BRAT: BRANCH PREDICTION VIA ADAPTIVE TRAINING

A Dissertation
Presented to
The Academic Faculty

By

Jonathan Lafiandra

In Partial Fulfillment
of the Requirements for the Degree
Master of Computer Science in the
School of Computer Science
Center for Research into Novel Computing Hierarchies

Georgia Institute of Technology

August 2021

BRAT: BRANCH PREDICTION VIA ADAPTIVE TRAINING

Thesis committee:

Dr. Tom Conte
Computer Science
Georgia Institute of Technology

Date approved: July 30, 2021

ACKNOWLEDGMENTS

I would like to thank the members of the CRNCH lab for their help in preparation of this work – Pulkit Gupta, without whom this work would not be nearly as complete, Tom Conte, who helped to challenge my preconceived notions and provided key insights, and Anirudh Jain, who helped me solve any problems that cropped up along the way.

Special thanks are due to the good people at Northrop Grumman for funding this research and especially to Brian Konigsburg and Paul Tschirhart for their help and advice throughout the project.

The author gratefully acknowledges the support for this work offered by the NSF MRI award #1828187: ”MRI: Acquisition of an HPC System for Data-Driven Discovery in Computational Astrophysics, Biology, Chemistry, and Materials Science.” Without these computing resources, the research would have progressed much slower.

TABLE OF CONTENTS

Acknowledgments iii

List of Tables vii

List of Figures viii

Summary ix

Chapter 1: Introduction 1

Chapter 2: Background and Related Work 4

2.0.1 Online Training 4

2.0.2 Offline Training 6

2.0.3 Fixed Point 7

Chapter 3: Neural Branch Prediction 8

3.1 Key Terms for a Multi-Layer Neural Network 8

3.1.1 Fully Connected Layer 8

3.1.2 LeakyReLU 9

3.1.3 Max-Pooling 9

3.1.4 Sigmoid 9

3.2 The Layout of BRAT 10

3.2.1 Inputs to the Network 10

3.2.2 Topology of BRAT 10

3.2.3 Online Training vs Offline Training 10

3.3 Forward Propagation 11

3.4 Backpropagation 11

Chapter 4: Architecture Overview 13

4.0.1 Hardware Optimizations for Forward Propagation 13

4.0.2 Pipelined Implementation of Forward Propagation 14

4.1 Backwards Propagation 15

4.1.1 Backwards Propagation and Optimizations 16

4.1.2 Pipelined Implementation of Backwards Propagation 18

4.2 Side-Effects of Pipelining BRAT 18

4.3 Prediction Latency and Mitigation Techniques 19

4.4 Training the HWNN 19

4.5 Methodology 20

4.5.1 Traces Selected 20

4.5.2 Architecture 21

4.5.3 Other Simulated Predictors 23

Chapter 5: Evaluation and Results 24

5.1 Accuracy 24

Chapter 6: Conclusion 28

References 29

LIST OF TABLES

4.1 Traces from the Competition Branch Prediction Traces (CBP). Each trace has $>$ 25M conditional branches. Traces are ordered by their performance on a 64KB gshare predictor. 20

4.2 Selected Traces from SPEC2017 Integer. Traces are selected such that enough SimPoint PinBalls are available to achieve $> 90\%$ coverage. 21

4.3 Table of best performing configurations across all traces based on memory footprint, Bimodal Height $\geq 2^{14}$ uses G-Share instead 22

4.4 Size of Arithmetic Portion of BRAT Predictor 22

LIST OF FIGURES

3.1 Perceptron 8

4.1 A Neural Network with ReLU Activation Layer 13

4.2 Error calculation flow 15

4.3 Training 15

5.1 Accuracy of the BRAT predictor across the spread of CBP traces at various memory budgets. The best performing configuration for each memory budget is chosen for each benchmark. 24

5.2 Accuracy of the BRAT predictor across the spread of SPEC benchmarks at various memory budgets. The best performing configuration for each memory budget is chosen for each benchmark. 25

5.3 Misprediction rates compared to state-of-the-art predictors with a 64KB memory budget for CBP 26

5.4 Misprediction rates compared to state-of-the-art predictors with a 64KB memory budget for SPEC 27

SUMMARY

In this thesis, BRAT is researched as a new hardware structure for cost-efficient branch prediction. Relying on the fundamentals of machine learning, BRAT computes a branch decision through a multi-layer neural network. To demonstrate the merits of BRAT, it is used to predict branches in a typical pipeline and evaluate its accuracy. By utilizing a hidden layer and activation functions, BRAT is able to introduce non-linearity and enable more accurate prediction of branch outcomes because this structure exposes relationships that may not be easily captured by a perceptron based approach or other popular methods. The memory utilized by BRAT scales linearly with the number of inputs in the decision process. At most memory footprints, BRAT is competitive with state-of-the-art branch predictors of equivalent memory budgets. Additionally, as the memory footprint is increased, it is shown how BRAT scales and how larger predictors in the future may perform.

CHAPTER 1

INTRODUCTION

Branch prediction continues to be a bottleneck in general purpose CPU performance and energy efficiency despite several recent proposals [1]. With ever increasing pipeline depths, and a stagnation in state of the art branch prediction techniques, there is a need for a new approach to push forward that frontier. Most recent proposals to improve branch prediction accuracy are built as additions to TAGE [2, 3, 4], or use pre-trained (offline) techniques such as the neural network approach used by the BranchNet predictor to augment a primary branch predictor for a small subset of branches in a program [5].

Conventional online approaches to branch prediction, such as the Perceptron predictor and TAGE, operate on a combination of global and path histories in order to learn correlations. While these predictors are agile and quickly adapt to different program phases, they are limited in their complexity and effectiveness due to online training, and subsequently struggle with non-linearity of branch correlations in the case of the Perceptron, and of ever increasing storage demands in the cases where branches depend on longer histories as is the case with TAGE. BranchNet, a recent proposal, attempts to alleviate these shortcomings by augmenting TAGE with an offline trained convolutional neural network tailored to predict a handful of hard-to-predict branches. However, the offline training approach suffers from various shortcomings including a prohibitive dependence on a primary predictor, apriori knowledge of the target workloads, long training times even with multiple GPUs, and unreasonably large storage requirements for tracking more than a handful of branches (~ 512 Bytes per branch).

Neural binary predictors, defined as multi-layer neural networks with binary inputs and online training, are well-fitted solutions to binary decision problems. in 1991, [6] showed that a multi-layer feed-forward network is capable of universal approximation. Since then, neural architectures have exploded in popularity in a variety of domains such as image recognition [7] and network intrusion detection [8]. Neural binary predictors have been shown to be perfect fit for branch prediction due to the non-linearity of branch behavior, the importance of correct prediction in modern, deep pipelines, and the ability to train over time; the ability to map non-linearity is important because it theoretically allows the network to, given enough size and time, learn every branch function [9].

However, the problem faced in most of the previous work regarding neural architectures for branch prediction is that the hardware cost of online training and prediction is prohibitively expensive[5]. This is largely due to the back-propagation phase of a neural network requiring either floating point arithmetic or high precision fixed-point arithmetic. Many papers have proposed novel solutions to the hardware cost of neural networks such as [10, 5, 11]; however, these solutions require offline training such as in [10, 5] or digital-analog hardware such as in [11].

This paper presents Branch Prediction via Adaptive Training (BRAT), a novel multi-layered neural network based branch predictor that overcomes the shortcomings of prior neural based branch prediction without the overheads of offline training while still retaining the adaptability and swift training of online approaches. Because deeper neural networks are capable of learning much more complex relationships than shallow networks such as the perceptron, BRAT is designed to have a hidden layer and non-linear activation functions. However, even this small increase in complexity for the neural network has substantial impacts on the physical architecture. In order to reduce some of the cost, BRAT takes inspiration from the perceptron predictor [9] and utilizes binary inputs to significantly reduce the number of multipliers and enable other optimizations on the network.

This paper will first discuss related work and state-of-the art branch predictors (section chapter 2), then detail an overview of the multi-layer neural approach to branch prediction (section chapter 3), followed by details of BRAT's architecture (section chapter 4). This will be followed by a discussion of the evaluation methodology, and then the results will be compared and discussed against other branch predictors (sections section 4.5 and chapter 5). Finally, the paper is concluded in section chapter 6.

CHAPTER 2
BACKGROUND AND RELATED WORK

Branch predictors can broadly be categorised into online (runtime) and offline based on their internal state update method. Online training has increasingly been hampered with exponentially increasing history lengths when presented with long complex and noisy histories, exploding the implementation cost for negligible performance improvements. Recent works have proposed using neuromorphic approaches in an offline setting to remedy this weakness [5]. However, these approaches rely on a traditional primary predictor that they augment with an expensive in memory neural predictor that is used for a susbset of tough to predict branches. Moreover, offline approaches suffer from prohibitively expensive (in resource and time) offline training required for each workload (Up to 16 hours across 4 GPUs [5]).

2.0.1 Online Training

Online branch predictors are typically organized as table based predictors that use a combination of global and path history to index into one or more prediction tables. The current state of the art predictors are derivatives of the TAGE – TAgged GEometic history length [4] and hashed perceptron [9] predictors.

TAGE uses an approximate PPM history compression technique to track the most common branch histories and hashes the global branch and local path histories to lookup tables of tagged saturating counters that provide the final prediction. Each one of its various counter tables uses a unique history length, and longer history lengths are used when shorter histories provide insufficient prediction accuracy. When predictions rely on deeper history lengths, i.e. the branch history contains uncorrelated branches or the positions of correlated branches in the history is non-deterministic, TAGE is forced to allocate a dedicated prediction counter per history pattern, causing an over-dependence on the larger history tables, causing TAGE to behave like a global 2-level predictor in the worst case scenario [12]. TAGE-SC-L [2] is the most state of the art TAGE family predictor that augments vanilla TAGE with a Loop predictor (L) and a Statistical Corrector (SC). The former is used to improve branches in a loop by correlating with iteration count, while the later tracks and corrects branches that TAGE repeatedly gets incorrect. The Statistical Corrector portion of TAGE-SC-L can easily be adapted to BRAT with minor modifications in its implementation timeline to meet the timing constraints and get timely predictions in a pipeline.

The second category of recent online branch proposals are based on the perceptron predictor. A perceptron is a neural network in its simplest form, i.e. it is a single layered neural network that learns correlations between branch outcomes and the global history. The perceptron predictor computes a summation of each history bit with a individual correlation factors and then compares the result with a branch bias in order to make the final prediction. However, (i) the perceptron predictor is unable to learn non-linear correlations, and (ii) non-deterministic branch locations in the history bits can cause miss predictions. The hashed perceptron [9] alleviates the later problem by hashing the global branch and path history and learning correlation factors on these hashed values. However, aliasing among history patterns continues to be a problem, resulting in loss of prediction accuracy. Multi-layered neural branch predictors solve the problem of capturing non-linear correlations between branches. However, their online training has apriori been considered too expensive. However, as explained in the subsequent sections, BRAT presents a fully online multi-layered neural network that is implementable in today's processors with various performance benefits over the existing state of the art.

Other proposals of online training are older architectures such as the Bi-Mode predictor proposed in [13]. Bi-Mode uses two gshare predictors and chooses which one is used for the final prediction based on a choice-predictor. Using multiple gshares and mechanisms to choose between them limits the destruction in longer histories a problem in traditional

2-level predictors.

2.0.2 Offline Training

Offline predictors use application profiling to augment branch prediction accuracy. The simplest form of this offline training is to learn the statistical bias of branches via compile time optimizations such as value range propagation that are then applied during runtime to increase the prediction accuracy [14, 15, 16, 17]. More recent work uses profiling for training application-specific predictors such as the Spotlight predictor that augments a gshare like predictor with the more useful global history segments. However, for Spotlight to be effective, correlated branches must exist at the same location during runtime that they did during the profiling. The most recent offline training proposal, BranchNet, uses an on-chip convolution neural network that is trained offline to augment a primary predictor, TAGE-SC-L, in order to increase the correctly predicted fraction of a handful of hard to predict branches in a program.

BranchNet uses three mutually exclusive traces (training, validation and test) to train its convolutional neural network. The the 100 highest MPKI branches from the validation set are identified and the network is trained and tested on the training and test sets. A maximum of 41 branches are then encoded into the predictor based on the branches that offer the best improvements in prediction accuracy. Branchnet is limited by two main factors, first it requires a primary predictor for all the branches except the 41 hard to predict branches that can be encoded into it. This is due to the exponential memory requirements of adding more branches, making it infeasible as a primary predictor. Second, offline training is both inconvenient and expensive – the target program must be known beforehand, and still requires between 6-18 hours of training time when using 4 state of the art GPUs in parallel per benchmark program. Both these factors together make BranchNet unsuitable for modern day processors with ever evolving target applications and program behaviors.

Considering the inability of BranchNet to be used as a primary branch predictor, qualitative comparisons are against the state of the art online training branch predictors, TAGE-SC-L and perceptron in the evaluation chapter 5.

2.0.3 Fixed Point

In order to reduce the implementation cost of the BRAT predictor, fixed point representation is utilized for storage and arithmetic [18]. Fixed point definitions can be represented as ”Q[I].[F], where a 2’s compliment binary number of consisting of $I + F$ bits represents a integer dynamic range defined by I bits and a fractional precision of 2^{-F}. By utilizing fixed point representations, BRAT is able to use integer arithmetic hardware instead of the more costly floating point operations traditionally used in software defined neural networks. While fixed point values do not represent the same level of precision or dynamic range as floating point values, initial experiments showed that the accuracy of the network is not significantly impacted by the change. As neural networks are used to approximate relationships, they are tolerant to less precise values and arithmetic. According to [19] it is well known that deep networks are able to achieve similar levels of accuracy using 16 bit fixed point values compared to 32 bit floating point values.

CHAPTER 3
NEURAL BRANCH PREDICTION

BRAT is a robust network that can be applied to branch prediction by selecting appropriate inputs and a topology which allows for accurate and timely predictions. BRAT is built on a neural network with a forward propagation pass, used for inferencing, and a back propagation pass used for updating weights. This section discusses the high-level description of the BRAT architecture.

3.1 Key Terms for a Multi-Layer Neural Network

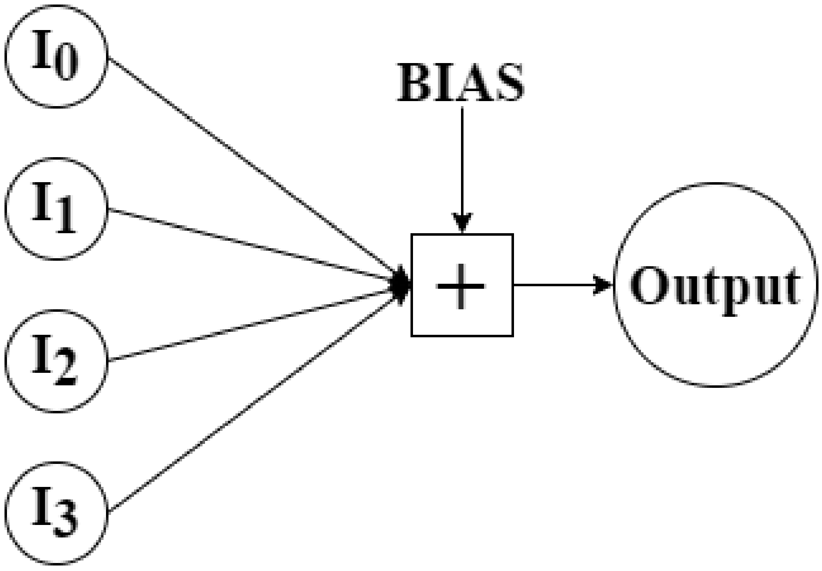

Figure 3.1: Perceptron

3.1.1 Fully Connected Layer

A Fully Connected Layer is the core of a neural network where each input into the layer is multiplied by an associated weight before they are added together. In this way, it mimics a linear function; keeping with this analogy, there is an additional bias for each output neuron as well which can be thought of as the constant. The simplest example of a fully-connected layer predictor is the perceptron as defined by [9] which uses the bits of the GHR register as bipolar binary inputs and uses the sign bit of the resultant value to determine whether or not a branch is predicted as taken as shown in Figure 3.1.

3.1.2 LeakyReLU

ReLU is an activation function in a neural network that helps introduce non-linearity. It takes the positives as themselves, and the negative values as 0. LeakyReLU is a slight modification of this where it takes the negative values as a very small positive constant times the negative input in order to make sure that the network doesn't get stuck, known as the vanishing gradient problem.

3.1.3 Max-Pooling

Max-pooling is the act of using the max between neighboring inputs. Max-Pooling acts beneficial in two ways: firstly it reduces the computations required for the next layer, and secondly it can help in capturing non-linear relationships.

3.1.4 Sigmoid

Sigmoid can be considered a simple mapping of any number into the range (0,1). This function can be especially helpful at the end of the network for binary classification problems such as branch prediction in order to map any network output onto a clear range between two classes.

3.2 The Layout of BRAT

3.2.1 Inputs to the Network

BRAT uses two inputs into the network instead of one. The Global History Register (GHR) is one of the primary inputs, the GHR is used to provide a significant portion of the data for the input vector into the predictor. The Program Counter(PC) was originally used as another portion for the input vector, but BRAT performs better when using the PC to index into a local history table and using those bits as the other portion of the input vector; this completely decouples the PC from the inputs into the network. The PC is, however, still used to index into a table of neural networks in order to lessen the load on any single network and increase the per network accuracy. To create the input vector into the network, the local history and global history are fed into the network with each single bit acting as a single input into the network creating a bit vector, similar to the perceptron predictor [9].

3.2.2 Topology of BRAT

The Topology of BRAT is shown in Figure 4.1. Following this Topology, BRAT starts with a single fully-connected layer. After this layer, LeakyReLU is implemented. After the LeakyReLU layer, there is a sum-pooling layer. Finally, there is another fully-connected layer. Then there is a sigmoid function at the end of the network.

3.2.3 Online Training vs Offline Training

The online training of the network is the most complex portion of the architecture; however, it is also the most essential part of the network. While other papers [5] have shown the potential of offline training, and currently the results of offline training on the network are only slightly below that of online training, offline training is not a viable approach for a main branch predictor solution, which this research is targeting, at this point in time. This is due to two main reasons: offline training as a primary predictor requires an initial training session where the entire work-load would be put on the secondary predictor for the initial run-through and training time is high. Currently, it takes the offline version of the network around 8 hours to train for 100M on a 1 core CPU. BranchNet [5] required anywhere from 6 to 18 hours across 4 GPUs for their training. For these reasons, offline networks currently show potential, but for now remain primarily as supplementary predictors.

BRAT is able to accomplish online training by utilizing a Sigmoid activation function on the output of the forward propagation and then using Binary Cross-Entropy loss function in order to train effectively for branch prediction.

3.3 Forward Propagation

Forward propagation is the process of taking the input vector and turning it into a prediction. This means that the critical path to get a prediction is through the forward propagation. The output after going through the forward propagation is a value between 0 and 1, where values $>= 0.5$ classify as taken and < 0.5 classify as not taken. This mapping between 0 and 1 is handled by the Sigmoid activation function. While the Sigmoid activation function is considered part of the forward propagation from a software perspective, in hardware it is handled after the prediction is made since the classification is known based on the sign of the input fed into the Sigmoid layer. As such, Sigmoid is discussed in the back-propagation section of the architecture.

3.4 Backpropagation

Back-propagation is the process of updating the weights in the network. In this process, BRAT calculates the loss of the prediction and propagates it backwards through the network in order to update the weights for the fully-connected layers. Figure 4.2 is an example of the error calculation process. The T node represents the true taken value from the pipeline, the S and S' nodes refer to the Sigmoid and its derivative, the L nodes represent the loss values, which are then used to calculate the Error which is stored in the E node. From here, the error is used throughout the Back-propagation in order to calculate how much to modify the weights by and update them. This process and the relevant optimizations are detailed in chapter 4.

CHAPTER 4

ARCHITECTURE OVERVIEW

A major issue with implementing neural networks in hardware is the expensive cost of the hardware. This section addresses the optimizations and costs associated with the BRAT architecture.

4.0.1 Hardware Optimizations for Forward Propagation

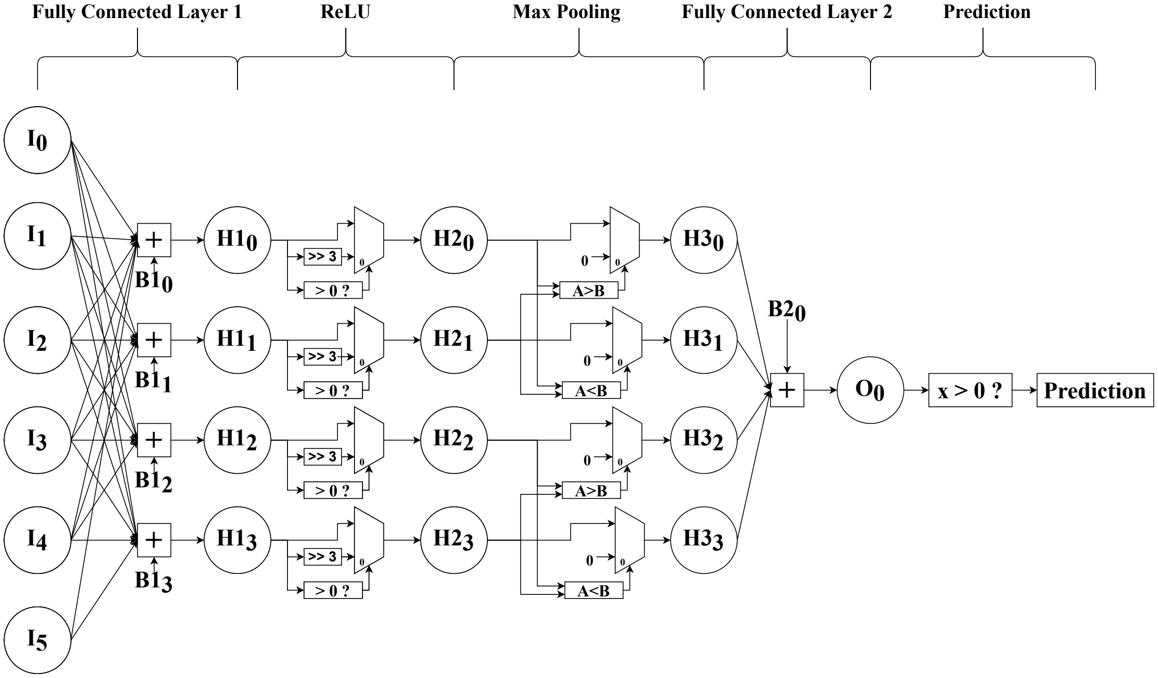

Figure 4.1: A Neural Network with ReLU Activation Layer

An important thing to note with BRAT going forward in the forward propagation is that, all mathematical operations are performed with Q6.6 fixed point representation. While the backpropagation and weight storage is done in Q6.18, the forward propagation needs to be fast and thus 6 bits of integer precision and 6 bits of float precision limit the representation range to numbers between (-33,32) and any value that would exceed this bound will be truncated to either -33 or 32.

Referring to Figure 4.1, the edges from the the input nodes I to the wide input adder signify a multiplication with a unique weight for each edge. As the inputs are binary, this logic reduces to a basic multiplexer or an AND gate such that when the input is 1, the output of the edge is the weight itself and otherwise it is 0. The adder before each hidden node H1 is comprised of a wide adder tree built using carry save adders combined with carry look-ahead adders such as the Kogge Stone Adder. [20].From here, the sign bit is used to select between the input value, if positive, and a small constant times the input value, if negative. This function is LeakyReLU, and in order to implement this efficiently, the constant is chosen as a power of two$(1/8)$; this reduces the multiply to a bit shift while maintaining the benefits of LeakyReLU as opposed to ReLU. Once the LeakyReLU values are stored in the H2 nodes, they are max-pooled,in groups of two, and multiplied. The calculation of max pooling is done by subtracting one grouped value by the other and taking the sign. For timing, this is done in parallel with the multiplication of each of the inputs by a weight. Once the multiplied values are computed and the appropriate ones selected according to max pooling, they are added together and the sign bit of the resultant computation is taken to produce the prediction value. From here the output is sent into the error calculation phase of the back-propagation where the Sigmoid and update calculations will be performed.

4.0.2 Pipelined Implementation of Forward Propagation

The logic for the forward propagation is quite significant and BRAT requires at least 3 cycles to compute the prediction once the inputs are known. Using Cadence Genus, BRAT is able to operate at a 2.5GHz frequency. The pipeline register is immediately following the ReLU stage of the forward propagation; This results in a 2 cycle implementation. However, as the local history must be retrieved from the 512 entry Local History Table and the weights must be retrieved from the correct entry in the network table, there is likely an additional cycle of latency to prepare the inputs for the branch prediction engine. This results in a 3 cycle prediction latency which is comparable to the latency of TAGE-SCL which is 4 cycles of latency.

4.1 Backwards Propagation

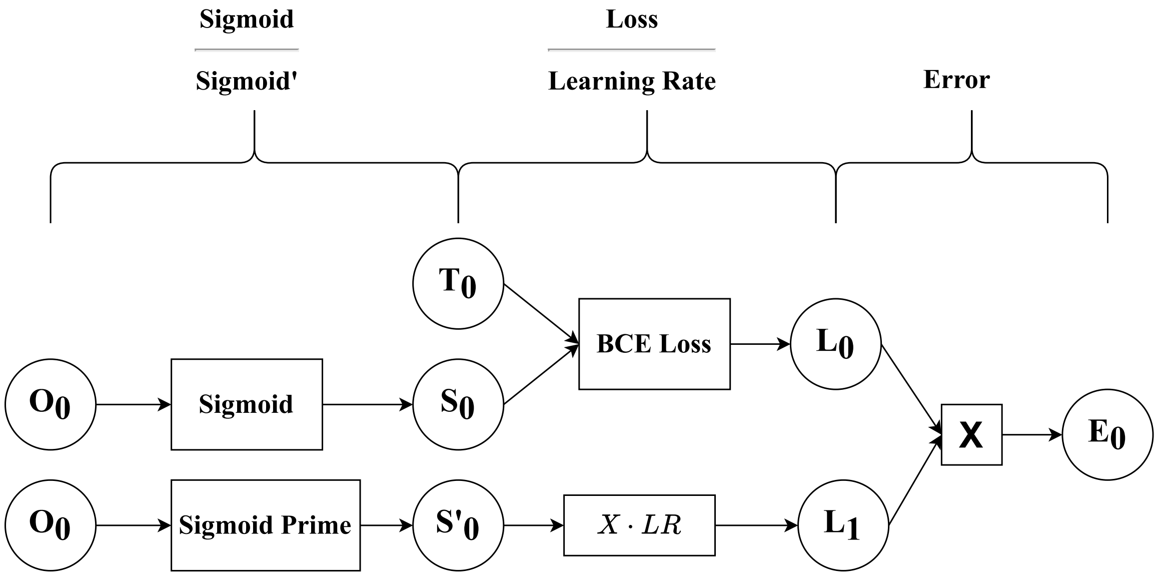

Figure 4.2: Error calculation flow

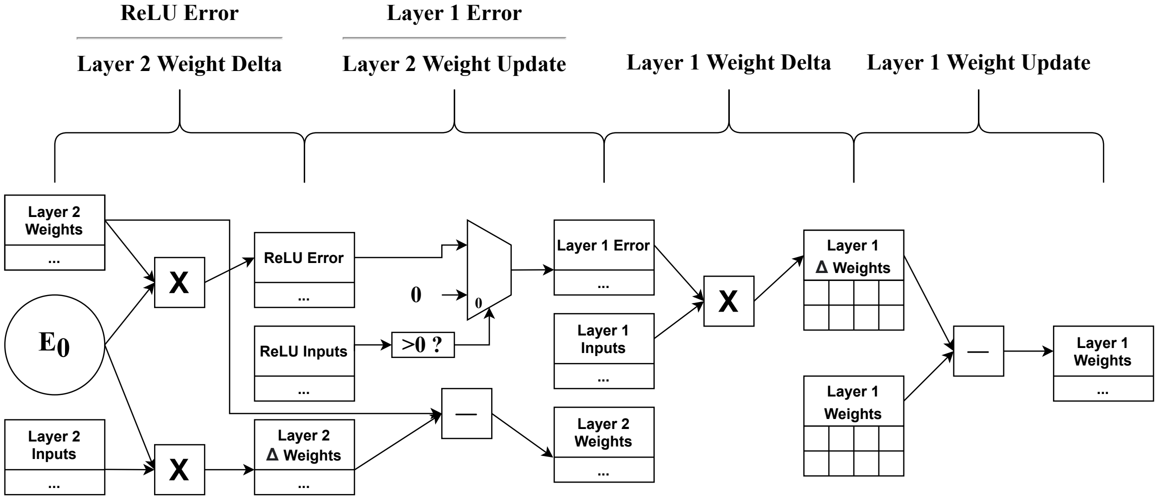

Figure 4.3: Training

4.1.1 Backwards Propagation and Optimizations

Back-propagation is an expensive procedure due to the large number of multipliers and adders needed to tune and update the weights. Multiplying every input by the associated error would be infeasible in the case of the first fully connected layer, and the activation functions such as Sigmoid and Binary Cross Entropy (BCE) Loss are far too expensive to implement directly in hardware. There are several optimizations that must be made in order to mitigate some of the cost of back-propagation and allow us to reasonably implement it as a circuit. The optimizations can be followed along in Figure 4.2 and Figure 4.3.

The first optimization moves the application of the Sigmoid from the forward propagation pass to the more latency tolerant backward propagation pass. The Sigmoid activation function (Equation Equation 4.1) is applied to the output of the forward propagation in order to train the network for a binary decision (output). To adapt this to physical hardware, BRAT uses a piece-wise linear function that approximates the Sigmoid function. In parallel with this stage is the calculation of the derivative of Sigmoid (Equation Equation 4.2) using the same technique. This reduces the complex exponential and division logic to a multiplier and an adder. The constant values for the slope-intercept form can be stored in a lookup table that can be indexed by the input value. The outputs of these calculations will be used in the backwards propagation and by calculating these in parallel, the latency of training is reduced.

$$\frac{1}{1+e^{-x}} \tag{4.1}$$

$$\frac{1}{1+e^{-1}}\left(1-\frac{1}{1+e^{-x}}\right) \tag{4.2}$$

$$\frac{-T_0}{S_0}+\frac{1-T_0}{1-S_0} \tag{4.3}$$

Following this, BRAT calculates the loss using BCE (Equation Equation 4.3), the most apt loss function for binary classification. However, BCE uses a divide which is infeasible for timely predictions. Furthermore, BRAT cannot calculate loss until the true label of a branch, whether it was taken or not taken, is known. BRAT is able to mitigate this again by applying a piecewise linear function for when the value is taken and a separate piecewise function for when the value is not taken. By speculating between these, BRAT is able to calculate loss before the label is known. Furthermore, by multiplying the sigmoid derivative by the learning rate, BRAT avoids having to apply the learning rate throughout the pipeline. Throughout the architecture, the learning rate is a power of two, specifically 2^{-7}, in order to have the multiply take the form of a fixed bit-shift. However, due to the wide output of multipliers, this shift can be accomplished by selecting different bits for the output of the multiply and avoid the shifting logic entirely.

After calculating the loss, the first error for backwards propagation is calculated by multiplying the previously calculated derivative of Sigmoid by the aforementioned loss.

Taking this value, BRAT moves to update the weights of the hidden layer. In order to do this, BRAT multiplies the current weights by the error, using as many multipliers as there are nodes in the hidden layer. BRAT then subtract these values from the weights to modify them. Due to the usage of maxpooling, only half of the weights will be modified in this step as only half of the weights for this layer are used in any given prediction. In parallel with this, BRAT multiplies the error by the H inputs in order to calculate the error for the next layer.

After this layer, BRAT applies the derivative of LeakyReLU, which takes as many two-input multiplexers as there are hidden layer nodes. After computing the errors for this layer, BRAT needs to initiate the final update of the weights for the first Fully Connected Layer. Since the inputs at this level are all either 0 or 1, BRAT can again take advantage of the AND gate to replace the multiplies here. After this, BRAT simply subtracts the errors from all the weights in this layer.

4.1.2 Pipelined Implementation of Backwards Propagation

The backwards propagation process is quite lengthy and will require many pipeline stages. The Sigmoid piecewise function will take 2 cycles to compute, which can be done in parallel with the Sigmoid derivative calculation. Applying the scalar factor of the learning rate can be optimized by changing which bits from the Sigmoid Derivative calculation are used for the resultant value. Calculating the BCE loss will take another 2 cycles, and the final multiplication for error calculation will take an additional cycle. As such the error calculation will take 5 cycles. The BCE loss calculation requires knowledge of the branch outcome and can stall the pipeline. Instead, BRAT can speculate the branch outcome and by replicating the BCE loss computation, BRAT can then utilize the true branch behavior to select between the two possible error values. In shallower pipelines, this optimization may avoid unnecessary stalls to the update process.

In the 6th stage, BRAT calculates the ReLU Error and the Δ that is to be applied to the weights for the 2nd fully connected layer. In the 7th stage BRAT subtract this Δ from the original weights to update them and use the sign bits from the ReLU layer inputs to calculate the error for the first fully connected layer. The 8th pipeline stage applies the first layer's inputs as and gates to generate the Δ matrix and subtract the value from the original weights. In total the backwards propagation should take 8 cycles to complete. Depending on the design constraints of a system, it may require up to 10 cycles to update.

4.2 Side-Effects of Pipelining BRAT

Due to pipelining in the forward and backward propagation, when two branches in close succession index into the same network, the second branch will use stagnant weights that would have been updated in a non-pipelined architecture, and will also update weights that have been changed from when they were last used. Stalling the forward pipeline until the backwards pipeline has committed would greatly hinder throughput for predictions and is not a viable solution. However, as this occurrence is rare and the low learning rate causes minimal changes in prediction accuracy, the BRAT architecture simply lets the second branch update the new weights.

4.3 Prediction Latency and Mitigation Techniques

When used in conjunction with pipelined processors, the BRAT is expected to have a response latency of 3 cycles. As it is proposed in this paper, the BRAT predictor should be used with a small bimodal table or G-Share predictor that it can override and correct predictions for. As mentioned by [4], ahead prediction [21] is also a viable method for reducing the latency of predictions by estimating the program counter of a branch instruction cycles before the branch is decoded or fetched. [22] also finds that using ahead prediction does not have significant impact on the performance of predictors. Combining ahead prediction with a small overridable predictor will allow for timely predictions without significantly impacting the accuracy of the prediction engine.

4.4 Training the HWNN

Due to the slow learning rate of neural networks, BRAT uses a bimodal or gshare predictor to assist the BRAT using a tournament style prediction inspired by the work presented in [23]. The tournament can thus be used to determine when the BRAT result should be used for overriding the result of the 2-bit counter table. Depending on the available memory budget, the gshare is more performant for larger tables and bimodal predictors perform better with smaller tables that experience more collisions. While the simple predictor is used primarily for the cold start of BRAT on context switches, BRAT is trained on all of the conditional branches encountered by the network. A small 256 byte tournament (2^{10} entry) is sufficient for this purpose. While the BRAT predictor may reach a stage where further training is not necessary, BRAT continues training on every conditional branch to ensure confidence and allow for the detection of changes in branch patterns and a timely correction. Since BRAT may learn slowly, it is necessary to learn at all times so that any deviations or changes in branching patterns can be learned from. Since the small secondary predictor is already required as a latency mitigation technique, it serves this dual purpose without increasing the hardware cost any more than is absolutely necessary.

4.5 Methodology

4.5.1 Traces Selected

Table 4.1: Traces from the Competition Branch Prediction Traces (CBP). Each trace has > 25M conditional branches. Traces are ordered by their performance on a 64KB gshare predictor.

Trace Name	Baseline Accuracy (%)
SHORT_SERVER_138	79.59
SHORT_SERVER_139	82.82
SHORT_SERVER_146	82.84
SHORT_SERVER_133	84.01
SHORT_SERVER_187	85.54
SHORT_SERVER_136	85.79
SHORT_SERVER_145	86.38
SHORT_SERVER_144	86.70
SHORT_SERVER_143	87.21
SHORT_SERVER_130	87.83
SHORT_SERVER_185	87.93
SHORT_MOBILE_16	88.09
SHORT_SERVER_134	88.14
SHORT_SERVER_162	88.25
LONG_MOBILE_8	88.50

To validate the architecture a trace based cycle accurate simulator is used that models each pipeline stage of the BRAT predictor. Traces are selected from the SPEC2017 Integer suite and the 2016 Championship Branch Prediction evaluation traces. Using a similar approach to Branchnet, experiments are done with a set of SimPoints [24] for each SPEC 2017

Table 4.2: Selected Traces from SPEC2017 Integer. Traces are selected such that enough SimPoint PinBalls are available to achieve $> 90\%$ coverage.

Trace Name
MCF
XZ
exchange2
gcc
leela
omnetpp
perlbench
x264

benchmark such that coverage of the program is 90+%. The $deepsjeng$ and $xalancbmk$ benchmarks are excluded due to incomplete SimPoints, Table Table 4.2 shows the selected SPEC2017 Integer Benchmarks. The results of simulations on these traces is a weighted average based on the proportion of the benchmark they represent. In order to select traces from the CBP 2016 suite, the 15 worst performers on a 64KB gshare predictor are selected that also have more than 25 million conditional branches executed. Table Table 4.1 lists the selected CBP traces.

4.5.2 Architecture

Considering memory footprints from 2K bytes to 256K bytes, the configurations that average the highest accuracy across the selected traces are selected. A table of these configurations is provided as Table Table 4.3. As mentioned in chapter 4, the configurations are limited to have at most 8 nodes in each hidden layer, which is achieved by having one-eighth as many hidden layer nodes as inputs. BRAT indexes into the table of networks using the lower bits of the PC excluding the lowest 2 bits as the experiments are run on x86 traces with variable instruction lengths. Predictors larger than 256 KB are not considered in the experiments due to the infeasibility of implementing such large predictors. With the

Table 4.3: Table of best performing configurations across all traces based on memory footprint, Bimodal Height $\geq 2^{14}$ uses G-Share instead

Size (KB)	GHR Bits	LH Bits	Bimodal Height	Networks
2	15	16	2^{12}	2^{2}
4	31	16	2^{12}	2^{2}
8	31	16	2^{14}	2^{2}
16	31	16	2^{14}	2^{4}
32	47	16	2^{16}	2^{3}
64	47	16	2^{17}	2^{4}
128	47	16	2^{18}	2^{5}
256	47	16	2^{18}	2^{7}

available overhead in memory footprint, it is possible that a TAGE predictor or perceptron predictor could be used in place of the bimodal table to regain some accuracy in future work.

Table 4.4: Size of Arithmetic Portion of BRAT Predictor

Portion of Network	32 input transistor count	32 input SRAM equiv.	64 input transistor count	64 input SRAM equiv.
Forward	46K	0.93KB	154K	3.13KB
Backward	161K	3.88KB	504K	10.24KB
Total	207K	4.81KB	658K	13.37KB

As a neural architecture has a significantly higher amount of arithmetic than historically popular branch predictors, even exceeding the arithmetic performed by the perceptron based predictor. As such, it is important to acknowledge the physical area of the prediction engine that is not attributed to the memory footprint. RTL models were created for the different portions of the BRAT predictor. Each module is synthesized and mapped using Cadence Genus to determine the utilization of the gates made available by the NanGate15 FreePDK[25]. These models do not include the memory footprints as these are dependent on the configuration and calculated using the CACTI [26] utility made publicly available by HP Labs. By using the gate counts and publicly available information on the transistors in each gate, the number of transistors is found for each model. In order to make this data easier to understand in relation to the other popular branch predictors, it is represented in terms of equivalent area in KiloBytes of SRAM using the 6 transistors per bit model. In order to account for the memory access penalties and the local history lookup, it is expected that a physical implementation would require 3 cycles to calculate a prediction at 2.5 GHz. To be fair in comparison, these hardware costs are added to the memory footprints in the graphs presented in chapter 5 which causes the minimum size to be 8 KB.

4.5.3 Other Simulated Predictors

An implementation of the perceptron predictor [9] and the Bi-Mode predictor[13] that accepts the traces selected is also simulated. In order to simulate the TAGE-SCL predictor[2], the simulator developed by the authors of BranchNet which builds off of an implementation submitted to the Championship Branch Prediction 2016 workshop is used. For the perceptron simulation, the table of optimal configurations for each memory footprint as described here [9] is used. The Bi-Mode predictor configurations are swept through in order to find the ones that perform the best on average across the workloads. While not actively explored, the Statistical Corrector (SC) and Loop Predictor (L) supplementary predictors could be applied to the BRAT predictor to improve performance in future works.

CHAPTER 5

EVALUATION AND RESULTS

5.1 Accuracy

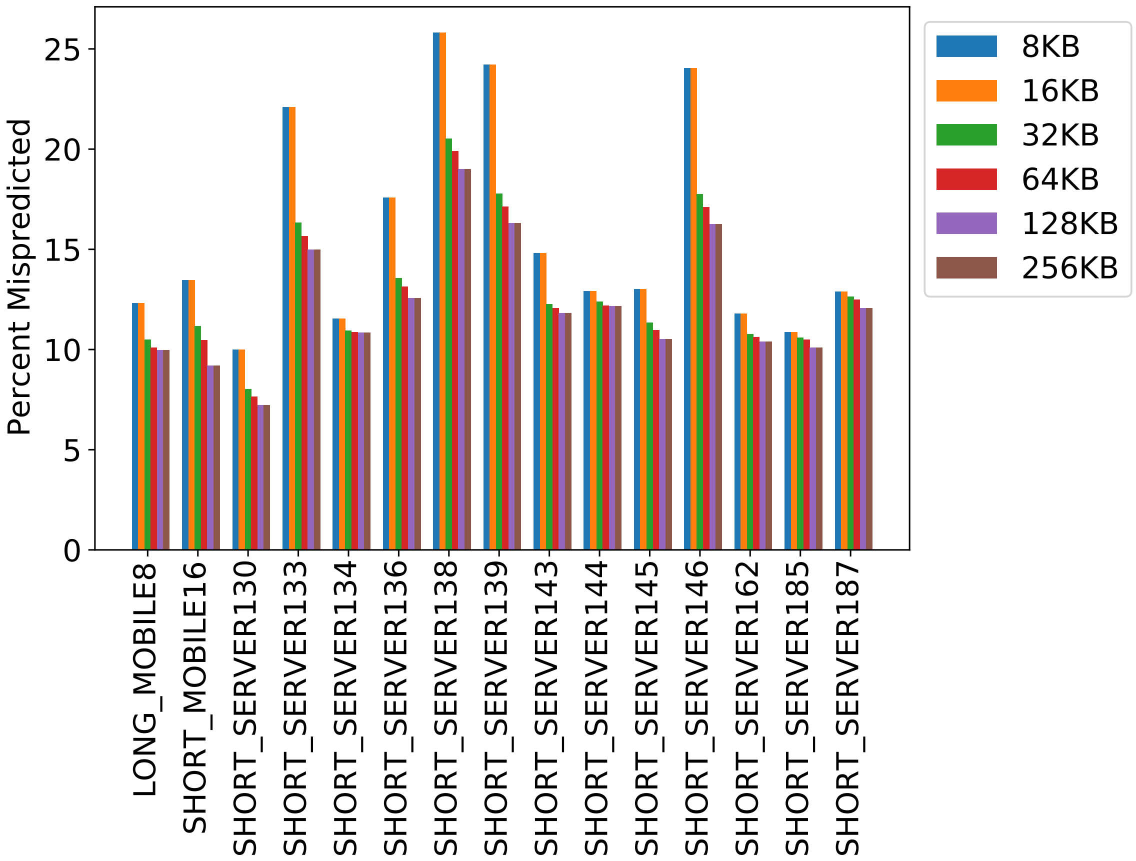

Figure 5.1: Accuracy of the BRAT predictor across the spread of CBP traces at various memory budgets. The best performing configuration for each memory budget is chosen for each benchmark.

Based on the misprediction rates in Figure 5.1 and Figure 5.2, the BRAT predictor is able to learn branch relationships quite well once it hits around 64KB. Due to the hardware costs, there are not configurations that substantially improve BRAT at lower KBs, and any values below 8KB are infeasible. It is apparent that for benchmarks such as `omnetpp`, `gcc`, and `x264`, increasing the size of the predictor does not substantially improve the

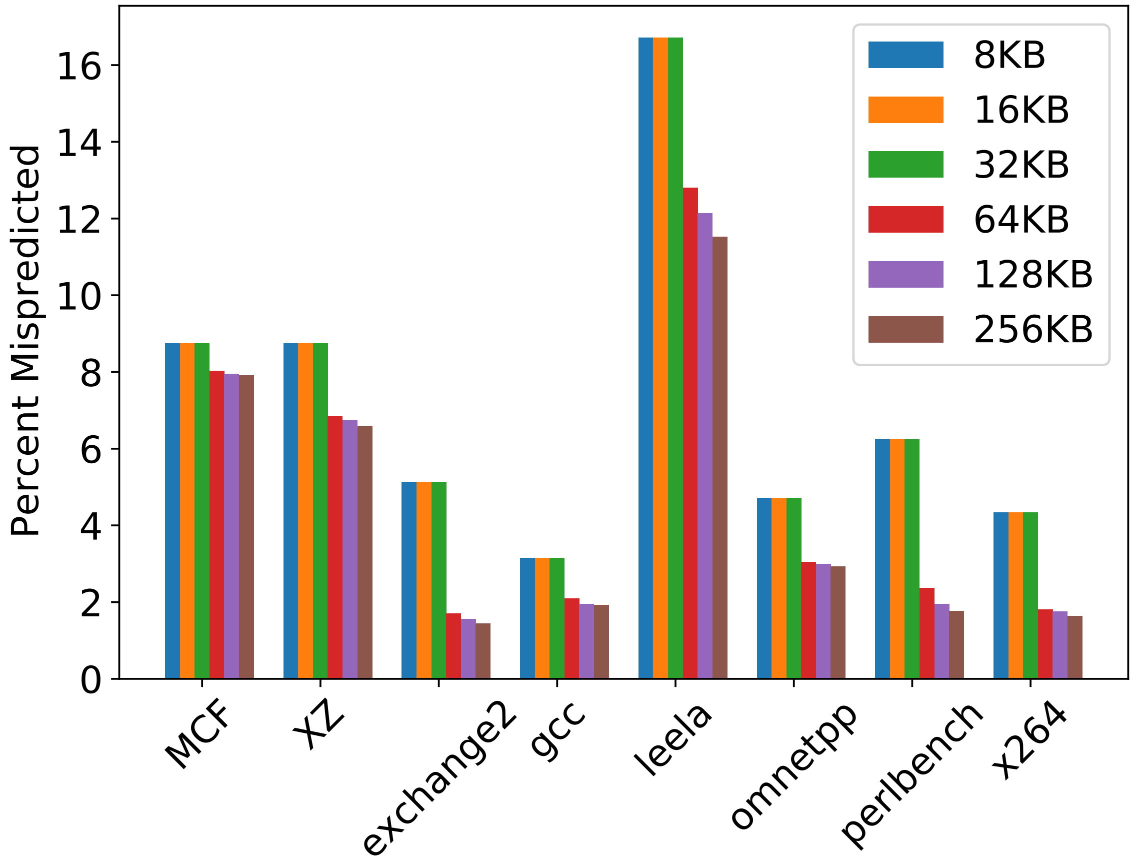

Figure 5.2: Accuracy of the BRAT predictor across the spread of SPEC benchmarks at various memory budgets. The best performing configuration for each memory budget is chosen for each benchmark.

predictor accuracy. However, the accuracy for the other traces improves significantly as the memory footprint is expanded. BRAT sees some diminishing returns as the size increases, though this is likely due to collisions in the network table. The best size that BRAT seems to perform at on a footprint vs accuracy improvement metric is 64KB.

As seen in Figure 5.4 and Figure 5.3, the performance of BRAT is competitive with state-of-the-art predictors across all of the traces. While the heavily optimized TAGE-SC-L does outperform BRAT, BRAT is extremely close on some traces such as in `omnetpp`. Unfortunately, TAGE-SC-L handily outperforms BRAT on smaller traces in CBP such as `SHORT_MOBILE16`. Still, across the suite of benchmarks, The BRAT predictor's performance scales strongly as the size of the predictor increases. Due to the large number of

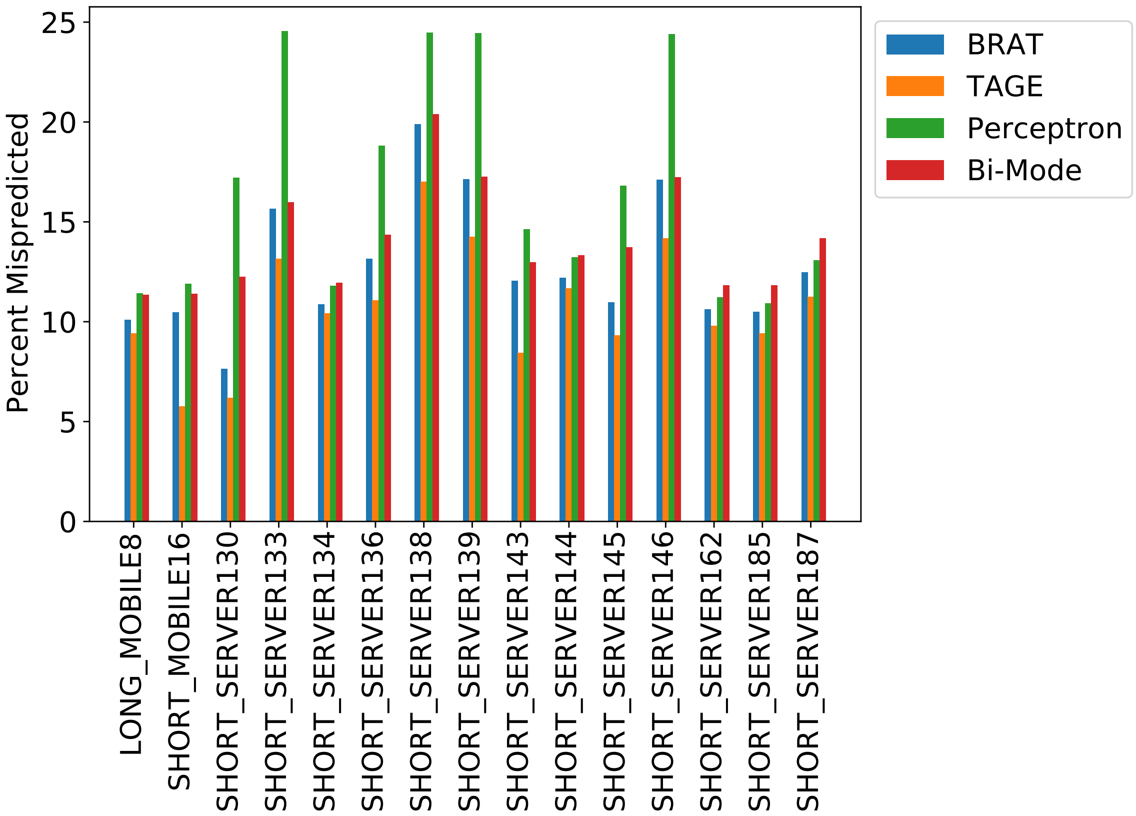

Figure 5.3: Misprediction rates compared to state-of-the-art predictors with a 64KB memory budget for CBP

weights in each network from the network table, each network is able to learn substantially more about the branching behavior it is responsible for predicting than the light networks of perceptron.

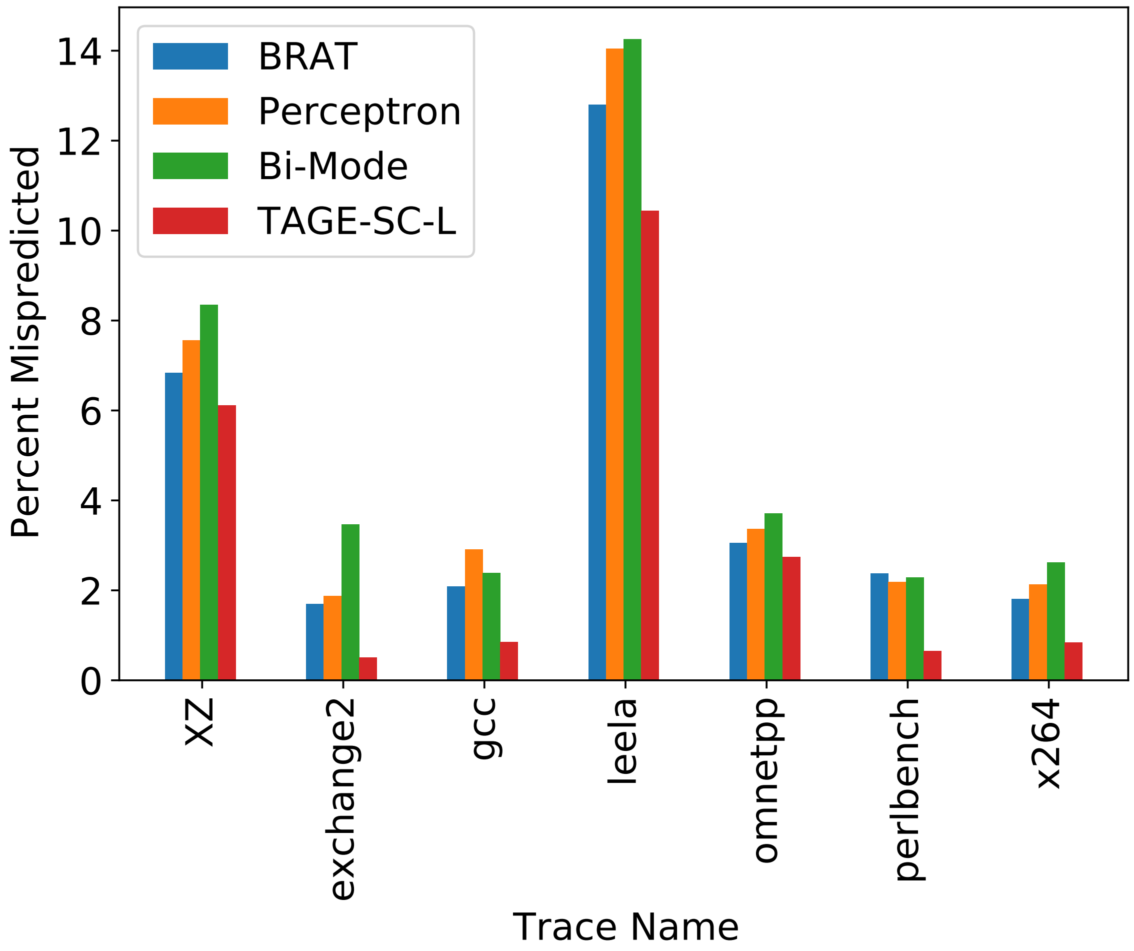

Figure 5.4: Misprediction rates compared to state-of-the-art predictors with a 64KB memory budget for SPEC

CHAPTER 6
CONCLUSION

On exceptionally hard to predict workloads, the BRAT branch predictor is competitive with TAGE-SC-L though slightly outperformed. BRAT shows strong promise on hard to predict traces and has an advantage in timing requirements over TAGE-SC-L which takes 4 cycles compared to BRAT's 3. The BRAT structure and its application to branch prediction pave the way for further research in online, multi-layer, neural network based branch prediction approaches.

The current implementation allows updates to be applied to stagnant weights as queuing updates would be costly. Future implementations may experiment with batched updates or aggregation to save power and reduce the complexity of queuing these values. Currently, BRAT uses a traditional binary definition in order to eliminate the 2's complement inversion that would be necessary along the input edges. However, should binary bipolar provide significant improvement, it is possible that a small modification to the pipeline and its' latency could increase performance substantially. Furthermore, this paper lacks supplementary predictors for BRAT that could be researched and added in the future. Another potential avenue of research is to create a hybrid BRAT-TAGE predictor. This kind of predictor would let TAGE handle most Branches, while BRAT would focus on learning especially hard to predict branches.

REFERENCES

[1] C.-K. Lin and S. J. Tarsa, "Branch Prediction Is Not a Solved Problem: Measurements, Opportunities, and Future Directions," *2019 IEEE International Symposium on Workload Characterization (IISWC)*, pp. 228–238, Nov. 2019, arXiv: 1906.08170.

[2] Seznec, "Tage-sc-l branch predictors again," in *5th JILP Workshop on Computer Architecture Competitions (JWAC-5): Championship Branch Prediction (CBP-5)*, 2016.

[3] A. Seznec, "The L-TAGE Branch Predictor," *J. Instr. Level Parallelism*, 2007.

[4] A. Seznec and P. Michaud, "A case for (partially) tagged geometric history length branch prediction," *J. Instr. Level Parallelism*, vol. 8, 2006.

[5] S. Zangeneh, S. Pruett, S. Lym, and Y. N. Patt, "BranchNet: A Convolutional Neural Network to Predict Hard-To-Predict Branches," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Athens, Greece: IEEE, Oct. 2020, pp. 118–130, ISBN: 978-1-72817-383-2.

[6] K. Hornik, "Approximation capabilities of multilayer feedforward networks," *Neural Networks*, vol. 4, no. 2, pp. 251–257, Jan. 1991.

[7] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," *Communications of the ACM*, vol. 60, no. 6, pp. 84–90, May 2017.

[8] M.-J. Kang and J.-W. Kang, "Intrusion Detection System Using Deep Neural Network for In-Vehicle Network Security," *PloS one*, 2016.

[9] D. Jimenez and C. Lin, "Dynamic branch prediction with perceptrons," in *Proceedings HPCA Seventh International Symposium on High-Performance Computer Architecture*, Monterrey, Mexico: IEEE Comput. Soc, 2001, pp. 197–206, ISBN: 978-0-7695-1019-4.

[10] M. Kim and P. Smaragdis, "Bitwise Neural Networks," *arXiv:1601.06071 [cs]*, Jan. 2016, arXiv: 1601.06071.

[11] R. St. Amant, D. A. Jimenez, and D. Burger, "Low-power, high-performance analog neural branch prediction," in *2008 41st IEEE/ACM International Symposium on Microarchitecture*, ISSN: 2379-3155, Nov. 2008, pp. 447–458.

[12] T.-Y. Yeh and Y. N. Patt, "Two-level adaptive training branch prediction," in *Proceedings of the 24th Annual International Symposium on Microarchitecture*, ser. MI-

CRO 24, Albuquerque, New Mexico, Puerto Rico: Association for Computing Machinery, 1991, pp. 51–61, ISBN: 0897914600.

[13] C.-C. Lee, I.-C. Chen, and T. Mudge, “The bi-mode branch predictor,” in *Proceedings of 30th Annual International Symposium on Microarchitecture*, ISSN: 1072-4451, Dec. 1997, pp. 4–13.

[14] A. Krall, “Improving semi-static branch prediction by code replication,” *SIGPLAN Not.*, vol. 29, no. 6, pp. 97–106, Jun. 1994.

[15] C. Young and M. D. Smith, “Improving the accuracy of static branch prediction using branch correlation,” in *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS VI, San Jose, California, USA: Association for Computing Machinery, 1994, pp. 232–241, ISBN: 0897916603.

[16] B. Calder, D. Grunwald, M. Jones, D. Lindsay, J. Martin, M. Mozer, and B. Zorn, “Evidence-based static branch prediction using machine learning,” *ACM Trans. Program. Lang. Syst.*, vol. 19, no. 1, pp. 188–222, Jan. 1997.

[17] J. R. C. Patterson, “Accurate static branch prediction by value range propagation,” in *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation*, ser. PLDI ’95, La Jolla, California, USA: Association for Computing Machinery, 1995, pp. 67–78, ISBN: 0897916972.

[18] E. L. Oberstar and O. Consulting, “Fixed-Point Representation & Fractional Math,” pp. 1–19, Aug. 2007.

[19] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan, “Deep Learning with Limited Numerical Precision,” *arXiv:1502.02551 [cs, stat]*, Feb. 2015, arXiv: 1502.02551.

[20] Z. Moudallal, I. Issa, M. Mansour, A. Chehab, and A. Kayssi, “A low-power methodology for configurable wide kogge-stone adders,” in *2011 International Conference on Energy Aware Computing*, ISSN: 2381-0947, Nov. 2011, pp. 1–5.

[21] A. Seznec, S. Jourdan, P. Sainrat, and P. Michaud, “Multiple-block ahead branch predictors,” in *Proceedings of the seventh international conference on Architectural support for programming languages and operating systems*, ser. ASPLOS VII, New York, NY, USA: Association for Computing Machinery, Sep. 1996, pp. 116–127, ISBN: 978-0-89791-767-4.

[22] A. Seznec and A. Fraboulet, “Effective ahead pipelining of instruction block address generation,” in *30th Annual International Symposium on Computer Architecture, 2003. Proceedings.*, ISSN: 1063-6897, Jun. 2003, pp. 241–252.

[23] S. McFarling, “Combining branch predictors,” Citeseer, Tech. Rep., 1993.

[24] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, “Automatically characterizing large scale program behavior,” in *Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, ser. ASPLOS X, New York, NY, USA: Association for Computing Machinery, Oct. 2002, pp. 45–57, ISBN: 978-1-58113-574-9.

[25] *Freepdk15*, https://www.eda.ncsu.edu/wiki/FreePDK15:Contents.

[26] S. Li, K. Chen, J. H. Ahn, J. B. Brockman, and N. P. Jouppi, “Cacti-p: Architecture-level modeling for sram-based structures with advanced leakage reduction techniques,” in *Proceedings of the International Conference on Computer-Aided Design*, ser. ICCAD ’11, San Jose, California: IEEE Press, 2011, pp. 694–701, ISBN: 9781457713989.