Neural Methods for Dynamic Branch Prediction

DANIEL A. JIMÉNEZ
Rutgers University
and
CALVIN LIN
The University of Texas at Austin

This article presents a new and highly accurate method for branch prediction. The key idea is to use one of the simplest possible neural methods, the perceptron, as an alternative to the commonly used two-bit counters. The source of our predictor's accuracy is its ability to use long history lengths, because the hardware resources for our method scale linearly, rather than exponentially, with the history length. We describe two versions of perceptron predictors, and we evaluate these predictors with respect to five well-known predictors. We show that for a 4 KB hardware budget, a simple version of our method that uses a global history achieves a misprediction rate of 4.6% on the SPEC 2000 integer benchmarks, an improvement of 26% over *gshare*. We also introduce a global/local version of our predictor that is 14% more accurate than the McFarling-style hybrid predictor of the Alpha 21264. We show that for hardware budgets of up to 256 KB, this global/local perceptron predictor is more accurate than Evers' multicomponent predictor, so we conclude that ours is the most accurate dynamic predictor currently available. To explore the feasibility of our ideas, we provide a circuit-level design of the perceptron predictor and describe techniques that allow our complex predictor to operate quickly. Finally, we show how the relatively complex perceptron predictor can be used in modern CPUs by having it override a simpler, quicker Smith predictor, providing IPC improvements of 15.8% over *gshare* and 5.7% over the McFarling hybrid predictor.

Categories and Subject Descriptors: C.1.1 [**Computer Systems Organization**]: Processor Architectures—*Single data stream architectures*

General Terms: Performance

Additional Key Words and Phrases: Branch prediction, neural networks

1. INTRODUCTION

Modern computer architectures increasingly rely on speculation to boost instruction-level parallelism. For example, data that are likely to be read in

D. A. Jiménez was supported by a fellowship from Intel Corporation. This research was supported in part by NSF CAREERS grant ACI-9984660, by ONR grant N00014-99-1-0402, and by the Defense Advanced Research Projects Agency under contract F33615-01-C-1892.

Authors' addresses: D. Jiménez, Rutgers University, Department of Computer Science, 110 Frelinghuysen Rd., Piscataway, NJ, 08854; email: djimenez@cs.rutgers.edu; C. Lin, Department of Computer Sciences, Taylor 2.124, University of Texas, Austin TX, 78712; email: lin@cs.utexas.edu.

the near future are speculatively prefetched, and predicted values are speculatively used before actual values are available [Hennessy and Patterson 1996; Wang and Franklin 1997]. Accurate prediction mechanisms have been the driving force behind these techniques, so increasing the accuracy of predictors increases the performance benefit of speculation. Machine learning techniques offer the possibility of further improving performance by increasing prediction accuracy. In this article, we show that one particular machine learning technique can be implemented in hardware to improve branch prediction.

Branch prediction is an essential part of modern microarchitectures. Rather than stall when a branch is encountered, a pipelined processor uses branch prediction to speculatively fetch and execute instructions along the predicted path. As pipelines deepen and the number of instructions issued per cycle increases, the penalty for a misprediction increases, as does the benefit of accurate branch prediction. Recent efforts to improve branch prediction focus primarily on eliminating aliasing in two-level adaptive predictors [McFarling 1993; Lee et al. 1997; Sprangle et al. 1997; Eden and Mudge 1998], which occurs when two unrelated branches destructively interfere by using the same prediction resources. We take a different approach—one that is largely orthogonal to previous work—by improving the accuracy of the prediction mechanism itself.

Our work builds on the observation that all existing two-level techniques use tables of saturating counters. Since neural networks are known to provide good predictive capabilities, it is natural to ask whether we can improve accuracy by replacing saturating counters with neural networks. However, most neural networks would be prohibitively expensive to implement as branch predictors, so we explore the use of simple artificial neurons from which these neural networks are built. These artificial neurons, such as the perceptron [Rosenblatt 1962], have several attractive properties that differentiate them from neural networks. They are easier to understand, they are simpler to implement and tune, they train faster, and they are computationally much less expensive.

In this article, we explore various types of artificial neurons and propose a two-level scheme that uses perceptrons instead of two-bit counters. Because the size of perceptrons scales linearly with the size of their inputs, which in our case is the branch history, our predictor can exploit long history lengths. By contrast, traditional two-level adaptive schemes use pattern history tables (PHTs), which are indexed by the branch history and which therefore grow exponentially with the history length. Thus the PHT structure limits the length of the history register to the logarithm of the number of counters. As a result, for the same hardware budget, our predictor can consider much longer histories than PHT-based schemes. For example, for a 4 KB hardware budget, a PHT-based predictor can use a history length of 14, whereas a version of our predictor can use a history length of 34. These longer history lengths lead to higher accuracy.

A perceptron is a learning device that takes a set of input values and combines them with a set of weights (which are learned through training) to produce an output value. In our predictor, each weight represents the degree of

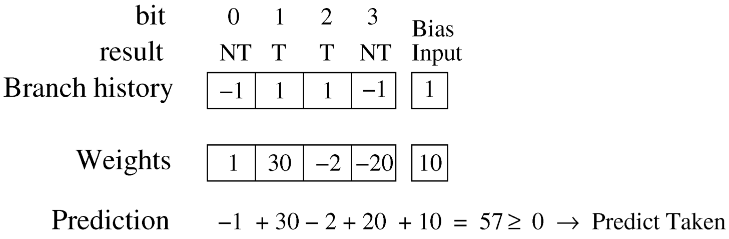

Fig. 1. The perceptron prediction mechanism. The prediction is the sign of the dot product of the branch history and the perceptron weights. The *taken* branches (T) in the branch history are represented as 1's, and *not taken* branches (NT) are represented as −1's. The bias weight represents the bias of the branch independent of branch history, so its input bit is hardwired to 1.

correlation between the behavior of a past branch and the behavior of the branch being predicted. Positive weights represent positive correlation, and negative weights represent negative correlation. To make a prediction, each weight contributes in proportion to its magnitude in the following manner. If its corresponding branch was taken, we add the weight; otherwise we subtract the weight. If the resulting sum is positive, we predict *taken*; otherwise we predict *not taken*. To make this solution work, the branch history uses 1 to represent *taken* and −1 to represent *not taken*. The perceptrons are trained by an algorithm that increments a weight when the branch outcome agrees with the weight's correlation and decrements the weight otherwise.

Figure 1 shows an example with a four-bit history length. We see that the second and fourth branches (corresponding to bits 1 and 3, respectively) contribute the most to the prediction of the next branch. In particular, the second weight indicates that there is a strong positive correlation between the direction of the second branch and the direction of the predicted branch. The fourth weight reveals a strong negative correlation between the outcome of the fourth branch and the outcome of the predicted branch; because the fourth branch was not taken, this strong negative correlation suggests that the predicted branch will be taken. Finally, the figure shows that an additional bias weight, which learns the bias of the branch independent of branch history, also contributes to the prediction.

This article describes and evaluates various perceptron predictors. We show that the perceptron works well for the class of *linearly separable branches*, a term we introduce. We also show that programs tend to have many linearly separable branches and that although perceptrons are unable to predict linearly inseparable branches, PHT-based schemes also have trouble predicting such branches.

This article makes the following contributions.

—We describe the perceptron predictor [Jiménez and Lin 2001], the first dynamic predictor to successfully use neural networks, and we show that it is more accurate than existing dynamic global branch predictors. For a 4 KB hardware budget, our global predictor improves misprediction rates on the SPEC 2000 integer benchmarks by 26% over a *gshare* predictor of the same size and by 12% over the McFarling-style hybrid predictor of the Alpha 21264.[1]

—We introduce a version of the perceptron predictor that uses both global and per-branch information, yielding misprediction rates that are 14% more accurate than the McFarling-style hybrid predictor, which is the most accurate predictor that is known to have been implemented in silicon. We also show that our predictor is more accurate than Evers' multicomponent predictor [Evers 2000], making it the most accurate known dynamic predictor.

—We provide a circuit-level design of the perceptron predictor. Using concepts from binary arithmetic, we show how to construct an efficient circuit for computing the perceptron output. With transistor-level simulations, we measure the latency of our perceptron output circuit.

—We suggest how the perceptron predictor, despite its complex design, can be implemented and used in modern CPUs. In particular, we introduce a hierarchical predictor in which a perceptron predictor overrides a faster Smith predictor. We show that this overriding perceptron predictor improves IPC by 15.8% over *gshare* and by 5.7% over the McFarling-style hybrid predictor.

—We show that the chief advantage of our predictor over PHT-based predictors is the ability to use long history lengths.

The remainder of this article is organized as follows. Section 2 summarizes related work, and Section 3 provides background on neural learning methods and their potential applications in microarchitectures. Section 4 reviews the characteristics of perceptrons, in preparation for Section 5, where we discuss details of the perceptron predictor. In Section 6 we present experimental results, and we conclude in Section 7.

2. RELATED WORK

This section reviews related work in dynamic branch prediction and neural systems.

2.1 Dynamic Branch Prediction

Dynamic branch prediction has been the focus of intense study in the literature. Recent research focuses on refining the two-level scheme of Yeh and Patt [1991]. In this scheme, a pattern history table (PHT) of two-bit saturating counters is indexed by a combination of branch address and global or per-branch history. The high bit of the counter is taken as the prediction. Once the branch outcome is known, the counter is incremented if the branch is taken, and decremented otherwise. An important problem in two-level predictors is aliasing [Sechrest et al. 1996], and many of the recently proposed branch predictors seek to reduce the aliasing problem [McFarling 1993; Lee et al. 1997; Sprangle et al. 1997;

[1] These results differ from our previously published numbers [Jiménez and Lin 2001] because our new methodology uses the Alpha instruction set, which allows us to get simulated IPC results from SimpleScalar [Burger and Austin 1997]. We discuss the impact of this methodological change in Section 6.1.1.

Eden and Mudge 1998] but do not change the basic prediction mechanism. Given a generous hardware budget, many of these two-level schemes perform about the same as one another [Eden and Mudge 1998].

Most two-level predictors cannot consider long history lengths, which becomes a problem when the distance between correlated branches is longer than the length of a global history shift register [Evers et al. 1998]. Even if a PHT scheme could somehow implement longer history lengths, it would not help because longer history lengths require longer training times for these methods [Michaud et al. 1997].

Variable length path branch prediction [Stark et al. 1998] is one scheme for considering longer paths. It avoids the PHT capacity problem by computing a hash function of the addresses along the path to the branch. It uses a complex multipass profiling and compiler-feedback mechanism that is impractical for a real architecture, but it achieves good performance because of its ability to consider longer histories.

2.2 Neural Methods and Computer Architecture

Neural systems and other forms of machine learning have been suggested for several computer architecture applications.

2.2.1 *The Perceptron Predictor.* In a previous paper [Jiménez and Lin 2001] we introduce the basic perceptron predictor that uses only global history information, and we compare it with two dynamic global predictors, *gshare* and *bi-mode*.

2.2.2 *Branch Prediction with Neural Networks.* Neural networks have been used to perform static branch prediction [Calder et al. 1997], where the likely direction of a branch is predicted at compile-time by supplying program features, such as control-flow and opcode information, as input to a trained neural network. This approach achieves a 20% misprediction rate compared to a 25% misprediction rate for static heuristics [Ball and Larus 1993; Calder et al. 1997]. Static branch prediction performs worse than existing dynamic techniques, but can be useful for performing static compiler optimizations and providing extra information to dynamic branch predictors such as the *agree* predictor [Sprangle et al. 1997].

Learning vector quantization (LVQ), another neural method, has been suggested for dynamic branch prediction by Vintan and Iridon [1999]. LVQ prediction is about as accurate as a table-based branch predictor. Unfortunately, LVQ does not lend itself well to high-speed implementation because it performs complex computations involving floating point numbers. By contrast, our predictor has accuracy superior to any table-based method and can be implemented efficiently.

2.2.3 *Branch Prediction and Genetic Algorithms.* Neural networks are part of the field of machine learning, which also includes genetic algorithms. Emer and Gloy [1997] use genetic algorithms to “evolve” branch predictors, [1997] but it is important to note the difference between their work and ours. Their work uses evolution to design more accurate predictors, but the end result is something similar to a highly tuned traditional predictor. We instead place intelligence in the microarchitecture, so the branch predictor can learn and adapt online. In fact, Emer and Gloy's approach cannot describe our new predictor.

2.2.4 *Neural Networks for Resource Allocation.* Neural networks learned through evolutionary computation have been proposed as a method for managing on chip resources for chip multiprocessors [Gomez et al. 2001]. When compared with static partitioning, performance is improved 13% when a neural network is used to dynamically assign a pool of L2 cache banks to a set of cores.

3. NEURAL SYSTEMS

In this section we describe the basics of how artificial neural systems work, we explain how neural methods might be applied to dynamic branch prediction, and we explain why we choose the perceptron in particular for branch prediction.

Neural systems employ some of the properties of biological neural networks, such as brains and nervous systems, for computational tasks such as prediction and regression. Artificial neural networks learn to compute a function using example inputs and outputs. Neural networks have been used for a variety of applications, including pattern recognition, classification [Faucett 1994], and image understanding [Kulkarni 1993; Jiménez and Walsh 1998]. The general idea of neural computation is that many processing nodes, known as *neurons*, are connected to each other in a network. Data are fed into input unit neurons, and propagated through the network to output unit neurons, where the results of the computation can be read. A training algorithm strengthens or weakens the connections between neurons.

Neural systems learn a general solution to a problem from specific examples. Generally, the more examples there are, the better the solution will be. Neural systems seem particularly well suited for microarchitectural prediction problems, since processors execute hundreds of millions of instructions each second, providing ample learning examples.

3.1 Prediction with Neural Methods

Prediction with neural methods is a rich area of study. Neural methods are capable of *classification*, that is, predicting into which set of classes a particular instance will fall. Suppose a set S is partitioned into n classes, and we are faced with the problem of determining, for an arbitrary element $s \in S$, in what class s is. The elements of S have certain features that correlate with their classifications. An artificial neural network can learn correlations between these features and the classification. An artificial neural network is a collection of neurons, some of which receive input and some of which produce output, that are connected by links. Each link has an associated weight that determines the strength of the connection [Faucett 1994]. For a classification problem, such as deciding to which of n classes an input s belongs, there are n output neurons. In the special case where there are only two classes, there is only one output neuron. Each neuron computes its output from the sum of its input using an activation function. During a training phase, the weights are adjusted using a training algorithm. The algorithm uses a set of training data, which are ordered pairs of inputs and corresponding outputs. The neural network learns correlations between the inputs and outputs, and generalizes this learning to other inputs. To predict which class a new input s is in, we supply s to the input units of the trained neural network, propagate the values through the network, and examine the n output neurons. We classify s according to the neuron with the strongest output. In the special case where $n = 2$ and there is only one output neuron, we classify s according to whether the output value exceeds a certain threshold, typically 0 or 1/2.

3.2 Potential Applications to Microarchitecture

Neural learning methods have the potential to enhance microarchitectural techniques, replacing the more primitive predictors currently used. Some possible applications are the following.

—*Branch Prediction*. For dynamic branch prediction, the inputs to a neural learning method are the binary outcomes of recently executed branches, and the output is a prediction of whether a branch will be taken. Each time a branch is executed and the true outcome becomes known, the history that led to this outcome can be used to train the neural method online to produce a more accurate result in the future.

—*Value Prediction*. Neural networks could be used to predict which of a set of values is likely to be the result of a load operation, enabling speculation on that value.

—*Indirect Branch Prediction*. Indirect branches, such as virtual method calls in object-oriented programs, also need to be predicted. Neural networks could be used to help predict the targets of such indirect branches.

—*Next Trace Prediction*. As a natural extension of the branch prediction capabilities of neural learning techniques, neural networks could be used to predict which of several possible traces should be fetched from a trace cache.

—*Cache Replacement Policy*. Neural networks could be used to implement specialized cache replacement policies that reduce cache miss rates by adapting to the program's access patterns.

It is not trivial to extend our work on branch prediction to these other microarchitectural problems. Neural learning works best when classifying an input as coming from one of a few classes. Predicting branch directions requires a single perceptron to classify a pattern as either *taken* or *not taken*. Predicting a value or a branch target would require more than one perceptron for each prediction, as well as an auxiliary table of choices, for example, previously observed targets or values.

3.3 Neural Learning for Dynamic Branch Prediction

There are several simple neural learning methods that could potentially be used in a dynamic branch predictor. In particular, the ADALINE neuron [Widrow and

Hoff, Jr. 1960], Hebb learning [Faucett 1994], and the Block perceptron [Block 1962] are simple methods in which a single neuron is used for computation and trained with a simple algorithm. We used the SPEC95 benchmarks to compare the accuracy of each of these methods. We also evaluated the accuracy of a more complex multilayer perceptron with back-propagation [Faucett 1994]. This back-propagation method is representative of commonly used neural networks, and it was included solely to explore the limits of neural learning techniques in dynamic branch prediction. Because of its implementation complexity, there is no way to implement back-propagation in hardware such that a prediction can be produced in just a few cycles.

Our results showed that the perceptron was the most accurate of the four techniques. We found that Hebb learning yields poor branch prediction accuracy due to its inability to learn even simple patterns. We found that ADALINE yields similar prediction accuracy to the perceptron, but ADALINE requires much more space. ADALINE neurons are sensitive to a parameter known as the learning rate; because we found good learning rates to be small (e.g., 0.03), and because the learning rate is multiplied by the ADALINE output to produce a result, ADALINE neurons require twice as much space as perceptrons to represent the weights with sufficient accuracy. Interestingly, we found that the perceptron learns faster and yields more accurate prediction than back-propagation. For instance, on the SPEC95 benchmark `126.gcc`, perceptrons achieve a 2.44% misprediction rate, compared with 3.33% for back-propagation.

Another benefit of perceptrons is that by examining their *weights*, that is, the correlations that they learn, it is easy to understand the decisions that they make. By contrast, a criticism of many neural networks is that it is difficult or impossible to determine exactly how the neural network is making its decision. Techniques have been proposed to extract rules from neural networks [Setiono and Liu 1995], but these rules are not always accurate. Perceptrons do not suffer from this opaqueness; the perceptron's decision-making process is easy to understand as the result of a simple mathematical formula.

4. BRANCH PREDICTION WITH PERCEPTRONS

This section provides the background needed to understand our predictor. We describe perceptrons, explain how they can be used in branch prediction, and discuss their strengths and weaknesses. We then describe our basic prediction mechanism, which is essentially a two-level predictor that replaces the pattern history table with a table of perceptrons.

4.1 How Perceptrons Work

The perceptron was introduced in 1962 [Rosenblatt 1962] as a way to study brain function. We consider the simplest of many types of perceptrons [Block 1962], a single-layer perceptron consisting of one artificial neuron connecting several input units by weighted edges to one output unit. A perceptron learns a target Boolean function $t(x_1, \ldots, x_n)$ of n inputs. In our case, the x_i are the bits of a global branch history shift register, and the target function predicts whether a particular branch will be taken. Intuitively, a perceptron keeps track of positive

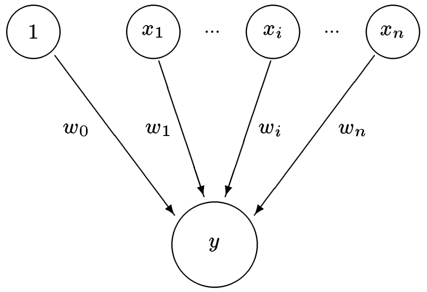

Fig. 2. Perceptron Model. The input values $x_1, \ldots, x_n$, are propagated through the weighted connections by taking their respective products with the weights $w_1, \ldots, w_n$. These products are summed, along with the bias weight w_0, to produce the output value y.

and negative correlations between branch outcomes in the global history and the branch being predicted.

Figure 2 shows a graphical model of a perceptron. A perceptron is represented by a vector whose elements are the weights. For our purposes, the weights are signed integers. The output is the dot product of the weights vector, $w_{0..n}$, and the input vector, $x_{1..n}$ (x_0 is always set to 1, providing a "bias" input). The output y of a perceptron is computed as

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

The inputs to our perceptrons are *bipolar*; that is, each x_i is either -1, meaning *not taken* or 1, meaning *taken.* A negative output is interpreted as *predict not taken.* A nonnegative output is interpreted as *predict taken.*

4.2 Training Perceptrons

Once the perceptron output y has been computed, the following algorithm is used to train the perceptron. Let t be -1 if the branch was not taken, or 1 if it was taken, and let be the *threshold*, a parameter to the training algorithm used to decide when enough training has been done.

```
if sign($y_{out}$) $\neq t$ or $|y_{out}| \leq$  then
    for $i := 0$ to $n$ do
        $w_i := w_i + t x_i$
    end for
end if
```

Since t and x_i are always either -1 or 1, this algorithm increments the ith weight when the branch outcome agrees with x_i, and decrements the weight when it disagrees. Intuitively, when there is mostly agreement (i.e., positive correlation), the weight becomes large. When there is mostly disagreement (i.e., negative correlation), the weight becomes negative with large magnitude. In both cases, the weight has a large influence on the prediction. When there is weak correlation, the weight remains close to 0 and contributes little to the output of the perceptron.

4.3 Linear Separability

A limitation of perceptrons is that they are only capable of learning linearly separable functions [Faucett 1994]. To understand the notion of linearly separable functions, imagine the set of all possible inputs to a perceptron as an n-dimensional space. The solution to the equation

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

is a hyperplane (e.g., a line, if $n = 2$) dividing the space into the set of inputs for which the perceptron will respond *false* and the set for which the perceptron will respond *true* [Faucett 1994]. A Boolean function over variables $x_{1..n}$ is linearly separable if and only if there exist values for $w_{0..n}$ such that all of the *true* instances can be separated from all of the *false* instances by that hyperplane. Since the output of a perceptron is decided by the above equation, only linearly separable functions can be learned perfectly by perceptrons. For instance, a perceptron can learn the logical AND of two inputs, but not the exclusive-OR, since there is no line on the Boolean plane separating *true* instances of the exclusive-OR function from *false* ones.

As we show later, many of the functions describing the behavior of branches in programs are linearly separable. Also, since we allow the perceptron to learn over time, it can adapt to the nonlinearity introduced by phase transitions in program behavior. A perceptron can still give good predictions when learning a linearly inseparable function, but it will not achieve 100% accuracy. By contrast, two-level PHT schemes like *gshare* can learn any Boolean function if given enough training time.

4.4 Using Perceptrons in Branch Predictors

We can use a perceptron to learn correlations between particular branch outcomes in the global history and the behavior of the current branch. These correlations are represented by the weights. The larger the weight, the stronger the correlation, and the more that particular branch in the global history contributes to the prediction of the current branch. The input to the bias weight is always 1, so instead of learning a correlation with a previous branch outcome, the bias weight w_0 learns the bias of the branch, independent of the history.

The processor keeps a table of N perceptrons in fast SRAM, similar to the table of two-bit counters in other branch prediction schemes. The number of perceptrons N is dictated by the hardware budget and number of weights, which itself is determined by the amount of branch history we keep. Special circuitry computes the value of y and performs the training. We discuss this circuitry in Section 5. When the processor encounters a branch in the fetch stage, the following steps are conceptually taken.

1. The branch address is hashed to produce an index $i \in 0..N-1$ into the table of perceptrons.
2. The ith perceptron is fetched from the table into a vector register $P_{0..n}$ of weights.

3. The value of y is computed as the dot product of P and the global history register.
4. The branch is predicted *not taken* when y is negative, or *taken* otherwise.
5. Once the actual outcome of the branch becomes known, the training algorithm uses this outcome and the value of y to update the weights in P.
6. P is written back to the ith entry in the table.

It may appear that prediction is slow because many computations and SRAM transactions take place in Steps 1 through 5. However, Section 5 shows that a number of arithmetic and microarchitectural tricks enable a prediction in a single cycle, even for long history lengths.

5. DESIGN AND IMPLEMENTATION

This section explores the design space for perceptron predictors and discusses details of a circuit-level implementation. We then describe two versions of the perceptron predictor, one that improves accuracy by utilizing both global and local information, and one that compensates for delay in computing the perceptron output.

5.1 Design Space

Given a fixed hardware budget, three parameters need to be tuned to achieve the best performance: the history length, the number of bits used to represent the weights, and the threshold.

5.1.1 *History Length.* Long history lengths can yield more accurate predictions [Evers et al. 1998] but for a fixed hardware budget they also reduce the number of table entries, thereby increasing aliasing. In our experiments, the best history lengths ranged from 4 to 66, depending on the hardware budget. The perceptron predictor can use more than one kind of history. We have used both purely global history as well as a combination of global and per-branch history.

5.1.2 *Representation of Weights.* The weights for the perceptron predictor are signed integers. Although many neural networks have floating-point weights, we found that integers are sufficient for our perceptrons, and they simplify the design. We find that using 8-bit weights provides the best trade-off between accuracy and hardware budget.

5.1.3 *Threshold.* The threshold is a parameter to the perceptron training algorithm that is used to decide whether the predictor needs more training.

5.2 Circuit-Level Implementation

Here, we discuss general techniques that will allow us to implement a quick perceptron predictor. We then give more detailed results of a transistor-level simulation.

5.2.1 *Computing the Perceptron Output.* The critical path for making a branch prediction includes the computation of the perceptron output. Thus, the circuit that evaluates the perceptron should be as fast as possible. Several properties of the problem allow us to make a fast prediction. Since -1 and 1 are the only possible input values to the perceptron, multiplication is not needed to compute the dot product. Instead, we simply add when the input bit is 1 and subtract (add the two's-complement) when the input bit is -1. In practice, we have found that adding the one's-complement, which is a good estimate for the two's-complement, works just as well and lets us avoid the delay of a small carry-propagate adder. This computation is similar to that performed by multiplication circuits, which must find the sum of partial products that are each a function of an integer and a single bit. Furthermore, only the sign bit of the result is needed to make a prediction, so the other bits of the output can be computed more slowly without having to wait for a prediction. In this article, we report only results that simulate this complementation idea.

At the circuit level, the perceptron output circuit accepts input signals from the weights array and from the history register. As weights are read, they are bitwise exclusive-ORed with the corresponding bits of the history register. If the ith history bit is set, then this operation has the effect of taking the one's-complement of the ith weight; otherwise, the weight is passed unchanged. After the weights are processed, their sum is found using a Wallace-tree of 3-to-2 carry-save adders [Cormen et al. 1990], which reduces the problem of finding the sum of n numbers to the problem of finding the sum of two numbers. The final two numbers are summed with a carry-lookahead adder. The Wallace-tree has depth $O(\log n)$, and the carry-lookahead adder has depth $O(\log n)$, so the computation is relatively quick. The sign of the sum is inverted and taken as the prediction.

5.2.2 *Training.* The training algorithm of Section 4.2 can be implemented efficiently in hardware. Since there are no dependences between loop iterations, all iterations can execute in parallel. Since in our case both x_i and t can only be -1 or 1, the loop body can be restated as "increment w_i by 1 if $t = x_i$, and decrement otherwise," a quick arithmetic operation since the w_i are 8-bit numbers.

```
for each bit in parallel
    if $t = x_i$ then
        $w_i := w_i + 1$
    else
        $w_i := w_i - 1$
    end if
```

5.2.3 *Circuit-Level Simulation.* Using a custom logic design program and the HSPICE and CACTI 2.0 simulators, we designed and simulated a hardware implementation of the elements of the critical path for the perceptron predictor for several table sizes and history lengths. We used CACTI, a cache modeling tool, to estimate the amount of time taken to read the table of perceptrons, and we used HSPICE to measure the latency of our perceptron output circuit.

Table I shows the delay of the perceptron predictor for several hardware budgets and history lengths, simulated with HSPICE and CACTI for 180-nm process technology. We obtain these delay estimates by selecting inputs designed to elicit the worst-case gate delay. We measure the time it takes for one

Table I. Perceptron Predictor Delay

History Length	Table Size (Bytes)	Perceptron Delay (ps)	Table Delay (ps)	Total Delay (ps)	# Clock Cycles	
					@ 833 MHz	@ 1.76 GHz
4	128	811	386	1197	1.0	2.1
7	256	808	411	1219	1.1	2.2
9	512	725	432	1157	1.0	2.0
13	1 K	1090	468	1558	1.3	2.7
17	2 K	1170	504	1674	1.4	2.9
23	4 K	1700	571	2271	1.9	4.0
24	4 K	1860	571	2431	2.0	4.3

of the input signals to cross half of V_{DD} until the time the perceptron predictor yields a steady usable signal. For a 4 KB hardware budget and history length of 24, the total time taken for a perceptron prediction is 2.4 nanoseconds. This works out to slightly less than two clock cycles for a CPU with a clock rate of 833 MHz, the clock rate of the fastest 180-nm Alpha 21264 processor as of this writing. The Alpha 21264 branch predictor itself takes two clock cycles to deliver a prediction, so our predictor is within the bounds of existing technology. Note that a perceptron predictor with a history of 23 is about 10% faster than one with a history length of 24; a predictor with 24 weights (23 for history plus 1 for bias) can be organized more efficiently than a predictor with 25 weights, for reasons specific to our Wallace-tree design.

5.3 Global/Local Perceptron Predictor

As described, the perceptron predictor uses global history information to correlate branch history and branch outcomes, but our predictor can be improved by incorporating per-branch history as well. For some branches, the outcome is more strongly correlated with per-branch, or *local,* history, than with global history [Yeh and Patt 1991], so many predictors use both types of information in making predictions. In particular, *alloyed predictors* use a PHT index that is created by concatenating the global history and the local history (along with some branch address bits). At low hardware budgets, alloyed predictors are more effective than hybrid predictors that select from between two predictors [McFarling 1993], because hybrids not only require a third choice predictor, but they partition the hardware into separate global and local components.

Our *global/local* version of the perceptron predictor is similar to the alloyed predictor: a single table of perceptrons is used, but some perceptron input units receive their input from a global history register, and others receive their input from a local history register that is maintained for the particular branch being predicted. Thus, our global/local perceptron considers both global and local histories together and can achieve higher accuracy than one that only uses global histories.

5.4 Overriding Perceptron Predictor

A potential problem with any complex predictor is delay, as a branch predictor ideally operates in a single processor clock cycle. Jiménez et al. [2000] study a number of techniques for reducing the impact of delay on branch predictors. For example, a *cascading* predictor uses a simple predictor to anticipate the address of the next branch to be fetched, and it uses a more complex predictor to begin predicting the anticipated address. If the branch were to arrive before the complex predictor were finished, or if the anticipated branch address were found to be incorrect, a small *gshare* table would be consulted for a quick prediction. The study shows that a cascading predictor, using two *gshare* tables, is able to use the larger table 47% of the time. An alternate solution is to use an *overriding* predictor, in which a prediction is initiated by both a quick first-level predictor and a more complex second-level predictor at the same time. The first-level predictor gets an immediate prediction, which the second-level predictor can later override. If the quick prediction is overridden, the actions taken by the fetch engine are rolled back and restarted with the new prediction, incurring a small penalty.

The perceptron can be used with either the cascading or overriding schemes. The overriding strategy is particularly appropriate since, as pipelines continue to deepen, the cost of overriding a less accurate predictor decreases as a percentage of the cost of a full misprediction. Thus, in this article we evaluate an *overriding perceptron predictor*, in which a second-level perceptron predictor is combined with a first-level *gshare* predictor. When a branch is encountered, there are four possibilities.

—The first- and second-level predictions agree and are correct. In this case, there is no penalty.

—The first- and second-level predictions disagree, and the second one is correct. In this case, the second predictor overrides the first, with a small penalty.

—The first- and second-level predictions disagree, and the second one is incorrect. In this case, there is a penalty equal to the overriding penalty from the previous case as well as the penalty of a full misprediction. Fortunately, the second predictor is more accurate that the first, so this case is unlikely to occur.

—The first- and second-level predictions agree and are both incorrect. In this case, there is no overriding, but the prediction is wrong, so a full misprediction penalty is incurred.

The Alpha 21264 uses a similar branch predictor, with a slower hybrid branch predictor overriding a less accurate but faster line predictor [Kessler 1999]. When a line prediction is overridden, the Alpha predictor incurs a single-cycle penalty, which is small compared to the seven-cycle penalty for a branch misprediction. We present a detailed analysis of these overriding predictors in Section 6.3.

6. RESULTS AND ANALYSIS

This section compares the perceptron-based predictors against well-known techniques from the literature. We present three sets of results: the first evaluates accuracy at realistic hardware budgets, the second studies the limits of our approach by considering accuracy at extremely large hardware budgets, and the third evaluates overall processor performance using IPC as the metric. We also present analysis to explain why the perceptron predictor performs well.

Before proceeding, we now describe some methodological points that are common to all of our experiments. We gather traces using SimpleScalar/Alpha [Burger and Austin 1997]. Each time the simulator executes a conditional branch, it records the branch address and outcome in a trace file. The traces are then fed to a program that simulates the different branch prediction techniques. Branches in libraries are not profiled.

We use as benchmarks the 12 SPEC 2000 integer programs. We allow each benchmark to execute 300 million branches, which causes each benchmark to execute at least one billion instructions. To measure only the steady-state prediction accuracy, without effects from the benchmarks' initializations, we skip the first 50 million branches in the trace. To tune the predictors, we use the SPEC `train` inputs; to report misprediction rates, we test the predictors on the `ref` inputs.

6.1 Accuracy

We evaluate the accuracy of the perceptron predictor by comparing it first with four well-known predictors at hardware budgets of 256 bytes to 8 KB, which reflect the sizes of branch predictors found in commercial microprocessors.

6.1.1 *Methodology.* We now present our experimental methodology for the first set of experimental results, describing the predictors that we simulate and explaining how they were tuned.

6.1.1.1 *Predictors Simulated.* We simulate the *gshare* predictor [McFarling 1993], the *bi-mode* predictor [Lee et al. 1997], and a combination *gshare* and PAg McFarling-style hybrid predictor [McFarling 1993] similar to that of the Alpha 21264, with all tables scaled exponentially for increasing hardware budgets. For the perceptron predictor, we simulate a purely global predictor and a global/local perceptron predictor. For the global/local perceptron predictor, the extra state used by the table of local histories was constrained to be within 35% of the hardware budget for the rest of the predictor, reflecting the design of the Alpha 21264 hybrid predictor. For the *gshare* and the perceptron predictors, we also simulate the *agree* mechanism [Sprangle et al. 1997], which predicts whether a branch outcome will agree with a bias bit set in the branch instruction. The *agree* mechanism turns destructive aliasing into constructive aliasing, increasing accuracy at small hardware budgets.

Our methodology differs from our previous work on the perceptron predictor [Jiménez and Lin 2001], which used traces from x86 executables of SPEC2000 and only explored global versions of the perceptron predictor. We find that the perceptron predictor achieves a larger improvement over other predictors for the Alpha instruction set than for the x86 instruction set. We believe that this difference stems from the Alpha's RISC instruction set, which requires more dynamic branches to accomplish the same work, and which thus requires longer histories for accurate prediction. Because the perceptron predictor can make use of longer histories than other predictors, it performs better for RISC instruction sets.

Table II. Best History Lengths

Hardware	*gshare*		Global Perceptron		Global/Local Perceptron	
Budget	History		History		Global/Local	
(Bytes)	Length	# Entries	Length	# Entries	History	# Entries
128	2	512	4	25	8/2	11
256	1	1 K	7	32	10/2	19
512	11	2 K	9	51	23/2	19
1 K	12	4 K	13	73	25/5	33
2 K	13	8 K	17	113	31/5	55
4 K	14	16 K	24	163	34/10	91
8 K	15	32 K	28	282	34/10	182
16 K	16	64 K	47	348	36/11	341

6.1.1.2 *Tuning the Predictors.* We tune each predictor for history length using traces gathered from each of the 12 benchmarks and the `train` inputs. We exhaustively test every possible history length at each hardware budget for each predictor, keeping the history length that yields the lowest arithmetic mean misprediction rate. For the global/local perceptron predictor, we exhaustively test each pair of history lengths such that the sum of global and local history lengths is at most 50. For the *agree* mechanism, we set bias bits in the branch instructions using branch biases learned from the `train` inputs.

For the global perceptron predictor, we find, for each history length, the value of the threshold by using an intelligent search of the space of values, pruning areas of the space that give poor performance. We reuse the same thresholds for the global/local and *agree* perceptron predictors.

Table II shows the results of the history length tuning. We find an interesting relationship between history length and threshold: the best threshold for a given history length h is always exactly $= \lfloor 1.93h + 14 \rfloor$. This is because adding another weight to a perceptron increases its average output by some constant, so the threshold must be increased by a constant, yielding a linear relationship between history length and threshold. Through experimentation, we determine that using eight bits for the perceptron weights yields the best results.

6.1.2 *Impact of History Length on Accuracy.* One of the strengths of the perceptron predictor is its ability to consider much longer history lengths than traditional two-level schemes, which helps because highly correlated branches sometimes occur at a large distance from each other [Evers et al. 1998]. Any global branch prediction technique that uses a fixed amount of history information will have an optimal history length for a given set of benchmarks. As we can see from Table II, the perceptron predictor works best with much longer histories than the *gshare* predictor. For example, with a 4 KB hardware budget, *gshare* works best with a history length of 14, the maximum possible length for *gshare*. At the same hardware budget, the global perceptron predictor works best with a history length of 24.

6.1.3 *Misprediction Rates.* Figure 3 shows the arithmetic mean of misprediction rates achieved with increasing hardware budgets on the SPEC 2000 benchmarks. At a 4 KB hardware budget, the global perceptron predictor has

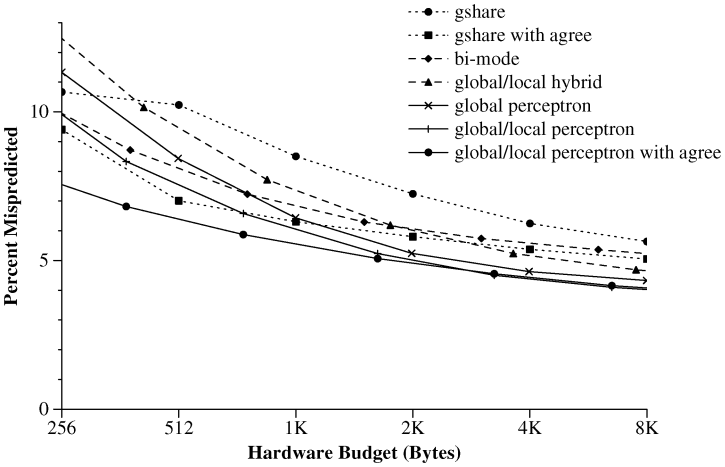

Fig. 3. Hardware budget versus prediction rate on SPEC 2000. This graph shows the misprediction rates of various predictors as a function of the hardware budget.

a misprediction rate of 4.6%, an improvement of 26% over *gshare* at 6.2% and an improvement of 15% over a 6 KB bimode at 5.4%. When both global and local history information are used, the perceptron predictor still has superior accuracy. A global/local hybrid predictor with the same configuration as a 3,712 byte version of the Alpha 21264 predictor has a misprediction rate of 5.2%. A global/local perceptron predictor with 3,315 bytes of state has a misprediction rate of 4.5%, representing a 14% decrease in misprediction rate over the Alpha hybrid. The *agree* mechanism improves accuracy, especially at small hardware budgets. With a small budget of only 750 bytes, the global/local perceptron predictor achieves a misprediction rate of 5.9%, which is less than the misprediction rate of a *gshare* predictor with five times the hardware budget, and about the same as the misprediction rate of a *gshare/agree* predictor with a 2 KB budget. Figure 4 shows the misprediction rates of two PHT-based methods and two perceptron predictors on the SPEC 2000 benchmarks for hardware budgets of 4 KB and 16 KB.

6.2 Accuracy at Large Hardware Budgets

As transistor densities continue to increase dramatically, it makes sense to explore much larger hardware budgets for branch predictors. Evers' [2000] thesis explores the design space for multicomponent hybrid predictors using large hardware budgets, from 18 to 368 KB. Thus, to understand the limits of our approach, we compare the global/local perceptron predictor with Evers' multicomponent hybrid predictor, which to date is the most accurate known fully dynamic predictor.

6.2.1 *Methodology.* Evers' [2000] multicomponent hybrid predictor uses a McFarling-style chooser to select between two other McFarling-style hybrid predictors. The first hybrid component joins a *gshare* that uses a short history to a *gshare* that uses a long history. The other hybrid component consists of

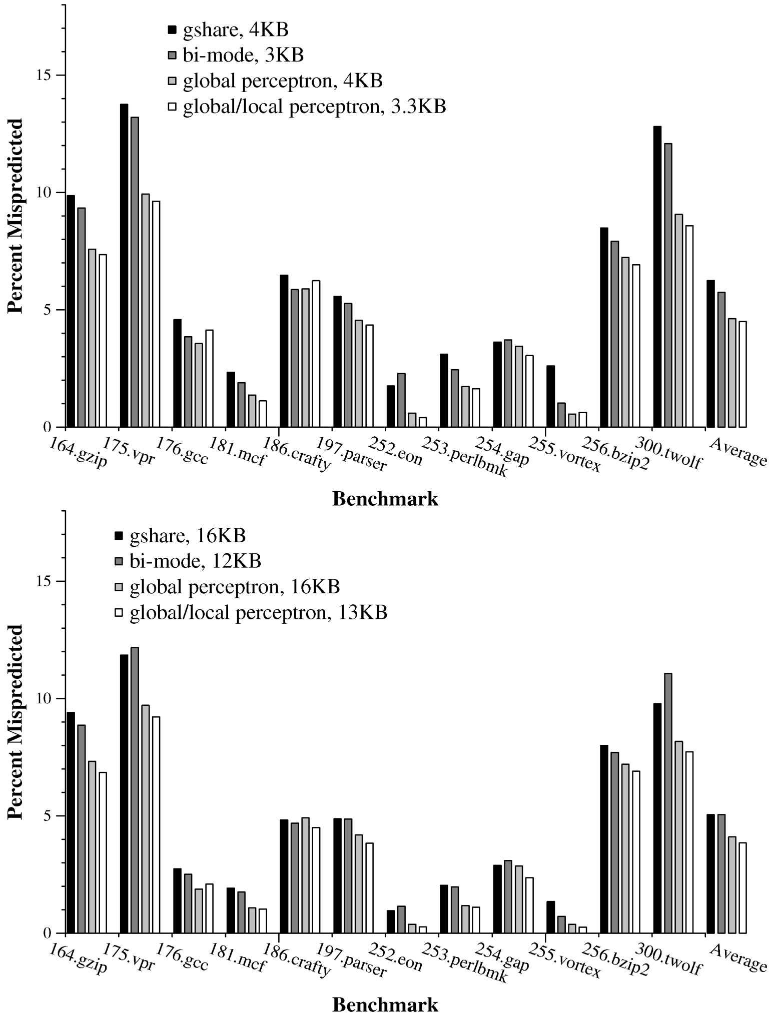

Fig. 4. Misprediction rates for individual benchmarks. These charts show the misprediction rates of global perceptron, *gshare*, and bimode predictors at hardware budgets of 4 KB and 16 KB.

a PAs hybridized with a *loop predictor*, which is capable of recognizing regular looping behavior even for loops with long trip counts.

We simulate Evers' multicomponent predictors using the same configuration parameters given in his thesis. At the same set of hardware budgets, we simulate a global/local version of the perceptron predictor. Due to the huge design space, we do not tune this large perceptron predictor as exhaustively as we do for the smaller hardware budgets. Instead, we tune for the best global history length on the SPEC `train` inputs, and then for the best fraction of global versus local history at a single hardware budget, extrapolating this fraction to the entire set of hardware budgets. As with our previous global/local perceptron experiments, we allocate 35% of the hardware budgets to the table of local histories. The configurations of these large perceptron predictors are given in Table III.

Table III. Configurations for Large Budget Perceptron Predictors

Size (KB)	Global History	Local History	# Perceptrons	# Local Histories
18	38	14	280	2,048
30	40	14	428	4,096
53	50	18	519	8,192
98	54	19	1093	8,192
188	64	23	1652	16,384
368	66	24	3060	32,768

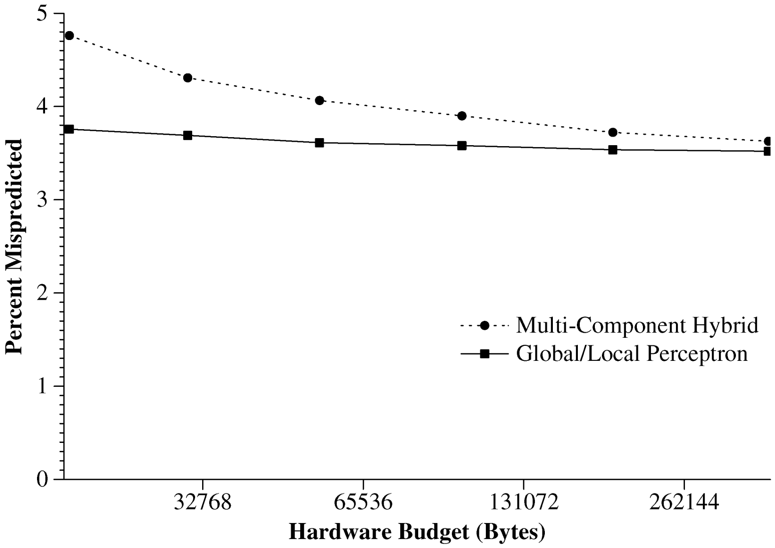

Fig. 5. Hardware budget versus misprediction rate for large predictors.

6.2.2 *Results.* Figure 5 shows, for the SPEC 2000 integer benchmarks, the arithmetic mean misprediction rates of Evers' multicomponent predictor and the global/local perceptron predictor. The perceptron predictor outperforms the multicomponent predictor at every hardware budget, with the misprediction rates converging as the hardware budget increases. Both predictors reach low misprediction rates at the 368 KB hardware budget. The perceptron predictor is slightly more accurate at 3.52% than the multicomponent predictor at 3.62%. Of course, power and delay issues may preclude the use of such huge predictors.

These results provide evidence that the perceptron predictor is currently the most accurate fully dynamic branch predictor. We must emphasize that we have not exhaustively tuned either the multicomponent or the perceptron predictors because of the huge computational challenge. Nevertheless, there is a clear separation between the misprediction rates of the multicomponent and perceptron predictors, and between the perceptron and all other predictors we have examined at lower hardware budgets.

6.3 IPC

We have seen that the perceptron predictor is highly accurate but has a multicycle delay associated with it. If the delay is too large, overall performance may suffer as the processor stalls waiting for predictions. We now evaluate the perceptron predictor in terms of overall processor performance, measured in IPC and taking into account predictor access delay.

6.3.1 *Methodology.* To evaluate overall processor performance, we compare an overriding perceptron predictor (see Section 5.4) against the overriding hybrid predictor of the Alpha 21264. In each case, we use as a first-level predictor a simple 256-entry Smith predictor [Smith 1981], that is, a simple one-level table of two-bit saturating counters indexed by branch address. This fast predictor roughly simulates the line predictor of the overriding Alpha predictor, and it achieves an arithmetic mean accuracy of 85.0%, which is consistent with the accuracy quoted for the Alpha line predictor [Kessler 1999]. For the second-level predictor, we simulate both the global/local perceptron predictor and the Alpha hybrid predictor, which both incur a single-cycle penalty when they override the Smith predictor.

Since performance is dependent on clock rate, we consider two processor configurations. One configuration uses a moderate clock rate that matches the latest Alpha processor, and the other approximates the more aggressive clock rate and deeper pipeline of the Intel Pentium 4. We again simulate the 12 SPEC 2000 integer benchmarks, this time allowing each benchmark to execute two billion instructions.

The details of the overriding predictors depend on clock rate assumptions, so for each of the two clock rates, we first describe our predictor configurations and then report on simulated IPC.

6.3.2 *Moderate Clock Rate Simulations.* Our moderate clock rate simulations use an 833-MHz clock, which matches that of the fastest Alpha processor in 180-nm technology. We also simulate the seven-cycle misprediction penalty of the Alpha 21264. At this clock rate, both the perceptron predictor and Alpha hybrid predictor deliver a prediction in two clock cycles.

We use a perceptron predictor that has 133 perceptrons and uses a history length of 23. Although our simulations show that a history length of 24 is the most accurate at this hardware budget, a history length of 23 gives much the same accuracy while being 10% faster. We have observed that the ideal ratio of per-branch history bits to total history bits is roughly 20%, so rather than tune this predictor exhaustively, we use 19 bits of global history and 4 bits of per-branch history from a table of 1,024 histories. The total state required for this predictor is 3,704 bytes, approximately the same as the 3,712 bytes used by the Alpha hybrid predictor.

We also simulate a 2,048-entry non-overriding *gshare* predictor for reference. This *gshare* uses less state since it operates in a single cycle; note that this is the amount of state allocated to the branch predictor in the HP-PA/RISC 8500 [Lesartre and Hunt 1997], which uses a clock rate similar to that of the Alpha.

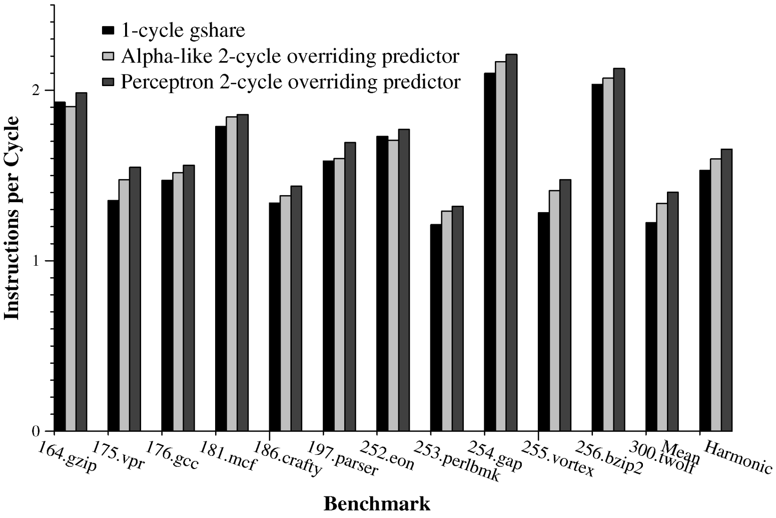

Fig. 6. IPC for overriding perceptron and hybrid predictors. This chart shows the IPCs yielded by *gshare*, an Alpha-like hybrid, and global/local perceptron predictor given a seven-cycle misprediction penalty. The hybrid and perceptron predictors have a two-cycle latency and are used as overriding predictors with a small Smith predictor.

Figure 6 shows the IPC for each of the predictors. Even though there is a penalty when the overriding Alpha and perceptron predictors disagree with the Smith predictor, their increased accuracies more than compensate for this effect, achieving higher IPCs than a single-cycle *gshare*. The perceptron predictor yields a harmonic mean IPC of 1.65, which is higher than the overriding predictor at 1.59, which itself is higher than *gshare* at 1.53.

6.3.3 *Aggressive Clock Rate Simulations.* The current trend in microarchitecture is to create deeply pipelined microprocessors, sacrificing some IPC for the ability to use much higher clock rates. For instance, the Intel Pentium 4 uses a 20-stage integer pipeline at a clock rate of 1.76 GHz. In this situation, one might expect the perceptron predictor to yield poor performance, since it requires so much time to make a prediction relative to the short clock period. We thus present results for a more aggressively clocked microarchitecture that has characteristics of the Pentium 4. We show, in fact, that the perceptron predictor can improve performance even more than in the previous case, because the benefits of low misprediction rates are greater.

At a 1.76-GHz clock rate, the perceptron predictor described above would take four clock cycles: one to read the table of perceptrons and three to propagate signals to compute the perceptron output. Pipelining the perceptron predictor will allow us to get one prediction each cycle, so that branches that come close together don't have to wait until the predictor is finished predicting the previous branch. The Wallace-tree for this perceptron has seven levels. With a small cost in latch delay, we can pipeline the Wallace-tree in four stages: one to read the perceptron from the table, another for the first three levels of the tree, another for the second three levels, and a fourth for the final level and the

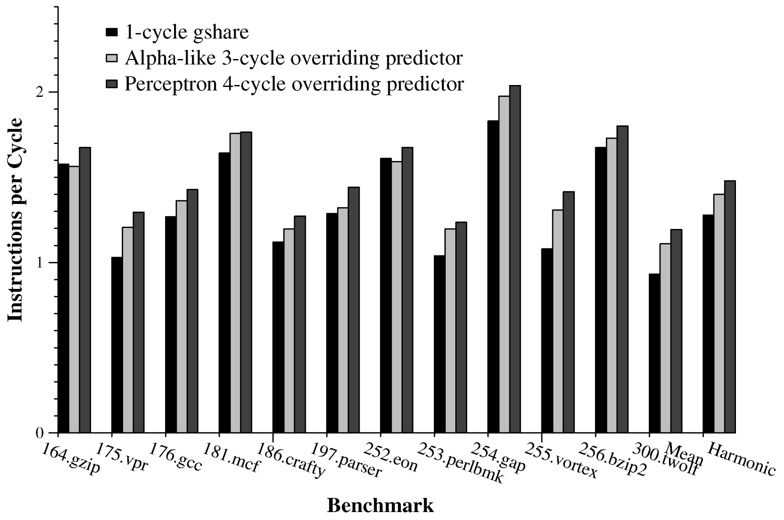

Fig. 7. IPC for overriding perceptron and hybrid predictors with long pipelines. This chart shows the IPCs yielded by *gshare*, a hybrid predictor, and a global/local perceptron predictor with a large misprediction penalty and high clock rate.

carry-lookahead adder at the root of the tree. The new perceptron predictor operates as follows.

1. When a branch is encountered, it is immediately predicted with a small Smith predictor. Execution continues along the predicted path.
2. Simultaneously, the local history table and perceptron tables are accessed using the branch address as an index.
3. The circuit that computes the perceptron output takes its input from the global and local history registers and the perceptron weights.
4. Four cycles after the initial prediction, the perceptron prediction is available. If it differs from the initial prediction, instructions executed since that prediction are squashed and execution continues along the other path.
5. When the branch executes, the corresponding perceptron is quickly trained and stored back to the table of perceptrons.

We use a misprediction penalty of 20 cycles, which simulates the long pipeline of the Pentium 4. The Alpha overriding hybrid predictor is conservatively scaled to take three clock cycles, and the overriding perceptron predictor takes four clock cycles. The 2,048-entry *gshare* predictor is unmodified.

Figure 7 shows the performance results for this aggressive clock rate. Even though the perceptron predictor takes longer to make a prediction, it still yields the highest IPC in all benchmarks because of its superior accuracy. The perceptron predictor yields an IPC of 1.48, which is 5.7% higher than that of the hybrid predictor at 1.40.

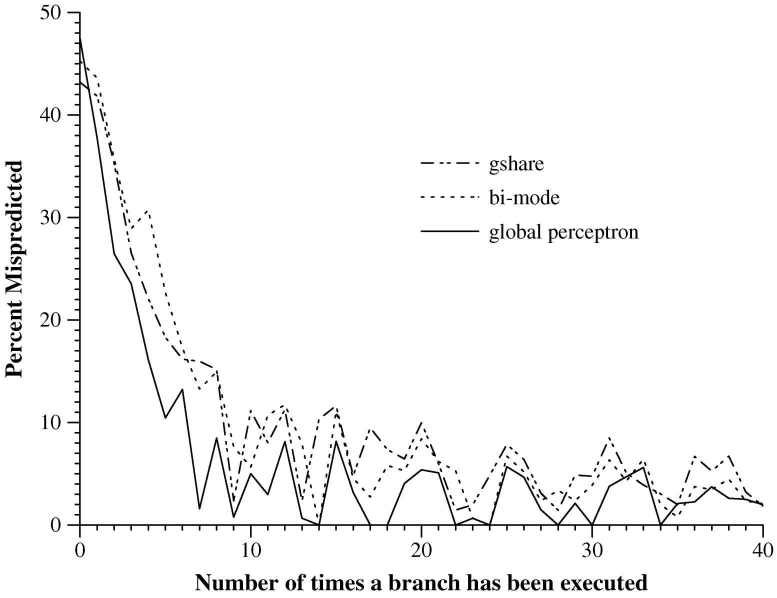

Fig. 8. Average training times for the SPEC 2000 benchmarks. The x-axis is the number of times a branch has been executed. The y-axis is the average, over all branches in the program, of 1 if the branch was mispredicted, 0 otherwise. Over time, this statistic tracks how quickly each predictor learns. The perceptron predictor achieves greater accuracy earlier than the other two methods.

6.4 Training Times

To compare the training speeds of the perceptron predictor with PHT methods, we examine the first 100 times each branch in each of the SPEC 2000 benchmarks is executed (for those branches executing at least 100 times). Figure 8 shows the average accuracy of each of the 100 predictions for each of the static branches with a 4 KB hardware budget. The average is weighted by the relative frequencies of each branch.

The perceptron method learns more quickly than *gshare* or bimode. For the perceptron predictor, training time is independent of history length. For techniques such as *gshare* that index a table of counters, training time depends on the amount of history considered; a longer history may lead to a larger working set of two-bit counters that must be initialized when the predictor is first learning the branch. This effect has a negative impact on prediction rates, and at a certain point, longer histories begin to hurt performance for these schemes [Michaud et al. 1997]. As we show in the next section, the perceptron prediction does not have this weakness, as it always does better with a longer history length.

6.5 Advantages of the Perceptron Predictor

We hypothesize that the main advantage of the perceptron predictor is its ability to make use of longer history lengths. Schemes such as *gshare* that use the history register as an index into a table require space exponential in the history

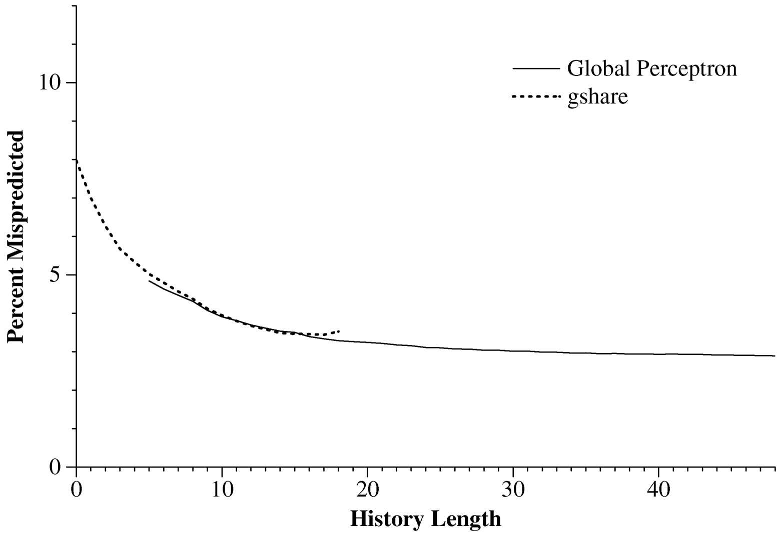

Fig. 9. History length versus performance. This graph shows how accuracy for *gshare* and the perceptron predictor improves as history length is increased. The perceptron predictor is able to consider much longer histories with the same hardware budget.

length, whereas the perceptron predictor requires space linear in the history length.

To provide experimental support for our hypothesis, we simulate *gshare* and the perceptron predictor at a 64 KB hardware budget, where the perceptron predictor normally outperforms *gshare*. However, by only allowing the perceptron predictor to use as many history bits as *gshare* (18 bits), we find that *gshare* performs better, with a misprediction rate of 4.83% compared with 5.35% for the perceptron predictor. The inferior performance of this crippled predictor is likely due to increased destructive aliasing, as perceptrons are larger, and thus fewer, than *gshare*'s two-bit counters.

Figure 9 shows the result of simulating *gshare* and the perceptron predictor with varying history lengths on the SPEC 2000 benchmarks. Here, we use a 4 MB hardware budget to allow *gshare* to consider longer history lengths than usual. As we increase history length, *gshare* becomes more accurate until it degrades slightly at 18 bits and then runs out of bits (since *gshare* requires resources exponential in the number of history bits). By contrast, the perceptron predictor's accuracy only improves with longer histories. With this unrealistically large hardware budget, *gshare* performs best with a history length of 17, where it achieves a misprediction rate of 3.4%. The perceptron predictor is best at a history length of 48, where it achieves a misprediction rate of 2.9%.

6.6 Impact of Linearly Inseparable Branches

In Section 4.3 we pointed out a fundamental limitation of perceptrons that perform offline training: they cannot learn linearly inseparable functions. We now explore the impact of this limitation on branch prediction.

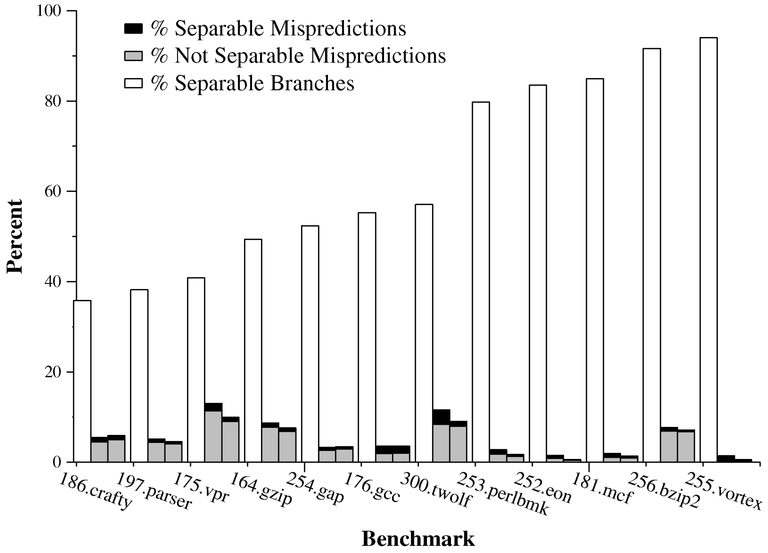

Fig. 10. Linear separability versus accuracy at a 4 KB budget. For each benchmark, the leftmost bar shows the number of linearly separable dynamic branches in the benchmark, the middle bar shows the misprediction rate of *gshare* at a 4 KB hardware budget, and the right bar shows the misprediction rate of the perceptron predictor at the same hardware budget.

To relate linear separability to branch prediction, we define the notion of *linearly separable branches*. Let h_n be the most recent n bits of global branch history. For a static branch B, there exists a Boolean function $f_B(h_n)$ that best predicts B's behavior. It is this function, f_B, that all branch predictors strive to learn. If f_B is linearly separable, we say that branch B is a linearly separable branch; otherwise, B is a *linearly inseparable branch*.

Theoretically, offline perceptrons cannot predict linearly inseparable branches with complete accuracy, whereas PHT-based predictors have no such limitation when given enough training time. Does *gshare* predict linearly inseparable functions better than the perceptron predictor? To answer this question, we compute $f_B(h_{14})$ for each static branch B in our benchmark suite and test whether the functions are linearly separable.

Figure 10 shows the misprediction rates for each benchmark for a 4 KB budget, as well as the percentage of dynamically executed branches that are linearly inseparable. For each benchmark, the bar on the left shows the misprediction rate of *gshare*, and the bar on the right shows the misprediction rate of a global perceptron predictor. Each bar also shows, using shading, the portion of mispredictions due to linearly inseparable branches and linearly separable branches. We observe two interesting features of this chart. First, most mispredicted branches are linearly inseparable, so linear inseparability correlates highly with unpredictability in general. Second, although it is difficult to determine whether the perceptron predictor performs worse than *gshare* on linearly inseparable branches, we do see that the perceptron predictor outperforms *gshare* in all cases except for `186.crafty`, the benchmark with the highest fraction of linearly inseparable branches.

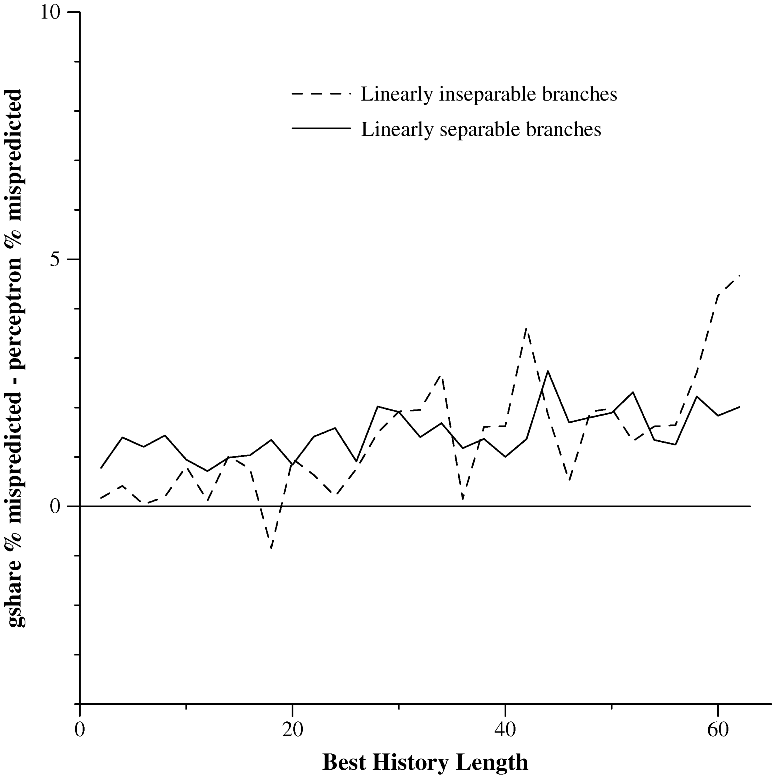

Fig. 11. Classifying the advantage of the perceptron predictor. Each datapoint is the average difference in misprediction rates of the perceptron predictor and *gshare* (on the x-axis) for those branches (on the y-axis). Above the x-axis, the perceptron predictor is better on average. Below the x-axis, *gshare* is better on average. For linearly separable branches, our predictor is on average more accurate than *gshare*. For inseparable branches, our predictor is sometimes less accurate for branches that require short histories, and it is more accurate on average for branches that require long histories.

Some branches require longer histories than others for accurate prediction, and the perceptron predictor often has an advantage for these branches. Figure 11 shows the relationship between this advantage and the required history length, with one curve for linearly separable branches and one for inseparable branches. The y-axis represents the advantage of our predictor, computed by subtracting the misprediction rate of the perceptron predictor from that of *gshare*. We sorted all static branches according to their "best" history length, which is represented on the x-axis. Each datapoint represents the average misprediction rate of static branches (without regard to execution frequency) that have a given best history length. We use the perceptron predictor in our methodology for finding these best lengths: using a perceptron trained for each branch, we find the most distant of the three weights with the greatest magnitude. This methodology is motivated by the work of Evers et al. [1998] who show that most branches can be predicted by looking at three previous branches. As the best history length increases, the advantage of the perceptron predictor generally increases as well. We also see that our predictor is more accurate for linearly separable branches. For linearly inseparable branches, our predictor generally performs better when the branches require long histories, and *gshare* sometimes performs better when branches require short histories.

Linearly inseparable branches requiring longer histories, as well as all linearly separable branches, are always better predicted with the perceptron predictor. Linearly inseparable branches requiring fewer bits of history are better predicted by *gshare*. Thus, the longer the history required, the better is the performance of the perceptron predictor, even on the linearly inseparable branches.

6.7 Additional Advantages of the Perceptron Predictor

6.7.1 *Assigning Confidence to Decisions.* Our predictor can provide a confidence level in its predictions that can be useful in guiding hardware speculation. The output y of the perceptron predictor is not a Boolean value, but a number that we interpret as *taken* if $y \geq 0$. The value of y provides important information about the branch since the distance of y from 0 is proportional to the certainty that the branch will be taken [Jiménez and Walsh 1998]. This confidence can be used, for example, to allow a microarchitecture to speculatively execute both branch paths when confidence is low, and to execute only the predicted path when confidence is high. Some branch prediction schemes explicitly compute a confidence in their predictions [Jacobsen et al. 1996], but in our predictor this information comes for free. We have observed experimentally that the probability that a branch will be taken can be accurately estimated as a linear function of the output of the perceptron predictor.

6.7.2 *Analyzing Branch Behavior with Perceptrons.* Perceptrons can be used to analyze correlations among branches. The perceptron predictor assigns a weight to each bit in the branch history. When a particular bit is strongly correlated with a particular branch outcome, the magnitude of the weight is higher than when there is less or no correlation. Thus, the perceptron predictor learns to recognize the bits in the history of a particular branch that are important for prediction, and it learns to ignore the unimportant bits. This property of the perceptron predictor can be used with profiling to provide feedback for other branch prediction schemes. For example, the methodology that we use in Section 6.6 could be used with a profiler to provide path length information to the variable length path predictor [Stark et al. 1998].

7. CONCLUSIONS

In this article we have introduced a new branch predictor that uses neural learning techniques—the perceptron in particular—as the basic prediction mechanism. The key advantage of perceptrons is their ability to use long history lengths without requiring exponential resources. These long history lengths lead to extremely high accuracy. In particular, for the SPEC 2000 integer benchmarks, our new global/local perceptron has 36% fewer mispredictions than a McFarling-style hybrid predictor, which is the most accurate known predictor that has been implemented in silicon. Our global/local perceptron is also more accurate than the multicomponent predictor of Evers et al., which was previously the most accurate known predictor in the literature.

A potential weakness of perceptrons is their increased computational complexity when compared with two-bit counters, but we have shown how a perceptron predictor can be implemented efficiently with respect to both area and delay. In particular, we believe that the most feasible implementation is the overriding perceptron predictor, which uses a simple Smith predictor to provide a quick prediction that can be later overridden. For an aggressive 1.76-GHz clock rate, this overriding predictor provides an IPC improvement of 5.7% over a McFarling-style hybrid predictor. Another weakness of perceptrons is their inability to learn linearly inseparable functions, but we have shown that this is a limitation of existing branch predictors as well.

This article has also shown that there is benefit to considering longer history lengths than those previously considered. Variable length path branch prediction considers history lengths of up to 23 [Stark et al. 1998], and a study of the effects of long branch histories on branch prediction only considers lengths up to 32 [Evers et al. 1998]. We have found that additional performance gains can be found for branch history lengths of up to 66.

Finally, perceptrons have interesting characteristics that open up new avenues for future work. As noted in Section 6.7, perceptrons can also be used to guide speculation based on branch prediction confidence levels, and perceptron predictors can be used in recognizing important bits in the history of a particular branch.

ACKNOWLEDGMENTS

We thank the anonymous referees for their valuable comments. We are also grateful to Steve Keckler and Kathryn McKinley for many stimulating discussions on this topic, and we thank Steve, Kathryn, Sam Guyer, and Ibrahim Hur for their comments on earlier versions of this article.

REFERENCES

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

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

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., JONES, M., LINDSAY, D., MARTIN, J., MOZER, M., AND ZORN, B. 1997. Evidence-based static branch prediction using machine learning. *ACM Trans. Program. Lang. Syst. 19*, 1, 188–222.

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

EDEN, A. AND MUDGE, T. 1998. The YAGS branch prediction scheme. In *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture*, 69–80.

EMER, J. AND GLOY, N. 1997. A language for describing predictors and its application to automatic synthesis. In *Proceedings of the 24th International Symposium on Computer Architecture*, 304–314.

EVERS, M. 2000. Improving branch prediction by understanding branch behavior. PhD Thesis, University of Michigan, Department of Computer Science and Engineering.

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.

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

GOMEZ, F., BURGER, D., AND MIIKKULAINEN, R. 2001. A neuroevolution method for dynamic resource allocation on a chip multiprocessor. In *Proceedings of the International Joint Conference on Neural Networks (IJCNN-01)*, 2355–2360.

HENNESSY, J. L. AND PATTERSON, D. A. 1996. *Computer Architecture: A Quantitative Approach, second edition*. Morgan Kaufmann, San Francisco.

JACOBSEN, E., ROTENBERG, E., AND SMITH, J. E. 1996. Assigning confidence to conditional branch predictions. In *Proceedings of the 29th Annual International Symposium on Microarchitecture*, 142–152.

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

JIMÉNEZ, D. A. AND WALSH, N. 1998. Dynamically weighted ensemble neural networks for classification. In *Proceedings of the 1998 International Joint Conference on Neural Networks*, 753–756.

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.

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

KULKARNI, A. D. 1993. *Artificial Neural Networks for Image Understanding*. Van Nostrand Reinhold, New York.

LEE, C.-C., CHEN, C., AND MUDGE, T. 1997. The bi-mode branch predictor. In *Proceedings of the Thirtieth Annual International Symposium on Microarchitecture*, 4–13.

LESARTRE, G. AND HUNT, D. 1997. PA-8500: The continuing evolution of the PA-8000 family. In *42nd IEEE International Computer Conference*.

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

MICHAUD, P., SEZNEC, A., AND UHLIG, R. 1997. Trading conflict and capacity aliasing in conditional branch predictors. In *Proceedings of the 24th International Symposium on Computer Architecture*, 292–303.

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

SECHREST, S., LEE, C.-C., AND MUDGE, T. 1996. Correlation and aliasing in dynamic branch predictors. In *Proceedings of the 23rd International Symposium on Computer Architecture*, 22–32.

SETIONO, R. AND LIU, H. 1995. Understanding neural networks via rule extraction. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, 480–485.

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

SPRANGLE, E., CHAPPELL, R., ALSUP, M., AND PATT, Y. N. 1997. The Agree predictor: A mechanism for reducing negative branch history interference. In *Proceedings of the 24th International Symposium on Computer Architecture*, 284–291.

STARK, J., EVERS, M., AND PATT, Y. N. 1998. Variable length path branch prediction. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, 170–179.

VINTAN, L. AND IRIDON, M. 1999. Towards a high performance neural branch predictor. In *Proceedings of the International Joint Conference on Neural Networks (IJCNN)*, vol. 2. 868–873.

WANG, K. AND FRANKLIN, M. 1997. Highly accurate data value prediction using hybrid predictors. In *Proceedings of the Thirtieth Annual International Symposium on Microarchitecture*, 281–291.

WIDROW, B. AND HOFF JR., M. 1960. Adaptive switching circuits. In *IRE WESCON Convention Record, part 4*, 96–104.

YEH, T.-Y. AND PATT, Y. N. 1991. Two-level adaptive branch prediction. In *Proceedings of the 24th ACM/IEEE International Symposium on Microarchitecture*, 51–61.

Received January 2002; revised April 2002; accepted May 2002