Bit-level Perceptron Prediction for Indirect Branches

Elba Garza
elba@tamu.edu
Texas A&M University and AMD

Samira Mirbagher-Ajorpaz
Tahsin Ahmad Khan
parisamir@tamu.edu
tahsinkhan@tamu.edu
Texas A&M University

Daniel A. Jiménez
djimenez@tamu.edu
Texas A&M University and
Barcelona Supercomputing Center

ACM Reference Format:
Elba Garza, Samira Mirbagher-Ajorpaz, Tahsin Ahmad Khan, and Daniel A. Jiménez. 2019. Bit-level Perceptron Prediction for Indirect Branches. In *The 46th Annual International Symposium on Computer Architecture (ISCA '19), June 22–26, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/https://doi.org/10.1145/3307650.3322217

ABSTRACT

Modern software uses indirect branches for various purposes including, but not limited to, virtual method dispatch and implementation of switch statements. Because an indirect branch's target address cannot be determined prior to execution, high-performance processors depend on highly-accurate indirect branch prediction techniques to mitigate control hazards.

This paper proposes a new indirect branch prediction scheme that predicts target addresses at the bit level. Using a series of perceptron-based predictors, our predictor predicts individual branch target address bits based on correlations within branch history. Our evaluations show this new branch target predictor is competitive with state-of-the-art branch target predictors at an equivalent hardware budget. For instance, over a set of workloads including SPEC and mobile applications, our predictor achieves a misprediction rate of 0.183 mispredictions per 1000 instructions, compared with 0.193 for the state-of-the-art ITTAGE predictor and 0.29 for a VPC-based indirect predictor.

1 INTRODUCTION

As object-oriented languages have become ubiquitous within software application design, so have indirect branch instructions. Early work by Calder & Grunwald [1] shows that programs written in object-oriented languages like C++ contain many indirect branch instructions, on average 23 times as many compared to C programs, due to polymorphism. Polymorphism [2] uses dynamically-dispatched function calls implemented through indirect branches to support dynamic subtyping. Some object-oriented languages may generate a virtual function call for *every* polymorphic object call [3].

More recently, Kim et al. [4] examined Windows applications on real hardware, showing that 28% of branch mispredictions were due to indirect branches. For a some programs, indirect branches accounted for almost half the mispredictions measured. Since indirect branch instructions incur the same runtime misprediction penalty as conditional branch instructions, it is imperative to have accurate indirect branch prediction mechanisms in place.

Indirect branches are not just necessary for virtual function calls. Other common program constructs like switch-case statements, jump tables, function pointer calls, procedure returns, and interface calls also depend on indirect branching. Indirect branch instructions are inherently more difficult to predict than conditional instructions. Rather than simply predict *taken* or *not taken*, an indirect branch's prediction requires predicting a specific target address value. As an indirect branch instruction may lead to any number of different known branch target values at runtime, its prediction is more difficult than a simple decision between two values. Some control-flow breaks caused by indirect branches are predictable. Kaeli and Emma's call/return stack [5] is shown to accurately predict procedure returns' target address values.

Several software-based strategies have been proposed to reduce the prevalence of indirect branch instructions in programs [6–10]. Such *devirtualization* techniques lower the rate of indirect branch instructions by substituting them with one or more direct conditional branches [11]. Unfortunately, these methods are costly, requiring static analysis of the whole program [6, 12], extensive profiling [7, 8], or a combination of both [9, 10]. Additionally, type inference for C++ programs using static analysis is already a known NP-hard problem [13]. As such, indirect branch instructions are inevitable breaks in control flow that must be properly addressed. Section 2.1 talks about these software techniques in more depth.

An assortment of hardware-based techniques to predict indirect branch instruction targets have been proposed [4, 6, 14–22]. These predictors output a target prediction at runtime based on branch history, *i.e.* target address values from previous branch executions. The most well-known example of these is the branch target buffer, or BTB [14]. Many hardware-based indirect branch predictors maintain target values in dedicated storage [15–19, 21], which can account for significant die area and lead to increased power consumption. Other approaches take advantage of already-present hardware structures to facilitate indirect prediction [4, 22]. Section 2.2 discusses these hardware-based prediction strategies at length.

This paper introduces a new indirect branch prediction algorithm, Bit-Level Perceptron-Based Indirect Branch Predictor (BLBP). It uses branch history and perceptron-based learning [23, 24] to predict individual target value bits. The processor accesses a specialized BTB-like structure to select the target that matches most closely at the bit level among all observed targets of the branch.

ISCA '19, June 22–26, 2019, Phoenix, AZ, USA

ACM ISBN 978-1-4503-6669-4/19/06…$15.00
https://doi.org/https://doi.org/10.1145/3307650.3322217

The predictor is competitive with state-of-the-art indirect branch prediction schemes, reducing misprediction rates by 5% over ITTAGE [25] while maintaining the same hardware budget.

2 RELATED WORK

Control flow breaks due to indirect branch instructions can be mitigated by two main strategies. The first strategy requires software-based schemes: profiling and compiler optimizations can learn from programs to predict branch targets, or minimize the frequency of indirect branches. The other strategy depends on hardware mechanisms that predict branch target addresses dynamically. The following two sections detail these strategies' related work.

2.1 Indirect Branch Target Prediction Via Software

Wall proposed the first methods to statically predict indirect branches based on profiling [26]. Calder & Grunwald [6] also used static profiling along with compiler-driven optimizations and hardware modifications (see Subsection 2.2) to reduce C++ indirect function call overhead. Aigner & Hölzle proposed compiler optimizations that reduces the frequency of virtual function calls [7] using profile driven type feedback and class hierarchy analysis [12]. Porat et al. [10] introduced similar inlining optimizations. Roth et al. [27] devised a virtual function call prediction technique which uses precomputation to perform target address computation before execution time. Joao et al. [28] proposed Dynamic Indirect Jump Predication, using dynamic predication to handle hard-to-predict indirect branches without prediction. Farooq et al. [29] proposed Value-Based BTB Indexing, or VBBI identifies hard-to-predict indirect branches through profiling and flags via an ISA format augmentation, storing multiple targets for an indirect branch in the BTB. Compiler-Guided Value Pattern (CVP) prediction by Tan et al. [30] is similarly value-based and compiler-driven.

Software-based strategies are not free of hardware costs, however. Many works proposed above require supplemental hardware structures (even if minimal) and/or explicit ISA support [6, 28–31].

2.2 Hardware Indirect Branch Prediction

The simplest target prediction hardware mechanism is the Branch Target Buffer, or BTB [14], using the branch address to index and cache a branch's last-taken target address. The stored target address is later fetched for target prediction. A BTB is sufficient for *monomorphic* branches leading to just one target address, like direct branch instructions. However, because indirect branches can be *polymorphic*, or lead to different addresses, the BTB's last-used prediction strategy is often insufficient [16, 19]. Calder & Grunwald improve this scheme using a 2-bit Branch Target Buffer[6] that replaces a target in the BTB only after two consecutive mispredictions. However, BTB accuracy remains poor for indirect branches.

Path-based history is also correlated with branch behavior [15, 32, 33]. Unlike correlation with pattern history [34–36], path-based correlation uses the sequence of basic block addresses leading up to a branch. Chang et al. [16] explored tracking both pattern and path-based history in their indirect branch prediction scheme, the Target Cache.

Driesen & Hölzle proposed Cascaded Predictors [18, 20] that are a hybrid of two target predictors, where the first BTB-based predictor acts as a filter to catch easy-to-predict branches, while hard-to-predict branches are predicted by a second two-level adaptive predictor. The multi-stage predictor [20] generalizes the cascaded predictor.

Taking inspiration from work on conditional branch prediction [37], Kalamatianos & Kaelite use prediction by partial matching (PPM) for indirect branch prediction [19]. Seznec's TAGE, ITTAGE, and COTTAGE predictors [21] take inspiration from a PPM-like predictor [38]. The TAGE predictor predicts conditional branch directions while the ITTAGE predictor predicts indirect branch targets. The COTTAGE predictor incorporates both a TAGE and ITTAGE predictor in one to predict both branch directions and targets. The predictors use geometric history lengths [39] to index several partially-tagged predicting tables.

Kim et al.'s Virtual Program Counter (VPC) predictor uses a hardware-based devirtualization technique to predict indirect branch targets. VPC is based on the idea that a polymorphic branch instruction with T different known targets can be thought of as a series of T individual direct branch instructions. The predictor attempts to "devirtualize" indirect branch instructions in hardware to predict which of the T targets is the correct target output. VPC's main advantage is that, rather than use dedicated hardware, it reuses the existing conditional branch predictor and BTB. The conditional branch predictor is queried and the BTB accessed using a sequence of "virtual PCs" corresponding to at least the T targets of the indirect branch. The first virtual PC to output a *taken* prediction (if any) has its BTB target value output as the target prediction. Since the virtual PCs are are visited in series, the worst case scenario of no taken target may result in many wasted cycles. Sorting the targets by frequency allows the average latency to be low. A disadvantage of the VPC approach is that the conditional branch predictor and indirect branch predictor rely on one central prediction component that cannot be specialized to either task, but must be tuned to give good general overall performance.

TAP prediction [22], predicts the *address of a BTB entry* that contains the predicted target value. TAP predicts the BTB entry's address bit-by-bit using several small branch predictors (called sub-predictors) fashioned from the main conditional branch predictor. The idea is similar to BLBP, but TAP predicts the address of a BTB entry, not the target address itself. Note that TAP prediction modifies hardware to allow multiple different BTB entries for a single indirect branch address.

2.3 Perceptron-based Branch Prediction

Jiménez & Lin [24] first proposed the perceptron predictor, which bases its prediction scheme on the most basic neural-learning structure, the perceptron [23]. A single-layer perceptron is composed of a vector of integer weights $w_0 \dots w_n$. The output of a perceptron is computed as the dot product of the weighted vector with an input vector, $x_1 \dots x_n$, where w_0 behaves as the bias input. As such, the perceptron predictor maintains a table of weighted vectors rather than the more common saturating bits [14, 34, 40, 41].

To predict a branch outcome, a weighted vector is chosen by indexing the table using the branch's PC address. The dot product

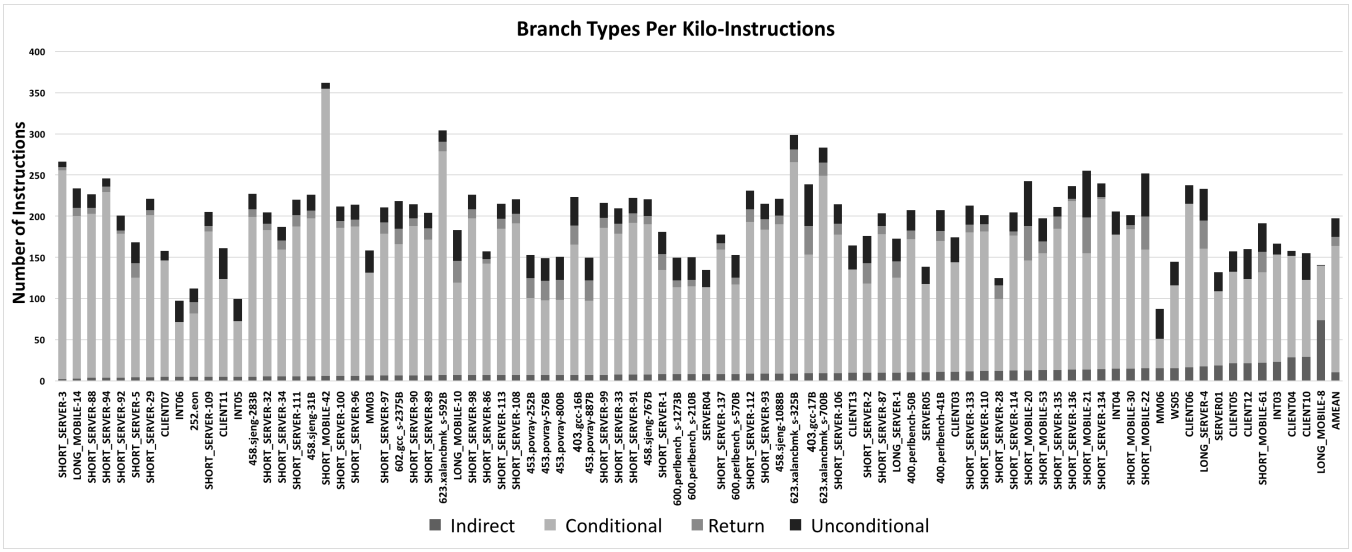

Figure 1: Breakdown of the prevalence of each branch type per kilo-instruction. Benchmarks are sorted by increasing prevalence of indirect branches.

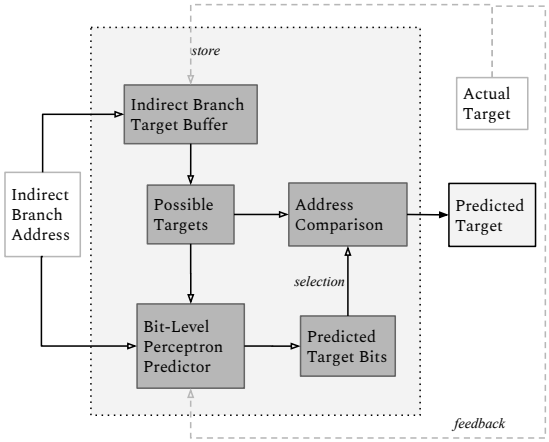

Figure 2: An overview of BLBP Target Address Prediction & Selection

is computed with the chosen weighted vector and the global history register (of binary branch outcome results) as input vector. If the dot product result is positive, a *taken* prediction is output; otherwise, a *not taken* output is given.

Due to the perceptron predictor's long prediction latency, Jiménez later proposed the path-based neural predictor [42]. The path-based neural predictor increases prediction accuracy while diminishing its prediction delay.

Piecewise linear branch predictor [43] improves on the original perceptron predictor by learning previously-unlearnable linearly inseparable branches.

Tarjan & Skadron's hashed perceptron predictor [44] improves upon the path-based predictor to further reduce prediction latency and increase prediction accuracy while maintaining less physical state.

3 BIT-LEVEL PERCEPTRON-BASED INDIRECT BRANCH PREDICTOR

BLBP is based on the Scaled Neural Indirect Prediction (SNIP) predictor [45], which originally presented at a branch prediction workshop without published proceedings. The SNIP predictor proposes a novel indirect branch predictor that predicts the individual lower-order bits of target addresses, then finding the closest matching address in an indirect branch target buffer.

Our extension to SNIP, BLBP, improves accuracy beyond the state-of-the-art and provides a path to a feasible implementation, greatly reducing the number of SRAM arrays that would be needed for a practical implementation from 44 to 8. The individual bits are predicted with neural predictors trained with perceptron learning.

3.1 Indirect Branch Target Buffer(IBTB)

BLBP uses a 64-way set-associative indirect branch target buffer (IBTB) indexed by indirect branch PC to store up to 64 indirect branch targets. Ideally there will be one set of targets per branch, but in practice many branches may index the same set. Thus each IBTB entry is tagged with 9 bits from the branch PC. The IBTB is managed with re-reference interval prediction [46] replacement. Figure 2 gives an overview of the process for predicting a target with BLBP.

3.2 Perceptrons For Each Bit of the Target

BLBP trains perceptrons the same way as a hashed perceptron [44], but rather than training a single weight for each target, a K-length vector of weights is trained, one for each bit of the target address that is being predicted.

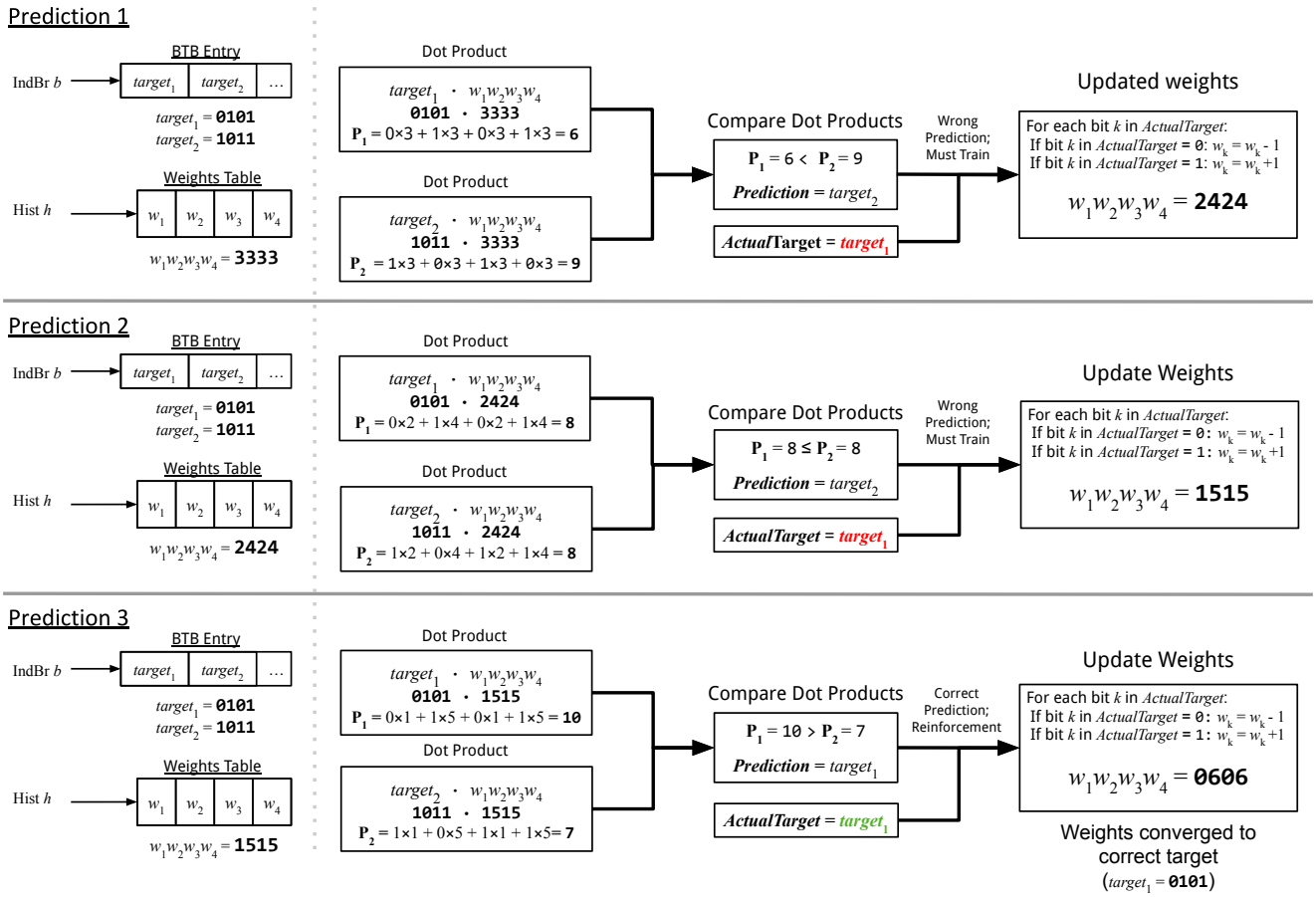

Figure 3: Through perceptron learning, BLBP trains weights to converge into bits of the correct target. Normalizing the trained weight vector $0, 6, 0, 6$ would equal $0, 1, 0, 1$, which is equal to the correct target bits in the above example.

Training by updating the K-length vector is shown in Algorithm 2 and described in detail with example in Section 3.5.

BLBP has eight history features indexing tables we call *sub-predictors*. Each sub-predictor predicts the values for each possible target that reflects the confidence of that sub-predictor that the corresponding bit in the target is 1. For each bit in each possible target, BLBP adds the output of each sub-predictor for that bit and build a vector (y_{out}). BLBP then computes the similarity of each possible target with the vector. The target with highest similarity will be selected as the prediction. Figure 4 shows an example for 4 bit target prediction. The first target has a higher similarity (Dot Product 51) with the trained perceptron than the second target(Dot Product 43).

3.3 Training For Multiple Histories

BLBP records the outcomes (*taken* or *not taken*) of recent conditional branches into 630 bit global history(*GHIST*). Similar to hashed perceptron learning [44], BLBP uses different lengths and sections of global history to train seven different sub-predictors. BLBP also records the outcomes(*taken* or *not taken*) of previous iterations of the last 256 branches individually into 10 bit local histories. BLBP indexes one sub-predictor with local history. More details on various histories used in BLBP is explained in Section 3.6

3.4 Predicting with Perceptrons

Figure 4 shows how BLBP makes predictions by aggregating the perceptrons from its eight sub-predictors and computing the similarity with each possible target. The first sub-predictor is trained with local history. The next seven sub-predictors are trained with different history lengths and history intervals explained in 3.3.

Figure 3 shows a simplified numeric example of the prediction and training algorithm with only one sub-predictor. For simplicity we show only two targets. We also assume only 4 bits of targets to be predicted.

First the predictor uses the history to index the weights table and extract 4 weights, one weight for each target bit. Weights are between -7 and 7. Here we assume each weight is equal to 3 initially. Then the predictor calculates the dot product of weights with the first targets bits. The same 4 weights will be used to compute the

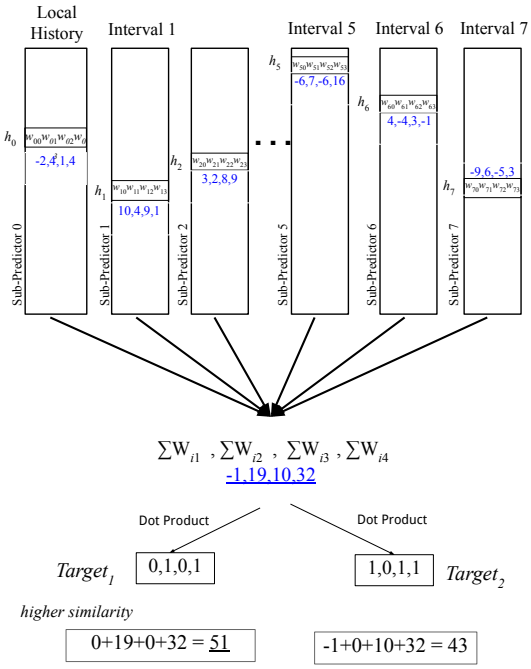

Figure 4: BLBP makes a prediction by aggregating the likelihood for two example targets from eight sub-predictors. Each sub-predictor is trained with different history lengths or local history.

Algorithm 1 Predict Algorithm

```
1:  function PREDICT(pc: address) : integer vector
2:      for i = 1...N in parallel do
3:          j = hashGHIST1...G_i mod M
4:          y_out 1...K ← 0
5:          for k = 1...K do
6:              y_out k ← y_out k + W i j k
7:      sum 1...T ← 0
8:      max ← 0
9:      maxTarget ← 0
10:     for t = 1...T in parallel do
11:         for k = 1...K do
12:             sumt ← sum[t] + y_out k × target[t][k]
13:         if sum[t]>max then
14:             max ← sum[t]
15:             maxTarget ← t
        return maxTarget
```

dot product for the second target. Next, the dot products will be compared. The target with the maximum dot product will be chosen as the predicted target. In this example in the first prediction the output of the second target's dot product had a higher value than the dot product for the first target. Thus, the second target is chosen as the predicted target.

3.5 Training Perceptrons

Algorithm 2 Update Algorithm

```
1:  function UPDATE(bits: address, target: address, suppress: address)
2:      for k = 1...K in parallel do
3:          if suppressk then
4:              continue
5:          a ← |bitsk|
6:          correct ← targetk = bitsk ≥ 0
7:          adaptive_training(correct, a, k)
8:          if (!correct or a < thetak then
9:              for i = 1...N do
10:                 j = hashGHIST1...G_i mod M
11:                 if targetk then
12:                     increment_unless_at_max(W[i][j][k])
13:                 else
14:                     decrement_unless_at_min(W[i][j][k])
```

To show how the predictor learns from a misprediction, we continue the same example from Figure 3. In the first prediction, we saw how the second target was chosen as the predicted target through the dot product and comparison. Now we assume the first target was the actual target. Then each of the 4 weights that were used to make this prediction is adjusted with following rule:

For each weight used to predict bit i of the actual target,

(1) if i is one then the weight is decremented,
(2) if i is zero then the weight is incremented.

Incrementing and decrementing weights will be saturated at the maximum or minimum values for the width of the weight. See Algorithm 2 for more details. In our example after the first prediction, the actual target's bits are (0,1,0,1) so the weights are update from (3,3,3,3) to (2,4,2,4). Figure 3 shows how after the second misprediction the weights are updated from (2,4,2,4) to (1,5,1,5), leading to a correct prediction. Assuming that the dot product for the actual target (here 10 for the first target) does not exceed the training threshold the weights will be updated after a correct prediction too. Figure 3 shows in our example the four weights after the correct prediction are update to (0,6,0,6). Normalizing the trained weight vector (0,6,0,6) would be (0,1,0,1) which is equal to the correct target bits. It is an example of how BLBP trains weights to converge to bits of correct targets through perceptron learning.

3.6 Optimization Techniques

We use several optimizations to improve the accuracy of the predictor:

Local History. The first weights vector predicting a given bit is indexed with a hash of *local history*. In conditional branch prediction, local history is a shift register with the outcome of previous iterations of the branch being predicted. For our predictor, we use shift registers with bit 3 of the target.

History Intervals. The remaining weights vectors are indexed using a hash of global history according to history intervals inspired by the interval capability in multiperspective perceptron prediction [47]

and strided sampling hashed perceptron prediction [48]. The intervals are tuned to achieve low MPKI. The following 7 global history intervals were used for the 7 non-local vectors: (0,13), (1,33), (23,49), (44,85), (77,149), (159,270), (252,630). For example, the second non-local vector was indexed by a hash of global history outcomes from position 1 in the global history through position 33. The tuned intervals were found by starting with geometric histories [39] and improving with hill-climbing, changing the start or end of an interval randomly and keeping the change if it improved MPKI.

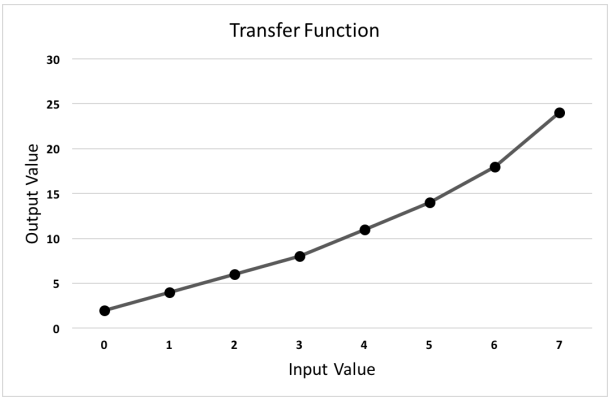

Figure 5: A transfer function amplifies the effect of higher weights and diminishes the effect of lower weights.

Transfer Functions. As with multiperspective perceptron prediction [47], we find that applying a non-linear transfer function to the weights before summation boosts accuracy. From previous work, we know that a convex function can amplify the effect of higher weights and diminishes the effect of lower weights, allowing the limited range of weights values to more precisely model the predicted bits. Figure 5 illustrates the transfer function we developed empirically.

Adaptive Threshold Training. As with O-GEHL and previous perceptron predictors, we use Seznec's adaptive threshold training algorithm to adjust the threshold for training such that the number of instances of training on correct predictions roughly equals the number of mispredictions [39].

Selective Bit Training. As described, the predictor predicts every lower-order bit in the target address. However, many branches have a small set of potential targets. The probability that a given bit in each target is the same is somewhat high for these small sets. Thus, the predictor only predicts and trains on a target bit if that bit differs in the set of potential targets. This selective bit training reduces the impact of destructive aliasing among weights that are used for more than one branch.

BTB Compression. Branch targets in the BTB are represented in a compressed format using the region-based organization proposed by Seznec for ITTAGE [25]. There are a number of regions of memory represented as higher-order target bits and stored in an array. Targets in the BTB are represented as one of these regions plus an offset. The number of regions is small and the number of bits in an offset is less than a full target, so we can represent a 64-bit target in about half as many bits. The region array is managed with least-recently-used (LRU) replacement. The BTB stores only partial tags for branch PCs, trading off the low probability of an accidental tag match with the storage benefit of partial tags.

3.7 Implementation

The prediction algorithm has several components that may seem challenging for implementation. Here we give some ideas for balancing complexity and timing.

Computing y_out. Perceptron predictors from the literature and in current processors compute a sum from a vector of weights read out from distinct tables. BLBP computes K such sums, one per bit of the target address being predicted. There are N tables whose entries are K-length vectors of weights. The summation operation is carried out in parallel on each of the K positions across the N tables. In our design, $N = 8$, fewer than the number of tables in recently proposed perceptron predictors in the literature or industry. Thus, the latency of the computation is at most the latency of a perceptron-based conditional branch predictor. The number of adders needed is $N \times K$. In our case, $K = 12$ so the predictor requires 96 small bit width adders for a parallel implementation. A mixed-signal implementation of perceptron-based branch predictors can be very quick and power-efficient even with a large number of inputs [49]. Weights are added as analog signals according to Kirchoff's law, and the transfer function can be implemented by changing the sizes of transistors in the digital to analog converters. For a more conventional digital implementation, the adders are organized as K adder trees with a depth no more than $\log_2 N$, or 3. The output of the final adders are 8 bits. Thus, the adders are small.

The parameter θ is tuned dynamically as described in Section 3.6. We find four bits per weight sufficient to maintain a good trade-off between accuracy and space-efficiency.

Computing the Cosine Similarity. Once the values in y_{out} have settled, the algorithm finds the non-normalized cosine similarity between y_{out} and the bit vectors representing the potential target addresses.

Algorithm 1 shows the code on how the BLBP prediction scheme computes a vector of K dot product values. Value K corresponds with the number of lower-order bits of the target address predicted by BLBP. N is the number of different histories. In effect, each of the N history lengths gets its own sub-predictor, M is the number of rows in the table of perceptron weights, *i.e.* the number of possible indices produced by the hash of global history.(Thus, indexing value j is bound by M.) G_i is the history length for the ith position in the perceptron predictor, determined as geometric history lengths [39]. WN, M, K is a three-dimensional array of 4-bit sign/magnitude integer weights. The W array is realized as N SRAM arrays each indexed by a different hash of global history to yield a vector of perceptron weights, one weight per address bit predicted.

This dot product computation is equivalent to measuring the cosine similarity between two vectors, *i.e.* the cosine of the angle between two vectors. A higher cosine similarity implies a closer match between the two vectors. The cosine similarity is usually normalized by the product of the magnitude of the two vectors, but in our case this is unnecessary because each bit vector is multiplied

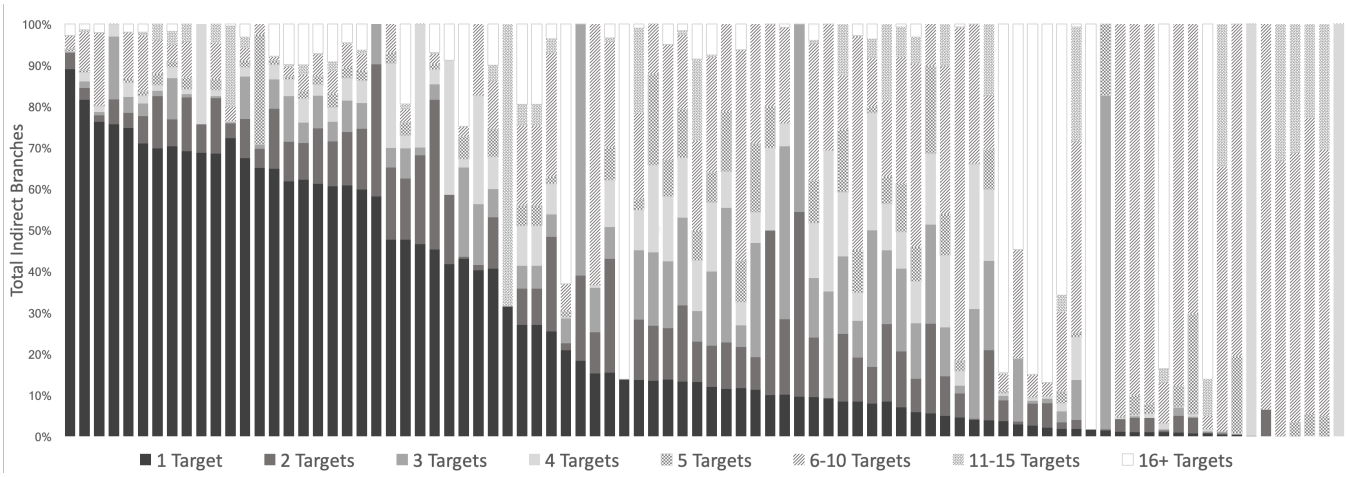

Figure 6: Polymorphism in workloads. Traces are ordered from fewest to most targets, relative to the percentage of instructions with more than one target address.

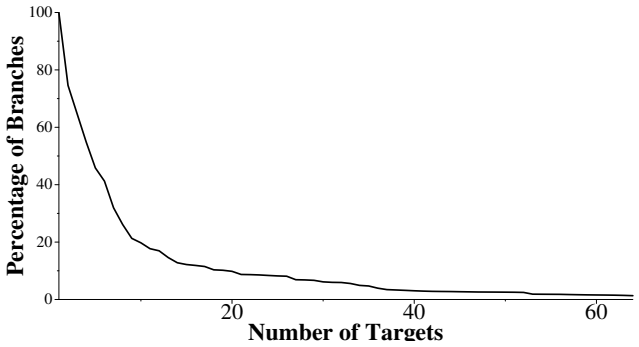

Figure 7: Distribution of Number of Potential Targets

by the same value of y_{out}, thus the dot product values are directly comparable.

Computing the non-normalized cosine similarity is simply taking the dot product of y_{out} and a bit vector. This dot product computation can be carried out very efficiently by taking the sum of the bitwise AND of each element of y_{out} and the corresponding sign-extended bit in the vector. This can be accomplished with an adder tree similar to the computation of y_{out} itself.

Computing the Cosine Similarity Repeatedly. There are up to 64 potential target addresses, thus up to 64 possible cosine similarities to compute. An associative lookup and parallel cosine similarity computation would be fast but somewhat expensive in terms of power and complexity. A sequential search would be more efficient but have higher latency in cases where there are many targets matching the current branch. Fortunately, in practice, the vast majority of indirect branches have only a few observed targets. Figure 7 shows the distribution of numbers of targets. For each value from 1 to 64 on the *x*-axis, the height of the curve gives the percentage of all indirect branches having at least that many distinct targets. We can see from the graph that the majority of indirect branches have no more than 5 potential targets. Only 10% of indirect branches have more than 20 potential targets. Thus, a feasible implementation could compute 5 cosine similarities per cycle in parallel at a modest cost, taking only one cycle for over half of all predictions and no more than 4 cycles for 90% of the predictions. With current decoupled fetch engines, this modest latency would be hidden for most branches and only exposed in those few cases with many targets where getting the right prediction matters the most. Note that VPC may also incur significant latency in extreme cases with many potential targets.

3.8 Searching the BTB

The algorithm requires collecting all branch targets that match a given branch PC from the BTB. This can be accomplished with a content addressable memory (CAM) or by a sequential lookup similar to the strategy taken by the VPC algorithm [4]. Although there are up to 64 potential targets for a given branch, in practice there are very few targets for each branch and the sequential strategy can be quick.

4 TESTING METHODOLOGY

4.1 Benchmarks

For evaluation, we use a suite of 88 workloads gathered from multiple benchmark suite sources. Represented are SPEC CPU2000 [50], SPEC CPU2006 [51] and SPEC CPU2017 [52] benchmark suites and traces from most recent JILP Branch Prediction Competition [53] held at ISCA 2016.

Traces were selected if they averaged more than one misprediction per 1000 instructions for an initial simulation using an infinite-sized branch target buffer. Table 1 summarizes the sources and qualities of our benchmarks.

The Samsung-sourced benchmark `LONG-MOBILE-8` contains *more* indirect branches than conditional. In general, the Samsung workloads tend to have more indirect branches compared to the other workload sources. Although Samsung did not provide much information about the benchmarks used, it is reasonable to expect that many of the workloads come from Android workloads in the Java

Benchmark Source	Number of Benchmarks	Details of Workloads
SPEC CPU2000	1	`252.eon`
SPEC CPU2006	12	`400.perlbench`, `403.gcc`, `453.povray`, `458.sjeng`
SPEC CPU2017	7	`600.perlbench`, `602.gcc`, `623.xalancbmk`
CBP-5 Competition	68	Industry-sourced workloads offered for CBP-5 by host Samsung. Benchmarks are divided into two categories: `MOBILE` and `SERVER`. Trace lengths also fall in two categories: `SHORT` and `LONG`.

Table 1: Description of the 88 workloads used for testing and evaluation. The benchmarks come from four sources, including a Championship Branch Prediction competition.

Indirect Branch Predictor	Implementation Configuration	Total Hardware Budget
BTB	32K-entry, partially-tagged, direct-mapped branch target buffer	64 KB
VPC	32K-entry, partially-tagged, direct-mapped BTB with Multi-perspective Perceptron Predictor for conditional branch prediction	128 KB
ITTAGE	as described in the original paper [25]	64 KB
BLBP	64-entry, 64-way set-associative, partially-tagged IBTB, 256 10-bit local histories, 630-bit global history, 8 correlating-weights tables, and 128-entry region array	64.08 KB

Table 2: The implementation setups for the indirect branch predictors evaluated.

language. Since Java relies on virtual method dispatch by default, it is not surprising to find many indirect branches in these traces.

Figure 6 shows the degree of polymorphism present in the traces. Many benchmarks are dominated by monomorphic branches, but many have a great number of indirect branches with multiple targets.

4.2 Simulation Setup

We use the branch prediction simulation infrastructure released for the Championship Branch Prediction competition [53]. The simulation infrastructure has been augmented with additional code for a BTB and an assortment of indirect branch predictors including BLBP. For conditional branch prediction, we use a hashed perceptron predictor [44].

The appropriate metric for branch prediction studies is mispredictions per kilo-instruction (MPKI). Unlike simple misprediction rate, MPKI takes into account the relative frequency of branches compared to other instructions. Previous work has demonstrated a linear relationship between MPKI and performance [54]. Thus, it is sufficient to measure MPKI to infer an impact on performance.

Table 2 gives detailed information on the indirect branch predictors implemented for this study. We use a BTB as our indirect branch prediction baseline. The baseline BTB is a 32K-entry cache indexed by branch address and filled with the most recently observed branch target for that branch address. This BTB is large compared with recent examples from industry. For example, the BTB in Samsung's recent Mongoose processor has 4,096 entries [55]. Thus, this BTB provides a practical upper limit on the accuracy of modern BTBs.

We implement the Virtual Program Counter (VPC) predictor from Kim *et al.* [4] using a 64KB Multi-perspective Perceptron Predictor (MPP) [47] as its underlying conditional branch predictor. The original VPC implementation relies on global branch history to make predictions for indirect branches. However, our implementation is a hashed perceptron predictor that uses a set of 37 features to provide alternate perspectives on branch history. Our implementation of VPC has a 32K-entry BTB for target storage. This modified VPC gave an MPKI of 0.29 for indirect branches while maintaining a low degradation of 2.05% in the prediction accuracy of conditional branches.

Finally, we set up BLBP as described in Section 3. There are 8 independently accessed SRAM arrays for 8 different history intervals. Each SRAM array has rows of 12 4-bit vectors implementing the perceptron weights. There is a 630-bit global history and a table of 256 10-bit local histories. There is an indirect branch target buffer (IBTB) consisting of 64 sets of 64 entries, each of which contains an 8-bit partial tag, a 7-bit region number, a 20-bit region offset, and a 2-bit re-reference interval prediction [56] for replacement. The region number and descriptor allow a compressed representation of targets in the same manner as the ITTAGE predictor [25]. The total state for the prediction tables, histories, IBTB, and region table is approximately 64KB, allowing an iso-area comparison with the ITTAGE implementation from the second branch prediction competition.

Each of the SPEC traces is a simpoint [57] of one billion instructions. The Samsung-sourced traces are of variable length averaging in the 100s of millions of instructions. For our experiments we measure branch target MPKI over each entire trace.

5 RESULTS

5.1 Overall Performance

Among the four predictor implementations, the baseline BTB predictor performed the worst, with an arithmetic mean of 3.40 MPKI across the benchmarks. Next was VPC with a mean of 0.29 MPKI. ITTAGE and BLBP rounded out our results with 0.193 and 0.183 MPKI, respectively. Thus, for the benchmark suite tested, BLBP led to a 5% improvement in MPKI over the state-of-the-art.

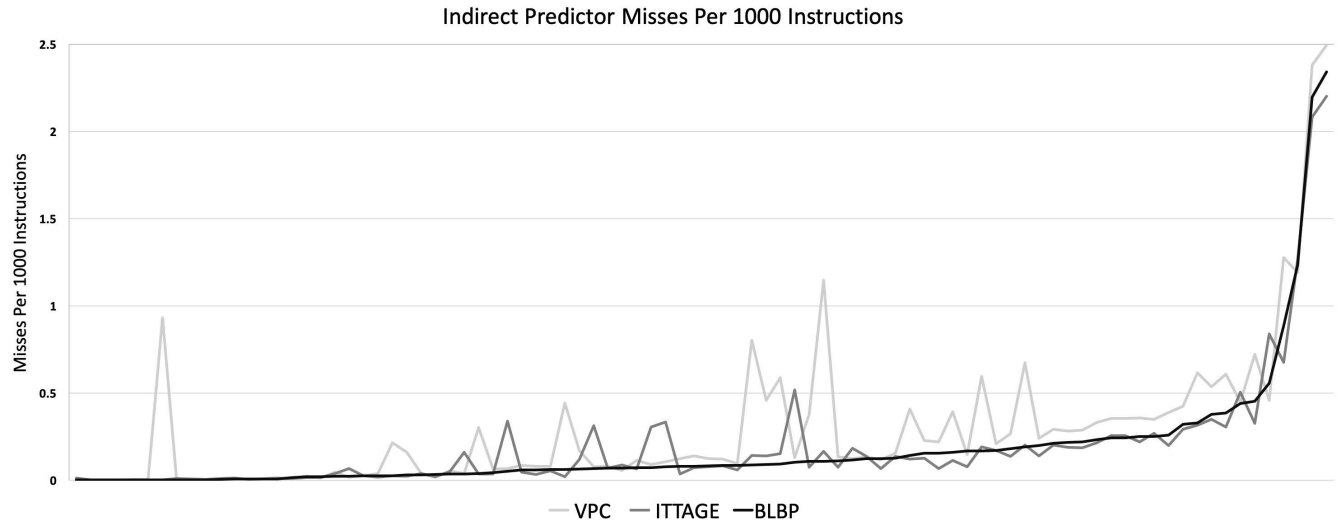

Figure 8: MPKI for 3 of 4 indirect branch predictors on a suite of 88 higher-MPKI benchmarks. BTB MPKI has been omitted due to much-higher MPKI relative to the other predictors. Benchmarks are sorted by BLBP MPKI.

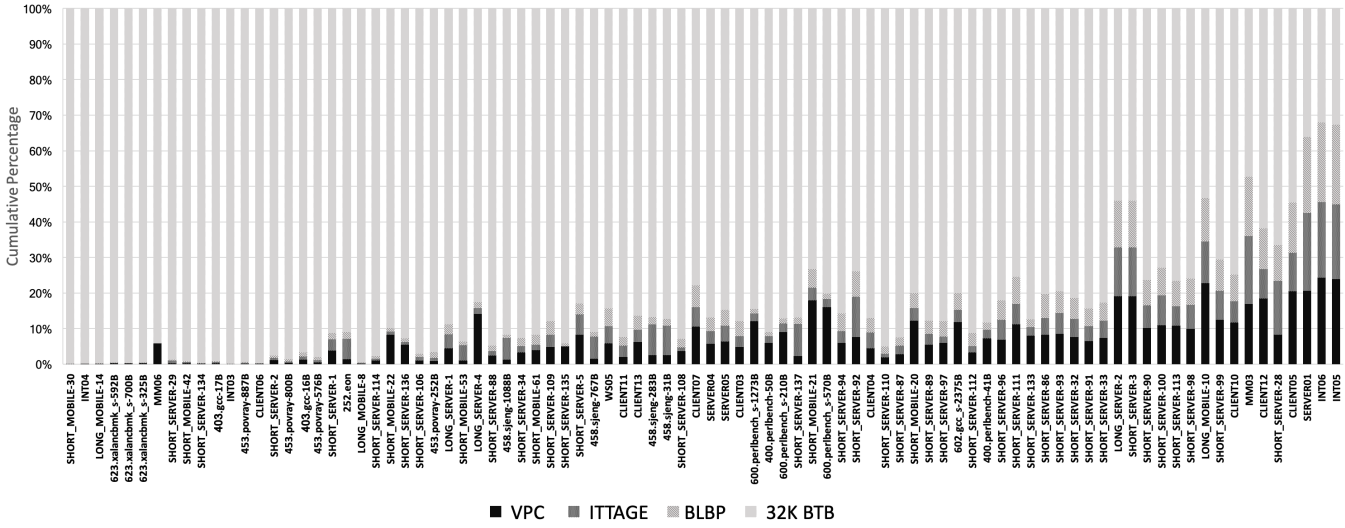

Figure 9: Percentage breakdown of predictor MPKI performance relative to each other. Benchmarks are sorted by BLBP MPKI.

In response to reviwer concerns, we also tested the predictors with the CBP4 [58] workloads, which are different from the CBP5 traces used to develop BLBP. Without tuning either predictor, ITTAGE yields 0.028 MPKI while BLBP achieves 0.027 MPKI, outperforming ITTAGE by 3.5%.

Figure 8 presents the MPKI results for three of the four predictors, with the BTB baseline being omitted due to its very high MPKI relative to the other predictors. The benchmarks (unlabeled) are sorted by monotonically increasing BLBP MPKI performance.

Figure 9 shows similar data, but reveals MPKIs of the four predictors compared to each other.

5.2 Effect of Optimizations

Figure 10 illustrates the effect of the optimizations applied to BLBP. The figure shows the percent improvement in average MPKI over ITTAGE for versions of the predictor for configurations where only one optimization is enabled, one of the optimizations is disabled, and all or none of the optimizations are enabled. With all optimizations turned on, BLBP achieves 5.3% improvement over ITTAGE. With no optimizations, BLBP achieves an average 8.8% higher MPKI than ITTAGE. Thus, doing at least some optimizations is essential to BLBP.

Let us explore what happens when turning on only one of the optimizations at a time. Turning on the local history improves the

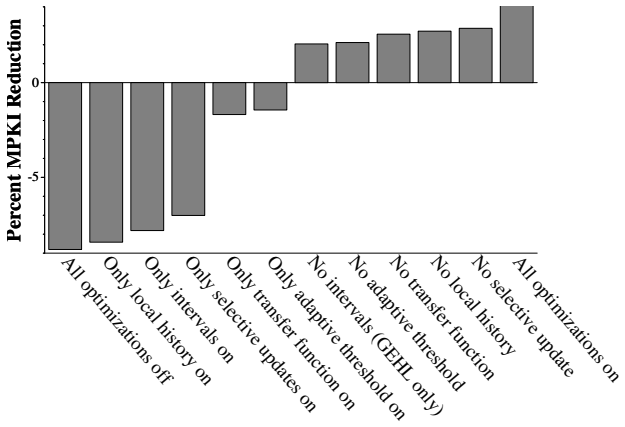

Figure 10: Effect of Optimizations

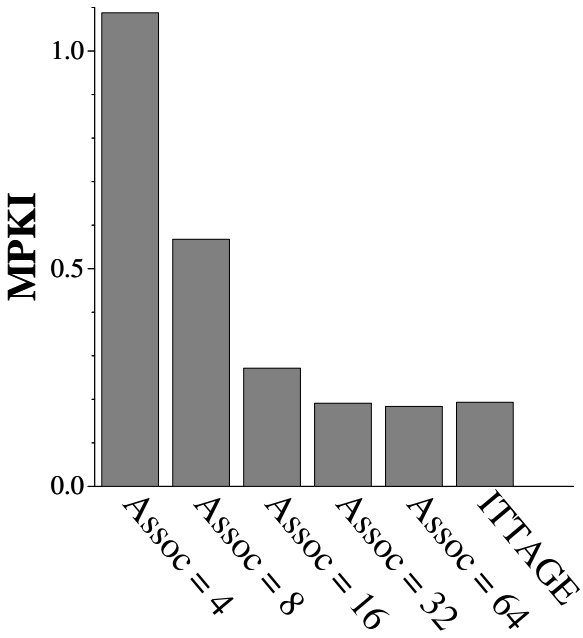

Figure 11: Effect of Associativity

predictor the least, followed by using intervals instead of geometric history lengths (GEHLs), and selectively updating only bits that differ in targets. Using a non-linear transfer function provides a large boost to the otherwise unoptimized predictor, bringing it within 1.7% of ITTAGE. Adaptive training provides the largest boost, only 1.4% higher MPKI than ITTAGE.

Now let us see what happens when turning off only one optimization, keeping the rest. The biggest change occurs when using GEHL histories instead of intervals; the improvement over ITTAGE drops from 5.3% to 2.04%. Thus, in concert with the other features, intervals seem to provide an important boost. Omitting selective bit updates seems to hurt the least, lowering MPKI improvement to 2.86%. However, omitting any of the optimizations produces a significant reduction in MPKI improvement. Thus, the optimizations seem to work synergistically (if not additively) to improve accuracy.

5.3 Effect of Associativity

The IBTB is a set-associative structure. Each set stores 64 targets observed from indirect branches that hash to that set. Although, as we have seen in Section 3.7, most branches have at most 5 targets, many branches may hash to the same IBTB set causing collisions, and some branches actually require many targets. Thus, the associativity of the structure must be large enough to combat conflict misses and to accommodate branches with many targets. Figure 11 shows the effect on accuracy of varying the associativity of the IBTB, keeping the number of IBTB entries the same at 4,096. A 4-way IBTB gives an unacceptably high 1.09 MPKI. An 8-way IBTB cuts the MPKI almost in half to 0.57. A 16-way BTB gives another large improvement, down to 0.27 MPKI. With 32-way set associativity, the IBTB yields an MPKI of 0.19, equivalent to ITTAGE. The 64-way IBTB gives 0.183 MPKI, a 5% improvement over ITTAGE.

6 CONCLUSIONS & FUTURE WORK

In this paper, we introduced Bit-Level Perceptron-Based Indirect Branch Predictor, or BLBP. BLBP predicts indirect branch targets' select lower-order bits using perceptron-based learning. A selection of known target addresses for the branch are gathered from the IBTB for comparison; the target address that matches closest with the predicted bits is output as BLBP's prediction.

We have shown BLBP to outperform the state-of-the-art. Using a suite of 88 benchmarks with significant indirect branches, we show BLBP improves upon ITTAGE's prediction performance by 5%, reducing MPKI from 0.193 to 0.183.

In future work we plan to improve accuracy by exploring features beyond global and per-branch history. For example, the recently proposed multiperspective perceptron predictor uses a variety of control-flow features to improve accuracy [47]. We plan to reduce the complexity of the computations involved. The original perceptron predictor was quite complex with a high latency, but follow-up work reduced the latency through ahead-pipelining and other techniques [42, 43, 59]. We are confident that the same sorts of techniques can be applied to BLBP to achieve lower latency and power while maintaining high accuracy. We plan to explore ways of avoiding the high-associativity of the IBTB, perhaps using a hierarchy of structures [18, 60]. We also plan to explore how BLBP might be used to predict conditional branches as well as indirect branches as VPC does, allowing consolidation of the two structures.

7 ACKNOWLEDGMENTS

We thank the anonymous reviewers for their valuable comments and feedback to improve the content and quality of this paper. We thank Samsung for supporting this work through their Global Outreach Program. We also thank the National Science Foundation for supporting this work through grant CCF-1649242. Portions of this research were conducted with the advanced computing resources provided by Texas A&M High Performance Research Computing.

REFERENCES

[1] B. Calder, D. Grunwald, and B. Zorn, "Quantifying behavioral differences between c and c++ programs," *Journal of Programming languages*, vol. 2, no. 4, pp. 313–351, 1994.

[2] L. Cardelli and P. Wegner, "On understanding types, data abstraction, and polymorphism," *ACM Comput. Surv.*, vol. 17, pp. 471–523, December 1985.

[3] H. Srinivasan and P. F. Sweeney, "Evaluating virtual dispatch mechanisms for c++," Tech. Rep. Technical Report RC 20330, IBM TJ Watson Center, January 1996.

[4] H. Kim, J. A. Joao, O. Mutlu, C. J. Lee, Y. N. Patt, and R. Cohn, "Vpc prediction: Reducing the cost of indirect branches via hardware-based dynamic devirtualization," in *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA '07, (New York, NY, USA), pp. 424–435, ACM, 2007.

[5] D. R. Kaeli and P. G. Emma, "Branch history table prediction of moving target branches due to subroutine returns," in *[1991] Proceedings. The 18th Annual International Symposium on Computer Architecture*, pp. 34–42, May 1991.

[6] B. Calder and D. Grunwald, "Reducing indirect function call overhead in c++ programs," in *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '94, (New York, NY, USA), pp. 397–408, ACM, 1994.

[7] U. Hölzle and D. Ungar, "Optimizing dynamically-dispatched calls with run-time type feedback," in *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, PLDI '94, (New York, NY, USA), pp. 326–336, ACM, 1994.

[8] D. Grove, J. Dean, C. Garrett, and C. Chambers, "Profile-guided receiver class prediction," in *Proceedings of the Tenth Annual Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '95, (New York, NY, USA), pp. 108–123, ACM, 1995.

[9] G. Aigner and U. Hölzle, "Eliminating virtual function calls in c++ programs," tech. rep., Santa Barbara, CA, USA, 1996.

[10] S. Porat, D. Bernstein, Y. Fedorov, J. Rodrigue, and E. Yahav, "Compiler optimization of c++ virtual function calls," in *Proceedings of the 2Nd Conference on USENIX Conference on Object-Oriented Technologies (COOTS) - Volume 2*, COOTS'96, (Berkeley, CA, USA), pp. 1–1, USENIX Association, 1996.

[11] K. Ishizaki, M. Kawahito, T. Yasue, H. Komatsu, and T. Nakatani, "A study of devirtualization techniques for a java just-in-time compiler," in *Proceedings of the 15th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '00, (New York, NY, USA), pp. 294–310, ACM, 2000.

[12] J. Dean, D. Grove, and C. Chambers, "Optimization of object-oriented programs using static class hierarchy analysis," in *Proceedings of the 9th European Conference on Object-Oriented Programming*, ECOOP '95, (London, UK, UK), pp. 77–101, Springer-Verlag, 1995.

[13] H. D. Pande and B. G. Ryder, "Static type determination for c++," in *Proceedings of the 6th Conference on USENIX Sixth C++ Technical Conference - Volume 6*, CTEC'94, (Berkeley, CA, USA), pp. 5–5, USENIX Association, 1994.

[14] J. K. F. Lee and A. J. Smith, "Branch prediction strategies and branch target buffer design," *Computer*, vol. 17, pp. 6–22, Jan 1984.

[15] R. Nair, "Dynamic path-based branch correlation," in *Proceedings of the 28th Annual International Symposium on Microarchitecture*, pp. 15–23, December 1995.

[16] P.-Y. Chang, E. Hao, and Y. N. Patt, "Target prediction for indirect jumps," in *Conference Proceedings. The 24th Annual International Symposium on Computer Architecture*, pp. 274–283, June 1997.

[17] K. Driesen and U. Hölzle, "Accurate indirect branch prediction," in *Proceedings. 25th Annual International Symposium on Computer Architecture (Cat. No.98CB36235)*, pp. 167–178, Jun 1998.

[18] K. Driesen and U. Hölzle, "The cascaded predictor: Economical and adaptive branch target prediction," in *Proceedings of the 31th International Symposium on Microarchitecture*, December 1998.

[19] J. Kalamatianos and D. R. Kaeli, "Predicting indirect branches via data compression," in *Proceedings. 31st Annual ACM/IEEE International Symposium on Microarchitecture*, pp. 272–281, Nov 1998.

[20] K. Driesen and U. Hölzle, "Multi-stage cascaded prediction," in *Euro-Par'99 Parallel Processing* (P. Amestoy, P. Berger, M. Daydé, D. Ruiz, I. Duff, V. Frayssé, and L. Giraud, eds.), (Berlin, Heidelberg), pp. 1312–1321, Springer Berlin Heidelberg, 1999.

[21] A. Seznec, "A case for (partially) tagged geometric history length branch prediction," *Journal of Instruction-Level Parallelism (JILP) Special Issue: The Third Championship Branch Prediction Competition (CBP-3)*, vol. 9, February 2006.

[22] Z. Xie, D. Tong, M. Huang, X. Wang, Q. Shi, and X. Cheng, "Tap prediction: Reusing conditional branch predictor for indirect branches with target address pointers," in *2011 IEEE 29th International Conference on Computer Design (ICCD)*, pp. 119–126, Oct 2011.

[23] H. D. Block, "The perceptron: A model for brain functioning," *Reviews of Modern Physics*, vol. 34, pp. 123–135, 1962.

[24] D. A. Jiménez and C. Lin, "Dynamic branch prediction with perceptrons," in *Proceedings of the 7th International Symposium on High Performance Computer Architecture (HPCA-7)*, pp. 197–206, January 2001.

[25] A. Seznec, "A 64-kbytes ittage indirect branch predictor," in *Proceedings of the JWAC-2: Championship Branch Prediction*, June 2011.

[26] D. W. Wall, "Limits of instruction-level parallelism," in *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS IV, (New York, NY, USA), pp. 176–188, ACM, 1991.

[27] A. Roth, A. Moshovos, and G. S. Sohi, "Improving virtual function call target prediction via dependence-based pre-computation," in *Proceedings of the 13th International Conference on Supercomputing*, ICS '99, (New York, NY, USA), pp. 356–364, ACM, 1999.

[28] J. A. Joao, O. Mutlu, H. Kim, R. Agarwal, and Y. N. Patt, "Improving the performance of object-oriented languages with dynamic predication of indirect jumps," *SIGARCH Comput. Archit. News*, vol. 36, pp. 80–90, March 2008.

[29] M. U. Farooq, L. Chen, and L. Kurian, "Value based btb indexing for indirect jump prediction," in *HPCA - 16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*, pp. 1–11, Jan 2010.

[30] M. Tan, X. Liu, T. Tong, and X. Cheng, "Cvp: An energy-efficient indirect branch prediction with compiler-guided value pattern," in *Proceedings of the 26th ACM International Conference on Supercomputing*, ICS '12, (New York, NY, USA), pp. 111–120, ACM, 2012.

[31] W. J. Ghandour and N. J. Ghandour, "Leveraging dynamic slicing to enhance indirect branch prediction," in *2014 IEEE 32nd International Conference on Computer Design (ICCD)*, pp. 292–299, Oct 2014.

[32] C. Young and M. D. Smith, "Improving the accuracy of static branch prediction using branch correlation," in *Proceedings of ASPLOS VI*, pp. 232–241, 1994.

[33] C. Young, N. Gloy, and M. D. Smith, "A comparative analysis of schemes for correlated branch prediction," in *Proceedings 22nd Annual International Symposium on Computer Architecture*, pp. 276–286, June 1995.

[34] T.-Y. Yeh and Y. N. Patt, "Two-level adaptive training branch prediction," in *Proceedings of the 24th ACM/IEEE International Symposium on Microarchitecture*, pp. 51–61, November 1991.

[35] S.-T. Pan, K. So, and J. T. Rahmeh, "Improving the accuracy of dynamic branch prediction using branch correlation," in *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS V, (New York, NY, USA), pp. 76–84, ACM, 1992.

[36] T.-Y. Yeh and Y. N. Patt, "A comparison of dynamic branch predictors that use two levels of branch history," in *Proceedings of the 20th Annual International Symposium on Computer Architecture*, May 1993.

[37] I.-C. K. Chen, J. T. Coffey, and T. N. Mudge, "Analysis of branch prediction via data compression," in *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VII, (New York, NY, USA), pp. 128–137, ACM, 1996.

[38] P. Michaud, "A ppm-like, tag-based branch predictor," *Journal of Instruction Level Parallelism*, vol. 7, no. 1, pp. 1–10, 2005.

[39] A. Seznec, "Genesis of the o-gehl branch predictor," *Journal of Instruction-Level Parallelism (JILP)*, vol. 7, April 2005.

[40] J. E. Smith, "A study of branch prediction strategies," in *Proceedings of the 8th Annual International Symposium on Computer Architecture*, pp. 135–148, May 1981.

[41] S. McFarling, "Combining branch predictors," tech. rep., Technical Report TN-36, Digital Western Research Laboratory, 1993.

[42] D. A. Jiménez, "Fast path-based neural branch prediction," in *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-36)*, pp. 243–252, IEEE Computer Society, December 2003.

[43] D. A. Jiménez, "Piecewise linear branch prediction," in *Proceedings of the 32nd Annual International Symposium on Computer Architecture (ISCA-32)*, June 2005.

[44] D. Tarjan and K. Skadron, "Merging path and gshare indexing in perceptron branch prediction," *ACM Trans. Archit. Code Optim.*, vol. 2, pp. 280–300, September 2005.

[45] D. A. Jiménez, "Snip: Scaled neural indirect predictor," in *Proceedings of the JWAC-2: Championship Branch Prediction*, June 2011.

[46] A. Jaleel, K. B. Theobald, S. C. Steely Jr, and J. Emer, "High performance cache replacement using re-reference interval prediction (rrip)," in *ACM SIGARCH Computer Architecture News*, vol. 38, pp. 60–71, ACM, 2010.

[47] D. A. Jiménez, "Multiperspective perceptron predictor," *The Fifth Championship Branch Prediction Competition (CBP-5)*, June 2016.

[48] D. A. Jiménez, "Strided sampling hashed perceptron predictor," in *Proceedings of JWAC-4: Championship Branch Prediction*, June 2014.

[49] R. S. Amant, D. A. Jiménez, and D. Burger, "Low-power, high-performance analog neural branch prediction," in *Proceedings of the 41th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-41)*, IEEE Computer Society, November 2008.

[50] Standard Performance Evaluation Corporation, *SPEC CPU 2000, http://www.spec.org/osg/cpu2000*, April 2000.

[51] J. L. Henning, "SPEC CPU2006 benchmark descriptions," *SIGARCH Comput. Archit. News*, vol. 34, pp. 1–17, September 2006.
[52] J. L. Henning, "The SPEC CPU2017 benchmark package." https://www.spec.org/cpu2017/Docs/overview.html#benchmarks.
[53] The Journal of Instruction-Level Parallelism, *The 5th JILP Championship Branch Prediction Competition (CBP-5), https://www.jilp.org/cbp2016*, June 2016.
[54] Z. Wang and D. A. Jiménez, "Program inteferometry," in *Proceedings of the 2011 IEEE International Symposium on Workload Characerization*, November 2011.
[55] B. Burgess, "Samsung's exynos-m1 cpu," in *Hot Chips: A Symposium on High Performance Chips*, August 2016.
[56] A. Jaleel, K. Theobald, S. S. Jr., and J. Emer, "High performance cache replacement using re-reference interval prediction (rrip)," in *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA-37)*, June 2010.
[57] 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*, October 2002.
[58] The Journal of Instruction-Level Parallelism, *The 4th JILP Championship Branch Prediction Competition (CBP-4), https://www.jilp.org/cbp2014*, June 2014.
[59] G. Loh and D. A. Jiménez, "Modulo path history for the reduction of pipeline overheads in path-based neural branch predictors," *International Journal of Parallel Programming (IJPP)*, vol. 36, no. 2, pp. 267–286, 2008.
[60] D. A. Jiménez, S. W. Keckler, and C. Lin, "The impact of delay on the design of branch predictors," in *Proceedings of the 33rd Annual International Symposium on Microarchitecture (MICRO-33)*, pp. 67–76, December 2000.