STATIC AND DYNAMIC BRANCH PREDICTION USING NEURAL NETWORKS

Marius SBERA*, Lucian N. VINTAN, Adrian FLOREA****

* *"S.C. Consultens Informationstechnik S.R.L." Sibiu, ROMANIA, E-mail: sbmarius@usa.net*

** *"Lucian Blaga" University of Sibiu, Computer Science Department, Sibiu, ROMANIA*
E-mail: vintan@jupiter.sibiu.ro, aflorea@vectra.sibiu.ro

Abstract: In this short paper we investigated a new static branch prediction technique. The main idea of this technique is to use a large body of different programs (benchmarks) to identify and infer common C program behaviour. Then, this knowledge is used to predict new "unseen" branches belonging to new programs. The common behaviour is represented as a set of static features of branches that are mapped using a neural network to the probability that the branch will be taken. In this way the predictor does not predict a program behaviour based on previous execution of the same program or based on some program profiles but uses the knowledge gathered from other programs (knowledge experience). Also we combined static and a dynamic neural branch predictor in order to investigate how much influences the static predictor the dynamic one.

Key words: Architectures, Multiple Instruction Issue, Pipelining, Neural Branch Prediction, Neural Networks, Modeling and Simulation, Performance Evaluations

1. Introduction

Branch prediction represents the process of correctly predicting the branch's direction and target address before it is actually executed. High accuracy branch prediction is increasingly important in today's wide-issue superscalar and/or deep pipeline processor architecture. Wide-issue computer architectures rely on predictable control flow and failure; to correctly predict a branch involves delays in evacuating the instructions from the wrong path of execution entered into the pipeline structures [7,8]. Statistically was proven that conditional branches are executed about every 7-8 instructions at average. Current wide-issue architectures can execute four or more independent instructions per cycle so a branch instruction is likely to be executed every two cycles or less. This means that branch prediction is crucial for processor performance. So in the example of the Alpha 21164 processor, about 12 instructions may have to be flushed on a misprediction. This translates to a severe performance penalty. Many approaches have been proposed to branch prediction, some of which mainly involve hardware (dynamic branch prediction) while others involve software (static branch prediction). Software methods usually cooperate with hardware methods. For example, some architecture have a "likely" bit into the instruction opcode that can be set by the compiler if a branch is determined to be likely taken. Recently there is a big interest in hybrid branch prediction where it is exploited the synergism between some different branch prediction schemes. Hence, a combination of basic schemes might – at the same cost – involve serious advantages. As an example, the recent microprocessor Alpha 21264 uses a large hybrid predictor [3]. This paper investigates neural static branch prediction as proposed in [1] but it goes further and links it with a dynamic neural branch prediction as stated in [5,8].

2. Static Branch Prediction

Good static branch predictions are invaluable information for compiler optimisation or performance estimation. Typically there are two general approaches for static branch prediction: profile-based predictions and program-based predictions [8]. Profile-based predictions use program profiles to determine the certain path executed frequency. This can be extremely successful in reducing the number of instructions executed between mispredicted branches but additional work is required on the part of the programmer to

generate the program profiles. Program-based branch prediction methods attempt to rely only on a program's structure and to avoid programmer supplementary work. Some of these techniques use heuristics based on local knowledge that can be encoded in the architecture. Other techniques rely on applying heuristics based on less program structure in an effort to predict branch behavior. The method described in this paper does not rely on such heuristics. Rather than using heuristics it uses a large body of different programs to identify and infer common behavior. Then, this knowledge is used to predict some new "unseen" programs. The common behavior is represented in this work as static features of branches that are mapped using a neural network to the probability that the branch will be taken. In this way the predictor does not predict a program behavior based on previous execution of the same program but uses the knowledge gathered from other programs.

3. Useful Information for Program-based Branch Prediction

One of the first and most simple methods for branch prediction is called "backward-taken/forward-not-taken" (BTFNT). This technique relies on the statistical facts that backward branches are usually loop branches, and as such are likely to be taken. While simple, BTFNT is also quite successful. Using this technique we obtained in our experiments a static prediction accuracy rate of 57,83%. Applying other simple program-based heuristics information (branch opcode, operands, and characteristics of the branch successor blocks, knowledge about common programming idioms) can significantly improve the branch prediction accuracy over the simple BTFNT technique.

The work presented in this paper does not use any of the heuristics stated above. Instead, a body of programs is used to extract common branch behavior that can be used to predict other branches. Every branch of those programs is processed and some static information is automatically extracted. The programs are then executed and the corresponding dynamic behavior is associated with each static element. Now we have accumulated a body of knowledge about the relation between the static program elements and it's dynamic behavior. This body of knowledge can then be used at a later time to predict the behavior of instructions with similar static features from programs not yet "seen" (i.e. processed). However the prediction process is yet not complete. The static prediction is not linked with real time execution unless there is no connection with dynamic prediction (likely bits). Of course, compilers or integrated schedulers may still use static prediction alone in order to schedule the static program [9]. So we further generate such bits and use them as inputs into a dynamic predictor.

The only program elements that we are interested in are the conditional branches. Other kinds of branches are quite trivial from the direction prediction point of view. However, there are some branches that remain very interesting from the target prediction challenge. For each conditional branch in the program a set of static useful features are recorded (Table 1). Some of these features are properties of the branch instruction itself (the branch opcode, branch direction, etc.), others are properties of the previous branch, while others are properties of the basic blocks that follow on the two program paths after the current branch. We mention that characteristics no. 5, 11 and 12 belonging to Table 1, are new useful characteristics proposed by the authors of this work, completely different by those stated in [1]. Of course, finding new relevant prediction features might be a very important open problem in our opinion.

Table 1. The Static Features Set

Index	Feature Name	Feature Description
1	Branch opcode	The opcode of the branch instruction
2	Branch direction	The branch direction (forward or backward)
3	Loop header	The basic block is a loop header
4	RB type	RB is a register or an immediate constant
5	RB register index or constant	The RB register index or the LSB bits from the immediate constant
Features of the taken successor of the branch		
6	Succ. Type	The branch type from the successor basic block
7	Succ. Loopheader	The successor basic block is a loop header
8	Succ. Backedge	The edge getting to the successor is a back edge
9	Succ. Exitedge	The edge getting to the successor is a loop exit edge
10	Suc. Call	The successor basic block contains a procedure call
11	Succ. Store	The successor basic block contains at least one store instruction
12	Succ. Load	The successor basic block contains at least one load instruction
Features of the not taken successor of the branch		
13-19	As features 6 to 12	

4. The Statically Neural Predictor

Our goal into this work is to predict the branch probability for a particular branch from its associated static features. The prediction rate obtained is then used as input into a dynamic branch predictor besides other inputs. Also the static prediction will be used as an important information in the static scheduler (as an example, for the well-known trace scheduling optimization technique).

The static features of a branch are mapped to a probability that the branch will be taken. A simple way to do this is using a feed-forward neural network. The neural network uses as input a numerical vector and maps it to a probability. Here, the numerical input vector is binary coded and consists of the feature values belonging to the static set (Table 1) and the output is a scalar indicating the branch's probability.

Similarly, the same type of feed-forward neural network, but another input vector, is used to predict into the dynamic predictor. This time the input vector consists of binary representation of the branch address concatenated with the transformed prediction obtained from the static predictor (the static prediction obtained may be a binary value taken/not taken or a "percentage value"), and, possible other useful dynamic information [6]. The output is represented by one bit value ('1' for taken and '0' for not taken).

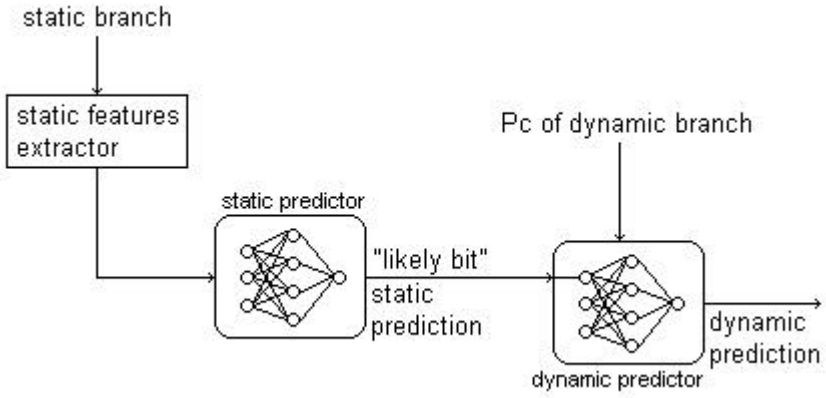

Figure 1. A general picture of the two neural predictors working together

Before this ensemble of predictors (static and dynamic) starting to will work we have to record a body of knowledge into the static predictor's neural network. So, a set of programs (benchmarks) is used to train the static neural network. The static features – automatically extracted - for every conditional branch belonging to these programs are mapped using the neural network to the probability that the branch will be taken. The output probability itself is obtained after running those programs and recording statistical information about each branch's dynamical behavior. The process is reiterated until the difference between the output of the network and the statistical information gathered drops under a certain value. After the process ends we may say that we have recorded into the neural network weights a body of knowledge about mapping static features of branches to the probability that the branch will be taken. The static predictor may now be used to predict branches belonging to new unprocessed programs and the outcome is used to influence the dynamic predictor (there are several levels of influence, see further). The outcome of the static predictor is transformed according to the following rule: scalars greater than 0.5 are mapped to '1' and less than 0.5 to '0'. This partial result is then concatenated with the branch's address and set as input into the dynamic predictor. To evaluate the influence of this "likely bit" upon the dynamic prediction it is applied in various levels (multiplied horizontally as number of occurrence, obtaining more that one "likely bit", all identical) as stated by the "Static Influence" parameter in our developed software dedicated program. The neural networks used in this work in order to map static features to a branch taken probability and to dynamically predict a branch outcome, were feed-forward kinds of networks. An artificial neural network is composed from processing units or neurons. Each processing unit outputs a value known as its activity. Connections or weights links processing units are indicating the direction of activity flow. The connection pattern that links the units separates one type of neural network from another. Basically there are two types of neural nets: feed-forward nets (which have no loops) and recurrent nets (in which loops occurs because of feedback connections). Both neural networks used during this work falls into the first category, being thus feed-forward nets. In this kind of networks the activity flows from input to output. The activities of the hidden layer, denoted h_i, and respectively of the output layer y_k, are computed based on the very well-known formulas, as [2]:

$$h_j = f(\sum_i w_{ij} \cdot x_i + a_j) \qquad \text{eqn (1)}$$

and

$$y_k = f(\sum_j v_{jk} \cdot h_j + b_k), \qquad \text{eqn (2)}$$

The activation function used into this work is a fairly sigmoid standard one:

$$f = \frac{1}{1+e^{-u}} \qquad \text{eqn (3)}$$

The adaptive behavior of the neural network is achieved by setting its parameters, the weights w_{ij} and v_{jk} and biases a_j and b_k. The well-known back propagation algorithm based on the square root error (E) computation sets these parameters:

$$E = \sum_k (y^k - t^k)^2 \qquad \text{eqn (4)}$$

5. Experimental Methodology and Obtained Results

To quantify the performance of this static prediction scheme, during our experiments, we have used a set of 8 programs called globally the Stanford HSA (Hatfield Superscalar Architecture) benchmarks suite. The benchmark's HSA assembler sources were used for the static analyze and static feature extraction while the statistics of the dynamic execution were extracted from the traces of the same benchmarks. These traces were obtained by running the Stanford benchmarks into an instruction-level simulator special dedicated to the HSA processor. The HSA is a high-performance multiple-instruction-issue architecture developed to exploit instruction-level parallelism through static instruction scheduling [6,7].

To collect a body of knowledge only 7 (of the total of 8) programs are used. From the sources of these 7 programs the static feature sets are automatically extracted and a statistical evaluation is performed on the corresponding traces. Then, the neural net belonging to the static predictor is used to map these static feature sets to the probability that the branch will be taken or not taken. This probability is extracted from the statistical information earlier gathered. Now taking into account that the body of needed knowledge is available, we may proceed in predicting programs yet "unseen" by our neural static predictor. In order to do this, we used the eighth program from the HSA benchmarks suite, left out until now, which we further call it the test benchmark. Because every program from this suite covers just a particular domain (puzzle games, recursion programs, matrix problems, etc.), each experiment is executed 8-th times and every time is left out another program (as a test program). Finally an average of the eight results is computed. The neural net's configuration presented in the results charts are represented as: [number of input units] *x* [number of hidden units] *x* [number of output units]. We have performed two distinct kinds of experiments. In the first one we predicted using just the first (static) neural network. The static prediction extracted from this first neural network output is compared with the perfect static prediction. By "perfect static prediction" we actually mean the asymptotic limit to which we could target out the maximum static prediction accuracy, considering an ideal "*Oracle*" processing model. In order to achieve this we considered the dynamic trace from which we extracted statistical information and for each benchmark we calculated its associated perfect static prediction (P.S.P.) using our following formula:

$$P.S.P. = \frac{\sum_{k=1}^{L} \frac{MBR_k^{\,2}}{NBR_k}}{\sum_{k=1}^{L} NBR_k} \quad \text{eqn (5),}$$

where:

NBR_k – total number of dynamic instances of a branch belonging to a certain benchmark
$NBR_k = NT_k$ (no. of taken instances for the k branch) + NNT_k (no. of not taken instances for the k branch)
$MBR_k = Max(NT_k, NNT_k)$
L = total number of static branches belonging to a certain benchmark

Now we will define the (imperfect) static prediction (S.P.) as the following formula:

$$S.P. = \frac{\sum_{k=1}^{L} \left(\frac{NT_k^{\,2}}{NBR_k} * R_k + \frac{NNT_k^{\,2}}{NBR_k} * (1 - R_k) \right)}{\sum_{k=1}^{L} NBR_k} \quad \text{eqn (6),}$$

where:

R_k – the output of the static neural network predictor for the branch k (1 – taken; 0-not taken). It's mathematical obviously that: $P.S.P. \geq S.P.$ eqn (7)

Prediction results obtained from the hybrid predictor (static + dynamic) are computed using the same formulas but while in formula (5) the input indices (MBR_k) are "coming" from trace statistics, in this case these indices are gathered from the output of the first static neural network (R_k). Also, the branch's PC constitutes an input parameter for the neural dynamic branch predictor.

For the first type of experiments we have considered two ways of representing the output: as binary output (1 = taken, 0 = not taken) and as percentages of branch taken behavior (measured probability). Figure 2 and 3 presents the static prediction accuracies (P.S.P. and S.P.) according to these previous described two distinct conditions.

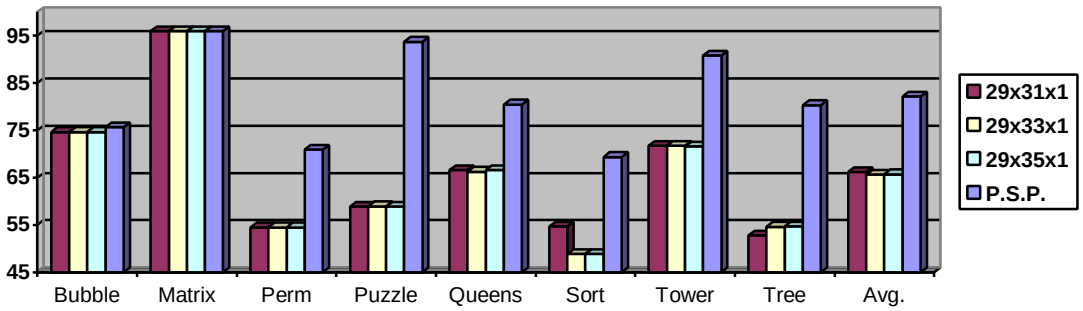

Figure 2. Static prediction results with binary output

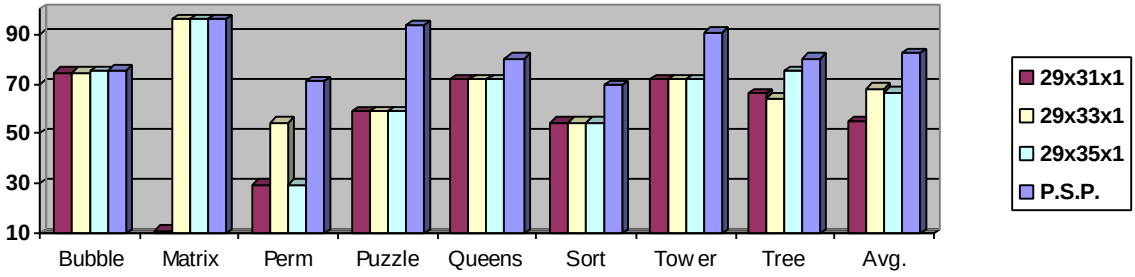

Figure 3. Static prediction results with percentage output

The second type of experiments consists in measuring the Static Influence (SI) of the static neural predictor regarded to the dynamic neural predictor's behavior. More precisely, we added supplementary outputs belonging to the static network as entries for the dynamic neural predictor. The number of these supplementary inputs – having each of them the same associated value (probability) - is noted here with SI. The influence of SI parameter is presented in Figure 4 (SI=2 and 4). Related to Figure 4, it means that the dynamic predictor's totally number of input cells is (8+SI) respectively (10+SI). A bigger number of inputs units derived from the static predictor means, theoretically, a bigger influence related to the neural dynamic predictor's accuracy. The neural network configurations are represented as:

- the configuration for the static net as: [number of input units] *x* [number of hidden units] *x* [number of output units]
- the configuration for the dynamic net as: [number of input units] *x* [number of hidden units] *x* [1 output unit]

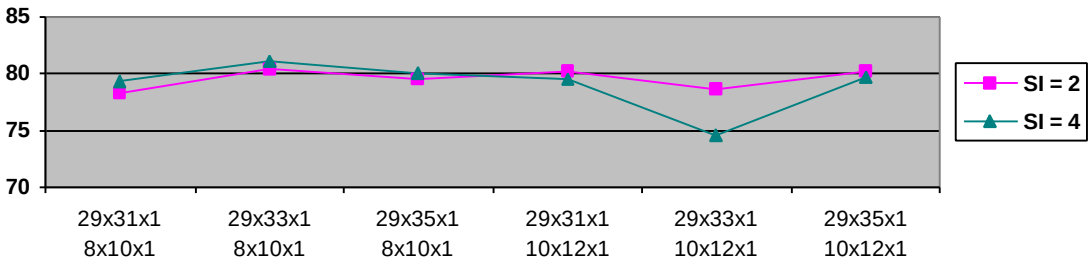

Figure 4. Hybrid (static + dynamic) prediction results influenced by a binary static prediction resolution

6. Conclusions and Further Work

The method investigated through this paper has the advantages over the existing static prediction methods that rather than being based on heuristics it is based on general program structures and behaviour (experience). Another advantage is that no supplementary time or programmer intervention is required to generate profiling information.

Our simulations focused on the neural static prediction (figures 2 and 3) suggests that predicting binary results (taken / not taken) is less efficient than generating percentage results (prediction accuracies). The best average result performed for the binary output static prediction was at average 66,21% while with

percentage output it reached 68.32%. Both these static prediction results must be compared with the perfect static prediction that was about 82.16%. The neural net's percentage outputs, even if are harder to be assimilated in the learning phase, contains more information than a simple binary prediction. Also based on our simulations the influence (through simulated likely bits) of static predictor upon a neural dynamic predictor is minimal. The best results obtained were 80.34% for a SI=2 and respectively 81.1% for SI=4, both using the same hybrid configuration (29x33x1, 8x10x1). Figure 4 exposes that as static influence grows for smaller neural nets configurations the result is also positively influenced, but on bigger neural nets the static influence becomes negative by expanding too much the input layer of the net and being harder to assimilate by the neural net.

As an immediately further work we are intending to develop also other neural nets topologies, including recurrent neural nets (NN). These NN contain connections from the hidden cells to the context cells in order to store the state of the net on the previous step of time. Some recent research prove that some recurrent nets with asymmetric connections or partially recurrent nets, perform sequence recognition far better than other (simple recurrent) NN. Also we intend to use the SPEC'2000 benchmarks that is known that are more predictable than our used benchmarks.

Acknowledgments

This work was partially supported by two research grants (CNCSIS 8/2001 and ANSTI CG 12/05.06.2001) offered by Romanian Ministry of Education and Research (M.E.C.). Our gratitude to Professor Gordon B. Steven and Dr. Colin Egan from the University of Hertfordshire, England, for providing HSA Stanford benchmarks and for their useful helps. Also we like to thank Professor Theo Ungerer from Karlsruhe University, Germany, for a very useful professional discussion in January 2001.

References

[1] Calder, B., Grunwald, D., and Lindsay, D. - *Corpus-based Static Branch Prediction*, SIGPLAN Notices, June 1995, pp79-92

[2] S.I. Gallant - *Neural Networks and Expert Systems*, MIT Press. 1993.

[3] D. Grunwald, D. Lindsay, B. Zorn – *Static Methods in Hybrid Branch Prediction*, Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT), 1998

[4] M. Sbera – *MSc thesis: Some Contributions to Static and Dynamic Branch Prediction Challenge*, University "L. Blaga", Sibiu, Romania, July 2001

[5] M. Sbera – BSc thesis: *A Quantitative Estimation upon Dynamic Neural Branch Prediction*, University "L. Blaga", Sibiu, Romania, June 2000

[6] G. Steven, C. Egan, L. Vintan – *Dynamic Branch Prediction using Neural Networks*, Proceedings of International Euromicro Conference DSD '2001, Warsaw, Poland, September, 2001

[7] G. B. Steven, B. Christianson, R. Collins, R. Potter, and F. Steven – *A Superscalar Architecture to Exploit Instruction Level Parallelism*, Microprocessors and Microsystems, Vol.20, No 7, March 1997, pp.391-400.

[8] L. N. Vintan – *Instruction Level Parallel Processors*, Romanian Academy Publishing House, Bucharest, 2000 (264 pp., in Romanian)

[9] W. Wong – *Source Level Static Branch Prediction*, The Computer Journal, vol, 42, No,2, 1999