Using Convolutionan Neural Networks to Improve Branch Prediction

Siavash Zangeneh Kamali

High Performance Systems Group
Department of Electrical and Computer Engineering
The University of Texas at Austin
Austin, Texas 78712-0240

TR-HPS-2022-002
August, 2022

The Dissertation Committee for Siavash Zangeneh Kamali
certifies that this is the approved version of the following dissertation:

Using Convolutional Neural Networks to Improve Branch Prediction

Committee:

Yale Patt, Supervisor

Mattan Erez

Andreas Gerstlauer

Calvin Lin

Tse-Yu Yeh

Using Convolutional Neural Networks to Improve Branch Prediction

by

Siavash Zangeneh Kamali

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

August 2022

Dedicated to my parents, Parviz Zangeneh Kamali and Hayedeh Gharehbaghi.

Acknowledgments

I owe the completion of this dissertation to many people who have helped and supported me.

My advisor, Prof. Yale Patt has been instrumental in the completion of my Ph.D. program. Thank you for inspiring my interest in computer architecture, for convincing me to pursue my Ph.D., for demonstrating effective teaching, and for teaching me the importance of sound fundamentals in research and scholarship. Your insistence on clear simple, concise, and clear presentation has had a significant positive impact on my communication skills. Also, thank you for providing me with opportunities to enhance my professional career.

I thank my Ph.D. committee members, Prof. Mattan Erez, Prof. Calvin Lin, Prof. Andreas Gerstlauer, and Tse-Yu Yeh. Your feedback after my proposal exam and the final defense helped shape this dissertation, fix several mistakes, and improve its writing quality.

I thank previous and current members of the HPS research group. In particular:

- Eric Sprangle, for teaching me the fundamentals of effective research and clear presentation, and for your generosity. Also thank you for getting me interested in the topic of machine learning, which had a direct impact on my decision to work on the project that led to this dissertation.

- Milad Hashemi, for providing helpful feedback on my research, especially for writing the BranchNet paper, for helping me find an internship position at Google, and for always being generous with your time.

- Faruk Guvenilir, for taking care of anything IT-related in the group, teaching me everything IT-related, and for your high energy when discussing research and when watching football!

- Ben Lin, for your humor that allowed me to cope with all the difficulties associated with graduate school, for always providing precise feedback on my research, for listening to me talk about obscure C++ techniques, for being interested in machine learning trivia, for all our pair-programming days when maintaining Scarab, for our many cultural and historical discussions, and for getting through the eventful Spring 2021 semester with me!

- Stephen Pruett, for teaching me about branch prediction, for teaching me the importance of writing good scripts, for co-authoring all my papers, for your persistence in making sure that all disagreements about our research directions are resolved, for your appreciation of good software engineering practices, for all the time we spent TAing together, for all our non-research related discussions about movies, TV shows, politics, and sports, and for the numerous times that you gave me a ride home late at night!

- Aniket Deshmukh, for not letting me remain the youngest HPS student, for your impactful contributions to creating a new functional model for Scarab, and for being willing to voice your honest opinion on research topics.

- Chester Cai, for re-introducing young positive energy in the group, for always being willing to help, and most importantly, for our mutual love of fried chicken!

I worked with many people during my various internships who helped me learn about computer architecture and machine learning from the industry perspective. I thank Georgi Gaydadjiev, Sundar Dev, Alex Ramirez, Danny Tarlow, Hassan Abolhassani, Kulin Kothari, Ethan Schuchman, and Niket Choudhary.

My friends have been a constant source of support throughout my life. I Thank:

- Yongkee Kwon, for always helping me and giving me good advice. You were like an older brother to me and I learned a lot from you.

- Esha Choukse, for making graduate school fun and being a great friend inside and outside of school.

- Ali Fakhrzadehgan, Majid Jalili, Sepideh Maleki, Kamyar Mirzazad, and Masoomeh Jasemi, for your friendship throughout many years. I value all the time we spent traveling, trying new restaurants, going to events, or simply talking about nonsensical topics. I hope our friendship continues to grow.

- Behzad Eftekhari, for accepting me as your roommate, teaching me how to live independently in the US, and teaching me about graduate school, which inspired me to pursue my Ph.D.

- Hossein Ganjizadeh, for being a very loyal friend who helped me get accustomed to the American culture, and for your unique sense of humor.

- Mohammad Amini, for being the best roommate I have ever had, and for your friendship.

- Tian Tan, for teaching me about FPGAs, and Chinese food, and for being a fun roommate.

- Sangkug Lym for all our discussions about machine learning and co-authoring papers.

- Alex Hsu, for bailing me out before a paper submission deadline, where I was struggling with counting gate delays.

- Arash Bassak, Hossein Safaei, and Ali Parsi, for your friendship and our many shared fun memories in Malaysia.

- Amin Sheibani, and Salar Nosrati, for being good friends since high school and keeping in touch with me after all these years.

I thank my girlfriend, Keiarra Ortiz-Cedeno, for your love and support through this stressful last year of graduate school, being there for me when I needed it, and motivating me to finish my work.

I would not be a student in a foreign country without the support of my family. Thank you Maman Hayedeh and Baba Parviz for your constant support. You taught me the value of hard work and perseverance from early childhood. There are no words that can sufficiently describe my gratitude for your sacrifices. You helped my siblings and me move abroad for better opportunities even though this meant we could not visit you regularly. I also thank my brothers. Thank you to my older brother, Khosro, for being a great older brother, a patient roommate, always making everyone in the family laugh with your sense of humor, teaching me how to live independently, and guiding me through early adulthood. Also, thank you to my younger brother, Fereidoon, for being a great companion even when we lived in different countries, talking to me about random pointless topics, and getting me interested in rock climbing. Thank you to my grandparents for your love and support. I hope hearing the news that I have graduated makes you happy. Rest in peace, Bababozorg Bahram.

Using Convolutional Neural Networks to Improve Branch Prediction

by

Siavash Zangeneh Kamali, Ph.D.
The University of Texas at Austin, 2022

SUPERVISOR: Yale Patt

The state-of-the-art branch predictor, TAGE, remains inefficient at identifying correlated branches deep in a noisy global branch history. This dissertation argues this inefficiency is a fundamental limitation of runtime branch prediction and not a coincidental artifact due to the design of TAGE. To further improve branch prediction, we need to relax the constraint of runtime only training and adopt more sophisticated prediction mechanisms. To this end, I propose using convolutional neural networks (CNNs) that are trained at compile-time to accurately predict branches that TAGE cannot. Given enough profiling coverage, CNNs learn input-independent branch correlations that can accurately predict branches when running a program with unseen inputs. I describe two practical approaches for using CNNs. First, I build on the work of Tarsa et al. and introduce BranchNet, a CNN with a storage-efficient on-chip inference engine tailored to the needs of branch prediction. At runtime, BranchNet predicts a few hard-to-predict branches, while TAGE-SC-L predicts the remaining

branches. This hybrid approach reduces the MPKI of SPEC2017 Integer benchmarks by 9.6% (and up to 17.7%) compared to a 64KB TAGE-SC-L without increasing the prediction latency. Alternatively, instead of using BranchNet as a black-box predictor, I use it to explicitly identify correlated branches and filter the global branch history of TAGE to include only the outcomes of correlated branches. Filtering the branch history leads to less allocation pressure and faster warmup time in TAGE, resulting in improved prediction accuracy and better storage-efficiency. Filtering TAGE histories achieves a notable fraction of BranchNet's accuracy improvements (average 3.7% MPKI reduction, up to 9.4%) with a simpler predictor design.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 The Problem

Branch prediction remains a major bottleneck in improving single-thread performance. Even with TAGE-SC-L [65], the state-of-the-art branch predictor, many SPEC2017 Integer benchmarks suffer from high branch mispredictions per kilo instructions (MPKI), resulting in significant loss of performance. Moreover, the branch misprediction penalty worsens as processors move towards deeper and wider pipelines [45, 49, 37, 74]. Unfortunately, fundamental breakthroughs in branch prediction have become rare [48]. All predictors submitted to the 2016 Championship Branch Prediction competition were variants of existing TAGE and Perceptron designs [65, 64, 30, 31, 56]. Branch prediction research needs new insights to further improve the prediction accuracy.

Traditional branch predictors like TAGE [66] and Perceptron [34] are designed to be updated at runtime. Thus, their update algorithms have to be simple, cheap, and quick to adapt to execution phase behavior. While simplicity and adaptivity are necessary for predicting most branches at runtime, limitations in training time and processing power make it difficult for runtime branch predictors to learn complex correlations in the branch history. To learn these correlations, it is necessary to adopt more sophisticated prediction mechanisms that require more computationally-heavy training algorithms.

An important subset of hard-to-predict branches for TAGE are those with many uncorrelated branches in their global histories. These uncorrelated branches produce redundant history bits (*noise*) that do not contribute to prediction accuracy. Since state-of-the-art predictors do not have any mechanisms to distinguish correlated branches from noise, they need to learn a prediction for each observable history pattern. This behavior is problematic because the number of history patterns grows exponentially with the number of redundant noisy bits. Thus, with enough uncorrelated branches in the history, the total number of observable history patterns will exceed the memorization capacity of runtime predictors, which makes these branches fundamentally hard to predict for state-of-the-art predictors.

1.2 Convolution Neural Networks for Branch Prediction

Convolutional Neural Networks (CNNs) address a key weakness of runtime predictors: identifying correlated branches in noisy global histories. A CNN learns to ignore uncorrelated branches and identify correlated branch patterns anywhere in the history, enabling expressive prediction functions that remain efficient and accurate even with long noisy histories. This increase in prediction capability, however, comes at the cost of computationally-expensive training and reliance on large training data. Therefore, it is not possible to train CNNs at runtime. Instead, CNN models should be trained offline (i.e., compile-time) by profiling targeted applications. Offline training works if a predictor can learn invariant branch relationships that are true at all phases of a program with any inputs. By profiling runs of a program with multiple inputs, one can collect diverse training examples to train powerful machine learning models that can infer such invariant relationships. A key observation of this dissertation is that the training algorithm of CNNs requires high coverage in its training examples. In contrast, previous compile-time branch prediction techniques relied on the representativeness of training inputs. Since achieving coverage is easier than achieving representativeness, training CNNs offline is more viable than previous compile-time techniques.

Tarsa et al. [82] are the first to propose using CNNs with offline training to predict hard-to-predict branches. They show that (1) CNNs can identify individual correlated branches in the global branch history, and (2) CNNs can be trained offline to avoid their expensive training algorithms at runtime. However, they do not address all challenges in using CNNs as branch predictors and their proposed CNN model is impractical. First, while they empirically show that offline training is possible, they do not identify that representativeness is not needed for offline training, which makes it easier to adopt CNNs in practice. Second, their CNN inference engine requires an impractical amount of storage. In this dissertation, I attribute this inefficiency to the general-purpose nature of their CNN model and demonstrate that specialization is the key to achieving high accuracy and storage efficiency for CNN branch predictors. Most importantly, using a few detailed case studies, I show that CNN prediction functions often count the occurrences of correlated patterns, which can be simplified using specialized sum-pooling layers in the model. Using the sum-pooling layers and other architectural changes (e.g., geometric history lengths), I design BranchNet, a CNN model that is significantly more accurate and storage-efficient than Tarsa's CNN.

I propose using trained BranchNet models in two ways. The most straightforward way is to directly use BranchNet as a black-box branch predictor similar to Tarsa's CNN. I design practical and storage-efficient BranchNet inference engines that are used along TAGE-SC-L at runtime. The rationale behind this hybrid approach is that in typical programs, a small number of static branches is responsible for the majority of mispredictions. Thus, it is more storage-efficient to use the expensive BranchNet inference engines to predict only those static branches with noisy histories, and use TAGE-SC-L for all the other branches.

I also propose a novel alternative way of using CNN as a tool to explicitly identify correlated branches, i.e., instead of using BranchNet models as black-box helper predictors, we can examine the trained BranchNet models to *explicitly* identify the minimum set of correlated branches that can be used to accurately predict branches with noisy histories. At runtime, we use a modified TAGE that maintains *filtered* histories. Filtered histories record only the outcomes of correlated branches for each hard-to-predict branch. Because noisy bits are eliminated from the filtered histories, TAGE can now more accurately predict these previously hard-to-predict branches. This approach yields a notable fraction of BranchNet's accuracy improvement with only minor design changes to the baseline predictor.

1.3 Contributions

- I make a new case for branch prediction with offline training. I show that, unlike previously proposed offline training techniques, CNNs rely less on the representativeness of the training data and more on their coverage. The key is exposing enough control flow paths to detect input-independent branch correlations that can be generalized to unseen inputs. Since achieving high coverage is easier than gathering representative inputs, training CNNs offline is more practical than prior compile-time branch prediction techniques.

- I demonstrate that counting the occurrences of correlated branch patterns is useful for an important subset of hard-to-predict branches. With the help of a sum-pooling layer, a CNN learns to count correlated branch patterns, which is an efficient way of aggregating information in long global histories. However, a tabled-based predictor (e.g., TAGE) relies on allocating predictor entries for each history pattern, which is infeasible for long histories with noisy histories.

- I propose BranchNet, a CNN architecture tailored to branch prediction requirements in two ways. One, BranchNet draws inspiration from traditional branch predictors and uses geometric history lengths as inputs. Two, BranchNet uses sum-pooling layers to aggressively compress the information in the global branch history. Because of its specialized design, BranchNet significantly outperforms its predecessor CNN branch predictor. Big-BranchNet (an unrealistically large BranchNet configuration) reduces the average MPKI of SPEC2017 Integer benchmarks by 7.6% (up to 15.7% for the most improved benchmark) compared to an unlimited MTAGE-SC baseline.

- I demonstrate that specialization is key for building storage-efficient and accurate CNN branch predictor inference engines. In addition to the architectural optimizations of BranchNet, I use a novel way to approximate wide convolution filters and sum-pooling layers to build BranchNet inference engines. These approximations enable BranchNet to have the same prediction latency as TAGE-SC-L (4 cycles) and be more storage-efficient. By using BranchNet inference engines along with a 64KB TAGE-SC-L, we can reduce the MPKI by 9.6% (up to 17.7%) and increase the IPC by 1.3% (up to 7.9%) over a 64KB TAGE-SC-L baseline.

- I propose an alternative design for BranchNet inference engines where the convolution layer is replaced by simple correlated branch counters. While this design still uses CNNs during offline training to identify the correlated branches, the inference engines do not need convolution tables and are much simpler to build. This simplicity results in storage efficiency in lower total storage budgets.

- I show that even if on-chip inference engines are infeasible, neural networks are still useful tools to learn information that can be used by simpler predictors. In particular, I demonstrate that BranchNet models can explicitly identify correlated branches at compile-time. I design a TAGE-based branch predictor which uses the information extracted from BranchNet models to remove the uncorrelated branches from the global history. I explain why filtering uncorrelated branches improves the prediction accuracy of TAGE and reduces its storage needs. I show that compared to a 64KB TAGE-SC-L, an iso-latency filtered TAGE reduces the branch MPKI by 3.7% on SPEC 2017 Integer benchmarks, up to 9.4% on the most improved benchmark. To achieve the same order of improvement without filtering, we need a 128KB TAGE-SC-L, which uses 40% more storage than the iso-latency filtered TAGE and also incurs additional prediction latency.

1.4 Thesis Statement

High coverage training sets and model specialization enable Convolutional Neural Networks to learn efficient input-independent prediction functions that count the occurrences of correlated branch patterns in a noisy branch history, resulting in improved branch prediction accuracy, either through storage-efficient helper predictors or by filtering the global branch history of conventional predictors.

1.5 Dissertation Organization

Chapter 2 explains the necessary background to understand this dissertation and provides an overview of the related prior work. Chapter 3 describes the problem of noisy global branch histories for state-of-the-art predictors, demonstrates why convolutional neural networks are a good solution to overcome noise, describes the architecture of BranchNet, and evaluates the effectiveness of BranchNet as an abstract software model without considering hardware limitations. This chapter also details the difference between coverage and representativeness to justify why offline training is viable for BranchNet but does not work for prior branch predictors. Chapter 4 describes how to use BranchNet as a tool to identify correlated branches. Chapter 5 uses case studies to motivate why specialization is needed for accurate and storage-efficient inference engines, proposes a practical design for BranchNet inference engines, and evaluates its impact on accuracy. This chapter also describes the design of a simpler inference engine design that works by counting previously identified correlated branches. Chapter 6 describes how to harness the benefits of BranchNet by filtering the global history of TAGE instead of using on-chip BranchNet inference engines. While inference engines are ultimately more accurate, this alternative approach is significantly easier to implement and adopt. Chapter 7 concludes the dissertation and discusses potential future work.

Chapter 2

Background and Prior Work

From the early years of pipelined processor designs [6, 61, 19], control-flow instructions have been a critical impediment to single-thread performance. High performance relies on a steady supply of instructions to the processor to take advantage of available hardware resources and extract instruction-level parallelism. Without control-flow instructions, achieving high instruction supply throughput is easily achievable as instruction addresses are simply an ordered sequence of numbers, starting from the initial value of the program counter. However, control-flow instructions may change the program counter to any arbitrary address in the program. Thus, the instruction supply unit in the processor cannot determine the correct fetch address until the control-flow instruction is executed and the next instruction address is determined. This problem is particularly worse for conditional branches, which cause the fetch unit to wait until both the address and the outcome (whether the branch was taken or not) are known. To mitigate this problem, processors use compile-time or runtime mechanisms to predict the targets and the outcomes of branches. After a correct prediction, the fetch unit fetches the correct target of the branch, and the instruction supply is not disrupted. However, after an incorrect prediction, the fetch unit fetches instructions from incorrect addresses (i.e., *wrong-path*), and should discard all such wrong-path instructions. Therefore, high prediction accuracy is critical for achieving high fetch bandwidth and performance. In particular, accurately predicting the direction of conditional branches has proven to be challenging, which is the focus of this dissertation. For brevity, I use branch prediction as a shorter term for conditional branch prediction.

The evolution of branch predictors is the result of decades of research towards addressing a diverse set of challenges in accurately predicting branch directions. Some prior work in this domain directly influences the design of state-of-the-art branch predictors, while some research directions are not adopted. Nonetheless, it is important to understand all relevant attempts at improving branch prediction to put the contributions of this dissertation in its proper context. This chapter discusses the related prior work, organized by their role in the design of state-of-the-art branch predictors, and provides the necessary background to the contributions of this dissertation.

2.1 Counter-Based Branch Predictors

Smith's branch prediction strategies [73] are the first use of saturating counters for branch prediction. Smith's most accurate and practical strategy is to use a table of 2-bit saturating counters, which the branch predictor accesses to determine the likely direction of a given branch. The predictor uses a hash of the branch program counter (PC) as the index to access the counter table. Given enough capacity, there is a unique counter corresponding to each static conditional branch in the program. The counter value for a given branch is incremented (saturated at 11) every time the branch is taken, and the counter value is decremented (saturated at 00) each time the branch is not taken. At prediction time, the predictor uses the most significant bit of the counter as the prediction for the branch. Ignoring the marginal warm-up time, this prediction mechanism is 100% accurate for branches that rarely change directions. The 2-bit width of the counter also allows the predictor to ignore occasional outliers in the branch pattern (e.g., a loop is almost always taken, except for the exit case). This predictor is commonly referred to as the *bimodal* branch predictor.

Two problems significantly hinder the accuracy of the bimodal branch predictor. First, many branches continuously change directions, so learning the recent outcomes of the branch does not necessarily provide any information about its future outcomes. Second, multiple branches may easily alias into the same table entry (i.e., the hash of the branch program counters are the same), resulting in ineffectual training for the conflicting branches. Next generations of counter-based branch predictors address these problems.

Lee and Smith [43] identify that prediction based on *branch histories* outperforms the bimodal predictor. The main insight is that short sequences of branch outcomes are often repetitive. Thus, we can examine branch history patterns and identify the most likely next outcome for each possible history pattern. For example, if a branch is alternating between taken and not taken, the branch history looks like ...$NTNTNTNT$, where N represents a not taken instance, and T represents a taken instance. In this case, the likelihood of observing $NTNT \rightarrow N$ is higher than $NTNT \rightarrow T$. Using this observation, Lee and Smith's predictor learns the most likely branch patterns at compile time. At runtime, their predictor keeps an n-bit branch history per static branch and predicts each branch by referring to its corresponding branch history and the compile-time information about the likelihood of branch patterns. While this approach is an improvement over saturating counters, there are two key problems: compile-time training relies on access to representative branch patterns, and not all branches in a program have similar branch patterns. Yeh and Patt [88] solve these two problems by designing an adaptive two-level predictor.

The two-level branch predictor [88] uses a two-step prediction strategy to learn branch history patterns at runtime. First, a history register keeps track of the most recent outcomes of branches. Second, the predictor uses the history register as an index to access a table of 2-bit saturating counters. The two-level predictor uses and updates the counters in the same way as Smith's saturating counters [73]. The table of counters is called the pattern history table (PHT). In effect, PHT learns the most likely next outcomes for each branch history, but unlike Lee and Smith's methodology [43], the patterns are adaptable to the current behavior of a program. Follow-up studies [89, 90] introduce three design choices for the PHT: a single PHT for all branches (*Global*), a unique PHT per branch (*Per-Address*), or a fixed number of PHTs that are used based on a hash of the branch PC (*Per-Set*). Orthogonal to the PHT choice, the same three choices are available for the history register, resulting in $3 \times 3 = 9$ total design choices. For the total storage size of 128K bits, Yeh and Patt found global branch histories with Per-Set PHTs to be the most cost-effective.

The gshare predictor (designed by McFarling [46]) further improves the two-level predictor. Gshare uses a two-level predictor with a global branch history and a global PHT as the baseline, but it uses the exclusive OR of the branch PC and the branch history as the index into the PHT. In effect, this simple modification enables the cost-efficiency of a global two-level predictor, with the additional benefit of separating the 2-bit counters in the PHT by the branch PC. Similar to Gshare, several predictors [75, 42, 50, 11, 18] propose additional modifications to the two-level predictor, which improve the storage efficiency and reduce the destructive aliasing among branches.

A key improvement in counter-based predictors is based on a theoretical observation by Chen et al. [15], which states that branch prediction can be viewed through the lens of the PPM compression algorithm [16]. The key insight is that the two-level predictor and its follow-ups are attempts at learning branch history patterns across all program branches in the most cost-effective manner. Based on this insight, Michaud [47] proposes the PPM-like branch predictor. The PPM-like predictor approximates the PPM compression algorithm by using the combination of a bimodal table and multiple tagged PHTs associated with different history lengths. To make a prediction, the bimodal table and all the PHTs are looked up in parallel. The matching PHT with the longest history length provides the chosen prediction. Note that since the PHT entries are tagged, a PHT does not always provide a prediction. If no PHT provides a prediction, the bimodal predictor is used. The update mechanism inserts new PHT entries at longer history lengths only when the current prediction is inaccurate. As a whole, the PPM-like predictor tries to use only as many history bits as needed, resulting in better storage efficiency.

Seznec and Michaud [66] refine the idea of the PPM-like predictor and introduce TAGE. While the core mechanism is similar (a bimodal predictor with a sequence of tagged PHTs with different history lengths), TAGE improves the insertion and replacement policy and slightly modifies the prediction mechanisms to avoid useless entries. TAGE is often considered the last major breakthrough in counter-based branch predictors. BATAGE [48] is an alternative TAGE-like predictor that improves the prediction accuracy of TAGE without a statistical corrector. However, it does not improve the accuracy compared to TAGE-SC-L [65]. The bias-free predictor [24] removes biased and redundant branches from branch histories, while the TAGE component remains unchanged. The inner-most loop iteration (IMLI) counters [67] are new inputs to branch predictors that are helpful when the outcomes of branches are correlated with the loop iteration counters. The statistical corrector of TAGE-SC-L uses IMLI counters and a history of all branches with the same IMLI counter as additional input features. In summary, while complementary improvements have existed, TAGE is the main component in state-of-the-art branch predictors.

2.2 Perceptron-Based Branch Predictors

The Perceptron branch predictor, designed by by Jimenez and Lin [34], uses a novel alternative approach to counter-based branch prediction. A perceptron [8] is a single-layer neural network that learns a linear function of its inputs to classify the input space into a binary outcome. In the case of the Perceptron branch predictors, the inputs are the branch history bits, and the output is the direction of the next branch. The perceptron output is the dot product of all the input bits and their corresponding *weights*, added to a bias weight (the history bits are treated as -1 and +1 when computing the dot product). The sign of the output determines the prediction. The weights (including the bias) are incremented or decremented to move the output towards a more positive or negative value depending on the actual direction of the branch after it is executed. The result is that Perceptron identifies the linear correlation value of each history bit to the outcome.

The Perceptron predictors suffers from two major weaknesses. First, single-layer neural networks cannot learn nonlinear functions of the inputs, which makes them fundamentally incapable of learning many history patterns. This problem can only be solved by using multi-layer neural networks, which have a much more computationally-expensive training algorithm. Second, the position of branches in the history may be nondeterministic. Thus, learning a correlation with a history bit may not always be the right approach. Follow-up work on Perceptron attempt to solve the second problem.

Path-based neural predictor [35] changes the perceptron predictor to also use branch program counters as input (the term *path history* refers to some representation of program counters of the branches in the global history). In this design, the perceptron weights are associated with both the position of the branch in the history and the program counter of the branch, i.e., branch PCs select the perceptron weights to use as inputs to the dot product. Thus, the path-based neural predictor can identify correlation for branches with nondeterministic positions in the history. The piecewise-linear predictor [36] is a generalization of the original Perceptron and the path-based neural predictor that further improves the accuracy by allowing different path weights per each static branch.

The O-GEHL predictors [63, 62] is a perceptron-based predictor that is built of multiple weight tables, where each table is accessed by a hash of the branch address and the history register. Each table uses a different history length, where the history lengths form a geometric sequence. Since individual weights do not correspond to history bits, instead of performing a dot product, the weights are simply summed together. Tarjan and Skadron [80] generalize this concept and introduce the hashed perceptron, which refers to the family of perceptron-based predictors that use hashes of the branch history to access weight tables. The most recent perceptron-based branch predictor is the Multiperspective Perceptron [30], which is a hashed perceptron that uses a collection of novel and exotic input features in addition to the traditional branch history and path.

It is important to note that the high accuracy of perceptron-based predictors is mainly due to their better scalability with longer branch history lengths. However, TAGE has achieved better storage efficiency and scalability and is now the most accurate single-component branch predictor. TAGE-SC-L [65], the most accurate runtime branch predictor, uses TAGE as its main component and uses a perceptron-based (GEHL) component as a statistical corrector that sometimes overrides TAGE's prediction.

2.3 Identifying Correlated Branches in the Global History

Both counter-based and perceptron-based predictors use the global branch history and the path history as their main input. Global histories are effective because the outcome of many branches is correlated to the outcome of future branches. Evers et al. [20] comprehensively analyze branch correlation and predictability for two-level branch predictors. They show that typically only a few branches in the history matter for prediction accuracy. For each branch in the program, they compute a linear correlation coefficient between the branch and all other branches in their global history. Then they select up to 3 top correlated branches that can most accurately predict the outcome of each branch. Their results show that three correlated branches are sufficient to be as accurate as the g-share predictor [46], which was the best branch predictor at the time. Unfortunately, their brute-force correlation detection algorithm is not computationally feasible when identifying significantly more correlated branches in much longer global histories of current-day branch predictors. This dissertation instead uses convolutional neural networks, which are effective at identifying as many correlated branches as needed in longer global histories.

Evers et al. [20] also attempt to identify the nature of correlated branches. They observe that correlation among static branches is often due to some commonality in the dependence chain. Figure 2.1 provides examples of two common scenarios: when the outcome of a branch impacts the outcome of the next branch (*affectors*), and when the outcomes of two branches are at least partially based on the same condition (*forerunners*). The terms affectors and forerunners were coined by Thomas et al. [83].

Affector branches

```
x = 0
if (some condition) {
    x = 1
}

if (x == 1) {...}
```

Forerunner Branches

```
x = some integer
if (x > 0) {
    ...
}

if (x <= 0) { ... }
```

Figure 2.1: Correlated branches based on dependence chain commonalities.

Thomas et al. [83] design a runtime branch predictor that tracks dataflow dependencies among branches and filters the global history to contain only affector branches. By filtering the history, they improve the prediction accuracy of two contemporary branch predictors, YAGS [18] and Perceptron [34]. Sazeides et al. [60] provides additional analysis on the same approach. While the key insight of this line of prior work is in line with this dissertation, there are several limitations in using dependence chains for identifying correlated branches. First, not all branches in the dependence chain are correlated, and not all correlated branches appear in the dependence chain. Second, there is no mechanism to rank the most correlated branches in the dependence chain. On the other hand, a machine learning approach can directly learn which branches matter the most and rank them accordingly, which is much more suitable for aggressively eliminating all uncorrelated branches.

The Spotlight [84] branch predictor uses profiling to identify a contiguous window in the global history that contains the most correlated branches. This approach is not sufficient for two fundamental reasons. One, correlated branches may appear in nondeterministic positions in the global history. Two, the correlated branches may not show up in a contiguous window in the global history.

The bias-free predictor [24] identifies two types of potentially uncorrelated branches and does not insert them into the history. First, it filters out conditional branches that never change directions, which are by definition uncorrelated. Second, it only maintains the latest outcome of each static branch in the history. While the Bias-free predictor is a step in the right direction, it cannot eliminate all uncorrelated branches. Many uncorrelated branches are not biased, which the bias-free predictor does not eliminate. To completely eliminate noise from the history, we need a filtering mechanism that eliminates all uncorrelated branches.

2.4 Branch Predictors with Offline Training

Many prior studies propose using offline profiling to improve branch prediction. Some train static predictors that simply learn the statistical bias of branches, which is useful for compile-time optimizations, but not for predicting hard-to-predict branches [40, 10, 54, 91]. Some work use profiling to train application-specific predictors, resulting in a comparable accuracy to contemporary dynamic branch predictors [32, 69, 81, 77, 84]. Among them, Spotlight [84] is a gshare-like predictor that uses profiling to identify the most useful fragment of the global branch history. However, Spotlight is still susceptible to shifts in the history and cannot identify correlated branches that appear in nondeterministic positions in the history. Spotlight's training mechanism also relies on exhaustively comparing all possible views of history, which does not scale when training more complicated predictors with long histories. Similar to Spotlight, most prior predictors are either too simple to help with hard-to-predict branches or there is no known way to use them in conjunction with state-of-the-art online predictors. As a result, until recently, the conventional wisdom was that branch prediction using offline training is a dead-end. This dissertation challenges this notion by proposing and analyzing offline training with multi-layer neural networks.

Representativeness vs. Coverage. In addition to the low accuracy of prior predictors, another challenge with offline training was that prior training mechanisms relied on the repetition of exact history patterns (they needed *representativeness*).

Thus, prior work could only perform well when the input sets used for profiling were representative of future runs, which is challenging. For example, for Spotlight to be effective, the positions of correlated branches in the global history should be exactly the same during profiling and at runtime. However, the positions of branches that appear deep in the global history are rarely generalizable to other inputs, especially for the hard-to-predict branches of state-of-the-art runtime predictors. In contrast, deep learning does not need representative input sets; it just needs enough coverage in the training set to expose generalizable input-independent relationships between branches. As long as the training set includes enough examples of different branch behavior (i.e., different program phases that exercise different control flows), deep learning algorithms can identify input-independent correlations that are always true. I use *coverage* to denote the notion that the training set contains examples from all possible branch behaviors. Section 3.4 defines and analyzes coverage in the context of CNN branch predictors.

2.5 Convolutional Neural Networks for Branch Prediction

Tarsa et al. [82] are the first to propose using Convolutional Neural Networks (CNNs) with offline training to predict hard-to-predict branches. Their results show that (1) CNN branch predictors could identify individual correlated branches in the global branch history, and (2) that CNNs could be trained offline to avoid their expensive training algorithms at runtime. This dissertation builds on the insights proposed by their approach, further analyzes the importance of offline training, tailors their CNN architecture to branch prediction, and studies alternative ways of using

CNNs for branch prediction. I will refer to this prior work as Tarsa's CNN and provide more details of its contributions throughout this dissertation and contrast their work with mine.

2.6 Complementary Techniques to History-based Predictors

In general, the remaining branch mispredictions are due to branches that are hard to predict for very different reasons. Because a one-size-fits-all solution is no longer viable, recent research in branch prediction can be viewed as complementary techniques that need to work together.

Some proposed branch predictors use specialized predictors for a particular subset of hard-to-predict branches. Adileh et al. [3] propose extensions to the ISA to completely eliminate mispredictions of probabilistic branches. The key idea is that the correctness of the program does not depend on the outcomes of each dynamic instance of a probabilistic branch. Instead, the past outcomes of probabilistic branches can be buffered and used as a correct prediction. Farooq et al. [21] design a specialized branch predictor for simple data-dependent branches. The compiler searches the dependence chain of branches at compile time and detects if there is a single store-load pair leading to the branch. For such cases, the direction of the branch can be pre-computed when the store executes, saved in a data structure, and used as the prediction for the branch.

An important body of prior work is pre-computing branch outcomes instead of predicting them [94, 95, 58, 13, 12, 68]. Among recent proposals, Srinivasan et al. [76] propose Slipstream 2.0, which is an improved Slipstream [78] processor that runs ahead of the main core, pre-computes branch directions in a shortened version of the program, and sends the pre-computed directions back to the main core to be used as predictions. Pruett and Patt [55] also use pre-computation, but instead of shortening the whole program, their predictor extracts lightweight dependence chains for hard-to-predict branches and executes them on a small dedicated engine in the main core. Pre-computation is promising, especially for data-dependent branches, however, improving the prediction accuracy of the baseline history-based branch prediction still remains critical and complements the pre-computation-based predictors.

2.7 Machine Learning for Other Computer Architecture Prediction Tasks

Branch prediction is not the only prediction task that can be improved with machine learning. Prior work has used machine learning to improve cache replacement policy, memory controller scheduling, and data prefetching. While these prediction tasks have different requirements and concerns from branch prediction, some key insights may transfer among prediction tasks.

Ipek et al. [29] use reinforcement learning to optimize the scheduling policies of a DRAM memory controller. While reinforcement learning is not a natural formulation for branch prediction, this work is a good example of adapting simple table-based machine learning methods to the practical constraints of a processor.

Shi et al. [71] use deep learning to learn cache replacement strategies. Using a heavy-weight, they discover two key insights about the requirements of an accurate cache replacement policy: a long control-flow history is useful for cache replacement, and only a few program counters in the history provide all the necessary information. Using these insights, they develop a practical runtime mechanism that improved the cache replacement policy. This work is a good example of how insights learned from computationally-expensive neural networks may be used to develop simpler mechanisms.

Hashemi et al. [25] demonstrate that Long Short-Term Memories (LSTMs) are in theory capable of learning access patterns to prefetch memory locations. Shi et al. [72] improve their model and show that there is headroom to reduce the size of deep learning models by taking advantage of the inherent properties of a given prediction task.

Pythia [7] is a table-based reinforcement learning data prefetcher. Pythia is another example of a simple and practical adaptation of a classical machine learning technique to the constraints of microarchitecture design.

2.8 Convolutional Neural Network Basics

Convolutional Neural Networks (CNN) are state-of-the-art in both image classification [79, 26] and sequential tasks like natural language understanding [87]. When used as a branch predictor, a CNN first identifies important branch patterns in the global history and then classifies the branch as taken or not taken using the identified patterns.

This section provides a high-level description of a simple CNN branch predictor. The goal is to introduce the terminology and provide an intuition for how the

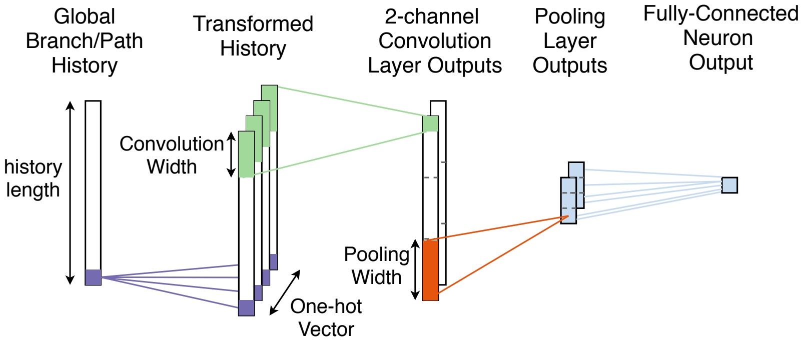


Figure 2.2: Dataflow in a simple CNN branch predictor.

CNN components work together to predict branches.

2.8.1 CNN Building Blocks

Figure 2.2 shows the data flow for branch prediction using a simple CNN. The CNN takes the global branch and path history (program counters and directions of branches) as input, operates on the input using a sequence of operations, and finally produces a prediction. The critical operations are referred to as layers. The layers operate using a collection of trainable parameters (*weights*). The combination of the CNN layers and their trained parameters form a *CNN model*.

Input as one-hot vectors. CNNs assume that the magnitude of each input conveys information about the input. For example, the inputs to a CNN image classifier convey the color intensity of an image at each pixel. However, the inputs to a branch predictor are branch program counters and directions, whose magnitudes convey nothing about the branches. Thus, we need to represent branches in a format that makes it easier for CNNs to distinguish different program counters. One solution is to represent components in the history as one-hot vectors.

Input as embeddings. Embeddings are the state-of-the-art method of converting discrete inputs to vectors of real numbers that are more suited as inputs to neural networks [23]. Embeddings are implemented as a 2-dimensional table, where each discrete value of an input corresponds to a row in the table. If the input to a neural network is a sequence of discrete inputs, an embedding table can be used to convert them to a sequence of vectors of trainable real-number values. For large-enough discrete numbers, embeddings often lead to a more efficient solution than simply using one-hot vectors.[1] For example, Hashemi et al. [25] use embeddings to represent PC and memory addresses in a model for data prefetching, which is very similar to a branch predictor that represents each branch using its PC and direction. Thus, as an alternative to one-hot-vectors, CNN branch predictors may use embeddings to feed the branch history into the subsequent layers.

Convolutional layers. At a high level, a convolution layer identifies the occurrences of features in its input [41, 23]. The set of weights that are trained to identify a feature is called a *filter*. The *convolution width* controls the number of neighboring items that form a feature. For branch prediction, the neighboring items are the neighboring entries in the branch/path history. Applying a filter to the inputs produces an *output channel*. For branch prediction, each filter identifies the presence of a specific correlated branch pattern in the history and marks its location by outputting a non-zero value to the corresponding output channel for the filter.

[1] E.g., representing a 12-bit program counter as a one-hot vector requires $2^{12} = 4096$ trainable weights for a 1-wide convolution filter, but embeddings can still be effective with much fewer weights (e.g., 32).

Sum-pooling layers. A sum-pooling layer reduces the computational requirements of subsequent layers by combing the neighboring outputs of the convolution output channel into a sum [23]. The *pooling width* defines the number of neighboring outputs that are summed together. Effectively, the outputs (i.e. generated sums) of a sum-pooling layer indicate the occurrence counts of the feature identified in each channel. Sum-pooling reduces the computational needs of the next CNN layer at the cost of discarding fine-grained positions of identified features. This is often a good trade-off for branch prediction because the exact positions of correlated branches do not matter.

Fully-connected layers. A fully-connected layer is made of multiple neurons, where each neuron learns a linear function of all its inputs [23]. It is possible to cascade fully-connected layers to learn nonlinear functions of convolution outputs. For branch prediction, the fully-connected layers map the identified feature counts to a prediction.

Activation Functions. Activation functions are non-linear element-wise transformations that are used by convolution and fully-connected operators. Without activation functions, neural networks cannot learn non-linear functions. I use ReLU [53], Sigmoid, and Tanh (hyperbolic tangent) activations throughout this dissertation.

Batch Normalization. A batch normalization operation normalizes each output channel to a standard normal distribution using the mean and the variance of its outputs during training [28]. While the normalization operation does not directly add to the prediction capability of a neural network, it has been shown to guide the optimization algorithms toward better solutions and mitigates overfitting. In this dissertation, I use a normalization operation before the activations in convolutional and fully-connected layers.

2.8.2 Training Algorithm

CNNs are trained using a large set of input and expected output pairs (*the training set*) that define the desired behavior of the model. Conceptually, the training algorithm constantly iterates through the examples in the training set and identifies consistent signals for producing the expected output. Since this algorithm (Stochastic Gradient Descent [57] using Backpropagation [59]) is computationally expensive, the training has to be done offline using profiling. Thus, a good training set for branch prediction should contain examples from multiple input sets and exercise different control flow paths, which enables the CNN to learn invariant branch relationships.

Chapter 3

BranchNet: a Convolution Neural Network for Branch Prediction

This chapter describes BranchNet, a Convolutional Neural Network (CNN) that can accurately predict a category of branches that are fundamentally hard to predict for state-of-the-art runtime branch predictors.[1] In particular, using offline training, BranchNet can identify correlated branches even in the presence of noise in the global history. BranchNet is the central component of all the contributions of this dissertation and can be used directly using on-chip inference engines or as a tool to explicitly identify correlated branches that can enable other branch prediction techniques.

3.1 The Problem of Noise in the Global Branch History

As explained in Chapter 2, not all branches in the global branch history are correlated to the next branch outcome. In this section, I explain how the existence of uncorrelated branches in the branch history is harmful to state-of-the-art runtime branch predictors. For brevity, I use *noise* to generally refer to all the uncorrelated

[1] The main contributions of this chapter have been previously published in a paper that I co-authored [92].

branches that are not useful for accurate branch prediction, and I use *noisy branch* to denote a branch that is hard to predict due to excessive noise in its history.

The root cause of the difficulty in predicting noisy branches is that the storage requirements of state-of-the-art runtime branch predictors grow exponentially with the addition of each noisy bit in the history. Despite differences in their prediction mechanism, both TAGE [66] and the hashed Perceptron [80] work by hashing the global branch and path history into one or more indices to access prediction tables. Ideally, the predictors would allocate unique table entries for each history pattern they observe. In practice, they employ storage-saving mechanisms to avoid redundant allocations for the most common branch behaviors (e.g. TAGE uses an approximation of PPM compression [16]). However, when the global history is noisy, i.e., uncorrelated branches constantly change directions or branches appear in nondeterministic positions in the history, these storage-saving mechanisms do not work well, requiring the online predictors to allocate unique entries for all possible history patterns. The number of entries required to remember all history patterns is an exponential function of the history size. When these entries are not available, the predictors cannot produce accurate predictions. Even if capacity were available, the runtime predictors would require a long time to warm up the large number of table entries, and can never generalize their predictions to unseen history patterns.

3.1.1 Example: Uncorrelated Branches

Figure 3.1 uses a simple example to demonstrate the impact of noise because of uncorrelated branches. In the example program, the direction of the branch in

```
 1 int x = rand();
 2 int y1 = rand(), y2 = rand(), y3 = rand();
 3
 4 if (x > 0) {...} // correlated branch
 5
 6 if (y1 > 0) {...} // uncorrelated branch
 7 if (y2 > 0) {...} // uncorrelated branch
 8 if (y3 > 0) {...} // uncorrelated branch
 9
10 if (x > 0) {...} // example branch
```

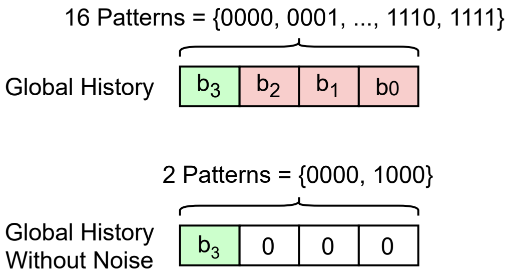

16 Patterns = {0000, 0001, ..., 1110, 1111}
Global History
b3
b2
b1
b0
2 Patterns = {0000, 1000}
Global History
Without Noise
b3
0
0
0

Figure 3.1: A noisy branch due to uncorrelated branches.

line 10 can be perfectly predicted by knowing the direction of the first branch in line 4. In theory, a two-level predictor needs only two prediction counters to predict this branch: one counter to learn the outcome if the branch in line 4 is not taken, and one counter if it is taken. Unfortunately, due to the presence of intermediate uncorrelated branches, a two-level predictor with a 4-bit global history needs 16 counters to learn a prediction for each possible history pattern: {0000, 0001, 0010, ... 1111}. But if the history had no noise (e.g., if branches in lines 6-8 were always not taken), the total number of patterns would be two {0000, 1000}, which is the ideal case. While the state-of-the-art runtime branch predictors are more complicated than a simple two-level predictor, the exponential growth of history patterns remains a problem.

```
int x = rand();
if (x > 0) {...} // correlated branch

int N = rand();
for (int i = 0; i < N; ++i) {
    ... // unrelated computation
}

if (x > 0) {...} // example branch
```

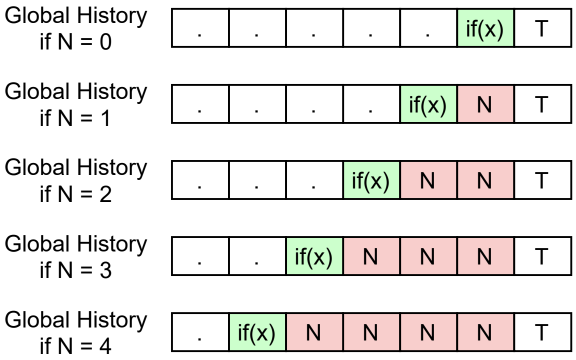


Figure 3.2: A noisy branch due to nondeterministic history positions.

3.1.2 Example: Nondeterministic Positions of Correlated Branches

Figure 3.2 demonstrates noisy history bits that result in nondeterministic positions of correlated branches in the history. In this example, instead of having uncorrelated branches that may change directions, there is a loop that iterates for a random number of iterations. Each iteration of the loop inserts a not taken instance of the loop branch (represented by N) in the history, ended with a taken instance (represented by T). As a result, the correlated branch (represented by *if(x)*) appears in different positions in the global history. Not only does this introduce redundant history patterns, but it also causes prediction mechanisms like the Original Percep-

```
1 int x = 0;
2 int N = rand();
3 for (int i = 0; i < N; ++i) {
4     if (some condition) {
5         x += 1;
6     }
7 }
8
9 if (x > 2) {...} // example branch
```

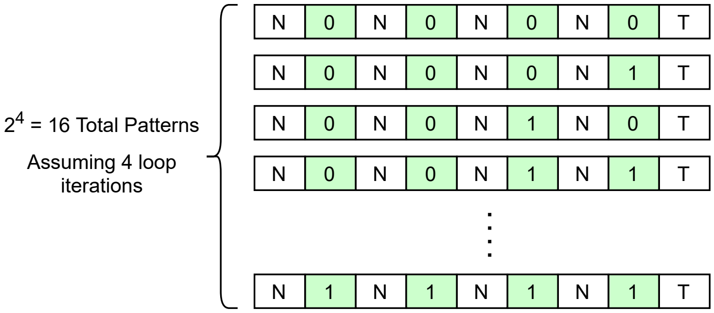


Figure 3.3: A correlated branch that causes an exponential number of patterns.

tron to not reliably identify the correlated branches. An accurate predictor has to identify correlated branches using their program counters regardless of their positions in the history.

3.1.3 Example: Exponential Number of Correlated Branch Patterns

Figure 3.3 shows a case where correlated branches have noisy behavior. The branch in line 4 is the correlated branch that determines if the variable x is incremented or not. The loop branches (indicated by N for each fall-through loop instance, and T for the exit case) are not noisy because they do not produce redundant history patterns. Despite the lack of noise, the total number of history patterns is an exponential function of the number of loop iterations. This is an example where the correlated branches produce history patterns beyond the capacity of state-of-the-art branch predictors.

3.1.4 Learning Prediction Functions to Overcome Noise

To predict noisy branches, we need an alternative prediction mechanism that can ignore any noise in the history and is resilient to nondeterministic history positions and an exponential number of history patterns. This dissertation demonstrates that a suitable prediction function is a 3-step process. First, the prediction function identifies the correlated branches using their program counters regardless of their positions, and thus can ignore noisy uncorrelated branches. Second, the prediction function aggregates the information across long global histories by simply counting the occurrences of correlated patterns, which often reduces an exponential number of history patterns to only a few representative counts with only negligible loss of information. Third, the prediction function should be able to learn arbitrary functions of the aggregated view of the correlated branches. The rest of this chapter demonstrates that convolutional neural networks are suitable for learning these prediction functions.

3.2 Motivation Example: Using Convolutional Neural Networks to Predict a Noisy Branch

This section uses the source code in Figure 3.4 to show how CNNs can predict otherwise hard-to-predict branches. The code is a simplified version of a hot segment

```
int x = 0;
for (int i = 0; i < N; ++i) {
  if (random_condition(alpha)) { // Branch A
    x += 1; // x increments if Branch A is not taken
  }
}

uncorrelated_function();

for (int j = 0; j < x; ++j) { // Branch B
  ...
} // exits when Branch B is taken
```

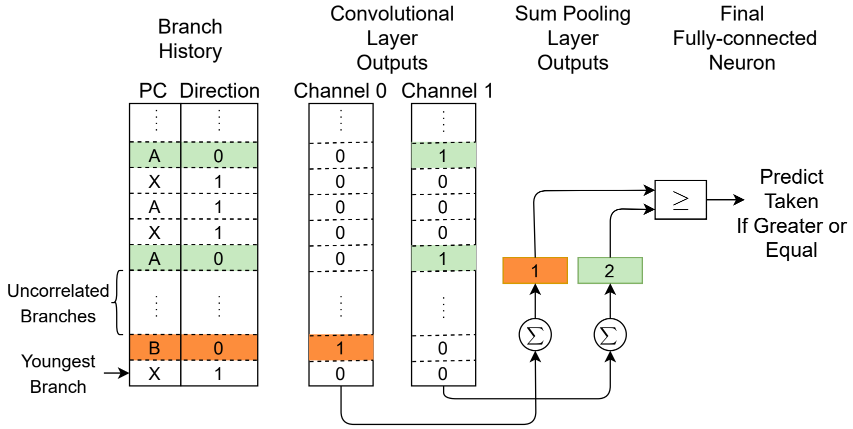


Figure 3.4: A program with a hard-to-predict branch (Branch B) and a trained CNN that can accurately predict the branch.

of the benchmark *leela*, which is responsible for a significant fraction of the total number of mispredictions.

3.2.1 Can We Predict Branch B Using the Global History?

Branch B is the exit branch of the second loop in the source code. The number of iterations of the second loop equals the variable x, which is set by the first loop.

Branch B is taken only if the variable j (the loop variable) is equal to the variable x. There is enough information in the global history to infer the values of x and j: x equals the number of not taken instances of Branch A in the history, and j equals the number of not taken instances of branch B. Thus, in theory, a branch predictor should be able to predict this branch accurately.

3.2.2 Why Do State-of-the-Art Predictors Fail to Predict Branch B?

Unfortunately, state-of-the-art predictors have no way of knowing which branches in the global history are useful for prediction. Thus, as explained in Chapter 2, they hash the whole global history and attempt to learn a prediction for the history pattern as a whole. However, due to the large number of loop iterations, the probabilistic nature of the correlated branches, and the uncorrelated branches close to Branch B, the number of observable history patterns for Branch B is beyond what online predictors can predict. For example, if *N=10*, and *uncorrelated_function* has 20 conditional branches, a TAGE-like predictor has to allocate storage for at least $10 \times 2^{(10+20)}$ history patterns. This amount of storage is infeasible. As a result, Multi-Perspective Perceptron and TAGE-SC-L predict branch B with 81% accuracy, which is only slightly more accurate than always predicting not taken with 78% accuracy.

Note that even if a runtime table-based predictor has enough storage to remember all history patterns it sees, it will take a long time to warm up and can never generalize its predictions to the history patterns it has not seen.

3.2.3 How Does a CNN Predict Branch B Accurately?

A CNN can directly infer the values of variables x and j from the global history, allowing it to predict Branch B both accurately and efficiently. Figure 3.4 shows the outputs of a manually trained CNN that predicts the direction of Branch B 100% accurately. The input on the left is a snapshot of the global history before predicting branch B. The program counters of branches that are not involved in the prediction (i.e. uncorrelated branches) are marked as X. The history is encoded as one-hot vectors[2] (not shown in the figure for brevity) and fed into a convolutional layer. The convolution width is 1 and there are 2 channels. Channel 0 is trained to identify the not-taken instances of Branch B. Channel 1 is trained to identify not-taken instances of Branch A. For this example, the CNN uses a sum-pooling layer as wide as the history. Thus, the outputs of sum-pooling are simply the counts of not taken instances of Branch A and Branch B, which equal the values of variables j and x right before the branch executes. The final fully-connected neuron is trained to predict taken only if $j \geq x$ (sum-pooled channel 0 $\geq$ sum-pooled channel 1), resulting in 100% prediction accuracy.

3.2.4 Does Offline Training Work?

Thus far, I have shown that a manually configured CNN can predict Branch B. Now, I show that we can train a CNN offline using profiling. Suppose the random condition in line 3 of Figure 3.4 is set using a Bernoulli distribution that is true with

[2] As Section 2.8.1 explains, one can replace one-hot transformations with trainable embeddings for a more efficient representation of branches.

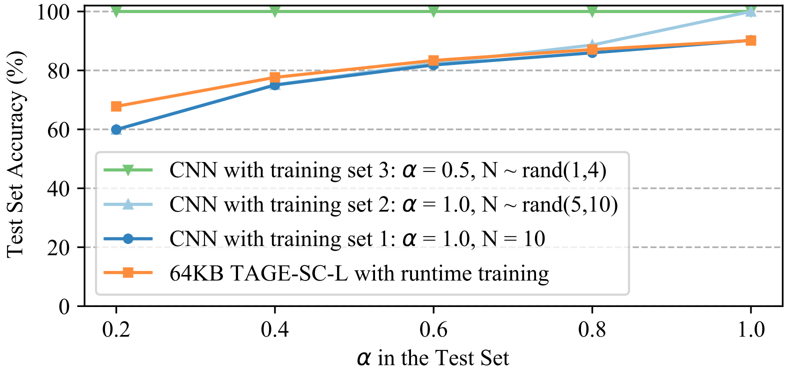


Figure 3.5: Accuracy of predicting Branch B from Figure 3.4. $N \sim rand(5, 10)$ in the test set.

probability α, and N is set using a uniform distribution with adjustable minimum and maximum. I collected three different training sets for Branch B with three program inputs: (1) $N = 10$, $\alpha = 1$, (2) $N \sim rand(5, 10)$, $\alpha = 1$, and (3) $N \sim rand(1, 4)$, $\alpha = 0.5$. I then evaluated the accuracy of CNNs trained on each of the three training sets on runs of the program with $N \sim rand(5, 10)$ and α ranging from 0.2 to 1. I also evaluated the accuracy of a 64KB TAGE-SC-L (with normal runtime training) on the same test sets. Figure 3.5 shows the results. We see that CNNs trained using sets (1) and (2) perform even worse than TAGE-SC-L, especially when $\alpha < 1$. These two training sets do not expose input-independent branch relationships to the CNN. When training with the set (1), the CNN likely learns that the length of the second loop is always 10, which is not true. When training with the set (2), since Branch A is always not taken, the CNN might learn that the length of the second loop equals the length of the first loop, which is true only when $\alpha = 1$. However, the branch behavior in the set (3) is diverse enough to expose the input-independent correlation.

Thus, the CNN trained with the set (3) can predict Branch B with 100% accuracy for runs with any value of α.

3.2.5 Is Representativeness of Profiling Required?

No! Note that the range of N in the set (3) ($N \sim rand(1, 4)$) does not overlap with the range of N on evaluation runs ($N \sim rand(5, 10)$) at all. Yet, the trained model still generalizes perfectly to history patterns it has not seen. The key criterion for a good training set is good coverage of different branch behaviors, not representativeness of history patterns.

3.2.6 Can a CNN Predict All Branches?

A CNN is only accurate if there exist persistent branch relationships that are independent of input data and program phase behavior. Sometimes no branch in the global history can provide any information about the outcome of the target branch. For example, some branches depend on data that was stored in memory long before the branch executes. In this case, there is nothing in the recent branch history that is correlated to the data in memory. Using only global branch history as input, it is impossible to learn any branch prediction strategy offline. Thus, the baseline runtime branch predictor is still needed to predict these branches.

3.2.7 Can Other Machine Learning Models Predict Branches?

Any sophisticated learning model can learn invariant branch relationships from large training sets. For example, Recurrent Neural Networks (RNNs) can also predict the same type of hard-to-predict branches as BranchNet. While this Chapter provides some quantitative comparison of different types of neural networks, the focus of this dissertation is on using convolutional neural networks because of a clearer path towards practical inference engines. Section 5.2.5.5 discusses the problems of using RNNs as inference engines.

3.3 BranchNet

Having described the general principles behind using CNNs for branch prediction, I now present BranchNet. BranchNet refers to a family of CNN models, with configurable parameters and design features. The goal of this section is to show the available headroom in using CNNs for branch prediction. Thus, this section introduces Big-BranchNet, a version of BranchNet that is optimized for prediction accuracy and does not have a practical on-chip inference engine. This section also details the training process of BranchNet and shows the results of using Big-BranchNet for predicting noisy branches compared to a TAGE-based baseline.

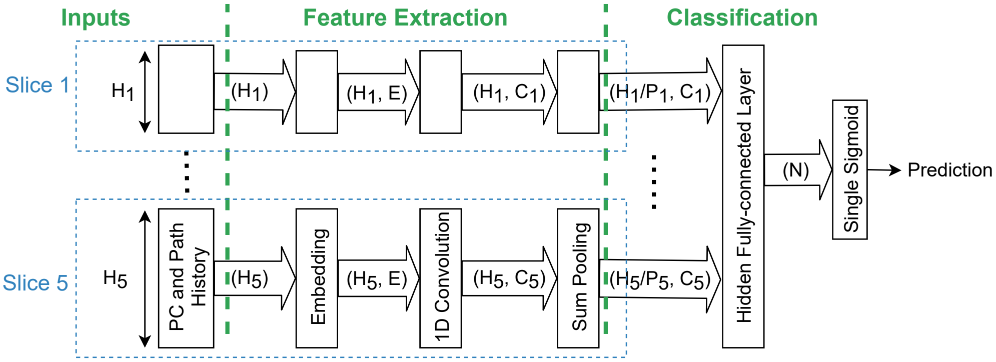


Figure 3.6: High-level diagram of the BranchNet CNN architecture.

3.3.1 BranchNet Building Blocks

Figure 3.6 shows a high-level diagram of BranchNet. The model is composed of 5 feature extraction sub-networks and two fully-connected layers. I call each feature extraction sub-network a *slice*.[3] Each slice uses an embedding layer, a convolution layer, and a sum-pooling layer to extract features out of the branch history. Different slices operate on different history lengths, with the history lengths forming a geometric series. The benefits of using geometric history lengths are well studied for branch predictors [62]. Finally, the outputs of the slices are concatenated and fed into two sequential fully-connected layers to make a prediction.

The following paragraphs explain the functionality of all major components of the model. Note that BranchNet is defined in terms of a set of architecture knobs that determine the size of the model. Typically increasing the size of the model results in better prediction accuracy albeit with diminishing returns.

History Format. To represent each branch, the least significant bits of its program counter and its direction are concatenated and treated as an integer. Thus, if we use p bits of PC, and a history size of H for a slice, the input history is a 1-dimensional array of H integers, ranging from 0 to $2^{p+1} - 1$.

Embedding Layers. Embeddings transform each branch in the input history to a dense vector of numbers. The size of the embedding vectors is controlled by knob E. Note that as mentioned in Section 2.8.1, we could have used one-hot

[3] In deep learning, sub-networks in a larger neural network are often called branches. I avoid this terminology and use the term ”slice” to avoid confusion with branch instructions.

encodings instead of the embeddings, but I found that using embeddings improved the convergence and training time of BranchNet.

Convolutional Layers. C_i denotes the number of output channels for slice i and K denotes the convolution width. With more output channels, BranchNet can learn more correlated features in the branch history. With a larger K, BranchNet can identify longer sequences of correlated branches. BranchNet always uses a convolution stride of 1. The convolution operation is followed by batch normalization and ReLU activations. Since BranchNet has only a few layers, the choice of activation does not significantly impact its training quality. Thus, even though using ReLU activations slightly improves the training time compared to using Tanh or Sigmoid functions, it does not significantly impact the accuracy if the model is always trained until convergence.

Sum-Pooling Layers. In each slice, a sum-pooling layer down-samples the convolution outputs with a width and stride of P_i. BranchNet uses geometric pooling widths proportional to the history lengths of each slice. Larger pooling widths for longer history lengths work well because branches become noisier deeper into the history. By eliminating fine-grained positions of the identified features, wide sum-pooling layers make BranchNet resilient against shifts in history.

Fully-connected Layers. The first fully-connected layer consists of N neurons. Each neuron is connected to the outputs of all slices. The fully-connected neurons are followed by batch normalization and ReLU activation functions. The final fully-connected layer is made of a single neuron with a Sigmoid activation function to make the final prediction.

Table 3.1: Big-BranchNet and Tarsa-Float architecture knobs.

Knob	Big-BranchNet	Tarsa-Float
H: History sizes	42,78,150,294,582	200
C: Convolution channels	32,32,32,32,32	32
P: Pooling widths	3,6,12,24,48	1
p: Branch PC width	12	7
E: Embedding dimensions	32	N/A
K: Convolution width	7	1
N: Hidden neurons	128, 128	N/A

3.3.2 Big-BranchNet and Tarsa's CNN

BranchNet can be viewed as a generalization of Tarsa's CNN.

The reasoning behind using BranchNet for branch prediction is equally true for Tarsa's CNN, the first CNN branch predictor that I described in Section 2.5. In fact, the architecture knobs of BranchNet can be configured to be the same architecture as Tarsa's CNN. Thus, BranchNet can be considered a generalization of Tarsa's CNN. Table 3.1 reports the architecture knobs for Big-BranchNet and Tarsa's CNN (denoted by *Tarsa-Float*).

The major differences between Big-BranchNet and Tarsa's CNN are as follows. (1) Through using multiple slices, BranchNet uses geometric history lengths. This is a good example of how we can adapt insights from conventional branch predictors to CNN models. (2) Big-BranchNet uses pooling widths that are proportional to the history length. In contrast, Tarsa's CNN does not use any pooling. The pooling layer enables Big-BranchNet to efficiently aggregate information across longer histories and it results in smaller fully-connected layers. After sum-pooling, the size of the fully-connected layer inputs is 1920. While the size of the fully-connected layer inputs of

Tarsa's CNN is 6400. (3) BranchNet uses wide convolution layers (each filter examines a 7-branch window). Tarsa's convolution filters examine one branch at a time. (4) Big-BranchNet uses hidden fully-connected layers.

At the first glance, it may seem like that Big-BranchNet is just a bigger and more accurate version of Tarsa's CNN. However, the design differences of BranchNet (namely geometric history lengths, sum-poolings, and wide convolutions) enable better storage-efficiency for building practical inference engines. This is described in detail later in Chapter 5. For now, this chapter investigates the prediction accuracy without considering the size and the storage efficiency of the models.

3.3.3 Prediction Strategy

BranchNet as a Helper Predictor. Compared to traditional runtime predictors, BranchNet is a complex and costly prediction mechanism. For noisy branches, using BranchNet is cost-efficient. For other branches, BranchNet does not provide any value because the baseline runtime predictor already predicts those branches accurately, or because the branch is hard to predict for some reason other than noisy history bits. Thus, BranchNet is not used as a replacement for state-of-the-art runtime predictors. Instead, BranchNet is added as a helper predictor to the runtime predictors and provides predictions for only the branches that benefit the most from BranchNet.

Per-Branch Models. There are two general choices in using CNNs as helper predictors: a CNN model responsible for predicting all noisy branches, or a collection of CNN models where each model is responsible for predicting one noisy branch. The latency of evaluating small per-branch models is less than evaluating a big global model. On the other hand, a global model may be more storage-efficient because different branches may be correlated with similar history patterns. In this dissertation, I use per-branch BranchNet models to optimize for the prediction latency.

3.3.4 Training Process

The prediction capability of BranchNet comes at the cost of computationally-expensive training and the need for large training data. Therefore, it is not possible to train BranchNet at runtime. Instead, BranchNet uses offline (i.e., compile-time) training by profiling targeted applications. Offline training works if a predictor can learn invariant branch relationships that are true at all phases of a program with any inputs. By profiling runs of a program with multiple inputs, one can collect diverse training examples to train powerful machine learning models that can infer such invariant relationships. After offline training, one can attach the trained models (i.e., the collection of weights that represent the branch relationships) to the program binary. At runtime, the branch predictor uses the trained models to predict the directions of these hard-to-predict branches without further training.

Step-by-Step Process. First, profile the target program with a diverse set of inputs and collect example branch traces. Divide the profiled traces into two mutually exclusive sets: the training set and the validation set. Measure the MPKI of the baseline runtime predictor (e.g., TAGE-SC-L) on the validation set and identify the 100 most mispredicting branches. Train a BranchNet model for each of the 100 most mispredicting branches using the branch traces from the training set. Measure the MPKI of the trained BranchNet models on the validation set. Compute the MPKI reduction compared to the baseline predictor for all 100 branches. Select the most improved branches and mark them to be predicted by BranchNet. When using BranchNet inference engines (Chapter 5), the number of selected BranchNet models is limited by the capacity of the engine. For this dissertation, I also produce a test set: a third set of branch traces generated with input data different from the training set and the validation set. I use the test set to report the final accuracy of BranchNet.

3.3.5 Results

This section shows the effectiveness of BranchNet on SPEC2017 Integer Speed Benchmarks. I chose SPEC benchmarks because of access to various inputs for the same benchmark. Having a diverse set of inputs for each benchmark is necessary to test the generalization of offline training to unseen data. While this section does not show the results for any practical branch predictor, Big-BranchNet results demonstrate the available headroom of branch prediction with offline deep learning.

3.3.5.1 Evaluation Methodology

I run each SPEC2017 Integer Speed benchmark using inputs provided by SPEC (*train* and *ref* inputs) and Alberta inputs [5]. I collect up to 10 branch traces from each workload's representative regions using SimPoints [70]. I then train BranchNet models using the process described in Section 3.3.4. Table 3.2 shows the partitioning of inputs to generate the datasets needed for offline training. All numbers reported in this section refer to measurements on the test set (the SPEC ref inputs), adjusted

Table 3.2: Inputs of SPEC workloads used to evaluate BranchNet.

	Inputs	Purpose
The training set	Alberta	Training BranchNet models
The validation set	SPEC train	Identifying best BranchNet branches
The test set	SPEC ref	Final evaluation of accuracy

according to SimPoint weights. I do not use SimPoint weights during training to encourage a general solution that works even for infrequent phases. My training infrastructure takes about 1 hour and 20 minutes on 4 Nvidia GTX 1080 Ti GPUs to train all BranchNet models for a given benchmark and evaluate the BranchNet models on all validation set and test set traces. Training could be easily sped up with more GPUs since BranchNet models are trained in parallel. The evaluation infrastructure is open-sourced and available on GitHub [1].

I make a slight adjustment to the training and validation inputs of *gcc* and *xz*. As part of their inputs, these two benchmarks have high-level control flags (optimization settings and compression level, respectively). Since these control flags likely do not change frequently in deployment, it is reasonable to train specialized CNN models targeting runs with certain execution flags. The data inputs remain different in training, validation, and test sets.

The Adam optimizer [39] with a binary cross-entropy loss is used for training. Training is always done for 4000 steps regardless of the training set size. The training batch size is 2048. The training steps are divided into groups of 1000 steps each with learning rates of 0.1, 0.02, 0.004, and 0.0008, respectively. The same training hyperparameters are used for training BranchNet and Tarsa's CNN.

I use arithmetic mean to report the average MPKI of a collection of bench-

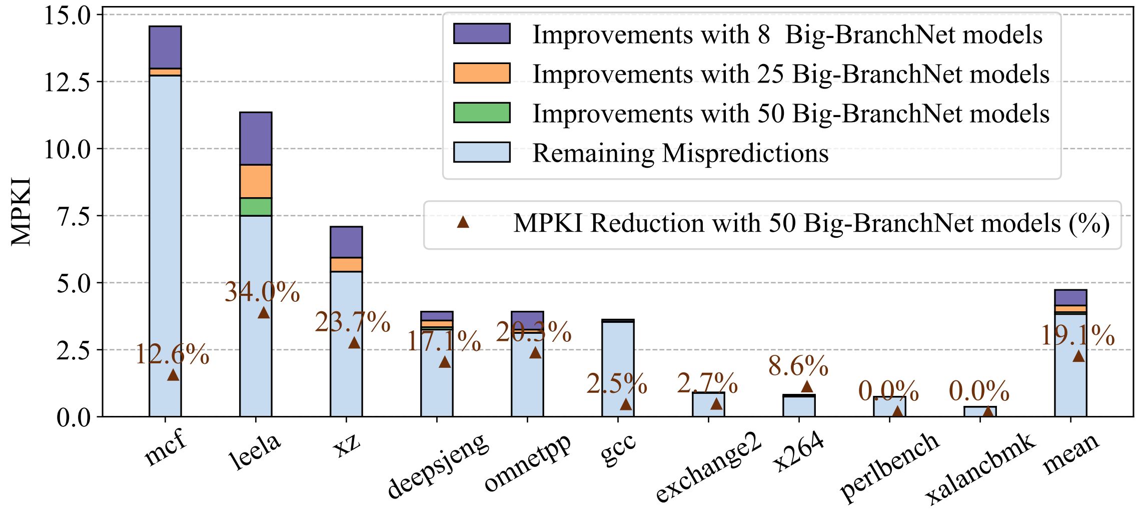


Figure 3.7: MPKI Reduction of using Big-BranchNet to predict a few noisy branches along a 64KB TAGE-SC-L.

marks. The mean MPKI reduction is the relative MPKI of reduction of the mean MPKI of a given configuration compared to the mean MPKI of the baseline.

3.3.5.2 Measuring the Impact of Improving a Few Hard-to-Predict Branches

Figure 3.7 shows the potential of using Big-BranchNet to predict the top few noisy branches that benefit the most from CNNs. Each bar shows the MPKI of 64KB TAGE-SC-L when running SPEC2017 Integer Speed benchmarks. The segments in each bar show the mispredictions that could be avoided if we use Big-BranchNet to predict up to 8, 25, or 50 static branches. The figure demonstrates that for most benchmarks, predicting 8 branches with Big-BranchNet is sufficient for significant overall MPKI reduction, and often predicting more than 25 branches with CNNs has diminishing returns. This result justifies the hybrid approach of using BranchNet to

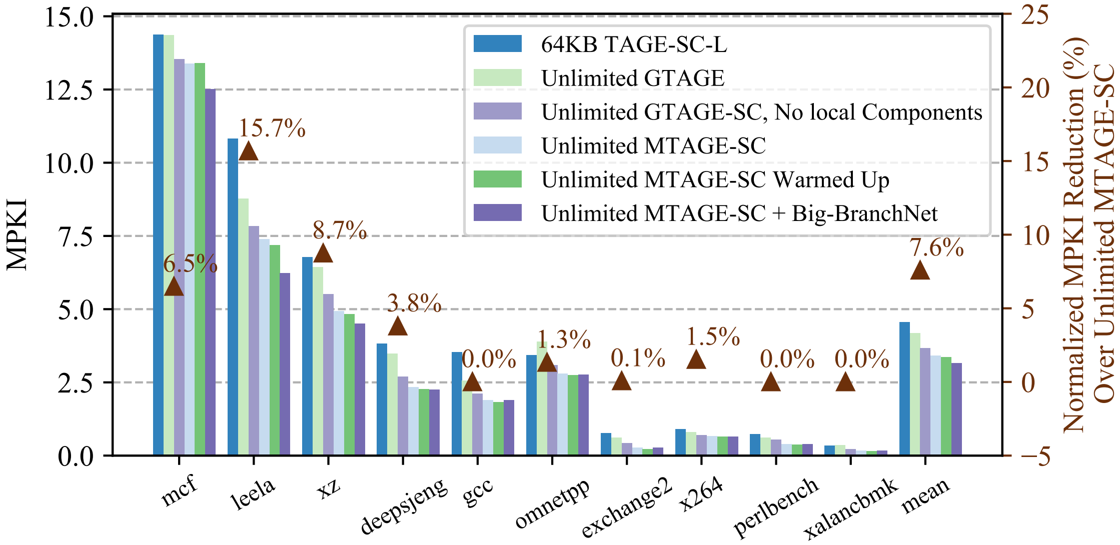


Figure 3.8: MPKI of MTAGE-SC and Big-BranchNet on SPEC2017 benchmarks.

predict a few noisy static branches and using state-of-the-art runtime predictors for all other branches.

Note that since the same input signals for TAGE-SC-L and Big-BranchNet, the difference in prediction accuracy is mainly due to the capability of CNNs in identifying useful information in the global history. Thus, the 19.1% reduction in MPKI can be interpreted as an approximation for the fraction of branch mispredictions due to noisy history. The remaining mispredictions are due to data-dependent or inherently unpredictable branches.

3.3.5.3 Comparison to Unlimited MTAGE-SC

Figure 3.8 shows the MPKI reduction of using Big-BranchNet along with MTAGE-SC, the best predictor in the unlimited storage category of CBP 2016 [64].

Adding Big-BranchNet to MTAGE-SC reduces the average MPKI from 3.42 to 3.16 (7.6% reduction). On average, BranchNet improves the prediction accuracy on 19 static branches per benchmark, varying from 71 improved static branches in *leela* to no improved branches in *gcc*, *xalancbmk*, and *perlbench*.

There is a large variance in MPKI reduction among the ten benchmarks. In general, high-MPKI benchmarks tend to have hard-to-predict branches that are more suitable for BranchNet. In particular, the MPKI of benchmarks *leela*, *xz*, *mcf*, and *deepsjeng* are reduced significantly. On the other hand, the MPKI reduction on *omnetpp* is small since the main hard-to-predict branches in *omnetpp* are data-dependent branches, which BranchNet cannot improve. Even worse, there is almost no MPKI gain for *gcc*. *Gcc* contains many static branches that equally contribute to the total MPKI because of its large code footprint and many execution phases. Our current methodology cannot improve such benchmarks significantly. *Exchange2*, *x264*, *perlbench*, and *xalancbmk* do not have many hard-to-predict branches, so there is little opportunity for BranchNet.

To better understand the limitations of TAGE-SC, Figure 3.8 also shows the MPKI of MTAGE-SC without certain key components (GTAGE is the global history component of MTAGE). Most of the accuracy gap between TAGE-SC-L and MTAGE-SC is due to the larger size of the global history TAGE and the Statistical Corrector. This means that high-MPKI benchmarks exert high allocation pressure on the predictor tables, which is a sign that their global histories are indeed noisy. The local history components are also significant for a few benchmarks.

I also evaluated MTAGE-SC with an additional warm-up phase of 20 million instructions. The MPKI improvement due to warm-up is not significant.

3.3.5.4 Characteristics of Improved Branches

To better understand why BranchNet outperforms TAGE predictors, I have examined the source code of some of the most improved branches in *mcf* and *leela*. I describe my observations on the nature of these branches.

Most mispredicting branches of *mcf* appear in the *qsort* function. Branches in the comparison function are naturally hard to predict as they depend on data in an unsorted array. BranchNet does not improve these data-dependent branches. However, there are many branches in the body of *qsort* that depend on the results of these comparisons. TAGE does not learn these relationships because of the noisy nature of the history when running *qsort*. BranchNet, on the other hand, learns to ignore the noise.

Leela spends most of its execution time evaluating the properties of a Go board. The directions of most mispredicting branches are functions of these properties. In theory, many of these branches should be predictable because there are often other branches in the global history that depend on a shared property. However, there are also many uncorrelated branches, which make the history too noisy. Again, BranchNet circumvents the noisy history by only counting the correlated branches. Although the exact forms of the necessary prediction functions vary (e.g., the number of required filters, the nonlinear function, the minimum history length), they are conceptually similar to the example provided in Section 3.2.

Figure 3.9 shows the accuracy of the 16 most improved branches of *leela* and

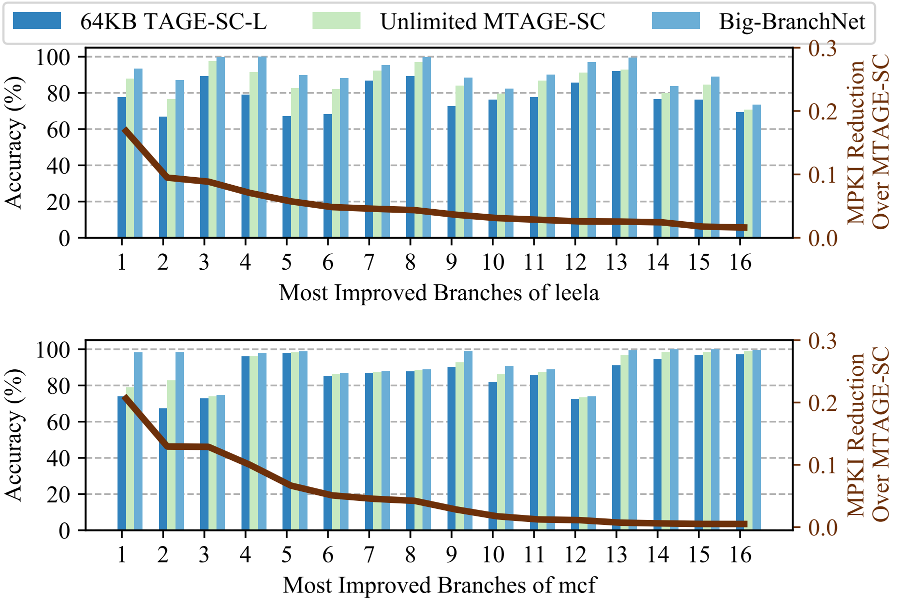


Figure 3.9: Accuracy of most improved branches using Big-BranchNet.

mcf compared to unlimited MTAGE-SC. The branches are sorted using MPKI reduction from left to right. In many cases, Big-BranchNet improves the prediction accuracy to almost 100%. For example, take the fourth branch in *leela* and the top two branches in *mcf*, BranchNet improves their accuracies from 79.1%, 73.9%, and 67.4% to 99.98%, 98.4%, and 98.6%. Even with its large storage budget, MTAGE-SC predicts the same branches with much lower accuracy (91.4%, 78.9%, and 82.6% respectively). Note that even if BranchNet cannot predict these branches 100% accurately, any improvement in accuracy results in high MPKI reduction because these branches are among the most frequently mispredicted branches.

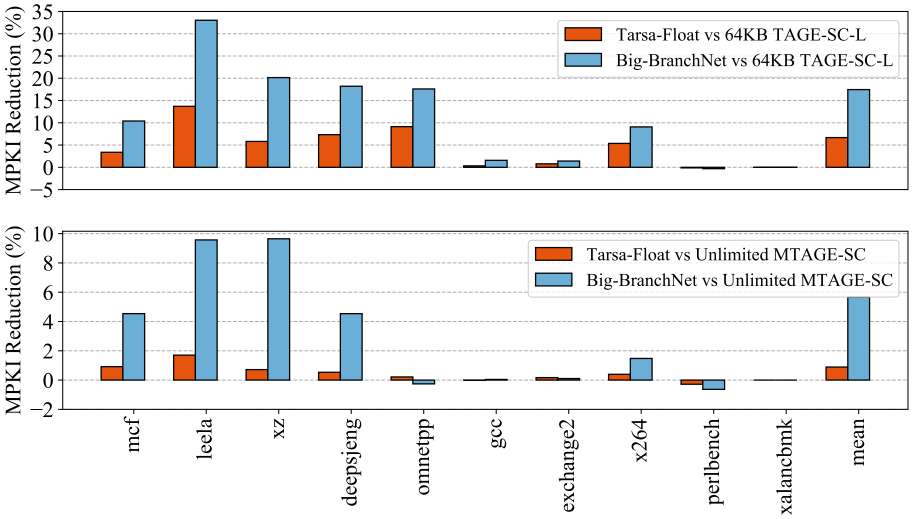


Figure 3.10: MPKI Reduction of Big-BranchNet and Tarsa-Float.

3.3.5.5 Comparison to Tarsa's CNN

Figure 3.10 shows the MPKI reduction of Big-BranchNet and Tarsa's CNN compared to 64KB TAGE-SC-L (the top figure) and Unlimited MTAGE-SC (the bottom figure). Both CNN models are impractical as inference engines. However, note that Big-BranchNet is much more accurate than Tarsa's CNN due to an architecture that is more tailored toward the needs of branch prediction. Also, note that the difference between Big-BranchNet and Tarsa-Float is more prominent when compared to an unlimited MTAGE-SC baseline. This result means that the improved accuracy due to Tarsa-Float can also be achieved by increasing TAGE's capacity, while Big-BranchNet's accuracy improvements cannot be replicated with increased TAGE capacity. Since Tarsa's CNN and BranchNet's prediction mechanisms are fundamentally the same, the higher accuracy of Big-BranchNet is only the consequence of different architectural choices. Big-BranchNet has a higher capacity to learn com-

Table 3.3: Sensitivity of BranchNet to the hidden fully-connected layers.

	MPKI Reduction				
Configuration	leela	mcf	xz	deepsjeng	average
Hidden neurons = {}	20.2%	5.7%	10.2%	13.8%	11.7%
Hidden neurons = {128}	31.1%	10.4%	18.0%	17.9%	18.7%
Hidden neurons = {32, 32}	32.1%	10.8%	19.5%	18.1%	19.5%
Hidden neurons = {128, 128} (Big-BranchNet)	33.3%	11.4%	20.1%	18.5%	20.2%

plicated prediction functions. The next section analyzes which design choices are the most important for optimizing the accuracy.

3.3.5.6 Comparison to Other Model Choices

Sensitivity to CNN architecture. This section reports the sensitivity of the MPKI reduction of Big-BranchNet to its architecture knobs. To save time and computing resources, I only ran the sensitivity studies on benchmarks *leela*, *mcf*, *xz*, and *deepjseng*, which are improved the most by BranchNet.

Table 3.3 shows that the hidden fully-connected layers are critical for achieving Big-BranchNet's high accuracy. Without any hidden layers, the MPKI reduction of the CNN model is 40% less than Big-BranchNet. The number of layers and the number of neurons per layer are also important factors for achieving high accuracy.

Table 3.4 shows that the slices with longer histories are critical for high accuracy. In fact, Big-BranchNet is not the most accurate configuration as its history is not as long as it could be. Furthermore, the last two configurations show that a single-slice design is not as accurate as a multi-slice design with geometric history lengths. Note that I increased the number of convolution filters to match the total number of filters across the five slices of Big-BranchNet. Geometric history lengths

Table 3.4: Sensitivity of BranchNet to the slice history lengths.

	MPKI Reduction				
Configuration	leela	mcf	xz	deepsjeng	average
Histories = {42,78,150}	29.1%	8.1%	11.5%	9.9%	15.0%
Histories = {42,78,150,294}	32.1%	10.4%	18.2%	13.9%	18.5%
Histories = {42,78,150,294,582} (Big-BranchNet)	33.3%	11.4%	20.1%	18.5%	20.2%
Histories = {42,78,150,294, 582, 870}	33.9%	11.6%	21.5%	20.0%	20.9%
Histories = {582}, 160 convolution filters Pooling width = 48	30.3%	8.4%	16.2%	17.8%	17.3%
Histories = {582}, 160 convolution filters Pooling width = 3	30.5%	10.3%	18.1%	17.8%	18.5%

allow the training algorithm to more easily distinguish correlated branches based on their coarse-grained distance to the noisy branch.

Table 3.5 shows that geometric pooling width proportional to the geometric history lengths is the correct design choice. The rationale is that the farther the correlated branch is, the more nondeterministic its position is in the history. Thus, a wider pooling width helps ignore the nondeterministic positions. But, BranchNet still benefits from smaller pooling widths for slices with shorter history lengths. Pooling widths proportional to the history lengths is a good middle ground that maximizes the accuracy.

Table 3.5: Sensitivity of BranchNet to the pooling width.

	MPKI Reduction				
Configuration	leela	mcf	xz	deepsjeng	average
Pooling Widths = {36,36,36,36,36}	32.4%	9.8%	17.0%	18.0%	18.6%
Pooling Widths = {3,3,3,3,3}	32.2%	11.1%	19.3%	17.9%	19.5%
Pooling Widths = {12,24,48,96,192}	33.2%	10.3%	20.0%	17.9%	19.7%
Pooling Widths = {1,2,4,8,16}	33.1%	11.3%	20.0%	18.4%	20.1%
Pooling Widths = {3,6,12,24,48} (Big-BranchNet)	33.3%	11.4%	20.1%	18.5%	20.2%

Table 3.6: Sensitivity of BranchNet to convolution width.

	MPKI Reduction				
Configuration	leela	mcf	xz	deepsjeng	average
Convolution width = 1	30.5%	10.5%	17.5%	17.6%	18.4%
Convolution width = 4	32.6%	11.1%	19.5%	18.2%	19.7%
Convolution width = 7 (Big-BranchNet)	33.3%	11.4%	20.1%	18.5%	20.2%
Convolution width = 10	33.5%	11.5%	20.9%	18.6%	20.5%

Table 3.7: Sensitivity of BranchNet to number of convolution filters.

	MPKI Reduction				
Configuration	leela	mcf	xz	deepsjeng	average
8 filters per slice	32.1%	11.0%	19.6%	18.1%	19.6%
16 filters per slice	32.7%	11.2%	20.0%	18.1%	19.9%
32 filters per slice (Big-BranchNet)	33.3%	11.4%	20.1%	18.5%	20.2%
64 filters per slice	33.7%	11.5%	20.1%	18.6%	20.4%

Table 3.6 shows the impact of the convolution width on BranchNet's accuracy. A wider convolution width allows the model to easier distinguish branch patterns when their order is relevant in a very small window. This result justifies the choice of wide convolution filters compared to Tarsa's CNN.

Table 3.7 shows the impact of the number of convolution filters on BranchNet's accuracy. While adding more convolution filters generally improves the accuracy, it has diminishing returns. Convolution filters are essential for detecting distinct branch patterns. The fact that BranchNet remains accurate with such a low number of convolution filters shows that the model uses only a few correlated branch patterns to make accurate predictions.

Table 3.8 shows that BranchNet is not sensitive to the embedding dimensions and an embedding table with 32 dimensions is sufficient for high accuracy.

Table 3.8: Sensitivity of BranchNet to embedding width.

	MPKI Reduction				
Configuration	leela	mcf	xz	deepsjeng	average
Embedding Width = 32 (Big-BranchNet)	33.3%	11.4%	20.1%	18.5%	20.2%
Embedding Width = 64	33.2%	11.2%	20.1%	18.5%	20.1%
Embedding Width = 128	33.2%	11.3%	20.1%	18.3%	20.1%

Big-BranchNet vs. Recurrent Neural Networks. Instead of CNNs, we can use Recurrent Neural Networks (RNNs) to learn branch prediction functions. Like CNNs, RNNs can isolate correlated branches and ignore any noise. RNNs are typically used for sequential tasks whether the order of individual items in a sequence of items conveys significant information (e.g., RNNs are commonly used for language translation, where the order of words in sentences conveys grammatical information). While it is intuitive to think that sequential processing is also useful for branch prediction, Figure 3.11 shows that Big-BranchNet is more accurate than LSTM and GRU models sized similarly to Big-BranchNet. I have observed that the fine-grain order of branches either does not matter or can be efficiently captured by wide convolution filters. Furthermore, CNN models are faster to train than RNN models. Finally, I see a clearer path for practical inference engines for CNNs compared to RNNs (discussed in Chapter 5.2.5.5). Thus, in this dissertation, I focus only on convolutional neural networks.

3.4 Coverage vs. Representativeness

Using offline training for branch prediction has traditionally been difficult because of the reliance of prior work on the representativeness of the profiled programs

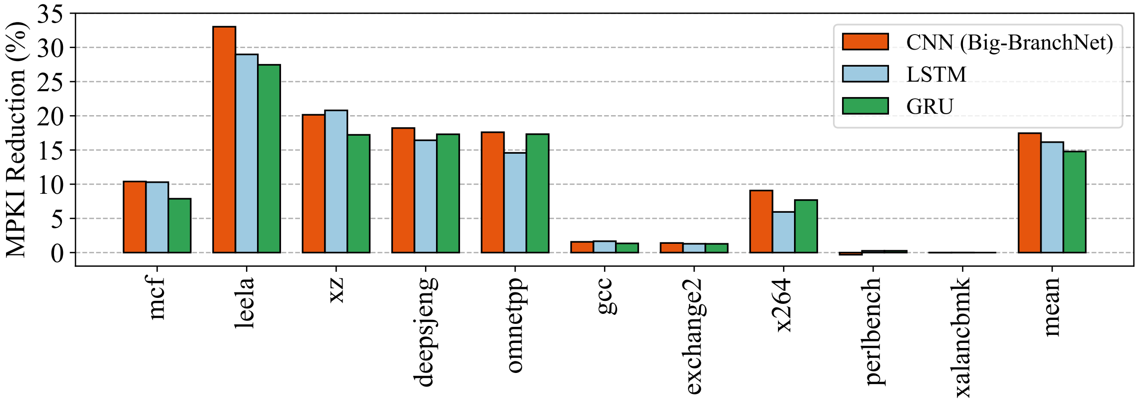


Figure 3.11: MPKI reduction of Big-BranchNet vs. Recurrent Neural Network models.

(Section 2.4). However, BranchNet is easier to adopt because it needs high coverage in the training set, not representativeness. While providing an extract metric for the coverage and the representativeness of a training set is not straightforward, I aim to showcase this distinction using two ways. One, I show the sensitivity of BranchNet to an imperfect metric of coverage. Two, I show the ineffectiveness of training TAGE-SC-L offline with high-coverage but not representative training sets to differentiate between the training mechanisms of CNNs and table-based predictors.

3.4.1 Sensitivity of BranchNet to Coverage

Ideally, a high coverage value should imply that there are enough examples in the training set to expose input-independent branch behavior to BranchNet. However, defining a precise metric with this definition is not straightforward. One option is to use branch coverage metrics commonly used for software testing, e.g., the fraction of the branches that are executed at least once, or the fraction of conditional branches that were both taken and not taken at least once. The problem with such metrics is that they do not say anything about the occurrence of longer branch patterns (either different static branches or different dynamic instances of the same branch). For example, suppose a loop in the training set is always executed for exactly one iteration. According to software testing coverage metrics, the training set has 100% coverage as the loop branch is not taken at least once and taken at least once. However, this may not be enough to expose input-independent branch behavior for cases when the loop is never executed or is executed for more than one iteration. On the other extreme, capturing all possible branch patterns to define coverage is also not feasible (exponentially high number of total possible patterns) and not useful, because the whole point of offline training for BranchNet is that the training set does not need to include exactly the same branch patterns as the test set. Nonetheless, I will attempt to show the sensitivity of BranchNet to training set coverage through imperfect proxies or metrics.

3.4.1.1 Sensitivity to the Training Set Size

An imperfect but convenient way to demonstrate the impact of coverage on prediction accuracy is to artificially limit the training to only a subset of available training examples. Figure 3.12 shows the MPKI reduction of BranchNet over unlimited MTAGE-SC using different training set sizes. Training with all the SimPoints of one program provides much better coverage of branch behavior compared to using only one SimPoint, which improves the generalization of the trained models. Similarly, using more than one input further improves the MPKI reduction.

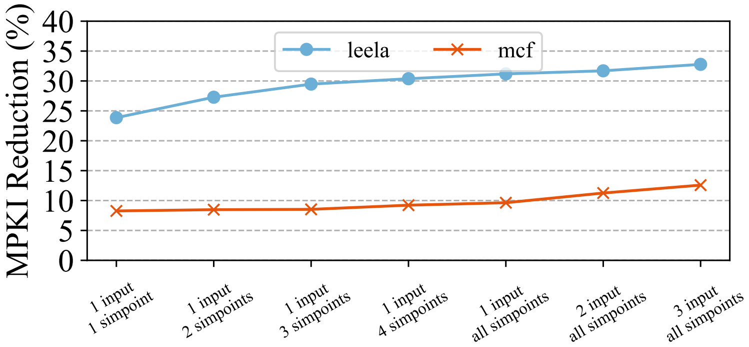


Figure 3.12: Sensitivity of Big-BranchNet to the training set size.

The first left-most data points corresponding to benchmark *leela* also make another strong case that BranchNet relies on Coverage for training. The first data point with the lowest accuracy uses the SimPoint with the largest weight (i.e., most representative region) of the training traces. But as more SimPoints are added to the training set, the accuracy of BranchNet improves. In fact, the biggest jump in MPKI reduction is when the least representative regions of the training trace (SimPoints with the smallest weights) are added to the trace. This sudden jump in accuracy shows that the key to learning input-independent behavior is exposing less frequently occurring patterns to the training set for better coverage.

Note that Alberta inputs for *mcf* are randomly generated and have no connections to the SPEC *ref* input. It is quite unlikely that all Alberta inputs for *mcf* are representative of the SPEC ref input. But BranchNet can infer persistent branch relationships from these randomly generated inputs to improve the accuracy of *mcf* on other inputs.

It is also worth noting the different sensitivity of *leela* and *mcf* to the training set size. While BranchNet's accuracy for *leela* is significantly increasing with using more SimPoints, BranchNet's accuracy for *mcf* is mainly sensitive to the number of inputs. This difference is because SimPoints of *leela* generally cover the same region of the code with different runtime behavior, but SimPoints of *mcf* either cover completely different code regions (which does not improve the coverage for many noisy branches) or cover regions with predictable branch behavior. Thus, better coverage of *mcf* is achieved only with more program inputs.

3.4.1.2 Quantifying Coverage

Here, I use an imperfect metric to quantify coverage to demonstrate the correlation between BranchNet's accuracy and the training set coverage. I define coverage in terms of branch patterns in the training set and the test set and the maximum history length of BranchNet. Since I use a separate BranchNet model for each noisy branch, I also use a per-branch coverage metric. Coverage is defined as follows:

$$Coverage = 1 - \frac{|S_{test} - S_{training}|}{|S_{test}|}$$

S_{test} is the set of all branch-direction pairs that appear in the history of the noisy branch in the test set (using the maximum history length of BranchNet). Similarly, $S_{training}$ is the set of all branch-direction pairs in the training set.

Using this definition, full coverage means that any branch-direction pair in the test set also appear at least once in the training set. No coverage means that none of the test set branch-direction pairs appear in the training set. This is an imperfect measure for coverage because it does not say anything about the order of branches in the history, the number of appearances of branches in each history instance, and the frequency of observing the branches across the dataset. Still, I use this metric as partial evidence for the relationship between coverage and prediction accuracy.

Figure 3.13 shows how BranchNet accuracy changes as the training set coverage changes. The training sets used to generate these data points are the same as the training sets used in Figure 3.12. The correlation between accuracy and coverage is clearer in the case of *leela*. For each branch, as coverage increases, the accuracy also tends to increase. Note that the correlation is not linear. In the case of *mcf*, this coverage metric is insufficient to explain the training quality. Many data points have equal coverage values despite having different training sets and different test set accuracies. Thus, it is hard to concluded any statistically significant observations using only a limited number of data points and an imperfect coverage metric.

Table 3.9 reports the Spearman Rank-order Correlation between the accuracy and coverage of the same 16 branches in *leela* and *mcf*. Spearman Rank-order measures the monotonicity of accuracy and coverage. The results convey the same in-

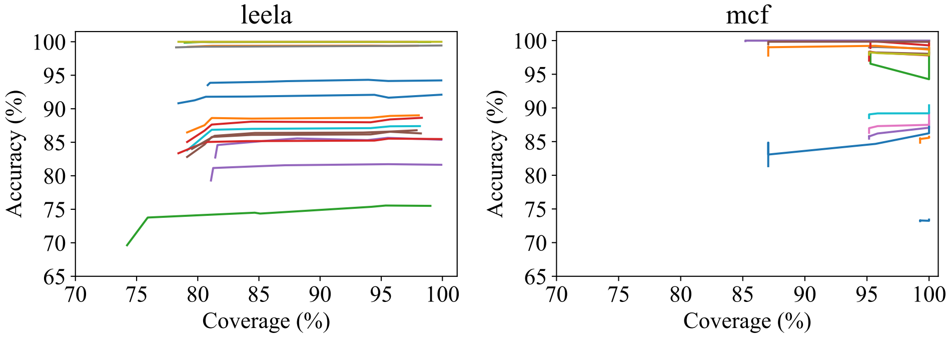


Figure 3.13: Accuracy vs. Coverage for top 16 improved branches using Big-BranchNet.

Table 3.9: Spearman Rank-Order Correlation between accuracy and coverage for top 16 most improved branches of *leela* and *mcf*.

	leela		mcf	
	Correlation	p-value	Correlation	p-value
Branch 1	0.79	0.04	0.29	0.53
Branch 2	0.89	0.01	0.87	0.01
Branch 3	0.57	0.18	-0.29	0.53
Branch 4	0.96	0.00	0.00	1.00
Branch 5	0.75	0.05	0.93	0.00
Branch 6	0.96	0.00	-0.39	0.39
Branch 7	0.75	0.05	0.93	0.00
Branch 8	0.96	0.00	-0.54	0.21
Branch 9	0.57	0.18	-0.15	0.74
Branch 10	1.0	0.0	0.93	0.00
Branch 11	0.89	0.01	0.85	0.02
Branch 12	0.96	0.00	0.15	0.74
Branch 13	0.93	0.00	0.30	0.52
Branch 14	0.96	0.00	0.58	0.17
Branch 15	0.89	0.01	0.93	0.00
Branch 16	1.0	0.0	0.15	0.74

formation as Figure 3.13, coverage and accuracy are strongly correlated in the case of *leela* with most branches having a small p-value. But in the case of *mcf*, the p-values are often big values, which means that we cannot reach any statistically significant conclusions.

The problem with using the simple coverage metric for *mcf* is that most of its noisy branches appear in *qsort*, which is a recursive function. The recursive nature means that branches that appear in the history all belong to the same function and it is likely to observe all the branches even in a small training set. Thus, most *mcf* branches have 100% coverage using only one training input. In contrast, *leela* branches never reach 100% coverage using the same metric. Thus, to better study the impact of coverage for *mcf*, we either need a more precise measure of coverage that

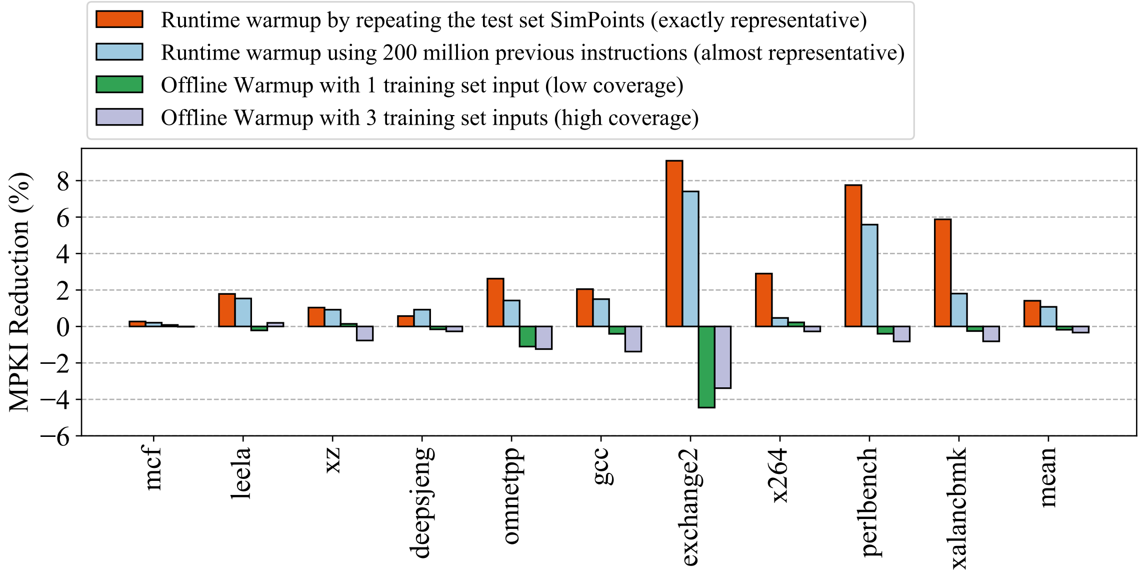


Figure 3.14: MPKI of TAGE-SC-L with additional warm-up.

takes the order and counts of branches into account, or reduce the experimental noise by using more data points. I leave a more detailed study on quantifying coverage for future work.

3.4.2 The Need for Representativeness to Train TAGE-SC-L

Unlike BranchNet, TAGE-SC-L predicts branches by learning predictions for specific branch history patterns. This means training the predictor on unrepresentative input sets is harmful even if the training set as a whole has high coverage. To demonstrate this distinction, Figure 3.14 shows the MPKI reduction of TAGE-SC-L with 4 different warm-up scenarios. First, I warm up TAGE-SC-L by repeating the test set SimPoints in the simulator. This method is unrealistic but is a good approximation of achievable speedup with representative training sets. Second, I warm up

TAGE-SC-L by using 200 million instructions before each test set SimPoints, simulating the realistic impact of warm-up on TAGE-SC-L. Finally, I warm up TAGE-SC-L on 1 or 3 inputs from the BranchNet training set. The results show that warming up on the training set almost always reduces the accuracy of TAGE-SC-L, which supports the assertion that TAGE-SC-L relies on exact representativeness. Note that, unlike BranchNet, training TAGE-SC-L with 3 inputs (more coverage) results in worse accuracy compared to training with 1 input.

In the above experiments, after the warm-up is finished, TAGE-SC-L is updated using its runtime training algorithm as usual. So the change in accuracy is mainly due to the initial state of TAGE-SC-L after the warm-up phase finishes. If the warm-up phase is representative, TAGE-SC-L does not need any further time in allocating and updating the relevant prediction counters, resulting in improved overall accuracy. If the warm-up phase is not representative, TAGE-SC-L needs time to first deallocate useless entries and allocate and update relevant entries, resulting in decreased overall accuracy. But in both cases, TAGE-SC-L quickly adapts to the test set SimPoints and becomes very accurate. Thus, the impact of warm-up strategies on overall accuracy is not dramatic.

To more clearly isolate the difference between warm-up strategies. I repeat the experiments but I stop updating TAGE-SC-L table entries after the warm-up phase is over. Since TAGE-SC-L does not get the opportunity to re-train according to the test set patterns on the fly, these experiments better demonstrate the difference in the representativeness of the training sets. Table 3.10 reports the increase in branch mispredictions. Unsurprisingly, not updating TAGE-SC-L after warm-up always results

Table 3.10: Increase in branch mispredictions when disabling TAGE-SC-L updates after warm-up, compared to a TAGE-SC-L that is updated normally without warm-up.

	mcf	leela	xz	deepsjeng	omnetpp	gcc	exchange2	x264	perlbench	xalancbmk	mean
Warmup with same Simpoints	1.4x	1.1x	1.4x	2.1x	9.9x	4.3x	4.6x	2.2x	4.2x	3.1x	2.1x
Warmup with previous region	1.7x	1.1x	1.3x	2.3x	11.7x	4.8x	5.7x	8.6x	3.3x	7.1x	2.6x
Warmup with 1 training set input	1.4x	1.4x	1.8x	3.8x	39.2x	14.4x	108.4x	3.6x	29.8x	140.1x	8.6x
Warmup with 3 training set inputs	1.7x	1.4x	3.0x	3.2x	49.5x	22.6x	115.9x	3.1x	37.7x	223.1x	11.2x

in more branch mispredictions compared to the baseline. But, warming up with the same input set is an order of magnitude more effective than warming up with different training set inputs. Again, unlike BranchNet, warming up with more training input sets results in less accuracy. This shows that the training of TAGE-SC-L relies on representativeness, in contrast to BranchNet, which relies on coverage.

Chapter 4

Explicitly Identifying Correlated Branches Using BranchNet

So far, this dissertation has described BranchNet as a branch predictor model to directly predict noisy branches. This chapter investigates a different way of using BranchNet models. Instead of using BranchNet as a black-box predictor, the trained BranchNet models are examined to explicitly identify correlated branches. This information can then be used to improve BranchNet by customizing its on-chip inference engines (Chapter 5) or improve traditional runtime branch predictors by filtering the global branch history (Chapter 6). This Chapter explains the process and the rationale of using BranchNet for identifying correlated branches.

First, I describe how examining the embedding table in a trained BranchNet model reveals which branches in the history are truly needed for accurate prediction (i.e., which branches are correlated). Second, I describe the modifications to the BranchNet architecture and training process to interpret the trained weights in the embedding table. Finally, I demonstrate the results of correlation extraction on SPEC 2017 benchmarks.

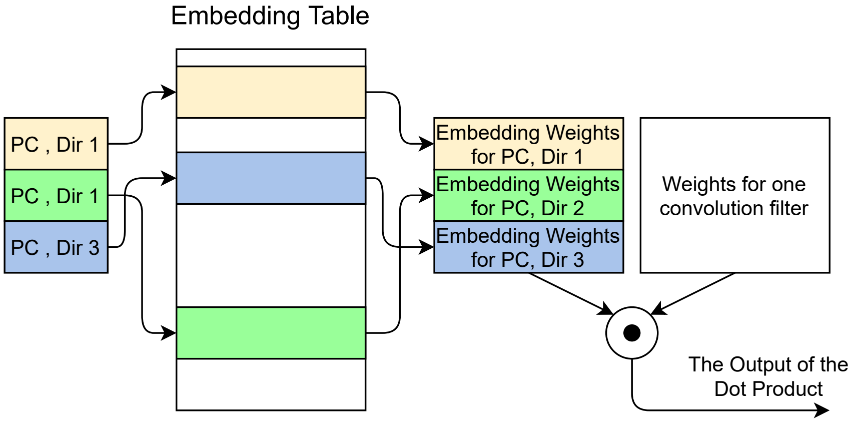

Embedding Table
PC , Dir 1
PC , Dir 1
PC , Dir 3
Embedding Weights for PC, Dir 1
Embedding Weights for PC, Dir 2
Embedding Weights for PC, Dir 3
Weights for one convolution filter
The Output of the Dot Product

Figure 4.1: Embedding and convolution in BranchNet.

4.1 BranchNet Architecture for Correlation Detection

Figure 4.1 shows the embedding lookup and the convolution operation for a single branch window in a BranchNet model. The embedding table is simply a collection of weights organized as a 2-dimensional array. Each row in the table holds the weights that represent the *value* of a given branch. Branches are identified by their program counters and directions. BranchNet uses the least significant 12 bits of PC, so the table contains $2^{12+1} = 2^{13}$ rows. The figure shows a convolution for a 3-wide branch window. The PC and direction of each branch are used as an index to lookup the embedding tables and the embedding weights for the three branches are gathered together. Then, the convolution output is computed by taking the dot product of the embedding weights and the convolution filter weights.

During training, the embedding and convolution weights are trained together to gradually detect the best branch patterns that can be used to predict the branch accurately. Once the model is fully trained, we can examine the embedding table to detect which branches are actually used by the CNN model. If the embedding weights corresponding to a branch are all zeros, the branch contributes nothing to the convolution outputs, i.e., the CNN model deems that branch useless for prediction accuracy. Thus, to know which branches are actually used by the CNN model, we can simply scan each row in the embedding table and identify which branches have non-zero embedding weights.

4.1.1 Regularization to Discourage Useless Embedding Weights

To prime a model for information extraction, we need to adjust the training algorithm such that the neural network is encouraged to find the information of interest. In the case of identifying correlated branches, it is important to incentivize the training algorithm to use only as many branches as needed to make a prediction, otherwise the neural network will use any small correlation that might exist between branches, regardless of their small contribution to the overall prediction accuracy. I use L1 regularization [23] to incentivize the network to use fewer branches. L1 Regularization is a technique that gradually decreases the magnitudes of network weights throughout the training process. As a result, only the weights that are truly needed for prediction accuracy can maintain non-zero values.

L1 regularization is implemented by adding a penalty term to the training *loss function*. The loss function can be any function that is a measure of how well the model is behaving. During training, the weights are adjusted step-by-step to minimize the loss function. By default, BranchNet uses the *Cross Entropy Loss* function that maximizes the prediction accuracy (i.e., the smaller the value of the loss, the more accurate the model is on the training set). This means that as long as a static branch is slightly correlated and can improve the overall accuracy, the embedding weights may be trained to have non-zero values. To avoid this, I add L1 regularization of the embeddings weights to the loss function of BranchNet:

$$Loss\ Function = CrossEntropyLoss + \lambda \sum |W_{embedding}|$$

The second component is the L1 regularization term which is the sum of the absolute values of all embedding weights, multiplied by coefficient λ. The regularization coefficient controls how much we value minimizing the embedding weights relative to the model accuracy. The best coefficient value is typically found empirically through trial and error.

4.1.2 Minor Modifications to the BranchNet Architecture

Removing branch aliasing: BranchNet uses the 12 least significant bits of branch PC along with the branch direction as the index into the embedding tables. This works fine for general prediction accuracy, but multiple branches may alias to the same embedding index. To avoid any potential aliasing, before starting the training process, I assign a unique index to each PC and direction pair in the validation set traces. Then, I transform the history to use these unique indices. After the training is done, I can then reliably associate each embedding row with a unique branch and direction pair. In the test set traces, I map any branch and direction pair that was not observed during training to index 0.

Modifications to BranchNet: Since the goal of this section is to demonstrate the ability of neural networks to detect correlation, I use a large configuration of BranchNet without optimizing for training time. I use the following architecture knobs: a single 600-branch history slice, 256 convolution channels, pooling width of 3, embedding width of 128, and three hidden 256-neuron fully-connected layers. The rest of the architecture is the original Big-BranchNet. In Chapter 6, I will revisit the model configuration to balance correlation detection and training time.

4.1.3 Shortcomings of L1 Regularization

While L1 regularization empirically works well in my evaluations, it is an indirect way to enforce a limit on the number of correlated branches. The regularization coefficient (λ) is an arbitrary number that dictates the aggressiveness of penalizing useless weights, i.e., the number of non-zero embedding rows after training with L1 regularization is unpredictable. To compensate for this unpredictability, we sort branches by their embedding values and pick the top branches. However, in theory, the most useful branches may not be among those branches with the highest embedding values. Embedding values are simply proxies that happen to work well in practice. Furthermore, the best value of λ can only be found empirically through trial and error, which may not be desirable in environments with constantly changing model requirements.

Perhaps a more robust approach is to use learning methods that more directly identify the most important branches. For example, data-driven pruning algorithms [27] may be more effective than using L1 regularization. Alternatively, interpretable

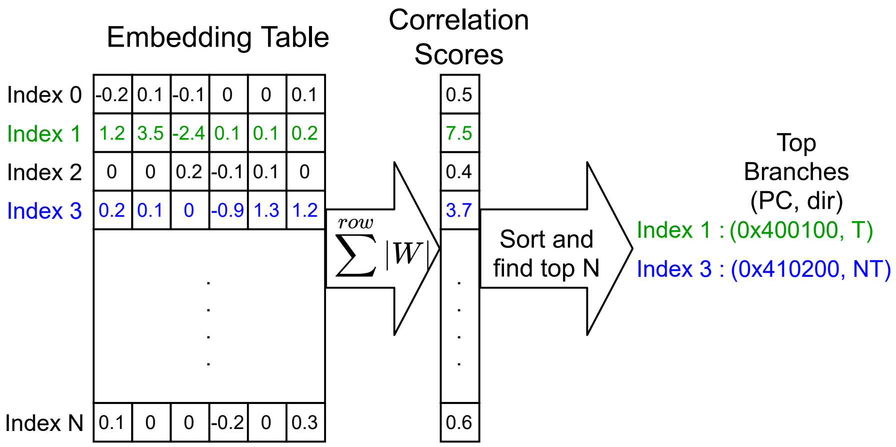


Figure 4.2: Example: interpreting embedding weights to identify the top correlated branches.

machine learning approaches [51] can be used, where either the model is designed with inherent interpretability, or model-agnostic techniques are used to understand the behavior of a model. Since I show that L1 regularization is effective at identifying good correlated branches, I leave the evaluation of alternative techniques for future work.

4.2 Putting Everything Together

Figure 4.2 shows how we examine the trained embedding weights to identify top correlated branches. We first compute the sum of the absolute values of the embedding weights in each row. We then use this sum as a score for the usefulness of each row in the embedding table. The higher the score, the more likely it is that the embedding row is contributing to the prediction outcome. We then sort the embedding indices by their scores and select the top embedding indices with

Algorithm 1 Identifying top N correlated branches for a predicting a given hard-to-predict branch.

```
Scan the validation set traces and assign a unique index to each (PC, dir)
for $\lambda$ in [5e-6, 1e-5, 2e-5] do
    Train a BranchNet model for each branch, L1-regularization coefficient = $\lambda$
    Compute the magnitude of embedding weights for each branch index
    Save the top N PC,dir pairs with the highest embedding magnitudes
    Force all other embedding weights to be zero
    Continue the training process with L1-regularization coefficient = 0
    Compute the prediction accuracy on the validation set
end for
Pick the top N PC,dir pairs for the training with the $\lambda$ that led to the highest
validation accuracy
```

the highest scores. Next, we look up the mapping that we generated during trace generation for converting each embedding index back to its corresponding unique branch. Now, we have the top correlated branches in the form of PC and direction pairs.

Algorithm 1 summarizes the step-by-step process to detect top correlated branches for predicting a given hard-to-predict branch. We first train a different model for each of the three values of λ as the L1 regularization coefficient.[1] Once the models are trained, we scan the embedding tables and identify the top branches with the highest embedding magnitudes (sum of absolute values of embedding weights). Since branches that are not selected as the top correlated branches may still have non-zero embedding weights, we force their embedding weights to be zeros for the rest of the training. Then, we fine-tune the model to adapt to the filtered out branches

[1] The choice of best values of λ is arbitrary, so we empirically found three candidate values for λ with the best results for a few branches.

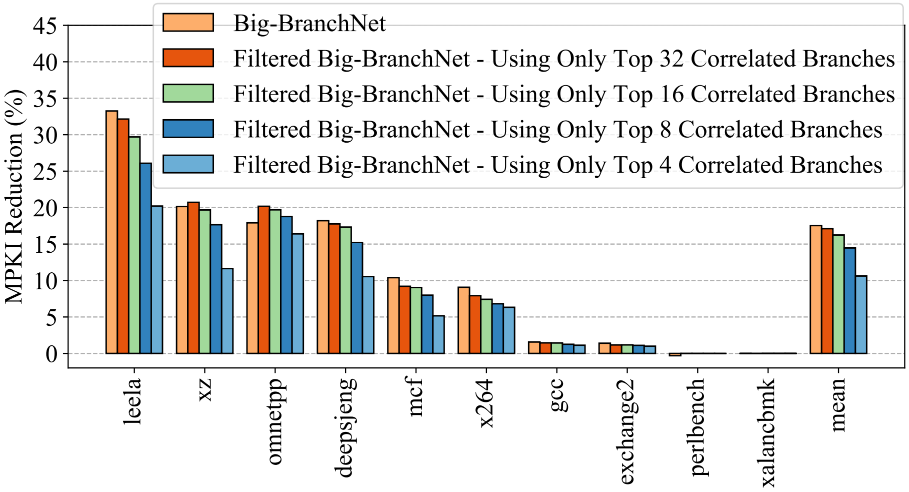


Figure 4.3: MPKI Reduction of BranchNet and Filtered BranchNet models compared to a 64KB TAGE-SC-L.

and maximize the accuracy without any regularization penalty. Finally, we record the validation accuracy after fine-tuning and select the top correlated branches from the training with a λ that resulted in the highest validation accuracy.

4.3 Methodology and Results

Offline training is done with the same methodology as Section 3.3.5. That is, I use Alberta Workloads [5] to generate branch traces used for offline training, SPEC train inputs for selecting the most improved hard-to-predict branches, and SPEC reference inputs to report the final results.

4.3.1 Impact of Using Only Top Correlated Branches on Accuracy

Figure 4.3 demonstrates the effectiveness of our approach in identifying the most correlated branches. The left-most bar is the MPKI reduction of BranchNet for

Table 4.1: Breakdown of prediction accuracy of CNN models for most improved hard-to-predict branches of *leela*.

	Big BranchNet	CNN Top 32	CNN Top 16	CNN Top 8	CNN Top 4	TAGE-SC-L
Br1	90.59%	86.72%	85.08%	82.59%	81.18%	77.92%
Br2	99.22%	98.76%	98.66%	98.14%	93.85%	89.21%
Br3	99.67%	99.70%	99.70%	99.69%	99.70%	76.06%
Br4	99.96%	99.96%	99.96%	99.95%	95.88%	75.65%
Br5	85.42%	80.11%	77.07%	72.85%	67.29%	66.85%
Br6	88.23%	86.56%	85.32%	82.93%	80.53%	67.40%
Br7	99.39%	99.22%	99.17%	99.06%	97.15%	89.21%
Br8	86.29%	83.76%	81.86%	80.01%	78.10%	68.33%

SPEC 2017 Integer benchmarks compared to a 64KB TAGE-SC-L. The other bars show the MPKI reduction of modified BranchNet models that are trained to use only the top 32, 16, 8, or 4 correlated static branches. The results show that many of the hard-to-predict branches that BranchNet improves can be accurately predicted using only 16 or 32 static branches. BranchNet's benefit drops when limited to only 8 or 4 static branches, but there is still room for improving prediction even when using only 4 highly correlated static branches.

Table 4.1 shows the prediction accuracy of the most improved hard-to-predict branches of benchmark *leela* using BranchNet, filtered BranchNet using top 32, 16, 8, or 4 branches, and 64KB TAGE-SC-L. In each row, an entry is colorcoded to highlight the minimum number of correlated branches that BranchNet needs to maintain a similar accuracy to big-BranchNet. Br3, Br4, and Br7, color-coded in green, can be predicted almost as accurately as BranchNet with only the top 4 or top 8 correlated branches. The rest of the branches, color-coded in red, generally benefit from having more than 8 correlated branches in the global history. Br1, Br5, Br6, and Br8 observe

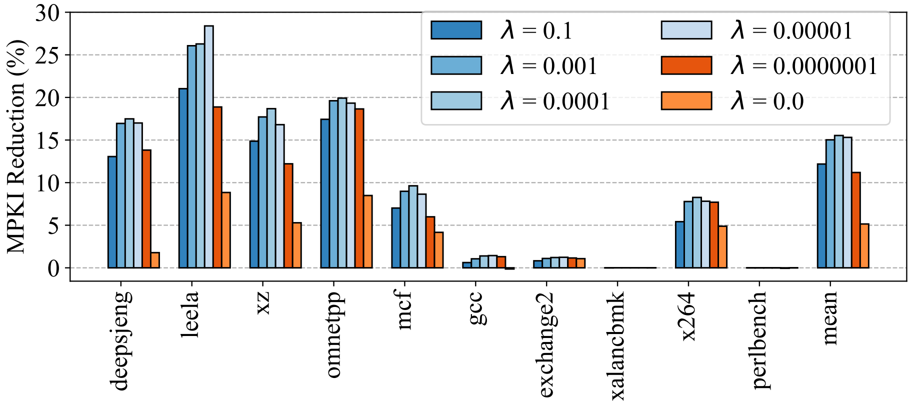


Figure 4.4: MPKI Reduction of Filtered BranchNet models using only top 16 correlated branches.

non-negligible drop in accuracy even when using top 32 correlated branches.

Figure 4.4 shows how L1 regularization impacts the effectiveness of identifying the most correlated branches. Note that regularization is only active during the first training phase, and the model is fine-tuned without regularization after all branches except the top 16 are filtered. Training without any regularization ($\lambda = 0$) results in the lowest MPKI reduction, meaning that without regularization, the embedding magnitudes are not a good metric for identifying the top correlated branches. A similar problem exists to a lesser extent if the coefficient is too small ($\lambda = 0.0000001$). If the coefficient is too big ($\lambda = 0.1$), the trained model is not as accurate as it can be, resulting in sub-optimal correlated branches.

4.3.2 Understanding the Identified Correlated Branches

4.3.2.1 Manual Source Code Cross-Examination

To verify whether the correct static branches are identified as correlated, I looked at the source code and the compiled binary of a few example branches. For two of the green branches in Table 4.1, I was able to manually identify 100% accurate prediction strategies based on the execution dataflow. Then, I verified that the static branches needed by the manually identified prediction functions are among the highly correlated branches identified by BranchNet. While two examples do not prove optimality, they provide evidence that the machine learning model can learn true branch correlations.

I also examined the source code corresponding to some red branches in Table 4.1 but was not able to manually identify any accurate prediction strategies. The correlated branches identified by BranchNet were also not necessarily in the dependence chains of the hard-to-predict branches. I hypothesize that the CNN model is combining signals from lots of slightly correlated branches to make a prediction, which improves the prediction accuracy compared to TAGE-SC-L but is not sufficient for 100% accuracy. Still, using a few somewhat correlated branches is better than using a lot of completely uncorrelated branches. Thus, our correlation extraction algorithm is still useful for improving branch prediction accuracy.

4.3.2.2 Correlated Branches vs. Dependent Branches

As explained in Section 2.3, Thomas et al. proposed using dependence chains to identify affector branches to filter the branch predictor's history. An affector branch

Table 4.2: Number of correlated branches and affectors. Br1-Br8 are the same as Br1-Br8 in Table 4.1. To identify correlated branches, we use the best configuration identified in Table 4.1.

	Affectors	Correlated Branches	Both
Br1	8	24	4
Br2	104	13	5
Br3	55	4	1
Br4	64	7	5
Br5	12	28	3
Br6	43	24	4
Br7	25	7	3
Br8	44	25	3

is a branch in the global history that guards any instruction in the dependence chain of the branch we want to predict, i.e., the execution of an affector branch changes the dataflow leading to the branch we want to predict. To compare the overlap of affectors with correlated branches, for each hard-to-predict branch, I collected the PCs of all affectors in a history length equivalent to BranchNet. I also collected the minimum number of identified correlated branches that led to a high prediction accuracy. Table 4.2 summarizes the results for the most improved branches of *leela*. The results show that although there is often some overlap between correlated branches and affectors, they are not the same, and affectors include many branches that are not useful for accurate branch prediction.

Chapter 5

Practical BranchNet Inference Engines

This chapter demonstrates specialized low-latency and storage-efficient inference engines for BranchNet models.[1] For some of the most hard-to-predict branches, BranchNet inferences engines are more accurate and more storage-efficient than TAGE-SC-L. A key contribution of this chapter is that achieving this high accuracy and storage-efficiency is only possible by specializing the model and the inference engines to the requirements of branch prediction.

I first present a few case studies of hard-to-predict noisy branches from SPEC 2017 Integer benchmarks. Using these case studies, I highlight the inefficiencies of Tarsa's CNN (which is a more straightforward adoption of CNNs for branch prediction) and showcase the room for improvement. Then, I present Mini-BranchNet, a variant of BranchNet models with practical and storage-efficient inference engines. I also propose an alternative inference engine, Counter-BranchNet, which uses correlated branch counters instead of convolution slices. Counter-BranchNet is slightly less accurate than Mini-BranchNet but is simpler to build. Combining Mini-BranchNet and Counter-BranchNet further improves the total MPKI reduction.

[1] The main contributions of this chapter have been previously published in papers that I co-authored [92, 93].

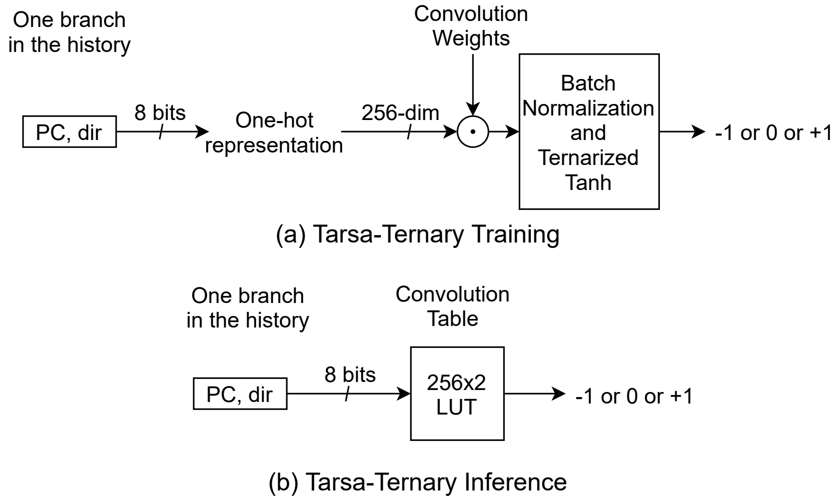


Figure 5.1: The convolutional layer of Tarsa-ternary.

5.1 Prior Work: Tarsa's Inference Engine

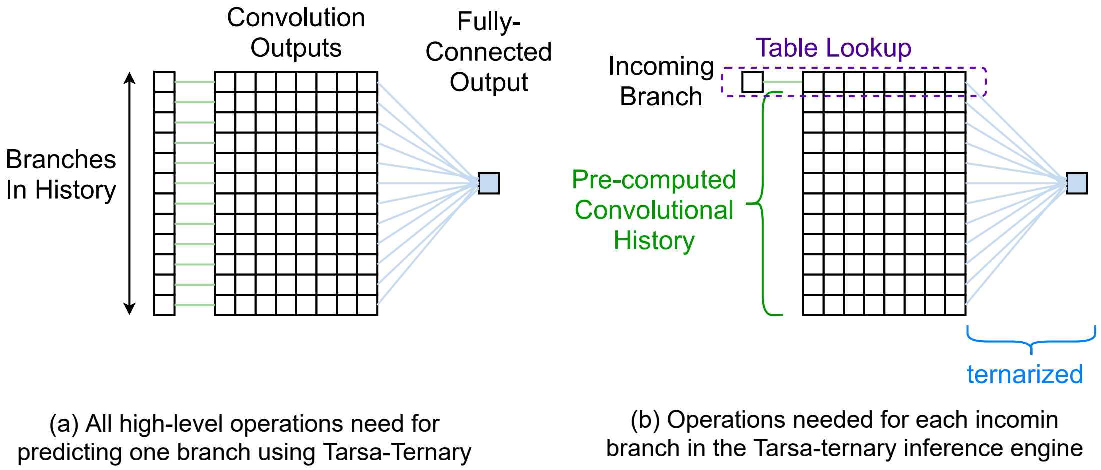
As the first people to use CNNs for branch prediction, Tarsa et al. [82] proposed an inference engine design for their CNN model. To reduce the size of the inference engine, they ternarized the model, i.e., all bits and intermediate outputs can have only three values: -1, 0, or +1. Hence, I refer to this ternarized version of Tarsa's CNN as Tarsa-ternary. Furthermore, Tarsa-ternary also uses a buffer to pre-compute all convolution outputs as branches are inserted into the history. This is possible because the output of the convolutional layer is independent of history positions. With these two techniques, Tarsa-ternary requires 5.125KB of storage per branch.

Convolution Table Lookups. A convolution operation on a single branch in the history involves a dot product operation. Figure 5.1 shows how Tarsa-ternary computes one convolution output. Tarsa-ternary eliminates all the arithmetic operations in two steps. During training, it adds a ternarizer to approximate the output of the activation function (normally a real number between -1 and +1) using three integers: -1, 0, or + 1 (Figure 5.1a). After training is done, for each possible branch index, the convolution output (one-hot representation + dot product + normalization + ternarized Tanh) is pre-computed, which results in either -1, 0 or +1. These ternary values are then stored in a small table that the Tarsa-ternary inference engine looks up to get the convolution output for each branch hash (Figure 5.1b). No arithmetic operation is needed at runtime, eliminating a 256-dimensional inner product per convolution operation, the normalization operation, and the activation function.

Pre-computing the Convolutional Outputs. Computing all 32 convolution channels across a 200-branch history requires 6400 convolution operations per branch prediction. Even when using table lookups, designing a high throughput inference engine under these requirements is too expensive. Thus, instead of doing all these operations at prediction time, Tarsa-ternary processes incoming branches one at a time and buffers their convolution outputs for future use. I call these buffers *Convolutional Histories.*

Figure 5.2 shows all optimizations needed to make Tarsa-ternary a small and somewhat practical (although still storage-inefficient) in one picture as explained in the two above paragraphs.

Figure 5.3 shows the block diagram of a Tarsa-ternary inference engine that can predict up to N static branches in a program. The history pipeline maintains the convolutional histories of all 32 Tarsa-ternary models. To make a prediction, the prediction pipeline simply selects the convolutional histories corresponding to the



(a) All high-level operations need for predicting one branch using Tarsa-Ternary

(b) Operations needed for each incomin branch in the Tarsa-ternary inference engine

Figure 5.2: All Tarsa-ternary inference optimizations.

next branch and computes only the fully-connected layer. If a branch does not reside in the inference engine, the baseline predictor, TAGE-SC-L is used to produce the final prediction.

Storage Cost. Table 5.1 shows the breakdown of storage needed to predict a single hard-to-predict branch using the Tarsa-ternary inference engine. Note how despite all the optimizations, Tarsa-ternary is still a very expensive model with 5.125KB total storage for predicting only one noisy branch.

Table 5.1: Breakdown of Tarsa-Ternary inference engine storage requirements for one static branch.

	Using Architecture Knobs	Total Storage
Convolution Tables	$2C_0(2^{h+1})$	2 KB
Convolutional History Buffers	$2C_0H_0$	1.5625 KB
Fully-connected weights	$2C_0H_0$	1.5625 KB

Latency. The convolutional layer has the least impact on latency since all arithmetic is replaced with a table lookup and all operations along the long global

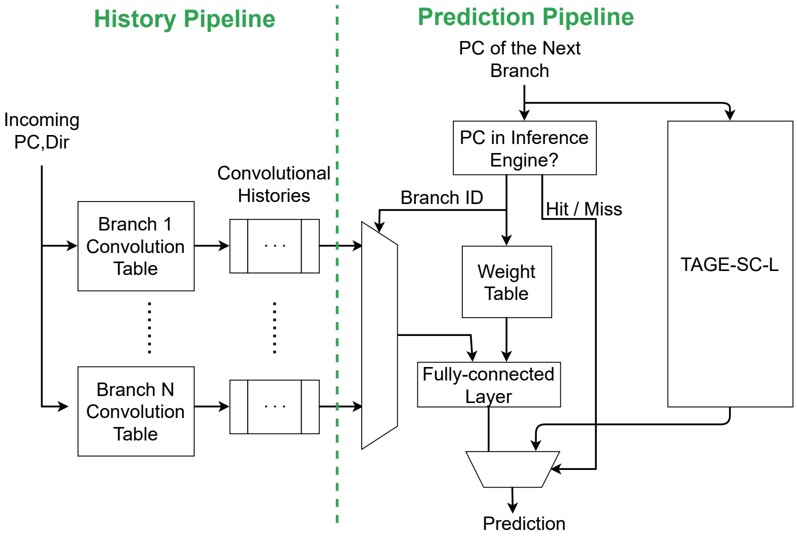


Figure 5.3: Overview of Tarsa-ternary inference engine (N = maximum number of noisy branches that fit within the engine).

branch history are pre-computed. On the other hand, the fully-connected layer needs to compute a dot product of vectors of size 6400 (32 channels of 200 elements each). Even after ternarization, this remains a high-latency operation. Thus, A good direction for reducing the latency of the inference engine is to reduce the size of the inputs to the fully-connected layer.

5.2 Case Studies of CNN Inference Engine Inefficiencies

This section uses a few case studies to delve into the inefficiencies of Tarsa-Ternary. The example branches are all taken from SPEC 2017 Integer benchmarks (*mcf* and *leela*). I use the most representative SimPoint of the benchmarks with SPEC ref inputs to report the prediction accuracy. For each example, I present a custom predictor that predicts the branch almost perfectly. Then, I show why a CNN

```
void qsort(void* a, size_t n, ...) {
    if (n < 7) {
        insertion_sort(a, n, ...);
        return;
    }

    // start partitioning
    for (int i = 0 to n) {
        if (a[i] < pivot) {
            ... // insert a[i] in the left partition
        }
        if (a[i] > pivot) {
            ... // insert a[i] in the right partition
        }
    }
    // finished partitioning

    if (size of the left partition > 1) {
        qsort(...) // recurse on the left partition
    }
    if (size of the right partition > 1) {
        qsort(...) // recurse on the right partition
    }
}
```

Figure 5.4: Simplified pseudo-code of *qsort*.

model can predict the branch much more accurately than TAGE. Finally, I identify inefficiencies in Tarsa-Ternary by contrasting it to the targeted custom predictor, which predicts with higher accuracy, lower latency, and better storage-efficiency.

5.2.1 Case Study 1: Qsort — Correlation with a Branch Count

Figure 5.4 shows a simplified pseudo-code for SPEC's implementation of *qsort* with two noisy branches highlighted in yellow. *Qsort* is a C library function for in-place sorting, typically implemented using the Quicksort algorithm. The Figure does not show all of the implementation details required for high performance but captures the branch behavior of the two example branches. I chose *qsort* for the first two case studies because it is hot code in *mcf*, and because it contains several branches that have sophisticated correlations with the branch history in the presence of noise. Both highlighted noisy branches can theoretically be predicted 100% accurately; however, both are predicted poorly by TAGE-SC-L. This case study focuses on the noisy branch highlighted in line 18.

After the partitioning phase of *qsort* is done (lines 7-16), the algorithm decides whether to recursively call *qsort* on each partition (lines 19 and 22). In this case study, we focus on the branch in line 18 of Figure 5.4. Assume that for each if-statement, the taken direction of the guard branch skips the body. TAGE-SC-L predicts this branch with 94.7% accuracy, which is only slightly better than the static bias of the branch (92.7% not-taken). However, we know the branch will be not-taken (i.e., *qsort* will be recursively called) only if the partition has at least two items in it. Therefore, the size of the partition can be used to predict the branch with 100% accuracy — predict not-taken only if the partitioning phase has inserted at least two items in the left partition (i.e., line 10 was executed at least 2 times). We can use the branch history to determine the number of times line 10 was executed. If the count reaches two, then we know there are at least two elements in the left partition and the branch at line 18 should be predicted not-taken (or taken if the count is less than two). Note, there are two caveats to this approach. First, the predictor must know when the partitioning phase begins to initialize the count to zero. Second, once compiled, the resulting assembly code for Figure 5.4 contains 3 branches (instead of just one) that

```
int left_partition_size;

void update(next_execution_line) {
   switch(next_execution_line) {
      case qsort::line7: left_partition_size = 0;
      case qsort::line10: left_partition_size += 1;
   }
}

bool pred() {
   return (left_partition_size < 2);
}
```

Figure 5.5: Perfect custom predictor for the if-statement in line 18 of Figure 5.4.

guard the execution of line 10. Therefore, the branch predictor should isolate more than one branch to count the number of elements inserted into the left partition.

Figure 5.5 defines the update and prediction algorithm for a targeted branch predictor that implements a perfect prediction strategy for this case study. It consists of three components: the predictor state (left_partition_size), an update algorithm that updates the state every time a branch is fetched, and a prediction function. To implement this algorithm in hardware, we require a 2-bit saturating counter for left_partition_size, a single register to track the PC of the branch leading to qsort::line7, and 3 registers used to track the PCs of branches that guard qsort::line10, amounting to 198 bits of storage.

Even though this simple prediction algorithm exists, TAGE cannot predict this branch accurately because it cannot distinguish the correlated branches in the history (line 9) from the uncorrelated ones (all other branches). As a result, the pollution in the branch history creates too many patterns for TAGE to memorize, resulting in low accuracy.

A CNN, however, uses a convolution layer that acts as a filter for identifying correlated branches and removes uncorrelated branches. Once the uncorrelated branches have been removed, the fully-connected layers can easily check the size of the partition by counting the not-taken occurrences of branches corresponding to line 9. Both the ternary and the floating-point version of Tarsa's CNN predict this branch with high accuracy (99.35% and 99.7% respectively). Note, however, that we only want to calculate the size of the most recent partition in the history. Our targeted solution handled this by resetting the counters before the partitioning began. Tarsa's CNN, however, must learn on its own which regions of the history register are important. This leads to inefficiency in Tarsa-Ternary compared to my custom predictor.

Unfortunately, the high prediction accuracy of Tarsa's CNN comes at a high storage cost. Table 5.2 compares the accuracy and storage of TAGE, Tarsa's CNN, Big-BranchNet, and custom logic. The Custom Predictor is what was defined in Figure 5.5. I do not quantify the storage cost of TAGE per branch because the per-branch storage usage is dynamic and depends on the allocation pressure from other

Table 5.2: CNN case study 1: accuracy and storage of predictors.

	TAGE	Tarsa	Tarsa	Big-BranchNet	Custom
		Ternary	Float	Float	Logic
Accuracy	94.7%	99.35%	99.7%	~100.0%	100%
Size	N/A	5.1 KB	82 KB	17.7 MB	198 bits

branches. Even though Tarsa's CNN is much more accurate than TAGE, it is far from perfect and requires unnecessarily large storage.

5.2.1.1 Why Does Tarsa's CNN Not Reach 100% Accuracy?

Identifying the most immediate partitioning phase in the global branch history is a highly non-linear task. Thus, Tarsa's CNN with only one fully-connected layer cannot ever learn to predict this branch optimally. The CNN compensates for this inability by learning to use any correlated branch in the history to improve its accuracy. For example, if the sizes of left and right partitions are correlated in the training set, the CNN will use the right partition size for prediction. In general, this overfitting may lower prediction accuracy at runtime, but in this case, such a data-driven approach is sufficient for $> 99\%$ accuracy.

Another source of inaccuracy may be the existence of data-dependent correlated branches. For example, *qsort* chooses its partitioning pivot based on heuristics to increase the likelihood of balanced partitions. Thus, it may be that the training algorithm learns to also use the size of the right partition as a signal for predicting the size of the left partition. Such high-quality but occasionally unreliable signals in the history may lead the training algorithm to overfit to a non-optimal solution.

Finally, a disadvantage of a CNN compared to the optimal algorithm is the relatively small history length. We measured the minimum history length to identify that at least two items have been inserted into the left partition in our test set. The median distance is 30 branches, the 99th percentile is 156, and the maximum distance is 710. Since the history length of Tarsa's CNN is 200 branches, it cannot accurately determine the correct size of the partitions. In this case, however, a 200-branch history happens to be enough for determining whether the partition has more than one element or not, which is all that is needed for correct prediction. In general, sometimes a very long history is needed to capture all required correlated branches to maximize prediction accuracy.

5.2.1.2 Sources of Storage-Inefficiency

As described earlier, Tarsa's CNN model is incapable of learning the optimal prediction algorithm and instead relies on using any correlated branches in the history. Thus, it needs many convolution filters to identify all useful correlated branches. Moreover, some degree of over-parameterization is necessary for convergence when training multi-layer neural networks [4], which by definition implies sub-optimal predictor size. This factor can be somewhat alleviated with post-training network pruning.

Another source of storage-inefficiency is the fully-connected layer, which is a much more general function than needed. Note that the inference engine not only needs to store all the fully-connected weights but also should buffer the convolution outputs that feed the fully-connected layer. In our custom design, this was replaced by incrementing a 2-bit saturating counter at the appropriate times.

5.2.2 Case Study 2: Qsort — Correlation with a Specfic Segment in the History

A common technique for speeding up quick-sort is to switch sorting algorithms once a partition is smaller than some threshold. Line 2 in Figure 5.4 is the if-statement that controls this switch. This branch is very hard-to-predict for TAGE with 66.4% accuracy (the static bias of the branch is 56.2% taken).

At the first glance, this branch may seem as predictable as the branch in case study 1. For all recursive calls to *qsort*, the value of *n* is produced by the partitioning phase in the caller instance of *qsort*. Thus, similar to case study 1, a branch predictor can determine *n* by tracking insertions into the left and right partitions. However, this prediction task is much more difficult than the prediction task in case study 1 because the caller instance of *qsort* may appear arbitrarily deep into the branch history. A single fully-connected layer, especially if ternarized, is too simple to learn this behavior. As a result, the accuracy of the ternary and the floating-point version of Tarsa's CNN are 82.4% and 88.4% respectively.

The poor accuracy of Tarsa's CNN does not mean that CNNs cannot predict this branch accurately. Big-BranchNet predicts this branch with 98.2% accuracy, albeit at the cost of a 17.7MB model. Of course, such a large prediction model is not helpful at runtime. The more interesting question is whether a CNN can be accurate and cost-efficient at the same time (recall the goal of this Chapter is to design cost-efficient BranchNet inference engines).

Figure 5.6 shows an accurate and cost-efficient custom algorithm for predicting this branch by observing the incoming branch stream. First, similar to Figure 5.5, it tracks insertions into the left and right partitions using saturating counters (lines 7-12). After the partitioning, the algorithm produces a prediction for the if-statement in qsort::line2 of the right partition and pushes the prediction in a prediction stack (lines 16-19). Furthermore, before a recursive call on either the left or right partition,

```
int left_partition_size, right_partition_size;
bool left_recursion;
stack<int> right_partition_predictions;

void update(next_execution_line) {
   switch(next_execution_line) {
      // Determining partition sizes
      case qsort::line7:
         left_partition_size = 0;
         right_partition_size = 0;
      case qsort::line10: left_partition_size += 1;
      case qsort::line13: right_partition_size += 1;

      // Push the prediction for the right
      // partition in a stack
      case qsort::line16:
         if right_partition_size > 1:
             pred = (right_partition_size > 6);
             right_partition_predictions.push(pred);

      // Update left_recursion before each call
      case: qsort::line19: left_recursion = true;
      case qsort::line22: left_recursion = false;
   }
}

bool pred() {
   if left_recursion:
      return (left_partition_size > 6);
   else:
      return right_stack.pop();
}
```

Figure 5.6: Perfect custom predictor for the if-statement in line 2 of Figure 5.4.

it sets the flag *left_recursion* accordingly (lines 22-23). Finally, to make a prediction, if the *left_recursion* flag is set, it simply uses the size of the left partition to make a prediction, otherwise, it pops a prediction off the stack. This algorithm fails when predicting the root of the recursion tree, but is otherwise completely accurate. To implement this algorithm in hardware, the predictor needs 11 registers to track the PC-direction pairs, two 3-bit counters for determining the left and right partition sizes, and a 64-entry stack (1-bit per entry) used to hold the predictions for the right partition, amounting to a total of 609 bits.

Similar to the previous case study, representing this optimal algorithm by a CNN is unrealistic because of insufficient history length. This case study is even more difficult because if *qsort* is entered because of a recursive call on the right partition, the global branch history is polluted with an unknown number of partitioning branches because of the earlier recursion on the left partition. For predicting 90% of the instances of this branch in *qsort*, the optimal algorithm discards up to 641 youngest branches in the global history. Table 5.3 summarizes the results and shows the gap between Tarsa-ternary and bigger CNNs and the custom predictor. The inability of Big-BranchNet to reach the accuracy of the optimal predictors shows that CNNs may not be suitable to learn very complicated relationships. It is more likely that

Table 5.3: CNN case study 2: accuracy and storage of predictors.

	TAGE	Tarsa Ternary	Tarsa Float	Big-BranchNet Float	Custom Logic
Accuracy	66.4%	82.4%	88.4%	98.2%	100%
Size	N/A	5.1 KB	82 KB	17.7 MB	609 bits

```
void add_global_captures(int& x, ...) {
    while(...)
        if (...)
            x += 1;
}
void save_critical_neighbours(int& x, ...) {
    while(...)
        if (...)
            x += 1;
}
void add_pattern_moves(int& x, ...) {
    while(...)
        if (...)
            x += 1;
}

void play_random_move(...) {
    int x = 0;
    add_global_captures(&x, ... );
    save_critical_neighbours(&x, ... );
    add_pattern_moves(&x, ... );

    ...

    for (int i = 0 to x) {...}
}
```

Figure 5.7: Simplified pseudo-code of a noisy branch in *leela* (case study 3).

Big-BranchNet is picking correlated but not perfect signals from the near history, as opposed to the optimal strategy of identifying the partitioning phase of the parent node in the recursion tree.

5.2.3 Case Study 3: Leela — Correlation with Branch Counts Again

This case study covers the same hard-to-predict branch that was the inspiration for the example used in Section 3.2. The pseudo-code of the branch and its correlated branches is shown in Figure 5.7. The noisy branch (highlighted in yellow) is the exit branch of a loop that iterates for a variable number of iterations. The number of loop iterations depends on a variable that starts with 0 at the start of the function and is occasionally incremented by three functions: *add_global_captures*, *save_critical_neighbours*, and *add_pattern_moves*. At a high level, each function consists of a loop that sometimes increments the variable of interest. Thus, in theory, the branch could be predicted by counting the loop branch itself and the branches corresponding to the if statements in three functions. Thus, a perfect predictor consists of two counters that are reset at the beginning of *play_random_move()*, with one counter incremented with each iteration of the for loop (line 25), and one counter incremented every time x is incremented (lines 4, 9, and 14). The predictor can then predict the noisy branch 100% accurately by comparing the two counters with the total storage cost of 2030 bits (the high-level if statements result in 29 branches after all compiler optimizations are applied, which increases the total storage cost). Table 5.4 summarizes the results. A CNN could learn to mimic a similar prediction strategy and achieve high accuracy (100% in the case of Big-BranchNet, and 98.4% in the case of Tarsa-Ternary). The small inaccuracy of Tarsa-Ternary is probably due to difficulties in identifying the most relevant region in the history. The fact that there are so many correlated branches that need to be tracked together also make the

Table 5.4: CNN case study 3: accuracy and storage of predictors.

	TAGE	Tarsa Ternary	Tarsa Float	Big-BranchNet Float	Custom Logic
Accuracy	77.5%	98.4%	99.6%	∼100.0%	100%
Size	N/A	5.1 KB	82 KB	17.7 MB	2030 bits

training process more difficult.

5.2.4 Case Study 4: Leela — Inherently Complicated Branch Relations

Figure 5.8 highlights one of the most mispredicting branches of Benchmark *leela*. 64KB TAGE-SC-L predicts this branch 89.4% accurately (the branch bias is 80.2% not taken). Among the case study noisy branches, I found it the most difficult to design a custom predictor for this branch. The problem is that the branch dependence chain is both complicated and insufficient for perfect prediction accuracy. Still, I developed a heuristic that predicts this branch almost as accurately as Big-BranchNet, which I will use as a reference for a cost-efficient predictor.

The function *play_random_move* is responsible for deciding the next move in an AI for the game of Go. First, using functions *add_global_captures*, *save_critical_neighbours*, and *add_pattern_moves*, it creates a list of candidate next moves. Then the function iterates over all candidate moves to assign a score that helps the AI decide which move to take. As part of the score, the AI needs to know whether a candidate move is a neighbor to the last move that was taken on the board (The last move's position is an input argument of *play_random_move*). So, the function compares each candidate move against all the neighbors of the last move. This comparison is hard to predict for TAGE.

To design a custom predictor, I examined the source code of the three inner functions and made an observation. The candidate moves inserted by *add_global_captures* and *save_critical_neighbours* are most likely (but not definitely) not the neighbors of the last move. On the other hand, the candidates moves inserted

```
void add_global_captures(int& x, ...) {
    while(...)
        if (...)
            candidate_move_list.append(...)
}
void save_critical_neighbours(int& x, ...) {
    while(...)
        if (...)
            candidate_move_list.append(...)
}
void add_pattern_moves(int& x, last_move) {
    for (move in neighbors(last_move))
        if (...)
            candidate_move_list.append(move)
}

void play_random_move(last_move, ...) {
    candidate_move_list = []
    add_global_captures(candidate_move_list, ...);
    save_critical_neighbours(candidate_move_list, ...);
    add_pattern_moves(candidate_move_list, last_move, ...);

    for (candidate_move in candidate_move_list) {
        for (last_move_neighbor in neighbors(last_move)) {
            if (candidate_move == last_move_neighbor) break;
            ...
        }
    }
}
```

Figure 5.8: Simplified pseudo-code of a noisy branch in *leela* (case study 4).

by *add_pattern_moves* are by definition neighbors of the last move. Thus, a simple prediction heuristic is to identify which inner function was responsible for inserting the candidate move that the noisy branch is currently examining. If the candidate move was inserted by *add_global_captures* and *save_critical_neighbours*, the heuristic predicts that the noisy branch is never taken (not a neighbor). If the candidate move was inserted by *add_pattern_moves*, the heuristic predicts that the noisy branch is taken only if the current iteration count of the for loop in line 24 is the same as the iteration count of the for loop in line 12 at the time that the candidate move was inserted. This custom prediction strategy achieves a 98.8% accuracy with 1082 bits of storage.

While the custom predictor is smaller than the custom predictor for Case Study 3, the logic needed to implement this predictor is much more complicated, thus, a more complicated CNN model is needed to learn a similar prediction function. Associating the iteration counts of the loops in lines 12 and 24 is a much more complicated (and nonlinear) task than counting correlated branches. My custom predictor uses a dynamic vector that maintains the loop counts of the candidate moves inserted by *add_pattern_moves*. This highly dynamic and control-flow-dependent behavior is hard to represent with fully-connected neural networks (the model needs to have many

Table 5.5: CNN case study 4: accuracy and storage of predictors.

	TAGE	Tarsa Ternary	Tarsa Float	Big-BranchNet Float	Custom Logic
Accuracy	89.4%	85.8%	91.8%	99.3%	98.8%
Size	N/A	5.1 KB	82 KB	17.7 MB	1082 bits

layers and many neurons). Thus, a simple CNN like Tarsa-Ternary is not capable of predicting this noisy branch at all. Table 5.5 summarizes the results. Tarsa-ternary is less accurate than TAGE-SC-L because the required prediction function is too complicated to represent in the ternarized version of Tarsa's CNN. Tarsa-Float is more accurate than TAGE-SC-L, but it still is nowhere close to the optimal accuracy. I believe the main limiting factor is that Tarsa's CNN does not have any hidden fully-connected layers, which are needed for learning non-linear branch correlations. However, not only Big-BranchNet can predict this branch accurately, it is even more accurate than the custom-designed predictor. I speculate that Big-BranchNet picks up on some signals that sometimes identify whether the candidate moves of *add_global_captures* and *save_critical_neighbours* are neighbors.

5.2.5 Lessons From the Case Studies

5.2.5.1 Long History Lengths

Case study 2 showed that some branches benefit from longer history lengths, but naively increasing the history length is impractical because of inference hardware constraints. BranchNet uses a longer history length than Tarsa by using geometric history lengths and an aggregation function (sum-pooling) to cheaply cover longer histories.

5.2.5.2 Specialized Structures

In both case studies, CNNs are worse than custom predictors in terms of both accuracy and storage-efficiency. This is because the CNN must learn functions that the custom logic was directly programmed to perform. On the other hand, the custom predictors are not general enough to deploy in an actual branch predictor. What we need is a predictor that is general enough to learn to predict branches in new algorithms, but contains enough custom logic that it does not need to re-learn functions that are common among many branches. If we implement custom structures targeted toward common operations, we can bridge the gap between a trainable predictor and our custom solutions. A key challenge will be finding ways to plug the custom logic into the network during training. Network designers may need to use functions that work with backpropagation, or use regularization, pruning, and/or post-training transformations to steer neurons toward the targeted custom hardware. For example, since branch counting was a common aggregation strategy for information in the global history, BranchNet uses sum-pooling layers. Mini-BranchNet inference engines also support sliding sum-pooling (Section 5.3), which is an approximation strategy to reduce the storage cost of the predictor. I also propose an alternative to Mini-BranchNet which further specializes BranchNet by replacing convolutions with branch counters (Section 5.4).

5.2.5.3 Hardware-Aware Training Algorithms

As discussed throughout this paper, on-chip branch predictors have tight latency and storage constraints that must be obeyed. This makes quantization, pruning, and regularization very important. Tarsa et al. use the training algorithm of Courbariau et al. [17] to ternarize their CNN models. However, unlike the binarized neural networks studied by Courbariau et al., branch prediction accuracy significantly drops with quantization. This is partially because Tarsa's CNN is many orders of magnitude smaller than the CNN models that are evaluated in prior quantization work, increasing the likelihood of converging to bad local optimum solutions [4]. A more effective training strategy is to initially over-provision the network, then gradually regularize and prune the network to meet the hardware constraints. Such approaches are well studied in prior work [85, 14, 86], albeit on larger models and more flexible inference engines. Hardware-aware training algorithms would allow us to use an over-parameterized network to assist with training, while still fitting in the predefined hardware budget for inference. For example, let us revisit case study 1. In theory, a 1-filter CNN can predict the branch almost perfectly; however, training a 1-filter CNN results in only 93.0% accuracy, with a 2-filter CNN reaching 97.5%, and a 3-filter CNN reaching 99.7%. Now, if we over-parameterize the CNN with 4 filters, then use regularization to penalize redundant filters and prune the unused filters, we can achieve 99.7% accuracy with only 2 filters. Based on this observation, to train quantized Mini-BranchNet (Section 5.3) and Counter-BranchNet (Section 5.4) models, I over-provision the hidden neurons and prune out the least needed neurons.

5.2.5.4 Input Pre-Processing

Branch history is a convenient method of providing recent control-flow information to a branch predictor. The trade-off is that it also contains unnecessary and obfuscated information, some of which may not be easy for a CNN to filter out. Instead of relying on the CNN to filter out all the noise, we can use other methods to pre-process the history before feeding it into the CNN predictor. For example, let us reconsider case study 1. If the algorithm for identifying the relevant region in the history was produced through other means (e.g., through an intermediate training phase with Big-BranchNet), the role of the CNN would be to simply count the insertions into the partition up to a threshold. This task would only need 1 convolution layer and, in ideal training conditions, can be learned by a single fully-connected layer, resulting in a CNN with only a 0.21KB of storage with approximately 99.94% accuracy (the remaining inaccuracy is due to limited history length). While I have done preliminary experiments on generalizing this approach, I leave a more comprehensive study on input-preprocessing for future work.

5.2.5.5 Recurrent Neural Networks

The most expensive component of a CNN branch predictor is the fully-connected layer. The fully connected layer is responsible for combining all of the signals extracted from the history by the CNN layer into a final prediction. To accomplish this, the hardware must buffer all signals produced by the CNN layer, producing buffers that require storage proportional to the length of the branch history register. An alternative approach would be to use Recurrent Neural Networks (RNNs). An ideal RNN branch predictor would process branches one at a time, updating its hidden state as branches are fetched. Sequential processing can simplify prediction tasks that rely on the order of branches in the history. For example, identifying the partitioning region of the case studies using RNNs is a relatively trivial task, or keeping track of loop iteration counts (case study 4) is easier with sequential processing.

However, designing a storage-efficient inference engine for RNNs is not trivial.

The two design optimizations of Tarsa's CNN are not applicable. Most importantly, pre-computing and buffering partial RNN outputs is not as cheap as doing the same for CNNs. Unlike the convolution operation in a CNN, the intermediate output of an RNN is position-dependent. If we were to pre-compute the RNN outputs for models trained with fixed history lengths, we need to assume each incoming branch could potentially start a new history sequence, which results in the need to maintain the state of hundreds of RNNs for predicting a single branch. Thus, I do not see a clear path for designing practical inference engines for RNN models with fixed history lengths.

A more practical approach is perhaps using RNNs with variable history lengths. In particular, inspired by the custom predictor in Case Studies 1, 2, and 4, we can associate the start of a history sequence with a static instruction. I call this static instruction a *history marker*. The occurrence of a marker resets the state of the RNN predictor, and other incoming branches would simply update the RNN state. At prediction time, fully-connected layers will make a prediction based on the RNN state. Even if this approach works, it relies on developing an automated method to identify good markers for each noisy branch. I leave the design of practical RNN inference engines for future work.

5.3 Mini-BranchNet

Mini-BranchNet is a smaller variant of BranchNet that I co-designed with an inference engine that could work as a practical branch predictor. For the most part, Mini-BranchNet is similar to Big-BranchNet with architecture knobs that I tuned

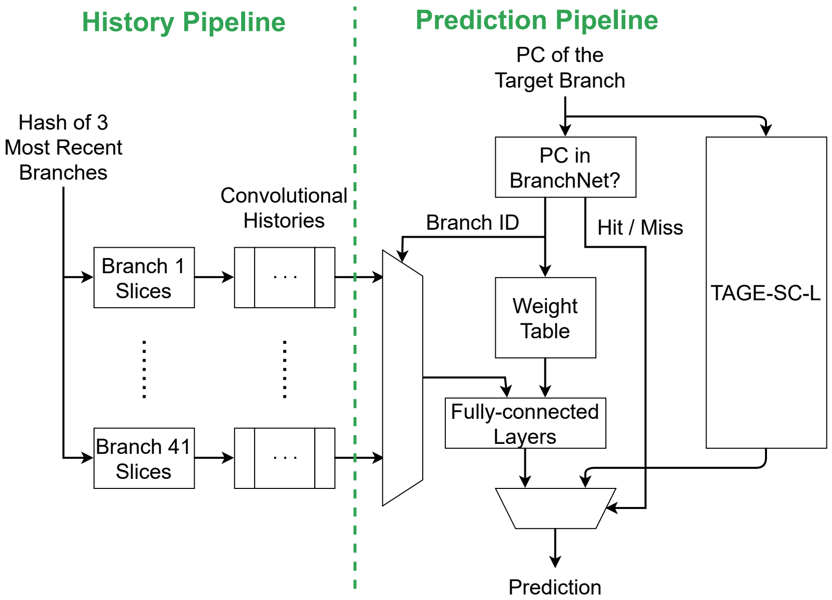
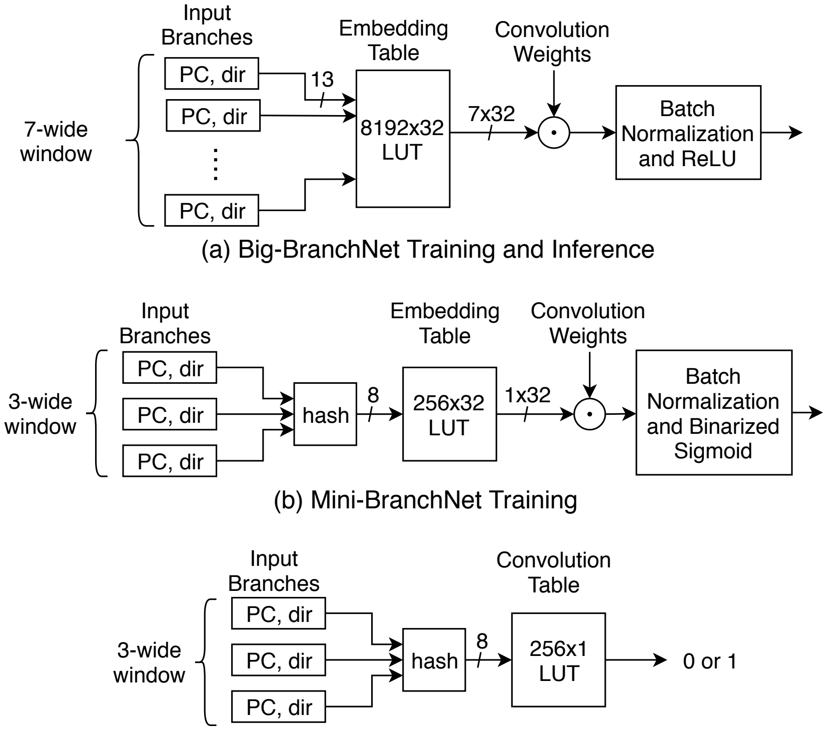


Figure 5.9: Mini-BranchNet inference engine.

to minimize storage and latency overheads. In the rest of this section, I describe key optimizations in designing the inference engine for Mini-BranchNet and some modifications to the BranchNet CNN architecture. Some of these optimizations were first proposed by Tarsa et al. [82] (explained in Section 5.1).

5.3.1 Design of the Inference Engine

Optimization 1: Maintaining Convolutional Histories. Similar to Tarsa's inference engine, the Mini-BranchNet inference engine pre-computes convolution outputs and stores the results in buffers that I call convolutional histories. Without this optimization, the predictor would need to compute 4865 convolution operations for each prediction. With this optimization, the engine computes 521 convolution operations every time that a branch is inserted into the global history. One difference with



(a) Big-BranchNet Training and Inference

(b) Mini-BranchNet Training

(c) Mini-BranchNet Inference

Figure 5.10: BranchNet convolutional layer.

Tarsa's inference engine is that the sum-pooling operation is also pre-computed, so the addition of the sum-pooling layer does not directly impact the prediction latency. Figure 5.9 shows the block diagram of a Mini-BranchNet inference engine that can predict up to 41 static branches in a program.

Optimization 2: Replacing Convolutions with Table Lookups. This optimization is also similar to Tarsa's inference engine, with the difference that Mini-BranchNet convolution windows cover three neighboring branches, not just one branch. Storing the pre-computed convolution outputs for all possible 3-branch combinations is not feasible. Thus, I slightly change the architecture of Mini-BranchNet

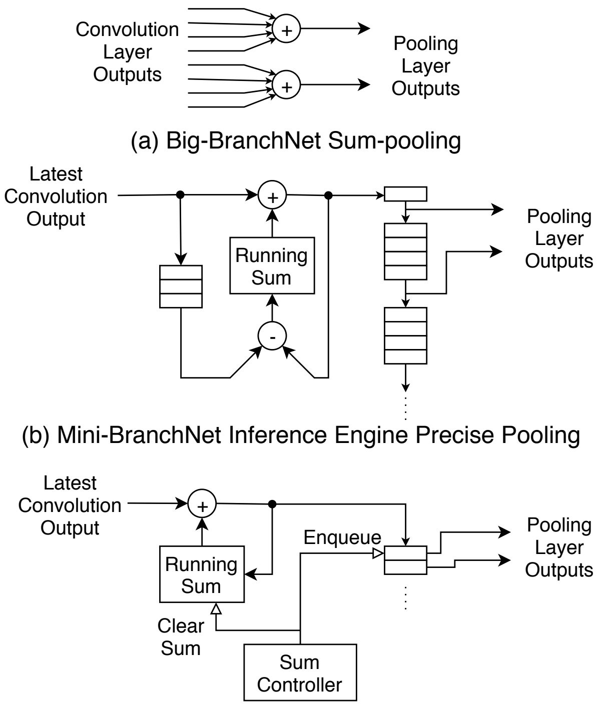

Convolution Layer Outputs
+
+
Pooling Layer Outputs
(a) Big-BranchNet Sum-pooling
Latest Convolution Output
+
Running Sum
-
Pooling Layer Outputs
(b) Mini-BranchNet Inference Engine Precise Pooling
Latest Convolution Output
+
Running Sum
Enqueue
Pooling Layer Outputs
Clear Sum
Sum Controller
(c) Mini-BranchNet Inference Engine Sliding Pooling

Figure 5.11: BranchNet 4-wide sum-pooling.

to hash the neighboring branches to approximate the convolution operation across three branches. Figure 5.10a shows how Big-BranchNet computes one convolution output. Figure 5.10b shows how Mini-BranchNet computes one convolution output during training. Figure 5.10c shows the inference engine hardware for one convolution operation, which is simply a lookup table like Tarsa's CNN. The convolutional layer of Mini-BranchNet uses binarized sigmoid activations, resulting in either 0 or 1, which is another minor difference between Tarsa's CNN and Mini-BranchNet.

Optimization 3: Using Running Sum Registers. Figure 5.11a shows the sum-pooling operation of Big-BranchNet. Mini-BranchNet inference engine uses two designs to compute the sum-pooling outputs. For shorter history slices, the engine implements *precise pooling* (Figure 5.11b). Precise pooling uses a buffer and a running sum register to constantly compute the output of the most recent pooling window and inserts the pooling outputs into a second set of buffers. As a result, this second set of buffers contains the pooling outputs of overlapping windows. At prediction time, only 1 out of P pooling outputs (recall P = pooling width) are fed into the next layer. The buffer space needed to implement precise pooling grows linearly with the history size. To reduce storage needs for longer history slices, the Mini-BranchNet inference engine uses *sliding pooling* (Figure 5.11c). Sliding pooling accumulates the pooling output of a window over multiple cycles and inserts the output in the pooling buffer once every P cycles. The trade-off is that at prediction time, the most recent convolution outputs may not have formed a complete pooling window. Thus, some of the most recent branches in the history are not used for prediction, and in general, the pooling windows have nondeterministic boundaries. In practice, this is not a problem because I only use sliding poolings in long-history slices of Mini-BranchNet, which do not rely on fine-grained positions of identified features because of their proportionally wide pooling widths. To account for sliding poolings during training, I randomly discard some of the most recent branches (0 to $P-1$ branches) that are fed into the long-history slices. This randomization makes the training algorithm resilient against nondeterministic pooling boundaries at runtime.

Optimization 4: Quantizing Fully-connected Layers. Mini-BranchNet uses fixed-point arithmetic to compute the outputs of the fully-connected layers. I empirically found that using 3 or 4 bits of precision (denoted by architecture knob

Table 5.6: Mini-BranchNet architecture knobs.

Knob	Mini-BranchNet 2KB	Mini-BranchNet 1KB	Mini-BranchNet 0.5KB	Mini-BranchNet 0.25KB	Tarsa-Ternary 5.125KB
H: History sizes	37,77,152,302,603	37,77,152,302,603	37,77,152,302,603	44,92,182	200
C: Convolution channels	4,5,5,4,4	3,3,4,4,3	3,3,3,2,2	2,2,2	32
P: Pooling widths	7,15,30,60,120	7,15,30,60,120	7,15,30,60,120	7,15,30	N/A
Use Precise pooling	Y,Y,Y,N,N	Y,Y,N,N,N	Y,Y,N,N,N	Y,Y,N	N/A
p: Branch PC bits	12	12	12	12	7
h: Convolution hash width	8	8	7	7	N/A
E: Embedding dimension	32	32	32	32	N/A
K: Convolution width	3	3	3	3	1
N: Hidden neurons	10	8	6	4	N/A
q: Fully-connected quantization	4	3	3	3	2

q) is sufficient for the sum-pooling outputs and the first fully-connected weights. The outputs of the first fully-connected layers need even less precision and can be binarized. I replace ReLU activations with Tanh to restrict the layer outputs to be between -1 and 1, which helps with quantization [17]. I also insert batch normalization and Tanh after the sum-pooling layer to stabilize the inputs to the fully-connected layers. After training is done, I fuse the batch normalization operations with the fully-connected dot products to eliminate their latency. Since the hidden fully-connected outputs are binarized, I use a lookup table to eliminate the arithmetic operations of the last layer.

Optimal Architecture Knobs. It is not storage-efficient to use the same architecture knobs for all hard-to-predict branches. Some branches need larger CNN models for good prediction accuracy, while some can be predicted well with much smaller storage budgets. Thus, Mini-BranchNet comes in 4 model configurations with varying storage budgets per branch. Table 5.6 reports the architecture knob

Table 5.7: Breakdown of the Mini-BranchNet inference engine storage requirements for one static branch.

	Using Architecture Knobs	2KB Config	1KB Config	0.5KB Config	0.25KB Config
Convolution Tables	$\sum(C_i 2^{h+1})$	0.69KB	0.53KB	0.20KB	0.09KB
Precise Pooling Buffers	$\sum(C_i(5 + P_i + q(1 + H_i - P_i)))$	0.54KB	0.11KB	0.11KB	0.09KB
Sliding Pooling Buffers	$\sum(C_i(7 + log_2(P_i) + q(H_i/P_i)))$	0.04KB	0.04KB	0.03KB	0.01KB
Fully-connected Weights	$qN\sum(C_i(H_i/P_i) + 2^N$	0.68KB	0.29KB	0.16KB	0.06KB

values for each configuration.

5.3.2 On-chip Constraints

Storage. Table 5.7 shows the breakdown of storage needed to predict a single hard-to-predict branch using the Mini-BranchNet inference engine.

Prediction Latency. Modern processors typically have two tiers of branch predictors: a less accurate light-weight predictor that provides early single-cycle predictions and a heavy-weight predictor that can later correct the prediction if necessary [33]. BranchNet is a heavy-weight predictor with multi-cycle latency.

The critical path of updating the convolutional histories consists of hashing the most recent branches, the convolution table look-up, an addition (7-bit running sum), quantization, and insertion into a convolution history buffer. Using CACTI [52] for the table lookups and counting the gate delays of the arithmetic operations, the update latency is roughly equal to the latency of a 64-bit Kogge-Stone adder (21 gate delays). Since 64-bit additions are single-cycle operations in modern processors

[22], Mini-BranchNet updates are estimated to be single-cycle operations. The critical path of the prediction pipeline for a 2KB Mini-BranchNet model includes the weight table look-up, the selection of the convolutional history, and a forward pass of the fully-connected layers (a 4-bit multiply, a 110-input 8-bit adder tree, a comparison, and accessing a 1024-entry table). The prediction latency is roughly 4 times the latency of a 64-bit Kogge-Stone adder. The latency of a 64KB TAGE-SC-L is 1.1 times the latency of the Mini-BranchNet inference engine.[2] Thus, I conservatively estimate both Mini-BranchNet and 64KB TAGE-SC-L are 4-cycle predictors.

Recovery. At the time of a pipeline flush, the convolutional histories and accumulator registers can easily be recovered using a mechanism similar to what already exists to restore long global histories. Extra shadow space is reserved in each register to hold the n most recently shifted out entries of each register. This allows us to recover the state of the predictor by shifting back the lost state, as long as we restrict our design to allow n branches in flight. The values of the sum-pooling counters need to be checkpointed.

Why Is Mini-BranchNet More Storage-Efficient than Tarsa's CNN? The sum-pooling layers are critical in enabling BranchNet to be more storage-efficient and have lower prediction latency. Without sum-poolings, each convolutional history in Tarsa-Ternary has to buffer 200 ternary values (proportional to history length). In contrast, Mini-BranchNet's convolutional histories using sliding sum-poolings need

[2] The critical path of TAGE-SC-L: accessing banked TAGE tables, tag comparisons, TAGE mux tree (with a depth of log(n)), selection logic for alternative prediction and the loop predictor, accessing the statistical corrector GEHL tables, a 20-input 6-bit adder tree, and final selection logic.

to buffer only five 4-bit values (independent of history length). Because of the large storage and latency savings of using sum-poolings, Mini-BranchNet can use longer history lengths and a second fully-connected layer (necessary for higher accuracy), while remaining smaller and faster than Tarsa-Ternary.

5.3.3 System and ISA Requirements

Mini-BranchNet requires collaboration across the software stack for loading trained Mini-BranchNet models to the on-chip unit at runtime. I envision an approach where the program is modified to load trained BranchNet models into the on-chip inference engines at the beginning and the operating system (OS) is responsible for managing the state of the engines during context switches. The ISA should provide BranchNet instructions that the program and OS use to enable, disable, or update the on-chip engine. As a design choice, these instructions may be implemented as non-blocking instructions to hide the overhead of loading BranchNet models. Lee et al. [44] proposed a similar approach for using the OS to save and restore the state of runtime branch predictors during context switches, albeit for a different goal of mitigating context switch penalties on branch prediction accuracy. I leave a more detailed analysis and evaluation of System and ISA requirements or alternative approaches to future work.

5.4 Counter-BranchNet

In this section, I introduce Counter-BranchNet, which is a variant of BranchNet that uses the correlated branch information to process the branch history. By

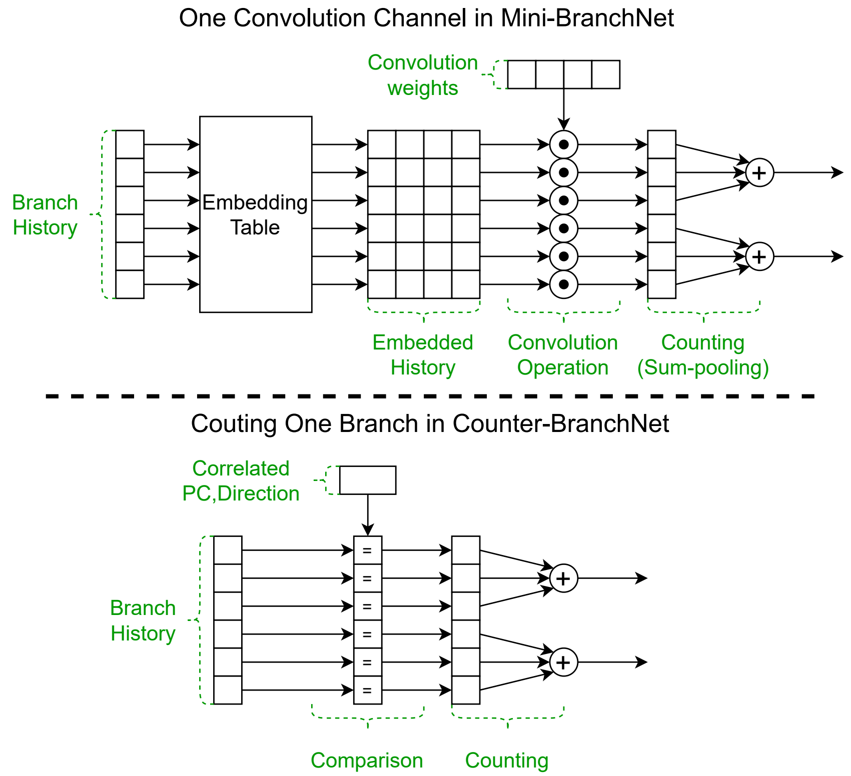


Figure 5.12: Mini-BranchNet vs. Counter-BranchNet.

explicitly tracking the most correlated branches, using Counter-BranchNet alongside Mini-BranchNet improves the total MPKI reduction.

Counter-BranchNet replaces the convolutional and sum-pooling layers with customized structures for counting just the most correlated branches. As long as there are only a few correlated branches, this would result in a smaller and more accurate inference engine compared to the Mini-BranchNet inference engine. Figure 5.12 compares Mini-BranchNet and Counter-BranchNet at a high level. Note that using Counter-BranchNet relies on knowing which branches need to be counted in the history (counting all static branches is expensive). Thus, Counter-BranchNet

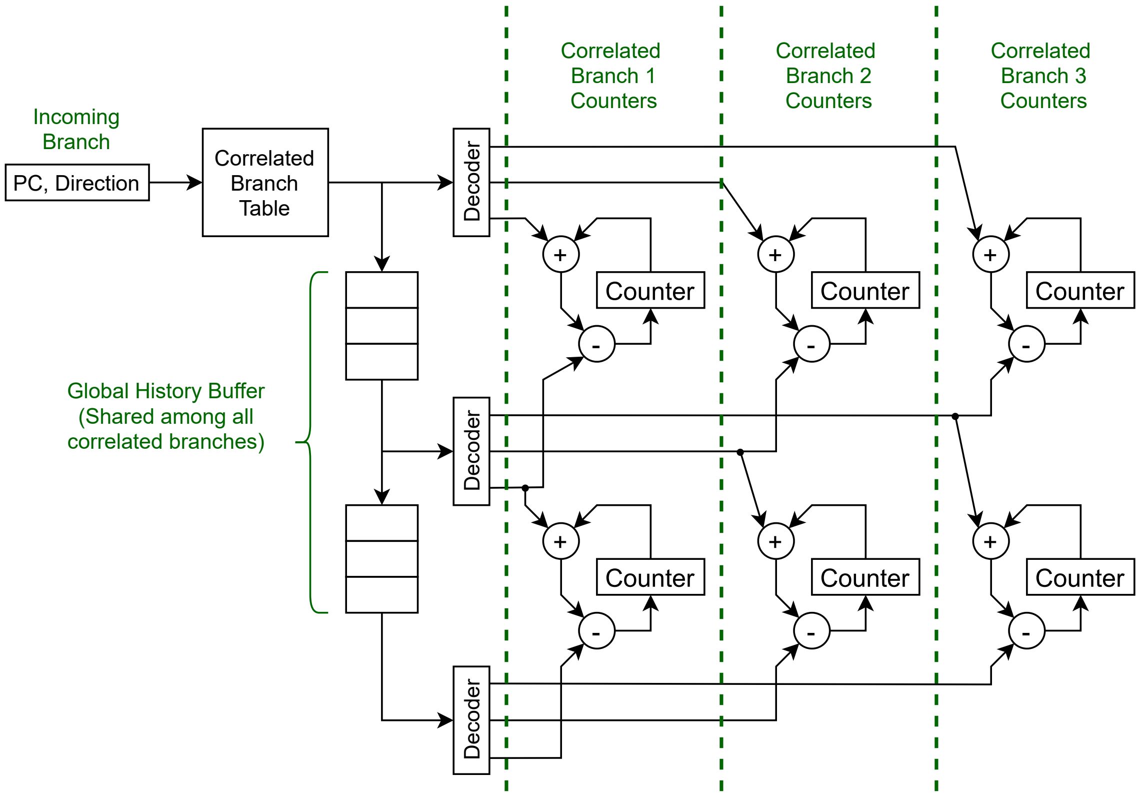

Correlated Branch 1 Counters
Correlated Branch 2 Counters
Correlated Branch 3 Counters
Incoming Branch
PC, Direction
Correlated Branch Table
Decoder
Decoder
Decoder
+
-
Counter
Global History Buffer (Shared among all correlated branches)

Figure 5.13: Branch counting hardware for three correlated branches.

needs a primary training pass, where a Big-BranchNet model is used to identify the most correlated branches (as explained in Chapter 4). After the correlated branches are identified, Counter-BranchNet is then trained from scratch to learn an accurate prediction function based on the counts of the correlated branches.

The design of the inference engine is similar to Mini-BranchNet. Most importantly, the inference engine of Counter-BranchNet still processes branches as they are inserted into the history. The key difference is that instead of accessing convolution tables to determine the value of an incoming branch, we compare the incoming branch

PC and direction to the PC and directions of correlated branches. Figure 5.13 shows how a simple inference engine can maintain the counts of a correlated branch. The *Correlated Branch Table* assigns a unique positive index for correlated branches and assigns zero to all uncorrelated branches. For example, if there are three correlated branches like in the figure, the branch index is a 2-bit number, with a value of 1, 2, or 3 for the three correlated branches, or the value of 0 for all other branches. These branch indices are pushed in a queue that needs to be as long as the history length. At counter boundaries (Figure 5.13 uses 3-wide counters with a history length of 6), the branch indices are fed into binary decoders. The decoders control how the counters are updated to maintain the correct count of each correlated branch in its corresponding branch window. These counters are then used as inputs to quantized fully-connected layers, which are the same as the fully-connected layers of Mini-BranchNet.

Similar to Mini-BranchNet, Counter-BranchNet uses multiple slices with geometric history lengths and pooling widths (pooling width is the window of history that each counter tracks). Even though there are no convolution operations in Counter-BranchNet, I still use *channel* to refer to the counters corresponding to one correlated branch in one history slice. For storage optimizations, I train the model in two phases. First, I train with all available channels with some regularization penalty on the fully-connected layers. Second, I prune out the slices with the least corresponding fully-connected weight magnitudes. This is a common pruning technique for neural networks in general.

This design could be much smaller than Mini-BranchNet if there are only a few correlated branches. Correlated Branch Table only needs to be as large as the

Table 5.8: Counter-BranchNet architecture knobs.

Knob	Counter-BranchNet 0.1KB	Counter-BranchNet 0.35KB	Counter-BranchNet 0.8KB	Counter-BranchNet 1.0KB
History sizes	35,150,300	35,150,600	36,75,150,300,600	36,75,150,300,600
Pooling widths	7,30,60	7,30,120	6,15,30,60,120	6,15,30,60,120
Number of Correlated branches	4	16	32	32
Counter channels after pruning	4	15	30	45
Hidden neurons	5	5	6	5
Counter bits	4	4	4	4
Fully-connected quantization bits	6	6	6	6

maximum number of correlated branches. The global history buffer is shared among all correlated branches (in contrast with Mini-BranchNet, where each convolution channel requires its own buffer). Since correlated branches are sparse in the history, no normalization is needed after the counter values (Mini-BranchNet normalizes the values of the counters and quantizes them to a smaller range). Table 5.8 summarizes the configurations that I use for evaluation, along with the total storage costs.

5.5 Results

I use the same methodology as Section 3.3.5 to create the training set, the validation set, and the set to train Mini-BranchNet and report the results. The training hyperparameters are almost the same, except that I use the quantization algorithm of Courbariaux et al. [17] during training. I found the training process for quantized small Mini-BranchNet models to be unstable, so I train each model in multiple phases. In the first phase, I train all layers. At the end of the first phase, I convert the convolution layers to lookup tables and freeze the tables (i.e., the convolution tables will not be trained further). Then I fine-tune the rest of the layers and gradually freeze the next layer until the whole model is trained.

Training Counter-BranchNet models is similar with the following differences. First, I use the results of Section 4.3 to identify the most correlated branches for each noisy hard-to-predict branch. Then, I train Counter-BranchNet without quantization. Then, I prune out the least impactful channels and fine-tune the model. Finally, I quantize the fully-connected layers and fine-tune the model again.

I evaluate the IPC of benchmarks using Scarab [2], an execution-driven, cycle-level simulator for x86-64 processors, which accurately models branch misprediction behavior by fetching and executing wrong-path instructions. I use a 4KB gshare predictor as the single-cycle lightweight predictor and TAGE-SC-L and BranchNet as 4-cycle late predictors. If the prediction of the late predictor disagrees with the early predictor, Scarab flushes the frontend and re-fetches the instructions after the branch using the new prediction. I configure the processor to resemble a high-performance processor: 6-wide fetch, 512-entry ROB, 2MB L2 Cache in a two-level cache hierarchy, 10-stage frontend pipeline, execution latency similar to an Intel Skylake processor [22], and DDR4 main memory simulated with Ramulator [38].

Similar to previous chapters, to report the average MPKI reduction, I first compute the arithmetic mean of MPKI with and without a CNN predictor. Then, I compute the relative MPKI reduction of the average. To report the average IPC improvements, I compute the speedup of each benchmark, then compute the geometric mean of all the benchmark speedups, and convert the speed to relative IPC improvement.

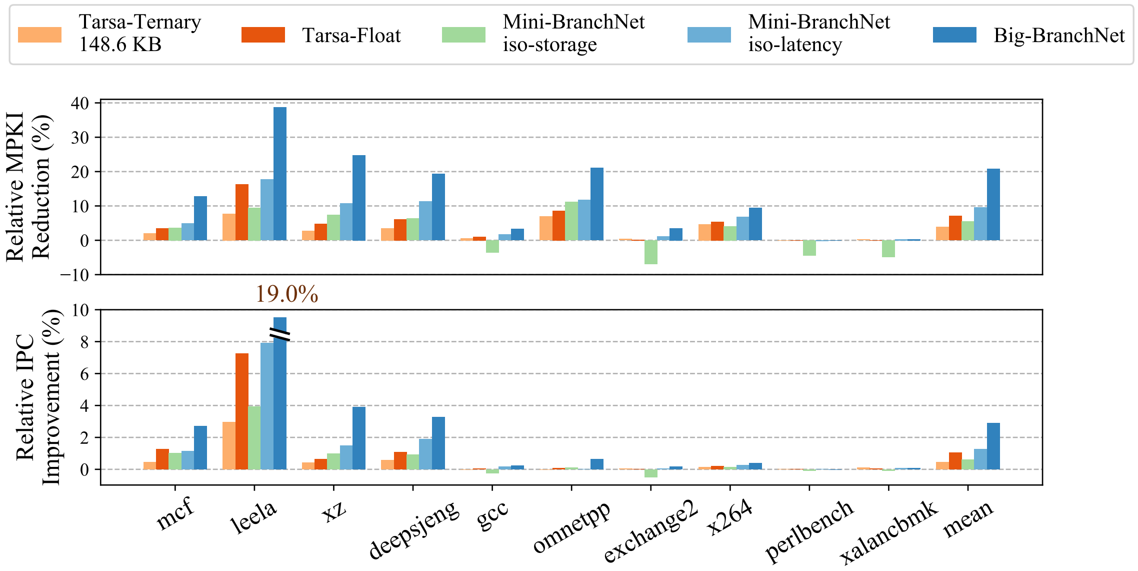


Figure 5.14: MPKI and IPC improvement of BranchNet and Tarsa's CNN compared to 64KB TAGE-SC-L.

5.5.1 Mini-BranchNet

Figure 5.14 shows the MPKI and IPC improvement of different configurations of BranchNet and Tarsa's CNN compared to a 64KB TAGE-SC-L baseline. I disabled the local history components of the Statistical Corrector because realistic processors avoid maintaining speculative local histories because of design challenges. For each Mini-BranchNet storage budget, I tried all possible assignments of top hard-to-predict branches to configurations and used the best combination of models across all SPEC benchmarks.

I evaluated BranchNet in three settings. The iso-storage setting pairs an 8KB Mini-BrachNet (one 2KB model, one 1KB model, seven 0.5KB models, and six 0.25KB models) with a 56KB TAGE-SC-L,[3] showing 5.5% average MPKI reduction, up to 9.5%, and 0.6% average IPC improvement, up to 3.9%. The iso-latency setting pairs a 32KB Mini-BranchNet (eight 2KB models, seven 1KB models, ten 0.5KB models, and sixteen 0.25KB models) with the baseline 64KB TAGE-SC-L, showing 9.6% MPKI reduction on average (up to 17.7%) and a geometric mean of 1.3% IPC Improvement (up to 7.9%). Finally, the Big-BranchNet setting shows the opportunity if it were possible to get the full benefits of floating-point BranchNet models with a 4-cycle latency at runtime: 2.9% average speedup, up to 19.0% for the best benchmark.

I evaluated both configurations of Tarsa's CNNs: Tarsa-Float and Tarsa-ternary. Tarsa-ternary is analogous to iso-latency Mini-BranchNet but with a much larger storage budget (5.125KB per branch, up to 29 static branches). Mini-BranchNet architecture and optimizations allow it to use longer histories and a deeper network with less storage. Thus, as Figure 5.14 shows, BranchNet is significantly more accurate than Tarsa's CNN.

Figure 5.15 shows the sensitivity of iso-latency Mini-BranchNet to its storage budget. Since storage more than 32KB shows diminishing returns, I chose 32KB as the budget for iso-latency Mini-BranchNet.

Table 5.9 illustrates the negative impact of various constraints and approximations needed to make Mini-BranchNet practical. Quantization of convolution layers has the least significant impact on MPKI reduction, which agrees with our intuition

[3] I built the 56KB TAGE-SC-L by decreasing the number of table entries and tag bits of TAGE.

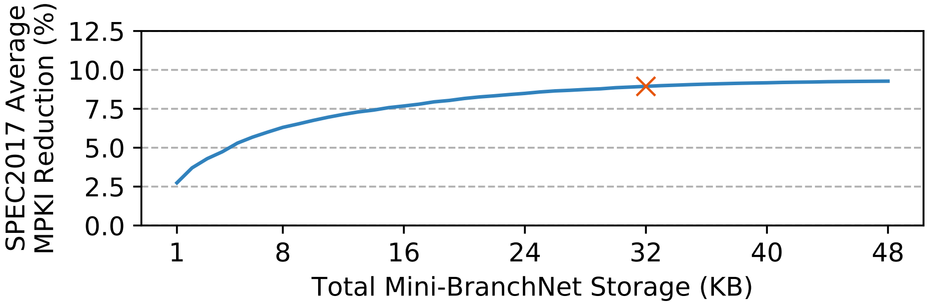


Figure 5.15: Sensitivity of iso-latency Mini-BranchNet to its storage budget on SPEC2017 benchmarks.

Table 5.9: Progression of MPKI reduction of *leela* from Big-BranchNet to Mini-Branchnet.

Big-BranchNet: No branch capacity limit	35.8 %
Big-BranchNet: Same branches as Mini-BranchNet	25.1 %
Mini-BranchNet: Floating-point	20.0 %
Mini-BranchNet: Quantized convolution	18.7 %
Mini-BranchNet: Fully-quantized	15.7 %

that the role of the convolution layer is to simply identify correlated branch patterns, so a binary output should be sufficient.

Note: these sensitivity studies were done with a slightly different training setup, resulting in lower MPKI reduction compared to Figure 5.14.

5.5.2 Counter-BranchNet

Figure 5.16 shows the MPKI reduction of iso-latency inference engines in three settings: using only Mini-BranchNet models, using only Counter-BranchNet,

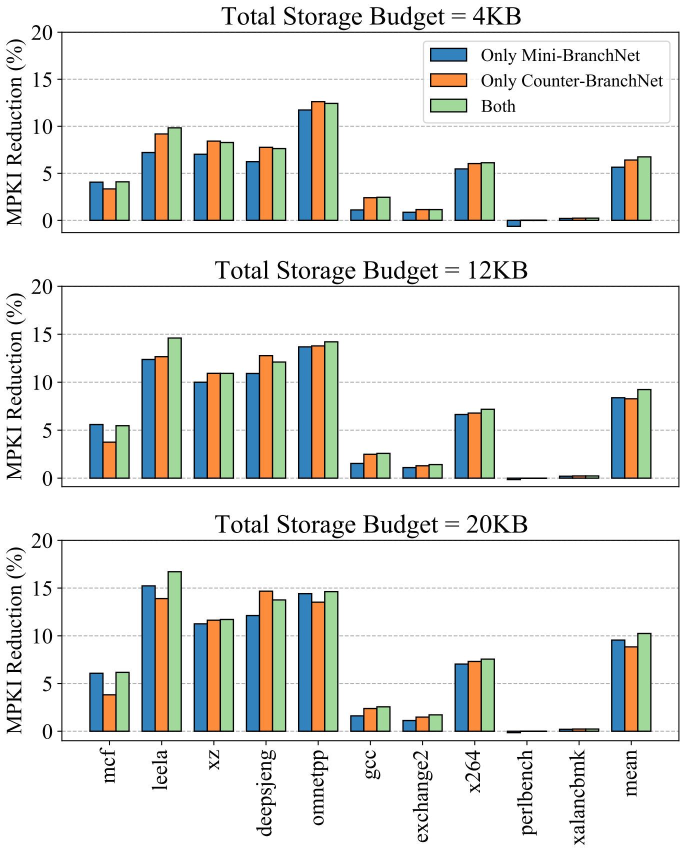


Figure 5.16: The MPKI reduction of iso-latency Mini-BranchNet and Counter-BranchNet in three storage budgets.

or mixing both to maximize the accuracy.[4] In each configuration, I use dynamic programming to figure out the best model architecture (Mini-BranchNet and Counter-BranchNet) and the best model size to assign to each static noisy branch. The results show that the best inference engine depends on the storage budget. Counter-based inference engines are more storage-efficient at lower storage budgets where a few small

[4] To save computation time, the mixed approach uses 0.1KB Counter-BranchNet, 0.35 Counter-BranchNet, 0.5KB Mini-BranchNet, 1.0KB Mini-BranchNet, and 2.0KB Mini-BranchNet.

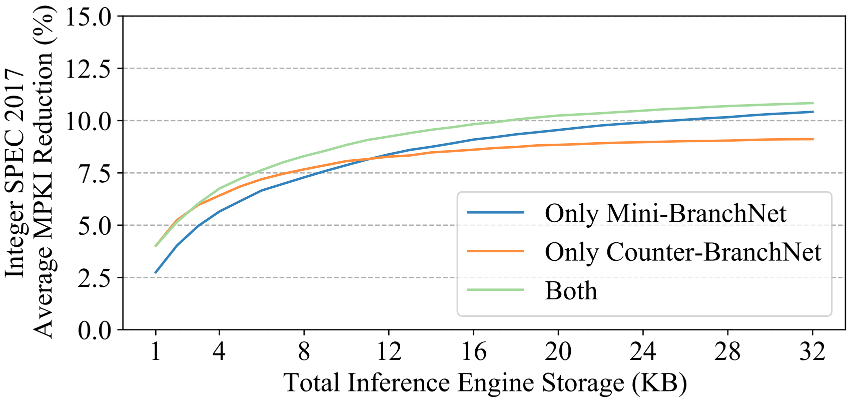


Figure 5.17: Sensitivity of iso-latency Mini-BranchNet and Counter-BranchNet to the storage budget.

Table 5.10: Accuracy of Mini-BranchNet and Counter-BranchNet for some noisy branches of *leela*.

Model	Mini-BranchNet				Counter-BranchNet			
Storage	0.25KB	0.5KB	1.0KB	2.0KB	0.1KB	0.35KB	0.8KB	1.0KB
Br1	93.9%	97.5%	99.0%	99.3%	99.95%	99.73%	99.87%	99.93%
Br2 (case study 3)	92.7%	95.9%	97.1%	98.7%	71.8%	99.8%	99.7%	99.8%
Br3 (case study 4)	88.4%	91.5%	93.0%	94.4%	81.3%	88.2%	91.6%	91.4%
Br4	74.9%	77.4%	77.5%	79.9%	71.8%	74.6%	76.6%	75.9%

models can achieve significant MPKI reduction. At higher storage budgets, we can afford the cost of Mini-BranchNet models to achieve higher accuracy. Mixing both models gets the benefits of both inference engines and maximizes the accuracy at all storage budgets. Figure 5.17 shows the sensitivity across more storage budgets.

Table 5.10 reports the accuracy of some of the most improved branches of *leela*. Counter-BranchNet is both more accurate and more storage-efficient in predicting Br1 and Br2. Mini-BranchNet is more accurate and more storage-efficient in predicting Br3 and Br4. The difference is explained by examining the source code of the branches. Br1 and Br2 have simple branch relationships that could be perfectly replicated by counting a few correlated branches and learning a simple prediction function. Br2 is the same branch as Case Study 3 in Section 5.2.3 that can be predicted accurately with a simple custom predictor. On the other hand, Br3 and Br4 have complicated branch relationships that cannot be represented by counting correlated branches. Br3 is the same branch as Case Study 4 in Section 5.2.4, where the branch relationship was much more complicated than what a small neural network could learn. Br4 is even more complicated and I could not find any obvious prediction function to improve its accuracy. Still, 2KB Mini-BranchNet marginally improves its accuracy compared to 64KB TAGE-SC-L, which contributes to the overall MPKI reduction but Counter-BranchNet is less accurate than TAGE-SC-L. In summary, Counter-BranchNet is more effective at smaller storage budgets for predicting branches with simple prediction functions but Mini-BranchNet is better to maximize the accuracy at higher storage budgets.

Takeaway: While the improvements of the counter-based BranchNet are not groundbreaking, it is simpler to build and is more storage-efficient with smaller storage budgets. Further investigation and analysis are needed to understand if the counter-based design can be improved to accurately predict branches with more complicated branch relationships. Nonetheless, Counter-BranchNet provides insight into how BranchNet works and how there is room for simplifying the inference engine to ease adoption.

Chapter 6

Filtering Uncorrelated Branches in TAGE Histories Using BranchNet

Chapter 5 described practical BranchNet inference engines to predict noisy branches. While using on-chip inference engines is the most effective way of using BranchNet for maximizing the prediction accuracy, it requires significant engineering effort for adoption in practice. In this chapter, I propose an alternative approach for using BranchNet. Instead of using BranchNet as a black-box predictor, I examine trained CNN models to explicitly identify correlated branches. Using this information, I filter the global branch history of TAGE to include only the outcomes of correlated branches. Eliminating all the noise from the history significantly reduces the total number of history patterns observable by TAGE. Thus, filtering the branch history leads to less allocation pressure and faster warmup time in TAGE, resulting in improved prediction accuracy and better storage-efficiency.

The process of explicitly identifying correlated branches was already discussed in Chapter 4. This chapter introduces *Filtered TAGE*, a slightly modified TAGE branch predictor with filtered global branch histories for a few noisy branches that benefit the most from filtering. This section details the predictor design, discusses the system and ISA requirements of filtered TAGE, and evaluates the improvements

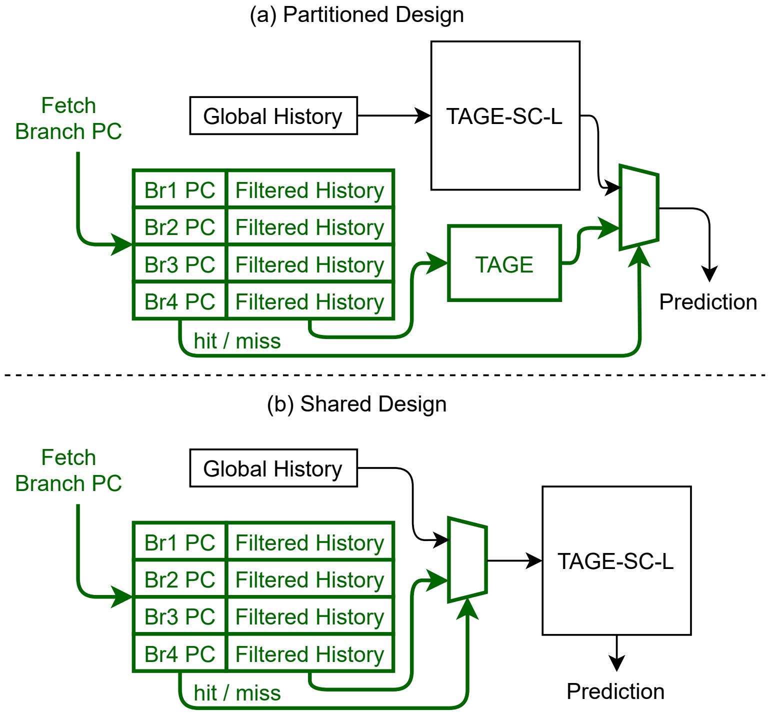


Figure 6.1: Prediction using filtered TAGE.

due to filtered TAGE.

6.1 Filtered TAGE Design

Prediction: In addition to global history, filtered TAGE has a table of filtered histories. To make a prediction at runtime, filtered TAGE first looks up the filtered history table using the fetched branch PC. If the lookup is a hit, the corresponding filtered history is used as the history input to a TAGE partition dedicated for the filtered hard-to-predict branches. If the lookup is a miss, the main global branch

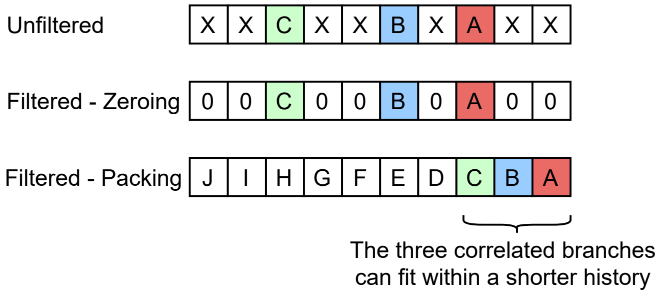
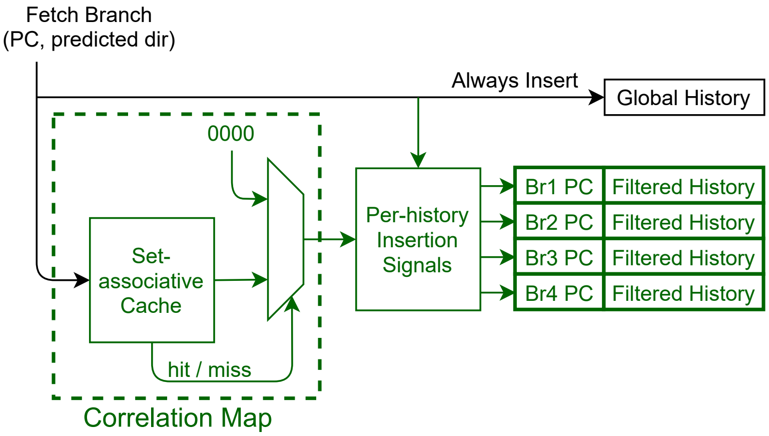


Figure 6.2: Zeroing and packing filtering strategies.

history and the baseline TAGE-SC-L are used. Figure 6.1a shows the changes to the prediction pipeline of TAGE, with my modifications bolded and colored in green.

Figure 6.1b shows an alternative design, where both filtered and unfiltered branches use the same TAGE-SC-L tables. This design is more storage-efficient as branches with filtered histories simply compete with all other branches to allocate entries in TAGE tables. However, the filtered history lookup time is on the critical path of the overall prediction latency.

Filtering: There are two methods to maintain filtered histories. The first method is to mask the uncorrelated branches, i.e., always update all histories but zero out the bits corresponding to uncorrelated branches. The second method is to not insert uncorrelated branches in the history at all, effectively packing all the relevant history bits into the most recent bits of the branch history. Using the terminology of Thomas et al. [83], I refer to these two methods as zeroing and packing, respectively. As Figure 6.2 shows, zeroing maintains the absolute position of branches in the global history while packing increases the density of useful bits in the history. Packing results in the correlated branches fit in a shorter history. Packing is a more effective choice



inin

Figure 6.3: Updating histories in filtered TAGE.

for filtered TAGE because packing enables prediction from shorter history tables of TAGE, which in turn reduces allocation pressure and warmup time.

Updating the filtered histories: In addition to inserting all branches into the baseline global history, filtered TAGE examines each branch and decides how to update the filtered histories. To achieve this, filtered TAGE uses a structure called the *correlation map*. As Figure 6.3 shows, the correlation map takes the incoming branch as the input and produces a bit vector as the output. The bit vector indicates which filtered histories should contain the incoming branch.[1] If a branch misses in the correlation map, the bit vector is implicitly assumed to be all zeros, meaning none of the filtered branches are correlated with the incoming branch. Depending on the filtering strategy, the bit vector is used differently. With zeroing, all filtered histories are updated, but the bit vector clears the bits that are inserted into the

[1] Each incoming branch could be correlated with any combination of the filtered histories, therefore the bit vector is as wide as the number of filtered histories.

uncorrelated histories. With packing, the bit vector is used as a load-enable signal to determine which histories are updated. Internally, the correlation map is organized as a partially-tagged set-associate cache. The correlation map is pre-populated before the program starts or at context switches, thus, no runtime replacement policy is needed. Section 6.2 details the software algorithm for deciding which branches should be inserted into the correlation map.

Along with the main global branch history (which can be very long), TAGE also uses a 27-bit path history. The path history complements the traditional direction history by maintaining the program counters of the branches that are inserted into the global history. Both packing and zeroing strategies can also be applied to the path history. In this case, zeroing means shifting the path register for each incoming branch without folding in the uncorrelated branch.

Number of insertion bits: The most recent variant of TAGE inserts a hash of the branch PC and its outcome into the global history, which could be 2 or 3 bits depending on the branch type. The filtered histories are updated with the same hashing algorithm used by the baseline TAGE.

Filtered history lengths: Filtered history lengths do not need to be as long as the main global branch history. I use a shorter history length for filtered histories to reduce the storage overhead of filtering.

Recovery: The recovery mechanism depends on the filtering strategy. If using zeroing filtering, we need to extend the width of filtered histories to cover the maximum number of speculative branches in the history. Then, for recovery, we can simply shift the registers by the number of bits corresponding to the flushed branches in the pipeline.

If using packing filtering, since the branches in packed histories are not aligned, filtered TAGE cannot use the same recovery mechanism as the main global history. Instead, it needs to checkpoint all updates to the filtered histories. To save storage, filtered TAGE uses a buffer of history checkpoints that should ideally be as large as the maximum number of in-flight speculative updates to the filtered histories. After every update to a filtered history, the predictor allocates an entry in the checkpoint buffer for that update. In addition, along with each in-flight branch in the processor (both filtered and unfiltered branches), the processor maintains pointers to the checkpoints of all the filtered histories. If any branch mispredicts, the processor uses these pointers to load the corresponding checkpoints for the filtered histories before the branch was fetched.

Regardless of the filtering strategy, we need to recover the filtered path histories using checkpoints.

Storage: The main storage cost of filtered TAGE with a partitioned TAGE design is due to the TAGE tables dedicated to branches with filtered histories. This cost does not exist for a filtered TAGE with shared TAGE-SC-L tables.

Table 6.1 outlines the storage overhead of maintaining filtered histories (using packing filtering) as a function of design parameters with the following definitions: N = number of filtered histories, H = history length, P = path history length, S = number of correlation map sets, W = number of correlation map ways, tag = width

Table 6.1: Storage Overhead of maintaining filtered histories.

	Formula	Total size Main filtered TAGE configuration
Filtered Histories	N.(1 + 48 + H + P)	0.4KB
Correlation Map	S.W.(1 + tag + N)	1.4KB
Checkpoint Buffer	C.(H + P)	5.5KB
Checkpoint Pointers	MaxBr.N.log2(C)	2.6KB

of correlation map partial tag, C = number of checkpoint buffer entries, MaxBr = maximum number of in-flight speculative branches. My chosen configuration uses 24 packing filtered histories with a history length of 64 bits, a path history of 27 bits, a 384-entry 6-way set-associative correlation map partially-tagged with 4 bits, 100 maximum in-flight branches, and a 500-entry checkpoint buffer. The total storage overhead is 10KB. Note that most of the storage overhead is due to the recovery checkpoints.

Latency: The latency overhead of filtered TAGE with a partitioned TAGE design is marginal, as the filtered histories and the filtered TAGE tables are looked up in parallel to the main unfiltered TAGE-SC-L.

In the case of filtered TAGE with a shared TAGE-SC-L design, accessing the filtered history table is on the critical path. Using CACTI for table accesses and by counting gate delays for main logical components, I estimate the lookup time of the filtered histories to be about 10% of the prediction latency of a 64KB TAGE-SC-L, which may lead to increasing the overall prediction latency by 1 cycle.

System and ISA Requirements: The System and ISA requirements of filtered TAGE are similar to those of BranchNet inference engines. Filtered TAGE requires modifications to several layers of the computing stack to enable offline training. A profiling pass is needed to train BranchNet models to identify correlated branches. This information must be appended to the program binary. The ISA is then augmented with new instructions, which enable or disable filtered TAGE and update the correlation map information. Finally, the OS should use this interface to save/restore the correlation map state during context switches.

6.2 Evaluation Methodology

Simulation Infrastructure: I use the same methodology as previous chapters to train BranchNet models offline and use SPEC ref inputs to report the final numbers.

To report branch MPKI, I use a trace-driven simulation with a methodology Similar to Championship Branch Prediction 2016, i.e., only the branch predictor is simulated, and the predictor is updated immediately after each prediction. I use the implementation of TAGE-SC-L in Championship Branch Prediction 2016 [65] as the baseline, and add support for filtered histories. The Statistical Corrector and the Loop Predictor are not modified, i.e., they do not use filtered histories in any way.

To report performance, I use Scarab [2], a cycle-level x86 simulator. The core is configured as follows: 8-wide fetch, 1024-entry ROB, 256-entry unified reservation station, 100-entry branch buffer, single-cycle branch predictor, DDR4 main memory, 128KB data cache, and 1MB L2 cache.

Identifying correlated branches with BranchNet: While offline training has no runtime cost, it is still desirable to reduce the training time and required computational resources. Thus, I modify the BranchNet architecture knobs to balance training time and predictive capabilities. I use 3 history lengths (100, 200, and 600) with pooling widths of (10, 20, and 60), a convolution width of 8, an embedding width of 32, and two 16-neuron hidden fully-connected layers. I train with a batch size of 128 for a maximum of 6000 steps with an embedding regularization coefficient of 0.0001. If the training loss does not improve in 100 steps, I stop the training process to save time. I use the concatenation of branch direction and the least significant bits PC (12 PC bits to match the correlation map tag bits) as the embedding index. With these hyperparameters, it takes less than 20 minutes to train 100 BranchNet models on a single Nvidia Titan Xp GPU.

Selecting the most improved hard-to-predict branches and their correlated branches: The optimal number of selected correlated branches for each branch is different. The results in Section 4.3 show that some branches may be predicted accurately using only 4 correlated branches, while others may need 16 or even 32 correlated branches. Using more branches than necessary increases the number of history patterns that TAGE observes, which defeats the purpose of filtering. Furthermore, since the hardware only supports a fixed number of filtered histories, we need to enable filtering for branches that benefit the most from filtering. To make these decisions at compile-time, similar to BranchNet's methodology, I use a validation set to estimate the MPKI reduction due to improving each hard-to-predict branch.

First, I run TAGE-SC-L on the validation traces in 5 configurations: baseline TAGE-SC-L, and four configurations of filtered TAGE where each candidate noisy branch uses filtered histories that contain either top 4, top 8, top 16, or top 32 correlation branches. In this phase, filtered TAGE is configured without a capacity limit on the number of filtered histories and the size of the correlation map. I compare the four filtered TAGE-SC-L configurations to the baseline to estimate the MPKI reduction per branch. Then for each hard-to-predict branch, I greedily choose the filtering configuration (4, 8, 16, or 32 correlated branches) that leads to the highest MPKI reduction in the validation set. Branches with no MPKI reduction among the 4 configurations are eliminated from the candidate list. I sort the remaining candidate branches based on their estimated MPKI reduction and pick the most improved ones to use the limited number of available filtered histories. The result is a list of noisy branches that benefit the most from filtering, where each noisy branch has a corresponding set of 4, 8, 16, or 32 correlated branches. Finally, the inverse function of this list is loaded into the correlation map. Recall that the correlation map is a set-associative structure that identifies if an incoming branch should be inserted in any of the filtered histories.

Because of the limited capacity and associativity of the correlation map, not all selected correlated branches may fit inside the on-chip predictor. Thus, we also need an algorithm to select the most likely useful correlated branches. For this purpose, I sort the correlated branches by their embedding score rank in the identification step and the number of noisy branches that they are correlated with. When filling a set in the correlation map, if the number of candidate branches is more than the associativity, I drop the least needed branches according to the final sort order.

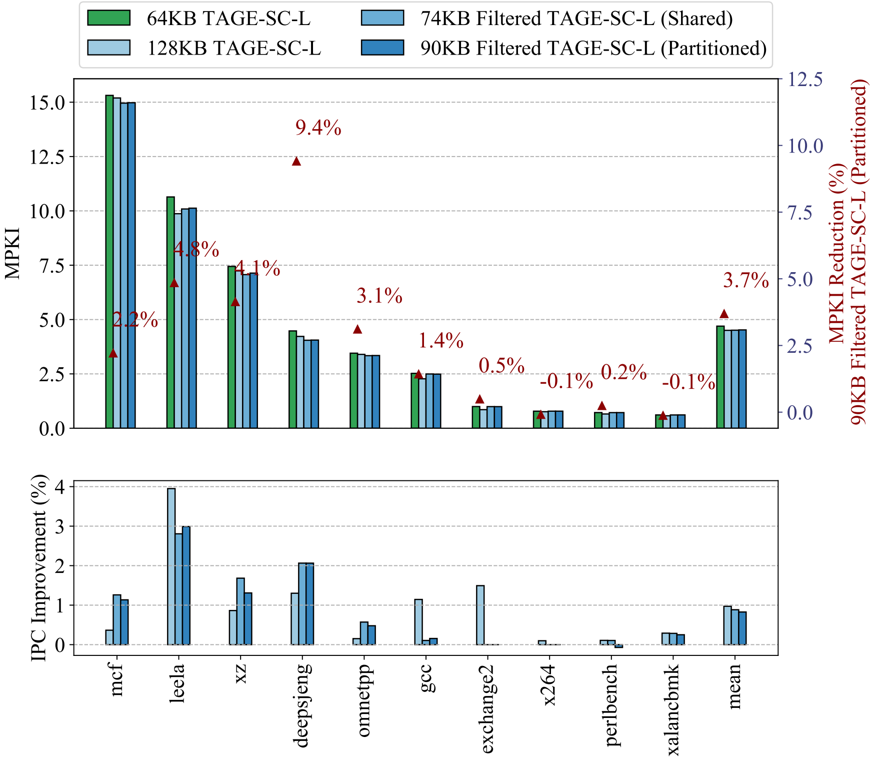


Figure 6.4: MPKI reduction and speedup of using filtered TAGE with a 64KB TAGE-SC-L baseline.

6.3 Results

I compare the MPKI and IPC of using 64KB TAGE-SC-L, 128KB TAGE-SC-L,[2] filtered TAGE with shared TAGE tables (10KB storage overhead), and filtered TAGE with an extra 16KB TAGE partition for filtered branches (26KB storage over-

[2] To build a 128KB TAGE-SC-L, I created a few TAGE-SC-L configurations by adjusting the following parameters: bimodal table size, TAGE table bank sizes, number of TAGE table banks, number of TAGE tag bits, Statistical Corrector counter width, Statistical Corrector GEHL table sizes. I report the MPKI and IPC of the best 128KB configuration I could find on SPEC benchmarks.

head). Both filtered TAGE configurations improve the MPKI by 3.7% across all SPEC 2017 Integer benchmarks, up to 9.4% for benchmark *deepsjeng*. This amount of improvement is about the same as expanding the baseline TAGE-SC-L to 128KB, which needs 70% and 40% more storage compared to the two TAGE configurations, demonstrating the storage-efficiency and scalability of filtering TAGE. Figure 6.4 shows the MPKI of all 4 configurations, the MPKI reduction of partitioned filtered TAGE compared to the baseline, and the speedup of 128KB TAGE-SC-L and the two filtered TAGE configurations. Note that not only 128KB TAGE-SC-L is less storage-efficient than filtered TAGE, but it is also worse for prediction latency as expanding the main table sizes increases the prediction latency. The IPC gain of filtered TAGE is 0.9% on average (geometric mean), and up to 3.0% for the best benchmark *leela*.

The benchmarks that benefit the most are generally the same as those that benefit the most from BranchNet. Filtered TAGE has the most impact on *deepsjeng*, *leela*, and *xz* which contain many hard-to-predict branches that become predictable if we isolate the relevant correlated branches. *Mcf* and *omnetpp* are also improved, but most mispredictions in these two benchmarks are due to data-dependent branches which do not benefit from filtering. Still, filtered TAGE is more effective than 128KB TAGE-SC-L for these benchmarks. Filtered TAGE improves a few branches in *gcc* and *exchange2*, but not enough to significantly reduce the overall MPKI, resulting in less MPKI reduction and speedup compared to 128KB TAGE-SC-L. Filtered TAGE only reduces the MPKI of *x264*, *perlbench*, and *xalancbmk* by less than 1%. However, these benchmarks do not suffer from high misprediction rates in the first place, so the MPKI reduction is not as important.

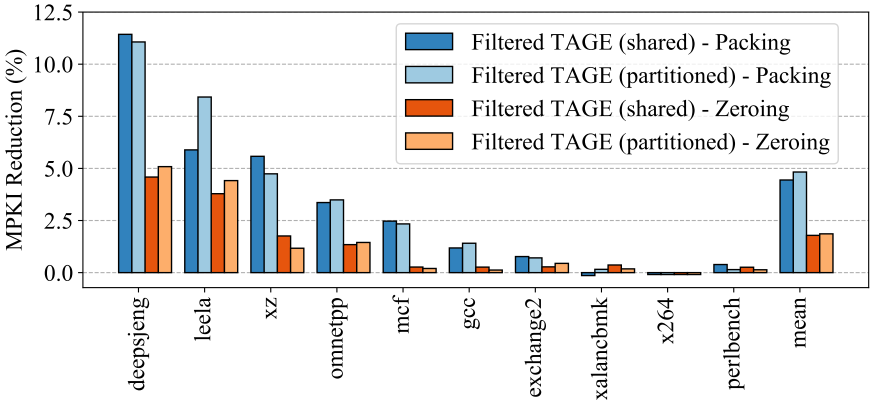


Figure 6.5: MPKI Reduction of filtered TAGE compared to TAGE-SC-L. All filtered TAGE use up to 32 active filtered histories and 3000 bits of history.

6.3.1 Filtering Mechanism

Figure 6.5 shows the MPKI reduction of two different filtering mechanisms (zeroing and packing) and two filtered TAGE designs (shared and partitioned TAGE tables). The results show that packing is the best filtering mechanism to maximize prediction accuracy. This is in contrast to the results of Thomas et al. [83], where zeroing has a marginal advantage over filtering. The primary difference is that they use the filtered histories as an input to a predictor with a single history length, so the packing does not affect the history length. But, for TAGE, packing the relevant bits into a shorter history length enables TAGE to use shorter history tables to make a prediction, which has a first-order impact on the allocation pressure on the tables and warm-up time. Furthermore, my packing strategy is slightly different because of different recovery methods. Thomas et al. always maintained unpacked histories (for ease of recovery) and packed the histories on the critical path of making

Table 6.2: Internal TAGE statistics for some of the most improved branches on *leela*'s most representative simpoint.

	Accuracy			Mean Prediction Table ID			Number of Allocations			Number of Unique Entries		
	base	zeroing	packing	base	zeroing	packing	base	zeroing	packing	base	zeroing	packing
Br1	79.4	80.3	97.0	8.2	9.8	4.7	175K	147k	34K	86K	74K	24K
Br2	89.3	89.4	95.7	6.7	11.2	3.1	36K	26k	18K	19K	11K	14K
Br3	85.0	96.3	94.1	13.3	16.7	7.1	110K	22K	59K	60K	4K	31K
Br4	83.5	93.1	93.1	14.6	15.8	11.3	39K	11K	14K	23K	3K	8K
Br5	90.8	92.9	95.1	13.1	15.0	7.2	127K	83K	69K	68K	34K	41K
Br6	87.3	88.3	89.5	10.3	13.2	7.8	165K	105K	152K	81K	38K	105K
Br7	89.7	90.4	91.1	14.1	14.5	7.9	400K	333K	405K	248K	186K	218K
Br8	94.6	95.5	96.6	7.2	7.6	3.0	114K	90K	94K	59K	43K	55K

a prediction. This approach is infeasible for very long histories because of the latency of the packing logic. Instead, filtered TAGE maintains packed histories and uses checkpoints for recovery. Maintaining packed histories increases the effective history length with fewer bits, which can also improve the prediction accuracy.

To verify my hypothesis about the impact of packing on the history length, I collected per-branch statistics about the behavior of baseline TAGE, filtered TAGE with zeroing, and filtered TAGE with packing. Table 6.2 shows the results for some of the most improved branches of *leela*. The relevant statistic is *Mean Prediction Table ID*, which is the arithmetic mean of the table ID that is ultimately used to provide a prediction. Tables with shorter history lengths have a smaller ID, thus, the lower average prediction table ID of filtered TAGE is an indicator that TAGE is successfully using shorter history tables.

Table 6.2 also shows the change in TAGE allocation behavior due to filtering. *Number of Allocations* is the total number of allocation attempts, whether successful

Table 6.3: Prediction accuracy of baseline 64KB TAGE-SC-L and filtered TAGE-SC-L with packing filtering. Br1-Br8 are the same branches as the branches in Table 4.1.

	Baseline	Filtered Top 32	Filtered Top 16	Filtered Top 8	Filtered Top 4
Br1	77.92%	78.43%	78.72%	77.02%	76.68%
Br2	89.21%	90.45%	87.38%	83.54%	83.55%
Br3	76.06%	99.97%	99.92%	99.97%	99.94%
Br4	75.65%	76.38%	77.56%	95.15%	90.57%
Br5	66.85%	65.68%	61.51%	59.12%	58.29%
Br6	67.40%	66.18%	65.26%	63.82%	64.87%
Br7	89.21%	93.72%	92.99%	94.50%	91.08%
Br8	68.33%	68.57%	66.81%	65.51%	65.47%

or not, which is a measure of allocation pressure caused by each branch. *Number of Unique Entries* is the total number of unique allocation attempts as specified by a (table ID, index, tag) tuple, which is a measure of distinct history patterns observed by each branch. Both zeroing and packing consistently reduce allocation and the number of observed patterns, which is most likely the cause of improved accuracy.

6.3.2 Number of Correlated Branches

Table 6.3 shows the prediction accuracy of the baseline and filtered TAGE-SC-L for some of the hard-to-predict branches of SPEC benchmark *leela*. Br1-Br8 are the same branches that were used in Table 4.1. Filtered Top 32 refers to using a filtered history that contains 32 static correlated branches. The run with the best prediction is highlighted in green, orange, or red. Green identifies cases where filtering the history significantly improves the accuracy, orange identifies the cases where filtering somewhat improves the accuracy, and red identifies the cases where filtering does not help at all. Interestingly, the green branches are the same as the green branches in

Table 6.4: The selected number of correlated branches for each filtered history and the total number of combined correlated branches. Filtered TAGE supports up to a total of 16 filtered histories in this configuration.

	leela	xz	omnetpp	deepsjeng	mcf	x264	gcc	exchange2	perlbench	xalancbmk
Filtered histories with 4 correlated branches	7	4	4	4	3	3	5	6	4	1
Filtered histories with 8 correlated branches	3	7	5	4	6	3	3	5	4	2
Filtered histories with 16 correlated branches	2	3	4	4	3	4	2	2	2	0
Filtered histories with 32 correlated branches	4	2	3	4	3	6	3	3	0	2
Total number of correlated branches	113	129	153	163	94	218	186	137	54	60

Table 4.1, i.e., the branches that can be accurately predicted with filtered BranchNet are the same as branches that benefit the most from filtered TAGE. The reason is that for filtered TAGE to be effective, the number of correlated branches should be small enough such that TAGE could capture all the observable history patterns. For branches like Br4, having more correlated branches is not worth the extra allocation pressure caused by including those branches in the history.

Table 6.4 shows the selected number of correlated branches for filtered TAGE with up to 16 filtered histories. All choices of using 4, 8, 16, or 32 branches are deemed useful for some branches. The table also shows the total number of correlated branches that are selected across all filtered histories. The correlation map should ideally be large enough to support keeping an entry for each of the correlated branches. However, as explained in Section 6.2, if there is not enough capacity for all correlated branches in a given correlation map configuration, I will sort the branches according to some heuristic and drop the least likely needed correlated branches.

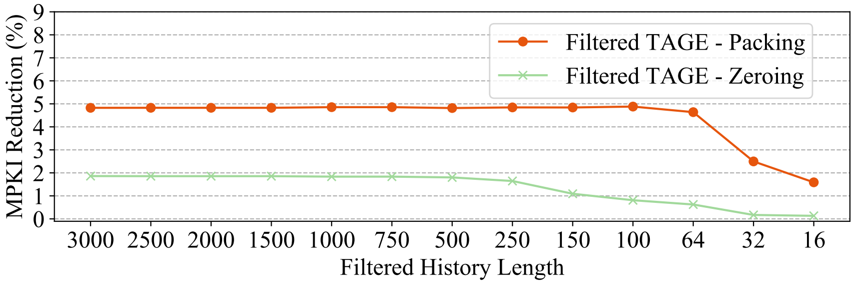


Figure 6.6: Impact of history length on MPKI reduction of filtered TAGE. Filtered TAGE is configured to support up to 32 filtered histories.

6.3.3 Sensitivity Studies

Figure 6.6 shows the impact of reducing the history length from 3000 bits (baseline 64KB TAGE-SC-L) down to 16. As mentioned above, when using packing filtered histories, most predictions are produced by short history tables, thus, filtered TAGE can tolerate much shorter history lengths without any significant loss of accuracy. Compared to using 3000 history bits, the first significant drop in MPKI reduction occurs when the history length is 64 bits. On the other hand, filtered TAGE with zeroing filtering is sensitive to the history length because long history tables are still needed for high accuracy (the first significant drop occurs when the history length is 500 bits).

Figure 6.7 shows the impact of the number of filtered histories. In general, most benchmarks only need up to 16 filtered histories. The exceptions are *deepsjeng* and *xz*, which can use up to 32 filtered histories to further improve the branch prediction accuracy. For the purpose of improving TAGE without significant storage overhead, using 24 filtered histories is a good design choice.

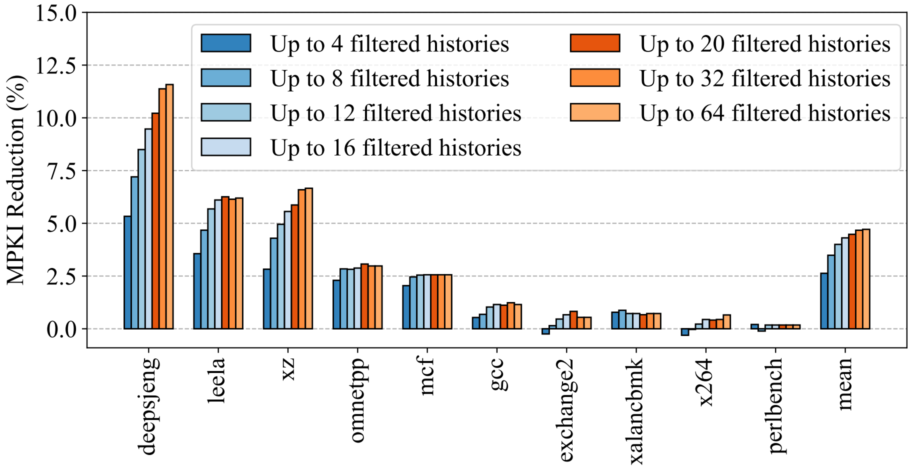


Figure 6.7: Impact of number of filtered histories on MPKI reduction of filtered TAGE. Filtered TAGE uses 250-bit packing histories.

Figure 6.8 shows that TAGE-SC-L storage can be reduced to 40KB while maintaining positive MPKI reduction. Since filtering reduces the allocation pressure on TAGE entries, a smaller TAGE with filtering can be more accurate than a larger TAGE without filtering. The main negative outliers are *gcc* and *exchange2*, where the improvement due to filtering is small, but the overall accuracy is sensitive to baseline TAGE-SC-L storage.

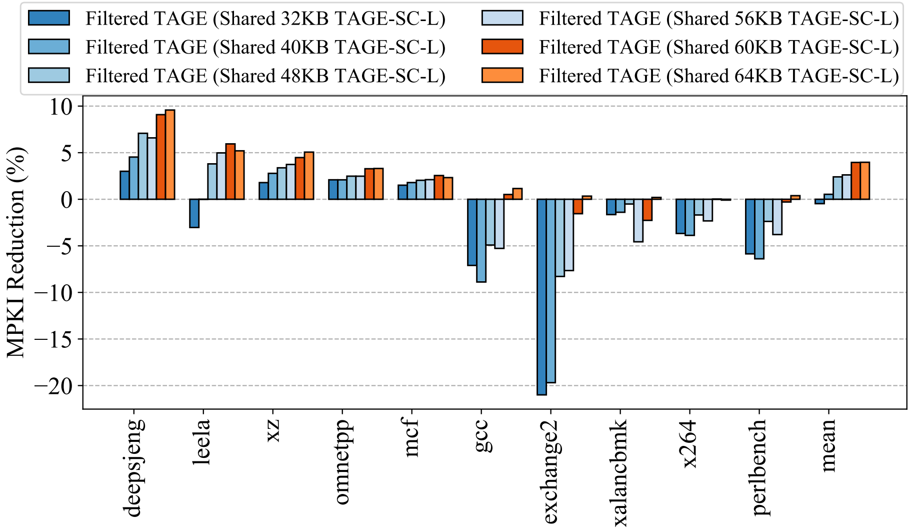


Figure 6.8: Impact of shrinking baseline TAGE-SC-L storage on filtered TAGE.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

BranchNet is a convolutional neural network that is trained offline to predict many branches that are fundamentally hard to predict for state-of-the-art predictors. State-of-the-art branch predictors fail to accurately predict these branches because they need exponentially large storage to identify branch correlations that appear deep into a noisy global history. In contrast, by using the abundant data and computation available during offline training, BranchNet learns to ignore uncorrelated noise in the history and uses only the correlated branches to make a prediction. After offline training, BranchNet can be used either directly as a helper predictor to augment state-of-the-art runtime predictors, or as a tool to explicitly identify correlated branches and filter the global branch history of TAGE. Both approaches result in improved overall prediction accuracy. This dissertation describes and evaluates both approaches and discusses the trade-offs.

To show the inherent advantage of CNNs in predicting this category of branches, I compared the prediction accuracy of Big-BranchNet to MTAGE-SC without considering practical constraints. Big-BranchNet outperforms MTAGE-SC on some of the most mispredicting branches among the SPEC2017 benchmarks, resulting in 7.6%

MPKI reduction. Furthermore, to show the effectiveness of CNNs as practical branch predictors, I compared Mini-BranchNet to 64KB TAGE-SC-L. Without increasing the prediction latency, Mini-BranchNet reduces the MPKI by 9.6%.

Using BranchNet as a tool to identify correlated branches enables filtered TAGE. Filtered TAGE uses the correlation information produced by BranchNet to maintain filtered histories that only contain the most useful branches to predict noisy branches. Because of eliminating redundant history patterns, the allocation pressure and the warmup time of TAGE are decreased. To show the effectiveness of using BranchNet to filter TAGE histories, I compared the accuracy of filtered TAGE to a 64KB TAGE-SC-L baseline. In an iso-latency configuration with 26KB storage overhead, filtered TAGE reduces the MPKI of SPEC 2017 Integer benchmarks by 3.7%, up to 9.4% for the most improved benchmark (0.9% speedup, up to 3.0%). To achieve the same order of performance benefits, an unfiltered TAGE-SC-L requires 128KB, which is 40% more than the size of the chosen filtered TAGE configuration, demonstrating the storage-efficiency of filtering.

While the IPC gains of using BranchNet in practical settings are limited (e.g., iso-latency Mini-BranchNet results in 1.3% average speed-up, up to 7.9% speed-up for the best benchmark), these results should not be interpreted as a limit to the potential benefits of deep learning for branch prediction. The key takeaway from BranchNet is that offline deep learning is a powerful approach to address the weaknesses of state-of-the-art runtime branch predictors. Further work can complement and enhance the key insights and observations of this dissertation.

7.2 Future Work

Using per-branch prediction models is a major performance bottleneck if the mispredictions of a program are distributed among many static branches. In this case, BranchNet cannot significantly improve its accuracy by improving the prediction accuracy of just a few branches. Even if we can train an accurate CNN model for each mispredicting branch, we need a large storage area to keep the models. One possible direction is to use the methodology of Predictor Virtualization [9] to maintain all the models in the main memory and use either a runtime mechanism or explicit BranchNet instructions to load the BranchNet models into the inference engine as needed. Another strategy is to use larger models that can predict multiple branches. This strategy is particularly interesting for filtering TAGE histories, as the sets of correlated branches for predicting noisy histories often have significant overlap. If correlated branch sets overlap, a shared filtered history is an effective way of filtering the global history for multiple noisy branches at the cost of one extra branch history.

It may be possible to reduce the large gap between the accuracy of Big-BranchNet and Mini-BranchNet. This dissertation argues that specialization is key for further reducing this gap. For example, a major source of inefficiency is identifying the relevant portion of the history that contains useful information. Fully-connected layers are not a storage-efficient solution. If the relevant region can be identified through other means (e.g., information extraction from a trained Big-BranchNet model), maybe a custom specialized hardware can replace the fully-connected neurons.

Perhaps the biggest weakness of BranchNet is predicting data-dependent branches. Despite this current weakness, I believe the combination of deep learning and offline training has the potential to further push branch prediction by using signals other than the global branch history that can help to predict data-dependent branches.

Finally, while this dissertation focuses on branch prediction, many key insights of observations are transferable to other domains. For example, a key insight of this dissertation is that machine learning models can be designed as tools for guiding other prediction mechanisms. It may be possible to use neural networks to learn data access patterns and use the information to adjust conventional runtime data prefetchers.

Bibliography

[1] "Branchnet," https://github.com/siavashzk/BranchNet.

[2] "Scarab," https://github.com/hpsresearchgroup/scarab.

[3] A. Adileh, D. Lilja, and L. Eeckhout, "Architectural support for probabilistic branches," in 51st annual IEEE/ACM International Symposium on Microarchitecture, Fukuoka, Japan, Oct 2018.

[4] Z. Allen-Zhu, Y. Li, and Z. Song, "A convergence theory for deep learning via over-parameterization," arXiv preprint arXiv:1811.03962, 2018.

[5] J. N. Amaral, E. Borin, D. R. Ashley, C. Benedicto, E. Colp, J. H. S. Hoffmam, M. Karpoff, E. Ochoa, M. Redshaw, and R. E. Rodrigues, "The alberta workloads for the spec cpu 2017 benchmark suite," in 2018 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), April 2018, pp. 159–168.

[6] D. W. Anderson, F. J. Sparacio, and R. M. Tomasulo, "The ibm system/360 model 91: Machine philosophy and instruction-handling," IBM Journal of Research and Development, vol. 11, no. 1, pp. 8–24, 1967.

[7] R. Bera, K. Kanellopoulos, A. Nori, T. Shahroodi, S. Subramoney, and O. Mutlu, "Pythia: A customizable hardware prefetching framework using

online reinforcement learning," in MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture, ser. MICRO '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 1121–1137. [Online]. Available: https://doi.org/10.1145/3466752.3480114

[8] H. D. Block, "The perceptron: A model for brain functioning. i," Rev. Mod. Phys., vol. 34, pp. 123–135, Jan 1962. [Online]. Available: https://link.aps.org/doi/10.1103/RevModPhys.34.123

[9] I. Burcea, S. Somogyi, A. Moshovos, and B. Falsafi, "Predictor virtualization," in Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems, ser. ASPLOS XIII. New York, NY, USA: Association for Computing Machinery, 2008, p. 157–167. [Online]. Available: https://doi.org/10.1145/1346281.1346301

[10] 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. [Online]. Available: http://doi.acm.org.ezproxy.lib.utexas.edu/10.1145/239912.239923

[11] P.-Y. Chang, M. Evers, and Y. Patt, "Improving branch prediction accuracy by reducing pattern history table interference," in Proceedings of the 1996 Conference on Parallel Architectures and Compilation Technique, 1996, pp. 48–57.

[12] R. S. Chappell, F. Tseng, A. Yoaz, and Y. N. Patt, "Difficult-path branch prediction using subordinate microthreads," in Proceedings 29th Annual International Symposium on Computer Architecture, May 2002, pp. 307–317.

[13] R. S. Chappell, J. Stark, S. P. Kim, S. K. Reinhardt, and Y. N. Patt, "Simultaneous subordinate microthreading (ssmt)," in Proceedings of the 26th Annual International Symposium on Computer Architecture, ser. ISCA '99. Washington, DC, USA: IEEE Computer Society, 1999, pp. 186–195. [Online]. Available: http://dx.doi.org/10.1145/300979.300995

[14] C. Chen, F. Tung, N. Vedula, and G. Mori, "Constraint-aware deep neural network compression," in Proceedings of the European Conference on Computer Vision (ECCV), 2018, pp. 400–415.

[15] I.-C. K. Chen, J. T. Coffey, and T. N. Mudge, "Analysis of branch prediction via data compression," SIGPLAN Not., vol. 31, no. 9, p. 128–137, sep 1996. [Online]. Available: https://doi.org/10.1145/248209.237171

[16] J. Cleary and I. Witten, "Data compression using adaptive coding and partial string matching," IEEE Transactions on Communications, vol. 32, no. 4, pp. 396–402, April 1984.

[17] M. Courbariaux, I. Hubara, D. Soudry, R. El-Yaniv, and Y. Bengio, "Binarized neural networks: Training deep neural networks with weights and activations constrained to +1 or -1," 2016.

[18] A. N. Eden and T. Mudge, "The yags branch prediction scheme," in Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture, ser. MICRO 31. Washington, DC, USA: IEEE Computer Society Press, 1998, p. 69–77.

[19] R. E.M. and F. C.C., "The inhibition of potential parallelism by conditional jumps," IEEE Transactions on Computers, vol. C-21, no. 12, pp. 1405–1411, 1972.

[20] M. Evers, S. J. Patel, R. S. Chappell, and Y. N. Patt, "An analysis of correlation and predictability: What makes two-level branch predictors work," in Proceedings of the 25th Annual International Symposium on Computer Architecture, ser. ISCA '98. USA: IEEE Computer Society, 1998, p. 52–61. [Online]. Available: https://doi.org/10.1145/279358.279368

[21] M. U. Farooq, Khubaib, and L. K. John, "Store-load-branch (slb) predictor: A compiler assisted branch prediction for data dependent branches," in 2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA), 2013, pp. 59–70.

[22] A. Fog, "Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for intel, amd and via cpus," Technical University of Denmark, Tech. Rep. [Online]. Available: https://www.agner.org/optimize/instruction_tables.pdf

[23] I. Goodfellow, Y. Bengio, and A. Courville, Deep Learning. MIT Press, 2016, http://www.deeplearningbook.org.

[24] D. Gope and M. H. Lipasti, "Bias-free branch predictor," in 2014 47th Annual IEEE/ACM International Symposium on Microarchitecture, Dec 2014, pp. 521–532.

[25] M. Hashemi, K. Swersky, J. Smith, G. Ayers, H. Litz, J. Chang, C. Kozyrakis, and P. Ranganathan, “Learning memory access patterns,” in Proceedings of the 35th International Conference on Machine Learning, ser. Proceedings of Machine Learning Research, J. Dy and A. Krause, Eds., vol. 80. Stockholmsmässan, Stockholm Sweden: PMLR, 10–15 Jul 2018, pp. 1919–1928. [Online]. Available: http://proceedings.mlr.press/v80/hashemi18a.html

[26] J. Hu, L. Shen, and G. Sun, “Squeeze-and-excitation networks,” 2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition, Jun 2018. [Online]. Available: http://dx.doi.org/10.1109/CVPR.2018.00745

[27] Q. Huang, K. Zhou, S. You, and U. Neumann, “Learning to prune filters in convolutional neural networks,” in 2018 IEEE Winter Conference on Applications of Computer Vision (WACV), 2018, pp. 709–718.

[28] S. Ioffe and C. Szegedy, “Batch normalization: Accelerating deep network training by reducing internal covariate shift,” in Proceedings of the 32Nd International Conference on International Conference on Machine Learning - Volume 37, ser. ICML’15. JMLR.org, 2015, pp. 448–456. [Online]. Available: http://dl.acm.org/citation.cfm?id=3045118.3045167

[29] E. Ipek, O. Mutlu, J. F. Martínez, and R. Caruana, “Self-optimizing memory controllers: A reinforcement learning approach,” in 2008 International Symposium on Computer Architecture, 2008, pp. 39–50.

[30] D. Jiménez, “Multiperspective perceptron predictor,” in 5th JILP Workshop on

Computer Architecture Competitions (JWAC-5): Championship Branch Prediction (CBP-5), 2016.

[31] D. Jiménez, “Multiperspective perceptron predictor with tage,” in 5th JILP Workshop on Computer Architecture Competitions (JWAC-5): Championship Branch Prediction (CBP-5), 2016.

[32] D. A. Jimenez, H. L. Hanson, and C. Lin, “Boolean formula-based branch prediction for future technologies,” in Proceedings 2001 International Conference on Parallel Architectures and Compilation Techniques, Sep. 2001, pp. 97–106.

[33] D. A. Jimenez, S. W. Keckler, and C. Lin, “The impact of delay on the design of branch predictors,” in Proceedings 33rd Annual IEEE/ACM International Symposium on Microarchitecture. MICRO-33 2000, Dec 2000, pp. 67–76.

[34] D. A. Jimenez and C. Lin, “Dynamic branch prediction with perceptrons,” in Proceedings HPCA Seventh International Symposium on High-Performance Computer Architecture, Jan 2001, pp. 197–206.

[35] D. Jimenez, “Fast path-based neural branch prediction,” in Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36., 2003, pp. 243–252.

[36] D. A. Jimenez, “Piecewise linear branch prediction,” in Proceedings of the 32Nd Annual International Symposium on Computer Architecture, ser. ISCA ’05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 382–393. [Online]. Available: https://doi.org/10.1109/ISCA.2005.40

[37] T. S. Karkhanis and J. E. Smith, “A first-order superscalar processor model,” in Proceedings. 31st Annual International Symposium on Computer Architecture, 2004., June 2004, pp. 338–349.

[38] Y. Kim, W. Yang, and O. Mutlu, “Ramulator: A fast and extensible dram simulator,” IEEE Computer Architecture Letters, vol. 15, no. 1, pp. 45–49, Jan 2016.

[39] D. P. Kingma and J. Ba, “Adam: A method for stochastic optimization,” CoRR, vol. abs/1412.6980, 2015.

[40] A. Krall, “Improving semi-static branch prediction by code replication,” SIGPLAN Not., vol. 29, no. 6, pp. 97–106, Jun. 1994. [Online]. Available: http://doi.acm.org/10.1145/773473.178252

[41] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel, “Backpropagation applied to handwritten zip code recognition,” Neural Comput., vol. 1, no. 4, pp. 541–551, Dec. 1989. [Online]. Available: http://dx.doi.org/10.1162/neco.1989.1.4.541

[42] C.-C. Lee, I.-C. K. Chen, and T. N. Mudge, “The bi-mode branch predictor,” Proceedings of 30th Annual International Symposium on Microarchitecture, pp. 4–13, 1997.

[43] J. K. F. Lee and A. J. Smith, “Branch prediction strategies and branch target buffer design,” Computer, vol. 17, no. 1, pp. 6–22, 1984.

[44] M.-S. Lee, Y.-J. Kang, J.-W. Lee, and S.-R. Maeng, “Opts: increasing branch prediction accuracy under context switch,” Microprocessors and Microsystems, vol. 26, no. 6, pp. 291 – 300, 2002. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0141933102000418

[45] C.-K. Lin and S. J. Tarsa, “Branch prediction is not a solved problem: Measurements, opportunities, and future directions,” in IEEE International Symposium on Workload Characterization, 2019.

[46] S. Mcfarling, “Combining branch predictors,” Digital Equipment Corporation, Western Research Lab, Tech. Rep., 1993.

[47] P. Michaud, “A ppm-like, tag-based branch predictor,” in In Proceedings of the First Workshop on Championship Branch Prediction (in conjunction with MICRO-37), 2004.

[48] P. Michaud, “An alternative tage-like conditional branch predictor,” ACM Trans. Archit. Code Optim., vol. 15, no. 3, Aug. 2018. [Online]. Available: https://doi.org/10.1145/3226098

[49] P. Michaud, A. Seznec, and S. Jourdan, “An exploration of instruction fetch requirement in out-of-order superscalar processors,” International Journal of Parallel Programming, vol. 29, no. 1, pp. 35–58, Feb 2001. [Online]. Available: https://doi.org/10.1023/A:1026431920605

[50] P. Michaud, A. Seznec, and R. Uhlig, “Trading conflict and capacity aliasing in conditional branch predictors,” SIGARCH Comput. Archit.

News, vol. 25, no. 2, p. 292–303, may 1997. [Online]. Available: https://doi.org/10.1145/384286.264211

[51] C. Molnar, G. Casalicchio, and B. Bischl, "Interpretable machine learning–a brief history, state-of-the-art and challenges," in Joint European Conference on Machine Learning and Knowledge Discovery in Databases. Springer, 2020, pp. 417–431.

[52] N. Muralimanohar, R. Balasubramonian, and N. Jouppi, "Optimizing nuca organizations and wiring alternatives for large caches with cacti 6.0," in 40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007), Dec 2007, pp. 3–14.

[53] V. Nair and G. E. Hinton, "Rectified linear units improve restricted boltzmann machines," in Proceedings of the 27th International Conference on International Conference on Machine Learning, ser. ICML'10. USA: Omnipress, 2010, pp. 807–814. [Online]. Available: http://dl.acm.org/citation.cfm?id=3104322.3104425

[54] J. R. C. Patterson, "Accurate static branch prediction by value range propagation," SIGPLAN Not., vol. 30, no. 6, pp. 67–78, Jun. 1995. [Online]. Available: http://doi.acm.org.ezproxy.lib.utexas.edu/10.1145/223428.207117

[55] S. Pruett and Y. Patt, "Branch runahead: An alternative to branch prediction for impossible to predict branches," in MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture, 2021, pp. 804–815.

[56] S. Pruett, S. Zangeneh, A. Fakhrzadehgan, B. Lin, and Y. Patt, "Dynamically sizing the tage branch predictor," in 5th JILP Workshop on Computer Architecture Competitions (JWAC-5): Championship Branch Prediction (CBP-5), 2016.

[57] H. Robbins and S. Monro, "A stochastic approximation method," Ann. Math. Statist., vol. 22, no. 3, pp. 400–407, 09 1951. [Online]. Available: https://doi.org/10.1214/aoms/1177729586

[58] A. Roth and G. S. Sohi, "Speculative data-driven multithreading," in Proceedings HPCA Seventh International Symposium on High-Performance Computer Architecture, Jan 2001, pp. 37–48.

[59] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, Learning Representations by Back-Propagating Errors. Cambridge, MA, USA: MIT Press, 1988, p. 696–699.

[60] Y. Sazeides, A. Moustakas, K. Constantinides, and M. Kleanthous, "The significance of affectors and affectees correlations for branch prediction," vol. 4917, 01 2008, pp. 243–257.

[61] H. Schorr et al., "Design principles for a high-performance system," in Proceedings of the Symposium on Computers and Automata, 1971.

[62] A. Seznec, "Analysis of the o-geometric history length branch predictor," in 32nd International Symposium on Computer Architecture (ISCA'05), June 2005, pp. 394–405.

[63] A. Seznec, “The o-gehl branch predictor,” in The 1st JILP Championship Branch Prediction (CBP-1), 2004.

[64] A. Seznec, “Exploring branch predictability limits with the MTAGE+SC predictor,” in 5th JILP Workshop on Computer Architecture Competitions (JWAC-5): Championship
Seoul, South Korea, Jun. 2016, p. 4. [Online]. Available: https://hal.inria.fr/hal-01354251

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

[66] A. Seznec and P. Michaud, “A case for (partially) tagged geometric history length branch prediction,” J. Instruction-Level Parallelism, vol. 8, 2006.

[67] A. Seznec, J. S. Miguel, and J. Albericio, “The inner most loop iteration counter: A new dimension in branch history,” in Proceedings of the 48th International Symposium on Microarchitecture, ser. MICRO-48. New York, NY, USA: ACM, 2015, pp. 347–357. [Online]. Available: http://doi.acm.org/10.1145/2830772.2830831

[68] R. Sheikh, J. Tuck, and E. Rotenberg, “Control-flow decoupling: An approach for timely, non-speculative branching,” IEEE Transactions on Computers, vol. 64, no. 8, pp. 2182–2203, Aug 2015.

[69] T. Sherwood and B. Calder, “Automated design of finite state machine predictors for customized processors,” in Proceedings 28th Annual International

Symposium on Computer Architecture, June 2001, pp. 86–97.

[70] 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: ACM, 2002, pp. 45–57. [Online]. Available: http://doi.acm.org/10.1145/605397.605403

[71] Z. Shi, X. Huang, A. Jain, and C. Lin, "Applying deep learning to the cache replacement problem," in Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, 2019, pp. 413–425.

[72] Z. Shi, A. Jain, K. Swersky, M. Hashemi, P. Ranganathan, and C. Lin, "A hierarchical neural model of data prefetching," in Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, ser. ASPLOS 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 861–873. [Online]. Available: https://doi.org/10.1145/3445814.3446752

[73] J. E. Smith, "A study of branch prediction strategies," in Proceedings of the 8th Annual Symposium on Computer Architecture, ser. ISCA '81. Washington, DC, USA: IEEE Computer Society Press, 1981, p. 135–148.

[74] E. Sprangle and D. Carmean, "Increasing processor performance by implementing deeper pipelines," in Computer Architecture, 2002. Proceedings. 29th Annual International Symposium on. IEEE, 2002, pp. 25–34.

[75] E. Sprangle, R. S. Chappell, M. Alsup, and Y. N. Patt, “The agree predictor: A mechanism for reducing negative branch history interference,” SIGARCH Comput. Archit. News, vol. 25, no. 2, p. 284–291, may 1997. [Online]. Available: https://doi.org/10.1145/384286.264210

[76] V. Srinivasan, R. B. R. Chowdhury, and E. Rotenberg, “Slipstream processors revisited: Exploiting branch sets,” in 2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA), 2020, pp. 105–117.

[77] J. Stark, M. Evers, and Y. N. Patt, “Variable length path branch prediction,” SIGOPS Oper. Syst. Rev., vol. 32, no. 5, pp. 170–179, Oct. 1998. [Online]. Available: http://doi.acm.org/10.1145/384265.291042

[78] K. Sundaramoorthy, Z. Purser, and E. Rotenburg, “Slipstream processors: Improving both performance and fault tolerance,” in Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems, ser. ASPLOS IX. New York, NY, USA: Association for Computing Machinery, 2000, p. 257–268. [Online]. Available: https://doi.org/10.1145/378993.379247

[79] C. Szegedy, S. Ioffe, V. Vanhoucke, and A. Alemi, “Inception-v4, inception-resnet and the impact of residual connections on learning,” 2016.

[80] D. Tarjan and K. Skadron, “Merging path and gshare indexing in perceptron branch prediction,” ACM Trans. Archit. Code Optim., vol. 2, no. 3, pp. 280–300, Sep. 2005. [Online]. Available: http://doi.acm.org/10.1145/1089008.1089011

[81] M. . Tarlescu, K. B. Theobald, and G. R. Gao, “Elastic history buffer: a low-cost method to improve branch prediction accuracy,” in Proceedings International Conference on Computer Design VLSI in Computers and Processors, Oct 1997, pp. 82–87.

[82] S. J. Tarsa, C.-K. Lin, G. Keskin, G. Chinya, and H. Wang, “Improving branch prediction by modeling global history with convolutional neural networks,” in The 2nd International Workshop on AI-assisted Design for Architecture, 2019.

[83] R. Thomas, M. Franklin, C. Wilkerson, and J. Stark, “Improving branch prediction by dynamic dataflow-based identification of correlated branches from a large global history,” in Proceedings of the 30th Annual International Symposium on Computer Architecture, ser. ISCA ’03. New York, NY, USA: Association for Computing Machinery, 2003, p. 314–323. [Online]. Available: https://doi.org/10.1145/859618.859655

[84] S. Verma, B. Maderazo, and D. M. Koppelman, “Spotlight - a low complexity highly accurate profile-based branch predictor,” in 2009 IEEE 28th International Performance Computing and Communications Conference, Dec 2009, pp. 239–247.

[85] K. Wang, Z. Liu, Y. Lin, J. Lin, and S. Han, “Haq: Hardware-aware automated quantization with mixed precision,” 2018.

[86] W. Wen, C. Wu, Y. Wang, Y. Chen, and H. Li, “Learning structured sparsity in deep neural networks,” in Advances in neural information processing systems, 2016, pp. 2074–2082.

[87] F. Wu, A. Fan, A. Baevski, Y. N. Dauphin, and M. Auli, “Pay less attention with lightweight and dynamic convolutions,” 2019.

[88] 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. New York, NY, USA: ACM, 1991, pp. 51–61.

[89] T.-Y. Yeh and Y. N. Patt, “Alternative implementations of two-level adaptive branch prediction,” SIGARCH Comput. Archit. News, vol. 20, no. 2, p. 124–134, apr 1992. [Online]. Available: https://doi.org/10.1145/146628.139709

[90] T.-Y. Yeh and Y. N. Patt, “A comparison of dynamic branch predictors that use two levels of branch history,” in Proceedings of the 20th Annual International Symposium on Computer Architecture, ser. ISCA ’93. New York, NY, USA: Association for Computing Machinery, 1993, p. 257–266. [Online]. Available: https://doi.org/10.1145/165123.165161

[91] C. Young and M. D. Smith, “Improving the accuracy of static branch prediction using branch correlation,” SIGOPS Oper. Syst. Rev., vol. 28, no. 5, pp. 232–241, Nov. 1994. [Online]. Available: http://doi.acm.org/10.1145/381792.195549

[92] 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), 2020, pp. 118–130.

[93] S. Zangeneh, S. Pruett, and Y. Patt, “Branch prediction with multi-layer neural networks: The value of specialization,” ML for Computer Architecture and Systems, 2020.

[94] C. B. Zilles and G. S. Sohi, “Understanding the backward slices of performance degrading instructions,” in Proceedings of 27th International Symposium on Computer Architecture (IEEE Cat. No.RS00201), June 2000, pp. 172–181.

[95] C. Zilles and G. Sohi, “Execution-based prediction using speculative slices,” SIGARCH Comput. Archit. News, vol. 29, no. 2, pp. 2–13, May 2001. [Online]. Available: http://doi.acm.org/10.1145/384285.379246