Appears in the Proceedings of the 32^{nd} *Annual International Conference on Computer Architecture (ISCA-32)*

Piecewise Linear Branch Prediction

Daniel A. Jiménez

Department of Computer Science
Rutgers University
Piscataway, New Jersey, USA

Abstract

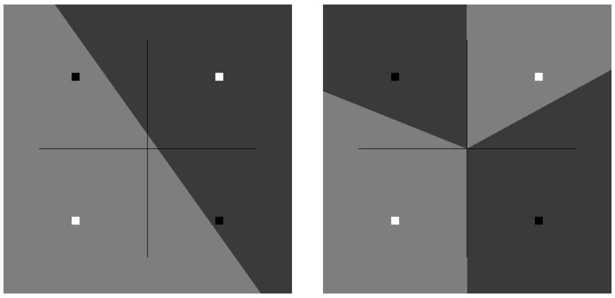
Improved branch prediction accuracy is essential to sustaining instruction throughput with today's deep pipelines. We introduce piecewise linear branch prediction*, an idealized branch predictor that develops a set of linear functions, one for each program path to the branch to be predicted, that separate predicted taken from predicted not taken branches. Taken together, all of these linear functions form a piecewise linear decision surface. We present a limit study of this predictor showing its potential to greatly improve predictor accuracy.*

*We then introduce a practical implementable branch predictor based on piecewise linear branch prediction. In making our predictor practical, we show how a parameterized version of it unifies the previously distinct concepts of perceptron prediction and path-based neural prediction. Our new branch predictor has implementation costs comparable to current prominent predictors in the literature while significantly improving accuracy. For a deeply pipelined simulated microarchitecture our predictor with a 256KB hardware budget improves the harmonic mean normalized instructions-per-cycle rate by 8% over both the original path-based neural predictor and 2Bc-*gskew*. The average misprediction rate is decreased by 16% over the path-based neural predictor and by 22% over 2Bc-*gskew.

1. Introduction

Deeper pipelines improve overall performance by allowing more aggressive clock rates. However, some potential performance is lost due to the resulting increase in branch misprediction latencies. Indeed, branch misprediction latency is the most important component of performance degradation as microarchitectures become more deeply pipelined [20]. Branch predictors must improve to avoid the increasing penalties of mispredictions.

In this paper, we introduce *piecewise linear branch prediction* that works by learning a set of linear functions for each branch that together comprise a piecewise linear surface in a space of branch outcome pattern histories. This surface separates predicted taken branches from predicted not taken branches. A piecewise linear surface allows the predictor to learn the behavior of certain linearly inseparable branches that previous neural predictors were unable to learn [12]. Figure 1 shows examples of decisions surfaces learned by the perceptron predictor (a) and piecewise linear branch prediction (b) for the linearly inseparable exclusive-OR (XOR) function. The two input variables are represented by the x and y axes, negative for false and positive for true. The output is represented by the color white for false and black for true. XOR cannot be learned by the perceptron predictor, but it is easily learned by piecewise linear branch prediction.



(a) (b)

Figure 1. The XOR function cannot be learned by a perceptron (a), but can be learned using a piecewise linear decision surface (b).

We first describe an idealized version of piecewise linear branch prediction, giving algorithms to explain the concepts but paying no attention to microarchitectural implementation constraints. In the context of a boundless hardware and computational budget, we give the results of a limit study comparing piecewise linear branch prediction to idealized versions of previous prediction mechanisms. We show that piecewise linear branch prediction reduces mispredictions by an average of 16% over the next best predictor.

We then derive a practical piecewise linear branch predictor suitable for microarchitectural implementation by constraining the state used by the idealized predictor and using ahead-pipelining to mitigate its latency. Using a cycle-accurate microarchitectural simulator, we show that the practical piecewise linear branch predictor improves normalized harmonic mean instructions-per-cycle rate (IPC) by 8% over both 2Bc-*gskew* [16] and the path-based neural predictor [9] for a 256KB hardware budget. At lower hardware budgets the new predictor also yields a significant speedup.

2. Background and Related Work

In this section, we discuss related work in branch prediction. The predictor we introduce in this paper is an extension of neural branch predictor, so we focus on neural branch prediction research.

2.1. The Perceptron Predictor

The perceptron predictor [11] uses a simple linear neuron known as a perceptron [2] to perform branch direction prediction. The most accurate single-component branch predictors in the literature are neural branch predictors [14, 12]. Unfortunately, the high latency of the original perceptron predictor and makes it impractical for improving performance.

2.2. Research Related to Neural Branch Prediction

Many studies have extended the perceptron predictor. Loh and Henry use the perceptron predictor as a component of a larger hybrid predictor [14]. Thomas *et al.* find salient history bits for the perceptron predictor using dynamic data-flow analysis. Akkary and Srinivasan adapt the perceptron predictor to provide confidence estimates for speculation control [1]. Intel includes the perceptron predictor in one of its IA-64 simulators for researching future microarchitectures [3]. Falcón *et al.* use a perceptron predictor as a component of a prophet/critic hybrid predictor that runs the branch predictor ahead to second-guess previous predictions and possibly reverse them [7].

2.3. Neural Branch Prediction Background

Neural branch predictors keep a table of *weights vectors*, i.e., vectors of small integers that are learned through the perceptron learning rule [11, 2]. As in global two-level adaptive branch prediction [22, 15], a shift register records a global history of outcomes of conditional branches, recording *true* for *taken*, or *false* for *not taken*.

To predict a branch outcome, a weights vector is selected by indexing the table with the branch address modulo the number of weights vectors. The dot product of the selected vector and the global history register is computed, where *true* in the history represents 1 and *false* represents -1. If the dot product is at least 0, then the branch is predicted taken, otherwise it is predicted not taken.

2.4. Path-Based Neural Branch Prediction

In a recent paper, we describe a neural predictor that achieves lower latency and improved accuracy over previous neural branch predictors [9] by staggering computations in time, predicting a branch using a weights vector selected dynamically along the path to that branch. The path-based neural branch predictor has improved accuracy over previous neural predictors because it is able to find correlations with path history as well as pattern history. In this paper, we present a new neural predictor that achieves even better accuracy at much the same latency.

3. Piecewise Linear Branch Prediction

This section introduces our new predictor, the piecewise-linear branch predictor. We discuss the intuition behind the idea, then give the algorithm. It is important to note that, at this point, we are presenting an idealized predictor without concern for its implementation since we are only concerned with its predictive power. In Section 5, we derive a practical branch prediction from this idealized predictor.

3.1. Intuition

Branch predictors exploit the correlation between the history of a branch and its outcome. One way to conceptually organize the notion of branch history is to consider all of the program paths of a given length h ending in a branch B. For our purposes, a path is a dynamic sequence of branches ending in B. The identities, positions, and outcomes of such a path usually correlate highly with the outcome of a branch. For each branch B, our predictor tracks the elements of every path leading to B. The predictor keeps track of the tendency of a given branch in a given position in the history to agree with the outcome of B. That is, for every component of every path, the predictor tracks the correlation of that component with the outcome of B. To make a prediction, the correlations of each component of the current path are aggregated to form a prediction.

3.1.1. What is "Piecewise Linear?" This aggregation is a linear function of the correlations just for the current path. This linear function induces a hyperplane that is used to decide whether to predict taken or not taken: if the global branch outcome pattern history lies on one side of the hyperplane, the branch is predicted taken, otherwise it is predicted not taken. Since there are many paths leading to B, there are many different linear functions used to predict B. Taken as a whole, the linear functions used to predict B

form a piecewise-linear surface separating paths that lead to predicted taken branches from paths that lead to predicted not taken branches. In machine learning terminology, such a separating surface is called a *decision surface*. In Section 6 we will show that piecewise linear branch prediction is a generalization of neural branch prediction [11], which uses a single linear function for a given branch, and path-based neural branch prediction [9], which uses a single global piecewise-linear function to predict all branches. As a perceptron-like algorithm, the predictor finds the sum of weights or their negations based on whether the corresponding history bits represent taken or not taken branches.

3.2. Description of the Algorithm

The algorithm has two components: a prediction function and an update procedure. The following variables are used by the algorithm:

W A three-dimensional array of integers. The indices of this array are the branch address, the address of a branch in the path history, and the position in the history. W keeps track of correlations for every branch in the program. We can think of W as a set of matrices, one for each branch, whose columns correspond to branches in the path history and whose rows correspond to positions in the history. $W[B, 0, 0]$ is the weight that keeps track of the tendency of branch B to be taken. This weight is the *bias weight* for B. Addition and subtraction on elements of W saturate at +127 and -128. The dimensions of the array are arbitrarily large, i.e., large enough to accommodate any access that might be made during the algorithm. This fact alone makes the predictor infeasible for actual implementation. Nevertheless, we constrain the storage in our practical version of the predictor presented later.

h The global history length. This is a small integer.

GHR The global history register. This vector of bits accumulates the outcomes of branches as they are executed. Branch outcomes are shifted into the first position of the vector.

GA An array of addresses. As branches are executed, their addresses are shifted into the first position of this array. Taken together, GHR and GA give the path history for the current branch to be predicted.

output An integer. This integer is the value of the linear function computed to predict the current branch.

Figure 2 shows the function *predict* that computes the Boolean prediction function. The function accepts as a parameter the address of the branch to be predicted. The branch is predicted taken if *predict* returns `true`, not taken otherwise. Figure 3 shows the procedure *train* used to update the predictor. It accepts as parameters the address and Boolean outcome of the branch. It assumes that all variables retain the values they had at the end of the invocation of *predict* for this branch. The training algorithm uses a threshold parameter θ to decide when to stop updating the predictor; when the magnitude of the output of the predictor exceeds θ, the predictor has learned the behavior of the branch sufficiently well. Without such a threshold parameter, the predictor might overtrain and be slow to respond to changes in branch behavior. This algorithm is similar to other neural predictors such as the perceptron predictor [11] and the path-based neural predictor [9]. We choose the value of θ using the same formula from the path-based neural predictor paper, $\theta = 2.14(h + 1) + 20.58$.

4. Limit Study

This section presents a limit study of branch prediction using piecewise linear branch prediction as well as idealized versions of other predictors. We show that our new predictor is able to achieve very low misprediction rates compared with other idealized predictors.

4.1. Predictors Simulated

We simulate the following predictors to compare with our new predictor:

2Bc-gskew We simulate 2Bc-*gskew*, a hybrid predictor combining a bimodal component with *egskew* that predicts using the majority of three components: the bimodal predictor and two global history predictors indexed by special hash functions that mitigate destructive interference. A version of this predictor was planned for the Alpha EV8 processor [16]. We have observed that 2Bc-*gskew* is the most accurate branch predictor based on two-level adaptive prediction in the academic literature.

Perceptron Predictor We simulate a version of the perceptron predictor that combines global and per-branch history information [12]. This predictor has been shown to be more accurate than even the most aggressive multi-component hybrid predictor [12]. Thus, it would be superfluous to compare against other combined global and per-branch hybrid predictors. For this study, a history length of h for the global/local perceptron means that h global and h local history bits are used.

Path-Based Neural Predictor We simulate the path-based neural predictor [9]. As in piecewise linear branch prediction, this predictor aggregates weights from the elements of the path to the branch to be predicted, but the weights are shared globally among all branches instead of being associated with a particular branch. This path-based predictor is highly accurate. Thus, we do not include other, less accurate path-based techniques.

4.2. Limit Study Methodology

In this paper, we present the results of two sets of experiments. In this section, we present a limit study focused only on accuracy. In Section 8, we present a study of our

```
function predict (address: integer): boolean
begin
    output := W[address, 0, 0]                        (* Output is initialized to bias weight *)
    for i in 1..h do                                  (* Find the sum of weights (or their negations) chosen *)
        if GHR[i] = true then                         (* using the addresses of the last h branches *)
            output := output + W[address, GA[i], i]   (* If the i^th branch in the history was taken, *)
        else                                          (* add the chosen weight *)
            output := output − W[address, GA[i], i]   (* otherwise subtract it *)
        end if
    end for
    predict := output ≥ 0                             (* Predict the branch taken if the output is at least 0 *)
end
```

Figure 2. Prediction algorithm

```
procedure train (address: integer; taken: boolean)
begin                                                         (* If magnitude of output is less than θ or prediction was *)
    if |output| < θ or predict ≠ taken then                   (* incorrect then update the weights *)
        if taken = true then
            W[address, 0, 0] := W[address, 0, 0] + 1          (* If branch was taken, then increment the bias weight, *)
        else
            W[address, 0, 0] := W[address, 0, 0] − 1          (* otherwise decrement it (with saturating arithmetic) *)
        end if
        for i in 1..h                                         (* For each address and branch outcome in recent history... *)
            if GHR[i] = taken then                            (* If the i^th most recent outcome is equal to current outcome *)
                W[address, GA[i], i] := W[address, GA[i], i] + 1   (* then increment the weight that contributed to this prediction *)
            else
                W[address, GA[i], i] := W[address, GA[i], i] − 1   (* otherwise decrement it (with saturating arithmetic) *)
            end if
        end for
    end if
    GA[2..h] := GA[1..h − 1]                                  (* Shift the current address into the global address array *)
    GA[1] := address
    GHR[2..h] := GHR[1..h − 1]                                (* Shift the current outcome into the global history register *)
    GHR[1] := taken
end
```

Figure 3. Training algorithm

proposed practical branch predictor in a cycle-accurate microarchitectural framework. For this limit study, we use traces generated by SimpleScalar/Alpha on the same 15 SPEC CPU integer benchmarks we use for Section 8.

To isolate the predictive power of each prediction mechanism from the effects of branch interference, each predictor is allowed to use an arbitrary amount of storage. For the perceptron predictor and path-based neural predictor, one weights vector is allocated to each static branch. For 2Bc-*gskew*, each table in the predictor is given a virtual 48-bit address space. In other words, the 2Bc-*gskew* predictor is given a virtual hardware budget of 256 terabytes. The simulation of this budget is facilitated through the use of a data structure that brings a predictor entry into existence only when it is first accessed.

Branch predictor accuracy is highly sensitive to history length [6]. We simulate each predictor on all of the traces for a variety of history lengths. For each predictor except for piecewise linear prediction, we simulate a large enough range of history lengths to show that there is a best history length delivering the lowest misprediction rate for that predictor. For piecewise linear prediction there does not seem to be a best history length; accuracy continues to improve with longer histories.

Figure 4 shows the average misprediction rate for each predictor at various history lengths. For history lengths of up to 30, the 2Bc-*gskew* mechanism delivers the best accuracy. After that point, piecewise linear branch prediction is the best.

Figure 5 shows the misprediction rate for each benchmark as well as the arithmetic mean. The chart shows the misprediction rates for each predictor using the history length that yields the best average misprediction rate for that predictor. The values of these best history lengths are given in the legend of the graph. Each of these predictors is one of the very best predictors in recent literature. Still, the piece-

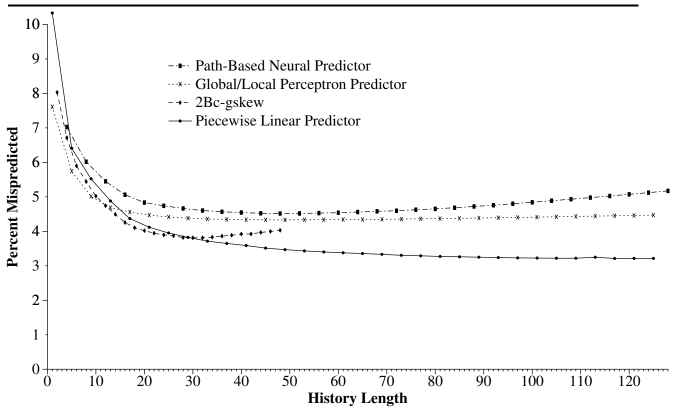


Figure 4. History lengths vs. misprediction rates

wise linear branch predictor manages to outperform them all. The next best predictor, 2Bc-*gskew*, gives an average misprediction rate of 3.81%. Piecewise linear branch prediction gives an average misprediction rate of 3.21%, a reduction of 16%. It reduces mispredictions by 26% over the third most accurate predictor, a global/local perceptron predictor, which yields a misprediction rate of 4.33%. Piecewise linear branch prediction gives the lowest misprediction rate on 13 out of the 15 benchmarks. It yields the second lowest misprediction rate on the other 2 benchmarks, and on those 2 benchmarks 2 different predictors are the best. Thus, without regard to implementation concerns, piecewise linear branch prediction is a consistently better prediction mechanism than any of the other predictors.

5. A Practical Piecewise Linear Branch Predictor

In this section, we present a practical version of piecewise linear branch prediction with implementation constraints similar to other practical predictors. There are two constraints on a practical predictor that the idealized version presented so far does not satisfy: limited area and limited latency.

5.1. Limiting Area of the W Array

The indices of the W array must be limited to keep them from exceeding the practical bounds of an implementable branch predictor. We limit the first two indices by taking them modulo two integers n and m. In a realistic implementation, n and m would be chosen as powers of 2 to make the modulo operation a simple mask. We limit the third index by choosing an appropriate value h for the history length. Thus, W becomes an $n \times m \times (h+1)$ 3-dimensional array of 8-bit weights. Limiting the second array index to $0..n-1$ also reduces the bits required to store the *GA* array since each address can be represented modulo n requiring at most $\log_2 n + 1$ bits.

5.2. Limiting Latency of the Prediction

Computing the output used to predict the branches requires adding $h + 1$ numbers each retrieved from different, possibly non-adjacent positions in W. We use ahead pipelining to mitigate the latency of this computation. Ahead pipelining means the computation is pipelined and begins before the branch to be predicted is fetched. It has been applied to traditional two-level branch predictors [8, 17] as well as neural branch predictors [9, 21].

5.3. An Ahead-Pipelined Piecewise Linear Branch Predictor

Branch predictions speculatively drive partial computations of the outputs of h future branches. The W array is now an $n \times m \times (h+1)$ 3-dimensional array of 8-bit weights. The bias weight for a branch with address B is now kept in $W[b \bmod n, b \bmod m, 0]$ providing a more uniform utilization of the now limited number of weights in W. The W array is indexed by the branch address modulo n. Since this index is not known ahead of time, the algorithm keeps n copies of the speculative predictor state to drive n possible predictions. Once the branch address becomes known, it is used to select one of the n predictions. Little extra speculative state is required for small values of n.

5.3.1. Extra State for the Ahead-Pipelined Predictor Ahead-pipelining the predictor requires additional state to store intermediate results of computations. R and SR are $n \times h$ 2-dimensional arrays of small integers used as intermediate storage for computing the output of the predictor. $SR[i, j]$ holds the speculative partial sum for predicting the j^{th} branch in the future whose address modulo n is i. Extending terminology from the original path-based neural predictor [9], SR and R are *shift matrices* composed of n shift vectors. SR is like a queue through which partial sums proceed. Partial sums enter the queue as 0, are added to while in the queue, and on exiting are added to the bias weight and used to compute the prediction. SR is speculative since it assumes the correctness of predictions for unresolved branches. R is a duplicate of SR that is maintained non-speculatively when a branch is resolved so that the predictor state can be restored on a misprediction. We have observed that 10 bits are sufficient for the elements of the R and SR arrays.

5.3.2. Making a Prediction Figure 6 shows the new prediction algorithm for piecewise linear branch prediction with ahead-pipelining. To predict a branch that will occur h branches in the future, the algorithm starts a partial sum for the output for that prediction with the value 0. Each time a branch is predicted, its result is used to add another term to the partial sum to predict h future branches. When a branch

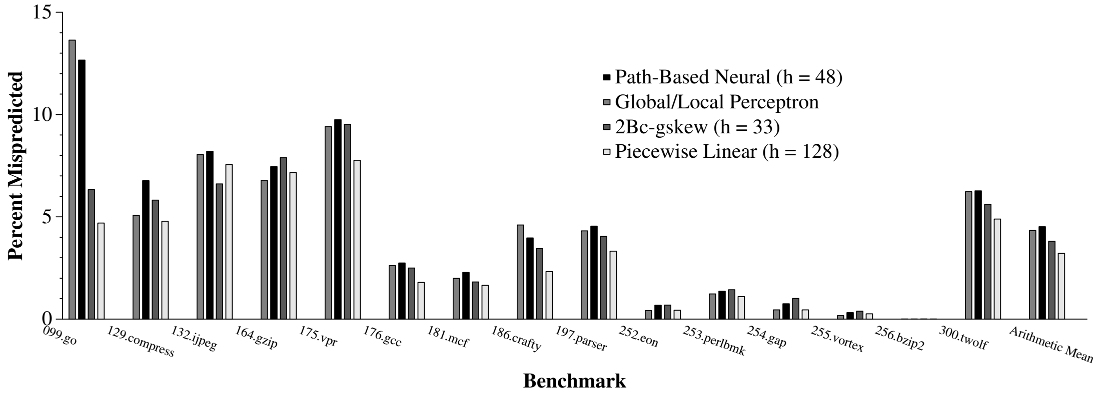


Figure 5. Misprediction rate per benchmark with the best history for each predictor

needs to be predicted, a partial sum is selected from one of n candidate partial sums and added to the bias weight for that branch. Thus, the critical path for making a prediction is a table lookup, a multiplexer, and an addition. This incurs a delay comparable to other branch predictors, in particular the ahead-pipelined path-based neural branch predictor that is estimated to have a delay of from 2 to 3 cycles [9].

5.3.3. Updating the Predictor The algorithm for updating the predictor when a branch is executed is similar to the algorithm for the idealized predictor described in Section 3 with an extra step. The non-speculative shift matrix R is updated in the update phase using the outcome of the branch. Updating R is similar to updating SR except that the branch address and outcome are non-speculative.

5.3.4. Misprediction Recovery On a misprediction, the speculative contents of SR must be overwritten with the non-speculative version kept in R. This copying can be accomplished with low latency in parallel to other recovery activities taking place in the processor, such as restoring the register file from non-speculative state. Indeed, we can think of the n vectors in SR as long registers in a special register file and apply known microarchitectural techniques to managing the speculative and non-speculative versions of register files. In Section 8, we see that the size of SR in terms of numbers of bits is no more than the size of a typical physical register file.

5.3.5. Implementation of Shift Matrix It has been observed that partial sums for computing neural branch predictor outputs require no more than 11 bits to represent each partial sum [9]. We find the 10 bits are sufficient for the piecewise linear branch predictor. Thus, an $n \times h$ shift matrix requires $n \times h$ 10-bit adders as well as $10 \times n \times h$ latches to store each partial sum. We propose an implementation that would use fast ripple-carry adders (RCA), which provide the best area and delay trade-off for small bit widths [5].

5.3.6. Implementation of W Weights for neural branch prediction require no more than 8 bits to provide high accuracy [12, 9]. The path-based neural predictor proposed using $h+1$ independently addressable tagless memories, each eight bits wide, to implement the W matrix. For the piecewise linear predictor, we propose using $h+1$ independently addressable tagless memories that are arranged as n 8-bit words wide. That is, W is organized as $h+1$ memories with n blocks each with m bytes. Thus, the training algorithm can update in parallel each of the $h+1$ weights responsible for predicting the branch.

5.3.7. Implementation of Parallel Algorithm The prediction algorithm requires a large number of operations to occur in parallel. On each cycle, the speculative shift matrix receives $n \times h$ results in parallel, so $n \times h$ 10-bit adders are required. For example, our most aggressive predictor design has $n = 8$ and $h = 51$, so it would require 408 10-bit adders.

The non-speculative version of the shift matrix, R, may be filled from the contents of the speculative version for right-path branches whose outcomes become known. This technique obviates the need for a second set of adders to maintain non-speculative results.

Each byte of a block in each of the $h+1$ independently addressable memories corresponds to a different combination of lower address bits. Accessing the entire block once provides input to every iteration of the $1..n$ loop in the algorithm for making a prediction. Thus, each memory is accessed only once for each prediction, and once again for each update.

6. Piecewise Linear Branch Prediction is a Generalized Neural Predictor

A happy consequence of parameterizing piecewise linear branch prediction with n and m is that the new predictor becomes a generalization of concepts found in previous neural predictors. The perceptron predictor [11] and path-

```
function predict (address: integer): boolean
begin
    i := address mod n                          (* First index in W and SR is the branch address modulo n *)
    j := address mod m                          (* Second index in W is the branch address modulo m *)
    output := SR[i,h] + W[i,j,0]                (* Complete the computation for this prediction *)
    if output ≥ 0 then
        predict := taken                        (* Predict taken if output is at least 0 *)
    else
        predict := not_taken                    (* Predict not taken otherwise *)
    end if
(* This point in the algorithm is the end of the critical timing path for making a prediction *)
(* The rest of the algorithm updates speculative state for making the next h predictions *)
    for i in 1..n in parallel do                (* For each of the shift vectors in SR... *)
        for k in 1..h in parallel do            (* For each partial sum in the i^th row of SR... *)
            a_k = h − k                         (* a_k is an index in the i^th shift vector *)
            if predict = taken then             (* If there is positive correlation between history *)
                SR'[i, a_k + 1] := SR[i, a_k] + W[i,j,k]   (* and outcome, then add the i, j, k^th weight to the *)
            else                                (* partial sum in SR *)
                SR'[i, a_k + 1] := SR[i, a_k] − W[i,j,k]   (* otherwise subtract instead of adding *)
            end if
        end for
        SR[i, 0..h] := SR[i, 0..h]'             (* Copy results of computations to SR *)
    end for
    for i in 1..n in parallel do                (* For each of the n speculative shift vectors, *)
        SR[i, 0] := 0                           (* reinitialize the first partial sum *)
    end for
end
```

Figure 6. Ahead-pipelined prediction algorithm

based neural prediction [9] turn out to be extreme ends of a family of parameterized piecewise linear branch predictors. Thus, piecewise linear branch prediction unifies previously distinct predictors and allows them to be studied in the same conceptual framework.

6.1. With $m = 1$, Piecewise Linear Prediction is the Perceptron Predictor

The perceptron predictor keeps an array of n weights vectors, each with $h + 1$ integer weights. One weight in each vector is a bias weight and the other h track correlation with branch outcome pattern history [11]. When a branch is predicted, the branch address modulo n is used to select a weights vector that is then used to compute the output of the predictor as the dot product of the weight vector and the branch history register.

In the piecewise linear branch predictor, if we let $m = 1$, then every bit in the GA array is 0. The resulting piecewise linear branch predictor is equivalent to a perceptron predictor with n perceptrons and a global history length of h. This is because letting $m = 1$ effectively removes the second index of W, so W collapses into a matrix whose n rows represent perceptron weights vectors and whose $h+1$ columns correspond to the bias weights and weights correlating with branch outcome pattern history.

6.2. With $n = 1$, Piecewise Linear Branch Prediction is Path-Based Neural Branch Prediction

The path-based neural predictor keeps an array of m weights vectors, each with $h + 1$ integer weights, again a bias and h weights to correlate with history. However, unlike the perceptron predictor, the path-based neural predictor uses the path history to select weights from up to $h + 1$ distinct weights vectors for each prediction [9].

In the piecewise linear branch predictor, letting $n = 1$ effectively removes the first index of the W and collapse W into an $m \times (h + 1)$ matrix of weights. The m rows correspond to the m weights vectors of the path-based neural predictor. The first column again corresponds to the bias weights for each branch address modulo m. The rest of the h columns correspond to the correlating weights chosen by path history. Since $n = 1$, only the global path history is used to choose weights to make a prediction, which is just what path-based neural prediction does.

6.3. Illustration of Generalized Predictor

Figure 7 shows the results of experiments performed using our limit study infrastructure for piecewise linear branch predictors with a constant history length of $h = 63$ and varying n and m such that their product is always 65,536. Thus, the number of elements in the W array is kept at a constant $64 \times 65,536$ = 4MB, but the first and second di-

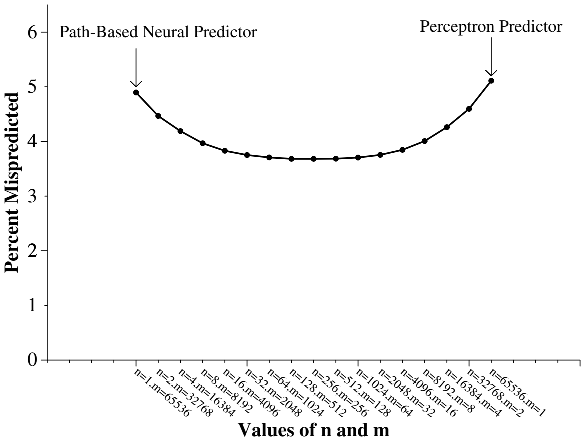


Figure 7. Misprediction rates with values of n and m whose product is a constant 64K.

mensions of the W array are varied to represent different points in the space of neural predictors. The misprediction rates are averaged across all 15 benchmarks.

At the left end of the graph, we have $n = 1$ and $m = 65,536$. This predictor is equivalent to path-based neural prediction with 64K weights vectors. It achieves a misprediction rate of 4.9%. At the right end of the graph, we have $n = 65,536$ and $m = 1$, equivalent to a perceptron predictor with 64K weights vectors. It achieves a misprediction rate of 5.1%. In between these extrema are different configurations of piecewise linear branch predictors. *Each of them outperforms both the perceptron predictor and path-based linear predictor.* Thus, the neural predictors so far presented in the literature are the two worst-case examples of a better predictor! The minimum misprediction rates are slightly below 3.7%, achieved when n and m are both between 128 and 512. With a value of $n = 8$ suitable for ahead-pipelined implementation, the misprediction rate is 3.9%.

7. Methodology for Performance Results

This section describes the experimental methodology for obtaining simulated results for the realistic, ahead-pipelined version of piecewise linear branch prediction using a cycle-accurate simulator.

7.1. Microarchitectural Framework

We use 15 SPEC CPU integer benchmarks running under MASE/Alpha [13], a significantly modified version of SimpleScalar/Alpha [4], a cycle-accurate out-of-order execution simulator that has been enhanced to include our branch predictors and to simulate overriding predictors at various latencies. We simulate all of the SPEC CPU 2000 integer benchmarks, and all of the SPEC CPU 95 integer benchmarks that are not duplicated in SPEC CPU 2000, except for `130.li` and `124.m88ksim` because they would not work under our checkpoint-based simulation framework. The benchmarks are compiled with the CompaQ GEM compiler with the optimization flags `-fast -O4 -arch ev6`.

We use SimPoint 1.1 to identify regions of the execution of each benchmark that characterize the entire run of the program on a given input. By analyzing statistics gathered from a functional simulation of the entire run of a benchmark on a given input, SimPoint finds *simulation points*, i.e., regions of 100 million instruction executions [18]. We simulate these regions with MASE, recording statistics such as instructions-per-cycle rates and branch misprediction rates. We then aggregate these numbers in a weighted average to give a precise estimate of what the statistics would have been if the benchmarks had been run to completion.

Parameter	Configuration
L1 I-cache	16 KB, 64B blocks, 2-way
L1 D-cache	16 KB, 64B blocks, 4-way
L2 unified cache	1MB, 128B blocks, 4-way
BTB	4096 entry, 2-way
Fetch/Decode/Issue/Commit	16 wide
Pipeline depth	40
Reorder buffer size	512
LSQ entries	128
L2 hit latency	7 cycles
L2 miss latency	500 cycles

Table 1. Microarchitectural parameters

Table 1 shows the base microarchitectural parameters used for the simulations. We started with a configuration loosely based on the Intel Pentium 4, but with an issue width of 16 and a deeper pipeline of 40 stages to provide a reasonable model of a future aggressively clocked microarchitecture. A recent study from Intel's Pentium Processor architecture group concludes that performance of aggressively clocked microarchitectures continues to improve until pipelines reach a depth of 52 [20]. Since that study was presented, Intel has increased the depth of its Pentium 4 pipeline from 20 to 31 stages in a microprocessor named Prescott available for purchase as of this writing. Thus, while our 40-stage pipeline is aggressive for current technology, it is conservative with respect to what is possible in future technologies.

7.2. Branch Predictors Simulated

We simulate the same predictors used in the limit study that appears earlier in this paper. We simulate realistic, resource-bounded versions of 2Bc-*gskew*, a global/local perceptron predictor, the path-based neural predictor, and the piecewise linear predictor. Since each predictor has a certain delay associated with it, even with ahead-pipelining, we use a two-level overriding organization [10] to mitigate predictor latency: A first-level 2K-entry bimodal predictor gives a prediction in a single cycle and instructions are fetched down the predicted path. If the second-level predictor disagrees with the initial prediction, the instructions fetched so far are dropped and fetching continues from the other path. We also simulate an oracle branch predictor that always predicts correctly as well as the idealized version of piecewise linear branch prediction.

7.3. Tuning The Predictors

Using the `train` inputs of the benchmarks along with SimPoint and trace-driven simulation, we find the history lengths that minimize the average misprediction rate for each hardware budget and branch predictor. We use these history lengths in the execution-driven simulations on the `ref` inputs. For 2Bc-*gskew*, we test history lengths exhaustively, keeping the lengths that results in the lowest aggregate misprediction rate. For the global/local perceptron predictor, we exhaustively tune the global history, keeping the local history at a constant 10 and the percentage of the hardware budget allocated to the local history tables at approximately 35%. Keeping the best global histories, we then tune the local histories exhaustively. For the path-based neural predictor, we tune the history length exhaustively. We use the formula $\theta = 2.14(h+1) + 20.58$ to set the threshold parameter θ in the path-based neural algorithm. We found this formula to give optimal accuracy at all history lengths in previous research.

For piecewise linear branch prediction, we tune history length as well as n and m, the moduli for the first and second indices of the W array, exhaustively. However, we constrain n to be a power of two such that the number of bits required to store each of the SR and R, matrices, i.e., $10 \times n \times h$, never exceeds 2,048. Thus, restoring the SR matrix from the R matrix is comparable to restoring a register file of 32 64-bit registers after a mis-speculation. We use the same formula for θ as for the path-based neural predictor. Table 2 shows the tuned history lengths for each hardware budget for each predictor as well as values for n and m for the piecewise linear predictor. For many of the neural predictors, the chosen history lengths and hardware budgets would lead to a number of weights vectors that is not a power of 2. We assume that a real design would use a power of 2 number of weights vectors for easy implementation. We present results with these configurations so that they may be compared against one another and against traditional table-based branch predictors using the same amount of state.

7.4. Estimating Branch Predictor Latency

We use CACTI 3.0 [19] to estimate the latencies of the various memories accessed by the predictors. We use HSPICE along with a custom logic design program to estimate the latency of the circuits used to compute the perceptron output for the perceptron predictor, the path-based neural predictor, and piecewise linear prediction. For reasons of limited space we omit details of this estimation except to say that it is similar to our previous work [9].

Hardware Budget	2Bc-*gskew*	global/local perceptron	path-based neural	piecewise linear h	n	m
4 KB	10	34/12	19	19	1	215
8 KB	11	34/12	19	19	2	176
16 KB	14	38/14	24	23	4	138
32 KB	15	40/14	31	26	8	118
64 KB	16	50/18	34	43	8	151
128 KB	17	54/19	38	50	8	288
256 KB	18	64/23	40	51	8	603

Table 2. Tuned history lengths for the predictors and values of n and m for piecewise linear prediction

8. Results from Microarchitectural Simulation

This section presents results of detailed microarchitectural simulation. Our main results compare the practical versions of piecewise linear branch prediction against practical versions of other branch predictors. We characterize the performance of these predictors using misprediction rates as well as instructions-per-cycle (IPC).

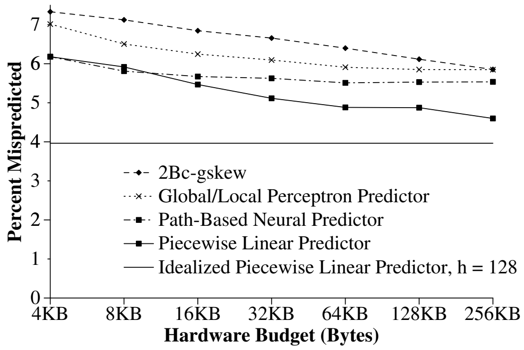


Figure 8. Average misprediction rates per hardware budget

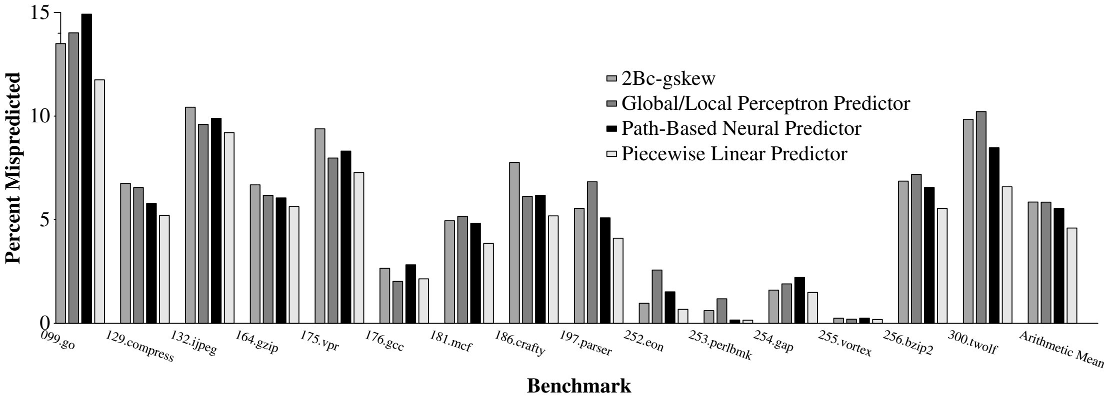


Figure 9. Average misprediction rates per benchmark with a 256KB hardware budget

8.1. Misprediction Rates

Figure 8 shows the arithmetic mean misprediction rate of each predictor over all 15 benchmarks at hardware budgets from 4KB to 256KB. It also shows the misprediction rate using the same cycle-accurate simulation methodology for the idealized piecewise linear predictor with a history length of 128. As the hardware budget is increased, the advantage of the piecewise linear predictor over the other predictors increases. On average, at a 32KB hardware budget, piecewise linear prediction mispredicts 5.1% of the time. That is 9% more accurate than path-based neural prediction which mispredicts 5.6% of the time and is the most accurate of the other predictors. At an aggressive 256KB hardware budget, piecewise linear prediction with a 4.6% misprediction rate is 16% more accurate than path-based neural prediction with a 5.5% misprediction rate, and 22% more accurate than 2Bc-*gskew* with a 5.9% misprediction rate. It is important to note that this improvement represents a departure from previous improvements in branch prediction accuracy – while all the other predictors in Figure 8 seem to approach an asymptotic misprediction rate of about 5.5%, piecewise linear prediction continues getting better. The idealized piecewise linear predictor with a history length of 128 achieves a misprediction rate of just below 4.0%. As we can see from the graph, the practical versions of this predictor reach within 15% of this limit. Indeed, the misprediction rate of piecewise linear branch prediction with a 256KB hardware budget is closer to the idealized predictor's misprediction rate than it is to the other predictors' rates.

Figure 9 shows the misprediction rates for 256KB branch predictors broken down by benchmark. Piecewise linear branch prediction yields the best accuracy for every benchmark except for `176.gcc`, on which it achieves a 2.1% misprediction rate compared with a 2.0% misprediction rate for the global/local perceptron predictor. On `252.eon` the improvement is particularly good: piecewise linear prediction has a misprediction rate of 0.67%, an improvement of 31% over 2Bc-*gskew* at 0.97%, 56% over path-based neural prediction at 1.52%, and 74% over the perceptron predictor at 2.57%.

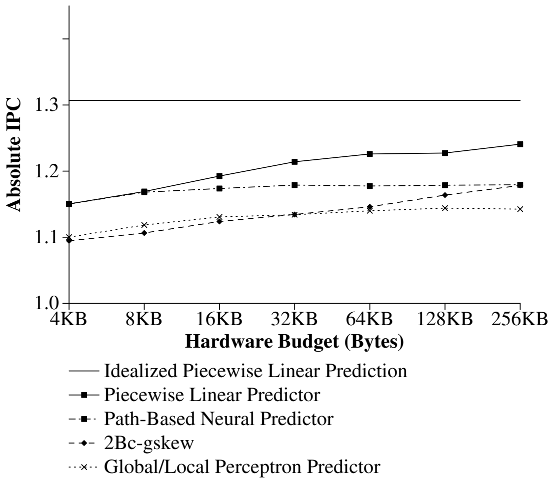


Figure 10. Instructions per cycle

8.2. IPC

Figure 10 shows a graph giving the IPC for each predictor at hardware budgets ranging from 4KB to 256KB. The graph shows the harmonic mean of the raw IPCs. Many factors independent of the branch predictor, such as locality, instruction mix, etc., affect the IPCs of benchmarks with respect to one another. To isolate the effect of the branch predictor on performance, Figure 11 shows the harmonic mean of IPCs that have been normalized with respect to the IPC given by an oracle branch predictor that always predicts directions and targets correctly. Again, as with misprediction rates, the IPCs given by piecewise linear prediction improve

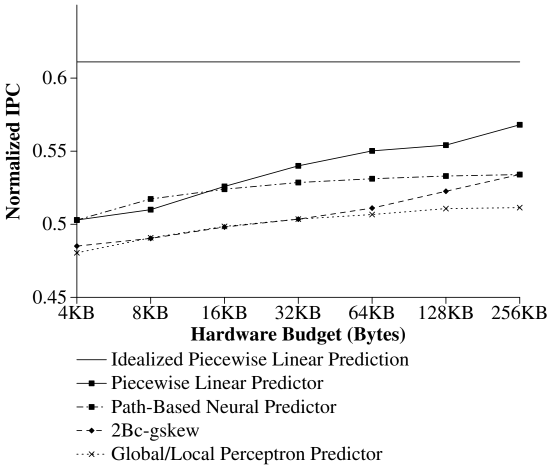


Figure 11. Normalized IPC

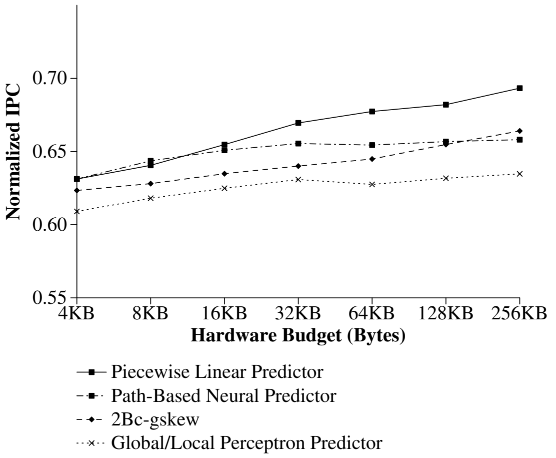


Figure 12. Normalized IPC with moderate pipeline depth

with respect to the other predictors as the hardware budget increases. At a 16KB hardware budget, piecewise linear prediction gives a speedup of 7% over 2Bc-*gskew*.

Figure 13 shows the normalized IPCs for each benchmark using branch predictors with a hardware budget of 256KB. Piecewise linear branch prediction outperforms the other branch predictors on every benchmark except for `175.vpr`, for which 2Bc-*gskew* yields a 2% speedup over piecewise linear prediction, most likely because of the higher latency of piecewise linear prediction compared with that of 2Bc-*gskew*, and `176.gcc`. Although there is a significant variance among the IPCs given by the various branch predictors, all of them but piecewise linear branch prediction yield a harmonic mean normalized IPC of no more than 0.53. Piecewise linear branch prediction gives a harmonic mean normalized IPC of 0.57, a speedup of 8% over the other predictors. In the case of `186.crafty`, piecewise linear branch prediction gives a speedup of 8% over path-based neural prediction, 20% over 2Bc-*gskew*, and 15% over the perceptron predictor.

8.3. Moderate Pipeline Depths

Figure 12 shows the normalized IPCs achieved by simulating the various predictors for a machine with a more moderate pipeline depth of 20 stages. At a hardware budget of 64KB, piecewise linear branch prediction yields an improvement of 5% over 2Bc-*gskew* and 4% over the path-based neural predictor. The gains are more modest because of reducing the pipeline depth reduces the penalty of mispredicted branches and also allow increases the relative impact of branch predictor latency, which is highest for the piecewise linear branch predictor.

9. Conclusions and Future Work

Our practical piecewise linear branch prediction generalizes previous work on neural branch prediction. In doing so, it expands the design space of neural predictors giving rise to more accurate predictors that yield significantly better performance.

In future work, we plan to improve upon piecewise linear branch prediction by finding ways to reduce the extra hardware it requires and reduce the latency. We also plan to generalize this idea further by incorporating, for instance, per-branch history information into the piecewise linear predictor.

The normalized IPCs in Section 8 demonstrate that there is ample room for improvement in the performance delivered by branch predictors. We believe piecewise linear branch prediction will be a driving force for achieving this improvement.

10. Acknowledgements

This research is supported by a grant from NSF, CCR-0311091 as well as a grant from the Ministerio de Educación y Ciencia (Spanish Ministry of Education and Science), SB2003-0357. While preparing the final version of this paper and the related presentation, the author has been on sabbatical leave at the Dept. de Arquitectura de Computadores at the Universidad Politécnica de Cataluña in Barcelona, Spain. The author gratefully acknowledges the generous support of this institution. Thanks to Mateo Valero for facilitating this stay.

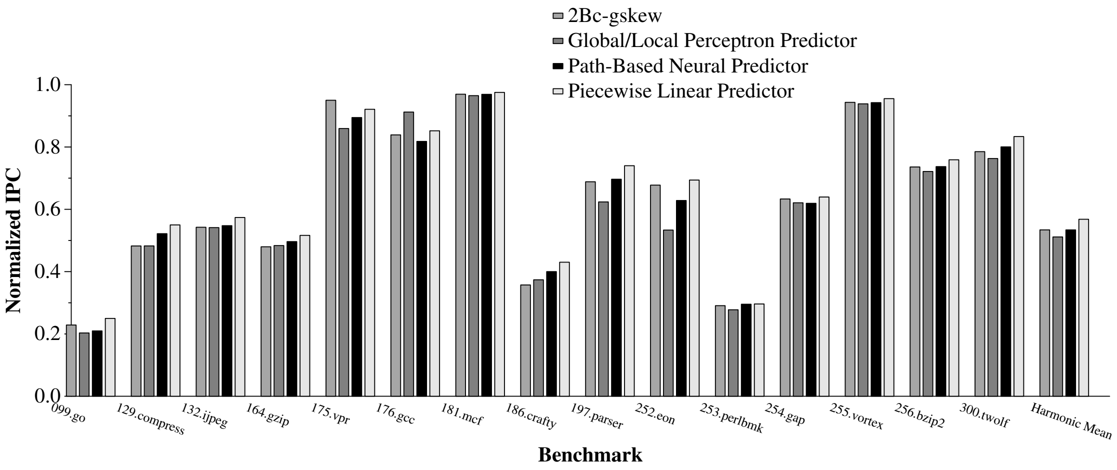


Figure 13. IPC per benchmark with a hardware budget of 256KB

References

[1] Haitham Akkary and Srikanth Srinivasan. Using perceptron-based branch confidence estimation for speculation control. In *Proceedings of the 10th International Symposium on High Performance Computer Architecture*, February 2004.

[2] H. D. Block. The perceptron: A model for brain functioning. *Reviews of Modern Physics*, 34:123–135, 1962.

[3] Edward Brekelbaum, Jeff Rupley, Chris Wilkerson, and Bryan Black. Hierarchical scheduling windows. In *Proceedings of the 35th International Symposium on Microarchitecture*, Istanbul, Turkey, December 2002.

[4] Doug Burger and Todd M. Austin. The SimpleScalar tool set version 2.0. Technical Report 1342, Computer Sciences Department, University of Wisconsin, June 1997.

[5] Henrik Eriksson, Tomas Henriksson, and Per Larsson-Edefors. Full-custom vs. standard-cell based design - an adder comparison. In *Proceedings of the 2002 Swedish System-on-Chip Conference*, March 2002.

[6] Marius Evers, Sanjay J. Patel, Robert S. Chappell, and Yale. N. Patt. An analysis of correlation and predictability: What makes two-level branch predictors work. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 52–61, July 1998.

[7] Ayose Falcón, Jared Stark, Alex Ramirez, Konrad Lai, and Mateo Valero. Prophet/critic hybrid branch prediction. In *Proceedings of the 31st annual international symposium on Computer architecture*, page 250, 2004.

[8] Daniel A. Jiménez. Reconsidering complex branch predictors. In *Proceedings of the 9th International Symposium on High Performance Computer Architecture (HPCA-9)*, pages 43–52, February 2002.

[9] Daniel A. Jiménez. Fast path-based neural branch prediction. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-36)*, pages 243–252. IEEE Computer Society, December 2003.

[10] Daniel A. Jiménez, Stephen W. Keckler, and Calvin Lin. The impact of delay on the design of branch predictors. In *Proceedings of the 33rd Annual International Symposium on Microarchitecture (MICRO-33)*, pages 67–76, December 2000.

[11] Daniel A. Jiménez and Calvin Lin. Dynamic branch prediction with perceptrons. In *Proceedings of the 7th International Symposium on High Performance Computer Architecture (HPCA-7)*, pages 197–206, January 2001.

[12] Daniel A. Jiménez and Calvin Lin. Neural methods for dynamic branch prediction. *ACM Transactions on Computer Systems*, 20(4):369–397, November 2002.

[13] Eric Larson, Saugata Chatterjee, and Todd Austin. MASE: A novel infrastructure for detailed microarchitectural modeling. In *Proceedings of the 2001 International Symposium on Performance Analysis of Systems and Software*, November 2001.

[14] Gabriel H. Loh and Dana S. Henry. Predicting conditional branches with fusion-based hybrid predictors. In *Proceedings of the 11th Conference on Parallel Architectures and Compilation Techniques*, pages 165–176, Charlottesville, Virginia, September 2002.

[15] Scott McFarling. Combining branch predictors. Technical Report TN-36m, Digital Western Research Laboratory, June 1993.

[16] André Seznec, Stephen Felix, Venkata Krishnan, and Yiannakakis Sazeides. Design tradeoffs for the Alpha EV8 conditional branch predictor. In *Proceedings of the 29th International Symposium on Computer Architecture*, May 2002.

[17] André Seznec and Antony Fraboulet. Effective ahead pipelining of instruction block address generation. In *Proceedings of the 30th International Symposium on Computer Architecture*, San Diego, California, June 2003.

[18] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. Automatically characterizing large scale program behavior. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 2002.

[19] Premkishore Shivakumar and Norman P. Jouppi. Cacti 3.0: An integrated cache timing, power and area model. Technical Report 2001/2, Compaq Computer Corporation, August 2001.

[20] Eric Sprangle and Doug Carmean. Increasing processor performance by implementing deeper pipelines. In *Proceedings of the 29th International Symposium on Computer Architecture*, pages 25–34, Anchorage, Alaska, May 2002.

[21] David Tarjan, Kevin Skadron, and M. Stan. An ahead pipelined alloyed perceptron with single cycle access time. In *Proceedings of the Workshop on Complexity Effective Design (WCED)*, June 2004.

[22] T.-Y. Yeh and Yale 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*, May 1993.