. The number of correct predictions is used as the weight of the sub-predictor. For a not-taken prediction, the weight is taken as negative and for taken predictions it is positive. The overall TAGE bagging prediction is the sign of the sum of the weights, negative being not-taken and it is taken otherwise. This method was slightly better than using majority vote for the final prediction.

In all random updates, RandUpd, simulations, updates are performed randomly for 0, 1, or 2 times in a row for 20%, 60% and 20% of the time, respectively, using trinomial distribution. That is, 60% of the time update is done as usual, 20% of the time no update is performed and 20% of the time update is done twice in a row.

The original TAGE also uses the PC when forming the hashed index for its tagged components. However, because of its operation and its ability to exploit very long history lengths, the PC does not significantly affect performance. In my experiments, the best TAGE configuration using PC in table indexing and the one that does not use PC achieve the similar performance. Therefore, to further increase variability among sub-predictors, some sub-predictors do not use the PC when indexing tagged tables. To the best of my knowledge, no previous work has studied the effects of not using PC in table indexing.

CHAPTER 3

METHODOLOGY

Simulations are done by using publicly available software provided by 4th Championship Branch Prediction (CBP-4) [15]. The goal of CBP-4 is to compare different branch prediction algorithms in a common framework. Contestants are responsible for implementing and evaluating their algorithm in the distributed framework. This is done by modifying a single file to implement predictor class for simulation. All code is written in C++. Framework provides a class template for predictor. Competitors needs to code some set functions. These are;

- PREDICTOR(void);
- bool GetPrediction(UINT32 PC);
- void UpdatePredictor(UINT32 PC, bool resolveDir, bool predDir, UINT32 branchTarget);
- void TrackOtherInst(UINT32 PC, OpType opType, UINT32 branchTarget);

First one is a class constructer. It is used for initializing variables for startup. Second is the prediction function. As an input contestants only allowed to use program counter (pc). Third is for updating the predictor after branch outcome is known. It takes pc as an argument alongside branch outcome (resolveDir), branch prediction made by predictor (predDir), and pc value of next instruction form correct path (branchTarget). Fourth and last mandatory function is for tracking instructions other than branches. It is considered optional as many of branch predictors doesn't keep track of non-branch instructions.

Traces used for evaluation is also provided with CBP-4 framework. There are total of 40 program traces. They are in two categories, long and short. Shorts are further divided into four categories. These are integer, floating point, multimedia, and server. Each of these have five different traces, making in total 20 traces. Long traces are taken from SPEC2006 [16] benchmarks and they are total of 20 traces. SPEC2006 have 31 benchmarks and CBP-4 didn't provide information about which of them are used for making these traces. Shorts traces are 30 million instruction long, and long traces are 150 million instruction long. 30 million instruction traces are of the considered as short traces for branch prediction studies. However 30 million instructions represent approximately the workload that is executed by a PC in 10 millisecond, i.e., the OS time slice. The evaluation metric used by CBP is misprediction per kilo instructions (misp/KI). The characteristics of the traces are summarized in Table 1.

Table 1: Characteristics of the CBP-4 traces

	NUMBER OF INSTRUCTIONS	CONDITIONAL BRANCHES	UNCONDITIONAL BRANCHES
SHORT-FP-1	29499988	2213673	259086
SHORT-FP-2	29499869	1792835	12168
SHORT-FP-3	29499978	1546797	20701
SHORT-FP-4	29499976	895842	17707
SHORT-FP-5	29499969	2422049	175239
SHORT-INT-1	29499987	4184792	576698
SHORT-INT-2	29499985	2866495	577615
SHORT-INT-3	29499978	3771697	336363
SHORT-INT-4	29499960	2069894	221596
SHORT-INT-5	29499990	3755315	46121
SHORT-MM-1	29499979	2229289	410598
SHORT-MM-2	29499970	3809780	294136
SHORT-MM-3	29499970	3014850	1112543
SHORT-MM-4	29499993	4874888	131433
SHORT-MM-5	29499791	2563897	537772
SHORT-SERV-1	29499316	3660616	1253826
SHORT-SERV-2	29499198	3537562	1236437
SHORT-SERV-3	29499817	3811906	1100627
SHORT-SERV-4	29498081	4266796	1381876
SHORT-SERV-5	29497759	4291964	1452124
LONG-SPEC2K6-00	149970336	25181955	6029289
LONG-SPEC2K6-01	150000004	25323638	2192945
LONG-SPEC2K6-02	149999988	22628704	7937909
LONG-SPEC2K6-03	150000001	16754009	324425
LONG-SPEC2K6-04	150000004	31520616	4688658
LONG-SPEC2K6-05	150000001	9409564	1495445
LONG-SPEC2K6-06	150000001	27139020	5521536
LONG-SPEC2K6-07	150000106	23532921	3393843
LONG-SPEC2K6-08	149999996	14565465	4445841
LONG-SPEC2K6-09	149999993	20449090	1343335
LONG-SPEC2K6-10	150000002	14312999	6528434
LONG-SPEC2K6-11	150000001	16145141	373115
LONG-SPEC2K6-12	150000008	19679814	173822
LONG-SPEC2K6-13	149999996	27946011	4967261
LONG-SPEC2K6-14	150000001	29462517	46
LONG-SPEC2K6-15	150000001	16836233	2520156
LONG-SPEC2K6-16	149999870	22064822	9019488
LONG-SPEC2K6-17	149999964	14796021	4428201
LONG-SPEC2K6-18	150000001	19691402	381488
LONG-SPEC2K6-19	150000026	14435009	432619

CHAPTER 4

RESULTS

My goal is finding a branch predictor that outperforms TAGE predictor. For this, first I tuned the TAGE predictor for CBP-4 traces. This is done to find peak performance TAGE can achieve. This way any further improvement on performance would be because of bagging. Secondly I applied bagging only on TAGE predictor. Thirdly, I used a different types of branch predictor, OH-SNAP.

For reference purposes Table 2 shows the result of simple gshare, TAGE and OH-SNAP results. All shown predictors in Table 2 is 64 KB in size. CBP-4 evaluates success based on arithmetic mean of all traces (AMEAN).

While tuning TAGE predictor my focus was on multiple parameters. Such as size of predictor, number of tables, counter width on tables, and history length. Table 3 show the effect of increasing size of TAGE predictor. As seen from results increasing the size have a significant effect on performance, but increasing the size more than 1 MB has minimal effect, therefore as a base configuration size of 1 MB is used.

Table 2: Base Simulation Results for Used Branch Predictors

	GSHARE	TAGE	OH-SNAP
SHORT-FP-1	3,307	1,088	0,987
SHORT-FP-2	1,056	0,429	0,853
SHORT-FP-3	0,444	0,014	0,042
SHORT-FP-4	0,259	0,015	0,093
SHORT-FP-5	0,788	0,007	0,015
SHORT-INT-1	6,27	0,128	0,244
SHORT-INT-2	7,683	3,686	4,654
SHORT-INT-3	10,81	6,035	5,674
SHORT-INT-4	1,931	0,459	0,645
SHORT-INT-5	0,417	0,062	0,285
SHORT-MM-1	9,48	6,649	6,473
SHORT-MM-2	10,614	8,399	8,454
SHORT-MM-3	3,53	0,06	0,069
SHORT-MM-4	1,794	0,897	1,38
SHORT-MM-5	4,993	2,395	3,311
SHORT-SERV-1	2,929	0,65	0,822
SHORT-SERV-2	2,859	0,631	0,803
SHORT-SERV-3	5,38	1,953	2,712
SHORT-SERV-4	4,949	1,445	1,941
SHORT-SERV-5	4,706	1,323	1,627
LONG-SPEC2K6-00	3,664	1,102	1,895
LONG-SPEC2K6-01	8,612	6,596	6,718
LONG-SPEC2K6-02	4,661	0,275	1,093
LONG-SPEC2K6-03	5,429	0,141	0,948
LONG-SPEC2K6-04	10,772	7,66	8,907
LONG-SPEC2K6-05	5,717	4,543	4,424
LONG-SPEC2K6-06	3,281	0,614	0,68
LONG-SPEC2K6-07	10,546	3,969	8,349
LONG-SPEC2K6-08	1,76	0,59	0,765
LONG-SPEC2K6-09	5,456	2,929	4,839
LONG-SPEC2K6-10	3,029	0,487	0,727
LONG-SPEC2K6-11	3,748	0,411	0,565
LONG-SPEC2K6-12	12,727	10,848	10,593
LONG-SPEC2K6-13	8,137	4,286	5,392
LONG-SPEC2K6-14	3,925	0,001	0,002
LONG-SPEC2K6-15	2,16	0,206	0,436
LONG-SPEC2K6-16	4,177	2,915	2,863
LONG-SPEC2K6-17	4,609	1,836	2,972
LONG-SPEC2K6-18	1,525	0,003	0,056
LONG-SPEC2K6-19	2,601	0,865	1,225
AMEAN	4,768	2,16505	2,613

Table 3: Effect of Increasing Size of TAGE Predictor

SIZE	64KB	256KB	1MB	4MB	32MB
SHORT-FP-1	1,088	1,087	1,083	1,08	1,084
SHORT-FP-2	0,429	0,43	0,429	0,43	0,43
SHORT-FP-3	0,014	0,014	0,014	0,014	0,014
SHORT-FP-4	0,015	0,015	0,015	0,015	0,015
SHORT-FP-5	0,007	0,007	0,007	0,007	0,007
SHORT-INT-1	0,128	0,129	0,129	0,128	0,128
SHORT-INT-2	3,686	3,66	3,653	3,662	3,655
SHORT-INT-3	6,035	5,904	5,869	5,846	5,835
SHORT-INT-4	0,459	0,456	0,455	0,456	0,456
SHORT-INT-5	0,062	0,059	0,059	0,061	0,059
SHORT-MM-1	6,649	6,64	6,636	6,631	6,63
SHORT-MM-2	8,399	8,356	8,349	8,368	8,331
SHORT-MM-3	0,06	0,06	0,06	0,06	0,06
SHORT-MM-4	0,897	0,875	0,871	0,863	0,865
SHORT-MM-5	2,395	2,346	2,34	2,335	2,334
SHORT-SERV-1	0,65	0,646	0,645	0,645	0,645
SHORT-SERV-2	0,631	0,626	0,628	0,628	0,628
SHORT-SERV-3	1,953	1,916	1,907	1,904	1,899
SHORT-SERV-4	1,445	1,433	1,43	1,434	1,436
SHORT-SERV-5	1,323	1,311	1,308	1,308	1,309
LONG-SPEC2K6-00	1,102	1,074	1,068	1,069	1,065
LONG-SPEC2K6-01	6,596	6,548	6,516	6,519	6,513
LONG-SPEC2K6-02	0,275	0,274	0,274	0,274	0,274
LONG-SPEC2K6-03	0,141	0,135	0,136	0,134	0,133
LONG-SPEC2K6-04	7,66	6,438	6,415	6,389	6,387
LONG-SPEC2K6-05	4,543	4,484	4,472	4,458	4,458
LONG-SPEC2K6-06	0,614	0,612	0,617	0,613	0,612
LONG-SPEC2K6-07	3,969	3,763	3,717	3,708	3,709
LONG-SPEC2K6-08	0,59	0,604	0,587	0,589	0,59
LONG-SPEC2K6-09	2,929	2,729	2,696	2,687	2,685
LONG-SPEC2K6-10	0,487	0,48	0,48	0,479	0,48
LONG-SPEC2K6-11	0,411	0,41	0,415	0,452	0,393
LONG-SPEC2K6-12	10,848	10,741	10,734	10,773	10,724
LONG-SPEC2K6-13	4,286	4,157	4,12	4,11	4,112
LONG-SPEC2K6-14	0,001	0,001	0,001	0,001	0,001
LONG-SPEC2K6-15	0,206	0,204	0,204	0,204	0,204
LONG-SPEC2K6-16	2,915	2,889	2,866	2,872	2,873
LONG-SPEC2K6-17	1,836	1,73	1,701	1,692	1,692
LONG-SPEC2K6-18	0,003	0,003	0,003	0,003	0,003
LONG-SPEC2K6-19	0,865	0,846	0,843	0,841	0,842
AMEAN	2,16505	2,1023	2,0938	2,09355	2,08925

Second parameter that I have tuned is optimal number of tables to use. But number of tables is closely correlated with history length. Therefore I made a parameter sweep for number of tables and history length parameters. In this simulation total size of predictor is kept the same. Total size is divided between tables. Therefore as I increase number of tables, I decreased size of a table to keep total size the same. As seen in Table 4 increasing number of tables increases the performance only if we increase history length with it. This is understandable since if I use small history length with a large number of tables, indexing for tables will be closer to each other and will not differ much. As a result, short history length won't be able capture long history patterns. On the other hand with long history length and small number of tables pattern length will grow too fast and there won't be enough space for short patterns. Therefore sweet spots for each configuration is show in bold. Last parameter is counter width used in tables. I simulated different counter width with different history lengths and 3-bit counter outperformed every time.

Based on gathered knowledge best TAGE parameters are 100000 max history length, 7 min history length, 38 tables, and 3-bit counters. I used 32 TAGE predictor with these configurations together with random update as described in section 3. Random value for updating can be 0, 1, or 2. And their probability is 20, 60, and 20 percent respectively. To compare this with TAGE, I also increased size of TAGE by 32 times. Simulations resulted at 1.95 misp/KI, and 2.003 misp/KI respectively. TAGE bagging is better than just increasing the size of TAGE as they both have the same size.

Table 4: Correlation Between Number of Tables and Max History Length in TAGE Predictor

Number of Tables \ MAX HISTORY LENGTH	200	400	600	800	1000	1200	1400	1600	1800	2000	2500	3000	4000	5000	10000	30000	100000
8	2,153	2,11	2,106	2,102	2,104	2,107	2,113	2,12	2,125	2,122	2,129	2,122	2,124	2,124	2,151	2,173	2,208
9	2,144	2,102	2,096	2,088	2,091	2,095	2,096	2,099	2,097	2,097	2,1	2,107	2,12	2,121	2,12	2,153	2,172
13	2,127	2,075	2,062	2,054	2,056	2,057	2,054	2,055	2,053	2,057	2,061	2,062	2,063	2,069	2,074	2,093	2,11
14	**2,124**	2,074	2,058	2,052	2,049	2,049	2,051	2,051	2,055	2,055	2,053	2,052	2,058	2,061	2,073	2,074	2,095
15	2,125	2,072	2,058	2,049	2,046	2,045	2,045	2,046	2,046	2,047	2,053	2,054	2,05	2,051	2,062	2,074	2,089
16	2,125	2,07	2,054	2,046	2,046	2,045	2,042	2,043	2,042	2,046	2,045	2,045	2,052	2,054	2,056	2,068	2,078
17	**2,124**	2,069	2,054	2,046	2,043	2,042	2,044	2,044	2,043	2,04	2,042	2,045	2,045	2,046	2,051	2,063	2,077
18	2,125	2,07	2,054	2,043	2,041	2,039	2,04	2,038	2,04	2,041	2,041	2,04	2,041	2,045	2,049	2,056	2,068
19	2,125	2,069	2,051	2,044	2,038	2,037	2,038	2,038	2,037	2,037	2,038	2,04	2,041	2,04	2,044	2,051	2,064
20	2,125	**2,068**	**2,05**	2,041	2,039	2,038	2,035	2,035	2,035	2,036	2,036	2,036	2,037	2,04	2,042	2,052	2,059
21	2,126	**2,068**	2,051	**2,04**	2,038	2,037	2,036	2,036	2,035	2,034	2,035	2,036	2,036	2,035	2,038	2,044	2,051
22	2,127	2,069	**2,05**	**2,04**	**2,037**	**2,035**	2,035	2,034	2,033	2,034	2,034	2,033	2,035	2,034	2,039	2,042	2,054
23	2,128	2,07	2,052	2,041	**2,037**	2,036	**2,034**	**2,033**	2,034	**2,032**	2,033	2,034	2,034	2,033	2,034	2,04	2,049
24	2,129	2,069	2,053	2,041	**2,037**	2,036	2,035	2,035	2,033	**2,032**	2,033	2,032	2,033	2,032	2,034	2,041	2,044
25	2,131	2,071	2,053	2,042	2,038	2,036	2,034	2,034	**2,032**	2,033	**2,032**	2,033	2,032	2,032	2,033	2,039	2,045
26	2,133	2,073	2,053	2,042	2,038	2,037	2,035	2,034	**2,032**	2,033	2,033	2,032	2,032	2,03	2,032	2,034	2,041
27	2,134	2,073	2,055	2,043	2,039	2,037	2,036	2,034	2,033	2,033	**2,032**	**2,031**	2,032	2,031	2,033	2,034	2,041
28	2,136	2,075	2,057	2,045	2,039	2,039	2,036	2,035	2,035	2,034	2,033	2,032	**2,031**	2,031	**2,031**	2,034	2,039
29	2,138	2,076	2,057	2,045	2,041	2,038	2,037	2,035	2,035	2,034	2,034	2,033	**2,031**	**2,03**	**2,031**	2,032	2,036
30	2,14	2,078	2,058	2,047	2,042	2,04	2,037	2,036	2,036	2,034	2,034	2,034	2,033	2,031	**2,031**	2,032	2,035
32														2,033	**2,031**	**2,03**	2,032
34														2,032	**2,031**	**2,03**	2,032
36														2,034	2,032	**2,03**	2,03
38														2,037	2,034	2,032	**2,029**
40														2,039	2,036	2,033	2,03
42														2,043	2,038	2,034	2,032
44														2,046	2,041	2,036	2,032

I also experimented with increased variety in TAGE predictors. I changed configuration of each predictor a little to make them different from other. I have used a fixed total size for each sub-predictor. That is, the number of table entries for a 38 component predictor is half the number of entries for a 20-component predictor for most of the tables. Counter width is fixed as 3 bits. The minimum history size varies between 5 and 13. The maximum history varies between 1000 and 100,000. Finally, the number of tagged table components in each sub-predictor varies between 20 and 38. Table 5 show detailed information about these configurations. Performance increased further to 19.1 misp/KI.

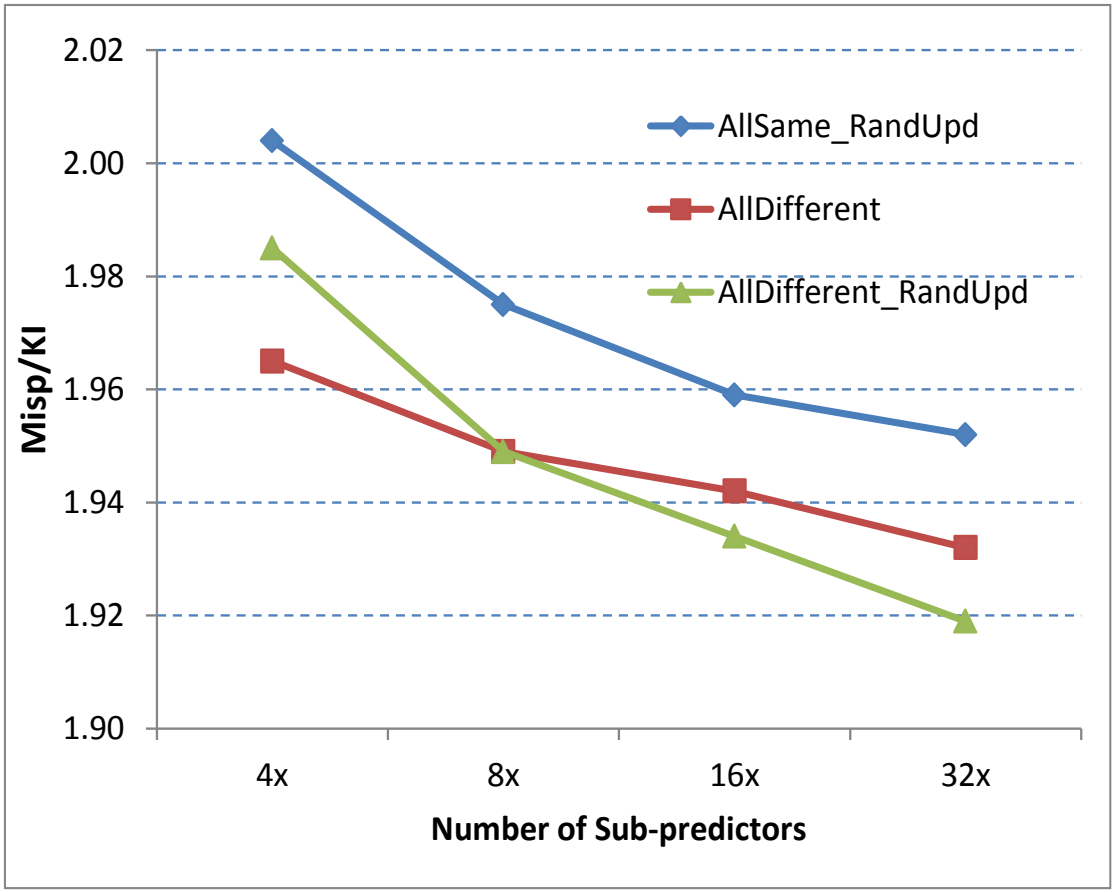

Figure 15: Comparison of Different Bagging Configurations

Figure 15 shows the overall effect of bagging. As I mentioned above, one could use the same configuration for all the sub-predictors. This configuration called AllSame. In this case, the only variability in sub-predictor predictions comes from the random updates. In this configuration, the sub-predictor parameters that we have used are: counter width = 3, number of tagged tables = 38, the minimum and maximum history lengths = 7 and 100,000, respectively. AllDifferent refers to variety between predictors. For this configuration both random update and regular update is simulated. From this figure two outcomes can be made. First AllDifferent is always better than AllSame. Secondly, to use random update there need to be some sufficient number of predictors to justify usage. For this case using more than 8 predictors is breaking point. Using random update for 8 or more predictor gives better result.

Lastly I used OH-SNAP to see effect of bagging on different type of predictor. I didn't tuned OH-SNAP and every predictor used is 64 KB. Their individual results can be seen in Table 6. Base configuration achieved 2.613 misp/KI. Increasing size by 2 and 4 times made minimal result and achieved 2.611misp/KI, and 2.608 misp/KI respectively. Using 2 OH-SNAP with bagging resulted in 2,616 misp/KI which is worse than increasing size by two times. Reason for this is we didn't use sufficient amount of predictor. At 4 predictor, bagging out performs the just size increase with 2,602 misp/KI.

Table 5: Sub-predictor Configuration

SUB-PREDICTOR	NUMBER OF TABLES	MIN HISTORY	MAX HISTORY	USE PC?
1	24	9	2000	Yes
2	32	7	30000	Yes
3	30	9	10000	Yes
4	29	6	5000	Yes
5	28	8	4000	Yes
6	27	10	3000	Yes
7	25	6	2500	Yes
8	38	5	100000	Yes
9	23	4	2000	Yes
10	23	5	1800	Yes
11	22	3	1600	Yes
12	22	8	1500	Yes
13	21	9	1400	Yes
14	21	10	1300	Yes
15	20	6	1200	Yes
16	20	7	1000	Yes
17	38	12	100000	No
18	32	10	30000	No
19	30	9	10000	No
20	29	11	5000	No
21	28	10	4000	No
22	27	13	3000	No
23	25	11	2500	No
24	24	12	2000	No
25	20	9	100000	No
26	20	10	85000	No
27	20	11	70000	No
28	20	13	55000	No
29	20	12	40000	No
30	20	8	25000	No
31	20	10	10000	No
32	20	7	8000	No

Table 6: OH-SNAP Bagging Results

	BASE	2X BASE	4X BASE	BAGGING X2	BAGGING X4
SHORT-FP-1	0,987	0,986	0,983	0,986	0,983
SHORT-FP-2	0,853	0,849	0,848	0,852	0,848
SHORT-FP-3	0,042	0,041	0,041	0,04	0,04
SHORT-FP-4	0,093	0,093	0,093	0,093	0,093
SHORT-FP-5	0,015	0,015	0,015	0,015	0,015
SHORT-INT-1	0,244	0,246	0,246	0,247	0,245
SHORT-INT-2	4,654	4,631	4,622	4,664	4,618
SHORT-INT-3	5,674	5,67	5,659	5,667	5,629
SHORT-INT-4	0,645	0,648	0,646	0,648	0,633
SHORT-INT-5	0,285	0,283	0,282	0,286	0,285
SHORT-MM-1	6,473	6,465	6,473	6,479	6,467
SHORT-MM-2	8,454	8,45	8,44	8,449	8,44
SHORT-MM-3	0,069	0,068	0,068	0,069	0,069
SHORT-MM-4	1,38	1,381	1,384	1,384	1,381
SHORT-MM-5	3,311	3,315	3,306	3,318	3,294
SHORT-SERV-1	0,822	0,822	0,82	0,829	0,824
SHORT-SERV-2	0,803	0,803	0,799	0,81	0,801
SHORT-SERV-3	2,712	2,71	2,704	2,729	2,708
SHORT-SERV-4	1,941	1,935	1,933	1,946	1,934
SHORT-SERV-5	1,627	1,628	1,623	1,636	1,625
LONG-SPEC2K6-00	1,895	1,895	1,891	1,898	1,888
LONG-SPEC2K6-01	6,718	6,717	6,716	6,723	6,716
LONG-SPEC2K6-02	1,093	1,094	1,093	1,099	1,063
LONG-SPEC2K6-03	0,948	0,948	0,946	0,945	0,94
LONG-SPEC2K6-04	8,907	8,9	8,892	8,905	8,891
LONG-SPEC2K6-05	4,424	4,42	4,423	4,424	4,416
LONG-SPEC2K6-06	0,68	0,678	0,679	0,68	0,678
LONG-SPEC2K6-07	8,349	8,329	8,313	8,365	8,245
LONG-SPEC2K6-08	0,765	0,763	0,76	0,766	0,765
LONG-SPEC2K6-09	4,839	4,837	4,835	4,844	4,832
LONG-SPEC2K6-10	0,727	0,728	0,723	0,732	0,718
LONG-SPEC2K6-11	0,565	0,566	0,565	0,567	0,565
LONG-SPEC2K6-12	10,593	10,593	10,591	10,603	10,586
LONG-SPEC2K6-13	5,392	5,374	5,368	5,399	5,32
LONG-SPEC2K6-14	0,002	0,002	0,002	0,002	0,002
LONG-SPEC2K6-15	0,436	0,435	0,435	0,436	0,429
LONG-SPEC2K6-16	2,863	2,86	2,855	2,869	2,859
LONG-SPEC2K6-17	2,972	2,967	2,965	2,972	2,954
LONG-SPEC2K6-18	0,056	0,055	0,055	0,052	0,055
LONG-SPEC2K6-19	1,225	1,223	1,223	1,225	1,224
AMEAN	2,613	2,611	2,608	2,616	2,602

CHAPTER 5

CONCLUSION

High-performance microarchitectures use, among other structures, deep pipelines to help speed up execution. It is very important to have a good branch predictor to keep all stages of pipeline executing instructions from correct path.

Bootstrap aggregating (bagging) is a statistical method to improve the accuracy of predictors by reducing variance and over fitting. It is applicable to any unstable learning algorithm. In this work, I applied bagging to branch prediction. Branch predictor forms an ensemble of slightly different predictors each of which is updated with slightly different data.

My results show that using bagging can increase performance further than what branch predictor capable of. TAGE predictor scales well with the predictor size and OH-SNAP has a minimal dependency to its size. But in both cases bagging was able to outperform both.

Bagging shows promise as a future research direction. Although online bagging method used in this work provides a way to apply bagging to branch prediction, it assumes independent samples, which is not the case for branch history. Different online bagging methods may prove better and are subject to future research. Finally, my analysis was done by mostly using TAGE as the base predictor. I looked into OH-SNAP briefly. It is possible to use more variety of predictors that use different methods for prediction.

Another thing I want to mention is, with this idea I entered CBP-4 competition and took fourth place in unlimited size category [15].

BIBLIOGRAPHY

[1] Seznec, André, and Pierre Michaud. "A case for (partially) TAgged GEometric history length branch prediction." Journal of Instruction Level Parallelism 8 (2006): 1-23.

[2] J.K.L. Lee and A.J. Smith. "Branch prediction strategies and branch target buffer design." Computer, 17(1), January 1984.

[3] Nair, Ravi. "Dynamic path-based branch correlation." Proceedings of the 28th annual international symposium on Microarchitecture. IEEE Computer Society Press, 1995.

[4] Yeh, Tse-Yu, and Yale N. Patt. "Two-level adaptive training branch prediction." Proceedings of the 24th annual international symposium on Microarchitecture. ACM, 1991.

[5] McFarling, Scott. Combining branch predictors. Vol. 49. Technical Report TN-36, Digital Western Research Laboratory, 1993

[6] Young, Cliff, and Michael D. Smith. "Improving the accuracy of static branch prediction using branch correlation." ACM Sigplan Notices. Vol. 29. No. 11. ACM, 1994.

[7] Young, Cliff, Nicolas Gloy, and Michael D. Smith. A comparative analysis of schemes for correlated branch prediction. Vol. 23. No. 2. ACM, 1995.

[8] Seznec, Andr. "The o-gehl branch predictor." The 1st JILP Championship Branch Prediction Competition (CBP-1) (2004).

[9] Jiménez, Daniel A., and Calvin Lin. "Dynamic branch prediction with perceptrons." High-Performance Computer Architecture, 2001. HPCA. The Seventh International Symposium on. IEEE, 2001.

[10] Seznec, André. "Redundant history skewed perceptron predictors: Pushing limits on global history branch predictors." Publication interne- IRISA (2003).

[11] Breiman, Leo. "Bias, variance, and arcing classifiers." Technical Report 460, Department of Statistics, University of California, Berkeley (1996).

[12] Bauer, Eric, and Ron Kohavi. "An empirical comparison of voting classification algorithms: Bagging, boosting, and variants." Machine learning 36.1-2 (1999): 105-139.

[13] Oza, Nikunj C., and Stuart Russell. "Online bagging and boosting." Systems, man and cybernetics, 2005 IEEE international conference on. Vol. 3. IEEE, 2005.

[14] Härdle, Wolfgang, Joel Horowitz, and Jens-Peter Kreiss. "Bootstrap methods for time series." International Statistical Review 71.2 (2003): 435-459.

[15] http://www.jilp.org/cbp2014/

[16] http://www.spec.org/cpu2006/