The Combined Perceptron Branch Predictor

Matteo Monchiero and Gianluca Palermo

Politecnico di Milano
Dipartimento di Elettronica e Informazione, Via Ponzio 34/5, 20133 Milano, Italy
{monchier, gpalermo}@elet.polimi.it

Abstract. Previous works have shown that neural branch prediction techniques achieve far lower misprediction rate than traditional approaches. We propose a neural predictor based on two perceptron networks: the *Combined Perceptron Branch Predictor*. The predictor consists of two concurrent perceptron-like neural networks, one using as inputs branch history information, the other one using program counter bits. We carried out experiments proving that this approach provides lower misprediction rate than state-of-the-art conventional and neural predictors. In particular, when compared with an advanced path-based perceptron predictor, it features 12% improvement of the prediction accuracy.

1 Introduction

Modern high-performance microprocessors rely on sophisticated and accurate branch predictors to efficiently exploit Instruction Level Parallelism (ILP). Complex front-ends, capable of filling large instruction windows, are required to sustain high operating frequency and aggressive parallelism. Branch prediction is a key element of such a system, providing correct fetch beyond branch boundary and, therefore, large throughput instruction delivery.

Several advanced branch predictors have been suggested so far in the literature. Most of them are 2-bit counters table based predictors [1], organized in order to minimize interference which may occur in the counter tables. For example, the *2Bc-gskew* predictor [2] is composed of four 2-bit counter tables: a bimodal table (BIM), two gshare-like tables (G0 and G1) and a metapredictor table (META). Depending on the outcome of the META table, the final prediction is given either by the BIM table or by the majority vote of the predictions from the BIM, G0 and G1 tables. Other complex schemes have been recently proposed, e.g. the *Prophet/Critic* hybrid branch predictor [3], based on the combination of two components: the *prophet* and the *critic*. The critic uses both branch history and future to give a critique of the prophet prediction, which is used to make the final prediction for the current branch.

In this paper, we present an innovative branch predictor architecture, based on a neural approach, first proposed by Jimenez *et al.* in [4] (the *Perceptron* predictor). Our proposal features a novel mechanism, based on an additional *address-based* perceptron, using some PC bits as inputs, to achieve superior accuracy with respect to a single perceptron approach. Using PC bits as input of the neural network, the proposed predictor can separate branches otherwise collapsing in the same perceptron. We prove that this approach improves significantly prediction accuracy with respect to state-of-the-art conventional and neural predictors.

The paper is organized as follows: Section 2 introduces some background about neural branch prediction. In Section 3, our proposal is presented. Section 4 shows obtained experimental results. Finally, Section 5 concludes the paper.

2 Neural Branch Prediction

Branch predictors based on neural methods have been recently studied [4–7], showing that they are the most accurate predictors in the literature. In fact, neural networks can exploit much longer histories than conventional branch prediction algorithms, resulting in better performance.

The simplest neural network is the *perceptron*. For this network, the output signal, *pred*, is a non-linear function (*activation function*) of y, which is a linear combination of the network inputs, as stated by following equations:

$$pred = step(y) \tag{1}$$

$$y = \mathbf{w} \bullet [1\ \mathbf{x}]^{\mathbf{T}} = w_0 + \sum_{i=1}^{i=N} w_i x_i \tag{2}$$

where w_i are $N+1$ weights and x_i are N inputs; w_0 is called bias weight. The activation function can assume various shapes for a generic neural network. Perceptron uses the *step* function, which is a natural choice, when dealing with branch prediction patterns. The *step* function means taken when it is 1 and not-taken when 0. Vector $\mathbf{w} = [w_0, w_1, \cdots, w_N]$ is said *weight vector* and specifies the perceptron. Vector $\mathbf{x} = [x_1, \cdots, x_N]$ is the input vector, whose elements are the inputs of the network. Weights can be dynamically trained, so that prediction run-time adapts to the real taken/not-taken branch pattern.

The *Perceptron* predictor, presented in [4], uses perceptrons to predict branch outcome. It is a history-based predictor, since it maintains a Global Branch History Register (GBHR) and a set of local Branch History Registers (BHR), collected in a Branch History Table (BHT). A history register, obtained concatenating local and global history, is used as input of a perceptron network to perform the prediction. The perceptron to use is chosen by using the PC bits of the current branch. Weights are 8-bit integers and they are selected in a $n \times (h+1)$ matrix, called *Weight Table* (WT). n and h are design parameters: n has the meaning of number of entries of the WT, while h is the size of the history register, which is the network input. Each row of the matrix is a $(h+1)$-length weight vector, which determines the perceptron. When the prediction is performed, the least significant bits of the PC are used to select the row corresponding to the weight vector to use. A fast adder provides to generate the summation of the weights, according to applied inputs (see Equation 2), and a comparator makes the prediction (see Equation 1). Every time the effective

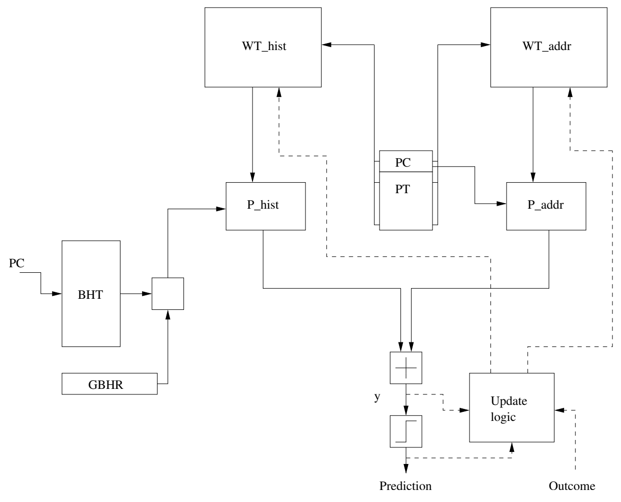


Fig. 1. Block diagram of the *Combined Perceptron Branch Predictor*

branch target is computed, the corresponding weight vector is updated, training the weights values with the outcome pattern.

An improvement to the Perceptron predictor is the *Path-Based Perceptron* predictor. Branch path information is used when selecting neurons to get superior accuracy. The path of a branch is composed of the past branch PCs. In a path-based perceptron, the i-th weight of the weight vector to use for the prediction, is the element of the i-th column of the Weight Table, indexed by the i-th element of the Path Table, that is $w_i = WT[PT[i]]$. This idea has been presented in [7], where is proposed the *Fast Path-Based* Perceptron predictor. The predictor is based on enhanced microarchitecture to minimize prediction latency. It staggers computations in time, predicting a branch using a neuron selected dynamically along the path to the branch, rather than selecting the neuron all at once.

In [8], Seznec proposes *pseudo-tagging*, a technique to reduce aliasing in the perceptron table. Pseudo-tagging consists in using a few bits of the address of a branch in the vector of weights. The author reports that this technique achieves only a slight performance improvement with respect to the conventional perceptron predictor (2.52 % on average, with a 16KB predictor, on 10 SPEC2000 integer benchmark).

3 Proposed Predictor Architecture

The *Combined Perceptron Branch Predictor*, proposed in the paper, combines two different kinds of Perceptron: a *history-based* one and an *address-based* one. The address-based Perceptron has as inputs some bits of the PC. Its output is sensitive to the branch address and, if combined with the output of the history-based Perceptron, which is sensitive to branch history, adds a contribution which

Algorithm 1 Prediction algorithm

```
/* Calculate the output of the
   history-based perceptron */
y_hist=W_hist[PC][0];
for (j = 1; j <= HISTORY_LENGTH; j++)
{
    k = PT[j-1];
    if (history_reg[j-1])
        y_hist += W_hist[k][j];
    else
        y_hist -= W_hist[k][j];
}

/* Calculate the output of the
   address-based perceptron */
y_addr=W_addr[PC][0];
for ( j=1; j <= N_ADDR_BITS; j++)
{
    k = PT[j-1];
    if (PC[j-1])
        y_addr += W_addr[k][j];
    else
        y_addr -= W_addr[k][j];
}
y = y_hist + y_addr;

if ( y >= 0 ) prediction = true;
else prediction = false;
```

Algorithm 2 Update algorithm

```
if (last_prediction!=outcome ||
   (last_y <= THETA && last_y >= -THETA))
{
   /*update the history-based perceptron*/
   if (outcome) weight_inc(W_hist[PC][0]);
   else weight_dec(W_hist[PC][0]);

   for (j = 1; j <= HISTORY_LENGTH ; j++)
   {
     k = PT[j-1];
     if (outcome == hist[j-1])
         weight_inc(W_hist[k][j]);
     else
         weight_dec(W_hist[k][j]);
   }

   /*update the address-based perceptron*/
   if (outcome)  weight_inc(W_addr[PC][0]);
   else weight_dec(W_addr[PC][0]);

   for (j = 1; j <= N_ADDR_BITS; j++)
   {
     k = PT[j-1];
     if (outcome == PC[j-1])
         weight_inc(W_addr[k][j]);
     else
         weight_dec(W_addr[k][j]);
   }
}
update_ghist();
update_lhist();
update_path();
```

Fig. 2. Algorithms for the prediction and update phase for the *Combined Perceptron Branch Predictor*

significantly improves the prediction accuracy. The basic idea of our proposal is to add to the final prediction a contribution to take into account branch address related information, dealiasing branch which collapsed in a single entry of the other component of the predictor.

Both subpredictors (history-based and address-based), which compose the whole predictor, are Perceptron predictors which exploit branch path information in the selection of the weight vector. The history-based predictor has the same structure of the Path-Based predictor described in the previous section. The address-based predictor uses a perceptron selected by the branch path, but the input of the perceptron are the least significant bits of the address of the current branch itself. We designed the update policy of this component to make that a weight is decremented/incremented if the corresponding input (that is, an address bit) has given a negative/positive contribution to the final prediction.

Furthermore, the activation function application is moved from the output of the two single subpredictors to the output of the whole predictor. In this way, each component cooperates in calculating the input value of the activation function, which is subsequently applied.

The whole predictor output is ruled by following equations, which replace Equation 1 and Equation 2.

$$pred = step(y) \tag{3}$$

$$y = \mathbf{w_{hist}} \bullet [1\ \mathbf{h}]^{\mathbf{T}} + \mathbf{w_{addr}} \bullet [1\ \mathbf{x}]^{\mathbf{T}} \tag{4}$$

where $\mathbf{w_{hist}}$ is the the weight vector of the history-based perceptron, while $\mathbf{w_{addr}}$ is the weight vector of the address-based perceptron. $\mathbf{h}$ and $\mathbf{x}$ are, respectively, the vectors of the input history and address bits. The activation function is the *step* function and applies the summation of the two components of the predictor.

Figure 1 shows the block diagram of the proposed predictor. We indicated as *WT_hist* and *P_hist* the Weight Table and the perceptron logic of the history-based subpredictor, while *WT_addr* and *P_addr* are related to the address-based one. Perceptron logic is substantially composed of an adder which sums selected weights depending on the inputs. The Path Table (PT) holds the branch path, that is, last branch PCs, and it is used to index into the Weight Tables. GBHR and a BHT store information related to branch outcome history and supply the history register which is the input of the history-based subpredictor. The *update logic* is the circuitry needed to update the predictor Weight Tables. Dashed lines represent data transfers needed by the update phase.

The prediction algorithm is shown, as C-like pseudocode, in Algorithm 1 (Figure 2).Weight Tables of both predictors are concurrently accessed to get weight vectors. The outputs of the perceptrons are calculated and summed together. Finally, a comparator decides the prediction whether the obtained value is greater or less than zero.

Update algorithm details are shown in Algorithm 2.The weights of the two subpredictors tables are modified on mispredictions or when the value of y is too small. A threshold is established for this purpose. Its value has been set, so that weight vectors are updated if y falls into a value range which is half of the weight range (that is, $THETA = 64$). The update is performed following the rule: $\mathit{\Delta w} = \mathit{outcome} \oplus \mathit{input}$, which means that a weight is incremented if the corresponding input has given positive contribution, otherwise it is decremented. The GBHR, the BHT and the PT are also updated in this phase.

3.1 Implementation Issues

Implementation of perceptron-based branch predictors has been studied in [4, 7]. The most critical component, heavily impacting on prediction latency, is the weights summation, which can be effectively implemented using a Wallace compressor. A pipelined implementation has been proposed for the Path-Based Perceptron [7], but it is feasible only considering global history. Local-global Perceptron predictors, if pipelined, would need too large hardware budget, since one pipeline per local history table (i.e. BHT) entry would be needed.

The architecture of the Combined Perceptron Branch Predictor can be implemented as shown in Figure 3. Since each column of the WTs is independently accessed by the program counter or by an element of the Path Table, WTs can be sliced and organized in banks, each of them containing as many ways as the Weight Tables (WTs) columns accessed by using the same addressing logic. The summation relative to the history-based and address-based perceptrons are implemented by a single block which generates the final value of the activation function, composed by a *Selective Inverter*, a Wallace tree and a parallel adder. The Selective Inverter, is a circuit to selectively invert fetched weights, according to the predictor inputs, i.e. the history register and a portion of the program counter bits. The Wallace tree is used to reduce the number of the operands to

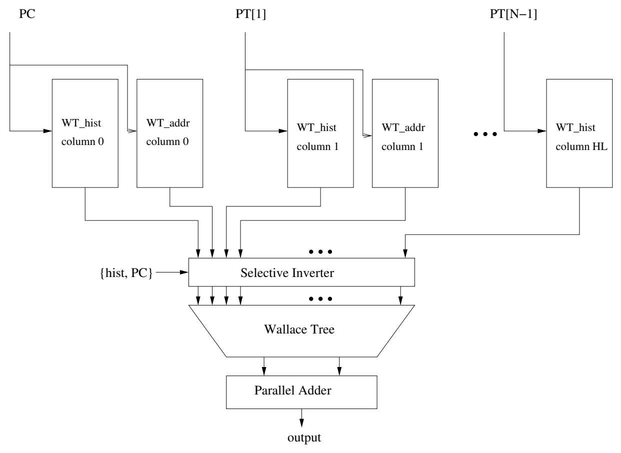


Fig. 3. Implementation of the *Combined Perceptron Branch Predictor*

2 operands, which are added by the final adder. The most significant bit of the output is the prediction.

The proposed implementation results only slightly more complex than the implementation of the Perceptron Predictor [4]. In fact, the same design can be adopted also for a Perceptron. The main difference is the width of the summation, and therefore of the Wallace tree. For example, regarding 8KB predictor as shown in Table 1, the Combined Perceptron requires 42 weights, while the Perceptron 27, resulting in 2 gate levels of the Wallace tree saving. Considering 90 nm CMOS process estimated latency for a prediction is 905 ps for the Combined Perceptron and 770 ps for the the Perceptron (-15%).

4 Experimental Results

In order to evaluate the proposed architecture, in terms of prediction accuracy, we measured misprediction rate for the proposed predictor and several different predictors. Reported results have been obtained using the Championship Branch Prediction (CBP) [9] framework, which is a trace-driven μop-based Intel IA32 simulation environment. Branch predictors were simulated on conditional branches of the given input trace. The Combined Perceptron predictor source code is available on the web [10, 11].

We used the instructions traces provided within the CBP framework. The benchmark set is composed of 20 traces, 30M instructions per trace, Each trace belongs to a specific class: INT (integer), FP (floating point), MM (multimedia), SERV (server). The INT/FP workloads are components of SPEC; the multimedia has some video/speech recognition; for the most part the server is tpcc/web server stuff.

In this paper, we compare the proposed predictor to well known state-of-the-art predictors:

Table 1. Simulated predictor configurations

		Total hardware budget		
		8KB	16KB	32KB
GShare	history length	15	16	17
2Bc-gskew	# entries (per table)	8K	16K	32K
	history length	13	14	15
Perceptron	WT # entries	304	443	630
	history length	26	36	51
Path-Based Perceptron	WT # entries	325	639	644
	global history length	17	19	40
	BHT # entries	2048	2048	2048
	local history length	4	4	7
Combined Perceptron	WT_hist # entries	137	214	493
	global history length	25	32	32
	BHT # entries	2048	2048	2048
	local history length	4	5	11
	WT_addr # entries	254	462	457
	# address bits	11	14	17

- **GShare.** It is a global two-level adaptive predictor, which uses the XOR between the global history and the branch address to minimize aliasing in the 2-bit counter table [1].
- **2Bc-gskew.** We simulated the predictor proposed by Seznec [2].
- **Perceptron.** This is the Jimenez's *Perceptron* predictor proposed in [12]. Only a global history information is used to compute the perceptron output.
- **Path-Based Perceptron.** It is an improved version of the *Perceptron* predictor. Weights are selected exploiting path information and a mixed local/global history is used.

Parameters space of the simulated predictors, including the proposed predictor, has been explored. More than 10,000 random generated configurations have been simulated and best predictors have been selected. Table 1 reports the parameters values of the optimal configurations for each predictor.

Figure 4 shows the average misprediction rate of the different branch predictors, when the size is varied. We simulated 8KB, 16KB and 32KB predictors. It can be observed that the Combined Perceptron Branch Predictor features the smallest misprediction rate for every size ranging from 3.5 to 2.8 mispredictions/K-instruction. The overall behavior of the proposed predictor is 12.5% better than the optimized configuration of the Path-Based predictor and 34% better than the GShare predictor.

In Figure 5, the plot of the misprediction rate of the simulated predictors for each benchmark, for the size of 8KB, is reported. It can be observed that the Combined Perceptron predictor achieves better performance over the other predictors for every benchmark (except for INT-4 and MM-1). A large decrease of the mispredictions is evident on integer and server benchmarks. All the simulated predictors behave homogeneously across all the benchmarks. On the other hand, the 2Bc-gskew predictor clearly results in fewer misprediction on integer programs than on server ones.

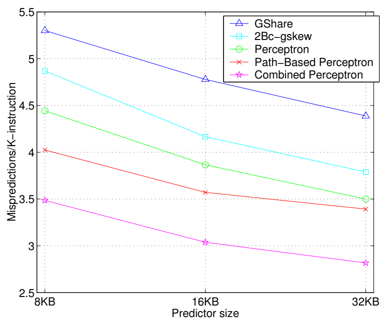


Fig. 4. Average misprediction rate of different branch predictors and sizes

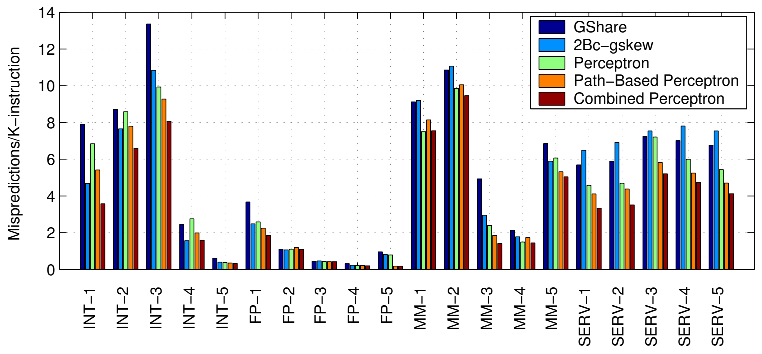
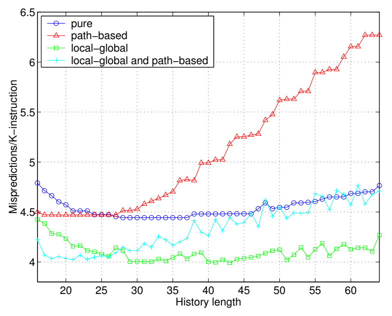
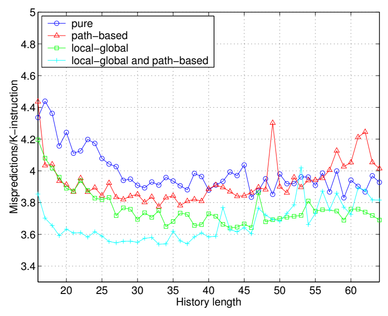


Fig. 5. Misprediction rate of the simulated branch predictors for different benchmarks and resource budget of 8KB

Results about exploration of different addressing modes into the Weight Tables are reported in Figure 6, for both the Perceptron Predictor and the Combined Perceptron Predictor. We consider *pure* addressing mode, which means that Weight Tables are accessed only by global history bits, and the *local-global* one, which mixes local and global history. Furthermore, the *path-based* and *local-global path-based* addressing modes, which use also path information. Observing Figure 6(a), it is evident that *path-based* solutions feature good accuracy for a relatively small history length (up to 24–26), while for longer history performance decreases rapidly. This is mainly due to path interference, which occurs in the path-based predictors, since weights ideally belonging to different path may collapse into the same weight, because some weights of the paths may physically overlap. This does not happen for the others two configurations, because a weight vector is maintained for each branch. *Local-global* history significantly impact on predictor performance. In fact, 10% accuracy improvement is achieved by using local-global addressing. Although *local-global* predictors feature more complex implementation, since pipelining is not possible, they represent a worthwhile choice.

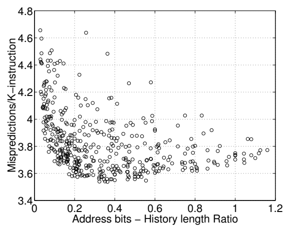


(a) Perceptron

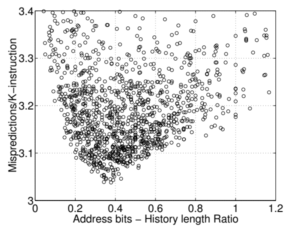


(b) Combined Perceptron

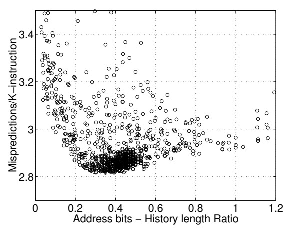
Fig. 6. Misprediction rate versus history length, considering different addressing modes into the Weight Tables (8KB predictor size)



(a) 8 KB



(b) 16 KB



(c) 32KB

Fig. 7. Address bits – History length Ratio versus misprediction rate for the simulated configurations and different predictor size

Figure 6(b) shows addressing mode analysis for the Combined Perceptron Branch Predictor. While path interference makes that path information is not effectively exploited in the Perceptron Predictor, the Combined Predictor succeeds in getting far lower misprediction rate by de-interfering paths. In fact, for history length shorter than 40 bits, up to 8% performance gain is obtained by *path-based* predictor, both for the *local-global* and the *global* one. In addition to this it is evident that the *local-global path-based* predictor reaches a misprediction rate minimum, since effectively exploit information related to both path and local-global history.

Figure 7 shows scatter plots of the misprediction rate versus the *Address bits – History length Ratio* (defined as the ratio of the number of address bits used as input of the address-based predictor and length of the history used as input of the history-based predictor) for the simulated configurations and predictor size of 8KB, 16KB, 32KB. These results show that the contribution of the *address-based* component significantly improves perceptron predictor accuracy. The phenomenon is evident for each predictor size: if the points on the Pareto curve of each plot are observed, a minimum for the misprediction rate can be found, when the value of the Address bits – History length Ratio is approximately 0.4.

5 Concluding Remarks

An innovative branch predictor architecture, based on a neural approach, has been presented. We show that combining a *history-based* perceptron with an *address-based* perceptron significantly improves prediction accuracy. We carried out experiments on a set of the benchmark traces. The proposed predictor achieves 34% lower misprediction rate than the baseline GShare predictor and 12% lower misprediction than a state-of-the-art *Perceptron* predictor. Results prove that the branch predictor architecture we propose succeeds in exploiting information related to branch path, unlike conventional path-based architecture which suffers from path interference in the Weight Table.

6 Acknowledgments

We wish to thank the CBP committee, in particular Chris Wilkerson and Jared Stark, for providing the simulation framework. We also thank Mariagiovanna Sami and Cristina Silvano for helpful discussions.

References

1. Evers, M., Yeh, T.Y.: Understanding branches and designing branch predictors for high performance microprocessors. Proceedings of the IEEE **89** (2001) 1610–1620
2. Seznec, A., Felix, S., Krishnan, V., Sazeides, Y.: Design tradeoffs for the Alpha EV8 conditional branch predictor. In: Proceedings of ISCA'02. (2002)
3. Falcon, A., Stark, J., Ramirez, A., Lai, K., Valero, M.: Prophet/critic hybrid btanch prediction. In: Proceedings of ISCA'04. (2004)
4. Jimenez, D.A., Lin, C.: Neural methods for dynamic branch prediction. ACM Transactions on Computer Systems **20** (2002) 369–397
5. Vintan, L.N., Iridon, M.: Towards a high performance neural branch predictor. In: Proceedings of the International Joint Conference on Neural Networks. (1999)
6. Egan, C., Steven, G., Quick, P., Anguera, R., Vintan, L.: Two-level branch prediction using neural networks. Journal of Systems Architecture **49** (2003) 557–570
7. Jimenez, D.: Fast path-based neural branch prediction. In: Proceedings of MICRO-36. (2003)
8. Seznec, A.: Redundant history skewed perceptron predictors: Pushing limits on global history branch predictors. Technical Report 1554, IRISA (2003)
9. CBP URL: (www.jilp.org/cbp/)
10. Monchiero, M., Palermo, G.: The combined perceptron branch predictor. Technical Report 2004.35, Politecnico di Milano (2004)
11. URL: (www.elet.polimi.it/upload/monchier/high-performance-bp.htm)
12. Jimenez, D., Lin, C.: Dynamic branch prediction with perceptrons. In: Proceedings of HPCA-7. (2001)