2025 IEEE International Symposium on High Performance Computer Architecture (HPCA) | 979-8-3315-0647-6/25/$31.00 ©2025 IEEE | DOI: 10.1109/HPCA61900.2025.00060

From Optimal to Practical: Efficient Micro-op Cache Replacement Policies for Data Center Applications

Kan Zhu[1*], Yilong Zhao[2*], Yufei Gao[1,3], Peter Braun[4], Tanvir Ahmed Khan[5], Heiner Litz[4], Baris Kasikci[1], Shuwen Deng[3†]
University of Washington[1], UC Berkeley[2], Department of Electronic Engineering, Tsinghua University[3], UC Santa Cruz[4], Columbia University[5]
{kanzhu,yufeig21,baris}@cs.washington.edu, yilongzhao@berkeley.edu, {pvbraun,hlitz}@ucsc.edu, tk3070@columbia.edu, shuwend@tsinghua.edu.cn

***Abstract*—Optimizing the CPU frontend has become crucial for modern processors with intricate instruction decoding logic, especially for efficiently running planet-scale data center applications. Micro-operation (micro-op) cache is a key unit to help improve the energy efficiency of the CPU frontend. Unfortunately, we find that data center applications suffer from frequent micro-op cache misses due to the lack of an effective micro-op cache replacement policy. Developing micro-op cache-specific replacement policies is challenging, as there currently does not exist an optimal theoretical solution akin to Belady's algorithm for conventional caches. As a result, it is unknown by how much replacement policies can be improved and how to get there.**

To address these challenges, we introduce FLACK, a new near-optimal offline policy that considers the key features of the micro-op cache, such as variable and disproportional costs of micro-op cache misses and partial hits. We show that FLACK substantially outperforms Belady's algorithm, thus establishing a new baseline for micro-op cache replacement policies. We then design FURBYS, a practical policy that mimics FLACK via profile-guided methods. FURBYS has three key components to perform cache replacement decisions: (1) it uses profiles of the whole-execution hit/miss behavior, (2) it detects locally (transiently) hot data, and (3) it selectively ignores data with profiled low hit rates. We evaluate FLACK and FURBYS using 11 data center applications and find that FLACK demonstrates an average bound of 30.21% miss reduction, achieving 4.46% greater miss reduction than Belady's algorithm. Our practical policy, FURBYS, provides 14.34% average miss reduction compared to LRU, which is $1.84\times$ greater than the current state-of-the-art replacement policy, contributing to 3.10% of performance-per-watt improvement for the CPU core. On average, in terms of miss reduction and IPC gain, FURBYS is equivalent to LRU policy on $1.5\times$ micro-op cache sizes (up to $2\times$), demonstrating the effectiveness of the proposed replacement policy.

I. Introduction

Data centers are responsible for storing, managing, and distributing a significant portion of the world's data, resulting in notable energy demand and substantial carbon and water footprints. Recent data indicates that between 196 to 400 terawatt-hours (TWh) of energy were consumed by data centers in 2020, corresponding to 1% to 2% of the total annual energy consumption. As the data center industry expands, this energy consumption is expected to increase [13], [31], [75].

Given the necessity to significantly improve the energy efficiency (performance-per-watt, the number of instructions executed per Joule of energy) of data center applications, optimizing the CPU frontend has become a critical endeavor, particularly for complex instruction set architectures (ISA). ISAs such as x86 utilize variable length instructions that require complex, deeply-pipelined decoding logic [19]–[21], [51], [53], [58], [59], [83] which consumes significant power. To address this challenge, Intel and AMD have deployed micro-op caches, storing decoded instructions (micro-ops) to reduce power consumption and improve performance. In particular, micro-op caches enable clock-gating of the decoders and L1 instruction cache (icache) to save power while providing a higher instruction fetch bandwidth [7], [81].

In Section III we show that an optimal micro-op cache can provide an average performance-per-watt gain of 7.41%, exceeding the potential benefits of other microarchitectural structures such as the branch target buffer, the branch predictor or even the icache. Furthermore, we find that due to the large code footprint, more than 99% of micro-op cache misses are due to insufficient capacity or set conflicts. The tight capacity constraints call for a good replacement policy to keep micro-ops in the cache selectively for higher hit rate.

To determine how the frontend's performance-per-watt can be increased, we investigate the existing state-of-the-art micro-op cache replacement policies in Section III. We find that existing policies (including Belady [22]) are inefficient as they ignore the following key features of micro-op caches:

- **Disproportionate cache miss costs.** The micro-op cache sends micro-ops to the processor at the granularity of a Prediction Window (PW), while internally, it uses fixed-sized entries as a storage unit. As a result, PWs can consume multiple entries, while the last entry of a PW is generally only partially filled. All entries of a PW must be kept or evicted as a whole. As a result, the replacement policy needs to consider both the value of a PW (number

*The authors contributed equally to this work.
†Shuwen Deng is the corresponding author.

979-8-3315-0647-6/25/$31.00 ©2025 IEEE
DOI 10.1109/HPCA61900.2025.00060

of micro-ops) as well as the size of the PW (number of entries) to minimize the miss cost.

- **Partial hits.** As PWs terminate on predicted *taken* branches, multiple partially overlapping PWs can exist if a terminal conditional branch is sometimes taken and sometimes not. As PWs are looked up by their start address, retrieved PWs may sometimes be "incomplete" or only partially used. Such overlapping PWs need to be considered by the replacement policy.
- **Asynchronous lookup and insertion.** The lookup and insertion operations to micro-op cache may complete out-of-order. For instance, one access a may miss in the micro-op cache, causing an insertion via the decode pipeline to happen several cycles later, while before the insertion, access b may lookup the micro-op cache, causing asynchronous lookup and insertion that is not supported by existing variable-cost offline optimal replacement policies.

An optimal micro-op cache replacement policy needs to consider the properties above that prior works have not considered so far. Even if impossible to implement in hardware, developing such a policy is crucial, as it provides an upper bound for performance and can guide the development of new online policies. We devise FLACK, a novel policy that approximates optimal replacement decisions for the micro-op cache. It is only near-optimal, as prior works [41] have shown that determining optimal replacement decisions in caches with variable-cost and variable-size entries is an NP-complete problem. Yet we show that by considering the three micro-op cache properties above, FLACK can substantially outperform policies such as Belady, approximating optimal replacement for the micro-op cache. Our proposed FLACK policy (1) addresses disproportionate cache miss costs, by considering the number of cache entries and the number of micro-ops for each PW, (2) manages partial PW hits by inserting and keeping larger PWs in the cache while throttling bypassing to increase the likelihood of partial hits when sufficient micro-op cache space is available, (3) mitigates asynchronous lookups and insertions by late evictions to maintain the micro-op cache entries until the insertion completes.

While FLACK executes in polynomial time and hence can be used to simulate near-optimal replacement decisions in an acceptable time, it is infeasible to implement it in hardware. Based on the insights obtained from FLACK, we then design a new practical replacement policy, FURBYS, which outperforms all prior state-of-the-art replacement policies including SRRIP [45], SHiP++ [93], GHRP [66], Mockingjay [77] and Thermometer [82]. Our practical replacement policy reduces micro-op cache misses and bypasses micro-op cache insertions to increase performance-per-watt. FURBYS uses whole-execution micro-op cache profiles obtained using FLACK to group PWs by hit rate. Furthermore, FURBYS introduces support for evicting PWs with a high average whole-execution hit rate but a low temporary hit rate. Finally, FURBYS also selectively bypasses PWs with low hit rates to prevent cold

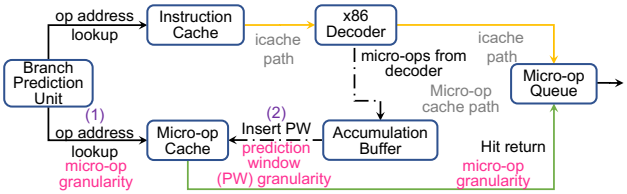

Fig. 1. A typical x86 frontend. The micro-op cache performs lookup (1) and insertion (2) asynchronously and needs to accommodate different data units, including micro-ops and PWs. Since the output of the micro-op cache is a stream of micro-ops, we define micro-op cache misses in terms of micro-ops.

micro-ops from consuming micro-op cache capacity [26] and reduces energy consumption.

We evaluate FLACK and FURBYS in the context of 11 data center applications to demonstrate their effectiveness. In particular, we show that FLACK achieves an average miss reduction of 30.21% over the baseline least-recently-used (LRU) replacement policy, outperforming even Belady's "optimal" policy by 4.46%. Our online policy, FURBYS, achieves an average miss reduction of 14.34% and 3.10% performance-per-watt gain. To reach the same miss reduction and performance of FURBYS, the LRU policy requires a cache with almost twice the capacity for some applications. On average, FURBYS provides the equivalent of a 50% micro-op cache size increase while improving power efficiency substantially. We cross-validate the usefulness of profiling information with different inputs, showing its robustness. We also conduct sensitivity analyses to investigate the impact of configuration changes on FLACK and FURBYS.

In summary, we make the following contributions:

- We demonstrate the importance of the micro-op cache and the necessity of optimizing the micro-op cache's replacement policy.
- We identify the shortcomings of state-of-the-art optimal replacement policies such as Belady and flow-based offline optimal (FOO) policies by providing a comprehensive study of essential features of the micro-op cache.
- We devise FLACK, a new offline replacement policy for approximating optimal behavior for the micro-op cache by considering micro-op cache-specific properties.
- We propose FURBYS, the first online replacement policy that is tailored for the micro-op cache.

II. Micro-op Cache Background

A. Overall Micro-op Cache Architecture

As demonstrated in Figure 1, a typical x86 processor frontend contains the original legacy decode pipeline (icache path), fetching CISC instructions from the icache and decoding them into RISC-like micro-ops. The micro-ops are then fed into the micro-op queue from where they can be provided to the backend. As fetching and decoding instructions is complex and power-consuming in variable-length ISAs such as x86 [65], modern processors store decoded micro-ops in a micro-op

cache. Our model of micro-op cache is strictly included by the L1 icache, i.e., every icache eviction will trigger the eviction of corresponding items in the micro-op cache, following the popular industry paradigms [7], [28], [79], [87]. Retrieving micro-ops from the micro-op cache can improve performance due to the lower latency and higher bandwidth, but more importantly, it allows power down the decode pipeline and the icache, leading to significant power savings [57]. Note that due to these properties, even RISC-based architectures such as ARM have introduced micro-op caches [5].

B. Lookup and Insertion

To populate the micro-op cache, the legacy decode pipeline inserts decoded micro-ops into the accumulation buffer, forming the content of a prediction window (PW), the basic granularity of micro-op cache operations. A PW is similar to a basic block in that it is terminated by a taken branch, however, PWs are also terminated by the last instruction of a cache line. After accumulation, the PW is inserted into the micro-op cache and indexed by the starting address [57], [85]. If the same PW is accessed later again, which frequently occurs in loops, the system will switch to the micro-op cache for providing the micro-ops. In this case, the decode pipeline and the icache can be turned off via clock-gating, leading to substantial energy savings. The switch can only happen at predicted taken branches or icache cache block boundaries [7], [85]. However, when a micro-op cache miss occurs, the frontend needs to switch back to the legacy pipeline introducing a one-cycle overhead [7], [8]. At this point, the next PWs are fetched from the icache, and then decoded, causing a delay of several cycles due to the deep decoder pipeline. It is possible that while an earlier miss is still in the decode pipeline, later addresses lookup (and hit) the micro-op cache. This introduces the *asynchronous lookup and insertion* issue described in the previous section, causing out-of-order insertion and lookup.

C. Micro-op Cache Entry Organization

The micro-op cache stores a fixed number of micro-ops per line. However, as the length of a PW is determined by icache cache line boundaries and predicted taken branches, PWs have a variable number of micro-ops [56]. If the number of micro-ops within a PW is smaller than the maximum number of micro-ops per line, some capacity of the micro-op cache is left unused. As the energy cost of looking up an entry in the micro-op cache is unaffected by the number of valid micro-ops in an entry, it is beneficial to preferably store fully populated entries in the micro-op cache as the energy consumption of the decode pipeline scales with the number of decoded micro-ops. Therefore, we define the number of micro-ops of a PW as its **cost** and accordingly, define the miss rate at the micro-op level to reflect the performance-per-watt. The replacement policy needs to consider this variable cost of a PW.

If the number of micro-ops of a PW is greater than the number of micro-ops per entry, a PW needs to be distributed across multiple micro-op cache entries. Therefore, the micro-op cache reserves multiple entries within the same cache set for a single PW and then tags those entries so that they are fetched together on a lookup (during multiple cycles) or evicted as a whole [57]. Both these aspects need to be considered by the replacement policy. We define the number of entries occupied by a PW as its **size**. The replacement policy needs to consider both the size and cost of PWs to achieve the best performance-per-watt.

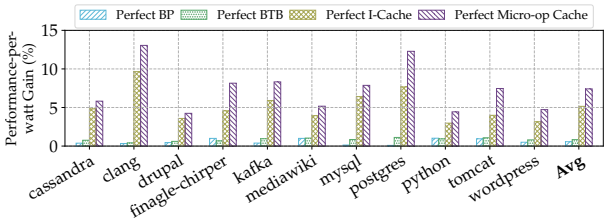

Fig. 2. Per core performance-per-watt gain compared with LRU baseline when using perfect structures. A perfect micro-op cache provides 7.41% performance-per-watt benefit.

D. Overlapping Prediction Windows

By determining predicted taken branches, modern decoupled frontend forms PW boundaries [73], where a PW starts with the target of a control flow instruction and ends with a control flow changing instruction or the boundary of an I-cache line [57]. Predicted non-taken branches are considered sequential instructions and hence do not terminate a PW, introducing the *partial hit* issue. In particular, multiple PW lookups may exist with the same start address (which is used to index into the micro-op cache) but different sizes, resulting in partially overlapping PWs. According to AMD [85], when a larger PW is stored in the micro-op cache its entries may contain multiple ”intermediate exit points” referring to smaller PWs within the large one. As a result, on a lookup of a smaller PW (same start address and smaller size), the micro-op cache is capable of serving partial PWs from the large PW. This behavior needs to be considered by the replacement policy. There exists a second scenario in which the micro-op cache stores only a smaller PW, and a larger PW is being looked up. In this case, [85] states that only a partial PW is served from the micro-op cache and then the frontend switches to the legacy decode path to obtain the remaining micro-ops and form a new PW to be inserted. This scenario must be considered, and newly constructed windows must be handled accordingly.

III. Challenges of Optimizing Micro-op Cache Replacement

In this section, we first motivate the importance of micro-op cache replacement policies for data center applications and then analyze the root cause of why existing replacement policies fall short. We identify the key challenges of deriving efficient micro-op cache replacement policies, which, to the best of our knowledge, have been ignored by prior works.

A. Why is the micro-op cache important for data center applications?

As fetching and decoding instructions become more and more energy-consuming in modern processors, the micro-op cache plays an important role in improving the performance-per-watt of data center applications [81]. To quantify it, we use McPat [61], which is widely used for power modeling [33], [70], to measure the energy consumed per core by running 100M instructions for data center applications as described in Table II. For each experiment, we change the configuration of a single structure to be perfect (always hit). The result is shown in Figure 2. The perfect micro-op cache provides the largest energy efficiency improvement.

B. Why is the replacement policy of micro-op cache important for data center applications?

Like other caches, when inserting to micro-op cache, a *replacement policy* must be consulted for selecting an eviction victim. While micro-op replacement policies have not been thoroughly investigated in prior research, our results demonstrate that the replacement policy has a significant impact on performance due to the large instruction footprint of the data center applications. We evaluate three different types of misses, including cold, capacity, and conflict miss, of an 8-way 512-set micro-op cache under LRU and a near-optimal offline replacement policy for 11 data center applications. With LRU replacement policy, only 0.89% of total misses are cold misses, while 88.31% of total misses are capacity misses, and the remaining 10.8% are conflict misses. In contrast, with a near-optimal offline policy, the capacity and conflict misses can be reduced by 23.9% and 31.6% respectively, which contributes to 24.5% miss reduction in total. This result demonstrates that large code footprints put high pressure on the limited capacity of the micro-op cache, motivating a high-performance replacement policy to selectively keep the valuable objects, minimizing conflict misses and maximizing the hit rate.

C. Why is Belady's Algorithm Sub-Optimal for the micro-op cache?

Despite the importance of the replacement policy for the micro-op cache, existing offline optimal replacement policies, which were designed for normal caches, do not perform well in the micro-op cache. We first show why Belady's Algorithm [22], which is generally considered the optimal cache replacement policy, is sub-optimal for the micro-op cache due to its key characteristics.

(1) Variable disproportional costs. The micro-op cache has variable-size PWs, which can span multiple cache entries. As a result, the insertion of a multi-entry PW needs to evict potentially multiple other PWs, which consist of different number of micro-ops. We represent these properties by defining a PW's *size* as the number of entries occupied and the PW's *cost* as the total number of its micro-ops. However, Belady assumes every object has the same size and cost, causing non-optimal replacement decisions in the case of the micro-op cache.

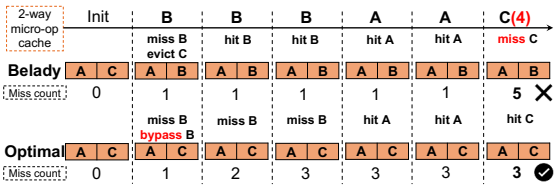

Fig. 3. Example of Belady's suboptimal replacement decision due to variable disproportional costs. Belady does not consider the cost difference between same-size PWs and fails to make the optimal decision.

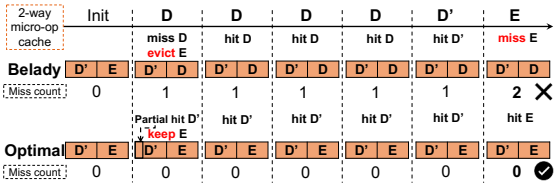

Fig. 4. Example of Belady's suboptimal replacement decision due to partial hits and generation of new PWs. In this case, D' and D are considered to be irrelevant by Belady, which leads to a deviation from the optimal result.

Figure 3 provides an example that illustrates this case. The cache has a capacity of 2 and initially contains two PWs, A (size=1, cost=1) and C (size=1, cost=4). If the lookup sequence is $BBBAAC$ with PW B (size=1, cost=1), Belady's algorithm chooses to evict C, as all PWs are treated equally. However, since costs are different in our case, bypassing B and evicting A is the optimal solution, with a cost of only 3 instead of 5.

(2) Partial hits and generation of new PWs. As PWs are terminated on a taken branch, there may exist two PWs with the same start address but different lengths causing the partial lookup issue. Belady's algorithm cannot handle such cases. Despite their tight relationship, Belady's algorithm considers PWs with the same start address but different sizes as entirely separate windows, which can lead to inefficiencies. For instance, as is demonstrated in Figure 4, consider three PWs: D' (size=1, cost=3) containing ($op1$, $op2$, $op3$), D (size=1, cost=1) containing only $op1$, and E (size=1, cost=1) containing $op4$, with a cache capacity of 2 initially holding D' and E. If the access sequence is D, D, D, D, D', E, Belady will consider D and D' as distinct and insert D to replace E. However, keeping D' and E in the cache would be more efficient since D' already allows D to hit, resulting in 0 misses. This renders Belady's decisions sub-optimal.

(3) Asynchronous lookup and insertion. As we discussed in subsection II-B, the micro-op cache is a non-blocking cache, that suffers from the asynchronous lookup and insertion issue. This asynchrony causes inaccuracies in Belady's algorithm. Consider three PWs, A (size=1, cost=1), B (size=1, cost=1), and C (size=1, cost=4), with a cache size of 2 initially containing A and B. If we assume that lookup and insertion are simultaneous (as naive Belady's algorithm assumes), and we have an access trace of $CCBAC$, Belady decides to evict

A to cache C, leading to a cost of 5. However, if the insertion of C takes exactly three intervals right before A is looked up, it is better to evict A after accessing it to achieve a lower cost of 4. Belady must be modified to make decision at insertion time to address the problem. However, we will demonstrate that combined with variable cost and partial hit, asynchrony cannot be easily dealt with in Section III-D.

D. Flow-based Offline Optimal (FOO) to the Rescue?

In the previous section, we showed that Belady's algorithm performs sub-optimal replacement decisions for the micro-op cache, which motivates us to explore better alternatives. Unfortunately, prior work [41] has shown that performing optimal replacement in the presence of variable-sized data elements is an NP-complete problem and hence cannot be solved optimally in a practicable amount of time. To address this issue, state-of-the-art algorithms leverage Flow-based Offline Optimal [23] (FOO) mechanisms to approximate optimal replacement with reasonable time complexity.

FOO is designed based on the observation that an optimal replacement decision for a particular PW remains unchanged between consecutive accesses. With FOO, the cache replacement policy is transformed into a decision-making problem determining whether to keep or evict a PW after each lookup, subject to the constraint of the cache's capacity.

For a PW, the FOO model offers two optimization goals. Object-Hit-Ratio (OHR) minimizes the total number of missed objects (PWs), regardless of the size of the object. Byte-Hit-Ratio (BHR) optimizes the number of bytes missed. In our case, it is equivalent to minimizing the missed entries. Utilizing FOO, we can derive a near-optimal arrangement of which PW should be bypassed and which ones should be kept in the cache to achieve the optimization goal. This problem can be solved efficiently using min-cost-flow solvers [23] in $O(n^{\frac{3}{2}})$ time, where n is the length of the access sequence.

Although it has been shown that FOO successfully approximates optimal replacement decisions for caches with variable size entries, it does not address the challenges of asynchronous insertion or partial hits. Unlike Belady, which handles asynchrony by making decisions at insertion time, FOO cannot easily extend to this scenario. Belady's strategy involves greedily evicting the PW with the furthest future access. As a result, Belady can efficiently recompute future optimal decisions as needed when the insertion time is changed due to asynchrony. In contrast, FOO relies on a flow-based solution using a static lookup traces. This makes managing asynchronous insertions challenging, as FOO cannot efficiently recompute future decisions for every asynchronous insertion due to its long runtime.

Moreover, the partial hits can also be affected by different insertion times, which will change the future lookup and insertion traces, making it impossible to derive the lookup trace in advance. Therefore, the FOO falls short in terms of asynchrony and partial hits.

Besides, while FOO only optimizes towards two cost metrics (OHR and BHR), in the micro-op cache, the cost of a missed PW is not directly proportional to its size. In particular, the cost of replacing an entry depends on the number of micro-ops (1-8) it contains. This disproportionality in costs causes FOO to perform suboptimal replacement decisions. For instance, for the example in Figure 3, FOO will still incorrectly prioritize C for eviction as all PWs are treated equally.

In Section IV, we introduce FLACK, a replacement policy that extends FOO to consider all essential properties of the micro-op cache. We also evaluate its performance gain.

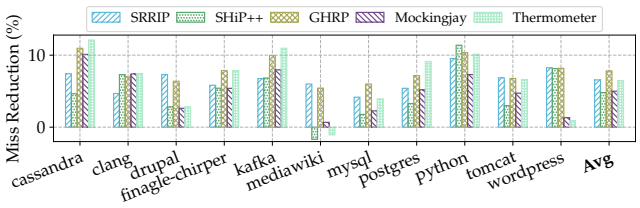

Fig. 5. Miss rate reduction of existing replacement policies. Existing policies do not provide considerable miss reductions over the LRU baseline. State-of-art policy GHRP only achieves 31.52% miss reduction compared to FLACK, which is a near-optimal policy we propose in Section IV.

E. Why do existing online replacement policies fall short?

In the previous sections, we showed that offline policies such as Belady's algorithm fail to produce optimal replacement decisions for the micro-op cache as they ignore important architectural properties. Existing online practical cache replacement policies also ignore the micro-op cache properties discussed in Section III-C and hence we expect them to perform sub-optimal replacement decisions as well. To quantify their effectiveness, We investigate the micro-op cache miss reduction of representative existing cache replacement policies including SRRIP [45], SHiP++ [93], Mockingjay [77], GHRP [66] and LRU. We also include Thermometer [82], a state-of-the-art profile-guided policy.

SRRIP [45] uses 2 bits per cache entry to store one of $2^2 = 4$ possible Re-reference Prediction Values (RRPV) to instruct the replacement. *SHiP++* [93] maintains a Signature History Counter Table (SHCT), which contains a counter (SHCTR) for each signature of each line (14-bit hash of the miss-causing PC) and a reuse (R) bit. *Mockingjay* [77] aims to simulate the optimal replacement policy [22] based on an access history. *GHRP* [66] uses the history of past instruction addresses and the reuse behaviors to predict dead blocks. *Thermometer* [82] uses the profiled results to divide PWs into hot, warm, and cold according to the PW hit rate.

Figure 5 shows the miss reduction over the LRU baseline for different replacement policies. All studied replacement policies show only limited improvements over LRU, a fraction of the 30.21% miss reduction achieved by our offline replacement policy proposed in Section IV.

Existing replacement policies fall short due to the following reasons. Firstly, the reuse distance for PW accesses is more scattered compared to other caches, such as the icache or BTB. For micro-op cache, over 20% of the PWs have a reuse

distance larger than 30, but only 10% and 2% lines have such a long reuse distance for icache and BTB. Therefore, the replacement policies designed for icache or BTB cannot be simply migrated to the micro-op cache. Second, SRRIP and SHiP++ aim to predict the re-reference interval for mimicking Belady's algorithm. GHRP tries to predict the object that will be accessed furthest in the future, which also follows Belady's decision. However, we have shown that Belady's algorithm is not the optimal replacement policy for micro-op cache. Both the cost and re-reference intervals should be considered to make the replacement decision, unlike the greedy method implemented by Belady's algorithm. Mockingjay follows the assumption that blocks accessed by the same PC will become Dead at the same time, however, in the case of the micro-op cache, this is not helpful since every PC is only associated with one PW. Therefore, Mockingjay must sample all the sets to achieve high accuracy causing a large space overhead. Thermometer groups PWs according to the average hit rate which captures holistic information, however, it lacks the mechanism to adjust to the transient pattern.

To overcome the limitations of existing replacement policies, we propose a novel profile-guided replacement policy in Section V, named FURBYS, based on the traces and design of our improved optimal replacement policy FLACK.

IV. FLACK: An offline Near-Optimal Replacement Policy for the micro-op cache

In Section III, we have shown that Belady fails to generate optimal replacement decisions in the presence of variable length cache entries. The FOO replacement algorithm mitigates this shortcoming, however, it can neither handle asynchronous lookups/insertions nor partial hits. We now describe the design of FLACK (FOO-based seLectively-bypassing Asynchronizing Cost-varying selective-data-Keeping), a new offline near-optimal replacement policy that considers all of the key characteristics of the micro-op cache.

Incorporate variable costs of PWs. FLACK extends the original FOO design to consider the variable amount of valid micro-ops per PW, respectively micro-op cache entry. Specifically, instead of assigning a cost of $1/size$ (OHR) or 1 (BHR) to each cache miss, we assign the unit cost as $cost/size$, where the cost refers to the number of micro-ops within a PW and size refers to the number of entries of the PW. As a result, FLACK not only approximates optimal cache replacement for a variable number of entries per PW but also for a variable number of micro-ops per entry.

Selective bypass to handle partial hits. An optimal replacement policy for the micro-op cache needs to consider partial hits. For instance, consider a large PW that contains a non-taken branch and hence incorporates two small PWs. The large PW can be either stored as a single PW yielding more compact storage (in terms of required entries) or as two separate PWs, enabling successful lookup (hit) for both PWs independently. In the original FOO, PWs are eagerly bypassed if not entirely used in the near future, which causes unnecessary misses if only a part of the PW is referenced. FLACK addresses this issue by keeping such bypassed PWs in the cache if space is available and it finds a partial hit in the near future, thus increasing the likelihood of partial hits while reducing misses. When the newly accumulated window and PWs in the cache have the same starting address, we will prioritize keeping the larger window.

Lazy eviction to mitigate asynchronous lookup and insertion. As discussed in Section III-C, FOO performs suboptimal replacement decisions in the presence of asynchronous lookup and insertion. FLACK improves FOO by 1) approximate insertion time eviction and 2) safeguarding late insertions. When FOO decides to evict one window already in the cache, FLACK will keep the window in the cache until another window is inserted. On the other hand, when FOO decides to evict one window that is not present in the cache due to the insertion delay, FLACK will use a buffer to queue this eviction and directly bypass the insertion to meet cache capacity constraints.

By combining the above techniques, FLACK closely approximates the optimal replacement policy. In Section VI-B, we will show that FLACK significantly outperforms existing optimal replacement policies, including Belady's algorithm. Therefore, we establish FLACK as a new upper-bound reference for micro-op cache replacement policies.

V. FURBYS: A practical replacement policy for the micro-op cache

While FLACK enables near-optimal replacement decisions in polynomial time, it is still infeasible to implement in real systems at runtime. We now present FURBYS (FLACK-based groUping-by-hit-Rate BYpassing-coldness detecting-miSses), a practical replacement policy for micro-op cache leveraging insights from our FLACK optimal replacement policy. At a high level, FURBYS works as follows. We first profile an application using Intel PT [1] to obtain a trace of executed PWs. We then feed this trace to the FLACK mechanism to generate a sequence of near-optimal replacement decisions. Based on this sequence, we count the hit rate of each PW and group the PWs into buckets according to the Jenks Natural Breaks [46] algorithm. Then, we tag every PW with its group information by injecting hints into the program binary via a compiler pass. Finally, we modify the micro-op cache replacement policy to consider such group hints. In particular, we prioritize PWs with a higher global average hit rate, and switch to SRRIP when we enter a phase in which some globally cold PWs are hot for a short period of time. We also bypass the PWs whose weight group is less than any other PWs in the cache. The detailed process is described below.

Group by whole-execution hit rates. FURBYS aims to mimic replacement decisions of FLACK by exploiting whole execution information obtained in the profiling step. In particular, we track the lookup sequence to calculate the hit rate of each address. Since FLACK provides near-optimal decisions considering all micro-op cache features (asynchronous lookup/insertion, disproportional size and cost, and partial hits), the hit rate of the PWs produced by FLACK's decision

sequence represents each PW's importance. Therefore, for each PW, we assign a static weight based on its average hit rate. By providing the micro-op cache with these weights, it can prioritize PWs with a high hit rate. We compute the weights via the Jenks Natural Breaks [46] algorithm. Jenks natural breaks determines the optimal arrangement of values into certain distinct classes or groups by minimizing within-class variance and maximizing between-class variance, which aligns with our goals. We assign weights 0-7 to the 8 clusters formed by the Jenks Natural Breaks, where 0 represents the PWs with the lowest hit rate. As replacement decisions are performed for each cache set individually, we compute the weights for PWs at set granularity.

Identify local miss pitfalls. We observe that some PWs achieve high hit rates in some program phases but low hit rates in others, reducing the effectiveness of measuring the average whole-execution hit rate and leading to non-optimal replacement decisions. For instance, consider a 4-way cache with the following lines in a target set (weight group in parentheses, higher weight means higher hit rate): A (1), B (20), C (20), D (10). Suppose PW I (2) is to be inserted. Based on the weight, PW A has the smallest weight and will be selected as the victim. However, if subsequent PWs are accessed in the format: $\{A, I\}^n (n > N)$ (N is a large natural number), this replacement decision can cause consecutive misses. To address this issue, we implement a local pitfall detector that detects whether a victim is repeatedly chosen in a set. If similar entries are evicted repeatedly, which indicates holistically hot PWs possibly become locally cold, FURBYS degrades to the SRRIP [45] replacement policy. With a normally maintained status, SRRIP is able to make replacements based on local information. After the locally cold PW is evicted, FURBYS takes over for the following replacement decisions. As shown in Section VI-C, this scenario occurs with an average percentage of 11.32%. Therefore, for most cases, FURBYS's decisions are more beneficial than those of SRRIP.

Selective bypass of PWs with low weights. Besides determining which PWs to evict, FURBYS also specifies PWs that should be bypassed instead of inserted into the micro-op cache. We bypass a new PW if its weight is lower than the minimum weight of PWs residing in the cache minus K, where K is a hyperparameter. We have empirically determined $K = 1$ to be best. This approach reduces the risk of polluting the cache. Moreover, bypassing PWs can greatly reduce the insertion energy, which contributes to performance-per-watt.

To summarize, our practical replacement policy, FURBYS, considers factors such as size, cost, partial hits, local effects, and bypass opportunities. We present evaluation details in Section VI, providing a comprehensive solution for designing an effective and practical replacement policy.

A. Procedure of FURBYS

Based on the analysis, we have developed FURBYS, a profile-guided micro-op cache replacement procedure whose steps are shown in Figure 6.

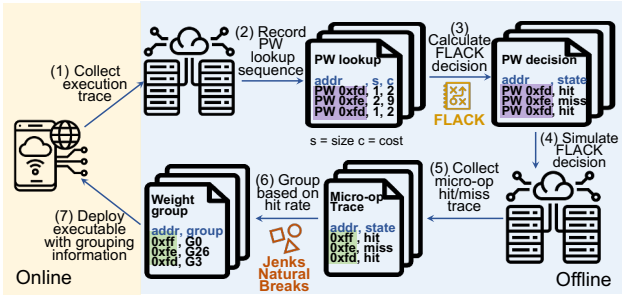

Fig. 6. FURBYS micro-op cache replacement procedure. STEP(1) collects the execution trace using Intel PT, STEP(2) records the PW lookup sequence, STEP(3) calculates the PW hit/miss decision using FLACK from the offline behavior simulator, STEP(4) refines the FLACK decision to the micro-op granularity in the Scarab simulator, STEP(5) collects micro-op hit/miss decisions, STEP(6) groups hit rates using Jenks natural breaks, and STEP(7) deploys the actual online evaluation with grouping information. Steps STEP(2)-(6) are performed offline, while STEP(1) and STEP(7) are online.

STEP(1): low-overhead execution trace collection. The execution trace is collected by Intel PT [1]. Similar to prior work [54], [82], Intel PT is chosen because of its low runtime overhead (only up to 1% [48]–[50], [94]) and general adoption in today's data centers [30], [35]. The Intel PT trace captures dynamically executed branch instructions, including branch decisions and branch targets. We can reproduce the micro-op sequence using the binary files. The trace serves as the input for FURBYS.

STEP(2): record PW lookup sequence. As shown in Figure 6, STEP(2) involves simulating the trace file with the configuration where the micro-op cache size is set to 0 to collect the PW lookup traces. With a micro-op cache size of 0, every lookup will result in a miss and will be accumulated and inserted (even though the insertion will ultimately fail). By observing the insertion behavior, we can learn the PW lookup sequence independently of replacement operations.

STEP(3)-(5): obtain FLACK decision. The trace is then executed on Scarab modeling a realistically sized micro-op cache to obtain hit/miss rate information for each micro-op. This information is stored in a file in STEP(5) for further use.

STEP(6): group micro-ops based on FLACK hit rate. We calculate the hit rate of each micro-op based on the Scarab simulation result obtained from FLACK. We then apply Jenks natural breaks to group the micro-ops according to their hit rate. We modify the binary file to insert the hint into instructions using the reserved bits. For every occurred PW, only one weight is needed. Since most PWs end with a branch or contain at least a branch, similar to Thermometer, we can use the reserved bits in branches [82] to inject such information. The micro-op cache observes these hints when accumulating PWs and sets their corresponding group according to the encountered hint.

STEP(7): deploy executable with grouping information. In the online application evaluation, the processor receives the grouping information through decoding. In the hardware, each

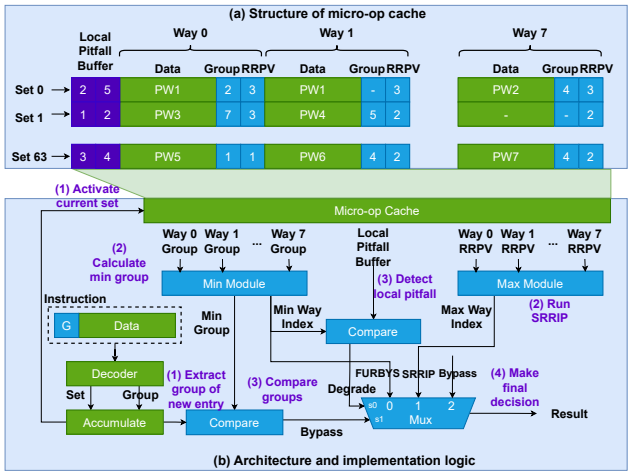

Fig. 7. (a) Structure of micro-op cache with newly added hardware units; (b) Architecture and implementation logic of FURBYS. Here green boxes refer to the original microarchitecture, and blue and purple boxes are the new microarchitectures or logic we added for FURBYS.

entry in the micro-op cache reserves 3 bits to keep track of the hit rate for each micro-op and compare for replacement. Additionally, a two-slot miss-pitfall detector is used for each cache set to store recently evicted ways. When we evict the same way twice, we will switch to SRRIP to evict hot PWs that are temporarily cold. After that, we switch back to normal FURBYS operations.

B. FURBYS architecture and replacement policy

When deploying the executable with grouping information in STEP(7), modifications to the decode unit, accumulation unit, and micro-op cache are needed to implement the FURBYS architecture. Figure 7 illustrates these modifications. The decoder is altered to extract group information from binary instructions. When a marked instruction is encountered, specific bits are interpreted as group information and sent along with the micro-ops to the accumulator. The accumulator retains the first group tag within the PW and, upon meeting the end of accumulation, passes the PW, along with the weight, to the micro-op cache for insertion. In the micro-op cache, structures need to be modified to add 3 additional bits to each entry to store the weight and 2 RRPV bits to store SRRIP metadata for each entry. When a PW is inserted, its weight is assigned during the copying of the PW content, and the RRPV bits are initialized to 2. Three additional bits for each cache set are added to record the most recently evicted way to accommodate the metadata for the local miss-pitfall detector.

When searching for a victim in the micro-op cache, the start address of the accumulated window triggers a set activation (step 1). The minimum weight in the set and its cache location is found using a $min\ module$ (step 2). The minimum weight is then compared with the pending PW to decide whether to bypass it (step 3). Meanwhile, the selected way is compared to the local miss-pitfall detector's records to determine whether to switch to SRRIP (SRRIP selection using $max\ module$ is done in step 2). A Mux is then used to select the final victim among the SRRIP victim, FURBYS victim, and bypassing (step 4). If we decide not to bypass the pending PW, the corresponding victim is evicted, and the new PW is inserted.[1]

VI. Evaluation

TABLE I
SIMULATION PARAMETERS

Parameter	Value
CPU	3.2GHz, 6-wide Out-of-Order, 256-entry Reorder Buffer, 96-entry Reservation Station
Decoder	4-wide decoder, 5-cycles latency
Branch Predictor	8192-entry 4-way BTB, 32-entry RAS, 64KB TAGE-SC-L, 4096-entry IBTB
Micro-op cache	512-entry, 8-way, 8 micro-ops per entry, inclusive with L1i, 1-cycle switch delay,
L1i	64B-line, 32KiB, 8-way, 1-cycle, LRU
L1d	64B-line, 32KiB, 8-way, 2-cycles, LRU
L2	64B-line, 512KiB, 8-way, 16-cycles, LRU
DRAM	8-GiB, 19.2-GiB/s, 100-cycles

A. Experimental methodology

Simulation parameters. We perform our evaluation utilizing a cycle-accurate and widely-used simulator, Scarab [14], [33], [62], [70]. We adjust simulation parameters to resemble the AMD Zen3 architecture, as listed in Table I. We implement several reference offline micro-op replacement policies including Belady's algorithm [22] and flow-based offline optimal (FOO) [23], as well as other existing online policies such as SRRIP [45], SHiP++ [93], GHRP [66] and Mockingjay [77]. For comparison, we also implement a state-of-the-art profile-guided replacement policy, Thermometer [82].

Data center applications. To perform analysis, we studied data center applications with large instruction footprints. It is critical to optimize the performance-per-watt of these applications due to their large planet-scale energy and carbon footprint. As we do not have access to proprietary data center workloads, we evaluate 11 widely-used, open-source data center applications shown in Table II. We collect traces of the applications using Intel PT [1] for trace-driven simulations and showcase their frontend statistic. We change 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`) to get multiple traces after warmup. We use one set of the traces to conduct the majority of the experiments on miss reduction and energy saving, while validates the adaptability of FURBYS using all of the traces in Section VI-C.

[1]Insertion is not on the critical path of a non-blocking cache which can process requests during insertion, thus easing timing constraints.

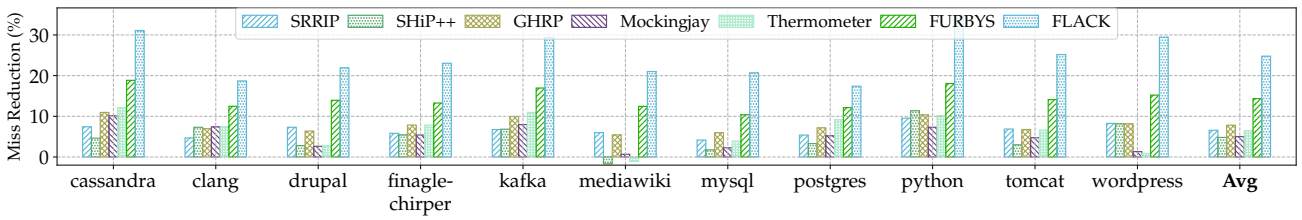

Fig. 8. Miss rate reductions of FURBYS compared with existing replacement policies. We significantly outperform the existing replacement policy on every application. FURBYS reaches 57.85% miss reduction of FLACK.

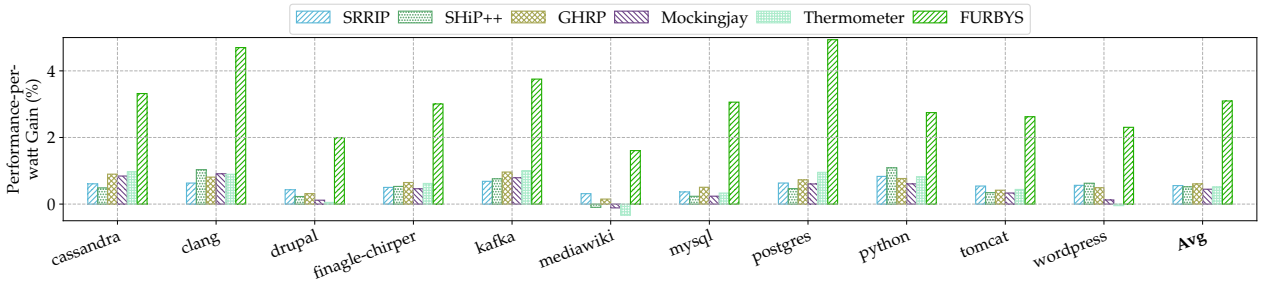

Fig. 9. Performance-per-watt gain of FURBYS. Our replacement policy increases 3.10% performance-per-watt.

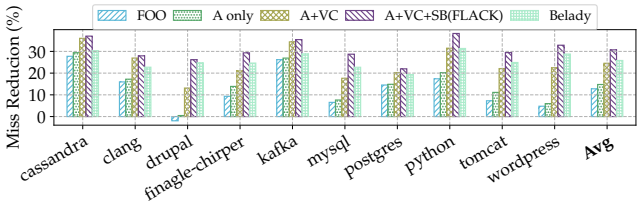

Fig. 10. Miss reductions of different features we added to FLACK compared with FOO and Belady's algorithm over LRU. A: asynchrony considerations, VC: variable cost, SB: selectively bypass. Every feature leads to considerable performance gain, and our approximation to optimum beats Belady's algorithm by 4.46% miss reduction on average.

TABLE II
LIST OF THE DATA CENTER APPLICATIONS WE STUDY

Applications	Description	Branch MPKI
Cassandra [2]	From the Java DaCapo benchmark suite [24]	1.78
Kafka [3]		1.77
Tomcat [4]		4.45
Drupal [88]	From Facebook's OSS performance benchmark suite [17]	1.89
Wordpress [90]		2.35
Mediawiki [89]		5.64
PostgreSQL [11]	Collected when used to serve pgbench [10] queries	0.41
MySQL [9]	Collected while serving TPC-C [29] queries	0.66
Python [16]	Collected while running the pyperformance [12] benchmark suite	4.73
Finagle-chirper [15]	Twitter's microblogging service	4.76
Clang [6]	Collected while building LLVM	1.86

B. FLACK: Near-optimal Replacement Policy

We conduct an ablation evaluation of cache miss reductions by applying the features step by step from our proposed optimization (FLACK) over the original FOO policy and compare the results of Belady's algorithm. To avoid the effect of icache eviction on the micro-op cache due to the cache inclusiveness, we evaluate them under perfect icache setups to show the difference. The result is shown in Figure 10. We observe that the original FOO implementation performs poorly in the micro-op cache scenario, with results even worse than LRU for some applications. After incorporating lazy eviction (A-only), variable cost (A+VC) and selective bypassing (A+VC+SB), our proposed FLACK significantly outperforms the original FOO policy and reduces 14.34% misses compared to LRU, which is 4.46% more effective than Belady's algorithm. We will demonstrate the effect of a better offline replacement policy on the FURBYS in the Section VI-D.

C. FURBYS: Practical Replacement Policy

Miss Reduction. We employ LRU as our baseline and compare the results of existing replacement policies with our FURBYS. The outcomes are in Figure 8. As depicted, our replacement policy reduces misses by an average of 14.34% compared to LRU. Among the existing policies, GHRP [66] performs best, yet it only reduces 7.81% of misses, while FURBYS achieves 57.85% of miss reductions compared with near-optimal offline FLACK.

Performance-per-watt gain. Our power modeling is based on McPAT and CACTI, which are widely used frameworks that take inputs including both static configurations (such as architecture and circuit technology details) and dynamic activity statistics (hardware utilization data during runtime) to predict the power usage [61], [91]. In evaluation, we choose 22nm transistor technology, 3200MHz frequency, and 1.25 core Vdd as static configurations, and feed runtime information

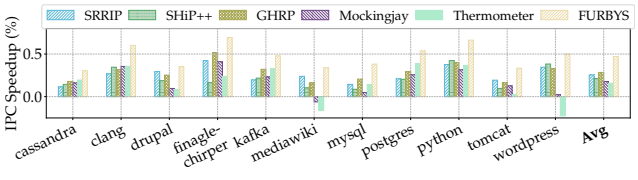

Fig. 11. IPC Speedup of FURBYS over LRU compared to existing replacement policies on 11 applications. FURBYS achieves an average of 0.47% IPC speedup, surpassing all baselines.

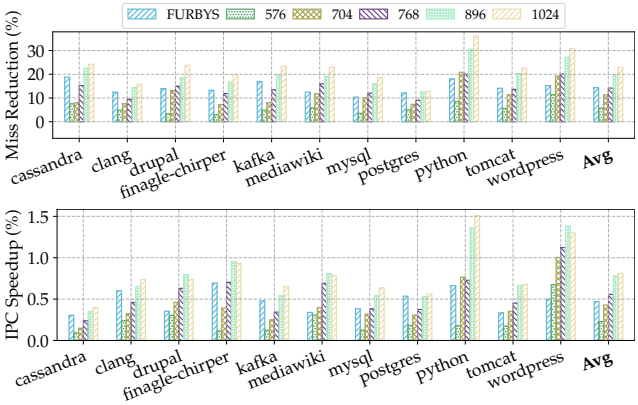

Fig. 12. ISO-performance test. FURBYS operate on 512 entries micro-op cache. The baseline is LRU operating on 512 entries. We enlarge the micro-op cache for LRU policy to compare IPC and miss reduction. To achieve a similar IPC gain or miss reduction as FURBYS, LRU need to on average operate on 50% larger caches. For Postgres, FURBYS beats LRU on caches with doubled sizes.

from Scarab including icache access, micro-op cache access, decoder active cycles, backend usages, and other statistics, into McPAT. As the micro-op cache is not modeled by CACTI by default, we implement the power model following the same structure of the icache but with micro-op cache parameters including size, associativity, etc.

We cross-check our power modeling with existing works [40], [65] and visualize the per-core power breakdown of FURBYS with an example application, Clang. As shown in Figure 13, the baseline without micro-op cache spends 12.5% and 7.7% per-core power on the decoder and icache. Adding a micro-op cache with LRU provides 8.1% power saving by bypassing the decoder and icache. FURBYS further saves 2.2% power by reducing micro-op cache insertion via the proposed dynamic bypass mechanism. We extend the evaluation into all 11 applications, as shown in Figure 9. FURBYS increases the performance-per-watt of the CPU core by 3.10% compared with LRU, surpassing existing policies by 5.1×. We investigate the breakdown of the reduced energy. As Figure 14 shows, around 7.75% of performance-per-watt gain comes from reduced access of icache, 73.26% comes from fewer insertions of micro-op cache and 16.35% from reduced decoder usage.

To quantify the real-world impact, we further estimate the practical benefit of improving 3.10% core performance-per-watt. According to previous works [38], [39], cores can take up to 86% of the CPU power. Therefore, 3.10 of core power reduction can be equivalent to 2.7% of CPU power reduction. To further convert the CPU power reduction to server power reduction, prior work [27] reports that CPU power contributes to 32% of the server power while the server accounts for 80% of the total energy. Thus, 2.7% of CPU power reduction is equivalent to 0.69% of the data center power. Given the large scale of data centers, 0.69% of data center power reduction can save 5 Twh of energy per year [13].

IPC speedup. We measure the IPC speedup of FURBYS and baselines compared to LRU as shown in Figure 11. Note that reducing micro-op cache cache misses only indirectly improves IPC. While the micro-op cache has lower access latency than the legacy decode pipeline, the benefit of this low latency can only be translated into frontend throughput when the frontend recovers from a branch miss [79]. Moreover, since only one PW can be released for one cycle, the limited length of PW hinders the effect of a higher (8 micro-ops per cycle) fetch bandwidth. As a result, lower miss rates only partially translate into performance gain. Nonetheless, our FURBYS achieves 0.49% IPC speedup, which is 60% of the near-optimal FLACK replacement policy, 28.48% of the infinitely large micro-op cache and 1.65× to the state-of-the-art GHRP which corresponds to 0.19% IPC speedup gain. This improvement is comparable to prior works on profile-guided icache replacement policies such as Ripple's IPC gain, which achieves 46.10% of the optimal replacement policy [54].

ISO-performance Evaluation. We also conduct an ISO-performance investigation. We draw the miss rate and IPC of differently sized micro-op cache using the LRU policy in Figure 12 while comparing it to a 512 entry micro-op cache with FURBYS. As shown, the size of the LRU-based micro-op cache must be increased on average by 1.5× to achieve comparable performance with FURBYS. On some workloads (e.g. Postgres), FURBYS achieves a performance comparable to an LRU-cache even with 2× larger entries.

Hardware and runtime overhead. To address concerns about the additional hardware storage required for our FURBYS replacement policy, we analyze the hardware overhead of FURBYS. The overhead per set is FURBYS is 46 bits[2]. The micro-op cache set size is 4608 bits[3]. Therefore, the overhead is only 46 / 4608 = 1%. As on average we are equivalent to 40% cache size gain, 1% overhead is negligible. Furthermore, Since data center applications have already routinely profiled with Intel PT [21], [25], [30], [35], [68], [69], FURBYS will not generate extra online costs from profiling. The first-phase profiling on baseline configuration described in Section V is performed in an offline manner.

Cross validation. To demonstrate the effectiveness of our profile-guided approach, we evaluate whether profiling an application using one set of inputs can improve performance for different inputs. As outlined in Section VI-A, we collect PT traces from data center applications using diverse inputs. These

[2](3 bits weight + 2 bits srrip) × 8 way + 2 × 3 bits pitfall detector = 46 bits.

[3](56 bits per uop × 8 uop per entry + 32 bits per immediate number × 4 imm per entry) × 8 way = 4608 bits

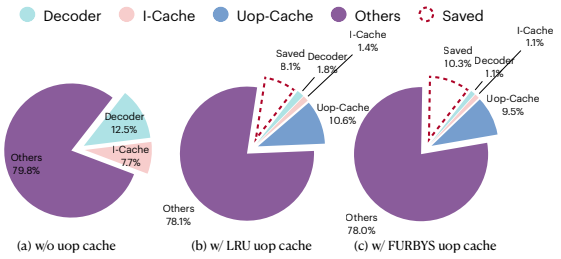

Fig. 13. Per-core energy consumption breakdown of (a) baseline without micro-op cache, (b) LRU, and (c) FLACK micro-op cache on Clang, which has an average branch MPKI, 1.86. We classify energy consumption into decoder, icache, micro-op cache, and others (including BP, BTB, and backend). Both energy of LRU and FLACK micro-op cache are normalized to baseline without micro-op cache, and calculate the amount of saving energy.

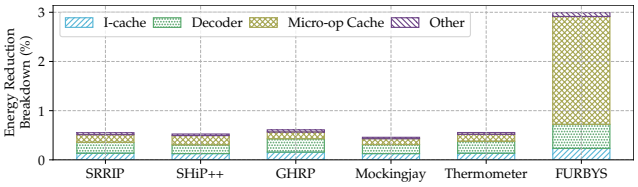

Fig. 14. Average energy reduction breakdown of different applications. FURBYS reduces significantly more energy from the decoder and micro-op cache compared with other replacement policies.

traces are then divided into a training set and a testing set. We profile the training set, group PWs based on their average hit rates, and generate merged hints for FURBYS. Next, we simulate the testing set traces to measure the miss reduction achieved by FURBYS, comparing it to the miss reduction observed when applying FURBYS to the training set. The results of this cross-validation analysis are presented in Figure 18. [4] The results show that our approach can achieve 94.34% of the same-input result, resulting in an average reduction in misses of 13.51% compared to LRU surpassing the best performing online policy GHRP.

Replacement coverage. We conducted an experiment to evaluate the percentage of victim selection made by FURBYS. When the local miss-pitfall detector detects repetitive evictions of a PW, it degrades the replacement policy to SRRIP. After one replacement decision made by SRRIP, it switches back to FURBYS. Our experiment shows that our FURBYS policy is responsible for selecting the victim over 88.68% of the time on average, which showcases the effectiveness of FURBYS.

Bypass coverage. We evaluated the effectiveness of the dynamic bypass mechanism by showing the miss reduction difference when enabling/disabling the bypassing in Figure 21. The results demonstrate that, on average, the bypass mechanism can help us reduce 4.33% more misses. We further check the percentage of bypasses. We show that we can bypass nearly 30% of total insertions, which contributes to per core performance-per-watt gain as shown in Figure 14.

[4]For MySQL, we need to derive the combined profile considering all operations, including insert, delete, and other operations, to get good performance.

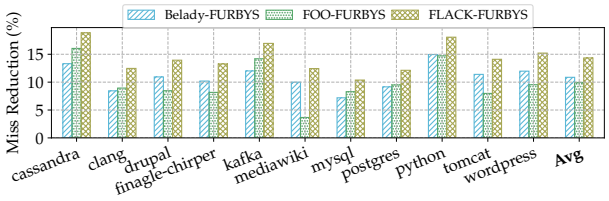

Fig. 15. Miss reduction of FURBYS over LRU when using different offline policies as profile traces. FURBYS gets 3% more miss reduction on average with FLACK, showing the effectiveness of our FLACK policy.

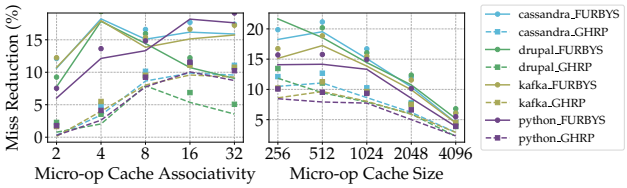

Fig. 16. Miss reduction of FURBYS over LRU with various cache sizes (number of cache entries) and associativities. FURBYS surpasses existing replacement policies under all configurations.

D. Sensitivity Tests

Input traces of offline policies. As discussed in earlier sections, our FLACK policy exhibits a significant improvement in performance compared to Belady's and FOO's policies, with an increase in miss reduction of 4.46% and 17.93%, respectively. In Figure 15, we compare the performance of our online FURBYS policy using the decisions made by Belady, FOO, and FLACK. It shows that using our FLACK policy results in approximately 3.47% fewer misses compared to using Belady as the profile source and 4.39% fewer misses compared to using FOO policy. This finding highlights the effectiveness and importance of using FLACK policy decision as the training input.

Numbers of weight groups. We conducted an experiment to determine the optimal balance between the number of bits grouped by Jenks natural breaks and miss reduction. We varied the number of representative bits from 1 to 8 (resulting in total group counts ranging from 2 to 256) to monitor the changes in miss reduction. As illustrated in Figure 19, using 3 bits strikes a suitable balance between overhead and miss reductions. Increasing the number of representative bits beyond this point does not result in significant performance gains while increasing the hardware overhead.

Depth of local pitfall detector. To optimize the eviction of inactive high-hit rate PWs, we use local miss-pitfall detectors to record previous evictions and change eviction decisions accordingly. The depth of the detector indicates the number of previous evictions recorded for potential degradation. In our experiment, we varied the depth of the detector to evaluate its effect on performance. The results shown in Figure 20 indicate that depth 2 is the best choice that we picked.

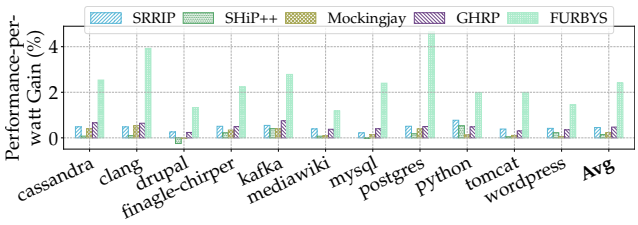

Fig. 17. Performance-per-watt gain of FURBYS over LRU with a micro-op cache of AMD Zen4 configuration. FURBYS surpasses existing replacement policies under all configurations, with an average 2.41% improvement.

Micro-op cache size and associativity. To demonstrate that the proposed method generalizes to different micro-op cache configurations besides AMD Zen3, we evaluate FURBYS under varying cache sizes and associativities. For simplicity, we only present the comparison results between our replacement policy and the best existing policy, GHRP. Figure 16 shows that FURBYS consistently outperforms GHRP across all configurations. As the micro-op cache size increases, the performance gap between FURBYS and GHRP shrinks due to the decrease of capacity misses.

CPU frontend configurations. To showcase the effectiveness of FURBYS on energy efficiency for other frontend configurations(BP, BTP, I-cache, issue width, etc.), we collect performance-per-watt of different policies on 11 application with AMD Zen4 configuration as shown in Figure 17. FURBYS acheives 2.41% performance-per-watt gain, surpassing all other replacement policies. The miss reduction of FURBYS on Zen4 is also included in the previous sensitivity test of cache size and associativity.

VII. Discussion

Hardware configurations. This work focuses on the hit/miss behavior of the micro-op cache. Therefore, we do not model hardware modules that are not strongly related to the micro-op cache miss rate, such as FDIP and LLC. However, precise simulation of these configurations is crucial for future research on IPC performance. A proper warmup would also benefit the simulation accuracy. We leave the development of a more detailed hardware simulator for future work.

Interplay between icache and micro-op cache. Following current industry standards, we model the micro-op cache as inclusive of the icache, based on which we find that the miss reduction of micro-op cache does not proportionally translate into IPC speedup. However, with a non-inclusive micro-op cache, FURBYS achieves an average IPC speedup of 2.5%, surpassing the 0.48% IPC gain observed in the inclusive setup. This substantial performance improvement is primarily due to the equivalent increase in icache size enabled by the non-inclusive micro-op cache and thus reduces the expensive access to L2 cache. Despite these gains, the non-inclusive cache design complicates the invalidation of cache lines for self-modifying code. We leave this trade-off between performance and hardware complexity for future research.

Better micro-op cache replacement policy. To pinpoint potential improvement of the micro-op cache replacement policy, we analyze the performance gap between online policies and FLACK. We classify all PWs into three categories—hot, warm, and cold—based on their total access counts. Additionally, we measure the hit rates of different replacement policies across these PW categories. As shown in Figure 22, the performance gap between FURBYS and FLACK mostly lies in warm and cold PWs. This suggests that a better policy should consider more globally cold but locally hot PWs.

VIII. Related Work

Micro-op cache study. Despite the fact that micro-op caches significantly contribute to frontend efficiency, there is not much work focusing on the study of micro-op cache. Some works [32], [56], [74] focus on the security issue of micro-op caches, and on the performance side, CLASP and compaction [57] solve the fragmentation to increase the utilization. Besides, [79] investigates how to use prefetch to boost performance, with energy overhead introduced by additional decoder. We focus on the replacement policy of the micro-op cache, which is complementary to previous works.

Replacement policies. Cache replacement policies have also been extensively studied by the community. Some works are based on heuristics, for example, [45], [47], [60], [67], [71], [80] use the recent accesses to predict the future reuse distance. Others [34], [36], [64] make decisions based on protecting distance to protect cache lines until reuse. There are other prediction-based replacement policies [18], [37], [42], [55], [72], [76], [84], [86]. Some of them use history accesses to classify cache blocks [44], [52], [92], and others [43], [63], [78] mimic Belady's algorithm [22] to generate learning data. However, all of the above replacement policies are designed for regular caches and do not incorporate essential features of micro-op caches. Therefore, we proposed FURBYS for micro-op cache replacement.

Frontend optimizations. There is a significant body of work that focuses on improving the frontend efficiency through optimizations on different microarchitectures. [20], [51], [53], [58], [59] are recent works that focus on improving performance through icache or BTB prefetching. Others [19], [83] improve performance by optimizing BTB or BP accuracy. We complement these works by optimizing the energy efficiency of frontend with a specific target on the micro-op cache.

IX. Conclusion

Our study demonstrates that cache replacement policies hold significant potential for enhancing CPU energy efficiency. However, existing offline optimal and online practical policies do not fit the key characteristics of the micro-op cache, including disproportionate cost, partial hits and asynchrony. To fill this gap, we propose a new offline replacement policy, FLACK, tailored for the micro-op cache, which achieves a higher theoretical miss reduction of 4.46% than Belady. Building on this new theoretical model, we present FURBYS, which leverages both whole-execution and transient information to

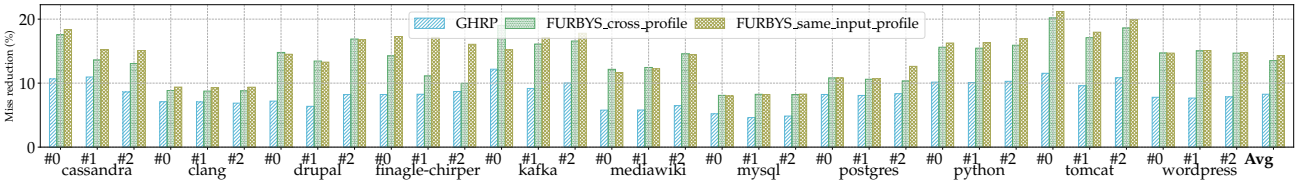

Fig. 18. The result of cross-validation. We utilize half of the traces to generate a combined profile and use this profile to evaluate the remaining traces. Our FURBYS policy achieves a significant portion of the performance compared to using the same-input profile, demonstrating its robustness.

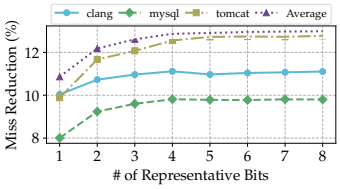

Fig. 19. Miss reduction when changing the number of bits (groups). We chose 3 for the balance between performance and hardware overhead.

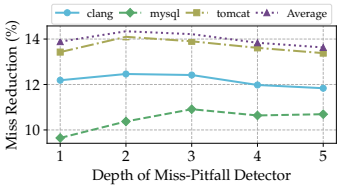

Fig. 20. Miss reduction with various depths of local pitfall detector. We can see that depth 2 provides the best miss reduction performance.

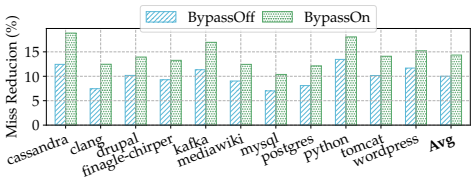

Fig. 21. FURBYS' bypass mechanism helps reduce 4.33% more misses compared to the LRU baseline.

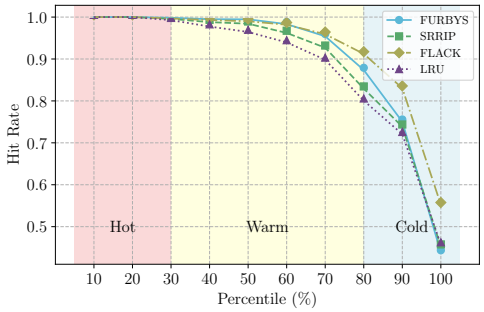

Fig. 22. Hit rate of different PWs with various replacement policies on Kafka. We first sort PWs based on the access counts, based on which we classify all PWs into hot, warm, and cold ones. For example, PWs within 10% to 20% represent ones having top 10% to 20% access times. Results show that all policies work well on the hot PWs, with less than 1% difference in hit rate. FURBYS significantly surpasses other online replacement policies for warm PWs, which explains its overall better miss reduction. The gap between online replacement policies and FLACK mainly lies in the cold PWs.

mimic FLACK. FURBYS achieves a 1.84× greater reduction in miss rate than the state-of-the-art replacement policy and 3.10% performance-per-watt gain for CPU cores.

Acknowledgments

We thank Mingsheng Xu for his discussion and feedback on developing customized simulators. This work was partly supported by the Beijing Natural Science Foundation (L247013), NSFC (U24A6009), and BNRist. This work was also generously supported by Intel's Center for Transformative Server Architectures (TSA), the PRISM Research Center, a JUMP Center cosponsored by SRC and DARPA.

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] "Arm cortex-a710 core technical reference manual," https://developer.arm.com/documentation/101800/0201/RAS-Extension-support-/Cache-protection-behavior.
[6] "Clang c language family frontend for llvm," [Online; accessed 19-Nov-2021]. [Online]. Available: https://clang.llvm.org/
[7] "Intel 64 and ia-32 architectures software developer's manual: Volume 3," https://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-software-developer-system-programming-manual -325384.html.
[8] "The microarchitecture of intel, amd, and via cpus:an optimization guide for assembly programmers and compiler makers," https://www.agner.org/optimize/microarchitecture.pdf.
[9] "Mysql," [Online; accessed 19-Nov-2021]. [Online]. Available: https://www.mysql.com
[10] "Postgresql: Documentation: 14: pgbench," [Online; accessed 19-Nov-2021]. [Online]. Available: https://www.postgresql.org/docs/current/pgbench.html
[11] "Postgresql: The world's most advanced open source database," [Online; accessed 19-Nov-2021]. [Online]. Available: https://www.postgresql.org/
[12] "The python performance benchmark suite," [Online; accessed 19-Nov-2021]. [Online]. Available: https://pyperformance.readthedocs.io/
[13] "The real amount of energy a data center uses," https://www.akcp.com/blog/the-real-amount-of-energy-a-data-center-use/.
[14] "Scarab," https://github.com/hpsresearchgroup/scarab.
[15] "Twitter finagle," https://twitter.github.io/finagle/.
[16] "Welcome to python.org," [Online; accessed 19-Nov-2021]. [Online]. Available: https://www.python.org/
[17] "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).
[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] 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.
[20] T. Asheim, R. Kumar, and B. Grot, "Fetch-directed instruction prefetching revisited," *arXiv preprint arXiv:2006.13547*, 2020.
[21] 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.

[22] L. A. Belady, "A study of replacement algorithms for a virtual-storage computer," *IBM Systems journal*, vol. 5, no. 2, pp. 78–101, 1966.

[23] D. S. Berger, N. Beckmann, and M. Harchol-Balter, "Practical bounds on optimal caching with variable object sizes," *Proc. ACM Meas. Anal. Comput. Syst.*, vol. 2, no. 2, jun 2018. [Online]. Available: https://doi.org/10.1145/3224427

[24] 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.

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

[26] C. Chi and H. Dietz, "Improving cache performance by selective cache bypass," in *Proceedings of the Twenty-Second Annual Hawaii International Conference on System Sciences. Volume 1: Architecture Track*, vol. 1. Los Alamitos, CA, USA: IEEE Computer Society, jan 1989, pp. 277,278,279,280,281,282,283,284,285. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/HICSS.1989.47168

[27] Y. Cho and Y. M. Ko, "Power- and qos-aware job assignment with dynamic speed scaling for cloud data center computing," *IEEE Access*, vol. 10, pp. 38 284–38 298, 2022.

[28] clamchowder, "AMD's Zen 4 Part 1: Frontend and Execution Engine — chipsandcheese.com," https://chipsandcheese.com/2022/11/05/amds-zen-4-part-1-frontend-and-execution-engine/, 2022, [Accessed 01-08-2024].

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

[30] 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.

[31] M. Dayarathna, Y. Wen, and R. Fan, "Data center energy consumption modeling: A survey," *IEEE Communications Surveys & Tutorials*, vol. 18, no. 1, pp. 732–794, 2016.

[32] S. Deng, B. Huang, and J. Szefer, "Leaky frontends: Security vulnerabilities in processor frontends," in *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2022, pp. 53–66.

[33] A. Deshmukh and Y. N. Patt, "Criticality driven fetch," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 380–391. [Online]. Available: https://doi.org/10.1145/3466752.3480115

[34] 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.

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

[36] 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.

[37] 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.

[38] B. Goel and S. A. McKee, "A methodology for modeling dynamic and static power consumption for multicore processors," in *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2016, pp. 273–282.

[39] J. Haj-Yahya, A. Mendelson, Y. Ben Asher, A. Chattopadhyay, J. Haj-Yahya, A. Mendelson, Y. Ben Asher, and A. Chattopadhyay, "Power modeling at high-performance computing processors," *Energy Efficient High Performance Processors: Recent Approaches for Designing Green High Performance Computing*, pp. 73–105, 2018.

[40] M. Hirki, Z. Ou, K. N. Khan, J. K. Nurminen, and T. Niemi, "Empirical study of the power consumption of the x86-64 instruction decoder," in *USENIX Workshop on Cool Topics on Sustainable Data Centers (CoolDC 16)*, 2016.

[41] S. Hosseini-Khayat, "On optimal replacement of nonuniform cache objects," *IEEE Transactions on Computers*, vol. 49, no. 8, pp. 769–778, 2000.

[42] 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.

[43] 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.

[44] 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.

[45] 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.

[46] G. F. Jenks, "The data model concept in statistical mapping," *International yearbook of cartography*, vol. 7, pp. 186–190, 1967.

[47] 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.

[48] 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.

[49] 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.

[50] 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.

[51] 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.

[52] 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.

[53] 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.

[54] 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.

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

[56] J. Kim, H. Jang, H. Lee, S. Lee, and J. Kim, "Uc-check: Characterizing micro-operation caches in x86 processors and implications in security and performance," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 550–564. [Online]. Available: https://doi.org/10.1145/3466752.3480079

[57] J. B. Kotra and J. Kalamatianos, "Improving the utilization of micro-operation caches in x86 processors," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2020, pp. 160–172.

[58] 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.

[59] 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.

[60] 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.

[61] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "Mcpat: An integrated power, area, and timing modeling framework for multicore and manycore architectures," in *2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2009, pp. 469–480.

[62] 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

[63] 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.

[64] 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.

[65] S. Manne, A. Klauser, and D. Grunwald, "Pipeline gating: speculation control for energy reduction," in *Proceedings. 25th Annual International Symposium on Computer Architecture (Cat. No.98CB36235)*, 1998, pp. 132–141.

[66] S. Mirbagher 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)*, 2018, pp. 519–532.

[67] 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.

[68] 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.

[69] 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.

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

[71] 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.

[72] 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.

[73] 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.

[74] X. Ren, L. Moody, M. Taram, M. Jordan, D. M. Tullsen, and A. Venkat, "I see dead µops: Leaking secrets via intel/amd micro-op caches," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, 2021, pp. 361–374.

[75] H. Rong, H. Zhang, S. Xiao, C. Li, and C. Hu, "Optimizing energy consumption for data centers," *Renewable and Sustainable Energy Reviews*, vol. 58, pp. 674–691, 2016. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1364032115016664

[76] 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.

[77] I. Shah, A. Jain, and C. Lin, "Effective mimicry of belady's min policy," in *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2022, pp. 558–572.

[78] 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.

[79] S. Singh, A. Perais, A. Jimborean, and A. Ros, "Alternate path µ-op cache prefetching," in *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2024.

[80] 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.

[81] B. Solomon, A. Mendelson, D. Orenstien, Y. Almog, and R. Ronen, "Micro-operation cache: a power aware frontend for variable instruction length isa," in *ISLPED'01: Proceedings of the 2001 International Symposium on Low Power Electronics and Design (IEEE Cat. No.01TH8581)*, 2001, pp. 4–9.

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

[83] 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.

[84] 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.

[85] D. N. Suggs, "Operation cache," Patent No. WO2018106736A1, World Intellectual Property Organization (WIPO), 6 2018, filed by Advanced Micro Devices, Inc. [Online]. Available: https://patentscope.wipo.int/search/en/detail.jsf?docId=WO2018106736

[86] 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.

[87] WikiChip, "Sandy Bridge (client) - Microarchitectures - Intel - WikiChip — en.wikichip.org," https://en.wikichip.org/wiki/intel/microarchitectures/sandy_bridge_(client), 2012, [Accessed 01-08-2024].

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

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

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

[91] S. J. Wilton and N. P. Jouppi, "Cacti: An enhanced cache access and cycle time model," *IEEE Journal of solid-state circuits*, vol. 31, no. 5, pp. 677–688, 1996.

[92] 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.

[93] V. Young, C.-C. Chou, A. Jaleel, and M. Qureshi, "Ship++: Enhancing signature-based hit predictor for improved cache performance," in *The 2nd Cache Replacement Championship (CRC-2 Workshop in ISCA 2017)*, 2017.

[94] 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.

APPENDIX

A. Abstract

In the artifact, we provide materials and instructions that enable the community to reproduce our main experiment results. This artifact includes our customized CPU simulator scarab that implements FLACK, FURBYS, and other baselines. We include the traces collected from data center applications used for performance evaluation. We also include scripts for environment setups and evaluations. We provide the required environments in a docker image for reproducibility and use Google Drive to share large files like traces and images.

B. Artifact check-list (meta-information)

- **Algorithm:** cache replacement policy
- **Program:** Scarab
- **Compilation:** g++
- **Data set:** included traces of data center applications
- **Run-time environment:** Ubuntu 22.04 docker image
- **Hardware:** High performance CPU machines
- **Metrics:** miss reduction, performance-per-watt gain
- **Experiments:** FURBYS and FLACK miss reduction and PPW gain, perfect structure PPW gain, cross-validation, etc.
- **How much disk space is required (approximately)?:** 1TB
- **How much memory (DRAM) is required?:** 512GB
- **How much time is needed to prepare workflow?:** 0.5h
- **How much time is needed to complete experiments?:** 36h
- **Publicly available?:** Yes, via Google Drive
- **Code licenses (if publicly available)?:** Apache-2.0 license
- **Data licenses (if publicly available)?:** Apache-2.0 license

C. Description

1) How to access: We provide a docker image with a ready runtime environment, which can be obtained from the provided link. The traces can be downloaded from google drive.

2) Hardware dependencies: The artifact does not have strict requirements for hardware. We recommend using multi-core CPUs (around 256 cores) and around 500GB of memory to speed up the simulation.

3) Software dependencies: The artifact includes a docker for runtime. To reproduce the result, we recommend using Linux with docker.

4) Data sets: The dataset is provided via Google Drive.

D. Installation

The installation can be divided into several steps:

1) Container setup. First download provided image from Google Drive and decompress using `tar`.

```
tar -xvzf uop-ae-image-v1.tar.gz
docker load -i uop-ae-image-v1.tar
docker run -it uop-ae-image-v1
```

2) Trace setup: download traces from Google Drive and decompress it to `/code/datacenterTrace`.
3) Environment variables setup:

```
cd /code/script
source source.sh
```

4) Simulator setup:

```
bash 1-simulators/run.sh
bash 3-power-model/run.sh
```

5) Profile-guided traces setup:

```
bash 2-prepare-traces/run.sh
bash 10-prepare-cross-traces/run.sh
bash 5-prepare-belady-traces/run.sh
```

E. Evaluation and expected results

PPW gain of perfect structures (Figure 2) We compare the performance per-watt (PPW) assuming different micro-architectures are perfect. The result is located in `4-perfect-arch/power.csv`. This should align with the numbers shown in the figure.

```
bash 4-perfect-arch/run.sh
```

FURBYS miss reduction, PPW gain (Figure 8, 9) This experiment shows the main results of the paper. We compare the miss reduction of performance per-watt (PPW) of FLACK and FURBYS against existing techniques. The PPW results are in `power.csv` and the miss reduction is plotted in folder `plot` under `6-furbys-main-evaluation`.

```
bash 6-furbys-main-evaluation/run.sh
```

FLACK ablation study (Figure 10) This experiment shows the FLACK ablation study. We incorporate different components of FLACK to see their impact on performance. The result is in the `plot` folder.

```
bash 7-flack-ablation/run.sh
```

FURBYS cross-validation (Figure 18) We derive FURBYS hints using a set of traces and use the trace to evaluate the performance of FURBYS on the rest of the traces to demonstrate the generalization of FURBYS. The result is visualized in the `plot` folder.

```
bash 11-cross-validation/run.sh
```

FURBYS with different input traces (Figure 15) We use FLACK, FOO and Belady to generate traces for FURBYS and evaluate its performance. The result is in the `plot` folder.

```
bash 9-furbys-different-traces/run.sh
```

ISO-IPC test (Figure 12) We compare the miss reduction and IPC speedup of FURBYS against LRU with various cache sizes. The result is in the `plot` folder.

```
bash 8-iso-ipc/run.sh
```