Dynamic Branch Prediction with Perceptrons [*]

Daniel A. Jiménez Calvin Lin

Department of Computer Sciences
The University of Texas
Austin, TX 78712 USA

June 2, 2000

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*, which provides better predictive capabilities than commonly used two-bit counters, and which allows our predictor to consider longer branch histories. The hardware resources needed for our method scale linearly with the history length, in contrast with other purely dynamic schemes that require exponential memory.

This paper describes our design and evaluates it with respect to two well known predictors. We show that for a 4K byte hardware budget our method improves misprediction rates on a composite trace of SPEC2000 benchmarks by 14.7% over the gshare predictor. Our experiments provide a better understanding of the situations in which traditional predictors do and do not perform well. We show that because our predictor works well for a particular class of branches, it works well with traditional schemes as a component of a hybrid predictor. 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 [12, 25]. 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. This paper proposes 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. Two-level adaptive predictors yield good performance and are commonly used [27, 15]. Recent efforts to improve branch prediction focus primarily on eliminating *aliasing*, 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 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. Empirical results show significant improvements for our approach. Our predictor outperforms two high quality predictors on a composite of the SPEC2000 bencmarks, but the performance advantage is not uniform across benchmarks. For example, Figure 1 shows that on the SPEC95 benchmark `126.gcc` our predictor improves the misprediction rate by 31% over gshare when using

[*]This research was supported in part by DARPA Contract #F30602-97-1-0150 from the US Air Force Research Laboratory and by NSF CAREERS grant ACI-9984660.

a hardware budget of 256K bytes. At the other extreme, our predictor does not perform well on the `099.go` benchmark, degrading the misprediction rate by 25% at a 256K hardware budget.

This paper explains why and when our predictor performs well. The neural network we have chosen works well for the class of *linearly separable branches*, a term we introduce. We show that programs tend to have many linearly separable branches, but when they do not, our predictor may not perform as well as other techniques. Thus, our predictor works best as a component of a hybrid prediction scheme, along with a more traditional predictor. For example, on the two extreme cases mentioned above, an untuned hybrid gshare/perceptron predictor with a 256K budget achieves a misprediction rate that is 40% better than gshare's for `126.gcc`, and 16% better than gshare for `099.go`.

This paper makes the following contributions. (1) We introduce the perceptron predictor, a new kind of branch predictor that is often more accurate than existing techniques. (2) We explore the design space for two-level branch predictors based on perceptrons, empirically identifying good values for key parameters. (3) We carefully evaluate our method against other dynamic global branch predictors. (4) We provide insights as to why our new predictor performs better. (5) We describe a novel pipelined approach that allows our technique to make a prediction in less than one cycle. (6) Finally, we explain why perceptron-based predictors introduce interesting new ideas for future research.

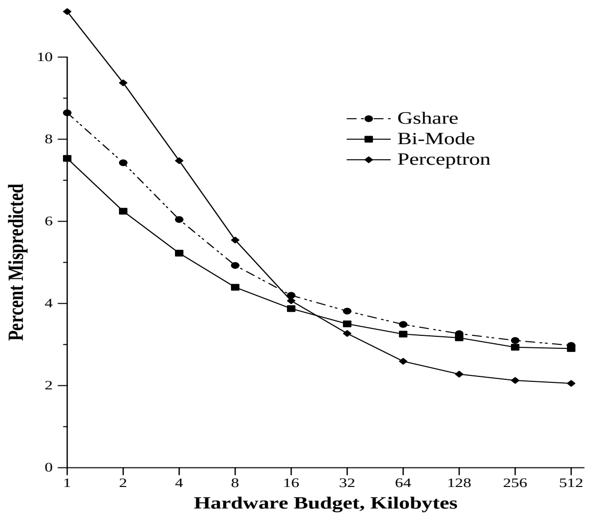

Figure 1: Hardware Budget vs. Prediction Rate for `126.gcc`. The perceptron predictor is more accurate than gshare and bi-mode for hardware budgets over 16K.

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 [10], image processing, and image understanding [16, 14].

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 extracting program features such as control-flow and opcode information and supplying these features as input to a trained neural network. This approach achieves an 80% correct prediction rate, compared to 75% for static heuristics [2, 4]. 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 [7], but it is important to note the difference between their work and ours. Their work used 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 [27]. 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 decremented if the branch is not taken, or incremented otherwise. An important problem in two-level predictors is aliasing [21], and many of the recently proposed branch predictors seek to reduce the aliasing problem [18, 17, 23, 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 may not help because longer history lengths require longer training times for these methods [19].

Variable length path branch prediction [24] 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. Using a complex multi-pass profiling and compiler-feedback mechanism, this technique achieves a misprediction rate of approximately 2.9% on the SPEC95 `126.gcc` benchmark when the hardware budget is 256K bytes. Our predictor achieves superior performance *without compiler assistance or profiling*. For the same hardware budget, our predictor achieves a misprediction rate of 2.1%, and our hybrid gshare/perceptron improves this to 1.8%.

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 back-propagation, and other forms of machine learning, such as decision trees, are less attractive because of excessive implementation costs. We also considered other simple neural architectures, such as ADALINE [26] and Hebb learning [10], but 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 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 [22], 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 5.6.

3.2 How Perceptrons Work

The perceptron was introduced in 1962 [20] as a way to study brain function. We consider the simplest of many types of perceptrons [3], 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 2 shows a graphical model of a perceptron. A perceptron is represented by a vector whose elements are the weights. For our purposes, the weights are signed integers. The output is the dot product of the weights vector, $w_{0..n}$, and the input vector, $x_{1..n}$ (x_0 is always set to 1, providing a "bias" input). The output y of a perceptron is computed as

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

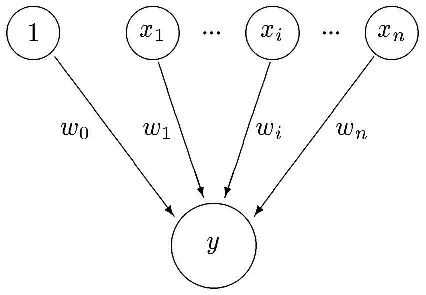

Figure 2: Perceptron Model. The input values $x_1, ..., x_n$, are propagated through the weighted connections by taking their respective products with the weights $w_1, ..., w_n$. These products are summed, along with the bias weight w_0, to produce the output value y.

The inputs to our perceptrons are *bipolar*, i.e., each x_i is either -1, meaning *untaken* or 1, meaning *taken*. A negative output is interpreted as *predict untaken*. A non-negative output is interpreted as *predict taken*.

3.3 Training Perceptrons

Once 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.

$$y_{out} = \begin{cases} 1 & \text{if } y > \theta \\ 0 & \text{if } -\theta \le y \le \theta \\ -1 & \text{if } y < -\theta \end{cases}$$

```
if $y_{out} \neq t$ 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 [10]. 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* [10]. 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[1]. As we will show later, many of the functions describing the behavior of branches in real programs are linearly separable. 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 3 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 appears 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 allow this prediction step to be squeezed into one clock cycle, even for long history lengths, with good accuracy from the resulting predictor.

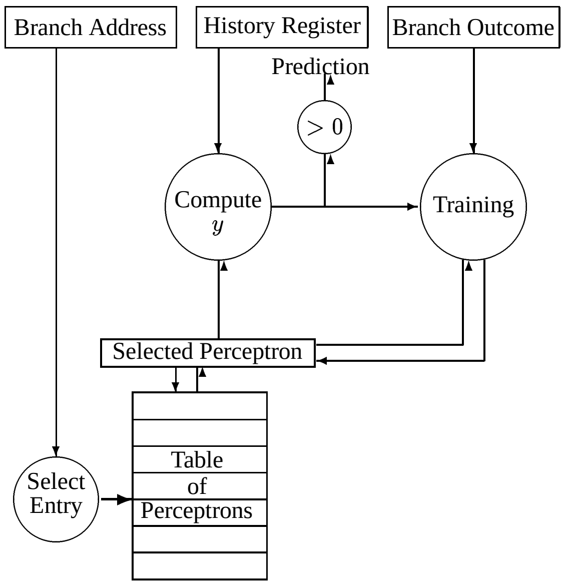

Figure 3: Perceptron Predictor Block Diagram. The branch address is hashed to select a perceptron that is read from the table. Together with the global history register, the output of the perceptron is computed, giving the prediction. The perceptron is updated with the training algorithm, then written back to the table.

[1]This is strictly true only when the learning is done statically, i.e., predictions are made only after the learning is finished. In our case, learning is dynamic, so a perceptron may learn to adapt to some non-linearity.

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 [9] 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. Varying the number of bits allowed us to trade hardware budget for accuracy. We found that, depending on the hardware budget and history lengths, using 7 to 9 bits give the best results.

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

5 Experimental Results

We simulated the SPEC2000 integer benchmarks to compare the perceptron predictor against two highly regarded techniques from the literature.

5.1 Methodology

Predictors simulated. We chose to compare our new predictor against gshare [18] and bi-mode [17], two of the best purely dynamic global predictors from the branch prediction literature. We also simulated an untuned hybrid gshare/perceptron predictor that uses a 2K byte choice table and the same choice mechanism as that of the Compaq 21264 [15]. The simulated predictors use only global pattern information, i.e., neither per-branch nor path information was used. Additional information can yield greater accuracy [8, 15], but our restriction to global information is typical of recent work in branch prediction [17, 6], and our new technique is largely orthogonal to these other techniques.

Gathering traces. Our simulations used 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 to generate traces for all conditional branch instructions. For each branch, the instrumented program makes a call to a profiling procedure giving the branch address and outcome. 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 simulated the 12 SPEC2000 integer benchmarks, as well as two SPEC95 benchmarks, `126.gcc` and `099.go`, that have been widely used in previous work. All benchmarks were simulated to completion using the SPEC `test` inputs. For `253.perlbmk`, the `test` run executes `perl` on many small inputs, so the concatenation of the resulting traces was used. For `099.go`, a smaller board size of 20×20 was used so that the program would produce a manageable number of traces. For Figure 1, the graph presented earlier, we computed the average misprediction rates for all 26 of the `ref` inputs for `126.gcc`; we believe this best represents the performance of all the predictors for the purpose of comparing our results to those in other papers where it is unclear which inputs are used.

We also generated a composite trace of the first 100 million branch traces from each of the SPEC2000 integer benchmarks to measure the overall performance of the predictors. When a benchmark executed fewer than 100 million branches, traces were copied from the beginning until there were enough. Since the median number of branches generated by the benchmarks is approximately 100 million, we believe this trace best represents the average workload for a high performance computer.

Tuning the predictors. We used the composite trace to tune the parameters of each predictor for a variety of hardware budgets. For gshare and bi-mode, we tuned the history lengths by exhaustively trying every value from 1 to the the maximum possible history length for each hardware budget, keeping the value that gave the best prediction accuracy. For the perceptron predictor, we found for each history length and number of bits per weight, the best value of the threshold by using an intelligent search of the space of values, pruning areas of the space that gave poor performance. We then tuned the history length and number of bits per weight for each hardware budget by exhaustive search. Table 1 shows the results of the history length tuning.

Our hybrid gshare/perceptron predictor was not tuned.[2] We simply combined two predictors of equal size using the parameters for the individually tuned predictors, and we added a mechanism, similar to the one in the Compaq 21264 [15], that dynamically chooses between the two using a 2K byte table of two-bit saturating counters. Our graphs reflect this added hardware expense. We believe that this lack of tuning has greatest impact at low hardware budgets.

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

[2]The final paper will provide results for a tuned hybrid gshare/perceptron predictor.

lengths, this cost is approximately the same as that of 1K of SRAM. Using the parameters tuned for the 4K hardware budget, we estimate 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.

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 [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 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.

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.

5.3 Performance

Figure 4 shows the prediction rates achieved with increasing hardware budgets on our composite trace. The perceptron predictor's advantage over the PHT methods is largest at the smaller hardware budgets. At a budget of 4K bytes, the perceptron predictor has a misprediction rate of 5.77%, an improvement of 14.7% over gshare and 10.0% over bimode. At a large budget of 256K, the perceptron predictor has a misprediction rate of 4.74%, an improvement of 4.7% over gshare and 5.3% over bimode.

On the `126.gcc` benchmark, the perceptron predictor performs particularly well at large hardware budgets. Using a 256 kilobyte hardware budget, the perceptron predictor achieves a

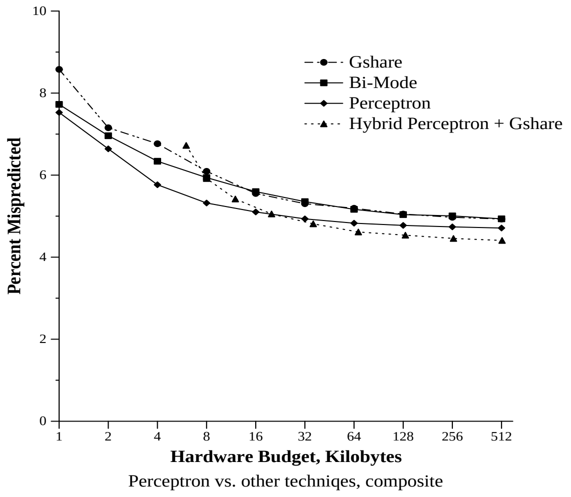

Figure 4: Hardware Budget vs. Prediction Rate on the Composite Trace. The perceptron predictor is more accurate than the two PHT methods at all hardware budgets. The untuned hybrid gshare/perceptron predictor is superior at hardware budgets greater than 32K bytes.

misprediction rate of 2.12%, an improvement of 31.4% from gshare and 39.9% from bimode. Figures 5 and 6 show the misprediction rates on all of the SPEC2000 benchmarks, as well as SPEC95 `126.gcc` and `099.go`, for the three predictors and the gshare/perceptron hybrid predictor for hardware budgets of 4K and 16K bytes.

5.4 Why Does it Do Well?

The main advantage of the perceptron predictor is its ability to consider longer history lengths. We support this observation with an experiment. We simulated 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 the perceptron predictor is capable of learning only linearly separable functions of its 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 composite SPEC2000 trace. An 8M byte hardware budget was used to allow gshare to consider longer history lengths than usual. Each predictor becomes more accurate as it is allowed to consider long histories, until gshare becomes worse and then runs out of bits (since only logarithmically many history bits can be considered), while the perceptron predictor continues to im-

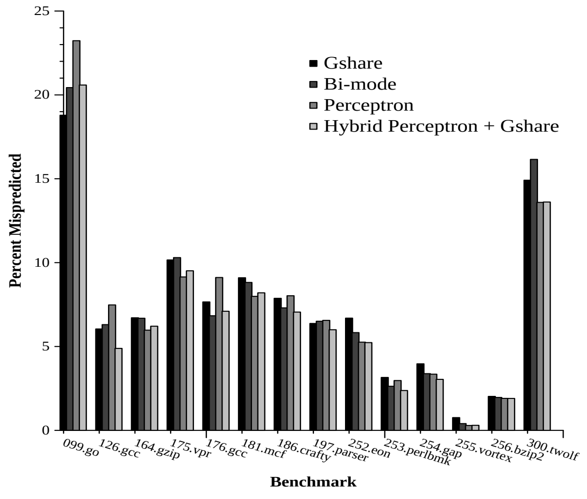

Figure 5: Misprediction Rates at a 4K budget. The perceptron predictor has a lower misprediction rate than gshare for all benchmarks except for `099.go`, `176.gcc`, `186.crafty` and `197.parser`. The hybrid predictor is consistently better than the PHT schemes.

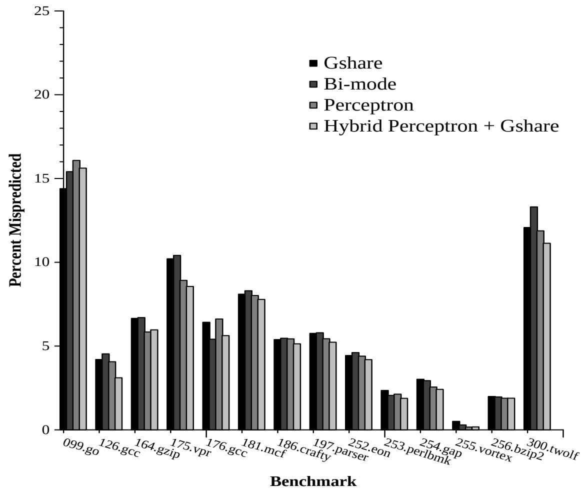

Figure 6: Misprediction Rates at a 16K budget. Gshare outperforms the perceptron predictor only on `099.go`, `176.gcc` and `186.crafty`.

prove. The best performance from gshare is 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%. Thus, the primary benefit of the perceptron predictor appears to be its ability to handle longer branch histories.

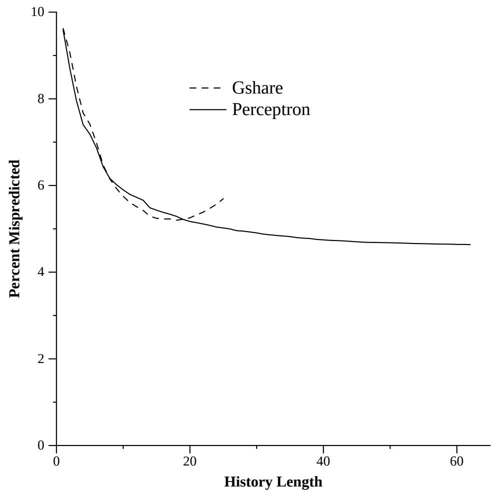

Figure 7: History Length vs. Performance. The accuracy of the perceptron predictor improves with history length, while gshare's accuracy bottoms out at 18.

5.5 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. If f_B is not linearly separable, then gshare may predict B better than the perceptron predictor. We say such branches are *linearly inseparable*. It is this function, f_B, that all branch predictors strive to learn. We computed $f_B(h_{10})$ for each static branch B in the first 100 million branches of each benchmark and tested for linear separability of the function. (Our algorithm for this test takes time superexponential in n, so we were 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.)

Intuitively, a linearly inseparable branch is one best predicted by a complex function of its history. For instance, if a branch is taken when the exclusive-OR of the third and fifth most recent branches is *true*, it is linearly inseparable, since there is no line separating *true* instances of inputs to the exclusive-OR function from *false* ones on the plane.

Figure 8 shows the misprediction rates on each benchmark for a 512K budget, as well as the percentage of dynamically executed branches that were linearly inseparable. We chose 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 also has good performance overall.

The `099.go` benchmark is particularly hard to predict for both gshare and the perceptron predictor. Figure 8 shows that 82.82% of the dynamically executed branches were linearly inseparable. On these branches alone, the perceptron predictor achieved a misprediction rate of 12.07%, compared with 8.77% for gshare. However, on the other 17.18% of the branches in `099.go`, the perceptron predictor achieved a misprediction rate of 3.68%, slightly better than gshare's 3.80%.

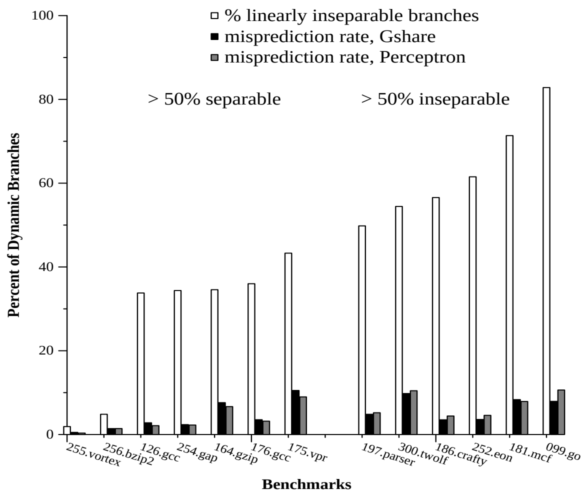

Figure 8: Linear Separability vs. Performance at a 512K budget. The perceptron predictor is better than gshare when the dynamic branches are mostly linearly separable, and it tends to be less accurate than gshare otherwise.

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 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. Since Evers *et al* show that most branches can be predicted by looking at three previous branches [9], we can use the perceptron predictor to find these best lengths: Using a perceptron trained for each branch, we find the most distant of the three weights with the greatest magnitude. As the best history length increases, the advantage of the perceptron predictor generally increases as well. Our predictor performs 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.

Knowing that the perceptron predictor does well on a particular type of frequently executed branches motivated the hybrid perceptron/gshare predictor, which is very good at distinguishing between predictors.

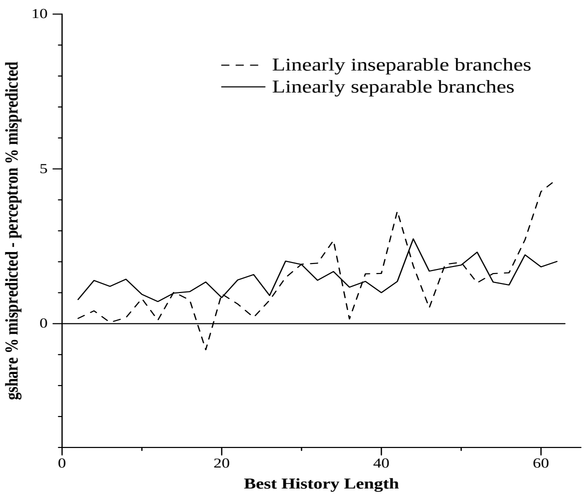

Figure 9: Classifying the Advantage of our Predictor. Above the x axis, the perceptron predictor is better on average. Below the x axis, gshare is better on average. For linearly separable branches, our predictor is more accurate than gshare on average. For inseparable branches, our predictor is sometimes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

5.6 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 [14]. 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 [13], but in our predictor this information comes free.

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 5.5 could be used with a profiler to provide path length information to the variable length path predictor [24].

5.7 Effects of Context Switching

Branch predictors can suffer a loss in performance after a context switch, having to warm up while relearning patterns [8]. We simulated the effects of context switching by interleaving branch traces from each of the SPEC2000 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 [6]. Note that previous studies have used the eight SPEC95 integer benchmarks, so our use of the 12 SPEC2000 benchmarks will likely lead to higher misprediction rates.

Figure 10 shows that context switching affects the perceptron predictor more significantly than the other two predictors. For example, without heavy context switching (Figure 4) the perceptron predictor is better than the PHT schemes at every hardware budget, but with context switching the perceptron scheme meets the performance of the others at a 16K budget. The perceptron predictor is affected by context switching more than the other techniques because it is more susceptible to aliasing. The hybrid gshare/perceptron predictor performs much better in the presence of context switching; this benefit of hybrid predictors has been noticed before [8]. From these results we conclude that to achieve good performance in conditions adverse to the perceptron predictor, we should use a hardware budget of at least 16K bytes, the same size as the Compaq 21264's branch predictor.

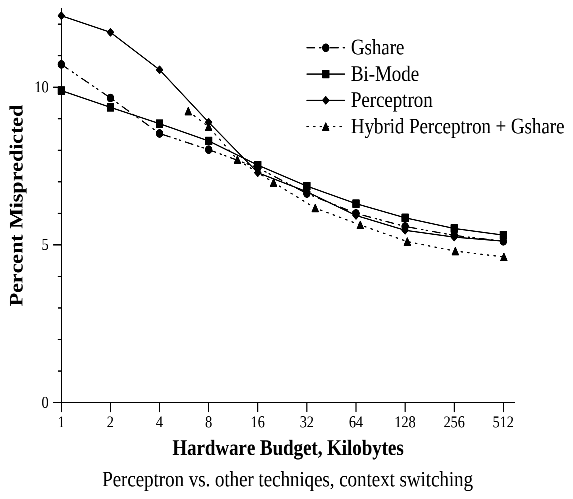

Figure 10: Budget vs. Misprediction Rate for Simulated Context Switching. The perceptron predictor is more affected by heavy context switching than gshare or bi-mode.

6 Implementation

We now show how 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.

Delay. A 54×54 multiplier in a 0.25μm process can operate in 2.7 nanoseconds [11], 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 [24]. That work proposes pipelining the predictor to reduce delay. We next describe a similar technique for our predictor that will often result in an effective prediction time of zero cycles.

Pipelined Operation. As we have described it, our scheme fetches a perceptron from SRAM and computes the sign of its output, all during the instruction fetch stage. To avoid prediction delays of more than one cycle, we have pipelined the prediction mechanism, so that the address of the i^{th} branch in a dynamic sequence is used to select a neuron for the $i + 1^{\text{st}}$ branch. We reorder the operations described in Section 3 as follows:

1. When a prediction for branch i is requested, we return the prediction computed by the previous iteration of this algorithm; if the preceding branch occurred more than a few cycles ago, this takes no time at all, since the bit is already available.

2. When the actual outcome of branch i is known, the current contents of the P register are trained and then written back to the table of perceptrons.

3. The global history register is updated. At the same time, the address of branch i is concatenated with the outcome of branch i (0 or 1) and hashed to select a perceptron for branch $i + 1$. (In our simulations, the hashing function is simply modulus.)

4. The selected perceptron is read into P.

5. A prediction for branch $i + 1$ is made using the updated global history register and the contents of P, and sent to a latch to be read when branch prediction is next requested.

If there are no indirect branches (e.g. jumps through tables, procedure returns, etc.) between branch i and branch $i + 1$, then the combination of the address and the outcome of branch i fully determines the identity of branch i, so the "right" perceptron is fetched before branch $i + 1$ is ever encountered. If there is an indirect branch between branches i and $i + 1$, then the perceptron chosen for branch $i+1$ may not always be the same. We have found that in practice accuracy is diminished by only about 0.1%. As long as branches do not occur in close succession, this mechanism effectively provides branch predictions in zero cycles. This scheme can likely be adapted to work with other global branch predictors to provide faster 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
```

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 for our composite SPEC2000 trace.

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 [24], 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 62.

We have also learned 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. Because these two classes represent a large number of dynamic branches, our predictor performs well.

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 [23]. We can also employ perceptrons in a hybrid predictor that uses both global and local histories, which have proven to work better than purely global schemes [8]. 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. Along these lines, we plan to tune our hybrid gshare/perceptron predictor and to explore other hybrid techniques. 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 5.6, 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.

Finally, studies have shown that as clock rates increase over the next 15 years, wire delays will make large data structures less feasible [1]. Finding ways to reduce the aliasing problem for small hardware budgets will be essential to the continued success of this and other branch prediction techniques. Using branch classification [5] to allow the compiler to choose among different predictors, e.g. between the perceptron predictor or gshare, will likely allow the same performance at a significant hardware savings. Thus, a deeper understanding the nature of branches for which the perceptron predictor performs better will be important for developing classification strategies.

Acknowledgments. It's our pleasure to thank Steve Keckler and Kathryn McKinley for many stimulating discussions on this topic, and to thank Steve, Kathryn, and Ibrahim Hur for their comments on earlier drafts of this paper.

References

[1] V. Agarwal, M.S. Hrishikesh, S. W. Keckler, and D. Burger. Clock rate versus ipc: The end of the road for conventional microarchitectures. In *the 27th Annual International Symposium on Computer Architecture (to appear)*, May 2000.

[2] 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.

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

[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] P.-Y. Chang and U. Banerjee. Branch classification: a new mechanism for improving branch predictor performance. In *Proceedings of the 27th International Symposium on Microarchitecture*, November 1994.

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

[11] 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.

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

[13] 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.

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

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

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

[17] 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.

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

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

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

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

[22] 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.

[23] 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 Conference on Computer Architecture*, June 1997.

[24] 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.

[25] 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.

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

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