ResearchGate

See discussions, stats, and author profiles for this publication at: https://www.researchgate.net/publication/2835582

# Neural Methods for Dynamic Branch Prediction

**Article** *in* ACM Transactions on Computer Systems · November 2002

DOI: 10.1145/571637.571639 · Source: CiteSeer

CITATIONS
177

READS
992

**2 authors:**

Daniel Andrés Jimenez
Universidade Federal da Integração Latino-Americana
**14** PUBLICATIONS **979** CITATIONS

SEE PROFILE

Calvin Lin
University of Texas at Austin
**137** PUBLICATIONS **3,639** CITATIONS

SEE PROFILE

All content following this page was uploaded by Daniel Andrés Jimenez on 07 August 2013.

The user has requested enhancement of the downloaded file.

# Neural Methods for Dynamic Branch Prediction

Daniel A. Jiménez    Calvin Lin

Department of Computer Sciences
The University of Texas at Austin
Austin, TX 78712
{djimenez,lin}@cs.utexas.edu

This report is a revised and expanded version of our conference paper, *Dynamic Branch Prediction with Perceptrons*, in Proceedings of the Seventh International Symposium on High Performance Computer Architecture (HPCA-7) pp. 197 – 206, Monterrey, NL, Mexico 2001

## Abstract

*This paper presents a new method for branch prediction that is highly accurate. 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 4K byte hardware budget, a simple version of our method that uses a global history achieves a misprediction rate of 1.94% on the SPEC 2000 integer benchmarks, an improvement of 53% over* gshare*. We also describe a global/local version of our predictor that is 36% more accurate than the McFarling-style hybrid predictor of the Alpha 21264. Because this variant of the perceptron predictor is more accurate than Evers' multi-component predictor for hardware budgets of up to 256KB, 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.*

# 1 Introduction

Modern computer architectures increasingly rely on speculation to boost instruction-level parallelism. For example, data that is likely to be read in the near future is speculatively prefetched, and predicted values are speculatively used before actual values are available [13, 30]. 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 paper, 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 and the benefit of accurate branch prediction increases. Recent efforts to improve branch prediction focus primarily on eliminating *aliasing* in two-level adaptive predictors [22, 20, 28, 6], 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 these 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 such from which these neural networks are built. These artificial neurons, such as the perceptron [24], 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 paper, we explore various types of artificial neurons and propose a two-level scheme that uses perceptrons instead of two-bit counters. A key advantage of our approach is its ability to utilize long branch history lengths. In our predictor, each static branch is ideally allocated its own perceptron to predict the branch outcome, and the space required by our scheme scales linearly with the history length. In contrast, traditional two-level adaptive schemes use a pattern history table (PHT) of two-bit saturating counters, indexed by a history shift register that stores the outcomes of previous branches. This PHT structure limits the length of the history register to the logarithm of the number of counters. Thus, for the same hardware budget, our predictor can consider much longer histories than PHT-based schemes, which leads to high accuracy. For example, for a 4KB hardware budget, a PHT-based predictor can use a history length of 14, while a version of our predictor can use a history length of 34.

This paper 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 while perceptrons are unable to predict linearly inseparable branches, PHT-based schemes also have trouble predicting such branches.

This paper makes the following contributions:

- We introduce the perceptron predictor, 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 4K byte hardware budget, our global predictor improves misprediction rates on the SPEC 2000 integer benchmarks by 53% over a *gshare* predictor of the same size and by 27% over the McFarling-style hybrid predictor of the Alpha 21264. [1]

[1] These results differ from our previously published numbers because our new methodology uses the Alpha instruction set,

- We describe a version of the perceptron predictor that uses both global and per-branch information, yielding misprediction rates that are 36% 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 this predictor is more accurate than Evers' multi-component predictor [10], 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 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.

# 2 Related Work

In this section, we explore 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 [32]. 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 [25], and many of the recently proposed branch predictors seek to reduce the aliasing problem [22, 20, 28, 6] 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 [6].

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 [9]. 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 [23].

Variable length path branch prediction [29] 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 multi-pass 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.

which allows us to get simulated IPC results from SimpleScalar [3]. We discuss the impact of this methodological change in Section 6.1.

**Static branch prediction with neural networks.** Neural networks have been used to perform *static* branch prediction [4], 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 an 80% correct prediction rate, compared to 75% for static heuristics [1, 4]. 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 [28].

**Branch prediction and genetic algorithms.** Neural networks are part of the field of machine learning, which also includes genetic algorithms. Emer and Gloy use genetic algorithms to "evolve" branch predictors [7], 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 on-line. In fact, Emer and Gloy's approach cannot describe our new predictor.

**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 [12]. 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 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 [11], and image understanding [19, 15]. The general idea of neural computation is that many processing nodes, known as *neurons*, are connected to each other in a network. Data is 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*, i.e., predicting which of a set of classes a particular instance will fall into. Suppose a set $S$ is partitioned into $n$ classes, and we are faced with the problem of determining, for an arbitrary element $s \in S$, what class $s$ is in. The elements of $S$ have certain features which 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 a weight associated with it that determines the strength of the connection [11]. 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 generalize 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 $n = 2$, there is only one output neuron; in this case, we classify $s$ according to whether the output value exceeds a certain threshold, typically 0 or $\frac{1}{2}$.

### 3.2 Potential Applications to Microarchitecture

Neural learning methods have the potential to enhance microarchitectural techniques, replacing the more primitive predictors currently used. Some of the possible applications not yet explored are:

- 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 or not 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 on-line 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.

- Cache Replacement Policy. Neural networks could be used to adapt to the behavior of program access pattern and implement specialized cache replacement policies for reduced cache miss rates.

- 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 the trace cache.

### 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 [31], Hebb learning [11], and the Block perceptron [2] 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 multi-layer perceptron with back-propagation [11]. 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. While ADALINE yielded similar prediction accuracy to the perceptron, the ADALINE neurons require twice as much space 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*, i.e., 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 [26], 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. Our method is essentially a two-level predictor, replacing the pattern history table with a table of perceptrons.

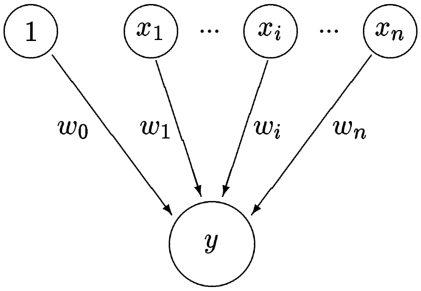

Figure 1: Perceptron Model. The input values $x_1, ..., x_n$, are propagated through the weighted connections by taking their respective products with the weights $w_1, ..., w_n$. These products are summed, along with the bias weight $w_0$, to produce the output value $y$.

## 4.1 How Perceptrons Work

The perceptron was introduced in 1962 [24] as a way to study brain function. We consider the simplest of many types of perceptrons [2], 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 $t(x_1, ..., x_n)$ 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 and negative correlations between branch outcomes in the global history and the branch being predicted.

Figure 1 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*, i.e., each $x_i$ is either -1, meaning *not taken* or 1, meaning *taken.* A negative output is interpreted as *predict not taken.* A non-negative 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 $\theta$ 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 $|y_{out}| \leq \theta$ then
        for $i := 0$ to $n$ do
                $w_i := w_i + tx_i$
        end for
end if
```

Since $t$ and $x_i$ are always either -1 or 1, this algorithm increments the $i^{\text{th}}$ 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 [11]. 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* [11]. 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 separating *true* instances of the exclusive-OR function from *false* ones on the Boolean plane.

As we will 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 non-linearity 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 Branch Prediction with Perceptrons

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 $i^{\text{th}}$ 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 $i^{\text{th}}$ 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 Implementation

This section explores the design space for perceptron predictors and discusses details of a circuit-level implementation.

## 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.

**History length.** Long history lengths can yield more accurate predictions [9] but 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.

**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.

**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.

**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 paper, we report only results that simulate this complementation idea.

**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
```

**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.

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 $i^{\text{th}}$ history bit is set, then this operation has the effect of taking the one's-complement of the $i^{\text{th}}$ 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 [5], which reduces the problem of finding the sum of $n$ numbers to the problem of finding the sum of 2 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.

Table 1 shows the delay of the perceptron predictor for several hardware budgets and history lengths, simulated with HSPICE and CACTI for 180nm 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 of the input signals to cross half of $V_{DD}$ until the time the perceptron predictor yields a steady, usable signal. For a 4KB 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 2 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 2 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 instead of 24 takes only 2.2 nanoseconds; it is about 10% faster because a predictor with 24 weights (23 for history plus 1 for bias) can be organized more efficiently than predictor with 25 weights, for reasons specific to our Wallace-tree design.

| 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 | 1K | 1090 | 468 | 1558 | 1.3 | 2.7 |
| 17 | 2K | 1170 | 504 | 1674 | 1.4 | 2.9 |
| 23 | 4K | 1700 | 571 | 2271 | 1.9 | 4.0 |
| 24 | 4K | 1860 | 571 | 2431 | 2.0 | 4.3 |

Table 1: Perceptron Predictor Delay. This table shows the simulated delay of the perceptron predictor at several table sizes and history length configurations. The delay of computing the output and fetching the perceptron from the table are shown separately, as well as in total.

**Compensating for Delay.** Ideally, a branch predictor operates in a single processor clock cycle. Jiménez *et al.* study a number of techniques for reducing the impact of delay on branch predictors [16]. For example, a *cascading* perceptron predictor would use a simple predictor to anticipate the address of the next branch to be fetched, and it would use a perceptron to begin predicting the anticipated address. If the branch were to arrive before the perceptron 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 similar predictor, using two *gshare* tables, is able to use the larger table 47% of the time.

An *overriding* perceptron predictor would use a quick *gshare* predictor to get an immediate prediction, starting a perceptron prediction at the same time. The *gshare* prediction is acted upon by the fetch engine. Once the perceptron prediction completes, both predictions are compared. If they differ, the actions taken by the fetch engine are rolled back and restarted with the new prediction, incurring a small penalty. The Alpha 21264 uses this kind of branch predictor, with a slower hybrid branch predictor overriding a less accurate but faster line predictor [18]. When a line prediction is overridden, the Alpha predictor incurs a single-cycle penalty, which is small compared to the 7-cycle penalty for a branch misprediction. By pipelining the perceptron predictor, or using the hierarchical techniques mentioned above, the perceptron predictor can be used successfully in future microprocessors. The overriding strategy seems particularly appropriate since, as pipelines continue to become deeper, the cost of overriding a less accurate predictor decreases as a percentage of the cost of a full misprediction. We present a detailed analysis of the overriding scheme in Section 6.4.

# 6 Results and Analysis

To evaluate the perceptron predictor, we use simulation to compare it against well-known techniques from the literature. We first compare the accuracy of two versions of the perceptron predictor against 5 predictors. We then evaluate performance using IPC as the metric, comparing an overriding perceptron predictor against an overriding McFarling-style predictor at two different clock rates. Finally, we present analysis to explain why the perceptron predictor performs well.

### 6.1 Methodology

Here we describe our experimental methodology. We discuss the other predictors simulated, the benchmarks used, the tuning of the predictors, and other issues.

**Predictors simulated.** We compare our new predictor against *gshare* [22], and bi-mode [20], and a combination *gshare* and PAg McFarling-style hybrid predictor [22] similar to that of the Alpha 21264, with all tables scaled exponentially for increasing hardware budgets. For the perceptron predictor, we simulate both a purely global predictor, as well as a predictor that uses both global and local history. This global/local predictor takes some input to the perceptron from the global history register, and other input from a set of per-branch histories. 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 *gshare* and the perceptron predictors, we also simulate the *agree* mechanism [28], 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 [17], 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.

**Gathering traces.** We use SimpleScalar/Alpha [3] to gather traces. Each time the simulator executes a conditional branch, it records the branch address and outcome in a trace file. Branches in libraries are not profiled. The traces are fed to a program that simulates the different branch prediction techniques.

**Benchmarks simulated.** We use the 12 SPEC 2000 integer benchmarks. We allow each benchmark to execute 300 million branches, which causes each benchmark to execute at least one billion instructions. We skip past the first 50 million branches in the trace to measure only the steady state prediction accuracy, without effects from the benchmarks' initializations. For tuning the predictors, we use the SPEC `train` inputs. For reporting misprediction rates, we test the predictors on the `ref` inputs.

**Tuning the predictors.** We tune each predictor for history length using traces gathered from the 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 yielding the lowest harmonic 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 length 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 best value of the threshold by using an intelligent search of the space of values, pruning areas of the space that give poor performance. We re-use the same thresholds for the global/local and *agree* perceptron predictors.

Table 2 shows the results of the history length tuning. We find an interesting relationship between history length and threshold: the best threshold $\theta$ for a given history length $h$ is always *exactly* $\theta = \lfloor 1.93h + 14 \rfloor$. 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 8 bits for the perceptron weights yields the best results.

### 6.2 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 can occur at a large distance from each other [9]. 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 2, the perceptron predictor works best with much longer histories than the other two predictors. For example, with a 4K byte 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.

| hardware budget in bytes | *gshare* |  | global perceptron |  | global/local perceptron |  |
|---|---|---|---|---|---|---|
|  | history length | number of entries | history length | number of entries | global/local history | number of entries |
| 128 | 2 | 512 | 4 | 25 | 8/2 | 11 |
| 256 | 1 | 1K | 7 | 32 | 10/2 | 19 |
| 512 | 11 | 2K | 9 | 51 | 23/2 | 19 |
| 1K | 12 | 4K | 13 | 73 | 25/5 | 33 |
| 2K | 13 | 8K | 17 | 113 | 31/5 | 55 |
| 4K | 14 | 16K | 24 | 163 | 34/10 | 91 |
| 8K | 15 | 32K | 28 | 282 | 34/10 | 182 |
| 16K | 16 | 64K | 47 | 348 | 36/11 | 341 |

Table 2: Best History Lengths. This table shows the best amount of global history to keep for *gshare* and two versions of the perceptron predictor, as well as the number of table entries for each predictor.

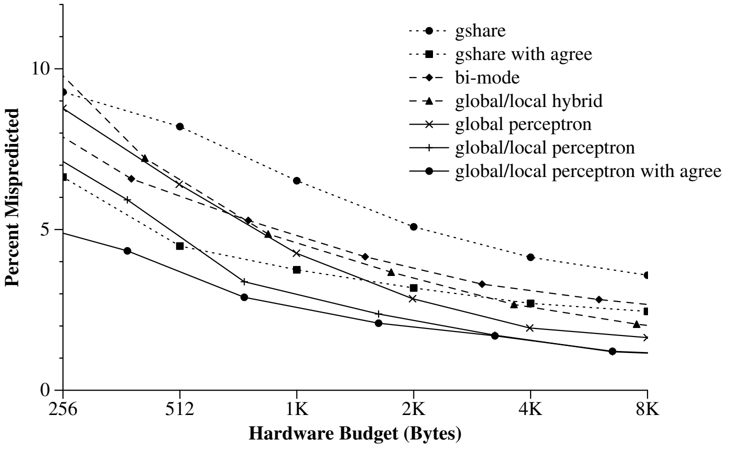

Figure 2: Hardware Budget vs. Prediction Rate on SPEC 2000. This graph shows the misprediction rates of various predictors as a function of the hardware budget.

## 6.3 Misprediction Rates

Figure 2 shows the harmonic mean of misprediction rates achieved with increasing hardware budgets on the SPEC 2000 benchmarks. At a 4K byte hardware budget, the global perceptron predictor has a misprediction rate of 1.94%, an improvement of 53% over *gshare* at 4.13% and 31% over a 6K byte bi-mode at 2.82%. When both global and local history information is used, the perceptron predictor still has superior accuracy. A global/local hybrid predictor with the same configuration as the Alpha 21264 predictor using 3712 bytes has a misprediction rate of 2.67%. A global/local perceptron predictor with 3315 bytes of state has a misprediction rate of 1.71%, representing a 36% 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 2.89%, which is less than the misprediction rate of a *gshare* predictor with 11 times the hardware budget, and less than the misprediction rate of a *gshare/agree* predictor with a 2K byte budget. Figure 3 show the misprediction rates of two PHT-based methods and two perceptron predictors on the SPEC 2000 benchmarks for hardware budgets of 4K and 16K bytes.

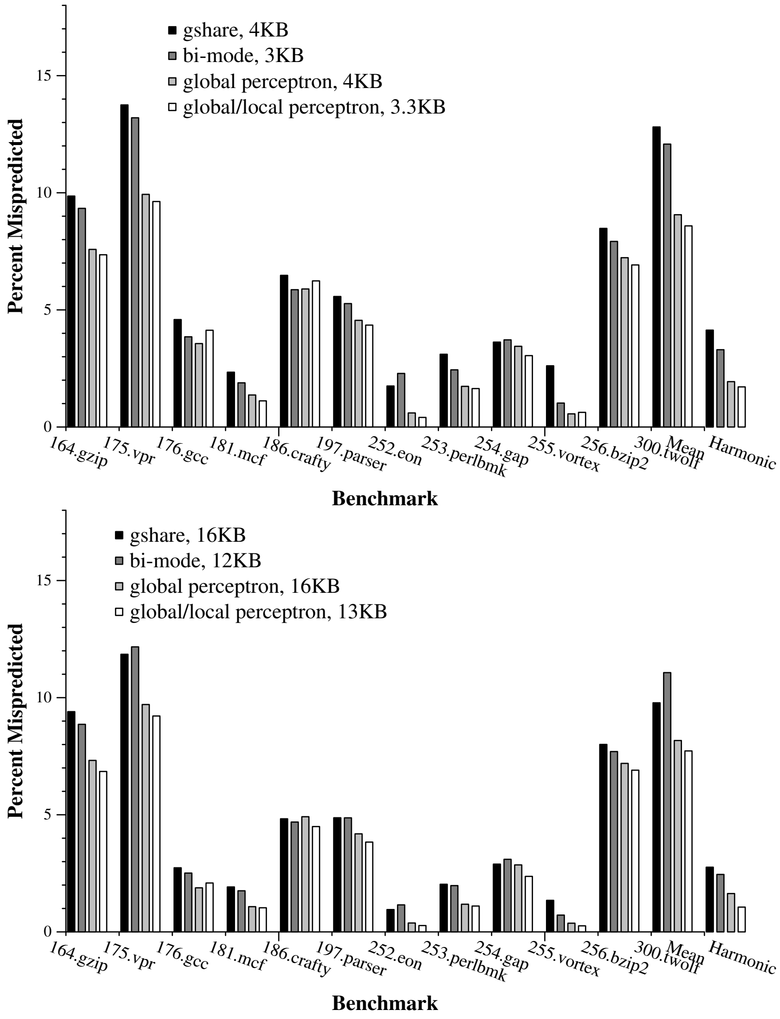

Figure 3: Misprediction Rates for Individual Benchmarks. These charts show the misprediction rates of global perceptron, *gshare* and bi-mode predictors at hardware budgets of 4 KB and 16 KB.

### 6.3.1 Large Hardware Budgets

As Moore's Law continues to provide more and more transistors in the same area, it makes sense to explore much larger hardware budgets for branch predictors. Evers' thesis [10] explores the design space for multi-component hybrid predictors using large hardware budgets, from 18 KB to 368 KB. To our knowledge, the multi-component predictors presented in Evers' thesis are the most accurate fully dynamic branch predictors known in previous work. This predictor uses a McFarling-style chooser to choose between two other McFarling-style hybrid predictors. The first hybrid component joins a *gshare* with a short history to a *gshare* with a long history. The other hybrid component consists of 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' multi-component 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. The tuning of this large perceptron predictor is not as exhaustive as for the smaller hardware budgets, due to the huge design space. 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 storing the table of local histories. The configuration of the perceptron predictor is given in Table 3.

| Size (KB) | Global History | Local History | Number of Perceptrons | Number of 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 |

Table 3: Configurations for Large Budget Perceptron Predictors.

Figure 4 shows the harmonic mean misprediction rates of Evers' multi-component predictor and the global/local perceptron predictor on the SPEC 2000 integer benchmarks. The perceptron predictor outperforms the multi-component predictor at every hardware budget, with the misprediction rates getting closer to one another as the hardware budget is increased. Both predictors are capable of reaching amazingly low misprediction rates at the 368 KB hardware budget, with the perceptron at 0.85% and the multi-component predictor at 0.93%.

We claim that these results are evidence that the perceptron predictor is the most accurate fully dynamic branch predictor known. We must point out that we have not exhaustively tuned either the multi-component or the perceptron predictors because of the huge computational challenge. Nevertheless, there is a clear separation between the misprediction rates of the multi-component and perceptron predictors, and between the perceptron and all other predictors we have examined at lower hardware budgets; thus, we are confident that our claim can be verified by independent researchers.

## 6.4 IPC

We have seen that the perceptron predictor is highly accurate but has a multi-cycle delay associated with it. If the delay is too large, overall performance may suffer as the processor stalls waiting for predictions. We

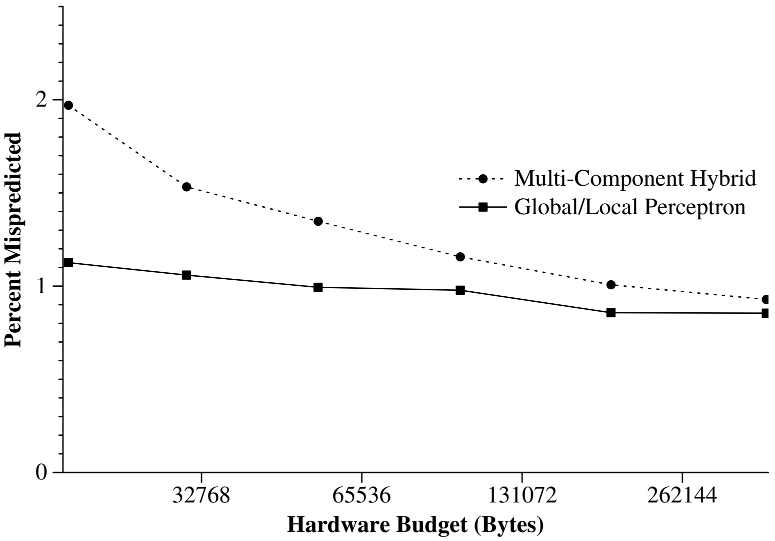

Figure 4: Hardware Budget vs. Misprediction Rate for Large Predictors.

now evaluate the perceptron predictor in terms of overall processor performance, measured in IPC, and taking into account predictor access delay. In particular, we compare an overriding perceptron predictor against the overriding hybrid predictor of the Alpha 21264, and we consider two processor configurations. One configuration uses a moderate clock rate that matches the latest Alpha processor, while the other approximates the more aggressive clock rate and deeper pipeline of the Intel Pentium 4.

This remainder of this section describes configurations of the overriding perceptron predictor for these two clock rates and reports on simulated IPC for the SPEC 2000 benchmark.

### 6.4.1 Moderate Clock Rate Simulations

Currently, the fastest Alpha processor in 180 nm technology is clocked at a rate of 833 MHz. At this clock rate, both the perceptron predictor and Alpha hybrid predictor deliver a prediction in two clock cycles.

Using SimpleScalar/Alpha, we simulate a two-level overriding predictors at 833 MHz. The first level is a 256-entry Smith predictor [27], i.e., a simple one-level table of two-bit saturating counters indexed by branch address. This predictor roughly simulates the line predictor of the overriding Alpha predictor. Our Smith predictor achieves a harmonic mean accuracy of 85.2%, which is the same accuracy quoted for the Alpha line predictor [18]. For the second level predictor, we simulate both the perceptron predictor and the Alpha hybrid predictor. The perceptron predictor consists of 133 perceptrons, each with 24 weights. Although the 25 weight perceptron predictor was the best choice at this hardware budget in our simulations, the 24 weight version has much the same accuracy but is 10% faster. We have observed that the ideal ratio of per-branch history bits to total history bits is roughly 20%, so we use 19 bits of global history and 4 bits of per-branch history from a table of 1024 histories. The total state required for this predictor is 3704 bytes, approximately the same as the Alpha hybrid predictor, which uses 3712 bytes. Both the Alpha hybrid predictor and the perceptron predictor incur a single-cycle penalty when they override the Smith predictor. We also simulate a 2048-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 [21], which uses a clock rate similar to that of the Alpha. We again simulate the 12 SPEC int 2000 benchmarks, this time allowing each benchmark to execute 2 billion instructions. We simulate the 7-cycle misprediction penalty of the Alpha 21264.

When a branch is encountered, there are four possibilities with the overriding predictor:

- 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 predictor agree and are both incorrect. In this case, there is no overriding, but the prediction is wrong, so a full misprediction penalty is incurred.

Figure 5 shows the instructions per cycle (IPC) for each of the predictors. Even though there is a penalty when the overriding Alpha and perceptron predictors override 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.

#### 6.4.2 Aggressive Clock Rate Simulations

The current trend in microarchitecture is to deeply pipeline 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. Nevertheless, we will show 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 7 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 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.

Figure 6 shows the result of simulating predictors in a microarchitecture with characteristics of the Pentium 4. The misprediction penalty is 20, which simulates the long pipeline of the Pentium 4. The Alpha overriding hybrid predictor is conservatively scaled to take 3 clock cycles, while the overriding perceptron predictor takes 4 clock cycles. The 2048-entry *gshare* predictor is unmodified. 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.

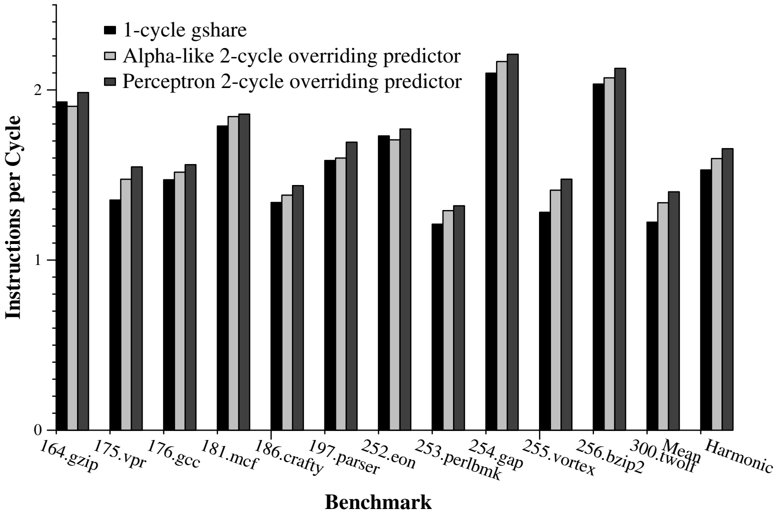

Figure 5: IPC for overriding perceptron and hybrid predictors. This chart shows the IPCs yielded by *gshare*, an Alpha-like hybrid, and global/local perceptron predictor given a 7-cycle misprediction penalty. The hybrid and perceptron predictors have a 2-cycle latency, and are used as overriding predictors with a small Smith predictor.

## 6.5 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 7 shows the average accuracy of each of the 100 predictions for each of the static branches with a 4KB hardware budget. The average is weighted by the relative frequencies of each branch.

The perceptron method learns more quickly the *gshare* or bi-mode. 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 [23]. As we will see in the next section, the perceptron prediction does not have this weakness, as it always does better with a longer history length.

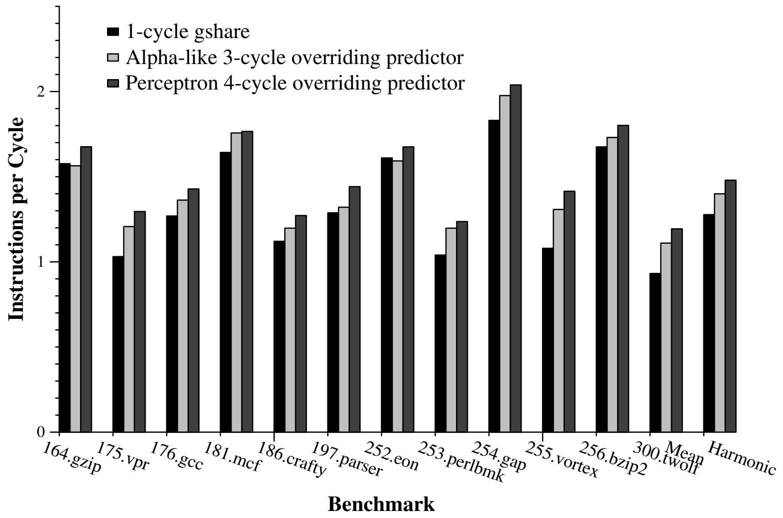

Figure 6: IPC for overriding perceptron and hybrid predictors with long pipelines. This chart shows the IPCs yields by *gshare*, a hybrid predictor and a global/local perceptron predictor with a large misprediction penalty and high clock rate.

## 6.6 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 like *gshare* that use the history register as an index into a table require space exponential in the history length, while 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 64K 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 1.86% compared with 1.96% for the perceptron predictor. The inferior performance of this crippled predictor has two likely causes: there is more destructive aliasing with perceptrons because they are larger, and thus fewer, than *gshare*'s two-bit counters, and perceptrons are capable of learning only linearly separable functions of their input, while *gshare* can potentially learn any Boolean function.

Figure 8 shows the result of simulating *gshare* and the perceptron predictor with varying history lengths on the SPEC 2000 benchmarks. Here, we use a 4M byte hardware budget is used to allow *gshare* to consider longer history lengths than usual. As we allow each predictor to consider longer histories, each becomes more accurate until *gshare* becomes worse and then runs out of bits (since *gshare* requires resources exponential in the number of history bits), while the perceptron predictor continues to improve. With this unrealistically huge hardware budget, *gshare* performs best with a history length of 23, where it achieves a misprediction rate of 1.55%. The perceptron predictor is best at a history length of 66, where it achieves a misprediction rate of 1.09%.

## 6.7 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.

To relate linear separability to branch prediction, we define the notion of *linearly separable branches.*

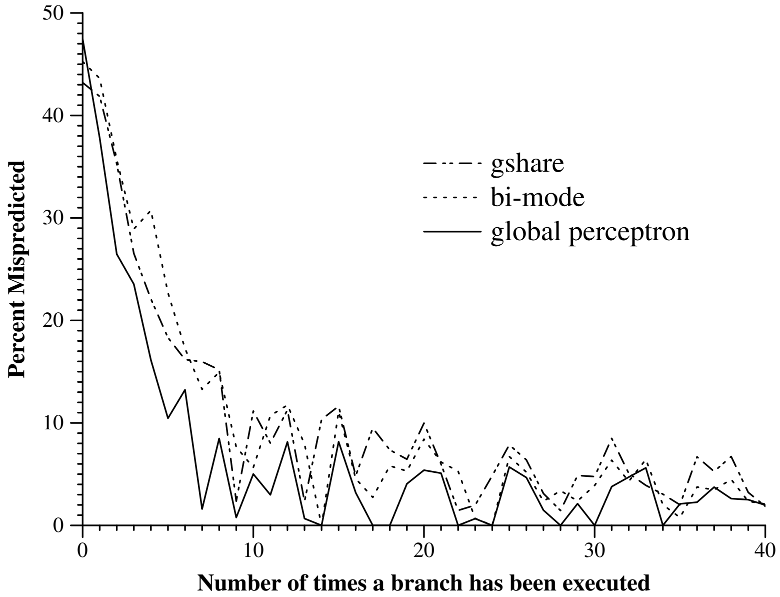

Figure 7: Average Training Times for SPEC 2000 benchmarks. The $x$ axis is the number of times a branch has been executed. The $y$-axis is the average, over all branches in the program, of 1 if the branch was mispredicted, 0 otherwise. Over time, this statistic tracks how quickly each predictor learns. The perceptron predictor achieves greater accuracy earlier than the other two methods.

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, while 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 9 shows the misprediction rates for each benchmark for a 4K budget, as well as the percentage of dynamically executed branches that is linearly inseparable. For each benchmark, the bar on the left shows the misprediction rate of *gshare*, while 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, while 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.

Some branches require longer histories than others for accurate prediction, and the perceptron predictor often has an advantage for these branches. Figure 10 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

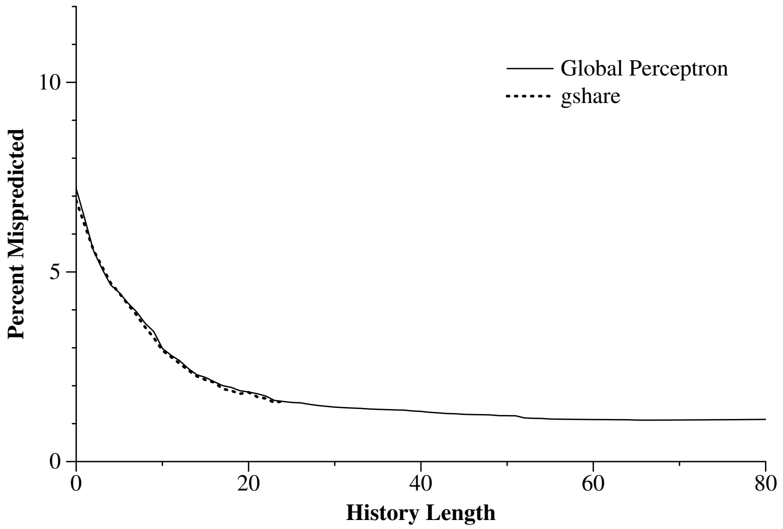

Figure 8: History Length vs. Performance. This graph shows how accuracy for *gshare* and the perceptron predictor improves as history length is increased. The perceptron predictor is able to consider much longer histories with the same hardware budget.

length, which is represented on the $x$ axis. Each data point 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.*, who show that most branches can be predicted by looking at three previous branches [9]. 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 performs generally better when the branches require long histories, while *gshare* sometimes performs better when branches require short histories.

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

We found this history length by finding the most distant of the three weights with the greatest magnitude in a perceptron trained for each branch, an application of the perceptron predictor for analyzing branch behavior.

## 6.8 Additional Advantages of the Perceptron Predictor

**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 [15]. 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 [14], but in our predictor this information comes for free. We have observed experimentally that the probability that a

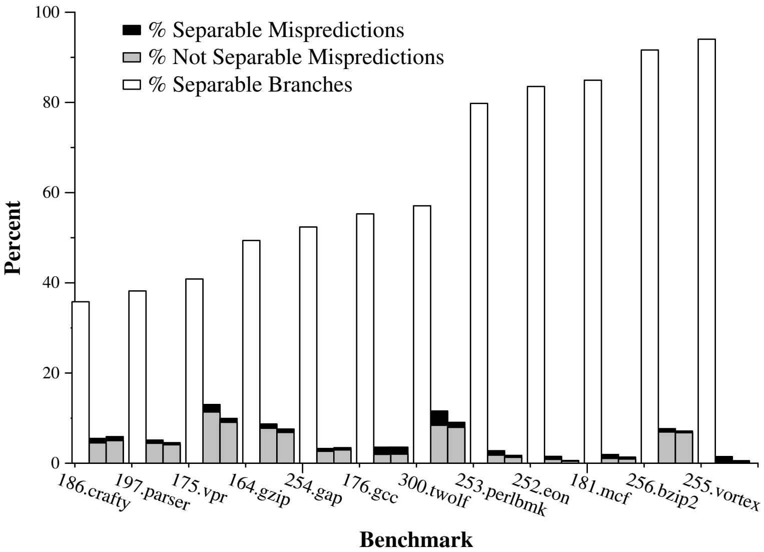

Figure 9: Linear Separability vs. Accuracy at a 4KB budget. For each benchmark, the leftmost bar shows the number of linearly separable dynamic branches in the benchmark, the middle bar shows the misprediction rate of *gshare* at a 4KB hardware budget, and the right bar shows the misprediction rate of the perceptron predictor at the same hardware budget.

branch will be taken can be accurately estimated as a linear function of the output of the perceptron predictor.

**Analyzing branch behavior with perceptrons.** Perceptrons can be used to analyze correlations among branches. The perceptron predictor assigns each bit in the branch history a weight. 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 6.7 could be used with a profiler to provide path length information to the variable length path predictor [29].

## 7 Conclusions

In this paper we have introduced a new branch predictor that uses neural learning techniques—the perceptron in particular—as the basic prediction mechanism. Perceptrons are attractive because they can use long history lengths without requiring exponential resources. 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 simpler Smith predictor to provide a quick prediction that may be later overridden. For the SPEC 2000 integer benchmarks, this overriding predictor results in 36% fewer mispredictions than 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.

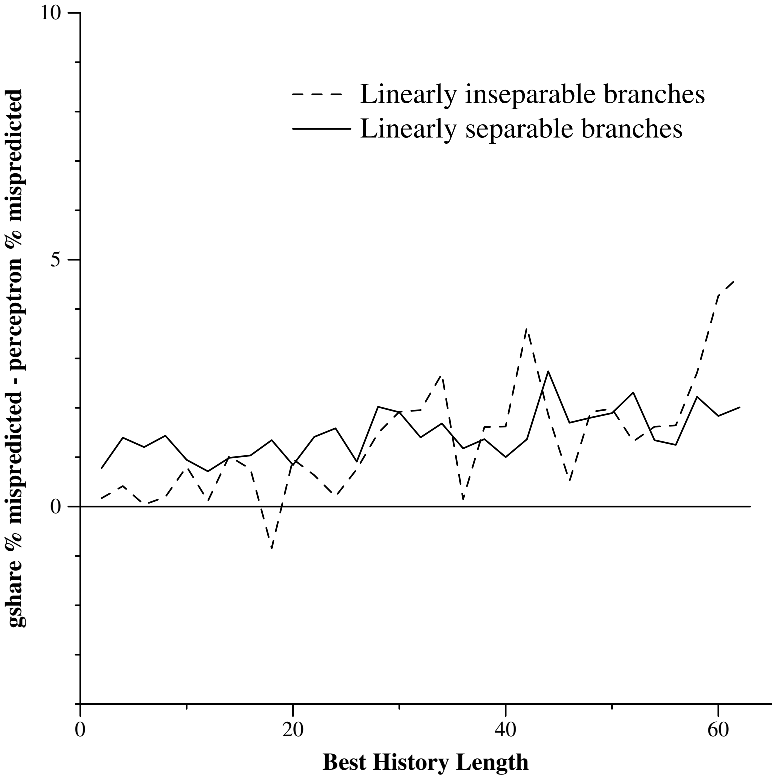

Figure 10: Classifying the Advantage of the Perceptron Predictor. Each data point is the average difference in misprediction rates of the perceptron predictor and gshare (on the $x$ axis) for length for those branches (on the $y$ axis). Above the $x$ axis, the perceptron predictor is better on average. 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.

We have shown that there is benefit to considering history lengths longer than those previously considered. Variable length path branch prediction considers history lengths of up to 23 [29], and a study of the effects of long branch histories on branch prediction only considers lengths up to 32 [9]. We have found that additional performance gains can be found for branch history lengths of up to 66.

We have also shown why the perceptron predictor is accurate. PHT techniques provide a general mechanism that does not scale well with history length. Our predictor instead performs particularly well on two classes of branches—those that are linearly separable and those that require long history lengths—that represent a large number of dynamic branches.

Perceptrons have interesting characteristics that open up new avenues for future work. As noted in Section 6.8, 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 Steve Keckler and Kathryn McKinley for many stimulating discussions on this topic, and we thank Steve, Kathryn, and Ibrahim Hur for their comments on earlier versions of this paper. This research was supported in part by DARPA Contract #F30602-97-1-0150 from the US Air Force Research Laboratory, NSF CAREERS grant ACI-9984660, and by ONR grant N00014-99-1-0402.

# References

[1] T. Ball and J. Larus. Branch prediction for free. In *Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 300–313, June 1993.

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

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

[4] B. Calder, D. Grunwald, M. Jones, D. Lindsay, J. Martin, M. Mozer, and B. Zorn. Evidence-based static branch prediction using machine learning. *ACM Transactions on Programming Languages and Systems*, 19(1), 1997.

[5] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. McGraw Hill, 1990.

[6] A.N. Eden and T.N. Mudge. The YAGS branch prediction scheme. In *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture*, November 1998.

[7] J. Emer and N. Gloy. A language for describing predictors and its application to automatic synthesis. In *Proceedings of the 24th International Symposium on Computer Architecture*, June 1997.

[8] M. Evers, P.-Y. Chang, and Y. N. Patt. Using hybrid branch predictors to improve branch prediction accuracy in the presence of context switches. In *Proceedings of the 23rd International Symposium on Computer Architecture*, May 1996.

[9] M. Evers, S. J. Patel, R. S. Chappell, and Y. N. Patt. An analysis of correlation and predictability: What makes two-level branch predictors work. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, July 1998.

[10] Marius Evers. *Improving Branch Prediction by Understanding Branch Behavior*. PhD thesis, University of Michigan, Department of Computer Science and Engineering, 2000.

[11] L. Faucett. *Fundamentals of Neural Networks: Architectures, Algorithms and Applications*. Prentice-Hall, Englewood Cliffs, NJ, 1994.

[12] Faustino Gomez, Doug Burger, and Risto Miikkulainen. A neuroevolution method for dynamic resource allocation on a chip multiprocessor. In *Proceedings of the International Joint Conference on Neural Networks (IJCNN-01)*, July 2001.

[13] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach, Second Edition*. Morgan Kaufmann Publishers, 1996.

[14] Erik Jacobsen, Eric Rotenberg, and James E. Smith. Assigning confidence to conditional branch predictions. In *Proceedings of the 29th Annual International Symposium on Microarchitecture*, December 1996.

[15] D. A. Jiménez and N. Walsh. Dynamically weighted ensemble neural networks for classification. In *Proceedings of the 1998 International Joint Conference on Neural Networks*, May 1998.

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

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

[18] Richard E. Kessler. The Alpha 21264 microprocessor. *IEEE Micro*, 19(2):24–36, March/April 1999.

[19] A. D. Kulkarni. *Artificial Neural Networks for Image Understanding*. Van Nostrand Reinhold, 1993.

[20] C.-C. Lee, C.C. Chen, and T.N. Mudge. The bi-mode branch predictor. In *Proceedings of the 30th Annual International Symposium on Microarchitecture*, November 1997.

[21] Gregg Lesartre and Doug Hunt. PA-8500: The continuing evolution of the PA-8000 family. In *42nd IEEE International Computer Conference*, February 1997.

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

[23] P. Michaud, A. Seznec, and R. Uhlig. Trading conflict and capacity aliasing in conditional branch predictors. In *Proceedings of the 24th International Symposium on Computer Architecture*, June 1997.

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

[25] S. Sechrest, C.-C. Lee, and T.N. Mudge. Correlation and aliasing in dynamic branch predictors. In *Proceedings of the 23rd International Symposium on Computer Architecture*, May 1999.

[26] R. Setiono and H. Liu. Understanding neural networks via rule extraction. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence*, pages 480–485, 1995.

[27] James E. Smith. A study of branch prediction strategies. In *Proceedings of the 8th Annual International Symposium on Computer Architecture*, May 1981.

[28] E. Sprangle, R.S. Chappell, M. Alsup, and Y. N. Patt. The Agree predictor: A mechanism for reducing negative branch history interference. In *Proceedings of the 24th International Symposium on Computer Architecture*, June 1997.

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

[30] K. Wang and M. Franklin. Highly accurate data value prediction using hybrid predictors. In *Proceedings of the 30th Annual International Symposium on Microarchitecture*, December 1997.

[31] B. Widrow and M.E. Hoff Jr. Adaptive switching circuits. In *IRE WESCON Convention Record, part 4*, pages 96–104, 1960.

[32] T.-Y. Yeh and Yale N. Patt. Two-level adaptive branch prediction. In *Proceedings of the $24^{th}$ ACM/IEEE Int'l Symposium on Microarchitecture*, November 1991.