This is to certify that the

thesis entitled

ARTIFICIAL NEURAL NETWORKS
FOR BRANCH PREDICTION

presented by

Brian Adam Dazsi

has been accepted towards fulfillment
of the requirements for

Master's degree in Electrical Eng

Major professor

Date 8/9/01

ARTIFICIAL NEURAL NETWORKS FOR BRANCH PREDICTION

By

Brian Adam Dazsi

AN ABSTRACT OF A THESIS

Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

MASTER OF SCIENCE

Department of Electrical and Computer Engineering
2001

Dr. Richard Enbody

ABSTRACT

ARTIFICIAL NEURAL NETWORKS FOR BRANCH PREDICTION

By

Brian Adam Dazsi

Current microprocessor technology is being enhanced at an amazing rate. According to "Moore's Law", microprocessor transistor count and clock speed doubles every 18 months. With the speed that superscalar microprocessors can execute multiple instructions out-of-order, it is imperative that an extremely efficient branch predictor is implemented on the microprocessor, to keep the frequency of recovering from mispredicted instructions low. With current transistor counts in microprocessors being so high, more complex microprocessor components can now be considered, such as multiple pipelines, larger caches, and more complex branch predictors.

Artificial Neural Networks have recently been showing amazing usefulness in the areas of pattern recognition. They have also been used rather well for prediction applications. Through the use of C programming, the SPEC95 benchmarks, and a microprocessor simulator called SimpleScalar, this thesis explores the possibility of using artificial neural networks for branch prediction. The feed-forward, back-propagation artificial neural network coded for this thesis did not perform as well as expected; however, the area of Artificial Neural Networks is a rapidly growing field, and this thesis is just the beginning of the possibilities for using Artificial Neural Networks in branch prediction.

For my parents, who have helped me become the man I am today,

and my wife, who will continue life's journey with me.

ACKNOWLEDGEMENTS

There are many people throughout the years who have helped me get to where I am today. All of my professors and teachers over the years, and all of my family and friends have made this educational journey a bearable trip. To them I am eternally grateful. There are also a few special people who have helped me with this thesis - the toughest obstacle I had to overcome in all my education. I would like to thank them here.

My thanks go to Dr. Richard Enbody for his extreme patience. There were times that I doubted that I would get this thesis completed, but he never let me think that was an option. I would also like to thank him for his enthusiasm and teaching ability. His Advanced Computer Architecture class was my favorite graduate class - it was the most fun and most rewarding challenge. It instilled the desire to learn and explore further - hence this thesis.

Mark Brehob assisted me with using the SimpleScalar simulator. His few minutes of help here and there saved me from hours of wild goose chasing when I ran into problems that weren't really problems at all, or had simple solutions. His help is priceless.

I would like to thank Jackie Carlson for her endless support, encouragement and great C reference. Without her help I would not have had all the resources I needed to complete this project. Her time to sit and talk put my mind at ease and helped me get through the day-to-day struggles of this thesis and other things that distracted me from the thesis.

Dr. Fathi Salam introduced me to Artificial Neural Networks. His Neural Network classes were a wonderful learning experience, and helped me develop the idea for this thesis. For that I thank him.

My undying love and thanks goes to my wife, Sara, for putting up with me, and helping me through it all. I could not have done it without her support and love. Together we can accomplish anything.

I am thankful to my parents for absolutely everything. For obvious reasons, none of this would be possible without them; but they have also always been a guiding light for me. They have always being there for me - they have never let me down. They have always been tremendous role models. They have instilled in me the personal characteristics and responsibility that have guided me through my education and this thesis. Without those traits, I could never have overcome all of life's struggles to get this far.

I would like to thank God, for the opportunity to share my talents.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

INTRODUCTION

Background

The number of transistors in a microprocessor is growing at an enormous rate and how to use all those transistors is a topic of much debate. The number of, and specialization of, the execution units in the microprocessor pipeline are increasing. As a result, out-of-order instruction processing is standard practice. Since so many instructions are being fetched and executed out-of-order, when a branch instruction is encountered, it is important that the next instruction fetched is really the instruction that would be fetched if the correct answer to the branch decision was already known (at least two clock cycles are needed in order to process a branch instruction and calculate the branch decision). If the wrong path is chosen, then instructions start being executed that should not be executed and the microprocessor will have to recover from executing those invalid instructions. Therefore, it is imperative to have a good branch predictor.

Artificial Neural Networks are becoming more useful in the areas of pattern recognition and prediction. ANNs are starting to be used alongside standard statistical prediction models used for years in the fields of finance and marketing. The ANNs are performing as well as, if not better than, the statistical models [West]. Because of their success in these fields, an Artificial Neural Network might perform well as a microprocessor branch predictor.

Goals

When this thesis was started, there was no readily available documentation published about using a neural network for branch prediction. The goal of this thesis was to obtain a working simulation to compare a neural network branch predictor with current branch prediction technology. In order to achieve that goal, four tasks were established:

- Develop an Artificial Neural Network

Before an ANN could be used for branch prediction a set of tools and structures needed to be defined. The ANN was to be programmed in C, so that it could be incorporated into SimpleScalar.

- Modify the SimpleScalar simulator to accept an ANN branch predictor

To gather branch predictor performance data and add another predictor to the simulator, the coding for SimpleScalar needed to be explored.

- Train the Neural Network

Training data was to be gathered from a normal run of SimpleScalar and train the neural network. In order to obtain the working simulation in a reasonable amount of time, a feed-forward, back-propagation neural network using Hebbian learning would be used.

- Evaluate the ANN branch predictor

Finally, information about branch predictor performance was to be gathered by running SimpleScalar for each branch predictor. The branch predictor performance could then be plotted and examined.

Thesis Organization

The rest of the thesis is organized as follows. Chapter 1 discusses the background behind Branch Prediction. The Post-RISC architecture is discussed to show the importance of branch prediction in the microprocessor pipeline. Artificial Neural Networks are discussed in Chapter 2. A brief introduction of neural networks from a biological and mathematical perspective is given, and the Hebbian learning algorithm is talked about. Chapter 3 briefly discusses the SimpleScalar microprocessor simulator, its components and how it was utilized in this project. Chapter 4 discusses the methodologies used to add a new branch predictor to SimpleScalar and programming, verifying and training the neural network. Chapter 5 examines the results of the neural network training and the branch predictors' performance. Finally, the possibilities of future work using Artificial Neural Networks for Branch Prediction are examined in Chapter 6.

Chapter 1: Branch Prediction

1.1 Microprocessor Architecture

To understand the importance of branch prediction, an examination of the overall microprocessor must be done. Most microprocessors today are of a superscalar design, meaning that they execute multiple instructions at once. The Post-RISC architecture [Brehob] is worth a brief examination since most current microprocessors share much in common with this superscalar architecture [Hsieh]. Figure 1.1 [Brehob] shows the generic layout of a Post-RISC processor pipeline.

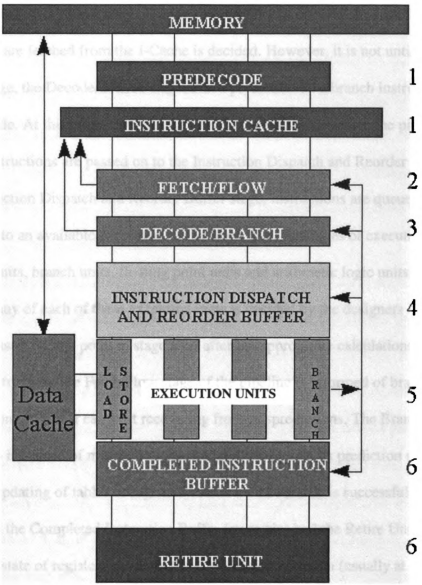


Figure 1.1: The Post-RISC Architecture

From examining Figure 1.1, there are six important steps in the processor pipeline. First, instructions are read from memory in the Predecode stage and stored into the Instruction Cache (I-Cache). In the superscalar architecture, multiple instructions are read into the cache at one time (for most current processors, four instructions are read [Hsieh]). As a part of predecoding, extra bits are appended to the instructions in order to

assist in decoding in a later stage. During the Fetch/Flow stage of the pipeline, which instructions are fetched from the I-Cache is decided. However, it is not until the third pipeline stage, the Decode/Branch stage, that a prediction on a branch instruction is actually made. At this point, "not taken" branches are discarded from the pipeline, and decoded instructions are passed on to the Instruction Dispatch and Reorder Buffer stage. In the Instruction Dispatch and Reorder Buffer stage, instructions are queued up and wait to move on to an available execution unit in stage five. Examples of execution units are load/store units, branch units, floating point units and arithmetic logic units. Which types and how many of each of these execution units is decided by the designers of the microprocessor. At this point in stage five, after the appropriate calculations are done for a branch instruction, the Fetch/Flow stage of the pipeline is informed of branch mispredictions so that it can start recovering from mispredictions. The Branch/Decode stage is also informed of mispredictions so that it can update its prediction methodology (typically, updating of tables or registers). After an instruction is successfully executed it is sent on to the Completed Instruction Buffer (stage six) and the Retire Unit successfully updates the state of registers based on the completed instruction (usually at a rate equal to that of Predecode stage instruction fetching). Instructions could be waiting in the Completed Instruction Buffer until information about a branch instruction becomes available so that they can be retired or erased. Figure 1.1 shows that branch prediction effects multiple stages of the pipeline.

1.2 Branch Prediction

An examination of the superscalar structure presented in Section 1.1 shows the importance of a good branch predictor. With multiple instructions being executed out-of-order in parallel, recovering from a misprediction can be extremely difficult and can cost valuable processor time and resources. Instructions that have been placed in the pipeline that should not have been, have to be nullified, and the correct instructions have to start being fetched. This recovery period is referred to as a stall.

In order to prevent misprediction stalls, a good branch predictor is needed. A few easy methods exist to provide branch prediction. First, a branch could always be assumed "taken" or always be assumed "not taken". In these static cases, the logic for the prediction is extremely simple, but also allows for an average 50% misprediction rate. This is just not acceptable. A more dynamic approach is to use a branch prediction buffer, which can be used to keep track of prediction history. In the simplest form, this history table is referenced by the lower portion of the branch instruction address and uses a 1-bit counter to record if the branch was recently "taken" or not. This simple method also does not provide for very accurate predictions [Hennessy]. However, a slight modification by adding another bit helps overcome the one-bit method's shortcoming.

1.2.1 Two-bit Predictor

In this method, two bits are used to keep track of the prediction history. Only after two consecutive mispredictions is the prediction state changed from *predict taken* to *predict not taken*. Figure 1.2 [Hennessy] shows how this is achieved in a state diagram.

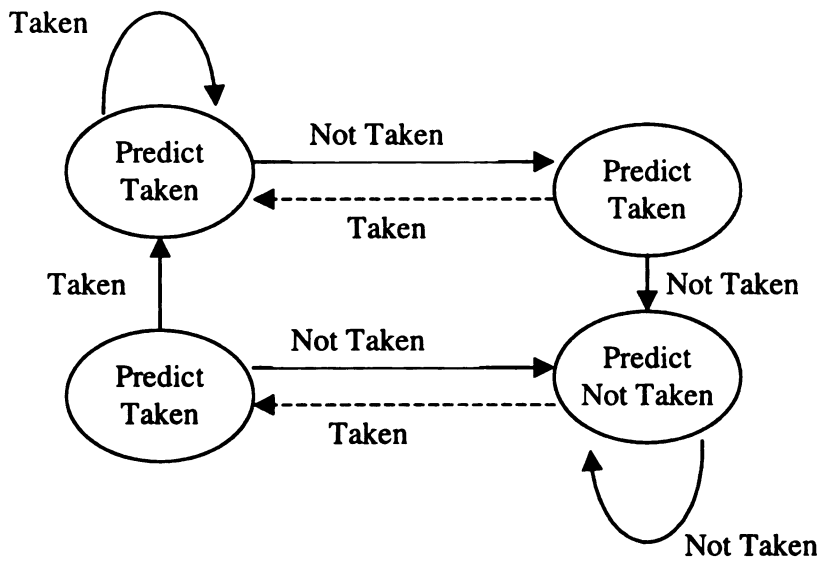


Figure 1.2: Two-bit prediction method states

While this method provides fairly good prediction accuracy, it is still not as ideal as current technology needs, nor as good as current technology can provide.

1.2.2 Two-level Adaptive Branch Prediction

This prediction method, introduced by Tse-Yu Yeh and Yale N. Patt, is one of the most successful prediction methods used today. It provides the most accurate predictions [Yeh93]. In the two-level prediction scheme, two levels of branch history are stored. In the first level, a record of the last *n* branches is stored in the Branch History Table (BHR). In the second level, the branch behavior of the last *x* occurrences of a specific pattern in the BHR is recorded. The second level table is called the Pattern History Table (PHT). Figure 1.3 [Driesen] shows the general layout of the two-level method.

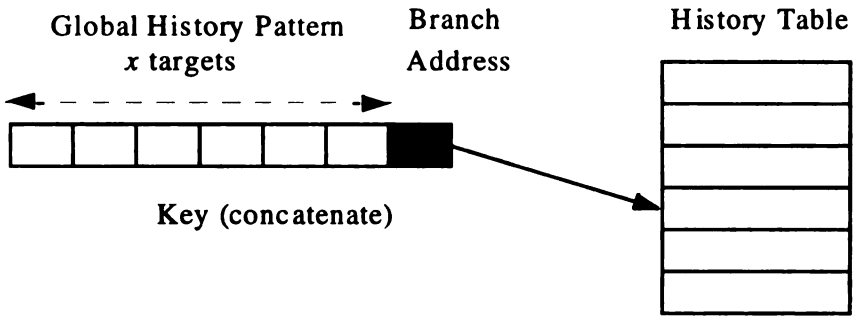


Figure 1.3: Generic Two-Level Branch Prediction

There are three manners in which information is kept in these tables: globally, per-address and per-subset (which was introduced by Pan, So and Rahmeh [Yeh93]). When referring to the Branch History Table, these methods are labeled as GA, PA, SA, respectively. When referring to the Pattern History Table, these methods are labeled as g, p, and s, respectively. This gives rise to nine different variations of the Two-Level Adaptive Branch Predictor, as shown in Table 1.1 [Yeh93].

Table 1.1: Two-Level Branch Predictor Variations

Variation	Description
GAg	Global Adaptive Branch Prediction using one global pattern history table.
GAp	Global Adaptive Branch Prediction using per-address pattern history tables.
GAs	Global Adaptive Branch Prediction using per-set pattern history tables.
PAg	Per-Address Adaptive Branch Prediction using one global pattern history table.
PAp	Per-Address Adaptive Branch Prediction using per-address pattern history tables.

Variation	Description
PAs	Per-Address Adaptive Branch Prediction using per-set pattern history tables
SAg	Per-set Adaptive Branch Prediction using one global pattern history table.
SAp	Per-set Adaptive Branch Prediction using per-address pattern history tables.
SAs	Per-set Adaptive Branch Prediction using per-set pattern history tables.

Figure 1.4, Figure 1.5, and Figure 1.6 show how the pattern history tables are referenced in the variations of each type of Two-level Adaptive Branch Prediction.

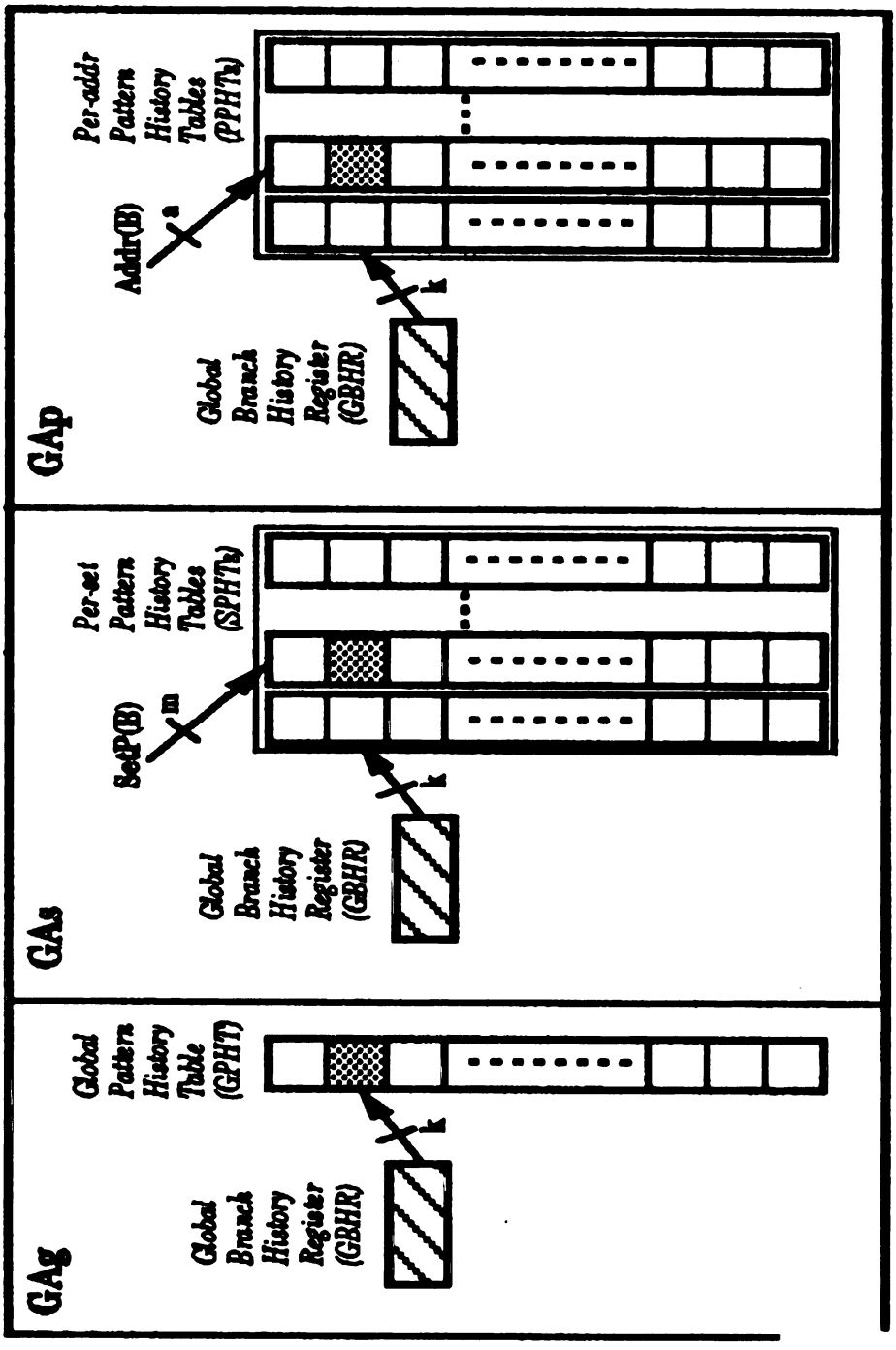


Figure 1.4: Global Adaptive Branch Prediction Methods

In the Global Adaptive Branch Prediction methods, the global history of the last k branches is recorded. Therefore, the history of all branches influences each prediction, since each prediction uses the same history register [Yeh93].

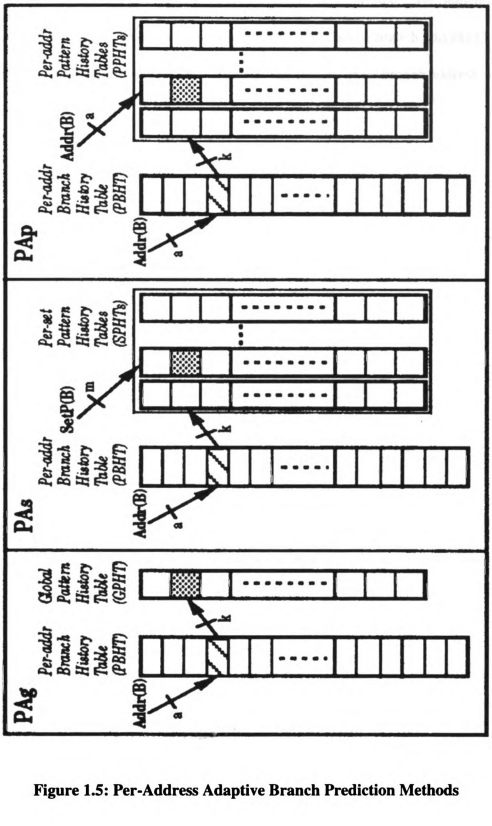


Figure 1.5: Per-Address Adaptive Branch Prediction Methods

In the Per-Address Adaptive Branch Prediction methods, the first-level branch history accesses the pattern history of the last k occurrences of the same branch address [Yeh93]. Each prediction is, therefore, only influenced by the history of its branch address, and not by other branches.

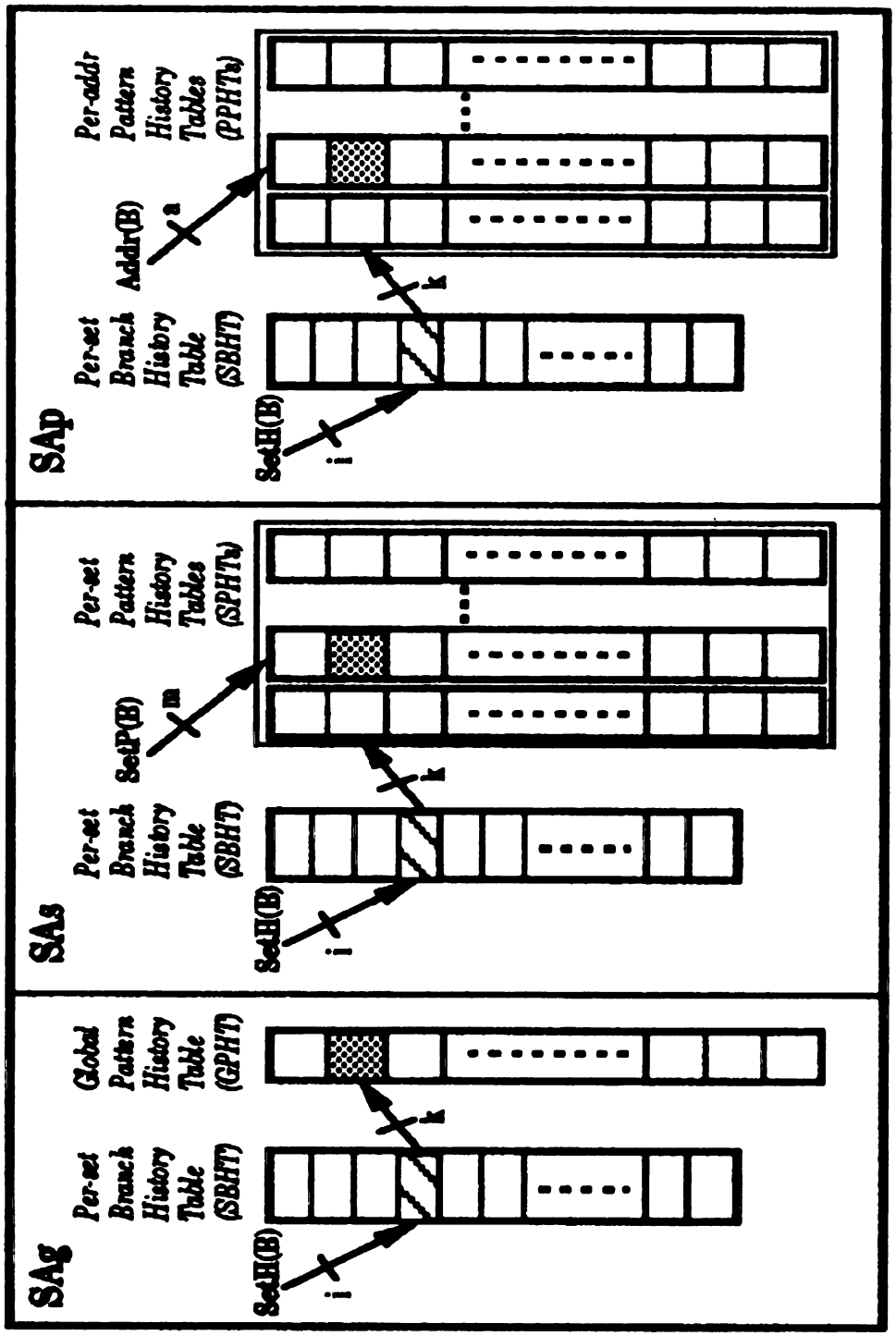


Figure 1.6: Per-Set Adaptive Branch Prediction Methods

For per-subset references, the set attribute can be any of the following: the branch opcode, the branch class (which is assigned by the compiler), or the branch address. In this way, the behavior of a branch effects other branches of the same type, or subset. The first-level of Per-set Adaptive Branch Prediction refers to the last *k* occurrences of a branch of the same set attribute. The second level can be accessed either globally, by a subset of a set, or by branch address [Yeh93].

1.2.3 The Branch-Target Buffer

To further improve prediction methods, the branch-target buffer (or cache) is introduced. In the BTB, the branch address and respective predicted address (or branch target) are stored. The branch address is used as a reference to the BTB to predict what the branch target should be. If the branch address is not in the table, then the branch is predicted "not taken". Using a BTB in combination with the Two-bit predictor increases prediction accuracy significantly [Driesen].

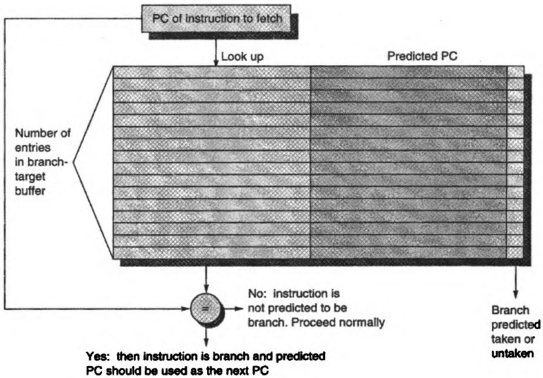


Figure 1.7: Branch Target Buffer

1.2.4 Hybrid or combinational predictors

Another solution when using more resources (transistors) on a microprocessor chip for branch prediction is to combine multiple dynamic branch predictors [Driesen]. When using two predictors at the same time a special Branch Predictor Selection Table, or metapredictor, must be used to choose which predictor to use for a given situation. This metapredictor is similar to the two-bit predictor discussed above. It uses a two-bit counter to keep track of which predictor is more accurate for a specific branch address.

There are a couple of variations of this metapredictor. One variation uses run-time or compile-time information to choose which predictor outcome to use. Another variation is

to use an n-bit confidence counter to keep track of the success rate of a predictor over the last 2^{n-1} prediction requests [Driesen].

1.3 Current Branch Prediction Methods

1.3.1 AMD K-6 2

AMD's K-6 2 microprocessor contains a 2-level branch predictor with an 8192 entry Branch History Table, a 16 entry BTC, and a 16 entry RAS.

1.3.2 Pentium III

The branch predictor methodology in Intel's Pentium III microprocessor is referred to as "deep" branch prediction. It contains a 512 entry Branch-Target Buffer.

1.3.3 Power PC 630

Motorola's Power PC 630 uses a Branch-Target Buffer and a two-bit predictor.

Chapter 2: Artificial Neural Networks

2.1 The Neuron

2.1.1 Biological Model

The basic model of the neuron is founded upon the functionality of a biological neuron. "Neurons are the basic signaling units of the nervous system" and "each neuron is a discrete cell whose several processes arise from its cell body" [Kandel].

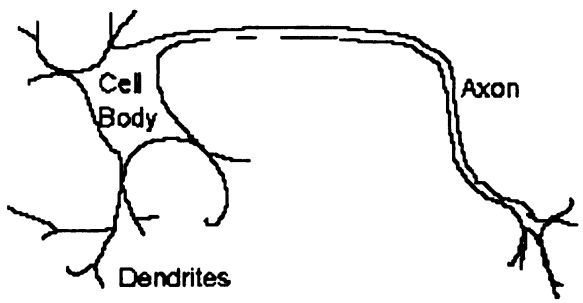


Figure 2.1: A Neuron

The neuron has four main regions to its structure. The cell body, or soma, has two offshoots from it, the dendrites, and the axon, which end in presynaptic terminals. The cell body is the heart of the cell, containing the nucleus and maintaining protein synthesis. A neuron may have many dendrites, which branch out in a treelike structure, and receive signals from other neurons. A neuron usually only has one axon which grows out from a part of the cell body called the axon hillock. The axon conducts electric signals generated at the axon hillock down its length. These electric signals are called

action potentials. The other end of the axon may split into several branches, which end in a presynaptic terminal.

Action potentials are the electric signals that neurons use to convey information to the brain. All these signals are identical. Therefore, the brain determines what type of information is being received based on the path that the signal took. The brain analyzes the patterns of signals being sent and from that information it can interpret the type of information being received.

Myelin is the fatty tissue that surrounds and insulates the axon. Often short axons do not need this insulation. There are uninsulated parts of the axon. These areas are called Nodes of Ranvier. At these nodes, the signal traveling down the axon is regenerated. This ensures that the signal traveling down the axon travels fast and remains constant (i.e. very short propagation delay and no weakening of the signal).

The synapse is the area of contact between two neurons. The neurons do not actually physically touch. They are separated by the synaptic cleft, and electric signals are sent through chemical interaction. The neuron sending the signal is called the presynaptic cell and the neuron receiving the signal is called the postsynaptic cell. The signals are generated by the membrane potential, which is based on the differences in concentration of sodium and potassium ions inside and outside the cell membrane.

Neurons can be classified by their number of processes (or appendages), or by their function. If they are classified by the number of processes, they fall into three categories. Unipolar neurons have a single process (dendrites and axon are located on the same stem), and are most common in invertebrates. In bipolar neurons, the dendrite and axon are the neuron's two separate processes. Bipolar neurons have a subclass called pseudo-

bipolar neurons, which are used to send sensory information to the spinal cord. Finally, multipolar neurons are most common in mammals. Examples of these neurons are spinal motor neurons, pyramidal cells and Purkinje cells (in the cerebellum).

If classified by function, neurons again fall into three separate categories. The first group is sensory, or afferent, neurons, which provide information for perception and motor coordination. The second group provides information (or instructions) to muscles and glands and is therefore called motor neurons. The last group, interneuronal, contains all other neurons and has two subclasses. One group called relay or projection interneurons have long axons and connect different parts of the brain. The other group called local interneurons are only used in local circuits.

2.1.2 Mathematical Model

When creating a functional model of the biological neuron, there are three basic components of importance. First, the synapses of the neuron are modeled as weights. The strength of the connection between an input and a neuron is noted by the value of the weight. Negative weight values reflect inhibitory connections, while positive values designate excitatory connections [Haykin]. The next two components model the actual activity within the neuron cell. An adder sums up all the inputs modified by their respective weights. This activity is referred to as linear combination. Finally, an activation function controls the amplitude of the output of the neuron. An acceptable range of output is usually between 0 and 1, or -1 and 1.

Mathematically, this process is described in Figure 2.2 [Haykin].

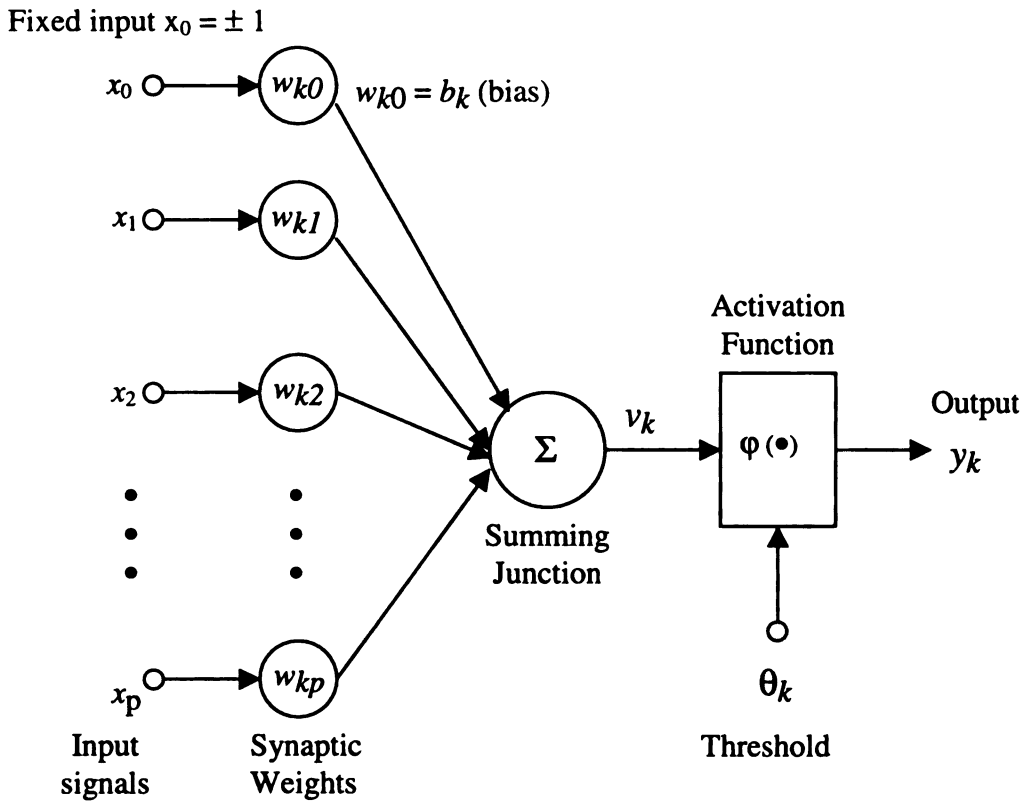


Figure 2.2: Model of a Neuron

From this model the interval activity of the neuron can be shown to be:

$$v_k = \sum_{j=1}^{p} w_{kj} x_j \tag{2.1}$$

The output of the neuron, y_k, would therefore be the outcome of some activation function on the value of v_k.

2.1.3 Activation functions

As mentioned previously, the activation function acts as a squashing function, such that the output of a neuron is between certain values (usually 0 and 1, or -1 and 1). In

general, there are three types of activation functions, denoted by $\varphi(\bullet)$ in Figure 2.2. First, there is the Threshold Function which takes on a value of 0 if the summed input is less than a certain threshold value (v), and the value 1 if the summed input is greater than or equal to the threshold value.

$$\varphi(v) = \begin{cases} 1 & \text{if } v \geq 0 \\ 0 & \text{if } v < 0 \end{cases} \tag{2.2}$$

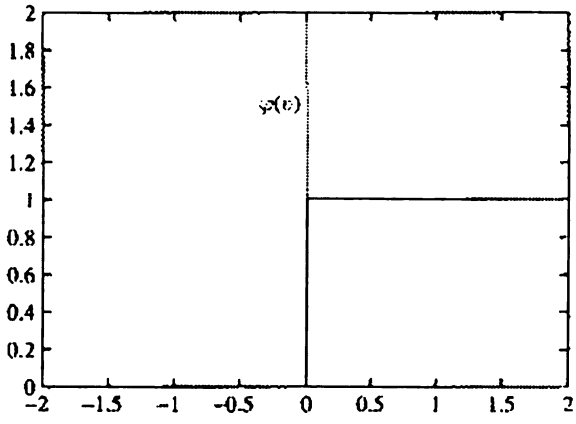
Secondly, there is the Piecewise-Linear function. This function again can take on the values of 0 or 1, but can also take on values between that depending on the amplification factor in a certain region of linear operation.

$$\varphi(v) = \begin{cases} 1 & v \geq \frac{1}{2} \\ v & -\frac{1}{2} > v > \frac{1}{2} \\ 0 & v \leq -\frac{1}{2} \end{cases} \tag{2.3}$$

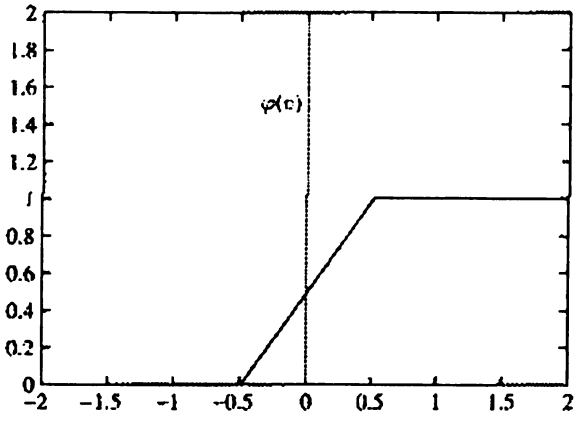
Thirdly, there is the sigmoid function. This function can range between 0 and 1, but it is also sometimes useful to use the -1 to 1 range. An example of the sigmoid function is the hyperbolic tangent function.

$$\varphi(v) = \tanh\left(\frac{v}{2}\right) = \frac{1-\exp(-v)}{1+\exp(-v)} \tag{2.4}$$

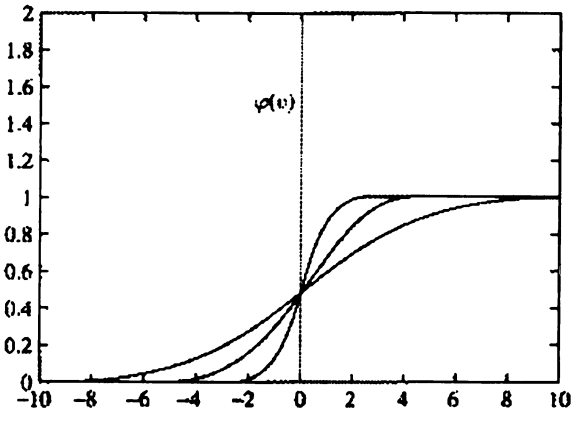
Figure 2.3 shows these activation functions plotted [Haykin].



theshold



piecewise-linear



sigmoid functions

Figure 2.3: Activation Functions

In Figure 2.2, two special values should be noted: the bias and the threshold. These are optional values that can be used, determined by the specific situation for which the neural network will be used. The bias can be eliminated from the network by making the bias weight, w_{k0}, equal to zero. The threshold value can also be set equal to zero to eliminate is effect on the network. For obvious reasons, if a Threshold Activation Function is not being used, the threshold value is not needed.

2.1.4 Directed Graphs

The neuron model described in Section 2.1.2 can be described in two other manners in order to simplify discussions. First, there is the signal-flow graph of a neuron, shown in Figure 2.4 [Haykin], where the symbols used are simplified.

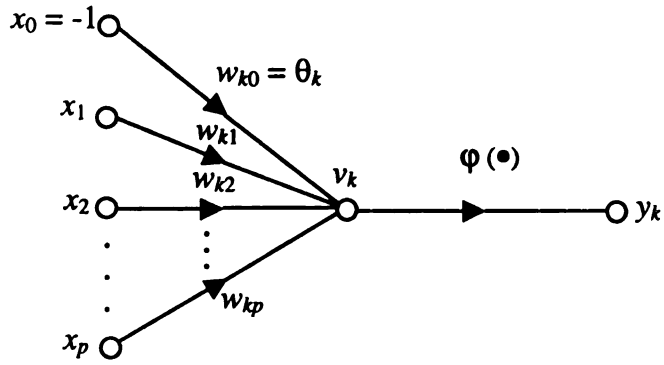


Figure 2.4: Signal-Flow Graph of a Neuron

An architectural graph can also be used to show the general layout of a neuron, as displayed in Figure 2.5 [Haykin]. This is useful when describing multilayer networks.

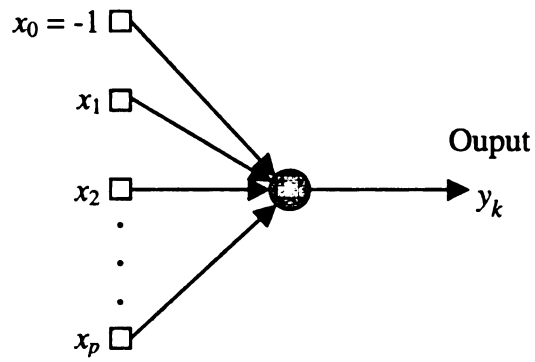


Figure 2.5: Architectural Graph of a Neuron

2.2 Learning

2.2.1 Description

One of the most important aspects of Artificial Neural Networks is the ability to learn and therefore update and improve performance. The definition of this neural network learning according to Haykin is:

> *Learning is a process by which the free parameters of a neural network are adapted through a continuing process of stimulation by the environment in which the network is embedded. The type of learning is determined by the manner in which the parameter changes take place.*

Haykin goes on to state three important events that occur during learning:

1. The neural network is stimulated by an environment.
2. The neural network undergoes changes as a result of this stimulation.
3. The neural network responds in a new way to the environment, because of the changes that have occurred in its internal structure.

Overall, the updating of the neural network is done by changing (or updating) the interconnected weights between neurons. Mathematically, this is described as follows:

$$w_{kj}(n+1) = w_{kj}(n) + \Delta w_{kj}(n) \tag{2.5}$$

How the update, or Δw, is calculated is determined by a set of rules applied with a certain paradigm. A general overview of the Artificial Neural Network learning process is shown in Figure 2.6 [Haykin].

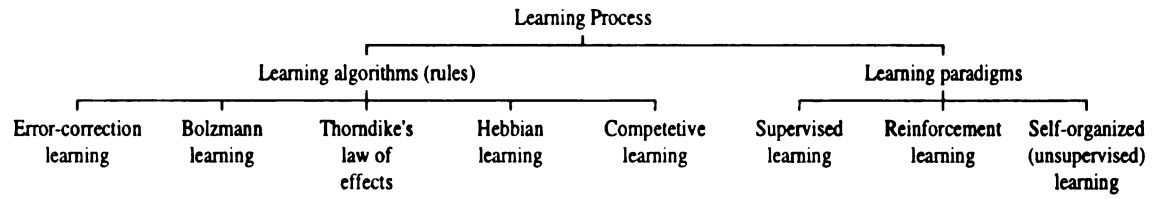


Figure 2.6: The Taxonomy of Learning

Learning algorithms are a set of rules by which the network is updated. The Learning paradigms are the styles or methodology in which the learning algorithm is performed. There are many different styles of learning, and new algorithms are continuously being explored and developed. For this thesis, Hebbian learning will be discussed.

2.2.2 Hebbian Learning

Hebbian learning is one of the oldest and most popular learning methods today [Haykin]. It derives its methodology from the idea that the more often a synapse between two neurons is fired, the stronger the connection between the two neurons becomes. Expanding that statement, it can be said that when the connection between two neurons is

excited simultaneously (or synchronously), the connection is strengthened, and when the connection is excited asynchronously, the connection is weakened [Haykin].

Mathematically, the change of weight in Hebbian learning is a function of both presynaptic (input) and postsynaptic (output) activities. In defining the weight update, a parameter called the learning rate is introduced. This learning rate, a positive value, defined the speed at which the weight changes. In order to keep the weight from growing too fast (which may lead to saturation), another parameter called the forgetting factor is introduced (this is also a positive value). Finally, a formula for the weight update for Hebbian learning can be defined as:

$$\Delta w_{kj}(n) = \eta y_k(n) x_j(n) - \alpha y_k(n) w_{kj}(n) \tag{2.6}$$

2.3 The Perceptron and Multilayer Perceptrons

The perceptron is the simplest form of a neural network used for the classification of linearly separable patterns [Haykin]. Its structure can be easily shown using the signal-flow graph described above, since a single layer perceptron operates as a single neuron. Figure 2.7 shows this [Haykin].

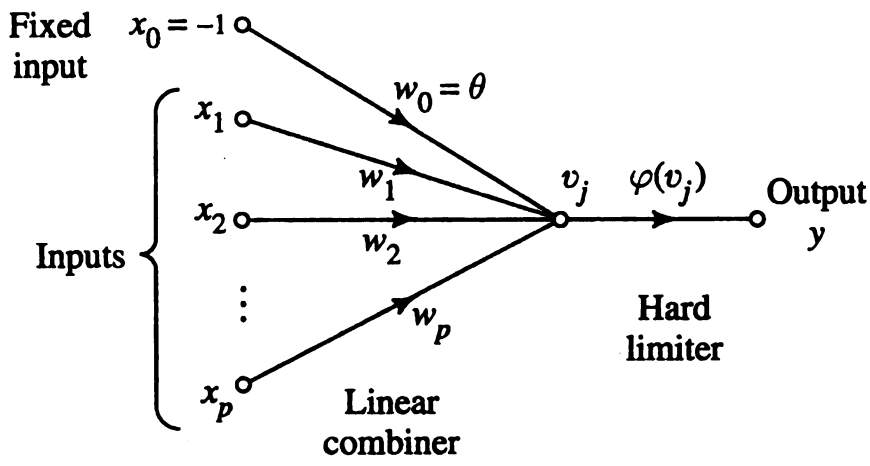


Figure 2.7: A Single Layer Perceptron

Multilayer perceptrons are an expansion of the perceptron idea, and can be used to solve much more difficult problems. They consist of an input layer, one or more hidden layers and an output layer. The hidden layers give the network its power and allow for it to extract extra features from the input. Figure 2.8 shows a multilayer perceptron with two hidden layers [Haykin].

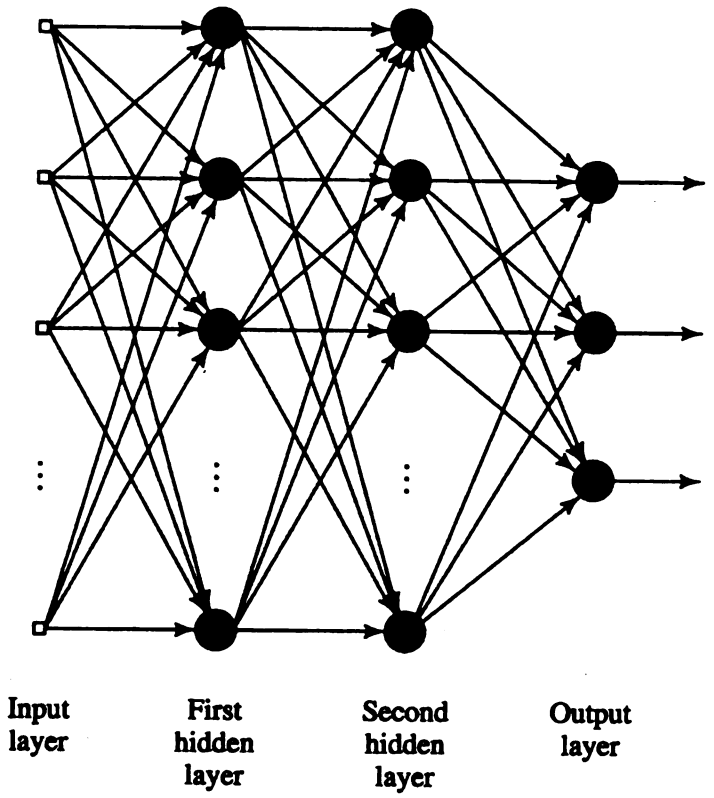


Figure 2.8: Multilayer Perceptron

One of the most popular methods used in training a multilayer perceptron is the error back-propagation method, which includes two passes through each layer of the network: a forward pass and a backward pass [Haykin].

2.4 Feedforward and Backpropagation

In error back-propagation, there are two types of signal flow. First, signals are calculated in a forward manner, from left to right, through each layer. These are called

function signals. Second, the error at each neuron is calculated, from right to left, in a backward manner through each layer. These are called error signals.

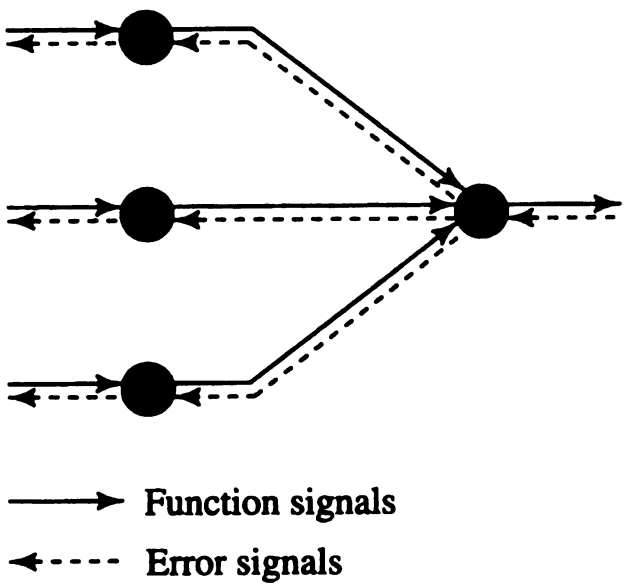


Figure 2.9: Signals in a Multilayer Perceptron

The goal of the error back-propagation method is to make the average squared error of the entire network as small as possible. Once the function signals have been calculated, the error back be back-propagated though the network. The error is calculated using the local gradient, $\delta_{j(n)}$[Haykin]. At the error of the output layer, the local gradient is just defined as:

$$\delta_j(n) = e_j(n)\varphi_j'(v_j(n)) \tag{2.7}$$

When calculating the error for a neuron in the hidden layer, the local gradient is defined as:

$$\delta_j(n) = \varphi_j'(v_j(n))\sum_k \delta_k(n)w_{kj}(n) \tag{2.8}$$

The weight correction, $\Delta w_{ji(n)}$, is defined as the learning rate parameter (η) times the local gradient (δ) times the input signal of the neuron (j):

$$\Delta w_{ji}(n) = \eta \cdot \delta_j(n) \cdot y_i(n) \tag{2.9}$$

A momentum term is also used in calculating the weight update.

$$\Delta w_{ji}(n) = \alpha \Delta w_{ji}(n-1) + \eta \delta_j(n) y_i(n) \tag{2.10}$$

Chapter 3: SimpleScalar

3.1 SimpleScalar

3.1.1 Software Architecture

SimpleScalar has a modular layout. This layout allows for great versatility. Components can be added or modified easily. Figure 3.1 shows the software structure for SimpleScalar. Most of the performance core are optional modules [Austin].

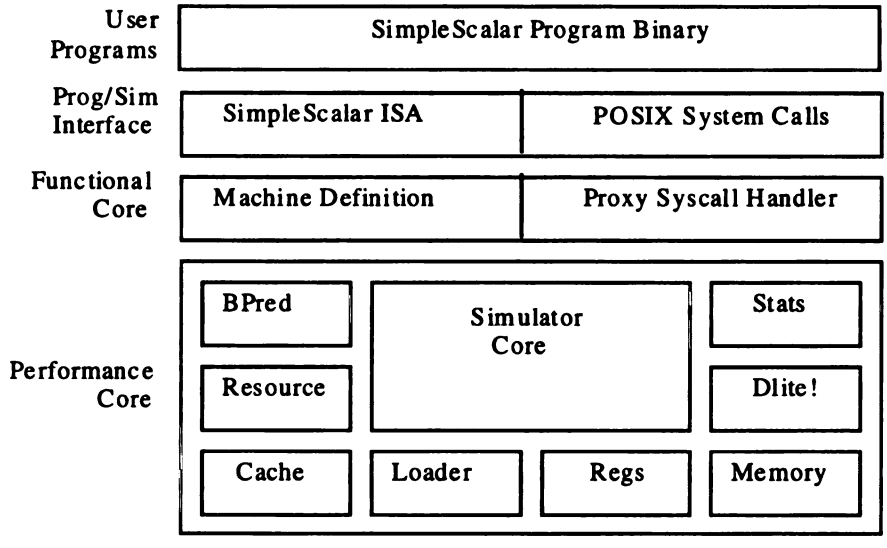


Figure 3.1: SimpleScalar Software Architecture

3.1.2 Hardware Architecture

The Hardware architecture of the SimpleScalar simulator closely follows the Post-RISC architecture previously described. Figure 3.2 shows the out-of-order pipeline for the SimpleScalar hardware architecture.

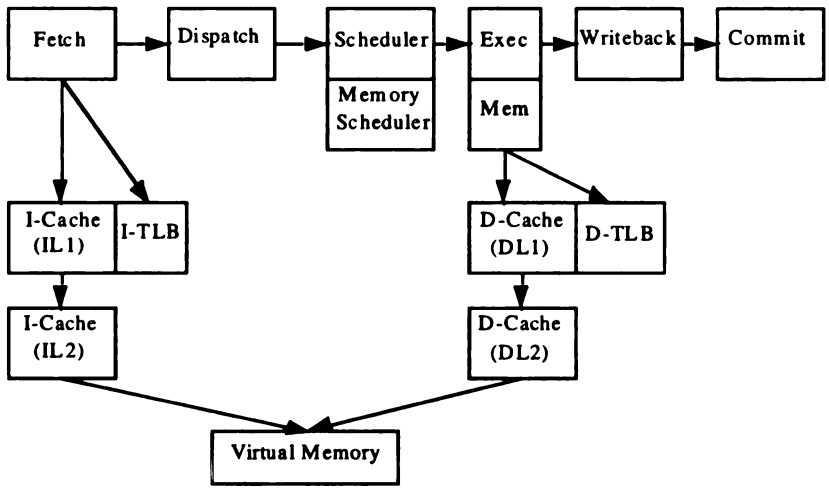


Figure 3.2: Out-of-Order Issue Architecture

3.1.3 Instruction Set Architecture

The instruction set architecture is based on MIPS and DLX [Patterson] instruction set architectures, with some additional addressing modes [Austin]. It is a "big-endian" instruction set architecture definition, which allows for easier porting of the simulator to new hosts since the simulator is compiled to match the host's endian. The instructions are 64-bit. Only 48 bits are currently used; the extra 16 bits allow for expansion or research of instructions.

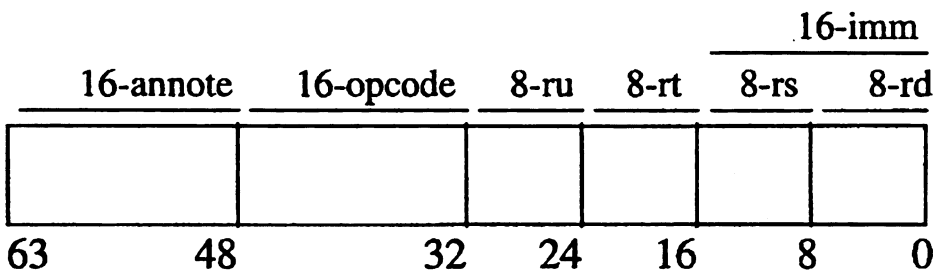


Figure 3.3: Instruction Format

3.1.4 Running SimpleScalar

For this thesis, the Out-of-Order execution simulator will be used. The correct command-line format for using the out-of-order simulator is as follows: sim-outorder <options> <sim_binary>. The <options> include the branch prediction configuration.

3.1.5 Branch Prediction

SimpleScalar supports a variety of branch prediction methods. It allows for the static always "taken" and always "not taken" prediction methods. It also has options for the Two-Level Adaptive Branch Predictors. It supports a Two-bit Predictor referred to as a bimodal predictor. SimpleScalar even supports a Hybrid predictor, a combination of the two-level and bimodal predictors, using a metapredictor to choose between the two predictors. SimpleScalar also utilizes a branch address predictor through Branch Target Buffer.

The command line option for specifying the branch predictor is -bpred <type>. The types are as described in Table 3.1.

Table 3.1: Branch Prediction Types

predictor option	Description
nottaken	always predict not taken
taken	always predict taken
perfect	perfect predictor
bimod	bimodal predictor: BTB with 2-bit counters
2lev	two-level adaptive predictor
comb	hybrid predictor: combination of 2level and bimodal

Figure 3.4 shows the Two-level adaptive branch predictor layout.

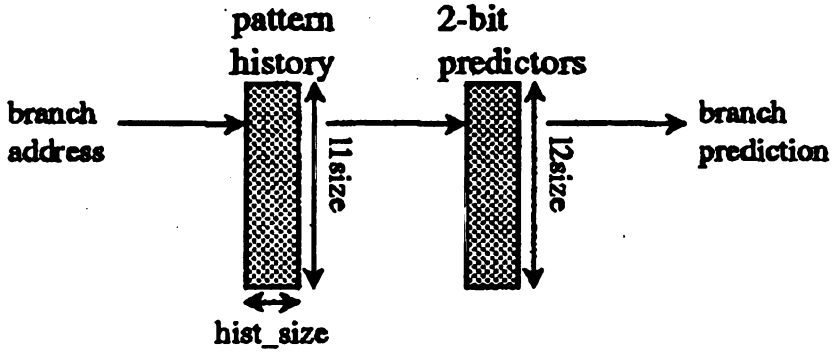


Figure 3.4: Two-level Predictor Layout

The configuration options for each predictor are described in Table 3.2.

Table 3.2: Branch Prediction Options

configuration parameters	Description
-bpred:bimod <size>	the size of the direct mapped BTB
-bpred:2lev <l1size> <l2size> <hist_size>	l1size - the size of the first level table (number of shift registers) l2size - the size of the second level table (number of counters) hist_size - the history pattern width (width of shift registers)
-bpred:comb <size>	The number of entries in the metapredictor table

Table 3.3shows how specific 2-level design options may be implemented in the SimpleScalar simulator. Note that each counter has one BTB entry.

Table 3.3: Two-Level Branch Prediction

type	l1size	l2size	hist_size
counter based	1	m	0
GAg	1	2^w	w
GAp	1	m(m>2^w)	w
PAg	n	2^w	w
PAp	n	m(m == 2^(N+W))	w

When selecting the hybrid predictor, both the bimod and 2lev configuration options may be specified.

3.2 Using the SPEC95 benchmarks

Available with the SimpleScalar simulator are SimpleScalar binaries of the SPEC95 benchmark suite. These binaries are useless without the input files available only by

purchasing the license to use the benchmarks. In conjunction with a PERL script, Run.pl, written by Dirk Grunwald and Artur Klauser, the SPEC95 benchmarks can be run on the SimpleScalar simulator. This thesis uses the following SPECint95 benchmarks to test the branch predictors: 099.go, 126.gcc, 130.li, 134.perl; and the following SPECfp95 benchmarks: 101.tomcatv, 107.mgrid, 145.fpppp. The correct command-line implementation used by Run.pl for running the SPEC95 benchmarks on the SimpleScalar architecture is as follows: sim-outorder <sim_options> <benchmark_binary> <benchmark_options> < <input_file> > <output_file>.

Chapter 4: Methodology

4.1 Programming the Neural Network

To more easily integrate the ANN code into SimpleScalar, the neural network was programmed in C. Also, the error subroutines used in the SimpleScalar package were used in programming the neural network (specifically the *fatal* subroutine for run-time error messages, which are included in the misc.h and misc.c files).

The two files that define the feed-forward back-propagation artificial neural network code are a header file and a C code file, ffbpann.h and ffbpann.c. Three structures were created to define the overall design of the FFBPANN: bounds, layer and ffbpann. Bounds is a simple template of two integers that define the minimum and maximum number that will be represented as input to the FFBPANN. These values were used to normalize the inputs. Layer is a more complicated structure that is used in defining the overall FFBPANN structure. A layer contains a variable of the number of nodes in the layer, an array for the error calculations of each node, an array for the output calculations of each node, a multi-dimensional array for the weights, a multi-dimensional array for saveweights (used during training of the FFBPANN), and a multi-dimensional array of deltas (used during back-propagation). The FFBPANN structure itself possesses four variables: the number of layers in the FFBPANN, the momentum, learning rate and sigmoid function gain (all used for training of the FFBPANN), and the net error calculated for the ANN. It also has a pointer to the array of input bounds, a pointer to the

input layer, a pointer to the output layer, and a pointer to an array of layers (described above).

Several functions were defined so that the FFBPANN would behave properly. Create_ffbpann allocates memory for the entire FFBPANN, and ffbpann_free frees up the memory allocated for the FFBPANN. Init_ffbpann initializes the FFBPANN weights either to random numbers, or predefined weights (pre-trained values saved in a text file). Nnet_random is used to calculate the random values used when initializing a weight matrix. Calc_ffbpann calculates the output of the FFBPANN given the current weight matrices. Backprop performs the back-propagation algorithm of the FFBPANN, and is usually followed by the updateweights function, which calculates the new weight values, based on the recent back-propagation. Restore_weights and save_weights are used during training to either revert to the last saved weight matrices or save the currently calculated matrices, depending on how successful the current epoch of training was. Dump_weights is the function used to write the current weight matrix to a file. The format of this file is as follows: the bounds for each input node are at the beginning of the file (one pair per line); each subsequent line contains a weight value starting with the first node of the first layer, then the second node of the first layer, and continuing until the last node of the last (output) layer. Train_ffbpann and learn_ffbpann are short sample functions for setting up training and testing routines.

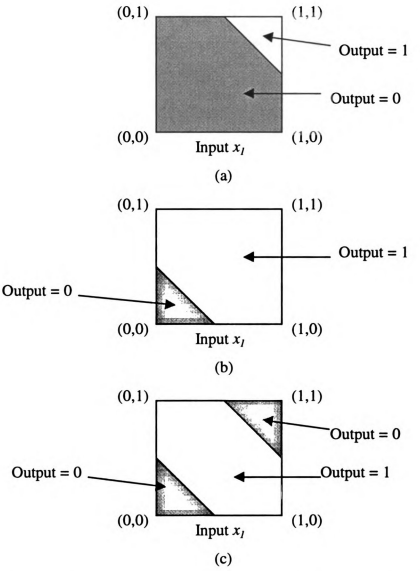
To accommodate testing and training, small C programs were used as "wrappers" to the four main files used for the FFBPANN. The layer and node parameters are defined in the "wrapper" program so that various FFBPANN configurations could be tested.

Examples of these programs programs can be found in Appendix B of technical report MSU-CSE-01-22, and are discussed below.

4.2 Verifying the FFBPANN Code

4.2.1 The XOR Problem

To verify that the FFBPANN code was working properly the standard XOR (exclusive OR) test was used. The XOR test is a very specific application of a common problem of a single-layer perceptron. A single-layer perceptron is incapable of classifying input patterns that are not linearly separable, as is the case in the XOR problem. To overcome this limitation, a multi-layer neural network is used to classify the XOR inputs as shown in Figure 4.1 and Figure 4.2 [Haykin].



(a) Decision boundary constructed by hidden neuron 1 of the network in Figure 1.2.
(b) Decision boundary constructed by hidden neuron 2 of the network.
(c) Decision boundaries constructed by the complete network.

Figure 4.1: XOR Classifications

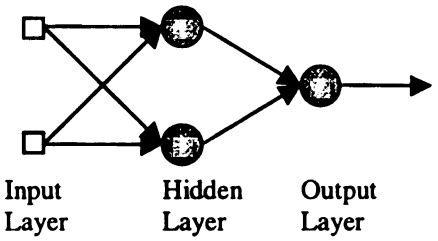


Figure 4.2: Multi-layer network for solving the XOR Problem

A 2x1 FFBPANN was trained using the very small data set of the XOR function. The weights were saved in a file and another short program was used to verify that the saved weights were correct to solve the XOR problem. The training and test files programs can be found in Appendix B of technical report MSU-CSE-01-22. The output from the testing file is as follows:

```
inputs: 0.000000 0.000000   output: 0.000068
inputs: 0.000000 1.000000   output: 0.999886
inputs: 1.000000 0.000000   output: 0.999923
inputs: 1.000000 1.000000   output: 0.000093
```

4.2.2 Predicting Sunspots

A second test was used to verify that the FFBPANN used for this experiment could indeed do prediction. A data set of sunspot levels was discovered [Kutza] and was used to train and test a 10x1 FFBPANN. The inputs to the FFPBANN were the sunspot levels from the previous 10 years, and the output is the prediction of the sunspot level for the current year. The code used follows in programs can be found in Appendix B of technical report MSU-CSE-01-22. The output of the test program follows in Table 4.1.

Table 4.1: Sunspots Test Program Output

Year	Sunspots	Prediction
1960	0.587	0.547
1961	0.282	0.306
1962	0.196	0.048
1963	0.146	0.082
1964	0.053	0.183
1965	0.079	0.153
1966	0.246	0.174
1967	0.491	0.454
1968	0.554	0.620
1969	0.552	0.533
1970	0.546	0.431
1971	0.348	0.473
1972	0.360	0.157
1973	0.199	0.209
1974	0.180	0.121
1975	0.081	0.099
1976	0.066	0.047
1977	0.143	0.076
1978	0.484	0.313
1979	0.813	0.615

4.3 Adding a new predictor to SimpleScalar

To add a new predictor to SimpleScalar the following files in the package had to be modified: bpred.c, bpred.h, sim-outorder.c, sim-bpred.c and the Makefile. The Makefile was modified so that the extra FFBPANN code was included during compilation. The modifications to sim-outorder.c and sim-bpred.c accomplished three tasks. Firstly, so that additional information (variables) would be sent to the branch predictor functions specifically for use by the FFBPANN predictor; secondly, to define the FFBPANN predictor command line options and how the predictor is called; and lastly, to display the

usage information for the command line options. The majority of modifications were made to the bpred.h and bpred.c files. The predictor structure was defined in bpred.h, and the predictor functions were coded in detail in bpred.c. For more detailed information, see Appendix C of technical report MSU-CSE-01-22 for the code listing.

The procedure for adding the new predictor was to add another hybrid predictor to the code so that there existed two hybrid predictors, but the second one possessed a different name (the FFPBANN predictor). Then, slowly, code for the 2-level component of the new hybrid predictor was modified to accommodate the FFPBANN predictor code. See the code listing in Appendix C of technical report MSU-CSE-01-22 for more detailed information, and Section 4.4.1 for a discussion of the predictor design.

4.4 The new predictor

4.4.1 Design

The overall structure of the new predictor (the FFBPANN predictor) is similar to the Combination (or Hybrid) predictor. In the FFBPANN predictor, a two-bit predictor and a purely artificial neural network predictor are used in combination, with a meta predictor used to choose between the outputs of the two predictors.

The purely artificial neural network predictor is the feed-forward back-propagating artificial neural network described in Section 4.1. There are four inputs to the network and one output. The inputs are the following variables in the SimpleScalar program: the branch address, the opcode portion of the instruction, and the RS and RT registers. The single output is the branch target ("taken" or "not taken"). The number of neurons in each

layer can be configured at runtime, and there may be from 1 to 4 hidden layers (the input and output layers are, of course, required). The only prerequisite for selecting a certain configuration at runtime is that a weights file must exist for the ANN to function. The format for the file name of the weights file is comprised of the layer configuration followed by the string ".weights.dump". For example, a 4x3x2x1 FFBPANN would have a weights matrix file named `4x3x2x1.weights.dump`. This weights file can be a set of randomly generated numbers or a set of trained data. It is recommended that a set of trained data be used. This file can be created using the dump_weights function. The FFBPANN will adjust, or update, its weights as SimpleScalar runs, and therefore improve its predictions. Figure 4.3 shows a simple network diagram of the ANN used in this scenario.

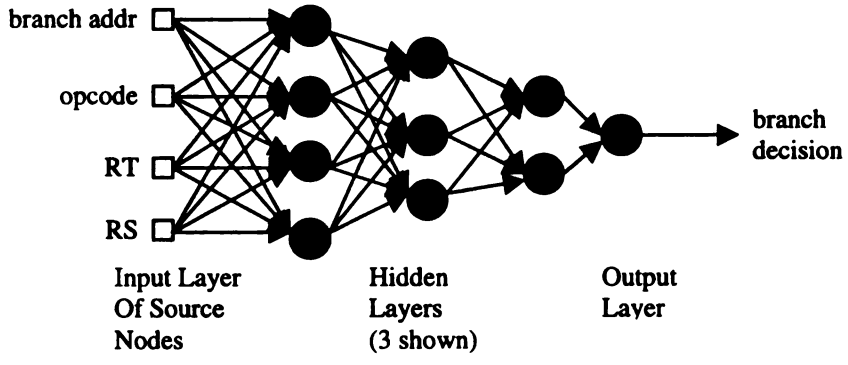


Figure 4.3: FFBPANN Structure

4.4.2 Training the FFBAPNN

To obtain a data set to train the FFBPANN for branch prediction, the SimpleScalar code was modified to output branch information to a file while it ran a simulation. The

branch information consists of data available to the branch predictor function that could be used as input for the FFBPANN (which was the aforementioned branch address, the opcode portion of the instruction, and the RS and RT registers), and the correct prediction.

Wrapper programs, discussed in Section 4.1, were written to train and test a weight matrix using this branch information. The training data was divided into three portions for training and testing. The first 60% of the data was used for training, the next 30% was used for testing the progress of the training routines (used to calculate the stopping criteria for training), and the remaining 10% was reserved exclusively for testing a trained FFBPANN.

Several different techniques were used for training. This was done to find the quickest and most efficient way to train the neural network. This was also done because, early on, the training was not producing a quality branch predictor. Therefore, modifications to the training data set, varying the training parameters, and modifying the ANN structure were tried in the hopes of improving the training procedure. The following procedures were taken to vary the attempts of finding a weight matrix that would classify the inputs.

- The whole of the raw training data set was used.
- Modifications to the raw data set were made to reduce the size of the data set. The most common entries in the data set – lines that appeared more than 10 times – were extracted. This was attempted again for lines that appeared only more that 50 times. This left only single entries in the resulting data set for the most common entries.

- Another attempt to reduce the data set was to pull the same frequently occurring entries, described above, out of the raw data set; however, this time the total number of occurrences was reduced by a fraction instead of being reduced to one entry. Now an entry occurring 10000 times in the raw data set would only occur 10 times in the reduced data set.
- The momentum (alpha) and the learning rate (eta) parameters of the FFBPANN were varied. These values changed only slightly, as the values initially chosen were suitable for most cases.
- Different layer structures were tested. The following four FFBPANN configurations were tried: 4x3x2x1, 4x4x3x1, 4x4x2x1, 4x3x3x1. Though, the 4x3x2x1 structure was the one most commonly used for training and testing.

Chapter 5: Results and Discussion

5.1 Training

The training data was gathered, as described in the previous chapter, by running a version of SimpleScalar modified to output the branch information (inputs and output) to a file. Several experiments to train a FFBPANN branch predictor were done using this data by varying the training slightly with each training attempt. None of the training techniques used produced a neural network branch predictor that performed more effectively than the other branch predictors already used in the SimpleScalar simulator. The results of these training attempts are as follows.

The modifications to the raw data sets did not produce better results than the attempts on the raw data sets described below. The attempt to reduce the data set by extracting only unique lines from the data would not help since frequently occurring data (i.e. a pattern for the ANN to pick out) would not be represented. The other reductions of the data set just pulled out infrequently occurring lines, but did not help in the training attempts either. The reason for reducing the size of the data sets was to speed up training so that the following two modification to the training procedures could be tried numerous times.

The attempts to vary the momentum (alpha) and the learning rate (eta) of the FFBPANN also did not produce satisfactory results. Again, these values only varied slightly, since the standard momentum and learning rate were used. Changing these parameters should help speed up the training procedure (or give it a little boost). However, since the training was performing poorly, all that these modifications did was

assist the training perform poorly faster. However, changing these values had little or no effect on the outcome, and at worst would cause the training to finish quicker, but with poorly calculated weights. The testing of these weights never produced satisfactory predictions.

Surprisingly, changing the layer structure of the FFBPANN had no effect on the training of the FFBPANN. With the reduced data sets, multiple layer changes could be tested more frequently. Since training with the full raw data sets would take so long, the only way to test multiple later options was to use the reduced data sets. None of the four FFBPANN architectures tested (4x3x2x1, 4x4x3x1, 4x4x2x1, 4x3x3x1) produced improved training performance or more accurate prediction results.

5.2 Discussion

One oddity to note occurred while gathering the training data. The SimpleScalar code was modified to write the selected inputs for the FFBPANN to a file. However, the branch data for every branch was not captured due to a file size limitation of Solaris 2.6. The maximum file size turned out to be about 2 Gigabytes (2^{31} bytes). When this maximum was reached, subsequent branch prediction information was no longer gathered. The gcc SPEC95 benchmark was the only benchmark for which the full branch prediction data was gathered. It is significant to note the importance of gathering the gcc branch information, since gcc performance is used by the industry to gauge its own improvements to such components as compilers. The limitation on the amount of the branch information that could be gathered could be a contributing factor for the poor

FFBPANN training. Table 5.1 shows the amount of branch prediction information that could be gathered.

Table 5.1: Limitations of Branches Recorded

	fpppp	gcc	fpppp and gcc	mgrid	go
total branches	2774271287	50378431	2824649718	1444135667	2423767703
precent of total branches recorded	4.63%	100.00%	6.33%	9.28%	5.38%
total branches recorded	128405869	50378431	178784300	134019707	130442404
unique entries	127421801	49530881	176952682	16040941	128871783
precent of unique entries	99.23%	98.32%	98.98%	11.97%	98.80%
unique inputs	126942800	49298484	176241284	13146619	128249968

Another possible reason for non-convergence of the FFBPANN weights is that the training data is too chaotic. An ANN is well suited to pick out patterns from the input. In this situation, no unique patterns exist for the FFBPANN to effectively classify. Even though the FFBPANN is much more suited to this task than the human eye given the amount of inputs examined, using the four inputs available to the predictor the FFBPANN was still unable to recognize any consistency in the data. Table 5.1 shows that almost every branch examined for the gcc, fpppp, and go benchmarks had a unique pattern – all four inputs and the branch outcome were different. However, this does not necessarily prove that the data is too chaotic for the FFBPANN weights to converge, since when examining the case of the mgrid benchmark, only about 12% of the branches were unique. However, during training with the full raw data set from the mgrid benchmark, something very interesting was noticed.

A closer look at the training progress using the mgrid data set shows that the FFBPANN was behaving almost exclusively as a "branch-always-taken" predictor. This is shown in Figure 5.1. While the output never fully reached the value of "1," it was extremely close at a maximum value of 0.969877.

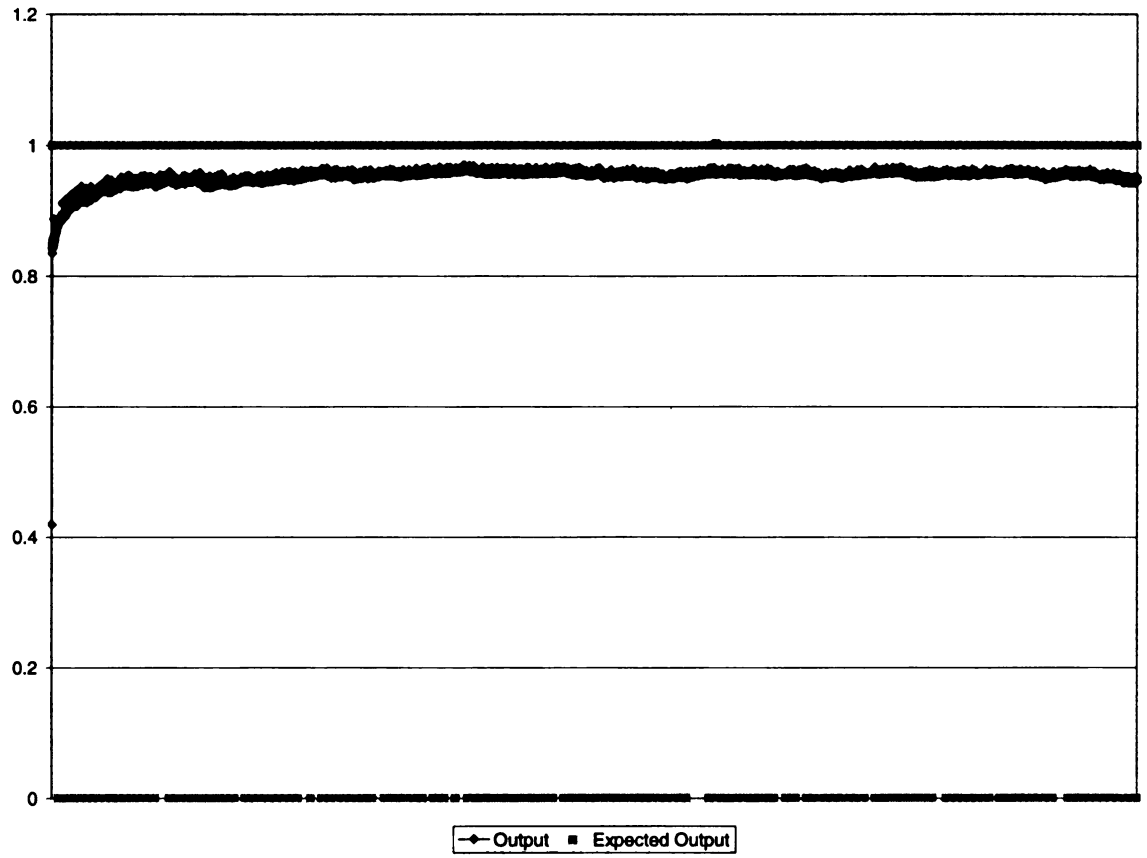


Figure 5.1: Mgrid Training Progress

Figure 5.2: Mgrid Training Error

Another item to note about the mgrid training data is that the majority of the branches were "taken". The training error shown in Figure 5.2 more clearly shows how often the branch should have been "not taken". About 95% (127569741 of 134019707 recorded branches) of the mgrid branch decisions were "taken". This could definitely account for why the FFBPANN appeared to be acting as a "branch-always-taken" predictor. Taking a closer look at the training output shows how quickly the ANN starts behaving this way. Figure 5.3 zooms in on the beginning of the mgrid training progress.

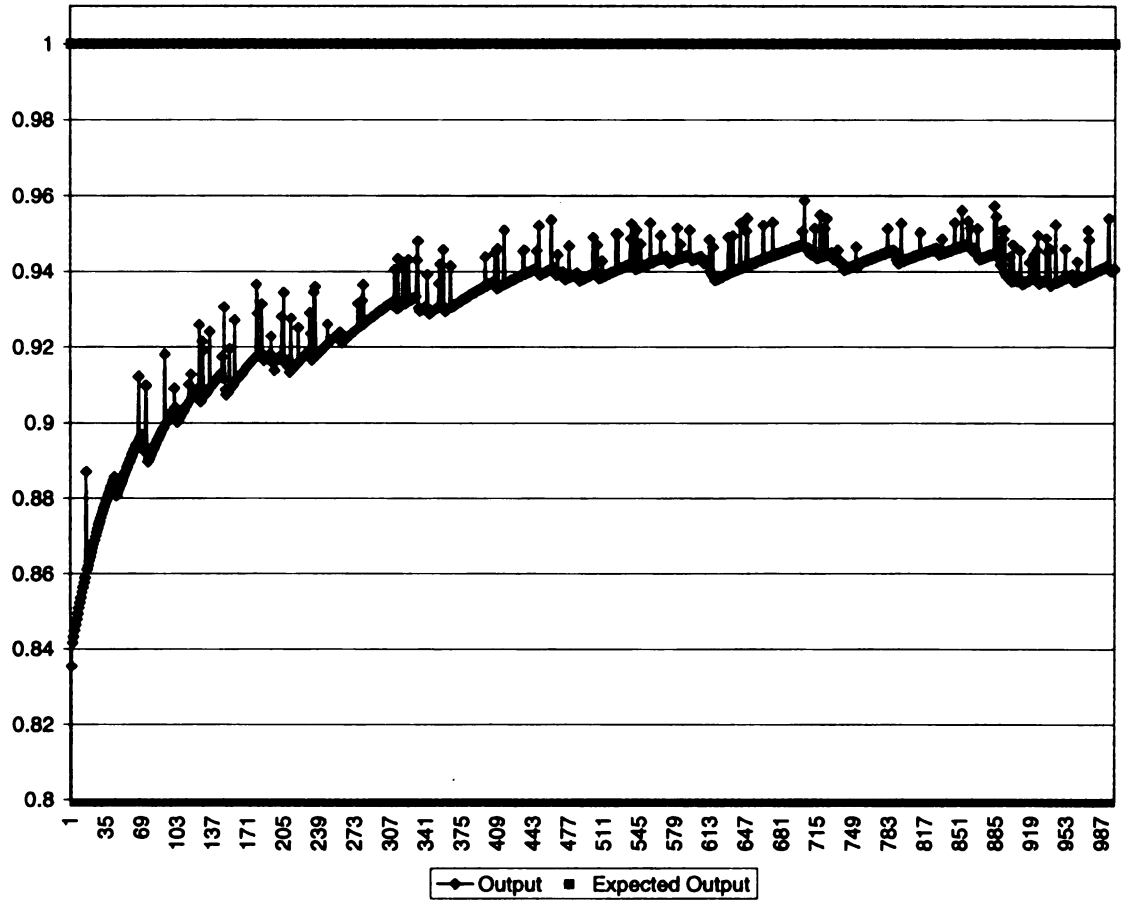


Figure 5.3: Mgrid Training Progress Zoom

Table 5.2 examines how frequently the other SPEC95 benchmarks' branches were "taken" or "not taken".

Table 5.2: Branch Data

Branches	**Gcc**	**fpppp**	**mgrid**	**go**
"taken"	30558126	77594080	127569741	84216932

Branches	Gcc	fpppp	mgrid	go
"not taken"	19820305	50662550	6449966	46225472
Total	50378431	128256630	134019707	130442404
Percent "taken"	60.66%	60.50%	95.19%	64.56%

Interestingly enough, while training using the gcc and go benchmark data, which have a more even distribution of "taken" and "not taken" branches (about 61% and 65%, respectively), the same training error occurred. The ANNs produced by those training attempts also performed very closely to a "branch-always-taken" predictor. Figure 5.4 and Figure 5.5 show the training progress using the gcc and go branch prediction data, respectively.

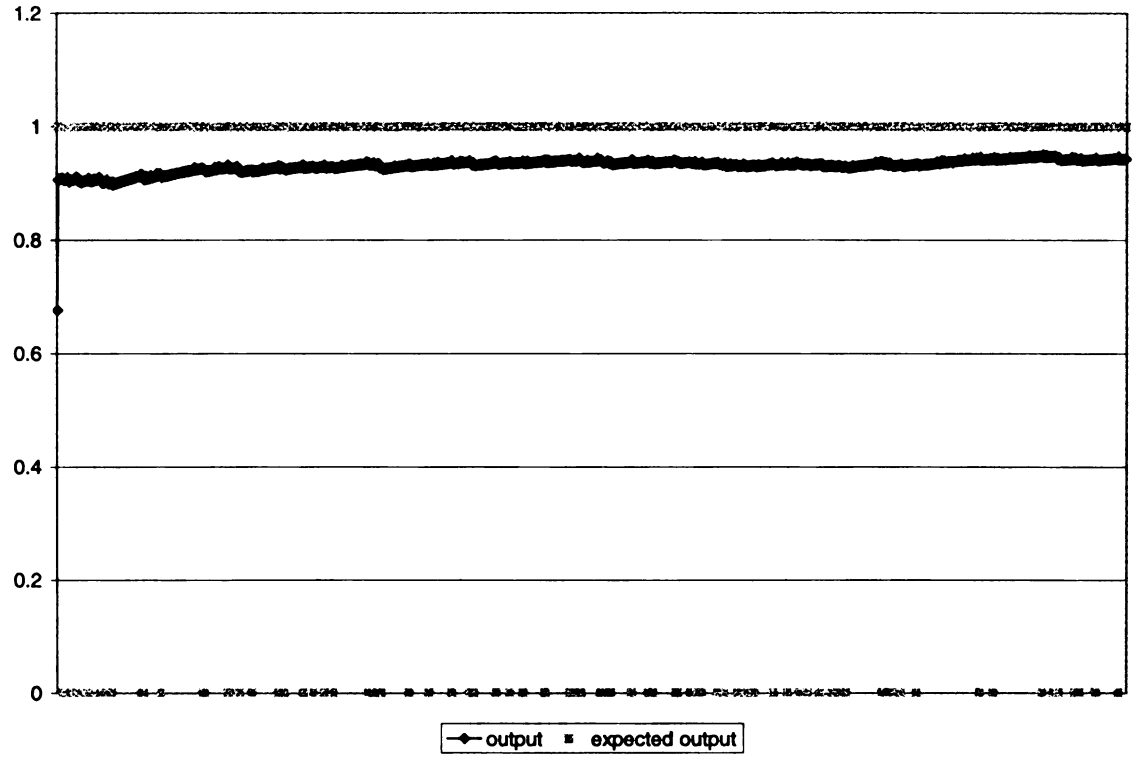


Figure 5.4: Gcc Training Progress

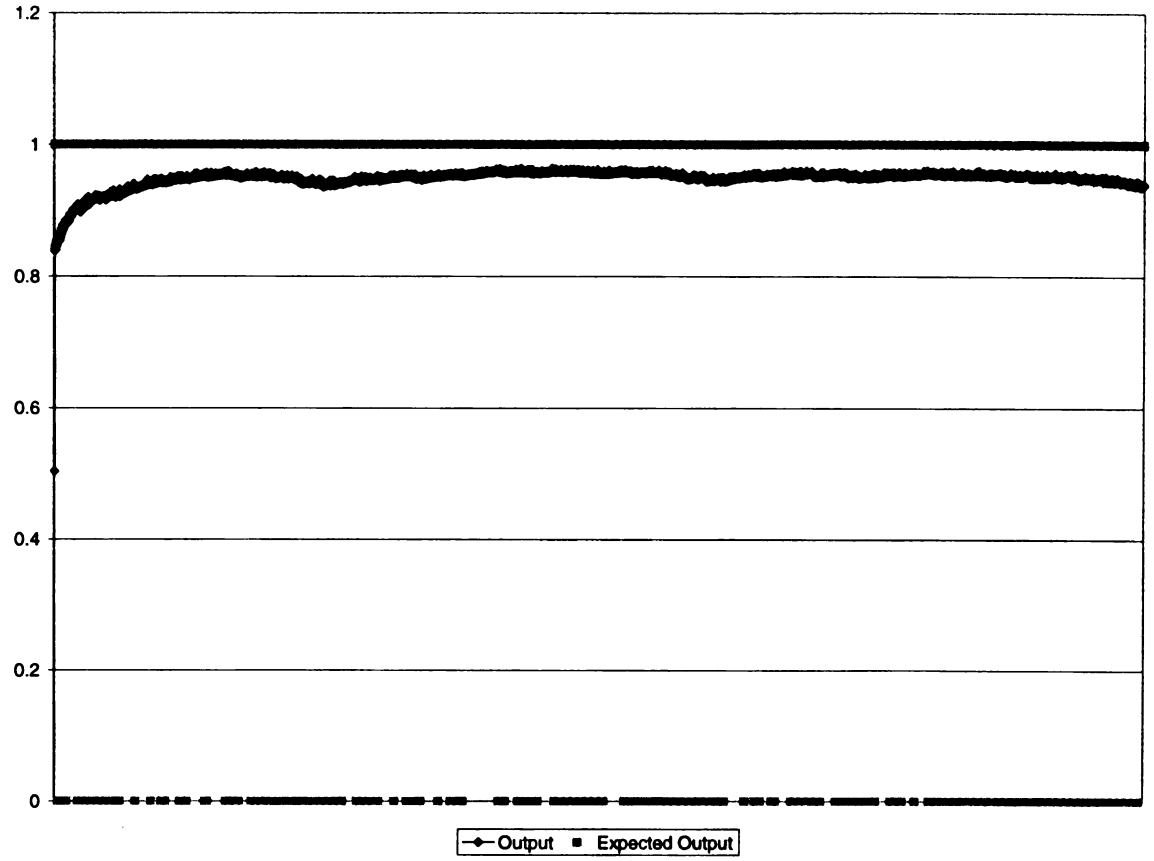


Figure 5.5: Go Training Progress

Overall, it appears that the FFBPANN predictor is defaulting to a purely "branch-always-taken" predictor since it cannot pick out any patterns in the training data set. One idea that is not fully discovered from this data is that the predictor is performing as a branch always predictor because the majority of the branches were "taken". Since none of the benchmarks had more exclusively "not taken" branches, it cannot be determined whether the predictor would perform exclusively as a "branch-always-not-taken" predictor if there were more "not taken" branches than "taken" branches in the training data.

However, using the Unix command `sed`, the gcc training data was modified to create a data set that had more "not taken" branches that "taken" branches. The lines that had a branch decision of "not taken" were changed to "taken," and the lines with "taken" branches were changed to "not taken." This would effectively give the opposite training data set. The training procedure was run again. The training progress that was expected was that the neural network would very quickly start functioning as a "branch-always-not-taken" predictor. Figure 5.6 shows the true training progress of this "opposite" run.

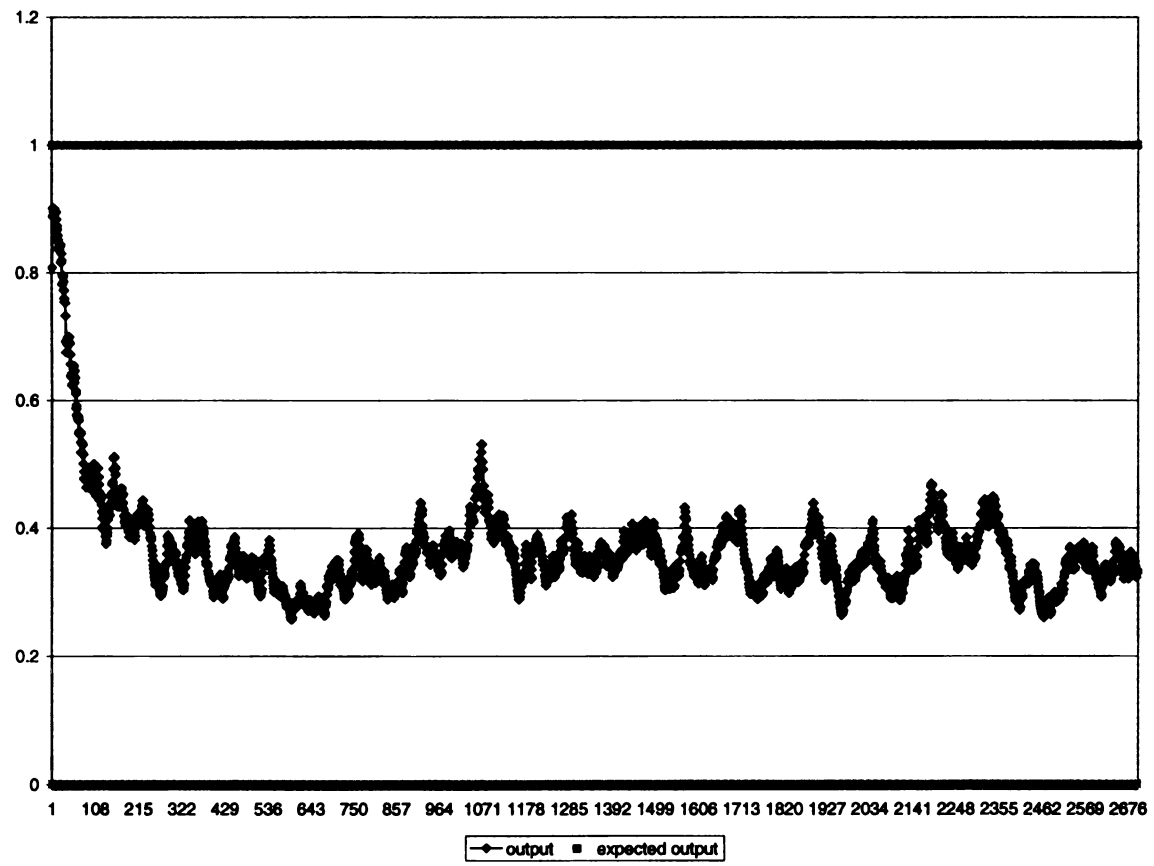


Figure 5.6: "Opposite" Gcc Training Progress

The output of the neural network in this case did trend toward behaving as a "branch-always-not-taken" predictor. The values started high (around 0.7) and tapered off toward

0.3. However, it did not reach these values as quickly as the real gcc data trained toward a "branch-always" predictor. Also, to truly be classified as a "branch-always-not-taken" predictor the output should be closer to 0 (such as a value of 0.12) – just as the "branch-always" predictor trained with the true gcc training data had output values much closer to 1 (about 0.97).

5.3 Branch Predictor Results

For completeness, the other branch predictor results gathered are included here. Table 5.3, Table 5.4, and Table 5.5 show the results for the 2-level, bimodal, and hybrid predictors, respectively. The number of instructions and branches are, of course, the same in each case. The prediction rate is calculated from the total number of successful branches (total branches minus the mispredictions) divided by the total number of branches.

$$\text{prediction rate} = \frac{\text{branches - mispredictions}}{\text{branches}} \tag{5.1}$$

Table 5.3: Branch Predictor Results - 2-level

Simulation	Instructions	Branches	Mispredictions	Prediction Rate
fpppp	174687565856	2774271287	177200663	0.9361
gcc	253018428	50378431	5476501	0.8920
go	32718301958	4828119436	941659343	0.8050
li	55389884875	13200007732	519133700	0.9607
mgrid	110557152489	1444135667	35077426	0.9757
perl	14237817453	2713714020	19806753	0.9927

Table 5.4: Branch Predictor Results - Bimodal

Simulation	Instructions	Branches	Mispredictions	Prediction Rate
fpppp	174687565856	2774271287	216288715	0.9220
gcc	253018428	50378431	5476501	0.8913
go	32718301958	4828119436	880996257	0.8175
li	55389884875	13200007732	1056731094	0.9199
mgrid	110557152489	1444135667	36025665	0.9751
perl	14237817476	2713714028	106292585	0.9608

Table 5.5: Branch Prediction Results - Hybrid

Simulation	Instructions	Branches	Mispredictions	Prediction Rate
fpppp	174687565856	2774271287	166465777	0.9400
gcc	253018428	50378431	4699856	0.9067
go	32718301958	4828119436	848553348	0.8242
li	55389884875	13200007732	513342336	0.9611
mgrid	110557152489	1444135667	34751699	0.9759
perl	14237817453	2713714020	19742856	0.9927

Unfortunately, results for the FFBPANN predictor are not included, because the training did not produce a better branch predictor. Since the FFBPANN appears to have been behaving as a "branch-always-taken" predictor, the prediction rate would be equal to the percent of "taken" branches in the benchmark. Except for the mgrid benchmark, in which 95% of the training data were "taken" branches, the prediction rate would be significantly poorer compared to the other predictors (i.e. about 60%). Even though the FFBPANN predictor was actually a hybrid predictor – a combination of a bimodal and a purely neural network predictor – it would have performed at best as well as the bimodal

predictor. Effectively, the FFBPANN predictor was a combination of a "branch always" predictor and the bimodal predictor.

The more common predictors (2-level and bimodal), used in SimpleScalar, employ much simpler techniques for branch prediction – i.e., from a software perspective, shift registers and tables. They perform very well under most conditions – however, sometimes performance drops drastically, as in the case of the go benchmark. This can be easily observed in Figure 5.7, and Figure 5.8. These figures graph the predictor data presented in the preceding tables. Figure 5.7 shows the predictor performance per predictor. Figure 5.8 displays how the predictors performed in each of the 6 different SPEC95 simulations.

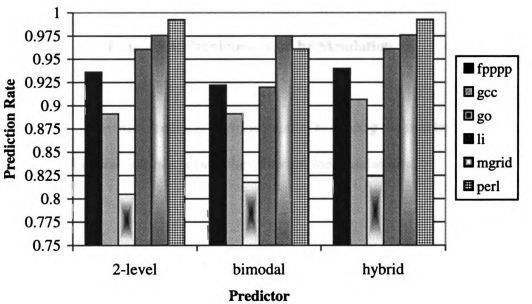


Figure 5.7: Prediction Rate by Predictor

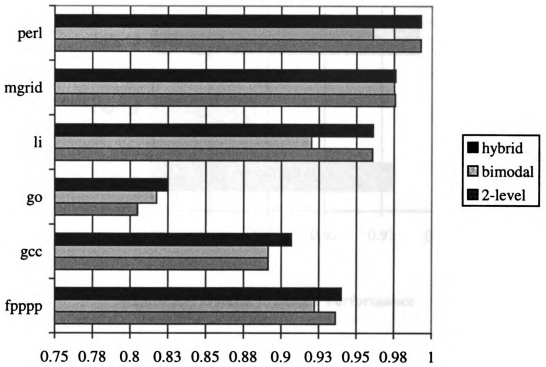


Figure 5.8: Prediction Rate by Simulation

Figure 5.9 shows the overall predictor performance by averaging the number of correct predictions over the total number of branches from all the simulations.

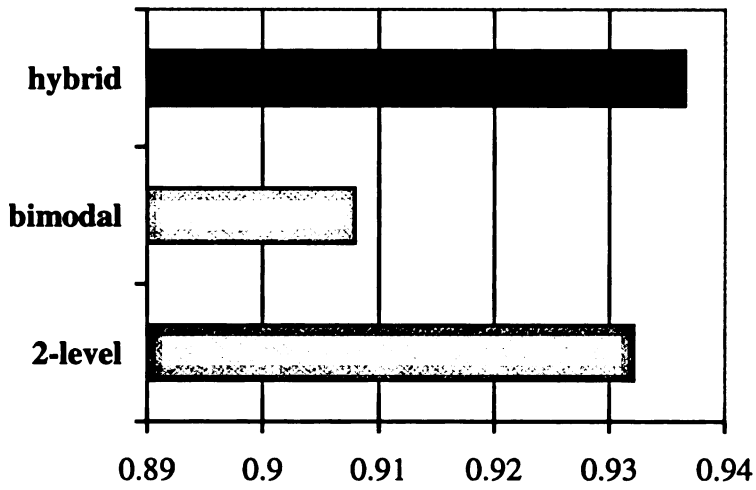


Figure 5.9: Overall Predictor Performance

Chapter 6: Conclusions and Future Work

6.1 Conclusions

From the results, the hybrid, or combination, predictor performs the best. For branch prediction to benefit from an artificial neural network, it can safely be stated that a combination of a purely ANN predictor and another more common predictor (one that has shown consistently good results, such as the bimodal or 2-level predictors) is the proper solution. The fact remains that artificial neural networks' characteristics lend itself well to prediction. However, in this case, a purely feed-forward back-propagation ANN did not perform well at all. Two tasks exist to continue on with testing the use of artificial neural networks for branch prediction. In-depth statistical analysis of branch instruction behavior is needed to determine if another ANN design is better suited for prediction of microprocessor branches. Also, the inputs used for the ANN should be examined closer to determine which inputs should be used and not used to successfully train the ANN.

6.2 Future Work

6.2.1 Other inputs

Further examination of the variables available in SimpleScalar to be used as inputs should be done. Certain inputs may not be desirable and others may have been overlooked. Specifically, the branch address could probably not be used. This prospect should also be tied to the following area of exploration: the selection of better training

methods and use of other ANN designs. By changing or adding new inputs a different learning method could be used. For example, if previous branch decisions are taken as inputs, a recurrent learning method could be utilized. Also, variables from previous branch instructions could be examined and taken as inputs.

6.2.2 Other Neural Networks and Training methods

The methodology used for training in this thesis was fairly simple, as the focus was to obtain a working simulation. Other methods of training the ANN should be explored. New training methodologies for Artificial Neural Networks are constantly a topic of research, and new methods may be found that may allow an ANN for branch prediction to be successful. Other artificial neural network styles should be explored. Statistical analysis of branch prediction may hint toward another neural network structure that is better suited for this task. Figure 6.1 shows other learning processes that could be explored.

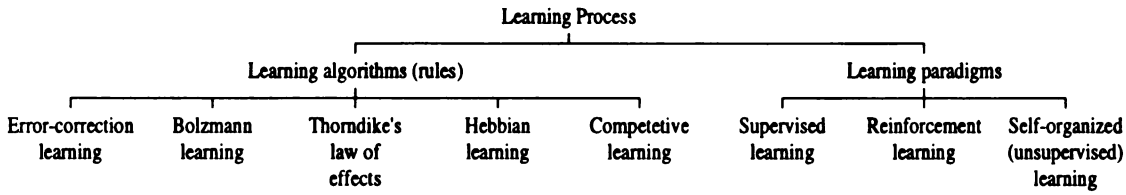


Figure 6.1: the taxonomy of learning

A recurrent network should probably be explored. By choosing a recurrent network, previous branch prediction performance could be taken into account. An example of a recurrent network is shown in Figure 6.2.

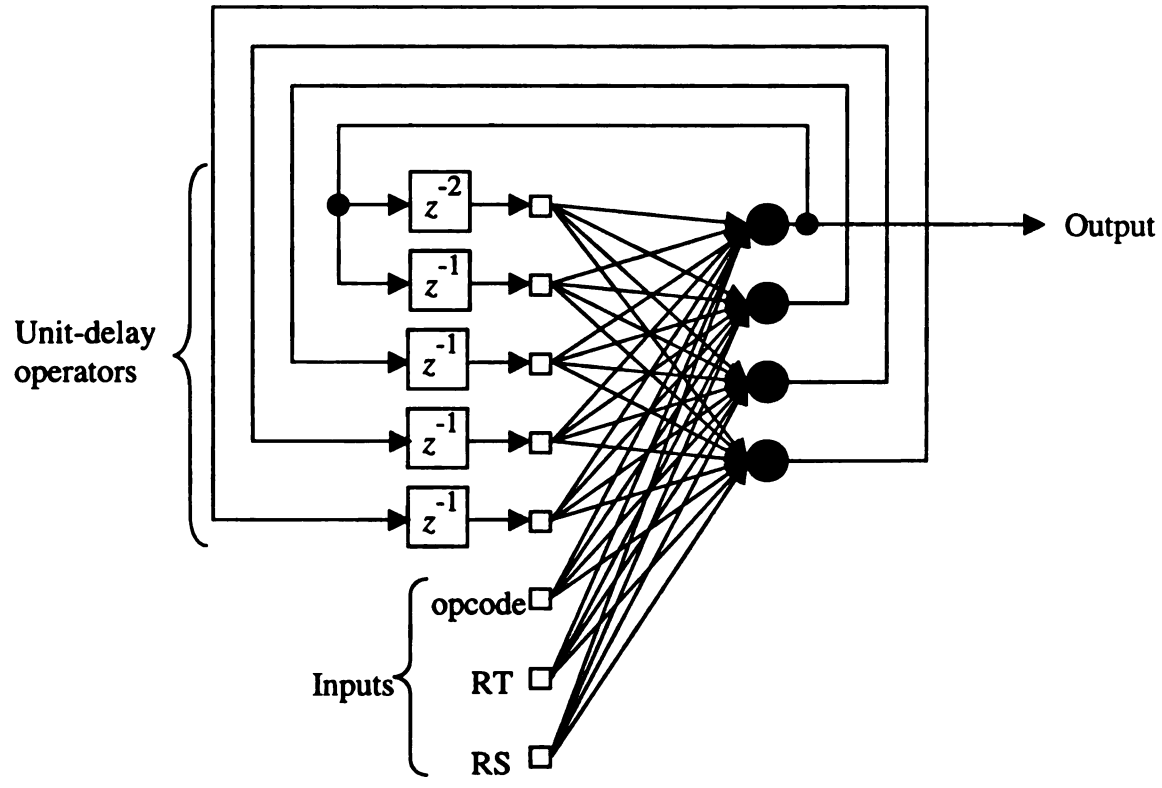


Figure 6.2: Recurrent network

One such recurrent network is a Hopfield network. However, the Hopfield network is equivalent to an associative memory or content-addressable memory [Haykin]. Because of this, the Hopfield network might not be the best choice for branch prediction. Since it is a form of memory – good at retrieving a pattern from memory – and it has been determined that there not very many recurring patterns in the branch prediction training data, the Hopfield network may not improve the neural network as a branch predictor. One limitation of a Hopfield network is that is does not have any hidden neurons. Also, the Hopfield network cannot be operated in a supervised manner, which could impede training of the network.

If using a recurrent network is expanded upon (i.e. lifting the two limitations discussed for Hopfield networks), one learning method that looks promising is Boltzmann Learning. Taking a stochastic approach to branch prediction seems reasonable, since the question that this would answer is: "what is the probability that this branch instruction will be 'taken'?" Boltzmann learning should probably operate in the Clamped Condition since the visible neurons should be used to supply the network with inputs from the environment. The Boltzmann learning is a very different learning method from Hebbian learning since there is no error-correction learning [Haykin]. Boltzmann learning has symmetric connections, which allow for two-way communications between neurons. The weight update function, therefore, becomes a function of correlations:

$$\Delta w_{ji} = \eta(\rho_{ji}^{+} - \rho_{ji}^{-}), \qquad \begin{matrix} i, j = 1,2,...,N \\ i \neq j \end{matrix}$$

In this gradient descent rule, the two averages ρ_{ji}^{+} and ρ_{ji}^{-} are the conditional and unconditional correlations, respectively, between the states of neuron *j* and neuron *i* [Haykin]. Figure 6.3 shows a basic Boltzmann machine.

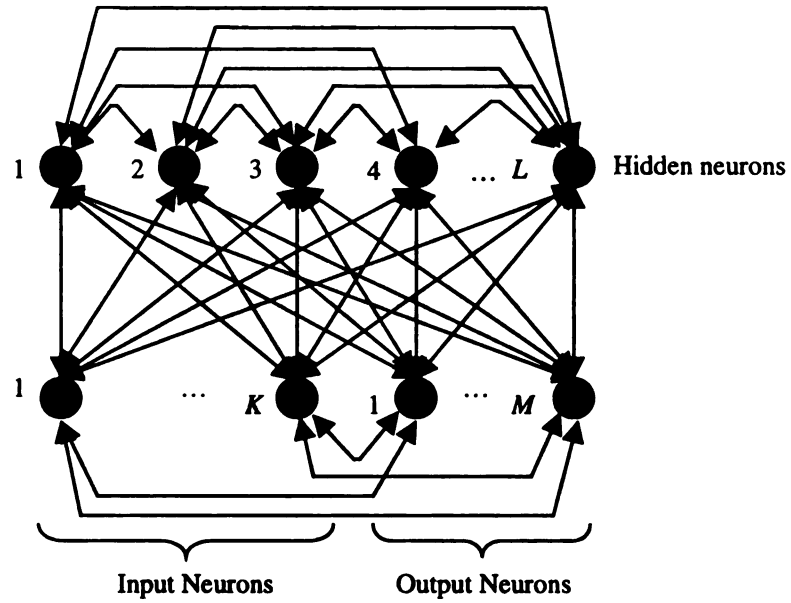


Figure 6.3: Boltzmann Machine

Another option could be to use a Competitive Learning technique. In this manner, two output neurons, one representing a "taken" decision, and the other representing a "not taken" decision, would compete for an active, or "on" state. The weight update function for a competitive network is determined by which neuron wins the competition:

$$\Delta w = \begin{cases} \eta(x_i - w_{ji}) & \text{if neuron } j \text{ wins the competition} \\ 0 & \text{if neuron } j \text{ loses the competition} \end{cases}$$

Also, it is assumed that the sum of the synaptic weight for a given node is 1:

$$\sum_i w_{ji}^2 = 1 \quad \text{for all } j$$

A simple possibility for a competitive learning network is shown in Figure 6.4.

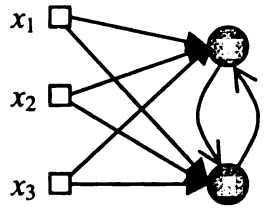


Figure 6.4: Competitive learning network

6.2.3 BTB

Another way that an ANN could be used to improve branch prediction within the SimpleScalar architecture would be to use an ANN for address prediction. The ANN in this thesis was used just for direction prediction (do not take or take the branch). SimpleScalar has the capability for branch address prediction through the use of a Branch-Target Buffer. An ANN could be used to enhance the branch address prediction.

6.2.4 State Output

One possibility that was not explored in this thesis is to train the ANN to be a branch state predictor – using the state of the bimodal or 2-level predictor as the output for training. While re-examining the code for SimpleScalar, it was determined that the coding for the bimodal and 2-level predictors determine which state the predictor is in (0, 1, 2, or 3). When the state is 2 or 3, the prediction is "taken", 0 and 1 are "not taken". Instead of recording the 0 or 1 ("not taken" or "taken") output, the state output could be gathered with the appropriate inputs. Then, instead of training for the "taken" or "not taken" output, the predictor state could be trained for. In essence, this would be training the ANN to be a better bimodal or 2-level predictor.

6.2.5 Hardware Feasibility

This thesis did not explore the hardware implementation of ANNs for branch prediction. However, both Haykin (in his book, Neural Networks: A Comprehensive Foundation) and Wang (in his dissertation, CMOS VLSI Implementations of a New Feedback Neural Network Architecture) state that VLSI implementations of ANNs are possible, and they are good for real time operations like control, signal processing and pattern recognition. A 6 neuron ANN would occupy 2.2 x 2.2 mm^2 using 2μm n-well technology. A 50 neuron CMOS analog chip uses 63,025 transistors and occupies 6.8 mm x 4.6 mm using 2μm CMOS n-well technology [Wang]. The 4x3x2x1 ANN discussed in this thesis uses 10 neurons. In comparison, Intel's Pentium 4 microprocessor contains 42 millions transistors [Intel]. So, if an ANN as a branch predictor can be successfully simulated, a hardware design could be attainable.

BIBLIOGRAPHY

BIBLIOGRAPHY

[Anderson] Anderson, Paul and Gail Anderson. Advanced C Tips and Techniques. Indianapolis: Hayden Books, 1988.

[Austin] Austin, Todd and Doug Burger. SimpleScalar Tutorial. [Online] Available http://www.cs.wisc.edu/~mscalar/ss/tutorial.html, January 1998.

[Austin97] Austin, Todd. A User's and Hacker's Guide to the SimpleScalar Architectural Research Toolset. January, 1997.

[Brehob] Brehob, Mark, Travis Doom, Richard Enbody, William H. Moore, Sherry Q. Moore,Ron Sass, and Charles Severance. Beyond RISC - The Post-RISC Architecture. [Online] Available http://www.egr.msu.edu/~crs/papers/postrisc2/, August 2, 1999.

[Dazsi] Dazsi, Brian, and Richard Enbody. Artificial Neural Networks for Branch Prediction. MSU-CSE-01-22. Computer Science and Engineering, Michigan State University, 2001.

[Demuth] Demuth, Howard and MArk Beale. Neural Network Toolbox User's Guide. Natick, Massachusetts: The MathWorks, Inc., 1994.

[Driesen] Driesen, Karel and Urs Holzle. Accurate Indirect Branch Prediction. The 25th International Symposium on Computer Architecture. IEEE, Inc., 1998.

[Emer] Emer, Joel and Nikolas Gloy. *A Language for Describing Predictors and its Application to Automatic Synthesis.* The 24th Annual International Symposium on Computer Architecture. Association for Computing Machinery Press, 1997.

[Haykin] Haykin, Simon. Neural Networks: A Comprehensive Foundation. New Jersey: Prentice-Hall, Inc., 1994.

[Hsieh] Hsieh, Paul. Sixth Generation CPU Comparisons. [Online] Available http://www.azillionmonkeys.com/qed/cpuwar.html, August 2, 1999.

[Intel] Intel Corporation. Micoprocessor Quick Reference Guide. [Online] Available http://www.intel.com/pressroom/kits/quickrefyr.htm, August 8, 2001.

[Kandel] Kandel, Eric; Schwartz, James; and Thomas, Jessell. Essentials of Neural Science and Behavior. Appleton & Lange. Norwalk, Connecticut, 1995.

[Kutza] Kutza, Karsten. Neural Networks at Your Fingertips. [Online] Available http://www.geocities.com/CapeCanaveral/1624/, July 2,1999.

[Patterson] Patterson, David A. and John L. Hennessy. Computer Architecture A Quantitative Approach. San Francisco: Morgan Kaufmann Publishers, Inc., 1996.

[SimpleScalar] The SimpleScalar Architectural Research Tool Set, Version 2.0 [Software] http://www.cs.wisc.edu/~mscalar/simplescalar.html

[Wang] Wang, Yiwen. CMOS VLSI Implementations of a New Feedback Neural Network Architecture. diss. Michigan State University, 1991.

[West] West, Patricia M., Patrick L. Brockett, Linda L. Golden. "A Comparative Analysis of Neural Networks and Statistical Methods for Predicting Consumer Choice." *Marketing Science*, vol.16, no. 4, 1997, pp.370-391.

[Yeh93] Yeh, Tse-Yu and Yale Patt. *A Comparison of Dynamic Branch Predictors that use Two Levels of Branch History.* The 20th Annual International Symposium on Computer Architecture. Los Alimitos, CA: IEEE Computer Society Press, 1993.

[Yeh92] Yeh, Tsu-Yu and Yale N. Patt. *Alternative Implementations of 2-Level Adaptive Branch Prediction.* The 19th Annual International Symposium on Computer Architecture. Association for Computing Machinery, 1992.