Perceptron Learning for Predicting the Behavior of Conditional Branches

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

Abstract

Branch prediction*, i.e., predicting the outcome of a conditional branch instruction, is essential to the performance of current and future microprocessors. We show how perceptrons can be used to improve the state of the art in branch prediction. We explore the unusual challenges this domain presents for neural systems, and we show why other neural methods, such as back-propagation, provide no additional accuracy in this context. Finally, we identify other areas where neural systems can be applied to microprocessor implementation.*

1 Introduction

Modern microprocessors achieve good performance by executing many instructions in parallel. This *instruction-level parallelism* (ILP) can be limited by various bottlenecks, so microprocessors often perform speculative work to reduce the impact of these bottlenecks. One particularly important type of speculation is *dynamic branch prediction,* which predicts the likely direction of conditional branch instructions before the conditions have been decided. Current techniques can achieve correct branch prediction rates of 95% [1], i.e., *misprediction rates* of 5%, but the high cost of recovering from misprediction [2] remains one of the largest impediments to performance on current and future processors. Small improvements in accuracy can have a large impact on performance; decreasing the misprediction rate from, say, 5% to 4% can decrease the execution time of a typical program by as much as 14%, given reasonable assumptions about other aspects of the microarchitecture.

This paper describes how we have successfully used perceptrons in branch prediction [3], explains why the particular constraints of the problem favor simple methods over other methods such as back-propagation, and argues that the general area of hardware speculation is a rich one for neural systems.

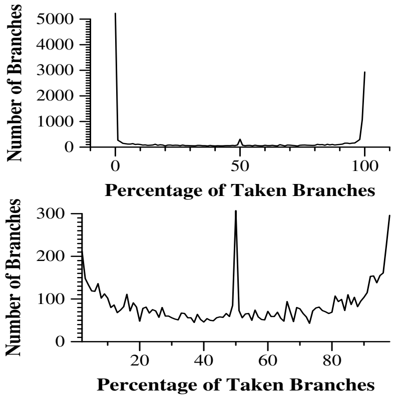

Figure 1: Bias in branches. The x axis gives the *bias* of a branch, i.e., the percentage of time a branch is *taken*, and the y axis shows the number of branches with a given bias in the SPEC 2000 integer benchmarks. Of all branches, 53% are *taken* at least 98% or at most 2% of the time. The graph on the right excludes these branches, again showing clear biases and a surprising number of branches taken exactly half the time.

2 Background and Related Work

Dynamic Branch Prediction. The outcome of a given branch is often highly correlated with the outcomes of other recent branches [4]. This history of branch outcomes forms a pattern that can be used to provide a dynamic context for prediction. Most modern branch predictors are based on this pattern history. Recent branch prediction work focuses on refining the scheme of Yeh and Patt [4, 5, 6]. In this scheme, every time a branch outcome becomes known, a single bit (0 for *not taken*, 1 for *taken*) is shifted into a pattern history register. A pattern history table (PHT) of two-bit saturating counters is indexed by a combination of branch address and history register. 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, and the pattern history is updated. One problem with such schemes is *aliasing*, where the limited memory causes two unrelated branches to use the same prediction resources, resulting in poor performance. Many techniques have been proposed to reduce aliasing [1, 5, 6]; these techniques work well in practice.

Neural Networks in Compilers. *Static* branch prediction uses program features, such as control-flow and opcode information, to predict branch behavior at compile time [7, 8]. Calder, et al. have shown how static prediction can achieve misprediction rates of 20% by supplying program information as input to a feed-forward neural network trained with back-propagation [8]. In general, static branch prediction performs worse than dynamic techniques, but can be useful for performing static compiler optimizations. Neural networks have also been used to schedule straight-line machine code in a compiler [9] to increase ILP.

Characteristics of Branch Prediction. Dynamic branch prediction has four characteristics that present challenges for neural methods. First, branches, which can only have two outcomes, *taken* and *not taken,* are highly biased. For instance, a branch that transfers control from the end of a loop back to the beginning will usually be *taken*, since loops usually iterate many times before finishing. Figure 1 shows the bias of branches in the SPEC 2000 integer benchmark suite. Second, branch predictors must operate while they are being trained, and they can never stop learning, since branch behavior may change over the course of a program. Thus, the prediction mechanism must learn quickly and adapt to changing behavior. Third, for a conditional branch to have a significant impact on the performance of a program, it must be executed many millions of times. Thus, for important branches, there are many training samples. Finally, branch predictors must meet strict physical constraints. They must operate in one CPU cycle, typically one or two nanoseconds, and be small enough to fit on a chip. Most of the hardware devoted to branch predictors is memory for large tables, so the *hardware budget* of a predictor, i.e., the cost of the predictor as a component of the chip, is appropriately measured in kilobytes. A typical predictor occupies 16K bytes of SRAM [10].

3 Neural Branch Prediction

Our dynamic branch predictor replaces the typically used table of two-bit counters with a table of neurons [3]. When a branch instruction is encountered, the address of the branch is hashed to select a neuron from the table. The selected neuron is used to predict the likely direction of the branch. Once the actual branch outcome is known, the neuron is updated with the training rule.

We use the Block neuron [11], a type of perceptron [12, 13], because it provides good accuracy and lends itself to a fast and compact representation in hardware. Although perceptrons cannot learn linearly inseparable functions with 100% accuracy [14], we have found that they work well in practice for branch prediction.

The inputs to the perceptron are the bits of the pattern history register, i.e. the outcome of the last several branches, represented as bipolar values. For a history length of n, the perceptron has $n + 1$ inputs: the n bits of pattern history and a constant bias input of one. For each input, there is a corresponding weight; the output is the dot product of the input and weight vectors. The branch is predicted *taken* if the output is at least zero, *not taken* otherwise. When the prediction is incorrect or the magnitude of the output is below a constant threshold θ, training is done by perceptron learning [15] with unit learning rate. To simplify the implementation, we use small integer weights with saturating arithmetic. The bipolar nature of the input and output data allows several optimizations so that a hardware implementation can operate quickly. The output of the perceptron is the dot product of the weights and input vector. The entire operation closely resembles an unsigned integer multiply, for which very efficient circuits exist [16]. The predictor can also be pipelined, i.e., consecutive predictions and trainings can be overlapped in time to provide faster operation. These two observations lead to a single-cycle implementation of our predictor. More details are available from our technical report [3].

3.1 Design Space

Given a fixed hardware budget, three parameters control the design of the perceptron branch predictor. We tuned each of these parameters for a set of hardware budgets using training data representing approximately 10% of the branches executed in the SPEC 2000 integer benchmarks.

History Length. The number of branch outcomes to store in the pattern history register has a great impact on the accuracy of the predictor. Longer history lengths yield greater accuracy but also require longer weights vectors. As the history length increases, the number of neurons that can be represented in the the same space decreases, resulting in increased aliasing. Thus, the history length must be tuned for each given hardware budget. (See Table 1.)

Table 1: Best History Lengths for Various Hardware Budgets. The perceptron predictor can use longer history lengths than the well-known *gshare* predictor.

Hardware budget in kilobytes	History Length
1	12
2	22
4	28
8	34
16	36
32	59
64	59
128	62
256	62
512	62

Number of bits per weight. Each weight is represented by a signed integer. The size of this integer affects accuracy. If the weights are too small, the perceptron will be unable to effectively learn correlations between the history and the branch outcome. If the weights are too large, aliasing can adversely affect accuracy. We have empirically found that 9 bit weights provide the best balance. The accuracy of the predictor is not particularly sensitive to this parameter.

Threshold. The threshold θ controls the invocation of the training algorithm. To achieve best accuracy, this parameter must be tuned for each possible history length. Interestingly, we have experimentally found that the best threshold θ for a given history length h is always *exactly* $\theta = \lfloor 1.93h + 14 \rfloor$ for our benchmarks. This is because adding another weight to a neuron increases the average output of the neuron (before thresholding) by some constant, so the threshold must be increased by a constant, yielding a linear relationship between history length and threshold.

3.2 Accuracy

We simulated the perceptron predictor along with the *gshare* predictor (tuned for history length), a PHT scheme that is a standard against which other predictors are judged. The *gshare* scheme uses the exclusive-OR of the branch address and history register as an index into the PHT; this scheme has proven effective in distributing two-bit counters evenly among branches, reducing destructive aliasing. Using simulation, we measured misprediction rates at various hardware budgets. Figure 2 shows the accuracy of each predictor for the `126.gcc` program, a standard benchmark for branch prediction, and for a composite sample of 100 million branches from each of the SPEC 2000 integer benchmarks. For `126.gcc`, the perceptron predictor improves the misprediction rate by 25.6% over *gshare* at a hardware budgets of 128K bytes. This improvement is significant: Given reasonable assumptions about the microarchitecture, this improvement would result in a decrease of up to 18% in the execution time of the program. For the composite sample, the perceptron predictor improves accuracy by 5.4% over *gshare* with a budget of 128K bytes. Note that since learning is done online, there is no need for separate training and testing sets to report accuracy.

3.3 Analysis of the Predictor

One of the main benefits of the perceptron predictor is its ability to consider much longer histories than other methods [3]. PHT-based predictors can only consider history lengths of about 17 since the number of table entries is exponential in the history length. This is a problem when the distance between correlated branches is longer than the length of a pattern history shift register [17]. 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 [18]. One scheme has been proposed

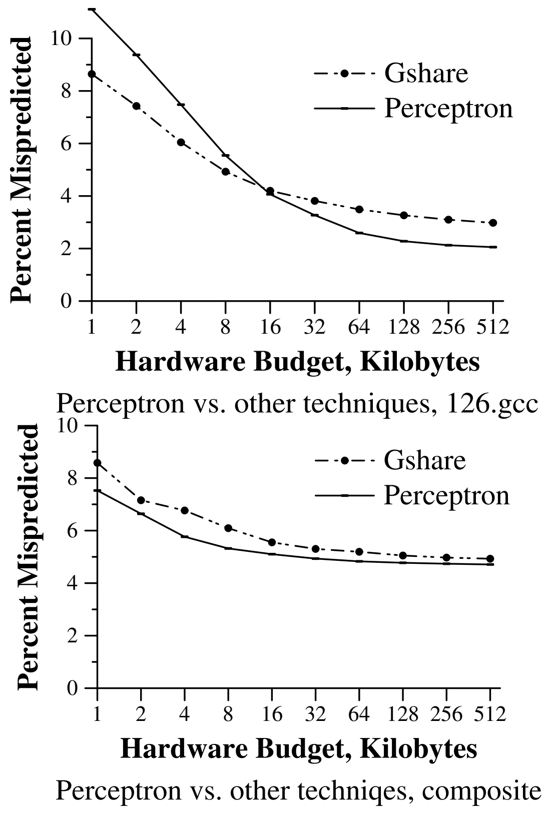

Figure 2: Hardware Budget vs. Prediction Rate. The perceptron predictor is more accurate than *gshare* for hardware budgets over 16K for `126.gcc`. On the composite of all benchmarks, the perceptron predictor achieves greater accuracy at all hardware budgets.

to support long histories by using variable length path histories [19], but this requires complex profiling that is impractical.

Perceptrons are unable to learn linearly inseparable functions with 100% accuracy. To measure this effect on our predictor, we performed an experiment that computed the "ideal" prediction function for each branch, i.e., the function of the branch history with the lowest misprediction rate, for each benchmark, and simulated the perceptron predictor and *gshare*, each with a 512K byte budget. When more than 50% of these prediction functions are linearly separable, the perceptron predictor is always more accurate than *gshare*. When the opposite is true, the perceptron predictor is less accurate. On average, 40% of the branches in a benchmark are linearly inseparable, so the perceptron predictor is usually more accurate. Also, joining *gshare* and the perceptron predictor in a hybrid predictor results in even better accuracy [3].

Comparison with Back-Propagation. We compared the perceptron predictor with another standard neural technique, back-propagation. A hardware implementation of back-propagation would be too slow for branch prediction, since the more complex multi-layer architecture would require many cycles for both prediction and training. For this experiment, we ignored the hardware budget and the need to use a fixed number of bits for the weights; the goal was to understand how different neural approaches perform independent of these constraints. We compared the performance of both predictors for history lengths between 5 and 100. For back-propagation, we used one hidden layer with eight units, and we tuned the learning rate. On the `126.gcc` benchmark, with a history length of 60, the perceptron predictor has a misprediction rate of 2.44%, compared with 3.33% for back-propagation. We would expect back-propagation to perform better than perceptron, since it can learn linearly inseparable functions. However, back-propagation takes longer to learn branch behavior than the perceptron. Figure 3 shows the accuracy and training times of back-propagation and perceptrons on the `126.gcc` benchmark. At each history length, the perceptron predictor is more accurate than back-propagation. Although back-propagation should be able to asymptotically exceed the accuracy of the perceptron, the longer training time for back-propagation causes it to be slightly less accurate overall.

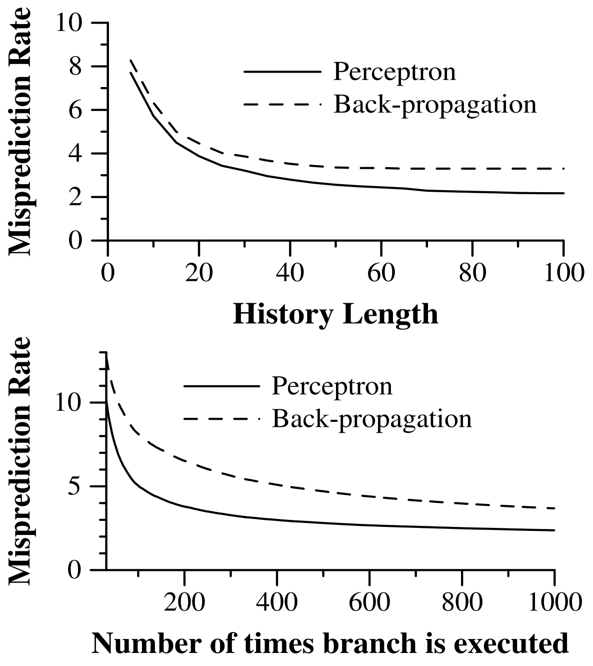

Figure 3: Comparison of Perceptron and Back-Propagation. On the left, the x-axis is history length and the y-axis is misprediction rate. As history lengths increase, the advantage of the perceptron predictor over back-propagation also increases. On the right, the x axis is the number of times a branch has been executed (starting at 30, where the learning rates start to diverge) and the y axis shows the average misprediction rate at that time; only branches executed at least 1000 times are included. Although back-propagation is more powerful, the perceptron is able to learn more quickly.

4 Future Directions

The increasing use of speculation in modern processors has opened many opportunities for new research. Currently, *ad hoc* schemes are used to drive predictions. We believe that accuracy can be increased by using neural techniques. As mentioned earlier, neural networks can be applied to data prefetching and compiler optimizations, but there are many other areas in which neural predictors can be helpful:

Indirect Branch Prediction. Indirect branches, i.e., branches through a pointer such as virtual method calls, calls through jump tables, and function returns, also need to be predicted. A neural branch predictor could be extended to predict the target address of these branches by choosing from among a small set of recently used addresses.

Assigning confidence to decisions. A processor can use a confidence in a prediction to guide speculation. For example, if a branch is predicted *taken,* but confidence in that prediction is low, the processor may choose to speculatively execute both the *taken* and *not taken* branches, dropping the wrong path when the branch condition becomes available. When confidence is high, the processor may choose to execute the predicted path only, saving execution resources for other concurrent computations. A scheme to compute confidence levels for branch predictions has been proposed [20], but the output of a neural system, when interpreted as a probability [21], can provide the same confidence level as a free side-effect.

Value Prediction. Some processors predict the value that an instruction will compute before it has been executed, so that the value can speculatively be fed to another instruction executing in parallel [22]. Neural systems could be used to predict which of a set of previously computed values the instruction is likely to compute. Value predictors require a level of confidence in a prediction before they will work, so neural systems seem particularly well-suited to this application.

5 Conclusion

Modern microprocessors increasingly rely on speculation to boost ILP. Thus, improvements in prediction mechanisms are critical to the performance of microprocessors. Until now, hardware-based prediction techniques have ignored neural methods. However, technology trends, such as the availability of large hardware budgets, now make such methods feasible, particularly if they can be designed to produce results quickly with on-line training. The perceptron predictor is one such technique, and we can anticipate the development of many others if the computer architecture and neural computing communities come together to study speculation.

Acknowledgments. This research was funded in part by DARPA Contract #F30602-97-1-0150 and by NSF CAREER grant ACI-9984660. We thank Steve Keckler, Kathryn McKinley and Risto Miikkulainen for their help in this work.

References

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

[2] Brad Calder and Dirk Grunwald. Fast and accurate instruction fetch and branch prediction. In *Proceedings of the 21st International Symposium on Computer Architecture*, pages 2–11, April 1994.

[3] Daniel A. Jiménez and Calvin Lin. Dynamic branch prediction with perceptrons. In *Proceedings of the Seventh International Symposium on High Performance Computer Architecture*, January 2001.

[4] Tse-Yu Yeh and Yale N. Patt. Two-level adaptive branch prediction. In *Proceedings of the 24^{th} ACM/IEEE International Symposium on Microarchitecture*, November 1991.

[5] 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.

[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] 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.

[8] 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.

[9] Elliot Moss, Paul Utgoff, John Cavazos, Doina Precup, and Darko Stefanovic. Learning to schedule straight-line code. In *Neural Information Processing Systems*, December 1997.

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

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

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

[13] M. L. Minsky and S. A. Papert. *Perceptrons, Expanded Edition*. MIT Press, 1988.

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

[15] Christopher M. Bishop. *Neural Networks for Pattern Recognition*. Oxford University Press, 1995.

[16] 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.

[17] 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.

[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] 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.

[20] 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.

[21] 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.

[22] 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.