Dynamic Branch Prediction with Perceptrons

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

Abstract

This paper presents a new method for branch prediction. The key idea is to use one of the simplest possible neural networks, the perceptron *as an alternative to the commonly used two-bit counters. Our predictor achieves increased accuracy by making use of long branch histories, which are possible because the hardware resources for our method scale linearly with the history length. By contrast, other purely dynamic schemes require exponential resources.*

We describe our design and evaluate it with respect to two well known predictors. We show that for a 4K byte hardware budget our method improves misprediction rates for the SPEC 2000 benchmarks by 10.1% over the gshare *predictor. Our experiments also provide a better understanding of the situations in which traditional predictors do and do not perform well. Finally, we describe techniques that allow our complex predictor to operate in one cycle.*

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 [10, 24]. 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 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. Recent efforts to improve branch prediction focus primarily on eliminating *aliasing* in two-level adaptive predictors [17, 16, 22, 4], 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. It's natural to ask whether we can improve accuracy by replacing these counters with neural networks, which provide good predictive capabilities. Since most neural networks would be prohibitively expensive to implement as branch predictors, we explore the use of perceptrons, one of the simplest possible neural networks. Perceptrons are easy to understand, simple to implement, and have several attractive properties that differentiate them from more complex neural networks.

We propose a two-level scheme that uses fast perceptrons instead of two-bit counters. Ideally, each static branch is allocated its own perceptron to predict its outcome. Traditional two-level adaptive schemes use a pattern history table (PHT) of two-bit saturating counters, indexed by a global history shift register that stores the outcomes of previous branches. This structure limits the length of the history register to the logarithm of the number of counters. Our scheme not only uses a more sophisticated prediction mechanism, but it can consider much longer histories than saturating counters.

This paper explains why and when our predictor performs well. We show that the neural network we have chosen 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.

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 predictor improves misprediction rates on the SPEC 2000 integer benchmarks by 10.1%.

- We explore the design space for two-level branch predictors based on perceptrons, empirically identifying good values for key parameters.

- We provide new insights into the behavior of branches, classifying them as either linearly separable or inseparable. We show that our predictor performs better on linearly separable branches, but worse on linearly inseparable branches. Thus, our predictor is complementary to existing predictors and works well as part of a hybrid predictor.

- We explain why perceptron-based predictors introduce interesting new ideas for future research.

2 Related Work

2.1 Neural networks

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 [8], and image understanding [15, 12].

Static branch prediction with neural networks. Neural networks have been used to perform *static* branch prediction [3], 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, 3]. Static branch prediction performs worse than existing dynamic techniques, but is useful for performing static compiler optimizations.

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 [5], 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 propose putting intelligence in the microarchitecture, so the branch predictor can learn and adapt on-line. In fact, their approach cannot describe our new predictor.

2.2 Dynamic Branch Prediction

Dynamic branch prediction has a rich history in the literature. Recent research focuses on refining the two-level scheme of Yeh and Patt [26]. 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 [20], and many of the recently proposed branch predictors seek to reduce the aliasing problem [17, 16, 22, 4] 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 [4].

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 [7]. 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 [18].

Variable length path branch prediction [23] 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.

3 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.

3.1 Why perceptrons?

Perceptrons are a natural choice for branch prediction because they can be efficiently implemented in hardware. Other forms of neural networks, such as those trained by backpropagation, and other forms of machine learning, such as decision trees, are less attractive because of excessive implementation costs. For this work, we also considered other simple neural architectures, such as ADALINE [25] and Hebb learning [8], but we found that these were less effective than perceptrons (lower hardware efficiency for ADALINE, less accuracy for Hebb).

One 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 [21], 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. We discuss this property in more detail in Section 5.7.

3.2 How Perceptrons Work

The perceptron was introduced in 1962 [19] 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.*

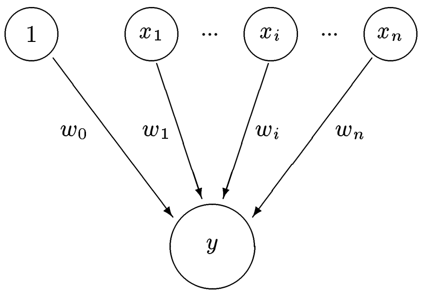

Figure 1: Perceptron Model. The input values $x_1, \ldots, x_n$, are propagated through the weighted connections by taking their respective products with the weights $w_1, \ldots, w_n$. These products are summed, along with the bias weight w_0, to produce the output value y.

3.3 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 θ be the *threshold*, a parameter to the training algorithm used to decide when enough training has been done.

```
if sign(y_out) ≠ t or |y_out| ≤ θ 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 i^{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.

3.4 Linear Separability

A limitation of perceptrons is that they are only capable of learning *linearly separable* functions [8]. 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* [8]. 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.

3.5 Putting it All Together

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.

Figure 2 shows a block diagram for the perceptron predictor. 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 6. 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^{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^{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 6 shows that a number of arithmetic and microarchitectural tricks enable a prediction in a single cycle, even for long history lengths.

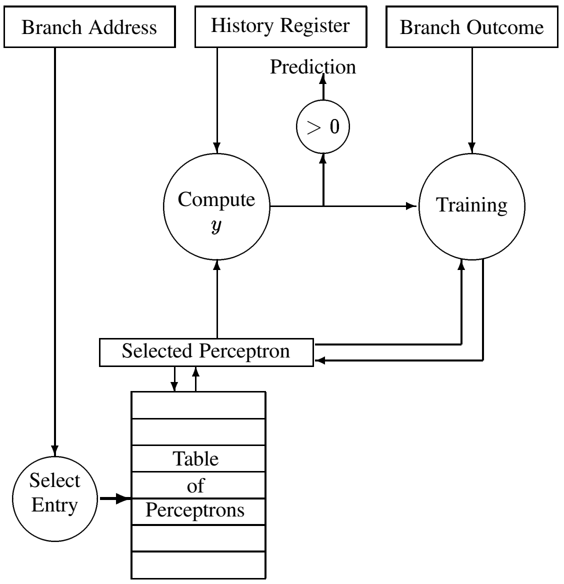

Figure 2: Perceptron Predictor Block Diagram. The branch address is hashed to select a perceptron that is read from the table. Together with the global history register, the output of the perceptron is computed, giving the prediction. The perceptron is updated with the training algorithm, then written back to the table.

4 Design Space

This section explores the design space for perceptron predictors. 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 [7] but also reduce the number of table entries, thereby increasing aliasing. In our experiments, the best history lengths ranged from 12 to 62, depending on the hardware budget.

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.

Threshold. The threshold is a parameter to the perceptron training algorithm that is used to decide whether the predictor needs more training. Because the training algorithm will only change a weight when the magnitude of y_{out} is less than the threshold θ, no weight can exceed the value of θ. Thus, the number of bits needed to represent a weight is one (for the sign bit) plus $\lfloor \log_2 \theta \rfloor$.

5 Experimental Results

We use simulations of the SPEC 2000 integer benchmarks to compare the perceptron predictor against two highly regarded techniques from the literature.

5.1 Methodology

Predictors simulated. We compare our new predictor against *gshare* [17] and bi-mode [16], two of the best purely dynamic global predictors from the branch prediction literature. We also evaluate a hybrid *gshare*/perceptron predictor that uses a 2K byte choice table and the same choice mechanism as that of the Alpha 21264 [14]. The goal of our hybrid predictor is to show that because the perceptron has complementary strengths to *gshare*, a hybrid of the two performs well.

All of the simulated predictors use only global pattern information, i.e., neither per-branch nor path information is used. Thus, we have not yet compared our hybrid against existing global/per-branch hybrid schemes. Per-branch and path information can yield greater accuracy [6, 14], but our restriction to global information is typical of recent work in branch prediction [16, 4].

Gathering traces. Our simulations use the instrumented assembly output of the gcc 2.95.1 compiler with optimization flags `-O3 -fomit-frame-pointer` running on an AMD K6-III under Linux. Each conditional branch instruction is instrumented to make a call to a trace-generating procedure. Branches in libraries or system calls are not profiled. The traces, consisting of branch addresses and outcomes, are fed to a program that simulates the different branch prediction techniques.

Benchmarks simulated. We use the 12 SPEC 2000 integer benchmarks. All benchmarks are simulated using the SPEC `test` inputs. For `253.perlbmk`, the `test` run executes `perl` on many small inputs, so the concatenation of the resulting traces is used. We feed up to 100 million branch traces from each benchmark to our simulation program; this is roughly equivalent to simulating half a billion instructions.

Tuning the predictors. We use a composite trace of the first 10 million branches of each SPEC 2000 benchmark to tune the parameters of each predictor for a variety of hardware budgets. For *gshare* and bi-mode, we tune the history lengths by exhaustively trying every possible history length for each hardware budget, keeping the value that gives the best prediction accuracy. For the 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. For each hardware budget, we tune the history length by exhaustive search.

Table 1 shows the results of the history length tuning. We find an interesting relationship between history length and threshold: the best threshold θ 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. Since the number of bits needed to represent a perceptron weight is one (for the sign bit) plus $\lfloor \log_2 \theta \rfloor$, the number of bits per weight range from 7 (for a history length of 12) to 9 (for a history length of 62).

Our hybrid *gshare*/perceptron predictor consists of *gshare* and perceptron predictor components, along with a mechanism, similar to the one in the Alpha 21264 [14], that dynamically chooses between the two using a 2K byte table of two-bit saturating counters. Our graphs reflect this added hardware expense. For each hardware budget, we tune the hybrid predictor by examining every combination of table sizes for the *gshare* and perceptron components and choosing the combination yielding the best performance. In almost every case, the best configuration has resources distributed equally among the two prediction components.

Estimating area costs. Our hardware budgets do not include the cost of the logic required to do the computation. By examining die photos, we estimate that at the longest history lengths, this cost is approximately the same as that of 1K of SRAM. Using the parameters tuned for the 4K hardware budget, we estimate that the extra hardware will consume about the same logic as 256 bytes of SRAM. Thus, the cost for the computation hardware is small compared to the size of the table.

Hardware budget in kilobytes	History Length		
	gshare	bi-mode	perceptron
1	6	7	12
2	8	9	22
4	8	11	28
8	11	13	34
16	14	14	36
32	15	15	59
64	15	16	59
128	16	17	62
256	17	17	62
512	18	19	62

Table 1: Best History Lengths. This table shows the best amount of global history to keep for each of the branch prediction schemes.

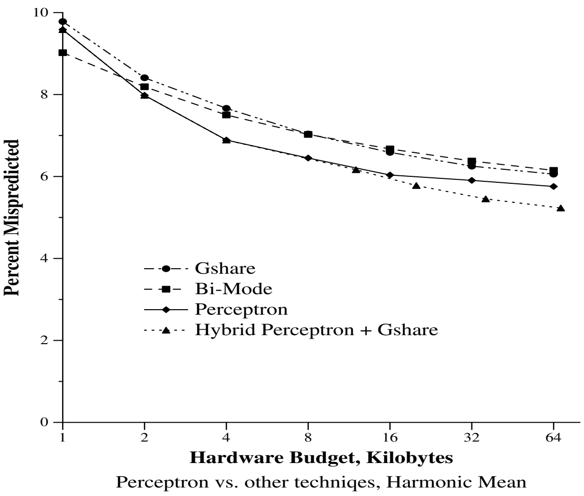

Figure 3: Hardware Budget vs. Prediction Rate on SPEC 2000. The perceptron predictor is more accurate than the two PHT methods at all hardware budgets over one kilobyte.

5.2 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 [7]. 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 1, the perceptron predictor works best with much longer histories than the other two predictors. For example, with a 64K byte hardware budget, *gshare* works best with a history length of 15, even though the maximum possible length for *gshare* at 64K is 18. At the same hardware budget, the perceptron predictor works best with a history length of 62.

5.3 Performance

Figure 3 shows the harmonic mean of prediction rates achieved with increasing hardware budgets on the SPEC 2000 benchmarks. The perceptron predictor's advantage over the PHT methods is largest at a 4K byte hardware budget, where the perceptron predictor has a misprediction rate of 6.89%, an improvement of 10.1% over *gshare* and 8.2% over bimode. For comparison, the bi-mode predictor improves only 2.1% over *gshare* at the 4K budget. Interestingly, the SPEC 2000 integer benchmarks are, as a whole, easier for branch predictors than the SPEC95 benchmarks, explaining the smaller separation between *gshare* and bi-mode than observed previously [16].

Figures 4 and 5 show the misprediction rates on the SPEC 2000 benchmarks for hardware budgets of 4K and 16K bytes, respectively. The hybrid predictor has no advantage at the 4K budget, since three tables must be squeezed into a small space. At the 16K budget, the hybrid predictor has a slight advantage over the perceptron predictor by itself.

5.4 Training Times

To compare the training speeds of the three methods, we examine the first 40 times each branch in the `176.gcc` benchmark is executed (for those branches executing at least 40 times). Figure 6 shows the average accuracy of each of the

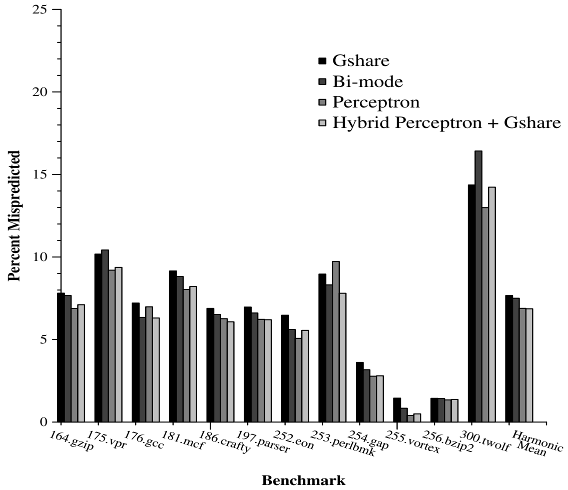

Figure 4: Misprediction Rates at a 4K budget. The perceptron predictor has a lower misprediction rate than *gshare* for all benchmarks except for `186.crafty` and `197.parser`.

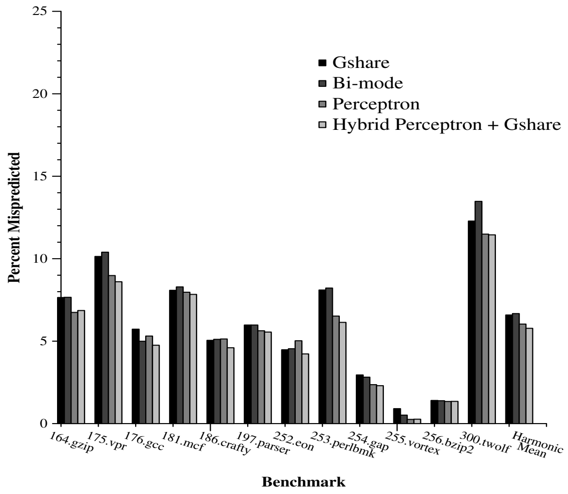

Figure 5: Misprediction Rates at a 16K budget. Gshare outperforms the perceptron predictor only on `186.crafty`. The hybrid predictor is consistently better than the PHT schemes.

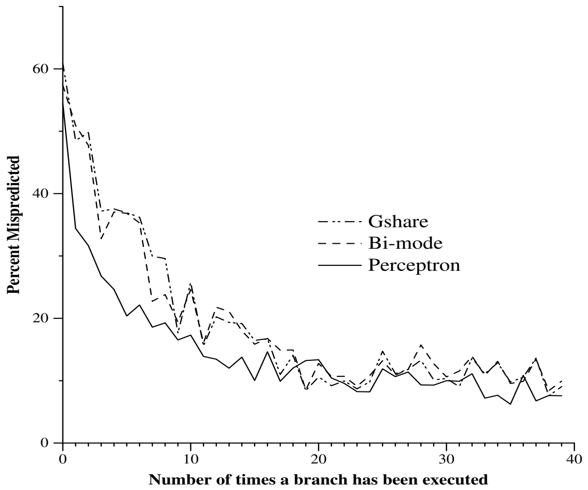

Figure 6: Average Training Times for gcc. The x axis is the number of times a branch has been executed. The y-axis is the average, over all branches in the program, of 1 if the branch was mispredicted, 0 otherwise. Over time, this statistic tracks how quickly each predictor learns. The perceptron predictor achieves greater accuracy earlier than the other two methods.

40 predictions for each of the static branches. The average is weighted by the relative frequencies of each branch. We choose `176.gcc` because it has the most static branches of all the SPEC benchmarks.

The perceptron method learns more quickly the other two. 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 [18]. 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.

5.5 Why Does it Do Well?

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 512K 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 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 7 shows the result of simulating *gshare* and the perceptron predictor with varying history lengths on the SPEC 2000 benchmarks. Here, an 8M 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 18, where it achieves a misprediction rate of 5.20%. The perceptron predictor is best at a history length of 62, the longest history considered, where it achieves a misprediction rate of 4.64%.

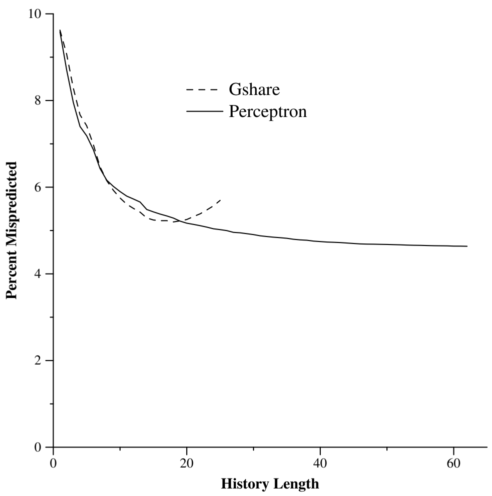

Figure 7: History Length vs. Performance. The accuracy of the perceptron predictor improves with history length, while *gshare*'s accuracy bottoms out at 18.

5.6 When Does It Do Well?

The perceptron predictor does well when the branch being predicted exhibits *linearly separable behavior.* To define this term, 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 not linearly separable, then *gshare* may predict B better than the perceptron predictor, and we say that such branches are *linearly inseparable*. We compute $f_B(h_{10})$ for each static branch B in the first 100 million branches of each benchmark and test for linear separability of the function. (Our algorithm for this test takes time superexponential in n, so we are unable to go beyond 10 bits of history or 100 million dynamic branches. We believe these numbers are good estimates for the purpose of this discussion.)

Figure 8 shows the misprediction rates for each benchmark for a 512K budget, as well as the percentage of dynamically executed branches that is linearly inseparable. We choose a large hardware budget to minimize the effects of aliasing and to isolate the effects of linear separability. We see that the perceptron predictor performs better than *gshare* for the benchmarks to the left, which have more linearly separable branches than inseparable branches. Conversely, for all but one of the benchmarks for which there are more linearly inseparable branches, *gshare* performs better. Note that although the perceptron predictor performs best on linearly separable branches, it still has good performance overall.

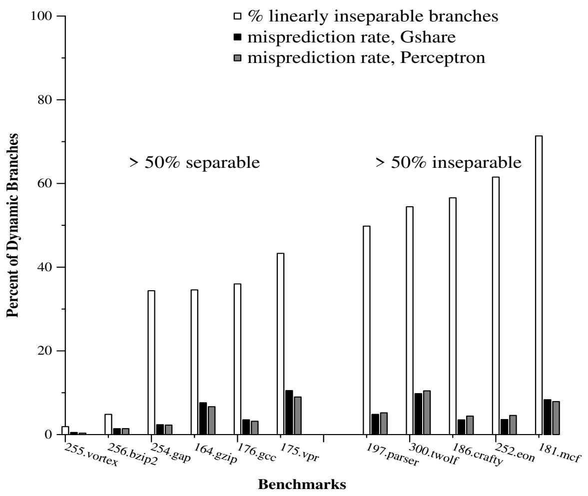

Figure 8: Linear Separability vs. Performance at a 512K budget. The perceptron predictor is better than *gshare* when the dynamic branches are mostly linearly separable, and it tends to be less accurate than *gshare* otherwise.

Some branches require longer histories than others for accurate prediction, and the perceptron predictor often has an advantage for these branches. Figure 9 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 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 [7]. 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.

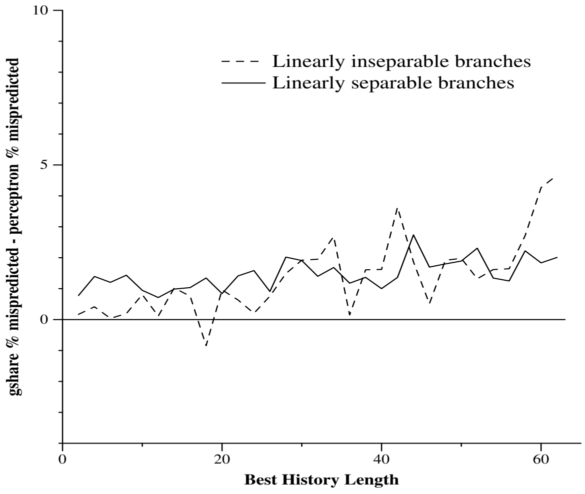

Figure 9: Classifying the Advantage of our Predictor. Above the x axis, the perceptron predictor is better on average. Below the x axis, *gshare* is better on average. For linearly separable branches, our predictor is on average more accurate than *gshare*. For inseparable branches, our predictor is sometimes less accurate for branches that require short histories, and it is more accurate on average for branches that require long histories.

5.7 Additional Advantages of Our 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 [12]. 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 [11], 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.

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, our methodology in Section 5.6 could be used with a profiler to provide path length information to the variable length path predictor [23].

5.8 Effects of Context Switching

Branch predictors can suffer a loss in performance after a context switch, having to warm up while relearning patterns [6]. We simulate the effects of context switching by interleaving branch traces from each of the SPEC 2000 integer benchmarks, switching to the next program after 60,000 branches. This workload represents an unrealistically heavy amount of context switching, but it serves as a good indicator of performance in extreme conditions, and it uses the same methodology as other recent work [4]. Note that previous studies have used the 8 SPEC 95 integer benchmarks, so our use of the 12 SPEC 2000 benchmarks will likely lead to higher misprediction rates. For each predictor, we consider the effect of re-initializing the table of counters after each context switch (which would be done with a privileged instruction in a real operating system) and use this technique when it gives better performance.

Figure 10 shows that context switching affects the perceptron predictor more significantly than the other two predictors. Nevertheless, the perceptron predictor still maintains an advantage over the other two predictors at hardware budgets of 4K bytes or more. The hybrid *gshare*/perceptron predictor performs better in the presence of context switching; this benefit of hybrid predictors has been noticed by others [6].

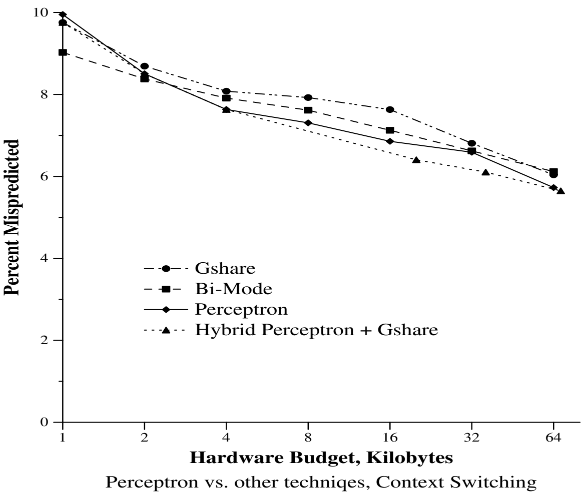

Figure 10: Budget vs. Misprediction Rate for Simulated Context Switching. The perceptron predictor is more affected by heavy context switching than *gshare* or bi-mode.

6 Implementation

We now suggest ways to implement our predictor efficiently.

Computing the Perceptron Output. 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. 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.

Training. The training algorithm of Section 3.3 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 at most 9-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
```

Delay. A 54×54 multiplier in a 0.25μm process can operate in 2.7 nanoseconds [9], which is approximately two clock cycles with a 700 MHz clock. At the longer history lengths, an implementation of our predictor resembles a 54×54 multiply, but the data corresponding to the partial products (i.e., the weights) are narrower, at most 9 bits. Thus, any carry-propagate adders, of which there must be at least one in a multiplier circuit, will not need to be as deep. We believe that a good implementation of our predictor at a large hardware budget will take no more than two clock cycles to make a prediction. For smaller hardware budgets, one cycle operation is feasible. Two cycles is also the amount of time claimed for the variable length path branch predictor [23]. That work proposes pipelining the predictor to reduce delay.

Jiménez *et al* study a number of techniques for reducing the impact of delay on branch predictors [13]. 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.

7 Conclusions

In this paper we have introduced a new branch predictor that uses neural networks—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. Another weakness of perceptrons is their inability to learn linearly inseparable functions, but despite this weakness the perceptron predictor performs well, achieving a lower misprediction rate, at all hardware budgets, than two well-known global predictors on the SPEC 2000 integer benchmarks.

We have shown that there is benefit to considering history lengths longer than those previously considered. Variable length path prediction considers history lengths of up to 23 [23], and a study of the effects of long branch histories on branch prediction only considers lengths up to 32 [7]. We have found that additional performance gains can be found for branch history lengths of up to 62.

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.

Because our approach is largely orthogonal to many of the recent ideas in branch prediction, there is considerable room for future work. We can decrease aliasing by tuning our predictor to use the bias bits that were introduced by the Agree predictor [22]. We can also employ perceptrons in a hybrid predictor that uses both global and local histories, since hybrid predictors have proven to work better than purely global schemes [6]. We have preliminary experimental evidence that such hybrid schemes can be improved by using perceptrons, and we intend to continue this study in more detail.

More significantly, perceptrons have interesting characteristics that open up new avenues for future work. Because the perceptron predictor has different strengths and weaknesses from counter-based predictors, new hybrid schemes can be developed. We also plan to develop compiler-based branch classification techniques to make such hybrid predictors even more effective. We already have a starting point for this work, which is to focus on the distinction between linearly separable and inseparable branches, and between branches that require short history lengths and long history lengths. As noted in Section 5.7, 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 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 drafts 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] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

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

[9] Y. Hagihara, S. Inui, A. Yoshikawa, S. Nakazato, S. Iriki, R. Ikeda, Y. Shibue, T. Inaba, M. Kagamihara, and M. Yamashina. A 2.7ns 0.25um CMOS 54 $\times$ 54b multiplier. In *Proceedings of the IEEE International Solid-State Circuits Conference*, February 1998.

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

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

[12] 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.

[13] 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*, December 2000.

[14] R.E. Kessler, E.J. McLellan, and D.A. Webb. The Alpha 21264 microprocessor architecture. Technical report, Compaq Computer Corporation, 1998.

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

[16] 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.

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

[18] 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.

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

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

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

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