Simulating a simple neural network on branch prediction

Vinicius Marra Ribas, Maurício Fernandes Figueiredo and Ronaldo Augusto de Lara Gonçalves*

Departamento de Informática, Universidade Estadual de Maringá, Av. Colombo, 5790, 87020-900, Maringá, Paraná, Brasil.
**Autor para correspondência. e-mail: ronaldo@din.uem.br*

ABSTRACT. This work evaluates the usability of perceptron and its efficiency on branch prediction in superscalar architectures, using two simulators. Firstly, we simulated the use of perceptron on the classification of colored points on the Cartesian plane. The classifier showed that perceptron could be used on branch prediction, because the action of classifying is similar to the action of predicting. Based on this previous analysis, we also simulated the use of perceptron on branch prediction using branch traces automatically generated. The predictor presented satisfactory results. In all cases, we concluded that simulation is a good strategy to measure performance of branch predictor based on perceptron before its implementation in hardware.

Key words: superscalar processors, branch prediction, perceptron.

RESUMO. Simulando uma rede neural simples na previsão de desvios. Este trabalho avalia a usabilidade do perceptron e sua eficiência na previsão de desvios em arquiteturas superescalares, usando dois simuladores. Primeiro, nós simulamos o uso do perceptron na classificação de pontos coloridos sobre o plano Cartesiano. O classificador mostrou que o perceptron poderia ser usado na previsão de desvios pois a ação de classificar é similar a ação de prever. Com base nesta análise prévia, nós também simulamos o uso do perceptron na previsão de desvios usando traços de desvios gerados automaticamente. O previsor apresentou resultados satisfatórios. Em todos os casos, nós concluímos que a simulação é uma boa estratégia para medir o desempenho de previsores de desvios baseados no perceptron, antes de sua implementação em hardware.

Palavras-chave: processadores superescalares, previsão de desvios, perceptron.

Introduction

Many commercial processors are implemented on superscalar architectures, like *Pentium* (Intel), *Atlon* (AMD), *UltraSparc* (SUN), *PowerPC 7455* (Motorola), *Alpha 21264* (DEC) and others. Branch prediction is a fundamental strategy used in these processors, to predict the probable path of next instructions to be fetched, anticipating the execution of instructions and providing high instruction level parallelism.

Techniques for branch prediction predict the branch direction (taken or not-taken) as well as the target address. They can be static or dynamic, if they always choose a predefined direction (Smith and Sohi, 1995) or use hierarchical tables containing branch history (Bray and Flynn, 1991; Lee and Smith, 1984; Yeh and Patt, 1991), respectively. Recently, interesting results were obtained from studies of branch prediction based on neural networks (Calder *et al*., 1997; Jiménez and Lin, 2000, 2001). In this context, some works adopt a single neuron perceptron, which is able to classify linearly separable areas and predict time series (Koskela *et al.*, 1996; Cholewo and Zurada, 1997; Thiesing *et al.,* 1997). Being a recent area of study, it requires further investigation.

The main objective of this work is to simulate and evaluate the use of perceptron on branch prediction. Firstly, we simulated the use of perceptron in the colored point classification, in order to understand its behavior. After that, we simulated a modified version of the perceptron predictor kernel proposed by Jiménez (Jiménez and Lin, 2000, 2001).

This paper is organized as follows: Section 2 presents concepts and techniques about branch prediction. Fundamentals of neural networks and a brief description of perceptron are addressed in Section 3. Section 4 describes a colored point classification tool based on perceptron. Section 5 describes a branch predictor simulator also based on

perceptron. Section 6 shows the analysis of results for branch prediction. Section 7 summarizes main conclusions and proposes future works. Finally, the references used in this work appear in the last section.

Branch prediction

In superscalar architecture, branch instructions may reduce the parallelism because the branch direction or the target address during the instruction fetch cannot be known. Branch predictors may predict the branch direction, as well as the target address, avoiding the interruption of instruction stream inside pipeline and anticipating the fetch of the probable path. Thus, it is an essential mechanism to ensure high performance for superscalar processors, providing continuous instruction stream and expanding the possibilities to detect parallel instructions (Gonçalves *et al.*, 2001).

Smith (1981) classified branch prediction techniques in static prediction (prefixed solution by hardware or software) and dynamic prediction (solved at run time according to the history of past branches). Regarding static prediction, there are techniques which always predict either taken or not-taken for any branches, techniques which always predict taken for some types of branches (defined by operation code) and not-taken for others, or still, techniques which always predict taken only for backward branches and not-taken for forward branches.

Dynamic prediction techniques are more complex ones. A well-known dynamic technique uses a table to keep *n* most recently not-taken branches. If a branch is matched in this table, it is predicted as not taken; otherwise, it is predicted as taken. Another technique predicts a branch in accordance with its last execution. In this case, a table is used to keep a 1-bit history for each branch instruction (1 for taken branch and 0 for not-taken branch). This technique can be improved keeping the last *n* occurrences of each branch using n-bit history. A different version of this technique uses 2-complement counters instead of a bit history. In this case, a branch is predicted taken when the signal bit of the respective counter is 0 and not taken when it is 1. The counter is updated after each branch solution.

Nowadays, the most used techniques for branch prediction, providing better results, make use of a specific cache named *Branch Target Buffer* (BTB). Several proposals for BTBs are discussed in (Lee and Smith, 1984; Perleberg and Smith, 1993). The BTB is organized as a table (see Figure 1), where each line consists of information to identify branches (normally their addresses) and to supply predictions (normally counters or histories), target addresses for taken branches and probably some target instructions.

branch address	branch history	branch target	first target instructions
address #1	history #1	target #1	bytes #1
address #2	history #2	target #2	bytes #2
address #3	history #3	target #3	bytes #3
: :	: :	: :	: :
address #n	history #n	target #n	bytes #n

Figure 1. Branch target buffer (BTB).

When a branch instruction is fetched, its address is compared with branch addresses stored in the BTB. If the address is stored there, the branch is predicted using the respective branch history field. Moreover, if the target address is not known yet, it is obtained from the branch target field. If the branch is predicted taken, the processor gets the first target instructions from the appropriated field of the BTB, while the fetch is redirected to the target address (jumping those first instructions). If the branch is predicted not taken, the processor continues fetching on the natural path of the program.

After the branch solution, the predicted outcome is compared with the branch real outcome and both branch direction and target address are verified. If the prediction was predicted correctly, the processor follows its normal execution; if not, the pipeline is flushed and the processor starts to fetch from the right path. Also, the prediction field and/or target address of BTB are updated.

Yeh and Patt (1991) proposed a new organization for BTB, which uses more than one level of history and the predictions are based on the individual branch behavior, as well as on the relationship among them. That prediction technique was named Two-Level Adaptive Training and has three basic variations: GAg, PAg and PAp.

GAg has a two-level structure. In the first level, there is a Global Branch History Register (GBHR), which is a shift register containing *n* outcomes from *n* last branches. After each branch solution, the register is shifted in order to insert a new bit, producing a new situation inside that register. However, the same situation can appear several times, generating different patterns. In the second level, there is a Global Pattern History Table (GPHT), which is indexed by different patterns

provided by GBHR. Each line in GPHT contains the *s* last occurrences of the same pattern, which can be generated by different branch instructions. The prediction is based on the history of the second level, which is indexed by the pattern from the first level. Consequently, the behavior of a branch can influence other branches. Moreover, during the next prediction for the same branch, the pattern can be changed.

PAg and PAp are extended models. In both cases, the first level uses a Per-Address Branch History Table (PBHT) rather than a unique global register. In this table, each different branch instruction addresses an individual pattern register. However, in the second level, PAp and PAg are different because PAp has several GPHTs rather than one, where each branch instruction addresses its own pattern table on the second level; this is named Per-Address Pattern History Tables (PPHT). All models are sketched in Figure 2. In fact, there are other variations (Yeh and Patt, 1992).

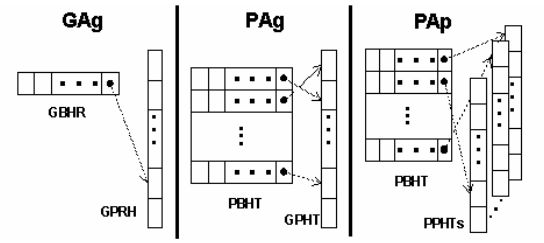

Figure 2. Variations of two-level adaptive BTB.

Nowadays, new approaches for branch prediction based on neural networks (Calder *et al*., 1997; Jiménez and Lin, 2000, 2001) have been investigated. Simple models, like perceptron, have been analyzed due to their capabilities to support long streams of branch histories without requiring additional computing or additional hardware resources. More details about those techniques are described in the next section.

Neural networks

The utilization of neural networks in several fields of human knowledge has grown significantly, due to their singular features, like learning, classification, prediction, optimization, approximation and others (Haykin, 1994). Jain and Mao (1996) said that neural network applicability is vast and can resolve many computational problems, like time series prediction. Branch prediction fits in this context, whose problem is similar to predict the next element in a sequence of 0s and 1s (taken or not-taken).

The fundamental component of neural networks is the artificial neuron, which is modeled from the biological neuron. The neuron receives n inputs X_i ($1 \leq i \leq n$). They are weighted by synaptic weights W_j ($1 \leq j \leq n$), which are associated to the learning capabilities of the neuron. The output y is defined by an activation function f, which operates on both the sum of those weighed inputs and an activate threshold represented by a weight W_0. Mathematically, the neuron is represented by the equation bellow:

$$y = f(W_0 + \sum_{j=1}^{n} W_j X_j) \qquad \text{(E1)}$$

Neural networks can be organized under different models, depending on the number of layers, number of neurons in each layer, number of inputs and outputs, neuron connection types, synaptic weights, activation function and threshold. The main feature of neural networks is its learning capability, which is done by adjusting the synaptic weights, using results obtained either from real-time execution or from a training period (Figueiredo and Gomide, 1999). The learning techniques are classified in four basic types (Jain and Mao, 1996): Error Correction, Boltzmann learning, Hebb learning and Competition learning.

The basic principle of the Error Correction learning is to use neuron output error (the difference between desired output and neuron output) to adjust the synaptic weights, in order to avoid the same error later. The purpose of Boltzmann learning is adjusting the synaptic weights in order to generate neuron states according to some probability distribution. In Hebb learning, the intensity of synaptic connections is changed, according to output line correlations between pre and pos synaptic neurons (Jain and Mao, 1996). In Competition learning, neurons dispute activation among themselves. The winner establishes which neurons adjust their weights.

Perceptron is one of the first implemented models for artificial neuron. Originally imagined by Rosemblatt, its activation function is $y = f(\sum)$, such that $y = -1$ when $\sum \leq 0$ and $y = +1$ when $\sum > 0$. Figure 3 represents this model.

The perceptron is able to classify elements (Russell and Norvig, 1985). Hence, if the set of all possible inputs of perceptron is a n-dimensional space ruled by equation E1, there is an hyperspace (line, if $n=2$) that separates the space in a set of inputs for which perceptron returns -1 and a set of inputs it returns +1.

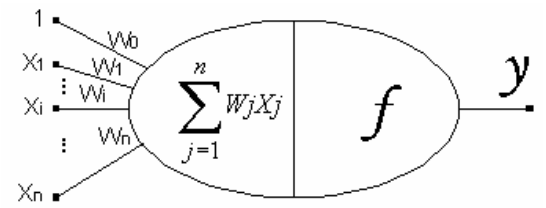

Figure 3. Model of perceptron neuron.

Rosemblatt showed that, when the learning pattern submitted to perceptron has linearly separable classes, the learning process converges to a finite number of iterations, adjusting the synaptic weights and the activation threshold of neuron (Jain and Mao, 1996; Jiménez and Lin, 2001). This effect is called Perceptron Convergence Theorem. The perceptron learning process uses an error correction technique where the initial synaptic weights are values randomly generated, in the range [-0.5, +0.5].

According to Jain and Mao (Jain and Mao, 1996), the applicability of neural networks is wide and it can solve many computational problems, like the ones related to classifying and prediction of time series. Branch prediction appears in this context, because to predict the next branch is similar to predict the next element of time series composed of 1s and 0s (taken and not-taken). Thus, two simulators based on perceptron were developed: a colored point classification tool and a branch predictor, discussed in the next sections.

Classification simulator

This simulator, called classifier, was developed just to provide information about the behavior of the perceptron neuron in a classification process. Our interest was to investigate the possibility of using perceptron in branch prediction. The simulator was developed using Borland Delphi 5.0 Tool on Windows platform and its interface is shown in Figure 4.

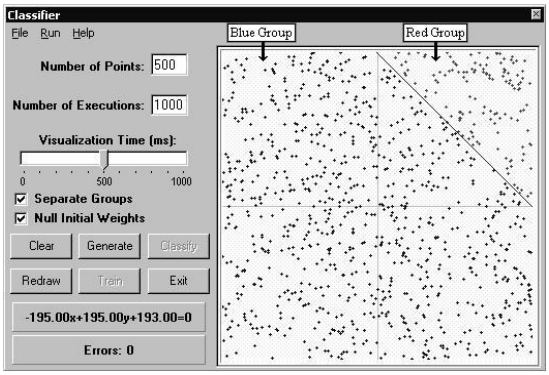

Figure 4. Classifier user interface.

Using the "*generate*" button, the classifier allows random generation of points in the Cartesian plane. This point set is called *global point set*. Each point is represented by a coordinate (x, y) in the interval [-1,+1] and by one of two colors (red or blue). Depending on the configuration, the global point set can be generated in two color groups or with a well-defined linear separation, either mixed (no separation) in the plane. Figure 4 shows an example of the first case. However, the perceptron neuron does not know the configuration and tries to find the boundary between the red and blue groups.

After point generation, the perceptron can execute two phases: training (for learning) and classification. Clicking on the "*train*" button starts the training. The "*number of points*" field adjusts the number of training points (500 is default). The initial synaptic weights can be adjusted to zero by checking the "null *initiate weights*" box, otherwise they will be chosen randomly in the interval [0,+1]. The *training point set* is a subset of the global point set. The perceptron trains either until a solution is found (boundary) or until executing the maximum number of iterations, defined by the "*number of executions*" field (1,000 is default).

In any training iteration, the perceptron analyzes the whole training point set and for each point it makes a supposition about its color. After analyzing all training points, even if only a point is wrong, the synaptic weights will be recalculated (learning) and a new iteration will start. The aim of the perceptron training is to find, dynamically, the better synaptic weights for that point set. Thus, the perceptron will be able to classify another point set with the same characteristics (same boundary).

During the training phase, all learning actions of the perceptron are graphically animated. The "*visualization time*" field adjusts the animation speed. After the training, the user interface will show the linear equation required to separate the two point groups in the best possible way. Clicking on "*classify*" button starts the next phase.

During the classification phase, the *classification point set* (difference between the global point set and the training point set) is submitted to the perceptron, but no information about the colors is provided. The perceptron tries to predict the color for each point and find the boundary, using the synaptic weights adjusted during the training phase. The results are shown in the user interface. The points correctly classified are plotted using the group color and all wrong points are plotted using another color. The interface shows the total number of errors.

The current simulation (global point set and its respective weights obtained during its training phase) can be stored in a text file. This information can be recovered by the "*File*" option in the menu bar. This feature allows the reevaluation of any specific situation, as well as the execution of an exhaustive simulation. The exhaustive simulation provides statistics about the classification accuracy for different sizes of training point set (default is from 2 to 500 points). For each variation, the global point set is classified and the accuracy rate is plotted in a graph. We can notice there is a little instability when the perceptron uses a few points for training. That situation happens because the perceptron is not able to find the appropriated linear equation.

Figures 5 and 6 show the perceptron engine and learning codes, respectively.

```
procedure PerceptronOutput(PointIndex: integer);
var in : integer; Result : real;
begin
  Result := Weights[0];
  for in := 1 to InputNumber do
     Result := Result + Points[PointIndex].Input[in]
                                    * Weights[in];
  if Result <= 0
    then Results[PointIndex] := -1
    else Results[PointIndex] := +1;
end;
```

Figure 5. Perceptron output code.

```
procedure PerceptronLearning();
var en, in : integer; Result : real;
begin
  for en := 1 to ElementNumber do
      begin
        Result := GainStep * (1 - Points[en].Color
                * Results[en]) * Points[en].Color;
        Weights[0] := Weights[0] + Result;
        for in := 1 to InputNumber do
            begin
              Result := GainStep * (1 - Points[en].Color
                      * Results[en]) * Points[en].Color;
              Weights[in] := Weights[in] + Result
                           * Points[en].Input[in];
            end;
      end;
end;
```

Figure 6. Perceptron learning code.

Our experiments showed that perceptron neuron is able to classify Cartesian elements in one of two possible groups, if they are linearly separable. In a sense, the perceptron could be used to predict branches due to two reasons. First, the branch prediction tries to classify branches in one of two possible groups: taken or not-taken branches, such as the classification. Second, both point classification and branch prediction consider previous actions to guess the next one. Thus, the classifier was modified in order to make our predictor, which is explained in the next section.

Prediction simulator

Our branch prediction simulator, also called predictor, uses a prediction code kernel developed by Jimenéz and Lin (Jiménez and Lin, 2001). That kernel was adapted and extended using Borland C^{++}Builder 4 Tool for Windows. Figure 7 shows the predictor interface. It works as follows: the simulator generates a branch stream randomly, containing a sequence of 0s and 1s, which simulates a trace of branch outcomes such as generated by execution of one real application. In this stream, 1 represents "taken" and 0 "not-taken".

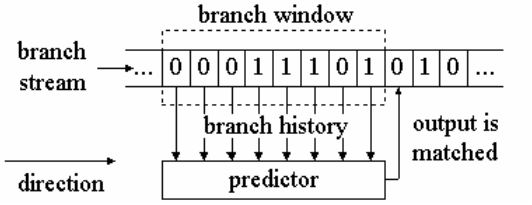

Figure 7. Predictor execution model.

The simulator scans the branch stream using a history window. For each new prediction, this window is moved on the stream, bit after bit, and the simulator extracts the branch history from it, which is used as perceptron inputs. The predictor uses the branch history to predict the next branch outcome. In other words, the branch stream simulates the branch history being modified in run-time, which is shown in Figure 7.

The branch stream is generated according to the configuration specified in the simulator interface. Users can define stream size, history (input) size, execution type, execution number, distribution and other parameters. The stream size is defined by "*i-number*" option. The perceptron neuron used in the predictor works with different input sizes (branch history), such as shown in the fields of "*history size*" option. These sizes also define the prediction window sizes.

There are two possibilities to generate a branch

stream, depending on the "*distribution*" option to be random or pattern. In the first case, the whole stream is randomly generated, with no relationship between bits. As a matter of fact, users can define the occurrence probability for taken branches in the "*taken prediction*" option. In the second case, the stream is composed of a same random pattern repetition, separated by zero segments. The distance between zero segments and the size of each zero segment are defined by "*spacing*" and "*repetition*" options, respectively.

In addition, the simulator allows exhaustive execution. In this case, the perceptron works on a branch stream several times (until 1 million of repetitions in the current version, according to "*iterations*" options). This feature provides more reliable statistics. That branch stream can be the same or a newer one in each repetition, depending on the "*instruction set*" option to be adjusted for "*fixed*" or "*variable*". Notice that, using the second option, it is hard to evaluate the ability of the perceptron, since the synaptic weights obtained in one previous stream doesn't satisfy the next one.

Usually, the initial synaptic weights for the current repetition are the final synaptic weights obtained in the previous repetition, for either fixed or variable option. However, being ticked the "*fixed learni*ng" box, the perceptron uses the same synaptic weights between repetitions. When this option is not ticked, the synaptic weights are reinitiated to zeroes between repetitions.

Pressing the "*go*" button starts the simulation. The errors, execution steps and statistic results of the predictor are presented in the text window. Results can be shown in short mode (only final statistics) or partial mode (partial statistics between iterations) depending on "*short presentation*" box to be ticked or non-ticked, respectively.

Some other functions could be noticed. The "*clear*" button stops the simulation (if necessary) and cleans the text window. "*Save*" button writes the simulation results in a text file. Ticking on the *"combined execution"* box allows the complete execution of all possible configuration options and writes the results in a text file. Next section presents a performance evaluation of its branch prediction based on this predictor.

Performance evaluation

According to Smith (1981), when the prediction policy is "*always taken*", the prediction accuracy rate has as value the percentage of branches really taken. For example, considering a branch stream containing 80% of taken branches the accuracy rate will be 80%. For many benchmarks, the number of taken branches is bigger than not-taken branches and that policy reaches an average rate of about 76.68%. We consider this rate as referential parameter for evaluating our predictor.

In our simulations, we have proven that, if the perceptron does not learn (disabled learning), its prediction accuracy is lower. That is obvious, because if there is no learning, the perceptron makes the same mistakes, independently of the number of repetitions. Besides that, for a branch stream composed only of taken or not-taken branches, the accuracy rate is 100%. In this case, perceptron starts making a few mistakes and quickly starts to get its predictions right.

If the branch stream is composed of randomly generated branches and the number of taken branches is larger than the number of not-taken branches (or conversely), the performance of our predictor is similar to that policy *"always taken"* (or "*always not-taken*" in the opposite case).

Anyway, the performance of perceptron is never smaller than 50%, even in the worst case (totally random situation, in which the number of taken and not-taken branches is evenly distributed - 50% for both). This behavior is understandable, since the perceptron tries to find a branch pattern in the inputs, which means a classification function. That is hard in a random group.

However, it is worthwhile to notice that the accuracy rate of perceptron increases according to largest percentage occurrence between taken and not-taken branches and therefore, it is never lower than 50%. Besides, it has always a growing behavior when the number of taken branches grows up to 100% or decreases up to 0%.

Another important remark is that, using streams containing sequences of a same branch pattern, our predictor got a very satisfactory performance, reaching more than 97% of accuracy rate. Even without learning, the performance reaches above 95%. If the branch pattern is variable for each stream, the performance reaches more than 87% if learning is considered; otherwise, the perceptron performance reaches more than 85%. The average results can be seen in Tables 1 and 2.

Table 1. Accuracy for taken/not-taken branches.

Branch Distribution Stream	Accuracy
100% Taken	100%
100% Not-Taken	100%

Table 2. Accuracy for random branches.

Input Type	Branch Distribution Stream			
	Random		Pattern Repetition	
	NL	WL	NL	WL
Fixed	75%	75%	95%	97%
Variable	75%	75%	85%	87%

NL (No Learning) - WL (With Learning)

Conclusion and future works

In this work, a classification tool to understand the perceptron behavior is developed. Simulations with this tool showed that the perceptron could be used to predict branch instructions in superscalar architectures. Based on that preliminary evaluation, we built a branch prediction simulator using only one perceptron neuron. Our simulations presented satisfactory results.

Using streams containing sequences of a same branch pattern, our predictor reaches more than 97% of accuracy rate. Even without learning, the performance reaches above 95%. If the branch pattern is variable for each stream, the performance reaches more than 87% if considering learning; otherwise the perceptron performance reaches more than 85%. For values randomly generated, our predictor presented accuracy around 75%, equivalent that one reached by "always taken/not-taken" policy. These results show moderate efficiency of the perceptron on branch prediction using non-real inputs. It may be concluded that simulation is a good strategy to measure the performance of a branch predictor based on perceptron before its implementation in hardware.

For future works, we intend to analyze the performance of perceptron on real benchmarks. In this case, we will implement our predictor inside the *SimpleScalar Tool Set* (Burger and Austin, 1997), in order to measure the effects of the speculative execution on prediction accuracy. We also will work on two level prediction models, using more than one perceptron to improve performance.

References

BRAY, B. K.; FLYNN, M. J. *Strategies for branch target buffers*. ACM - Association for Computing Machinery, Jun. 1991, *Proceedings...* Annual International. p. 42-50.

BURGER, D.; AUSTIN, T. M. *The SimpleScalar tool set, Version 2.0*. TR #1342. Madison: Computer Sciences Department, University of Wisconsin-Madison, 1997.

CALDER, B. *et al*. *Evidence-based static branch prediction using machine learning*. Maryland: ACM Transactions on Programming Languages and Systems, 1997.

CHOLEWO, T.J.; ZURADA, J.M. Sequential Networks Construction for Time Series Prediction. In: IEEE JOINT CONFERENCE ON NEURAL NETWORKS, COMPUTER, Houston, Jun. 1997. *Proceedings...* Houston, 1997. p. 2034-2039.

FIGUEIREDO, M.; GOMIDE, F. Design of fuzzy systems using neurofuzzy networks. *IEEE Transactions on Neural Networks*, New York, v. 10, no. 4, p.815-827, 1999.

GONÇALVES, R. *et al*. Evaluating the Effects of Branch Prediction Accuracy on the Performance of SMT Architectures. In: EUROMICRO. 9. 2001: EUROMICRO WORKSHOP PARALLEL AND DISTRIBUTED PROCESSING. 9. Mantova, 2001. *Proceedings...* Mantova, 2001. p. 355-362.

HAYKIN, S. *Neural Networks.* New York: Prentice Hall. 1994.

JAIN, A. K.; MAO, J. Artificial Neural Networks: a tutorial. *Proceedings of the IEEE Computer*, Los Alamitos, p. 31-44, 1996.

JIMÉNEZ, D. A.; LIN, C. Dynamic branch prediction with perceptrons. In: INTERNATIONAL SYMPOSIUM ON HIGH PERFORMANCE COMPUTER ARCHITECTURE., 7. Monterrey. 2001. *Proceedings...* Monterrey, 2001. p. 20-24.

JIMÉNEZ, D. A.; LIN, C. Perceptron learning for prediction the behavior of conditional branches. In: INNS-IEEE INTERNATIONAL JOINT CONFERENCE ON NEURAL NETOWORKS. Washington, DC. 2000. *Proceedings...* Washington: University of Texas, 2000.

KOSKELA, T. *et al*. Time series prediction with multilayer perceptron, FIR and Elman Neural Networks. In: WORLD CONGRESS ON NEURAL NETWORKS, San Diego, 1996. *Proceedings...* San Diego: INNS Press, 1996. p. 491-496.

LEE, J. K. F.; SMITH, A. J. Branch prediction strategies and branch target buffer design. *IEEE Computer Magazine*, New York, p. 6-22., 1984.

PERLEBERG, C. H.; SMITH, A. J. Branch target buffer design and optimization. *IEEE Transactions on Computers*, v. 2, n. 4, p. 396-412, 1993.

RUSSELL, S. J.; NORVIG, P. *Artificial intelligence*: a modern approach. Englewood Cliffs: Prentice Hall Inc., 1985, p. 563-597.

SMITH, J. E. A Study of branch prediction strategies. In: ANNUAL INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE, 8., Minneapolis, 1981. *Proceedings...*Minneapolis, 1981, p. 135-148.

SMITH, J. E.; SOHI, G. S. The microarchitecture of superscalar processors. IEEE, Los Alamitos, 1995. *Proceedings...* Los Alamitos, 1995. p. 1609-1624.

THIESING, F. M. *et al.* Parallel back-propagation for prediction of time series. EUROPEAN PVM USERS' GROUP MEETING, 1., Rome, 1997. *Proceedings...* Rome, 1997.

YEH, T.; PATT, Y. N. Two-Level Adaptive Training Branch Prediction. In: ACM/IEEE INTERN. SYMPOSIUM AND WORKSHOP ON MICROARCHITECTURE, 24., Albuquerque, 1991.

Proceedings... Albuquerque, 1991, p.51-61.

YEH, T.; PATT, Y. N. Alternative implementation of two-level adaptive branch prediction. In: ANNUAL INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE, 19., Queensland, 1992. *Proceedings...* Queensland, 1992, p. 124-134.

Received on June 29, 2003.
Accepted on October 14, 2003.