ABSTRACT

Title of Document: A STUDY ON THE NEURAL-BASED PERCEPTRON BRANCH PREDICTOR AND ITS BEHAVIOR

Priyadarshini Rajakumar, M.S, 2006

Directed By: Associate Professor, Dr. Manoj Franklin, Department of Electrical and Computer Engineering

Branch predictors are very critical in modern superscalar processors and are responsible for achieving high performance. As the depth of pipeline and instruction issue rate of high-performance superscalar processors increase, a branch predictor with high accuracy becomes indispensable. It has been speculated that by 2010 branch prediction will become the most limiting factor in the performance of a processor, than the memory system. Branch mispredictions have heavy penalty, causing flushing of the pipeline and re-fetching of instructions from the correct location.

In recent times, neural based branch predictors, like perceptron predictor, are found to have an edge over other popular two-level branch predictors. Branch predictors based on neural learning are the most accurate predictors in the literature as they have sophisticated learning ability to make predictions based on previous outcomes and predictions. However, they are expensive to implement. But perceptron based branch predictors are simple and are easy to implement with less hardware resources. One major advantage of perceptron predictors over the two-level schemes is that we can have longer global or local history length, and

consequently the perceptron predictor is robust to aliasing, resulting in better prediction accuracy.

In this thesis, the behavior and the intricacies of the perceptron predictor are extensively studied. The perceptron predictor has outperformed the classic Gshare predictor with lesser hardware resource. For a memory size of 64KB, the perceptron branch predictor has prediction accuracy about 2-10% higher than that of Gshare. The advantage of having longer history lengths was exploited to determine the performance and the IPC values for the perceptron predictor and showed commendable results. Also, varying the training parameter and the number of perceptrons for prediction helped in analyzing the behavior of the perceptron predictor under different environments.

A STUDY ON THE NEURAL-BASED PERCEPTRON BRANCH PREDICTOR AND ITS BEHAVIOR

By

PRIYADARSHINI RAJAKUMAR

Thesis submitted to the Faculty of the Graduate School of the
University of Maryland, College Park, in partial fulfillment
of the requirements for the degree of
MASTERS OF SCIENCE
2006

Advisory Committee:
Professor Dr. Manoj Franklin, Chair
Professor Dr. Yavuz Oruc
Professor Dr. Charles B. Silio

© Copyright by
PRIYADARSHINI RAJAKUMAR
2006

Dedication

I dedicate this work to God Almighty, for His providence, Strength and His blessings in helping me complete this work. All Praise be to Jesus Christ alone, and to His glory I dedicate this work.

Acknowledgements

I am grateful to my advisor, Dr. Manoj Franklin, for his guidance and encouragement during my study here at the University of Maryland. He was always supportive and his suggestions towards the successful completion of this work are invaluable. This work would not have been possible without him.

Further, I would like to express my gratitude towards Dr. Charles B. Silio and Dr. Yavuz Oruc for agreeing to be on my committee and for their suggestions.

I would like to thank my father and my mother, my beloved brother and my sisters for all their support and prayers that kept me going and helped me finish my work.

I would also like to thank my Uncle and Aunt who have been such a source of encouragement and support to me. I am always indebted to them for their help and prayers.

I thank all my friends who have been there for me in time of need and for their suggestions and help.

TABLE OF CONTENTS

LIST OF TABLES vi

LIST OF FIGURES vii

CHAPTER 1: INTRODUCTION 1

1.1. Importance of branch predictors 1

1.2. Limitations of Two-level Techniques using Table-based Predictors 2

1.3. The Neural Branch Predictor Scheme using Perceptron 4

CHAPTER 2: BACKGROUND 6

2.1 Basics of branch prediction 6

2.2. Limitations of Table-based Branch Predictors 9

2.3. Perceptron-based Branch Predictors 10

CHAPTER 3: DESIGN ASPECTS AND HARDWARE SUPPORT 18

3.1 Tuning parameters 18

3.2. Hardware cost 20

3.3Methodology and implementation 21

CHAPTER 4: EXPERIMENTAL RESULTS 25

4.1. Prediction Accuracies of Gshare Vs Perceptron Predictor 25

4.2. IPC Vs Memory size of Perceptron Predictor 28

4.3. Prediction Accuracy Vs Number of Perceptrons 29

4.4. History Length Vs Prediction Accuracy 31

4.5. Training Threshold Vs Prediction Accuracy 34

CHAPTER 5: SUMMARY AND CONCLUSION 36

5.1. Summary 36

5.2. Conclusion 36

APPENDICES

APPENDIX A 38

APPENDIX B 39

APPENDIC C 41

APPENDIX D 42

Bibliography 43

LIST OF TABLES

1.Table.4.1. Prediction accuracy for Gshare Vs Perceptron predictor for go 25

2.Table.4.2. Prediction accuracy for Gshare Vs Perceptron predictor for cc1 26

LIST of Figures

1. Fig 2.1. Branch Prediction Unit Architecture 6
2. Fig. 2.2. Gshare Branch predictor 8
3. Fig .2.3. Perceptron Model 11
4. Fig. 2.4. Perceptron weighted Matrix model. 12
5. Fig 2.5 Block diagram of Perceptron branch predictor 15
6. Fig. 2.6. Linear separability property 17
7. Fig. 4.1. Memory size Vs prediction accuracy with different History length for go benchmark 26
8. Fig.4.2. Memory size Vs prediction accuracy with different History length for cc1 benchmark 27
9. Fig 4.3 Prediction Accuracy of Gshare Vs Perceptron predictor 27
10. Fig.4.4. Memory size Vs IPC for go 28
11. Fig.4.5 Memory size Vs IPC for cc1 29
12. Fig.4.6. Prediction Rate Vs N for go 30
13. Fig.4.7. Prediction Rate Vs N for cc1 30
14. Fig.4.8. Prediction Rate Vs N for treeadd 31
15. Fig.4.9. Prediction accuracy Vs History length for go 32
16. Fig.4.10. Prediction accuracy Vs History length for cc1 32
17. Fig.4.11. Prediction accuracy Vs History length for treeadd 33
18. Fig.4.12. Theta Vs prediction accuracy for different N and constant h=15 for go 34

19. Fig.4.13. Theta Vs prediction accuracy for different N and constant h=15 for cc1 35

CHAPTER1. INTRODUCTION

1.1. Importance of Branch prediction

Instruction-level parallelism is made possible with speculation involving predicting the values even before the actual values are made available. We require accurate prediction schemes to enhance the performance of speculating these data. Branch prediction is an indispensable component of modern microarchitectures. When a branch is encountered in a processor pipeline, stalling the remaining instructions would degrade the performance of the processor. Instead, predicting the branch outcome and speculative fetching from the predicted address would allow the processor to execute instructions along the direction of the predicted path [8].

Speculating the outcome of a branch instruction allows the processor to continue fetching instructions from the predicted target without the knowledge of whether that is the correct location. If the prediction is correct, the throughput of the system is not affected and execution of the remaining instructions continues without interruption. If the prediction is wrong, the incorrect instructions have to be flushed from the pipeline, and instruction fetching should continue from the correct address.

Branch misprediction is one of the most important causes of performance degradation as the number of pipeline stages becomes large. Branch predictors must be improvised in order to prevent the penalties of mispredicting branches, more specifically the conditional branches, and enjoy the benefits of predicting a branch correctly.

In out-of-order processors, mispredictions divide the instruction window into sequentially executed segments, limiting the ILP. As the instruction window increases in size, the limitation on performance increases.

It is difficult to design a branch prediction technique that performs with higher accuracy for all kinds of inputs. With a fixed hardware budget as a constraint, it is important to design a predictor that provides good accuracy.

1.2. Limitations of Two-level Techniques using Table-based Predictors

Two-level predictors use saturating counters that are indexed using the address of the branch and use the MSB bits of the counter to make the prediction. Whenever the branch is taken, the counter is incremented; whenever it is not taken, the counter is decremented.

It has been found that branch outcomes are highly correlated to other branches in the program. To incorporate global information, two-level predictors have been proposed [2]. Given the outcome of the previous 'n' branches, a Pattern History Table (PHT) is accessed. Inside this Pattern history table is a two-bit saturating counter that decrements or increments based on whether the branch was predicted correctly or incorrectly.

These are called two-level table-based branch predictors. In the first level we have a Branch History Register (BHR) that stores information of past executed branches. If it stores information of all recently executed branches, it is called a Global history register and if it stores information pertaining to recent executions of the same static branch, it is called a Local history register. This information is used to index the second level, which is the PHT having a saturating counter.

There are different techniques to design a branch predictor, depending upon the hardware budget and the accuracy required. Based on the way by which the PHT is indexed using the BHR, we have different kinds of table-based branch predictors. The common Bi-Modal predictor has 2 bits for the saturating counter whose outcome is used to predict the branches. The GShare (Global Index Sharing) [1] proposed by McFarling has both the address and the BHR bits Ex-OR'ed to index into the PHT. A variation of the GShare is PShare (Per Address Index sharing). Yeh and Matt [2] came up with multiple layers of BHR and PHT to account for both global and local history of branches that has information regarding the branch history. But this had high hardware cost and was not feasible. There are also combinational predictors having two of the above-mentioned predictors combined to predict the branches. For better prediction accuracies, a compromise on hardware requirement has to be made for most of the branch predictors.

1.2.1. Demerits of Table based predictors

The table-based branch predicting schemes have some basic limitations, one of them being Aliasing, wherein more than one location on the PHT is indexed causing interference and inaccurate prediction. Another disadvantage is that longer history length could not be used for better prediction, as the hardware budget is limited. So, there is not much scope for very good prediction accuracy beyond a level.

1.3. The Neural Branch Predictor Scheme using Perceptron

Recently, a perceptron- based branch predictor was proposed by Jimenez and Lin [3] [4]. The perceptron replaces the finite state machine used for state transition in table-based predictors. The Perceptron predictor uses percepton learning to predict the directions of

conditional branches. It is a correlating predictor that makes a prediction for the current branch based on the history pattern observed for the previous branches. A perceptron is a simple learning device that multiplies the input data with weights and sums it up to give a single output.

The perceptron based branch predictor thus combines the set of inputs which is the Global BHR with the weights, which attempt to capture the correlation between the past branch outcomes and the behavior of the branch being predicted, and gives an output that would determine whether a branch is to be taken or not. The advantage of using perceptrons is that we can exploit their ability to make use of longer history length, as the size of the perceptrons linearly scales with the history length, whereas in other schemes the history length is used to hash the PHT and would cause it to increase exponentially (2^n for n bits of BHR).

This thesis makes the following contributions. We have performed a detailed study of perceptron based branch predictors. Although a few studies on perceptron branch predictors have been done before, these studies have focused on a subset of the factors affecting the performance of perceptron predictors. Our study looks at all the relevant parameters, and presents all of the results in one place. For comparison purposes, we use a Gshare predictor also. Perceptron based predictors could achieve better prediction accuracies with smaller history length than that of Gshare; with increase in the history length the perceptron predictor performed even better. But with Gshare, just as in most table-based predictors, increase in history length is not feasible. Yet another finding was

that with an increase in the number of perceptrons, the prediction was better, as more weights were used in order to predict an outcome.

The rest of this thesis is organized as follows. Chapter 2 describes in detail the background and related area of research, and presents a detailed look at the perceptron branch predictor. Chapter 3 describes various issues and techniques for improving the perceptron predictor. Chapter 4 discusses the experimental framework and the results from simulation experiments. Chapter 5 summarizes and draws conclusions.

CHAPTER 2: BACKGROUND

2.1. Basics of Branch Prediction

2.1.1. How Branch Predictors Function?

Branch predictors combine the information from the branch history register (BHR) ─ that stores the previous outcomes of branches ─ and the address of the fetched branch instruction to predict the branch outcome. This is done at the fetch stage of the pipeline. As described in Section 1.2, there are various schemes for branch prediction, but the basic underlying principle remains the same. Figure 2.1 describes the general architecture of branch predictors [6].

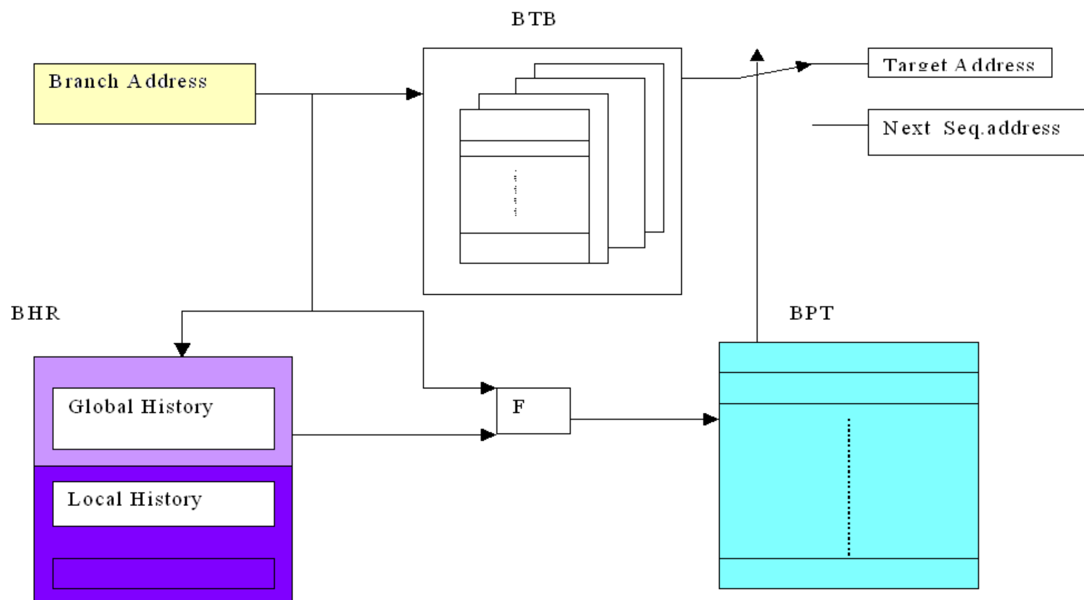

Fig.2.1. Branch Prediction Unit Architecture (adapted from [6])

The BPU consists of mainly two logical parts, the BTB and the predictor. The BTB is the buffer where the CPU stores the target addresses of the previous branches if the branch is to be taken and we need to fetch the next instruction from that target address. The predictor

is that part of the BPU that makes the prediction on the outcome of the branch under question. There are different parts in a predictor: Branch History Registers (BHR) like the global history register or local history registers — storing the outcome of previously executed branches — and branch prediction tables (BPT) having an asynchronous sequential machine, more generally a saturating counter.

This work limits itself to the branch predictor part involving prediction of the outcome of the predictor and does not delve into the details of the Branch target buffer. So, different branch prediction schemes vary from each other in the way the BPT is accessed using the BHR and the branch address. Some of the branch prediction schemes are discussed briefly below, but emphasis is laid on Gshare, which has been found to be very accurate.

2.1.1. Common Branch Predictors: Gshare

A very simple branch predictor is the bimodal branch predictor, which hashes the BPT consisting of a saturating 2 -bit counter. The state of the counters is stored in this counter table that records all the branches' history. Each branch will then map to a unique counter. The branch history table is indexed by some bits of the branch address.

The correlated branch predictors absolves the mapping collision problem faced by Bimodal predictors by using two branch history tables, one for keeping the recent branch history records and the other one for keeping the state of branches in each entry contained 2-bit counter. So, it takes the advantage of the relationship between different branch instructions that is certain repetitive branch pattern of several consecutive branches. The local, global and global selection, are all correlated predictors, making use of history information from

either local or global or both tables. They vary from each other, depending upon whether they are globally adaptive or per-address adaptive and the way the PHT is indexed.

The Gshare predictor is one of the best schemes among the correlating branch prediction schemes and has been shown to have better accuracy than other correlating predictors. In this thesis work, the performance of the perceptron predictor is compared against Gshare, and thus it calls for an understanding as to how Gshare works. All of these predicitng schemes are called table-based branch predictors, as they use a history table to predict branches.

The Gshare predictor proposed by McFarling [1] hashes the PHT by Ex-OR'ing the branch address bits and the bits from the history table (BHT). This makes the index more explicit and hence mapping to the same location on the PHT is reduced.

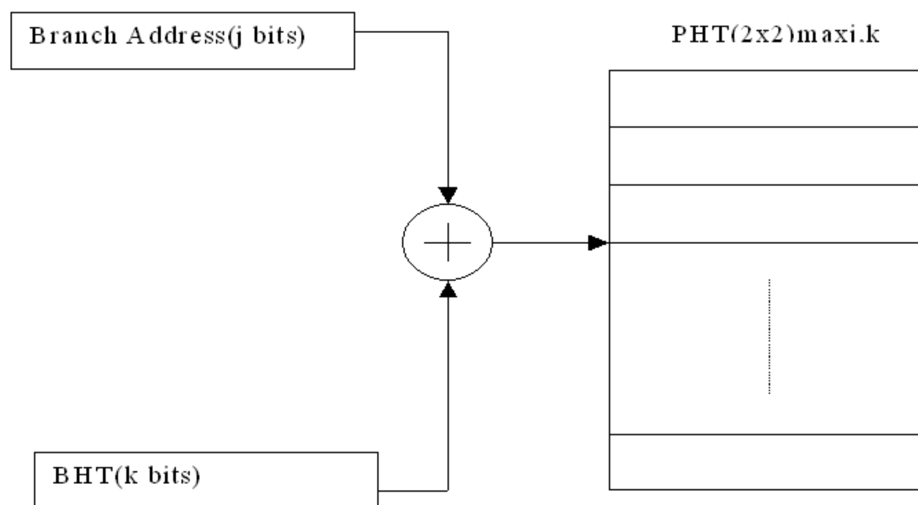

Fig. 2.2. Gshare Branch Predictor (adapted from [1]).

2.2. Limitations of Table-based Branch Predictors

These predictors also come up with other limitations cited below:

1. Aliasing: The PHT is limited in size, resulting in many branches mapping to the same location in the PHT. This causes several branches to address and change values in the same counter. This is called aliasing or interference. Interference can be positive (outcomes of other branches helps the current prediction), negative (outcome of other branches misguides the current prediction), or neutral [9].

2. Period to warm up: With larger more accurate predictors it takes some time for the PHT to be filled with values, allowing for accurate predictions to be made.

This is a problem in systems with context switches. If the PHT is flushed, then the PHT needs to be refilled, restarting the learning process. If the PHT is not flushed the branch characteristics of the program executing previously may degrade performance and require unlearning to get back to the accuracy prior to the switch.

3. History length: With increase in history length, some of the branch predictors like Gshare have a detrimental effect as it runs out of bits since *gshare* requires resources exponential in the number of history bits. So the scope of improvement with a little upgrade in hardware budget is absent.

The perceptron based branch prediction has been found to have solved these limitations faced by table-based predictors and has better prediction accuracy.

2. 3. Perceptron-based Branch Predictors

2. 3. 1. Basic Working of a Perceptron

The perceptron was introduced in 1962 [10] as a way to study brain function [3] [4]. The Perceptron predictor is a learning hardware structure that predicts the directions of conditional branches. It is a correlating predictor and makes a prediction for the current branch based on the history pattern observed for the previous branches.

A perceptron is a learning device [10] that takes a set of input values and combines them with a set of weights (which are learned through training) to produce an output value. In our predictor, each weight represents the degree of correlation between the behavior of a past branch and the behavior of the branch being predicted. Positive weights represent positive correlation, and negative weights represent negative correlation. To make a prediction, each weight contributes in proportion to its magnitude in the following manner. If its corresponding branch was taken, the weight is added; otherwise the weight is subtracted. If the resulting sum is positive, the branch is predicted to be *taken*; otherwise it is predicted to be *not taken*. The perceptrons are trained by an algorithm that increments a weight when the branch outcome agrees with the weight's correlation and decrements the weight otherwise. This way we ensure the assertiveness of branch outcomes that are positive.

The perceptron behaves like a neuron (brain cell), which accepts various signals it receives through dendrites and if the combined signal strength (weights) is greater than the threshold then it outputs the combined signal. Now a perceptron accepts all the inputs and assigns them weights depending on the correlation between previous outcomes of that branch.

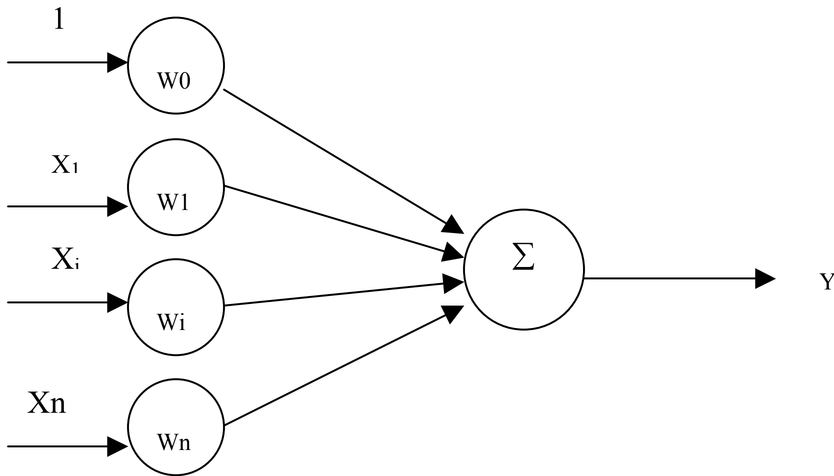

Fig.2.3. The Perceptron Model (adapted from [3])

Fig. 2.3 depicts the Perceptron model [3], where input vectors $X_{1\ldots}$ X_n are multiplied by their corresponding weights $W_{1\ldots}$ Wn. This product is summed up to give the output Y. X_0 is always 1 to have a bias input. The perceptron is made to learn a Boolean function Y = f($X_{1,} \ldots\ldots X_n$), where X_i are the bits of the global history shift register for n inputs and the weights $W_{1} \ldots\ldots$ Wn give the corresponding correlation weights and are signed integers. The input vector bits are either 1 for *taken* and –1 for *not taken,* and the output Y is 1 for predict *taken* and –1 for predict *not taken.*

The output function Y is given by [3]

$$Y = W_0 + \sum X_i Wi$$

A perceptron predictor can be represented by an N × (h+1) matrix having an entry of W weights, where N is the number of perceptrons and h is the history length. Each row of the matrix is a (h+1) length weights vector. Each weights vector stores the weights of one perceptron, and is updated as the perceptron learns from training. In the weights vector W

[0…h], the first weight W_0 is known as the bias weight and is set to 1. Weights are typically 8 bits. So, the first column W contains the bias weights of each weights vector. The global history shift register is represented by a single row matrix H [1...h], which contains the outcome of the previous h branches represented by a 1 or –1 for taken and not taken, respectively.

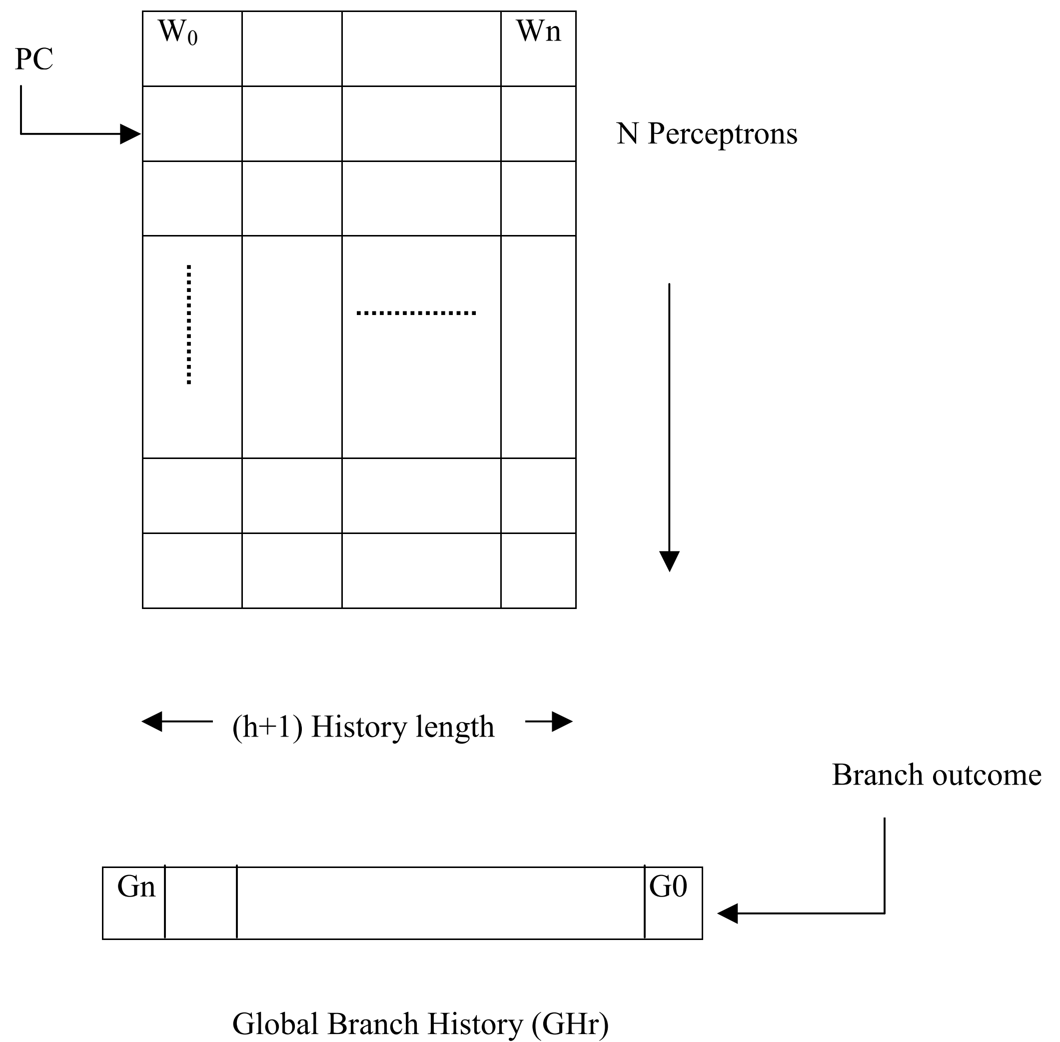

Fig.2.4. Perceptron Weighted Matrix Model.

2.3.2. Training the Perceptrons

The PC of the branch instruction is hashed to a particular row of the Weights matrix. This row acts as a perceptron that is responsible for the prediction of that particular branch. A single perceptron can be responsible for the predicting of multiple branch instructions. The outcome of the branch is based on the weight values in that row and on the outcome of the most recent branches, which are stored in the global history register G.

Once the outcome is known, the perceptron that was used for the prediction is trained, depending on whether the prediction was correct, and also based on the pattern history stored in the global history register G. This is done by updating the weights in the corresponding row of the W matrix used for predicting the branch.

Let θ be the threshold to train the perceptron [3], which determines the extent to which the perceptron is trained, and t be –1 if not taken and 1 if the branch is taken. Then the training algorithm is given by [3]:

```
If sign (yout) ≠ t or |yout| ≤  θ then
        For i = 0 to n do
        Wi: = Wi + tXi
        End
End if
```

It has been suggested [3] that the best value for the training threshold θ is

$$\theta = 1.93 \times h + 14$$, where h is the history length.

This algorithm increments or decrements the weights, depending upon whether a branch is taken or not, as essentially t and X_i are either 1 or –1. When outcomes are predicted

correctly, we have a positive correlation and the weights become larger. Likewise, when the outcomes are predicted incorrectly, the negative correlation decreases the weights. So the correlation ─ both negative and positive ─ impacts the weights and in turn influences the prediction [7].

2. 3. 3. The Perceptron Predictor

Now that the structure and basic working principle of the perceptron is known, we incorporate this perceptron predictor in our hardware. The block diagram in Fig 2.4 shows the perceptron predictor as a whole. The diagram shows that instead of 2-bit saturating counters we have a table on N perceptrons and also a training logic. The hardware resource allocated depends on the number of perceptrons, and the history length for the weights used.

During the fetch stage,

i) The PC of the branch address is made to index into the perceptron table where index *i* belongs to 0…N-1 in the table of perceptrons. So, essentially one row of the weight matrix is selected.

ii) The row of the perceptron table containing the weights is selected and is stored as a vector register P.

iii) The output Y is the dot product of the entries in the global history register and the weights from the perceptron table.

iv) The branch is predicted taken when the output Y is positive and is predicted as not taken when the output is negative.

Execution stage:

i) Once the outcome of the prediction is known, the training algorithm uses this predicted outcome and the actual outcome of the branch to update the weights. So if the prediction was correct, the weights are incremented; else it is decremented

ii) This information is then updated in the *ith* entry of the perceptron table

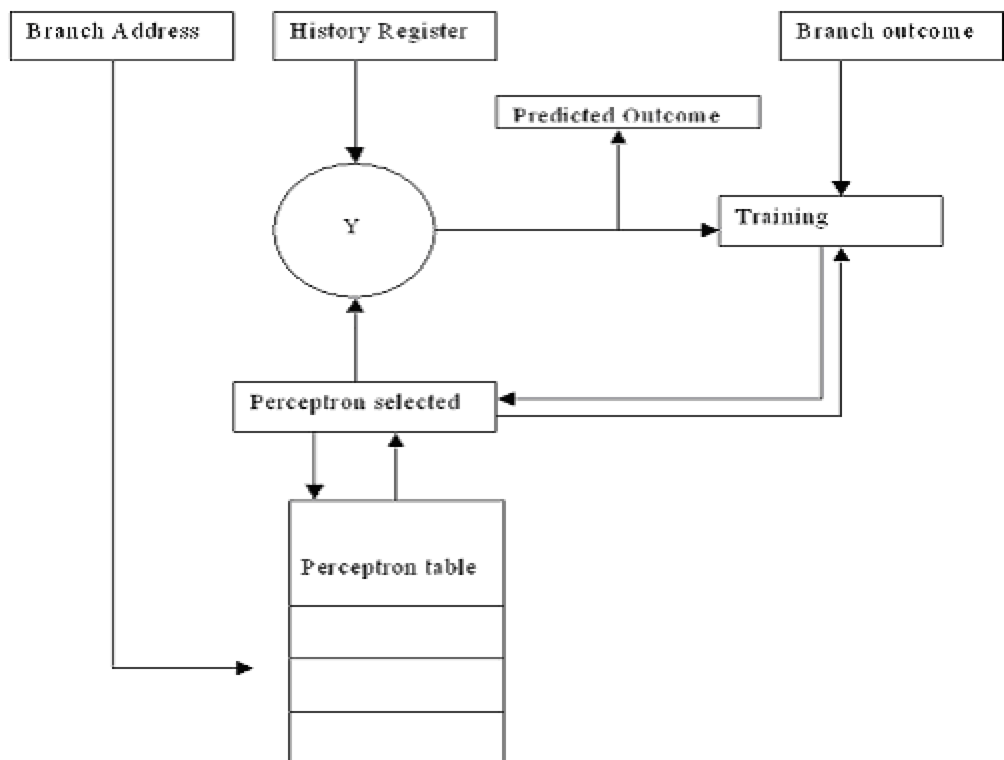

Fig 2.5 Block diagram of Perceptron Predictor (adapted from [3]).

Every time the branch predictor encounters a conditional branch instruction, it computes the product and makes a prediction. Based on the outcome of the prediction,

instructions are fetched along the direction (path) of the flow of the program and the branch command is moved to the next pipe stage. After the branch condition calculation, a check is made to see if the prediction was correct. If not, the subsequent instructions are flushed from the pipeline, and the PC is updated to continue fetching from the correct location. In any case, the perceptron is trained, i.e., the predictor is updated according to the history register at the time of prediction.

Linear Separability:

Perceptrons can only learn a limited class of functions. The output formula for perceptron is given by:

$$O = g\,(W.I)$$

Given the sigmoid function noted previously, the perceptron outputs something close to 1 if the inner product of the weight vector and input vector is greater than zero; otherwise it outputs something close to 0.

So the perceptron is distinguishing inputs based on where they fall with respect to a hyperplane in input space (whose coefficients are the weights). The perceptron will learn correctly if there exists a hyperplane that divides the inputs correctly, i.e., if the function is linearly separable (in each output for perceptron networks).

The perceptron can only learn two types of inputs, i.e., it is bipolar. It responds to either a true (a 1) or a false (–1) condition and outputs either 1 or –1. Figure 2.6 [3] illustrates the linear separability exhibited by perceptrons.

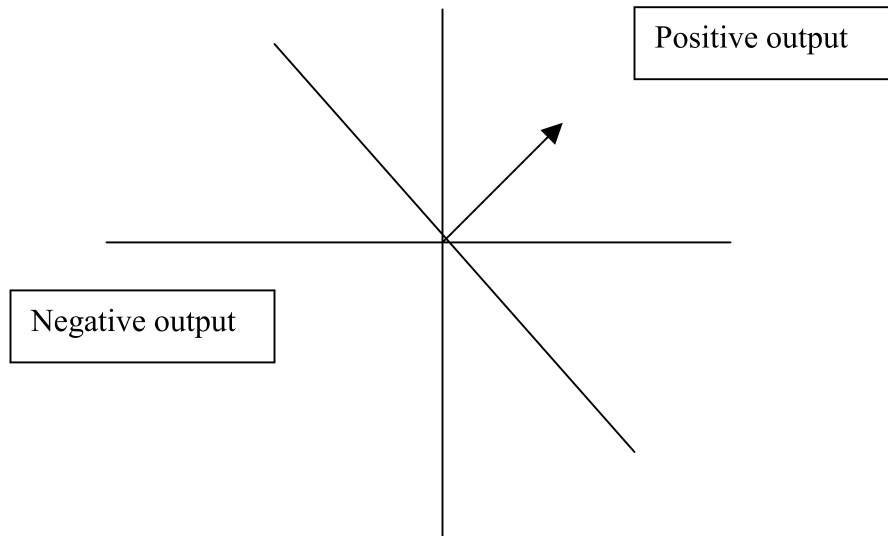

Fig 2.6. Linear Separability Property of Perceptrons (adapted from [4]).

If the set of all possible inputs to a perceptron can be imagined as an n-dimension space, then for the equation

$$Y = W0 + \sum_{i=1::n} Wi\ Xi$$

there exists a solution that is a hyperplane having 2 sets of inputs dividing the plane what the perceptron distinguishes. Therefore, the perceptron can only be given bipolar inputs.

Recent developments on the proposed perceptron branch predictors by Jimenez and Lin have been suggested to be a promising technology for future microprocessors [18]. It has been used in studies of hybrid predictors [3]

It has also been suggested by another study that perceptron predictors can also be implemented using techniques from high-speed arithmetic [17], but the latency of the predictor is more than 4 cycles with an aggressive clock rate. Also the perceptron predictor is found to achieve superior accuracy and low latency by choosing the neural weights based on the path taken to reach a branch rather than the branch address itself [4].

CHAPTER 3: DESIGN ASPECTS AND HARDWARE SUPPORT

For a fixed hardware budget for the predictor, there are certain parameters that can be changed in order to get good prediction accuracies. The parameters that are tuned are the History length and weights, the Number of perceptrons, and the training threshold. A brief review of our design: we have a Matrix W of N× (h+1) perceptrons, where N is the Number of perceptrons and h is the History length. This matrix contains the weights of the perceptrons (Wi…Wn), where weights represent the correlation between the predicted outcome and the actual outcome of the branch. Typically the weights are of 8 bits.

3.1. Tuning Parameters

3.1.1. History Length

The longer the history length, better is the prediction accuracy [13]. But very long history length reduces the number of entries in the weight matrix and therefore multiple branches are indexed into the same location. In a Gshare predictor, increase in History length causes detrimental effects, because the PHT size increases exponentially with increase in history length (2^n entries in the PHT for a history length of n). But in a perceptron predictor, an increase in the history length to an optimum number gives very good prediction accuracy compared to Gshare. When we increase the history length, we increase the weights for that perceptron. So the decision it makes is more accurate, as it has more information to make that decision. On the other hand, when we have a very long history length, the number of entries in the table is reduced, causing aliasing.

In this thesis work, we have varied the history length for the perceptron until the total hardware cost is 4KB, i.e., h equaling 30, whereas in Gshare that corresponds to h equaling

about 12-14. So clearly for the same hardware cost, we can have a longer history length than that of two-level predictors like Gshare. And with longer history length, the perceptron predictor outperforms the Gshare predictor.

3.1.2. Perceptron Weights

The weights used for the perceptrons are signed integers. Typically, they are represented with 8 bits each. Though we require just 1 bit for the sign, extra bits are required to represent the threshold θ that influences the weights and hence we need an additional [$\log_2 \theta$] bits. So the total number of bits assigned is the sum of the bits required to assign the sign bit (which is 1) plus the bits to represent [$\log_2 \theta$].

3.1.3. Training Threshold (θ)

The training threshold determines the extent to which the perceptron must be trained. From the training algorithm we can see that when the magnitude of the outcome *y* is less than that of the threshold θ, the weights are changed. So the weights depend on the threshold value. In this work, we show that increasing the value of θ up to a certain optimal point increases the prediction accuracy, as the perceptron is trained more. But beyond that point it does not seem to impact the outcome and enters into saturation. The threshold value depends on the history length, as adding more history length would mean adding more perceptron weights and hence the threshold has to be increased to train them. The suggested value for θ given in [3] is $1.93 \times h + 14$.

3.1.4. Perceptron Size

The number of perceptrons also influences the prediction accuracy of the perceptron predictor. With a given a hardware budget, we can increase the number of perceptrons and by keeping the history length to an optimum value, we can have very good prediction accuracy. This is because, we have more perceptrons, and hence all their weights contribute collectively to predict the outcome and therefore it is more accurate. For example, with N x h constant at 64 KB, we increase the number of perceptrons, and reduce the history length to an optimal value. In this work we have used perceptron sizes ranging from 64 to 2048. For the same memory size, the perceptron predictor outperforms the Gshare predictor.

3.2. Hardware Cost

The hardware budget to be allocated for the perceptron predictor depends on the prediction accuracy desired. The hardware cost is distributed between the history length and the number of perceptrons. The perceptron predictor gives better prediction accuracy than that of Gshare for the same hardware budget. For example, for a 4 KB hardware budget, a PHT based predictor can use only a history length of 14 because the PHT increases exponentially with increase in the history length, whereas a version of the perceptron predictor can use a history length of 30. These longer history lengths lead to higher accuracy. To obtain even higher prediction accuracy we can increase the number of perceptrons, which demands more hardware. So it essentially is a trade off between accuracy and memory size. Also, for applications having programs that have fewer instructions or fewer branches, a branch predictor with a smaller history length is

preferable, as the distance between correlating branches is not too large and does not require that many weights for prediction.

3.3. Methodology and Implementation

We use software simulation to evaluate the performance of perceptron predictors. We compare their performance with the widely used Gshare predictor.

3.3.1. Platform

The platform we used for software simulation was the SimpleScalar simulator (www.simplescalar.com) [19]. The simulator has its own branch predictors among which Gshare is also present. A neural networks branch prediction mechanism was implemented over the existing platform. The simulation was done in an out-of-order fashion, and was tested on SPEC 2000 benchmarks (www.spec.org) [21] and on Olden benchmarks [22]. The Alpha binaries of these benchmarks were used.

3.3.2. Algorithms

The code for the perceptron predictor was added onto the existing branch predictor of the SimpleScalar simulator, and hence had to follow some of the conventions and declarations made in the SimpleScalar branch predictor. In order to do this, some structures were written (See Appendix A) to emulate the perceptron behavior. The algorithms for determining the output/outcome of the prediction and for training the perceptron predictor are discussed in this section.

Some of the essential variables used are:

Global History Register:

The History register where every bit that is on represents a taken branch.

Speculated Global History:

History as seen by the perceptron predictor during lookup

Perceptron weights matrix W:

Table of perceptrons containing entries of perceptron, that are represented by its weights.

Algorithm for determining the outcome of prediction, adapted from [3] and included in bpred.c :

The outcome function:

i) The hashed perceptron is found in the weights table and the index into this table is determined:

```
index = NEURAL_HASH(prediction_dir, branch_addr);
```

ii) The weights of the perceptron are used to compute the output

```
for (mask=1,i=0;i<PERCEPTRON_HISTORY;
 i++, mask<<=1,w++) {
     if (spec_global_history & mask)
          output += *w;
     else
          output += -*w;
```

The first entry of the weight is the bias weight (=1) and does not depend on history.

iii) Then the perceptron prediction is made:

```
y = output >= 0;
```

The Training function:

The perceptron is trained after the branch is executed and the outcome is known.

i) Maximum and minimum weights are checked before update:

```
for (mask=1, i=0;i<PERCEPTRON_HISTORY; i++,
   mask<<=1,w++) {
   if (!!(history & mask) == taken) {
         (*w)++;
         if (*w > MAX_WEIGHT) *w = MAX_WEIGHT;
   } else {
         (*w)--;
         if (*w < MIN_WEIGHT) *w = MIN_WEIGHT;
         }
   }
```

ii) Then, the real global history register is updated:

dirpred.bimod is the bimodal predictor and is configured according to the neural predictor.

```
pred->dirpred.bimod->config.neural.glolabl_history <<= 1;
          pred->dirpred.bimod->config.neural.global_history
|= taken;
```

iii) If the branch was mispredicted, the global history is restored with the speculative history .

```
if (u->prediction != taken)
     pred>dirpred.bimod>config.neural.spec_global_
     history
 =pred->dirpred.bimod->config.neural.global_history
;
```

iv) If |y| > THETA, update is not necessary. Else, for each weight and corresponding bit in the history register, increment weights if taken or decrement if not taken:

```
if (!!(history & mask ) == taken) {
     (*w)++;
else      (*w)--;
```

.

With the algorithms described above, the perceptron predictor can be made to predict a branch outcome as well as train the perceptron to adjust its weights according to actual outcome of the prediction. These algorithms were made use of in the existing SimpleScalar branch predictor. Also, the out-of-order simulator had to be modified to suit the neural predictor. Thus, the test environment was arranged to determine the performance of the neural predictor against the two-level predictor Gshare.

CHAPTER 4: EXPERIMENTAL RESULTS

The performance of perceptron predictors was compared against the two-level Gshare predictor on SPEC2000 [19] Olden benchmarks [20]. The results showed that the prediction accuracy of perceptron predictors was higher than that of Gshare predictors. The GShare predictor was tested with a global memory register length varying from 8 to 15, and the memory register length of perceptron predictors varying from 10 to 30 and the number of entries in the perceptrons table varying from 64 to 2048.

4.1. Prediction Accuracies of Gshare Vs Perceptron Predictor

The perceptron predictor could achieve better prediction accuracies even with less memory size. Also, for the same history length, the perceptron predictor outperformed Gshare. Another exciting result is that the perceptron predictor could achieve better prediction accuracies than the Gshare with smaller history length. Tables 4.1 and 4.2 show the performance of Gshare and perceptron predictor for the go and cc1 benchmarks.

Perceptron			Gshare		
History length/N	Pred Accuracy(%)	Memory size(Bytes)	History length	Pred Accuracy(%)	Memory size(Bytes)
10/64	91.11	3840	8	88.2	512
10/128	91.14	7680	9	89.23	1024
15/64	91.89	5760	10	89.69	2048
15/128	92.1	11520	11	90.217	4096
20/64	92.12	7680	12	90.317	8192
20/128	92.44	15360	13	90.89	16384
25/64	92.46	9600	14	90.94	32768
25/128	92.69	19200	15	90.86	65536
30/64	92.46	11520			
30/128	92.71	23040			

Table 4.1. Prediction Accuracy for Gshare Vs Perceptron Predictor for go

Perceptron			Gshare		
History length/N	Pred Accuracy(%)	Memorysize (Bytes)	History length	Pred Accuracy(%)	Memory size(Bytes)
10/64	85.5	3840	8	83.02	512
10/128	86.3	7680	9	83.36	1024
15/64	85.8	5760	10	84.49	2048
15/128	86.64	11520	11	85.03	4096
20/64	86.2	7680	12	84.06	8192
20/128	86.75	15360	13	85.4	16384
25/64	86.4	9600	14	85.88	32768
25/128	86.9	19200	15	86.09	65536
30/64	86.63	11520			
30/128	87.6	23040			

Table 4.2. Prediction Accuracy for Gshare Vs Perceptron Predictor for cc1

The complete data for the above table for both go and cc1 benchmarks are shown in the Fig 4.1 and 4.2. It shows that the prediction accuracy for perceptron predictor is much higher than that of Gshare for smaller memory size.

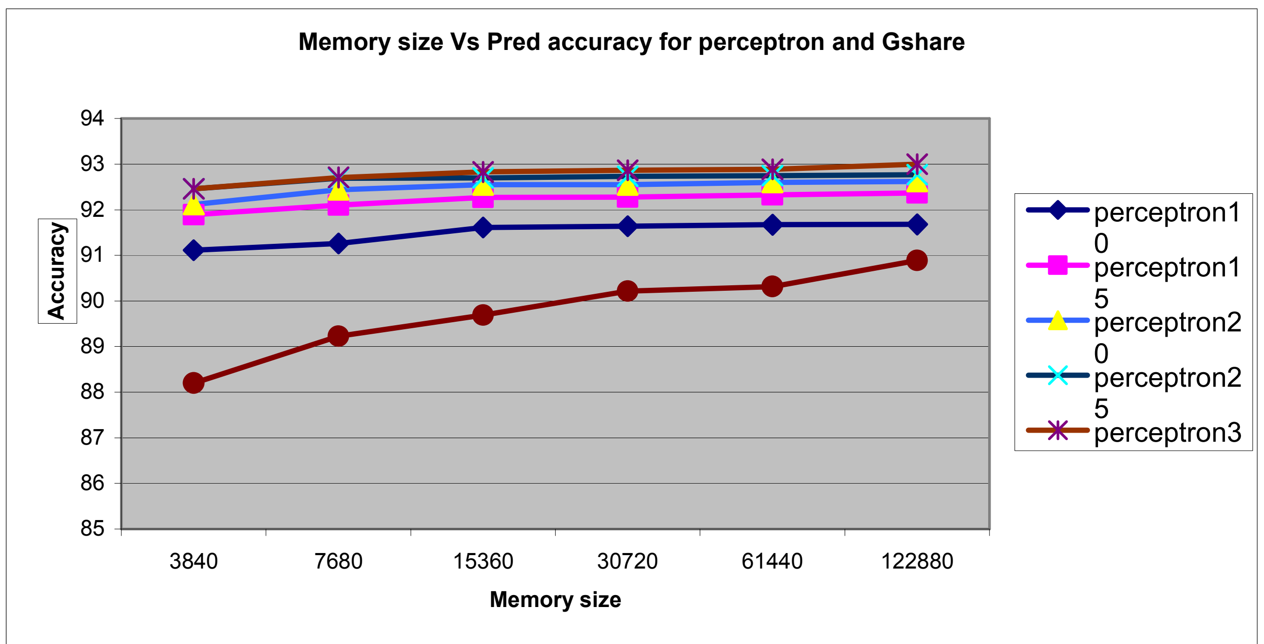

Fig. 4.1. Memory Sizes Vs Prediction Accuracy with Different History Lengths for go

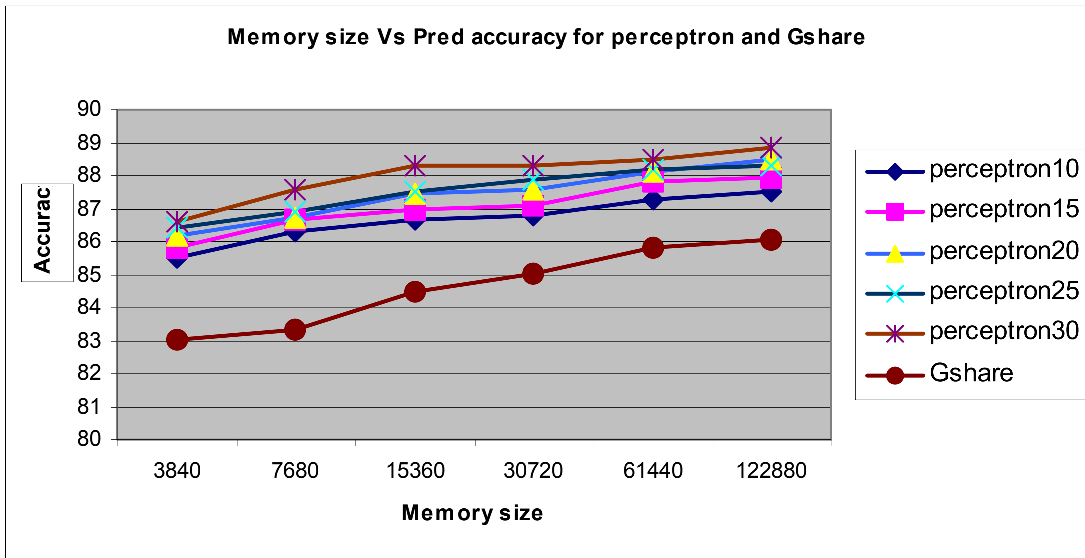

Fig.4.2. Memory Size Vs Prediction Accuracy with Different History Lengths for cc1

The graph below shows the prediction accuracy for Gshare Vs perceptron predictor for different benchmarks. It shows that the perceptron predictor has better prediction accuracy for the same history length of 15 and for a memory size of 64KB.

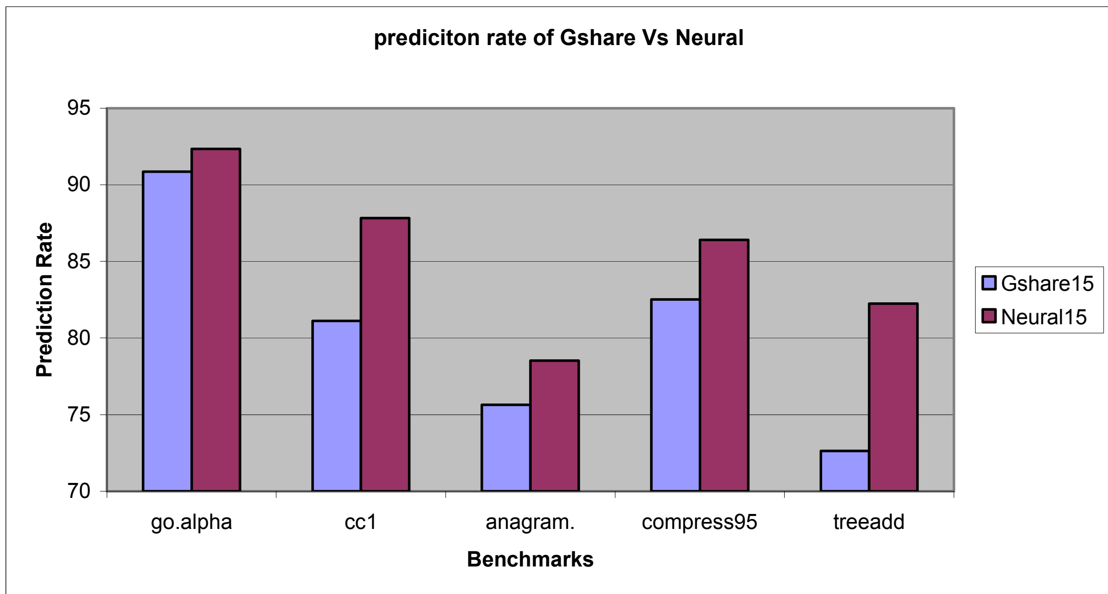

Fig.4.3. Prediction Accuracy of Gshare Vs Neural

4.2. IPC Vs Memory size of Perceptron Predictor

IPC is an excellent metric to measure the performance of a system. Again, the perceptron predictor showed better performance than that of Gshare. The perceptron predictor deals with more instructions per cycle and has a higher IPC than Gshare . Increasing the history length of the perceptron predictor does not seem to affect the IPC much. On the other hand, for the Gshare predictor, upon increasing the memory size (history length), the IPC seems to increase but is still less than that of perceptron predictors.

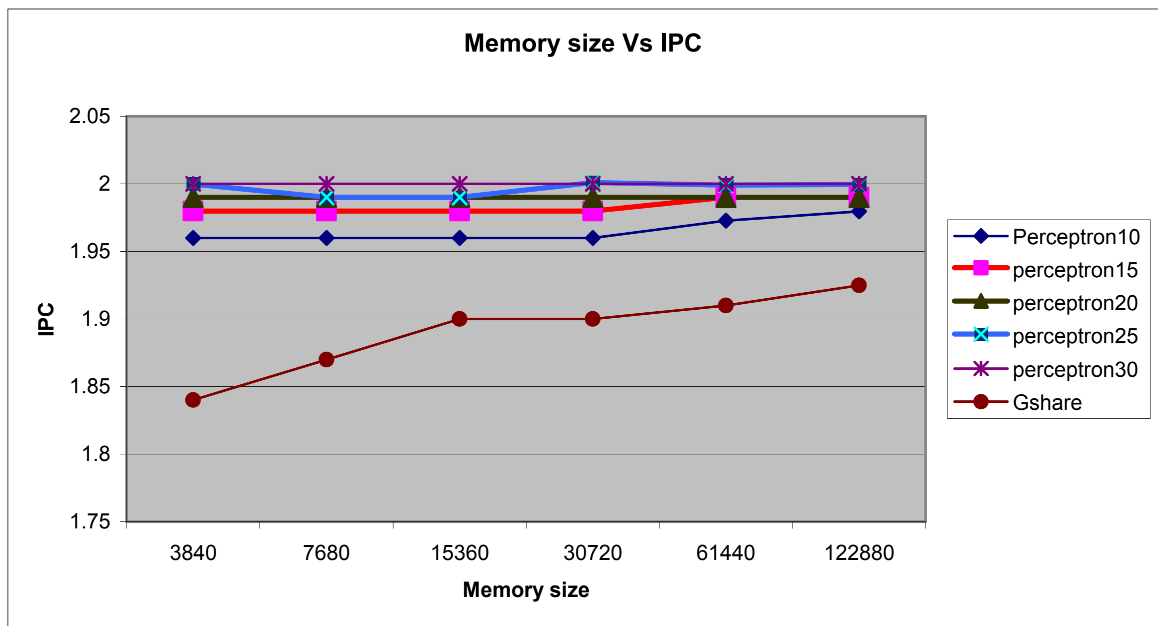

Fig. 4.4. Memory Size Vs IPC for go

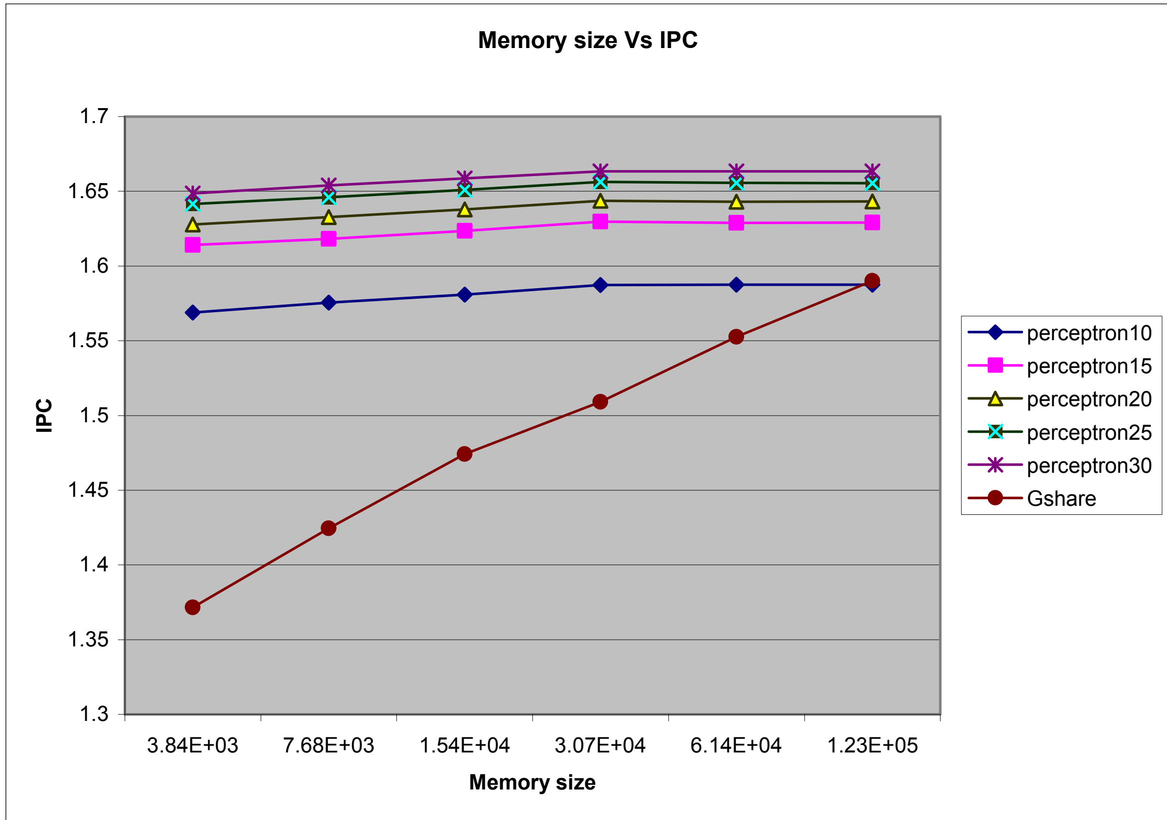

Fig. 4.5 Memory Size Vs IPC for cc1

Intuitively, since the computation required for prediction and update algorithm for the perceptron predictor is more, the gain in accuracy is counterbalanced by the increase in the computation time and [14] hence the overall IPC does not increase so much and almost remains flat for the most part.

4.3. Prediction Accuracy Vs Number of Perceptrons

Given a hardware budget, we have seen that the perceptron predictor performs better with lesser memory size than Gshare. With more available hardware, increasing the number of perceptrons gives very good prediction accuracies. This is because the predictor is able to decide more accurately as more weights are available to base the decision. So the

correlation factor is more, and this builds confidence [15] for the predictor to predict the outcome [12].

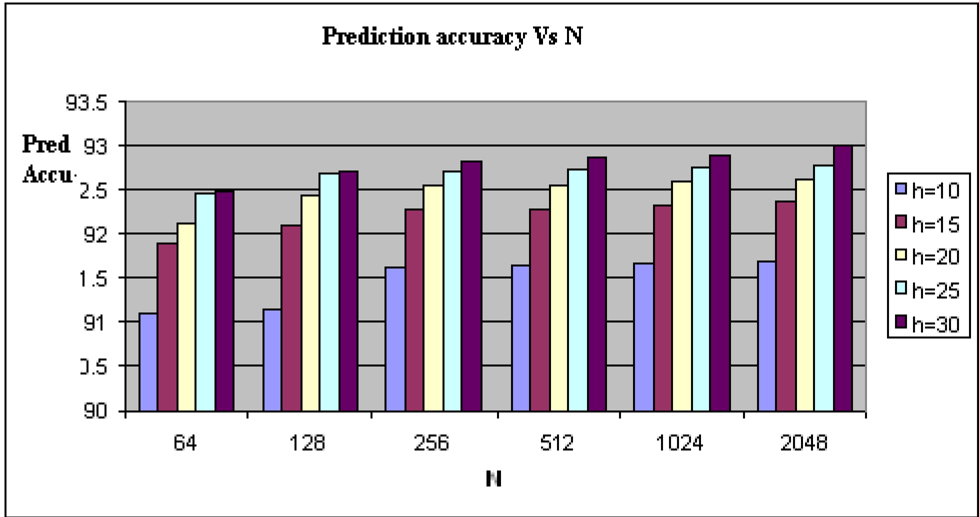

Fig.4.6. Prediction Accuracy Vs N for go

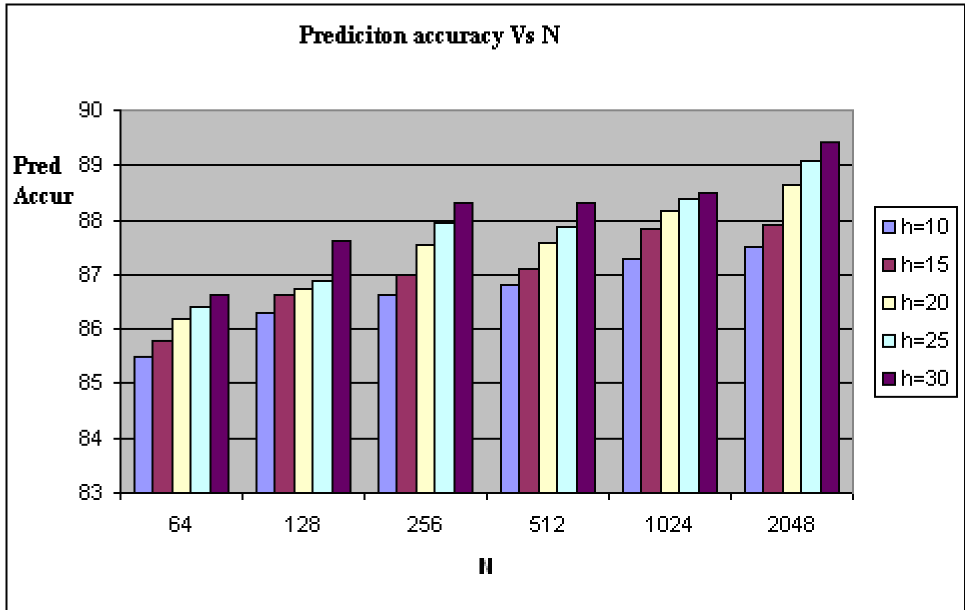

Fig.4.7. Prediction Accuracy Vs N for cc1

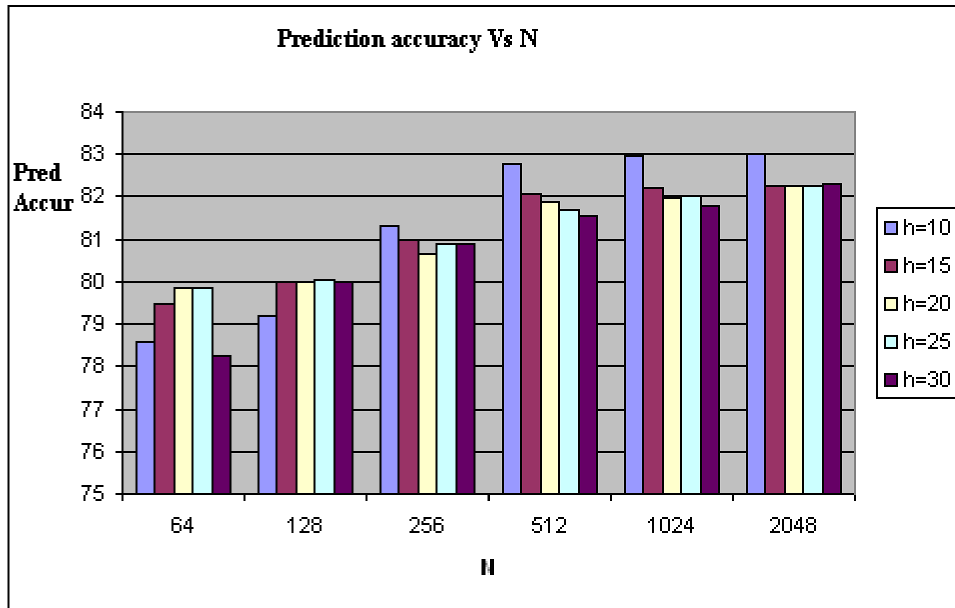

Fig.4.8. Prediction Accuracy Vs N for treeadd

There is about 5% increase in the prediction accuracy when we increase the number of perceptrons from 64 to 2048. This is not too much of an increase in hardware resource when compared to the Gshare predictions, where to improve the prediction accuracy by 3% the hardware resource increases exponentially and is about four times the hardware required by that of perceptron predictor.

4.4. History Length Vs Prediction Accuracy

The major advantage of the perceptron predictor over the Gshare predictor is that we can have longer history length [5] to avoid aliasing and have better prediction accuracy, which is shown in Tables 4.1 and 4.1. And, as already discussed, we cannot have longer history length for Gshare because it demands more memory size (2^n times).

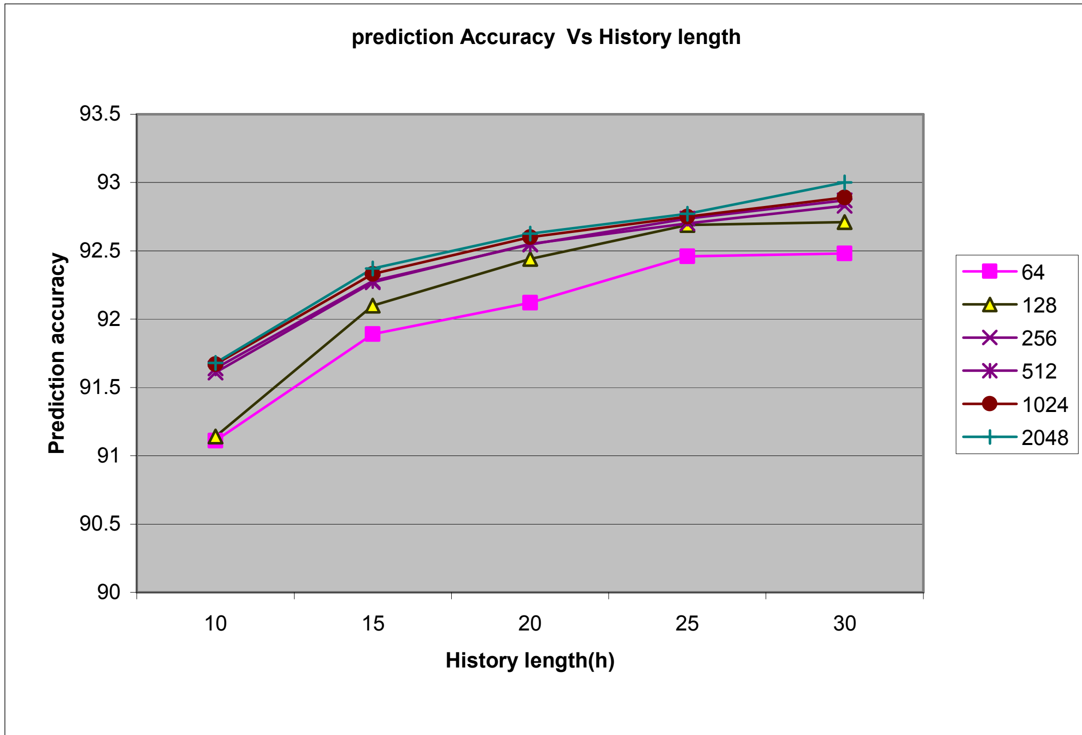

Fig.4.9. Prediction Accuracy Vs History Length for go

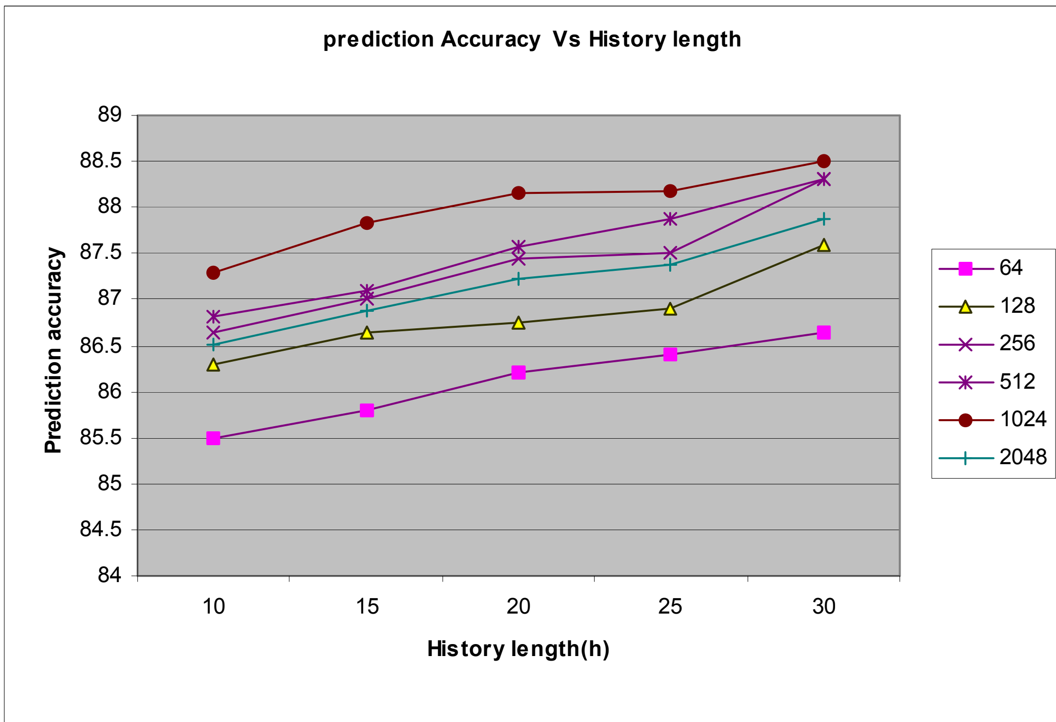

Fig.4.10. Prediction Accuracy Vs History Length for cc1

The benchmarks go and cc1 are programs with a large number of instructions and consequently a large number of branches are being committed and executed. So, increasing the history length will result in more weights and consequently the prediction accuracy increases. This is because, by increasing ‘h’ we are able to capture more of the pattern history and the correlation is stronger. But after a point, increasing the history length seems to have minimal effect on the prediction accuracy.

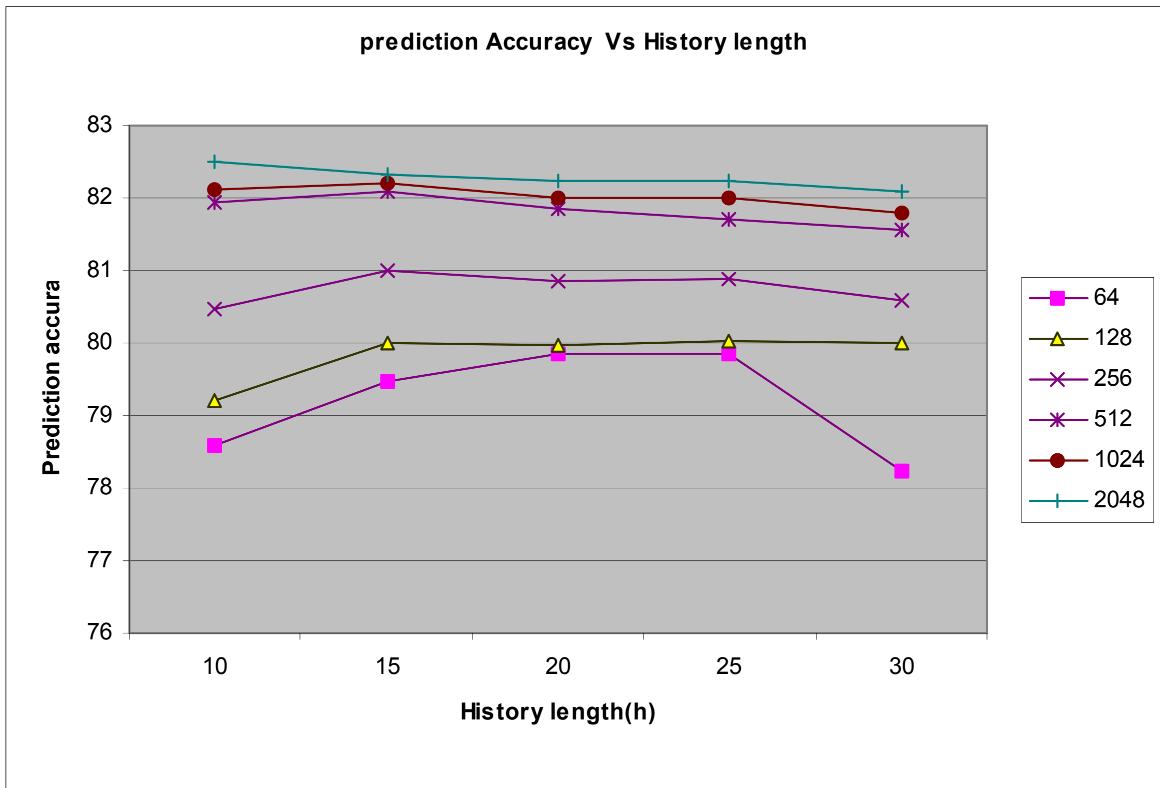

Fig.4.11. Prediction Accuracy Vs History Length for treeadd

For programs with fewer branch instructions, having a very long history length seems to have a detrimental effect. This because some of the entries in the weight matrix of the perceptron predictor are empty, and incorrect values are entered in them during training and when weights are updated. Each row in the W matrix corresponds to a perceptron and

so this would result in incorrect decisions. Hence, for programs with fewer branch instructions, the prediction accuracy seems to increase at first and remains constant until a certain point. Beyond that point the prediction accuracy decreases.

4.5. Training Threshold Vs Prediction Accuracy

The training threshold parameter determines the extent to which the perceptron predictor has to be trained. If the prediction is true and the output is within the range of θ the weights are not updated. So as we increase the threshold, the perceptron requires more weights to be updated and hence the correlation is stronger.

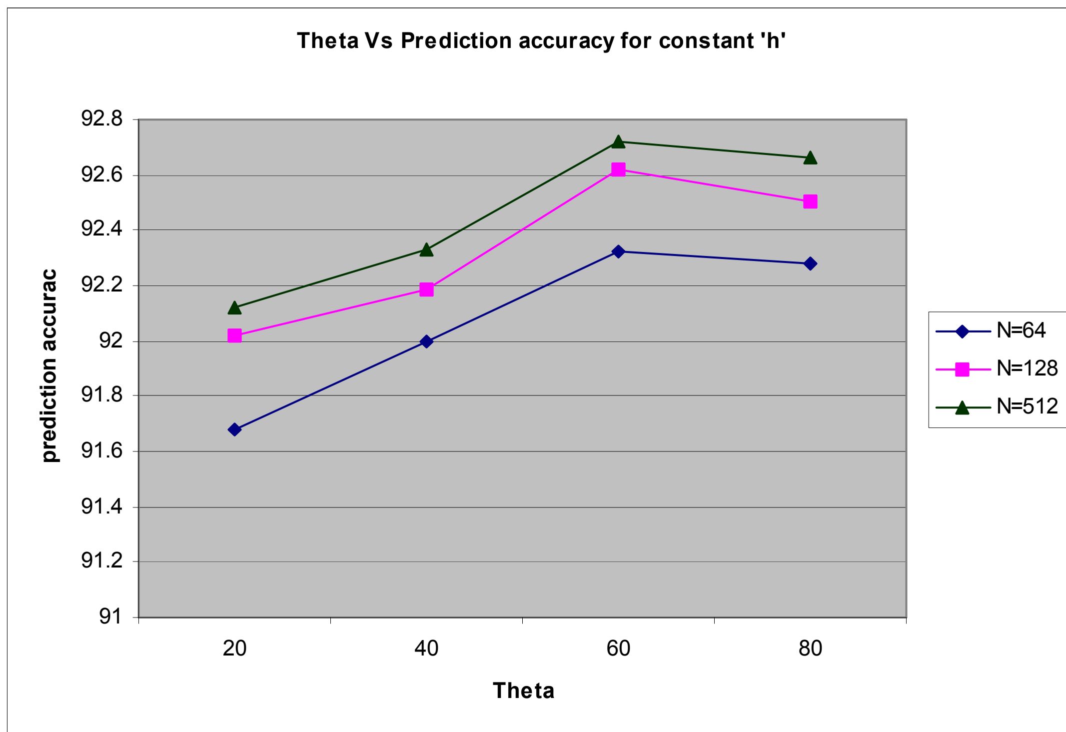

Fig. 4.12. θ Vs Prediction Accuracy for different N and constant h=15

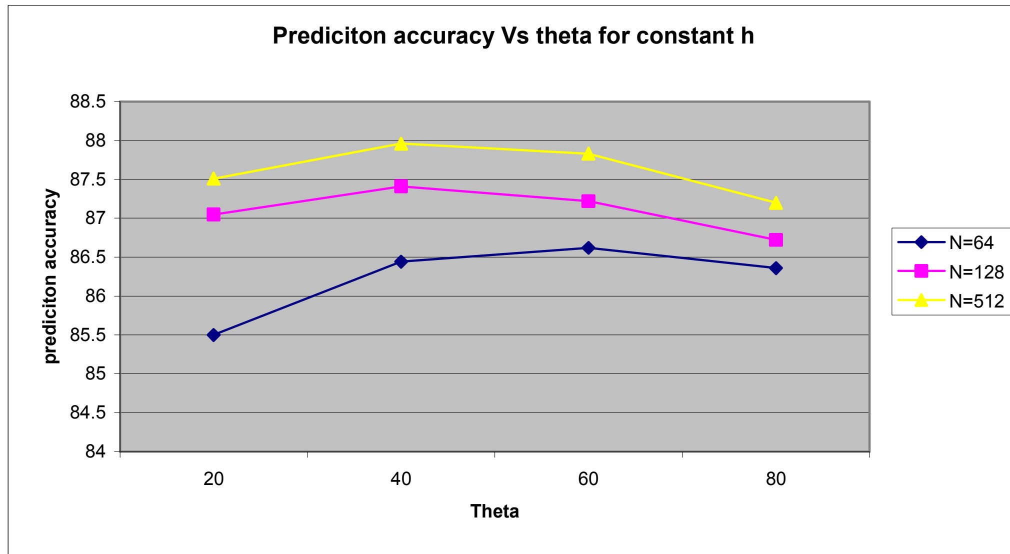

Fig. 4.13 θVs Prediction accuracy for different N and constant h=15

Figures 4.12 and 4.13 show that as we increase the threshold (θ), the prediction accuracy increases up to a certain value and flattens out after that. As we keep increasing the threshold further, the prediction accuracy decreases. For large values of the training parameter θ, the absolute values of the weight vectors in the W matrix would be large. Therefore, if the program suddenly enters a new transitional phase in which the branches follow a different pattern, it would take a very long time for the predictor to adjust to the new pattern. But, if the branches do not enter into any new phase in which they tend to behave differently, a large θ would not be detrimental and is likely to be advantageous.

CHAPTER 5: SUMMARY AND CONCLUSION

5.1. Summary

With an increasing preference to accurate branch predictors, neural predictors such as perceptron predictors seem to be more advantageous than traditional table-based branch prediction techniques. One major merit is their ability to use longer global history length for prediction. The perceptron predictor has an intrinsic ability to learn from its past predictions and the actual branch outcome and uses a training algorithm to update its weights. Also, a small increase in the hardware resources in the form of additional perceptrons helps the predictor to improve its prediction accuracy further because of the increased number of weights for prediction. By tuning the parameters of the perceptron predictor, we have obtained prediction accuracies better than the traditional branch prediction schemes, more specifically the popular Gshare predictor.

5.2. Conclusion

In this thesis, a thorough study on perceptron branch predictor and its behavior have been done. The contribution of this thesis includes findings on the improved performance of the predictor due to increase in number of perceptrons. Also, the response of the predictor to the training threshold parameter was studied.

Results show that the perceptron predictor outperformed the Gshare predictor even with less hardware resources. Also, by tuning various parameters like longer history length, higher threshold value, and increased number of perceptrons, the perceptron predictor achieved better accuracy, than the Gshare.

In order to maintain fixed hardware budget, we increase the number of perceptrons while keeping the history length constant and the results showed an increase in prediction accuracy and this concept was dealt with in this thesis work. The behavior of the perceptron, when programs with fewer branch instructions use longer history length is addressed in this thesis to help understand the behavior of the perceptron predictor better.

APPENDIX

APPENDIX A

Structures used for the perceptron predictor

- The structure given below was a modification made to the existing “**bpred.c**” program of Simplescalar simulator [18].

```
enum bpred_class {
      …
      BPredPerceptron, /* perceptron predictor/*
      BPred_NUMBER_OF_PERCEPTRONS;
};
struct bpred_dir_t{
      enum bpred_class class;
      union{
              …
           struct{
                int number_of_perceptrons;
                int weight_vector_bits;
                int perceptron_history_table;
                int *weights_table;
                unsigned long long global_history;
                unsigned          long           long
spec_global_history;
           } perceptronl;
      } config;
};
```

- The following structure has been added to “bpred.c” of the Simplescalar branch predictor:

```
typedef struct{
      char temp;
      int prediction;    /* prediction: 1 for taken,
                            0 for not taken    */
      int output;         /* perceptron output    */

      unsigned long long int history;
      int *weights_table;
      int *masks_entry;
      unsigned long long *counter_entry_table;
      unsigned long long *counter;
} perceptron_update;
```

APPENDIX B

Configuration of the out of order simulator [18]:

instruction fetch queue size (in inst)
-fetch:ifqsize 8

extra branch mis-prediction latency
-fetch:mplat 3

speed of front-end of machine relative to execution core
-fetch:speed 1

instruction decode B/W (insts/cycle)
-decode:width 4

instruction issue B/W (insts/cycle)
-issue:width 4

-bpred:neural 512 8 15 neural predictor config (<nr_of_perceptrons> <nr_of_weights>
< history_length>)

run pipeline with in-order issue
-issue:inorder false

issue instructions down wrong execution paths
-issue:wrongpath true

instruction commit B/W (insts/cycle)
-commit:width 4

l1 data cache config, i.e., {<config>|none}
-cache:dl1 dl1:128:32:4:l

l1 data cache hit latency (in cycles)
-cache:dl1lat 1

l2 data cache config, i.e., {<config>|none}
-cache:dl2 ul2:1024:64:8:l

l2 data cache hit latency (in cycles)
-cache:dl2lat 10

memory access latency (<first_chunk> <inter_chunk>)
-mem:lat 18 2

memory access bus width (in bytes)
-mem:width 8

-tlb:itlb itlb:16:4096:4:1 instruction TLB config

total number of integer ALU's available
-res:ialu 4

total number of integer multiplier/dividers available
-res:imult 1

total number of memory system ports available
-res:memport 4

APPENDIX C

Example 1:

Branch address counter table

20019b6c 1 −1 1 −1 1 1 −1 −1 −1 −1 −1 1 --- (Xi) – GHR

20028b10 1 8 8 3 8 4 4 1 1 4 −1 1 --- (Wi) - Selected Perceptron

Computation of y =2> 0 so positive and therefore branch is taken

and weights are same:

20028b10 1 8 8 3 8 4 4 1 1 4 1 1 --- (Wi) weights

Example 2:

Branch address counter table

20041a8c 1 1 1 −1 −1 −1 1 1 −1 1 −1 −1 --- (Xi)

20056b78 1 6 −8 13 6 4 4 5 1 −4 1 1 --- (W_i)

computation of y = −21 < 0; so it is not taken and t = −1

Weights are updated to:

t x xi = 1 −1 −1 1 1 1 −1 −1 1 −1 1 --- (t *Xi)

+

20056b78 1 6 −8 13 6 4 4 5 1 −4 1 --- (Wi)

=

20056b78 1 5 −9 14 7 5 3 4 2 −5 2 --- (Wi) weights

APPENDIX D

D.1. Compiling and Running Simplescalar Simulator

This section describes the procedure to compile and run simplescalar simulator

To compile the Simplescalar simulator:

```
cd $IDIR/simplesim-3.0
make
```

Now, build the compiler itself:

```
cd $IDIR/gcc-2.6.3
configure --host=$HOST --target=ssbig-na-sstrix --with-gnu-as
          --with-gnu-ld --prefix=$IDIR
make LANGUAGES=c
../simplesim-3.0/sim-safe ./enquire -f >! float.h-cross
make install
```

Running simplescalar for test benchmark go.alpha which is an alpha binary, using outorder simulator:

```
Sim-outorder –bpred neural –bpred:neural  <number_of_perceptrons>
<nr_of_weights_ bits> <history_length> go.alpha
```

The output is then generated and we obtain the simulator statistics that also contain the branch predictor statistics.

BIBILIOGRAPHY

[1] S. McFarling, Combining Branch Predictors, Technical Report TN-36m, Digital Western Research Laboratory, June 1993.

[2] T.-Y. Yeh and Y. N. Patt. Two-level Adaptive Branch Prediction, In Proceedings of the 24th ACM/IEEE Int'l Symposium on Microarchitecture, November 1991.

[3] D. A. Jiménez and C. Lin, Dynamic Branch Prediction with Perceptrons, In Proceedings of the 7th International Symposium on High Performance Computer Architecture (HPCA), Monterrey, NL, Mexico 2001.

[4] D. A. Jiménez, Fast Path-Based Neural Branch Prediction, In Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO Department of Computer Science), 2003.

[5] Daniel A. Jiménez and Calvin Lin, Perceptron Learning for Predicting the Behavior of Conditional Branches, In Proceedings of the INNS-IEEE International Joint Conference on Neural Networks (IJCNN), Washington, DC, July, 2001.

[6] O. Aciiçmez, Ç. K. Koç, and J.-P. Seifert, On the Power of Simple Branch Prediction Analysis, Cryptology ePrint Archive, Report 2006/351, 2006.

[7] L. N. Vintan and M. Iridon. Towards a High Performance Neural Branch Predictor, In Proceedings of the International Joint Conference on Neural Networks, volume 2, pages 868–873, July 1999.

[8] J. L. Hennessy and D. A. Patterson, Computer Architecture: A Quantitative Approach,

Second Edition, Morgan Kaufmann Publishers, 1996.

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

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

[11] S. Sechrest, C.C. Lee, and T. N. Mudge, Correlation and Aliasing in Dynamic Branch Predictors, In Proceedings of the23rd International Symposium on Computer Architecture, May, 1999.

[12] H. D. Block. The Perceptron: A model for Brain Functioning. Reviews of Modern Physics, 34:123–135, 1962.

[13] J. Stark, M. Evers, and Y. N. Patt, Variable Length Path Branch Prediction. In Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems, October 1998.

[14] M. Evers, S. J. Patel, R. S. Chappell, and Y. N. Patt. An Analysis of Correlation and Predictability: What Makes Two-level Branch Predictors Work? In Proceedings of the 25th Annual International Symposium on Computer Architecture, pages 52–61, July 1998.

[15] V. Desmet, L. Eeckhout, K. De Bosschere. Improved Composite Confidence Mechanisms for a Perceptron Branch Predictor, Ghent University—Ugent Department of Electronics and Information Systems (ELIS), Parallel Information, Systems (PARIS) Group, member HiPEAC, Sint-Pietersnieuwstraat, Gent, Belgium. Journal of System Architecture, March 2006.

[16] O. Cadenas, G. Megson, and D. Jones, Implementation of a Block based Neural Branch predictor, The University of Reading, Reading RG6 6AY, England, UK. In the Proceedings of the 8th Euromicro conference, on Digital System Design (DSD'05), 2005.

[17] Daniel A. Jim´enez and Calvin Lin. Neural methods for dynamic branch prediction. ACM Transactions on Computer ,Systems, 20(4):369–397, November 2002.

[18] Andr´e Seznec, Stephen Felix, Venkata Krishnan, and Yiannakakis Sazeides. Design tradeoffs for the Alpha EV8 conditional branch predictor. In Proceedings of the 29th International Symposium on Computer Architecture, May 2002.

[19] Simplescalar Suite Version 3.0, http://www.simplescalar.com

[20] D. Burger and T. Austin, The SimpleScalar tool set, Version 2.0, University of Wisconsin-Madison Computer Science Department, TechnicalReport #1342, June, 1997.

[21] SPEC 2000 benchmarks, http://www.spec.org

[22] Olden Benchmarks suite-1.0, http://www.cs.princeton.edu/~mcc/olden.html