Computational and Storage Based Power and Performance Optimizations for Highly Accurate Branch Predictors Relying on Neural Networks

by

Kaveh Aasaraai
B.Sc. Sharif University of Technology 2005, Tehran, Iran

A Thesis Submitted in Partial Fullfillment of the
Requirements for the Degree of

MASTER OF APPLIED SCIENCE

in the Department of Electrical and Computer Engineering

Computational and Storage Based Power and Performance Optimizations for Highly Accurate Branch Predictors Relying on Neural Networks

by

Kaveh Aasaraai
B.Sc. Sharif University of Technology 2005, Tehran, Iran

Supervisory Committee

Dr. Amirali Baniasadi (Department of Electrical and Computer Engineering)
Supervisor

Dr. Daler Rakhmatov (Department of Electrical and Computer Engineering)
Departmental Member

Dr. Mihai Sima (Department of Electrical and Computer Engineering)
Departmental Member

Dr. Kui Wu (Department of Computer Science)
External Examiner

Supervisory Committee

Dr. Amirali Baniasadi (Department of Electrical and Computer Engineering)
Supervisor

Dr. Daler Rakhmatov (Department of Electrical and Computer Engineering)
Departmental Member

Dr. Mihai Sima (Department of Electrical and Computer Engineering)
Departmental Member

Dr. Kui Wu (Department of Computer Science)
External Examiner

Abstract

In recent years, highly accurate branch predictors have been proposed primarily for high performance processors. Unfortunately such predictors are extremely energy consuming and in some cases not practical as they come with excessive prediction latency. Perceptron and O-GEHL are two examples of such predictors. To achieve high accuracy, these predictors rely on large tables and extensive computations and require high energy and long prediction delay. In this thesis we propose power optimization techniques that aim at reducing both computational complexity and storage size for these predictors. We show that by eliminating unnecessary data from computations, we can reduce both predictor's energy consumption and prediction latency. Moreover, we apply information theory findings to remove noneffective storage used by O-GEHL, without any significant accuracy penalty. We reduce the dynamic and static power dissipated in the computational parts of the predictors. Meantime we improve performance as we make faster prediction possible.

Table of Contents

List of Tables

List of Figures

Acknowledgements

I would like to express my deepest gratitude, first and foremost, to my supervisor, Dr. Amirali Baniasadi for supervision, advice, and guidance from the very early stage of this research as well as giving me extraordinary experiences through out the work. Above all and the most needed, he is a friend to me that I would never lose contact with.

I am pleased to thank my supervisory committee members for directing me in my research and giving me valuable advices. I am also glad that I made a valuable and knowledgeable friend, Scott Miller, during my research at University of Victoria. He has always been of many helps to my family and my research. I will always be thankful to him.

Many people helped me to pursue my studies, including the technical staff at Electrical and Computer Engineering department. I'd like to thank Steve Campbell, Erik Laxdal and Duncan Hoggs for maintaining our reliable network and research facilities.

I would also like to thank my parents, for being supportive of my studies away from home, and constantly showing me the right path to take in my studies. Also my wife who made it easy to pass the difficulties of this work. Thank you all who are always in my heart.

Chapter 1

Introduction

A pipelined processor requires a continuous stream of instructions to get close to its potential performance. However, program flow may be changed by branching instructions, whose outputs are not determined until their execution is complete. As a result, such instructions disrupt instruction flow, consequently, reducing instruction throughput. Therefore, in order to maximize instruction throughput, the processor should speculate on both branch target address and direction.

High performance processors exploit branch predictors to speculate on branch instructions. To determine the next instruction, the processor predicts both branch's target address and direction. However, mispredicting branch output results in executing instructions from the wrong path, consequently degrading performance. In order to avoid wrong path execution, accurate branch prediction techniques are necessary.

In recent years, designers have proposed highly accurate branch predictors based on neural networks [9, 18]. Such predictors, while favoring uniformity, perform excessive computations in their aggressive approach to predict branch direction. Quite often, these predictors come with excessive latency and power dissipation resulting in major implementation difficulties [10].

We propose power optimization techniques for perceptron and Optimized GEometric History Length (O-GEHL), two of the most accurate and well documented branch predictors. We show that these predictors perform noneffective operations that can be eliminated without noticeable impact on their prediction accuracy. We show that both power dissipation and prediction latency can be reduced by identifying and eliminating such noneffective operations.

1.1 Branch Prediction Essentials

Studies have shown that branch instructions' behavior is highly predictable [14]. It has been shown that branches tend to correlate to their past behavior. Also branches are found to be correlated to each other. Therefore, by determining a branch output, subsequent occurrences of the branch and correlated branches will be predictable.

Branch predictors use branch instructions past behavior as a basis for prediction. While branch history is easy to obtain, exploiting history patterns and determining correlations among branch instructions is not trivial. Early branch predictors used simple saturating counters to store the past behavior of branch instructions. However, such techniques can not exploit long history lengths as they are not able to distinguish between different patterns of long history. In modern and accurate table based predictors, which exploit neural networks, long history lengths are used resulting in exponentially growing storage requirements [9, 18].

In addition to branch direction, processor requires branch target address to fetch the next instruction. Processor uses simple methods based on branch's previous executions to predict branch target address. In the following sections, we discuss methods used for both branch target address and direction speculation.

1.2 Branch Target Speculation

Branch instructions use two addressing modes to specify the target address. In the PC relative mode, the branch instruction opcode specifies the target address as an offset to the branch address. Therefore, the fetch engine resolves the branch target address by adding the offset to the PC. However, in register indirect addressing mode, the target address can not be resolved until the content of the register is retrieved. This postpones the target address resolution to the decode stage where registers are renamed [19].

In order to avoid processor stall due to target address resolution, the predictor stores a resolved branch target address to be used for subsequent executions of the branch. For this matter, the predictor uses a branch target buffer (BTB). BTB is a cache-like memory, accessed at the fetch stage using the branch instruction address. BTB entries are distinguished by a tag field, representing the branch instruction address hashed to the entry.

The first time a branch instruction is fetched, an entry in BTB is allocated. Later when the actual target address is calculated, the predictor stores the target address in BTB. In subsequent executions of the branch instruction, the target address is retrieved from BTB and is used as the next instruction address.

Regardless of using a target address from BTB or not, the predictor computes the branch target address. If an address was used from BTB, the predictor compares the actual target address with the predicted one. In the event of a mismatch, the processor flushes the instructions in the wrong path and redirects the fetch.

1.3 Branch Direction Speculation

The simplest way to speculate on branch direction is to assume all branches are not taken. In this way, when the fetch engine encounters a branch instruction, regardless of the branch target address, it continues to fetch down the sequential execution path. This form of prediction is easy to implement, however, is not effective as a high number of branch instructions are taken.

A more dynamic form of prediction makes prediction based on branch target address. In the event of hitting an entry in the BTB, the target address is used to predict the branch outcome. If the target address is pointing to a position above the current branch instruction, it is predicted as taken. This is mostly the case for branches ending loops which are pointing to the beginning of the loop. This branch

prediction technique is used in the original IBM RS/6000 design [4].

The most common branch direction speculation technique employed by contemporary superscalar processors is history-based. The predictor decides a direction for the branch based on its behavior in the past. Bimodal predictor [21], for example, uses a simple way to exploit the history patterns of branch instructions. Bimodal exploits an n-bit counter for each branch instruction to store past behavior. When a branch is found to be taken, the corresponding counter is incremented. When the branch outcome is not taken, the counter is decremented. The predictor uses the most significant bit of the counter to predict the branch outcome where '1' and '0' predict the branch as taken and not taken respectively. Figure 1.1 shows a 2-bit bimodal, with the corresponding states of the counter.

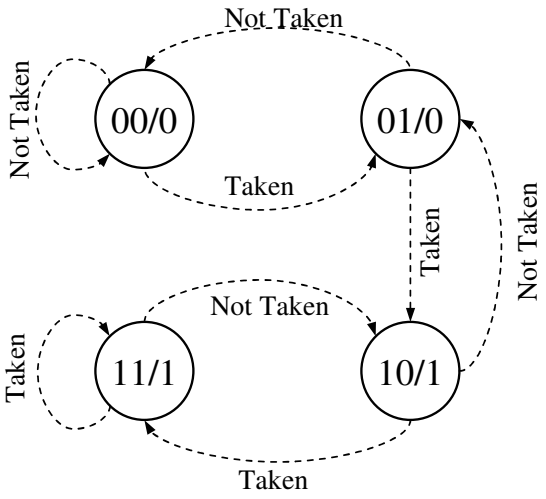


Figure 1.1: A 2-bit bimodal predictor. Values inside each state represent the value of the counter in that state, followed by the prediction direction. Transitions between states occur after actual branch outcome resolution.

A more advanced dynamic prediction technique was proposed by Yeh and Patt [23]. Their predictor had two parts. The first part of the predictor stores global branch

outcome history as an array of 1-bits, into a register. The second part is a table of 2-bit counters which store past behavior of branches, similar to bimodal predictor. The table is indexed by concatenation of a hash of the branch address with the global history. The predictor uses the retrieved counter to predict the branch outcome.

1.4 Branch Prediction Procedures

A branch predictor has three main procedures, Lookup, Update and gathering outcome history. In the following sections, we will discuss these steps and their role in branch prediction.

1.4.1 Lookup

At the fetch stage, with the occurrence of fetching a branch instruction, a lookup procedure is followed in order to provide the fetch engine with both branch target address and direction. The predictor looks up BTB for branch target address. The fetch engine uses this target address as the address of the next instruction.

In addition to target address prediction, a direction prediction occurs for the branch instruction. The predictor performs the necessary steps to predict the branch direction. For example, in a bimodal predictor, an index is created using a hash of the branch address. Then the index is used to retrieve the n-bit counter from the predictor table. Finally, the predictor uses the most significant bit of the counter as the prediction for branch direction.

1.4.2 Update

A branch instruction is resolved when it is completely executed. At this time, both its target address and direction are determined. With this information, the processor updates the branch predictor to predict subsequent occurrences of the branch instruction correctly. The predictor stores the branch target address in the BTB.

Also, the predictor updates the counter corresponding to the branch, to represent its most recent behavior.

Due to branch speculation, a branch instruction itself may belong to a wrong path. Therefore, updating the predictor state upon execution completion of a branch may result in gathering wrong information. However, if an instruction retires, it belongs to the correct execution path, as there is no unresolved branch before current instruction [19]. Retired instructions are allowed to change processor state and therefore are used to update predictor state.

1.4.3 History Collection

Branch outcome history is used as the key element in branch predictors. The predictor obtains both branches' behavior and correlations among them using the branch outcome history. Branch history is stored as one bit, '0' and '1' representing not taken and taken outcomes respectively. The string of bits is stored in a shift register, where the new outcome is shifted to one end, and the oldest one is shifted out of the other end.

While it is reasonable to collect the branch outcome in conjunction with update procedure, studies have shown that collecting history speculatively can enhance prediction accuracy [5]. Since the number of unresolved branch instructions present in the processor pipeline varies, omitting unresolved branch instructions from history register substantially degrades predictor's accuracy.

1.5 The Importance of Accurate Branch Prediction

A speculative pipelined processor requires branch direction prediction to achieve its best performance. A mispredicted branch results in flushing subsequent fetched instructions in the pipeline, imposing a misprediction penalty. This penalty increases as the number of pipeline stages grows, since the time to resolve branch instruc-

tion's outcome increases. Therefore, in a deeply pipelined processor, upon a branch misprediction, the processor could spend long time periods executing mispredicted instructions. As a result, it is crucial to predict branch outcome correctly as the processor performance would degrade significantly due to branch misprediction penalty.

1.6 High Performance Branch Predictors Deficiency

Highly accurate branch predictors can significantly improve processor performance. However, such predictors come with implementation difficulties which make them impractical. The two most important deficiencies of such predictors are known to be their prediction latency and energy consumption [10, 8]. We discuss each one of these aspects in the following two sections.

1.6.1 Timing Overhead

As processor pipeline execution bandwidth continues to grow, more and more instructions are being fetched at every cycle. This results in an increase in the number of fetched and predicted branch instructions. In order to avoid instruction flow disruption, the fetch engine requires the next instruction address as soon as possible. Therefore, branch predictions must be performed with minimum delay.

If branch prediction takes multiple cycles to complete, the fetch engine would not be able to fetch subsequent instructions immediately. As a result, the fetching process is stalled until the branch instruction is predicted. This results in significant performance degradation.

Jimenez et. al. in [8] have shown that a processor using an oracle branch predictor, i.e. 100% accurate, with two clock cycles latency, achieves lower performance compared to a processor using a moderate branch predictor with one clock cycle latency. According to this observation, high performance predictors may not be able to deliver overall performance improvement as a result of the timing overhead.

In the following chapters we will show that two of most accurate branch predictors use large tables to store behavioral correlation information. These tables are accessed at the prediction time. Since these tables are large, their access time is also long which in turn increases prediction latency.

In addition to their storage size, the predictors studied in the thesis perform a summation on the data obtained from the tables [9, 18]. This process requires an adder tree which its size is relevant to the number of inputs and their size. Unfortunately, these predictors quite often use a relatively large number of wide inputs, which results in high prediction latency.

1.6.2 Power Overhead

A pipelined processor performs a branch target address and direction prediction for each branch instruction. Due to wide super pipelines of high performance processors, the processor could fetch more than one branch instruction at a time requiring accessing the predictor more than once. This in turn can increase power dissipation.

High performance branch predictors use large structures and complex arithmetic units to achieve high accuracy. Having a large number of rows, their storage tables could require a large, energy consuming decoder. This increases both static and dynamic power dissipation in the predictor. Moreover, each row of the table stores several elements, which the predictor loads for performing prediction. The increased number of bitlines and wordlines for each row increases the dynamic and static power dissipation of the predictor.

Additionally, as explained earlier, some predictors perform an accumulation upon the branch correlation data [9, 18]. The structure of the adder tree used for accumulation process dissipates high amount of static and dynamic power. The power dissipation of such adders depends on the number of inputs and the width of each input, which is relatively large in high performance predictors [9, 18].

1.7 Contributions

In this thesis, we have proposed new techniques to reduce power dissipation of perceptron and O-GEHL, two of the most accurate branch predictors [9, 18]. We identify and eliminate noneffective operations such predictors perform. As a result, we reduce their power dissipation, meanwhile due to lower amount of operations, we reduce prediction latency as well. We also exploit information theory findings to reduce and eliminate the noneffective storage used in the O-GEHL predictor. We show that by eliminating unnecessary data from computations, we can reduce both predictor's energy consumption and prediction latency. We reduce the dynamic and static power dissipated in the computational parts of the predictors. Meantime we improve performance as we make faster prediction possible.

1.8 Thesis Organization

Chapter 2 discusses previous work done in the area of power-ware branch prediction. Chapter 3 introduces perceptron branch predictor and illustrates inefficiencies of the predictor from the energy point of view, while proposes power optimizations for the predictor. Chapter 4 addresses computational and storage redundancies in the O-GEHL branch predictor and discusses power optimizations for this predictor. Finally the thesis is concluded in Chapter 5 with a summary of major contributions and possible directions for future work.

Chapter 2
Related Work

In this chapter we discuss previous work done in the area of power-ware branch prediction. Jimenez and Lin [9] have suggested identifying important history bits in the perceptron predictor as a future research project. Loh and Jimenez [13] introduced two optimization techniques for perceptron. They proposed a modulo path-history mechanism to decouple the branch outcome history length from the path length. They also suggested bias-based filtering exploiting the fact that neural predictors can easily track strongly biased branches whose frequencies are high. Therefore, the number of accesses to the predictor tables is reduced due to the fact that only bias weight is used for prediction.

Parikh et al. explored how branch predictor impacts processor power dissipation. They introduced banking to reduce the active portion of the predictor. They also introduced prediction probe detector (PPD) to identify when a cache line has no branches so that a lookup in the predictor buffer (BTB) can be avoided [15].

Baniasadi and Moshovos introduce Branch Predictor Prediction (BPP) for a two level branch predictor [1]. They stored information regarding the sub-predictors accessed by the most recent branch instructions executed and avoided accessing underlying structures. They also introduced Selective Predictor Access (SEPAS) [2] which selectively accessed a small filter to avoid unnecessary lookups or updates to the branch predictor.

Huang et al. used profiling to reduce branch predictor's power dissipation [7]. They show that large structures of the predictors are often underutilized. They disabled a hybrid predictor's components that do not improve accuracy. They also

reduced predictor power dissipation by dynamically adapting branch target buffer size to application's demand.

Hu et al. [6] show that as branch predictor's structure grows in size, the leakage energy importance increases. Furthermore, they show that branch predictor is a thermal hot spot, thus increases its leakage. They show that the same decay techniques often used in reducing cache structure's power dissipation [11], is also applicable to branch predictors.

Jimenez et al. [8] suggested using an overriding branch predictor to reduce delay. They used two predictions, where the first one is a faster, less accurate predictor compared to the second one. The first predictor is responsible for providing the processor with a fast direction prediction for the branch. The second predictor, however, predicts the branch after a certain number of cycles, and redirects the fetch on the event of not conforming to the first predictor.

Loh et al. [12] used the hysteresis bit bias to reduce table size in branch predictors using 2-bit saturating counters. They used data compression techniques and showed that hysteresis bit's entropy in 2-bit saturating counters is less than 1-bit.

Chapter 3

Power-Aware Perceptron Branch Predictor

In this chapter we introduce a power-aware alternative for the perceptron branch predictor. We identify noneffective operations performed by the predictor and suggest mechanisms to eliminate them. We modify the predictor structure to accommodate our optimizations and reduce both prediction latency and power dissipation.

3.1 Introduction

Perceptron based predictors are highly accurate compared to conventional table based branch predictors [9]. This high accuracy is the result of exploiting long history lengths and is achieved at the expense of high complexity of the predictor structure [9].

Perceptron relies on exploiting behavior correlations among branch instructions. To collect as much information as possible, perceptron stores past behavior for a large number of previously encountered branch instructions. This aggressive approach, while providing high accuracy, is inefficient (i.e., from the energy point of view) as it favors uniformity. This uniform approach assumes that behavior correlations are distributed evenly among branch instructions. Therefore, the predictor stores information for as many as possible previously seen branch instructions for every branch instruction. This in turn results in performing a large number of computations per branch instruction.

The key opportunity for reducing perceptron power dissipation lies in that not all the branch instructions are highly correlated. As we show in this chapter, not all the information stored impacts the overall prediction accuracy. Moreover, a considerable

share of the computations performed by perceptron is unnecessary as it does not make any difference in the branch prediction outcome.

In this chapter we introduce techniques to reduce power in perceptron by eliminating the unnecessary and noneffective computations performed for every branch instruction.

Power optimization techniques often reduce power at the expense of performance. Our technique, however, reduces predictor power dissipation while improving processor computation performance. This is due to the fact that as we eliminate unnecessary computations, we also reduce prediction latency making faster yet highly accurate branch prediction possible. However, our modifications come with slightly extra silicon area requirement as we increase predictor's components.

Identifying non-correlated behaviors comes with some latency overhead. Accordingly, we study two different possible timing scenarios where either all or only a subset of easy-to-identify noneffective computations is eliminated. We show that power savings are higher if only the easy-to-identify noneffective operations are removed. For this scenario power savings varies from 20% to 34% across different configurations while performance improvement reaches a maximum of 16%. It should be noted that better performance improvements are achieved when all unnecessary computations are eliminated. For this scenario the extra power overhead results in power savings between 2% and 31%. Performance improvement, however, reaches a maximum of 19%.

In Section 3.2 we remind perceptron branch predictor. In Section 3.3 we show that perceptron performs noneffective operations while in Section 3.4, we introduce a classification scheme to identify such operations. In Section 3.5, we propose a scheme to eliminate the identified noneffective operations. In Section 3.6, we propose optimization modifications to the branch predictor structure, to accommodate our proposed optimizations. In Section 3.7, we discuss the overheads imposed by our

optimizations and in Section 3.8 we evaluate our optimization scheme and provide simulation results.

3.2 Perceptron Branch Predictor

Perceptron relies on exploiting behavior correlations among branch instructions. Perceptron uses branch correlation information to predict the branch outcome. To collect as much information as possible, perceptron stores past behavior for a large number of previously encountered branch instructions. Perceptron relies on many components to perform direction prediction and to obtain correlation data.

3.2.1 Predictor Components

Figure 3.1 shows a perceptron branch predictor. Perceptron stores behavioral correlation information as signed integer numbers, known as weights, in the weight table. Weights are stored using 1's complement representation. For each branch instruction, a vector of weights is tracked and updated according to branch behavior. w_i, being the i-th weight in the weights vector, is used to collect the correlation between the current branch and the i-th previously encountered branch instruction. The first weight in the vector is the bias weight, indicating branch bias independent of branch history. This weight is essentially representing branch correlation to its past behavior.

Perceptron stores all weight vectors in a table. This table is indexed using the branch instruction address [9]. However, due to storage limitations [9], this table can not be large enough to accommodate vectors for all branch instructions. Therefore the same vector is used for multiple branch instructions. This is the result of hashing different branch instructions into a single table entry.

In addition to the weights vectors, branch outcome history is also recorded. Each outcome is stored using one bit, in which "1" and "0" represent taken and not taken

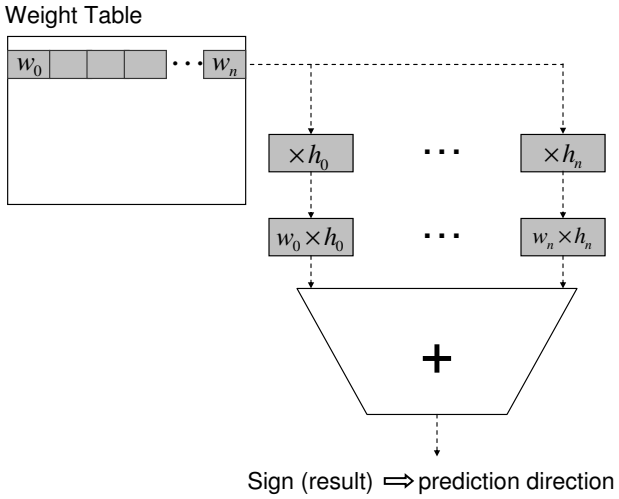


Figure 3.1: The perceptron branch predictor. A table is used to store all weights vectors. An adder tree accumulates all the elements and provides the predictor with the dot product value.

outcomes respectively. The outcome history represents the branch behavior and provides critical information. Moreover, branch instructions' local outcome histories may also be used to achieve more accurate predictions [9].

The main component of the perceptron predictor is the adder tree. The predictor uses the adder tree to compute the dot product of the two weights and history vectors at the prediction time. This is described more in the next section.

3.2.2 Lookup Procedure

For performing prediction, first the predictor hashes the predictor table using the branch address and retrieves the weight vector corresponding to the branch instruction. The predictor may also, in addition to the weights vector, retrieve local history data, if present. The local and global history vectors, concatenated to each other,

form the history vector.

The predictor uses the dot product of the weights vector and history vector. In the dot product process, the predictor replaces history elements that are stored as "0", representing a not taken outcome in the past, with a value of "-1". The dot product process requires an adder tree to compute the sum of all the partial products. Figure 3.1 shows this process in more detail. After calculating the dot product of the two vectors, the predictor uses the sign of the dot product value for prediction: a non-negative value predicts the branch as taken and a negative one predicts as not taken.

3.2.3 Update Procedure

The perceptron predictor is essentially a neural network and requires training for pairs of inputs and outputs. At the update time, the actual outcome of the branch instruction is determined by the processor. At this time, the predictor updates the neural network by updating the weights corresponding to the branch. However, excessive training may degrade predictor's accuracy [9]. Therefore, the dot product value is used to decide when sufficient training has been done. Once this value exceeds a predetermined threshold, the predictor is believed to be trained enough, and is not updated any longer. On a misprediction occurrence, however, the predictor is always updated, regardless of the dot product value. Equation 3.1 shows the threshold proposed in [9], in which θ is the threshold and h is the history length, global and local combined. This is found to be the best threshold because adding another weight to a perceptron increases its average output by some constant. Therefore, the threshold must be increased by a constant, resulting in a linear relationship between history length and threshold. It should also be noted that we use the same set of benchmarks used in [9], which enables us to exploit this threshold.

$$\theta = \lfloor 1.93h + 14 \rfloor \tag{3.1}$$

Updating the predictor requires updating the weights vector used to predict the branch instruction outcome. Each weight in the vector is either incremented or decremented. If the history corresponding to a weight conforms to the branch actual outcome, the weight is incremented, otherwise it is decremented. Equation 3.2 shows the update equation for each weight.

$$w_i = \begin{cases} w_i + 1 & h_i = outcome \\ w_i - 1 & h_i \neq outcome \end{cases} \tag{3.2}$$

It should also be noted that weights are saturating counters.

3.3 Noneffective Operations

In Section 3.2 we showed that the perceptron predictor computes the dot product of a weights vector and history vector at the prediction time. The predictor uses the dot product value to guess the branch outcome. However, the prediction outcome depends only on the sign of the dot product. Consequently, calculations which do not impact the outcome sign are noneffective and impose unnecessary power dissipation.

To provide better understanding, in Figure 3.2, we present a simple example. As shown in Figure 3.2, the dot product value is negative. Therefore the predictor predicts the branch outcome as not taken. However, by using only the weight with value of "100", the predictor is able to predict the branch outcome with no change. Figure 3.3 shows a sample code which results in a weight vector like shown in Figure 3.2.

To provide better insight, in Figure 3.4 we compare the accuracy of two predictors. The first bar shows the accuracy of a perceptron predictor which uses a weights table of size 128Kbits. The second bar, titled as bias weight, is a perceptron predictor with

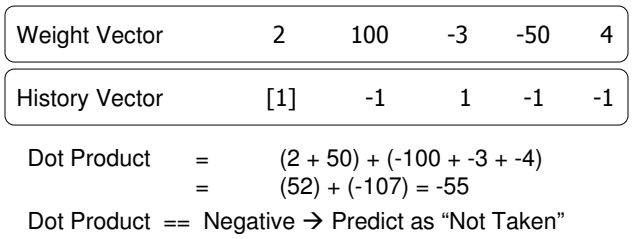


Figure 3.2: Example of weight and history vectors and the branch outcome prediction made using the dot product value. The second weight, having the value of "100", single handedly decides the outcome. Accurate prediction is possible by using only the second weight's value and the corresponding outcome history. All other calculations are noneffective.

history and weights vector lengths of one. Essentially, this predictor is only using the bias weight to predict the branch outcome. As presented, 60% of the time, by using only the bias weight, the predictor can make accurate predictions. We conclude from this figure that while the complexity associated with storing multiple weights and performing several computations per branch improves accuracy, not all predictions require exploiting a full-blown predictor. In fact identifying and eliminating such noneffective operations does not impact prediction accuracy and potentially will increase predictor's power efficiency.

3.4 Identifying Noneffective Operations

In order to eliminate the noneffective operations specified in the previous section, we need to identify such operations first. With this respect, we exploit a weight classification technique which is described as following.

Let w_i be the i-th weight of each branch's weight vector. For each w_i, there exists an h_i, which is the corresponding outcome history. Let DP represent the dot product

```
while(true)
{
        var a = input();
        var b = input();
   (1)  if (b > 0)
        {…}
   (2)  if (a < -1)
        {…}
   (3)  if (b != 0)
        {…}
   (4)  if (a >= 0)
        {…}
   (5)  if (a > 0)
        {…}
}
```

Figure 3.3: Branch number 5 is currently being predicted. Branch 5 is highly correlated to branch 4, as they depend on the same values of variable "a". This relates to the weight value of "100" in Figure 3.2. However, branch 5 is negatively correlated to branch 2, as they depend on two separate sets of "a" values. This corresponds to the weight "-50". Note that branches 1 and 3 are not correlated to this branch as they depend on a separate variable, i.e., "b", with no correlation to variable "a".

value of two vectors, weights and histories. We have:

$$DP = \sum_{i=0}^{L} h_i w_i = \sum_{i=0}^{L} e_i \tag{3.3}$$

We refer to each $h_i w_i$ as a vector element or e_i. We can then categorize the values e_is into two groups according to their sign. Let CS hold the indexes of elements with

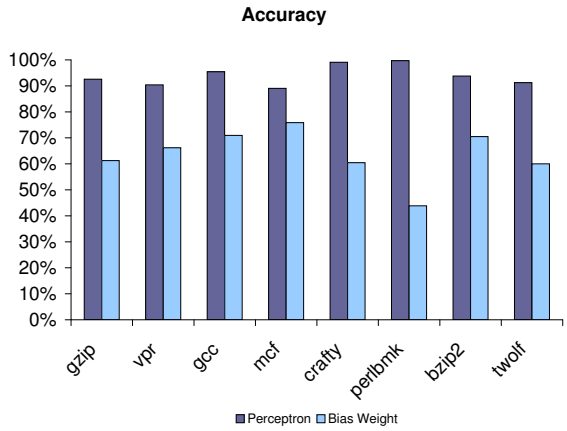


Figure 3.4: Example of weight and history vectors and the branch outcome prediction made using the dot product value. The second weight, having the value of "100", single handedly decides the outcome. Accurate prediction is possible by using only the second weight's value and the corresponding outcome history. All other calculations are noneffective.

a sign similar to DP's, and NCS hold the indexes of elements with opposite sign. We can rewrite Equation 3.3 as Equation 3.4.

$$DP = CDP + NCDP = \sum_{i \in CS} e_i + \sum_{i \in NCS} e_i \tag{3.4}$$

where CDP represents the summation of CS elements and $NCDP$ represents summation over NCS elements.

Regardless of the DP's sign, CS elements are effective and essential in deciding the branch outcome. Meantime NCS elements appear to be non-effective and noneffective as they only reduce the DP's absolute value, not changing its sign. We refer to CS elements as the effective or E class. We refer to NCS elements as the non-effective or NE class.

For clarification, assume a positive value for DP. Under such circumstances, the

predictor predicts the branch as taken. Also, CDP has a positive value as all of its elements are positive and $NCDP$ has a negative value while all its elements are negative. Similar arguments could be made for negative values of DP.

Note that not all elements in the E class are equally effective. We categorize effective elements into two subclasses, semi-effective (SE) and fully-effective (FE). Elements' subclasses are identified as follows:

$$\begin{aligned} e_i \in FE &\Leftrightarrow e_i \in E, |e_i| \geq |NCDP| \\ e_i \in SE &\Leftrightarrow e_i \in E, |e_i| < |NCDP| \end{aligned} \tag{3.5}$$

Recall that $NCDP$ is the sum of all the elements in the NE class. As defined in Equation 3.5, FE elements have values greater than the sum of all the elements in the NE class. Therefore, a single FE element negates all NE elements effectively deciding the outcome single handedly. In other words, in the presence of an FE element, all elements in other classes are unnecessary to perform the prediction. This also includes SE elements since they increase the DP's value but do not change its sign. To clarify this further, in Figure 3.5 we show a detailed example.

In Figure 3.6, we report how often each weight class appears in a 128Kbits perceptron (see Appendix-A for methodology). We observe that NE elements have a high frequency. FE elements, on the other hand, are infrequent.

We conclude from Figure 3.6 that the predictor, quite often, performs a large number of unnecessary computations due to high frequency of NE and SE elements. We use this classification and specify the calculations the predictor performs for NE and SE elements as noneffective operations.

3.5 Eliminating Noneffective Operations

In order to eliminate the noneffective operations identified by the classification scheme, we propose the following scheme. At the prediction time, the predictor,

Weight Vector	2	100	-3	-50	4
History Vector	[1]	-1	1	-1	-1

Dot Product = (2 + 50) + (-100 + -3 + -4) = (52) + (-107) = -55

Dot Product == Negative → Predict as "Not Taken"

Elements = {2, -100, -3, 50, -4}

CS = {-100, -3, -4} NCS = {2, 50}

NCDP = 2 + 50 = 52 CDP = -100 + -3 + -4 = -107

Class NE: {2, 50} Class E : {-100, -3, -4}

Class SE: {-3, -4} Class FE: {-100}

Figure 3.5: The dot product value is negative, predicting the branch as not taken. Those negative elements are classified in the E class, where other elements are in the NE class. The element "-100" whose absolute value is greater than $NCDP$ is in the FE subclass. Remaining elements, -3, -4 are classified as SE elements.

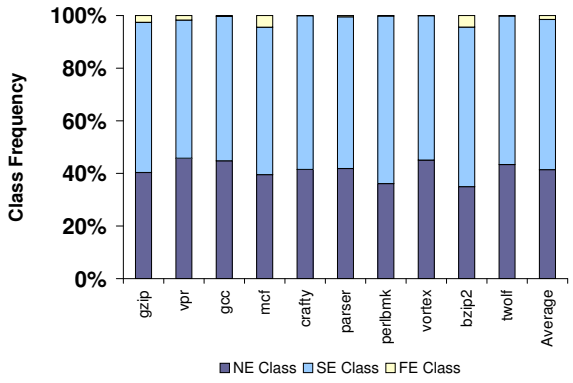


Figure 3.6: The frequency of different weight classes in a 128KBits perceptron predictor.

in conjunction with performing prediction, classifies the elements associated with the branch instruction. As the result, elements are classified to three classes of FE, SE and NE. In the occurrence of having at least one element in the FE class, all the elements in the SE and NE classes are believed to be noneffective and are disabled. This involves exclusion from subsequent operations the predictor performs for predicting branches corresponding to the current weight vector.

While an unnecessary element could indicate disabling both the history and the weight corresponding to the element, we only disable the weight. We avoid disabling history bits as they are used globally. Also they shift with every branch outcome and are needed for other dot products in the future cycles. We also avoid disabling the bias weight, as most of the branches are highly self-correlated.

To assure fast and efficient predictor learning we use our technique once we are confident that the predictor has passed the learning phase [9]. Accordingly, we disable elements if DP exceeds a dynamically decided threshold. This threshold is decided using Equation 3.1. As presented in Equation 3.1, this threshold should be directly proportional to the history length which is equal to the number of weights. By disabling noneffective weights we reduce the effective number of weights. By using Equation 3.1 we assure that the threshold is changed according to the number of enabled weights.

3.6 Restructuring the Predictor

In this section we discuss the modifications we propose to the predictor structure to accommodate the power optimizations we have proposed.

In order to classify elements, we start with determining each element's basic class (E or NE). This is done by comparing DP's sign and each element's sign.

We assume that an adder tree is used to calculate the dot product summation

[9]. In our scheme we use two adder trees, one for summing non-negative elements (referred here to as P_tree) and one for summing negative ones (referred here to as N_tree). The dot product value is obtained as:

$$DP = result(P\_tree) + result(N\_tree) \tag{3.6}$$

where $result(P\_tree)$ and $result(N\_tree)$ are the summation of elements included in P-tree and N-tree respectively.

Non-negative and negative elements are processed by the P-tree and N-tree respectively. We use $sign(e_i)$ to decide which tree to assign e_i to. Accordingly:

$$\begin{aligned} Bypass(Adder_i(P\_tree)) &\Leftrightarrow sign(e_i) = 1 \\ Bypass(Adder_i(N\_tree)) &\Leftrightarrow sign(e_i) = 0 \end{aligned} \tag{3.7}$$

Note that by using the final dot product's sign, $sign(DP)$, we can determine the tree including each element class. The tree including elements with signs opposite of DP's sign includes NE elements and is computing $NCDP$.

Using the $NCDP$ value, further classification of elements in the E class is possible. To do so, we compare each E element with $NCDP$. If the element's value is greater than $NCDP$, we mark it as an FE, otherwise we mark it as an SE. At the end, if there is at least one element in the FE class, all elements in the NE and SE classes are noneffective and can be disabled.

3.6.1 Implementation

In Figure 3.7 we show the schematic of the extended adder we propose to form N- and P-trees. The extended adder relies on bypassing signals to bypass one or both inputs (for example, input-1 is directly sent to the output if input-0 is bypassed). If both inputs should be bypassed (e.g., both inputs represent NE weights), the output bypass signal is set to "1", directing the next extended adder in the hierarchy to

bypass the associated input. With any bypassed input the extended adder no longer performs any computation and could be power gated [16].

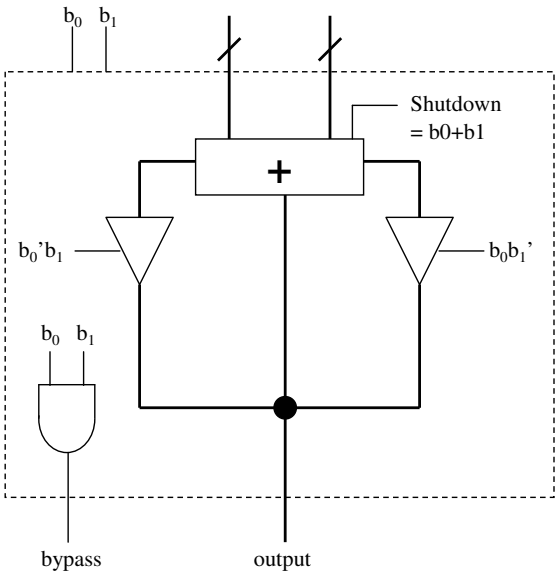


Figure 3.7: The extended adder used in the implementation of the adder tree capable of bypassing inputs.

Compared to a conventional adder, the extended adder comes with an overhead. This overhead mainly consists of the bypass logic. The output bypass signal is generated using a single AND gate overhead. In this study we take this energy overhead into account (more on this in Section 3.7).

While both trees receive all inputs, trees are directed to bypass some of their inputs using bypass signals. An element is bypassed in both trees if it is currently disabled. Negative elements are disabled in the P-tree. Similarly, positive elements are disabled in the N-tree. The bypassing signals can be represented using the following:

$$\begin{aligned} P\_Bypass_i &= disabled_i + sign(e_i) \\ N\_Bypass_i &= disabled_i + \overline{sign(e_i)} \end{aligned} \tag{3.8}$$

Any element's sign is determined by XOR-ing its weight sign (weight's MSB) and the corresponding outcome history:

$$sign(e_i) = MSB(w_i) \oplus h_i \tag{3.9}$$

To decide the branch outcome we compare the absolute values of the partial sums obtained from the two trees. The greater value decides the prediction outcome. Once the outcome is known, element classification can start immediately and in parallel with the instruction execution.

By using the $NCDP$ (the value of the tree with the lesser absolute value), we identify NE, SE and FE elements. We classify elements with a sign different from the prediction direction as an NE element.

$$NE_i = \overline{prediction} \oplus sign(e_i) \tag{3.10}$$

Furthermore, SE and FE elements are identified using already determined NE signals. Those elements not being classified as NE with a value less than $NCDP$'s absolute value are marked as SE. This can be done easily as such elements have a sign different from $NCDP$. Furthermore, FE elements are those which are not classified as NE nor SE.

$$\begin{aligned} SE_i &= \overline{NE_i} \;.\; ((w_i \oplus sign(w_i)) < NCDP) \\ FE_i &= \overline{NE_i} \;.\; \overline{SE_i} \end{aligned} \tag{3.11}$$

Figure 3.8 reports how the extended adder is exploited in our scheme. The element vector which is shown by e_is is the product of the weight vector and outcome

history vector. The b_is vector decides which elements should be disabled and could be removed from the dot product computation.

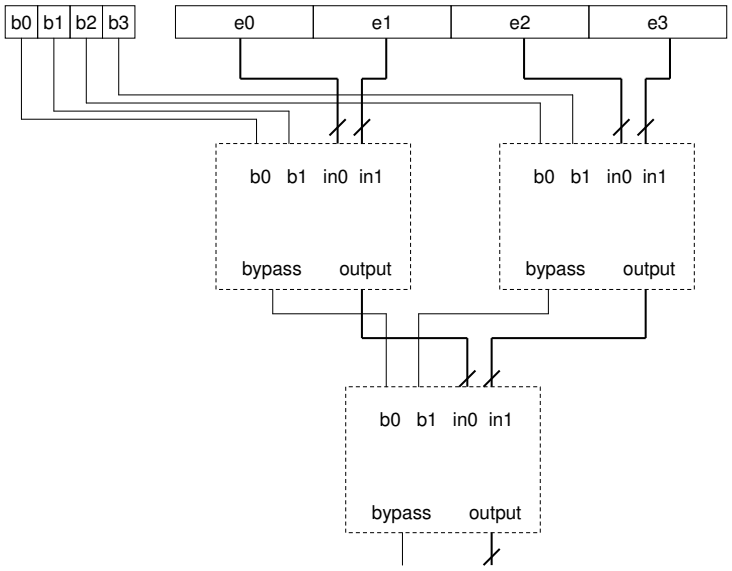


Figure 3.8: Using the extended adder in a 4 input adder tree as an adder/bypassing module.

3.7 Overhead

The optimizations proposed for the predictor come with timing and power overheads. In this section we study these overheads.

3.7.1 Timing Overhead

The computation delay associated with the adder tree is decided by the maximum number of sequential additions performed. Under the worst case, where no adder is bypassed, the delay is equal to $log(e)$ times single adder delay, where e is the number of elements. For extended bypassed adders, however, the delay is equal to

the propagation delay of the bypass logic. This significantly reduces the adder delay and hence the overall tree delay. When noneffective inputs are bypassed, the number of sequential additions performed in the tree is reduced to $log(e - d)$, where d is the number of disabled elements.

Although identifying and disabling noneffective weights reduces prediction time, weight classification comes with timing overhead. The timing overhead should not impact critical path delay as classification can be postponed to the predictor update time and long before the next outcome is calculated. To provide better understanding, however, we evaluate our technique assuming two extreme timing scenarios. In the first (optimistic) scenario, we assume that the timing overhead does not impose any additional restrictions. Under this scenario, both SE and NE elements are identified and disabled. In the second (pessimistic) scenario we assume that timing complexities would not allow identifying both NE and SE elements. Note that NE elements are easier to detect as they have a different sign compared to the outcome. Detecting SE elements, on the other hand, is more complicated and requires an extra comparison to be performed.

3.7.2 Power Overhead

We have measured the power overhead associated with identifying SE and NE elements separately (see Section 3.8 for more details). Our results show that the average overhead associated with identifying NE elements (or the pessimistic timing scenario) is about 30% of the original design's power dissipation (the original design's adders dissipate 9.9 uW while the extra logic overhead comes with an extra 3.3uW). Under the optimistic timing scenario, where both NE and SE elements are identified, our measurements indicate relatively higher power overhead for some configurations. As we show in Section 3.8, for some configurations, identifying both NE and SE elements, while improving performance considerably, may not be justifiable from the

energy point of view. For this group of configurations it would make more sense to limit our technique to removing NE elements.

From the area point of view, our design increases the predictor's area usage by 3% for a 4KBytes perceptron. This increase in area is mainly due to the duplicated adder tree we use for classification purpose. Also the extra logic used in the extended adders increases the circuit area.

3.8 Results and Analysis

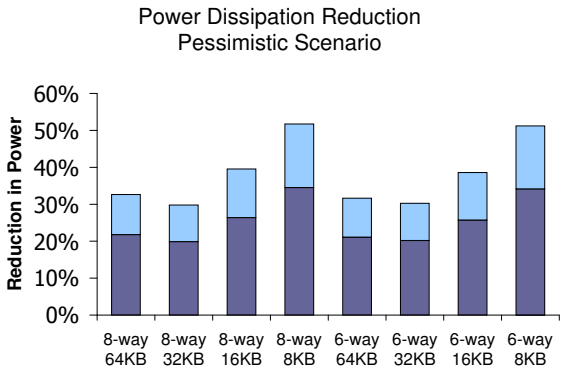
In this section we evaluate the optimizations for the perceptron branch predictor. We report both power and prediction delay reduction. We also report how eliminating noneffective calculations impacts performance and misprediction rate.

For power reports, we synthesize the perceptron predictor circuits, both conventional [9] and the optimized one propose in this chapter. We also use SimpleScalar tool set [3] running SPEC-2K integer benchmarks to obtain predictor accuracy and processor performance. In Appendix-A, we describe the experimental methodology and environment used to obtain the results in more detail.

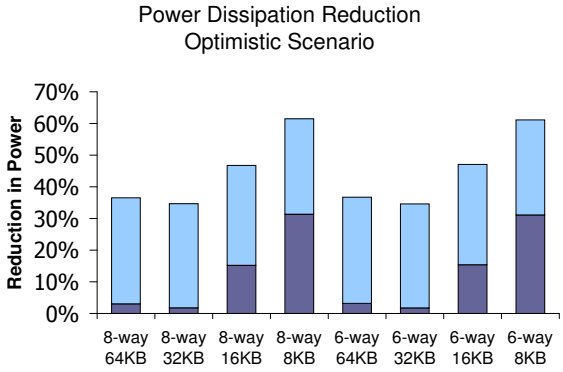
3.8.1 Power Dissipation Reduction

Figure 3.9 reports the predictor average computational power dissipation reduction under both optimistic and pessimistic timing scenarios (for 6- and 8-way processors) having different predictor hardware budgets (8, 16, 32 and 64 Kbytes). We report raw energy savings, the overhead associated with the extra logic and net energy savings.

Under the pessimistic timing scenario, average reduction in power dissipation is 25%. Under the optimistic timing scenario, as the result of the extra overhead, energy savings reduces to 12%. Note that for four of the configurations (i.e., configurations using 64KB and 32KB budgets) net savings are very low under the optimistic scenario.



(a)



(b)

Figure 3.9: The entire bar reports potential power reduction under by disabling elements. The lower bar shows the reduction after paying the overhead for the bypass logic. a) The pessimistic scenario, b) The Optimistic scenario. Results are shown for 8, 16, 32 and 64 Kbytes of hardware budget for 6- and 8-way processors.

For this group of configurations eliminating only the NE elements seems a reasonable approach.

For four configurations (i.e., configurations with 8KB and 16KB budgets) net energy savings are considerable under both timing scenarios.

As an example, by looking at the second bar of the Figure 3.9-a, we can see that for an 8-way processor using a 32KBytes of budget for the predictor, power dissipation can be reduced as much as 30%. However, the extra logic comes with 10% power overhead, which reduces the net power savings to 20%.

3.8.2 Prediction Delay Reduction

As explained earlier we assume that our technique reduces the computation latency and does not impact the table access latency. We assume that computation latency is directly proportional to the logarithm of number of participating weights. Since we use two adder trees, the overall computation delay is the maximum of each tree's delay. Since each tree's computation delay depends on the number of weights participating in that tree, the tree delay reduces as weights are disabled. In Figure 3.10 we report overall delay reduction under both timing scenarios for 6- and 8-way processors having different predictor budgets. On average, delay is reduced by at least 17% for both scenarios.

3.8.3 Prediction Accuracy

Figures 3.11, 3.12 report misprediction rate for a conventional perceptron and the low-power perceptron proposed in this chapter under both optimistic and pessimistic timing scenarios. The results are shown for 8, 16, 32 and 64 Kbytes of hardware budgets for both 6- and 8-way processors. As illustrated, there is a slight increase in the misprediction rate. As we show in the next section the performance cost associated with this increase is less than the improvements achieved by reducing latency.

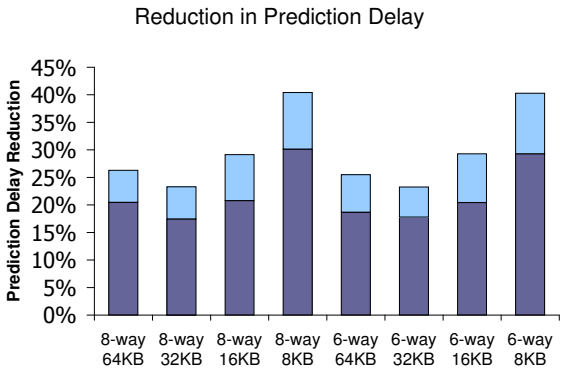


Figure 3.10: The entire bar reports reduction in prediction delay under the optimistic scenario. The lower bar shows the reduction under the pessimistic scenario. The results are shown for 8, 16, 32 and 64 Kbytes of hardware budget for 6- and 8-way processors.

3.8.4 Performance

In Figures 3.13, 3.14 we report performance for processors using the conventional perceptron and the low-complexity perceptron under both scenarios. On average and for the majority of applications, the low-power perceptron outperforms the conventional perceptron, under both timing scenarios. This is the result of lower prediction delay achieved by eliminating extra calculations. As reported performance improvement can be as high as 19% and 16% for optimistic and pessimistic scenarios for different configurations respectively.

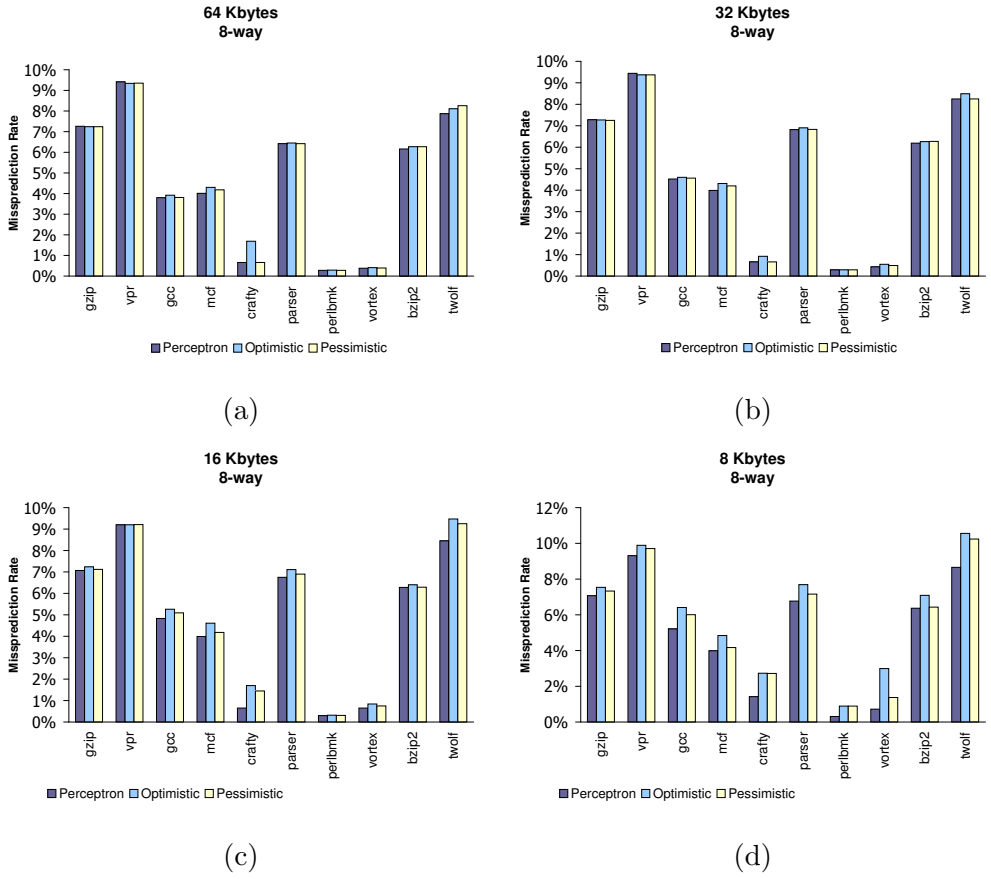


Figure 3.11: Branch misprediction rate for the conventional and low-power perceptron predictor (for both optimistic and pessimistic scenarios). Predictor and processor configurations are (a) 64Kbytes, 8-way, (b) 32Kbytes, 8-way, (c) 16Kbytes, 8-way, (d) 8Kbytes 8-way.

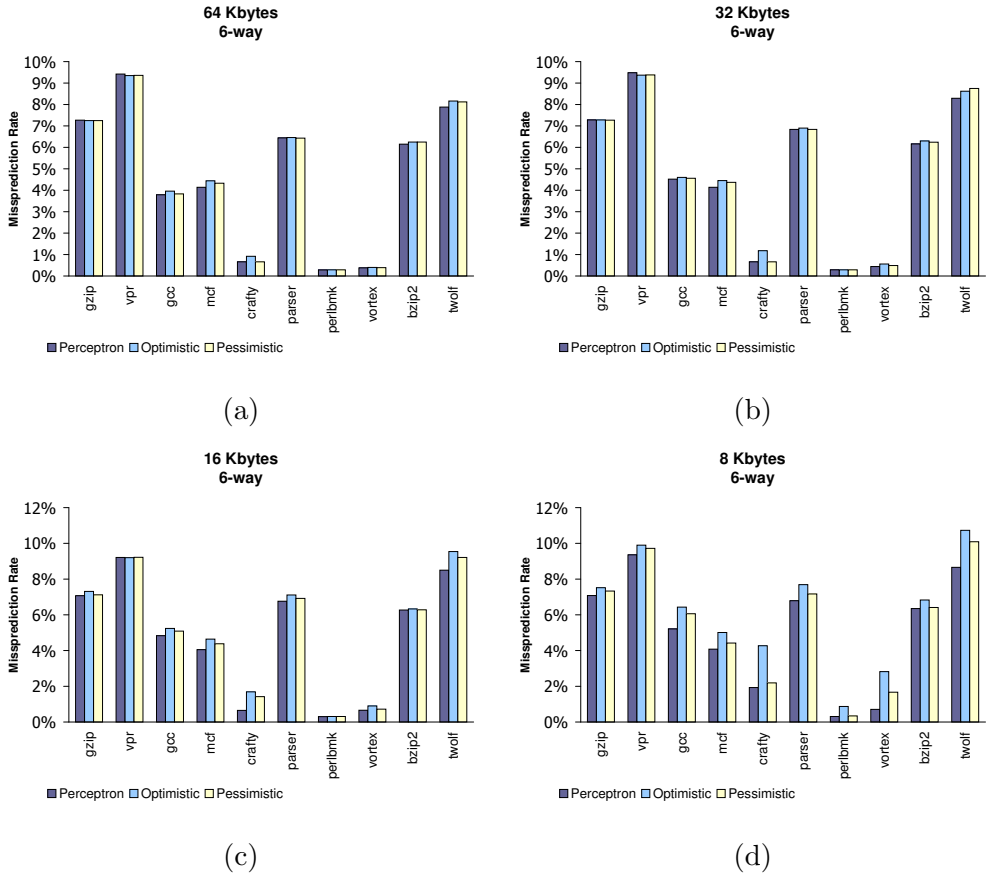


Figure 3.12: Branch misprediction rate for the conventional and low-power perceptron predictor (for both optimistic and pessimistic scenarios). Predictor and processor configurations are (a) 64Kbytes, 6-way, (b) 32Kbytes, 6-way, (c) 16Kbytes, 6-way, (d) 8Kbytes 6-way.

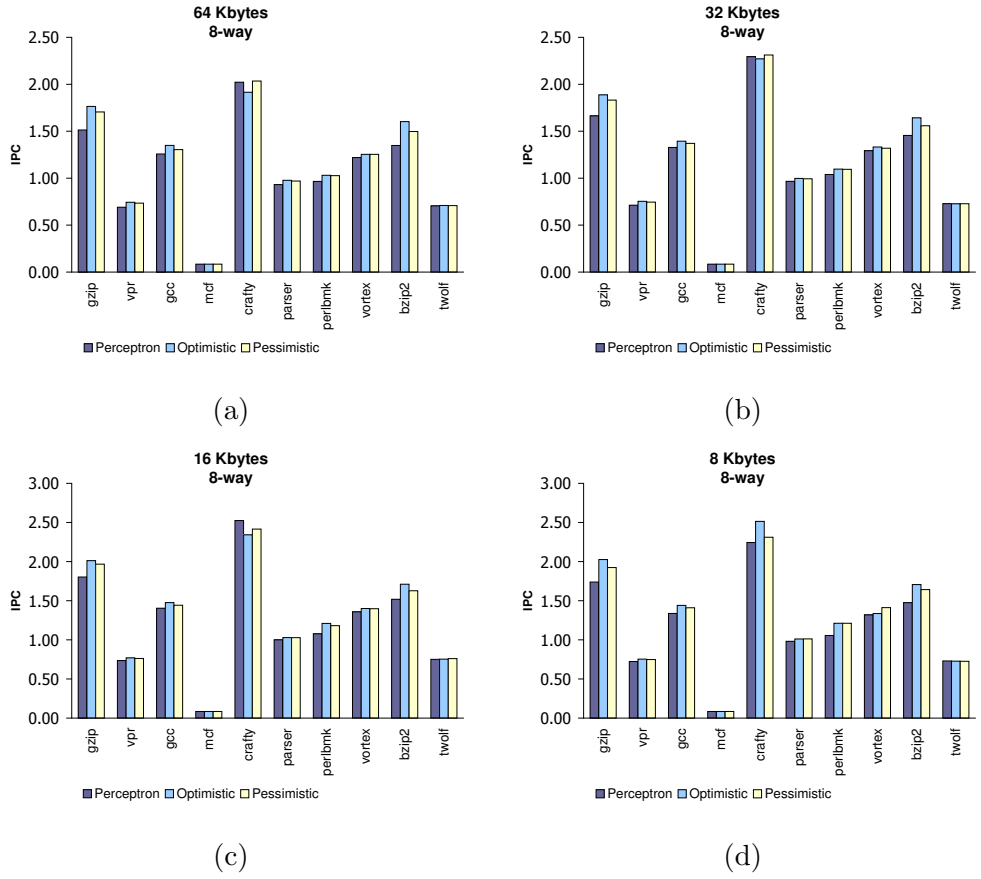


Figure 3.13: Performance for processors using the conventional and low-power perceptron predictors (both under optimistic and pessimistic scenarios). Predictor and processor configurations are (a) 64Kbytes, 8-way, (b) 32Kbytes, 8-way, (c) 16Kbytes, 8-way, (d) 8Kbytes 8-way.

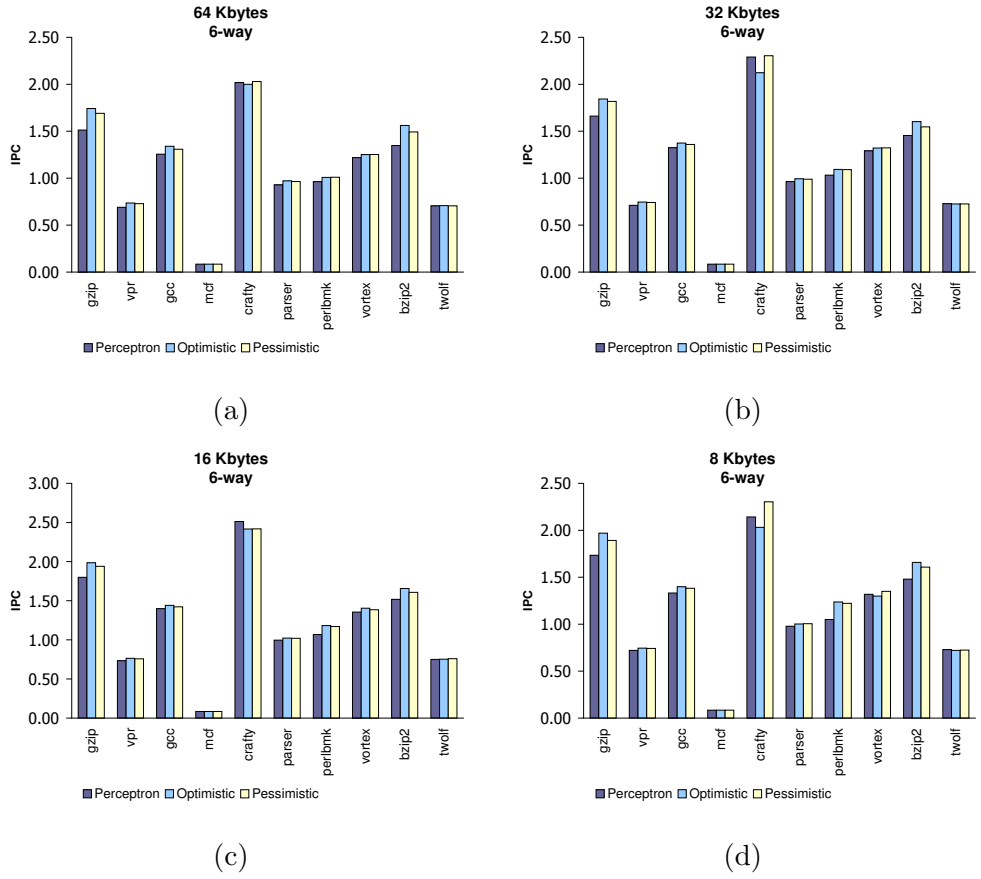


Figure 3.14: Performance for processors using the conventional and low-power perceptron predictors (both under optimistic and pessimistic scenarios). Predictor and processor configurations are (a) 64Kbytes, 6-way, (b) 32Kbytes, 6-way, (c) 16Kbytes, 6-way, (d) 8Kbytes 6-way.

Chapter 4

Power-Aware O-GEHL Branch Predictor

In this chapter we introduce a power-aware alternative for the O-GEHL branch predictor. We identify noneffective operations performed by the predictor and suggest mechanisms to eliminate them. We modify the predictor structure to accommodate our optimizations and reduce both prediction latency and power dissipation.

4.1 Introduction

The Optimized GEometric History Length (or simply O-GEHL) predictor is an example of a perceptron like predictor. O-GEHL relies on exploiting behavior correlations among branch instructions. To collect and store as much information as possible, the O-GEHL branch predictor uses multiple tables equipped with wide counters. The predictor uses the collected data and performs many steps before making the prediction. These steps include reading several counters from the tables and performing several computations (e.g., additions and comparisons) on the collected data.

In this chapter, we revisit the O-GEHL predictor and show that while the conventional scheme provides high prediction accuracy, it is not efficient from the energy point of view. We are motivated by the following observations. First, our study shows that not all the computations performed by O-GEHL are necessary. This is particularly true for computations performed on counter lower bits. As we show later, not all counter bits always impact the prediction outcome. Therefore excluding less important bits from the computations, while reducing energy consumption, may not impact accuracy. Second, we have observed that the tables used by O-GEHL store noneffective data. We show that the stored data can be represented using less storage

if this redundancy is taken into account.

We rely on the above observations and introduce two power optimization techniques. Our techniques aim at reducing the power dissipated by the computation and storage resources. We reduce power for computation resources by eliminating unnecessary and noneffective counter bits from computations and by accessing and using fewer bits at the prediction time. We reduce power for storage resources by representing the required data using fewer bits. We achieve this by having multiple counters share their lower bits. We show that by intelligent bit sharing it is possible to reduce predictor size while maintaining its accuracy. It should be noted that since our optimizations are not performed dynamically, they come with no latency or power overhead at runtime.

By applying our techniques we not only reduce power but also improve processor performance. This is due to the fact that by eliminating unnecessary computations we reduce prediction latency, resulting in a faster yet highly accurate prediction. We reduce the dynamic and static power dissipation associated with predictor computations by 74% and 65% respectively while improving performance up to 12%.

In Section 4.2 we discuss the O-GEHL branch predictor. In Sections 4.3 and 4.4 we discuss our motivation and show that O-GEHL uses noneffective information and storage for predictions. We propose our optimizations in Section 4.5. In Section 4.6, we provide simulation results and analyze the optimized predictor.

4.2 The O-GEHL Branch Predictor

O-GEHL is a highly accurate, perceptron based branch predictor. The ability to exploit long history lengths makes O-GEHL superior to conventional table based branch predictors. O-GEHL relies on exploiting behavior correlations among branch instructions. To collect and store as much information as possible, O-GEHL uses

multiple tables equipped with wide counters. The predictor uses the collected data and performs many steps before making the prediction. These steps include reading several counters from the tables and performing several computations (e.g., additions and comparisons) on the collected data.

4.2.1 Predictor Components

O-GEHL stores behavioral correlation information as signed integer numbers. For each branch instruction, several counters from different tables are used to predict the branch outcome. Each table is indexed with a different hashing function in order to reduce destructive aliasing and collect behavioral correlation among branch instructions [18].

The predictor uses as many hashing functions as the number of tables used to store counters. Each hashing function exploits a portion of the branch outcome history and the branch address to index a table entry. Hashing functions are differentiated by using different history lengths in their functions. The history lengths used in hashing functions form a geometric history length which enables the predictor to exploit correlations in short histories, as well as long ones.

The O-GEHL predictor uses only branch instructions global history to perform predictions. As storing local history increases predictor's latency and complexity, O-GEHL does not use it.

O-GEHL, being a perceptron like predictor, requires an adder tree to perform the accumulation process. The size of the required adder tree depends on the number of counters involved in the accumulation process and the width of the counters.

4.2.2 Prediction Procedure

As presented in Figure 4.1, the O-GEHL predictor takes the following steps to make a prediction. First, hashing functions generate the indexes for each table, based

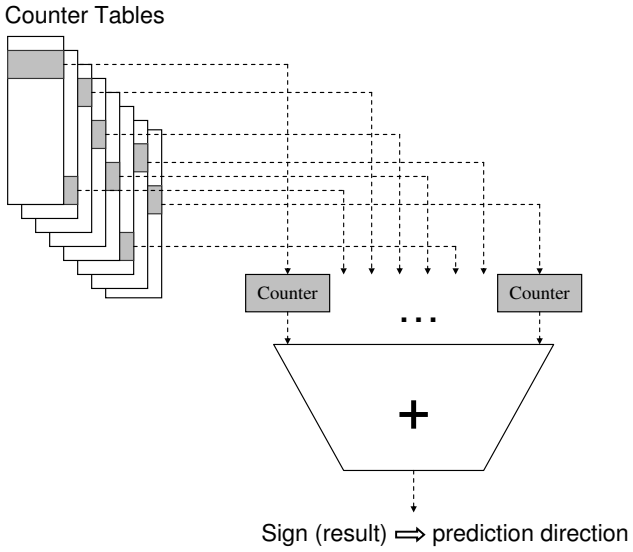


Figure 4.1: The O-GEHL branch predictor using multiple tables and different indexes. Sum of all the counters is used to make the prediction.

on the branch instruction's address and the outcome history. The history length used in the hashing function is determined by Equation 4.1, in which M is the number of tables and $L(1)$ and $L(M)$ are chosen by the designer.

$$\begin{aligned} L(0) &= 0 \\ L(i) &= \alpha^{i-1} L(1) \\ \alpha &= \sqrt[M-2]{\tfrac{L(M-1)}{L(1)}} \end{aligned} \tag{4.1}$$

Second, the predictor loads counters from different tables using the corresponding indexes generated by the hashing functions. Third, an adder tree accumulates all the counters and provides the predictor with a signed value. Additionally, an offset of $\frac{M}{2}$ is always added to this value. Fourth, the predictor makes the prediction based on the sign of the summation, where a non-negative value indicates a taken and a

negative value indicates a not taken branch.

4.2.3 Update Procedure

At the update time, the O-GEHL predictor updates the counters used to predict the branch outcome at the prediction time. The predictor, using Equation 4.2, increments or decrements the counters based on the actual outcome of the branch instruction. As stated in Equation 4.2, the changes applied to the counters are done independent of the outcome history.

$$0 \leq i < M, counter_i = \begin{cases} counter_i + 1 & taken \\ counter_i - 1 & not-taken \end{cases} \tag{4.2}$$

To avoid predictor over-training, the predictor uses a pre-decided threshold. If the absolute value of the summation used at the prediction time exceeds the threshold, the predictor is believed to be well-trained and is no longer updated. In the case of a misprediction, the predictor updates the counters regularly, regardless of the summation value.

The O-GEHL predictor dynamically updates its threshold. In two situations the predictor is updated, one is when the predictor mispredicts, and the other is when the summation value is less than the threshold. In [18], the optimal threshold is observed to be causing the number of updates in two situations be in the same range. A simple algorithm based on a saturating counter is exploited to update the threshold. The threshold is initialized by $M/2$. On every misprediction, the counter is incremented. The counter is decremented if summation value is below the threshold. If the counter saturates positive, the threshold is incremented. On the other hand, if the counter saturates negative, the threshold is decremented.

4.3 Identifying Noneffective Operations

As presented in Figure 4.1, the O-GEHL predictor uses multiple counters per branch instruction. For every direction prediction, the predictor computes the sum of all the indexed counters.

The complexity of the computations involved in the summation process makes it a slow and energy hungry one. This process requires an adder tree, with a size and complexity proportional to the counters' widths. The wider the counters are, the more complex the summation process will be. Note that the 64Kbits O-GEHL predictor [17] uses a combination of 4- and 5-bit counters to achieve the best accuracy.

Our study shows that not all counter bits are equally important in making accurate predictions. In particular, higher order bits are more likely to impact the final outcome compared to lower order bits. In Figure 4.2 we present an example to provide better understanding.

Counter Values

Decimal	1	-3	2	-1	-8	5	4	-2
Binary	0001	1101	0010	1111	1000	0101	0100	1110

Result (All Bits) = 1 + -3 + 2 + -1 + -8 + 5 + 4 + -2 = -2 → Not Taken
Result (High Bits) = 0 + -1 + 0 + -1 + -2 + 1 + 1 + -1 = -3 → Not Taken

Figure 4.2: The first calculation uses all counter bits and predicts the branch outcome as "Not taken". The second one uses only higher bits (underlined) and results in the same direction.

To investigate this further, in Figure 4.3 we report how excluding the lower n bits of each counter impacts prediction outcome. As reported, on average, 0.4%, 0.9%, and 1.8% of time eliminating the lower one, two or three bits results in a different outcome respectively. This difference is worst (2.7%) when the lower three bits are excluded for *twolf* (See Appendix-A for methodology).

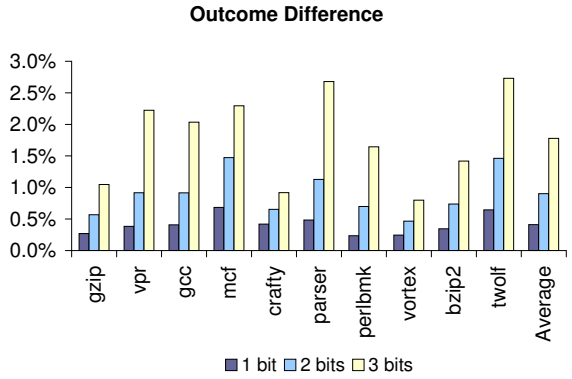


Figure 4.3: How often removing the lower n bits from the computations results in a different outcome compared to the scenario where all bits are considered. Bars from left to right report for scenarios where one, two or three lower bits are excluded.

We conclude from Figure 4.3 that eliminating lower order bits (referred to as LOBs) of the counters from the prediction process and using only higher order bits (referred to as HOBs) would not significantly affect the predictor's accuracy. We use this observation in Section 4.5 and reduce predictor's latency and power dissipation.

4.4 Identifying Noneffective Storage

The formal definition of the information a message carries depends on the probability of that message. The information carried by each message can be represented as:

$$I = -log_2 P \tag{4.3}$$

where P is the message occurrence probability.

Shannon's theorem states that number of bits needed to represent a messages set is equal to its entropy. Entropy is defined as:

$$H = \sum_{i \in M} P_i I_i \tag{4.4}$$

where M is the message set. If we have a b-bit counter, we have 2^b possible messages. Therefore we can compute its entropy as:

$$H = \sum_{i=0}^{2^b-1} P_i I_i = \sum_{i=0}^{2^b-1} -P_i log_2 P_i \tag{4.5}$$

where P_i is the probability of the counter having a value of i . In the case of having equal probabilities for all counter values, message set entropy is equal to b bits, which is the number of bits we have already dedicated. However, if any message has a higher probability, message set's entropy decreases to a value less than b, meaning that it is possible to represent this message set with less than b bits.

Knowing the amount of information carried by each message in the set, we can derive the set's entropy. Therefore, we can determine the actual number of bits required for representation.

To study whether the entropy of the counters stored in the O-GEHL tables is less than the number of bits dedicated, we use the following scheme. An m-bit counter with entropy of m, should be in zero or $2^m - 1$ states with a probability of $\frac{2}{2^m-1}$. Therefore, for a 3-bit counter, 12.5% of times the counter must be in each possible state. In Figure 4.4 we report how often the three lower order bits of the counters used by O-GEHL have the maximum or minimum possible values. As reported on average more than 60% of the time the counter value is biased. This suggests that using three bits per counter may be too much as not all counter values are equally frequent. This bias motivates us to reduce storage size by representing lower order bits using fewer resources (more on this in Section 4.5).

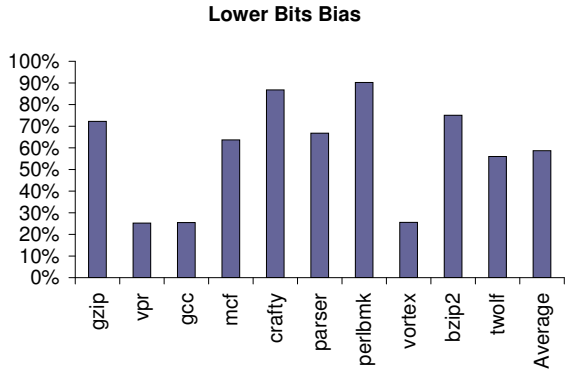


Figure 4.4: Bars show the percentage of time three lower bits of the counters are biased. On average 60% of time counters are biased compared to 25% on a uniform distribution.

4.5 Improving the Predictor

In this section we propose our optimizations for the O-GEHL branch predictor, based on observations discussed in Sections 4.3 and 4.4.

4.5.1 Eliminating Noneffective Operations

Considering the data presented in Section 3, we suggest eliminating the LOBs of the counters from the lookup and summation process. During predictor update, however, we increment/decrement counters without excluding any of the lower bits.

In Figure 4.5 we present our scheme. In this scheme, the adder tree does not load or use all counter bits. Instead, the adder tree bypasses the LOBs of the counters, and performs the accumulation process only on the HOBs. Eliminating LOBs reduces power but can impact accuracy and performance.

A) Accuracy and Performance: The number of input bits decides the adder

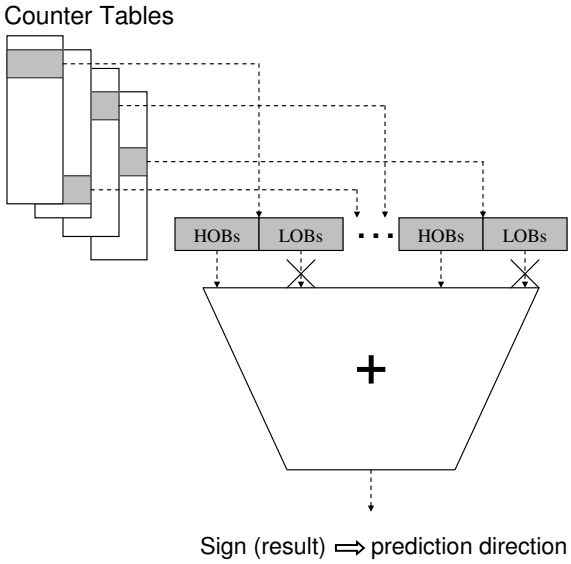


Figure 4.5: The optimized adder tree bypasses LOBs of the counters and performs the addition only on the HOBs. Eliminating LOBs results in a smaller and faster adder tree.

tree size used in the O-GEHL predictor. Excluding LOBs from the computation reduces adder tree's input size resulting in a shorter computation time. This makes faster prediction possible but can potentially harm accuracy.

A previous study on branch prediction delay shows that further prediction accuracy improvement may not be worthwhile if it results in a slower prediction scheme [8]. Reportedly, a relatively accurate single-cycle latency predictor outperforms a 100% accurate predictor with two cycles latency [8]. This observation motivates us to study whether the prediction speedup obtained by eliminating LOBs is worth the accuracy cost. In Section 4.6 we investigate this trade-off and show that for the benchmarks used in this study the performance improvements achieved by faster prediction outweigh the cost associated with the extra mispredictions.

b) Power: We reduce both the dynamic and the static power dissipated by the predictor. As we reduce adder tree's size, fewer gates are used in its structure. Consequently, it dissipates less static power. We also reduce dynamic power as fewer switching activities occur.

4.5.2 Splitting Predictor Tables

The optimization proposed in Section 4.5.1 suggests eliminating LOBs from the computations necessary at the prediction time. Therefore when retrieving counters from predictor tables, reading all bitlines is no longer necessary. One straightforward mechanism to make this possible is to decouple LOBs and HOBs. Accordingly, we store LOBs and HOBs in two separate set of tables. Figure 4.6 shows this in more detail.

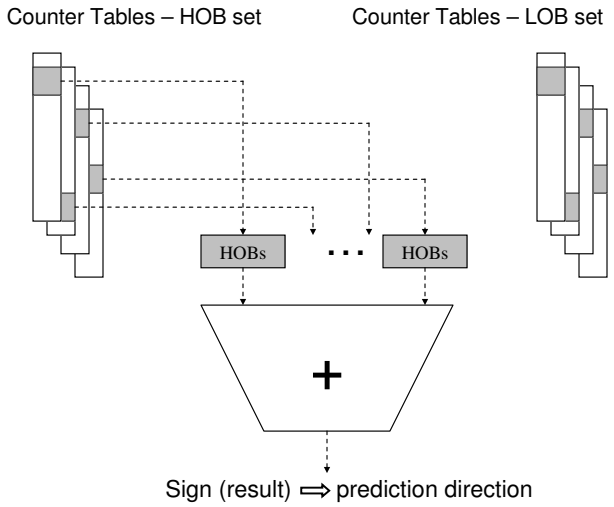


Figure 4.6: Predictor tables are divided into two sets, HOB set and LOB set. Only the HOB set is accessed at the prediction time in order to reduce power dissipation.

At the prediction time, the predictor accesses only the set of tables storing the

HOBs, saving the energy consumed for accessing LOBs in the conventional O-GEHL predictor. Note that while we save the energy spent on wordline, bitline and sense amplifiers, we do not reduce the decoder energy consumption as we do not reduce the number of table entries for this optimization.

4.5.3 Entropy-Aware Storage

In this Section we propose a scheme to reduce predictor tables' sizes. We measure the information conveyed by the counters and compute their entropy and the necessary number of bits required to carry this information. Knowing the exact entropy of these bits, we can avoid dedicating unnecessary storage.

Note that the entropy of the counter may not always be an integer number, requiring several entries sharing a single bit. To this end, we exploit sharing groups in which counters share their bits. Under this situation, the group size determines the real number of bits being dedicated to each counter. A previous study has used a similar approach to share the hysteresis bits of the Gshare predictor [12]. In Figure 4.7 we show an example clarifying this further.

To reduce storage size we exploit the bias of the LOBs presented in Section 4.4, and change the information representation. In order to measure LOBs' entropy, we record counters values at prediction time. We compress the stored information using gzip program with "–best" flag. Similar to previous study we use the compression ratio to estimate the entropy of the message set [12].

Figure 4.8 shows the compression ratio for the data carried by three lower bits of the O-GEHL counters. As reported on average less than 25% of current reserved bits suffice to carry the information. In order to facilitate using fewer bits, we have multiple entries share their LOBs. We form counter groups of up to size four, effectively avoiding using the noneffective storage. However, and in order to maintain accuracy, conservatively, we use a group size of two where every two counters share their LOBs.

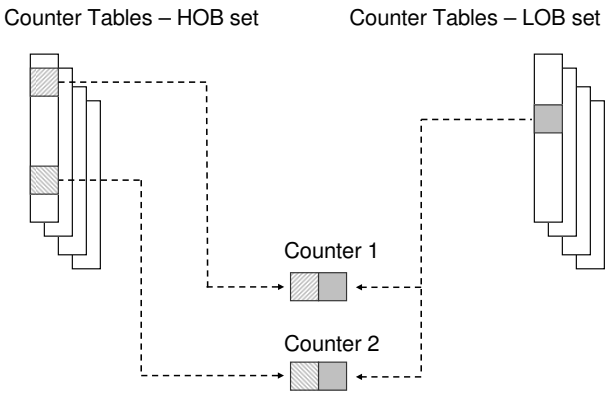


Figure 4.7: Two counters share their LOBs. The predictor indexes the same LOB entry for counters using different HOBs.

Note that having a power of two group size simplifies implementation issues. As a result, we reduce the table size by half.

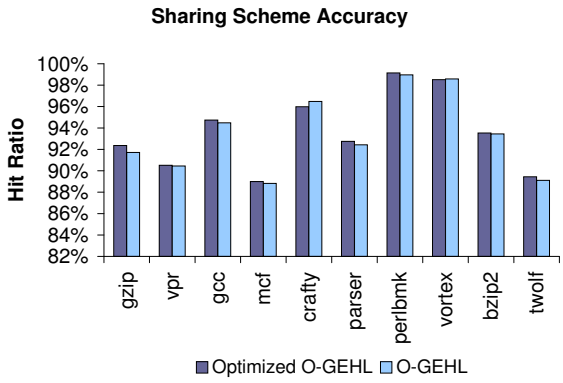


Figure 4.8: Compression ratio for data carried by three LOBs. On average, the data can be compressed to one-fourth of its size.

4.6 Results and Analysis

In this section we validate the optimizations proposed to the O-GEHL branch predictor. We evaluate our scheme with respect to timing, power dissipation, prediction accuracy and processor performance. We describe the experimental methodology in full details in Appendix-A.

4.6.1 Timing

Figure 4.9 reports time (in nanoseconds) and the number of cycles required to compute the predictor computation result. We report results for the original O-GEHL predictor and three different optimized ones. As reported, the original predictor takes 5 clock cycles to compute the result. By eliminating one bit from the computation process no clock cycle is saved. While our study shows that by removing two or three bits one and two clock cycles can be saved respectively, in this study we take a conservative approach and assume that only one clock cycle is saved after eliminating two or three lower bits.

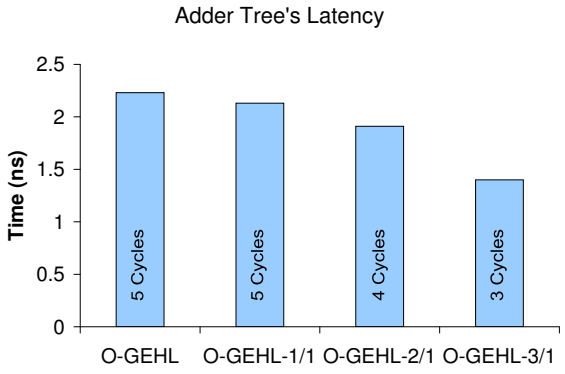


Figure 4.9: Time / Cycle required to compute the sum of counters.

Table 4.6.1 reports predictor's table access time for four different O-GEHL predictors. Since the optimized predictor accesses only the LOB tables at the prediction time, access time is slightly lower.

Table 4.1: Table Access Time / Power Dissipation

Predictor	Access Time (ns)		Energy (pJ)	
	Predict	Update	Predict	Update
O-GEHL	1.15	1.15	167.04	167.04
O-GEHL-1/1	1.15	1.15	149.02	167.55
O-GEHL-2/1	1.10	1.10	130.68	167.52
O-GEHL-3/1	1.08	1.10	112.34	167.52

4.6.2 Power Dissipation

Figure 4.10 reports the reduction in both leakage and dynamic energy consumption for the predictor's adder tree. Results are obtained by gate level synthesis of the circuit. Eliminating one bit saves up to 41% of the dynamic energy and 44% of the static energy consumption. Energy reduction is higher when the number of eliminated bits increases. This is the result of exploiting smaller adders.

Table 4.6.1 reports the power dissipated while accessing HOB and LOB tables. At the prediction time, only LOB tables are accessed whereas at the update time, both sets are accessed. Since the same decoder is used for each pair of tables, accessing two tables at the update time does not impose any noticeable power overhead.

4.6.3 Prediction Accuracy

As we use fewer bits for making prediction, we can potentially harm accuracy. To investigate this further in Figure 4.11 we compare prediction accuracy for five

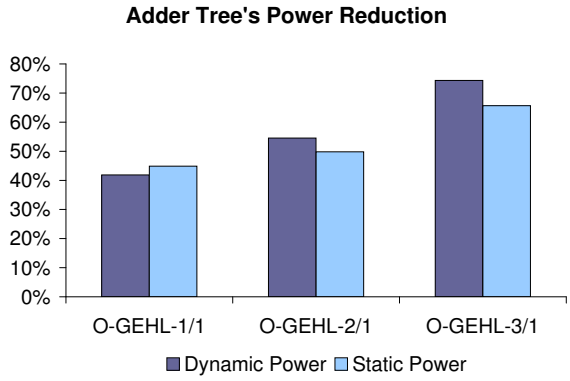


Figure 4.10: Energy reduction of the adder tree compared to conventional O-GEHL. Results are shown for O-GEHL-1/1, O-GEHL-2/1 and O-GEHL-3/1.

different predictors: The original O-GEHL predictor, O-GEHL-1/1, O-GEHL-2/1, O-GEHL-3/1 and O-GEHL-3/2. On average prediction accuracy is decreased by 0.06%, 0.1%, 0.3% and 0.5% for O-GEHL-1/1, O-GEHL-2/1, O-GEHL-3/1 and O-GEHL-3/2 respectively. Maximum accuracy loss is 1.1% for *twolf* under O-GEHL-3/2.

Sharing the LOBs would result in an overall smaller storage. To make better evaluation of our scheme possible, we also compare the accuracy of an optimized O-GEHL predictor (sharing groups of size two for three low order bits) to a conventional O-GEHL using the same real-estate budget (but not sharing the LOBs). As reported in Figure 4.12, the optimized O-GEHL predictor achieves higher accuracy across all applications except for *crafty*.

4.6.4 Performance

Figure 4.13 reports processor's overall performance compared to a processor using the original O-GEHL predictor. We report for four different processors using O-

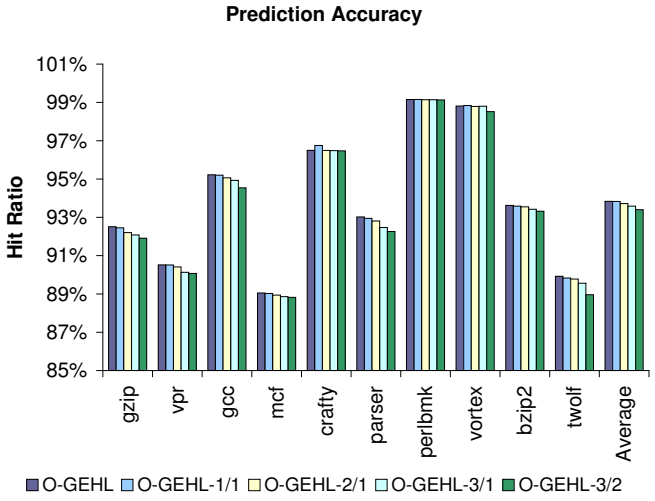


Figure 4.11: Prediction accuracy for the conventional O-GEHL predictor and four optimized versions, O-GEHL-1/1, O-GEHL-2/1, O-GEHL-3/1 and O-GEHL-3/2. The accuracy loss is negligible.

GEHL-1/1, O-GEHL-2/1, O-GEHL-3/1 and O-GEHL-3/2 branch predictors. As reported on average we improve performance by 4.7%, 4.6%, 4.3% and 3.9% for O-GEHL-1/1, O-GEHL-2/1, O-GEHL-3/1 and O-GEHL-3/2 respectively. Although the optimized O-GEHL predictors achieve slightly lower accuracy compared to the original one, the overall processor performance is higher. As explained earlier, this is the result of achieving faster prediction by eliminating LOBs.

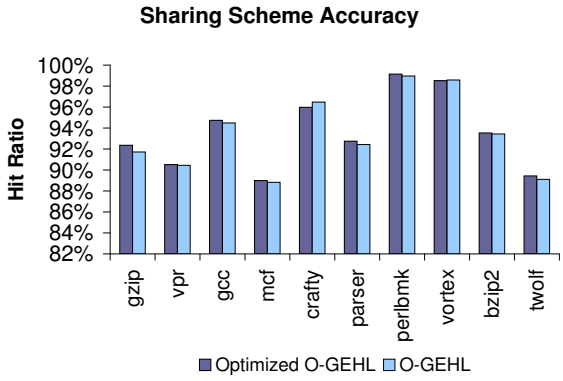


Figure 4.12: Accuracy for an optimized O-GEHL predictor (sharing groups of size two for three LOBs) and a conventional O-GEHL using the same real-estate budget.

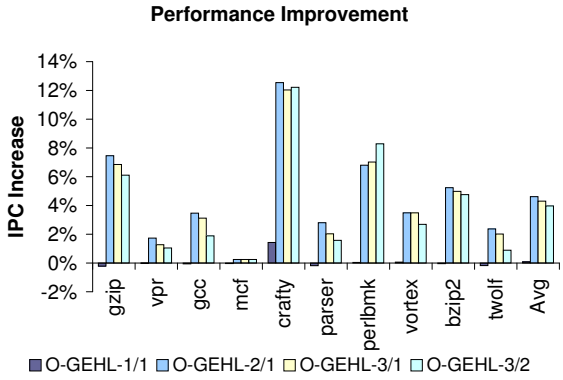


Figure 4.13: Performance improvement compared to a processor using the conventional O-GEHL predictor. Results are shown for processors using O-GEHL-1/1, O-GEHL-2/1, O-GEHL-3/1 and O-GEHL-3/2 predictors.

Chapter 5
Conclusions and Future Works

The perceptron branch predictor is one of the most accurate predictors. However, the energy and latency overhead associated with this predictor imposes practical restrictions. We reduced the predictor power dissipation by identifying and eliminating non-effective computations used in the predictor. We showed that unnecessary calculations exist in this predictor and can be eliminated without significant impact on the predictor accuracy. Our optimizations to the perceptron branch predictor only impose an extra silicon area requirement as we increase predictor components.

We also studied the O-GEHL branch predictor and showed that by identifying and eliminating unnecessary computations, both static and dynamic power dissipations can be reduced considerably. We also showed that avoiding such computations reduces prediction latency, which results in better performance.

Moreover, we suggested exploiting sharing groups in which counters share their bits to reduce the necessary storage. Using sharing groups, we reduce predictor's table sizes and therefore power dissipation.

Our optimizations for the O-GEHL branch predictor come with no overheads as we do not perform any operations at runtime. Also, our optimizations do not require any extra hardware, as the predictor structure can be reconfigured to address our modifications.

5.1 Future Works

We have studied perceptron and O-GEHL branch predictors and realized that such predictors perform unnecessary computations and use noneffective storage. We have

based our studies upon software simulations which approximate processor behavior. However, real processor evaluations can be considered as a future work, as they might reveal new aspects which are not observable in software simulations.

Perceptron and O-GEHL branch predictors are highly accurate branch predictors. However, combining these predictors may result in a high accurate, low power predictor which outperforms both predictors. Combining the ability of exploiting long history lengths in O-GEHL and the higher adaptivity of perceptron can result in a more accurate, energy efficient branch predictor.

Bibliography

[1] Amirali Baniasadi and Andreas Moshovos. Branch predictor prediction: A power-aware branch predictor for high-performance processors. In *Proceedings of 20th International Conference on Computer Design (ICCD 2002)*, pages 458–461, 2002.

[2] Amirali Baniasadi and Andreas Moshovos. Sepas: A highly accurate energy-efficient branch predictor. In *Proceedings of the 2004 International Symposium on Low Power Electronics and Design*, pages 38–43, 2004.

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

[4] G. F. Grohoski. Machine organization of the ibm risc system/6000 processor. *IBM Journal of Research and Development*, 34:37–58, 1990.

[5] Eric Hao, Po-Yung Chang, and Yale N. Patt. The effect of speculatively updating branch history on branch prediction accuracy, revisited. In *The 27th annual International Symposium on Microarchitecture*, pages 228–232, 1994.

[6] Zhigang Hu, Philo Juang, Kevin Skadron, Douglas W. Clark, and Margaret Martonosi. Applying decay strategies to branch predictors for leakage energy savings. In *IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 442– 445, 2002.

[7] Michael C. Huang, Daniel Chaver, Luis Pinuel, Manuel Prieto, and Francisco Tirado. Customizing the branch predictor to reduce complexity and energy consumption. In *Proceedings of 33rd International Symposium on Microarchitecture*, pages 12–25, 2003.

[8] Daniel A. Jimenez, Stephen W. Keckler, and Calvin Lin. The impact of delay on the design of branch predictors. In *Proceedings of the 33rd International Symposium on Microarchitecture (MICRO-33)*, pages 66–77, 2000.

[9] Daniel A. Jimenez and Calvin Lin. Neural methods for dynamic branch prediction. *ACM Transactions on Computer Systems*, pages 369–397, 2002.

[10] Daniel A. Jimnez. Fast path-based neural branch prediction. In *The 36th annual IEEE/ACM International Symposium on Microarchitecture (MICRO-36)*, page 243, 2003.

[11] Stefanos Kaxiras, Zhigang Hu, and Margaret Martonosi. Cache decay: exploiting generational behavior to reduce cache leakage power. *ACM SIGARCH Computer Architecture News*, 29:240–251, May 2001.

[12] Gabriel H. Loh, Dana S. Henry, and Arvind Krishnamurthy. Exploiting bias in the hysteresis bit of 2-bit saturating counters in branch predictors. *Journal of Instruction Level Parallelism (JILP)*, 5:1–32, 2003.

[13] Gabriel H. Loh and Daniel A. Jimenez. Reducing the power and complexity of path-based neural branch prediction. In *Proceedings of the 5th Workshop on Complexity Effective Design (WCED)*, pages 1–8, 2005.

[14] Scott McFarling. Combining branch predictors. Technical report, Digital Western Research Laboratory, 1993.

[15] Dharmesh Parikh, Kevin Skadron, Yan Zhang, Marco Barcella, and Mircea Stan. Power issues related to branch prediction. In *Proceedings of the Eighth International Symposium on High-Performance Computer Architecture*, pages 233–, 2002.

[16] Michael Powell, Se-Hyun Yang, Babak Falsafi, Kaushik Roy, and T.N. Vijaykumar. Gated-v dd : A circuit technique to reduce leakage in deep-submicron cache memories. In *Proceedings of the International Symposium on Low-Power Electronics and Design (ISLPED)*, July 2000.

[17] Andr Seznec. The o-gehl branch predictor. The 1st JILP Championship Branch Prediction Competition (CBP-1), in conjunction with MICRO-37, 2004.

[18] Andr Seznec. Analysis of the o-geometric history length branch predictor. In *32nd Annual International Symposium on Computer Architecture*, 2005.

[19] John P. Shen and Mikko H. Lipasti. *Modern Processor Design: Fundamentals of Superscalar Processors*. McGraw-Hill Science/Engineering/Math, 2004.

[20] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. Automatically characterizing large scale program behavior. In *Proceedings of the Tenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2002.

[21] Jim. E. Smith. A study of branch prediction techniques. In *The 8th annual Symposium on Computer Architecture*, pages 135–147, 1981.

[22] David Tarjan, Shyamkumar Thoziyoor, and Norman P. Jouppi. Cacti 4.0. Technical report, HP Laboratories Palo Alto, 2006.

[23] T. Y. Yehh and Y.N. Patt. Two-level adaptive training branch prediction. In *The 24th annual International Symposium on Microarchitecture*, pages 51–61, 1991.

Appendix A

Methodology

For our simulations, we modify the SimpleScalar tool set [3] to include the conventional perceptron and O-GEHL branch predictor and our proposed optimizations. SimpleScalar is a cycle accurate superscalar processor simulator. We also use Simpoint [20] to identify representative 500 million instructions regions of the benchmarks for simulations. We use a subset of SPEC2K-INT benchmarks for our simulations with reference inputs. For our simulations, we assume the processor has a 2GHz frequency.

Table A.1 reports the baseline microarchitectural properties of the processor used in the simulations.

Table A.1: Processor Microarchitectural Configurations

Fetch/Decode/Commit	6 & 8
BTB	512 entries
L1 I-Cache	32 KB, 32B blk, 2 way
L1 D-Cache	32 KB, 32B blk, 4 way
L2 Unified-Cache	512 KB, 64B blk, 2 way
L2 Hit Latency	6
L2 Miss Latency	100
Predictor Budget	64Kbits

For predictor energy and timing reports, we use Synopsys Design Compiler synthesis tool assuming the 0.18 um technology. We use the high effort optimization option of the Design Compiler, and optimize the circuit for delay. For tables access

latencies and power dissipations we use CACTI [22].

Tables A.2 and A.3 report the configurations used for the perceptron and O-GEHL branch predictors during simualtions, respectively.

Table A.2: Perceptron Budget / Configuration

Budget	Table Entries	No. Weights	Weight Width (bits)
64Kbytes	1024	64	8
32Kbytes	512	64	8
16Kbytes	348	47	8
8Kbytes	282	28	8

Table A.3: O-GEHL Configuration

Budget	64 Kbits
No. Tables	8
Tables[0-1] Counters Width	5 bits
Tables[2-7] Counters Width	4 bits
Table[1] No. Entries	1024
Tables[0, 2-7] No. Entries	2048
Short History Lengths	0, 3, 5, 8, 12, 19, 31, 49
Long History Lengths	0, 3, 79, 8, 125, 19, 200, 49

Partial Copyright License

Title of Thesis:

Computational and Storage Based Power and Performance Optimizations for Highly Accurate Branch Predictors Relying on Neural Networks

Author: _____

Kaveh Aasaraai

Signed: August 3, 2007