Whisper: Profile-Guided Branch Misprediction Elimination for Data Center Applications

Tanvir Ahmed Khan∗  Muhammed Ugur†  Krishnendra Nathella‡  Dam Sunwoo§  Heiner Litz‡
Daniel A. Jiménez‡  Baris Kasikci∗

∗University of Michigan  †ARM  ‡University of California, Santa Cruz  §Texas A&M University
∗{takh, meugur, barisk}@umich.edu  †{Krishnendra.Nathella, Dam.Sunwoo}@arm.com  ‡hlitz@ucsc.edu
§djjimenez@acm.org

Abstract—Modern data center applications experience frequent branch mispredictions – degrading performance, increasing cost, and reducing energy efficiency in data centers. Even the state-of-the-art branch predictor, TAGE-SC-L, suffers from an average branch Mispredictions Per Kilo Instructions (branch-MPKI) of 3.0 (0.5-7.2) for these applications since their large code footprints exhaust TAGE-SC-L’s intended capacity.

In this work, we propose Whisper, a novel profile-guided mechanism to avoid branch mispredictions. Whisper investigates the in-production profile of data center applications to identify precise program contexts that lead to branch mispredictions. Corresponding prediction hints are then inserted into code to strategically avoid those mispredictions during program execution. Whisper presents three novel profile-guided techniques: (1) hashed history correlation which efficiently encodes hard-to-predict correlations in branch history using lightweight Boolean formulas, (2) randomized formula testing which selects a locally-optimal Boolean formula from a randomly selected subset of possible formulas to predict a branch, and (3) the extension of Read-Once Monotone Boolean Formulas with Implication and Converse Non-Implication to improve the branch history coverage of these formulas with minimal overhead.

We evaluate Whisper on 12 widely-used data center applications and demonstrate that Whisper enables traditional branch predictors to achieve a speedup close to that of an ideal branch predictor. Specifically, Whisper achieves an average speedup of 2.8% (0.4%-4.6%) by reducing 16.8% (1.7%-32.4%) of branch mispredictions over TAGE-SC-L, and outperforms the state-of-the-art profile-guided branch prediction mechanisms by 7.9% on average.

I. INTRODUCTION

Modern data center applications exhibit large instruction footprints and suffer from frequent frontend and misprediction stalls, incurring performance losses worth millions of dollars [1, 2, 3, 4, 5, 6, 7]. These applications contain complex application logic [1, 2, 3] and frequently use different libraries [6], language runtimes [8, 9], and kernel modules [3, 10]. As a result, these applications’ hot code footprints range from tens to hundreds of megabytes [1, 3, 6, 11] which overwhelm on-chip cache structures like the Instruction cache (I-cache), Branch Target Buffer (BTB), and the branch predictor, whose sizes are only hundreds of kilobytes [5]. Consequently, processors are unable to sufficiently fetch useful instructions [3] when executing modern data center applications – leading to frequent frontend and misprediction stalls [5]. These stalls notably increase the Total Cost of Ownership (TCO) of a data center [3, 6], and even a single-digit reduction of these stalls can save millions of dollars in management and energy costs while significantly reducing the global carbon footprint [4].

Several techniques have been proposed to address these challenges including decoupled frontends [12] leveraging Fetch Directed Instruction Prefetching (FDIP) [13, 14, 15] and Profile-Guided Optimizations (PGO) [11, 16, 17, 18, 19, 20, 21] that are efficiently supported by today’s hardware [22, 23, 24, 25, 26] and software [1, 3, 17, 27, 28] systems.

On the hardware side, FDIP avoids the tight coupling between branch prediction and instruction fetch, enabling branch predictor-guided instruction prefetching to avoid frontend stalls. As long as FDIP can run sufficiently ahead, it can eliminate frontend stalls effectively. Thereby, FDIP’s performance depends on the accuracy of the branch predictor, as frequent mispredictions limit FDIP’s effectiveness in mitigating frontend stalls [29, 30, 31, 32].

Profile-guided code layout optimizations address the large instruction footprint problem by placing frequently executed I-cache lines together, thereby improving instruction locality. These techniques do not require any hardware modifications, and although these techniques are sensitive to profile quality [33], they work well in practice. Profiles for data center applications change slowly over several weeks [17] while companies like Google and Facebook deploy new binaries every few days – giving PGO techniques ample opportunity to adapt to changing profiles [1, 11, 17]. As a result, these techniques are widely-used in today’s data centers [1, 3, 11, 17, 27]. For example, half of all CPU cycles in Google data centers execute instructions from PGO-optimized applications [17]. Unfortunately, existing PGO techniques primarily reduce frontend stalls and eliminate less than 10% of all branch mispredictions [11].

To quantify the performance implications of branch mispredictions, we extensively investigate the behavior of 12 modern data center applications to show that their large code footprints trigger frequent branch mispredictions, significantly impeding the efficacy of state-of-the-art techniques. In particular, we find that even a 64KB TAGE-SC-L [34] predictor experiences an average branch-MPKI of 3.0 (0.5-7.2) for these applications primarily due to capacity reasons. Furthermore, our investigation reveals that state-of-the-art profile-guided branch prediction mechanisms, BranchNet [35] and Read-Once
Monotone Boolean Formulas (ROMBF) [36] reduce only 8.9% of all branch mispredictions that TAGE-SC-L incurs as they also fail to scale for large code footprints.

In this work, we focus on eliminating branch mispredictions with Whisper—a profile-guided technique that identifies branch instructions causing frequent mispredictions, correlates their direction with many prior branch directions (i.e., history), and efficiently encodes this correlation using Boolean formulas. In particular, Whisper introduces three novel techniques to improve profile-guided branch prediction and reduces 16.8% of all mispredictions by leveraging (1) hashed history correlation, (2) randomized Boolean formula testing, and (3) an extension of ROMBF [36] with Boolean Implication and Converse Non-Implication operations.

**Hashed history correlation.** Prior profile-guided techniques either consider extremely long histories requiring kilobytes of metadata storage per static branch [35], or utilize short (typically 4 or 8) fixed-length histories that fail to predict many branches accurately [36]. To consider long histories without incurring metadata overhead, we propose hashed history correlation that correlates branch outcomes with a hash of variable-length histories in a profile-guided manner. To find the best history length for predicting a branch, Whisper considers different lengths from a geometric series and picks the length that shows the strongest correlation. Whisper converts histories of that length into a fixed-length (8-bit) hashed history and efficiently encodes this hashed history using Boolean formulas.

**Randomized formula testing.** Determining the optimal boolean formula for predicting the branch outcome based on an $N$-bit history, requires exploring a search space of size $2^N$. To address this challenge, Whisper proposes randomized formula testing, a technique that only considers a random, yet uniform, subset of all prediction formulas as candidates, selecting the best formula for predicting branches. Whisper finds near optimal formulas, comparable to exhaustive exploration (88.3% on average) while considering only 0.1% of all possible prediction formulas.

**Implication and Converse Non-Implication operations.** Besides reducing the search space of Boolean formulas, Whisper also improves their prediction accuracy. In particular, Whisper introduces Implication and Converse Non-Implication that improve prediction accuracy over ROMBF by 1.5% while maintaining the low storage cost of ROMBF.

Whisper enables these three contributions with a novel PGO technique. In particular, it collects the execution profile of data center applications in production using efficient hardware support [37, 38] and then performs an offline branch analysis. The analysis yields optimized ROMBF enabling the injection of brhint instructions for branches that cause frequent mispredictions. The brhint instruction efficiently encodes precise history lengths, a Boolean formula to differentiate taken histories from not-taken histories (and vice versa), and a pointer to the corresponding branch instruction. Using the state-of-the-art profile-guided correlation algorithm [18, 20, 21], Whisper inserts the brhint instruction in a suitable predecessor of the branch at link time to ensure hint timeliness. At run time, Whisper utilizes the hint of a corresponding branch instruction to compare the hashed dynamic history against the Boolean formula for predicting the branch outcome. Thus, Whisper leverages hardware/software co-design to eliminate data center applications’ branch mispredictions in a profile-guided manner.

We evaluate Whisper for 12 popular data center applications that suffer from frequent frontend and misprediction stalls and show that, on average, Whisper eliminates 16.8% of all branch mispredictions over the 64KB state-of-the-art TAGE-SC-L [34] baseline. Due to this 1.7%-32.4% reduction in mispredictions, Whisper achieves an average speedup of 2.8% (0.4%-4.6%) for data center applications. Compared to state-of-the-art profile-guided branch prediction mechanisms [35, 36], Whisper achieves 1.1% greater speedup while reducing 7.9% more branch mispredictions. By injecting brhint instructions, Whisper increases the code footprint by 11.4% and executes 9.8% extra dynamic instructions.

We make the following contributions:

- An extensive investigation of branch instructions’ behavior in data center applications demonstrating that large code footprints of these applications trigger frequent branch mispredictions, significantly limiting the overall performance.
- **Whisper:** a novel profile-guided mechanism to eliminate branch mispredictions in data center applications. Whisper correlates a given branch’s direction with many prior branch directions, efficiently encodes this correlation using Boolean formulas, and improves the overall efficacy of branch prediction.
- A comprehensive evaluation of Whisper for 12 data center applications that shows that Whisper can eliminate costly branch mispredictions (16.8% on average) and achieve substantial performance benefits (2.8% on average).

II. **Branch Prediction Challenges for Data Center Applications**

In this section, we thoroughly investigate the behavior of branch instructions from 12 real-world data center applications to show that branch mispredictions significantly limit their overall performance. Then, we explain why state-of-the-art branch predictors fail to eliminate these branch mispredictions. Finally, we provide valuable insights on how to overcome branch mispredictions for data center applications.

A. **Experimental methodology**

**Data center applications.** Recent work from Facebook and Google reports that their widely-deployed data center applications exhibit multi-megabyte code footprints [1, 3, 6, 11] and consequently lose more than 15% of all pipeline slots directly due to branch mispredictions [4, 5]. Due to large instruction footprints, these applications also lose more than 29% of all pipeline slots due to frontend stalls [3, 4, 5, 6]. Accurate and timely branch predictions can effectively hide a large fraction of these frontend stalls because of the decoupled nature [12, 13] of modern processor frontends [22, 23, 24, 25]. Since these applications and their corresponding workloads are proprietary, we use open-source applications and workloads used by prior
work [1, 11, 18, 20, 21, 39, 40, 41, 42] with large code footprints that similarly cause frequent branch mispredictions and frontend stalls. We describe these data center applications and their workloads in Table I.

**Trace collection and simulation parameters.** We collect these applications’ traces using Intel PT [37] and simulate these traces using the Scarab [61] simulator. Table II lists different simulation parameters that resemble a recent state-of-the-art industry baseline [14, 15].

### B. Why is branch prediction important for data center applications?

To understand the importance of the branch prediction mechanism for data center applications, we perform a limit study to measure the maximum performance benefits of an ideal branch direction predictor over the state-of-the-art 64KB TAGE-SC-L [34] predictor. For this ideal branch predictor, only the prediction direction is ideal, *i.e.*, it always predicts taken and not-taken branches correctly. In Fig. 1, we show that the ideal branch direction predictor achieves an average Instructions Per Cycle (IPC) speedup of 12.4% (1.3%-26.4%) over the state-of-the-art TAGE-SC-L branch predictor.

To understand the reason behind this significant performance gap, we break down the speedup into two categories: (1) speedup due to avoiding branch misprediction stalls (i.e., pipeline squashes [31]) and (2) speedup due to avoiding frontend stalls by performing FDIP [12, 13]. For traditional benchmarks (e.g., SPEC2017), avoiding misprediction stalls is the primary benefit of ideal branch prediction. However, for data center applications, eliminating branch mispredictions is also important as it reduces l-cache misses through FDIP.

As also shown in Fig. 1, among the 12.4% mean IPC speedup provided by the ideal branch predictor, an average IPC speedup of 7.9% (0.7%-17.1%) is provided by eliminating all branch misprediction stalls for these applications. On top of that, the ideal branch predictor achieves an additional 4.5% speedup on average (0.5%-11.5%) by eliminating frontend stalls (I-cache misses) for these applications. Therefore, eliminating branch mispredictions is extremely critical for data center applications.

**C. Why does the state-of-the-art TAGE-SC-L branch predictor fall short?**

We now investigate why the state-of-the-art TAGE-SC-L branch predictor is insufficient for data center applications with large code footprints.

![Fig. 1: Data center application limit study: an ideal branch predictor achieves an average IPC speedup of 12.4% (1.3%-26.4%) over the state-of-the-art 64KB TAGE-SC-L baseline.](image1)

![Fig. 2: Branch Mispredictions Per Kilo Instructions (branch-MPKI) for 12 data center applications: 64KB TAGE-SC-L experiences an average branch-MPKI of 3.0 (0.5-7.2) for these applications.](image2)

**Fig. 2: Branch Mispredictions Per Kilo Instructions (branch-MPKI) for 12 data center applications: 64KB TAGE-SC-L experiences an average branch-MPKI of 3.0 (0.5-7.2) for these applications.**

We now investigate why the state-of-the-art TAGE-SC-L branch predictor is insufficient for data center applications with large code footprints.

Fig. 2 shows the branch-MPKI of 64KB TAGE-SC-L across all 12 data center applications. While measuring the branch-MPKI, we only consider mispredictions caused by conditional branch instructions, following the methodology of 5th Championship Branch Prediction (CBP-5) [62]. As shown in Fig. 2, TAGE-SC-L exhibits a branch-MPKI in the range of 0.5-7.2 (3.0 on average) for the analyzed data center applications. To understand the reason behind these frequent branch mispredictions, we categorize all branch mispredictions TAGE-SC-L induces among four different classes: (1) Compulsory mispredictions, (2) Capacity mispredictions, (3) Conflict mispredictions, and (4) Conditional-on-data mispredictions. We perform this classification by analyzing consecutive accesses of a branch substream—the combination [63, 64, 65, 66, 67, 68, 69] of branch instruction’s Program Counter (PC) and history of different lengths.

**Compulsory** [70, 71, 72] mispredictions occur when TAGE-SC-L predicts a branch for the first time and the predicted direction does not match with the true direction. **Capacity** [70, 71, 72] mispredictions occur when the reuse distance [73, 74] of a branch is too large so that the substream is evicted from the TAGE-SC-L tables. **Conflict** [70, 71, 72] mispredictions occur when the associativity or the replacement mechanism for

---

### Table I: Data center applications and workloads we study.

<table>
<thead>
<tr>
<th>Applications</th>
<th>Workloads</th>
</tr>
</thead>
<tbody>
<tr>
<td>MySQL [43]</td>
<td>Different IPC-C queries [44]</td>
</tr>
<tr>
<td>PostgreSQL [45]</td>
<td>Different pgbench queries [46]</td>
</tr>
<tr>
<td>clang</td>
<td>Building LLVM [48]</td>
</tr>
<tr>
<td>Python [49]</td>
<td>pyperformance benchmarks [50]</td>
</tr>
<tr>
<td>Finagle-chirper [51]</td>
<td>Java Renaissance benchmark suite [52]</td>
</tr>
<tr>
<td>Finagle-http [51]</td>
<td>Java DaCapo benchmark suite [54]</td>
</tr>
<tr>
<td>Cassandra [53]</td>
<td>Facebook’s OSS-performance suite [58]</td>
</tr>
<tr>
<td>Kafka [55]</td>
<td>Different MySQL benchmarks</td>
</tr>
<tr>
<td>Tomcat [56]</td>
<td>Different Java DaCapo benchmarks</td>
</tr>
<tr>
<td>Drupal [57]</td>
<td>Different PostgreSQL benchmarks</td>
</tr>
<tr>
<td>Nordpresso [59]</td>
<td>Different Python benchmarks</td>
</tr>
<tr>
<td>Mediawiki [60]</td>
<td>Different Java DaCapo benchmarks</td>
</tr>
</tbody>
</table>

### Table II: Simulator parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>3.2GHz, 6-wide OOO, 24-entry ITQ, 224-entry ROB, 97-entry RS</td>
</tr>
<tr>
<td>Branch prediction unit</td>
<td>64KB TAGE-SC-L [34] (up to 12-instruction), 8192-entry 4-way BTB, 32-entry RAS, 4096-entry IBTB</td>
</tr>
<tr>
<td>Caches</td>
<td>32KB 8-way L1i, 32KB 8-way L1d, 1MB 16-way unified L2, 10MB 20-way shared L3 per socket</td>
</tr>
</tbody>
</table>

---

**TABLE I: Data center applications and workloads we study.**

**TABLE II: Simulator parameters**
TAGE-SC-L tables is not effective enough to retain the branch substream between two consecutive accesses. Conditional-on-data mispredictions occur when the branch’s direction depends on data values and does not correlate with prior history. Consequently, history-based predictors like TAGE-SC-L cannot achieve high prediction accuracy for such branches [75].

Fig. 3 shows the breakdown of all branch mispredictions TAGE-SC-L incurs across different categories. As shown, the majority of these mispredictions occur due to capacity reasons (on average 76.4%). This result reveals that the working set size of branch substreams for data center applications is significantly larger than the capacity of even the 64KB state-of-the-art TAGE-SC-L branch predictor. Furthermore, this characterization confirms that large instruction footprints of modern data center applications put extreme pressure on branch predictors in addition to the instruction cache, instruction translation lookaside buffer, and branch target buffer as prior works have observed [1, 2, 3, 4, 5, 6, 7, 9, 11, 17, 18, 20, 21, 39, 40, 41, 42].

D. Why do existing profile-guided techniques fall short?

We now investigate the degree to which prior profile-guided branch prediction techniques solve the large branch footprint problem of modern data center applications. We primarily present the analysis for BranchNet [35], the most recent profile-guided branch prediction technique, and ROMBF [36], the most effective profile-guided technique for data center applications in our study. These techniques are hybrid in nature as they use profile-guided techniques for hard-to-predict branches and use TAGE-SC-L for remaining branches.

BranchNet. BranchNet [35] deploys Convolutional Neural Networks (CNNs) for hard-to-predict branches together with traditional online branch predictors (e.g., TAGE-SC-L). To train CNNs for these branches, BranchNet leverages offline profiles from multiple application inputs. At run time, TAGE-SC-L makes predictions for the vast majority of branches while CNNs predict the few hard-to-predict branches. Based on metadata storage, BranchNet also proposes different variants of CNNs: (1) 8KB-BranchNet and (2) 32KB-BranchNet. To understand the potential of CNNs for predicting branches, we also study BranchNet with no storage restrictions, unlimited-BranchNet.

Read-Once Monotone Boolean Formulas (ROMBF). Prior work [36] utilizes Boolean formulas to predict branch outcomes based on history. In particular, every branch outcome in the history represents a Boolean variable that is combined using logical operations (e.g., and, or) to predict a branch’s direction. Branch prediction using Boolean formulas faces two key challenges. First, to determine the optimal Boolean formula that provides the best prediction accuracy for a history length of $N$, the approach has to explore $2^N$ all possible formulas. Second, to encode the Boolean formula, the approach requires $2^N$-bit storage. Prior work [36] addresses only the second challenge by using a subset of Boolean formulas where every variable appears exactly once and by allowing only two logical operations and and or. Consequently, prior work [36] encodes a ROMBF of $N$ variables using only $N \sim 1$ bits. Using such a compact encoding, prior work annotates branch instructions with $N$-bit hints to make branch predictions based on the outcome of the last $N$ branches. The study also proposes different variants of ROMBF (4-bit and 8-bit) for different values of $N$. For brevity, we refer to this prior work [36] as ROMBF.

To assess the potential of these existing profile-guided branch prediction mechanisms, we evaluate BranchNet and ROMBF over the 64KB TAGE-SC-L baseline. As shown in Fig. 4, data center applications do not significantly benefit from these existing mechanisms. Specifically, the state-of-the-art profile-guided technique, BranchNet, reduces only 3.4% and 6.6% of all branch mispredictions with 8KB and 32KB metadata storage. Even with the unlimited metadata storage, BranchNet only avoids 11.9% of all branch mispredictions. On the other hand, ROMBF reduces 8.4% and 8.9% of all branch mispredictions using 4-bit and 8-bit formulas. Next, we investigate the performance of these prior profile-guided techniques to understand why they fail to avoid so many branch mispredictions.

BranchNet employs CNNs to predict hard-to-predict branches assuming that only a few static branches disproportionately cause the vast majority of all mispredictions for an application. For example, as shown in Fig. 5, the top 50 static branches experience more than 60% of all mispredictions for SPEC2017 integer speed benchmarks (e.g., leela, xz, omnetpp, deepsjeng, and mcf). Consequently, for these benchmarks, BranchNet can reduce 12.6%-34% of all
Fig. 5: The distribution of all branch mispredictions across different branch instructions using TAGE-SC-L. In general, SPEC benchmarks satisfy BranchNet’s [35] assumption as only a top-few (e.g., 50) branch instructions cause the majority (e.g., > 60%) of all mispredictions. Data center applications, however, do not satisfy this assumption as mispredictions are distributed across thousands of different branches.

Fig. 6: Distributions of all branch mispredictions among different history lengths. Predicting a branch requires correlating its direction with even 1024 prior branch outcomes.

Fig. 7: Distributions of branch executions among different logical operations used in the Boolean formula to predict a branch. And (28.9%), always-taken (23.3%), converse non-implication (9.2%), implication (8.8%), never-taken (5.9%), and Or (5.3%) operations together can predict more than 80% of all branch executions.

Our investigation reveals that ideal branch prediction significantly improves the performance of data center applications as their large branch footprints exhaust 64KB TAGE-SC-L [34]. State-of-the-art profile-guided mechanisms [35, 36] also fail to eliminate a large majority of branch mispredictions for these applications. We propose Whisper, a combination of three novel profile-guided techniques to improve branch prediction. Whisper introduces hashed history correlation to predict branches that correlate with long variable-length histories. Furthermore, Whisper proposes randomized formula testing to reduce the massive offline training time of existing profile-guided branch prediction techniques [35, 36] without affecting the prediction accuracy. Finally, Whisper extends ROMBF by including Implication and Converse Non-Implication operations to predict branches accurately.

Whisper leverages profile-guided analysis at link time to correlate branches with previous branch outcomes using efficient hardware-based control flow tracing support such as Intel PT [37] and LBR [38]. Next, Whisper maps values of variable-length histories corresponding to different branch outcomes into fixed-length hashed values and encodes these hashed values using an extended ROMBF formula. To pick the formula for any given branch, Whisper considers a randomized subset of all formulas and selects the formula yielding the fewest mispredictions for all hashed histories of the branch. Whisper annotates every hard-to-predict branch with its corresponding formula to provide the branch predictor in hardware with the following information: (1) how many prior branches in the global history are relevant for predicting the current branch, and (2) how the outcome of these prior branches need to be combined to compute the direction of the current branch.
Whisper introduces minor hardware modifications to match the corresponding branch outcome. We describe Whisper’s in-depth usage model in §IV. Now, we discuss the novel techniques Whisper proposes along with its micro-architectural modifications in greater detail.

A. Hashed history correlation

As shown in §II-D, the computational complexity of learning optimal ROMBF and their storage overhead increases linearly with the number of considered variables. In the context of branch prediction, these variables are previous branch directions, i.e., the global branch history leading into a branch. As a result, prior work [36] is limited in accuracy by only considering short histories. To address this challenge, Whisper introduces hashed history correlation, providing three key capabilities: (1) an efficient encoding scheme of large and variable-length histories, (2) a technique to correlate a subset of the branch history with a specific branch outcome, and (3) a mechanism to represent different history values utilizing a single formula.

History hashing. Whisper introduces history hashing that converts the history of any arbitrary length into a fixed length. For example, Whisper transforms the 64-bit history (i.e., the outcome of the most recent 64 branches) into a 16-bit hashed history by dividing the 64-bit value into four 16-bit chunks and applying logical operations (e.g., and, or, xor) to these 16-bit chunks. We empirically study the sensitivity of Whisper’s hashing mechanism for different hashed lengths and different logical operations to find that the 8-bit hash and xor operations provide a good trade-off between instruction footprint overhead and prediction accuracy. As branch predictors used in today’s hardware already use a similar hashing mechanism [76], Whisper does not introduce significant micro-architectural modifications to perform history hashing.

History correlation. Directions for different branches correlate with prior histories of different lengths. Some branches correlate with the outcome of only the most recent branches while other branches correlate with the outcome of relatively older branches. Whisper addresses this challenge by considering various history lengths for each static branch and selecting the length that provides the highest accuracy for that branch using profile samples.

Whisper’s hashed history correlation technique requires three parameters: (1) the minimum history length $a$, (2) the maximum history length $N$, and (3) the number of different history lengths $m$. To find the best history length for a branch, Whisper analyzes all execution samples, referred to as substreams, for that branch using an application profile collected via efficient hardware support (Intel PT [37] and LBR [38]). Each substream contains two components: (1) the actual direction of the branch execution and (2) the directions of the most recent $N$ branches before that branch. Using these scenarios, Whisper determines the best history length and formula for a given branch by evaluating a list of potential history lengths.

For each branch, Whisper considers different history lengths that follow a geometric series [77], up to the $m$-th term, starting with the minimum history length, $a$, i.e., $a, ar, ar^2, \cdots, ar^{m-1}$, where $r = (N/a)^{1/(N-1)}$. At each history length in the series, Whisper encodes the branch history up to this length. As described above, to minimize storage costs, Whisper does not evaluate raw, full-length histories. Instead, Whisper operates on hashed histories, allowing it to compare histories of different original lengths. Next, Whisper determines a Boolean formula that best fits the substream (see §III-B). This is done by using the total number of taken and not-taken counts for the hashed partial history across all samples for that branch. Then, Whisper counts the total number of mispredictions that the current history length and formula incur. If there is a history length that results in the fewest mispredictions for that branch, then that history length is considered the best and used later at run time. If none of these history lengths improve accuracy when compared to the profiled results, then Whisper indicates that the given branch should be predicted in a purely dynamic manner (i.e., using the underlying branch predictor). Additionally, we empirically study Whisper’s sensitivity to different parameters ($a$, $N$, and $m$) and observe that the values $a = 8$, $N = 1024$, and $m = 16$ work well.

History representation. The primary goal of Whisper’s profile-guided analysis is to annotate a static branch with a Boolean formula that efficiently encodes relevant historical branch outcomes to predict the directions of the branch accurately. As we describe in §II-D, in an $N$-bit history, where each branch can be either taken or not-taken, there exist $2^N$ potential branch scenarios. Whisper needs to partition these $2^N$ branch scenarios into two groups using a Boolean formula, where one group reflects the scenarios in which the branch is taken and the other group where the branch is not taken. To achieve this goal, Whisper considers several Boolean formulas for each static branch and selects the Boolean formula that can predict the branch with the highest accuracy based on the collected profile. Algorithm 1 shows a simplified version of the technique Whisper utilizes to find the best formula for representing each branch’s history.

Algorithm 1 takes two hash tables, $T$ and $NT$ as inputs. They contain the hashed history as keys and the number of profile samples as values. $T$ and $NT$ denote taken and not-taken samples respectively. As output, Algorithm 1 generates the Boolean formula, $f$, that incurs the minimum number of mispredictions, $m'$.

As shown in Algorithm 1, Whisper initializes the minimum number of mispredictions, $m'$, with the value $\infty$ (Line 1) and the best Boolean formula, $f$, with a default value of $\emptyset$ (Line 2). Next, Whisper generates the list of all Boolean formulas that will be considered as candidates for predicting the branch (Line 3). We will later (§III-B and §III-C) describe how Whisper finds only a subset of all Boolean formulas that approximates the full potential of all Boolean formulas with high accuracy and efficiency.
Algorithm 1 Finding the best Boolean formula to differentiate taken histories from not-taken histories.

**Input**: $T$ and $NT$ contain different hashed history as keys and the number of profile samples as values. $T$ and $NT$ denote taken and not-taken samples respectively.

**Output**: The Boolean formula, $f$ which incurs the minimum number of mispredictions, $m'$

1: $m' \leftarrow \infty$
2: $f \leftarrow \emptyset$
3: $T \leftarrow \text{List-of-Considered-Formulas}()$
4: for each $f' \in F$ do
5:     $t \leftarrow 0$
6:     for each $k \in T \cdot \text{keys}$ do
7:         if satisfy $(k, f') \neq 1$ then
8:             $t \leftarrow t + T[k]$
9:     for each $k \in NT \cdot \text{keys}$ do
10:        if satisfy $(k, f') = 1$ then
11:           $t \leftarrow t + NT[k]$
12:       if $t < m'$ then
13:           $f \leftarrow f'$
14:           $m' \leftarrow t$
15:       return $(f, m')$

For each formula $f'$, Whisper initializes the total number of mispredictions the formula sustains, $t$, as 0 (Line 5). Next, Whisper iterates over each key-value pair of $T$ (Lines 6-8) and $NT$ (Lines 9-11) to calculate the value of $t$. Since each key $k$ denotes the hashed history, Whisper first determines whether $k$ satisfies the Boolean formula $f'$ (Line 7 and 10 for $T$ and $NT$ respectively). For taken samples ($T$), if $k$ does not satisfy $f'$, predicting the branch using $f'$ will result in mispredictions. Therefore, Whisper adds the corresponding number of profile samples, $T[k]$, to $t$ (Line 8). Similarly, for not-taken samples ($NT$), if $k$ satisfies $f'$, predicting the branch using $f'$ will also result in mispredictions, so Whisper also adds the corresponding number of profile samples, $NT[k]$, to $t$ (Line 11). Thus, Whisper counts the total number of mispredictions $f'$ incurred for all profile samples.

Finally, Whisper compares $t$ with $m'$ to decide whether the current formula, $f'$ causes the minimum number of mispredictions (Line 12). If $t$ is smaller than $m'$, Whisper updates $f$ and $m'$ with the values $f'$ and $t$ correspondingly (Lines 13-14). Whisper produces the final values of $f$ and $m'$ as output after iterating over all formulas from the subset of considered Boolean formulas (Line 15). Next, we explain how Whisper efficiently generates only a subset of all Boolean formulas that effectively achieves the high accuracy of considering all Boolean formulas.

B. Randomized formula testing

As we discuss in §II-D, any $N$-bit variable can take $2^N$ different values. Therefore, finding the best formula that predicts a branch with the least number of mispredictions requires exhaustively searching the search space of all $2^N$ different formulas. For example, predicting a branch based on the outcome of the last 4 branches will require testing 65536 ($= 2^{14}$) different possible formulas. While testing one formula does not depend on the outcome of a different formula, i.e., checking all formulas is embarrassingly parallelizable, it still requires a large amount of computational operations. Whisper leverages randomized formula testing to reduce this exponential search space.

To perform randomized formula testing, Whisper first generates a random permutation of all formulas using the Fisher-Yates shuffle algorithm [78, 79]. The Fisher-Yates shuffle algorithm ensures that Whisper generates the random order only once and reuses this order for all different branches. For each branch, Whisper selects only a fraction of all formulas to consider as potential candidates to predict the branch. Among these selected candidates, Whisper picks the best formula using Algorithm 1. We investigate the implications of randomized formula testing to the fraction of all formulas tested in §V-B (Fig. 15) and show that Whisper achieves comparable performance to exhaustive search (88.3%) even after checking only 0.1% of all Boolean formulas.

C. Implication and Converse Non-Implication

As discussed in §II-D, when considering arbitrary Boolean formulas for $N$-bit variables, we need to evaluate $2^N$ formulas and also need $2^N$-bits of storage for tagging each hard-to-predict branch. As accurate branch prediction often requires significantly larger histories, prior work [36] proposed ROMBF to reduce the storage overheads of these formulas to $N$-bits. Unfortunately, considering every variable only once leads to sub-optimal Boolean formulas as it is impossible to represent formulas where variables appear twice (e.g., $(a\&\&b)||(a\&\&c)$). Whisper addresses the reduced accuracy provided by ROMBF by introducing additional operations such as contradiction, tautology, and, or, implication, and converse non-implication. This approach enables more powerful Boolean formulas, improving branch prediction accuracy while increasing storage only linearly. In particular, Whisper requires $\log_2(op) + \text{hash}(n)$-bits for each formula, where $op$ represents the number of supported operations and $n$ denotes the number of branches considered in the history. As discussed in §III-A, Whisper also utilizes hashing to represent longer histories of size $n$ because fewer bits are produced by the hash function.

Micro-architectural implementation. Adding Implication and Converse Non-Implication requires minor micro-architectural modifications to the original hardware implementation of ROMBF [36]. Fig. 8 shows an implementation for predicting the branch direction based on the outcome of the last two branches ($N = 2$). For two data inputs ($b_0$ and $b_1$), Whisper requires three control inputs ($\bar{b}_0$, $\bar{b}_1$, and $\overline{b_0\&\&\&b_1}$). As a single unit, Whisper produces the outcome of all four logical operations using $b_0$ and $b_1$. Then, Whisper selects the output based on the two control inputs ($\bar{b}_0$ and $\bar{b}_1$) using a $4 \times 1$ multiplexer. Finally, Whisper selects either the output of the multiplexer or its inverted value based on the remaining control input, $\overline{b_0\&\&\&b_1}$ using another $2 \times 1$ multiplexer. Next, we describe how Whisper combines multiple single units in general ($N > 2$).
Fig. 8: Micro-architecture of the Read-Once Monotone Boolean Formulas Whisper extends with Implication and Converse Non-Implication. It shows the single unit to predict a branch based on the outcome of the last 2 branches.

Fig. 9 shows the micro-architectural requirements of Whisper’s mechanism to predict a branch based on the direction of the last 8 branches. Whisper uses four single units that operate on the outcomes of prior branches, $b_0, b_1, \cdots, b_7$. Then, Whisper uses outputs of these single units as inputs to two single units in the next layer. Next, Whisper uses the output of these two single units as inputs to a single unit in the last layer. All of these single units at different layers require $14$ ($2 \times (8 - 1) = 2 \times 7$) control inputs, $\{0\}^7_3$ to $\{0\}^7_0$. Finally, Whisper uses a $2 \times 1$ multiplexer to select either the last layer’s output or its inverted value based on $\{0\}^7_0$.

As shown in Fig. 9, Whisper performs most of the Boolean operations at a single layer in parallel. The longest delay Whisper incurs is due to 3 sequential single units at different layers following the final step that uses the $2 \times 1$ multiplexer. Every single unit has a maximum delay of 5 logic gates: Not gate, And/Or gate, and three gates for the $4 \times 1$ multiplexer. The final step’s maximum delay is 4 logic gates: Not gate and three gates for the $2 \times 1$ multiplexer. The hashing operation does not incur any extra overhead as existing processors already perform similar hashing operations [76]. Thus, Whisper incurs a maximum delay of only 19 logic gates. Even if Whisper can not compute this entire logic in a single cycle, Whisper can easily pipeline these operations, e.g., by registering the results of the first ten operations in one cycle and performing the last nine operations in the next cycle. In any event, Whisper generates its prediction in parallel with TAGE-SC-L, whose logical depth and complexity with hashed SRAM table lookups, tag comparisons, and adder tree for the SC component exceed Whisper’s complexity.

IV. USAGE MODEL

We show the high-level usage model of Whisper in Fig. 10. Whisper collects data center applications’ execution profiles in production and analyzes these profiles offline to inject branch hint instructions.

Run-time profiling. First, Whisper collects the execution trace of branch instructions for data center applications in production (step 1) using Intel PT [37] and LBR [38]. Similar to recent work [3, 18, 20, 21], Whisper leverages Intel PT and LBR as they are widely adopted in today’s data centers [1, 11, 17, 80, 81]. Intel PT captures the trace of dynamically executed branch instructions with low overhead (only up to 1% [82, 83, 84, 85]). As shown in Fig. 10, this trace contains a branch direction (taken, $T$ or not-taken, $NT$) for each branch instruction along with the next instruction’s address when an indirect branch is taken. Intel LBR provides Whisper with the prediction accuracy of each dynamically executed branch instruction for the underlying branch predictor. Similar to PT, LBR also incurs minimal overhead [19, 86].

Branch analysis. Next, in step 2, Whisper analyzes the in-production execution trace of branch instructions. For a static branch instruction, Whisper considers all of its dynamic executions and the profiled processor’s prediction accuracy of the branch to find the best history length using the hashed history correlation technique (§III-A). Also, Whisper determines the best history length for a branch only if Boolean formula-based prediction achieves better accuracy than the profiled processor’s predictor for the branch. For such branches, Whisper injects an extra instruction per branch in the binary specifying hint to predict the branch.
**Hint injection.** Whisper’s offline analysis identifies branches for which history-based Boolean formulas achieve better prediction accuracy than the profiled processor’s predictor. Whisper injects a hint instruction, `brhint`, for each of these branches. A `brhint` instruction includes 4 specific components as we show in Fig. 11.

The first component specifies the history length from a geometric series. As described in §III-A, Whisper uses the geometric series \((i.e., 8, 11, 15, \ldots, 1024)\) with parameter values \(a = 8, N = 1024\), and \(m = 16\) based on empirical results. The 4-bit history specifies which of these 16 history lengths Whisper should use to predict the corresponding branch.

The second component specifies the 15-bit Boolean formula that Whisper uses to predict the branch. As described in §III-C, Whisper needs \(2N - 1\) bits to encode a Boolean formula that predicts a branch based on the outcome of the last \(N\) branches. Consequently, the 15-bit Boolean formula can directly predict a branch with a history length of 8. To predict a branch with longer history lengths \((i.e., 11, 15, \ldots, 1024)\), Whisper transforms the long histories into 8-bit histories via hashing as we describe in §III-A.

The third component specifies the 2-bit bias for always-taken and never-taken branches. The fourth component, PC pointer, specifies the branch instruction’s program counter (PC). Whisper uses a 12-bit offset to represent branch instructions. As described, Whisper uses the branch predictor to predict the branch. Furthermore, Whisper ensures that the branch predictor does not allocate new entries for these branches. Thus, Whisper allows the branch predictor to allocate its storage for the remaining branches and provide better prediction accuracy.

**V. Evaluation**

**A. Methodology**

**Data center applications and their workloads/inputs.** We evaluate Whisper using 12 widely-used data center applications (as described in §II-A). We vary workloads/inputs for these applications by changing different database queries \((e.g., \text{oltp}_\text{read}\_\text{only} \text{vs} \text{oltp}_\text{write}\_\text{only})\), different database scaling factors \((e.g., 100 \text{ vs} 8000)\), different input data and file sizes \((e.g., \text{large} \text{ vs} \text{small})\), different query mapping styles \((e.g., \text{imperative} \text{ vs} \text{declarative})\), different webpages client requests \((e.g., \text{feed} = \text{rss}2 \text{ vs} \text{p}=37)\), different numbers of concurrent clients \((e.g., 2 \text{ vs} 10)\), and different random number seeds \((e.g., 1 \text{ vs} 10)\). We optimize each of these applications with Whisper using the profile from one workload/input and test the performance of Whisper’s optimization on a different workload/input.

**Profile collection.** We collect data center applications’ profile using Intel LBR [38] and PT [37], and use the hardware performance event, “\text{br}_\text{misp}\_\text{retired.conditional}” to identify branch mispredictions.

**Simulation setup.** We evaluate Whisper using Scarab [61] where we implement support for the `brhint` instruction and micro-architectural modifications Whisper proposes. We also modify Scarab to simulate instruction traces collected via Intel PT and evaluate Whisper by simulating 100 million representative, steady-state instructions for each application using simulation parameters listed in Table II.

**B. Performance analysis**

**Speedup.** We show Whisper’s speedup for 12 data center applications in Fig. 12. For comparison, we also show speedups that recent techniques (different variants of ROMB [36] and BranchNet [35]) offer. To understand the limit, we also show speedups provided by the ideal branch predictor and MTAGE-SC, the best predictor in the unlimited storage category of CBP-5 [62]. As shown, Whisper provides an average speedup of 2.8% (0.4%-4.6%) that is 44.1% of the average speedup (6.3%) MTAGE-SC achieves with unlimited storage.

The speedup gap between Whisper and MTAGE-SC originates from several reasons. Whisper can not eliminate some mispredictions for previously unobserved branch instructions as Whisper optimizes applications using only one different input.
profile in this case. We quantify the performance implications of this input sensitivity later in this section. Furthermore, the brhint instructions Whisper injects incur static and dynamic instruction increases which we also quantify later in the section. Nevertheless, Whisper achieves greater speedup than prior works, ROMBF and BranchNet, as they only provide 1.7% and 0.8% on average. Furthermore, on average, Whisper provides greater speedup than BranchNet even when it leverages unlimited metadata storage. Next, we investigate how Whisper achieves this speedup by reducing a substantial amount of branch mispredictions.

**Misprediction reduction.** We evaluate how well Whisper reduces branch mispredictions compared to prior techniques and show the results in Fig. 13. As shown, on average, Whisper reduces 16.8% of all branch mispredictions (1.7%-32.4%) the TAGE-SC-L baseline incurs for these data center applications and significantly outperforms all prior mechanisms. Specifically, Whisper reduces 7.9% more mispredictions than the best performing prior technique that can be used in a practical scenario. Furthermore, Whisper outperforms the state-of-the-art, BranchNet, by 4.9% even when BranchNet uses unlimited metadata storage. This unlimited-BranchNet outperforms Whisper only for three applications (mediawiki, python, and wordpress) that exhibit the behavior BranchNet assumes, i.e., the top-few branch instructions cause the majority of all mispredictions, as shown in Fig. 5. Nevertheless, Whisper eliminates more mispredictions than the practical variants (8KB and 32KB) of BranchNet even for these three applications as shown in Fig. 13. Next, we provide a breakdown of mispredictions Whisper eliminates among different sources of optimizations.

**Breakdown of misprediction reduction.** In Fig. 14, we show the contributions of hashed history correlation and Implication and Converse Non-Implication to Whisper’s overall performance. We quantify the reduction in branch mispredictions these two novel techniques offer over 8-bit ROMBF. As shown, hashed history correlation achieves an average misprediction reduction of 6.4% while Implication and Converse Non-Implication eliminate 1.5% of all mispredictions.

### Implications of randomized formula testing and training time.

Whisper’s randomized formula testing does not eliminate any new mispredictions. Instead, randomized formula testing reduces Whisper’s offline training time (i.e., time to find the best Boolean formula to predict a branch) without sacrificing prediction accuracy. Fig. 15 shows this tradeoff between Whisper’s average misprediction reduction and average training time with an increase in the percentage of formulas Whisper explores via randomized formula testing. As shown, Whisper eliminates 16.8% of all mispredictions even after exploring only 0.1% of all formulas. This reduction is comparable (88.3% on average) to the reduction Whisper achieves after considering 100% of all formulas. In terms of training time, randomized formula testing is also efficient as it reduces the exploration time by an order of magnitude (Fig. 15). Consequently, Whisper’s training time is lower than training times for 8-bit ROMBF and BranchNet (Fig. 16).

### Performance across different workloads/inputs.

As we mention in §V-A, we optimize data center applications with Whisper using the profile from one input and test the performance of Whisper’s optimization on a different input. Now, we investigate Whisper’s performance across three separate input configurations (‘#0’ to ‘#3’). We optimize each application using the training input’s profile ‘#0’ and measure mispredictions Whisper eliminates for different test inputs ‘#1’, ‘#2’, ‘#3’. For each input, we also measure the performance when Whisper optimizes the application with the same input’s profile. As shown in Fig. 17, Whisper avoids 6.6% more mispredictions with input-specific profiles compared to profiles that are not input-specific.

To address this input sensitivity, prior work [35] recommended merging profiles from multiple inputs. We study the impact of merging profiles on Whisper’s performance in Fig. 18. We compare Whisper’s performance against prior works after merging profiles from different application inputs. As shown, Whisper outperforms prior techniques even for merged profiles. Furthermore, Whisper’s effectiveness increases as profiles from multiple inputs are merged.
Design parameter | Value | Design parameter | Value
--- | --- | --- | ---
Minimum history length | 8 | Length of the hashed history | 8
Maximum history length | 1024 | Logical operations used | 4
Different history lengths | 16 | Hint buffer’s size | 32

**Fig. 13:** Whisper’s reduction in branch mispredictions compared with BranchNet and ROMBF: Whisper eliminates 7.9% more mispredictions than the best performing realistic prior work. Whisper even removes 4.9% more mispredictions than the unlimited-BranchNet.

**Fig. 14:** Misprediction reduction (%) achieved by hashed history correlation and Implication and Converse Non-Implication over 8-bit ROMBF: hashed history correlation reduces more branch mispredictions than Implication and Converse Non-Implication.

**Fig. 15:** Thanks to randomized formula testing, Whisper achieves high misprediction reduction even after exploring only 0.1% of all formulas (left) while significantly reducing the training time (right, the y-axis is log-10 scale).

**Hint overhead.** Unlike BranchNet, Whisper does not incur any extra metadata overhead. Hence, Whisper’s only overhead is `brhint` instructions added in the program binary and executed at run time. We estimate the static and dynamic overhead of these `brhint` instructions in Fig. 19. As shown, on average, Whisper increases these applications’ static footprint by 11.4% (9.8%-13%) while introducing 9.8% (5.3%-14.7%) extra dynamic instructions.

**Sensitivity analysis.** As we describe in §III, Whisper’s design includes several parameters including a minimum, maximum, and different history lengths, hashed history length, different logical operations used, and hint buffer’s size. We determine these parameters’ values empirically via sensitivity studies. For brevity, we do not present detailed results corresponding to these studies. As a summary, Table III shows these parameters’ values we use to evaluate Whisper.

**128KB TAGE-SC-L as baseline.** We evaluate Whisper’s effectiveness for a much larger, 128KB TAGE-SC-L baseline and show the results in Fig. 20. The 128KB TAGE-SC-L exhibits a branch-MPKI in the range of 0.4-5.4 (2.4 on average) for 12 data center applications. As shown, Whisper achieves an average misprediction reduction of 13.4% over the 128KB TAGE-SC-L baseline highlighting Whisper’s effectiveness even for a larger TAGE-SC-L branch predictor.

**Predictor size.** We evaluate Whisper’s sensitivity to the baseline branch predictor’s size by varying TAGE-SC-L’s capacity from 8KB to 1MB. Fig. 21 shows the results. As shown, Whisper consistently reduces more than 10% of all mispredictions

---

**TABLE III: Different design parameters’ values.**

<table>
<thead>
<tr>
<th>Design parameter</th>
<th>Value</th>
<th>Design parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Minimum history length</td>
<td>8</td>
<td>Length of the hashed history</td>
<td>8</td>
</tr>
<tr>
<td>Maximum history length</td>
<td>1024</td>
<td>Logical operations used</td>
<td>4</td>
</tr>
<tr>
<td>Different history lengths</td>
<td>16</td>
<td>Hint buffer’s size</td>
<td>32</td>
</tr>
</tbody>
</table>

---

**Fig. 16:** Average training time for Whisper compared to prior techniques (the y-axis is log-10 scale): BranchNet requires training times of more than thousands of seconds, even when trained on an NVIDIA Tesla V100 GPU. The training time for ROMBF grows exponentially with an increase in history length. The training time for Whisper is significantly lower than training times for 8-bit ROMBF and BranchNet.
irrespective of the predictor’s capacity. Even the 1MB TAGE-SC-L incurs an average branch-MPKI of 1.9 compared to MTAGE-SC’s branch-MPKI of 1.4. As even the 1MB TAGE-SC-L suffers from capacity and conflict mispredictions, Whisper still has the potential to reduce a significant number of mispredictions. Consequently, Whisper reduces mispredictions by 11.2% for the 1MB TAGE-SC-L.

**Predictor warm-up.** We evaluate Whisper’s sensitivity to baseline branch predictor’s (TAGE-SC-L) state by varying % of warm-up instructions from 0% to 90%. Fig. 22 shows the results. As shown, Whisper reduces all mispredictions TAGE-SC-L incurs by 17.5% without any warm-up. As TAGE-SC-L’s warm-up period increases and TAGE-SC-L incurs fewer mispredictions, Whisper’s average misprediction reduction (%) over to TAGE-SC-L drops slightly. Nevertheless, Whisper still avoids a large number of mispredictions as it reduces TAGE-SC-L’s mispredictions by 16.8% even when warm-up instructions account for 50% of all instructions.

**Simulated instructions.** We evaluate Whisper’s sensitivity to the total number of instructions simulated by varying the number of instructions from 100 million to 1 billion. Fig. 23 shows the results. As shown, Whisper reduces 14.7% of all mispredictions even when one billion instructions are simulated.

**VI. RELATED WORK**

**PGO for data center applications.** The large instruction footprint and software complexity of modern data center applications make them a prime target for PGO [3,4,5,6,7,90,91]. Prior PGO techniques include code layout optimizations [1,11,17,27,33,92,93,94,95,96,97,98], L-cache prefetching [3,18] and replacement [20], and BTB prefetching [21] and replacement [99]. These techniques primarily
focus on reducing frontend stalls while Whisper focuses on reducing branch mispredictions. Consequently, Whisper should be equally effective even in the presence of these techniques.

Online branch predictors. Most state-of-the-art online branch predictors are variants of TAGE [100] and Perceptron [101]. TAGE hashes global branch and path histories of different lengths to index into various tables composed of tagged saturating counters. TAGE-SC-L [34, 102], which won CBP-5 [62], is a popular TAGE variant that uses additional loop predictor and statistical corrector components to improve accuracy. Perceptron-based predictors, such as the Multi-perspective Perceptron [103, 104], use a single-layer neural network to compute a sum of weights that represent a learned correlation in branch history. A fundamental limitation of TAGE and Perceptron-based predictors is their inability to learn increasingly complex branch histories due to storage and run-time constraints. Other work in online branch prediction includes domain-specific branch predictors and predictors targeting data-dependent branches [105, 106, 107, 108].

Fig. 22: Whisper’s performance for various TAGE-SC-L warm-up periods: Whisper reduces 16.8% of TAGE-SC-L’s mispredictions with 50% of instructions considered as warm-up. On the other hand, Whisper avoids 17.5% of TAGE-SC-L’s mispredictions without any warm-up.

Fig. 23: Whisper’s performance for various numbers of simulated instructions: on average, Whisper avoids 14.7% of all mispredictions after simulating one billion instructions.

Summary. Whisper achieves significant reductions in branch mispredictions compared to existing predictors and offline methods. It outperforms BranchNet by using efficient hardware to predict branch histories and leverages existing online branch predictors for accuracy. This demonstrates the potential for integration of machine learning and hardware techniques to improve performance in modern architectures.

VII. Conclusion

The state-of-the-art branch predictor, TAGE-SC-L, suffers frequent branch mispredictions for data center applications as their large branch footprints overwhelm TAGE-SC-L’s 64KB capacity. We propose, Whisper, a profile-guided hardware/software mechanism to efficiently reduce branch mispredictions in these data center applications through extended Read-Once Monotone Boolean Formulas that encode hard-to-predict correlations in branch history. Whisper inserts lightweight formulas in application code at link time using a new brhint instruction that is complemented by micro-architectural support for ROMBF. Through efficient offline analysis of application profiles, only select branches use these new micro-architectural changes; the remaining are predicted using the underlying branch predictor – requiring no changes to the predictor itself. On average, Whisper reduces 16.8% (1.7%-32.4%) of branch mispredictions over TAGE-SC-L for 12 widely-used data center applications, with an average speedup of 2.8% (0.4%-4.6%), and outperforms existing profile-guided branch prediction mechanisms, such as BranchNet, by 7.9%.

Acknowledgments

We thank the anonymous reviewers for their insightful feedback. This work was supported by generous gifts from Intel Labs, Google, NSF/Intel joint grants #2010810 and #1912617, NSF grants #1942754 and CNS-1938064, a Rackham Predoctoral Fellowship, and the Applications Driving Architectures (ADA) Research Center, a JUMP Center cosponsored by SRC and DARPA. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the funding agencies. We thank Jaekyu Lee, Jumana Mundichipparakkal, Prakash Ramakhyani, José A. Joao, Matt Horsnell from ARM, Akansha Jain from Google, Zhenhang He from the University of Michigan, and Siavash Zangeneh from the University of Texas at Austin for helping us at various stages of this work.