Thermometer: Profile-Guided BTB Replacement for Data Center Applications

Shixin Song
University of Michigan
USA
shixins@umich.edu

Tanvir Ahmed Khan
University of Michigan
USA
takh@umich.edu

Sara Mahdizadeh Shahri
University of Michigan
USA
smahdiz@umich.edu

Akshitha Sriraman
Carnegie Mellon University
USA
akshitha@cmu.edu

Niranjan K Soundararajan
Intel Corporation
India
niranjan.k.soundararajan@intel.com

Sreenivas Subramoney
Intel Corporation
India
sreenivas.subramoney@intel.com

Daniel A. Jiménez
Texas A&M University
USA
djimenez@acm.org

Heiner Litz
University of California, Santa Cruz
USA
hlitz@ucsc.edu

Baris Kasikci
University of Michigan
USA
barisk@umich.edu

ABSTRACT

Modern processors employ a decoupled frontend with Fetch Directed Instruction Prefetching (FDIP) to avoid frontend stalls in data center applications. However, the large branch footprint of data center applications precipitates frequent Branch Target Buffer (BTB) misses that prohibit FDIP from eliminating more than 40% of all frontend stalls. We find that the state-of-the-art BTB optimization techniques (*e.g.*, BTB prefetching and replacement mechanisms) cannot eliminate these misses due to their inadequate understanding of branch reuse behavior in data center applications.

In this paper, we first perform a comprehensive characterization of the branch behavior of data center applications, and determine that identifying optimal BTB replacement decisions requires considering both transient and holistic (*i.e.*, across the entire execution) branch behavior. We then present *Thermometer*, a novel BTB replacement technique that realizes the holistic branch behavior via a profile-guided analysis. Based on the collected profile, *Thermometer* generates useful BTB replacement hints that the underlying hardware can leverage. We evaluate *Thermometer* using 13 widely-used data center applications and demonstrate that it provides an average speedup of 8.7% (0.4%-64.9%) while outperforming the state-of-the-art BTB replacement techniques by 5.6× (on average, the best performing prior work achieves 1.5% speedup). We also demonstrate that *Thermometer* achieves a performance speedup that is, on average, 83.6% of the speedup achieved by the optimal BTB replacement policy.

ISCA '22, June 18–22, 2022, New York, NY, USA

ACM ISBN 978-1-4503-8610-4/22/06…$15.00
https://doi.org/10.1145/3470496.3527430

CCS CONCEPTS

• **Computer systems organization** → *Pipeline computing*.

KEYWORDS

Cache replacement, frontend stalls, branch target buffer, data center

ACM Reference Format:
Shixin Song, Tanvir Ahmed Khan, Sara Mahdizadeh Shahri, Akshitha Sriraman, Niranjan K Soundararajan, Sreenivas Subramoney, Daniel A. Jiménez, Heiner Litz, and Baris Kasikci. 2022. Thermometer: Profile-Guided BTB Replacement for Data Center Applications. In *The 49th Annual International Symposium on Computer Architecture (ISCA '22), June 18–22, 2022, New York, NY, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3470496.3527430

1 INTRODUCTION

Large instruction footprints exhibited by modern data center applications induce significant stalls in the frontend of the processor pipeline, introducing performance losses worth millions of dollars [25, 27, 42, 67, 104, 107, 133]. Modern data center applications exhibit multi-megabyte code footprints [27, 67, 106, 107] due to their complex application logic [104] and frequent use of different libraries [67], language runtimes [19, 103], and kernel modules [27]. Data center applications' large code footprints do not fit in the processor's instruction cache (I-cache) [25]. As a result, the processor fails to fetch sufficient instructions, leading to frequent frontend stalls. Since even single-digit performance gains in data center applications can minimize the Total Cost of Ownership (TCO) [27, 67] and reduce data center carbon emissions [133], there is a critical need to mitigate frontend stalls to improve data center efficiency.

Prior works have proposed numerous techniques to mitigate frontend stalls including compiler-based Profile-Guided Optimizations (PGO) [33, 47, 106, 107, 113] and hardware-based instruction prefetchers [43, 44, 72, 73, 82–84, 108, 119, 121, 131]. On the software side, PGO techniques improve instruction locality by putting frequently executed I-cache lines together. Though, in theory, these

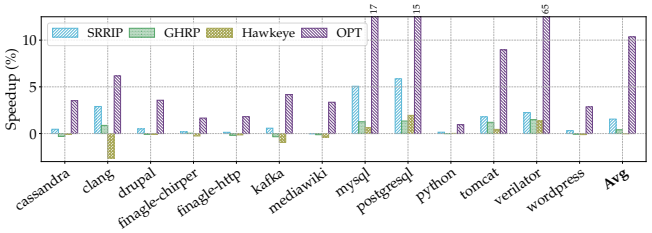

Figure 1: Speedup for state-of-the-art BTB replacement policies [20, 60, 62] over an LRU baseline: existing mechanisms provide an average speedup of 1.5%, although an optimal replacement policy [29] provides an average speedup of 10.4%. Hence, there is a significant performance gap between the state-of-the-art and the optimal replacement policy.

code layout optimization techniques are sensitive to profile quality [55], they work exceptionally well in practice [27, 33, 47, 100, 106, 107]. Profiles for data center applications change slowly over several weeks [33] while data center operators profile and recompile applications multiple times a day [21, 33, 106, 107]. Consequently, these automated techniques have ample opportunity to adapt to changing application profiles and are widely used in today's data centers [27, 33, 47, 106, 107]. For example, half of all CPU cycles in Google data centers are spent in PGO-optimized binaries [33]. Therefore, we leverage PGO techniques' effectiveness in this work.

Among hardware techniques, Fetch Directed Instruction Prefetching (FDIP) [118, 119] is an effective technique employed by modern processors [49, 109, 123, 135] to reduce frontend stalls. FDIP decouples the branch prediction unit from the instruction fetch unit so that the frontend can run ahead, producing the instruction addresses likely to be executed in the near future. Prefetching I-cache lines corresponding to these future accesses avoids potential frontend stalls [83, 84], providing performance similar to aggressive I-cache prefetchers [57, 58].

However, FDIP performs well only as long as the Branch Target Buffer (BTB) supplies correct targets for all taken branches [23, 44, 75, 83, 84, 132]. Prior works have found that FDIP's performance is significantly limited by BTB misses that stall FDIP's prefetching [23, 75, 83, 84] or cause FDIP to prefetch incorrect instructions on the wrong path [44, 132]. As we and others [23, 75, 83, 84, 132] show, this limitation inhibits FDIP from eliminating more than 40% of all frontend stalls in data center applications.

To this end, we thoroughly analyze the BTB access behavior of modern data center applications that limit FDIP's effectiveness. We find that data center applications exhibit a unique branch reuse behavior that is difficult to capture, causing wasteful BTB evictions. As a result, existing BTB prefetching mechanisms [73, 83] fall short as they bring in unused branch entries into the BTB, failing to avoid the majority of frontend stalls. Since avoiding wasteful evictions is the main responsibility of an effective BTB replacement policy, we evaluate state-of-the-art replacement policies (GHRP [20], Hawkeye [60], and SRRIP [62]) in the context of data center applications' BTB access patterns. As shown in Fig. 1, these policies provide a negligible speedup (1.5% on average) over the Least Recently Used (LRU [96]) replacement policy. In contrast, an optimal BTB replacement policy provides 10.4% average speedup over LRU.

The key takeaway from our characterization is that existing replacement policies do not account for the diversity of BTB access patterns among different executions of the same branch, inhibiting them from predicting and evicting the branch that is taken furthest in the future.

We quantify the diversity of BTB access patterns using reuse distance [37, 62] and introduce the concept of *transient* and *holistic* reuse distance. The transient reuse distance is the most recent reuse distance that the BTB entry experiences. The holistic reuse distance of a BTB entry is the average reuse distance for all instances of a branch across the entire execution. For data center applications, we show that the transient reuse distance varies significantly (more than 2×) from the holistic reuse distance. Consequently, we observe that replacing BTB entries based on a holistic pattern is more beneficial than replacement decisions made using a transient pattern used by prior work [20, 60, 62].

To classify branches using their holistic pattern, we introduce a metric, "branch temperature" based on the *hit-to-taken percentage* of a branch under the optimal BTB replacement policy, which measures the benefit (*i.e.*, the number of BTB hits) per given execution of a branch (*i.e.*, the number of times the branch is taken). We find that a branch's *hit-to-taken percentage* captures the holistic pattern of that branch as 'hot' branches with a high *hit-to-taken percentage* result in more hits and are more valuable to keep in the BTB than 'cold' branches with a low *hit-to-taken percentage*.

Driven by our characterization's insights, we propose *Thermometer*, a novel BTB replacement technique that accommodates both holistic and transient patterns of branches in data center applications. *Thermometer* calculates the holistic pattern via an offline profile-guided analysis. *Thermometer* performs this analysis on a trace of executed branch instructions collected via efficient hardware support (*e.g.*, Intel PT [1]). Based on this profile-guided analysis, *Thermometer* tags each branch with a hint defining its holistic pattern. Finally, *Thermometer* introduces a small hardware enhancement to the BTB replacement policy to enable eviction decisions based on both the injected hint and the transient pattern.

We evaluate *Thermometer* for (1) 13 widely-used data center applications that experience frequent frontend stalls, (2) 663 industry traces from 5^{th} Championship Branch Prediction (CBP-5) [15], and (3) 50 traces from 1^{st} Instruction Prefetching Championship (IPC-1) [17]. Across all applications, *Thermometer* achieves an average IPC speedup of 8.7% (0.4%-64.9%) by avoiding 21.3% of all BTB misses. In comparison, the best performing prior work [62] provides an average IPC speedup of 1.5% and covers 6.7% of all BTB misses. Consequently, *Thermometer* achieves 5.6× greater speedup by eliminating 3.2× additional misses compared to the state-of-the-art BTB replacement techniques [20, 60, 62]. Across 663 CBP-5 traces (that do not allow generating IPC numbers [20]), *Thermometer* provides an average BTB miss reduction of 2.25% over the best performing prior work [20]. Across 50 IPC-1 traces, *Thermometer* achieves an average IPC speedup of 1.07% compared to 0.45% mean speedup provided by the best performing prior work [62]. Overall, *Thermometer* achieves a performance speedup that is, on average, 83.6% of the speedup offered by the optimal BTB replacement policy.

In summary, we contribute:

- A comprehensive characterization of the branch behavior of data center applications that shows that considering both holistic and

Table 1: Simulation parameters

Parameter	Value
CPU	6-wide, 24-entry (192-instruction) FTQ, 60-entry Decode Queue, 352-entry Re-order Buffer, 128-entry Reservation Station
Branch prediction units	8192-entry 4-way BTB, 4096-entry IBTB, 32-entry RAS, 64KB TAGE-SC-L [126]
Caches	64B block: 32KB, 8-way L1I, 48KB, 12-way L1D, 512KB 8-way L2C, 2MB 16-way LLC

transient access patterns is critical to achieve near-ideal frontend performance.

- *Thermometer*: A novel profile-guided BTB replacement mechanism that identifies holistic branch patterns offline and considers both holistic and transient patterns online to make close-to-optimal BTB replacement decisions.
- An extensive evaluation of *Thermometer* in the context of frontend-bound data center applications, demonstrating *Thermometer*'s potential to avoid costly BTB misses and achieve significant performance improvements.

2 UNDERSTANDING THE CHALLENGES OF BTB REPLACEMENT

In this section, we analyze the frontend performance of 13 data center applications. We find that performance, to a large degree, is determined by the BTB's hit rate, which in turn is limited by the efficacy of the BTB replacement policy. We show that existing replacement policies exhibit a large performance gap compared to an optimal policy. We also provide insights to close this performance gap by 83.6%.

2.1 Experimental methodology

Simulation parameters. We simulate and evaluate *Thermometer* using the ChampSim [5] simulator and adjust simulation parameters to resemble a recent state-of-the-art industry FDIP baseline [57, 58], as listed in Table 1. We implement the optimal BTB replacement policy (Belady's algorithm [29, 61]) and other existing policies including SRRIP [62], GHRP [20], and Hawkeye [60] to compare them with *Thermometer*.

Data center applications. Prior work from Google and Facebook shows that their widely-deployed data center applications lose more than 15% of all pipeline slots due to frontend stalls [25, 27, 67, 133]. As these applications are proprietary, we use the applications used by prior work [75, 77, 78, 86, 100, 138, 150], where frontend stalls are similarly frequent (more than 15%) due to large instruction footprints. These applications include `cassandra` [2], `kafka` [3], and `tomcat` [4] from the Java DaCapo benchmark suite [31], `drupal` [142], `wordpress` [144], and `mediawiki` [143] from Facebook's OSS - performance benchmark suite [16], `finagle-chirper` and `finagle-http` [12] from the Java Renaissance benchmark suite [114], `clang` [6] while building `LLVM` [85], `PostgreSQL` [10] while serving `pgbench` [9] queries, `Python` [14] while running the `pyperformance` [11] benchmark suite, `MySQL` [146] while serving `TPC-C` queries [35], and `verilator` [13] while emulating the `Rocket Chip` [7].

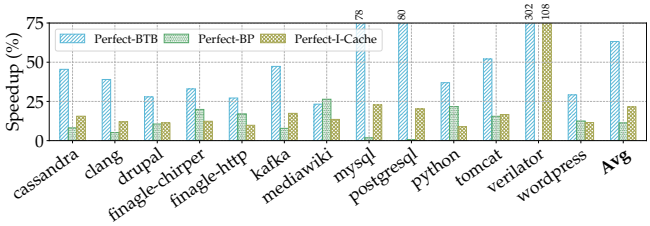

Figure 2: Limit study of FDIP: a perfect BTB provides 63.2% average speedup that is significantly more than average speedups provided by a perfect I-cache (21.5%) and a perfect branch direction predictor (11.3%).

We use traces collected via Intel PT [1] and modify ChampSim to simulate these traces. Using these traces, we characterize BTB replacement challenges to design *Thermometer*, a novel profile-guided BTB replacement technique. We validate *Thermometer*'s effectiveness on 13 data center applications and on CBP-5 [15] and IPC-1 [17] traces that prior work [20, 24, 57, 58] evaluate their frontend optimizations.

2.2 Why is the BTB replacement policy important?

To establish the importance of the BTB performance in modern Out-of-Order (OoO) cores, we perform limit studies of different frontend structures determining their individual impact on performance. In Fig. 2, we measure the Instructions Per Cycle (IPC) speedup achieved by a perfect BTB that faces no misses (*i.e.*, every BTB access is a hit), a perfect branch predictor that always predicts taken and not taken branches correctly, and a perfect I-cache with no misses. On average, a perfect BTB achieves 63.2% speedup. In contrast, perfect branch direction prediction achieves merely 11.3% speedup and a perfect I-cache achieves only 21.5% speedup. These results indicate that with a perfect BTB, FDIP can provide more benefits than with a perfect I-cache or a perfect branch predictor. Hence, optimizing BTB performance is critical to eliminate frontend stalls in the most efficient manner (as also reported by prior work [75, 83, 84, 132]).

As shown in Fig. 2, the perfect BTB and perfect I-cache provides significantly greater speedup for `verilator` than any other applications. As shown in Fig. 3, this is because `verilator` exhibits at least 300× greater L2 cache level instructions Misses Per Kilo Instructions (L2iMPKI) than any other applications in the study. Recent works from data center providers [25, 133] observe that their workloads' L2iMPKIs range from 10-40, which are considerably greater than L2iMPKIs of all 12 other applications and closer to `verilator`'s L2iMPKI (42). Therefore, we study `verilator`'s behavior as a proxy [53] for real world data center applications.

We next investigate if the performance gap between a practical BTB and a perfect BTB can be closed by existing BTB optimization techniques. Prior work such as Confluence [73] and Shotgun [83] use BTB prefetching to reduce BTB misses and improve FDIP performance. In Fig. 4, we compare the IPC speedups of these prior techniques against a perfect BTB's speedup. We assume that the baseline BTB does not have any prefetching and uses the LRU replacement policy.

As also observed by recent work [75, 132], we find in Fig. 4 that Confluence [73] achieves merely 1.4% average speedup while

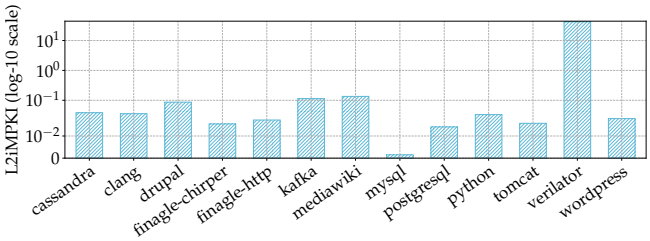

Figure 3: L2 cache level instructions Misses Per Kilo Instructions (L2iMPKI): `verilator` suffers from at least 300× greater L2iMPKI compared to other applications in our study. The y-axis is log-10 scale.

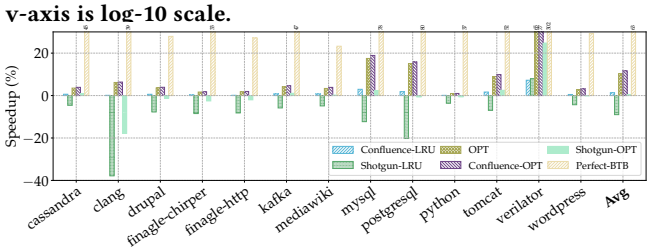

Figure 4: Speedup for different BTB configurations over a baseline BTB using the LRU replacement policy without prefetching: existing BTB prefetching mechanisms [73, 83] provide merely 1.4% mean speedup, while a perfect BTB offers 63.2% mean speedup. The optimal BTB replacement policy [29, 61] significantly reduces this performance gap by providing 10.4% mean speedup.

Shotgun [83] faces a slight slowdown as it wastes critical BTB capacity to store unused prefetch metadata. We corroborate the findings [23, 75, 132] of recent work and identify three reasons behind the performance degradation induced by these prior BTB prefetching policies [73, 83]. First, like any temporal prefetcher [139–141], Confluence and Shotgun cannot avoid new and non-recurring temporal streams of BTB misses which constitute almost half (48%) of all BTB misses [75]. Second, Shotgun statically partitions the BTB according to the branch type (*e.g.*, conditional, unconditional). However, Shotgun's static partitioning does not necessarily match the working set sizes of conditional and unconditional branches for data center applications [23, 75]. Third, Shotgun allocates valuable on-chip storage for not-taken branches [132], resulting in 26 − 45% of all conditional branches not fitting in the BTB [75]. Hence, both these prior BTB prefetching techniques significantly fall short of the 63.2% average speedup achieved by a perfect BTB.

In Fig. 4, we also show the speedup achieved by a BTB with an optimal replacement policy (Belady's algorithm [29, 61]). This provably optimal yet impractical replacement policy evicts the BTB entry that will be used furthest in the future [30, 96, 98]. Such a BTB makes optimal replacement decisions with perfect future knowledge and achieves an average IPC speedup of 10.4%. Moreover, the optimal BTB replacement policy improves the performance of these prior BTB prefetching techniques by avoiding prefetch-induced wasteful evictions. These results demonstrate that a BTB replacement policy that is more optimized than prior work [73, 83] can better close the performance gap between a baseline and a perfect BTB.

Observation: An optimal BTB replacement policy enables FDIP to achieve near-ideal performance.
Insight: *An efficient BTB replacement policy is crucial to improve FDIP's performance.*

2.3 Why do prior replacement policies fall short?

In §2.2, we showed that the optimal BTB replacement policy achieves 10.4% average speedup. Now, we investigate whether existing replacement policies can provide similar speedup. To our knowledge, GHRP [20] is the only replacement policy designed for the BTB. To expand the scope of our analysis, we also adapt existing data cache replacement policies such as Hawkeye [60] and SRRIP [62] to BTB.

GHRP [20] predicts dead BTB entries (entries that do not experience hits until eviction [97]) using the global control flow history. To make a replacement decision, GHRP evicts the BTB entry that is most likely to be dead based on the prediction results.

Hawkeye [60] simulates the optimal replacement policy [29] on an access history to determine if a given branch instruction is "BTB-friendly" or "BTB-averse", *i.e.*, whether storing the branch information in the BTB results in a hit or a miss. When making a replacement decision, Hawkeye favors keeping BTB-friendly entries in the BTB and evicting BTB-averse entries.

SRRIP [62] assumes that all newly-executed branch instructions are BTB-averse. SRRIP only marks a branch as BTB-friendly when the branch is executed again after it has been inserted into the BTB. When making replacement decisions, SRRIP prefers to evict BTB-averse entries.

Fig. 1 shows the speedup for different BTB replacement policies over the LRU baseline. As shown, none of the 13 applications we study significantly benefit from these existing replacement policies. Specifically, the state-of-the-art BTB replacement policy, GHRP does not perform well for applications with large working sets [78].

As purely hardware techniques, GHRP, Hawkeye, and SRRIP have no information about branches currently not in the BTB. Moreover, they lose all the information about a branch every time the corresponding BTB entry is evicted. Since large working set sizes (both instruction and branch footprint) are the key characteristics of data center applications [27, 67, 75, 77, 78, 132], it is necessary to retain branch reuse behavior even when the corresponding entry is not present in the BTB.

Among existing policies, only SRRIP provides a speedup (1.5% on average, up to 5.9%) for these data center applications. Still, SRRIP falls short of the optimal BTB replacement policy which offers an average IPC speedup of 10.4%.

To understand why existing BTB replacement policies perform poorly for data center applications, we introduce the concept of *transient* and *holistic* reuse distances. For a given BTB entry X, reuse distance [37, 62] is the number of unique BTB entries accessed between two consecutive accesses to X (within the associative set to which X belongs). The transient reuse distance refers to the reuse distance between the last two references of a BTB entry (*e.g.*, the LRU replacement policy considers the transient reuse distance of accesses). The holistic reuse distance is the average reuse distance for all instances of a branch across the entire execution of a program.

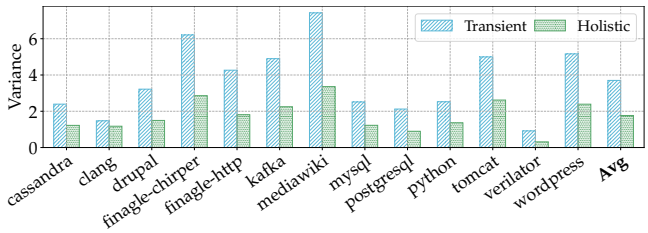

Figure 5: Average transient and holistic reuse distance variance for data center applications: the transient variance is significantly larger (more than 2×) than the holistic variance.

We find that existing BTB replacement policies perform poorly for data center applications because they only consider the transient reuse distance. For data center applications, this transient reuse distance significantly differs from the holistic reuse distance as we observe that the reuse distance for a given branch instruction varies widely during the program execution.

To quantify the variance of branch instructions' reuse distances, we define the reuse distance vector a_i of a certain branch a, where i represents the i^{th} execution of that branch for $i = 2, 3 \ldots, n$. Prior techniques [20, 60, 62] perform BTB replacement decisions based on a branch's transient (most recent) reuse distance and hence, they experience *transient* variance defined as follows:

$$\text{Transient variance} = \frac{1}{n-2}\sum_{i=2}^{n-1}(a_i - a_{i+1})^2$$

Instead, we recommend performing BTB replacement decisions based on the holistic (average) reuse distance, $\overline{a}$, which experience *holistic* variance defined as follows:

$$\text{Holistic variance:} = \frac{1}{n-1}\sum_{i=2}^{n}(a_i - \overline{a})^2.$$

In Fig. 5, we show the average transient and holistic variance for all 13 data center applications. As shown, transient variance for data center applications is significantly greater than the holistic variance. Consequently, replacement decisions made based on the transient reuse distance are less likely to be accurate as they suffer from higher variance than replacement decisions made using the holistic reuse distance.

Qualitatively, holistic reuse distance is more accurate than transient reuse distance as holistic reuse distance is computed using reuse distance samples from the entire execution. On the other hand, transient reuse distance is computed using samples from a short execution fragment. Consequently, holistic reuse distance is more accurate and representative of the broad dynamic behavior of a program. Moreover, data center applications' dynamic behavior shows increasing variation due to growing software complexity [67, 103, 104, 107], making holistic reuse distance more useful.

Our observation also explains why prior replacement policies fall significantly short of the optimal replacement policy, as shown earlier in this subsection. The optimal BTB replacement policy makes replacement decisions using more holistic, future knowledge. Consequently, the optimal BTB replacement policy can compute a perfect reuse distance, making it more accurate than a policy using the transient reuse distance.

Observation: Existing replacement policies fall short when applied to BTB.
Insight: *It is necessary to analyze holistic branch execution patterns to design a new replacement policy.*

2.4 How do we redesign BTB replacement?

As shown in §2.3, existing replacement policies suffer from a high transient variance due to their limited knowledge of the behavior of branches over time, and hence, perform poorly for data center applications. Since the optimal BTB replacement policy allows determining the perfect reuse distance for a given branch based on its future knowledge, we analyze the optimal BTB replacement policy to determine the holistic behavior of branches.

Our goal is to capture the relative benefit of caching a BTB entry. To do this, we define and compute a normalized metric called *hit-to-taken percentage*, which measures the benefit (*i.e.*, the number of BTB hits) per given execution of a branch instruction (*i.e.*, the number of times the branch is taken).

Fig. 6 shows the distribution of *hit-to-taken percentage* for the optimal BTB replacement policy on several data center applications' execution traces. Due to space constraints, we only portray the behavior of three data center applications, `drupal`, `kafka`, and `verilator`; the remaining applications exhibit similar behaviors to `drupal` and `kafka`.

In Fig. 6, the X-axis represents the percentage of unique taken branches and the Y-axis represents the corresponding *hit-to-taken percentage* for the optimal BTB replacement policy. The *hit-to-taken percentage* indicates which branches would result in more hits (relative to how many times the branch is taken) and are hence more valuable to retain in the BTB. As shown in Fig. 6, all branches from these applications can be categorized to three different types based on their *hit-to-taken percentage*. We mark branches with the highest *hit-to-taken percentage* as "hot" branches (marked by the red region), branches with the lowest *hit-to-taken percentage* as "cold" branches (marked by the blue region), and branches with a medium *hit-to-taken percentage* as "warm" branches (marked by the yellow region).

Consequently, we introduce a new metric based on the *hit-to-taken percentage*, called the "branch temperature". The temperature of a branch indicates the branch's "hot/warm/cold" access behavior as observed under the optimal BTB replacement policy. In particular, for a given branch x with a *hit-to-taken percentage* equal to y, we define x's branch temperature as:

$$\text{Temperature } (x) = \begin{cases} \text{cold} & y \le y_1 \\ \text{warm} & y_1 < y \le y_2 \\ \text{hot} & y > y_2, \end{cases}$$

where y_1 and y_2 are two empirically decided thresholds such that $0 \le y_1 \le y_2 \le 1$. In our experiments, we find that using $y_1 = 50\%, y_2 = 80\%$ works best. As shown in Fig. 6, only half of all unique branches are hot and consistently retained in the BTB by the optimal replacement policy.

Next, we analyze all dynamic BTB accesses to classify them based on the branch temperature. In Fig. 7, the X-axis represents the percentage of unique taken branches while the Y-axis represents

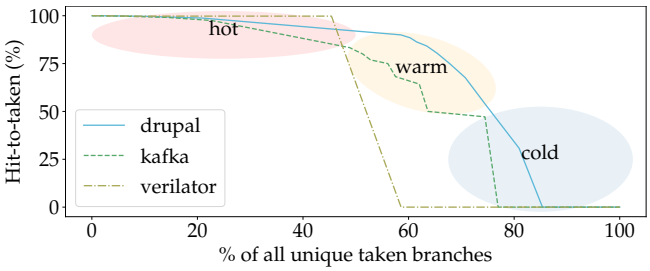

Figure 6: The distribution of *hit-to-taken percentage* using the optimal BTB replacement policy for all unique taken branches. Branches are sorted (in descending order) based on their *hit-to-taken percentage*. The optimal policy consistently retains "hot" branches (half of all unique branches) and rarely stores "cold" branches (20% of all unique branches).

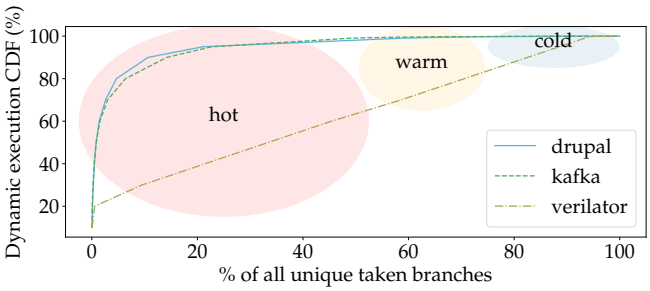

Figure 7: The cumulative distribution of all dynamic BTB accesses for all unique taken branches. Branches are sorted (in descending order) based on their *hit-to-taken percentage* for the optimal BTB replacement policy. "Hot" branches constitute a large majority (90%) of all BTB accesses.

the percentage of dynamically taken branches, *i.e.*, the percentage of BTB accesses. We also mark regions of the corresponding "hot/warm/cold" branches as defined in Fig. 6.

We find that the "hot" branches marked in Fig. 7 account for more than 90% of dynamically taken branches. Therefore, when making replacement decisions, we can achieve a near-optimal performance if we retained more "hot" branches in the BTB and evict the "cold" and "warm" branches.

Finally, we compute the correlation between branch temperature and the holistic (average) reuse distance. As shown in Fig. 8, branch temperature is strongly correlated with the holistic reuse distance. Therefore, branch temperature is able to capture the holistic behavior of branches over time.

In Fig. 8, we also show if branch temperature has any correlation with properties of branch instructions such as the branch type (*e.g.*, conditional and unconditional branches), branch target distance, and branch bias. If any of these properties have a strong correlation with the branch temperature, we could predict branch temperature based on those correlated properties without simulating the optimal BTB replacement policy on the entire application's trace. However, we observe that these branch properties do not have any strong correlation with the branch temperature. Therefore, we must compute the branch temperature by simulating the optimal BTB replacement policy on a data center application's trace.

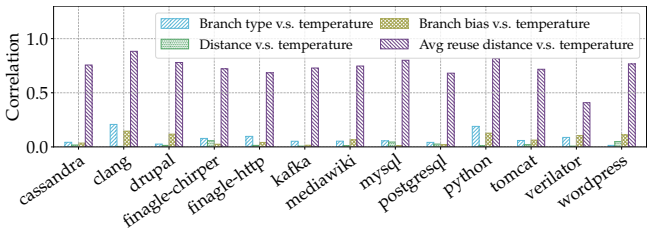

Figure 8: Correlation between branch type, target distance, bias, holistic reuse distance and branch temperature: holistic reuse distance and branch temperature are strongly correlated, however, branch type, bias, or target distance do not have strong correlation with branch temperature.

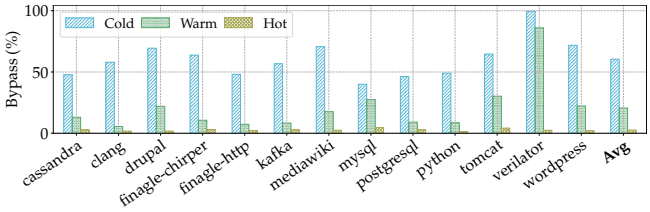

Figure 9: Average percentage of bypass out of all misses for branches with different temperature: the optimal replacement policy does not even insert cold branches into the BTB in more than 50% cases.

Observation: Branch "temperature" (determined by a branch's *hit-to-taken percentage*) is a good metric to drive efficient BTB replacement decisions.
Insight: *Evicting cold or warm branches while keeping hot branches in the BTB can help attain near-optimal performance. However, accurate measurement of branch temperature requires simulation of the optimal BTB replacement policy on data center applications' traces.*

2.5 Which entries are worth inserting into the BTB?

In §2.4, we showed how to redesign BTB replacement using the branch temperature. A BTB entry is replaced when a new entry is inserted. Now, we investigate if some of these insertions can be avoided to begin with, *i.e.*, whether the BTB can be bypassed [45, 90] for some branches based on its temperature. For this, we measure the number of times a branch is inserted into the BTB and the number of times it bypasses the BTB, using the optimal replacement policy. We use these measurements to compute the average bypass ratio for branches in each temperature category.

As shown in Fig. 9, both cold and warm branches have a higher bypass ratio, while hot branches have a lower bypass ratio. Hence, for a given cold or warm branch, we must compare the branch's temperature with the temperature of branches currently in the BTB to determine whether this branch must bypass the BTB. In contrast, based on Fig. 9, we must always insert hot branches into the BTB to make near-optimal replacement decisions.

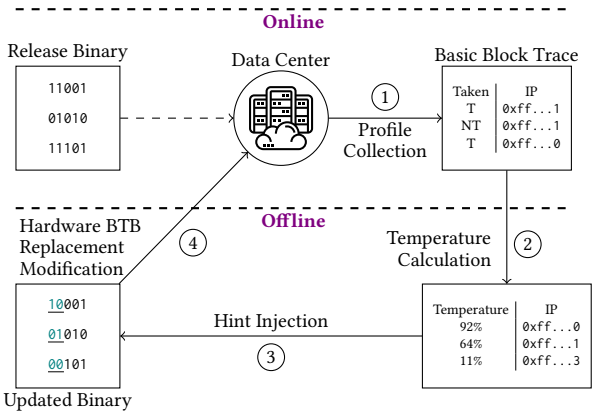

Figure 10: High level design of *Thermometer*

Observation: The optimal replacement policy inserts most hot branches into the BTB, while (on average) half of all cold branches are not inserted.
Insight: *Before inserting a branch in the BTB, comparing it's temperature with branches that are already in the BTB to determine whether to bypass or not, can help make near-optimal decisions.*

3 DESIGN OF THERMOMETER

Our analysis shows that BTB replacement policies significantly affect the performance of data center applications. While the optimal BTB replacement policy achieves an average IPC speedup of 10.4% for data center applications, prior replacement policies [20, 60, 62] are unable to provide a substantial performance benefit (only 1.5% mean IPC speedup) over LRU. Prior replacement policies fall short since they only leverage transient branch information and do not consider holistic branch behavior. We now present *Thermometer*, a novel BTB replacement technique that leverages hardware-software co-design to accommodate both holistic and transient branch behavior of data center applications. Specifically, *Thermometer* introduces a profile-guided software mechanism to learn holistic branch behavior and then introduces minor hardware modifications to the replacement policy to consider both behaviors, enabling near-optimal replacement decisions.

Thermometer determines branch temperature based on the *hit-to-taken percentage* under the optimal replacement policy using a profile-guided analysis. Branch instructions are annotated with their temperature and stored as part of a BTB entry whenever a branch is inserted into the BTB. Whenever the replacement policy needs to determine an eviction candidate it considers both temperature and LRU information of the candidates as described in Algorithm 1. In particular, *Thermometer* first selects the coldest branch (including the branch to be inserted in BTB) for eviction or, in the case of a tie, selects a candidate based on LRU.

We show all four of *Thermometer*'s design components in Fig. 10. In step ① (§3.1), *Thermometer* collects the basic block execution profile of data center applications at run time with the help of efficient hardware mechanisms [1, 8, 76, 79]. In step ② (§3.2), *Thermometer* simulates the optimal BTB replacement policy offline on the branch execution profile to determine the temperature of all branch instructions. In step ③ (§3.3), *Thermometer* encodes the temperature as a hint in the branch instruction. Finally, in step ④ (§3.4), *Thermometer*'s updated BTB replacement policy leverages *Thermometer*-injected hints to make close-to-optimal replacement decisions. Now, we describe each of these four components in detail.

3.1 Profile Collection

Thermometer collects the basic block execution trace using Intel PT [1]. Similar to prior work [78], *Thermometer* uses Intel PT due to its low runtime overhead (only up to 1% [69–71, 151]) and widespread adoption in today's data centers [36, 39]. Intel PT provides *Thermometer* with a trace of dynamically executed branch instructions. As we show in Fig. 10, the trace contains two specific data points for each branch instruction. First, the trace includes the direction of a branch, *i.e.*, taken (T) or not-taken (NT). Second, in case of a taken indirect branch, the trace also contains the address of the next executed instruction. While Intel PT provides a comprehensive execution history that enables control-flow analysis, it does not collect any data about BTB replacement actions.

3.2 Measuring the Branch Temperature

Thermometer simulates the branch execution trace offline using the optimal replacement policy (Belady's algorithm [29]) to measure the temperature of all branch instructions in the application. As we show in §4.2, the overhead of simulating the optimal replacement policy is similar to those of widely-adopted profile-guided optimization techniques [106, 107]. To calculate branch temperature, *Thermometer* counts two metrics for each branch instruction. First, *Thermometer* counts the times a given branch instruction is taken during the program execution. Second, *Thermometer* counts the times when the taken branch's target can be found in the BTB while operating under the optimal replacement policy. *Thermometer* computes the temperature for each branch instruction after dividing the second value (BTB hit count under optimal replacement policy) by the first value (branch taken count) and expressing the division result as a fraction of 100.

3.3 Hint Injection

The goal of *Thermometer*'s hint injection is to mark hot and cold branch instructions differently so that the BTB replacement policy can evict cold branches while keeping hot branches in the BTB. In the context of hint injection, *Thermometer* faces two main design decisions: (1) how many temperature categories (and resulting bits) to use and (2) which temperature thresholds to use for classifying branches into one of these categories.

Hint size. Encoding the temperature as part of every branch instruction increases the instruction working set size and also requires additional storage in the BTB. For example, assuming an 8K entry BTB and 16 temperature categories, *Thermometer* introduces a storage overhead of 4KB which may be better invested in additional BTB entries. Using a small number of bits, on the other hand, introduces quantization errors as *Thermometer* may inject the same hint for two branches with a high temperature difference. It is therefore important to pick a suitable number of temperature categories while minimizing storage overheads. In practice, we find that a 2-bit

hint offers a good trade-off between quantization inaccuracy and storage overhead. Such a 2-bit hint is practically implementable since both x86 and arch64 branch instruction formats have at least 2 unused bits reserved for future usage [50, 124]. Consequently, a 2-bit hint enables *Thermometer* to provide the temperature category as a replacement hint without incurring any overhead to the application's code footprint. We study *Thermometer*'s sensitivity for different hint sizes in §4.3.

Temperature thresholds. Depending on the number of temperature categories, *Thermometer* must determine how to classify branches into categories. A naive approach is to divide branches uniformly so that all temperature categories have an equal number of branches. However, with such an approach, *Thermometer* may group two branches with high temperature difference together in the same category. Such a categorization might occur especially for branches near the cliffs [28] (areas with high slope in Figure 6, where temperature drops sharply). To address this issue, *Thermometer* assigns temperature categories to different branch instructions based on empirically determined temperature thresholds. Specifically, with three categories (*i.e.*, hot, warm, and cold), we observe that the temperature thresholds, 80% and 50% yield the best performance for data center applications. If needed, these thresholds are configurable to meet different applications' behavior.

3.4 Microarchitectural Modifications

Thermometer extends the BTB replacement policy implemented in hardware to consider the temperature hint encoded in the branch instruction. As *Thermometer* encodes a 2-bit hint for each branch instruction, *Thermometer* modifies the baseline BTB to include 2 extra bits per BTB entry. For an 8K-entry, 4-way, 75KB baseline BTB, this modification introduces a 2KB of extra storage overhead (2.67%). Using these extra bits, the BTB replacement policy can distinguish hot (or BTB-friendly entries) from cold (or BTB-averse entries) to make eviction decisions during the program execution.

Algorithm 1 presents a simplified version of *Thermometer*'s BTB replacement policy implemented in hardware. The algorithm takes a list of branch instructions as input and returns the victim branch instruction to be evicted from the BTB as output. Along with branches that are already in the BTB, the algorithm also considers the current branch instruction, x_0 for which the new entry would be inserted into the BTB, as the potential victim. For all these instructions, the algorithm populates the temperature (Line 1- 2) and then finds the coldest temperature, t (Line 3) among them to leverage holistic reuse behavior. Next, the algorithm considers all branch instructions with the coldest temperature t as possible victim candidates (Line 4). Among those victim candidates *Thermometer* selects the final line according to the least recently used heuristic (Line 7) leveraging transient reuse behavior. Thus, *Thermometer* combines the best of both worlds: holistic and transient branch reuse behavior to make effective BTB replacement decisions.

Thermometer adds one extra operation over the LRU baseline, which is finding the coldest temperature. For a 4-way BTB, this operation requires comparing five (A, B, C, D, E) 2-bit values. We can compute whether A is the coldest way as, A_c = $(A<B)$ & $(A<C)$ & $(A<D)$ & $(A<E)$. Each of these comparisons has an overall gate delay of only 3 logic gates (*e.g.*, $A<B = (B_1 \& !A_1) \mid (B_0 \& B_1 \& !A_0) \mid (!A_1 \& !A_0 \& B_0)$) and different comparisons (*e.g.*, $A<B$ and $A<C$) can be performed in parallel. Similarly, the logic to compute B_c and C_c can also be performed in parallel. Even if this whole logic cannot be computed in a single cycle it can be easily pipelined, *e.g.*, by registering the results of $A<B$, $A<C$, $A<D$, and $A<E$ in one cycle and performing the & operation in the next cycle. Finally, *Thermometer* can also ensure fast lookup for the newly inserted BTB entries by placing them in a small replacement buffer similar to 32-entry prefetch buffer used by state-of-the-art BTB prefetching solutions [75, 83].

Algorithm 1 BTB replacement policy (implemented in hardware) to consider both holistic and transient reuse behavior.

Input: Current branch to insert, x_0, branches already in the BTB, $x_i, i = 1, 2, \ldots, n$, n is the number of BTB ways.
Output: Victim, z

1: **for** $i = 0, 1, 2, \ldots, n$ **do**
2: $y_i \leftarrow$ temperature of x_i
3: $t \leftarrow \min(y_0, y_1, y_2, \ldots y_n)$ ▷ Find the coldest temperature
4: $S \leftarrow \{x_j : y_j = t\}$
5: **if** $x_0 \in S$ && $|S| = 1$ **then**
6: **return** x_0 ▷ Bypass
7: $z \leftarrow$ the least recently used branch in S.
8: **return** z

BTB size dependency. *Thermometer* categorizes branch instructions based on their temperature for a specific BTB size and associativity. While this classification is target architecture dependent, such target-dependent optimizations are already deployed in today's data centers by widely-used profile-guided optimization techniques [33, 85, 106, 107]. Data center operators (*e.g.*, Google and Facebook) already compile and deploy individual binaries for diverse processor types in their fleet [25, 33, 106, 107, 133]. Hence, *Thermometer* can be combined with existing build and deployment mechanisms used in real data centers today.

4 EVALUATION

In this section, we first describe our experimental methodology. Next, we evaluate how *Thermometer* improves data center applications' performance using several key metrics. Finally, we present various sensitivity studies, showing how different design parameters affect *Thermometer*'s effectiveness.

4.1 Methodology

Data center applications and inputs. As described in §2.1, we evaluate *Thermometer* using 13 widely-used data center applications. For these applications, we vary input configurations by changing the input data size (*e.g.*, `large` vs `small`), the webpage requested by the client (*e.g.*, `feed=rss2` vs `p=37`), the number of client requests per second (*e.g.*, `2` vs `10`), random number seeds (*e.g.*, `1` vs `10`), different query mapping styles (*e.g.*, `imperative` vs `declarative`), different database scaling factors (*e.g.*, `100` vs `8000`), and different database queries (*e.g.*, `oltp_read_only` vs `oltp_write_only`). We profile only a portion of each application's execution; this portion is different from the tested execution and uses different inputs. Apart from evaluating *Thermometer* on these 13 real-world applications, we also

evaluate *Thermometer* on a wide-range of common traces (663 CBP-5 [15] traces and 50 IPC-1 [17] traces) like prior work [20, 24, 57, 58].

4.2 Performance analysis

We evaluate *Thermometer*'s effectiveness using key performance metrics. First, we compare *Thermometer*'s IPC speedup to the speedup offered by the optimal BTB replacement policy and state-of-the-art BTB replacement techniques [20, 60, 62]. We also evaluate the BTB miss reduction *Thermometer* achieves. Next, we show how *Thermometer* generalizes across different application inputs. We also evaluate *Thermometer*'s replacement coverage and accuracy.

IPC speedup. We measure the IPC speedup that *Thermometer* achieves over an LRU baseline for 13 data center applications. Fig. 11 shows *Thermometer*'s speedup compared against speedups achieved by the optimal BTB replacement policy and three existing replacement policies (SRRIP [62], GHRP [20], and Hawkeye [60]). We find that *Thermometer* always outperforms prior replacement policies and achieves comparable performance to the theoretically optimal BTB replacement policy. In particular, *Thermometer* provides 8.7% average speedup compared to 10.4% average speedup achieved by the optimal BTB replacement policy. In other words, *Thermometer* achieves an average speedup that is 83.6% of the average speedup achieved by the optimal BTB replacement policy. The small performance gap between *Thermometer* and the optimal BTB replacement policy stems from few cases where the branch behavior temporally diverges from both the profiled holistic and transient branch behavior.

We also investigate whether *Thermometer* would improve performance of a 75KB, 8K entry BTB when considering 2-bit overhead for each branch in BTB. We measure the speedup gained by a 7979-entry BTB that uses *Thermometer* over an 8K entry BTB that uses LRU. We ensure the same BTB size since 7979 × (entry size + 2 bits overhead) = 8192 × entry size = 75K. As shown, a 7979-entry BTB that uses *Thermometer* significantly outperforms existing BTB replacement mechanisms and achieves comparable performance to the optimal BTB replacement policy.

We use address modulo total number of BTB sets as the BTB hash function. For this function, the 7979-entry BTB distributes branches for some applications (*e.g.*, `cassandra`, `kafka`, `mysql`) more uniformly than the 8192-entry BTB. Consequently, *Thermometer* achieves slightly better performance with the 7979-entry BTB than the 8192-entry BTB for these applications.

BTB miss reduction. Fig. 12 shows the BTB miss reduction over LRU achieved by *Thermometer* and prior replacement policies. *Thermometer* achieves an average BTB miss reduction of 21.3%, outperforming existing replacement policies which achieve at most 6.7% average miss reduction. *Thermometer*'s performance corresponds to 62.6% of the performance of the optimal BTB replacement policy which achieves an average miss reduction of 34%.

Performance across different application inputs. Computing branch temperatures for one program input still provides replacement benefits for a different application input since on average 81% of all branches fall in the same temperature category across different inputs. We quantify this benefit in terms of *Thermometer*'s speedup using three separate input configurations ('#1' to '#3'). We optimize each application using the training profile from input '#0' and measure *Thermometer*'s speedup for all three different test inputs ('#1, #2, #3') in Fig. 13 (indicated as 'training-profile'). Next, we measure the speedup when *Thermometer* optimizes each application with the same input's profile for comparison, *i.e.*, *Thermometer*'s speedup for input '#1' using profile information that is also gathered using input '#1'. Fig. 13 shows this result as 'Same-input-profile'. As shown, *Thermometer* provides significant speedup across different application inputs even with the training input's (different from the test input) profile since most branches have same temperature across different inputs.

For some applications (*e.g.*, `finagle-chirper` and `postgresql`), *Thermometer*'s speedup with the training input's profile is even greater than *Thermometer*'s speedup with the same input's profile even though the training input's profile causes slightly more BTB misses than the same input's profile. In these cases, we find that training input's profile triggers less expensive BTB misses than the same input's profile as BTB misses incur variable miss penalty.

Static and dynamic overhead. For each branch instruction, *Thermometer* introduces 2 bits for encoding the temperature category. Both ARM and x86 branch instructions have at least 2 unused bits reserved in the ISA for future optimizations [50, 124], which we can use to encode the category information without any overhead in the new binary.

Profiling overhead and cost of optimal replacement policy simulation. There is no extra online cost of *Thermometer*'s profiling, as data center applications are already routinely profiled with Intel LBR and PT [27, 33, 36, 39, 106, 107]. As we describe in §3, *Thermometer* simulates the optimal BTB replacement policy offline. The execution time for this offline simulation is in the order of seconds (4.18-167 seconds and 23.53 seconds on average), as shown in Fig.14. These durations are similar to those of existing post-link profile-guided optimization techniques [106, 107] (19.5-168.3 seconds [107]).

Replacement coverage and accuracy. We define "not covered by *Thermometer*" as the cases when all branches in a target set are in the "coldest category", and *Thermometer* relies on LRU to choose a victim. This case will be similar to the LRU baseline. We measure *Thermometer*'s replacement coverage in terms of the percentage of evictions that are "covered by *Thermometer*". As show in Fig. 15, *Thermometer* achieves an average coverage of 61.4%.

In Fig. 16, we also show the replacement accuracy by measuring the percentage of victims whose reuse distance is equal to or larger than the number of BTB ways. In particular, we evaluate 3 techniques. The first technique only considers transient reuse behavior. The second technique only considers holistic reuse behavior. The third technique, *Thermometer*, utilizes both transient and holistic reuse information. Note that the optimal replacement policy always ensures 100% replacement accuracy. On average, transient behavior achieves 46.06% accuracy, holistic behavior achieves 63.72% accuracy, and *Thermometer* achieves 68.20% accuracy.

BTB miss reduction on CBP-5 traces. We also validate *Thermometer*'s effectiveness in reducing BTB misses for 663 CBP-5 traces [15]. Since these traces do not allow generating IPC numbers [20], we measure the BTB miss reduction (%) achieved by *Thermometer* over the best performing prior work, GHRP [20] for

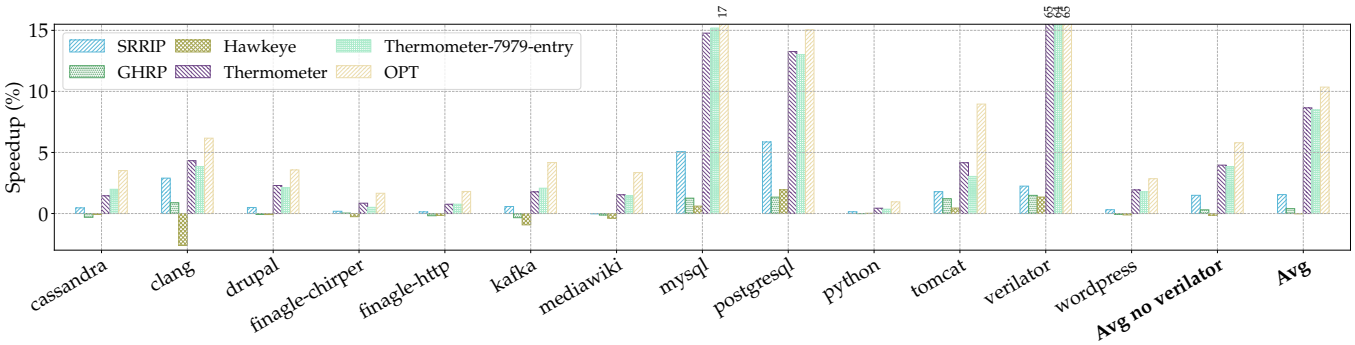

Figure 11: *Thermometer*'s IPC speedup compared to optimal and state-of-the-art replacement policies over an LRU baseline (with FDIP): *Thermometer* achieves an average speedup of 8.7% that is 83.6% of the average speedup provided by the optimal BTB replacement policy.

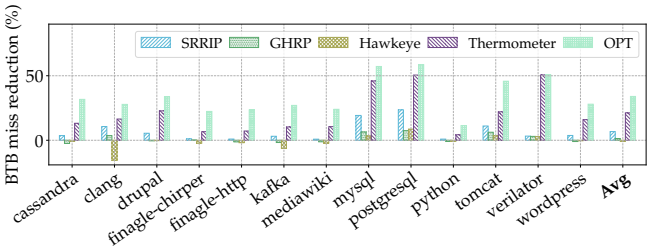

Figure 12: *Thermometer*'s BTB miss reduction over an LRU baseline (with FDIP): *Thermometer* reduces 21.3% of all BTB misses compared to 34% miss reduction achieved by the optimal replacement policy.

these traces. As shown in Fig. 17, *Thermometer* achieves an average BTB miss reduction of 2.25% over GHRP.

Among all traces (52) that face BTB Misses Per Kilo Instructions (MPKI) of 1 or greater, *Thermometer* outperforms GHRP by 11.48% on average. *Thermometer* outperforms GHRP for 306 out of 663 traces while GHRP outperforms *Thermometer* only for 59 traces. The remaining 298 traces only suffer from compulsory BTB misses and all replacement policies achieve identical performance.

Thermometer could not outperform GHRP for 59 traces because the 50% and 80% thresholds do not yield the best performance for those traces. Consequently, when we find a better threshold using two-fold cross-validation [145] for these 663 traces, *Thermometer* outperforms GHRP for all but 32 traces, as also shown in Fig. 17.

Speedup on IPC-1 traces. To further evaluate *Thermometer*, we also measure and show the IPC speedup achieved by *Thermometer* for 50 IPC-1 [17] traces in Fig. 18. *Thermometer* achieves an average IPC speedup of 1.07% (up to 5.36%) and outperforms prior replacement mechanisms. The best-performing prior work, SRRIP offers merely 0.45% speedup on average. Among all traces (9) that suffer from a BTB MPKI of at least 1, *Thermometer* achieves an average IPC speedup of 3.59%. For one of the IPC-1 traces (`server_010`), both *Thermometer* and SRRIP outperform the optimal replacement policy in terms of IPC speedup though they incur more BTB misses than the optimal replacement policy. That is because the optimal replacement policy is optimal in terms of reducing the total number of misses and does not consider the variable latency incurred by each individual misses [116]. Nevertheless, *Thermometer* achieves 85.7% of the speedup (on average 1.25%) provided by the optimal BTB replacement policy. Therefore, *Thermometer* is able to make near-optimal BTB replacement decisions for IPC-1 traces.

4.3 Sensitivity analysis

Number of BTB entries. We vary the number of BTB entries from 1024 to 32768 to measure how sensitive *Thermometer* is to the BTB size. As shown in Fig. 19 (left), *Thermometer* outperforms SRRIP significantly for any BTB size and performs better relative to the optimal BTB replacement policy with a larger BTB size.

BTB associativity. We also measure *Thermometer*'s sensitivity to the BTB associativity by varying the number of BTB ways from 4 to 128. As shown in Fig. 19 (right), *Thermometer* outperforms SRRIP significantly for any number of BTB ways. For some traces like `cassandra` and `drupal`, *Thermometer*'s performance relative to the optimal BTB replacement policy decreases as the number of BTB ways increases, while for other traces like `tomcat`, *Thermometer*'s performance increases as the number of BTB ways increases.

Number of bits encoding branch temperature. We investigate *Thermometer*'s effectiveness with various hint sizes to encode branch temperatures. We change the number of encoding bits from 1 to 4 and choose 2, 3, 4, 8, 16 categories to measure *Thermometer*'s speedup compared to the optimal BTB replacement policy. As shown in Fig. 20 (left), *Thermometer* achieves the best performance when (1) using 2 bits to encode categories and (2) classifying branches into 3 or 4 categories. With fewer categories (*e.g.*, 2 categories), *Thermometer* cannot characterize all branches' reuse behavior, reducing coverage as more branches are misclassified into the same category. Using more categories such as 8 or 16, leads to separation of branches with similar reuse behavior into different categories, reducing the opportunity for the backing LRU policy to determine transient changes in the reuse behavior dynamically.

FDIP run-ahead. We evaluate *Thermometer*'s sensitivity to the size of Fetch Target Queue (FTQ), *i.e.*, the maximum run-ahead distance of the decoupled frontend. We carry out the experiment using FTQ sizes of {64, 128, 192, 256} and measure the optimal BTB speedup percentage achieved by *Thermometer*. As shown in Fig. 20 (right),

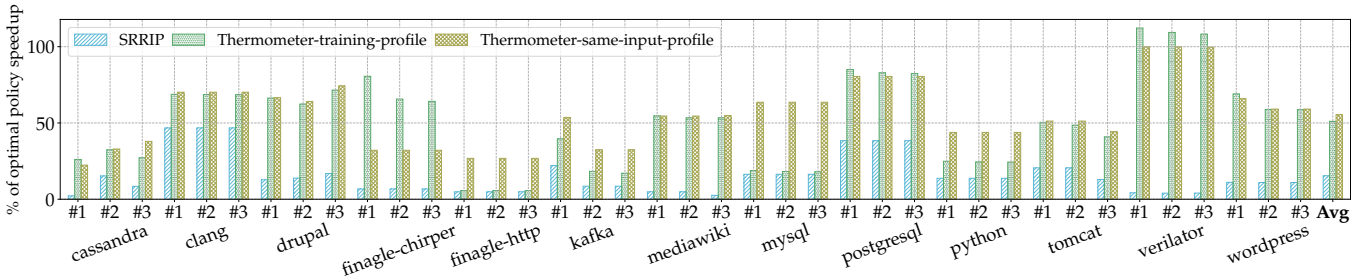

Figure 13: *Thermometer*'s IPC speedup across different application inputs as the percentage of the optimal BTB performance

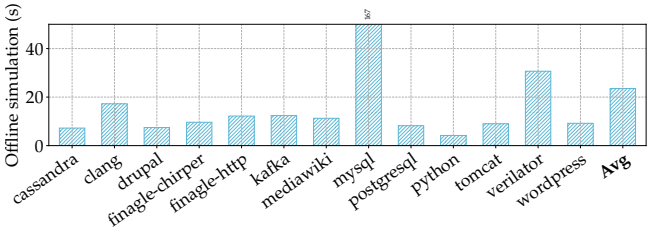

Figure 14: Offline simulation time for the optimal replacement policy. The execution time is 4.18-167 seconds and 23.53 seconds on average.

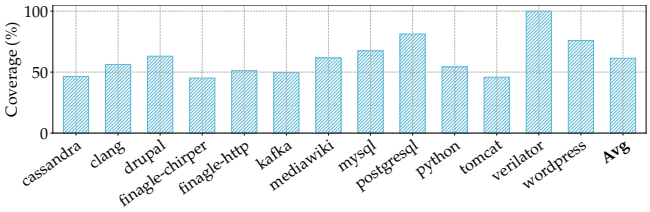

Figure 15: Replacement coverage of *Thermometer* for various applications: 61.4% of replacement requests are processed by evicting BTB entries that are marked colder by *Thermometer*.

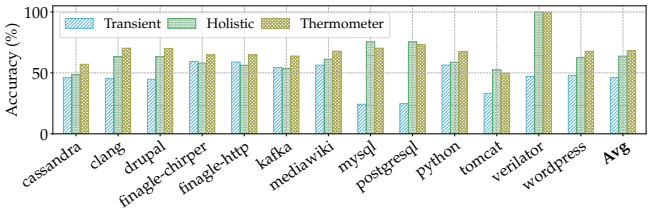

Figure 16: Replacement accuracy of *Thermometer*, transient and holistic behavior: on average transient reuse behavior based replacement decisions achieve 46.06% accuracy, holistic reuse behavior based replacement decisions achieve 63.72% accuracy, combining both of them, *Thermometer* achieves 68.20% accuracy.

Thermometer achieves almost constant speedup relative to the optimal BTB replacement policy with different FTQ sizes. Therefore, *Thermometer* generalizes well for different FDIP implementations.

Prefetch-aware replacement. We evaluate *Thermometer*'s sensitivity to the state-of-the-art BTB prefetching mechanism, Twig [75] and show the results in Fig. 21. As shown, the combination of *Thermometer* and Twig provides an average IPC speedup of 30.9% over

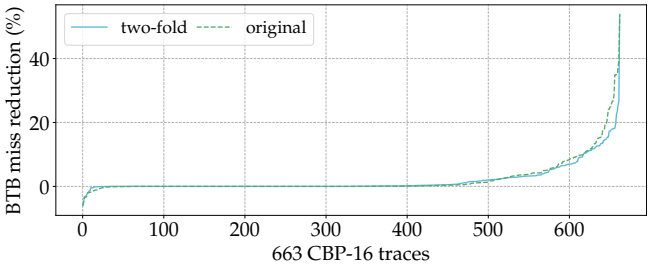

Figure 17: *Thermometer*'s BTB miss reduction over the best performing prior work, GHRP [20] on CBP-5 traces [15]: *Thermometer* achieves an average BTB miss reduction of 2.25% over the GHRP baseline.

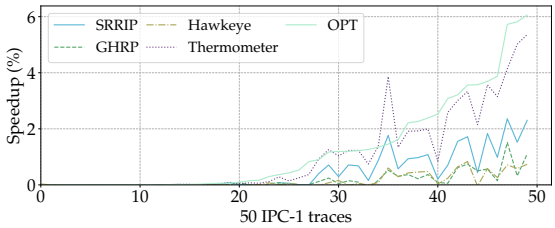

Figure 18: *Thermometer*'s IPC speedup over an LRU baseline (with FDIP) for 50 IPC-1 traces [17]: *Thermometer* outperforms all prior replacement techniques and provides speedups comparable to the optimal BTB replacement policy.

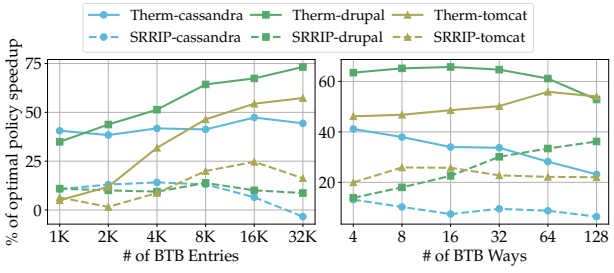

Figure 19: % of speedup *Thermometer* obtains compared to the optimal BTB replacement policy while varying the number of BTB entries (left) and the number of BTB ways (right).

the baseline combination of LRU and Twig. Even with BTB prefetching, *Thermometer* significantly outperforms the best performing prior replacement policy (SRRIP) which provides only 1.37% mean speedup. On average, *Thermometer*'s speedup is 95.9% of the average speedup (32.2%) provided by the optimal replacement policy.

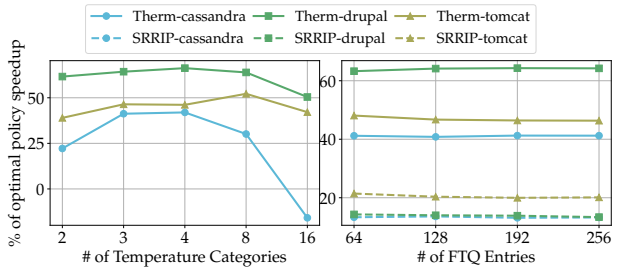

Figure 20: % of speedup *Thermometer* obtains compared to the optimal BTB replacement policy while varying the number of branch temperature categories (left) and the FTQ size (right).

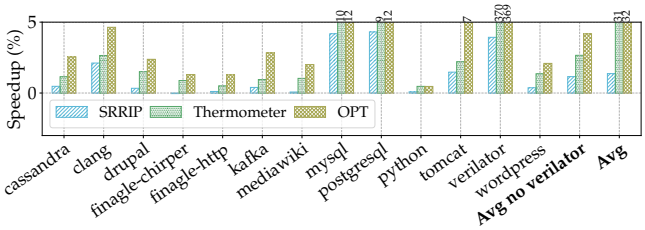

Figure 21: *Thermometer*'s speedup over an LRU baseline with state-of-the-art BTB prefetching, Twig [75]. *Thermometer* +Twig achieves an average speedup of 30.9% over the LRU+Twig baseline. Thus, *Thermometer* achieves 95.9% of the average speedup (32.2%) provided by the optimal replacement policy.

5 RELATED WORK

I-cache performance optimization. There are three main classes of prior techniques that optimize I-cache performance: software, hardware, and hybrid software-hardware. Software techniques include hot/cold splitting [34], block reordering [105, 113], and other profile-guided optimization approaches [26, 33, 52, 54, 63, 89, 91, 94, 95, 106, 111, 117, 149]. While software techniques improve instruction locality and eliminate a subset of all I-cache misses, in practice it is intractable to find the optimal code layout [27, 112]. Hardware techniques include next-line prefetchers [131], path-based prefetchers [59, 122], record-and-replay prefetchers [43, 44, 72, 73], and BTB-directed prefetchers [83, 84]. These techniques usually have one of two limitations: (1) the design is more complex [72, 73, 82] than real hardware prefetching implementations [119, 122] or (2) the on-chip storage cost is too high [43, 44]. Hybrid techniques [27, 77] combine the effectiveness of software and hardware mechanisms but further increase the code footprint of data center applications. FDIP [118, 119] offers a good trade-off among existing I-cache prefetchers as it provides performance comparable to state-of-the-art prefetchers [22, 46, 48, 51, 99, 101, 120, 127] without incurring a high storage cost [57, 58]. Consequently, FDIP and its variants are employed in modern processors [49, 109, 123, 135]. Therefore, we attempted to improve the effectiveness of FDIP in this work through an optimized BTB replacement policy which requires negligible extra storage and hardware modification.

BTB redesign and compression. In addition to storing branch targets [87, 110], BTB entries generally contain a tag and prediction information, while block-oriented BTBs also need to store the size of the corresponding basic block [148]. Existing work [24, 32, 40, 65, 81, 110, 118, 128, 132] proposes BTB compression techniques including using tags of fewer bits, removing page number, storing the branch target as a distance from the branch PC, or using a multi-level BTB. These techniques are orthogonal and can be combined with *Thermometer* to further improve the BTB storage efficiency.

Cache and BTB replacement mechanisms. Prior cache replacement policies can be categorized into two types: heuristic-based and learning-based replacement policies. The former category consists of LRU [68, 88, 102, 130], MRU [115], reuse distance prediction [38, 41, 93], re-reference interval prediction (RRIP [62]), and additional policies [18, 45, 56, 80, 116, 125, 134, 136]. In the latter category, some learning-based replacement policies [61, 74, 147] classify cache lines as cache-friendly and cache-adverse, others [60, 92, 129] use information provided by the optimal replacement policy [29]. Some recent replacement policies also use machine learning mechanisms[64, 66, 137]. However, existing replacement polices are mainly designed for data caches and fall short when applied to the BTB. GHRP [20] is a replacement policy designed for the BTB, but it falls short for data center applications due to their large instruction footprints. We propose a novel profile-guided BTB replacement approach that outperforms these existing replacement policies.

6 CONCLUSION

Data center applications exhibit large branch footprints and suffer from frequent BTB misses. Prior BTB prefetching and replacement mechanisms cannot mitigate these misses as they lack proper understanding of branch temperature in data center applications. In this work, we propose *Thermometer*, a profile-guided BTB replacement mechanism that considers both holistic and transient branch behaviour in data center applications. For 13 widely-used data center applications, *Thermometer* provides on average 8.7% (0.4%-64.9%) speedup that is 83.6% of the mean speedup achieved by the optimal BTB replacement policy.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their insightful feedback and suggestions. This work was supported by the generous gifts from Intel Labs, NSF grants #1942754, #2010810, CCF-1912617, CNS-1938064, a Rackham Predoctoral Fellowship, and the Applications Driving Architectures (ADA) Research Center, a JUMP Center co-sponsored by SRC and DARPA. We thank the University of Michigan Summer Undergraduate Research in Engineering (SURE) program as Shixin Song completed this work as her summer undergraduate research project. 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.

REFERENCES

[1] "Adding processor trace support to linux," https://lwn.net/Articles/648154/.
[2] "Apache cassandra," http://cassandra.apache.org/.
[3] "Apache kafka," https://kafka.apache.org/powered-by.
[4] "Apache tomcat," https://tomcat.apache.org/.
[5] "Champsim," https://github.com/ChampSim/ChampSim.
[6] "Clang c language family frontend for llvm," [Online; accessed 19-Nov-2021]. [Online]. Available: https://clang.llvm.org/

[7] "Github - chipsalliance/rocket-chip: Rocket chip generator," [Online; accessed 19-Nov-2021]. [Online]. Available: https://github.com/chipsalliance/rocket-chip
[8] "An introduction to last branch records," https://lwn.net/Articles/680985/.
[9] "Postgresql: Documentation: 14: pgbench," [Online; accessed 19-Nov-2021]. [Online]. Available: https://www.postgresql.org/docs/current/pgbench.html
[10] "Postgresql: The world's most advanced open source database," [Online; accessed 19-Nov-2021]. [Online]. Available: https://www.postgresql.org/
[11] "The python performance benchmark suite," [Online; accessed 19-Nov-2021]. [Online]. Available: https://pyperformance.readthedocs.io/
[12] "Twitter finagle," https://twitter.github.io/finagle/.
[13] "Verilator," https://www.veripool.org/wiki/verilator.
[14] "Welcome to python.org," [Online; accessed 19-Nov-2021]. [Online]. Available: https://www.python.org/
[15] "Championship branch prediction," https://jilp.org/cbp2016/, 2016.
[16] "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).
[17] "The 1st instruction prefetching championship," https://research.ece.ncsu.edu/ipc/, 2020.
[18] J. Abella, A. González, X. Vera, and M. F. O'Boyle, "Iatac: a smart predictor to turn-off l2 cache lines," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 2, no. 1, pp. 55–77, 2005.
[19] 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.
[20] S. M. Ajorpaz, E. Garza, S. Jindal, and D. A. Jiménez, "Exploring predictive replacement policies for instruction cache and branch target buffer," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2018, pp. 519–532.
[21] S. Ananthanarayanan, M. S. Ardekani, D. Haenikel, B. Varadarajan, S. Soriano, D. Patel, and A.-R. Adl-Tabatabai, "Keeping master green at scale," in *Proceedings of the Fourteenth EuroSys Conference 2019*, ser. EuroSys '19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3302424.3303970
[22] A. Ansari, F. Golshan, P. Lotfi-Kamran, and H. Sarbazi-Azad, "Mana: Microarchitecting an instruction prefetcher," *The First Instruction Prefetching Championship*, 2020.
[23] A. Ansari, P. Lotfi-Kamran, and H. Sarbazi-Azad, "Divide and conquer frontend bottleneck," in *Proceedings of the 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020.
[24] T. Asheim, B. Grot, and R. Kumar, "Btb-x: A storage-effective btb organization," *IEEE Computer Architecture Letters*, vol. 20, no. 2, pp. 134–137, 2021.
[25] 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.
[26] G. Ayers, H. Litz, C. Kozyrakis, and P. Ranganathan, "Classifying memory access patterns for prefetching," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 513–526.
[27] 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.
[28] N. Beckmann and D. Sanchez, "Talus: A simple way to remove cliffs in cache performance," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2015, pp. 64–75.
[29] L. A. Belady, "A study of replacement algorithms for a virtual-storage computer," *IBM Systems journal*, vol. 5, no. 2, pp. 78–101, 1966.
[30] L. A. Belady and F. P. Palermo, "On-line measurement of paging behavior by the multivalued min algorithm," *IBM Journal of Research and Development*, vol. 18, no. 1, pp. 2–19, 1974.
[31] 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.
[32] M. Butler, L. Barnes, D. D. Sarma, and B. Gelinas, "Bulldozer: An approach to multithreaded compute performance," *IEEE Micro*, vol. 31, no. 2, pp. 6–15, 2011.
[33] D. Chen, T. Moseley, and D. X. Li, "Autofdo: Automatic feedback-directed optimization for warehouse-scale applications," in *CGO*, 2016.
[34] R. Cohn and P. G. Lowney, "Hot cold optimization of large windows/nt applications," in *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture. MICRO 29*. IEEE, 1996, pp. 80–89.
[35] T. P. P. Council, "Tpc-c," [Online; accessed 19-Nov-2021]. [Online]. Available: http://www.tpc.org/tpcc/
[36] 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.
[37] 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.
[38] N. Duong, D. Zhao, T. Kim, R. Cammarota, M. Valero, and A. V. Veidenbaum, "Improving cache management policies using dynamic reuse distances," in *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 2012, pp. 389–400.
[39] W. Erquinigo, D. Carrillo-Cisneros, and A. Tang, "Reverse debugging at scale," https://engineering.fb.com/2021/04/27/developer-tools/reverse-debugging/.
[40] B. Fagin, "Partial resolution in branch target buffers," *IEEE Transactions on Computers*, vol. 46, no. 10, pp. 1142–1145, 1997.
[41] P. Faldu and B. Grot, "Leeway: Addressing variability in dead-block prediction for last-level caches," in *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 2017, pp. 180–193.
[42] 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.
[43] M. Ferdman, C. Kaynak, and B. Falsafi, "Proactive instruction fetch," in *International Symposium on Microarchitecture*, 2011.
[44] M. Ferdman, T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos, "Temporal instruction fetch streaming," in *International Symposium on Microarchitecture*, 2008.
[45] H. Gao and C. Wilkerson, "A dueling segmented lru replacement algorithm with adaptive bypassing," in *JWAC 2010-1st JILP Worshop on Computer Architecture Competitions: Cache Replacement Championship*, 2010.
[46] N. Gober, G. Chacon, D. Jiménez, and P. V. Gratz, "The temporal ancestry prefetcher."
[47] Google, "Propeller: Profile guided optimizing large scale llvm-based relinker," https://github.com/google/llvm-propeller, 2020.
[48] D. A. J. P. V. Gratz and G. C. N. Gober, "Barca: Branch agnostic region searching algorithm."
[49] 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.
[50] P. Guide, "Intel® 64 and ia-32 architectures software developer's manual," *Volume 3B: System programming Guide, Part*, vol. 2, no. 11, 2011.
[51] V. Gupta, N. S. Kalani, and B. Panda, "Run-jump-run: Bouquet of instruction pointer jumpers for high performance instruction prefetching."
[52] S. Harizopoulos and A. Ailamaki, "Steps towards cache-resident transaction processing," in *International conference on Very large data bases*, 2004.
[53] I. Harshard Sane, Principle Software Engineer, "Active benchmarking for better performance predictions," https://www.intel.com/content/dam/www/central-libraries/us/en/documents/dpm-workloads-explainer-tech-brief.pdf.
[54] M. Hashemi, K. Swersky, J. A. Smith, G. Ayers, H. Litz, J. Chang, C. Kozyrakis, and P. Ranganathan, "Learning memory access patterns," *arXiv preprint arXiv:1803.02329*, 2018.
[55] 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.
[56] Z. Hu, S. Kaxiras, and M. Martonosi, "Timekeeping in the memory system: predicting and optimizing memory behavior," in *Proceedings 29th Annual International Symposium on Computer Architecture*. IEEE, 2002, pp. 209–220.
[57] Y. Ishii, J. Lee, K. Nathella, and D. Sunwoo, "Rebasing instruction prefetching: An industry perspective," *IEEE Computer Architecture Letters*, 2020.
[58] Y. Ishii, J. Lee, K. Nathella, and D. Sunwoo, "Re-establishing fetch-directed instruction prefetching: An industry perspective," *IEEE International Symposium on Performance Analysis of Systems and Software*, 2021.
[59] Q. Jacobson, E. Rotenberg, and J. E. Smith, "Path-based next trace prediction," in *Proceedings of 30th Annual International Symposium on Microarchitecture*. IEEE, 1997, pp. 14–23.
[60] A. Jain and C. Lin, "Back to the future: leveraging belady's algorithm for improved cache replacement," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2016, pp. 78–89.
[61] A. Jain and C. Lin, "Rethinking belady's algorithm to accommodate prefetching," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2018, pp. 110–123.
[62] 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.
[63] S. Jamilan, T. A. Khan, G. Ayers, B. Kasikci, and H. Litz, "Apt-get: Profile-guided timely software prefetching," in *Proceedings of the Seventeenth European Conference on Computer Systems*, 2022, pp. 747–764.

[64] D. A. Jiménez, "Insertion and promotion for tree-based pseudolru last-level caches," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, 2013, pp. 284–296.

[65] D. A. Jiménez, S. W. Keckler, and C. Lin, "The impact of delay on the design of branch predictors," in *Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, 2000, pp. 67–76.

[66] D. A. Jiménez and E. Teran, "Multiperspective reuse prediction," in *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2017, pp. 436–448.

[67] 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.

[68] R. Karedla, J. S. Love, and B. G. Wherry, "Caching strategies to improve disk system performance," *Computer*, vol. 27, no. 3, pp. 38–46, 1994.

[69] 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.

[70] 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.

[71] 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.

[72] C. Kaynak, B. Grot, and B. Falsafi, "Shift: Shared history instruction fetch for lean-core server processors," in *International Symposium on Microarchitecture*, 2013.

[73] C. Kaynak, B. Grot, and B. Falsafi, "Confluence: unified instruction supply for scale-out servers," in *Proceedings of the 48th International Symposium on Microarchitecture*, 2015, pp. 166–177.

[74] S. M. Khan, Y. Tian, and D. A. Jimenez, "Sampling dead block prediction for last-level caches," in *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 2010, pp. 175–186.

[75] 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.

[76] 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 *15th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 21)*, 2021, pp. 163–181.

[77] 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.

[78] 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.

[79] 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.

[80] M. Kharbutli and Y. Solihin, "Counter-based cache replacement algorithms," in *2005 International Conference on Computer Design*. IEEE, 2005, pp. 61–68.

[81] R. Kobayashi, Y. Yamada, H. Ando, and T. Shimada, "A cost-effective branch target buffer with a two-level table organization," in *Proceedings of the 2nd International Symposium of Low-Power and High-Speed Chips (COOL Chips II)*, 1999.

[82] A. Kolli, A. Saidi, and T. F. Wenisch, "Rdip: return-address-stack directed instruction prefetching," in *2013 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2013, pp. 260–271.

[83] 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.

[84] 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.

[85] 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.

[86] 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.

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

[88] D. Lee, J. Choi, J.-H. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim, "On the existence of a spectrum of policies that subsumes the least recently used (lru) and least frequently used (lfu) policies," in *Proceedings of the 1999 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, 1999, pp. 134–143.

[89] 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.

[90] L. Li, D. Tong, Z. Xie, J. Lu, and X. Cheng, "Optimal bypass monitor for high performance last-level caches," in *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, 2012, pp. 315–324.

[91] H. Litz, G. Ayers, and P. Ranganathan, "CRISP: critical slice prefetching," in *ASPLOS '22: 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, 28 February 2022 - 4 March 2022*, B. Falsafi, M. Ferdman, S. Lu, and T. F. Wenisch, Eds. ACM, 2022, pp. 300–313. [Online]. Available: https://doi.org/10.1145/3503222.3507745

[92] E. Z. Liu, M. Hashemi, K. Swersky, P. Ranganathan, and J. Ahn, "An imitation learning approach for cache replacement," *arXiv preprint arXiv:2006.16239*, 2020.

[93] H. Liu, M. Ferdman, J. Huh, and D. Burger, "Cache bursts: A new approach for eliminating dead blocks and increasing cache efficiency," in *2008 41st IEEE/ACM International Symposium on Microarchitecture*. IEEE, 2008, pp. 222–233.

[94] 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.

[95] 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.

[96] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger, "Evaluation techniques for storage hierarchies," *IBM Systems journal*, vol. 9, no. 2, pp. 78–117, 1970.

[97] C. Mazumdar, P. Mitra, and A. Basu, "Dead page and dead block predictors: Cleaning tlbs and caches together," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2021, pp. 507–519.

[98] P. Michaud, "Some mathematical facts about optimal cache replacement," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 13, no. 4, pp. 1–19, 2016.

[99] P. Michaud, "Pips: Prefetching instructions with probabilistic scouts," in *The 1st Instruction Prefetching Championship*, 2020.

[100] 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.

[101] T. Nakamura, T. Koizumi, Y. Degawa, H. Irie, S. Sakai, and R. Shioya, "D-jolt: Distant jolt prefetcher."

[102] E. J. O'neil, P. E. O'neil, and G. Weikum, "The lru-k page replacement algorithm for database disk buffering," *Acm Sigmod Record*, vol. 22, no. 2, pp. 297–306, 1993.

[103] 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.

[104] 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, pp. 340–350.

[105] G. Ottoni and B. Maher, "Optimizing function placement for large-scale data-center applications," in *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2017, pp. 233–244.

[106] 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.

[107] 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.

[108] R. Panda, P. V. Gratz, and D. A. Jiménez, "B-fetch: Branch prediction directed prefetching for in-order processors," *IEEE Computer Architecture Letters*, vol. 11, no. 2, pp. 41–44, 2011.

[109] 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.

[110] C. H. Perleberg and A. J. Smith, "Branch target buffer design and optimization," *IEEE transactions on computers*, vol. 42, no. 4, pp. 396–412, 1993.

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

[112] E. Petrank and D. Rawitz, "The hardness of cache conscious data placement," in *POPL*, 2002.

[113] 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.

[114] 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.

[115] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer, "Adaptive insertion policies for high performance caching," *ACM SIGARCH Computer Architecture News*, vol. 35, no. 2, pp. 381–391, 2007.

[116] M. K. Qureshi, D. N. Lynch, O. Mutlu, and Y. N. Patt, "A case for mlp-aware cache replacement," in *33rd International Symposium on Computer Architecture (ISCA'06)*. IEEE, 2006, pp. 167–178.

[117] A. Ramirez, L. A. Barroso, K. Gharachorloo, R. Cohn, J. Larriba-Pey, P. G. Lowney, and M. Valero, "Code layout optimizations for transaction processing workloads," *ACM SIGARCH Computer Architecture News*, 2001.

[118] 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.

[119] 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.

[120] A. Ros and A. Jimborean, "The entangling instruction prefetcher," *IEEE Computer Architecture Letters*, vol. 19, no. 2, pp. 84–87, 2020.

[121] A. Ros and A. Jimborean, "A cost-effective entangling prefetcher for instructions," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2021, pp. 99–111.

[122] E. Rotenberg, S. Bennett, and J. E. Smith, "Trace cache: a low latency approach to high bandwidth instruction fetching," in *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture. MICRO 29*. IEEE, 1996, pp. 24–34.

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

[124] D. Seal, *ARM architecture reference manual*. Pearson Education, 2001.

[125] V. Seshadri, O. Mutlu, M. A. Kozuch, and T. C. Mowry, "The evicted-address filter: A unified mechanism to address both cache pollution and thrashing," in *2012 21st International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 2012, pp. 355–366.

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

[127] A. Seznec, "The fnl+ mma instruction cache prefetcher," in *IPC-1-First Instruction Prefetching Championship*, 2020.

[128] S. Seznec, "Don't use the page number, but a pointer to it," in *23rd Annual International Symposium on Computer Architecture (ISCA'96)*. IEEE, 1996, pp. 104–104.

[129] Z. Shi, X. Huang, A. Jain, and C. Lin, "Applying deep learning to the cache replacement problem," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 413–425.

[130] Y. Smaragdakis, S. Kaplan, and P. Wilson, "Eelru: simple and effective adaptive page replacement," *ACM SIGMETRICS Performance Evaluation Review*, vol. 27, no. 1, pp. 122–133, 1999.

[131] A. J. Smith, "Sequential program prefetching in memory hierarchies," *Computer*, no. 12, pp. 7–21, 1978.

[132] 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.

[133] 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.

[134] R. Subramanian, Y. Smaragdakis, and G. H. Loh, "Adaptive caches: Effective shaping of cache behavior to workloads," in *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*. IEEE, 2006, pp. 385–396.

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

[136] M. Takagi and K. Hiraki, "Inter-reference gap distribution replacement: an improved replacement algorithm for set-associative caches," in *Proceedings of the 18th annual international conference on Supercomputing*, 2004, pp. 20–30.

[137] E. Teran, Z. Wang, and D. A. Jiménez, "Perceptron learning for reuse prediction," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2016, pp. 1–12.

[138] 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.

[139] T. F. Wenisch, M. Ferdman, A. Ailamaki, B. Falsafi, and A. Moshovos, "Temporal streams in commercial server applications," in *2008 IEEE International Symposium on Workload Characterization*. IEEE, 2008, pp. 99–108.

[140] T. F. Wenisch, M. Ferdman, A. Ailamaki, B. Falsafi, and A. Moshovos, "Practical off-chip meta-data for temporal memory streaming," in *2009 IEEE 15th International Symposium on High Performance Computer Architecture*. IEEE, 2009, pp. 79–90.

[141] T. F. Wenisch, S. Somogyi, N. Hardavellas, J. Kim, A. Ailamaki, and B. Falsafi, "Temporal streaming of shared memory," in *32nd International Symposium on Computer Architecture (ISCA'05)*. IEEE, 2005, pp. 222–233.

[142] Wikipedia contributors, "Drupal — Wikipedia, the free encyclopedia," https://en.wikipedia.org/w/index.php?title=Drupal&oldid=989582664, 2020, [Online; accessed 23-November-2020].

[143] Wikipedia contributors, "Mediawiki — Wikipedia, the free encyclopedia," https://en.wikipedia.org/w/index.php?title=MediaWiki&oldid=989993176, 2020, [Online; accessed 23-November-2020].

[144] Wikipedia contributors, "Wordpress — Wikipedia, the free encyclopedia," https://en.wikipedia.org/w/index.php?title=WordPress&oldid=977243718, 2020, [Online; accessed 23-November-2020].

[145] Wikipedia contributors, "Cross-validation (statistics) — Wikipedia, the free encyclopedia," https://en.wikipedia.org/w/index.php?title=Cross-validation_(statistics)&oldid=1055904460, 2021, [Online; accessed 24-November-2021].

[146] Wikipedia contributors, "Mysql — Wikipedia, the free encyclopedia," https://en.wikipedia.org/w/index.php?title=MySQL&oldid=1054628857, 2021, [Online; accessed 19-November-2021].

[147] C.-J. Wu, A. Jaleel, W. Hasenplaugh, M. Martonosi, S. C. Steely Jr, and J. Emer, "Ship: Signature-based hit predictor for high performance caching," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, 2011, pp. 430–441.

[148] T.-Y. Yeh and Y. N. Patt, "A comprehensive instruction fetch mechanism for a processor supporting speculative execution," *ACM SIGMICRO Newsletter*, vol. 23, no. 1-2, pp. 129–139, 1992.

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

[150] 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.

[151] 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.