Improved Latency and Accuracy for Neural Branch Prediction

DANIEL A. JIMÉNEZ
Department of Computer Science
Rutgers University

Microarchitectural prediction based on neural learning has received increasing attention in recent years. However, neural prediction remains impractical because its superior accuracy over conventional predictors is not enough to offset the cost imposed by its high latency. We present a new neural branch predictor that solves the problem from both directions: it is both more accurate and much faster than previous neural predictors. Our predictor improves accuracy by combining path and pattern history to overcome limitations inherent to previous predictors. It also has much lower latency than previous neural predictors. The result is a predictor with accuracy far superior to conventional predictors but with latency comparable to predictors from industrial designs. Our simulations show that a path-based neural predictor improves the instructions-per-cycle (IPC) rate of an aggressively clocked microarchitecture by 16% over the original perceptron predictor. One reason for the improved accuracy is the ability of our new predictor to learn *linearly inseparable* branches; we show that these branches account for 50% of all branches and almost all branch mispredictions.

Categories and Subject Descriptors: C.1.1 [**Processor Architectures**]: Single Data Stream Architectures

General Terms: Performance

Additional Key Words and Phrases: Branch prediction, machine learning

1. INTRODUCTION

Branch misprediction latency is the most important component of performance degradation as microarchitectures become more deeply pipelined [Sprangle and Carmean 2002]. Branch predictors must improve to avoid the increasing penalties of mispredictions. Branch predictors based on neural learning are the most accurate predictors in the literature [Loh and Henry 2002; Jiménez and Lin 2002], but they are impractical because the advantage of the extra

This research was supported by National Science Foundation (NSF) grant CCR-0311091.
Author's address: Department of Computer Science, Rutgers University, 110 Frelinghuysen Rd., Piscataway, N.J. 08854; email: djimenez@cs.rutgers.edu.
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or direct commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 1515 Broadway, New York, NY 10036 USA, fax: +1 (212) 869-0481, or permissions@acm.org.
© 2005 ACM 1529-3785/05/0100-0197 $5.00

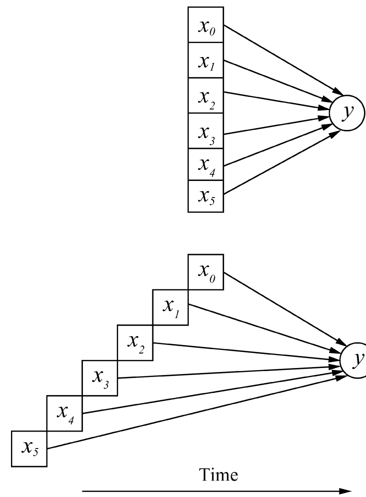

Fig. 1. Rather than being done all at once (above), computation is staggered (below).

accuracy is nullified by high access latency, even when latency-sensitive predictor organizations are used [Jiménez 2002]. This latency is due primarily to the complex computation that must be carried out to determine the excitation of an artificial neuron.

We present a practical neural branch predictor. Its latency is much lower than previous designs and is comparable to that of conventional predictors used in industrial designs, making it practical for implementation in a high-frequency microprocessor. At the same time, its accuracy is superior to that of previous highly accurate predictors.

1.1 Better Latency through Path-Based Prediction

Figure 1 illustrates how our new predictor achieves low latency by beginning well ahead of time. The predictor staggers computations in time, predicting a branch using a neuron selected dynamically along the path to that branch, rather than selecting the neuron all at once based solely on the branch address. A happy side-effect of this selection process is improved accuracy because the predictor is able to correlate with path history as well as pattern history.

An interesting result of the selection process is that this new predictor becomes a non-linear classifier, able to predict a wider range of branch behavior to achieve better accuracy than previous linear neural branch predictors.

We show that our path-based neural predictor has a misprediction rate 7% lower than that of the original perceptron predictor, and because of its improved latency it delivers an IPC 16% higher than that predictor at a 64 KB hardware budget.

1.2 Better Accuracy for Linearly Inseparable Branches

Linear classifiers such as the perceptron predictor [Jiménez and Lin 2001, 2002] use a set of weights to define a flat surface or *hyperplane* that divides the space of branch histories into "predict taken" and "predict not

taken." Such predictors can only learn to predict *linearly separable* branches whose histories are separable by such a hyperplane. However, over half of all branches cannot be characterized by a single hyperplane and are thus *linearly inseparable*.

Our new path-based neural predictor uses a slightly different set of weights, and thus a different hyperplane, for each distinct path leading to the predicted branch, giving it the ability to fully learn linearly inseparable branch behavior. A branch may be linearly inseparable as a whole, but it may be piecewise linearly separable with respect to the distinct program paths. In other words, the path-based neural predictor combines path history with pattern history, resulting in superior learning than a neural predictor that relies only on pattern history.

1.3 Contributions

This article makes the following contributions:

(1) We present a path-based neural predictor with a latency comparable to a conventional branch predictor. By contrast, previous neural predictors have a very high latency relative to other branch predictors, which severely diminishes their capacity to improve performance [Jiménez 2002]. Thus, the path-based neural predictor can replace a less accurate conventional branch predictor and guarantee improved performance.

(2) We show that *linearly inseparable* branches account for over half of all branches and over 90% of mispredictions for most benchmarks and predictors we studied. The path-based neural predictor can learn to predict linearly inseparable branches that previous predictors cannot. Thus, the new predictor is more accurate than previous predictors.

(3) The perceptron predictor must use a long history length to achieve high accuracy; with equivalent history lengths, it is less accurate than *gshare* [Jiménez and Lin 2001]. The path-based neural predictor predictor achieves superior accuracy as shorter history lengths. Since the hardware budgets of both predictors are proportional to history length, the path-based neural predictor can deliver superior accuracy with a lower hardware budget.

This article is organized as follows: Section 2 briefly discusses related work. Section 3 gives background in neural branch prediction and explains the new prediction algorithm. Section 4 describes our experimental methodology. Section 5 gives the accuracy and performance results of our experiments. Section 6 gives the results of more experiments explaining why the new predictor works well. Finally, Section 7 concludes the article.

2. RELATED WORK

2.1 Neural Prediction

Calder et al. [1995] use neural networks to perform static branch prediction at compile time. Features such as control-flow information are used to train

a neural network to distinguish between branches that are likely to be biased taken from branches that are likely to be biased not taken. This approach achieves a 20% misprediction rate, ompared to 25% for static heuristics [Ball and Larus 1993; Calder et al. 1995].

Dynamic branch prediction with neural methods was first proposed by Vintan and Iridon [1999] who explore the use of learning vector quantization, a neural method. The resulting branch predictor achieves an accuracy comparable to a table-based branch predictor. This method does not lend itself well to high-speed implementation because it performs complex computations involving floating point numbers.

2.1.1 *Branch Prediction with Perceptrons.* The original perceptron predictor [Jiménez and Lin 2001] uses a simple linear neuron known as a perceptron [Block 1962] to perform branch prediction. Perceptrons are vectors of weights trained with a machine learning algorithm such that the sign of the dot product of the weights vector and an input vector classifies the input vector into one of two classes. In the case of branch prediction, the classes are *taken* and *not taken*. Perceptrons achieve better accuracy than two-level adaptive branch prediction because of their ability to exploit long history lengths which have been shown to provide additional correlation for branch predictors [Evers et al. 1998]. Another study suggests ways to implement the predictor using techniques from high-speed arithmetic [Jiménez and Lin 2002], but the latency of the predictor is more than 4 cycles with an aggressive clock rate. Despite its drawbacks, neural prediction has been suggested as a promising technology for future microprocessors [Seznec et al. 2002]. It has become part of one of Intel's IA-64 simulators for researching future microarchitectures [Brekelbaum et al. 2002]. It has been used as a component in studies of hybrid predictors [Loh and Henry 2002; Thomas et al. 2003] and is the most accurate single-component branch predictor in the literature [Loh and Henry 2002; Jiménez and Lin 2002]. The path-based neural branch predictor described in this paper was presented at the 36th International Symposium on Microarchitecture [Jiménez 2003].

2.2 Path-Based Prediction

Our path-based neural predictor achieves superior accuracy and low latency by choosing the neural weights based on the path taken to reach a branch rather than the branch address itself. Branch outcomes are highly correlated both with path and pattern histories [Nair 1995; Stark et al. 1998]. Previous work has also explored the use of path information to improve branch predictor accuracy. For instance, the variable length path branch predictor [Stark et al. 1998] computes a hash of past branch addresses to form an index into a table of counters. It chooses the hash function based on information gathered through a feedback-directed procedure that requires profiling runs.

2.3 Latency-Sensitive Prediction

As hardware budgets for branch predictors expand, research has begun to focus on balancing the tradeoff between accuracy and latency important for large

predictors with high latencies. Jiménez et al. [2000] survey several techniques for mitigating branch predictor delay. The most common technique is *overriding*, in which a quick but relatively inaccurate predictor guides instruction fetch in a single cycle, and may be corrected by a slower but more accurate multi-cycle predictor. This approach was used for the Alpha EV6 and EV7 cores [Kessler 1999] and was proposed for the Alpha EV8 [Seznec et al. 2002]. The overriding technique does not scale well as branch predictor latency increases because the penalty for an overriding event becomes substantial [Jiménez 2002].

Other studies propose pipelined branch predictors [Stark et al. 1998; Jiménez 2002; Seznec and Fraboulet 2003] to mitigate latency. The main source of latency for most large branch predictors is the access delay to the memories used to implement the pattern history tables. The latency of the perceptron predictor is dominated by computation time.

3. A PATH-BASED NEURAL PREDICTOR

In this section, we review the relevant details of previous work on neural branch prediction. In this context, we give the intuition behind the path-based neural predictor. We then give a detailed explanation of the path-based neural predictor.

3.1 Branch Prediction with Perceptrons

The perceptron predictor uses perceptron learning [Rosenblatt 1962; Block 1962] to predict the directions of conditional branches [Jiménez and Lin 2001, 2002]. We review the design of the perceptron predictor, describing algorithms using an Algol-like pseudocode with keywords in **boldface** and comments in *italics*. We use *taken* and *not_taken* as meaningful names for Boolean constants.

Throughout this article, we use a notation in which brackets signify vector or matrix indexing, for example, $W[i, j]$ is the i, jth element of the matrix W. Subscripts indicate a member of a sequence, for example, b_i is the ith of a sequence labeled b.

The perceptron predictor is similar to other predictors in that it keeps a global history shift register that records the outcomes of branches as they are executed, or speculatively as they are predicted. The width of this register is the history length for the predictor, hereafter referred to as h.

The perceptron predictor keeps an $n \times (h + 1)$ matrix $W[0..n - 1, 0..h]$ of integer *weights*, where n is a design parameter. Weights are 8-bit bytes. Each row of the matrix is an $(h+1)$-length *weights vector*. Each weights vector stores the weights of one perceptron that is controlled by perceptron learning. In a weights vector $w[0..h]$, the first weight, $w[0]$, is known as the *bias weight*. Thus, the first column of W contains the bias weights of each weights vector. The Boolean vector $G[1..h] \in \{1..h\} \times \{taken, not_taken\}$ represents the global history shift register.

3.1.1 *Prediction and Update Algorithms.*

Figure 2 gives pseudocode for the prediction and update algorithms for the original perceptron predictor. The *prediction* algorithm returns a Boolean value predicting the branch at address *pc*.

```
function prediction (pc: integer): { taken , not_taken };
begin
    (* Hash the pc to select a row of W *)
    i := pc mod n
    (* Compute output of i^th perceptron using G[1..h] as input *)
    y_out := W[i,0] + sum_{j=1}^{h} { W[i,j]   if G[j] = taken
                                     { -W[i,j]  if G[j] = not_taken
    (* Make the prediction based on the sign of y_out *)
    if y_out >= 0 then
        prediction := taken
    else
        prediction := not_taken
    end if
end

procedure train (i, y_out: integer; prediction , outcome : { taken , not_taken });
    (* If incorrect or y_out below threshold then adjust weights *)
    if prediction ≠ outcome or |y_out| <= θ then
        (* Increment bias weight if taken, decrement if not taken *)
        W[i,0] := W[i,0] + { 1   if outcome = taken
                           { -1  if outcome = not_taken
        for j in 1..h in parallel do
            (* Increment j^th weight for positive correlation,
            (* decrement for negative correlation *)
            W[i,j] := W[i,j] + { 1   if outcome = G[j]
                               { -1  if outcome ≠ G[j]
        end for
    end if
    (* Update the global history shift register *)
    G := (G << 1) or outcome
end
```

Fig. 2. Perceptron prediction and update algorithm.

When a branch outcome becomes known, the *train* algorithm is invoked to update the predictor. The training algorithm takes an integer parameter θ that controls the trade-off between long-term accuracy and the ability to adapt to phase behavior. It has been empirically determined that choosing $\theta = \lfloor 1.93h + 14 \rfloor$ gives the best accuracy [Jiménez and Lin 2002]. Thus, θ is a constant for a given history length. Once the outcome of a branch becomes known, the following algorithm is used to update the perceptron predictor, taking as parameters the outcome as well as the values of i, *prediction*, and y_{out} computed during the prediction phase.

From the algorithm we can see that the bias weight is incremented (decremented) if the branch is taken (not taken), while the rest of the weights are incremented (decremented) if the branch outcome is equal (not equal) to the corresponding bit of the global history shift register.

3.1.2 *Implementation.* We review some of the suggestions for a practical implementation of the perceptron predictor.

The matrix W should be implemented as a tagless direct-mapped memory of n blocks with the ith block containing h 8-bit weights that form the weights

vector of the ith perceptron. Thus, each time a prediction is needed, the weights vector corresponding to that value of *pc* is read from memory.

Instead of negating the weights to produce summands for the computation of y_{out}, they can be bitwise complemented with very little impact on accuracy. This speeds the computation of the summands.

The computation of y_{out} can be arranged as a Wallace-tree [Cormen et al. 1990] adder to add the summands. This allows the circuit performing this computation to have a depth of $O(\log h)$ gate delays, as opposed to $O(h)$ gate delays with a naive summing algorithm.

Two global shift history registers should be kept: a speculative one that is updated with predictions, and a nonspeculative one that is updated when a branch completes. The speculative history is used for all predictions, while the non-speculative history is used to correct the speculative one on a misprediction.

3.1.3 *Disadvantage of the Perceptron Predictor.*

The main disadvantage of the perceptron predictor is its high latency. Even using the high-speed arithmetic tricks mentioned above, the latency of the computation of y_{out} is high relative to the clock period of a deeply pipelined microarchitecture. It has been shown that performance is highly sensitive to high-branch predictor latency [Jiménez et al. 2000], even when special techniques are used to mitigate latency [Jiménez 2002].

The latency has been estimated at 4 cycles for a small version of the predictor [Jiménez and Lin 2002]. Our simulations, detailed in Section 4, show that the latency would be 6 cycles for a version of the predictor capable of achieving the same accuracy as the 2Bc-*gskew* predictor that was proposed for the Alpha EV8 [Seznec et al. 2002], with a latency of only 2 cycles.

3.2 A Path-Based Neural Predictor

Our alternative to the perceptron predictor is a neural predictor that chooses its weights vector according to the path leading up to a branch, rather than according to the branch address alone. This technique has two advantages. First, latency is mitigated because computation of y_{out} can begin in advance of the prediction, with each step proceeding as soon as a new element of the path is executed. Second, accuracy is improved because the predictor incorporates path information into the prediction.

3.2.1 *Intuitive Description.*

Our new predictor has much the same structure as the perceptron predictor. It keeps a matrix W of weights vectors. Each time a branch is fetched and requires a prediction, one of the weights vectors from W is read. However, only the 0th weight, that is, the bias weight, is used to help predict the current branch. Its value is added to a running total that has been kept for the last h branches, with each summand added during the processing of a previous branch.

Figure 3 illustrates the difference between the perceptron predictor (a) and our new predictor (b). The diagrams show the progress of the two predictors predicting a sequence of branches labeled b_{t-7} through b_t with b_t fetched most recently. The vertical columns correspond to the rows of W accessed at each

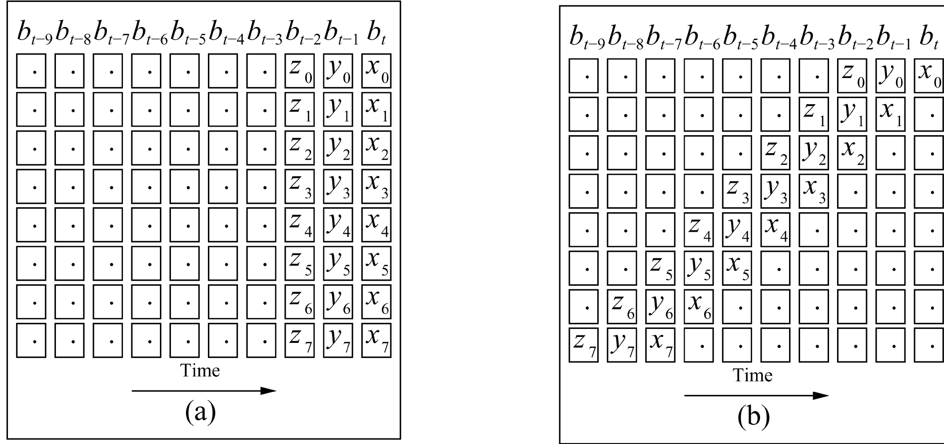

Fig. 3. Illustration of the weights used to predict branch b_t with the perceptron predictor (a) and the path-based neural predictor (b) with history length of 7. Vertical columns are weights vectors.

time step. Each predictor has a history length $h = 7$. For each predictor, the set of weights used to predict branch b_t is $x[0..7]$. For the perceptron predictor (a) at time t, the vector is accessed, each weight processed into a summand, and y_{out} computed all at once.

By contrast, the $x[0..7]$ weights for the new predictor (b) are built up by accessing different positions in the weights vectors associated with branches b_{t-7} through b_t. For the new predictor, a running total (not shown) is kept of the summands in the computation of y_{out}. By time t, the only summand left to be added is the bias weight, $x[0]$. Note that the prediction generated for branch b_{t-i} is the ith most recent speculative history bit for branch b_t, so the relevant parts of the speculative global history shift register become available as they are needed. To further clarify the intuition, Figure 3 also shows the positions of the weights $y[0..7]$ and $z[0..7]$ used to compute the predictions for the previous two branches b_{t-1} and b_{t-2}.

Another way to see the difference between the two predictors is to look at which weights are used to predict which branches. In the original perceptron predictor, each of the weights in the weights vector associated with branch b_j is used to predict branch b_t. In the new predictor, the ith weight in the weights vector associated with branch b_j is used to predict branch b_{j+i}.

3.2.2 *The Prediction Algorithm.* Figure 4 shows a parallel algorithm for predicting the current branch and updating computations for predicting the next h branches. Let W, n, and h be defined as before. Let $SR[0..h]$ and $R[0..h]$ be vectors of $h + 1$ integers. The first column of W form the bias weights. $SR[h - j]$ contains the running total computing the perceptron output that will be used to predict the jth branch after the current one. SR is updated speculatively, so R, used in the updating algorithm described later, holds the most up-to-date non-speculative version of SR. Think of SR as a queue that holds the partial sums for the perceptron output computation as they are being computed. A zero enters the tail of the queue at $SR[0]$ and the perceptron output, minus the bias

```
function prediction (pc: integer) : { taken , not_taken }
begin
    (* Hash pc to produce i, a row number in W *)
    i := pc mod n
    (* y is the partial sum from the last prediction plus the bias weight *)
    y := SR[h] + W[i, 0]
    (* The prediction is the complement of the sign bit of y *)
    if y ≥ 0 then
        prediction := taken
    else
        prediction := not_taken
    end if
    (* Update the next h partial sums *)
    for j in 1..h in parallel do
        (* SR[k_j] is the partial sum for predicting the j^th next branch *)
        k_j := h − j
        (* Do the next step in the perceptron output computation for the j^th
        next branch, speculating that this prediction is correct, and shifting each
        partial sum to the next position. Use the current prediction as the j^th
        most recent history bit for the j^th next branch *)
        if prediction = taken then
            SR'[k_j + 1] := SR[k_j] + W[i, j]
        else
            SR'[k_j + 1] := SR[k_j] − W[i, j]
        end if
    end for
    (* Copy the resulting partial sums into SR *)
    SR := SR'
    (* Initialize the partial sum for h^th next branch *)
    SR[0] := 0
    (* Shift the prediction into the speculative global history *)
    SG := (SG << 1) or prediction
end
```

Fig. 4. Path-based neural prediction algorithm to predict branch at address *pc*.

weight, emerges at $SR[h]$. SG and G are shift registers that hold speculative and nonspeculative global history, respectively.

3.2.3 *Update Algorithm.* Updating the path-based neural predictor is conceptually similar to updating the original perceptron predictor. However, the new update algorithm has to deal with the fact that each weights vector is associated with h branches, rather than one branch as in the original predictor. When branch b_t completes and its outcome is ready to be used to update the predictor, most of the weights vector associated with b_t cannot be updated because they are being used to predict future branches that have not completed yet. Thus, we design the matrix W as $h + 1$ independently addressable high-speed memories, each representing the n weights of a single column of W. When the predictor is updated the corresponding weights can be accessed independently. The memory with the bias weights are kept closest to the logic that computes the final y_{out} value for low latency.

Figure 5 gives a parallel algorithm for updating the path-based neural predictor. It accepts as parameters the values of i and y_{out} computed during the

```
procedure train ( i, y_out: integer; prediction , outcome : { taken , not_taken },
    v: array [1..h] of integer; H: array [1..h] of { taken , not_taken });
begin
    (* If incorrect or y_out below threshold then adjust weights *)
    if prediction ≠ outcome or |y_out| ≤ θ then
        (* Update the bias weight *)
        W[i,0] := W[i,0] + { 1   if outcome = taken
                           { −1  if outcome = not_taken
        (* k is the row in W whence came the j^th weight for this prediction *)
        for j in 1..h in parallel do
            k := v[j]
        (* Increment j^th weight for positive correlation, decrement for negative correlation *)
            W[k,j] := W[k,j] + { 1   if outcome = H[j]
                               { −1  if outcome ≠ H[j]
        end for
    end if
    (* Update non-speculative global history shift register *)
    G := (G << 1) or outcome
    (* Restore speculative history and shift vector on a misprediction
    from non-speculative sources (not shown) *)
    if prediction ≠ outcome then
        SG := G
        SR := R
    end if
end
```

Fig. 5. Path-based predictor update algorithm.

prediction algorithm as well as the Boolean outcome of the branch, a vector H representing the value of the speculative global history shift register *SR* when the branch was predicted, and an array $v[1..h]$ of integers representing the addresses of the last h branches predicted modulo n. That is, $v[i]$ is the index of the row in W used for predicting the ith most recent branch instruction. This array can be implemented as a small circular buffer global to all invocations of the training procedure with speculative and non-speculative versions as with the prediction algorithm. Note that the address modulo n was computed in the prediction algorithm, so it can be recorded in the circular buffer at that time. Also, the modulo operation need not be expensive: it is simply a masking operation if the number of weights vectors is chosen to be a power of two.

Some of the details of these algorithms have been omitted for clarity and brevity, for example, details the maintenance of the circular buffer of weights vector indices and the maintenance of the contents of R, which is simply a nonspeculative copy of the circuitry that maintains *SR*. A detailed Java implementation of the algorithm will be made available upon request.

3.2.4 *Recovery After Misprediction.* When the path-based neural predictor predicts incorrectly, the *SR* vector is restored to the value stored in R during the predictor update for the last committed branch. Since all of the branches up to the last committed branch were correctly predicted and committed in-order, the restored value of *SR* is as it was when the mispredicted branch was fetched, and prediction will continue normally. The recovery takes less than one cycle,

and its latency is completely hidden by the latency of other actions taken by the microarchitecture to recover from the misprediction.

3.2.5 *Area and Latency.* Clearly, the prediction algorithm uses a slower method for computing y_{out} than the original perceptron method. However, since it begins the summation process h branches before the prediction is needed, *the latency is almost completely hidden*. The only elements on the critical path to making a prediction are reading the bias weight and adding it to the current partial sum (i.e., $SR[h]$). This is much faster than computing y_{out} all at once with a Wallace-tree and also consumes less area. The Wallace-tree for the original perceptron predictor has $O(h \log h)$ carry-save adders as well as a carry-lookahead adder for the final addition, while the new algorithm requires only $O(h)$ independent adders for updating SR at each prediction step. For reasonable-sized predictors and history lengths, we estimate that the path-based neural predictor would take approximately two clock cycles to produce a prediction given a branch address. This is the same latency tolerated by branch predictors from industrial designs [Seznec et al. 2002]. We give details of these estimates later in Section 4.

4. METHODOLOGY

In this section, we describe our experimental methodology for evaluating the path-based neural predictor.

4.1 Microarchitectural Framework

We use 17 SPEC CPU integer benchmarks running under a version of SimpleScalar / Alpha [Burger and Austin 1997], a cycle-accurate out-of-order execution simulator that has been enhanced to include our branch predictors, simulate overriding predictors at various latencies, and simulate deep pipelines. 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. The benchmarks are compiled with the CompaQ GEM compiler with the optimization flags `-fast -O4 -arch ev6`.

Table I describes each of the benchmarks.

To better capture the steady-state performance behavior of the programs, our experiments skip the first billion instructions, as several of the benchmarks have an initialization period lasting fewer than one billion instructions during which program behavior is not characteristic of the many billions of subsequent instructions. After skipping those instructions, each benchmark executes 500 million instructions on the `ref` inputs before the simulation ends.

Table II shows the base microarchitectural parameters used for the simulations. We started with a configuration loosely based on the Intel Pentium 4, with a deeper pipeline of 32 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 [Sprangle and Carmean 2002]. Thus, while our 32-stage pipeline is

Table I. Description of SPEC CPU Integer Benchmarks

Benchmark	Description
`099.go`	Plays the game of *go*. Pattern matching.
`124.m88ksim`	Simulator for the Motorola 88100 microprocessor.
`129.compress`	Compresses files with Lempel-Ziv adaptive encoding.
`130.li`	Lisp interpreter running the Gabriel benchmarks.
`132.ijpeg`	Compression/decompression for JPEG images.
`164.gzip`	Compresses files with Lempel-Ziv coding.
`175.vpr`	Placement and routing program for FPGAs.
`176.gcc`	C compiler (gcc 2.7.2.2) for the Motorola 88100.
`181.mcf`	Single-depot scheduling for mass transportation.
`186.crafty`	Plays chess using alpha-beta search.
`197.parser`	Parses English text to produce grammar analysis.
`252.eon`	Probabilistic ray tracing program.
`253.perlbmk`	Stripped-down version of Perl v5.005_03.
`254.gap`	Language for group-theoretic computation.
`255.vortex`	Object-oriented database program.
`256.bzip2`	Compresses files with block-sorting compression.
`300.twolf`	Standard-cell placement and routing.

Table II. Microarchitectural Parameters

Parameter	Configuration
L1 I-cache	16 KB, 6 4B blocks, 2-way
L1 D-cache	8 KB, 64B blocks, 4-way
L2 unified cache	512 KB, 128B blocks, 8-way
BTB	4096 entry, 2-way
Issue width	8
Pipeline depth	32
RUU entries	128
LSQ entries	128
L2 hit latency	7 cycles
L2 miss latency	200 cycles

aggressive for current technology, it is conservative with respect to what is possible in future technologies.

We simulate extra pipeline stages beyond the 5 provided by `sim-outorder` by adding an extra stages that simply buffer instructions from the fetch stage to the decode stage. Thus, our modeling of wrong-path effects is conservative with respect to branch prediction studies, since misspeculated loads that miss in the data cache can only be issued near the final stage.

4.2 Branch Predictors Simulated

We simulate the following predictors to compare with the path-based neural predictor:

4.2.1 *2Bc-gskew*. We simulate a 2Bc-*gskew* predictor, which is a McFarling-style [McFarling 1993] hybrid predictor combining a bimodal predictor with an *egskew* predictor that predicts using the majority prediction of three components: the bimodal predictor and two *gshare*-like predictors

indexed by special hash functions so as to minimize the chance that both predictors will suffer destructive interference at the same time. A version of this predictor would have been used in the Alpha EV8 processor [Seznec et al. 2002]. In our latency-sensitive simulation, 2Bc-*gskew* takes more than one cycle to return a result. We use a two-level overriding organization [Jiménez et al. 2000] to mitigate this 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 2Bc-*gskew* predictor disagrees with the initial prediction, the instructions fetched so far are dropped and fetching continues from the other path. This technique closely reflects the design of the EV8 predictor, in which 2Bc-*gskew* overrides a less accurate instruction cache line predictor.

4.2.2 *Perceptron Predictor.* We simulate a recent [Jiménez and Lin 2002], highly accurate version of the perceptron predictor that combines global and per-branch history information in a manner reminiscent of the alloyed branch predictors of Skadron et al. [2000] and Jiménez and Lin [2002]. We again use an overriding organization with a first-level 2K-entry bimodal predictor, this time backed up with a second-level perceptron predictor. We note that this predictor has been shown to be more accurate than even the most aggressive multicomponent hybrid predictor [Jiménez and Lin 2002]. Thus, including other combined global and per-branch hybrid predictors in this study would be superfluous.

4.2.3 *gshare.fast.* We simulate a specialized version of the *gshare* predictor that has been pipelined to return a result in a single cycle. By using older branch history to prefetch a portion of the pattern history table in a previous cycle and then using the exclusive-OR of more recent history and the low bits of the current branch address to select from that portion, *gshare.fast* has an effective latency of one cycle [Jiménez 2002]. It has been shown to yield higher instruction per cycle rates than highly accurate predictors such as 2Bc-*gskew* and the perceptron predictor at large hardware budgets [Jiménez 2002]. For this study, our simulation of *gshare.fast* is idealized, assuming that there is no overlap or missing gap between the older history and more recent history.

4.2.4 *Fixed-Length Path Predictor.* We simulate a fixed length path branch predictor that forms a hash of the history of branch target addresses leading up to the branch to be predictor [Stark et al. 1998]. The hash function XORs the addresses, first rotating each address by a number of bits equal to it position in the branch history. The hash is used to index a table of two-bit saturating counters as in a two-level scheme. We use the same fixed length for each benchmark, as opposed to using a variable-length path branch predictor which requires expensive profiling [Stark et al. 1998]. (Note that none of the schemes used for this article require profiling.)

4.2.5 *Path-Based Neural Predictor.* We simulate the path-based neural predictor as described above, using an overriding organization with a first-level 2K-entry bimodal predictor as with the other overriding predictors.

Each simulated predictor is pipelined so that it can be accessed on every cycle, for example, for a predictor with a latency of two cycles, the prediction

Table III. Tuned History Lengths

Hardware Budget	Fixed Length Path	2Bc-*gskew*	Global/ Local	Path-Based Neural
1 KB	10	10	25/9	13
2 KB	10	10	31/11	18
4 KB	12	10	34/12	20
8 KB	15	11	34/12	32
16 KB	20	14	38/14	34
32 KB	20	15	40/14	34
64 KB	20	16	50/18	37

requested two cycles ago is available in the current cycle. Each predictor's history registers are updated speculatively and corrected on a misprediction. The neural predictors, that is, perceptron and path-based, use 8-bit weights. This number of bits was chosen empirically.

4.3 Multiple Branch Prediction

As described, the perceptron predictor and path-based neural predictor can only predict one branch in a fetch group. Although multiple branch prediction will be addressed in future work, we currently make no provision for predicting more than one branch with the neural predictors.

We simulate 2Bc-*gskew* and *gshare.fast* as multiple branch predictors. They are allowed to return up to eight predictions for a single fetch group. Each prediction uses speculative history updated by the previous prediction so that stale histories are not used. The perceptron predictor and path-based neural predictor are simulated as single branch predictors allowed to predict at most one branch per cycle. By avoiding stale histories for the multiple branch predictors and restricting the neural predictors to single branch prediction, we conservatively estimate the performance improvement yielded by the neural techniques over the conventional predictors.

4.4 Tuning the Predictors

Using the `train` inputs of the benchmarks and trace-driven simulation, we find the history lengths that minimize the average misprediction rate for each hardware budget and branch predictor, exploring hardware budgets from 1 KB to 64 KB. We use these history lengths in the execution-driven simulations on the `ref` inputs. Table III shows the tuned history lengths for each hardware budgets. Note that *gshare.fast* is not shown, as its history length is fully constrained by the details of its implementation, and is equal to the base-2 logarithm of the number of elements in the pattern history table.

4.5 Estimating Branch Predictor Latency

We use CACTI 3.0 [Shivakumar and Jouppi 2001] to estimate the latency 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 as well as the latency

Table IV. Estimated Access Latencies

Hardware Budget	2Bc-*gskew* (Cycles)	Global/Local Perceptron	Path-Based Neural
1 KB	2	5	2
2 KB	2	5	2
4 KB	2	5	2
8 KB	2	6	2
16 KB	2	6	2
32 KB	2	6	2
64 KB	3	7	3

of the adders used for the path-based neural predictor. Table IV shows the latencies we derived for each branch predictor and hardware budget except for *gshare.fast*, giving the amount of time it takes from the time a branch address is known to the time a prediction becomes available. The CACTI estimates take into account the fact that $h + 1$ independently addressable memories are used to implement the update phase of the predictor. For *gshare.fast*, the latency is always at most one cycle. For 2Bc-*gskew*, we estimate the latency of the predictor as the delay in accessing the slowest table plus one fan-out-of-four (FO4) delay for taking the majority and choosing the hybrid prediction from the two component predictions. For the global/local perceptron predictor, the latency is the sum of the access delay to the table of weights vectors measured by CACTI and the worst-case delay of the perceptron output circuit as measured by HSPICE. We optimistically ignore the access time to the first-level table of per-branch histories. The fixed-length path branch predictor is computationally expensive to implement because it requires hashing many addresses to produce one prediction. Nevertheless, we optimistically assume that it can be pipelined to produce a result with the same latency as 2Bc-*gskew*. For the path-based neural predictor, the latency is the sum of the access delay to the table of bias weights and the worse-case delay of the adder that adds the bias weight to the next partial sum in the *SR* vector. For consistency, we use the same adder circuits that were used in the original perceptron predictor study [Jiménez and Lin 2002]. All of the estimates assume a 90 nm technology and an aggressive 8 FO4 delays, that is, 3.86 GHz.

5. EXPERIMENTAL RESULTS

In this section, we give the results of our experimental studies. We discuss the misprediction rates of the various branch predictors. We then discuss the performance achieved by the predictors in terms of instructions-per-cycle (IPC).

5.1 Misprediction Rates

Figure 6 shows the arithmetic mean misprediction rates for the four predictors ranging over hardware budgets from 1 KB to 64 KB over all benchmarks as measured by the microarchitectural simulator. Clearly, the path-based neural predictor has the lowest misprediction rate of all the predictors for all hardware budgets. Figure 7 shows the misprediction rates for each benchmark at a 8 KB

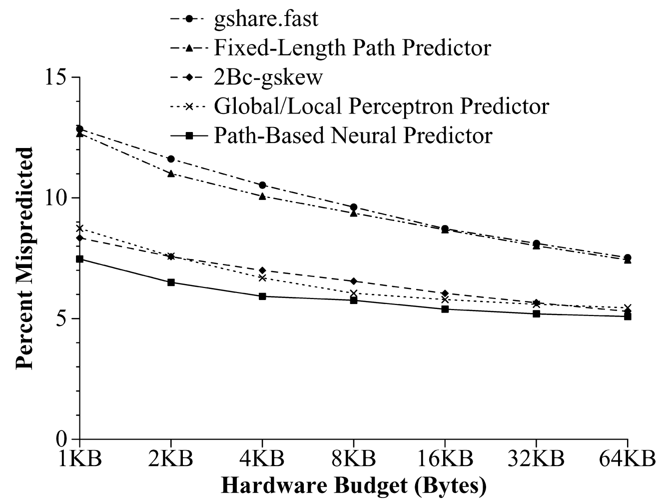

Fig. 6. Average misprediction rates per hardware budget.

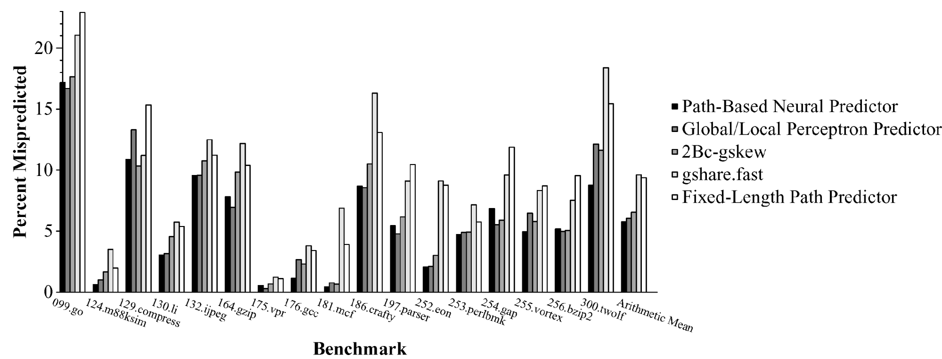

Fig. 7. Misprediction rates per benchmark at an 8 KB hardware budget.

hardware budget. The path-based neural predictor achieves an average misprediction rate of 5.7%, which is 7% lower than that of the global/local perceptron predictor at 6.1%, 13% lower than that of 2Bc-*gskew* at 6.6%, and 40% lower than that of the fixed-length path branch predictor at 9.4%. The path-based neural predictor has the lowest misprediction rate of all the predictors in 9 out of the 17 benchmarks. Ignoring the global/local perceptron predictor, the path-based neural predictor is the best predictor for 14 of the benchmarks.

Note that the average misprediction rates are slightly higher than those reported in previous work for two reasons. First, the set of benchmarks includes elements from both SPEC CPU 95 and SPEC CPU 2000, where previous work has focused on a smaller set of benchmarks. Second, the misprediction rates are those reported by the cycle-accurate deep-pipeline simulator simulating the realistic lag in time between prediction and update that is not experienced by a simulator such as `sim-bpred` that models instantaneous predictor update. We

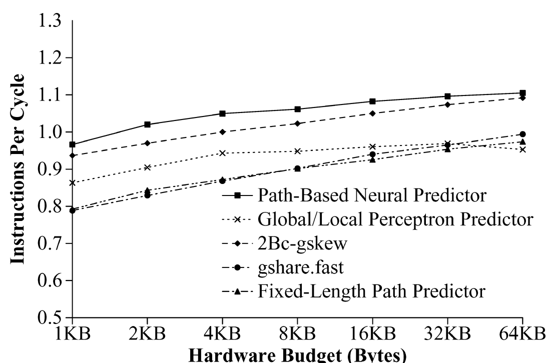

Fig. 8. Average IPC per hardware budget.

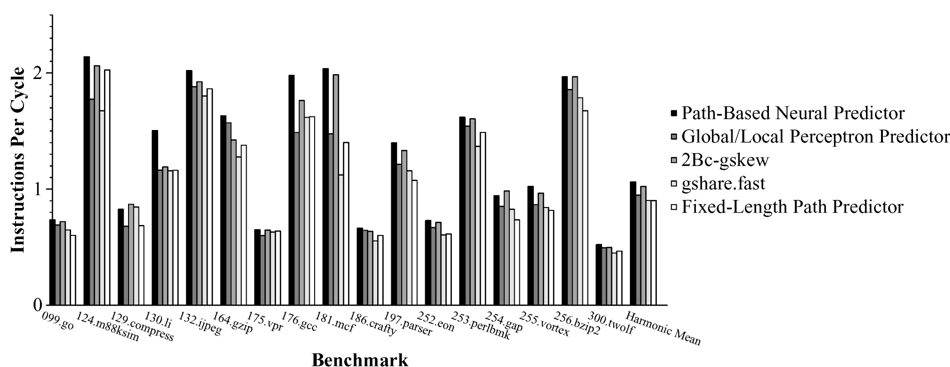

Fig. 9. IPC per benchmark at an 8 KB hardware budget.

believe that both of these points yield a more realistic estimate of what the branch predictor accuracy would be like for a real machine.

5.2 Instructions Per Cycle

Figure 8 shows the number of instructions executed per cycle (IPC) for each branch predictor and hardware budget. Clearly, the path-based neural predictor yields the best performance at every hardware budget. The key reason is the combination of superior accuracy and low latency. For instance, the global/local perceptron predictor, which is the second most accurate of all the branch predictors, yields the worse performance at higher hardware budgets because of its high latency. At the same time, 2Bc-*gskew*, a McFarling-style hybrid with approximately the same latency as the path-based neural predictor, delivers less accuracy and performance than the single-component path-based neural predictor. At a 64 KB hardware budget, the path-based neural predictor delivers an IPC 16% higher than that of the perceptron predictor because of that predictor's high latency.

Figure 9 shows the IPC for each benchmark and each predictor at an 8-KB hardware budget. The path-based neural predictor yields the best IPC in 15 out of the 17 benchmarks. It achieves a harmonic mean IPC of 1.06, giving a speedup

of 12% over the global/local perceptron predictor at 0.95 IPC, 4% over 2Bc-*gskew* at 1.02 IPC, 18% over *gshare.fast* at 0.90 IPC, and 18% over the fixed length path branch predictor at 0.90 IPC. At this hardware budget, both 2Bc-*gskew* and the path-based neural predictor have a latency of two cycles, while *gshare.fast* has a single-cycle latency. The global/local perceptron predictor has a latency of six cycles at this hardware budget. Although it is more accurate than *gshare.fast* and 2Bc-*gskew*, its higher latency cancels any advantage it might have for performance.

5.2.1 *Area vs. Hardware Budget.* Although standard for branch prediction research, equating the term *hardware budget* with number of bits of predictor state is problematic in our case. As described in Section 3.2.3, an implementation of the path-based neural predictor may use $h + 1$ independently addressable memories, each with its own selection logic, to facilitate the update algorithm. The path-based neural predictor also requires a number of adder circuits proportional to the history length. We estimate that a naive implementation of a path-based neural predictor using 8 KB of state could require 80% more area than a 8 KB 2Bc-*gskew* predictor. Even so, the path-based neural predictor is still the best choice. A path-based neural predictor with a hardware budget of 4 KB, consuming approximately 10% less total area than a 8 KB 2Bc-*gskew*, achieves a harmonic mean IPC of 1.05 which is less than 1% lower than that of an 8 KB path-based neural predictor and 3% higher than that of a 8 KB 2Bc-*gskew*. Indeed, a path-based neural predictor with only 2 KB of state achieves the same IPC as an 8 KB 2Bc-*gskew*.

6. ANALYSIS

In this section, we discuss the ability of the path-based neural predictor to adapt to a wider range of branch behavior than the original perceptron predictor. In particular, the path-based neural predictor can learn to predict branches characterized by linearly inseparable functions (hereafter, "linearly inseparable branches"), while the perceptron predictor will mispredict some of these branches.

6.1 Linear Separability

A branch direction predictor can be thought of as a device that attempts to learn a function $f : X \mapsto \{taken, not_taken\}$, where X is the domain of the function. The domain X is usually the set of n-length binary vectors representing pattern histories of length n. The assumption for branch prediction is that there exists such a function f for every branch that best characterizes the behavior of that branch.

Perceptrons are limited by the fact that they can only learn *linearly separable* functions. Let $x_{1..h} \in X$, that is, $x_{1..h}$ represents an instance of a branch history such that $f(x_{1..h})$ is either *taken* or *not_taken*. A branch prediction function f is linearly separable if and only if there exist integers $w_0, w_1, \dots, w_n$ such that the hyperplane with the equation $w_0 + \sum_{i=1}^{n} x_i w_i = 0$ separates X into one half-space containing only *taken* instances and another half-space containing

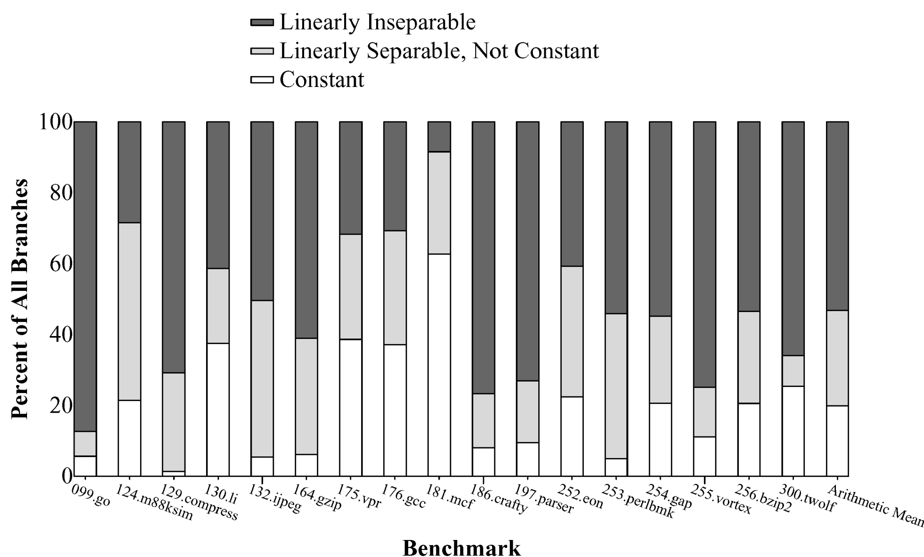

Fig. 10. Break-down of branches best predicted by constant, linearly separable, and linearly inseparable functions.

only *not_taken* instances. The weights in the equation of the hyperplane correspond to the weights in a trained perceptron.

In other words, perceptrons can only completely learn functions that can be characterized by a linear equation [Fausett 1994]. Perceptrons can only partially learn other functions. For instance, perceptrons can learn the Boolean functions AND and OR with 100% accuracy, but can only learn XOR with at most 75% accuracy. Constant functions, for example, always predict *taken*, are trivially linearly separable. On the other hand, two-level adaptive branch predictors such as *gshare* and 2Bc-*gskew* can theoretically learn arbitrary functions.

6.1.1 *Impact of Linear Separability.* What does linear separability have to do with branch prediction in real programs? Through our experiments, we find two surprising facts. First, over half of all branches executed are linearly inseparable. Second, *nearly all mispredictions* come from linearly inseparable branches for all the branch predictors we studied. Thus, linearly inseparable branches are the most important branches for branch predictors, but they are the hardest for perceptrons to learn. Note that traditional branch predictors based on two-level adaptive branch prediction are not limited by linear separability.

Figure 10 illustrates the phenomenon of linear separability in branch behavior. For each branch in each benchmark, we find a Boolean function with a history length of 10 that minimizes the misprediction rate for that branch. Our algorithm for finding the best function is exponential in the history length, and there are thousands of branches in the 17 benchmarks, so we limit the history length to 10 to keep simulation time reasonable. We then test each function for linear separability. For each benchmark, the differently shaded bars break

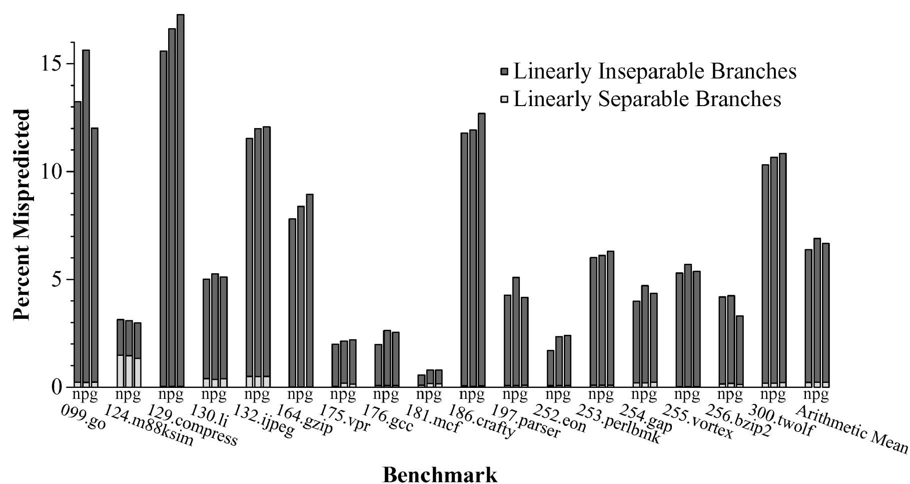

Fig. 11. Break-down of mispredictions for the path-based (n), perceptron (p), and *gshare* (g) predictors into linearly separable and linearly inseparable branches.

down the fraction of constant, linearly separable but not constant, and linearly inseparable branches weighted by their execution frequencies. Although there is a wide variance, on average 55% of all branches are linearly inseparable. Of the linearly separable branches, 20% are best predicted by a constant *taken* or *not_taken*, that is, they could be predicted just as well with a simple static or bimodal predictor [Smith 1981] as with a perceptron predictor. Thus, although the perceptron predictor is highly accurate, it cannot completely learn the functions that best predict over half of all branches.

Figure 11 breaks down mispredictions in terms of their linear separability for three branch predictors: the path-based neural predictor (n), the perceptron predictor using only global history (p), and *gshare* (g). Again, each predictor is simulated with a history length of 10 so that we can test for linear separability in a reasonable amount of time. Each branch predictor is simulated with a virtually unlimited hardware budget to suppress aliasing effects. Thus, the *gshare* we simulate should have the same accuracy as any other two-level predictor with a more sophisticated mechanism for dealing with aliasing.

For all benchmarks except for `124.m88ksim`, over 90% of all mispredictions come from linearly inseparable branches. Since the history lengths for each predictor are equal, we do not see a wide separation in the average misprediction rates. Nevertheless, note that the path-based neural predictor is the most accurate of the three, and that *gshare* is more accurate than the original perceptron predictor because the history lengths are equal.

If two-level predictors such as *gshare* can learn linearly inseparable functions, then why should the path-based neural predictor still outperform *gshare*? There are two answers to this question. First, it has been observed that neural predictors can learn more quickly than two-level predictors because of warm-up effects [Jiménez and Lin 2002]. Second, the path-based neural predictor combines path and pattern history, while *gshare* uses only pattern history.

7. CONCLUSION

We have presented a new neural branch predictor that has lower latency and superior accuracy to previous neural branch predictors. Our new predictor achieves high accuracy and low latency by predicting a branch using a neuron selected dynamically along the path to that branch. This work is only the beginning of path-based neural prediction; we have yet to fully exploit the potential of this technique. We have shown that our predictor has better accuracy and yields higher performance than conventional predictors. By incorporating our path-based neural predictor into new microarchitectures, designers will be able to improve IPC rates while increasing pipeline depths and clock frequencies.

ACKNOWLEDGMENTS

I thank Calvin Lin and Doug Burger for their helpful comments on the first draft of this paper. Thanks also to Charles Ganansia for working on circuit models for this research.

REFERENCES

BALL, T. AND LARUS, J. 1993. Branch prediction for free. In *Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation*. ACM, New York, 300–313.

BLOCK, H. D. 1962. The perceptron: A model for brain functioning. *Rev. Mod. Phys. 34*, 123–135.

BREKELBAUM, E., RUPLEY, J., WILKERSON, C., AND BLACK, B. 2002. Hierarchical scheduling windows. In *Proceedings of the 35th International Symposium on Microarchitecture* (Istanbul, Turkey).

BURGER, D. AND AUSTIN, T. M. 1997. The SimpleScalar tool set version 2.0. Tech. Rep. 1342, Computer Sciences Department, University of Wisconsin. June.

CALDER, B., GRUNWALD, D., LINDSAY, D., MARTIN, J., MOZER, M., AND ZORN, B. 1995. Corpus-based static branch prediction. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*. ACM, New York, 79–92.

CORMEN, T. H., LEISERSON, C. E., AND RIVEST, R. L. 1990. *Introduction to Algorithms*. McGraw Hill, New York.

EVERS, M., PATEL, S. J., CHAPPELL, R. S., AND PATT, Y. N. 1998. An analysis of correlation and predictability: What makes two-level branch predictors work. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*. 52–61.

FAUSETT, L. 1994. *Fundamentals of Neural Networks: Architectures, Algorithms and Applications*. Prentice-Hall, Englewood Cliffs, N.J.

JIMÉNEZ, D. A. 2002. Reconsidering complex branch predictors. In *Proceedings of the 9th International Symposium on High Performance Computer Architecture*. 43–52.

JIMÉNEZ, D. A. 2003. Fast path-based neural branch prediction. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, Press, Los Alamitos, Calif., 243–252.

JIMÉNEZ, D. A., KECKLER, S. W., AND LIN, C. 2000. The impact of delay on the design of branch predictors. In *Proceedings of the 33rd Annual International Symposium on Microarchitecture*. 67–76.

JIMÉNEZ, D. A. AND LIN, C. 2001. Dynamic branch prediction with perceptrons. In *Proceedings of the 7th International Symposium on High Performance Computer Architecture*. 197–206.

JIMÉNEZ, D. A. AND LIN, C. 2002. Neural methods for dynamic branch prediction. *ACM Trans. Comput. Syst. 20*, 4 (Nov.), 369–397.

KESSLER, R. E. 1999. The Alpha 21264 microprocessor. *IEEE Micro 19*, 2 (Mar./Apr.), 24–36.

LOH, G. H. AND HENRY, D. S. 2002. Predicting conditional branches with fusion-based hybrid predictors. In *Proceedings of the 11th Conference on Parallel Architectures and Compilation Techniques* (Charlottesville, Va.), 165–176.

McFarling, S. 1993. Combining branch predictors. Tech. Rep. TN-36m, Digital Western Research Laboratory. June.

Nair, R. 1995. Dynamic path-based branch correlation. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*. 15–23.

Rosenblatt, F. 1962. *Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms*. Spartan.

Seznec, A., Felix, S., Krishnan, V., and Sazeides, Y. 2002. Design tradeoffs for the Alpha EV8 conditional branch predictor. In *Proceedings of the 29th International Symposium on Computer Architecture*.

Seznec, A. and Fraboulet, A. 2003. Effective ahead pipelining of instruction block address generation. In *Proceedings of the 30th International Symposium on Computer Architecture* (San Diego, Calif.).

Shivakumar, P. and Jouppi, N. P. 2001. Cacti 3.0: An integrated cache timing, power and area model. Tech. Rep. 2001/2, Compaq Computer Corporation. August.

Skadron, K., Martonosi, M., and Clark, D. W. 2000. A taxonomy of branch mispredictions, and alloyed prediction as a robust solution to wrong-history mispredictions. In *Proceedings of the 2000 International Conference on Parallel Architectures and Compilation Techniques*. 199–206.

Smith, J. E. 1981. A study of branch prediction strategies. In *Proceedings of the 8th Annual International Symposium on Computer Architecture*. 135–148.

Sprangle, E. and Carmean, D. 2002. Increasing processor performance by implementing deeper pipelines. In *Proceedings of the 29th International Symposium on Computer Architecture* (Anchorage, Alaska). 25–34.

Stark, J., Evers, M., and Patt, Y. N. 1998. Variable length path branch prediction. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*. 170–179.

Thomas, R., Franklin, M., Wilkerson, C., and Stark, J. 2003. Improving branch prediction by dynamic dataflow-based identification of correlated branches from a large global history. In *Proceedings of the 30th International Symposium on Computer Architecture* (San Diego, Calif.).

Vintan, L. N. and Iridon, M. 1999. Towards a high performance neural branch predictor. In *Proceedings of the International Joint Conference on Neural Networks*. *2*, 868–873.

Received December 2003; revised November 2004; accepted November 2004