BranchNet: Using Offline Deep Learning To Predict Hard-To-Predict Branches

Siavash Zangeneh *Stephen Pruett* *Sangkug Lym* *Yale N. Patt*

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

TR-HPS-2019-002
August 2019

This page is intentionally left blank.

BranchNet: Using Offline Deep Learning To Predict Hard-To-Predict Branches

Siavash Zangeneh Stephen Pruett Sangkug Lym Yale N. Patt

Electrical and Computer Engineering
The University of Texas at Austin
{siavash.zangeneh,stephen.pruett,sklym,}@utexas.edu,
patt@ece.utexas.edu

Abstract

Conventional wisdom says that high-performance branch predictors only work if they can adapt to the run-time behavior of programs. In this paper, we show that this is not always true. The relationships between many branches are defined by program semantics and are invariant of program phase behavior. By taking advantage of large training datasets across input sets, one can train a branch predictor to learn these invariant branch relationships. As a first step towards high-performance branch predictors with offline training, we propose BranchNet, a convolutional neural network that can accurately predict many branches that are currently hard to predict for TAGE-SC-L, the best branch predictor today. At run time, we use a hybrid approach by combining TAGE-SC-L and BranchNet. We use BranchNet to predict a few hard-to-predict branches and TAGE-SC-L to predict the rest. Our hybrid approach, without area constraints, reduces the average branch mispredictions per kilo instructions (MPKI) of SPEC2017 Integer benchmarks by 6.2% (and up to 11.2% on the most improved benchmark) when compared to an unlimited MTAGE-SC baseline predictor. We also show that a latency and area constrained version of BranchNet reduces the average MPKI by 2.4% (up to 6.3%) compared to an iso-storage TAGE-SC-L baseline.

1 Introduction

Branch prediction accuracy remains a major bottleneck in improving the single-thread performance of applications. Even using TAGE-SCL, the best state-of-the-art branch predictor, many of SPEC2017 Integer benchmarks suffer from high branch mispredictions per kilo instructions (MPKI) (Figure 1), resulting in significant loss of IPC. In particular, *leela_r*, *mcf_r*, and *xz_r* are 52%, 42%, and 24% frontend-bound according to top-down analysis [56] on an Intel Skylake processor [33]. Moreover, the branch misprediction penalty worsens as processors move towards deeper and wider pipelines [26, 30, 21, 46]. Unfortunately, fundamental breakthroughs in branch prediction have become rare. All predictors entered in the 2016 Championship Branch Prediction competition were variants of existing TAGE and Perceptron designs [40, 39, 16, 17, 35]. Branch prediction research needs new insights to further improve the prediction accuracy.

Traditional branch predictors like TAGE and Perceptron are designed to be updated online, i.e., at run time. 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 run time, limitations in training time and

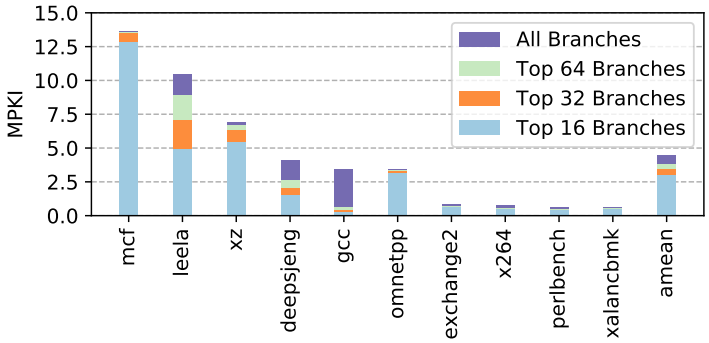

Figure 1. MPKI of 64KB TAGE-SC-L on SPEC2017 Integer Speed benchmarks.

processing power make it difficult for online branch predictors to learn complex correlations in the branch history. As a result, when the positions of branches in the branch history vary frequently, online predictors fail to distinguish truly correlated branches from noise in the branch history. Furthermore, since online predictors need to quickly adapt to program phases, their learning algorithms are heavily skewed towards the most recent instances of branch behavior. Bias towards recent run-time behavior prevents the predictors from learning branch correlations that are only visible across program phases or across different runs of a program. In this paper, we show the surprising result that online training is not always necessary, and one can overcome the limitations of online predictors through offline training.

Offline training (i.e., compile time) 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 run time, the branch predictor uses the trained models to predict the directions of these hard-to-predict branches without further training.

We propose BranchNet, a Convolutional Neural Network (CNN) that we train offline to accurately predict a subset of branches that are typically hard-to-predict by traditional branch predictors. CNNs have the ability to identify correlated features located anywhere in their inputs, which is immensely useful for branch prediction. In particular, CNNs are a good fit for branch prediction because they are naturally resilient against shifts in input patterns. Thus, BranchNet can learn to ignore uncorrelated noise in the branch history while identifying important features (correlated history patterns) anywhere in the global branch history.

However, sometimes offline training cannot find any invariant relationships that can predict a branch. Thus, BranchNet cannot be used as a complete replacement for traditional online branch predictors. Instead, we use

BranchNet only for the few static branches that benefit from offline training, and let a traditional online branch predictor predict the rest of the branches. A specialized predictor for a few static branches can have a significant positive impact on overall accuracy because only a few static branches are responsible for most branch mispredictions in a typical program. Figure 1 shows the branch MPKI of SPEC2017 Integer Speed benchmarks using 64KB TAGE-SC-L [40] as the branch predictor. Each bar shows the contribution to the total MPKI from the top 16, top 32, top 64, and rest of mispredicting branches. For all benchmarks except *gcc*, the top 32 mispredicting branches produce more than half of the total MPKI. Thus, by focusing on improving the accuracy of these few hard-to-predict branches, we can dramatically reduce the total MPKI of these benchmarks.

We are not the first to propose using CNNs as a solution for improving branch prediction. Tarsa et al. [53] have proposed an idea similar to BranchNet, to use CNNs as helper predictors for a few hard-to-predict branches. While we agree with their approach, BranchNet optimizes the network architecture to be specialized for branch prediction as opposed to general architectures used in image recognition tasks. This results in a higher prediction accuracy. The contributions of this paper are:

- We show that offline training can significantly improve the prediction accuracy of many currently hard-to-predict branches. Specifically, one can train a CNN offline by profiling a program with many input sets and use the trained CNN to predict the branches of the same program with previously unseen input data. Offline deep learning opens the door to using data and compute intensive learning models to improve branch prediction accuracy.

- We show that the ability of convolutional neural networks to identify spatial correlated patterns in branch history enables CNNs to be powerful branch predictors. A CNN can accurately predict many branches that are categorically hard to predict for conventional online branch predictors even with unlimited storage budgets.

- We propose BranchNet, a 4-layer CNN tailored to branch prediction requirements in two ways. (1) BranchNet draws inspiration from traditional branch predictors and uses geometric history lengths as inputs. (2) BranchNet exploits the sparsity of important branches in the branch history to aggressively down-sample convolutional outputs. As a proof-of-concept, we propose a practical on-chip inference engine that can improve the prediction accuracy of high MPKI benchmarks with the same storage budget as state-of-the-art predictors.

In the rest of the paper, we first motivate why CNNs with offline training can overcome the key limitation of prior work. We then describe the architecture of BranchNet, the process to train BranchNet offline, the design of

an inference engine to make predictions at run time, and the ISA/OS support needed to use BranchNet. We show that without area constraints, BranchNet reduces the average MPKI of SPEC2017 Integer benchmarks by 6.2% (up to 11.2% for the most improved benchmark) when compared to an unlimited MTAGE-SC baseline. We also compare BranchNet to TAGE-SC-L with a 64KB storage budget and show that using BranchNet can still be a win (up to 6.3% MPKI reduction) for high-MPKI programs.

These results, however, should not be interpreted as a limit to the potential benefits of offline deep learning. BranchNet is simply a first step towards practical branch predictors using deep neural networks. This approach opens the door to using many other sophisticated deep learning algorithms that may use inputs other than branch history. We expect that with further optimizations and innovations, offline deep learning for branch prediction can become more effective and a viable complement to traditional online predictors.

2 Limitations of Prior Work

State-of-the-art online branch predictors need to adapt quickly to program runtime behavior, while meeting the design constraints of an out-of-order processor. Thus, they rely on simple learning mechanisms that cannot capture complex branch relationships among large branch histories. In this section, we explain the limitations of online state-of-the-art branch predictors. We then explain why prior proposals for using offline training do not address the main limitations of state-of-the-art predictors.

2.1 State-of-the-Art Online Branch Predictors

The main input to modern branch predictors is the global branch and path history. Prior work shows that only a few branches in the history are truly important for branch prediction, while the other branches in the history are don't care terms that can be taken or not taken without affecting the target branch outcome [7]. Unfortunately, identifying the few relevant branches in the global history is difficult. Furthermore, branches in the history are constantly shifting positions, which exacerbates the problem. Thus, a key challenge for all branch predictors is figuring out how to efficiently account for the noise from uncorrelated and constantly shifting branches in the history. Since all modern branch predictors tend to be variants of TAGE and Perceptron, we will study the negative effects of noisy history on these two predictors.

TAGE. TAGE-SC-L is the winner of Championship Branch Prediction 2016 [40]. The main component of the predictor is TAGE [41], which uses the intuition behind the PPM compression algorithm [5] to learn an efficient mapping from global branch history to target branch outcome. TAGE is implemented using a collection of partially-tagged tables of saturating counters. Each table is indexed and tagged by hashes of target branch PC,

branch history, and path. The lengths of the branch histories form a geometric series, with the intuition that each branch should only use longer histories when shorter histories fail to make accurate predictions.

Each uncorrelated branch in the history doubles the number of possible history patterns that can lead to a branch since the uncorrelated branch can be either taken or not taken in the history. Non-deterministic positions of both correlated and uncorrelated branches further increase the number of possible history patterns. Thus, TAGE requires a lot of storage to learn a mapping for all possible history patterns. PPM-like compression does not help much when the important correlated branches are deep into a noisy history. In extremely noisy histories, TAGE acts as a global 2-level predictor [57], which requires $O(2^n)$ table entries for an n-bit history input, which is infeasible.

Perceptron. The Perceptron branch predictor [20] uses a single-layer neural network to predict the direction of branches. Perceptron learns the linear correlation of the target branch outcome to the directions of branches in the global history. Since Perceptron learns a correlation factor for each branch in the history, it is very efficient in ignoring uncorrelated branches, assuming the positions of branches remain constant. In practice, shifts in the history occur and cause branches to have non-deterministic positions in the global history. The hashed perceptron predictor [50] improves Perceptron by (1) including path history as input and (2) learning linear weights for multiple hashes of the global history as a whole. However, using hashes of the global history as input re-introduces the same challenge that TAGE-like predictors face; the number of weights needed to learn the behavior of a noisy history grows exponentially with the size of the history. Therefore, while this approach is generally efficient in predicting branches with stable history patterns, it is not useful for learning noisy histories. The Statistical Corrector in TAGE-SC-L [40] and Multiperspective Perceptron [16] use hashed perceptron predictors.

Another fundamental limitation of Perceptron is that it can only learn linearly-separable functions of their inputs. More sophisticated neural networks (deep neural networks) can learn more complicated (nonlinear) relationships in branch history. However, training a deep neural network is computationally demanding and not feasible for online branch predictors. In this paper, we take a different approach and show that we can use deep neural networks through offline training.

2.2 Branch Predictors with Offline Profiling

Prior Predictors. Many prior studies propose to use offline profiling to improve branch prediction. Some train static predictors that simply learn the general bias of branches, which is useful for compile-time optimizations, but not for predicting hard-to-predict branches [23, 4, 34, 58]. Some prior works use profiling to train predictors that were comparable to their contemporary dynamic branch predictors in terms of overall accuracy [18, 43, 54, 51, 48]. For example, Verma et al. propose Spotlight [54], a gshare-like predictor [29], that uses profiling to identify the

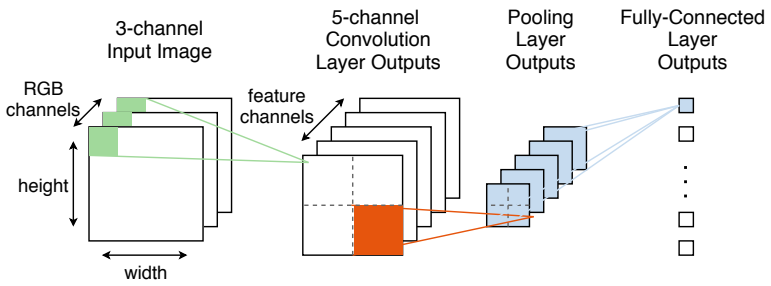

Figure 2. A Simple CNN image classifier.

branches that matter the most in a stable view of history. However, Spotlight is still susceptible to shifts in the history and cannot identify correlated branches that appear in random positions in the history. Their learning mechanisms also relies on exhaustively comparing all different configurations, which does not scale when training more complicated predictors with long histories. Similar to Spotlight, other prior works are also too simple to help with hard-to-predict branches and cannot be easily adapted to work in condjunction with state-of-the-art dynamic predictors. CNNs, however, are designed to take advantage of large training datasets to extract correlated features out of large noisy inputs, which makes them very effective in predicting a category of branches unpredictable by any prior works.

Profiling Methodology. Traditionally, offline profiling has been limited by the ability of software programmers to identify inputs that are representative of the real world. However, with the rise of cloud computing and large-scale deployment of software in data centers, the process of profiling and software-tuning could be more easily automated. In such environments, programst are deployed on machines under the control of the developers and run on hundreds of thousands of real input sets. This new model of computation, which was not widely available when most of the prior works were introduced, creates new opportunities for offline training. Programmers are no longer directly responsible for profiling their programs because programs can be monitored using automated frameworks. This dramatic shift in the computational model greatly bolsters offline profiling for improving performance and power efficiency. This new opportunity for reinvigorating profling methods is highlighted in a few recent studies that rely on profiling in data center environments [53, 52, 47].

3 Convolutional Neural Networks

Convolutional Neural Networks (CNN) are state-of-the-art in both image classification [49, 14] and sequential tasks like natural language understanding [55]. A typical CNN is made of a sequence of operations (*layers*) that gradually extract important features from its input and make a prediction about the nature of the input (e.g. image

classification). The layers operate using a number of parameters that are trained using deep learning algorithms. The collection of the CNN layers and their trained parameters form a *CNN model*. In this section, we define the terminology used in the paper and provide a high-level overview of CNNs and draw connections to branch prediction.

3.1 CNN Building Blocks

Figure 2 shows a simple CNN model for classification of 2D images. All layers except the final fully-connected layer operate on 3-dimensional arrays. Two dimensions correspond to the width and height of the input image and its identified features. The third dimension represents *channels*, which quantify the occurrence of a feature. Channels enable a CNN to learn multiple features of inputs. Finally, the fully-connected layers classify the input using the features identified by all the channels.

Convolutional Layers. At a high level, a convolution layer identifies the occurrences of features in its input [24, 10]. For each output channel, the layer learns a collection of weights (i.e., a *filter*) that describe the feature that the channel represents. For example, in image classification, if a filter describes a circle, its output channel quantifies the presence of circles at the given position in the image. For branch prediction, features are correlated branch patterns.

Pooling Layers. A pooling layer combines the neighboring outputs of a convolution layer to a single value through a reduction operation like maximum, average, or addition [10]. Effectively, it down-samples the convolution outputs to reduce the computational costs of the model at the expense of losing fine-grained positional information of features identified by the preceding convolution layer.

Fully-connected Layers. A fully-connected layer is made of multiple neurons, where each neuron learns a linear function of all its inputs [10]. In a CNN, fully-connected layers are used to combine the features identified by the convolution layers to make the final prediction about the input image. It is possible to cascade fully-connected layers, together with activation functions, to learn nonlinear functions of convolution outputs.

Activation Functions. Activation functions are non-linear element-wise transformations that are inserted after convolution and fully-connected layers. Without activation functions, neural networks cannot learn non-linear functions. We use ReLU [32], Sigmoid and Tanh (hyperbolic tangent) activations for BranchNet.

Batch Normalization Layers. A batch normalization layer normalizes each output channel to a standard normal distribution using the mean and the variance of its outputs during training [15]. While the normalization layer does not directly add to the prediction capability of a neural network, it has been shown to guide the optimization algorithms towards better solutions and mitigates overfitting. We use a normalization layer before a layer of

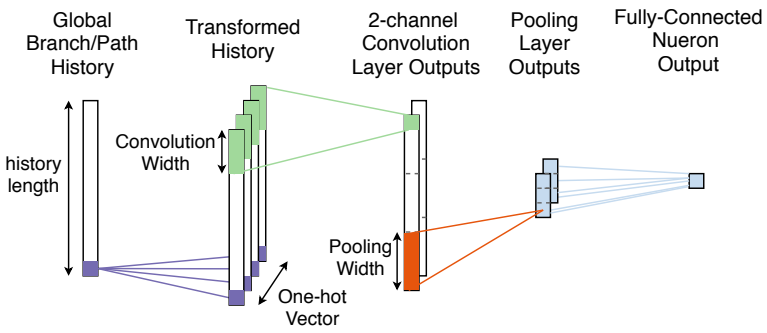

Figure 3. A simple CNN branch predictor.

activation functions.

One-hot Transformations. 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 program counter hashes as one-hot vectors. However, this approach does not perform well with large hash widths. For example, a 12-bit program counter hash is transformed into a $2^{12} = 4096$ bit vector, which is expensive.

Embeddings. Similar to one-hot vectors, embeddings transform a discrete number into a multi-dimensional vector. The difference is that embeddings are trained to represent the inputs in a dense, compressed space according to the needs of the problem that a model is trying to solve. Embeddings are made of adjustable parameters that could be trained along with the rest of a neural network. For example, Hashemi et al. [12] have used embeddings to represent PC and memory addresses in a model for data prefetching. Similarly, we use embeddings to feed a branch and path history into a neural network and let the training algorithm find an effective but dense representation of branches.

3.2 Training CNNs with Supervised Learning

Supervised learning is a method used to train machine learning models using a large set of input and expected output pairs (*the training set*) that define the desired behavior of the model. Supervised learning trains a model by iterating through the training set, computing the current outputs of the model and comparing them to the expected outputs. The model parameters are adjusted to decrease the output errors using Backpropagation [37] and a variant of the Stochastic Gradient Descent optimization algorithm [36]. This process is repeated until the prediction error of the model is minimized for the inputs in the training set.

```
1  int x = 0;
2  for (int i = 0; i < N; ++i) {
3    if (random_condition()) { // Branch A
4      // x increments if Branch A is not taken
5      x += 1;
6    }
7  }
8
9  for (int j = 0; j < x; ++j) { // Branch B
10   ...
11 } // exits when Branch B is taken
```

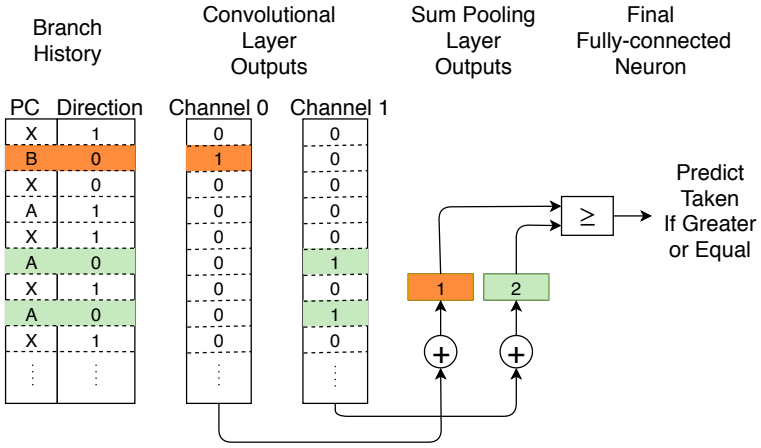

Figure 4. A program with a hard-to-predict branch (Branch B) and a trained CNN that can accurately predict the branch.

4 CNNs for Branch Prediction

CNNs are a natural fit for branch prediction because of their ability to identify correlated spatial features in large inputs. Convolutional layers search the global branch and path history to identify correlated branch patterns. Fully-connected layers then use the identified features to make a prediction about the direction of the next branch. Since the expected number of correlated branch patterns in the history is small [7], pooling layers can aggressively down-sample the convolution outputs to reduce the computational costs of the CNN.

Figure 3 shows a simple CNN branch predictor. The input is a history of branch direction and program counters. We transform each PC and direction pair in the history to a one-hot vector.[1] The convolutional layer has two channels, each identifying the occurrences of a given feature (a particular sequence of branch PCs and directions). The number of branches in the feature is determined by *the width* of the convolution filter. The pooling layer then

[1]As Section 3.1 explains, one can replace one-hot transformations with trainable embeddings for a more efficient representation of branches.

performs down-sampling by summing up convolution outputs in sequential windows. The size of each pooling window is determined by *the pooling width*. Finally, a single fully-connected neuron uses the down-sampled convolution outputs to make a final prediction.

4.1 Why Is Extracting Features From Long Histories Useful?

We use the program fragment in Figure 4 to show how CNNs can use extracted features in the history to predict branches. Figure 4 is a simplified version of a hot segment of the benchmark *leela*. The first loop sets the value of variable x by counting the occurrences of some probabilistic condition. The second loop iterates for as many times as the number of true conditions in the first loop. Because of the randomness of x, the exit branch of the second loop (Branch B) is hard to predict for traditional branch predictors. Perceptron and TAGE can predict branch B with an accuracy of 83% and 90% respectively.

However, there is a clear way to determine the direction of Branch B. Branch B is only taken if the variable j is equal to variable x. A CNN can infer the values of both variables from the global branch history: j equals the number of not taken instances of branch B in the history, and x equals the number of not taken instances of Branch A. Figure 4 shows the outputs of a manually trained CNN to predict the direction of Branch B. The convolution width is 1, and the pooling width is the same as history size. The branch history consists of pairs of branch program counters and directions that precede an instance of Branch B. The program counters of branches that are not involved in the prediction (i.e. uncorrelated branches) are marked as X. The one-hot transformation is not shown for brevity. Convolution channel 0 identifies the not taken instances of branch B using binary outputs. Convolution Channel 1 identifies the not taken instances of branch A. The pooling layer simply adds up the occurrences of identified branches for each channel. Thus, the down-sampled output of convolution channels represent the variables j and x. The final fully-connected neuron acts as a comparator and predicts taken only if $j \geq x$, resulting in 100% prediction accuracy.

Traditional branch predictors like TAGE and Perceptron cannot extract the values of j and x from the global history. They have to somehow make a prediction based on the global history as one unit. Because of the noisy and probabilistic nature of branches in the history, there are simply too many possible global history patterns to remember for any predictor that uses hashes of the history as input. In our example, all the uncorrelated branches marked as X can have arbitrary PCs and directions and can be randomly inserted/deleted in the history. Even if a TAGE-like predictor has enough storage to remember all history patterns it sees, the TAGE-like predictor will take a long time to warm up and can never generalize its predictions to history patterns it has not seen.

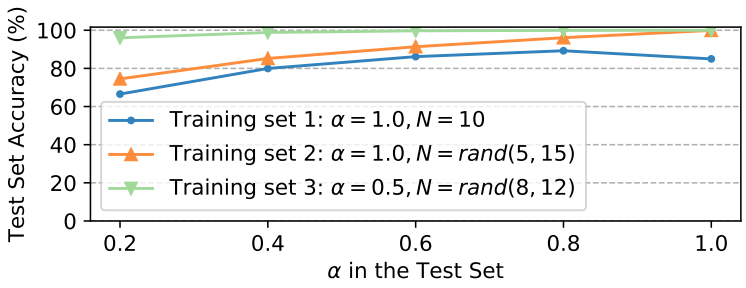

Figure 5. Accuracy of the CNN predictor to predict Branch B of Figure 4 using offline training.

4.2 When does offline training work?

Offline training only works if there exist persistent branch relationships that are independent of input data and program phase behavior. Sometimes there is no branch in the global history that can provide any information about the outcome of the target branch. For example, some branches depend on data which 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, we defer to the baseline online branch predictor to predict these branches.

When invariantly correlated branches in the global history exist, the challenge is to correctly identify these correlations without overfitting. Using the code in Figure 4 as an example again, if the value of x happened to always be 3 in all the training examples, the CNN learns that the length of the second loop is 3, which is not always true.

To illustrate the importance of training with a diverse set of inputs, suppose the probability of the random condition in line 3 of Figure 4 is controlled by variable α. We collected three different training sets for Branch B with three program inputs: (1) $N = 10$, $\alpha = 1$, (2) $N = rand(5, 15)$, $\alpha = 1$, and (3) $N = rand(8, 12)$, $\alpha = 0.5$. We then evaluated the accuracy of CNNs trained on each of the three training sets on runs of the program with $N = rand(5, 15)$ and α ranging from 0.2 to 1. Figure 5 shows the results. We see that CNNs trained using sets (1) and (2) perform poorly when $\alpha < 1$. However, the CNN trained with set (3) can predict Branch B with almost 100% accuracy for runs with any value of α.

Although the range of N in training set 3 is smaller than the range of N on evaluation runs, the trained model still generalizes almost perfectly to runs with different inputs. Thus, offline training does not require the training inputs to exactly represent the future branch behavior of a program. The training inputs just need to provide enough coverage that a CNN can learn invariant input-independent branch correlations.

4.3 Can Other Machine Learning Models Predict Branches?

Offline training opens the door to any learning model that can learn invariant branch relationships from large training sets. For example, the inherently sequential nature of branch history makes Recurrent Neural Networks, and in particular, LSTMs [13] a natural fit for branch prediction. Shi et al. [45] have studied learning dynamic execution using Gated Graph Neural Networks [25], which shows another direction for improving branch prediction. However, we limit the scope of this paper to the study of CNN for branch prediction because we see a clearer path towards a practical branch predictor with CNNs. We leave the study of the trade-offs of CNNs and other learning models (e.g. LSTMs) for future work.

5 BranchNet

Having described the general principles behind using CNNs for branch prediction, we now present Big-BranchNet and Mini-BranchNet. Both variants of BranchNet are 4-layer CNN models that we train offline to accurately predict many branches that are hard to predict for traditional branch predictors. We use Big-BranchNet to show available headroom in using CNNs for branch prediction. Big-BranchNet does not have a practical on-chip inference engine. Mini-BranchNet is a smaller model co-designed with a practical inference engine.

BranchNet is able to predict hard-to-predict branches by searching a very large branch and path history for correlated branch patterns. Offline training is key in empowering BranchNet to find correlated branch patterns deep into the history. Without offline training, branch predictors are limited to use very simple update algorithms, e.g., memorizing predictions for history patterns, which do not work well for noisy history.

Offline training, however, requires abundant computation resources and large training data. Thus, offline training is only viable when (1) one can collect representative training data, and (2) the training cost is justified by the benefits of improved execution latency over repeated usage. Thus, BranchNet aims to improve branch prediction in large-scale environments (e.g. data centers), where a program is deployed for usage over days or weeks of usage. In such environments, programmers are not necessarily directly responsible for profiling because programs can be monitored and profiled using automated software infrastructure.

In this section, we describe the offline training process, Big-BranchNet CNN model, Mini-BranchNet CNN model, Mini-BranchNet inference engine, and ISA modifications and OS support needed to use BranchNet at run time.

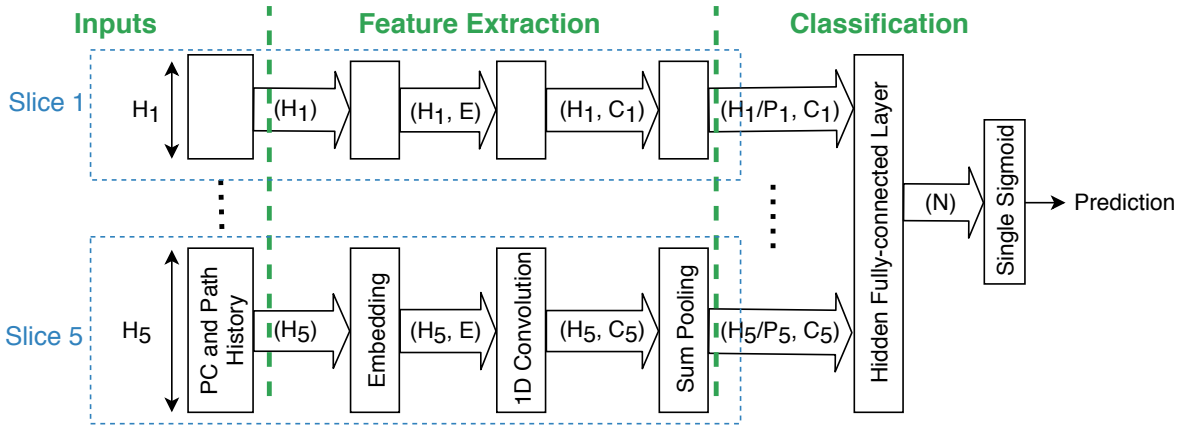

Figure 6. High-level diagram of BranchNet CNN architecture for one branch.

5.1 Offline Training Process

We profile the target program with a diverse set of inputs to collect branch traces. We divide these traces into three mutually exclusive sets: the training set, the validation set, and the test set. We then train BranchNet using the training set and the validation in a 3-step process. First, we select the highest MPKI branches (hard-to-predict branches) in the validation set. [2] Then, we train one CNN model for each hard-to-predict branch using the training set. Finally, we estimate the MPKI reduction of each branch on the validation set and attach the BranchNet models for the most improved branches to the program binary. To measure the final accuracy of on unseen inputs, we report the accuracy of BranchNet on the test set.

5.2 Big-BranchNet

Big-BranchNet is a variant of BranchNet that we use to show the available headroom in using CNNs for branch prediction. Big-BranchNet is a pure software model and we do not propose using it as a practical branch predictor. Figure 6 shows a high-level view of Big-BranchNet architecture: it is composed of 5 feature extraction sub-networks and two fully-connected layers. We call each feature extraction sub-network a slice. [3] Each slice uses an embedding layer, a convolution layer, and a pooling layer to extract features out of 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 [38]. Finally, the outputs of the slices are

[2] In this paper, we compute branch MPKI by simulating TAGE-SC-L, the best-known branch predictor today. In practice, one should attempt to estimate the MPKI on the target machine instead.

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

concatenated and fed into two sequential fully-connected layers. The fully-connected layers learn a nonlinear function of the identified features to make a prediction (taken or not taken).

We define Big-BranchNet in terms of a set of architecture knobs. We explain the operation of all Big-BranchNet layers using these architecture knobs and report the knob values that we used for Big-BranchNet in Table 1.

Table 1. BranchNet archtiecture knobs.

Knob	Big-BranchNet	Mini-BranchNet
H: history sizes	42, 78, 150, 294, 582	39, 79, 154, 304, 604
C: Convolution channels	32, 32, 32, 32, 32	2, 3, 4, 4, 4
P: Pooling widths	3, 6, 12, 24, 48	7, 15, 30, 60, 120
p: branch PC width	12	N/A
h: convolution branches hash width	N/A	8
E: Embedding dimensions	32	32
K: Convolution width	7	5
N: Hidden fully-connected neurons	32	6

Input History Format. We concatenate the direction and the least significant bits of the program counter of each branch to represent it as an integer. Thus, if we use p bits of PC, and a history size of H for a slice, the input history is 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 continuous numbers. This transformation allows the next layers to more easily distinguish different branches in the history. Note that we could have instead used one-hot encoding to transform each branch into a 2^{p+1}-dimensional vector. But, the dimensions of one-hot vectors grow exponentially with p. Embeddings, on the other hand, produce a vector with a tunable size (controlled by knob E). In our experiments, using embeddings was as accurate as one-hot vectors with the benefit of faster convergence and training time.

Convolutional Layers. C_i denotes the number of output channels and K denotes the convolution width. With more output channels, BranchNet can learn more independent features of branch history. With larger K, BranchNet can identify longer sequences of correlated branches. We always use a convolution with a stride of 1. The convolution layer is followed by batch normalization and ReLU activations.

Pooling Layers. In each slice, a pooling layer down-samples the convolution outputs using a sum pooling operator with a width and stride of P_i. We use geometric pooling sizes proportional to the history lengths of each slice. Larger pooling widths for larger history lengths work well because history becomes noisier deeper into the history. Aggressive pooling for features found deep into the history makes BranchNet resilient against shifts in history by eliminating fine-grained positions of the identified features in the history.

Fully-connected Layers. The first fully-connected layer consists of N neurons. Each neuron is connected

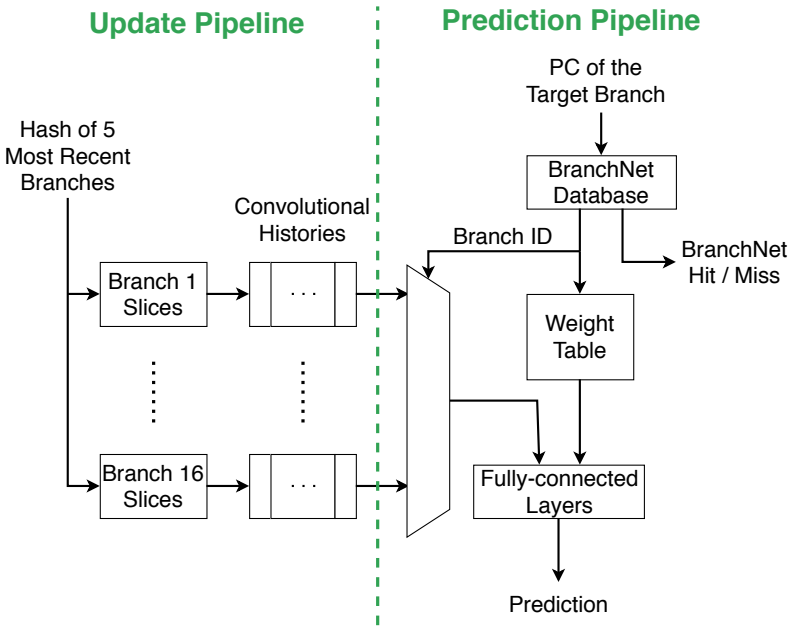

Figure 7. Mini-BranchNet inference engine.

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. BranchNet predicts a branch to be taken if the Sigmoid output is greater than 0.5.

5.3 Mini-BranchNet

Mini-BranchNet is a smaller variant of BranchNet that we co-designed with an inference engine that works as a practical branch predictor. For the most part, Mini-BranchNet is similar to a Big-BranchNet with architecture knobs that we tuned to minimize area and latency overheads. Table 1 reports the architecture knobs of Mini-BranchNet. In the rest of this subsection, we describe key optimizations in designing an inference engine for Mini-BranchNet. We will explain other modifications to BranchNet CNN architecture as they pertain to the inference engine optimizations.

Optimization 1: Maintaining Convolutional Histories. Computing the outputs of BranchNet slices (feature extraction layers) involves operations on hundreds of branches in the global history. Instead of doing all these operations at the time of prediction, the inference engine processes incoming branches one at a time and buffers their down-sampled convolution outputs for future use. We call these buffers Convolutional Histories. Figure 7 shows the block diagram of a Mini-BranchNet inference engine that can predict up to 16 static branches in a program. The update pipeline maintains the convolutional histories of all 16 Mini-BranchNet models. To make

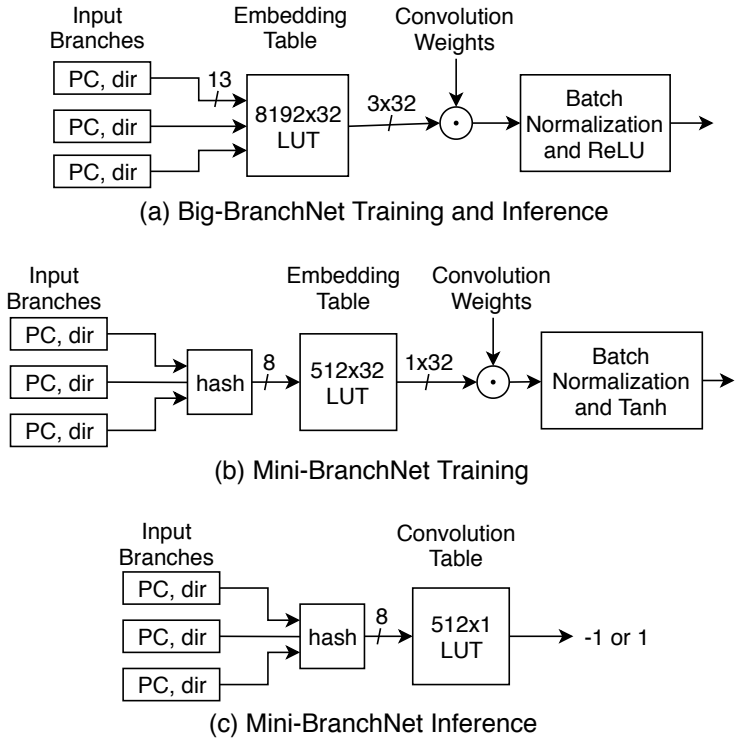

Figure 8. BranchNet 3-wide convolutional layer.

a prediction, the prediction pipeline simply selects the convolutional histories corresponding to the target branch and computes only the two fully-connected layers.

Optimization 2: Replacing Convolutions with Table Lookups. A convolution operation on a single window of branches involves a dot product operation. Figure 8a shows how BranchNet computes one convolution output for a 3-wide window ($K = 3$). Mini-BranchNet eliminates all the arithmetic operation in two steps. During training, instead of embedding each branch in the convolution window independently, it embeds a smaller hash of the 3 branches (Figure 8b) and uses Tanh activations instead of ReLU. Tanh activations bind the outputs to be between -1 and 1. After training is done, for each possible branch hash, we compute the convolution output (embedding + dot product + normalization + Tanh) and quantize the output to a binary value (-1 or 1).[4] These binary values can now be stored in small tables that the Mini-BranchNet Inference engine looks up to get the convolution output for a branch hash (Figure 8c). No arithmetic operation is needed at run time.

[4]Because of loss of precision, the fully-connected layers need to be retrained after quantization of convolution outputs.

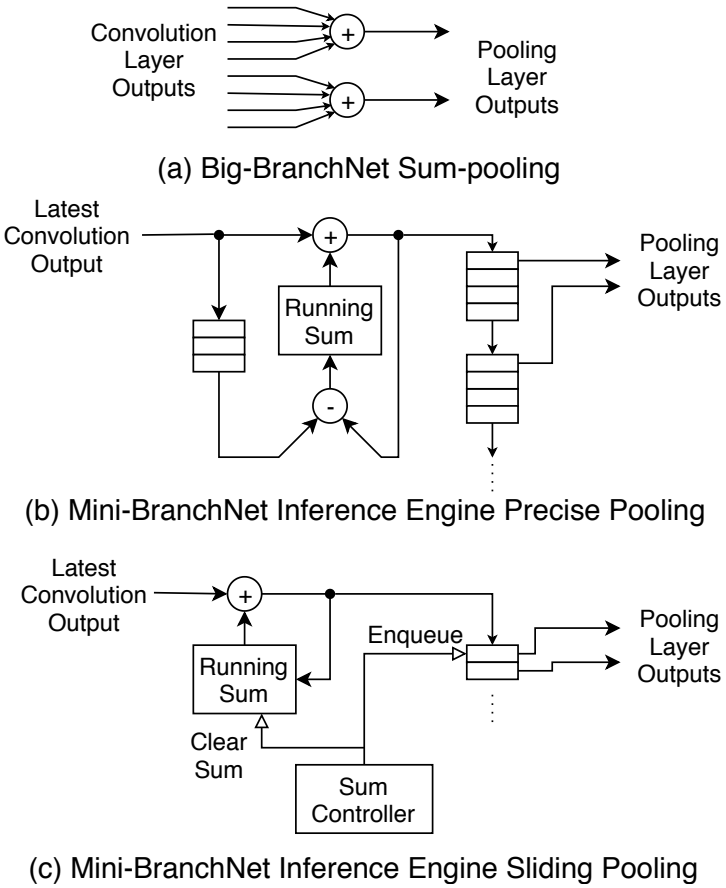

Figure 9. BranchNet 4-wide sum-pooling.

Optimization 3: Using Running Sum Registers. Figure 9a shows the sum-pooling operation of Big-BranchNet. Mini-BranchNet inference engine uses two design to compute the sum pooling outputs. For shorter history slices, the engine implements *precise pooling* (Figure 9b). 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 of buffers contains the pooling outputs of overlapping windows. At prediction time, only 1 out of P pooling outputs (recall $P = pooling\ window$) are fed into the next layer. The buffer space needed to implement sliding pooling grows linearly with the history size. Thus, for longer history slices, Mini-BranchNet inference engine uses *sliding pooling* (Figure 9c) which does not need as much buffer space. 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 we only use sliding poolings the three slices of Mini-BranchNet with the highest history lengths, which do not rely on fine-grained positions of identified features. To account for sliding poolings during training, we randomly discard some of the most recent branches (0 to $P-1$ branches) that are fed into the longer history slices. This randomization makes the training algorithm resilient against nondeterministic pooling boundaries.

Optimization 4: Quantizing Fully-connected Layers. Mini-BranchNet uses fixed-point arithmetic to compute the outputs of the fully-connected layers. We use 3 bits of precision for sum-pooling outputs, 4 bits for the first fully-connected weights, and 2 bits (ternary values of -1, 0, and 1) for the first fully-connected outputs and the final fully-connected weights. Again, we replace ReLU activations with Tanh to restrict the layer outputs between -1 and 1, which helps with quantization. We also insert batch normalization and Tanh after the sum-pooling layer to stabilize the inputs to the fully-connected layers. After training is done, we fuse the batch normalization operations with the fully-connected dot products to eliminate their latency.

5.4 Mini-BranchNet Inference Engine Area and Latency

Modern processors have two levels 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 [19]. We envision BranchNet to be 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[31], we computed the update latency to be 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[9], we estimate that mini-BranchNet updates are also single-cycle operations. The critical path of the prediction pipeline 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, an 85-input 6-bit adder tree, a 2-bit multiply, and a 6-input 2-bit adder tree). The latency of the mux and the two fully-connected layers is only 73 gate delays. Again, using an adder as the reference, we estimate an upper bound of 4 CPU cycles for the prediction latency.

Table 2. Breakdown of Mini-BranchNet inference engine storage requirements for one static branch.

	In terms of Architecture Knobs	Mini-BranchNet Storage
Convolution Tables	$\sum_{i=1}^{5} C_i.(2^h)$	4352 bits (0.53 KB)
Precise Pooling Buffers	$\sum_{i=1}^{2}(5 + P_i + 3.(1 + H_i - P_i))$	1063 bits (0.13 KB)
Sliding Pooling Buffers	$\sum_{i=3}^{5}(7 + log_2(P_i) + 3.(H_i/P_i))$	408 bits (0.05 KB)
Fully-connected weights	$4.N.(\sum_{i=1}^{5} C_i.(H_i/P_i)) + 2.N$	2286 bits (0.28 KB)
Total	Sum of All Components Above	8109 bits (0.99 KB)

Table 2 shows the breakdown of storage needed to predict a single hard-to-predict branch using the the Mini-BranchNet inference engine. In our experiments, we use an engine with a capacity of 16 branches, for a total of 16KB storage.

5.5 System and ISA Requirements

The ISA of a processor with a BranchNet predictor should be augmented with three instructions: LOAD_BRANCHNET, ENABLE_BRANCHNET, and DISABLE_BRANCHNET. The operating system is responsible for using these instructions to manage the state of the on-chip BranchNet engine. If a program binary contains trained BranchNet models, the program loader executes LOAD_BRANCHNET initialize the on-chip predictor with pre-trained models and then executes ENABLE_BRANCHNET to start prediction with BranchNet. Since branch prediction accuracy does not affect the correctness of execution, LOAD_BRANCHNET is a non-blocking instruction that simply starts the initialization process. While the on-chip engine is being loaded, the execution continues using only the dynamic branch predictor. If the program does not contain trained BranchNet models, the loader simply executes DISABLE_BRANCHNET.

To support context switches, the operating system should save a pointer to the pre-trained models in each process state. At a context switch, the operating system uses BRANCHNET instruction to update the state of the on-chip engine for the incoming process.

6 Results

In this Section, we show the effectiveness of BranchNet on SPEC2017 Integer Speed Benchmarks. We chose SPEC benchmarks because we could use various inputs for the same benchmark to test the generalization of offline training to unseen data. We measure the available headroom of offline training using Big-BranchNet and show the storage efficiency of CNNs using Mini-BranchNet.

6.1 Evaluation Methodology

We run each SPEC2017 Integer Speed benchmark using inputs provided by SPEC (*train* and *ref* inputs) and Alberta inputs[3]. We collect branch traces from each workload's representative regions using SimPoints [44]. [5] We train BranchNet models using the process described in Section 5.1. Table 3 shows how we partition the inputs to generate the datasets needed for offline training.

To evaluate the MPKI on the validation set and the test set, we use the weighted average of MPKI across Simpoints. However, we do not use Simpoint weights in the training set. By ignoring Simpoint weights in the training set, we encourage the training algorithm to find a general solution that works even for infrequent control flow behavior of the programs.

Each input of the benchmarks *gcc* and *xz* consists of a data file, and a set of execution flags. The execution flags of *gcc* control the optimization settings, and the control flags of *xz* control the compression level. Since these high-level execution flags likely do not change frequently in deployment, it is reasonable to train specialized CNN models targeting runs with certain execution flags. Thus, to collect the training and validation sets of *xz* and *gcc*, we used variants of SPEC *train* and Alberta inputs that use the same execution flags as SPEC *ref* inputs.

To train BranchNet, we use a minibatch size of 512, an Adam optimizer [22], and an initial learning rate of 0.002 with stepwise learning rate decay. We train each model for 15000 steps (independent of the training set size and the convergence rate).

Table 3. Inputs of SPEC workloads that we use to evaluate BranchNet.

The training set	Alberta inputs	Used for training BranchNet
The validation set	SPEC train inputs	Used for selecting the most improved branches
The test set	SPEC ref inputs	Used for final evaluation of BranchNet accuracy

[5] We collect up to 10 simpoints.

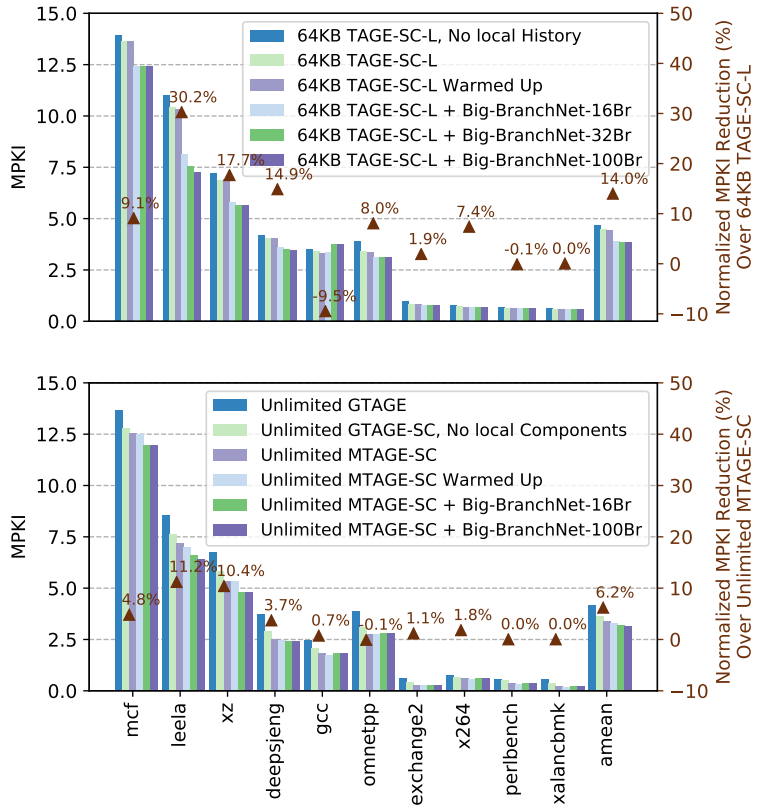

Figure 10. MPKI of Big-BranchNet on SPEC2017 benchmarks.

6.2 Measuring Headroom with Big-BranchNet

Figure 10 shows the MPKI reduction of using Big-BranchNet along with 64KB TAGE-SC-L [40] and unlimited MTAGE-SC [39]. Using 64KB TAGE-SC-L as the baseline, Big-BranchNet reduces the average MPKI of SPEC2017 Integer benchmarks from 4.46 to 3.84 (14.0% reduction). Compared to Unlimited MTAGE-SC, Big-BranchNet reduces the average MPKI from 3.36 to 3.15 (6.2% reduction). Big-BranchNet improves the overall MPKI by only predicting a few branches that are hard to predict for TAGE. To illustrate the impact of the number of branches that BranchNet predicts, we limited Big-BranchNet to predict at most 16, 32, or 100 static branches per each benchmark. *leela* is the only benchmark with non-negligible improvement when BranchNet predicts more than 32 static branches.

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*, and *deepsjeng* are reduced by more 10% (30.2% on leela) compared to 64KB TAGE-SC-L. Big-BranchNet also moderately improves benchmarks *mcf* and *omnetpp* (9.1% and 8.0%). However, we should

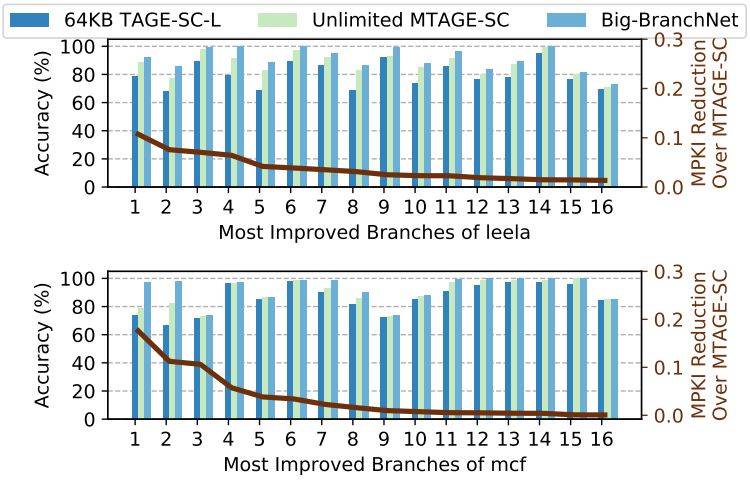

Figure 11. Accuracy of most improved branches using Big-BranchNet.

note that the most significant hard-to-predict branches of these benchmarks are memory-dependent. BranchNet cannot learn anything from the branch history to reliably predict such branches.

The only high MPKI benchmark that Big-BranchNet fails to improve is *gcc*. In fact, when predicting more than 16 branches, BranchNet hurts the overall prediction accuracy. The fall in accuracy is caused by a single static branch ranked as the 18th best branch for BranchNet using the validation set.[6] However, when using SPEC ref inputs, we found TAGE-SC-L significantly outperforms Big-BranchNet (99.97% to 96.43% accuracy). After further examination, we observed there are only 50,000 dynamic instances of this branch in the validation set, but more than 4 million dynamic instances of the same branch in the test set. The validation set clearly did not cover the branch behavior of the test set for this branch. In hindsight, we should have collected larger samples and more simpoints for *gcc*. *gcc* has a large number of static branches and has significant variation in phase behavior. Thus, larger training sets and validation sets are critical to avoid overfitting to particular phases and inputs.

The rest of the benchmarks (*exchange2*, *x264*, *perlbench*, and *xalancbmk*) have low MPKI on TAGE-SC-L. Since branch prediction is not a performance bottleneck for these benchmarks, we did not study why BranchNet does not significantly improve their prediction accuracy.

To better understand the limitations of TAGE-SC, Figure 10 also shows the MPKI of TAGE-SC-L and MTAGE-SC without certain key componets. Most of the accuracy gap between TAGE-SC-L and MTAGE-SC is due to the size of global history TAGE and the Statistical Corrector. The local history components are also significant, specially for a few benchmarks. We also evaluated TAGE-SC-L and MTAGE-SC with 20 millions of warmup

[6]The 18th best branch is excluded from BranchNet-16Br, but included in BranchNet-32Br/100Br.

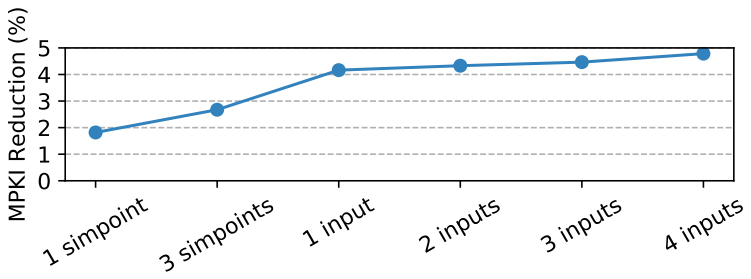

Figure 12. Sensitivity of Big-BranchNet to the training set size.

instructions. The MPKI improvement due to warmup is not significant compared to the benefits of BranchNet.

Figure 11 shows the accuracy of 16 most improved branches of *leela* and *mcf* compared to unlimited MTAGE-SC. The branches are sorted using MPKI reduction from left to right. In many cases, Big-BranchNet is able to improve the prediction accuracy to almost 100%. Branch B in Figure 4 is inspired by one of the branches of *leela*, whose accuracy is improved from 79.2% on 64KB TAGE-SC-L to 99.97% on BranchNet. Note that even if the improvement in accuracy is not as high for all branches, since these branches are among the most frequently mispredicted branches of the program, even small improvements in accuracy result in high MPKI reduction. The top two improved branches of *mcf* are also particularly interesting. BranchNet predicts these branches 97.3% and 98.2% accurately, while 64KB TAGE-SC-L predicts the same branches with an accuracy of 73.5% and 66.8% respectively. Even with unlimited storage budget, MTAGE-SC only improves the accuracies to 78.6% and 82.5%. These two branches showcase how BranchNet is able to predict some branches that are categorically impossible to predict using a TAGE-like prediction mechanism.

Figure 12 shows the MPKI reduction of BranchNet over unlimited MTAGE-SC using different training set sizes. More training inputs and larger samples generally increase the MPKI reduction of BranchNet. Training with only one simpoint does not work well because BranchNet overfits to the simpoint. Training with all the simpoints of one program, however, allows the training to be more generalizable. Using more than one input shows diminishing returns. The diminishing return of using more inputs is expected. Once the training set has enough coverage of branch behavior, BranchNet learns invariant correlations among branches that are true for all inputs. More training inputs cannot help any further. Note that Alberta inputs for *mcf* are randomly generated and have no connections to 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.

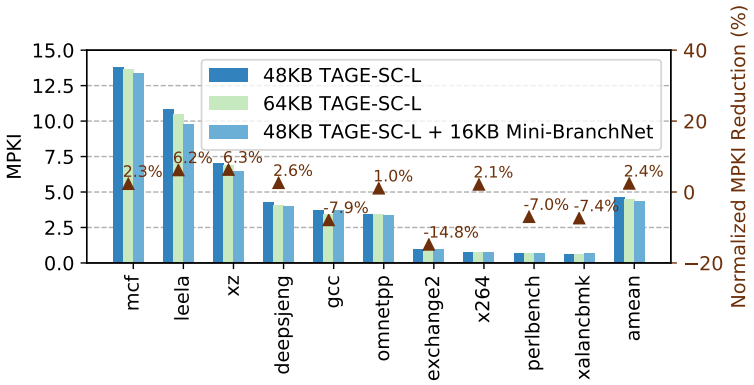

Figure 13. MPKI of Mini-BranchNet on SPEC2017 benchmarks.

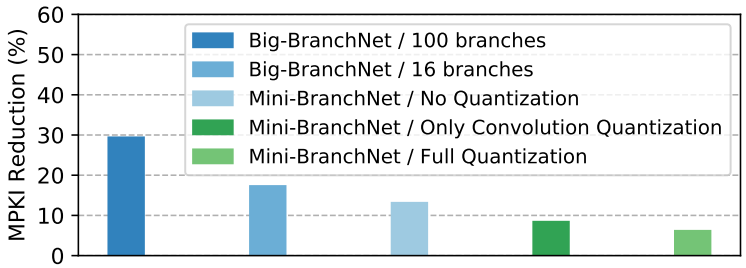

Figure 14. Progression of MPKI reduction of *leela* from Big-BranchNet to Mini-BranchNet.

6.3 Practical Mini-BranchNet Results.

In order to compare BranchNet and TAGE-SC-L with the same 64KB storage budget, we paired a 16KB Mini-BranchNet with a 48KB TAGE-SC-L. We built the 48KB TAGE-SC-L by decreasing the number of banks and tag widths of TAGE component of 64KB TAGE-SC-L. Figure 13 shows that the combination of 48KB TAGE-SC-L and Mini-BranchNet outperforms 64KB TAGE-SC-L on average. Mini-BranchNet is more effective for high-MPKI benchmarks, where it can improve the overall MPKI by improving the accuracy of just a few hard-to-predict branches. However, on benchmarks with either too many (*gcc*) or too few hard-to-predict branches, Mini-BranchNet does not have a significant impact, and the overall MPKI suffers because of a smaller TAGE-SC-L predictor.

Figure 14 illustrates why the MPKI reduction of Mini-BranchNet inference engine is significantly lower than Big-BranchNet for benchmark *leela*. Much of the loss in improvement is because Mini-BranchNet inference engine can only predict 16 hard-to-predict branches. The other huge factor is precision loss due to quantization. Smaller CNN architecture (e.g. fewer convolution channels, fewer hidden neurons) is also responsible for some

loss in MPKI reduction. We believe all these factors could be alleviated with more optimization. For example, there is room to improve the efficiency of convolution tables by using different hash functions for each channel. Another approach is to completely binarize the fully-connected layers using the training algorithm proposed by Courbariaux et al. [6].

The poor performance of BranchNet and Mini-BranchNet on *gcc* highlights a fundamental weakness of BranchNet. Some programs have many hard-to-predict branches, which limits the potential of improving the overall MPKI by improving the accuracy of a few branches. To improve such programs, a run time replacement mechanism is needed to help BranchNet load the CNN models of only the branches that occur in the current phase. Such a mechanism enables BranchNet to work well with programs that contain many hard-to-predict branches. We leave the evaluation of run time replacement mechanisms for future work.

7 Other Related Work

BranchNet uses the main fundamental insight of the work of Tarsa et al. [53], that is, training CNNs offline to help improve prediction accuracy of hard-to-predict branches. The architecture of BranchNet, however, is optimized to scale well with long history length through the use of geometric history lengths and aggressive sum pooling. Furthermore, BranchNet inference engine optimizations enable multi-branch convolution windows, which is necessary for accurately predicting many of the hard-to-predict branches in SPEC2017 benchmark.

Mao et al. [27, 28] have explored the use of deep belief network and deep convolutional neural networks for branch prediction. There are two problems with their work. One, their proposed CNN architecture consists of 10 layers and hundreds of convolution filters, too large to be an effective branch predictor. Two, they use the same input for training and evaluation, which of course is unacceptable for a practical branch predictor.

Shi et al. [45] introduce an alternative to history-based branch predictors by learning dynamic program behavior with Graph Neural Networks. Their work does not propose a practical design for a branch predictor nor evaluates their prediction model against state-of-the-art branch predictors. However, they show an important insight that deep learning can be a valuable method of learning program behavior (including branch behavior) from elements of the execution state that have traditionally not been used for prediction.

Farooq et al. [8] propose Store-Load-Branch (SLB) predictor, which predicts memory-dependent branches by identifying dependent store-load-branch chains in a program using the compiler. At run time, if the program executes a store instruction in a store-load-branch chain, SLB predictor determines the direction of the branch in the chain. When SLB has to predict the branch in the same chain, SLB simply looks up the previously resolved direction of the branch. SLB and BranchNet target orthogonal classes of branches and can work well together.

Adileh et al. [1] propose Probabilistic Branch Support (PBS), which eliminates all mispredictions due to prob-

abilistic branches in programs such as Monte Carlo algorithms. Probabilistic branches are by definition hard to predict. Instead of predicting such branches, PBS buffers prior randomly generated numbers and uses an older value to resolve the branch. Even though PBS breaks the program order of instructions, the algorithms still converge to the correct result as long as the branch probability distribution does not change. Again, PBS and BranchNet target orthogonal classes of branches.

Note that SLB, PBS, and BranchNet share the same insight that we can improve branch prediction by exposing the predictors to the ISA and the user. SLB uses the compiler to get information about program data flow. PBS gets help from the programmer to identify probabilistic branches and relax program order requirements. BranchNet improves branch prediction by using offline profiling. Exposing branch prediction to the ISA is key in all these techniques.

Seznec et al. [42] propose using Inner Most Loop Iteration (IMLI) counters to identify correlated branches in history. Inspired by the Wormhole predictor [2], IMLI counters are useful for predicting branches within nested loops that are correlated to the branches in the previous iterations of the outer loop. BranchNet is compatible with using IMLI counters as inputs. We leave the study of using IMLI counters as inputs to BranchNet for future work.

Gope and Lipasti [11] propose bias-free branch predictors to remove biased and redundant branches from branch histories. BranchNet and the bias-free predictor both target the same problem that not all branches in the history matter. The bias-free predictor addresses this problem using a simple run time filtering mechanism. However, offline deep learning allows BranchNet to be more effective.

8 Conclusion

BranchNet is the first profiling approach to branch prediction with significant gains over the highly accurate dynamic predictors that exist today. BranchNet shows that contrary to conventional wisdom, one can train highly accurate branch predictors offline that can predict a certain category of branches that are hard to predict for state-of-the-art dynamic branch predictors. We have demonstrated the powerful potential of this approach by comparing Big-BranchNet against MTAGE-SC, the best state-of-the-art predictor without area constraints. By taking advantage of large datasets of branch behavior across program runs, BranchNet learns invariant branch relationships that can predict future execution of programs with unseen inputs. Overall, BranchNet without area constraints improves the average MPKI of SPEC benchmarks by 6.2%, and up to 11.2%.

Mini-BranchNet is a first step towards a practical CNN branch predictor and demonstrates that there are plenty of opportunities for optimizing a CNN inference engine. With a 64KB storage budget, Mini-BranchNet improves the average MPKI of SPEC benchmarks by 2.4%, and up to 6.3%. These results, however, should not be interpreted as a limit to the potential benefits of BranchNet. We believe future research can close the accuracy gap between

practical CNN predictors and the high potential of deep learning. The key takeaway from BranchNet is that offline deep learning is a powerful approach to train highly accurate complex predictors that were previously deemed to be impractical.

References

[1] 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.

[2] J. Albericio, J. S. Miguel, N. E. Jerger, and A. Moshovos. Wormhole: Wisely predicting multidimensional branches. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-47, pages 509–520, Washington, DC, USA, 2014. IEEE Computer Society.

[3] 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)*, pages 159–168, April 2018.

[4] 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.*, 19(1):188–222, Jan. 1997.

[5] J. Cleary and I. Witten. Data compression using adaptive coding and partial string matching. *IEEE Transactions on Communications*, 32(4):396–402, April 1984.

[6] 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.

[7] M. Evers. *Improving Branch Prediction by Understanding Branch Behavior*. PhD thesis, Ann Arbor, MI, USA, 2000. AAI9963778.

[8] M. Farooq, K. Khubaib, and L. John. Store-load-branch (slb) predictor: A compiler assisted branch prediction for data dependent branches. pages 59–70, 02 2013.

[9] A. Fog. Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for intel, amd and via cpus. Technical report, Technical University of Denmark.

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

[11] D. Gope and M. H. Lipasti. Bias-free branch predictor. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 521–532, Dec 2014.

[12] M. Hashemi, K. Swersky, J. Smith, G. Ayers, H. Litz, J. Chang, C. Kozyrakis, and P. Ranganathan. Learning memory access patterns. In J. Dy and A. Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 1919–1928, Stockholmsmässan, Stockholm Sweden, 10–15 Jul 2018. PMLR.

[13] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural computation*, 9:1735–80, 12 1997.

[14] J. Hu, L. Shen, and G. Sun. Squeeze-and-excitation networks. *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, Jun 2018.

[15] 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*, ICML'15, pages 448–456. JMLR.org, 2015.

[16] D. Jiménez. Multiperspective perceptron predictor. In *5th JILP Workshop on Computer Architecture Competitions (JWAC-5): Championship Branch Prediction (CBP-5)*, 2016.

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

[18] 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*, pages 97–106, Sep. 2001.

[19] 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*, pages 67–76, Dec 2000.

[20] D. A. Jimenez and C. Lin. Dynamic branch prediction with perceptrons. In *Proceedings HPCA Seventh International Symposium on High-Performance Computer Architecture*, pages 197–206, Jan 2001.

[21] T. S. Karkhanis and J. E. Smith. A first-order superscalar processor model. In *Proceedings. 31st Annual International Symposium on Computer Architecture, 2004.*, pages 338–349, June 2004.

[22] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2015.
[23] A. Krall. Improving semi-static branch prediction by code replication. *SIGPLAN Not.*, 29(6):97–106, June 1994.
[24] 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.*, 1(4):541–551, Dec. 1989.
[25] Y. Li, D. Tarlow, M. Brockschmidt, and R. Zemel. Gated graph sequence neural networks, 2015.
[26] C.-K. Lin and S. J. Tarsa. Branch prediction is not a solved problem: Measurements, opportunities, and future directions, 2019.
[27] Y. Mao, J. Shen, and X. Gui. A study on deep belief net for branch prediction. *IEEE Access*, 6:10779–10786, 2018.
[28] Y. Mao, H. Zhou, and X. Gui. Exploring deep neural networks for branch prediction. 2018.
[29] S. Mcfarling. Combining branch predictors. Technical report, Digital Equipment Corporation, Western Research Lab, 1993.
[30] P. Michaud, A. Seznec, and S. Jourdan. An exploration of instruction fetch requirement in out-of-order superscalar processors. *International Journal of Parallel Programming*, 29(1):35–58, Feb 2001.
[31] 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)*, pages 3–14, Dec 2007.
[32] 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*, ICML'10, pages 807–814, USA, 2010. Omnipress.
[33] R. Panda, S. Song, J. Dean, and L. K. John. Wait of a decade: Did spec cpu 2017 broaden the performance horizon? In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 271–282, Feb 2018.
[34] J. R. C. Patterson. Accurate static branch prediction by value range propagation. *SIGPLAN Not.*, 30(6):67–78, June 1995.
[35] 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.
[36] H. Robbins and S. Monro. A stochastic approximation method. *Ann. Math. Statist.*, 22(3):400–407, 09 1951.
[37] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Neurocomputing: Foundations of research. chapter Learning Representations by Back-propagating Errors, pages 696–699. MIT Press, Cambridge, MA, USA, 1988.
[38] A. Seznec. Analysis of the o-geometric history length branch predictor. In *32nd International Symposium on Computer Architecture (ISCA'05)*, pages 394–405, June 2005.
[39] A. Seznec. Exploring branch predictability limits with the MTAGE+SC predictor *. In *5th JILP Workshop on Computer Architecture Competitions (JWAC-5): Championship Branch Prediction (CBP-5)*, page 4, Seoul, South Korea, June 2016.
[40] A. Seznec. Tage-sc-l branch predictors again. In *5th JILP Workshop on Computer Architecture Competitions (JWAC-5): Championship Branch Prediction (CBP-5)*, 2016.
[41] A. Seznec and P. Michaud. A case for (partially) tagged geometric history length branch prediction. *J. Instruction-Level Parallelism*, 8, 2006.
[42] 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*, MICRO-48, pages 347–357, New York, NY, USA, 2015. ACM.
[43] T. Sherwood and B. Calder. Automated design of finite state machine predictors for customized processors. In *Proceedings 28th Annual International Symposium on Computer Architecture*, pages 86–97, June 2001.
[44] 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*, ASPLOS X, pages 45–57, New York, NY, USA, 2002. ACM.
[45] Z. Shi, K. Swersky, D. Tarlow, P. Ranganathan, and M. Hashemi. Learning execution through neural code fusion, 2019.
[46] E. Sprangle and D. Carmean. Increasing processor performance by implementing deeper pipelines. In *Computer Architecture, 2002. Proceedings. 29th Annual International Symposium on*, pages 25–34. IEEE, 2002.
[47] A. Sriraman, A. Dhanotia, and T. F. Wenisch. Softsku: Optimizing server architectures for microservice diversity @scale. In *Proceedings of the 46th International Symposium on Computer Architecture*, ISCA '19, pages 513–526, New York, NY, USA, 2019. ACM.
[48] J. Stark, M. Evers, and Y. N. Patt. Variable length path branch prediction. *SIGOPS Oper. Syst. Rev.*, 32(5):170–179, Oct. 1998.
[49] C. Szegedy, S. Ioffe, V. Vanhoucke, and A. Alemi. Inception-v4, inception-resnet and the impact of residual connections on learning, 2016.

[50] D. Tarjan and K. Skadron. Merging path and gshare indexing in perceptron branch prediction. *ACM Trans. Archit. Code Optim.*, 2(3):280–300, Sept. 2005.

[51] 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*, pages 82–87, Oct 1997.

[52] S. J. Tarsa, R. B. R. Chowdhury, J. Sebot, G. Chinya, J. Gaur, K. Sankaranarayanan, C.-K. Lin, R. Chappell, R. Singhal, and H. Wang. Post-silicon cpu adaptation made practical using machine learning. In *Proceedings of the 46th International Symposium on Computer Architecture*, ISCA '19, pages 14–26, New York, NY, USA, 2019. ACM.

[53] S. J. Tarsa, C.-K. Lin, G. Keskin, G. Chinya, and H. Wang. Improving branch prediction by modeling global history with convolutional neural networks, 2019.

[54] 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*, pages 239–247, Dec 2009.

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

[56] A. Yasin. A top-down method for performance analysis and counters architecture. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 35–44, March 2014.

[57] T.-Y. Yeh and Y. N. Patt. Two-level adaptive training branch prediction. In *Proceedings of the 24th Annual International Symposium on Microarchitecture*, MICRO 24, pages 51–61, New York, NY, USA, 1991. ACM.

[58] C. Young and M. D. Smith. Improving the accuracy of static branch prediction using branch correlation. *SIGOPS Oper. Syst. Rev.*, 28(5):232–241, Nov. 1994.