Improving Branch Prediction By Modeling Global History with Convolutional Neural Networks

Stephen J. Tarsa, Chit-Kwan Lin, Gokce Keskin, Gautham Chinya, Hong Wang
Intel Corporation
Santa Clara, CA, USA
{stephen.j.tarsa, chit-kwan.lin}@intel.com

***Abstract*—CPU branch prediction has hit a wall—existing techniques achieve near-perfect accuracy on 99% of static branches, and yet the mispredictions that remain hide major performance gains. In a companion report, we show that a primary source of mispredictions is a handful of systematically hard-to-predict branches (H2Ps), e.g. just 10 static instructions per SimPoint phase in SPECint 2017. The lost opportunity posed by these mispredictions is significant to the CPU: 14.0% in instructions-per-cycle (IPC) on Intel SkyLake and 37.4% IPC when the pipeline is scaled four-fold, on par with gains from process technology. However, up to 80% of this upside is unreachable by the best known branch predictors, even when afforded exponentially more resources.**

New approaches are needed, and machine learning (ML) provides a palette of powerful predictors. A growing body of work has shown that ML models are deployable within the microarchitecture to optimize hardware at runtime, and are one way to customize CPUs post-silicon by training to customer applications. We develop this scenario for branch prediction using convolutional neural networks (CNNs) to boost accuracy for H2Ps. Step-by-step, we (1) map CNNs to the global history data used by existing branch predictors; (2) show how CNNs improve H2P prediction in SPEC 2017; (3) adapt 2-bit CNN inference to the constraints of current branch prediction units; and (4) establish that *CNN helper predictors* are reusable across application executions on different inputs, enabling us to amortize offline training and deploy ML pattern matching to improve IPC.

I. Introduction

CPU branch predictors enable speculative execution and are a critical tool for hiding latency in out-of-order cores. They work by inferring the unresolved direction of a branch instruction when it is fetched, based on a model trained to previously observed directions. Today, branch prediction units (BPUs) perform both prediction and training *online* within a CPU's front-end, as an application runs. Though tightly constrained, e.g. in storage and latency, existing predictors achieve >99% accuracy on 99% of static branch instructions in SPEC 2017 [1]–[3].

However, the mispredictions that remain hide a major performance upside. Our data [3] shows that a small number of static branch instructions, just 10 on average per SPECint 2017 SimPoint phase, are systematically mispredicted. Improving accuracy on these hard-to-predict branches (H2Ps) would boost instructions per cycle (IPC) on an Intel SkyLake core up-to 14.0%, and up-to 37.4% on a projected future CPU pipeline with width and depth scaled by $4\times$. But when the best-known branch predictors are afforded exponentially more resources, 80% of this opportunity remains untapped. New approaches are needed to extract this performance, which lies in just a handful of static branches in each application.

For the first time, branch prediction poses an attractive deployment scenario for machine learning (ML). Gains in branch predictors over past decades have balanced strict BPU constraints with the need for high accuracy on thousands of static branches at a time. Solutions have favored simple, lightweight pattern-matching [2]–[4], while comparatively powerful, yet expensive ML models such as deep neural networks, support vector machines, and random forests have been left unexplored. The large IPC opportunity that remains, and its concentration in a few H2Ps that resist existing techiques leads us to pursue ML models that implement more sophisticated pattern matching within the BPU.

We propose ML-driven *helper predictors* that operate alongside a baseline predictor to boost accuracy for individual H2Ps. This report provides a tutorial developing convolutional neural network (CNN) helpers to improve pattern matching on the same global history data used by existing branch predictors. We show how convolutional filters better tolerate distortions in history data caused by control structures with variable iteration counts. We then train CNN helpers with 2-bit weights and translate their inference procedure into a small number of table lookups that meet BPU constraints. Finally, we evaluate CNN helpers on applications traced over multiple inputs to establish that gains hold in future executions. At full precision, CNN helpers reduce mispredictions by an average of 36.6% on 47% of H2Ps in SPECint 2017; our implementable design improves 24% of H2Ps by 14.4%.

We adopt a deployment scenario in which helpers are trained to runtime data *offline* and uploaded to the BPU in future application executions to generate predictions *online* [5], [6]. This approach amortizes training over the lifetime of a device and across devices that run the same application, e.g. in a datacenter. The result is an application-specific IPC boost that requires no access to source code or painstaking expert analysis. Given a rich set of ML helper predictors, we envision an optimization service that automatically fits the best helper to each H2P and packages those that maximize IPC as application metadata. CNN helpers solve one source of systematic misprediction, and we intend this report as a blueprint for the development of other ML-driven helpers.

2nd ISCA International Workshop on AI Assisted Design for Architecture (AIDArc), June 2019, Phoenix, AZ, USA

```c
 1  int f(int k, int *uvec, int *vvec) {
 2    int val1 = 0;
 3    int val2 = 0;
 4
 5    if (uvec[k] % 3 > 0) /*Data-Dependent Branch*/
 6      val1 += 1;
 7
 8    for(int j = 0; j < (vvec[k]); j++)
 9      if (vvec[j] % 2 > 0) val2 += vvec[j];
10
11    if (val1 > 0)         /*H2P-1*/
12      return val2;
13    return 0;
14  }
```

Listing 1: A simple C function illustrates how common program structures cause systematic branch mispredictions.

II. Mispredictions Due to Variable Iteration Control Structures

We motivate CNN helpers by showing one class of H2P that arises due to control structures with data-dependent iteration counts. Two examples, one illustrative and the other drawn from `deepsjeng` in SPEC 2017 demonstrate that this common motif causes *positional variations* in data available to the BPU. Such distortions confound state-of-the-art predictors that rely on exact sequence matching or positional correlations, but are tolerated by convolutional filters. These examples are predicated on the following:

- We consider conditional branches only;
- When a branch is fetched, its **global history** is the sequence of instruction pointer values (IPs) and the directions of branches executed leading up to the current instruction;
- **TAGE-SC-L** is the state-of-the-art branch predictor [2]. It conditions each prediction on the longest recognized subsequence of global history by approximating Partial Pattern Matching (PPM) [7]. It is implemented by hashing global history subsequences into tagged tables. Table entries hold a saturating counter that tallies previously observed directions, and can be thresholded to make a prediction. TAGE-SC-L also implements a loop predictor, arbitrating between this and the longest-matching PPM predictions using a *statistical corrector*, itself a perceptron;
- The **perceptron *predictor*** (distinct from the statistical corrector above) is an alternative to PPM predictors that trains weights for each global history position, isolating directions correlated to the current prediction [8]. This mechanism filters noisy data that affects TAGE-SC-L's exact-match hash lookups, but requires positional weights to be stored and retrieved for many branches;
- We identify as **H2Ps** any branch predicted to <99% accuracy under TAGE-SC-L, and which generates at least 1,000 mispredictions per 30M-instruction SimPoint [3], [9].

Illustrative Example – Listing 1 showcases an H2P (Line 11, H2P-1) whose global history is affected by a loop with a variable iteration count. H2P-1 is exactly correlated to the data-dependent branch at Line 5, and both branches are biased to be taken 33% of the time when `uvec`'s values are uniformly distributed. Crucially, they are separated by a loop whose iteration count depends on data. This code contains a simple, stable relationship that predicts H2P-1—with no additional information on data values, these two branches should be predicted to 66% and 100% accuracy, respectively.

```c
 1  typeA_t func1(typeB_t *a, const int b) {
 2    // ...
 3    if (Arr23[v4] & v2) { v1 |= (v2 & func1b(A2, b)); }
 4    if (Arr21[b] & v3) { v1 |= (v3 & func1c(A2, b)); }
 5    if (Arr22[b] & v3) { v1 |= (v3 & func1d(A2, b)); }
 6    return v1;
 7  }
 8
 9  int func2(typeB_t *a, const int b, const int c, const int c, const int d) {
10    // ...
11    lc = func1(a, c);
12    // ...
13    while (lc) {
14      if (v8 == CONST_6) {
15        v1 = Arr2 & lc;
16        if (v1) { v7 = CONST_1; }
17        else {
18          v1 = Arr6 & lc;
19          if (v1) { v7 = CONST_2; }
20          else {
21            v1 = Arr8 & lc;
22            if (v1) { v7 = CONST_3; }
23            else {
24              v1 = Arr4 & lc;
25              if (v1) { v7 = CONST_4; }   /* CORRELATED BRANCH A */
26              else {
27                v1 = Arr10 & lc;
28                if (v1) { v7 = CONST_5; }
29                else {
30                  v1 = Arr12 & lc;
31                  if (!v1) { break; }
32                  v7 = CONST_0;
33                }
34      // ...
35      } else {
36        //...6 more nested correlated branches (not shown)...
37      }
38      // ...
39      if (v4 & Arr29[c]) { lc |= func2a(a, c) & v4; }    /** H2P BRANCH **/
40      if (v5 & Arr30[c]) { lc |= func2b(a, c) & v5; }
41      lc &= a->A2;
42      // ...
43    }
44    // ...
45  }
```

Listing 2: An H2P in `deepsjeng` (Line 39, red) has many correlated branches (orange, yellow). The loop condition is set on Line 11 (blue) but modified over time (Lines 39–41), causing correlated branches to shift position in the H2P's global history.

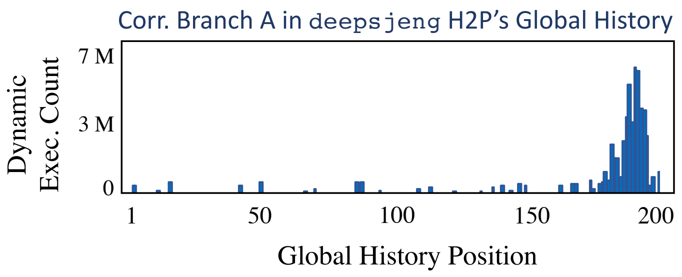

Fig. 1: Histogram of Correlated Branch A's position (Listing 2, Line 25, yellow), which exhibits positional variations in an H2P's global histories. History position 200 is the most recent.

When a simple program calls function f repeatedly with random inputs, H2P-1's global histories exhibit significant variations. The loop at Line 8 injects different numbers of uncorrelated branches into history data, causing the position of the predictive data-dependent branch to change relative to H2P-1. This positional variation explodes the number of unique histories memorized by a PPM predictor and breaks perceptron predictors that require positional consistency. Consequently, TAGE-SC-L predicts H2P-1 with 68% accuracy, storing statistics in table entries corresponding to all tracked subsequence lengths, while reusing few for prediction. Training a perceptron on H2P-1's global history gives a similar 69% accuracy. In contrast, a CNN helper predicts H2P-1 with 100% accuracy (see Section III).

SPECint 2017 deepsjeng – Positional variations appear in the wild in SPECint 2017, for example in `deepsjeng`. Listing 2 shows a code fragment from the `deepsjeng` source containing an H2P branch at Line 39 (red). Lines 3-5, 13,

Source Code\Machine Code	IP LSBs	Not Taken Index	Taken Index
`5: if (uvec[k] % 3 > 0)`			
`400585 test %eax,%eax`	... 0000111	14	15
`400587 jle,400590`			
`8: for(...;j < (vvec[k]);...)`			
`400627 cmp -0x4(%rbp), %eax`	... 0101010	84	85
`40062a jg 400599`			
`9: if (vvec[j] % 2 > 0)`			
`4005cb cmp -0x4(%rbp),%eax`	... 1001101	154	155
`4005cd jle 4005e8`			
`11: if (val1 > 0)`			
`400630 cmpl $0x0,-0xc(%rbp)`	... 0110100	104	105
`400634 jle40063b`			

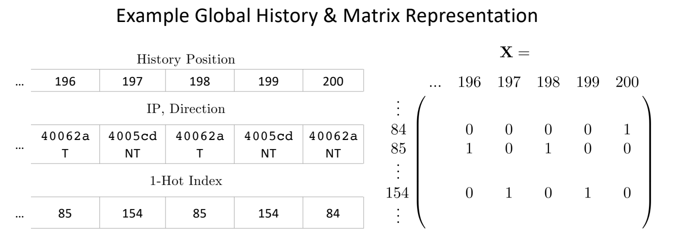

Fig. 2: A CNN is fed H2P-1's global history as a matrix of 1-hot columns with 1's in indices $((IP \ll 1) + Dir)\&(2^p - 1)$.

```
class TwoLayerFullPrecisionBPCNN(chainer.Chain):

  # Network definition
  def __init__(self, tableSize=256, numFilters=2, historyLen=200):
    super(TwoLayerFullPrecisionBPCNN, self).__init__()
    with self.init_scope():
      # Embed via identity matrix:
      self.E  = cupy.identity(tableSize, dtype=cupy.float32)

      # Convolution layer, 1D in effect; Linear layer:
      self.c1 = L.Convolution2D(16,  numFilters, (1, 1))
      self.l2 = L.Linear(historyLen*numFilters, 1)

  # Forward pass
  def __call__(self, x):
    # Embed by expanding sequence of ((IP << 7) + dir) & (255)
    # integers into 1-hot history matrix during training:
    xFull = cupy.rollaxis( cupy.take(self.E, x.data, axis=0), 3, 1)

    h1  = self.c1(xFull)
    h1a = tanh(h1)
    h2  = self.l2(h1a)

    return F.sigmoid(self.h2)
```

Listing 3: A simple CNN gives 100% accuracy for H2P-1.

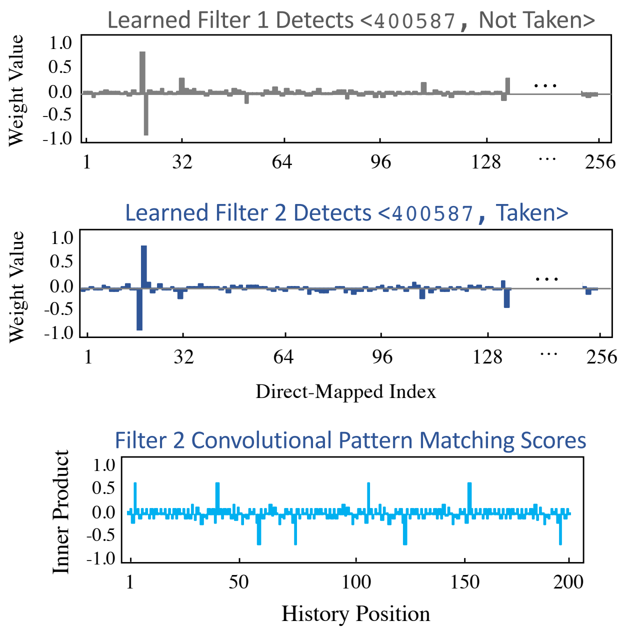

Fig. 3: (Top 2) 1-wide convolutional filters trained on H2P-1's history learn large weights at indices of the data-dependent branch; (Bot.) convolutional pattern matching gives large values at positions where that branch appears in H2P-1's global history.

14, 16, 19, 22, 25, 28, 31, and 40 (orange and yellow) show correlated branches in the H2P's global history. They all reside within a loop conditioned on variable `lc`, whose value is initialized at Line 11 but modified over iterations (Lines 39-41). As a result, the loop iteration count is variable and correlated branches shift position in the H2P's global history.

This is evident on examination of just one of the correlated branches, Correlated Branch A (Line 25, yellow). Fig. 1 shows that the distribution of its position in the H2P's histories roughly spans the most recent 25 positions. Increasing global history subsequence length, an optimization made when scaling TAGE-SC-L from 8KB to 64 KB, does not directly address this variation, which is the root cause of this H2P. Positional variations are also exhibited by all other (orange) correlated branches in Listing 2. As a result, we find that TAGE-SC-L predicts the H2P on Line 39 with just 93.8% accuracy, while a CNN will predict it with 100% accuracy.

III. A CNN Global History Model

To show how a CNN predicts H2P-1, we first walk through the forward pass of the two-layer CNN in Listing 3 to produce a prediction, and then describe how it is trained. We initially use the full network representation, but in Section IV translate it into a mechanism that meets BPU constraints.

A. *Encoding History Data*

Given a dynamic instance of an H2P, we convert its global history sequence of ⟨IP, direction⟩ tuples into an algebraic vector representation. IPs are discrete and take on a large number of possible values, so we use a hash function to index into a "1-hot" vector, which contains a one at the index position and zeros otherwise. For example, setting vector dimension to $2^p = 256$, we map each tuple to $[0, ..., 255]$ by concatenating the observed-direction bit onto the $p-1$ LSBs of the IP: $((IP \ll 1) + Dir)\&(2^p - 1)$. This process is shown in Fig. 2 for H2P-1. Four branches from Listing 1 are shown alongside their IP values, observed directions, and the indices used to generate 1-hot vectors. We concatenate column vectors to form a global history matrix X, which is input to the CNN.

Though 1-hot history matrices appear costly in terms of storage, we choose this encoding because matrices can be replaced on-BPU with direct-mapped table lookups (Section IV). Our experiments show that our CNNs perform well with as few as seven LSBs from each IP, making them agnostic to an application's base virtual address. To ensure history encodings behave consistently across executions, we set $p = 8$.

B. *Layer 1: Convolutional Correlation*

CNNs perform pattern matching using inner-product computations between a data vector x and a weight vector w, also called a *filter*, with an optional bias term b. A similar computation is used in perceptron predictors, however our

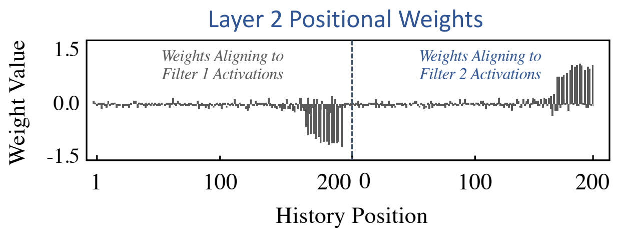

Fig. 4: Layer 2 filter weights represent how much each history position contributes to the final prediction.

CNNs differ by performing the same filter matches at every history position, and by also matching on IPs.

$$y = \sum_i w_i x_i + b. \tag{1}$$

To illustrate, we instantiate our CNN with two filters and plot their values in Fig. 3 (Top) after training on history matrices and observed directions for H2P-1. We see that Filter 1 learns a large positive weight at index 14, aligning to correlated branch `0x400587` being not-taken, while Filter 2 exhibits a large weight at index 15 for the same IP being taken. Small weights adjust for branches that are biased in H2P-1's history, though magnitudes are negligible in comparison.

Evaluating Eq. 1 for each filter against each column of the history matrix produces $200*2$ inner product scores for history length 200. Fig. 3 (Bottom) shows the 200 scores computed from Filter 2. We call `f()` from a loop, so H2P-1's global history also contains stale appearances of the correlated data-dependent branch, and each produces a large filter response.

C. Layer 2: Positional Prediction

Scores computed in the convolution layer above are passed to a perceptron-like *linear layer*, which contains a single filter that is matched against the output of Layer 1. The trained weights of Layer 2's filter are shown in Fig. 4. Near-zero weights damp positions beyond the most recent 30, filtering out stale appearances of IP `0x400587`. Eq. 1 is applied once at Layer 2 using its filter and the Layer 1 scores as inputs.

D. Stacking the Layers Together

The result of Layer 2's pattern matching operation predicts "taken" if greater than zero and "not-taken" otherwise. The two layers of this CNN handle different aspects of predicting H2P-1's direction: the first layer is position-agnostic and is designed to identify which ⟨IP, direction⟩ tuples in a branch history correlate highly with the H2P's direction; the second layer is designed to identify which positions in a branch history contribute most to the prediction. The combined filtering action of these stacked layers allows the CNN to precisely latch onto the predictive signal in H2P-1's histories, even as it shifts position—*it is this mechanism, the result of stacking convolutional and linear layers, that gives our CNNs a pattern matching advantage over PPM and perceptron predictors.*

E. Offline Training

The training dataset for a CNN helper consists of history matrices of an H2P alongside its observed directions, which we collect using the Pin binary instrumentation tool [10]. We train networks using Chainer [11] and find, for the CNN configurations used in Section V, 5,000 history-matrix/H2P-direction pairs sampled uniformly from runtime data are sufficient to converge in 40 epochs using the Adam optimizer [12].

```
class TwoLayerTernaryBPCNN(chainer.Chain):

  # Network definition
  def __init__(self, tableSize=256, numFilters=2, historyLen=200):
    super(TwoLayerTernaryBPCNN, self).__init__()
    with self.init_scope():
      # Embed via identity matrix:
      self.E  = cupy.identity(tableSize, dtype=cupy.float32)

      # Ternary convolution layer, now with batch normalization
      self.c1 = TernConv2D(tableSize, numFilters, kernel=(1,1))
      self.b1 = BatchNormalization(numFilters)

      # Ternary linear Layer, now with batch normalization
      self.l2 = TernLinear(numFilters*historyLen, 1, nobias=True)
      self.b2 = L.BatchNormalization(1)

  # Forward pass
  def __call__(self, x):
    # Embed by expanding sequence of ((IP << 7) + dir) & (255)
    # integers into 1-hot history matrix during training:
    xFull = cupy.rollaxis( cupy.take(self.E, x.data, axis=0), 3, 1)

    h1  = self.c1(xFull)
    h1n = self.b1(h1)
    h1t = F.TernQuant(h1n)

    h2  = self.l2(h1t)
    h2n = self.b2(h2)

    return F.sigmoid(self.h2n)
```

Listing 4: CNNs can be constrained to have 2-bit weights during training by adding quantization and normalization steps.

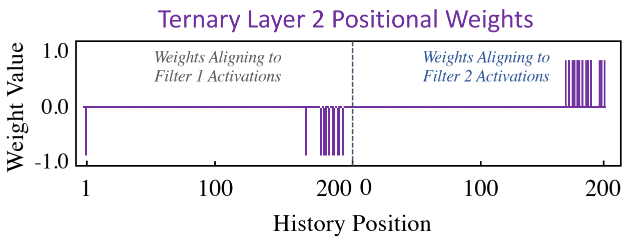

Fig. 5: 2-bit CNN helpers lose fidelity encoding the magnitude of each position's contribution to predictions, but accurately detect ⟨IP, direction ⟩ tuples despite positional variations.

IV. On-BPU Inference with 2-Bit CNNs

To deploy our CNN helper in a BPU, we train networks with 2-bit weights and show that they need only modest on-chip storage and bitwise-parallel logic at prediction time. CNNs provide strong pattern recognition even when their weights are constrained to values in $\{+1, 0, -1\}$ [13], [14], allowing logical operations to replace arithmetic during inference.

Following Courbariaux *et al.* [13], we impose low-precision constraints during training by clipping weights to $[-1, +1]$, normalizing activations, and quantizing during forward CNN computations (Listing 4). We train the resulting *ternary* CNN helper for H2P-1 on the same training data. Fig. 5 shows ternary Layer 2 weights. Compared to the full-precision weights in Fig. 4, quantized weights lose accuracy encoding the magnitude of each position's contribution to predictions, but still detect correlated ⟨IP, direction⟩ tuples and damp stale data. This ternary CNN helper yields 98% accuracy for H2P-1.

Not only does accuracy remain high, but ternary CNN inference for our network can also be made far more efficient than its full-precision counterpart based on the following observations:

Multiplication by a 1-hot vector yields a scalar. The inner product of a 1-hot input vector and a filter is the filter value that aligns to the input's sole non-zero. We therefore sidestep history matrices completely on-BPU by indexing ⟨IP, direction⟩ tuples into a table of filter values. Subsequent normalization and quantization steps can also be folded into this table, since they produce a 2-bit value from each possible filter value after training. We precompute this half of the inference computation for any input by populating a lookup table as follows: For m filters of length 2^p, denoted $W = [w_1, \ldots, w_m]$; indices $i \in [1, 2^p], j \in [1, m]$; learned parameters $\mu_{1j}, \sigma_{1j}, \gamma_{1j}, \beta_{1j}$ from a normalization layer that transforms data according to $\hat{y_j} = (y_j - \mu_{1j})(\gamma_{1j}/\sigma_{1j}) + \beta_{1j}$; and quantization bins defined over the ranges $[-1, -q]$, $[-q, +q]$, $[q, 1]$, we populate a $2^p \times m \times$ 2-bit table $\mathcal{T}$ as:

$$\mathcal{T}[i,j] = \begin{cases} 01, \text{ if } w_{ij} < \frac{-\beta_{1j}}{\gamma_{1j}}\sigma_{1j} + \mu_{1j} - q \\ 11, \text{ if } w_{ij} > \frac{-\beta_{1j}}{\gamma_{1j}}\sigma_{1j} + \mu_{1j} + q \\ 00, \text{ otherwise.} \end{cases} \tag{2}$$

q defines quantization buckets for ternary CNN weights [13], [14]; we set $q = 0.8$ but note its value may be learned [15].

1-wide convolutions can be computed as soon as history tuples are available. When applying convolutions of width 1, filter responses for each history position are independent of their neighbors. This allows us to retrieve the Layer 1 responses well before an H2P is encountered, as ⟨IP, direction⟩ tuples become available. Whenever a prior branch's direction is predicted, the corresponding Layer 1 responses are retrieved from $\mathcal{T}$ and pushed into a FIFO buffer. When an H2P is fetched and a CNN prediction needed, Layer 1 outputs are already available in the FIFO buffer.

Ternary inner products require only bitwise parallel logic, popcount, and subtraction. At prediction time, we evaluate Layer 2 and its normalization layer. This entails an inner product between the FIFO buffer's contents and ternary weights, scaling and shifting the resulting integer value by learned normalization parameters, and comparing to 0 to give a "taken" or "not-taken" prediction. We implement the ternary inner product as:

$$\begin{aligned} P = \texttt{popcount}(\neg(L1_S \wedge L2_S) \& (L1_V \& L2_V)) - \\ \texttt{popcount}((L1_S \wedge L2_S) \& (L1_V \& L2_V)) \end{aligned} \tag{3}$$

where $L1_S$ and $L1_V$ are the sign and value bits of the FIFO buffer, respectively, and $L2_S$ and $L2_V$ contain those for the Layer 2 filter. We apply the inverse of normalization to 0 to solve for a threshold t, above which we predict taken:

$$\text{Pred} = \begin{cases} 1, \text{ if } P > t, \text{where } t = \frac{-\sigma_2}{\gamma_2}\beta_2 - \mu_2 \\ 0, \text{ otherwise} \end{cases} \tag{4}$$

On-BPU CNN inference thus consists of two steps, defined by Algorithms 1 and 2. The first is a table lookup to update the FIFO buffer of Layer 1 filter responses whenever any dynamic conditional branch is fetched. The second is a ternary inner product between the FIFO buffer and Layer 2 filter when the H2P is fetched and a prediction needed. Any time a branch is mispredicted, the CPU is rolled back to that instruction, and wrong-path entries are simply shifted off the FIFO buffer.

Algorithm 1 FIFO Update

```
Require: ⟨IP, Dir⟩ of most recent branch, $\mathcal{T}$ as in Eq. 2; L1, a
    $200 \times m \times 2$ bit buffer; $p = 8$.
1: function CNNHistoryUpdate(IP, Dir)
2:     i ← (IP ≪ 1 + Dir) & $(2^p - 1)$
3:     L1 ← (L1 ≪ 2*m) | $\mathcal{T}$[i, 0...2*m]
4: end function
```

Algorithm 2 2-Bit CNN Branch Prediction

```
Require: L1, a $200 \times m \times 2$ bit buffer filled per Alg. 1; L2, a
    $200 \times m \times 2$ bit buffer of Layer 2 weights; $t$ as in Eq. 4
1: function CNNPredict
2:     s_bits ← $L1_S \wedge L2_S$
3:     v_bits ← $L1_V$ & $L2_V$
4:     P ← popcount(¬(s_bits & v_bits))
           - popcount(s_bits & v_bits)
5:     return (P > t)
6: end function
```

A. *On-BPU Storage and Latency*

To install a CNN helper in a BPU, we must store four components: (1) a $2^p \times m \times$ 2-bit table to hold Layer 1 filter responses; (2) a `historyLen` $\times$ m $\times$ 2-bit FIFO buffer to hold convolution results; (3) a `historyLen` $\times$ m $\times$ 2-bit buffer to hold the Layer 2 weights; (4) a buffer to hold the precomputed integer threshold. Our network, with $p = 8$, $m = 2$, and `historyLen` $= 200$, requires 336 bytes per helper. For $m = 32$, storage is 5.2KB.

While a full layout of our CNN helper is beyond the scope of this report, we can compare the relative latency between ternary CNN inference and TAGE-SC-L by analyzing the computation graphs of their prediction procedures (Table II). For example, in Algorithm 2, we are able to compute Lines 2–3 in parallel, compute (s_bits & v_bits) on Line 4 and ¬ operations serially, the `popcounts` in parallel, and finally the subtract and comparison serially. Predictions from a 2-bit CNN helper thus require six serial computations. The bottleneck computation is `popcount`, which requires a 13 or 15 stage circuit depending on m [16]. In contrast, TAGE-SC-L 8KB and 64KB require 34 and 32 serial computations, respectively (TAGE-SC-L 8KB uses more complex hashing). Their bottleneck computations are back-to-back lookups to 4k and 8k entry tables, depending on predictor. This comparison shows that a ternary CNN helper requires a similar number of computation steps to existing predictors.

V. CNN Helper Gains & Reusability

We demonstrate CNN helpers on SPECint 2017 and assess reusability with the dataset of [3], which traces each benchmark over multiple inputs. For each benchmark, we screen for H2Ps using TAGE-SC-L 8KB as the baseline predictor in the Championship Branch Prediction 2016 simulator [4], and train a CNN helper for any H2P appearing in 3 or more application inputs (i.e. workloads) to support k-fold cross-validation. We train on data from the entirety of a single workload and report performance averaged across all held-out workloads; this constitutes one fold, and we average all possible folds to compute the *expected* gains in future executions, assuming

SPECint2017 Benchmark	# Training Folds	# H2Ps (All Phases)	FP-CNN with TAGE 8KB Baseline		TP-CNN with TAGE 8KB Baseline		FP-CNN, Gains Beyond TAGE 64KB	
			% Winners	Mispred. Red. per H2P	% Winners	Mispred. Red. per H2P	% Winners	Mispred. Red. per H2P
600.perlbench_s	4	16	51%	63.2%	18%	26.6%	4%	8.2%
605.mcf_s	8	20	55%	44.8%	28%	27.9%	35%	19.3%
620.omnetpp_s	5	28	71%	33.6%	30%	16.3%	24%	11.2%
623.xalancbmk_s	4	8	39%	27.4%	0%	0.0%	23%	12.8%
625.x264_s	14	7	44%	16.8%	35%	12.0%	33%	12.2%
631.deepsjeng_s	12	49	56%	31.2%	24%	10.0%	12%	15.3%
641.leela_s	10	68	68%	40.7%	44%	15.3%	41%	19.7%
645.exchange2_s	5	19	9%	46.5%	4%	6.0%	0%	0.0%
657.xz_s	5	50	28%	25.2%	29%	15.4%	15%	12.3%
MEAN	7.3	29	47%	36.6%	24%	14.4%	21%	12.3%

TABLE I: CNN Helpers reusably improve accuracy for a large portion of H2Ps. Gains for 21% of H2Ps are beyond the capabilities of TAGE-SC-L when scaled by 8x.

	Prediction Generation Complexity			
	TAGE 8 KB	TAGE 64 KB	TP-CNN 8 filter	TP-CNN 32 filter
# Serial Computations	**34**	**32**	**6**	**6**
# Serial Tbl. Lkups.	**2**	**2**	**0**	**0**
Latency Limiting Computation	2× lookup, 4k-entry table	2× lookup, 8k-entry table	`popcount` *(13 stage circuit)*	`popcount` *(15 stage circuit)*

TABLE II: At prediction time, CNN Helpers perform a ternary inner product followed by popcount.

we train on data from an arbitrary execution. *Training on one workload and testing on the hold-outs demonstrates the reusability of our CNNs.* We evaluate full-precision CNN Helpers (FP-CNN) as a limit study and ternary CNNs (TP-CNN). For both, we use a history length of 200, encode 7 bits of each IP and 1 direction bit, and 32 Layer 1 filters.

Table I breaks out the portion of CNN helper predictors that improved H2P accuracy (% Winners) by benchmark, alongside the accuracy improvement per H2P (% Reduction in Mispredictions). On average, the FP-CNN shows that pattern matching with tolerance for positional variations improves accuracy on 47% of H2Ps by an average 36.6% reduction in mispredictions, *reusably across workloads*. When we use FP-CNN helpers *and* scale TAGE-SC-L to 64KB, we still find additional gains—21% of H2Ps improve by 12.3% on average, improving gains in mispredictions-per-kilo-instruction (MPKI) from 21.2% to 22.3% over the baseline. This shows one example when improved pattern matching provides a fundamental advantage over scaling existing algorithms.

TP-CNN helpers improve 24% of H2Ps by 14.4% on average, capturing roughly half the gain of FP-CNNs. Given that quantizing Layer 2 weights in TP-CNN (Fig. 5) tempers the positional precedence captured by a full-precision Layer 2 (Fig. 4), this comparison shows that arbitrating with potentially stale data is also an important contributor to prediction accuracy.

VI. Directions for Future ML Helpers

This paper details how a two-layer CNN reduces systematic branch mispredictions caused by positional-variations in global history data. We demonstrate a path to deployment that (1) meets on-BPU constraints for prediction generation, and (2) can amortize iterative batch training through reuse across application executions. Several natural future directions exist:

1) CNNs provide an expressive pattern matching framework and support rapid experimentation; exploring topologies, e.g. to learn predictive multi-IP subsequences or extract patterns from arbitrarily long global histories using recurrence can address different causes of misprediction;
2) The gap between FP-CNN and TP-CNN shows the need for alternative on-BPU designs, that, e.g., integrate dependent branch IPs identified by a CNN into lightweight predictors. In such a design, ML models act as an automated analysis tool, rather than an on-BPU predictor directly;
3) Feeding additional data such as register values into ML models may boost prediction accuracy for data-dependent branches. In this manner, an ML model acts as a approximate value predictor, possibly exploiting idle multiply-accumulate cycles in the core.

These avenues and others will provide fruitful ground for machine learning in branch predictor development.

References

[1] A Fog. The microarchitecture of intel, amd, and via cpus. *An Optimization Guide for Assembly Programmers and Compiler Makers. Copenhagen University College of Engineering*, 2018.

[2] A Seznec. TAGE-SC-L Branch Predictors Again. In *Proc. 5th Championship on Branch Prediction*, 2016.

[3] C-K Lin and SJ Tarsa. Branch Prediction is Not a Solved Problem: Measurements, Opportunities, and Future Directions. *arXiv:1906.08170*, 2019.

[4] CBP-5 Kit. In *Proc. 5th Championship on Branch Prediction*, 2016.

[5] GS Ravi and MH Lipasti. CHARSTAR: Clock Hierarchy Aware Resource Scaling in Tiled Architectures. *ACM SIGARCH*, 2017.

[6] SJ Tarsa, RBR Chowdhury, J Sebot, GN Chinya, J Gaur, K Sankaranarayanan, C-K Lin, R Chappell, R Singhal, and H Wang. Practical Post-Silicon CPU Adaptation Using Machine Learning. In *ISCA*, 2019.

[7] J Cleary and I Witten. Data Compression Using Adaptive Coding and Partial String Matching. *IEEE Trans Comms*, 1984.

[8] DA Jiménez. Multiperspective Perceptron Predictor. In *Proc. 5th Championship on Branch Prediction*, 2016.

[9] S Song, Q Wu, S Flolid, and J et al Dean. Experiments with SPEC CPU 2017: Similarity, Balance, Phase Behavior and Simpoints. Technical report, TR-180515-01, Dept. of ECE, UT-Austin, 2018.

[10] C-K Luk, R Cohn, R Muth, H Patil, A Klauser, G Lowney, S Wallace, VJ Reddi, and K Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. 2005.

[11] S Tokui, K Oono, S Hido, and J Clayton. Chainer: A Next-Generation Open Source Framework for Deep Learning. In *LearnSys*, 2015.

[12] D Kingma and J Ba. Adam: A Method for Stochastic Optimization. *arXiv:1412.6980*, 2014.

[13] 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. *arXiv:1602.02830*, 2016.

[14] M Rastegari, V Ordonez, J Redmon, and A Farhadi. XNOR-Net: Imagenet Classification Using Binary Convolutional Neural Networks. In *ECCV*, 2016.

[15] C Zhu, S Han, H Mao, and WJ Dally. Trained Ternary Quantization. *arXiv:1612.01064*, 2016.

[16] R Ramanarayanan, S Mathew, V Erraguntla, R Krishnamurthy, and S Gueron. A 2.1Ghz 6.5mW 64-bit Unified Popcount/Bitscan Datapath Unit for 65nm High-Performance Microprocessor Execution Cores. In *VLSID*, 2008.