Application of Neural Network to Improve Dynamic Branch Prediction of Superscalar Microprocessors

P.B. Osofisan, Ph.D.*[*] and O.A. Afunlehin, M.Sc.(Eng.)

Department of Electrical and Electronics Engineering
University of Lagos, Akoka, Lagos, Nigeria

ABSTRACT

This research shows that using an Artificial Neural Network as the hardware branch predictor of a superscalar microprocessor leads to performance as good as standard branch predictors for comparable chip area. The results were obtained running several Spec95 benchmarks on an augmented version of the simple-scalar architecture simulator. The approach taken in this research is an attempt to use Neural Networks to improve the design of hardware branch predictors. It points to a combination of static and dynamic techniques using artificial intelligence. The prediction rates achieved by the holistic-non-adaptive Neural Network (NN) predictor designed are promising. Even a simple Neural Network structure without an on-line adaptive mechanism performed better than current techniques for small predictor sizes. The neural net predictor achieved almost the same rates for most of the benchmarks of the Spec95 set and it was even 20% more accurate for one of them.

However, the NN predictors developed were not able to achieve the same prediction rates as bigger standard predictor configurations. The performance of the non-adaptive NN predictors substantially decreases when the number of dynamic branches in the benchmark increases, showing that the dynamic characteristic of the benchmarks negatively affects the behaviour of the non-adaptive Neural Network predictor. This indicates that in order to increase the prediction rate in highly dynamic programs it would be necessary to incorporate an adaptive mechanism, to yield Neural Network predictors competitive with the larger standard configurations.

The method used in this study was to train an Artificial Neural Network on the dynamics of programs, and particularly conditional branch instructions, when they are being executed in a microprocessor. After training the Neural Network on traces of programs, it was implemented in the simulator to replace the existing standard predictor. In order to achieve better CPU performance, many schemes of branch prediction have been utilized. These schemes sometimes can be categorized as program-based predictors vs. profile based predictors, or static vs. dynamic schemes. This paper focuses on the study of the dynamic branch predictors since the dynamic approach of branch prediction has been developed much more than the static approach of branch prediction. However, their performances always have new and interesting discoveries based on different benchmarks and architectures.

(Keywords: neural network, branch prediction, superscalar microprocessor, benchmarks)

INTRODUCTION

The Artificial Neural Network learns about the dynamics of programs when they are being executed in a microprocessor, and then it is implemented in the microprocessor as its branch predictor.

The number of integrated circuits (ICs) in a microprocessor is growing at an enormous rate and how to use all those ICs is a topic of much debate. The number 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 (ANNs) 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 [38]. Because of their success in these fields, an Artificial Neural Network might perform well as a microprocessor branch predictor.

The goal of this paper 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:

1) Develop an Artificial Neural Network

2) Modify the Simple-Scalar simulator to accept an ANN branch predictor

3) Train the Neural Network

4) Evaluate the ANN branch predictor:

BRANCH PREDICTION

According to Schlansker et al., "branch prediction is the process of correctly predicting whether branches will be taken or not before they are actually executed" [1]. Branch prediction is a fundamental factor for achieving high performance in today's microprocessors, since in regular programs 1 of every 3 to 5 instructions is a branch instruction [16]. Correctly predicting a branch will avoid wasting clock cycles waiting for the result of a branch destination calculation to become available, and in that way they help to keep the processor pipelines as busy as possible by fetching instructions from the predicted branch destination.

Modern microprocessors use more internal parallelization and added functionality for increasing microprocessor throughput, which puts a high demand on the provision of useful instructions to execute.

If the microprocessor does not implement a branch prediction mechanism, pipeline stalls will happen. When a conditional branch is decoded, the information that is required to know its outcome might be not available. The branch condition can be the result of an instruction depending on others waiting to be computed. Until the branch condition is known, the processor cannot issue more instructions.

To overcome this obstacle, the processor must predict the outcome of branches, that is, predict the result of the branch condition before it is executed. It must also implement a mechanism to speculatively fetch the instructions along the predicted path. These instructions are issued and executed before the branch condition is determined. Therefore, if the prediction is correct, the processor can keep working without stalls. In the case that the prediction is wrong, the processor needs to discard all the instructions incorrectly fetched, issued, and executed. This situation is called a *branch misprediction*, and the clock cycles wasted on restoring the correct processor's state are called the *branch misprediction penalty*.

Two-level Branch Predictors

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 shows how this is achieved in a state diagram.

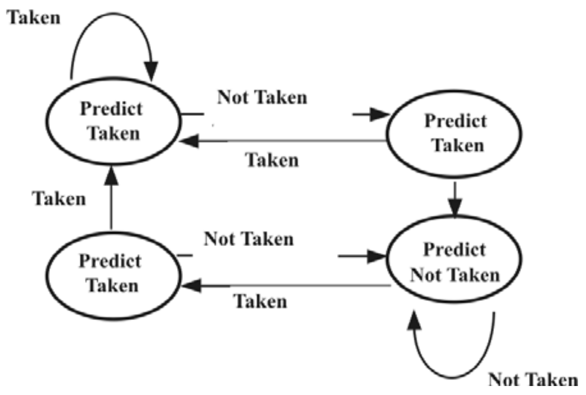


Figure 1: Two-Bit Prediction Method States.

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.

Among the current dynamic techniques, two-level branch predictors have been shown to be one of the best mechanisms to predict the outcome of conditional branches. A typical two-level branch predictor configuration has one level that generates an index, which is then used in the second level to access a table containing information about the history of branches. The table is called the Pattern History Table (PHT).

Each entry in the table contains information about the previous behavior of the branches mapped to it. The indexing function is based on the address of the current branch to be predicted and a register called the Branch History Register (BHR), which keeps track of the outcome of the most recent branches.

There are several variations of the two-level branch predictor scheme, depending upon how the PHT is organized, what information is stored in it, and what information is used to index its content. The different combinations can index the PHT using history kept globally for a set of addresses or for individual addresses. The PHT can contain a two-bit saturating counter or a fixed 1-bit prediction in each cell. This class of predictors often requires large and costly tables.

The gshare predictor is a scheme that attempts to reduce the amount of hardware needed by combining the address information and the BHR register. It performs an XOR operation (exclusive or operation) between the BHR register and the branch address, and the result is then used to index the PHT. This scheme has a limitation; it increases interference in the pattern history table. [3].

The prediction is obtained based on dynamically stored information related to the branch to be predicted, its previous outcomes, the previous outcomes of the most recent branches, or any other architectural information available at the time of making the prediction. Table 1 summarizes the different configurations for two-level adaptive branch predictors according to its organization.

Table 1: Two-Level Adaptive Predictor's Combinations.

History Kept	Table Content
Globally (G)	2-bit Saturating Counters (A)
For a Set of Addresses (S)	Fixed Prediction (S)
Individual Addresses (P)	---

Selected Predictor Configurations

The structures of some particular configurations to be used as the base of proposed predictor are illustrated below. The predictors selected for this study were examined because they are variations of the Two-Level Adaptive Branch Predictors which have been researched, implemented and improved in many previous studies [3].

The GA Predictor

In the architecture known as Global Address (GA) predictor (Figure 2), the PHT has its rows indexed by the BHR and its columns indexed by the branch address. The BHR is an n-bits wide register used to address the rows of the PHT, which contains information about the behaviour of the (n) most recently executed branches. The columns are indexed by the (j) lower bits of the branch address, where the number of columns of the PHT determines (j).

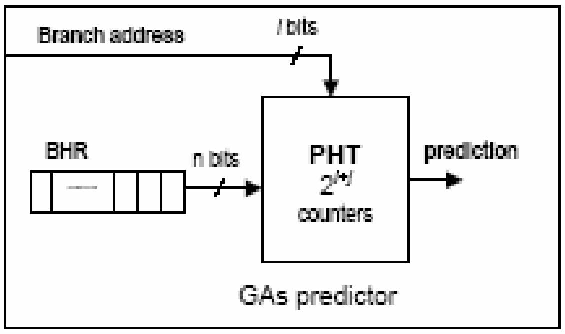


Figure 2: Schematic of GA predictor.

The Gshare Predictor

Figure 3 shows another approach called the gshare predictor. This scheme reduces the amount of hardware needed by combining the address information and the BHR register. The operation between the BHR and the branch address is an XOR function; the result of this operation is then used to address the rows of a single column PHT. As mentioned before, this scheme has a limitation; it increases interference in the pattern history table.

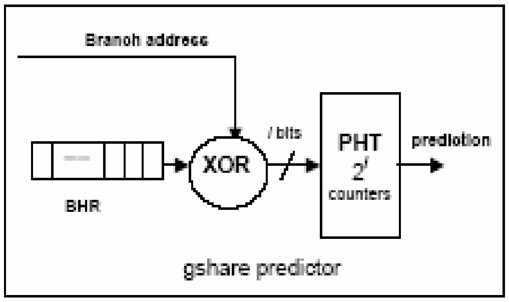


Figure 3: Schematic of a gshare Predictor.

The PA Predictor

Another approach is the one known as Per Address (PA) predictors. This scheme, shown in Figure 4, uses information about the previous behaviour of the branch to be predicted to address the rows of the PHT. The information is stored in a table (BHRs) addressed by the current branch address. The output of this table is then used to index the PHT.

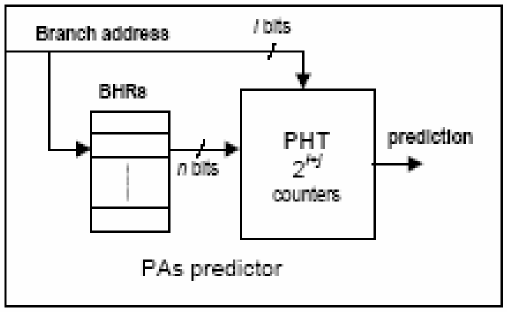


Figure 4: Schematic of Per–Address Predictor.

Hybrid Predictors

Hybrid branch predictors combine several prediction strategies into a single predictor; these predictors use a mechanism for selecting the best predictor for a given branch. What a hybrid branch predictor scheme does is to decide which registers are important for every different type of branch. A comparison of different combinations of single-scheme predictors is presented in [3], which shows the misprediction rates for each scheme with six different predictor sizes, varying from 8KB to 256KB. The single-scheme predictors used in that study are gshare, PA, a single column PHT indexed with the branch address, and a static branch predictor.

For all the predictors and their possible combinations, the misprediction rate was obtained and compared. The misprediction rates given were the average of the rates achieved for the six SPECint92 benchmarks [3]. Their results showed that for every different predictor size the ordering of the predictor class combinations was the same. The gshare/PA combination achieved the lowest misprediction rate which was on average 13% lower than that of its closest competitor, gshare/static. The best single-scheme predictor at all levels of cost was gshare [3].

The gshare/PA was the only combination of single-scheme predictors able to obtain a benefit of both types of correlation between branches.

The two types of correlation are the correlation between the last branch outcomes and the branch to be predicted, and the correlation between previous outcomes of the current branch and itself.

The gshare component predicts more accurately branches whose outcomes are dependent on the outcomes of other static branches, and the PA component predicts more accurately branches whose outcomes are dependent on previous outcomes of the same static branch [3]. Since this predictor achieved the highest performance, we can infer that both types of history information are critical for achieving high levels of accuracy.

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. Most current microprocessors share much in common with this superscalar architecture [33] the post-RISC architecture. Figure 6 [28] shows the generic layout of a Post-RISC processor pipeline.

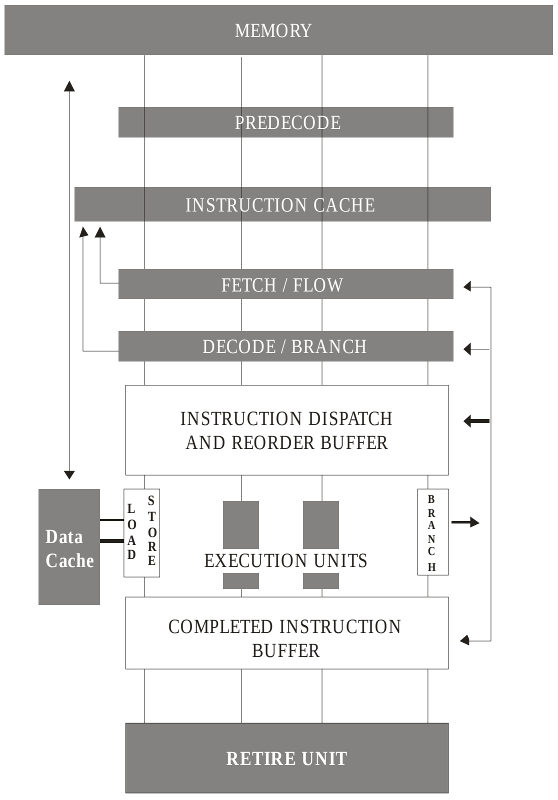


Figure 6: The Post-Reduced Instruction Set Computer (RISC) Architecture.

From examining Figure 6, there are six important steps in the processor pipeline. First, instructions are read from memory in the Pre-decode 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 [33].

As a part of pre-decoding, 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, instructions fetched from the I-Cache are 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 are needed 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 pre-decode 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 6 shows that branch prediction effects multiple stages of the pipeline.

Simple-Scalar Software Architecture

Simple-Scalar has a modular layout. This layout allows for great versatility. Components can be added or modified easily. Figure 7 shows the software structure for Simple-Scalar. Most of the performance core are optional modules [29].

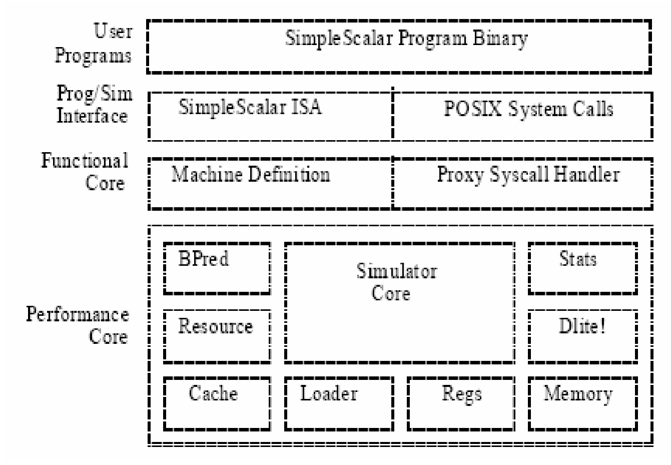


Figure7: Simple-Scalar Software Architecture.

Hardware Architecture

The Hardware architecture of the Simple-Scalar simulator closely follows the Post-RISC architecture previously described. Figure 8 shows the out-of-order pipeline for the Simple-Scalar hardware architecture.

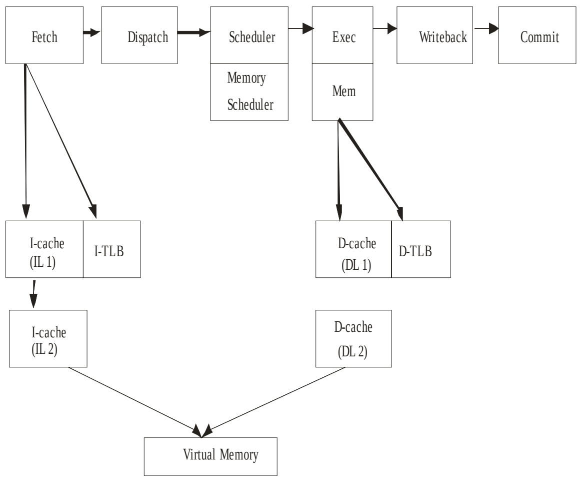


Figure 8: Out-of-Order Issue Architecture.

Superscalar Microprocessor Architecture Of PowerPC™ 603e Microprocessor.

This presentation discusses techniques for optimizing instruction execution in a superscalar microprocessor architecture such as the PowerPC™ 603e microprocessor (Figure 9).

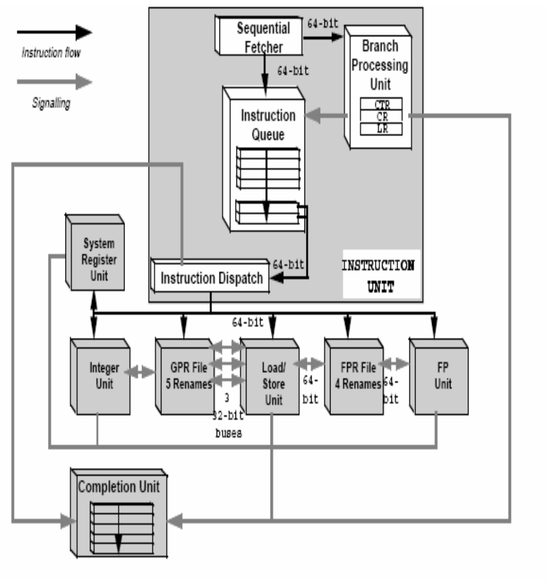


Figure 9: Block Diagram of 603e.

Instruction execution in a superscalar processor is enhanced by allowing the parallel execution of multiple instructions. In order to enable the maximum potential of most superscalar processors, one needs to be aware of their instruction flow and execution mechanisms. Optimal performance in a microprocessor can be attained by ensuring a continuous flow of instructions through the instruction pipeline.

Being aware of the dependencies and constraints of the instruction flow mechanisms allows one to generate code that can most effectively and optimally take advantage of all the capabilities of a superscalar processor such as the PowerPC 603e microprocessor. The 603e is a low-power implementation of the PowerPC family of reduced instruction set computer (RISC) microprocessors.

The 603e is a superscalar processor capable of issuing and retiring as many as three instructions per clock. Instructions can “execute” out-of-order for increased performance, but they “retire” in-order to ensure functional correctness and well-ordered behaviour. In this work, it is important to discuss the instruction flow mechanism of the PowerPC 603e microprocessor and then describe dependencies and constraints that should be avoided to reduce stalls in the instruction pipeline and maximize performance.

By closely examining the instruction flow mechanism of the 603e, a software developer will not only be able to optimize code for the 603e, but will also be able to understand some of the general principles behind Superscalar microprocessors that can impact performance.

ANNS USED IN THIS RESEARCH

For the present research the structure of the neural net is kept as simple as possible. A simple NN used has an input layer with 40 elements and a single processing element on the output layer. Slightly more complex examples add 10 and 20 elements in a hidden layer. While the data reported for this study used this simple feedforward structure, the implementation of other structures which would simplify the extraction of the PHT indexing function were also considered.

The other NNs used were Learning Vector Quantization (LVQ) nets. The LVQ NN has one competitive layer (Kohonen layer) that performs the quantization and an output layer that performs the classification. Each element in the output layer is assigned an equal number of elements from the Kohonen layer.

Figure 5 shows a diagram of an LVQ net. During training, every element in the Kohonen layer is presented with each entire input vector. The distance (*d*$_i$) of every PE's weight vector *(w*$_{ij}$) to the input vector *(*x) is computed and the closest one is declared the winner. Euclidean distance is used to determine the distance between the input vectors and the weight values for a particular PE.

.

$$di = \|w_i - x\| = \left\{ \sum_{j=1}^{N} \left(w_{ij} - x_j\right)^2 \right\}^{1/2}$$

Then if the PE is in the class of the input vector, it is moved closer to the input vector. If the winning PE is not in the class of the input vector, it is moved away from the input vector. This way, the PEs group together in regions associated with each class.

During recall, the distance from the input vector to each PE is calculated and the closest PE is declared to be the winner. Then, the input vector is classified as belonging to the class corresponding to the winner PE. The two output classes are the not-taken branch (NT) and taken (T) branch. The NT class provides a 0, and the T class provides a 1 to the predictor's output. The LVQ NNs used in this research has 128 and 256 elements in the Kohonen layer and two outputs.

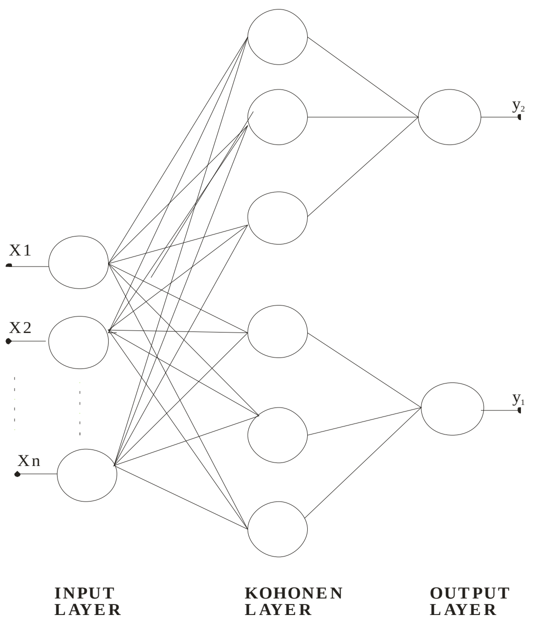


Figure 5: Learning Vector Quantization Neural Net.

Neural Net Predictor

The aim of the neural net (NN) is to find a better function for combining the information available to the predictor. The idea is to use a NN to find the optimal function to merge the information available in the microprocessor architecture to create a more accurate predictor. This function could then be implemented in the system.

The methodology is to let the NN learn about the dynamics of programs, and particularly, conditional branch instructions, while they are being executed in a microprocessor. The available information consists of snapshots of all the registers involved in the prediction mechanism for every branch in the program, and during training, the correct outcome for the branch provided by a perfect theoretical predictor.

The NN's task is to learn the function of the whole predictor. After training, the NN is to be studied to extract knowledge for use in designing hardware architecture of the predictor. Figure 10 shows a scheme of a possible prediction mechanism using a NN to implement the hardware predictor. This is the approach explored here. The PHT is included in the neural net architecture.

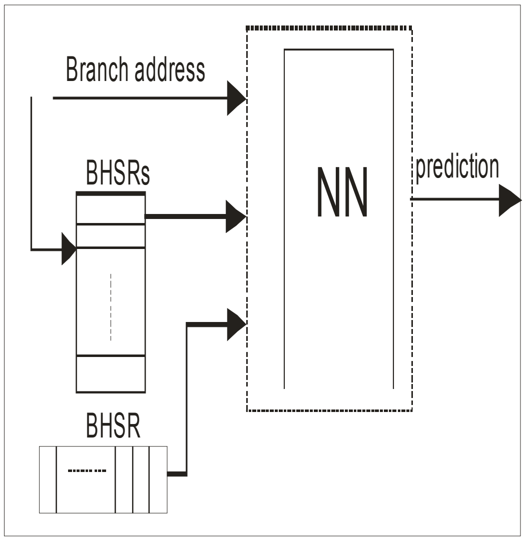


Figure10: Schematic of predictor using a NN to implement the entire predictor

The net is provided with all the dynamic information related to the current and previous branches that other schemes utilize. This way we let the NN decide which variables are important and which could be discarded. Hybrid branch predictors combine several prediction strategies into a single predictor; these predictors use a mechanism for selecting the best predictor for a given branch. All current and historical behaviour is available to the NN during training. Hence, the NN should be able to identify the information important to predicting each branch. This, in effect, implements an internal selection mechanism like that of a hybrid branch predictor.

The dynamic data provided to the NN is created by tracing a compiled program. Hence, the NN will learn characteristics of one particular compiler and platform for which they were compiled. However, in order to keep the compiled program as generic as possible, the compiler used to create the binaries executed by the simulator is a generic compiler (gcc-2.6.3) run with standard switches [11].

The Simple-scalar Simulator

A modification of the **Simple-scalar Tool Set** is used to perform the simulation of the microprocessor architecture. "This tool set consists of compiler, assembler, linker, simulation, and visualization tools for the Simple-Scalar architecture" [7]. Simple-scalar is an execution-driven simulator; the architecture that it models is a close derivative of the MIPS architecture. In this research the out-of-order version of the simulator is used. The out-of-order issue processor simulated supports non-blocking caches, speculative execution, and implements several branch predictors [7].

Simulations Outline

In the first part of the study the simulator was modified to extract snapshots of the state of the machine every time a conditional branch has to be predicted (see Figure 11).

The snapshots consist of information about the branch to be predicted: its address, the contents of the global address branch history register (BHR), the contents of the per-address branch history register (BHRs), and the actual outcome of the branch. All the magnitudes are binary {0, 1}. The branch address is 24 bits wide, the general and per-address history registers are 8 bits wide each, and the output is 1 bit. The outcome of the branch is actually obtained in the cycle after the prediction, when the branch is executed. Instructions are internally executed at the dispatch stage, and not in the execute stage as in a real processor, due to the way that the simulator is implemented.

The data obtained from the simulator was used to train the neural net. After training the neural net, it was implemented in the simulator as a predictor (see Figure 12). The routine called NNpredict is presented with data coming from the registers inside the processor simulator and the NN's output is presented to the processor as the direction prediction, replacing the complete direction prediction structure in the processor simulator. Running the modified simulator, we then obtained the run-time performance of the neural net predictor.

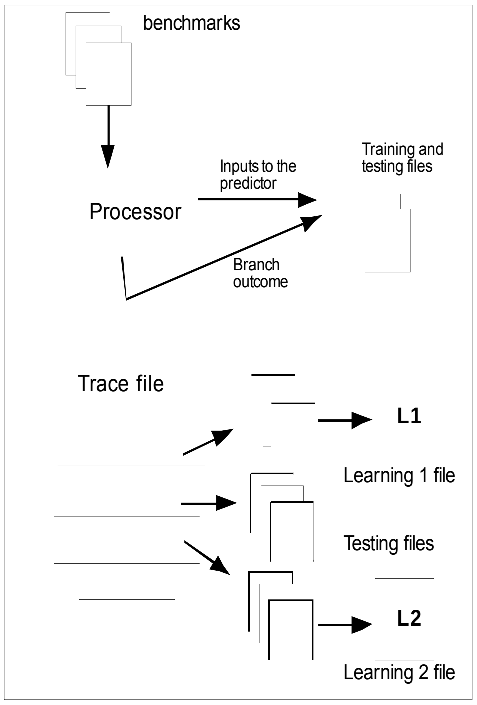


Figure11: Data Collection Scheme.

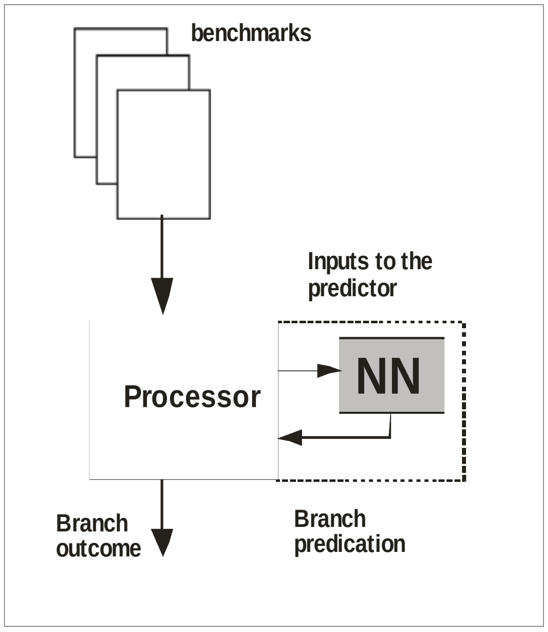


Figure 12: Integration of the Neural Net Predictor.

Prediction rate =

$$\frac{(number - of\ - branches) - (misprediction)}{(number - of\ - branches)}$$

Data Files Creation

The simulator was compiled and run on a SPARC Solaris 2. The benchmarks used are from the SPECint95 benchmarks set [21] using reference inputs. The eight benchmarks used are compress, gcc, go, ijpeg, li, m88ksim, perl and vortex.

The results reported are based on the direction misprediction rate and compared to the ones achieved by different predictors for identical processor configurations. All the benchmarks are executed by the simulator and data traces are obtained for each one of them. Every line of the files created refers to a conditional branch executed by the processor.

After approximately one million instructions, the predictor reaches a steady state and its accuracy stabilizes [20]. Therefore, capturing over one million instructions ensures that the resulting trace files contain transient and steady state information about the predictor behavior. Output files are created by executing four million instructions, this way all the benchmarks create output files with at least 300,000 lines, the selected amount of data for NN training. The number of branches to trace was selected because of the previous mentioned reasons and for practical file size reasons.

The output files are split into 3 segments, to create 3 files of 100,000 lines each. The first set is the training-1 set, it contains the first 100,000 instructions from the trace file, The second is the testing-set and the third is the training-2 set, containing the second and third 100,000 instructions respectively. Since there are eight benchmarks, each file contains 800,000 training samples.

The data format for the training and generalization set files is that of NeuralWorks input (.nna) files. The first section of the record contains all the bits which are the content of the address and branch history registers, plus a prediction bit which is the prediction of the standard two-level predictor included in the simulator. The second part of the record contains the desired network output value, in this case the real outcome of the branch, which can be not taken (' 0') or taken (' 1'). The records are all ASCII.

The frequency distributions of the 2 branch outcome classes for these files are:

Training-1 set:

File	Not-Taken branch		Taken branch	
L_compress	29283	29.3%	70717	70.7%
L_gcc	44435	44.4%	55565	55.6%
L_go	44925	44.9%	55075	55.1%
L_ijpeg	58273	58.3%	41727	41.8%
L_li	47875	47.9%	52125	52.1%
L_m88ksim	27347	27.4%	72653	72.7%
L_perl	53006	53.0%	46994	47.0%
L_vortex	29430	29.4%	70570	70.6%
L1	334574	41.8%	465426	58.2%

Training-2 set:

File	Not-Taken branch		Taken branch	
L2_compress	6213	56.2%	43787	43.8%
L2_gcc	41383	41.4%	58617	58.6%
L2_go	21883	21.9%	78117	78.1%
L2_ijpeg	59954	60.0%	40046	40.0%
L2_li	55455	55.5%	44545	44.5%
L2_m88ksim	49219	49.2%	50781	50.8%
L2_perl	50729	50.7%	49271	49.3%
L2_vortex	36361	36.4%	63639	63.6%
L2	371197	46.4%	428803	53.6%

Testing set:

File	Not-Taken branch		Taken branch	
T_compress	55998	56.0%	44002	44.0%
T_gcc	53470	53.5%	46530	46.5%
T_go	21921	21.9%	78079	78.1%
T_ijpeg	59940	59.9%	40060	40.1%
T_li	52485	52.5%	47515	47.5%
T_m88ksim	48029	48.0%	51971	52.0%
T_perl	56137	56.1%	43863	43.9%
T_vortex	31874	31.9%	68126	68.1%

Training Procedure

For both types of nets, it is necessary to adjust the I/O parameters to fit the input (.nna) files. Under the I/O menu in parameters, we find the following parameters: The input field start is 1, and the output field start is 42 for the back-propagation nets and 43 for the LVQs. The network ranges are 0.00 to 1.00 for both input and output since all the magnitudes are binary bits. Learning and recall are read from a file in sequential order, since the randomized reading option cannot be used with the size of the files used in this research.

The back propagation nets are trained on L1.nna and L2.nna for 1, 2, 3 and 4 passes over the training file. That is, 800,000, 1,600,000, 2,400,000 and 3,200,000 iterations. The LVQ nets are trained on the same files, but the number of iterations is: (1) 800,000 for LVQ1 plus 100,000 for LVQ2. (2) 1,600,000 for LVQ1 plus 100,000 for LVQ2. (3) 3,200,000 for LVQ1 plus 100,000 for LVQ2 and (4) 4,000,000 for LVQ1 plus 1,000,000 for LVQ2.

The training procedure takes between two and fourteen hours (computer running time), depending on the size of the NN, the amount of pairs presented to it, and its architecture.

RESULTS

Neural Networks Training Comparison

The neural network simulator used in this project is NeuralWorks Professional II/Plus®, of NeuralWare, Inc. It provides a classification rate table containing the classification rate for each class and an average of the classification rates over the different classes. The classification rate can be defined for a given class, as the number of correct answers divided by the total number of pairs provided, in that class.

We can see from Figure 13 that for some of the NNs, the average classification result on the test files is higher than the classification result on the training file. This is because the training file is composed of fractions of the eight different trace files and the transitions between benchmarks during learning cause high errors. We can also see that the NNs that achieved high classification results during training did not generalize well, that is they did not perform as well on the test files. The networks tested are:

. Backpropagation: Input layer: 40 elements; hidden: 0, 10 and 20; output: 1

LVQ: Input layer: 40 elements, Kohonen: 128 and 256; output: 2

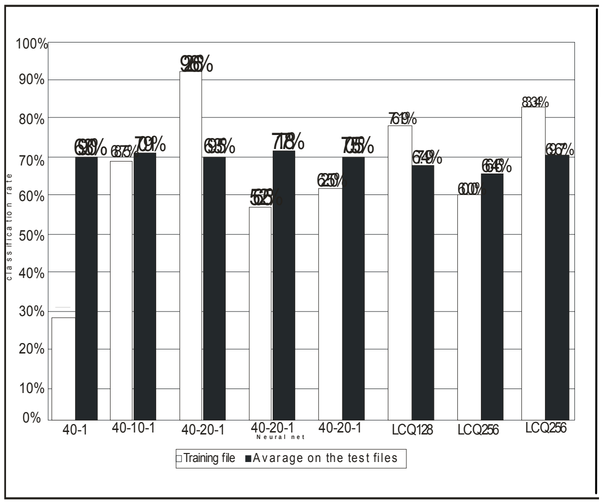
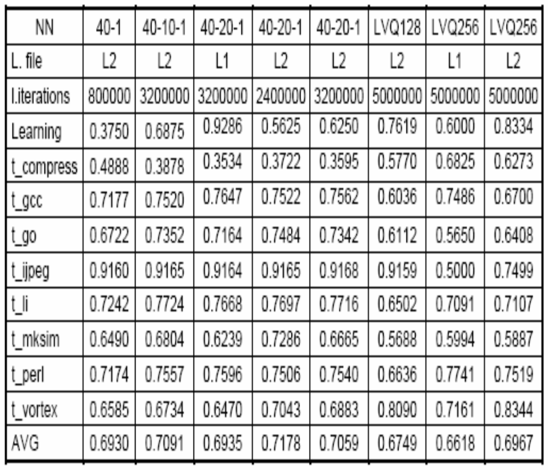


Figure 13: Classification results for the best NNs.

Table 2 summarizes the best nets based on the average testing classification rate. Figure 13 shows the learning and average testing classification rates for those best nets.



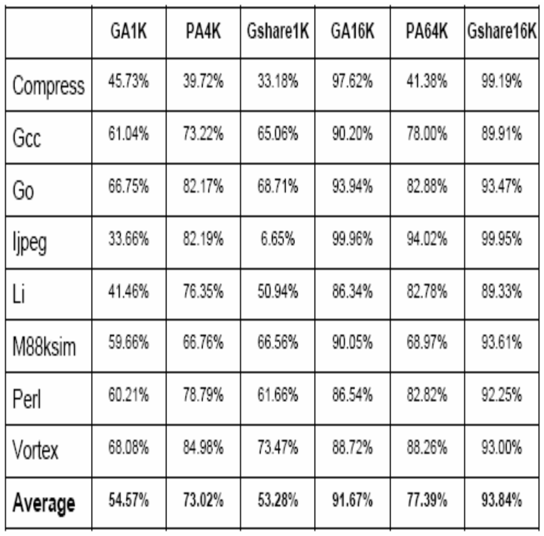
NN	40-1	40-10-1	40-20-1	40-20-1	40-20-1	LVQ128	LVQ256	LVQ256
L. file	L2	L2	L1	L2	L2	L2	L1	L2
l.iterations	800000	3200000	3200000	2400000	3200000	5000000	5000000	5000000
Learning	0.3750	0.6875	0.9286	0.5625	0.6250	0.7619	0.6000	0.8334
t_compress	0.4888	0.3878	0.3534	0.3722	0.3595	0.5770	0.6825	0.6273
t_gcc	0.7177	0.7520	0.7647	0.7522	0.7562	0.6036	0.7486	0.6700
t_go	0.6722	0.7352	0.7164	0.7484	0.7342	0.6112	0.5650	0.6408
t_ijpeg	0.9160	0.9165	0.9164	0.9165	0.9168	0.9159	0.5000	0.7499
t_li	0.7242	0.7724	0.7668	0.7697	0.7716	0.6502	0.7091	0.7107
t_mksim	0.6490	0.6804	0.6239	0.7286	0.6665	0.5688	0.5994	0.5887
t_perl	0.7174	0.7557	0.7596	0.7506	0.7540	0.6636	0.7741	0.7519
t_vortex	0.6585	0.6734	0.6470	0.7043	0.6883	0.8090	0.7161	0.8344
AVG	0.6930	0.7091	0.6935	0.7178	0.7059	0.6749	0.6618	0.6967

Table 2: Classification Results from NeuralWorks for the best NNs.

Simulator Prediction Accuracy Results

Tables 3, 4, and 5 show the prediction rates achieved by the standard and the NN predictors for each one of the eight benchmarks and the arithmetic mean for each predictor.

Table 3 compares the prediction rates achieved by different configurations of standard predictors and the two best NN predictors. These values are the prediction rates achieved only for those branches that actually use the direction predictor that is conditional branches. The prediction rates are obtained from the trace files obtained when running the benchmark programs, they are not the prediction rates reported by the simulator (which include all branches executed by the processor). Before showing a performance comparison, the results for different standard predictors and NN predictors are presented.



	GA1K	PA4K	Gshare1K	GA16K	PA64K	Gshare16K
Compress	45.73%	39.72%	33.18%	97.62%	41.38%	99.19%
Gcc	61.04%	73.22%	65.06%	90.20%	78.00%	89.91%
Go	66.75%	82.17%	68.71%	93.94%	82.88%	93.47%
Ijpeg	33.66%	82.19%	6.65%	99.96%	94.02%	99.95%
Li	41.46%	76.35%	50.94%	86.34%	82.78%	89.33%
M88ksim	59.66%	66.76%	66.56%	90.05%	68.97%	93.61%
Perl	60.21%	78.79%	61.66%	86.54%	82.82%	92.25%
Vortex	68.08%	84.98%	73.47%	88.72%	88.26%	93.00%
Average	**54.57%**	**73.02%**	**53.28%**	**91.67%**	**77.39%**	**93.84%**

Table 3: Prediction Rates for the Standard Predictors Tested.

Standard Predictors Tested

Table 3 shows the prediction rates achieved by the different standard predictor configurations. Figure 14 shows the average prediction rates for the different configurations.

The GA1K and GA16K predictors are global address two-level branch predictors whose level-1 table has 1 entry and PHT has 1024 and 16384 entries respectively.

The PA4K and PA64K are two-level branch predictor whose level-1 table has 4 and 8 entries and its PHTs have 4096 and 65536 entries respectively.

The gshare1K and gshare16K predictors are gshare predictors whose PHT has 1024 and 16384 entries respectively, its level-1table has 1 entry and the XOR flag is turned on.

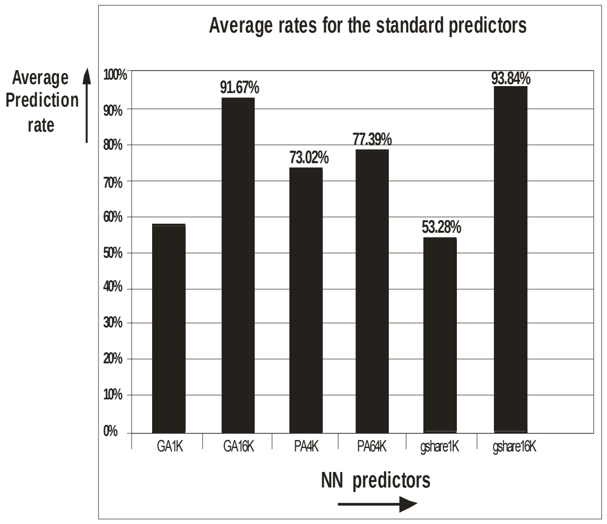


Figure 14: Average Prediction Rate for the Different Standard Predictor Configurations.

Neural Net Predictors Tested

The first part of the name is the training file used (L1, L2), the second part is the type of net (bkp or LVQ), and the third part is the number of elements in the different layers of the net.

L1 bkp40-1: back propagation network trained for 2,400,000 iterations on L1.nna

L2 bkp40-1: back propagation network trained for 800,000 iterations on L2.nna

L1 bkp40-10-1: back propagation network trained for 3,200,000 iterations on L1.nna

L2 bkp40-10-1: back propagation network trained for 3,200,000 iterations on L2.nna

L1 bkp40-20-1: back propagation network trained for 3,200,000 iterations on L1.nna

L2 bkp40-20-1: back propagation network trained for 3,200,000 iterations on L2.nna

L1 LVQ40-128-2: LVQ network trained for 5,000,000 iterations on L1.nna

L1 LVQ40-256-2: LVQ network trained for 5,000,000 iterations on L1.nna

L2 LVQ40-256-2: LVQ network trained for 5,000,000 iterations on L2.nna

Tables 4 and 5 show the prediction rates achieved by the different NN predictors. Figure 15 shows the average prediction rates over the benchmarks set for the different predictors.

Table 4: Back-propagation NNs Prediction Rates.

NN	L1 40-1	L2 40-1	L1 40-10-1	L2 40-10-1	L1 40-20-1	L2 40-20-1
Compress	0.2903	0.3196	0.2546	0.3462	0.2872	0.3260
Gcc	0.5352	0.6008	0.7169	0.7365	0.7263	0.7238
Go	0.6551	0.6843	0.8133	0.7969	0.8113	0.7904
Ijpeg	0.5890	0.9770	0.8071	0.9854	0.9177	0.9820
Li	0.5336	0.7426	0.7291	0.7913	0.7619	0.7792
M88ksim	0.5950	0.6626	0.7720	0.6683	0.7511	0.6577
Perl	0.5777	0.7111	0.7517	0.7589	0.7488	0.7457
Vortex	0.7529	0.7741	0.7981	0.8034	0.8116	0.7891
Average	**0.5661**	**0.6840**	**0.7054**	**0.7359**	**0.7270**	**0.7242**

Table 5: LVQ NNs Prediction Rates.

NN	L2 LVQ128	L1 LVQ256	L2 LVQ256
Compress	0.6642	0.6243	0.5958
Gcc	0.6019	0.7282	0.6477
Go	0.6311	0.6093	0.6089
Ijpeg	0.9221	0.6437	0.6738
Li	0.6469	0.7404	0.7115
M88ksim	0.5762	0.6080	0.6100
Perl	0.6581	0.7693	0.7233
Vortex	0.7511	0.8377	0.7621
Average	**0.6815**	**0.6951**	**0.6666**

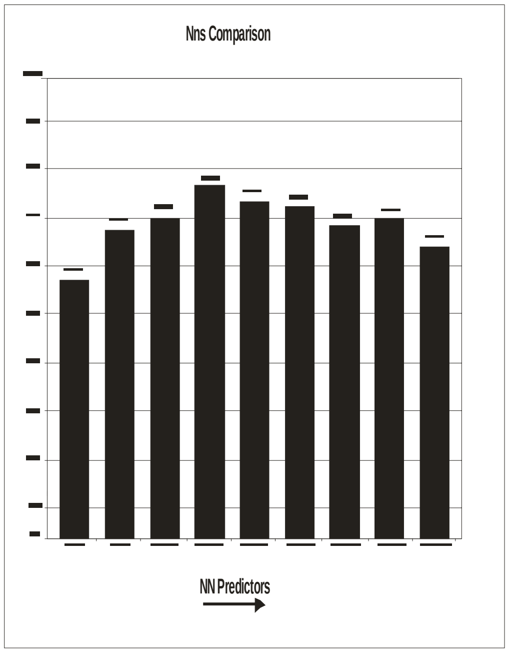


Figure 15: Average prediction rate for the NN predictors comparison

Performance Comparison

Figure 16 shows a comparison of average prediction rate for the NN predictors. The NN predictors are only compared to the small size standard predictors, not only because they were trained on traces of data created by predictors of similar size, but also because the estimated implementation size would be comparable.

There are several different ways to implement a neural network in a chip. "Implementations include digital, analog, and hybrids while the network architectures include layered networks with feed forward processing, fully interconnected recurrent networks, single layer winner-take-all networks, radial basis functions, etc. Some have on-chip learning while others may have no learning capability and only execute fast recall processing" [17]. For the type of neural networks used in this research, a hybrid/analog implementation is the most appropriate. Using this type of implementation, a processing element usually implies a small number of transistors comparable to the number than a static memory cell would require [26].

The weights of the neural network do not require additional hardware, because they are embedded in the transistor geometry. Therefore, the size of the neural network implemented in the chip would be smaller than the standard predictors used in the comparisons. Table 6 contains the prediction rate for the best NN predictors (bkp40-10-1 and bkp40-20-1) and the standard predictors; the benchmarks are sorted in the table according to their percentage of dynamic branches. Figure 17 shows the prediction rate of the best NN predictor versus the percentage of branches in the benchmark.

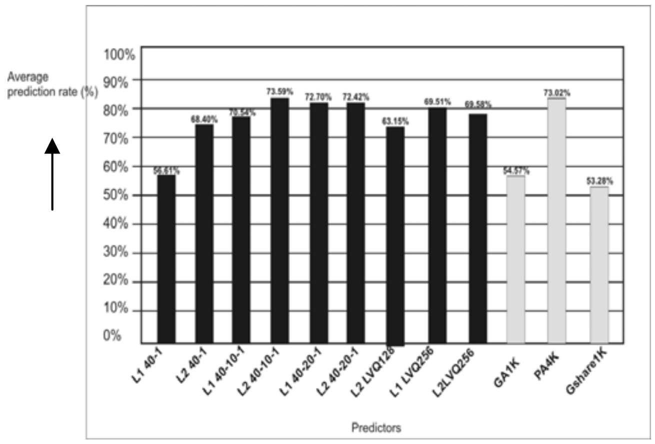
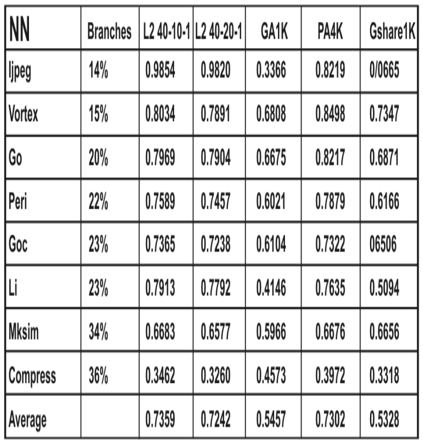


Figure 16: Average Prediction Rate of the NN Predictors.

Table 6: Best NNs and Standard Predictors Comparison.



NN	Branches	L2 40-10-1	L2 40-20-1	GA1K	PA4K	Gshare1K
Ijpeg	14%	0.9854	0.9820	0.3366	0.8219	0/0665
Vortex	15%	0.8034	0.7891	0.6808	0.8498	0.7347
Go	20%	0.7969	0.7904	0.6675	0.8217	0.6871
Peri	22%	0.7589	0.7457	0.6021	0.7879	0.6166
Goc	23%	0.7365	0.7238	0.6104	0.7322	06506
Li	23%	0.7913	0.7792	0.4146	0.7635	0.5094
Mksim	34%	0.6683	0.6577	0.5966	0.6676	0.6656
Compress	36%	0.3462	0.3260	0.4573	0.3972	0.3318
Average		0.7359	0.7242	0.5457	0.7302	0.5328

Retraining the Best NNs with Bigger Predictor Sizes

In order to make a fair comparison of the NN predictors against the standard predictors with bigger tables, it was necessary to retrain the

NNs. The best two NN configurations were retrained on a new set of data created with a new predictor configuration. The new configuration has 128 entries in the BHSR table and a PHT with 65536 entries. The eight benchmarks in the set were run with this configuration and traces of them were collected and processed. The training file was created with the L2 set that is, with the third 100,000 lines of the trace files for each benchmark. The test files were created with the second 100,000 lines of the traces.

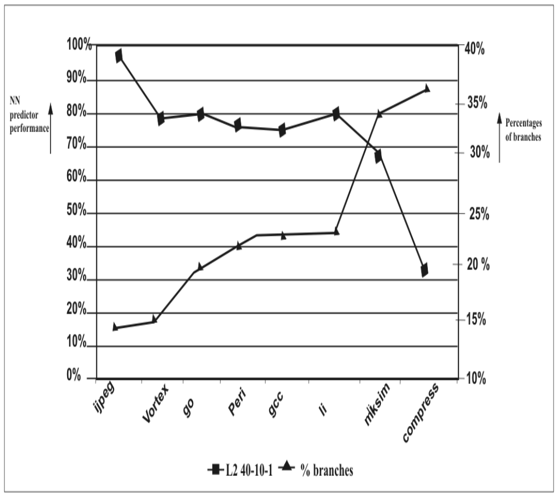


Figure 17: NN Predictor Performance vs. Percentage of Branches.

A new NN was also trained on the new training file. Since the resources used for the standard predictor changed from a few kilobytes to 64 kilobytes, this new NN was given 200 elements on the hidden layer instead of 10 or 20 as in the previous case.

T642_compress.nna	second 100,000 pairs of compress' output
T642_gcc.nna	second 100,000 pairs of gcc's output
T642_go.nna	second 100,000 pairs of go's output
T642_ijpeg.nna	second 100,000 pairs of ijpeg's output
T642_li.nna	second 100,000 pairs of li's output
T642_mksim.nna	second100,000 pairs of m88ksim's output
T642_perl.nna	second 100,000 pairs of perl's output
T642_vortex.nna	second 100,000 pairs of vortex's output
L12864.nna	800,000 pairs from the L2_ files

The NNs selected for retraining were backpropagation nets with 10 and 20 elements in the hidden layer (bkp40-10-1 and bkp40-20-1), and the new NN has 200 elements in the hidden layer. The NNs were trained for 3,200,000 iterations on the L12864.nna file. Table 7 shows the prediction rates achieved by the NN predictors. Figure 18 shows the average prediction rate over the benchmarks set.

Table 7: Prediction Rates from NeuralWorks for the Best NNs bkp.

NN	40-10-1	40-20-1	40-200-1
L. file	L12864	L12864	L12864
l.iterations	3200000	3200000	3200000
Learning	0.8500	0.7000	0.7333
t_compress	0.3589	0.3452	0.5219
t_gcc	0.7629	0.7645	0.7597
t_go	0.7331	0.7192	0.7341
t_ijpeg	0.9166	0.9164	0.9166
t_li	0.7745	0.7697	0.7870
t_mksim	0.6794	0.6864	0.6753
t_perl	0.7733	0.7730	0.7465
t_vortex	0.6898	0.6464	0.7163
Average	0.7111	0.7026	0.7322

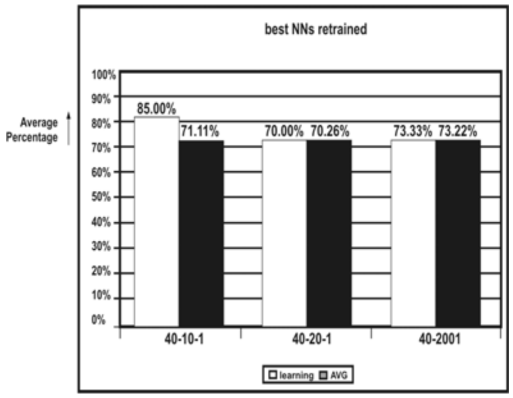


Figure 18: Average prediction rates over the benchmarks set

Then the code for these NNs was extracted and integrated into the simulator to obtain the prediction rates. The simulator was run for 1,000,000 instructions on each benchmark. Table 8 shows the prediction rates achieved by the different NN predictors. Figure 19 shows the average prediction rates NN predictors.

Table 8: Back-propagation NNs bkp Prediction Rates.

NN	40-10-1	40-20-1	40-200-1
Compress	0.3019	0.2760	0.6737
Gcc	0.7259	0.7286	0.7273
Go	0.7939	0.8057	0.7721
Ijpeg	0.9832	0.9327	0.9772
Li	0.7786	0.7685	0.8011
Mksim	0.6587	0.6753	0.6562
Peri	0.7498	0.7526	0.7436
Vortex	0.8079	0.7943	0.8143
Average	0.7250	0.7167	0.7707

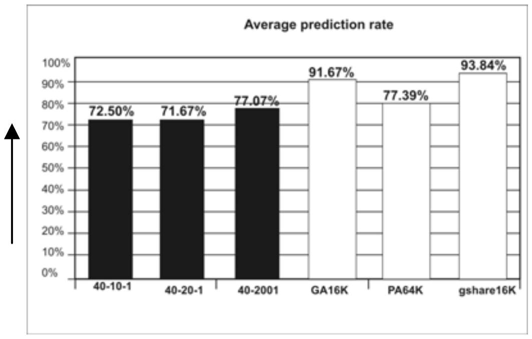


Figure 19: Average Prediction Rate of the NN and Standard Predictors Over the Benchmarks Set.

The approach taken to use the added resources was increasing the number of neurons in the hidden layer. Another approach could be adding neurons in the input and/or output layer acting as delay elements, providing this way the NN with local memory.

Results Analysis

As can be seen in the comparisons (Table 6), the best of the neural net predictors (bkp40-10-1) performs better than the other techniques, for small predictor sizes, even without an adaptive mechanism. For bigger predictor sizes, the standard techniques outperform the non-adaptive NN predictors. It can be seen in Table 6 and Figure 17 that the performance of the NN predictors decreases when the number of dynamic branches in the benchmark increases. This shows that the high percentage of branches in the benchmarks negatively affects the behaviour of the non-adaptive NN predictor.

The non-adaptive characteristic of the NN predictor affects its performance in benchmarks like compress, which has the highest number of dynamic branches as compared to the other benchmarks. The benchmark gcc when run for 1,000,000 instructions executes 228,814 conditional branches; that is, a 23% of the instructions executed are conditional branches that have to be predicted (Table 6).

Studying the output file created by the benchmark gcc, we can see that among those 228,814 conditional branches, 44,637 unique combinations of branch address and history registers are involved, that is 19.50%. In contrast, the benchmark ijpeg for 1,000,000 instructions executed executes only 142,794 (14.28%) conditional branches, with only 1,040 unique combinations (0.72%). For this benchmark, the best of the neural net predictors achieved a prediction accuracy of 98.54%, which is 20% higher than the highest achieved by current techniques.

This reinforces the idea that the next step to improve the performance of the NN predictors for all cases is not increasing the size, but adding an adaptive mechanism to them. The adaptive mechanism could be either an online refinement of the predictor's design or more support hardware to assist the NN predictor. Once the adaptive predictor is designed and tested, its implementation in hardware can be studied.

CONCLUSION

Modern microprocessors use high internal parallelization and added functionality for increasing microprocessor throughput, which puts a high demand on the provision of useful instructions to execute. Correctly predicting a branch in the instruction flow will avoid wasting clock cycles waiting for the result of a branch destination calculation to become available. This keeps the processor pipelines as busy as possible by fetching instructions from the predicted branch destination.

The processor must correctly predict the outcome of branches, because in the case that the prediction is wrong, the processor needs to discard all the instructions incorrectly fetched, issued, and executed. Even if current dynamic techniques achieve prediction accuracies in the order of 85%-90%, a misprediction rate of 10% is too high for the wide and deep pipelines present in almost all the microprocessors. Moreover, if we can increase the accuracy e.g. from 90% to 92% we are decreasing the misprediction from 10% to 8% that is a 20% improvement in the misprediction rate.

Several static and dynamic approaches have been designed and tested [3]. The approach taken in this research points to a combination of both techniques. The methodology is to let an Artificial Neural Network learn about the dynamics of programs, and particularly, conditional branch instructions, when they are being executed in a microprocessor. After studying the standard hardware-predictor configurations, two-level branch predictors were selected.

There are more complex and accurate branch predictors, but they are usually a modification of the basic structures [3]. The information available to the predictor consists of snapshots of all the registers involved in the prediction mechanism for every branch in the program, and during training, the correct outcome for the branch is provided by a perfect theoretical predictor.

All current and historical behavior are available to the NN during training, as it is provided with all the dynamic information related to the current and previous branches that other dynamic schemes utilize. The NN's task is to learn the function of the whole predictor. After training, the NN is to be implemented in the simulator to replace the standard predictor implemented in it. This is a pseudo-dynamic implementation because even if the predictor is simulated online, it is trained offline with traces of programs and it doesn't dynamically gather information from the current program being executed.

Implementation of the Neural Network Predictor and Testing

After training the neural nets, they were implemented in the simulator as the predictor. The routine called NNpredict contains NN code. It is presented with data coming from the registers inside the processor simulator and its output is supplied to the processor as the direction prediction, replacing the complete direction prediction structure in the processor simulator. Running the modified simulator, the run-time performance of the NN predictors were obtained. These values are the prediction rates achieved only for those branches that actually use the direction predictor, that is, conditional branches.

Several of the trained NNs were implemented in the simulator and tested as well as standard predictor configurations. The size of the neural network implemented in the chip would be smaller than the standard predictors used in the comparisons.

In order to make a fair comparison of the NN predictors against the standard predictors with bigger tables, it was necessary to retrain the NNs. The best two NN configurations were retrained on a new set of data created with a new predictor configuration. The prediction rates achieved by the NN predictors are lower than the ones achieved by the standard predictors, even the new NN with more elements couldn't achieve the high rates that the standard predictors achieved.

Even without an adaptive mechanism, the best of the neural net predictors performs better than the small standard predictors. For bigger predictor sizes, the standard techniques outperform the non-adaptive NN predictors. It can also be seen on the comparison that the performance of the NN predictors decreases when the number of dynamic branches in the benchmark increases. The focus of this research was evaluating the improvement that can be achieved by using a neural network to predict the outcome of branches.

Adaptive mechanism

The non-adaptive characteristic of the NN predictor used in this research affects its performance in benchmarks like gcc, which has the highest number of dynamic branches as compared to the other benchmarks, reducing dramatically the overall performance. As can be seen in Table 8 (back-propagation NNs prediction rates), the standard techniques outperform the non-adaptive NN predictors for bigger predictor sizes, showing that an adaptive mechanism is needed instead of an increase on the predictor's size. The adaptive mechanism could be either an online refinement of the predictor's design or more hardware support to assist the NN predictor.

Together with developing an adaptive mechanism, the implementation of the NN predictor in hardware has to be studied. The type of implementation chosen for the predictor will highly affect the design of the adaptive mechanism. This is one of the reasons why the present research does not go further in respect of the design of the on-line adaptive mechanism.

A design consideration for the adaptive NN predictor would be the initial state of its weight values. The weights can start with the values assigned to the offline trained NN or they can be randomly initialized. Experiments carried out in this research do not clearly show if the weights of the offline-trained NN are good initial values for a new training or not.

REFERENCES

1. Schlansker, M.S., B. Ramakrishna Rau, S. Mahlke, V. Kathail, R. Johnson, S. Anik, and S. Santosh. 1994. "Achieving High Levels of Instruction-Level Parallelism with Reduced Hardware Complexity". Technical Report HPL-96-120. Abraham Computer Research Center.

2. Calder, B., D. Grunwald, D. Lindsay, J. Martin, M. Mozer, and B. Zorn. 1995. "Corpus-based Static Branch Prediction". *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*. 79-92, June 1995.

3. Po-Yung Chang, E. Hao, and Y.N.Patt. 1995. "Alternative Implementations of Hybrid Branch Predictors". In: *Proceedings of the 28th ACM/IEEE International Symposium on Microarchitecture.* November 1995.

4. Young, C. and M.D. Smith. 1994. "Improving the Accuracy of Static Branch Prediction Using Branch Correlation". *Proc. 6th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*. 232-241. October 1994.

5. Sechrest, C., C. Lee, and T. Mudge. 1996. "Correlation and Aliasing in Dynamic Branch Predictors". In: *ISCA '96. Proceedings of the 23rd Annual International Symposium on Computer Architecture.* 22-32. May 1996.

6. Chang, P.Y., E. Hao, and Y.N. Patt. 1997. "Target Prediction for Indirect Jumps". In: *Proceedings of the 24th Annual International Symposium on Computer Architecture (ISCA 24).*

7. NeuralWorks. 1996. *Neural Computing: ATechnology Handbook for Professional II/PLUS and NeuralWorks Explorer.* NeuralWare, Inc.: Pittsburgh.

8. Hagan, M.T., H.B.Demuth, M. Beale. 1996. *Neural Network Design*. *PWS Publishing*: Boston.

9. Burger, D.C. and T.M. Austin. 1997. *TheSimpleScalar Tool Set, Version 2.0.* University of Wisconsin-Madison. Computer Sciences Technical Report #1342.

10. Barto, A.G. 1990. *Connectionist Learning for Control.* W.T. Miller, III, R.S. Sutton, and P.J. Werbos, eds. Massachusetts Institute of Technology: Boston.

11. Lendaris, G. and C. Paintz. 1997. "Training Strategies for Critic and Action Neural Networks in Dual Heuristic Programming Method". *Proceedings of International Conference on Neural Networks'97 (ICNN'97).* IEEE Press: Houston. 712-717.

12. Lendaris, G., C. Paintz, and T. Shannon. 1997. "More on Training Strategies for Critic and Action neural Networks in Dual Heuristic Programming Method". *Proceedings of Systems Man & Cybernetics Society International Conference'97.* IEEE Press: Orlando.

13. Lendaris, G. and T. Shannon. 1998. "Application Considerations for the DHP Methodology". *Proceedings of the International Joint Conference on Neural Networks'98(IJCNN'98).* IEEE Press: Anchorage.

14. Hwu, W.W., T.M. Conte, and P.P. Chang. 1989. "Comparing Software and Hardware Schemes for Reducing the Cost of Branches". In: *Proceedings of the 16th Annual International Symposium on Computer Architecture*. 224- 233.

15. Ienne, P. and G. Kuhn. 1995. "Digital Systems for Neural Networks". In: *Digital Signal Processing Technology, Volume 57 of Critical Review Series*.

P. Papamichalis and R. Kerwin, eds. SPIE Optical Engineering: Orlando. 314-45.

16. Ienne, P. 1995. "Digital Hardware Architectures for Neural Networks". *SPEEDUP Journal*. 9(1): 18-25.

17. Lindsey, C.S. and T. Lindblad. 1995. “Survey of Neural Network Hardware (Invited paper)”. *Proceedings, Application and Science of Artificial Neural Networks*, Volume 2492 part 2. S.K. Roger and D.W. Ruck. SPIE: Orlando. 1194-1294.

18. Zhou, M. and Z. Su. 1995. "A Comparative Analysis of Branch Prediction Schemes." Computer Science Division, University of California at Berkeley, Final Project. December 1995.

19. Reilly, J. 1995. "SPEC Describes SPEC95 Products And Benchmarks". Intel Corporation: Santa Clara.

20. Gloy, N., C. Young, J.B. Chen, and M.D. Smith, 1996. "An Analysis of Dynamic Branch Prediction Schemes on System Workloads". In: *Proc. of the 23rd Annual International Symposium on Computer Architecture.* May 1996.

21. Yeh, T.Y., D.T. Marr, Y.N. Patt. 1993. "Increasing the Instruction Fetch Rate via Multiple Branch Prediction and a Branch Address Cache". *The 7th ACM International Conference on Supercomputing.* July 19 - 23,1993. Tokyo, Japan. 67 – 76.

22. Evers, M., S. Patel, R.S. Chappell, and Y. Patt. 1998. "An Analysis of Correlation 99 and Predictability: What Makes Two-Level Branch Predictors Work". *International Symposium on Computer Architecture (ISCA-25).* June 1998. Barcelona, Spain. 52-61.

23. Driesen, K. and U. Holzle. 1998. “Accurate Indirect Branch Prediction”. *The 25th International Symposium on Computer Architecture*. IEEE.

24. “The SimpleScalar Architectural Research Tool Set,Version 2.0”. (software) http://www.cs.wisc.edu/~mscalar/simplescalar.html

25. Yeh, T.Y. and Y. Patt. 1993. “A Comparison of Dynamic Branch Predictors that use Two Levels of Branch History”. *The 20th Annual International Symposium on Computer Architecture*. IEEE Computer Society Press: Los Alimitos.

26. Muchnick, S.S. 1997. “Advanced Compiler Design & Implementation”. 1997.

27. Austin, T.M. and D. Burger. 1989. *The SimpleScalar Tool Set.*

28. Gallant, S.I. 1988. “Connectionist Expert Systems”. *Communication of the ACM*. 31(2):152-169.

ABOUT THE AUTHORS

P.B. Osofisan, Ph.D. obtained his B.Sc.(Eng) and M.Sc.(Eng) in Electrical Engineering from the University of Stuttgart, Stuttgart, Germany. He earned his Ph.D. in Control Systems Engineering from the same University. He then worked in a cable manufacturing plant as the Production/Quality Control Manager for over 15years, before he joined the University of Lagos as Senior Lecturer in the Electrical and Electronics Engineering Department. His research interests include the application of Fuzzy Logic Theory and Neural Network in the process control of industrial processes.

O.A. Afunlehin, B.Sc. obtained his B.Sc. (Techn.) degree from Ladoke Akintola University of Technology, Ogbomosho, Oyo State in Nigeria and has just concluded his M.Sc. (Eng) program from the University of Lagos, Lagos, Nigeria.

SUGGESTED CITATION

Osofisan, P.B. and O.A. Afunlehin. 2007. “Application of Neural Network to Improve Dynamic Branch Prediction of Superscalar Microprocessors”. *Pacific Journal of Science and Technology*. 8(1):80-97.

Pacific Journal of Science and Technology