

On Accurate and Efficient Perceptron-Based Branch Prediction

Engin Ipek, Sally A. McKee, Martin Schulz, and Shai Ben David

School of Electrical and Computer Engineering
Cornell University
Ithaca, NY, 14853
{engin,sam,schulz}@csl.cornell.edu, shai@ece.cornell.edu

Abstract

Exploiting the huge computing power of modern microprocessors requires fast, accurate branch predictors: as clock rates rise and pipeline lengths grow, so do branch misprediction penalties. These latencies currently represent one of the largest microarchitectural performance bottlenecks. The literature is rich with techniques that increase accuracy, reduce prediction latencies, or both. Nonetheless, even with seemingly high prediction accuracies, branch misprediction latencies still have significant, negative performance effects on codes that are not memory bound.

Some of the most novel recent proposals are based on the perceptron, a simple neural network component. We study of a set of perceptron-based predictors that improve prediction accuracies by 18.2% to 69.7%, with a mean of 37.6%. On the basis of these findings, we explore performance for a set of new perceptron-based predictors, comparing them to traditional schemes and the best perceptron-based predictors published at the time of this writing. For an Alpha 21264-like architecture, the new predictors increase IPC by up to 14.14% over the 21264, and up to 9.28% over the original perceptron (with means of 5.9% and 3.67%, respectively). For a 20-stage pipeline, our designs improve IPCs by up to 19.35% over the original perceptron.

1 Introduction

As feature sizes shrink, clock frequencies rise, and pipeline lengths grow, the performance penalty for pipeline flushes on mispredicted branches grows in severity: for deeply pipelined superscalar processors, branch misprediction latencies currently represent one of the largest microarchitectural performance bottlenecks [?]. Fast, accurate branch prediction is crucial to performance, and its importance continues to increase.

Most dynamic, adaptive branch prediction techniques rely on one or more Pattern History Tables (PHTs) of saturating up-down counters [?] indexed by some func-

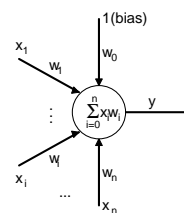


Figure 1. Perceptron for Learning Branch Behavior

tion of the branch address, a global Branch History Register (BHR), and/or local information tracked per branch. Myriad variations and hybrid combinations are possible. The simplicity of the saturating counter approach combined with effective cost/performance tradeoffs has made this general family of predictors the mainstay for branch prediction, as well as for many other types of speculation.

In contrast, Loh and Henry [?, ?] and Jiménez and Lin [?] address speculation from a Machine Learning point of view. The latter explore the application of neural networks to dynamic branch prediction [?]. Training algorithms for most neural nets are too complex to render efficiently in hardware, but the *perceptron* [?] algorithm is simple enough to enable perceptron-based, dynamic, hardware branch predictors, and it describes a sufficiently rich class of classifiers to yield increased prediction accuracy.

A perceptron is a linear threshold function: it computes a weighted sum of its inputs, compares this sum to a prescribed threshold value, and outputs 1 or -1 to reflect the result. It computes its weights based on the sequence of inputs and target values it observes. The inputs may represent the taken/not-taken bits stored in a Branch History Register (BHR), and the output may be used to predict whether the branch associated with this perceptron will be taken. Figure 1 depicts a single-layer, single-neuron perceptron for branch prediction: each input x_i is associated with a weight w_i to indicate the correlation between this input (behavior of a recent branch) and the output y (current branch decision). A *bias input* with w_0 is used to learn the general branch behavior. These perceptrons are stored in a Weight Table (WT) that replaces the traditional PHT.

Practical challenges in implementing high-performance, perceptron-based predictors remain manifold. It is widely held that perceptron-based predictors cannot predict inseparable branches [?]. Nonetheless, one of our findings is that the perceptron’s online learning algorithm can adapt well to branch behaviors that are linearly separable for sufficient windows of their dynamic executions. Exactly how to best exploit this property is an open question, as is how to alleviate sources of nonlinearity, in general. Theoretical issues aside, the requirements of aggressively speculative, high clock-rate architectures constrain the set of implementable perceptron-based predictors. Latencies to access the weight table (WT) and compute the weighted sum of the inputs impose limits on predictor size, affecting accuracy.

We classify the theoretical and practical limitations of perceptron-based prediction in Section 3, and propose techniques to address them in Section 4, describing the design and implementation of a set of perceptron-based predictors. In Section 5, we study the accuracy of our proposed approaches on eight of the SPECint 2000 benchmarks, narrowing our design space to those configurations most promising in terms of accuracy/latency tradeoffs. We then provide detailed simulation performance comparisons of these predictors to others with similar access times, using SimPoint [?] to conduct more comprehensive cycle-accurate simulation experiments than have been previously published for perceptron-based branch predictors (full details are in the accompanying technical report [?]).

Our designs reduce misprediction rates by 1.37-11.11% (for accuracy improvements of 18.2-69.7%) compared to the best published alternative predictors with the same latencies. On a subset of eight of the SPECint 2000 benchmarks, these increases in accuracy translate to mean speedups of 5.9% compared to the Alpha 21264 predictor, 3.67% compared to the original perceptron predictor, and 5.90% compared to the original perceptron adapted to an Alpha-like machine with an extended, 20-stage pipeline.

2 A Brief Tour of Prior Work

Our work builds on the tremendous amount of prior research in dynamic, adaptive branch prediction. Citing the complete wealth of work would alone exceed this article’s length limitations. The next section provides a description of proposed perceptron-based prediction techniques, including a discussion of the limitations that inhibit their adoption in commercial systems. Uht et al. survey branch effect reduction techniques, including dynamic branch prediction [?]. Yeh and Patt identify a taxonomy of two-level branch predictors [?], and Lu et al. provide a taxonomy of mispredictions [?]; together, these allow architects to speak precisely about performance of many variations.

The simplest approach uses a single bit to predict that

a branch will behave the same as its previous execution. Smith proposes dynamic techniques based on a table indexed by a function of the branch address and containing two-bit up/down saturating counters [?]. These structures assume a bimodal distribution for a branch’s behavior: it will either typically be taken or not taken, but it will not be heavily randomly. Yeh and Patt introduce *two-level branch prediction*, keeping a global Branch History Register (BHR) to record outcomes of recent branches [?]. The BHR is used to index a Pattern History Table (PHT), whose entry is used to predict the current branch and is updated when the true direction is known. In contrast to *global prediction*, *local prediction* keeps history information for each branch. Different predictor organizations assign branches or branch streams differently to entries in the PHT: an index may be computed using local or global branch history, alone or in combination with branch address information. Yeh and Patt explore implementation tradeoffs for two-level predictors [?, ?], and Yeh et al. add a Branch Address Cache (BAC) and speculate through more than one branch prediction [?]. Pan et al. combine global history with branch address information to index the PHT [?]. Since each approach — bimodal, local, and global — works well under some circumstances, McFarling combines them, adding per-branch history to track the most accurate predictor; he exploits global history effectively by hashing it with the branch address [?]. This hybrid *gshare* predictor achieves better accuracy than previous approaches (even with multiple levels of history) at smaller hardware budgets.

Evers et al. provide insight into the behavior of the two-level, bimodal predictors that provide the foundation for most modern, adaptive, dynamic branch prediction techniques [?]. Implementation constraints can pose significant performance limitations: e.g., length of recorded history information is usually limited to be at most the log of the entries in the table being indexed. Performance suffers when highly correlated branches occur at relative distances too large for the BHR to capture. Longer histories may improve accuracy, but they increase hardware complexity and training and access times. Keeping latencies tolerable usually requires limiting PHT sizes. Size limitations can cause problems when more than one index maps to a PHT entry. Attempts to reduce destructive aliasing in two-level predictors include partitioning them into sections representative of different modes of execution [?], partitioning and skewing indices [?, ?], and converting destructive to neutral aliasing. For instance, the *agree* predictor keeps a bias bit indicating a branch’s direction when it is loaded into the Branch Target Buffer (BTB); and the prediction then becomes *agree* or *disagree* rather than *taken* or *not taken* [?].

Several sophisticated schemes circumvent size limitations by using lookahead or cascading predictors, or by taking a fast, simple predictor (such as *gshare*), backed by

a slower, more accurate overriding predictor that can correct mispredictions at a smaller penalty than a full pipeline flush [?, ?, ?, ?]. Attempts to increase accuracy by choosing the appropriate history on which to base a prediction give rise to another interesting set of hybrid predictors. Nair considers previous branch addresses (paths) in computing the predictor index [?]. Alloying local and global history improves performance of two-level hybrid predictors, reducing aliasing by choosing bits from both histories, and increasing accuracy by using as many local history bits as possible [?]. Loh and Henry apply a Machine Learning approach to selection in their fusion-based hybrid predictors [?]. Seznec et al. [?] give a particularly detailed description of a modern, commercial design, describing trade-offs in the branch predictor in the Alpha EV8 processor (a project cancelled before production). This 2Bc-gskew [?] pipelined predictor uses compressed global branch and path history to index its tables; different sizes for different tables prediction and hysteresis tables, which are accessed at different pipeline stages; and a metapredictor to choose which prediction to follow. Seznec proposes optimizations to 2Bc-gskew [?]: sharing physical tables among logical tables, and introducing randomness into predictor update functions to avoid oscillating behavior. 2Bc-gskew predictors use much longer histories than most previous schemes, including the original perceptron-based predictor.

Jiménez [?] argues that technology trends preclude the use of large, complex predictors, and shows that access latency can dominate prediction accuracy with respect to performance. He proposes a pipelined organization similar to one we use in Section 4, applying it to a McFarling-style [?] gshare predictor. Pipelining enables larger tables with single-cycle prediction latencies, and Jiménez shows that at large hardware budgets, this approach can outperform the best techniques from the literature.

Recent departures from traditional schemes include using value prediction of dependent register to aid in predicting the branch outcome [?, ?], and applying Machine Learning [?, ?] to the prediction problem. Our work takes the latter approach; in particular, we leverage components from neural networks [?, ?, ?].

3 Perceptron-Based Branch Prediction

The perceptron algorithm is a fast *online* procedure for training a weighted majority predictor to fit input data. Since the perceptron views one data point at a time (rather than examining its training data all at once), information gathered from arbitrarily long history sequences is represented by a fixed size set of parameters (the perceptron weights). Perceptron-based branch predictors have hardware complexities that scale linearly with the number of inputs, and thus they can use longer histories to index their

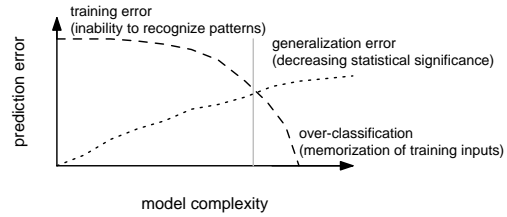


Figure 2. Sources of Prediction Error

	aliasing effects	insufficient input data	inherent data limitations
insufficiency	ia	id	ii
nonlinearity	na	ni	ni

Table 1. Characteristics of Perceptron-Based Prediction

tables than other predictors that rely on global history (with the exception of 2Bc-gskew described above).

3.1 Sources of Prediction Error

Machine Learning uses training data to select a predictor from a limited class of functions. Learning theory provides guarantees on the accuracy of such predictors in terms of error on the training data and complexity of the class of functions from which they are selected. The larger the class, the more likely is finding a predictor fitting the training data well (small *training error*), but the richer this class, the greater the risk of overfitting the data (larger *generalization error*).

The perceptron algorithm chooses from a severely limited class of functions, namely, the class of half-spaces (equivalently, weighted majority functions, hyperplanes, or linear separators) over its set of input features. Given m training data points, each described by k features (a k -dimensional vector), there are 2^m possible ways to classify these points, while there are only $O(m^{k+1})$ linear classifiers from which to choose. Using multiple perceptrons (instead of one perceptron for all points) increases the probability of finding a linear classifier. The mapping of inputs to weights is decided statically for branch prediction — it’s encoded in the parameters of the dynamic branch prediction hardware. In a reconfigurable architecture, many possibilities exist, and machine learning algorithms (either offline or online) could be used to generate this mapping.

From the point of view of theoretical performance guarantees, enriching the class of potential predictors may decrease the statistical significance of our model. Figure 2 broadly depicts these tradeoffs. Existing perceptron-based predictors fall to the left of the gray line where the error curves cross. A deeper discussion of the mathematics involved and their implications is part of future work.

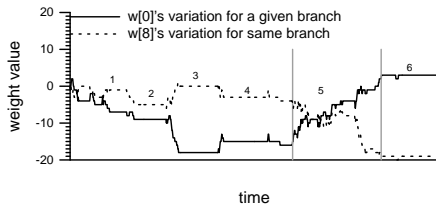


Figure 3. Weight Fluctuations for a Perceptron in 176.gcc

3.2 Implementation Limitations

The limitations of any perceptron-based scheme are closely tied to the predictors implementation parameters and to application program characteristics. These limitations can be classified by two problems with respect to data considered in making predictions: *insufficiency* of the features used by the perceptron, and *nonlinearity* of the relationship between data features and labels. No good linear predictor may exist: training data may contain identical feature vectors with different labels (hence *any* function of the feature vectors will fail to predict accurately), or the only accurate predictors may be non-linear.

Each of these may be due to aliasing among WT indices, limitations on the amount of information that may be input to the perceptron, or inherent limitations in information provided by available data (e.g., the branch history and address being insufficient for making a correct decision). Table 1 shows these relationships, allowing us to classify existing approaches and to motivate the new designs we propose. This is but a coarse analysis, and there exist complex inter-relationships among these limitations, e.g., due to the use of the perceptron *algorithm*, limitations due to the implementation parameters chosen for a perceptron-based predictor.

For instance, as the number of perceptrons (table height) in a predictor increases, the hardware budget may require reducing the weights stored per line (table width). Applying more perceptrons may help overcome non-linearity issues, but puts more burden on the WT access stage. Similarly, as the number of weights per perceptron grows, insufficiency with respect to input data increases (id, ii). However, access times and hardware budgets prevent extremes in the context of branch prediction. Carrying either parameter to the extreme puts us to the right of the vertical bar in Figure 2, degrading predictor accuracy. Ultimately, the architect is limited by access time constraints, and must choose a good balance among parameters. In spite of its inability to capture all interactions, the table helps frame our discussion of tradeoffs within and among existing and proposed perceptron-based dynamic branch-prediction schemes.

Jiménez and Lin [?] study two schemes: a *global perceptron predictor* (our baseline) using fast perceptrons in place of saturating counters [?], and a *global/local perceptron predictor* tracking per-branch history along with the

perceptron weights [?]. Proposed configurations include using their predictors as secondary or correcting predictors backing a small, fast primary predictor (such as gshare).

The baseline perceptron-based predictor suffers from multiple problems in Table 1), and in spite of good empirical accuracies, no performance guarantees exist in the face of linearly inseparable branches. (Figure 3, discussed below, helps explain why this predictor still achieves good accuracies, even though Jiménez and Lin [?] find many branches to be linearly inseparable.) Practical limits on prediction latency require smaller tables (fewer perceptrons) and shorter histories (fewer inputs) than might deliver best accuracies, and hence the information available to the perceptrons is limited by aliasing (ia) and the branch history that can be utilized (id), as well as insufficiencies inherent in the inputs (ii). Dynamic hardware techniques cannot overcome inherent data insufficiencies (ii), but compiler techniques could potentially introduce additional information or otherwise favorably alter program characteristics (ni, ii) to facilitate better prediction accuracy.

Seznec compares *redundant history skewed perceptron* predictors [?] to a hypothetical, conflict-free perceptron predictor, finding the former more accurate than the idealized version — often significantly more. He splits the perceptron WT into several physical tables indexed by different hashing functions, and evaluates his optimizations via trace-driven simulation with immediate predictor updates, just as we do in our accuracy design-space studies in Section 5.2. We examined similar redundant-history techniques in our early explorations, achieving results similar to Seznec’s.

3.3 Understanding Nonlinearity

Figure 3 illustrates the values of two weights, the 0^{th} and 8^{th} , for a single static branch in 176.gcc (all chosen arbitrarily). The figure shows general perceptron behavior over 1000 dynamic instances of the branch, and is broken into six, numbered, *windows* of time. These correspond to program phases, and transitions between them are readily observed. For windows 1-4 and 6, the weights converge and maintain their values over most of the phase. This shows the perceptron adapting to *locally* separable inputs (with respect to time), even given occasional inseparable inputs. In contrast, in window 5 (marked by the gray vertical lines), frequent variations in branch behavior make it linearly inseparable. Prediction during this phase is less accurate.

4 Design Optimizations

Figure 4 shows the organization of Jiménez and Lin’s initial, global perceptron predictor [?]. We refer to this as the *baseline perceptron* predictor throughout the rest of this

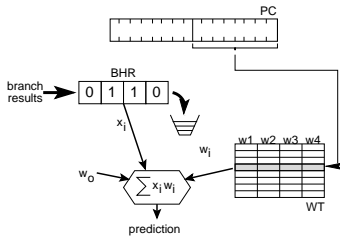


Figure 4. Global Perceptron-Based Predictor

article. The WT is indexed by a function of the branch address, and every static branch is ideally assigned its own perceptron, i.e., a separate set of weights for the bit positions in the global BHR. In the weighted sum, a zero bit in the BHR is interpreted as -1. If the sum exceeds zero, the perceptron predicts *taken*. Otherwise, it predicts *not taken*. In updating the perceptron, the weights for the bit positions whose entries agree with the branch outcome are incremented, and those whose disagree are decremented.¹ We study several variations on perceptron-based branch predictors: 1) learning PC tag bits (*pseudotagging*); 2) partitioning the prediction table (attempting to increase piecewise linear separability of branches); 3) backup caching of perceptrons from a fast, first-level predictor (effectively increasing the number of perceptrons, and hence the number of possible linear classifiers); and 4) inverting and pipelining the perceptron table (so each perceptron becomes an expert on expected behavior given a recent global history instead of expected behavior for a particular branch).

Pseudotagging. We reduce branch aliasing effects by learning branch tags (addressing limitations ia, id, na, nd from Table 1), maintaining reasonable hardware budgets and fast access times. This technique was independently proposed independently by Seznec [?], and we adopt his terminology. Figure 6 shows our predictor organization.

Partitioning. If black circles and gray diamonds represent different input samples in a two-dimensional space, then Figure 5(a) illustrates an example for which the optimal classifier is linearly inseparable (e.g., the gray dotted curve). No straight line divides the circles from the diamonds. Figure 5(b) illustrates a partitioning of the problem space such that there exists a linearly separable boolean function dividing the data *within each partition*. The more partitions, the more likely that there exists a linearly separable function that divides the data, but design tradeoffs must be balanced for best performance (addressing nd, nd and id, id). We statically partition the prediction table, increasing the number of linearly separable functions the predictor uses to map

¹An article entitled “Fast Path-Based Neural Branch Prediction” by D. Jiménez, will be published in MICRO-36, December, 2003. Prof. Jiménez’s patent lawyers insist that he refrain from discussing that work prior to the conference. The ideas presented and evaluated here have been developed independently.

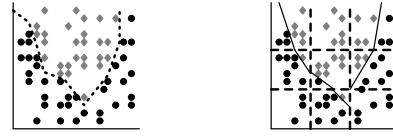


Figure 5. Linear Inseparability and Partitioning

branches to their predicted outcomes (addressing nd, nd). Some of the unused global history bits are used to select one of N possible perceptrons for a given branch, and only the partition that makes the prediction is updated. Figure 7 shows the organization of our predictor.

Weight Caching. Aliasing (ia, na) remains a significant source of error for perceptron-based branch predictors. If two different static branches with different behaviors map to the same line in the WT, a perceptron is likely to make mistakes. Destructive aliasing is most severe when the branch mapping to an entry changes frequently, leaving no time for the perceptron-based predictor to adjust to the non-linearity. Larger WTs reduce destructive aliasing, but they take longer to decode/access, and can relegate the perceptron-based predictor to be used only a high-latency overriding predictor. To prevent destructive aliasing while keeping the low access times, we propose a *weight cache* (WC) organization: a small, fast WT backed by a much larger, slower second-level WC. The WT and WC lines are tagged with the PCs of branches currently using them. The predictor performs the weighted sum calculation using weights resident in the WT, and performs the WT tag comparison simultaneously. Prediction is always made on the basis of this weighted sum calculation. If the tag comparison fails, the WT initiates a writeback of the current weights, requests the weights for the new PC, and sets the corresponding WT entries to zero. Until the WC access completes, the WT predicts branches based on its current contents. If the branch hits in the WC, the corresponding perceptron is returned to the WT, overwriting the previous perceptron in the line to which the branch maps. If the branch misses in the WC, the perceptron uses the current WT contents. The WC writeback policy and zeroing of weights on a mispredict guarantee that any perceptron in the WC is alias-free: weights written to the WC are generated by updates of a single static branch instruction. Since the prediction is made based on the WT’s weights (regardless of whether the tag hits), the cycle time of the predictor is unaffected. Figure 8 shows our predictor organization.

Combinations. We combine partitioning for the first-level WTs with backing, second-level WCs, efficiently reducing non-linearity due to aliasing and linearly inseparable branches (na, nd). Every partition is given its own weight cache, and works in parallel with all of the other partitions.

We omit accuracy graphs for these organizations, but provide detailed performance results in Section 5.3. Figure 10 shows our predictor organization.

Inverting the Weight Table. Destructive aliasing can also be reduced via pipelining the predictor so that larger tables can be used without incurring longer prediction latencies. Jimenez [?] shows a simple and effective mechanism for pipelining any branch predictor that uses the global history as an indexing function. Since the baseline perceptron predictor uses branch instruction addresses to index its WC, and since PC information is available only at the beginning of the cycle in which the prediction must be made, straight-forward pipelining is prevented.

In order to circumvent this shortcoming of the baseline perceptron branch predictor, we propose keeping an *inverted weight table*. Instead of using the branch PC to index the WT and the history bits as inputs to the perceptron, we use the n -branch old history bits from the BHR to index the WT, and use the PC bits in conjunction with the newer history bits (not available when calculating the WT index) as inputs to the perceptron. For a 2^M -line WT pipelined to be accessed in N cycles, the predictor functions as follows:

1. The lower M bits of the BHR are calculated as an index into the WT, and the WT access is initiated. The index is also placed in a queue of length N .
2. When the WT returns the weights, they are placed in the corresponding queue entry.
3. When the index/weight combination reaches the end of the queue, the lower bits of the current PC are alloyed with the lower N bits of global history, and these are used as inputs into the perceptron. The weights for the perceptron are copied from the queue.

The queue ensures that N new history bits have been pushed into the BHR by the time the prediction is made. Otherwise, there exist cases where the table access takes N cycles but during which the BHR is updated fewer than N times. The inverted WT can be pipelined so that the predictor access time is decoupled from the table access, making prediction latency a function of just the weighted sum calculation. Using this technique, it is possible to leverage taller tables (more perceptrons, id,id) as well as more inputs per branch in the weighted sum calculation id, nd. The technique relies on pipelined memory structures for the WT, so that reads and updates can be in flight simultaneously. There are a number of ways to achieve this, and we discuss some options in our technical report [?]. Figure 9 illustrates the organization of our predictor.

5 Comparative Results

5.1 Experimental Setup

We use a representative subset of the SPECint 2000 benchmarks with test inputs for our trace-based accuracy

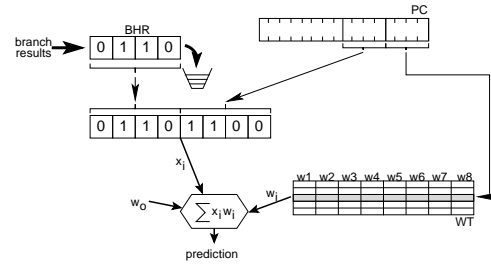


Figure 6. Pseudotag-Learning Perceptron Predictor

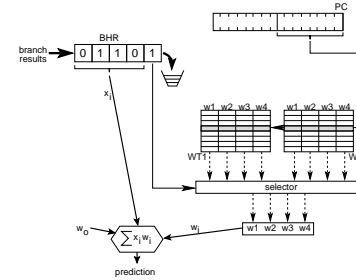


Figure 7. Partitioning to Reduce Linear Inseparability

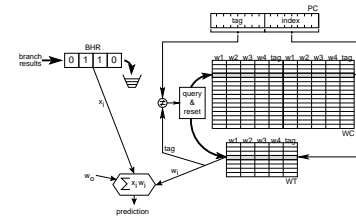


Figure 8. Caching Perceptron Weights

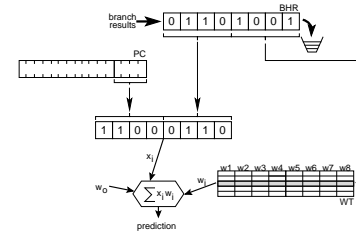


Figure 9. Inverted Perceptron Predictor

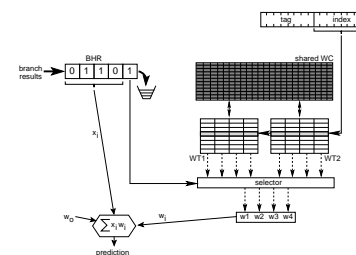


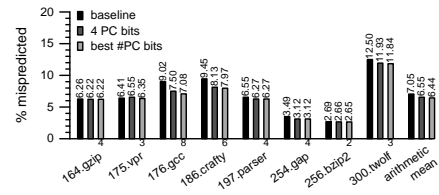
Figure 10. Sharing a Weight Cache among Partitions

studies, and with reference inputs for our simulation studies². We choose test inputs for the former because we conduct tens of thousands of experiments in our exploration of a very large predictor design space, and using reference inputs seems unnecessary for choosing the interesting design points to simulate. Statistics (input set, dynamic instruction count, error rates for IPC and branch prediction of the multiple simpoints) for each benchmark used here are shown on the SimPoint website [?]. When multiple reference inputs exist, we choose those with lowest observed simulation error under SimPoint.

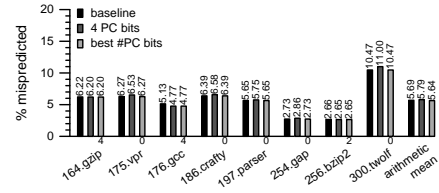
Jiménez and Lin employ trace-based analysis, skipping 50 million branches and then analyzing the following 250 million or performing cycle-accurate simulation of the first few billion instructions for each benchmark. Unfortunately, our results are not directly comparable to those of Jiménez and Lin: we also conduct trace-based experiments, but for whole program runs using SPECint 2000 test inputs (this let us pare down the predictor design space and validate our initial SimpleScalar branch prediction statistics); we use a different simulation model (4.0 vs. 3.0); and we study comparative branch predictor performance across whole-program behavior for realistic workloads, leveraging the multiple-SimPoint techniques developed by Sherwood et al. [?]. Since we simulate multiple simpoints [?], our approach should produce high-confidence statistics. We repeat many of Jiménez and Lin’s experiments in our experimental contexts, observing the same baseline trends they report.

Our trace-based studies use ATOM [?] on Alpha 21264 platforms. The target code is instrumented to perform callbacks on every dynamic branch, and these callbacks model the branch predictor under investigation. For cycle-accurate simulation experiments, we use SimpleScalar 4.0 with the Alpha instruction set. For most experiments, we configure the simulator as closely as possible to the validated model of a Compaq DS-10L Alpha Server, as described in previous studies [?, ?]. The memory system is a 64KB, two-way associative L1 cache with 64-Byte lines and three-cycle latency followed by a 2MB direct-mapped L2 cache with a 13-cycle latency. We use the default eight-entry victim cache, and eight MSHRs per cache. This version of the simulator models the bus and the SDRAMs, in contrast to previously released versions of the toolset. We then extend the pipeline to 20 stages to model expected effects of using our branch predictors on more aggressively clocked machines. We experiment with 833MHz and 1.76GHz frequency processor models, the latter with a 20-stage pipeline. We choose our baseline configurations at each frequency with respect to the delay estimates reported by Jiménez in [?] (in which delays of the Wallace trees and table accesses for im-

²We drop the SPECfp 2000 benchmarks, since their behavior tends to be more regular, and hence less interesting for studying branch predictors.



(a) 128-Line WT, 24 weights



(b) 1K-line WT, 24 weights

Figure 11. Accuracy for Pseudotagging

plementing perceptron-based predictors are analyzed with HSpice and CACTII), such that configurations we directly compare have equal prediction latencies.

5.2 Accuracy Results

We examine WT lengths of 128-4K entries, perceptron widths of 8-64 weights, and training thresholds from five-30, in addition to parameter ranges specific to each optimization (those we described, and those we omit for brevity or performance reasons) [?]. Figure 11 through Figure 15 illustrate sample accuracy statistics across benchmarks for a representative training threshold of $\phi=15$. The leftmost bar in each set indicates the misprediction rate of our baseline Jiménez and Lin-style perceptron predictor.

Pseudotagging. In Figure 11, the bars show prediction accuracy (assuming immediate update of prediction tables) as we increase the number of PC tag bits learned by the perceptrons. These data show that learning a few PC bits may help performance, but we find that as the number of PC bits increases relative to the number of BHR bits used, accuracy plummets. For most benchmarks, there is a threshold number of BHR bits that the perceptrons require for accuracy; learning a few PC bits beyond this threshold helps a little, but learning more PC bits does not further improve performance [?]. Figure 12 shows this for 176.gcc: the optimal number of PC bits to use as perceptron inputs always requires 21BHR input bits for perceptrons with more than 21 weights. The x axis shows how accuracy changes as the ratio of PC to BHR bits varies for perceptrons with eight-

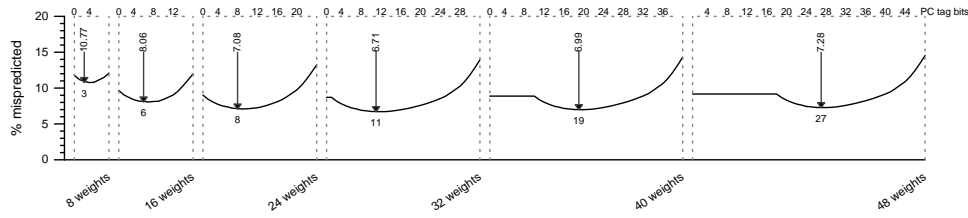


Figure 12. Accuracy for Pseudotagging and Differing Perceptron Widths, for 176.gcc

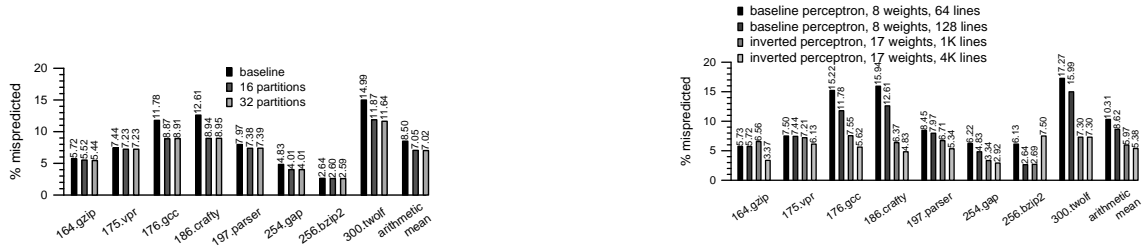
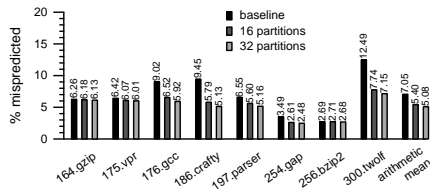
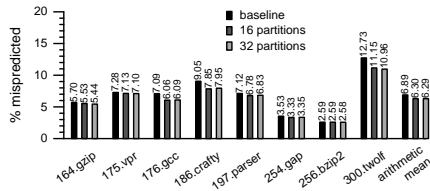


Figure 14. Accuracy for Inverted Perceptron

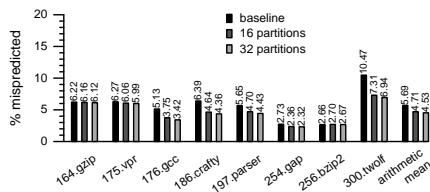
(a) 128-Line WT, 8 input bits



(b) 128-line WT, 24 input bits



(c) 1K-Line WT, 8 input bits



(d) 1K-Line WT, 24 input bits

Figure 13. Accuracy for Partitioning

48 weights. The arrows show the position and value of the optimal number of PC bits to learn. The eight-weight experiments show clearly that sacrificing BHR bits in favor of PC tag bits degrades performance.

Figure 11(a)-(b) show accuracy results using 128 and 1K-line perceptrons with 23-bit inputs and one bias input bit. The baseline configuration devotes all of these 23 bits to the entries in the global BHR, whereas the pseudotagged implementation uses four PC bits and 19 BHR bits. For every benchmark, the figure also shows performance obtained for the optimal mix of PC vs. BHR bits is chosen (the annotations under the bars indicate the optimal number of PC bits). These results indicate that pseudotagging effectively reduces destructive aliasing for shorter tables and for benchmarks with high numbers of static branches. As we increase the number of lines from 128 to 1K, aliasing in the WT decreases, and performance improvements from pseudotagging diminish. For benchmarks with many static branches (such as 176.gcc), pseudotagging decreases misprediction rates by as much as 1.94%. For other benchmarks, the benefits are less significant, although pseudotagging never hurts performance in our experiments. Results for other training thresholds are similar [?].

Partitioning. Figure 13 shows how accuracy varies with partitioning the input space and varying the numbers of perceptrons and weights. Partitioning is useful for predictors containing fewer perceptrons for all the numbers of weights studied (eight, 24, 40, and 64 in our studies); shorter tables give rise to linear inseparability due to aliasing, and partitioning effectively addresses this problem. Partitioning is still useful in the larger tables we study, but its effect on ac-

curacy is reduced. The graphs in Figure 13 compare results when all bits (for eight and 24-entry BHRs) are inputs to a single perceptron, when four bits choose among 16 partitions and 20 bits are inputs, and when five bits choose, and 19 bits are inputs. The results show that using part of the history information to partition the input space helps performance in all cases. The effects of partitioning are more significant at small hardware budgets, as these small tables suffer from nonlinearity caused by aliasing, in addition to the inherent nonlinearity of static, linearly inseparable branches. Partitioning can still improve performance when aliasing is not an issue, since it can “break” nonlinearities in the input space. The results for the 1K-line setting show this: despite increased table size, partitioned implementations still perform significantly better than the baseline. In contrast, the benefits gained from pseudotagging (which only addresses the aliasing problem) diminish at large table sizes. Results for other training thresholds are again similar.

Weight Caching. Figure 15 shows accuracy results for various WC configurations in conjunction with 1KB and 3KB WTs (128 lines \times eight weights; 128 \times 24 weights). Increasing associativity of the WC hides aliasing effects in the WT, decreasing the misprediction rate. For a given WC hardware budget, increasing associativity always helps more than increasing the number of lines. This is reflected in the mean misprediction rates for the WC configurations: a 1K-line, 4-way associative WC performs better than the 2K-line, 2-way associative WC, which in turn outperforms the 4K-line, direct-mapped configuration. For benchmarks suffering serious aliasing problems, the weight cache can decrease the misprediction rate by as much as 5% (e.g., from 91 to 96% for 176.gcc). Compared to pseudotagging, weight caching is much more effective in alleviating aliasing effects without increasing access times. On the other hand, pseudotagging requires no additional storage, and may be more suitable for low-power designs.

Inverting and Pipelining. We compare a gshare predictor typically used as a baseline in other studies (citations needed here) to the original perceptron and to our improved perceptron-based predictors. Figure 14 compares accuracies for our inverted, pipelined perceptron predictor and a baseline global perceptron predictor for organizations that support single-cycle access times (but have differing hardware budgets — the global perceptron predictor is at a disadvantage, since its size is limited for this aggressive prediction latency). Comparisons against predictors with similar hardware budgets or against overriding predictors require cycle-accurate simulation, and are presented in Section 5.3. For the experiments depicted in Figure 14, we use a training threshold of 15, since it has generally performed well in our other explorations. The pipeline latency is eight cycles, and the number of inputs that this organization can take while maintaining the single-cycle prediction latency is 17, and

the resulting WT is 34KB. The original predictor is limited to eight inputs, and uses a WT of 512B. Changing *what* the perceptrons learn and pipelining accesses to the WT thus allows for a weight table that is over a factor of 66 larger in terms of overall storage. The original predictor holds 64 perceptrons, whereas the pipelined predictor holds 2000.

Figure 14 compares accuracy of the inverted WT perceptron to accuracy of the best single-cycle perceptron configuration at 833MHz. Since the table access is pipelined and completed in advance, the inverted WT perceptron can afford to spend one full clock cycle on the weighted sum calculation (although our future work includes investigating methods to speed this calculation), and hence has a 17-bit input vector. A conservative eight-cycle access time is assumed for the pipelined WT access, and hence the 17-bit input vector is formed by alloying the eight lowest BHR entries to nine PC bits. Our results indicate that accuracy of this pipelined implementation can be up to 10% higher than that of the best single-cycle, traditional perceptron.

5.3 Performance Results

We choose design points based on analyzing hardware budgets that support the same access times. We investigated a single-cycle, original perceptron (intended to be a baseline candidate), but this configuration performs poorly, and so we choose the best two-cycle overriding configuration as our point of comparison. This baseline has 128-lines and 24-weight (for a hardware budget of 3KB). We scale the latency to four cycles for the 20-stage pipeline model. For the weight caching and partitioning experiments reported here, we use a single-cycle WT with 64 lines and nine weights.

Figure 16 and Figure 17 show the IPC results for our moderate and aggressive clock-rate experiments, respectively. We include results for gshare, bimode, and the Alpha 21264 hybrid branch predictors with similar 3KB hardware budgets for reference. At both clock speeds, we simulate WTs with 64 lines and nine weights per line, backed up by 1024-line, four-way associative WCs. We keep two-byte partial tags per line, yielding a 45KB hardware budget.

The latencies of the WTs are modeled to be one and two cycles, and the latencies of the WCs are conservatively modeled as eight and 10 cycles at our moderate (Figure 16) and aggressive (Figure 17) clock-rate experiments, respectively. For partitioning, we consider 16 partitioned WTs, each with 64 lines and nine weights per line. For the pipelined perceptron with inverted WT, we implement a 2K-line inverted WT with 17 weights per line. We model an eight-cycle delay for the table access, and start the WT reads nine cycles in advance, leaving two cycles for the 17-bit weighted sum calculation at aggressive clock speeds.

The results indicate that at moderate clock rates, our optimizations can increase performance by up to 9.28%, while

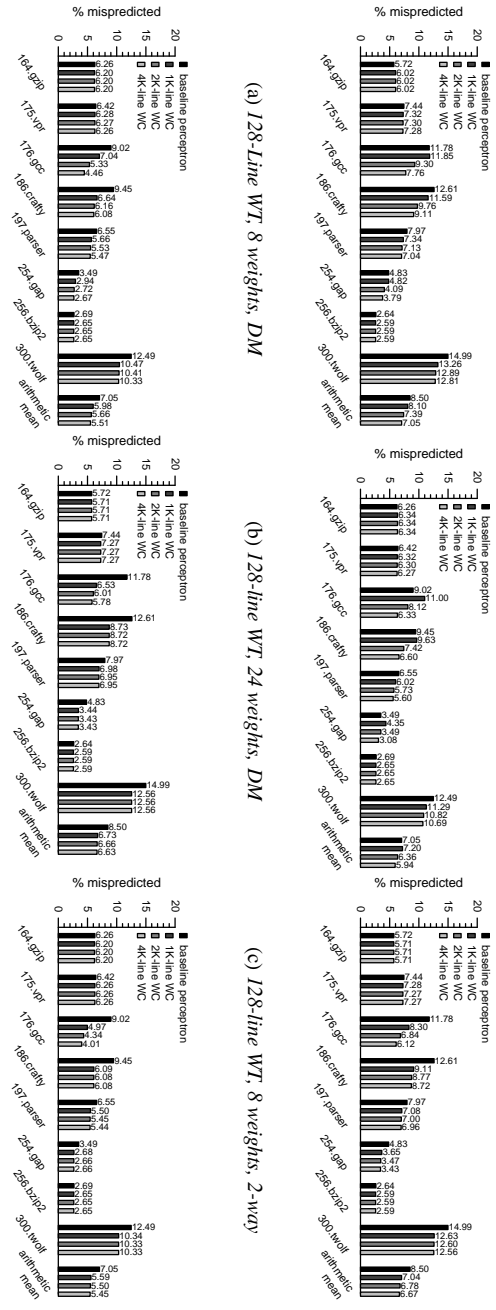


Figure 15. Accuracy for Weight Caching

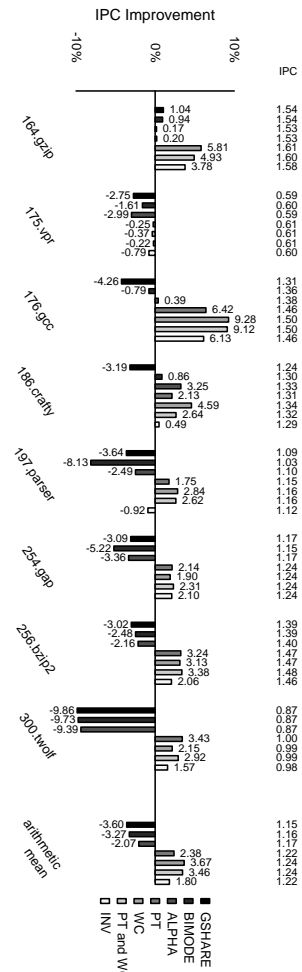


Figure 16. Comparative IPC Performance Effects

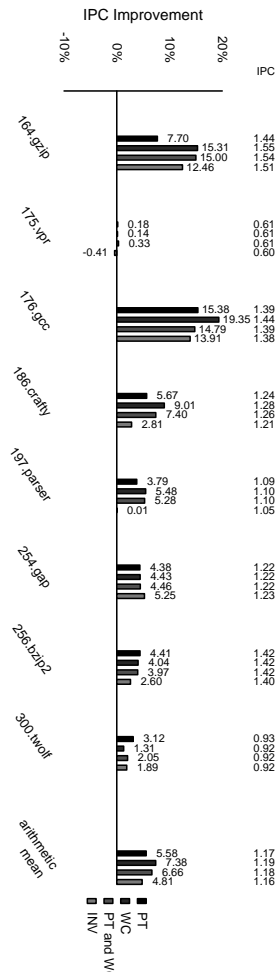


Figure 17. Comparative IPC Performance Effects for 20 stage pipeline

hurting the performance of only one benchmark (175.vpr) by at most 0.75%, with mean performance improvements of 1.8-3.67%. This is possible because we use much smaller weight tables that can be accessed in a single cycle, and we overcome nonlinearity problems usually seen at small hardware budgets by using caching and partitioning. In the case of the inverted WT perceptron, single-cycle access is achieved through pipelining, thereby maintaining accuracies comparable to the original, non-inverted perceptron.

For our aggressive clock rate simulations, we scale our WC and partitioned WT implementations to two clock cycles, keeping the same hardware configurations from the moderate clock rate simulations. At a clock rate of 1.76 GHz, we model a 20-cycle branch misprediction penalty representative of modern, deeply pipelined superscalar machines. At this configuration, IPC improves up to 19.35% over the baseline, with mean improvements between 4.74% and 7.29% for the various optimizations.

To summarize our findings, the new single-cycle predictors decrease misprediction rates by 1.37-11.11%, which amounts to improvements of 18.2-69.7% over the baseline perceptron-based predictor. The corresponding decrease in arithmetic mean is 3.24% — an improvement of 37.6%. The best two-cycle, overriding predictor decreases misprediction rates by 0.02% (i.e, we never hurt accuracy) to 5.1%, decreasing arithmetic mean by 1.78%. The equivalent accuracy improvements are 0.9-55.5%, with a mean of 24.1%. Our performance experiments show IPC changes by -0.79-9.28%, with a mean of 3.67% for the baseline SimpleScalar 4.0 configuration. Note that 175.vpr is memory-limited, and thus branch prediction is not its limiting performance factor. This is the only benchmark for which our predictors cause negative (albeit very small) performance effects. For the aggressively clocked, 20-stage microprocessor model, our best optimization — weight caching with a 1K-line WC and 4-way set associativity — improves IPC by -0.41% (again, for 175.vpr) to 19.35%, with a mean of 7.29%.

6 Conclusions and Future Work

Current work seeks to derive limits to perceptron-based approaches by exploring performance of the best online linear classifier for the problem of branch prediction. We are investigating profile-directed feedback optimizations for choosing the best partitioning scheme to minimize the number of linearly inseparable branches; these offline algorithms are based on binary decision trees. With respect to predictor implementation, we continue to study techniques to help reduce or cope with nonlinearity, and we study variations on the pipelined perceptron approach (in particular, using earlier branches to predict the outcomes of later — and possibly different — static branches).

In this paper we have studied cases that limit the per-

formance of perceptron-based branch predictors. From this analysis, we have proposed a set of optimized perceptron-based branch predictors and have investigated their performance. We find that our predictors yield increased accuracies that translate into performance improvements of up to 14.14% over the Alpha 21264 predictor, and up to 9.28% over the original perceptron (with means of 5.9% and 3.67%, respectively). For a more aggressively clocked processor model and a 20-stage pipeline, our designs improve IPCs by up to 19.35% over the original perceptron.