Ripple: Profile-Guided Instruction Cache Replacement for Data Center Applications

Tanvir Ahmed Khan*   Dexin Zhang†   Akshitha Sriraman*   Joseph Devietti‡
Gilles Pokam§   Heiner Litz¶   Baris Kasikci*
*University of Michigan   †University of Science and Technology of China   ‡University of Pennsylvania
§Intel Corporation   ¶University of California, Santa Cruz
*{takh, akshitha, barisk}@umich.edu   †zhangdexin@mail.ustc.edu.cn
‡devietti@cis.upenn.edu   §gilles.a.pokam@intel.com   ¶hlitz@ucsc.edu

Abstract—Modern data center applications exhibit deep software stacks, resulting in large instruction footprints that frequently cause instruction cache misses degrading performance, cost, and energy efficiency. Although numerous mechanisms have been proposed to mitigate instruction cache misses, they still fall short of ideal cache behavior, and furthermore, introduce significant hardware overheads. We first investigate why existing I-cache miss mitigation mechanisms achieve sub-optimal performance for data center applications. We find that widely-studied instruction prefetchers fall short due to wasteful prefetch-induced cache line evictions that are not handled by existing replacement policies. Existing replacement policies are unable to mitigate wasteful evictions since they lack complete knowledge of a data center application's complex program behavior.

To make existing replacement policies aware of these eviction-inducing program behaviors, we propose *Ripple*, a novel software-only technique that profiles programs and uses program context to inform the underlying replacement policy about efficient replacement decisions. *Ripple* carefully identifies program contexts that lead to I-cache misses and sparingly injects "cache line eviction" instructions in suitable program locations at link time. We evaluate *Ripple* using nine popular data center applications and demonstrate that *Ripple* enables any replacement policy to achieve speedup that is closer to that of an ideal I-cache. Specifically, *Ripple* achieves an average performance improvement of 1.6% (up to 2.13%) over prior work due to a mean 19% (up to 28.6%) I-cache miss reduction.

I. INTRODUCTION

Modern data center applications are becoming increasingly complex. These applications are composed of deep and complex software stacks that include various kernel and networking modules, compression elements, serialization code, and remote procedure call libraries. Such complex code stacks often have intricate inter-dependencies, causing millions of unique instructions to be executed to serve a single user request. As a result, modern data center applications face instruction working set sizes that are several orders of magnitude larger than the instruction cache (I-cache) sizes supported by today's processors [13, 46].

Large instruction working sets precipitate frequent I-cache misses that cannot be effectively hidden by modern out-of-order mechanisms, manifesting as glaring stalls in the critical path of execution [60]. Such stalls deteriorate application performance at scale, costing millions of dollars and consuming significant energy [13, 95]. Hence, eliminating instruction

misses to achieve even single-digit percent speedups can yield immense performance-per-watt benefits [95].

I-cache miss reduction mechanisms have been extensively studied in the past. Several prior works proposed next-line [9, 89, 92], branch-predictor-guided [60, 61, 82], or history-based [25, 26, 31, 51, 59, 70, 77, 83] hardware instruction prefetchers and others designed software mechanisms to perform code layout optimizations for improving instruction locality [17, 64, 67, 74–76]. Although these techniques are promising, they (1) require additional hardware support to be implemented on existing processors and (2) fall short of the ideal I-cache behavior, *i.e.*, an I-cache that incurs no misses. To completely eliminate I-cache misses, it is critical to first understand: why do existing I-cache miss mitigation mechanisms achieve sub-optimal performance for data center applications? How can we further close the performance gap to achieve near-ideal application speedup?

To this end, we comprehensively investigate why existing I-cache miss mitigation techniques fall short of an ideal I-cache, and precipitate significant I-cache Misses Per Kilo Instruction (MPKI) in data center applications (§II). Our investigation finds that the most widely-studied I-cache miss mitigation technique, instruction prefetching, still falls short of ideal I-cache behavior. In particular, existing prefetchers perform many unnecessary prefetches, polluting the I-cache, causing wasteful evictions. Since wasteful evictions can be avoided by effective cache replacement policies, we study previous proposals such as the Global History Reuse Predictor (GHRP) [7] (the only replacement policy specifically targeting the I-cache, to the best of our knowledge) as well as additional techniques that were originally proposed for data caches, such as Hawkeye [40]/Harmony [41], SRRIP [43], and DRRIP [43].

Driven by our investigation results, we propose *Ripple*, a profile-guided technique to optimize I-cache replacement policy decisions for data center applications. *Ripple* first performs an offline analysis of the basic blocks (*i.e.*, sequence of instructions without a branch) executed by a data center application, recorded via efficient hardware tracing (*e.g.*, Intel's Processor Trace [19, 58]). For each basic block, *Ripple* then determines the cache line that an ideal replacement policy would evict based on the recorded basic block trace. *Ripple* computes basic blocks whose executions likely signal a future eviction

for an ideal replacement policy. If this likelihood is above a certain threshold (which we explore and determine empirically in §III), *Ripple* injects an invalidation instruction to evict the victim cache line. Intel recently introduced such an invalidation instruction — `CLDemote`, and hence *Ripple* can readily be implemented on upcoming processors.

We evaluate *Ripple* in combination with I-cache prefetching mechanisms, and show that *Ripple* yields on average 1.6% (up to 2.13%) improvement over prior work as it reduces I-cache misses by on average 19% (up to 28.6%). As *Ripple* is primarily a software-based technique, it can be implemented on top of any replacement policy that already exists in hardware. In particular, we evaluate two variants of *Ripple*. *Ripple*-Least Recently Used (LRU) is optimized for highest performance and reduces I-cache MPKI by up to 28.6% over previous proposals, including Hawkeye/Harmony, DRRIP, SRRIP, and GHRP. On the other hand, *Ripple*-Random is optimized for lowest storage overhead, eliminating all meta data storage overheads, while outperforming prior work by up to 19%. *Ripple* executes only 2.2% extra dynamic instructions and inserts only 3.4% new static instructions on average. In summary, we show that *Ripple* provides significant performance gains compared to the state-of-the-art I-cache miss mitigation mechanisms while minimizing the meta data storage overheads of the replacement policy.

In summary, we make the following contributions:

- A detailed analysis of why existing I-cache miss mitigation mechanisms fall short for data center applications
- Profile-guided replacement: A software mechanism that uses program behavior to inform replacement decisions
- *Ripple*: A novel profile-guided instruction cache miss mitigation mechanism that can readily work on any existing replacement policy
- An evaluation demonstrating *Ripple*'s efficacy at achieving near-ideal application speedup.

II. WHY DO EXISTING I-CACHE MISS MITIGATION TECHNIQUES FALL SHORT?

In this section, we analyze why existing techniques to mitigate I-cache misses fall short, precipitating high miss rates in data center applications. We first present background information on the data center applications we study (§II-A). We then perform a limit study to determine the maximum speedup that can be obtained with an ideal I-cache for applications with large instruction footprints (§II-B). Next, we evaluate existing prefetching mechanisms, including next-line prefetcher and FDIP [82], to analyze why these techniques achieve sub-optimal performance (§II-C). Finally, we analyze existing cache replacement policies, including LRU, Harmony, DRRIP, SRRIP, and GHRP, to quantify their performance gap with the optimal replacement policy (§II-D). This analysis provides the foundation for *Ripple*, a novel prefetch-aware I-cache replacement policy that achieves high performance with minimal hardware overheads.
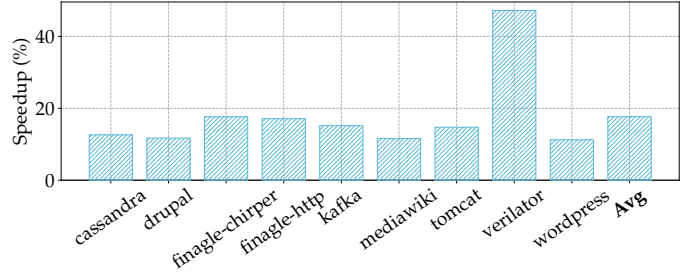

Fig. 1: Ideal I-cache speedup over an LRU baseline without any prefetching: These data center applications can gain on average 17.7% speedup with an ideal I-cache with no misses.

A. Background on evaluated applications

We study nine widely-used real-world data center applications that suffer from substantial I-cache misses [55]—these applications lose 23-80% of their pipeline slots due to frequent I-cache misses. We study three HHVM applications from Facebook's OSS-performance benchmark suite [5], including `drupal` [103] (a PHP content management system), `mediawiki` [104] (a wiki engine), and `wordpress` [105] (a popular content management system). We investigate three Java applications from the DaCapo benchmark suite [16], including `cassandra` [1] (a NoSQL database used by companies like Netflix), `kafka` [102] (a stream processing system used by companies like Uber), and `tomcat` [2] (Apache's implementation of Java Servlet and Websocket). From the Java Renaissance [79] benchmark suite, we analyze *Finagle-Chirper* (Twitter's microblogging service) and *Finagle-HTTP* [3] (Twitter's HTTP server). We also study *Verilator* [4, 10] (used by cloud companies for hardware simulation). We describe our complete experimental setup and simulation parameters in §IV.

B. Ideal I-cache: The theoretical upper bound

An out-of-order processor's performance greatly depends on how effectively it can supply itself with instructions. Therefore, these processors use fast dedicated I-caches that can typically be accessed in 3-4 cycles [93]. To maintain a low access latency, modern processors typically have small I-cache sizes (*e.g.*, 32KB) that are overwhelmed by data center applications' multi-megabyte instruction footprints [11, 13, 46, 76] incurring frequent I-cache misses. To evaluate the true cost of these I-cache misses as well as the potential gain of I-cache optimizations, we explore the speedup that can be obtained for data center applications with an ideal I-cache that incurs no misses. Similar to prior work [13, 55], we compute the speedup relative to a baseline cache configuration with no prefetching and with an LRU replacement policy. As shown in Fig. 1, an ideal I-cache can provide between 11-47% (average of 17.7%) speedup over the baseline cache configuration.

C. Why do modern instruction prefetchers fall short?

Prior works [13, 24, 55, 82] have proposed prefetching techniques to overcome the performance challenge induced by insufficiently sized I-caches. Fetch Directed Instruction
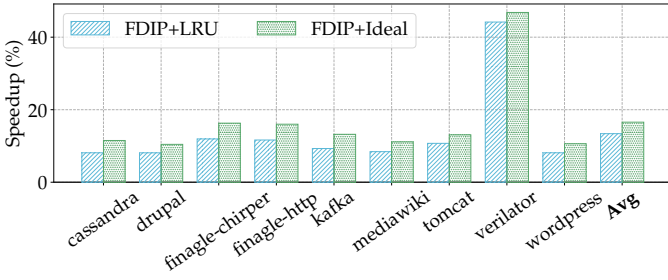
Fig. 2: Fetch directed instruction prefetching (FDIP) speedup over an LRU baseline without any prefetching: FDIP provides 13.4% mean speedup with LRU replacement policy. However, with an ideal cache replacement policy FDIP can provide 16.6% average speedup which is much closer to ideal cache speedup.

Prefetching (FDIP) [82] is the state-of-the art mechanism that is implemented on multiple real-world processors [32, 78, 85, 97] due to its performance and moderate implementation complexity. Fig. 2 shows FDIP's speedup over the baseline I-cache configuration without any prefetching. Both FDIP and baseline configurations use the LRU replacement policy. As shown, FDIP+LRU provides between 8-44% (average of 13.4%) speedup over the baseline. This represents a 4.3% performance loss over the ideal cache speedup (17.7%).

To analyze why FDIP falls short of delivering ideal performance, we equip the I-cache with a prefetch-aware ideal replacement policy. In particular, when leveraging a revised version of the Demand-MIN prefetch-aware replacement policy [41], we find that the speedup increases to on average 16.6% falling short of the ideal cache by just 1.14%. In other words, FDIP with prefetch-aware ideal replacement policy outperforms FDIP with LRU by 3.16%. This observation highlights the importance of combining state-of-the-art I-cache prefetching mechanisms with better replacement policies.

To confirm the generality of our observation, we repeat the above experiment with a standard Next-Line Prefetcher (NLP) [92]. We find that the combination of NLP prefetching with ideal cache replacement results in a 3.87% speedup over the NLP baseline without a perfect replacement policy.

To understand the key reasons behind the near-ideal speedups provided by the prefetch-aware ideal replacement policy, we first briefly describe how the policy works and then summarize the key reasons for near-ideal speedups [41]. We also quantify the speedups that an ideal replacement policy can provide for the data center applications we evaluate.

Prefetch-aware ideal replacement policy. Our ideal prefetch-aware replacement policy is based on a revised version of Demand-MIN [41]. In its revised form, Demand-MIN evicts the cache line that is prefetched farthest in the future if there is no earlier demand access to that line. If there exists no such prefetch for a given cache set, Demand-MIN evicts the line whose demand access is farthest in the future. We now detail and quantify two observations that were originally made by Demand-MIN: (1) evicting inaccurately prefetched cache lines

reduces I-cache misses and (2) not evicting hard-to-prefetch cache lines reduces I-cache misses.

Observation #1: Early eviction of inaccurately prefetched cache lines reduces I-cache misses. The ideal replacement policy can evict inaccurately prefetched cache lines (*i.e.*, ones that will not be used) early, improving performance. Like most practical prefetchers, FDIP inaccurately prefetches many cache lines as its decisions are guided by a branch predictor, which occasionally mispredicts branch outcomes. However, the ideal replacement policy has knowledge of all future accesses, so it can immediately evict inaccurately prefetched cache lines, minimizing their negative performance impact. Across our nine data center applications, the ideal cache replacement policy combined with FDIP, provides 1.35% average speedup (out of 3.16% total speedup of FDIP+ideal over FDIP+LRU) relative to an LRU-based baseline replacement policy (also combined with FDIP) due to the early eviction of inaccurately-prefetched cache lines.

Observation #2: Not evicting hard-to-prefetch cache lines reduces I-cache misses. An ideal replacement policy can keep hard-to-prefetch cache lines in the cache while evicting easy-to-prefetch lines. Cache lines that cannot be prefetched with good accuracy or at all, are considered hard-to-prefetch cache lines. For example, FDIP is guided by the branch predictor. A cache line that will be prefetched based on the outcome of a branch, may not be prefetched if the predictor cannot easily predict the branch outcome (*e.g.*, due to an indirect branch)—in that case, the line is hard-to-prefetch. Easy-to-prefetch cache lines are cache lines that the prefetcher is often able to prefetch accurately. For example, a cache line that FDIP can prefetch based on the outcome of a direct unconditional branch is an easy-to-prefetch cache line. Since the ideal replacement policy has knowledge of all accesses and prefetches, it can (1) accurately identify hard-to-prefetch and easy-to-prefetch lines for any given prefetching policy and (2) prioritize the eviction of easy-to-prefetch lines over hard-to-prefetch lines. Across our nine data center applications, the ideal cache replacement policy combined with FDIP, provides 1.81% average speedup (out of 3.16% total speedup of FDIP+ideal over FDIP+LRU) relative to an LRU-based baseline replacement policy (also combined with FDIP) due to not evicting hard-to-prefetch lines.

Summary: Exploiting the above observations for an optimized prefetch-aware replacement policy requires knowledge about future instruction sequences that are likely to be executed. We find that this information can be provided by the static control-flow analysis based on execution profiles and instruction traces. As described in §IV, *Ripple* leverages these analysis techniques and performs well-informed replacement decisions in concert with the prefetcher, to achieve near-ideal performance.

D. Why do existing replacement policies fall short?

In the previous section, we demonstrated that a prefetch-aware ideal cache replacement policy can provide on average 3.16% speedup relative to a baseline LRU replacement policy. In this section, we explore the extent to which existing replacement policies close this speedup gap. As there exist

TABLE I: Storage overheads of different replacement policies for a 32KB, 8-way set associative instruction cache that has 64B cache lines.

Replacement Policy	Overhead	Notes
LRU	64B	1-bit per line
GHRP	4.13KB	3KB prediction table, 64B prediction bits , 1KB signature, 2B history register
SRRIP	128B	2-bits× associativity
DRRIP	128B	2-bits× associativity
Hawkeye/Harmony	5.1875KB	1KB sampler (200 entries), 1KB occupancy vector, 3KB predictor, 192B RRIP counters

only few works on I-cache replacement policies apart from GHRP [7], we also explore data cache replacement policies such as LRU [69], Hawkeye [40]/Harmony [41], SRRIP [43], and DRRIP [43] applied to the I-cache.

GHRP [7] was designed to eliminate I-cache and Branch Target Buffer (BTB) misses. During execution, GHRP populates a prediction table indexed by control flow information to predict whether a given cache line is *dead* or *alive*. While making replacement decisions, GHRP favors evicting lines that are more likely to be dead. Every time GHRP evicts a cache line, it uses a counter to update the predictor table that the evicted cache line is more likely to be dead. Similarly, GHRP updates the predictor table after each hit in the I-cache to indicate that that the hit cache line is more likely to be alive. GHRP uses 4.13KB extra on-chip metadata for a 32KB I-cache to primarily store this prediction table.

Hawkeye/Harmony [40] was designed for the data cache, specifically for the Last Level Cache (LLC). By simulating the ideal cache replacement policy [15] on access history, Hawkeye determines whether a Program Counter (PC) is "cache-friendly" or "cache-averse", *i.e.*, whether the data accessed while the processor executes the instruction corresponding to this PC follows a cache-friendly access pattern [42] or not. Cache lines accessed at a cache-friendly PC are maintained using the LRU cache replacement policy, while lines accessed by a cache-averse PC are marked to be removed at the earliest opportunity. Harmony [41] is a state-of-the-art replacement policy that adds prefetch-awareness to Hawkeye. It simulates Demand-MIN [41] on the access history in hardware to further categorize PCs as either prefetch-friendly or prefetch-averse.

SRRIP [43] was mainly designed to eliminate the adverse effects of the *scanning* [14] cache access pattern, where a large number of cache lines are accessed without any temporal locality (*i.e.*, a sequence of accesses that never repeat). SRRIP assumes that all newly-accessed cache lines are cache-averse (*i.e.*, scans). Only when a cache line is accessed for a second time, SRRIP promotes the status of the line to cache-friendly.

DRRIP [43] improves over SRRIP by considering thrashing access patterns, *i.e.*, when the working set of the application exceeds the cache size [20]. DRRIP reserves positions for both cache-friendly and cache-averse lines via set-dueling [80].

Fig. 3 shows the performance for different cache replacement policies over the LRU baseline with FDIP. Tab. I shows the metadata storage overheads induced by each replacement policy. As shown, none of the existing replacement policies provide any performance or storage benefits over LRU even though the ideal cache replacement policy provides 3.16% average speedup over LRU. We now explain why each of these prior replacement policies do not provide any significant benefit.

GHRP classifies cache lines into dead or alive based on the prediction table, to inform eviction decisions. One issue with GHRP is that it increases the classification confidence in the prediction table after eviction even if the decision was incorrect (*e.g.*, evicted a line that was still needed). We modified GHRP so that it decreases the confidence in the prediction table after each eviction. With this optimization, GHRP outperforms LRU by 0.1%.

Hawkeye/Harmony predicts whether a PC is likely to access a cache-friendly or cache-averse cache line. This insight works well for D-caches where an instruction at a given PC is responsible for accessing many D-cache lines that exhibit similar cache-friendly or cache-averse access patterns. However, for I-cache, an instruction at a given PC is responsible for accessing just one cache line that contains the instruction itself. If the line has multiple cache-friendly accesses followed by a single cache-averse access, Hawkeye predicts the line as cache-friendly. Therefore, Hawkeye cannot identify that single cache-averse access and cannot adapt to dynamic I-cache behavior. For I-cache accesses in data center applications, Hawkeye predicts almost all PCs (more than 99%) as cache friendly and hence fails to provide performance benefits over LRU.

SRRIP and DRRIP can provide significant performance benefits over LRU if the cache accesses follow a scanning access pattern. Moreover, DRRIP provides further support for thrashing access patterns [20]. For the I-cache, scanning access patterns are rare and hence classifying a line as a scan introduces a penalty over plain LRU. We quantify the scanning access pattern for our data center applications by measuring the compulsory MPKI (misses that happen when a cache line is accessed for the first time [34]). For these applications, compulsory MPKI is very small (0.1-0.3 and 0.16 on average). Moreover, both SRRIP and DRRIP arbitrarily assume that all cache lines will have similar access patterns (either scan or thrash) which further hurts data center applications' I-cache performance. Consequently, SRRIP and DRRIP cannot outperform LRU for I-cache accesses in data center applications.

We observe that data center applications tend to exhibit a unique reuse distance behavior, *i.e.*, the number of unique cache lines accessed in the current associative set between two consecutive accesses to the same cache line, or the re-reference interval [43] of a given cache line varies widely across the program life time. Due to this variance, a single I-cache line can be both cache-friendly and cache-averse at different stages of the program execution. Existing works do not adapt to this dynamic variance and hence fail to improve performance over LRU. We combine these insights with our observations in §II-C to design *Ripple*, a profile-guided replacement policy for data center applications.
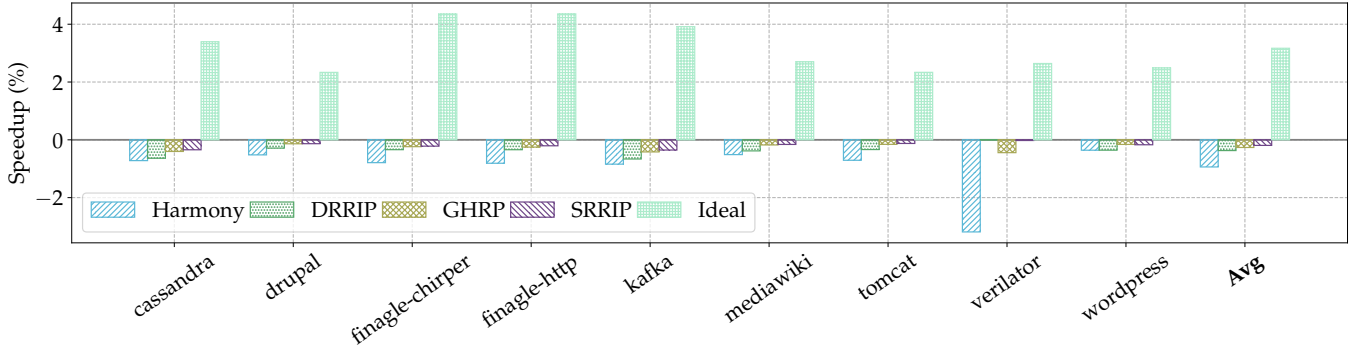
Fig. 3: Speedup for different cache replacement policies over an LRU baseline with FDIP at the L1 I-cache: None of the existing policies outperform LRU, although an ideal replacement policy provides on average 3.16% speedup.

III. THE RIPPLE REPLACEMENT MECHANISM

As we show in our analysis, an ideal cache replacement policy provides on average 3.16% speedup over an LRU I-cache for data center applications. Moreover, we find that existing instruction and data cache replacement policies [7, 40, 41, 43] fall short of the LRU baseline, since they are ineffective at avoiding wasteful evictions due to the complex instruction access behaviors. Hence, there is a critical need to assist the underlying replacement policy in making smarter eviction decisions by informing it about complex instruction accesses.

To this end, we propose augmenting existing replacement mechanisms with *Ripple*—a novel profile-guided replacement technique that carefully identifies program contexts leading to I-cache misses and strives to evict the cache lines that would be evicted by the ideal policy. *Ripple*'s operation is agnostic of the underlying I-cache replacement policy. It sparingly injects "cache line eviction" instructions in suitable program locations at link time to assist an arbitrary replacement policy implemented in hardware. *Ripple* introduces no additional hardware overhead and can be readily implemented on soon-to-be-released processors. *Ripple* enables an existing replacement policy to further close the performance gap in achieving the ideal I-cache performance.

Fig. 4 shows *Ripple*'s design components. First, at run time (online), *Ripple* profiles a program's basic block execution sequence using efficient hardware-based control flow tracing support such as Intel PT [58] or Last Branch Record (LBR) [21] (step ①, §III-A). *Ripple* then analyzes the program trace offline using the ideal I-cache replacement policy (step ②, §III-B) to compute a set of *cue blocks*. A *cue block* is a basic block whose execution almost always leads to the ideal *victim* cache line to be evicted. The key idea behind *Ripple*'s analysis is to mimic an ideal policy that would evict a line that will be used farthest in the future. During recompilation, *Ripple* then injects an instruction in the cue block that invalidates the victim line (step ③ §III-C). Consequently, the next time a cache line needs to be inserted into the cache set that the victim line belongs to, the victim line will be evicted. In contrast to prior work [7, 40, 41, 43, 43], *Ripple* moves the compute-intensive task of identifying the victim line from the hardware to the
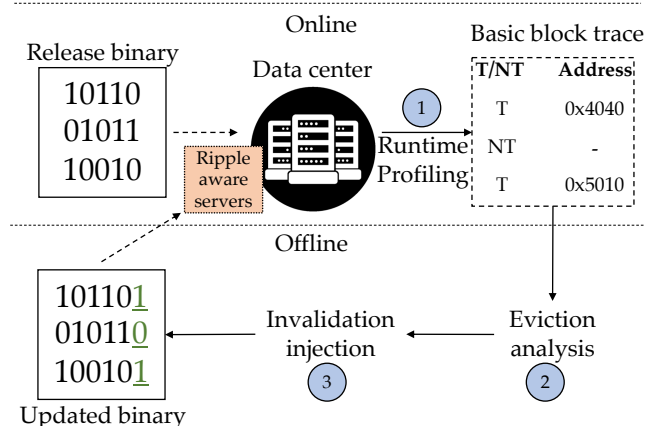

Fig. 4: High-level design of *Ripple*

software, thereby reducing hardware overheads. We now detail *Ripple*'s components.

A. Runtime Profiling

Ripple profiles data center applications at run time using Intel PT [58] to collect a trace of the dynamically-executed basic blocks. As shown in Fig. 4, the collected program trace includes two pieces of information for each control-flow instruction in the program. The first bit (T/NT), denotes whether the branch in question was taken (T) or the fall-through path was followed (NT). If the program follows the taken path of an indirect branch, the program trace also includes the address of the next instruction on the taken path. *Ripple* leverages this program execution trace, to perform (1) eviction analysis and (2) invalidation injection offline at link-time. During eviction analysis, *Ripple* identifies the I-cache lines that will be touched (omitting speculative accesses) in real hardware based on the execution trace. *Ripple*'s eviction analysis does not require recording the I-cache lines that will be evicted in hardware.

Ripple leverages Intel PT [58] to collect the precise basic block execution order with low runtime performance overhead (less than 1% [48, 109]). *Ripple* uses Intel PT since it is efficient in real-world production scenarios [19, 28, 49, 50].
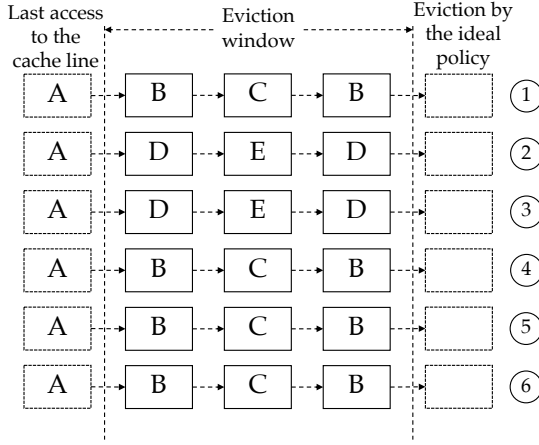
(a) Cache line *A*'s eviction window includes all basic blocks executed since *A*'s last execution until *A*'s eviction by the ideal cache replacement policy.

Basic block	Total executed	# of eviction windows where basic block is executed at least once	P(Eviction | Basic block)
B	16	4	0.25
C	8	4	0.5
D	6	2	0.33
E	3	2	0.66

(b) How *Ripple* calculates the conditional probability of the eviction of the cache line *A*, given the execution of a particular basic block.

Fig. 5: An example of *Ripple*'s eviction analysis process

B. Eviction Analysis

The goal of *Ripple*'s eviction analysis is to mimic an ideal replacement policy, which would evict cache lines that will not be accessed for the longest time in the future. The basic block trace collected at run time allows *Ripple* to retroactively determine points in the execution that would benefit from invalidating certain cache lines to help with cache replacement.

Eviction analysis determines a *cue block*, whose execution can identify the eviction of a particular *victim* cache line with high probability, if an ideal cache replacement policy were used. To determine the cue block in the collected runtime profile, *Ripple* analyzes all the blocks in the *eviction window* of each cache line, *i.e.*, the time window spanning between the last access to that cache line to the access that would trigger the eviction of the same line, given an ideal replacement policy.

Fig. 5a shows examples of eviction windows for the cache line, *A*. In this example, the cache line *A* gets evicted six times by the ideal cache replacement policy over the execution of the program. To compute each eviction window, *Ripple* iterates backward in the basic block trace from each point where *A* would be evicted by an ideal replacement policy until it reaches a basic block containing (even partially) the cache line *A*. *Ripple* identifies the basic blocks across all eviction windows that can accurately signal the eviction as *candidate cue blocks* (described further in Sec. III-C), where it can insert an invalidation instruction to mimic the ideal cache replacement

behavior. In this example, *Ripple* identifies basic blocks, *B*, *C*, *D*, and *E* as candidate cue blocks.

Next, *Ripple* calculates the conditional probability of a cache line eviction given the execution of each candidate cue block. Fig. 5b shows an example of this probability calculation for the cache line, *A*. To calculate this conditional probability, *Ripple* calculates two metrics. First, it computes how many times each candidate cue block was executed during the application's lifetime. In this example, the candidate cue blocks *B*, *C*, *D*, and *E* are executed 16, 8, 6, and 3 times respectively. Second, for each candidate cue block, *Ripple* computes the number of unique eviction windows which include the corresponding candidate cue block. In our example, basic blocks *B*, *C*, *D*, and *E* are included in 4, 4, 2, and 2 unique eviction windows, respectively. *Ripple* calculates the conditional probability as the ratio of the second value (count of windows containing the candidate cue block) to the first value (execution count of the cue block). For instance, $P((Eviction, A)|(Execute, B)) = 0.25$ denotes that for each execution of *B*, there is a 25% chance that the cache line *A* may be evicted.

Finally, for each eviction window, *Ripple* selects the cue block with the highest conditional probability, breaking ties arbitrarily. In our example, *Ripple* will select basic blocks *C* and *E* as cue blocks for 4 (windows 1, 4, 5, 6) and 2 (windows 2, 3) eviction windows, respectively. If the conditional probability of the selected basic block is larger than a threshold, *Ripple* will inject an explicit invalidation request in the basic block during recompilation. Next, we describe the process by which the invalidation instructions are injected as well as the trade-off that is associated with this probability threshold.

C. Injection of Invalidation Instructions

Based on the eviction analysis, *Ripple* selects the cue basic block for each eviction window. Next, *Ripple* inserts an explicit invalidation instruction into the cue block to invalidate the victim cache line. *Ripple*'s decision to insert an invalidation instruction is informed by the conditional probability it computes for each candidate cue block. Specifically, *Ripple* inserts an invalidation instruction into the cue block only if the conditional probability is higher than the *invalidation threshold*. We now describe how *Ripple* determines the invalidation threshold and the invalidation granularity (*i.e.*, why *Ripple* decides to inject invalidation instructions in a basic block to evict a cache line). We then give details on the invalidation instruction that *Ripple* relies on.

Determining the invalidation threshold. *Ripple* considers two key metrics when selecting the value of the invalidation threshold: *replacement coverage* and *replacement accuracy*. We first define these metrics and then explain the trade-off between them.

Replacement-Coverage. We define *replacement-coverage* as the ratio of the total number of replacement decisions performed by a given policy divided by the total number of replacement decisions performed by the ideal replacement policy. A policy that exhibits less than 100% replacement-coverage omits some

invalidation candidates that the optimal replacement policy would have chosen for eviction.

Replacement-Accuracy. We define *replacement-accuracy* as the ratio of total optimal replacement decisions of a given policy divided by the replacement decisions performed by the ideal replacement policy. Therefore, if *Ripple* induces x invalidations over a program's lifetime, and y of those invalidations do not introduce any new misses over the ideal cache replacement policy, then *Ripple*'s accuracy (in percentage) is: $\frac{100*y}{x}$. A policy that exhibits less than 100% replacement-accuracy will evict cache lines that the ideal cache replacement policy would not have evicted.

Coverage-Accuracy Trade-off. Replacement-coverage and replacement-accuracy represent useful metrics to measure a cache replacement policy's optimality. A software-guided policy with a low replacement-coverage will frequently need to revert to the underlying hardware policy suffering from its sub-optimal decisions. On the other hand, a policy with low replacement-accuracy will frequently evict lines that the program could still use. As shown in Fig. 6, *Ripple* leverages the invalidation-threshold to control the aggressiveness of its evictions, allowing to trade-off coverage and accuracy. Although this figure presents data from a single application (*i.e.*, `finagle-http`), we observe similar trends across all the data center applications that we evaluate.

At a lower threshold (0-20%), *Ripple* has almost 100% coverage, because all the replacement decisions are made by *Ripple*'s invalidations. At the same time, *Ripple*'s accuracy suffers greatly because it invalidates many cache lines that introduce new misses over the ideal cache replacement policy. Consequently, at a lower threshold, *Ripple* does not provide additional performance over the underlying replacement policy.

Similarly, at a higher threshold (80-100%), *Ripple* achieves near-perfect accuracy as cache lines invalidated by *Ripple* do not incur extra misses over the ideal replacement policy. However, *Ripple*'s coverage drops sharply as more replacement decisions are not served by *Ripple*-inserted invalidations. Therefore, *Ripple*'s performance benefit over the underlying hardware replacement policy declines rapidly.

Only at the middle ground, *i.e.*, when the invalidation threshold ranges from 40-60%, *Ripple* simultaneously achieves both high coverage (greater than 50%) and high accuracy (greater than 80%). As a result, *Ripple* provides the highest performance benefit at this invalidation threshold range. For each application, *Ripple* chooses the invalidation threshold that provides the best performance for a given application. Across 9 applications, this invalidation threshold varies from 45-65%.

Invalidation granularity. *Ripple* injects invalidation instructions at the basic block granularity while invalidation instructions evict cache lines. In practice, we find that *Ripple* does not suffer a performance loss due to this mismatch. In particular, *Ripple* provides a higher speedup when evicting at the basic block granularity than when evicting at the cache line or combination of basic block and cache line granularity.

The Invalidation instruction. We propose a new invalidation instruction, `invalidate` that takes the address of a cache line
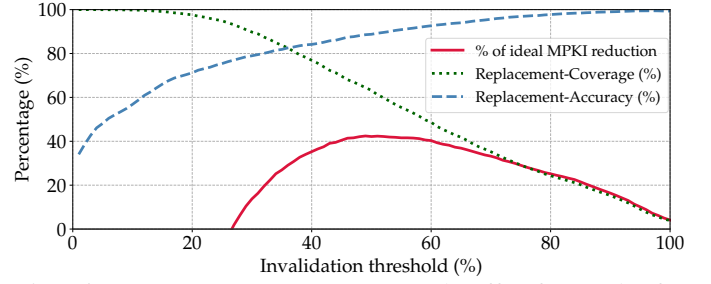

Fig. 6: Coverage vs. accuracy trade-off of *Ripple* for `finagle-http`. Other applications also exhibit a similar trade-off curve. The invalidation threshold providing the best performance across the 9 data center applications we studied varies between 45-65%.

as an operand and invalidates it if the cache line resides in the I-cache. Our proposed `invalidate` instruction exhibits one key difference compared to existing cache line flushing instructions (*e.g.*, the `clflush` instruction on Intel processors) in that it does not invalidate the cache line from other caches in the cache hierarchy. Instead, our proposed `invalidate` instruction invalidates the cache line only in the local I-cache, thereby avoiding costly cache-coherency transactions and unnecessary invalidations in remote caches. Furthermore, the `invalidate` instruction has low latency as it does not have to wait for the potentially dirty cache line to be written back to the lower cache levels. Instead, `invalidate` can be regarded as a hint that can be freely reordered with fences and synchronization instructions. Intel recently introduced [98] such an invalidation instruction called (`cldemote`) slated to be supported in its future servers, and hence *Ripple* will be readily implementable on such upcoming processors.

IV. EVALUATION

In this section, we first describe our experimental methodology and then evaluate *Ripple* using key performance metrics.

Trace collection. We collect the execution trace of data center applications using Intel Processor Trace (PT). Specifically, we record traces for 100 million instructions in the application's steady-state containing both user and kernel mode instructions as Intel PT allows to capture both. We find that for most applications, the percentage of kernel mode instruction induced I-cache misses is small ($< 1\%$). However, for `drupal`, `mediawiki`, and `wordpress`, kernel code is responsible for 15% of all I-cache misses.

Simulation. At the time of this writing, no commercially-available processor supports our proposed `invalidate` instruction, even though future Intel processors will support the functionally-equivalent `cldemote` instruction [101]. To simulate this invalidate instruction, we evaluate *Ripple* using simulation. This also allows us to evaluate additional replacement policies and their interactions with *Ripple*. We extend the ZSim simulator [86] by implementing our proposed `invalidate` instruction. We list several important parameters of the trace-

TABLE II: Simulator Parameters

Parameter	Value
CPU	Intel Xeon Haswell
Number of cores per socket	20
L1 instruction cache	32 KiB, 8-way
L1 data cache	32 KiB, 8-way
L2 unified cache	1 MB, 16-way
L3 unified cache	Shared 10 MiB per socket, 20-way
All-core turbo frequency	2.5 GHz
L1 I-cache latency	3 cycles
L1 D-cache latency	4 cycles
L2 cache latency	12 cycles
L3 cache latency	36 cycles
Memory latency	260 cycles
Memory bandwidth	6.25 GB/s

driven out-of-order ZSim simulation in Table II. We implement *Ripple* on the L1 I-cache in our experiments.

Data center applications and inputs. We use nine widely-used data center applications described in §II to evaluate *Ripple*. We study these applications with different input parameters offered to the client's load generator (*e.g.*, number of requests per second or the number of threads). We evaluate *Ripple* using different inputs for training (profile collection) and evaluation.

We now evaluate *Ripple* using key performance metrics on all nine data center applications described in Sec. II. First, we measure how much speedup *Ripple* provides compared to ideal and other prior cache replacement policies. Next, we compare L1 I-cache MPKI reduction (%) for *Ripple*, ideal, and other policies for different prefetching configurations. Then, we evaluate *Ripple*'s replacement-coverage and replacement-accuracy as described in Sec. III-C. Next, we measure how much extra static and dynamic instructions *Ripple* introduces into the application binary. Finally, we evaluate how *Ripple* performs across multiple application inputs.

Speedup. We measure the speedup (*i.e.*, percentage improvement in instructions per cycle [IPC]) provided by *Ripple* over an LRU baseline. We also compare *Ripple*'s speedup to speedups provided by the prefetch-aware ideal replacement policy as well as four additional prior cache replacement policies including Hawkeye/Harmony, DRRIP, SRRIP, and GHRP, whose details were discussed in §II. To show that *Ripple*'s speedup is not primarily due to the underlying hardware replacement policy, we also provide *Ripple*'s speedup with two different underlying hardware replacement policies (random and LRU). Finally, we measure the speedups for all replacement policies by altering the underlying I-cache prefetching mechanisms (no prefetching, NLP, and FDIP).

Fig. 7 shows the speedup results. *Ripple*, with an underlying LRU-based hardware replacement policy (*i.e.*, *Ripple*-LRU in Fig. 7), always outperforms all prior replacement policies across all different prefetcher configurations. In particular, *Ripple*-LRU provides on average 1.25% (no prefetching), 2.13% (NLP), 1.4% (FDIP) speedups over a pure-LRU replacement policy baseline. These speedups correspond to 37% (no prefetching), 55% (NLP), and 44% (FDIP) of the speedups of an ideal cache replacement policy. Notably, even *Ripple*-Random, which operates with an underlying random hardware replacement

policy (which itself is on average 1% slower than LRU), provides 0.86% average speedup over the LRU baseline across the three different I-cache prefetchers. In combination with *Ripple*, Random becomes a feasible replacement policy that eliminates all meta-data storage overheads in hardware.

The performance gap between *Ripple* and the ideal cache replacement policy stems from two primary reasons. First, *Ripple* cannot cover all eviction windows via software invalidation as covering all eviction windows requires *Ripple* to sacrifice eviction accuracy which hurts performance. Second, software invalidation instructions inserted by *Ripple* introduce static and dynamic code bloat, causing additional cache pressure that contributes to the performance gap (we quantify this overhead later in this section).

I-cache MPKI reduction. Fig. 8 shows the L1 I-cache miss reduction provided by *Ripple* (with underlying hardware replacement policies of LRU and Random) and the prior work policies. As shown, *Ripple*-LRU reduces I-cache misses over all prior policies across all applications. Across different prefetching configurations, *Ripple* can avoid 33% (no prefetching), 53% (NLP), and 41% (FDIP) of I-cache misses that are avoided by the ideal replacement policy. *Ripple* reduces I-cache MPKI regardless of the underlying replacement policy. Even when the underlying replacement policy is random (causing 12.71% more misses in average than LRU), *Ripple*-Random incurs 9.5% fewer misses on average than LRU for different applications and prefetching configurations.

Replacement-Coverage. As described in §III-C, *Ripple*'s coverage is the percentage of all replacement decisions (over the program life time) that were initiated by *Ripple*'s invalidations. Fig. 9 shows *Ripple*'s coverage for all applications. As shown, *Ripple* achieves on average more than 50% coverage. Only for three HHVM applications (*i.e.*, drupal, mediawiki, and wordpress), *Ripple*'s coverage is lower than 50%, as for these applications *Ripple* does not insert invalidate instructions on the just-in-time compiled basic blocks. Just-in-time (Jit) compiled code may reuse the same instruction addresses for different basic blocks over the course of an execution rendering compile-time instruction injection techniques challenging. Nevertheless, even for these Jit applications there remains enough static code that *Ripple* is able to optimize.

Accuracy. In Fig. 10, we show *Ripple*'s replacement-accuracy (as defined in §III-C). As shown, *Ripple* achieves 92% accuracy on average (with a minimum improvement of 88%). *Ripple*'s accuracy is on average 14% higher than LRU's average accuracy (77.8%). Thanks to its higher accuracy, *Ripple* avoids many inaccurate replacement decisions due to the underlying LRU-based hardware replacement policy (which has an average accuracy of 77.8%), and, therefore, the overall replacement accuracy for Ripple-LRU is on average 86% (8.2% higher than the LRU baseline).

Instruction overhead. Fig. 11 and 12 quantify the static and dynamic code footprint increase introduced by the injected invalidate instructions. The static instruction overhead of *Ripple* is less than 4.4% for all cases while the dynamic instruction overhead is less than 2% in most cases, except for verilator.
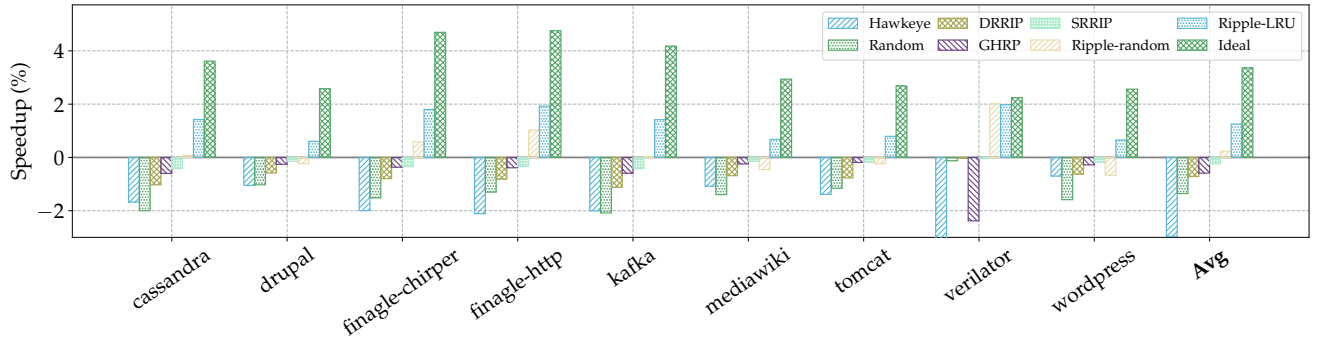
(a) Over no prefetching baseline, *Ripple* provides 1.25% speedup compared to 3.36% ideal speedup on average.
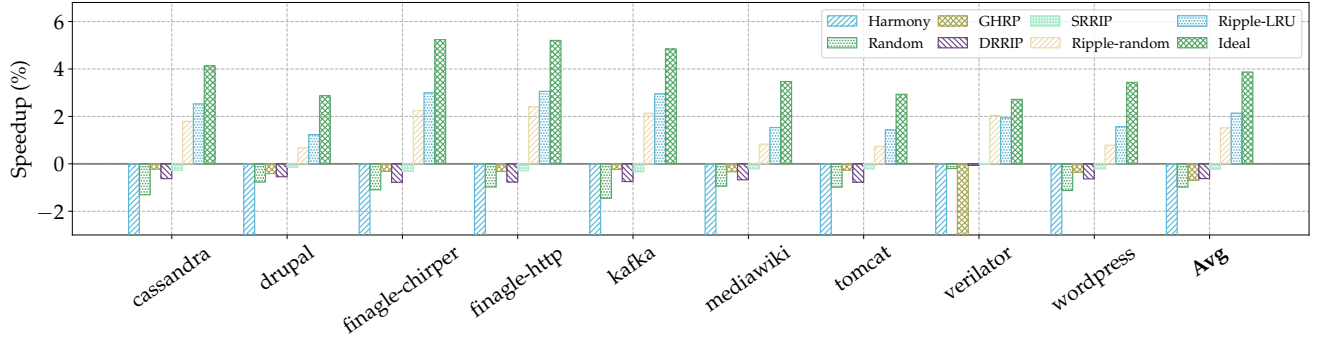
(b) Over next-line prefetching baseline, *Ripple* provides 2.13% speedup compared to 3.87% ideal speedup on average.
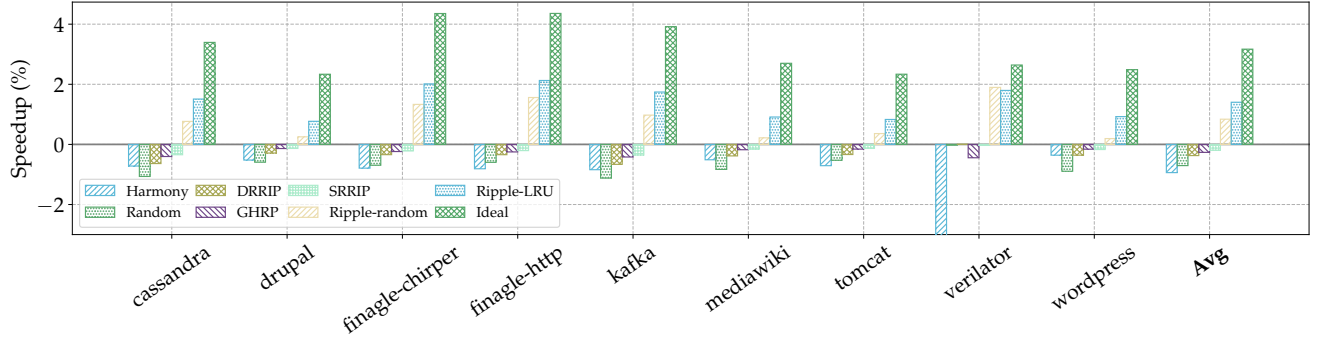
(c) Over fetch directed instruction prefetching baseline, *Ripple* provides 1.4% speedup compared to 3.16% ideal speedup on average.

Fig. 7: *Ripple*'s speedup compared to ideal and state-of-the-art replacement policies over an LRU baseline (with different hardware prefetching): On average, *Ripple* provides 1.6% speedup compared to 3.47% ideal speedup.

For this application, *Ripple* executes 10% extra instructions to invalidate cache lines. This is because for `verilator`, *Ripple* covers almost all replacement policy decisions via software invalidation (98.7% coverage as shown in Fig. 9). Similarly, *Ripple*'s accuracy for `verilator` is very high (99.9% as shown in Fig. 10). Therefore, though *Ripple* executes a relatively greater number of invalidation instructions for `verilator`, it does not execute unnecessary invalidation instructions.

Profiling and offline analysis overhead. *Ripple* leverages Intel PT to collect basic block traces from data center application executions because of its low overhead (less than 1%) and adoption in production settings [19, 28]. While *Ripple*'s extraction and analysis on this trace takes longer (up to 10 minutes), we do not expect that this expensive analysis will be deployed in production servers. Instead, we anticipate the

extraction, analysis, and invalidation injection component of *Ripple* will be performed offline, similar to how existing profile-guided optimizations for data center applications are performed [17, 30, 54, 75, 76]. Therefore, we consider the overhead for *Ripple*'s offline analysis acceptable.

Invalidation vs. reducing LRU priority. When the underlying hardware cache replacement policy is LRU, moving a cache line to the bottom of the LRU chain is sufficient to cause eviction. This LRU-specific optimization improved Ripple's IPC speedup from 1.6% to 1.7% (apart from `verilator`, all other applications benefited from this optimization). This shows that Ripple's profiling mechanism works well independent of the particular eviction mechanism.

Performance across multiple application inputs. We investigate *Ripple*'s performance for data center applications with
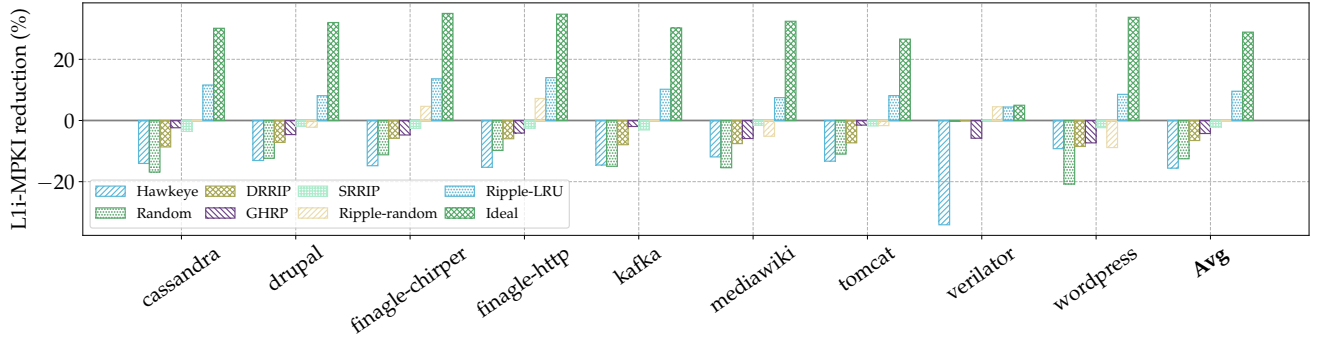
(a) With no prefetching, *Ripple*-LRU reduces 9.57% of all I-cache misses compared to 28.88% miss reduction provided by the ideal replacement policy.
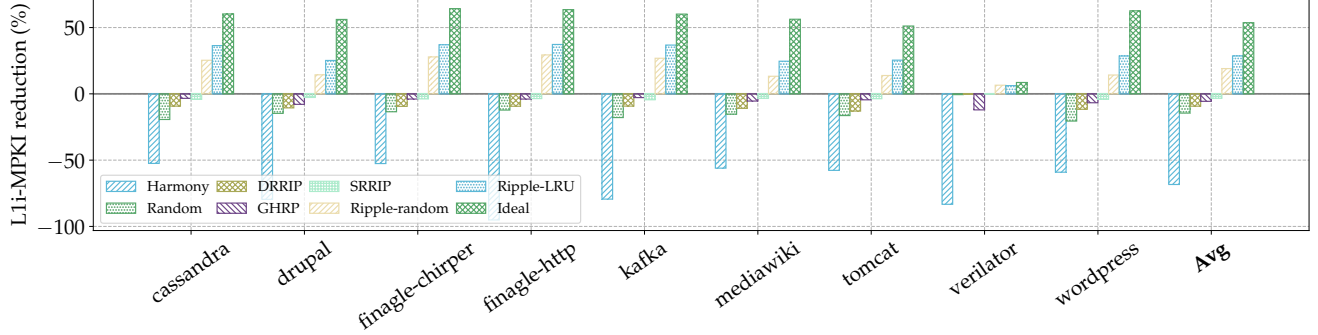
(b) With next-line prefetching, *Ripple*-LRU reduces on average 28.6% of all I-cache misses compared to 53.66% reduction provided by the ideal replacement policy.
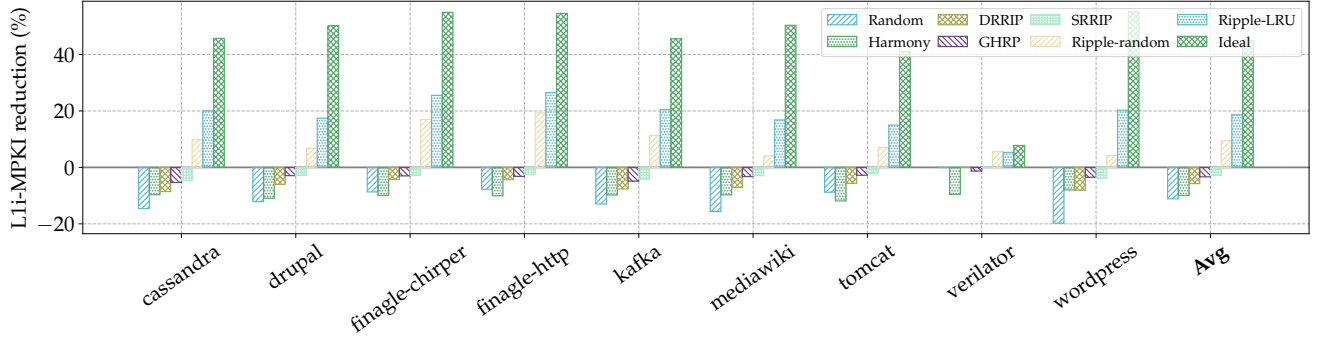
(c) With fetch directed instruction prefetching, *Ripple*-LRU reduces on average 18.61% of all I-cache misses compared to 45% reduction provided by the ideal replacement policy.

Fig. 8: *Ripple*'s L1 I-cache miss reduction compared to ideal and state-of-the-art replacement policies over an LRU baseline (with different hardware prefetching): On average, *Ripple* reduces 19% of all LRU I-cache misses compared to 42.5% miss reduction by the ideal replacement policy.
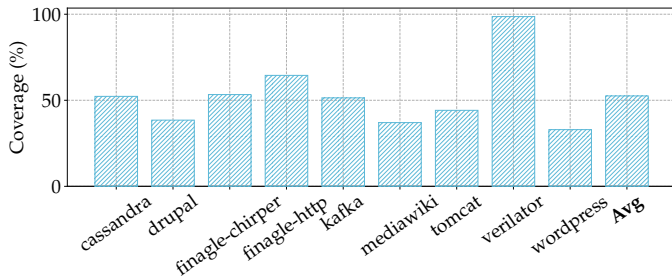

Fig. 9: *Ripple*'s coverage for different applications: On average 50% of replacement requests are processed by evicting cache lines that *Ripple* invalidates.

three separate input configurations ('#1' to '#3'). We vary these applications' input configurations by changing the webpage, the client requests, the number of client requests per second, the number of server threads, random number seeds, and the size of input data. We optimize each application using the profile from input '#0' and measure *Ripple*'s performance benefits for different test inputs '#1, #2, #3'. For each input, we also measure the performance improvement when *Ripple* optimizes the application with a profile for the same input. As shown in Fig. 13, *Ripple* provides 17% more IPC gains with input-specific profiles compared to profiles that are not input specific.
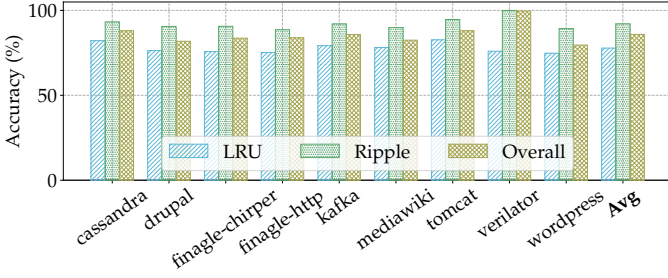
Fig. 10: *Ripple*'s accuracy for different applications: On average *Ripple* provides 92% accuracy which ensures that the overall accuracy is 86% even though underlying LRU has an accuracy of 77.8%.

Fig. 12: Dynamic instruction overhead introduced by *Ripple*: On average *Ripple* executes 2.2% extra dynamic instructions.
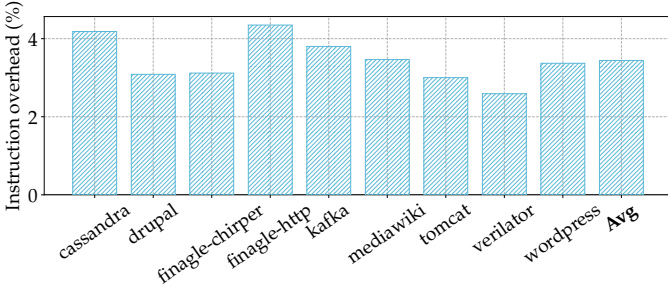

Fig. 11: Static instruction overhead introduced by *Ripple*: On average *Ripple* inserts 3.4% new static instructions.

For brevity, we only show the results for the FDIP baseline. Results with the other prefetching baselines are similar.

V. DISCUSSION

Ripple generates optimized binaries for different target architectures considering the processor's I-cache size and associativity. Such a process is common in data centers deploying profile-guided [17, 62] and post-link-time-based optimization [75, 76] techniques. Therefore, *Ripple* can be conveniently integrated into the existing build and optimization processes. Moreover, as the I-cache size (32KB) and associativity (8-way) for Intel data center processors has been stable for the last 10 years, the number of different target architectures that *Ripple* needs to support is small.

VI. RELATED WORK

Instruction prefetching. Hardware instruction prefetchers such as next-line and decoupled fetch directed prefetchers [18, 39, 82, 84, 92] have been pervasively deployed in commercial designs [32, 78, 85, 97]. While complex techniques [25, 26, 51, 52] employing record and replay prefetchers are highly effective in reducing I-cache misses, they require impractical on-chip metadata storage. Branch predictor-guided prefetchers [9, 60, 61], on the other hand, follow the same principle as FDIP to reduce on-chip metadata storage, however, they also require a complete overhaul of the underlying branch target prediction unit. Even recent proposals [8, 29, 31, 33, 70, 72, 83, 89] from 1st Instruction Prefetching Championship (IPC1)
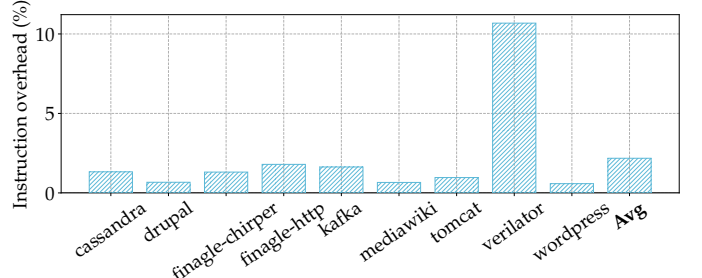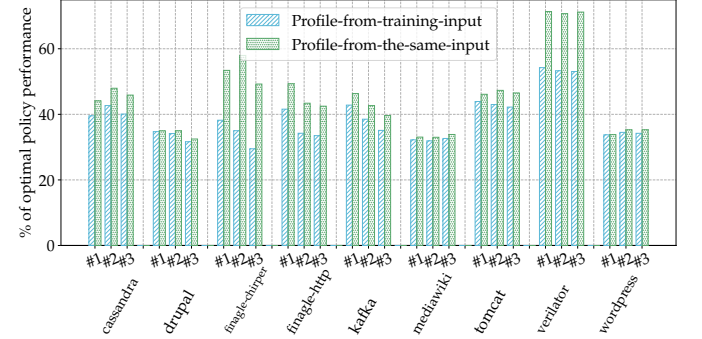

Fig. 13: *Ripple*'s performance for multiple application inputs with the FDIP baseline: On average *Ripple* provides 17% more speedup with input-specific profiles compared to profiles from different inputs.

require kilobytes of extra on-chip storage to provide near-ideal performance even on workloads where FDIP with a large enough fetch target queue provides most of the potential performance benefit [38]. Hybrid hardware-software prefetchers [12, 13, 55, 68, 71] analyze a program's control flow information in software and inject dedicated prefetching instructions in code which does not exist in today's hardware. In contrast, we show that instruction prefetchers alone do not close the performance gap due to wasteful evictions that must be handled by smarter cache line replacement.

Cache replacement policies. Heuristic-based hardware data cache replacement policies have been studied for a long time, including LRU and its variations [47, 63, 73, 91, 106], MRU [80], re-reference interval prediction [43], reuse prediction [22, 23, 66] and others [6, 27, 35, 56, 81, 87, 96, 99]. Learning-based data cache replacement policies [40, 41, 53, 107] consider replacement as a binary classification problem of cache-friendly or cache-averse. Recent methods introduce machine learning techniques like perceptrons [45, 100] and genetic algorithms [44]. Some learning-based policies use information of Belady's optimal solution [15], including Hawkeye [40], Glider [90] and Parrot [65]. However, these policies are mostly designed for data caches and do not work well for instruction caches as we show earlier (Sec. II). We also propose a profile-guided approach that can work on top of any of these policies.

Prefetch-aware replacement policy. Prefetch-aware replacement policies focus on avoiding cache pollution caused by inaccurate prefetches. Some prefetch-aware policies [36, 37, 57] get feedback from prefetchers to identify inaccurate prefetches, and need co-design or prefetcher modifications. Others [88, 94, 108] work independently from the prefetcher and estimate prefetch accuracy from cache behavior.

With prefetching, Belady's optimal policy [15] becomes incomplete as it cannot distinguish easy-to-prefetch cache lines from hard-to-prefetch cache lines [94, 108]. To address this limitation, Demand-MIN [41] revised Belady's optimal policy to accommodate prefetching and proposed a program counter (PC) classification based predictor, Harmony to emulate the ideal performance. In this work, we not only revise Demand-MIN to cover an extra corner case, but also show that a PC-classification based predictor performs poorly for I-cache. We address this imprecision and effectively emulate optimal I-cache behavior in our work via a profile-guided software technique.

VII. CONCLUSION

Modern data center applications have large instruction footprints, leading to significant I-cache misses. Although numerous prior proposals aim to mitigate I-cache misses, they still fall short of an ideal cache. We investigated why existing I-cache miss mitigation mechanisms achieve sub-optimal speedup, and found that widely-studied instruction prefetchers incur wasteful prefetch-induced evictions that existing replacement policies do not mitigate. To enable smarter evictions, we proposed *Ripple*, a novel profile-guided replacement technique that uses program context to inform the underlying replacement policy about efficient replacement decisions. *Ripple* identifies program contexts that lead to I-cache misses and sparingly injects "cache line eviction" instructions in suitable program locations at link time. We evaluated *Ripple* using nine popular data center applications and demonstrated that it is replacement policy agnostic, *i.e.*, it enables any replacement policy to achieve speedup that is 44% closer to that of an ideal I-cache.

ACKNOWLEDGMENTS

REFERENCES

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

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

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

[4] "Verilator," https://www.veripool.org/wiki/verilator.

[5] "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).

[6] 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.

[7] S. M. Ajorpaz, E. Garza, S. Jindal, and D. A. Jiménez, "Exploring predictive replacement policies for instruction cache and branch target buffer," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2018, pp. 519–532.

[8] A. Ansari, F. Golshan, P. Lotfi-Kamran, and H. Sarbazi-Azad, "Mana: Microarchitecting an instruction prefetcher," *The First Instruction Prefetching Championship*, 2020.

[9] A. Ansari, P. Lotfi-Kamran, and H. Sarbazi-Azad, "Divide and conquer frontend bottleneck," in *Proceedings of the 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020.

[10] K. Asanovic, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, D. Dabbelt, J. Hauser, A. Izraelevitz *et al.*, "The rocket chip generator," *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17*, 2016.

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

[12] G. Ayers, H. Litz, C. Kozyrakis, and P. Ranganathan, "Classifying memory access patterns for prefetching," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 513–526.

[13] 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.

[14] S. Bansal and D. S. Modha, "Car: Clock with adaptive replacement." in *FAST*, vol. 4, 2004, pp. 187–200.

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

[16] 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.

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

[18] T.-F. Chen and J.-L. Baer, "Reducing memory latency via non-blocking and prefetching caches," *ACM SIGPLAN Notices*, vol. 27, no. 9, pp. 51–61, 1992.

[19] 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.

[20] P. J. Denning, "Thrashing: Its causes and prevention," in *Proceedings of the December 9-11, 1968, fall joint computer conference, part I*, 1968, pp. 915–922.

[21] J. Doweck, W.-F. Kao, A. K.-y. Lu, J. Mandelblat, A. Rahatekar, L. Rappoport, E. Rotem, A. Yasin, and A. Yoaz, "Inside 6th-generation intel core: New microarchitecture code-named skylake," *IEEE Micro*, 2017.

[22] 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.

[23] 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.

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

[25] M. Ferdman, C. Kaynak, and B. Falsafi, "Proactive instruction fetch," in *International Symposium on Microarchitecture*, 2011.

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

[27] H. Gao and C. Wilkerson, "A dueling segmented lru replacement algorithm with adaptive bypassing," 2010.

[28] X. Ge, B. Niu, and W. Cui, "Reverse debugging of kernel failures in deployed systems," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, Jul. 2020, pp. 281–292. [Online]. Available: https://www.usenix.org/conference/atc20/presentation/ge

[29] N. Gober, G. Chacon, D. Jiménez, and P. V. Gratz, "The temporal ancestry prefetcher."

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

[31] D. A. J. P. V. Gratz and G. C. N. Gober, "Barca: Branch agnostic region searching algorithm."

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

[33] V. Gupta, N. S. Kalani, and B. Panda, "Run-jump-run: Bouquet of instruction pointer jumpers for high performance instruction prefetching."

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

[35] 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.

[36] Y. Ishii, M. Inaba, and K. Hiraki, "Access map pattern matching for high performance data cache prefetch," *Journal of Instruction-Level Parallelism*, vol. 13, no. 2011, pp. 1–24, 2011.

[37] Y. Ishii, M. Inaba, and K. Hiraki, "Unified memory optimizing architecture: memory subsystem control with a unified predictor," in *Proceedings of the 26th ACM international conference on Supercomputing*, 2012, pp. 267–278.

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

[39] Q. Jacobson, E. Rotenberg, and J. E. Smith, "Path-based next trace prediction," in *Proceedings of 30th Annual International Symposium on Microarchitecture*. IEEE, 1997, pp. 14–23.

[40] 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.

[41] 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.

[42] A. Jain and C. Lin, "Cache replacement policies," *Synthesis Lectures on Computer Architecture*, vol. 14, no. 1, pp. 1–87, 2019.

[43] 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.

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

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

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

[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," in *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*. Kartause Ittingen, Switzerland: USENIX Association, May 2015.

[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 *SOSP*, Monterey, CA, October 2015.

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

[52] 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.

[53] 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.

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

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

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

[57] J. Kim, E. Teran, P. V. Gratz, D. A. Jiménez, S. H. Pugsley, and C. Wilkerson, "Kill the program counter: Reconstructing program behavior in the processor cache hierarchy," *ACM SIGPLAN Notices*, vol. 52, no. 4, pp. 737–749, 2017.

[58] A. Kleen and B. Strong, "Intel processor trace on linux," *Tracing Summit*, 2015.

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

[60] R. Kumar, B. Grot, and V. Nagarajan, "Blasting through the front-end bottleneck with shotgun," *Proceedings of the 23rd ASPLOS*, 2018.

[61] 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.

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

[63] 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.

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

[65] 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.

[66] 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.

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

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

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

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

[71] N. P. Nagendra, G. Ayers, 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," *IEEE Micro*, vol. 40, no. 3, pp. 56–63, 2020.

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

[73] 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.

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

[75] 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.

[76] 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.

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

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

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

[80] 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.

[81] 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.

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

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

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

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

[86] D. Sanchez and C. Kozyrakis, "Zsim: Fast and accurate microarchitectural simulation of thousand-core systems," in *International Symposium on Computer Architecture*, 2013.

[87] 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.

[88] V. Seshadri, S. Yedkar, H. Xin, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "Mitigating prefetcher-caused pollution using

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

[90] 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.

[91] 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.

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

[93] A. J. Smith, "Cache memories," *ACM Computing Surveys (CSUR)*, vol. 14, no. 3, pp. 473–530, 1982.

[94] S. Srinath, O. Mutlu, H. Kim, and Y. N. Patt, "Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers," in *2007 IEEE 13th International Symposium on High Performance Computer Architecture*. IEEE, 2007, pp. 63–74.

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

[96] 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.

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

[98] V. Sukhomlinov and K. Doshi, "Selective execution of cache line flush operations," Oct. 8 2020, uS Patent App. 16/907,729.

[99] 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.

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

[101] Wikipedia contributors, "Alder lake (microprocessor) — Wikipedia, the free encyclopedia," https://en.wikipedia.org/w/index.php?title=Alder_Lake_(microprocessor)&oldid=990207738, 2020, [Online; accessed 25-November-2020].

[102] Wikipedia contributors, "Apache kafka — Wikipedia, the free encyclopedia," https://en.wikipedia.org/w/index.php?title=Apache_Kafka&oldid=988898935, 2020, [Online; accessed 23-November-2020].

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

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

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

[106] W. A. Wong and J.-L. Baer, "Modified lru policies for improving second-level cache behavior," in *Proceedings Sixth International Symposium on High-Performance Computer Architecture. HPCA-6 (Cat. No. PR00550)*. IEEE, 2000, pp. 49–60.

[107] 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.

[108] C.-J. Wu, A. Jaleel, M. Martonosi, S. C. Steely Jr, and J. Emer, "Pacman: prefetch-aware cache management for high performance caching," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, 2011, pp. 442–453.

[109] G. Zuo, J. Ma, A. Quinn, P. Bhatotia, P. Fonseca, and B. Kasikci, "Execution reconstruction: Harnessing failure reoccurrences for failure reproduction," in *ACM SIGPLAN conference on Programming language design and implementation*, 2021.

informed caching policies for prefetched blocks," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 11, no. 4, pp. 1–22, 2015.