

# Combining Local and Global History Hashing in Perceptron Branch Prediction

C. Y. Ho and Anthony S. S. Fong

Department of Electronic Engineering, City University of Hong Kong

## Abstract

As the instruction issue rate and depth of pipelining increase, branch prediction is considered as a performance hurdle for modern processors. Extremely high branch prediction accuracy is essential to deliver their potential performance. Many perceptron branch predictors have been investigated to improve the dynamic branch prediction in recent years. This paper introduces combining local history hashing and global history hashing in perceptron branch prediction. This proposed perceptron predictor utilizes self-history as well as global history in indexing different weights of a perceptron. The simulation results show that our proposed perceptron predictor is more accurate than the one using either global history hashing or local history hashing alone. Our proposed perceptron predictor is able to achieve 4.13% misprediction rate and even 0.45% misprediction rate in some cases. And it has an improvement of 9.21% over using global history hashing alone, the mapping scheme proposed by Tarjan and Skadron.

## 1. Introduction

Before the direction of a conditional branch is resolved, processor speculatively fetches and executes subsequent instructions according to branch prediction outcome. Since the speculative work must be flushed if a prediction is incorrect, the penalty for a misprediction could be substantial. And the penalty incurred by misprediction increases as the instruction issue rate widen as well as the pipeline depth deepen. Therefore, branch prediction accuracy is considered to be critical design issue for modern microarchitectures.

In the past, most research works for branch prediction have focused primarily on 2-bit saturating up-down counter. Yeh and Patt [11] investigated exploiting two level branch histories to capture the behavior of branches. Pan et al. [12] and Sechrist et al. [13] examined the correlation between branches. Sprangle et al. [9] and Lee et al. [10] explored reducing the effect of negative interference between branches.

In recent years, perceptron is introduced as an alternative to the 2-bit saturating up-down counter for branch prediction. Jiménez and Lin [1, 2] proposed the

original perceptron predictor. Jiménez [3, 4] investigated the path-based neural branch predictor. Tarjan and Skadron [5] explored merging path and gshare indexing in perceptron predictor. These perceptron-based branch predictors have been shown to achieve superior accuracy compared with counter-based branch predictors.

In this paper, we introduce incorporating local history hashing with global history hashing in perceptron branch predictor. It attempts to exploit local history together with global history in hash indexing to select different weights of a perceptron. It benefits branches exhibiting repetitive behaviors as well as branches strongly correlated with previous neighboring branches. In addition, our proposed perceptron predictor considers more history information to make a prediction and is able to learn linearly inseparable behaviors, resulting in improved accuracy.

The remainder of this paper is organized as follows. Section 2 describes the fundamental concepts of perceptron and perceptron branch predictor. Section 3 presents our proposed approach of perceptron branch prediction. Section 4 provides the simulation results. Section 5 gives the conclusion finally.

## 2. Perceptron branch predictors

This section provides the fundamental background of perceptron and perceptron branch predictor.

### 2.1. Perceptrons

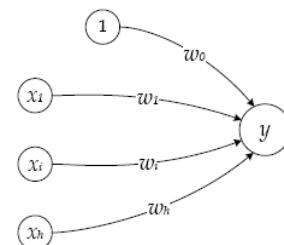


Figure 1. Perceptron

Perceptron is the simplest model of an artificial neural network which is used for pattern classification [6]. As depicted in figure 1, each perceptron keeps track of a weight vector  $w$ . The  $w_0$  is a bias weight to which the corresponding input unit is always set to +1. The

perceptron receives an input vector  $x$ . Each input unit  $x_i$  is associated with a weight  $w_i$ . The output  $y$ , computed by the following equation, is the dot product of input vector and weight vector. The sign of output  $y$  serves to make prediction, that non-negative output indicates “*taken*” and negative output indicates “*not taken*”.

$$y = w_0 + \sum_{i=1}^h x_i w_i$$

Once the actual branch outcome becomes known, the training algorithm is performed to modify the weight values. Let  $t$  be the actual branch outcome. And let  $\theta$  be the training threshold which is a parameter to determine whether the perceptron needs further update or not. The following pseudocode is the training algorithm:

```

if sign(y) ≠ t or |y| ≤ θ then
    for j in 0..h do
        w_i := w_i + tx_i
    end for
end if

```

The weights are updated only when the prediction outcome does not agree with the actual branch outcome or when the absolute value of  $y$  remains below the threshold value.

The drawback of perceptrons is the ability of learning linearly separable functions only [7]. A decision boundary, called hyperplane, separating two decision regions, is defined by the equation,

$$w_0 + \sum_{i=1}^h x_i w_i = 0$$

Intuitively, the two decision regions are the non-negative outputs and the negative outputs. A function is said to be “linearly separable” if given a set of input patterns, all of the non-negative outputs are sufficiently separated from all of the negative outputs by the hyperplane [7]. Boolean functions such as AND, OR are linearly separable whereas XOR is linearly inseparable.

## 2.2. Original perceptron branch predictor

Perceptron branch predictor was introduced originally by Jiménez and Lin [1, 2]. As depicted in figure 2, a table of perceptrons,  $[w_0, \dots, w_h]$ , is indexed by the address of branch to be predicted. The global branch history,  $[x_1, \dots, x_h]$ , representing the outcomes of recently executed branches, serves as an input vector to the selected perceptron. The input values are bipolar that means the values are either +1 (*taken*) or -1 (*not taken*). The weights are 8-bit signed integers. The sign of the computed output  $y$  determines the direction of prediction. The threshold  $\theta$  is set to be  $[1.93h + 14]$  where  $h$  is the length of global history inputted to the perceptron.

Actually, the perceptron attempts to learn the correlations between the behavior of previous resolved branches and the behavior of current branch. The degree

of these correlations is indicated by the magnitude of the weights. Due to the limitation of learning linearly separable function only, original perceptron predictor is unable to attain 100% overall prediction accuracy.

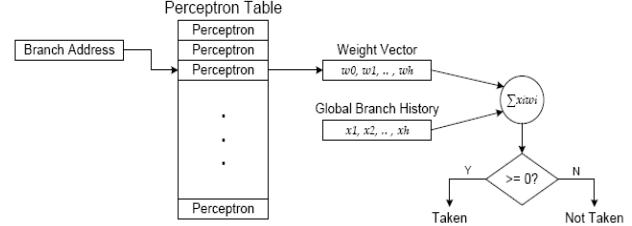


Figure 2. Original perceptron predictor

## 3. Global/Local hashed perceptron predictor

We propose the perceptron branch predictor combining local history XOR hashing with global history XOR hashing. It exploits both local branch history and global branch history in gshare indexing to select weights. It takes the advantages of global and local branch history, as well as the benefits of gshare indexing. The superior characteristic of our proposed perceptron predictor is that, by using local branch history (or self-history) as well, it does reliably predict the branches exhibiting repetitive behaviors. Secondly, using XOR hash function can reduces the destructive aliasing among weights. Thirdly, it is able to learn linearly inseparable branches. Consequently, prediction accuracy is improved.

### 3.1. Multiple indexing methods

A table of perceptrons can be interpreted as an  $n \times (h+1)$  matrix  $W[0..n-1, 0..h]$ , where  $n$  is the number of perceptrons and  $h$  is the number of weights in a perceptron. For each column of the matrix, one weight is selected and considered as a weight element of a perceptron. Different weights of a perceptron can be indexed by different methods. And all of the indexed weights serves as a weight vector,  $[w_0, \dots, w_h]$ , for calculation of dot product of a perceptron. The indexing methods are differentiated from different hash function and branch information used for that hash function. In this work, the following three indexing methods are incorporated in a perceptron branch predictor. Here *addr* is the current branch address. The *local* represents local branch history segment of the branch to be predicted and *global* represents global branch history segment.

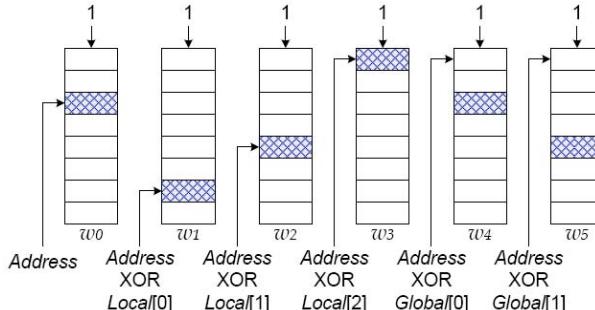
$$\begin{aligned} index[j] &= addr \bmod n \\ index[j] &= (local[j] \oplus addr) \bmod n \\ index[j] &= (global[j] \oplus addr) \bmod n \end{aligned}$$

The first hash function is using branch address alone, which can effectively identify different branches. The second and third hash function involves XOR of global and local branch history with branch address separately.

These are the idea behind the gshare actually. In gshare predictors, the exclusive OR of global branch history and branch address is used to generate index into a table of 2-bit saturating up-down counters. It is well known that gshare predictors are able to distinguish branches having same global history patterns, resulting in reduction of destructive aliasing [8]. Moreover, Tarjan and Skadron proposed hashed perceptron predictor which XOR a segment of global branch history with the branch address to select weights [5]. It predicts the direction of current branch based upon the global branch history. However, it does not utilize the information content of local branch history effectively.

Our alternative to perceptron branch predictor is inspired by that of Tarjan and Skadron. Rather than XORing the global branch history with the branch address alone, our proposed perceptron branch predictor exploits local branch history in addition to global branch history XORed with branch address to select weights. This proposed perceptron predictor is referred to as global/local hashed perceptron predictor in this work. The advantage of XOR hash function is that it is able to generate more useful distribution of indexing into the perceptron matrix. Besides, using global branch history in hash function favors branches having strong correlation with most recently executed branches. The introduction of local branch history in hash function provides benefits of predicting branches which exhibit repetitive loop patterns. Since both local and global branch histories are in use, the global/local hashed perceptron predictor would provide higher accuracy than using global branch history alone.

### 3.2. Weights selection approach



**Figure 3. Weights selection of global/local hashed perceptron predictor**

Figure 3 illustrates the proposed weights selection approach. Except the bias weight, some weights are selected by XOR of local branch history with branch address, and others are selected by XOR of global branch history with branch address. The bias weight  $w_0$  is accessed by the branch address directly. And all input values to the selected weights are +1. The weights are 8-bit signed integers.

In this example, the number of perceptrons  $n$  is 8 and the number of weights in a perceptron  $h$  is 5. And the number of weights indexed by XORing local branch history with branch address is 3. The remaining weights of a perceptron (5-3) are indexed by XORing global branch history with branch address.

### 3.3. Prediction algorithm

The following pseudocode shows the prediction algorithm. Here the definitions of  $W$ ,  $n$ ,  $h$ ,  $addr$ ,  $local$ ,  $global$  are the same as that of previous section. Let  $length$  be the number of weights in a perceptron indexed by the local history XOR mapping. Then the number of weights indexed by the global history XOR mapping is ( $h-length$ ).

```

function prediction (addr: integer): {taken, not_taken}
begin
    index[0] := addr mod n
    y := W[index[0], 0]
    for j in 1..h do
        if j ≤ length then
            index[j] := (local[j-1] ⊕ addr) mod n
            y := y + W[index[j], j
        else
            index[j] := (global[j-length] ⊕ addr) mod n
            y := y + W[index[j], j
        end if
    end for
    if y ≥ 0 then
        prediction := taken
    else
        prediction := not_taken
    end if
end

```

Whenever a branch is to be predicted, the branch address modulo number of perceptrons is computed to index the bias weight  $w_0$ . The local branch history of that branch is XORed with branch address to select a portion of weights. And the global branch history is XORed with branch address to select remaining weights. All of these chosen weights are considered as a set of weights of a perceptron. Since all of the input values to the weights are +1, the output  $y$  of perceptron is calculated by following equation,

$$y = w_0 + \sum_{i=1}^h w_i$$

The output  $y$  is the sum of all weight elements  $w_i$ . The prediction outcome is made according to the sign of computed output  $y$ . The branch is predicted as *taken* if the computed output is non-negative, whereas it is predicted as *not taken* if the computed output is negative.

### 3.4. Training algorithm

Once the actual outcome of the branch is resolved, the training algorithm is invoked to modify the set of weights. Below we give the pseudocode of the training algorithm. It is similar to that of hashed perceptron predictor [5] except that it updates the local history shift register together with global history shift register.

```

function training (index[], y: integer; prediction, outcome:  

{taken, not_taken})  

begin  

  if prediction ≠ outcome or  $|y| \leq \theta$  then  

    for j in 0..h do  

      if outcome = taken then  

        W[index[j], j] := W[index[j], j] + 1  

      else  

        W[index[j], j] := W[index[j], j] - 1  

      end if  

    end for  

  end if  

  local := (local << 1) or outcome  

  global := (global << 1) or outcome  

end
```

As with previous perceptron predictors, the perceptron is trained only when the actual outcome does not agree with the prediction result or the absolute value of computed output *y* remains below the threshold value  $\theta$ . Here the threshold value is set to be  $[1.93*h + h/2]$ , which is the same as that of hashed perceptron predictor [5]. The threshold value is a constant for a fixed number of weights in a perceptron and it determines whether the perceptron required to be updated or not. The weight values are incremented by 1 if the actual outcome is taken. They are decremented by 1 otherwise. In addition, the actual outcome is shifted left into the local history shift register of that branch as well as the global history shift register in the least significant bit position.

### 3.5. Advantages of global/local hashed perceptron predictor

Using both local and global branch history in making a prediction has more information than either component alone. The global branch history benefits making predictions for branches which are strongly correlated with the most recently neighboring branches, such as if-then-else branches. On the other hand, some branches exhibit periodic behavior, such as loop control branches. These branches are better predicted using local branch history rather than global branch history. Thus, combining local branch history and global branch history in perceptron predictor is able to learn branches having

repetitive behaviors in addition to branches correlated with previous branches.

It has been shown that XORing the branch history with branch address is an effective hash function which is able to distinguish branches having same history pattern and so alleviate the destructive aliasing [8]. In our proposed perceptron predictor, branch history is incorporated with branch address by XOR hash function to generate more useful distribution of indexing across perceptron matrix. This is able to map the interfering branches into distinct weights and hence reduce the impact of interference between branches with opposite biased behaviors.

Furthermore, XOR hash function allows more branch history information exploited in making a prediction, resulting in improved prediction accuracy. Previous perceptron predictors use branch history bits as input units to perceptrons to predict branches. Since one weight receives one history bit only, this restricts the number of history bits used. In contrast, XOR hash function involves a number of history bits with branch address to index one weight. This increases the total number of history bits in making a prediction, and consequently has advantage when the distance between correlated branches is long in some cases [14].

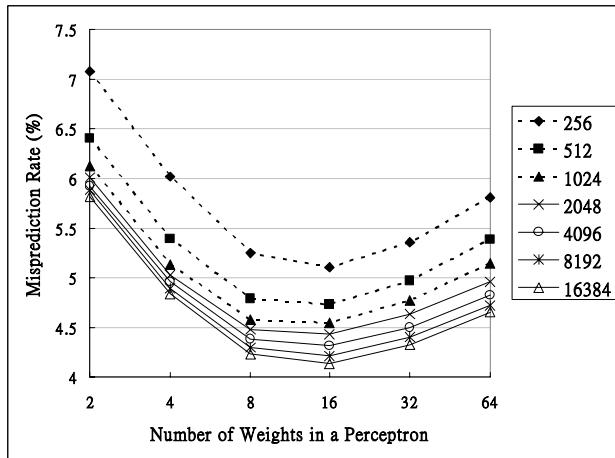
Also, the superior characteristic of global/local hashed perceptron predictor is the capability of learning linearly inseparable functions, which is a limitation of perceptrons. Since every weight element  $w_i$  is selected by XOR hashing separately and all of the input units to the weights are +1, the output *y* is computed as the sum of weights, which does not suffer from inability of learning linearly inseparable functions. In other words, the whole perceptron table can be considered to be composed of a number of sub-tables. Each of these sub-tables behaves like a small gshare predictor which can reliably predict linearly inseparable branches. Thus, our proposed perceptron predictor provides extra prediction accuracy for linearly inseparable branches.

## 4. Simulation Results

In order to evaluate the prediction accuracy of global/local hashed perceptron predictor, we conduct a trace-driven simulation. We use 12 SPEC CPU integer benchmarks including *164.gzip*, *175.vpr*, *176.gcc*, *181.mcf*, *186.crafty*, *197.parser*, *252.eon*, *253.perlbmk*, *254.gap*, *255.vortex*, *256.bzip2*, *300.twolf*. All of the conditional branches are gathered by recording their branch addresses together with actual outcomes in a trace file. The sequence of traces gathered from these integer benchmarks are inputted to a program that is the simulator of global/local hashed perceptron predictor. This simulator gets the branch addresses and predicts the direction of branches. It then compares the predictions with the actual outcomes to collect the statistics of prediction accuracy.

We evaluate the global/local hashed perceptron predictor with increasing number of perceptrons across a range of number of weights in a perceptron. Figure 4 shows the results of average misprediction rate with the number of weights in a perceptron ranging from 2 to 64. Each curve represents a given number of perceptrons. Note that among different number of weights indexed by local history hashing, only the one having the best accuracy is shown. The misprediction rate reaches about 4.13% when the number of perceptrons is 16384 and the number of weights in a perceptron is 16. It can be seen that the performance increases with the number of perceptrons. At smaller number of perceptrons, the interfering branches are more likely to be mapped to the same weight for prediction. Therefore, the predictor suffers from more aliasing and it is less accurate. However, as the number of perceptrons becomes larger, the impact of aliasing is relatively small, resulting in better performance. For any fixed number of weights in a perceptron, the global/local hashed perceptron predictor is more accurate as the number of perceptrons increases.

On the other hand, the performance improves when the number of weights in a perceptron ranges from 2 to 16. However, the predictor becomes less accurate when the number of weights in a perceptron further increases from 16 to 64. For all given number of perceptrons, the best performance occurs at 16. This implies that having sufficient number of perceptrons, the number of weights in a perceptron is not necessary to be large to provide optimal prediction accuracy. In particular, the performance varies slightly when the number of weights in a perceptron is between 8 and 32 at a given number of perceptrons. Within this range, the performance improvement is subject to variations in number of perceptrons rather than number of weights in a perceptron.

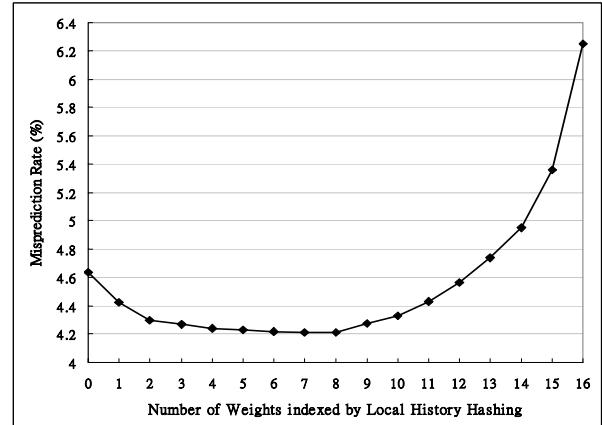


**Figure 4. Misprediction rates of different configurations**

In addition, we evaluate the impact of number of weights selected by local history hashing at a given number of weights in a perceptron. Figure 5 shows the

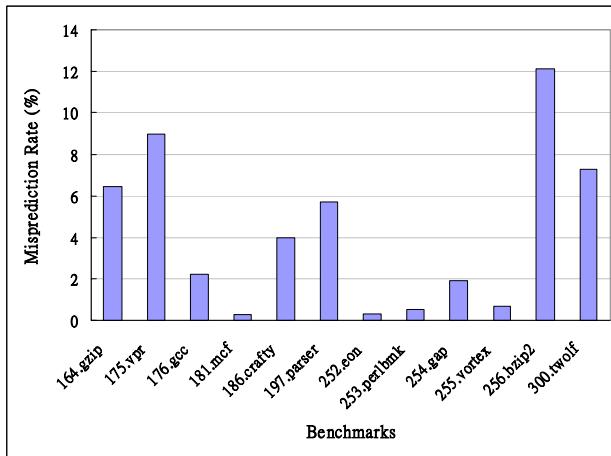
results of average misprediction rate with increasing number of weights indexed by local history hashing. Here we keep the number of weights in a perceptron at 16, and so the number of weights indexed by local history hashing ranges from 0 to 16. And the number of perceptrons is 8192. We can see that the performance improves from 0 to 8, but it degrades from 8 to 16. This provides evidence that the number of weights indexed by local history hashing should be approximately half of the number of weights in order to attain the best performance. Too many or too few number of weights indexed by local history hashing as opposed to global history hashing will degrade the performance.

At the extreme ends in figure 5, 0 and 16 represent all of the weights of a perceptron are indexed by global history hashing alone and local history hashing alone respectively. Note that the configuration with global history hashing alone is identical to the mapping scheme proposed by Tarjan and Skadron [5]. The simulation results shows that combining local history hashing with global history hashing gives more accurate prediction than either component alone. And our global/local hashed perceptron predictor achieves a superior misprediction rate of 4.21%, an improvement of 9.21% over using global history hashing alone, the mapping scheme proposed by Tarjan and Skadron. Using local history hashing alone to fetch all weights of a perceptron yields a misprediction rate of 6.25%, which is comparatively worst performance.



**Figure 5. Misprediction rates for Number of weights indexed by local history hashing**

Furthermore, we measure the prediction accuracy according to benchmarks. The number of perceptrons is also set to 8192. And the number of weights in a perceptron is 16, in which 8 weights are selected by local history hashing. The best prediction accuracy occurs on benchmarks *181.mcf*, *252.eon*, *253.perlbench* and *255.vortex*, reaching an average misprediction rate of 0.45%. The predictor performs worst on benchmarks *164.gzip*, *175.vpr*, *256.bzip2* and *300.twolf*, giving about 8.71% average misprediction rate.



**Figure 6. Misprediction rates of individual benchmarks**

## 5. Conclusions

As the issue rate and depth of pipeline continue to increase, highly accurate branch predictor is essential to delivering the potential performance of modern computer architectures. The perceptron branch predictors which achieve superior accuracy would be the focus of branch prediction research in future.

In this paper, we have introduced a global/local hashed perceptron predictor, which combines local history XOR hashing and global history XOR hashing to fetch different weights of a perceptron. The chief advantage is that it utilizes the local history as well as global history in XOR indexing. The local and global histories contribute prediction of branches with different behaviors separately. Also, the XOR hash function allows more history information associated with branch address to make a prediction. Moreover, our proposed global/local hashed perceptron predictor is able to learn linearly inseparable behaviors, which is the key limitation of perceptrons.

We have evaluated the global/local hashed perceptron predictor with different configurations. We have analyzed the prediction accuracy across a range of number of weights in a perceptron according to number of perceptrons. In addition, we have explored the number of weights should be selected by local history hashing in comparison to those selected by global history hashing at a given number of weights in a perceptron. And we have measured the misprediction rates for individual benchmarks.

The simulation results provide evidence that the performance improves with the number of perceptrons. For any given number of perceptrons, the best accuracy occurs when number of weights in a perceptron is 16. In some cases, the performance improvement is less sensitive to the variations in number of weights in a perceptron, compared with number of perceptrons. And

the optimal ratio of number of weights indexed by local history hashing to total number of weights in a perceptron should be approximately half in order to achieve the best performance. Our proposed perceptron predictor hashing both global and local histories with branch address achieve higher accuracy than one that only uses either global history hashing or local history hashing.

## 6. Acknowledgement

The work described in this paper was partially supported by the City University of Hong Kong, Strategic Research Grant 7001847.

## 7. References

- [1] D. A. Jiménez and C. Lin, "Dynamic Branch Prediction with Perceptrons", In *Proc. of the 7th Int'l Symposium on High-Performance Computer Architecture*, pp. 197-206, 2001.
- [2] D. A. Jiménez and C. Lin, "Neural Methods for Dynamic Branch Prediction", *ACM Transactions on Computer Systems*, 20(4):369-397, 2002.
- [3] D. A. Jiménez, "Fast Path-Based Neural Branch Prediction", In *Proc. of the 36th Annual IEEE/ACM Int'l Symposium on Microarchitecture*, pp. 243-252, IEEE Computer Society, 2003.
- [4] D. A. Jiménez, "Improved Latency and Accuracy for Neural Branch Prediction", *ACM Transactions on Computer Systems*, 23(2):197-218, 2005.
- [5] D. Tarjan and K. Skadron, "Merging Path and Gshare Indexing in Perceptron Branch Prediction", *ACM Transactions on Architecture and Code Optimization*, 2(3):280-300, 2005.
- [6] S. Haykin, *Neural Networks: A Comprehensive Foundation*, 2nd Edition, Prentice Hall, 1999.
- [7] L. Fausett, *Fundamentals of Neural Networks: Architectures, Algorithms, and Applications*, Prentice Hall, 1994.
- [8] S. McFarling, "Combining Branch Predictors", *Western Research Laboratory Technical Note TN-36*, June 1993.
- [9] E. Sprangle, R. S. Chappell, M. Alsup, and Y. N. Patt, "The Agree Predictor: A Mechanism for Reducing Negative Branch History Interference", In *Proc. of the 24th Annual Int'l Symposium on Computer Architecture*, pp. 284-291, 1997.
- [10] C.-C. Lee, I.-C. K. Chen, and T. N. Mudge, "The Bi-Mode Branch Predictor", In *Proc. of the 30th Annual IEEE/ACM Int'l Symposium on Microarchitecture*, pp. 4-13, 1997.
- [11] T.-Y. Yeh and Y. N. Patt, "A Comparison of Dynamic Branch Predictors that use Two Levels of Branch History", In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pp. 257-266, 1993.
- [12] S.-T. Pan, K. So, and J. T. Rahmeh, "Improving the Accuracy of Dynamic Branch Prediction Using Branch Correlation", In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 76-84, 1992.
- [13] S. Sechrest, C.-C. Lee, and T. Mudge, "Correlation and Aliasing in Dynamic Branch Predictors", In *Proc. of the 23rd Annual Int'l Symp. on Computer Architecture*, pp. 22-32, 1996.
- [14] M. Evers, S. J. Patel, R. S. Chappell, and Y. N. Patt, "An analysis of correlation and predictability: What makes two-level branch predictors work", In *Proc. of the 25th Annual Int'l Symposium on Computer Architecture*, pp. 52-61, 1998.