Exploring Perceptrons in Branch Prediction

Nick Deibel **Kevin Sikorski**

Abstract

Branch prediction has become a more important component of the modern processor, due to longer pipelines and the desire to exploit more instruction-level parallelism. Perceptrons, well-understood tools in machine learning, have recently been applied to this area of research with excellent results. We use this investigation to show tracking local and global branch histories can improve perceptrons' performance. We also explore the effects of using various caching mechanisms to combat aliasing.

1 Introduction

Over the past few years, branch prediction has become an increasingly important feature in the modern microprocessor. This is largely due to the ever-growing emphasis on processor speed: as pipeline lengths increase, more CPU time is wasted on a branch mispredict. There is also the desire to exploit instruction-level parallelism: if a branch predictor can tell the CPU its confidence level for a particular prediction, the CPU can better decide if it should speculatively execute both branch outcomes (low confidence), or simply execute the predicted outcome, freeing functional units for use on other tasks.

There are two main types of branch prediction. *Static* prediction issues its decision based on some stationary set of criteria. A particular instruction placed at a particular memory address will always generate the same guess under static prediction. Common examples are "always-taken" (all branches are assumed to be always taken), "backwards-taken" (backward-pointing branches are assumed to be always taken), and other strategies where some instructions are assumed always taken, and other instructions are assumed always not taken.

Dynamic prediction often produces superior results to static prediction by using simple statistical methods to decide if a branch is likely to be taken. Examples are "same-as-last" (branch is predicted taken if and only if it was taken last time), gshare and other SUD counter-based approaches, and perceptrons. The improved performance over static prediction is due to this class's ability to adapt to the particular quirks of the code that is being executed. However, the increased performance also comes at a cost of hardware complexity.

It has been recently shown that perceptrons can be applied to the branch prediction problem with excellent results. In past reports, perceptrons had only been used with global history of branch outcomes. In this investigation, we explore how perceptrons fare when taking both the global history as input, several bits of local history, and several bits of regional history. We will also examine the possible performance improvement of perceptron branch predictors via several caching techniques.

2 Background and Related Work

This section provides a brief introduction to perceptrons. For further details, we ask the read to consult [1] and [2]. A history of the application of perceptrons in branch predictors is also given.

2.1 Perceptrons

A perceptrons is a tool developed in Machine Learning as a simple model of a human neuron. The perceptron receives a set of input signals x, and multiplies each of the signals according to the perceptron's set of weights w. It then adds the results together, and outputs 1 if the sum is ≥ 0, and -1 otherwise. More formally, given a vector of inputs and a vector of weights, the perceptron's output is $\text{sgn}(x \cdot w)$. In practice, an extra input is often added, with its value tied to 1. This allows the perceptron to learn its activation threshold (i.e., how large the sum must be for it to "fire", and output a 1). For branch prediction, the input to the perceptron is usually the global history of branch outcomes, and semantically, the output is "taken" if the sum is ≥ 0, and "not taken" if the sum is <0. A sample perceptron is presented in figure 1.

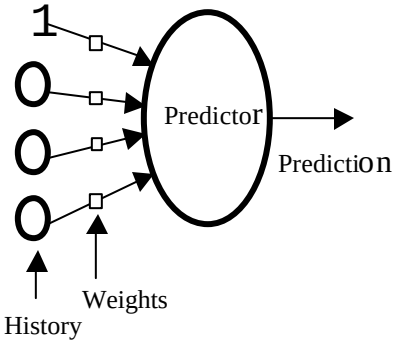


Figure 1: A Perceptron.

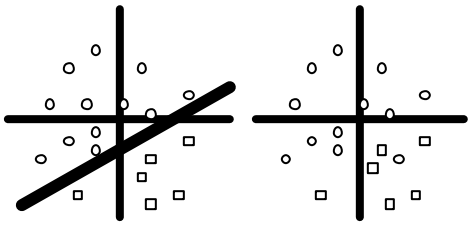


Figure 2: Linear Separability. The graph on the left is linearly separable by the indicated line. The graph on the right is not linearly separable.

Perceptrons are limited in what they can learn. From their structure, it is trivial to see that they can only learn partitions of the form $1 \cdot x_0 + A \cdot x_1 + B \cdot x_2 + \ldots + Zx_n = 0$. In other words, given n input signals, perceptrons can only classify sets that can be separated by an n-dimensional hyperplane. Such sets are otherwise known as *linearly separable* sets. Linear separability is illustrated above in figure 2. The basic boolean functions, AND, OR, and NOT are linearly separable and can thus be learned by a perceptron. A classic example of a linear inseparable function is XOR

It is important to note that while this constrains what a perceptron can learn with 100% accuracy, a perceptron can perform reasonably well on linearly inseparable data as well. For example, one could produce a line in figure 2 that would separate the second graph with the exception of a single data point - the weights the perceptron would learn for such an example would be correct for the vast majority of the examples it sees, assuming a uniform distribution of examples.

The method for training a perceptron itself is inherently simple. A perceptron's weights are typically stored as floating point numbers, and are trained according to the training rule $w_i \leftarrow w_i + \eta(t - o) \cdot x_i$, where η is the learning rate, *t* is the value the perceptron should have produced, and *o* is the value the perceptron actually produced [1]. For linearly inseparable functions, this method is guaranteed to approach an optimal weight vector in finite time given a small enough η. In practice, this convergence often occurs quite rapidly.

2.2 Previous Work

The simplicity of a perceptron is exactly what makes it well suited to the branch prediction problem. Given the short clock cycles that permeate the current state-of-the-art processors, very little computation can be done in the time allotted for prediction. The amount of specialized data (history, the states of counters, etc.) that can be accessed in this time is limited, too. Because of this, heavier-weight machine learning techniques such as decision trees, back-propagation neural networks, and nearest-neighbor are infeasible. However, perceptrons can be trained with a simple update rule, and can render decisions in roughly the time required to perform a few additions. They are also candidates for pipelining.

Perceptrons were introduced to the branch prediction arena by Jiménez and Lin [2], where they found that perceptrons are often more effective than gshare, a respected branch predictor in use today. They also produced a hybrid predictor that combined gshare and perceptrons, and often outperformed them both. Finally, they examined the effects of various history lengths, and found that perceptrons were able to gleam advantages from very long global histories, as apposed to gshare, which performs worse with longer histories.

One of the major contributions of their work was designing a realistic hardware implementation for a perceptron branch predictor (which we will refer to as a PBP). The greater computational demands of floating point arithmetic make the standard a training rule infeasible in a hardware implementation. Instead, Jiménez and Lin's approach for branch prediction used a faster strategy for updates and predictions where all weights are stored as integers. The update rule becomes $w_i \leftarrow w_i + t \cdot x_i$, where *t* is the actual outcome (1 for "branch taken", -1 for "branch not taken"). The weights are also capped, such that the magnitude of any weight is not allowed to exceed a parameter θ. It was found empirically in [2] that the optimal value to be $\theta = \lfloor 1.93h + 14 \rfloor$, where *h* is the length of the history list.

Michaud and Seznec produced an internal publication [4] that evaluated the effectiveness of the perceptron branch predictor. They found that including a few bits from the address of a branch as input to the perceptron showed an improvement, largely due to improving linear separability. Their approach was also shown to be more cost-efficient than adding more entries in the table of perceptrons. Similarly, the authors explored two-level predictors, where the output decision from one predictor is fed into the input of a perceptron, along with the global history. This also produces a more efficient predictor. Finally, they corroborated results from [2], finding that certain branches were better predicted by classical predictors than by perceptrons.

Agrawal and Woo investigated perceptrons as part of a class project at CMU, [1]. Their contribution was a simple "patch" that on one benchmark provided a 49% improvement over classical perceptron performance. The patch consisted of tracking the number of branches executed in total, and the number of branches since the last misprediction. On each misprediction, the number of branches executed and the number of branches since the last misprediction would be stored. When it came time to predict an outcome, the perceptron would make its decision, and if the gap between this prediction and the last misprediction was the same as the last misprediction and the one before that, then the decision would be flipped. They continue to say that this might just be a quirk of the particular benchmark. They also observed that using a 2-bit saturating counter to control decision flips might improve performance.

3 Implementation

In this section, we explain our choice of implementing a PBP in the architecture simulator, sim-alpha. This implementation is markedly different from previous work and has consequences on our experimental results.

3.1 Previous Implementations

It is important to note that the researchers in [2] and [1] did not actually run benchmarks with an inline perceptron branch predictor. Instead, they did a program trace of their benchmarks, recording each branch and its associated outcome. Once this data was collected, they ran their perceptrons on the extracted branches and outcomes. While this approach has the advantage of being faster than an architecture simulator, it fails to accurately portray the working environment of a predictor.

When a branch prediction is made, it will be several cycles until it is know if the prediction was correct. During this interval, the CPU might come across another branch instruction. Since stalling till the first prediction becomes known is unacceptable, a speculative prediction is made about this second branch. While this is good in terms of promoting throughput, speculative prediction creates an interesting problem for stateful predictors (predictors that keep a history). Speculative prediction will lead to mistakes in the history, effectively making it "garbage."

To understand this, consider the following scenario. A branch X is reached and is predicted as taken. Because it is dependent on a high-latency function (perhaps a floating-point multiply), the actual outcome will not be known for 40 cycles. Along the way, Branch Y is encountered, predicted, and is determined to be not taken all within this 40 cycle period. When the not taken outcome of Y becomes known, the global history is updated. This is the heart of the problem. Assume X was supposed to be not taken. Then, Y should never have been encountered and its history bit should not be in the register. On the other hand, assume X is taken. While Y will be encountered, X's history bit should come before Y's, but in reality, they will be reversed since Y's outcome became known first.

This situation is not reflected in the trace implementation of [1] and [2]. Because the perceptron is learning on-line, it will very likely make several bad predictions early on. Each time a misprediction is made, the global history risks contamination by branches that were speculatively executed, but were flushed out of the pipeline later. This means that the previous studies only examined how well a PBP could learn how to predict branches by watching another predictor work, instead of how well a it could predict branches. The noise created by these speculated branches and garbage history could perturb the results in either direction. However at the quoted prediction rates (90-98%), even one percentage point would be a significant difference.

3.2 Sim-alpha Implementation

By implementing a PBP in sim-alpha, any performance data collected should reflect the effect of speculative predictions. It is our belief that this

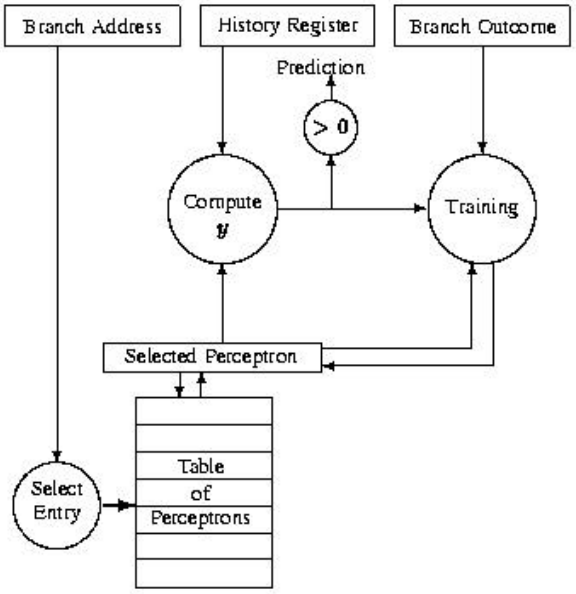
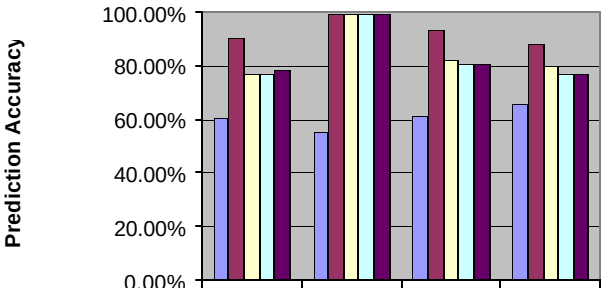


Figure 3. A hardware model for a PBP, or perceptron branch predictor. From [2].



	TWOLF	LUCAS	GCC	VPR
Always Taken	60.28%	54.91%	60.93%	66.00%
21264	90.24%	99.27%	93.25%	88.01%
Gshare (8)	76.63%	99.06%	81.75%	79.93%
Gshare (10)	76.63%	98.95%	80.80%	76.80%
Gshare (16)	78.57%	98.95%	80.80%	76.80%

Table 1. Performance of other predictors on the benchmarks.

will give a more accurate representation of the prediction accuracy.

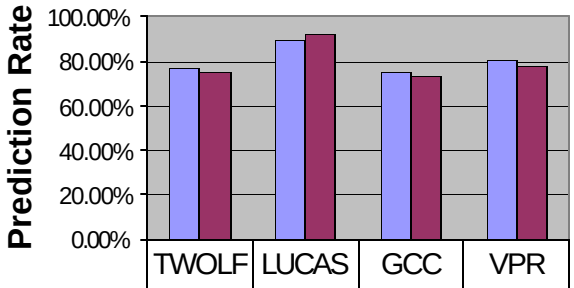
Sim-alpha also comes with several built-in branch predictors, including always-taken, gshare, and an approximation to the Alpha 21264 predictor. Without having to code on our own versions of these, we are provided with an easy means of doing a comparison evaluation among different providers.

We have successfully implemented the perceptron branch predictor hardware model from [2] into sim-alpha. Within software, we simulate the following hardware actions (as seen in figure 3):

1. Hash the branch address to get an index into a table of perceptrons.
2. Fetch the appropriate perceptron.
3. Compute the branch prediction.
4. Act on the prediction (taken if ≥ 0).
5. Train the given perceptron on the outcome.
6. Write the trained perceptron back to table.

In our simulation, the weights for all perceptrons are initially set to 0. The only exception is the always 1 input. Its weight is set to 1 initially to bias the perceptron into always-taking the branch for at least the first time a perceptron is consulted. Other approaches to setting this w_0 weight can be considered, including 1 if the branch is backwards and 0 otherwise. This is equivalent to the "always-backward, never-forward" heuristic.

All other extensions to the PBP presented in this paper have also been implemented into sim-alpha. Eventually, this code artifact will be made available to others for future research.



	TWOLF	LUCAS	GCC	VPR
512	76.84%	89.66%	75.06%	80.17%
128	75.57%	92.50%	73.21%	78.09%

Table 2. Effect of perceptron table size on prediction accuracy. Each PBP uses a 20 bit global history and a global weight threshold of 52.

3.3 Testing Approach

To test the performance of our PBP's, we chose four of the programs from the SPEC2000 benchmark suite: lucas from CFP2000 and gcc, twolf, and vpr from CINT2000. These programs were chosen for their relatively fast runtimes. Similarly, due to time constraints, we ran these programs using the *test* inputs, instead of the more research-oriented *ref* inputs.

For comparison, we ran the same benchmarks on sim-alpha using five other predictors:

1. Always-taken predictor
2. Alpha 21264 predictor
3. Gshare with history lengths of 8, 10, and 16

The always-taken and Alpha 21264 predictors provided good lower and upper bounds on the prediction accuracy. Our main goal was for the PBP's to perform as well or better than the three gshare predictors. Table 1 shows the performance of these five predictors on the four benchmarks.

4 General Performance

In this section, we discuss the general performance of the perceptron branch predictors compared to the other branch predictors. We also consider the hardware cost of a PBP.

4.1 Comparison of Performances

Table 2 contains the performance of two PBP's on the benchmarks. Both use 20 bits global histories and a weight threshold of 52. The only difference is the number of perceptrons, 512 versus 128. It is difficult to make a direct comparison between our results and [2] do to the use of different benchmarks and history lengths. However, it appears that our

PBP's fare significantly worse than those of Jiménez and Lin. For example, on gcc, their predictors average about 92%. This poorer performance is not surprising given our different implementation. We believe that a majority of this degradation is due to the effects of speculative prediction.

Comparing this table to Table 1 and 2 shows that our basic perceptron branch predictors are performing only moderately well. On twolf and vpr, the PBP's are performing the same as the gshare predictors. On lucas and gcc, however, gshare is performing significantly better. In later sections, we will show ways of significantly improving the performance of the PBP's so that they perform comparably to gshare.

4.2 Perceptron Table Size

For many stateful branch predictors, one of the major sources for prediction inaccuracies is when two branches hash to the same predictor component (in our case, the same perceptron). This effect, known as aliasing, can cause contradictory learning in the perceptrons. One branch will say to increase a weight, while another one will say to decrease that same weight. Another possibility is that one weight gets increased too much, causing erroneous predictions with seemingly high confidence.

One solution to reduce aliasing is to reduce the number of collisions that occur in the hash table. A simple approach entails just increasing the hash table size. Table 2 shows the performance of two different PBP's on the four benchmarks. The first PBP uses 512 perceptrons, while the second uses only 128 perceptrons. For three out of the four benchmarks, the 128 perceptrons performed slightly worse than its counterpart with 512 perceptrons. The exception with the lucas benchmark is an interesting anomaly. Even though more aliasing is occurring due to the smaller table, the branches hashing to the same slot are not mutually destructive. The training induced by one is beneficial to the other.
Unfortunately, this is a unique situation. Consistently across other configurations, larger table sizes give higher prediction rates for all four benchmarks. However, the overall loss in performance is usually less than 3%.

Simply increasing the table size does not come without a price, though. Unlike gshare and other predictors who use small saturating up-down counters, adding one perceptron requires a much larger amount of hardware. In general, a PBP's hardware cost is high; it requires storage for the weights of the perceptron and specialized functional units to perform the predictions and training. With an attempt to balance the amount of hardware used between gshare and our PBP's, we will only consider using a perceptron table of size 128 for the remainder of this paper.

5 Local History

Global history is limited in its ability to provide information about branching history. For instance, consider a large loop. If more branches occur within this loop than there are bits in the global history, each time the loop condition is reached, no information will exist explicitly about that particular branch. To combat this situation, we propose keeping local branch histories to be used along with global history in branch prediction.

5.1 The Idea of Local History

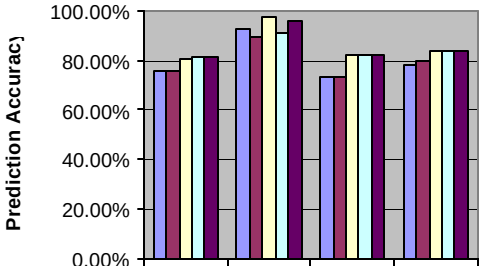
Local history is a record of past outcomes for a particular branch. This supplements global history in several ways. One, it guarantees that there will always be some information about the branch's history. The global history might not contain any bits relevant to the branch in question, but the local history obviously will. Two, local history is not a replacement to global history. Consider a local history of 1111. This will certainly weight the prediction towards taking the branch. However, previous branches can also influence this decision. It might very well be the case that if the global history is 10100 and the local history is 1111, the branch should not be taken, but should be taken any for any other global history. The combination of these two histories allows for this finer-grain prediction.

For each branch, we would ideally like to keep track of its last *m* outcomes. This is impossible, though, since this would require memory for every possible instruction address. As an alternative, we keep a local history for each perceptron. This local history will not be unique to a particular branch, but will still represent a small subset of all the branches. This will not be as fine-grain as we could hope for, but it is more feasible to realize in hardware.

Adding local history does not come cheap, though. For each perceptron, we have to add a shift register. We also have to increase the number of weights. This can potentially slowdown the prediction and training calculations. One has to carefully decide how much local history to add, but this hardware expense does pay off in terms of increased accuracy.

5.2 Performance

Before analyzing the performance gain of using local history, the issue of how to handle θ should be



	TWOLF	LUCAS	GCC	VPR
(0, 20)	75.57%	92.50%	73.21%	78.09%
(0, 25)	75.79%	89.36%	73.29%	79.49%
(5, 20)	80.81%	97.27%	82.15%	83.43%
(10, 20)	81.33%	90.92%	82.29%	83.45%
(20, 20)	81.53%	95.73%	82.12%	83.45%

Table 3. Effect of Local History.
The numbers inside the parentheses represent the size of the local and global histories, respectively. Separate weight thresholds are used.

mentioned. Originally, to accommodate local history, we extended the weight threshold formula to $\theta = \lfloor 1.93(h_L + h_G) + 14 \rfloor$, where h_L is the local history size and h_G is the global history size. Surprisingly, this produced a noticeable drop in performance. We then found that using two weight thresholds, one for local history inputs and one for global history inputs, performed significantly better. While a formula $\theta = \lfloor ah_L + bh_G + c \rfloor$ would be interesting to find, such a task was beyond the scope of this project.

Table 3 shows the prediction accuracy for several configurations that use local history (and the double threshold approach). The second and third data sets are of particular interest. Adding five global history bits only improves (from the first series) the accuracy by 0.22%, while adding five local history bits improves the accuracy by 5.24%. Furthermore, with the addition of this small local history, we are not performing better than gshare on every benchmark except lucas. In this exception, the difference in accuracy is only less than two percent.

The last two data series, which use 10 and 20 bit local histories, reveal that throwing more local history at the predictor will not always dramatically improve performance. It should be noted that without more data, it is difficult to fully describe the extent of these diminishing returns. In particular, the accuracy for the lucas benchmark drops dramatically with 10 local history bits, but improves with 20 bits.

This data clearly suggests that using local history will improve PBP performance. A more thorough study for finding an optimal local history length is warranted, but beyond the scope of this project. We believe that this value will be around 10 bits, and for the remainder of the data runs in this paper, we will use a local history of length 10 and a global history of length 20.

6 Set Associative PBP's

Most table-based branch prediction schemes face the problem of aliasing when multiple branches in program memory are mapped to the same entry in the predictor table. In these cases, training a predictor on the results of one branch will affect the performance of that predictor for each branch that is mapped to that predictor. Direct-mapped memory caches exhibit a similar problem: multiple memory locations map to the same entry in the cache, and because only one word of memory can be present in a single entry at a time, extra cache misses result.

Caches avoid this problem by exchanging the direct-mapped format for an n-way set-associative format, where multiple memory locations map to a set of n entries in the cache, and the requested memory location can be in any of those n entries. This approach has the advantage that if multiple memory locations mapping to the same entry are used heavily, more of these locations can be stored in the cache. There are a few disadvantages, including increased power consumption and larger area, and as always, these must be considered when examining trade-offs.

We can extend the concept of set-associativity to the branch prediction domain. Instead of using a hash function of the branch address to index into a direct-mapped table of perceptrons, we can instead use the hash function to index into a 4-way set-associative table of perceptrons. In this manner, if we have several branches mapping to the same entry in the table, we can store several in the table at once, and avoid "poisoning" the performance of one branch by training it with data from another branch.

As with caching, the use of set-associativity in branch predictors has drawbacks: power consumption is still a concern, and many extra bits are needed for bookkeeping (Least-recently-used bits, address tags). To fully evaluate the trade-off, investigations must made to determine if the extra bits are better used for the aforementioned bookkeeping, or to increase the size of the table.

We also note that this approach is not limited to perceptron predictors for any reason – set associativity can just as easily be applied to other

table-based prediction schemes such as saturating up-down counters, gshare, and hybrid predictors.

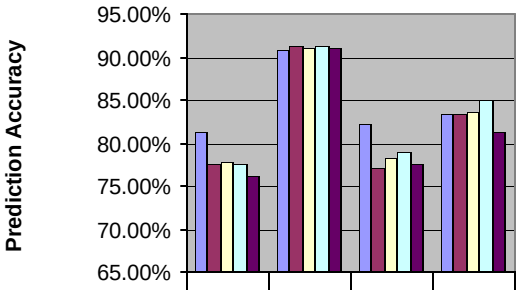
6.1 Set Associativity in PBP's

Implementation-wise, this approach is nearly identical to that used in caches. For all experiments conducted in this investigation, we used 4-way set-associativity, and proportionately scaled down the size of the table to equalize the number of perceptrons (instead of a 128-perceptron direct-mapped table, we now have a 32-set, 4-way set-associative table). Each entry in the table is now a set of 4 perceptrons, plus 2 bits of LRU data for each perceptron, as well as sufficient bits for storing the addresses of the branches that is predicted by each perceptron. When it comes time to index into the table, we compute the hash, and use it to find the set of entries in which the branch may be. We then compare the address of the current branch with the address tag of all the entries in the set. If there is a match, then we use the matching perceptron to predict the branch, and later for training when the actual outcome of the branch is resolved.

If there is no match within the set, then we must select an old perceptron to remove from the set, and load a "new" perceptron. We use LRU replacement to identify the old perceptron. Then, we have a choice of four strategies for how to initialize the new perceptron – we can set all the weights and local history to zero (which we refer to as ZERO replacement); set all the weights and local history to zero except for w_0, which is set to one (ZEROONE replacement); or do nothing and simply leave the weights and local history from the old perceptron (NONE replacement); or load a "common" perceptron that represents a sort of average of the perceptrons predicting the branches in the set (COMMON replacement). We discuss COMMON replacement in more detail later.

There are arguments for each of these strategies. ZERO replacement may be effective because there is no way to predict if the weights of a perceptron will be positive or negative, and presetting them to zero leaves them at a middle ground. ZEROONE replacement started off as a miscommunication within the team, however its results hint at a possible future replacement strategy, discussed later. NONE replacement is the simplest to implement in hardware, and is the strategy employed in previous work.

COMMON replacement is significantly more complex than the other strategies. The idea is maintain a "common" perceptron that tries to give the new perceptron a head start in converging to its final weights. To do this, we use 4-way set-associativity, but only use 3 entries in the set in the conventional manner. The fourth entry in each set is dedicated to storing the common perceptron. Any time a perceptron in a set is trained, we also train the common perceptron. Thus, the common perceptron is never used for predictions, but will always have some information about all the branches that map to the set. When it comes time to perform COMMON replacement, we simply copy the weights and local history from this perceptron into the destination entry in the set. There is a considerable trade-off in using this approach – the number of perceptrons usable for branch prediction is cut by 25%. While this means the likelihood of aliasing sharply increases when this strategy is used, the hope is that the "head-start" given by the common perceptron speeds convergence enough that this becomes less of an issue.



	TWOLF	LUCAS	GCC	VPR
DIRECTMAP	81.33%	90.92%	82.29%	83.45%
NONE	77.46%	91.21%	77.10%	83.35%
ZERO	77.84%	90.94%	78.30%	83.63%
ZERONONE	77.60%	91.24%	79.02%	85.04%
COMMON	76.23%	91.00%	77.61%	81.33%

Table 4. Effects of set-associativity.
All PBP's used 128 perceptrons with a 10-bit local history and 20-bit global history. The DIRECTMAP does not use set-associativity.

6.2 Set Associativity Performance

Set-associative branch prediction tables do not appear to gleam additional performance compared to direct-mapped tables. In fact, the prediction accuracies in Table 4 shows that this decreases performance almost across the board. The only exception is a slight improvement for the lucas benchmark. These results are somewhat surprising. After all, caches must derive some benefit, otherwise set-associativity would not be as prevalent as it is today. It is unclear

why this does not hold in the perceptron branch prediction domain. We theorize that the particular benchmarks we selected, or perhaps benchmarks in general, have unusually non-linearly-separable or non-uniform distributions of branch instructions throughout the table, with most branches mapping to the same set. Set-associativity can mitigate some non-uniformity, but it cannot stand against more severe distributions.

There is also a significant difference in the performance of replacement strategies. ZERO and ZEROONE perform the best. This is most likely because the perceptron is reweighted back to zero, and it's local history "cleared" (set to all not-taken), which allows the weights to quickly change between positive and negative values. The slightly better performance from ZEROONE leads to a possible performance improvement we will discuss in Section 8.3.

NONE replacement performs moderately well. However, it appears to take extra time to overcome any "inertia" left from the previous perceptron in the same entry. This causes a slowing of convergence and thus, performance is hindered. COMMON replacement performs worse. This is likely because we are devoting 25% of all perceptrons to the "common" perceptrons. Also, because perceptrons only encode linearly separable functions, it is difficult for the common perceptrons to overcome the extra aliasing that occurs during their training.

7 Victim Caching

Victim caching is another technique originally developed for use in memory systems that is can be easily adapted for use in branch predictors. Instead of outright discarding an entry that is already in the cache but slated for replacement, the entry is placed in a small, fully-associative victim cache. This effectively allows the cache system to dynamically allocate more entries in more frequently used sets. The victim cache is kept small enough to allow probes to be performed quickly and in parallel with the L1 cache look-up, and not increase L1 hit delays.

Just like set-associativity, this approach is not limited to use in perceptron predictors, and may be useful in other predictors despite its greater hardware requirements.

7.1 Victim Caching and PBP's

We implement Victim Caches much as they are implemented in memory systems. The cache is placed just behind the main perceptron table. All entries in the main perceptron table must be augmented with address bits (indicating the address of the branch instruction that is being modeled with that entry). While this does not pose a problem for implementations that are already set-associative, this means using many extra bits to store address data in a direct-mapped table. As always, there is a trade-off in whether or not the extra address bits needed in a direct-mapped, victim-cached implementation would be better spent as extra perceptrons in a slightly larger, but not victim-cached table.

The cache comes into use when we index into the perceptron table and find that the requested perceptron is not there, determined by comparing the entry's address bits with the branch instruction's address. Then we check the victim cache: if none of the perceptrons in the cache match the branch instruction's address (a victim cache miss), then we identify the least-recently-used entry in the table, and save it into the least-recently-used entry in the victim cache. We then execute one of the replacement strategies listed in section 6.1 to find an appropriate start perceptron, and store it in the table.

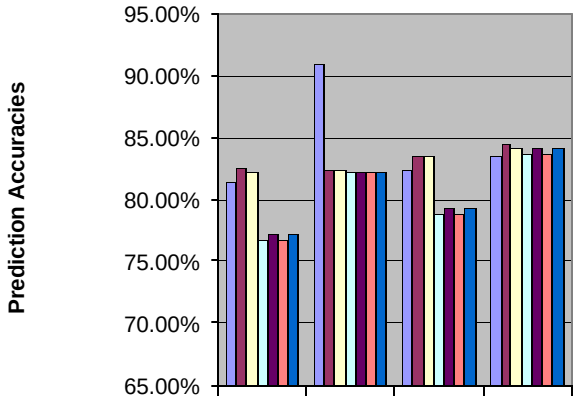
If instead we find the branch instruction's address in the victim cache (a victim cache hit), we can simply swap the victim cache's copy of the requested perceptron with the least-recently-used entry in the table's set. This way, we load the requested perceptron, and store the old perceptron into the cache.

This approach significantly increases the complexity of direct-mapped caches. In addition to requiring the use of address bits for all entries, it also mandates the use of the replacement strategies described in section 6.1 in the case that a perceptron is not found in either the table or the cache. In this case, we can use the ZERO, ZEROONE, or NONE replacement strategies, but not the COMMON replacement strategies – there is no room for storing a common perceptron in the direct-mapped case, though one could use a 2-way set-associative table and COMMON replacement to get the same effect.

7.2 Victim Cache Performance

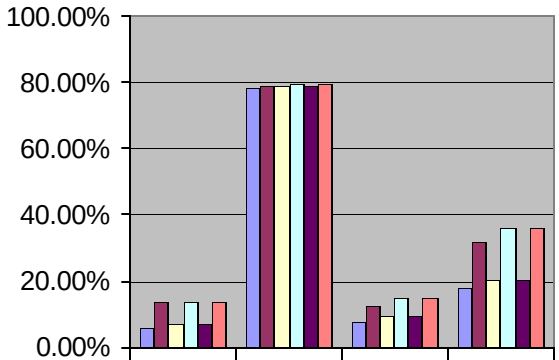
One of the most heartening aspects of implementing a victim cache is that (in theory) it always increases performance, as long is it doesn't increase the amount of time a branch prediction will take. Similarly, a large victim cache is (in theory) always better than a small one, given the same constraints.

Table 5 illustrates the effects of victim caches on direct-mapped caches. The replacement strategies have a significant effect on performance, and



	TWOLF	LUCAS	GCC	VPR
DIRECTMAP	81.33%	90.92%	82.29%	83.45%
5-NONE	82.45%	82.29%	83.54%	84.40%
10-NONE	82.14%	82.29%	83.54%	84.21%
5-ZERO	76.72%	82.26%	78.77%	83.65%
10-ZERO	77.18%	82.26%	79.37%	84.14%
5-ZERONONE	76.73%	82.26%	78.78%	83.71%
10-ZERONONE	77.18%	82.26%	79.38%	84.19%

Table 5. Effects of victim-caching on direct-mapped PBP's. The numbers in front of the different replacement strategies represents the size of the victim caches.



	TWOLF	LUCAS	GCC	VPR
5-NONE	6.07%	78.24%	7.90%	17.57%
10-NONE	13.76%	78.77%	12.68%	31.99%
5-ZERO	6.80%	79.17%	9.40%	20.18%
10-ZERO	13.87%	79.62%	14.60%	35.88%
5-ZERONONE	6.87%	79.19%	9.40%	20.26%
10-ZERONONE	13.76%	79.62%	14.61%	35.99%

Table 6. Victim Cache Hit-Rates for direct-mapped PBP's.

surprisingly, the best strategy is different for a direct-mapped and a set-associative table. For direct-mapped perceptron tables, NONE replacement performs best. We theorize this is because several branches map to the same entry, helping to prevent overtraining, and allowing the perceptron to encode some information about all of the branches at once. ZERO and ZEROONE replacement seems to work better for associative tables in the presence of a victim cache. For these tables, using NONE replacement probably leads to overtraining, because while we have more branches mapped to the set as a whole, they are still spread out over the entire set. The inertia left by this overtraining slows convergence, so it is best to wipe it out entirely using ZERO or ZEROONE replacement.

As before, it is difficult to say if table set-associativity is useful in branch prediction. The answer to this question does not become any clearer when victim caching is added.

The LUCAS benchmark exhibits a couple of strange qualities: 5-entry victim caches do slightly better than their 10-entry counterparts, and the non-victim-cached, direct-mapped table performs better than any other configuration. It is not clear why this would happen. Executing a wider variety of benchmarks may provide some insight as to what it going on. Other than for this one benchmark, victim caches using the NONE replacement strategy appear to be a modest improvement for branch predictors.

Table 6 depicts the victim cache hit rates, which provides another indication of how well the cache is performing. Due to the wide variation in hit rates, it is very difficult to draw any useful conclusions from this data. Clearly, more benchmarks must be run. However, one can see that for direct-mapped victim caches, the ZERO and ZEROONE replacement strategies yield a higher hit rate than NONE replacement. This is unexpected, as these configurations did not give better branch prediction performance compared to NONE. The effects of speculation due to mispredicted branches could be one reason for this odd behavior.

8 Future Work

In the previous sections, we have occasionally noted possible directions for future research on perceptron branch predictors. In this section, we will briefly describe these and other topics of interest.

8.1 Further Empirical Testing

Having considered only four of the SPEC2000 benchmarks, we do not have a strong representation of all branches. The data points that we have called odd might truly be odd, or they might actually occur

frequently. Thus, running the PBP's on other benchmarks (with possibly the *ref* inputs) seems prudent.

Along the same lines, we would like to run many more configurations for our PBP's. We would like to determine empirically good local history lengths as well as determine good weight thresholds. Along the same lines, more table sizes for direct-mapped and set-associative designs should be looked at. The size of the victim cache should be explored in more detail. All of these will work towards defining a good PBP configuration.

Finally, we note that when collecting data, we sometimes noticed eccentric behavior from sim-alpha. Sometimes, we were able to track this down to a bug in our code. Other times, however, we were not able to identify the cause of these oddities. While we are willing to admit that flaws might exist in our perceptron code, we do suspect that some bugs exist in sim-alpha. Through further testing, we hope to the correct source of these errors.

8.2 Further Caching Options

We have already taken some ideas from caching and applied them to our PBP's. Another possible application of caching is to create an L2 cache of perceptrons. Currently, our perceptron table is an L1 cache. Like with our set-associative example, we associate each perceptron with a particular branch. When a perceptron is removed from the cache, we could store it in an L2 cache. If we encounter the branch again, we can load this perceptron from the L2 cache back into the L1 cache.

In order to make this realistic, though, we cannot expect to load the perceptron from the L2 cache and use it to make a prediction within 2-3 cycles. Instead, we should use a common perceptron or some other means (perhaps always-take the branch) for the prediction. The idea is that the perceptron will be available in the L1 cache for later predictions on the branch.

Like with all our suggestions, a thorough study is needed to see if the hardware and power used by this L2 cache could be better used in making more perceptrons or adding more history bits. Furthermore, it is entirely possible that cache thrashing would often occur. It is conceivable for a perceptron to constantly be shuffled between the two caches but never being used to make predictions.

8.3 Tweaking the Perceptron

Most of the ideas presented in this paper involve throwing more and more hardware into the problem. At no time have we attempted to improve the actual heart of the PBP—the perceptron.

A simple consideration is to adjust the starting weights of the perceptron. Through accidents with our code, we observed at times that starting with non-zero weights appeared to slightly improve performance. This effect is also seen in the different performances between the ZERO and ZEROONE replacement strategies. In a sense, there might be an "average" weight vector that we should bias our initial weights towards. To do this would require a deep understanding of the relationship between branch history and branch prediction.

Another possibility is to change the learning rule used by the perceptron. Consider a human manually adjusting the weights. If a long stream of mispredictions is encountered, he or she might begin to cheat and change the weights by more than 1. This could potentially improve the convergence of the perceptron. Since every misprediction is a pipeline flush, we want to converge as quickly as possible.

One approach to doing this in hardware is to associate a saturating counter with each perceptron. Whenever the perceptron predicts correctly, set it to 0. Otherwise, increment it. When the counter is full, adjust the weights by some other amount, say 2.

A second approach is to consider how inaccurate the perceptron actually was on a mispredict. For example, a branch was supposed to be not taken, but the output was a large, positive value. It will likely take a while before the perceptron's output is lowered to below zero. It follows then that if we are very far off, we should adjust the weights by a value larger than 1. This value could change depending on how large the error is.

These changes to the learning rule could improve the convergence rate of the perceptron and thus increasing prediction accuracy. It could also hurt convergence by constantly oscillating between two bad weight vectors. The nature of the program also matters. These adjusted learning rules would be beneficial if we were in a tight loop, but troublesome otherwise. Finally, this requires more hardware that could possibly be better utilized elsewhere.

9 Conclusions

In this paper, we discussed our implementation of the perceptron branch predictor work of Jiménez and Lin into the architecture simulator sim-alpha. Due to speculative prediction, the accuracy of our predictor was lower than those in the original paper, [2]. As this reflects a more realistic performance, we proceeded to propose several extensions to enhance the accuracy of a PBP to the rates originally reported.

The addition of a local history register to each perceptron was found to greatly benefit the prediction accuracy. While it appears that more local history increases the accuracy, we noticed a diminishing returns effect. Further study is warranted in deciding how to best balance the benefits of local history with the small additional hardware costs it requires.

In an attempt to combat aliasing, concepts from memory caches were explored. Treating the perceptron as a 4-way set-associative cache proved to perform poorly. It is our intuition that the loss of performance was due to a very non-uniform distribution of branches throughout the table. Larger tables would most likely fare better, although the additional hardware cost would probably be better used in a larger direct-mapped table with longer histories.

Victim caching proved to be more successful. Although the effect on accuracy was mixed across the various benchmarks, considerable improvement was seen on a couple of these programs. Although a victim cache requires somewhat complicated hardware, the slight performance increases we observed suggest further exploration of the use of these in not only PBP's, but other branch predictors as well.

Hardware costs will always have to be weighed against performance increase in designing branch predictors. This challenge becomes even more difficult with perceptron branch predictors as they require extensive hardware even in the most basic implementation. Regardless, the study of these predictors should be continued. While they may never outperform other branch predictor schemes, the high accuracy rate of PBP's suggests that branches are inherently linearly separable in terms of branch history. Further study of how perceptrons do so well will reveal powerful insights that might be used to improve all branch predictors.

Bibliography

[1] Agrawal and Woo. Improving Perceptrons for Dynamic Branch Prediction. Class Project at Carnegie Mellon University, 2001.

[2] Jiménez and Lin. Dynamic Branch Prediction with Perceptrons. *Proceedings of the 7th International Symposium on High-Performance Computer Architecture*, Barcelona, Spain, 2001.

[3] Kessler. The Alpha 21264 Microprocessor. *IEEE*, 1999.

[4] Michaud and Seznec. A Comprehensive Study of Dynamic Global History Branch Prediction, *Internal Publication No. 1406*, 2001.

[5] Mitchell. *Machine Learning.* McGraw-Hill, 1997.

[6] Russell and Norvig. *Artificial Intelligence: A Modern Approach*, Prentice Hall Series in Artificial Intelligence, 1995.

[7] Sim-alpha simulator. http://www.simplescalar.org

[8] Smith. A Study of Branch Prediction Strategies. *IEEE*, 1981.

TWOLF Benchmark

Predictor	Direction Hits	Direction Misses	Total	Percentage Predicted
Perceptron: 512,0,20,0,52	22142137	6674649	28816786	76.84%
Perceptron: 512,5,20,52,52	23756633	5065043	28821676	82.43%
Perceptron: 512,5,20,62,62	23753593	5068003	28821596	82.42%
Perceptron: 512,0,25,0,62	22269653	6546963	28816616	77.28%
Perceptron: 512,10,20,81,81	23923112	4623753	28546865	83.80%
Perceptron: 128,0,20,0,52	21774636	7039083	28813719	75.57%
Perceptron: 128,5,20,52,52	22970074	5845772	28815846	79.71%
Perceptron: 12,5,20,62,62	22962573	5854140	28816713	79.68%
Perceptron: 128,0,25,0,62	21838594	6977490	28816084	75.79%
Perceptron: 128,10,20,81,81	23347979	5467511	28815490	81.03%
Perceptron: 128,5,20,23,52	23286884	5531168	28818052	80.81%
Perceptron: 128,10,20,33,52	23437470	5381102	28818572	81.33%
Perceptron: 128,20,20,52,52	23497285	5322471	28819756	81.53%
Plain Zero: 128,10,20,33,52	23383542	5442099	28825641	81.12%
Plain One: 128,10,20,33,52	21906486	6907627	28814113	76.03%
Assoc Nothing: 128,10,20,33,52	22318087	6493342	28811429	77.46%
Assoc Zero: 128,10,20,33,52	22430951	6384629	28815580	77.84%
Assoc One: 128,10,20,33,52	22358041	6455616	28813657	77.60%
Assoc Common: 128,10,20,33,52	21965529	6848623	28814152	76.23%
5-Victim Nothing: 128,10,20,33,52	22426098	6388061	28814159	77.83%
5-Victim Zero: 128,10,20,33,52	22529353	6284614	28813967	78.19%
5-Victim One: 128,10,20,33,52	22554085	6259536	28813621	78.28%
5-Victim Common: 128,10,20,33,52	22002639	6811384	28814023	76.36%
10-Victim Nothing: 128,10,20,33,52	22504855	6308805	28813660	78.10%
10-Victim Zero: 128,10,20,33,52	22813155	6000519	28813674	79.17%
10-Victim One: 128,10,20,33,52	22826069	5987690	28813759	79.22%
10-Victim Common: 128,10,20,33,52	22039364	6774698	28814062	76.49%
5-VO Zero: 128,10,20,33,52	22104871	6709062	28813933	76.72%
5-VO One: 128,10,20,33,52	22108775	6705911	28814686	76.73%
5-VO Nothing: 128,10,20,33,52	23760984	5059392	28820376	82.45%
10-VO Zero: 128,10,20,33,52	22240345	6574247	28814592	77.18%
10-VO One: 128,10,20,33,52	22240779	6574217	28814996	77.18%
10-VO Nothing: 128,10,20,33,52	23673727	5147625	28821352	82.14%
Gshare: 1,1024,10,1	22081639	6733710	28815349	76.63%
Gshare: 1,1024,8,1	22637658	6173204	28810862	78.57%
Gshare: 1,1024,16,1	22078294	6737307	28815601	76.62%
Always Taken	17369415	11443470	28812885	60.28%
21264	26005992	2814296	28820288	90.24%

LUCAS Benchmark

Predictor	Direction Hits	Direction Misses	Total	Percentage Predicted
Perceptron: 512,0,20,0,52	54427288	6275528	60702816	89.66%
Perceptron: 512,5,20,52,52	56330667	4378034	60708701	92.79%
Perceptron: 512,5,20,62,62	56303495	4403769	60707264	92.75%
Perceptron: 512,0,25,0,62	54238404	6464393	60702797	89.35%
Perceptron: 512,10,20,81,81	57969537	2733352	60702889	95.50%
Perceptron: 128,0,20,0,52	54449625	4416050	58865675	92.50%
Perceptron: 128,5,20,52,52	56305660	6267851	62573511	89.98%
Perceptron: 12,5,20,62,62	56303209	4416050	60719259	92.73%
Perceptron: 128,0,25,0,62	54259338	6458105	60717443	89.36%
Perceptron: 128,10,20,81,81	57696697	3024724	60721421	95.02%
Perceptron: 128,5,20,23,52	59072448	1659562	60732010	97.27%
Perceptron: 128,10,20,33,52	55209938	5513784	60723722	90.92%
Perceptron: 128,20,20,52,52	58139330	2595650	60734980	95.73%
Plain Zero: 128,10,20,33,52	54813407	5921254	60734661	90.25%
Plain One: 128,10,20,33,52	55149516	5576946	60726462	90.82%
Assoc Nothing: 128,10,20,33,52	55366854	5336418	60703272	91.21%
Assoc Zero: 128,10,20,33,52	55211853	5502773	60714626	90.94%
Assoc One: 128,10,20,33,52	55384009	5318903	60702912	91.24%
Assoc Common: 128,10,20,33,52	55248899	5467387	60716286	91.00%
5-Victim Nothing: 128,10,20,33,52	55385990	5316908	60702898	91.24%
5-Victim Zero: 128,10,20,33,52	55385990	5316908	60702898	91.24%
5-Victim One: 128,10,20,33,52	55384165	5318674	60702839	91.24%
5-Victim Common: 128,10,20,33,52	55285837	5430197	60716034	91.06%
10-Victim Nothing: 128,10,20,33,52	54675138	6027975	60703113	90.07%
10-Victim Zero: 128,10,20,33,52	55230967	5471931	60702898	90.99%
10-Victim One: 128,10,20,33,52	55194805	5508026	60702831	90.93%
10-Victim Common: 128,10,20,33,52	55331280	5384848	60716128	91.13%
5-VO Zero: 128,10,20,33,52	49936727	10766214	60702941	82.26%
5-VO One: 128,10,20,33,52	49936627	10766865	60703492	82.26%
5-VO Nothing: 128,10,20,33,52	49955207	10748029	60703236	82.29%
10-VO Zero: 128,10,20,33,52	49935670	10767299	60702969	82.26%
10-VO One: 128,10,20,33,52	49936825	10766665	60703490	82.26%
10-VO Nothing: 128,10,20,33,52	49953375	10749879	60703254	82.29%
Gshare: 1,1024,10,1	60077585	638701	60716286	98.95%
Gshare: 1,1024,8,1	60133831	569167	60702998	99.06%
Gshare: 1,1024,16,1	60077585	638701	60716286	98.95%
Always Taken	33329579	27373239	60702818	54.91%
21264	60447939	442841	60890780	99.27%

GCC Benchmark

Predictor	Direction Hits	Direction Misses	Total	Percentage Predicted
Perceptron: 512,0,20,0,52	238195317	79164520	317359837	75.06%
Perceptron: 512,5,20,52,52	265314537	52053250	317367787	83.60%
Perceptron: 512,5,20,62,62	265157819	52207845	317365664	83.55%
Perceptron: 512,0,25,0,62	238519623	78841970	317361593	75.16%
Perceptron: 512,10,20,81,81	272613439	44760254	317373693	85.90%
Perceptron: 128,0,20,0,52	232351139	85012059	317363198	73.21%
Perceptron: 128,5,20,52,52	253213011	64152929	317365940	79.79%
Perceptron: 12,5,20,62,62	253003396	64363919	317367315	79.72%
Perceptron: 128,0,25,0,62	232594537	84767405	317361942	73.29%
Perceptron: 128,10,20,81,81	258691841	58677654	317369495	81.51%
Perceptron: 128,5,20,23,52	260719655	56651017	317370672	82.15%
Perceptron: 128,10,20,33,52	261165473	56201465	317366938	82.29%
Perceptron: 128,20,20,52,52	260614168	56749298	317363466	82.12%
Plain Zero: 128,10,20,33,52	258291699	59080837	317372536	81.38%
Plain One: 128,10,20,33,52	245873593	71490898	317364491	77.47%
Assoc Nothing: 128,10,20,33,52	244688109	72688928	317377037	77.10%
Assoc Zero: 128,10,20,33,52	248492746	68883451	317376197	78.30%
Assoc One: 128,10,20,33,52	250778638	66588263	317366901	79.02%
Assoc Common: 128,10,20,33,52	246293879	71074388	317368267	77.61%
5-Victim Nothing: 128,10,20,33,52	249672760	67698661	317371421	78.67%
5-Victim Zero: 128,10,20,33,52	252368338	64998798	317367136	79.52%
5-Victim One: 128,10,20,33,52	252320332	65045601	317365933	79.50%
5-Victim Common: 128,10,20,33,52	246869595	70502189	317371784	77.79%
10-Victim Nothing: 128,10,20,33,52	250183437	67190085	317373522	78.83%
10-Victim Zero: 128,10,20,33,52	253242957	64123171	317366128	79.80%
10-Victim One: 128,10,20,33,52	253230693	64135558	317366251	79.79%
10-Victim Common: 128,10,20,33,52	247315672	70052814	317368486	77.93%
5-VO Zero: 128,10,20,33,52	249979714	67384951	317364665	78.77%
5-VO One: 128,10,20,33,52	250004719	67360404	317365123	78.78%
5-VO Nothing: 128,10,20,33,52	265116582	52254379	317370961	83.54%
10-VO Zero: 128,10,20,33,52	251886812	65479417	317366229	79.37%
10-VO One: 128,10,20,33,52	251922930	65442892	317365822	79.38%
10-VO Nothing: 128,10,20,33,52	265126065	52242557	317368622	83.54%
Gshare: 1,1024,10,1	256427183	60948583	317375766	80.80%
Gshare: 1,1024,8,1	259465549	57910011	317375560	81.75%
Gshare: 1,1024,16,1	256427183	60948583	317375766	80.80%
Always Taken	193367708	123988295	317356003	60.93%
21264	295950630	21422272	317372902	93.25%

VPR Benchmark

Predictor	Direction Hits	Direction Misses	Total	Percentage Predicted
Perceptron: 512,0,20,0,52	135736780	33568145	169304925	80.17%
Perceptron: 512,5,20,52,52	144529426	24740611	169270037	85.38%
Perceptron: 512,5,20,62,62	144419425	24850240	169269665	85.32%
Perceptron: 512,0,25,0,62	138117729	31222795	169340524	81.56%
Perceptron: 512,10,20,81,81	145989304	23271561	169260865	86.25%
Perceptron: 128,0,20,0,52	132199263	37081921	169281184	78.09%
Perceptron: 128,5,20,52,52	139721249	29568671	169289920	82.53%
Perceptron: 12,5,20,62,62	139682496	29570126	169252622	82.53%
Perceptron: 128,0,25,0,62	134569808	34729286	169299094	79.49%
Perceptron: 128,10,20,81,81	140973906	28291892	169265798	83.29%
Perceptron: 128,5,20,23,52	141215344	28053306	169268650	83.43%
Perceptron: 128,10,20,33,52	141272244	28008585	169280829	83.45%
Perceptron: 128,20,20,52,52	141271913	28014662	169286575	83.45%
Plain Zero: 128,10,20,33,52	140216271	29063129	169279400	82.83%
Plain One: 128,10,20,33,52	139320234	30013235	169333469	82.28%
Assoc Nothing: 128,10,20,33,52	141090698	28191699	169282397	83.35%
Assoc Zero: 128,10,20,33,52	141611133	27711779	169322912	83.63%
Assoc One: 128,10,20,33,52	144057714	25337455	169395169	85.04%
Assoc Common: 128,10,20,33,52	137728685	31606849	169335534	81.33%
5-Victim Nothing: 128,10,20,33,52	143735446	25614187	169349633	84.87%
5-Victim Zero: 128,10,20,33,52	145194615	24150821	169345436	85.74%
5-Victim One: 128,10,20,33,52	145151029	24139892	169290921	85.74%
5-Victim Common: 128,10,20,33,52	138788038	30543060	169331098	81.96%
10-Victim Nothing: 128,10,20,33,52	145118312	24184612	169302924	85.72%
10-Victim Zero: 128,10,20,33,52	145678373	23646043	169324416	86.04%
10-Victim One: 128,10,20,33,52	145737172	23531448	169268620	86.10%
10-Victim Common: 128,10,20,33,52	139520746	29805647	169326393	82.40%
5-VO Zero: 128,10,20,33,52	141638339	27685224	169323563	83.65%
5-VO One: 128,10,20,33,52	141743027	27580926	169323953	83.71%
5-VO Nothing: 128,10,20,33,52	142911665	26420778	169332443	84.40%
10-VO Zero: 128,10,20,33,52	142457910	26856688	169314598	84.14%
10-VO One: 128,10,20,33,52	142552551	26763350	169315901	84.19%
10-VO Nothing: 128,10,20,33,52	142657771	26746217	169403988	84.21%
Gshare: 1,1024,10,1	130055454	39279074	169334528	76.80%
Gshare: 1,1024,8,1	135305911	33985059	169290970	79.93%
Gshare: 1,1024,16,1	130055454	39279074	169334528	76.80%
Always Taken	111862451	57626273	169488724	66.00%
21264	148976825	20293112	169269937	88.01%