BranchNet: A Convolutional Neural Network to Predict Hard-To-Predict Branches

Siavash Zangeneh*, Stephen Pruett*, Sangkug Lym†, and Yale N. Patt*
siavash.zangeneh@utexas.edu, stephen.pruett@utexas.edu, slym@nvidia.com, patt@ece.utexas.edu
*University of Texas at Austin †Nvidia

***Abstract*—The state-of-the-art branch predictor, TAGE, remains inefficient at identifying correlated branches deep in a noisy global branch history. We argue 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, Tarsa et al. proposed 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. We build on their work and introduce BranchNet, a CNN with a practical 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 7.6% (and up to 15.7%) when compared to a very large (impractical) MTAGE-SC baseline, demonstrating a fundamental advantage in the prediction capabilities of BranchNet compared to TAGE-like predictors. We also propose a practical resource-constrained variant of BranchNet that improves the MPKI by 9.6% (and up to 17.7%) compared to a 64KB TAGE-SC-L without increasing the prediction latency.**

I. Introduction

Branch prediction remains a major bottleneck in improving single-thread performance. Even with TAGE-SC-L [1], the state-of-the-art branch predictor, many SPEC2017 Integer benchmarks still 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 [2]–[5]. Unfortunately, fundamental breakthroughs in branch prediction have become rare [6]. All predictors submitted to the 2016 Championship Branch Prediction competition were variants of existing TAGE and Perceptron designs [1], [7]–[10]. Branch prediction research needs new insights to further improve the prediction accuracy.

Traditional branch predictors like TAGE [11] and Perceptron [12] 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 runtime, limitations in training time and processing power make it difficult for online 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 and additional compiler support.

Building on the work of Tarsa et al. [13], we propose BranchNet, a convolutional neural network (CNN) that addresses a key weakness of TAGE-like predictors: identifying correlated branches in a noisy global history. When the global history is noisy, (i.e., the global history contains uncorrelated branches that constantly change directions, or the positions of correlated branches in the history are nondeterministic), a TAGE-like predictor has to dedicate unique prediction counters for each possible history pattern. Thus, the number of counters needed grows exponentially with the size of the history, making a TAGE-like approach infeasible when correlated branches appear deep into a noisy history. A CNN, however, learns to ignore uncorrelated branches and identify correlated branch patterns anywhere in the history, enabling expressive prediction functions that remain efficient even with a long noisy history.

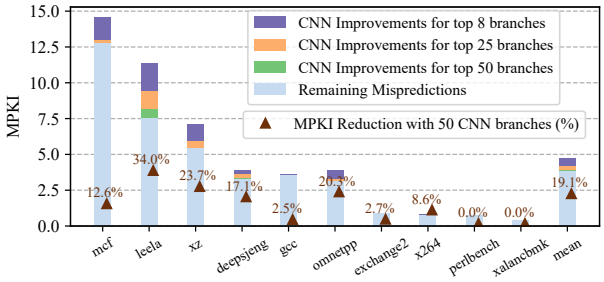

Fig. 1. MPKI Reduction of using large CNNs to predict a few hard-to-predict branches along 64KB TAGE-SC-L.

This increase in prediction capability, however, 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, we use 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.

Fig. 1 shows the potential of using large CNNs to predict the top few hard-to-predict 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 CNNs to predict up to 8, 25, or 50 static branches. The figure demonstrates that for most benchmarks, predicting 8

978-1-7281-7383-2/20/$31.00 ©2020 IEEE
DOI 10.1109/MICRO50266.2020.00022

branches with CNNs is sufficient for significant overall MPKI reduction, and often predicting more than 25 branches with CNNs has diminishing returns. Thus, we opt for a hybrid approach, using CNNs to predict a few hard-to-predict static branches and using state-of-the-art runtime predictors for all other branches.

Tarsa et al. [13] were the first to propose using CNNs with offline training to predict hard-to-predict branches. They showed 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. BranchNet builds on their approach and tailors the CNN architecture to branch prediction, resulting in higher prediction accuracy and better storage-efficiency. The contributions of this paper are:

- We identify a class of branches that are hard-to-predict by conventional runtime branch predictors but can be predicted accurately by convolutional neural networks. These branches are correlated to the counts of other braches in a noisy global history. When the history is noisy, a table-based predictor (e.g. TAGE) relies on allocating predictor entries for each history pattern, which is infeasible for long histories. However, a CNN expresses the actual branch relationship by counting the explicitly identified correlated branches in the global history.
- We make a new case for branch prediction with offline training. We show that unlike previously proposed offline training techniques, BranchNet relies less on the representativeness of the training data and more on coverage. The key is exposing enough control flow paths to detect input-independent branch correlations that can be generalized to unseen inputs.
- We propose a CNN architecture tailored to branch prediction requirements in two ways. One, we draw inspiration from traditional branch predictors and use geometric history lengths as inputs. Two, we use 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.
- We demonstrate a novel way to approximate wide convolution filters and sum-pooling layers. These approximations enable BranchNet to have the same prediction latency as TAGE-SC-L (4 cycles) and be more storage-efficient.

In the rest of the paper, we first motivate why CNNs with offline training can overcome a 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 runtime, 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 7.6% (up to 15.7% for the most improved benchmark) when compared to an unlimited MTAGE-SC baseline. We also show that by using our area/latency constrained BranchNet inference engine along with a 64KB TAGE-SC-L, we can improve the MPKI by 9.6% (up to 17.7%), and IPC by 1.3% (up to 7.9%) over a 64KB TAGE-SC-L baseline.

II. Limitations of state-of-the-art

Current-day online (runtime) branch predictors have a key weakness: they need an exponentially growing capacity in the presence of long noisy histories. Even though we argue this weakness is a fundamental consequence of runtime training, prior branch predictors that use offline training (with profiling) do not remedy this weakness. In this section, we describe the phenomenon of noisy history and make a case for why deep learning can succeed where previous attempts at offline training failed.

A. Online Branch Predictors

All state-of-the-art branch predictors are variants of TAGE [11] and the hashed perceptron [14]. While their prediction mechanisms may differ, all conventional predictors hash 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 [15]). 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.

TAGE-SC-L. TAGE-SC-L is the winner of Championship Branch Prediction 2016 [1]. Its main component, TAGE [11], hashes the global branch and path history to lookup tables of tagged saturating counters that provide the prediction. It uses multiple counter tables, each corresponding to a unique history length. Longer history tables are used only when shorter history tables cannot provide accurate predictions. If prediction accuracy depends on correlated branches deep into a noisy global history, short history tables do not provide any value, resulting in high allocation pressure on the long history tables. In the worst case, a long history TAGE table behaves similarly to a global 2-level predictor [16], which requires $O(2^n)$ table entries for an n-bit history input, which is infeasible for a large n.

Perceptron. Perceptron-based branch predictors use a single-layer neural network to learn the correlations of the branch outcome to its history bits. To make a prediction, the predictors add the correlation factors and compare the sum to the branch bias. The Perceptron predictor [12] finds an individual correlation factor for each bit position in the global history. When the positions of branches in the global history are nondeterministic, the correlations to the history bits become unreliable. The hashed perceptron [14] (also used in the newest perceptron-based predictor, Multiperspective Perceptron [8]) learns correlation factors for hashes of the global branch and path history, which mitigates the problem of nondeterministic positions. Still, when the history is noisy, the hashed perceptron suffers from significant aliasing among history patterns and their hashes, resulting in significant loss of accuracy.

Perceptron-based branch predictors have another inherent limitation: because they use a single-layer neural network, they cannot learn non-linear relationships between branches. The fact that all prior perceptron-based predictors use a single neuron is a consequence of runtime training. The training algorithm for multi-layer perceptrons is too expensive to be implementable at runtime.

Since TAGE predictors outperform Perceptron-based predictors on our benchmarks, we use TAGE as our baseline for state-of-the-art runtime branch predictors.

B. Branch Predictors with Offline Training

Prior predictors. 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 [17]–[20]. Some work use profiling to train application-specific predictors, resulting in a comparable accuracy to contemporary dynamic branch predictors [21]–[25]. The most recent proposal, Spotlight [25], is a gshare-like predictor [26] 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. The only offline method we can easily apply to TAGE-SC-L is to use static branch biases when TAGE-SC-L is not confident. However, we have observed that using static biases only slightly improves the accuracy of TAGE-SC-L (0.3% MPKI reduction for the best benchmark, 0.0% for many) and its benefits are orthogonal to the contributions of BranchNet.

Since state-of-the-art predictors do not benefit from using prior offline techniques, we only compare BranchNet to the best online predictors.

Representativeness vs. coverage. The key advantage of offline training is the removal of time and compute constraints from training, enabling arbitrarily complex training algorithms. However, prior offline predictors never fully leverage this because they only train simple prediction mechanisms that rely on the repetition of exact history patterns. 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. Section IV provides a concrete example of this distinction.

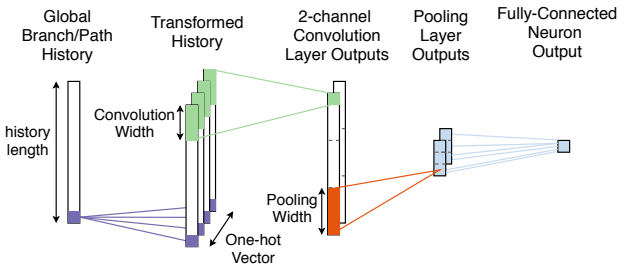

Fig. 2. Dataflow in a simple CNN branch predictor.

III. Background

Convolutional Neural Networks (CNN) are state-of-the-art in both image classification [27], [28], and sequential tasks like natural language understanding [29]. 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.

In this section, we provide a high-level description of a simple CNN branch predictor. The goal is to introduce the terminology and provide an intuition for how the CNN components work together to predict branches.

A. CNN Building Blocks

Fig. 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. Alternatively, we can use embeddings to transform each history element into a vector representation trained specifically for the problem we want to solve [30]. 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. [31] use embeddings to represent PC and memory addresses in a model for data prefetching, which is very similar to BranchNet representing branch PC and directions.

Convolutional layers. At a high level, a convolution layer identifies the occurrences of features in its input [30], [32]. The set of weights that are trained to identify a feature is

[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).

```
1  int x = 0;
2  for (int i = 0; i < N; ++i) {
3    if (random_condition(alpha)) { // Branch A
4      // x increments if Branch A is not taken
5      x += 1;
6    }
7  }
8
9  uncorrelated_function();
10
11 for (int j = 0; j < x; ++j) { // Branch B
12   ...
13 } // exits when Branch B is taken
```

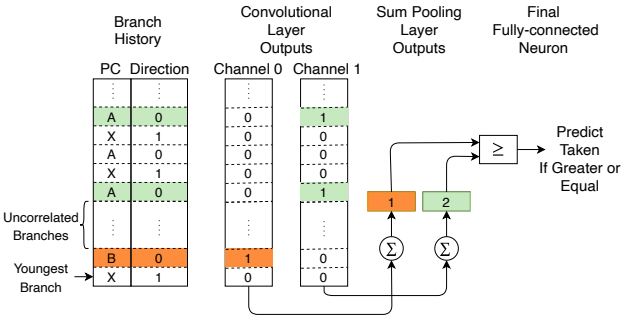

Fig. 3. A program with a hard-to-predict branch (Branch B) and a trained CNN that can accurately predict the branch.

called a *filter*. The *convolution width* controls the number of neighboring items that form a feature. For branch prediction, the neighboring items we consider are 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.

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 [30]. 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 layers at the cost of discarding fine-grained positions of identified features. As we show later in Section IV, 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 [30]. 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.

B. Training Algorithm

We train CNNs 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 [33] using Backpropagation [34]) is computationally expensive, the training has to be done offline using profiling. Thus, a good training set for

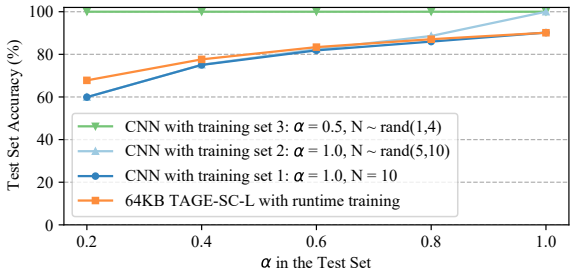

Fig. 4. Accuracy of predicting Branch B from Fig. 3. $N \sim rand(5, 10)$ in the test set.

branch prediction should contain examples from multiple input sets and exercise different control flow paths, which enables the CNN to learn invariant branch relationships.

IV. Motivation

We use the source code in Fig. 3 to show how CNNs can predict otherwise hard-to-predict branches. The code is a simplified version of a hot segment of the benchmark *leela*, which is responsible for a significant fraction of the total number of mispredictions.

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.

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 actually useful for prediction. Thus, as explained in Section II-A, 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 an online 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.

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. Fig. 3 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

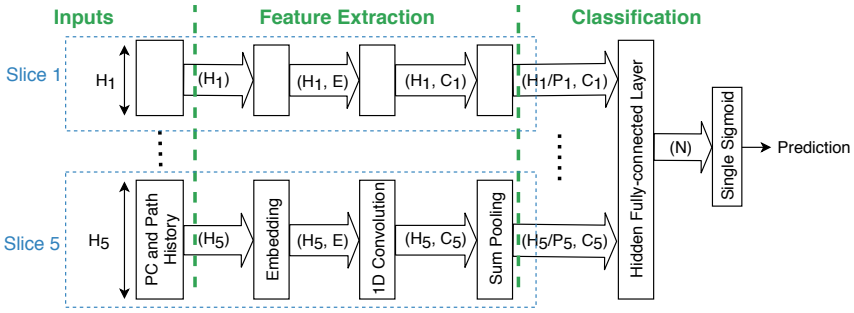

Fig. 5. High-level diagram of Big-BranchNet CNN architecture for one branch.

not involved in the prediction (i.e. uncorrelated branches) are marked as X. The history is encoded as one-hot vectors (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. We use 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.

Does offline training work? Thus far, we have shown that a manually configured CNN can predict Branch B. Now, we show that we can train a CNN offline using profiling. Suppose the random condition in line 3 of Fig. 3 is set using a Bernoulli distribution that is true with probability α, and N is set using a uniform distribution with adjustable minimum and maximum. We 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) $N \sim rand(1, 4)$, $\alpha = 0.5$. We 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. We also evaluated the accuracy of a 64KB TAGE-SC-L (with normal runtime training) on the same test sets. Fig. 4 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.

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.

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 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 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, we defer to the baseline online branch predictor to predict these branches.

As discussed earlier in Section I, Fig. 1 shows the MPKI reduction of using large CNN models to predict the top hard-to-predict branches in SPEC2017 benchmarks. Since we use the same input signals for TAGE-SC-L and the CNN models, 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.

Can Other Machine Learning Models Predict Branches? Any sophisticated learning model can learn invariant branch relationships from large training sets. For example, Recurrent Neural Network can also predict the same type of hard-to-predict branches as BranchNet. However, we limit the scope of this paper to the study of CNNs for branch prediction because we see a clearer path towards low-latency and storage-efficient branch predictors with CNNs.

V. 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 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.

A. Big-BranchNet

Big-BranchNet is a pure software model and we do not propose using it as a practical branch predictor. Big-BranchNet is composed of 5 feature extraction sub-networks and two fully-connected layers. We call each feature extraction sub-

TABLE I
BRANCHNET ARCHITECTURE KNOBS.

Knob	Big-BranchNet	Mini-BranchNet 2KB	Mini-BranchNet 1KB	Mini-BranchNet 0.5KB	Mini-BranchNet 0.25KB	Tarsa-Ternary 5.125KB
H: History sizes	42,78,150,294,582	37,77,152,302,603	37,77,152,302,603	37,77,152,302,603	44,92,182	200
C: Convolution channels	32,32,32,32,32	4,5,5,4,4	3,3,4,4,3	3,3,3,2,2	2,2,2	32
P: Pooling widths	3,6,12,24,48	7,15,30,60,120	7,15,30,60,120	7,15,30,60,120	7,15,30	N/A
Use Precise pooling	N/A	Y,Y,Y,N,N	Y,Y,N,N,N	Y,Y,N,N,N	Y,Y,N	N/A
p: Branch PC width	12	12	12	12	12	7
h: Convolution hash width	N/A	8	8	7	7	N/A
E: Embedding dimensions	32	32	32	32	32	N/A
K: Convolution width	7	3	3	3	3	1
N: Hidden neurons	128, 128	10	8	6	4	N/A
q: Fully-connected quantization	N/A	4	3	3	3	2

network a *slice*[2]. 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 [35]. Finally, the outputs of the slices are concatenated and fed into two sequential fully-connected layers to make a prediction. Fig. 5 shows a high-level diagram of Big-BranchNet.

We define Big-BranchNet in terms of a set of architecture knobs. For now, we explain the functionality of all Big-BranchNet layers using these architecture knobs. We report the knob values we used for Big-BranchNet and other related CNN models in Table I.

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 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 III, we could have used one-hot encodings instead of the embeddings, but we 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 independent features of the branch history. With a larger K, BranchNet can identify longer sequences of correlated branches. We always use a convolution stride of 1. The convolution operation is followed by batch normalization[3] and ReLU activations[4]. The type of activation functions is not important for Big-BranchNet's accuracy.

Sum-Pooling Layers. In each slice, a sum-pooling layer down-samples the convolution outputs 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 longer 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.

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

[3]Batch normalization converts each output channel to a standard normal distribution, which guides the learning algorithms towards better solutions without affecting the prediction capability [36].

[4]Activations are non-linear element-wise functions that are applied after convolution and fully-connected operations [30].

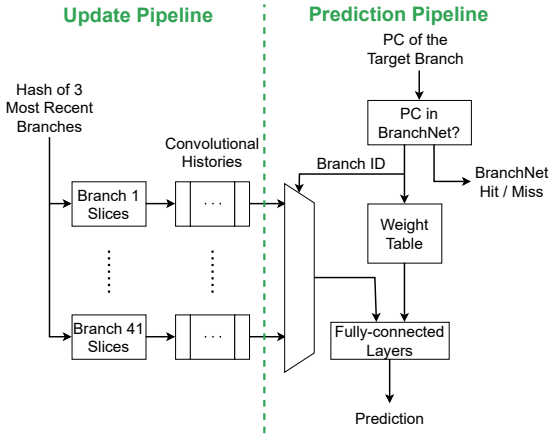

Fig. 6. Mini-BranchNet inference engine.

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.

B. Mini-BranchNet

Mini-BranchNet is a smaller variant of BranchNet that we co-design 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 we tuned to minimize storage and latency overheads. In the rest of this subsection, we describe key optimizations in designing an inference engine for Mini-BranchNet. We also explain other modifications to the BranchNet CNN architecture as they pertain to the inference engine optimizations.

Optimization 1: Maintaining Convolutional Histories. Computing the outputs of the various slices of BranchNet feature extraction layers involves operations on hundreds of branches in the global history. Instead of doing all these operations at prediction-time, 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*. Fig. 6 shows the block diagram of a Mini-BranchNet inference engine that can predict up to 41 static branches in a program. The update pipeline maintains the convolutional histories of all 41 Mini-BranchNet

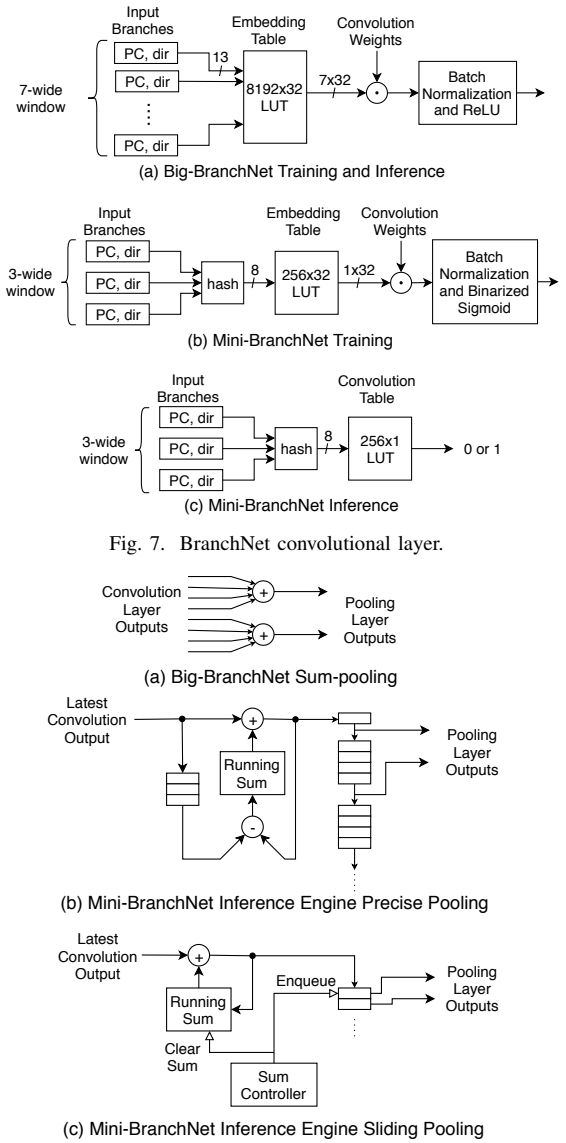

Fig. 7. BranchNet convolutional layer.

Fig. 8. BranchNet 4-wide sum-pooling.

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

Optimization 2: Replacing Convolutions with Table Lookups. A convolution operation on a single window of branches involves a dot product operation. Fig. 7a shows how Big-BranchNet computes one convolution output. 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 branches in a window (Fig. 7b) and uses binarized sigmoid [37] activations instead of ReLU. After training is done, for each possible branch hash, we compute the convolution output (embedding + dot product + normalization + binarized sigmoid), which is exactly 0 or 1. 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 (Fig. 7c). No arithmetic operation is needed at runtime, eliminating a 32-dimensional inner product per convolution operation.

Optimization 3: Using Running Sum Registers. Fig. 8a 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* (Fig. 8b). 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* (Fig. 8c). 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 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, we 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. We empirically found that using 3 or 4 bits of precision (denoted by architecture knob 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. We replace ReLU activations with Tanh to restrict the layer outputs to be between -1 and 1, which helps with quantization [37]. 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. Since the hidden fully-connected outputs are binarized, we can use a lookup table to eliminate 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, we evaluate four Mini-BranchNet models with varying storage budgets per branch. Table I reports the architecture knob values for each configu-

TABLE II
BREAKDOWN OF THE MINI-BRANCHNET INFERENCE ENGINE STORAGE REQUIREMENTS FOR ONE STATIC BRANCH.

	Using Architecture Knobs	1KB Config
Convolution Tables	$\sum(2^h)$	0.53 KB
Precise Pooling Buffers	$\sum(5 + P_i + q(1 + H_i - P_i))$	0.11 KB
Sliding Pooling Buffers	$\sum(7 + log_2(P_i) + q(H_i/P_i))$	0.04 KB
Fully-connected weights	$qN\sum(C_i(H_i/P_i) + 2^N$	0.29 KB

ration.

C. On-chip Constraints

Storage. Table II shows the breakdown of storage needed to predict a single hard-to-predict branch using the Mini-BranchNet inference engine. As an example, it also shows the storage breakdown for a 1KB Mini-BranchNet model.

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 [38]. 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 [39] for the table lookups and counting the gate delays of the arithmetic operations, 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 [40], we estimate that Mini-BranchNet updates are also 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[5] is 1.1 times the latency of the Mini-BranchNet inference engine. Thus, we 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 in the lost state, as long as we restrict our design to allow n branches in flight.

D. Differences with Prior Work

BranchNet builds on the CNN predictor of Tarsa et al. [13]. We refer to their proposed model as Tarsa-Ternary and define Tarsa-Ternary in terms of BranchNet architecture knobs in Table I. BranchNet is different from Tarsa-Ternary in five ways: it uses sum-pooling layers, it approximates 3-wide convolution filters, it uses multiple history lengths, it has an additional fully-connected layer, and it uses heterogeneous model sizes based on the needs of each branch. As a result, Mini-BranchNet is smaller, faster, and more accurate than Tarsa-Ternary.

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 to buffer only five 4-bit values (independent of history length). Because of 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.

E. Offline Training Process

We profile target programs 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 set in a 3-step process. First, we select the 100 highest MPKI branches (hard-to-predict branches) in the validation set. Then, we train one CNN model for each hard-to-predict branch using the training set. Finally, we measure the MPKI reduction of each branch on the validation set and attach the BranchNet models for the most improved branches (up to 41 branches for iso-latency Mini-BranchNet) to the program binary. To measure the final accuracy on unseen inputs, we report the accuracy of BranchNet on the test set.

F. System and ISA Requirements

BranchNet requires collaboration across the software stack for loading trained BranchNet models to the on-chip unit at runtime. We envision an approach where the trained BranchNet models are augmented to the program binary and the operating system (OS) is responsible for loading these models into the on-chip BranchNet engine at load-time or during context switches. The ISA should provide BranchNet instructions that the OS uses 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. [41] 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. We leave a more detailed analysis and evaluation of System and ISA requirements or alternative approaches to future work.

VI. 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. Big-BranchNet results demonstrate the available headroom of branch prediction with offline deep learning. Mini-BranchNet results show the benefits of using CNN branch predictors in practical settings.

[5]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.

TABLE III
INPUTS OF SPEC WORKLOADS THAT WE USE 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

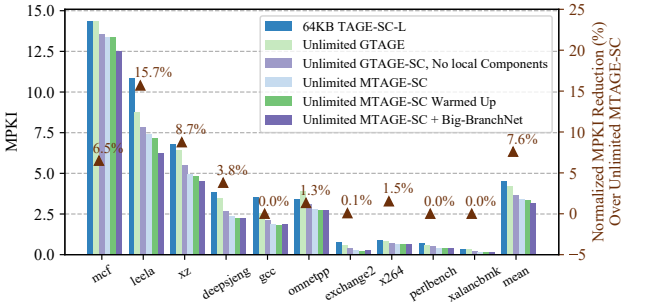

Fig. 9. MPKI of MTAGE-SC and Big-BranchNet on SPEC2017 benchmarks.

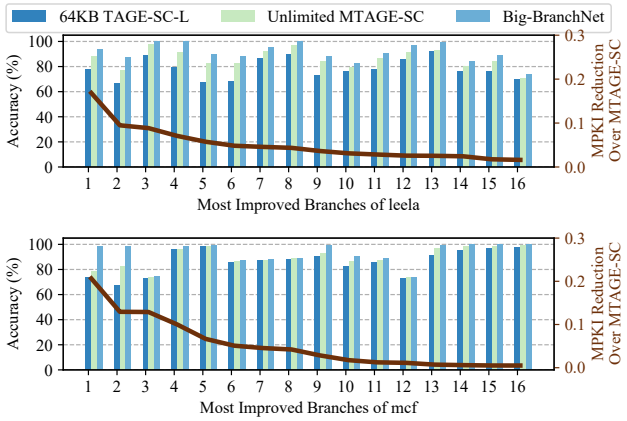

Fig. 10. Accuracy of most improved branches using Big-BranchNet.

A. Evaluation Methodology

We run each SPEC2017 Integer Speed benchmark using inputs provided by SPEC (*train* and *ref* inputs) and Alberta inputs [42]. We collect up to 10 branch traces from each workload's representative regions using SimPoints [43]. We then train BranchNet models using the process described in Section V-E. Table III shows how we partition the 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 according to SimPoint weights. Depending on the configuration, our training infrastructure takes between 6 to 18 hours on 4 GPUs to train all BranchNet models for a given benchmark. Training could be easily sped up with more GPUs since BranchNet models are trained in parallel. We have open-sourced our evaluation infrastructure [44].

We 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.

We evaluate the IPC of benchmarks using Scarab [45], an execution-driven, cycle-level simulator for x86-64 processors, which accurately models branch misprediction behavior by fetching and executing wrong-path instructions. We 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, we flush the frontend and re-fetch the instructions after the branch using the new prediction. We configure the processor to resemble a high-performance processor: 6-wide fetch, 512-entry ROB, 2MB LLC, 10-stage frontend pipeline, execution latency similar to an Intel Skylake processor [40], and DDR4 main memory simulated with Ramulator [46].

B. Measuring Headroom with Big-BranchNet

Fig. 9 shows the MPKI reduction of using Big-BranchNet along with MTAGE-SC, the best predictor in the unlimited storage category of CBP 2016 [7]. Adding Big-BranchNet to MTAGE-SC reduces the average MPKI from 3.42 to 3.16 (7.6% reduction). Big-BranchNet improves the overall MPKI by predicting only a few static branches that are hard-to-predict. 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, Fig. 9 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.

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

C. Characteristics of Improved Branches

To better understand why BranchNet outperforms TAGE predictors, we have examined the source code of some of the most improved branches in *mcf* and *leela*. We describe our 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.

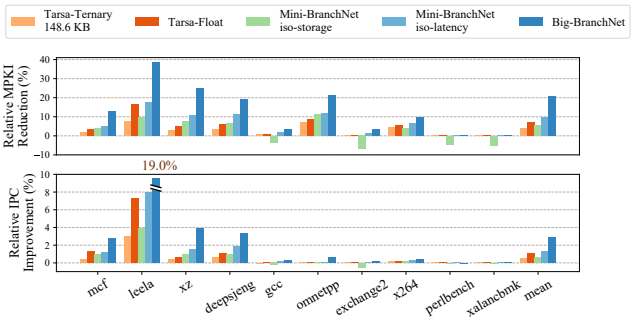

Fig. 11. MPKI and IPC improvement of BranchNet and other CNN branch predictors compared to 64KB TAGE-SC-L.

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 and learn the relationships.

leela spends most of its execution time in 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 form of trained models vary (e.g., the number of required filters, the nonlinear function, the minimum history length), they are conceptually similar to the example we provided in Section IV.

Fig. 10 shows the accuracy of the 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 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.

D. Practical Mini-BranchNet Results

Fig. 11 shows the MPKI and IPC improvement of BranchNet and the CNN branch predictor of Tarsa et al. [13] compared to a 64KB TAGE-SC-L baseline. We disable 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, we try all possible assignments of top hard-to-predict branches to configurations and use the best combination of models across all SPEC benchmarks.

We 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)

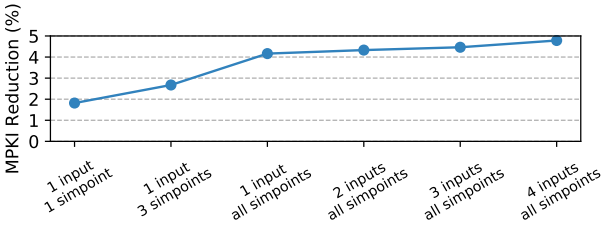

Fig. 12. Sensitivity of Big-BranchNet to the training set size.

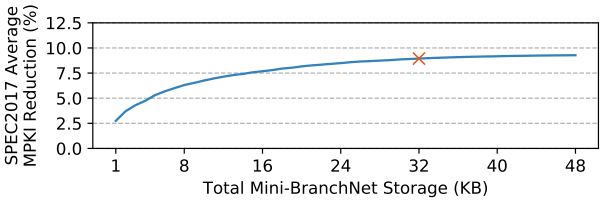

Fig. 13. Sensitivity of iso-latency Mini-BranchNet to its storage budget on SPEC2017 benchmarks.

with a 56KB TAGE-SC-L[6], 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 improvement, up to 19.0% for the best benchmark.

We evaluated two configurations of Tarsa's CNNs. Tarsa-Float is an oracular software model, analogous to Big-BranchNet. 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). As discussed in Section V-D, Mini-BranchNet architecture and optimizations allow it to use longer histories and a deeper network with less storage. Thus, as Fig. 11 shows, BranchNet is significantly more accurate than Tarsa's CNNs.

E. Sensitivity Analysis

Fig. 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. However, once the coverage is enough to expose all input-independent correlations, using additional inputs shows diminishing returns.

Fig. 13 shows the sensitivity of iso-latency Mini-BranchNet to its storage budget. Since storage more than 32KB shows diminishing returns, we chose 32KB as the budget for iso-latency Mini-BranchNet.

Table IV illustrates the negative impact of various constraints and approximations needed to make Mini-BranchNet

[6]We build the 56KB TAGE-SC-L by decreasing the number of table entries and tag bits of TAGE.

TABLE IV
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 %

practical. Quantization of convolution layers has the least significant impact on MPKI reduction, which agrees with our intuition 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 what we reported in earlier sections.

F. Weaknesses and Future Directions

The poor performance of BranchNet on *gcc* highlights the first weakness: if the mispredictions of a program are distributed among many static branches, BranchNet cannot significantly improve its accuracy by improving the prediction 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 [47] 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.

The large gap between the accuracy of Big-BranchNet and Mini-BranchNet is another weakness that can be improved. Training multi-layer neural networks often relies on some degree of overparameterization, i.e., there is redundancy in trained models. Regularization and pruning are machine-learning tools to combat this inefficiency. Furthermore, static analysis and input preprocessing can help to learn even more specialized prediction functions for branch prediction [48].

Finally, perhaps the biggest weakness of BranchNet is data-dependent branches. Our goal for BranchNet is to improve branch prediction using the global branch history. However, 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.

VII. OTHER RELATED WORK

Store-Load-Branch (SLB) predictor [49] and Probabilistic Branch Support (PBS) [50] improve branch prediction for data-dependent and probabilistic branches. Although the goal of SLB and PBS is different from BranchNet, they all break the runtime abstraction around branch prediction. SLB uses the compiler to identify data-dependent branches and exposes their dependence chains to the branch predictor. PBS uses programmer directives to change program semantics for probabilistic branches to simplify branch prediction. While different from profiling, these approaches agree with our general assertion that we need to revisit compile-time support for branch prediction.

Gope and Lipasti [51] 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 runtime filtering mechanism. However, offline deep learning allows BranchNet to be more powerful.

Evers et al. [52] describe how identifying correlated branches in the history is useful for improving branch prediction accuracy. To this end, Thomas et al. [53] use an on-chip mechanism to track dataflow dependencies to identify correlated branches in the history. However, their mechanism cannot track dataflow through memory. This is particularly problematic for identifying correlations that appear deep into a long history because dataflow dependencies through memory become more likely.

Seznec et al. [54] propose using Inner Most Loop Iteration (IMLI) counters to identify correlated branches in history. Inspired by the Wormhole predictor [55], 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.

VIII. CONCLUSION

BranchNet is a convolutional neural network that we train 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 in the 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 use only the correlated branches to make a prediction. To show the inherent advantage of CNNs in predicting this category of branches, we have compared Big-BranchNet to MTAGE-SC without considering practical constraints. We have shown that 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, we have compared Mini-BranchNet to 64KB TAGE-SC-L. Without increasing the prediction latency, Mini-BranchNet reduces the MPKI by 9.6%. While the IPC gains of Mini-BranchNet are limited (average 1.3%, up to 7.9%), 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 research and innovation can complement BranchNet to remedy the remaining weaknesses of runtime predictors.

ACKNOWLEDGMENT

We thank the anonymous reviewers, members of HPS research group, and Yongkee Kwon for their feedback and help with improving this paper. We thank Intel, Arm, and Microsoft for their financial support. We acknowledge the Texas Advanced Computing Center (TACC) for providing compute resources.

REFERENCES

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

[2] 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.

[3] 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

[4] 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.

[5] 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.

[6] 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

[7] 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)*, Seoul, South Korea, Jun. 2016, p. 4. [Online]. Available: https://hal.inria.fr/hal-01354251

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

[9] ——, "Multiperspective perceptron predictor with tage," in *5th JILP Workshop on Computer Architecture Competitions (JWAC-5): Championship Branch Prediction (CBP-5)*, 2016.

[10] 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.

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

[12] 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.

[13] 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.

[14] 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

[15] 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.

[16] 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.

[17] 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

[18] 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

[19] 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

[20] 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

[21] 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.

[22] 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.

[23] 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.

[24] 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

[25] 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.

[26] S. Mcfarling, "Combining branch predictors," Digital Equipment Corporation, Western Research Lab, Tech. Rep., 1993.

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

[28] 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

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

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

[31] 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. Stockholmsmssan, Stockholm Sweden: PMLR, 10–15 Jul 2018, pp. 1919–1928. [Online]. Available: http://proceedings.mlr.press/v80/hashemi18a.html

[32] 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

[33] 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

[34] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, *Learning Representations by Back-Propagating Errors*. Cambridge, MA, USA: MIT Press, 1988, p. 696699.

[35] 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.

[36] 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

[37] 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.

[38] 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.

[39] 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.

[40] 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

[41] 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

[42] 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.

[43] 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

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

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

[46] 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.

[47] 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. 157167. [Online]. Available: https://doi.org/10.1145/1346281.1346301

[48] 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.

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

[50] 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.

[51] 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.

[52] 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. 5261. [Online]. Available: https://doi.org/10.1145/279358.279368

[53] 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. 314323. [Online]. Available: https://doi.org/10.1145/859618.859655

[54] 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

[55] 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*, ser. MICRO-47. Washington, DC, USA: IEEE Computer Society, 2014, pp. 509–520. [Online]. Available: http://dx.doi.org/10.1109/MICRO.2014.40