NEURAL METHODS FOR RESOLVING HARD-TO-PREDICT BRANCHES

A Dissertation
Presented to
The Academic Faculty

By

Pulkit Gupta

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

December 2021

NEURAL METHODS FOR RESOLVING HARD-TO-PREDICT BRANCHES

Thesis committee:

Dr. Thomas M. Conte
Computer Science
Georgia Institute of Technology

Date approved: December 10, 2021

ACKNOWLEDGMENTS

I would like to thank first and foremost Dr. Tom Conte, who is responsible for helping manifest my passion for this field and whose advice and insights were key to this work. Other members of the CRNCH lab were vital to this work – Jack Lafiandra, who laid the groundwork for this project and has provided keen insights at every step, Anirudh Jain, who was key to the simulation effort and provided a wealth of resources and time, and my remaining peers in the lab who acted as sounding boards through the entirety of my work and inspired confidence when problems seemed insurmountable.

I give special thanks to our friends at Northrop Grumman, Brian Konigsburg and Paul Tschirhart for providing funding for this research, and, more importantly, key insights and guidance that helped shape this work into its current form.

This work used the Hive cluster, which is supported by the National Science Foundation under grant number 1828187. This research was supported in part through research cyberinfrastructure resources and services provided by the Partnership for an Advanced Computing Environment (PACE) at the Georgia Institute of Technology, Atlanta, Georgia, USA. Without the computing resources offered by PACE, this work would not have been possible.

TABLE OF CONTENTS

Acknowledgments iii

List of Tables vii

List of Figures viii

List of Acronyms ix

Chapter 1: Introduction 1

1.1 The State of Branch Prediction 1

1.2 Neural Network Predictors 2

Chapter 2: Background and Related work 4

2.1 Classical Online Trained Predictors 4

2.1.1 Counter-Based Predictors 5

2.1.2 TAGE 5

2.1.3 Hashed Perceptron 6

2.2 Offline Trained Predictors 6

2.3 Combining Predictors 7

2.4 Fixed Point 7

Chapter 3: A Brief Overview of Neural Processes 8

3.1 Key Structures 8
3.2 Topology of a Shallow Neural Network 10
3.3 Input selection for Branch Prediction Applications 12
3.4 The Many Network Problem/Solution 12
3.5 High Level Design Concerns 13
3.5.1 Criteria for Training 13
3.5.2 Criteria for Utilization 13
3.5.3 Base Predictors 13

Chapter 4: Design of a Neural Architecture for Branch Prediction 15

4.1 A 128 MB Neural Network 15
4.2 A 16MB Model 15
4.3 A 64KB Model 16
4.4 Realizability of the 64KB Model 17
4.4.1 Forwards Propagation 17
4.4.2 Backwards Propagation 18
4.5 Side Effects of Realizability 19
4.5.1 Prediction Latency 19
4.5.2 Validity of Pipelining 19

Chapter 5: Methodology 20

5.1 Trace Selection 20
5.2 Cost of Realizable implementation 20
5.3 State of the Art Competitors 21

Chapter 6: Evaluations 22

Chapter 7: TAGE Filtering 27

Chapter 8: Future Work 30

Chapter 9: Conclusion 31

References 32

LIST OF TABLES

5.1 Selected Traces from SPEC2017 Integer 20

5.2 Area of 64KB Q-SON Predictor 21

7.1 Percentage of Mispredictions in groups of 4 tables 27

7.2 Percentage of Mispredictions for each Confidence Level 27

LIST OF FIGURES

3.1 Perceptron 8

3.2 A Neural Network with LeakyReLU Activation Layer and Max-Pooling . . 10

3.3 Error calculation flow for Sigmoid with BCE Loss 11

3.4 Training for Figure 3.2 12

3.5 A Simple Tournament Scheme 14

4.1 Network for 16MB configuration 16

6.1 Accuracy of the 128MB SON Predictor configuration compared to the state of the art 22

6.2 Accuracy of the 16MB SON Predictor configuration compared to the state of the art 23

6.3 Accuracy of the 64KB SON Predictor configuration compared to the state of the art 24

6.4 Accuracy of the physically realizable 64KB Q-SON Predictor compared to the state of the art 25

6.5 Performance of the (Q) SON predictor at different sizes 26

7.1 Performance of the predictor model at different configurations and the OBFN design 29

LIST OF ACRONYMS

BRAT BRanch prediction via Adaptive Training

GCHR Global Correctness History Register

GHR Global History Register

ISA Instruction Set Architecture

LCHR Local Correctness History Register

LeakyReLU Leaky Rectified Linear Unit

LHR Local History Register

MPKI MisPredictions per Kilo-Instruction

Q-SON Quantized Shallow Online Neural

ReLU Rectified Linear Unit

SON Shallow Online Neural

TAGE TAgged GEometric

SUMMARY

In this thesis a few neural-inspired branch prediction architectures are presented and discussed. Inspired by the successful Hashed Perceptron predictor, and digging further into the depth of machine learning, a multiple layer neural network is applied to the realm of branch prediction. These predictors are categorized as Shallow Online Neural (SON) Predictors as they make use of shallow neural networks and are trained online as opposed to a large number of prior neural inspired approaches.. The first model is a theoretical design that shows that a very large neural network with a single hidden layer is quite adept at learning branch patterns. This is due in part to the ability for neural networks to learn non-linear relationships and expose inter-branch correlations that may not be perceivable to TAGE or the Hashed Perceptron based predictor. The next model considers combining the large neural predictor with existing state of the art prediction mechanisms such as GShare, Perceptron, and TAGE predictors. A basic predictor selection mechanism is discussed and more complex mechanisms are explored. As a network's data storage budget scales linearly with the number of inputs, it requires substantial reduction in both size of network and number of networks to reduce the network's memory footprint to a reasonable size. Following this, substantial effort is made to transform the computational complexity of the network to a more physically implementable state. The final model is a physically realizable hybrid predictor that is competitive with the current state of the art in certain workloads. Key insights regarding classification of TAGE mispredictions are discussed and an alternative predictor is explored with aims to predict the correctness of TAGE rather than the outcome of branch instructions. The possibility and scope of further work that may be used to improve prediction accuracy, decrease implementation overhead, or further classify TAGE mispredictions is also discussed.

CHAPTER 1
INTRODUCTION

1.1 The State of Branch Prediction

Branch mispredictions continue to be a substantial limiting factor in the performance and energy efficiency of modern Out-of-Order processors with deep pipelines. Despite efforts to identify and classify Hard-to-Predict branches, performance is still limited [1]. Branch predictor designs have largely stagnated and recent approaches have only sought to augment the TAgged GEometric (TAGE) [2] predictor with correction mechanisms [3, 4], or with convolutional neural mechanisms [5] that must be trained on a per-program basis and only impact a small subset of branches.

The majority of branch predictors explored and in use today utilize an online learning approach, largely disqualifying approaches that follow the footsteps of BranchNet from likely inclusion in new processor designs. Online trained predictors largely fall into three categories, History-based, Tag-based, and Neural-based. All three of these categories operate using vectors based on global and local branch outcome histories and branch path histories to identify and learn any correlations with subsequent branch encounters.

- History-based predictors: These predictors use a function of the common input vectors to index into tables of counters
- Tag-based predictors: These predictors attempt to hash the common input vectors and store direction, confidence, and usefulness as entries
- Neural-based predictors: These predictors take inspiration from neural-networks and machine learning and apply them to the context of branch prediction.

Online predictors are able to quickly adapt to changes in program flow and path diver-

gence; however, they still have some shortcomings. The Hashed Preceptron predictor [6] struggles to scale to larger histories that TAGE is able to use and experiences substantial branch address aliasing on account of a limited number of perceptrons. Additionally, the Perceptron predictor is not able to learn non-linear branch relations due to fundamental limitation of perceptrons. The TAGE predictor is unable to capture some branch relations due to conflict misses in the tagged tables and capturing some relations that are not purely tag based. These realizations led to the development of BranchNet, an attempt to augment TAGE with a small convolutional neural network that is trained offline to resolve 16-48 hard-to-predict branches. However, due to the offline training approach, BranchNet requires extensive training times with multiple GPUs, apriori knowledge of the workload, and an exceptionally large storage budget (~ 1 Kilo-Byte per branch).

1.2 Neural Network Predictors

We define Neural Network Predictors as a multi-layer neural network with binary inputs and online training applied to branch prediction. In 1991, [7] showed that a feed forward neural network with multiple layers is capable of universal approximation, inspiring neural approaches in many domains, including image classification [8] and network security [9]. Applying binary inputs to neural networks allows them to be made more feasible in the computer architecture realm. Due to the importance of correct prediction and the ability to learn both linear and non-linear branch relations, neural network predictors are a good fit for this application. With adequate activation functions and well tuned hyper-parameters, it is highly likely that a large neural network predictor can learn *any* branch relationship given enough time.

The historic problem with the implementation of neural architectures for branch prediction and other schemes in a processor is the cost of implementation in terms of area and power. This is largely due to the cost of the back-propagation phase of training networks as this phase requires floating point arithmetic or very high precision fixed point arithmetic.

Many previous works have attempted to mitigate this [5, 10, 11], however they either require offline training [5, 10] or analog computation [11]. Analog computation would allow for substantially faster computation and lower area implementations, but remains infeasible due to the latency of the bi-directional Analog-Digital Conversions and the complexity of embedding analog computing hardware within a digital chip on the same substrate.

This thesis presents a neural network prediction architecture built off of BRanch prediction via Adaptive Training (BRAT) [12] and refines its design. The network is re-engineered to act as a supplement to a stronger primary predictor, namely the 32KB TAGE predictor. This new predictor retains the flexibility of online approaches while mitigating many overheads of offline training and large neural networks. This model takes inspiration from the Perceptron predictor by utilizing binary inputs and a table of networks.

First this paper will establish the background and related works for branch predictors (chapter 2). Then, chapter 3 gives a brief introduction to neural networks for branch prediction, followed by chapter 4, which goes through the design of three neural network models, concluding with a physically realizable model, the Quantized Shallow Online Neural (Q-SON) predictor. Experimentation and evaluation procedures are discussed in chapter 5 and chapter 6. An alternative application of the neural architecture is discussed in chapter 7, followed by future works in chapter 8 and finally the conclusion in chapter 9.

CHAPTER 2
BACKGROUND AND RELATED WORK

We categorize branch predictors based on how and when their state is modified or *trained*, namely Online-trained and Offline-trained. Online training has been the standard for branch prediction due to its ability to quickly respond to changes in patterns, however they struggle to capture extremely variable relationships based on complex or noisy histories; as such, they are unable to capture more intricate relationships despite ballooning in size. Offline trained predictors attempt to capture only these complex relationships and filter predictions based on these relationships. The offline approach works well, but requires immense training time and substantial memory footprint to make reasonable performance improvements. Additionally, current designs for these predictors require substantial Instruction Set Architecture (ISA) changes and do not yet have physical implementations.

The timeliness of predictions must also be considered, deep convolutinoal networks with high accuracy cannot be used in this context due to the very long latency of computation, instead prediction latency should be limited to 3-5 cycles in order to be reasonably used for branch prediction in modern multi-processors.

2.1 Classical Online Trained Predictors

Classical Online Trained Predictors are typically organized as tables of predictor entries indexed by a combination of Global History Register (GHR) and path histories in the form of Local History Register (LHR). Current state of the art predictors are built on TAGE and the hashed perceptron predictors.

2.1.1 Counter-Based Predictors

The simplest predictors are built on tables of 2-bit saturating counters, letting the higher bit provide a direction prediction for Taken (1) and Not-Taken (0). The Bimodal Predictor utilizes the branch address as an index into this table. This approach only considers the locality of the branch and as such suffers from *aliasing* and many entries are unused. In an attempt to reduce aliasing and establish a basic relation with global history, GShare computes an index by taking the xor of the branch address (Program Counter or PC) and GHR. Modern predictors do not use the counter based approach in entirety doe to XOR's inability to capture many relationships. The two level predictor [13] uses the PC to index into a table of local histories (LHR), then uses this local history as an index into the table of 2-bit saturating counters.

2.1.2 TAGE

The TAGE predictor uses a partial match compression technique to fold the PC, very long GHRs, and a small LHR into indices that are used to search a series of tables for an entry with a matching tag. Aside from the tag, predictor entries consist of wide counters, flags, and usefulness counters. Each table corresponds to a unique geometric history length, where there are more tables for shorter history lengths and fewer tables for longer more complex history lengths that are used when short history tables are unable to effectively capture branch behavior. When longer histories are necessary, whether from noise on account of unrelated branches or non-deterministic subsections of history being relevant, these long-history tables are oversaturated and suffer from capacity misses. These conditions result in the associated branches performing equivalently to a small 2-level predictor. TAGE-SC-L [4] is the best performing predictor in the state-of-the-art. It combines TAGE with a Loop Predictor (L) and a Statistical Corrector (SC). The loop predictor is used to count the iterations of a looping branch and establish a confidence in the number of iterations; this is helpful for filtering out and correctly predicting the outcomes of branches

in loops with a fixed number of iterations (matrix multiply, etc). The statistical corrector tracks and corrects errors in the base TAGE by using a small perceptron-like structure.

2.1.3 Hashed Perceptron

The hashed perceptron predictor uses perceptrons to compute the branch outcome. Perceptrons are the most basic form of neural architecture and capture linear relationships between the inputs by computing a linear combination of the inputs using a set of weights and applying a bias. However, it is fundamentally incapable of learning non-linear relations and suffers from the same issue of non-deterministic branch orderings.

2.2 Offline Trained Predictors

Offline predictors such as BranchNet utilize program traces and profiling to augment prediction accuracy for a baseline predictor [14, 15, 16, 17]. Other offline methods use the compiler to learn the statistical bias of certain branches or perform value range propagation to improve the performance of a predictor. Another example, the Spotlight predictor [18] augments GShare with useful history segments.

BranchNet uses traces from the runtime of a program to train a convolutional neural network (CNN) on up to 41 of the highest MisPredictions per Kilo-Instruction (MPKI) branches. A set of 100 highest MPKI branches are identified and used for initial training, of which up to 41 are then selected and encoded into the CNN based predictor. Then a filter is developed that is used to direct a very small subset of branches to the BranchNet portion at runtime. Branchnet is quite limited in that it can only learn a small number of branches, these branches each take $\sim$ 1 KB of SRAM based storage. The predictor as defined is unrealistic to implement in physical systems due to the prohibitively costly computation and the unlimited size TAGE that it is paired with. As mentioned before, offline training is not a solution for general purpose environments where many programs are constantly context switched and moved from core to core and apriori knowledge of

workloads and program conditions is not available. This style of predictor may be feasible for task specific systems, however other compiler efforts are likely to increase performance beyond the limits of BranchNet.

2.3 Combining Predictors

BranchNet is not the first prediction mechanism to combine 2 predictors, in fact a hybrid predictor approach was proposed in 1993 [19] and used in the Alpha 21264 processor [20] as combined a local history predictor (two level predictor) and a GHR indexed bimodal table, deciding between the 2 with a smaller counter table indexed by the GHR. The small counter table is used as a tournament. The Intel P6 processor also implemented a combined predictor approach inspired by the PAs predictor [21].

2.4 Fixed Point

Neural networks rely on floating point multiply and add units that are impossible to implement given the resource constraints of a branch predictor. In order to reduce the cost and latency of a neural network implementation, Fixed point representations are used [22]. Fixed point values 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, we avoid the cost of normalization and barrel-shifters present in floating point hardware and can instead use integer arithmetic. While these representations are unable to represent the dynamic range or arbitrary precision of sub-normal values, experiments in the context of this application indicate that substantial storage reductions and area reductions are feasible without sacrificing accuracy when using a 16-24 bit fixed point representation as opposed to a 32 bit floating point value.

CHAPTER 3

A BRIEF OVERVIEW OF NEURAL PROCESSES

A neural architecture is comprised of a series of layers. These layers can either perform a linear or non-linear operation. The layers are used in the forwards order for inferencing and computing a prediction, and in the backwards order for updating the weights and parameters of layers. This following section explains terms and features of neural networks that will be used for the specific topology used for this application.

3.1 Key Structures

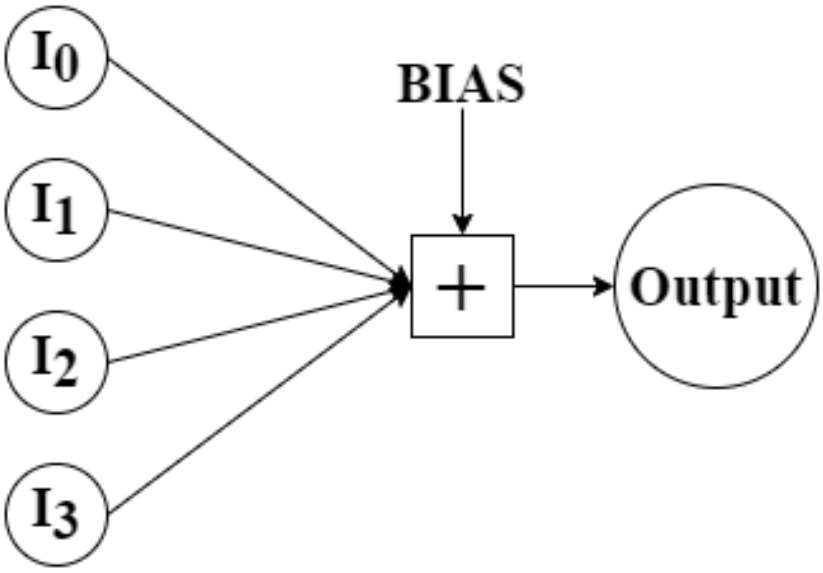

Figure 3.1: Perceptron

- Perceptron: A perceptron (Figure 3.1) is the smallest example of a neural process, it takes a vector of inputs, performs a dot product with a vector of weights and adds a bias component. the resultant value is the output of the process. These are the

fundamental piece of the Perceptron Predictor [6], which uses the bits of the GHR as an input vector.

- Fully Connected Layer: A Fully Connected Layer can be thought of as a set of perceptrons that are used to calculate many linear combinations based on their weight vectors.

- ReLU: Rectified Linear Unit (ReLU) is an activation function that introduces non-linearity by truncating all non-positive values to zero, leaving positive values unmodified.

- LeakyReLU: Leaky Rectified Linear Unit (LeakyReLU) is an activation function that introduces non-linearity by multiplying all non-positive values by a small constant, leaving positive values unmodified.

- Max Pooling: Max-Pooling is a layer that groups values into pairs, retaining the largest value.

- Sigmoid: Sigmoid is an activation function that performs an exponential based mapping of values in the real number space to the finite range [0,1]. This is especially helpful for binary classification problems where a network is tasked with producing a binary decision. this is well fit for applications such as branch prediction. For the sake of clarity, the equation for Sigmoid (Equation 3.1) and its derivative (Equation 3.2) are included.

$$\frac{1}{1+e^{-x}} \tag{3.1}$$

$$\frac{1}{1+e^{-x}}\left(1-\frac{1}{1+e^{-x}}\right) \tag{3.2}$$

- Binary Cross Entropy: Binary Cross Entropy (BCE) is a loss function well suited for binary classification. The Binary Cross Entropy Loss function's derivative is Equation 3.3 and while seemingly simple, utilizes division which is untenable for latency sensitive applications.

$$\frac{-T_0}{S_0} + \frac{1 - T_0}{1 - S_0} \tag{3.3}$$

- Learning Rate: The Learning Rate (LR) is a scaling factor applied to the error calculation used to prevent the network from over-correcting and converging on local minima or maxima.

3.2 Topology of a Shallow Neural Network

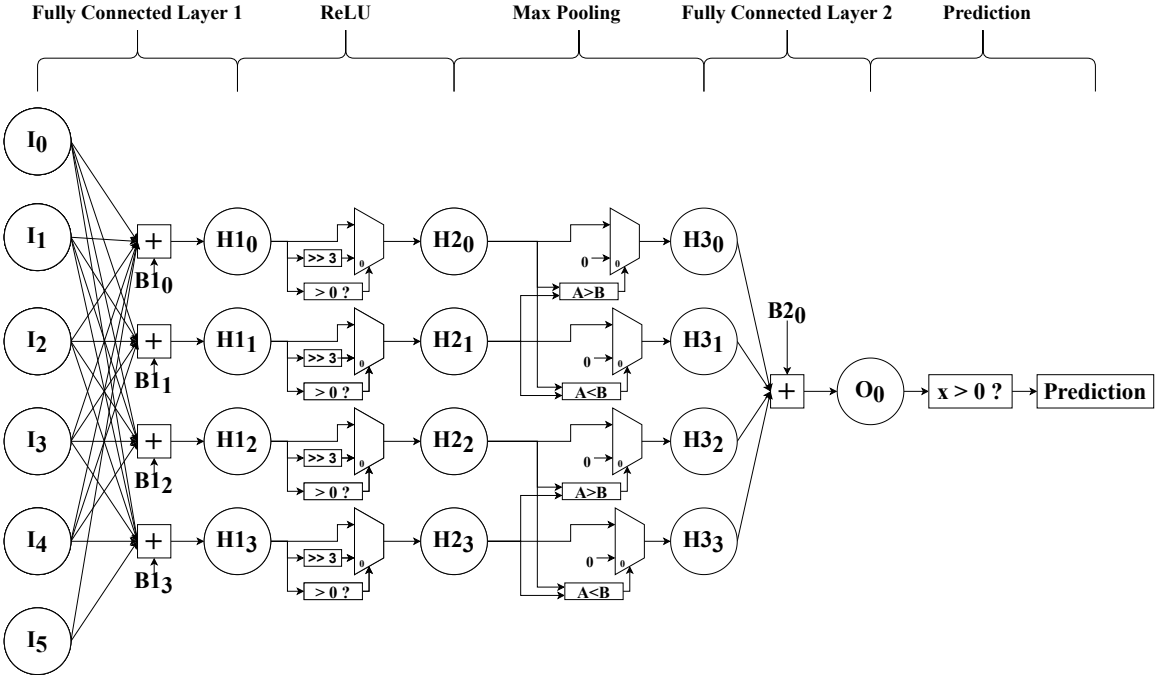

Figure 3.2: A Neural Network with LeakyReLU Activation Layer and Max-Pooling

Figure 3.2 depicts the Forwards Propagation for a neural network that will be the initial model for the branch prediction application. In order to provide clarity a heavy reduction in number of inputs is used to scale the number of nodes to a readable level. The first

layer is a fully connected layer, producing the vector of values *H1*. A LeakyReLU layer is applied with a leakiness factor of $1/8$ which is represented as a right-shift by 3, producing *H2*. After this a MaxPooling layer is applied that either allows a value to pass through or be replaced with 0, producing *H3*. The final Fully connected layer is applied condensing the output to a single value. Here, we can use the sign of the value as the output, in a traditional model, the sigmoid activation function would be applied before retrieving the output. However, because sigmoid is a zero-centered function, it can be applied afterwards.

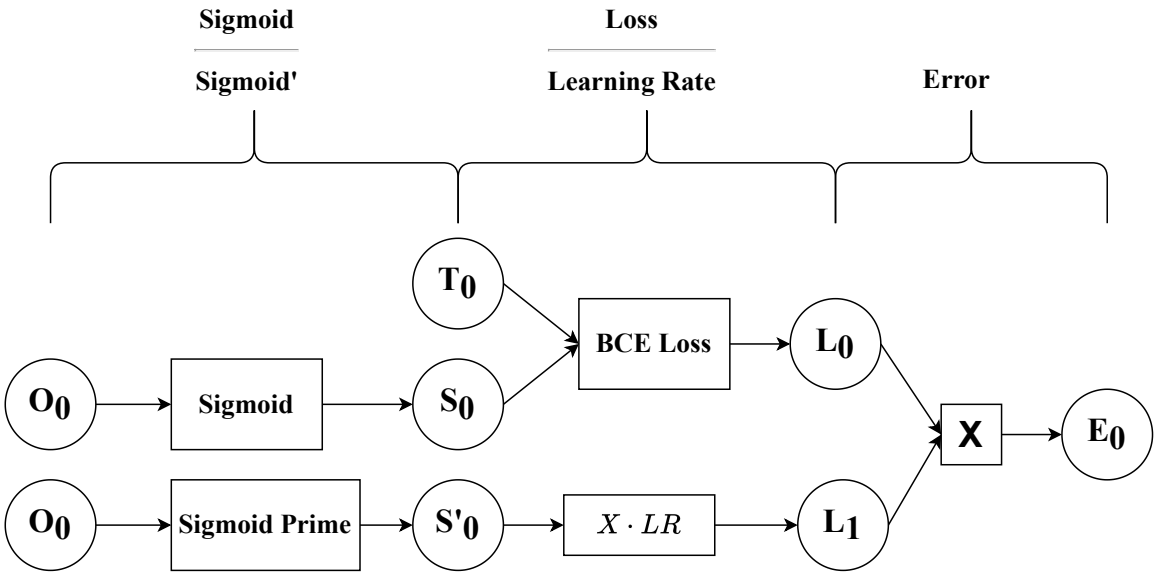

Figure 3.3: Error calculation flow for Sigmoid with BCE Loss

In Figure 3.3 first the sigmoid is computed and then the Binary Cross Entropy Loss is computed. In parallel, training values such as the derivative of sigmoid and learning rate are applied. A final error value E_0 is computed with a multiplication.

In Backwards-Propagation (Figure 3.4) the calculated error/loss is applied by taking the derivative of each of the layers and applying it to the weights to calculate the updated weights which will be used for subsequent predictions.

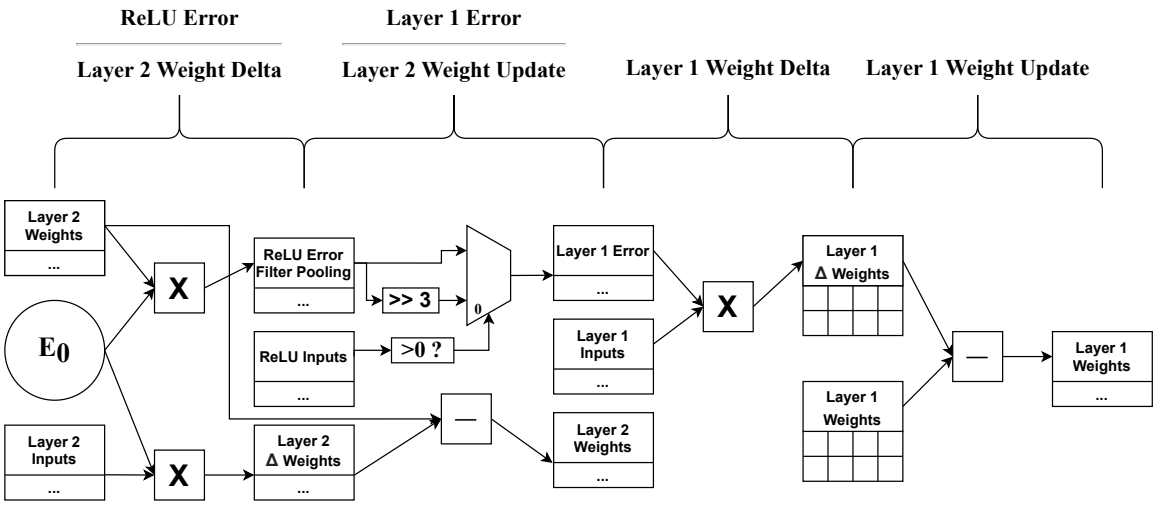

Figure 3.4: Training for Figure 3.2

3.3 Input selection for Branch Prediction Applications

For Branch Prediction, we utilize a combination of Global History and a table of Local Histories that will be used as a bit-vector input to the network. Like the Perceptron predictor, bipolar binary inputs may be used, meaning that in the input vector, 0 is replaced with -1. This allows not taken branches to provide a negative input to the network instead of having a non-impact as is the case with traditional binary inputs.

3.4 The Many Network Problem/Solution

The Perceptron predictor shows us that a single network is not capable of learning *all* branch behaviors and instead uses a table of networks such that the structure of the network is the same but many sets of weights are stored. The same logic can be applied here, however due to the width of weights and larger number of weights for a neural network, far fewer networks are used.

3.5 High Level Design Concerns

With a neural network design in place, the application and use of it is highly impactful to overall performance. Improper use of neural or overtraining could largely affect the accuracy.

3.5.1 Criteria for Training

The neural network can be trained in one of 3 ways:

- Train Always: The network is trained always despite the performance of the base predictor
- Train When Used: The network is trained only when the tournament or choice process selects the neural network
- Train When Almost Used: The network is trained when the tournament selects the neural network or when the tournament selects the assistive predictor with low confidence.

3.5.2 Criteria for Utilization

The neural network can be selected based on a small tournament as in [20] or based on other criteria. For example, misses in TAGE's tagged tables or when TAGE has low confidence. A basic example with a pc indexed 2-bit counter table can be seen in Figure 3.5.

3.5.3 Base Predictors

The Neural predictor is used in conjunction with a performant base-predictor. We consider utilization in conjunction with GShare and with 32 KB TAGE-SC-L.

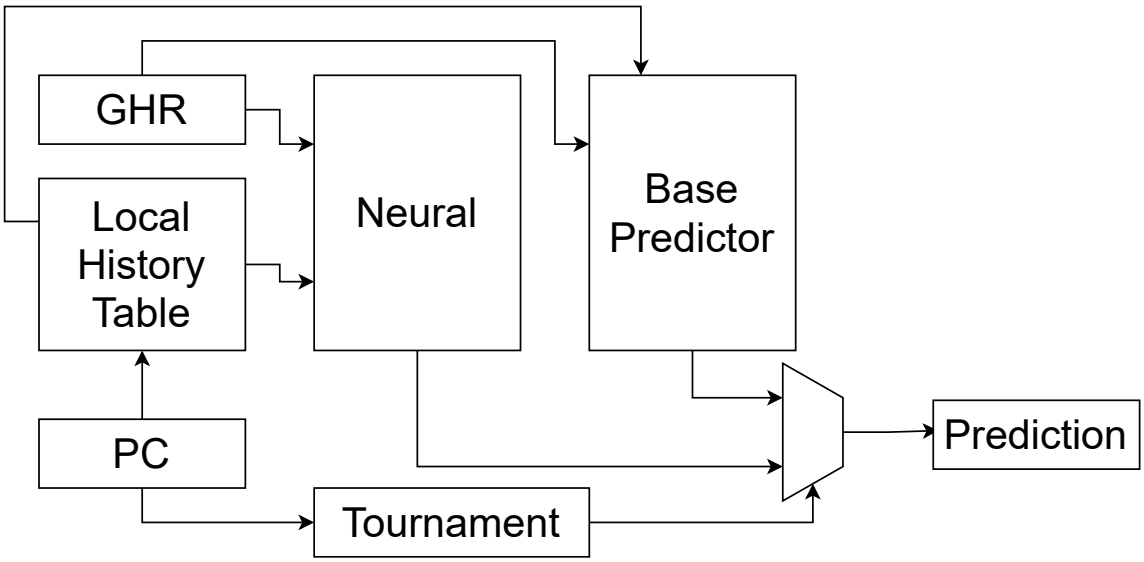

Figure 3.5: A Simple Tournament Scheme

CHAPTER 4

DESIGN OF A NEURAL ARCHITECTURE FOR BRANCH PREDICTION

In order to establish a theoretical bound for performance of a neural inspired predictor, we consider a very large neural network. This predictor fits the criteria of a Shallow Online Network (SON) Predictor. We slowly condense and improve on the network until we arrive at a physically realizable network.

4.1 A 128 MB Neural Network

We construct a SON network with inputs comprised of a GHR length of 128 and a 512 entry 32-bit Local History table. With 80 Hidden Layer Nodes, LeakyRelu with a coefficient of $1/8$, MaxPooling, Sigmoid activation function, Learning Rate of $1/64$, and full fp32 precision. While impossible to construct, this network allows us to explore the bounds of neural learning. This network follows the model of Figure 3.2 with 2^12 networks indexed by the PC. We also consider appending a 32KB GShare with a clairvoyant tournament. The tournament utilizes apriori knowledge of the branch outcome to decide best between the 2 predictors.

4.2 A 16MB Model

From the 128MB model we filter the structure down to use a 48-bit GHR and the same 512-entry 16-bit Local History Table, however, we introduce bipolar inputs as opposed to the normal inputs. This network has 16 Hidden Layer Nodes, LeakyReLU with a coefficient of $1/8$, Sigmoid activation function, Learning Rate of $1/64$, and full fp32 precision. There are 2^12 networks in this model (Figure 4.1). This network, while still unrealizable is a substantial reduction in size with similar performance characteristics. We pair this predictor

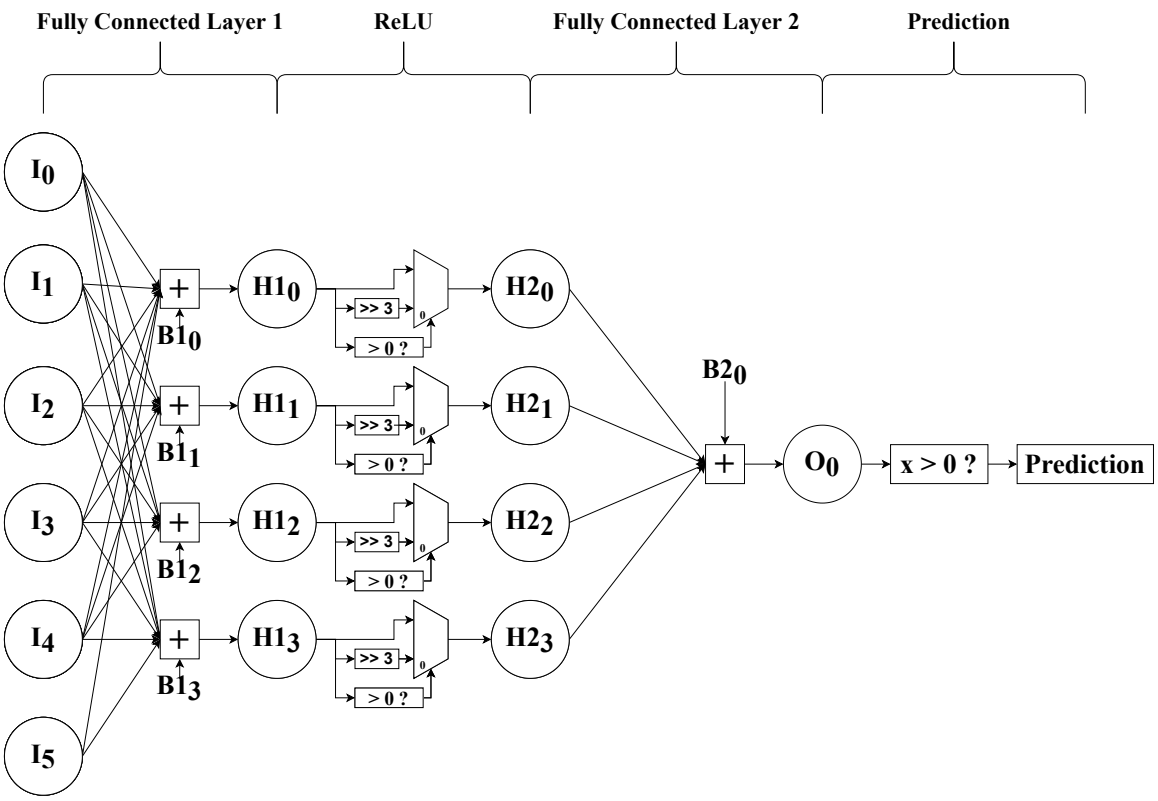

Figure 4.1: Network for 16MB configuration

with a 32KB GShare and a 32KB TAGE-SC-L and apply a tournament comprising of either a 2-bit counter table or using the confidence and table index from a TAGE hit.

4.3 A 64KB Model

This is the most realizable model, It is very similar to the 16MB model, but utilizes 2^4 networks and 8 hidden layer nodes instead. The neural aspect of the model uses $\sim$ 20KB of the area, the remaining area is occupied by a 32KB TAGE-SC-L. We find that the accuracy is largely unaffected despite the substantial increase in aliasing from the reduction in networks.

4.4 Realizability of the 64KB Model

The 64KB model now utilizes fixed point arithmetic to substantially reduce arithmetic hardware. In order to reduce latency, a Q6.6 fixed point representation is used for the forward propagation. A Q6.18 fixed point representation is used for the backwards propagation to maintain higher precision required for training. The Sigmoid activation function, derivative of sigmoid, and BCE loss function are approximated by using piecewise linear functions that eliminate all division and exponentiation replacing it with multiplication and addition. By quantizing the network for the forward pass and even with the backwards propagation steps and general weight storage, we transform the SON-Predictor into a Q-SON Predictor, or Quantized Shallow Online Neural Predictor.

4.4.1 Forwards Propagation

By using the Q6.6 fixed point representation, each computed value will be within the range (-33,32) and values that would pass the bounds will be saturated to the extremes. Interestingly, using binary and bipolar binary inputs allows for substantial simplification of the multiplication required in the first fully connected layer. With binary inputs, an input can simply be ANDed with the associated weight to accomplish the multiplication. For bipolar binary inputs, the logic is simply a multiplexer between the weight and its complement, the remainder of the computation for valid 2's compliment inversion can be precomputed with ahead prediction and added inline easily due to the commutativity of addition. The wide adder is comprised of carry look-ahead adders that are able to compute the sum quickly in a small timeframe [23]. The sign bit of this value is used to select between the value and $1/8th$ of itself, applying LeakyReLU. The resultant values are then multiplied by the Q6.6 weights and added to produce the output. The sign bit of the value is adequate for application as the prediction output. Sigmoid calculation is delayed until backward propagation to reduce the latency of forward propagation. A pipeline register is inserted immediately

following the LeakyReLU stage and another following the output calculation. Factoring in the weight access penalties, the design will take 3 cycles to produce an output. However, this will be resolved in subsection 4.5.1. The initial RTL model is able to operate the entire forward propagation at 2GHz in 1 cycle ignoring weight access penalties, but a more desirable 4GHz is attainable using the 2-cycle pipelined model.

4.4.2 Backwards Propagation

Backwards propagation is substantially more expensive than forwards propagation due to the large number of multipliers and adders required. The multipliers for the first fully connected layer would be prohibitively expensive to implement, thankfully a similar trick to the one applied for the forwards propagation can be applied. The Sigmoid activation functions and BCE loss function are prohibitively expensive to implement due to the exponential arithmetic and division operations. We replace these functions with piece-wise linear approximations stored in slope intercept form and stored in a small table. The sigmoid and its derivatives are calculated at the same time. A power of 2 learning rate is applied to the sigmoid prime calculation by selecting lower precision bits from the wide output of the multiplication. Instead of waiting for the true branch outcome T_0, both options are applied and speculation is performed through to the E_0 calculation and the correct value is selected when it is available. The error for each layer is now calculated and applied using the inputs to the layer at time of prediction and the weights. These values are maintained in the pipeline buffers. The inverse LeakyReLU is applied and the first layer is finally updated. Again, the AND gate/multiplexer trick is applied to reduce the number of necessary multipliers by 512. This design is pipelined into 8 stages to reach the desired 4GHz frequency.

4.5 Side Effects of Realizability

Disregarding possible arithmetic rounding errors due to the optimizations, there are 2 major concerns with the pipelined approach to the Q-SON predictor.

4.5.1 Prediction Latency

With substantial inference and update latencies there is a real concern regarding the timeliness of predictions. TAGE [2] utilizes ahead prediction [24] to search the tagged tables earlier than the branch address is known. This can be applied to the neural approach to also limit the impact of the latency. Additionally it is found that ahead pipelining does not substantially impact the performance of branch predictors [25] and thus allows for timeliness.

4.5.2 Validity of Pipelining

Pipelining the architecture introduces an interesting dilemma. When the network is used for inference, arithmetically, it should not be used for inference again until the weights have been updated. This would make the predictor usable only once every 10 cycles. This is clearly not desirable so we choose to allow weights to be modified by branches that occurred before their most recent update. This is an unlikely event that only occurs if more than one branch alias to the same network table index within a 10 cycle window. Experiments show that this has a marginal effect on the predictor accuracy 0.05%.

CHAPTER 5

METHODOLOGY

5.1 Trace Selection

Table 5.1: Selected Traces from SPEC2017 Integer

Trace Name	simpoints	32KB TAGE Accuracy	64KB TAGE Accuracy	64KB Perceptron Accuracy
MCF	14	7.32%	7.24%	8.53 %
XZ	19	5.63%	5.51%	7.48 %
exchange2	19	0.54%	0.45%	1.85 %
gcc	5	0.92%	0.85%	2.55 %
leela	29	10.05%	9.55%	13.73 %
omnetpp	5	2.63%	2.58%	3.46 %
perlbench	8	0.81%	0.65%	1.67 %
x264	10	0.84%	0.82%	2.03 %

Experiments are conducted using benchmarks from the SPEC 2017 Integer Suite [26]. We simulate using SimPoints [27] such that enough simpoints are used that 90+% of each trace is covered. If enough coverage is not achieved, the trace is not evaluated, *deepsjeng* and *xalancbmk* are not considered because of this. Table 5.1 shows the selected benchmarks.

5.2 Cost of Realizable implementation

In order to determine the area and resource utilization of the neural predictor, a cycle accurate RTL model was developed. We determine the area of computation elements because the cost of implementing multipliers and wide tree adders is substantially higher than the cost of the xors and adders used by TAGE and perceptron predictors. The model is synthe-

Table 5.2: Area of 64KB Q-SON Predictor

-	Forward Propagation	Backwards Propagation	Weight Storage	Base Predictor	Total
Size (Transistors)	154K	504K	-	-	658K
Size (Eq. SRAM size)	3.13KB	10.24KB	18.5KB	32KB	63.87KB

sized using Cadence Genus in tandem with the NanGate 15 FreePDK [28]. Combining the model with area characteristics from HP Labs' CACTI utility [29]. Converting the transistor utilization of the model to an equivalent amount of SRAM (Table 5.2), we are able to see that the area estimate is comparable to other 64KB predictors that we aim to compare against. The predictor model is synthesizable at 4 GHz for a 2 cycle prediction latency using naive libraries in RTL synthesis. Factoring in weight access penalties and local history lookups, it is expected that a prediction will be made in 3 cycles at 4 GHz. However, with more advanced gate libraries and other optimizations, the frequency can likely reach beyond 4GHz or require fewer back propagation stages. With ahead-pipelining, a prediction can likely be available within 1-2 cycles.

5.3 State of the Art Competitors

An evaluation of the state of the art predictors is made for use in comparisons with the model. The predictors selected are: 64KB TAGE-SC-L, 32KB TAGE-SC-L, 64KB Perceptron

CHAPTER 6

EVALUATIONS

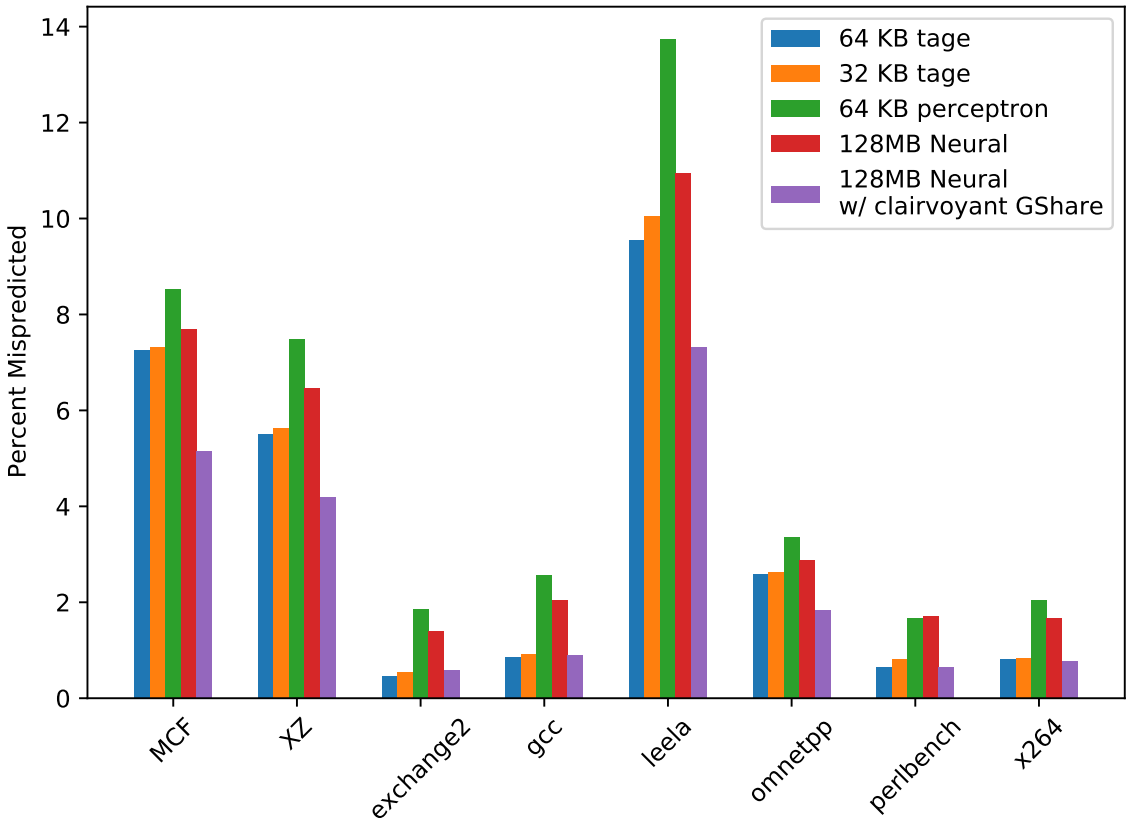

Figure 6.1: Accuracy of the 128MB SON Predictor configuration compared to the state of the art

In Figure 6.1 we see that the 128MB SON predictor is quite adept at determining the outcomes of conditional branches. Data considering a clairvoyant tournament with a small GShare is considered as well. We see with the clairvoyant tournament that an intelligent choice prediction is likely to substantially improve performance compared to the state of the art.

We explore the 16MB configuration when used in tandem with a base predictor. When GShare is used, the 2KB tournament is a table of counters indexed by PC. When TAGE is

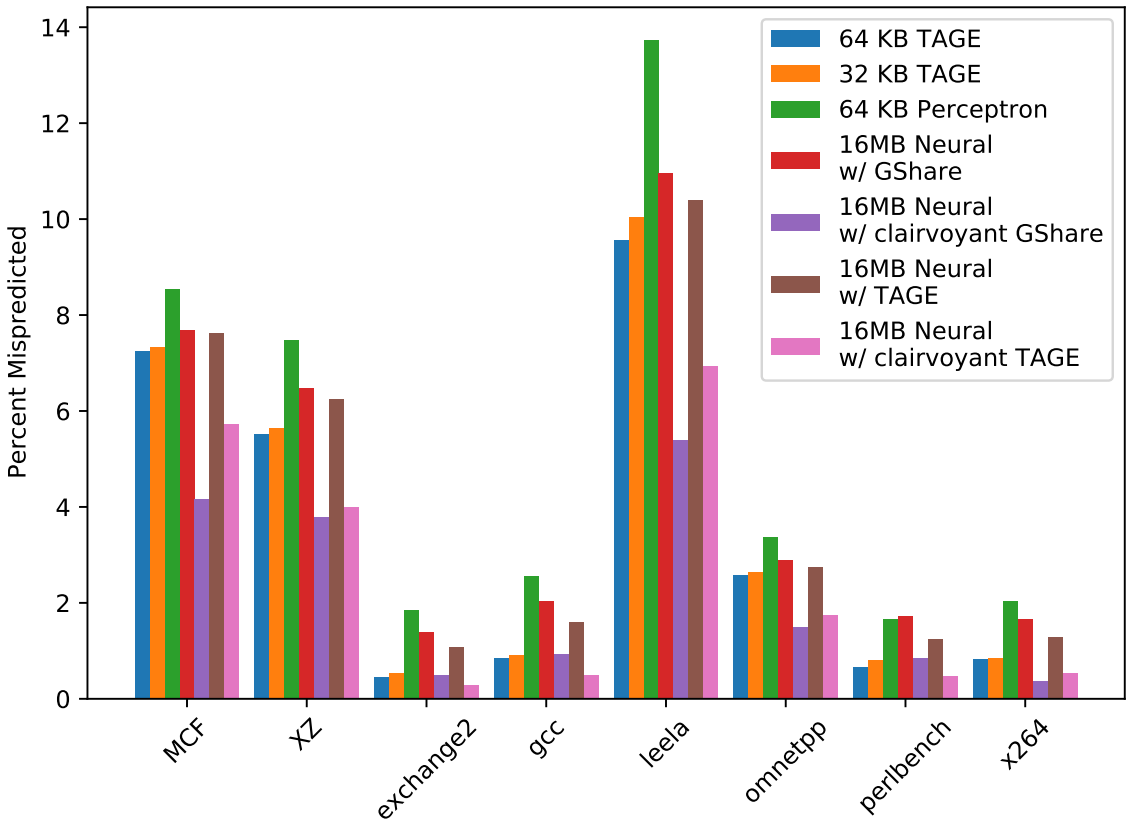

Figure 6.2: Accuracy of the 16MB SON Predictor configuration compared to the state of the art

used, the tournament is instead a logical function based on TAGE's confidence and whether or not there was a miss. We see in Figure 6.2 that the SON predictor performs admirably compared to TAGE32 but does not outperform the TAGE32 model by itself. This is entirely due to the tournament process as we see that a clairvoyant tournament gives admiral performance improvement.

The 64KB theoretical model performs similarly to the 16MB model based on Figure 6.3, indicating that this model is quite close to the lower limit for the computational complexity vs accuracy trade-off.

From Figure 6.4 we see that the 64KB Q-SON predictor performs quite admirably compared to the state of the art, especially considering the substantial arithmetic changes made to support physical implementation. However, it does not create appreciable deviations

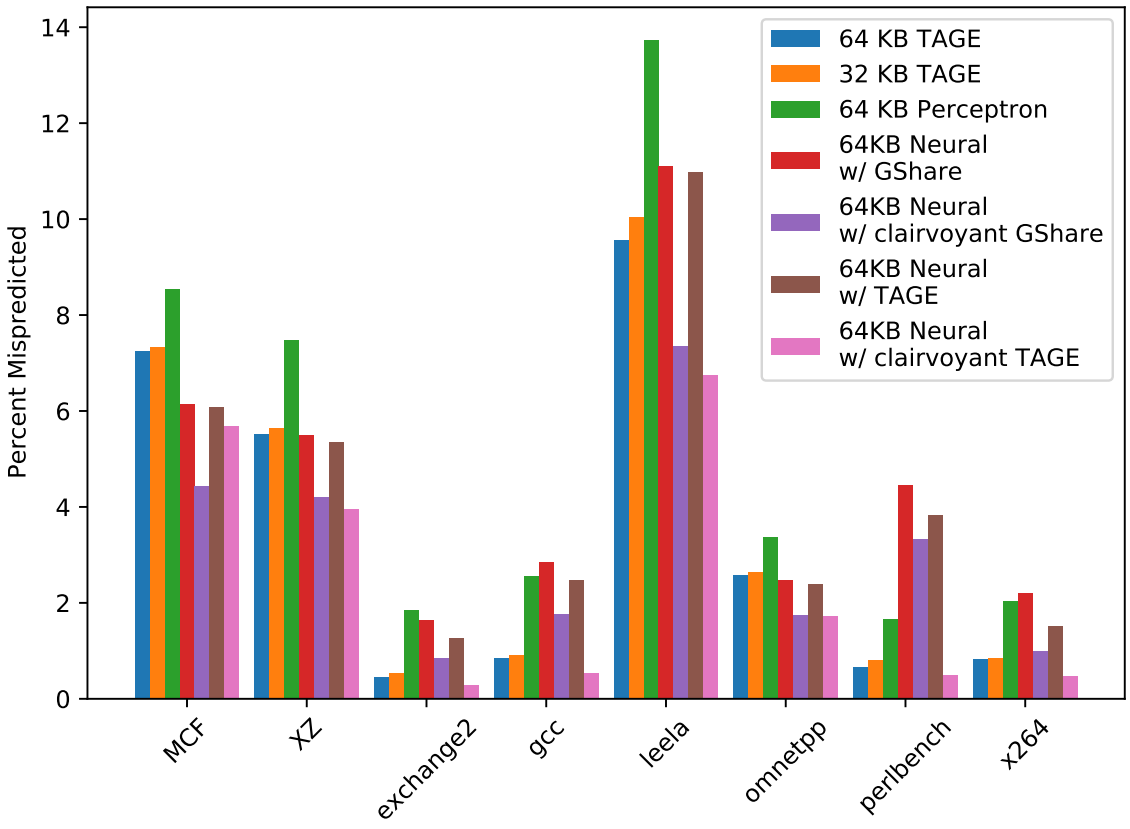

Figure 6.3: Accuracy of the 64KB SON Predictor configuration compared to the state of the art

in accuracy whether helpful or unhelpful. This is again likely due to the choice prediction mechanism. Here a clairvoyant tournament is not explored as it is not a physically realizable concept.

Based on Figure 6.5, we see that when easy to predict branches are filtered out, the cost reduction mechanisms are able to retain a substantial amount of the predictor performance.

Based on runtime analysis of simulations, we find that the neural portion of the predictor is used $\sim 30\%$ of the time, and the accuracy is not as high as would be desirable. With a more developed tournament, neural utilization should decrease and accuracy should increase. The ideal ratio for this will likely result in improvements over TAGE.

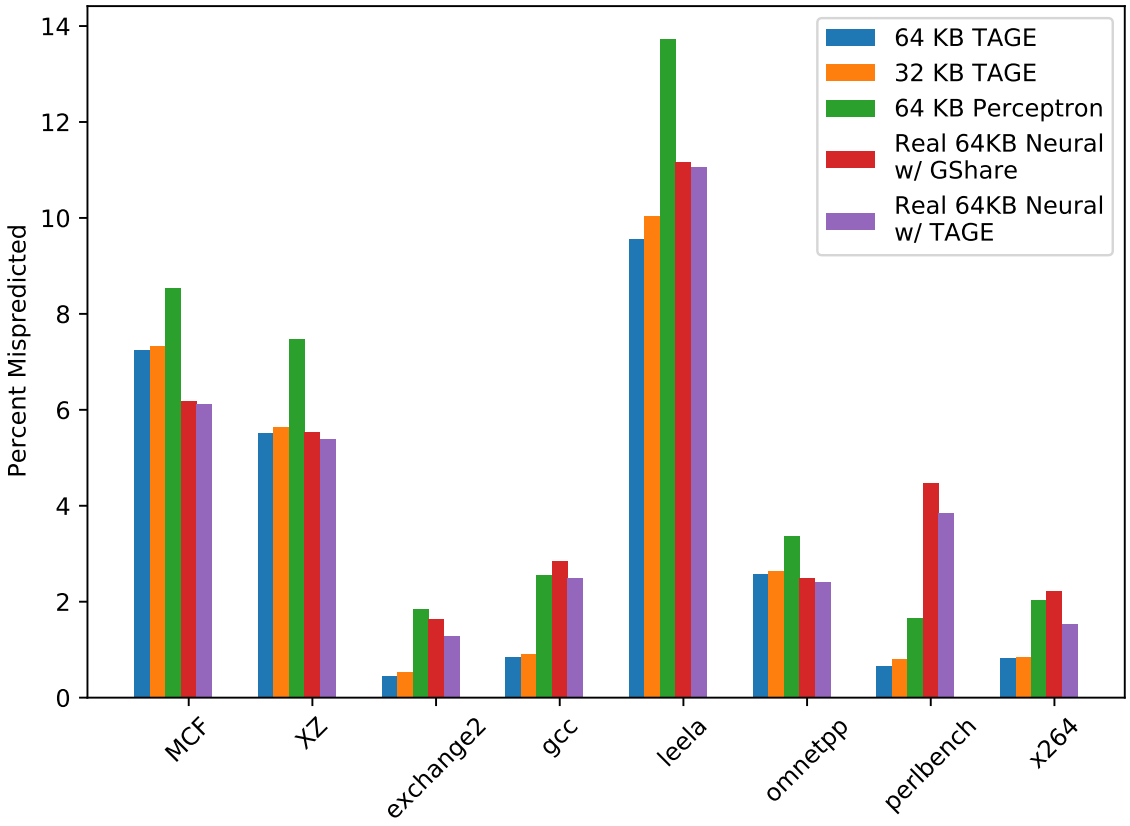

Figure 6.4: Accuracy of the physically realizable 64KB Q-SON Predictor compared to the state of the art

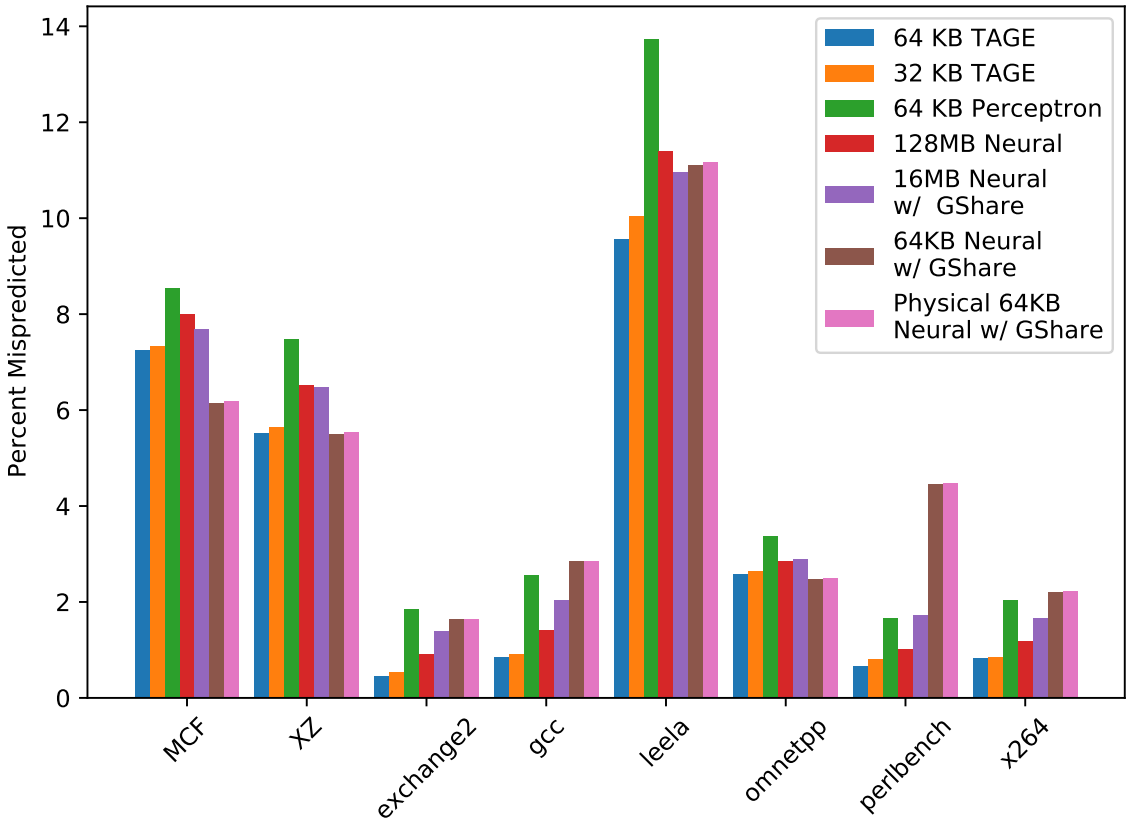

Figure 6.5: Performance of the (Q) SON predictor at different sizes

CHAPTER 7

TAGE FILTERING

It is apparent that determining when TAGE will be incorrect is a substantial factor in improving the performance of Branch Prediction. Some interesting data is collected from a subset of the SimPoints, amounting to 70% coverage of the entire SPEC 2017 integer suite

Table 7.1: Percentage of Mispredictions in groups of 4 tables

	0	1	2	3	4	5	6	7	8
Mispredictions	3.1	1.6	16.2	32.5	25.7	6.2	3.3	2.0	0.8

From Table 7.1 we see that the majority of mispredictions occur in groups 2-6. These groups span the 36 tagged tables in TAGE, partitioned into groups of 4, indexed from 0-8. Interestingly, within these buckets, a few tables are not very impactful. In fact there are 16 tables that account for the majority (75+%) of mispredictions.

Table 7.2: Percentage of Mispredictions for each Confidence Level

	0	1	2
Mispredictions	41.4	19.8	38.8

From Table 7.2 we see that 50% of mispredictions from 32KB TAGE-SC-L occur when the confidence level is Low or Medium

Combining these 2 categories could yield a good filter for deciding when TAGE will mispredict. As a misprediction can be converted into a correct prediction by applying an inversion to the prediction, a large logic element can be attributed to the filtering mechanism and the secondary prediction trivialized.

We consider using the existing neural architecture to determine a much more complex confidence in TAGE's prediction. We must modify the inputs to the network to increase the

likelyhood that a function learned by this model is viable. We call this network the Online Branch Filtering Network (OBFN). We also modify the training scheme based on the filter.

The Bank ID for the matching entry in TAGE is retrieved early using ahead prediction, and based on our observations across a majority of SimPoints, this ID is mapped to a value between 0 and 15 and used as an index into the table of neural networks. Tag matches in banks that are not commonly inaccurate are ignored. instead of using global branch history and local branch history, a Global Correctness History Register (GCHR) and a table of Local Correctness History Register (LCHR) are used. These are indexed by a function of the BankID and branch address (PC). The neural network output is then used as a factor in a function used to determine a secondary confidence in TAGE. If the tag matching entry in TAGE has a low or medium confidence and the network predicts a TAGE inaccuracy, the output of TAGE is inverted, otherwise the prediction is left unperturbed.

We see that the accuracy is again unperturbed from the baseline 32KB TAGE-SC-L predictor. It is apparent that this iteration of the OBFN is incapable of effectively determining the likelyhood that TAGE will mispredict.

Another method of filtering would be to look at the cases when TAGE mispredicts a branch that Neural is able to predict correctly. By isolating the cases where both predictors mispredict and only TAGE mispredicts, we can establish a filter that is able to isolate a majority of these events without conflicting in the set of correct TAGE predictions. Such a filter could utilize path histories, microarchitectural state, register file values, etc. This is a topic for future work.

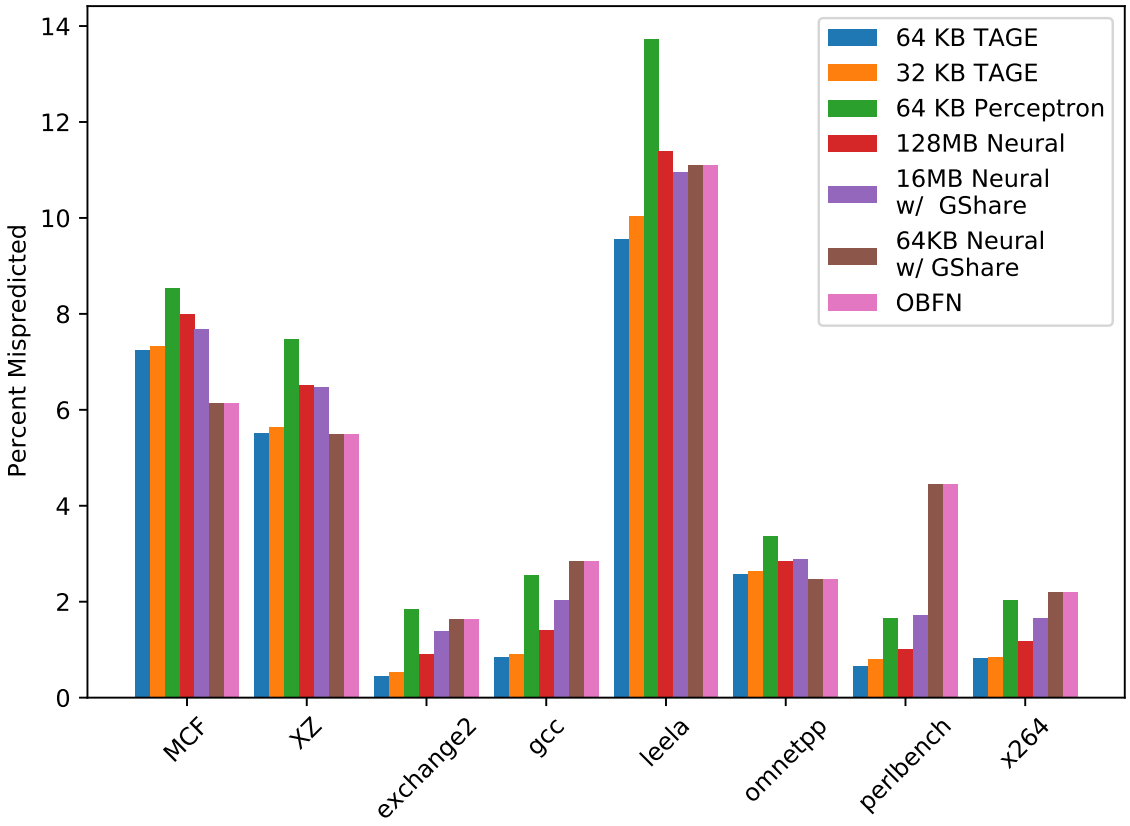

Figure 7.1: Performance of the predictor model at different configurations and the OBFN design

CHAPTER 8

FUTURE WORK

The neural prediction architecture, while not substantially improved compared to TAGE, has given keen insights into neural inspired learning methods. Further cost reduction methods such as variable fixed-width weights, alternative input selection, analog computation, and alternative activation functions may prove vital to improving the accuracy of the neural portion of the predictor. If the area and implementation cost can be reduced substantially, a larger input set can be considered. Further work may explore a Deep Online Neural (DON) predictor that uses a much deeper network with more interesting activation functions to learn complex relationships that the SON predictor cannot. in a similar vein the Convolutional Online Neural (CON) predictor could be designed in such a manner that specific hard to predict branches are isolated and analyzed at runtime to create a more nuanced prediction engine.

As mentioned in chapter 7, Many other methods of filtering and making a better choice predictor should be explored in a search for a better combined predictor.

CHAPTER 9
CONCLUSION

The neural prediction method as defined in this thesis is able to near the performance of TAGE with the ability to correctly predict branches TAGE is unable to. The current implementation is able to provide predictions 1-2 cycles prior to TAGE. The neural prediction architecture as discussed can be applied to other binary classification problems as is evident by the TAGE filtering mechanism (chapter 7) producing comparable results to the 64KB theoretical model.

As a more advanced neural architecture than the hashed perceptron predictor, the neural model outperforms its predecessor in the Hashed Perceptron predictor with substantially fewer networks, indicating that a neural approach is able to learn substantial relationships despite aliasing. This is indicative that a deeper network may be capable of learning relationships that TAGE and perceptron cannot and by utilizing fewer networks and an extensively precise filter that allows for more inputs and more complex layers, a more interesting outcome may be achieved.

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

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

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

[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] 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.

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

[8] 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.

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

[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] J. Lafiandra, "Brat: Branch prediction via online training", M.S. thesis, Georgia Institute of Technology, Atlanta, Georgia, Jan. 2021.

[13] T.-Y. Yeh and Y. N. Patt, “Two-level adaptive training branch prediction”, in *Proceedings of the 24th Annual International Symposium on Microarchitecture*, ser. MICRO 24, Albuquerque, New Mexico, Puerto Rico: Association for Computing Machinery, 1991, pp. 51–61, ISBN: 0897914600.

[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.

[16] B. Calder *et al.*, “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.

[18] Verma, “Spotlight - a low complexity highly accurate profile-based branch predictor”, in *2009 IEEE 28th International Performance Computing and Communications Conference*, 2009.

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

[20] R. Kessler, “The alpha 21264 microprocessor”, *IEEE Micro*, vol. 19, no. 2, pp. 24–36, 1999.

[21] T.-Y. Yeh and Y. N. Patt, “A comparison of dynamic branch predictors that use two levels of branch history”, *ACM SIGARCH Computer Architecture News*, vol. 21, no. 2, pp. 257–266, May 1993.

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

[23] 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.

[24] 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.

[25] 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.

[26] J. Bucek, K.-D. Lange, and J. v. Kistowski, "Spec cpu2017: Next-generation compute benchmark", in *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '18, Berlin, Germany: Association for Computing Machinery, 2018, pp. 41–42, ISBN: 9781450356299.

[27] 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.

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

[29] 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.