2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO) | 978-1-6654-6272-3/22/$31.00 ©2022 IEEE | DOI: 10.1109/MICRO56248.2022.00017

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
§djimenez@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[1] 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

[1]we use 'branch misprediction' and 'misprediction' interchangeably

978-1-6654-6272-3/22/$31.00 ©2022 IEEE
DOI 10.1109/MICRO56248.2022.00017

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^{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

TABLE I: Data center applications and workloads we study.

Applications	Workloads
`MySQL` [43]	Different `TPC-C` queries [44]
`PostgreSQL` [45]	Different `pgbench` queries [46]
`Clang` [47]	Building `LLVM` [48]
`Python` [49]	`pyperformance` benchmarks [50]
`Finagle-chirper` [51] `Finagle-http` [51]	Java Renaissance benchmark suite [52]
`Cassandra` [53] `Kafka` [55] `Tomcat` [56]	Java DaCapo benchmark suite [54]
`Drupal` [57] `Wordpress` [59] `Mediawiki` [60]	Facebook's OSS-performance suite [58]

TABLE II: Simulator parameters

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

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 I-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.

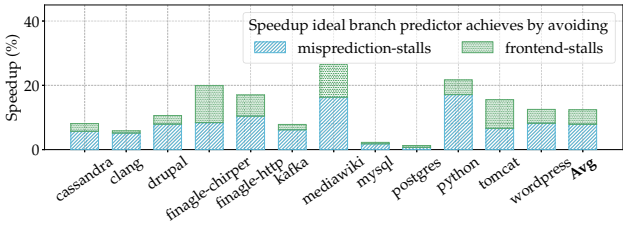


Fig. 1: Data center application limit study: an ideal branch predictor achieves an average IPC speedup of 12.4% (1.3%-26.4%) over the state-of-the-art 64KB TAGE-SC-L baseline.

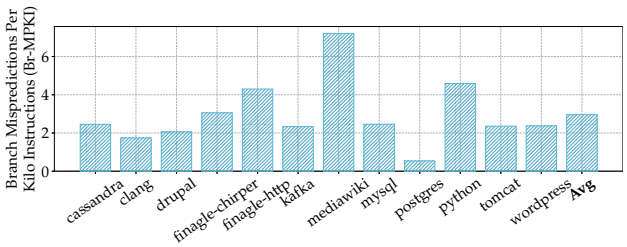


Fig. 2: Branch Mispredictions Per Kilo Instructions (branch-MPKI) for 12 data center applications: 64KB TAGE-SC-L experiences an average branch-MPKI of 3.0 (0.5-7.2) for these applications.

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. 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 5^{th} 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

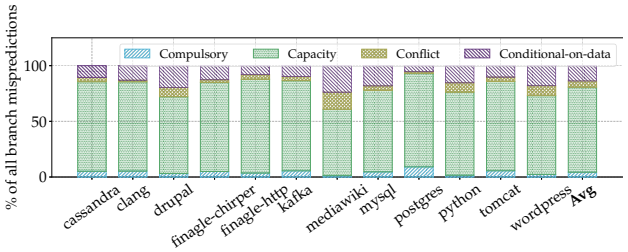


Fig. 3: Breakdown of all branch mispredictions among 4 different classes [70, 71, 72]: data center applications suffer from frequent branch mispredictions primarily (76.4% of all mispredictions) due to capacity issues.

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

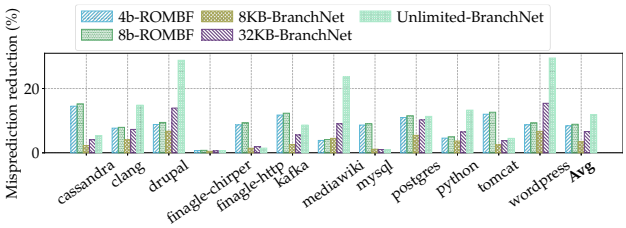
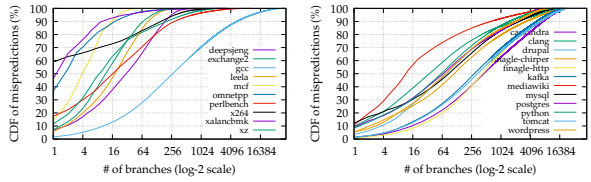


Fig. 4: Performance of prior profile-guided branch prediction techniques [35, 36] over the 64KB TAGE-SC-L baseline: these techniques reduce only 3.4%-8.9% of all branch mispredictions TAGE-SC-L incurs. Even with unlimited storage, this impractical variant of BranchNet [35] achieves an average misprediction reduction of only 11.9%.

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^{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 $N-1$ 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



(a) SPEC2017 integer benchmarks (b) Data center applications

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.

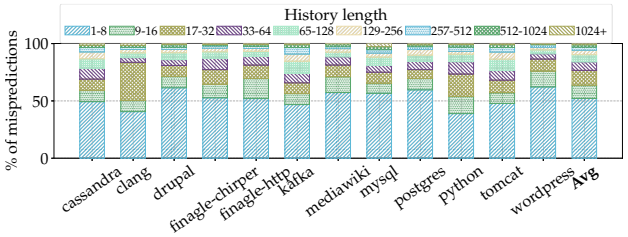


Fig. 6: Distributions of all branch mispredictions among different history lengths. Predicting a branch requires correlating its direction with even 1024 prior branch outcomes.

mispredictions by allocating 256B-2KB metadata storage for each of these branches' CNNs. However, as also shown in Fig. 5, mispredictions for data center applications and `gcc` (from SPEC) are more uniformly distributed across many static branches. Consequently, for these applications, even unlimited-BranchNet can only avoid 11.9% of all mispredictions while using 2KB CNNs for each static branch.

ROMBF predicts a branch by applying an N-bit formula to the last N branch outcomes. For example, 4-bit and 8-bit formulas can predict branches based on only the last 4 and 8 branch outcomes. As shown in Fig. 6, most branches in our data center applications correlate with branch histories of size 32-1024 and, consequently, 4-bit and 8-bit formulas are insufficient. As ROMBF requires N-bit hints to consider N-bit histories, it does not scale well for long branch histories.

Furthermore, ROMBF only considers `and` and `or` operators to compute Boolean formulas along with contradiction (*i.e.*, never taken) and tautology (*i.e.*, always taken). This limitation assumes that these formulas can encode relevant histories for the large majority of branches without any quantitative insight. We characterize the implications of this assumption in Fig. 7 by showing the distribution of all branches among formulas using contradiction, tautology, `and`, `or`, implication, and converse non-implication. As shown, while formulas using `and` (28.9%) and `or` (5.3%) operations represent histories of a significant number of branches, formulas using implication (8.8%) and converse non-implication (9.2%) operations also

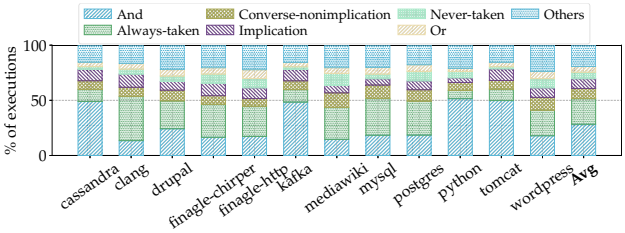


Fig. 7: Distributions of branch executions among different logical operations used in the Boolean formula to predict a branch. And (28.9%), always-taken (23.3%), converse non-implication (9.2%), implication (8.8%), never-taken (5.9%), and Or (5.3%) operations together can predict more than 80% of all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

encode histories of a large number of branches. Consequently, ROMBF can not avoid mispredictions for these branches.

In §V-B (Fig. 16) we will show that BranchNet requires orders of magnitude higher training time than ROMBF, while *Whisper* outperforms both approaches. Next, we use the insights from these characterizations to design *Whisper*, our profile-guided technique to eliminate branch mispredictions for data center applications.

III. Design of Whisper

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 dynamic history with an annotated formula, predicting 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 $r = (\frac{N}{a})^{(\frac{1}{m-1})}$. 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 ∞ (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 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.

FIND-BOOLEAN-FORMULA (T, NT)

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' ← ∞
2:  f ← ∅
3:  F ← List-of-Considered-Formulas ()
4:  for each f' ∈ F do
5:      t ← 0
6:      for each k ∈ T.keys do
7:          if satisfy (k, f') ≠ 1 then
8:              t ← t + T[k]
9:      for each k ∈ NT.keys do
10:         if satisfy (k, f') = 1 then
11:             t ← t + NT[k]
12:     if t < m' then
13:         f ← f'
14:         m' ← 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^{2^N} different formulas. For example, predicting a branch based on the outcome of the last 4 branches will require testing 65536 ($=2^{2^4}$) 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^{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 $log_2(op) * hash(n)$-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 (Ⓞ$_0$, Ⓞ$_1$, and Ⓘ). 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 (Ⓞ$_0$ and Ⓞ$_1$) using a 4×1 multiplexer. Finally, *Whisper* selects either the output of the multiplexer or its inverted value based on the remaining control input, Ⓘ using another 2×1 multiplexer. Next, we describe how *Whisper* combines multiple single units in general ($N>2$).

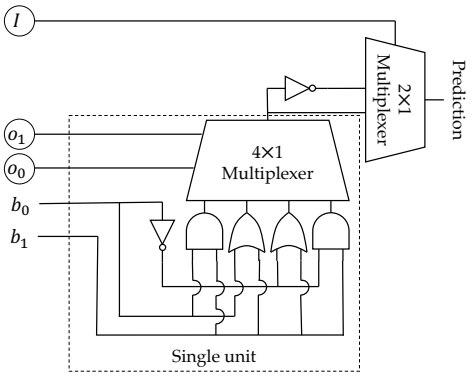


Fig. 8: Micro-architecture of the Read-Once Monotone Boolean Formulas *Whisper* extends with Implication and Converse Non-Implication. It shows the single unit to predict a branch based on the outcome of the last 2 branches.

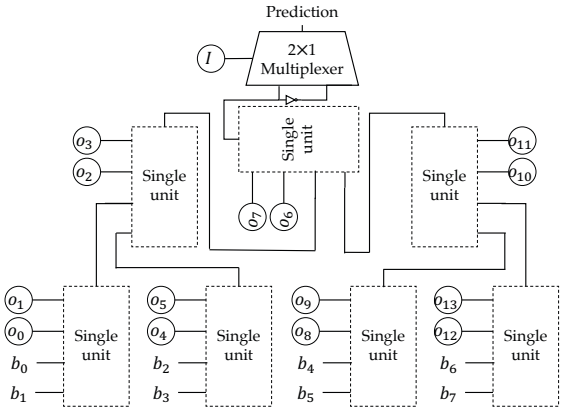


Fig. 9: Micro-architecture showing how *Whisper* combines multiple single units in general. This shows how *Whisper* predicts a branch based on the outcome of the last 8 branches.

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, $\textcircled{o_0}$ to $\textcircled{o_{13}}$. Finally, *Whisper* uses a 2×1 multiplexer to select either the last layer's output or its inverted value based on $\textcircled{I}$.

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 2×1 multiplexer. Every single unit has a maximum delay of 5 logic gates: Not gate, And/Or gate, and three gates for the 4×1 multiplexer. The final step's maximum delay is 4 logic gates: Not gate and three gates for the 2×1 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

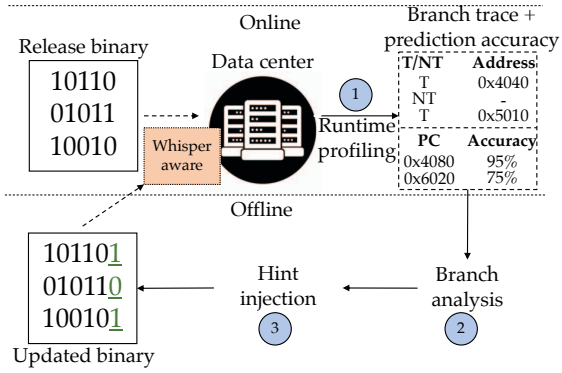


Fig. 10: *Whisper*'s usage model.

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.

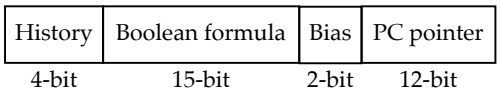


Fig. 11: Different components of the `brhint` instruction *Whisper* proposes.

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.*, $8, 11, 15, \cdots, 1024$) 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.*, $11, 15, \cdots 1024$), *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 instruction pointers as such an offset is enough to cover the vast majority ($> 80\%$) of all branch instructions [21, 87].

Instead of directly encoding the hint in the branch instruction, *Whisper* injects a separate `brhint` instruction for mainly two reasons. First, it avoids the instruction footprint growth for branch instructions for which *Whisper* does not inject any hint as these branches are predicted dynamically. Second, it also ensures hint timeliness by avoiding the requirement of pre-decoding the branch instruction. Conditional branch instructions in x86 format already support similar prefix opcodes for biased branches [88]. We extend these opcodes with additional bytes to implement the `brhint` instruction.

For a given branch, *Whisper* injects the corresponding `brhint` instruction in one of the predecessor basic blocks for the branch. To find the appropriate predecessor, *Whisper* leverages the execution trace collected in production and applies a conditional probability-based correlation algorithm [18, 20, 21].

Run-time hint usage. *Whisper* produces an updated binary for an application after injecting the `brhint` instructions. This updated binary is deployed in production during the next build and deployment cycle. When a data center application executes a `brhint` instruction at run time, *Whisper* places the corresponding four parameters in a small hint buffer. We empirically study *Whisper*'s sensitivity to the size of this hint buffer and observe that *Whisper* provides high performance even with a 32-entry hint buffer.

At run time, while predicting a branch, *Whisper* simultaneously queries the branch predictor (*e.g.*, TAGE-SC-L) and the hint buffer. For branch PCs currently not in the hint buffer, *Whisper* uses the branch predictor to predict the branch. If the hint buffer includes the branch PC, *Whisper* uses the hint information and the micro-architectural implementation described in §III-C 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.*, `oltp_read_only` vs `oltp_write_only`), different database scaling factors (*e.g.*, `100` vs `8000`), different input data and file sizes (*e.g.*, `large` vs `small`), different query mapping styles (*e.g.*, `imperative` vs `declarative`), different webpages client requests (*e.g.*, `feed=rss2` vs `p=37`), different numbers of concurrent clients (*e.g.*, `2` vs `10`), and different random number seeds (*e.g.*, `1` 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, "`br_misp_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 ROMBF [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

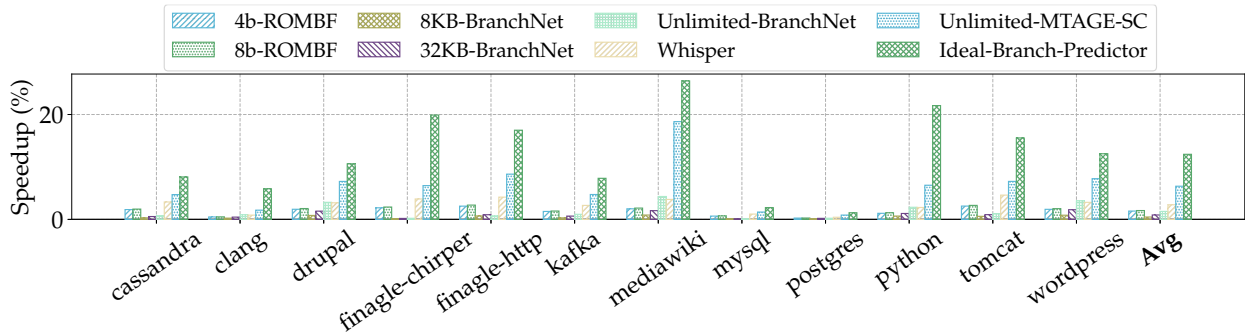


Fig. 12: Speedup over 64KB TAGE-SC-L: *Whisper* achieves an average speedup of 2.8% (0.4%-4.6%) and outperforms state-of-the-art profile-guided prediction techniques [35, 36]. *Whisper*'s speedup corresponds to 44.1% of the speedup MTAGE-SC offers with unlimited storage [89].

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 ('`#1`' 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.

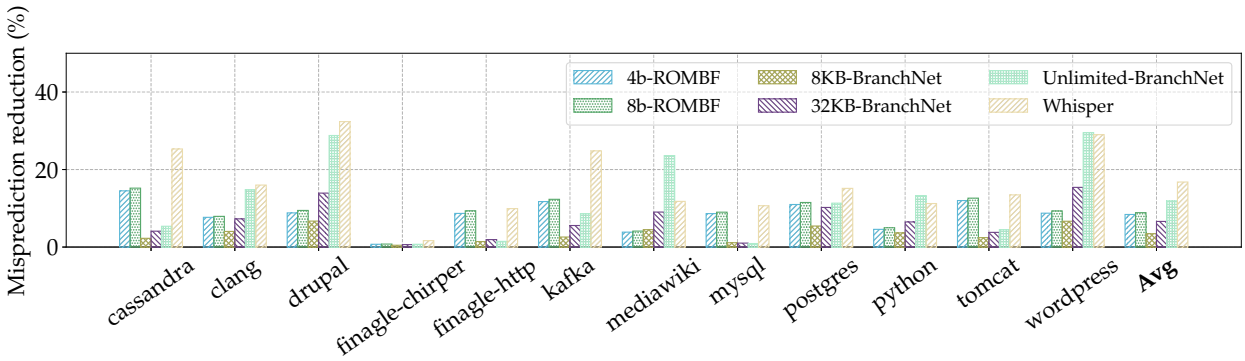


Fig. 13: *Whisper*'s reduction in branch mispredictions compared with BranchNet and ROMBF: *Whisper* eliminates 7.9% more mispredictions than the best performing realistic prior work. *Whisper* even removes 4.9% more mispredictions than the unlimited-BranchNet.

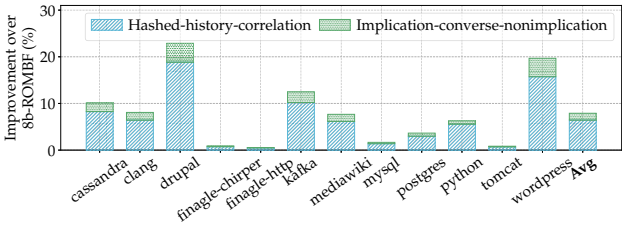


Fig. 14: Misprediction reduction (%) achieved by hashed history correlation and Implication and Converse Non-Implication over 8-bit ROMBF: hashed history correlation reduces more branch mispredictions than Implication and Converse Non-Implication.

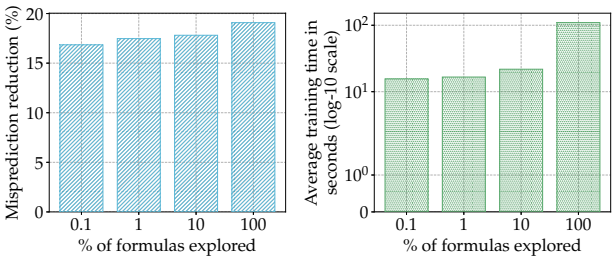


Fig. 15: Thanks to randomized formula testing, *Whisper* achieves high misprediction reduction even after exploring only 0.1% of all formulas (left) while significantly reducing the training time (right, the y-axis is log-10 scale).

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

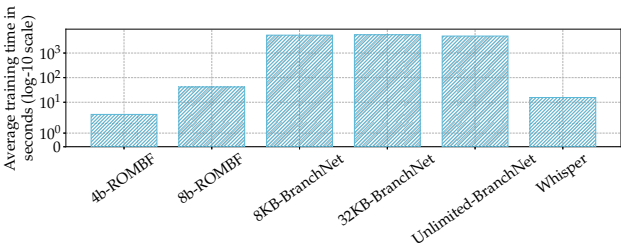


Fig. 16: Average training time for *Whisper* compared to prior techniques (the y-axis is log-10 scale): BranchNet requires training times of more than thousands of seconds, even when trained on an NVIDIA Tesla V100 GPU. The training time for ROMBF grows exponentially with an increase in history length. The training time for *Whisper* is significantly lower than training times for 8-bit ROMBF and BranchNet.

TABLE III: Different design parameters' values.

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

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

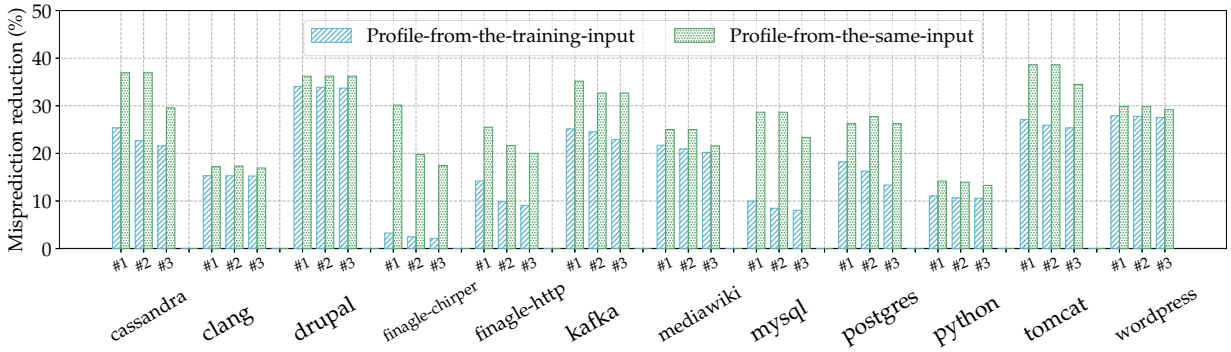


Fig. 17: *Whisper*'s performance for various application inputs: On average *Whisper* reduces 6.6% more branch mispredictions with input-specific profiles compared to profiles from different inputs.

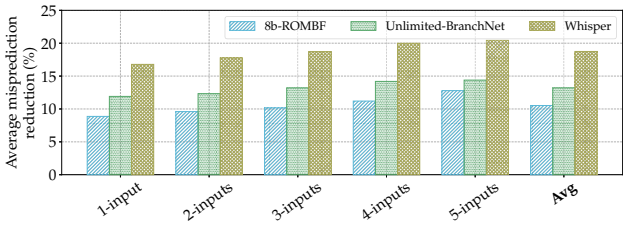


Fig. 18: *Whisper* eliminates more branch mispredictions after merging profiles from various inputs.

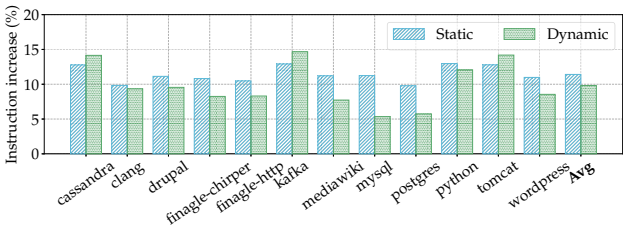


Fig. 19: *Whisper*'s overhead in static and dynamic instruction increase: on average, *Whisper* incurs a static overhead of 11.4% (9.8%-13%) and executes 9.8% (5.3%-14.7%) extra dynamic instructions due to `brhint` instructions.

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.

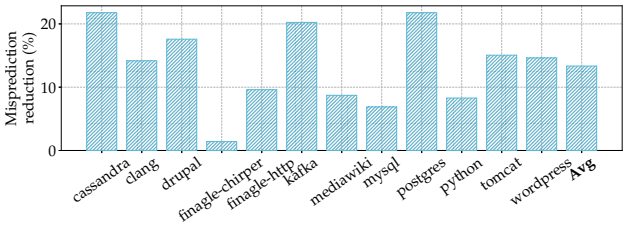


Fig. 20: *Whisper*'s reduction in branch mispredictions over the 128KB TAGE-SC-L baseline: *Whisper* reduces 13.4% of all mispredictions the 128KB TAGE-SC-L incurs.

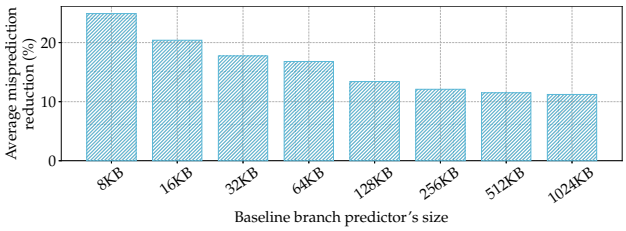


Fig. 21: *Whisper*'s performance for various baseline branch predictor's sizes: *Whisper* reduces even 1MB TAGE-SC-L's mispredictions by 11.2%.

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], I-cache prefetching [3, 18] and replacement [20], and BTB prefetching [21] and replacement [99]. These techniques primarily

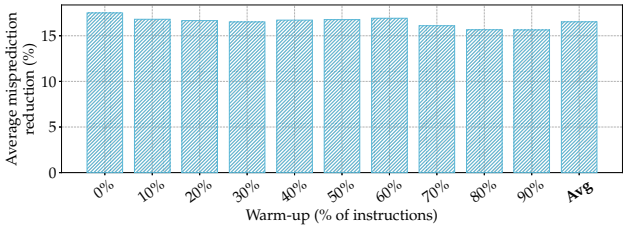


Fig. 22: *Whisper*'s performance for various TAGE-SC-L warm-up periods: *Whisper* reduces 16.8% of TAGE-SC-L's mispredictions with 50% of instructions considered as warm-up. On the other hand, *Whisper* avoids 17.5% of TAGE-SC-L's mispredictions without any warm-up.

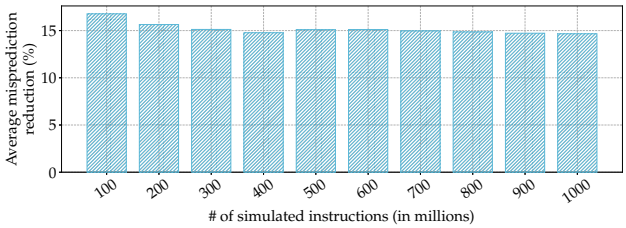


Fig. 23: *Whisper*'s performance for various numbers of simulated instructions: on average, *Whisper* avoids 14.7% of all mispredictions after simulating one billion instructions.

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].

Considering prior limitations, *Whisper* still leverages online branch predictors in the common case. Offline profiling and hardware support for ROMBF are then used to predict branches that online predictors struggle to predict accurately. This approach allows *Whisper* to reduce the resource burden placed on traditional online predictors from applications with noisy branch histories. Also, *Whisper* does not attempt to alter existing online branch predictors in hardware, which simplifies its implementation in modern processors.

Offline methods for branch prediction. Offline techniques, such as profiling and compiler-based optimizations, have been used extensively to improve accuracy for branch prediction [36, 75, 109, 110, 111, 112, 112, 113, 114, 115, 116, 117, 118, 119, 120, 121, 122, 123, 124, 125]. BranchNet [35] is a recent offline method for reducing branch mispredictions. It uses CNNs, with a hardware-based inference component, to handle branches that online predictors struggle to predict accurately. The main limitation of BranchNet is its resource requirements (*i.e.*, multiple GPUs for efficient training, one CNN model per static branch) and implementation complexity in hardware. Whereas for *Whisper*, "training" or analyzing execution profiles can be done relatively cheaply using commodity CPUs and the hardware implementation is less demanding than hardware inference for deep learning. Additionally, BranchNet struggles to cover mispredictions spread out across many unique static branches. *Whisper* has less overhead per static branch due to the lightweight design of ROMBF compared to a CNN model in BranchNet.

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 Ramrakhyani, José A. Joao, Matt Horsnell from ARM, Akanksha 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.

REFERENCES

[1] M. Panchenko, R. Auler, L. Sakka, and G. Ottoni, "Lightning bolt: powerful, fast, and scalable binary optimization," in *Proceedings of the 30th ACM SIGPLAN International Conference on Compiler Construction*, 2021, pp. 119–130.

[2] G. Ottoni and B. Liu, "Hhvm jump-start: Boosting both warmup and steady-state performance at scale," in *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2021, pp. 340–350.

[3] G. Ayers, N. P. Nagendra, D. I. August, H. K. Cho, S. Kanev, C. Kozyrakis, T. Krishnamurthy, H. Litz, T. Moseley, and P. Ranganathan, "Asmdb: understanding and mitigating front-end stalls in warehouse-scale computers," in *Proceedings of the 46th ISCA*, 2019.

[4] A. Sriraman, A. Dhanotia, and T. F. Wenisch, "Softsku: Optimizing server architectures for microservice diversity@ scale," in *Proceedings of the 46th International Symposium on Computer Architecture*, 2019, pp. 513–526.

[5] G. Ayers, J. H. Ahn, C. Kozyrakis, and P. Ranganathan, "Memory hierarchy for web search," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2018, pp. 643–656.

[6] S. Kanev, J. P. Darago, K. Hazelwood, P. Ranganathan, T. Moseley, G.-Y. Wei, and D. Brooks, "Profiling a warehouse-scale computer," in *Proceedings of the 42nd ISCA*, 2015.

[7] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi, "Clearing the clouds: a study of emerging scale-out workloads on modern hardware," *Acm sigplan notices*, vol. 47, no. 4, pp. 37–48, 2012.

[8] K. Adams, J. Evans, B. Maher, G. Ottoni, A. Paroski, B. Simmers, E. Smith, and O. Yamauchi, "The hiphop virtual machine," in *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, 2014, pp. 777–790.

[9] G. Ottoni, "Hhvm jit: A profile-guided, region-based compiler for php and hack," in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2018, pp. 151–165.

[10] M. Ugur, C. Jiang, A. Erf, T. A. Khan, and B. Kasikci, "One profile fits all: Profile-guided linux kernel optimizations for data center applications," *ACM SIGOPS Operating Systems Review*, vol. 56, no. 1, pp. 26–33, Jun. 2022.

[11] M. Panchenko, R. Auler, B. Nell, and G. Ottoni, "Bolt: a practical binary optimizer for data centers and beyond," in *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2019, pp. 2–14.

[12] G. Reinman, T. Austin, and B. Calder, "A scalable front-end architecture for fast instruction delivery," *ACM SIGARCH Computer Architecture News*, vol. 27, no. 2, pp. 234–245, 1999.

[13] G. Reinman, B. Calder, and T. Austin, "Fetch directed instruction prefetching," in *MICRO-32. Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture*. IEEE, 1999, pp. 16–27.

[14] Y. Ishii, J. Lee, K. Nathella, and D. Sunwoo, "Rebasing instruction prefetching: An industry perspective," *IEEE Computer Architecture Letters*, 2020.

[15] ——, "Re-establishing fetch-directed instruction prefetching: An industry perspective," *IEEE International Symposium on Performance Analysis of Systems and Software*, 2021.

[16] K. Pettis and R. C. Hansen, "Profile guided code positioning," in *Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*, 1990, pp. 16–27.

[17] D. Chen, T. Moseley, and D. X. Li, "Autofdo: Automatic feedback-directed optimization for warehouse-scale applications," in *CGO*, 2016.

[18] T. A. Khan, A. Sriraman, J. Devietti, G. Pokam, H. Litz, and B. Kasikci, "I-spy: Context-driven conditional instruction prefetching with coalescing," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2020, pp. 146–159.

[19] T. A. Khan, I. Neal, G. Pokam, B. Mozafari, and B. Kasikci, "Dmon: Efficient detection and correction of data locality problems using selective profiling," in *Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, ser. OSDI 2021. USENIX Association, Jul. 2021.

[20] T. A. Khan, D. Zhang, A. Sriraman, J. Devietti, G. Pokam, H. Litz, and B. Kasikci, "Ripple: Profile-guided instruction cache replacement for data center applications," in *Proceedings (to appear) of the 48th International Symposium on Computer Architecture (ISCA)*, ser. ISCA 2021, Jun. 2021.

[21] T. A. Khan, N. Brown, A. Sriraman, N. K. Soundararajan, R. Kumar, J. Devietti, S. Subramoney, G. A. Pokam, H. Litz, and B. Kasikci, "Twig: Profile-guided btb prefetching for data center applications," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, 2021, pp. 816–829.

[22] D. Suggs, M. Subramony, and D. Bouvier, "The amd "zen 2" processor," *IEEE Micro*, vol. 40, no. 2, pp. 45–52, 2020.

[23] A. Pellegrini, N. Stephens, M. Bruce, Y. Ishii, J. Pusdesris, A. Raja, C. Abernathy, J. Koppanalil, T. Ringe, A. Tummala *et al.*, "The arm neoverse n1 platform: Building blocks for the next-gen cloud-to-edge infrastructure soc," *IEEE Micro*, vol. 40, no. 2, pp. 53–62, 2020.

[24] J. Rupley, "Samsung exynos m3 processor," *IEEE Hot Chips*, vol. 30, 2018.

[25] B. Grayson, J. Rupley, G. Z. Zuraski, E. Quinnell, D. A. Jiménez, T. Nakra, P. Kitchin, R. Hensley, E. Brekelbaum, V. Sinha *et al.*, "Evolution of the samsung exynos cpu microarchitecture," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2020, pp. 40–51.

[26] A. Yasin, L. Rappoport, J. W. Stark, J. Baxter, I. Diamand, P. Fridman, I. Hur, and N. Tell, "Code prefetch instruction," Nov. 4 2021, uS Patent App. 17/033,751.

[27] Google, "Propeller: Profile guided optimizing large scale llvm-based relinker," https://github.com/google/llvm-propeller, 2020.

[28] J. Mundichipparakkal, K. Nathella, and T. A. K. Khan, "Arm neoverse n1 core: Performance analysis methodology," https://armkeil.blob.core.windows.net/developer/Files/pdf/white-paper/neoverse-n1-core-performance-v2.pdf, 2021.

[29] M. Ferdman, T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos, "Temporal instruction fetch streaming," in *International Symposium on Microarchitecture*, 2008.

[30] B. Falsafi and T. F. Wenisch, "A primer on hardware prefetching," *Synthesis Lectures on Computer Architecture*, vol. 9, no. 1, pp. 1–67, 2014.

[31] R. Kumar, C.-C. Huang, B. Grot, and V. Nagarajan, "Boomerang: A metadata-free architecture for control flow delivery," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2017, pp. 493–504.

[32] R. Kumar, B. Grot, and V. Nagarajan, "Blasting through the front-end bottleneck with shotgun," *ACM SIGPLAN Notices*, vol. 53, no. 2, pp. 30–42, 2018.

[33] W. He, J. Mestre, S. Pupyrev, L. Wang, and H. Yu, "Profile inference revisited," *Proceedings of the ACM on Programming Languages*, vol. 6, no. POPL, pp. 1–24, 2022.

[34] A. Seznec, "Tage-sc-l branch predictors," in *JILP-Championship Branch Prediction*, 2014.

[35] S. Zangeneh, S. Pruett, S. Lym, and Y. N. Patt, "Branchnet: A convolutional neural network to predict hard-to-predict branches," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2020, pp. 118–130.

[36] D. A. Jiménez, H. L. Hanson, and C. Lin, "Boolean formula-based branch prediction for future technologies," in *Proceedings 2001 International Conference on Parallel Architectures and Compilation Techniques*. IEEE, 2001, pp. 97–106.

[37] "Adding processor trace support to linux," https://lwn.net/Articles/648154/.

[38] "An introduction to last branch records," https://lwn.net/Articles/680985/.

[39] Y. Zhou, X. Dong, A. L. Cox, and S. Dwarkadas, "On the impact of instruction address translation overhead," in *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2019, pp. 106–116.

[40] R. Lavaee, J. Criswell, and C. Ding, "Codestitcher: inter-procedural basic block layout optimization," in *Proceedings of the 28th International Conference on Compiler Construction*, 2019, pp. 65–75.

[41] G. Vavouliotis, L. Alvarez, B. Grot, D. Jiménez, and M. Casas, "Morrigan: A composite instruction tlb prefetcher," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, 2021, pp. 1138–1153.

[42] A. A. Moreira, G. Ottoni, and F. M. Quintão Pereira, "Vespa: static profiling for binary optimization," *Proceedings of the ACM on Programming Languages*, vol. 5, no. OOPSLA, pp. 1–28, 2021.

[43] "Mysql," [Online; accessed 19-Nov-2021]. [Online]. Available: https://www.mysql.com

[44] T. P. P. Council, "Tpc-c," [Online; accessed 19-Nov-2021]. [Online]. Available: http://www.tpc.org/tpcc/

[45] "Postgresql: The world's most advanced open source database," [Online; accessed 19-Nov-2021]. [Online]. Available: https://www.postgresql.org/

[46] "Postgresql: Documentation: 14: pgbench," [Online; accessed 19-Nov-2021]. [Online]. Available: https://www.postgresql.org/docs/current/pgbench.html

[47] "Clang c language family frontend for llvm," [Online; accessed 19-Nov-2021]. [Online]. Available: https://clang.llvm.org/

[48] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* IEEE, 2004, pp. 75–86.

[49] "Welcome to python.org," [Online; accessed 19-Nov-2021]. [Online]. Available: https://www.python.org/

[50] "The python performance benchmark suite," [Online; accessed 19-Nov-2021]. [Online]. Available: https://pyperformance.readthedocs.io/

[51] "Twitter finagle," https://twitter.github.io/finagle/.

[52] A. Prokopec, A. Rosà, D. Leopoldseder, G. Duboscq, P. Tůma, M. Studener, L. Bulej, Y. Zheng, A. Villazón, D. Simon, T. Würthinger, and W. Binder, "Renaissance: Benchmarking suite for parallel applications on the jvm," in *Programming Language Design and Implementation*, 2019.

[53] "Apache cassandra," http://cassandra.apache.org/.

[54] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer *et al.*, "The dacapo benchmarks: Java benchmarking development and analysis," in *Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, 2006, pp. 169–190.

[55] "Apache kafka," https://kafka.apache.org/powered-by.

[56] "Apache tomcat," https://tomcat.apache.org/.

[57] "Drupal," https://www.drupal.org, [Online; accessed 30-June-2022].

[58] "facebookarchive/oss-performance: Scripts for benchmarking various php implementations when running open source software," https://github.com/facebookarchive/oss-performance, 2019, (Online; last accessed 15-November-2019).

[59] "Wordpress," https://wordpress.org, [Online; accessed 30-June-2022].

[60] "Mediawiki," https://www.mediawiki.org/wiki/MediaWiki, [Online; accessed 30-June-2022].

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

[62] "Championship branch prediction," https://jilp.org/cbp2016/, 2016.

[63] C. Young, N. Gloy, and M. D. Smith, "A comparative analysis of schemes for correlated branch prediction," *ACM SIGARCH Computer Architecture News*, vol. 23, no. 2, pp. 276–286, 1995.

[64] Lee and Smith, "Branch prediction strategies and branch target buffer design," *Computer*, vol. 17, no. 1, pp. 6–22, 1984.

[65] S. McFarling, "Combining branch predictors," Citeseer, Tech. Rep., 1993.

[66] R. Nair, "Dynamic path-based branch correlation," in *Proceedings of the 28th annual international symposium on Microarchitecture*. IEEE, 1995, pp. 15–23.

[67] 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*, 1992, pp. 76–84.

[68] J. E. Smith, "A study of branch prediction strategies," in *25 years of the international symposia on Computer architecture (selected papers)*, 1998, pp. 202–215.

[69] T.-Y. Yeh and Y. N. Patt, "Two-level adaptive training branch prediction," in *Proceedings of the 24th annual international symposium on Microarchitecture*, 1991, pp. 51–61.

[70] M. D. Hill and A. J. Smith, "Evaluating associativity in cpu caches," *IEEE Transactions on Computers*, vol. 38, no. 12, pp. 1612–1630, 1989.

[71] R. A. Sugumar and S. G. Abraham, "Efficient simulation of caches under optimal replacement with applications to miss characterization," in *Proceedings of the 1993 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, 1993, pp. 24–35.

[72] P. Michaud, A. Seznec, and R. Uhlig, "Trading conflict and capacity aliasing in conditional branch predictors," in *Proceedings of the 24th annual international symposium on Computer architecture*, 1997, pp. 292–303.

[73] A. Jaleel, K. B. Theobald, S. C. Steely Jr, and J. Emer, "High performance cache replacement using re-reference interval prediction (rrip)," *ACM SIGARCH Computer Architecture News*, vol. 38, no. 3, pp. 60–71, 2010.

[74] C. Ding and Y. Zhong, "Predicting whole-program locality through reuse distance analysis," in *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, 2003, pp. 245–257.

[75] M. U. Farooq, L. K. John *et al.*, "Store-load-branch (slb) predictor: A compiler assisted branch prediction for data dependent branches," in *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2013, pp. 59–70.

[76] N. Adiga, J. Bonanno, A. Collura, M. Heizmann, B. R. Prasky, and A. Saporito, "The ibm z15 high frequency mainframe branch predictor industrial product," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2020, pp. 27–39.

[77] A. Seznec, "Analysis of the o-geometric history length branch predictor," in *32nd International Symposium on Computer Architecture (ISCA'05)*. IEEE, 2005, pp. 394–405.

[78] R. A. Fisher and F. Yates, *Statistical tables for biological, agricultural and medical research*. Hafner Publishing Company, 1953.

[79] R. Durstenfeld, "Algorithm 235: random permutation," *Communications of the ACM*, vol. 7, no. 7, p. 420, 1964.

[80] W. Cui, X. Ge, B. Kasikci, B. Niu, U. Sharma, R. Wang, and I. Yun, "{REPT}: Reverse debugging of failures in deployed software," in *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, 2018, pp. 17–32.

[81] W. Erquinigo, D. Carrillo-Cisneros, and A. Tang, "Reverse debugging at scale," https://engineering.fb.com/2021/04/27/developer-tools/reverse-debugging/.

[82] B. Kasikci, W. Cui, X. Ge, and B. Niu, "Lazy diagnosis of in-production concurrency bugs," in *Proceedings of the 26th Symposium on Operating Systems Principles*, 2017, pp. 582–598.

[83] B. Kasikci, C. Pereira, G. Pokam, B. Schubert, M. Musuvathi, and G. Candea, "Failure sketches: A better way to debug," ser. Hot Topics in Operating Systems, 2015, p. 5.

[84] B. Kasikci, B. Schubert, C. Pereira, G. Pokam, and G. Candea, "Failure sketching: A technique for automated root cause diagnosis of in-production failures," in *Proceedings of the 25th Symposium on Operating Systems Principles*, 2015, p. 344–360.

[85] G. Zuo, J. Ma, A. Quinn, P. Bhatotia, P. Fonseca, and B. Kasikci, "Execution reconstruction: Harnessing failure reoccurrences for failure reproduction," in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, 2021, p. 1155–1170.

[86] S. Jamilan, T. A. Khan, G. Ayers, B. Kasikci, and H. Litz, "Apt-get: Profile-guided timely software prefetching," in *Proceedings of the 17th European Conference on Computer Systems (EuroSys)*, ser. EuroSys 2022, Apr. 2022.

[87] N. K. Soundararajan, P. Braun, T. A. Khan, B. Kasikci, H. Litz, and S. Subramoney, "Pdede: Partitioned, deduplicated, delta branch target buffer," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, 2021, pp. 779–791.

[88] I. Corparation, "Intel (r) 64 and ia-32 architectures software developer's manual," *Combined Volumes, Dec*, 2016.

[89] A. Seznec, "Exploring branch predictability limits with the mtage+ sc predictor," in *5th JILP Workshop on Computer Architecture Competitions (JWAC-5): Championship Branch Prediction (CBP-5)*, 2016, p. 4.

[90] T. A. Khan, Y. Zhao, G. Pokam, B. Mozafari, and B. Kasikci, "Huron: hybrid false sharing detection and repair," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2019, pp. 453–468.

[91] H. Litz, G. Ayers, and P. Ranganathan, "Crisp: critical slice prefetching," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2022, pp. 300–313.

[92] C.-K. Luk and T. C. Mowry, "Cooperative prefetching: Compiler and hardware support for effective instruction prefetching in modern processors," in *International Symposium on Microarchitecture*, 1998.

[93] S. Harizopoulos and A. Ailamaki, "Steps towards cache-resident transaction processing," in *International conference on Very large data bases*, 2004.

[94] J. Zhou and K. A. Ross, "Buffering databse operations for enhanced instruction cache performance," in *International conference on Management of data*, 2004.

[95] L. L. Peterson, "Architectural and compiler support for effective instruction prefetching: a cooperative approach," *ACM Transactions on Computer Systems*, 2001.

[96] D. X. Li, R. Ashok, and R. Hundt, "Lightweight feedback-directed cross-module optimization," in *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, 2010, pp. 53–61.

[97] C.-K. Luk, R. Muth, H. Patil, R. Cohn, and G. Lowney, "Ispike: a post-link optimizer for the intel/spl reg/itanium/spl reg/architecture," in *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* IEEE, 2004, pp. 15–26.

[98] Y. Zhang, T. A. Khan, G. Pokam, B. Kasikci, H. Litz, and J. Devietti, "Ocolos: Online code layout optimizations," in *Proceedings of the 55th International Symposium on Microarchitecture (MICRO)*, Oct. 2022.

[99] S. Song, T. A. Khan, S. M. Shahri, A. Sriraman, N. K. Soundararajan, S. Subramoney, D. A. Jiménez, H. Litz, and B. Kasikci, "Thermometer: profile-guided btb replacement for data center applications," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, 2022, pp. 742–756.

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

[101] D. A. Jiménez and C. Lin, "Dynamic branch prediction with perceptrons," in *Proceedings of the Seventh International Symposium on High-Performance Computer Architecture (HPCA'01), Nuevo Leone, Mexico, January 20-24, 2001*. IEEE Computer Society, 2001, pp. 197–206. [Online]. Available: https://doi.org/10.1109/HPCA.2001.903263

[102] A. Seznec, "TAGE-SC-L Branch Predictors Again," in *5th JILP Workshop on Computer Architecture Competitions (JWAC-5): Championship Branch Prediction (CBP-5)*, Seoul, South Korea, Jun. 2016.

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

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

[105] S. Pruett and Y. Patt, "Branch runahead: An alternative to branch prediction for impossible to predict branches," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 804–815. [Online]. Available: https://doi.org/10.1145/3466752.3480053

[106] S. Gupta, N. Soundararajan, R. Natarajan, and S. Subramoney, "Opportunistic early pipeline re-steering for data-dependent branches," in *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 305–316. [Online]. Available: https://doi.org/10.1145/3410463.3414628

[107] A. Samara and J. Tuck, "The case for domain-specialized branch predictors for graph-processing," *IEEE Computer Architecture Letters*,

[114] B. Calder, D. Grunwald, M. Jones, D. Lindsay, J. Martin, M. Mozer, and B. Zorn, "Evidence-based static branch prediction using machine learning," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 19, no. 1, pp. 188–222, 1997.

vol. 19, no. 2, pp. 101–104, 2020.

[108] A. Sridhar, N. Kabylkas, and J. Renau, "Load driven branch predictor (LDBP)," *CoRR*, vol. abs/2009.09064, 2020. [Online]. Available: https://arxiv.org/abs/2009.09064

[109] J. A. Fisher and S. M. Freudenberger, "Predicting conditional branch directions from previous runs of a program," *ACM SIGPLAN Notices*, vol. 27, no. 9, pp. 85–95, 1992.

[110] T. Ball and J. R. Larus, "Branch prediction for free," *ACM SIGPLAN Notices*, vol. 28, no. 6, pp. 300–313, 1993.

[111] Y. Wu and J. R. Larus, "Static branch frequency and program profile analysis," in *Proceedings of the 27th annual international symposium on Microarchitecture*, 1994, pp. 1–11.

[112] C. Young and M. D. Smith, "Improving the accuracy of static branch prediction using branch correlation," *ACM SIGOPS Operating Systems Review*, vol. 28, no. 5, pp. 232–241, 1994.

[113] A. Krall, "Improving semi-static branch prediction by code replication," *ACM SIGPLAN Notices*, vol. 29, no. 6, pp. 97–106, 1994.

[115] J. R. Patterson, "Accurate static branch prediction by value range propagation," in *Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*, 1995, pp. 67–78.

[116] D. A. Jiménez and C. Lin, "Branch path re-aliasing," in *Proceedings of the 4th Workshop on Feedback Directed and Dynamic Optimization (FDDO-4)*. Citeseer, 2001.

[117] D. A. Jiménez, "Code placement for improving dynamic branch prediction accuracy," *ACM SIGPLAN Notices*, vol. 40, no. 6, pp. 107–116, 2005.

[118] T. Sherwood and B. Calder, "Automated design of finite state machine predictors for customized processors," in *Proceedings 28th Annual International Symposium on Computer Architecture*. IEEE, 2001, pp. 86–97.

[119] M.-D. Tarlescu, K. B. Theobald, and G. R. Gao, "Elastic history buffer: A low-cost method to improve branch prediction accuracy," in *Proceedings International Conference on Computer Design VLSI in Computers and Processors*. IEEE, 1997, pp. 82–87.

[120] J. Stark, M. Evers, and Y. N. Patt, "Variable length path branch prediction," in *Proceedings of the eighth international conference on Architectural support for programming languages and operating systems*, 1998, pp. 170–179.

[121] A. Adileh, D. J. Lilja, and L. Eeckhout, "Architectural support for probabilistic branches," in *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-51. IEEE Press, 2018, p. 108–120. [Online]. Available: https://doi.org/10.1109/MICRO.2018.00018

[122] S. Verma, B. Maderazo, and D. M. Koppelman, "Spotlight-a low complexity highly accurate profile-based branch predictor," in *2009 IEEE 28th International Performance Computing and Communications Conference*. IEEE, 2009, pp. 239–247.

[123] V. Desmet, L. Eeckhout, and K. D. Bosschere, "Using decision trees to improve program-based and profile-based static branch prediction," in *Asia-Pacific Conference on Advances in Computer Systems Architecture*. Springer, 2005, pp. 336–352.

[124] Y. Mao, J. Shen, and X. Gui, "A study on deep belief net for branch prediction," *IEEE Access*, vol. 6, pp. 10 779–10 786, 2017.

[125] S. J. Tarsa, C.-K. Lin, G. Keskin, G. Chinya, and H. Wang, "Improving branch prediction by modeling global history with convolutional neural networks," *arXiv preprint arXiv:1906.09889*, 2019.