ABSTRACT

Title of Document:	APPLYING PERCEPTRONS TO SPECULATION IN COMPUTER ARCHITECTURE
	Michael David Black, Ph.D., 2007
Directed By:	Professor Manoj Franklin, Electrical and Computer Engineering

Speculation plays an ever-increasing role in optimizing the execution of programs in computer architecture. Speculative decision-makers are typically required to have high speed and small size, thus limiting their complexity and capability. Because of these restrictions, predictors often consider only a small subset of the available data in making decisions, and consequently do not realize their potential accuracy. Perceptrons, or simple neural networks, can be highly useful in speculation for their ability to examine larger quantities of available data, and identify which data lead to accurate results. Recent research has demonstrated that perceptrons can operate successfully within the strict size and latency restrictions of speculation in computer architecture.

This dissertation first studies how perceptrons can be made to predict accurately when they directly replace the traditional pattern table predictor. Several weight training methods and multiple-bit perceptron topologies are modeled and evaluated in their ability to learn data patterns that pattern tables can learn. The effects of interference between past data on perceptrons are evaluated, and different interference reduction strategies are explored.

Perceptrons are then applied to two speculative applications: data value prediction and dataflow critical path prediction. Several new perceptron value predictors are proposed that can consider longer or more varied data histories than existing table-based value predictors. These include a global-based local predictor that uses global correlations between data values to predict past local values, a global-based global predictor that uses global correlations to predict past global values, and a bitwise predictor that can use global correlations to generate new data values. Several new perceptron criticality predictors are proposed that use global correlations between instruction behaviors to accurately determine whether instructions lie on the critical path. These predictors are evaluated against local table-based approaches on a custom cycle-accurate processor simulator, and are shown on average to have both superior accuracy and higher instruction-per-cycle performance.

Finally, the perceptron predictors are simulated using the different weight training approaches and multiple-bit topologies. It is shown that for these applications, perceptron topologies and training approaches must be selected that respond well to highly imbalanced and poorly correlated past data patterns.

APPLYING PERCEPTRONS TO SPECULATION IN COMPUTER ARCHITECTURE

By

Michael David Black

Dissertation 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
Doctor of Philosophy
2007

Advisory Committee:
Professor Manoj Franklin, Chair
Professor Donald Yeung
Professor Peter Petrov
Professor Robert Newcomb
Professor Alan Sussman

Acknowledgements

I need to acknowledge several people without whose help and advice I would not have completed this dissertation.

First and foremost I thank my wife Jennifer. Not only did she provide unlimited love and encouragement through the long process and supported me through six years of graduate school, but she also contributed greatly to the figures and tables, and read through and corrected my work.

I also thank my advisor, Dr. Manoj Franklin, for giving me advice on possible topics, feedback on writing, and generally teaching me how to do computer architecture research. I also thank Dr. Donald Yeung for his suggestion of possible topics and general feedback on my work.

I would like to thank several fellow graduate students who discussed my ideas with me and gave me suggestions on how to proceed with them. Thanks especially to Gregory Bard for his help with analyzing perceptron behavior, and to Nitin Madnani and Sean Leventhal for giving me feedback. Also thanks to other supportive denizens of the SCA Lab, especially Rania Mameesh and Kursad Albayraktaroglu.

Finally I need to thank my parents, Ann and Bruce Black, for listening to my ideas and giving me much support and encouragement. I also thank Jennifer's parents, Dick and Linda Koch, for cheering me on.

Table of Contents

List of Tables, Figures, and Illustrations

Chapter 1: Introduction

This dissertation studies how perceptrons perform as speculators in microprocessors. I analyze the accuracy and learning capability of perceptron-based predictors and compare them against the more commonly used pattern table-based predictors. I then propose and study perceptron-based predictors in two applications where they have not been widely used before: data value prediction, and dataflow critical-path prediction.

1.1. Speculation

Over the last several years, a series of perceptron-based dynamic branch predictors have been proposed, primarily by Daniel Jimenez [Jim00,Jim04,Jim05]. These predictors use very simple single-layer perceptrons to predict the outcome of a branch instruction in a program at runtime. The perceptron is a simple and early form of neural network, of which more complex versions have been widely used in classification and pattern recognition [Sch92,Sch96]. However, up until the work by Jimenez and Lin, neural networks have generally been absent from processor architecture.

Perceptrons, as well as more advanced neural networks, are an artificial intelligence technique meant to mimic the brain. They learn mathematical functions through an iterative process of guessing and training. The neural network is given an input value to a function it is meant to learn. From the function that it has learned so far, it produces an output value based on that input value. The output value is compared to the output value it should have given had the function been accurate, and

the difference is used to adjust the neural network. Over a period of training iterations, the neural network learns the mathematical function without ever being explicitly told what the function is. In fact, the function need not even be known.

At first sight, neural networks seem completely inapplicable to the strict determinism of computer architecture. Computers process data by taking a series of instructions from the programmer and executing them sequentially. An essential characteristic of a computer is its deterministic nature -- for a given set of input data and a given sequence of instructions, the computer will always produce the same output data after executing the instructions. Neural networks, with all their guesswork and approximations, appear to have no place in computer architecture.

However, this is not entirely the case. Modern computer design is very concerned with optimizations. A computer must execute a program correctly, but within that constraint it should run its program quickly, consume little power, be physically small, and cheap to produce. Modern computers must be sensitive to the needs of the application: while a computer running computationally intensive software must execute quickly, a handheld computer should sacrifice speed for low power consumption. An efficient computer processor should be able to adapt its optimization tactics during the execution of a single program and even from one instruction to another.

Speculation plays an increasingly essential role in computer optimization. It is used to create parallelism in sequential programs, to make frequently used data more accessible to the processor, to adjust the speed of computation, and even to determine whether to apply additional speculation [Bur99]. Speculative systems

generally follow the same model. They accumulate information based on earlier program execution. They use the information to make a decision affecting how data is allocated or how the processor executes instructions. The decision can have correct or incorrect results, where a correct result causes an increase in performance, and an incorrect result frequently causes a decrease in performance (as the system has to backtrack and remedy the results of the incorrect decision). In most cases the ideal result, when known, is used to tune the decision maker.

From an intuitive standpoint, neural networks ought to be ideal for making speculative decisions in a computer system. One can imagine a neural network used as follows. Previous execution information can be fed as input to the network. The output can be used as the speculative decision. When the correct result is known later, it can be used to train the network. There are several reasons, however, that have traditionally barred neural networks from microarchitecture. Neural networks suffer from a large hardware complexity. They are slow and training can take many iterations. Additionally, perceptrons suffer from intrinsic limitations that limit what functions they can learn.

This is not to say that neural networks have been completely absent from computer architecture. However, the few previous applications of neural networks in computer architecture have been in situations where slow speed and high physical complexity are permissible, allowing for large multilayer neural networks [Cav97]. High speed speculative problems, particularly those used to increase instruction level parallelism, have until recently been unable to use neural networks.

Branch prediction research, however, has shown that perceptrons' time in computer architecture has come. Single-layer perceptrons do not have the massive size and training time problems of larger neural networks, and the mathematical limitations do not prevent it from performing well in branch prediction. With the shrinking size and cost of hardware removing the barriers, it is time to introduce more intelligent approaches to speculation problems in microarchitecture.

Many speculators in computer architecture tend to use similar prediction approaches. A typical decision-making approach is a hash table of saturating counters, indexed by a history of past data. The value of the selected counter, is compared to a predetermined threshold, and the result relative to that threshold becomes the prediction. The predictor is later trained when the actual result is known by incrementing or decrementing the counter. Such counter based approaches have been proposed for branch prediction [Yeh92], value prediction [Lip96], criticality prediction [Tun01], confidence estimation [Bur99], last touch cache use prediction [Lai00], voltage and frequency scaling [Gov95], and other applications.

The weakness of the saturating counter approach is its physical size. Speculative applications tend to perform better as the past data history size is increased [Yeh92]. However, by using this history size to index the counter table, a single bit increase in history size doubles the size of the table. This exponential growth strictly limits the history size that can be considered. Thus for table-based dynamic branch prediction, a history size of 17 branches was considered a maximum [Yeh93], despite the fact that greater history sizes could further improve the prediction accuracy.

To cope with this limitation, many predictor designs have severely limited the scope of the past data to values that can be most easily used in making predictions. Local predictors were designed, for which only values observed at past instances of the current static instruction were considered in prediction. Because many applications have high local data locality, reasonably good prediction accuracies could be obtained by focusing exclusively on recent local values. This was done, for example, in branch prediction with the PAp predictor [Yeh93], value prediction with the stride and context predictors [Saz97_2], and criticality prediction with the criteria-based predictors [Tun01]. These table-based predictors obtained fair accuracies while only indexing their tables with a small quantity of past values.

By limiting themselves only to local data, these predictors lose information available globally, or from other instructions, that could allow them to predict more accurately. It has been shown for branch prediction [Yeh93], criticality prediction [Tun02], and confidence estimation [Bla03] that there is information available globally that is not available locally which can improve the accuracy of the predictors. In some studies [Nak99], impractical global predictors were simulated and were shown to substantially outperform the local predictors.

In the perceptron branch predictor, the pattern table indexed by past history is entirely replaced by a perceptron. The advantage of the perceptron is that it grows largely linearly with the past history, not exponentially. The perceptron is thus able to consider significantly longer history sizes than tables and yet remain feasible to implement. Perceptrons are thus able to be used as global predictors. This was the

key factor behind the excellent performance of the perceptron branch predictor [Jim02].

However, the perceptron branch predictor did not perform as well as a global table-based predictor considering an equal size history. The weakness of perceptrons is that they are limited to learning only linearly separable functions. This will be defined in detail in the next chapter. It was found that branch prediction history information often exhibits linearly inseparable functions. The effect of this is that while a perceptron is capable of considering a larger history size than a table, it is typically incapable of extracting as much information from the history as the table.

Despite this learning limitation, the perceptron approach did perform better for branch prediction than other practical predictors. As there are many other speculative applications in which the predictor models are very similar to those in branch prediction, there are other applications that may benefit from a perceptron replacing the pattern-table.

Since the original branch prediction work, perceptrons have been proposed for branch confidence estimation [Akk04] and value prediction confidence estimation [Bla03]. In both of these applications, the table was simply replaced by a perceptron. In some cases, the perceptron performed better. In other cases, it did not [Bla03]. Simply replacing the table with a perceptron without considering the capabilities of the perceptron is likely to produce good predictors only by accident. It is important to understand when and why perceptrons perform better than a table-predictor, what exactly the pattern table learns that they do not learn, and when those unlearnable patterns arise. Knowing this allows a perceptron to be designed that is a good fit for

the application. Perceptrons can be designed with different training procedures or topologies. It is important to understand how to choose the right perceptron for an application.

1.2. Dissertation Overview

This dissertation has three core parts. In the first part I seek to understand how perceptrons behave and learn in theory when compared to the pattern table. The second and third parts explore different perceptron approaches for data value prediction and critical instruction prediction, respectively.

Why these two applications? Recent past perceptron applications other than branch prediction, such as confidence estimation, use very similar predictor designs to branch prediction. Both confidence estimation and branch prediction use single bit outputs (take/don't take). Both have single bit past inputs. Both can be trained soon after a prediction is made. Value prediction and criticality prediction are interesting because, while being similar to branch prediction in many ways such as latency requirements, each of them pose challenges that branch prediction does not pose.

Value prediction, unlike branch prediction, requires a multiple bit value to be predicted. This raises many challenges. How can a perceptron be best designed to produce multiple bits? Do perceptrons learn the same for multiple bits as for one bit?

Criticality prediction only requires a single bit decision: instruction is on the critical path / instruction is not on the critical path. However, unlike branch prediction, criticality cannot be immediately evaluated for an instruction, even after that instruction commits. How can the predictor be trained? A solution to this is not to train on criticality directly, but to train on whether the instruction exhibits critical

behaviors (or criteria) [Tun01]. However, this introduces additional questions. Which criteria should the perceptron use to train? How should the perceptron train when there are multiple criteria, and multiple correct answers?

My research in this dissertation follows the following methodology. In the theory chapter, I describe different perceptron training approaches and topologies, and determine when one approach works better than another. For both value prediction and criticality, I propose and evaluate many different predictors using different perceptron styles and configurations. These different perceptron predictor configurations are chosen without regard for which theoretically makes the best use of the perceptron for that application. All are evaluated, and through the evaluation it becomes apparent which is the better perceptron approach for that application. Finally, I look at how the perceptrons approaches learned in each application, and analyze why one approach turned out to be a better fit for that application than another.

The dissertation is organized in the following way. The next chapter, Chapter 2, covers the origins and background of perceptrons, and discusses how they are used in branch prediction and confidence estimation.

Chapter 3 contains my theoretical contributions. The chapter first looks at how perceptrons learn relative to how tables learn, and then analyzes several perceptron training approaches. Next it proposes several multibit perceptron topologies, and analyzes how they learn. The chapter finally discusses interference in the history and discusses several ways of overcoming it.

Chapter 4 presents the simulation methodology. It provides a detailed description of the processor simulator I designed for this work. The chapter also provides the simulation parameters used in the subsequent chapter. It finally describes the algorithms used to simulate the perceptrons.

Chapters 5 and 6 respectively propose and evaluate several perceptron value prediction and criticality prediction approaches. Both chapters commence with a background discussing previous work in value prediction and criticality prediction. In Chapter 5 I next propose two perceptron predictors that only consider local value history, a perceptron predictor that considers global history to predict locally available values, a perceptron predictor that predicts past globally available values, and a perceptron predictor that can produce new data values. In Chapter 6 I propose four different configurations for a perceptron criticality predictor. Chapters 5 and 6 conclude by evaluating each perceptron predictor against a standard baseline predictor. In Chapter 5, improvements are shown over the baseline in both prediction accuracy and instruction-per-cycle performance for several of the value predictors. In Chapter 6, improvements are shown over the baseline in prediction accuracy for several of the criticality predictors. Performance improvement is then demonstrated by using the criticality predictors as confidence estimators for value prediction.

In Chapter 7, which concludes the dissertation, the perceptron weight values and performances for different training styles are used to determine why one perceptron approach works better than another perceptron approach for each application. The chapter concludes by summarizing the dissertation findings and results, and proposes some future areas of study.

1.3. Contributions

The following is a concise list of the contributions of this dissertation:

In Chapter 3:

- An analysis of how perceptrons learn context patterns with regard to imbalance between patterns and compatibility between patterns
- An analysis of two perceptron training strategies and their learning rates with regard to the number of correlated inputs
- Three multibit perceptron topologies: disjoint, fully coupled, weight for each input value and an analysis of the number of value correlations each can learn
- An analysis of history interference, its effect on perceptron learning, and two strategies for combating it: assigned seats, piecewise linear

In Chapter 4:

- A completely new execution-driven out-of-order processor simulator

In Chapter 5:

- Two perceptron-based local value predictors: perceptrons in value table, perceptrons in pattern table
- Three perceptron-based local predictors using global information, based on the three multibit topologies.
- A perceptron-based global predictor using a global value cache
- A perceptron-based global predictor using no stored past values (bitwise)

In Chapter 6:

- Three perceptron-based critical criteria prediction approaches

Chapter 2. Background

2.1. The Perceptron

The perceptron model used in the recent branch prediction research is possibly the simplest and earliest nontrivial neural network model in existence. It is common for textbooks on neural networks to open with that example before proceeding to more complex neural networks [Rus95]. The reasons for its use in branch prediction, as mentioned previously, are due to strict speed and training latency restrictions.

Modern neural network research has largely forsaken the original perceptron, due to its learning restrictions and simplicity. In fact, a scan of all the papers published in the IEEE Transactions on Neural Networks since 1990 shows only 3 papers that even mention this perceptron model in the title or abstract, and no papers that deal with it exclusively. Besides computer architecture, there would appear to be no major current applications of the basic perceptron. Consequently, to find any analyses of the perceptron, it is necessary to step back 40-50 years to the original work that proposed it.

2.1.1. Rosenblatt's perceptron

The earliest form of neural network, the perceptron, was first formally proposed by Frank Rosenblatt in 1957, and was inspired partly from a symbolic logic representation of neuron cells introduced by McCulloch and Pitts in 1943 [Nag91]. Modeled after collections of neurons, the perceptron was among the first so-called "black box" artificial intelligence approaches, which could learn functions and perform tasks without being explicitly told the rules [Rus95]. Although the

perceptron and neural networks were at first supposed to be the key to artificial intelligence, not to mention a tool for understanding the brain, perceptrons have since become generally limited to the role of pattern recognition and classification.

In his book “Principles of Neurodynamics”, Rosenblatt defines perceptrons thus: “a set of signal generating units (or “neurons”) connected together to form a network. Each of these units, upon receiving a suitable input signal (either from other units in the network or from the environment) responds by generating an output signal, which may be transmitted, through connections, to a selected set of receiving units” [Ros62].

Rosenblatt defined the perceptron in terms of S (sensory) units, A (association) units, and R (response) units, the coupling of which is defined by an interaction matrix. At this time, perceptrons were considered in terms of a computer software model (to be simulated on the Mark I) and the interaction matrix comprised the memory of the neural network. The S units were defined as a “transducer responding to physical energy” which “generates an output signal si=+1 if its input signal exceeds a given threshold, and 0 otherwise.” The A unit is “a logical decision element which generates an output signal if the algebraic sum of the input signals alpha is equal or greater than a threshold quantity theta>0. The output signal is equal to 1 if alpha>=theta and 0 otherwise. If alpha=+1, the unit is said to be active.” The R unit “emits the output r=+1 is the sum of it input signals is strictly positive, and r=-1 if the sum of the input signals is strictly negative. If the sum of the input signals is zero, the output can be considered equal to zero or indeterminate.” The interaction matrix is “the matrix of coupling coefficients for all pairs of units.” Each pair has a

value in the matrix; if the value is zero the units are considered unconnected. Rosenblatt defines a “simple perceptron” as a perceptron satisfying five conditions, among them that there is only one R unit with a connection from every A unit, that the perceptron has connections only from S unit to A units and A to R units, the S to A connections have an unchanging unit value, and that the connections are unidirectional. An example of this is shown in Figure 2.1. In modern work, this simple perceptron has become what is meant when the word “perceptron” is used.

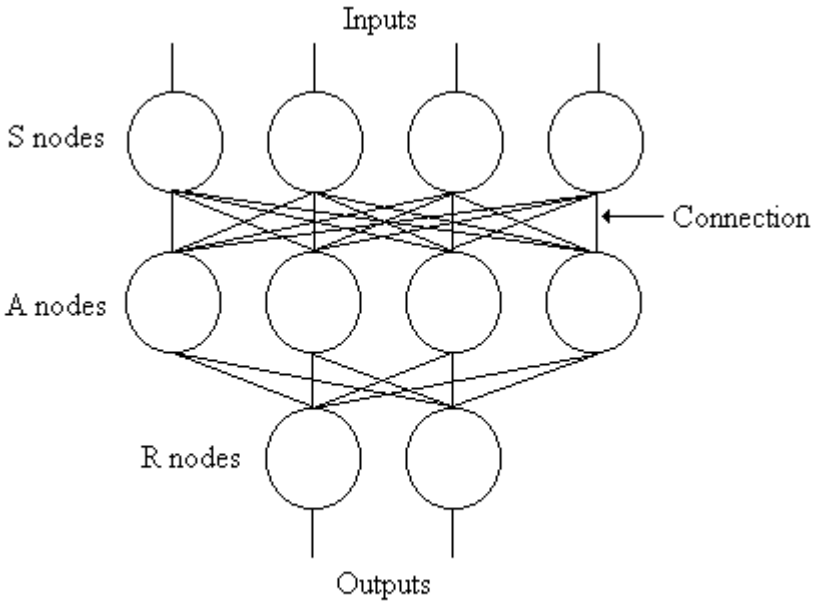


Figure 2.1. Rosenblatt’s perceptron

In this dissertation, the units will be called with more modern names. The S units are called “inputs,” and always output a value of 1 or -1. In most of the applications discussed, the stimuli are single bits; thus the purpose of the input is to simply convert 0 to -1. The A units, which were later renamed “hidden nodes”, are simply dispensed with, and the inputs are directly connected to the R unit. The value

of the connection between each input and the R unit is called a “weight” - these weights comprise the entire storage of the network. Like in Rosenblatt’s definitions, the value arriving at the R unit from the inputs is the 1 / -1 input value times the weight. The R unit performs two functions: it sums the weights, and it compares with the threshold to produce the output value. In this dissertation the unit will simply be considered as a sum unit and a threshold unit. Figure 2.2 shows this perceptron. It effectively takes the dot produce of the inputs and the weights, and returns 1 if that dot produce exceeds the threshold, and -1 otherwise. Thus the perceptron basically predicts using:

$$\sum W_k I_k > 0? \;\; 1 \,/\, -1.$$

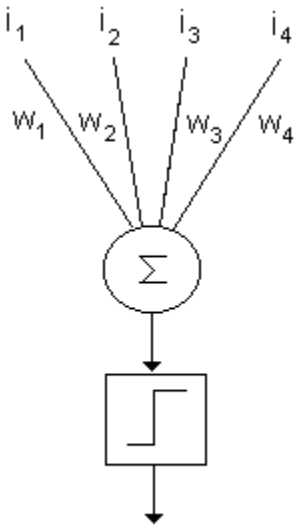


Figure 2.2. Basic perceptron

2.1.2. Training

The purpose of the training procedures is to adjust the weights in response to a desired output, so that the perceptron can learn to predict that output for a current set of inputs. Rosenblatt introduces three training procedures: response-controlled

reinforcement, stimulus-controlled reinforcement, and error-corrective reinforcement. Two weight adjustment methods are proposed: “alpha system reinforcement”, which adjusts the weights by a constant value *eta*, and “gamma system reinforcement”, which adjusts in such a way that the total quantity of all weights is zero. In this dissertation, only alpha system reinforcement is used. Response-controlled reinforcement determines eta entirely from the output value of the perceptron, and adjusts all weights equally. This approach, being highly limited, is not used in this dissertation, and as far as I can tell, has not been seriously considered since Rosenblatt. Stimulus-controlled reinforcement, which I term “training-by-correlation”, has been used in the perceptron branch prediction work. It uses the input values to determine eta for each weight. Error-corrective reinforcement, which I term “training-by-error”, adjusts weights only when the perceptron is wrong; it determines an error value using the output value of the perceptron, and uses the input values to determine an eta for each input. The magnitude of eta for alpha system reinforcement is called alpha; in this dissertation, an alpha of 1 is always used.

The objective of the perceptron is to learn correlations between each input value and the output. Each weight determines what and how much effect its input has on the output. A positive weight means that the input has a direct effect on the output, whereas a negative weight means the input has an inverse effect on the output. If the weight is close to zero, the input is found to have little effect on the output; if the weight has a large magnitude it has a strong effect on the output. Thus the perceptron is able to judge which inputs affect the output, and to what degree they do.

The objective of training is to adjust the weight value according to the perceived correlation.

Training-by-correlation works as follows. The weights are adjusted in response to the correlation observed for each input. Thus if an input is the same as the desired output, the weight for that input is incremented; if it is different, the weight is decremented. The prediction output of the perceptron is not taken into account.

Training-by-error only adjusts if the perceptron was wrong. An error value å is computed as å = desired output - predicted output. The perceptron is trained by multiplying å by each input and adding it to the corresponding weight:

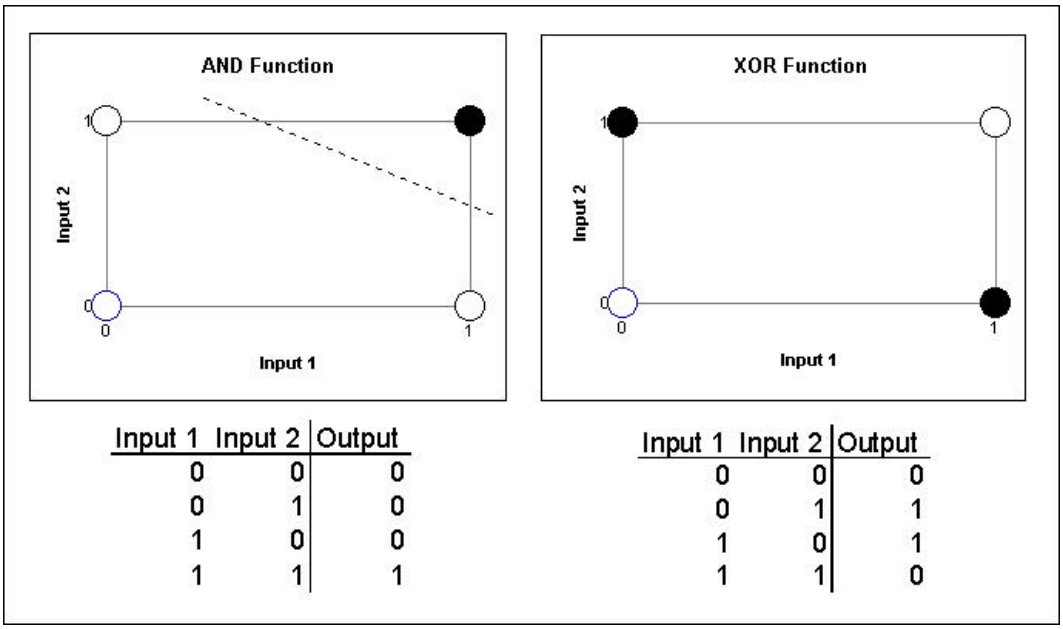
($w_k = w_k + i_k e$).

Not yet covered is how the threshold value theta is chosen. One simple approach is to use a constant value, such as 0. However, it is generally considered desirable to dynamically adjust theta in such a way that it reflect the proportion of desired 1 outputs to desired 0 outputs (the more 1's, the lower the theta). This can be accomplished by subtracting the desired output from theta, in training-by-correlation, or the error, in training-by-error. A more easily implemented way, however, which is mathematically equivalent is to have an extra weight "bias weight" connected to an extra input hardwired to 1 [Rus95]. The bias weight is adjusted like any other weight, according to the training policy, and is added to the sum to produce the perceptron output. By including a bias weight, theta can be permanently set to 0.

2.1.3. Linear Inseparability

It was originally theorized that neural networks, of unlimited size, could learn all continuous functions. In a 1969 work by Minsky and Papert [Min69], it was shown that this was not the case for perceptrons; that they were in fact limited to learning only functions that are "linearly separable." Minsky's work originally claimed that this was the case for all neural networks, but it was later discovered that linearly inseparable functions can be learned in larger neural networks using hidden layers and more advanced training mechanisms. However, this is still a handicap for the simple single-layered perceptron.

Linear separability is classically pictured geometrically in an *n*-dimensional space, where *n* is the number of inputs. All the possible outputs are placed in the space. If the space can be divided by a plane so that all positive outputs are on one side of the plane and all negative outputs are on the other side, the function is linearly separable [Rus95]. If no plane can be drawn, the function cannot be learned by a perceptron. This is illustrated in Figure 2.3 with the AND function, which is linearly separable, and the XOR function, which is not. This geometrical analysis most likely became popular because image classification was one of the first major applications of neural networks.



Input 1	Input 2	Output
0	0	0
0	1	0
1	0	0
1	1	1

Input 1	Input 2	Output
0	0	0
0	1	1
1	0	1
1	1	0

Figure 2.3. Linear inseparability

Linear separability may be better illustrated for computer architecture applications by looking intuitively at why perceptrons can only learn such functions. In a perceptron, the effect of an input on the output is determined by its weight. As stated before, a positive weight means that the output varies directly with the input, while a negative weight causes the output to vary inversely with the input. Based on its weight, a 1 at a particular input can make the total output more positive or more negative. However, a 1 at a particular input cannot make the total output more positive sometimes and more negative at other times. Functions tend not to be linearly separable if one input's effect on the output relies on another input's effect. The effect of limiting a perceptron to linear separable functions is analyzed in more depth in the next chapter.

2.1.4. The Perceptron in Hardware

Figure 2.4 shows how a perceptron can be implemented in hardware. The weights are implemented as up-down binary counters that saturate at maximum and minimum values (the minimum has the same magnitude as the maximum and opposite sign). The range of the weights needed to learn effectively is analyzed in Chapter 7; weights with a size ranging from 6 to 9 bits tend to suffice. The analyses in [Jim02] used an 8 bit weight.

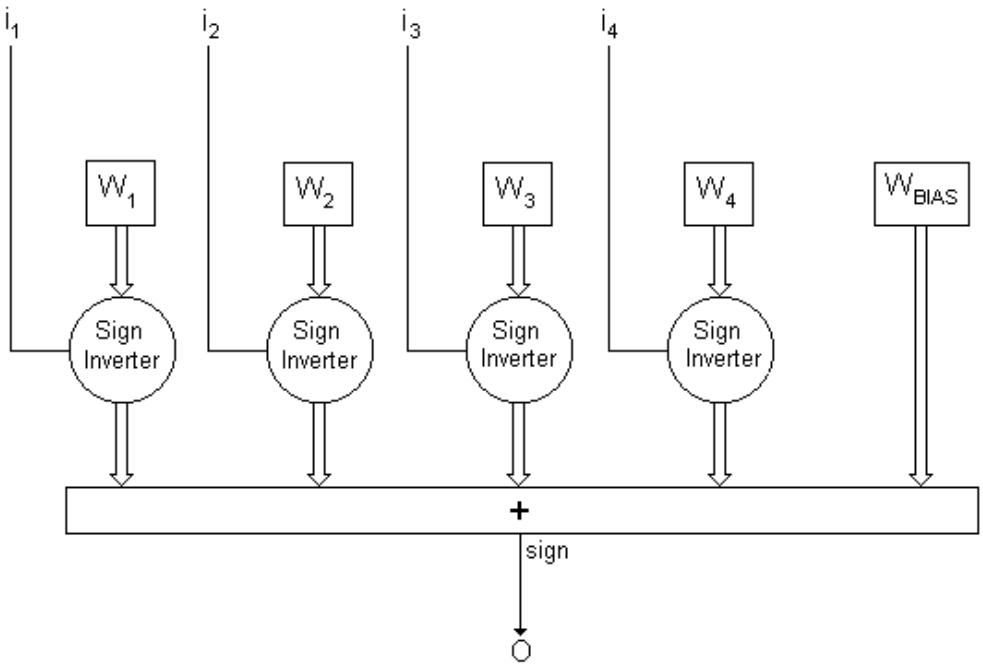


Figure 2.4. Perceptron modeled in hardware

Because an input bit is interpreted only as -1 or 1, the product between the input bit and its corresponding weight can be implemented simply by using the input bit to choose whether to invert the sign of the weight. These products are summed together, but by using a threshold of 0, only the sign of the total sum is used as the output.

To implement training-by-error, the weights are only adjusted if actual XOR predicted is 1. In this case, the error is 1 if a-p is 2, and -1 if a-p is -2; thus the error is simply a if the weights are adjusted. The weight adjustment, eta, is simply a XOR error (with 0 changed to -1) added to the weights. Because eta is always of magnitude 1, the weights can be simply implemented as a binary counter.

Thus the primary hardware costs are 1) the storage of each weight bit, and 2) the binary adder to sum all the weights, with the remaining logic being of trivial size in comparison. Of these, the storage complexity grows for n inputs at O(n) for the weights, and O(n log n) for the adder [Jim02]. The prediction delay grows at O(log n) for the adder, and is O(1) elsewhere, if a Wallace-Tree adder is used. Training time is O(1). In their analysis, the authors determined that the physical space cost was dominated by the weight storage. In the cost analyses of the various perceptron approach described in this dissertation, the weight storage space for 8-bit weights is used as physical cost of the perceptron.

The hardware latencies were formally analyzed in [Jim02] using HSPICE to model the perceptron and CACTI to model tables of perceptrons. Table 2.1 shows the perceptron delay as a function of perceptron inputs using 180 nm technology as reported in [Jim02]. The delay in indexing a 4096 entry perceptron table was reported as 571 ps. Thus the total latency for making predictions can be determined by adding the two figures. On a 1 GHz processor, a prediction for a perceptron with a 16 bit history takes 1.7 cycles. As processor speeds increase, this number will grow worse, but as transistor technology improves, it should grow better. In this

dissertation, it is assumed that all predictions can be made in 2 cycles or less, and training can be performed in 1 cycle.

History Size	Perceptron Delay (ps)
4	811
9	725
13	1090
17	1170
23	1860

Table 2.1. Perceptron delays as reported in [Jim02]

This 2 cycle latency was a problem for branch prediction because branch predictions must be made at Fetch. In the other applications analyzed in this dissertation, predictions are not needed until Dispatch, while the information needed to make predictions (the instruction PC) is available at Fetch. Consequently, the prediction latency is not considered as an issue in this work.

2.1.5. Multilayer Neural Networks

Most modern neural networks have multiple layers and more elaborate training approaches. A standard approach used for training is backpropagation, which was described in [Rus95]. A three layer neural network employing backpropagation is capable of learning linearly inseparable functions.

Although multilevel neural networks, once trained, might theoretically outperform perceptrons in speculation, there are several problems with employing them. The first problem is the physical size of adding additional layers with additional weights. Adding a hidden layer effectively doubles the size of the network. A more serious problem is the additional latency of performing another summation. Doubling the latency to 4 cycles would have a substantial deleterious

effect on neural approaches to the applications discussed here; even making the predictor redundant.

However, the most serious problems relate to training. Backpropagation requires a continuous threshold function because the derivative of the threshold function is used to adjust the weights. The perceptron threshold function is the step function, which cannot be differentiated. Most neural networks employing backpropagation use the sigmoid function ($\frac{1}{1+e^{-x}}$) as a threshold function because it approximates the shape and mathematical characteristics of the step function while being continuous (and differentiable). However, using a continuous function requires floating point numbers (or at least large integers), substantially complicating the hardware costs and increasing the latencies. This may be compensated by implementing the neural network as analog components, but it is not clear that analog neural networks yet run at the desired latencies.

The most serious concern is training time. As will be shown in the next chapter, perceptrons can typically be trained in approximately the same number of training iterations as the table-based predictors they replace. Because of the slower learning rate and multiple layers, larger neural networks require substantially more (orders of magnitude higher) training iterations to learn. This makes them slow to predict correctly at first, and slow to adapt to context changes in programs. Thus even if a high latency, implementable multilevel neural network could be implemented, it would be highly unlikely to predict accurately as rapidly as the perceptron, and would consequently almost certainly perform worse.

2.2. Perceptron Branch Prediction and Other Architecture Applications

Branch prediction, being the first successful perceptron application, becomes my template for designing other perceptron predictors. Before discussing other approaches, it is necessary to cover perceptron branch prediction, how it works, and where it evolved from.

2.2.1. The Two-Level Branch Predictor

The two-level branch predictor, proposed by Yeh and Patt in 1992 [Yeh92], became a standard for branch predictor design. The predictor was based off of the original dynamic branch predictor by Smith, which worked by using a table of counters hashed by the branch PC. The Yeh and Patt predictor took this a step further by using information from other branch instructions to make predictions. A shift register holding the history of global branch outcomes was used to hash a second table of counters. Their significantly more accurate two-level predictor captured correlations between the outcomes of different branch instructions. A variation on the two-level predictor that uses a combination of global branch history and branch PC to hash the counter table, McFarling's gshare predictor became widely used as a baseline predictor for performance comparisons.

The Yeh and Patt PAg predictor works as follows. A global history of branch directions is stored in a shift register. Branch outcomes are shifted into the table as soon as they are known. A pattern table is selected from a table of pattern tables using the lower bits of the current branch program counter. The concatenated binary branch outcomes form an index to this pattern table, selecting a saturating counter.

This counter value determines whether to take the branch prediction; if it is greater than a threshold, it predicts take, otherwise it predicts to not take.

The problem with this prediction approach is its large size. The pattern table grows exponentially with the number of bits in the history register, and it then must be replicated for each branch instruction in the first level table. Because this predictor was too massive to be practical (it was estimated that the global history size cannot exceed 17 and be practical [Yeh93]), the gshare predictor emerged. It uses a global pattern table, but is indexed by the global branch history XORed with the current branch PC, making a unique index. The gshare predictor was claimed to achieve 97% accuracy for 32k hardware size [McF93]. However, the weakness of the predictor is the aliasing between hashes to the global table. McFarling's own measurements showed a local PAp predictor performing significantly better.

2.2.2. Perceptron Branch Prediction

The success of the perceptron branch prediction, proposed by Jimenez and Lin [Jim00], over gshare is partly due to the fact that it is effectively a PAp predictor without the problems of exponential table growth. The aliasing problems of gshare are thus avoided. Figure 2.5 shows a block diagram of this predictor. A global branch history stores the recent branch outcomes. The last bits of the branch instruction address index a table of perceptrons (analogous to the table of pattern tables) and choose a perceptron. Each branch outcome is converted to a 1 or -1 and is fed to a separate perceptron input. The perceptron output is simply the decision of whether to take the branch or not. Training is performed using the training-by-correlations approach.

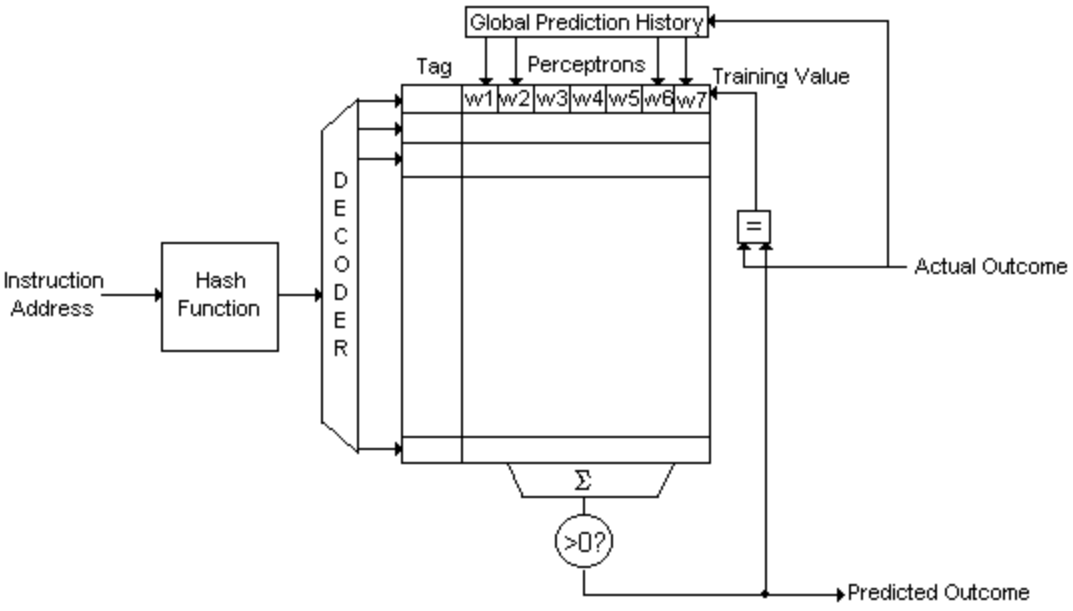


Figure 2.5. Perceptron-based branch predictor block diagram

In their earliest paper, the authors claimed a 25.6% reduction in relative misprediction rate over gshare at a 128k hardware size, and estimated a 18% increase in program performance [Jim00]. In follow up work the authors increased the misprediction rate to 27% [Jim02]. However, the predictor suffered from a 2 cycle long latency, making it impractical to achieve this rate initially. To compensate for latency issues, the authors used a gshare predictor to make the initial prediction, and then used a perceptron on the next cycle to overturn the gshare prediction if inaccurate [Jim00].

In a more extensive follow-up work that considered latency and hardware size issues, the authors also tried to quantify the branch information that the perceptron could not learn. They determined linearly inseparable branches as sets of branch history patterns that required the perceptron to learn a linearly inseparable function. They found that approximately 3-4% of branches had completely linearly inseparable

functions [Jim02]. They also found that when the history size was increased, the perceptron was more capable of learning the linearly inseparable functions. My analysis and rationale behind this phenomenon is explained in detail in the first part of Chapter 3.

It is important to note that this is not the very first neural branch predictor proposed. Two neural approaches were proposed in a paper by Vintan in 1999 [Vin99]. These predictors were not even close to practical from a hardware standpoint, however, but did manage to achieve accuracies comparable to the table-based approaches. An even earlier seminar paper in 1996 by Kuvayev [Kuv96] claims to have developed a neural branch predictor; however, the prediction algorithms, predictor topology, and methodology are never mentioned in the paper.

2.2.3. Piecewise Linear Predictor

Jimenez refined his predictor in a follow up work [Jim05], which addressed interference problems in the global history table. His piecewise linear predictor maintains the past branch addresses as well as past branch history. Rather than have a single weight for each history entry, an array of weights is maintained, and is selected using the history. This avoids multiple branches from occurring at the same global history entry and interfering with each other. A 3.21% misprediction rate was claimed for this predictor at a history length of over 80. This predictor is discussed in detail at the end of Chapter 3.

2.2.4. Other Perceptron Applications in Architecture

Two additional notable perceptron applications are both in confidence estimation. Branch confidence estimation was explored in [Akk04]. Confidence estimation for value prediction was explored in [Bla03,Bla04,Bla05_2].

2.2.4.1. Perceptron-based Confidence Estimation for Branch Prediction

Perceptron-based branch confidence estimation, while suggested as a future work by Jimenez [Jim00], was first performed by Akkary et al in 2004 [Akk04]. Confidence estimation for branch prediction has been proposed to limit CPU resources wasted in predicting unpredictable branches. It is practical if the CPU resources dedicated to prediction could be used for other tasks, or if the branch predictor consumes sufficient power so that not predicting can significantly reduce the CPU energy usage. As branch predictors become more complex (the perceptron branch predictor being a case in point), reducing the energy consumption of the branch predictor becomes increasingly useful [Gru98].

The perceptron-based branch confidence estimator is virtually identical to the perceptron branch predictor, with a global branch history and a table of perceptron indexed by the branch instruction address. The key difference is that the accuracy of the branch prediction is stored in the global history rather than the direction of the branch. The authors evaluated their predictor against a preexisting table-based branch confidence estimator that was organized similarly to gshare. The authors claimed a 10% reduction in the number of microoperations performed by the CPU without a loss in performance.

An interesting facet of this branch confidence estimator is that, while it is clearly based on Jimenez's perceptron branch predictor, it uses a training-by-error training strategy rather than the training-by-correlation strategy which Jimenez consistently used in his predictor. Since the reason for the change is not discussed (and in fact the authors even explicitly claim that the training approach is based on Jimenez) it is tempting to assume that the authors were not aware that they were using a different training strategy.

2.2.4.2. Perceptron-based Confidence Estimation for Value Prediction

Confidence estimation for value prediction was introduced by this author in my Master's thesis and is detailed in [Bla03]. Confidence estimation is used to lessen the value prediction misprediction penalty by guessing whether or not to use a value prediction result. The perceptron approach was compared to the local saturating counter approach used by Lipasti's value predictor [Lip97_2] and in subsequent approaches. The perceptron-based confidence estimator is shown in Figure 2.6 and is structured very similarly to the perceptron-based branch predictor. Past global value prediction accuracies were stored in a global history table. A perceptron was selected from a table of perceptrons by the instruction address, and the global prediction accuracy history was sourced to the inputs of the selected perceptron. The perceptron output decided whether the value prediction would be used. The perceptron was trained using training-by-error.

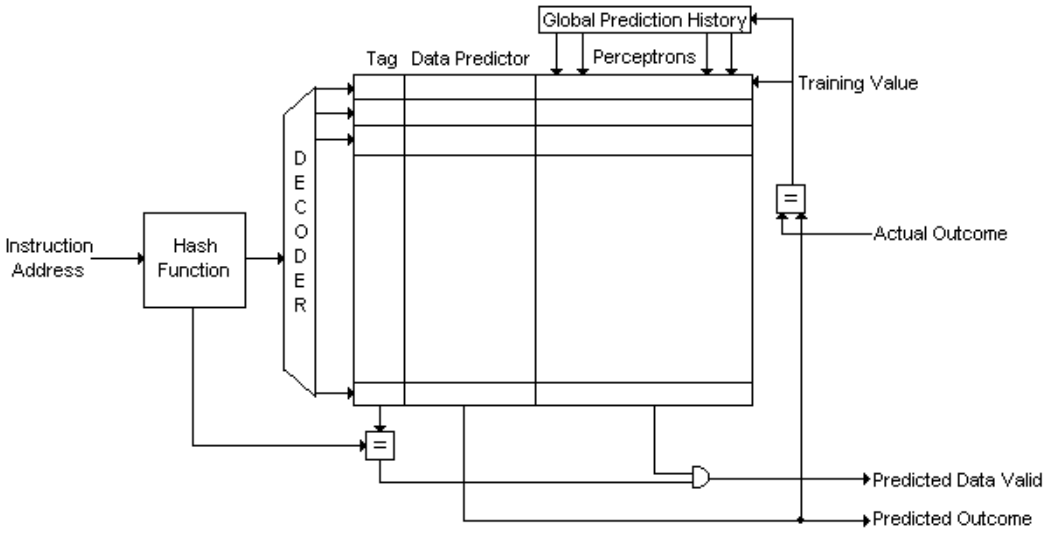


Figure 2.6. Perceptron-based confidence estimator for value prediction

Evaluation was performed using the confidence estimator on three value prediction approaches: stride, last value, and context. The value predictor employing the confidence estimator was evaluated on its prediction accuracy and its coverage, or the percentage of the time that predictions were used. The evaluations showed a coverage improvement of 6% to 10%, and accuracy improvement of 2% to 6%.

An important note about the perceptron confidence estimator is that it replaced a local approach that only used past iterations of the instruction to make a value prediction with a global approach that used the prediction accuracies of past global instructions. While global confidence estimation had been proposed using tables [Cal98], the massive size of the tables made it impractical. By using perceptrons, global value prediction predictability information could be harnessed with suffering the exponential growth of a globally indexed table.

Chapter 3. Theory

Prior to the use of perceptrons, the most accurate branch predictor was the two-level table-based branch predictor. When replacing the table-based predictor with the perceptron, the authors largely mimicked the basic table-based layout [Jim00]. Their perceptron predictor still used the same per-branch-address construction, the same value table, and the same global branch history. The principal differences are that the pattern table is replaced by the perceptron, and the summation of perceptron weights is used to determine the branch prediction, not the value of a saturating counter. The perceptrons are thus given the same information as the table, the same past branch sequences, and are asked to make predictions for the same branch instructions in the same order. In the implementations in this dissertation for value prediction and critical instruction prediction, the perceptron approaches will likewise replace a previously existing table-based predictor. The real question is consequently not how accurately the perceptron predictor predicts in isolation, but how accurately the perceptron predictor predicts when compared to a similarly constructed two-level table predictor incorporating saturating counters. When does the perceptron learn faster than the table? When does the perceptron fail to learn patterns that the table easily grasps?

This chapter explores several facets of how a perceptron behaves when it directly replaces a lookup table. The first topic I examine is under what circumstances a perceptron can learn the same patterns a table can learn. Are there patterns a table can learn that a perceptron cannot learn at all, and what are they?

When can a table learn input data faster than a perceptron, and when could a perceptron learn faster?

The second topic is how the perceptron training strategy affects its learning when the perceptron replaces the lookup table. Past works have used two different training strategies for perceptrons in computer architecture applications. Under what circumstances is one training strategy better than another? What input data characteristics affect the performance of each training approach?

In prior computer architecture work using perceptrons, the applications have all required only a single bit output. Branch prediction and confidence estimation need only a binary "yes/no" decision from their predictors. However, there are many speculative applications that require a multiple bit decision, and lookup tables have been designed for these problems. As the third topic, I examine several ways that perceptrons can be used to predict multiple-bit values. When are perceptrons unable to predict values that lookup tables can predict? Is there any way a multibit perceptron-based predictor can be designed so that it has the same learning power as a multibit table-based predictor?

The last topic I explore is how perceptron predictors cope with interference and aliasing. Because no predictor can be designed that is massive enough to independently consider every single instruction in a program, aliasing between instructions has always reduced table-based predictors' accuracies and learning potential. Perceptron predictors will likewise suffer from interference. Do perceptron predictors respond to interference in the same way as table-based predictors? Is there any way of reducing aliasing's harmful effects in perceptrons?

3.1. Perceptron Learning

The basic function of a perceptron is to learn correlations between pairs of single-bit data points. The classic perceptron has a single weight for each binary input. The polarity of that weight tells whether that input is directly correlated with the target (a positive weight), or inversely correlated with the target (a negative weight). The magnitude of the weight tells the degree of correlation between the input and the target. A large weight implies that the input is greatly correlated, and always carries the same value relative to the target, while a small weight implies that the input is modestly correlated, and is not necessarily a trustworthy guide for predicting the target. A large weight carries a great degree of influence on the final decision, while a small weight has little influence. Thus training a perceptron for a given target means detecting which inputs are correlated with the target, in what way they are correlated, to what degree they are correlated; and setting a weight appropriately.

The lookup table uses a sequence of past values as input. A hash of the particular value sequence chooses a particular counter which makes the prediction. In contrast to the perceptron, which considers the effect of each input value independently, the table considers the effect of each combination of values. What happens when a perceptron is directly substituted for the lookup table and is asked to learn the effects of combinations of values?

3.1.1 Perceptron Context-based Prediction

Recall how prediction is performed in the two-level table-based branch predictor. A sequence of past branch results is concatenated to form an index, which

chooses a counter from a pattern table. The prediction is then made from the counter state. While this form of prediction is capable of learning correlations between branches, it is not overtly designed to pick out correlations. Instead, it picks out sequences of branches. It learns that a series of particular branch results is always followed by a particular branch result. This form of prediction can be loosely classified as "context-based prediction", because it uses the context of a specific pattern of branch results to determine the result of the next branch.

Context-based predictors are greatly dependent on the quantity of data points forming their context pattern. A basic first-order context-based predictor learns pairs of data values: value "a" is always followed by value "b", value "c" by value "d". A second-order context-based predictor learns triplets of data values: value sequence "ab" is always followed by "c". The order of the context-based predictor is based on its history size.

Just like table based predictors are designed for context-based learning but can learn individual correlations, perceptrons, while being designed for correlational learning, can pick up some context patterns. There are two important limitations, however: 1) linearly inseparable patterns will be ignored, and 2) all the patterns will need to occur with equal frequency. If the set of patterns conflict with each other, the perceptron will be typically unable to learn all of the patterns in that set. If any pattern occurs significantly more often than another pattern, it can bias the perceptron and prevent it from predicting the less common pattern correctly.

What does it mean for two patterns to conflict? Recall that a perceptron learns by seeking a correlation, either direct or inverse, between each input and the target. If

two patterns contradict each other on an input, one pattern producing a direct correlation for the input, and the other producing an inverse correlation for the input, the two patterns cancel each other out, driving that input's weight to zero. This is acceptable, provided that another input that can be used to predict is not cancelled out. However, if the patterns contradict each other on every input, the perceptron cannot learn both patterns. Thus a set of patterns can be in conflict and cannot be learned by a perceptron. For a set of patterns to be compatible, there has to be at least one input that has the same correlation for every pattern in the set. If there are no inputs that have the same correlation throughout the set, the set of patterns is not compatible.

For example, consider that a third-order perceptron context-based predictor is taught that the sequence 101 is always followed by 1. The perceptron will train its weights accordingly: the first weight will learn a direct correlation, the second an inverse, and the third a direct. Next suppose that the predictor is taught 001 is followed by 0. The perceptron will train its first and second weights to learn a direct correlation, and the third an inverse. There is a conflict on the second input and third inputs; however, because there is no conflict on the first input, the perceptron can learn both patterns. However, suppose that the perceptron is then taught that the sequence 100 is followed by 0. In this case, the perceptron trains the first weight to learn an inverse correlation, and the second and third weights to learn a direct correlation. This 100 pattern conflicts with the 101 pattern on the first and second inputs, and with the 001 pattern on the third input. The three patterns are thus in

conflict with each other, and the perceptron cannot learn them (although it could learn any two of the three patterns).

Compatible Patterns				
0	1	0	1	1
1	0	0	1	0
0	1	1	0	1
inverse	direct	conflict	conflict	
Conflicting Patterns				
0	1	0	1	1
1	0	0	1	0
0	1	1	0	0
conflict	conflict	conflict	conflict	

Figure 3.1. Compatible patterns and conflicting patterns

So what happens when a perceptron context-based predictor is taught conflicting patterns? If there are two patterns in conflict, and they occur equally often, all the perceptron weights will cancelled to zero. The perceptron will thus predict arbitrarily. However, if there are three or more patterns in conflict, some of the perceptron weights may not cancel to zero. When this happens, the perceptron may predict arbitrarily. It may nevertheless learn the patterns.

In the above example, suppose that 101-1, 001-0, and 100-0 occur equally often. For each input, two of the three patterns will bias the weight. Figure 3.2

shows what happens when a perceptron is fed these patterns. Notice that while the perceptron does not precisely learn any input, the dominant two weights at any point happen to force the correct answer to occur for all three inputs. It is thus possible for a perceptron to consistently predict correctly on a conflicted pattern.

Why can a perceptron learn a conflicted pattern? The reason is because pattern compatibility is not exactly the same as linear separability. Recall the definition of linear inseparability given in 2.1.3. The perceptron may learn a set of inputs if the positive cases and negative cases can be separated by a straight line (or plane for three dimensions). The three patterns in Figure 3.2 can be separated by a straight plane when plotted by their input variables; however, the plane is a diagonal plane. My above definition of conflict requires that the cases be separated by a single variable; to be compatible, the patterns must be separated by a horizontal or vertical line or plane. Consequently, not all conflicted patterns are linearly inseparable. However, all compatible patterns are linearly separable.

Iteration	Pattern				Weights				Output	Correct?
0	1	0	1	1	0	0	0		0/0	N
1	0	0	1	0	1	-1	1		1/1	N
2	1	0	0	0	2	0	0		2/1	N
3	1	0	1	1	1	1	1		2/1	Y
4	0	0	1	0	2	0	2		0/0	Y
5	1	0	0	0	3	1	1		1/1	N
6	1	0	1	1	2	2	2		2/1	Y
7	0	0	1	0	3	1	3		-1/0	Y
8	1	0	0	0	4	2	2		0/0	Y
9	1	0	1	1	3	3	3		6/1	Y
10	0	0	1	0	4	2	4		2/0	Y
11	1	0	0	0	5	3	3		-1/0	Y

Figure 3.2. Learning incompatible patterns

Figure 3.3 shows the chance that *p* randomly chosen patterns are in conflict for a 16 input perceptron. For this study, I run 1000 tests for each value of *p* from 1 to 16. In each test, *p* random 16-bit input patterns and *p* random target bits were generated. Conflict was determined by checking whether each input for each pattern follows either a direct or inverse relationship with the output. If no bits are learnable, the patterns are considered to be in conflict. The figure shows the average chance that *p* patterns are in conflict over 1000 iterations of *p* randomly selected patterns. For 16 inputs, 5 patterns can be learned over 50% of the time.

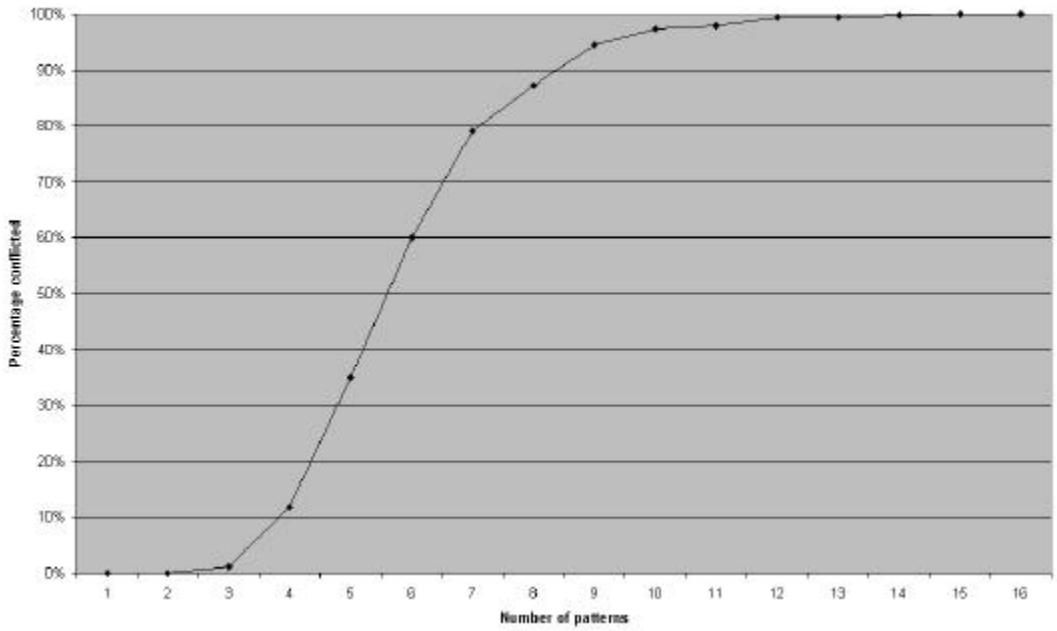


Figure 3.3. Chance that *p* patterns will be in conflict

Figure 3.4 shows the chance that *p* randomly chosen patterns are unlearnable for a 16 input perceptron. I ran 1000 tests for each value of *p*, and in each test, *p* random patterns and target bits were created. A perceptron using training-by-correlation is given 1000 iterations to learn the *p* patterns. If it gets every value correct for 2*p* iterations, the patterns are said to be learnable by this perceptron. If after 1000 iterations the perceptron has not learned the pattern, they are said to be unlearnable by the perceptron.

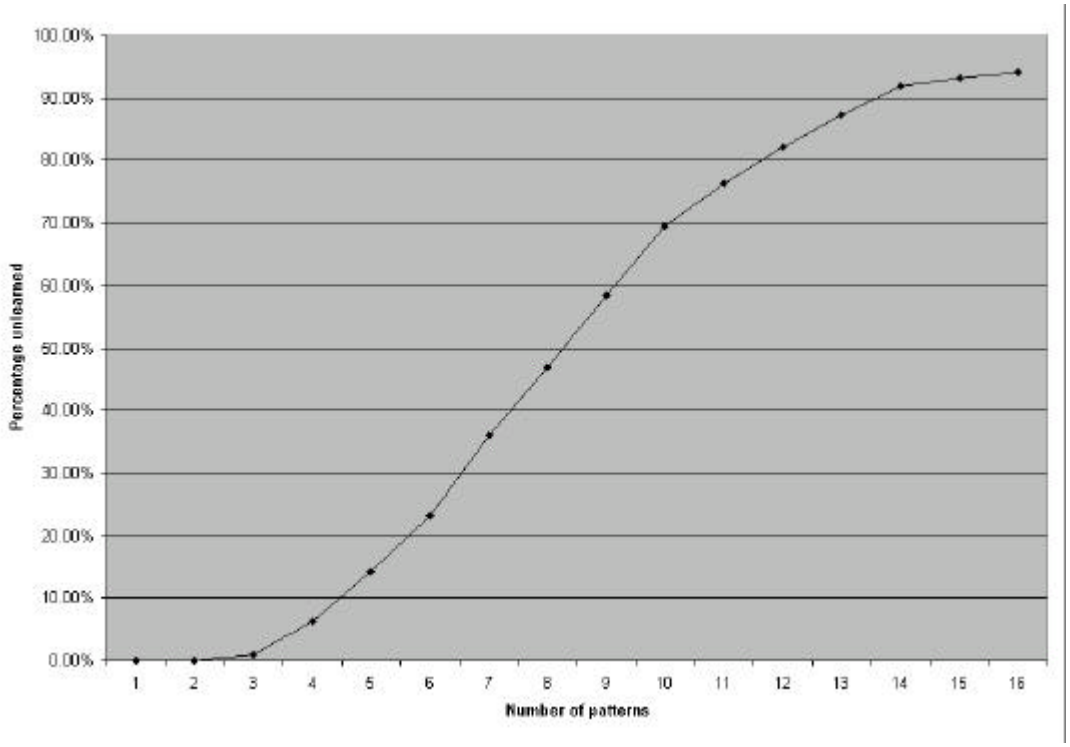


Figure 3.4. Percentage of the time the perceptron cannot learn *p* patterns

Even if a set of patterns is compatible, the perceptron may nevertheless be unable to learn it if some patterns occur more often than other patterns. When this happens, a dominant pattern can bias the perceptron weights. When the less common pattern occurs, even though the weights are able to represent the pattern, they are unable to set the threshold.

An example of this is shown in Figure 3.5. Suppose that pattern 101-1 occurs four times, and pattern 001-0 occurs once. These two patterns are compatible: the first weight learns a direct correlation in both cases, even though the other two weights do conflict. However, the more common 101-1 pattern biases the weights. When the less common 001-0 pattern occurs, the sum total is positive, even though the second and third weights are weaker than the first. The perceptron would

consequently predict incorrectly on a subsequent 001-0 pattern, even though both patterns are compatible.

Patterns				Weights				Output	Correct?
1	0	1	1	0	0	0		0/0	N
1	0	1	1	1	-1	1		3/1	Y
1	0	1	1	2	-2	2		6/1	Y
1	0	1	1	3	-3	3		9/1	Y
0	0	1	0	4	-4	4		4/1	N
0	0	1	0	5	-3	3		1/1	N

Figure 3.5. Effect of imbalance on learning

There are several factors that affect the severity of this biasing problem. Among these are: 1) the degree of imbalance, 2) the amount of training, 3) the degree of conflict between the patterns, and 4) the training strategy.

It is easy to see why the degree of imbalance affects the problem. If the 101-1 pattern occurred slightly more often than the 001-0 pattern, the second and third weights would remain small due to the conflict, while the first weight would grow large. Although the second and third weights would not be precisely zero because the 101-1 pattern occurs more often, the magnitude of the first weight would overcome them and determine the perceptron output.

If the perceptron patterns are greatly imbalanced, the imbalance could mean that the perceptron never actually learns the patterns, regardless of the fact that the patterns are compatible. This is shown in Figure 3.6 by extending the sequence shown in Figure 3.5. Notice that the difference gap between the magnitude of the

first weight and the combined second and third weights grows with each training iteration, meaning that the perceptron can never learn the pattern.

Iteration	Pattern			Weights				Output	Correct?
0	1	0	1	1	0	0	0	0/0	N
1	1	0	1	1	1	-1	1	3/1	Y
2	1	0	1	1	2	-2	2	6/1	Y
3	1	0	1	1	3	-3	3	9/1	Y
4	0	0	1	0	4	-4	4	4/1	N
5	1	0	1	1	5	-3	3	11/1	Y
6	1	0	1	1	6	-4	4	14/1	Y
7	1	0	1	1	7	-5	5	17/1	Y
8	1	0	1	1	8	-6	6	20/1	Y
9	0	0	1	0	9	-7	7	5/1	N
10	1	0	1	1	10	-6	6	22/1	Y
11	1	0	1	1	11	-7	7	25/1	Y
12	1	0	1	1	12	-8	8	28/1	Y
13	1	0	1	1	13	-9	9	31/1	Y
14	0	0	1	0	14	-10	10	6/1	N

Figure 3.6. Unlearnable patterns due to imbalance

The imbalance problem is exacerbated by the number of conflicting weights between the patterns. Even though the 101-1 and 001-0 patterns are compatible due to the first weight, only one of the three weights is not in conflict. Suppose that the perceptron were asked to learn the patterns 101-1 and 011-0, where two of the three

weights are not in conflict. Figure 3.7 shows the result with the same imbalance as before. Observe that the two conflict-free weights are able to easily overcome the unbalanced conflicted third weight. Pattern imbalance only affects conflicted weights. The more conflict-free weights that exist between the patterns, the more imbalance the perceptron can handle.

Iteration	Pattern			Weights				Output	Correct?
0	1	0	1	1	0	0	0	0/0	N
1	1	0	1	1	1	-1	1	3/1	Y
2	1	0	1	1	2	-2	2	6/1	Y
3	1	0	1	1	3	-3	3	9/1	Y
4	0	1	1	0	4	-4	4	-4/0	Y
5	1	0	1	1	5	-5	3	13/1	Y
6	1	0	1	1	6	-6	4	16/1	Y
7	1	0	1	1	7	-7	5	19/1	Y
8	1	0	1	1	8	-8	6	22/1	Y
9	0	1	1	0	9	-9	7	-11/1	Y

Figure 3.7 Imbalanced patterns are learnable with sufficient compatible inputs

The construction of the perceptron can have a great deal to do with the amount of pattern imbalance it can handle. The training strategy used in the above example handles pattern imbalance very poorly by allowing conflicted weights to grow away from zero. As will be discussed later, alternative training mechanisms are able to reduce the effects of pattern imbalance.

In the following study I try to quantify the effect of pattern imbalance on learning time. I implemented a perceptron in C with *n* inputs, where *n* is fixed at 16. I choose *p* compatible patterns and target values in such a way that *c* of the *n* inputs are in conflict. This is done as follows. First, *p* target values are chosen randomly. Second, *n-c* nonconflicted correlation directions are chosen at random for the first *n-c* inputs. Because *n-c* is greater than 0, the patterns are guaranteed to be compatible. The first *n-c* bits are then chosen for each pattern based on these correlations. Third, for each of *c* remaining bits, *p* bit values are randomly chosen. If these values are not in conflict, they are repeatedly discarded and chosen again.

The balance *b* between the patterns is quantified as the ratio between how often the last pattern is supplied to the inputs versus how often the first *p*-1 patterns are. The last pattern is replicated *b*-1 times to form the complete pattern set.

The perceptron is repeatedly supplied these inputs and trained using the training-by-correlation strategy discussed later. The perceptron is considered to have learned the *p* patterns when it predicts the correct value every iteration for 2*(*b*+(*p*-1)) iterations (in other words, it predicts the correct output for every input pattern twice in a row). The training time for the perceptron to learn these patterns is computed as the average of the number of iterations needed to learn minus *b*+(*p*-1), for 1000 tests with different randomly generated input patterns. A pattern is considered unlearnable if it is not trained after 1000 training iterations.

Figures 3.8, 3.9, and 3.10 show the percentage of the patterns that were learnable as a function of *b* and the number of conflicted inputs *c* for *p*=4, 8, and 16. Figures 3.11, 3.12, and 3.13 show the training times for those patterns that were

learnable. As *b* increases, the percentage of unlearnable patterns increases. Interestingly, however, the training time for those patterns that are learnable is not affected by the balance. Notice that balance is never a problem if the percentage of conflicted inputs is under 50%.

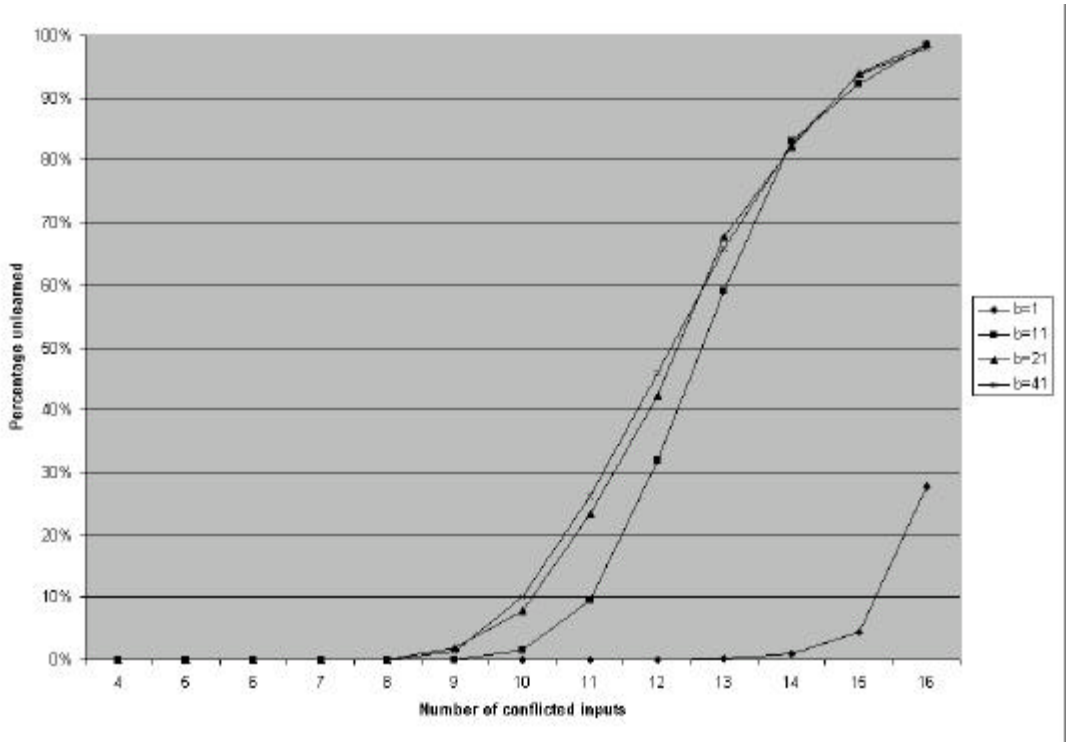


Figure 3.8: Learnability for 4 patterns

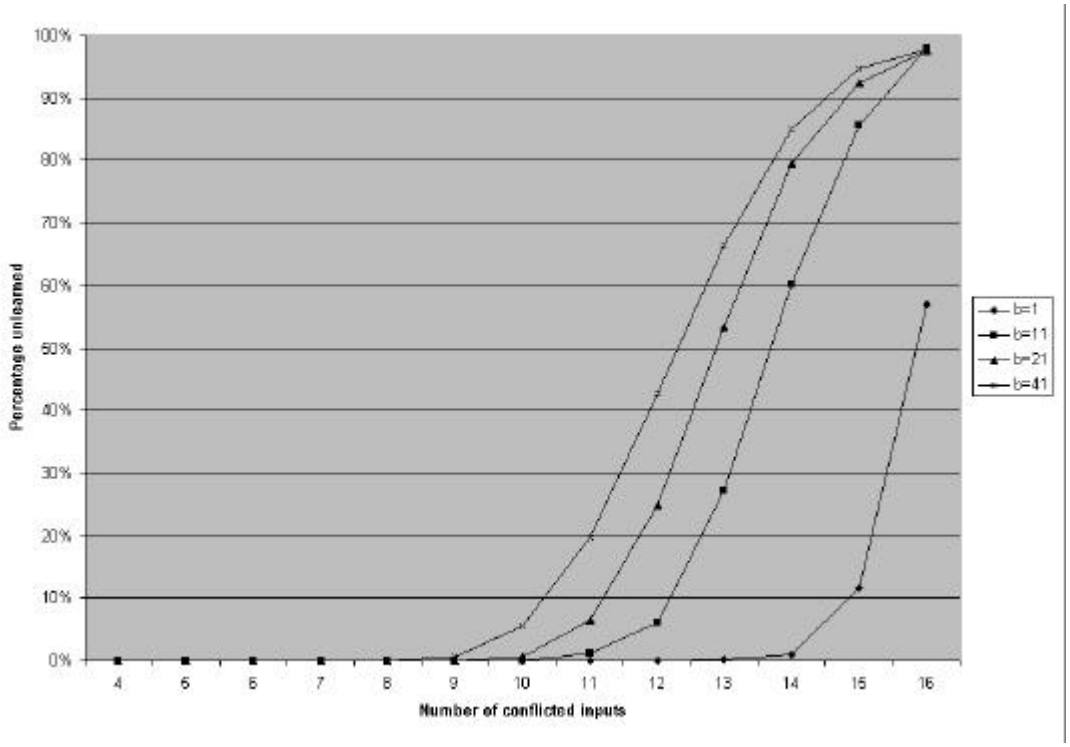


Figure 3.9: Learnability for 8 patterns

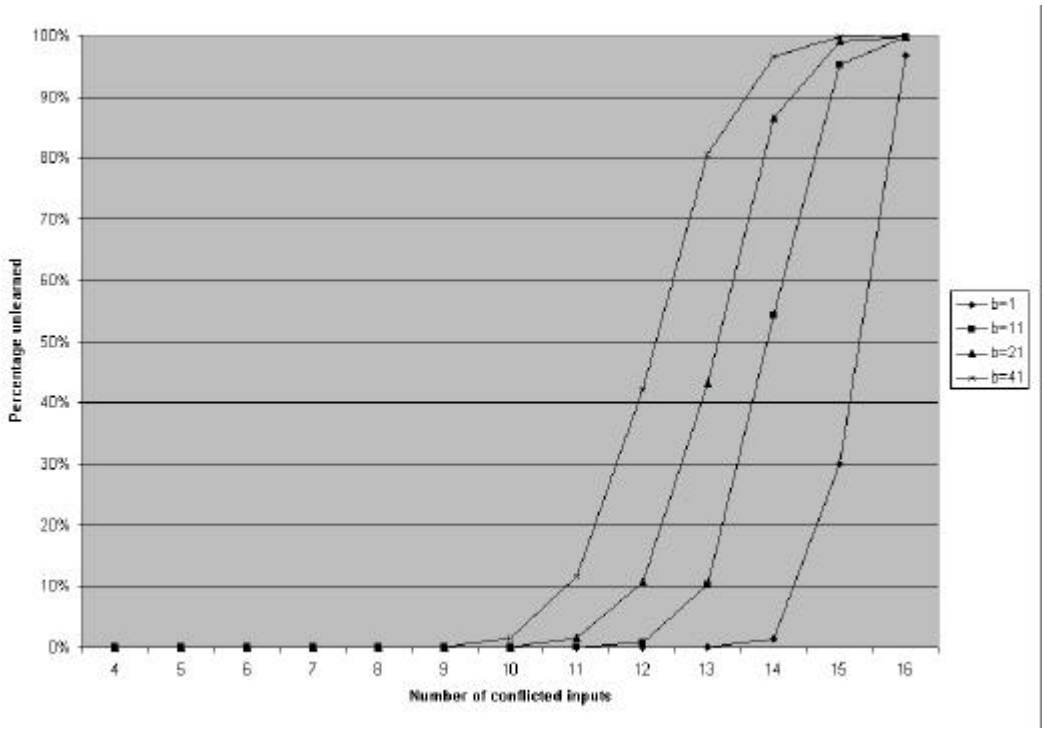


Figure 3.10: Learnability for 16 patterns

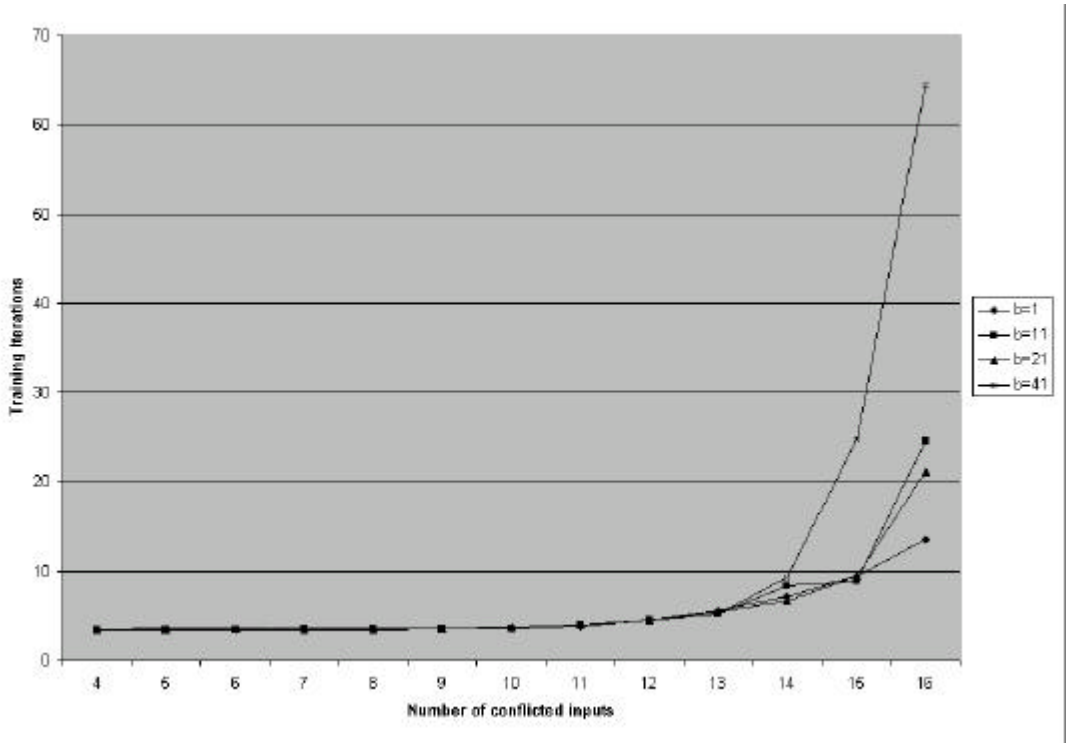


Figure 3.11: Training time for 4 patterns

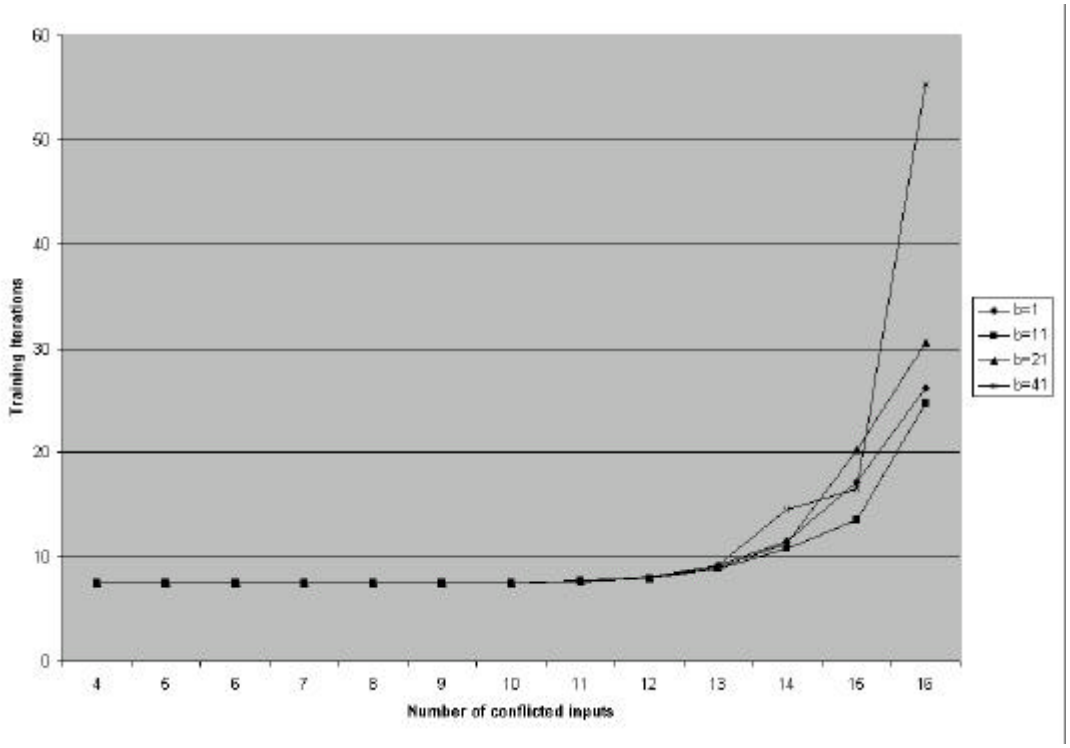


Figure 3.12: Training time for 8 patterns

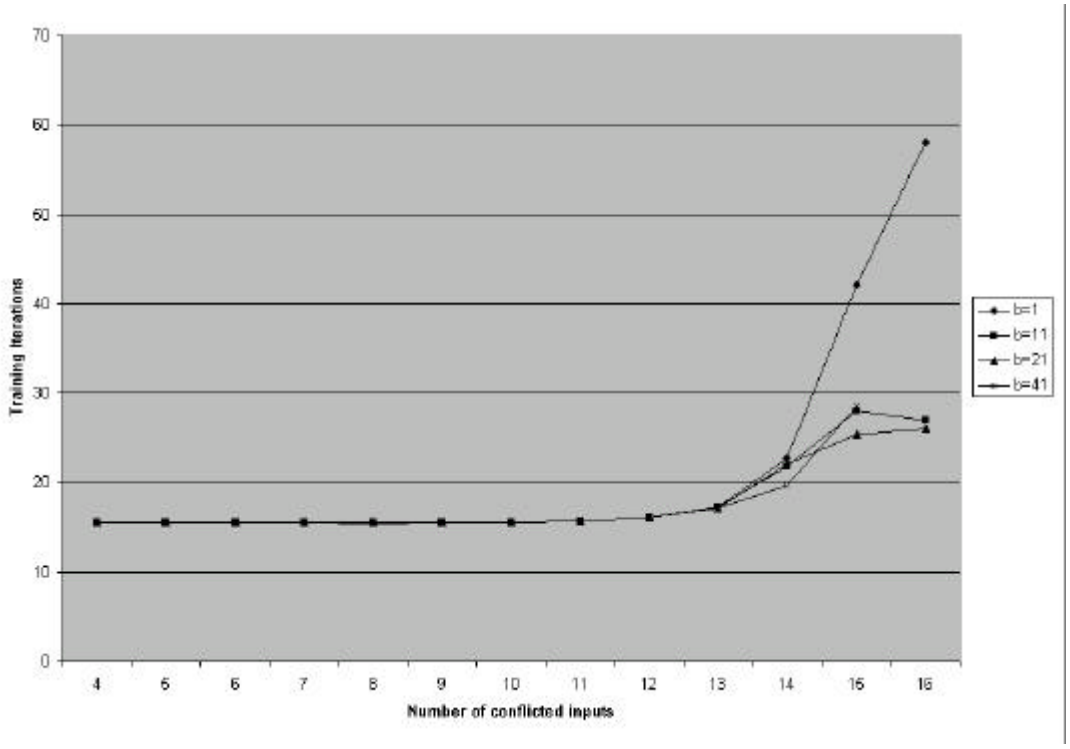


Figure 3.13: Training time for 16 patterns

3.1.2 Can a Perceptron Outperform a Table?

The perceptron's primary method of learning, as discussed earlier, is correlational, capturing the relationship between each input and the output. The table, on the other hand, learns from a context, with the combination of branch results, combined, choosing the pattern table entry. The question, consequently, is whether the table learns correlations as well as a perceptron? If so, a table with equal history size would always perform at least as well as a perceptron. If not, then there are some input sets (those best learned correlationally) for which a perceptron would outperform a table.

Recall that while a perceptron can learn all single input correlational input sets, there are some context-based input sets that a perceptron cannot learn - those pattern sets previously defined as incompatible. A table on the other hand can learn correlational inputs, as well as all context-based inputs. This is because a set of correlations can be mapped to a set of contexts, as shown in Figure 3.14's example. However, a table cannot necessarily learn the correlational inputs in as few iterations as a perceptron can. As mentioned above, the perceptron can infer the negative case of a correlation from the positive case, and vice versa. The lookup table, however, must observe both cases before they can be learned. An example of this is shown in Figure 3.15; notice that the perceptron can learn the patterns in 1 iteration, while the table takes 2 iterations.

i	d	d	i	
1	0	0	1	0
0	1	1	0	1

Figure 3.14. Correlation converted to context patterns

Iteration	Pattern					Perceptron output	Table output
0	1	0	0	1	0	0	0
1	0	1	1	0	1	1	0
2	1	0	0	1	0	0	0
3	0	1	1	0	1	1	1

Figure 3.15. A perceptron can learn faster than a table

This difference between the rate of perceptron learning versus the rate of table-based learning becomes more severe with larger input sets containing greater quantities of correlated inputs. The perceptron learns a correlation between an input and the target independently of the other inputs. The table approach cannot learn an input correlation independently of the other inputs. To learn the same correlation, the table, on the other hand, must observe the input's negative and positive case for all

possible values of the other input. If there are *e* correlated inputs, the perceptron can learn all the possible correlations in as few as *e* iterations. The table, however, requires a minimum of 2^{inputs} iterations to learn the correlation, assuming the remaining uncorrelated inputs are random (or noise). This is because each combination of noisy inputs maps to a different table entry. Before producing reliable outputs, the table must observe every possible combination of noisy inputs. Consequently, tables learn correlations significantly more slowly if one or more inputs are both uncorrelated with the target and random.

The table performs better when only a few different patterns are referenced, as the table can be trained on a few patterns quickly. There are consequently two cases when a table can learn a single input correlation rapidly. The first case is when the other inputs are also correlated with the target. As correlation is transitive, two inputs correlated with the target are also correlated with each other. Thus the two inputs will always have the same value relative to each other. The table will consequently not need to observe all combinations of the two inputs, as the inputs together will never form more than two patterns. The second case is when the other inputs always keep the same values from one iteration to the next. If a two input history has one correlated input and one constant input, there will likewise be only two patterns to be learned, as the constant input never changes.

It is interesting to note that while both the table and the perceptron can mask uncorrelated inputs, the behavior of a masked input is different for a table and a perceptron. A perceptron ignores an input if it is uncorrelated with the target. The table ignores an input if it maintains a constant, unchanging value. As showed above,

a table is unable to cope with an uncorrelated input that is not constant. For reasons to be discussed later, a perceptron likewise can be stymied by false correlations, where a constant input appears to be correlated with the target.

3.1.3. Perceptron Learning Beyond Context

3.1.3.1. Masking

As part of learning which inputs are correlated with a target, a perceptron also learns which inputs are not correlated with the target. These inputs are assigned weight values of zero, or near zero, and are consequently inhibited from affecting the perceptron decision. The ability of the perceptron thus to mask uncorrelated inputs has greatly contributed to its success in branch prediction. In table-based branch prediction, these uncorrelated branches create substantial wasted table space and make learning slower, as discussed above. The only wasted space that uncorrelated inputs cause in a perceptron are their weight bits, and the only slowdown in learning is the time needed for the perceptron to learn which inputs are uncorrelated. This masking of uncorrelated inputs allows the correlated inputs to have a greater effect on the actual prediction.

Previous perceptron implementations in computer architecture have focused on using perceptrons to detect correlations, with uncorrelated input masking being a pleasant side effect. However, a perceptron could be instead used exclusively for classifying inputs as correlated and uncorrelated. This is shown in Figure 3.16. The perceptron determines which inputs contain useful information and which inputs

contain irrelevant information. The perceptron is piggy-backed on a table-based predictor that uses only the useful inputs to hash its table and make predictions.

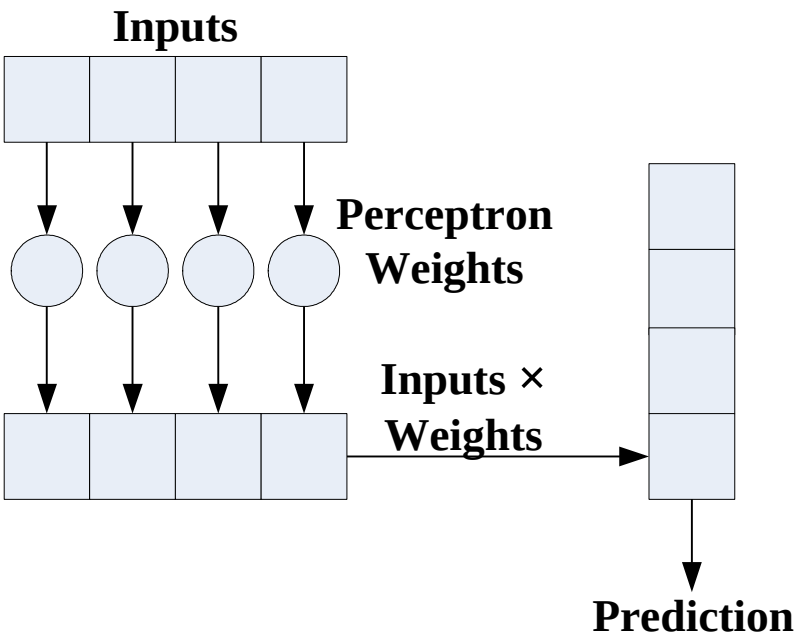


Figure 3.16. Perceptron masks uncorrelated inputs for a lookup table

It should be clear why a masking perceptron can be generally useful for speculation in architecture. Computer architecture speculative applications typically have large quantities of past data available, of which only part of it contains useful patterns. As will be seen later with value prediction, promising prediction strategies are often impractical due to their inabilities to cope with massive amounts of irrelevant past data.

It is easy, however, to abuse masking perceptrons. This can be illustrated in the following example. Suppose that a novice architect were trying to design a branch predictor that uses other data besides past branches to make a prediction, on the premise that branches could be correlated with data values and other readily available information. However, the architect does not know which types of past data

is useful and which is not. The architect might take a huge perceptron, and feed it hundreds of past data values, load/store target addresses, processor state information, and even the time of day, assuming that the perceptron will somehow "sort it all out." Unfortunately, the perceptron probably will not.

There are several reasons for this. First, perceptrons can learn false patterns. Suppose that a loop branch is taken (a perceptron output of "1") 100 times, and not taken on the 101st iteration. Suppose also that an irrelevant data point, always "1", is sourced to one of the perceptron inputs. The perceptron will falsely learn a direct correlation for that input very well over 100 iterations. That correlation will fail on the 101st iteration. However, during those first 100 iterations, the perceptron will incorrectly identify the input as correlated. Second, increasing the number of inputs of a perceptron also increases the noise from uncorrelated inputs, lowering the perceptron accuracy. If it is clear that a past value will be uncorrelated, it does no good to the perceptron to include it.

There is a more subtle problem in piggy-backing a masking perceptron on top of another predictor. Suppose that a table-based branch predictor is used instead of a perceptron because a table-based predictor is able to learn linearly inseparable patterns between past branches. However, a perceptron is used to weed out uncorrelated branches before they are used in the table hash value. This would appear to be the best of both worlds: the table would supposedly capture a large variety of branch patterns, while remaining small because only correlated branches are used in the index. The problem is that, to a perceptron, two inputs that, while themselves not individually correlated with the target, together form a linearly

inseparable pattern, are indistinguishable from two uncorrelated inputs. These inputs would be assigned zero weights and would be weeded out before reaching the table. Thus while the table could predict using those past branches, the masking perceptron would prevent those branches from ever reaching the table.

3.1.3.2. Recognizing new patterns

If there is one thing neural networks are known for, it is their ability to learn a generalization from a limited set of examples, and apply their generalization to new input patterns. In many other neural network applications, the network weights are set through repeated application of a training set of patterns. After the neural network is trained, it is given actual patterns, which may or may not have been part of the training set. Having learned a generalized function from the training set, the network is able to produce correct outputs from these previously unseen patterns. The perceptron, being a small neural network, is also able to learn certain generalizations from training and apply them to new patterns. However, for reasons discussed below, this ability is not likely to be very useful in computer architecture applications.

Recall that the basic function of a perceptron is to learn individual correlations between many binary inputs and a binary target. Each perceptron weight reflects the correlation learned for the respective input, with a positive weight meaning a direct correlation, a negative meaning an inverse, and a zero meaning no correlation observed. Depending on how the perceptron is used, a previously unseen input could mean one of two things: it could be a change in a specific perceptron input, such as a branch not being taken that had always previously been taken, or it could be a new pattern of input values together correlating with a new output value. Unless the

perceptron is untrained, it will produce some output value for the new input. However, whether that output value is useful or not depends on the application.

Suppose that a perceptron is used to determine correlations between individual inputs (maybe different past branches), and the target (the branch to be predicted). The weights thus reflect how each individual past branch is correlated with the target. Suppose that a particular past branch always produced an input of "taken" (1) when the target branch was taken (1), and a direct correlation was learned. If the branch produces the previously unseen value of not taken (0), the perceptron will assume that this means that the target branch should not be taken. In this way, the perceptron has been able to determine an output from a previously unseen input value, extrapolating on the generalization learned: that the input correlates directly with the output. However, this may not necessarily be a correct generalization. The input might have been uncorrelated with the target branch, but both might have been taken most of the time. A direct correlation might have been observed and well learned, but prove useless in making predictions. Consequently, for an application to be able to use a perceptron to predict for previously unseen input values, the new values must follow the same correlations observed for the past values.

Suppose that a perceptron is instead used as a context-based predictor, determining an output value from the pattern of input values. The weights reflect how an output should be chosen from the set of patterns. In this case, a previously unseen input pattern requires the perceptron to apply a previously made generalization to a new input. However, the generalization that the perceptron learned is simply the emphasizing of nonconflicted inputs between the pattern, and

the ignoring of conflicted inputs. The ability of the perceptron to guess the new pattern depends entirely on how much it conflicts with other patterns. If the majority of the inputs in the pattern do not conflict with other patterns, the perceptron will likely correctly guess the output. If most of the pattern inputs conflict with other patterns, the perceptron will most likely guess incorrectly. Consequently, in order for an application to be able to use a perceptron to predict for new patterns, the new patterns must conflict minimally with the old patterns.

3.2. Training

A perceptron's training approach greatly determines not only the speed at which it can learn a particular set of input values, but whether it can learn those input values at all. Interestingly enough, prior perceptron work in computer architecture have used two different perceptron training mechanisms almost interchangeably. While both of these mechanisms have the same effect of teaching a perceptron to learn direct and inverse correlations, the actual effects the two mechanisms have on the perceptron weights are drastically different. As we shall see, the two training mechanisms both have good points and bad points, and there are definite reasons in most applications to use one instead of the other.

The main objective of training a perceptron is to adjust each weight so that it reflects the correlation between the corresponding input and the target, and is able to influence the perceptron output appropriately. A weight should tell whether there is a correlation (by whether it is zero or nonzero), what type of correlation it is (by the sign: positive if direct, negative if inverse), and how strong the correlation is (by the magnitude). The weights should be adjusted so that inputs for which the perceptron

is confident about the correlation have a strong effect on the perceptron result, while inputs for which the perceptron is not-confident about the correlation should have a negligible effect on the result.

3.2.1. Training Issues

There are several issues when designing a training strategy for computer architecture applications. Might many noncorrelated weights together override a correlated weight? Can a pattern that is not a correlation be mistaken for one? Will weight patterns be quickly unlearned on a context switch? How susceptible is the predictor to biasing from pattern imbalance? How many training iterations are needed? Choices made for the above issues should suit the application and its data patterns.

Weights associated with a noncorrelated input do not necessarily have a value of exactly zero. A noncorrelated input produces arbitrary (or noisy) values that cause its weight to fluctuate continually. Such a weight may have a value of zero or a value close to zero, depending on the iteration. Clearly, in the presence of a large magnitude weight reflecting a strong correlation, these noncorrelated weights have little influence on the result. However, if there are sufficiently more noncorrelated weights than correlated weights, the noise from the noncorrelated weights could drive the result. An example of this is shown in Figure 3.17. The correlated weight, 5, is overruled by the uncorrelated weights. Left unsolved, this problem creates an upper bound to the perceptron input size. With too many inputs, and too few of them correlated, the uncorrelated weights tend to dominate the output.

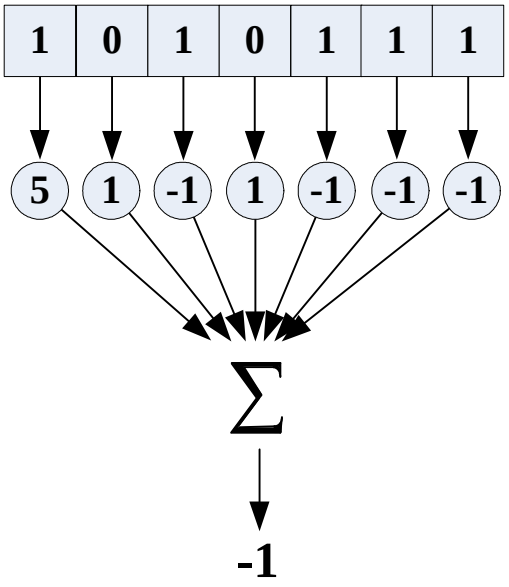


Figure 3.17. Uncorrelated weight noise can bias a perceptron

A perceptron's weights can sometimes learn correlations where none exist from heavily biased inputs. This occurs when both the target and a particular input tend to have one value occur much more frequently than the other value. Suppose, for example, the perceptron target value is typically 1, and seldom 0, and another perceptron input is also virtually always 1 as well. However, they are uncorrelated, because the target does not produce 0 when the input produces 0. The perceptron might nevertheless observe a strong positive correlation, not because one exists, but because the input and target are so often 1 at the same time. If the corresponding weight is allowed to grow large, a 0 at that input will strongly influence the perceptron output to 0, producing an incorrect output. This problem is very common in branch prediction, where both the target branch and a past input branch may control iteration in a loop, and will both consequently be taken most of the time.

Another problem particular to computer architecture applications is context switches, when one phase of a program ends and another begins. When this happens,

previous patterns may no longer occur, and new patterns must be learned. A perceptron undergoing a context switch will need to unlearn the past patterns before it can reliably learn the new patterns. If the weights are allowed to grow too large, the perceptron will need to spend many iterations reducing the large weights before the weight can be adjusted to learn the new pattern.

As mentioned earlier, perceptrons being used as context predictors can suffer from pattern imbalance, when one data pattern occurs much more frequently than another data pattern. This requires more training iterations to learn the pattern set. This can be countered by preventing perceptron weights from growing excessively large on majority patterns so that they are able to learn the infrequent patterns.

Training time is crucial in many computer architecture applications. Unlike other applications of neural networks where the networks are subjected to thousands of training iterations before being required to make accurate predictions, perceptrons must make correct predictions after only a few training iterations. There are two reasons for this. First, many predictable patterns in programs only occur a few times, not thousands of times; if a perceptron is not trained rapidly enough, it could miss the pattern entirely. Second, since a perceptron in architecture applications is being used while it is being trained, it needs to make accurate predictions almost immediately. It should be noted that table-based predictors, the alternative to perceptrons, do not typically require thousands of iterations to predict accurately.

It should be observed that the above problems require conflicting solutions. Uncorrelated weight noise can be countered by making the correlated weights grow very large. Large weights, however, make training take longer, context switch

retraining take longer, and pattern imbalance more severe. If weights are allowed to grow large because of heavily biased inputs, the false correlation problem becomes more severe. Thus, when determining how large the weights can be allowed to grow, the percentage of uncorrelated inputs must be weighed against frequency of pattern imbalance and biased inputs.

3.2.2. Training using an error value

In [Akk04], the perceptron weights were trained using an error value. An error is computed on each training iteration by subtracting the predicted output from the desired output (e=a-p). This error is then applied to the weights by multiplying it by each weight's corresponding input and adding it to the weight. It should be pointed out that in order for this approach to work correctly, an input of 0 should be treated as -1 so that the error is added negatively on 0 inputs. However, when the error is calculated, a prediction of 0 should be treated as 0 so that the error is always 0, -1, or 1.

A variation on this training approach is to multiply the error value by some integer constant alpha [Ros62]. A larger training factor causes the weights to grow faster in the same number of iterations, and makes for larger weight values.

The interesting characteristic of this error value based training approach is that the weights are adjusted until correct output values are obtained, and then training is stopped when the error value becomes zero. Subsequent training only occurs when an incorrect prediction is made. Assuming that the perceptron starts producing correct outputs after a few training iterations, no weight will ever become very large. This is, of course, just fine if the perceptron fully learned the correlations in those

iterations. However, the perceptron might have only learned enough of the correlation set to predict correctly most of the time.

The basic positive side to this approach is that it is focused on the final goal of having the perceptron produce the correct result. If the perceptron is already predicting correctly, why change the weights and disturb it?

The negative side to this approach is that it needs incorrect predictions to drive it. Because weights are not changed on correct predictions, training can only occur on incorrect predictions. The result is that all training is in response to perceptron mispredictions. An alternative is preventative training. In preventative training, even though the perceptron is predicting correct outputs, the more well correlated weights are strengthened further. In future predictions when more weakly correlated inputs produce unreliable values, the perceptron will have learned to identify the more strongly correlated inputs, and can rely on them without suffering a misprediction. With error based training, however, the perceptron cannot identify the less reliably correlated inputs until they force it to mispredict.

This error-based training approach is very susceptible to uncorrelated weight noise because correlated weights are not permitted to grow past the point where predictions become correct most of the time. The correlated weights may rise above the typical noise level fairly quickly, because the perceptron will initially be producing incorrect outputs. However, they will stop rising after that, leaving the perceptron susceptible to bursts of noise. It is true that on each burst of noise sufficient to cause an incorrect prediction, the correlated weights will be trained. But the incorrect prediction nevertheless occurred. Had the correlated weights been

trained further even after correct predictions were produced, they would have been resilient to the noise burst.

Because the weights do not grow large, this training approach makes the perceptron less susceptible to being biased by false correlations. An example of this is shown in Figure 3.18. Although the perceptron is initially misled by input 2's false correlation and sets weight 2 equal to weight 1, the misprediction on iteration 3 reduces weight 2. Because weight 2 was never permitted to grow large, this reduction greatly reduces input 2's influence relative to input 1. On the other hand, the perceptron had to actually mispredict for input 2's false correlation to be observed. Had input 2 not caused a misprediction in iteration 3, its weight would not have been reduced.

Iteration	input 0	input 1	input 2		Weights			Output	Correct?
0	0	1	1	1	0	0	0	0/0	N
1	1	1	1	1	-1	1	1	1/1	Y
2	1	0	1	0	-1	1	1	-1/0	Y
3	0	0	1	0	-1	1	1	1/1	N
4	0	0	1	0	0	2	0	-2/0	Y

Figure 3.18. Training-by-error's handling of false correlations

An even greater advantage is that this training approach always eventually learns any set of compatible patterns, no matter how imbalanced they are or whether false correlations are present. Recall the example in Figure 3.2 of the imbalanced pattern set that the perceptron did not learn. Figure 3.19 shows the learning process again with training-by-error. Because the biased weights from an imbalanced input do not keep growing after a correct pattern is obtained, the other weights are able to

catch up when the minority pattern occurs. As will be shown below, this training approach is always able to learn compatible pattern sets.

Iter-ation	input 0	input 1	input 2	Weights				Output	Correct?
0	1	0	1	1	0	0	0	0/0	N
1	1	0	1	1	1	-1	1	3/1	Y
2	1	0	1	1	1	-1	1	3/1	Y
3	1	0	1	1	1	-1	1	3/1	Y
4	0	0	1	0	1	-1	1	1/1	N
5	1	0	1	1	2	0	0	2/1	Y
6	1	0	1	1	2	0	0	2/1	Y
7	1	0	1	1	2	0	0	2/1	Y
8	1	0	1	1	2	0	0	2/1	Y
9	0	0	1	0	2	0	0	-2/0	Y
10	1	0	1	1	2	0	0	2/1	Y
11	1	0	1	1	2	0	0	2/1	Y
12	1	0	1	1	2	0	0	2/1	Y
13	1	0	1	1	2	0	0	2/1	Y
14	0	0	1	0	2	0	0	-2/0	Y

Figure 3.19. Training-by-error can learn the imbalanced pattern

The error value based training approach responds well to context switches. The low weight values mean that weights can be more rapidly unlearned when they need to be changed. Likewise, the low weight values make the approach less susceptible to pattern imbalance, as the majority pattern is unable to heavily bias the weights.

3.2.3. Training using correlations

3.2.3.1. Without training cutoff

An alternative training strategy was used in [Jim00]. No error value is computed from the perceptron's prediction. Instead, the desired value is compared with each input value. If they are equal, the corresponding weight is incremented. If they are not equal, the weight is decremented. The approach effectively works thus: a

correlation is observed between each input and the target. The weights are then adjusted for the correlation; they are made more positive on a direct correlation, and are made more negative on an inverse correlation.

In this training approach, the perceptron weights are always changed, whether the prediction was correct or not. Over many training iterations, a correlated weight can potentially become very large in magnitude. An uncorrelated weight, however, will typically oscillate around zero.

The basic advantage to training using correlations is that new information is always used. Even though the perceptron may already be predicting correctly, training nevertheless continues. Thus, in theory, the weights come to more precisely reflect the correlation between each input and the perceptron target. This training approach effectively performs preventative training, determining which inputs are strongly and weakly correlated even after the perceptron begins predicting correctly, and adjusting those weights accordingly. Additionally, this approach is somewhat simpler from an implementational standpoint, because the perceptron prediction does not need to be remembered in order to train.

The disadvantage is that weights must constantly change, even when the perceptron is predicting correctly. It is thus possible for a perceptron to mess up a good set of weight values. More problematically, this approach allows some weights to grow very large, making biasing and untraining more severe issues.

This training approach is fairly resilient to uncorrelated weight noise. This is because correlated weights are allowed to grow substantially bigger than the uncorrelated weights. However, it is very susceptible to learning false correlations

from heavily biased inputs. The training strategy will repeatedly increment a weight if the input and target values are repeatedly equal, even if they are both always 1.

3.2.3.2. With training cutoff

A variation on training using correlations was used in several past works and has each weight magnitude saturated at a cutoff value (referred to as theta in [Jim02]). This theta is chosen to be big enough so that the perceptron is not susceptible to uncorrelated weight noise, yet small enough so that the biasing and retraining problems of large weights do not cripple the perceptron. In [Jim02], the authors empirically decided that 1.93*inputs+14 is the optimal theta for their perceptron branch prediction approach.

This correlational training with training cutoff approach has both its good and bad sides. On the plus side, it creates a compromise; allowing for preventative training without allowing any weight to become big enough to completely bias the perceptron. On the minus side, it does not really solve any of the problems, while trying to force a single cutoff value on every weight. Regardless of what cutoff value is used, it will tend to be too small for some weights, allowing correlated inputs to be overwhelmed by uncorrelated inputs, and too large for others, allowing falsely correlated inputs to bias the perceptron.

A future area of study could look at dynamically varying the theta for each perceptron in a predictor (or even each weight). A detector could try to determine if the perceptron is being overwhelmed by uncorrelated noise, and raise theta, or if falsely correlated weights are becoming too large, and lower theta.

3.2.4. Exponential weight growth

In all previous perceptron proposals in computer architecture, the weights have always been increased or decreased in training by a constant value (typically 1 or -1). As an alternative approach, I propose to raise or lower the weights by multiplying or dividing them by a factor. This exponential training approach, as opposed to the previous linear training approaches, would allow weights for correlated inputs to quickly rise above the uncorrelated weight noise, while being able to be rapidly untrained. It could be applied to either of the above two training methods.

Exponential weight growth should be particularly useful in countering uncorrelated weight noise. A correlated weight will grow much larger than an uncorrelated weight in few training iterations, and will consequently be more influential than a greater number of combined uncorrelated weights than it would be under linear growth. This is shown in Figure 3.20.

A second advantage is that correlated weights can become large more rapidly than in linear weight growth. This means that fewer iterations are needed to train the perceptron.

A third advantage is that, on a context change, previously correlated large weights can be untrained rapidly. This is also beneficial for countering false correlations in the training by error value approach, as shown in Figure 3.19. When input 3 demonstrates that it is falsely correlated in iteration 2, its weight is not decreased by 1, but cut in half. It consequently becomes significantly less influential than correlated input 2.

Iteration	Pattern			Weights				Output	Correct?
0	0	1	1	1	-2	8	8	14/1	Y
1	1	0	1	0	-2	8	8	-2/0	Y
2	0	0	1	0	-2	8	8	2/1	N
3	0	0	1	0	-1	16	4	-11/0	Y

Figure 3.20. Countering weight noise with exponential growth

On the other hand, there are a couple disadvantages. First, inputs that demonstrate a correlation sooner become significantly larger than inputs that a demonstrate correlation later. This can cause mispredictions if the inputs that became correlated sooner turn out to be less reliable (although, after the misprediction, this is corrected). Second, exponential growth lacks the fine resolution of linear growth. In linear growth, an 8-bit weight can have 256 possible values, whereas in exponential growth, leaving 1 bit over for the sign, it can have only 15 possible values (7 positive values, 7 negative values, and zero). If the perceptron weights need to be finely balanced, with one input being only marginally less significant that another input, training exponential growth will fail.

Exponential growth has implementation advantages. In linear growth, a weight must be incremented or decremented, requiring binary addition or subtraction. In exponential growth, if the growth factor is 2, the weight need only be logically shifted left or right, a less complex operation. The challenge, however, is how to handle the zero case and sign reversal.

The loss of resolution in exponential growth can have an implementation advantage in compressing the size of the weight. Rather than having the weight contents represent the actual weight value in two's complement form, the weight bits

could instead represent a power of 2 (less one weight bit for the sign). This can greatly reduce the overall size of the perceptron, as the weight storage is the largest hardware cost. However, this comes at the cost of requiring extra hardware and latency to decode the weight value.

3.2.5 Comparing Training Strategies

In the following studies, the I quantitatively compare the above training strategies in their ability to deal with biased inputs and their susceptibility to weight noise. The first study repeats the study from section 3.1.1 with p=8 for the error-based training strategy. Figure 3.21 shows the effect on training time, and Figure 3.22 shows the percentage that of the patterns that are learnable for both training approaches. The training time is slightly worse for training-by-error than for training-by-correlations. However, in training-by-error, the perceptron learns every compatible pattern all the time, regardless of how much biasing is present! This shows a crucial benefit of training-by-error: it is guaranteed to converge for every compatible pattern.

Figure 3.23 repeats the test shown in Figure 3.4; it shows the percentage of the time both training strategies do not learn random patterns as a function of the number of patterns. Notice that training-by-error is significantly more capable of learning random patterns that training-by-correlations.

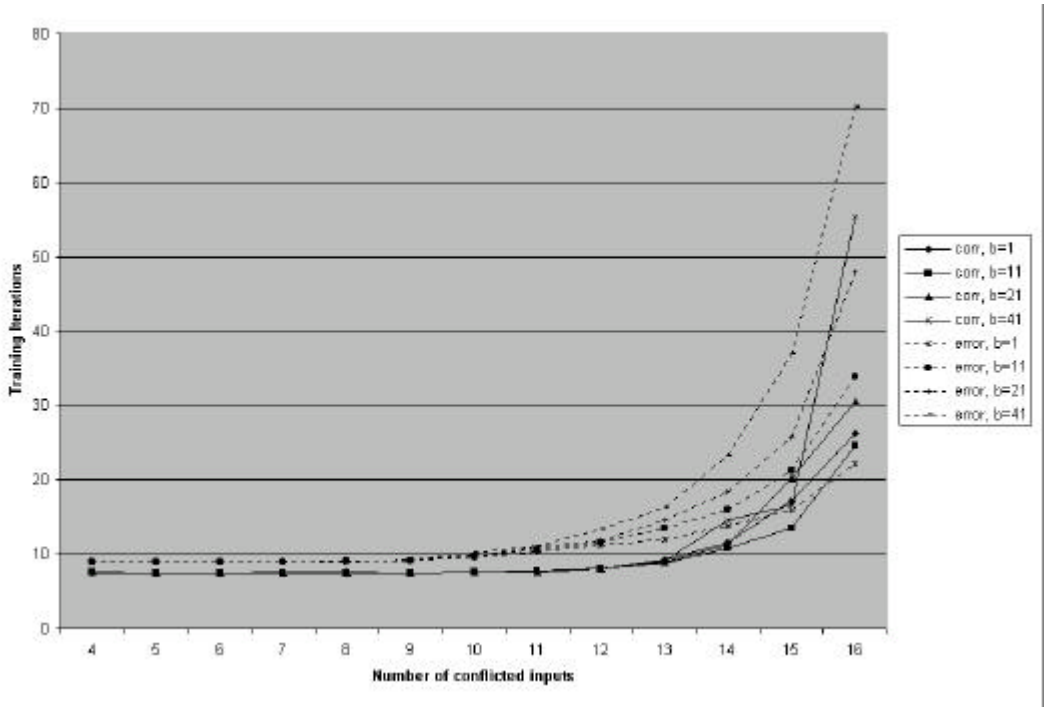


Figure 3.21. Training time for both training strategies

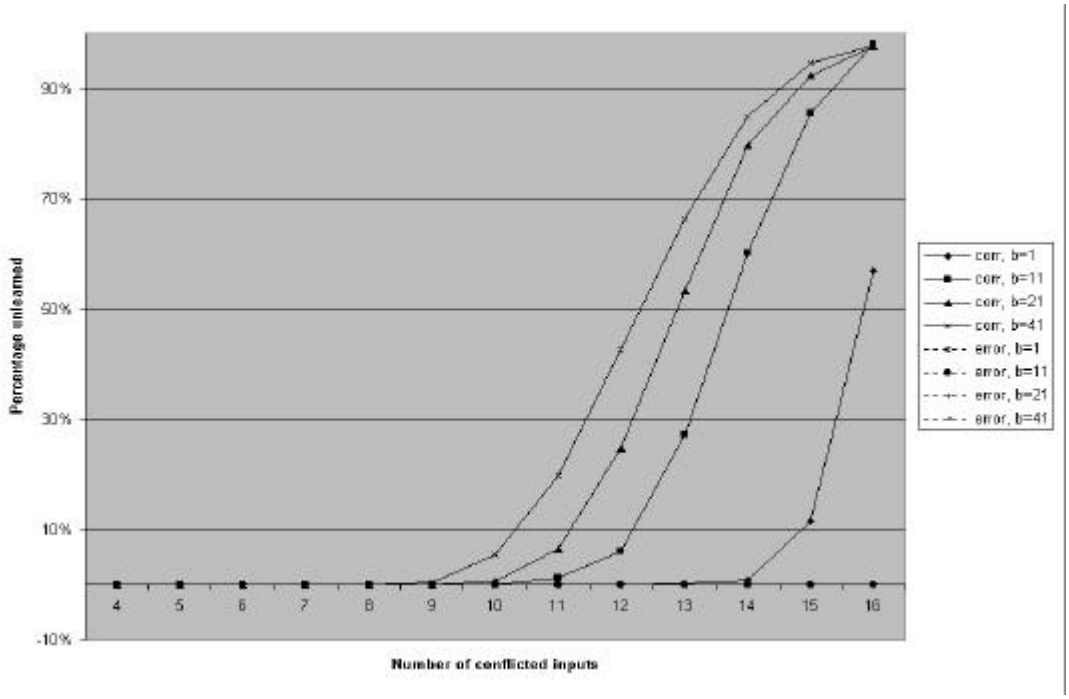


Figure 3.22. Percentage of patterns unlearned for both training strategies

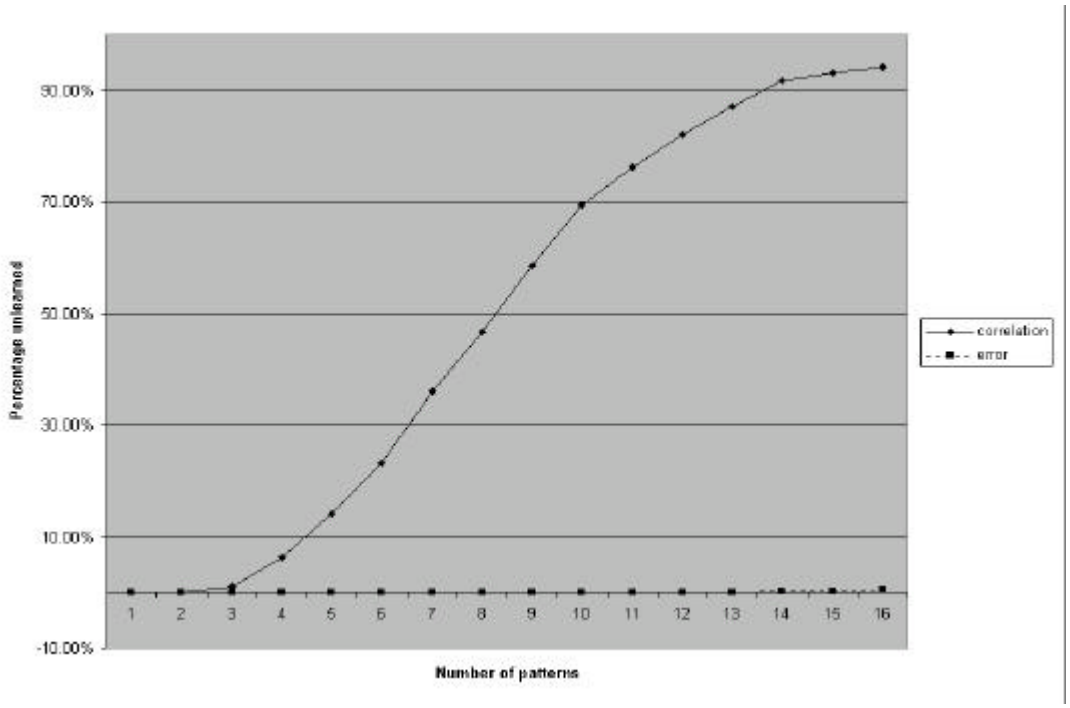


Figure 3.23. Percentage of the time each training strategy cannot learn

In the second study, I examine how well each strategy deals with uncorrelated inputs. I determine the learning time as a function of the percentage of correlated inputs, for each training strategy. My perceptron, implemented in C, has *n* inputs, of which *c* inputs are correlated. In each test, the correlation direction for each of the *c* inputs is chosen randomly. The perceptron is then trained on random values until it learns. Training works as follows: a random "correct" output value is determined. The *c* inputs are given the appropriate input value relative to that correct value (a directly correlated input would get the same value, an inversely correlated input would get the opposite value). The remaining inputs are given a random value. The perceptron produces a guess and is trained according to the training strategy. The perceptron is considered to have learned when it produces correct guesses for 10 iterations. The average training time is computed as the average of the training times

for 1000 tests. The training time for each individual test is determined as the number of iterations until the perceptron learns minus 10.

Figure 3.24 shows the average training time for each training strategy as a function of correlated inputs *c*, for $n = 16$. The training strategies considered are: training-by-error, training-by-error with exponential weight growth, training-by-correlation, training-by-correlation with exponential weight growth, and training-by-correlation with a weight growth cutoff of $1.93n+14$. The susceptibility to noise is shown by the higher average training times when few weights are correlated. In general, weight noise ceases to be a problem when a quarter of the inputs or more are correlated. The study shows that training-by-correlation is slightly less susceptible to noise than training-by-error, but only when a very small percentage of the inputs (1/16) are correlated. Using exponential weight growth significantly improves both strategies' noise tolerance. Enforcing a cutoff on training by correlations substantially improves its learning time, but does not appear to affect its noise susceptibility.

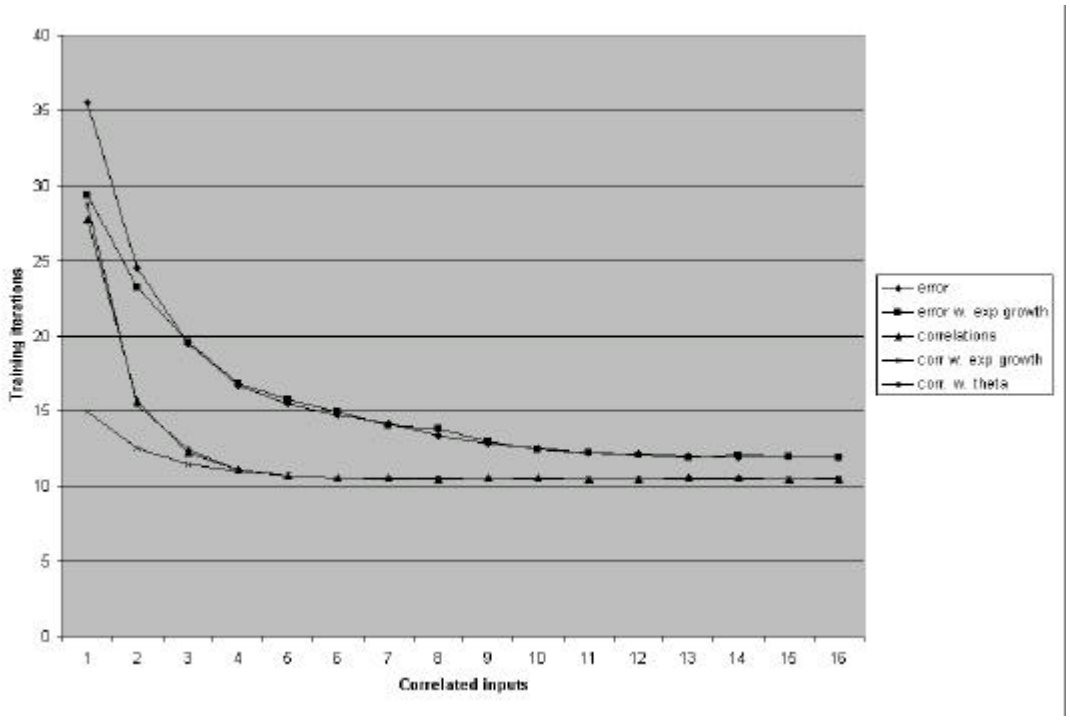


Figure 3.24 Training time compared for training strategies

3.3. The Multibit Perceptron

3.3.1. Defining the Multibit Perceptron

Thus far, I have only considered the branch prediction predictor model, with single bit inputs and a single bit output. This works fine for predictors that only need to choose between two alternatives, such as a predictor making a decision, or a predictor predicting whether a characteristic exists. However, this does not work so well for predictors that need to choose between multiple alternatives, or predictors that need to predict a value. A data value predictor, as will be discussed in depth in Chapter 5, must produce either an entire data value or an index to a data value. In either case, the predictor's output must be more than one bit.

Figure 3.25 shows a diagram of a generalized multibit perceptron. Like the single bit perceptron, it has multiple inputs and a single output. Unlike the single bit

perceptron, however, each of these inputs and the output consist of multiple separate bits, the number of which is constant across the perceptron. Regardless of the multiple bit size of the input and output, the perceptron should function equivalently to the single bit perceptron, and detect correlations between each multibit input and the multibit target.

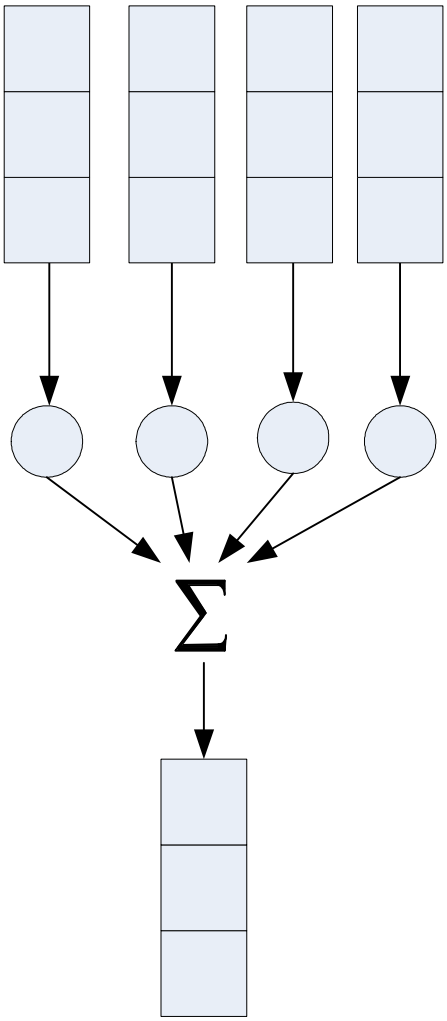


Figure 3.25. A Multibit Perceptron

There are three challenges to making this multibit perceptron. The first challenge is in determining what it even means for there to be a correlation between a multibit input and target. The second challenge is how to devise the multibit

perceptron so that it learns and behaves analogously to the single bit perceptron. The third challenge is how to keep the complexity of the multibit perceptron small, so that training time, latency, and physical size stay within reasonable limits.

3.3.1.1. Defining Multibit Correlations

In the first case, it is necessary to determine what a multibit correlation is. Let's look first at the familiar single bit correlation. Single bit correlations between an input and the target were previously categorized as direct or inverse. A direct correlation means that if the target has value 1, the input always has value 1, and if the target has value 0, the input always has value 0. An inverse correlation means the opposite: if the target has value 1, the input always has value 0, and vice versa. For all other cases, the input is deemed uncorrelated. An uncorrelated input could be one which has the same value regardless of whether the target has value 1 or 0. It could also be an input which produces both values 0 and 1 for a single target value. Basically, an input is correlated if its value infers a target value. A particular input value necessarily means a particular target value. Additionally, a change in input value necessarily means a change in target value.

The key difference between the multibit and single bit cases is that a multibit may have an arbitrary number of possible values instead of two, limited only by the number of bits. Nevertheless, I will use the same definition. An input is correlated with the target if each input value that occurs infers a particular target value. This definition is far broader for the multibit case than the single bit case, and needs some clarifications.

First, the set of input values that occur may be smaller than the set of all possible input values. I propose to neglect those input values that do not occur from the definition; since they never occur, it does not matter how the target responds to them. Likewise, all target values that never occur can be neglected.

Second, unlike the single bit case, there does not necessarily have to be a one-to-one mapping between each input value and target value for a correlated input; instead, several different input values may each map to a single target value. This does not violate the rule that each input value infers an output value. However, two or more target values may not both map to the same input value; otherwise, how can it be determined which target value the input value infers? In short, each target value that occurs must have a set of one or more input values, and these input value sets cannot intersect.

Third, recall that in single bit correlations, if one input value is correlated with one target value, the opposite value is inferred. For example, if the target produces 1 when a correlated input produces 1, the target must produce 0 when the input produces 0. This is not the case with multibit correlations. In a multibit predictor, if the target produces a 3 whenever a correlated input produces a 2, if the target is not 3, the input cannot be 2. However, this 2-3 correlation does not infer that any particular target value will be produced for any other input value. It also does not infer that the target value will not produce the number 3 again for a different input value. The consequence of this is that, unlike the single bit predictor, the multibit correlational predictor cannot use previously observed input values to learn the correct response to

unobserved input values. The multibit predictor can only learn an inference between an input value and a target value after an example of them has been observed.

3.3.1.2. Multibit Perceptron Complexity

The massive problem in designing a perceptron approach to handle multibit correlations is the complexity of the perceptron required. In the worst case, the storage size needed to completely learn the correlation between a single multibit input and the target is exponential in the number of bits. This can be easily shown as follows. Suppose that the target produces every possible value. A correlated input would need to produce a different value for each target value. Assuming that there is no function producing target values from input values, the predictor would need to store all of the value mappings. If there are b bits, 2^b mappings would need to be stored for the input.

A perceptron observes not only the presence of a correlation, but the degree of correlation using the magnitude of a weight. For the perceptron to not only learn all the value mappings but the reliability of each value mapping, the perceptron would need a separate weight for each possible value mapping, or 2^b weights. In the single bit perceptron, only a single weight was needed, because one value mapping inferred the other. However, since with multibit correlations one value mapping does not infer another, every value mapping needs its own weight.

It is not necessarily feasible to design a perceptron with 2^b weights per input. As will be shown below, multibit perceptrons can still be designed with smaller numbers of weights per input. The consequence, however, is that the resulting perceptron cannot be guaranteed to learn the full correlation between any input and

the target. Much like single bit perceptrons with context-based inputs can only learn sets of compatible input patterns, the multibit perceptron can only learn compatible sets of value mappings between any input and the target.

3.3.2. Multibit Perceptron Topologies

The concept of a multibit perceptron has its origins in Rosenblatt's book. Rosenblatt proposed three different multibit topologies, which he called "fully coupled", "disjoint", and "randomly selected." These names refer to the connections between A units and R units. The randomly selected approach, where, on each prediction, A units for each bit are randomly drawn from a larger pool of A units, is probably unsuitable for most computer architecture applications. Both the fully coupled and disjoint approaches, however, are worth considering further.

3.3.2.1. The Disjoint Perceptron

The disjoint perceptron approach is shown in Figure 3.26 and is modeled after Rosenblatt's disjoint topology. Each target bit has its own independent single bit perceptron, whose inputs are the corresponding bit of each input. Correlations are learned independently for each bit. If a single bit perceptron can be thought of as a line, with individual weights as points along the line, this b-bit multibit perceptron can be though of as a b-dimensional hypercube, with each multibit weight occupying a point in the b-dimensional hyperspace. The dot product of the weights for each dimension determines which sector the prediction lies in; that sector becomes the perceptron's decision.

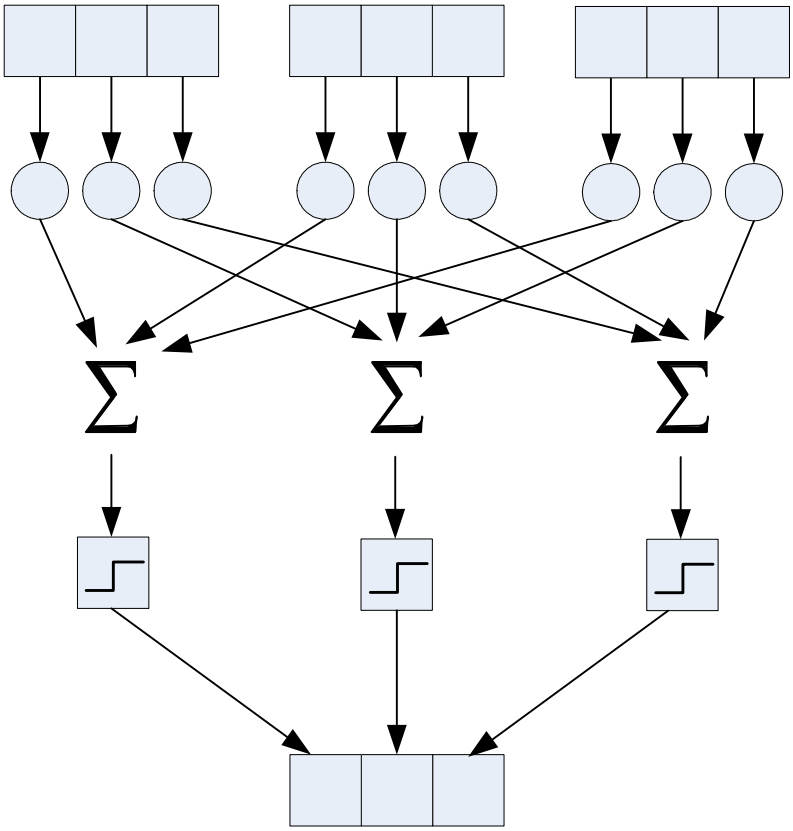


Figure 3.26. Disjoint Perceptron

The advantage of the disjoint perceptron is that for b bits and i inputs, it requires b*i weights, which is significantly fewer than for the fully coupled perceptron. The disadvantage, however, is in its ability to learn value mappings.

Figure 3.27 shows the learning limitations of this type of perceptron. Suppose that a single input 3-bit disjoint perceptron is asked to learn a value mapping 5-1 (input value 5 infers target value 1). The three single bit component perceptrons will each learn the correlation for their respective bits to generate this mapping, and from most to least significant will learn inverse, direct, direct. To learn a second value mapping without conflicts, that mapping will also need to set the weights to inverse, direct, direct. Effectively, by learning one value mapping, the perceptron learns a set of value mappings, the rest of which may or may not be accurate. Consequently, this

perceptron cannot be guaranteed to learn more than one value mapping for each input without conflicts. Since a correlation may consist of many different value mappings, this would appear to severely limit the usefulness of the disjoint perceptron approach.

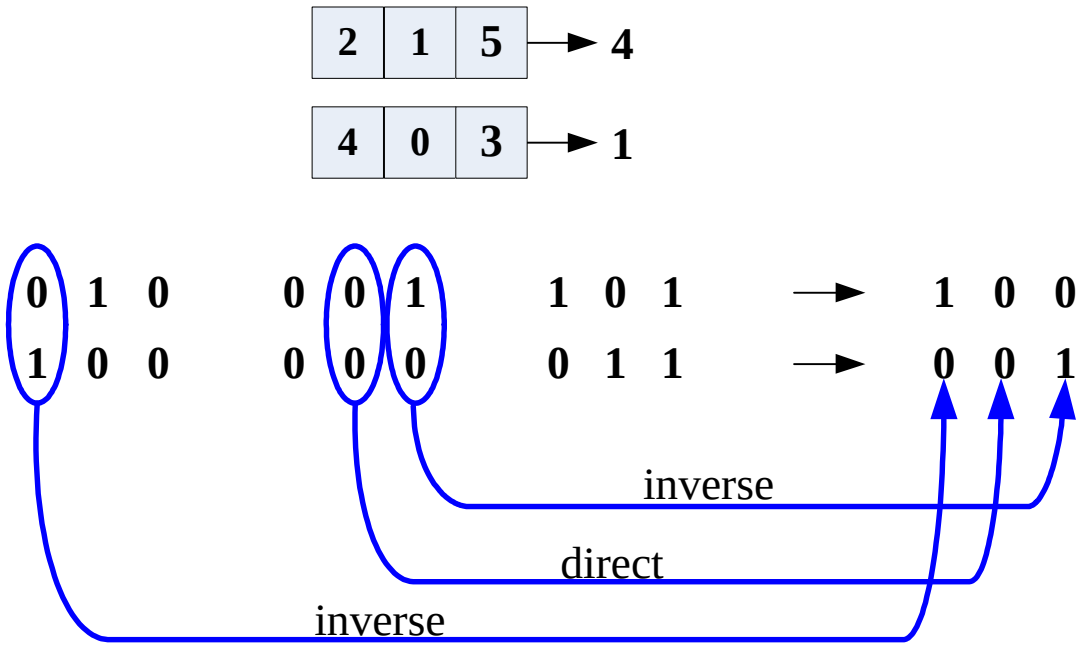


Figure 3.27. Disjoint perceptron learning from corresponding bits

Such conflicts, however, do not make disjoint perceptrons useless. Although the perceptron may not be able to learn a full correlation from a single input, it can learn the correlation from several correlated inputs put together. Consider the 3-input 3-bit disjoint perceptron in Figure 3.26 that is learning two sets of value mappings. The conflicts occur at different bits for different inputs. Although the perceptron cannot learn the entire mapping from any particular input, it can learn one bit of the mapping from one input and another bit from a second input. If the perceptron has sufficiently many correlated inputs, it can learn any complete mapping from the combination of the inputs.

3.3.2.2. The Fully Coupled Perceptron

Figure 3.28 shows the fully coupled perceptron approach, modeled after Rosenblatt's fully coupled topology. In the fully coupled perceptron, each target bit has a weight not just for the corresponding input bits, but for every bit of every input. This approach has the clear disadvantage over the disjoint approach that the perceptron requires b^2*i weights. The additional weights mean additional storage, and additional potential for uncorrelated weight noise. However, with additional weights learning correlations between different bits, the fully coupled perceptron is theoretically capable of learning a full correlation from fewer correlated inputs than the disjoint perceptron.

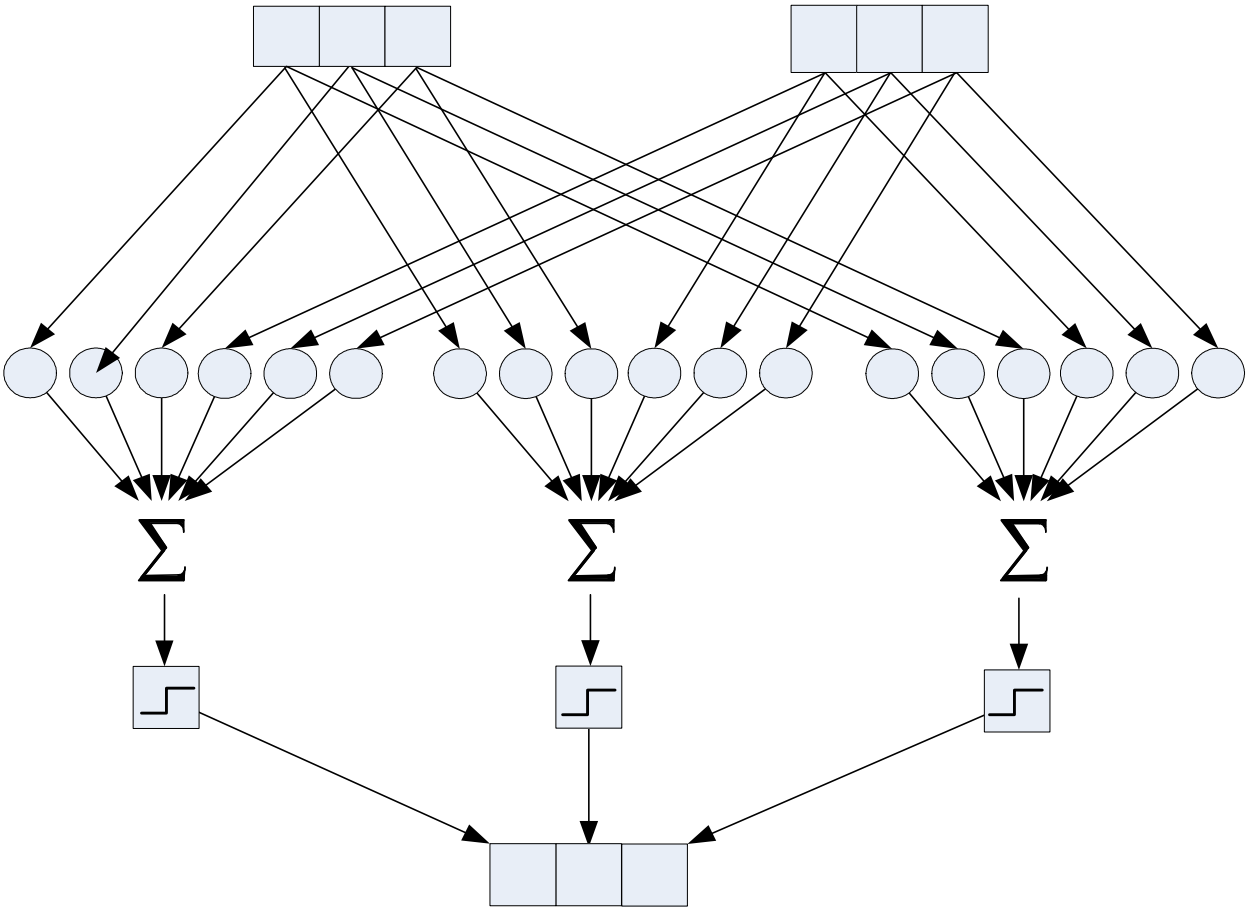


Figure 3.28. Fully-Coupled Perceptron

Figure 3.29 shows the fully coupled perceptron's ability to learn of value mappings between a single input and the target. While the disjoint perceptron could be guaranteed to learn only one value mapping, the fully coupled perceptron can learn any two value mappings without conflict. It cannot, however, learn any three value mappings without the possibility of conflict. However, even for a set of three or more mappings, the probability of conflict with the fully coupled perceptron is less than that of the disjoint perceptron.

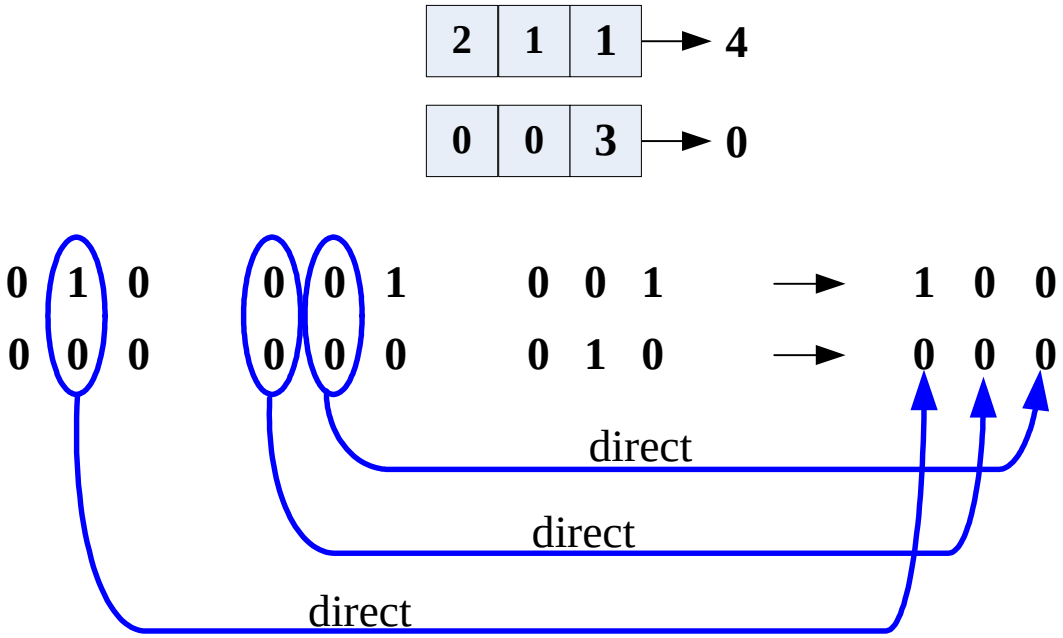


Figure 3.29. Disjoint perceptron learning from any bits

3.3.2.3 Disjoint and Fully Coupled Compared

In this study I compare the relative abilities of the disjoint and fully coupled perceptrons to learn a set of values. I implement a disjoint perceptron and a fully coupled perceptron in C. Each perceptron has *n* total inputs, where *n* is fixed at 16. Each input and the output have *b* bits. Values are generated so that there are *v* different values occurring at each correlated input and *v* different target values; thus

the perceptrons must learn *v* value mappings for each correlated input. The variable in this study is *c*, or the total number of correlated multibit perceptron inputs. Values are generated at each input as follows: If the input is designated as correlated, *v* input values are randomly generated and associated with each of the *v* output values.

Once the values are chosen for a given test, the perceptron is trained. Each of the *v* output values are repeatedly chosen as a correct value over successive iterations, and the corresponding input values are supplied to the *c* correlated perceptron inputs. The uncorrelated perceptron inputs are supplied completely random values. Up to 1000*v* training iterations are performed. The perceptron is considered to have learned the input values if it is correct for 2*v* successive iterations (it has correctly predicted each output value in turn twice), and the test is terminated. If after 1000*v* iterations it has not produced 2*v* correct predictions in a row, it is considered unable to learn the input values. A battery of 1000 tests are performed for each value of *c* from 1 to *n* and an average learnability rate is determined for each *c*.

Figures 3.30 and 3.31 show the learnability rate as a function of *c* for both multibit perceptron types with $v = 2$ and 4. As may be expected, there is no guarantee that the perceptron will learn an arbitrary set of input values, even when $c = 16$. However, the perceptron performs significantly better when *c* is at least $n/2$ than when *c* is less than $n/2$. There are two reasons for this. The first is that a larger *c* means a greater chance of finding a weight that is not in conflict. The second is that the potentially correlated inputs outnumber the inputs that are not correlated, reducing the effects of noise.

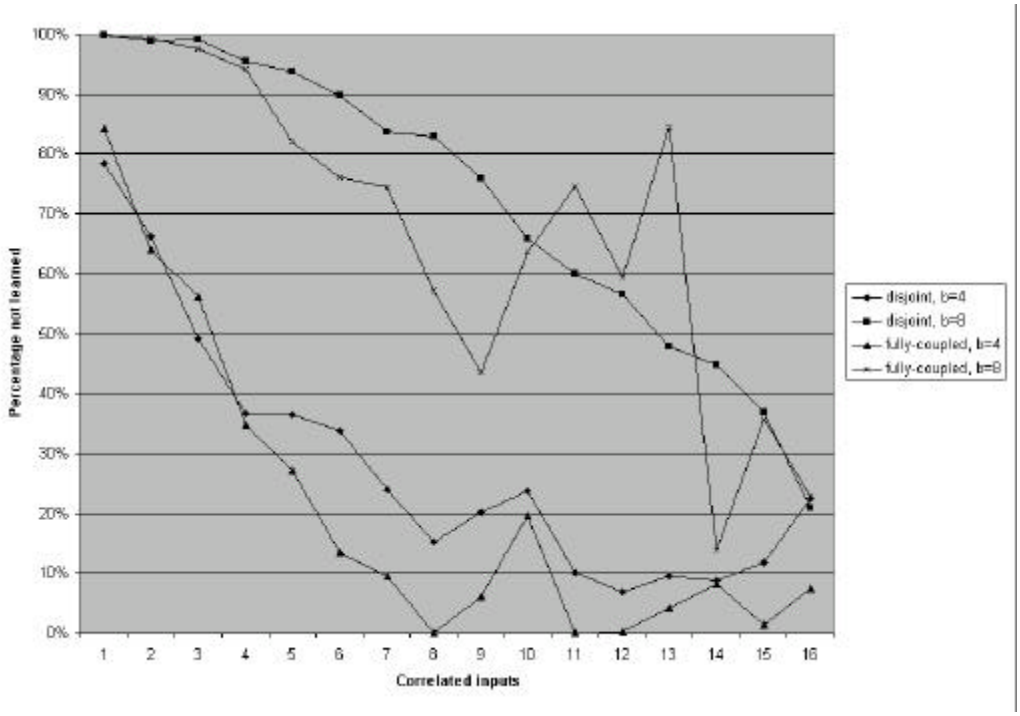


Figure 3.30. Learning rates with 2 values per input

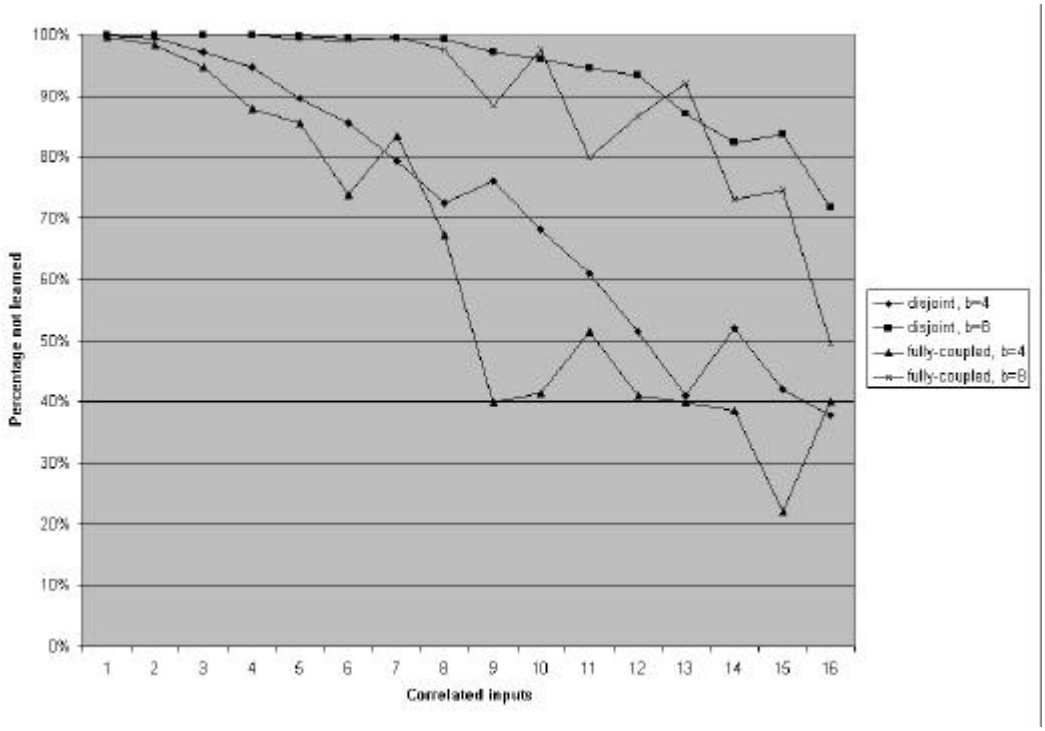


Figure 3.31. Learning rates with 4 values per input

Learnability itself is only part of the equation; even if the perceptron can learn the patterns it needs to learn them rapidly if it is to be useful in value prediction and other applications. Figure 3.32 shows the average training time as a function of *c* for *v*=2. Training time is computed as the average number of training iterations required to learn, minus 2*v* (since it was correct for 2*v* iterations, it is assumed to have already learned the patterns before those iterations). Test iterations in which the perceptron never learned are excluded.

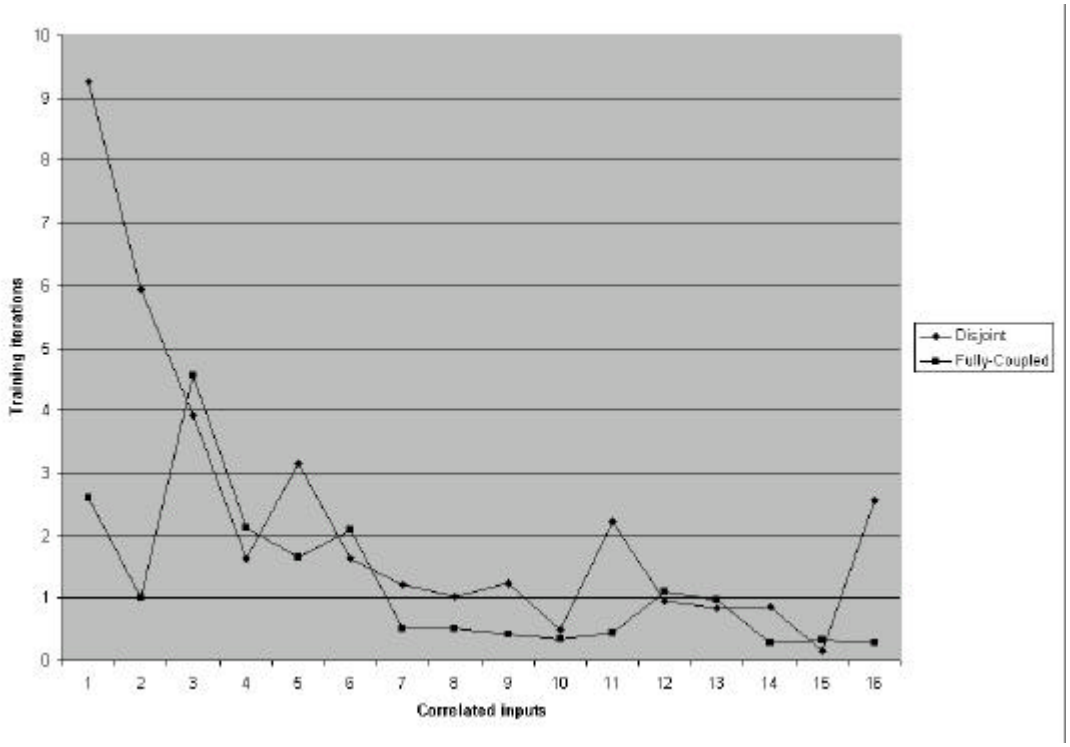


Figure 3.32. Training times with 2 values per input and 4 bits

3.3.2.4. A Weight for Each Value

If the number of input and target data values that could ever be predicted are limited to a small enough number, it is possible to design a practical perceptron that can learn all the value mappings, and hence the full correlation, for every correlated input. This proposed perceptron is shown in Figure 3.33 with 2 inputs, 2 bits, and 3

values/input. Like the previous approaches, this perceptron is comprised of single bit perceptrons for each bit of the target. Each of these single bit perceptrons have a separate weight for each possible value of each input. The input to the weight is simply "1" if that value is observed for the input, and "-1" otherwise (the perceptron lacks formal S units).

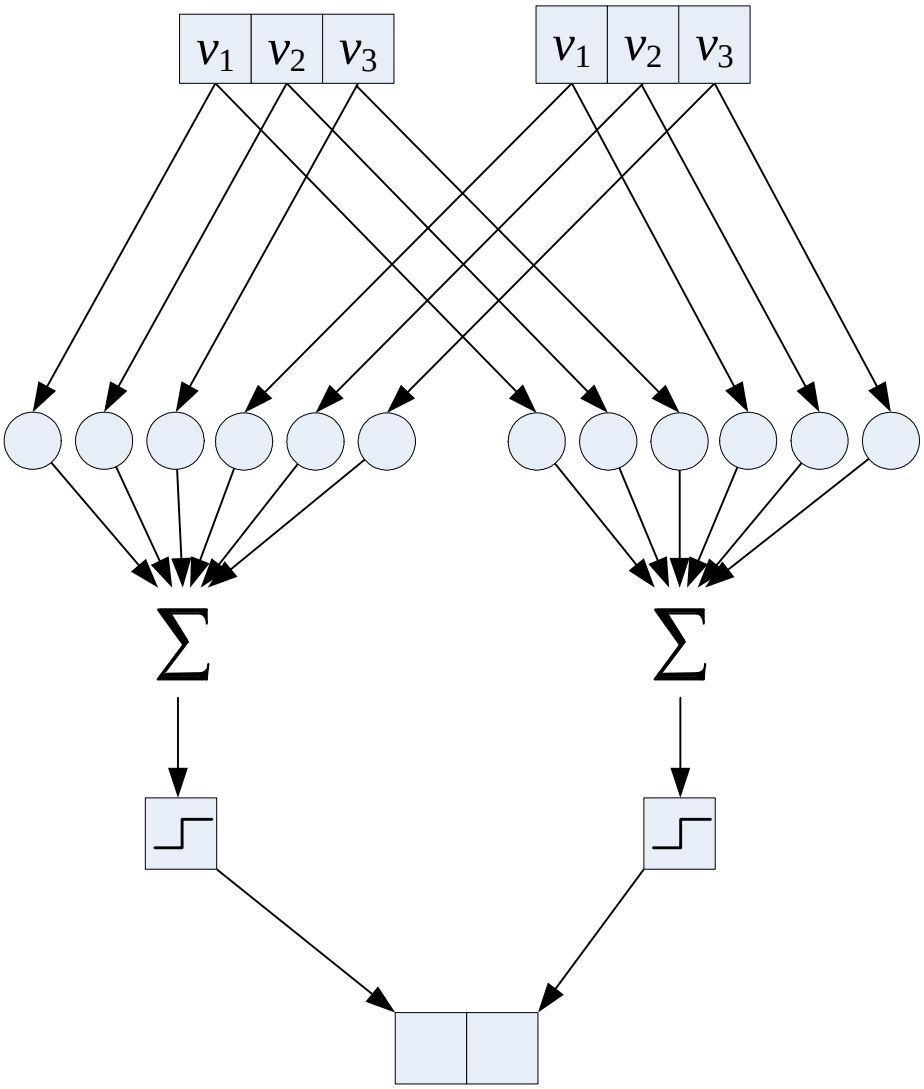


Figure 3.33. Weight-per-value Perceptron

The clear advantage of this perceptron approach is that it has sufficient weights to learn all the value mappings for each correlated input, because each value

has its own weight to learn its correlation. Fewer correlated inputs are consequently needed to produce accurate predictions, and the perceptron can rely less on weakly correlated inputs and focus instead on the strongly correlated inputs. If a small input set is used, or the input set has few strongly correlated inputs, this approach is likely to outlearn the above perceptron approaches.

The biggest drawback to this approach is the sheer number of weights, which for i inputs and b bits is $b*2^b*i$ weights. For small numbers of b, however, this approach is not necessarily impractical. Although it suffers from exponential growth with one of its parameters, it is still more space efficient than the equivalent table-based approach, whose size must vary exponentially with i as well. Consequently, if b is kept small, the number of weights may still be dominated by i, with which they grow linearly.

In the value prediction application described in Chapter 5, b must be fixed at 32. However, this approach can still be kept to a manageable size if it is determined that there will be no more than v values learned for each input. If there can be more than v values occurring, some approach must be used to select the v values to be learned. With this limitation, the weights needed can be kept to a manageable b*v*i.

Nevertheless, the quantity of weights for this perceptron approach must clearly be more than the above approaches. There are two problems with this. The first is simply the physical size and power consumption of the storage, as well as the latency from having to add more weights together. The second problem is uncorrelated weight noise. If many of the values for many of the inputs occur infrequently, the noise quantity of uncorrelated weights can dominate the output.

A second, more subtle problem, is that, for a correlated input, only a positive input case is correlated with the target bit. Since only one value can be produced at a time, all but one of the single-bit perceptron inputs for any input will be -1. This can create problems because the perceptron is heavily biased negatively. A possible solution to this is to supply not a -1 input when a value does not occur, but a 0 input, which cancels out the weight. Thus the weight for a value is not trained when that value does not occur, and the weight for a correlated value will not learn conflicts. This approach of using a 0 input value is used for the weight-per-value value predictor in Chapter 5.

3.3.2.5. A Set of Weights for Each Target Value

The logical fourth alternative approach is to have a separate perceptron for each output value. A single-bit perceptron is associated not with each target bit but with each target value, and has an input for each value of each input. A 1 at a single bit input means that the value associated with that input was observed for the multibit input, and a 1 at the target means that the target value should be taken. Clearly, this multibit perceptron can learn all the value mappings for all the inputs, as it has a weight dedicated for each potential value mapping. However, there are two massive problems with this approach.

The first problem is one of sheer size. For b bits and i inputs, the perceptron requires 2^b*2^b*i weights. Although the quantity of weights still grows linearly with i, b must be exceedingly small for this approach to be practical to implement.

The second problem is what to do if more than one single-bit perceptron decides that a value should be taken, or no perceptron decides to choose a value. A

couple approaches might be used. The threshold could be omitted from each perceptron and the value could be chosen whose perceptron has the biggest sum. Alternatively, a second predictor, such as a set of counters for each value, could decide between the perceptrons. Neither of these are particularly satisfactory. The counters really defeat the purpose of the perceptrons in the first place (why not just use counters and omit the perceptrons?) The biggest sum approach is mildly better, except that by eliminating the threshold function, there is no longer a clean decision made as to which value is right.

Finally, it is not clear that this approach provides any real gain over the previous approach, which already had sufficient weights to learn all possible value mappings for each input.

3.4. Interference

Even a "perfect" perceptron, with fast learning and accurate prediction, can be fouled up by bad input data. Interference occurs when different sources of input data, each perhaps easily predictable by themselves, are all mapped to the same perceptron input in some erratic, unpredictable way. Because the perceptron only observes the scrambled interfering data, and not the original sources, it is unable to learn patterns and produce accurate results. Interference is not a problem of the perceptrons themselves, but a result of how the perceptrons are implemented as a predictor. The perceptron implementation strategy used in branch prediction, the model for perceptron predictors in other applications, suffers from two different forms of interference.

3.4.1. Aliasing

The first source of interference is caused by the per-address organization of the predictor. Both the perceptron branch predictor and the table-based branch predictor associate a separate predictor with each branch instruction. They accomplish this by creating a table of perceptrons/pattern tables, and using the address of the branch instruction to index that table and choose a particular predictor. Unfortunately, the massive size of the predictor prohibits actually associating a predictor/pattern table with every single address. Consequently the table size is limited, and a hash of the address, typically the lower bits of the PC, are used to choose the entry. It is thus possible for two different branch instructions to use and train the same predictor. This phenomenon has been well studied for table-based predictors [Sec96] and has been examined in perceptron-based predictors as well [Jim03, Jim05]. It is known as aliasing.

The most trivial way of countering aliasing is by simply increasing the size of the predictor table, thus making it less likely that two branches would map to the same location. The obvious negative side to this approach is the increased size of the table, which grows exponentially in the number of PC bits that are used to index it. The next approach is to use a more creative hash to index the table than simply using the lower bits of the PC, the problems here being 1) finding such a hash, and 2) implementing a hash so that it does not greatly increase the latency in indexing the table (dividing the PC by a large prime number, for example, is unlikely to be a suitable hash function).

After giving up on trying to eliminate aliasing altogether, the next approach is to live with aliasing, and try to prevent it from compromising the predictor. The

approach here is to keep a tag bit of the upper PC bits, in order to detect aliasing. Once aliasing is detected, the predictor might reset its pattern counters so that the table previously trained for a different branch will not produce erratic results for the interfering branch. The analogy for a perceptron would be to reset the weight bits so that the previous branch pattern would not need to be unlearned.

Aliasing in table-based branch predictors has never been fully eliminated; neither has it been in previous perceptron branch prediction work. The general strategy for coping has been to make the table big enough to reduce aliasing to "reasonable" levels, and then simply ignore the problem. Jimenez's work with perceptron branch prediction determined a table size of 4096 creates negligible aliasing degradation. Considering the quantity of research that has explored table-based branch prediction aliasing, it is unlikely that there exists a simple, satisfactory way of eliminating aliasing in per-address perceptron prediction.

3.4.2. History Interference

The second, somewhat less explored source of interference occurs in the mapping of past global branch results to perceptron inputs and specific pattern table bits. This form of interference is shown in Figure 3.34. It has been previously assumed that each perceptron input is associated with a single past static branch instruction. Each perceptron weight thus learns the relationship with its associated static branch instruction and the target branch. However, the global history is produced by shifting in dynamic branch results as they are known. Control flow changes in the program can mean that on some instances of a target branch, one or more past branches may be present, and on other instances, they may not be. For

example, suppose that on past iterations of a branch instruction, the past branch outcomes were from branches *a, b, c, d, e*. Changes in control flow may add or remove past branches on the next iteration, changing the sequence to *a, f, b, c, d*. The addition of branch *f* pushes all subsequent branch results to different perceptron inputs. Thus weight 3 is trained on the result of branch *c* on one iteration, and branch *b* on the next iteration. The effect of this is that the actual placement of past static branch instructions in the global history can change from one iteration to the next. A perceptron input tied to a particular global history entry may in fact be monitoring several past branches. Although each past branch may individually be well correlated with the target, the erratic combination of these branches need not be correlated.

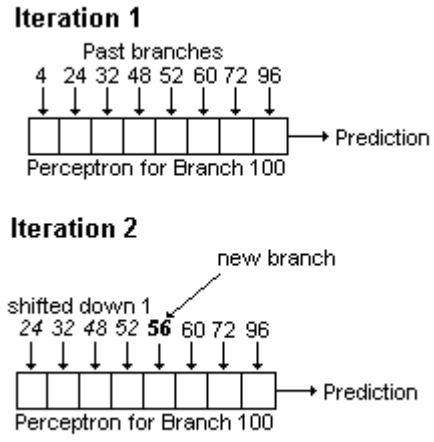


Figure 3.34. Interference in the branch history

Before looking at how to overcome this interference, it is instructive first to examine to what degree this form of interference actually poses a problem. Three questions must be answered. First, do control flow changes that affect the global history really occur frequently enough to affect accuracy? Second, even if they do, are they harmful or benign? Third, this dissertation is not concerned so much with

perceptron branch prediction but with other perceptron applications. Is this problem likely to affect these other perceptron applications as well?

3.4.3. History Interference Does Happen

Does this form of interference, with control flow changes shifting branches to different places in the global history on different iterations, really occur often in branch prediction? I performed the following studies using a perceptron branch predictor identical to Jimenez's implemented in SimpleScalar. The first study, shown in Figure 3.35, gives an initial quantification of interference between branches in the global history. It shows the percentage of the time for each input, over all static branches, that the branch results being sourced to that input come from the same past branch as in the previous iteration. The results are fairly dire: the most recent past branch is a different instruction than in the past iteration nearly 15% of the time. The results show that the problem becomes significantly more severe with longer histories. Figure 3.36 shows how many different past branches are routed to the same predictor input, on average. The 16th input typically gets results from nearly three different branches.

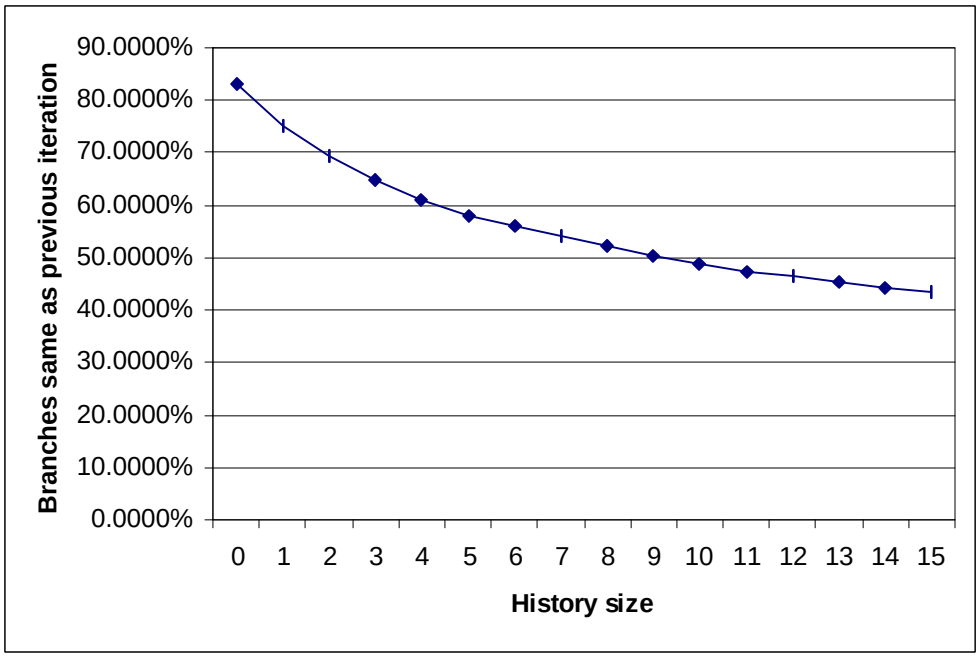


Figure 3.35. Percentage of branch inputs with the same instruction as the last iteration

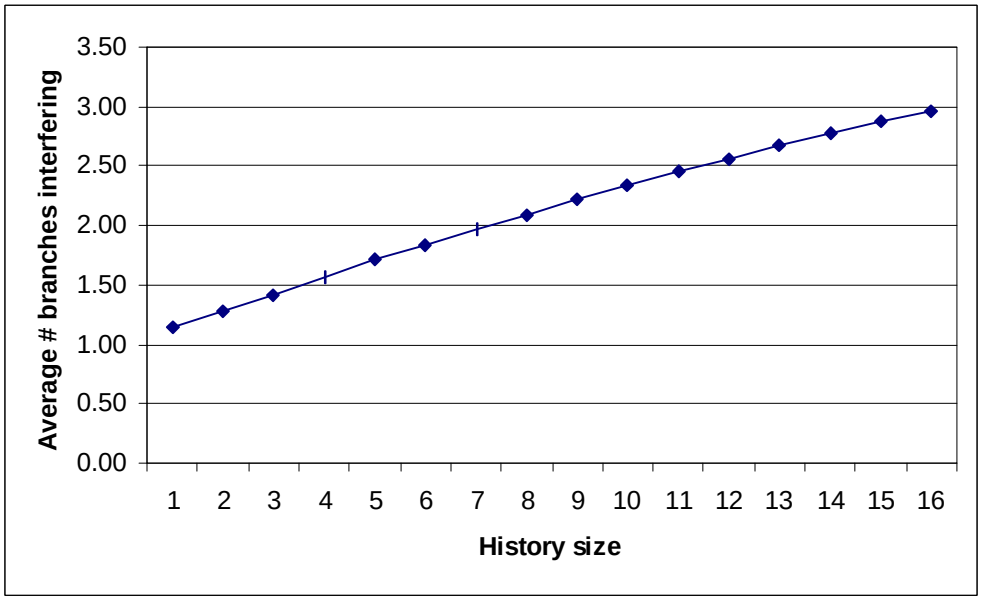


Figure 3.36. Average number of branches interfering at each input

To design a predictor that is tolerant of branch interference, it is important to know not only the average number of interfering branches, but also the maximum. Figure 3.37 shows, for a predictor with 16 inputs, the percentage of branch inputs that

suffer from no interference, at most 2 branches interfering, at most 3 branches interfering, and so on. As shown in the figure, over 50% of predictor inputs never have more than 4 branches interfering with each other.

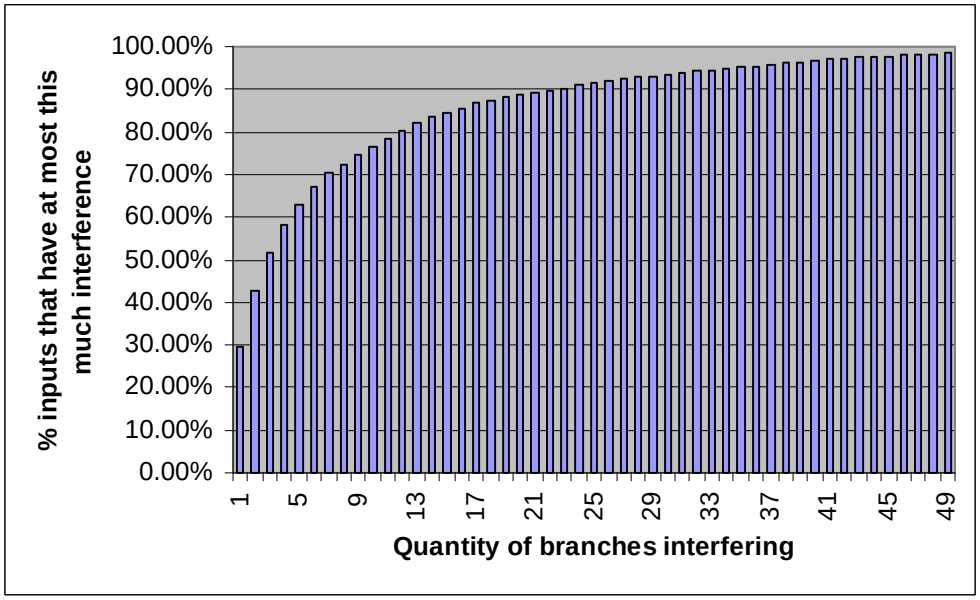


Figure 3.37. Percentage of inputs suffering from varying interference amounts

What this evaluation does not address, however, is the distribution of interference. There is a difference between two branches interfering evenly, so that half the time the input gets the result of one branch and half the time the other, and two branches interfering unevenly, so that one branch dominates the input. If interference is highly uneven, an interference-tolerant predictor could simply treat the more occasional branch as a nuisance and mask its results, whereas if it is even, both branches must be considered. This distribution is approximated by determining the percentage of the time that the most dominating branch is seen by the input; if the percentage for two interfering branches is 50%, the distribution is even; if it is 90%, the distribution is highly uneven. The results, on average, are shown in Figure 3.38.

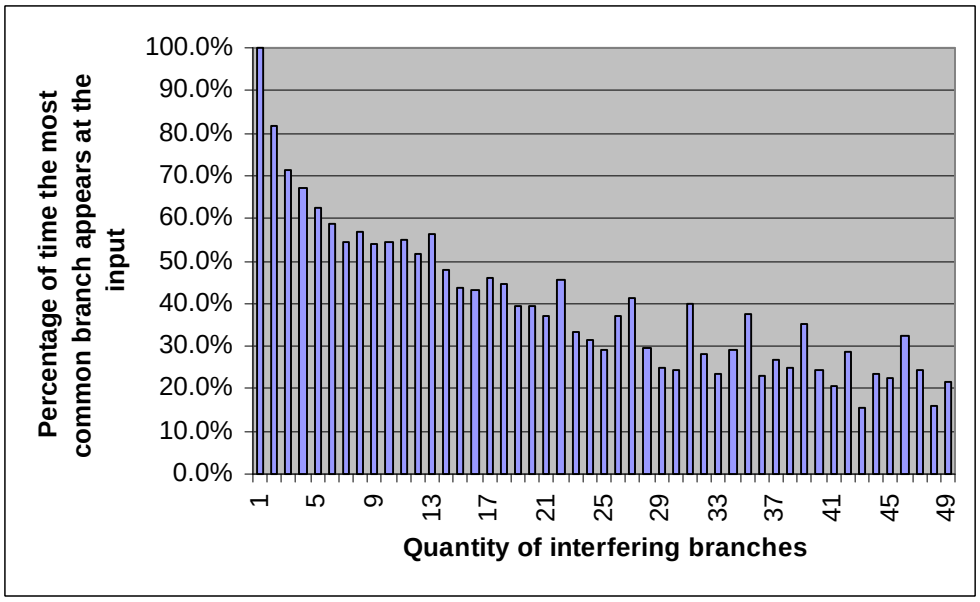


Figure 3.38. Percentage of the time that a dominating branch is seen at the input

Figure 3.38 shows that there does tend to be a dominating branch, though clearly the nondominating branches are not negligible. Interestingly, even for inputs where there are more than 30 conflicting branches, one branch still tends to dominate about 20% to 40% of the time. This suggests that while large quantities of branches may interfere, only a handful of them have any significant effect.

3.4.4. Classifying Interference

Interference need not necessarily be a bad thing. Suppose two different past branches have the same result each iteration: both are taken, or both are not taken. Even though they interfere, they both exhibit the same correlation. These branches appear no different to the predictor than they would if they did not interfere. As these branches both train the predictor in the same way, their interference can be considered constructive interference.

Alternatively, suppose two interfering past branches were both correlated with the target, but in different directions. When one branch was taken, the other was not taken. As these branches train the predictor opposite to each other, their interference can be considered destructive interference. Destructive interference is clearly detrimental to a predictor.

The third possibility is that a correlated branch is interfered with by a non-correlated branch. Relative to the correlated branch, the non-correlated branch sometimes produces one result, and sometimes the other result. This form of interference can be termed neutral interference, since it is neither constructive nor destructive. Neutral interference in fact must be looked at from two perspectives, from the point of view of the correlated branch, and the point of view of the non-correlated branch. The correlated branch sees the addition of noise. The non-correlated branch sees the addition of bias.

3.4.5. Interference Effects

The effects of constructive, destructive, and neutral interference on perceptron accuracy and learning rate are summarized below. For obvious reasons, constructive interference is non-problematic in both prediction approaches, as the predictors do not need to distinguish the interfering branches. It should also be clear why destructive interference is a problem in perceptrons. The destructively interfering branches have different correlations. The conflicted weight is thus trained to be both positive and negative at the same time, resulting in cancellation and a zero weight (especially if the conflicting patterns occur equally often at an input). If the conflict occurs unequally, with one branch occurring at the input more often than the other, the

common branch will have trained a large weight. This large weight magnitude means that more weight is given when the rarer, conflicting branch occurs at the input, causing a misprediction. Clearly constructive interference is benign and destructive interference is harmful.

What happens on "neutral" interference? This interference can be considered from two points of view; from that of a noncorrelated branch occasionally interfered with by a correlated branch, and from that of a correlated branch occasionally interfered with by a noncorrelated branch.

Suppose that a strongly correlated branch interferes with a noncorrelated branch. The noncorrelated branch desires a weight value of near zero. From the point of view of the noncorrelated branch, the interfering correlated branch causes no immediate trouble, as the low weight value means that the perceptron disregards the correlated branch's input. However, the correlated branch trains the weight value away from zero towards the correlation. Thus the noise from the noncorrelated branch is amplified and may affect future predictions. In this case, the correlated branch causes little short term damage but may cause long term damage.

Suppose that a noncorrelated branch interferes with a correlated branch. The correlated branch desires a high magnitude weight value. The noncorrelated branch will not change this; sometimes it will increase the weight, sometimes it will decrease the weight. Consequently it causes no long term damage. In the short term, the non-correlated branch's noise will be greatly amplified by the high magnitude weight and drive the perceptron to mispredict. In this case, the uncorrelated branch causes little long term damage but may cause short term damage.

If the interference is balanced, with both branches interfering equally, both problems occur. The weight is trained to a low but nonzero magnitude. The correlated branch has some voice but may not be able to sway the perceptron as much as it should. The noncorrelated branch is muted to some degree, but its noise is amplified more than it should be. Accuracy is thus reduced for two reasons: a noncorrelated branch is amplified and can drive the perceptron to produce sporadic results, and a correlated branch is muted and its benefits lost.

Figure 3.39 shows the frequency of each type of interference for several benchmarks. The most prevalent form of interference is from uncorrelated branches interfering with directly-correlated or inversely-correlated inputs, occurring significantly more frequently that either constructive or truly destructive interference.

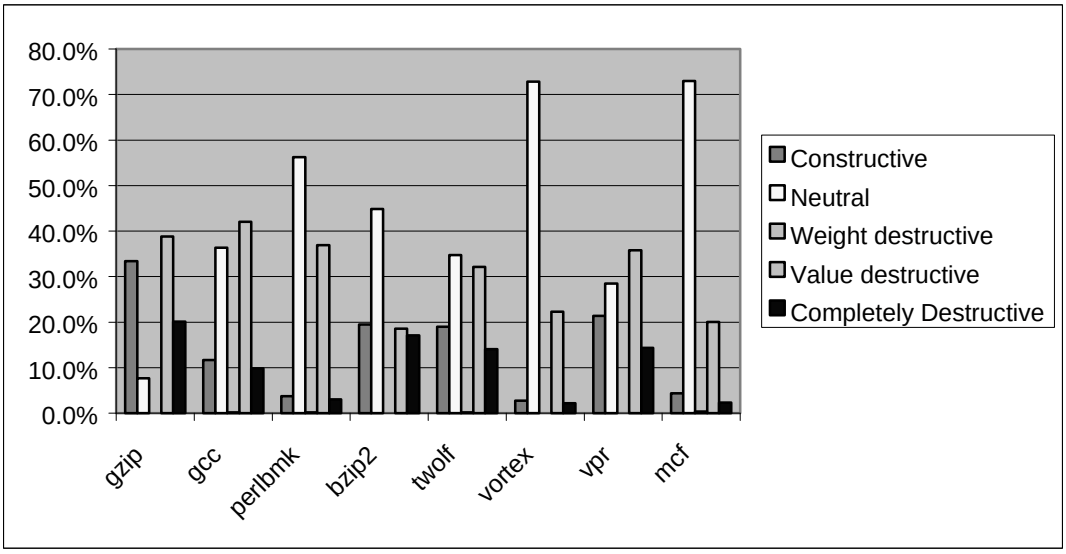


Figure 3.39. Frequency of each type of branch interference

3.4.6. History Interference in the Multibit Perceptron

The above analysis deals with single-bit perceptrons. What about perceptrons with multibit inputs and output? The problem can still occur. In this case, two past multibit sources are both mapped to the same multibit input on different iterations.

In the single-bit perceptron, interference occurs between two single bit sources. Since a single source can exhibit only one type of correlation (direct, inverse, or none), these two sources can interfere only in a single way (constructively, destructively, or neutrally). In contrast, each individual bit of a multibit source effectively exhibits its own type of correlation. When two multibit sources collide, interference occurs on each bit. Each individual bit of interference can be different, with some interfering constructively, some destructively, and some neutrally.

The problem with multibit interference is that it is only benign if every single bit interferes constructively. Consider two multibit sources, each one fully correlated with a set of value mappings. Source 1 produces value mappings 01-11 and 10-00. Source 2 produces mappings 01-10 and 10-01. What happens when they interfere? The first bit is inverse for both the first and second sources; they interfere constructively. The second bit is direct for the first source but inverse for the second source. They interfere destructively and their weights are reduced to 0. When the interference occurs, the perceptron can only learn the first bit, and not the whole value mapping. Thus while the perceptron might have learned a correlation from the single input had there been no interference, it will need more inputs to learn the correlations from the cancelled bits.

A multibit perceptron undergoing interference can be analyzed much the same way as a multibit perceptron learning a correlation from a set of value mappings.

Recall that the multibit perceptron input can only learn a set of value mapping if the set is compatible. Otherwise, it learns the compatible bits (those that correlate the same way), and must learn the other bits from other perceptron inputs. Multibit interference has the effect of increasing the set of value mappings to the union of the sets of the interfering inputs. Thus it is less likely that a particular bit will be compatible. This bit must then be learned from another perceptron input. Thus interference means that more correlated inputs are needed in order to learn the bits cancelled by destructive interference.

3.4.7. Coping with History Interference

Here I examine three methods of coping with history interference. Neither way is really ideal; each has reasons to recommend it and problems. In later chapters, each method is applied to the application under test, and the effectiveness of each method will be compared.

3.4.7.1. "Assigned Seats"

The most effective way of eliminating history interference would be to ensure that every source is always mapped to the same perceptron input on every iteration. To do that each source must be identified and assigned to a perceptron input. There are at least three issues that must be tackled: 1) identifying the sources, 2) providing a perceptron input for each source, 3) mapping each source to its input.

Identifying the sources, in branch prediction, is simple. Each past dynamic branch has an address. This address tells which particular branch a result belongs to.

Previously only the branch result was stored in the history register. The history register can be easily updated to store both the branch result and the branch address.

Providing a perceptron input for each source is less simple. It is just not clear how many different static past branch instructions might occur in the history for a given branch, looking at the static code. Additionally, each branch can have a different quantity of past static branch instructions in the same global history length. Thus choosing a fixed quantity of perceptron inputs for each perceptron means that some perceptrons will have too many inputs for the given history size while others have too few.

Mapping each source to a different perceptron input is the biggest challenge. One method could be to choose a perceptron input by hashing the branch address. Because of latency concerns, a simple method, such as using the last bits of the address to choose an input, must be employed. Routing the result to the input poses another problem. This can be accomplished at the point that the branch result is placed in the history. Instead of shifting the branch in, the branch's place in the history can be chosen by the last bits of the branch address. The history can then be mapped directly to the perceptron inputs as before. This has the additional advantage that the branch address need not be stored in the history, as it is already implicitly stored by the branch's placement.

Mapping creates another challenge. What happens if two branches have the same last bits? This is, of course, the problem with this approach. One of the branches would need to be discarded. I propose that the older branch (the one placed in the history first) be discarded. There are two reasons for this. First, the history

will be constantly updated, as every new branch result will be placed, instead of filling up with old branches. Second, more recent branches generally correlate better than less recent branches. This is likely to be true with other applications as well.

The advantage to this approach is that it guarantees that the same branch instruction will always be sourced to the same perceptron input. There are two problems, however. The first problem is when two branches map to the same history entry. It is conceivable, and even likely, that an uncorrelated branch will always overwrite a correlated branch. This problem is simply ignored. The second problem is that interference can still occur when a past branch sometimes occurs and sometimes does not occur. If a branch does not occur, an older branch's result will still occupy that history entry. These branches thus still interfere. This interference can be countered, to an extent, by zeroing out all entries that do not occur within the last *n* dynamic branches (to a perceptron, inputs are -1 and 1, and a 0 input means to ignore the input). The interference can still exist, if two branches occur within the last *n* branches, but it removes the problem of a newer, more reliable branch interfering with an older, less reliable branch.

Figure 3.40 shows this approach, which I term "Assigned Seats." When a branch result is known, the lower bits of the address (excluding any address bits that are always zero) are used to choose a global history entry. The branch result is stored at that entry as a -1 or 1 (requiring 2 bits per entry). On a prediction, the bits stored at each history entry becomes the corresponding perceptron input.

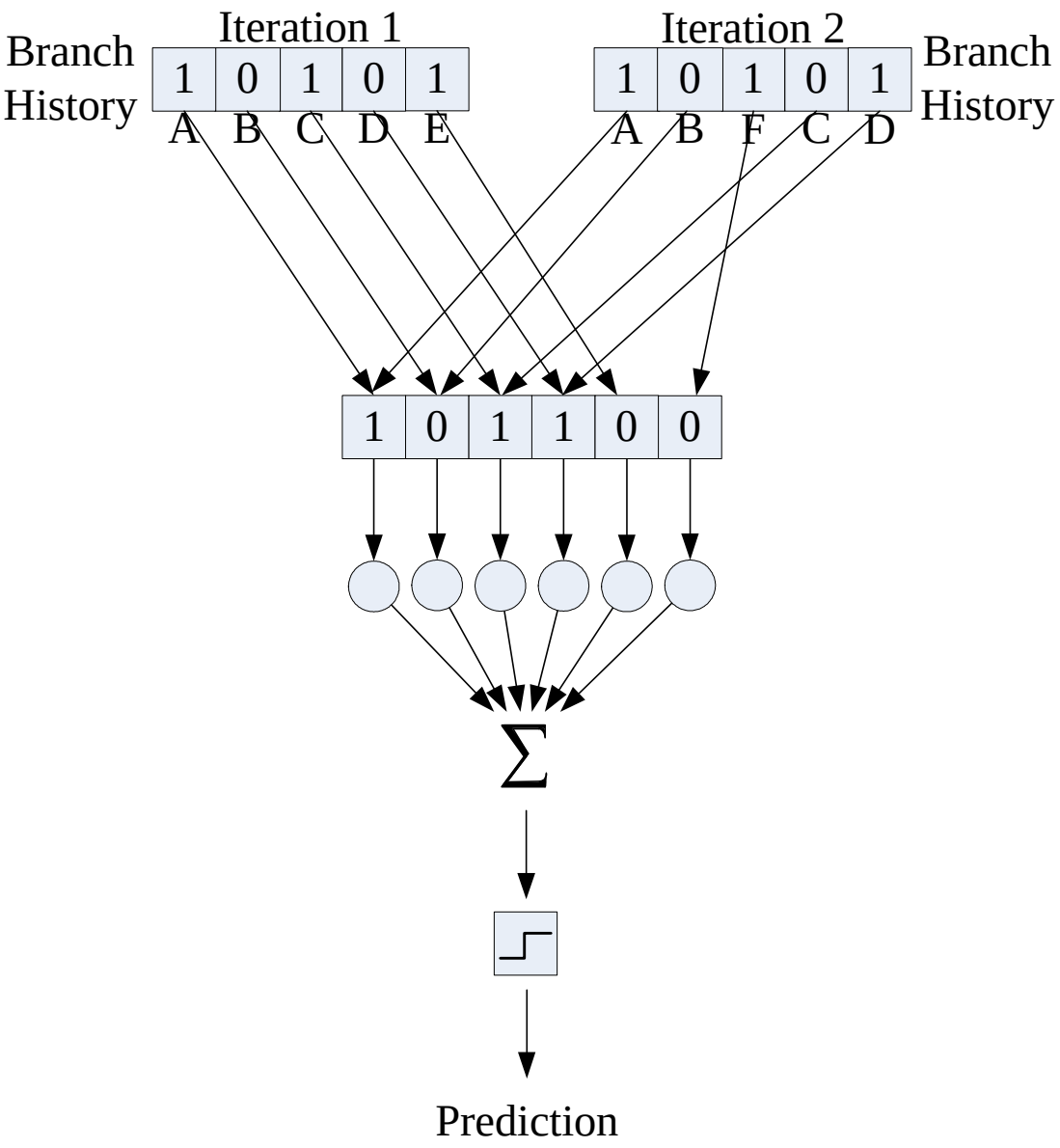


Figure 3.40. Assigned Seats

A variation on this is "Assigned Seats with Cancellation." Each entry has a counter associated with it. The entry's counter is set to a fixed upper bound *n* when a branch result is stored at that entry. When a branch result is stored at any other entry, the counter is decreased. When the counter reaches 0, the bit at that entry is changed to 0. This effectively forces the predictor to consider only the last *n* dynamic

branches, and no more, and avoids interference between a recent past branch that sometimes does not occur, and a less recent past branch.

3.4.7.2. Piecewise Linear

An alternative approach avoids the problems with Assigned Seats. This approach was independently developed both by me and by Jimenez in [Jim05], where he called it the "Piecewise Linear Predictor." Using Jimenez's terminology, the Piecewise Linear predictor associates multiple weights with each perceptron input. The weight is chosen by the address of the branch at that input. The effect of this approach is to separate interfering branches, but not assign them to the same input.

Figure 3.41 shows this approach. The branch address is stored in the history alongside the branch result. When a prediction is made, each past branch result is sourced to corresponding perceptron input. At the same time, the last b bits of the corresponding branch address choose a weight from an array of 2^b weights for each input. Later, on training, only the selected weight for each input is trained.

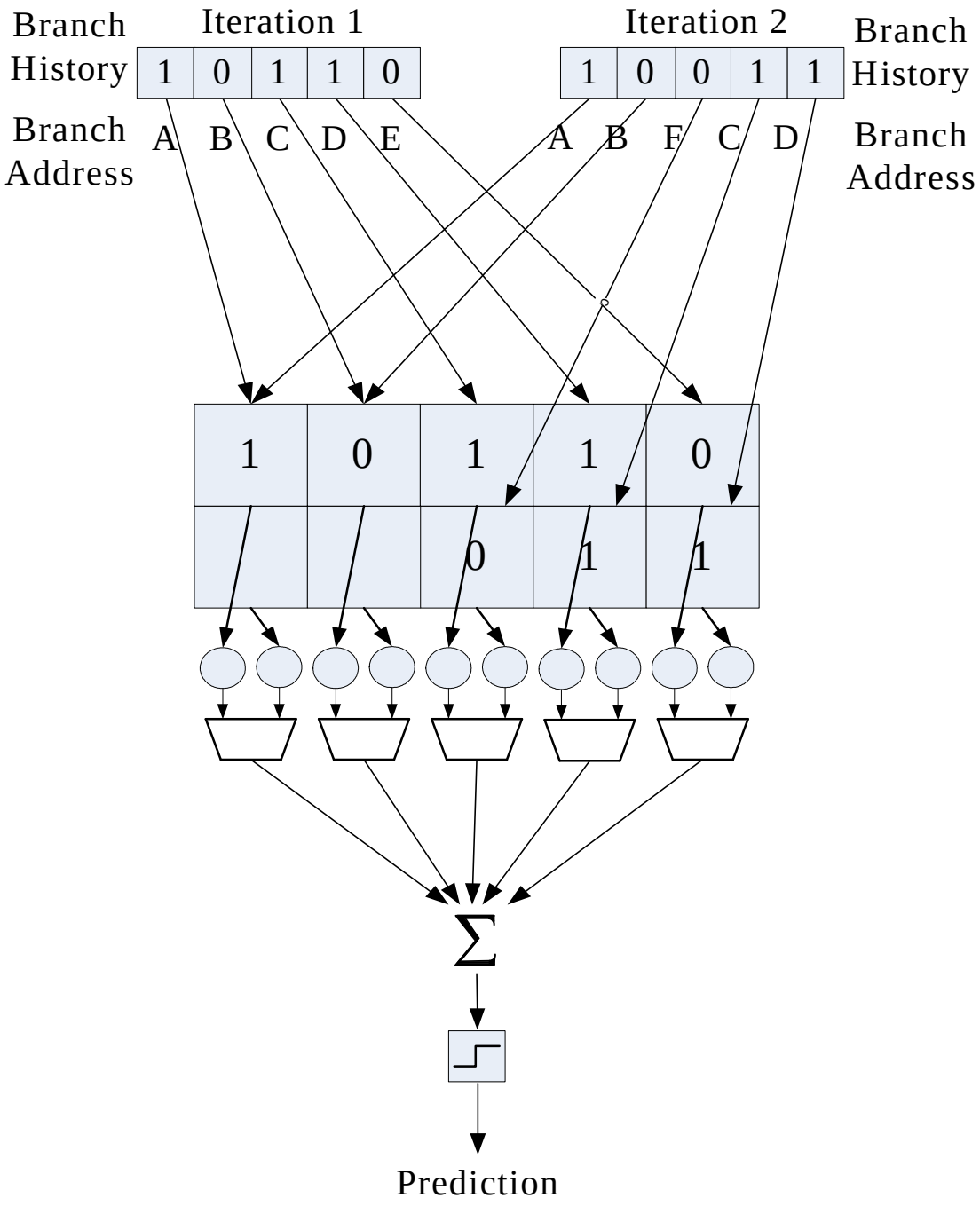


Figure 3.41. Piecewise Linear Predictor

The advantage of this predictor is that no past data is lost. All *n* past branch results in an *n* entry history are used by the perceptron. There are several disadvantages. First, each perceptron input must have several weights associated

with it; not just one. As weights are the biggest contributor to the size of a perceptron, this effectively greatly increases the cost of the predictor. Second, interference can still occur if two branch addresses have the same last bits and map to the same weight. This can only be countered by increasing the number of weights for each input. Third, each branch is still potentially spread across several perceptron inputs. Training is thus spread across multiple inputs. If a branch occurs equally at three perceptron inputs, it will take three times as many iterations to train the perceptron as it would if a branch occurs only at one input.

3.4.7.3. Ignore the problem

If history interference is not a massive problem for a particular application, it may be most cost and performance effective to simply ignore the problem altogether and allow interference to occur.

Chapter 4: Experimental Methodology

This chapter lists the simulation parameters and steps that are common to all the studies in Chapters 5 and 6. It also describes in depth my processor simulator on which these studies are performed, what it simulates, how it works, and how it is evaluated.

4.1. The Simulator

Mysim is a new CPU simulator that I wrote explicitly for the purpose of this dissertation research. It is a cycle-accurate, execution-driven, out-of-order simulator that models a PISA machine. The simulator is written in C and runs on a Linux platform. It is similar to the SimpleScalar simulator in that it simulates at the same abstraction-level and models the same type of machine with similar components and characteristics. Additionally, the code to handle system calls and loading the benchmark program into simulator memory has been partially copied from the SimpleScalar. Apart from these exceptions, and the power simulation add-on, Mysim consists entirely of original code.

4.1.1. Mysim overview

Mysim is capable of modeling three types of machines: a non-cycle-accurate functional machine, a cycle-accurate in-order five-stage pipelined machine, and a cycle-accurate out-of-order machine employing a variant of Tomasulo's architecture. All the simulations in this dissertation are performed only on the latter machine.

The functional simulator simply executes PISA instructions sequentially without modeling any underlying microarchitecture. It was implemented first in order to verify the more complex simulators.

The pipeline simulator simulates a five-stage in-order pipelined PISA machine. All instructions are executed in five stages: Fetch, Register Decode, Execution, Memory Access, and Register Writeback. In Fetch, instruction codes are read from memory; in Decode, register contents are read using the instruction operand bits. In Execution, arithmetic is performed to obtain the result for arithmetic instructions, the address for load/store instructions, or the branch decision for branch instructions. In Memory, the virtual memory is either read or written to, and in Writeback, the execution or memory result is written back to a register. Data hazards are avoided entirely; data forwarding is employed from the Execution and Memory stages to the Decode stage in the event of a hazard. Control hazards are dealt with through calls to a branch predictor.

The out-of-order simulator simulates a superscalar PISA machine based on the Tomasulo algorithm [Tom67]. The architecture is shown in Figure 4.1. Instructions are executed in six stages: Fetch, Dispatch, Issue, Execute, Writeback, and Commit. In Fetch, instruction codes are read from memory into a dispatch queue. In Dispatch, instruction codes are read from the dispatch queue and placed into available reservation stations, where they wait until their operands are available. In Issue, instructions that are ready to execute are assigned to available functional units. In Execute, the functional units execute the instructions assigned to them. In Writeback, the results of completed instructions are written to dependent instructions in reservation stations, and are removed from the reservation stations. In Commit, the results of completed instructions are written in order to the registers. This out-of-order simulator employs a set of reservation stations to hold executing or waiting instructions, a dispatch queue to hold instructions waiting for reservation stations, an issue queue to hold instructions waiting for functional

units, a load/store queue to ensure in-order memory access when necessary, and a re-order buffer to hold all instructions that have not yet committed. It includes forwarding mechanisms to pass completed data to dependent instructions, and a squashing mechanism to remove instructions from the pipeline.

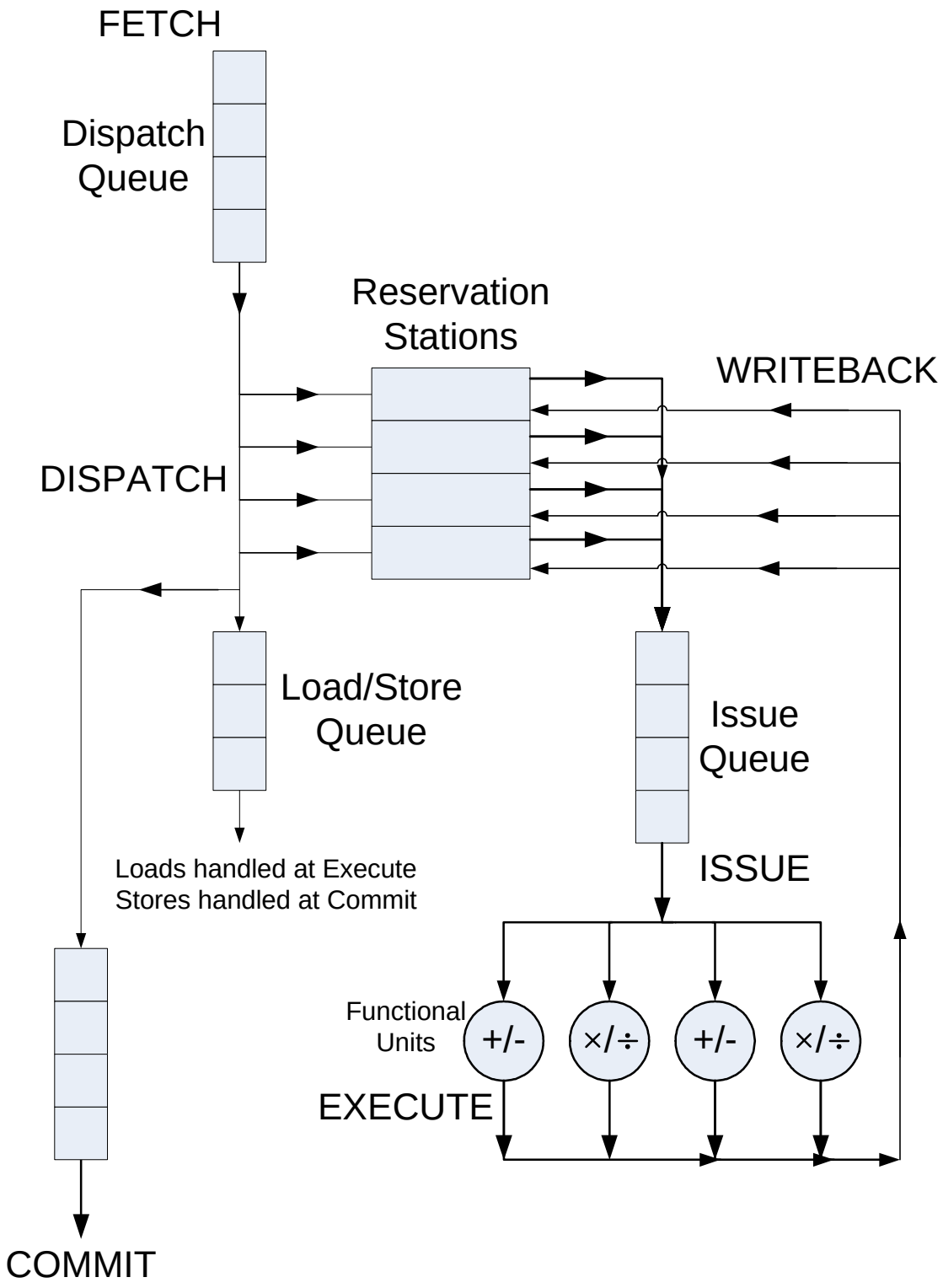


Figure 4.1. Simulator Block Diagram

The simulator also employs by default a virtual memory of unlimited size, and a system call interface that performs actual POSIX calls based on system calls in the benchmark programs. With appropriate command line flags, the simulator can also simulate a 2-level instruction and data cache of variable size, a bimodal branch predictor, a branch-target buffer of variable size, and the Wattch power simulator.

4.1.2. Mysim Core Anatomy

The core of Mysim consists of the following files: mymain.c, Mysimoutorder.c, myinstoutorder.c, mymemory.c, Mysim.c, myinst.c, mysyscall.c, and myloader.c, as well as supporting files myloader.h, mymemory.h, Mysim.h, Mysimoutorder.h, and mysyscall.h. mymain.c handles launching of the simulator and the command line flags. mymemory.c handles the simulator's virtual memory, and myloader.c loads the benchmark program into the virtual memory. mysyscall.c handles the system calls made by the benchmark programs. The main simulation is divided between Mysim.c (for the functional and pipelined processors) and Mysimoutorder.c (for the superscalar processor) on one hand, modeling the datapath, and myinst.c (functional and pipelined) and myinstoutorder.c (superscalar) on the other hand, modeling the instruction set. This separation makes it possible to change the instruction set of the simulator without having to make major modifications to the simulator itself.

This section covers the workings of each of these component files in greater depth.

4.1.2.1. Starting simulation

The simulator is launched using the command line:

mysim <flags> <benchmark program> <benchmark parameters>

Simulation launching is performed primarily in mymain.c and myloader.c. mymain.c calls additional initialization functions in mymemory.c, mycache.c, mybpred.c.

4.1.2.1.1. mymain.c

The file mymain.c contains the code to launch the simulator. It handles command line flags, fatal exceptions, and prints out runtime statistics when the simulation terminates.

Most simulation variables can be adjusted via command-line flags. These flags determine the simulation type, size and latency of the cache, branch predictor parameters, and the parameters of the out-of-order pipeline. A full listing of the flags occurs as comments in mymain.c.

Three flags are intended to aid in debugging: -ti, -mk, and -di. The flag -ti is used to specify the total number of instructions to be executed. Simulation is terminated, and the statistics are printed, after that many instructions are completed. In the superscalar simulator, where more than one instruction may be committed in a cycle, simulation terminates on the cycle when that total number of instructions is reached.

The flag -di specifies a number of instructions to be executed until a debugger is launched. The simulator runs normally until the specified number of instructions is reached. The simulator then prints out, per cycle, the contents of all the registers and the

pipeline contents, and allows the user to single step the program cycle-by-cycle (by pressing Enter). The debugger is implemented in Mysim.c and Mysimoutorder.c.

The flag -mk takes a number *n*. On the completion of every *n*th instruction, at the end of the cycle, the current PC contents and most recently completed instruction is printed out. This flag is used for validating the simulator.

The function fatal() is called when a fatal error occurs in simulation. It prints out an error message and terminates simulation without printing statistics. Fatal exceptions are used principally in dynamic memory allocation; if the memory needs of the simulator exceed the memory available to it, a fatal exception is caused. They were also used on invalid or unimplemented system calls, and in verifying the validity of the simulator.

The function exit_routine() is called when the benchmark program ends naturally with the appropriate system call, or the number of instructions specified in -ti is reached. It prints out simulation statistics, including memory usage, cache miss rates, and the number of cycles needed. It also calls statistic printout routines in the supplemental simulator files, such as power usage, and value and branch prediction miss rates.

When the simulator is launched, mymain.c first loads in the appropriate flag values. It then calls mymemory.c to initialize the page table and virtual memory, mycache.c to initialize the cache (if there is one), and mybpred.c to initialize the branch target buffer (if one exists). myloader.c is then called to load the benchmark program into virtual memory, and finally the appropriate function (dofuncsim(), doinordersim(), or dooutordersim()) is called in either Mysim.c or Mysimoutorder.c to start simulation.

4.1.2.1.2. myloader.c

myloader.c reads the benchmark program machine code into memory. It is one of the two simulation files largely inspired by SimpleScalar, although little code was copied verbatim. Loading occurs entirely in one function: load().

The loader first reads the sizes and starting locations of the data and stack segments, and determines the appropriate virtual address for the stack pointer. It then reads the benchmark code into the virtual text segment, and the global data into the virtual data segment. When loading has concluded, the remaining command line parameters are written onto the virtual stack.

4.1.2.2. Simulator Support Files

4.1.2.2.1. mymemory.c

The virtual memory is handled entirely in mymemory.c. Memory is dynamically allocated per-page as needed. A page-table tells whether a page of memory has yet been allocated. If the benchmark program requires more memory than the physical system can support, simulation terminates with a fatal error.

Memory is accessed per-byte with the functions memory_write() and memory_read(). Each function initially checks the page table to check whether memory is allocated for that page. If not, memory_addpage() is called to allocate memory. The function then performs the write or read.

Three additional functions are included to facilitate memory access. memory_read_word() reads four bytes of memory at a time. memory_write_array and memory_read_array copies a specified quantity of bytes between the virtual memory and an array.

4.1.2.2.2. mysyscall.c

POSIX calls made by benchmark programs are handled by mysyscall.c. Register r2 is used to choose the handling routine. The handling routines for most of these calls simply handles the call by making the appropriate actual POSIX call, and transferring the result to the simulated registers or virtual memory, as appropriate. The exception is system call 0x01, which is called to end the benchmark program, and terminates simulation. The handling routines have been implemented on an as-needed basis for the SPECINT benchmark suite. Many system calls have not been implemented because they are not needed by any of the benchmark programs; if one of these calls are made, simulation ends with a fatal error.

The system call handling routines in mysyscall.c have been largely copied, as needed, from the SimpleScalar simulator. The benefit of this is that the benchmark programs behave exactly the same running under Mysim as they do running under SimpleScalar. This aids in validating the Mysim simulator.

4.1.2.3. Functional Simulation

The functional simulator is contained in the functions functional_simulate() and dofuncsim() in Mysim.c, and doinstruction() in myinst.c.

Function dofuncsim() initializes the simulator, which in this case means setting the register contents to 0 and the PC and SP registers to the appropriate addresses. Function functional_simulate() repeatedly reads the 8-bit PISA instruction word from memory, calls doinstruction() to execute it, and prints out statistical information, as needed. The bulk of the simulation is performed in doinstruction(). The instruction word is parsed into fields, and the opcode field chooses the PISA instruction function. The

entire function for each instruction, including memory access, register decoding and writing, and system calls, are performed here.

4.1.2.4. In-order Pipeline Simulation

The remainder of Mysim.c and myinst.c implement the five-stage in-order pipeline simulator. The pipeline is initialized to empty in function doinordersim(). It then calls the simulation loop in function inorder_simulate(). The function calls functions to handle each of the stages, in reverse: writeback, memory, execute, decode, and fetch. It then updates the cycle statistic counter.

Function stage_fetch() loads the 8-bit instruction word into the fetch-decode register using two memory reads. The PC is used to read memory, except when a branch prediction flag is triggered, in which case a speculative PC is used. The fetch is stalled, and a NOP instruction is copied to the fetch-decode register, under two cases. First, if the instruction at decode is a load instruction, fetch is stalled to prevent potential data hazards (this is determined by a call to function fetch_check_loads()). Second, if the branch predictor is disabled, fetch is stalled after a branch instruction. The fetch stage also determines the next PC value. PC+8 is assumed; however, a call is made to dofetch() in myinst.c to check for branches. In dofetch(), the opcode field is parsed. If the instruction is a conditional branch, the branch predictor is called to speculatively determine a branch direction; if it is any form of branch, other than “j”, the branch target buffer is read to speculatively choose a new PC.

In stage_decode(), the register fields are parsed, the register values are read into the decode-execute register, and data forwarding is performed as needed. decode_data_forwarding() is called to detect data hazards between the registers sourced

by the execute and memory instructions, and the registers read by the decode instruction. In case of a data hazard, one or more values in the decode-execute register are overwritten. Finally, dodecode() in myinst.c is called. This function determines if a branch misprediction has been made. If so, it determines the correct target address, and converts the fetch instruction to a stall.

In stage_execute(), a call is made to function doexecute() in myinst.c, which performs most of the execute work. Function doexecute() performs the arithmetic or address calculation required using the decode-execute register, and stores the result in the execute-memory register. Function stage_memory() also simply makes a call to domemory() in myinst.c, which performs the appropriate memory loads and stores.

Function stage_writeback() first checks if the instruction at writeback is a system call. If so, it performs the appropriate system call with handle_syscalls() in mysyscall.c, and calls pipeline_flush() to clear the entire pipeline. It next writes the memory-writeback register contents to the appropriate general purpose register, and finally gathers statistics.

4.1.2.5. Superscalar Simulation

Because of the size of the superscalar simulator code, it is located in separate files from the pipeline and functional code. The superscalar simulator is contained in files Mysimoutorder.c and myinstoutorder.c.

The superscalar simulator contains several internal storage components. A reorder buffer holds all of the instructions that have been dispatched but have not yet been committed. A set of reservation stations hold all instructions, one per station, which have been dispatched but have not yet finished executing. A ready queue holds a list of

reservation stations containing instructions that have not yet executed, but have all of their source values and are ready to be executed. A dispatch queue holds all instructions that have been fetched but have not yet been assigned a reservation station.

Several flags allow processor features to be modeled that do not require additional storage components. Global flags fu_integer, fu_integer_multdiv, fu_float, and fu_float_multdiv hold the number of functional units that are available (not in use) to handle integer arithmetic, integer multiplication/division, floating point, and floating point multiplication/division respectively. The busy flag, belonging to each reservation station, tells the status of the instruction in that station. The contents of the flag tells whether the instruction is waiting on the result of another instruction (3), ready to execute and waiting on a functional unit to become available (2), currently executing (1), or finished executing and waiting for writeback (0). A flag is also associated with each source register for each reservation station. If negative, the flag tells whether the operand value is available (-1) or not needed (-2). Otherwise, the flag holds the number of the reservation station sourcing that operand.

Function dooutordersim() initializes the registers, processor components, and starts the out-of-order simulator. The registers are initialized in the same way as in the functional and pipelined simulators. Since the size of the dispatch queue, ready queue, and reorder buffer, and the number of reservation stations can be set with command line flags, these processor components are dynamically initialized at this time. The number of each type of functional unit is also determined here from command line flags. When the processor components have been constructed, the function outorder_simulate() is called to perform the simulation.

Function outorder_simulate loops, calling the following functions in this order to perform the simulation: ooo_commit(), ooo_writeback(), ooo_execute(), ooo_issue(), ooo_dispatch, and ooo_fetch(). It then gathers cycle statistics and makes calls to the power modeler if it is enabled.

Function ooo_fetch() reads PISA instructions from memory into the dispatch queue. Instructions are fetched starting from the current address in the program counter, regardless of whether it is valid. The number of instructions read is determined by a parameter "fetches_per_cycle", which is set by a command line parameter. In no case are more instructions fetched, however, than there is room for in the dispatch queue. Fetching may also be limited by cache misses. The memory cycle latency is determined from a function call to cache_access_latency() in mycache(). If the result is greater than 1, no fetch takes place; instead, the latency is stored in a counter which is decreased each cycle. When the counter reaches 0, the fetch can occur.

Function ooo_dispatch() removes instructions in order from the dispatch queue and assigns each instruction a reservation station. It is limited by the number of free reservation stations and reorder buffer entries available. For each instruction, if there is a reservation station ready and the reorder buffer is not full, the instruction is assigned to that reservation station and its busy flag set to 3 (operands not ready), and it is copied to the bottom of the reorder buffer. If the instruction is a load or store instruction, it is copied to the tail of the load/store queue to ensure that loads and stores occur in program order.

Next, function set_resstat_registers() is called to detect dependencies. This function first checks, for each source register, if any instruction in another reservation

station is sourcing that register. If so, the number of that reservation station is stored at the decode instruction. If not, the function then checks if any instruction in the reorder buffer, but not in a reservation station, sources that register. If so, the value produced by that instruction is forwarded to the decode instruction. If not, the function copies the value in that register to the decode instruction.

Finally, ooo_dispatch() checks for branches. If the instruction is a branch, one of several things happens. If the instruction word contains the target address (such as in a "j" or "jalr" instruction), the target address is copied to the program counter, and the dispatch queue is emptied. Otherwise, if the branch is unconditional but the target address is unknown (such as in a "jr" instruction), the branch target buffer is read to produce a speculative PC value, the dispatch queue is cleared, and a flag is set at the reservation station to signify that all subsequent instructions are speculative. If the branch is conditional, the branch predictor is accessed (if enabled). The branch predictor outcome determines whether the branch target buffer is read. If not (branch assumed not taken or no branch predictor is simulated), the speculative flag is raised but the dispatch queue is not cleared. If so (branch assumed taken), the speculative flag is raised, the branch target buffer is read to determine the next PC, and the dispatch queue is cleared.

Function ooo_execute() handles each instruction while it is in the reservation station. It searches through each reservation station until it finds one that is occupied. If the instruction's busy level is 3 (not all operands are available), it checks whether any operand is still waiting on another instruction. If not, it upgrades the busy level to 2, and adds that instruction to the ready queue if it requires a functional unit. If the instruction

does not require a functional unit, and is not a memory instruction, it is further upgraded to busy level 1.

Next, if the instruction has a busy level 2 (operands ready), and is a load instruction, it is executed if it is at the front of the load/store queue, or if there are no stores preceding it in the load/store queue. If so, its busy level is set to 1. The memory cycle latency is determined using function dcache_access_latency() in mycache.c, and is put into a counter, which is decremented on each cycle.

If the instruction has a busy level 1 (executing or waiting on memory), its "time_left" counter is decremented. When that counter reaches 0, if it is a load instruction, the memory access is performed and the busy level is set to 0. If it is an arithmetic instruction, the instruction is executed by calling ooo_doexecute() in myinstoutorder.c. This function performs the instruction execution, and copies the result to a field in the instruction's reservation station entry. Then the appropriate functional unit count is incremented, to signify that the instruction's functional unit is available again, and the instruction's busy level is set to 0.

If the instruction has a busy level of 0 (completed), ooo_execute() checks whether the instruction is a branch instruction and the speculative flag is set. If so, it checks whether the result matches the speculative next program counter value. If not, ooo_squash() is called to remove all instructions following that branch in the reorder buffer from the pipeline, and the dispatch queue is flushed. Regardless of whether squashing occurs, the branch predictor and branch target buffer are both trained.

Function ooo_issue() assigns functional units to instructions in the ready queue. The ready queue is searched in order. If an instruction in it desires a functional unit

whose type is available, that instruction is removed from the ready queue, its busy level is set to 1 (executing), and the available functional unit count of that type is decremented.

Function ooo_writeback() removes instructions whose busy level is 0 (completed) from the reservation stations. It calls function write_to_reservation_inputs() to copy the completed value to any other reservation stations who depend on it. It then removes the instruction from the reservation station (but not from the reorder buffer), and copies its result value to a field in the reorder buffer entry. If the instruction is a store instruction, and at the head of the load/store queue, its memory latency is determined and a counter set.

Function ooo_commit() writes instruction results to memory and registers in order. It starts at the head of the reorder buffer and tries to commit as many instructions as possible. When it reaches an instruction that cannot be committed, the stage ends.

An instruction is committed if it reaches the head of the reorder buffer and is not located in a reservation station. The instruction is removed from the reorder buffer and its value written to the appropriate register. If the instruction is a store instruction, however, it is not removed until its memory latency counter reaches 0, only after which its value is written to memory. If the instruction is a system call, ooo_squash() is called to remove all subsequent instructions from the pipeline. Finally, ooo_commit() gathers cycle statistics.

A supplemental function, ooo_squash() removes all instructions from the pipeline after the reorder buffer stage given as a parameter. It removes the instructions from the reservation stations, then from the load/store queue and ready queue, and increments the count of any functional unit it occupies. It then clears the dispatch queue.

4.1.3. Mysim Extensions

Mysim contains four extensions, not including the additional extensions documented later in this dissertation. These extensions are: a cache, simulated in mycache.c, a branch predictor and branch target buffer, simulated in mybpred.c, the Wattch power modeler, implemented in wattch-power.c, wattch-time.c, and wattch-power.h, and a clock frequency simulator, simulated along with the superscalar processor in Mysimoutorder.c. These extensions, apart from the branch predictor, are used only by the superscalar simulator.

4.1.3.1. Cache

A two layer data cache and a two layer instruction cache, of variable size, are modeled in mycache.c. The cache is organized set associatively, the number of sets, ways, and block sizes being determined through command line flags. It uses a writeback policy on stores for dirty cache blocks and a least recently used replacement policy. While cache accesses are performed immediately upon request, the modeled cache does have a function to estimate the cache latency of a memory request. To model a cycle accurate cache, the processor simulator first requests the latency of the cache access. It then delays the memory access for the appropriate number of cycles, as described earlier, after which it performs the actual cache access.

The cache need not be enabled. If the cache is disabled, all memory accesses are assumed to have a single cycle latency. All memory accesses in Mysimoutorder.c are made to functions in mycache.c, not to mymemory.c, whether the cache is enabled or not. If the cache is disabled, the cache functions simply call the appropriate function in mymemory.c.

The cache is initialized in mymain.c by a function call to cache_init(). This function dynamically allocates memory for the cache. The function uses several parameters obtained through the command line to determine the size and structure of the cache. Parameters il1_cache_size, il2_cache_size, dl1_cache_size, dl2_cache_size determine the total number of bytes total stored in each cache layer. Parameters il1_cache_sets, il2_cache_sets, dl1_cache_sets, and dl2_cache_sets specify the number of sets in each cache layer; the number of ways are calculated from the size and number of sets. Parameter cache_block_size specifies the number of bytes in a cache block; for simplicity in modeling the cache, all cache layers in Mysim's cache have the same block size. Function cache_init() constructs several arrays for each layer: a cache array containing the actual data, a tag array holding the upper bits of the block address for each set, a valid array holding whether each cache set contains a valid block, a dirty array telling whether each cache block has been written to by a store instruction, and a replace array holding LRU information for each cache block. The entire cache is initialized to empty (invalid).

Cache accesses by the processor simulator are performed by calling the functions dcache_read(), dcache_write(), icache_read(), and icache_write() with a memory address. These functions simply check whether the cache is enabled; if not, they simply call the appropriate functions in mymemory.c. If so, they call the respective functions d_cache_doread(), d_cache_dowrite(), i_cache_doread(), and i_cache_dowrite(). Each function works largely the same. The L1 cache is searched for the address. If found, the data is returned and the LRU bits for each block in the set is updated. If not found, i_cache_miss() or d_cache_miss() is called to search the L2 cache. The LRU block is

evicted from the L1 cache, if necessary, by calling il1_cache_evict() or dl1_cache_evict(), and is copied to the L2 cache if dirty. If the data is found in the L2 cache, the appropriate block is copied to the L1 cache. If not, eviction occurs at the L2 cache by calling il2_cache_evict() or dl2_cache_evict(), if needed. A memory access is then performed by calling memory_read() or memory_write() functions in mymemory.c, and the block is copied into the L2 cache and to the L1 cache. Meanwhile, statistics are gathered on the number of cache hits, misses, and replacements for each layer.

The cycle latency for a cache access is calculated by calling icache_access_latency() or dcache_access_latency() for a particular address. The functions check whether the address is present at the L1 and L2 caches to determine whether there will be a miss. It then uses hit and miss latency parameters specified by command line flags to determine the cycle penalty of the memory access.

4.1.3.2. Branch Prediction

A branch target buffer and several varieties of dynamic branch predictors are simulated in mybpred.c. The branch target buffer is organized as a direct-mapped table indexed using the lower bits of the branch instruction address. It is initialized by a call to BTB_init() before simulation, which dynamically allocates memory for the BTB based on a command line parameter. Function get_BTB() reads the predicted target address for a given branch instruction address. Function update_BTB() is called after the branch instruction is resolved; it updates the BTB with the actual target address.

The branch predictor type and size is determined through command line parameters; it is also initialized in function BTB_init(). Function get_branch_prediction(), called with the branch instruction address, returns a prediction.

Function train_branch_predictor() is called after the branch instruction is resolved; it updates the branch predictor.

Five branch prediction strategies have been implemented, and there is an option to disable the branch predictor (in which case the pipelined simulator simply stalls, and the superscalar simulator assumes branch not taken). The implemented strategies are: 1) predict the same as the last global branch, 2) predict the same as the last local branch, 3) always predict “don’t take”, 4) always predict “take”, and 5) bimodal. The bimodal branch predictor associates a two-bit saturating counter for each branch table entry which is decremented when the branch is not taken and incremented when it is. A counter value of 2 or 3 predicts “taken”, a value of 0 or 1 predicts “not taken.”

4.1.3.3. Power Modeling

The Sim-Wattch power modeler has been adapted to work with the Mysim superscalar simulator. Power modeling is performed in files wattch-power.c, wattch-power.h, and wattch-time.c. Calls are made to the power modeler in Mysimoutorder.c. The power modeler estimates the total dynamic energy consumption of the processor over the course of simulation by monitoring the number of times that various pipeline events occur.

If power monitoring is enabled, before simulation, the energy consumption of a large variety of pipeline and cache events is calculated. This takes as a parameter, given on the command line, the clock frequency of the processor. During simulation, whenever a pipeline event occurs for which energy consumption was calculated, a counter associated with that event is incremented. At the conclusion of simulation, the counters

are used to estimate the total energy consumption of the processor and cache, and the energy consumed in each stage.

Because the power modeler uses the clock frequency as a parameter, the processor's clock frequency can be specified on the command line. This frequency is used to determine the total simulated time to run the benchmark application. Mysim also supports frequency scaling. If specified as a parameter, the energy consumption is calculated before simulation for two different clock frequencies. At runtime, the frequency of the processor can be switched by calling function set_clock_speed(). Two sets of event counters are kept, one for each frequency. The total energy consumption at the end of simulation is then calculated by multiplying each set of counters by the appropriate energy rates. Additionally, the total simulated execution time is calculated using the amount of time spent on each frequency. This allows dynamic frequency scaling approaches to be studied using Mysim.

4.1.4. Mysim Validation and Performance

To prove that Mysim works correctly, eight benchmarks programs were run on both Mysim and the SimpleScalar PISA simulator. Several tests were performed to show that the benchmarks ran the same on both processor simulators.

Table 4.1 shows the eight benchmarks from the SPEC2000 integer suite and their input sets. The benchmarks were compiled by the PISA gcc compiler which comes in the SimpleScalar simpletools package. The benchmarks were run to 1 billion dynamic instructions or conclusion, whichever came first (mcf was the only one to terminate before 1 billion instructions).

256.bzip2	input.graphic
176.gcc	166.i
164.gzip	input.program
181.mcf	inp.in
253.perlbmk	-I/lib makerand.pl
300.twolf	ref
255.vortex	lendian.raw
175.vpr	net.in arch.in place.out dum.out -place_only -init_t 5 -exit_t 0.005 -alpha_t 0.9412 -inner_num 2

Table 4.1. Benchmarks and parameters

Several tests were performed to prove accuracy. The register contents at the end of each instruction for the first 1 million instructions were saved in a log file for both SimpleScalar and Mysim. The log files for each benchmark were compared and found identical. The register contents and instruction word were then saved to a log file for each 100th instruction thereafter up to 1 billion instructions for both simulators. These were compared for each benchmark and found identical, except on certain benchmarks (perlbmk) which made system calls to get the time of day. However, when the system calls handlers were modified to always return the same results, the register results were found to be identical on each 100th instruction. These tests were performed under the simulation parameters shown in Table 4.3.

Executing the benchmark program correctly does not necessarily mean that the simulator is simulating cycle accurately. Figure 4.2 shows the IPC for Mysim and Simplescalar 2.0 at the parameters in Table 4.3, for 500 million instructions across the benchmarks. The average absolute difference in IPC is 4.94%.

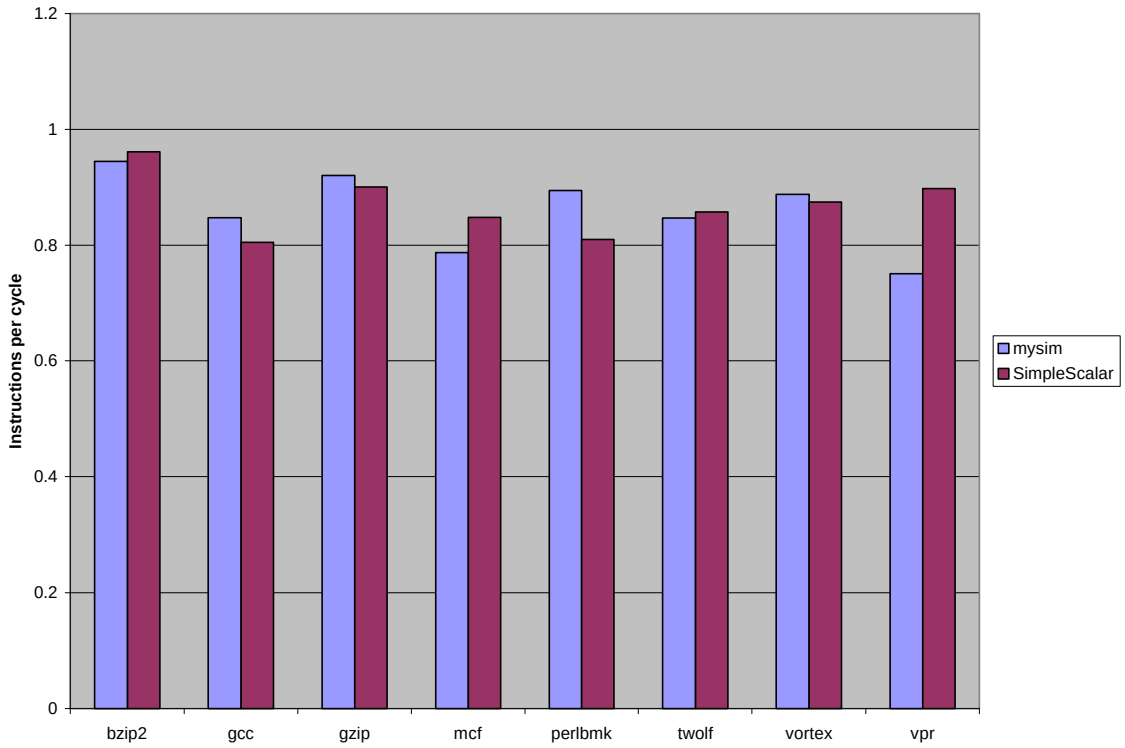


Figure 4.2. IPC compared for Mysim and SimpleScalar

Because Mysim is a true execution driven simulator and is not implemented as efficiently, it runs more slowly than SimpleScalar. Table 4.2 shows the relative simulation time increase for 500 million instructions for Mysim over SimpleScalar, using the parameters in Table 4.3.

bzip2	2.0
gcc	1.8
gzip	17.0
mcf	21.4
perlbmk	16.0
twolf	10.5
vortex	13.6
vpr	9.7

Table 4.2. Relative simulation time of Mysim over Simplescalar

4.2. Simulation Methodology

In this dissertation, all quantitative results, unless otherwise specified, were obtained using the Mysim superscalar simulator. Results were obtained by running the

benchmarks and input sets in Table 4.1 to 500 million instructions or completion, whichever comes first (benchmark mcf terminates at 173 million instructions, the rest run for a complete 500 million). Unless otherwise specified, Table 4.3 show the default simulation parameters for the processor, cache, and extensions.

Decode width	16 instructions
Reservation Stations	128
ROB Size	128
Load/Store Queue Size	64
Functional units	4 integer add/sub, 2 integer mult/div, 2 fp add/sub, 1 fp mult/div
Functional unit latencies	Integer add/sub: 1 cycle, all others: 10 cycles
Inst and data L1 cache	64k, 512 sets, 64 byte block size, LRU, 1 cycle latency
Inst and data L2 cache	1M, 8192 sets, 64 byte block size, LRU, 6 cycle latency
Branch predictor	Bimodal, 1024 entries

Table 4.3. Simulation parameters

4.3. Simulating Perceptrons

Perceptrons are implemented in this dissertation in files myvpred.c and mycritical.c. Perceptrons are separately implemented for each of the value predictors and criticality predictors; however, most of them share the same training function *train_perceptron_weight* to implement the training procedures to train an individual weight. All perceptron implementations consist of three functions: at *get_* function, a *train_* function, and an *initialize_* function.

The *initialize_* function is run at the beginning of simulation. It dynamically allocates storage for the perceptron weights for each entry of the perceptron table and initializes each weight to 0.

The *get_* function is run when a prediction must be obtained, typically at fetch or dispatch. It inputs the global history to the perceptron, performs the dot product and

threshold, and returns a decision. The function also stores the global history in the table entry alongside the weights so that it can be used in training.

The *train_* function is run as soon as the information is available to train, which is at execution for value prediction and commit for criticality prediction. It first compares tags to ensure that the prediction table entry is valid. Next it obtains the prediction by reproducing the prediction in *get_* Third it calls *train_perceptron_weight* using the input, pointer to the weight, prediction, and correct value. Fourth it updates the global history by calling *update_global_history_table* with the correct value. This function inserts the value into the history according to the anti-interference approach, either by shifting the correct value into the history register, or by assigning it a location in the history register based on its instruction address. Finally, *train_* updates accuracy statistics so that the approach can be evaluated.

The following is the pseudocode for obtaining a prediction:

```
1.  table_entry = (PC >> 3) MOD table_size
2.  sum = 0
3.  for i = 1 to history_size
4.    if global_history[i] = 1 then
5.      sum = sum + table[table_entry].weight[i] * 1
6.    else
7.      sum = sum + table[table_entry].weight[i] * -1
8.  sum = sum + table[table_entry].weight[BIAS] * 1
9.  if sum > 0 then
10.   return 1
```

```
11. else
12.   return 0
```

The following is the pseudocode for training the predictor using training-by-error:

```
1.  table_entry = (PC >> 3) MOD table_size
2.  if predicted = 0 AND actual = 1
3.    error = 1
4.  else if predicted = 1 AND actual = 0
5.    error = -1
6.  else
7.    error = 0
8.  for i = 1 to history_size
9.    if input[i] = 1 then
10.     table[table_entry].weight[i] += error
11.   else
12.     table[table_entry].weight[i] -= error
13. table[table_entry].weight[BIAS] += error
```

The following is the pseudocode for training using training-by-correlation:

```
1.  table_entry = (PC >> 3) MOD table_size
2.  theta = 1.93 * history_size + 14
3.  for I = 1 to history_size
4.    if input[i] = actual then
```

```
5.      table[table_entry].weight[i] += 1
6.      if table[table_entry].weight[i] > theta then
7.       table[table_entry].weight[i] = theta
8.    else
9.      table[table_entry].weight[i] -= 1
10.     if table[table_entry].weight[i] < -theta then
11.       table[table_entry].weight[i] = -theta
12. if actual = 1 then
13.   table[table_entry].weight[BIAS] += 1
14.   if table[table_entry].weight[BIAS] > theta then
15.     table[table_entry].weight[BIAS] = theta
16. else
17.   table[table_entry].weight[BIAS] -= 1
18.   if table[table_entry].weight[BIAS] < -theta then
19.     table[table_entry].weight[BIAS] = -theta
```

Chapter 5: Value Prediction

5.1. Introduction

In this chapter I will look at using perceptrons to improve the accuracy of data value prediction. Value prediction was proposed nearly ten years ago as a way of speculatively removing data dependencies in superscalar processors. A value predictor allows instructions that are dependent on the result of a long latency instruction to execute by guessing the outcome of that instruction and feeding that guess to dependent instructions. These dependent instructions can then execute simultaneously with their parent. The guess is, of course, verified when the parent instruction finishes execution. If the guess is correct, the dependent instructions are permitted to commit; otherwise, they must be executed again.

Accurate value prediction may be counter-intuitive, considering the quantity of different possible values that could be produced. However, prediction is possible because data values used by programs often follow easily discernable patterns. Prior research has demonstrated the existence of value locality, or the reuse of data values in a program. In general, a given section of a typical program has a small quantity of data values that it reuses over and over again [Lip96]. Value predictors focus on observing patterns in this value reuse to guess the data value that will be produced by a given instruction.

The original work in value prediction was focused solely on predicting the results of load instructions, particularly those that are undergoing a cache miss [Lip96_2]. With memory latencies ever increasing, load value predictors remain attractive. Subsequent work extended the research to predicting the results of any

long latency instruction, such as floating point arithmetic, multiplication, and division, which can be valuable for computation-intensive programs [Lip96]. More recent work has extended value prediction research to multithreaded or multiprocessor architectures [Mar99, Tuc05].

While data value prediction has attracted a fair amount of research, it has yet to be widely implemented in actual processors. There are two reasons that are most likely to be responsible for this. The first is that value predictors that have been proposed so far that are feasible to implement typically have fairly poor accuracy rates, ranging broadly from 30 to 80%, depending on the benchmark and processor characteristics. The second reason is that it is difficult to reexecute dependent instructions without high performance penalties. A highly accurate value predictor might be able to withstand high misprediction cycle penalties, while a good misprediction recovery method might be able to allow a low accuracy value predictor to produce performance gains. However, the combination of these two problems presently hinders the actual construction of value predictors.

There is, however, hope for value prediction. Previously proposed value prediction strategies have typically captured only a part of the existing value locality. Traditional table-based predictors have difficulty observing value patterns stretching globally between instructions, without becoming too massive to be implementable. Alternative value prediction strategies have already shown themselves to have higher prediction accuracy rates than the table-based approaches. However, in many cases these strategies are either themselves impractical, or capture only part of the global value locality.

There are several reasons why I choose to apply perceptrons to value prediction in this dissertation. First, perceptrons can capture global value correlations that a table-based predictor cannot capture, allowing potentially greater prediction accuracy. Later in this chapter, I will explore the reasons why this is the case. Second, value prediction has several characteristics in common with branch prediction that make a similar perceptron approach promising: prediction times must be low latency, predictions are made by instruction requiring a per-address framework, some past values correlate while others do not, and so on. Third, value prediction requires the learning of correlations between whole data values, instead of correlations between individual binary decisions. The perceptron model used in branch prediction consequently cannot be directly applied to value prediction. Thus a novel approach is required, giving further insights into the perceptron.

In this chapter I will present four basic perceptron approaches to value prediction. The first approach is a local approach that makes a prediction using information solely from previous instances of the instruction under prediction. This approach directly replaces the previous table-based approaches. The second approach uses global information to predict a local value; it can only predict a value previously seen locally, but it uses information from other instructions to choose that value. The third approach uses both global information and global past values to make a prediction. The fourth approach is a bitwise prediction approach that does not explicitly predict a past data value. Instead it tries to detect correlations between individual bits of past data values to potentially predict new data values.

5.2. Value Prediction Background

5.2.1. Groundwork and Local Predictors

The pioneering work in value prediction was done largely by Mikko Lipasti [Lip96, Lip96_2]. These papers respectively cover the prediction of load instructions and the prediction of all data producing instructions. The papers were not primarily focused on producing a viable prediction framework. Rather, the papers focused on quantifying value locality between instructions, and arguing the merits of creating a predictor to break the data dependencies.

Sazeides and Smith authored the first highly significant follow-up work [Saz97_2]. This work examined actual value prediction strategies. The authors broke value prediction strategies into two broad categories: context-based predictors which predict data values that have been seen before, and arithmetic predictors which detect mathematical sequences in past data, applying a mathematical function to past data values to produce potentially new data values. They proposed two very general prediction strategies, one for each category.

The arithmetic predictor proposed was the local stride predictor. This predictor uses the difference between the last two data values produced by a static instruction to compute a stride. This stride is then added to the last data value to make a prediction. The stride predictor is thus able to detect monotonically increasing or decreasing patterns over repeated instances of an instruction. It is easy to see why such a predictor would be powerful. The typical for-loop iterator, for example, is incremented by 1 each iteration of the loop. Consequently, each instruction producing that iterator could be predicted by a stride predictor.

Additionally, the stride predictor can predict that trivial but all-too-common case where a particular instruction produces the same value on every iteration. Subsequent major value prediction works have never explored arithmetic prediction strategies beyond the simple stride. The reason is simple: no non-stride arithmetic pattern is sufficiently common in typical programs to justify a more advanced arithmetic predictor.

Of more interest in this work is Sazeides and Smith's context-based predictor, clearly inspired by the successful two-level branch predictor. Their approach is simple in theory but less so in implementation. Their two-level context-based predictor keeps track of the local value history: the past values produced, per-instruction. When a prediction must be made for a given instruction, these local past values, or some subset of them, is hashed, and the hash is used to index a pattern table. This pattern table entry holds the last value seen for that hash, which is put forth as the prediction. Discounting the effects of aliasing (which must be significant for any conceivable practical implementation), the context-based predictor can learn any repeating local value pattern.

A subsequent paper by Wang and Franklin proposed a more well-specified value predictor [Wan97]. It is a hybrid predictor that combines a variation on the Sazeides and Smith context-based predictor with a stride predictor. This paper is highly significant for several reasons. First, it is the first paper to propose a hybrid arithmetic / context-based predictor. Second, and much more importantly, it is the first to propose a well-defined, reproducible, feasible context-based value prediction strategy. The result is that this predictor has been informally adopted as the de-facto

baseline predictor in most subsequent value prediction research. Some later papers have claimed more accurate predictors, but none have been as widely adopted for use in research comparisons. I will go into more detail on this predictor later in the chapter, and discuss its strengths and weaknesses.

5.2.2. Global Predictors

The chief weakness of the above prediction strategies is that they are limited to predicting from the local value history, and disregard the correlations that can be made globally between the values produced by different static instructions. A few notable works have attempted to look at global value prediction.

The first was a paper by Nakra, Gupta, and Soffa [Nak99]. Among the contributions of this work was a value predictor that attempted to apply the context-based predictor framework globally. The predictor was only meant as a theoretical study, and the authors admitted that it is not a practical predictor.

A paper by Zhou, Flanagan, and Conte [Zho03] proposed a global stride predictor that detected strides between the values produced by past dynamic instructions. Despite some hardware complexity, the authors claimed that their predictor was able to achieve high accuracy rates of between 35% and 80%, especially when compared to the local stride predictor.

A study by Thomas and Franklin [Tho01_2] into the reasons behind the limits of local prediction strategies resulted in an innovative path-based predictor. The authors used a unique index generated from the dependence history of the target instruction to index the pattern history table. This predictor could consequently

capture local value patterns that were previously unpredictable due to unpredictable control flow changes.

5.2.3. Simulating Value Prediction

A weakness of much past value prediction research is the fact that few authors actually tested their value prediction strategies on a cycle-accurate simulator. Typically, value prediction results were given in terms of prediction accuracy. This is due to the problems of dealing with incorrect predictions.

In theory, ignoring the realities of the pipeline, a value predictor should have a negligible misprediction cycle penalty [Bur02]. Suppose a long-latency instruction A is followed by dependent instructions B and C. In a machine without value prediction, B and C are held up in dispatch until A completes execution. The next cycle, B and C are forwarded A's result and may begin executing. In a machine with value prediction, B and C may commence execution before A's execution is completed. If the prediction of A's result was incorrect, and the machine has an "ideal" misprediction handling mechanism, B and C can be restarted in the next cycle. Since B and C start in the same cycle whether there was no value prediction or an incorrect prediction was made, there is, in the ideal case, a zero cycle penalty for mispredictions. A value predictor could be simulated without worrying about the misprediction method by simply precomputing A, which tells whether the prediction will be accurate. If so, B and C are started immediately, if not, they are delayed for A's result.

This scenario omits the fact that even the most optimistic value predictor would need a cycle to verify that the prediction is incorrect, squash B and C, and

reschedule them for execution. Consequently, several value prediction papers attempt to estimate the performance effect of their value prediction approach by simply restarting B and C one cycle later on a misprediction. This estimate, however, ignores the structural problems of mispredictions, their effects on branches, and other pipeline issues.

There are really three practical ways that have been proposed for dealing with a misprediction: ReFetch, ReIssue, and ReExecute [Cal98, Bur02]. ReFetch throws the dependent instructions and their results out of the pipeline altogether, and fetches them again later. In reality, all instructions subsequent to a mispredicted instruction must be thrown out so that order is preserved in case of branch mispredictions and traps. ReFetch is fairly straight-forward to implement, but has a high performance cost on mispredictions. ReIssue puts dependent instructions back in the dispatch queue and ReExecute gives them a functional unit and executes them again. These, in theory, have lower misprediction penalties. However, they are surprisingly difficult to implement because of the problems of dealing with CPU resources that may be used by other instructions, preserving instruction ordering, and other concerns. ReFetch has been simulated by authors seeking realistic performance numbers [Cal98].

When simulating a practical misprediction-handling strategy, it is common to have a confidence estimator associated with the value predictor. The confidence estimator is itself a speculator and guesses whether the prediction is likely to be correct. If the confidence estimator chooses not to predict, the dependent instructions are forced to wait for the parent to finish execution, but no misprediction penalty is

incurred. An accurate confidence estimator can partially make up for an expensive misprediction handling policy. A value predictor that uses ReFetch for misprediction recovery but lacks a confidence estimator could very well reduce performance instead of increasing it!

Confidence estimators for value prediction have been included since the topic was first proposed and have been extant in nearly every serious proposed predictor [Lip96]. They have been independently studied in [Bur99].

5.3. Local context-based prediction

Why predict from the local value history in the first place? Before discussing local value predictors in depth, it is important to answer this question.

The big advantage to focusing on patterns in past local values to make predictions is that when local data values are predictable, they tend to be highly predictable. Past studies have shown that the local value history for many instructions consist of alternating values, repeating patterns, and strides, or even simply the same value over and over again [Saz97_2]. Consequently, there is no need for complicated prediction schemes; a simple predictor, reproduced for each instruction, can have high accuracies.

5.3.1. The two-level hybrid predictor

5.3.1.1. How it works

As I mentioned earlier, the value predictor proposed by Wang and Franklin [Wan97] has become the informally official value predictor used as a baseline in subsequent value prediction studies. Consequently, it is important to understand how

that predictor works and what its strengths and weaknesses are before considering novel value prediction approaches.

A representation of the predictor is shown in Figure 5.1. This value predictor is made of two tables (hence its name): a value table and a pattern table. The value table is organized per-instruction by instruction address and holds four data values for that instruction, chosen via a least-recently-used replacement strategy. Each data value has a two-bit index. The indices of the last four local data values are stored and, concatenated together, form an index to the pattern table (which consequently contains a fixed 256 entries). The pattern table consists of four up-down saturating counters, each corresponding to one of the data values in the value table. The highest counter value chooses the value to predict. If no counter value exceeds a certain threshold, no prediction is made - this acting as a form of confidence estimation. The predictor is trained when a correct value is known by incrementing the counter value in the pattern table corresponding to the correct value, and decrementing the other counters.

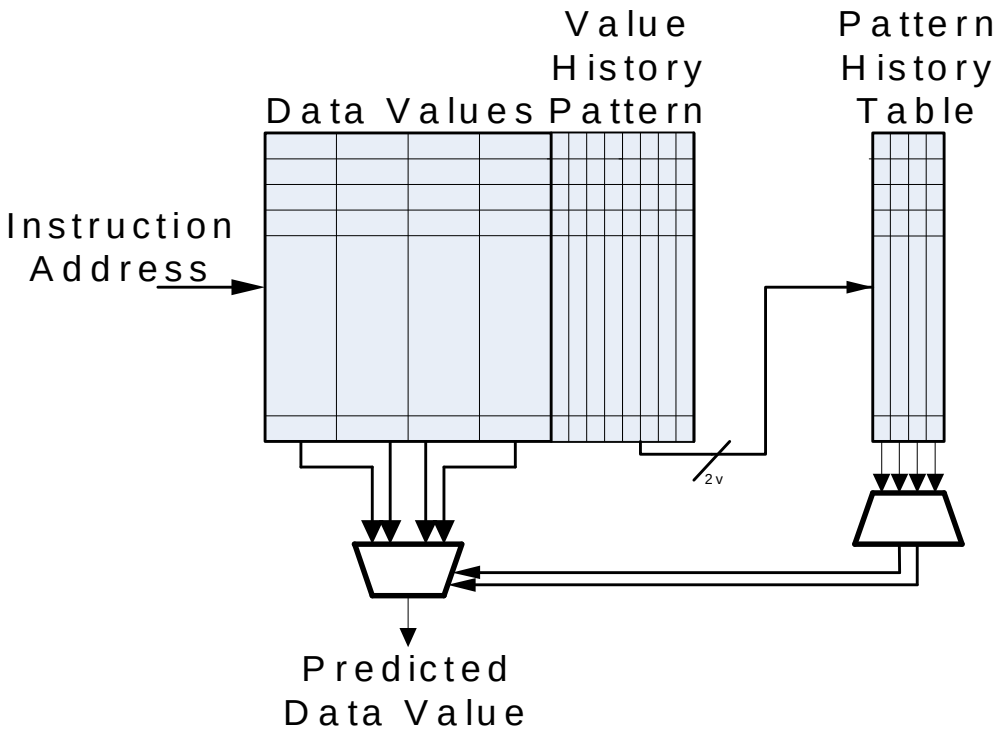


Figure 5.1. Table-based Value Predictor

A stride predictor is added on by storing a last value, and a stride between the last two static values in each value table entry. The stride is accompanied by its own confidence estimating up-down counter, which is later incremented if the stride is correct and decremented otherwise. If the pattern table chooses not to predict, but the stride counter chooses to predict, a stride prediction is made by adding the stride to the last value. If neither the pattern table nor the stride chooses to predict, no value prediction is made.

5.3.1.2. What it can and cannot do

The above value predictor has several advantages. First, it is reasonably small in size, especially the pattern table component. Second, it is able to detect any local

repeating pattern that repeats on every fourth value or fewer. It also has a number of limitations. It is local in scope, it scales exponentially, it is susceptible to aliasing.

First and foremost, the predictor is a local predictor. Apart from the stride, values predicted can only be from the set of values seen in previous instances of the target instruction. Additionally, no information is used to make the prediction other than the value patterns produced by that instruction. The data values produced by other instructions have no effect on how the value is chosen. I will examine the implications and limitations of local value prediction later.

Second, the predictor is unscalable. Dramatically so. The number of different past data values and the length of the value history are fixed at four. If the length of the value history is increased by one entry to five, the pattern table's size is quadrupled to 1024 entries. Additionally, with five history entries, there is the risk that one of the past data values cannot be indexed. If the history size is held at four, but an additional past value is added, the value index size must be increased to three bits, adding four extra bits to the pattern index, and multiplying the pattern table size by 16 to 4096 entries (it would also increase the size of each entry, but only linearly in this case). Raising both parameters to eight would increase the pattern table size from 256 entries to over 16 million entries, and raising to 16 would require an inconceivably massive pattern table of 2^{64} entries!

Third, the predictor is susceptible to aliasing in both the value table and the pattern table. Since the value table cannot have an entry for every conceivable instruction, only the last several bits of the address are used to index the table and a tag field is kept to detect aliasing and reset the instruction's entry if aliasing occurs.

This requires that the value table be kept fairly large (about 4096 entries were cited). More problematically, the pattern table suffers from aliasing between different instructions producing the same value index history. Although the resulting interference need not necessarily be destructive, the use of an LRU strategy in determining value index has the effect of making the aliasing interference potentially chaotic.

5.3.2. Perceptron-based local context predictors

5.3.2.1. Why perceptrons?

A perceptron predictor should have at least three major potential advantages over the table-based predictor. First, it does not suffer from exponential growth as the history size or the number of past values is increased. It can thus track repeating patterns of more than four values without an explosion in storage space. Second, as discussed earlier in Chapter 3, it can exclude noise from unpredictable past values and track a single, repeating past value. Although the table-based predictor can eventually detect all the possible patterns, the perceptron predictor should start predicting correctly earlier. Third, a perceptron predictor may be able to dispense with the second-level pattern table, thus eliminating a potential source of harmful interference. When designing a perceptron predictor, it is important to ensure that it enjoys all three advantages.

This potential benefit of a perceptron local predictor is, of course, built on several assumption about the past local data. First, if there are no repeating patterns of more than four values, a bigger history size will make no difference. Second, if a

past value repeats but not at regular intervals, noise exclusion will not matter. Third, if the interference in the second-level pattern table is not harmful, or is nonexistent, removing it will not help.

5.3.2.2. Perceptrons in the Pattern Table

In a short paper in a value prediction workshop, Thomas and Kaeli present a two-level perceptron value predictor directly modeled after the two-level table-based predictor [Tho04]. This is, to my knowledge, the only previously published perceptron-based value prediction approach. The work is flawed: the simulation parameters are largely undefined, it is unclear how the performance modeling was done, and the results are unexplained and fairly implausible, for reasons I will go into below. However, it makes a good starting point for designing a perceptron-based local context predictor.

In this approach, the two-level predictor scheme is kept intact. In each pattern table entry, however, the counters are replaced with a perceptron modeled directly after the perceptron branch predictor. Only two past values are stored in the value table, and the perceptron chooses between the two past values.

I take this approach one step further, using a multibit perceptron to choose between four values or more, allowing the perceptron approach to capture the same history size as the table approach. A block diagram of this predictor is shown in Figure 5.2. Predictions work as follows:

- Like the table-based predictor, four past values are stored for each entry. Each value is given a two-bit index. A Least-Recently-Used (LRU) replacement policy is used to choose values.

- A local value history (for now, also of size four) is stored for each entry. This history is expressed in terms of the indices.
- A perceptron is chosen from the pattern table. The pattern table is indexed by the concatenated bits of the value history, and consequently consists of 256 entries.
- The value index history bits are used as inputs to the perceptron.
- The perceptron output is an index, which is used to choose one of the past values.
- The perceptron is later trained with the index of the actual value. If the actual value does not exist in the local value history, the LRU value is replaced with the actual value, and the index of that LRU value is used to train the predictor, it being the actual value's index now.

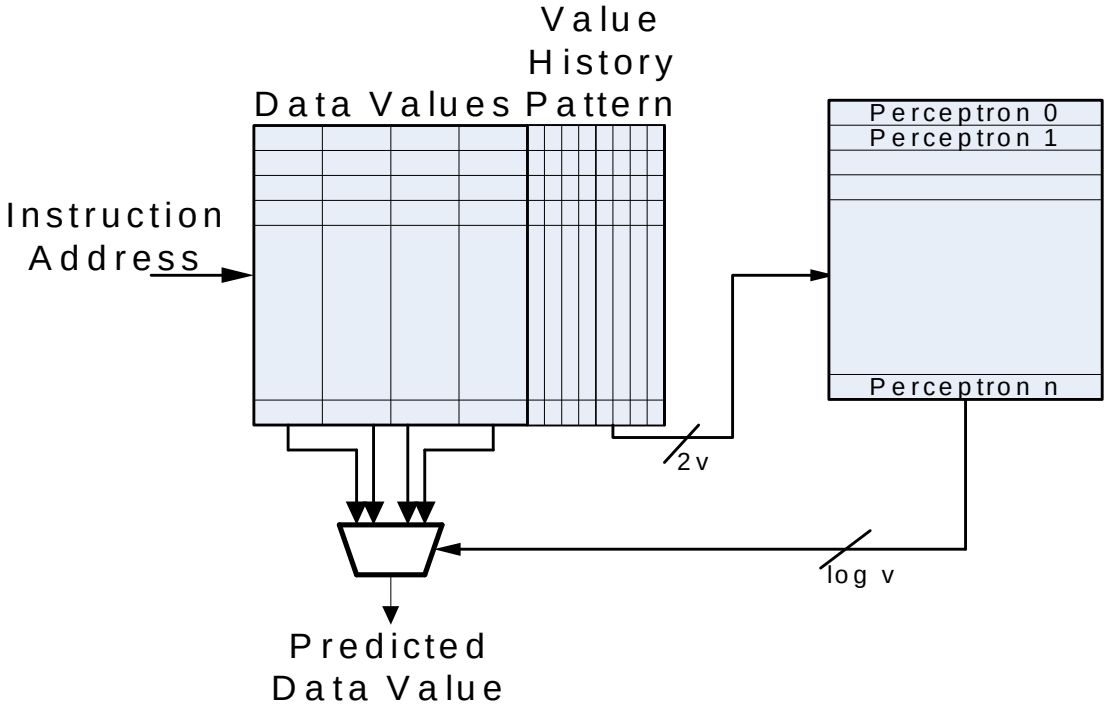


Figure 5.2: The perceptrons-in-the-pattern-table (PPT) predictor

The problems with this two-level perceptron approach should be apparent. First, it has largely the same scalability issues that the previous two-level approach had. An increase in the number of past values stored or in the value history size results in the same exponential increase in the pattern table size described earlier. Second, aliasing effects aside, the perceptron is always fed the same input. Since the concatenated value index history is used both to select the perceptron and as input to the perceptron, each perceptron will only ever see one input value. This unfortunately defeats the whole purpose of having a perceptron; a counter is smaller. The perceptron's capability as a pattern predictor is unused.

Fortunately, there are a few potential advantages to recommend this approach. The first advantage is that a larger value index history size can be considered. While the last four value indices are still used to choose a perceptron from the pattern table, a longer index history can be kept and used as input to the perceptron. The growth in this case is linear, as only the perceptrons' size, not the pattern table size, is changed. The last four values in the history can be excluded from the perceptron input as redundant. A problem does arise if the value history includes a value that is not one of the four past values stored. This can be worked around by including an extra bit in each value index history entry telling whether that entry is "valid" or "invalid." If an index is "invalid", a zero can be fed to the corresponding perceptron bits, eliminating that entry from consideration.

The second advantage is that, with a larger value index history size, the perceptron might handle pattern table aliasing better than the counter approach.

Recall that aliasing occurs when the concatenated value index history for two different value table entries is the same. Suppose that destructive interference occurs; a different index should be predicted for two different static instructions. Further suppose that the worst case scenario occurs, and the two instructions alternate repeatedly. An example of this occurs when two instructions are undergoing a repetitive sequence of five values. In the counter approach, no correct prediction can be made on the aliased history; the counters are always choosing the wrong value. The perceptron, however, could correlate on the fifth most recent value and differentiate between the two instructions.

A third advantage is simply one of size. The pattern table is small, with 256 entries. The value table is significantly bigger, needing at least 4096 entries to reduce the effects of aliasing between instruction addresses. Since a perceptron requires a non-negligible amount of storage, putting it in the smaller table makes for a more space and power efficient predictor.

5.3.2.3. Perceptrons in the Value Table

Clearly, simply replacing the counters with perceptrons is not likely to create the best perceptron-based local predictor. An alternative approach is to eliminate the pattern table altogether and let the perceptrons detect the patterns; after all, that is what they are intended for in the first place. This would require moving a perceptron into each value table entry. The perceptron would take as input the past data, or at least the indices of past data, and derive the local value pattern. For now, I will neglect the size considerations (which are not insignificant) of having a perceptron in each value history entry, and focus on how to maximize the prediction accuracy.

The big question is what to use as input to the perceptron. A first possibility is the actual past values themselves. On a 32 bit machine, this would require 32 perceptrons, each requiring 32 weights per history entry. Naturally, the size requirement would be huge. But is it worth it? There are two intuitive reasons to reject this approach. First, local value patterns do not tend to be subtle enough for such a complicated predictor; they tend to be easily predictable or unpredictable. Second, for reasons I explained earlier, a multibit perceptron requires a large value history because of the quantity of bits that must be stored. A 32 bit perceptron requires a massive value history. The local value history is typically too short for this approach to give accurate predictions.

A second approach is to mimic the table-based approach: a small cache of past values could be stored, an index can be associated with each value, and the value indices could be fed to the perceptron. The most recent value index would go to the first input, the second most recent value index would go to the second input, and so on. This is similar to the above 32 bit scenario, except now the perceptron needs fewer bits. Alternatively, it is also possible to associate a perceptron multibit input with each value, and feed to the perceptron input the order that that value appeared in the history. However, it is not clear that this gives any advantage (if all the values periodically repeat, the perceptron inputs will be the same).

Figure 5.3 shows my proposed perceptron local value predictor. Like the table-based predictor, a value cache is maintained with a LRU policy, and a local history stores the indices of the most recent value cache entries. A multibit perceptron input is sourced by each history entry, and the perceptron output is an

index of a value cache entry. To make a prediction, the perceptron is sourced with the indices in the local history; it makes a prediction of a cache index, and the value at that cache becomes the prediction. To train, the index of the actual value in the value cache is used. If the actual value is not in the value cache, it is inserted using the replacement policy, assigned an index, and the perceptron is trained using that index.

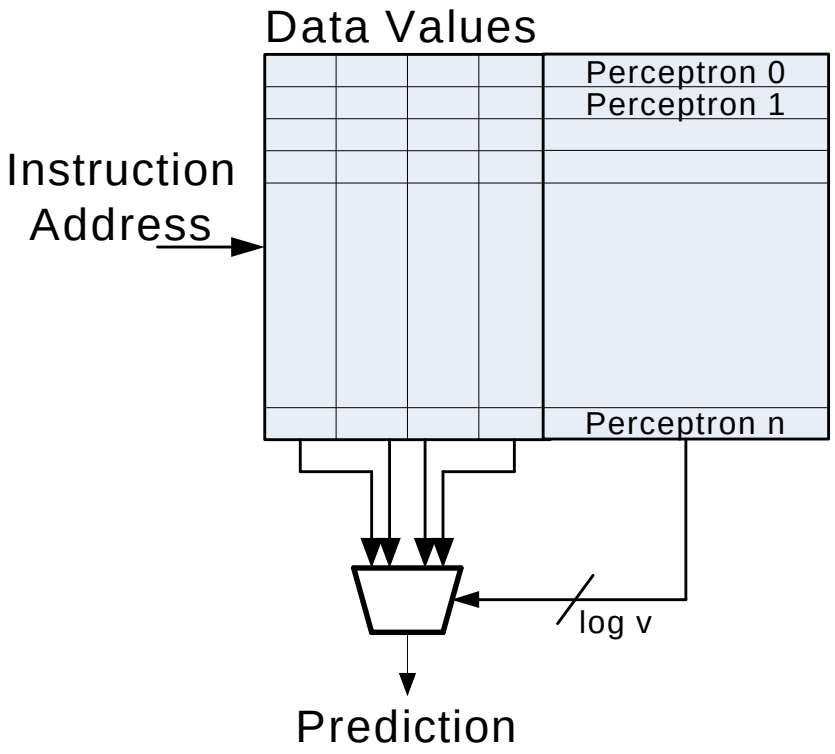


Figure 5.3. The perceptrons-in-the-value-table (PVT) predictor

This perceptron approach is clearly capable of learning repeating value patterns, even with the simplest multibit perceptron. If, for example, there is a three value repeating pattern, the weights associated with the third input will all learn a direct correlation, and become large. The other weights will observe no correlation, and become close to zero. The perceptron will simply predict the third value in the history.

Will the proposed perceptron approach outperform a table approach? It depends on the patterns in the local value history. The table approach favors short repeating patterns, which it can learn using less storage space than the perceptron. It can also capture value pairs, triplets, and quadruplets reliably. It has trouble with regularly repeating numbers that are not part of a pattern. The perceptron, however, favors longer patterns for two reasons. First, it handles growth better than a table. Second, the longer the pattern, the greater chance there is of finding past bits to correlate with for each target bit.

5.4. Global context-based prediction

As mentioned earlier, a satisfactory global context-based value predictor has yet to be proposed. In this section, I introduce three novel perceptron-based global value predictors and discuss why they should intuitively perform better than local value predictors.

5.4.1. Why global?

There are three major reasons why a global value predictor can perform better than a local predictor. First, a global predictor can take advantage of value correlations between different static instructions to make predictions. Second, a global predictor can predict a value that has not yet been seen in the local history, but has been seen in the global history. Third, as the local history is a subset of the global history, all the prediction information that is available to a local predictor is also available to a global predictor, provided that the global predictor considers a big enough past history (this third reason may not be very compelling, as the global

history may need to be huge if it is to hold sufficient local information.) I will now look at the first two reasons in greater depth.

5.4.1.1. Values Available Globally

There are many instances in a program where a value is always produced by an instruction that was produced by a recent past instruction. An example is illustrated in Figure 5.4. The load instruction produces the value previously saved to memory by the store instruction, which was in turn produced by the add instruction. Thus the add and the load always produce the same resulting value, although that value may differ from the values produced over previous iterations of the add and load. This example is particularly valuable to value prediction, as load instructions undergoing a cache miss produce considerable cycle savings when correctly predicted.

```
add $r6, $r1, $r1
sw $r6,400
...
lw $r2,400
```

Figure 5.4. Global value propagation

Figure 5.5 shows the percentage of values that have been produced before globally, in the last 50 dynamic instructions; locally, in the last 50 instances of the current static instruction; both; or neither (also considered are "cold" cases where an instruction is in its first instance and has no local history). This translates directly into the potential accuracy of a value predictor. The most common case of values, 47% on average, are available in both the local and global history. A minority of

values, 22% on average, are unavailable in either history. Of the remaining values, a substantial quantity, 8%, are available only in the global history. A predictor that neglects these values cannot reach its accuracy potential.

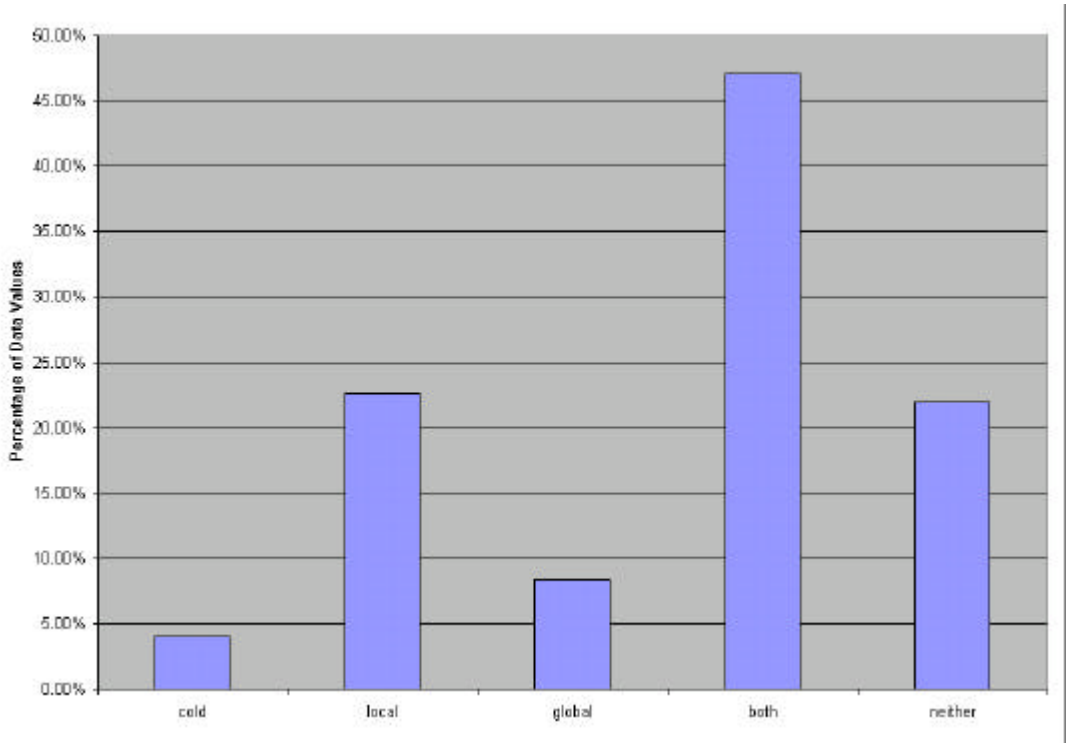


Figure 5.5. Previous places the current value has been seen

5.4.1.2. Value Correlations Available Globally

Even if only local values are predicted, the choice of which local value to use can be made using global information. For example, the local value sequence shown for instruction 2 in Figure 5.6 is clearly unpredictable. However, suppose a static instruction, iterating just before the target instruction, produced the value sequence shown under instruction 1. Instruction 2's predictor could use the choice of values produced by instruction 1 to indicate which local value to predict.

Iteration	$Inst_1$ Result	$Inst_2$ Result
0	1	5
1	7	4
2	7	4
3	1	5
4	7	4
5	1	5
6	1	5

Figure 5.6. Global correlations for local values

Do such correlations actually exist in real programs? Past research strongly supports this. One example of this is Thomas's work [Tho01_2] where he showed that unpredictable value sequences often occur when two easily predictable sequences are chaotically merged by unpredictable control flow changes. An example of this intuition is shown in Figure 5.7. Two value sequences are unpredictably merged at instruction 4, producing a locally unpredictable sequence at instruction. Instruction 4's value predictor would not be able to predict from local patterns. However, only one of the two instructions 1 or 3 will be in 4's global history. If instruction 4 uses its global history, it can determine which local value to choose.

Iteration	Instruction
0	beq 3
1	lw $r1, x
2	b 4
3	lw $r1, y
4	add $r1, $r1, 6

Figure 5.7. Global correlations between instructions

In a second study, I try to approximate the frequency of global value correlations. In this study, all static instructions that produce value-predictable results are examined. Those static instructions that produce more than two different local values, with two of those values appearing at least five times, are considered target instructions for this study. Static instructions that produce only one value are omitted for being trivially predictable, and static instructions that do not produce any value at least five times are omitted as being unpredictable with any local context-based predictor. Furthermore, I only consider those instances of target instructions as target instances if they produced one of the values that was seen at least five times. Other instances are omitted, since a context-based predictor could not be trained to capture those instances. Table 5.1 shows the percentage of static instructions that are considered target instructions, and the percentage of dynamic instructions that are considered target instances. Notice that few instructions meet these criteria; the majority of instructions produce either only one value most of the time, or produce many different values only once or a few times.

	%static	%dynamic
bzip2	7.64%	44.75%
gcc	2.35%	28.57%
gzip	18.86%	75.91%
mcf	14.08%	47.28%
perlbmk	0.04%	4.75%
twolf	9.05%	46.32%
vortex	1.57%	17.82%
vpr	0.83%	40.65%

Table 5.1: Percentage of instructions that repeatedly produce the same 2 values

A true (100%) correlation exists for a target instruction and a past instruction if there is a one-to-one mapping between each value produced by the target instruction during a target instance and the value produced by that past instruction during the same instance. That is to say, if the target instruction produces three different values X, Y, and Z, the past instruction will produce a value A each time the target produces X, a different value B each time the target produces Y, and another value C each time the target produces Z.

Since this is a very high standard, I also look at 90% correlations. A 90% correlation exists if the correlation holds on 90% of the target instances. These correlations are important to look at because 90% is an exceptionally high prediction accuracy for a value predictor.

Table 5.2 shows the results of this study. It shows the percentage of all target instances that are part of a target instruction which has a correlation within the last 50

instructions. It also shows how many instructions have a single value (occurring more than five times) correlated 100% with a past value. Perfect value correlations between past global instructions appear to be fairly rare. However, partial global correlations between one or some of the values tends to be more common.

	100%	90%	100% on one value
bzip2	7.79%	20.78%	48.70%
gcc	5.38%	12.14%	46.72%
gzip	3.73%	12.67%	29.06%
mcf	2.95%	3.30%	24.09%
perlbmk	6.12%	8.90%	56.64%
twolf	5.60%	5.68%	36.51%
vortex	21.31%	29.98%	57.27%
vpr	2.61%	6.76%	22.32%

Table 5.2. Percentage of instructions globally correlated with a past instruction

5.4.2. Perceptron Global-based Local

The first perceptron global predictor I propose is a "Global-Local predictor" that uses global value correlations to choose a local past value. It keeps a record of the past values produced globally by value-predictable instructions. It then uses perceptrons to detect correlations between these past values and the values produced locally. Although this predictor is limited to predicting only values seen before locally, it can use global correlations to decipher patterns that could not be predicted solely from the local value history.

The organization of the predictor is very similar to the PVT perceptron local predictor mentioned earlier, and is shown in Figure 5.8. The predictor has, per instruction address, a small value cache and a multibit perceptron. The output of the perceptron is used to choose a value from the local value cache. Unlike the local predictor, however, the inputs of the perceptron are fed from a global value index history, with a multibit input for each entry in that history. The global value index history is simply a shift register. When a value is actually produced by a value-predictable instruction, that value's index, local to that instruction, is shifted into the global value index history. Thus the global value index history does not actually contain values, but indices to values; the index for each value being determined by its producing instruction's local value cache.

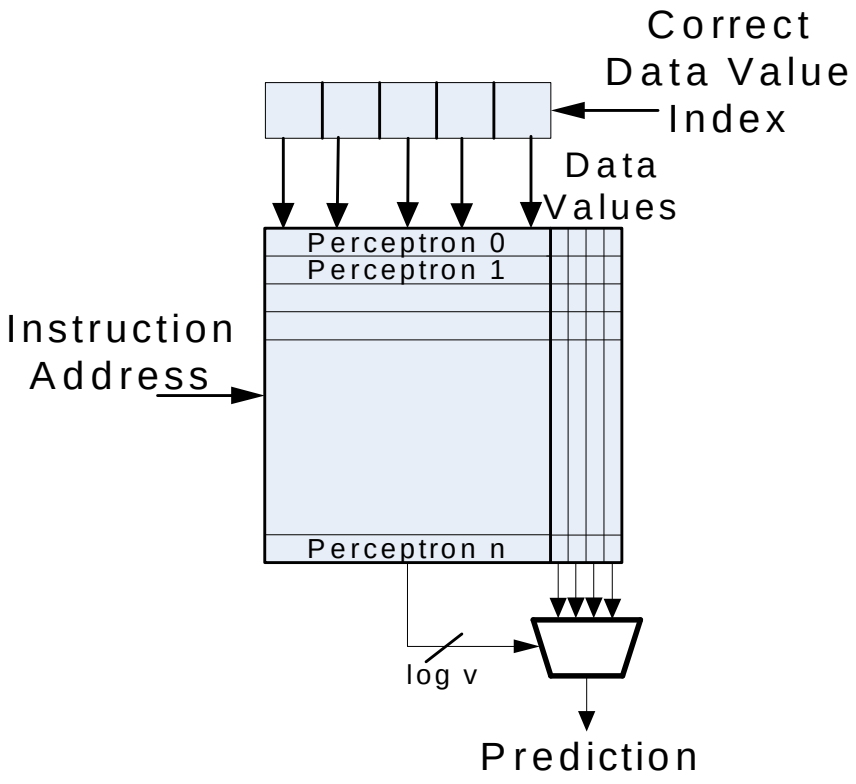


Figure 5.8. Global-Local Predictor

There are two important reasons why correlations are made with the local value indices rather than the values themselves. The first reason is because the predictor is smaller. If the entire value were used to correlate, each multibit input will need a bit for each bit in the value; in this case, the input will only need a bit for each bit in the index. The second reason is because it makes no difference to the perceptron. A perceptron can just as easily learn correlations between value indices as it can between values; in fact, with fewer bits to correlate, it can learn even better. It is not even a problem if two past static instructions assign the same index to two different values, as long as the instructions always send their values to the same perceptron inputs each iteration (I will discuss whether this is a good assumption later).

There are two major design parameters to this Global-Local predictor, apart from the multibit perceptron implementation details. The first is the global value history size, and the second is the local value cache size. The effect of each parameter on the physical size is discussed in 5.6.4.

The global value history size determines the size of the global value index history and the size of the perceptrons. Both sizes grow linearly with the global value history size; a one-entry increase means that the global value index history must hold another value index, and each perceptron will need another set of weights to handle another multibit input. As discussed in Chapter 3, an increase in the global value history size will have a couple positive effects on the perceptron accuracies. First, there will be more past values to correlate with, which increases the chance of finding good correlations in general. Second, given more correlating past values, a multibit

perceptron is more likely to find a correlation for each bit, allowing the perceptron to fully learn the correlation. On the other hand, more past values means more uncorrelated past values as well, resulting in more noise.

The local value cache size determines the global value index history size and the number of perceptron weights. In this case, both sizes grow logarithmically with the local value cache size (depending, of course, on the multibit perceptron implementation). Additionally, the local value cache itself must increase, creating an overall linear value cache growth with increased size. The local value cache must be big enough to hold most or all of the different values repeatedly produced by each value-predictable static instruction. If the cache is too small, the perceptron may not be able to predict the correct value due to it not having a local entry. There is no purpose in making the cache size too big, however. Once the value cache holds all of the repeating local values, there is nothing gained by it holding anything more.

5.4.3. Perceptron Global-based Global

As a second global approach, I make the local value cache global. I refer to this approach as the "Global-Global predictor." Making the value cache global potentially reduces the size of the overall predictor, and makes it possible for the predictor to predict values that have not been seen before locally.

The Global-Global predictor is depicted in Figure 5.9. The perceptrons, global value index history, and value table all function identically to the Global-Local predictor, with one major exception. Instead of having a value cache for each value table entry, the predictor maintains a single value cache. The indices shifted in the

global value index history, and the indices produced by the perceptrons, are the index of values in this global value cache.

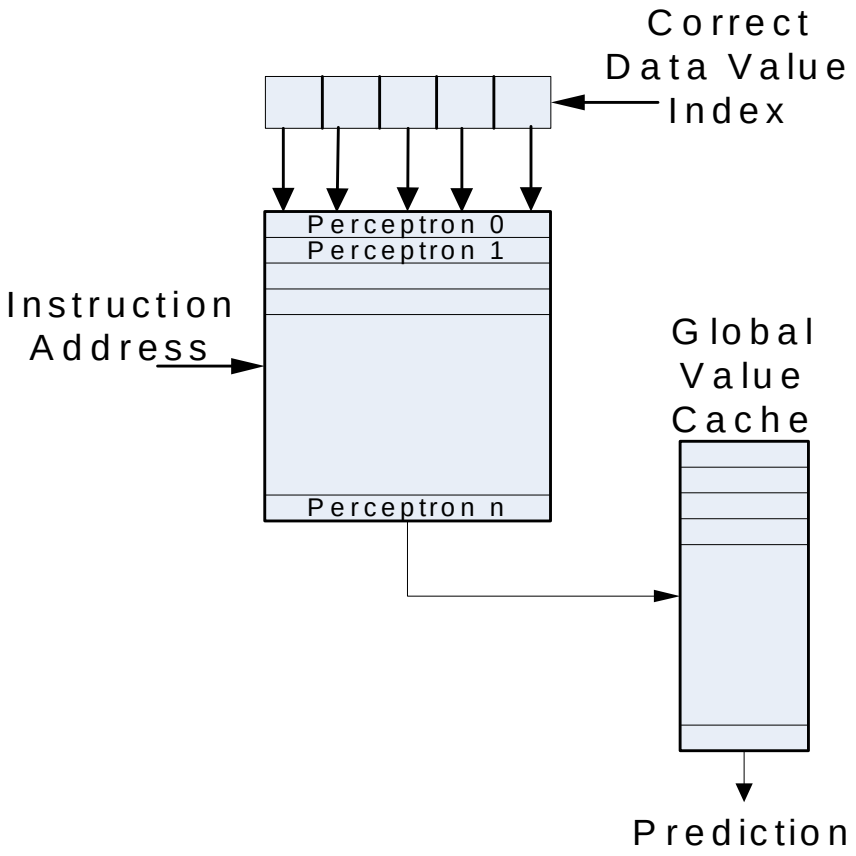


Figure 5.9. Global-Global Predictor

There are two advantages to making the value cache global. First, individual value table entries no longer need to hold a local value cache. This reduces the overall size of the predictor, and improves the predictor's flexibility. Second, the perceptron is no longer absolutely restricted to predicting local values. Thus, it is technically possible (though not necessarily likely), that the perceptron could correctly predict a value that has not been seen before locally, but has been observed globally. There are also a few disadvantages to making the value cache global. First, since the global value cache must hold more entries than any of the local value

caches, the number of bits in the value indices increase. Second, an untrained (or untrainable) perceptron is less likely to make lucky guesses with a global value cache than with a local value cache. Third, because of the larger cache size, value replacement becomes a bigger issue. I will explore each of these advantages and disadvantages in greater depth.

5.4.3.1. Global-Global Advantages

The first advantage of a global value cache is that it necessarily contains fewer entries than the total space consumed by the local value caches. The intuition behind this is simple. As was shown earlier, many different static instructions produce the same values, either by accident, or because a value is passed around through several instructions. In a prediction system consisting of local value caches, many values are stored redundantly in several value table entries. With a global value cache, however, no data value is stored more than once.

A related advantage to this is flexibility. Some instructions produce a large quantity of different values, exhausting their local value caches. Other instructions produce a small quantity of different values, or just a single value, wasting their local value cache space. If the local value cache size is made too small, those instructions that produce many different values become unpredictable; if it is made too big, value cache space is wasted. With a global value cache, no value storage space is wasted on the single-value instructions, yet those instructions that produce many different values are still predictable.

The second advantage is that it is now possible that a perceptron could predict a value that has been seen globally but not locally. However, this is unfortunately

unlikely to happen. The reason for this is that the perceptrons are still organized per-address. Because all the training data is local, a perceptron can only predict a value unseen locally by accident.

5.4.3.2. Global-Global Disadvantages

The biggest disadvantage in making the value cache global comes from the increase in the number of bits needed to represent the value indices. This increase is necessary, as the global value cache must be bigger than any individual local value cache. However, it results in an increase in size of all the perceptrons in the predictor linear with the growth of the index size.

The total size of the Global-Global predictor, given the assumptions made earlier with the Global-Local predictor, is $8htlogv+32v+hlogv$, where h is the history size, v is the value cache size, and t is the perceptron table size. This is explained further in 5.6.4. The Global-Local size is $32v_lt+8htlogv_l+hlogv_l$, where v_l is the local value cache size. For a h of 32, a t of 4096, and a v_l of 32 (the default values used in simulation) the Global-Global predictor is approximately the same size as the Global-Local predictor when the value cache size v holds 512 entries.

A second disadvantage with a global value cache is that an untrained perceptron is less likely to accidentally guess the correct value. Since there is such a high probability that a local value will be produced again, simply guessing any past local value has a high chance of being correct. In a local predictor, a perceptron associated with an instruction that is always producing the same value will make correct predictions by default, since there are no alternative values to predict. A perceptron choosing randomly from a larger global pool, however, is unlikely to

accidentally choose a local value at all, much less the correct value. This disadvantage, however, is not likely to be a concern if a confidence estimator is used with the value predictor. In general, for these perceptron approaches, it is unwise to blindly accept the perceptron's output until the perceptron is fully trained. Random guesses, lucky or unlucky, are best ignored if accuracy is a concern.

A third issue is the replacement policy for the Global-Global predictor. Recall that the Global-Local predictor and the local predictors used an LRU replacement policy. There are two problems with using LRU with a global value cache. The first problem is complexity. In the previous cases, the LRU policy could be easily implemented by associating a small counter with each value, and when accessing a value, incrementing the counters for all the other values whose count is smaller than the accessed value's counter. This is easily implemented with 4 values, but less so with 1024 values, simply because of the latencies involved.

The second problem with LRU is that it is not necessarily the best replacement policy for a global value cache. Consider, for example, a frequently-running static instruction that produces stride sequences. Because that instruction produces a large quantity of different values and runs frequently, it fills up a lot of space in the global value cache. However, since stride sequences cannot be captured with a context-based predictor, that instruction is neither predictable nor can be used to correlate with another instructions. Its values are effectively wasted space in the value cache. A more effective policy might be a least-frequently-used (LFU) policy that retains values that are produced repeatedly or in many instructions. However,

some sort of aging mechanism would be needed to clean out old values that are no longer being produced.

5.4.4. Perceptron Bitwise

The third perceptron approach that I propose is to eliminate the value cache altogether and let the perceptrons directly predict the data values. Although this runs the risk of making the predictor size huge, it offers several advantages that make it potentially the most accurate of the perceptron value predictors.

The bitwise approach is shown in Figure 5.10. A global value history holds the actual past dynamic values. A single multibit perceptron is associated with every instruction, and has a multibit input for each of the past values, up to a certain value history size. The multibit perceptron output is used as the predicted value.

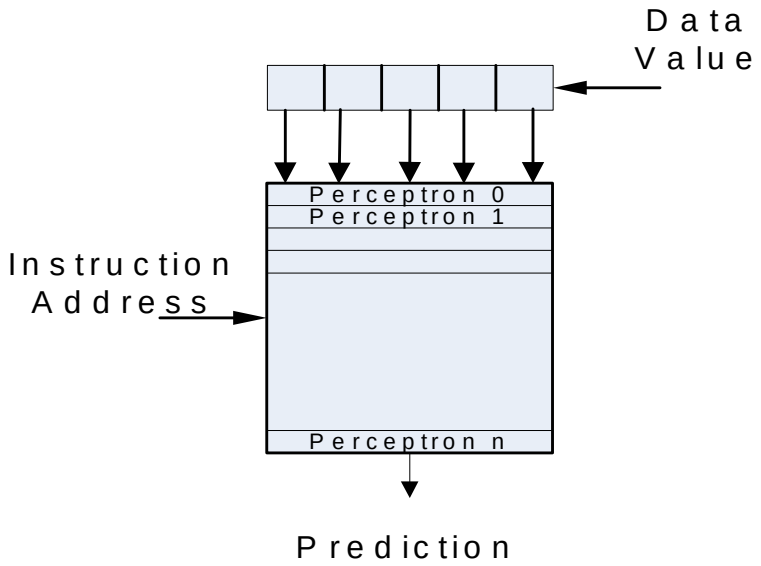


Figure 5.10. Bitwise Predictor

I will first address the glaring issue with this approach. The perceptrons are large. Each perceptron input is no longer a 2 or 3 bit local index, or even a 9 or 10 bit global index, but a 32 bit value. However, the predictor does not need to store any

past values. The predictor size, with the assumptions stated before, is a little over 4MB, as will be derived in 5.6.4., which is slightly bigger than the other predictors.

That stated, there are a couple advantages to this bitwise approach. First, all the disadvantages associated with having a value cache, such as storage and a replacement policy, are no longer a problem. Second, without confining the perceptron to predicting only values seen before, the perceptron could conceivably predict new values, much like an arithmetic predictor. Later, when I simulate the behavior of a perceptron bitwise predictor, it will be shown that the predictor actually does occasionally predict previously unproduced values.

5.5. Value Prediction Implementation Details

Before looking at simulation results, it is necessary to describe how value prediction is actually simulated. In fact, actually simulating value prediction brings up some design issues unrelated to the actual value prediction method used. One of these, how to handle misprediction recovery, was mentioned earlier. In this section I will describe how simulation is performed, and describe two issues that arise when implementing value prediction in simulation.

5.5.1. Simulating Value Predictors

Earlier in Chapter 4, I described my processor simulator and how it works. The value prediction strategies mentioned above are all simulated as an add-on to the simulator and implemented in the file myvpred.c.

Value prediction is simulated cycle accurately. That is to say, value predictions are made at the point that an instruction is dispatched, subsequent data

dependent instructions are executed with the speculative values, and mispredictions are detected and squashed after the original instruction finishes execution using a ReFetch strategy. The interface between the out-of-order simulator and the value predictor is implemented through three function calls to the value predictor: get_value_prediction, which takes the instruction address and returns a predicted value; get_value_prediction_confidence, which takes the instruction address and returns a decision to "use" or "don't use" the value; and train_value_predictor, which takes the instruction address, the predicted value, the actual value, and the confidence decision.

The prediction simulation is performed as follows. When an instruction is dispatched to a reservation station, and data dependencies are looked up for register renaming, if the instruction consumes a value produced by an instruction that is still executing, for which a value prediction has been made, that predicted value is forwarded to the newly dispatched instruction. Also at dispatch, if the instruction is an integer arithmetic instruction or a load instruction that produces a single output value (this includes lw, but excludes dlw), the value predictor is called to make a prediction for the output value of the instruction. This predicted value is stored in a field in the instruction's reservation station. At the same time, a value prediction confidence estimator is called, and its result is also stored in the instruction's reservation station. If the confidence estimator flags the instruction as "don't use", a prediction is still made and stored for value prediction training purposes, but the speculative value is not forwarded on to dependent instructions.

The actual result is known at the execution stage, at the point that the instruction's reservation station's busy flag is set to "ready for writeback." At this point, if a value prediction is made, the value prediction is incorrect, and the confidence estimator predicted "use", misprediction recovery is performed. All subsequent instructions, whether dependent or not, are removed entirely from the reservation stations, reorder buffer, and dispatch queue. The PC is set to point to the instruction following the mispredicted instruction.

It may seem like overkill to squash instructions that are not data dependent on the mispredicted instruction. However, there are several reasons why this greatly simplifies the recovery process. First, if one of the data dependent instructions affects the control flow, all subsequent instructions, while not data dependent, are nevertheless executed incorrectly. Second, if a data dependent instruction is squashed while a non-data dependent subsequent instruction is not squashed, the data dependent instruction, upon refetch, will enter the reorder buffer after the subsequent instruction. Third, by squashing instructions indiscriminately, it is not necessary to keep track of the data dependency graphs and other cumbersome details, nor have to trace the dependency graphs on a misprediction. Squashing is consequently a simple, rapid procedure, requiring no substantial extra storage, and could be easily constructed in an actual hardware implementation.

To improve performance, a "prediction_used" flag is added to each reservation station entry, and is set only if a data dependent instruction actually uses the predicted value. Misprediction recovery is not performed if no subsequent

instruction consumed the speculative value, even if the value was predicted incorrectly. This prevents a significant number of needless squashes.

Value predictor training is also performed in the execution stage at the same time (in the simulated processor, immediately afterwards). If a prediction was made, whether or not it was actually used, the train_value_predictor function is called with the correct and predicted values.

When the get_value_prediction and train_value_predictor functions are called, the function uses a command line flag to choose which value prediction strategy to use, and calls the corresponding function for the appropriate prediction strategy. The get_value_prediction_confidence function works similarly, except that at this point it always returns “use prediction.”

At get_value_prediction, the value prediction is made for a given instruction. The lower bits of the instruction address, shifted right by 3 (since instructions are 8 bytes long in the PISA architecture), is used to index the value table. A predicted value is determined from the table entry and the value index history, and is returned.

Most of the meat of the value predictor is implemented at train_value_prediction. Using the appropriate bits of the instruction address, the relevant value table entry is chosen. Next the upper bits of the instruction address is compared to a tag field stored in the value table entry to detect value table aliasing. If the tag does not match the address bits, the value table entry is cleared, as described in more detail below. The value table entry is then trained, using the value index history, and predicted and actual values. After training, the value index history is

updated as described below. Finally, simulation study counters that quantify prediction accuracy and other metrics are updated appropriately.

5.5.2. Global Value History Register

Recall that nearly all the value prediction strategies mentioned above use a value index history (or in some cases, a value history). In most of the above strategies, this value index history is global, meaning that there is a single shift register used on every prediction. This value index history register is used both when the prediction is made and when the predictor is trained. When the prediction is made, the value index history is used as inputs to the perceptron. At training, the value index history register is used to provide the input values needed to train the perceptron. At the end of training, the register itself is updated. All the entries are shifted over by one place, and the index of the current training value (or the value itself) is shifted into the first entry.

A major problem with implementing this strategy simply as stated above is that the value index history may be different at training from what it was at get_prediction. If the history has changed, the inputs to the perceptron at training are different from the perceptron inputs when the prediction was made, and the perceptron is consequently trained incorrectly. This change in history occurs because training is performed on a different instruction between an instruction's dispatch and the completion of its execution. As value prediction is performed on every instruction with a single integer output value, such history changes happen very frequently; too frequently, in fact, for global value prediction to produce accurate results unless this problem is addressed.

If the number of intervening instructions being trained between get_prediction and train_predictor were fixed, the intuitive approach would be just to train using the value index history shifted by a fixed amount. Unfortunately, in an out-of-order processor, the number of intervening instructions can vary greatly, even between different iterations of a static instruction.

The most straight-forward approach, consequently, is to back up the value index history at get_prediction and use the copy at training. There are two places that the history can be backed up. One is at the value table entry. The other is the instruction's reorder buffer entry. For simplicity, and to preserve modularity, in these simulations the value index history copy is stored at the value table entry. However, in an actual implementation, placing the copy at the reorder buffer entry would be more desirable for two reasons. First, for most processor implementations, the reorder buffer is likely to be much smaller than the value table. Second, if two iterations of a static instruction occur in rapid succession (it is unlikely, but possible), or if two instructions both aliased to the same value table entry occur in rapid succession (also unlikely for large table sizes), the value index history copy could be overwritten with another copy before it is used in training.

An alternative approach would be to associate a counter with each value table entry or reorder buffer entry, and count the number of intervening trainings. The value index history would then be shifted over by the appropriate count at training. While the value index history would need to be somewhat longer than what is used by the perceptrons to fill in the gaps from shifting, storage space would not need to be consumed from storing backup copies of the history. This challenge, of course, to

this approach, is that it would require some logic to increment the relevant counters on each training, and a fast shifting mechanism for the value index history.

5.6. Experimental Results

The above perceptron value predictors were evaluated on the Mysim simulator with the benchmarks and simulator setups described in 4.2, except that the number of instructions for the local predictors is only 100 million / benchmark. The baseline predictor is the table-based context-based predictor described in 5.3.1. Two baselines are evaluated: at a history size of 4, which is smaller than the perceptron predictors, and at a history size of 8, which is larger than the perceptron predictors. Since the smaller baseline actually performs with slightly better accuracy, it will be used as the primary baseline in the comparisons. Both the raw accuracy of the value predictors and the IPC are evaluated.

Performance evaluation is performed completely cycle-accurately, with a ReFetch squashing policy employed on mispredictions. Because of the drastic performance degradation from ReFetch squashing, the value predictor performance for nearly all the results below is actually worse than if no value predictor is employed. This can be observed by comparing the IPC results here to the ones reported under the same simulator parameters in Chapter 4. Consequently, all performance results should be considered relative to the baseline, rather than in absolute terms. The performance of each value predictor can be substantially improved with a very conservative confidence estimator; however, the results below are shown without any confidence estimation in order to show the full IPC effect of each prediction scheme.

5.6.1. Local Value Predictors

There are two local perceptron value predictor approaches evaluated. The PPT approach places the perceptrons in the pattern table, replacing the counters. The PVT approach places the perceptrons in the value table, replacing the entire pattern table. The perceptrons use the disjoint multibit topology discussed in 3.3.2.1. By default, the perceptrons are implemented with training-by-error and linear weight growth.

The baseline predictor cannot consider more than four different past values or a local history of more than four without suffering from excessive size. The PPT approach cannot consider more than four different past values, but the local history size can be varied as a parameter. In the PVT approach, both the number of past values and the local history size can be varied as parameters.

Figures 5.11 and 5.12 compare the prediction accuracy and IPC, respectively, for the PPT over each benchmark, for varying local history sizes. In every case, the baseline predictor outperforms the perceptron predictor. This is not surprising. As discussed earlier, these local perceptron predictors can learn little that the baseline predictor cannot learn, and the selection of pattern table entry means that the predictor suffers from the same pattern table aliasing problems as the baseline. Furthermore, the PPT predictor carries the perceptron training time overhead and learning restrictions.

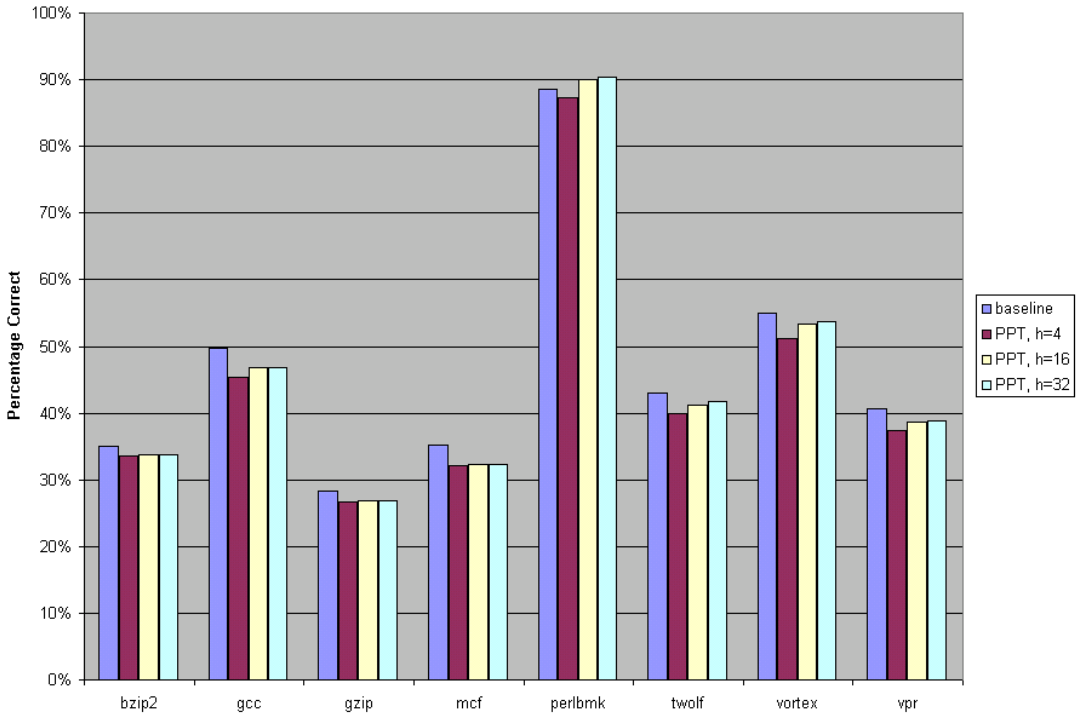


Figure 5.11 PPT predictor accuracies

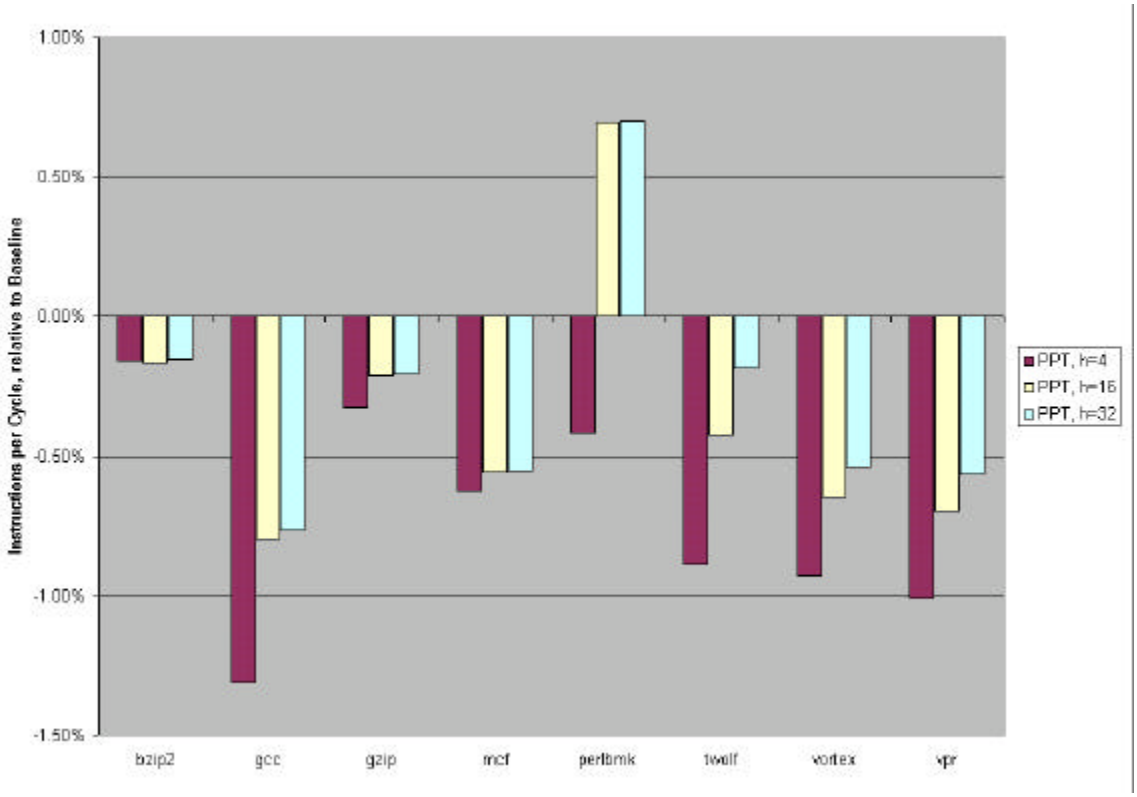


Figure 5.12. PPT predictor performance

Figures 5.13 and 5.14 compare the prediction accuracy and IPC for the PVT predictor at different parameters. Both the past value number and history sizes are varied, but the past value number never exceeds the history size as this could not result in any performance improvement. The perceptron predictor outperforms the baseline by a modest 2.47-4.76% accuracy. While this demonstrates that the perceptron approach is superior, at least as far as performance is concerned, it has only a very slight advantage. There are two reasons that the perceptron predictor considering 16 times more history performs only slightly better. The first is that the perceptron's learning restrictions gives it a natural disadvantage over the table-based approach. The second reason is that there is only a certain amount of prediction information available in the local history at all, and the table-based predictor largely captures it.

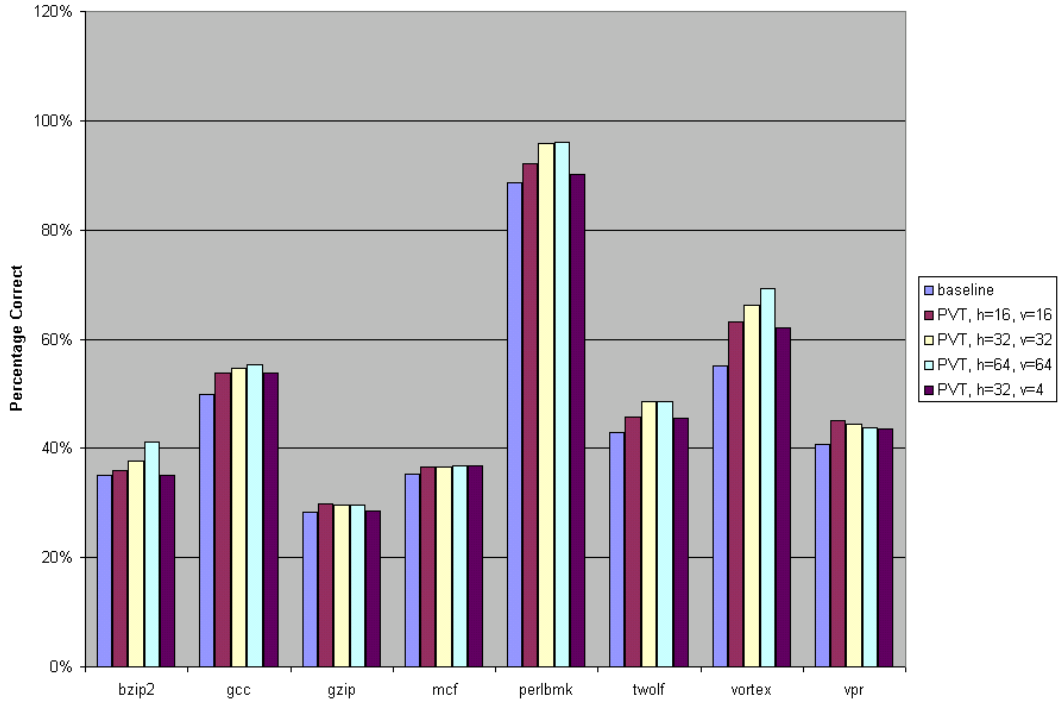


Figure 5.13. PVT predictor accuracies

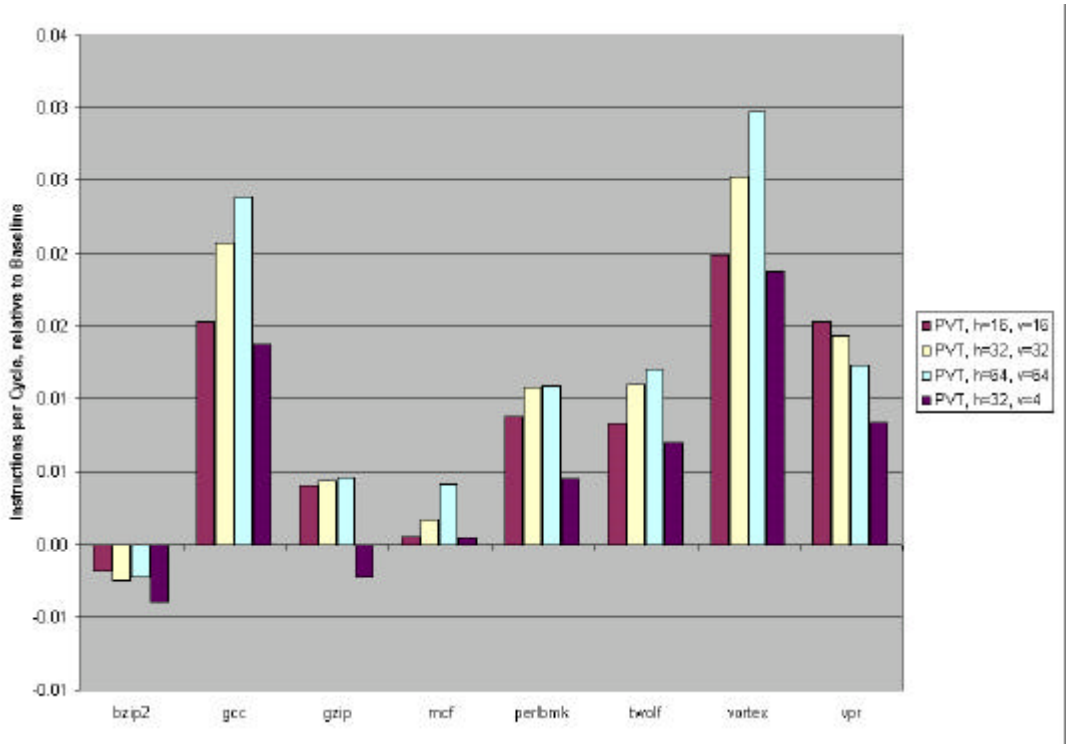


Figure 5.14. PVT predictor performance

5.6.2. Global-based Local Predictors

Three perceptron-based global-local value predictors are evaluated, comparing different multibit perceptron topologies. The first uses a disjoint topology, the second a fully coupled topology, and the third a weight-per-value topology as described in 3.3.2.1, 3.3.2.2, and 3.3.2.4. These global-local predictors are limited to predicting locally available data, but use global correlations and a global history to choose the local value. The default global history size of the predictors, and the number of past local values stored, is 32. By default, the perceptrons use training-by-error and linear weight-growth. Aliasing is countered in the global history using the assigned-seats method detailed in 3.4.7.1.

Figure 5.15 shows the accuracies of the three predictors across the benchmarks, and Figure 5.16 shows the IPC performance. On average, the disjoint perceptron approach shows an absolute accuracy increase of 3.12% and a relative performance increase of 1.59%. Due to its unrestrained ability to learn value correlations, the weight-per-value approach shows an even better 10.67% accuracy increase and 4.36% relative performance increase. Interestingly, however, the fully coupled perceptron approach, with its superior ability to learn value correlations, suffers a cross-benchmark accuracy decrease of 6.83% and a 1.48% IPC decrease.

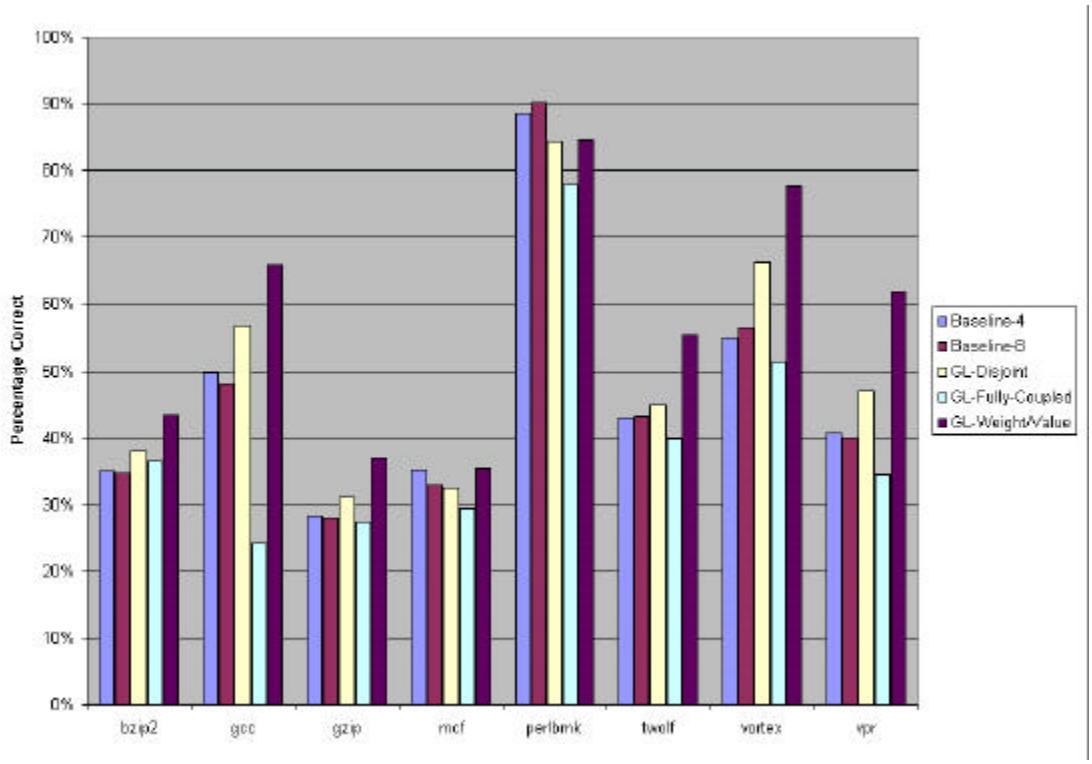


Figure 5.15. Global-Local Predictor Accuracy for Different Multibit Topologies

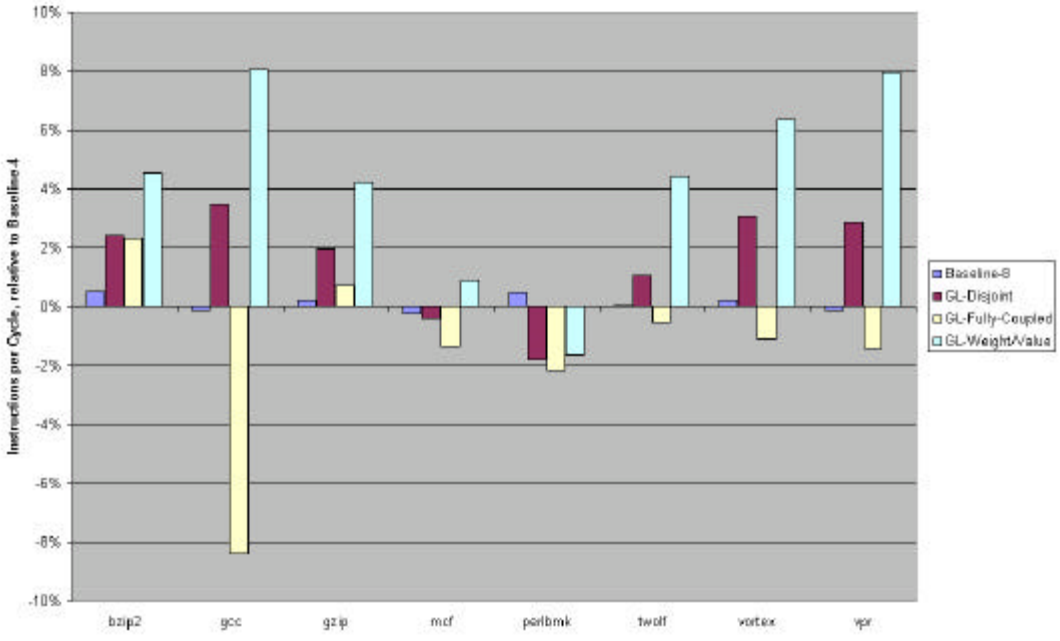


Figure 5.16. Global-Local Predictor Performance for Different Topologies

There are two interesting questions that arise from the above results. First, why do some benchmarks, such as perlbmk, buck the trend of the other benchmarks and show a performance decrease for the global-local predictors? Second, why does the fully coupled approach actually show a decrease, despite its higher learning potential?

The poor performance of perlbmk reflects the fact that local value patterns are very easily observed in that benchmark. Because value patterns can be easily predicted using local patterns, it is unnecessary to look for global correlations to obtain highly accurate results. This is confirmed by the high performance of the PVT predictor on this benchmark. A hybrid global-local / local-local predictor could be a consideration if other benchmarks followed the same trend as perlbmk.

The poor performance of the fully coupled perceptron is a consequence of it having a lower percentage of correlated weights. Recall that this approach has 32 times more weights than the disjoint perceptron on the PISA architecture. Of course, the studies in Chapter 3 showed that the quantity of weights does not matter as long as the proportion of correlated weights stays constant. However, this does not happen, as will be explained in more detail in Chapter 7. This decreased percentage of correlated weights makes the fully coupled perceptron more susceptible to both weight noise and false correlations, causing a substantial performance decrease in spite of its increased learning ability.

The weight-per-value value predictor has significantly more weights than the fully couple perceptron, as the fully coupled perceptron has a weight for each of 5 bits to handle 32 past values, while the weight-per-value perceptron has a weight for each

of the 32 past values. However, it does not suffer the same performance decrease. The reason for this is because the predictor is implemented so that if a past value is not present at a particular input, a 0 appears at that input instead of a 1 or -1, as was described in 3.3.2.4. The 0 value cancels out that weight, removing that weight as a potential source of noise or imbalance. Because only one value can appear at any particular multibit input, only one weight is active at any time for each multibit input. Thus the weight-per-value predictor effectively has the same number of weights as the disjoint predictor.

Figure 5.17 shows the sensitivity of the disjoint perceptron global-local predictor to changes in history size. It is no surprise that the predictor performs better with greater history sizes, as there is both more opportunity for correlations and more correlated inputs for the perceptron to learn the correlations. However, history size demonstrates diminishing returns. First, correlations are more common in recent history than in far off history, which means that the chances of finding a correlated weight get smaller as the history is increased. Second, the increase in history without an increase in the number of correlations means a decrease in the percentage of correlated inputs as the history size grows. This leads to an increase in noise and also correlation problems.

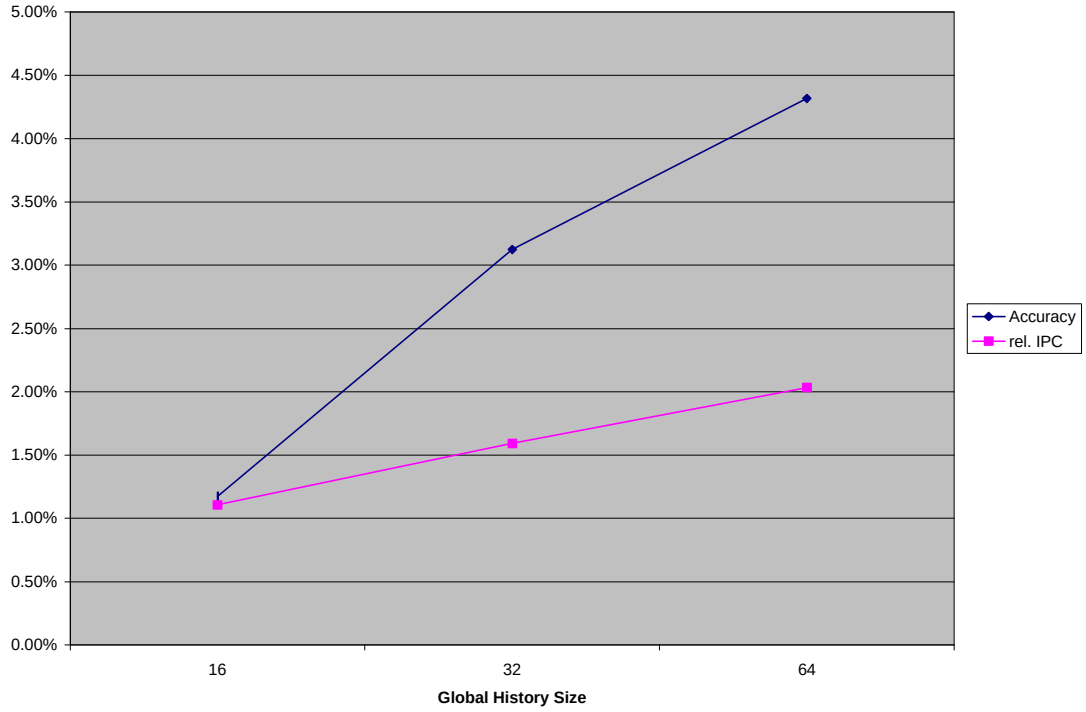


Figure 5.17. Effect of history size on Global-Local accuracy

5.6.3. Global Value Predictors

Two global value predictors are evaluated below. One is the global-based global predictor which uses a 1024 entry global value cache. This ideal cache size was determined empirically; larger global value cache sizes of 2048 and 4096 performed only negligibly better, while smaller caches of 256 and 512 performed substantially worse. Thus for an average program it can be assumed that there are typically 1024 data values on average that are repeatedly used at any time. A LRU replacement strategy is used to place values in the global value cache. The global-based global predictor is implemented as a disjoint perceptron predictor with 10 bits (for 1024 value entries). The perceptrons employ training-by-error with linear weight growth, and the assigned-seat interference reducing strategy is employed for the global history.

The second global predictor is the perceptron bitwise predictor. It also used linear weight growth, training-by-error, and assigned seat anti-interference.

Figures 5.18 and 5.19 show the accuracy and performance for the two global value predictors. The global-global predictor shows an average accuracy increase of 7.56% and an average relative performance increase of 6.69%. The bitwise predictor shows a 12.67% accuracy increase and 5.28% performance increase.

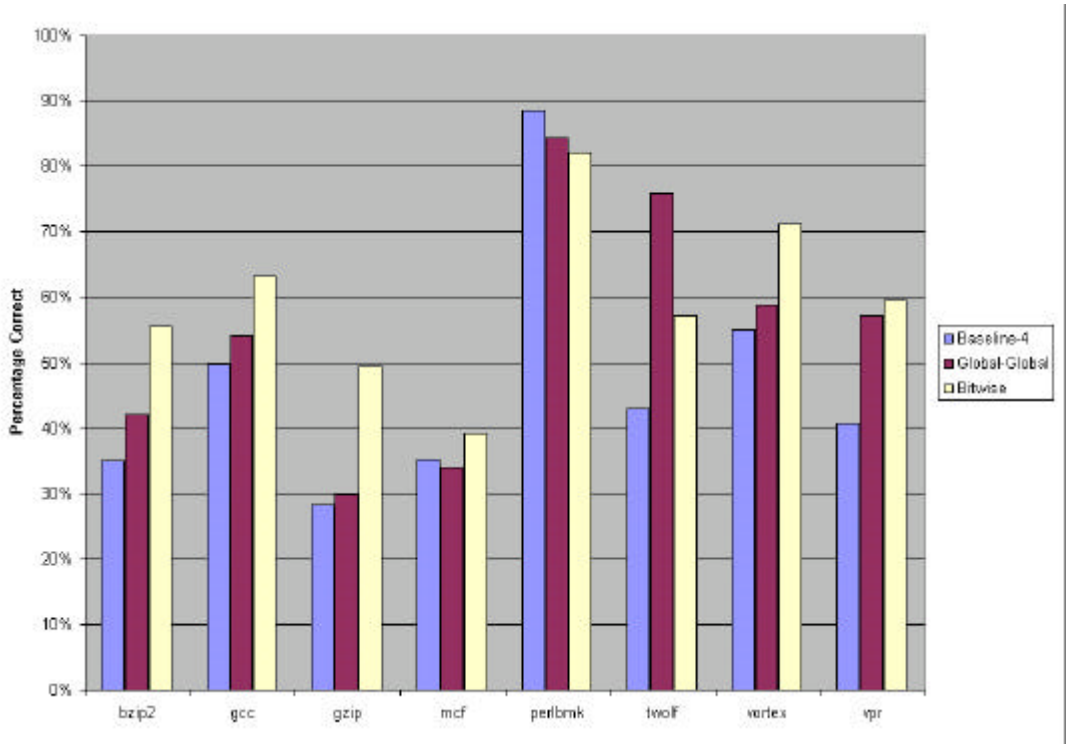


Figure 5.18. Global Predictor Accuracies

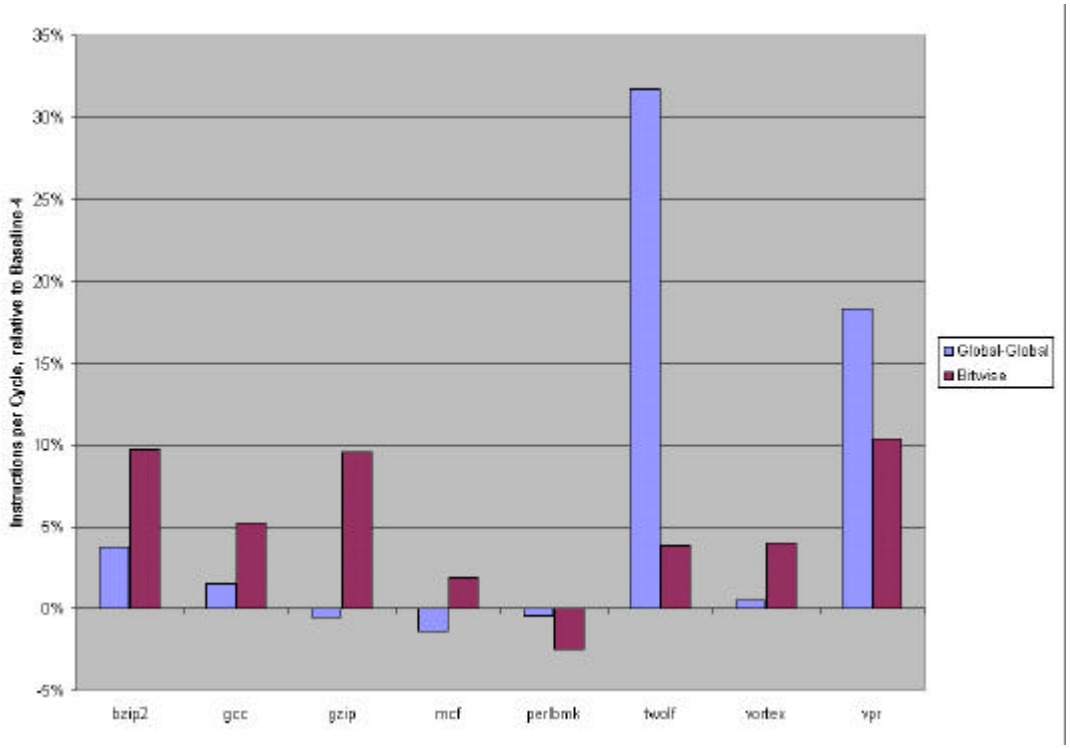


Figure 5.19. Global Predictor Performance

Unlike the local-local and global-local predictors, and even the global-global predictor, the bitwise predictor is capable of producing values that have not seen before locally. This is because it can learn correlations for each bit of the value independently of the actual value, and can consequently produce whole values bit-by-bit from several different bit-correlated inputs. Figure 5.20 shows the percentage of all data values that the bitwise predictor produces that are both correct and have not been produced before in the last 50 local history entries, and the percentage of values that are both correct and have not been produced before either in the last 50 local history entries or the last 50 global history entries. On average, 5.0% of the bitwise predictor's guesses are correct values that have not been seen before. A further 9.8% of the predictor's guesses are correct values that appear in the global history but not

the local history. These correctly predicted values are unobtainable with any of the other prediction approaches.

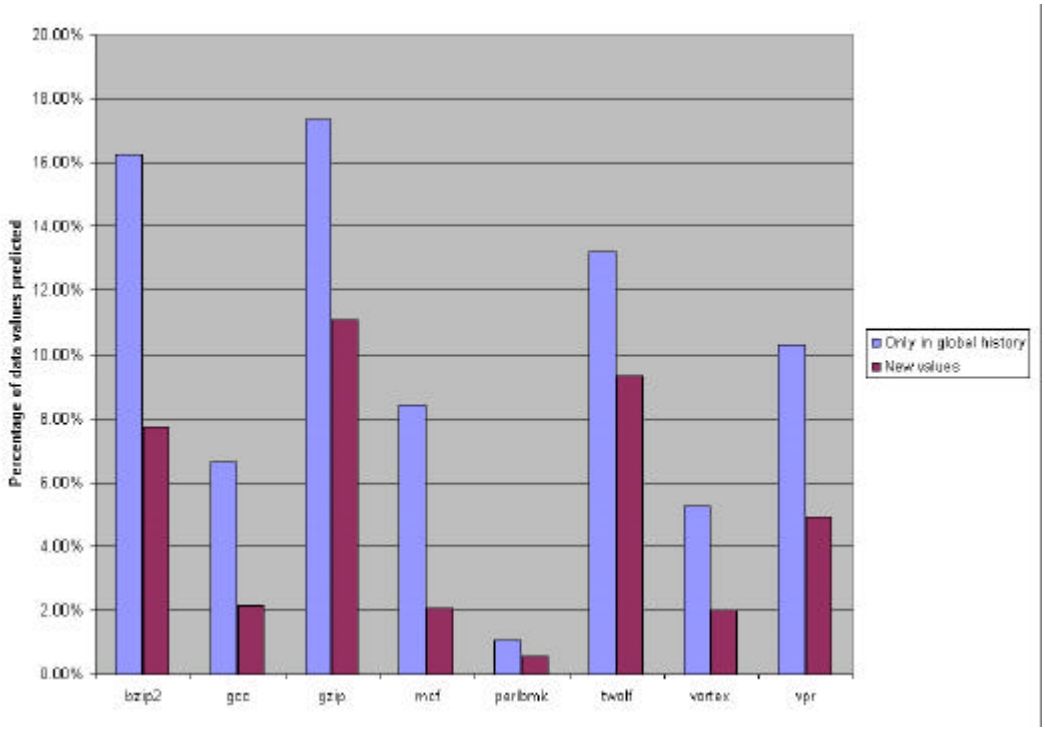


Figure 5.20. Correctly predicted data values that have not been produced before

5.6.4. Comparing Physical Size

An important factor in adopting one predictor over another is the physical size. As mentioned in Chapter 2, physical size is primarily determined by the storage size needed. The factors determining this size are the number of past values stored v, the number of perceptron table entries t, and the value history size h. It is assumed below that 32 bits are needed for each value, and 8 bits for each perceptron weight.

The baseline predictor has two components: the value table holding each value and the pattern table holding the counters. The value table contains t entries each consisting of v values and an index history of $v\log v$, creating a total size of $32tv+vt\log v$ bits. By design, the history size h must equal v. The pattern table has

$2^{v\log v}$ entries, each entry containing v counters, each counter being 2 bits. The total storage directly relating to making predictions (thus excluding tag fields and LRU replacement bits), comes to $32tv+vt\log v+2v2^{v\log v}$ bits. If v equals 4 and t equals 4096, this requires approximately 69.9kB of storage. Were a v of 8 used, the total would be 33.7MB of storage, more than any of the perceptron approaches, and a v of 16 would need a little over $73.7*10^{18}$ bytes. Consequently a baseline with a v of 16 or more is not considered.

The perceptron in the pattern table (PPT) predictor requires the same number of pattern table entries. Each value table entry would need an index history of $h\log v$ and a value storage of $32v$. Each pattern table entry needs $8h\log v$ storage, assuming a disjoint perceptron approach is used. The total storage is thus $32tv+ht\log v+8h\log v2^{v\log v}$. For a v of 4 and a h of 4, little over 71.6kB is needed. At a v of 4 and a h of 32, 114.7kB are needed.

The perceptron in the value table (PVT) predictor contains only the value table. Each entry requires $32v$ past value storage plus $8h\log v$ weight storage (for a disjoint approach) plus $v\log v$ local storage, making a total size of $32vt+8ht\log v+vt\log v$. At v=4 and h=4, this comes to 102kB. At v=32 and h=32, this requires 1.26MB.

The disjoint global-local approach has a value table with local past value and perceptrons, and a global history register. Each value table entry requires $32v$ past value storage plus $8ht\log v$ perceptron weight storage. The global history register contains h past value indices with $\log v$ bits/index, making a total of $h\log v$ storage.

The total thus is $32vt+8ht\log v+h\log v$. At v=32 and h=32, this requires 1.18MB of storage.

The fully coupled global-local approach requires $8th\log v\log v$ for the total weight storage. At v=32 and h=32, this requires 3.80MB. The weight-per-value global-local approach requires $8thv\log v$ for the total weight storage. 21.50MB are required.

The disjoint global-global approach has a value table entry containing only perceptrons, a global history register, and a global value cache. Assuming the value cache has v entries, $32v$ bits is needed to store it. The global history register requires $h\log v$ storage. The value table requires $8ht\log v$ bits, making a total of $8ht\log v+32v+h\log v$. Using a v=1024 and h=32, 1.31MB are required.

The bitwise approach requires a perceptron width of 32 bits. No past values are stored. The global history register requires $32h$ bits of storage. The value table requires $32*8ht$ storage. The total thus is $32h+256ht$. For h=32, this requires 4.19MB of storage.

5.6.5. Comparing Training Procedures

The above predictors were all trained using training-by-error. Figure 5.21 shows the results for the disjoint global-local predictor when using training-by-correlations and training-by-error with exponential weight growth. In Chapter 3 it was shown that training-by-correlations and exponential weight growth both improved the prediction learning rate. However, as can be seen, exponential weight growth causes a 4.51% accuracy decrease on average, and training-by-correlation causes a 4.82% accuracy decrease.

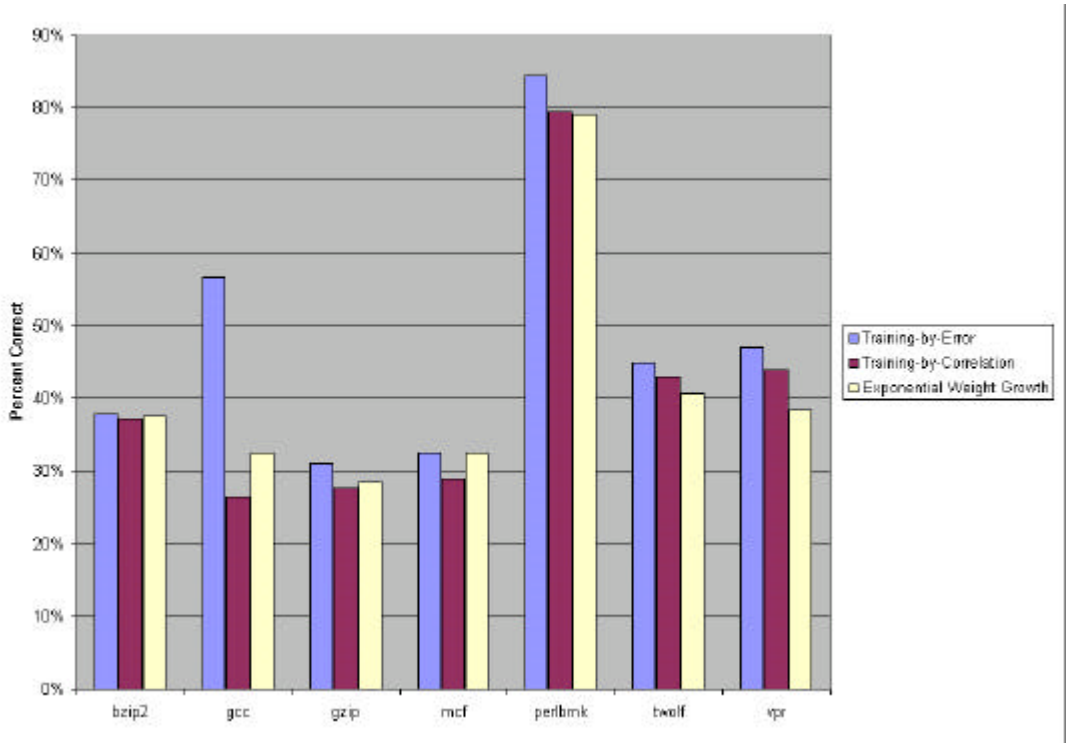


Figure 5.21. Global-Local accuracies for different training procedures

Clearly, training-by-correlations is an inferior training policy to training-by-error for performing value prediction. This is surprising, considering the excellent performance of that policy in perceptron branch prediction.

Figure 5.22 shows how the PVT predictor, with a history size of 4 and 4 past values responds to the two training approaches. For this predictor, training-by-correlation performs with a 0.3% higher accuracy than training-by-error. Recall that the recent local value history tends to be very well correlated. As will be shown in Chapter 7, the global value history is poorly correlated. Earlier in Chapter 3, I showed that training-by-correlation performs poorly if the input data is both poorly correlated and imbalance. This limitation of training-by-correlation explain why the approach performs well with PVT and poorly with Global-Local.

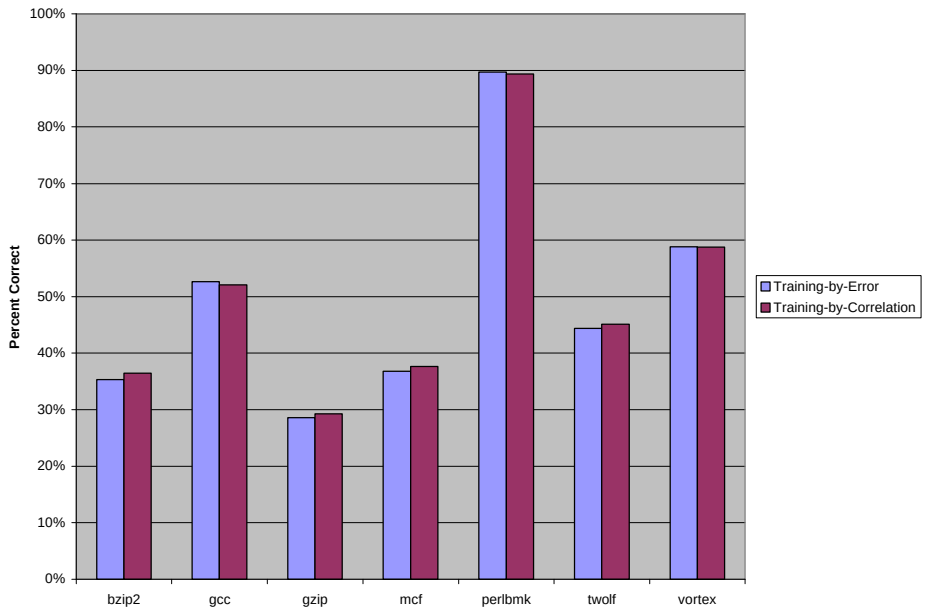


Figure 5.22. PVT accuracies for different training procedures

5.6.6. Interference

Recall that global predictors can suffer from interference in the global history table. The above approaches used the "Assigned Seats" interference reduction policy. Figure 5.23 shows the effect of the different interference reduction approaches on the disjoint global-local perceptron. No interference reduction, assigned seats, and piecewise linear are all considered. For the piecewise approach, 32 different instructions are handled at each input (this is clearly more than are needed, but 32 is chosen to show the potential). On average, assigned seats performs 0.92% better than no interference reduction, and piecewise performs 3.28% better than no interference reduction.

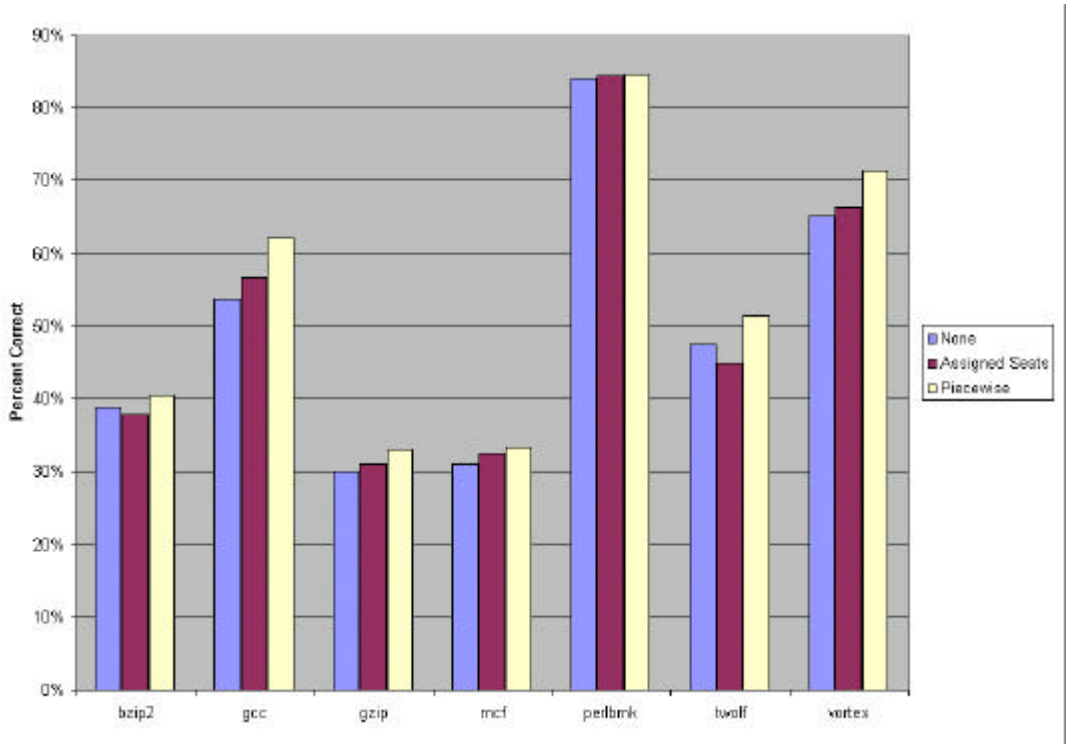


Figure 5.23. Prediction accuracies for different interference reduction methods

Since piecewise clearly performs the best, is it the answer to interference reduction? Not if storage cost is a consideration. The additional storage cost associated with assigned seats is marginal; the global history table also needs to store the instruction addresses. For a 32 entry table, this requires 5 bits of storage / entry, or an additional 20 bytes. Piecewise linear requires not only the instruction addresses be stored, but also additional weights for each of the possible different instructions that could appear at each input. In the above example, the entire physical size is effectively multiplied by 32, with the piecewise predictor consuming over 37MB of space.

Chapter 6: Critical Instruction Prediction

In recent years there has been growing interest in predicting whether individual instructions lie on the dataflow critical path. In superscalar processors with sufficiently many functional units, the data dependencies between instructions effectively determine the order in which instructions are executed. These data dependencies create a dataflow graph through the code, with the latencies of instructions forming the graph edges. The critical path is the longest route through this dataflow graph. A critical instruction is one that lies on this critical path. The essential characteristic of a critical instruction is that an incremental speedup in that instruction creates a speedup overall [Tun01]. Speeding up a noncritical instruction, on the other hand, has no effect on the overall execution time of the program.

Identifying critical instructions in advance is very useful in making other speculation techniques more effective. Since speeding up a noncritical instruction does not produce any benefit, resources are best allocated to speeding up only critical instructions. An example of this is value prediction. Since value prediction carries high misprediction penalties, there is no point in taking a risk by making a value prediction for a noncritical instruction. Performing value prediction only on critical instructions means that the value predictor will have the same performance increase as it would otherwise, in theory, while reducing the number of mispredictions [Tun01]. By the same token, noncritical instructions can be deemphasized without performance cost. An example of this is energy savings. A noncritical instruction can be executed by a slower but more energy efficient functional unit without degrading the overall CPU performance [Gov95]. This saves energy by executing

those instructions more slowly, but does not cost time as those instructions are not on the critical path.

A caveat, however, with this is that a critical instruction can be sped up only so far until it ceases to be on the critical path and a noncritical instruction becomes critical [Tun02]. Likewise, a noncritical instruction can be slowed down only so far until it becomes critical and starts affecting performance. Focusing all the resources on critical instructions while completely neglecting noncritical instructions will only produce so much performance improvement.

An even larger problem, however, is in identifying whether an instruction is critical in advance. The criticality of an instruction needs to be known before the instruction is executed so that the CPU can respond appropriately. However, not only is it impossible to definitely say whether an instruction is critical in advance, it is not even possible to know whether any arbitrary past instruction was critical without running the entire program [Tun01]. This is because the entire dataflow graph for the program needs to be known to know with certainty what the critical path is. When the program is only partially executed, part of the dataflow graph is still unknown. This makes criticality prediction different from other forms of speculation. In branch prediction, for example, the correct result is known after the branch instruction is executed, and the predictor can be trained with an exact result. In criticality, however, the correct result for a particular instruction is never known. Obtaining training data for the criticality predictor is a problem in itself.

As will be described in more depth below, a table-based criticality predictor has been proposed and has been shown to be reasonably accurate in spite of the above

problem. However, the table-based approach is limited in scope and suffers for it. In this chapter I describe how a perceptron can be used to predict instruction criticality and propose several perceptron-based criticality predictor approaches.

6.1. Past Work

6.1.1. Predicting Critical Behavior

The first significant work in predicting whether instructions lie on the critical path was performed by Tune, Calder, and Tullsen [Tun01]. The authors recognize that determining whether an instruction is critical at runtime is not easily possible. Instead, they predict whether an instruction was critical by whether it exhibits behavior that is likely to mean that it is critical. The authors propose several behaviors that would make instructions likely to be critical that are easily measurable after an instruction completes. If an instruction exhibits any one of these behaviors, it is considered critical. However, the criticality of an instruction is known only after the instruction is executing, when it is too late to take advantage of its criticality. The authors consequently propose a table-based prediction methodology, shown in Figure 6.1. The PC is hashed to associate an up-down saturating counter with each static instruction. An instruction's counter is incremented if it exhibits critical behavior, and decremented if it does not. To make a criticality prediction, the counter value is compared to a threshold; if it exceeds the threshold it is predicted critical, otherwise it is predicted noncritical. In their tests, they decided that incrementing by 8, decrementing by 1, a maximum value of 16, and a threshold value of 8 is a good approach. The large increment is used because using the criticality information to

change the way the CPU executes can result in critical instructions acting noncritically. Incrementing by 8 means that an instruction once found critical will be considered critical for the next 8 iterations.

The authors propose five criteria to indicate criticality, of which four were found to be reasonably accurate. Criterion QOLD is met if the instruction is the oldest instruction in the CPU that cannot run because it is dependent on an executing instruction. QOLDDEP is met for any instructions that cause another instruction to meet QOLD. ALOLD is met if the instruction is the oldest executing instruction in the machine. QCONS is met if the instruction has the most consumer instructions of any instruction currently executing. If an instruction has met any of these four criteria by the time it completes execution (it can often meet more than one), it is considered critical; if it never met any of the criteria, it is considered noncritical.

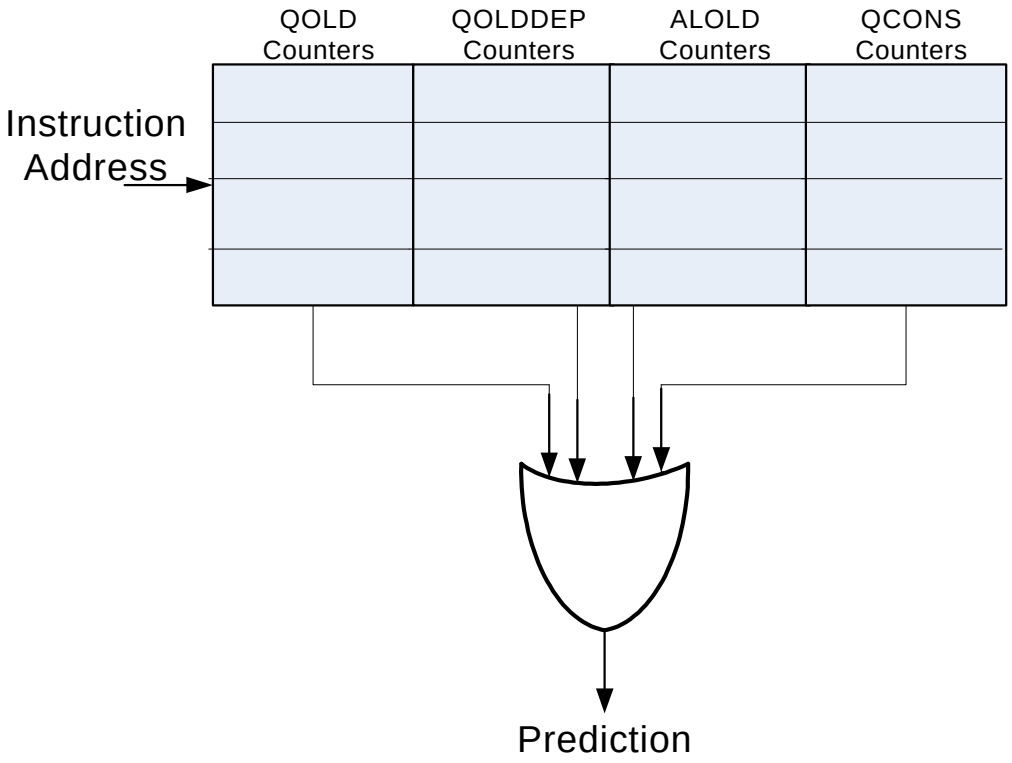


Figure 6.1. Table-based criticality predictor

This prediction approach has some significant disadvantages but one important advantage. The first disadvantage is that the predictor is not trained on whether an instruction was actually critical, but on whether it exhibited behavior likely to mean that it was critical. The second disadvantage is more significant for this dissertation. The predictor only uses local information, the criticality of past iterations of the static instructions, in predicting criticality. It does not use the criticality of other global instructions. While this would not be a problem if the criticality of a static instruction does not change, the authors found that it does; in their studies, they found that 23% of instructions tend to change their criticality over 100 iterations [Tun02]. These instructions would be imperfectly predictable with their local approach.

The big advantage to this criticality prediction approach is its simplicity. Unlike the next approach described below, a criticality estimate can be obtained for every instruction. Consequently, when making predictions, there is substantial information available.

6.1.2. Predicting Criticality More Precisely

An alternative approach by Fields, Rubin, and Bodik tries to measure criticality more exactly with a token-passing algorithm [Fie01]. A token bit is added to each ROB entry so as to be associated with every active instruction. The token bit is normally 0 unless the instruction possesses the token. To determine whether an instruction i is critical, a token is created at i. This token is passed to an instruction j if j's last operand to arrive came from i. This models the longest edge in a dataflow graph. If the token is not passed on and dies, this means that i is not critical. If it is, it

does not necessarily mean that i is critical; however, as time goes on and the token continues to live, it becomes more and more likely that i is critical. The authors propose that i be considered critical if the token survives for 500 dynamic instructions plus every instruction in the ROB. Prediction is then performed as above. The criticality of an instruction is used to increment a counter associated with the static instruction. If the counter exceeds a threshold, it is predicted critical the next time around.

The key advantage to this approach is that it solves the problem of the above approach. Rather than measure whether an instruction behaves like a critical instruction, it actually measures whether it is a critical instruction. The noncriticality of an instruction is known exactly. The criticality of an instruction, while not known with absolute certainty, is known significantly more accurately than it is in the above approach.

This approach unfortunately has two problems. First, an instruction is not known to be critical until many cycles afterwards. This can mean that the static instruction could reoccur several times before its criticality is known. Second, only one instruction can be evaluated for criticality at a time. The authors stretch this by having 8 separate tokens, allowing 8 instructions to be evaluated at once. However, since evaluating an instruction requires an extra bit added to each ROB entry, to say nothing of the token passing hardware, it is impractical to measure whether every instruction is critical. The token-passing approach is consequently best used when the CPU wants to predict the criticality of only a small percentage of the instructions.

A subsequent work by Fields, Bodik, and Hill [Fie02] gave a more accurate analysis of the criticality of an instruction. In their work, they proposed a mechanism for measuring an instruction's slack, or the number of cycles a noncritical instruction can be delayed before it becomes critical and starts affecting performance. Their work significantly builds upon the previous work for two reasons: 1) they determine the degree to which an instruction is critical, and 2) they determine whether an instruction is actually critical or not, rather than whether the instruction exhibits critical behavior. While highly accurate and useful, their approach has two issues. First, it builds upon their token-passing algorithm with all its limitations. Second, their approach for evaluating slack requires that instructions be delayed in order to measure their effect on performance. Delaying every instruction would hurt the CPU performance; consequently they recommend measuring slack only on a static instruction's first iteration. Their algorithm consequently cannot account for changes in a static instruction's criticality from one iteration to the next.

6.1.3. Perceptron Criticality

The objective of the work detailed in this chapter is to build a perceptron-based criticality predictor that outperforms the Tune, Calder, and Tullsen predictor. Their prediction strategy had two important weaknesses. The first, which was mentioned above, is that it trains on criteria which may or may not accurately indicate criticality, rather than training on whether an instruction is actually on the critical path. The second is that when predicting an instruction's criticality, their predictor only looked at the local information, or the criticality of past iterations of that instruction, rather than the global information from the criticality of surrounding

instructions. As the authors themselves found that the criticality of many static instructions change from one iteration to the next, this scope limitation to training on local information only limits the prediction accuracy. A perceptron predictor, however, is not limited to local information, and should conceptually be more accurate.

It is important to point out why this perceptron work is building upon the weaker criticality predicting approach and not the more accurate approach by Fields et al. As mentioned previously, the more accurate approach has two core problems. First, the token-passing approach cannot feasibly be applied to every instruction. This means that criticality can only really be determined for selected important instructions. Whether this is a problem or not is determined by the application; if the application needs to know the criticality of only a handful of instructions, this not a liability. However, if an application needs to know the criticality of every instruction, this approach is useless. Additionally, it is unsuitable for a global predictor, which relies on criticality information coming from many different instructions. Second, because the token-passing approach can only be applied to selected instructions, the authors only use it on the first iteration of a selected instruction. Thus changes in the criticality of that instruction from one iteration to the next are completely ignored. While the criticality of the first instance of the instruction is determined more or less exactly, the criticality of subsequent instances are not known. Third, the token-passing algorithm requires hundreds of cycles of evaluation per-instruction in order to be reasonably accurate. The criticality decision is not known until long after the instruction commits. If an instruction frequently reiterates, many iterations could go

by before the criticality of that instruction is known. This makes dynamic prediction less useful. In contrast, the approach by Tune et al, while less accurate, can be reasonably applied to every instance of every instruction, and produces a result immediately after the instruction commits. For a predictor that needs to know the criticality of practically every dynamic instruction, it is a much more practical approach.

6.2. Analysis

6.2.1. Evaluating Criticality

It is essential to know the accuracy of a predictor in order to evaluate it. A criticality predictor being trained on critical instruction behaviors could be very good at predicting the behaviors, while not predicting criticality very well at all. Fortunately, although the criticality of an instruction cannot be known at runtime, it can be accurately determined at program completion. Recall that by its definition, a reduction in latency of a critical instruction means a reduction in overall program time. If the latency of a particular dynamic instruction is reduced, the program is run, and the overall program latency is also reduced, the instruction is known to be critical. If not, the instruction was by definition not on the critical path. While this information is useless in training a dynamic predictor, it is useful in determining, in retrospect, whether that predictor was accurate.

By how much should the latency of an instruction be reduced? Recall the effect of slack: a critical instruction can be only sped up so much before a noncritical

instruction becomes critical. Consequently, critical instructions should not be sped more than the smallest possible increment, or one cycle.

A criticality predictor can thus, at least in theory, be evaluated fairly simply. Every time that it guesses an instruction is critical, that instruction's latency is reduced by one cycle. The program is run twice; once with criticality prediction and once without. The quantity of instructions sped times one cycle each gives the overall predicted criticality. The decrease in the number of cycles between the program run without criticality prediction and the one with criticality prediction tells how much of that predicted criticality was genuinely critical. Dividing this by the predicted criticality tells the overall accuracy of the predictor for that program.

From a practical standpoint, it is not necessarily easy to reduce an instruction's latency in simulation, especially for instructions that require only one cycle to execute. One way to deal with this is to increase every instruction's execution latency by one cycle across the board. From a graph theoretic point of view, this will have no effect on the critical path [Fie01]. Instructions predicted critical are simply sped back to their original latencies. This is equivalent to increasing the latencies of all instructions predicted noncritical by one cycle each.

6.2.2. The Critical Behavior Criteria

The perceptron criticality predictors that I will propose use the four critical behavior criteria defined by Tune, Calder, and Tullsen to train the predictor: QOLD, QOLDDEP, ALOLD, and QCONS. There are several assumptions that must be made about these criteria. First, they are assumed to be reasonably accurate indicators of instruction criticality. Second, it is assumed that these criteria must be

predicted; they cannot be instantly known when the instruction is fetched. Third, it must be possible to determine when an instruction commits whether it met any of the criteria or not. Fourth, it is assumed that an instruction could meet a criterion on one iteration and not on another; otherwise, criticality would only need to be evaluated once for each static instruction. Fifth, it is assumed that there are correlations between the criticality of nearby global instructions. If these assumptions are not valid, there is little point in creating a global predictor to predict these criteria.

From an intuitive standpoint, it is easy to see why an instruction that meets any of the criteria is likely to be critical. A QOLD instruction, one that becomes the oldest instruction still waiting on a dependency, has the longest latency outgoing edges on the dependency graph of all not-yet-executing instructions. While the longest edges on a graph need not necessarily lie on the critical path, chances are that they do. If a QOLD instruction is critical, so must be at least one of the instructions that sourced it. A QOLDDEP instruction, or a still-active instruction that sources a QOLD instruction, must lie on the critical path if the QOLD instruction is critical, because it is the instruction that the QOLD instruction is waiting on to execute. An ALOLD instruction, the oldest still-executing instruction, is likely to be critical for the same reason that the QOLD instruction is likely to be critical, as it has the longest latency outgoing edges of any instruction in the processor. ALOLD also captures those instructions with long execution latencies, such as floating point instructions and some loads and stores, which QOLD does not capture. The case for a QCONS instruction, the instruction with the largest number of directly consuming instructions,

is somewhat weaker, but from an intuitive standpoint the more outgoing edges an instruction has, the more likely it is that one of those edges lies on the critical path.

The second and third assumptions are clearly true by just looking at the criteria. An instruction can be evaluated as to whether it meets each criterion before the instruction completes writeback. QOLD is known before the instruction issues to a functional unit, ALOLD and QCONS are known while an instruction is executing, QOLDDEP is known before an instruction finishes execution. It cannot be determined conclusively at fetch, however, whether an instruction meets any of these criteria. QOLD and ALOLD depend on how quickly preceding instructions execute; these instruction may not even yet be executing. QOLDDEP cannot be known until a subsequent instruction meets QOLD. While QCONS may be guessed at looking ahead at the code, it is not known for certain which instructions will follow because of control flow uncertainty. Thus each criterion must be predicted.

The actual correlation between each criterion and criticality is evaluated in Table 6.1 as averaged across all eight benchmarks. An extra cycle is added to the normal execution latency of every instruction. If an instruction is marked with the appropriate criterion before or while it is executing, its execution latency is reduced by one cycle. In the ANY case, instructions marked with any criterion are sped up by a cycle. The table shows the percentage of instructions marked with each criterion. Because QOLDDEP, ALOLD, and QCONS may be identified in the last cycle of execution, some marked instructions cannot be sped up in time. The percentage of instructions that were marked but not evaluated are also shown in the table; this shows the level of uncertainty in the evaluation of these criteria.

Two additional runs are performed for each benchmark. The first run runs each instruction at normal latency. This, IPC_{none}, is considered the IPC when every instruction is treated as critical and sped up. The second run runs each instruction with an extra cycle latency. This, IPC_{all} is considered the IPC when no instruction is treated as critical. Using these two IPCs, the expected IPC is determined for each criterion when each instruction marked for that criterion is sped up one cycle. In theory, if a criterion perfectly indicates the criticality of an instruction, the percentage of instructions marked by that instruction, times one cycle saved for each instruction, should equal the percentage increase in IPC. The expected relative IPC is calculated as %marked * (IPC_{none}-IPC_{all}) / IPC_{all}. The actual relative IPC is what was actually observed when every marked instruction is sped up by one cycle, divided by IPC_{all}. The error is the absolute difference between them.

	%marked	%should have been marked	Expected IPC	Actual IPC	Error
AOLD	8.78%	15.47%	104.31%	111.31%	7.00%
QCONS	18.49%	3.24%	104.94%	106.62%	2.68%
QOLD	39.93%	0.00%	110.28%	118.87%	8.58%
QOLDDEP	16.01%	4.63%	105.16%	111.02%	6.35%
Any	54.69%	3.47%	114.27%	125.69%	11.42%

Table 6.1. Correlation of each criterion with actual criticality

6.2.3. Global Correlations

Clearly if many instructions change whether they meet criteria from one iteration to the next, the local history is not an ideal source for predicting criticality. The alternative is the global history. The question now is whether there are correlations between whether nearby global instructions meet each criterion.

Is there an intuitive reason to believe that the criteria-meeting of one instruction correlates with the criteria-meeting of a later instruction? This depends on why an instruction meets a criterion once and does not meet it later. QOLD depends on the execution order of prior instructions. Changes in this order occurs largely from control flow changes. QOLDDEP depends on QOLD, so it changes for the same reasons. ALOLD depends on the execution ordering of prior instructions and on the latency of the instruction itself. The primary case of an instruction's execution latency changing is a memory instruction undergoing a cache miss. Changes in QCONS depends on which instruction follow; this changes on control flow.

Intuitively a program could have two or more alternative critical paths through a section of static code which change due to control flow. The order in which later instructions are executed depend largely on the order in which earlier instructions are executed. If the criteria are good indicators of criticality, whether an earlier instruction is marked for a criterion should be a good indication of whether a later instruction should be marked.

6.3. Perceptron Predictor Configurations

My basic perceptron criticality predictor is organized per-address like the perceptron branch predictor. A table of perceptrons is addressed by the lower bits of the instruction address. A global history, with four bits per past instruction, tells whether each past global instruction met each criterion. A single output tells whether the instruction is to be predicted as critical.

It is expected that predictions will need to be made before the most recent past global instructions are marked. Additionally, some past instructions may have a

marking available on some criteria but not others (for example, QOLD is known before ALOLD). Consequently, the global history includes four more bits telling, for each criterion, whether information is yet available for that instruction.

What differs between my perceptron approaches is how the different criteria are used. Below I propose three perceptron predictors. The first approach uses separate perceptrons to predict each individual criterion and combines the results. The second approach has a single perceptron that combines the criteria at the input. The third approach has a single perceptron with an input for each criterion.

6.3.1. A Perceptron For Each Criterion (PEC)

Figure 6.2 shows the first predictor approach. Each table entry has four separate perceptrons: one for each criterion. Each perceptron is sourced only by its respective criterion; thus the QOLD perceptron would have as each of its inputs whether each past global instruction was marked as QOLD. An instruction is marked as critical if the quantity of perceptrons producing an output of 1 meets or exceeds a fixed threshold. For simplicity, a threshold of 1 is assumed unless otherwise stated; an instruction is predicted critical if any perceptron predicts a criterion. There are two possible training strategies. In PEC_EACH, training is performed by criterion. Each perceptron is trained based on the presence of its own criterion. In PEC_OR, the actual result of whether any criterion was met (the OR of the observed criteria) is used to train every perceptron.

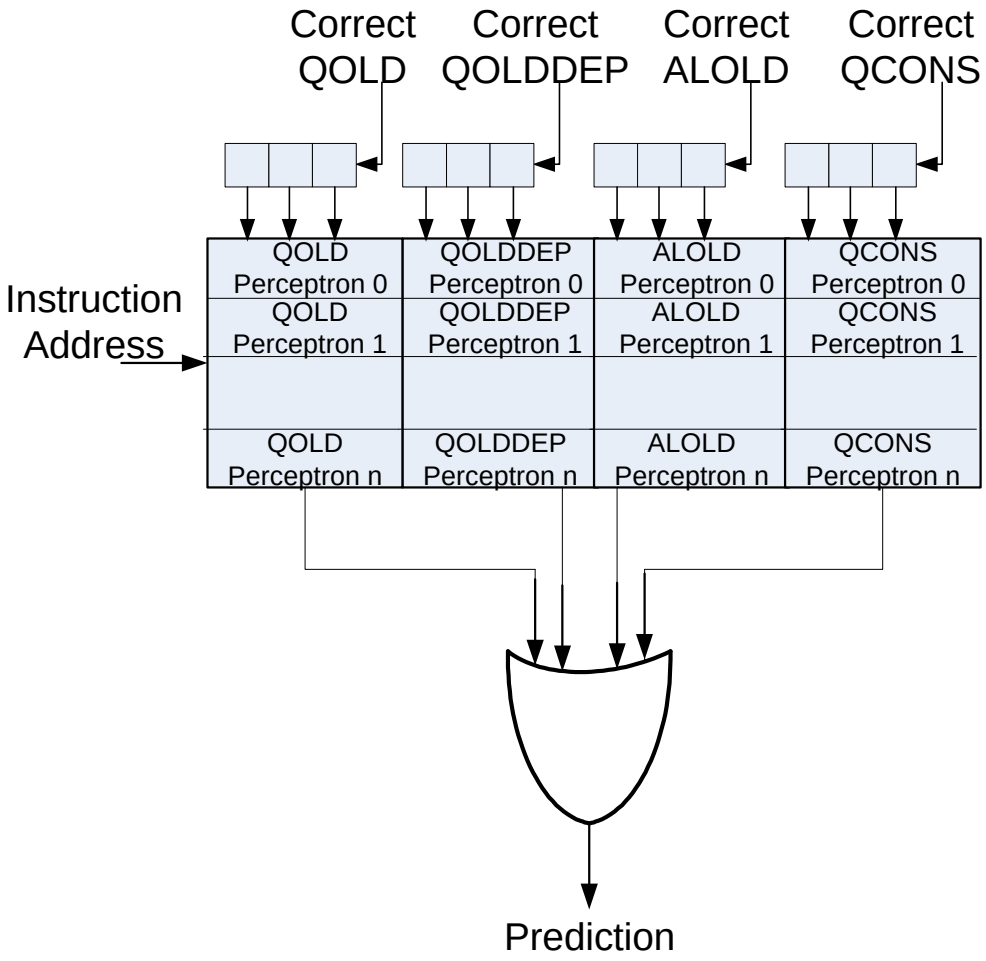


Figure 6.2: Perceptron for each criterion (PEC) criticality predictor

This approach is the closest perceptron analogue to the Tune et al's counter-based approach. It assumes that the four criteria correlate differently from each other, so that it is best to have separate predictors for each one. This has a particular advantage in biasing. Because the criteria appear with different frequency, each needs its own bias to balance the predictor. Having a separate bias weight for each criterion means that the predictor is likely to be better tuned for that criterion.

The key downside to this approach is that some of the criteria occur on significantly fewer than 50% of the instructions. Having a perceptron for each criterion means that the quantity of negative inputs will greatly exceed the quantity of

positive inputs, making for an imbalanced predictor. A second problem is that by giving each criterion its own perceptron, no perceptron can learn correlations between criteria.

6.3.2. A Single Perceptron (SP)

An alternative approach, which compensates for the balancing problem, is to combine the criteria at the input rather than the output. This configuration is shown in Figure 6.3. In this approach, a single perceptron determines whether the instruction is predicted critical or not. Each input to the perceptron is sourced by a single past instruction. If the quantity of criteria marked at that past instruction exceeds a fixed threshold (1 is assumed), a 1 is sourced to that input; otherwise, -1 is sourced. This approach is more balanced because the probability of any input being 1 is much closer to 50%. It also consumes under one-fourth of the storage space of the above approach.

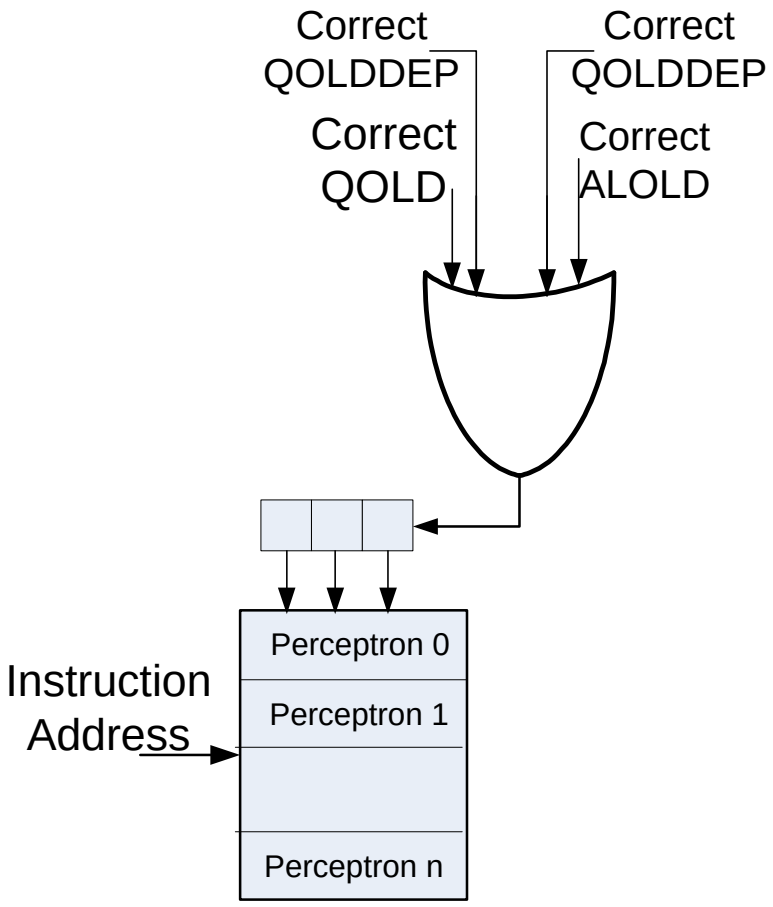


Figure 6.3. Single perceptron (SP) criticality predictor

A drawback to this approach is that the perceptron is unable to capture the correlation of any individual criterion. If one of the criteria is a less reliable global correlator than the other criteria, that criterion could cause the perceptron to mispredict.

A second drawback is that the perceptron cannot be trained on the presence of individual criteria; it must be trained on the OR of the criteria, or whether the quantity of criteria present exceeds a fixed threshold. This means that the perceptron could be dominated by a single criterion such as QOLD and would be less capable of observing the others.

6.3.3. Single Perceptron, Input for Each Criterion (SPC)

A third approach is shown in Figure 6.4. Each table entry contains only one perceptron. However, the perceptron has four inputs for each history entry. Each of the four inputs is sourced by whether the corresponding past instruction was marked for each of the four criteria.

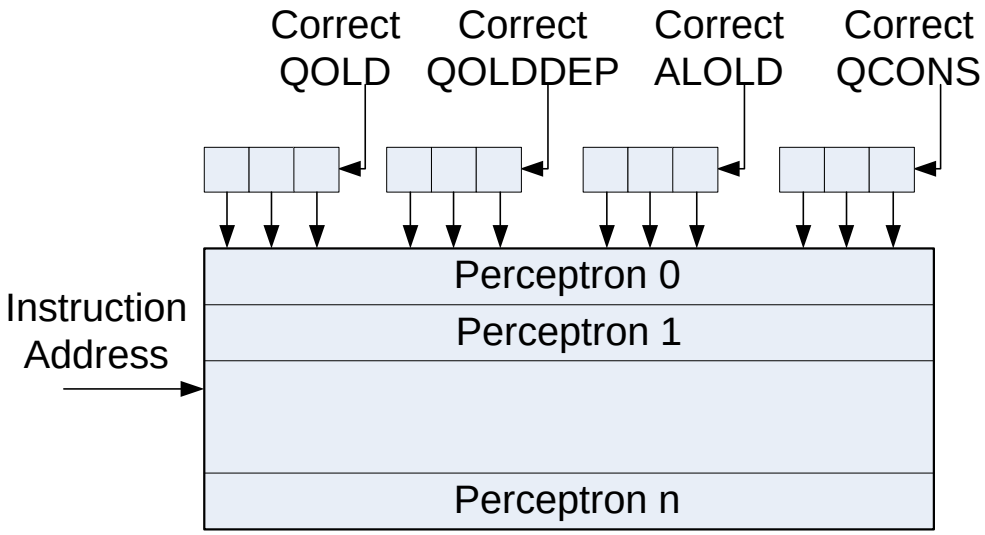


Figure 6.4. Single perceptron input for each criterion (SPC) criticality predictor

The advantage to this approach is that, rather than using a fixed threshold for criteria quantity, the perceptron is able to figure that out itself. It can furthermore determine that a particular criterion is a more important indicator on one instruction, while another criterion is more important indicator on another. The disadvantage, besides its size, is that it suffers from the balancing issue of the first approach, yet has only one bias weight.

Like PEC, there are two training variations on this approach. The first is to train each weight using the OR of the criteria like the predictor in 6.2.2 (SPC_OR). The second is to train each weight using a separate error value based on its own criterion, like in 6.2.1 (SPC_EACH).

6.3.4. Training

A major challenge to perceptron criticality prediction is in training. In branch prediction, the prediction of whether the branch is taken or not has no effect on whether the branch is actually taken. The prediction does not affect the actual branch, only subsequent instructions. However, criticality prediction is actually intended to affect the instruction being predicted. If the prediction is made without changing how the instruction is executed, the instruction can be later identified as meeting a criterion, and the perceptron can be trained as normal. However, what happens if the latency for that instruction is reduced by a cycle? The instruction will most likely not be marked for a criticality criterion. The perceptron will be trained that the instruction was not critical, while in fact the instruction would have most likely been marked for a criterion had it not been perturbed.

In their approach, Tune et al deal with this by incrementing their local counter by 8 on a criterion being marked, while decrementing by 1 on no marking. Since the instruction could only be marked when it was predicted noncritical because of the perturbation, this effectively meant that their predictor was trained once every 8 iterations.

A similar approach can be used for the perceptron approaches. Rather than training on every iteration, training is performed only on specific training iterations. On every training iteration, a perceptron prediction is obtained but not used. The instruction is treated as noncritical, and the perceptron is trained using its prediction.

On other iterations, the perceptron prediction is used in practice but not used in training.

A challenge in this approach is determining when to train the perceptron. Perceptrons cannot be expected to learn a correlation after one iteration; consequently, on training, several training iterations would need to occur sequentially. Since the criticality predictor will be useless during these iterations, they cannot occur too often. However, they need to occur sufficiently often that the perceptron can adapt to changes.

A second approach could mimic the counter approach by using two different error values. If the perceptron predicts noncritical but the actual result is marked, a large error value, such as 8, is used to train. However, if the perceptron predicts critical but the actual result is noncritical, a smaller error value of 1 is used to train.

6.4. Experimental Results

6.4.1. Simulation

The criticality prediction is largely implemented in mycritical.c, with some components implemented in mysimoutorder.c.

Criticality information on an instruction is stored as additional fields in the instruction's ROB entry. Each criterion has two ROB entries: whether the criterion is currently set, which is used to determine whether other criteria must be set, and whether the criterion was ever set for that instruction, which is used in training the predictor. Two additional fields in the ROB track the number of cycles that have elapsed since the instruction was dispatched (cycles_in_ROB), and the number of

cycles that the instruction has been waiting for an instruction on which it is data dependent to complete (cycles_not_ready). Training occurs when the instruction exits the CPU in the commit stage. Criticality predictions are performed for an instruction at the point it is dispatched.

All eligible instructions are analyzed on every cycle to determine if they meet any of the criteria. This analysis occurs in function update_criticality_flags() in mycritical.c. update_criticality_flags() first steps through all the instructions in the ROB and increments cycles_in_ROB and cycles_not_ready as needed. It then steps through each instruction and sets the QOLD flag on the instruction with the largest cycles_not_ready value. If there are multiple instructions with that value, the QOLD flag is set on all of them. Next it steps through each instruction that is still waiting on dependencies, and checks for each parent instruction whether the QOLD flag is set. If so, the QOLDDEP flag is set on that instruction. Third it steps through each instruction to find the instruction with the largest cycles_in_ROB value. That instruction's (or instructions') ALOLD flag is set. Fourth, it steps through each instruction, and looks for all instructions that are dependent on that instruction. The instruction (or instructions) with the greatest number of dependent instructions has its QCONS flag set. Finally, if any of the flags are set for a given instruction, the everset flag is set on that instruction for the appropriate criterion so as to state that that instruction was presumably once on the critical path.

For the analysis in 6.2, the cycle time of an instruction was reduced by one if it meets a criterion, in order to determine the relationship between the criteria and criticality. This is done as follows. An extra cycle is added to the latency of each

instruction. This is accomplished by incrementing the execution latency of each arithmetic unit and the load/store latency. If an instruction is in the execution stage and is currently executing (a busy value of 1), and its time_left flag is at 1 (meaning that it will complete execution in the next cycle), the instruction is evaluated for meeting any of the criteria. If so, the instruction's time_left flag is decremented and it completes execution on that cycle. This effectively reduces the instruction latency by 1 for every critical instruction. If the instruction does not meet any criteria, its time_left flag is not decremented, but it is evaluated again on the next cycle. If at this point it does meet one or more criteria, it is marked as "should have been marked."

6.4.2. Baseline

The baseline against which the perceptron approach is evaluated is the Tune et al saturating counter approach [Tun01, Tun02]. Each counter saturates at 3 and 0. If the counter for any criterion is at 2 or 3, the instruction is marked as critical. Training is performed by incrementing or decrementing the counter for each criterion when it is known whether that criterion was met.

This is a change from the higher saturation level used in the past work, which was chosen so that the counter would continue to predict an instruction as critical even though the results of the counter perturb the instruction and make it noncritical. The change to 3 is made because, at this point, the results of the predictor are not used to perturb the processor. 0-3 yields the most accurate results for this predictor.

6.4.3 Accuracy Results

The first set of results compare the accuracy of the predictor to predict criteria. The predictor is not actually used to change the behavior of the processor. This judges the relative performances of the approaches exclusive of any particular criticality application.

Three perceptron-based global criticality predictors are evaluated. The first (SPC) is a single perceptron with an input for each criterion. Its weights are trained on the OR of the criteria. The second (SP) is a single perceptron with single bit input for each past history; the OR of the criteria at that history point is its input. The third (PEC) has four separate perceptrons, one for each criteria, with the OR of the perceptron outputs determining the prediction. A 256 entry global history size is used as default; the effect when history size is varied is shown in 6.4.6.

Figure 6.5 shows the accuracy of the three perceptron predictors relative to the baseline. A prediction is considered accurate if any criterion was exhibited. Since the predictions are not being used, and do not perturb the processor, the predictors are trained on every iteration. Figure 6.6 shows the balance of the five predictors, or the percentage of the time a predictor correctly predicted “critical” over all the time in predicted correctly. This is compared to the percentage of time each criterion was actually exhibited. On average, SPC_OR predicts with 6.56% better accuracy than the baseline, SP predicts with 4.07% better accuracy, and PEC_OR with 2.87% better accuracy.

It is interesting to note that the SPC_EACH approach performs significantly more poorly than the SPC_OR approach, and the PEC_EACH approach performs slightly more poorly. Why is this? The actual objective of the predictor is to predict

whether any criterion (the OR of the criteria) will occur. With the _OR approaches, the perceptron is actually trained on this information. In the _EACH approaches, the perceptron weights for each criterion are only trained on whether that criterion occurs. The perceptron is thus unable to learn the relationship between the occurrence of each criterion and the OR of the criteria; its learning is limited. This is reflected in the _EACH predictor accuracies.

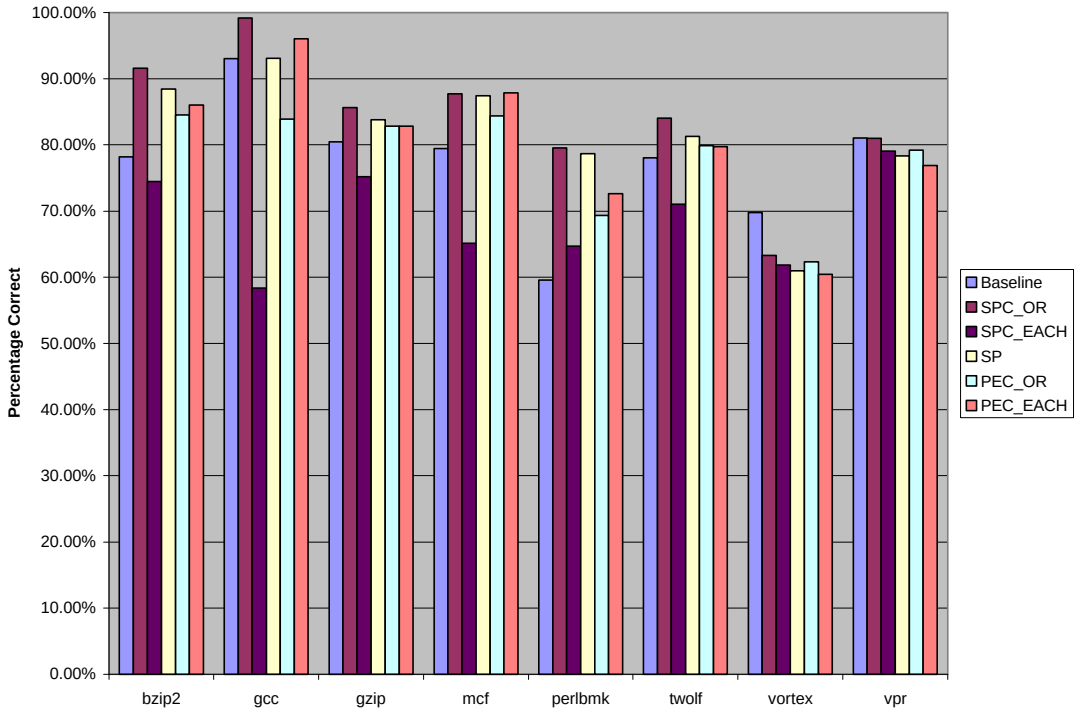


Figure 6.5. Accuracy of the predictors

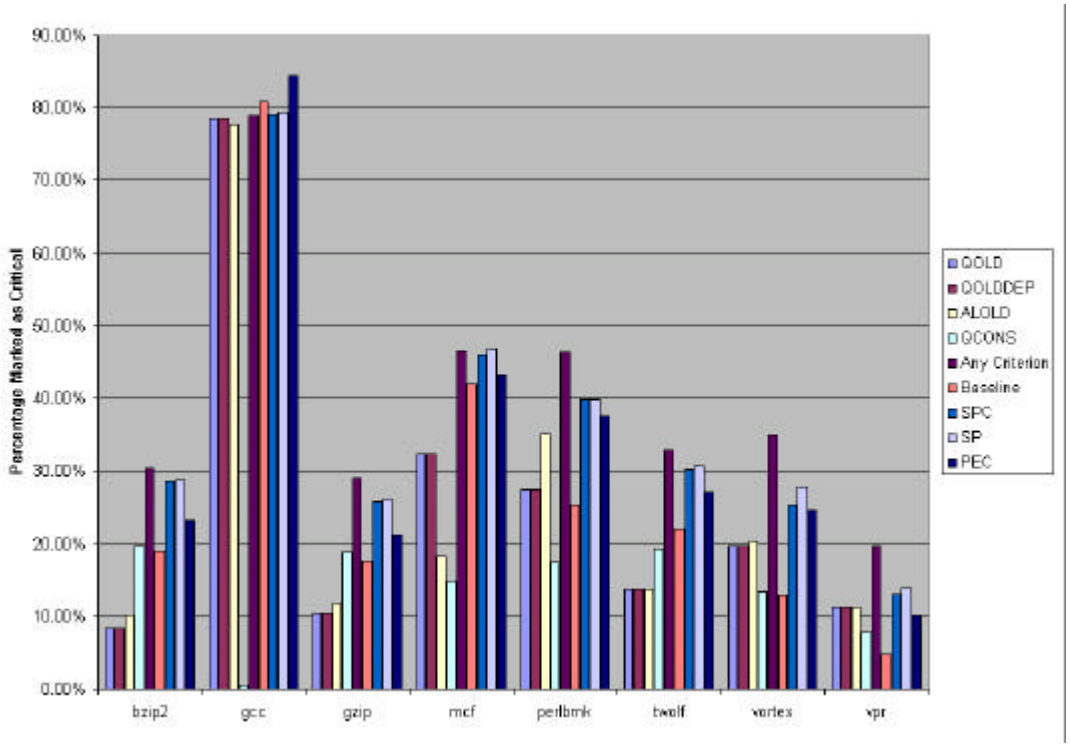


Figure 6.6. Balance of the predictor results

6.4.4. Value Prediction Application

Critical path prediction is never an end in itself; rather, it is intended to be used to make another prediction approach more effective. One prediction approach that can benefit greatly from criticality prediction is value prediction. As mentioned in Chapter 5, any simple implementation of value prediction suffers from a large misprediction penalty. However, value prediction is only beneficial when applied to instructions on the critical path. To produce a value prediction on a noncritical instruction has little chance of improving performance, while subjecting the processor to an unnecessary misprediction risk. If value predictions are performed only on critical instructions, the performance increase from useful predictions should remain

the same, while the misprediction rate, and the resulting performance decrease, should drop.

To test how well the perceptron criticality predictors improve performance when piggy-backed on another application, the criticality predictors are used to make confidence decisions for a value predictor. Value predictions are made, and the value predictor trained, on every instruction. However, the value prediction is used only on those instructions predicted critical. A simple stride predictor is used as the value predictor approach.

If the criticality predictor's results are actually used to change the performance of the processor, the predictor changes the behavior of the instruction it is trying to predict. Consequently the predictor should not be trained on the same iteration that its results are being used. In these tests, the predictors results are used on 3 out of every 4 iterations. On the fourth iteration, the results are not used, value prediction is inhibited, and the predictor is trained. Since the perceptrons tend to learn sufficiently quickly, only one training iteration is performed at a time.

Figure 6.7 shows the accuracy results when the predictor is used as a confidence estimator and is trained on every fourth iteration. Figure 6.8 shows the IPC performance for each prediction scheme. The uninhibited stride predictor's performance is included for comparison. The SPC approach performs at 3.09% higher IPC than the counter, and the PEC performs at 1.44% higher IPC. Both of these exhibit higher accuracies than the counter on average, at 4.06% and 1.10% higher absolute accuracies, respectively. These accuracies are comparable to the accuracies they achieved when criticality information is not used.

SP on the other hand achieves a 9.47% accuracy increase, but strangely suffers a 5.02% decrease in IPC. This is can best be explained by noticing that the high accuracy results mainly from predicting "critical" more often correctly than from predicting "not critical." This means that the value predictor is told to predict much more frequently. These additional instructions need not be correct, as criticality does not infer correct predictions. If the value predictor is not correct on those additional critical instructions, it will mispredict more often, and cause a performance decrease. Since the criticality predictors do not actually produce confidence information for the value predictor, they can cause the value predictor to mispredict more often.

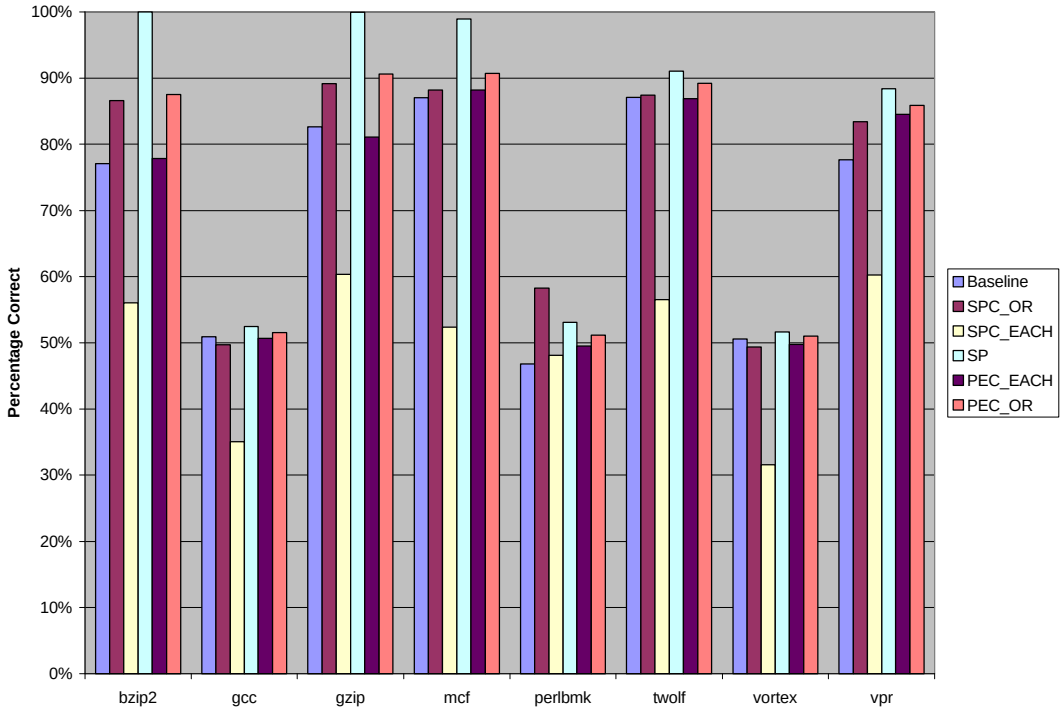


Figure 6.7. Accuracy of each predictor when used to control value prediction

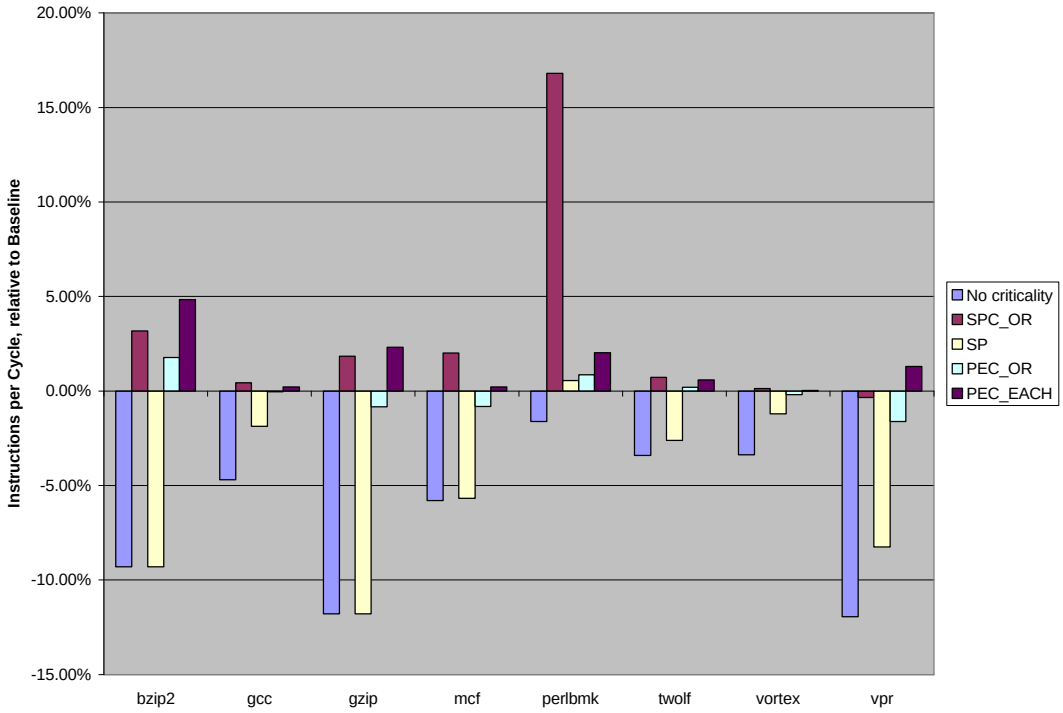


Figure 6.8. Performance when criticality predictors control value prediction

6.4.5. Physical Size

The physical size of the baseline criticality approach is determined by the counter table size *t* and the bit width of the counters *b*. There are 4 counters per entry and *t* entries, so 4*tb* bits are needed. If it is assumed that *b*=2 and *t*=4096, 4kB are required.

The perceptron approaches have two additional parameters: the history size *h* and the perceptron weight width, which is assumed to be 8.

The first perceptron approach, a perceptron-for-each-criterion, requires four perceptrons per entry, each having *h* weights. The global history required is *h* bits for each criterion, or 4*h*. The storage is thus 4*8*ht*+4*h*. If *h*=256, the storage size is 4.2MB.

The second perceptron approach, a single perceptron, requires one perceptron per entry with *h* weights. The global history is only *h* bits total. The storage size is thus 8*ht*+*h*, or 1.0MB for *h*=256.

The third perceptron approach, a single perceptron with inputs for each criterion, requires one perceptron per entry with 4*h* weights. The global history is 4*h* bits total. The storage size is 32*ht*+4*h*, or 4.2MB for *h*=256.

6.4.6. Perceptron Parameters

Figure 6.9 shows the effect of the history size on the SPC predictor. This predictor is chosen because it already has the largest quantity of inputs for each perceptron, and is thus the most sensitive to the negative effects of a large history

size. The 256 history predictor performs with 4.50% greater accuracy over the 64 history predictor, while the 512 history predictor performs with only 5.07% greater accuracy than the 64 history predictor. This shows a larger history size does not necessarily yield significantly better results.

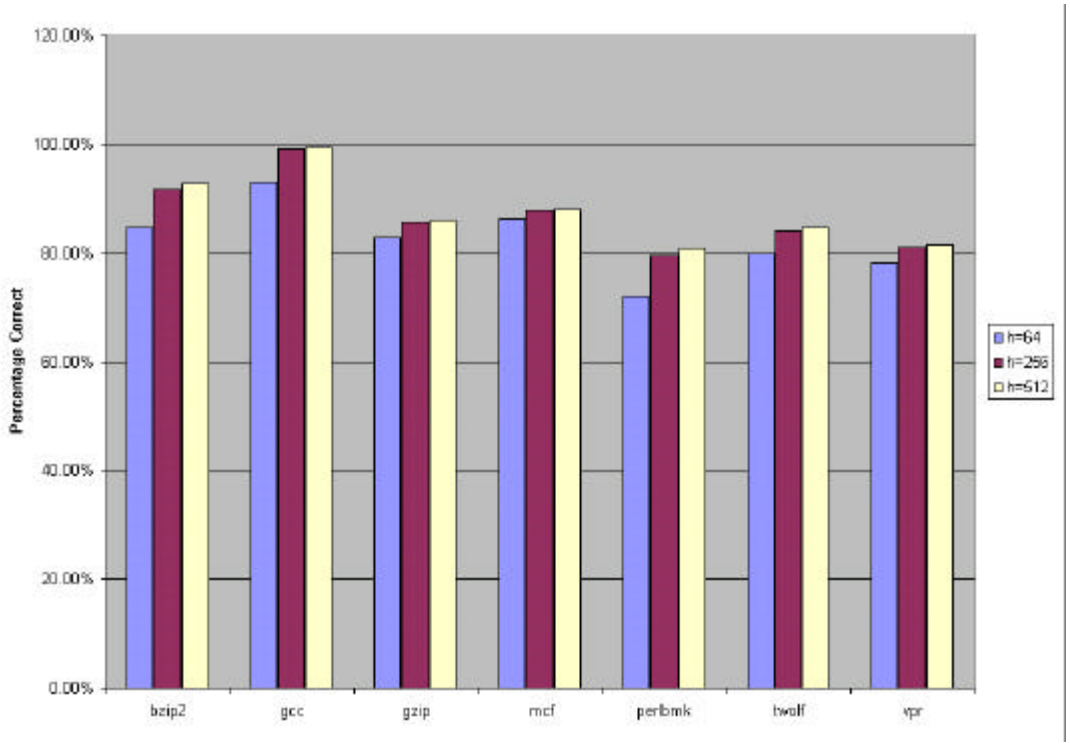


Figure 6.9. Effect of history size on prediction accuracy for SPC

The criticality approaches used above used no interference reduction. Figure 6.10 shows the accuracy comparison for the SPC predictor when the Assigned Seats approach is used to reduce interference in the global history. The average accuracy increase is 2.21%.

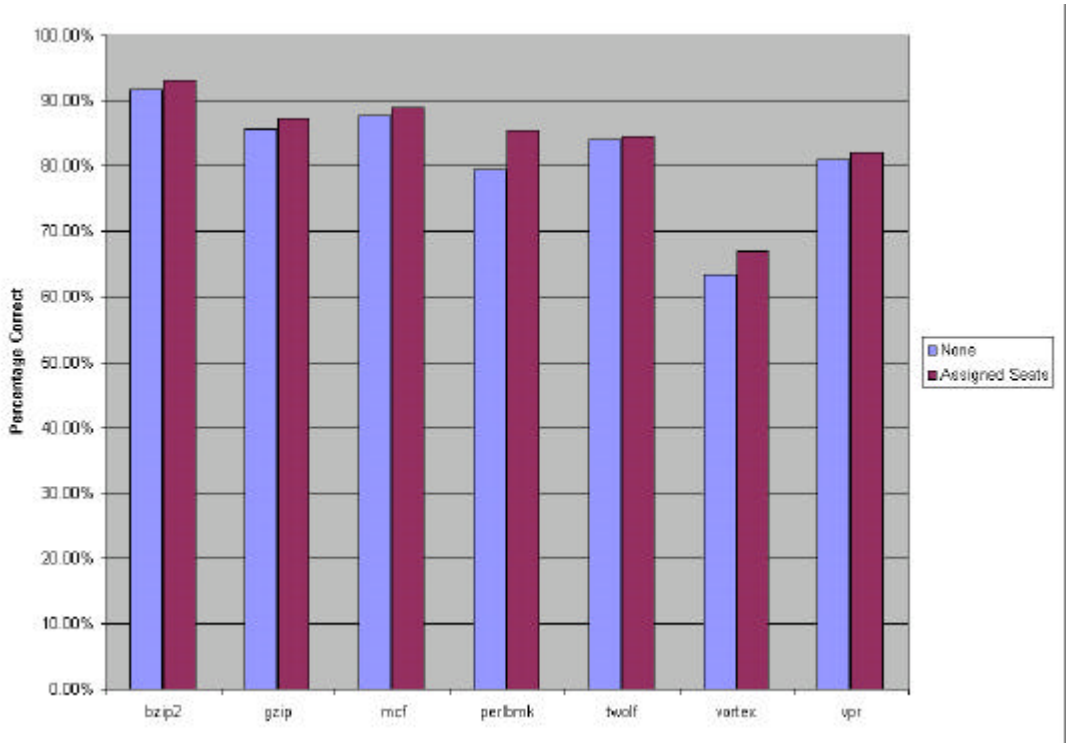


Figure 6.10. Effect of anti-interference on criticality prediction accuracy

Figure 6.11 compares the two training strategies on the SPC. In the tests above, training-by-error was used. The figure compares this with the training-by-correlation approach. Like in value prediction, this training approach performs poorly across most benchmarks, with an average accuracy decrease of 6.57% over the baseline, and an average accuracy decrease of 13.13% over the training-by-error approach. The reasons for this decrease are explored in the next chapter.

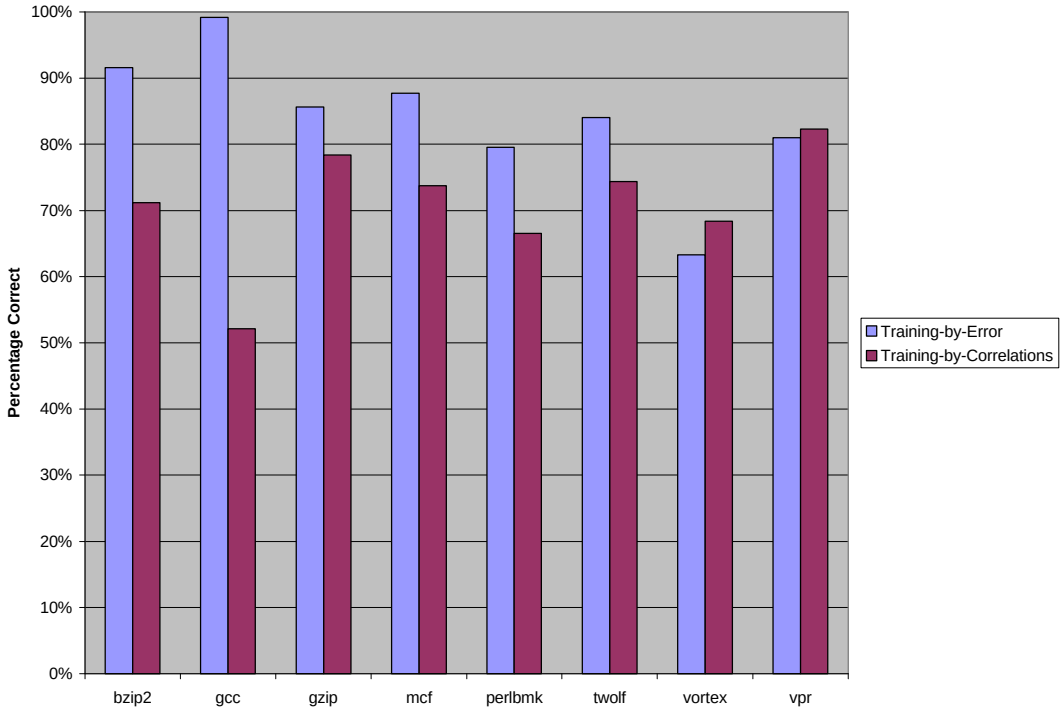


Figure 6.11. Effect of training approach on criticality prediction accuracy

Chapter 7. Conclusions

The value prediction and criticality results presented in Chapters 5 and 6 raise many questions. Why did the disjoint global-local value predictor only perform 6% better than the local predictor, despite its access to significantly more past history information? Why did the training-by-correlation approach, which performed so well for Jimenez's branch predictor, now perform so poorly for value prediction and criticality prediction? Why was the fully connected multibit perceptron approach, with its superior learning capabilities, significantly outperformed by the less capable disjoint approach? In contrast, why did the bitwise perceptron perform so well? It turns out that all of these questions have the same answer. In this chapter I look at the final perceptron weights and return to the earlier theoretical analyses to understand why some approaches succeeded while others failed.

7.1. Weights

7.1.1. Training-by-error

The final perceptron weight values, at the end of simulation runs, give much insight into how and why the perceptrons performed. The following figures show the weight distributions for disjoint global-local, which performed fairly well, fully coupled global-local, which did not, and the SPC criticality predictor.

Figure 7.1 shows the weight distribution, averaged for each benchmark, at the end of 100 million instructions. The weight distribution is computed as the average percentage of each weight value within each perceptron, and is computed for each static instruction when it is either replaced in the table or simulation ends. This is

then averaged across all instructions, and then across all benchmarks, to produce the graph. This graph shows the distribution for the disjoint global-local predictor, the fully-coupled global-local predictor. It also shows the distribution for the disjoint global-local when trained by correlations, and when trained by error using exponential weight growth. Figure 7.2 shows the weight distribution for the SPC criticality predictor for both training-by-error and training-by-correlations.

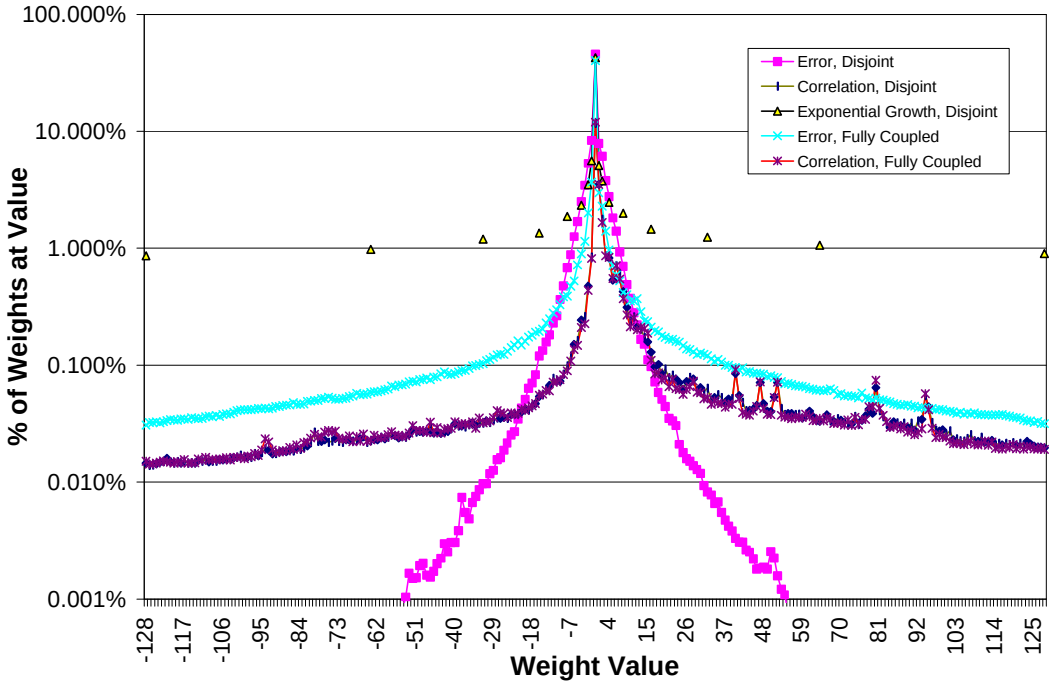


Figure 7.1. Perceptron weight distribution for the Global-Local predictor

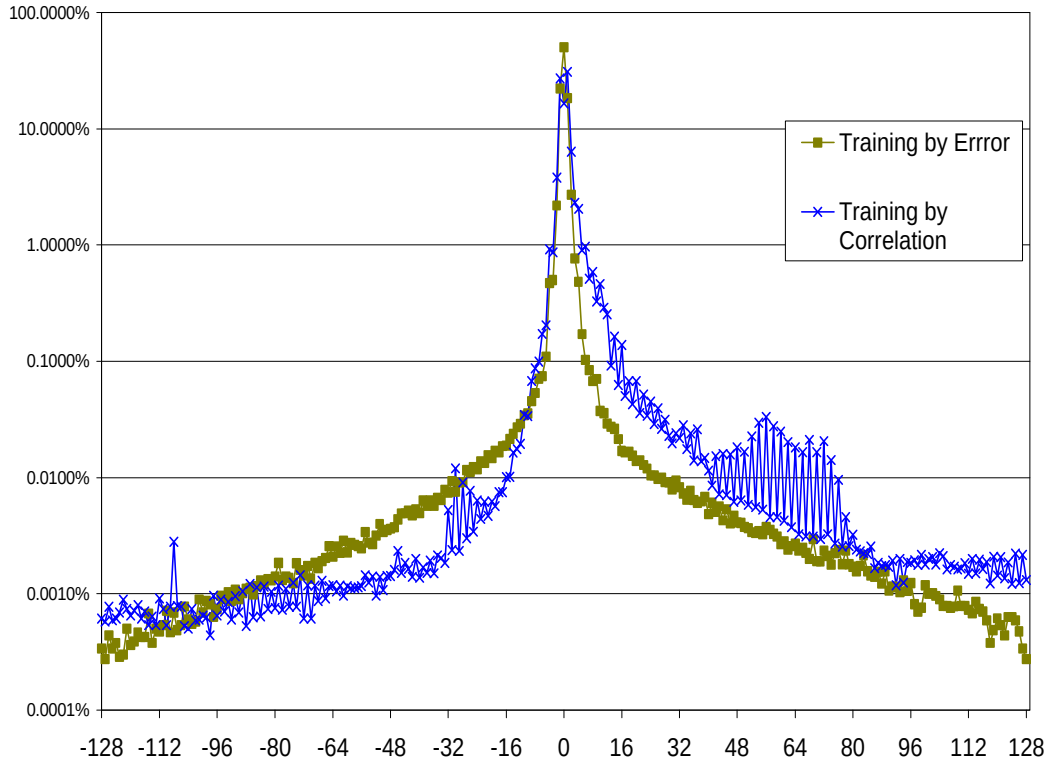


Figure 7.2. Perceptron weight distribution for the SPC criticality predictor

Notice that the weight distribution for the disjoint perceptron using training-by-error is almost entirely close to zero. In contrast, the distribution for training-by-correlations is spread out, with some weights becoming large. This is expected, as training-by-error grows until it predicts correctly and stops, while training-by-correlations keeps growing. The training-by-error with exponential weight growth exhibits the same general shape as linear growth, except that it is spread out because of the faster growth, and clumped at powers of 2, because weights cannot be non-powers of 2.

The interesting case is that of fully coupled, which is spread out even in the training-by-error approach. How does a weight distribution for training-by-error

become broadly distributed? The only method is through frequent prediction errors, which drive up the values of the weights, as weights do not change on correct results. Thus most of the fully coupled perceptrons are having difficulty learning. This implies that very few of the fully coupled single bit-inputs are conflict-free or correlated compared to the disjoint perceptron.

Finally notice that the smaller positive weights tend to be bigger than the smaller negative weights for training-by-correlation by nearly half an order of magnitude. Since this occurs on training-by-correlation, and not on training-by-error, it implies that there are large quantities of false positive correlations. How do we know this? Because when using training-by-correlations, false correlations cause the falsely correlated weights to keep growing; training-by-error does not, and eventually corrects the weights by returning them to zero. Notice how training-by-error has half an order of magnitude more weights at zero than training-by-correlations. These additional weights at zero represent the false correlations that training-by-correlations made positive.

Figure 7.3 shows the average accuracies of weights at different values for the global-local value predictor approaches. The accuracy of a weight is determined by the percentage of the time the weight's correlation and input matches the actual value (independent of the accuracy of the perceptron as a whole). Notice that weights at 0 cannot have an accuracy, because a 0 weight has no correlation.

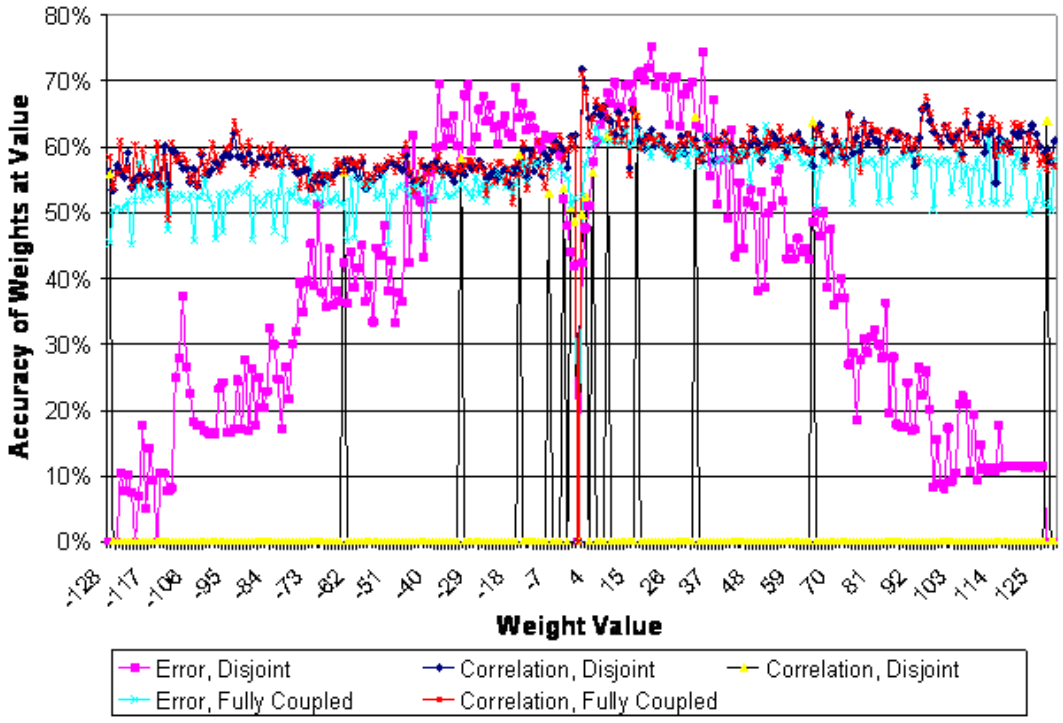


Figure 7.3. Weight accuracies by final value for the Global-Local predictor

Notice how different the accuracies are for large weights in training-by-error and training-by-correlations. True, the very large weights in training-by-error perform with poor accuracy, but in Figures 7.1 and 7.2 we see that they occur very rarely. The large weights that do occur are between -10 and -40 and 10 and 40. These should be expected to perform better than the near-zero weights, and they do for training-by-error, reaching approximately 65-75% accuracy. However, for training-by-correlations, there is no correlation between weight magnitude and accuracy, with all large weights performing at about 60% accuracy. Why is this? Because with training-by-correlation all weights exhibiting a correlation grow, whether the correlation is true or false due to imbalance at an input. The distribution of weights consequently show all true or false correlated weights at the magnitude when the static instruction stops running. These are uniform because the entire

number of iterations of static instructions tends to be generally uniform between 1 iteration and 128 iterations. The result is that these false correlations are exhibiting as much control over the output in training-by-correlations as the true correlations. A large quantity of the patterns would appear to be unlearnable with training-by-correlations.

The criticality results in Figure 7.2 sheds light on its behavior as well. Recall that the SPC predictor performs much better with training-by-error than by training-by-correlation, but it is capable of handling longer history lengths (such as 512) without the accuracy decreasing. The weight graph for SPC shows two things. First, it tends to be better correlated than global value prediction because training-by-error's weights tend to largely be very small, meaning that it is not having difficulty learning. Second, it tends to be very imbalanced, since training-by-correlation's positive weight values are an entire order of magnitude higher than its negative weight values. What does this show? First, since it is better correlated than global value prediction, noise is less of a problem, allowing longer history lengths to be considered. Second, since it is highly imbalanced, and still relatively poorly correlated, training-by-correlation has difficulty learning patterns.

7.1.3. Implications

The following can be seen in the above weight plots:

- These global applications, particularly value prediction, are poorly correlated, because the overwhelming majority of the weights are at 0 for the training-by-error, and the large weights are inaccurate for the training-by-correlations.

- They suffer from false correlations due to imbalancing, because training-by-correlations has more positive weights than negative, while training-by-error does not. If these were true correlations, training-by-error would also exhibit more positive weights.
- Fully coupled global-local is more poorly correlated than disjoint global-local, otherwise training-by-error would have learned it with smaller weight magnitudes.

Recall that poor correlation occurs when conflicts occur in most or all of the inputs. Patterns can still be learned under these circumstance; after all, a perceptron can theoretically learn a set of patterns from one correlated input. However, when the majority of the inputs are conflicted, problems happen that do not occur when most of the inputs are correlated. Two of these problems explain some of the poorer results. First, the learning time is increased when most inputs are conflicted. This is because weights need to be trained to overcome the noise. Second, training-by-correlation can become incapable of learning compatible patterns when well over a majority of weights are conflicted.

Imbalance occurs when one pattern occurs significantly more often than another pattern. It was showed before that imbalance does not tend to affect the perceptron training time. However, imbalance can cause training-by-correlation to be unable to learn compatible patterns when a majority of weights are conflicted, because it creates false correlations at some of the bits that swells the weights.

The combination of imbalance and poor correlation explains why training-by-correlation performs poorly in value prediction and criticality. Both applications are

in general poorly correlated, since relatively few global instructions share the same data values. In contrast, branch prediction is a well correlated application, as a majority of the past branches tend to exhibit a correlation with the target branch [Jim02]. Local value prediction is likewise a well correlated application, which is why a simple lookup table tended to perform well. Training-by-correlation can consequently learn all the compatible patterns in branch prediction and value prediction, and can do so more accurately due to its superior learning rate. However, because global value prediction and criticality are poorly correlated, training-by-correlation tends to be unable to learn even the compatible patterns.

Fully coupled global-local performs more poorly than disjoint global-local because of the low perceptron learning rate due to low correlations. Why is fully coupled more poorly correlated than disjoint in this application? The answer is because while the quantity of weights is increased in fully coupled, the quantity of correlated global inputs is not increased. In value prediction, the global history only contains a certain percentage of correlated instructions for a given history. A fully coupled perceptron has five times as many weights as a disjoint perceptron, yet the number of correlated past instructions does not change. Thus the percentage of correlated inputs to the perceptron is effectively reduced to one-fifth that of disjoint, and the learning rate suffers accordingly.

Why does the bitwise perceptron perform well while having even more weights than the fully coupled perceptron? Notice that the bitwise perceptron is in fact comprised of 32 single-bit perceptrons. Each of these single-bit perceptrons have only one input and one weight for each past value in the history, rather than the five

inputs and weights of the fully coupled perceptron. Thus the bitwise perceptron has the same percentage of correlated inputs for each of component perceptrons as the disjoint perceptron does.

7.1.4. Lessons

As mentioned above, the fundamental difference between global value prediction and criticality on one hand and branch prediction on the other hand is the percentage of correlated inputs. Value prediction and criticality tend to have a low percentage of correlated inputs, while branch prediction has a higher percentage. This affects the type of perceptron approach that is suitable for each application. Branch prediction does well with training-by-correlation, as does local value prediction. Global value prediction and global criticality do poorly with it. Perceptron branch predictors perform significantly better with a history of 64 over a history of 32. Value prediction and criticality perform only marginally better with a history of 64, and in some cases worse. This is not to say that value prediction and criticality cannot use perceptron approaches. As shown in the previous chapters, they can with a reasonable performance increase. However, they do not respond nearly as well to perceptron approaches as does branch prediction.

The first lesson that can be learned from this is that it is important, prior to applying perceptrons to an application, to determine if the application's past values tend to be highly correlated or poorly correlated. This affects the training style, the best multibit topology, and the optimum history size. An application with a well correlated history should focus on maximizing its learning rate. It can therefore use training-by-correlations, which learns faster in the face of imbalance and noise. It can

use a fully coupled multibit perceptron that can use its many weights to learn more patterns. An application with a poorly correlated history, however, should focus on increasing as much as possible its percentage of correlated inputs on each perceptron. It should in general ensure that each perceptron has only one input per history entry. And it must use training-by-error.

The second lesson is that there is a real limit to the useful history size. While it is tempting to suppose that a perceptron, because of its linear growth, can consider hundreds or thousands of inputs, such a design would yield a poor predictor. In most applications, including branch and value prediction, the less recent instructions correlate more poorly than more recent instructions. As the history size grows, the overall percentage of correlated perceptron inputs decreases. At a certain point, therefore, the perceptron learning rate becomes sufficiently poor that its accuracy begins to decrease, rather than increase.

7.2. Summary

7.2.1. Perceptron Context Learning

The first contribution of this dissertation was an analysis of how perceptrons learn context patterns when they directly replace a lookup pattern table. Perceptrons look for correlations between each bit of the pattern and the target. If a correlation exists among the patterns for at least one bit, the perceptron can theoretically learn the pattern set; otherwise, the patterns are in conflict, and the perceptron is not guaranteed to learn them. However, even if the perceptron can theoretically learn the

pattern set, it may not learn it quickly. I show that perceptron learning improves dramatically as the number of bits that correlate increase.

Two weight training strategies are compared: training-by-correlation, which adjusts each weight on every iteration according to the perceived correlation, and training-by-error, which trains the weights only in response to a misprediction. I show that training-by-correlation learns faster than training-by-error and responds better to imbalance between patterns. However, training-by-correlation may never learn a set of compatible patterns if over 50% of the pattern bits are in conflict, and the patterns are imbalanced, with one pattern occurring much more often than another. In contrast, training-by-error will always learn compatible patterns, regardless of the imbalance.

7.2.2. Value Prediction

The local table-based context-based predictor [Wan97] is generally considered one of the best practical value predictors. I propose two perceptron-based local value predictors that are based on the table-based predictor. The first, which replaces the counters in the pattern table with perceptrons, has a 1.4% to 2.8% lower accuracy than the table-based predictor. The second, which eliminates the second-level pattern table, and uses the local value history to train, is capable of considering significantly longer local histories than the table-based predictor. It performs with 2.4 to 5.6% better accuracy, and 0.5 to 1.2% higher instructions-per-cycle.

I propose a perceptron-based predictor that uses the past global value history to choose a past local value. I use three different perceptron topologies to learn multiple-bit value correlations: a disjoint topology that considers correlations only

between corresponding bits of the different inputs, a fully-coupled topology that considers correlations between all bits of the different inputs, and a weight-per-value topology that considers correlations between past values for each input. The global-local predictor using disjoint perceptron achieves an average accuracy increase of 3.12% and an average relative performance increase of 1.59%, with a storage requirement of 1.18MB. With a weight-per-value perceptron it achieves an accuracy increase of 10.67% and a performance increase of 4.36%, but with a prohibitive storage of 21.5MB. However, with a fully-coupled perceptron it performs more poorly, with an accuracy decrease of 6.83% and a performance decrease of 1.48%. This is due to the fully-coupled perceptron having a substantially higher percentage of uncorrelated inputs. These are compared to the table-based predictor with a history size of 4 and a history size of 8; the first consumes 69.9kB of storage and the second 33.7 MB of storage, however, they both perform within 0.26% of each other.

I propose a perceptron-based predictor that uses the past global value history to choose a value from a global value cache. When implemented using a disjoint perceptron topology, it achieves an average accuracy increase of 7.56% and a performance increase of 6.69%, with a storage of 1.31 MB.

I finally propose a bitwise perceptron-predictor that does not save past values, but instead learns correlations between individual bits of each past value and the target values. This perceptron achieves an accuracy increase of 12.67% and a performance increase of 5.28%, while requiring a storage of 4.19MB.

Training-by-error is used as a training strategy for each predictor. Both training strategies are evaluated on the global-local predictor; training-by-correlation

performs with 4.82% lower accuracy than training-by-error. This is due to the low percentage of correlated inputs in global value prediction. Exponential weight growth is also considered on the global-local predictor, but it results in an accuracy decrease of 4.51%.

7.2.3. Criticality Prediction

The counter-based critical criteria predictor is the only implementable critical path predictor that can make predictions for every instruction. The predictor is limited, however, to considering only the local past history when making predictions. I propose three perceptron critical criteria predictors that can use a global past history when making predictions.

The first predictor (PEC) contains a perceptron for each criterion, and uses the OR of the perceptrons to produce a prediction. It achieves an average accuracy increase of 2.87% with a storage of 4.2MB. The second predictor contains a single perceptron, and uses the OR of criteria at each past instruction as input. It achieves an accuracy increase of 4.07% with a storage of 1.0MB. The third predictor contains a single perceptron with an input for each criterion of each past instruction. When trained with the OR of the criteria, it achieves an accuracy increase of 6.56%, and a storage requirement of 4.2MB. For contrast, the counter predictor requires 4kB of storage. The storage of the criticality predictors is directly proportional to history size; they can be reduced significantly if a smaller history is used than 256.

Training-by-error is used as a training strategy for each predictor. Both training strategies are evaluated on the single perceptron with an input for each criterion predictor; training-by-correlation performs with 13.13% lower accuracy than

training-by-error. This is due to the low percentage of correlated inputs in global criticality prediction, and the high input pattern imbalance.

7.2.4. History Interference

Interference between past instructions in the global history can cause the performance of global predictors to suffer. I consider two anti-interference measures. The first, "Assigned Seats", uses the lower instruction address bits to assign the past value to a specific entry in the global history register. The perceptron itself is unchanged. In the second, "Piecewise Linear", each perceptron has multiple weights for each history entry. The lower instruction address bits are used to choose which weight is used. The two anti-interference measures are compared on the global-local value predictor. Assigned Seats performs with 0.9 to 2.2% better accuracy for both applications while incurring negligible extra storage costs. Piecewise Linear performs with 4% better accuracy in value prediction but at the cost of 32 times as much storage as an implementation with no anti-interference measures. Because Piecewise Linear results in only modest additional improvement with significant additional hardware, Assigned Seats is generally recommended as a better anti-interference approach unless high prediction accuracy is critical to the application.

7.3 Future Work

While there are many potential areas of future work, I will mention four in particular which I believe worthy of study.

The first area of future work is applying the studies from Chapter 3 to the already existing fields of perceptron-based branch prediction, perceptron-based

branch confidence estimation, and perceptron-based confidence estimation for value prediction. Of particular interest is the degree of correlation in these applications. How much noise is present from uncorrelated weights? How much imbalance is present between patterns? How many patterns occur per perceptron input on average? An analysis answering these questions might be used to substantially improve the existing implementations.

In this dissertation, critical-path prediction was used to improve the performance of value prediction. Because of the severe misprediction penalty, value prediction tends to nearly always perform better when fewer predictions are made, whether they are on the critical path or not. A second future work could look at applying criticality prediction to other applications, such as power reduction, selecting functional units, and selectively applying performance increasing measures such as branch prediction.

One of the limitations of the value prediction work is that a ReFetch misprediction policy is used. In the absence of confidence estimation, ReFetch typically performs worse than no value prediction. A third area of future work is to implement a ReExecute policy which reduces as much as possible the misprediction penalty. A ReExecute method with minimal additional hardware and a single cycle penalty or less is the objective.

There are several future perceptron applications. One promising application is frequency scaling on cache misses. In this application, the processor frequency is reduced at the beginning of an L1 cache miss and sped up at the end of the miss, saving CPU energy. The challenge is that the processor is not necessarily idle on

every cache miss; consequently, not all cache misses should be slowed. A perceptron could be used to predict whether the CPU will be idle on a particular cache miss, and make the decision of whether to slow the processor.

References

[Akk04] H. Akkary, S. Srinivasan, R Koltur, Y. Patil, and W. Refaai. "Perceptron-Based Branch Confidence Estimation." International Symposium on High Performance Computer Architecture, 2004.

[Bha93] S. Bhama and H. Singh. "Single Layer Neural Networks for Linear System Identification Using Gradiant Descent Technique." IEEE Transactions on Neural Networks, Vol. 4, No. 5, September, 1993.

[Bla05] M. Black and M. Franklin. "Applying Perceptrons to Computer Architecture." Proceedings of the Second International Conference on Intelligent Sensors and Information Processing, Jan. 2005.

[Bla05_2] M. Black and M. Franklin. "Neural Confidence Estimation for More Accurate Value Prediction." International Conference on High Performance Computing, 2005.

[Bla04] M. Black and M. Franklin. "Perceptron-based Confidence Estimation for Value Prediction." International Conference on Intelligent Sensors and Information Processing, Jan. 2004.

[Bla03] M. Black. "Perceptron-based Global Confidence Estimation for Value Prediction." M.S. Thesis, Department of Electrical and Computer Engineering, University of Maryland, June 2003.

[Burg] D. Burger and T. M. Austin. "The SimpleScalar Tool Set, Version 2.0." Technical report, University of Wisconsin-Madison Computer Science Department.

[Bur02] M. Burtscher, B. G. Zorn. "Hybrid Load Value Predictors." IEEE Transactions on Computers, July, 2002.

[Bur98] M. Burtscher, B. G. Zorn. "Load Value Prediction Using Prediction Outcome Histories." Technical Report CU-CS-873-98, University of Colorado at Boulder, Oct. 1998.

[Bur99] M. Burtscher and B. G. Zorn. "Prediction Outcome History-based Confidence Estimation for Load Value Prediction." Journal of Instruction Level Parallelism, May 1999.

[Cal98] B. Calder, G. Reinman, and D. Tullsen. "Selective value prediction." Technical Report UCSD-CS98-597, University of California, San Diego, Sep. 1998.

[Cav97] J. Cavazos, D. Stefanovic. "Adaptive Prefetching using Neural Networks." Proposal to NEC, 1997.

[Eve98] M. Evers, S. Patel, R. Chappell, Y. Patt. "An Analysis of Correlation and Predictability: What Makes Two-Level Branch Predictors Work." Proceedings of the 25th Annual International Symposium on Computer Architecture, 1998.

[Fie01] B. Fields, S. Rubin, and R. Bodik. "Focusing Processor Policies via Critical-Path Prediction." International Symposium on Computer Architecture, 2001.

[Fie02] B. Fields, R. Bodik, and M. Hill. "Slack: Maximizing Performance Under Technological Constraints." 29th International Symposium on Computer Architecture, 2002.

[Gal90] S. Gallant. "Perceptron-Based Learning Algorithms." IEEE Transactions on Neural Networks, Vol. 1, No. 2, June, 1990.

[Gov95] K. Govil, E. Chan, and H. Wasserman. "Comparing Algorithms for Dynamic Speed-Setting of a Low-Power CPU." ACM International Conference on Mobile Computing and Networking, Nov. 1995.

[Gru98] D. Grunwald, A. Klauser, S. Manne, and A. Pleszkun. "Confidence Estimation for Speculation Control." 25th International Symposium on Computer Architecture, Barcelona, Spain, June 1998.

[Hen96] J. Hennessy and D. Patterson. "Computer Architecture: A Quantitative Approach." Morgan Kaufmann Publishers, Inc., 1996.

[Ipe04] E. Ipek, S. McKee, M. Schulz, S. Ben David. "On Accurate and Efficient Perceptron-Based Branch Prediction." Cornell University; Unpublished, 2004.

[Jac96] E. Jacobsen, E. Rotenberg, and J. E. Smith. "Assigning Confidence to Conditional Branch Predictions." 29th Annual International Symposium on Microarchitecture, Paris, France, Dec. 1996.

[Jim00] D. Jimenez and C. Lin. "Dynamic branch prediction with perceptrons." Technical Report TR2000-08, Dept. of Computer Sciences, University of Texas at Austin, 2000.

[Jim03] D. Jimenez. "Fast Path-Based Neural Branch Prediction." International Symposium on Microarchitecture, 2003.

[JimKec00] D. Jimenez, S. Keckler, and C. Lin. "The Impact of Delay on the Design of Branch Predictors." 33rd Annual International Symposium on Microarchitecture, Dec. 2000.

[Jim02] D. Jimenez and C. Lin. “Neural methods for dynamic branch prediction.” ACM Transactions on Computer Systems, 20(4), Nov. 2002.

[Jim01] D. Jimenez and C. Lin. “Perceptron learning for predicting the behavior of conditional branches.” Proceedings of the International Joint Conference on Neural Networks (IJCNN-01), July 2001.

[Jim05] D. Jimenez. “Piecewise Linear Branch Prediction.” International Symposium on Computer Architecture, 2005.

[Jim03] D. Jimenez. “Reconsidering Complex Branch Predictors.” International Symposium on High Performance Computer Architecture, 2003.

[Kuv96] L. Kuvayev. “Using Neural Networks for Branch Prediction.” Seminar in Applying Learning to Systems Problems, 1996.

[Lai00] A. Lai and B. Falsafi. “Selective, Accurate, and Timely Self-Invalidation Using Last-Touch Prediction.” International Symposium on Computer Architecture, 2000.

[Lev91] D. Levine. “Introduction to Neural and Cognitive Modeling.” Lawrence Erlbaum Associates, Hillsdale, NJ, 1991, pp. 19-26.

[Lip96] M. H. Lipasti and J. P. Shen. “Exceeding the Dataflow Limit via Value Prediction.” Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture, Dec., 1996.

[Lip96_2] M. H. Lipasti, C. B. Wilkerson and J. P. Shen. “Value Locality and Load Value Prediction.” Seventh International Conference on Architectural Support for Programming Languages and Operating Systems, Oct.1996.

[Mar99] P. Marcuello, J. Tubella, A. Gonzalez. “Value Prediction for Speculative Multithreaded Architectures.” International Symposium on Microarchitecture, 1999.

[McF93] S. McFarling. “Combining Branch Predictors.” Technical Report TN-36m, Digital Research Laboratory, June 1993.

[Min69] M. Minsky and S. Papert. “Perceptrons.” MIT press, 1969.

[Nag91] G. Nagy. “Neural Networks - Then and Now.” IEEE Transactions on Neural Networks, Vol. 2, No. 2., March, 1991.

[Nai95] R. Nair. “Dynamic path-based branch correlation,” Proceedings of the 28th Annual International Symposium on Microarchitecture, 1995.

[Nak99] T. Nakra, R. Gupta, and M. L. Soffa. “Global context-based value prediction.” Proceedings of the 5th International Symposium on High Performance Computer Architecture, Jan. 1999.

[Rei98] G. Reinman and B. Calder. “Predictive techniques for aggressive load speculation.” 31st International Symposium on Microarchitecture, Dec. 1998.

[Ros62] F. Rosenblatt. “Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms.” Spartan, 1962.

[Rus95] S. Russell and P. Norvig. “Artificial Intelligence: A Modern Approach.” Prentice-Hall, Inc., Upper Saddle River, NJ, 1995, pp. 563-593.

[Saz97] Y. Sazeides, J. E. Smith. “Implementations of Context Based Value Predictors.” Technical Report ECE-97-8, University of Wisconsin-Madison, Dec. 1997.

[Saz97_2] Y. Sazeides and J. E. Smith. “The Predictability of Data Values.” Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture, Dec. 1997.

[Sch92] R. Schalkoff. “Pattern Recognition: Statistical, Structural, and Neural Approaches.” John Wiley and Sons, New York, 1992, pp. 204-220.

[Sch96] J. Schurmann. “Pattern Classification.” John Wiley and Sons, New York, 1996, pp. 193-194.

[Seb62] G. Sebestyen. “Decision-Making in Pattern Recognition.” Macmillan Company, New York, 1962, pp. 108-112.

[Sec96] S. Sechrest, C. Lee, and T. Mudge. “Correlation and Aliasing in Dynamic Branch Predictors,” Proceedings of the 23rd International Symposium on Computer Architecture, 1996.

[Sen04] J. Seng, G. Hamerly. “Exploring Perceptron-Based Register Value Prediction,” Second Value Prediction and Value-Based Optimization Workshop, held in conjuction with ASPLOS, October, 2004.

[Sri01] S. Srinivasan, R. Ju, A. Lebeck, and C. Wilkerson. “Locality vs. Criticality.” 28th International Symposium on Computer Architecture, July, 2001.

[Tho04] A. Thomas and D. Kaeli, “Value Prediction With Perceptrons.” Second Value Prediction and Value-Based Optimization Workshop, held in conjuction with ASPLOS, October, 2004.

[Tho01] R. Thomas and M. Franklin, “Characterization of data value unpredictability to improve predictability.” International Conference on High Performance Computing, 2001.

[Tho01_2] R. Thomas and M. Franklin. “Using dataflow based context for accurate value prediction,” International Conference on Parallel Architectures and Compilation Techniques, 2001.

[Tom67] R. Tomasulo. “An Efficient Algorithm for Exploiting Multiple Arithmetic Units.” IBM Journal, 11(1):25-33, January, 1967.

[Tuc05] N. Tuck, D. Tullsen. “Multithreaded Value Prediction.” International Symposium on High-Performance Computer Architecture, February, 2005.

[Tul98] D. Tullsen, B. Calder. “Computing Along the Critical Path.” UCSD Technical Report, Oct. 1998.

[Tun01] E. Tune, D. Liang, D. Tullsen, B. Calder. “Dynamic Prediction of Critical Path Instructions.” 7th International Symposium on High Performance Computer Architecture, 2001.

[Tun02] E. Tune, D. Tullsen, B. Calder. “Quantifying Instruction Criticality.” International Conference on Parallel Architectures and Compilation Techniques, Sept. 2002.

[Vin99] L. Vintan “Towards a High Performance Neural Branch Predictor,” Proceedings of The International Joint Conference on Neural Networks, 1999.

[Wan97] K. Wang and M. Franklin. “Highly accurate data value prediction using hybrid predictors.” 30th Annual International Symposium on Microarchitecture, Dec. 1997.

[Yeh92] T. Yeh and Y. Patt. “Alternative Implementations of Two-Level Adaptive Branch Prediction.” 19th International Symposium on Computer Architecture, May 1992.

[Yeh93] T. Yeh and Y. Patt. “A Comparison of Dynamic Branch Predictors that use Two Levels of Branch History.” 20th International Symposium on Computer Architecture, May 1993.

[You95] C. Young, N. Gloy, and M. Smith. “A Comparative Analysis of Schemes for Correlated Branch Prediction.” Proceedings of the 22nd Annual International Symposium on Computer Architecture, June, 1995.

[Zho03] H. Zhou, J. Flanagan, and T. Conte. “Detecting Global Stride Locality in Value Streams.” International Symposium on Computer Architecture, 2003.