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.

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
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 5th Championship Branch Prediction (CBP-5) [15], and (3) 50 traces from 1st 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
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% of all pipeline slots due to frontend stalls [25, 27, 67, 133].

### Table 1: Simulation parameters

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

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
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 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.
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, \( \bar{a} \), which experience holistic variance defined as follows:

\[
\text{Holistic variance} = \frac{1}{n-1} \sum_{i=2}^{n} (a_i - \bar{a})^2.
\]

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.

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

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:

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

where \( y_1 \) and \( y_2 \) are two empirically decided thresholds such that \( 0 \leq y_1 \leq y_2 \leq 1 \). 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...
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.

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.


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 (IP92%) and not-taken (11%) 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

Figure 10: High level design of Thermometer
which is finding the coldest temperature. For a 4-way BTB, this is the coldest way as, $A < B$ 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 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., $A < B = (B_1 & A_1) | (B_0 & B_1 & A_0)$ 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. \hspace{1cm} $y_i \leftarrow$ temperature of $x_i$
3. \hspace{1cm} $t \leftarrow \min(y_0, y_1, y_2, \ldots, y_n)$ \hspace{1cm} \(\text{Find the coldest temperature}\)
4. \hspace{1cm} $S \leftarrow \{x_j : y_j = t\}$
5. \hspace{1cm} if $x_0 \in S$ \&\& $|S| = 1$ then
6. \hspace{2cm} return $x_0$ \hspace{1cm} \(\text{Bypass}\)
7. \hspace{1cm} $z \leftarrow$ 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, 135]. 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 page), 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 for the test execution and uses different inputs. Apart from evaluating Thermometer on these 13 real-world applications, we also...

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 temporarily 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 \times (entry size + 2 bits overhead) = 8192 \times 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 shown in Fig. 15, Thermometer achieves an average coverage of 64.4%.

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
Thermometer achieves an average speedup of 8.7% that is 83.6% of the average speedup provided by the optimal BTB replacement policy.

Figure 11: Thermometer’s IPC speedup compared to optimal and state-of-the-art replacement policies 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.

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.

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 8 to 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),
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 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.
In addition to storing branch targets, BTB entries generally contain a tag and a prediction 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


