B.Comp. Dissertation

Branch prediction using deep learning

By

Liu Dongxu

Department of Computer Science

School of Computing

National University of Singapore

2023/04

B.Comp. Dissertation

Branch prediction using deep learning

By

Liu Dongxu

Department of Computer Science

School of Computing

National University of Singapore

2023/04

Project No: CP3106
Advisor: Prof. Weng-Fai Wong
Deliverables:
Report: 1 Volume

Abstract

This report explores the topic of branch prediction, a technique commonly used in modern computer processors to optimize instruction execution. The current state-of-the-art branch predictor, TAGE, struggles to identify correlated branches deep within a noisy global branch history, and fundamental breakthroughs in branch prediction have become increasingly rare. To address this issue and further improve branch prediction, this report suggests relaxing the constraint of runtime-only training and adopting more sophisticated prediction mechanisms that incorporate deep learning. Building on the work of BranchNet, a convolutional neural networks (CNNs) with a practical on-chip inference engine tailored to the needs of branch prediction, the report proposes a better CNN model that requires fewer resources. In an effort to reduce the expensive computational cost of multiplication operations, the report implements a novel deep learning model called Spiking Neural Networks (SNNs) for prediction, which only utilizes addition operations while maintaining higher accuracy than TAGE.

Keywords:
Branch prediction, Deep learning, CNN, SNN

Implementation Software and Hardware:
C++, Python, Pytorch, NVIDIA TITAN V

Acknowledgement

I would like to thank my friends, families, members of the laboratory and advisors. Without them, I would not have been able to complete this project. In addition, thanks to the NGNE program for giving me the opportunity to do research.

List of Figures

List of Tables

Table of Contents

Chapter 1

Introduction

1.1 Background

In this section, we briefly discuss the history and background of the problem. A detail literature survey is presented in Chapter 2.

The technique of branch prediction is a crucial aspect of modern computer processors, specifically in optimizing the execution of programs that contain conditional branching statements, like if-then-else and loops. Its primary objective is to predict whether a conditional branch will be taken or not, allowing the processor to speculatively execute the subsequent instructions before the condition is evaluated. This prediction accuracy is essential as it determines whether the processor can avoid delays that can slow down program execution.

The significance of branch prediction lies in its ability to enhance processor performance by reducing the number of pipeline stalls that occur due to conditional branches. The correct prediction of branch outcomes enables the processor to continue with speculative instruction execution, leading to faster program execution. However, it is essential to note that a wrong prediction can have severe consequences, causing a pipeline flush that can take several clock cycles to resume normal execution. This penalty is more significant in deeper pipelines where numerous instructions are in flight simultaneously.

Therefore, the accuracy of branch prediction is a vital factor in minimizing pipeline flushes and improving processor performance.

1.2 The Problem

Continuously striving to enhance processor performance, we are constantly working towards improving the accuracy of branch prediction.

The logical bottleneck in enhancing single-thread performance is branch prediction, which despite state-of-the-art TAGE-SC-L predictor(Seznec, 2016b), causes many SPEC2017 Integer benchmarks to experience high branch mispredictions per kilo instructions (MPKI), ultimately leading to a notable decline in performance. Furthermore, as processors shift towards deeper and wider pipelines(Sprangle & Carmean, 2002), the penalty for branch misprediction exacerbates.

Regrettably, there have been few groundbreaking advances in traditional branch prediction(Michaud, 2018). Consequently, we are attempting to introduce novel approaches in this field by leveraging deep learning.

1.3 Our Solution

Our primary objective is to explore new approaches to achieve highly accurate branch prediction while minimizing resource utilization.

Related to the BranchNet, which proposes a novel and viable solution using convolutional neural networks (CNNs) for branch prediction has proven successful in addressing a critical weakness of TAGE-like predictors by identifying correlated branches in a noisy global history.

To achieve this, sophisticated prediction mechanisms that require computationally-heavy training algorithms and additional compiler support must be adopted to learn these correlations.

Expanding on their findings, our efforts involve reproducing their work and subsequently streamlining and improving the performance of the CNN model. Furthermore, we aim to utilize a Spiking Neural Networks (SNNs) as an alternative deep learning model to reduce computational resource usage while maintaining high accuracy, surpassing the current state-of-the-art branch predictors.

Chapter 2

Related Work

2.1 Pipelining

Pipelining is a technique used in CPUs to improve performance by allowing multiple instructions to be executed at the same time. It takes advantage of the parallelism that exists among the different actions needed to execute an instruction. Essentially, pipelining is like an assembly line, where each step completes a part of a different instruction in parallel. Just like in an automobile assembly line, each step operates in parallel with the other steps, but on a different car.

In a computer pipeline, each step in the pipeline completes a part of an instruction. Different steps complete different parts of different instructions in parallel. Each of these steps is called a pipe stage or a pipe segment. The stages are connected one to the next to form a pipe. Instructions enter at one end, progress through the stages, and exit at the other end, just as cars would in an assembly line.

There are situations called hazards that can prevent the next instruction in the instruction stream from executing during its designated clock cycle. These hazards can reduce the performance from the ideal speedup gained by pipelining.

Structural hazards arise from resource conflicts when the hardware cannot support all possible combinations of instructions simultaneously in overlapped execution. This can cause a delay in the execution of instructions and can negatively affect the performance of the CPU.

When a branch is executed, it may or may not change the PC to something other than its current value plus 4. If a branch changes the PC to its target address, it is a taken branch. If it falls through, it is not taken or untaken. As pipelines get deeper and the potential penalty of branches increases, using delayed branches and similar schemes becomes insufficient. Instead, we need to turn to more aggressive means for predicting branches.

To address the issue of branch prediction, there are low-cost static and dynamic schemes that can be used. The simplest dynamic branch-prediction scheme is a branch-prediction buffer or branch history table. This is a small memory that is indexed by the lower portion of the address of the branch instruction. The memory contains a bit that says whether the branch was recently taken or not. This scheme is the simplest sort of buffer, and it has no tags. It is useful only to reduce the branch delay when it is longer than the time to compute the possible target PCs.

A branch-prediction buffer can be implemented as a small, special "cache" accessed with the instruction address during the IF pipe stage. It can also be implemented as a pair of bits attached to each block in the instruction cache and fetched with the instruction. If the instruction is decoded as a branch and if the branch is predicted as taken, fetching begins from the target as soon as the PC is known. Otherwise, sequential fetching and executing continue.

2.2 The state-of-the-art Predictor - TAGE

TAGE (Tagged Geometric History Length Predictor)(Seznec & Michaud, 2006) is a highly effective runtime branch predictor algorithm that leverages multiple components to predict the outcomes of conditional branches in computer programs.

Initially introduced by Andre Seznec in 2006, TAGE has demonstrated remarkable success in achieving high accuracy and low miss rates in contemporary processors. This hybrid predictor employs a combination of various predictors, including GEHL, PPM, and others, and relies on a tagged history table to store recent branch outcomes.

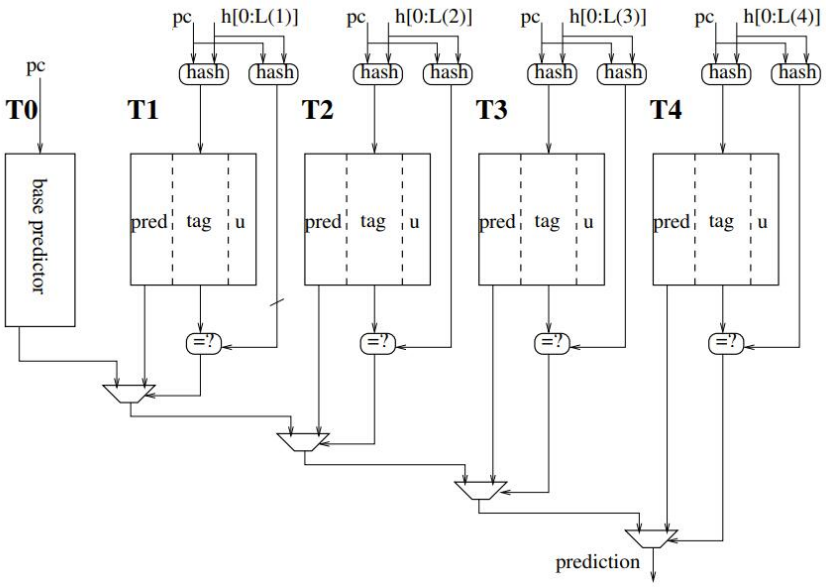

Figure 2.1: TAGE features 5 banks

TAGE utilizes geometric history lengths to select the most suitable predictor for each branch. The geometric history length is a variable-length history that adapts dynamically to the program's behavior. This adaptability enables TAGE to adjust to different types of branches and program behavior, making it highly suitable for complex modern workloads.

In various benchmarks and workloads, the TAGE predictor has outperformed other state-of-the-art predictors in terms of accuracy and performance. This highlights the effectiveness and versatility of TAGE in modern processors.

However, runtime branch predictors in current times have a significant drawback wherein they require exponentially increasing capacity in the presence of long, noisy histories. This weakness is deemed a fundamental consequence of runtime training.

Despite this, previous branch predictors that relied on offline training with profiling failed to address this weakness. Hence, it is clear that this is an inherent limitation of runtime training, and alternative approaches must be adopted to overcome this weakness.

2.3 Previous work in branch prediction using deep learning - BranchNet

Tarsa et al.(Tarsa, Lin, Keskin, Chinya, & Wang, 2019) proposed the use of CNN that are trained at compile-time to accurately predict branches that TAGE cannot. By providing enough profiling coverage, CNN can learn input-independent branch correlations, which enables them to accurately predict branches when a program is run with unseen inputs.

Building on their work, this paper introduces BranchNet(Zangeneh, Pruett, Lym, & Patt, 2020), which is a CNN with a practical on-chip inference engine tailored specifically for branch prediction needs. At runtime, BranchNet predicts a few hard-to-predict branches, while TAGE-SC-L predicts the remaining branches. This hybrid approach leads to a reduction of the MPKI of SPEC2017 Integer benchmarks when compared to a very large MTAGE-SC baseline. This demonstrates a fundamental advantage in the prediction capabilities of BranchNet when compared to TAGE-like predictors.

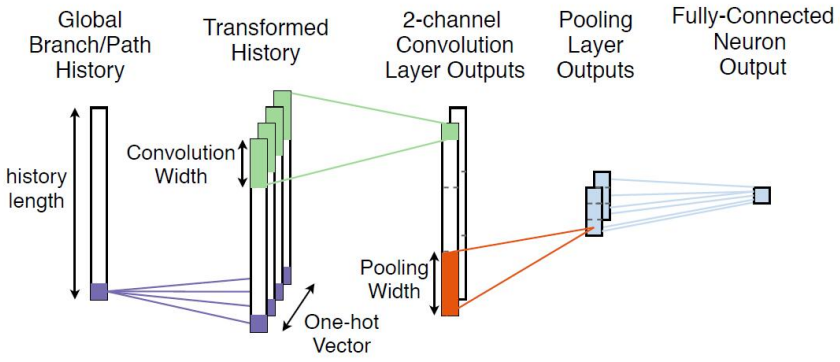

Figure 2.2: BranchNet dataflow in CNN

In addition, they propose a practical resource-constrained variant of BranchNet that improves the MPKI when compared to a 64KB TAGE-SC-L without increasing the prediction latency. This highlights the potential of BranchNet in achieving high accuracy in branch prediction while minimizing computational resources.

Chapter 3

Problem and Solution

Our workflow involves several steps. Firstly, we aim to reproduce the previous work. Once we have replicated the previous research, we will apply our own CNN model to improve upon the existing methods. Finally, we will utilize the Spiking Neural Networks (SNNs) as another deep learning model to make predictions.

3.1 Get Base Data by pin

To get the necessary features from the programs, we use the Pin.

The Pin tool provides a convenient way to instrument programs and extract features related to branches.

Pin allows users to dynamically insert custom code in specific locations within an executable, which can be particularly useful for capturing branch instructions during runtime. Pin's API further abstracts the underlying instruction set details and allows for context information to be passed as parameters to the inserted code, which can provide additional insights into the program behavior.

By using Pin, we can get the instructions we want (the branch instructions) and extract relevant information such as the branch instructions' pc, target address, direction (taken or not), branch types, opcode, etc., and store this information in a binary file for later use in branch prediction.

3.2 Get TAGE's result

Using the base dataset file, we can directly evaluate the performance of the baseline predictor TAGE. We compare TAGE's prediction accuracy with that of MTAGE-SC(Seznec, 2016a), which is the best predictor in the unlimited storage category of CBP 2016.

Based on the comparison results, we identify the hard-to-predict branches with the highest misprediction rate per kilo instructions (MPKI). These branches are selected as the target for our deep learning approach.

3.3 Get dataset for deep learning

To effectively compress the necessary data into an hdf5 file, we need to follow a sequence of steps.

Firstly, we must generate "branch scenarios" {branch PC, direction} for the entire branch history.

Secondly, we need to identify the specific branch PC required and obtain the corresponding index list in the branch scenario history for each PC.

Then, we can proceed to compress this data into the desired hdf5 file format.

We choose to split the scenario history into two parts from the middle, using the first part for training and the second part for testing in order to utilize this dataset.

Here is a example of the content in hdf5: 3.1

3.4 Previous work using CNN

Firstly, we replicated the previous work using a CNN model.

To effectively identify correlated branches amidst a noisy global history, we can utilize the branch scenario history preceding a specific branch as inputs. Retrieving this chunk data from an hdf5 dataset is a straightforward process.

However, merely utilizing the branch scenario as a training input is insufficient, as it is

Table 3.1: an hdf5 structure example for deep learning

```
/ (group)
├── history (dataset)
│   ├── branch scenario
│   ├── branch scenario
│   └── ...
├── br_indices_PC1 (dataset)
│   ├── index
│   ├── index
│   └── ...
└── br_indices_PC2 (dataset)
    └── ...
```

represented by integers and may fail to account for the relationship between different scenarios.

To address this issue, inspired by Hashemi et al.(Hashemi, Swersky, Smith, Ayers, Litz, Chang, Kozyrakis, & Ranganathan, 2018) we can transform the inputs into embedding, mapping each branch scenario integer to a list of floating-point numbers. This approach enables us to convert the relationship between different scenarios into relationships between different dimensions, thereby providing a more comprehensive and nuanced understanding of the underlying data.

With the CNN architecture now established, we are able to commence training the model. 3.1 3.2

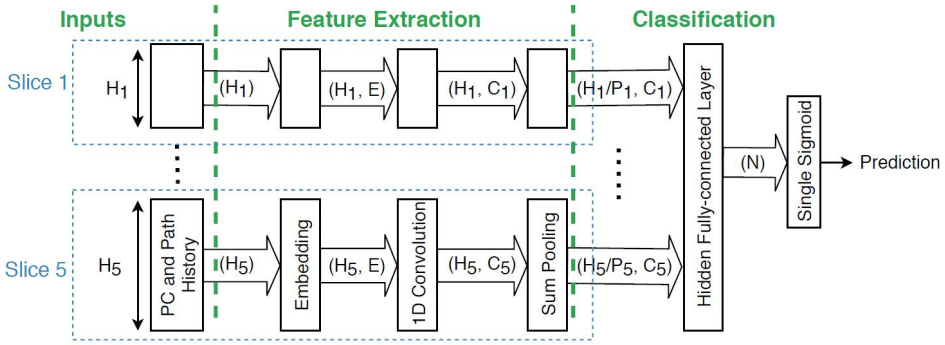

Figure 3.1: High-level diagram of Big-BranchNet CNN architecture

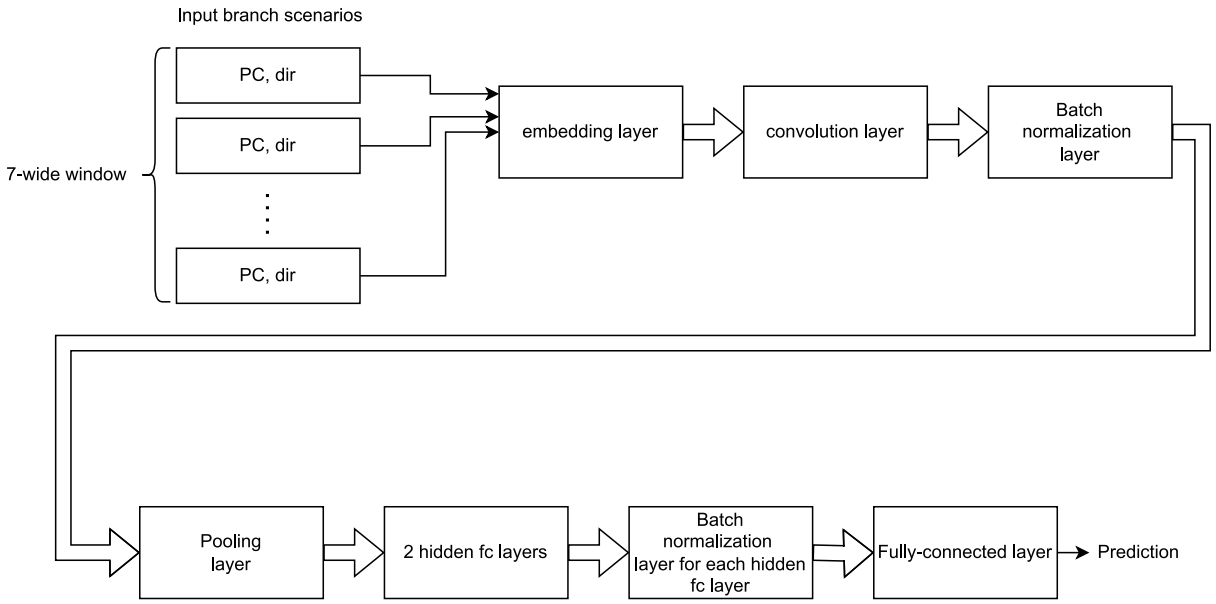

Figure 3.2: BranchNet CNN architecture

When we encounter a branch PC, we have access to a sequence of historical branch scenarios leading up to it. These scenarios are used as inputs to the model. Firstly, the input sequence is transformed into an embedding in the embedding layer. The convolution operation is then applied to each 7-wide window of branch scenarios. The output is normalized using the batch normalization layer, after which we apply pooling to reduce the output size and pass it to the next fully connected layer. The model has two hidden fully connected layers and the last fully connected layer produces a single value for prediction.

To process a larger sequence of branch scenarios, we repeat the aforementioned operations five times across the sequence, each time processing a slice of the input. The output of each slice is concatenated and fed into the final fully connected layer for prediction.

3.5 My CNN Model

Our goal is to develop a CNN model that balances accuracy and computational efficiency. To achieve this, we will explore ways to increase accuracy while reducing the computational resources required.

We will remove some layer content to reduce the model's computation, while also testing approaches to improve accuracy without adding too much computational overhead. Our ideal outcome is to achieve higher accuracy while reducing computational resources, demonstrating more convincingly that our model is superior to the original. If we increase computational resources to achieve higher accuracy, it would be less compelling evidence to prove the superiority of our model, as the original model may also improve accuracy by increasing computational resources, which is difficult to demonstrate. Moreover, we hope to apply this model more effectively in practical applications by reducing computational resources.

Compared to the BranchNet's best-performing model, Big-BranchNet (Zangeneh et al., 2020), which utilizes 5 slices with a convolution layer of varying history length, we found that only the longest history length slice significantly contributed to the prediction. The result presented here highlights that utilizing a single slice with the longest history yields accuracy that closely rivals that of employing five slices.

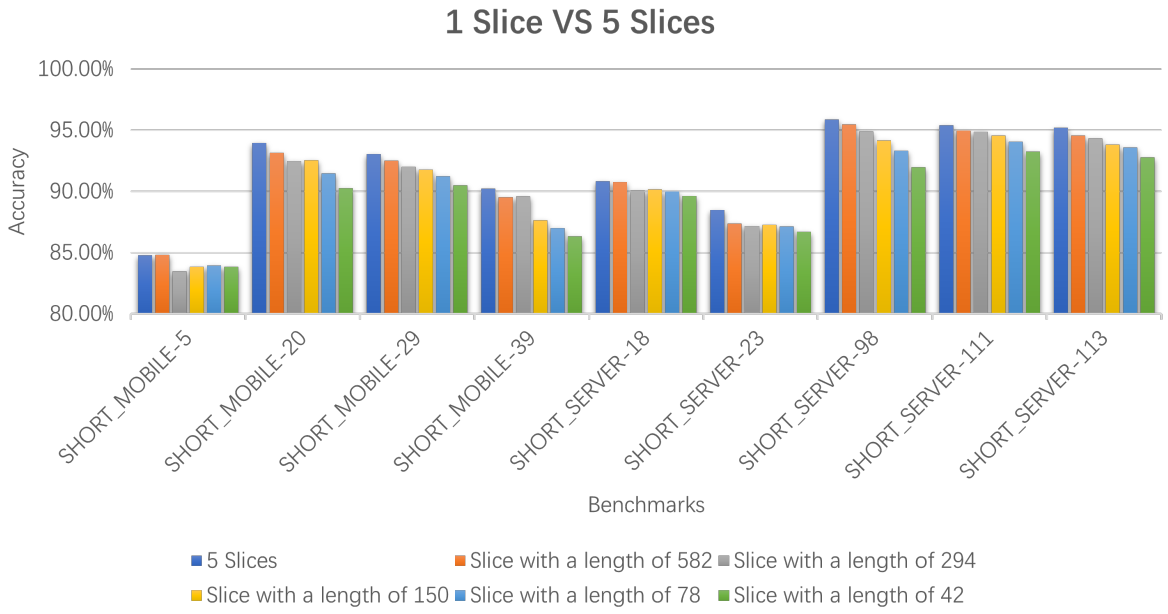

Figure 3.3: 1 Slice vs 5 Slices

Our findings indicate that augmenting the history length leads to a corresponding increase in accuracy. And of course, this would require a greater computational resources. Our solution

is that employs a single convolution layer with an extended history length.

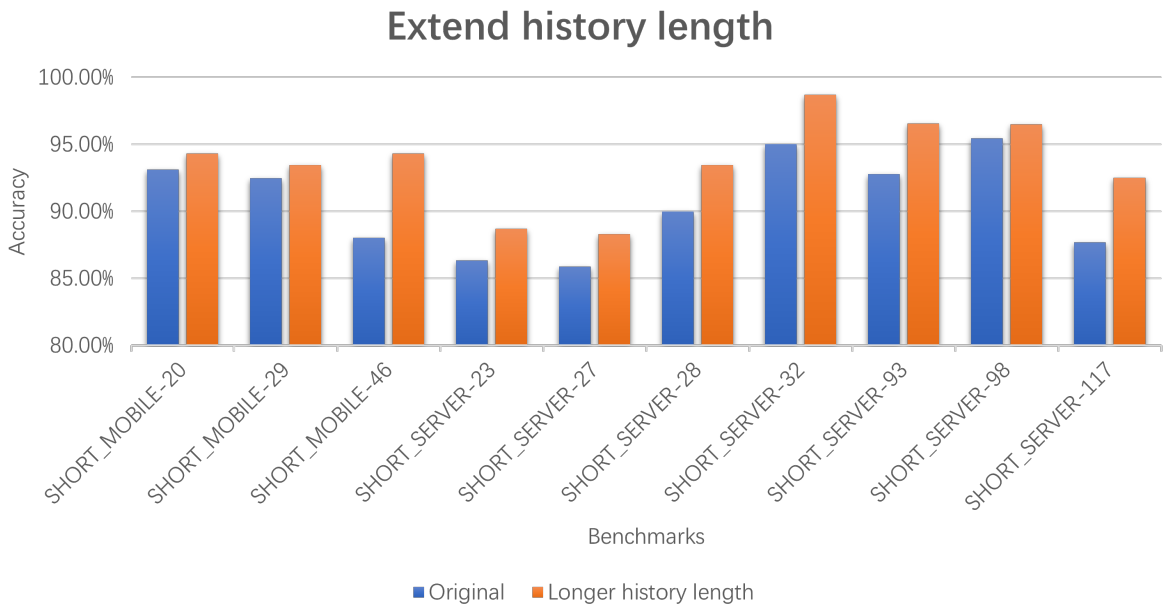

Figure 3.4: extended history length vs origin

We are continually reducing the content by removing hidden fully connected layers. Through various attempts, we have discovered that removing both hidden fc layers resulted in decreased prediction accuracy due to a lack of non-linearity. It is important to note that the BranchNet model contains two hidden fc layers, each with 128 hidden neurons. To strike a balance, we attempted to remove one hidden fc layer and reduce the number of neurons for the other hidden fc layer, which had minimal impact on prediction accuracy. The following figure illustrates the varying degrees of accuracy achieved using different numbers of neurons.

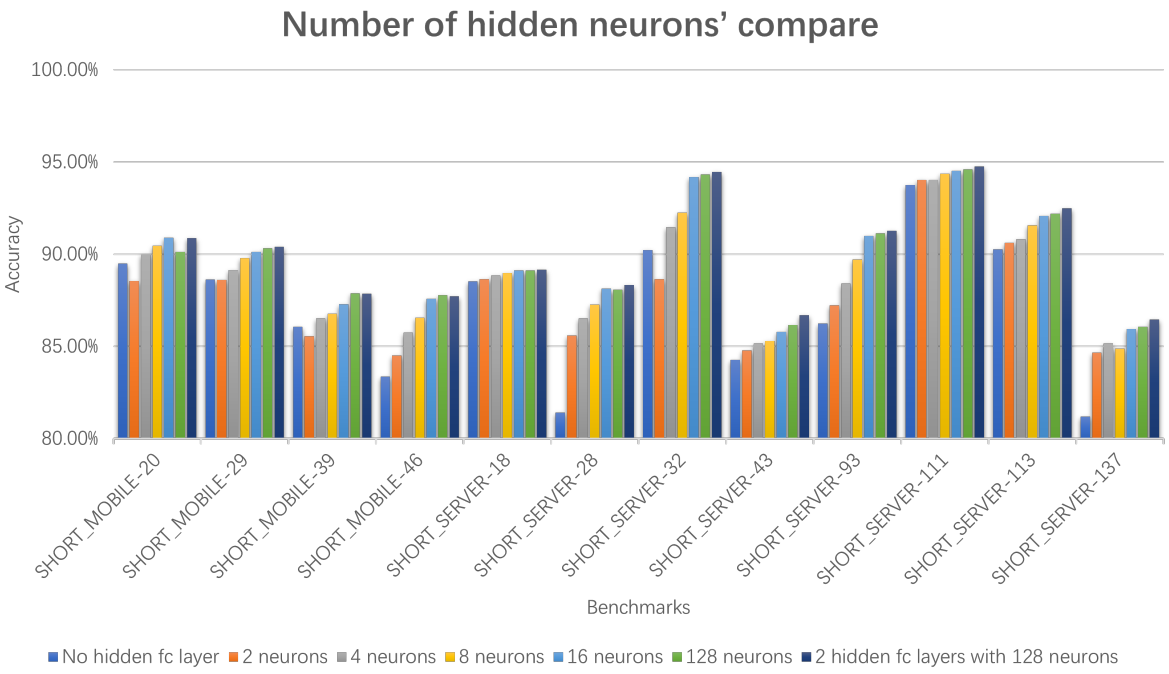

Figure 3.5: Hidden neurons' compare

Based on the results, it is evident that the performance of No hidden fc layer and those with only two neurons is not satisfactory. It appears that accuracy improves with an increase in the number of neurons, specifically with 4, 8 and 16 neurons, but the differences are minimal. To explore the upper limit of the results, we added one hidden fc layer with 128 neurons and the full model (two hidden fc layers, each containing 128 hidden neurons). The findings reveal that accuracy with 8 and 16 neurons is close to the upper limit.

It is important to note that the hidden fc layer is relatively small, and the preceding layer is a pooling layer designed to reduce the output size for the next layer. Our observations indicate that removing the pooling layer enhances accuracy without the additional computational resources required for pooling.

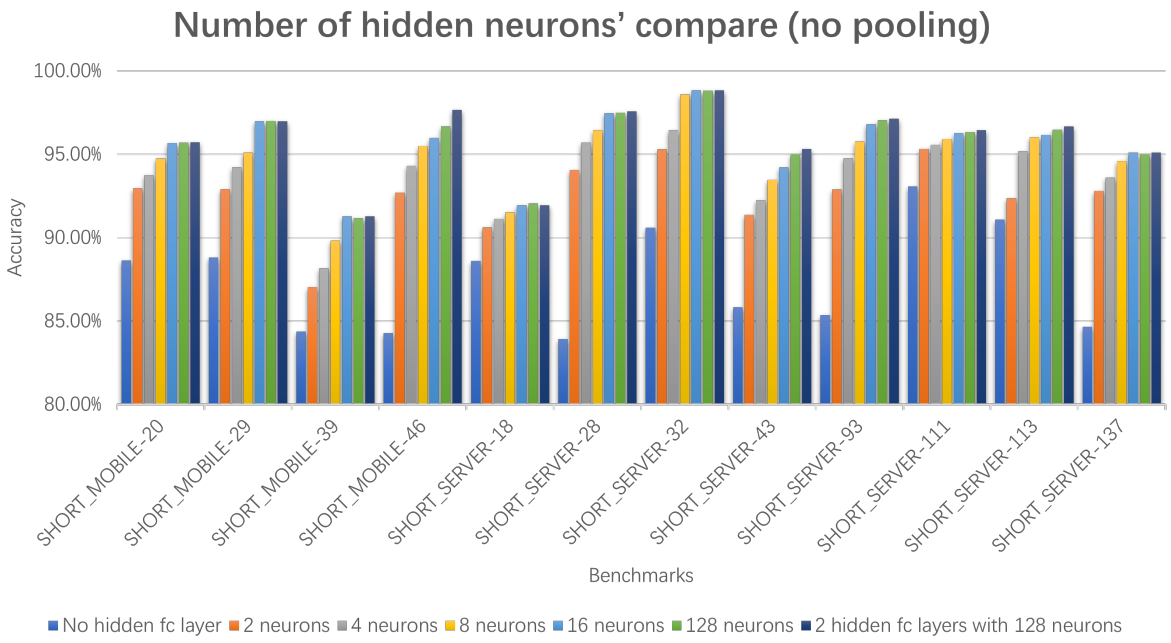

Figure 3.6: Hidden neurons' compare (no pooling)

The results clearly indicate that removing both hidden fc layers yields significantly worse performance compared to other models. It is clear that accuracy with 8 and 16 neurons is close to the upper limit. It is a trade off that we opt to use one hidden fc layer with 8 hidden neurons.

By removing the pooling layer, we have observed an increase in accuracy. Previously, we utilized the pooling layer to decrease the computational demands of the subsequent layer. However, since the following fc layer is now quite small, there is no longer a need for the pooling layer.

After multiple iterations, we propose a new CNN model.

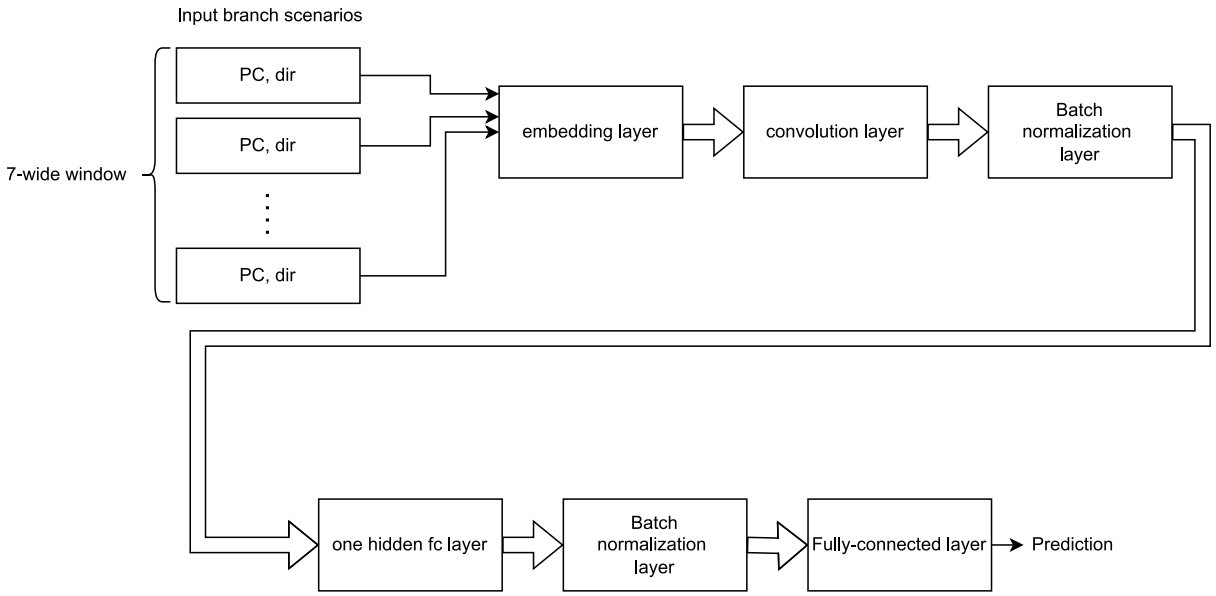

Figure 3.7: My CNN architecture

3.6 Branch Prediction using SNNs (Spiking Neural Networks) Model

One of the main reasons we decide to transform CNN to SNN is to address the issue of high computational resources consumption. As we know, CNN consume a lot of resources, especially when using multipliers. This is not ideal for branch prediction models that require energy efficiency. Therefore, in this section, we propose an alternative approach using SNN.

SNN represent and process information using the timing and frequency of spikes. This approach simulates floating-point numbers with a sequence of time steps and corresponding spike frequencies. One significant advantage of SNN is that they reduce energy consumption by replacing weight multiplications with additions, making them suitable for energy-constrained platforms.

So we can avoid multiplication operations and only use addition operations while CNN using a lot of operations in the convolution.

However, training SNN remains a challenge due to their discrete activation. Therefore, we

propose to transform CNN to SNN as a solution. In (Yan, Zhou, & Wong, 2021), they propose CQ training (Clamped and Quantized training), an SNN-compatible CNN training algorithm with clamp and quantization that achieves near-zero conversion accuracy loss to transform.

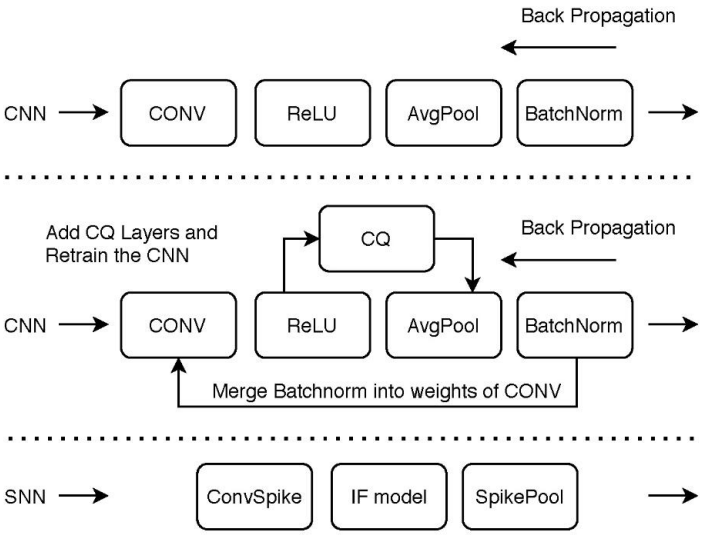

Figure 3.8: SNN Training Workflow

With this technique, we are able to utilize SNN for branch prediction.

Chapter 4

Evaluation

4.1 Implementation Details

We list our experiment environment:

OS: Ubuntu 22.04.2 LTS

Kernel: Linux 5.15.0-67-generic x86_64

CPU: Intel(R) Xeon(R) Gold 5118 CPU @ 2.30GHz

GPU: NVIDIA TITAN V

RAM: 12288MiB

our dataset is from: CBP 2016 dataset

Pin tool: pin-3.23

Tage code: Championship Branch Prediction Program

Transfer learning for SNN: ShenjingCat: An SNN framework with PyTorch

4.2 Experimental Setup

This Convolutional Neural Network (CNN) has been configured with a batch size of 2048, which means that 2048 samples will be processed at a time during training. The optimizer used is Adam, a popular optimization algorithm for deep learning models. The branch history length

is set to 1000, meaning that for the current branch, the model will consider the previous 1000 branches of input data when making predictions.

The embedding dimension, or the size of the vectors used to represent input data, has been set to 32. The CNN has 32 convolutional filters, each of which has a width of 7. These filters will be applied to the input data to extract important features. Additionally, the model has 16 hidden neurons, which are used to process the output from the convolutional layers.

The CNN has been trained for 100 epochs, meaning that the entire dataset has been passed through the model 100 times during training.

The timestep of spiking neural network (SNN) model is 8, which means that SNN is operating on discrete time steps of 8. This parameter determines the resolution at which the model can process temporal information, as it specifies how frequently the neuron's membrane potential is updated and how frequently synaptic connections between neurons are updated.

4.3 CNN Results

The results of our analysis reveal that the accuracy of our approach is superior to the BranchNet.

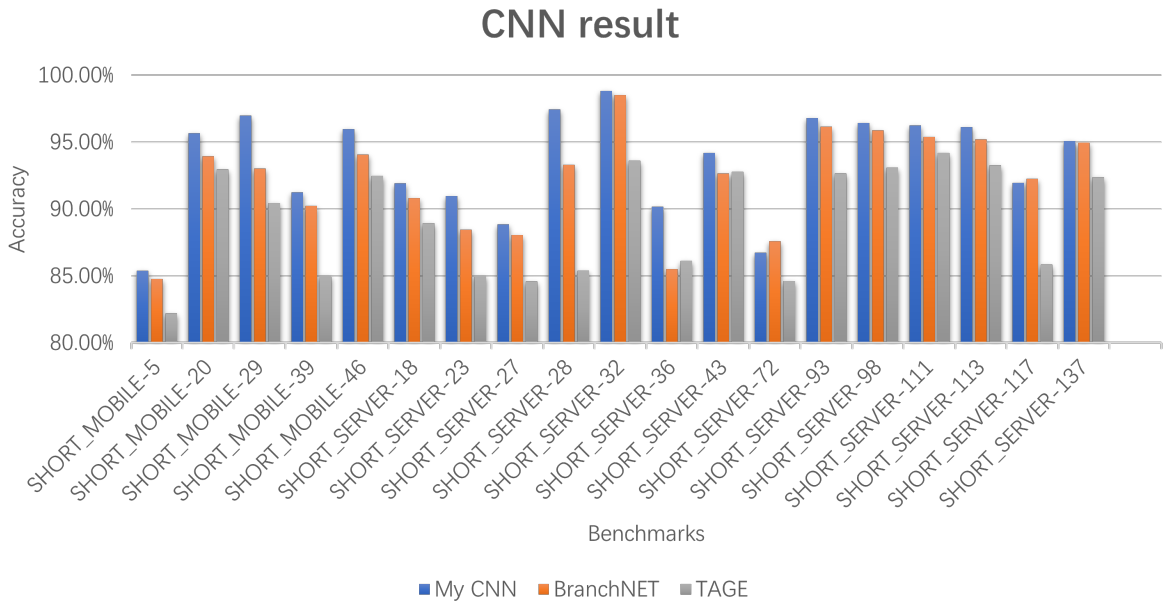

Figure 4.1: CNN accuracy result

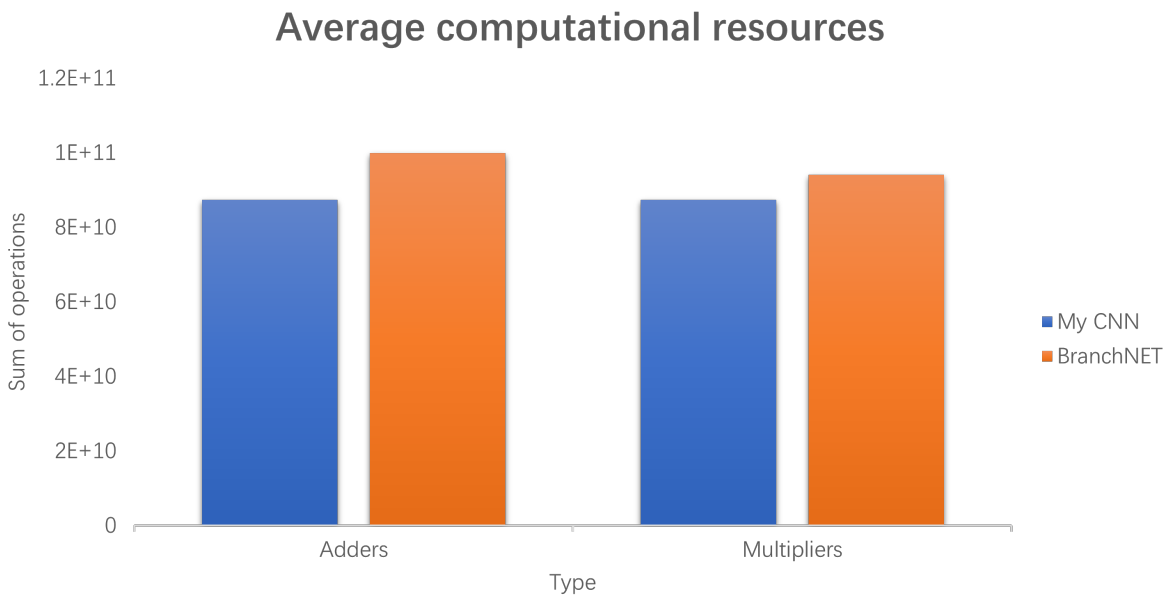

Figure 4.2: CNN operation result

Importantly, this higher accuracy is achieved while utilizing fewer resources, representing a clear improvement over previous methods.

Notably, even when focusing on the branches that were selected due to high mispredictions in TAGE, we still achieved remarkable accuracy.

Further examination of the data revealed that branches that were accurately predicted by TAGE were also effectively predicted by our approach using CNN.

However, for those branches that exhibited low accuracy in TAGE, we observed a significant improvement in accuracy when utilizing our CNN approach. This finding underscores the value of our approach in improving predictions for challenging and complex datasets, even in cases where prior methods have struggled to achieve accurate results.

4.4 SNN Results

Our study provides compelling evidence that the SNN model offers a promising approach to reducing computational costs while maintaining accuracy. Taken together, these findings suggest

that the SNN model represents a valuable and exciting innovation in branch prediction using deep learning.

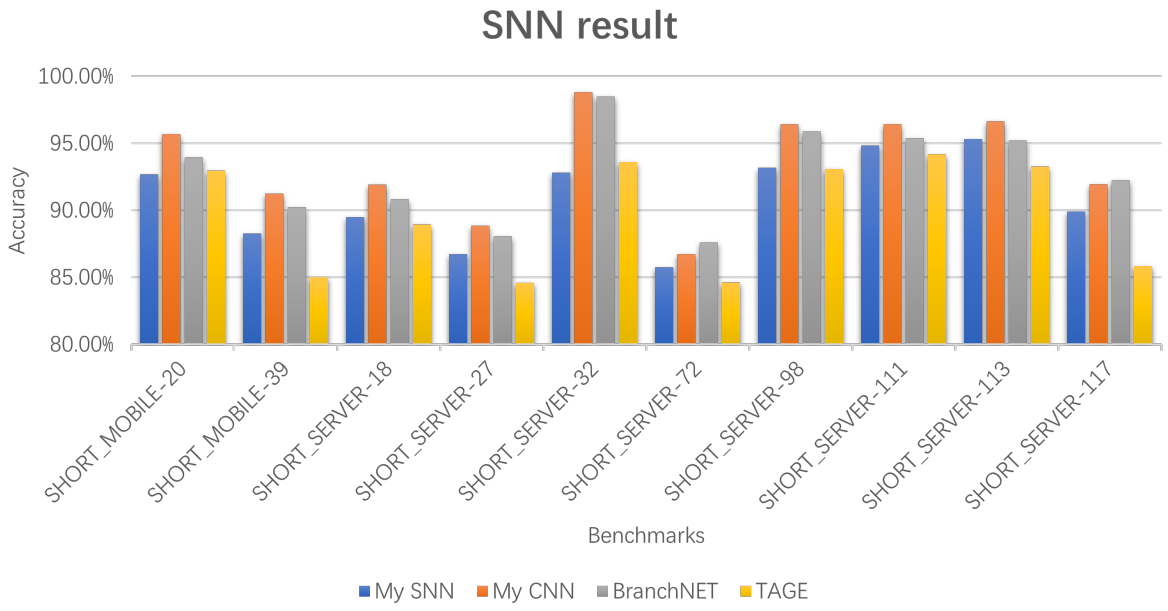

Figure 4.3: SNN accuracy result

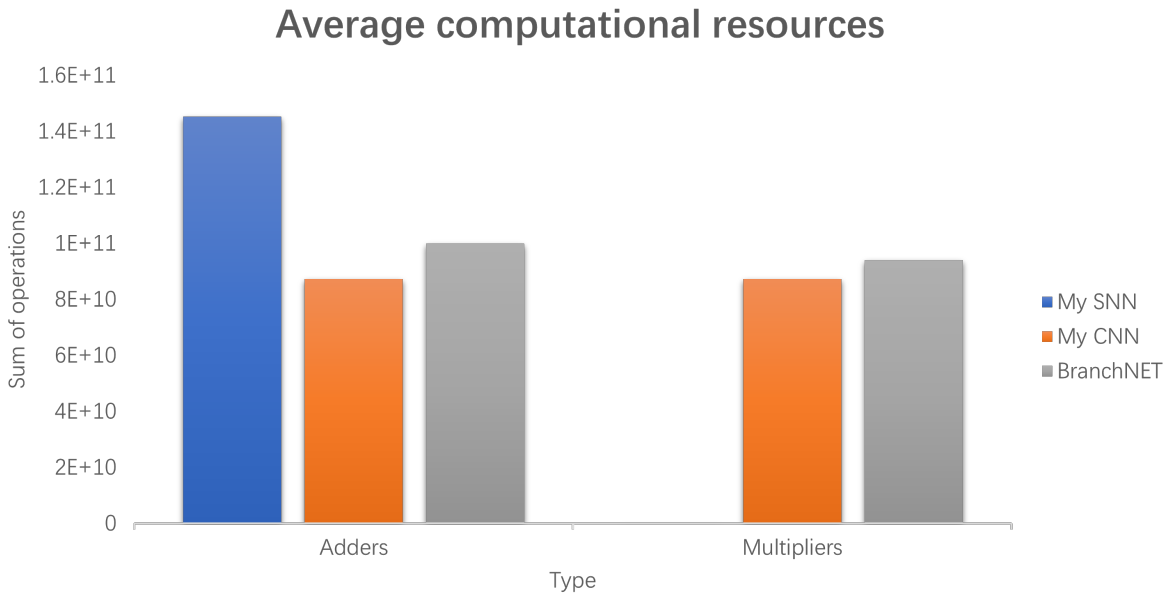

Figure 4.4: SNN operation result

In comparing the use of adders and multipliers in the evaluation process, we observed that the SNN model only utilizes adders and does not require multipliers at all. This is particularly significant given that multipliers are often among the most resource-intensive components of computational systems.

Thus, our study provides compelling evidence that the SNN model offers a promising approach to reducing computational costs while maintaining accuracy. Taken together, these findings suggest that the SNN model represents a valuable and exciting innovation in branch prediction using deep learning.

Chapter 5

Conclusion and Future Work

5.1 Conclusion

In conclusion, we have highlighted the challenges of branch prediction and the limitations of the current state-of-the-art branch predictor, TAGE. To address these challenges, we propose a new approach to branch prediction by adopting deep learning techniques, specifically using a CNN model tailored to the needs of branch prediction. The proposed CNN model builds on the work of BranchNet but requires fewer resources, making it a more practical solution. Additionally, we suggest the use of a novel deep learning model SNN to further reduce the computational cost. The results show impressive results. Overall, we present a promising approach to branch prediction that could lead to significant improvements in instruction execution optimization.

5.2 Future Work

5.2.1 SNN model improvements

The results clearly indicate a significant gap between the performance of the CNN and SNN models. It is evident that this disparity is due to the shortcomings in my transformation from CNN to SNN, which needs to be improved. Furthermore, adjusting the parameters to suit the SNN model better could be another way to bridge this gap. Despite these limitations, we believe that SNN still has substantial room for improvement and holds great potential for application

in fields where computational resources are limited.

5.2.2 New inputs

Although our deep learning model shows promising results, there are still some limitations that need to be addressed.

One of the limitations that currently impedes the implementation of our model in real branch prediction is our reliance on PC as part of the input. Although this approach is feasible for runtime branch predictors like TAGE, it presents a challenge for our offline deep learning model. As each program execution generates a different PC, the use of PC can only prove theoretical feasibility.

To overcome this challenge, we plan to explore alternative inputs for our model. One approach is to replace PC with part of the instructions. Another possibility is to use a vector of registers that have been used in the past instructions as input for training. To obtain these features, we can also use pin tool. These alternatives show great potential and are worthy of further exploration in future research.

References

Hashemi, M., Swersky, K., Smith, J., Ayers, G., Litz, H., Chang, J., Kozyrakis, C., & Ranganathan, P. (2018). Learning memory access patterns. In J. Dy, & A. Krause (Eds.), *Proceedings of the 35th International Conference on Machine Learning*, Vol. 80 of *Proceedings of Machine Learning Research* (pp. 1919–1928), 10–15 Jul, 2018: PMLR.

Michaud, P. (2018). An alternative tage-like conditional branch predictor. *ACM Trans. Archit. Code Optim.*, *15*(3), aug, 2018.

Seznec, A. (2016a). Exploring branch predictability limits with the mtage+sc predictor *, 2016.

Seznec, A. (2016b). Tage-sc-l branch predictors again, 2016.

Seznec, A., & Michaud, P. (2006). A case for (partially) tagged geometric history length branch prediction. *Journal of Instruction-level Parallelism - JILP*, *8*, 02, 2006.

Sprangle, E., & Carmean, D. (2002). Increasing processor performance by implementing deeper pipelines. *Proceedings of the 29th Annual International Symposium on Computer Architecture*, ISCA '02 (p. 25–34), USA, 2002: IEEE Computer Society.

Tarsa, S. J., Lin, C.-K., Keskin, G., Chinya, G., & Wang, H. (2019). Improving branch prediction by modeling global history with convolutional neural networks.

Yan, Z., Zhou, J., & Wong, W.-F. (2021). Near lossless transfer learning for spiking neural networks. *Proceedings of the AAAI Conference on Artificial Intelligence*, *35*(12), May, 2021, 10577–10584.

Zangeneh, S., Pruett, S., Lym, S., & Patt, Y. N. (2020). Branchnet: A convolutional neural network to predict hard-to-predict branches. *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)* (pp. 118–130), 2020.

Appendix A

Code

Our code is available on Branch-prediction-using-deep-learning.