Journal of Supercomputing manuscript No.
(will be inserted by the editor)

Balancing the Learning Ability and Memory Demand of a Perceptron-Based Dynamically Trainable Neural Network

Edward Richter · Spencer Valancius · Josiah McClanahan · John Mixter · Ali Akoglu

the date of receipt and acceptance should be inserted later

Abstract Artificial neural networks (ANNs) have become a popular means of solving complex problems in prediction based applications such as image and natural language processing. Two challenges prominent in the neural network domain are the practicality of hardware implementation and dynamically training the network. In this study we address these challenges with a development methodology that balances the hardware footprint and the quality of the ANN. We use the well-known perceptron-based branch prediction problem as a case study for demonstrating this methodology. This problem is perfect to analyze dynamic hardware implementations of ANNs because it exists in hardware and trains dynamically. Using our hierarchical configuration search space exploration, we show that we can decrease the memory footprint of a standard perceptron-based branch predictor by 2.3x with only a 0.6% decrease in prediction accuracy.

Keywords Artificial neural network · Branch prediction · Perceptron · SimpleScalar

Edward Richter
edwardrichter@email.arizona.edu

Spencer Valancius
svalancius12@email.arizona.edu

Josiah McClanahan
josiahmcclanahan@email.arizona.edu

John Mixter
jmixter6011@email.arizona.edu

Ali Akoglu
akoglu@email.arizona.edu

Department of Electrical and Computer Engineering, University of Arizona, Tucson, AZ, USA

1 Introduction

Neural networks offer a unique mechanism to solve problems in a number of applications [25]. Traditional neural networks follow a ”train and execute” model using offline learning. The neural network forward propagates the data, applies an error function, back propagates the error, and adjusts its parameters accordingly. After this initial training the data is presented to the network and is executed. The end result is a prediction based on the input and its training. This is a static process, that is, once the network is trained and deployed, there is no further updating of the weights. This model is popular due to its simplicity. However, there are many problems where it is either impractical or ill advised to cease training prior to execution. This can be because there is too much data to train offline, or the data is inherently dynamic, and should be treated as such. Online training, in contrast to offline training, refers to a dynamic adaptation model where the network's parameters are updated during execution. In this work, we use the terms dynamic training and online learning interchangeably.

We identify two major challenges seen today in implementing neural networks. First is the complexity of a dynamic adaptation model. Conventional ANNs [9, 18] follow a static training process where the training and execution are separate. However, when one must train while executing, a new method of training must be implemented that is more cohesive to execution. Second, is the ability to implement a neural network in hardware [2, 15, 16]. This is more difficult than implementing a neural network in software, as a software developer has complex data structures and a large number of resources at their disposal. Implementing a neural network inside of hardware that is capable of learning represents the pinnacle of neuromorphic research. Achieving this requires that neural network to possess the smallest hardware overhead possible.

In this work, we embrace these two challenges by investigating a hardware-friendly implementation of a perceptron-based, dynamically trained, neural network. We use branch prediction as a case study in order to quantify the learning and predictive characteristics of our design. Using branch prediction, we are able to compare our perceptron design in terms of memory footprint and prediction performance to the state-of-the-art neural branch predictors [12, 13, 14, 41].

A perceptron is a simple neuron model, which can be used to make powerful predictions in many application domains [26, 27, 39] when it is part of an ANN. Studies have shown that a perceptron based ANNs predictive power can also be utilized in the branch prediction context [12, 13, 14]. A perceptron-based branch predictor uses an applications branch history as the input vector, and outputs a prediction on whether a branch should be taken or not. Furthermore, for resource limited and low power embedded platforms, perceptron-based branch predictors have been shown to outperform the state of the art [7]. The perceptron branch prediction algorithm represents a very simple neural network implementation. Its simplicity exposes the two main challenges we are addressing. In the branch direction prediction algorithm, the network continually adapts as instructions are executed on the microprocessor. There is never a point where the perceptrons cease to continue their training, as such their weights are never static. By necessity, the branch predictor must be as small as possible and

the branch prediction must happen within as few clock cycles as possible. Meeting these two requirements results in a fast predicting, hardware based neural network, capable of learning dynamically.

Our paper makes the following contributions:

- Analyze the performance-memory footprint trade-off of the perceptron-based branch predictor.
- Introduce a hierarchical configuration search space exploration approach to identify the optimal perceptron configuration.
- Show that our branch predictor has significantly less memory footprint and similar prediction rates to the state-of-the-art designs.
- Investigate the use of different hashing algorithms as a means of decreasing memory footprint.

We show that it is possible to reduce the hardware cost by 14x with only a 1% decrease in branch prediction accuracy, while also showing that our perceptron-based branch predictor achieves 2.3x reduced memory footprint with only a 0.6% decrease in prediction accuracy against the standard perceptron branch predictor. Our methodology for developing a neural module for branch prediction can be adapted to solve modern problems that require high-speed pattern recognition.

A perceptron-based branch predictor is designed to continuously adapt to the changing patterns of past branch directions. This continual adaptation is only possible because the perceptron eventually gets the correct answer to every misprediction. When it receives the correct answer, the perceptron trains its weights to better predict its next direction. This continuous adaptation presents us with a useful function, anomalous pattern detection. During the review of our data, we observed that training activity changed significantly as the programs moved from one function to another. This can be explained by noting that when a perceptron is exposed to patterns for which the weights are not trained for, mispredictions occur. Because the perceptron only trains on a misprediction, the training activity increases. One of the byproducts of the continually adapting weights is the ability to detect changes in the input data. A helpful statistic to monitor in a dynamic adaptation model is the training rate, which measures the rate at which the neural network predicts incorrectly and must be retrained. If the training rate is monitored, it becomes evident when the benchmark software has changed functions, because the training rate changes. For instance, when within a loop of a repeating function, the training rate stabilizes, but as soon as it jumps out, the training rate changes. By extending this idea to an input of digital packets, or bus traffic, it becomes a simple matter not only to detect anomalous patterns, but by monitoring the training rate one could determine long term pattern changes. This also applies to detecting impending system failures by learning nominal operating parameters, and then detecting when these patterns deviate.

The rest of the paper is organized as follows: In Section 2 we discuss the related works. In Section 3, we give an overview of the the training, execution, and parameters of the perceptron branch predictor. In Section 4, we introduce our simulation environment used to run all our experiments. We present our parameter sweep methodology and results in Section 5. We analyze the results of our parameter sweeping experiments and present detailed discussions on the trade-offs between prediction

accuracy and hardware resource overhead in Section 6. Additionally, in the same section, we also compare our branch predictor with the state-of-the-art implementations in terms of resource requirements and prediction accuracy, as well as the evaluation of the impact of utilizing sophisticated hashing techniques on prediction accuracy. Lastly, in Section 7 we discuss our conclusions.

2 Related Work

There is a large body of work in the literature covering the branch prediction problem [22, 24, 33, 34, 35]. Das et al.[7] has shown that two state-of-the-art branch predictors: ISL-TAGE and LTAGE, have poor performance when implemented in an architecture with restricted storage. We are aware of more recent branch predictors that utilize perceptrons as a component in a larger prediction scheme [28]. However, due to the prohibitive cost of the perceptron-based predictor in terms of its memory requirement compared to simpler schemes, and its reduced accuracy compared to more complex schemes, solely perceptron-based branch predictor research ceased in 2013 [16]. In light of the machine learning renaissance, we revisit this problem with an exhaustive design space exploration approach with the goal of addressing this major memory footprint bottleneck. Therefore, our literature review is restricted to perceptron-based branch prediction studies.

Neural networks can be implemented to perform both static and dynamic branch predictions. Calder et al. [6] proposed strategies for predicting branches utilizing neural networks during compile time, where an applications opcode information is the input for deciding if a branch will be predicted taken or not taken. While neural networks can be more easily implemented within software, of the two methods, static branch predictors tend to perform worse than dynamic configurations. This is due to data changing from compilation to run time, which jeopardizes the validity of the static predictor [40]. Jimenez and Lin [13] proposed a dynamic branch prediction scheme using neural networks to predict branches at run time. This dynamic branch predictor implements a simple neuron model, the perceptron, using global history as inputs, to generate predictions for conditional branches at run time. Since the initial proposal of a dynamic neural branch predictor in 2000, there has been heavy research in the area of improving both accuracy and latency of such mechanisms. The majority of this research is on the topics involving execution, trace based simulation, or both. The following proposed schemes are prime material of such research:

2.1 Global/Local History

A global branch predictor makes a prediction employing the combined history of all recent branches. While a local branch predictor is using only the branch history of the branch in question to make its prediction. McFarling [23] proposed a global/local hybrid predictor which utilizes both types of history in order to generate a more learned judgment. Jimenez and Lin [14] make use of this concept in order to improve the inputs to the perceptrons within a perceptron-based predictor. They then proposed

a perceptron which takes inputs from both local and global histories to improve its decision making.

2.2 Path-Based History

Yeh and Patt [40] showed that using path-based history was helpful in making branch predictions. This can be seen in the Two-Level Adaptive Branch Predictor, where a shift register is used to keep track of path information before then being used to index into a pattern history table where a 2-bit saturating counter makes the prediction. Jimenez [12] proposed the Fast Path-Based Neural Branch Predictor, which takes advantage of path information to reduce the latency of the scheme as well as increase the prediction rates. Seznec [32] uses multiple predictor tables, hashing into them using various ranges of the path history. Zhou et al. [41] looks deeper into the findings of Seznecs paper and brings some key insights into the hashing and parameter configurations.

2.3 Training Threshold

Jimenez and Lin's first implementation of the perceptron branch predictor [13] trained the perceptron if the prediction was incorrect, or if the prediction was correct but the confidence was below a certain training threshold. Using this implementation, when the perceptrons outcome is around zero, it is a sign that the perceptron is not trained enough to make confident predictions. Zhou et al. [41] studied a mechanism for dynamically changing the training threshold through the use of a saturating counter.

2.4 Indexing Function

Many fields of computer architecture experience the issue of inappropriate indexing functions. Rau [30] explains the consequences with picking an ill-suited indexing function in memory applications, and describes multiple hashing functions that could be used to maximize the bandwidth of memory banks. The majority of perceptron branch predictors implement the use of a simple modulo mapping scheme in order to hash to a perceptron for making a branch prediction [12, 19]. Ma et al. [20] took the hashing algorithms described by Rau, and studied the effects that different indexing functions had on the performance of a perceptron predictor.

2.5 Scaling Coefficients

Amant, Jimenez, and Burger [3] observed that branches that occurred more recently have a greater impact on a given prediction than branches that occurred further in the past. As such, when a perceptron is considering a particularly large history, it is possible that the most significant bits of the shift register are damaging the prediction. They proposed using scaling of individual weights by predetermined coefficients based on

the history position. Zhou et al. [41] studied the effect of various implementations of scaling coefficients.

2.6 Analysis on Related Work

While much of the related work attempts to increase the branch prediction accuracy through minimizing destructive interference and utilizing more data, our goal is to minimize the resource requirement of the predictor. One of the main issues with the perceptron branch predictor is the large hardware overhead from implementing an ANN in a microprocessor. This study fills the gap of current research by proposing a hierarchical method for reducing the memory footprint of such a prediction scheme.

Algorithm 1 Perceptron making a prediction for a given branch instruction.

Input: h, $history\_array$, $weight\_array$
1: $sum = 0$
2: **for** $i = 0$ to h **do**
3: **if** $history\_array[i] == 1$ **then**
4: $sum = sum + weight\_array[i]$
5: **else**
6: $sum = sum - weight\_array[i]$
7: **end if**
8: **end for**
9: **if** $sum > 0$ **then**
10: predict taken
11: **else**
12: predict not taken
13: **end if**

Algorithm 2 Updating the weights of a perceptron on a misprediction.

Input: h, $actual\_direction$, $history\_array$, $weight\_array$
1: **for** $i = 0$ to h **do**
2: **if** $actual\_direction == history\_array[i]$ **then**
3: $weight\_array[i] = weight\_array[i] + 1$
4: **else**
5: $weight\_array[i] = weight\_array[i] - 1$
6: **end if**
7: **end for**

3 Background and Configuration Parameters

In a perceptron-based branch predictor, there exists a single array of history of length h, and n perceptrons. The array of history contains the direction of the last h branches, with a one representing 'taken' and a zero representing 'not taken'. An individual perceptron consists of an array of h weights. Each element of the array of history is paired

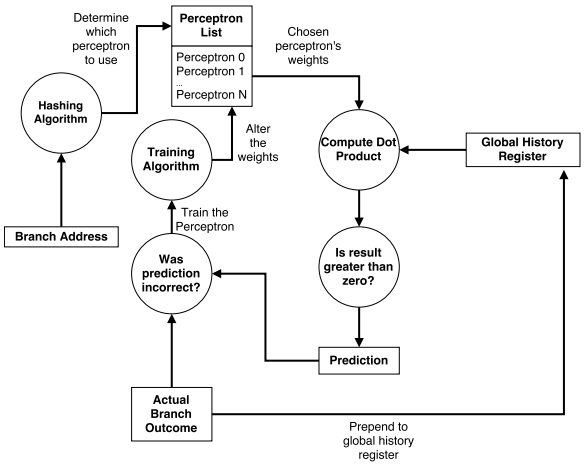


Fig. 1 Block diagram representing the hashing, prediction, and training flow of the perceptron-based branch predictor.

with an element of the array of weights. Ideally, there would be one perceptron per branch instruction, but this would be impractical for applications that include large amounts of branch instructions. Therefore, a hashing scheme is typically utilized to determine the perceptron that will be used to predict a particular branch.

At the start of execution, the value of each individual weight is chosen randomly. When the microprocessor executes a branch instruction, we determine which perceptron will make the prediction through the hashing scheme. A perceptron makes a prediction by taking the dot product of its weight and history arrays. The array of history is solely binary, so the dot product is defined as: adding the weight when the corresponding history element is one, and subtracting the weight if the corresponding history element is zero. If the result of the dot product is greater than zero, the perceptron predicts 'taken', otherwise it predicts 'not taken'. We show this process in Algorithm 1. If a perceptron mispredicts a branch direction, it is trained so it may make a more informed prediction in the future. Perceptrons train by processing the current state of the history and weight arrays associated with the misprediction. If an element in the array of history agrees with the actual direction of the branch, its weight value is incremented, otherwise it is decremented. This process can be seen in Algorithm 2. A block diagram representing the flow of choosing which perceptron to use, predicting the outcome, and then training can be seen in Figure 1. This scheme generates a design with the following parameters:

- Global History Length (*GHL*): Defined as the number of previous branches the predictor uses to make a prediction. The GHL is equivalent to h, and is used for determining the number of weights in a perceptron.
- List Size (*LS*): Defined as the number of perceptrons within the branch predictor.
- Weight Size (*WS*): Defined as the number of bits used to represent a single weight inside of a perceptron. The weight size affects two aspects of the weight:

1. Max Weight (*MxW*): Defined as the maximum possible weight allowed for a weight in a perceptron. This value is determined by the number of bits to represent the weight, i.e a weight size of 4-bits equates to a MxW of 7 as it is a signed number.
2. Min Weight (*MnW*): Defined as the minimum possible weight allowed for a weight in a perceptron. This value is the negative of the MxW to use the entire weight to its full capacity.

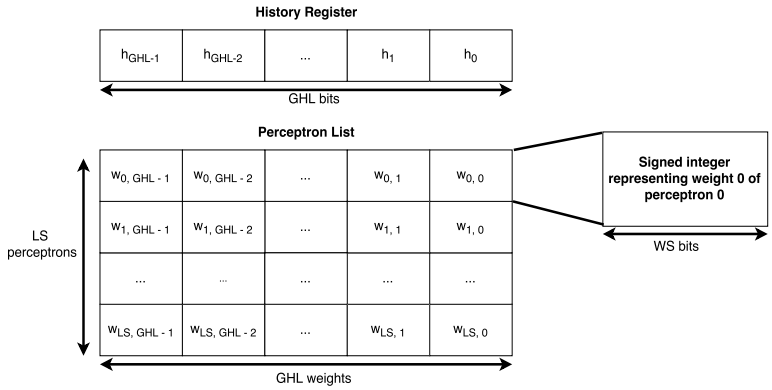


Fig. 2 Block diagram representing the architecture of the perceptron-based prediction scheme.

Figure 2 illustrates the hardware architecture of the prediction scheme. By using the three configuration parameters listed above, an accurate approximation for the memory footprint of the scheme in terms of the required number of bits is calculated using Equation 1 below. We use Equation 1 for measuring the memory footprint of the configurations in our design space. Other perceptron-based branch predictor studies [13, 14] define this metric as "Hardware Budget". We believe that the term "Memory Footprint" is a more accurate term for this metric.

$$Memory\ Footprint\ = GHL + WS * GHL * LS \tag{1}$$

Due to the *GHL*, *WS*, and *LS* being so intricately linked to the memory footprint, it becomes vital to refining them as much as possible. Studies have shown that prediction accuracy logarithmically increases with history length [13] and the number of perceptrons [19]. Jimenez and Lin [13] have further found that varying the number of bits used for weight size, based on the history length, increases the prediction performance. There is a point where the accuracy plateaus regardless of how large the *GHL*, *LS*, and *WS* are set. This presents an opportunity to identify a configuration that operates on lower memory footprint, while achieving a prediction performance that is close to the peak performance.

Algorithm 3 Algorithm describing the flow of a parameter sweep.

Input: *benchmarks*, *min_ghl*, *max_ghl*, *min_ls*, *max_ls*, *min_ws*, *max_ws*

```
for benchmark in benchmarks do
    for ghl = min_ghl to max_ghl do
        for ls = min_ls to max_ls do
            for ws = min_ws to max_ws do
                run_simulation(benchmark, ghl, ls, ws)
            end for
        end for
    end for
end for
```

Table 1 Integer and floating point benchmarks from the spec2000 suite.

Integer	Floating Point
vpr	applu
gcc	apsi
parser	mesa
gap	galgel
vortex	art
bzip2	equake
twolf	ammp
crafty	lucas
gzip	fma3d
mcf	mgrid
	swim

4 Simulation Environment

In this paper, the perceptron-based branch predictor [13] is applied in the SimpleScalar 3.0 [5] simulation environment. SimpleScalar is a CPU architecture simulator, which contains a branch prediction simulator. We use this simulator to run the various parameter sweeps described in detail in Section 5. We conduct the parameter search experiments using 500,000,000 instructions and execute them with no fast-forwarding as both the transient and steady-state response of the predictor are important for a complete analysis. We use the spec2000 benchmarks [8] shown in Table 1 to test the effectiveness of the perceptron-based predictor. We use integer and floating-point benchmarks to show that our scheme is relevant for a variety of applications. Algorithm 3 describes the process for which the parameter sweeps were conducted. We use the spec2000 Alpha binaries to run our experiments. Jimenez and Lin [14] use a trace-based simulation technique, and report their compiler information. Zhou et al. [41] use SimpleScalar for their simulations, but did not disclose any compiler information regarding their benchmarks.

We use two statistics to determine the effectiveness of a configuration: memory footprint and branch direction-prediction rate. Branch direction-prediction rate is the rate at which the direction of a branch is successfully predicted. These two statistics demonstrate our claim that it is possible to aggressively decrease the memory footprint of a perceptron prediction scheme while still maintaining a quality branch direction-prediction rate.

All our simulations are executed on the University of Arizona's High Performance Computing (HPC) system. This system consists of 336 nodes. Each node contains 192 GB of memory and two 28-core processors. The resources provided by the HPC system make it possible to run thousands of simulations in parallel, allowing us to feasibly explore a massive design space.

5 Parameter Sweep

To capture the memory footprint-performance trade-off that exists in the perceptron branch predictor, we run two parameter sweeps in a hierarchical manner. In Section 5.1, we explain our initial parameter sweep, a massive exploration of 3 out of 21 benchmarks that allows us to identify the trends such as rapid degradation of branch prediction performance with respect to changes in the perceptron configuration. This initial exploration ultimately reveals the relationships between the perceptron configuration parameters and the branch prediction performance, which allows us to narrow down the exploration search space for the second parameter sweep over all 21 benchmarks.

In Section 5.2 we present the final parameter sweep, a smaller design space closer to the optimal configuration. The second parameter sweep involves analysis over the entire benchmark suite, allowing us to determine a design that has minimal memory footprint while still performing comparably to the state-of-the-art.

For both parameter sweeping experiments, we plot the prediction accuracy over the sweeping range of a given configuration parameter. For each value in the sweeping range, we take the average of prediction accuracies over all configurations for that specific parameter value. For example, the prediction accuracy for 32 perceptrons is the average prediction accuracy for all configurations in the design space with 32 perceptrons.

As stated in Section 3, a hashing function is needed to select a specific perceptron to be used for the branch prediction. We use the modulus hashing function, as it is the most common hashing function found in the literature [12, 19]. We will later present our analysis over various hashing schemes in Section 6.4. Jimenez and Lin [13] determined that the GHL and WS that led to the best performance were in the range 12 - 62 and 7 - 9 respectively. As our goal is to reduce the memory footprint of the design, we chose to further the investigation of the lower bounds of the design space. We did not have any initial information on the optimal LS of a configuration, as such we chose a range with the intention of expanding upon further discovery. The final parameter sweep has a higher upper bound, as we determined that we could increase the LS and still maintain a low memory footprint.

5.1 Initial Parameter Sweep

The initial parameter sweep is substantial, involving 29,500 tests across three benchmarks for 88,500 total simulations. Table 2 contains the range of the parameters in the initial parameter sweep. We conduct sweeping experiments on three spec2000 integer

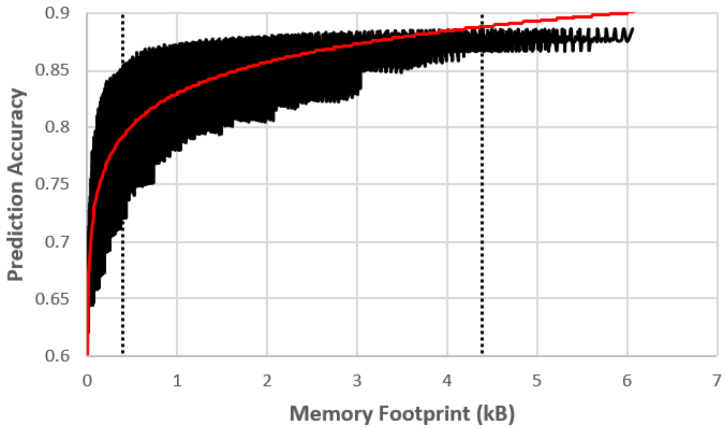


Fig. 3 Prediction accuracy vs memory footprint from the initial parameter sweep for the *twolf* benchmark. Between the vertical dotted lines indicates the area interest with regards to acceptable hardware complexity and sufficient prediction accuracy.

Table 2 Configuration range of the initial parameter sweep.

GHL	LS	WS
4 - 62 bits	1 - 100 perceptrons	4 - 8 bits

benchmarks: *gcc*, *twolf*, and *vpr*. We choose these benchmarks, because other branch prediction studies [12, 14, 41] highlight them as having particularly difficult branches to predict. As such, making configuration changes across these benchmarks allows us to conduct analysis over noisy data so that we don't exclude configurations that may have high impact on prediction accuracy with low hardware resource requirements.

Figure 3 displays a scatter plot presenting the prediction accuracy versus memory footprint relationship for the entire sweep using the *twolf* benchmark. The red line shows the logarithmic regression curve for the data set used to illustrate the diminishing returns generated from increasing the memory footprint. The highest prediction accuracy seen is 88.62% with a branch prediction configuration of 100 perceptrons with *GHL* of 62, and *WS* of 8 bits; equating to a memory footprint of 6.06 kB. On the other hand, 100 perceptrons with *GHL* of 18, and *WS* of 6 bits results in a memory footprint of 1.32 kB and a prediction accuracy of 87.62%. This shows that with only a 1% decrease in prediction accuracy, the memory overhead of a predictor can be reduced by a factor of 4.6. For a particularly lightweight implementation, 26 perceptrons with *GHL* of 16, and *WS* of 5 bits results in memory footprint of 0.256 kB with 80% accuracy for *twolf*. While this is a significant performance decrease of 8.62%, the memory footprint of the predictor operates at 23.7x less than the maximum memory footprint configuration. These memory footprint savings could create massive latency, area, and energy savings.

Table 3 Configuration range of the final parameter sweep.

GHL	LS	WS
20 - 40 bits	32, 64, 128 perceptrons	5 - 7 bits

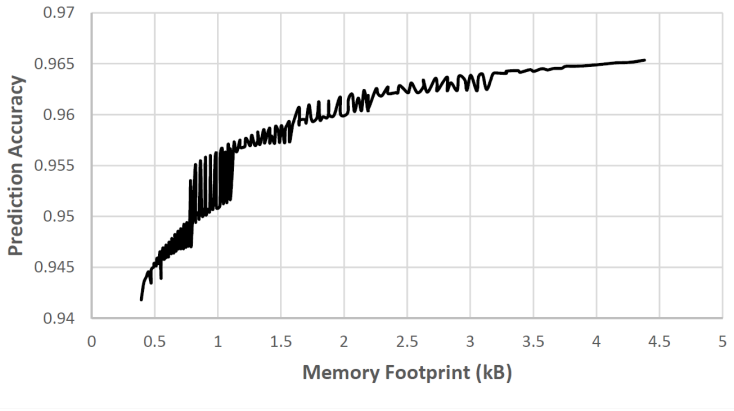


Fig. 4 Prediction accuracy vs memory footprint from the final parameter sweep using all benchmarks shown in Table 1.

As the first parameter sweep includes 29,500 simulations per benchmark, it is not feasible to implement on all 21 benchmarks. Therefore, it is necessary to narrow down the parameter ranges to the areas of interest and run that parameter sweep on every benchmark. In our experiments, we define an arbitrary threshold of 85% in order to narrow down the search space. In doing so we observe that configurations containing a *GHL* less than 20, *LS* less than 32, or *WS* of 4 rarely surpass a prediction accuracy of 85% on *twolf*. Therefore, we exclude those parameter ranges in the second round of sweeping experiments. Jimenez and Lin [14] report using a 4 kB perceptron branch predictor. Therefore, we also exclude the configurations that generate a memory footprint significantly larger than 4 kB. The upper and lower limits of the search space can be seen in Table 3. The black vertical lines on Figure 3 represent the minimum and maximum memory footprints in the final design space. Configurations before the first mark have minuscule memory footprints, but rapidly degrading prediction accuracies. Conversely, configurations after the second mark have high prediction accuracies, but the memory footprint becomes too costly for the performance gain observed.

5.2 Final Parameter Sweep

For the final parameter sweep, we use all benchmarks listed in Table 1 across the values shown in Table 3. The new ranges for the *GHL* and *WS* are directly determined based on the initial parameter sweep experiments. When determining the new range for the *LS*, we also consider the behavior of the modulus hashing function. As shown

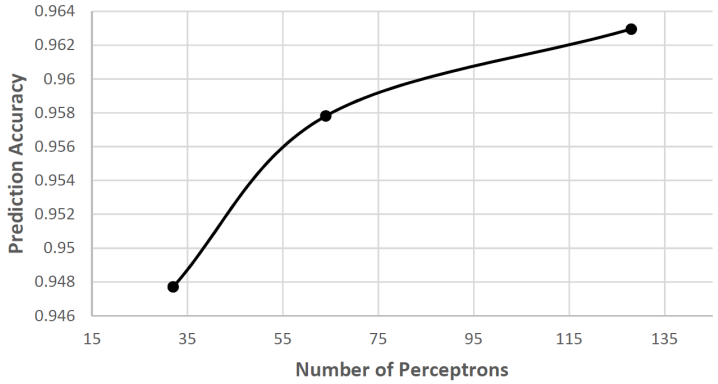


Fig. 5 Prediction accuracy vs number of perceptrons using all benchmarks shown in Table 1.

in Table 3, the *LS* takes values that are only powers of two. This is a critical design decision as having a number of perceptrons that is a power of two allows us to reduce the hardware complexity by implementing the modulus hashing function with a single *AND* gate.

Figure 4 displays the average prediction accuracy across all benchmarks versus the memory footprint for a given configuration. As in the initial sweeping experiments, we observe a logarithmic trend in the prediction accuracy. When we compare Figures 3 and 4 side by side, for each perceptron configuration, the prediction accuracy is much higher when sweeping experiments are based on all 21 of the benchmarks. Here we note that floating point benchmarks typically represent scientific computing applications with loop structures whose branch patterns are highly predictable. On the other hand, the integer benchmarks involve complex control flow structures making them much more challenging for extracting correlations between branch operations. Considering that 11 out of 21 benchmarks in this final parameter sweep are floating point benchmarks, and we are taking the average across all benchmarks for each configuration, we observe a significant increase in prediction accuracy.

After analyzing the effect that the memory footprint has on the prediction accuracy, it is necessary to observe the influence that the *LS*, *WS*, and *GHL* have on the prediction accuracy as well. Figure 5 shows that increasing the number of perceptrons increases the accuracy in a logarithmic fashion. Doubling the number of perceptrons from 32 to 64 increases the prediction accuracy by 1.01%, but doubling the number of perceptron again from 64 to 128, only increases the prediction accuracy by 0.514%. It can be inferred that the performance increase will only get smaller as the number of perceptrons rise past this point. This plateauing effect signifies that at some point after 128 perceptrons, it becomes increasingly less beneficial to use more perceptrons, as the accuracy does not increase enough to justify the resulting hardware cost.

After analyzing the *LS* of the branch predictor, we focus on the impact that the *GHL* and *WS* have on the prediction accuracy. Figure 6 shows that there is a log-

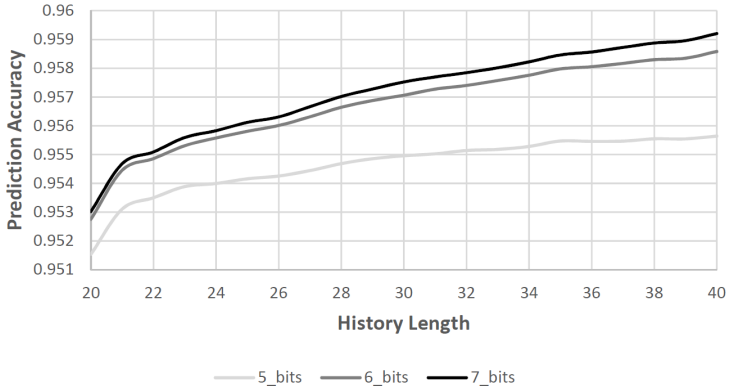


Fig. 6 Prediction accuracy vs the *GHL*. The different colored lines indicate varying WS. The prediction accuracy is the average of all benchmarks seen in Table 1.

arithmic relationship between the history length and the prediction accuracy. This relationship is similar to that of the *LS*'s.

Figure 6 also shows that the point at which the performance gain of increasing the history length falls off is dependent on the *WS*. This is due to the fact that the *WS* determines how much a perceptron can learn, and the *GHL* decides how much data the perceptron is required to learn. A smaller *GHL* allows for a smaller overall required *WS* for each perceptron to sufficiently learn the branch patterns of the application. Conversely a larger *GHL* would require a much larger *WS* to accurately maintain a sufficiently large peripheral of the application's branch pattern. It is seen that the average difference between using a *WS* of 5 bits and a *WS* of 6 bits is 0.209%, while the average difference between using a *WS* of 6 bits and 7 bits is 0.0409%. This joint relationship between the *WS* and *GHL* on prediction accuracy leads to an efficient means of cutting the memory footprint. If a designer is willing to reduce the *GHL* in order to save memory overhead, then it will come with little performance degradation to also reduce the *WS* as smaller history lengths don't need large weights in order to learn their patterns.

It is clear that there is an ideal operating point that lies somewhere in the trade-off between memory footprint and accuracy. By using the logarithmic nature of the relationship between accuracy and memory footprint, we can then choose a point that performs near peak performance, with a significantly smaller memory footprint.

In order to determine the optimized perceptron branch predictor, we create an ideal configuration to compare accuracies. This ideal configuration is composed of 256 perceptrons, a history length of 96 bits, and a weight size of 8 bits, all yielding a total memory footprint of 24.0 kB. This is found to have an average prediction accuracy of 97.1% across all 21 benchmarks. A configuration based on 128 perceptrons with history length of 22, and weight size of 5 bits results with an average accuracy of 96.1%. Our chosen configuration was selected as the configuration with a small memory footprint and a prediction accuracy within 1% of the best performing configuration in the final design space exploration. This configuration requires only 1.721

kB, a remarkable reduction, by 14x in hardware resource requirements compared to the ideal configuration.

There are four major relationships we use to generate this result. The logarithmic relationship between prediction accuracy and *LS*, *WS*, and *GHL* all allow the possibility to minimize the memory footprint with small decreases in accuracy. The last relationship that is useful in creating the optimized perceptron configuration is the combined relationship between the *WS*, *GHL* and prediction accuracy. These relationships allow us to reduce the memory footprint by 14x with only a 1% decrease in prediction accuracy.

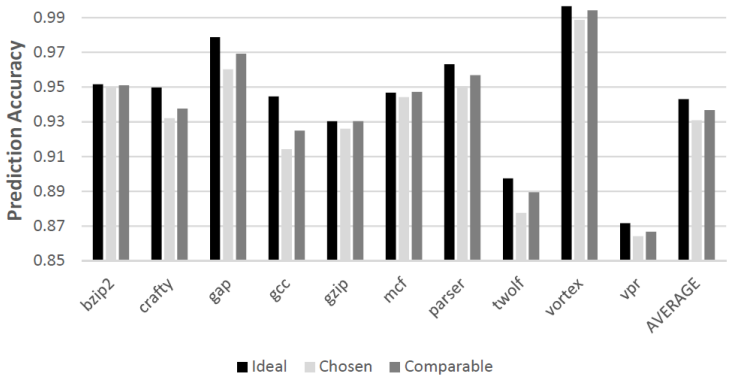
6 Experimental Results

The parameter sweep experiments in Section 5 highlight the relationships between the prediction accuracy and the three parameters which define the predictor configuration (*WS*, *GHL*, and *LS*). These relationships show that it is possible to aggressively decrease the memory footprint of the predictor with only a small decrease in prediction accuracy. In Section 6.1, we show a side-by-side comparison of three configurations with similar prediction accuracies, but very different memory footprints.

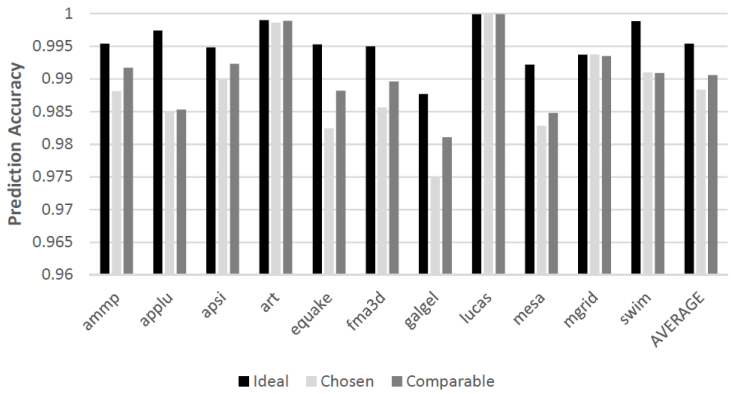
6.1 Memory Footprint Comparison

Figure 7 displays a histogram of the accuracies of the ideal configuration versus our selected configuration and the 4kB based implementation for all integer (7a) and floating point (7b) benchmarks. We use the 4kB version for comparing with Jimenez and Lin's implementation that is also based on a 4kB design. We refer to the 4kB configuration as Comparable, and has a *GHL* of 37. a *LS* of 128, and a *WS* of 7. As stated earlier, the ideal configuration is composed of a *GHL* of 96, *LS* of 256, and *WS* of 8, for a total memory footprint of 24.0 kB. The chosen configuration consists of a *GHL* of 22, *LS* of 128, and *WS* of 5, creating the memory footprint of 1.721 kB.

Our first observation is the ideal and comparable configurations both have higher or equal prediction accuracy when compared to the chosen configuration for every benchmark. The chosen configuration has an average prediction accuracy that is 0.954% lower than the ideal configuration, and 0.409% lower than the comparable configuration. For integer benchmarks, the chosen configuration has a prediction accuracy that is 1.23% lower than the ideal configuration, and 0.6% lower than the comparable configuration. The chosen configuration has a prediction accuracy that is 0.702% worse on floating point benchmarks compared to the ideal configuration, and 0.22% percent worse compared to the comparable configuration. When looking at particular integer benchmark, it is seen that the largest difference in prediction accuracies is between the ideal and chosen configuration on the *gcc* benchmark with 3.04%, and the smallest difference is seen between the ideal and comparable configuraiton on the *bzip* benchmark at 0.07%. When doing the same analysis for floating point benchmarks, the largest difference is seen in *equake* at 1.29%, while the chosen configuration, ideal configuration, and comparable configuration all have the same accuracies for both *lucas* and *mgrid*.



(a) Integer Benchmarks



(b) Floating point benchmarks

Fig. 7 Comparing prediction accuracy of the ideal configuration, with no hardware constraints, the chosen configuration, with a minimized memory footprint, and the comparable configuration to Jimenez and Lin [14] for all benchmarks seen in Table 1.

There are two reasons that the difference in prediction accuracy between the three configurations are smaller for floating point benchmarks versus integer benchmarks. First, as previously stated, floating point benchmarks contain more scientific computation than integer benchmarks, creating loop related branch behavior, which is easier to predict than control flow branch behavior. The simplicity in making predictions in a benchmark such as *lucas*, with a direction prediction rate of 100% for both configurations, does not require large amount of resources. The second reason, which is connected to the first, is that the parameter we decrease the most in the chosen predictor configuration is the *GHL*. The ideal configuration has a *GHL* of 96 and the chosen configuration has a *GHL* of 22. This is a significant reduction in history of 74 bits, meaning that the ideal configuration can use 74 previous branches as information that the chosen configuration cannot. Therefore, the chosen configuration's direction-

prediction rate will decrease as the distance between correlated branches increases. Integer benchmarks tend to have a larger distance between correlated branches than floating point, due to the if-else-then branching patterns of integer benchmarks and the loop oriented branch patterns of floating point benchmarks.

The selected configuration makes up for a slightly smaller hit rate by utilizing almost 14x less hardware than the ideal configuration. This clearly shows that while some benchmarks perform worse than others at this lower memory footprint, the average performance difference is small enough to make the hardware reduction practical. In summary, we show that it is possible to alter the predictor configuration in such a way that the memory footprint can be minimized with little effect on the prediction accuracy. We also show that a similar trend holds for a comparable configuration to Jimenez and Lin's [14] implementation. In the following section, we evaluate the performance of our branch predictor with respect to the state-of-the-art perceptron-based branch predictors in terms of prediction rate and memory footprint requirements.

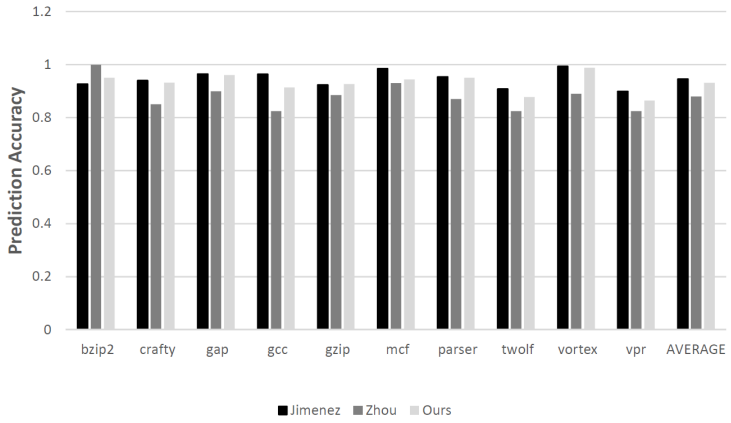
6.2 Performance Comparison

Our proposed perceptron prediction scheme has a small memory footprint obtained through two-level parameter sweeping strategy and analysis of the relationships between the configuration parameters and the prediction accuracy. In this section, we compare our proposed perceptron branch predictor configuration (*LS* of 128 perceptrons, *GHL* of 22, and *WS* of 5 totaling a memory footprint of 1.721 kB) with other perceptron-based predictors [14, 41].

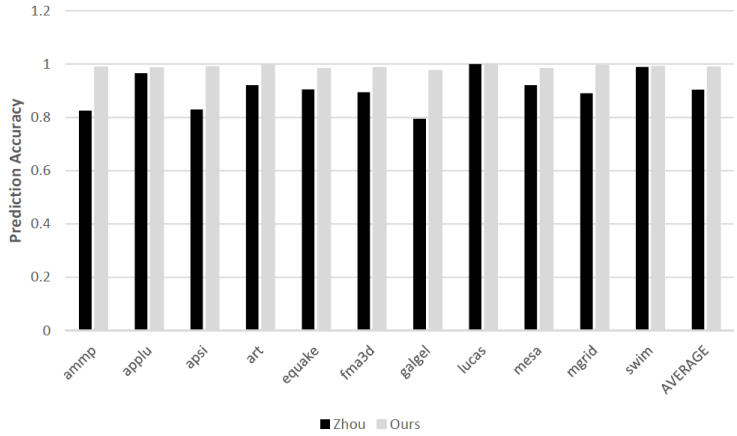
Jimenez and Lin [14] implement a 4 kB standard global perceptron branch predictor with a *GHL* of 24 and a *WS* of 8. Comparatively, our chosen configuration is able to reduce the memory footprint by a factor of 2.3x. Among all the configurations that result with close to a 4kB storage requirement, our best-case implementation reaches a prediction accuracy of 93.7% on the integer benchmarks. When we reduce the storage size to 1.721kB, the prediction accuracy is 93.1%. By sacrificing only 0.6% of accuracy, we are able to reduce the memory footprint by 2.3x. We believe that this comparison contains less noise than the comparison we make to Jimenez and Lin [14], and thus a better representation of our contribution.

Zhou et al. [41] implement a much more complicated perceptron branch predictor than ours with features such as path-based history and multiple tables of perceptrons hashed by the path information. These features are supported with mechanisms for adjusting the impact of each entry in the history register and updating the training frequency based on a saturating counter.

To make our comparison fair, and keep the configurations as close as possible, the results we present from Zhou et al. [41] are from the configuration where there is no saturating counter. The configuration of the predictor was: 7 bits to represent a weight and global history length of 128 bits. Zhou et al. [41] does not calculate the memory footprint of the scheme they use, and it would be inappropriate for us to attempt to guess the memory footprint. These prediction schemes, experiments, and implementations are different, thereby making the results difficult to compare. However, the goal of this analysis is to show that our optimized branch predictor is



(a) Integer Benchmarks



(b) Floating point benchmarks

Fig. 8 Comparison between our optimized perceptron configuration and the state-of-the-art perceptron-based branch predictors, by using all benchmarks seen in Table 1.

comparable and/or better than the state-of-the-art perceptron branch predictors, while maintaining an overhead low enough to be feasibly implemented in hardware.

Figure 8 displays a histogram similar to Figure 7, but instead compares the difference between our, Jimenez and Lin's [14], and Zhou et al.'s [41] implementations. As can be seen in Figure 8b, Jimenez and Lin [14] do not have results on floating point benchmarks, so we cannot compare those results.

Figure 8a shows that our optimized perceptron behavior is 1.54% less accurate than Jimenez's [14] and 5.1% more accurate than Zhou et al.'s [41] results on the integer benchmarks tested. The benchmark that has the greatest diversity in results between the three implementations is *gcc*. Jimenez and Lin [7] report a prediction accuracy of 96.5%, while Zhou et al. [16] report an accuracy of 82.5%. Our configu-

ration is in between these two with a prediction of 91.4%. The benchmark that has the least diversity between the three implementations is *gzip*. Jimenez and Lin [14] report a prediction accuracy of 92.5% while Zhou et al. [41] report a prediction accuracy of 88.5%. Our proposed configuration has a prediction accuracy of 92.6% for *gzip*.

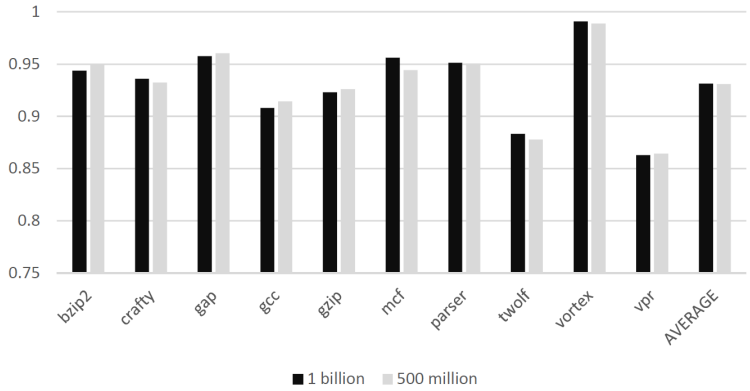
Figure 8b shows that our perceptron-based branch predictor has a hit rate on floating point benchmarks which is on average 8.52% more accurate than Zhou's [41] published work. The benchmark that has the greatest difference is *ammp* which has a 16.3% difference. The benchmark with the smallest difference is *lucas* which has the same prediction accuracy.

Figure 8 shows that our optimized perceptron configuration is 1.54% less accurate than Jimenez and Lin's implementation [14], and 6.89% more accurate than Zhou et al.'s [41]. As stated before, Zhou at al. [41] implement many different mechanisms to increase the prediction accuracy such as path based information, and scaling of the impact weights have on the prediction. These enhancements, despite not improving their prediction accuracy, must add hardware overhead. Jimenez and Lin [14] have a much more comparable configuration, and due to their increased memory footprint it makes sense that their average prediction accuracy is slightly higher. However, Figure 8 as a whole shows that the optimized perceptron configuration branch predictor behaves either significantly better or slightly worse than the state-of-the-art, with a minimal memory footprint.

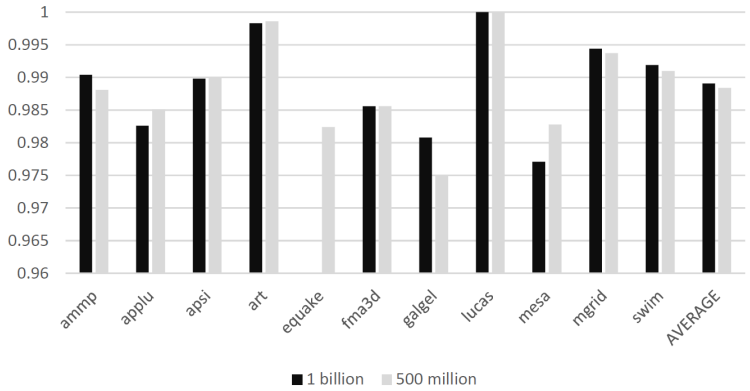
In overall the area minimized implementation results with a 1.54% prediction accuracy degradation with respect to Jimenez and Lin [14]. However, we believe there are multiple platforms or applications where the memory footprint reduction is worth the accuracy loss. For example, this prediction accuracy loss has less impact on processors with shallow depth pipelines, such as the ARM Cortex-M7 [1], than processors with deep pipelines, such as the ARM Cortex-A8. The ARM Cortex-M7 observes a maximum of a five-cycle branch misprediction penalty as it has six pipeline stages. On the other hand, the misprediction penalty is 13 cycles for the ARM Cortex-A8 processor [29]. Furthermore, for resource limited and low power embedded platforms, perceptron-based branch predictors have been shown to outperform the state of the art resource rich predictors such as TAGE [7]. The acceptable amount to decrease the prediction accuracy is also highly dependent on the application. Bhattacharjee [4] presents an implantable processor that uses a perceptron-based branch predictor for predicting the neuronal activity in mice. In this processor, the branch predictor switches between two modes to operate as a branch predictor or a neuronal predictor. The paper states that due to switching between two modes, there is a 3% decrease in branch prediction accuracy, which is not discernible on the workloads run on the processor.

6.3 Increasing Instruction Count

As stated earlier in Section 4, all of our simulations were based on 500,000,000 instructions. Sherwood et al. [36] has shown that the overall program behavior is vastly different from the initial phase of execution. Therefore, we run another experiment based on our chosen configuration (*GHL* of 22, *WS* of 128, and *LS* of 5) using the



(a) Integer Benchmarks



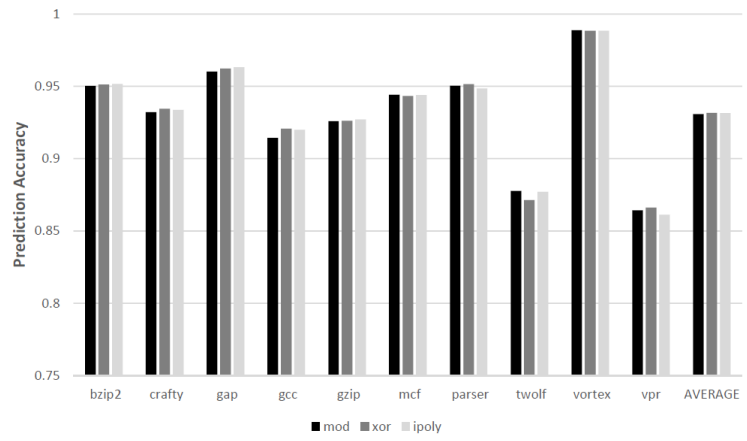
(b) Floating point benchmarks

Fig. 9 Comparing the prediction accuracies of different instruction counts using our chosen prediction scheme. All of the benchmarks can be found in Table 1.

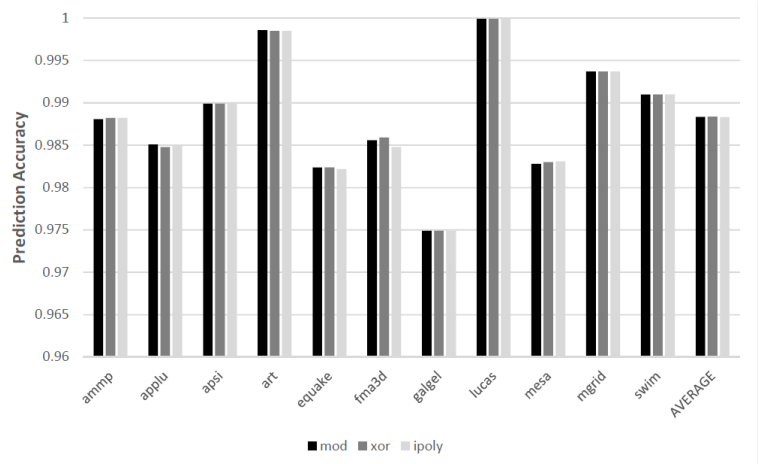
spec2000 benchmarks with 1,000,000,000 instructions. Based on the side by side comparison of prediction accuracy over the two instruction sizes for each benchmark shown in Figure 9, we observe that the accuracy remains unchanged. In the integer and floating point benchmarks, the 1,000,000,000 simulation performed 0.044% and 0.073% better than the 500,000,000 simulation respectively.

6.4 Hashing Algorithms and Standard Deviation

One difficulty that has been the focus of work throughout the history of branch prediction, is minimizing destructive interference/aliasing [20, 40]. Destructive aliasing occurs when two branch addresses with different branch patterns map to the same prediction unit, which in our case is a perceptron. This difficulty is also present in



(a) Integer Benchmarks



(b) Floating point benchmarks

Fig. 10 Comparing the effects of different hashing functions on the chosen configuration across all benchmarks seen in Table 1.

the memory domain, where mapping requests to the same module can reduce bandwidth. This problem is minimized by various mapping functions, which have been used to evenly distribute requests to modules [30]. Given that sophisticated hashing techniques have been shown to reduce the amount of conflicts, in this section we incorporate two techniques into the branch-to-perceptron mapping problem. Our aim is to identify the hashing scheme that results with high hit rate with minimal memory footprint. We experiment with utilizing more sophisticated indexing functions than the modulus function, such as bitwise-XORing [20] and Pseudo Random Interleaving [30] as a means of minimizing destructive aliasing by evenly distributing branch instructions across the perceptron list.

Pseudo Random Interleaving is a method which utilizes irreducible polynomial modulus mapping. In irreducible polynomial modulus mapping, the polynomial *R(x)*, which is found by calculating *R(x)* = *A(x)* mod *P(x)*, is the polynomial in the Galois Field GF2 associated with the integer that hashes into the perceptron list. For example, the integer 19 would be represented by the polynomial $x^4 + x + 1$ as the binary representation of 10011, which is 19 in decimal form. *A(x)* is the polynomial in Galois Field GF2 associated with the branch address. *P(x)* is an irreducible polynomial in GF2 of degree $log_2(LS)$ chosen at design time. In our experiments, we used $x^7 + x + 1$ (Poly 131) as *P(x)*. Rau [30] found a way in order to do this calculation through a series of XOR gates.

Bitwise-XOR-mapping[20] is a means of generating the index to the perceptron list by XORing the least significant n bits of the address, with the next n bits of the address, where $n = log_2(LS)$. This XORing with other bits distributes the accesses similar to the way a *gshare* [23] predictor would distribute accesses across a pattern history table in a two-level dynamic branch predictor.

Intuition leads us to the believe that a sophisticated hashing algorithm would allow us to further decrease the memory footprint of the perceptron branch predictor with no loss in accuracy. For example, utilizing an XOR based mapping scheme would generate overhead in the forms of gates, but this overhead would be far negated by the ability to decrease either the *LS*, *GHL*, or *WS* while still maintaining the same accuracy.

Figure 10 compares the hit rates of a modulus hashing function (*mod*), a bitwise-xor indexing function (*xor*), and an irreducible polynomial modulus mapping indexing function (*ipoly*) for both integer (10a) and floating point (10b) benchmarks. The perceptron branch predictor configuration throughout the hashing experiments is the same as the chosen configuration in Section 6.1. For the floating-point benchmarks, the prediction accuracy rate between *mod*, *xor*, and *ipoly* indexing functions are less than 0.001% different, meaning that modulus is the most optimal hashing for floating point benchmarks as it requires the smallest hardware overhead. For the integer benchmarks, the difference between the prediction accuracy between *mod* and *xor*, was 0.076%, and the difference between the prediction accuracy between *mod* and *ipoly* was 0.066%. As one can see, the negligible accuracy increase from using a more complex mapping function does not justify the increase in the hardware resource demand.

The results from these experiments show that all hashing functions have a similar performance across all benchmarks. This can be attributed to destructive aliasing. Despite the fact that the branches are more evenly distributed, there is still a certain amount of destructive aliasing that occurs. This makes us believe that the distribution difference would have to be significant in order to see a performance increase worth the added complexity.

7 Conclusion

In this paper, we embraced the challenge of a hardware implementation of a dynamically trained neural network in hardware. We did this using branch prediction as a

case study, in order to quantify our findings in a domain that has been highly researched. We proposed a hierarchical methodology to massively reduce the memory footprint of a perceptron-based branch predictor. We then used this methodology to reduce the memory footprint by 14x with only a 1% decrease in prediction accuracy. We showed that our configuration has a 2.3x memory reduction with only a 0.6% accuracy penalty when compared to the standard. We also showed our intent to further decrease the memory footprint with a sophisticated hashing algorithm, and explained the downfalls in this approach. These findings can now be used to take on the challenge of implementing a dynamic perceptron-based neural network in hardware.

Even though the work presented in this study is limited to the branch prediction application, the logarithmic trade-off between hardware resource requirement and prediction accuracy that we have observed in this study, also exists in other ANN applications such as image classification [10, 11, 38] and natural language processing [17]. These studies show that there is a diminishing return in prediction accuracy as we increase the bitwidth of design parameters such as the weight and bias in the ANN. Designing a hardware architecture in a resource limited environment for these applications would require a design space exploration that is similar to the method presented in this paper.

We see many exciting applications that can utilize our findings. Bhattacharjee [4] utilizes the perceptron branch predictor to predict neuronal activity in mice. With our studies, such applications can reduce the memory footprint of their scheme while only slightly degrading their prediction accuracy. This research is applicable to all ANN application classes, which are attempting to massively reduce the memory footprint with acceptable accuracy degradation. The design space for implementing a neural network is so large, it is difficult to know which configuration parameters can be altered. Our experiments give guidance on how altering our specific neural networks size of input vector (*GHL*), parameter bitwidth (*WS*), and number of neurons (*LS*) effect both the memory footprint and the prediction accuracy. The decision of the *WS* and *LS* is evident in all ANN applications, while the decision of choosing the *GHL* is only seen in applications where it is possible to change the size of the input vector. This choice occurs in the speech processing [21, 31] for denoising and classification applications, and security domain for creating a predictor to monitor the traffic on a bus or switch to detect anomalous patterns.

As future work, we plan to implement both our prediction method and Jimenez and Lins method using an FPGA platform, which will allow us to obtain an accurate hardware complexity analysis. For future work, we intend to look deeper into the branch prediction domain, as well as implement this neural network for other applications. We plan to use the branch predictor as a starting point to move the perceptron-based neural network into less researched fields and applications. In terms of branch prediction, we intend to look further into hashing as a means to further decrease the memory requirement of the branch predictor. It is proposed that if we are able to use another way to decrease destructive aliasing such as the Agree prediction mechanism [37] to couple with a sophisticated hashing algorithm it will largely increase performance. In terms of other applications, as mentioned before, if the network of perceptrons is used to monitor a bus or switch, it would be able to learn the underly-

ing behavior of that medium. If irregular behavior was occurring, it would be obvious because the neural network would be mispredicting what would happen next.

References

1. (2014) ARM Cortex-M7 Processor. ARM, revision r0p2
2. Akopyan F, Sawada J, Cassidy A, Alvarez-Icaza R, Arthur J, Merolla P, Imam N, Nakamura Y, Datta P, Nam GJ, Taba B, Beakes M, Brezzo B, Kuang JB, Manohar R, Risk WP, Jackson B, Modha DS (2015) Truenorth: Design and tool flow of a 65 mw 1 million neuron programmable neurosynaptic chip. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 34(10):1537–1557, DOI 10.1109/TCAD.2015.2474396
3. Amant RS, Jimenez DA, Burger D (2008) Low-power, high-performance analog neural branch prediction. In: 2008 41st IEEE/ACM International Symposium on Microarchitecture, pp 447–458, DOI 10.1109/MICRO.2008.4771812
4. Bhattacharjee A (2017) Using branch predictors to predict brain activity in brain-machine implants. In: Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture, ACM, New York, NY, USA, MICRO-50 '17, pp 409–422, DOI 10.1145/3123939.3123943, URL http://doi.acm.org/10.1145/3123939.3123943
5. Burger D, Austin TM (1997) The simplescalar tool set, version 2.0. SIGARCH Comput Archit News 25(3):13–25, DOI 10.1145/268806.268810, URL http://doi.acm.org/10.1145/268806.268810
6. Calder B, Grunwald D, Lindsay D, Martin J, Mozer M, Zorn B (1995) Corpus-based static branch prediction. SIGPLAN Not 30(6):79–92, DOI 10.1145/223428.207118, URL http://doi.acm.org/10.1145/223428.207118
7. Das M, Banerjee A, Sardar B (2017) An empirical study on performance of branch predictors with varying storage budgets. In: 2017 7th International Symposium on Embedded Computing and System Design (ISED), pp 1–5, DOI 10.1109/ISED.2017.8303913
8. Henning JL (2000) Spec cpu2000: measuring cpu performance in the new millennium. Computer 33(7):28–35, DOI 10.1109/2.869367
9. Hochreiter S, Schmidhuber J (1997) Long short-term memory. Neural Comput 9(8):1735–1780, DOI 10.1162/neco.1997.9.8.1735, URL http://dx.doi.org/10.1162/neco.1997.9.8.1735
10. Hubara I, Courbariaux M, Soudry D, El-Yaniv R, Bengio Y (2016) Binarized neural networks. In: Lee DD, Sugiyama M, Luxburg UV, Guyon I, Garnett R (eds) Advances in Neural Information Processing Systems 29, Curran Associates, Inc., pp 4107–4115, URL http://papers.nips.cc/paper/6573-binarized-neural-networks.pdf
11. Hubara I, Courbariaux M, Soudry D, El-Yaniv R, Bengio Y (2016) Quantized neural networks: Training neural networks with low precision weights and activations. CoRR abs/1609.07061, URL http://arxiv.org/abs/1609.07061, 1609.07061
12. Jimenez DA (2003) Fast path-based neural branch prediction. In: Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture,

IEEE Computer Society, Washington, DC, USA, MICRO 36, pp 243–, URL http://dl.acm.org/citation.cfm?id=956417.956562
13. Jimenez DA, Lin C (2001) Dynamic branch prediction with perceptrons. In: Proceedings HPCA Seventh International Symposium on High-Performance Computer Architecture, pp 197–206, DOI 10.1109/HPCA.2001.903263
14. Jimenez DA, Lin C (2002) Neural methods for dynamic branch prediction. ACM Trans Comput Syst 20(4):369–397, DOI 10.1145/571637.571639, URL http://doi.acm.org/10.1145/571637.571639
15. Jouppi NP, Young C, Patil N, Patterson D, Agrawal G, Bajwa R, Bates S, Bhatia S, Boden N, Borchers A, Boyle R, Cantin P, Chao C, Clark C, Coriell J, Daley M, Dau M, Dean J, Gelb B, Ghaemmaghami TV, Gottipati R, Gulland W, Hagmann R, Ho RC, Hogberg D, Hu J, Hundt R, Hurt D, Ibarz J, Jaffey A, Jaworski A, Kaplan A, Khaitan H, Koch A, Kumar N, Lacy S, Laudon J, Law J, Le D, Leary C, Liu Z, Lucke K, Lundin A, MacKean G, Maggiore A, Mahony M, Miller K, Nagarajan R, Narayanaswami R, Ni R, Nix K, Norrie T, Omernick M, Penukonda N, Phelps A, Ross J, Salek A, Samadiani E, Severn C, Sizikov G, Snelham M, Souter J, Steinberg D, Swing A, Tan M, Thorson G, Tian B, Toma H, Tuttle E, Vasudevan V, Walter R, Wang W, Wilcox E, Yoon DH (2017) In-datacenter performance analysis of a tensor processing unit. CoRR abs/1704.04760, URL http://arxiv.org/abs/1704.04760, 1704.04760
16. Khan MM, Lester DR, Plana LA, Rast A, Jin X, Painkras E, Furber SB (2008) Spinnaker: Mapping neural networks onto a massively-parallel chip multiprocessor. In: 2008 IEEE International Joint Conference on Neural Networks (IEEE World Congress on Computational Intelligence), pp 2849–2856, DOI 10.1109/IJCNN.2008.4634199
17. Ko JH, Fromm J, Philipose M, Tashev I, Zarar S (2017) Precision Scaling of Neural Networks for Efficient Audio Processing. ArXiv e-prints 1712.01340
18. Krizhevsky A, Sutskever I, Hinton GE (2012) Imagenet classification with deep convolutional neural networks. In: Pereira F, Burges CJC, Bottou L, Weinberger KQ (eds) Advances in Neural Information Processing Systems 25, Curran Associates, Inc., pp 1097–1105, URL http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf
19. Lu Y, Liu Y, Wang H (2011) A study of perceptron based branch prediction on simplescalar platform. In: 2011 IEEE International Conference on Computer Science and Automation Engineering, vol 4, pp 591–595, DOI 10.1109/CSAE.2011.5952918
20. Ma Y, Gao H, Zhou H (2006) Using indexing functions to reduce conflict aliasing in branch prediction tables. IEEE Transactions on Computers 55(8):1057–1061, DOI 10.1109/TC.2006.133
21. Maas A, Le QV, ONeil TM, Vinyals O, Nguyen P, Ng AY (2012) Recurrent neural networks for noise reduction in robust asr. In: INTERSPEECH
22. Mao Y, Shen J, Gui X (2018) A study on deep belief net for branch prediction. IEEE Access 6:10,779–10,786, DOI 10.1109/ACCESS.2017.2772334
23. McFarling S (1993) Combining branch predictors. Tech. Rep. TN-36m, Digital Western Research Laboratory, Palo Alto, CA

24. Michaud P, Seznec A (2014) Pushing the branch predictability limits with the multi-poTAGE+SC predictor. In: 4th JILP Workshop on Computer Architecture Competitions (JWAC-4): Championship Branch Prediction (CBP-4), Minneapolis, United States, URL https://hal.archives-ouvertes.fr/hal-01087719
25. Murray AF (1995) Applications of Neural Networks. Springer US
26. Nazzal J, M El-Emary I, A Najim S (2008) Multilayer perceptron neural network (mlps) for analyzing the properties of jordan oil shale. World Applied Sciences Journal 5
27. Orhan U, Hekim M, Ozer M (2011) Eeg signals classification using the k-means clustering and a multilayer perceptron neural network model. Expert Systems with Applications 38(10):13,475 – 13,481, DOI https://doi.org/10.1016/j.eswa.2011.04.149, URL http://www.sciencedirect.com/science/article/pii/S0957417411006762
28. Parasanna S, Sarma R, S B (2017) A study on improving branch prediction accuracy in the context of conditional branches. International Journal of Engineering Technology Science and Research 4
29. Patterson DA, Hennessy JL (2013) Computer Organization and Design, Fifth Edition: The Hardware/Software Interface, 5th edn. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA
30. Rau BR (1991) Pseudo-randomly interleaved memory. In: Proceedings of the 18th Annual International Symposium on Computer Architecture, ACM, New York, NY, USA, ISCA '91, pp 74–83, DOI 10.1145/115952.115961, URL http://doi.acm.org/10.1145/115952.115961
31. Sainath T, Vinyals O, Senior A, Sak H (2015) Convolutional, long short-term memory, fully connected deep neural networks. In: ICASSP
32. Seznec A (2005) Analysis of the o-geometric history length branch predictor. In: 32nd International Symposium on Computer Architecture (ISCA'05), pp 394–405, DOI 10.1109/ISCA.2005.13
33. Seznec A (2007) The L-TAGE Branch predictor. Journal of Instruction Level Parallelism URL http://wwwjilp.org/vol9
34. Seznec A (2011) A 64-kbytes ISL-TAGE branch predictor. In: Proceedings of the 3rd Championship Branch Prediction
35. Seznec A (2011) A new case for the tage branch predictor. In: Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture, ACM, New York, NY, USA, MICRO-44, pp 117–127, DOI 10.1145/2155620.2155635, URL http://doi.acm.org/10.1145/2155620.2155635
36. Sherwood T, Sair S, Calder B (2003) Phase tracking and prediction. In: Proceedings of the 30th Annual International Symposium on Computer Architecture, ACM, New York, NY, USA, ISCA '03, pp 336–349, DOI 10.1145/859618.859657, URL http://doi.acm.org/10.1145/859618.859657
37. Sprangle E, Chappell RS, Alsup M, Patt YN (1997) The agree predictor: A mechanism for reducing negative branch history interference. In: Conference Proceedings. The 24th Annual International Symposium on Computer Architecture, pp 284–291, DOI 10.1145/384286.264210
38. Umuroglu Y, Fraser NJ, Gambardella G, Blott M, Leong P, Jahre M, Vissers K (2017) Finn: A framework for fast, scalable binarized neu-

ral network inference. In: Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, ACM, New York, NY, USA, FPGA '17, pp 65–74, DOI 10.1145/3020078.3021744, URL http://doi.acm.org/10.1145/3020078.3021744

39. Vanzella E, Cristiani S, Fontana A, Nonino M, Arnouts S, Giallongo E, Grazian A, Fasano G, Popesso P, Saracco P, Zaggia S (2004) Photometric redshifts with the Multilayer Perceptron Neural Network: Application to the HDF-S and SDSS. AstronAstrophys 423:761–776, DOI 10.1051/0004-6361:20040176, astro-ph/0312064
40. Yeh TY, Patt YN (1991) Two-level adaptive training branch prediction. In: Proceedings of the 24th Annual International Symposium on Microarchitecture, ACM, New York, NY, USA, MICRO 24, pp 51–61, DOI 10.1145/123465.123475, URL http://doi.acm.org/10.1145/123465.123475
41. Zhou Z, Kejriwal M, Miikkulainen R (2013) Extended scaled neural predictor for improved branch prediction. In: The 2013 International Joint Conference on Neural Networks (IJCNN), pp 1–7, DOI 10.1109/IJCNN.2013.6707059